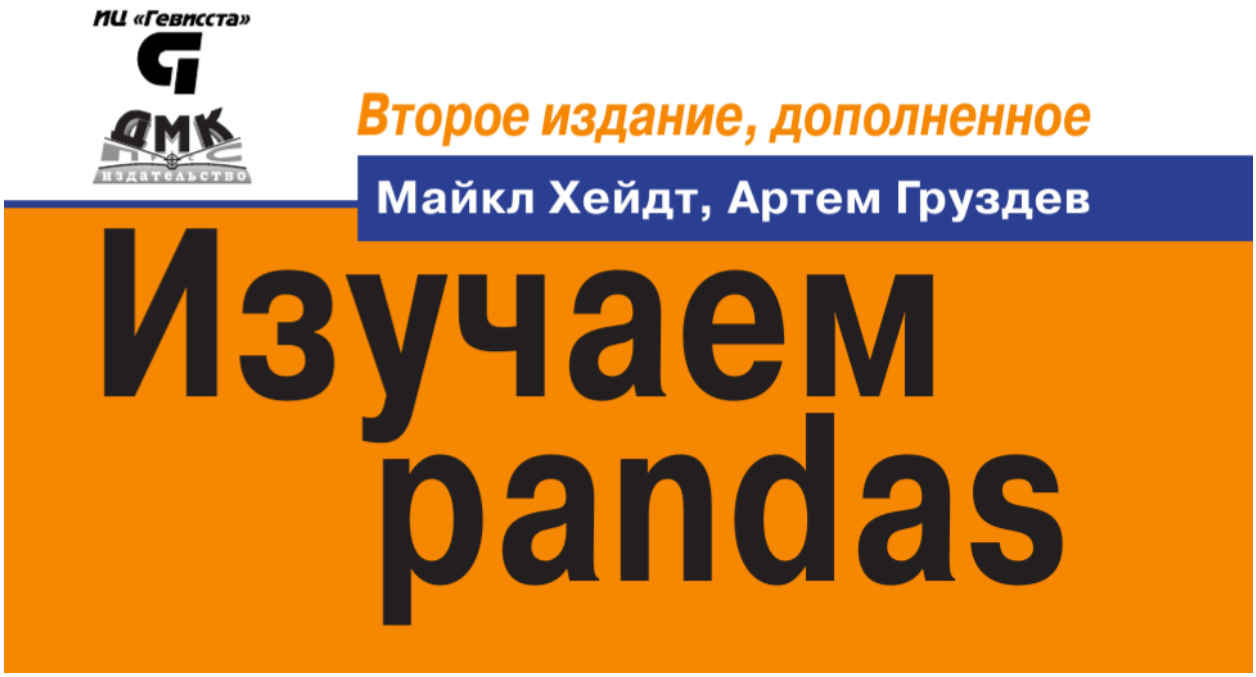


Полная авторская версия от 14.05.2019

Книга рекомендована в качестве учебного пособия
к курсу «Машинное обучение в R, Python и H2O»

https://www.facebook.com/groups/gewissta

СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ

Pandas – популярная библиотека Python, применяющаяся для практического анализа данных в реальных проектах. Она предлагает воспользоваться эффективными, быстрыми и высокопроизводительными структурами данных, которые упрощают предварительную обработку и анализ информации. Это учебное пособие существенно поможет вам, предоставив в ваше распоряжение внушительный набор инструментов, предлагаемых библиотекой pandas для выполнения различных операций с данными и их анализа.

О содержании книги

Глава 1 «Библиотека pandas и анализ данных» – это практическое введение в основные функции библиотеки pandas. Предназначение этой главы – дать некоторое представление об использовании библиотеке pandas в контексте статистики и науки о данных. В этой главе мы рассмотрим несколько принципов, лежащих в основе науки о данных и покажем, как они реализованы в библиотеке pandas. Эта глава задает контекст для каждой последующей главы, связанной с наукой о данных.
Глава 2 «Запуск библиотеки pandas» проинструктирует читателя по поводу того, как загрузить и установить библиотеку pandas и познакомит его с некоторыми базовыми понятиями библиотеки pandas. Мы также рассмотрим, как можно работать с примерами с помощью iPython и тетрадок Jupiter.
Глава 3 «Представление одномерных данных с помощью объекта Series» познакомит читателя со структурой данных Series, которая используется для представления одномерных индексированных данных. Читатель узнает о том, как создавать объекты Series и как работать с данными, хранящимися внутри этих объектах. Кроме того, он узнает об индексах и выравнивании данных, а также о том, как объект Series можно использовать для создание срезов данных.
Глава 4 «Представление табличных и многомерных данных с помощью объекта DataFrame» познакомит читателя со структурой данных DataFrame, которая используется для представления и индексации многомерных данных. В этой главе читатель научится создавать объекты DataFrame, используя различные наборов статических данных и выполнять отбор определенных столбцов и строк внутри датафрейма. Сложные запросы, операции с данными и индексация будут рассмотрены в следующей главе.
Глава 5 «Выполнение операций над объектом DataFrame и его содержимым» расширяет предыдущую главу и расскажет о том, как выполнять более сложные операции с объектом DataFrame. Мы начнем с добавления удаления столбцов и строк, рассмотрим модификацию данных внутри объекта DataFrame (а также создание измененной копии) и выполнение арифметических операций с данными, научимся создавать

иерархические индексы, а также вычислять популярные статистики по данным датафрейма.
Глава 6 «Индексация данных» расскажет об использовании различных типов индекса библиотеки pandas (Int64Index, RangeIndex, IntervalIndex, CategoricalIndex, DatetimeIndex, PeriodIndex).
Глава 7 «Категориальные данные» познакомит читателя с тем, как создавать объекты Categorical для представления категориальных данных и использовать их в работе.
В главе 8 «Численные и статистические методы» рассматриваются различные арифметические операции над объектами Series и DataFrame, а также вычисление статистик для объектов pandas.
Глава 9 «Загрузка данных» расскажет о том, как можно загрузить данные из внешних источников и записать в объекты Series и DataFrame. Кроме того, в этой главе рассматривается загрузка данных из разных источников, таких как файлы, HTTP-серверы, системы баз данных и веб-службы. Также рассматривается обработка данных в форматах CSV, HTML и JSON.
В главе 10 «Приведение данных в порядок» будет рассказано о том, как приводить данные в порядок, чтобы они были пригодны для анализа.
Глава 11 «Объединение, связывание и изменение формы данных» расскажет читателю о том, как можно взять несколько объектов pandas и объединить их с помощью операций соединения, слияния и конкатенации.
Глава 12 «Агрегация данных» расскажет о группировке и агрегации данных. В библиотеки pandas эти операции выполняются с помощью схемы «разделение − применение − объединение». Читатель научиться использовать эту схему для различных способов группировки данных, а также применять агрегирующие функции для вычисления результатов по каждой группе данных.
Глава 13 «Анализ временных рядов» расскажет о том, как работать с временными рядами в библиотеке pandas. В этой главе будут освещены широкие возможности библиотеки pandas, существенно облегчающие анализ временных рядов.
Глава 14 «Визуализация» научит вас создавать визуализации данных на основе данных, хранящихся в объектов Series и DataFrame. Мы начнем с изучения основ, создания простой диаграммы настройки нескольких параметров диаграммы (настройки легенд, меток и цветов). Мы рассмотрим создание нескольких распространенных типов графиков, которые используются для представления различных типов данных.
В приложении 1 «Советы по оптимизации вычислений в библиотеке pandas» даются некоторые рекомендации по ускорению вычислений в pandas.
Приложение 2 «Улучшение производительности pandas» представляет собой перевод одноименного раздела официального пособия по

библиотеке pandas https://pandas.pydata.org/pandas-docs/stable/enhancingperf.html.
Приложение 3 «Используем pandas для больших данных» расскажет, как за счет использования более эффективных типов данных можно уменьшить использование памяти.
В приложениях 4 и 5 на примере конкурсной задачи Tinkoff Data Science Challenge и конкурсной задачи предсказания отклика ОТП Банка детально показаны этапы предварительной обработки данных, в частности, приведение переменных к нужным типам, обработка редких категорий, импутация пропусков, конструирование признаков, также освещаются специальные процедуры предварительной обработки данных, позволяющие улучшить модель логистической регрессии.
Приложение 6 «Работа с датами и строками» посвящено таким задачам, как правильный парсинг дат различного формата, изменение регистра букв в строках, удаление лишних символов из строк, извлечение нужных символов из строк.
Приложение 7 «Работа с предупреждением SettingWithCopyWarning в библиотеке pandas» посвящено причинам появления предупреждения SettingWithCopyWarning и способам его устранения.
В приложении 8 «От pandas к scikit-learn – новый подход к управлению рабочими процессами» внимание уделено работе с конвейерами.

Что необходимо для чтения этой книги

Эта книга предполагает некоторое знакомство с принципами программирования, но те, кто не имеет опыта программирования или, в частности, опыта программирования на языке Python, будут довольны примерами, поскольку они в большей степени сосредоточены на конструкциях библиотеки pandas, нежели на языке Python или программировании вообще. Примеры приводятся для Anaconda 5.2 для Python 3.6 и pandas 0.23. Если вы не установили их, в главе 2 «Запуск библиотеки pandas» дается инструкция относительно установки pandas в системах Windows, OSX и Ubuntu.

На кого рассчитана эта книга

Эта книга идеально подходит для специалистов по работе с данными, аналитиков данных и программистов Python, которые хотят погрузиться в анализ данных с использованием библиотеки pandas, и всех, кто интересуется анализом данных. Некоторые познания в области статистики и программирования помогут вам извлечь максимальную пользу из этой книги, но они не обязательны. Предварительный опыт работы с pandas также не требуется.

Соглашения

В этой книге используется несколько стилей текста для разных видов информации. Вот несколько примеров этих стилей и объяснение их смысла.

Слова в тексте, обозначающие объекты, функции, методы и другие элементы программного кода отображаются так: «Эту информацию можно легко импортировать в датафрейм с помощью функции `pd.read_csv()` следующим образом».

Блок программного кода, введенный в интерпретаторе Python, отображается следующим образом:

```
import pandas as pd
df = pd.DataFrame.from_items ([('column1', [1, 2, 3])])
print(df)
```

Любой ввод или вывод в командной строке записывается следующим образом:

```
mh@ubuntu:~/Downloads$ chmod +x Anaconda-2.1.0-Linux-x86_64.sh
mh@ubuntu:~/Downloads$ ./Anaconda-2.1.0-Linux-x86_64.sh
```

Новые термины и **важные слова** выделены жирным шрифтом.

Предупреждения и важные примечания выглядят так.

Советы и рекомендации выглядят так.

Отзывы

Мы всегда рады отзывам читателей. Расскажите нам, что вы думаете об этой книге – что вам понравилось или, быть может, не понравилось. Читательские отзывы важны для нас, так как помогают выпускать книги, из которых вы черпаете максимум полезного для себя. Чтобы отправить обычный отзыв, просто пошлите письмо на адрес feedback@packtpub.com, указав название книги в качестве темы. Если вы являетесь специалистом в некоторой области и хотели бы стать автором или соавтором книги, познакомьтесь с инструкциями для авторов по адресу www.packtpub.com/authors.

Поддержка клиентов

Счастливым обладателям книг Packt мы можем предложить ряд услуг, которые позволят извлечь из своего приобретения максимум пользы.

Загрузка программного кода примеров

Вы можете скачать программный код примеров для этой книги, воспользовавшись своей учетной записью на сайте http://www.packtpub.com. Если вы приобрели эту книгу в другом месте,

вы можете посетить страницу http://www.packtpub.com/support и зарегистрироваться, чтобы получить файлы непосредственно по электронной почте. Вы можете скачать программный код примеров, выполнив следующие действия:

1. Войдите в систему или зарегистрируйтесь на нашем веб-сайте, используя свой адрес электронной почты и пароль.
2. Наведите указатель мыши на вкладку **SUPPORT** вверху.
3. Щелкните указателем мыши по **Code Downloads & Errata**.
4. Введите название книги в поле **Search**.
5. Выберите книгу, программный код которой хотите скачать.
6. В выпадающем меню укажите, где была приобретена книга.
7. Щелкните указателем мыши по **Code Download**.

После загрузки файла распакуйте его в определенную папку, используя последнюю версию архиватора:

- WinRAR / 7-Zip для Windows
- Zipeg / iZip / UnRarX для Mac
- 7-Zip / PeaZip для Linux

Кроме того, программный код для этой книги размещен на GitHub по адресу https://github.com/PacktPublishing/Learning-Pandas-Second-Edition. У нас есть и другие наборы программного кода благодаря нашему богатому каталогу книг и видеороликов, расположенного по адресу https://github.com/PacktPublishing/. Ознакомьтесь с ними!
Программный код с русифицированными комментариями размещен на GitHub по адресу https://github.com/Gewissta/Learning_pandas_russian_translation.

Опечатки

Мы проверяли содержимое книги со всем тщанием, но какие-то ошибки все же могли проскользнуть. Если вы найдете в нашей книге ошибку, в тексте или в коде, пожалуйста, сообщите нам о ней. Так вы избавите других читателей от разочарования и поможете нам сделать следующие издания книги лучше. При обнаружении опечатки просьба зайти на страницу http://www.packtpub.com/submit-errata, выбрать книгу, щелкнуть по ссылке **Errata Submission Form** и ввести информацию об опечатке. Проверив ваше сообщение, мы поместим информацию об опечатке на нашем сайте или добавим ее в список замеченных опечаток в разделе **Errata** для данной книги.
Список ранее отправленных опечаток можно просмотреть, выбрав название книги на странице http://www.packtpub.com/books/content/support. Запрошенная информация появится в разделе **Errata**.

Нарушение авторских прав

Незаконное размещение защищенного авторским правом материала в Интернете – проблема для всех носителей информации. В издательстве

Packt мы относимся к защите прав интеллектуальной собственности и лицензированию очень серьезно. Если вы обнаружите незаконные копии наших изданий в любой форме в Интернете, пожалуйста, незамедлительно сообщите нам адрес или название вебсайта, чтобы мы могли предпринять соответствующие меры.

Просим отправить ссылку на вызывающий подозрение в пиратстве материал по адресу copyright@packtpub.com. Мы будем признательны за помощь в защите прав наших авторов и содействие в наших стараниях предоставлять читателям полезные сведения.

Вопросы

Если вас смущает что-то в этой книге, вы можете связаться с нами по адресу questions@packtpub.com, и мы сделаем все возможное для решения проблемы.

ГЛАВА 1 БИБЛИОТЕКА PANDAS И АНАЛИЗ ДАННЫХ

Добро пожаловать на страницы книги «Изучаем pandas»! В этой книге мы отправимся в путешествие, в ходе которого вы научитесь работать с pandas, библиотекой анализа данных с открытым исходным кодом, предназначенной для языка программирования Python. Библиотека pandas предлагает высокопроизводительные и простые в использовании структуры данных и инструменты анализа, созданные с помощью языка Python. Библиотека pandas привнесла в Python массу полезных инструментов, взяв их из языка статистического программирования R, в частности, объекты **data frame (датафрейм)**, пакеты R, например, `plyr` и `reshape2`, и разместила их в одной библиотеке, которую вы можете использовать в среде Python.

В первой главе мы посвятим время базовому знакомству с библиотекой pandas и тому, как она вписывается в обширную картину анализа данных. Вместо того, чтобы полностью сосредоточиться на конкретных аспектах использования библиотеки pandas, эта глава призвана дать читателю ощущение своего места в обширной картине анализа данных. Цель состоит в том, чтобы при изучении библиотеки pandas вы также узнали о том, зачем нужны все эти различные функции, выполняющие задачи анализа данных.

Итак, давайте начнем. В этой главе мы рассмотрим:

- Что из себя представляет библиотека pandas, почему она была создана и что она вам даст
- Как библиотека pandas связана с анализом данных и наукой о данных
- Этапы анализа данных и их поддержка в библиотеке pandas
- Общие понятия «данные» и «аналитика»
- Основные понятия анализа данных и статистического анализа
- Типы данных и их использование в библиотеке pandas
- Другие библиотеки в экосистеме Python, которые вы, вероятно, будете использовать вместе с pandas

Знакомство с библиотекой pandas

Библиотека pandas – это библиотека Python, содержащая высокоуровневые структуры данных и инструменты, которые были созданы, чтобы помочь программистам Python осуществить полноценный анализ данных. Конечная цель библиотеки pandas заключается в том, чтобы помочь вам быстро найти необходимую информацию, скрытую в данных, при этом информацию содержательного характера.

Разработка библиотеки pandas была начата в 2008 году Уэсом Маккинни и представлена в 2009 году как проект с открытым исходным программным кодом. В настоящее время библиотека pandas активно

курируется и разрабатывается различными организациями и участниками.
Первоначально библиотека pandas предназначалась для применения в финансах, в частности, благодаря ее возможностям работы с временными рядами и обработке исторической информации об акциях. Обработка финансовой информации сопряжена с массой проблем, вот некоторые из них:

- Обработка данных (например, данных о котировках акций), меняющихся с течением времени
- Необходимость единого стандарта измерений нескольких потоков данных в один и тот же период времени
- Определение взаимосвязи (корреляции) между двумя и более потоками данных
- Представление дат и времени в качестве объектов первого класса
- Увеличение или уменьшение шага дискретизации временного ряда

Для выполнения этих операций необходим инструмент, который позволяет нам извлекать, индексировать, очищать и приводить в порядок, изменять и объединять данные, создавать срезы данных и выполнять различные виды анализа как для одномерных, так и для многомерных данных, включая данные разного типа, которые автоматически выравниваются по набору общих индексных меток. И вот как раз здесь на помощь приходит библиотека pandas, которая предлагает множество полезных и мощных функций, например:

- Быстрые и эффективные объекты Series и DataFrame для обработки данных с помощью встроенной индексации
- Интеллектуальное выравнивание данных с помощью индексов и меток
- Интегрированная обработка пропущенных данных
- Инструменты для приведения данных в порядок
- Встроенные инструменты для чтения и записи данных для обмена между объектами Series и DataFrame в памяти, файлами, базами данных и веб-службами
- Возможность обработки данных, хранящихся в различных популярных форматах, таких как CSV, Excel, HDF5 и JSON
- Изменение формы и поворот данных
- Интеллектуальное создание срезов данных на основе меток, сложная индексация и отбор из больших наборов данных подмножеств по определенному критерию
- Удаление и вставка столбцов из объектов Series и DataFrame для изменения размера
- Агрегирование или преобразование данных с помощью мощного инструмента «разделение – применение - объединение»
- Иерархическая индексация, облегчающая работу с высокоразмерными данными в низкоразмерной структуре данных
- Высокопроизводительное слияние и соединение наборов данных
- Разнообразные функции для работы с временными рядов, включая создание диапазона дат и преобразование частоты временного ряда, вычисление скользящих статистик, скользящих линейных регрессий, смещение дат и сдвиг временного ряда с запаздыванием

- Оптимизация для достижения более высокой производительности, включающая программный код, написанный на Cython или C

Мощный набор функций в сочетании с бесшовной интеграцией с Python и другими инструментами экосистемы Python позволил библиотеке pandas найти широкое применение во многих областях. Она используется в самых разных академических и коммерческих областях, включая финансы, нейробиологию, экономику, статистику, рекламу и веб-аналитику. Она стал одним из наиболее предпочтительных инструментов для специалистов по работе с данными.

Python долгое время широко использовался для сбора данных и подготовки, но при этом в меньшей степени был предназначен для анализа данных и моделирования. Библиотека pandas помогает заполнить этот пробел, позволяя вам выполнить весь рабочий процесс анализа данных в среде Python, не переходя на такой более специализированный язык, как R. Это очень важно, поскольку люди, знакомые с языком Python, являющимся более универсальным языком программирования, чем R (язык, ориентированный в большей степени на статистиков), получают в свое распоряжение массу функций по представлению и обработке данных, имеющихся в R, и при этом полностью остаются в невероятно богатой экосистеме Python.

В сочетании с IPython, тетрадками Jupyter и широким выбором других библиотек среда Python в плане выполнения анализа данных превосходит по производительности, эффективности и возможности совместной работы многие другие инструменты. Все это привело к тому, что многие пользователи широко применяют библиотеку pandas в самых различных отраслях.

Обработка данных, анализ, наука и библиотека pandas

Мы живем в мире, в котором каждый день генерируются и записываются огромные объемы данных. Эти данные поступают из множества информационных систем, устройств и датчиков. Почти все, что вы делаете, и все то, что вы используете в рамках своей деятельности, генерирует данные, которые можно собрать или они уже собраны.

Во многом такая ситуация была обусловлена повсеместным распространением услуг, связанных с информационными сетями, и стремительно возросшими возможностями хранения данных. Все это в сочетании с постоянно снижающейся стоимостью хранения повысило эффективность сбора и хранения даже самых тривиальных данных.

В итоге было накоплены огромные объемы данных, готовые для загрузки. Но эти данные сосредоточились по всему киберпространству и на самом деле их еще нельзя назвать **информацией (information)**. Данные представляют собой коллекции зарегистрированных событий, будь то финансовые данные или ваше общение в социальных сетях или ваш персональный трекер здоровья, отслеживающий сердцебиение в

течение дня. Эти данные хранятся в различных форматах, расположены в разных местах и исследование сырых данных может дать ценную информацию.
Логично, что весь процесс можно разбить на три основные дисциплины:

- Обработка данных
- Анализ данных
- Наука о данных

Эти три дисциплины часто пересекаются. Вопросы о том, где заканчивается одна дисциплина и начинается другая, остаются открытыми. В следующих разделах мы дадим определения этим дисциплинам.

Обработка данных

Данные разбросаны по всей планете. Они хранятся в разных форматах. Они имеют разный уровень качества. Поэтому существует потребность в инструментах и процессах сбора данных, дающим такое представление данных, которое можно использовать для принятия решений. Инструмент, который используется для работы с данными на этапе подготовки к анализу, должен уметь решать различные задачи. Функционал такого инструмента включает в себя:

- Программируемость для повторного использования и совместного использования
- Загрузка данных из внешних источников
- Локальное сохранение данных
- Индексация данных для их эффективного извлечения
- Выравнивание данных в разных наборах на основе атрибутов
- Объединение данных, расположенных в разных наборах
- Преобразование данных в другое представление
- Очистка данных от «мусора»
- Эффективная обработка загрязненных данных
- Группировка данных
- Агрегирование данных по схожим характеристикам
- Применение функций, вычисляющих среднее или выполняющих преобразования
- Выполнение запросов или создание срезов для исследования подмножеств данных
- Изменение формы данных
- Создание отдельных категорий данных
- Изменение частоты временного ряда

Существует масса инструментов для обработки данных. Каждый из них отличается поддержкой элементов этого списка, способами их развертывания и способами их использования. Эти инструменты включают в себя реляционные базы данных (SQL Server, Oracle), электронные таблицы (Excel), системы обработки событий (такие как Spark) и более общие инструменты, такие как R и pandas.

Анализ данных

Анализ данных – это процесс извлечения смысла из данных. Данные, представленные в количественном виде, часто называют **информацией (information)**. Анализ данных – это процесс получения информации из данных путем создания моделей и применения математического аппарата для поиска закономерностей. Он часто перекликается с обработкой данных и не всегда можно четко провести различие между ними. Многие инструменты обработки данных также содержат аналитические функции, а инструменты анализа данных часто предлагают возможности обработки данных.

Наука о данных

Наука о данных – это процесс использования статистики и анализа данных для понимания **явлений (phenomena)**, скрытых в данных. Наука о данных обычно начинается с информации и применяет к ней более сложный анализ на основе знаний, относящихся к разным предметным областям. К этим предметным областям относятся математика, статистика, информатика, компьютерные науки, машинное обучение, классификация, кластерный анализ, интеллектуальный анализ данных, базы данных и визуализация. Наука о данных носит многодисциплинарный характер. Ее методы анализа могут сильно отличаться друг от друга и зависеть от конкретной предметной области.

Предназначение библиотеки pandas

В первую очередь библиотека pandas – превосходный инструмент обработки данными. Все потребности, описанные ранее, будут рассмотрены в этой книге с использованием библиотеки pandas. На решение эти задач и направлен основной функционал библиотеки pandas и именно на решении большей части этих задач мы и сосредоточимся в этой книге.

Стоит отметить, что основное предназначение библиотеки pandas – это подготовка данных. Однако библиотека pandas также предоставляет несколько функций для выполнения анализа данных. Эти возможности подразумевают вычисление описательных статистик и функций, необходимых для финансового анализа, например, вычисления корреляции.

Поэтому сама по себе библиотека pandas не являются инструментом для научных исследований. Это скорее инструмент обработки данных с некоторыми возможностями анализа. Библиотека pandas явно оставляет за скобками сложный статистический, финансовый анализ, предлагая его выполнить другим библиотекам Python, таким как SciPy, NumPy, scikit-learn, а для визуализации данных использует графические библиотеки, например, **matplotlib** и **ggvis**.

Мы сосредоточимся на преимуществе библиотеки pandas над другими языками, такими как R, поскольку приложения на основе библиотеки pandas могут использовать обширную сеть надежных фреймворков Python, уже созданных и протестированных Python-сообществом.

Процесс анализа данных

Основная цель этой книги – научить вас использовать библиотеку pandas для обработки данных. Однако есть второстепенная и, возможно, не менее важная цель показать, как библиотека pandas встроена в те процессы, которые специалист по работе с данными/аналитик данных выполняет в повседневной жизни.
Описание шагов, связанных с процессом анализа данных, дается на веб-странице библиотеки pandas:

- Обработка и очистка данных
- Анализ/моделирование данных
- Преобразование данных в удобную форму

Этот небольшой список является хорошим исходным определением, но он не охватывает процесс анализа в целом и не может ответить на вопрос, почему в библиотеке pandas реализовано так много инструментов. В следующем разделе этот процесс будет рассмотрен подробнее.

Процесс

Предлагаемый процесс - это последовательность операций, которую называют процессом анализом данных, и он представлен на следующей диаграмме:

Эта схема задает структуру для определения логических шагов, которые будут предприняты в ходе работы с данными. Сейчас давайте кратко рассмотрим каждый этап этого процесса, а также некоторые задачи, которые вы как аналитик данных будете выполнять, используя библиотеку pandas.
Важно понимать, что это не простой линейный процесс. Лучше всего этот процесс осуществлять интерактивно и гибким/итеративным способом.

Выдвижение идей

Первый этап процесса анализа данных – определить, что вы хотите выяснить. Он называется **выдвижение идей (ideation)**, мы формулируем идеи по поводу того, что мы хотим сделать и доказать. Идея в целом связана с гипотезой о наличии тех или иных структур в данных, которые можно использовать для принятия решений.
Эти решения часто принимаются в контексте бизнеса, а также в других дисциплинах, например, в научно-исследовательской деятельности. Сейчас модно применять анализа данных для решения бизнес-задач, поскольку глубокое понимание данных может повысить прибыль компаний.

Однако какое решение мы обычно хотим принять? Ниже приводятся некоторые наиболее часто задаваемые вопросы:

- Почему это происходит?
- Можем ли мы предсказать будущее с использованием исторических данных?
- Как я могу оптимизировать бизнес-операции в будущем?

Этот список ни в коем случае не является исчерпывающим, но он охватывает значительную часть причин, по которым проводится анализ данных. Чтобы получить ответы на эти вопросы, нужно осуществить сбор и анализ данных, связанных с решаемой проблемой. Необходимо определить, какие данные мы будем изучать, какова польза от исследования, как будут получены данные, каковы критерии успешности решения и как в конечном итоге информация будет представлена.

Сама по себе библиотека pandas не предлагает инструментов, позволяющих формулировать идеи. Но как только вы научитесь использовать библиотеку pandas, вы, естественно, поймете, как pandas может помочь вам в формулировании идей. Ведь к этому моменту вы вооружитесь мощным инструментом, который можно использовать для выдвижения множества сложных гипотез.

Сбор данных

Как только вы сформулировали идею, вы должны получить данные, чтобы попытаться проверить свою гипотезу. Эти данные могут быть собраны вашей организацией или получены от внешних поставщиков данных. Эти данные обычно поставляются в виде архивных данных или могут быть предоставлены в режиме реального времени (хотя pandas не так широко известен как инструмент обработки данных в режиме реального времени).

Часто данные являются «сырыми», даже если они взяты из источников данных, созданных вами или вашей организацией. Понятие «сырые» означает, что данные могут быть неправильно сформированы, записаны в разных форматах или содержать ошибки. Они могут быть неполными и может потребоваться аугментация данных[1].

В мире имеется множество бесплатных данных. Многие данные не являются бесплатными и фактически требуют значительных денежных сумм. Некоторые из них есть в свободном доступе и имеют публичные API-интерфейсами, а другие можно получить по подписке. Часто данные, которые вы покупаете, являются более чистыми, но это не всегда так.

В любом случае библиотека pandas предлагает надежный и простой в использовании набор инструментов для сбора данных из разных источников, при этом данные могут иметь разный формат. Кроме того, библиотека pandas позволяет не только извлекать данные, но и выполнять первоначальную структуризацию данных с помощью

[1] Аугментация данных (data augmentation) – это создание дополнительных обучающих данных из имеющихся данных. – *Прим. пер.*

объектов Series и DataFrame, не прибегая к написанию сложного программного кода, который может потребоваться в других инструментах или языках программирования.

Подготовка данных

На этом этапе мы подготавливаем исходные данные для проведения дальнейшего исследования. Часто подготовка является весьма интересным этапом анализа данных. Как правило, все проблемы, связанные с данными, связаны с качеством этих данных. Вы, вероятно, потратите много времени на устранение проблем, связанных с качеством данных.

Зачем? Ну, есть ряд причин:

- Данные просто некорректны.
- Какая-то информация в наборе данных отсутствует
- Данные записаны в единицах измерения, не подходящих для вашего анализа
- Данные записаны в форматах, не удобных для вашего анализа
- Данные находятся на том уровне детализации, который не подходит для вашего анализа.
- Не все интересующие вас поля можно получить из одного источника
- Представление данных отличается в зависимости от поставщика

Процесс подготовки данных сосредоточен на решении вышеприведенных проблем. Библиотека pandas предлагает множество отличных возможностей для процесса подготовки данных, который часто называют **приведением данных в порядок (data tidying)**. Ее арсенал включает в себя интеллектуальные средства обработки пропущенных данных, преобразование типов данных, преобразование форматов, изменение частоты измерений, объединение данных, расположенных в нескольких наборах данных, кодировку/преобразование символов и группировку данных среди прочих операций. Мы подробно рассмотрим все эти операции.

Исследование данных

Исследование данных подразумевает изучение ваших данных, чтобы попытаться открыть какие-то закономерности в данных. Исследование может включать в себя различные задачи, например:

- Изучение того, как переменные связаны друг с другом
- Анализ распределения данных
- Обнаружение и исключение выбросов
- Создание быстрых визуализаций
- Быстрое создание новых представлений данных или моделей для последующего использования в ходе моделирования

Исследование данных является одной из самых сильных сторон библиотеки pandas. Хотя большинство языков программирования также позволяют выполнить исследование данных, каждый из них требует подготовительных действий, не связанный непосредственно с исследованием данных,.

Используя принцип **read-eval-print-loop (REPL)**, реализованный в IPython и/или тетрадках Jupyter, библиотека pandas создает исследовательскую среду, в которой объем этих подготовительных действий сведен к минимуму. Выразительность синтаксиса библиотеки pandas позволяет кратко описать сложные операции с данными, а немедленно появляющийся результат выполненной операции помогает быстро проверить правильность ваших действий, не прибегая к повторной компиляции и полному переписыванию своего программного кода.

Моделирование данных

На этапе моделирования вы формализуете свои открытия, найденные в ходе исследования данных. Для этого закономерностям, обнаруженным в данных и позволяющим извлечь из данных содержательный смысл, нужно дать ясную интерпретацию. Итогом становится **модель (model)**, позволяющая выразить обнаруженные закономерности в математическом виде и содержащая программный код, позволяющий преобразовать данные в полезную информацию.

Процесс моделирования является итеративным, когда с помощью исследования данных вы выбираете переменные, необходимые для вашего анализа, подаете переменные на вход модели, строите модель и определяете, насколько хорошо модель подтверждает ваши исходные предположения. Он может включать формальное моделирование структуры данных, а также может сочетать методы из различных областей, например, из статистики, машинного обучения и исследование операций.

Библиотека pandas предлагает мощные инструменты для моделирования данных. Именно на этом этапе вы формализуете модель данных, используя объектами DataFrame и стараясь сделать процесс построения модели максимально компактным. Кроме того, вы можете воспользоваться всеми возможностями языка Python, чтобы полностью автоматизировать процесс создания модели.

С аналитической точки зрения библиотека pandas предлагает в первую очередь интегрированную поддержку описательной статистики, которая понадобится вам при решении разных задач. Впрочем, библиотека pandas использует язык Python и поэтому если вам нужны более продвинутые аналитические возможности, вы очень легко можете интегрировать pandas с другими библиотеками обширной научной среды Python.

Представление результатов

Предпоследний этап процесса анализа данных – представление результатов, как правило, в виде отчета или презентации. Вам нужно дать убедительное и подробное объяснение своего решения. Часто это можно сделать с помощью различных инструментов визуализации в Python и затем вручную создать презентацию.

Тетрадки (ноутбуки) Jupiter – это мощный инструмент для создания презентаций по итогам вашего анализа. Эти тетрадки представляют собой инструмент как для выполнения кода, так и для подробного аннотирования выполняемых действий. Их можно использовать для создания высокоэффективных презентаций, включающих фрагменты программного кода, стилизованный текст и графику.
Мы кратко рассмотрим тетрадки Jupyter в главе 2 «Запуск библиотеки pandas».

Воспроизведение результатов анализа

Важный этап процесса анализа данных – обмен и воспроизведение результатов. Часто говорят, что если другие исследователи не смогут воспроизвести ваш эксперимент и полученные результаты, то вы ничего не доказали.
К счастью для вас, вы легко сможете сделать свой анализ воспроизводимым, воспользовавшись библиотекой pandas и Python. Это можно сделать, поделившись с другими исследователями программным кодом Python, который лежит в основе программного кода библиотеки pandas, а также данными.
Тетрадки Jupyter также являются удобным инструментом для хранения программного кода и проектов, которыми можно легко поделиться со всеми у кого установлен Jupyter Notebook. В Интернете существует много бесплатных и безопасных сайтов обмена, которые позволяют вам создавать или размещать свои тетрадки Jupyter для совместного использования.

По поводу итеративности и гибкости

Очень важно понимать, что обработка данных, анализ и наука – это итеративный процесс. Несмотря на то, что ранее мы рассмотрели этапы, выстроив в прямую последовательность, в конечном итоге вы будете двигаться как в прямом, так и в обратном порядке. Например, на этапе исследования вы можете выявить аномалии в данных, которые связаны с проблемами чистоты данных, они решаются на этапе подготовки и поэтому необходимо вернуться, чтобы исправить эти проблемы.
Это обычная ситуация, возникающая в процессе анализа данных. Вы словно совершаете путешествие, пытаетесь решить свою первоначальную задачу, все время получая новое, дополнительное представление о данных, с которыми работаете. Эта информация, возможно, побудит вас задать новые вопросы, более точные вопросы или осознать, что ваши первоначальные вопросы не являлись теми вопросами, на которые вы хотели бы найти ответы. Процесс анализа данных - это и в самом деле путешествие, а не конкретный пункт назначения.

Взаимосвязь между книгой и процессом анализа данных

Ниже приведено краткое описание этапов анализа данных, о которых вы узнаете в этой книге. Не волнуйтесь, что последовательность этапов не совпадает с последовательностью изложения материала в книге. Книга в логическом порядке знакомит вас с библиотекой pandas, и вы можете вернуться к любой главе, которая соответствует интересующему вас этапу анализа данных.

Этап процесса анализа данных	Глава книги
Выдвижение идей	Выдвижение идей – это самая творческая часть науки о данных. У вас должна быть идея. Она должна подтверждаться фактами (здесь вам понадобятся данные) и должна быть возможность наблюдать эти факты после проведения анализа (на основе фактов из прошлого строим модель, прогнозируя их в будущем)..
Сбор данных	Сбор данных главным образом освещается в главе «Загрузка данных».
Подготовка данных	Подготовка данных преимущественно освещается в главе 10 «Приведение данных в порядок», но это довольно общая тема, которая встречается в большинстве глав этой книги.
Исследование данных	Тема исследования данных охватывает материал, начинающий с главы 3 «Представление одномерных данных с помощью объекта Series» и заканчивающийся главой 14 «Визуализация». Таким образом, большая часть книги посвящена исследованию данных. Но наиболее подробное освещение данного этапа дается в главе 14 «Визуализация».
Моделирование данных	Моделирование данных подробно освещается в главе 3 «Представление одномерных данных с помощью объекта Series» и главе 4 «Представление табличных и многомерных данных с помощью объекта DataFrame», а также в главе 11 «Объединение, связывание и изменение формы данных» и главе 13 «Анализ временных рядов».
Представление результатов	Представление результатов – это главная тема главы 14 «Визуализация».
Воспроизведение результатов	Тема воспроизведения результатов проходит по всей книге, поскольку примеры представлены в виде тетрадок Jupyter. Работая в тетрадках, вы по умолчанию используете инструмент для воспроизведения результатов и у вас есть возможность поделиться этими тетрадками различными способами.

Понятия «данные» и «анализ» в контексте нашего знакомства с библиотекой pandas

По мере изучения библиотеки pandas и анализа данных вы будете сталкиваться с разными понятиями из области данных, моделирования и анализа. Давайте рассмотрим некоторые из этих понятий и выясним, как они связаны с библиотекой pandas.

Типы данных

При работе с сырыми данными вы столкнетесь с несколькими типами данных, которые нужно будет представить с помощью объектов pandas. Они важны для понимания, поскольку инструменты, необходимые для работы с каждым типом, отличаются друг от друга.

Библиотека pandas по сути используется для обработки структурированных данных. Однако помимо этого она предлагает несколько инструментов, позволяющих преобразовать неструктурированные данные в такое представление, которое мы уже сможем обработать.

Структурированные данные

Структурированные данные – это тип данных, при котором последние упорядочены в вертикальные столбцы (поля) и горизонтальные строки (записи или наблюдения). Примером таких данных являются данные в реляционных базах и электронных таблицах. Структурированные данные зависят от модели данных, которая представляет собой определенную структуру, содержательного смысла данных и часто от способа обработки данных. Сюда входит идентификация типа данных (целое число, число с плавающей точкой, строка и т.д.) и ограничения, накладываемые на данные, например, количество символов, максимальное и минимальное значения, или ограничение на определенный набор значений.

Структурированный данные - это тип данных, который предназначен для использования в библиотеке pandas. Как мы увидим сначала на примере объекта `Series`, а затем на примере объекта `DataFrame`, pandas помещает структурированные данные в один или несколько столбцов данных (при этом каждый столбец имеет один конкретный типа данных) и одну или несколько строк данных.

Неструктурированные данные

Неструктурированные данные – это данные, которые не имеют определенной структуры, не предполагают наличия заранее определенных столбцов определенного типа. Примером неструктурированных данных могут быть такие виды информации, как фотографии и графические изображения, видеоролики, потоковые данные датчиков, веб-страницы, PDF-файлы, презентации PowerPoint, электронные письма, записи в блогах, страницы Википедии и документы Word.

Хотя библиотека pandas не может обработать неструктурированные данные напрямую, она предлагает ряд инструментов для сбора структурированных данных из неструктурированных источников. В качестве конкретного примера мы рассмотрим инструменты pandas, позволяющие извлекать веб-страницы, определенные фрагменты контента и записывать в объект `DataFrame`.

Полуструктурированные данные

Полуструктурированные данные занимают промежуточное положение между неструктурированными и структурированными данными. Их можно рассматривать как тип структурированных данных, у которых нет строгой структуры моделей данных. Формат JSON – это пример

полуструктурированных данных. Хотя хороший JSON-файл будет иметь определенный формат, здесь нет определенной схемы данных, которая всегда строго соблюдаются. В большинстве случаев данные будут записаны в виде определенной схемы, которую легко преобразовать в структурированный тип данных, например, в объект `DataFrame`, но для этого, возможно, потребуется задать определенный тип данных.

Переменные

Моделируя данные в библиотеке pandas, мы будем исследовать одну или несколько переменных и искать статистический смысл, анализируя значения этой переменной или значения нескольких переменных. При этом термин «переменная» не тождественен понятию «переменная» в программировании, здесь речь идет о статистических переменных.
Переменной является любая характеристика, число или количество, которое можно измерить или сосчитать. Название «переменная» обусловлено, тем что значение характеристики может меняться от наблюдения к наблюдению, а также может меняться со временем. Значение цены акции, возраст, пол, доходы и расходы в бизнесе, страна рождения, капитальные затраты, школьные оценки, цвет глаз и тип транспортного средства являются примерами переменных.
Существует несколько общих типов статистических переменных, с которыми мы столкнемся при работе с библиотекой pandas:

- Категориальные переменные
- Непрерывные переменные

Категориальные переменные

Категориальная переменная (categorical variable) – это переменная, которая может принимать одно значение из ограниченного и обычно фиксированного набора возможных значений. Каждое из возможных значений часто называется **уровнем (level)**. Примерами категориальных переменных являются пол, штат, социальный класс, группа крови, гражданство, время наблюдения или рейтинг, например, шкала Лайкерта. Все категориальные переменные являются качественными характеристиками, которые могут описать продукт, но при этом не измеряются в непрерывной шкале. Например, возьмем переменную *Штат*. Допустим, он имеет уровни Аризона, Северная Каролина и Висконсин. Человек либо проживает в штате Аризона, либо в штате Северная Каролина, либо в штате Висконсин. Не существует золотой середины между штатами, нельзя вычислить среднее значение штата и нет естественного способа упорядочить эти категории, нельзя сказать, что штат Аризона хуже/лучше штата Северная Каролина, а штат Северная Каролина хуже/лучше штата Висконсин. Категориальные переменные в библиотеке pandas представлены объектами `Categorical`, специальным типом данных, который соответствует категориальным переменным в статистике.

Непрерывные переменные

Непрерывная переменная (continuous variable) – это переменная, которая может принимать бесконечное (неисчислимое) количество значений. Все непрерывные переменные являются характеристиками, которые количественно описывают продукт и измеряются в непрерывной шкале. Примерами непрерывных переменных являются возраст, высота, время и температура. Мы можем вычислить средний возраст, среднюю высоту, среднее время и среднюю температуру. Мы можем упорядочить значения. 20-летний младше 30-летнего, а 30-летний младше 40-летнего. Непрерывные переменные в библиотеке pandas представлены либо типом `float`, либо типом `integer` (нативными питоновскими типами данных).

Временные ряды

В библиотеке pandas временные ряды – это объект первого класса. Время добавляет важное дополнительное измерение. Часто переменные не зависят от времени их регистрации, то есть момент времени, в который их фиксировали, не имеет значения. Но во многих случаях это не так. Временной ряд представляет собой переменную, у которой значения зарегистрированы в определенные временные интервалы, таким образом, наблюдения изначально упорядочены по времени.

Стохастическая модель для временного ряда обычно отражает тот факт, что наблюдения, близкие друг к другу во времени, будут более тесно связаны, чем наблюдения, которые далеко отстоят друг от друга по времени. Модели временных рядов часто используют естественное одностороннее упорядочение времени, поэтому значения для заданного периода будут выражаться как полученные из прошлого, а не из будущего.

Чаще всего в библиотеке pandas работают с финансовыми данными, где переменная представляет собой стоимость акции, изменяющаяся через равные интервалы времени в течение дня. Нам часто нужно определить скорость изменения цены в определенные интервалы времени. Кроме того, мы можем скорректировать цены нескольких акций по определенным интервалам времени.

Это настолько важная и мощная возможность библиотеки pandas, что мы посвятим ей целую главу.

Общие понятия анализа и статистики

В этом книге мы только коснемся общих понятий статистики и технической стороны анализа данных. Однако стоит раскрыть несколько понятий, некоторые из которых реализованы непосредственно в библиотеке pandas. Остальные понятия связаны с другими библиотеками, например, с библиотекой SciPy, однако вы также можете

встретить их во время работы с библиотекой pandas, поэтому мы тоже о них расскажем.

Количественный и качественный анализ

Качественный анализ - это научное исследование данных, которые можно наблюдать, но нельзя количественно измерить. Основное внимание уделяется качественным характеристикам. Примерами качественных данных могут быть:

- Мягкость кожи
- Изящность бега

Количественный анализ – это исследование данных, которые можно измерить количественно. Примерами количественных данных могут быть:

- Количество
- Цена
- Высота

Библиотека pandas главным образом работает с количественными данными, предлагая разнообразные инструменты для их обработки. Библиотека pandas не предназначена для проведения качественного анализа, но позволяет вам представить качественную информацию в различных видах.

Одномерный и многомерный анализ

В каком-то смысле статистика представляет собой практику изучения переменных и, в частности, наблюдение за этими переменными. Статистика преимущественно опирается на анализ одной переменной, которая называется одномерным анализом. **Одномерный анализ (univariate analysis)** – это простейшая форма анализа данных. Он не имеет отношения к анализу причин или взаимосвязей и обычно используется для описания или подытоживания данных, а также поиска закономерностей в данных.

Многомерный анализ (multivariate analysis) – это метод моделирования, в котором участвуют две или более переменных, которые влияют на результат эксперимента. Многомерный анализ часто связан с такими понятиями, как корреляция и регрессия, которые помогают нам понять взаимосвязь между несколькими переменными, а также влияние этой взаимосвязи на результат.

Библиотека pandas в основном предлагает инструменты для проведения одномерного анализа. Главным образом речь идет о вычислении описательных статистик, хотя можно вычислить такой показатель, как корреляция (поскольку ее часто используют в финансах и других областях).

Остальные более сложные статистические процедуры можно выполнить с помощью библиотеки StatsModels. Опять же это не недостаток библиотеки pandas, а конкретное решение, направленное на то, чтобы

более сложные операции выполнять с помощью специализированных библиотек Python.

Описательные статистики

Описательные статистики – это показатели, которые подытоживают данные, обычно в тех случаях, где набор данных представляет собой генеральную совокупность или выборку из одной переменной (одномерные данные). Эти показатели описывают набор данных и являются мерами центральной тенденции, а также мерами изменчивости и дисперсии.

Например, следующие показатели являются описательными статистиками:

- Распределение (например, нормальное, пуассоновское)
- Центральная тенденция (например, среднее, медиана и мода)
- Дисперсия (например, дисперсия, стандартное отклонение)

Позже мы увидим, что объекты `Series` и `DataFrame` поддерживают вычисление большей части описательных статистик.

Индуктивные статистики

Индуктивные статистики отличаются от описательных тем, что с их помощью мы пытаемся сделать выводы о данных, а не просто подытожить данные. Примерами индуктивных статистик являются:

- t-тест
- хи-квадрат
- ANOVA
- бутстреп

Эти индуктивные методы были перенесены из библиотеки pandas в другие инструменты типа SciPy и StatsModels.

Стохастические модели

Стохастические модели представляют собой вид статистического моделирования, который включает в себя одну или несколько случайных величин, а также подразумевает использование временных рядов. Цель стохастической модели – оценить вероятность того, что результат будет находиться в пределах определенного интервала, и спрогнозировать условия для разных ситуаций.

Примером стохастического моделирования являются симуляции Монте-Карло. Симуляции Монте-Карло применяются для расчета стоимости финансовых портфелей, которые подвержены влиянию различных риск-факторов и имеют нетривиальные распределения доходности.

Библиотека pandas предлагает фундаментальную структуру данных DataFrame, ее часто используют при работе с временными рядами, чтобы на их основе потом создать и запустить стохастическую модель. Хотя можно создать свои собственные стохастические модели и провести анализ с помощью pandas и Python, во многих случаях более удобным

инструментом для стохастического моделирования будут специализированные библиотеки типа PyMC.

Вероятность и байесовская статистика

Байесовская статистика – это подход к статистическому оцениванию, основанный на теореме Байеса, математическом уравнении, использующем простые аксиомы теории вероятностей. Она позволяет аналитику вычислить любую условную вероятность интересующего события. Условная вероятность – это просто вероятность события A при условии, что наступило событие B.

Поэтому, если использовать термины теории вероятностей, события уже произошли и были зафиксированы (поскольку мы знаем вероятность). Используя теорему Байеса, мы можем вычислить вероятность различных интересующих нас событий, учитывая уже имеющиеся данные.

Байесовское моделирование выходит за рамки этой книги, но опять же соответствующие модели данных прекрасно обрабатываются с помощью библиотеки pandas и затем их можно анализировать с помощью таких библиотек как PyMC.

Корреляция

Корреляция является одной из наиболее распространенных статистик и ее можно вычислить с помощью соответствующего метода объекта DataFrame. Корреляция – это число, которое описывает степень взаимосвязи между двумя переменными.

Распространенным примером использования корреляции является определение того, насколько сильно цены двух акций меняются вместе с течением времени. Если изменения сильно связаны, эти две акции имеют высокую корреляцию, и если нет, между ними нет корреляции. Это ценная информация, которую можно использовать в ряде инвестиционных стратегий.

Кроме того, корреляция двух акций может зависеть от временного периода, который охватывает набор данных, а также от интервала внутри этого временного периода. К счастью, библиотека pandas обладает мощными возможностями, позволяющими учесть это и повторно вычислить корреляцию. Мы рассмотрим корреляции в нескольких главах.

Регрессия

Регрессия – это статистический метод, который оценивает силу взаимосвязи между зависимой переменной и рядом других переменных. Ее можно использовать для анализа взаимосвязей между переменными. Пример из области финансов – анализ взаимосвязи между ценами на товары и стоимостью акций предприятий, занимающихся производством этих товаров.

Первоначально библиотека pandas включала возможность построения регрессии, но затем она была перенесена в библиотеку StatsModels. Данный факт хорошо иллюстрирует процесс разработки библиотеки pandas. Часто в библиотеки pandas уже есть встроенные методы, но когда они достигают стадии «зрелости», принимается решение, что эффективнее будет перенести их в другие специализированные библиотеки Python. Это и хорошо и плохо. Поначалу здорово иметь такую возможность непосредственно в pandas, но по мере обновления версий такое нагромождение методов может сильно усложнить программный код!

Другие библиотеки Python, работающие вместе с библиотекой pandas

Библиотека pandas представляет собой небольшую, но важную, часть экосистемы анализа данных и науки о данных в Python. Для справки вот еще несколько важных библиотек Python, которые стоит отметить. Список не является исчерпывающим, но включает те библиотеки, с которыми вы, вероятно, столкнетесь.

Численные и научные вычисления – NumPy и SciPy

NumPy (http://www.numpy.org/) – базовый набор инструментов для осуществления научных вычислений с помощью Python, включенный в большинство современных дистрибутивов Python. Это действительно основной инструментарий, на базе которого была создана библиотека pandas, и работая с pandas, вы почти наверняка будете часто использовать NumPy. Среди прочего NumPy обеспечивает поддержку многомерных массивов, базовые операции над ними и полезные функции линейной алгебры.

Использование функций NumPy идет рука об руку с использованием библиотеки pandas, в частности, когда мы работаем с объектом Series. Большинство наших примеров будут ссылаться на NumPy, но функциональность объекта Series – это настолько сложная надстройка над массивом NumPy, в детали которой мы вникать не будем, за исключением лишь некоторых ситуаций.

SciPy (https://www.scipy.org/) предлагает набор численных алгоритмов и инструментарий для конкретных научных областей, включая обработку сигналов, оптимизацию, статистику и многое другое.

Статистический анализ – StatsModels

StatsModels (http://statsmodels.sourceforge.net/) – это модуль Python, который позволяет пользователям исследовать данные, строить статистические модели и выполнять статистические тесты. Широкий выбор описательных статистик, статистических тестов, графических функций доступен для разных типов данных и моделей. Исследователи разных областей могут обнаружить, что StatsModels полностью удовлетворяет их потребности в статистических вычислениях и анализе данных в среде Python.

Функционал StatsModels включает в себя:

- Линейные регрессионные модели
- Обобщенные линейные модели
- Модели дискретного выбора
- Робастные линейные модели
- Различные модели и функции для анализа временных рядов
- Непараметрические оценки
- Коллекции наборов данных в качестве примеров
- Широкий спектр статистических тестов
- Инструменты ввода-вывода для создания таблиц в различных форматах (текст, LaTex, HTML) и для чтения файлов Stata в массивы NumPy и датафреймы pandas
- Графические функции
- Различные модульные тесты для проверки правильности результатов

Машинное обучение – scikit-learn

scikit-learn (http://scikit-learn.org/) – это библиотека машинного обучения, созданная на основе NumPy, SciPy и matplotlib. Она предлагает простые и эффективные инструменты для общих задач анализа данных, таких как классификация, регрессия, кластеризация, снижение размерности, отбор модели и предварительная обработка.

PyMC – стохастическое байесовское моделирование

PyMC (https://github.com/pymc-devs/pymc) – это модуль Python, в котором реализованы байесовские статистические модели и алгоритмы подгонки, включая метод марковских цепей Монте-Карло. Гибкость и расширяемость этого модуля позволяют применить его к широкому кругу задач. Наряду с базовым функционалом – алгоритмами генерирования выборки PyMC включает в себя вывод полученных результатов, построение графиков, проверку качества подгонки и диагностику сходимости.

Визуализация данных – matplotlib и seaborn

Python предлагает богатый выбор фреймворков для визуализации данных. Два самых популярных из них – **matplotlib** и более новый **seaborn**.

Matplotlib

Matplotlib - это библиотека Python для построения 2D-графиков, которая позволяет создавать высококачественные диаграммы в различных форматах и интерактивные окружения на разных платформах. Matplotlib можно использовать в скриптах Python, оболочке Python и IPython, Jupyter Notebook, серверах веб-приложений и еще в четырех инструментах, предлагающий графический пользовательский интерфейс.

Библиотека pandas очень тесно интегрирована с библиотекой matplotlib, включая методы объектов `Series` и `DataFrame`, которые позволяют автоматически вызвать библиотеку matplotlib. Это не означает, что возможности визуализации в библиотеке pandas ограничиваются только библиотекой matplotlib. Как мы увидим, можно легко воспользоваться и другими библиотеками, например, ggplot2 и seaborn.

Seaborn

Seaborn (http://seaborn.pydata.org/introduction.html) – это библиотека для создания привлекательных и информативных статистических графиков в среде Python. Она надстраивается поверх библиотеки matplotlib и тесно интегрирована со стеком PyData, включая поддержку NumPy, объектов pandas и статистических процедур SciPy и StatsModels. Она обеспечивает дополнительную функциональность, выходящую за

пределы библиотеки matplotlib, а также по умолчанию демонстрирует более богатые и более современные способы визуализации, чем matplotlib.

Выводы

В этой главе мы продолжили наше знакомство с библиотекой pandas, обработкой данных/анализом и наукой. Оно началось с обсуждения причин появления библиотеки pandas, ее функционала и ее взаимосвязи с такими понятиями, как обработка данных, анализ и наука о данных.

Затем мы рассмотрели процесс анализа данных, чтобы понять предназначение некоторых функций библиотеки pandas. Они включают в себя сбор, организацию, чистку и исследование данных, а затем построение формальной модели, представление результатов, а также возможность совместного использования и воспроизведения результатов анализа.

Затем мы рассмотрели несколько понятий, связанных с данными и статистическим моделированием. Мы рассмотрели общие методы анализа и понятия, чтобы познакомить вас с ними перед их подробным изучением в последующих главах.

Кроме того, pandas является частью более обширной экосистемы Python, предлагающей массу библиотек, полезных для анализа данных и науки. Хотя эта книга будет сосредоточена только на библиотеке pandas, есть и другие библиотеки, с которыми вы столкнетесь, и поэтому мы рассказали о них для общего знакомства.

Итак, мы готовы к использованию библиотеки pandas. В следующей главе прежде чем углубляться в pandas мы разберем ее основы, начав с окружения Python и pandas, обзора тетрадок Jupyter и закончив кратким знакомством с объектами библиотеки pandas `Series` и `DataFrame`.

ГЛАВА 2 ЗАПУСК БИБЛИОТЕКИ PANDAS

В этой главе мы расскажем, как установить библиотеку pandas и начать использовать ее основные функции. Содержание книги представлено в виде тетрадок IPython и Jupyter, поэтому мы также быстро рассмотрим использование этих двух инструментов. В данной книге будет использоваться научный дистрибутив Anaconda Python от Continuum. Anaconda – популярный дистрибутив Python с бесплатными и платными компонентами. Anaconda обеспечивает кросс-платформенную поддержку, включая Windows, Mac и Linux. Базовый дистрибутив Anaconda включает pandas, IPython и Jupyter Notebook, что позволяет без каких-либо сложностей начать работу. В этой главе будут рассмотрены следующие темы:

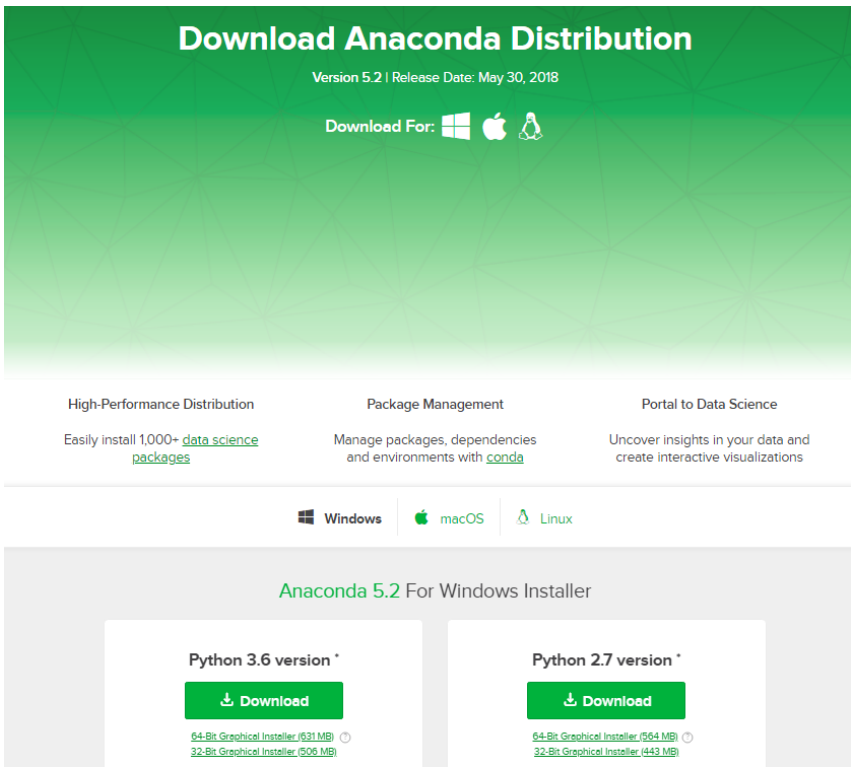
- Установка Anaconda, pandas и IPython/ Jupyter Notebook
- Использование IPython и Jupyter Notebook
- Jupyter и его тетрадки
- Настройка pandas
- Краткое знакомство со структурами данных Series и DataFrame библиотеки pandas
- Загрузка данных из CSV файла
- Визуализация данных в pandas

Установка Anaconda

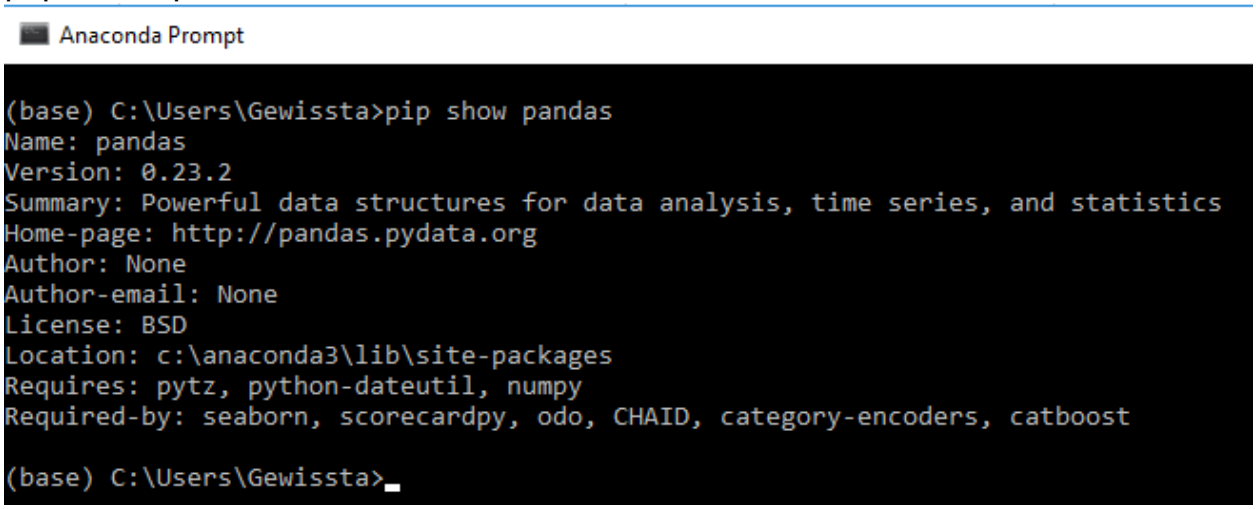
В этой книге будет использоваться Anaconda Python 3, в частности, версия 3.6.1. На момент написания книги была доступна библиотека pandas версии 0.20.2. По умолчанию инсталлятор Anaconda установит Python, IPython, Jupyter Notebook и pandas.

Anaconda Python можно загрузить с веб-сайта Continuum Analytics по адресу http://continuum.io/downloads. Веб-сервер определит операционную систему вашего браузера и предоставит вам соответствующий вашей системе файл загрузки.

При открытии этого URL-адреса в вашем браузере вы увидите страницу примерно следующего вида:



Загрузите инсталлятор для версии 3.6. Текущая версия Anaconda, которая будет использоваться в этой книге – 5.2 с Python 3.6. Запустите инсталлятор и установите Anaconda Python. В Anaconda Prompt вы можете проверить установленную версию pandas с помощью команды `pip show pandas`:



На момент написания книги текущая версия pandas была 0.23. Убедитесь, что вы используете версию 0.23 или новее, поскольку в данной версии появились некоторые изменения, которые будут в дальнейшем использоваться в книге.

Теперь, когда у нас установлено все необходимое, давайте перейдем к использованию IPython и Jupyter Notebook.

IPython и Jupyter Notebook

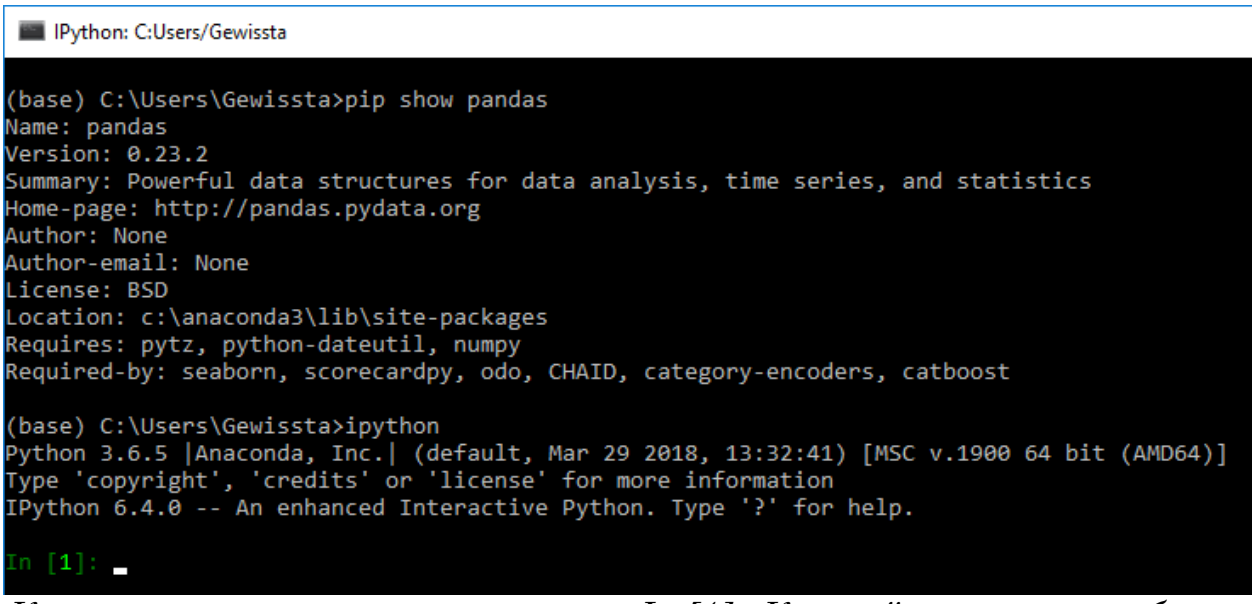
До сих пор мы запускали программный код Python из командной строки или терминала. Это форма организации простой интерактивной среды программирования в рамках средств интерфейса командной строки (REPL, от англ. *read-eval-print loop — цикл «чтение — вычисление — вывод»*)), которая поставляется вместе с Python. Ее можно использовать для запуска всех примеров в этой книге, однако книга будет использовать IPython и Jupyter Notebook. Давайте кратко рассмотрим оба инструмента.

IPython

IPython – это альтернативная оболочка для интерактивной работы с Python. Она предлагает несколько усовершенствований для REPL, поставляемой по умолчанию.

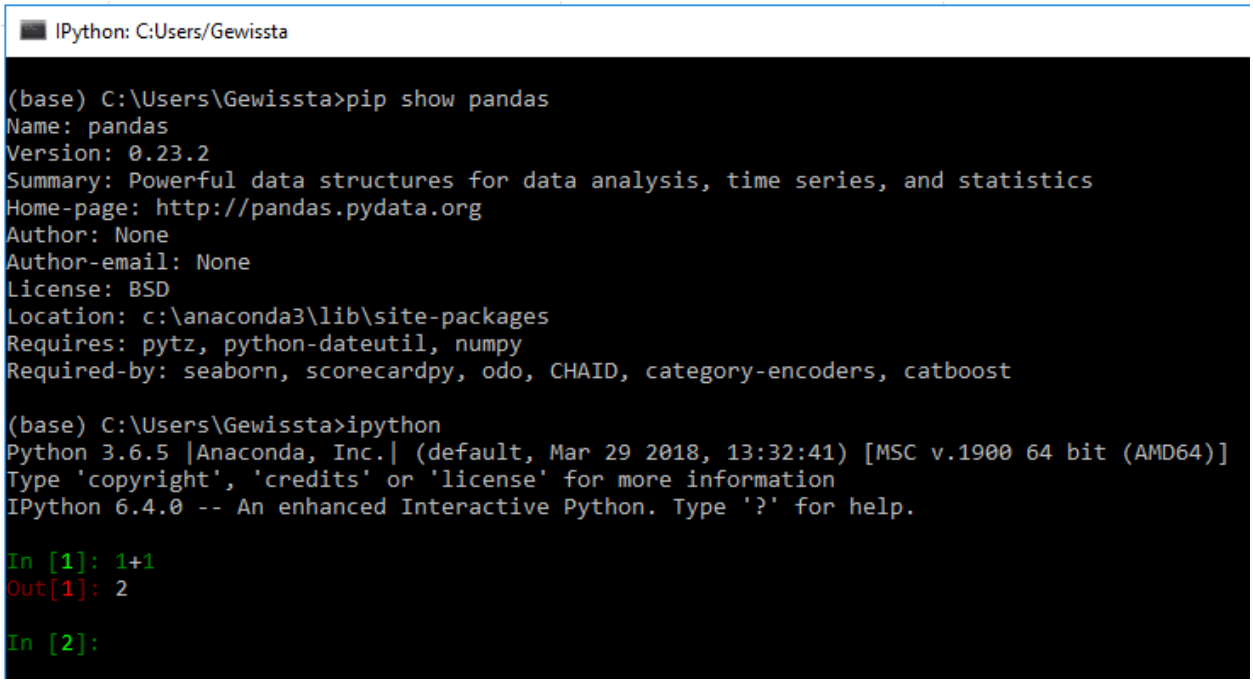
Если вы хотите более подробно узнать о IPython, ознакомьтесь с документацией по адресу https://ipython.org/ipython-doc/3/interactive/tutorial.html.

Чтобы запустить IPython, просто выполните команду `ipython` из командной строки/терминала. Вы увидите следующий результат:



Командная строка ввода показывает In [1]:. Каждый раз, когда вы будете вводить инструкцию в REPL IPython, число в командной строке будет увеличиваться.

Аналогично, вывод для какой-либо конкретной записи будет предваряться `Out[x]:`, где `x` соответствует номеру `In[x]:`. Следующий скриншот демонстрирует это:



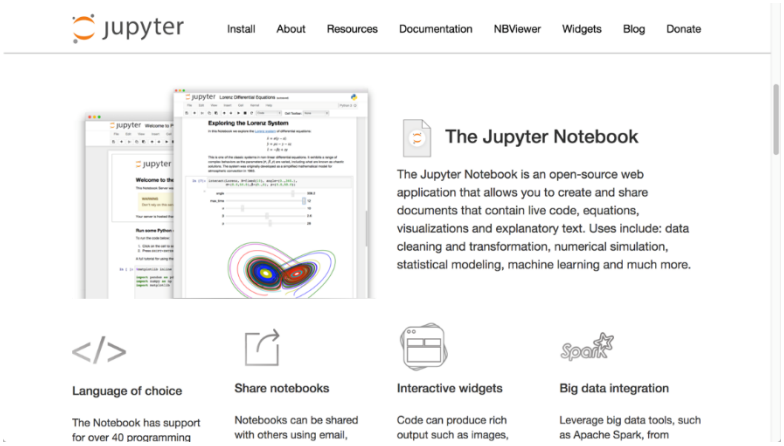
Данная нумерация операций ввода и выхода будет важна для примеров, поскольку все примеры будут предваряться `In[x]:` и `Out[x]:` и таким образом можно будет отследить последовательность выполнения операций.

Обратите внимание, что эти числа являются строго последовательными. Если вы запускаете программный код и при вводе возникают ошибки или вы вводите дополнительные инструкции, нумерация перестанет быть последовательной (ее можно сбросить, выйдя и перезапустив IPython). Пожалуйста, используйте нумерацию в качестве ориентира.

Jupyter Notebook

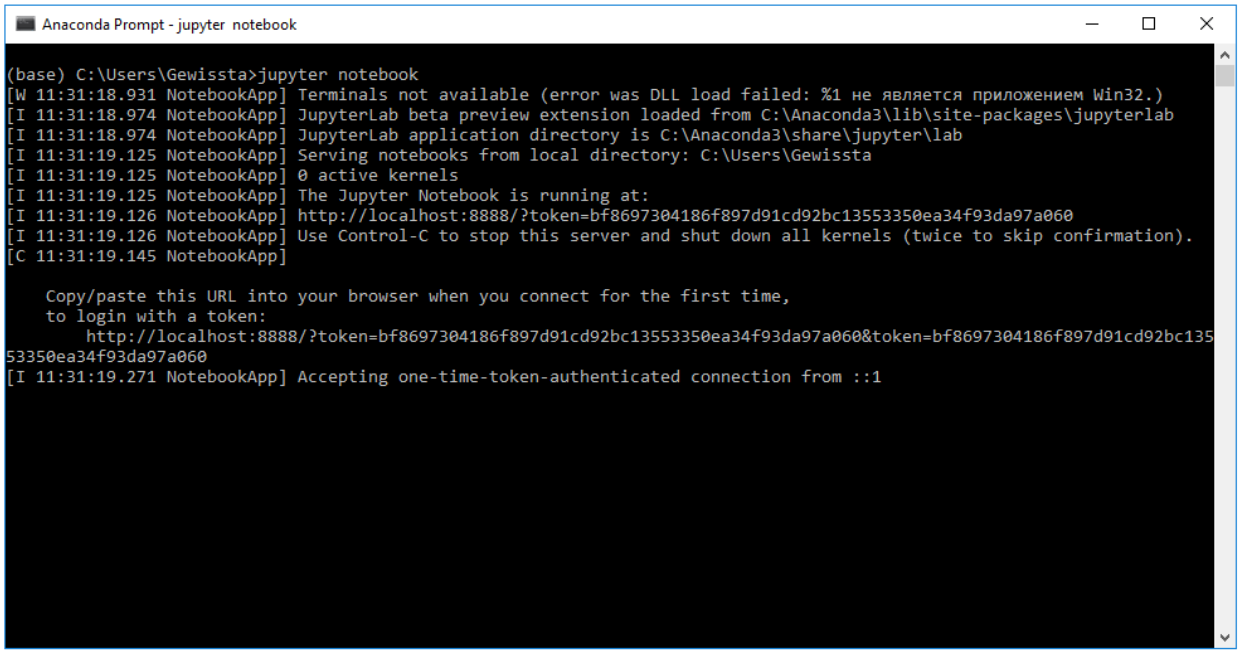
Jupyter Notebook – это результат эволюции IPython Notebook. Это веб-приложение с открытым исходным кодом, которое позволяет создавать и обмениваться документами, содержащими живой код, уравнения, визуализацию и разметку.

Первоначально IPython Notebook ограничивался лишь Python в качестве единственного языка. Jupyter Notebook позволил использовать многие языки программирования, включая Python, R, Julia, Scala и F#. Если вы хотите глубже познакомиться с Jupyter Notebook, перейдите на http://jupyter.org/, где вы увидите страницу следующего вида:

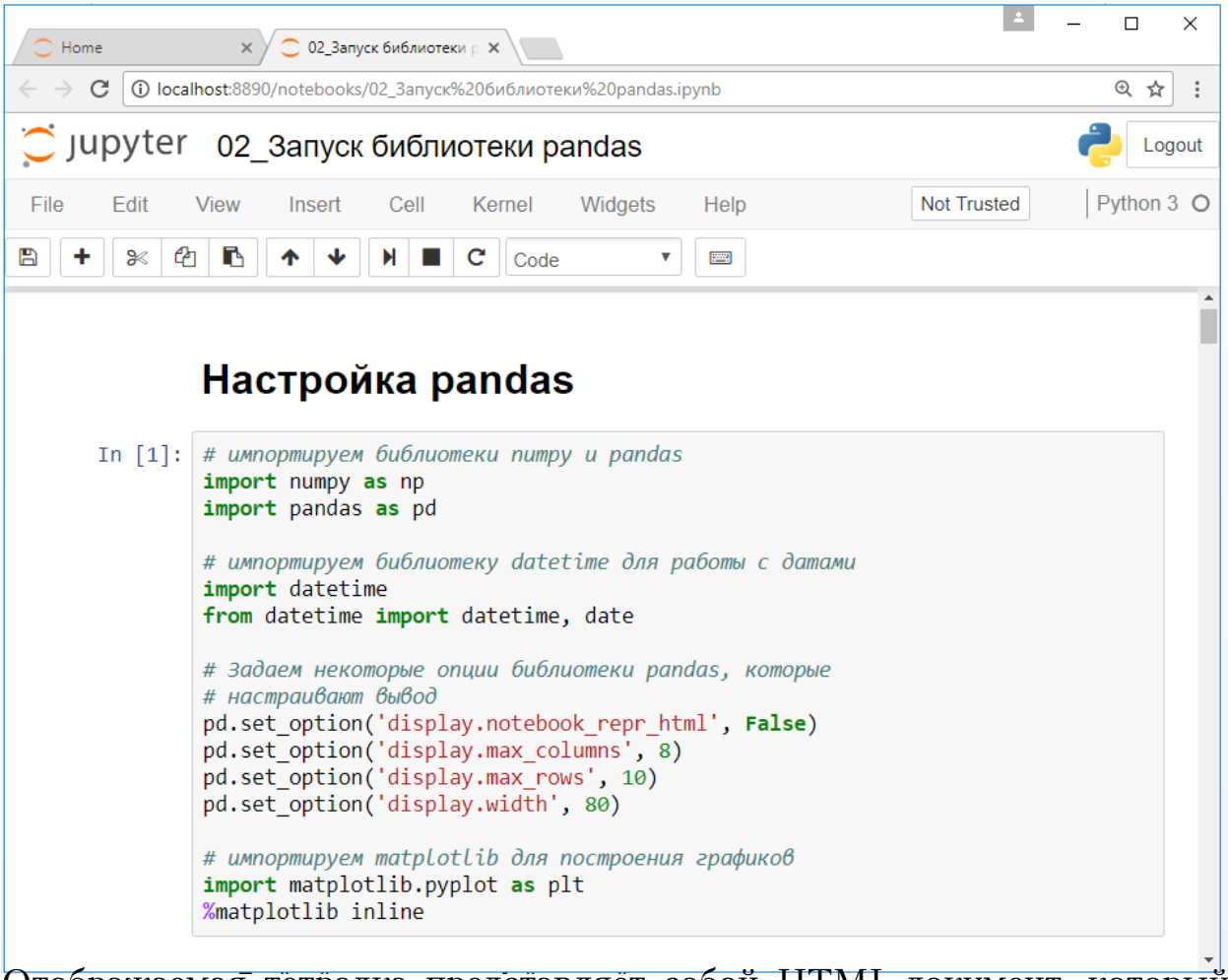


Jupyter Notebook можно скачать и использовать независимо от Python. Anaconda устанавливает его по умолчанию. Чтобы запустить Jupyter Notebook, введите в Anaconda Prompt следующую команду:

```
jupyter notebook
```



Откроется страница браузера, отображающая домашнюю страницу Jupyter Notebook (http://localhost:8888/tree).
Если щелкнуть по файлу с расширением .ipynb, откроется страница с тетрадкой. Если вы откроете тетрадку для этой главы, то увидите содержимое следующего вида:



Отображаемая тетрадка представляет собой HTML-документ, который был создан Jupyter и IPython. Он состоит из нескольких ячеек, которые могут быть одного из четырех типов: code, markdown, nbconvert и heading. Все примеры в этой книге используют либо ячейки code, либо ячейки markdown.

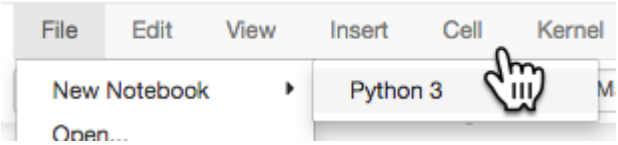
Jupyter запускает ядро IPython для каждой тетрадки. Ячейки, содержащие код Python, выполняются внутри этого ядра и результаты добавляются в тетрадку в формате HTML.

Двойной щелчок по любой из этой ячеек позволит отредактировать ее. По завершении редактирования содержимого ячейки, нажмите *Shift + Enter*, после чего Jupyter/IPython проанализирует содержимое и отобразит результаты.

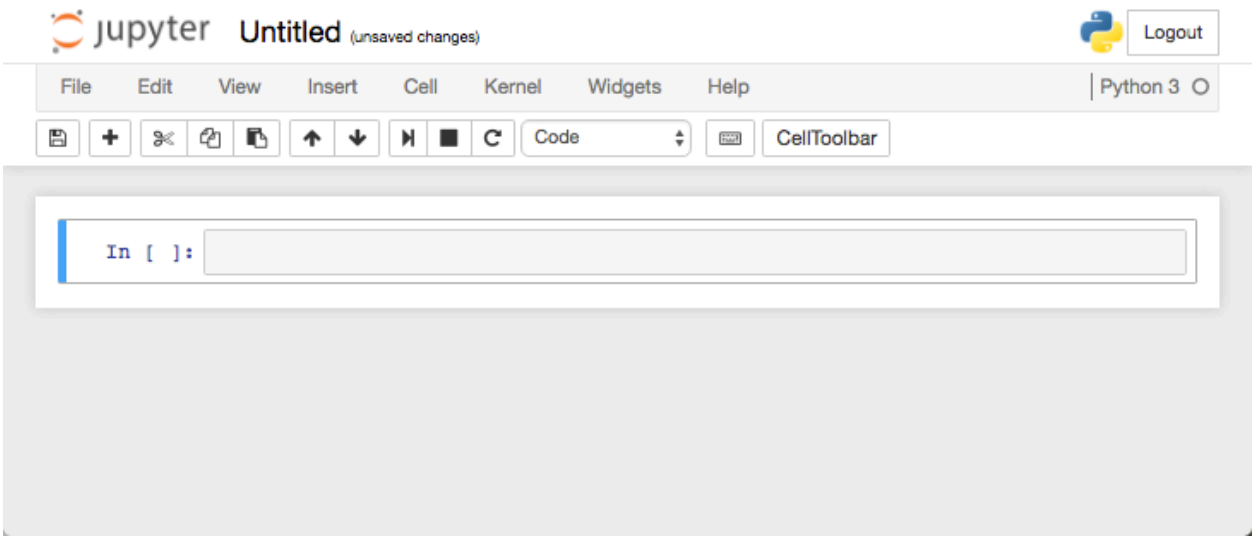
Если вы хотите узнать больше о формате тетрадки, лежащего в основе страниц, см. https: //ipython.org/ipython-doc/3/notebook/nbformat.html.

Панель инструментов в верхней части браузера предоставляет ряд возможностей по работе с тетрадкой. К ним относятся добавление, удаление и перемещение ячеек вверх и вниз в тетрадке. Также доступны команды для запуска ячеек, перезапуска ячеек и перезапуска основного ядра IPython.

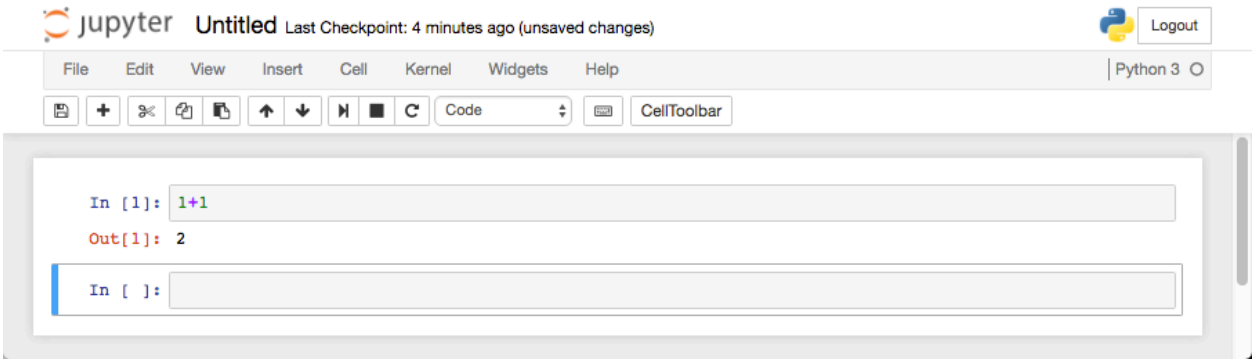
Чтобы создать новую тетрадку, перейдите в меню File | New Notebook | Python 3:



Страница новой тетрадки будет создана в новой вкладке браузера. Ее имя по умолчанию будет Untitled.



Тетрадка состоит из одной ячейки code, которая готова к вводу программного кода Python. Введите 1 + 1 в ячейку и нажмите *Shift* + *Enter* для выполнения.



Ячейка выполнена и результат показан как `Out[1]:`. Jupyter также создал новую ячейку, чтобы вы могли снова ввести код или разметку.
Jupyter Notebook автоматически сохраняет ваши изменения каждую минуту, но все равно неплохо бы сохранять результаты вручную время от времени.

Знакомство со структурами данных библиотеки pandas – Series и DataFrame

Давайте перейдем к использованию библиотеки pandas, познакомившись с двумя основными структурами данных – `Series` и `DataFrame`. Мы рассмотрим следующие задачи:

- Импорт библиотеки pandas
- Создание объекта `Series` и работа с ним
- Создание объекта `DataFrame` и работа с ним
- Загрузка данных из файла в `DataFrame`

Импорт pandas

Каждая используемая тетрадка начинается с импорта pandas и еще нескольких полезных питоновских библиотек. Кроме того, в тетрадке используется несколько опций для настройки вывода pandas в Jupyter Notebook. Первый блок программного кода имеет следующий вид:

```
In[1]:
# импортируем библиотеки numpy и pandas
import numpy as np
import pandas as pd

# импортируем библиотеку datetime для работы с датами
import datetime
from datetime import datetime, date

# Задаем некоторые опции библиотеки pandas, которые
# настраивают вывод
pd.set_option('display.notebook_repr_html', False)
pd.set_option('display.max_columns', 8)
pd.set_option('display.max_rows', 10)
pd.set_option('display.width', 80)

# импортируем библиотеку matplotlib для построения графиков
import matplotlib.pyplot as plt
%matplotlib inline
```

Первая инструкция импортирует библиотеку NumPy и ссылается на элементы библиотеки с помощью префикса `np.`. Мы не будем подробно останавливаться на NumPy в этой книге, однако в ряде случаев нам эта библиотека пригодится.

Вторая инструкция импортирует библиотеку pandas. Мы будем ссылаться на элементы в библиотеке с помощью префикса `pd.`. Инструкция `from pandas import Series, DataFrame` явно импортирует объекты `Series` и `DataFrame` в глобальное пространство имен. Это позволяет нам ссылаться на объекты `Series` и `DataFrame` без префикса `pd.`. Это удобно, поскольку мы будем использовать их довольно часто и это сэкономит время, затрачиваемое на ввод программного кода.

Инструкция `import datetime` ссылается на библиотеку `datetime`, которая обычно используется в pandas для анализа временных рядов. Она будет включен в импорт для каждой тетрадки.

Функция `pd.set_option()` вызывает настройки параметров, которые настраивают вывод в pandas. Первая функция задает вывод `Series` и `DataFrame` в виде текста, а не HTML. Следующие две функции указывают максимальное количество выводимых столбцов и строк. Последняя функция задает максимальное количество символов в каждой строке.

Вы можете просмотреть дополнительные опции по следующему адресу: http://pandas.pydata.org/pandas-docs/stable/options.html.

Внимательный читатель может заметить, что данная ячейка не имеет `Out[x]:`. Не все ячейки (или команды IPython) будут генерировать вывод.

Если вы хотите использовать IPython вместо Jupyter Notebook, вы также можете выполнить этот код в оболочке IPython. Например, вы можете просто вырезать и вставить код из ячейки тетрадки.

Оболочка IPython достаточно умна, чтобы понять, что вы вставляете несколько строк и соответственно создает отступы. И обратите внимание, что в оболочке IPython тоже нет `Out[x]:`. `pd.set_option` не возвращает никакого содержимого и, следовательно, нет никакой аннотации.

Объект Series

`Series` – это базовая структура данных в библиотеке pandas. Серия похожа на массив NumPy, но отличается от него тем, что имеет индекс, который позволяет осуществить гораздо более гибкий поиск элементов, а не только найти значение индекса массива, в котором отсчет начинается с нуля.

Программный код, приведенный ниже, создает серию из питоновского списка:

```
In[2]:
# создаем объект Series, состоящий
# из четырех элементов
s = pd.Series([1, 2, 3, 4])
s

Out[2]:
0    1
1    2
2    3
3    4
dtype: int64
```

Вывод состоит из двух столбцов информации. Первый – это индекс, а второй – данные в объекте `Series`. Каждая строка вывода представляет собой **метку** индекса (в первом столбце), а затем следует значение, связанное с этой меткой. Поскольку эта серия была создана без указания индекса, pandas автоматически создает целочисленный индекс с метками, начинающимися с 0 и увеличивающимися на единицу для каждого элемента данных.

Значения объекта `Series` можно посмотреть с помощью оператора `[]`, принимающего метку за требуемое значение. Следующий программный код извлекает значение для метки `1`:

```
In[3]:
# получаем значение для метки индекса 1
s[1]

Out[3]:
2
```

Это очень похоже на работу с обычным массивом в различных языках программирования. Однако, как мы потом увидим, индекс не обязательно должен начинаться с 0 и увеличиваться на единицу и могут использовать другие типы данных, а не только целые числа. Эта способность создавать гибкие индексы подобным образом является одной из замечательных возможностей pandas.
Можно извлечь несколько элементов, указав их метки в питоновском списке. Следующий программный код извлекает значения для меток `1` и `3`:

```
In[4]:
# возвращаем серию с элементами,
# у которых метки 1 и 3
s[[1, 3]]

Out[4]:
1    2
3    4
dtype: int64
```

Объект `Series` можно создать с помощью пользовательского индекса, воспользовавшись параметром `index` и указав метки индекса. Следующий программный код создает объект `Series` с теми же самыми значениями, однако метки индекса будут строковыми значениями:

```
In[5]:
# создаем серию, задав индекс в явном виде
s = pd.Series([1, 2, 3, 4],
              index = ['a', 'b', 'c', 'd'])
s

Out[5]:
a    1
b    2
c    3
d    4
dtype: int64
```

Теперь данные в объекте `Series` можно извлечь с помощью этих буквенно-цифровых меток индекса. Следующий программный код извлекает значения в метках индекса `'a'` и `'d'`:

```
In[6]:
# ищем элементы серии, у которых
# метки индекса 'a' и 'd'
s[['a', 'd']]
```

```
Out[6]:
a    1
d    4
dtype: int64
```

По-прежнему можно сослаться на элементы этого объекта `Series` с помощью их численной позиции с началом отсчета в 0:

```
In[7]:
# передаем список целочисленных значений в объект Series,
# у которого метки индекса записаны в виде букв,
# поиск буквенных меток будет осуществлен на основе
# числового индекса, начинающегося с 0,
# как если бы мы использовали обычный массив
s[[1, 2]]
```

```
Out[7]:
b    2
c    3
dtype: int64
```

Мы можем взглянуть на индекс объекта `Series` с помощью свойства `.index`:

```
In[8]:
# извлекаем только индекс объекта Series
s.index
```

```
Out[8]:
Index(['a', 'b', 'c', 'd'], dtype='object')
```

Индекс сам по себе является объектом библиотеки pandas, и этот вывод показывает нам значения индекса, а также тип данных, используемый для индекса. В данном случае обратите внимание, что тип данных в индексе (называемый `dtype`) является объектом, а не строкой. Позже мы рассмотрим, как это можно изменить.
Обычное использование объекта `Series` в pandas заключается в том, чтобы записать временной ряд, который свяжет метки индекса, записанные в виде дат и времени, со значениями. Следующий программный код как раз демонстрирует это, создавая диапазон дат с помощью функции `pd.date_range()` библиотеки pandas:

```
In[9]:
# создаем объект Series, индекс которого - серия дат
# между двумя определенными датами (включительно)
dates = pd.date_range('2016-04-01', '2016-04-06')
dates
```

```
Out[9]:
DatetimeIndex(['2016-04-01', '2016-04-02', '2016-04-03', '2016-04-04',
               '2016-04-05', '2016-04-06'],
              dtype='datetime64[ns]', freq='D')
```

Данный программный код создает специальный индекс `DatetimeIndex`, который является специализированным типом индекса библиотеки pandas, предназначенным для индексации данных, содержащих даты и время.
Теперь давайте создадим объект `Series`, используя этот индекс. Значения данных представляют собой высокие температуры (по Фаренгейту) в определенные дни:

```
In[10]:
# создаем объект Series, в котором значениям температуры
# будут соответствовать даты в индексе
temps1 = pd.Series([80, 82, 85, 90, 83, 87],
                   index = dates)
temps1

Out[10]:
2016-04-01    80
2016-04-02    82
2016-04-03    85
2016-04-04    90
2016-04-05    83
2016-04-06    87
Freq: D, dtype: int64
```

Данный тип серии с `DateTimeIndex` называется временным рядом.
Мы можем найти температуру для определенной даты, использовав дату в качестве строки:

```
In[11]:
# какая температура была 2016-04-04?
temps1['2016-04-04']

Out[11]:
90
```

Над объектами `Series` можно совершать арифметические операции. Следующий код создает второй объект `Series` и вычисляет разницу температур обеих серий:

```
In[12]:
# создаем вторую серию значений, используя
# тот же самый индекс
temps2 = pd.Series([70, 75, 69, 83, 79, 77],
                   index = dates)
# следующий программный код выравнивает
# обе серии по меткам индекса и вычисляет
# разницу температур в этих метках
temp_diffs = temps1 - temps2
temp_diffs

Out[12]:
2016-04-01    10
2016-04-02     7
2016-04-03    16
2016-04-04     7
2016-04-05     4
2016-04-06    10
Freq: D, dtype: int64
```

*Результат арифметической операции (+, -, /, *, ...) над двумя объектами Series, которые не являются скалярными значениями, возвращает еще один объект Series.*
Поскольку индекс не является целым числом, мы также можем найти значение температуры с помощью численной позиции, у которой отсчет начинается с 0.

In[13]:
```
# вычислим разницу температур для определенной числовой
# метки индекса, как если бы серия была массивом
temp_diffs[2]
```

Out[13]:
```
16
```

Наконец, библиотека pandas позволяет вычислить различные описательные статистики. Например, следующий программный код возвращает среднее значение разности температур:

In[14]:
```
# вычисляем среднее значение разности температур
temp_diffs.mean()
```

Out[14]:
```
9.0
```

Объект DataFrame

Объект `Series` библиотеки pandas может иметь только один столбец значений, связанных с метками индекса. Чтобы задать несколько столбцов значений для меток индекса, мы можем воспользоваться датафреймом. Датафрейм представляет собой один или несколько объектов `Series`, упорядоченных по меткам индекса. Каждая серия будет столбцом в датафрейме и каждый столбец может иметь соответствующее имя.
В некотором роде датафрейм аналогичен таблице реляционной базы данных, поскольку он содержит один или несколько столбцов данных различных типов (однако внутри каждого столбца элементы должны быть одного типа).
Следующий программный код создает объект `DataFrame` с двумя столбцами и использует значения температур в объектах `Series`:

In[15]:
```
# создаем объект DataFrame с двумя сериями temp1 и temp2
# и присваиваем им имена столбцов
temps_df = pd.DataFrame(
            {'Missoula': temps1,
             'Philadelphia': temps2})
temps_df
```

Out[15]:

```
            Missoula  Philadelphia
2016-04-01        80            70
2016-04-02        82            75
2016-04-03        85            69
2016-04-04        90            83
2016-04-05        83            79
2016-04-06        87            77
```

Итоговый датафрейм имеет два столбца с именами `Missoula` и `Philadelphia`. Эти столбцы представляют собой новые объекты `Series` в датафрейме со значениями, скопированными из исходных объектов `Series`.

Столбцы в объекте `DataFrame` можно вывести с помощью индексатора массива `[]`, задав имя столбца или список имен столбцов. Следующий программный код извлекает столбец `Missoula`:

In[16]:

```
# получаем столбец Missoula
temps_df['Missoula']
```

Out[16]:

```
2016-04-01    80
2016-04-02    82
2016-04-03    85
2016-04-04    90
2016-04-05    83
2016-04-06    87
Freq: D, Name: Missoula, dtype: int64
```

А программный код, приведенный ниже, извлекает столбец `Philadelphia`:

In[17]:

```
# аналогичным образом мы можем получить
# столбец Philadelphia
temps_df['Philadelphia']
```

Out[17]:

```
2016-04-01    70
2016-04-02    75
2016-04-03    69
2016-04-04    83
2016-04-05    79
2016-04-06    77
Freq: D, Name: Philadelphia, dtype: int64
```

Для вывода нескольких столбцов можно еще воспользоваться питоновским списком из имен столбцов:

In[18]:

```
# выводим оба столбца
temps_df[['Philadelphia', 'Missoula']]
```

Out[18]:

```
            Philadelphia  Missoula
2016-04-01            70        80
2016-04-02            75        82
2016-04-03            69        85
2016-04-04            83        90
2016-04-05            79        83
2016-04-06            77        87
```

Существует тонкая разница между объектом `DataFrame` *и объектом* `Series`*. Передав список в оператор* `[]` *объекта* `DataFrame`*, мы извлекаем указанные столбцы, тогда как объект* `Series` *будет возвращать строки.* Если имя столбца не имеет пробелов, его можно вывести, воспользовавшись «вычисляемым» свойством:

In[19]:
```
# извлекаем столбец Missoula с помощью
# "вычисляемого" свойства
temps_df.Missoula
```

Out[19]:
```
2016-04-01    80
2016-04-02    82
2016-04-03    85
2016-04-04    90
2016-04-05    83
2016-04-06    87
Freq: D, Name: Missoula, dtype: int64
```

Арифметические операции со столбцами в датафрейме идентичны операциям со столбцами, представленными в виде объектов `Series`. Для иллюстрации следующий программный код вычисляет разность температур с помощью «вычисляемого» свойства:

In[20]:
```
# вычисляем разницу температур двух городов
temps_df.Missoula - temps_df.Philadelphia
```

Out[20]:
```
2016-04-01    10
2016-04-02     7
2016-04-03    16
2016-04-04     7
2016-04-05     4
2016-04-06    10
Freq: D, dtype: int64
```

Новый столбец можно добавить в объект `DataFrame`, присвоив еще одну серию столбцу с помощью индексатора массива `[]`. Следующий программный код добавляет в `DataFrame` новый столбец, содержащий разности температур:

In[21]:
```
# добавляем в датафрейм temp_df столбец Difference,
# который содержит разности температур
temps_df['Difference'] = temp_diffs
temps_df
```

Out[21]:

	Missoula	Philadelphia	Difference
2016-04-01	80	70	10
2016-04-02	82	75	7
2016-04-03	85	69	16
2016-04-04	90	83	7
2016-04-05	83	79	4
2016-04-06	87	77	10

Имена столбцов в объекте `DataFrame` можно получить с помощью свойства `.columns`:

```
In[22]:
# получаем имена столбцов, которые
# к тому же являются объектом Index
temps_df.columns
```

```
Out[22]:
Index(['Missoula', 'Philadelphia', 'Difference'], dtype='object')
```

Можно создать «срез» (slice) объекта `DataFrame` и `Series`, чтобы извлечь определенные строки. Следующий программный код отбирает значения разностей температур со второй по четвертую строки:

```
In[23]:
# отбираем значения разностей температур в столбце Difference, начиная
# с позиции 1 (2016-04-02) и заканчивая позицией 4 (2016-04-05),
# как если бы это был массив
# обратите внимание, что конец диапазона не включается, то есть
# диапазон не включает элемент с позицией 4 (2016-04-05)
temps_df.Difference[1:4]
```

```
Out[23]:
2016-04-02     7
2016-04-03    16
2016-04-04     7
Freq: D, Name: Difference, dtype: int64
```

Полностью строки из датафрейма можно извлечь с помощью свойств `.loc` и `.iloc`. Свойство `.loc` осуществляет поиск по метке индекса, тогда как свойство `.iloc` осуществляет поиск по позиции, начинающейся с 0. Следующий программный код извлекает вторую строку датафрейма:

```
In[24]:
# получаем строку, которой соответствует
# метка индекса 1
temps_df.iloc[1]
```

```
Out[24]:
Missoula        82
Philadelphia    75
Difference       7
Name: 2016-04-02 00:00:00, dtype: int64
```

Обратите внимание, что данный программный код преобразовал строку в объект `Series` с именами столбцов датафрейма, которые, «повернувшись», стали теперь метками индекса полученной серии. Ниже показан итоговый индекс полученного результата:

```
In[25]:
# имена столбцов стали индексом,
# мы "повернули" их
temps_df.iloc[1].index
```

```
Out[25]:
Index(['Missoula', 'Philadelphia', 'Difference'], dtype='object')
```

Строки можно вывести явно с помощью метки индекса, воспользовавшись свойством `.loc`. Следующий программный код извлекает строку с помощью метки индекса:

```
In[26]:
# извлекаем строку с помощью метки индекса,
# воспользовавшись свойством .loc
```

```
temps_df.loc['2016-04-05']
```

```
Out[26]:
Missoula        83
Philadelphia    79
Difference       4
Name: 2016-04-05 00:00:00, dtype: int64
```

Конкретные строки в объекте `DataFrame` можно выбрать, используя список целочисленных позиций. Следующий программный код выбирает значения из столбца `Difference` в строках, соответствующих целочисленным позициям `1`, `3` и `5`:

```
In[27]:
# получаем значения столбца Difference в строках 1, 3 и 5,
# используя целочисленные позиции с началом отсчета в 0
temps_df.iloc[[1, 3, 5]].Difference
```

```
Out[27]:
2016-04-02     7
2016-04-04     7
2016-04-06    10
Freq: 2D, Name: Difference, dtype: int64
```

Строки датафрейма можно отобрать с помощью логического выражения, которое применяется к каждой строке данных. Ниже показаны значения температуры в столбце `Missoula`, которые превышают 82 градуса:

```
In[28]:
# какие значения в столбце Missoula > 82?
temps_df.Missoula > 82
```

```
Out[28]:
2016-04-01    False
2016-04-02    False
2016-04-03     True
2016-04-04     True
2016-04-05     True
2016-04-06     True
Freq: D, Name: Missoula, dtype: bool
```

Затем результаты выражения помещаем внутри квадратных скобок оператора датафрейма (и серии) `[]`, в итоге будут возвращены только те строки, в которых выражение принимает значение `True`:

```
In[29]:
# возвращаем строки, в которых значения температуры
# для столбца Missoula > 82
temps_df[temps_df.Missoula > 82]
```

```
Out[29]:
            Missoula  Philadelphia  Difference
2016-04-03        85            69          16
2016-04-04        90            83           7
2016-04-05        83            79           4
2016-04-06        87            77          10
```

В терминологии pandas этот метод называется **логический отбор** или **отбор на основе булевых значений (Boolean Selection)** и он будет основным способом отбора строк на основе значений конкретных столбцов (аналогично запросу в SQL с использованием условия `WHERE`, но, как мы увидим, логический отбор является намного более мощным способом).

Загрузка данных из CSV-файла в объект DataFrame

Библиотека pandas позволяет легко загрузить данные из различных источников данных в объекты pandas. В качестве краткого примера давайте рассмотрим возможность pandas загружать данные в формате CSV. В этом примере будет использоваться файл, который поставляется вместе с программным кодом этой книги, Data/goog.csv, а содержимое файла представляет собой временные ряды цен акций Google.
Эти данные можно легко импортировать в объект `DataFrame` с помощью функции `pd.read_csv()`:

```
In[30]:
# считываем содержимое файла в объект DataFrame
df = pd.read_csv('Data/goog.csv')
df

Out[30]:
          Date        Open        High         Low       Close   Volume
0   12/19/2016  790.219971  797.659973  786.270020  794.200012  1225900
1   12/20/2016  796.760010  798.650024  793.270020  796.419983   925100
2   12/21/2016  795.840027  796.676025  787.099976  794.559998  1208700
3   12/22/2016  792.359985  793.320007  788.580017  791.260010   969100
4   12/23/2016  790.900024  792.739990  787.280029  789.909973   623400
..         ...         ...         ...         ...         ...      ...
56   3/13/2017  844.000000  848.684998  843.250000  845.539978  1149500
57   3/14/2017  843.640015  847.239990  840.799988  845.619995   779900
58   3/15/2017  847.590027  848.630005  840.770020  847.200012  1379600
59   3/16/2017  849.030029  850.849976  846.130005  848.780029   970400
60   3/17/2017  851.609985  853.400024  847.109985  852.119995  1712300

[61 rows x 6 columns]
```

Библиотека pandas не знает, что первый столбец в файле – это даты, и поэтому она обработала содержимое поля `Date` как строки. Это можно проверить, воспользовавшись нижеприведенной инструкцией pandas. Она показывает, что столбец `Date` имеет строковый тип данных:

```
In[31]:
# выводим содержимое столбца с датами
df.Date

Out[31]:
0     12/19/2016
1     12/20/2016
```

```
2     12/21/2016
3     12/22/2016
4     12/23/2016
        ...
56     3/13/2017
57     3/14/2017
58     3/15/2017
59     3/16/2017
60     3/17/2017
Name: Date, Length: 61, dtype: object
```

In[32]:
```
# мы можем получить первое значение
# в столбце с датами
df.Date[0]
```

Out[32]:
```
'12/19/2016'
```

In[33]:
```
# оно является строкой
type(df.Date[0])
```

Out[33]:
```
str
```

Параметр `parse_dates` функции `pd.read_csv()` указывает pandas, как нужно преобразовать данные непосредственно в объект библиотеки pandas, предназначенный для хранения дат. Следующий программный код преобразовывает содержимое столбца `Date` в фактический объект `TimeStamp`:

In[34]:
```
# считываем данные и указываем библиотеке pandas,
# что в итоговом датафрейме значения столбца
# с датами должны быть фактическими датами
df = pd.read_csv('Data/goog.csv', parse_dates=['Date'])
df
```

Out[34]:

	Date	Open	High	Low	Close	Volume
0	2016-12-19	790.219971	797.659973	786.270020	794.200012	1225900
1	2016-12-20	796.760010	798.650024	793.270020	796.419983	925100
2	2016-12-21	795.840027	796.676025	787.099976	794.559998	1208700
3	2016-12-22	792.359985	793.320007	788.580017	791.260010	969100
4	2016-12-23	790.900024	792.739990	787.280029	789.909973	623400
..	...	...	...	...	...	...
56	2017-03-13	844.000000	848.684998	843.250000	845.539978	1149500
57	2017-03-14	843.640015	847.239990	840.799988	845.619995	779900
58	2017-03-15	847.590027	848.630005	840.770020	847.200012	1379600
59	2017-03-16	849.030029	850.849976	846.130005	848.780029	970400
60	2017-03-17	851.609985	853.400024	847.109985	852.119995	1712300

```
[61 rows x 6 columns]
```

Если мы захотим проверить, сработал ли этот программный код, то увидим, что даты теперь имеют тип данных `Timestamp`:

In[35]:
```
# проверяем, являются ли сейчас значения столбца
# Date датами, то есть фактически им присвоен
# тип Timestamp
type(df.Date[0])
```

Out[35]:
```
pandas._libs.tslib.Timestamp
```

К сожалению, поле с датами не использовалось в качестве индекса для датафрейма. Вместо этого по умолчанию используются целочисленные метки индекса с началом отсчета в 0:

In[36]:
```
# к сожалению, индекс использует числовые значения,
# что затрудняет вывод данных по дате
df.index
```

Out[36]:
```
RangeIndex(start=0, stop=61, step=1)
```

Обратите внимание, что теперь это `RangeIndex`, тогда как в предыдущих версиях pandas индекс был бы целочисленным индексом. Мы рассмотрим это различие чуть позже.

Программный код можно подправить с помощью параметра `index_col` функции `pd.read_csv()`, который указывает, какой столбец в файле следует использовать в качестве индекса:

In[37]:
```
# считываем данные снова, теперь задаем столбец с датами
# в качестве индекса датафрейма
df = pd.read_csv('Data/goog.csv',
                 parse_dates=['Date'],
                 index_col='Date')
df
```

Out[37]:

Date	Open	High	Low	Close	Volume
2016-12-19	790.219971	797.659973	786.270020	794.200012	1225900
2016-12-20	796.760010	798.650024	793.270020	796.419983	925100
2016-12-21	795.840027	796.676025	787.099976	794.559998	1208700
2016-12-22	792.359985	793.320007	788.580017	791.260010	969100
2016-12-23	790.900024	792.739990	787.280029	789.909973	623400
...	...	...	...	...	...
2017-03-13	844.000000	848.684998	843.250000	845.539978	1149500
2017-03-14	843.640015	847.239990	840.799988	845.619995	779900
2017-03-15	847.590027	848.630005	840.770020	847.200012	1379600
2017-03-16	849.030029	850.849976	846.130005	848.780029	970400
2017-03-17	851.609985	853.400024	847.109985	852.119995	1712300

[61 rows x 5 columns]

И теперь индекс – это `DateTimeIndex`, который позволяет искать строки, используя даты.

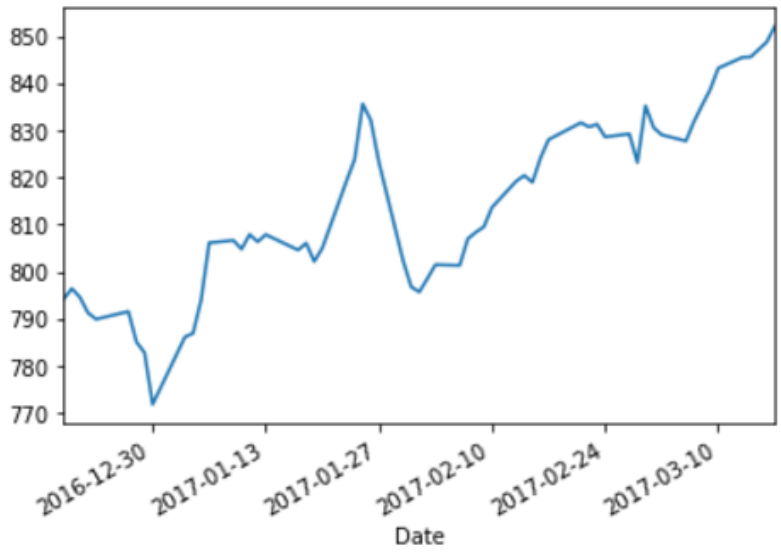
In[38]:
```
# и теперь индекс - это DatetimeIndex
df.index
```

```
Out[38]:
DatetimeIndex(['2016-12-19', '2016-12-20', '2016-12-21', '2016-12-22',
               '2016-12-23', '2016-12-27', '2016-12-28', '2016-12-29',
               '2016-12-30', '2017-01-03', '2017-01-04', '2017-01-05',
               '2017-01-06', '2017-01-09', '2017-01-10', '2017-01-11',
               '2017-01-12', '2017-01-13', '2017-01-17', '2017-01-18',
               '2017-01-19', '2017-01-20', '2017-01-23', '2017-01-24',
               '2017-01-25', '2017-01-26', '2017-01-27', '2017-01-30',
               '2017-01-31', '2017-02-01', '2017-02-02', '2017-02-03',
               '2017-02-06', '2017-02-07', '2017-02-08', '2017-02-09',
               '2017-02-10', '2017-02-13', '2017-02-14', '2017-02-15',
               '2017-02-16', '2017-02-17', '2017-02-21', '2017-02-22',
               '2017-02-23', '2017-02-24', '2017-02-27', '2017-02-28',
               '2017-03-01', '2017-03-02', '2017-03-03', '2017-03-06',
               '2017-03-07', '2017-03-08', '2017-03-09', '2017-03-10',
               '2017-03-13', '2017-03-14', '2017-03-15', '2017-03-16',
               '2017-03-17'],
              dtype='datetime64[ns]', name='Date', freq=None)
```

Визуализация

Вопросы визуализации мы рассмотрим довольно подробно в главе 14 «Визуализация», но перед этим мы иногда будем выполнять быструю визуализацию данных в pandas. Визуализировать данные с помощью pandas довольно просто. Все, что нужно сделать – это вызвать метод `.plot()`. Следующий программный код иллюстрирует построение графика на основе значений столбца `Close`:

```
In[39]:
# на основе значений столбца Close строим график
df.Close.plot();
```



Выводы

В этой главе мы установили научный дистрибутив Anaconda для Python. Он также устанавливает pandas и Jupyter Notebook, позволяя вам настроить среду для обработки и анализа данных, а также создавать тетрадки для визуализации, представления и совместного использования результатов анализа.

Мы также познакомились с объектами `Series` и `DataFrame` библиотеки pandas, продемонстрировав некоторые базовые возможности. Мы показали, как выполнить несколько базовых операций, которые можно использовать на ранних этапах работы в pandas перед тем, как погрузиться в изучение деталей.

В следующих нескольких главах мы подробно рассмотрим операции с объектами `Series` и `DataFrame`. Следующая глава будет конкретно посвящена объекту `Series`.

ГЛАВА 3 ПРЕДСТАВЛЕНИЕ ОДНОМЕРНЫХ ДАННЫХ С ПОМОЩЬЮ ОБЪЕКТА SERIES

Объект `Series` – это основной строительный блок библиотеки pandas. Он представляет собой набор значений одного и того же типа данных, похожий на одномерный массив. Объект `Series` часто используется для вычисления статистик по отдельной переменной. Несмотря на то, что объект `Series` может выглядеть как массив, он имеет соответствующий индекс, который можно использовать для очень эффективного отбора значений на основе меток.

Кроме того, объект `Series` выполняет автоматическое выравнивание данных между собой и другими объектами pandas. **Выравнивание данных (data alignment)** – это ключевая особенность библиотеки pandas. Данные представляют собой несколько объектов pandas, которые сопоставляются метке индекса перед выполнением любой операции. Это упрощает выполнение операций, не требуя написания дополнительного программного кода.

В этой главе мы рассмотрим, как исследовать переменную с помощью объекта `Series`, включая использование индекса для получения выборок. Это исследование будет включать в себя обзор нескольких способов, связанных с присвоением меток индекса, созданием срезов данных, формированием запросов к данным, выравниванием и переиндексацией данных. В частности, в этой главе мы рассмотрим следующие темы:

- Создание серии с использованием питоновских списков, словарей, функций NumPy и скалярных значений
- Доступ к значениям объекта `Series` с помощью свойств `.index` и `.values`
- Определение размера и формы объекта `Series`
- Установка индекса во время создания объекта `Series`
- Использование методов `.head()`, `.tail()` и `.take()` для вывода значений
- Поиск значений по метке индекса и позиции
- Создание срезов и популярные способы создания срезов
- Выравнивание данных с помощью меток индекса
- Выполнение логического отбора
- Переиндексация объекта `Series`
- Модификация значений на месте

Настройка библиотеки pandas

Мы начнем рассмотрение примеров в этой главе, задав следующие инструкции для импорта библиотек и настройки вывода:

```
In[1]:
# импортируем numpy и pandas
import numpy as np
import pandas as pd

# импортируем datetime
import datetime
from datetime import datetime, date

# задаем некоторые настройки pandas, регулирующие
# формат вывода
pd.set_option('display.notebook_repr_html', False)
pd.set_option('display.max_columns', 8)
pd.set_option('display.max_rows', 10)
pd.set_option('display.width', 80)

# импортируем matplotlib для построения графиков
import matplotlib.pyplot as plt
%matplotlib inline
```

Создание объекта Series

Объект `Series` можно создать с использованием нескольких методов. Мы рассмотрим следующие три метода:

- Создание серии с помощью питоновского списка или словаря
- Создание серии с помощью массивов NumPy
- Использование скалярного значения

Создание объекта Series с помощью питоновских списков и словарей

Объект `Series` можно создать из списка Python:

```
In[2]:
# создаем серию значений из списка
s = pd.Series([10, 11, 12, 13, 14])
s

Out[2]:
0    10
1    11
2    12
3    13
4    14
dtype: int64
```

Первый столбец чисел представляет собой индексные метки объекта `Series`. Второй столбец содержит значения. `dtype: int64` означает, что тип данных, к которому относятся значения серии – это `int64`.
По умолчанию pandas создает индекс, состоящий из последовательных целых чисел с началом отсчета в 0. Это делает серию похожей на массив, использующийся во многих других языках программирования. В качестве примера мы можем найти значение для `метки 3`:

```
In[3]:
# получаем значение, записанное в метке индекса 3
s[3]

Out[3]:
13
```

Обратите внимание, что поиск был осуществлен по значению метки, а не по позиции, начинающейся с 0. Мы рассмотрим это подробнее чуть позже.

Кроме целых чисел можно использовать и другие типы данных. Следующий программный код создает серию строковых значений:

```
In[4]:
# создаем серию строковых значений
pd.Series(['Mike', 'Marcia', 'Mikael', 'Bleu'])

Out[4]:
0      Mike
1    Marcia
2    Mikael
3      Bleu
dtype: object
```

Чтобы создать серию, состоящую из последовательности `n` одинаковых значений `v`, используйте сокращенную запись Python для создания списка `[v]*n`. Следующий программный код создает пять значений `2`:

```
In[5]:
# создаем последовательность из пяти двоек
pd.Series([2]*5)

Out[5]:
0    2
1    2
2    2
3    2
4    2
dtype: int64
```

Схожим сокращением является следующий программный код, который использует питоновское сокращение для использования каждого символа в качестве элемента списка:

```
In[6]:
# используем каждый символ в качестве значения
pd.Series(list('abcde'))

Out[6]:
0    a
1    b
2    c
3    d
4    e
dtype: object
```

Серию можно непосредственно инициализировать из словаря Python. При использовании словаря ключи словаря будут использованы в качестве меток индекса:

```
In[7]:
# создаем объект Series из словаря
pd.Series({'Mike': 'Dad',
           'Marcia': 'Mom',
           'Mikael': 'Son',
           'Bleu': 'Best doggie ever' })
```

```
Out[7]:
Bleu      Best doggie ever
Marcia                 Mom
Mikael                 Son
Mike                   Dad
dtype: object
```

Создание объекта Series с помощью функций NumPy

Создание объектов `Series` с помощью различных функций NumPy является обычной практикой. В качестве примера следующий программный код использует функцию NumPy `np.arange`, чтобы создать последовательность целочисленных значений от 4 до 8:

```
In[8]:
# создаем последовательность целочисленных
# значений от 4 до 8
pd.Series(np.arange(4, 9))
```

```
Out[8]:
0    4
1    5
2    6
3    7
4    8
dtype: int32
```

Метод `np.linspace()` похож по функционалу, однако он позволяет указать количество значений, которые должны быть созданы между двумя указанными значениями включительно, для этого задаем количество шагов:

```
In[9]:
# создаем последовательность из 5 значений,
# лежащих в интервале от 0 до 9
pd.Series(np.linspace(0, 9, 5))
```

```
Out[9]:
0    0.00
1    2.25
2    4.50
3    6.75
4    9.00
dtype: float64
```

Кроме того, часто генерируется набор случайных чисел с помощью `np.random.normal()`. Следующий программный код генерирует пять случайных чисел из нормального распределения:

```
In[10]:
# генерируем случайные числа
np.random.seed(12345) # всегда генерирует одни и те же числа
# создаем серию из 5 нормально распределенных случайных чисел
pd.Series(np.random.normal(size=5))
```

```
Out[10]:
0   -0.204708
1    0.478943
2   -0.519439
3   -0.555730
4    1.965781
dtype: float64
```

Создание объекта Series с помощью скалярного значения

Объект `Series` также можно создать с помощью скалярного значения.

```
In[11]:
# создаем объект Series, состоящий
# из одного элемента
s = pd.Series(2)
s

Out[11]:
0    2
dtype: int64
```

Результат похож на вырожденный случай, когда серия имеет только одно значение. Однако бывают ситуации, где это важно, например, когда серия умножается на скалярное значение:

```
In[12]:
# создаем объект Series
s = pd.Series(np.arange(0, 5))
# умножаем все значения на 2
s * 2

Out[12]:
0    0
1    2
2    4
3    6
4    8
dtype: int32
```

Сначала библиотека pandas создает объект `Series`, состоящий из целочисленных значений от 0 до 4. Затем она берет значение `2` и создает объект `Series` с индексом, соответствующим индексу серии `s`, а затем выполняет умножение, сопоставляя данные обеих серий. Мы снова рассмотрим этот пример более подробно в главе.

Свойства .index и .values

Каждый объект `Series` состоит из серии значений и индекса. Доступ к значениям можно получить с помощью свойства `.values`:

```
In[13]:
# получаем значения в объекте Series
s = pd.Series([1, 2, 3])
s.values

Out[13]:
array([1, 2, 3], dtype=int64)
```

Результатом является объект массива NumPy, как показано ниже:

```
In[14]:
# показываем, что результат - это массив NumPy
type(s.values)

Out[14]:
numpy.ndarray
```

Данный вывод приводится в информационных целях. Мы не будем рассматривать массивы NumPy в этой книге. Исторически pandas использовала массивы NumPy в своей основе, поэтому в прошлом важно было знать основы работы с массивами NumPy, однако в последних версиях эта зависимость между NumPy и pandas была удалена. Однако для удобства `.values` возвращает массив NumPy, даже если базовое представление не является массивом NumPy.
Кроме того, индекс для серии можно получить с помощью `.index`:

```
In[15]:
# получаем индекс объекта Series
s.index

Out[15]:
RangeIndex(start=0, stop=3, step=1)
```

Тип индекса, создаваемый pandas – это `RangeIndex`. Это изменение, внесенное в библиотеку pandas, о котором не говорилось в первом издании книги, тогда такого типа индекса не было. Объект `RangeIndex` представляет собой диапазон значений от стартового (`start`) до последнего значения (`stop`) с указанным шагом (`step`). Этот индекс более удобен для работы по сравнению с используемым ранее `Int64Index`.

`RangeIndex` - это просто один из типов индексов, которые мы рассмотрим позже (большая часть информации об индексах приводится в главе 6 «Индексация данных»).

Размер и форма объекта Series

Количество элементов в объекте `Series` можно определить несколькими способами, первый из которых – использовать питоновскую функцию `len()`:

```
In[16]:
# создаем серию
s = pd.Series([0, 1, 2, 3])
len(s)

Out[16]:
4
```

Тот же самый результат можно получить, используя свойство `.size`:

```
In[17]:
# .size также позволяет узнать о количестве
# элементов в объекте Series
s.size

Out[17]:
```

Альтернативным способом получения информации о размере объекта `Series` является использование свойства `.shape`. Оно возвращает кортеж из двух значений, однако здесь приводится лишь первое значение, представляющее размер:

In[18]:
```
# .shape - это кортеж с одним значением
s.shape
```

Out[18]:
```
(4,)
```

Установка индекса во время создания объекта Series

Метки в индексе могут быть заданы при создании объекта `Series` с помощью параметра `index`. Следующий программный код создает объект `Series` и присваивает строковое значение каждой метке индекса:

In[19]:
```
# явно создаем индекс
labels = ['Mike', 'Marcia', 'Mikael', 'Bleu']
role = ['Dad', 'Mom', 'Son', 'Dog']
s = pd.Series(labels, index=role)
s
```

Out[19]:
```
Dad      Mike
Mom    Marcia
Son    Mikael
Dog      Bleu
dtype: object
```

Воспользовавшись свойством `.index`, мы обнаруживаем, что был создан следующий индекс:

```
In[20]:
# исследуем индекс
s.index

Out[20]:
Index(['Dad', 'Mom', 'Son', 'Dog'], dtype='object')
```

Используя этот индекс, мы можем узнать, какой метке индекса соответствует значение `Dad`?:

```
In[21]:
# какой метке индекса соответствует
# значение Dad?
s['Dad']

Out[21]:
'Mike'
```

Использование методов .head(), .tail() и .take() для вывода значений

Библиотека pandas предлагает методы `.head()` и `.tail()` для исследования первых или последних строк в объекте `Series`. По умолчанию они возвращают первые или последние пять строк, однако это можно изменить с помощью параметра `n`.
Давайте создадим объект `Series`:

```
In[22]:
# создаем объект Series из 9 элементов
s = pd.Series(np.arange(1, 10),
              index=list('abcdefghi'))
```

Следующий программный код извлекает первые 5 строк:

```
In[23]:
# выводим первые 5 строк
s.head()

Out[23]:
a    1
b    2
c    3
d    4
e    5
dtype: int32
```

Количество элементов можно изменить с помощью параметра `n` (или просто указав число):

```
In[24]:
# выводим первые 3 строки
s.head(n = 3) # еще можно применить s.head(3)

Out[24]:
a    1
b    2
c    3
dtype: int32
```

Метод `.tail()` возвращает последние пять строк:

```
In[25]:
# выводим последние 5 строк
```

```
s.tail()
```

```
Out[25]:
e    5
f    6
g    7
h    8
i    9
dtype: int32
```

Если мы укажем число, отличное от 5, метод `.tail()` выведет последние *n* строк, где *n* – это заданное нами число.

```
In[26]:
# выводим последние 3 строки
s.tail(n = 3) # еще можно применить s.tail(3)
```

```
Out[26]:
g    7
h    8
i    9
dtype: int32
```

Метод `.take()` возвращает строки серии, соответствующие указанным целочисленным позициям:

```
In[27]:
# отобрать строки, соответствующие
# позициям 1, 5 и 8
s.take([1, 5, 8])
```

```
Out[27]:
b    2
f    6
i    9
dtype: int32
```

Получение значений в объекте Series по метке или позиции

Значения в объекте `Series` можно получить двумя способами: по метке индекса или по позиции с началом отсчета в 0. Библиотека pandas предлагает несколько способов для поиска значений. Давайте рассмотрим несколько распространенных методов.

Поиск по метке с помощью оператора [] и свойства .ix[]

Явный поиск по метке выполняется с помощью оператора `[]`. Этот оператор обычно просматривает значения, основываясь на заданных метках индекса.
Сначала создадим объект `Series`:

In[28]:
```
# мы создадим эту серию для проверки
# различных способов поиска
s1 = pd.Series(np.arange(10, 15), index=list('abcde'))
s1
```

Out[28]:
```
a    10
b    11
c    12
d    13
e    14
dtype: int32
```

Чтобы найти отдельное значение, воспользуйтесь меткой индекса нужного элемента:

In[29]:
```
# получаем значение с меткой 'a'
s1['a']
```

Out[29]:
```
10
```

Чтобы сразу извлечь несколько элементов, воспользуйтесь списком меток индекса:

In[30]:
```
# получаем несколько элементов
s1[['d', 'b']]
```

Out[30]:
```
d    13
b    11
dtype: int32
```

Кроме того, мы можем найти значения с помощью целочисленных позиций:

In[31]:
```
# получаем значения, указав их позиции
s1[[3, 1]]
```

Out[31]:
```
d    13
b    11
dtype: int32
```

Этот программный код работает исключительно потому, что индекс не использует целочисленные метки. Если целочисленные значения передать в [] и индекс будет использовать целочисленные значения, то будет осуществлен поиск совпадений с целочисленными метками. Данный факт можно продемонстрировать с помощью следующего программного кода:

In[32]:
```
# осуществляем поиск совпадений
# с целочисленными метками
s2 = pd.Series([1, 2, 3, 4], index=[10, 11, 12, 13])
s2
```

Out[32]:
```
10    1
```

```
11    2
12    3
13    4
dtype: int64
```

Следующий программный код ищет значения в метках `13` и `10`, а не в позициях `13` и `10`:

```
In[33]:
# поиск по метке, а не по позиции
s2[[13, 10]]

Out[33]:
13    4
10    1
dtype: int64
```

Поиск с помощью оператора `[]` идентичен использованию свойства `.ix[]`. Однако, начиная с версии pandas 0.20.1, применение свойства `.[ix]` объявлено устаревшим (deprecated). Причиной этого является путаница, вызываемая передачей целочисленных значений операторам и разницей получаемых результатов в зависимости от типа метки в индексе.
По причине этого ни `[]`, ни `.ix[]` не должны использоваться для поиска. Вместо этого используйте свойства `.loc[]` и `.iloc[]`, которые явно осуществляют поиск по метке или позиции.

Явный поиск по позиции с помощью свойства .iloc[]

Поиск значений по позиции можно выполнить с помощью свойства `.iloc[]`. Ниже показано использование целых чисел в качестве параметров:

```
In[34]:
# ищем явно по позиции
s1.iloc[[0, 2]]

Out[34]:
a    10
c    12
dtype: int32
```

Следующий программный код осуществляет поиск по позиции, хотя индекс имеет целочисленные метки:

```
In[35]:
# ищем явно по позиции
s2.iloc[[3, 2]]

Out[35]:
13    4
12    3
dtype: int64
```

Обратите внимание, что если вы укажете несуществующую позицию (ниже нуля или большее значение), то будет выдано исключение.

Явный поиск по меткам с помощью свойства .loc[]

Поиск по меткам еще можно выполнить с помощью свойства `.loc[]`:

```
In[36]:
# ищем явно по меткам
s1.loc[['a', 'd']]

Out[36]:
a    10
d    13
dtype: int32
```

Можно использовать целочисленные метки:

```
In[37]:
# получаем элементы в позициях 11 и 12
s2.loc[[11, 12]]

Out[37]:
11    2
12    3
dtype: int64
```

Обратите внимание, что при передаче метки, отсутствующей в индексе, свойство `.loc[]` ведет себя иначе, чем свойство `.iloc[]`. В этом случае библиотека pandas вернет значение `NaN` вместо выдачи исключения. Однако обратите внимание, что в будущих версиях библиотеки pandas передача списков с отсутствующей меткой в `.loc[]` будет приводить к ошибке, поэтому лучше использовать метод `.reindex()`.

```
In[38]:
# значение для несуществующей метки
# будет значением NaN

# s1.loc[['a', 'f']]
s1.reindex(['a', 'f'])

Out[38]:
a    10.0
f     NaN
dtype: float64
```

Что такое значение NaN? Мы рассмотрим его более подробно чуть позже в этой главе, однако pandas использует его для представления отсутствующих данных или чисел, которые не могут быть найдены с помощью индексных запросов. Кроме того, эти значения важны при работе с различными статистическими методами, которые мы также рассмотрим далее в этой главе.

Создание срезов объекта Series

Библиотека pandas позволяет создавать срезы данных. **Создание срезов или слайсинг (slicing)** – это мощный способ извлечения подмножеств данных из объекта библиотеки pandas. С помощью слайсинга мы можем отобрать данные с помощью позиций или меток индекса и получить больший контроль над последовательностью отбора элементов (отбирать

в прямом или обратном порядке) или интервалом отбора элементов (отбирать каждый элемент, каждый второй элемент).
При создании среза происходит перезагрузка обычного оператора массива `[]` (а также свойств `.loc[]`, `.iloc[]` и `.ix[]`), чтобы принять **объект-срез (slice object)**. Объект-срез создается с помощью синтаксиса `start:end:step`, компоненты представляют собой первый элемент (`start`), последний элемент (`end`) и приращение, которое еще называют шагом (`step`).
Каждый компонент среза является опциональным и за счет этого обеспечивается удобство, можно отбирать строки целиком, опуская тот или иной компонент среза.
Чтобы проиллюстрировать создание срезов, мы создадим следующий объект `Series`:

In[39]:
```
# создаем объект Series, который будем использовать
# для формирования срезов
# используем метки индекса, начинающиеся не с 0, чтобы
# продемонстрировать, как можно создать срез,
# используя позиции
s = pd.Series(np.arange(100, 110), index=np.arange(10, 20))
s
```

Out[39]:
```
10    100
11    101
12    102
13    103
14    104
15    105
16    106
17    107
18    108
19    109
dtype: int32
```

Мы можем последовательно отобрать элементы, используя для формирования среза синтаксис `start:end`. Следующий программный код отбирает в объекте `Series` пять элементов в позициях с `1`-ой по `5`-ю. Поскольку мы не указали компонент `step`, то по умолчанию он будет равен `1`. Кроме того, обратите внимание, что `end` метка не будет включена в результат:

In[40]:
```
# срез, содержащий элементы с позиции 1 по 5
s[1:6]
```

```
Out[40]:
11    101
12    102
13    103
14    104
15    105
dtype: int32
```

Этот результат примерно эквивалентен следующему программному коду:

```
In[41]:
# поиск с помощью списка позиций
s.iloc[[1, 2, 3, 4, 5]]

Out[41]:
11    101
12    102
13    103
14    104
15    105
dtype: int32
```

Этот код только примерно эквивалентен, потому что применение свойства `.iloc[]` возвращает копию исходных данных. Срез – это ссылка на исходные данные. Модификация содержимого полученного среза будет влиять на исходную серию. Мы рассмотрим этот процесс позже в разделе, посвященном модификации данных объекта `Series` на месте.

Можно сформировать срез путем отбора каждого второго элемента, указав шаг 2:

```
In[42]:
# отбираем элементы в позициях 1, 3, 5
s[1:6:2]

Out[42]:
11    101
13    103
15    105
dtype: int32
```

Как уже говорилось ранее, каждый компонент среза является опциональным. Если компонент `start` опустить, результаты будут выведены, начиная с первого элемента. Например, следующий программный код является лаконичной альтернативой методу `.head()`:

```
In[43]:
# создаем срез, содержащий первые пять
# элементов, эквивалентно .head(5)
s[:5]

Out[43]:
10    100
11    101
12    102
13    103
14    104
dtype: int32
```

Отбор элементов, начинающийся с определенной позиции, можно осуществить, задав компонент `start` и опустив компонент `end`. Следующий программный код отбирает все элементы, начиная с 4-го:

```
In[44]:
# создаем срез путем отбора всех
# элементов, начиная с 4-го
s[4:]
```

```
Out[44]:
14    104
15    105
16    106
17    107
18    108
19    109
dtype: int32
```

Кроме того, компонент `step` можно использовать в обоих предыдущих сценариях для пропуска элементов:

```
In[45]:
# отбираем каждый второй элемент
# в первых пяти позициях
s[:5:2]
```

```
Out[45]:

10    100
12    102
14    104
dtype: int32
```

```
In[46]:
# отбираем каждый второй элемент,
# начиная с четвертой позиции
s[4::2]
```

```
Out[46]:
14    104
16    106
18    108
dtype: int32
```

Использование отрицательного значения компонента `step` приведет к тому, что отбор элементов осуществляется в обратном порядке. Следующий программный код показывает, как можно отобрать элементы объекта `Series` в обратном порядке:

```
In[47]:
# отбираем элементы Series
# в обратном порядке
s[::-1]
```

```
Out[47]:
19    109
18    108
17    107
16    106
15    105
14    104
13    103
12    102
11    101
10    100
dtype: int32
```

Значение `-2` будет возвращать каждый второй элемент, исходя из стартовой позиции и двигаясь в обратном порядке к началу объекта `Series`. Следующий программный код возвращает в обратном порядке каждый второй элемент, начиная с 4-й позиции:

```
In[48]:
# отбираем в обратном порядке каждый
# второй элемент, начиная с 4-й позиции
s[4::-2]

Out[48]:
14    104
12    102
10    100
dtype: int32
```

Отрицательные значения компонентов `start` и `end` имеют особый смысл. Отрицательное стартовое значение `-n` означает отбор последних `n` строк:

```
In[49]:
# -4: означает отбор
# последних 4 строк
s[-4:]

Out[49]:
16    106
17    107
18    108
19    109
dtype: int32
```

Отрицательное стартовое значение `-n` вернет все строки, кроме последних `n` строк.

```
In[50]:
# :-4 означает отбор всех строк,
# кроме последних четырех
s[:-4]

Out[50]:
10    100
11    101
12    102
13    103
14    104
15    105
dtype: int32
```

Отрицательные значения компонентов `start` и `end` можно объединить. Следующий программный код сначала извлекает последние четыре

строки, а затем из них отбирает все строки, кроме последней (таким образом, извлекаем первые три строки):

```
In[51]:
# эквивалентно s.tail(4).head(3)
s[-4:-1]

Out[51]:
16    106
17    107
18    108
dtype: int32
```

Кроме того, можно создать срез серии с помощью нечислового индекса. Чтобы продемонстрировать это, давайте создадим следующий объект `Series`:

```
In[52]:
# создаем объект Series, чтобы получить
# следующие два среза из этой серии
s = pd.Series(np.arange(0, 5),
              index=['a', 'b', 'c', 'd', 'e'])
s

Out[52]:
a    0
b    1
c    2
d    3
e    4
dtype: int32
```

На основе этого объекта `Series` создадим срез, отобрав элементы, исходя из их позиций (как и раньше):

```
In[53]:
# создаем срез на основе позиций, поскольку
# индекс использует символы
s[1:3]

Out[53]:
b    1
c    2
dtype: int32
```

Однако при использовании в качестве компонентов среза значений, отличных от целочисленных, библиотека pandas попытаться понять тип данных и выбрать из серии соответствующие элементы. В качестве примера следующий программный код отбирает элементы с `'b'` по `'d'`:

```
In[54]:
# создаем срез по строковым меткам индекса
s['b':'d']

Out[54]:
b    1
c    2
d    3
dtype: int32
```

Выравнивание данных по меткам индекса

Выравнивание данных в объекте `Series` с помощью меток индекса является фундаментальным принципом библиотеки pandas, а также одним из ее самых главных преимуществ. Выравнивание осуществляет автоматическое согласование связанных друг с другом значений в нескольких объектах `Series` на основе меток индекса. Это позволяет избежать множества ошибок, возникающих при сопоставлении данных нескольких наборов, когда используются стандартные процедуры.
Чтобы продемонстрировать выравнивание, давайте возьмем пример с добавлением значений в двух объектах `Series`. Начнем со следующих двух объектов `Series`, представляющих собой две разные переменные (`a` и `b`):

```
In[55]:
# первая серия для выравнивания
s1 = pd.Series([1, 2], index=['a', 'b'])
s1

Out[55]:
a    1
b    2
dtype: int64

In[56]:
# вторая серия для выравнивания
s2 = pd.Series([4, 3], index=['b', 'a'])
s2

Out[56]:
b    4
a    3
dtype: int64
```

Теперь предположим, что нам нужно суммировать значения по этим переменным. Мы можем записать это просто как `s1 + s2`:

```
In[57]:
# складываем их
s1 + s2

Out[57]:
a    4
b    6
dtype: int64
```

С помощью заданной операции pandas сопоставляет значение каждой переменной в каждой серии, складывает эти значения и возвращает нам суммарные значения. Кроме того, к объекту `Series` можно применить скалярное значение. Результат будет заключаться в том, что скалярное значение будет применено к каждому значению в объекте `Series`:

```
In[58]:
# умножаем все значения в s3 на 2
s1 * 2

Out[58]:
a    2
b    4
```

```
dtype: int64
```

Помните, ранее мы говорили, что вернемся к созданию объекта `Series` на основе скалярного значения? При выполнении вышеописанной операции pandas фактически выполняет следующие действия:

```
In[59]:
# создаем серию на основе скалярного
# значения 2, но с индексом серии s1
t = pd.Series(2, s1.index)
t

Out[59]:
a    2
b    2
dtype: int64

In[60]:
# умножаем s1 на t
s1 * t

Out[60]:
a    2
b    4
dtype: int64
```

Первый этап – это создание объекта `Series` на основе скалярного значения, но с индексом исходной серии `s1`. Затем умножение применяется к выровненным значениям двух объектов `Series`, которые идеально выравниваются, потому что индекс у них идентичен.
Метки в индексах не обязательны для выравнивания. Если выравнивание не происходит, pandas в качестве результата возвращает значение `NaN`:

```
In[61]:
# мы создаем серию s3, чтобы сложить
# ее значения со значениями серии s1
s3 = pd.Series([5, 6], index=['b', 'c'])
s3

Out[61]:
b    5
c    6
dtype: int64

In[62]:
# серии s1 и s3 имеют разные метки индексов
# для a и c будут получены значения NaN
s1 + s3

Out[62]:
a    NaN
b    7.0
c    NaN
dtype: float64
```

По умолчанию значение `NaN` – это результат любого выравнивания в pandas, когда метка индекса в серии не совпадает с меткой индекса другой серии. Это важная особенность библиотеки pandas по сравнению с библиотекой NumPy. Даже если метки не выровнены, не должно быть исключений. Это помогает, когда некоторые данные отсутствуют, однако

это приемлемо для работы. Обработка продолжается, однако pandas позволяет узнать о наличии проблемы (хотя отсутствующие значения – это не обязательно проблема), возвращая значение `NaN`.
Метки в индексе не обязательно должны быть уникальным. Операция выравнивания фактически образует декартово произведение меток в двух объектах `Series`. Если в серии `1` имеется *n* меток, а в серии 2 - *m* меток, тогда результат будет состоять из *n*m* итоговых строк.
Чтобы продемонстрировать это, воспользуемся следующими двумя объектами `Series`:

```
In[63]:
# 2 метки 'a'
s1 = pd.Series([1.0, 2.0, 3.0], index=['a', 'a', 'b'])
s1

Out[63]:
a    1.0
a    2.0
b    3.0
dtype: float64

In[64]:
# 3 метки 'a'
s2 = pd.Series([4.0, 5.0, 6.0, 7.0], index=['a', 'a', 'c', 'a'])
s2

Out[64]:
a    4.0
a    5.0
c    6.0
a    7.0
dtype: float64
```

В итоге для меток `'a'` получим обычные значения (всего 6 меток `'a'`), а для меток `'b'` и `'c'` получим значения `NaN`.

```
In[65]:
# получим 6 меток 'a' и значения NaN
# для меток 'b' и 'c'
s1 + s2

Out[65]:
a    5.0
a    6.0
a    8.0
a    6.0
a    7.0
a    9.0
b    NaN
c    NaN
dtype: float64
```

Выполнение логического отбора

Индексы предлагают очень мощный и эффективный способ поиска значений в объекте `Series` на основе их меток. Но что, если вы хотите найти записи в объекте `Series` на основе значений?
Для решения этой задачи библиотека pandas предлагает воспользоваться логическим отбором (отбором на основе булевых значений). Он

применяет логическое выражение к значениям объекта `Series` и возвращает новую серию булевых значений, представляющую собой результат применения этого выражения к каждому значению. Затем этот результат можно использовать для извлечения лишь тех значений, где был получен результат `True`. Чтобы продемонстрировать логический отбор, давайте создадим объект `Series` и применим оператор `>=` (больше или равно), чтобы определить значения, которые больше или равны `3`:

```
In[66]:
# какие строки имеют значения больше или равно 3?
s = pd.Series(np.arange(0, 5), index=list('abcde'))
logical_results = s >= 3
logical_results

Out[66]:
a    False
b    False
c    False
d     True
e     True
dtype: bool
```

В результате получаем объект `Series` с точно такими же метками индекса, что и в исходной серии, и результатами применения логического выражения к значению каждой метки. Тип значений (`dtype`) является булевым или логическим (`bool`).
Затем эту серию можно использовать для отбора значений из исходной серии. Этот отбор осуществляется путем передачи булевых результатов в оператор `[]` исходной серии.

```
In[67]:
# отобрать строки со значением True
s[logical_results]

Out[67]:
d    3
e    4
dtype: int32
```

Синтаксис можно упростить, выполнив логическую операцию внутри оператора `[]`:

In[68]:
```
# более краткий вариант
s[s > 5]
```

Out[68]:
```
Series([], dtype: int32)
```

К сожалению, многие логические операторы нельзя задать с помощью стандартного синтаксиса Python. В качестве примера приведен следующий программный код, выполнение которого закончится выдачей исключения:

In[69]:
```
# программный код дан в комментарии, потому что
# его выполнение приведет к выдаче исключения
# s[s >= 2 and s < 5]
```

Существуют технические причины, в силу которых предыдущий программный код не сработает. Решение состоит в том, чтобы выразить уравнение иначе, взяв в круглые скобки каждое логическое условие и используя другие операторы для логических операций `and`/`or` (`|` и `&`):

In[70]:
```
# правильный синтаксис
s[(s >=2) & (s < 5)]
```

Out[70]:
```
c    2
d    3
e    4
dtype: int32
```

С помощью метода `.all()` можно определить, все ли значения в объекте `Series` соответствуют заданному выражению. Программный код, приведенный ниже, спрашивает, все ли элементы в серии больше или равны `0`:

In[71]:
```
# все ли элементы >= 0?
(s >= 0).all()
```

Out[71]:
```
True
```

Метод `.any()` возвращает значение `True`, если какое-либо значение удовлетворяет заданному выражению. Нижеприведенный программный спрашивает, есть ли в серии какой-либо элемент меньше `2`:

In[72]:
```
# есть ли элемент < 2?
s[s < 2].any()
```

Out[72]:
```
True
```

С помощью метода `.sum()` вы можете определить количество элементов, удовлетворяющих выражению. Это связано с тем, что когда задана серия

булевых значений метод `.sum()` объекта `Series` рассматривает `True` как `1` и `False` как `0`:

```
In[73]:
# сколько значений < 2?
(s < 2).sum()

Out[73]:
2
```

Переиндексация объекта Series

Переиндексация в pandas - это процесс, который согласовывает данные в серии с набором меток. Переиндексация используется для выполнения выравнивания и поэтому является фундаментальной операцией.
Переиндексация выполняет несколько вещей:

- переупорядочивание существующих данных в соответствии с набором меток;
- вставка маркеров `NaN`, когда отсутствуют данные для метки;
- заполнение отсутствующих данных для метки с помощью тех или иных логических операций (по умолчанию используются значения `NaN`).

Переиндексацию легко выполнить, просто присвоив новый индекс свойству `.index` объекта `Series`. Ниже показано изменение индекса объекта `Series` данным образом:

```
In[74]:
# создаем случайную серию из 5 элементов
np.random.seed(123456)
s = pd.Series(np.random.randn(5))
s

Out[74]:
0    0.469112
1   -0.282863
2   -1.509059
3   -1.135632
4    1.212112
dtype: float64

In[75]:
# изменяем индекс
s.index = ['a', 'b', 'c', 'd', 'e']
s

Out[75]:
a    0.469112
b   -0.282863
c   -1.509059
d   -1.135632
e    1.212112
dtype: float64
```

Количество элементов в списке, присвоенному свойству `.index`, должно соответствовать количеству строк или будет выдано исключение. Кроме того, переиндексация модифицировала объект `Series` на месте.
Возможность создания нового индекса реализована с помощью метода `.reindex()`. Возьмем случай с присвоением нового индекса, в котором количество меток не совпадает с количеством значений:

```
In[76]:
# создаем серию, которую
# заново индексируем
np.random.seed(123456)
s1 = pd.Series(np.random.randn(4), ['a', 'b', 'c', 'd'])
s1

Out[76]:
a    0.469112
b   -0.282863
c   -1.509059
d   -1.135632
dtype: float64
```

Следующий программный код заново индексирует объект `Series` с помощью набора меток, в котором есть как совершенно новая метка (метка `'g'`), некоторые метки отсутствуют (метки `'b'` и `'d'`), а некоторые метки совпадают (метки `'a'` и `'c'`):

```
In[77]:
# индексируем заново, используя другое количество меток
# в итоге некоторые строки удалены и/или получены значения NaN
s2 = s1.reindex(['a', 'c', 'g'])
s2

Out[77]:
a    0.469112
c   -1.509059
g         NaN
dtype: float64
```

Есть несколько моментов, о которых важно упомянуть, говоря о методе `.reindex()`. Во-первых, результат метода `.reindex()` – это новый объект `Series`, а не модификация на месте. У нового объекта `Series` будет индекс с метками, указанными при передаче в функцию. Данные копируются по каждой метке исходного объекта `Series`. Если метка в исходном объекте `Series` не найдена, будет присвоено значение `NaN`. Наконец, строки в объекте `Series` с метками, отсутствующими в новом индексе, отбрасываются.

Кроме того, переиндексация индекса бывает полезной в тех случаях, когда вы хотите выровнять два объекта `Series`, чтобы выполнить какие-либо операции со значениями обеих серий, однако по какой-то причине объекты `Series` не имеют меток, по которым можно выполнить выравнивание. Часто бывает ситуация, когда одна серия имеет метки целочисленного типа, а другая – метки строкового типа, однако метки по своему содержательному смыслу одинаковы (обычно это бывает при загрузке данных из удаленных источников). В качестве примера создадим следующие объекты `Series`:

```
In[78]:
# использование разных типов для одних и тех же
# меток порождает большие проблемы
s1 = pd.Series([0, 1, 2], index=[0, 1, 2])
s2 = pd.Series([3, 4, 5], index=['0', '1', '2'])
s1 + s2

Out[78]:
0   NaN
1   NaN
2   NaN
0   NaN
1   NaN
2   NaN
dtype: float64
```

Несмотря на то, что метки в обеих сериях одинаковы по смыслу, они будут выравниваться по-разному из-за разных типов данных. Обнаружив проблему, ее легко устранить:

```
In[79]:
# заново индексируем, согласовав типы меток,
# и получаем нужный результат
s2.index = s2.index.values.astype(int)
s1 + s2

Out[79]:
0    3
1    5
2    7
dtype: int64
```

Метод `.reindex()` по умолчанию вставляет `NaN` в качестве отсутствующих значений, если метки не были найдены в исходной серии. Это значение можно изменить с помощью параметра `fill_value`. Следующий пример демонстрирует использование `0` вместо `NaN`:

```
In[80]:
# заполняем отсутствующее значение нулем
# вместо NaN
s2 = s.copy()
s2.reindex(['a', 'f'], fill_value=0)

Out[80]:
a    0.469112
f    0.000000
dtype: float64
```

При переиндексации упорядоченных данных типа временных рядов можно выполнить интерполяцию или заполнение значений. Более подробное обсуждение интерполяции и заполнения значений будет дано в главе 10 «Данные временных рядов», однако следующие примеры познакомят с базовыми принципами. Начнем с того, что создадим следующий объект `Series`:

```
In[81]:
# создаем пример, чтобы продемонстрировать
# варианты заполнения
s3 = pd.Series(['red', 'green', 'blue'], index=[0, 3, 5])
s3

Out[81]:
0      red
3    green
5     blue
dtype: object
```

Следующий пример иллюстрирует принцип **прямого заполнения (forward filling)**, часто называемый методом **последнего известного значения (last known value)**. Объект `Series` был заново проиндексирован, чтобы создать непрерывный целочисленный индекс, и с помощью параметра `method='ffill'` любым новым меткам индекса присваивается ранее известное значение, отличное от `NaN`:

```
In[82]:
# пример прямого заполнения
s3.reindex(np.arange(0,7), method='ffill')

Out[82]:
0      red
1      red
2      red
3    green
4    green
5     blue
6     blue
dtype: object
```

Новым меткам индекса `1` и `2` присваивается значение `red`, потому что для них последним известным значением, отличным от `NaN`, является значение `red` для метки `0`. Новой метке индекса `4` присваивается значение `green`, потому что для нее последним известным значением, отличным от `NaN`, является значение `green` для метки `3`. Новой метке индекса `6` присваивается значение `blue`, потому что для нее последним известным значением, отличным от `NaN`, является значение `blue` для метки `5`.

Следующий программный код применяет **обратное заполнение (backward filling)** с помощью `method='bfill'`:

```
In[83]:
# пример обратного заполнения
s3.reindex(np.arange(0,7), method='bfill')

Out[83]:
0      red
1    green
2    green
3    green
4     blue
5     blue
6      NaN
dtype: object
```

Новая метка индекса `6` не имеет предыдущего значения, поэтому ей присвоено значение `NaN`. Новой метке `4` присвоено значение метки `5` (`blue`). Новым меткам `2` и `1` присвоено значение метки `3` (`green`).

Модификация объекта Series на месте

Модификация объекта `Series` на месте - это несколько спорная тема. Когда это возможно, предпочтительно выполнять операции, которые возвращают новый объект `Series`, то есть изменения записаны в новой серии. Но при необходимости можно изменять значения и добавлять/удалять строки на месте.

Можно добавить дополнительную строку в серию прямо на месте, присвоив значение еще несуществующей метке индекса. Следующий программный код создает объект `Series` и добавляет дополнительный элемент в серию:

```
In[84]:
# генерируем серию для примера
np.random.seed(123456)
s = pd.Series(np.random.randn(3), index=['a', 'b', 'c'])
s

Out[84]:
a    0.469112
b   -0.282863
c   -1.509059
dtype: float64

In[85]:
# добавляем значение в серию
# это делается сразу на месте
# новый объект Series не возвращается
s['d'] = 100
s

Out[85]:
a      0.469112
b     -0.282863
c     -1.509059
d    100.000000
dtype: float64
```

Значение в определенной метке индекса можно изменить с помощью операции присвоения:

```
In[86]:
# модифицируем на месте
# значение в метке 'd'
s['d'] = -100
s

Out[86]:
a      0.469112
b     -0.282863
c     -1.509059
d   -100.000000
dtype: float64
```

Из объекта `Series` можно удалить строки, передав их метки индекса в функцию `del()`. Следующий программный код демонстрирует удаление строки с меткой индекса `'a'`:

```
In[87]:
```

```
# удаляем строку/элемент
del(s['a'])
s
```

```
Out[87]:
b     -0.282863
c     -1.509059
d   -100.000000
dtype: float64
```

Для добавления и удаления элементов используйте `pd.concat()`. Это позволит вам применить логический отбор при добавлении и удалении элементов.

Важно помнить, что когда вы создаете срезы срез связан с исходным объектом `Series`. Модификация значений в сформированном срезе изменит исходную серию. Рассмотрим следующий программный код, который отбирает первые два элемента в объекте `Series` и сохраняет их в новом объекте:

```
In[88]:
copy = s.copy() # сохраняем s
slice = copy[:2] # создаем срез с первыми двумя строками
slice
```

```
Out[88]:
b   -0.282863
c   -1.509059
dtype: float64
```

Присвоив новое значение элементу среза, мы изменим значение в исходной серии:

```
In[89]:
# меняем значение в метке 'b'
# нашего среза
slice['b'] = 0
# и смотрим исходную серию
copy
```

```
Out[89]:
b      0.000000
c     -1.509059
d   -100.000000
dtype: float64
```

Выводы

В этой главе вы узнали об объекте `Series` библиотеки pandas и о том, как его можно использовать для индексированного представления значений переменной. Мы начали с того, что научились создавать и инициализировать объект `Series` и его индекс, а затем разобрали выполнение операций с данными в одном или нескольких объектах `Series`. Мы рассмотрели выравнивание объектов `Series` по метке индекса и выполнение математических операций над выровненными значениями. Затем мы обсудили поиск данных по индексу, а также

выполнение запросов на основе данных (логические выражения). В самом конце этой главы мы разобрали выполнение переиндексации для изменения индексов и выравнивания данных.

В следующей главе вы узнаете, как для представления нескольких серий данных в единой табличной структуре используется объект `DataFrame`.

ГЛАВА 4 ПРЕДСТАВЛЕНИЕ ТАБЛИЧНЫХ И МНОГОМЕРНЫХ ДАННЫХ С ПОМОЩЬЮ ОБЪЕКТА DATAFRAME

Объект `DataFrame` библиотеки pandas расширяет возможности объекта `Series`, позволяя работать с двумя измерениями. Вместо одной серии значений каждая строка может иметь несколько значений характеристик, которые записываются в виде столбцов. Каждая строка датафрейма описывает различные характеристики исследуемого субъекта и каждый столбец может представлять различные типы данных.

Каждый столбец датафрейма представляет собой объект `Series` библиотеки pandas, и датафрейм можно рассматривать как форму данных, аналогичную электронной таблице или таблице базы данных. Но эти сравнения не позволяют полностью раскрыть все свойства объекта `DataFrame`, поскольку датафрейм имеет особенности, характерные только для библиотеки pandas, например, автоматическое выравнивание данных объектов `Series`, которые представляют собой столбцы.

Автоматическое выравнивание делает датафрейм более удобным для разведывательного анализа данных, чем электронные таблицы или базы данных. Сочетание возможности создавать срезы одновременно по строкам и столбцам и способность выполнять различные операции с данными и исследовать их в рамках датафрейма делает библиотеку pandas невероятно эффективной для поиска необходимой информации.

В этой главе мы подробно рассмотрим объект `DataFrame` объект pandas. Многие понятия будут вам уже знакомы по объекту `Series`, за исключением добавления данных и выполнения разнообразных манипуляций с данными. В частности, в этой главе мы рассмотрим следующие темы:

- Создание объекта `DataFrame` из питоновских объектов, функций NumPy, питоновских словарей, объектов Series библиотеки pandas и CSV-файлов
- Определение размерности датафрейма
- Назначение и изменение имен столбцов в датафрейме
- Выравнивание строк при создании датафрейма
- Отбор определенных столбцов и строк датафрейма
- Создание среза датафрейма
- Отбор строк и столбцов датафрейма по позиции и метке
- Поиск скалярного значения
- Применение логического отбора к датафрейму

Настройка библиотеки pandas

Мы начнем рассмотрение примеров в этой главе, задав следующие инструкции для импорта библиотек и настройки вывода:

```
In[1]:
# импортируем numpy и pandas
import numpy as np
import pandas as pd

# импортируем datetime
import datetime
from datetime import datetime, date

# задаем некоторые настройки pandas, регулирующие
# формат вывода
pd.set_option('display.notebook_repr_html', False)
pd.set_option('display.max_columns', 8)
pd.set_option('display.max_rows', 10)
pd.set_option('display.width', 80)

# импортируем matplotlib для построения графиков
import matplotlib.pyplot as plt
%matplotlib inline
```

Создание объектов DataFrame

Существует несколько способов создания датафрейма. Датафрейм можно создать из одномерного или многомерного набора данных. Мы рассмотрим следующие методы:

- Использование результатов функций NumPy
- Использование данных из питоновского словаря, состоящего из списков или объектов `Series`
- Использование данных CSV-файла.

Изучив каждый метод, мы рассмотрим, как задавать имена столбцов, покажем, как выполнить выравнивание строк при создании датафрейма, а также обсудим, как определить размерность датафрейма.

Создание объекта DataFrame на основе результатов функций NumPy

Датафрейм можно создать на основе одномерного массива NumPy, состоящего из целых чисел от 1 до 5:

```
In[2]:
# создаем датафрейм из одномерного
# массива NumPy
pd.DataFrame(np.arange(1, 6))
```

```
Out[2]:
   0
0  1
1  2
2  3
3  4
4  5
```

В первом столбце вывода отображаются метки созданного индекса. Поскольку индекс не был задан во время создания датафрейма, библиотека pandas создал базовый индекс `RangeIndex` с метками, начинающимися с 0.

Данные находятся во втором столбце и состоят из значений с `1` по `5`. Обозначение `0` над столбцом данных - это имя столбца. Когда имена

столбцов не заданы при создании датафрейма, библиотека pandas присваивает столбцам в качестве названий постепенно увеличивающиеся целые числа, начиная с 0.
Кроме того, можно воспользоваться многомерным массивом NumPy и это приведет к созданию нескольких столбцов:

In[3]:
```
# создаем датафрейм из двумерного
# массива NumPy
df = pd.DataFrame(np.array([[10, 11], [20, 21]]))
df
```

Out[3]:
```
    0   1
0  10  11
1  20  21
```

Доступ к столбцам `DataFrame` можно получить с помощью свойства `.columns`:

In[4]:
```
# получаем индекс столбцов
df.columns
```

Out[4]:
```
RangeIndex(start=0, stop=2, step=1)
```

Вышеприведенный вывод показывает, что когда имена столбцов не заданы, библиотека pandas создаст `RangeIndex` для представления столбцов.
Имена столбцов можно задать с помощью параметра `columns`. Следующий программный код создает объект `DataFrame`, который состоит из двух столбцов и содержит по два значения температуры для каждого города-столбца:

In[5]:
```
# задаем имена столбцов
df = pd.DataFrame(np.array([[70, 71], [90, 91]]),
                  columns=['Missoula', 'Philadelphia'])
df
```

Out[5]:
```
   Missoula  Philadelphia
0        70            71
1        90            91
```

Количество строк в объекте `DataFrame` можно вычислить с помощью функции `len()`:

In[6]:
```
# сколько строк?
len(df)
```

Out[6]:
```
2
```

Размерность объекта `DataFrame` можно вычислить с помощью свойства `.shape`:

```
In[7]:
# какова размерность датафрейма?
df.shape

Out[7]:
(2, 2)
```

Создание объекта DataFrame с помощью питоновского словаря и объектов Series

Для создания объекта `DataFrame` можно использовать питоновский словарь. При использовании питоновского словаря библиотека `pandas` будут использовать ключи в качестве имен столбцов, а значения для каждого ключа – в качестве значений столбца:

```
In[8]:
# создание датафрейма с помощью
# питоновского словаря
temps_missoula = [70, 71]
temps_philly = [90, 91]
temperatures = {'Missoula': temps_missoula,
                'Philadelphia': temps_philly}
pd.DataFrame(temperatures)

Out[8]:
   Missoula  Philadelphia
0        70            90
1        71            91
```

Распространенным способом создания объекта `DataFrame` является использование списка объектов `Series`, которые будут в датафрейме строками:

```
In[9]:
# создаем датафрейм для списка объектов Series
temps_at_time0 = pd.Series([70, 90])
temps_at_time1 = pd.Series([71, 91])
df = pd.DataFrame([temps_at_time0, temps_at_time1])
df

Out[9]:
    0   1
0  70  90
1  71  91
```

В этом сценарии каждый объект `Series` представляет собой значения температур для каждого города. Чтобы присвоить имена столбцам, мы могли бы попытаться воспользоваться параметром `columns`:

```
In[10]:
# попытаемся задать имена столбцов
df = pd.DataFrame([temps_at_time0, temps_at_time1],
                  columns=['Missoula', 'Philadelphia'])
df

Out[10]:
   Missoula  Philadelphia
0       NaN           NaN
1       NaN           NaN
```

Полученный результат отличается от наших ожиданий, поскольку все значения являются значениями `NaN`. Это можно устранить двумя способами. Первый способов – присвоить имена столбцам с помощью свойства `.columns`:

```
In[11]:
# задаем имена столбцов после создания датафрейма
df = pd.DataFrame([temps_at_time0, temps_at_time1])
df.columns = ['Missoula', 'Philadelphia']
df

Out[11]:
   Missoula  Philadelphia
0        70            90
1        71            91
```

Еще один способ заключается в использовании питоновского словаря, где ключи – это имена столбцов, а значение для каждого ключа – это объект `Series`, представляющий собой значения данного столбца:

```
In[12]:
# создаем датафрейм с помощью словаря,
# состоящего из объектов Series
temps_mso_series = pd.Series(temps_missoula)
temps_phl_series = pd.Series(temps_philly)
df = pd.DataFrame({'Missoula': temps_mso_series,
                   'Philadelphia': temps_phl_series})
df

Out[12]:
   Missoula  Philadelphia
0        70            90
1        71            91
```

Обратите внимание, что при создании объекта `DataFrame` добавляемая серия будет выровнена. Следующий программный код демонстрирует это путем добавления третьего города, у которого другие значения индекса:

```
In[13]:
# выравнивание происходит при создании датафрейма
temps_nyc_series = pd.Series([85, 87], index=[1, 2])
df = pd.DataFrame({'Missoula': temps_mso_series,
                   'Philadelphia': temps_phl_series,
                   'New York': temps_nyc_series})
df

Out[13]:
   Missoula  Philadelphia  New York
0      70.0          90.0       NaN
1      71.0          91.0      85.0
2       NaN           NaN      87.0
```

Создание объекта DataFrame на основе CSV-файла

Датафрейм можно создать, считав данные из CSV-файла с помощью функции `pd.read_csv()`.

Функция `pd.read_csv()` будет более подробно рассмотрена в главе 9 «Доступ к данным».

Чтобы продемонстрировать этот процесс, мы загрузим данные из файла, содержащий моментальный снимок индекса S&P 500. Этот файл называется `sp500.csv` и находится в каталоге `data`.
Первая строка файла содержит имена каждой переменной/столбца, а остальные 500 строк представляют собой акции 500 крупнейших компаний США. Следующий программный код загружает данные, при этом указываем столбец, который должен использоваться в качестве индекса, а также указываем только четыре конкретных столбца (0, 2, 3 и 7):

In[14]:
```
# считываем данные и выводим первые пять строк
# используем столбец Symbol в качестве индекса
# и считываем только те столбцы, у которых
# позиции 0, 2, 3, 7
sp500 = pd.read_csv("Data/sp500.csv",
                    index_col='Symbol',
                    usecols=[0, 2, 3, 7])
```

Вывод первых пяти строк с помощью метода `.head()` показывает нам следующую структуру и содержимое получившегося датафрейма:

In[15]:
```
# взглянем на первые 5 строк данных с помощью
# метода .head()
sp500.head()
```

Out[15]:

Symbol	Sector	Price	Book Value
MMM	Industrials	141.14	26.668
ABT	Health Care	39.60	15.573
ABBV	Health Care	53.95	2.954
ACN	Information Technology	79.79	8.326
ACE	Financials	102.91	86.897

Давайте рассмотрим некоторые характеристики этого датафрейма. Он должен состоять из 500 строк данных. Это можно проверить, посмотрев длину датафрейма:

In[16]:
```
# сколько строк данных? должно быть 500
len(sp500)
```

Out[16]:
```
500
```

Согласно нашим ожиданиям датафрейм имеет 500 строк и три столбца:

```
In[17]:
# какова форма?
sp500.shape

Out[17]:
(500, 3)
```

Размер датафрейма можно найти с помощью свойства `.size`. Это свойство возвращает количество значений данных в датафрейме. Мы ожидаем 500 * 3 = 1500 значений:

```
In[18]:
# каков размер?
sp500.size

Out[18]:
1500
```

Индекс датафрейма состоит из сокращенных названий 500 компаний:

```
In[19]:
# исследуем индекс
sp500.index

Out[19]:
Index(['MMM', 'ABT', 'ABBV', 'ACN', 'ACE', 'ACT', 'ADBE', 'AES', 'AET', 'AFL',
       ...
       'XEL', 'XRX', 'XLNX', 'XL', 'XYL', 'YHOO', 'YUM', 'ZMH', 'ZION', 'ZTS'],
      dtype='object', name='Symbol', length=500)
```

Столбцы состоят из следующих трех имен:

```
In[20]:
# получаем столбцы
sp500.columns

Out[20]:
Index(['Sector', 'Price', 'Book Value'], dtype='object')
```

Обратите внимание, что хотя мы указали четыре столбца при считывании данных, результат содержит только три столбца, потому что для индекса использовался один из четырех столбцов исходного файла.

Доступ к данным внутри объекта DataFrame

Датафрейм состоит из строк и столбцов и имеет функционал, который позволяет отбирать данные по определенным строкам и столбцам. Для этих задач в объекте `DataFrame` используются те же самые операторы, что и в объекте `Series`, в том числе `[]`, `.loc[]` и `.iloc[]`.

Поскольку теперь мы работаем с несколькими измерениями, процесс применения этих операторов немного отличается. Мы рассмотрим их, сначала изучив отбор столбцов, затем отбор строк, комбинирование отбора строк и отбора столбцов в одном операторе, а также логический отбор. Кроме того, библиотека pandas предлагает функционал для отбора отдельного скалярного значения в определенной строке и определенном столбце. Этот метод важен и его наличие обусловлено тем, что он является высокоэффективным способом доступа к этим значениям.

Отбор столбцов в объекте DataFrame

Отбор определенных столбцов в объекте `DataFrame` выполняется с помощью оператора `[]`. Это отличается от отбора в объекте `Series`, где `[]` задавал строки. Оператору `[]` можно передать либо один объект, либо список объектов, представляющий собой извлекаемые столбцы. Следующий программный код извлекает столбец под названием `'Sector'`:

In[21]:
```
# извлекаем столбец Sector
sp500['Sector'].head()
```

Out[21]:
```
Symbol
MMM                Industrials
ABT                Health Care
ABBV               Health Care
ACN     Information Technology
ACE                 Financials
Name: Sector, dtype: object
```

Когда из объекта `DataFrame` извлекается один столбец, то результатом будет объект `Series`:

In[22]:
```
type(sp500['Sector'])
```

Out[22]:
```
pandas.core.series.Series
```

Извлечь несколько столбцов можно с помощью списка имен столбцов.

In[23]:
```
# извлекаем столбцы Price и Book Value
sp500[['Price', 'Book Value']].head()
```

Out[23]:

Symbol	Price	Book Value
MMM	141.14	26.668
ABT	39.60	15.573
ABBV	53.95	2.954
ACN	79.79	8.326
ACE	102.91	86.897

Поскольку извлекаем несколько столбцов, результатом будет объект `DataFrame` вместо объекта `Series`:

In[24]:
```
# покажем, что результат является объектом DataFrame
type(sp500[['Price', 'Book Value']])
```

```
Out[24]:
pandas.core.frame.DataFrame
```

Кроме того, столбцы можно извлечь с помощью атрибутивного доступа. Объект `DataFrame` получает свойства, которые представляют собой имена столбцов, если имя столбца не содержит пробелов. Следующий программный код извлекает столбец `Price` вышеописанным образом:

```
In[25]:
# атрибутивный доступ к столбцу по имени
sp500.Price
```

```
Out[25]:
Symbol
MMM     141.14
ABT      39.60
ABBV     53.95
ACN      79.79
ACE     102.91
         ...
YHOO     35.02
YUM      74.77
ZMH     101.84
ZION     28.43
ZTS      30.53
Name: Price, Length: 500, dtype: float64
```

Обратите внимание, что для столбца `Book Value` аналогичный программный код не сработает, потому что имя столбца содержит пробел.

Отбор строк в объекте DataFrame

Строки можно отбирать с помощью значения метки индекса, воспользовавшись свойством `.loc[]`:

```
In[26]:
# получаем строку с меткой индекса MMM,
# которая возвращается в виде объекта Series
sp500.loc['MMM']
```

```
Out[26]:
Sector        Industrials
Price              141.14
Book Value         26.668
Name: MMM, dtype: object
```

Кроме того, несколько строк можно получить с помощью списка меток:

```
In[27]:
# получаем строки MMM и MSFT
# результатом будет объект DataFrame
sp500.loc[['MMM', 'MSFT']]
```

```
Out[27]:
                          Sector   Price  Book Value
Symbol
MMM                   Industrials  141.14      26.668
MSFT    Information Technology    40.12      10.584
```

Строки можно извлечь по позиции с помощью `.iloc[]`:

In[28]:
```
# получаем строки, имеющие позиции 0 и 2
sp500.iloc[[0, 2]]
```

Out[28]:
```
             Sector   Price  Book Value
Symbol
MMM      Industrials  141.14      26.668
ABBV     Health Care   53.95       2.954
```

Можно найти позицию определенной метки индекса, а затем использовать эту информацию для извлечения строки по позиции:

In[29]:
```
# получаем позиции меток MMM и A в индексе
i1 = sp500.index.get_loc('MMM')
i2 = sp500.index.get_loc('A')
(i1, i2)
```

Out[29]:
```
(0, 10)
```

In[30]:
```
# и извлекаем строки
sp500.iloc[[i1, i2]]
```

Out[30]:
```
             Sector   Price  Book Value
Symbol
MMM      Industrials  141.14      26.668
A        Health Care   56.18      16.928
```

В заключение заметим, что эти операции еще можно было выполнить с помощью свойства `.ix[]`. Однако данный способ объявлен устаревшим. Для получения дополнительной информации см. http://pandas.pydata.org/pandas-docs/stable/indexing.html#different-choices-for-indexing.

Поиск скалярного значения по метке и позиции с помощью .at[] и .iat[]

Отдельные скалярные значения можно найти по метке с помощью свойства `.at[]`, передав ему одновременно метку строки и метку (имя) столбца:

In[31]:
```
# ищем скалярное значение по метке строки
# и метке (имени) столбца
sp500.at['MMM', 'Price']
```

Out[31]:
```
141.14
```

Кроме того, скалярные значения можно найти по позиции с помощью свойства `.iat[]`, передав позицию строки, а затем позицию столбца. Это предпочтительный метод доступа к отдельным значениям и дает максимальную производительность:

```
In[32]:
# ищем скалярное значение по позиции строки
# и позиции столбца
# извлекаем значение в строке 0, столбце 1
sp500.iat[0, 1]

Out[32]:
141.14
```

Создание среза датафрейма с помощью оператора []

Создание среза датафрейма по его индексу синтаксически идентичен выполнению аналогичной операции с серией. Поэтому в этом разделе мы не будем вдаваться в подробности различных вариантов срезов и рассмотрим лишь несколько типичных способов, применяемых к объекту `DataFrame`.

При создании среза с помощью оператора `[]` срез формируется по индексу, а не по столбцам. Следующий программный код извлекает первые пять строк:

```
In[33]:
# первые пять строк
sp500[:5]

Out[33]:
                        Sector   Price  Book Value
Symbol
MMM                 Industrials  141.14      26.668
ABT                 Health Care   39.60      15.573
ABBV                Health Care   53.95       2.954
ACN     Information Technology   79.79       8.326
ACE                  Financials  102.91      86.897
```

А следующий программный код возвращает строки, начиная с метки `ABT` и заканчивая меткой `ACN`:

```
In[34]:
# строки, начиная с метки ABT и заканчивая меткой ACN
sp500['ABT':'ACN']

Out[34]:
                        Sector  Price  Book Value
Symbol
ABT                 Health Care  39.60      15.573
ABBV                Health Care  53.95       2.954
ACN     Information Technology  79.79       8.326
```

Кроме того, создание среза объекта `DataFrame` можно выполнить с помощью свойств `.iloc[]` и `.loc[]`. Использование этих свойств считается лучшей практикой.

Логический отбор строк

Строки можно извлекать с помощью логического отбора. Применительно к датафрейму логический отбор может извлекать данные из нескольких столбцов. Рассмотрим следующий запрос, который находит все акции с ценой менее `100`:

```
In[35]:
# какие строки имеют значения Price < 100?
```

```
sp500.Price < 100
```

Out[35]:
```
Symbol
MMM     False
ABT      True
ABBV     True
ACN      True
ACE     False
        ...
YHOO     True
YUM      True
ZMH     False
ZION     True
ZTS      True
Name: Price, Length: 500, dtype: bool
```

Затем этот результат можно применить к объекту `DataFrame`, воспользовавшись оператором `[]`, который возвращает только те строки, в которых результат был `True`:

In[36]:
```
# теперь получим строки, в которых Price < 100
sp500[sp500.Price < 100]
```

Out[36]:
```
                        Sector  Price  Book Value
Symbol
ABT                Health Care  39.60      15.573
ABBV               Health Care  53.95       2.954
ACN     Information Technology  79.79       8.326
ADBE    Information Technology  64.30      13.262
AES                  Utilities  13.61       5.781
...                        ...    ...         ...
XYL                Industrials  38.42      12.127
YHOO    Information Technology  35.02      12.768
YUM     Consumer Discretionary  74.77       5.147
ZION                Financials  28.43      30.191
ZTS                Health Care  30.53       2.150

[407 rows x 3 columns]
```

С помощью круглых скобок можно объединить несколько условий. Следующий программный код извлекает строки со значениями столбца `Price` от `6` до `10`:

In[37]:
```
# извлекаем лишь те строки, в которых
# значение Price < 10 и > 6
r = sp500[(sp500.Price < 10) &
          (sp500.Price > 6)] ['Price']
r
```

Out[37]:
```
Symbol
HCBK    9.80
HBAN    9.10
SLM     8.82
WIN     9.38
Name: Price, dtype: float64
```

Обычно строки отбирают с использованием нескольких переменных. Следующий программный код демонстрирует это, извлекая все строки,

в которых переменная `Sector` принимает значение `Health Care`, а переменная `Price` больше или равна `100.00`:

In[38]:
```
# извлекаем строки, в которых переменная Sector
# принимает значение Health Care, а переменная
# Price больше или равна 100.00
r = sp500[(sp500.Sector == 'Health Care') &
          (sp500.Price > 100.00)] [['Price', 'Sector']]
r
```

Out[38]:
```
         Price       Sector
Symbol
ACT     213.77  Health Care
ALXN    162.30  Health Care
AGN     166.92  Health Care
AMGN    114.33  Health Care
BCR     146.62  Health Care
...        ...          ...
REGN    297.77  Health Care
TMO     115.74  Health Care
WAT     100.54  Health Care
WLP     108.82  Health Care
ZMH     101.84  Health Care

[19 rows x 2 columns]
```

Одновременный отбор строк и столбцов

Отбор подмножества данных, состоящего из набора строк и столбцов, является общепринятой практикой. Следующий программный код демонстрирует это, сначала отобрав строки, а затем нужные столбцы:

In[39]:
```
# отбираем строки с метками индекса ABT и ZTS
# для столбцов Sector и Price
sp500.loc[['ABT', 'ZTS']][['Sector', 'Price']]
```

Out[39]:
```
              Sector  Price
Symbol
ABT     Health Care  39.60
ZTS     Health Care  30.53
```

Выводы

В этой главе вы узнали, как создавать объекты `DataFrame`, познакомились с различными способами отбора данных на основе индексов и значений в разных столбцах. Эти примеры аналогичны примерам, которые приводились ранее для объекта `Series`, однако они показали, что, поскольку объект `DataFrame` имеет столбцы и соответствующий индекс столбцов, синтаксис при отборе данных в датафрейме отличается от синтаксиса, выполняющего отбор данных в серии. В следующей главе мы еще более подробно рассмотрим различные операции с данными, используя объект `DataFrame` и сосредоточившись на внесении изменений в структуру и содержимое датафрейма.

ГЛАВА 5 ВЫПОЛНЕНИЕ ОПЕРАЦИЙ НАД ОБЪЕКТОМ DATAFRAME И ЕГО СОДЕРЖИМЫМ

Библиотека pandas предлагает мощный функционал по работе с данными, который вы можете использовать в ваших исследовательских проектах. Исследование данных часто предполагает внесение изменений в структуру объектов `DataFrame`, чтобы удалить ненужные данные, изменить формата существующих данных или создать новые переменные на основе данных в других строках или столбцах. В этой главе мы продемонстрируем, как выполнять эти важные операции. В частности, в этой главе мы рассмотрим:

- Переименование столбцов
- Добавление новых столбцов с помощью `[]` и `.insert()`
- Добавление столбцов за счет расширения датафрейма
- Добавление столбцов с помощью конкатенации
- Переупорядочивание столбцов
- Замена содержимого столбцов
- Удаление столбцов
- Добавление новых строк
- Конкатенация строк
- Добавление и замена строк за счет расширения датафрейма
- Удаление строк с помощью `.drop()`
- Удаление строк с помощью логического отбора
- Удаление строк с помощью среза

Настройка библиотеки pandas

Нижеприведенный программный код настраивает среду pandas для наших примеров. Кроме того, он загружает набор данных S&P 500, чтобы его можно было использовать в наших примерах:

```
In[1]:
# импортируем numpy и pandas
import numpy as np
import pandas as pd

# импортируем datetime
import datetime
from datetime import datetime, date

# задаем некоторые настройки pandas, регулирующие
# формат вывода
pd.set_option('display.notebook_repr_html', False)
pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 10)
pd.set_option('display.width', 60)

# считываем данные в DataFrame, используя в качестве
# индекса столбец Symbol и записывая только те
# столбцы, которые имеют позиции 0, 2, 3, 7
sp500 = pd.read_csv("Data/sp500.csv",
                    index_col='Symbol',
                    usecols=[0, 2, 3, 7])
```

Переименование столбцов

Столбцы можно переименовать с помощью соответствующего метода `.rename()`. Этому методу можно передать словарь, в нем ключи представляют собой метки столбцов, которые нужна переименовать, а значения – это новые имена.
Следующий программный код изменит имя столбца `'Book Value'` на `'BookValue'`, удалив пробел.

In[2]:
```
# переименовываем столбец Book Value так, чтобы удалить пробел
# программный код возвращает копию датафрейма с переименованным
# столбцом
newSP500 = sp500.rename(columns=
                        {'Book Value': 'BookValue'})
# печатаем первые 2 строки
newSP500[:2]
```

Out[2]:
```
            Sector   Price  BookValue
Symbol
MMM     Industrials  141.14     26.668
ABT     Health Care   39.60     15.573
```

В итоге такого применения метода `.rename()` возвращается новый датафрейм с переименованным столбцом и данными, скопированными из исходного датафрейма. Следующий программный код подтверждает, что исходный датафрейм не был изменен.

In[3]:
```
# проверяем, не изменились ли имена столбцов
# в исходном датафрейме
sp500.columns
```

Out[3]:
```
Index(['Sector', 'Price', 'Book Value'], dtype='object')
```

Чтобы изменить датафрейм на месте без создания копии, можно воспользоваться параметром `inplace=True`.

In[4]:
```
# этот программный код переименовывает
# столбец на месте
sp500.rename(columns=
             {'Book Value': 'BookValue'},
             inplace=True)
# смотрим, изменилось ли имя столбца
sp500.columns
```

Out[4]:
```
Index(['Sector', 'Price', 'BookValue'], dtype='object')
```

Чтобы теперь взглянуть на данные в столбце, можно воспользоваться свойством `.BookValue`.

In[5]:
```
# и теперь мы можем воспользоваться свойством .BookValue
sp500.BookValue[:5]
```

Out[5]:
```
Symbol
MMM     26.668
ABT     15.573
ABBV     2.954
ACN      8.326
ACE     86.897
Name: BookValue, dtype: float64
```

Добавление новых столбцов с помощью оператора [] и метода .insert()

Добавить новые столбцы в датафрейм можно с помощью оператора `[]`. Давайте добавим новый столбец `RoundedPrice`, который будет представлять собой округленные значения столбца `Price`.

In[6]:
```
# создаем копию, чтобы исходные данные остались в неизменном виде
sp500_copy = sp500.copy()
# добавляем столбец
sp500_copy['RoundedPrice'] = sp500.Price.round()
sp500_copy[:2]
```

Out[6]:

Symbol	Sector	Price	BookValue	RoundedPrice
MMM	Industrials	141.14	26.668	141.0
ABT	Health Care	39.60	15.573	40.0

Библиотека pandas сначала отбирает данные столбца `Price` из исходного датафрейма `sp500`, а затем все значения в этом объекте `Series` округляются. Затем pandas выравнивает этот новый объект `Series` по копии датафрейма `sp500_copy` и добавляет ее как новый столбец под именем `RoundedPrice`. Новый столбец добавляется в конец индекса столбцов.

Чтобы добавить новый столбец в определенную позицию, можно воспользоваться методом `.insert()`. Следующий программный код вставляет столбец `RoundedPrice` между столбцами `Sector` и `Price`:

In[7]:
```
# создаем копию, чтобы исходные данные остались в неизменном виде
copy = sp500.copy()
# вставляем столбец RoundedPrice в качестве
# второго столбца датафрейма
copy.insert(1, 'RoundedPrice', sp500.Price.round())
copy[:2]
```

Out[7]:

Symbol	Sector	RoundedPrice	Price	BookValue
MMM	Industrials	141.0	141.14	26.668
ABT	Health Care	40.0	39.60	15.573

Добавление столбцов за счет расширения датафрейма

Столбец можно добавить с помощью свойства `.loc[]` и среза. Следующий программный код демонстрирует это, добавляя к поднабору sp500 новый столбец `PER`, в котором все значения равны `0`.

In[8]:
```
# создаем копию поднабора/среза
ss = sp500[:3].copy()
# добавляем столбец с нулевыми значениями
ss.loc[:,'PER'] = 0
# смотрим результаты
ss
```

Out[8]:
```
             Sector   Price  BookValue  PER
Symbol
MMM     Industrials  141.14     26.668    0
ABT     Health Care   39.60     15.573    0
ABBV    Health Care   53.95      2.954    0
```

Таким же образом можно добавить серию с уже существующими данными. Следующий программный код добавляет столбец `PER` на основе серии со случайно сгенерированными значениями. Поскольку для этой операции используется выравнивание, необходимо использовать тот же самый индекс, что и в интересующем нас датафрейме.

In[9]:
```
# создаем копию поднабора/среза
ss = sp500[:3].copy()
# добавляем новый столбец со случайно
# сгенерированными значениями
np.random.seed(123456)
ss.loc[:,'PER'] = pd.Series(np.random.normal(size=3), index=ss.index)
# смотрим результаты
ss
```

Out[9]:
```
             Sector   Price  BookValue       PER
Symbol
MMM     Industrials  141.14     26.668  0.469112
ABT     Health Care   39.60     15.573 -0.282863
ABBV    Health Care   53.95      2.954 -1.509059
```

Добавление столбцов с помощью конкатенации

И оператор `[]`, и метод `.insert()` модифицируют датафрейм на месте. Если нужен новый датафрейм с дополнительными столбцами (оставив исходный датафрейм без изменений), можно воспользоваться функцией `pd.concat()`. Эта функция создает новый датафрейм на основе всех указанных объектов `DataFrame`, конкатенированных в порядке перечисления.

Следующий программный код создает новый объект `DataFrame` с одним столбцом, содержащим округленные значения цены. Затем он использует функцию `pd.concat()` с параметром `axis=1`, который указывает, что указанные объекты `DataFrame` должны быть конкатенированы по оси столбцов (если необходима конкатенация по оси строк, нужно использовать параметр `axis=0`).

In[10]:
```
# создаем объект DataFrame с единственным
# столбцом RoundedPrice
rounded_price = pd.DataFrame({'RoundedPrice':
                              sp500.Price.round()})
# конкатенируем по оси столбцов
concatenated = pd.concat([sp500, rounded_price], axis=1)
concatenated[:5]
```

Out[10]:
```
                        Sector   Price  BookValue  \
Symbol
MMM                Industrials  141.14     26.668
ABT                Health Care   39.60     15.573
ABBV               Health Care   53.95      2.954
ACN     Information Technology   79.79      8.326
ACE                 Financials  102.91     86.897

        RoundedPrice
Symbol
MMM            141.0
ABT             40.0
ABBV            54.0
ACN             80.0
ACE            103.0
```

Конкатенация будет рассмотрена более подробно в главе 11 «Комбинирование, сопоставление и переформатирование данных».

В результате конкатенации возможно дублирование имен столбцов. Чтобы продемонстрировать это, давайте заново создадим `rounded_price`, но имя столбца с округленными значениями цен теперь будет `Price`.

In[11]:
```
# создаем объект DataFrame с единственным
# столбцом Price
rounded_price = pd.DataFrame({'Price': sp500.Price.round()})
rounded_price[:5]
```

Out[11]:
```
        Price
Symbol
MMM     141.0
ABT      40.0
ABBV     54.0
ACN      80.0
ACE     103.0
```

В результате конкатенации получаем дублирующиеся имена столбцов.

In[12]:
```
# в результате получаем два столбца Price
dups = pd.concat([sp500, rounded_price], axis=1)
dups[:5]
```

Out[12]:
```
                        Sector   Price  BookValue  Price
Symbol
MMM                Industrials  141.14     26.668  141.0
ABT                Health Care   39.60     15.573   40.0
ABBV               Health Care   53.95      2.954   54.0
ACN     Information Technology   79.79      8.326   80.0
ACE                 Financials  102.91     86.897  103.0
```

Любопытно, что с помощью свойства `.Price` можно извлечь оба столбца.

In[13]:
```
# извлекаем оба столбца Price
dups.Price[:5]
```

Out[13]:

```
         Price  Price
Symbol
MMM     141.14  141.0
ABT      39.60   40.0
ABBV     53.95   54.0
ACN      79.79   80.0
ACE     102.91  103.0
```

Переупорядочивание столбцов

Столбцы можно переупорядочить, отобрав столбцы в нужном порядке. Следующий программный код демонстрирует обратный порядок расположения столбцов.

In[14]:

```
# возвращаем новый объект DataFrame со столбцами,
# расположенными в обратном порядке
reversed_column_names = sp500.columns[::-1]
sp500[reversed_column_names][:5]
```

Out[14]:

```
        BookValue   Price                  Sector
Symbol
MMM        26.668  141.14             Industrials
ABT        15.573   39.60             Health Care
ABBV        2.954   53.95             Health Care
ACN         8.326   79.79  Information Technology
ACE        86.897  102.91              Financials
```

На самом деле нет способа изменить порядок столбцов на месте. См: http://stackoverflow.com/questions/25878198/change-pandas-dataframe-column-order-in-place.

Замена содержимого столбца

Содержимое объекта `DataFrame` можно заменить, присвоив новую серию существующему столбцу с помощью оператора `[]`. Следующий программный код демонстрирует замену столбца `Price` на столбец `Price` из датафрейма `rounded_price`.

In[15]:

```
# операция выполняется на месте, поэтому создадим копию
copy = sp500.copy()
# заменяем данные в столбце Price новыми значениями
# вместо добавления нового столбца
copy.Price = rounded_price.Price
copy[:5]
```

Out[15]:

```
                        Sector  Price  BookValue
Symbol
MMM                 Industrials  141.0     26.668
ABT                 Health Care   40.0     15.573
ABBV                Health Care   54.0      2.954
ACN     Information Technology   80.0      8.326
ACE                  Financials  103.0     86.897
```

Кроме того, значения столбца можно заменить (на месте) с помощью среза.

In[16]:

```
# операция выполняется на месте, поэтому создадим копию
copy = sp500.copy()
# заменяем данные в столбце Price округленными значениями
copy.loc[:,'Price'] = rounded_price.Price
copy[:5]
```

Out[16]:

```
                        Sector  Price  BookValue
Symbol
MMM                 Industrials  141.0     26.668
ABT                 Health Care   40.0     15.573
ABBV                Health Care   54.0      2.954
ACN     Information Technology   80.0      8.326
ACE                  Financials  103.0     86.897
```

Удаление столбцов

Удалить столбцы из объекта `DataFrame` можно с помощью ключевого слова `del` или методов датафрейма `.pop()` или `.drop()`. Работа каждого из них немного отличается:

- `del` просто удаляет серию из объекта `DataFrame` (на месте)
- `pop()` удаляет и возвращает в результате серию (также на месте)
- `drop(labels, axis=1)` возвращает новый датафрейм с удаленным столбцом (исходный объект `DataFrame` не будет изменен)

Следующий программный код демонстрирует применение ключевого слова `del` для удаления столбца `BookValue` из копии исходного датафрейма `sp500`:

In[17]:

```
# пример использования del для удаления столбца
# делаем копию, потому что операция выполняется на месте
copy = sp500.copy()
del copy['BookValue']
copy[:2]
```

Out[17]:

```
            Sector   Price
Symbol
MMM     Industrials  141.14
ABT     Health Care   39.60
```

Следующий программный код использует метод `.pop()` для удаления столбца `Sector`:

```
In[18]:
# пример использования pop для удаления столбца из датафрейма
# делаем копию, потому что операция выполняется на месте
copy = sp500.copy()
# эта строка удалит столбец Sector и возвратит его как серию
popped = copy.pop('Sector')
# столбец Sector удален на месте
copy[:2]

Out[18]:
         Price  BookValue
Symbol
MMM     141.14     26.668
ABT      39.60     15.573
```

Метод `.pop()` имеет то преимущество, что позволяет нам взглянуть на удаленные столбцы.

```
In[19]:
# и у нас есть столбец Sector, полученный
# в результате применения pop
popped[:5]

Out[19]:
Symbol
MMM                 Industrials
ABT                 Health Care
ABBV                Health Care
ACN     Information Technology
ACE                  Financials
Name: Sector, dtype: object
```

Метод `.drop()` можно использовать как для удаления строк, так и для удаления столбцов. Чтобы использовать его для удаления столбцов, укажите `axis=1`:

```
In[20]:
# пример использования drop для удаления столбца из датафрейма
# делаем копию
copy = sp500.copy()
# эта строка вернет новый датафрейм с удаленным столбцом 'Sector'
# копия датафрейма не изменится
afterdrop = copy.drop(['Sector'], axis = 1)
afterdrop[:5]

Out[20]:
         Price  BookValue
Symbol
MMM     141.14     26.668
ABT      39.60     15.573
ABBV     53.95      2.954
ACN      79.79      8.326
ACE     102.91     86.897
```

Присоединение новых строк

Присоединение строк выполняется с помощью метода `.append()` объекта `DataFrame`. Процесс присоединения возвращает новый объект `DataFrame`, в который сначала добавляются строки из исходного датафрейма, а к ним присоединяются строки из второго датафрейма. Присоединение не осуществляет выравнивания и может привести к дублированию меток индексов.

Следующий программный код демонстрирует присоединение двух объектов `DataFrame`, извлеченных из датафрейма `sp500`. Первый датафрейм состоит из строк `0`, `1` и `2` (перечислены по позициям), а второй состоит из строк `10`, `11` и `2` (также перечислены по позициям). Строка в позиции `2` (с меткой `ABBV`) включена в оба датафрейма, чтобы продемонстрировать получение дублирующихся меток индекса.

In[21]:
```
# копируем первые три строки датафрейма sp500
df1 = sp500.iloc[0:3].copy()
# копируем строки в позициях 10, 11 и 2
df2 = sp500.iloc[[10, 11, 2]]
# присоединяем к датафрейму df1 датафрейм df2
appended = df1.append(df2)
# в результате к строкам первого датафрейма
# будут присоединены строки второго датафрейма
appended
```

Out[21]:
```
             Sector   Price  BookValue
Symbol
MMM      Industrials  141.14     26.668
ABT      Health Care   39.60     15.573
ABBV     Health Care   53.95      2.954
A        Health Care   56.18     16.928
GAS        Utilities   52.98     32.462
ABBV     Health Care   53.95      2.954
```

Набор столбцов в датафреймах, использующихся в ходе присоединения, не обязательно должен быть одинаковым. Итоговый датафрейм будет состоять из столбцов, которые были в обоих датафреймах, а столбец, который отсутствовал в одном из датафреймов, для соответствующих наблюдений итогового датафрейма будет содержать значения `NaN`. Следующий программный код демонстрирует это: он создает третий датафрейм, используя тот же самый индекс, что и `df1`, но при этом он имеет единственный столбец с именем, отсутствующим в датафрейме `df1`.

In[22]:
```
# датафрейм df3 использует индекс датафрейма df1
# и у него один столбец PER
# кроме того, это еще и хороший пример
# использования скалярного значения
# для инициализации нескольких строк
df3 = pd.DataFrame(0.0,
                   index=df1.index,
                   columns=['PER'])
df3
```

Out[22]:
```
        PER
Symbol
MMM     0.0
ABT     0.0
ABBV    0.0
```

Теперь давайте присоединим к датафрейму `df1` датафрейм `df3`.

In[23]:
```
# присоединяем к датафрейму df1 датафрейм df3
# каждый датафрейм содержит по три строки, таким
# образом в итоге получим шесть строк
```

```
# df1 не имеет столбца PER, таким образом для строк df1, вошедших
# в итоговый датафрейм, в столбце PER будут получены значения NaN
# df3 не имеет столбцов BookValue, Price и Sector, таким образом для
# строк df3, вошедших в итоговый датафрейм, в столбцах BookValue, Price
# и Sector также будут получены значения NaN
df1.append(df3, sort=True)
```

Обратите внимание на параметр `sort`. Поскольку ось, не участвующая в присоединении (ось столбцов), не выровнена (проще говоря, наши датафреймы имеют неидентичный набор столбцов), мы получим предупреждение о сортировке столбцов: `FutureWarning: Sorting because non-concatenation axis is not aligned. A future version of pandas will change to not sort by default`. В данном случае можно задать значение `False` или `True`, чтобы избежать предупреждения. Если мы зададим значение `False`, столбцы будут упорядочены в том порядке, в каком они следуют в каждом датафрейме: сначала столбцы `Sector`, `Price` и `BookValue` датафрейма `df1`, а затем столбец `PER` датафрейма `df3`. Если мы зададим значение `True` (используется по умолчанию), столбцы будут отсортированы в алфавитном порядке. В будущих версиях библиотеки pandas сортировка столбцов будет отменена.

Out[23]:

Symbol	BookValue	PER	Price	Sector
MMM	26.668	NaN	141.14	Industrials
ABT	15.573	NaN	39.60	Health Care
ABBV	2.954	NaN	53.95	Health Care
MMM	NaN	0.0	NaN	NaN
ABT	NaN	0.0	NaN	NaN
ABBV	NaN	0.0	NaN	NaN

Параметр `ignore_index=True` можно использовать для присоединения, не прибегая к принудительному сохранению индекса из первого или второго датафрейма. Это удобно, когда значения индекса не имеют содержательного смысла и вы просто хотите конкатенировать датафреймы, чтобы в итоге получить датафрейм с последовательно увеличивающимися целочисленными значениями в качестве индекса:

In[24]:

```
# игнорируем метки индекса, создаем индекс по умолчанию
df1.append(df3, ignore_index=True, sort=True)
```

Out[24]:

	BookValue	PER	Price	Sector
0	26.668	NaN	141.14	Industrials
1	15.573	NaN	39.60	Health Care
2	2.954	NaN	53.95	Health Care
3	NaN	0.0	NaN	NaN
4	NaN	0.0	NaN	NaN
5	NaN	0.0	NaN	NaN

Обратите внимание, что полученный объект `DataFrame` имеет стандартный `RangeIndex` и данные прежнего индекса (столбец `Symbol`) полностью исключены из итогового датафрейма. Также обратите внимание на то, что мы использовали сортировку, потому что набор столбцов в конкатенируемых датафреймах не был идентичным.

Конкатенация строк

Строки из нескольких объектов `DataFrame` можно конкатенировать друг с другом с помощью функции `pd.concat()` и параметра `axis=0`. По умолчанию операция `pd.concat()` с параметром `axis=0` для двух объектов `DataFrame` работает так же, как и метод `.append()`.
Следующий программный код демонстрирует конкатенацию двух наборов данных, которые мы использовали ранее, когда рассказывали о присоединении.

In[25]:
```
# копируем первые три строки датафрейма sp500
df1 = sp500.iloc[0:3].copy()
# копируем строки в позициях 10, 11 и 2
df2 = sp500.iloc[[10, 11, 2]]
# передаем их в виде списка
pd.concat([df1, df2])
```

Out[25]:
```
             Sector   Price  BookValue
Symbol
MMM      Industrials  141.14     26.668
ABT      Health Care   39.60     15.573
ABBV     Health Care   53.95      2.954
A        Health Care   56.18     16.928
GAS        Utilities   52.98     32.462
ABBV     Health Care   53.95      2.954
```

Если набор столбцов в конкатенируемых датафреймах не был идентичным, то столбец, который отсутствовал в одном из датафреймов, для соответствующих наблюдений итогового датафрейма будет содержать значения `NaN`.

In[26]:
```
# копируем df2
df2_2 = df2.copy()
# добавляем в df2_2 столбец, которого нет в df1
df2_2.insert(3, 'Foo', pd.Series(0, index=df2.index))
# смотрим df2_2
df2_2
```

Out[26]:
```
             Sector  Price  BookValue  Foo
Symbol
A        Health Care  56.18     16.928    0
GAS        Utilities  52.98     32.462    0
ABBV     Health Care  53.95      2.954    0
```

Давайте сконкатенируем датафреймы `df1` и `df2_2`. Вновь обратите внимание, поскольку ось столбцов, не участвующая в конкатенации, не выровнена (датафреймы имеют неидентичный набор столбцов), нужно явно задать то или иное значение параметра `sort`, чтобы избежать предупреждения.

In[27]:
```
# теперь конкатенируем
pd.concat([df1, df2_2], sort=True)
```

Out[27]:
```
        BookValue  Foo   Price       Sector
Symbol
MMM        26.668  NaN  141.14  Industrials
```

```
ABT        15.573  NaN   39.60  Health Care
ABBV        2.954  NaN   53.95  Health Care
A          16.928  0.0   56.18  Health Care
GAS        32.462  0.0   52.98    Utilities
ABBV        2.954  0.0   53.95  Health Care
```

Если строки копируются из исходных объектов на основе одних и тех же меток, то в результате можно получить дублирующиеся метки индекса. Параметр `keys` позволяет определить, к какому датафрейму относится набор строк. Следующий программный код демонстрирует использование параметра `keys`, который добавляет к индексу уровень, идентифицирующий исходный объект:

In[28]:
```
# задаем ключи
r = pd.concat([df1, df2_2], keys=['df1', 'df2'], sort=True)
r
```

Out[28]:
```
            BookValue  Foo   Price       Sector
    Symbol
df1 MMM        26.668  NaN  141.14  Industrials
    ABT        15.573  NaN   39.60  Health Care
    ABBV        2.954  NaN   53.95  Health Care
df2 A          16.928  0.0   56.18  Health Care
    GAS        32.462  0.0   52.98    Utilities
    ABBV        2.954  0.0   53.95  Health Care
```

Мы рассмотрим иерархические индексы более подробно в главе 6 «Работа с индексами».

Добавление и замена строк за счет расширения датафрейма

Кроме того, добавить строки в объект `DataFrame` можно с помощью свойства `.loc`. С помощью свойства `.loc` мы задаем метку индекса, где должна быть размещена строка. Если метка не существует, значения добавляются в датафрейм с помощью заданной метки индекса. Если метка уже существует, значения в указанной строке заменяются.
Следующий программный код принимает поднабор `sp500` и добавляет строку с меткой `FOO`:

In[29]:
```
# создаем срез датафрейма sp500 и копируем его
ss = sp500[:3].copy()
# создаем новую строку с меткой индекса FOO
# и присваиваем столбцам некоторые значения
# с помощью списка
ss.loc['FOO'] = ['the sector', 100, 110]
ss
```

Out[29]:
```
             Sector   Price  BookValue
Symbol
MMM      Industrials  141.14     26.668
ABT      Health Care   39.60     15.573
ABBV     Health Care   53.95      2.954
FOO       the sector  100.00    110.000
```

Обратите внимание, что данная операция выполняется на месте, добавляя или заменяя строку.

Удаление строк с помощью метода .drop()

Метод `.drop()` объекта `DataFrame` можно использовать для удаления строк. Метод `.drop()` принимает список меток индекса, которые соответствуют удаляемым строкам, и возвращает копию `DataFrame` с удаленными указанными строками.

In[30]:
```
# получаем копию первых 5 строк датафрейма sp500
ss = sp500[:5]
ss
```

Out[30]:
```
                        Sector   Price  BookValue
Symbol
MMM                Industrials  141.14     26.668
ABT                Health Care   39.60     15.573
ABBV               Health Care   53.95      2.954
ACN     Information Technology   79.79      8.326
ACE                 Financials  102.91     86.897
```

In[31]:
```
# удаляем строки с метками ABT и ACN
afterdrop = ss.drop(['ABT', 'ACN'])
afterdrop[:5]
```

Out[31]:
```
             Sector   Price  BookValue
Symbol
MMM      Industrials  141.14     26.668
ABBV     Health Care   53.95      2.954
ACE       Financials  102.91     86.897
```

Удаление строк с помощью логического отбора

Кроме того, для удаления строк из объекта `DataFrame` можно воспользоваться логическим отбором. В результате логический отбор вернет копию датафрейма со строками, в которых выражение принимает значение `True`. Чтобы удалить строки, просто создайте выражение, которое возвращает `False` для строк, предназначенных для удаления, а затем примените выражение к датафрейму.

Следующий программный код демонстрирует удаление строк, в которых значения `Price` больше `300`. Сначала создаем выражение.

In[32]:
```
# определяем строки, в которых Price > 300
selection = sp500.Price > 300
# выводим информацию о количестве строк и
# количестве строк, которые будут удалены
(len(selection), selection.sum())
```

Out[32]:
```
(500, 10)
```

Исходя из этого результата, теперь мы знаем, что существует 10 строк, в которых Price больше 300. Чтобы получить датафрейм, в котором не будет этих 10 строк, применим побитовое отрицание (определение).

In[33]:
```
# для отбора применим побитовое отрицание
# к выражению selection
price_less_than_300 = sp500[~selection]
price_less_than_300
```

Out[33]:

Symbol	Sector	Price	BookValue
MMM	Industrials	141.14	26.668
ABT	Health Care	39.60	15.573
ABBV	Health Care	53.95	2.954
ACN	Information Technology	79.79	8.326
ACE	Financials	102.91	86.897
...	...	...	...
YHOO	Information Technology	35.02	12.768
YUM	Consumer Discretionary	74.77	5.147
ZMH	Health Care	101.84	37.181
ZION	Financials	28.43	30.191
ZTS	Health Care	30.53	2.150

[490 rows x 3 columns]

Удаление строк с помощью среза

Для удаления записей из датафрейма можно воспользоваться срезом. Эта операция похожа на логический отбор, в котором мы выбираем все строки, кроме тех, что нужно удалить.

Предположим, мы хотим удалить из датафрейма `sp500` все строки, кроме первых трех. Срез для выполнения этой задачи имеет вид `[:3]`, возвращая первые три строки.

In[34]:
```
# отбираем лишь первые три строки
only_first_three = sp500[:3]
only_first_three
```

Out[34]:

```
             Sector   Price  BookValue
Symbol
MMM      Industrials  141.14     26.668
ABT      Health Care   39.60     15.573
ABBV     Health Care   53.95      2.954
```

Помните, что поскольку это срез, он связан с исходным датафреймом. Строки не удалены из датафрейма `sp500` и изменения этих трех строк изменят датафрейм `sp500`. Чтобы предотвратить это, сделайте копию среза, которая приведет к созданию нового датафрейма с указанными скопированными строками.

In[35]:

```
# отбираем лишь первые три строки,
# но теперь копируем их
only_first_three = sp500[:3].copy()
only_first_three
```

Out[35]:

```
             Sector   Price  BookValue
Symbol
MMM      Industrials  141.14     26.668
ABT      Health Care   39.60     15.573
ABBV     Health Care   53.95      2.954
```

Выводы

В этой главе вы узнали, как выполнять несколько распространенных операций с данными, используя объект `DataFrame`, в частности операции, которые изменяют структуру `DataFrame`, добавляя или удаляя строки и столбцы. Кроме того, мы рассмотрели, как можно заменить данных в определенных строках и столбцах.

В следующей главе мы более подробно рассмотрим использование индексов, чтобы эффективно извлекать данные из объектов библиотеки pandas.

ГЛАВА 6 ИНДЕКСАЦИЯ ДАННЫХ

Индекс – это инструмент, предназначенный для оптимизации поиска значений в объекте `Series` или объекте `DataFrame`. Он очень похож на ключ реляционной базы данных, но дает гораздо большие возможности. Индекс позволяет выполнить выравнивание для нескольких наборов данных, а также понять, как выполнять различные операции с данными, например, категоризацию данных.

Большая часть процесса подготовки данных, осуществляемого с помощью библиотеки pandas, в решающей степени зависит от того, как вы настроили индексы. Правильно созданный индекс улучшит производительность и станет бесценным инструментом для осуществления анализа.

Ранее мы уже немного использовали индексы, а в этой главе мы рассмотрим их чуть более подробно. Мы осветим следующие темы:

- Важность применения индексов
- Типы индексов библиотеки pandas, включая `RangeIndex`, `Int64Index`, `CategoricalIndex`, `Float64Index`, `Datetimeindex` и `PeriodIndex`
- Установка и сброс индекса
- Создание иерархических индексов
- Отбор данных с использованием иерархических индексов

Настройка библиотеки pandas

Мы воспользуемся стандартными настройками библиотеки pandas, кроме того, мы загрузим набор данных S&P 500, чтобы его можно было использовать в наших примерах:

```
In[1]:
# импортируем numpy и pandas
import numpy as np
import pandas as pd

# импортируем datetime
import datetime
from datetime import datetime, date

# задаем некоторые настройки pandas, регулирующие
# формат вывода
pd.set_option('display.notebook_repr_html', False)
pd.set_option('display.max_columns', 8)
pd.set_option('display.max_rows', 10)
pd.set_option('display.width', 60)

# считываем данные в DataFrame, используя в качестве
# индекса столбец Symbol и записывая только те
# столбцы, которые имеют позиции 0, 2, 3, 7
sp500 = pd.read_csv("Data/sp500.csv",
                    index_col='Symbol',
                    usecols=[0, 2, 3, 7])
```

Важность применения индексов

Индексы библиотеки pandas позволяют эффективно находить значения. Если индекс отсутствует, потребуется линейный поиск по всем нашим

данным. Индексы создают оптимизированные ярлыки для конкретных значений, используя прямой поиск.
Чтобы начать изучение значений индексов, создадим объект `DataFrame` из `10000` случайных чисел:

In[2]:
```
# создаем DataFame, состоящий из случайных чисел и столбца key
np.random.seed(123456)
df = pd.DataFrame({'foo':np.random.random(10000), 'key':range(100, 10100)})
df[:5]
```

Out[2]:
```
        foo  key
0  0.126970  100
1  0.966718  101
2  0.260476  102
3  0.897237  103
4  0.376750  104
```

Предположим, мы хотим найти значение случайного числа для строки, в которой `key==10099` (я явно выбрал это значение, поскольку это последняя строка в датафрейме). Мы можем сделать это, используя логический отбор.

In[3]:
```
# отбираем строку, в котором значение столбца key равно 10099
df[df.key==10099]
```

Out[3]:
```
           foo    key
9999  0.272283  10099
```

Концептуально это просто. Но что, если мы хотим выполнить эту операцию многократно? Эту ситуацию можно смоделировать в Python, используя `%timeit` – magic-функцию оболочки IPython. Следующий программный код осуществляет поиск повторно и сообщает о производительности.

In[4]:
```
# измеряем время выполнения операции отбора
%timeit df[df.key==10099]
```

```
1000 loops, best of 3: 572 µs per loop
```

Согласно результату, три раза выполнено *1000* циклов, а самая быстрая из этих трех попыток осуществляла поиск в среднем за *0,00572* секунды на один цикл (всего *5,72* секунды для операции, состоящей из 1000 циклов).
Теперь давайте попробуем вычислить время выполнения, использовав индекс для поиска значений. Следующий программный код задает индекс датафрейма, который соответствует значениям столбца `key`.

In[5]:
```
# превращаем столбец key в index
df_with_index = df.set_index(['key'])
df_with_index[:5]
```

Out[5]:

```
          foo
key
100  0.126970
101  0.966718
102  0.260476
103  0.897237
104  0.376750
```

И теперь можно найти это значение с помощью свойства `.loc[]`.

```
In[6]:
# теперь можно найти это значение
# с помощью индекса
df_with_index.loc[10099]

Out[6]:
foo    0.272283
Name: 10099, dtype: float64
```

Это лишь обычный поиск. Вычисляем время выполнения с помощью magic-функции `%timeit`.

```
In[7]:
# и теперь операция выполняется намного быстрее
%timeit df_with_index.loc[10099]

10000 loops, best of 3: 115 µs per loop
```

Операция поиска с использованием индекса выполняется примерно в пять раз быстрее. В силу лучшей производительности выполнение поиска по индексу (в тех случаях, когда это возможно) обычно является оптимальным решением. Недостаток использования индекса заключается в том, что потребуется время на его создание, кроме того, он занимает больше памяти.

В ряде случаев вы уже знаете, что из себя должен представлять индекс и вы можете создать его заранее, а затем выполнить поиск. В других случаях сначала потребуется выяснить, как должен выглядеть оптимальный индекс. Кроме того, часто бывает ситуация, когда у вас недостаточно данных или подходящих полей для создания правильного индекса. В этих случаях вам может потребоваться частичный индекс, который возвращает результаты, удовлетворяющие условному выражению, и для этого он выполняет логический отбор данных.

При проведении разведочного анализа лучше всего сначала загрузить данные и исследовать их с помощью запросов/логического отбора. Затем создайте индекс, если ваши данные поддерживают его или если вам требуется повышенная производительность.

Типы индексов библиотеки pandas

Библиотека pandas предлагает массу встроенных индексов. Каждый из этих типов индекса предназначен для оптимизированного поиска на основе определенного типа данных или структуры данных. Давайте рассмотрим несколько широко используемых типов индекса.

Основной тип Index

Этот тип индекса является наиболее общим и представляет собой упорядоченный набор значений, на основе которого можно создать срезы. Значения, которые он содержит, должны быть хешируемыми питоновскими объектами. Это обусловлено тем, что индекс будет использовать хеш для эффективного поиска элементов, связанных с значением этого объекта. Хотя поиск хешей предпочтительнее, чем линейный поиск, существуют и другие типы индексов, которые можно дополнительно оптимизировать.

Обычно таким типом становится индекс столбцов. Следующий программный код демонстрирует использование этого типа индекса в качестве столбцов датафрейма.

```
In[8]:
# покажем, что столбцы фактически являются индексом
temps = pd.DataFrame({ "City": ["Missoula", "Philadelphia"],
                       "Temperature": [70, 80] })
temps
```

```
Out[8]:
          City  Temperature
0     Missoula           70
1  Philadelphia           80
```

```
In[9]:
# мы видим, что столбцы - это индекс
temps.columns
```

```
Out[9]:
Index(['City', 'Temperature'], dtype='object')
```

Хотя этот тип индекса обычно хорошо работает для буквенно-цифровых имен столбцов, при желании в качестве индекса столбцов можно использовать индексы других типов.

Индексы Int64Index и RangeIndex, в качестве меток используются целые числа

`Int64Index` представляет собой иммутабельный массив, состоящий из 64-битовых целых чисел, которые сопоставляются значениям. До недавнего времени это тип индекса применялся в библиотеке pandas по умолчанию в тех случаях, когда индекс не был указан или были использованы целые числа, как показано в следующем фрагменте программного кода:

```
In[10]:
# явно создаем Int64Index
df_i64 = pd.DataFrame(np.arange(10, 20), index=np.arange(0, 10))
df_i64[:5]
```

```
Out[10]:
    0
0  10
1  11
2  12
3  13
4  14
```

In[11]:
```
# смотрим индекс
df_i64.index
```

Out[11]:
```
Int64Index([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype='int64')
```

При использовании этого типа индекса поиск строк в объекте `DataFrame` становится очень эффективным, поскольку он выполняется с помощью непрерывного массива, обрабатываемого в оперативной памяти.
В новых версиях pandas в качестве оптимизации `Int64Index` был добавлен `RangeIndex`. В нем есть возможность создавать индекс на основе целочисленных значений, который имеет конкретное стартовое целочисленное значение, конечное целочисленное значение, а также у него может быть настраиваемый шаг.
Использование стартового значения, конечного значения и шага было общепринятым шаблоном, который заслуживал собственного подкласса в pandas. При использовании этих трех значений память используется рационально и время выполнения такое же, что и при использовании `Int64Index`.
`RangeIndex` стал стандартным индексом для объектов библиотеки pandas. Об этом свидетельствует следующий программный код, которое создает диапазон целочисленных значений, используя по умолчанию `RangeIndex`.

In[12]:
```
# по умолчанию мы получаем RangeIndex
df_range = pd.DataFrame(np.arange(10, 15))
df_range[:5]
```

Out[12]:
```
    0
0  10
1  11
2  12
3  13
4  14
```

In[13]:
```
df_range.index
```

Out[13]:
```
RangeIndex(start=0, stop=5, step=1)
```

Индекс Float64Index, в качестве меток используются числа с плавающей точкой

В индексе `Float64Index` в качестве меток используются числа с плавающей точкой.

In[14]:
```
# индексы, использующие Float64Index
df_f64 = pd.DataFrame(np.arange(0, 1000, 5),
                      np.arange(0.0, 100.0, 0.5))
df_f64.iloc[:5] # потребуется iloc для отбора первых пяти строк
```

Out[14]:
```
      0
0.0   0
0.5   5
1.0  10
```

```
1.5  15
2.0  20
```

In[15]:
```
df_f64.index
```

Out[15]:
```
Float64Index([ 0.0,  0.5,  1.0,  1.5,  2.0,  2.5,  3.0,
               3.5,  4.0,  4.5,
              ...
              95.0, 95.5, 96.0, 96.5, 97.0, 97.5, 98.0,
              98.5, 99.0, 99.5],
             dtype='float64', length=200)
```

Представление дискретных интервалов с использованием IntervalIndex

Отдельные интервалы меток можно представить с помощью `IntervalIndex`. Интервал закрывается либо слева, либо справа, то есть соответствующий конец включается в данный интервал. Согласно принятой в математике нотации интервалов круглая скобка означает, что соответствующий конец не включается (открыт), а квадратная – что включается (закрыт). Следующий программный код показывает создание датафрейма с помощью интервалов, взятых в качестве индекса (интервалы закрыты справа).

In[16]:
```
# датафрейм с IntervalIndex
df_interval = pd.DataFrame({ "A": [1, 2, 3, 4]},
                     index = pd.IntervalIndex.from_breaks(
                         [0, 0.5, 1.0, 1.5, 2.0]))
df_interval
```

Out[16]:
```
            A
(0.0, 0.5]  1
(0.5, 1.0]  2
(1.0, 1.5]  3
(1.5, 2.0]  4
```

In[17]:
```
df_interval.index
```

```
Out[17]:
IntervalIndex([(0.0, 0.5], (0.5, 1.0], (1.0, 1.5], (1.5, 2.0]]
              closed='right',
              dtype='interval[float64]')
```

Категории в качестве индекса – CategoricalIndex

`CategoricalIndex` представляет собой дискретный индекс для базового типа данных `Categorical`. Следующий программный код создает объект `DataFrame` с одним столбцом, имеющим тип `Categorical`.

```
In[18]:
# создаем датафрейм со столбцом, имеющим тип Categorical
df_categorical = pd.DataFrame({'A': np.arange(6),
                               'B': list('aabbca')})
# импортируем класс CategoricalDtype для работы
# с категориальными данными
from pandas.api.types import CategoricalDtype
df_categorical['B'] = df_categorical['B'].astype(CategoricalDtype(categories=list('cab')))
df_categorical
```

```
Out[18]:
   A  B
0  0  a
1  1  a
2  2  b
3  3  b
4  4  c
5  5  a
```

Превратив категориальный столбец (`B`) в индекс датафрейма, мы получили теперь `CategoricalIndex`.

```
In[19]:
# превращаем категориальный столбец в индекс
df_categorical = df_categorical.set_index('B')
df_categorical.index
```

```
Out[19]:
CategoricalIndex(['a', 'a', 'b', 'b', 'c', 'a'], categories=['c', 'a', 'b'], ordered=False,
name='B', dtype='category')
```

Затем можно выполнить поиск, используя уже существующие категории.

```
In[20]:
# ищем значения в категории 'a'
df_categorical.loc['a']
```

```
Out[20]:
   A
B
a  0
a  1
a  5
```

Тип данных `Categorical` будет подробно рассмотрен в главе 7 «Категориальные данные».

Индексация по датам и времени с помощью DatetimeIndex

`DatetimeIndex` используется для представления набора дат и времени. Он широко используются в данных, представленных в виде временных

рядов, когда измерения для наблюдений берутся через определенные промежутки времени. Чтобы вкратце продемонстрировать это, следующий программный код создает диапазон, состоящий из пяти часовых интервалов времени, и использует их в качестве индекса для данной серии.

```
In[21]:
# создаем DatetimeIndex на основе диапазона дат
rng = pd.date_range('5/1/2017', periods=5, freq='H')
ts = pd.Series(np.random.randn(len(rng)), index=rng)
ts

Out[21]:
2017-05-01 00:00:00    1.239792
2017-05-01 01:00:00   -0.400611
2017-05-01 02:00:00    0.718247
2017-05-01 03:00:00    0.430499
2017-05-01 04:00:00    1.155432
Freq: H, dtype: float64

In[22]:
ts.index

Out[22]:
DatetimeIndex(['2017-05-01 00:00:00',
               '2017-05-01 01:00:00',
               '2017-05-01 02:00:00',
               '2017-05-01 03:00:00',
               '2017-05-01 04:00:00'],
              dtype='datetime64[ns]', freq='H')
```

Основное представление даты/времени – это 64-битовое целое число, что делает поиск по дате и времени очень эффективным.

Индексация периодов времени с помощью PeriodIndex

Кроме того, иногда нужно представить в виде меток индекса периоды времени типа дня, месяца или года. Это очень похоже на работу с интервалами, но теперь речь идет о периоде времени. Данные сценарии можно смоделировать, воспользовавшись `PeriodIndex` и задав конкретную частоту периодов в индексе.
Следующий программный код создает три месячных периода, начиная с `2017-01`.

```
In[23]:
# явно создаем PeriodIndex
periods = pd.PeriodIndex(['2017-1', '2017-2', '2017-3'], freq='M')
periods

Out[23]:
PeriodIndex(['2017-01', '2017-02', '2017-03'], dtype='period[M]', freq='M')
```

Затем этот индекс можно использовать внутри объекта `Series` или объекта `DataFrame`.

```
In[24]:
# используем индекс в серии
period_series = pd.Series(np.random.randn(len(periods)),
                          index=periods)
period_series
```

```
Out[24]:
2017-01   -0.449276
2017-02    2.472977
2017-03   -0.716023
Freq: M, dtype: float64
```

Работа с индексами

Рассказав о том, как создать различные типы индекса, давайте теперь рассмотрим несколько распространенных схем использования этих индексов. В частности, мы рассмотрим:

- Создание индекса в объекте `Series` или объекте `DataFrame`
- Способы отбора значений с помощью индекса
- Преобразование данных в индекс и получение данных из индекса
- Переиндексация объекта библиотеки pandas

Создание и использование индекса в объекте Series или объекте DataFrame

Индексы можно создать либо явно, либо позволить библиотеке pandas создать их неявно. Явное создание индекса происходит, когда вы задаете индекс, используя параметр `index` конструктора.
Следующий программный код создает `DatetimeIndex` самостоятельно.

```
In[25]:
# создаем DatetimeIndex
date_times = pd.DatetimeIndex(pd.date_range('5/1/2017',
                                            periods=5,
                                            freq='H'))
date_times

Out[25]:
DatetimeIndex(['2017-05-01 00:00:00',
               '2017-05-01 01:00:00',
               '2017-05-01 02:00:00',
               '2017-05-01 03:00:00',
               '2017-05-01 04:00:00'],
              dtype='datetime64[ns]', freq='H')
```

Вы можете использовать этот индекс самостоятельно или связать его с объектом `Series` или объектом `DataFrame`. Программный код, приведенный ниже, создает объект `DataFrame`, используя уже созданный индекс.

```
In[26]:
# создаем объект DataFrame, используя индекс
df_date_times = pd.DataFrame(np.arange(0, len(date_times)),
                             index=date_times)
df_date_times
```

```
Out[26]:
                     0
2017-05-01 00:00:00  0
2017-05-01 01:00:00  1
2017-05-01 02:00:00  2
2017-05-01 03:00:00  3
2017-05-01 04:00:00  4
```

Кроме того, индекс можно непосредственно присвоить существующему датафрейму или серии, задав свойство `.index`.

```
In[27]:
# задаем индекс датафрейма
df_date_times.index = pd.DatetimeIndex(pd.date_range('6/1/2017',
                                                    periods=5,
                                                    freq='H'))
df_date_times
```

```
Out[27]:
                     0
2017-06-01 00:00:00  0
2017-06-01 01:00:00  1
2017-06-01 02:00:00  2
2017-06-01 03:00:00  3
2017-06-01 04:00:00  4
```

Отбор значений с помощью индекса

Поиск значений по индексу можно выполнить с помощью оператора `[]` или с помощью следующих индексаторов-свойств объекта `Series` или объекта `DataFrame`:

`.loc[]`	Выполняет поиск по метке, а не по позиции. Однако будьте осторожны: если метки являются целыми числами, то целые числа будут обрабатываться как метки!
`.at[]`	Работает как `.loc`, но при этом может извлечь только одно значение.
`.iloc[]`	Выполняет поиск по позиции, начинающейся с `0`, а не по метке индекса.
`.ix[]`	Может выполнять поиск как по метке, так и по позиции. Если задано целое число, будет выполнять поиск по позиции, начинающейся с 0, если задан другой тип данных, будет выполнять поиск по метке. Это свойство объявлено

устаревшим, поэтому используйте в своей работе первые три свойства.

Поиск значений в объекте `Series` можно выполнить с помощью оператора `[]`, как показано в следующем программном коде. который извлекает значение для метки `b`.

```
In[28]:
# создаем серию
s = pd.Series(np.arange(0, 5), index=list('abcde'))
s

Out[28]:
a    0
b    1
c    2
d    3
e    4
dtype: int32

In[29]:
# ищем по метке индекса
s['b']

Out[29]:
1
```

Поиск значений в серии с помощью оператора `[]` эквивалентен использованию свойства `.loc[]`.

```
In[30]:
# явно ищем по метке индекса
s.loc['b']

Out[30]:
1
```

Когда оператор `[]` применяется к датафрейму, он извлекает столбцы вместо строк.

```
In[31]:
# создаем датафрейм с двумя столбцами
df = pd.DataFrame([ np.arange(10, 12),
                    np.arange(12, 14)],
                  columns=list('ab'),
                  index=list('vw'))
df

Out[31]:
    a   b
v  10  11
w  12  13

In[32]:
# эта строка возвращает столбец 'a'
df['a']

Out[32]:
v    10
w    12
Name: a, dtype: int64
```

Чтобы выполнить поиск строки внутри датафрейма, необходимо воспользоваться одним из индексаторов-свойств.

In[33]:
```
# возвращает строку 'w' по метке
df.loc['w']
```

Out[33]:
```
a    12
b    13
Name: w, dtype: int64
```

Кроме того, индексаторы-свойства могут использовать срезы.

In[34]:
```
# создаем срез серии от метки b до метки d
s['b':'d']
```

Out[34]:
```
b    1
c    2
d    3
dtype: int32
```

In[35]:
```
# эта строка явно создает срез от метки b до метки d
s.loc['b':'d']
```

Out[35]:
```
b    1
c    2
d    3
dtype: int32
```

Кроме того, можно передать список значений.

In[36]:
```
# ищем строки по метке
s.loc[['a', 'c', 'e']]
```

Out[36]:
```
a    0
c    2
e    4
dtype: int32
```

Преобразование данных в индекс и получение данных из индекса

С помощью метода `.reset_index()` можно сбросить индекс датафрейма. Чаще всего этот метод используют, когда нужно поместить содержимое индекса в один или несколько столбцов.
Следующий программный код помещает символы в индексе датафрейма `sp500` в столбец и заменяет индекс стандартным целочисленным индексом.

In[37]:
```
# исследуем несколько строк датафрейма sp500
sp500[:5]
```

```
Out[37]:
                          Sector   Price  Book Value
Symbol
MMM                  Industrials  141.14      26.668
ABT                  Health Care   39.60      15.573
ABBV                 Health Care   53.95       2.954
ACN     Information Technology   79.79       8.326
ACE                   Financials  102.91      86.897

In[38]:
# сбрасываем индекс, помещая значения
# индекса в столбец
index_moved_to_col = sp500.reset_index()
index_moved_to_col[:5]

Out[38]:
  Symbol                  Sector   Price  Book Value
0    MMM             Industrials  141.14      26.668
1    ABT             Health Care   39.60      15.573
2   ABBV             Health Care   53.95       2.954
3    ACN  Information Technology   79.79       8.326
4    ACE              Financials  102.91      86.897
```

Столбец данных можно поместить в индекс объекта `DataFrame`, воспользовавшись методом `.set_index()` и указав перемещаемый столбец. Следующий программный код помещает столбец `Sector` в индекс.

```
In[39]:
# а теперь делаем столбец Sector индексом
index_moved_to_col.set_index('Sector')[:5]

Out[39]:
                        Symbol   Price  Book Value
Sector
Industrials                MMM  141.14      26.668
Health Care                ABT   39.60      15.573
Health Care               ABBV   53.95       2.954
Information Technology     ACN   79.79       8.326
Financials                 ACE  102.91      86.897
```

Сформировав иерархический индекс/мультииндекс, можно поместить в индекс несколько столбцов. В этой главе иерархические индексы будут рассмотрены чуть позже.

Переиндексация объекта библиотеки pandas

Объект `DataFrame` можно заново проиндексировать с помощью метода `.reindex()`. Переиндексация обеспечит совместимость датафрейма с новым индексом, выровняв данные, относящиеся к старому индексу, по новому индексу и вставив значения `NaN` там, где невозможно выполнить выравнивание. Программный код, приведенный ниже, демонстрирует повторную индексацию датафрейма `sp500` по трем указанным индексным меткам.

```
In[40]:
# делаем переиндексацию, задав метки MMM, ABBV и FOO
reindexed = sp500.reindex(index=['MMM', 'ABBV', 'FOO'])
# обратите внимание, что ABT и ACN удалены, а FOO содержит значения NaN
reindexed

Out[40]:
            Sector   Price  Book Value
```

```
Symbol
MMM      Industrials  141.14      26.668
ABBV     Health Care   53.95       2.954
FOO              NaN     NaN         NaN
```

Эта операция создает новый объект `DataFrame` с указанными строками. Если для конкретного значения строка не найдена, будет вставлены значения `NaN`, как это видно в случае с меткой `'FOO'`. На самом деле этот метод является хорошим способом фильтрации данных на основе меток индексов.

Кроме того, переиндексацию можно выполнить по столбцам, о чем свидетельствует следующий программный код, выполняющий переиндексацию по трем указанным именам столбцов:

```
In[41]:
# выполняем переиндексацию столбцов
sp500.reindex(columns=['Price',
                       'Book Value',
                       'NewCol'])[:5]
```

```
Out[41]:
         Price  Book Value  NewCol
Symbol
MMM     141.14      26.668     NaN
ABT      39.60      15.573     NaN
ABBV     53.95       2.954     NaN
ACN      79.79       8.326     NaN
ACE     102.91      86.897     NaN
```

Столбец `NewCol` будет содержать значения `NaN`, поскольку он отсутствовал в исходных данных.

Иерархическая индексация

Иерархическая индексация – это инструмент библиотеки pandas, который позволяет комбинировать использование двух или более индексов для каждой строки. Каждый из индексов в иерархическом индексе называется уровнем. Спецификация нескольких уровней в индексе позволяет эффективно отбирать разные подмножества данных, используя разные комбинации значений в каждом уровне. С технической точки зрения индекс библиотеки pandas, который имеет несколько уровней иерархии, называется `MultiIndex`.

Следующий программный код на примере датафрейма sp500 демонстрирует создание `MultiIndex` и получение доступа к данным с помощью `MultiIndex`. Предположим, мы хотим организовать эти данные по значениям столбцов `Sector` и `Symbol`, чтобы в дальнейшем осуществлять эффективный поиск информации на основе комбинаций значений обеих переменных. Мы можем выполнить это с помощью следующего программного кода, который помещает значения столбцов `Sector` и `Symbol` в `MultiIndex`:

```
In[42]:
# сначала помещаем символы в столбец
reindexed = sp500.reset_index()
# а теперь индексируем датафрейм sp500 по столбцам Sector и Symbol
multi_fi = reindexed.set_index(['Sector', 'Symbol'])
```

```
multi_fi[:5]
```

Out[42]:

```
                                Price  Book Value
Sector                 Symbol
Industrials            MMM     141.14      26.668
Health Care            ABT      39.60      15.573
                       ABBV     53.95       2.954
Information Technology ACN      79.79       8.326
Financials             ACE     102.91      86.897
```

Как уже говорилось, объект `MultiIndex` содержит два или более уровня, в данном случае он состоит из двух уровней:

In[43]:

```
# наш индекс - это MultiIndex
type(multi_fi.index)
```

Out[43]:

```
pandas.core.indexes.multi.MultiIndex
```

In[44]:

```
# он имеет два уровня
len(multi_fi.index.levels)
```

Out[44]:

```
2
```

Каждый уровень представляет собой отдельный объект `Index`:

In[45]:

```
# каждый уровень индекса - это индекс
multi_fi.index.levels[0]
```

Out[45]:

```
Index(['Consumer Discretionary', 'Consumer Discretionary ',
       'Consumer Staples', 'Consumer Staples ', 'Energy',
       'Financials', 'Health Care', 'Industrials',
       'Industries', 'Information Technology', 'Materials',
       'Telecommunications Services', 'Utilities'],
      dtype='object', name='Sector')
```

In[46]:

```
# каждый уровень индекса - это индекс
multi_fi.index.levels[1]
```

Out[46]:

```
Index(['A', 'AA', 'AAPL', 'ABBV', 'ABC', 'ABT', 'ACE',
       'ACN', 'ACT', 'ADBE',
       ...
       'XLNX', 'XOM', 'XRAY', 'XRX', 'XYL', 'YHOO', 'YUM',
       'ZION', 'ZMH', 'ZTS'],
      dtype='object', name='Symbol', length=500)
```

Значения самого индекса в определенном уровне для каждой строки можно извлечь с помощью метода `.get_level_values()`:

In[47]:
```
# значения в уровне индекса 0
multi_fi.index.get_level_values(0)
```

Out[47]:
```
Index(['Industrials', 'Health Care', 'Health Care',
       'Information Technology', 'Financials',
       'Health Care', 'Information Technology',
       'Utilities', 'Health Care', 'Financials',
       ...
       'Utilities', 'Information Technology',
       'Information Technology', 'Financials',
       'Industrials', 'Information Technology',
       'Consumer Discretionary', 'Health Care',
       'Financials', 'Health Care'],
      dtype='object', name='Sector', length=500)
```

Чтобы посмотреть значения датафрейма с помощью иерархического индекса, необходимо воспользоваться методом `.xs()`. Этот метод работает аналогично атрибуту `.ix`, но при этом имеет параметры, позволяющие задать многомерный индекс.
Следующий программный код отбирает все элементы с меткой `Industrials` для индекса уровня `0` (`Sector`):

In[48]:
```
# получаем все акции, которые имеют значение Industrials
# обратите внимание, что в результатах
# индекс уровня 0 не выводится
multi_fi.xs('Industrials')[:5]
```

Out[48]:
```
         Price  Book Value
Symbol
MMM     141.14      26.668
ALLE     52.46       0.000
APH      95.71      18.315
AVY      48.20      15.616
BA      132.41      19.870
```

Индекс полученного датафрейма состоит из уровней, которые не были заданы, в данном случае это `Symbol`. Уровни, для которых были указаны значения, удаляются из полученного индекса.
Параметр `level` можно использовать для отбора строк с определенным значением индекса в указанном уровне. Следующий программный код отбирает строки, в которых индекс уровня `1` (`Symbol`) имеет значение `ALLE`.

In[49]:
```
# отбираем строки, в которых индекс уровня 1
# (Symbol) имеет значение ALLE
# обратите внимание, что в результатах
# индекс уровня 1 (Symbol) не выводится
multi_fi.xs('ALLE', level=1)
```

Out[49]:
```
             Price  Book Value
Sector
Industrials  52.46         0.0
```

Чтобы предотвратить удаление уровней из результатов (если вы только не укажете каждый уровень), можно воспользоваться параметром `drop_levels=False`.

```
In[50]:
# отбираем строки, в которых индекс уровня 0
# (Sector) имеет значение Industrials,
# без удаления уровней
multi_fi.xs('Industrials', drop_level=False)[:5]
```

```
Out[50]:
                          Price  Book Value
Sector      Symbol
Industrials MMM      141.14      26.668
            ALLE      52.46       0.000
            APH       95.71      18.315
            AVY       48.20      15.616
            BA       132.41      19.870
```

Чтобы отобрать строку, используя иерархию индексов, вы можете связать вместе вызовы `.xs()` для разных уровней. Следующий программный код отбирает строку со значением `Industrials` в уровне `0` и значением `UPS` в уровне `1`.

```
In[51]:
# скомбинируем уровни индексов
multi_fi.xs('Industrials').xs('UPS')
```

```
Out[51]:
Price         102.73
Book Value      6.79
Name: UPS, dtype: float64
```

Как вариант, можно передать значения каждого уровня иерархического индекса в виде кортежа.

```
In[52]:
# комбинируем уровни индексов, используя кортеж
multi_fi.xs(('Industrials', 'UPS'))
```

```
Out[52]:
Price         102.73
Book Value      6.79
Name: (Industrials, UPS), dtype: float64
```

Обратите внимание, что с помощью метода `.xs()` можно только получать значения, а не устанавливать их.

Выводы

В этой главе мы более подробно рассмотрели использование индексов в библиотеке pandas для организации и извлечения данных. Мы разобрали массу полезных типов индексов и способы их использования с различными типами данных, чтобы получить эффективный доступ к значениям без необходимости запрашивать данные в строках. В заключение мы осветили вопросы использования иерархических индексов для эффективного поиска данных, соответствующих меткам в нескольких индексах и это дает нам мощный инструмент, позволяющий отбирать подмножества данных.

На данный момент мы рассмотрели значительную часть базовых операций по подготовке данных в библиотеке pandas. В следующей главе мы рассмотрим работу с категориальными переменными в pandas.

ГЛАВА 7 КАТЕГОРИАЛЬНЫЕ ДАННЫЕ

Категориальная переменная - это тип переменной в статистике, который представляет собой ограниченный и часто фиксированный набор значений. Этим категориальные переменные отличаются от непрерывных переменных, которые могут представлять бесконечное количество значений. К распространенным категориальным переменным относятся пол (когда есть два значения: мужчина и женщина) или группа крови (например, A, B и 0).

Библиотека pandas позволяет представлять категориальные переменные с помощью объекта `Categorical` библиотеки pandas. Эти объекты предназначены для эффективного представления переменных, которые являются наборами групп (категорий), при этом каждая группа имеет свой целочисленный код. Использование этих кодов дает библиотеке pandas возможность эффективно представлять наборы категорий и выполнять упорядочение и сравнение данных по нескольким категориальным переменным.

В этой главе мы осветим следующие темы, связанные с категориальными переменными:

- Создание категориальных переменных
- Переименование категорий
- Добавление новых категорий
- Удаление категорий
- Удаление неиспользуемых категорий
- Установка категорий
- Описательные статистики
- Частоты значений
- Минимум, максимум и мода
- Как использовать категориальную переменную, чтобы поставить ученикам буквенных оценки на основе их числовых оценок

Настройка библиотеки pandas

Мы начнем рассмотрение примеров в этой главе, задав следующие инструкции для импорта библиотек и настройки вывода:

```
In[1]:
# импортируем numpy и pandas
import numpy as np
import pandas as pd

# импортируем datetime
import datetime
from datetime import datetime, date

# задаем некоторые настройки pandas, регулирующие
# формат вывода
pd.set_option('display.notebook_repr_html', False)
pd.set_option('display.max_columns', 8)
pd.set_option('display.max_rows', 10)
pd.set_option('display.width', 65)

# импортируем matplotlib для построения графиков
import matplotlib.pyplot as plt
```

```
%matplotlib inline
```

Создание категориальных переменных

Для представления категориальной переменной используется объект `Categorical` библиотеки pandas. Категориальная переменная состоит из ограниченного набора значений. Она часто используется для представления значений в виде набора категорий и фиксации количества наблюдений в каждой категории. Еще одно предназначение категориальной переменной заключается в том, чтобы поместить непрерывные значения в дискретный набор буквенных меток, в качестве примера можно привести сопоставление числовых оценок буквенным оценкам. Мы рассмотрим, как выполнить это сопоставление в самом конце главы.

Существует несколько способов создать категориальную переменную. Следующий программный код демонстрирует создание объекта `Categorical` непосредственно из списка:

In[2]:
```
# создаем категориальную переменную непосредственно из списка
lmh_values = ["low", "high", "medium", "medium", "high"]
lmh_cat = pd.Categorical(lmh_values)
lmh_cat
```

Out[2]:
```
[low, high, medium, medium, high]
Categories (3, object): [high, low, medium]
```

Объект `Categorical` создается из списка, состоящего из пяти строк и трех различных значений: `low`, `medium` и `high`. При создании категориальной переменной библиотека pandas определяет каждое уникальное значение в списке и использует его в качестве категории.

Эти категории можно посмотреть с помощью свойства `.categories`:

In[3]:
```
# смотрим категории
lmh_cat.categories
```

Out[3]:
```
Index(['high', 'low', 'medium'], dtype='object')
```

Объект `Categorical` создает индекс, состоящий из трех различных значений в заданном списке. Значения объекта `Categorical` можно получить с помощью метода `.get_values()`:

In[4]:
```
# извлекаем значения
lmh_cat.get_values()
```

Out[4]:
```
array(['low', 'high', 'medium', 'medium', 'high'], dtype=object)
```

Каждой категории объекта `Categorical` присваивается целочисленное значение. Это значение называется **кодом** (code). Коды можно посмотреть с помощью свойства `.codes`.

```
In[5]:
# свойство .codes показывает коды (целочисленные значения)
# для каждого значения категориальной переменной
lmh_cat.codes

Out[5]:
array([1, 0, 2, 2, 0], dtype=int8)
```

Код представляет собой позицию категориального индекса для каждого значения. В данном случае категория `high` соответствует коду `0`, категория `low` – коду `1`, категория `medium` – коду `2`. Этот порядок, возможно, не имеет логического смысла и был определен библиотекой pandas путем серийной (последовательной) обработки строк в массиве `lmh_values`.
Данный порядок можно оптимизировать, указав категории с помощью параметра `categories`.

```
In[6]:
# создаем из списка, но при этом явно указываем категории
lmh_cat = pd.Categorical(lmh_values,
                         categories=["low", "medium", "high"])
lmh_cat

Out[6]:
[low, high, medium, medium, high]
Categories (3, object): [low, medium, high]
```

Обратите внимание на новый порядок категорий, указанный в конструкторе. В силу этого коды выглядят так:

```
In[7]:
# коды выглядят так
lmh_cat.codes

Out[7]:
array([0, 2, 1, 1, 2], dtype=int8)
```

Теперь категория `low` соответствует коду `0`, категория `medium` – коду `1`, категория `high` – коду `2`. Данная кодировка более полезна, так как ее можно использовать для сортировки значений в том порядке, который соответствует содержательному смыслу каждой категории и учитывает ее положение относительно других категорий. Сортировка значений в объекте `Categorical` выполняется на основе кодов, а не фактических значений. Ниже показана сортировка объекта `lmh_cat`:

```
In[8]:
# сортировка выполняется с помощью кодов,
# лежащих в основе каждого значения
lmh_cat.sort_values()

Out[8]:
[low, medium, medium, high, high]
Categories (3, object): [low, medium, high]
```

Кроме того, категориальную переменную можно представить в виде серии, у которой тип данных (`dtype`) обозначается как `"category"`. Следующий программный код берет массив категорий и создает объект `Series` с типом данных `category`:

```
In[9]:
```

```
# создаем категориальную переменную с помощью объекта Series и dtype
cat_series = pd.Series(lmh_values, dtype="category")
cat_series
```

```
Out[9]:
0       low
1      high
2    medium
3    medium
4      high
dtype: category
Categories (3, object): [high, low, medium]
```

Еще один способ создания категориальной переменной состоит в том, чтобы сначала создать объект `Series`, а затем преобразовать столбец с данными в категориальную переменную, используя метод `.astype('category')`. Следующий программный код демонстрирует это, создав серию, а затем преобразовав столбец с данными в категориальную переменную:

```
In[10]:
# создаем категориальную переменную с помощью метода .astype()
s = pd.Series(lmh_values)
as_cat = s.astype('category')
cat_series
```

```
Out[10]:
0       low
1      high
2    medium
3    medium
4      high
dtype: category
Categories (3, object): [high, low, medium]
```

Серия, созданная как объект `Categorical`, имеет свойство `.cat`, которое позволяет получить доступ к категориальной переменной:

```
In[11]:
# категориальная переменная имеет свойство .cat
cat_series.cat
```

```
Out[11]:
<pandas.core.categorical.CategoricalAccessor object at 0x000001396F8DC400>
```

Категориальная переменная `cat_series` является объектом `CategoricalAccessor`, который позволяет получить доступ к различным свойствам соответствующей категориальной переменной. Например, следующий программный код позволяет взглянуть на категории переменной:

```
In[12]:
# получаем индекс категориальной переменной
cat_series.cat.categories

Out[12]:
Index(['high', 'low', 'medium'], dtype='object')
```

Несколько функций библиотеки pandas также возвращают объекты `Categorical`. Одна из них – функция `pd.cut()`, которая создает группы наблюдений, расположенных внутри определенных диапазонов значений. Следующий программный код демонстрирует разбиение серии значений, представляющих из себя 5 случайных чисел от 0 до 100, на 10 групп, каждая из которых имеет ширину 10:

```
In[13]:
# создаем датафрейм из 100 значений
np.random.seed(123456)
values = np.random.randint(0, 100, 5)
bins = pd.DataFrame({ "Values": values})
bins

Out[13]:
   Values
0      65
1      49
2      56
3      43
4      43

In[14]:
# разбиваем значения на 10 групп
bins['Group'] = pd.cut(values, range(0, 101, 10))
bins

Out[14]:
   Values     Group
0      65  (60, 70]
1      49  (40, 50]
2      56  (50, 60]
3      43  (40, 50]
4      43  (40, 50]
```

Столбец `Group` представляет собой категориальную переменную, созданную функцией `cut()`, а каждый уровень индекса представляет собой определенную категорию, с которой было связано данное значение.

```
In[15]:
# проверяем, является ли созданная переменная категориальной
bins.Group

Out[15]:
0    (60, 70]
1    (40, 50]
2    (50, 60]
3    (40, 50]
4    (40, 50]
Name: Group, dtype: category
Categories (10, interval[int64]): [(0, 10] < (10, 20] < (20, 30] < (30, 40] ... (60, 70] <
(70, 80] < (80, 90] < (90, 100]]
```

Явный порядок категорий можно указать с помощью параметра `ordered=True`. Эта настройка означает, что порядок категорий важен и позволяет

сравнивать значения в нескольких сериях, представленных в виде объектов `Categorical`. Чтобы продемонстрировать это, следующий программный код создает упорядоченный объект `Categorical`, представляющий собой три категории металла (и четыре значения для этих трех категорий):

```
In[16]:
# создаем упорядоченную категориальную переменную из
# названий драгоценных металлов
# порядок важен для определения относительной
# ценности металла
metal_values = ["bronze", "gold", "silver", "bronze"]
metal_categories = ["bronze", "silver", "gold"]
metals = pd.Categorical(metal_values,
                        categories=metal_categories,
                        ordered = True)
metals

Out[16]:
[bronze, gold, silver, bronze]
Categories (3, object): [bronze < silver < gold]
```

Созданный объект `Categorical` имеет строго упорядоченные категории: `bronze` имеет меньшую ценность, чем `silver`, `silver` менее ценно, чем `gold`. Этот порядок можно использовать для сортировки или сравнения значений одного объекта `Categorical` со значениями другого. Чтобы продемонстрировать это, давайте создадим еще один объект `Categorical`, в котором меняем значения:

```
In[17]:
# меняем значения
metals_reversed_values = pd.Categorical(
    metals.get_values()[::-1],
    categories = metals.categories,
    ordered=True)
metals_reversed_values

Out[17]:
[bronze, silver, gold, bronze]
Categories (3, object): [bronze < silver < gold]
```

Значения этих двух категориальных переменных можно сравнить друг с другом. Следующий программный код сравнивает ценность металлов в одинаковых индексных метках объектов `Categorical`:

```
In[18]:
# сравниваем значения двух категориальных переменных
metals <= metals_reversed_values

Out[18]:
array([ True, False,  True,  True], dtype=bool)
```

Библиотека pandas выполняет это сравнение, сопоставляя коды значений. Переменная `metals` имеет следующее распределение кодов:

```
In[19]:
# смотрим распределение кодов переменной metals
# коды - это целочисленные значения,
# соответствующие каждому элементу
metals.codes

Out[19]:
array([0, 2, 1, 0], dtype=int8)
```

А переменная `metals_reversed_values` имеет такое распределение кодов:

```
In[20]:
# а теперь смотрим распределение кодов
# переменной metals_reversed_values
metals_reversed_values.codes

Out[20]:
array([0, 1, 2, 0], dtype=int8)
```

Применение логического оператора приведет к ранее показанному результату. В качестве заключительного примера следующий программный код демонстрирует создание объекта `Categorical`, который имеет значение `(copper)`, не относящееся ни к одной из указанных категорий. В данном случае для этого элемента будет получено значение `NaN`.

```
In[21]:
# создаем категориальную переменную со значением, которое
# нельзя отнести ни к одной из категорий, поэтому для него
# будет получено значение NaN
pd.Categorical(["bronze", "copper"],
               categories=metal_categories)

Out[21]:
[bronze, NaN]
Categories (3, object): [bronze, silver, gold]
```

Этот метод можно использовать для фильтрации значений, которые не нужно включать в объект `Categorical` при его создании.

Переименование категорий

Категории объекта `Categorical` можно переименовать, передав новые имена свойству `.categories` или воспользовавшись методом `.rename_categories()`.

```
In[22]:
# создаем категориальную переменную с 3 категориями
cat = pd.Categorical(["a","b","c","a"],
                     categories=["a", "b", "c"])
cat

Out[22]:
[a, b, c, a]
Categories (3, object): [a, b, c]

In[23]:
# переименовываем категории (а также значения)
cat.categories = ["bronze", "silver", "gold"]
cat
Out[23]:
[bronze, silver, gold, bronze]
```

```
Categories (3, object): [bronze, silver, gold]
```

Обратите внимание, что в данном случае речь идет об операции переименования на месте. Чтобы избежать переименования на месте, можно воспользоваться методом `.rename_categories()`.

In[24]:
```
# эта строка тоже переименовывает
cat.rename_categories(["x", "y", "z"])
```

Out[24]:
```
[x, y, z, x]
Categories (3, object): [x, y, z]
```

Мы можем убедиться в том, что операция не была выполнена на месте.

In[25]:
```
# убеждаемся, что переименование не было выполнено на месте
cat
```

Out[25]:
```
[bronze, silver, gold, bronze]
Categories (3, object): [bronze, silver, gold]
```

Добавление категорий

С помощью метода `.add_categories()` можно добавить категории. Следующий программный код добавляет категорию `platinum` в нашу переменную `metals`. Данную операцию невозможно выполнить на месте, если только добавление категорий не задано явно.

In[26]:
```
# добавляем категорию platinum
with_platinum = metals.add_categories(["platinum"])
with_platinum
```

Out[26]:
```
[bronze, gold, silver, bronze]
Categories (4, object): [bronze < silver < gold < platinum]
```

Удаление категорий

С помощью метода `.remove_categories()` можно удалить категории. Удаленные категории заменяются на значения `NaN` с помощью `np.NaN`. Следующий программный код демонстрирует это, удаляя категорию `bronze`:

In[27]:
```
# удаляем категорию bronze
no_bronze = metals.remove_categories(["bronze"])
no_bronze
```

Out[27]:
```
[NaN, gold, silver, NaN]
Categories (2, object): [silver < gold]
```

Удаление неиспользуемых категорий

С помощью метода `.remove_unused_categories()` можно удалить неиспользуемые категории, как показано ниже:

```
In[28]:
# удаляем любые неиспользуемые категории (в данном случае platinum)
with_platinum.remove_unused_categories()

Out[28]:
[bronze, gold, silver, bronze]
Categories (3, object): [bronze < silver < gold]
```

Установка категорий

Кроме того, с помощью метода `.set_categories()` можно сразу добавлять и удалять категории. Следующий программный код создает объект `Series`.

```
In[29]:
# создаем серию
s = pd.Series(["one","two","four", "five"], dtype="category")
s

Out[29]:
0     one
1     two
2    four
3    five
dtype: category
Categories (4, object): [five, four, one, two]
```

Следующий программный код задает категории `"one"` и `"four"`:

```
In[30]:
# удаляем категории "two", "three" и "five" (они заменяются на значения NaN)
s = s.cat.set_categories(["one","four"])
s

Out[30]:
0     one
1     NaN
2    four
3     NaN
dtype: category
Categories (2, object): [one, four]
```

В результате получаем значения `NaN` для несуществующих категорий.

Вычисление описательных статистик для категориальной переменной

Метод `.describe()` вычисляет описательные статистики для объекта `Categorical` точно так же, как для объекта `Series` или `DataFrame`.

```
In[31]:
# получаем описательную информацию
# о категориальной переменной metals
metals.describe()
Out[31]:
           counts  freqs
```

```
categories
bronze          2   0.50
silver          1   0.25
gold            1   0.25
```

В результате получаем количество наблюдений в каждой категории и распределение частот по категориям.
Количество значений для каждой категории можно получить с помощью `.value_counts()`.

In[32]:
```
# подсчитываем количество значений в каждой категории
```

Out[32]:
```
metals.value_counts()
bronze    2
silver    1
gold      1
dtype: int64
```

Минимум, максимум и моду можно вычислить с помощью соответствующих методов.

In[33]:
```
# вычислим минимум, максимум и моду для
# категориальной переменной metals
(metals.min(), metals.max(), metals.mode())
```

Out[33]:
```
('bronze', 'gold', [bronze]
 Categories (3, object): [bronze < silver < gold])
```

Обработка школьных оценок

Теперь давайте рассмотрим применение категориальных переменных, чтобы получить информацию на основе категорий, а не чисел. Задачу, которую мы рассмотрим, - это присвоение ученику буквенной оценки на основе его числовой оценки.

In[34]:
```
# 10 учеников со случайными оценками
np.random.seed(123456)
names = ['Ivana', 'Norris', 'Ruth', 'Lane', 'Skye', 'Sol',
         'Dylan', 'Katina', 'Alissa', "Marc"]
grades = np.random.randint(50, 101, len(names))
scores = pd.DataFrame({'Name': names, 'Grade': grades})
scores
```

Out[34]:
```
   Grade    Name
0     51   Ivana
1     92  Norris
2    100    Ruth
3     99    Lane
4     93    Skye
5     97     Sol
6     93   Dylan
7     77  Katina
8     82  Alissa
9     73    Marc
```

Этот датафрейм содержит исходные оценки для каждого из студентов. Затем мы разбиваем числовые оценки на буквенные коды. Следующий

программный код задает группы для каждой оценки и соответствующую буквенную оценку для каждой группы:

```
In[35]:
# задаем группы и соответствующие буквенные оценки
score_bins =    [ 0,  59,   62,  66,   69,   72,  76,   79,   82,
                  86,   89,   92,  99, 100]
letter_grades = ['F', 'D-', 'D', 'D+', 'C-', 'C', 'C+', 'B-', 'B',
                 'B+', 'A-', 'A', 'A+']
```

С помощью этих значений мы можем выполнить разбиение, которое присвоит буквенные оценки.

```
In[36]:
# разбиваем на основе групп и присваиваем буквенные оценки
letter_cats = pd.cut(scores.Grade, score_bins, labels=letter_grades)
scores['Letter'] = letter_cats
scores
```

```
Out[36]:
   Grade    Name Letter
0     51   Ivana      F
1     92  Norris     A-
2    100    Ruth     A+
3     99    Lane      A
4     93    Skye      A
5     97     Sol      A
6     93   Dylan      A
7     77  Katina     C+
8     82  Alissa     B-
9     73    Marc      C
```

Взглянув на интересующую категориальную переменную, можно понять, как был создан следующий код и как буквенные оценки связаны со значениями:

```
In[37]:
# исследуем интересующую категориальную переменную
letter_cats
```

```
Out[37]:
0     F
1    A-
2    A+
3     A
4     A
5     A
6     A
7    C+
8    B-
9     C
Name: Grade, dtype: category
Categories (13, object): [A < A+ < A- < B ... D < D+ < D- < F]
```

Чтобы определить количество учеников для каждой буквенной оценки, можно воспользоваться `.cat.value_counts()`:

```
In[38]:
# сколько наблюдений имеет каждая оценка?
scores.Letter.value_counts()
```

```
Out[38]:
A     4
F     1
C+    1
C     1
```

```
B-    1
      ..
D+    0
D     0
C-    0
B+    0
B     0
Name: Letter, Length: 13, dtype: int64
```

Поскольку более дальней по алфавиту букве соответствует меньшая числовая оценка, мы можем воспользоваться этим и упорядочить учеников в порядке убывания числовых оценок, отсортировав по категориальному столбцу `Letter`.

In[39]:
```
# сортируем по буквенным оценкам, а не числовым
scores.sort_values(by=['Letter'], ascending=False)
```

Out[39]:

	Grade	Name	Letter
0	51	Ivana	F
7	77	Katina	C+
9	73	Marc	C
8	82	Alissa	B-
1	92	Norris	A-
2	100	Ruth	A+
6	93	Dylan	A
5	97	Sol	A
4	93	Skye	A
3	99	Lane	A

Выводы

В этой главе мы рассмотрели создание категориальных переменных с помощью объектов `Categorical`. Мы начали с обзора способов создания объектов `Categorical`, а также рассмотрели несколько примеров использования целочисленных кодов для представления каждой категории. Затем мы рассмотрели несколько способов модификации категориальной переменной после ее создания. В заключение был приведен примером использования категориальной переменной для разбивки данных на группы – буквенные оценки успеваемости. В следующей главе мы рассмотрим выполнение численного и статистического анализа данных.

ГЛАВА 8 ЧИСЛЕННЫЕ И СТАТИСТИЧЕСКИЕ МЕТОДЫ

Библиотека pandas - очень эффективный инструмент для подготовки данных и выполнений манипуляций с ними, помимо этого она еще предлагает множество средств для проведения численного и статистического анализа. Все эти возможности тесно связаны со структурами данных библиотеки pandas и, таким образом, после подготовки данных выполнение сложных вычислений над ними становится очень простым.

Многие возможности будут рассмотрены в этой главе. Она начинается с обзора обычных численных методов, таких как арифметические операции с выравниванием по нескольким объектам, а также поиск определенных значений, например, минимумов и максимумов. Затем мы рассмотрим различные возможности pandas по выполнению статистического анализа, например, работу с квантилями, ранжирование значений, вычисление дисперсии, анализ корреляции и многое другое.

И последний, но не менее важный пункт: мы рассмотрим очень мощный инструмент pandas, известный под названием «скользящее окно» (rolling window). Суть инструмента «скользящее окно» заключается в многократном применении различных численных и статистических методов (например, вычисление среднего) с определенно заданным шагом движения и шириной «окна». Под «шириной» окна подразумевается подмножество данных. Данные операции необходимы в различных видах анализа, например, чтобы определить некоторые показатели котировок акций по мере их изменения во времени. В этой главе мы только познакомимся с инструментом «скользящее окно», а более подробно мы рассмотрим его в последующих главах.

В этой главе будут освещены следующие темы:

- Выполнение арифметических операций с объектами библиотеки pandas
- Вычисление количества значений
- Определение уникальных значений (и их встречаемости)
- Вычисление минимума и максимума
- Вывод n наименьших и n наибольших значений
- Вычисление накопленных значений
- Получение итоговых описательных статистик
- Измерение центральной тенденции (среднего, медианы и моды)
- Вычисление дисперсии, стандартного отклонения, ковариации и корреляции
- Выполнение дискретизации и квантования данных
- Вычисление ранга значений
- Вычисление процентного изменения для каждого наблюдения серии
- Выполнение операций со скользящим окном
- Создание случайной выборки данных

Настройка библиотеки pandas

Нижеприведенный программный код настраивает среду pandas с помощью импорта необходимых библиотек и опций вывода. Кроме того, он загружает набор данных S&P 500 и исторические данные о котировках акций:

```
In[1]:
# импортируем numpy и pandas
import numpy as np
import pandas as pd

# импортируем datetime
import datetime
from datetime import datetime, date

# задаем некоторые настройки pandas, регулирующие
# формат вывода
pd.set_option('display.notebook_repr_html', False)
pd.set_option('display.max_columns', 7)
pd.set_option('display.max_rows', 10)
pd.set_option('display.width', 60)

# импортируем matplotlib для построения графиков
import matplotlib.pyplot as plt
%matplotlib inline

# считываем данные в DataFrame, используя в качестве
# индекса столбец Symbol и записывая только те
# столбцы, которые имеют позиции 0, 2, 3, 7
sp500 = pd.read_csv("Data/sp500.csv",
                    index_col='Symbol',
                    usecols=[0, 2, 3, 7])

# считываем исторические данные о котировках акций
omh = pd.read_csv("Data/omh.csv")
```

Применение численных методов к объектам библиотеки pandas

Библиотека pandas предлагает богатый набор функций и операций, которые упрощают выполнение арифметических операций и вычисляют различные количественные характеристики данных. В этом разделе мы рассмотрим следующие операции:

- Выполнение арифметических операций над объектами `DataFrame` или `Series`
- Вычисление количества значений
- Определение уникальных значений (и их встречаемости)
- Вычисление минимума и максимума
- Вывод n наименьших и n наибольших значений
- Вычисление накопленных значений

Выполнение арифметических операций над объектами DataFrame или Series

С помощью операторов `+`, `-`, `/` и `*` можно выполнить арифметические операции над объектами `DataFrame` и `Series`. Несмотря на то что эти операции могут показаться тривиальными по своей сути, библиотека pandas делает эффектный финт, выполняя выравнивание значений в

левой и правой частях уравнения. Поэтому индексы играют большую роль при выполнении арифметических операций и пользователь pandas должен понимать, как это может повлиять на результаты.
Кроме того, библиотека pandas предлагает не только стандартные операторы для выполнения арифметических операций, но и предлагает методы `.add()`, `.sub()`, `.mul()` и `.div()`. Они обеспечивают более высокую производительность и гибкость при назначении осей, к которым нужно применить вызываемую функцию.
Арифметические операции с использованием скалярного значения можно применить к каждому элементу объекта `DataFrame`. Чтобы продемонстрировать это, мы создадим объект `DataFrame`, состоящий из четырех столбцов случайных чисел:

In[2]:
```
# задаем стартовое значение генератора случайных чисел
# для получения воспроизводимых результатов
np.random.seed(123456)
# создаем объект DataFrame
df = pd.DataFrame(np.random.randn(5, 4),
                  columns=['A', 'B', 'C', 'D'])
df
```

Out[2]:
```
          A         B         C         D
0  0.469112 -0.282863 -1.509059 -1.135632
1  1.212112 -0.173215  0.119209 -1.044236
2 -0.861849 -2.104569 -0.494929  1.071804
3  0.721555 -0.706771 -1.039575  0.271860
4 -0.424972  0.567020  0.276232 -1.087401
```

По умолчанию любая арифметическая операция будет применена ко всем строкам и столбцам датафрейма. Это приведет к созданию нового объекта `DataFrame`, содержащего полученные результаты, при этом исходный датафрейм останется без изменений:

In[3]:
```
# умножаем все на 2
df * 2
```

Out[3]:
```
          A         B         C         D
0  0.938225 -0.565727 -3.018117 -2.271265
1  2.424224 -0.346429  0.238417 -2.088472
2 -1.723698 -4.209138 -0.989859  2.143608
3  1.443110 -1.413542 -2.079150  0.543720
4 -0.849945  1.134041  0.552464 -2.174801
```

Выполняя операцию над объектом `DataFrame` и объектом `Series`, библиотека pandas выравнивают индекс объекта `Series` по столбцам объекта `DataFrame`, реализуя так называемое **построчное транслирование (row-wise broadcasting)**. Пожалуй, это немного противоречит интуиции, однако достаточно эффективно работает, когда нужно построчно обработать разные значения в каждом столбце.
Чтобы продемонстрировать данную операцию, следующий программный код извлекает первую строку `DataFrame` и затем вычитает ее из каждой

строки, по сути вычисляя разницу между значением каждой строки и значением первой строки:

```
In[4]:
# извлекаем первую строку
s = df.iloc[0]
# вычитаем первую строку из каждой строки объекта DataFrame
diff = df - s
diff

Out[4]:
          A         B         C         D
0  0.000000  0.000000  0.000000  0.000000
1  0.743000  0.109649  1.628267  0.091396
2 -1.330961 -1.821706  1.014129  2.207436
3  0.252443 -0.423908  0.469484  1.407492
4 -0.894085  0.849884  1.785291  0.048232
```

Кроме того, данный процесс работает при изменении порядка, вычитая значения объекта `DataFrame` из значений объекта `Series`, поскольку библиотека pandas достаточно умна, чтобы выяснить правильное применение:

```
In[5]:
# вычитаем объект DataFrame из объекта Series
diff2 = s - df
diff2

Out[5]:
          A         B         C         D
0  0.000000  0.000000  0.000000  0.000000
1 -0.743000 -0.109649 -1.628267 -0.091396
2  1.330961  1.821706 -1.014129 -2.207436
3 -0.252443  0.423908 -0.469484 -1.407492
4  0.894085 -0.849884 -1.785291 -0.048232
```

Набор столбцов, полученных в результате арифметической операции, будет включать индексные метки объекта `Series` и индексные метки столбцов объекта `DataFrame` (в соответствии с правилами выравнивания). Если метка, представляющая столбец, отсутствует либо в объекте `Series`, либо в объекте `DataFrame`, соответствующий столбец будет заполнен значениями `NaN`. Следующий программный код показывает это, создав объект `Series` с помощью индекса, представляющего подмножество столбцов в объекте `DataFrame`, но с дополнительной меткой:

In[6]:
```
# B, C
s2 = s[1:3]
# добавляем E
s2['E'] = 0
# смотрим, как применяется выравнивание
# в этой математической операции
df + s2
```

Out[6]:
```
    A         B         C   D   E
0 NaN -0.565727 -3.018117 NaN NaN
1 NaN -0.456078 -1.389850 NaN NaN
2 NaN -2.387433 -2.003988 NaN NaN
3 NaN -0.989634 -2.548633 NaN NaN
4 NaN  0.284157 -1.232826 NaN NaN
```

Библиотека pandas выравнивает индексные метки `df` по индексным меткам `s2`. Поскольку `s2` не имеет меток `A` или `D` в столбцах, то в результате получим значения `NaN` в этих столбцах. А поскольку `df` не имеет метки `E`, соответствующий столбец также будет заполнен значения `NaN` (несмотря на то что метка `E` присутствовала в объекте `Series`).
При выполнении арифметической операции между двумя объектами `DataFrame` выравнивание происходит как по меткам индекса, так и по меткам столбцов. Следующий программный код извлекает фрагмент `df` и вычитает его из исходного датафрейма. В результате видим, что выровненные значения, полученные в результате вычитания, равны 0, а остальные получают значения `NaN`:

In[7]:
```
# извлекаем строки в позициях с 1-й по 3-ю и только столбцы B и C
subframe = df[1:4][['B', 'C']]
# мы извлекаем небольшой квадрат из середины df
subframe
```

Out[7]:
```
          B         C
1 -0.173215  0.119209
2 -2.104569 -0.494929
3 -0.706771 -1.039575
```

In[8]:
```
# демонстрируем, как происходит выравнивание при
# выполнении операции вычитания
df - subframe
```

Out[8]:
```
    A    B    C   D
0 NaN  NaN  NaN NaN
1 NaN  0.0  0.0 NaN
2 NaN  0.0  0.0 NaN
3 NaN  0.0  0.0 NaN
4 NaN  NaN  NaN NaN
```

С помощью арифметических методов объекта `DataFrame` можно дополнить настроить выполнение арифметических операций. Данные методы позволяют задать конкретную ось. Следующий программный код вычитает значения столбца `A` из значений каждого столбца:

In[9]:
```
# извлекаем столбец A
a_col = df['A']
df.sub(a_col, axis=0)
```

Out[9]:
```
     A         B         C         D
0  0.0 -0.751976 -1.978171 -1.604745
1  0.0 -1.385327 -1.092903 -2.256348
2  0.0 -1.242720  0.366920  1.933653
3  0.0 -1.428326 -1.761130 -0.449695
4  0.0  0.991993  0.701204 -0.662428
```

Вычисление количества значений

Метод `.count()` вычисляет количество элементов, отличных от `NaN`, в объекте `Series`.

In[10]:
```
s = pd.Series(['a', 'a', 'b', 'c', np.NaN])
# подсчитываем количество значений
s.count()
```

Out[10]:
```
4
```

Определение уникальных значений (и их встречаемости)

Список уникальных значений в объекте `Series` можно получить с помощью метода `.unique()`:

In[11]:
```
# возвращает список уникальных элементов
s.unique()
```

Out[11]:
```
array(['a', 'b', 'c', nan], dtype=object)
```

Количество уникальных значений (исключая значения `NaN`) можно получить с помощью метода `.nunique()`:

In[12]:
```
s.nunique()
```

Out[12]:
```
3
```

Чтобы включить значение `NaN` в результаты, используйте параметр `dropna=False`.

In[13]:
```
s.nunique(dropna=False)
```

Out[13]:
```
4
```

Встречаемость каждого уникального значения можно определить с помощью метода `.value_counts()`. Он запускает процесс, еще известный как **гистограммирование (histogramming)**:

In[14]:
```
# вычисляем встречаемость каждого уникального
```

```
# значения для нечисловых данных
s.value_counts(dropna=False)
Out[14]:
a      2
c      1
b      1
NaN    1
dtype: int64
```

Вычисление минимума и максимума

С помощью методов `.min()` и `.max()` можно найти минимальное и максимальное значения.

```
In[15]:
# определяем минимальную цену для обеих акций
omh[['MSFT', 'AAPL']].min()

Out[15]:
MSFT     45.16
AAPL    106.75
dtype: float64

In[16]:
# определяем максимальную цену для обеих акций
omh[['MSFT', 'AAPL']].max()

Out[16]:
MSFT     48.84
AAPL    115.93
dtype: float64
```

Некоторые статистические методы pandas называют косвенными статистиками, поскольку они возвращают не действительные значения, а косвенно связанные с ними значения. Например, методы `.idxmin()` и `.idxmax()` возвращают позиции индекса, в которых находятся минимальное и максимальное значения соответственно.

```
In[17]:
# определяем минимальную цену для обеих акций
omh[['MSFT', 'AAPL']].idxmin()

Out[17]:
MSFT    11
AAPL    11
dtype: int64

In[18]:
# определяем максимальную цену для обеих акций
omh[['MSFT', 'AAPL']].idxmax()

Out[18]:
MSFT    3
AAPL    2
dtype: int64
```

Вычисление n наименьших значений и n наибольших значений

Иногда нам нужно вычислить *n* наименьших значений и *n* наибольших значений в наборе данных. Это можно сделать с помощью методов `.nsmallest()` и `.nlargest()`. Следующий программный код

демонстрирует это, возвращая 4 наименьших значения для переменной MSFT.

```
In[19]:
# вычисляем 4 наименьших значения
omh.nsmallest(4, ['MSFT'])['MSFT']

Out[19]:
11    45.16
12    45.74
21    46.45
10    46.67
Name: MSFT, dtype: float64
```

И аналогично вычисляем 4 наибольших значения.

```
In[20]:
# вычисляем 4 наибольших значения
omh.nlargest(4, ['MSFT'])['MSFT']

Out[20]:
3     48.84
0     48.62
1     48.46
16    48.45
Name: MSFT, dtype: float64
```

Программный код для вычисления *n* наименьших значений и *n* наибольших значений в объекте `Series` выглядит немного по-другому, потому что в серии отсутствуют столбцы, которые нужно указывать.

```
In[21]:
# вычисляем 4 наименьших значения в серии
omh.MSFT.nsmallest(4)

Out[21]:
11    45.16
12    45.74
21    46.45
10    46.67
Name: MSFT, dtype: float64
```

Вычисление накопленных значений

Методы аккумуляции представляют собой статистические методы, которые определяют значение, последовательно применяя следующее значение в серии к полученному результату. Хорошими примерами здесь являются накопленное произведение и накопленная сумма. Следующий программный код демонстрирует вычисление кумулятивного произведения.

```
In[22]:
# вычисляем накопленное произведение
pd.Series([1, 2, 3, 4]).cumprod()

Out[22]:
0     1
1     2
2     6
3    24
dtype: int64
```

Результатом будет еще одна серия, которая представляет собой накопленное значение, вычисленное для каждой позиции индекса. Ниже приводится вычисление накопленной суммы в той же самой серии.

```
In[23]:
# вычисляем накопленную сумму
pd.Series([1, 2, 3, 4]).cumsum()

Out[23]:
0     1
1     3
2     6
3    10
dtype: int64
```

Выполнение статистических операций с объектами библиотеки pandas

Описательная статистика позволяет нам вычислить различные метрики, которые дают представление о конкретных характеристиках исходных данных. В библиотеке pandas есть несколько статистик, которые можно вычислить для объекта `Series` или объекта `DataFrame`.

Давайте рассмотрим несколько аспектов статистического анализа/методов, предлагаемых библиотекой pandas:

- Итоговые описательные статистики
- Измерение центральной тенденции: среднее, медиана и мода
- Дисперсия и стандартное отклонение

Получение итоговых описательных статистик

Объекты библиотеки pandas позволяют воспользоваться методом `.describe()`, который возвращает сводку статистик для данных, записанных в объекте. Когда метод `.describe()` применяется к объекту `DataFrame`, он вычисляет итоговые статистики для каждого столбца. Следующий программный код вычисляет эти статистики для обоих столбцов `MSFT` и `AAPL` в датафрейме `omh`.

```
In[24]:
# получаем сводку статистик для датафрейма
omh.describe()
```

Out[24]:

```
            MSFT        AAPL
count  22.000000   22.000000
mean   47.493182  112.411364
std     0.933077    2.388772
min    45.160000  106.750000
25%    46.967500  111.660000
50%    47.625000  112.530000
75%    48.125000  114.087500
max    48.840000  115.930000
```

С помощью одного быстрого вызова метода мы вычислили количество значений, среднее значение, стандартное отклонение, минимум, максимум, и даже 25-й, 50-й и 75-й процентиль для обеих серий.
Кроме того, метод `.describe()` можно применить к серии. Следующий программный код вычисляет итоговые статистики только для `MSFT`.

In[25]:

```
# вычисляем сводку статистик для MSFT
omh.MSFT.describe()
```

Out[25]:

```
count    22.000000
mean     47.493182
std       0.933077
min      45.160000
25%      46.967500
50%      47.625000
75%      48.125000
max      48.840000
Name: MSFT, dtype: float64
```

А с помощью следующего программного кода можем вычислить для MSFT только среднее значение.

In[26]:

```
# вычисляем для MSFT только среднее значение
omh.MSFT.describe()['mean']
```

Out[26]:

```
47.493181818181817
```

Для нечисловых данных будет возвращен немного другой набор статистик, включая общее количество элементов (`count`), количество уникальных значений (`unique`), наиболее часто встречающееся значение (`top`) и встречаемость этого значения (`freq`):

In[27]:

```
# получаем сводку статистик для нечисловых данных
s = pd.Series(['a', 'a', 'b', 'c', np.NaN])
s.describe()
```

Out[27]:

```
count     4
unique    3
top       a
freq      2
dtype: object
```

Измерение центральной тенденции: среднее, медиана и мода

Среднее и медиана дают нам несколько полезных измерений данных, позволяющих понять распределение значений, а также форму этого распределения. Взаимосвязь между этими тремя значениями позволяет нам получить общее представление о форме распределения, как показано на следующей диаграмме:

Теперь давайте узнаем, как вычислить каждую из этих статистик с помощью библиотеки pandas.

Вычисление среднего значения

Среднее значение измеряет **центральную тенденцию** данных. Оно определяется суммированием всех значений с последующим делением полученного результата на количество значений.

Среднее значение можно вычислить с помощью метода `.mean()`. Следующий программный код вычисляет среднее значение цен для столбцов `AAPL` и `MSFT`:

```
In[28]:
# вычисляем среднее значение для всех
# столбцов в датафрейме omh
omh.mean()

Out[28]:
MSFT     47.493182
AAPL    112.411364
dtype: float64
```

Библиотека pandas берет каждый столбец и отдельно вычисляет для него среднее значение. Она возвращает результаты в виде значений серии, проиндексированной по именам столбцов. По умолчанию метод `.mean()` применяется к оси строк (`axis=0`), то есть двигается сверху вниз по каждому столбцу, чтобы вычислить среднее значение для него. Программный код, приведенный ниже переключает ось строк на ось столбцов (`axis=1`) и, двигаясь слева направо, возвращает цену, усредненную по всем столбцам (названиям акций), для по каждой строки (дня):

```
In[29]:
# вычисляем значение, усредненное по всем столбцам,
```

```
# для каждой строки
omh.mean(axis=1)[:5]
```

```
Out[29]:
0    81.845
1    81.545
2    82.005
3    82.165
4    81.710
dtype: float64
```

Вычисление медианы

Медиана занимает центральное положение в ряду значений. Согласно определению, медиана – это такое значение, которое делит ранжированные данные (отсортированные по возрастанию или убыванию) пополам, то есть ровно половина остальных значений больше него, а другая половина меньше его. Медиана является важной статистикой, потому что на нее в отличие от среднего в меньшей степени влияют выбросы и асимметричность распределения.

Медиану значений можно вычислить с помощью метода `.median()`:

```
In[30]:
# вычисляем медиану значений для каждого столбца
omh.median()
```

```
Out[30]:
MSFT     47.625
AAPL    112.530
dtype: float64
```

Вычисление моды

Мода – это самое часто встречающееся значение в серии и вычисляется с помощью метода `.mode()`. Следующий программный код вычисляет моду для созданной серии.

```
In[31]:
# вычисляем моду для этой серии
s = pd.Series([1, 2, 3, 3, 5])
s.mode()
```

```
Out[31]:
0    3
dtype: int64
```

Обратите внимание, что данная строка возвратила не скалярное значение, представляющее моду, а серию. Данный факт связан с тем, что мод может быть несколько. Это продемонстрировано в следующем примере:

```
In[32]:
# может быть несколько мод
s = pd.Series([1, 2, 3, 3, 5, 1])
s.mode()
```

```
Out[32]:
0    1
1    3
dtype: int64
```

Вычисление дисперсии и стандартного отклонения

В теории вероятностей и статистике стандартное отклонение и дисперсия дают нам представление о том, насколько сильно некоторые значения переменной отличаются от ее среднего значения. Давайте кратко рассмотрим каждый показатель.

Измерение дисперсии

Дисперсия позволяет нам понять, насколько сильно значения отклоняются от среднего. Она определяется следующим образом:

$$s^2 = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2$$

По сути эта формула говорит о том, что для каждого наблюдения мы вычисляем разницу между имеющимся значением и средним значением. Разница может быть положительным или отрицательным значением, поэтому мы возводим ее в квадрат, чтобы убедиться в том, что отрицательные значения оказывают кумулятивный эффект на результат. Затем эти значения суммируются и делятся на количество наблюдений минус 1, в итоге получаем аппроксимированное среднее значение различий.

В библиотеке pandas дисперсия вычисляется с помощью метода `.var()`. Следующий программный код вычисляет дисперсию цены для обеих акций:

```
In[33]:
# вычисляем дисперсию значений в каждом столбце
omh.var()
```

```
Out[33]:
MSFT    0.870632
AAPL    5.706231
dtype: float64
```

Вычисление стандартного отклонения

Стандартное отклонение – это метрика, аналогичная дисперсии. Оно определяется путем вычисления квадратного корня из дисперсии и имеет следующую формулу:

$$s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2}$$

Вспомним, что дисперсия отражает разницу между всеми значениями и средним значением. Поскольку используется возведение в квадрат,

единица измерения дисперсии отличается от единицы измерения фактических значений. Используя квадратный корень из дисперсии, стандартное отклонение измеряется в тех же единицах, что и значения исходного набора данных. Стандартное отклонение вычисляется с использованием метода `.std()`, как показано здесь:

```
In[34]:
# вычисляем стандартное отклонение
omh.std()

Out[34]:
MSFT    0.933077
AAPL    2.388772
dtype: float64
```

Вычисление ковариации и корреляции

Ковариация и корреляция описывают взаимосвязь между двумя переменных. Эта взаимосвязь может быть двух видов:
Переменные связаны прямой зависимостью, если их значения изменяются в одном направлении (большему значению одной переменной соответствует большее значение другой переменной)
Переменные связаны обратной зависимостью, если их значения изменяются в противоположных направлениях (большему значению одной переменной соответствует меньшее значение другой переменной)
И ковариация, и корреляция показывают, является ли взаимосвязь между переменными положительной или отрицательной. Кроме того, корреляция говорит о силе взаимосвязи между переменными.

Вычисление ковариации

Ковариация показывает, как взаимосвязаны две переменные. Положительная ковариация означает, что переменные связаны прямой зависимостью, а отрицательная ковариация указывает на обратную зависимость:

$$cov_{x,y} = \frac{\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y})}{N - 1}$$

Ковариацию можно вычислить с помощью метода `.cov()`. Следующий программный код вычисляет ковариацию между `MSFT` и `AAPL`:

```
In[35]:
# вычисляем ковариацию между MSFT и AAPL
omh.MSFT.cov(omh.AAPL)

Out[35]:
1.9261240259740264
```

Вычисление корреляции

Ковариация позволяет определить, связаны ли значения, однако она не дает понимания, насколько сильно значения переменных изменяются вместе. Чтобы измерить силу, с которой обе переменные меняются

вместе, нам нужно вычислить корреляцию. Корреляция вычисляется путем деления ковариации переменных на произведение стандартных отклонений обеих переменных:

$$r_{x,y} = \frac{cov_{x,y}}{s_x s_y}$$

Корреляция стандартизирует меру взаимосвязи между двумя переменными и, следовательно, показывает, насколько тесно связаны обе переменные. Мера корреляции, называемая коэффициентом корреляции, всегда будет принимать значение от 1 до -1, а интерпретация этого значения такова:

- Если коэффициент корреляции равен *1.0*, переменные имеют идеальную положительную корреляцию. Это означает, что если одна переменная изменяется на заданную величину, вторая переменная изменяется пропорционально в том же самом направлении. Положительный коэффициент корреляции меньше *1.0*, но больше 0.0, указывает на положительную корреляцию, отличную от идеальной. Сила положительной корреляции растет по мере приближения к *1*.
- Если коэффициент корреляции равен *0.0*, между переменными нет взаимосвязи. По изменению одной переменной вы не сможете спрогнозировать изменение другой переменной.
- Если коэффициент корреляции равен *-1.0*, переменные имеют идеальную отрицательную корреляцию (обратно коррелированы) и меняются в противоположных направлениях. Если одна переменная увеличивается, другая переменная пропорционально уменьшается. Отрицательный коэффициент корреляции, превышающий *-1,0*, но меньше 0.0, указывает на отрицательную корреляцию, отличную от идеальной. При этом сила отрицательной корреляции растет по мере приближения к *-1*.

В библиотеке pandas корреляция вычисляется с помощью метода `.corr()`. Следующий программный код вычисляет корреляцию между `MSFT` и `AAPL`.

In[36]:
```
# вычисляем корреляцию между MSFT и AAPL
omh.MSFT.corr(omh.AAPL)
```

Out[36]:
```
0.8641560684381171
```

Из вывода видно, что цены на акции MSFT и APPL в данный период времени демонстрируют высокий уровень корреляции. Это не означает, что они являются каузальными, то есть цена одной акции влияет на цену другой, скорее всего на них действует какой-то общий фактор, например, присутствие в одном и том же секторе рынка.

Дискретизация и квантилизация данных

Дискретизация - это способ разбить непрерывные данные на группы («бины»). Каждое значение затем записывается в соответствующую группу. Получившиеся группы можно использовать для понимания относительного распределения данных по различным бинам.
В библиотеке pandas дискретизация выполняется с помощью функций `pd.cut()` и `pd.qcut()`. Чтобы продемонстрировать дискретизацию, давайте начнем с того, что с помощью обычного генератора случайных чисел создадим набор данных из 10000 случайных чисел:

```
In[37]:
# генерируем 10000 случайных чисел
np.random.seed(123456)
dist = np.random.normal(size = 10000)
dist

Out[37]:
array([ 0.4691123 , -0.28286334, -1.5090585 , ...,  0.26296448,
       -0.83377412, -0.10418135])
```

А следующий программный код вычисляет среднее и стандартное отклонение для нашего набора данных. Мы ожидаем, что по мере увеличения размера выборки среднее будет приближаться к 0, а стандартное отклонение – к 1 (поскольку распределение является нормальным):

```
In[38]:
# выводим среднее и стандартное отклонение
(dist.mean(), dist.std())

Out[38]:
(-0.0028633240409066509, 1.0087162031998911)
```

С помощью функции `pd.cut()` мы можем разбить нашу переменную на группы одинакового размера. Следующий программный код создает пять одинаковых по размеру групп и разносит значения по этим группам:

```
In[39]:
# разбиваем на пять одинаковых по размеру групп
bins = pd.cut(dist, 5)
bins

Out[39]:
[(-0.633, 0.81], (-0.633, 0.81], (-2.077, -0.633], (-2.077, -0.633], (0.81, 2.254], ...,
(-2.077, -0.633], (-0.633, 0.81], (-0.633, 0.81], (-2.077, -0.633], (-0.633, 0.81]]
Length: 10000
Categories (5, interval[float64]): [(-3.528, -2.077] < (-2.077, -0.633] < (-0.633, 0.81] <
(0.81, 2.254] < (2.254, 3.698]]
```

Получившийся в результате категоризированный объект представляет собой категориальную переменную. Он состоит из набора меток и индекса, который описывает способ разбиения данных. Свойство `.categories` возвращает индекс и интервалы, которые были созданы библиотекой pandas с учетом диапазона значений и количества указанных групп:

```
In[40]:
# смотрим категории
bins.categories

Out[40]:
IntervalIndex([(-3.528, -2.077], (-2.077, -0.633], (-0.633, 0.81], (0.81, 2.254],
(2.254, 3.698]]
              closed='right',
              dtype='interval[float64]')
```

Свойство `.codes` возвращает массив, в котором указано, к какой группе (интервалу) был отнесен каждый элемент:

```
In[41]:
# свойство .codes показывает, в какой группе находится
# каждый элемент
bins.codes

Out[41]:
array([2, 2, 1, ..., 2, 1, 2], dtype=int8)
```

Обозначение интервалов соответствует общепринятым математическим соглашениям: круглая скобка указывает на открытый конец интервала, а квадратная скобка – на закрытый конец интервала. Закрытые концы включают соответствующее значение в конце интервала. По умолчанию библиотека pandas использует интервалы, закрытые справа. С помощью параметра `right=False` функции `pd.cut()` можно закрыть интервал слева:

```
In[42]:
# закрываем интервалы слева
pd.cut(dist, 5, right=False).categories

Out[42]:
IntervalIndex([[-3.521, -2.077), [-2.077, -0.633), [-0.633, 0.81), [0.81, 2.254), [2.254,
3.705)]
              closed='left',
              dtype='interval[float64]')
```

Вместо передачи в функцию `pd.cut()` количества групп для дискретизации данных, вы можете передать массив значений, которые будут представлять ширину каждой группы. Наиболее распространенный пример – разбивка значений возраста на возрастные группы. Чтобы продемонстрировать это, следующий программный код генерирует 50 значений возраста в диапазоне от 6 до 45.

```
In[43]:
# генерируем 50 значений возраста в диапазоне от 6 до 45
np.random.seed(123456)
ages = np.random.randint(6, 45, 50)
ages

Out[43]:
array([ 7, 33, 38, 29, 42, 14, 16, 16, 18, 17, 26, 28, 44, 40, 20, 12,  8,
       10, 36, 29, 26, 26, 11, 29, 42, 17, 41, 35, 22, 40, 24, 21, 38, 33,
       26, 23, 16, 34, 26, 20, 18, 42, 27, 13, 37, 37, 10,  7, 10, 23])
```

Мы можем указать диапазоны для групп, передав их в массив, в котором смежные значения будут определять ширину каждой группы. Программный код, приведенный ниже, разбивает данные в указанных значениях и выводит информацию о количестве и частоте значений в каждой группе.

In[44]:
```
# разбиваем на диапазоны и выводим статистику по ним
ranges = [6, 12, 18, 35, 50]
agebins = pd.cut(ages, ranges)
agebins.describe()
```

Out[44]:
```
            counts  freqs
categories
(6, 12]          8   0.16
(12, 18]         9   0.18
(18, 35]        21   0.42
(35, 50]        12   0.24
```

Чтобы задать для каждой группы имя вместо стандартного математического обозначения, используйте параметр `labels`:

In[45]:
```
# добавляем имена для групп
ranges = [6, 12, 18, 35, 50]
labels = ['Youth', 'Young Adult', 'Adult', 'Middle Aged']
agebins = pd.cut(ages, ranges, labels=labels)
agebins.describe()
```

Out[45]:
```
             counts  freqs
categories
Youth             8   0.16
Young Adult       9   0.18
Adult            21   0.42
Middle Aged      12   0.24
```

Присвоение меток удобно не только для текстового вывода, но для графического отображения групп, поскольку библиотека pandas передает названия групп, которые будут размещены на диаграмме.

Кроме того, данные можно разбить в соответствии с указанными квантилями с помощью функции `pd.qcut()`. Эта функция разбивает значения на группы таким образом, чтобы каждая группа имела одинаковое количество элементов. Исходя из этого, мы можем определить ширину интервалов так, чтобы в них попадало одинаковое количество элементов.

Следующий программный код разбивает полученные ранее случайные значения на 5 групп-квантилей:

In[46]:
```
# разбиваем на квантили
# 5 групп с одинаковым количеством элементов
qbin = pd.qcut(dist, 5)
# эта строка выводит диапазоны значений в каждом квантиле
qbin.describe()
```

Out[46]:
```
                counts  freqs
categories
(-3.522, -0.861]    2000    0.2
(-0.861, -0.241]    2000    0.2
(-0.241, 0.261]     2000    0.2
(0.261, 0.866]      2000    0.2
(0.866, 3.698]      2000    0.2
```

Кроме того, вместо целого числа категорий можно указать диапазоны квантилей. Следующий программный код выделяет группы на основе ± 3, 2 и 1 стандартных отклонений. Поскольку эти данные подчиняются нормальному распределению, мы ожидаем, что по каждую сторону от среднего окажутся 0.1%, 2.1%, 13.6% и 34.1% значений.

In[47]:
```
# создаем квантили на основе +/- 3, 2 и 1 стандартных отклонений
quantiles = [0,
             0.001,
             0.021,
             0.5-0.341,
             0.5,
             0.5+0.341,
             1.0-0.021,
             1.0-0.001,
             1.0]
qbin = pd.qcut(dist, quantiles)
# эти данные должны подчиняться идеальному нормальному распределению
qbin.describe()
```

Out[47]:
```
                  counts  freqs
categories
(-3.522, -3.131]      10  0.001
(-3.131, -2.056]     200  0.020
(-2.056, -1.033]    1380  0.138
(-1.033, -0.00363]  3410  0.341
(-0.00363, 1.011]   3410  0.341
(1.011, 2.043]      1380  0.138
(2.043, 3.062]       200  0.020
(3.062, 3.698]        10  0.001
```

Это именно те результаты, которые мы ожидали получить, исходя из данного распределения.

Вычисление ранга значений

Ранжирование помогает нам определить, *проранжирован* ли один из двух элементов *выше или ниже* другого. Ранжирование сокращает количество значений, преобразуя их в последовательность чисел, измеренных в порядковой шкале (ранги). Их можно использовать для оценки сложных критериев, основываясь на полученном порядке.

Чтобы продемонстрировать ранжирование, мы воспользуемся следующей серией данных:

In[48]:
```
# генерируем случайные данные
```

```
np.random.seed(12345)
s = pd.Series(np.random.np.random.randn(5), index=list('abcde'))
s
```

```
Out[48]:
a   -0.204708
b    0.478943
c   -0.519439
d   -0.555730
e    1.965781
dtype: float64
```

Затем эти значения можно проранжировать с помощью метода `.rank()`, который по умолчанию выполняет ранжирование меток от наименьшего значения к наибольшему:

```
In[49]:
# ранжируем значения
s.rank()
```

```
Out[49]:
a    3.0
b    4.0
c    2.0
d    1.0
e    5.0
dtype: float64
```

Результат представляет собой ранги – порядковые номера значений, проставленные в порядке возрастания (по сути, речь идет о сортировке). Самый низкий ранг – это ранг `1.0` с меткой индекса `d` (у которого было самое низкое значение -0.555730), самый высокий ранг – это ранг `5.0` с меткой индекса `e` (значение 1.965781).
Существует множество вариантов ранжирования, например, можно задать пользовательскую функцию ранжирования и способ обработки одинаковых (связанных) рангов.

Вычисление процентного изменения для каждого наблюдения серии

Процентное изменение за определенное количество периодов можно вычислить с помощью метода `.pct_change()`. Пример использования процентного изменения – вычисление скорости изменения цены на акцию. Следующий программный код вычисляет процентное изменение для `MSFT`:

```
In[50]:
# вычисляем %-ное изменение для MSFT
omh[['MSFT']].pct_change()[:5]
```

```
Out[50]:
       MSFT
0       NaN
1 -0.003291
2 -0.007842
3  0.015807
4 -0.008600
```

Выполнение операций со скользящим окном

Библиотека pandas предлагает ряд функций для вычисления перемещающихся (скользящих) статистик. С помощью скользящего окна можно вычислить заданную статистику для определенного интервала данных. Затем окно перемещается по данным с определенным интервалом и пересчитывается заново. Процесс продолжается до тех пор, пока окно не пройдет через весь набор данных.

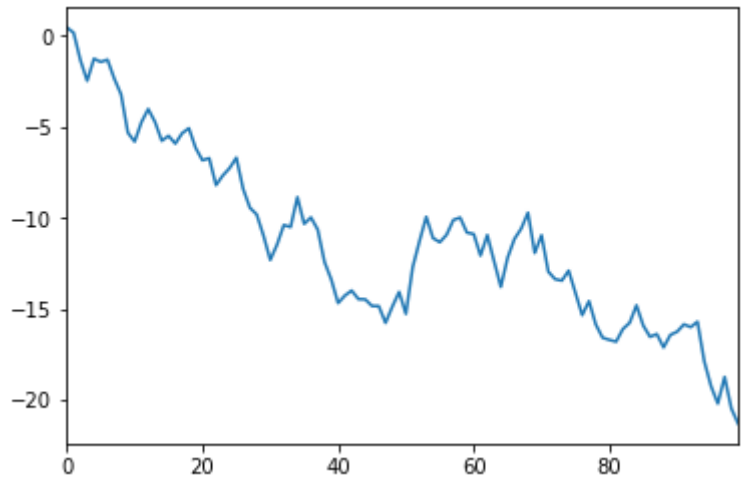
Для демонстрации мы создадим серию из 1000 случайных чисел, которые суммируются нарастающим итогом, чтобы получить случайное блуждание, то есть математическую модель процесса случайных изменений — шагов в дискретные моменты времени.

```
In[51]:
# создаем случайное блуждание
np.random.seed(123456)
s = pd.Series(np.random.randn(1000)).cumsum()
s[:5]
```

```
Out[51]:
0    0.469112
1    0.186249
2   -1.322810
3   -2.458442
4   -1.246330
dtype: float64
```

Сфокусировавшись на первых 100 значениях, мы можем увидеть изменение данных с помощью следующего графика:

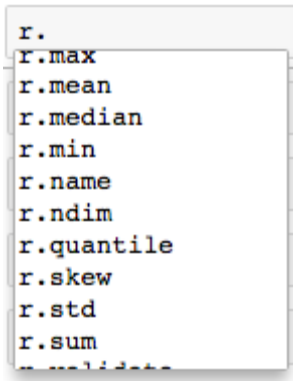
```
In[52]:
s[0:100].plot();
```



Чтобы создать скользящее окна, нам потребуется объект `Rolling`, который мы получаем с помощью метода `.rolling()`, указав ширину окна. В данном случае мы хотим создать скользящее окно шириной `3`:

```
In[53]:
# вычисляем скользящее окно шириной три дня
r = s.rolling(window=3)
r

Out[53]:
Rolling [window=3,center=False,axis=0]
```

Объект `Rolling` задает ширину окна, но при этом он не выполняет фактических вычислений. Чтобы выполнить их, можно выбрать один из многочисленных методов объекта `Rolling`, выполняющий определенную статистической операции (некоторые методы показаны на следующем рисунке):



Следующий программный код демонстрирует вычисление скользящего среднего по имеющимся данным:

```
In[54]:
# скользящее среднее по трем дням
means = r.mean()
means[:7]

Out[54]:
0         NaN
1         NaN
2   -0.222483
3   -1.198334
4   -1.675860
5   -1.708105
6   -1.322070
dtype: float64
```

Поскольку ширина нашего окна N=3, первое среднее значение в выводе вычисляется для индексной метки 2. Мы можем убедиться, что данное значение является средним первых трех чисел:

```
In[55]:
# проверяем, является ли значение средним
# первых трех чисел
s[0:3].mean()

Out[55]:
-0.22248276403642672
```

Затем окно перемещается на один интервал вдоль данных. Следующее значение вычисляется для индексной метки 3 и представляет среднее значений в метках 1, 2 и 3:

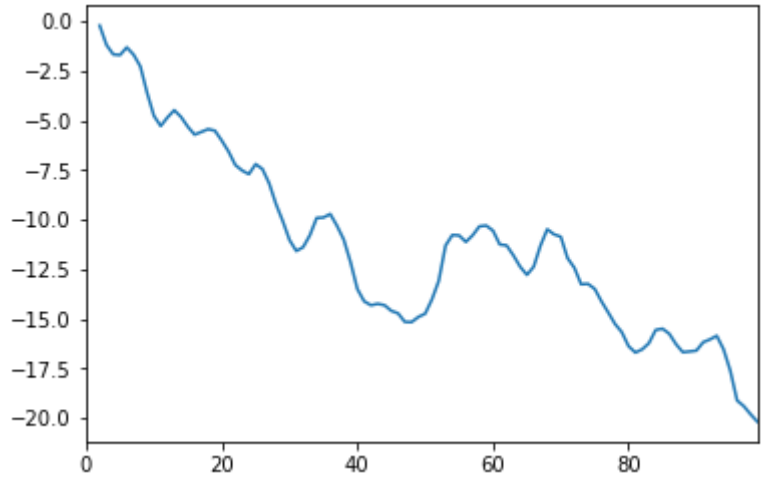
```
In[56]:
# среднее для меток с 1 по 3
```

```
s[1:4].mean()
```

Out[56]:
```
-1.1983341702095498
```

Вычисление скользящего среднего на основе трех дней для первых 100 значений дает нам следующий график:

In[57]:
```
# строим график на основе 3-дневного скользящего среднего
means[0:100].plot();
```



Сравнив этот график с предыдущим, можно увидеть, что скользящее среднее сглаживает данные по интервалам.

Создание случайной выборки данных

Случайная выборка – это процесс случайного отбора значений из набора данных. Начиная с pandas 0.19.2, данный функционал можно применять к объектам `Series` и `DataFrame`, тогда как в предыдущих версиях вам приходилось самостоятельно программировать этот процесс.

Чтобы продемонстрировать создание случайной выборки, давайте начнем с того, что создадим объект `DataFrame`, состоящий из 50 строк и четырех столбцов случайных чисел:

In[58]:
```
# создаем датафрейм, состоящий из 50 строк и 4 столбцов
# случайных чисел
np.random.seed(123456)
df = pd.DataFrame(np.random.randn(50, 4))
df[:5]
```

Out[58]:
```
          0         1         2         3
0  0.469112 -0.282863 -1.509059 -1.135632
1  1.212112 -0.173215  0.119209 -1.044236
2 -0.861849 -2.104569 -0.494929  1.071804
3  0.721555 -0.706771 -1.039575  0.271860
4 -0.424972  0.567020  0.276232 -1.087401
```

Мы можем отобрать данные (т.е. создать выборку данных) с помощью метода `.sample()`, указав количество извлекаемых наблюдений. Следующий программный код отбирает три случайные строки:

In[59]:
```
# отбираем три случайные строки
df.sample(n=3)
```

Out[59]:
```
          0         1         2         3
15 -0.076467 -1.187678  1.130127 -1.436737
28 -2.182937  0.380396  0.084844  0.432390
48 -0.693921  1.613616  0.464000  0.227371
```

Как вариант, можно указать процент случайно отбираемых данных. Программный код, приведенный ниже, извлекает 10% наблюдений (строк).

In[60]:
```
# отбираем 10% строк
df.sample(frac=0.1)
```

Out[60]:
```
          0         1         2         3
37  1.126203 -0.977349  1.474071 -0.064034
10 -1.294524  0.413738  0.276662 -0.472035
4  -0.424972  0.567020  0.276232 -1.087401
14  0.410835  0.813850  0.132003 -0.827317
48 -0.693921  1.613616  0.464000  0.227371
```

В библиотеке pandas можно выполнить случайный отбор с возвращением или без возвращения, при этом по умолчанию используется отбор без возвращения. Чтобы задать случайный отбор с возвращением, просто воспользуемся параметром `replace=True`:

In[61]:
```
# случайный отбор 10% наблюдений с возвращением
df.sample(frac=0.1, replace=True)
```

Out[61]:
```
          0         1         2         3
27 -1.236269  0.896171 -0.487602 -0.082240
9   0.357021 -0.674600 -1.776904 -0.968914
27 -1.236269  0.896171 -0.487602 -0.082240
15 -0.076467 -1.187678  1.130127 -1.436737
9   0.357021 -0.674600 -1.776904 -0.968914
```

Выводы

В этой главе вы узнали о том, как выполнять численный и статистический анализ объектов pandas. Мы рассмотрели массу распространенных методов, которые используются при вычислении значений и проведении различных видов анализа. Мы начали с основных арифметических операций и как выравнивание данных влияет на выполнение операций и получение результатов. Затем мы обсудили множество статистических операций, предлагаемых библиотекой pandas, начиная с описательных статистик и заканчивая дискретизацией, скользящими окнами и случайной выборкой. Эти знания позволят вам эффективно выполнять различные виды анализа данных в реальной практике.

В следующей главе мы сменим тему и рассмотрим способы загрузки данных из различных источников типа локальных файлов, баз данных и удаленных веб-сервисов.

ГЛАВА 9 ЗАГРУЗКА ДАННЫХ

В практически любом реальном анализе данных вам необходимо загрузить данные из внешних источников. Поскольку библиотека pandas построена на базе Python, вы можете использовать любые способы извлечения данных, доступные в языке Python. Это позволяет загрузить данные из почти неограниченного набора источников, включая, но не ограничиваясь, файлы, электронные таблицы Excel, веб-сайты и сервисы, базы данных и облачные сервисы.

Однако при использовании стандартных функций Python, предназначенных для загрузки данных, вам необходимо будет преобразовать объекты Python в объекты `Series` или `DataFrame` библиотеки pandas. Это повысит сложность вашего кода. Для настройки этой сложности библиотека pandas предлагает ряд способов, позволяющих загрузить данные из различных источников непосредственно в объекты библиотеки pandas. Мы рассмотрим многие из этих способов в данной главе.

В частности, в этой главе мы рассмотрим:

- Чтение CSV-файла в датафрейм
- Указание индекса столбца при чтении CSV-файла
- Вывод и спецификация типа данных
- Указание имен столбцов
- Указание конкретных столбцов для загрузки
- Сохранение данных в CSV-файл
- Работа с данными, в которых используются разделители полей
- Обработка загрязненных данных, в которых используются разделители полей
- Чтение и запись данных в формате Excel
- Чтение и запись JSON-файлов
- Чтение HTML-файлов из Интернета
- Чтение и запись HDF5-файлов
- Чтение из базы данных SQL и запись в базу данных SQL
- Загрузка данных о котировках акций с веб-сервисов Yahoo! и Google Finance
- Загрузка данных об опционах с веб-сервиса Google Finance
- Загрузка Базы данных по экономической статистике Федерального резервного банка Сент-Луиса
- Загрузка данных Кеннета Френча
- Загрузка данных Всемирного банка

Настройка библиотеки pandas

Нижеприведенный программный код настраивает среду pandas с помощью импорта необходимых библиотек и опций вывода.

```
In[1]:
# импортируем библиотеки numpy и pandas
import numpy as np
import pandas as pd

# импортируем библиотеку datetime для работы с датами
import datetime
from datetime import datetime, date

# Задаем некоторые опции библиотеки pandas, которые
# настраивают вывод
pd.set_option('display.notebook_repr_html', False)
pd.set_option('display.max_columns', 8)
pd.set_option('display.max_rows', 10)
pd.set_option('display.width', 90)

# импортируем библиотеку matplotlib для построения графиков
import matplotlib.pyplot as plt
%matplotlib inline
```

Работа с CSV-файлами и текстовыми/табличными данными

Данные со значениями, разделенными разделителем (CSV-данные), являются, вероятно, одним из наиболее распространенных форматов данных, который вы будете использовать в библиотеке pandas. Многие веб-сервисы, а также внутрикорпоративные информационные системы предлагают загрузить данные в формате CSV. Это простой формат, который обычно используется для экспорта данных из электронных таблиц типа Excel.

CSV-файл - это файл, который содержит значения, разделенные запятыми. Его можно рассматривать как таблицу данных, аналогичную листу электронной таблицы. Каждая строка данных соответствует строке файла, каждый столбец хранится в текстовом формате, при этом данные в каждом столбце разделены запятыми.

Более подробную информацию о структуре CSV-файлов вы можете узнать по адресу http://en.wikipedia.org/wiki/Comma-separated_values.

Поскольку CSV-формат очень распространен и легко понятен, мы большую часть времени посвятим тому, как считывать и записывать данные библиотеки pandas в этом формате. Знания, полученные в разделе, посвященном CSV, можно будет применить также и к другим форматам, а также они позволят использовать остальные форматы чуть более эффективно.

Исследование CSV-файла

Мы начнем с чтения простого CSV-файла `msft.csv`. Этот файл представляет собой снимок финансовых показателей для акций `MSFT`. Мы воспользуемся модулем `csv` и считаем первые 5 строк.

```
In[2]:
# с помощью модуля csv взглянем
# на первые 5 строк CSV-файла
import csv
with open('Data/msft.csv') as file:
    reader = csv.reader(file, delimiter=',')
    for i,row in enumerate(reader):
        print(row)
        if(i >= 5):
            break
```

```
['Date', 'Open', 'High', 'Low', 'Close', 'Volume']
['7/21/2014', '83.46', '83.53', '81.81', '81.93', '2359300']
['7/18/2014', '83.3', '83.4', '82.52', '83.35', '4020800']
['7/17/2014', '84.35', '84.63', '83.33', '83.63', '1974000']
['7/16/2014', '83.77', '84.91', '83.66', '84.91', '1755600']
['7/15/2014', '84.3', '84.38', '83.2', '83.58', '1874700']
```

Первая строка файла содержит имена всех столбцов, каждый отделен друг от друга запятой. Каждая строка представляет собой набор значений по конкретной дате.

Чтение CSV-файла в датафрейм

Данные в файле `msft.csv` идеально подходят для считывания в датафрейм. Все строки являются полными, не содержат пропущенных значений, в первой строке записаны имена столбцов. Все, что нам нужно сделать для считывания этих данных в объект `DataFrame`, это воспользоваться функцией `pd.read_csv()` библиотеки pandas:

```
In[3]:
# считываем msft.csv в датафрейм
msft = pd.read_csv("Data/msft.csv")
msft[:5]
```

```
Out[3]:
        Date   Open   High    Low  Close   Volume
0  7/21/2014  83.46  83.53  81.81  81.93  2359300
1  7/18/2014  83.30  83.40  82.52  83.35  4020800
2  7/17/2014  84.35  84.63  83.33  83.63  1974000
3  7/16/2014  83.77  84.91  83.66  84.91  1755600
4  7/15/2014  84.30  84.38  83.20  83.58  1874700
```

Ого, так просто! Библиотека pandas поняла, что первая строка файла содержит имена столбцов и считала данные в датафрейм.

Указание индекса столбца при чтении CSV-файла

В предыдущем примере индекс был представлен не в виде дат, а в виде чисел и начинался с `0`. Это связано с тем, что по умолчанию библиотека pandas не предполагает, что какой-то конкретный столбец в файле должен использоваться в качестве индекса. Чтобы разрешить эту ситуацию, вы можете указать, какой столбец (столбцы) должен быть индексом в вызове `read_csv()`. Для этого нужно воспользоваться параметром `index_col` и присвоить ему нулевую позицию столбца, который будет служить индексом.

Следующий программный код считывает данные и сообщает библиотеке pandas, что в качестве индекса нужно использовать столбец, имеющий позицию `0` (столбец `Date`):

```
In[4]:
# используем столбец 0 в качестве индекса
msft = pd.read_csv("Data/msft.csv", index_col=0)
msft[:5]
```

```
Out[4]:
             Open   High    Low  Close   Volume
Date
7/21/2014  83.46  83.53  81.81  81.93  2359300
7/18/2014  83.30  83.40  82.52  83.35  4020800
7/17/2014  84.35  84.63  83.33  83.63  1974000
7/16/2014  83.77  84.91  83.66  84.91  1755600
7/15/2014  84.30  84.38  83.20  83.58  1874700
```

Вывод и спецификация типа данных

Анализ типов столбцов показывает, что библиотека pandas пытается вывести типы столбцов, исходя из их содержимого:

```
In[5]:
# исследуем типы столбцов в этом датафрейме
msft.dtypes
```

```
Out[5]:
Open      float64
High      float64
Low       float64
Close     float64
Volume      int64
dtype: object
```

Чтобы принудительно задать тип столбца, воспользуйтесь параметром dtype функции `pd.read_csv()`. Следующий программный код преобразует столбец `Volume` в тип `float64`:

```
In[6]:
# указываем, что столбец Volume должен иметь тип float64
msft = pd.read_csv("Data/msft.csv",
                   dtype = { 'Volume' : np.float64})
msft.dtypes
```

```
Out[6]:
Date       object
Open      float64
High      float64
Low       float64
Close     float64
Volume    float64
dtype: object
```

Указание имен столбцов

Кроме того, можно указать имена столбцов во время чтения данных, воспользовавшись параметром `names`:

```
In[7]:
# задаем новый набор имен для столбцов
# все имеют нижний регистр,
# header=0 задает строку заголовков
df = pd.read_csv("Data/msft.csv",
                 header=0,
                 names=['date', 'open', 'high', 'low',
                        'close', 'volume'])
df[:5]
```

```
Out[7]:
        date   open   high    low  close   volume
0  7/21/2014  83.46  83.53  81.81  81.93  2359300
1  7/18/2014  83.30  83.40  82.52  83.35  4020800
2  7/17/2014  84.35  84.63  83.33  83.63  1974000
3  7/16/2014  83.77  84.91  83.66  84.91  1755600
4  7/15/2014  84.30  84.38  83.20  83.58  1874700
```

Поскольку мы указали имена столбцов, нам нужно задать строку «имен столбцов» в файле с помощью параметра `header=0`. Если этого не сделать, библиотека pandas предположит, что первая строка является частью данных и это в дальнейшем вызовет некоторые проблемы при обработке информации.

Указание конкретных столбцов для загрузки

Кроме того, можно указать, какие столбцы нужно загружать при чтении файла. Это может быть полезно, если файл содержит много столбцов, некоторые из которых не представляют интереса для вашего анализа, и вы хотите сэкономить время и память, необходимые для чтения и хранения этих данных. Указать конкретные столбцы для загрузки можно с помощью параметра `usecols`, которому можно передан список имен столбцов или индексов столбцов.

Для иллюстрации следующий программный код считывает только столбцы `Date` и `Close` и использует `Date` в качестве индекса:

```
In[8]:
# считываем в данных только столбцы Date и Close
# и индексируем по столбцу Date
df2 = pd.read_csv("Data/msft.csv",
                  usecols=['Date', 'Close'],
                  index_col=['Date'])
df2[:5]
```

```
Out[8]:
            Close
Date
7/21/2014  81.93
7/18/2014  83.35
7/17/2014  83.63
7/16/2014  84.91
7/15/2014  83.58
```

Сохранение датафрейма в CSV-файл

С помощью метода `.to_csv()` объект `DataFrame` можно сохранить в CSV-файл. Чтобы продемонстрировать сохранение данных в CSV-файл, мы сохраним датафрейм `df2` с переименованными именами столбцов в новый файл `msft_modified.csv`:

```
In[9]:
# сохраняем датафрейм df2 в новый csv-файл
# задаем имя индекса как date
df2.to_csv("Data/msft_modified.csv", index_label='date')
```

С помощью параметра `index_label='date'` необходимо указать, что именем индекса будет имя столбца `Date`. В противном случае индекс не получит имени, добавляемого в первую строку файла, и это затруднит правильное чтение данных.
Чтобы убедиться в том, что программный код сработал правильно, мы можем посмотреть содержимое нового файла:

```
In[10]:
# с помощью модуля csv взглянем
# на первые 5 строк CSV-файла
with open("Data/msft_modified.csv") as file:
    reader = csv.reader(file, delimiter=',')
    for i,row in enumerate(reader):
        print(row)
        if(i >= 5):
            break

['date', 'Close']
['7/21/2014', '81.93']
['7/18/2014', '83.35']
['7/17/2014', '83.63']
['7/16/2014', '84.91']
['7/15/2014', '83.58']
```

Работа с данными, в которых используются разделители полей

По сути CSV является конкретной реализацией того, что называется данными с разделителями полей. В подобного рода данных элементы (значения) в каждой строке разделяются определенным символом. В случае с CSV таким разделителем является запятой. Однако существуют и другие символы, например, такой символ | (вертикальная черта). При использовании символа | данные часто называются данными, разделенными вертикальными чертами.
Библиотека pandas предлагает функцию `pd.read_table()` для упрощения чтения данных с разделителями полей. В следующем примере эта функция используется для чтения файла данных `msft`, задав запятую в качестве значения параметра `sep`:

```
In[11]:
# используем функцию read_table с параметром sep=','
# чтобы прочитать CSV-файл
df = pd.read_table("Data/msft.csv", sep=',')
df[:5]
```

Out[11]:

	Date	Open	High	Low	Close	Volume
0	7/21/2014	83.46	83.53	81.81	81.93	2359300
1	7/18/2014	83.30	83.40	82.52	83.35	4020800
2	7/17/2014	84.35	84.63	83.33	83.63	1974000
3	7/16/2014	83.77	84.91	83.66	84.91	1755600
4	7/15/2014	84.30	84.38	83.20	83.58	1874700

У библиотеки pandas нет метода `.to_table()`, аналогичного методу `.to_csv()`. Однако метод `.to_csv()` можно применить для записи данных с разделителями полей, в том числе и в тех случаях, когда в качестве разделителя вместо запятой используется другой символ. В качестве примера следующий программный код записывает данные, где в качестве разделителя используются вертикальные черты:

In[12]:
```
# сохраняем как данные, в которых разделителем
# является вертикальная черта
df.to_csv("Data/msft_piped.txt", sep='|')
# смотрим, как сработал программный код
with open("Data/msft_piped.txt") as file:
    reader = csv.reader(file, delimiter=',')
    for i,row in enumerate(reader):
        print(row)
        if(i >= 5):
            break
```

```
['|Date|Open|High|Low|Close|Volume']
['0|7/21/2014|83.46|83.53|81.81|81.93|2359300']
['1|7/18/2014|83.3|83.4|82.52|83.35|4020800']
['2|7/17/2014|84.35|84.63|83.33|83.63|1974000']
['3|7/16/2014|83.77|84.91|83.66|84.91|1755600']
['4|7/15/2014|84.3|84.38|83.2|83.58|1874700']
```

Обработка загрязненных данных, в которых используются разделители полей

Данные с разделителями полей могут содержать посторонние строки в начале или в конце файла. В качестве примеров можно привести служебную информацию, размещаемую вверху, например, номер счета, адреса, сводные данные, размещаемые внизу. Кроме того, бывают случаи, когда данные хранятся в нескольких строках. Эти ситуации могут вызвать ошибки при загрузке данных. Чтобы разрешить эти ситуации, методы `pd.read_csv()` и `pd.read_table()` предлагают некоторые полезные параметры, которые выручат нас.

Для иллюстрации возьмем модифицированные данные о котировках акций, в них присутствуют лишние строки, которые можно отнести к «шуму»:

In[13]:
```
# смотрим первые 6 наблюдений файла msft2.csv
with open("Data/msft2.csv") as file:
    reader = csv.reader(file, delimiter=',')
    for i,row in enumerate(reader):
        print(row)
        if(i >= 6):
            break
```

```
['Данные начинаются не с первой строки', '', '', '', '', '']
['Date', 'Open', 'High', 'Low', 'Close', 'Volume']
['', '', '', '', '', '']
```

```
['А тут пробел между строкой заголовков и данными', '', '', '', '', '']
['7/21/2014', '83.46', '83.53', '81.81', '81.93', '2359300']
['7/18/2014', '83.3', '83.4', '82.52', '83.35', '4020800']
['7/17/2014', '84.35', '84.63', '83.33', '83.63', '1974000']
```

Эту ситуацию можно разрешить с помощью параметра `skiprows`, который проинформирует библиотеку pandas о пропуске строк `0`, `2` и `3`:

In[14]:
```
# считываем данные, пропустив строки 0, 2 и 3
df = pd.read_csv("Data/msft2.csv", skiprows=[0, 2, 3])
df[:5]
```

Out[14]:

	Date	Open	High	Low	Close	Volume
0	7/21/2014	83.46	83.53	81.81	81.93	2359300
1	7/18/2014	83.30	83.40	82.52	83.35	4020800
2	7/17/2014	84.35	84.63	83.33	83.63	1974000
3	7/16/2014	83.77	84.91	83.66	84.91	1755600
4	7/15/2014	84.30	84.38	83.20	83.58	1874700

Другой распространенной ситуацией является наличие лишней информации в конце файла, которую следует проигнорировать, чтобы правильно считать данные. В качестве примера возьмем следующие данные:

In[15]:
```
# смотрим файл msft_with_footer.csv
with open("Data/msft_with_footer.csv") as file:
    reader = csv.reader(file, delimiter=',')
    for row in reader:
        print(row)
```

```
['Date', 'Open', 'High', 'Low', 'Close', 'Volume']
['7/21/2014', '83.46', '83.53', '81.81', '81.93', '2359300']
['7/18/2014', '83.3', '83.4', '82.52', '83.35', '4020800']
[]
['Тут что-то есть в конце файла.']
```

При чтении такого файла будет выдано исключение, однако можно воспользоваться параметром `skip_footer`. Он задает количество строк в конце файла, игнорируемых при чтении файла:

In[16]:
```
# считываем, пропустив две строки в конце файла
df = pd.read_csv("Data/msft_with_footer.csv",
                 skipfooter=2,
                 engine = 'python')
df
```

Out[16]:

	Date	Open	High	Low	Close	Volume
0	7/21/2014	83.46	83.53	81.81	81.93	2359300
1	7/18/2014	83.30	83.40	82.52	83.35	4020800

Обратите внимание, что нам нужно задать параметр `engine = 'python'`. По крайней мере, в Anaconda без этой опции будет выдано предупреждение, поскольку работа с параметром `skip_footer` не поддерживается, если выбрано значение `engine='c'`. Это заставляет нас использовать реализацию на Python.

Предположим, у вас есть большой файл и нужно прочитать первые несколько строк, поскольку вам нужны только данные в начале файла и вы не хотите читать весь этот файл в памяти. Данный файл можно обработать с помощью параметра `nrows`:

```
In[17]:
# считаем только первые три строки
pd.read_csv("Data/msft.csv", nrows=3)
```

```
Out[17]:
        Date   Open   High    Low  Close   Volume
0  7/21/2014  83.46  83.53  81.81  81.93  2359300
1  7/18/2014  83.30  83.40  82.52  83.35  4020800
2  7/17/2014  84.35  84.63  83.33  83.63  1974000
```

Кроме того, вы можете либо пропустить определенное количество строк в начале файла и прочитать файл до конца, либо прочитать лишь несколько строк, достигнув определенной точки в файле. Для этого используйте параметр `skiprows`. Следующий программный код пропускает 100 строк, а затем считывает следующие 5 строк:

```
In[18]:
# пропускаем 100 строк, а затем считываем следующие 5 строк
pd.read_csv("Data/msft.csv", skiprows=100, nrows=5,
            header=0,
            names=['date', 'open', 'high', 'low',
                   'close', 'vol'])
```

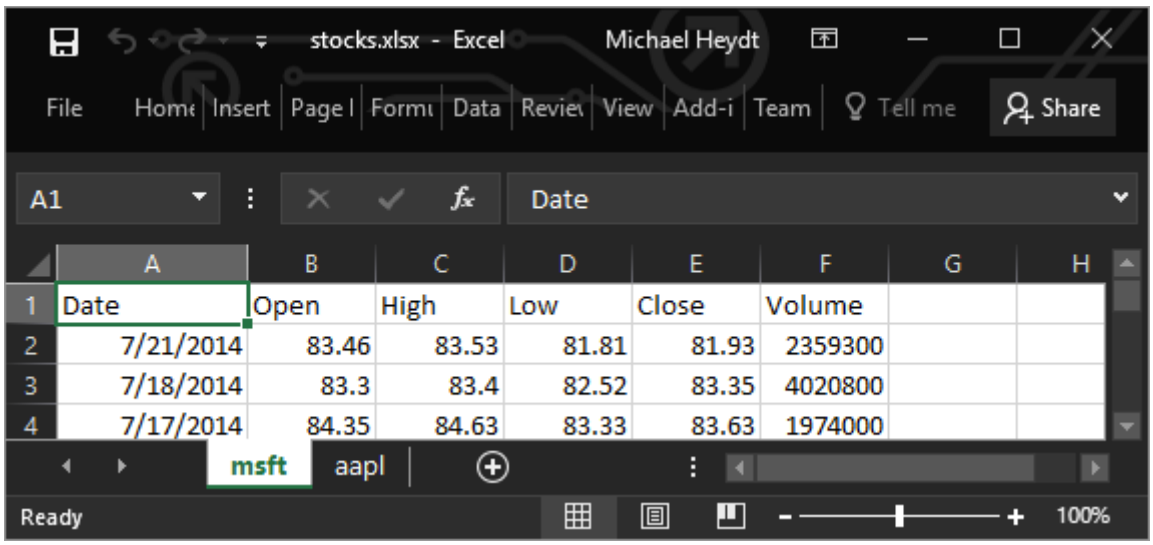
```
Out[18]:
        date   open   high    low  close      vol
0   3/3/2014  80.35  81.31  79.91  79.97  5004100
1  2/28/2014  82.40  83.42  82.17  83.42  2853200
2  2/27/2014  84.06  84.63  81.63  82.00  3676800
3  2/26/2014  82.92  84.03  82.43  83.81  2623600
4  2/25/2014  83.80  83.80  81.72  83.08  3579100
```

Предыдущий пример тоже пропускал чтение строки заголовков, поэтому необходимо сообщить библиотеке pandas, чтобы она не искала заголовки и использовала указанные имена.

Чтение и запись данных в формате Excel

Библиотека pandas поддерживает чтение данных в формате Excel 2003 и более поздних форматах с помощью функции `pd.read_excel()` или класса `ExcelFile`. Оба способа используют либо пакет `XLRD`, либо пакет `OpenPyXL`, поэтому вам необходимо убедиться в том, что один из них установлен в вашей среде Python.

Для иллюстрации возьмем файл `stocks.xlsx`. Если вы откроете его в Excel, вы увидите примерно следующее:



Книга Excel содержит два рабочих листа: `msft` и `aapl`, которые содержат данные о котировках акций.
Следующий программный код считывает файл `stocks.xlsx` в объект `DataFrame`:

In[19]:
```
# считываем файл Excel
# считываем только данные первого рабочего листа
# (msft в данном случае)
df = pd.read_excel("Data/stocks.xlsx")
df[:5]
```

Out[19]:
```
        Date   Open   High    Low  Close   Volume
0 2014-07-21  83.46  83.53  81.81  81.93  2359300
1 2014-07-18  83.30  83.40  82.52  83.35  4020800
2 2014-07-17  84.35  84.63  83.33  83.63  1974000
3 2014-07-16  83.77  84.91  83.66  84.91  1755600
4 2014-07-15  84.30  84.38  83.20  83.58  1874700
```

Этот программный код прочитал содержимое первого листа (рабочего листа `msft`) в файле Excel и использовал первую строку в качестве имен столбцов. Чтобы прочитать другой рабочий лист, можно передать имя рабочего листа, используя параметр `sheetname`:

In[20]:
```
# считываем данные рабочего листа aapl
aapl = pd.read_excel("Data/stocks.xlsx", sheet_name='aapl')
aapl[:5]
```

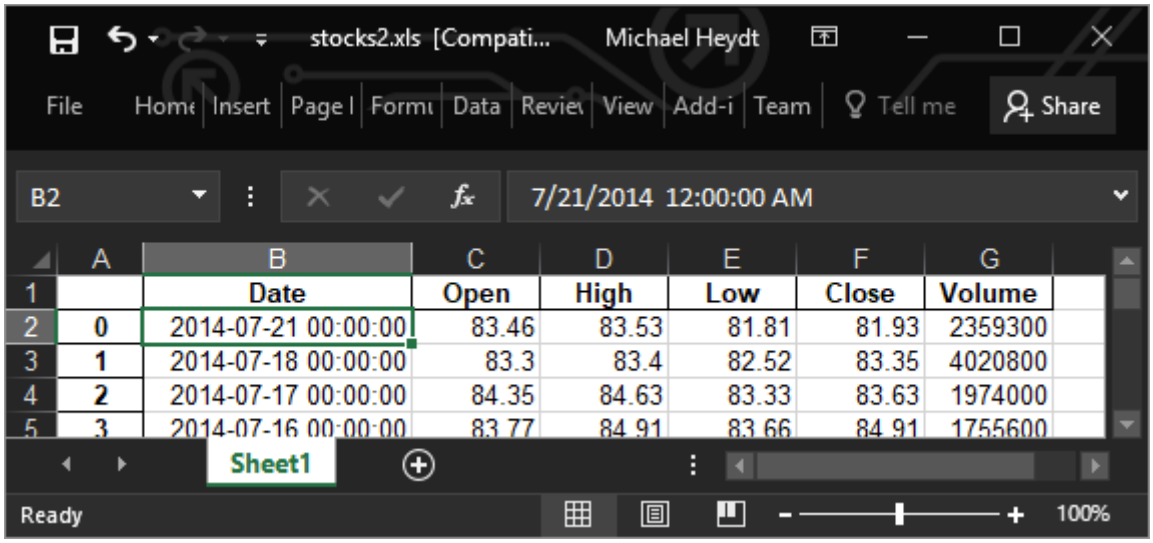
Out[20]:
```
        Date   Open   High    Low  Close    Volume
0 2014-07-21  94.99  95.00  93.72  93.94  38887700
1 2014-07-18  93.62  94.74  93.02  94.43  49898600
2 2014-07-17  95.03  95.28  92.57  93.09  57152000
3 2014-07-16  96.97  97.10  94.74  94.78  53396300
4 2014-07-15  96.80  96.85  95.03  95.32  45477900
```

Как и функция `pd.read_csv()`, функция `pd.read_excel()` позволяет аналогичным образом указывать имена столбцов, типы данных и

индексы. Все параметры, которые мы рассмотрели для функции `pd.read_csv()`, также применимы к функции `pd.read_excel()`.
Файлы Excel можно записать с помощью метода `.to_excel()` объекта `DataFrame`. Для записи в формат XLS потребуется пакет `XLWT`, поэтому перед его использованием убедитесь, что он загружен в вашу среду Python.
Следующий программный код записывает только что полученные данные в файл `tables2.xls`. По умолчанию выполняется сохранение датафрейма в рабочем листе `Sheet1`:

In[21]:
```
# сохраняем XLS-файл в рабочем листе 'Sheet1'
df.to_excel("Data/stocks2.xls")
```

Открыв этот файл в Excel, мы получим примерно следующую картину:

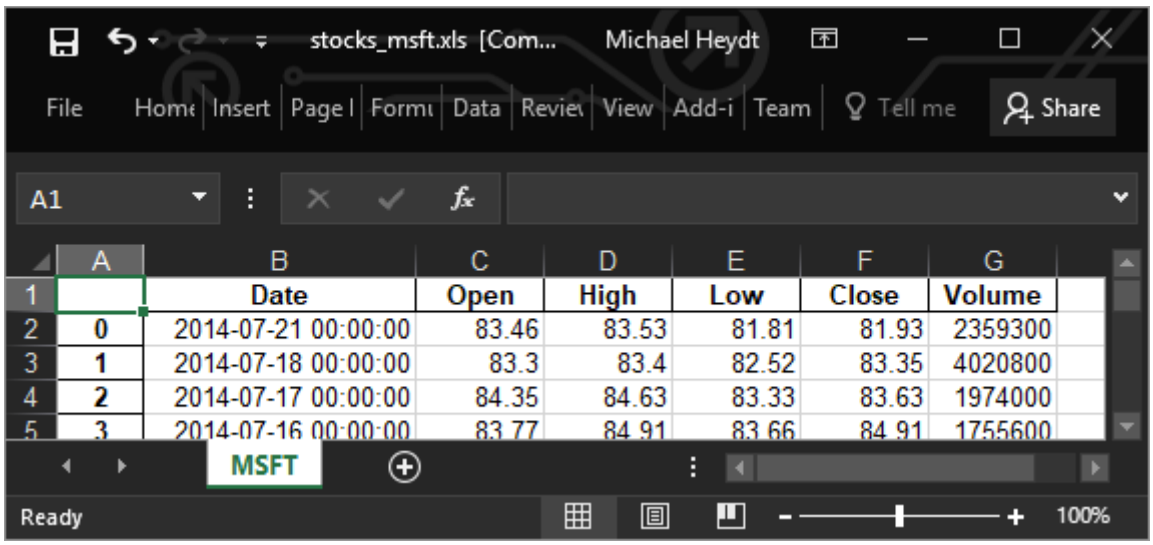


	A	B	C	D	E	F	G
1		Date	Open	High	Low	Close	Volume
2	0	2014-07-21 00:00:00	83.46	83.53	81.81	81.93	2359300
3	1	2014-07-18 00:00:00	83.3	83.4	82.52	83.35	4020800
4	2	2014-07-17 00:00:00	84.35	84.63	83.33	83.63	1974000
5	3	2014-07-16 00:00:00	83.77	84.91	83.66	84.91	1755600

С помощью параметра `sheet_name` можно задать имя рабочего листа:

In[22]:
```
# записываем, задав имя рабочего листа MSFT
df.to_excel("Data/stocks_msft.xls", sheet_name='MSFT')
```

В Excel мы видим, что рабочий лист получил имя `MSFT`:

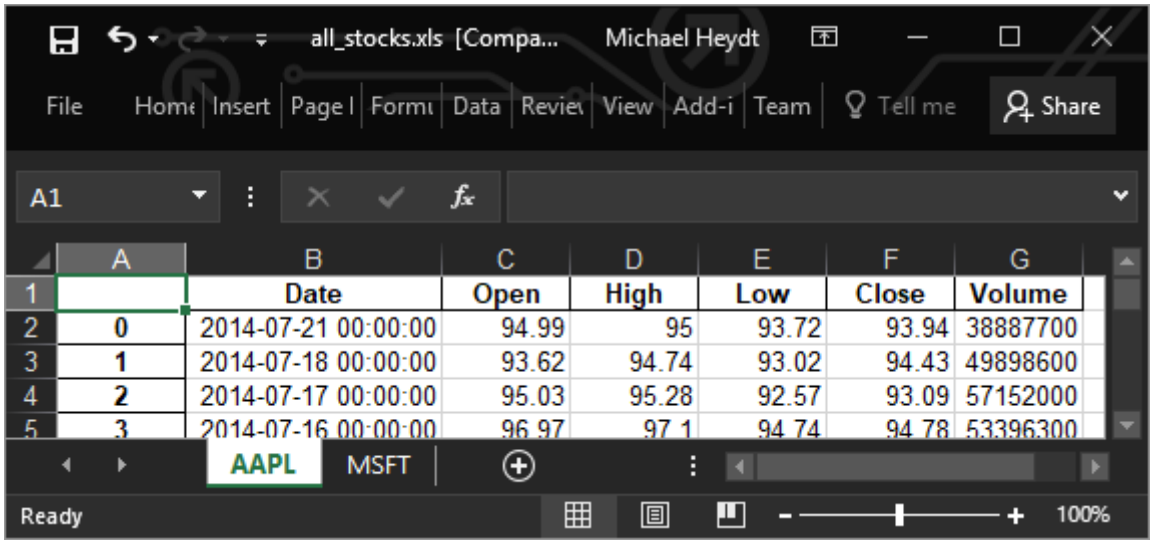


	A	Date	Open	High	Low	Close	Volume
2	0	2014-07-21 00:00:00	83.46	83.53	81.81	81.93	2359300
3	1	2014-07-18 00:00:00	83.3	83.4	82.52	83.35	4020800
4	2	2014-07-17 00:00:00	84.35	84.63	83.33	83.63	1974000
5	3	2014-07-16 00:00:00	83.77	84.91	83.66	84.91	1755600

Чтобы записать несколько датафреймов в один и тот же файл Excel, по одному объекту `DataFrame` на каждый рабочий лист, воспользуйтесь объектом `ExcelWriter` и ключевым словом `with`. `ExcelWriter` является частью библиотеки pandas, однако вам нужно убедиться в том, что он импортирован, поскольку данный объект отсутствует в пространстве имен верхнего уровня библиотеки pandas. Следующий программный код записывает два объекта `DataFrame` в два разных листа файла Excel:

In[23]:
```
# записываем несколько рабочих листов
# требуется класс ExcelWriter
from pandas import ExcelWriter
with ExcelWriter("Data/all_stocks.xls") as writer:
    aapl.to_excel(writer, sheet_name='AAPL')
    df.to_excel(writer, sheet_name='MSFT')
```

Мы видим, что книга Excel содержит два рабочих листа:



	A	Date	Open	High	Low	Close	Volume
2	0	2014-07-21 00:00:00	94.99	95	93.72	93.94	38887700
3	1	2014-07-18 00:00:00	93.62	94.74	93.02	94.43	49898600
4	2	2014-07-17 00:00:00	95.03	95.28	92.57	93.09	57152000
5	3	2014-07-16 00:00:00	96.97	97.1	94.74	94.78	53396300

Для записи в файлы XLSX используем ту же самую функцию `to_excel()`, однако в качестве расширения файла указываем `.XLSX`:

In[24]:
```
# записываем в xlsx
```

```
df.to_excel("Data/msft2.xlsx")
```

Чтение и запись JSON-файлов

Библиотека pandas может читать и записывать данные, хранящиеся в формате JavaScript Object Notation (JSON). Это один из моих любимых форматов в силу того, что его можно использовать для разных платформ и с различными языками программирования.

Чтобы продемонстрировать сохранение данных в формате JSON, мы сперва сохраним только что полученные данные Excel в файл JSON и рассмотрим его содержимое:

```
In[25]:
# записываем данные Excel в JSON-файл
df[:5].to_json("Data/stocks.json")
# теперь взглянем на JSON-файл
import json
from pprint import pprint

with open("Data/stocks.json") as data_file:
    data = json.load(data_file)

pprint(data)

Out[25]:
{'Close': {'0': 81.93, '1': 83.35, '2': 83.63, '3': 84.91, '4': 83.58},
 'Date': {'0': 1405900800000,
          '1': 1405641600000,
          '2': 1405555200000,
          '3': 1405468800000,
          '4': 1405382400000},
 'High': {'0': 83.53, '1': 83.4, '2': 84.63, '3': 84.91, '4': 84.38},
 'Low': {'0': 81.81, '1': 82.52, '2': 83.33, '3': 83.66, '4': 83.2},
 'Open': {'0': 83.46, '1': 83.3, '2': 84.35, '3': 83.77, '4': 84.3},
 'Volume': {'0': 2359300,
            '1': 4020800,
            '2': 1974000,
            '3': 1755600,
            '4': 1874700}}
```

Данные в формате JSON можно прочитать с помощью функции `pd.read_json()`:

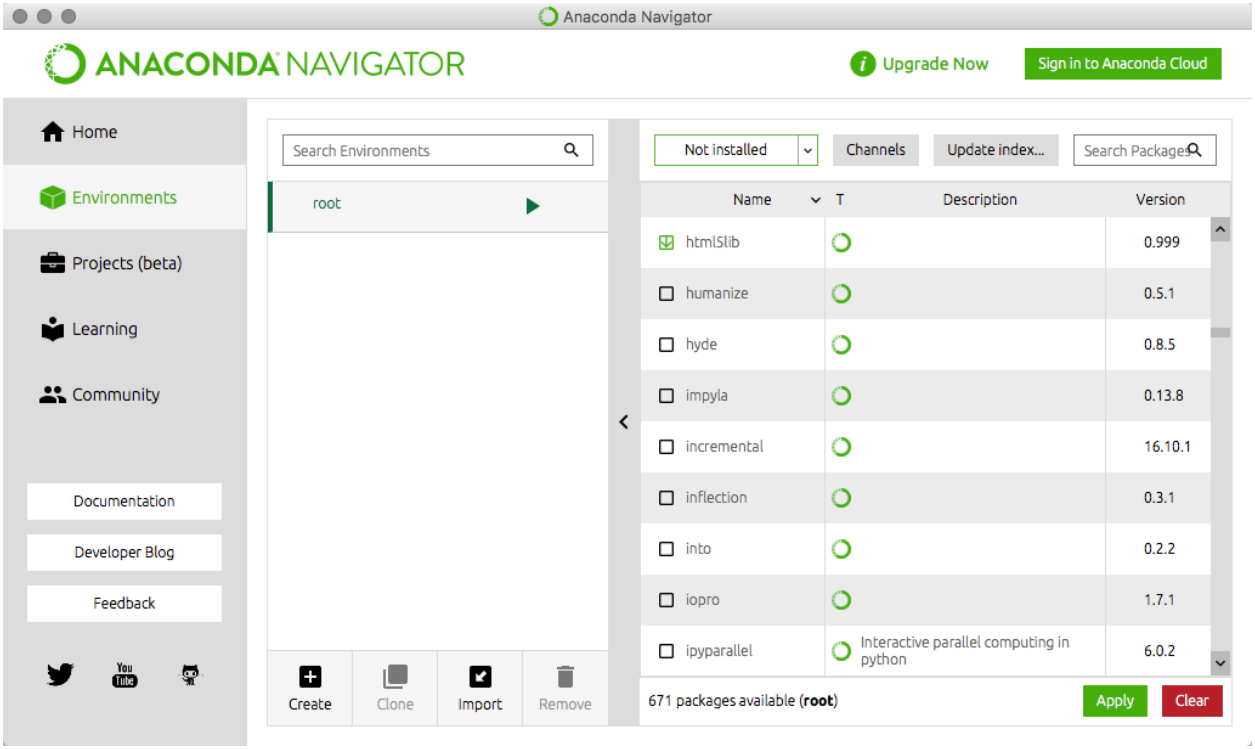
```
In[26]:
# считываем данные в формате JSON
df_from_json = pd.read_json("Data/stocks.json")
df_from_json[:5]

Out[26]:
   Close       Date   High    Low   Open   Volume
0  81.93 2014-07-21  83.53  81.81  83.46  2359300
1  83.35 2014-07-18  83.40  82.52  83.30  4020800
2  83.63 2014-07-17  84.63  83.33  84.35  1974000
3  84.91 2014-07-16  84.91  83.66  83.77  1755600
4  83.58 2014-07-15  84.38  83.20  84.30  1874700
```

Обратите внимание на две незначительные особенности, вызванные чтением/записью данных в формате JSON. Во-первых, столбцы были переупорядочены в алфавитном порядке. Во-вторых, индекс `DataFrame`, хотя и содержит контент, сортируется как строка. Эти проблемы можно

легко устранить, но в целях краткости мы не будем рассматривать их здесь.

Чтение HTML-файлов из Интернета

Библиотека pandas поддерживает чтение HTML-файлов (или HTML-файлов с URL-адресов). Внутри библиотека pandas использует пакеты `LXML`, `Html5Lib` и `BeautifulSoup4`. Эти пакеты предлагают впечатляющие возможности для чтения и записи HTML- таблиц.
Стандартный дистрибутив Anaconda может не включать эти пакеты. Если вы получаете ошибку, то исходя из ее содержания установите соответствующую библиотеку при помощи Anaconda Navigator:



Кроме того, вы можете использовать pip.
Функция `pd.read_html()` считывает HTML-файл (или HTML-файл с URL-адреса) и записывает все найденные HTML-таблицы в один или несколько объектов `DataFrame`. Функция всегда возвращает список объектов `DataFrame` (ноль или большее количество, в зависимости от количества таблиц, найденных в HTML-файле).
Для иллюстрации мы считаем данные, представляющие собой список банков-банкротов, опубликованный на сайте Федеральной корпорации по страхованию вкладов по адресу https://www.fdic.gov/bank/individual/failed/banklist.html. Просмотрев страницу, можно увидеть, что список обанкротившихся банков довольно внушителен.
На самом деле эти данные очень просто прочитать с помощью библиотеки pandas и ее функции `pd.read_html()`:

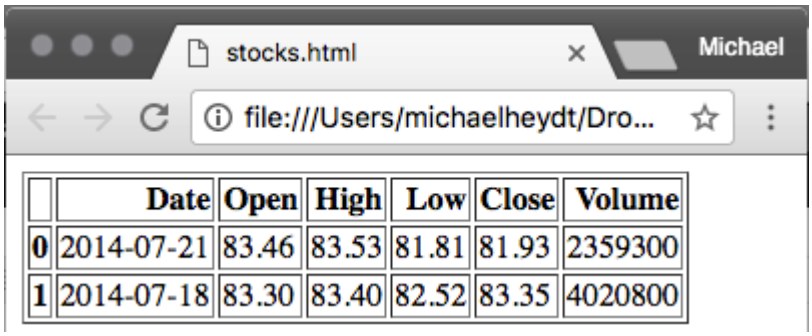
In[27]:
```
# задаем URL-адрес HTML-файла
url = "http://www.fdic.gov/bank/individual/failed/banklist.html"
# читаем его
banks = pd.read_html(url)
```

In[28]:
```
# проверяем, как была прочитана
# часть первой таблицы
banks[0][0:5].iloc[:,0:2]
```

Out[28]:
```
                                              Bank Name                City
0                                   Fayette County Bank           Saint Elmo
1  Guaranty Bank, (d/b/a BestBank in Georgia & Mi...            Milwaukee
2                                     First NBC Bank         New Orleans
3                                      Proficio Bank  Cottonwood Heights
4                       Seaway Bank and Trust Company             Chicago
```

И опять же это было очень просто!
С помощью метода `.to_html()` объект DataFrame можно записать в HTML-файл. Этот метод создает файл, содержащий тег `<table>` для данных (а не весь HTML-документ). Следующий программный код записывает ранее прочитанные нами данные о котировках акций в HTML-файл и выводит полученный результат в браузере:

In[29]:
```
# считываем данные о котировках акций
df = pd.read_excel("Data/stocks.xlsx")
# записываем первые две строки в HTML
df.head(2).to_html("Data/stocks.html")
# смотрим HTML-файл в браузере
import webbrowser
webbrowser.open("Data/stocks.html")
```



	Date	Open	High	Low	Close	Volume
0	2014-07-21	83.46	83.53	81.81	81.93	2359300
1	2014-07-18	83.30	83.40	82.52	83.35	4020800

Это удобно, поскольку вы можете использовать библиотеку pandas для записи HTML-фрагментов, встраиваемых в веб-сайт, обновлять их по мере необходимости и таким образом пользователям вашего сайта всегда будут доступны новые данные без необходимости выполнения сложных запросов и вызовов служб.

Чтение и запись HDF5-файлов

HDF5 – это модель данных, библиотека и файловый формат для хранения и управления данными. Он широко используется в научных вычислительных средах. HDF5 поддерживает неограниченное

количество типов данных и предназначен для гибкого и эффективного ввода-вывода, а также для больших и сложных данных.
HDF5 является портативным и расширяемым форматом, что позволяет приложениям развиваться при использовании HDF5. HDF5 Technology Suite включает инструменты и приложения для управления, обработки, просмотра и анализа данных в формате HDF5. HDF5 – это:

- универсальная модель данных, которая может представлять очень сложные объекты данных и широкий спектр метаданных
- полностью переносимый файловый формат, не ограничивающийся количеством или размером объектов данных в коллекции
- программная библиотека, которая работает на различных вычислительных платформах, начиная от портативных компьютеров и заканчивая массивными системами параллельных вычислений, и предлагает высокоуровневый API с интерфейсами C, C++, Fortran 90 и Java
- богатый набор интегрированных возможностей, позволяющих оптимизировать время доступа и пространство для хранения данных
- инструменты и приложения для управления, обработки, просмотра и анализа данных

`HDFStore` представляет собой иерархический, словареподобный объект, который считывает и записывает объекты библиотеки pandas в формат HDF5. Внутри HDFStore использует библиотеку `PyTables`, поэтому убедитесь, что она установлена, если хотите использовать данный формат.
Следующий программный код демонстрирует запись датафрейма в формат HDF5. Вывод показывает, что хранилище HDF5 имеет корневой объект под названием `df`, который является датафреймом и он состоит восьми строк и трех столбцов:

```
In[30]:
# задаем стартовое значение генератора случайных чисел
# для получения воспроизводимых результатов
np.random.seed(123456)
# создаем датафрейм, состоящий из дат и случайных чисел,
# записанных в трех столбцах
df = pd.DataFrame(np.random.randn(8, 3),
                  index=pd.date_range('1/1/2000', periods=8),
                  columns=['A', 'B', 'C'])

# создаем хранилище HDF5
store = pd.HDFStore("Data/store.h5")
store['df'] = df # сохранение произошло здесь
store

Out[30]:
<class 'pandas.io.pytables.HDFStore'>
File path: C:/data/store.h5
/df            frame        (shape->[8,3])
```

Следующий программный код считывает данные хранилища HDF5 и записывает в объект `DataFrame`:

```
In[31]:
# считываем данные хранилища HDF5
store = pd.HDFStore("Data/store.h5")
df = store['df']
df[:5]

Out[31]:
```

```
                   A         B         C
2000-01-01  0.469112 -0.282863 -1.509059
2000-01-02 -1.135632  1.212112 -0.173215
2000-01-03  0.119209 -1.044236 -0.861849
2000-01-04 -2.104569 -0.494929  1.071804
2000-01-05  0.721555 -0.706771 -1.039575
```

Объект `DataFrame` записывается в HDF5-файл в тот момент, когда он присваивается объекту-хранилищу. Изменения в объекте `DataFrame`, сделанные после этого момента, не будут сохранены, по крайней мере, до тех пор, пока этот объект не будет присвоен объекту-хранилищу еще раз. Следующий программный код демонстрирует это. Сначала он вносит изменения в объект `DataFrame`, затем присваивает его хранилищу HDF5, тем самым обновив хранилище данных:

In[32]:
```
# этот программный код меняет датафрейм, но изменения не сохраняются
df.iloc[0].A = 1
# чтобы сохранить изменения, присваиваем объект DataFrame
# объекту-хранилищу HDF5
store['df'] = df
# теперь изменения сохранены
# следующий программный код загружает хранилище и
# выводит первые две строки, демонстрируя,
# что сохранение выполнено
pd.HDFStore("Data/store.h5")['df'][:5] # сейчас датафрейм в хранилище
```

Out[32]:
```
                   A         B         C
2000-01-01  1.000000 -0.282863 -1.509059
2000-01-02 -1.135632  1.212112 -0.173215
2000-01-03  0.119209 -1.044236 -0.861849
2000-01-04 -2.104569 -0.494929  1.071804
2000-01-05  0.721555 -0.706771 -1.039575
```

Загрузка CSV-файлов из Интернета

Очень часто нужно прочитать данные из Интернета. Библиотека pandas упрощает чтение данных из Интернета. Все функции pandas, которые мы рассмотрели, позволяют задать URL-адрес, FTP-адрес или S3-адрес вместо локального пути к файлу, и все они работают так же, как работают с локальным файлом.

Следующий программный код демонстрирует, насколько просто выполнить прямой HTTP-запрос с помощью существующей функции `pd.read_csv()`.

```
In[33]:
# считываем csv непосредственно по URL-адресу
countries = pd.read_csv(
    "https://raw.githubusercontent.com/cs109/2014_data/master/countries.csv")
countries[:5]

Out[33]:
    Country  Region
0   Algeria  AFRICA
1    Angola  AFRICA
2     Benin  AFRICA
3  Botswana  AFRICA
4   Burkina  AFRICA
```

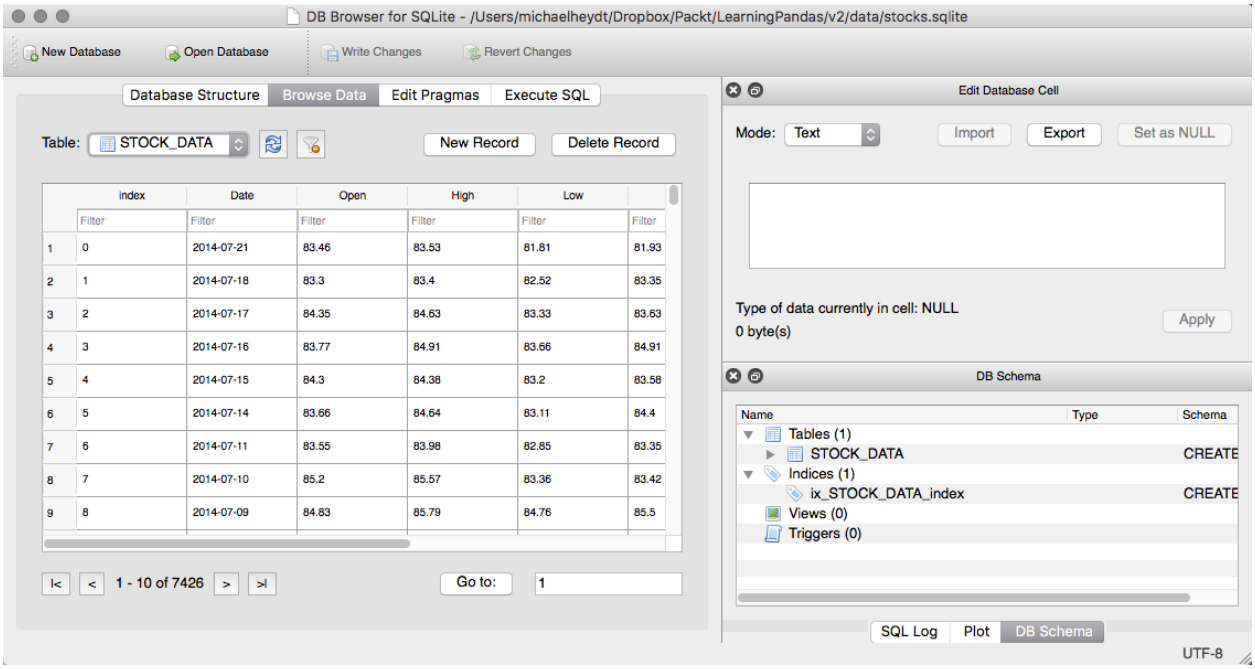
Чтение из базы данных SQL и запись в базу данных SQL

Библиотека pandas может считать данные из любой базы данных SQL, которая поддерживает адаптеры данных Python в рамках интерфейса Python DB-API. Чтение выполняется с помощью функции `pandas.io.sql.read_sql()`, а запись в базу данных SQL выполняется с помощью метода `.to_sql()` объекта `DataFrame`.

Для иллюстрации следующий программный код считывает данные о котировках акций из файлов `msft.csv` и `aapl.csv`. Затем он подключается к файлу базы данных SQLite3. Если файл не существует, он создается «на лету». Затем программный код записывает данные `MSFT` в таблицу под названием `STOCK_DATA`. Если таблица не существует, она также будет создана. Если она уже существует, все данные заменяются данными `MSFT`. Наконец, программный код добавляет в эту таблицу данные о котировках `AAPL`:

```
In[34]:
# импортируем библиотеку SQLite
import sqlite3

# считываем данные о котировках акций из CSV-файла
msft = pd.read_csv("Data/msft.csv")
msft["Symbol"]="MSFT"
aapl = pd.read_csv("Data/aapl.csv")
aapl["Symbol"]="AAPL"

# создаем подключение
connection = sqlite3.connect("Data/stocks.sqlite")
# .to_sql() создаст базу SQL для хранения датафрейма
# в указанной таблице.  if_exists задает
# действие, которое нужно выполнить в том случае,
# если таблица уже существует
msft.to_sql("STOCK_DATA", connection, if_exists="replace")
aapl.to_sql("STOCK_DATA", connection, if_exists="append")

# подтверждаем отправку данных в базу и закрываем подключение
connection.commit()
connection.close()
```

Чтобы убедиться в создании данных, можно открыть файл базы данных с помощью такого инструмента, как SQLite Data Browser (доступен по адресу https://github.com/sqlitebrowser/sqlitebrowser). Рисунок ниже показывает несколько записей в файле базы данных:



Данные из базы данных SQL можно прочитать с помощью функции `pd.io.sql.read_sql()`. Следующий программный код демонстрирует выполнение запроса к файлу `stocks.sqlite` с помощью SQL и сообщает об этом пользователю:

In[35]:
```
# подключаемся к файлу базы данных
connection = sqlite3.connect("Data/stocks.sqlite")

# запрос всех записей в STOCK_DATA
# возвращает датафрейм
# index_col задает столбец, который нужно сделать
# индексом датафрейма
stocks = pd.io.sql.read_sql("SELECT * FROM STOCK_DATA;",
                            connection, index_col='index')

# закрываем подключение
connection.close()

# выводим первые 5 наблюдений в извлеченных данных
stocks[:5]
```

Out[35]:

index	Date	Open	High	Low	Close	Volume	Symbol
0	7/21/2014	83.46	83.53	81.81	81.93	2359300	MSFT
1	7/18/2014	83.30	83.40	82.52	83.35	4020800	MSFT
2	7/17/2014	84.35	84.63	83.33	83.63	1974000	MSFT
3	7/16/2014	83.77	84.91	83.66	84.91	1755600	MSFT
4	7/15/2014	84.30	84.38	83.20	83.58	1874700	MSFT

Кроме того, для отбора столбцов еще можно использовать условие `WHERE` в SQL. Чтобы продемонстрировать это, следующий программный код отбирает записи, в которых количество проторгованных акций MSFT превышает `29200100`:

In[36]:
```
# открываем подключение
connection = sqlite3.connect("Data/stocks.sqlite")
# создаем строку-запрос
query = "SELECT * FROM STOCK_DATA WHERE " + \
```

```
        "Volume>29200100 AND Symbol='MSFT';"
# выполняем и закрываем подключение
items = pd.io.sql.read_sql(query, connection, index_col='index')
connection.close()
# выводим результат запроса
items
```

Out[36]:

index	Date	Open	High	Low	Close	Volume	Symbol
1081	5/21/2010	42.22	42.35	40.99	42.00	33610800	MSFT
1097	4/29/2010	46.80	46.95	44.65	45.92	47076200	MSFT
1826	6/15/2007	89.80	92.10	89.55	92.04	30656400	MSFT
3455	3/16/2001	47.00	47.80	46.10	45.33	40806400	MSFT
3712	3/17/2000	49.50	50.00	48.29	50.00	50860500	MSFT

Итоговым моментом является то, что большая часть программного кода в этих примерах была программным кодом SQLite3. Библиотека pandas в этих примерах используется лишь тогда, когда нужно применить методы `.to_sql()` и `.read_sql()`. Они принимают объект подключения, который может быть любым адаптером данных, совместимым с интерфейсом Python DB-API, поэтому вы можете работать с любой информацией базы данных, просто создав соответствующий объект подключения. Программный код на уровне pandas остается неизменным для любой поддерживаемой базы данных.

Загрузка данных с удаленных сервисов

Библиотека pandas вплоть до версии 0.19.0 напрямую поддерживала работу с различными источниками веб-данных в рамках пространства имен `pandas.io.data`. Теперь ситуация изменилось, поскольку был осуществлен рефракторинг библиотеки pandas и функционал по работе с различными веб-источниками данных был перенесен в пакет `pandas-datareader`.

Этот пакет обеспечивает доступ ко многим полезным источникам данных, включая:

- Ежедневные исторические цены на акции от Yahoo! или Google Finance
- Yahoo! и Google Options
- Enigma, поставщик структурированных данных
- Библиотека данных экономических данных Федерального резерва
- База данных Кеннета Френча
- Всемирный банк
- OECD
- Eurostat
- EDGAR Index
- Данные TSP Fund
- Данные об исторических валютных курсах Oanda
- Тикеры TraderNasdaq

Обратите внимание, что поскольку эти данные поступают из внешнего источника данных и фактические значения могут меняться со временем, возможно, при запуске программного кода вы получите значения, отличающиеся от значений, приведенных в книге.

Загрузка Базы данных по экономической статистике Федерального резервного банка Сент-Луиса

База данных по экономической статистике Федерального резервного банка Сент-Луиса[2] (Federal Reserve Economic Data или FRED), доступная по адресу http://research.stlouisfed.org/fred2/, представляет собой 240000 экономических показателей американских и международных компаний, расположенных в хронологическом порядке (240000 временных рядов). Она собирается из более чем 76 источников данных и постоянно увеличивается в объеме.

Для загрузки данных нам потребуется пакет `pandas-datareader`. Чтобы воспользоваться пакетом `pandas-datareader`, мы должны импортировать его (перед этим убедитесь, что он установлен[3]).

In[37]:
```
# импортируем пакет pandas_datareader
pd.core.common.is_list_like = pd.api.types.is_list_like
import pandas_datareader as pdr
```

Данные FRED можно получить с помощью класса `FredReader`, передав определенную метку временного ряда в качестве параметра `name`. Например, следующий программный код извлекает информацию о GDP[4] между двумя указанными датами:

In[38]:
```
# считываем данные по GDP из FRED
gdp = pdr.data.FredReader("GDP",
                          date(2012, 1, 1),
                          date(2014, 1, 27))
gdp.read()[:5]
```

[2] Федеральный резервный банк Сент-Луиса – это один из двенадцати банков, которые вместе образуют Федеральную резервную систему Соединенных штатов (Federal Reserve System) США. Они находятся в Бостоне, Нью-Йорке, Филадельфии, Кливленде, Ричмонде, Атланте, Чикаго, Сент-Луисе, Миннеаполисе, Канзас-Сити, Далласе и Сан-Франциско. Они выполняют функции центробанка (central bank services) и вместе с Советом управляющих Федерального резерва участвуют в разработке и осуществлении кредитно-денежной политики, а также в регулировании деятельности местных коммерческих и сберегательных банков. – *Прим. пер.*

[3] Если вы используете Anaconda for Python, достаточно запустить в Anaconda Promt строку `conda install -c anaconda pandas-datareader`.

[4] GDP или Gross Domestic Product – ВВП или валовый внутренний продукт. – *Прим. пер.*

```
Out[38]:
                GDP
DATE
2012-01-01  15973.9
2012-04-01  16121.9
2012-07-01  16227.9
2012-10-01  16297.3
2013-01-01  16475.4
```

Чтобы выбрать другой ряд, просто укажите идентификатор ряда в первом параметре. По сайту удобно перемещаться, переходя от одного ряда к другому и просматривая данные, визуализации которых даны непосредственно на сайте. Например, на следующем рисунке показан временной ряд Compensation of employees: Wages and salaries[5]:

Данные этого временного ряда представлены идентификатором `A576RC1A027NBEA`, и мы можем загрузить его с помощью следующего программного кода:

[5] Compensation of employees: Wages and salaries – Данные о заработной плате и вознаграждении работников наемного труда. Под salary подразумевается оклад, жалование, заработная плата, которая ежемесячно переводится на счет в банке согласно контракту. Этот тип заработной платы характерен для офисных работников. Он не зависит от объема выполненной работы и количества отработанных часов. Wages – заработная плата, которая выплачивается наемным работникам еженедельно либо ежедневно, часто наличными. Такой тип зарплаты получают рабочие, которые занимаются физическим трудом, строители. Он зависит от объема выполненной работы и количества отработанного времени. – *Прим. пер.*

In[39]:
```
# получаем данные по показателю Compensation of employees:
# Wages and salaries
pdr.data.FredReader("A576RC1A027NBEA",
                    date(1929, 1, 1),
                    date(2013, 1, 1)).read()[:5]
```

Out[39]:
```
            A576RC1A027NBEA
DATE
1929-01-01             50.5
1930-01-01             46.2
1931-01-01             39.2
1932-01-01             30.5
1933-01-01             29.0
```

Загрузка данных Кеннета Френча

Кеннет Френч является профессором финансов в бизнес-школе имени Амоса Така при Дартмутском колледже. Он создал обширную библиотеку экономических данных, которая доступна для скачивания через Интернет. Веб-сайт с его данными находится по адресу http://mba.tuck.dartmouth.edu/pages/faculty/ken.french/data_library.html и содержит подробное описание наборов данных.

Данные, выложенные на сайте, можно загрузить в виде ZIP-файлов и прочитать непосредственно в датафрейм, указав имя файла (предварительно распаковав) и воспользовавшись функцией `FameFrenchReader`. В качестве примера следующий программный код считывает данные Global Factors:

In[40]:
```
# считываем набор данных Global Factors из библиотеки Кеннета Френча
factors = pdr.data.FamaFrenchReader("Global_Factors").read()
factors[0][:5]
```

Out[40]:
```
         Mkt-RF   SMB   HML   WML    RF
Date
2010-01   -3.70  2.70 -0.29 -2.23  0.00
2010-02    1.24  0.14  0.10  1.59  0.00
2010-03    6.30 -0.26  3.18  4.26  0.01
2010-04    0.44  3.78  0.77  1.60  0.01
2010-05   -9.52  0.17 -2.54 -0.56  0.01
```

Загрузка данных Всемирного банка

Тысячи источников информации доступны в рамках базы данных Всемирного банка и их можно непосредственно прочитать в объект `DataFrame` библиотеки pandas. С каталогом данных Всемирного банка можно ознакомиться по адресу http://www.worldbank.org/.

Наборы данных Всемирного банка идентифицируются с помощью индикаторов, текстового кода, представляющего каждый набор данных. Полный список индикаторов можно получить с помощью функции `pandas.io.wb.get_indicators()`. На момент написания книги было 16167 показателей. Ниже приведены первые пять индикаторов:

In[41]:
```
from pandas_datareader import wb
```

```
# извлекаем все индикаторы
all_indicators = pdr.wb.get_indicators()
# выводим первые 5 индикаторов
all_indicators.iloc[:5,:2]
```

Out[41]:

```
                    id                                      name
0    1.0.HCount.1.90usd          Poverty Headcount ($1.90 a day)
1     1.0.HCount.2.5usd          Poverty Headcount ($2.50 a day)
2  1.0.HCount.Mid10to50    Middle Class ($10-50 a day) Headcount
3       1.0.HCount.Ofcl  Official Moderate Poverty Rate-National
4   1.0.HCount.Poor4uds             Poverty Headcount ($4 a day)
```

Эти индикаторы можно посмотреть на веб-сайте Всемирного банка, но если вы знаете, какой индикатор вам нужен, можно просто выполнить поиск. В качестве примера следующий программный код использует функцию `wb.search()` для поиска индикаторов данных, связанных со средней продолжительностью жизни:

In[42]:

```
# поиск индикаторов, связанных с продолжительностью жизни
le_indicators = pdr.wb.search("life expectancy")
# выводим первые три строки и первые два столбца
le_indicators.iloc[:5,:2]
```

Out[42]:

```
                     id                                               name
8773         SE.SCH.LIFE  School life expectancy, primary to tertiary, b...
10170  SP.DYN.LE00.FE.IN           Life expectancy at birth, female (years)
10171     SP.DYN.LE00.IN            Life expectancy at birth, total (years)
10172  SP.DYN.LE00.MA.IN             Life expectancy at birth, male (years)
10173  SP.DYN.LE60.FE.IN                  Life expectancy at age 60, female
```

Каждый индикатор разбит по странам. Полный список по странам можно получить с помощью функции `wb.get_countries()`:

In[43]:

```
# получаем список стран, показываем код и название
countries = pdr.wb.get_countries()
# выводим фрагмент списка стран
countries.loc[0:5,['name', 'capitalCity', 'iso2c']]
```

Out[43]:

```
          name       capitalCity iso2c
0        Aruba        Oranjestad    AW
1  Afghanistan             Kabul    AF
2       Africa                       A9
3       Angola            Luanda    AO
4      Albania            Tirane    AL
5      Andorra  Andorra la Vella    AD
```

Данные по каждому индикатору можно загрузить с помощью функции `wb.download()`, указав с помощью параметра `indicator` набор данных. Ниже приведены данные о продолжительности жизни для стран с 1980 по 2014 годы:

In[44]:

```
# получаем данные о продолжительности жизни
# для всех стран с 1980 по 2014 годы
le_data_all = pdr.wb.download(indicator="SP.DYN.LE00.IN",
                              start='1980',
                              end='2014')
le_data_all
```

Out[44]:

```
                   SP.DYN.LE00.IN
country       year
Canada        2014       81.953049
              2013       81.772049
              2012       81.583512
              2011       81.448780
              2010       81.197561
...                           ...
United States 1984       74.563415
              1983       74.463415
              1982       74.360976
              1981       74.009756
              1980       73.609756

[105 rows x 1 columns]
```

По умолчанию будут возвращены данные только для США, Канады и Мексики. Это можно увидеть, просмотрев индекс результатов предыдущего запроса:

In[45]:

```
# по умолчанию будут возвращены данные только для США, Канады и Мексики
le_data_all.index.levels[0]
```

Out[45]:

```
Index(['Canada', 'Mexico', 'United States'], dtype='object', name='country')
```

Чтобы получить данные для большего количества стран, укажите их явно, воспользовавшись параметром `country`. Следующий программный код извлекает данные для всех известных стран:

In[46]:

```
# отключаем предупреждения Anaconda
import warnings
warnings.simplefilter('ignore')
# получаем данные о продолжительности жизни
# для всех стран с 1980 по 2014 годы
le_data_all = wb.download(indicator="SP.DYN.LE00.IN",
                          country = countries['iso2c'],
                          start='1980',
                          end='2012')
le_data_all
```

Out[46]:

```
                SP.DYN.LE00.IN
country  year
Aruba    2012          75.299
         2011          75.158
         2010          75.016
         2009          74.872
         2008          74.725
...                       ...
Zimbabwe 1984          60.965
         1983          60.746
         1982          60.386
         1981          59.921
         1980          59.390

[8712 rows x 1 columns]
```

Мы можем выполнить разные интересные манипуляции с этими данными. В примере, рассмотренном ниже, мы для каждого года определяем страну с самой низкой продолжительностью жизни. Чтобы выполнить эту операцию, сначала нужно перевернуть (транспонировать) эти данные, чтобы индексом стало название страны, а годом стал столбец. Более подробно транспонирование данных мы рассмотрим в последующих главах, а пока вам важно просто знать, что следующий программный код преобразует страны в индекс, а года – в столбцы. Кроме того, каждое значение представляет собой продолжительность жизни для каждой страны за конкретный год:

In[47]:

```
#le_data_all.pivot(index='country', columns='year')
le_data = le_data_all.reset_index().pivot(index='country',
                                          columns='year')
# смотрим транспонированные данные
le_data.iloc[:5,0:3]
```

Out[47]:

```
                SP.DYN.LE00.IN
year                      1980    1981    1982
country
Afghanistan             41.853  42.513  43.217
Albania                 70.207  70.416  70.635
Algeria                 58.196  59.521  60.823
American Samoa             NaN     NaN     NaN
Andorra                    NaN     NaN     NaN
```

Получив данные в таком формате, мы можем для каждого года определить страну с наименьшей продолжительностью жизни, воспользовавшись параметром `.idxmin(axis=0)`:

In[48]:

```
# определяем для каждого года страну с
# наименьшей продолжительностью жизни
country_with_least_expectancy = le_data.idxmin(axis=0)
country_with_least_expectancy[:5]
```

Out[48]:

```
                year
SP.DYN.LE00.IN  1980       Cambodia
                1981       Cambodia
                1982    Timor-Leste
                1983    South Sudan
                1984    South Sudan
dtype: object
```

Минимальное значение продолжительности жизни для каждого года можно получить с помощью `.min(axis=0)`:

In[49]:
```
# определяем для каждого года минимальное значение
# продолжительности жизни
expectancy_for_least_country = le_data.min(axis=0)
expectancy_for_least_country[:5]
```

Out[49]:
```
                 year
SP.DYN.LE00.IN   1980    27.536
                 1981    33.342
                 1982    38.174
                 1983    39.671
                 1984    40.005
dtype: float64
```

Затем эти два результата можно объединить в новый датафрейм, в котором по каждому году будет приведена информация о стране с наименьшей продолжительностью жизни и минимальном значении продолжительности жизни:

In[50]:
```
# этот программный код объединяет два датафрейма вместе и мы получаем
# по каждому году страну с наименьшей продолжительностью жизни и
# наименьшее значение продолжительности жизни
least = pd.DataFrame(
    data = {'Country': country_with_least_expectancy.values,
            'Expectancy': expectancy_for_least_country.values},
    index = country_with_least_expectancy.index.levels[1])
least[:5]
```

Out[50]:
```
          Country  Expectancy
year
1980     Cambodia      27.536
1981     Cambodia      33.342
1982  Timor-Leste      38.174
1983  South Sudan      39.671
1984  South Sudan      40.005
```

Выводы

В этой главе мы рассмотрели, как библиотека pandas упрощает доступ к данным, расположенным в разных местах и представленным в различных форматах, обеспечивая автоматическое преобразование данных различного формата в объекты DataFrame. Мы начали с чтения и записи локальных файлов в форматах CSV, HTML, JSON, HDF5 и Excel, читая в объекты DataFrame и записывая непосредственно из объектов DataFrame, не беспокоясь о деталях отображения данных различных форматов.

Затем мы разобрали способы получения данных из удаленных источников. Во-первых, мы увидели, что функции и методы, которые работают с локальными файлами, также могут читать данные, размещенные на веб-ресурсах и в облачных источниках данных. Затем мы рассмотрели функционал библиотеки pandas, позволяющий

загружать разнообразные данные из Интернета и веб-сервисов типа Yahoo! Finance и Всемирного банка.

Теперь, когда мы можем загружать данные, следующим шагом, приближающим их использование, является приведение данных в порядок, потому что часто бывает, что полученная информация имеет такие проблемы, как отсутствующие или грязные данные. Следующая глава будет посвящена решению этим проблем в рамках процесса, который обычно называется **приведением данных в порядок**.

ГЛАВА 10 ПРИВЕДЕНИЕ ДАННЫХ В ПОРЯДОК

Сейчас мы находимся в том этапе обработки данных, где нам нужно взглянуть на полученные данные и устранить любые аномалии, которые могут возникнуть в ходе анализа. Эти аномалии могут быть обусловлены множеством причин. Иногда некоторые наблюдения не удается записать или, возможно, они пропущены. Часто некоторые наблюдения дублируются.

Этот процесс обработки аномалий часто называют **приведением данных в порядок (data tidying)**, и вы увидите, что этот термин часто используется в анализе данных. Это очень важный этап процесса работы с данными и он может занять большую часть вашего времени, прежде чем вы начнете выполнять анализ полученной информации.

Приведение данных в порядок может быть утомительным процессом, особенно, когда используются инструменты программирования, не предназначенные для решения конкретных задач чистки данных. К счастью для нас, библиотека pandas предлагает массу инструментов, которые можно использовать для решения этих задач, а также она помогает нам решать их весьма эффективно.

В этой главе мы рассмотрим различные задачи, связанные с приведением данных в порядок. В частности, вы узнаете:

- Что представляют собой аккуратные данные
- Как работать с пропущенными данными
- Как найти значения `NaN` в данных
- Как отфильтровывать (удалять) пропущенные данные
- Как pandas обрабатывает пропущенные значения в ходе вычислений
- Как найти, отфильтровать и исправить неизвестные значения
- Выполнение интерполяции пропущенных значений
- Как найти и удалить дублирующиеся наблюдения
- Как преобразовать значения с помощью методов `.replace()`, `.map()` и `.apply()`

Настройка библиотеки pandas

Мы начнем рассмотрение примеров в этой главе, задав следующие инструкции для импорта библиотек и настройки вывода:

```
In[1]:
# импортируем библиотеки numpy и pandas
import numpy as np
import pandas as pd

# импортируем библиотеку datetime для работы с датами
import datetime
from datetime import datetime, date

# Задаем некоторые опции библиотеки pandas, которые
# настраивают вывод
pd.set_option('display.notebook_repr_html', False)
pd.set_option('display.max_columns', 8)
pd.set_option('display.max_rows', 10)
pd.set_option('display.width', 80)

# импортируем библиотеку matplotlib для построения графиков
import matplotlib.pyplot as plt
%matplotlib inline
```

Что такое приведение данных в порядок?

Аккуратные данные (tidy data) - это термин, который был озвучен в статье под названием «Tidy Data» Хэдли Уикхэма. Я настоятельно рекомендую вам прочитать эту статью. Ее можно скачать по адресу http://vita.had.co.nz/papers/tidy-data.pdf.

В статье рассматриваются различные аспекты процесса создания аккуратных данных, конечным результатом которых является наличие данных, в которых нет сюрпризов и их можно использовать для анализа. Мы разберем различные инструменты библиотеки pandas, предназначенные для приведения данных в порядок. Они реализованы, потому что нам нужно разрешить следующие ситуации:

- Имена переменных отличаются от тех, которые вам нужны
- Есть *пропущенные* данные
- Значения указаны не в тех единицах измерения, которые вам требуются
- Временной период выборки наблюдений не тот, что вам нужен
- Переменные являются категориальными, а вам нужны количественные значения
- Присутствует *шум* в данных
- Информация неверного типа
- Данные неправильно ориентированы по осям
- Данные неправильно нормализированы
- Данные дублируются

Перед вами внушительный список, и я уверяю вас, что он еще не полный. Но это все те проблемы, с которыми я лично сталкиваюсь (и я уверен, что вы тоже). Используя инструменты и языки программирования, явно не предназначенные для решения этих проблем (чего не скажешь о библиотеке pandas), эти ситуации было бы довольно сложно разрешить. В данной главе мы рассмотрим способы устранения этих проблем с помощью библиотеки pandas.

Как работать с пропущенными данными

Данные являются **пропущенными (missing)**, если они содержат значения `NaN` (также обозначаются как `np.nan`). Значение `NaN` означает, что для конкретной метки индекса в конкретной серии не указано значение.

Как могут возникнуть пропущенные данные? Существует множество причин, по которым значение будет записано как значение `NaN`:

- Объединение двух наборов данных с несовпадающими названиями переменных
- Данные, извлеченные из внешнего источника, являются неполными
- Значение переменной неизвестно в данный момент времени и будет заполнено позднее
- При сборе информации был допущен пропуск, однако наблюдение все равно должно быть записано
- Переиндексация данных привело к индексу, у которого отсутствует значение.
- Форма данных изменилась со временем и теперь появились дополнительные строки или столбцы, которые до момента появления этих изменений нельзя было зафиксировать

Вероятно, существует большее количество причин, приводящих к появлению пропущенных данных, но общим здесь является то, что эти ситуации происходят, и вы как пользователь библиотеки pandas должны будете разрешить их, чтобы суметь выполнить эффективный анализ данных.

Давайте начнем рассмотрение способов обработки пропущенных данных, создав датафрейм с некоторыми отсутствующими значениями:

In[2]:
```
# создаем датафрейм с 5 строками и 3 столбцами
df = pd.DataFrame(np.arange(0, 15).reshape(5, 3),
                  index=['a', 'b', 'c', 'd', 'e'],
                  columns=['c1', 'c2', 'c3'])
df
```

Out[2]:

	c1	c2	c3
a	0	1	2
b	3	4	5
c	6	7	8
d	9	10	11
e	12	13	14

Здесь нет никаких пропущенных значений, поэтому давайте добавим несколько пропусков:

```
In[3]:
# добавляем несколько столбцов и строк в датафрейм
# столбец c4 со значениями NaN
df['c4'] = np.nan
# строка 'f' со значениями от 15 до 18
df.loc['f'] = np.arange(15, 19)
# строка 'g', состоящая из значений NaN
df.loc['g'] = np.nan
# столбец 'C5', состоящий из значений NaN
df['c5'] = np.nan
# меняем значение в столбце 'c4' строки 'a'
df['c4']['a'] = 20
df

Out[3]:
     c1    c2    c3    c4  c5
a   0.0   1.0   2.0  20.0 NaN
b   3.0   4.0   5.0   NaN NaN
c   6.0   7.0   8.0   NaN NaN
d   9.0  10.0  11.0   NaN NaN
e  12.0  13.0  14.0   NaN NaN
f  15.0  16.0  17.0  18.0 NaN
g   NaN   NaN   NaN   NaN NaN
```

Теперь этот датафрейм содержит пропущенные данные, которые имеют следующие характеристики:

- Одна строка, состоящая только из значений `NaN`
- Один столбец, состоящая только из значений `NaN`
- Несколько строк и столбцов, состоящих из числовых значений и значений `NaN`

Теперь давайте рассмотрим различные методы работы с этими пропущенными данными.

Поиск значений NaN в объектах библиотеки pandas

Значения `NaN` в объекте `DataFrame` можно найти с помощью метода `.isnull()`. Любое значение `True` означает, что элемент в этой позиции является значением `NaN`:

```
In[4]:
# какие элементы являются значениями NaN?
df.isnull()

Out[4]:
      c1     c2     c3     c4    c5
a  False  False  False  False  True
b  False  False  False   True  True
c  False  False  False   True  True
d  False  False  False   True  True
e  False  False  False   True  True
f  False  False  False  False  True
g   True   True   True   True  True
```

Мы можем использовать тот факт, что метод `.sum()` рассматривает `True` как `1` и `False` как `0`, чтобы вычислить количество значений `NaN` в объекте `DataFrame`:

```
In[5]:
# подсчитываем количество значений NaN
# в каждом столбце
df.isnull().sum()

Out[5]:
c1    1
```

```
c2    1
c3    1
c4    5
c5    7
dtype: int64
```

Применение метода `.sum()` к полученной серии дает общее количество значений `NaN` в исходном объекте `DataFrame`:

In[6]:
```
# вычисляем общее количество значений NaN
df.isnull().sum().sum()
```

Out[6]:
```
15
```

Это здорово, что существует простой способ выявить пропущенные значения на раннем этапе процесса обработки данных. Если вы полагаете, что ваши данные являются полными, и эта простая проверка отобразит значение, отличное от 0, значит нужно проанализировать данные более внимательно.
Еще один способ определить пропуски – использовать метод `.count()` объекта `Series` и объекта `DataFrame`. Для объекта `Series` этот метод возвращает число непропущенных значений (значений, отличных от `NaN`). Для объекта `DataFrame` он будет подсчитывать количество непропущенных значений в каждом столбце:

In[7]:
```
# вычисляем количество значений, отличных от NaN,
# по каждому столбцу
df.count()
```

Out[7]:
```
c1    6
c2    6
c3    6
c4    2
c5    0
dtype: int64
```

Теперь вычислим количество значений `NaN` немного другим способом:

In[8]:
```
# этот программный код тоже подсчитывает
# общее количество значений NaN
(len(df) - df.count()).sum()
```

Out[8]:
```
15
```

Кроме того, мы можем определить является ли элемент непропущенным значением, воспользовавшись методом `.notnull()`, который возвращает `True`, если значение является непропущенным, в противном случае он возвращает `False`:

In[9]:
```
# какие элементы являются непропущенными значениями?
df.notnull()
```
Out[9]:

	c1	c2	c3	c4	c5
a	True	True	True	True	False
b	True	True	True	False	False

```
c   True   True   True  False  False
d   True   True   True  False  False
e   True   True   True  False  False
f   True   True   True   True  False
g  False  False  False  False  False
```

Удаление пропущенных данных

Один из методов обработки пропущенных данных – простое удаление наблюдений с пропусками из вашего набора данных. Библиотека pandas делает это возможным с помощью нескольких методов. Один из них – логический отбор с использованием результатов методов `.isnull()` или `.notnull()`, которые извлекают из объекта `Series` пропущенные и непропущенные значения соответственно. Следующий программный код демонстрирует отбор всех непропущенных значений из столбца `c4` датафрейма `df`:

```
In[10]:
# отбираем непропущенные значения в столбце c4
df.c4[df.c4.notnull()]

Out[10]:
a    20.0
f    18.0
Name: c4, dtype: float64
```

Кроме того, библиотека pandas предлагает удобный метод `.dropna()`, которая удаляет пропущенные значения в серии:

```
In[11]:
# .dropna() также возвращает непропущенные значения
# этот программный код извлекает в столбце c4
# все значения, кроме значений NaN
df.c4.dropna()

Out[11]:
a    20.0
f    18.0
Name: c4, dtype: float64
```

Обратите внимание, что метод `.dropna()` фактически вернул копию датафрейма без некоторых строк. Исходный датафрейм не изменился:

```
In[12]:
# .dropna() возвращает копию с удаленными значениями
# исходный датафрейм/столбец не изменился
df.c4

Out[12]:
a    20.0
b     NaN
c     NaN
d     NaN
e     NaN
f    18.0
g     NaN
Name: c4, dtype: float64
```

Когда метод `.dropna()` применяется к объекту `DataFrame`, он удаляет из объекта `DataFrame` все строки, в которых есть хотя бы одно значение `NaN`. Следующий программный код показывает этот процесс в действии, и

поскольку каждая строка имеет по крайней мере одно значение NaN, в результате мы получим нулевые строки:

In[13]:
```
# метод .dropna() при применении к датафрейму
# удаляет целиком строки, в которых есть
# по крайней мере одно значение NaN
# в данном случае будут удалены все строки
df.dropna()
```

Out[13]:
```
Empty DataFrame
Columns: [c1, c2, c3, c4, c5]
Index: []
```

Если вы хотите удалить только те строки, в которых все значения являются значениями NaN, вы можете использовать параметр `how='all'`. Следующий программный код удаляет только строку `g`, так как у нее все значения являются значениями NaN:

In[14]:
```
# используя параметр how='all', удаляем лишь те строки,
# в которых все значения являются значениями NaN
df.dropna(how = 'all')
```

Out[14]:
```
     c1    c2    c3    c4  c5
a   0.0   1.0   2.0  20.0 NaN
b   3.0   4.0   5.0   NaN NaN
c   6.0   7.0   8.0   NaN NaN
d   9.0  10.0  11.0   NaN NaN
e  12.0  13.0  14.0   NaN NaN
f  15.0  16.0  17.0  18.0 NaN
```

Кроме того, этот программный код можно применить к столбцам вместо строк, изменив ось строк (используется по умолчанию) на ось столбцов, то есть задав `axis=1`. Следующий программный код удаляет столбец `c5`, поскольку только у него все значения являются значениями NaN:

In[15]:
```
# меняем ось, чтобы удалить столбцы со значениями NaN
# вместо строк
df.dropna(how='all', axis=1) # попрощаемся со столбцом c5
```

Out[15]:
```
     c1    c2    c3    c4
a   0.0   1.0   2.0  20.0
b   3.0   4.0   5.0   NaN
c   6.0   7.0   8.0   NaN
d   9.0  10.0  11.0   NaN
e  12.0  13.0  14.0   NaN
f  15.0  16.0  17.0  18.0
g   NaN   NaN   NaN   NaN
```

Теперь рассмотрим процесс удаления пропусков, используя немного другой объект `DataFrame`. В этом датафрейме только столбцы `c1` и `c3` не имеют значений NaN.

In[16]:
```
# создаем копию датафрейма df
df2 = df.copy()
# заменяем две ячейки с пропусками значениями 0
df2.loc['g'].c1 = 0
df2.loc['g'].c3 = 0
```

```
df2
```

Out[16]:

```
     c1    c2    c3    c4  c5
a   0.0   1.0   2.0  20.0 NaN
b   3.0   4.0   5.0   NaN NaN
c   6.0   7.0   8.0   NaN NaN
d   9.0  10.0  11.0   NaN NaN
e  12.0  13.0  14.0   NaN NaN
f  15.0  16.0  17.0  18.0 NaN
g   0.0   NaN   0.0   NaN NaN
```

С помощью параметра `how='any'` мы удалим столбцы, в котором есть хотя бы одно значение `NaN`. В данном случае все столбцы, за исключением `c1` и `c3`, будут удалены:

In[17]:

```
# а сейчас удаляем столбцы, в которых есть
# хотя бы одно значение NaN
df2.dropna(how='any', axis=1)
```

Out[17]:

```
     c1    c3
a   0.0   2.0
b   3.0   5.0
c   6.0   8.0
d   9.0  11.0
e  12.0  14.0
f  15.0  17.0
g   0.0   0.0
```

Кроме того, метод `.dropna()` имеет параметр `thresh`. Параметр `thresh` с помощью целочисленного значения задает минимальное количество значений `NaN`, которое требуется, чтобы выполнить операцию удаления строк или столбцов. Следующий программный код удаляет все столбцы, которые содержат по меньшей мере пять значений `NaN` (в данном случае это столбцы `c4` и `c5`):

In[18]:

```
# удаляем лишь те столбцы, в которых по меньшей мере
# 5 значений NaN
df.dropna(thresh=5, axis=1)
```

Out[18]:

```
     c1    c2    c3
a   0.0   1.0   2.0
b   3.0   4.0   5.0
c   6.0   7.0   8.0
d   9.0  10.0  11.0
e  12.0  13.0  14.0
f  15.0  16.0  17.0
g   NaN   NaN   NaN
```

Еще раз обратите внимание, метод `.dropna()` (и логический отбор) возвращает копию датафрейма и данные удаляются именно из этой копии. Если вы хотите удалить данные в исходном датафрейме, используйте параметр `inplace=True`.

Обработка значений NaN в ходе арифметических операций

Значения `NaN` обрабатываются в библиотеке pandas обрабатываются иначе, чем в библиотеке NumPy. Мы уже знаем это по предыдущим главах, но стоит еще раз напомнить здесь. Это отличие продемонстрировано в следующем примере:

```
In[19]:
# создаем массив NumPy с одним значением NaN
a = np.array([1, 2, np.nan, 3])
# создаем объект Series из массива
s = pd.Series(a)
# средние значения массива и серии отличаются
a.mean(), s.mean()

Out[19]:
(nan, 2.0)
```

Когда функция библиотеки NumPy встречает значение `NaN`, она возвращает `NaN`. Функции библиотеки pandas обычно игнорируют значения `NaN` и продолжают выполнять обработку данных, как если бы значения `NaN` не были частью объекта `Series`.

Обратите внимание, что среднее значение в предыдущей серии было вычислено как (1+2+3)/3 = 2, а не как (1+2+3)/4 или (1+2+0+3)/4. Это подтверждает, что значение `NaN` полностью игнорируется и даже не учитывается как элемент в объекте `Series`.

Если говорит более конкретно, способ, с помощью которого библиотека pandas обрабатывает значения `NaN`, выглядит следующим образом:

- При суммировании данных значения `NaN` обрабатываются как нулевые значения
- Если все значения являются значениями `NaN`, результатом будет значение `NaN`
- Методы типа `.cumsum()` и `.cumprod()` игнорируют значения `NaN`, но сохраняют их в полученных массивах.

Следующий программный код иллюстрирует все эти принципы:

```
In[20]:
# показываем, как методы .sum(), .mean() и .cumsum()
# обрабатывают значения NaN
# на примере столбца c4 датафрейма df
s = df.c4
s.sum(), # значения NaN обработаны как 0

Out[20]:
(38.0,)

In[21]:
s.mean() # NaN также обработаны как 0

Out[21]:
19.0
```

In[22]:
```
# в методе .cumsum() значения NaN тоже обрабатываются как 0,
# но в итоговом объекте Series значения NaN сохраняются
s.cumsum()
```

Out[22]:
```
a    20.0
b     NaN
c     NaN
d     NaN
e     NaN
f    38.0
g     NaN
Name: c4, dtype: float64
```

Но при использовании традиционных математических операторов значения `NaN` будут просто перенесены в итоговую серию:

In[23]:
```
# при выполнении арифметических операциий значение NaN
# будет перенесено в результат
df.c4 + 1
```

Out[23]:
```
a    21.0
b     NaN
c     NaN
d     NaN
e     NaN
f    19.0
g     NaN
Name: c4, dtype: float64
```

Заполнение пропущенных данных

Вместо того, чтобы оставить пропуски как есть или удалить строки/столбцы, содержащие их, можно воспользоваться методом `.fillna()`, который заменяет значения `NaN` на определенное значение. Следующий программный код демонстрирует это, заполняя значения `NaN` нулями:

In[24]:
```
# возвращаем новый датафрейм, в котором
# значения NaN заполнены нулями
filled = df.fillna(0)
filled
```

Out[24]:
```
     c1    c2    c3    c4   c5
a   0.0   1.0   2.0  20.0  0.0
b   3.0   4.0   5.0   0.0  0.0
c   6.0   7.0   8.0   0.0  0.0
d   9.0  10.0  11.0   0.0  0.0
e  12.0  13.0  14.0   0.0  0.0
f  15.0  16.0  17.0  18.0  0.0
g   0.0   0.0   0.0   0.0  0.0
```

Имейте в виду, что это изменит итоговые значения. В качестве примера следующий программный код показывает сравнительные результаты применения метода `.mean()` к датафрейму со значениями `NaN` и датафрейму, в котором значения `NaN` были заполнены нулями:

In[25]:
```
# значения NaN не учитываются при вычислении
# средних значений
df.mean()
```

Out[25]:
```
c1     7.5
c2     8.5
c3     9.5
c4    19.0
c5     NaN
dtype: float64
```

In[26]:
```
# после замены значений NaN на 0 получим
# другие средние значения
filled.mean()
```

Out[26]:
```
c1    6.428571
c2    7.285714
c3    8.142857
c4    5.428571
c5    0.000000
dtype: float64
```

Прямое и обратное заполнение пропущенных значений

Пропуски в данных можно заполнить с помощью последнего непропущенного значения в прямом и обратном порядке. Для иллюстрации следующий программный код заполнит столбец `c4` датафрейма `df` в прямом порядке:

In[27]:
```
# заполняем пропуски в столбце c4 датафрейма df
# в прямом порядке
df.c4.fillna(method="ffill")
```

Out[27]:
```
a    20.0
b    20.0
c    20.0
d    20.0
e    20.0
f    18.0
g    18.0
Name: c4, dtype: float64
```

При работе с данными, представленными в виде временных рядов, этот метод заполнения часто называют методом «последнего известного значения». Мы рассмотрим это в главе, посвященной анализу временных рядов.

Теперь заполним пропуски в обратном порядке с помощью `method='bfill'`.

In[28]:
```
# выполняем обратное заполнение
df.c4.fillna(method="bfill")
```

```
Out[28]:
a    20.0
b    18.0
c    18.0
d    18.0
e    18.0
f    18.0
g     NaN
Name: c4, dtype: float64
```

Чтобы сохранить небольшую типизацию, библиотека pandas еще предлагает глобальные функции `pd.ffill()` и `pd.bfill()`, которые эквивалентны `.fillna(method = "ffill")` и `.fillna(method="bfill")`.

Заполнение с помощью меток индекса

Данные можно заполнить с помощью меток объекта `Series` или ключей питоновского словаря. Это позволяет задавать для заполнения разные значения для различных элементов, используя индексные метки:

```
In[29]:
# создаем новую серию значений, которую используем
# для заполнения значений NaN там,
# где метки индекса будут совпадать
fill_values = pd.Series([100, 101, 102], index=['a', 'e', 'g'])
fill_values

Out[29]:
a    100
e    101
g    102
dtype: int64

In[30]:
# заполняем пропуски в столбце c4 с помощью fill_values
# a, e и g будут заполнены, поскольку метки совпали,
# однако значение a не изменится, потому что оно
# не является пропуском
df.c4.fillna(fill_values)

Out[30]:
a     20.0
b      NaN
c      NaN
d      NaN
e    101.0
f     18.0
g    102.0
Name: c4, dtype: float64
```

Заполнены только значения `NaN`. Обратите внимание, что значение с меткой `a` не изменилось.
Еще один распространенный сценарий – заполнение всех значений `NaN` в столбце средним значением столбца:

```
In[31]:
# заполняем значения NaN в каждом столбце
# средним значением этого столбца
df.fillna(df.mean())
```

```
Out[31]:
     c1    c2    c3    c4  c5
a   0.0   1.0   2.0  20.0 NaN
b   3.0   4.0   5.0  19.0 NaN
c   6.0   7.0   8.0  19.0 NaN
d   9.0  10.0  11.0  19.0 NaN
e  12.0  13.0  14.0  19.0 NaN
f  15.0  16.0  17.0  18.0 NaN
g   7.5   8.5   9.5  19.0 NaN
```

Это удобно, поскольку пропуски, замененные таким способом, искажают статистическое среднее в меньшей степени, чем заполнение пропусков нулями. В некоторых видах статистического анализа такой способ может быть приемлемым, когда большая вариабельность данных в силу использования нулевых значений стало причиной неверных прогнозов.

Выполнение интерполяции пропущенных значений

Метод `.interpolate()` объекта `Series` и объекта `DataFrame` выполняет по умолчанию линейную интерполяцию пропущенных значений:

```
In[32]:
# выполняем линейную интерполяцию
# значений NaN с 1 по 2
s = pd.Series([1, np.nan, np.nan, np.nan, 2])
s.interpolate()

Out[32]:
0    1.00
1    1.25
2    1.50
3    1.75
4    2.00
dtype: float64
```

Значение, используемое для интерполяции, вычисляется следующим образом. Сначала на основе первого значения, предшествующего ряду значений `NaN`, и последнего значения, следующего после ряда значений `NaN`, вычисляем инкремент (величину приращения). Затем пошагово увеличиваем стартовое значение на величину этого инкремента и заполняем им значения `NaN`. В нашем случае 2.0 и 1.0 представляют собой значения, окружающие пропуски, в результате получаем (2.0 - 1.0)/(5-1) = 0.25. Берем стартовое значение 1, увеличиваем на 0.25, получаем значение 1.25 и заполняем им первое значение `NaN`. Затем значение 1.25 увеличиваем снова на 0.25, получаем 1.50, заменяем им второе значение `NaN` и так далее. Это важно.

Представьте, что ваши данные представляют собой набор возрастающих значений, например, повышение температуры в течение дня. Если датчик перестает фиксировать температуру в определенные временные интервалы, пропуски можно заполнить путем интерполяции с высокой степенью достоверности. Это определенно лучше, чем замена пропусков нулями.

Кроме того, метод интерполяции позволяет указать конкретный метод интерполяции. Одним из распространенных методов является

интерполяция на основе времени. Рассмотрим следующий объект `Series` с датами и значениями:

```
In[33]:
# создаем временной ряд, но при этом значение
# по одной дате будет пропущено
ts = pd.Series([1, np.nan, 2],
               index=[datetime(2014, 1, 1),
                      datetime(2014, 2, 1),
                      datetime(2014, 4, 1)])
ts

Out[33]:
2014-01-01    1.0
2014-02-01    NaN
2014-04-01    2.0
dtype: float64
```

Предыдущий вид интерполяции приведет к следующему результату:

```
In[34]:
# выполняем линейную интерполяцию на основе
# количества элементов в данной серии
ts.interpolate()

Out[34]:
2014-01-01    1.0
2014-02-01    1.5
2014-04-01    2.0
dtype: float64
```

Значение для `2014-02-01` рассчитывается как 1.0+(2.0-1.0)/2=1.5, потому что только одно значение `NaN` находится между значениями `2.0` и `1.0`.
Важно отметить, что в серии отсутствует наблюдение для 2014-03-01. Если нам нужно интерполировать суточные значения, следует вычислить два значения: одно для 2014-02-01, а другое для 2014-03-01, что приведет к другой величине инкремента и соответственно другим результатам.
Давайте выполним интерполяцию суточных значений, указав метод интерполяции `time`:

```
In[35]:
# этот программный код учитывает тот факт,
# что у нас отсутствует запись для 2014-03-01
ts.interpolate(method="time")

Out[35]:
2014-01-01    1.000000
2014-02-01    1.344444
2014-04-01    2.000000
dtype: float64
```

Мы выполнили корректную интерполяцию для `2014-02-01` на основе дат. Кроме того, обратите внимание, что метка индекса и значение для `2014-03-01` не были добавлены в объект `Series`, это значение просто учитывается при выполнении интерполяции.
Помимо этого интерполяцию можно задать для вычисления значений относительно меток индекса, когда используются числовые индексные

метки. Чтобы продемонстрировать это, создадим следующий объект `Series`:

```
In[36]:
# создаем объект Series, чтобы продемонстрировать интерполяцию,
# основанную на индексных метках
s = pd.Series([0, np.nan, 100], index=[0, 1, 10])
s

Out[36]:
0       0.0
1       NaN
10    100.0
dtype: float64
```

Если мы выполним линейную интерполяцию, мы получим следующее значение для метки `1`:

```
In[37]:
# выполняем линейную интерполяцию
s.interpolate()

Out[37]:
0       0.0
1      50.0
10    100.0
dtype: float64
```

Значение для метки `1` было вычислено как 0,0+(100,0-0,0)/(3-1)=50. Однако что если мы хотим интерполировать значение относительно значения индекса? Для этого можно воспользоваться `method="values"`:

```
In[38]:
# выполняем интерполяцию на основе значений индекса
s.interpolate(method="values")

Out[38]:
0       0.0
1      10.0
10    100.0
dtype: float64
```

Значение, рассчитанное для пропуска теперь интерполируется с помощью относительного позиционирования на основе индексных меток. Значение `NaN` имеет метку `1`, что составляет одну десятую расстояния между `0` и `10`, поэтому интерполированное значение будет равно 0.0+ (100.0-0.0)/10 или `10`.

Обработка дублирующихся данных

Данные в вашем примере часто могут содержать дублирующиеся строки. Это обычная реальность, связанная с данными, собираемыми в автоматическом режиме или даже собираемыми вручную. Дублирующиеся данные считаются ошибкой, особенно в тех случаях, когда эти данные можно считать идемпотентными. Однако дублирующиеся данные могут увеличить размер набора данных, а если они не являются идемпотентными, то обрабатывать их как дублирующиеся было бы нецелесообразно.

Библиотека pandas предлагает метод `.duplicates()` для упрощения поиска дублирующихся данных. Этот метод возвращает серию логических значений, в которой каждая запись показывает, является ли строка дубликатом или нет. Значение `True` означает, что данная строка уже появлялась ранее в объекте `DataFrame`, значения в столбцах совпадают.
Следующий программный код иллюстрирует поиск дубликатов в действии, создавая объект `DataFrame` с повторяющимися строками:

```
In[39]:
# создаем датафрейм с дублирующимися строками
data = pd.DataFrame({'a': ['x'] * 3 + ['y'] * 4,
                     'b': [1, 1, 2, 3, 3, 4, 4]})
data
```

```
Out[39]:
   a  b
0  x  1
1  x  1
2  x  2
3  y  3
4  y  3
5  y  4
6  y  4
```

Теперь давайте найдем дублирующиеся строки:

```
In[40]:
# определяем, какие строки являются дублирующимися,
# то есть какие строки уже ранее встречались в датафрейме
data.duplicated()
```

```
Out[40]:
0    False
1     True
2    False
3    False
4     True
5    False
6     True
dtype: bool
```

Дублирующиеся строки можно удалить из датафрейма с помощью метода `.drop_duplicates()`. Этот метод возвращает копию датафрейма с удаленными дублирующимися строками:

```
In[41]:
# удаляем дублирующиеся строки, каждый раз оставляя
# первое из дублирующихся наблюдений
data.drop_duplicates()
```

```
Out[41]:
   a  b
0  x  1
2  x  2
3  y  3
5  y  4
```

Кроме того, можно воспользоваться параметром `inplace=True`, чтобы удалить строки, не создавая копию датафрейма.
Обратите внимание, что существует нюанс, связанный с метками индексов, остающимися при удалении дубликатов. Дублирующиеся записи могут иметь разные метки индексы (метки не учитываются при

поиске дубликатов). Таким образом, строка, которая будет оставлена при удалении дубликатов, может повлиять на набор меток в итоговом объекте `DataFrame`.
По умолчанию при удалении дубликатов каждый раз оставляется строка, встретившаяся первой. Если вы хотите, чтобы при удалении дубликатов каждый раз оставлялась последняя строка, используйте параметр `keep='last'`. Следующий программный код показывает, как будет отличаться результат при использовании этого параметра:

In[42]:
```
# удаляем дублирующиеся строки, каждый раз оставляя
# последнее из дублирующихся наблюдений
data.drop_duplicates(keep='last')
```

Out[42]:
```
   a  b
1  x  1
2  x  2
4  y  3
6  y  4
```

Если вы хотите проверить наличие дубликатов, используя меньший набора столбцов, вы можете задать список имен столбцов:

In[43]:
```
# добавляем столбец с со значениями от 0 до 6
# метод .duplicated() сообщает об отсутствии
# дублирующихся строк
data['c'] = range(7)
data.duplicated()
```

Out[43]:
```
0    False
1    False
2    False
3    False
4    False
5    False
6    False
dtype: bool
```

In[44]:
```
# но если мы укажем, что нужно удалить дублирующиеся строки
# с учетом значений в столбцах a и b,
# результаты будут выглядеть так
data.drop_duplicates(['a', 'b'])
```

Out[44]:
```
   a  b  c
0  x  1  0
2  x  2  2
3  y  3  3
5  y  4  5
```

Преобразование данных

Кроме того, приведение данных в порядок подразумевает преобразование существующих данных в другое представление. Это может быть обусловлено следующими причинами:

- Значения имеют неверные единицы измерения

- Значения являются качественными и нуждаются в преобразовании в соответствующие числовые значения
- Есть посторонние данные, которые либо потребляют лишнюю память и увеличивают время обработки, либо могут повлиять на результаты анализа

Чтобы разрешить эти ситуации, мы можем выполнить одно или несколько действий из нижеописанных:

- Сопоставляем значения другим значениям, используя поиск по таблице
- Явно заменяем определенные значения другими значениями (или даже другим типом данных)
- Применяем методы, чтоб преобразовать значения на основе алгоритма Просто удаляем посторонние столбцы и строки

Мы уже видели, как можно удалить строки и столбцы с помощью нескольких методов, поэтому не будем повторно разбирать их здесь. Теперь мы рассмотрим возможности, предлагаемые библиотекой pandas для сопоставления, замены и применения функций, преобразующих данные на основе их содержимого.

Сопоставление значений другим значениям

Одной из основных задач преобразования данных является сопоставление набора значений другому набору значений. Библиотека pandas позволяет сопоставлять значения с помощью таблицы поиска (по питоновскому словарю или серии библиотеки pandas). Для этого используется метод `.map()`.

Этот метод выполняет сопоставление, сперва сличая значения внешней серии с метками индекса внутренней серии. Затем он возвращает новую серию с метками индекса внешней серии, но со значениями внутренней серии.

Следующий пример показывает, как сопоставить метки индекса `x` значениям `y`:

```
In[45]:
# создаем два объекта Series для иллюстрации
# процесса сопоставления значений
x = pd.Series({"one": 1, "two": 2, "three": 3})
y = pd.Series({1: "a", 2: "b", 3: "c"})
x

Out[45]:
one      1
three    3
two      2
dtype: int64

In[46]:
y

Out[46]:
1    a
2    b
3    c
dtype: object

In[47]:
# сопоставляем значения серии x значениям серии y
x.map(y)
```

```
Out[47]:
one      a
two      b
three    c
dtype: object
```

Как и в случае остальных операций выравнивания, если библиотека pandas не найдет соответствия между значением внешней серии и индексной меткой внутренней серии, она пометит это значение как пропущенное. Для иллюстрации следующий программный код удаляет ключ `3` из внешней серии, что приводит к сбою выравнивания для данной записи и появлению значения `NaN`:

```
In[48]:
# если между значением серии y и индексной меткой серии x
# не будет найдено соответствие, будет выдано значение NaN
x = pd.Series({"one": 1, "two": 2, "three": 3})
y = pd.Series({1: "a", 2: "b"})
x.map(y)
```

```
Out[48]:
one        a
two        b
three    NaN
dtype: object
```

Замена значений

Ранее мы видели, как метод `.fillna()` может использоваться для замены значений `NaN` теми значениями, которые вы задали сами. Фактически метод `.fillna()` можно рассматривать как реализацию метода `.map()`, который сопоставляет значение `NaN` с определенным значением.

Еще говорить более широко, метод `.fillna()` может рассматривать как специальный случай более универсальной операции замены значений, которая реализована в рамках метода `.replace()`. Этот метод более гибок благодаря возможности заменить любое значение (не только значение `NaN`) другим значением.

Основное использование метода `.replace()` заключается в замене конкретного значения на другое:

```
In[49]:
# создаем объект Series, чтобы проиллюстрировать
# метод .replace()
s = pd.Series([0., 1., 2., 3., 2., 4.])
s
```

```
Out[49]:
0    0.0
1    1.0
2    2.0
3    3.0
4    2.0
5    4.0
dtype: float64
```

```
In[50]:
# заменяем значение, соответствующее
# индексной метке 2, на значение 5
s.replace(2, 5)
```

```
Out[50]:
0    0.0
1    1.0
2    5.0
3    3.0
4    5.0
5    4.0
dtype: float64
```

Кроме того, можно задать несколько элементов, которые нужно заменить, а также указать их заменяющие значения, передав два списка (первый - список заменяемых значений, а второй – список заменяющих значений):

```
In[51]:
# заменяем все элементы новыми значениями
s.replace([0, 1, 2, 3, 4], [4, 3, 2, 1, 0])

Out[51]:
0    4.0
1    3.0
2    2.0
3    1.0
4    2.0
5    0.0
dtype: float64
```

Еще замену значений можно выполнить, указав словарь для поиска (как вариант сопоставления значений, рассмотренного в предыдущем разделе):

```
In[52]:
# заменяем элементы, используя словарь
s.replace({0: 10, 1: 100})

Out[52]:
0     10.0
1    100.0
2      2.0
3      3.0
4      2.0
5      4.0
dtype: float64
```

Если вы используете метод `.replace()` объекта `DataFrame`, можно указать разные заменяемые значения для каждого столбца. Эта операция выполняется путем передачи питоновского словаря в метод `.replace()`. В этом словаре ключи представляют собой имена столбцов, в которых нужно заменить значения, а значения словаря указывают заменяемые значения в столбцах. Второй параметр метода – это значение, которое используется для замены. Следующий программный код демонстрирует данную операцию, создав объект `DataFrame` и затем заменив конкретные значения в каждом столбце на значение `100`:

```
In[53]:
# создаем датафрейм с двумя столбцами
df = pd.DataFrame({'a': [0, 1, 2, 3, 4], 'b': [5, 6, 7, 8, 9]})
df

Out[53]:
   a  b
0  0  5
```

```
1  1  6
2  2  7
3  3  8
4  4  9
```

In[54]:
```
# задаем разные заменяемые значения для каждого столбца
df.replace({'a': 1, 'b': 8}, 100)
```

Out[54]:
```
     a    b
0    0    5
1  100    6
2    2    7
3    3  100
4    4    9
```

Замена определенных значений в каждом столбце очень удобна, так как она экономит время, которое в противном случае потребовало бы для написания цикла, перебирающего все столбцы.
Кроме того, можно заменить значения в определенных позициях индекса, как если бы они были пропущены. Следующий программный код демонстрирует это путем прямого заполнения значений в позициях индекса 1, 2 и 3:

In[55]:
```
# иллюстрируем замену значений с помощью метода pad
# заменяем первое значение на 10
s[0] = 10
s
```

Out[55]:
```
0    10.0
1     1.0
2     2.0
3     3.0
4     2.0
5     4.0
dtype: float64
```

In[56]:
```
# заменяем элементы с индексными метками 1, 2, 3, используя
# для заполнения самое последнее значение, предшествующее
# заданным меткам (10)
s.replace([1, 2, 3], method='pad')
```

```
Out[56]:
0    10.0
1    10.0
2    10.0
3    10.0
4    10.0
5     4.0
dtype: float64
```

Кроме того, можно выполнить прямое и обратное заполнение с помощью `ffill` и `bfill` в качестве указанных методов, но они будут оставлены вам в качестве упражнения для самостоятельной работы.

Применение функций для преобразования данных

В ситуациях, когда прямого сопоставления или замены значений недостаточно, к данным можно применить функцию, выполняющую определенный алгоритм действий. Библиотека pandas позволяет применять функции к отдельным элементам, целым столбцам или целым строкам, обеспечивая невероятную гибкость преобразований.

При работе с данными мы можем применить различные функции, воспользовавшись методом с говорящим названием `.apply()`. Когда задана функция Python, метод `.apply()` итеративно вызывает ее, передавая каждое значение из объекта `Series`. Метод `.apply()` объекта `DataFrame` передает в функцию каждый столбец, представленный в виде объекта `Series`, или, если задан параметр `axis=1`, передает объект `Series`, представляющий каждую строку.

Следующий программный код демонстрирует это, применяя лямбда-функцию к каждому элементу объекта `Series`:

```
In[57]:
# иллюстрируем применение функции к каждому
# элементу объекта Series
s = pd.Series(np.arange(0, 5))
s.apply(lambda v: v * 2)
```

```
Out[57]:
0    0
1    2
2    4
3    6
4    8
dtype: int64
```

Когда функция применяется к элементам объекта `Series`, только значение каждого элемента серии (а не метка или значение индекса) передается в функцию.

Когда функция применяется к объекту `DataFrame`, по умолчанию происходит применение метода к каждому столбцу. Метод перебирает все столбцы, передавая каждый столбец, представленный в виде объекта `Series`, в вашу функцию. Итогом становится объект `Series`, который содержит метки индекса, соответствующие именам столбцов, и результаты функции, примененной к столбцам:

```
In[58]:
# создаем датафрейм, чтобы проиллюстрировать применение
# операции суммирования к каждому столбцу
df = pd.DataFrame(np.arange(12).reshape(4, 3),
                  columns=['a', 'b', 'c'])
df
```

```
Out[58]:
   a   b   c
0  0   1   2
1  3   4   5
2  6   7   8
3  9  10  11
```

```
In[59]:
# вычисляем сумму элементов в каждом столбце
df.apply(lambda col: col.sum())
```

```
Out[59]:
a    18
b    22
c    26
dtype: int64
```

Применение функции можно переключить на подсчет суммы элементов в каждой строке, указав `axis=1`:

```
In[60]:
# вычисляем сумму элементов в каждой строке
df.apply(lambda row: row.sum(), axis=1)
```

```
Out[60]:
0     3
1    12
2    21
3    30
dtype: int64
```

Общераспространенная практика заключается в том, чтобы взять результат операции и добавить его в качестве нового столбца объекта `DataFrame`. Это удобно, поскольку вы можете добавить в объект `DataFrame` результат одной или нескольких операций последовательного вычисления, выводя итоги вычислений на каждом этапе обработки.
Следующий программный код демонстрирует этот процесс. На первом этапе происходит умножение значений столбца `a` на значения столбца `b` и создается новый столбец под названием `interim`. На втором этапе происходит сложение значений столбца `interim` и значений столбца `c`, в итоге создается столбец `result` с получившимися значениями:

```
In[61]:
# создаем столбец 'interim' путем
# умножения столбцов a и b
df['interim'] = df.apply(lambda r: r.a * r.b, axis=1)
df
```

```
Out[61]:
   a   b   c  interim
0  0   1   2        0
1  3   4   5       12
2  6   7   8       42
3  9  10  11       90
In[62]:
# а теперь получаем столбец 'result' путем сложения
# столбцов 'interim' и 'c'
df['result'] = df.apply(lambda r: r.interim + r.c, axis=1)
df

Out[62]:
   a   b   c  interim  result
0  0   1   2        0       2
1  3   4   5       12      17
2  6   7   8       42      50
3  9  10  11       90     101
```

Можно изменить значения в существующем столбце, просто присвоив этому столбцу результат выполнения той или иной операции. Следующий программный код изменяет значения столбца `a` на сумму значений по строке:

```
In[63]:
# заменяем значения столбца a на сумму значений по строке
df.a = df.a + df.b + df.c
df

Out[63]:
    a   b   c  interim  result
0   3   1   2        0       2
1  12   4   5       12      17
2  21   7   8       42      50
3  30  10  11       90     101
```

Как правило, замена значений столбца совершенно новыми значениями – не лучшее решение и часто приводит к проблемам, обусловленным неудачным преобразованием данных. Поэтому в библиотеке pandas лучше просто добавлять новые строки или столбцы (или полностью новые объекты), а если впоследствии проблема памяти или производительности становится ощутимой, выполнять оптимизацию по мере необходимости.

Еще один момент, который необходимо отметить: объект `DataFrame` - это не электронная таблица, где ячейкам присваиваются формулы и их можно пересчитать, когда ячейки, на которые ссылается формула, изменяются. Если вам это нужно, необходимо всякий раз при изменении зависимых данных делать повторные вычисления. С другой стороны, это более эффективно, чем электронные таблицы, поскольку каждое небольшое изменение данных не вызывает целого каскада операций.

Метод `.apply()` всегда применяет заданную функцию ко всем элементам объекта `Series`, столбца или строки объекта `DataFrame`. Если вы хотите применить эту функцию к подмножеству данных, сначала выполните логический отбор, чтобы отфильтровать ненужные элементы.

Следующий программный код демонстрирует это, создав объект `DataFrame` со значениями и вставив одно значение `NaN` во вторую строку.

Затем он применяет функцию только к тем строкам, в которых отсутствуют значения NaN:

```
In[64]:
# создаем объект DataFrame из 3 строк и 5 столбцов
# только вторая строка содержит значение NaN
df = pd.DataFrame(np.arange(0, 15).reshape(3,5))
df.loc[1, 2] = np.nan
df

Out[64]:
    0   1     2   3   4
0   0   1   2.0   3   4
1   5   6   NaN   8   9
2  10  11  12.0  13  14

In[65]:
# демонстрируем применение функции только к тем строкам,
# в которых нет значений NaN
df.dropna().apply(lambda x: x.sum(), axis=1)

Out[65]:
0    10.0
2    60.0
dtype: float64
```

Последний способ, который мы рассмотрим в этой главе – применение функций с помощью метода `.applymap()` объекта `DataFrame`. В то время как метод `.apply()` всегда применяет функцию к строке или столбцу целиком, метод `.applymap()` применяет функцию к каждому отдельному значению. Следующий программный код демонстрирует практическое использование метода `.applymap()`, чтобы отформатировать каждое значение объекта `DateFrame` до указанного количества десятичных знаков:

```
In[66]:
# используем метод .applymap(), чтобы изменить формат
# всех элементов объекта DataFrame
df.applymap(lambda x: '%.2f' % x)

Out[66]:
       0      1      2      3      4
0   0.00   1.00   2.00   3.00   4.00
1   5.00   6.00    nan   8.00   9.00
2  10.00  11.00  12.00  13.00  14.00
```

Выводы

В этой главе мы рассмотрели различные методы приведения данных в порядок. Мы рассказали о том, как обнаружить пропущенные данные, заменить их другими значениями или удалить их из общего набора данных. Затем мы рассмотрели способы преобразовать значения в другие значения, которые, возможно, лучше подходят для проведения дальнейшего анализа.

Теперь, когда мы привели наши данные, записанные в объекте DataFrame или объекте Series, в порядок, нам нужно перейти к более сложным формам модификации структуры данных, таким как

конкатенация, слияние, объединение и вращение данных. Это и будет предметом обсуждения в следующей главе.

ГЛАВА 11 ОБЪЕДИНЕНИЕ, СВЯЗЫВАНИЕ И ИЗМЕНЕНИЕ ФОРМЫ ДАННЫХ

Данные часто можно представить в виде набора сущностей, логических структур взаимосвязанных значений, на которые ссылаются имена сущностей (свойства/переменные), при этом у сущностей есть наблюдения или экземпляры, представленные в виде строк. Сущности, как правило, представляют собой вполне реальные объекты, например, клиента или, если речь идет об интернете вещей, датчик. Затем каждую конкретную сущность и ее наблюдения можно представить с помощью отдельного датафрейма.

Часто возникает необходимость выполнить те или иные операции над сущностями и между ними. Возможно, потребуется объединить данные по нескольким сущностям-клиентам, которые были получены из нескольких источников, в один объект библиотеки pandas. Сущности «Клиент» и «Заказ» необходимо связать друг с другом, чтобы быстрее найти адрес доставки для выполнения заказа. Кроме того, возможно, что данные, хранящиеся в одной модели, потребуется преобразовать в другую модель, изменив форму данных, просто потому, что разные источники регистрируют один и тот же тип объектов по-разному.

В этой главе мы рассмотрим все эти операции, которые позволяют объединять, связывать и изменять форму данных в нашей модели. В частности, в этой главе мы рассмотрим следующие понятия:

- Конкатенация данных, расположенных в нескольких объектах библиотеки pandas
- Слияние данных, расположенных в нескольких объектов pandas
- Как настроить тип соединения при выполнении слияния
- Поворот данных для преобразования значений в индексы и наоборот
- Состыковка и расстыковка данных
- Расплавление данных для преобразования «широкого» формата в «длинный» и наоборот

Настройка библиотеки pandas

Мы начнем рассмотрение примеров в этой главе, задав следующие инструкции для импорта библиотек и настройки вывода:

```
In[1]:
# импортируем библиотеки numpy и pandas
import numpy as np
import pandas as pd

# импортируем библиотеку datetime для работы с датами
import datetime
from datetime import datetime, date

# Задаем некоторые опции библиотеки pandas, которые
# настраивают вывод
pd.set_option('display.notebook_repr_html', False)
pd.set_option('display.max_columns', 8)
pd.set_option('display.max_rows', 10)
pd.set_option('display.width', 60)

# импортируем библиотеку matplotlib для построения графиков
import matplotlib.pyplot as plt
%matplotlib inline
```

Конкатенация данных, расположенных в нескольких объектах

Конкатенация (concatenation) в библиотеке pandas представляет собой процесс объединения данных, расположенных в двух или более объектах, в новый объект. Конкатенация объектов `Series` просто приводит к созданию нового объекта `Series` с последовательно скопированными значениями.

Процесс конкатенации объектов `DataFrame` более сложен. Конкатенацию можно применять к любой оси указанных объектов. Вдоль заданной оси осуществляется логическая операция соединения индексных меток. Затем вдоль противоположной оси происходит выравнивание меток и заполнение пропущенных значений.

Поскольку существует ряд важных факторов, мы рассмотрим следующие темы:

- Понимание семантики конкатенации, принятой по умолчанию
- Переключение осей выравнивания
- Определение типа соединения
- Присоединение данных вместо конкатенации
- Игнорирование меток индекса

Понимание семантики конкатенации, принятой по умолчанию

Конкатенацию можно выполнить с помощью функции библиотеки pandas `pd.concat()`. Общий синтаксис, выполняющий конкатенацию данных, состоит в том, чтобы передать список объектов, которые будут конкатенированы. Следующий программный код демонстрирует простую конкатенацию двух объектов `Series` `s1` и `s2`:

```
In[2]:
# создаем два объекта Series для конкатенации
s1 = pd.Series(np.arange(0, 3))
s2 = pd.Series(np.arange(5, 8))
s1

Out[2]:
0    0
1    1
2    2
dtype: int64

In[3]:
s2

Out[3]:
0    5
1    6
2    7
dtype: int64

In[4]:
# конкатенируем их
pd.concat([s1, s2])

Out[4]:
0    0
1    1
2    2
0    5
1    6
2    7
dtype: int64
```

Вышеприведенный программный код присоединяет метки индекса и значения объекта `s2` к конечной метке индекса и конечному значению объекта `s1`. В результате получаем дублирующиеся индексные метки, поскольку в ходе этой операции выравнивание не выполняется.
Два объекта `DataFrame` тоже можно конкатенировать аналогичным образом:

```
In[5]:
# создаем два объекта DataFrame для конкатенации,
# используя те же самые индексные метки и имена столбцов,
# но другие значения
df1 = pd.DataFrame(np.arange(9).reshape(3, 3),
                   columns=['a', 'b', 'c'])
#df2 имеет значения 9 .. 18
df2 = pd.DataFrame(np.arange(9, 18).reshape(3, 3),
                   columns=['a', 'b', 'c'])
df1

Out[5]:
   a  b  c
0  0  1  2
1  3  4  5
2  6  7  8

In[6]:
df2

Out[6]:
    a   b   c
0   9  10  11
1  12  13  14
2  15  16  17

In[7]:
```

```
# выполняем конкатенацию
pd.concat([df1, df2])
```

```
Out[7]:
    a   b   c
0   0   1   2
1   3   4   5
2   6   7   8
0   9  10  11
1  12  13  14
2  15  16  17
```

По умолчанию строки добавляются по порядку и в результате мы можем получить дублирующиеся индексные метки вдоль индекса строк.
Итоговый набор меток столбцов получается в результате объединения индексных меток в указанных нами объектах `DataFrame`. По сути это выравнивание, которое применяется ко всем исходным объектам (их может быть больше двух). Теперь рассмотрим ситуацию, когда в обрабатываемом объекте `DataFrame` отсутствует столбец, но при этом он присутствует в другом обрабатываемом объекте `DataFrame`. Тогда в итоговом датафрейме отсутствующие значения столбца будут заполнены пропусками.
Следующий программный код демонстрирует выравнивание двух объектов `DataFrame` в ходе конкатенации, когда есть общие столбцы (`a` и `c`), а также столбцы, присутствующие только в одном датафрейме (столбец `b` в датафрейме `df1` и столбец `d` в датафрейме `df2`). Обратите внимание, поскольку ось столбцов, не участвующая в конкатенации, не выровнена (датафреймы имеют неидентичный набор столбцов), надо явно задать то или иное значение параметра `sort`, чтобы избежать предупреждения `FutureWarning: Sorting because non-concatenation axis is not aligned. A future version of pandas will change to not sort by default`.

```
In[8]:
# демонстрируем конкатенацию двух объектов DataFrame
# с разными столбцами
df1 = pd.DataFrame(np.arange(9).reshape(3, 3),
                   columns=['a', 'b', 'c'])
df2 = pd.DataFrame(np.arange(9, 18).reshape(3, 3),
                   columns=['a', 'c', 'd'])
df1
```

```
Out[8]:
   a  b  c
0  0  1  2
1  3  4  5
2  6  7  8
```

```
In[9]:
df2
```

```
Out[9]:
    a   c   d
0   9  10  11
1  12  13  14
2  15  16  17
```

```
In[10]:
# выполняем конкатенацию, пропусками будут заполнены
# значения столбца d в датафрейме df1 и
```

```
# значения столбца b в датафрейме df2
pd.concat([df1, df2], sort=True)
```

Out[10]:

```
    a    b   c     d
0   0  1.0   2   NaN
1   3  4.0   5   NaN
2   6  7.0   8   NaN
0   9  NaN  10  11.0
1  12  NaN  13  14.0
2  15  NaN  16  17.0
```

В датафрейме `df1` нет столбца `d`, поэтому значения в той части итогового датафрейма, что соответствует датафрейму `d1`, заполнены пропусками. То же самое происходит в столбце `b` датафрейма `d2`.

Каждой группе данных в итоговом датафрейме можно дать свое название, воспользовавшись параметром `keys`. Он создает иерархический индекс объекта `DataFrame`, который позволяет отдельно обращаться к каждой группе данных с помощью свойства `.loc` объекта `DataFrame`. Это удобно, если позже в итоговом объекте `DataFrame` вам потребуется определить источник получения данных.

Следующий программный код демонстрирует этот принцип: он присваивает имена каждому исходному объекту `DataFrame`, а затем извлекает строки, которые возникли в результате конкатенации датафрейма `df1` и датафрейма `df2` (строки теперь проиндексированы с помощью метки `'df2'`):

In[11]:

```
# выполняем конкатенацию двух объектов,
# но при этом создаем индекс с помощью
# заданных ключей
c = pd.concat([df1, df2], keys=['df1', 'df2'], sort=True)
# обратите внимание на метки строк в выводе
c
```

Out[11]:

```
        a    b   c     d
df1 0   0  1.0   2   NaN
    1   3  4.0   5   NaN
    2   6  7.0   8   NaN
df2 0   9  NaN  10  11.0
    1  12  NaN  13  14.0
    2  15  NaN  16  17.0
```

Затем эти ключи можно использовать, чтобы отобрать поднабор данных:

In[12]:

```
# мы можем извлечь данные, относящиеся к первому
# или второму исходному датафрейму
c.loc['df2']
```

Out[12]:

```
    a   b   c     d
0   9 NaN  10  11.0
1  12 NaN  13  14.0
2  15 NaN  16  17.0
```

Переключение осей выравнивания

Функция `pd.concat()` позволяет задать ось, к которой нужно применить выравнивание во время конкатенации. Следующий программный код конкатенирует два объекта `DataFrame` по оси столбцов, а выравнивание осуществляется по индексу строк. При этом явно задавать то или иное значение параметра `sort` не нужно, потому что ось строк, не участвующая в конкатенации, выровнена (датафреймы имеют одинаковое количество строк).

In[13]:
```
# конкатенируем датафреймы df1 и df2 по оси столбцов
# выравниваем по меткам строк,
# получаем дублирующиеся столбцы
pd.concat([df1, df2], axis=1)
```

Out[13]:

	a	b	c	a	c	d
0	0	1	2	9	10	11
1	3	4	5	12	13	14
2	6	7	8	15	16	17

В результате получаем дублирующиеся столбцы. Это связано с тем, что в ходе конкатенации сначала происходит выравнивание по меткам индекса строк каждого объекта `DataFrame`, а затем осуществляется заполнение значений столбцов для первого объекта `DataFrame` и второго объекта `DataFrame` независимо от меток индекса строк.

Следующий программный код демонстрирует конкатенацию двух объектов `DataFrame` по оси столбцов. При этом датафреймы имеют общую метку индекса строк (метка `2`), а также отличающиеся метки (метки `0` и `1` в датафрейме `df1`, метки `3` и `4` в датафрейме `df3`). Столбец `a` присутствует и в датафрейме `df1` и в датафрейме `df3`, столбцы `b` и `c` есть только в датафрейме `df1`, а столбец `d` есть только в датафрейме `df3`:

In[14]:
```
# создаем новый датафрейм d3, чтобы конкатенировать его
# с датафреймом df1
# датафрейм df3 имеет общую с датафреймом df1
# метку 2 и общий столбец a
df3 = pd.DataFrame(np.arange(20, 26).reshape(3, 2),
                   columns=['a', 'd'],
                   index=[2, 3, 4])
df3
```

Out[14]:

	a	d
2	20	21
3	22	23
4	24	25

In[15]:
```
# конкатерируем их по оси столбцов. Происходит выравнивание по меткам строк,
# осуществляется заполнение значений столбцов df1, а затем
# столбцов df3, получаем дублирующиеся столбцы
pd.concat([df1, df3], axis=1)
```

```
Out[15]:
     a    b    c     a     d
0  0.0  1.0  2.0   NaN   NaN
1  3.0  4.0  5.0   NaN   NaN
2  6.0  7.0  8.0  20.0  21.0
3  NaN  NaN  NaN  22.0  23.0
4  NaN  NaN  NaN  24.0  25.0
```

Поскольку выравнивание осуществлялось по меткам индекса строк, мы получаем дублирующиеся столбцы. Значения для меток строк `0` и `1` в столбцах `a` и `d`, соответствующих датафрейму `df3`, заполняются пропусками, поскольку эти метки есть только в датафрейме `df1`. Значения для меток строк `3` и `4` в столбцах `a`, `b` и `c`, соответствующих датафрейму `df1`, заполняются пропусками, поскольку эти метки есть только в датафрейме `df3`.

Определение типа соединения

По умолчанию конкатенация выполняет операцию **внешнего соединения (outer join)** индексных меток по оси, противоположной оси конкатенации (индексу строк)[6]. Итоговый набор меток представляет собой результат объединения этих меток.

Тип соединения можно изменить на **внутреннее соединение (inner join)**, указав `join='inner'` в качестве параметра. Внутреннее соединение вместо объединения выполняет логическую операцию пересечения индексных меток. Следующий программный код демонстрирует операцию пересечения и приводит к одной строке, потому что метка `2` – это единственная метка индекса строк, которая присутствует и в датафрейме `df1` и в датафрейме `df3`:

```
In[16]:
# выполняем внутреннее соединение вместо внешнего
# результат представлен в виде одной строки
pd.concat([df1, df3], axis=1, join='inner')
```

```
Out[16]:
   a  b  c   a   d
2  6  7  8  20  21
```

Кроме того, можно разметить группы данных вдоль столбцов, применив параметр `keys` при выполнении конкатенации по оси столбцов (`axis=1`):

```
In[17]:
# добавляем ключи к столбцам
df = pd.concat([df1, df2],
               axis=1,
               keys=['df1', 'df2'])
df
```

```
Out[17]:
  df1       df2
    a  b  c   a   c   d
```

---

[6] Проще говоря, по умолчанию мы выполняем конкатенацию вдоль оси строк (`axis=0`), осуществляя внешнее соединение индексных меток по оси столбцов. Если выполняем конкатенацию вдоль оси столбцов (`axis=1`), осуществляем внешнее соединение индексных меток по оси строк. – *Прим. пер.*

```
0   0  1  2   9  10  11
1   3  4  5  12  13  14
2   6  7  8  15  16  17
```

Отобрать различные группы данных можно с помощью свойства `.loc` и создания срезов:

In[18]:
```
# извлекаем данные из датафрейма
# с помощью ключа 'df2'
df.loc[:, 'df2']
```

Out[18]:
```
    a   c   d
0   9  10  11
1  12  13  14
2  15  16  17
```

Присоединение вместо конкатенации

Кроме того, можно воспользоваться методом `.append()` объекта `DataFrame` (и `Series`), который объединяет два указанных объекта `DataFrame` по меткам индекса строк. Помним, что ось столбцов, не участвующая в конкатенации, не выровнена (датафреймы имеют неидентичный набор столбцов), явно задаем то или иное значение параметра `sort`, чтобы избежать предупреждения.

In[19]:
```
# метод .append() выполняет конкатенацию по оси острок (axis=0),
# в результате получаем дублирующиеся индексные метки строк
df1.append(df2, sort=True)
```

Out[19]:
```
    a    b   c     d
0   0  1.0   2   NaN
1   3  4.0   5   NaN
2   6  7.0   8   NaN
0   9  NaN  10  11.0
1  12  NaN  13  14.0
2  15  NaN  16  17.0
```

Как и при конкатенации по оси строк (`axis=0`), при выполнении присоединения индексные метки строк копируются без учета того, что произойдет их дублирование, а метки столбцов объединяются таким способом, который не гарантирует отсутствия в итоговом датафрейме столбцов с дублирующимися именами.

Игнорирование меток индекса

Если вы хотите избавиться от дублирования меток в итоговом индексе и при этом сохранить все строки, вы можете воспользоваться параметром `ignore_index=True`. Это по сути возвращает тот же результат, за исключением того, что теперь наш индекс имеет тип `Int64Index`:

```
In[20]:
# избавляемся от дублирования меток в итоговом индексе,
# игнорируя индексные метки в датафреймах-источниках
df1.append(df2, ignore_index=True, sort=True)
```

```
Out[20]:
    a    b   c     d
0   0  1.0   2   NaN
1   3  4.0   5   NaN
2   6  7.0   8   NaN
3   9  NaN  10  11.0
4  12  NaN  13  14.0
5  15  NaN  16  17.0
```

Эта операция также применяется при выполнении конкатенации.

Слияние и соединение данных

Библиотека pandas позволяет выполнить слияние объектов с помощью операций, аналогичных операциям соединения для баз данных, используя функцию `pd.merge()` и метод `.merge()` объекта `DataFrame`. Процедура слияния объединяет данные двух объектов путем поиска совпадающих значений в одном или нескольких индексах столбцов или строк. Затем, применив семантику соединения к этим значениям, она возвращает новый объект – комбинацию данных обоих объектов.
Слияние полезно, поскольку оно позволяет нам создавать отдельный объект `DataFrame` для каждого типа данных (одно из правил, позволяющее получить аккуратные данные), а также связывать данные в разных объектах `DataFrame` с помощью значений, присутствующих в обоих наборах.

Слияние данных, расположенных в нескольких объектах

Практический пример использования слияния – это поиск имен и адресов клиентов в заказах. Чтобы продемонстрировать выполнение слияния в библиотеке pandas, мы создадим два следующих объекта `DataFrame`. Первый датафрейм содержит адрес клиента (`Address`), идентификационный номер клиента (`CustomerID`), имя клиента (`Name`), а второй датафрейм содержит идентификационный номер клиента (`CustomerID`) и дату заказа (`OrderDate`). Эти датафреймы будут связаны друг с другом с помощью общего столбца `CustomerID`.

```
In[21]:
# это наши клиенты
customers = {'CustomerID': [10, 11],
             'Name': ['Mike', 'Marcia'],
             'Address': ['Address for Mike',
                         'Address for Marcia']}
customers = pd.DataFrame(customers)
customers
```

Out[21]:

```
              Address  CustomerID    Name
0    Address for Mike          10    Mike
1  Address for Marcia          11  Marcia
```

In[22]:

```
# это наши заказы, сделанные клиентами,
# они связаны с клиентами с помощью столбца CustomerID
orders = {'CustomerID': [10, 11, 10],
          'OrderDate': [date(2014, 12, 1),
                        date(2014, 12, 1),
                        date(2014, 12, 1)]}
orders = pd.DataFrame(orders)
orders
```

Out[22]:

```
   CustomerID   OrderDate
0          10  2014-12-01
1          11  2014-12-01
2          10  2014-12-01
```

Теперь предположим, что нам нужно доставить заказы клиентам. Мы должны выполнить слияние датафреймов `orders` и `customers`, чтобы определить адрес доставки для каждого заказа. Это можно легко выполнить с помощью следующей инструкции:

In[23]:

```
# выполняем слияние датафреймов customers и orders так, чтобы
# мы могли отправить товары
customers.merge(orders)
```

Out[23]:

```
              Address  CustomerID    Name   OrderDate
0    Address for Mike          10    Mike  2014-12-01
1    Address for Mike          10    Mike  2014-12-01
2  Address for Marcia          11  Marcia  2014-12-01
```

Библиотека pandas совершила для нас какое-то волшебство, выполнив слияние данных с помощью такого простого кода. Она поняла, что у наших датафреймов `customers` и `orders` есть общий столбец `CustomerID` и с этими знаниями. Затем она воспользовалась общими значениями, найденными в столбце `CustomerID` обоих датафреймов, чтобы связать данные друг с другом и сформировать итоговые данные на основе семантики внутреннего соединения.

Если говорить подробно, библиотека pandas делает следующее:

1. В датафреймах `customers` и `orders` она определяет столбцы с общими метками. Эти столбцы рассматриваются в качестве ключей для выполнения соединения.
2. Она создает новый объект `DataFrame`, столбцы которого – это метки на основе ключей, определенных на шаге 1, после них следуют все остальные метки обоих объектов, не являющиеся ключами.
3. Она сопоставляет значения в столбцах-ключах обоих объектов `DataFrame`.
4. В итоговом датафрейме она создает строку для каждого набора совпавших меток.
5. Затем она копирует данные из совпавших строк каждого объекта-источника в соответствующие строки и столбцы итогового датафрейма.
6. Итоговому датафрейму она присваивает новый тип индекса `Int64Index`.

Операция соединения может использовать значения нескольких столбцов. Чтобы продемонстрировать это, следующий программный код

создает два объекта DataFrame и выполняет слияние, используя значения в столбцах key1 и key2 обоих объектов:

```
In[24]:
# создаем данные, которые будем использовать в качестве примеров
# в оставшейся части этого раздела
left_data = {'key1': ['a', 'b', 'c'],
             'key2': ['x', 'y', 'z'],
            'lval1': [ 0, 1, 2]}
right_data = {'key1': ['a', 'b', 'c'],
              'key2': ['x', 'a', 'z'],
             'rval1': [ 6, 7, 8 ]}
left = pd.DataFrame(left_data, index=[0, 1, 2])
right = pd.DataFrame(right_data, index=[1, 2, 3])
left

Out[24]:
  key1 key2  lval1
0    a    x      0
1    b    y      1
2    c    z      2

In[25]:
right

Out[25]:
  key1 key2  rval1
1    a    x      6
2    b    a      7
3    c    z      8

In[26]:
# демонстрируем слияние, не указывая столбцы, по которым
# нужно выполнить слияние
# этот программный код неявно выполняет слияние
# по всем общим столбцам
left.merge(right)

Out[26]:
  key1 key2  lval1  rval1
0    a    x      0      6
1    c    z      2      8
```

Процедура слияния определяет, что столбцы key1 и key2 являются общими для обоих объектов DataFrame. Соответствующие кортежи значений в обоих объектах DataFrame для этих столбцов выглядят как (a, x) и (c, z), поэтому в результате получаем две строки значений.
Чтобы явно задать столбец, используемый для связывания объектов, можно воспользоваться параметром on. Следующий программный код выполняет слияние, используя только значения столбца key1 обоих объектов DataFrame:

```
In[27]:
# демонстрируем слияние, явно задав столбец,
# по значениям которого нужно связать
# объекты DataFrame
left.merge(right, on='key1')
```

Out[27]:

```
  key1 key2_x  lval1 key2_y  rval1
0    a      x      0      x      6
1    b      y      1      a      7
2    c      z      2      z      8
```

Если сравнивать этот результат с предыдущим примером, у нас теперь три строки, поскольку в обоих объектах столбец `key` имеет значения `a`, `b` и `c`.
Кроме того, параметру `on` можно присвоить список имен столбцов. Следующий программный возвращает нас к использованию столбцов `key1` и `key2`, что приведет к такому же результату, что и в **`Out[26]`**, когда эти два столбца были неявно определены библиотекой pandas:

In[28]:

```
# явно выполняем слияние с помощью двух столбцов
left.merge(right, on=['key1', 'key2'])
```

Out[28]:

```
  key1 key2  lval1  rval1
0    a    x      0      6
1    c    z      2      8
```

Столбцы, указываемые с помощью параметра `on`, должны присутствовать в обоих объектах `DataFrame`. Если вы хотите выполнить слияние на основе столбцов с разными именами в каждом объекте, вы можете использовать параметры `left_on` и `right_on`, передав имя или имена столбцов каждому соответствующему параметру.
Чтобы выполнить слияние с помощью индексных меток строк обоих объектов `DataFrame`, можно воспользоваться параметрами `left_index=True` и `right_index=True` (нужно задать оба параметра):

In[29]:

```
# соединяем индексы строк обеих матриц
pd.merge(left, right, left_index=True, right_index=True)
```

Out[29]:

```
  key1_x key2_x  lval1 key1_y key2_y  rval1
1      b      y      1      a      x      6
2      c      z      2      b      a      7
```

Библиотека pandas определила, что общими индексными метками являются метки `1` и `2`, поэтому итоговый объект `DataFrame` имеет две строки со значениями, соответствующими индексным меткам `1` и `2`. Затем библиотека pandas создает в итоговом датафрейме столбец для каждого столбца датафрейма `left` и каждого столбца датафрейма `right`, а затем копирует значения.
Поскольку в обоих объектах `DataFrame` присутствовал столбец с одинаковым именем `key`, к именам столбцов в итоговом датафрейме добавляются суффиксы `_x` и `_y`, чтобы идентифицировать датафрейм-источник, к которому относится данный столбец. Суффикс `_x` указывает, что столбец принадлежит датафрейму `left`, а суффикс `_y` указывает, что столбец принадлежит датафрейму `right`. Вы можете задать эти

суффиксы с помощью параметра `suffixes` и передать последовательность из двух элементов.

Настройка семантики соединения при выполнении слияния

Тип соединения, выполняемый функцией `pd.merge()` по умолчанию – **внутреннее соединение (inner join)**. Чтобы использовать другой тип соединения, укажите его с помощью параметр `how` функции `pd.merge()` (или метода `.merge()`). Допустимыми параметрами являются:

- `inner`: выполняет пересечение ключей из обоих объектов `DataFrame`
- `outer`: выполняет объединение ключей из обоих объектов `DataFrame`
- `left`: использует только ключи из левого объекта `DataFrame`
- `right`: использует только ключи из правого объекта `DataFrame`

Мы уже видели, что внутреннее соединение является значением по умолчанию и возвращает слияние данных из обоих объектов `DataFrame` только когда значения совпадают.

Внешнее соединение, напротив, возвращает все строки, попавшие во внутреннее соединение (соединенные строки датафреймов `left` и `right`, у которых совпали значения в общих столбцах `key1` и `key2`) плюс все строки из датафрейма `left`, не попавшие во внутреннее соединение (для которых не нашлось пары в датафрейме `right`) плюс все строки из датафрейма `right`, не попавшие во внутреннее соединение (для которых не нашлось пары в датафрейме `left`). Отсутствующие значения заменяются значениями `NaN`. Следующий программный код демонстрирует внешнее соединение:

```
In[30]:
# внешнее соединение возвращает все строки из внутреннего соединения,
# а также строки датафреймов left и right,
# не попавшие во внутреннее соединение
# отсутствующие элементы заполняются значениями NaN
left.merge(right, how='outer')

Out[30]:
key1 key2  lval1  rval1
0    a     x     0.0    6.0
1    b     y     1.0    NaN
2    c     z     2.0    8.0
3    b     a     NaN    7.0
```

Левое соединение возвращает все строки, попавшие во внутреннее соединение (соединенные строки датафреймов `left` и `right`, у которых совпали значения в общих столбцах `key1` и `key2`) плюс все строки из датафрейма `left`, не попавшие во внутреннее соединение (для которых не нашлось пары в датафрейме `right`).

In[31]:
```
# левое соединение возвращает все строки из внутреннего соединения,
# а также строки датафрейма left,
# не попавшие во внутреннее соединение
# отсутствующие элементы заполняются значениями NaN
# итоговый датафрейм содержит общие строки датафреймов
# left и right с метками 0 и 2 (строки датафреймов left и right,
# у которых совпали значения в общих столбцах key1 и key2)
# а также уникальную строку датафрейма left с меткой 1
# уникальная строка датафрейма left в итоговом датафрейме
# в столбце rval1 заполняется значением NaN, потому что
# в датафрейме left этот столбец отсутствовал
left.merge(right, how='left')
```

Out[31]:
```
  key1 key2  lval1  rval1
0    a    x      0    6.0
1    b    y      1    NaN
2    c    z      2    8.0
```

Правое соединение возвращает все строки, попавшие во внутреннее соединение (соединенные строки датафреймов `left` и `right`, у которых совпали значения в общих столбцах `key1` и `key2`) плюс все строки из датафрейма `right`, не попавшие во внутреннее соединение (для которых не нашлось пары в датафрейме `left`).

In[32]:
```
# правое соединение возвращает все строки из внутреннего соединения,
# а также строки датафрейма right,
# не попавшие во внутреннее соединение
# отсутствующие элементы заполняются значениями NaN
# итоговый датафрейм содержит общие строки датафреймов
# left и right с метками 0 и 1 (строки датафреймов left и right,
# у которых совпали значения в общих столбцах key1 и key2)
# а также уникальную строку датафрейма right с меткой 2
# уникальная строка датафрейма right в итоговом датафрейме
# в столбце lval1 заполняется значением NaN, потому что
# в датафрейме right этот столбец отсутствовал
left.merge(right, how='right')
```

Out[32]:
```
  key1 key2  lval1  rval1
0    a    x    0.0      6
1    c    z    2.0      8
2    b    a    NaN      7
```

Кроме того, библиотека pandas предлагает метод `.join()`, который можно использовать для выполнения соединения с помощью индексных меток двух объектов `DataFrame` (вместо значений столбцов). Обратите внимание, что если столбцы в обоих объектах `DataFrame` не имеют уникальных имен, вы должны указать суффиксы с помощью параметров `lsuffix` и `rsuffix` (в отличие от слияния автоматический суффикс при выполнении этой операции не присваивается). Следующий программный код демонстрирует соединение и использование суффиксов:

In[33]:
```
# соединяем left с right (метод по умолчанию - outer)
# и поскольку эти объекты имеют дублирующиеся имена столбцов
# мы задаем параметры lsuffix и rsuffix
left.join(right, lsuffix='_left', rsuffix='_right')
```

Out[33]:
```
  key1_left key2_left  lval1 key1_right key2_right  rval1
0         a         x      0        NaN        NaN    NaN
1         b         y      1          a          x    6.0
2         c         z      2          b          a    7.0
```

Тип соединения, используемый по умолчанию – внешнее соединение. Обратите внимание, что это отличается от стандартного метода `.merge()`, в котором по умолчанию применяется внутреннее соединение. Чтобы выполнить внутреннее соединение, укажите `how='inner'`, как показано в следующем примере:

In[34]:
```
# соединяем left с right с помощью внутреннего соединения
left.join(right, lsuffix='_left', rsuffix='_right', how='inner')
```

Out[34]:
```
  key1_left key2_left  lval1 key1_right key2_right  rval1
1         b         y      1          a          x      6
2         c         z      2          b          a      7
```

Обратите внимание, что этот вывод примерно эквивалентен **Out[29]**, за исключением того, что в выводе у нас столбцы с немного разными именами.
Кроме того, можно выполнить правое и левое соединение, но они приведут к результатам, аналогичным предыдущим, поэтому будут опущены для краткости.

Поворот данных для преобразования значений в индексы и наоборот

Данные часто хранятся в состыкованном формате («в столбик»), который еще называют форматом записи. Он используется в базах данных, CSV-файлах и таблицах Excel. В состыкованном формате данные часто не нормированы и имеют повторяющиеся значения в нескольких столбцах или значения, которые логически должны относится к другими таблицам (нарушая концепцию аккуратных данных).
В качестве примера возьмем поток данных, регистрируемый акселерометром.

In[35]:
```
# считываем данные акселерометра
sensor_readings = pd.read_csv("Data/accel.csv")
sensor_readings
```

Out[35]:

```
    interval axis  reading
0          0    X      0.0
1          0    Y      0.5
2          0    Z      1.0
3          1    X      0.1
4          1    Y      0.4
..       ...  ...      ...
7          2    Y      0.3
8          2    Z      0.8
9          3    X      0.3
10         3    Y      0.2
11         3    Z      0.7

[12 rows x 3 columns]
```

Проблема такого представления данных заключается в том, что довольно трудно выделить показания, относящиеся к конкретной оси. Это можно сделать с помощью логического отбора:

In[36]:

```
# извлекаем показания по оси X
sensor_readings[sensor_readings['axis'] == 'X']
```

Out[36]:

```
   interval axis  reading
0         0    X      0.0
3         1    X      0.1
6         2    X      0.2
9         3    X      0.3
```

Проблема возникает, когда вам нужно узнать значения по всем осям в данный интервал времени, а не только по оси `X`. Чтобы получить значения по всем осям, можно выполнить отбор каждого значения оси, но это будет громоздкий код.
Более оптимальным станет такое представление данных, в котором столбцы будут представлять уникальные значения переменной `axis`. Чтобы преобразовать данные в этот формат, используйте метод `.pivot()` объекта `DataFrame`:

In[37]:

```
# поворачиваем данные. Интервалы становятся индексом, столбцы -
# это оси, а показания - значения столбцов
sensor_readings.pivot(index='interval',
                      columns='axis',
                      values='reading')
```

Out[37]:

```
axis        X    Y    Z
interval
0         0.0  0.5  1.0
1         0.1  0.4  0.9
2         0.2  0.3  0.8
3         0.3  0.2  0.7
```

Состыковка и расстыковка данных

Методы `.stack()` и `.unstack()` схожи с методом `.pivot()`. Процесс состыковки поворачивает уровень меток столбцов, превращая его в индекс строк. Расстыковка выполняет обратное действие, то есть поворачивает уровень индекса строк, превращая его в индекс столбцов.

Одно из различий между состыковкой/расстыковкой и поворотом заключается в том, что в отличии от поворота операции состыковки и расстыковки могут повернуть конкретные уровни иерархического индекса. Кроме того, если поворот сохраняет одинаковое количество уровней индекса, состыковка и расстыковка всегда увеличивают количество уровней по индексу одной оси (количество столбцов для расстыковки и количество строк для состыковки) и уменьшают количество уровней по другой оси.

Состыковка с помощью неиерархических индексов

Чтобы продемонстрировать состыковку, мы рассмотрим несколько примеров, используя объект `DataFrame` с неиерархическими индексами. Мы начнем с того, что создадим следующий объект `DataFrame`:

```
In[38]:
# создаем простой датафрейм с одним столбцом
df = pd.DataFrame({'a': [1, 2]}, index={'one', 'two'})
df

Out[38]:
     a
two  1
one  2
```

Состыковка помещает уровень индекса столбцов в новый уровень индекса строк. Поскольку наш объект `DataFrame` имеет только один уровень, происходит сворачивание объекта `DataFrame` в объект `Series` с иерархическим индексом строк:

```
In[39]:
# помещаем столбец в еще один уровень индекса строк
# результатом становится объект Series, в которой
# значения можно просмотреть с помощью мультииндекса
stacked1 = df.stack()
stacked1

Out[39]:
two  a    1
one  a    2
dtype: int64
```

Чтобы посмотреть значения, нам нужно передать кортеж в индексатор объекта `Series`, который выполняет поиск с помощью индекса:

```
In[40]:
# ищем значение для 'one'/'a', передав кортеж в индексатор
stacked1[('one', 'a')]

Out[40]:
2
```

Если объект `DataFrame` содержит несколько столбцов, то все столбцы помещаются в один и тот же дополнительный уровень нового объекта `Series`:

```
In[41]:
# создаем датафрейм с двумя столбцами
df = pd.DataFrame({'a': [1, 2],
                   'b': [3, 4]},
                  index={'one', 'two'})
df
```

```
Out[41]:
     a  b
two  1  3
one  2  4

In[42]:
# помещаем оба столбца в отдельный уровень индекса
stacked2 = df.stack()
stacked2

Out[42]:
two  a    1
     b    3
one  a    2
     b    4
dtype: int64
```

Значения в наших прежних столбцах можно посмотреть, вновь передав кортеж в индексатор:

```
In[43]:
# ищем значение с помощью индекса 'one' / 'b'
stacked2[('one', 'b')]

Out[43]:
4
```

Расстыковка выполняет противоположную операцию, помещая уровень индекса строк в уровень оси столбцов. Мы рассмотрим этот процесс в следующем разделе, так как при выполнении расстыковки, как правило, предполагается, что индекс, который расстыковывается, является иерархическим.

Расстыковка с помощью иерархических индексов

Чтобы продемонстрировать расстыковку с помощью иерархических индексов, мы обратимся к данным акселерометра, которые мы уже рассматривали ранее в этой главе. Однако теперь мы создадим две копии этих данных, по одной для каждого пользователя, и в каждую копию добавим дополнительный столбец. Следующий программный код создает эти данные:

In[44]:
```
# создаем две копии данных акселерометра,
# по одной для каждого пользователя
user1 = sensor_readings.copy()
user2 = sensor_readings.copy()
# добавляем столбец who в каждую копию
user1['who'] = 'Mike'
user2['who'] = 'Mikael'
# давайте отмасштабируем данные user2
user2['reading'] *= 100
# и организуем данные так, чтобы получить иерархический
# индекс строк
multi_user_sensor_data = pd.concat([user1, user2]) \
            .set_index(['who', 'interval', 'axis'])
multi_user_sensor_data
```

Out[44]:
```
                      reading
who    interval axis
Mike   0        X         0.0
                Y         0.5
                Z         1.0
       1        X         0.1
                Y         0.4
...                        ...
Mikael 2        Y        30.0
                Z        80.0
       3        X        30.0
                Y        20.0
                Z        70.0

[24 rows x 1 columns]
```

Получив такое представление данных мы можем посмотреть все показания по конкретному человеку, используя только индекс:

In[45]:
```
# извлекаем показания, относящиеся к пользователю Mike,
# с помощью индекса
multi_user_sensor_data.loc['Mike']
```

Out[45]:
```
                reading
interval axis
0        X         0.0
         Y         0.5
         Z         1.0
1        X         0.1
         Y         0.4
...                ...
2        Y         0.3
         Z         0.8
3        X         0.3
         Y         0.2
         Z         0.7

[12 rows x 1 columns]
```

Кроме того, мы можем получить все показания по всем осям и по всем пользователям в интервале 1 с помощью метода `.xs()`:

In[46]:
```
# извлекаем все показания по всем осям
# и по всем пользователям в интервале 1
multi_user_sensor_data.xs(1, level='interval')
```

Out[46]:

```
             reading
who    axis
Mike   X         0.1
       Y         0.4
       Z         0.9
Mikael X        10.0
       Y        40.0
       Z        90.0
```

Расстыковка помещает самый внутренний уровень индекса строк в новый уровень индекса столбцов, в результате получаем столбцы с типом индекса `MultiIndex`. Следующий программный код показывает, как в результате расстыковки самый внутренний уровень индекса строк (уровень `axis` индекса строк) стал уровнем индекса столбцов:

In[47]:

```
# выполняем расстыковку, в результате самый внутренний
# уровень индекса строк (уровень axis)
# стал уровнем индекса столбцов
multi_user_sensor_data.unstack()
```

Out[47]:

```
                reading
axis                  X     Y      Z
who    interval
Mikael 0            0.0  50.0  100.0
       1           10.0  40.0   90.0
       2           20.0  30.0   80.0
       3           30.0  20.0   70.0
Mike   0            0.0   0.5    1.0
       1            0.1   0.4    0.9
       2            0.2   0.3    0.8
       3            0.3   0.2    0.7
```

Чтобы выполнить расстыковку другого уровня, используйте параметр `level`. Следующий программный код выполняет расстыковку первого уровня (`level=0`):

In[48]:

```
# выполняем расстыковку по уровню 0
multi_user_sensor_data.unstack(level=0)
```

Out[48]:

```
             reading
who           Mikael Mike
interval axis
0        X       0.0  0.0
         Y      50.0  0.5
         Z     100.0  1.0
1        X      10.0  0.1
         Y      40.0  0.4
...              ...  ...
2        Y      30.0  0.3
         Z      80.0  0.8
3        X      30.0  0.3
         Y      20.0  0.2
         Z      70.0  0.7

[12 rows x 2 columns]
```

Для выполнения одновременной расстыковки нескольких уровней нужно передать в метод `.unstack()` список уровней. Кроме того, если уровни имеют имена, их можно перечислить с помощью имен, а не

позиций. Следующий программный код выполняет расстыковку уровней `who` и `axis` по именам:

In[49]:
```
# выполняем расстыковку уровней
# who и axis
unstacked = multi_user_sensor_data.unstack(['who', 'axis'])
unstacked
```

Out[49]:
```
        reading
who        Mike           Mikael
axis          X    Y    Z      X     Y      Z
interval
0           0.0  0.5  1.0    0.0  50.0  100.0
1           0.1  0.4  0.9   10.0  40.0   90.0
2           0.2  0.3  0.8   20.0  30.0   80.0
3           0.3  0.2  0.7   30.0  20.0   70.0
```

Для дотошности заново состыкуем эти данные. Следующий программный код выполнит состыковку уровня `who`, перенеся его из индекса столбцов обратно в индекс строк:

In[50]:
```
# и, конечно, мы можем выполнить состыковку уровней,
# которые расстыковали
# выполняем состыковку уровня who
unstacked.stack(level='who')
```

Out[50]:
```
                reading
axis                  X     Y      Z
interval who
0        Mikael     0.0  50.0  100.0
         Mike       0.0   0.5    1.0
1        Mikael    10.0  40.0   90.0
         Mike       0.1   0.4    0.9
2        Mikael    20.0  30.0   80.0
         Mike       0.2   0.3    0.8
3        Mikael    30.0  20.0   70.0
         Mike       0.3   0.2    0.7
```

Есть пара вещей, о которых стоит упомянуть, глядя на полученный результат. Во-первых, состыковка и расстыковка всегда помещают уровни в самые внутренние уровни другого индекса. Обратите внимание, что уровень `who` теперь стал самым внутренним уровнем индекса строк, а раньше он был первым уровнем. Это отразится на программном коде, который мы будем использовать для просмотра значений через этот индекс, поскольку индекс переместится на другой уровень. Если вы хотите поместить уровень индекса на другую глубину (например, при выполнении состыковки сделать уровень `who` не самым внутренним, а самым первым уровнем индекса строк), для такой организации индексов вместо состыковки и расстыковки нужно воспользоваться другими способами.

Во-вторых, при перемещении данных, осуществляемом в ходе состыковки и расстыковки (а также повороте), не происходит потери информации. При выполнении этих операций просто меняется способ организации и просмотра данных.

Расплавление данных для преобразования «широкого» формата в «длинный» и наоборот

Расплавление – это тип реорганизации данных, который часто называют преобразованием объекта `DataFrame` из **«широкого» формата (wide format)** в **«длинный» формат (long format)**. Этот формат является общепринятым при проведении различных видов статистического анализа, и данные, которые вы считываете, могут быть представлены уже в расплавленном виде. Либо вам, возможно, потребуется преобразовать данные в этот формат.

С технической точки зрения расплавление – это процесс изменения формы объекта `DataFrame`, в результате получаем формат, в котором два или более столбцов, названные `variable` и `value`, создаются путем сворачивания меток столбцов исходного датафрейма в столбец `variable`, а затем происходит перемещение значений из этих столбцов в соответствующую позицию столбца `value`. Все остальные столбцы превращаются в столбцы-идентификаторы, которые помогают описать данные.

Концепцию расплавления часто лучше всего понять, взяв простой пример. В этом примере мы создадим объект `DataFrame`, который содержит две переменные – столбцы `Height` и `Weight`, кроме того, есть еще одна дополнительная переменная – столбец `Name`, представляющий конкретного человека:

```
In[51]:
# продемонстрируем расплавление
# с помощью этого датафрейма
data = pd.DataFrame({'Name' : ['Mike', 'Mikael'],
                     'Height' : [6.1, 6.0],
                     'Weight' : [220, 185]})
data
```

```
Out[51]:
   Height    Name  Weight
0     6.1    Mike     220
1     6.0  Mikael     185
```

Следующий программный код расплавляет этот датафрейм, используя столбец `Name` в качестве столбца-идентификатора, а столбцы `Height` и `Weight` в качестве переменных. Столбец `Name` остается неизменным, при этом столбцы `Height` и `Weight` сворачиваются в столбец `variable`. Затем значения этих двух столбцов становятся значениями столбца `value` и соответствуют комбинациям значений переменных `Name` и `variable`.

In[52]:
```
# расплавляем датафрейм, используем Name
# в качестве идентификатора, а столбцы
# Height and Weight в качестве переменных
pd.melt(data,
        id_vars=['Name'],
        value_vars=['Height', 'Weight'])
```

Out[52]:
```
     Name variable  value
0    Mike   Height    6.1
1  Mikael   Height    6.0
2    Mike   Weight  220.0
3  Mikael   Weight  185.0
```

Данные теперь реструктурированы, поэтому легко извлечь значение для любой комбинации переменных `variable` и `Name`. Кроме того, в таком формате представления проще добавить новую переменную и наблюдение, поскольку можно просто добавить данные в виде новой строки и для этого не требуется менять структуру датафрейма, добавляя новый столбец.

Преимущества использования состыкованных данных

Наконец, мы выясним, зачем нужно состыковывать данные. Можно продемонстрировать, что поиск значений в состыкованных данных более эффективен, если сравнивать его с поиском значений по отдельному уровню индекса, а затем по столбцу и даже поиском на основе свойства `.iloc`, который определяет строку и столбец по месторасположению. Следующий программный код демонстрирует это:

In[53]:
```
# поиск значений в состыкованных данных может быть
# намного быстрее поиска в обычных данных

# время выполнения различных методов
import timeit
t = timeit.Timer("stacked1[('one', 'a')]",
                 "from __main__ import stacked1, df")
r1 = timeit.timeit(lambda: stacked1.loc[('one', 'a')],
                   number=10000)
r2 = timeit.timeit(lambda: df.loc['one']['a'],
                   number=10000)
r3 = timeit.timeit(lambda: df.iloc[1, 0],
                   number=10000)

# и вот наши результаты...  Да, это самый быстрый метод из всех трех
r1, r2, r3
```

Out[53]:
```
(24.243854494566037, 1.3032710876761477, 0.11965651392785048)
```

Если регулярно требуется извлекать большое количество скалярных значений из датафрейма, можно кардинальным образом повысить производительность приложения.

Выводы

В этой главе мы рассмотрели несколько методов объединения и изменения формы данных. Мы начали главу с рассмотрения различных способов объединения данных, расположенных в нескольких объектах библиотеки pandas. Затем мы изучили способы объединения нескольких объектов `DataFrame` как по оси строк, так и по оси столбцов. Затем мы посмотрели, как можно использовать библиотеку pandas для соединения и слияния данных на основе значений в нескольких объектах `DataFrame`. Затем мы разобрали способы изменения формы данных в объекте `DataFrame` с помощью поворота, состыковки/расстыковки и расплавления. Благодаря этому мы увидели, как каждая операция предлагает несколько вариантов модификации данных путем изменения формы индексов и перемещения значений в индексы и обратно. Это позволило нам выяснить, как преобразовать данные в такие форматы, которые будут эффективны для поиска в других формах баз данных и могут быть более удобными для разработчика данных.

В следующей главе мы узнаем о группировке и агрегировании данных в этих группах, что позволит нам получит результаты на основе одинаковых значений в данных.

ГЛАВА 12 АГРЕГИРОВАНИЕ ДАННЫХ

Агрегирование данных – это процесс группировки данных. Затем для каждой из групп выполняется анализ, чтобы вывести одну или несколько итоговых статистик по каждой категории. Подытоживание является общим термином в том смысле, что *подытоживание* может быть буквально суммированием (например, общее количество проданных единиц) или вычислением статистик, например, среднего значения или стандартного отклонения.

В этой главе будут рассмотрены возможности библиотеки pandas по выполнению агрегации данных. Эти возможности включают в себя схему «разделение–применение–объединение» для выполнения группировки, преобразований и анализа с последующим представлением результатов по каждой группе в итоговом объекте библиотеки pandas. В этом плане мы рассмотрим несколько методов группировки данных, применения функций на уровне групп и возможности включения/отключения фильтрации данных.

В частности, в этой главе мы рассмотрим:

- Обзор схемы «разделение–применение–объединение»
- Группировка по значениям одного столбца
- Просмотр результатов группировки
- Группировка по значениям в нескольких столбцах
- Группировка по уровням индекса
- Применение функций агрегации к сгруппированным данным
- Обзор преобразований данных
- Практические примеры преобразования данных: заполнение средними значениями и z-значениями
- Применение фильтрации для выборочного удаления групп данных
- Дискретизация и биннинг

Настройка библиотеки pandas

Мы начнем рассмотрение примеров в этой главе, задав следующие инструкции для импорта библиотек и настройки вывода:

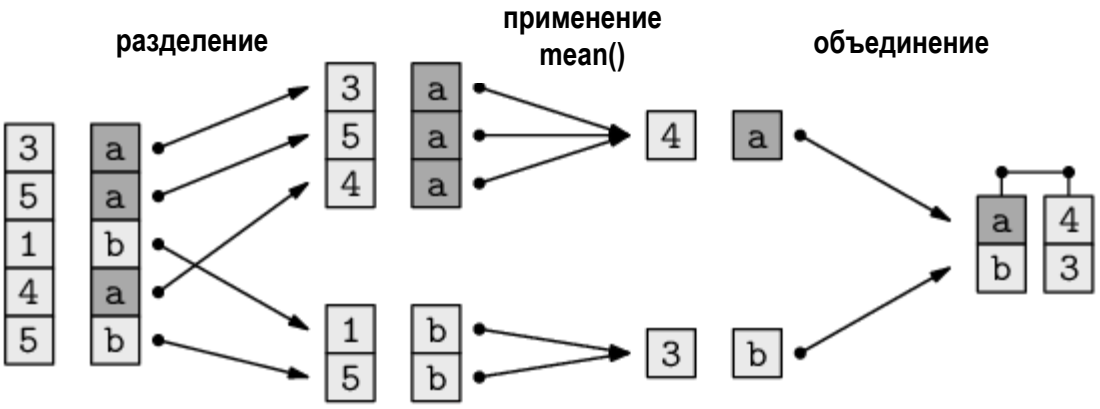
```
In[1]:
# импортируем библиотеки numpy и pandas
import numpy as np
import pandas as pd

# импортируем библиотеку datetime для работы с датами
import datetime
from datetime import datetime, date

# Задаем некоторые опции библиотеки pandas, которые
# настраивают вывод
pd.set_option('display.notebook_repr_html', False)
pd.set_option('display.max_columns', 8)
pd.set_option('display.max_rows', 10)
pd.set_option('display.width', 60)

# импортируем библиотеку matplotlib для построения графиков
import matplotlib.pyplot as plt
%matplotlib inline
```

Обзор схемы «разделение – применение - объединение»

Во многих задачах анализа данных используется схема обработки данных под названием **«разделение–применение–объединение» (split-apply-comb)**. Эта схема включает три этапа анализа данных:

- Набор данных разбивается на более мелкие фрагменты, основываясь на определенных критериях
- Каждый фрагмент обрабатывается независимо
- Затем все результаты объединяются обратно и представляют собой единое целое

На следующей диаграмме показан простой процесс, иллюстрирующий схему «разделение–применение–объединение». Она используется для вычисления среднего значения путем группировки по символьному ключу (a или b):



Данные разбиваются по индексным меткам на две группы (по одной группе для a и b). Вычисляется среднее значение в каждой группе. Полученные значения по каждой группе затем объединяются в один объект, который индексируется с помощью метки, представляющей каждую группу.

В библиотеке pandas разделение выполняется с помощью метода `.groupby()` объекта `Series` или объекта `DataFrame`. В этот метод передается одна или несколько индексных меток и/или имя столбца, в результате происходит группировка данных на основе соответствующих значений.

После разделения данных для каждой группы можно выполнить одну или несколько операций, приведенных ниже:

- **Агрегация:** вычисляет итоговую статистику, например, групповые средства или количество элементов в каждой группе
- **Преобразование:** выполнение вычислений по группам или элементам
- **Фильтрация:** удаление целых групп данных на основе вычислений по группам

Заключительный этап, этап комбинации, выполняется библиотекой pandas автоматически, она собирает результаты, полученные на этапе применения, в единое целое.

Для получения дополнительной информации о схеме «разделение–применение–объединение» *обратитесь к статье в Journal of Statistical Software под названием «The Split-Apply-Combine Strategy for Data Analysis». В ней довольно подробно описывается применение вышеупомянутой схемы и, хотя в примерах используется R, она все же представляет ценность для всех, кто изучает библиотеку pandas. Вы можете скачать эту статью по адресу* http://www.jstatsoft.org/v40/i01/paper.

Данные для примеров

В качестве примеров в этой главе будет использоваться набор данных, представляющий измерения нескольких датчиков. Данные включают показания по осям **X**, **Y** и **Z**, снятые как с акселерометра, так и с датчика ориентации:

In[2]:
```
# загружаем данные датчиков
sensor_data = pd.read_csv("Data/sensors.csv")
sensor_data[:5]
```

Out[2]:
```
   interval sensor axis  reading
0         0  accel    Z      0.0
1         0  accel    Y      0.5
2         0  accel    X      1.0
3         1  accel    Z      0.1
4         1  accel    Y      0.4
```

Разделение данных

Рассмотрение этапа разделения данных внутри объекта pandas будет разбито на несколько этапов. Сначала мы рассмотрим группировку на основе столбцов, а затем разберем свойства создаваемой группировки. Затем мы расскажем, как получить доступ к различным свойствам и результатам группировки, чтобы изучить характеристики созданных групп. Затем мы рассмотрим группировку, использующую индексные метки вместо содержимого столбцов.

Группировка по значениям отдельного столбца

Данные датчиков состоят из трех категориальных переменных (`sensor`, `interval` и `axis`) и одной количественной переменной (`reading`). Их можно сгруппировать с помощью любой категориальной переменной, передав имя переменной в метод `.groupby()`. Следующий программный код группирует данные датчиков по значениям столбца `sensor`:

In[3]:

```
# группировка этих данных по столбцу/переменной sensor
# возвращает объект DataFrameGroupBy
grouped_by_sensor = sensor_data.groupby('sensor')
grouped_by_sensor
```

```
Out[3]:
<pandas.core.groupby.DataFrameGroupBy object at 0x10683e828>
```

В результате применения метода `.groupby()` объекта `DataFrame` получаем подкласс объекта `GroupBy`, либо `DataFrameGroupBy` для датафрейма, либо `SeriesGroupBy` для серии. Этот объект представляет собой промежуточное описание группировки, которую мы в конечном итоге создадим. Этот объект помогает библиотеке pandas сначала проверить группировку перед выполнением. Он помогает выявить ошибки и дает возможность проверить определенные свойства до того, как мы запустим затратный вычислительный процесс.
Этот промежуточный объект имеет много полезных свойств. Свойство `.ngroups` извлекает количество групп, которые будут сформированы в итоге:

```
In[4]:
# получаем информацию о количестве групп,
# которые будут созданы
grouped_by_sensor.ngroups
```

```
Out[4]:
2
```

Свойство `.groups` вернет питоновский словарь, ключи которого будут представлять имена каждой группы (если указано несколько столбцов, это будет кортеж). Значения в словаре – это массив индексных меток, принадлежащих каждой соответствующей группе:

```
In[5]:
# что представляют из себя найденные группы?
grouped_by_sensor.groups
```

```
Out[5]:
{'accel': Int64Index([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11], dtype='int64'),
 'orientation': Int64Index([12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
             23],
            dtype='int64')}
```

Просмотр результатов группировки

Сгруппированную переменную можно рассматривать как набор поименованных групп и ее можно использовать для просмотра содержимого групп. Рассмотрим эти группы с помощью следующей функции:

In[6]:
```
# вспомогательная функция, печатающая содержимое групп
def print_groups (group_object):
    # итерируем по всем группам, печатая название группы
    # и первые пять наблюдений в группе
    for name, group in group_object:
        print (name)
        print (group[:5])
```

Эта функция пробегает по каждой группе, печатает ее название и первые пять строк в ней:

In[7]:
```
# смотрим содержимое созданных групп
print_groups(grouped_by_sensor)
```

Out[7]:
```
accel
   interval sensor axis  reading
0         0  accel    Z      0.0
1         0  accel    Y      0.5
2         0  accel    X      1.0
3         1  accel    Z      0.1
4         1  accel    Y      0.4
orientation
    interval       sensor axis  reading
12         0  orientation    Z      0.0
13         0  orientation    Y      0.1
14         0  orientation    X      0.0
15         1  orientation    Z      0.0
16         1  orientation    Y      0.2
```

Просмотр этих результатов дает нам некоторое представление о том, как библиотека pandas выполнила разделения данных. Для каждого уникального значения столбца `sensor` создается отдельная группа, названием каждой группы становится соответствующее значение столбца `sensor`. Каждая группа содержит объект `DataFrame`, в котором значение столбца `sensor` соответствует названию группы.
Метод `.size()` возвращает сводку о размере каждой группы:

In[8]:
```
# получаем информацию о количестве элементов
# в каждой группе
grouped_by_sensor.size()
```

Out[8]:
```
sensor
accel          12
orientation    12
dtype: int64
```

Метод `.count()` возвращает количество элементов в каждом столбце каждой группы:

In[9]:
```
# получаем информацию о количестве элементов
# в каждом столбце каждой группы
grouped_by_sensor.count()
```

Out[9]:

```
             interval  axis  reading
sensor
accel              12    12       12
orientation        12    12       12
```

С помощью метода `.get_group()` можно извлечь любую конкретную группу. Следующий программный код извлекает группу `accel`:

In[10]:

```
# получаем данные конкретной группы
grouped_by_sensor.get_group('accel')[:5]
```

Out[10]:

```
   interval sensor axis  reading
0         0  accel    Z      0.0
1         0  accel    Y      0.5
2         0  accel    X      1.0
3         1  accel    Z      0.1
4         1  accel    Y      0.4
```

Для вывода указанного количества элементов в каждой группе можно использовать методы `.head()` и `.tail()`. Следующий программный код извлекает первые три строки в каждой группе:

In[11]:

```
# извлекаем первые три строки в каждой группе
grouped_by_sensor.head(3)
```

Out[11]:

```
    interval       sensor axis  reading
0          0        accel    Z      0.0
1          0        accel    Y      0.5
2          0        accel    X      1.0
12         0  orientation    Z      0.0
13         0  orientation    Y      0.1
14         0  orientation    X      0.0
```

Метод `.nth()` возвращает *n*-ную строку в каждой группе. Следующий программный код иллюстрирует его применение для вывода второй строки каждой группы:

In[12]:

```
# извлекаем вторую строку каждой группы
grouped_by_sensor.nth(1)
```

Out[12]:

```
            axis  interval  reading
sensor
accel          Y         0      0.5
orientation    Y         0      0.1
```

Для вывода описательных статистик по каждой группе можно использовать метод `.describe()`:

In[13]:

```
# получаем описательные статистики по каждой группе
grouped_by_sensor.describe()
```

Out[13]:

```
            interval                      ...  reading  \
               count mean       std  min ...      25%
sensor                                    ...
accel           12.0  1.5  1.167748  0.0 ...      0.2
orientation     12.0  1.5  1.167748  0.0 ...      0.0


              50%    75%  max
sensor
accel        0.35  0.725  1.0
orientation  0.10  0.225  0.4

[2 rows x 16 columns]
```

Группы отсортированы по названиям в порядке возрастания. Если вы хотите предотвратить сортировку при выполнении группировки, воспользуйтесь параметром `sort=False`.

Группировка по нескольким столбцам

Группировку еще можно выполнить по нескольким столбцам, передав список имен столбцов. Следующий программный код группирует данные по столбцам `sensor` и `axis`:

In[14]:

```
# группируем по значениям столбцов sensor и axis
mcg = sensor_data.groupby(['sensor', 'axis'])
print_groups(mcg)
```

Out[14]:

```
('accel', 'X')
    interval sensor axis  reading
2          0  accel    X      1.0
5          1  accel    X      0.9
8          2  accel    X      0.8
11         3  accel    X      0.7
('accel', 'Y')
    interval sensor axis  reading
1          0  accel    Y      0.5
4          1  accel    Y      0.4
7          2  accel    Y      0.3
10         3  accel    Y      0.2
('accel', 'Z')
   interval sensor axis  reading
0         0  accel    Z      0.0
3         1  accel    Z      0.1
6         2  accel    Z      0.2
9         3  accel    Z      0.3
('orientation', 'X')
    interval       sensor axis  reading
14         0  orientation    X      0.0
17         1  orientation    X      0.1
20         2  orientation    X      0.2
23         3  orientation    X      0.3
('orientation', 'Y')
    interval       sensor axis  reading
13         0  orientation    Y      0.1
16         1  orientation    Y      0.2
19         2  orientation    Y      0.3
22         3  orientation    Y      0.4
('orientation', 'Z')
    interval       sensor axis  reading
12         0  orientation    Z      0.0
15         1  orientation    Z      0.0
18         2  orientation    Z      0.0
21         3  orientation    Z      0.0
```

Поскольку было указано несколько столбцов, название каждой группы теперь является кортежем, представляющим уникальную комбинацию значений столбцов `sensor` и `axis`.

Группировка по уровням индекса

Группировку можно выполнить, используя вместо столбцов значения индекса. Данные датчиков хорошо подходят для иерархического индекса, который можно использовать для иллюстрации такого способа. Давайте изменим форму этих данных, создав иерархический индекс, у которого уровнями будут столбцы `sensor` и `axis`:

In[15]:
```
# создаем копию данных и заново индексируем ее
mi = sensor_data.copy()
mi = mi.set_index(['sensor', 'axis'])
mi
```

Out[15]:
```
                interval  reading
sensor      axis
accel       Z            0      0.0
            Y            0      0.5
            X            0      1.0
            Z            1      0.1
            Y            1      0.4
...                    ...      ...
orientation Y            2      0.3
            X            2      0.2
            Z            3      0.0
            Y            3      0.4
            X            3      0.3

[24 rows x 2 columns]
```

Теперь можно выполнить группировку по различным уровням иерархического индекса. Следующие программный код сгруппирует данные по уровню индекса `0` (типу датчика):

In[16]:
```
# группируем по первому уровню индекса
print_groups(mi.groupby(level=0))
```

Out[16]:
```
accel
             interval  reading
sensor axis
accel  Z            0      0.0
       Y            0      0.5
       X            0      1.0
       Z            1      0.1
       Y            1      0.4
orientation
                interval  reading
sensor      axis
orientation Z            0      0.0
            Y            0      0.1
            X            0      0.0
            Z            1      0.0
            Y            1      0.2
```

Группировку по нескольким уровням можно выполнить, передав список уровней. Если у каждого уровня `MultiIndex` есть имена, то их можно

использовать вместо целочисленных значений. Следующий программный код демонстрирует группировку по уровням `sensor` и `axis`:

In[17]:
```
# группируем по нескольким уровням индекса
print_groups(mi.groupby(level=['sensor', 'axis']))
```

Out[17]:
```
('accel', 'X')
              interval  reading
sensor axis
accel  X             0      1.0
       X             1      0.9
       X             2      0.8
       X             3      0.7
('accel', 'Y')
              interval  reading
sensor axis
accel  Y             0      0.5
       Y             1      0.4
       Y             2      0.3
       Y             3      0.2
('accel', 'Z')
              interval  reading
sensor axis
accel  Z             0      0.0
       Z             1      0.1
       Z             2      0.2
       Z             3      0.3
('orientation', 'X')
                   interval  reading
sensor      axis
orientation X             0      0.0
            X             1      0.1
            X             2      0.2
            X             3      0.3
('orientation', 'Y')
                   interval  reading
sensor      axis
orientation Y             0      0.1
            Y             1      0.2
            Y             2      0.3
            Y             3      0.4
('orientation', 'Z')
                   interval  reading
sensor      axis
orientation Z             0      0.0
            Z             1      0.0
            Z             2      0.0
            Z             3      0.0
```

Применение агрегирующих функций, преобразований и фильтров

На этапе применения для каждой группы данных можно выполнить три различные операции:

- Применение агрегирующей функции.
- Преобразование
- Фильтрация – исключение всей группы из результатов.

Давайте рассмотрим каждую из этих операций.

Применение агрегирующих функций к группам

Агрегирующие функции можно применить к каждой группе с помощью метода `.aggregate()` (или более короткого `.agg()`) объекта `GroupBy`. Параметр `.agg()` – это ссылка на функцию, которая будет применяться к каждой группе. В случае с объектом `DataFrame` эта функция будет применяться к каждому столбцу данных внутри группы.
Следующий пример демонстрирует вычисление среднего значения для каждого типа датчика и оси:

In[18]:
```
# вычисляем среднее для каждого сенсора/оси
sensor_axis_grouping = mi.groupby(level=['sensor', 'axis'])
sensor_axis_grouping.agg(np.mean)
```

Out[18]:
```
                interval  reading
sensor      axis
accel       X          1.5     0.85
            Y          1.5     0.35
            Z          1.5     0.15
orientation X          1.5     0.15
            Y          1.5     0.25
            Z          1.5     0.00
```

Поскольку `.agg()` применяет метод к каждому столбцу в каждой группе, библиотека pandas еще вычисляет среднее значение интервала (что может не представлять большого практического интереса).
Результат агрегации получит идентично структурированный индекс, что и исходные данные. Для создания числового индекса и переноса уровней исходного индекса в столбцы можно воспользоваться параметром `as_index=False`:

In[19]:
```
# вместо индекса, совпадающего с индексом исходного объекта,
# создаем числовой индекс и переносим уровни исходного
# индекса в столбцы
sensor_data.groupby(['sensor', 'axis'], as_index=False).agg(np.mean)
```

Out[19]:
```
        sensor axis  interval  reading
0        accel    X       1.5     0.85
1        accel    Y       1.5     0.35
2        accel    Z       1.5     0.15
3  orientation    X       1.5     0.15
4  orientation    Y       1.5     0.25
5  orientation    Z       1.5     0.00
```

Многие агрегирующие функции встроены непосредственно в объект `GroupBy` для обеспечения определенной типизации. В частности, вот эти функции (с префиксом `gb.`):

```
gb.agg          gb.last         gb.aggregate    gb.max          gb.apply        gb.mean
gb.boxplot      gb.median       gb.count        gb.min          gb.cummax       gb.name
gb.cummin       gb.ngroups      gb.cumprod      gb.nth          gb.cumsum       gb.ohlc
gb.describe     gb.plot         gb.dtype        gb.prod         gb.fillna       gb.quantile
gb.filter       gb.rank         gb.first        gb.resample     gb.gender       gb.size
gb.get_group    gb.std          gb.groups       gb.sum          gb.head         gb.tail
gb.height       gb.transform    gb.hist         gb.var          gb.indices      gb.weight
```

В качестве иллюстрации следующий программный код тоже вычисляет среднее значение для каждой комбинации типа датчика и оси:

In[20]:
```
# можно просто применить агрегирующую функцию к группе
sensor_axis_grouping.mean()
```

Out[20]:
```
                 interval  reading
sensor      axis
accel       X         1.5     0.85
            Y         1.5     0.35
            Z         1.5     0.15
orientation X         1.5     0.15
            Y         1.5     0.25
            Z         1.5     0.00
```

Кроме того, можно применить несколько агрегирующих функций в рамках одной инструкции, передав список функций.

In[21]:
```
# применяем сразу несколько агрегирующих функций
sensor_axis_grouping.agg([np.sum, np.std])
```

Out[21]:
```
                interval           reading
                     sum       std     sum       std
sensor      axis
accel       X          6  1.290994     3.4  0.129099
            Y          6  1.290994     1.4  0.129099
            Z          6  1.290994     0.6  0.129099
orientation X          6  1.290994     0.6  0.129099
            Y          6  1.290994     1.0  0.129099
            Z          6  1.290994     0.0  0.000000
```

К каждому столбцу можно применить свою функцию, передав в `.agg()` питоновский словарь. Ключ словаря – это имя столбца, к которому нужно применить функцию, а значение словаря – название функции. Следующий программный код демонстрирует этот способ, вычисляя среднее значение для столбца `reading` и длину группы для столбца `interval`:

In[22]:
```
# применяем к каждому столбцу свою функцию
sensor_axis_grouping.agg({'interval' : len,
                          'reading': np.mean})
```

Out[22]:
```
                 interval  reading
sensor      axis
accel       X           4     0.85
            Y           4     0.35
            Z           4     0.15
orientation X           4     0.15
            Y           4     0.25
            Z           4     0.00
```

Еще агрегирование можно применить к определенным столбцам с помощью оператора `[]` объекта `GroupBy`. Следующий программный код вычисляет среднее значение только для столбца `reading`:

In[23]:
```
# вычисляем среднее только для столбца reading
```

```
sensor_axis_grouping['reading'].mean()
```

```
Out[23]:
sensor       axis
accel        X       0.85
             Y       0.35
             Z       0.15
orientation  X       0.15
             Y       0.25
             Z       0.00
Name: reading, dtype: float64
```

Преобразование групп данных

Объект `GroupBy` предлагает метод `.transform()`, который применяет ту или иную функцию ко всем значениям объекта `DataFrame` в каждой группе. Мы разберем общий процесс преобразования, а затем рассмотрим два реальных примера.

Общий процесс преобразования

Метод `.transform()` объекта `GroupBy` применяет функцию к каждому значению объекта `DataFrame` и возвращает другой объект `DataFrame`, который имеет следующие характеристики:

- Он идентично проиндексирован, чтобы выполнить конкатенацию индексов во всех группах
- Количество строк равно количеству строк, просуммированному по всем группам
- Он состоит из несгруппированных столбцов, к которым библиотека pandas успешно применила заданную функцию (некоторые столбцы могут быть исключены)

Чтобы продемонстрировать преобразование в действии, давайте создадим следующий объект `DataFrame`:

```
In[24]:
# создаем объект DataFrame, который
# будем использовать в примерах
transform_data = pd.DataFrame({ 'Label': ['A', 'C', 'B', 'A', 'C'],
                                'Values': [0, 1, 2, 3, 4],
                                'Values2': [5, 6, 7, 8, 9],
                                'Other': ['foo', 'bar', 'baz',
                                          'fiz', 'buz']},
                              index = list('VWXYZ'))
transform_data
```

```
Out[24]:
  Label  Values  Values2 Other
V     A       0        5   foo
W     C       1        6   bar
X     B       2        7   baz
Y     A       3        8   fiz
Z     C       4        9   buz
```

Теперь давайте сгруппируем данные по столбцу `Label`:

```
In[25]:
# сгруппируем по столбцу Label
grouped_by_label = transform_data.groupby('Label')
print_groups(grouped_by_label)

Out[25]:
A
  Label  Values  Values2 Other
V     A       0        5   foo
Y     A       3        8   fiz
B
  Label  Values  Values2 Other
X     B       2        7   baz
C
  Label  Values  Values2 Other
W     C       1        6   bar
Z     C       4        9   buz
```

Следующий программный код выполняет преобразование, в ходе которого применяется функция, добавляющая 10 к каждому значению:

```
In[26]:
# добавляем 10 ко всем значениям во всех столбцах
grouped_by_label.transform(lambda x: x + 10)

Out[26]:
   Values  Values2
V      10       15
W      11       16
X      12       17
Y      13       18
Z      14       19
```

Библиотека pandas пытается применить эту функцию ко всем столбцам, но поскольку столбцы `Label` и `Other` имеют строковые значения, функция, выполняющая преобразование, терпит неудачу (она выдает исключение). Из-за этого исключения два вышеупомянутых столбца исключены из результата.
Кроме того, полученный результат не сгруппирован, поскольку структура группировки удалена из него. Итоговый объект получит индекс, соответствующий индексу исходного объекта `DateFrame`, в данном случае `V`, `W`, `X`, `Y` и `Z`.

Заполнение пропущенных значений групповым средним

В статистическом анализе довольно распространенный вид преобразования, использующий сгруппированные данные – это замена пропущенных данных в каждой группе групповым средним (рассчитывается на основе непропущенных значений). Для иллюстрации следующий программный код создает объект `DataFrame` со столбцом `Label`, состоящим из двух значений (`A` и `B`), и столбцом `Values`, содержащим ряд целочисленных значений, при этом одно значение заменено на значение `NaN`. Затем данные будут сгруппированы по столбцу `Label`:

```
In[27]:
# создаем данные, чтобы продемонстрировать способ
# замены пропусков
```

```
df = pd.DataFrame({ 'Label': list("ABABAB"),
                    'Values': [10, 20, 11, np.nan, 12, 22]})
grouped = df.groupby('Label')
print_groups(grouped)
```

Out[27]:

```
A
  Label  Values
0     A    10.0
2     A    11.0
4     A    12.0
B
  Label  Values
1     B    20.0
3     B     NaN
5     B    22.0
```

С помощью метода `.mean()` можно вычислить среднее значение для каждой группы:

In[28]:

```
# вычисляем среднее для каждой группы
grouped.mean()
```

Out[28]:

```
       Values
Label
A        11.0
B        21.0
```

Теперь предположим, что вам нужно заполнить все значения `NaN` в группе B, потому что затем в ходе выполнения других операций могут возникнуть трудности при обработке значений `NaN`. Эту задачу можно лаконично выполнить с помощью следующего программного кода:

In[29]:

```
# используем метод .transform(), чтобы заполнить
# значения NaN групповым средним
filled_NaNs = grouped.transform(lambda x: x.fillna(x.mean()))
filled_NaNs
```

Out[29]:

```
   Values
0    10.0
1    20.0
2    11.0
3    21.0
4    12.0
5    22.0
```

Вычисление нормализованных z-значений с помощью преобразования

Еще один распространенный вид преобразования – вычисление нормализованных z-значений по группам данных. Для иллюстрации мы возьмем сгенерированный случайным образом ряд значений, используя нормальное распределение со средним значением 0,5 и стандартным отклонением 2. Данные индексируются по дням и на основе 100-дневного окна вычисляется скользящее среднее для каждого значения:

In[30]:

```
# генерируем временной ряд со скользящими средними
```

```
np.random.seed(123456)
data = pd.Series(np.random.normal(0.5, 2, 365*3),
                 pd.date_range('2013-01-01', periods=365*3))
periods = 100
rolling = data.rolling(
    window=periods,
    min_periods=periods,
    center=False).mean().dropna()
rolling[:5]
```

Out[30]:

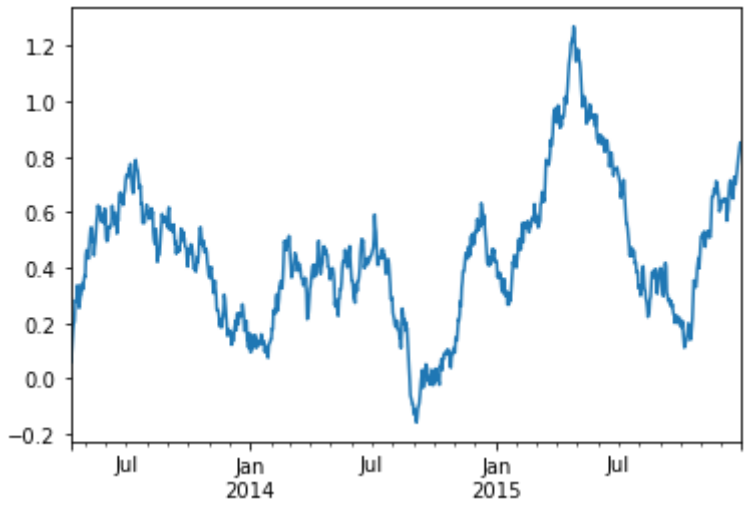
```
2013-04-10    0.073603
2013-04-11    0.057313
2013-04-12    0.089255
2013-04-13    0.133248
2013-04-14    0.175876
Freq: D, dtype: float64
```

Скользящие средние имеют следующий вид:

In[31]:

```
# визуализируем ряд
rolling.plot();
```

Out[31]:



А сейчас мы стандартизируем скользящие средние по каждому календарному году. Следующий программный код группирует данные по годам и сообщает о фактическом среднем значении и стандартном отклонении каждой группы:

In[32]:

```
# вычисляем среднее и стандартное отклонение
# по каждому году
group_key = lambda x: x.year
groups = rolling.groupby(group_key)
groups.agg([np.mean, np.std])
```

Out[32]:

	mean	std
2013	0.454233	0.171988
2014	0.286502	0.182040
2015	0.599447	0.275786

Для выполнения стандартизации следующий программный код задает функцию, вычисляющую z-значения, применяет ее в качестве преобразования к каждой группе и сообщает о новом среднем значении и новом стандартном отклонении:

In[33]:
```
# выполняем z-преобразование
z_score = lambda x: (x - x.mean()) / x.std()
normed = rolling.groupby(group_key).transform(z_score)
normed.groupby(group_key).agg([np.mean, np.std])
```

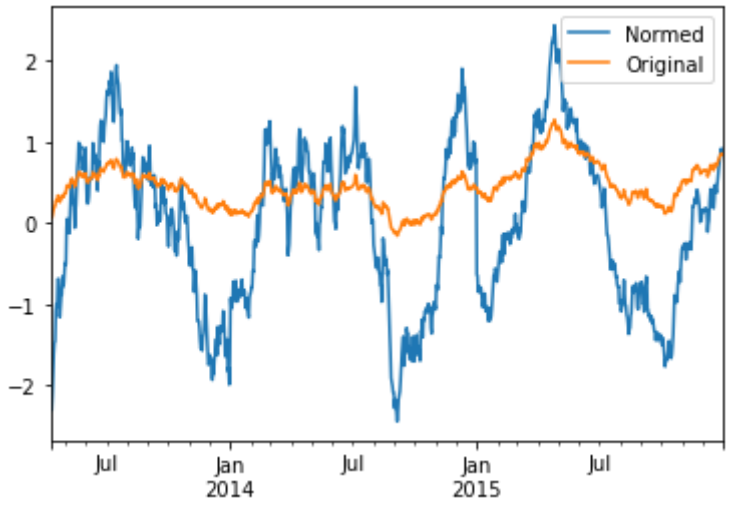
Out[33]:

	mean	std
2013	-3.172066e-17	1.0
2014	-1.881296e-15	1.0
2015	-1.492261e-15	1.0

Еще мы можем сравнить исходные и преобразованные данные:

In[34]:
```
# визуализируем исходные и
# стандартизированные данные
compared = pd.DataFrame({ 'Original': rolling,
                          'Normed': normed })
compared.plot();
```

Out[34]:



Исключение групп из процедуры агрегирования

С помощью метода `.filter()` можно выборочно удалить группы данных из обработки. В этот метод передают функцию, с помощью которой по каждой группе можно принять решение о включении данной группы в итоговый результат. Если функция возвращает значение `True`, группа включается в результат, если функция возвращает значение `False`, группа исключается.

Мы рассмотрим несколько сценариев, используя следующие данные:

In[35]:
```
# создаем данные для наших примеров
df = pd.DataFrame({'Label': list('AABCCC'),
                   'Values': [1, 2, 3, 4, np.nan, 8]})
df
```

Out[35]:
```
  Label  Values
0     A     1.0
1     A     2.0
2     B     3.0
3     C     4.0
4     C     NaN
5     C     8.0
```

Сначала удалим группы, которые не соответствуют установленному минимуму элементов. В частности, удалим группы с одним непропущенным значением и меньше:

In[36]:
```
# удаляем группы с одним непропущенным
# значением и меньше
f = lambda x: x.Values.count() > 1
df.groupby('Label').filter(f)
```

Out[36]:
```
  Label  Values
0     A     1.0
1     A     2.0
3     C     4.0
4     C     NaN
5     C     8.0
```

В следующем примере мы удалим группы с любым количеством пропущенных значений:

In[37]:
```
# удаляем группы, в которых есть пропуски
f = lambda x: x.Values.isnull().sum() == 0
df.groupby('Label').filter(f)
```

Out[37]:
```
  Label  Values
0     A     1.0
1     A     2.0
2     B     3.0
```

Теперь отберем только те группы, в которых среднее значение превышает значение 20 – среднее значение всего набора данных (это позволяет выбрать группы, которые отличаются от остальных):

In[38]:
```
# отбираем группы со средним 2.0 и выше
grouped = df.groupby('Label')
group_mean = grouped.mean().mean()
f = lambda x: abs(x.Values.mean() - group_mean) > 2.0
df.groupby('Label').filter(f)
```

```
Out[38]:
  Label  Values
3     C     4.0
4     C     NaN
5     C     8.0
```

Выводы

В этой главе мы рассмотрели различные методы группировки и разные виды анализа сгруппированных данных с помощью библиотеки pandas. Мы познакомились с общими принципами схемы «разделение – применение – объединение» и реализацией этой схемы в библиотеке pandas. Затем мы научились разбивать данные на группы на основе значений столбцов и уровней индекса. Позднее мы рассмотрели способы обработки данных в каждой группе с помощью агрегирующих функций и преобразований. В заключение мы быстро изучили способы фильтрации групп данных на основе их содержимого. В следующей главе мы подробно познакомимся с одним из самых мощных инструментов библиотеки pandas - анализ временных рядов.

ГЛАВА 13 АНАЛИЗ ВРЕМЕННЫХ РЯДОВ

Временной ряд представляет собой измерение одной или нескольких переменных за определенный период времени и с заданным интервалом. Например, необходимо ежедневно фиксировать стоимость акций в течение года. После того как данные временного ряда получены, часто проводится поиск паттернов временного ряда, по сути, мы выясняем, *что* происходит с течением времени. Возможность анализировать данные временных рядов востребована в современном мире, будь то анализ финансовой информации или мониторинг физических нагрузок, фиксируемых гаджетом, с последующей проверкой того, насколько ваши упражнения соответствуют заявленным целям и диете.

Библиотека pandas предлагает широкие возможности для анализа временных рядов. В этой главе мы рассмотрим массу таких возможностей, в том числе:

- Создание временного ряда с определенной частотой
- Представление дат, времени и интервалов
- Создание временной метки с помощью объекта Timestamp
- Использование объекта Timedelta для представления временного интервала
- Индексация с помощью DatetimeIndex
- Создание временных рядов с определенными частотами
- Представление интервалов данных с помощью смещений дат
- Привязка периодов к конкретным дням недели, месяца,
- квартала или года
- Моделирование интервала времени с помощью объекта Period
- Индексация с помощью PeriodIndex
- Обработка праздников с помощью календарей
- Нормализация временных меток с помощью часовых поясов
- Сдвиг временного ряда с опережением и с запаздыванием
- Преобразование частоты временного ряда
- Увеличение или уменьшение шага дискретизации временного ряда
- Применение к временному ряду операций на основе скользящего окна

Настройка библиотеки pandas

Мы начнем рассмотрение примеров в этой главе, задав следующие инструкции для импорта библиотек и настройки вывода:

```
In[1]:
# импортируем библиотеки numpy и pandas
import numpy as np
import pandas as pd

# импортируем библиотеку datetime для работы с датами
import datetime
from datetime import datetime, date

# Задаем некоторые опции библиотеки pandas, которые
# настраивают вывод
pd.set_option('display.notebook_repr_html', False)
pd.set_option('display.max_columns', 8)
pd.set_option('display.max_rows', 10)
pd.set_option('display.width', 60)
```

```
# импортируем библиотеку matplotlib для построения графиков
import matplotlib.pyplot as plt
%matplotlib inline
```

Представление дат, времени и интервалов

Для лучшего понимания временных рядов нам нужно сначала выяснить, какие объекты в библиотеке pandas используются для представления дат, времени и временных интервалов. Библиотека pandas предлагает встроенные средства для представления дат, времени и временных интервалов времени, поскольку в Python или NumPy таких инструментов, необходимых для обработки временных рядов, недостаточно.

Некоторые дополнительные возможности позволяют преобразовывать данные, меняя частоту, и использовать различные календари, чтобы учитывать в финансовых расчетах рабочие дни и праздники.

Объекты datetime, day и time

Объект `datetime` является частью библиотеки `datetime`, а не частью библиотеки pandas. Этот класс можно использовать для построения объектов, представляющих собой несколько распространенных паттернов, например, фиксированный момент времени, включающий дату и время, или просто день без компонента времени или время без компонента даты.

Объекты `datetime` не обеспечивают той точности, которая необходима большинству математиков, выполняющих тщательные расчеты на основе временных рядов. Однако именно они обычно используются для инициализации объектов pandas и под капотом библиотека pandas преобразуют их в объекты `Timestamp`. Поэтому они все-таки заслуживают краткого упоминания, поскольку будут часто использоваться при инициализации.

Объект `datetime` может инициализировать с помощью трех обязательных параметров, представляющих год, месяц и день:

```
In[2]:
# объект datetime для 15 декабря 2014 года
datetime(2014, 12, 15)
Out[2]:
datetime.datetime(2014, 12, 15, 0, 0)
```

По умолчанию, если часы и минуты не указаны, фиксируется `0` часов и `0` минут. Часы и минуты тоже можно указать с помощью двух дополнительных значений в конструкторе класса `datetime`. Следующий программный код создает объект `datetime.datetime`, задающий дополнительно время 17:30:

```
In[3]:
# задаем конкретную дату, а также время 17:30
datetime(2014, 12, 15, 17, 30)
```

```
Out[3]:
datetime.datetime(2014, 12, 15, 17, 30)
```

Текущие дату и время можно определить с помощью метода `now()`, который извлекает местные дату и время:

```
In[4]:
# получаем текущие дату и время
# для местного часового пояса
datetime.now()

Out[4]:
datetime.datetime(2018, 8, 14, 20, 1, 28, 168222)
```

Объект `datetime.date` представляет определенный день без компонента времени. Его можно создать, передав объект `datetime` конструктору класса `date`:

```
In[5]:
# можно записать дату без компонента времени,
# создав дату с помощью объекта datetime
datetime.date(datetime(2014, 12, 15))

Out[5]:
datetime.date(2014, 12, 15)
```

Текущую дату для данного часового пояса можно извлечь с помощью следующего программного кода:

```
In[6]:
# просто извлекаем текущую дату
datetime.now().date()

Out[6]:
datetime.date(2018, 8, 14)
```

Объект `datetime.time` представляет определенное время без компонента даты. Его можно создать, передав объект `datetime` конструктору класса `time`:

```
In[7]:
# просто извлекаем время из объекта datetime
datetime.time(datetime(2014, 12, 15, 17, 30))

Out[7]:
datetime.time(17, 30)
```

Текущее местное время можно получить с помощью следующего программного кода:

```
In[8]:
# получаем текущее местное время
datetime.now().time()
Out[8]:
datetime.time(20, 1, 28, 472703)
```

Создание временной метки с помощью объекта Timestamp

В библиотеке pandas дату и время можно представить с помощью класса `pandas.tslib.Timestamp`. Объект библиотеки pandas `Timestamp` основан на типе NumPy `datetime64` и имеет более высокую точность, чем питоновский объект `datetime`. В библиотеке pandas объекты `Timestamp` и объекты `datetime` обычно взаимозаменяемы, поэтому вы можете использовать объекты `Timestamp` везде, где можно использовать объекты `datetime`.

Вы можете создать объект `Timestamp` с помощью `pd.Timestamp` (ярлык для `pandas.tslib.Timestamp`), передав строку, представляющую дату, время или дату и время:

```
In[9]:
# временная метка, представляющая конкретную дату
pd.Timestamp('2014-12-15')

Out[9]:
Timestamp('2014-12-15 00:00:00')
```

Можно также указать компонент времени:

```
In[10]:
# временная метка, содержащая дату и время
pd.Timestamp('2014-12-15 17:30')

Out[10]:
Timestamp('2014-12-15 17:30:00')
```

Кроме того, объект `Timestamp` можно создать, задав лишь время, и по умолчанию будет добавлена местная текущая дата:

```
In[11]:
# задаем временную метку, указав только время
# по умолчанию ко времени будет добавлена
# местная текущая дата
pd.Timestamp('17:30')

Out[11]:
Timestamp('2018-08-14 17:30:00')
```

Следующий программный код показывает, как получить текущие дату и время с помощью объекта `Timestamp`:

```
In[12]:
# получаем текущие дату и время
pd.Timestamp("now")

Out[12]:
Timestamp('2018-08-14 20:01:28.960018')
```

Обычно пользователи pandas не создают объекты `Timestamp` напрямую. Многие функции pandas, которые работают с датами и временем, позволяют принять в качестве аргумента объект `datetime` или текстовое представление даты/времени, а затем выполняют преобразование.

Использование объекта Timedelta для представления временного интервала

Для представления разности между двумя моментами времени используется объект библиотеки pandas `Timedelta`. Он используется для вычисления временного интервала между двумя датами или для вычисления даты, отстоящей от другой даты и/или времени на определенный промежуток времени. Чтобы продемонстрировать использование объект `Timedelta`, следующий программный код использует объект `Timedelta` для вычисления даты, отстоящей от указанной даты на один день вперед:

```
In[13]:
# вычисляем дату, которая будет отстоять от
# даты 2014-11-30 на один день вперед
today = datetime(2014, 11, 30)
tomorrow = today + pd.Timedelta(days=1)
tomorrow

Out[13]:
datetime.datetime(2014, 12, 1, 0, 0)
```

Следующий программный код вычисляет количество дней между двумя датами:

```
In[14]:
# вычисляем количество дней между двумя датами
date1 = datetime(2014, 12, 2)
date2 = datetime(2014, 11, 28)
date1 - date2

Out[14]:
datetime.timedelta(4)
```

Введение во временные ряды

Библиотека pandas отлично подходит для работы с временными рядами. Это скорее всего обусловлено тем, что изначально она разрабатывалась для обработки финансовой информации. Арсенал средств для работы с временными рядами постоянно пополняется из версии в версию и тем самым возможности обработки временных рядов постепенно возрастают.

Индексация с помощью объекта DatetimeIndex

В основе работы с временными рядами лежит использование специализированных индексов. Эти индексы представляют собой измерения данных, зафиксированные по одной или нескольким временным меткам. В библиотеке pandas они являются объектами `DatetimeIndex`. Это невероятно функциональные объекты, и они позволяют автоматически выравнивать данные на основе дат и времени. Объекты `DatetimeIndex` в библиотеке pandas можно создать несколькими способами. Следующий программный код создает объект `DateTimeIndex`, передав список объектов `datetime` в объект `Series`:

```
In[15]:
```

```
# создаем очень простой временной ряд с двумя
# индексными метками и случайными значениями
dates = [datetime(2014, 8, 1), datetime(2014, 8, 2)]
ts = pd.Series(np.random.randn(2), dates)
ts
```

```
Out[15]:
2014-08-01    1.565070
2014-08-02    0.510995
dtype: float64
```

Мы взяли объекты `datetime` и создали на основе дат объект `DatetimeIndex`. Каждое значение этого индекса является объектом `Timestamp`.
Следующий программный код проверяет тип индекса и типы меток в индексе:

```
In[16]:
# смотрим тип индекса
type(ts.index)
```

```
Out[16]:
pandas.core.indexes.datetimes.DatetimeIndex
In[17]:
# можно увидеть, что это коллекция временных меток
type(ts.index[0])
```

```
Out[17]:
pandas._libs.tslib.Timestamp
```

Чтобы создать временной ряд, необязательно передавать список объектов `datetime`. Объект `Series` достаточно умен, чтобы распознать строку как объект `datetime` и выполнить преобразование за вас. Следующий пример эквивалентен предыдущему:

```
In[18]:
# создаем временной ряд из списка дат,
# записанных в виде строк!
np.random.seed(123456)
dates = ['2014-08-01', '2014-08-02']
ts = pd.Series(np.random.randn(2), dates)
ts
```

```
Out[18]:
2014-08-01    0.469112
2014-08-02   -0.282863
dtype: float64
```

Библиотека pandas предлагает удобную функцию `pd.to_datetime()`. Она принимает в качестве аргумента последовательность объектов одного и того же или смешанного типа, преобразует их в объекты `Timestamp`, а те в свою очередь сформируют объект `DatetimeIndex`. Если объект, записанный в последовательности, не удается преобразовать, библиотека pandas создает значение `NaT`, то есть значение не является временем (Not-a-Time):

```
In[19]:
# преобразовываем последовательность объектов
# в объект DatetimeIndex
dti = pd.to_datetime(['Aug 1, 2014',
                      '2014-08-02',
                      '2014.8.3',
                      None])
```

```
for l in dti: print (l)
```

Out[19]:
```
2014-08-01 00:00:00
2014-08-02 00:00:00
2014-08-03 00:00:00
NaT
```

Будьте внимательны, поскольку функция `pd.to_datetime()` выдаст исключение, если оно не сможет преобразовать значение в объект `Timestamp`:

In[20]:
```
# это список объектов, а не временных меток...
# в версии 0.20.1 нижеприведенный программный код
# выдает ошибку
# pd.to_datetime(['Aug 1, 2014', 'foo'])
```

Чтобы заставить функцию преобразовать даты вместо выброса исключения, вы можете воспользоваться параметром `errors="coerce"`. Значения, которые нельзя преобразовать, получат в итоговом индексе значение `NaT`:

In[21]:
```
# принудительно выполняем преобразование, значения, которые
# не удалось преобразовать, получат значения NaT
pd.to_datetime(['Aug 1, 2014', 'foo'], errors="coerce")
```

Out[21]:
```
DatetimeIndex(['2014-08-01', 'NaT'], dtype='datetime64[ns]', freq=None)
```

С помощью функции `pd.date_range()` можно легко создать диапазон временных меток с определенной частотой. Следующий программный код на основе объекта `DatetimeIndex`, состоящего из 10 последовательных дней, создает объект `Series`:

In[22]:
```
# создаем диапазон дат, начинающийся с определенной даты
# и включающий определеное количество дней,
# на его основе создаем объект Series
np.random.seed(123456)
periods = pd.date_range('8/1/2014', periods=10)
date_series = pd.Series(np.random.randn(10), index=periods)
date_series
```

Out[22]:
```
2014-08-01    0.469112
2014-08-02   -0.282863
2014-08-03   -1.509059
2014-08-04   -1.135632
2014-08-05    1.212112
2014-08-06   -0.173215
2014-08-07    0.119209
2014-08-08   -1.044236
2014-08-09   -0.861849
2014-08-10   -2.104569
Freq: D, dtype: float64
```

Объект `DatetimeIndex` можно использовать для выполнения различных операций с индексом типа выравнивания, отбора и создания срезов. Ниже показан пример создания среза с помощью позиций индекса:

In[23]:
```
# создаем срез, используя позиции индекса
subset = date_series[3:7]
subset
```

Out[23]:
```
2014-08-04   -1.135632
2014-08-05    1.212112
2014-08-06   -0.173215
2014-08-07    0.119209
Freq: D, dtype: float64
```

Для иллюстрации выравнивания мы воспользуемся объектом `Series`, который создадим с помощью индекса только что полученного среза:

In[24]:
```
# создаем объект Series для
# иллюстрации выравнивания
s2 = pd.Series([10, 100, 1000, 10000], subset.index)
s2
```

Out[24]:
```
2014-08-04       10
2014-08-05      100
2014-08-06     1000
2014-08-07    10000
Freq: D, dtype: int64
```

Когда мы складываем `date_series` и `s2` вместе, выполняется выравнивание, возвращающее значения `NaN` там, где элементы невозможно выровнять. Значение напротив определенной метки индекса – это сумма значений, найденных в соответствующих индексных метках объектов `date_series` и `s2`:

In[25]:
```
# демонстрируем выравнивание по дате,
# сложив вместе date_series и s2
date_series + s2
```

Out[25]:

```
2014-08-01             NaN
2014-08-02             NaN
2014-08-03             NaN
2014-08-04        8.864368
2014-08-05      101.212112
2014-08-06      999.826785
2014-08-07    10000.119209
2014-08-08             NaN
2014-08-09             NaN
2014-08-10             NaN
Freq: D, dtype: float64
```

Элементы серии с типом `DatetimeIndex` можно извлечь с помощью строкового представления даты, не указывая объект `datetime`:

In[26]:
```
# находим элемент с помощью строкового
# представления даты
date_series['2014-08-05']
```

Out[26]:
```
1.2121120250208506
```

Кроме того, можно создать срез объекта `DatetimeIndex`, задав диапазон, состоящий из строковых представлений дат:

In[27]:
```
# создаем срез объекта DatetimeIndex, задав диапазон
# из строковых представлений дат
date_series['2014-08-05':'2014-08-07']
```

Out[27]:
```
2014-08-05    1.212112
2014-08-06   -0.173215
2014-08-07    0.119209
Freq: D, dtype: float64
```

Еще одна удобная особенность библиотеки pandas заключается в том, что можно создать срез объекта `DatetimeIndex` с помощью детализированных спецификаций дат. Например, следующий программный код создает объект `Series` с датами, охватывающими 2-летний период, а затем отбирает только те даты, которые относятся к 2013 году:

In[28]:
```
# создаем диапазон дат, охватывающий 2-летний
# период с ежедневным интервалом
# отбираем лишь те даты, которые
# относятся к 2013 году
s3 = pd.Series(0, pd.date_range('2013-01-01', '2014-12-31'))
s3['2013']
```

Out[28]:
```
2013-01-01    0
2013-01-02    0
2013-01-03    0
2013-01-04    0
2013-01-05    0
             ..
2013-12-27    0
2013-12-28    0
2013-12-29    0
2013-12-30    0
2013-12-31    0
Freq: D, Length: 365, dtype: int64
```

Кроме того, можно отобрать даты, относящиеся к конкретному году и месяцу. Следующий программный код отбирает даты, относящиеся к маю 2014 года:

```
In[29]:
# отбираем 31 дату за май 2014 года
s3['2014-05']

Out[29]:
2014-05-01    0
2014-05-02    0
2014-05-03    0
2014-05-04    0
2014-05-05    0
              ..
2014-05-27    0
2014-05-28    0
2014-05-29    0
2014-05-30    0
2014-05-31    0
Freq: D, Length: 31, dtype: int64
```

Этот способ можно использовать при работе со срезами. Следующий программный код возвращает даты с августа по сентября 2014 года:

```
In[30]:
# извлекаем даты с августа
# по сентябрь 2014 года
s3['2014-08':'2014-09']

Out[30]:
2014-08-01    0
2014-08-02    0
2014-08-03    0
2014-08-04    0
2014-08-05    0
              ..
2014-09-26    0
2014-09-27    0
2014-09-28    0
2014-09-29    0
2014-09-30    0
Freq: D, Length: 61, dtype: int64
```

Создание временного ряда с определенной частотой

Частота временного ряда не обязательно должна быть ежедневной. С помощью функции `pd.date_range()` и параметра `freq` можно сгенерировать различные частоты. По умолчанию параметр `freq` имеет значение `'D'`, которое обозначает суточную частоту.

Для иллюстрации другой частоты временного ряда следующий программный код с помощью значения параметра `freq='T'` создает объект `DatetimeIndex`, состоящий из 1-минутных интервалов:

```
In[31]:
# создаем объект Series, состоящий
# из 1-минутных интервалов
np.random.seed(123456)
bymin = pd.Series(np.random.randn(24*60*90),
                  pd.date_range('2014-08-01',
                                '2014-10-29 23:59',
                                freq='T'))
bymin[:5]

Out[31]:
2014-08-01 00:00:00    0.469112
2014-08-01 00:01:00   -0.282863
2014-08-01 00:02:00   -1.509059
2014-08-01 00:03:00   -1.135632
2014-08-01 00:04:00    1.212112
Freq: T, dtype: float64
```

Этот временной ряд позволяет нам получить срез с более тонким разрешением. Следующий программный код создает срез с поминутными интервалами:

```
In[32]:
# создаем срез с поминутными
# интервалами
bymin['2014-08-01 00:02':'2014-08-01 00:07']

Out[32]:
2014-08-01 00:02:00   -1.509059
2014-08-01 00:03:00   -1.135632
2014-08-01 00:04:00    1.212112
2014-08-01 00:05:00   -0.173215
2014-08-01 00:06:00    0.119209
2014-08-01 00:07:00   -1.044236
Freq: T, dtype: float64
```

В нижеприведенной таблице перечислены возможные частоты:

Обозначение	Описание
B	Каждый рабочий день
C	Каждый указанный рабочий день
D	Ежедневно (по умолчанию)
W	Еженедельно
M	Каждый последний календарный день месяца
BM	Каждый последний рабочий день месяца
CBM	Каждый указанный последний рабочий день месяца
MS	Каждый первый календарный день месяца
BMS	Каждый первый рабочий день месяца
CBMS	Каждый указанный первый рабочий день месяца
Q	Ежеквартально
BQ	Каждый последний рабочий день квартала
QS	Каждый первый день квартала
BQS	Каждый первый рабочий день квартала
A	Каждый последний день года
BA	Каждый последний рабочий день года
AS	Каждый первый день года
BAS	Каждый первый рабочий день года
H	Ежечасно
T	Ежеминутно
S	Ежесекундно
L	Каждую миллисекунду
U	Каждую микросекунду

Чтобы создать временной ряд, состоящий только из рабочих дней, воспользуйтесь строковым значением `'B'`.

```
In[33]:
# создаем серию, состоящую из рабочих дней
days = pd.date_range('2014-08-29', '2014-09-05', freq='B')
days

Out[33]:
DatetimeIndex(['2014-08-29', '2014-09-01', '2014-09-02',
               '2014-09-03', '2014-09-04', '2014-09-05'],
              dtype='datetime64[ns]', freq='B')
```

Взглянув на метки индекса, видим, что два дня были пропущены, поскольку они являются выходными днями.
С помощью параметра `period` можно создать диапазон, начинающийся с определенной даты и времени, с определенной частотой и с определенным количеством периодов. Следующий программный код создает объект `DatetimeIndex`, состоящий из 5 дат. Временной ряд начинается с 2014-08-01 12:10:01 и содержит 5 посекундных интервалов:

```
In[34]:
# задаем периоды
pd.date_range('2014-08-01 12:10:01', freq='S', periods=5)

Out[34]:
DatetimeIndex(['2014-08-01 12:10:01',
               '2014-08-01 12:10:02',
               '2014-08-01 12:10:03',
               '2014-08-01 12:10:04',
               '2014-08-01 12:10:05'],
              dtype='datetime64[ns]', freq='S')
```

Вычисление новых дат с помощью смещений

В библиотеке pandas для каждой базовой частоты определен объект, называемый **смещением даты (data offset)**. Мы затронули это понятие в начале главы, когда рассматривали объекты `Timedelta`. Библиотека pandas расширяет эти возможности с помощью объекта `DateOffset`. Объекты `DataOffset` содержат информацию о смещении времени и частоте, которую можно использовать для изменения объектов `DatetimeIndex`.

Представление временных интервалов с помощью смещений дат

С помощью строковых значений `'M'`, `'W'` и `'BM'`, указываемых для параметра `freq` функции `pd.date_range()`, можно создавать объекты `DatetimeIndex` с разной частотой временного ряда. Под капотом строковое значение, задающее частоту, будет преобразовано в экземпляр объекта `DateOffset`.
Объект `DateOffset` представляет собой приращение частоты. Смещения дат, например, «месяц», «рабочий день» или «час», представлены в библиотеке pandas различными подклассами класса `DateOffset`. Класс

`DateOffset` позволяет задать способ вычисления конкретного временного интервала на основе исходных даты и времени. Это позволяет пользователю библиотеки pandas более гибко задавать смещение даты/ времени вместо простого использования фиксированного количественного интервала.
Практическим примером является вычисление следующего рабочего дня. Его нельзя определить, просто добавив один день к объекту `datetime`. Если дата представляет собой пятницу, следующий рабочий день на финансовом рынке США – это не суббота, а понедельник. А в некоторых случаях следующим рабочим днем после пятницы может быть вторник, если понедельник – это праздничный день. Библиотека pandas предлагает нам все инструменты, необходимые для работы с такими сложными сценариями.
Давайте рассмотрим это на конкретном примере, создав диапазон дат, используя в качестве частоты строковое значение `'B'`:

```
In[35]:
# извлекаем все рабочие дни в промежутке
# между двумя датами, включая эти даты
dti = pd.date_range('2014-08-29', '2014-09-05', freq='B')
dti.values

Out[35]:
array(['2014-08-29T00:00:00.000000000', '2014-09-01T00:00:00.000000000',
       '2014-09-02T00:00:00.000000000', '2014-09-03T00:00:00.000000000',
       '2014-09-04T00:00:00.000000000', '2014-09-05T00:00:00.000000000'], dtype='datetime64[n
s]')
```

Полученный временной ряд не содержит даты `2014-08-30` и `2014-08-30`, поскольку они являются субботой и воскресеньем и не считаются рабочими днями.
Объект `DatetimeIndex` имеет свойство `.freq`, которое позволяет узнать частоту временных меток в индексе:

```
In[36]:
# убеждаемся, что частотой является
# рабочий день
dti.freq

Out[36]:
<BusinessDay>
```

Обратите внимание, что для данного смещения библиотека pandas создала экземпляр класса `BusinessDay`. Как упоминалось ранее, библиотека pandas для каждого смещения даты определяет подкласс класса `DateOffset`. Ниже приведены встроенные классы для различных смещений дат:

Класс	Смещение
`DateOffset`	Календарный день (по умолчанию)
`BDay`	Рабочий день
`CDay`	Указанный рабочий день
`Week`	Неделя, опционно привязанная ко дню недели
`WeekOfMonth`	k-й день m-ной недели каждого месяца
`LastWeekOfMonth`	k-й день последней недели каждого месяца

MonthEnd	Последний календарный день месяца
MonthBegin	Первый календарный день месяца
BMonthEnd	Последний рабочий день месяца
BMonthBegin	Первый рабочий день месяца
CBMonthEnd	Указанный последний рабочий день месяца
CBMonthBegin	Указанный первый рабочий день месяца
QuarterEnd	Последний календарный день квартала
QuarterBegin	Первый календарный день квартала
BQuarterEnd	Последний рабочий день квартала
BQuarterBegin	Первый рабочий день квартала
FYS253Quarter	Торговый квартал (52-53 неделя)
YearEnd	Последний календарный день года
YearBegin	Первый календарный день года
BYearEnd	Последний рабочий день года
BYearBegin	Первый рабочий день года
FYS253	Торговый год (52-53 неделя)
Hour	Час
Minute	Минута
Second	Секунда
Milli	Миллисекунда
Micro	Микросекунда

Вышеперечисленные классы делают использование объектов `DateOffset` очень гибким. Объекты `DateOffset` можно использовать в различных сценариях:

- Их можно добавлять или вычитать для получения сдвинутой даты
- Их можно умножить на целое число (положительное или отрицательное) таким образом, что приращение будет применено несколько раз
- У них есть методы `rollforward` и `rollback` для сдвига даты вперед или назад к следующей или предыдущей «дате смещения»

Объекты `DateOffset` можно создать, передав их в объект `datetime`, который представляет собой фиксированный отрезок времени, или используя именованные аргументы. Именованные аргументы можно разделить на две основные категории. Первая категория – это ключевые слова, которые представляют собой абсолютные даты: год, месяц, день, час, минута, секунда и микросекунда. Вторая категория определяет относительную длительность и может быть представлена отрицательными значениями: годами, месяцами, неделями, днями, часами, минутами, секундами и микросекундами.

Следующий программный код создает смещение на один день и прибавляет его к объекту `datetime`:

```
In[37]:
# вычисляем однодневное смещение для 2014-8-29
d = datetime(2014, 8, 29)
do = pd.DateOffset(days = 1)
d + do
Out[37]:
Timestamp('2014-08-30 00:00:00')
```

Нижеприведенный программный код вычисляет дату, отстоящую от заданной даты на один рабочий день вперед:

```
In[38]:
# импортируем типы смещений дат
from pandas.tseries.offsets import *
```

```
# вычисляем дату, отстоящую от 2014-8-31
# на один рабочий день вперед
d + BusinessDay()
```

Out[38]:
```
Timestamp('2014-09-01 00:00:00')
```

Сдвиг сразу на несколько единиц частоты можно выполнить с помощью умножения:

In[39]:
```
# вычисляем дату, отстоящую от 2014-8-31
# на два рабочих дня вперед
d + 2 * BusinessDay()
```

Out[39]:
```
Timestamp('2014-09-02 00:00:00')
```

Нижеприведенный программный код демонстрирует использование объекта `BMonthEnd` для вычисления последнего рабочего дня месяца, следующего после определенной даты (в данном случае после `2014-09-02`):

In[40]:
```
# вычисляем последний рабочий день месяца,
# следующий после 2014-09-02
d + BMonthEnd()
```

Out[40]:
```
Timestamp('2014-09-30 00:00:00')
```

Нижеприведенный программный код для вычисления последнего рабочего дня месяца, следующего после `2014-09-15`, использует метод `.rollforward()` объекта `BMonthEnd`:

In[41]:
```
# вычисляем последний рабочий день месяца,
# следующий после 2014-09-15
BMonthEnd().rollforward(datetime(2014, 9, 15))
```

Out[41]:
```
Timestamp('2014-09-30 00:00:00')
```

Некоторые классы, задающие смещения, можно использовать в качестве параметров, чтобы точнее настроить сдвиг даты. В качестве примера следующий программный код вычисляет дату вторника (`weekday=1`), предшествующую дате `2014-08-31`:

```
In[42]:
# вычисляем дату вторника, предшествующую
# заданной дате
d - Week(weekday = 1)

Out[42]:
Timestamp('2014-08-26 00:00:00')
```

Привязанные смещения

Привязанные смещения – это определенные частоты, которые начинаются с определенного момента времени, например, с определенного дня недели, месяца или года. Для привязанных смещений используются заранее определенные сокращения. В качестве примера следующие строковые значения задают конкретный день недели:

Обозначение	Описание
W-SUN	Еженедельно в воскресенье
W-MON	Еженедельно в понедельник
W-TUE	Еженедельно во вторник
W-WED	Еженедельно в среду
W-THU	Еженедельно в четверг
W-FRI	Еженедельно в пятницу
W-SAT	Еженедельно в субботу

Например, следующий программный код создает индекс, который состоит из дат всех сред, приходящихся на промежуток между двумя указанными датами:

```
In[43]:
# вычисляем все среды в промежутке между
# 2014-06-01 и 2014-08-31
wednesdays = pd.date_range('2014-06-01',
                           '2014-07-31', freq="W-WED")
wednesdays.values

Out[43]:
array(['2014-06-04T00:00:00.000000000', '2014-06-11T00:00:00.000000000',
       '2014-06-18T00:00:00.000000000', '2014-06-25T00:00:00.000000000',
       '2014-07-02T00:00:00.000000000', '2014-07-09T00:00:00.000000000',
       '2014-07-16T00:00:00.000000000', '2014-07-23T00:00:00.000000000',
       '2014-07-30T00:00:00.000000000'], dtype='datetime64[ns]')
```

Привязанные смещения еще можно создать с помощью ежегодных и ежеквартальных частот. Эти привязанные частоты имеют общий вид `[B][A|Q][S]-[MON]`, где B (рабочие дни) и S (начало периода вместо конца) задаются по желанию, A – частота *ежегодно* или Q – частота *ежеквартально*, а MON - трехзначная аббревиатура для месяца (JAN, FEB, ...).

Для иллюстрации следующий программный код создает четыре ежеквартальные даты (каждая дата – первый рабочий день последнего месяца квартала), считая, что год заканчивается в июне:

```
In[44]:
# вычисляем первый рабочий день последнего
# месяца в каждом квартале
qends = pd.date_range('2014-01-01', '2014-12-31',
                      freq='BQS-JUN')
qends.values

Out[44]:
array(['2014-03-03T00:00:00.000000000', '2014-06-02T00:00:00.000000000',
       '2014-09-01T00:00:00.000000000', '2014-12-01T00:00:00.000000000'],
 dtype='datetime64[ns]')
```

Представление промежутков времени с помощью объектов Period

Многие операции над временными рядами требуют анализа событий за определенный промежуток времени. Простой пример – определение количества финансовых транзакций за определенный период.

Данный вид анализа можно выполнить с помощью объектов `Timestamp` и `DateOffset`, когда сначала задаются границы временного интервала, а затем определяется частота. Однако этот способ становится громоздким, когда вам нужно сгруппировать события в несколько периодов времени, поскольку в таком случае вам придётся работать с наборами объектов `Timestamp` и `DateOffset`.

Чтобы упростить организацию и анализ таких данных, библиотека pandas позволяет создавать интервалы времени как формальную конструкцию с помощью класса `Period`. Кроме того, библиотека pandas формализует работу с рядом объектов `Period` с помощью класса `PeriodIndex`, который позволяет выравнивать элементы данных на основе индексов соответствующих объектов `Period`.

Создание временного интервала с помощью объекта Period

Библиотека pandas формализует понятие временного интервала с помощью объекта `Period`. Объект `Period` позволяет вам задать промежутки времени, исходя из различных частот (ежедневной, еженедельной, ежемесячной, ежеквартальной, ежегодной и т.д.). При этом он будет включать в себя начальную и конечную временные метки, задающие определенный интервал времени.

Объект `Period` можно создать с помощью временной метки и частоты, при этом временная метка представляет собой «якорь», используемый в качестве точки отсчета, а частота – это продолжительность времени. Для иллюстрации следующий программный код создает период – один месяц, охватывающий август 2014 года:

In[45]:
```
# создаем период - один месяц, начинающийся
# в августе 2014 года
aug2014 = pd.Period('2014-08', freq='M')
aug2014
```

Out[45]:
```
Period('2014-08', 'M')
```

Свойства `start_time` и `end_time` объекта `Period` сообщают нам о начальной и конечной временных метках:

In[46]:
```
# смотрим начальную и конечную
# даты этого периода
aug2014.start_time, aug2014.end_time
```

Out[46]:
```
(Timestamp('2014-08-01 00:00:00'), Timestamp('2014-08-31 23:59:59.999999999'))
```

Поскольку мы указали период – август 2014 года, библиотека pandas определяет точку отсчета (`start_time`), а затем вычисляет конечную дату (`end_time`), исходя из указанной частоты. В данном случае она прибавляет один месяц к `start_time` и возвращает конечную дату.
Следующий программный код на основе переменной `aug2014` создает новый объект `Period`, который теперь сдвинут вперед на одну единицу частоты (на один месяц):

In[47]:
```
# вычисляем период, прибавив к периоду
# aug2014 единицу частоты (один месяц),
# теперь периодом будет сентябрь 2014 года
sep2014 = aug2014 + 1
sep2014
```

Out[47]:
```
Period('2014-09', 'M')
```

Понятие сдвига является очень важным. Прибавление `1` к объекту `Period` информирует последний о том, что нужно осуществить сдвиг на одну положительную единицу частоты периода. В данном случае мы сдвигаем период на один месяц вперед, чтобы получить новый период – сентябрь 2014 года.
Если мы взглянем на начальную и конечную временные метки объекта `sep2014`, мы увидим, что библиотека pandas правильно определила их:

In[48]:
```
sep2014.start_time, sep2014.end_time
```

Out[48]:
```
(Timestamp('2014-09-01 00:00:00'), Timestamp('2014-09-30 23:59:59.999999999'))
```

Обратите внимание, что объект `Period` учитывает, что сентябрь состоит из 30 дней, а не из 31. «Интеллектуальные возможности» объекта `Period` позволяют сэкономить время, затрачиваемое на ввод дополнительного программного кода, помогая нам решать различные сложные задачи, связанные с датами.

Индексация с помощью объекта PeriodIndex

Серию объектов `Period` можно объединить в специальный тип индекса, известный как `PeriodIndex`. Индекс `PeriodIndex` применяется в том случае, когда данные нужно связать с определенными временными интервалами, создать срезы и проанализировать события в каждом интервале.

Следующий программный код создает объект `PeriodIndex`, состоящий из 1-месячных интервалов 2013 года:

```
In[49]:
# создаем объект PeriodIndex, состоящий
# из 1-месячных интервалов 2013 года
mp2013 = pd.period_range('1/1/2013', '12/31/2013', freq='M')
mp2013

Out[49]:
PeriodIndex(['2013-01', '2013-02', '2013-03', '2013-04',
             '2013-05', '2013-06', '2013-07', '2013-08',
             '2013-09', '2013-10', '2013-11', '2013-12'],
            dtype='period[M]', freq='M')
```

`PeriodIndex` отличается от `DatetimeIndex` тем, что индексные метки являются объектами `Period`. Следующий программный код печатает начальную и конечную даты для всех объектов `Period` в индексе:

```
In[50]:
# пробегаем по всем объектам Period в индексе, печатая
# начальную и конечную даты для каждого объекта
for p in mp2013:
    print ("{0} {1}".format(p.start_time, p.end_time))

Out[50]:
2013-01-01 00:00:00 2013-01-31 23:59:59.999999999
2013-02-01 00:00:00 2013-02-28 23:59:59.999999999
2013-03-01 00:00:00 2013-03-31 23:59:59.999999999
2013-04-01 00:00:00 2013-04-30 23:59:59.999999999
2013-05-01 00:00:00 2013-05-31 23:59:59.999999999
2013-06-01 00:00:00 2013-06-30 23:59:59.999999999
2013-07-01 00:00:00 2013-07-31 23:59:59.999999999
2013-08-01 00:00:00 2013-08-31 23:59:59.999999999
2013-09-01 00:00:00 2013-09-30 23:59:59.999999999
2013-10-01 00:00:00 2013-10-31 23:59:59.999999999
2013-11-01 00:00:00 2013-11-30 23:59:59.999999999
2013-12-01 00:00:00 2013-12-31 23:59:59.999999999
```

Библиотека pandas определяет начальную и конечную даты каждого месяца, принимая во внимание фактическое количество дней в каждом конкретном месяце.

Мы можем создать объект `Series`, используя `PeriodIndex` в качестве индекса и связав каждое значение с соответствующим объектом `Period` в индексе:

```
In[51]:
# создаем объект Series, у которого
# индексом будет PeriodIndex
np.random.seed(123456)
ps = pd.Series(np.random.randn(12), mp2013)
ps[:5]

Out[51]:
2013-01    0.469112
2013-02   -0.282863
2013-03   -1.509059
2013-04   -1.135632
2013-05    1.212112
Freq: M, dtype: float64
```

Теперь у нас есть временной ряд, в котором значение, соответствующее конкретной индексной метке, представляет собой измерение, охватывающее определенный период времени. Примером такого ряда будет средняя стоимость ценной бумаги в данном месяце. Данная операция очень удобна, когда нужно изменить частоту временного ряда. Как и `DatetimeIndex`, `PeriodIndex` можно использовать для индексации значений с помощью объекта `Period` или строкового значения, задающего спецификацию периода. Для иллюстрации мы создадим еще один объект `Series`, похожий на предыдущий, но на этот раз он будет охватывать два года, 2013 и 2014:

```
In[52]:
# создаем объект Series с индексом PeriodIndex,
# который представляет собой все календарные
# месяцы-периоды 2013 и 2014 годов
np.random.seed(123456)
ps = pd.Series(np.random.randn(24),
               pd.period_range('1/1/2013',
                               '12/31/2014', freq='M'))
ps

Out[52]:
2013-01    0.469112
2013-02   -0.282863
2013-03   -1.509059
2013-04   -1.135632
2013-05    1.212112
             ...
2014-08   -1.087401
2014-09   -0.673690
2014-10    0.113648
2014-11   -1.478427
2014-12    0.524988
Freq: M, Length: 24, dtype: float64
```

Отдельные значения можно отобрать с помощью конкретной индексной метки, воспользовавшись либо объектом `Period`, либо строковым значением, представляющим период. Следующий программный код показывает, как можно отобрать отдельное значение с помощью строкового представления:

```
In[53]:
# извлекаем значение, соответствующее
# периоду 2014-06
ps['2014-06']

Out[53]:
0.567020349793672
```

Для получения значений можно воспользоваться и другими спецификациями. Например, следующий программный код извлекает значения для всех периодов в 2014 году:

```
In[54]:
# извлекаем значения для всех
# периодов в 2014 году
ps['2014']

Out[54]:
2014-01    0.721555
2014-02   -0.706771
2014-03   -1.039575
2014-04    0.271860
2014-05   -0.424972
             ...
2014-08   -1.087401
2014-09   -0.673690
2014-10    0.113648
2014-11   -1.478427
2014-12    0.524988
Freq: M, Length: 12, dtype: float64
```

Кроме того, можно создать срез объекта `PeriodIndex`. Следующий программный код извлекает все значения, соответствующие периодам с марта по июнь 2014 года включительно:

```
In[55]:
# извлекаем все значения, соответствующие периодам
# с марта по июнь 2014 года включительно
ps['2014-03':'2014-06']

Out[55]:
2014-03   -1.039575
2014-04    0.271860
2014-05   -0.424972
2014-06    0.567020
Freq: M, dtype: float64
```

Обработка праздников с помощью календарей

Ранее, когда мы вычисляли рабочий день, следующий после 29 августа 2014 года, мы получили ответ, что такой датой является 1 сентября 2014 года. На самом деле для США это не так: 1 сентября 2014 года приходится на 1-й понедельник сентября, который является федеральным праздником (Днем Труда), поэтому банки и биржи закрыты в этот день. Причиной такого ответа является тот факт, что по умолчанию библиотека pandas использует определенный календарь для расчета следующего рабочего дня, и этот календарь по умолчанию не считает 1 сентября 2014 года праздником.

Решение проблемы заключается в том, чтобы либо создать пользовательский календарь (здесь мы не будем вдаваться в подробности) или воспользоваться календарем `USFederalHolidayCalendar`. Затем этот календарь можно передать объекту `CustomBusinessDay`, который будет использоваться вместо объекта `BusinessDay`. Вычисления, осуществляемые с помощью объекта `CustomBusinessDay`, будут использовать новый календарь и учитывать федеральные праздники США.
Следующий программный код демонстрирует создание и использование объекта `USFederalCalendar`, позволяющего учесть даты, являющиеся праздниками:

```
In[56]:
# демонстрируем использование календаря
# федеральных праздников США
# сначала нужно импортировать его
from pandas.tseries.holiday import *
# создаем его и демонстрируем, что он
# учитывает праздники
cal = USFederalHolidayCalendar()
for d in cal.holidays(start='2014-01-01', end='2014-12-31'):
    print (d)

Out[56]:
2014-01-01 00:00:00
2014-01-20 00:00:00
2014-02-17 00:00:00
2014-05-26 00:00:00
2014-07-04 00:00:00
2014-09-01 00:00:00
2014-10-13 00:00:00
2014-11-11 00:00:00
2014-11-27 00:00:00
2014-12-25 00:00:00
```

Затем с помощью этого календаря можно вычислить рабочий день, следующий после 29 августа 2014 года:

```
In[57]:
# создаем объект CustomBusinessDay на основе
# календаря федеральных праздников США
cbd = CustomBusinessDay(holidays=cal.holidays())

# теперь вычисляем рабочий день,
# следующий после 2014-8-29
datetime(2014, 8, 29) + cbd

Out[57]:
Timestamp('2014-09-02 00:00:00')
```

В итоге операция вычисления теперь учитывает День Труда, не являющийся рабочим днем, и возвращает правильную дату `2014-09-02`.

Нормализация временных меток с помощью часовых поясов

Управление часовыми поясами, возможно, одна из самых сложных задач, с которой приходится сталкиваться при работе с временными

рядами. Данные часто собираются в разных уголках мира с использованием местного времени, и в какой-то момент потребуется согласовать данные, относящиеся к разным часовым поясам.
К счастью, библиотека pandas поддерживает работу с временными метками, принадлежащими к разным часовым поясам. Под капотом для выполнения операций, связанных с часовыми поясами, pandas использует библиотеки `pytz` и `dateutil`. Поддержка библиотеки `dateutil` является новинкой и в настоящее время реализована только для фиксированных смещений и `tzfile`-поясов. По умолчанию pandas использует библиотеку `pytz`, а также поддерживает библиотеку `dateutil` для обеспечения совместимости с другими приложениями.
Объекты, которые учитывают часовые пояса, поддерживают свойство `.tz`. По умолчанию объекты pandas, учитывающие часовые пояса, не используют объект `timezone` в целях повышения скорости вычислений. Следующий программный код извлекает текущее время и показывает, что по умолчанию информация о часовом поясе отсутствует:

In[58]:
```
# извлекаем текущее местное время и демонстрируем, что
# по умолчанию информация о часовом поясе отсутствует
now = pd.Timestamp('now')
now, now.tz is None
```

Out[58]:
```
(Timestamp('2018-08-14 20:01:33.160535'), True)
```

Этот программный код демонстрирует, что по умолчанию библиотека pandas обрабатывает Timestamp('now') как UTC-время[7], при этом информация о часовом поясе отсутствует. В целом я считаю, что если вы собираете данные, зависящие от времени, и их нужно сохранить для последующей обработки в будущем или эти данные нужно собрать из нескольких источников, оптимально всегда преобразовывать их в формат UTC.
Аналогично, `DatetimeIndex` и его объекты `Timestamp` по умолчанию не содержат информацию о часовом поясе:

[7] Всемирное координированное время (UTC) – международный стандарт времени, который пришел на смену Гринвичскому времени. Часовые пояса выражаются в виде смещений от UTC. Например, в Нью-Йорке время отстает от UTC на 4 часа в летний период(UTC-4) и на 5 часов (UTC-5 в остальное время года. В Москве время опережает UTC на три часа (UTC+3), переход на летнее время не применяется. – *Прим. пер.*

In[59]:
```
# по умолчанию DatetimeIndex и его объекты Timestamps
# не содержат информацию о часовом поясе
rng = pd.date_range('3/6/2012 00:00', periods=15, freq='D')
rng.tz is None, rng[0].tz is None
```

Out[59]:
```
(True, True)
```

Список названий стандартных часовых поясов можно получить с помощью программного кода, приведенного ниже. Если вы часто работаете с данными, которые записаны в разных часовых поясах, эти названия будут вам хорошо знакомы:

In[60]:
```
# импортируем стандартные часовые
# пояса из библиотеки pytz
from pytz import common_timezones
# выводим первые 5 часовых поясов
common_timezones[:5]
```

Out[60]:
```
['Africa/Abidjan',
 'Africa/Accra',
 'Africa/Addis_Ababa',
 'Africa/Algiers',
 'Africa/Asmara']
```

Местное UTC-время можно вычислить, воспользовавшись методом `.tz_localize()` объекта `Timestamp` и передав ему значение `UTC`:

In[61]:
```
# получаем текущее время и преобразуем в UTC
now = Timestamp("now")
local_now = now.tz_localize('UTC')
now, local_now
```

Out[61]:
```
(Timestamp('2018-08-14 20:01:33.494370'),
 Timestamp('2018-08-14 20:01:33.494370+0000', tz='UTC'))
```

Любой объект `Timestamp` можно преобразовать в определенный часовой пояс, передав название часового пояса в метод `.tz_localize()`:

In[62]:
```
# преобразуем временную метку
# в часовой пояс US/Mountain
tstamp = Timestamp('2014-08-01 12:00:00', tz='US/Mountain')
tstamp
```

Out[62]:
```
Timestamp('2014-08-01 12:00:00-0600', tz='US/Mountain')
```

С помощью параметра `tz` функции `pd.date_range()` можно создать индекс `DatetimeIndex` с определенным часовым поясом:

In[63]:
```
# создаем индекс DatetimeIndex
# с помощью часового пояса
rng = pd.date_range('3/6/2012 00:00:00',
                    periods=10, freq='D', tz='US/Mountain')
rng.tz, rng[0].tz
```

Out[63]:
```
(<DstTzInfo 'US/Mountain' LMT-1 day, 17:00:00 STD>,
 <DstTzInfo 'US/Mountain' MST-1 day, 17:00:00 STD>)
```

Кроме того, можно явно задать несколько часовых поясов. Следующий программный код создает два часовых пояса (два разных объекта `timezone`) и преобразует временную метку в каждый из них:

In[64]:
```
# демонстрируем использование
# объектов timezone
# необходимо импортировать
# библиотеку pytz
import pytz
# создаем два часовых пояса
mountain_tz = pytz.timezone("US/Mountain")
eastern_tz = pytz.timezone("US/Eastern")
# применяем оба к временной метке 'now'
mountain_tz.localize(now), eastern_tz.localize(now)
```

Out[64]:
```
(Timestamp('2018-08-14 20:01:33.494370-0600', tz='US/Mountain'),
 Timestamp('2018-08-14 20:01:33.494370-0400', tz='US/Eastern'))
```

Операции с несколькими временными рядами осуществляют выравнивание по объектам `Timestamp` в индексе, учитывая информацию о часовом поясе. Чтобы продемонстрировать это, мы воспользуемся следующим программным кодом, которое создает два объекта `Series` с индексами `DatetimeIndex`, каждый с одинаковой начальной датой, одинаковым количеством периодов и одинаковой частотой, но с разными часовыми поясами:

In[65]:
```
# создаем два объекта Series с одинаковой
# начальной датой, одинаковым количеством периодов,
# одинаковой частотой, но с разным часовым поясом
s_mountain = Series(np.arange(0, 5),
                    index=pd.date_range('2014-08-01',
                                        periods=5, freq="H",
                                        tz='US/Mountain'))
s_eastern = Series(np.arange(0, 5),
                   index=pd.date_range('2014-08-01',
                                       periods=5, freq="H",
                                       tz='US/Eastern'))
s_mountain
```

Out[65]:
```
2014-08-01 00:00:00-06:00    0
2014-08-01 01:00:00-06:00    1
2014-08-01 02:00:00-06:00    2
2014-08-01 03:00:00-06:00    3
2014-08-01 04:00:00-06:00    4
Freq: H, dtype: int32
```

In[66]:
```
s_eastern
```

Out[66]:
```
2014-08-01 00:00:00-04:00    0
2014-08-01 01:00:00-04:00    1
2014-08-01 02:00:00-04:00    2
2014-08-01 03:00:00-04:00    3
2014-08-01 04:00:00-04:00    4
Freq: H, dtype: int32
```

Следующий программный код демонстрирует выравнивание этих двух объектов `Series` по часовому поясу, сложив их вместе:

```
In[67]:
# складываем два объекта Series. В итоге происходит
# выравнивание только трех элементов
s_eastern + s_mountain

Out[67]:
2014-08-01 04:00:00+00:00    NaN
2014-08-01 05:00:00+00:00    NaN
2014-08-01 06:00:00+00:00    2.0
2014-08-01 07:00:00+00:00    4.0
2014-08-01 08:00:00+00:00    6.0
2014-08-01 09:00:00+00:00    NaN
2014-08-01 10:00:00+00:00    NaN
Freq: H, dtype: float64
```

Преобразовать часовой пояс в другой можно с помощью метода `.tz.convert()`:

```
In[68]:
# меняем часовой пояс с US/Eastern на US/Pacific
s_pacific = s_eastern.tz_convert("US/Pacific")
s_pacific

Out[68]:
2014-07-31 21:00:00-07:00    0
2014-07-31 22:00:00-07:00    1
2014-07-31 23:00:00-07:00    2
2014-08-01 00:00:00-07:00    3
2014-08-01 01:00:00-07:00    4
Freq: H, dtype: int32
```

Теперь, если мы сложим `s_pacific` и `s_mountain` вместе, выравнивание приведет к такому же результату:

```
In[69]:
# получим тот же результат, что и при сложении
# s_eastern и s_mountain, поскольку часовые
# пояса будут выровнены так же
s_mountain + s_pacific

Out[69]:
2014-08-01 04:00:00+00:00    NaN
2014-08-01 05:00:00+00:00    NaN
2014-08-01 06:00:00+00:00    2.0
2014-08-01 07:00:00+00:00    4.0
2014-08-01 08:00:00+00:00    6.0
2014-08-01 09:00:00+00:00    NaN
2014-08-01 10:00:00+00:00    NaN
Freq: H, dtype: float64
```

Операции с временными рядами

Теперь мы рассмотрим несколько распространенных операций, которые используются при работе с временными рядами. Эти операции включают в себя перегруппировку данных, изменение частоты временного ряда, изменение шага дискретизации временного ряда и вычисление агрегированных результатов по непрерывно скользящим подмножествам данных, чтобы отслеживать тенденции изменения данных с течением времени.

Опережение и запаздывание

Одна из распространенных операций, использующихся в работе с временными рядами, заключается в перемещении данных назад и вперед по временной оси. В библиотеке pandas для этого используется метод `.shift()`, который сдвигает значения серии или датафрейма на указанное количество единиц частоты в индексе. Чтобы продемонстрировать сдвиг, мы воспользуемся нижеприведенным объектом `Series`. Этот объект `Series` содержит пять значений, индексируется по датам, начиная с `2014-08-01`, и использует ежедневную частоту:

```
In[70]:
# создаем объект Series, с которым
# будем работать
np.random.seed(123456)
ts = Series([1, 2, 2.5, 1.5, 0.5],
            pd.date_range('2014-08-01', periods=5))
ts

Out[70]:
2014-08-01    1.0
2014-08-02    2.0
2014-08-03    2.5
2014-08-04    1.5
2014-08-05    0.5
Freq: D, dtype: float64
```

Следующий программный код сдвигает значения на 1 день вперед:

```
In[71]:
# сдвигаем значения на
# 1 день вперед
ts.shift(1)

Out[71]:
2014-08-01    NaN
2014-08-02    1.0
2014-08-03    2.0
2014-08-04    2.5
2014-08-05    1.5
Freq: D, dtype: float64
```

Библиотека pandas сдвинула значения вперед на одну единицу частоты индекса, которая равна одному дню. Сам индекс остается неизменным. Индексная метка `2014-08-01` теперь имеет значение `NaN`, поскольку отсутствует значение для замены.

Запаздывание - это сдвиг в отрицательном направлении. Следующий программный код сдвигает значения серии на `2` дня назад:

```
In[72]:
# сдвигаем значения на
# 2 дня назад
ts.shift(-2)

Out[72]:
2014-08-01    2.5
2014-08-02    1.5
2014-08-03    0.5
2014-08-04    NaN
2014-08-05    NaN
Freq: D, dtype: float64
```

Индексные метки `2014-08-04` и `2014-08-03` теперь имеют значения `NaN`, поскольку отсутствовали элементы для замены.
Распространенная операция, выполняемая с помощью сдвига, заключается в вычислении ежедневного процентного изменения какого-то финансового показателя. Эту операцию можно выполнить, поделив значения объекта `Series` на эти же значения, только сдвинутые на 1:

In[73]:
```
# вычисляем ежедневное процентное
# изменение
ts / ts.shift(1)
```

Out[73]:
```
2014-08-01         NaN
2014-08-02    2.000000
2014-08-03    1.250000
2014-08-04    0.600000
2014-08-05    0.333333
Freq: D, dtype: float64
```

Можно сдвинуть не значения, а сами временные метки. При этом индекс будет изменен, а значения останутся неизменными. В качестве примера следующий программный код сдвигает серию на один рабочий день вперед:

In[74]:
```
# сдвигаем вперед на один
# рабочий день
ts.shift(1, freq="B")
```

Out[74]:
```
2014-08-04    1.0
2014-08-04    2.0
2014-08-04    2.5
2014-08-05    1.5
2014-08-06    0.5
dtype: float64
```

Следующий программный код сдвигает даты на `5` часов вперед:

In[75]:
```
# сдвигаем на 5 часов вперед
ts.tshift(5, freq="H")
```

Out[75]:
```
2014-08-01 05:00:00    1.0
2014-08-02 05:00:00    2.0
2014-08-03 05:00:00    2.5
2014-08-04 05:00:00    1.5
2014-08-05 05:00:00    0.5
Freq: D, dtype: float64
```

Кроме того, временной ряд можно сдвинуть с помощью объекта `DateOffset`. Следующий программный код сдвигает временной ряд вперед на полминуты:

In[76]:
```
# сдвигаем с помощью объекта DateOffset
ts.shift(1, DateOffset(minutes=0.5))
```

Out[76]:

```
2014-08-01 00:00:30    1.0
2014-08-02 00:00:30    2.0
2014-08-03 00:00:30    2.5
2014-08-04 00:00:30    1.5
2014-08-05 00:00:30    0.5
Freq: D, dtype: float64
```

Существует еще один способ осуществить сдвиг дат, предлагаемый методом `.tshift()`. Этот метод сдвигает индексные метки на указанное количество единиц определенной частоты, при этом частоту задают с помощью параметра `freq`. Следующий программный код демонстрирует этот подход, сдвигая индекс на `-1` час:

In[77]:
```
# просто сдвигаем индексные метки
# на 1 час назад
ts.tshift(-1, freq='H')
```

Out[77]:
```
2014-07-31 23:00:00    1.0
2014-08-01 23:00:00    2.0
2014-08-02 23:00:00    2.5
2014-08-03 23:00:00    1.5
2014-08-04 23:00:00    0.5
Freq: D, dtype: float64
```

Преобразование частоты временного ряда

Частоту временного ряда можно преобразовать с помощью метода `.asfreq()` объекта, в котором записан временной ряд. В результате преобразования частоты будет создан новый объект `Series` с индексом `DatatimeIndex`. Индекс `DatetimeIndex` нового объекта начинается с первой временной метки исходной серии и последовательно изменяется с определенной частотой вплоть до последней временной метки исходной серии. Затем происходит выравнивание значений.

Чтобы продемонстрировать вышеописанную операцию, мы создадим временной ряд, состоящий из последовательно возрастающих целочисленных значений. Эти значения представляют собой часовые интервалы всех дней августа 2014 года, таким образом, наш временной ряд имеет ежечасную частоту.

In[78]:
```
# создаем серию последовательно возрастающих значений,
# представляющих собой часовые интервалы
# всех дней августа 2014 года
periods = 31 * 24
hourly = Series(np.arange(0, periods),
                pd.date_range('08-01-2014', freq="1H",
                              periods = periods))
hourly[:5]
```

```
Out[78]:
2014-08-01 00:00:00    0
2014-08-01 02:00:00    1
2014-08-01 04:00:00    2
2014-08-01 06:00:00    3
2014-08-01 08:00:00    4
Freq: 2H, dtype: int32
```

Следующий программный код преобразует ежечасную частоту временного ряда в ежесуточную с помощью метода `.asfreq('D')`:

```
In[79]:
# преобразуем ежечасную частоту
# временного ряда в ежесуточную
# многие элементы будут удалены
# из-за выравнивания
daily = hourly.asfreq('D')
daily[:5]

Out[79]:
2014-08-01     0
2014-08-02    24
2014-08-03    48
2014-08-04    72
2014-08-05    96
Freq: D, dtype: int32
```

Поскольку данные приведены в соответствие с новым ежесуточным временным рядом, полученным на основе исходного ежечасного временного ряда, копируются только те значения, которые соответствуют точному времени наступления суток.
Если мы преобразуем ежесуточную частоту обратно в ежечасную, мы увидим, что многие значения стали значениями `NaN`:

```
In[80]:
# преобразуем обратно в ежечасную частоту, в результате
# получим множество значений NaN, поскольку новый индекс
# содержит много меток, не соответствующих меткам
# исходного временного ряда
daily.asfreq('H')

Out[80]:
2014-08-01 00:00:00      0.0
2014-08-01 01:00:00      NaN
2014-08-01 02:00:00      NaN
2014-08-01 03:00:00      NaN
2014-08-01 04:00:00      NaN
                        ...
2014-08-30 20:00:00      NaN
2014-08-30 21:00:00      NaN
2014-08-30 22:00:00      NaN
2014-08-30 23:00:00      NaN
2014-08-31 00:00:00    720.0
Freq: H, Length: 721, dtype: float64
```

Новый индекс состоит из объектов `Timestamp` – часовых интервалов, поэтому только временные метки, соответствующие точному времени наступления суток (00:00:00) выравниваются в соответствии с ежесуточным временным рядом, что приводит к 670 значениям `NaN`.
Этот результат можно изменить с помощью параметра `method` метода `.asfreq()`. Данный параметр можно использовать для прямого заполнения, обратного заполнения или обычного заполнения значений

NaN. Метод ffill использует для заполнения последнее известное значение (метод pad делает то же самое):

```
In[81]:
# выполняем прямое заполнение значений
daily.asfreq('H', method='ffill')

Out[81]:
2014-08-01 00:00:00      0
2014-08-01 01:00:00      0
2014-08-01 02:00:00      0
2014-08-01 03:00:00      0
2014-08-01 04:00:00      0
                      ...
2014-08-30 20:00:00    696
2014-08-30 21:00:00    696
2014-08-30 22:00:00    696
2014-08-30 23:00:00    696
2014-08-31 00:00:00    720
Freq: H, Length: 721, dtype: int32
```

Метод bfill выполняет обратное заполнение значений:

```
In[82]:
daily.asfreq('H', method='bfill')

Out[82]:
2014-08-01 00:00:00      0
2014-08-01 01:00:00     24
2014-08-01 02:00:00     24
2014-08-01 03:00:00     24
2014-08-01 04:00:00     24
                      ...
2014-08-30 20:00:00    720
2014-08-30 21:00:00    720
2014-08-30 22:00:00    720
2014-08-30 23:00:00    720
2014-08-31 00:00:00    720
Freq: H, Length: 721, dtype: int32
```

Увеличение или уменьшение шага дискретизации временного ряда

Преобразование частоты – основной способ модифицировать интервалы временного ряда. Данные нового временного ряда выравниваются по данным исходного временного ряда, что может привести к появлению множества значений NaN. Частично это можно решить с помощью методов заполнения пропусков, но применение этих методов ограничено их возможностями.

Изменение шага дискретизации временного ряда отличается тем, что оно не выполняет выравнивание в чистом виде. Значения новой серии могут использовать те же самые параметры прямого и обратного заполнения, но их можно указать и с помощью других способов, предлагаемых библиотекой pandas или с помощью собственных функций. Чтобы продемонстрировать изменение шага дискретизации, мы создадим временной ряд, который представляет собой случайное блуждание за 5-дневный период:

```
In[83]:
# сгенерируем случайное блуждание за 5 дней
# по 1-секундным интервалам
# получим большое количество значений
```

```
count = 24 * 60 * 60 * 5
# создаем серию значений
np.random.seed(123456)
values = np.random.randn(count)
ws = pd.Series(values)
# вычисляем накопленную сумму
walk = ws.cumsum()
# создаем индекс
walk.index = pd.date_range('2014-08-01', periods=count, freq="S")
walk
```

```
Out[83]:
2014-08-01 00:00:00      0.469112
2014-08-01 00:00:01      0.186249
2014-08-01 00:00:02     -1.322810
2014-08-01 00:00:03     -2.458442
2014-08-01 00:00:04     -1.246330
                             ...
2014-08-05 23:59:55    456.529763
2014-08-05 23:59:56    456.052131
2014-08-05 23:59:57    455.202981
2014-08-05 23:59:58    454.947362
2014-08-05 23:59:59    456.191430
Freq: S, Length: 432000, dtype: float64
```

Изменение шага дискретизации (передискретизация) в библиотеке pandas выполняется с помощью метода `.resample()` и передачи в него новой частоты. Чтобы продемонстрировать его использование, следующий программный код уменьшит шаг дискретизации временного ряда, преобразовав односекундные интервалы в одноминутные. В данном случае речь идет о понижающей передискретизации, поскольку временной ряд теперь имеет более низкую частоту и состоит из меньшего количества значений:

```
In[84]:
# уменьшаем шаг дискретизации, преобразуем
# односекундные интервалы в одноминутные
walk.resample("1Min").mean()
```

```
Out[84]:
2014-08-01 00:00:00     -8.718220
2014-08-01 00:01:00    -15.239213
2014-08-01 00:02:00     -9.179315
2014-08-01 00:03:00     -8.338307
2014-08-01 00:04:00     -8.129554
                             ...
2014-08-05 23:55:00    453.773467
2014-08-05 23:56:00    450.857039
2014-08-05 23:57:00    450.078149
2014-08-05 23:58:00    444.637806
2014-08-05 23:59:00    453.837417
Freq: T, Length: 7200, dtype: float64
```

Обратите внимание, что первое наблюдение имеет значение -8.718220, в то время как в исходных данных первое наблюдение имеет значение 0,469112. Это связано с тем, что в ходе передискретизации не происходит копирования данных путем выравнивания. Передискретизация фактически делит данные на интервалы на основе новых периодов, а затем применяет конкретную операцию к данным каждого интервала, в нашем случае вычисляет среднее значение интервала. Это можно проверить с помощью следующего программного

кода, который создает срез – первую минуту случайного блуждания и вычисляет ее среднее значение:

In[85]:
```
# вычисляем среднее значение первой
# минуты блуждания
walk['2014-08-01 00:00'].mean()
```

Out[85]:
```
-8.718220052832644
```

Поскольку при понижающей передискретизации существующие данные помещаются в интервалы на основе новых периодов, часто возникает вопрос о том, какие значения должны находится на границах интервалов. Например, должен ли первый интервал предыдущего временного ряда начинаться с `2014-08-01 00:00:00` и заканчиваться в `2014-08-01 23:59:59`, или он должен начинаться в `2014-08-03 23:59:59`, а заканчиваться в `2014-08-04 00:00:00`?
По умолчанию используется первый вариант и такой интервал называют закрытым слева. Другой сценарий, исключающий левый конец интервала и включающий правый конец интервала, создает интервал, закрытый справа. Этот сценарий можно реализовать с помощью параметра `close='right'`. Следующий программный код как раз демонстрирует его и обратите на небольшую разницу в интервалах и значениях, которые мы в итоге получим:

In[86]:
```
# используем интервалы,
# закрытые справа
walk.resample("1Min", closed='right').mean()
```

Out[86]:
```
2014-07-31 23:59:00      0.469112
2014-08-01 00:00:00     -8.907477
2014-08-01 00:01:00    -15.280685
2014-08-01 00:02:00     -9.083865
2014-08-01 00:03:00     -8.285550
                           ...
2014-08-05 23:55:00    453.726168
2014-08-05 23:56:00    450.849039
2014-08-05 23:57:00    450.039159
2014-08-05 23:58:00    444.631719
2014-08-05 23:59:00    453.955377
Freq: T, Length: 7201, dtype: float64
```

В реальной практике решение о том, как закрывать интервал – слева или справа зависит от вас и ваших данных, но библиотека pandas предлагает различные варианты вычисления значений в интервалах.
Вычисление среднего для каждого интервала – это только один вариант. Программный код ниже приводит первое значение в каждом интервале:

In[87]:
```
# преобразовываем в 1-минутные интервалы
walk.resample("1Min").first()
```

Out[87]:
```
2014-08-01 00:00:00      0.469112
2014-08-01 00:01:00    -10.886314
```

```
2014-08-01 00:02:00    -13.374656
2014-08-01 00:03:00     -7.647693
2014-08-01 00:04:00     -4.482292
                            ...
2014-08-05 23:55:00    452.900335
2014-08-05 23:56:00    450.062374
2014-08-05 23:57:00    449.582419
2014-08-05 23:58:00    447.243014
2014-08-05 23:59:00    446.877810
Freq: T, Length: 7200, dtype: float64
```

Чтобы продемонстрировать повышающую передискретизацию, мы выполним повторное преобразование в одноминутные интервалы, а затем в односекундные интервалы:

In[88]:
```
# преобразовываем в 1-минутные интервалы,
# а затем в 1-секундные
bymin = walk.resample("1Min").mean()
bymin.resample('S').mean()
```

Out[88]:
```
2014-08-01 00:00:00     -8.718220
2014-08-01 00:00:01           NaN
2014-08-01 00:00:02           NaN
2014-08-01 00:00:03           NaN
2014-08-01 00:00:04           NaN
                             ...
2014-08-05 23:58:56           NaN
2014-08-05 23:58:57           NaN
2014-08-05 23:58:58           NaN
2014-08-05 23:58:59           NaN
2014-08-05 23:59:00    453.837417
Freq: S, Length: 431941, dtype: float64
```

Процедура повышающей дискретизации создала индексные метки для ежесекундного временного ряда, но при этом по умолчанию многие значения стали значениями `NaN`. Этот результат, получаемый по умолчанию, можно изменить с помощью параметра `fill_method`. Мы познакомились с ним, когда изменяли частоту и использовали прямое и обратное заполнение пропусков. Эти методы также можно использовать после применения передискретизации. Следующий программный код выполняет обратное заполнение:

In[89]:
```
# преобразуем в 1-секундные интервалы,
# используя обратное заполнение
bymin.resample("S").bfill()
```

```
Out[89]:
2014-08-01 00:00:00     -8.718220
2014-08-01 00:00:01    -15.239213
2014-08-01 00:00:02    -15.239213
2014-08-01 00:00:03    -15.239213
2014-08-01 00:00:04    -15.239213
                           ...
2014-08-05 23:58:56    453.837417
2014-08-05 23:58:57    453.837417
2014-08-05 23:58:58    453.837417
2014-08-05 23:58:59    453.837417
2014-08-05 23:59:00    453.837417
Freq: S, Length: 431941, dtype: float64
```

Кроме того, можно выполнить интерполяцию пропущенных значений с помощью метода `.interpolate()`. Он выполнит линейную интерполяцию всех значений `NaN`, полученных в результате передискретизации:

```
In[90]:
# демонстрируем интерполяцию значений NaN
interpolated = bymin.resample("S").interpolate()
interpolated

Out[90]:
2014-08-01 00:00:00     -8.718220
2014-08-01 00:00:01     -8.826903
2014-08-01 00:00:02     -8.935586
2014-08-01 00:00:03     -9.044270
2014-08-01 00:00:04     -9.152953
                           ...
2014-08-05 23:58:56    453.224110
2014-08-05 23:58:57    453.377437
2014-08-05 23:58:58    453.530764
2014-08-05 23:58:59    453.684090
2014-08-05 23:59:00    453.837417
Freq: S, Length: 431941, dtype: float64
```

Библиотека pandas предлагает еще один удобный способ передискретизации временного ряда под названием open high low close. Он часто используется в финансовых приложениях, когда временной ряд агрегируют, вычисляя четыре значения определенного финансового показателя (например, цены акции) для каждого интервала: первое (открытие – open), максимальное (high), минимальное (low) и последнее (закрытие – close). Данный способ реализован с помощью метода `.ohlc()`. Следующий программный код берет наш ежесекундный временной ряд и каждый час вычисляет четыре значения:

```
In[91]:
# демонстрируем передискретизацию ohlc
ohlc = walk.resample("H").ohlc()
ohlc
```

```
Out[91]:
                            open        high         low  \
2014-08-01 00:00:00    0.469112    0.469112  -67.873166
2014-08-01 01:00:00   -3.374321   23.793007  -56.585154
2014-08-01 02:00:00  -54.276885    5.232441  -87.809456
2014-08-01 03:00:00    0.260576   17.124638  -65.820652
2014-08-01 04:00:00  -38.436581    3.537231 -109.805294
...                         ...         ...         ...
2014-08-05 19:00:00  437.652077  550.408942  430.549178
2014-08-05 20:00:00  496.539759  510.371745  456.365565
2014-08-05 21:00:00  476.025498  506.952877  425.472410
2014-08-05 22:00:00  497.941355  506.599652  411.119919
2014-08-05 23:00:00  443.017962  489.083657  426.062444

                           close
2014-08-01 00:00:00   -2.922520
2014-08-01 01:00:00  -55.101543
2014-08-01 02:00:00    1.913276
2014-08-01 03:00:00  -38.530620
2014-08-01 04:00:00  -61.014553
...                         ...
2014-08-05 19:00:00  494.471788
2014-08-05 20:00:00  476.505765
2014-08-05 21:00:00  498.547578
2014-08-05 22:00:00  443.925832
2014-08-05 23:00:00  456.191430

[120 rows x 4 columns]
```

Применение к временному ряду операций на основе скользящего окна

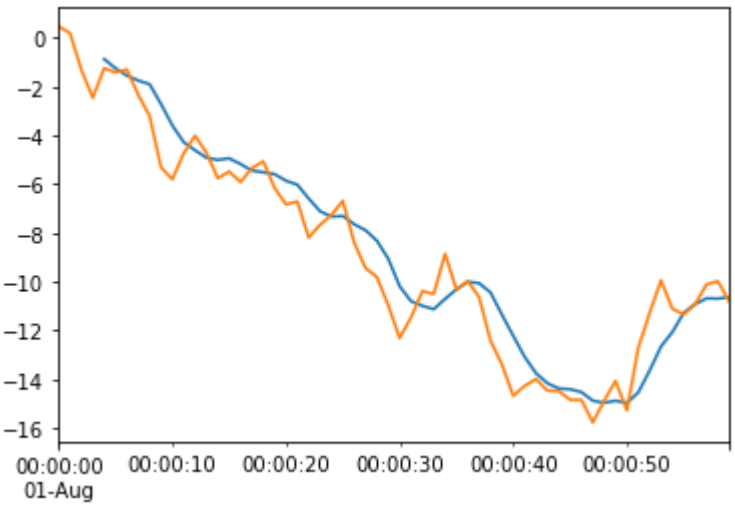
Библиотека pandas предлагает ряд функций, позволяющих вычислять статистики, меняющиеся со временем (также известные как скользящие статистики). В рамках подхода «скользящее окно» библиотека pandas вычисляет статистику по «окну» данных, представляющему определенный период времени. Затем окно смещается на определенный интервал времени и статистика постоянно вычисляется для каждого нового окна до тех пор, пока окно охватывает даты временного ряда.

Библиотека pandas непосредственно поддерживает скользящие оконные функции, предлагая метод `.rolling()` объекта `Series` и объекта `DataFrame`. В следующей таблице приводится ряд таких функций:

Функция	Описание
`.rolling().mean()`	Среднее значение в окне
`.rolling().std()`	Стандартное отклонение в окне
`.rolling().var()`	Дисперсия в окне
`.rolling().min()`	Минимальное значение в окне
`.rolling().max()`	Максимальное значение в окне
`.rolling().cov()`	Коэффициент ковариации в окне
`.rolling().quantile()`	Оценка для заданного процентиля/выборочного квантиля в окне
`.rolling().corr()`	Коэффициент корреляции в окне
`.rolling().median()`	Медиана в окне
`.rolling().sum()`	Сумма в окне
`.rolling().apply()`	Применение пользовательской функции к значениям окна
`.rolling().count()`	Количество непропущенных значений в окне
`.rolling().skew()`	Коэффициент асимметрии в окне
`.rolling().kurt()`	Коэффициент эксцесса в окне

На практике скользящее среднее используется для сглаживания краткосрочных колебаний и более четкого выделения долгосрочных тенденции в ряде данных и активно применяется в анализе временных рядов. Для иллюстрации мы вычислим скользящее среднее с шириной окна 5 по первой минуте случайного блуждания, созданного ранее в этой главе.

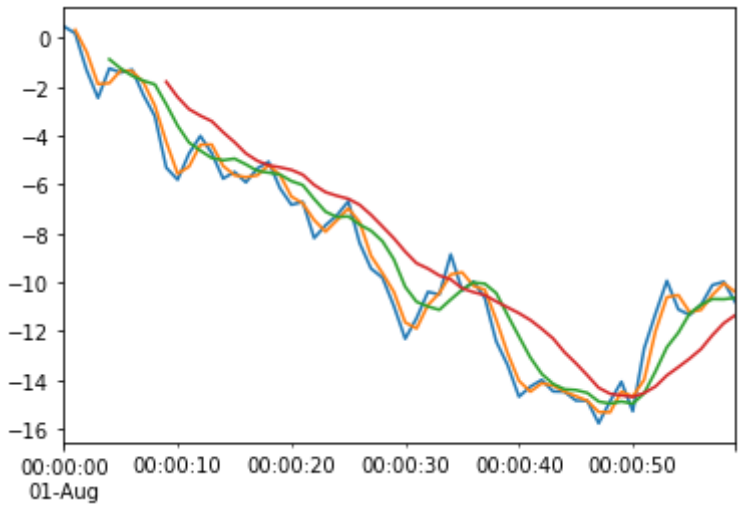
In[92]:
```
# извлекаем данные по одному 1-минутному интервалу
first_minute = walk['2014-08-01 00:00']
# вычисляем скользящее среднее с шириной окна 5 периодов
means = first_minute.rolling(window=5, center=False).mean()
# сравним средние с исходными данными
means.plot()
first_minute.plot();
```



Построение диаграмм будет рассмотрено более подробно в главе 14 «Визуализация».

Видно, что `.rolling().mean()` дает более сглаженное представление исходных данных. Более широкое окно дает меньший разброс значений, а более узкие окна приведут к более высокому разбросу значений (и так до тех пор, пока размер окна не станет равным 1, который будет идентичен исходному временному ряду).
Следующий программный код визуализирует скользящие средние с шириной окна 2, 5 и 10 и исходный временной ряд:

In[93]:
```
# демонстрируем разницу между скользящими
# окнами шириной 2, 5 и 10
h1w = walk['2014-08-01 00:00']
means2 = h1w.rolling(window=2, center=False).mean()
means5 = h1w.rolling(window=5, center=False).mean()
means10 = h1w.rolling(window=10, center=False).mean()
h1w.plot()
means2.plot()
means5.plot()
means10.plot();
```

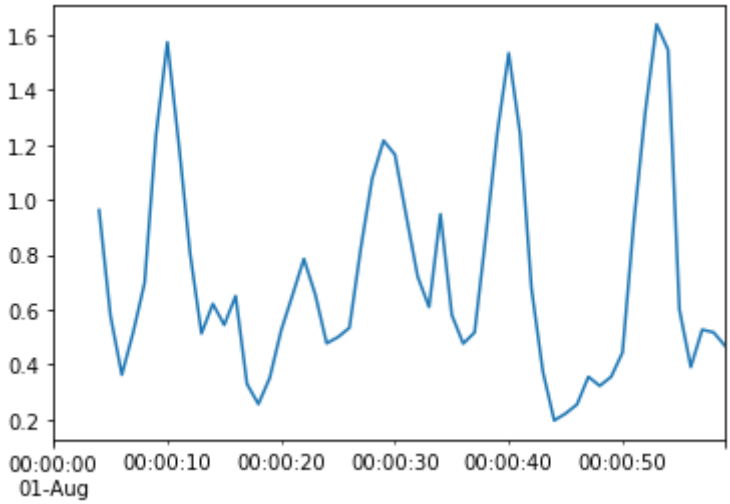


Обратите внимание, что чем больше окно, тем больше данных отсутствует в начале кривой. Для первоначального вычисления метрики окну размера *n* требуется *n* точек данных и, следовательно, получаем разрыв в начале графика.

С помощью метода `.rolling().apply()` к значениям, попавшим в окно, можно применить любую пользовательскую функцию. Этой функции можно передать массив значений, попавших в окно, и будет возвращено одно значение. Затем библиотека pandas объединяет результаты, полученные по каждому окну, во временной ряд.

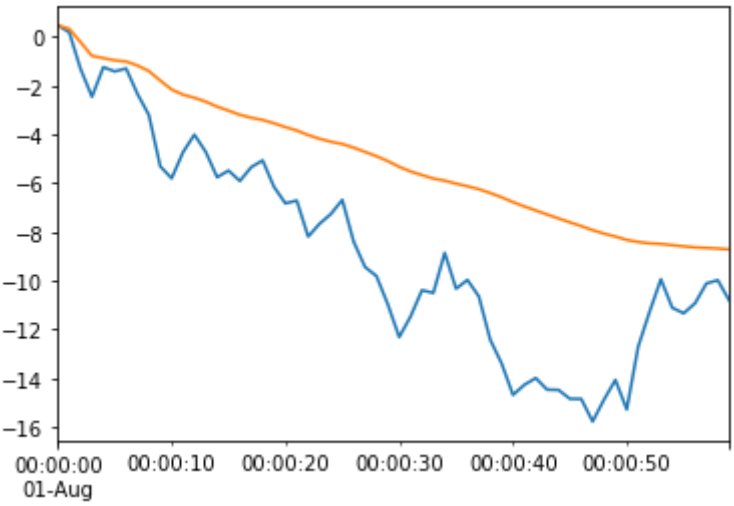
Для иллюстрации следующий программный код вычисляет среднее абсолютное отклонение, которое дает представление о том, насколько сильно значения выборки отличаются от общего среднего значения:

In[94]:
```
# вычисляем среднее абсолютное значение
# для окна с 5 интервалами
mean_abs_dev = lambda x: np.fabs(x - x.mean()).mean()
means = h1w.rolling(window=5, center=False).apply(mean_abs_dev, raw=True)
means.plot();
```



Среднее значение с расширяющимся окном можно вычислить с помощью небольшой модификации функции `pd.rolling_mean`, которая многократно вычисляет среднее значение, всегда начиная с первого значения временного ряда и в каждой итерации увеличивая размер окна на единицу. Среднее с расширяющимся окном будет более стабильным (менее чувствительным) показателем, чем обычное скользящее среднее, потому что чем больше размер окна, тем меньше влияние следующего значения:

In[95]:
```
# вычисляем среднее с расширяющимся окном
h1w.plot()
expanding = h1w.expanding(min_periods=1).mean()
expanding.plot();
```



Выводы

В этой главе мы рассмотрели различные способы представления событий, которые происходят в определенные моменты времени, а также методы обработки значений, изменяющихся с течением времени. Мы изучили объекты, использующиеся для представления даты и времени, объекты, представляющие интервалы и периоды, выполнили наиболее распространенные операции с временными рядами, включая преобразование частоты, передискретизацию шага и работу со скользящими окнами.

В оставшихся двух главах механизмы работы библиотеки pandas мы оставим позади и большее внимание уделим визуализации данных, а также рассмотрим применение библиотеки pandas для анализа финансовых данных.

ГЛАВА 14 ВИЗУАЛИЗАЦИЯ

Одним из наиболее важных этапов анализа данных является создание визуализаций для быстрой интерпретации содержательного смысла данных. Визуализация данных эффективна, поскольку мы, люди, являемся существами, активно использующими зрение. Мы эволюционировали и научились понимать смысл визуальной информации, когда данные представлены так, что наш мозг может интерпретировать их почти сразу, как только нервные импульсы, исходящие от сетчатки глаза, поступят в мозг.

За прошедшие годы были проведены значительные исследования, которые привели к появлению различных эффективных способов визуализации, позволяющих описать те или иные конкретные паттерны данных. Эти способы реализованы в специальных библиотеках по визуализации, а библиотека pandas как раз предназначена для их использования, что существенно упрощает визуализацию данных.

В этой главе будут рассмотрены некоторые техники визуализации, в первую очередь внимание будет уделено библиотеке matplotlib и наиболее распространенным графикам. Обзор возможностей визуализации будет состоять из трех этапов. В рамках первого этапа мы познакомимся с общими принципами создания графиков с помощью pandas, уделив особое внимание построению графиков временных рядов. Мы рассмотрим способы разметки осей и создания легенд, цветов, стилей линий и маркеров.

Второй этап будет посвящен различным видам визуализации данных, которые наиболее часто используются в pandas и анализе данных, мы рассмотрим такие темы, как:

- Демонстрация относительных различий с помощью столбиковых диаграмм
- Визуализация распределений данных с помощью гистограмм
- Визуализация распределений категориальных данных с помощью ящичных диаграмм
- Отображение накопленных итогов с помощью площадных диаграмм
- Визуализация взаимосвязи между двумя переменными с помощью диаграммы рассеяния
- Визуализация оценок распределения с помощью графика ядерной оценки плотности
- Визуализация корреляций между несколькими переменными с помощью матрицы диаграмм рассеяния
- Отображение взаимосвязей между несколькими переменными с помощью тепловых карт

На последнем этапе мы разберем создание сложных диаграмм путем разбиения графиков на подграфики, чтобы отобразить несколько диаграмм в рамках одного рисунка. Это позволит зрителю сразу визуально сопоставить друг с другом различные наборы данных.

Настройка библиотеки pandas

Мы начнем рассмотрение примеров в этой главе, задав нижеприведенные инструкции для импорта библиотек и настройки вывода. Однако в этой главе есть небольшая деталь, мы задаем стартовое значение для генератора случайных чисел:

```
In[1]:
# импортируем библиотеки numpy и pandas
import numpy as np
import pandas as pd

# импортируем библиотеку datetime для работы с датами
import datetime
from datetime import datetime, date

# Задаем некоторые опции библиотеки pandas, которые
# настраивают вывод
pd.set_option('display.notebook_repr_html', False)
pd.set_option('display.max_columns', 8)
pd.set_option('display.max_rows', 10)
pd.set_option('display.width', 60)

# импортируем библиотеку matplotlib для построения графиков
import matplotlib.pyplot as plt
%matplotlib inline

# задаем стартовое значение для генератора
# случайных чисел
seedval = 111111
```

Основные инструменты визуализации

Сама по себе библиотека pandas не выполняет визуализацию данных. Для выполнения этой задачи библиотека pandas предлагает тесную интеграцию с другими надежными библиотеками визуализации, которые являются частью экосистемы Python. Наиболее часто используемой библиотекой визуализации является библиотека `matplotlib`. Поэтому в примерах, приведенных в этой главе, будет использоваться `matplotlib`, однако есть и другие возможные библиотеки, с которыми вы можете поработать самостоятельно. Две из них заслуживают упоминания.

Seaborn – еще одна библиотека визуализации Python, которая также использует в своей основе библиотеку `matplotlib`. Она предлагает высокоуровневый интерфейс для построения привлекательных статистических графиков. Seaborn имеет встроенную поддержку структур данных NumPy и pandas. Целью Seaborn является создание графиков `matplotlib`, более удобных для интерпретации. Для получения более подробной информации о библиотеке seaborn посетите сайт http://seaborn.pydata.org/index.html.

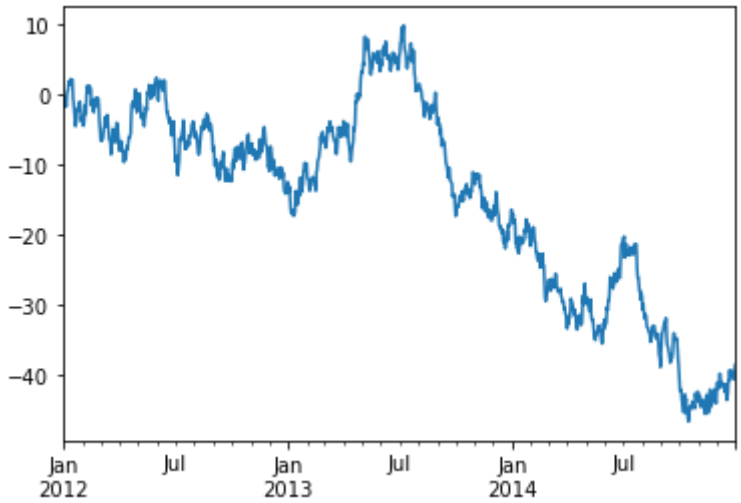
Хотя и seaborn и `matplotlib` являются замечательными инструментами отрисовки данных, обоим недостает интерактивности. И хотя в обоих инструментах графики могут отображаться в браузере с помощью таких инструментов, как Jupyter, сами визуализации не создаются с помощью DOM и не используют каких-либо возможностей браузера.

Чтобы упростить отрисовку данных в браузере и обеспечить гибкую интерактивность, было создано несколько библиотек для интеграции Python и pandas с `D3.js`. `D3.js` – это библиотека JavaScript для управления документами и создания высококачественных интерактивных визуализаций данных. Одна из самых популярных библиотек, использующих `D3.js` – это mpld3. К сожалению, на момент написания этой книги она не работала с Python 3.6 и поэтому не будет рассмотрена в книге. Однако регулярно проверяйте информацию о `D3.js` по адресу https://d3js.org и mpld3 по адресу http://mpld3.github.io/.

Создание графиков временных рядов

Временные ряды – один из самых часто визуализируемых типов данных. Визуализировать временной ряд в библиотеке pandas очень просто, достаточно применить метод `.plot()` объекта `DataFrame` или `Series`, в котором записан временной ряд.
Следующий программный код демонстрирует создание временного ряда, представляющего собой случайное блуждание – случайные изменения данных в дискретные моменты времени, схожие с изменением цены на акцию:

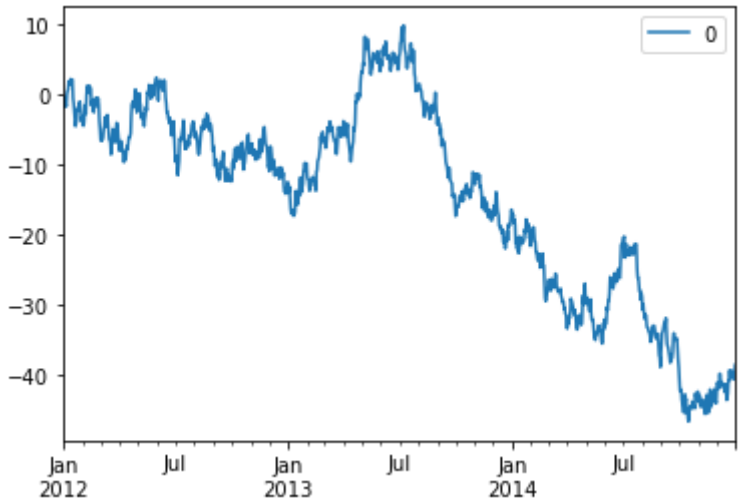
```
In[2]:
# генерируем временной ряд на
# основе случайного блуждания
np.random.seed(seedval)
s = pd.Series(np.random.randn(1096),
              index=pd.date_range('2012-01-01',
                                  '2014-12-31'))
walk_ts = s.cumsum()
# эта строка визуализирует случайное блуждание - так просто :)
walk_ts.plot();
```



Метод `.plot()` объектов pandas является функцией-оберткой вокруг функции `plot()` библиотеки `matplotlib`. Он делает построение графиков в pandas очень простой процедурой, поскольку программный код, лежащий в его основе, позволяет строить самые различные визуализации данных. Он выполняет такие операции, как отбор временного ряда, создание и маркировку осей.

В предыдущем примере метод `.plot()` определил, что объект `Series` проиндексирован по датам, поэтому ось x должна быть представлена в виде дат. Кроме того он задает цвет, который будет использоваться по умолчанию для отображения данных.
Визуализация объекта `DataFrame` с одним столбцом даст аналогичный результат, что и визуализация объекта `Series`. Следующий программный код подтверждает это, создав один и тот же график с одной небольшой разницей: теперь на график добавлена легенда. По умолчанию графики, создаваемые на основе объекта `DataFrame`, будут содержать легенду.

In[3]:
```
# Визуализация объекта DataFrame с одним столбцом даст
# аналогичный результат, что и визуализация объекта Series,
# только теперь будет еще добавлена легенда
walk_df = pd.DataFrame(walk_ts)
walk_df.plot();
```



Если объект `DataFrame` состоит из нескольких столбцов, метод `.plot()` добавит несколько элементов в легенду и подберет для каждой линии свой цвет:

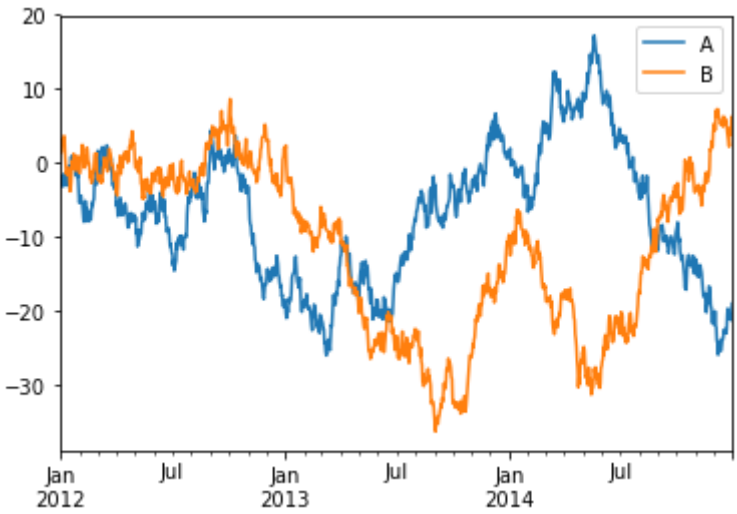
In[4]:
```
# сгенерируем два случайных блуждания, которые станут
# двумя столбцами объекта DataFrame
np.random.seed(seedval)
df = pd.DataFrame(np.random.randn(1096, 2),
                  index=walk_ts.index, columns=list('AB'))
walk_df = df.cumsum()
walk_df.head()
```

Out[4]:

```
                   A         B
2012-01-01 -1.878324  1.362367
2012-01-02 -2.804186  1.427261
2012-01-03 -3.241758  3.165368
2012-01-04 -2.750550  3.332685
2012-01-05 -1.620667  2.930017
```

In[5]:

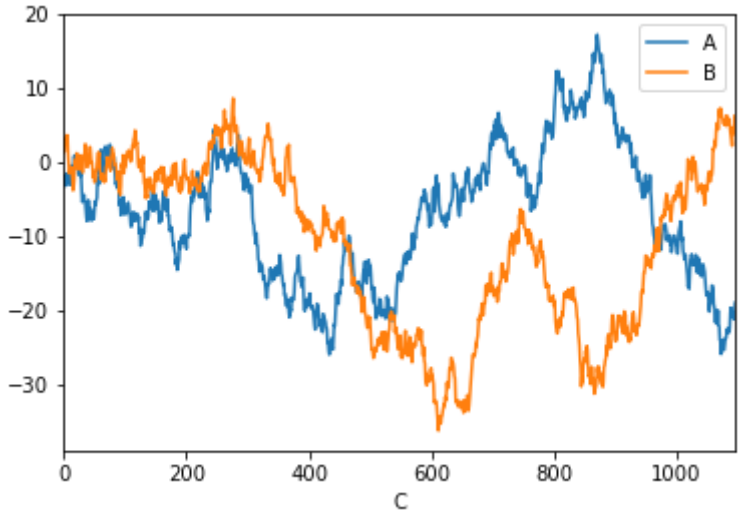
```
# визуализируем данные датафрейма, на графике каждый
# столбец будет представлен линией, а также будет
# добавлена легенда
walk_df.plot();
```



Если вы хотите в качестве меток оси x использовать значения столбца датафрейма (вместо индексных меток), используйте параметр `x`, который позволяет задать имя столбца, представляющего метки. Затем воспользуйтесь параметром `y`, чтобы указать столбцы, использующиеся в качестве данных:

In[6]:

```
# создаем копию случайного блуждания
df2 = walk_df.copy()
# добавляем столбец C, который принимает
# значения от 0 до 1096
df2['C'] = pd.Series(np.arange(0, len(df2)), index=df2.index)
# в качестве меток оси X вместо дат используем
# значения столбца 'C', получаем метки оси x в
# диапазоне от 0 до 1000
df2.plot(x='C', y=['A', 'B']);
```



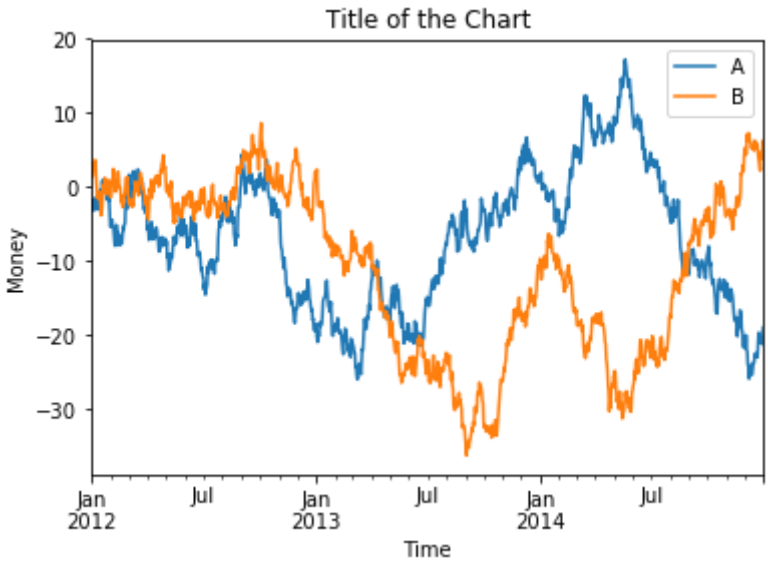
Настройка внешнего вида графика временного ряда

Встроенный метод `.plot()` предлагает множество параметров, которые вы можете использовать для изменения содержимого графика. Давайте рассмотрим некоторые часто используемые настройки графиков.

Добавление заголовка и изменение подписей осей

Заголовок графика можно задать с помощью параметра `title`. Метки осей задают с помощью функций `plt.ylabel()` и `plt.xlabel()` сразу после вызова метода `.plot()`:

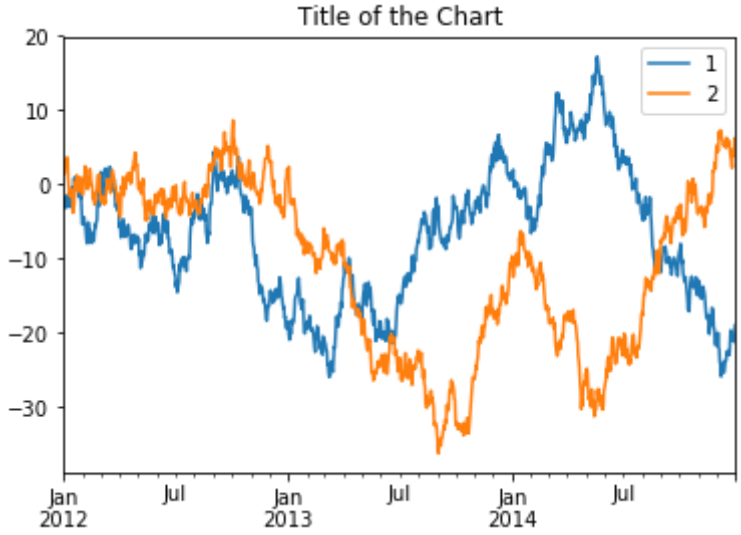
```
In[7]:
# создаем график временного ряда с заголовком и заданными
# подписями осей x и y

# заголовок задаем с помощью параметра title метода .plot()
walk_df.plot(title='Title of the Chart')
# после вызова метода .plot() явно
# задаем подписи осей x и y
plt.xlabel('Time')
plt.ylabel('Money');
```



Настройка содержимого легенды и ее расположения

Чтобы изменить текст, использующийся в легенде для идентификации каждой серии данных (по умолчанию это имя столбца датафрейма), запишите объект ax, который будет возвращен методом `.plot()`, и затем воспользуйтесь методом `.legend()` этого объекта. Данный объект является объектом `AxesSubplot` и его можно использовать для изменения различных настроек графика непосредственно перед его построением:

```
In[8]:
# изменяем элементы легенды, соответствующие
# именам столбцов датафрейма
ax = walk_df.plot(title='Title of the Chart')
# эта строка задает метки легенды
ax.legend(['1', '2']);
```



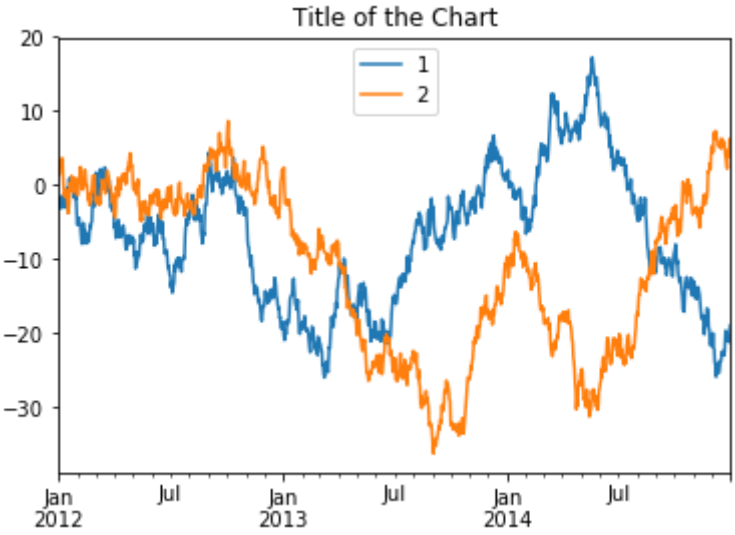
Местоположение легенды можно задать с помощью параметра `loc` метода `.legend()`. По умолчанию библиотека pandas задает местоположение `'best'`, которое дает инструкцию библиотеке `matplotlib` исследовать график и определить наилучшее местоположение легенды. Однако вы можете указать любой вариант местоположения из приведенных ниже, чтобы более точно разместить легенду (можно использовать либо строковое значение, либо числовой код):

Строковое значение	Код	Описание
`'best'`	**0**	наилучшее расположение с учетом исследования графика
`'upper right'`	**1**	расположение в верхнем правом углу
`'upper left'`	**2**	расположение в верхнем левом углу
`'lower left'`	**3**	расположение в нижнем левом углу
`'lower right'`	**4**	расположение в нижнем правом углу
`'right'`	**5**	горизонтальное выравнивание справа с вертикальным выравниванием по центру
`'center left'`	**6**	вертикальное выравнивание по центру с горизонтальным выравниванием слева
`'center right'`	**7**	вертикальное выравнивание по центру с горизонтальным выравниванием справа
`'lower center'`	**8**	вертикальное выравнивание снизу с горизонтальным выравниванием по центру
`'upper center'`	**9**	вертикальное выравнивание сверху с горизонтальным выравниванием

		по центру
'center'	**10**	вертикальное выравнивание по центру и горизонтальное выравнивание по центру

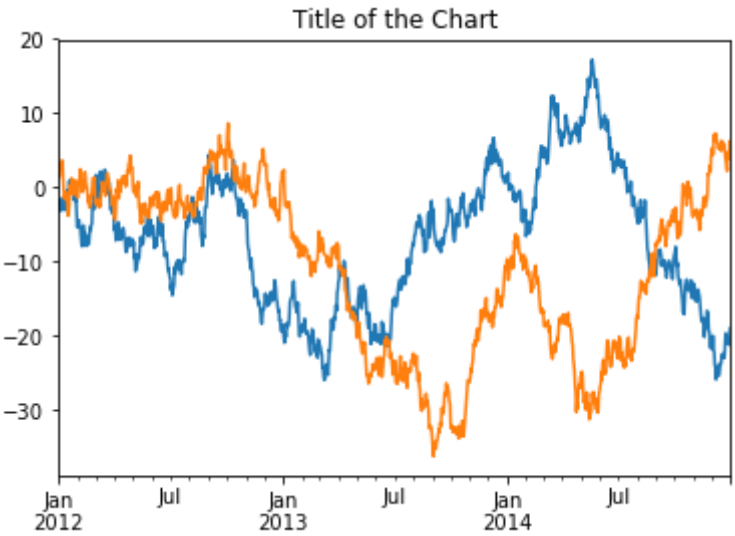
Следующий пример демонстрирует размещение легенды с вертикальным выравниванием сверху и горизонтальным выравниванием по центру:

In[9]:
```
# изменяем расположение легенды
ax = walk_df.plot(title='Title of the Chart')
# применяем к легенде вертикальное выравнивание сверху
# с горизонтальным выравниванием по центру
ax.legend(['1', '2'], loc='upper center');
```



Вывод легенды можно отключить с помощью значения параметра `legend=False`:

In[10]:
```
# отключаем легенду с помощью legend=False
walk_df.plot(title='Title of the Chart', legend=False);
```



Настройка цветов, стилей, толщины и маркеров линий

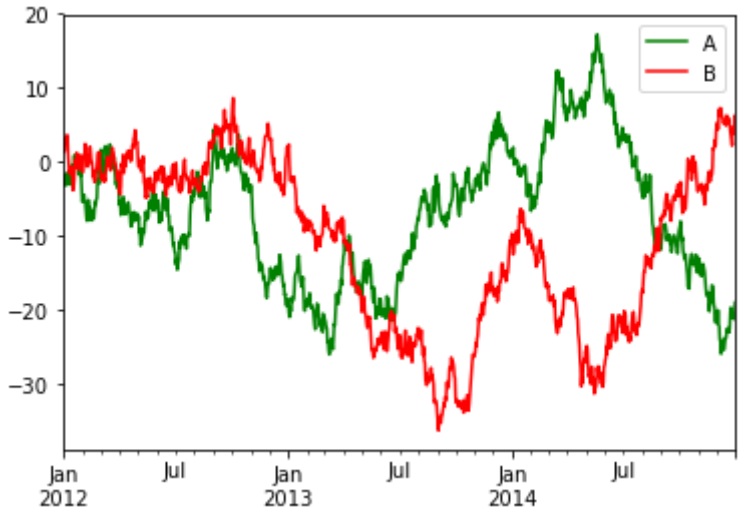
Библиотека pandas автоматически задает цвета для каждой серии при построении графика. Чтобы задать свои собственные цвета, просто передайте кодовые обозначения цветов параметру `style` функции `plot`. Библиотека pandas предлагает ряд встроенных односимвольных кодов, задающих различные цвета линии. Некоторые из них перечислены ниже:

`b`: синий
`g`: зеленый
`r`: красный
`c`: голубой
`m`: пурпурный
`y`: желтый
`k`: черный
`w`: белый

Кроме того, можно задать цвет с помощью шестнадцатеричного RGB-кода в формате **#RRGGBB**. Следующий пример демонстрирует настройку цвета для первой серии (задаем зеленый цвет с помощью односимвольного кода) и второй серии (задаем красный цвет с помощью шестнадцатеричного RGB-кода):

In[11]:

```
# меняем цвета линий графика, используя символьный
# код для первой линии и шестнадцатеричный
# RGB-код для второй
walk_df.plot(style=['g', '#FF0000']);
```

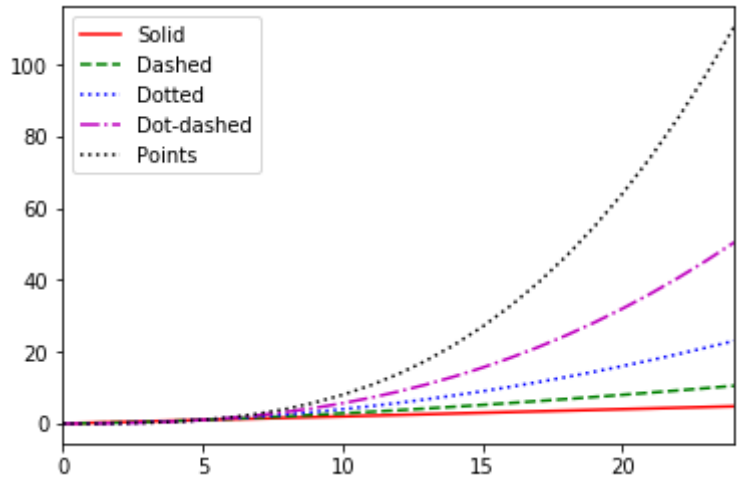


Стили линий можно задать с помощью кодов стиля линии. Их можно использовать в сочетании с кодами цветов, задав сразу после кодового обозначения цвета. Вот несколько примеров полезных кодовых обозначений, задающих различные стили линий:

`'-'` = сплошная линия;
`'--'` = пунктирная линия;
`':'` = точечная линия;
`'-.'` = точечно-пунктирная;
`'. '` = точки.

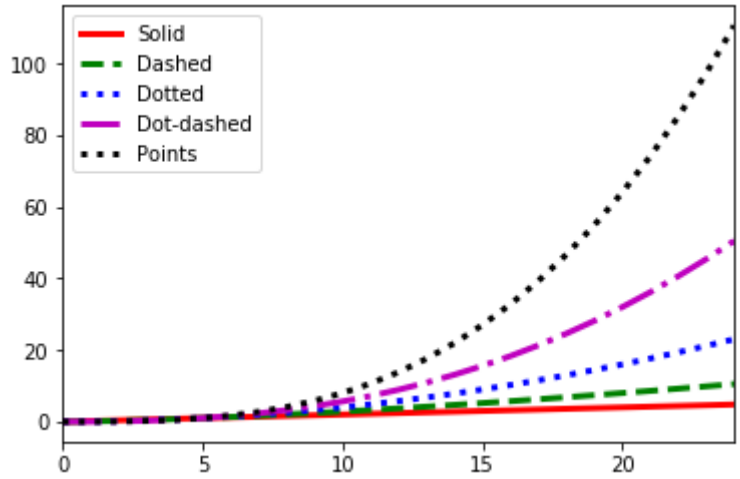
Следующий график демонстрирует использование пяти стилей, визуализируя пять серий данных, каждая из которых представлена одним из этих стилей:

In[12]:
```
# демонстрируем различные стили линий
t = np.arange(0., 5., 0.2)
legend_labels = ['Solid', 'Dashed', 'Dotted',
                 'Dot-dashed', 'Points']
line_style = pd.DataFrame({0 : t,
                           1 : t**1.5,
                           2 : t**2.0,
                           3 : t**2.5,
                           4 : t**3.0})
# создаем график, задав цвет и стиль каждой линии
ax = line_style.plot(style=['r-', 'g--', 'b:', 'm-.', 'k:'])
# задаем легенду
ax.legend(legend_labels, loc='upper left');
```



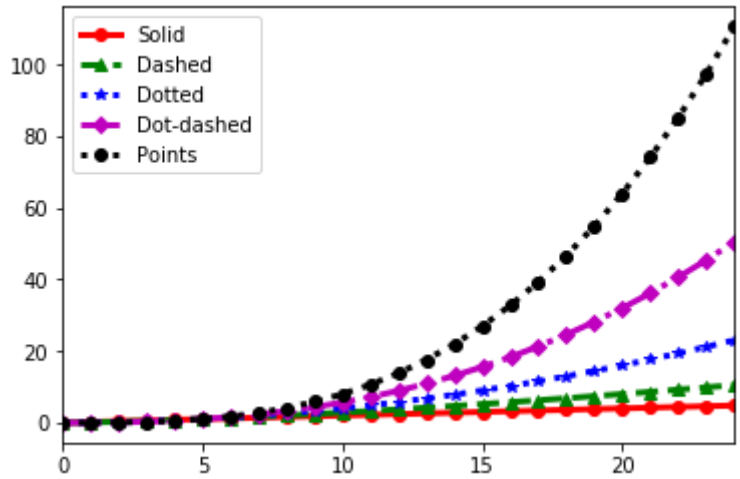
Толщину линии можно задать с помощью параметра `lw`. Можно настроить толщину сразу нескольких строк, передав этому параметру список значений ширины или одно значение ширины, которое будет применено ко всем линиям. Следующий программный код создает график с шириной линий 3, немного выделив линии:

In[13]:
```
# заново строим график, задав стиль и цвет каждой
# линии и толщину 3 для всех линий
ax = line_style.plot(style=['r-', 'g--', 'b:', 'm-.', 'k:'], lw=3)
ax.legend(legend_labels, loc='upper left');
```



Маркеры линий можно указать с помощью специальных обозначений в программном коде, задающим цвет и стиль линий. Существует немало типов маркеров и вы можете ознакомиться с ними по адресу http://matplotlib.org/api/markers_api.html. Мы рассмотрим пять типов маркеров на нижеприведенном графике, присвоим каждой серии свой маркер: кружки, звездочки, треугольники, ромбики и точки:

In[14]:
```
# рисуем заново, добавив маркеры линий
ax = line_style.plot(style=['r-o', 'g--^', 'b:*',
                            'm-.D', 'k:o'], lw=3)
ax.legend(legend_labels, loc='upper left');
```



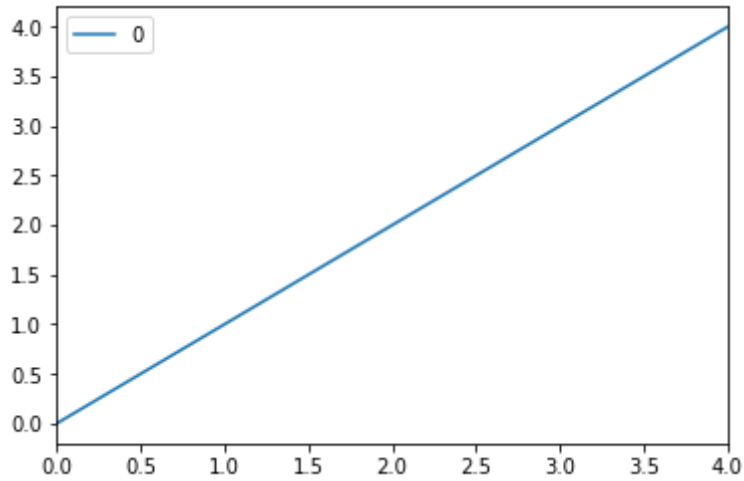
Настройка диапазона делений и меток делений шкалы

Диапазон делений шкалы и метки делений можно настроить с помощью различных функций. Следующий программный код содержит несколько примеров настройки делений шкалы.

Значения, которые библиотека pandas будет использовать в качестве меток делений, можно найти с помощью функции `plt.xticks()`. Эта функция возвращает два значения: значения для каждого деления и объекты, представляющие фактические метки делений:

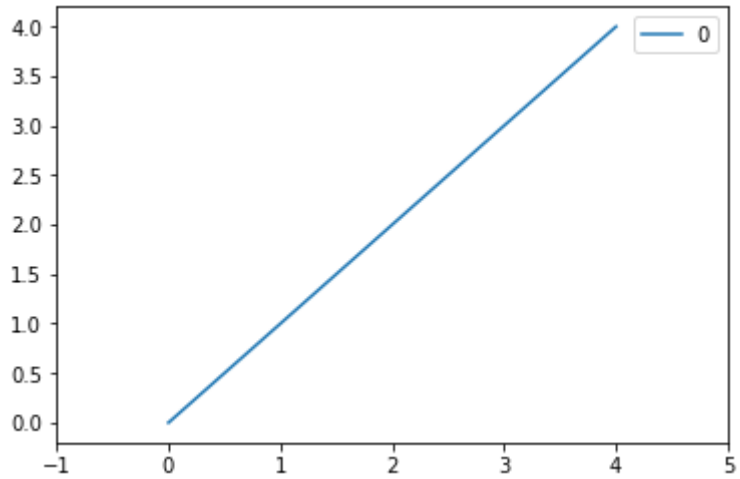
In[15]:
```
# строим простой график, чтобы продемонстрировать,
# как извлечь информацию о делениях шкалы
ticks_data = pd.DataFrame(np.arange(0,5))
ticks_data.plot()
ticks, labels = plt.xticks()
ticks
```

```
array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ])
```



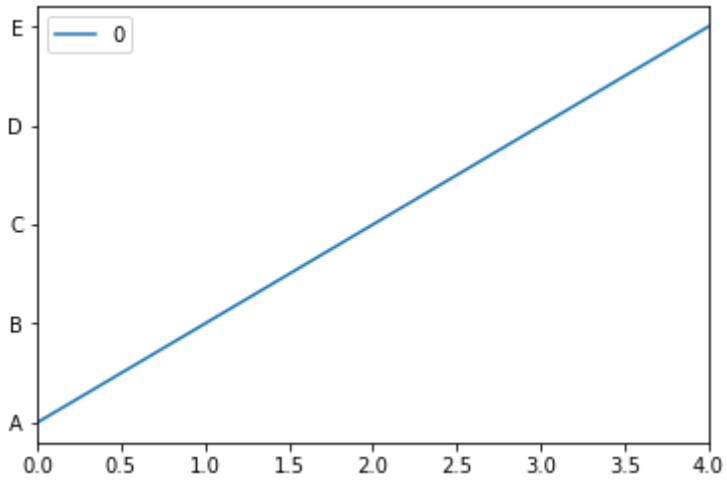
Массив делений содержит значения делений – значения фактических данных по оси *x*. В данном случае библиотека pandas решила, что диапазон от 0 до 4 (минимальное и максимальное значения соответственно) и интервал 0,5 будут наиболее подходящими.
Если вы хотите использовать другой диапазон делений, передайте значения в виде списка в `plt.xticks()`. Следующий программный код демонстрирует это, используя четные целые числа в диапазоне от `-1` до `5`. Этот набор значений изменяет как диапазон делений оси, а также выводит деления только для целочисленных значений:

In[16]:
```
# заново задаем диапазон оси x от -1 от 5 и выводим
# деления только для целочисленных значений
ticks_data = pd.DataFrame(np.arange(0,5))
ticks_data.plot()
plt.xticks(np.arange(-1, 6));
```



Метки делений можно задать, передав их в качестве второго параметра. Это показано в следующем примере: мы меняем метки оси *y* на последовательные алфавитные символы:

In[17]:
```
# переименовываем метки делений оси y
# в метки A, B, C, D и E
ticks_data = pd.DataFrame(np.arange(0,5))
ticks_data.plot()
plt.yticks(np.arange(0, 5), list("ABCDE"));
```



Форматирование меток делений, содержащих даты, с помощью форматтеров и локаторов

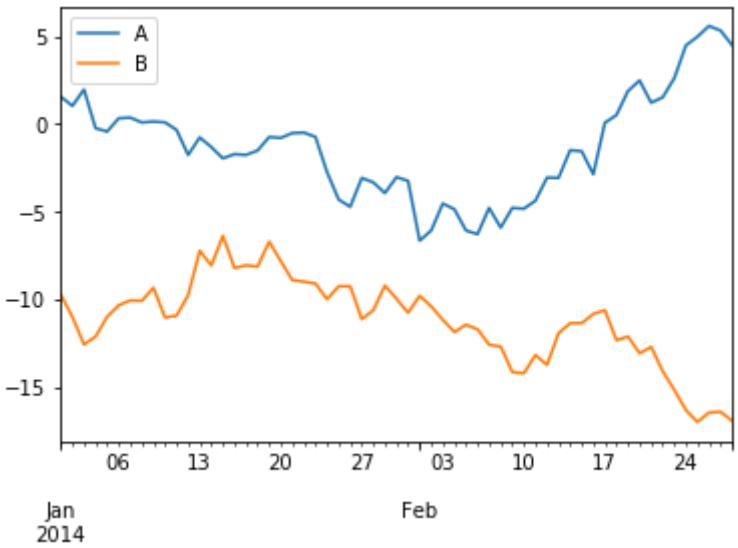
Форматирование меток осей, содержащих даты и время, выполняется с помощью **локаторов (locators)** и **форматтеров (formatters)**. Локаторы задают расположение делений, а форматтеры задают формат меток, выводимых под делениями.

Библиотека `matplotlib` в рамках модуля `matplotlib.dates` предлагает несколько классов для упрощения работы с такими осями. Специальные локаторы `MinuteLocator`, `HourLocator`, `DayLocator`, `WeekdayLocator`, `MonthLocator` и `YearLocator` задают расположение делений для каждого типа даты. Класс `DateFormatter` можно использовать для преобразования объектов, содержащих даты, в метки соответствующей оси.

По умолчанию используется локатор `AutoDateLocator` и форматтер `AutoDateFormatter`. Их можно легко изменить, выбрав другой локатор или форматтер.

Для иллюстрации давайте извлечем поднабор данных, полученный нами ранее на основе случайного блуждания. Он представляет собой данные, взятые лишь за период с января по февраль 2014 года. Визуализация этих данных дает нам следующий результат:

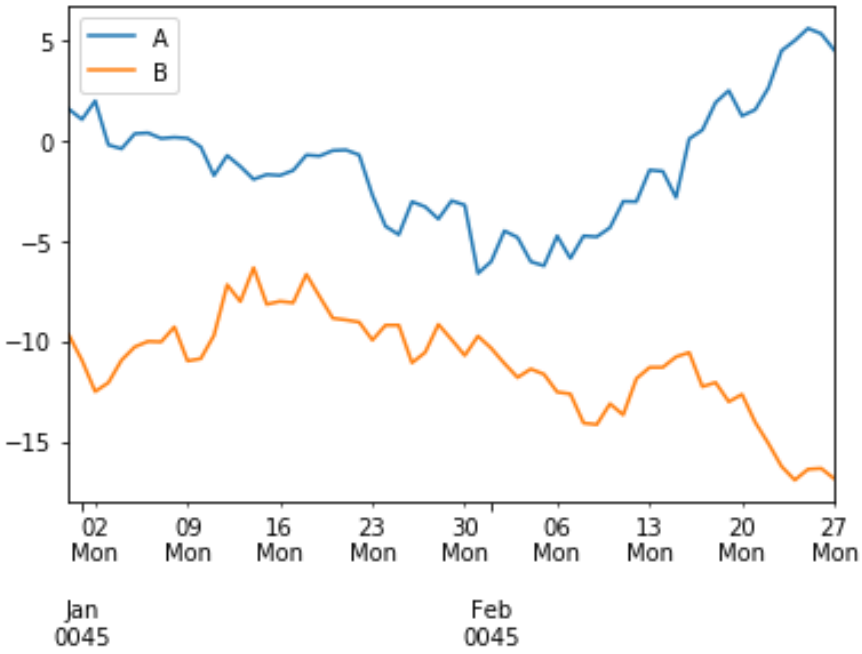
In[18]:
```
# визуализируем данные за январь-февраль 2014
# на основе случайного блуждания
walk_df.loc['2014-01':'2014-02'].plot();
```



Метки оси *x* содержат два временных ряда: основной и промежуточный. Метки промежуточных делений содержат день месяца, а метки основных делений содержат год и название месяца (год выводится только для первого месяца). Мы можем воспользоваться локаторами и форматтерами, чтобы настроить метки основных и промежуточных делений.

Мы изменим метки промежуточных делений так, чтобы они представляли собой понедельники каждой недели, а также содержали дату и название дня недели (на данный момент диаграмма использует еженедельную частоту и даты приводятся для пятницу каждой недели, при этом название дня недели не выводится). Метки основных делений будут представлять собой месяцы, а также включать название месяца и год:

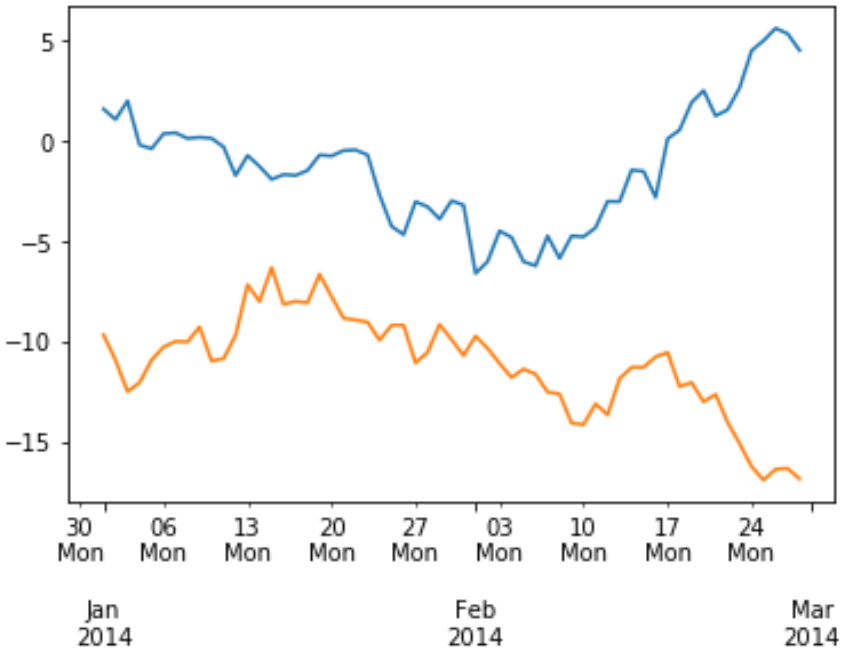
In[19]:
```
# импортируем необходимые локаторы и форматтеры
from matplotlib.dates import WeekdayLocator, \
DateFormatter, MonthLocator

# визуализируем данные за январь-февраль 2014
ax = walk_df.loc['2014-01':'2014-02'].plot()

# настраиваем метки промежуточных делений
weekday_locator = WeekdayLocator(byweekday=(0), interval=1)
ax.xaxis.set_minor_locator(weekday_locator)
ax.xaxis.set_minor_formatter(DateFormatter("%d\n%a"))

# настраиваем метки основных делений
ax.xaxis.set_major_locator(MonthLocator())
ax.xaxis.set_major_formatter(DateFormatter('\n\n\n%b\n%Y'))
```



Мы почти получили нужный нам результат. Однако обратите внимание, что год выводится как `0045`. Чтобы построить график с самостоятельно настраиваемыми метками дат, необходимо воздержаться от применения метода библиотеки pandas `.plot()` и полностью перейти к непосредственному использованию библиотеки `matplotlib`. К счастью, это не слишком сложно. В примере, приведенном ниже, мы немного изменили код и в итоге график показывает именно то, что мы хотели, и в нужном нам формате:

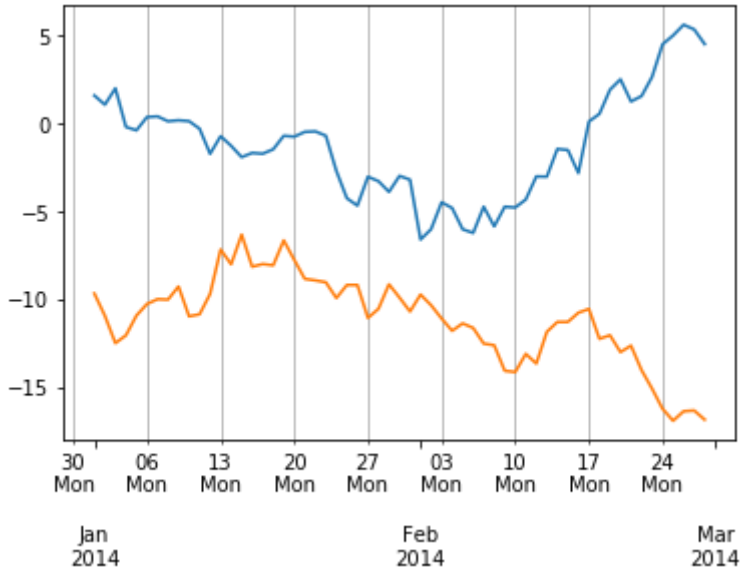
In[20]:

```
# создаем копию данных за январь-февраль 2014
walk_subset = walk_df['2014-01':'2014-02']


# строим график
fig, ax = plt.subplots()

# информируем matplotlib, что используем даты
# обратите внимание на то, что нам нужно преобразовать
# индекс DatetimeIndex в однородный многомерный массив Numpy,
# состоящий из объектов datetime.datetime
ax.plot_date(walk_subset.index.to_pydatetime(), walk_subset, '-')

# настраиваем метки промежуточных делений
weekday_locator = WeekdayLocator(byweekday=(0), interval=1)
ax.xaxis.set_minor_locator(weekday_locator)
ax.xaxis.set_minor_formatter(DateFormatter('%d\n%a'))

# настраиваем метки основных делений
ax.xaxis.set_major_locator(MonthLocator())
ax.xaxis.set_major_formatter(DateFormatter('\n\n\n%b\n%Y'));
```



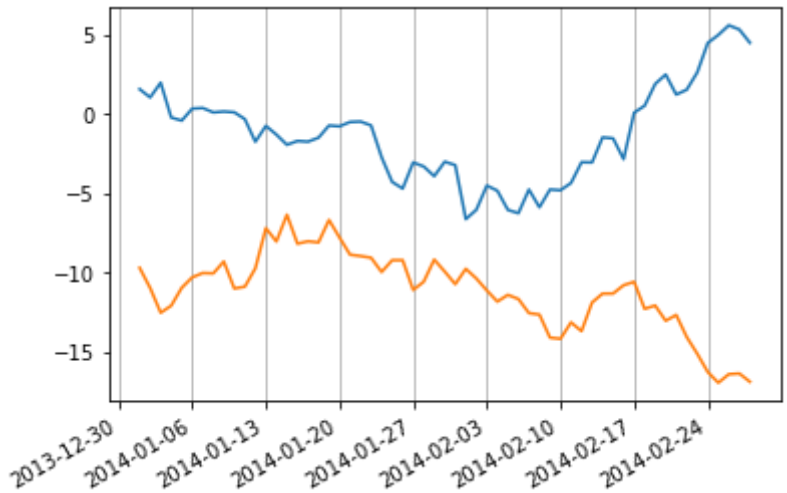
Мы можем воспользоваться методом `.grid()` объекта оси *x*, чтобы добавить направляющие линии сетки для промежуточных и основных делений оси. Первый параметр принимает значение `True`, чтобы выполнить отрисовку линий сетки, или значение `False`, чтобы скрыть их. Второй параметр задает тип делений – основные (`"major"`) или промежуточные (`"minor"`). Следующий программный код заново строит этот график, при этом линии сетки для основных делений шкалы отключены, а линии сетки для промежуточных делений, наоборот, включены:

In[21]:
```
# строим график
fig, ax = plt.subplots()

# информируем matplotlib, что используем даты
# обратите внимание на то, что нам нужно преобразовать
# индекс DatetimeIndex в однородный многомерный массив Numpy,
# состоящий из объектов datetime.datetime
ax.plot_date(walk_subset.index.to_pydatetime(), walk_subset, '-')

# настраиваем метки промежуточных делений
weekday_locator = WeekdayLocator(byweekday=(0), interval=1)
ax.xaxis.set_minor_locator(weekday_locator)
ax.xaxis.set_minor_formatter(DateFormatter('%d\n%a'))
ax.xaxis.grid(True, "minor") # включаем линии сетки для промежуточных делений
ax.xaxis.grid(False, "major") # отключаем линии сетки для основных делений

# настраиваем метки основных делений
ax.xaxis.set_major_locator(MonthLocator())
ax.xaxis.set_major_formatter(DateFormatter('\n\n\n%b\n%Y'));
```



И в заключение мы выведем метки и линии сетки только для основных делений на еженедельной основе и с использованием формата `YYYY-MM-DD`. Однако, поскольку они будут перекрываться, мы повернем их для предотвращения перекрытия. Это вращение можно задать с помощью функции `fig.autofmt_xdate()`:

In[22]:
```
# строим график
fig, ax = plt.subplots()

# информируем matplotlib, что используем даты
# обратите внимание на то, что нам нужно преобразовать
# индекс DatetimeIndex в однородный многомерный массив Numpy,
# состоящий из объектов datetime.datetime
ax.plot_date(walk_subset.index.to_pydatetime(), walk_subset, '-')

ax.xaxis.grid(True, "major") # выводим линии сетки для основных делений

# настраиваем метки основных делений
ax.xaxis.set_major_locator(weekday_locator)
ax.xaxis.set_major_formatter(DateFormatter('%Y-%m-%d'));

# выполняем поворот меток с датами
fig.autofmt_xdate();
```



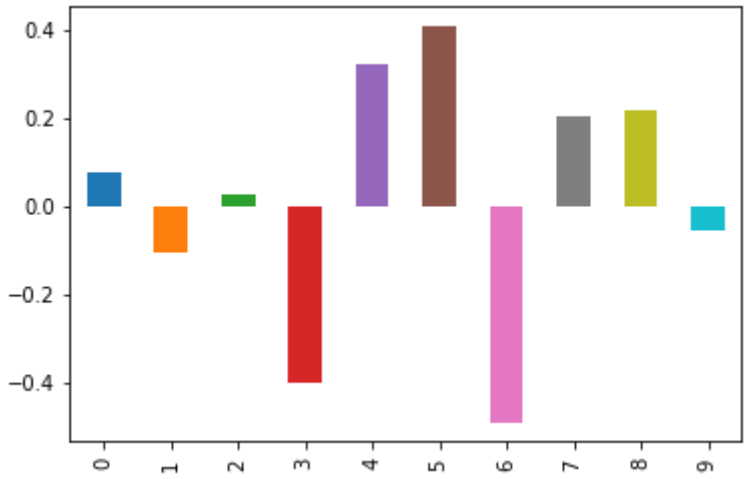
Виды графиков, часто использующиеся в статистическом анализе данных

Теперь, когда вы знаете, как создавать, настраивать и аннотировать графики временных рядов, мы рассмотрим построение различных диаграмм, которые могут пригодиться для визуализации статистической информации.

Демонстрация относительных различий с помощью столбиковых диаграмм

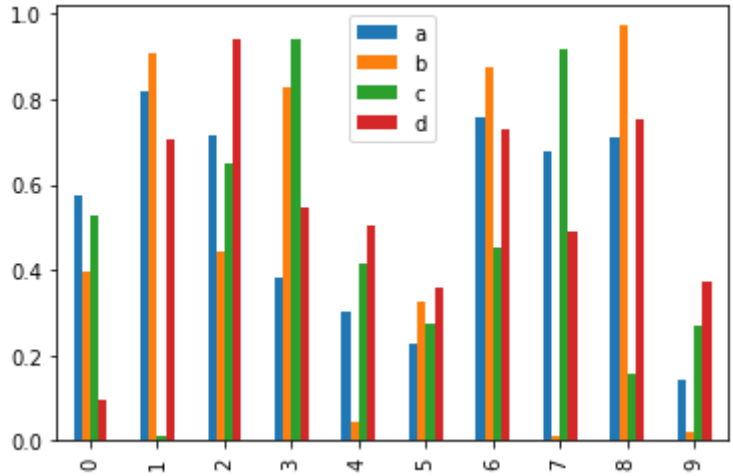
Столбиковые диаграммы позволяют визуализировать относительные различия между значениями обычных данных (не относящихся к временным рядам). Столбиковые диаграммы можно создать с помощью параметра `kind='bar'` метода `.plot()`:

In[23]:
```
# сгенерируем данные для столбиковой диаграммы
# сгенерируем небольшую серию, состоящую
# из 10 случайных значений
np.random.seed(seedval)
s = pd.Series(np.random.rand(10) - 0.5)
# строим столбиковую диаграмму
s.plot(kind='bar');
```



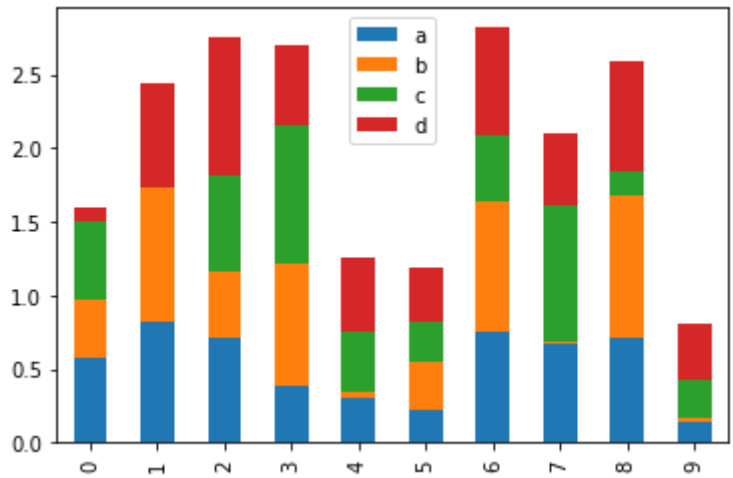
По умолчанию строится вертикальная столбиковая диаграмма. Для сравнения значений нескольких столбцов в каждой метке оси x можно построить столбиковую диаграмму на основе нескольких серий:

In[24]:
```
# сгенерируем данные для столбиковой диаграммы
# на основе нескольких серий
# сгенерируем 4 столбца, состоящих
# из 10 случайных значений
np.random.seed(seedval)
df2 = pd.DataFrame(np.random.rand(10, 4),
                   columns=['a', 'b', 'c', 'd'])
# строим столбиковую диаграмму
# на основе нескольких серий
df2.plot(kind='bar');
```



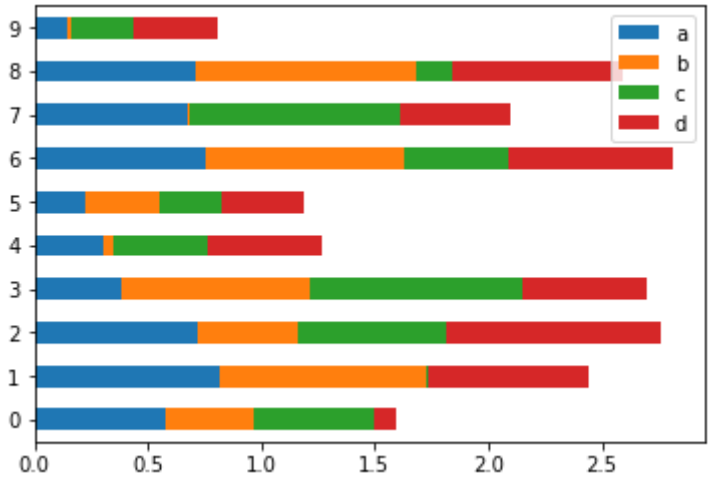
Параметр `stacked=True` можно использовать для построения состыкованной столбиковой диаграммы:

In[25]:
```
# вертикальная состыкованная
# столбиковая диаграмма
df2.plot(kind='bar', stacked=True);
```



Вертикальную ориентацию диаграммы, использующуюся по умолчанию, можно сменить на горизонтальную с помощью значения параметра `kind='barh'`:

In[26]:
```
# горизонтальная состыкованная
# столбиковая диаграмма
df2.plot(kind='barh', stacked=True);
```

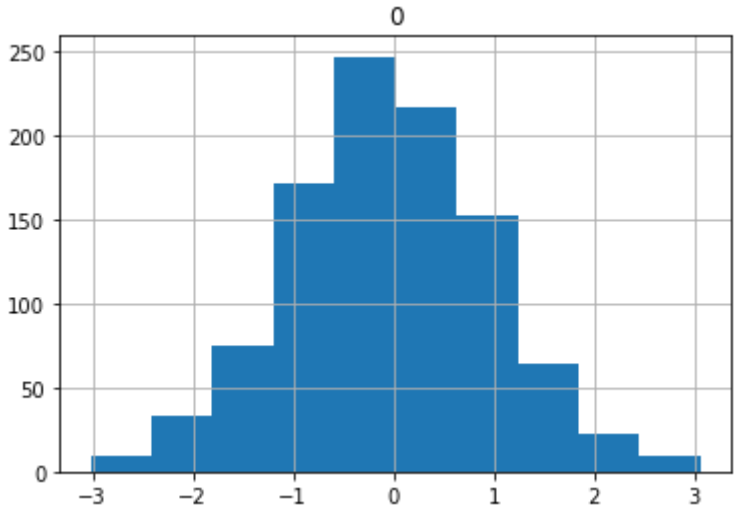


Визуализация распределений данных с помощью гистограмм

Гистограммы используют для визуализации распределений данных. Следующий программный код строит гистограмму на основе 1000 случайных чисел, подчиняющихся нормальному распределению:

In[27]:

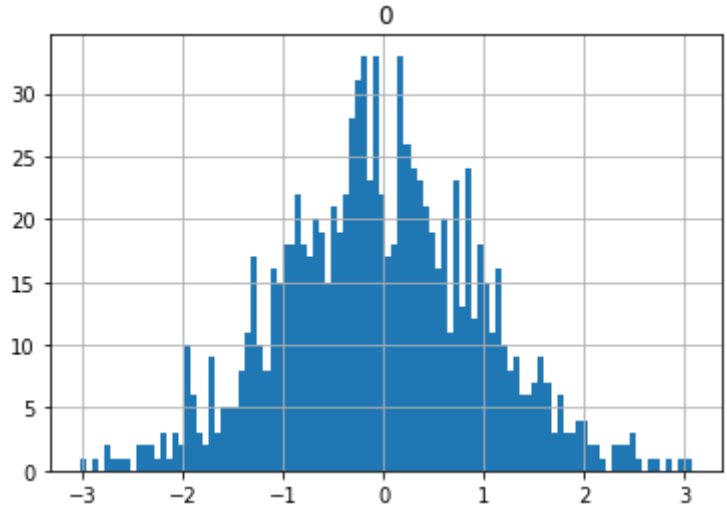
```
# сгенерируем данные для гистограммы
np.random.seed(seedval)
# генерируем 1000 случайных чисел
dfh = pd.DataFrame(np.random.randn(1000))
# строим гистограмму
dfh.hist();
```



Разрешающую способность гистограммы можно настроить, задав количество интервалов (бинов). Значение по умолчанию равно 10, увеличение количества интервалов детализирует гистограмму. Этот программный код увеличивает количество интервалов до `100`:

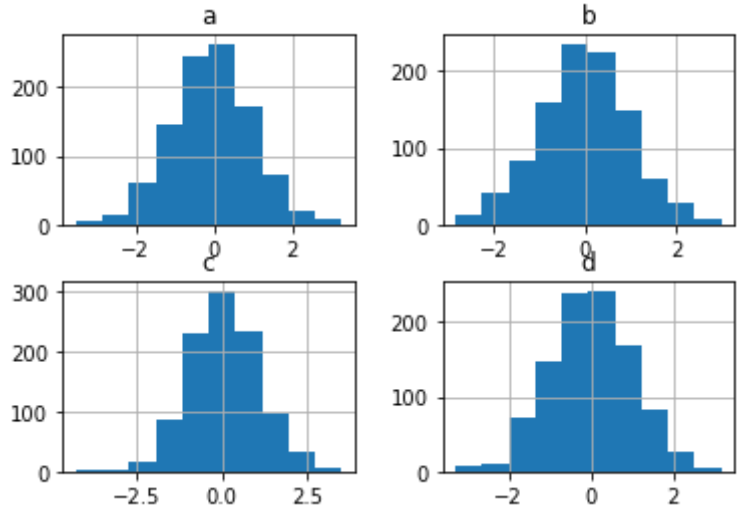
In[28]:

```
# снова строим гистограмму, но теперь с
# большим количеством интервалов (бинов)
dfh.hist(bins = 100);
```



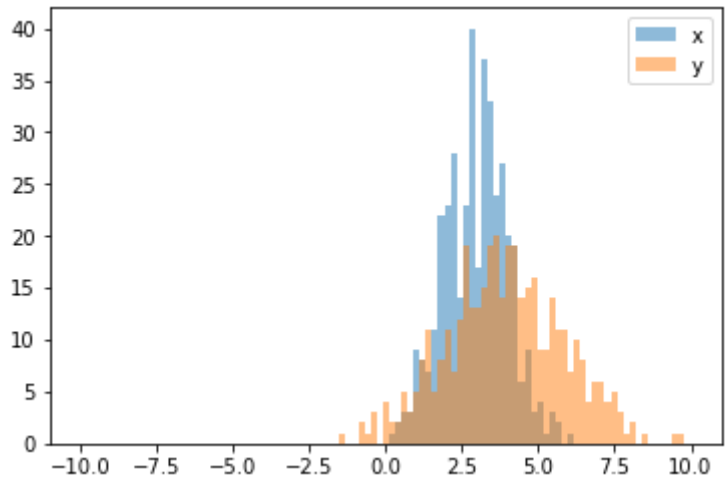
Если данные состоят из нескольких серий, функция построения гистограммы автоматически сгенерирует несколько гистограмм, по одной для каждой серии:

In[29]:
```
# сгенерируем данные для графика с
# несколькими гистограммами
# создаем датафрейм с 4 столбцами,
# каждый состоит из 1000 случайных чисел
np.random.seed(seedval)
dfh = pd.DataFrame(np.random.randn(1000, 4),
                   columns=['a', 'b', 'c', 'd'])
# строим график. Поскольку у нас 4 столбца,
# pandas строит четыре гистограммы
dfh.hist();
```



Чтобы наложить несколько гистограмм друг на друга в рамках одного и того же рисунка (и тем самым визуализировать разницу распределений), несколько раз вызовите функцию `pyplot.hist()` перед вызовом метода `.show()`:

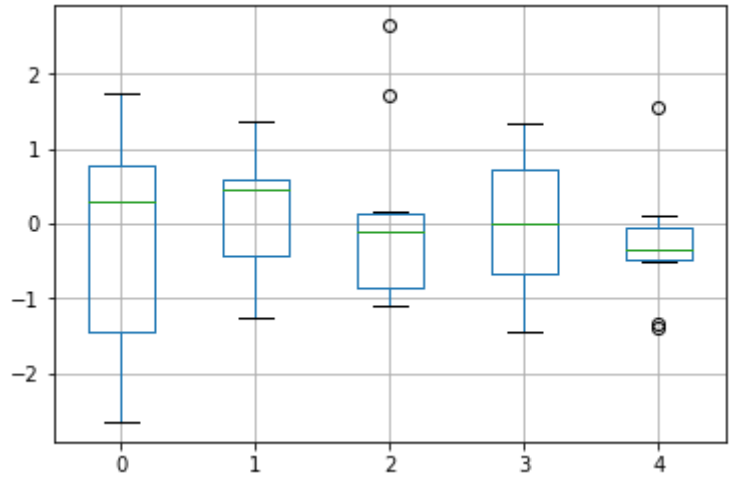
In[30]:
```
# напрямую воспользуемся модулем pyplot
# для наложения нескольких гистограмм
# сгенерируем два распределения, каждое со своим
# средним значением и стандартным отклонением
np.random.seed(seedval)
x = [np.random.normal(3,1) for _ in range(400)]
y = [np.random.normal(4,2) for _ in range(400)]

# задаем интервалы (диапазон от -10 до 10 и 100 интервалов)
bins = np.linspace(-10, 10, 100)

# строим график x с помощью plt.hist, 50% прозрачности
plt.hist(x, bins, alpha=0.5, label='x')
# строим график y с помощью plt.hist, 50% прозрачности
plt.hist(y, bins, alpha=0.5, label='y')
# размещаем легенду в верхнем правом углу
plt.legend(loc='upper right');
```



Визуализация распределений категориальных данных с помощью ящичных диаграмм с усами

Ящичные диаграммы основаны на описательных статистиках и являются полезным инструментом, позволяющим визуализировать распределения категориальных данных с помощью квартилей. Каждый ящик представляет значения, располагающиеся между первым и третьим квартилями, усы охватывают значения в пределах 1,5 межквартильных размахов от границ ящика:

In[31]:
```
# сгенерируем данные для ящичной диаграммы
# сгенерируем серию
np.random.seed(seedval)
dfb = pd.DataFrame(np.random.randn(10,5))
# строим график
dfb.boxplot(return_type='axes');
```

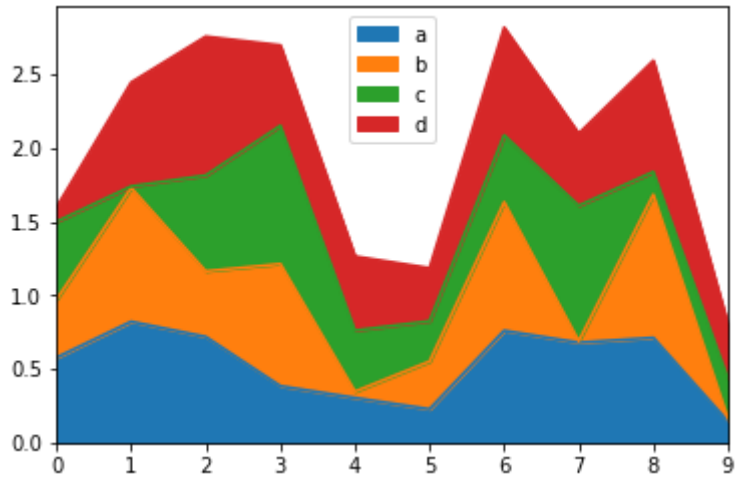


Отображение накопленных итогов с помощью площадных диаграмм

Площадные диаграммы используются для визуализации итогов, накопленных по прошествии определенного времени, и для демонстрации изменений трендов, происходящих с течением времени, по взаимосвязанным признакам. Кроме того, такие графики можно «сложить» для иллюстрации итоговых значений по всем переменным.

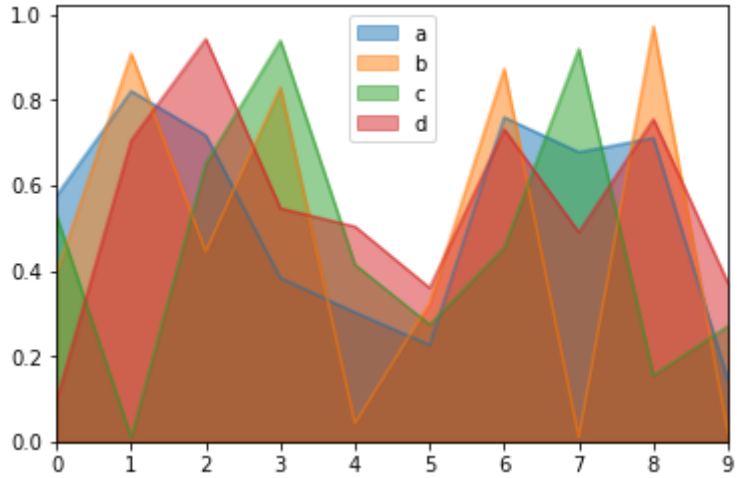
Площадную диаграмму можно создать, указав значение параметра `kind='area'`. По умолчанию строится состыкованная площадная диаграмма:

In[32]:
```
# сгенерируем данные для состыкованной
# площадной диаграммы
# создадим датафрейм из 4 столбцов,
# используя случайные числа
np.random.seed(seedval)
dfa = pd.DataFrame(np.random.rand(10, 4),
                   columns=['a', 'b', 'c', 'd'])
# создаем площадную диаграмму
dfa.plot(kind='area');
```



С помощью значения параметра `stacked=False` можно построить несостыкованную площадную диаграмму:

In[33]:
```
# строим несостыкованную площадную диаграмму
dfa.plot(kind='area', stacked=False);
```

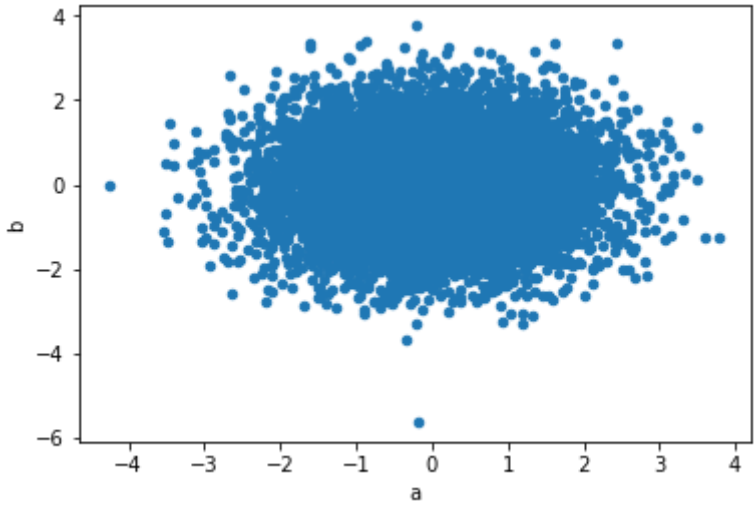


По умолчанию несостыкованные диаграммы имеют 50%-ную прозрачность (`alpha=0.5`), поэтому на графике видно, как несколько рядов данных перекрываются.

Визуализация взаимосвязи между двумя переменными с помощью диаграммы рассеяния

Диаграмма рассеяния визуализирует корреляцию между двумя переменных. Диаграмму рассеяния можно создать на основе объекта `DataFrame` с помощью метода `.plot()`, при этом указав значение параметра `kind='scatter'`, а также столбцы x и y исходного датафрейма:

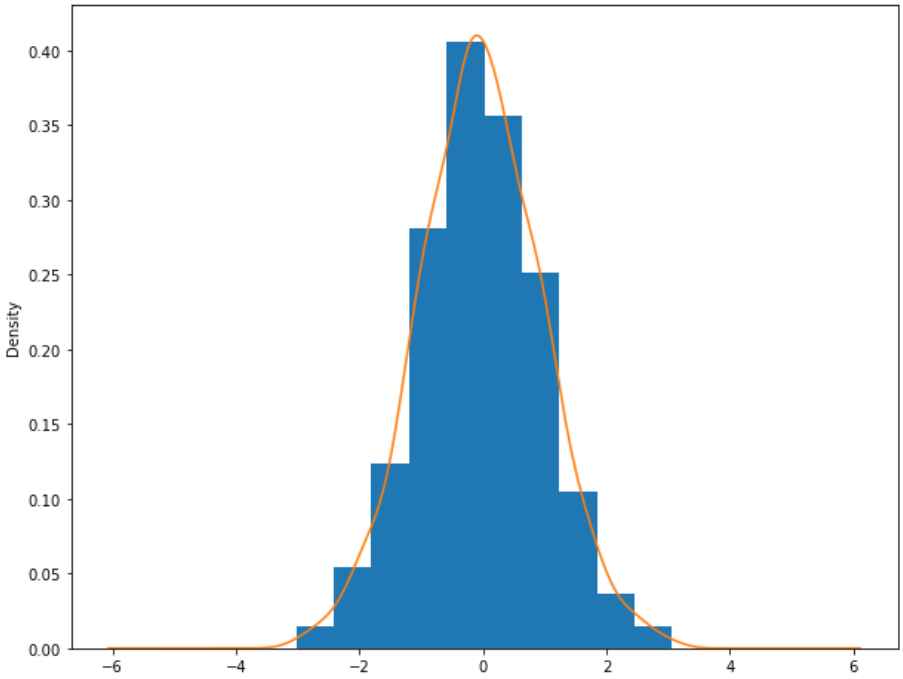
In[34]:
```
# создаем диаграмму рассеяния, состоящую из двух серий
# нормально распределенных случайных чисел
# мы ожидаем получить кластер с координатами центра 0,0
np.random.seed(seedval)
sp_df = pd.DataFrame(np.random.randn(10000, 2),
                     columns=['a', 'b'])
sp_df.plot(kind='scatter', x='a', y='b');
```



Визуализация оценок распределения с помощью графика ядерной оценки плотности

С гистограммой тесно связан график плотности, который строится на основе оценки непрерывного распределения вероятности по результатам измерений. Обычно стремятся аппроксимировать это распределение комбинацией ядер, т. е. комбинацией более простых распределений, например нормального (гауссова). Поэтому графики плотности еще называют графиками ядерной оценки плотности (KDE – kernel density estimate). Графики ядерной оценки плотности можно создать с помощью метода `.plot()` и значения параметра `kind='kde'`. Следующий программный код генерирует нормальный распределенный набор случайных чисел, отображает его как гистограмму и накладывает поверх нее график ядерной оценки плотности:

```
In[35]:
# создаем данные для графика ядерной
# оценки плотности
# сгенерируем серию из 1000 случайных чисел
np.random.seed(seedval)
s = pd.Series(np.random.randn(1000))
# строим график
s.hist(density=True) # выводим столбики
s.plot(kind='kde', figsize=(10,8));
```

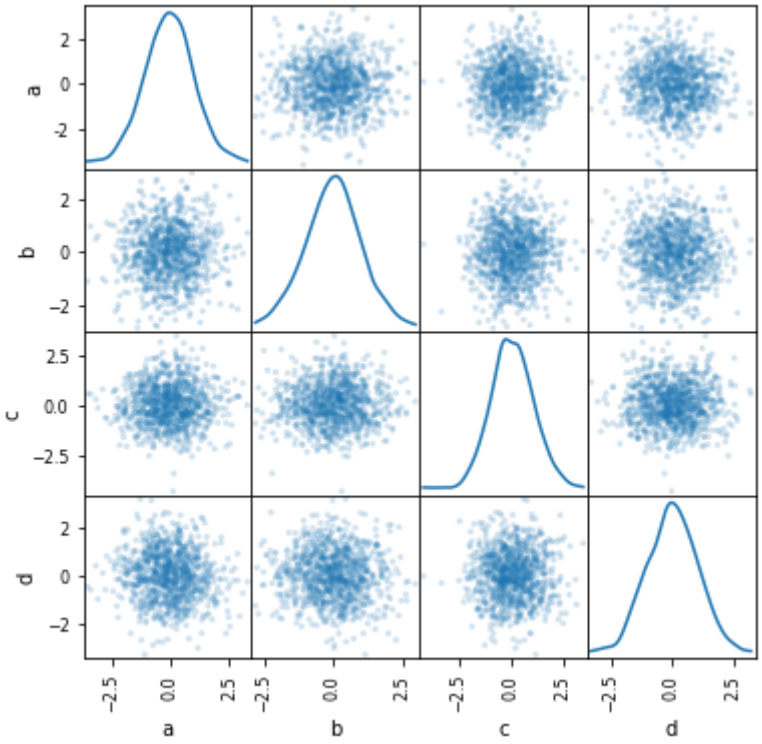


Визуализация корреляций между несколькими переменными с помощью матрицы диаграмм рассеяния

Матрица диаграмм рассеяния – популярный способ визуализации линейной корреляции между несколькими переменными. Программный код, приведенный ниже, создает матрицу диаграмм рассеяния на основе случайных чисел и строит диаграмму рассеяния для каждой комбинации переменных, а также график ядерной оценки плотности для каждой переменной по диагонали:

In[36]:

```
# создаем данные для матрицы
# диаграмм рассеяния
# импортируем класс scatter_matrix
from pandas.plotting import scatter_matrix

# создаем датафрейм с 4 столбцами,
# каждый состоит из 1000 случайных чисел
np.random.seed(seedval)
df_spm = pd.DataFrame(np.random.randn(1000, 4),
                      columns=['a', 'b', 'c', 'd'])
# строим матрицу диаграмм рассеяния
scatter_matrix(df_spm, alpha=0.2, figsize=(6, 6), diagonal='kde');
```



Мы вернемся к матрице диаграмм рассеяния снова, когда будем применять ее в анализе финансовых данных и вычислять корреляции между котировками акций.

Отображение взаимосвязей между несколькими переменными с помощью тепловых карт

Тепловая карта – это графическое представление данных, при котором значения внутри матрицы представлены цветами. Это эффективный инструмент, который позволяет визуализировать значения, получаемые на пересечении двух переменных.

Наиболее распространенный сценарий состоит в том, чтобы вычислить значения матрицы, нормированные в диапазоне от 0,0 по 1,0, и получить таблицу, которая образована пересечениями строк – корреляциями между двумя переменными. Значения с наименьшей корреляцией (0.0) будут отображаться в виде более темных тонов, а значения с наибольшей корреляцией (1.0) будут отображаться в виде более светлых тонов.

Тепловые карты можно легко создать с помощью функции `.imshow()`:

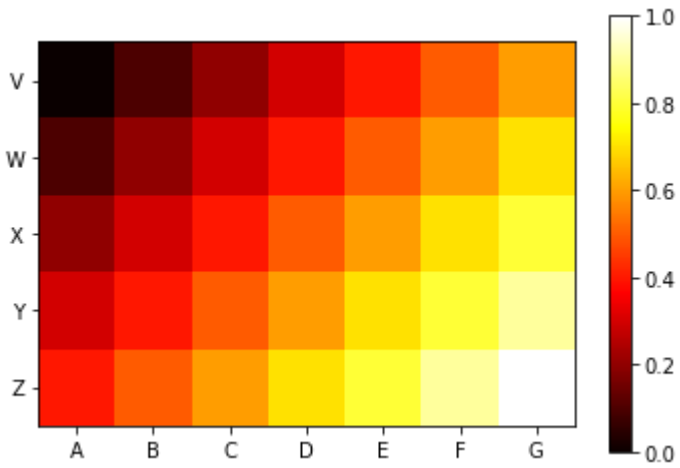
In[37]:

```
# создаем данные для теплокарты
s = pd.Series([0.0, 0.1, 0.2, 0.3, 0.4],
              ['V', 'W', 'X', 'Y', 'Z'])
heatmap_data = pd.DataFrame({'A' : s + 0.0,
                             'B' : s + 0.1,
                             'C' : s + 0.2,
                             'D' : s + 0.3,
                             'E' : s + 0.4,
                             'F' : s + 0.5,
                             'G' : s + 0.6
                    })
heatmap_data
```

```
     A    B    C    D    E    F    G
V  0.0  0.1  0.2  0.3  0.4  0.5  0.6
W  0.1  0.2  0.3  0.4  0.5  0.6  0.7
X  0.2  0.3  0.4  0.5  0.6  0.7  0.8
Y  0.3  0.4  0.5  0.6  0.7  0.8  0.9
Z  0.4  0.5  0.6  0.7  0.8  0.9  1.0
```

In[38]:

```
# строим теплокарту
plt.imshow(heatmap_data, cmap='hot', interpolation='none')
plt.colorbar()  # добавим шкалу интенсивности цвета
# задаем метки
plt.xticks(range(len(heatmap_data.columns)), heatmap_data.columns)
plt.yticks(range(len(heatmap_data)), heatmap_data.index);
```



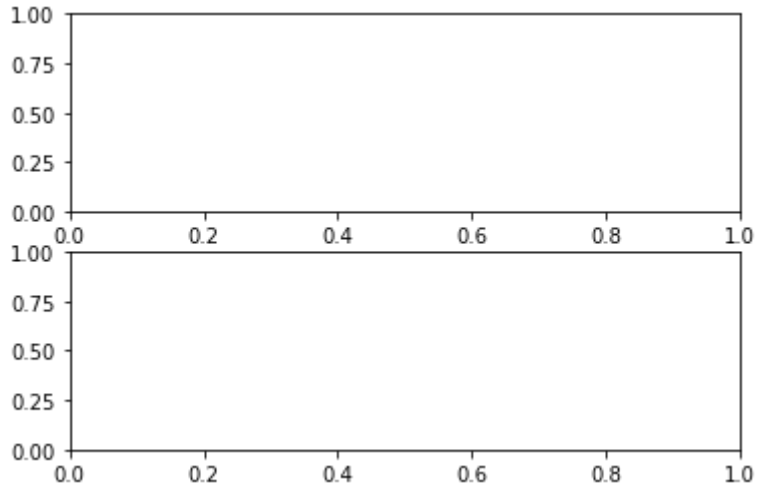
Размещение нескольких графиков на одном рисунке вручную

Часто бывает полезно сравнить данные, отобразив несколько графиков рядом друг с другом. Мы уже видели, что для нескольких типов графиков библиотека pandas делает это автоматически. Кроме того, можно вручную отобразить несколько графиков на одном и том же рисунке.

Чтобы разместить несколько подграфиков на одном и том же рисунке с помощью `matplotlib`, вызовите несколько раз функцию `plt.subplot2grid()`. На каждом проходе учитывается размер подграфика (`shape=(height, width)`) и его расположение (`loc=(row, column)`). Размер – это общее количество столбцов, а не пикселей.

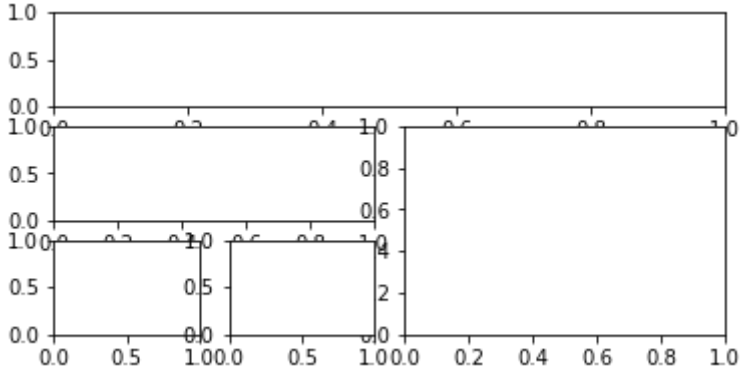
Значение, возвращаемое каждым вызовом функции `plt.subplot2grid()` – это уникальный объект `AxesSubplot`, который можно использовать для отрисовки подграфика.
Следующий программный код демонстрирует это, создав рисунок на основе двух строк и одного столбца (`shape = (2,1)`). Первый подграфик, на который ссылается `ax1`, находится в первой строке (`loc=(0,0)`), а второй, на который ссылается `ax2`, находится во второй строке (`loc=(1,0)`):

In[39]:
```
# создаем два подграфика на одном рисунке,
# используя сетку 2x1 (2 строки и 1 столбец)
# ax1 - верхняя строка
ax1 = plt.subplot2grid(shape=(2,1), loc=(0,0))
# и ax2 - нижняя строка
ax2 = plt.subplot2grid(shape=(2,1), loc=(1,0))
```



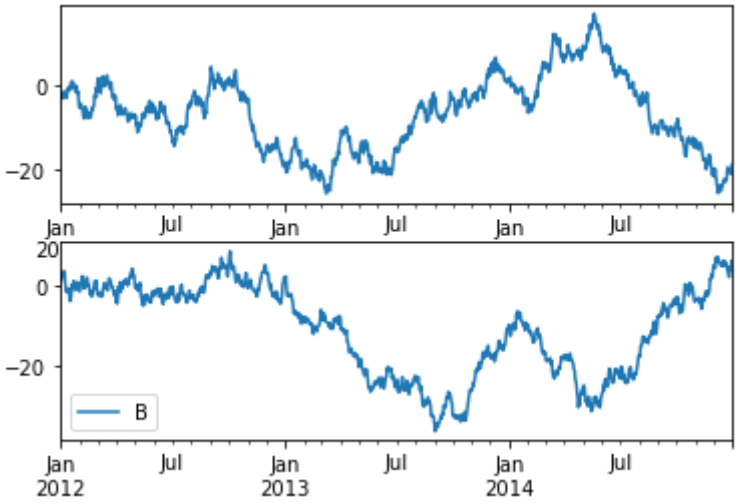
Макет с подграфиками создан, но сами подграфики не отрисованы. Размер любого подграфика можно задать с помощью параметров `rowspan` и `colspan` в каждом вызове функции `plt.subplot2grid()`. Значения этих параметров определяют количество строк/столбцов, объединяемых в одну ячейку. Эти параметры похожи на атрибуты rowspan/colspan в таблицах HTML, устанавливающие количество ячеек, которые нужно объединить по вертикали/по горизонтали. Программный код демонстрирует использование параметров rowspan и colspan для создания более сложного макета, состоящего из пяти графиков. Для каждого из этих графиков мы задаем разные строки, столбцы и разное количество объединяемых строк/столбцов:

In[40]:
```
# создаем макет с подграфиками, используя сетку 4x4
# ax1 расположен в первой строке, шириной 4 столбца,
# все 4 столбца объединены в одну ячейку
ax1 = plt.subplot2grid((4,4), (0,0), colspan=4)
# ax2 расположен во второй строке, крайний слева и имеет
# ширину в 2 столбца, при этом 2 столбца объединены в одну ячейку
ax2 = plt.subplot2grid((4,4), (1,0), colspan=2)
# ax3 имеет 2 столбца в ширину и 2 строки в высоту,
# при этом два столбца и две строки объединены в одну ячейку
ax3 = plt.subplot2grid((4,4), (1,2), colspan=2, rowspan=2)
# ax4 имеет 1 строку в высоту и 1 столбец в ширину,
# расположен в строке 4 и столбце 0,
# объединение строк и столбцов не задано
ax4 = plt.subplot2grid((4,4), (2,0))
# ax4 имеет 1 строку в высоту и 1 столбец в ширину,
# расположен в строке 4 и столбце 1
# объединение строк и столбцов не задано
ax5 = plt.subplot2grid((4,4), (2,1));
```



Чтобы построить конкретный подграфик, передайте интересующий объект axes в метод `.plot()` с помощью параметра `ax`. Следующий программный код демонстрирует это, извлекая две серии из данных, созданных на основе случайного блуждания в начале главы, а затем отрисовывает каждую в виде двух отдельных подграфиков:

In[41]:
```
# демонстрируем построение подграфиков
# создаем макет, используя сетку 2x1,
# в каждой строке по одному подграфику
ax5 = plt.subplot2grid((2,1), (0,0))
ax6 = plt.subplot2grid((2,1), (1,0))
# отрисовываем первый подграфик на основе столбца 0
# датафрейма walk_df в верхней строке сетки
walk_df[walk_df.columns[0]].plot(ax = ax5)
# отрисовываем второй подграфик на основе столбца 1
# датафрейма walk_df в нижней строке сетки
walk_df[[walk_df.columns[1]]].plot(ax = ax6);
```



Выводы

В этой главе мы рассмотрели различные популярные способы визуализации данных, предлагаемые библиотекой pandas. Визуализация данных – один из лучших способов быстро понять то, что хотят вам рассказать данные. Python, pandas и matplotlib (и еще несколько других библиотек) позволяют вам быстро понять содержательный смысл данных, красиво визуализировать его при помощи нескольких строчек программного кода.

ПРИЛОЖЕНИЕ 1. СОВЕТЫ ПО ОПТИМИЗАЦИИ ВЫЧИСЛЕНИЙ В БИБЛИОТЕКЕ PANDAS

Материал подготовлен Софией Хайслер в рамках конференции PyCon 2017.

Сейчас мы рассмотрим эффективность различных способов применения функций к объекту `DataFrame`, от самых медленных до самых быстрых:

- базовое итерирование по строкам объекта `DataFrame` с помощью индексов;
- итерирование с помощью метода `.iterrows()`;
- итерирование с применением метода `.apply()`;
- векторизация с помощью объектов `Series`;
- векторизация с помощью массивов NumPy.

Импортируем необходимые библиотеки

```
In[1]:
# загружаем основные библиотеки
import numpy as np
import pandas as pd
```

Для нашей примерной функции мы будем использовать формулу гаверсинуса (или великого круга). Формула гаверсинуса – важное уравнение, использующееся в навигации и позволяющее вычислить расстояние между точками на сфере по их долготе и широте. Она выглядит так:

$$d = 2r\arcsin\left(\sqrt{\sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + \cos(\phi_1) \times \cos(\phi_2) \times \sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$$

где:
d – расстояние между точками;
r – радиус сферы (то есть радиус Земли, который равен примерно 6371 км или 3959 милям);
φ_1, φ_2 – широта точки 1 и широта точки 2;
λ_1, λ_2 – долгота точки 1 и долгота точки 2.

Итак, наша функция принимает в качестве аргументов широту и долготу двух точек, учитывает кривизну поверхности Земли и вычисляет прямолинейное расстояние между ними. Функция выглядит примерно так:

```
In[2]:
# пишем функцию, которая вычисляет
# формулу гаверсинусов
def haversine(lat1, lon1, lat2, lon2):
    MILES = 3959
    lat1, lon1, lat2, lon2 = map(np.deg2rad, [lat1, lon1, lat2, lon2])
    dlat = lat2 - lat1
    dlon = lon2 - lon1
    a = np.sin(dlat/2)**2 + np.cos(lat1) * np.cos(lat2) * np.sin(dlon/2)**2
    c = 2 * np.arcsin(np.sqrt(a))
    total_miles = MILES * c
    return total_miles
```

Чтобы проверить работу нашей функции на реальных данных, мы воспользуемся набором данных, содержащим координаты всех отелей в штате Нью-Йорк, скачанными с сайта Expedia http://developer.ean.com/. Мы рассчитаем расстояние между каждым отелем и набором координат (который принадлежит небольшому магазину под названием Brooklyn Superhero Supply Store https://www.superherosupplies.com/ в Нью-Йорке).
Давайте загрузим этот набор.

```
In[3]:
# загружаем данные, на которых будем тестировать
# нашу функцию
df = pd.read_csv('Data/new_york_hotels.csv', encoding='cp1252')
df.head()
```

Базовое итерирование

Почти каждый начинающий пользователь библиотеки pandas пытается применить настраиваемую функцию, просто итерируя строки объекта `DataFrame` (то есть проходя одну строку за раз). Преимущество такого подхода заключается в том, что оно согласуется с тем, как можно было бы взаимодействовать с другими итерируемыми объектами Python, например, можно итерировать по списку или кортежу. И, наоборот, недостатком является то, что базовое итерирование в pandas является самым медленным способом добиться чего-либо. В отличие от подходов, которые мы обсудим ниже, базовое итерирование в pandas не использует никаких встроенных оптимизаций, что делает ее чрезвычайно неэффективной (и часто гораздо менее читаемой) в сравнении с другими способами.
Например, можно написать что-то вроде этого:

```
In[4]:
# задаем функцию, которая просто перебирает все строки
# и возвращает серию с расстояниями, вычисленными
# по формуле гаверсинусов
def haversine_looping(df):
    distance_list = []
    for i in range(0, len(df)):
        d = haversine(40.671, -73.985, df.iloc[i]['latitude'], df.iloc[i]['longitude'])
        distance_list.append(d)
    return distance_list
```

Чтобы получить представление о времени, которое потребуется для выполнения функции, приведенной выше, мы воспользуемся функцией `%timeit`. `%timeit` – это magic-функция, созданная специально для работы с тетрадками Jupyter. (Все magic-функции начинаются со знака `%`, если функция применяется к одной строке, и `%%`, если применяется ко всей ячейке Jupyter). `%timeit` многократно запускает функцию и печатает среднее и стандартное отклонение полученных результатов. Конечно, время выполнения, полученное с помощью `%timeit`, необязательно будет одинаковым для каждой системы, выполняющей эту функцию. Тем не менее она является полезным инструментом, позволяющим сравнить время выполнения различных функций в одной и той же системе для одного и того же набора данных.

In[5]:
```
%%timeit

# запускаем итеративную функцию haversine
df['distance'] = haversine_looping(df)
```

```
696 ms ± 53.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Наша функция выполняется 696 миллисекунд со стандартным отклонением 53.4 миллисекунды. Это может показаться быстрым, но на самом деле это довольно медленно, учитывая лишь одну функцию, необходимую для обработки 1600 строк. Давайте посмотрим, как можно улучшить этот результат.

Итерирование с помощью метода .iterrows()

В ситуации, когда необходимо итерирование, более быстрым способом итерирования строк будет использование метода `.iterrows()`. Метод `.iterrows()` оптимизирован для работы с датафреймами и хотя это наименее эффективный способ запуска большинства стандартных функций (подробнее об этом позже), он дает значительное улучшение по сравнению с базовым итерированием. В нашем случае метод `.iterrows()` решает ту же проблему в три раза быстрее, чем итерирование по строкам вручную.

In[6]:
```
%%timeit

# запускаем итерирование с помощью
# метода .iterrrows()
haversine_series = []
for index, row in df.iterrows():
    haversine_series.append(haversine(40.671, -73.985, row['latitude'], row['longitude']))
df['distance'] = haversine_series
```

```
215 ms ± 48.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Более лучший способ итерирования с помощью метода .apply()

Еще лучшим способом, чем метод `.iterrows()`, является использование метода `.apply()`, который применяет функцию вдоль определенной оси (вдоль строк или вдоль столбцов) объекта `DataFrame`. Хотя метод `.apply()` также по своей сути перебирает строки, он делает это намного эффективнее, чем метод `iterrows()`, используя ряд внутренних оптимизаций, например использование итераторов, написанных на Cython.

```
In[7]:
%%timeit

# применяем функцию haversine с помощью
# метода .apply()
df['distance'] = df.apply(lambda row: haversine(40.671, -73.985, row['latitude'],
                                                row['longitude']), axis=1)

81.2 ms ± 5.04 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Векторизация с помощью объектов Series

Чтобы понять, как мы можем уменьшить количество итераций, выполняемых функцией, напомним, что объекты библиотеки pandas `DataFrame` и `Series` используют в своей основе массивы. Встроенные функции pandas предназначены для работы с целыми массивами, а не для последовательной обработки отдельных значений (скаляров). Векторизация – это процесс выполнения операций над целыми массивами.

Библиотека pandas включает в себя богатую коллекцию векторизованных функций для любых операций: от математических операций до агрегирующих и строковых функций (с обширным списком доступных функций можно ознакомиться в документации pandas). Встроенные функции оптимизированы специально для работы с объектами `Series` и `DataFrame`. В результате использование векторизованных функций pandas почти всегда предпочтительнее пользовательских циклов.

До сих пор мы передавали скаляры только нашей функции `haversine`. Однако все функции, используемые внутри функции `haversine`, также могут работать с массивами. Это существенно упрощает процесс векторизации нашей функции: вместо передачи ей отдельных скалярных значений широты и долготы мы передадим ей серии (столбцы) `latitude` и `longitude` целиком. Это позволит библиотеке pandas извлечь выгоду благодаря обширному набору оптимизаций, доступных для векторизованных функций, включая, в частности, выполнение всех вычислений по всему массиву одновременно.

```
In[8]:
%%timeit

# векторизированная реализация функции haversine,
# используем объекты Series целиком
df['distance'] = haversine(40.671, -73.985,df['latitude'], df['longitude'])

2.09 ms ± 225 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Мы добились почти 40-кратного улучшения по сравнению с методом `.apply()` и более чем 100-кратного улучшения по сравнению с методом `.iterrows()` путем векторизации функции!

Векторизация с помощью массивов NumPy

На этом мы могли бы и закончить, ведь векторизация с помощью серий позволяет решить большую часть задач оптимизации, связанных с повседневными вычислениями. Однако, если скорость имеет наивысший приоритет, мы можем вызвать подкрепление в виде библиотеки NumPy. Библиотека NumPy, которую можно описать как «фундаментальный пакет для научных вычислений в Python», выполняет операции под капотом, используя оптимизированный, предварительно скомпилированный код на C. Как и pandas, NumPy работает с массивами (называемыми `ndarray`), однако она освобождена от дополнительных вычислительных затрат, связанных с операциями в pandas, такими как индексирование, проверка типов данных и т. д. В результате операции над массивами NumPy могут выполняться значительно быстрее, чем операции над объектами `Series`.

Массивы NumPy можно использовать вместо объектов `Series`, когда дополнительная функциональность, предлагаемая объектами `Series`, не является критичной. Например, векторизованная реализация нашей функции `haversine` фактически не использует индексы в сериях `latitude` и `longitude`, и поэтому отсутствие этих индексов не приведет к нарушению работы функции. Для сравнения, если бы мы выполняли операции типа соединения объектов `DataFrame`, когда нужно сослаться на ссылаться на значения по индексу, нам понадобилось бы использовать именно объекты `DataFrame`.

Мы преобразуем наши серии в массивы NumPy `latitude` и `longitude`, просто используя свойство `.values` объекта `Series`. Как и в случае векторизации с помощью объектов `Series`, передача массива NumPy непосредственно в функцию приведет к тому, что pandas применит эту функцию ко всему вектору.

```
In[9]:
%%timeit

# векторизированная реализация функции haversine,
# используем вместо объектов Series массивы NumPy
df['distance'] = haversine(40.671, -73.985, df['latitude'].values, df['longitude'].values)

372 µs ± 32.1 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Векторизация с помощью массивов Numpy позволила добиться почти 6-кратного улучшения по сравнению с предыдущим результатом.

Выводы

Подведем итоги, записав результаты в таблицу.

Способ	Среднее время выполнения	Улучшение производительности (сравнение с предыдущим результатом)
Базовое итерирование	696 мс	
Итерирование с помощью .iterrows()	215 мс	3.2x
Итерирование с помощью .apply()	81.2 мс	2.6x
Векторизация с помощью объектов Series	2.09 мс	38.9x
Векторизация с помощью массивов Numpy	0.37 мс	5.6x

Все это позволяет сделать несколько основных выводов по оптимизации вычислений в pandas:

1. Избегайте циклов: они медленно работают и в большинстве распространенных задач не нужны.
2. Если нужно выполнить цикл, используйте метод `.apply()`, а не итеративные функции.
3. Векторизация обычно лучше скалярных операций. Наиболее распространенные операции в pandas можно векторизовать.
4. Векторные операции над массивами NumPy более эффективны, чем операции над объектами `Series`.

ПРИЛОЖЕНИЕ 2. УЛУЧШЕНИЕ ПРОИЗВОДИТЕЛЬНОСТИ PANDAS (ИЗ ОФИЦИАЛЬНОГО ПОСОБИЯ ПО БИБЛИОТЕКЕ PANDAS)

Написание расширений на языке C для pandas

В большинстве случаев библиотеки pandas, использующей «чистый» Python и библиотеку NumPy, будет достаточно. Однако в некоторых приложениях, «тяжелых» с вычислительной точки зрения, можно добиться значительного ускорения за счет использования Cython.

В этом учебном пособии предполагается, что вы уже оптимизировали ваш питоновский код насколько это возможно, например, попытались избавиться от циклов for, использовали векторизацию с помощью массивов NumPy, потому что всегда стоит сначала оптимизировать исходный питоновский код.

В этом пособии рассматривается «типичный» процесс цитонизации медленных вычислений. Мы воспользуемся примером, взятым из документации по Cython, но в контексте библиотеки pandas. Наше итоговое цитонизированное решение будет примерно в 100 раз быстрее решения, написанного на «чистом» Python.

«Чистый» Python

У нас есть объект `DataFrame`, к строкам которого мы хотим применить функцию.

In[1]:
```
# импортируем numpy и pandas
import numpy as np
import pandas as pd

np.random.seed(12345)
df = pd.DataFrame({'a': np.random.randn(1000),
                   'b': np.random.randn(1000),
                   'N': np.random.randint(100, 1000, (1000)),
                   'x': 'x'})
```

In[2]:
```
df.head(5)
```

Out[2]:

	N	a	b	x
0	826	-0.204708	-0.983505	x
1	220	0.478943	0.930944	x
2	401	-0.519439	-0.811676	x
3	363	-0.555730	-1.830156	x
4	235	1.965781	-0.138730	x

Ниже приведены функции, написанные на «чистом» Python:

In[3]:
```
def f(x):
    return x * (x - 1)
```

In[4]:
```
def integrate_f(a, b, N):
    s = 0
    dx = (b - a) / N
    for i in range(N):
        s += f(a + i * dx)
    return s * dx
```

Мы получаем наш результат, применив метод `.apply()` построчно:

In[5]:
```
%timeit df.apply(lambda x: integrate_f(x['a'], x['b'], x['N']), axis=1)
```

```
1 loop, best of 3: 203 ms per loop
```

Но этого явно недостаточно для нас. Давайте посмотрим, на что было потрачено время во время выполнения этой операции (ограничиваясь четырьмя наиболее затратными вызовами) используя magic-функцию IPython `%prun`:

In[6]:
```
%prun -l 4 df.apply(lambda x: integrate_f(x['a'], x['b'], x['N']), axis=1)
```

```
   671118 function calls (666109 primitive calls) in 0.325 seconds

   Ordered by: internal time
   List reduced from 141 to 4 due to restriction <4>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1000    0.171    0.000    0.242    0.000 <ipython-input-4-
a877a66f40a5>:1(integrate_f)
   552542    0.072    0.000    0.072    0.000 <ipython-input-3-0b27f90c4c8a>:1(f)
     3000    0.010    0.000    0.054    0.000 base.py:2405(get_value)
     3000    0.006    0.000    0.062    0.000 series.py:598(__getitem__)
```

Безусловно, большая часть времени тратится на вызовы функций `integrate_f` и `f`, поэтому мы сосредоточим наши усилия на цитонизации именно этих двух функций.

Обычный Cython

Вам понадобится импортировать magic-функцию Cython в IPython (для версий cython <0.21 вы можете использовать `%load_ext cythonmagic`):

```
In[7]:
%load_ext Cython
```

Теперь давайте просто передадим наши функции в Cython, как есть (суффикс `plain` здесь используется для того, чтобы различать версии функций):

```
In[8]:
%%cython
def f_plain(x):
    return x * (x - 1)

def integrate_f_plain(a, b, N):
    s = 0
    dx = (b - a) / N
    for i in range(N):
        s += f_plain(a + i * dx)
    return s * dx
```

```
In[9]:
%timeit df.apply(lambda x: integrate_f_plain(x['a'], x['b'], x['N']), axis=1)

10 loops, best of 3: 118 ms per loop
```

Уже видим сокращение времени вычислений, что не так уж плохо для простого копирования и вставки.

Использование библиотеки Numba

Еще одна возможность ускорить вычисление – использование динамического JIT-компилятора библиотеки Numba.

Numba дает вам возможность ускорить работу приложений с помощью высокопроизводительных функций, написанных непосредственно на Python. С помощью нескольких аннотаций массиво-ориентированный и математически нагруженный питоновский код можно точно скомпилировать в машинный код, аналогичный по производительности языкам C, C ++ и Fortran, при этом можно избежать переключения языков или интерпретаторов Python.

Numba использует JIT-компилятор на основе инфраструктуры LLVM (Low-Level Virtual Machine), позволяющий транслировать код Python в машинный код во время исполнения программы, то есть «на лету» (just-in-time, отсюда и название). Numba поддерживает компиляцию Python в машинный код на любом CPU или GPU и предназначен для интеграции со стеком научного программного обеспечения Python.

Обратите внимание, вы должны установить библиотеку Numba. Это легко сделать с помощью conda: `conda install numba`. Начиная с Numba версии 0.20, объекты pandas нельзя непосредственно передать непосредственно в функции, скомпилированные с помощью Numba. Вместо этого в функцию, скомпилированную с помощью Numba, нужно передать массив NumPy, лежащий в основе объекта библиотеки pandas, как это будет показано ниже.

Jit

Итак, воспользуемся библиотекой Numba для компиляции нашего кода на «лету». Мы просто берем обычный питоновский код, приведенный выше, и аннотируем с помощью декоратора @jit.

In[10]:
```
import numba


@numba.jit
def f_plain(x):
   return x * (x - 1)

@numba.jit
def integrate_f_numba(a, b, N):
   s = 0
   dx = (b - a) / N
   for i in range(N):
       s += f_plain(a + i * dx)
   return s * dx

@numba.jit
def apply_integrate_f_numba(col_a, col_b, col_N):
   n = len(col_N)
   result = np.empty(n, dtype='float64')
   assert len(col_a) == len(col_b) == n
   for i in range(n):
      result[i] = integrate_f_numba(col_a[i], col_b[i], col_N[i])
   return result

def compute_numba(df):
   result = apply_integrate_f_numba(df['a'].values, df['b'].values, df['N'].values)
   return pd.Series(result, index=df.index, name='result')
```

In[11]:
```
%timeit compute_numba(df)
```

```
The slowest run took 438.90 times longer than the fastest. This could mean that an
intermediate result is being cached.
1 loop, best of 3: 1.09 ms per loop
```

Обратите внимание, что мы напрямую передаем массивы NumPy в функцию библиотеки Numba. `compute_numba` – это просто обертка, которая обеспечивает более удобный интерфейс, передавая/возвращая объекты pandas.

Vectorize

Кроме того, библиотеку numba можно использовать для написания векторизованных функций, которые не требуют, чтобы пользователь в явном виде итерировал наблюдения вектора. Рассмотрим следующий игрушечный пример, в котором мы умножаем все наблюдения в столбце `a` на 2:

In[12]:
```
# импортируем vectorize
from numba import vectorize

# пишем функцию, которая умножает каждое
# значение на 2, не используя numba
def double_every_value_nonumba(x):
    return x*2
```

```
# пишем функцию, которая умножает каждое
# значение на 2, используя numba
@vectorize
def double_every_value_withnumba(x):
    return x*2

In[13]:
# применяем самостоятельно написанную функцию
# без использования numba
%timeit df['col1_doubled'] = df.a.apply(double_every_value_nonumba)

In[14]:
# применяем реализацию по умолчанию (работает быстрее
# самостоятельно написанной функции, не использующей
# numba)
%timeit df['col1_doubled'] = df.a*2

1000 loops, best of 3: 292 µs per loop


In[15]:
# применяем самостоятельно написанную функцию
# с использованием библиотеки numba
%timeit df['col1_doubled'] = double_every_value_withnumba(df.a.values)

The slowest run took 556.64 times longer than the fastest. This could mean that an
intermediate result is being cached.
10000 loops, best of 3: 172 µs per loop
```

Обратите внимание, библиотека Numba может выполнить любую функцию, однако ускорить она может лишь определенные функции. Библиотека Numba – лучший инструмент для ускорения функций, выполняющие числовые операции над массивами NumPy. Когда мы передаем функцию, использующую только те операции, которые Numba знает, как ускорить, библиотека работает в режиме `nopython`.
Если библиотеке Numba передать функцию, которая использует то, с чем библиотека Numba не умеет работать – в настоящее время речь идет о множествах, списках, словарях и строковых функциях, она переключается в режим `object`. В режиме `object` Numba выполнит ваш программный код, но при этом значительного увеличения производительности не произойдет. Опционно библиотека Numba может выдать ошибку, если не сможет скомпилировать функцию так, чтобы ускорить выполнение вашего программного кода. Для этого передайте numba аргумент `nopython=True` (например, `@numba.jit(nopython=True)`). Для получения более подробной информации об устранении неполадок при работе с numba смотрите страницу устранения неполадок http://numba.pydata.org/numba-doc/0.20.0/user/troubleshoot.html#the-compiled-code-is-too-slow. Для получения более подробной информации о самой библиотеке numba смотрите документацию по библиотеке numba http://numba.pydata.org/.

Вычисление выражений с помощью функции eval()

Высокоуровневая функция `pandas.eval()` позволяет применять строковые выражения для эффективных вычислительных операций с

объектами `Series` и `DataFrame`. Обратите внимание, чтобы получить преимущество от использования `eval()`, вам необходимо установить библиотеку Numexpr.
Применение функции `eval()` вместо простого питоновского программного кода обусловлено двумя причинами: 1) большие объекты `DataFrame` оцениваются более эффективно и 2) большие арифметические и логические выражения оцениваются сразу (по умолчанию для вычислений используется библиотека `Numexpr`).
Обратите внимание, вы не должны использовать `eval()` для простых выражений или для выражений, в которых фигурируют небольшие объекты `DataFrame`. Фактически при работе с простыми выражениями/объектами функция `eval()` на несколько порядков медленнее простого питоновского кода. Хорошим правилом будет использование функции `eval()`, когда у вас есть объект `DataFrame`, содержащий более 10000 строк. Чем больше датафрейм и чем больше выражение, тем большего ускорения вы добьетесь, используя функцию `eval()`.

Поддерживаемый синтаксис

Функция `pandas.eval()` поддерживает следующие операции:

- арифметические операции, за исключением операторов сдвига влево (<<) и сдвига вправо (>>), например, `df + 2 * pi / s ** 4% 42 - the_golden_ratio`
- операции сравнения, включая последовательные сравнения, например, `2 <df <df2`
- логические операции, например, `df <df2 и df3 <df4 or not df_bool`
- список и кортеж, например, `[1, 2]` или `(1, 2)`
- атрибутивный доступ, например, `df.a`
- выражения с индексами, например, `df[0]`
- оценка простой переменной, например, pd.eval (`'df'`) (это не очень полезно)
- математические функции, например, *sin*, *cos*, *exp*, *log*, *expm1*, *log1p*, *sqrt*, *sinh*, *cosh*, *tanh*, *arcsin*, *arccos*, *arctan*, *arccosh*, *arcsinh*, *arctanh*, *abs* и *arctan2*.

Синтаксис Python, приведенный ниже, не поддерживается:

Выражения:

- вызовы функций, кроме математических
- операции `is/is not`
- выражения `if`
- `lambda` выражения
- абстракции списков/множеств/словарей
- выражения-литералы `dict` и `set`
- `yield`-выражения
- выражения-генераторы
- логические выражения, состоящие только из скалярных значений

Инструкции:

- простые и составные инструкции использовать нельзя (речь идет об инструкциях `for`, `while` и `if`).

Примеры использования функции eval()

Функция `pandas.eval()` хорошо работает с выражениями, содержащими большие массивы.
Сначала давайте создадим несколько больших массивов для нашего примера:

```
In[16]:
nrows, ncols = 20000, 100
df1, df2, df3, df4 = [pd.DataFrame(np.random.randn(nrows, ncols)) for _ in range(4)]

In[17]:
%timeit df1 + df2 + df3 + df4

10 loops, best of 3: 31 ms per loop

In[18]:
%timeit pd.eval('df1 + df2 + df3 + df4')

10 loops, best of 3: 16.2 ms per loop
```

Теперь выполним ту же самую операцию, добавив сравнения.

```
In[19]:
%timeit (df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0)

10 loops, best of 3: 67 ms per loop


In[20]:
%timeit pd.eval('(df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0)')

10 loops, best of 3: 23.1 ms per loop
```

Кроме того, функция `eval()` умеет работать с невыровненными объектами библиотеки pandas:

```
In[21]:
s = pd.Series(np.random.randn(50))
```

```
In[22]:
%timeit df1 + df2 + df3 + df4 + s

10 loops, best of 3: 52.3 ms per loop


In[23]:
%timeit pd.eval('df1 + df2 + df3 + df4 + s')

10 loops, best of 3: 17.4 ms per loop
```

Метод DataFrame.eval()

Помимо высокоуровневой функции `pandas.eval()` вы можете воспользоваться методом `DataFrame.eval()`.

```
In[24]:
np.random.seed(12345)
df = pd.DataFrame(np.random.randn(5, 2), columns=['a', 'b'])

In[25]:
df.eval('a + b')

Out[25]:
0    0.274236
1   -1.075169
2    3.359186
3    0.374654
4    2.015457
dtype: float64
```

Любое выражение, являющееся допустимым выражением `pandas.eval()`, также является допустимым выражением `DataFrame.eval()`. При этом у вас появляется дополнительное преимущество: вам не нужен префикс, задающий имя объекта `DataFrame` для столбца(ов), которые будут участвовать в ваших вычислениях.

Кроме того, вы можете присваивать столбцы объекту `DataFrame` внутри выражения. Речь может идти о новом имени столбца или существующем имени столбца, и оно должно быть допустимым идентификатором Python.

Новое в версии 0.18.0.

Ключевое слово `inplace` определяет, будет ли операция присваивания выполнена над исходным объектом `DataFrame` или нужно вернуть копию с новым столбцом, оставив исходный объект `DataFrame` неизменным.
Внимание! В целях обратной совместимости по умолчанию для параметра `inplace` используется значение `True`, если не указано иное. Это изменится в будущей версии pandas, поэтому если ваш код требует присваивания на месте, явно задайте `inplace=True`.

In[26]:
```
df = pd.DataFrame(dict(a=range(5), b=range(5, 10)))
```

In[27]:
```
df.eval('c = a + b', inplace=True)
```

In[28]:
```
df.eval('d = a + b + c', inplace=True)
```

In[29]:
```
df.eval('a = 1', inplace=True)
```

In[30]:
```
df
```

	a	b	c	d
0	1	5	5	10
1	1	6	7	14
2	1	7	9	18
3	1	8	11	22
4	1	9	13	26

Когда для параметра `inplace` задано значение `False`, возвращается копия объекта `DataFrame` с новыми или модифицированными столбцами, а исходный объект `DataFrame` остается неизменным.

In[31]:
```
df
```

	a	b	c	d
0	1	5	5	10
1	1	6	7	14
2	1	7	9	18
3	1	8	11	22
4	1	9	13	26

In[32]:
```
df.eval('e = a - c', inplace=False)
```

	a	b	c	d	e
0	1	5	5	10	-4
1	1	6	7	14	-6
2	1	7	9	18	-8
3	1	8	11	22	-10
4	1	9	13	26	-12

In[33]:
```
df
```

	a	b	c	d
0	1	5	5	10
1	1	6	7	14
2	1	7	9	18
3	1	8	11	22
4	1	9	13	26

Новое в версии 0.18.0.

Для удобства несколько операций присваивания можно выполнить с помощью набора строк.

In[34]:
```
df = pd.DataFrame(dict(a=range(5), b=range(5, 10)))
df.eval("""
c = a + b
d = a + b + c
a = 1""", inplace=False)
```

	a	b	c	d
0	1	5	5	10
1	1	6	7	14
2	1	7	9	18
3	1	8	11	22
4	1	9	13	26

Это будет эквивалентно стандартному коду Python, приведенному ниже:

In[35]:
```
df = pd.DataFrame(dict(a=range(5), b=range(5, 10)))
df['c'] = df.a + df.b
df['d'] = df.a + df.b + df.c
df['a'] = 1
df
```

	a	b	c	d
0	1	5	5	10
1	1	6	7	14
2	1	7	9	18
3	1	8	11	22
4	1	9	13	26

ПРИЛОЖЕНИЕ 3. ИСПОЛЬЗУЕМ PANDAS ДЛЯ БОЛЬШИХ ДАННЫХ

Материал подготовлен специалистом по работе с данными компании Dataquest.io Джошем Девлиным.

Когда мы работаем в библиотеке pandas с небольшими данными (до 100 мегабайт) производительность редко бывает проблемой. Когда мы переходим к более крупным данным (от 100 мегабайт до нескольких гигабайт), проблемы с производительностью могут увеличить время вычислений и привести к ошибкам из-за нехватки памяти.
Хотя такие инструменты, как Spark, могут обрабатывать большие наборы данных (от 100 гигабайт до нескольких терабайт), чтобы в полной мере использовать их возможности, обычно требуется более дорогое оборудование. И в отличие от pandas у них нет богатого набора высококачественных функций для очистки, исследования и анализа данных. При работе с данными среднего размера нам лучше попытаться максимально продуктивно использовать возможности библиотеки pandas, а не переключаться на другой инструмент.
В этом разделе мы узнаем об использовании памяти в библиотеке pandas, о том, как уменьшить объем датафрейма в памяти на 90%, просто выбрав соответствующие типы данных для столбцов.

Работаем с данными бейсбольных игр

Мы будем работать с данными, собранными Главной лигой бейсбола за последние 130 лет.
Первоначально данные были записаны в 127 отдельных CSV-файлах, однако мы использовали `csvkit` для объединения файлов и добавили имена столбцов в качестве первой строки. Если вы хотите загрузить нашу версию данных, ее можно скачать здесь https://data.world/dataquest/mlb-game-logs.
Давайте начнем с импорта наших данных и взглянем на первые пять строк.

In[1]:
```
# отключаем предупреждения Anaconda
import warnings
warnings.simplefilter('ignore')

# импортируем библиотеку pandas
import pandas as pd

# считываем данные и выводим первые 5 наблюдений
gl = pd.read_csv('Data/game_logs.csv')
gl.head()
```

Out[1]:

	date	number_of_game	day_of_week	v_name	v_league	v_game_number	h_name
0	18710504	0	Thu	CL1	na	1	FW1
1	18710505	0	Fri	BS1	na	1	WS3
2	18710506	0	Sat	CL1	na	2	RC1
3	18710508	0	Mon	CL1	na	3	CH1
4	18710509	0	Tue	BS1	na	2	TRO

Ниже мы привели некоторые важные столбцы, однако, если вы хотите посмотреть информацию по всем столбцам, мы создали словарь данных для всего набора данных:

- `date` – дата игры;
- `v_name` – название гостевой команды;
- `v_league` – лига гостевой команды;
- `h_name` – название домашней команды;
- `h_league` – лига домашней команды;
- `v_score` – счет гостевой команды;
- `h_score` – счет домашней команды;
- `v_line_score` – счет линии[8] гостевой команды, например, 010000(10)00;
- `h_line_score` – счет линии домашней команды, например, 010000(10)0X;
- `park_id` – ID парка, где проводилась игра;
- `attendance` – посещаемость матча.

Мы можем воспользоваться методом `.info()`, чтобы получить основную сведения о нашем датафрейме, включая его размер, информацию о типах данных и использовании памяти.

По умолчанию pandas вычисляет лишь приблизительный объем памяти, используемый датафреймом, в целях экономии времени. Поскольку нас интересует точность, мы установим для параметр `memory_usage` значение `'deep'`, чтобы получить точную информацию.

In[2]:
```
# выводим точную информацию об использовании памяти
gl.info(memory_usage='deep')
```
Out[2]:
```
<class 'pandas.core.frame.DataFrame'>
```

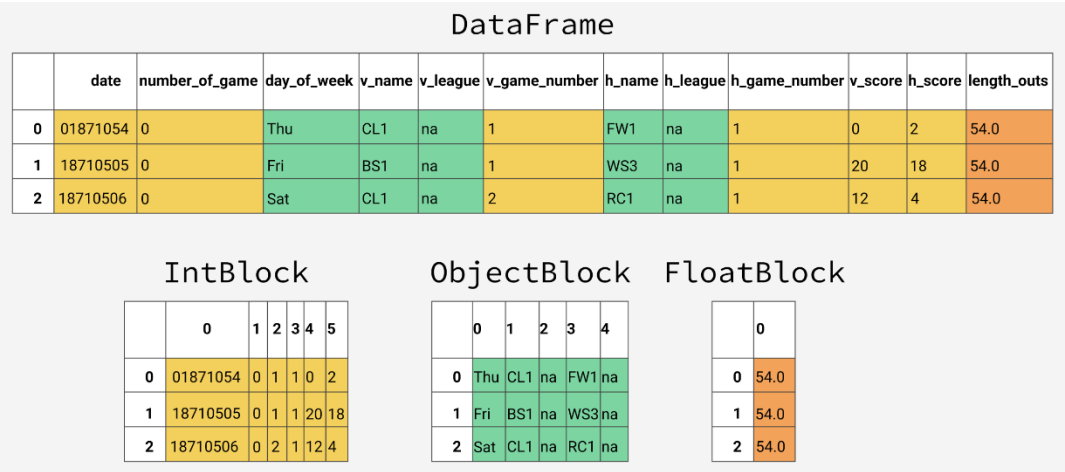
[8] Счет линии – это обычно диаграмма с двумя линиями, которая сообщает об общих количествах пробега каждой команды подачей, и полных пробегах, полных хитах и полных ошибках на линии. – *Прим. пер.*

```
RangeIndex: 171907 entries, 0 to 171906
Columns: 161 entries, date to acquisition_info
dtypes: float64(77), int64(6), object(78)
memory usage: 860.5 MB
```

Мы видим, что у нас 171907 строк и 161 столбец. Библиотека pandas автоматически определяет типы столбцов: у нас 77 столбцов типа `float`, 6 столбцов типа `int` и 78 столбцов типа `object`. Столбцы `object` используются для представления строковых значений или там, где столбец содержит смешанные типы данных.
Попытаемся лучше понять, как мы можем уменьшить использование памяти. Давайте посмотрим на то, как библиотека pandas хранит данные в памяти.

Внутреннее представление датафрейма

Под капотом библиотека pandas группирует столбцы в блоки значений одного и того же типа. Вот пример того, каким образом pandas будет хранить первые двенадцать столбцов нашего датафрейма.



	date	number_of_game	day_of_week	v_name	v_league	v_game_number	h_name	h_league	h_game_number	v_score	h_score	length_outs
0	01871054	0	Thu	CL1	na	1	FW1	na	1	0	2	54.0
1	18710505	0	Fri	BS1	na	1	WS3	na	1	20	18	54.0
2	18710506	0	Sat	CL1	na	2	RC1	na	1	12	4	54.0

	0	1	2	3	4	5
0	01871054	0	1	1	0	2
1	18710505	0	1	1	20	18
2	18710506	0	2	1	12	4

	0	1	2	3	4
0	Thu	CL1	na	FW1	na
1	Fri	BS1	na	WS3	na
2	Sat	CL1	na	RC1	na

	0
0	54.0
1	54.0
2	54.0

Рис. 1 Классы для представления значений разных типов

Заметьте, что блоки не хранят ссылки на имена столбцов. Это связано с тем, что блоки оптимизированы для хранения фактических значений в датафрейме. Класс `BlockManager` ответственен за сопоставление индексов строк/столбцов фактическим блокам. Он работает как API, который обеспечивает доступ к базовым данным. Всякий раз, когда мы выбираем, редактируем или удаляем значения, класс `dataframe` взаимодействует с классом `BlockManager` для преобразования наших запросов в вызовы функций и методов.
Каждый тип представлен специальным классом в модуле `pandas.core.internals`. Pandas использует класс `ObjectBlock` для представления блока, содержащего столбцы со строковыми значениями, класс `FloatBlock` для представления блока, содержащего числа с плавающей точкой, и класс `IntBlock` для представления блока, содержащего целые числа. Для работы с блоками, представляющими

числовые значения (целые числа и числа с плавающей точкой) pandas объединяет столбцы и сохраняет их как многомерный массив NumPy. Многомерный массив NumPy организован на основе массива C и значения хранятся в непрерывном блоке памяти. Благодаря этой схеме хранения доступ к срезу значений осуществляется невероятно быстро.
Поскольку каждый тип данных хранится отдельно, мы рассмотрим использование памяти в зависимости от типа данных. Давайте посмотрим, сколько памяти использует каждый тип данных в среднем.

```
In[3]:
# посмотрим, сколько памяти в среднем используют
# столбцы определенного типа
for dtype in ['float','int','object']:
    selected_dtype = gl.select_dtypes(include=[dtype])
    mean_usage_b = selected_dtype.memory_usage(deep=True).mean()
    mean_usage_mb = mean_usage_b / 1024 ** 2
    print("Использование памяти в среднем для {} столбцов: {:03.2f} MB".
          format(dtype,mean_usage_mb))

Out[3]:
Использование памяти в среднем для float столбцов: 1.29 MB
Использование памяти в среднем для int столбцов: 0.00 MB
Использование памяти в среднем для object столбцов: 9.51 MB
```

Сразу видно, что большая часть нашей памяти используется 78 столбцами типа `object`. Мы рассмотрим их позже, а сейчас давайте посмотрим, можем ли мы улучшить использование памяти для наших числовых столбцов.

Подтипы

Как мы уже упоминали ранее, под капотом библиотека pandas представляет числовые значения как многомерные массивы NumPy и сохраняет их в непрерывном блоке памяти. Эта модель хранения занимает меньше места и позволяет нам быстро получить доступ к этим значениям. Поскольку pandas представляет каждое значение одного и того же типа, используя одинаковое количество байтов, а многомерный массив NumPy хранит информацию о количестве значений, библиотека pandas может быстро и точно вернуть количество байтов, занимаемое числовым столбцом.
Многие типы в библиотеке pandas имеют несколько подтипов, которые могут использовать меньшее количество байтов для представления каждого значения. Например, тип `float` имеет подтипы `float16`, `float32` и `float64`. Число, следующее после названия типа, указывает количество битов, которое используется данным типом для представления значений. Например, подтипы, которые мы только что указали, используют соответственно `2`, `4`, `8` и `16` байтов. В следующей таблице показаны подтипы для наиболее распространенных типов библиотеки pandas:

| использование памяти | float | int | uint | datetime | bool | object |

| - | - | - | - | - | - |
1 байт		int8	uint8		bool	
2 байта	float16	int16	uint16			
4 байта	float32	int32	uint32			
8 байт	float64	int64	uint64	datetime64		
переменная					объект

Тип `int8` использует `1` байт (или `8` бит) для хранения значения и может представлять `256` значений (2^8) в двоичном формате. Это означает, что мы можем использовать этот подтип для представления значений от `-128` до `127` (включая `0`).
Мы можем воспользоваться классом `numpy.iinfo`, чтобы посмотреть минимальное и максимальное значения для каждого целочисленного подтипа. Давайте взглянем на пример:

```
In[4]:
# импортируем библиотеку numpy
import numpy as np

# посмотрим минимальное и максимальное значения для
# каждого целочисленного типа
int_types = ["uint8", "int8", "int16"]
for it in int_types:
    print(np.iinfo(it))

Out[4]:
Machine parameters for uint8
---------------------------------------------------------------
min = 0
max = 255
---------------------------------------------------------------

Machine parameters for int8
---------------------------------------------------------------
min = -128
max = 127
---------------------------------------------------------------

Machine parameters for int16
---------------------------------------------------------------
min = -32768
max = 32767
---------------------------------------------------------------
```

Здесь мы видим разницу между типами `uint` (`unsigned integer` – целые числа без знака) и `int` (`signed integer` – целые числа со знаком). Оба типа имеют одинаковую емкость хранения, но при этом целые числа без знака, представляющие только положительные значения, позволяют нам хранить столбцы с положительными значениями более эффективно.

Оптимизация числовых столбцов с помощью понижающего преобразования

Мы можем использовать функцию `pd.to_numeric()`, чтобы выполнить понижающее преобразование наших числовых типов[9]. Мы воспользуемся `DataFrame.select_dtypes` для выбора только целочисленных столбцов, тогда мы будем оптимизировать типы и сравнить использование памяти.

In[5]:

```
# мы будем довольно часто подсчитывать использование памяти,
# поэтому напишем функцию, которая сэкономит нам немного времени

def mem_usage(pandas_obj):
    if isinstance(pandas_obj,pd.DataFrame):
        usage_b = pandas_obj.memory_usage(deep=True).sum()
    else: # предположим, что если это не датафрейм, то серия
        usage_b = pandas_obj.memory_usage(deep=True)
    usage_mb = usage_b / 1024 ** 2 # преобразуем байты в мегабайты
    return "{:03.2f} MB".format(usage_mb)

# выполняем понижающее преобразование
# для столбцов типа int
gl_int = gl.select_dtypes(include=['int'])
converted_int = gl_int.apply(pd.to_numeric,downcast='unsigned')

print(mem_usage(gl_int))
print(mem_usage(converted_int))

compare_ints = pd.concat([gl_int.dtypes,converted_int.dtypes],axis=1)
compare_ints.columns = ['before','after']
compare_ints.apply(pd.Series.value_counts)

7.87 MB
1.48 MB
```

Out[5]:

	BEFORE	AFTER
UINT8	NaN	5.0
UINT32	NaN	1.0
INT64	6.0	NaN

Мы видим снижение использования памяти с 7,9 до 1,5 мегабайт, что составляет более 80%. Это не сильно повлияло на наш датафрейм из-за небольшого количества столбцов типа `int`.

Давайте сделаем то же самое с нашими столбцами `float`.

[9] Речь идет о понижающем преобразовании (downcasting) – операции приведения типа, ссылающегося на базовый класс, к одному из его производных классов. В данном случае речь идет о понижающем приведении типа `int64` к типам `uint8` и `uint32` – *Прим. пер.*

In[6]:

```
# выполняем понижающее преобразование
# для столбцов типа float
gl_float = gl.select_dtypes(include=['float'])
converted_float = gl_float.apply(pd.to_numeric,downcast='float')

print(mem_usage(gl_float))
print(mem_usage(converted_float))

compare_floats = pd.concat([gl_float.dtypes,converted_float.dtypes],axis=1)
compare_floats.columns = ['before','after']
compare_floats.apply(pd.Series.value_counts)

100.99 MB
50.49 MB
```

Out[6]:

	BEFORE	AFTER
float32	NaN	77.0
float64	77.0	NaN

Мы видим, что все наши столбцы типа `float` были преобразованы из `float64` во `float32`, что дает нам 50%-ное сокращение использования памяти.

Давайте создадим копию нашего исходного датафрейма, заменив исходные числовые столбцы оптимизированными числовыми столбцами, и посмотрим, каким будет общее использование памяти.

In[7]:

```
# создаем копию исходного датафрейма
optimized_gl = gl.copy()

# заменяем исходные числовые столбцы
# оптимизированными
optimized_gl[converted_int.columns] = converted_int
optimized_gl[converted_float.columns] = converted_float

# смотрим использование памяти
print(mem_usage(gl))
print(mem_usage(optimized_gl))

860.50 MB
810.01 MB
```

Хотя мы значительно сократили использование памяти нашими числовыми столбцами, в целом же мы уменьшили потребление памяти нашим датафреймом только на 7%. В значительной мере снижение использования памяти будет зависеть от оптимизации столбцов типа `object`.

Прежде чем мы это сделаем, давайте подробнее сравним способы хранения строковых и числовых значений в библиотеке pandas.

Сравнение способов хранения числовых и строковых значений

Тип `object` представляет значения, использующие питоновские объекты-строки, отчасти это обусловлено отсутствием поддержки пропущенных строковых значений в NumPy. Поскольку Python

является высокоуровневым, интерпретируемым языком, он не предполагает точной настройки способа хранения значений в памяти. Это ограничение приводит к тому, что строки хранятся фрагментированно, это потребляет больше памяти и замедляет доступ. Каждый элемент в столбце типа `object` является по сути указателем, который содержит «адрес» фактического значения в памяти.
Ниже приведена диаграмма, показывающая, как хранятся числовые данные в NumPy и как хранятся строки с использованием встроенных типов Python.

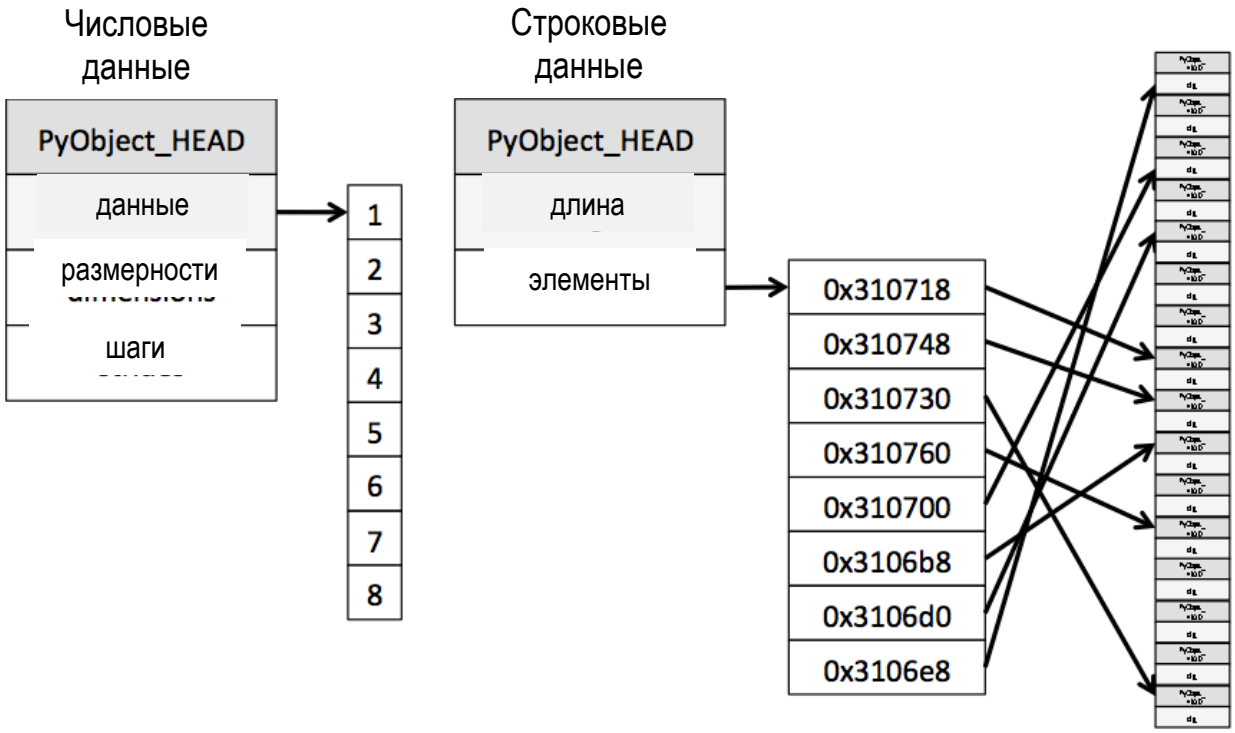


Рис. 2 Способы хранения числовых и строковых значений

Возможно, вы заметили, что наша диаграмма описала типы `object` как использующие разный объем памяти. Хотя каждый указатель занимает 1 байт памяти, каждое фактическое строковое значение использует такой объем памяти, какой строка использовала бы, если бы отдельно хранилась в Python. Чтобы доказать это, давайте воспользуемся `sys.getsizeof()`, сначала посмотрев размер отдельно хранящихся строк, а затем размер строк как элементов серии.

In[8]:
```
# смотрим размеры отдельно
# хранящихся строк
from sys import getsizeof

s1 = 'working out'
s2 = 'memory usage for'
s3 = 'strings in python is fun!'
s4 = 'strings in python is fun!'

for s in [s1, s2, s3, s4]:
    print(getsizeof(s))
```

```
60
```

```
65
74
74
```

```
In[9]:
# смотрим размеры строк, являющихся
# элементами серии
obj_series = pd.Series(['working out',
                        'memory usage for',
                        'strings in python is fun!',
                        'strings in python is fun!'])
obj_series.apply(getsizeof)
```

```
0    60
1    65
2    74
3    74
dtype: int64
```

Видно, что размер строк, хранящихся в объекте `Series`, идентичен размеру этих же строк, хранящихся отдельно в Python.

Оптимизация типов object с помощью типа category

Тип `category` под капотом для представления значений в столбце вместо исходных использует целочисленные значения. Для этого создается отдельный словарь, в котором исходным значениям сопоставлены целочисленные значения. Это сопоставление будет полезно там, где столбец содержит ограниченный набор значений. Когда мы конвертируем столбец в тип `category`, pandas использует самый эффективный с точки зрения занимаемого места подтип `int`, который может представить все уникальные значения в столбце.

Чтобы выяснить, в каких случаях можно воспользоваться этим типом для сокращения использования памяти, давайте посмотрим количество уникальных значений в каждом столбце типа `object`.

In[10]:

```
# смотрим количество уникальных значений
# по каждому столбцу типа object
gl_obj = gl.select_dtypes(include=['object']).copy()
gl_obj.describe()
```

Out[10]:

	day_of_week	v_name	v_league	h_name	h_league	day_night	completion	forefeit	protest	park_id
count	171907	171907	171907	171907	171907	140150	116	145	180	171907
unique	7	148	7	148	7	2	116	3	5	245
top	Sat	CHN	NL	CHN	NL	D	19810610,CHI11,1,2,45	H	V	STL07
freq	28891	8870	88866	9024	88867	82724	1	69	90	7022

Беглый просмотр показывает, что у нас много столбцов, содержащих несколько уникальных значений, по сравнению с общим возможным количеством значений, равным примерно 172000 (поскольку набор содержит примерно 172000 наблюдений).

Прежде чем погрузиться в детали, мы просто выберем один из столбцов типа `object` и посмотрим, что происходит под капотом, когда мы преобразуем его в категориальный тип. Мы воспользуемся вторым столбцом нашего набора данных под названием `day_of_week`.

Взглянув на таблицу выше, мы видим, что он содержит лишь семь уникальных значений. Мы преобразуем его в тип `category`, используя метод `.astype()`.

```
In[11]:
# запишем столбец day_of_week как dow
dow = gl_obj.day_of_week
print(dow.head())

# присваиваем столбцу dow тип category
dow_cat = dow.astype('category')
print(dow_cat.head())

0    Thu
1    Fri
2    Sat
3    Mon
4    Tue
Name: day_of_week, dtype: object
0    Thu
1    Fri
2    Sat
3    Mon
4    Tue
Name: day_of_week, dtype: category
Categories (7, object): [Fri, Mon, Sat, Sun, Thu, Tue, Wed]
```

Видно, что хотя тип изменился, данные выглядят точно так же. Давайте посмотрим, что происходит под капотом.
В следующем программном коде мы воспользуемся атрибутом `Series.cat.codes`. Он возвращает целочисленные значения, использующиеся типом `category` для представления каждого значения.

```
In[12]:
# выводим целочисленные значения,
# соответствующие исходным значениям
dow_cat.head().cat.codes

Out[12]:
0    4
1    0
2    2
3    1
4    5
dtype: int8
```

Видно, что каждому уникальному значению присваивается целое число, а базовым типом данных для столбца будет тип `int8`. Этот столбец не имеет пропусков, но если они будут, подтип `category` обрабатывает пропущенные значения, присвоив им значение `-1`.
Наконец, давайте взглянем, сколько памяти использует этот столбец до и после преобразования в тип `category`.

```
In[13]:
# смотрим, сколько памяти использует столбец dow
# до и после преобразования в тип category
print(mem_usage(dow))
print(mem_usage(dow_cat))

9.84 MB
0.16 MB
```

Мы снизили использование памяти с 9,8 МБ до 0,16 МБ использования памяти или на 98%! Обратите внимание, что этот столбец, вероятно, представляет собой наилучший пример того, как можно снизить

использование памяти – столбец с примерно 172000 записями и только 7 уникальными значениями.
Преобразование всех столбцов в тип `category` – привлекательная идея, однако важно помнить про ограничения, связанные с этой операцией. Самое большое ограничение – невозможность вычислений с помощью численных методов. Мы не можем выполнять арифметические операции со столбцами типа `category` или использовать такие методы, как `.min()` и `.max()`, не присвоив столбцам настоящий числовой тип.
Мы должны придерживаться типа `category` преимущественно при работе с такими столбцами `object`, в которых менее 50% значений являются уникальными. Если все значения в столбце являются уникальными, тип `category` в конечном итоге будет использовать *больший* объем памяти. Это обусловлено тем, что в столбце помимо целочисленных кодов, представляющих категории, хранятся все исходные строковые значения. Подробно об ограничениях, связанных с использованием типа `category`, читайте в документации по библиотеке pandas http://pandas.pydata.org/pandas-docs/stable/categorical.html.
Мы напишем цикл, которой переберет каждый столбец `object`, проверит его на соответствие заданному порогу (количество уникальных значений должно быть меньше 50% от общего количества значений), и если столбец удовлетворяет порогу, преобразует его в тип `category`.

In[14]:
```
converted_obj = pd.DataFrame()

# пишем цикл, которой перебирает каждый столбец object,
# проверяет его на соответствие заданному порогу
# (количество уникальных значений должно быть меньше 50%
# от общего количества значений), и если столбец
# удовлетворяет порогу, преобразовывает его в тип category
for col in gl_obj.columns:
    num_unique_values = len(gl_obj[col].unique())
    num_total_values = len(gl_obj[col])
    if num_unique_values / num_total_values < 0.5:
        converted_obj.loc[:,col] = gl_obj[col].astype('category')
    else:
        converted_obj.loc[:,col] = gl_obj[col]
```

Вновь применим нашу функцию `mem_usage`.

In[15]:
```
# снова применяем функцию mem_usage, смотрим,
# сколько памяти занимают все столбцы типа object
# до и после преобразования в тип category
print(mem_usage(gl_obj))
print(mem_usage(converted_obj))

compare_obj = pd.concat([gl_obj.dtypes,converted_obj.dtypes],axis=1)
compare_obj.columns = ['before','after']
compare_obj.apply(pd.Series.value_counts)
```

```
752.72 MB
51.67 MB
```

Out[15]:

	BEFORE	AFTER
object	78.0	NaN
category	NaN	78.0

В данном случае все наши столбцы типа `object` были преобразованы в тип `category`, однако данную операцию нельзя считать универсальной процедурой, которую можно применить ко всем наборам данных, поэтому вы должны с осторожностью применять ее.
Использование памяти нашими столбцами `object` снизилось с 752 МБ до 52 МБ или на 93%. Давайте объединим их с остальной частью нашего датафрейма и посмотрим, как теперь изменилось использование памяти по сравнению с начальными 861 МБ.

In[16]:
```
# смотрим, сколько памяти использует датафрейм
# после оптимизации типов
optimized_gl[converted_obj.columns] = converted_obj
mem_usage(optimized_gl)
```

Out[16]:
```
'103.64 MB'
```

Ого, мы действительно добились определенного прогресса! Кроме того, мы можем выполнить еще одну оптимизацию. Существует тип `datetime`, который можно использовать для представления значений, записанных в первом столбце нашего набора данных.

In[17]:
```
# смотрим, сколько памяти использует
# столбец date
date = optimized_gl.date
print(mem_usage(date))
date.head()
```

```
0.66 MB
```

Out[17]:
```
0    18710504
1    18710505
2    18710506
3    18710508
4    18710509
Name: date, dtype: int64
```

Вы, возможно, помните, что этот столбец был прочитан как целочисленный тип и уже оптимизирован как подтип `unint32`. В силу этого преобразование в тип `datetime` фактически в два раза увеличит использование памяти, поскольку тип `datetime` является 64-битным типом. В любом случае имеет смысл преобразовать этот столбец в тип `datetime`, поскольку это позволит нам более легко проводить анализ временных рядов.
Мы преобразуем его с помощью функции `pandas.to_datetime()`, используя параметр `format`, чтобы задать формат хранения наших дат `YYYY-MM-DD`.

```
In[18]:
# преобразуем столбец
# date в тип datetime
optimized_gl['date'] = pd.to_datetime(date, format='%Y%m%d')

# смотрим, сколько памяти использует
# столбец date
print(mem_usage(optimized_gl))
optimized_gl.date.head()

104.29 MB

Out[18]:
0   1871-05-04
1   1871-05-05
2   1871-05-06
3   1871-05-08
4   1871-05-09
Name: date, dtype: datetime64[ns]
```

Задаем типы во время считывания данных

До сих пор мы рассматривали способы уменьшения объема памяти, работая с **уже существующим** датафреймом. Сначала считывая датафрейм, а затем перебирая разные способы уменьшения использования памяти, мы смогли выяснить, какой объем памяти мы можем выиграть, применив ту или иную оптимизацию. Однако, как мы уже говорили ранее, нам часто не хватает памяти для представления всех значений в наборе данных. Как мы можем применить методы оптимизации памяти, если мы изначально даже не можем создать датафрейм?

К счастью, мы можем указать оптимальные типы столбцов в ходе считывания набора данных. Функция `read_csv()` имеет несколько параметров, которые позволяют нам это делать. Параметр `dtype` принимает словарь, в котором ключами будут строковые имена столбцов, а значениями – типы столбцов.

Сначала мы сохраним итоговые типы столбцов в словаре, предварительно удалив столбец с датами, поскольку его нужно обрабатывать отдельно.

In[19]:
```
# создаем словарь, в котором ключами будут имена
# столбцов, а значениями - типы столбцов
dtypes = optimized_gl.drop('date',axis=1).dtypes

dtypes_col = dtypes.index
dtypes_type = [i.name for i in dtypes.values]

column_types = dict(zip(dtypes_col, dtypes_type))

# вместо того, чтобы печатать 161 столбец,
# выберем 10 пар "ключ-значение" из словаря
# и красиво распечатаем их, используя prettyprint
preview = first2pairs = {key:value for key,value in list(column_types.items())[:10]}
import pprint
pp = pp = pprint.PrettyPrinter(indent=4)
pp.pprint(preview)
```

Out[19]:
```
{   'day_of_week': 'category',
    'h_game_number': 'int64',
    'h_league': 'category',
    'h_name': 'category',
    'h_score': 'int64',
    'number_of_game': 'int64',
    'v_game_number': 'int64',
    'v_league': 'category',
    'v_name': 'category',
    'v_score': 'int64'}
```

Теперь мы можем воспользоваться словарем, а также несколькими параметрами, предназначенными для обработки дат, чтобы прочитать данные с нужными нам типами. Для этого нам потребуется написать несколько строк:

In[20]:
```
# считываем данные с нужными нам типами
read_and_optimized = pd.read_csv('Data/game_logs.csv',
                                 dtype=column_types,
                                 parse_dates=['date'],
                                 infer_datetime_format=True)
# смотрим использование памяти
print(mem_usage(read_and_optimized))
read_and_optimized.head()
```

```
104.28 MB
```

Out[20]:

	date	number_of_game	day_of_week	v_name	v_league	v_game_number	h_name	h_league
0	1871-05-04	0	Thu	CL1	na	1	FW1	na
1	1871-05-05	0	Fri	BS1	na	1	WS3	na
2	1871-05-06	0	Sat	CL1	na	2	RC1	na
3	1871-05-08	0	Mon	CL1	na	3	CH1	na
4	1871-05-09	0	Tue	BS1	na	2	TRO	na

Оптимизировав типы столбцов, нам удалось сократить использование памяти в pandas с 861,6 МБ до 104,28 МБ – впечатляющее сокращение на 88%!

Выводы

Мы выяснили, как библиотека pandas хранит данные с помощью разных типов, и затем воспользовались полученными знаниями, чтобы уменьшить использование памяти нашим датафреймом почти на 90%, просто применив несколько простых методов:

- понижающее приведение числовых столбцов к более эффективным типам;
- преобразование столбцов типа `object` в столбцы типа `category`.

ПРИЛОЖЕНИЕ 4. ПРИМЕР ПРЕДВАРИТЕЛЬНОЙ ПОДГОТОВКИ ДАННЫХ В PANDAS (КОНКУРСНАЯ ЗАДАЧА TINKOFF DATA SCIENCE CHALLENGE)

Материал подготовлен директором по научной работе исследовательского центра «Гевисста» Артемом Груздевым.

До 80% времени аналитика тратится на предварительную подготовку данных. О ней мы и поговорим ниже. Необходимые нам данные записаны в файле *credit_train.csv*. Речь идет о задаче выбора кредита Tinkoff.ru в рамках соревнования Tinkoff Data Science Challenge. Исходная выборка содержит записи о 170746 клиентах, классифицированных на два класса: 0 – клиент не открыл кредитный счет в банке (140690 клиентов) и 1 – клиент открыл кредитный счет в банке (30056 клиентов). Необходимо классифицировать клиентов на тех, кто не откроет кредитного счета, и тех, кто его откроет. По каждому наблюдению (клиенту) фиксируются 15 исходных переменных.
Список исходных переменных включает в себя:

- категориальный предиктор *Идентификационный номер [client_id]*;
- категориальный предиктор *Пол [gender]*;
- количественный предиктор *Возраст [age]*;
- категориальный предиктор *Семейный статус [marital_status]*;
- категориальный предиктор *Сфера занятости [job_position]*;
- количественный предиктор *Сумма кредита [credit_sum];*
- количественный предиктор *Срок кредитования [credit_month]*;
- количественный предиктор *Внутренняя скоринговая оценка [score_shk]*;
- категориальный предиктор *Образование [education]*;
- категориальный предиктор *Идентификационный номер тарифа [tariff_id]*;
- количественный предиктор *Месячный заработок [monthly_income]*;
- количественный предиктор *Количество кредитов у клиента [credit_count]*;
- количественный предиктор *Количество просроченных кредитов у клиента [overdue_credit_count]*;
- категориальная зависимая переменная *Факт открытия кредитного счета в данном банке [open_account_flg]*.

Считывание CSV-файла в объект DataFrame

Сначала импортируем необходимые библиотеки.

```
In[1]:
# импортируем необходимые библиотеки
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import re
```

Записываем файл *credit_train.csv* в датафрейм `data` с помощью функции `pd.read_csv()`. У нас в файле наряду с латиницей используется кириллица, поэтому с помощью параметра `encoding`, определяющего тип кодировки, задаем значение `'cp1251'`. Поскольку в этом CSV-файле в качестве символа-разделителя используется точка с запятой, с помощью параметра `sep`, определяющего тип символа-разделителя, задаем значение `';'`. Если бы в качестве символа-разделителя использовалась запятая, мы бы задали значение `','`.

```
In[2]:
# записываем CSV-файл в объект DataFrame
data = pd.read_csv('Data/credit_train.csv', encoding='cp1251', sep=';')
```

ДОПОЛНИТЕЛЬНАЯ ИНФОРМАЦИЯ

Если в объект `DataFrame` нужно считать файл Excel, необходимо воспользоваться функцией `pd.read_excel()`, указав с помощью параметра `sheetname` название нужного рабочего листа. Например, так:

```
data=pd.read_excel("C:/Trees/Credit.xlsx", sheetname='Credit')
```

Выведем информацию о форме датафрейма с помощью свойства `shape`:

```
In[3]:
# смотрим форму датафрейма
data.shape
```

```
(170746, 15)
```

Видим, что датафрейм состоит из 170746 строк (наблюдений) и 15 столбцов (переменных). Теперь взглянем на первые 5 наблюдений нашего набора с помощью метода `.head()`:

In[4]:

```
# выводим первые 5 наблюдений датафрейма
data.head()
```

Out[4]:

	client_id	gender	age	marital_status	job_position	credit_sum	credit_month	tariff_id	score_shk	education	living_region	monthly_income
0	1	M	NaN	NaN	UMN	59998,00	10	1.6	NaN	GRD	КРАСНОДАРСКИЙ КРАЙ	30000.0
1	2	F	NaN	MAR	UMN	10889,00	6	1.1	NaN	NaN	МОСКВА	NaN
2	3	M	32.0	MAR	SPC	10728,00	12	1.1	NaN	NaN	ОБЛ САРАТОВСКАЯ	NaN
3	4	F	27.0	NaN	SPC	12009,09	12	1.1	NaN	NaN	ОБЛ ВОЛГОГРАДСКАЯ	NaN
4	5	M	45.0	NaN	SPC	NaN	10	1.1	0,421385	SCH	ЧЕЛЯБИНСКАЯ ОБЛАСТЬ	NaN

Перед нами – CSV-файл, записанный в датафрейм `data`.
Обычно план работы с данными делится на два этапа: формирование выборки и предварительная обработка данных.

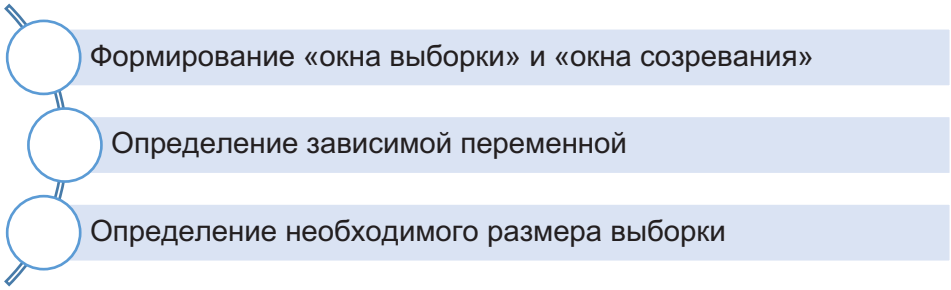


Рис. 1 План формирования выборки

Формирование выборки включает в себя три этапа: формирование «окна выборки» и «окна созревания», определение зависимой переменной и определение необходимого размера выборки.

Формирование «окна выборки» и «окна созревания»

Прогнозные модели разрабатываются, исходя из предположения «прошлое отражает будущее». На основе этого предположения мы анализируем поведение прошлых клиентов, чтобы спрогнозировать поведение будущих клиентов.
Для того, чтобы корректно выполнить этот анализ, нужно собрать необходимые данные о клиентах за определенный период времени, а затем осуществить мониторинг клиентов в течение другого определенного периода времени, оценив, были ли они «хорошими» или «плохими». Собранные данные (независимые переменные) наряду с соответствующей классификацией (зависимой переменной, которая принимает значение *Хороший* или *Плохой*) составят основу для разработки прогнозной модели.
Период времени, в ходе которого мы собираем данные о наших клиентах, т.е. фиксируем независимые переменные, называется «окном выборки» или «окном наблюдения» (sample window или observation period).
Период времени, в ходе которого мы отслеживаем статус клиента, т.е. фиксируем зависимую переменную, назвается «окном созревания»

(performance window или performance period). «Окно созревания» представляет собой тот промежуток времени, когда клиент имел возможность проявить себя: выйти в дефолт, отказаться от услуг компании. Его получают по данным винтажного анализа, выясняют момент времени, когда отмечается максимальный процент интересующих событий (например, максимальный процент просрочек 90+ возникает на 9-й месяц после выдачи кредитов и затем стабилизируется). Если «окно созревания» будет коротким, мы просто занизим уровень просрочек и получим неправдоподобную оптимистичную модель.

Обычно «окно созревания» следует после лага. Лаг представляет собой промежуток между «окном выборки» и «окном созревания». Он нужен нам, потому что разработанная модель применяется не сразу, требуется определенное время для ее внедрения.

Допустим, вы разрабатываете модель оттока для клиентов банка. Под оттоком понимается отказ клиента от услуг банка. У вас есть исторические данные с января 2013 года по декабрь 2015 года. В качества окна выборки для получения предикторов используем данные с января 2013 года по май 2015 года. Период с июля по декабрь 2015 года будет окном созревания. Месячный лаг между окном выборки и окном созревания будет использован в качестве периода, в течение которого мы вычислим прогнозы для новых клиентов в ходе внедрения модели.

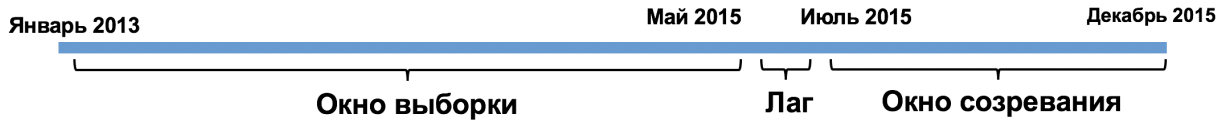


Рис. 2 Формирования «окна выборки» и «окна созревания»

Временной горизонт «созревания» зависит от продукта, определения «плохого» клиента и конкурентной среды. Счета по кредитным картам становятся «зрелыми» после 12-18 месяцев, в то время как счета сроком 2-3 года являются типичными для разработки скоринговой карты по ипотечным кредитам. Поведенческие скоринговые карты предусматривают окно созревания из расчета 6-12 месяцев. Коллекторские модели строятся, как правило, на данных одного месяца, но все чаще компании строят такие карты для более коротких временных интервалов – до двух недель, чтобы облегчить разработку более подходящих способов взыскания долгов. Когда речь идет о разработке прогнозных моделей в соответствии с определенными требованиями надзорных органов – например, в соответствии с требованиями соглашения Базель II – «окно созревания» может определяться регулирующим органом. В моделях оттока сотовых операторов «окно выборки» и «окно созревания» будут более короткими и составляют 6 месяцев.

Определение зависимой переменной

Выбор зависимой переменной определяется целью построения прогнозной модели. Цели могут быть общими, например, сокращение потерь по новым кредитным счетам, и конкретными, например, сокращение числа дефолтов по одобренным заявкам в течение 6 месяцев после принятия положительного решения.

Зависимая переменная может быть количественной и категориальной.

Примером количественной зависимой переменной является средняя сумма, которую погасит заемщик по просроченному кредиту.

В скоринге заявок зависимая переменная будет категориальной: погасит заемщик кредит («хороший») или не погасит («плохой»). Обычно к категории «плохой» относят клиентов, имеющих просроченную задолженность 90 дней и более. Этот период определяется требованиями банковского надзора. В соответствие с соглашением Базель II дефолт должника считается произошедшим, когда имело место одно или оба из следующих событий: банк считает, что должник не в состоянии полностью погасить свои кредитные обязательства перед банком без принятия банком таких мер, как реализация обеспечения (если таковое имеется); должник более чем на 90 дней просрочил погашение любых существенных кредитных обязательств перед банком.

Разумеется, банк может строить различные скоринговые карты с разными значениями зависимой переменной, вводя дополнительные критерии определения «плохого» и «хорошего» заемщика, а также меняя срок просрочки платежей. Примерами зависимой переменной могут быть наличие просроченной задолженности более 30 дней, 60 дней, 90 дней и более по одному кредиту на текущий момент или худший статус за все время кредитной истории, размер просроченной задолженности, количество просрочек более заданного числа дней и др.

Определение размера выборки

При определении минимального объема выборки опираются на следующие критерии: равномерность распределения категорий зависимой переменной, количество независимых переменных в модели, максимально допустимую ошибку выборки, экспертные оценки.

В практике банковского скоринга, когда используется случайное разбиение на обучающую и контрольную выборку или перекрестная проверка (роль обучающей выборки выполняют обучающие блоки, а роль контрольной выборки – контрольный блок), для ответа на вопрос о размере выборки часто используют правило «Number of Events Per Variable» (количество событий на одну переменную, NEPV), сформулированное Фрэнком Харреллом. Для задачи классификации оно связывает минимальный объем выборки с количеством «событий» – наблюдений в миноритарной (наименьшей по размеру) категории зависимой переменной и количеством предикторов, поданным на вход

модели. Согласно этому правилу, необходимо взять количество наблюдений в обучающей выборке, относящихся к миноритарной категории зависимой переменной (в кредитном скоринге это «плохие» заемщики). Это число наблюдений нужно разделить на количество заданных предикторов. Для регрессионной модели на один предиктор должно приходиться не менее 20 событий, при построении дерева решений CHAID на один предиктор должно приходиться не менее 50 событий, а для модели случайного леса, градиентного бустинга, SVM и нейронной сети на одну независимую переменную должно приходиться не менее 200 событий. Для задачи регрессии мы просто берем количество наблюдений и делим на количество предикторов и для регрессионной модели на одну независимую переменную должно приходиться не менее 20 наблюдений, для дерева решений CHAID (в тех случаях, когда реализация алгоритма позволяет решать задачу регрессии) на один предиктор должно приходиться не менее 50 наблюдений, для случайного леса и других сложных моделей на один предиктор должно приходиться не менее 200 наблюдений. По мнению Фрэнка Харрелла, для дерева решений CART правило NEPV невозможно сформулировать из-за высокой нестабильности метода и склонности к переобучению.

Если правило выполняется для обучающей выборки, то объем выборки для обучения является достаточным. В противном случае необходимо либо увеличить объем выборки, либо сократить количество предикторов, подаваемых на вход модели. Затем вся та же самая процедура применяется к контрольной выборке, если правило выполняется, объем выборки для проверки достаточен.

Таким образом, когда вы проектируете выборку, важно заранее позаботиться, чтобы при разбиении выборки на обучающую и контрольную, объем обучающей выборки был достаточен для обучения, а объем контрольной выборки достаточен для проверки.

Применительно к нашему случаю выборка уже сформирована, зависимая переменная определена и правило NEPV выполняется.

Особенности плана предварительной подготовки данных

План предварительной подготовки данных зависит от стратегии проверки модели. Если используются такие операции, как укрупнение редких категорий по порогу, импутация пропусков статистиками, стандартизация, биннинг и конструирование признаков на основе статистик (frequency encoding, likelihood encoding), предполагающие вычисления по набору данных, они должны быть осуществлены после разбиения на обучающую и контрольную выборки или внутри цикла перекрестной проверки.

Если мы используем случайное разбиение на обучающую и контрольную выборки и выполняем перечисленные операции до разбиения, получается, что для вычисления статистик, среднего и стандартного отклонения по каждому признаку для стандартизации, правил биннинга, частот и вероятностей положительного класса зависимой переменной в категориях предиктора использовались все наблюдения набора, часть из которых потом у нас войдут в контрольную выборку (по сути выборку новых данных).

Если мы используем перекрестную проверку и выполняем перечисленные операции до перекрестной проверки, получается, что в каждом проходе перекрестной проверки для вычисления статистик, среднего и стандартного отклонения по каждому признаку для стандартизации, правил биннинга, частот и вероятностей положительного класса зависимой переменной в категориях предиктора использовались все наблюдения набора, часть из которых у нас теперь находится в контрольном блоке (по сути выборке новых данных). В таких случаях в Python используем конвейер (класс `Pipeline`). Конвейер содержит список этапов, обычно этап-трансформер и этап-модель машинного обучения. Каждый этап представляет собой кортеж, содержащий имя (любая строка на ваш выбор) и экземпляр класса. Трансформер получаем с помощью класса `ColumnTransformer`, передав список трехэлементных кортежей, в котором первый элемент кортежа – название конвейера с преобразованиями для определенного типа признаков, второй элемент кортежа – собственно конвейер с преобразованиями для определенного типа признаков, третий элемент кортежа – список признаков определенного типа. Подробнее о работе с классами `Pipeline` и `ColumnTransformer` читайте в приложении 8.

Рис. 3 План предварительной подготовки данных

Удаление бесполезных переменных, переменных «из будущего», нестабильных переменных

Переменные, у которых количество категорий совпадает с количеством наблюдений, или переменные с одним уникальным значением (переменные-константы) бесполезны для анализа и поэтому их удаляют. Также необходимо удалить переменные «из будущего». Простой пример – мы предсказываем стоимость квадратного метра жилья. В нашем распоряжении есть исторические данные о реализованных сделках, среди предикторов – время экспозиции квартиры. Однако когда нам нужно будет применить нашу модель для оценки стоимости новой квартиры, выставляемой на продажу, у нас по этой квартире не будет данных об ее экспозиции.

Риск также несут переменные, в отношении которых мы знаем, что они не фиксировались в течение всего периода сбора исторических данных или мы предполагаем, что в будущем мы не сможем получить информацию по этим переменным. Например, МФО фиксировала предиктор *Служба в армии* при выдаче микрокредита, а затем после корректировки правил кредитной политики отказалась от этого предиктора. У нас есть данные, где в течение первых 8 месяцев исторических данных предиктор *Служба в армии* фиксировался, а в последующие 4 месяца исторических данных он перестал фиксироваться (значения записаны как пропуски) и в новых данных его не будет.

Для начала удалим идентификационную переменную *client_id*. Эта переменная имеет столько же уникальных значений, сколько у нас наблюдений, поэтому бесполезна для моделирования.

Для удаления переменной можно применить метод `.drop()`. При этом нужно воспользоваться параметром `axis` и задать значение `1`, поскольку при удалении переменных мы перемещаемся по оси 1, то есть по оси столбцов. Кроме того, чтобы операция была осуществлена на месте, нужно воспользоваться параметром `inplace` и задать для него значение `True`.

In[5]:

```
# удаляем переменную client_id
data.drop('client_id', axis=1, inplace=True)

# выводим первые 5 наблюдений датафрейма
data.head()
```

Out[5]:

	gender	age	marital_status	job_position	credit_sum	credit_month	tariff_id	score_shk	education	living_region	monthly_income	credit_count
0	M	NaN	NaN	UMN	59998,00	10	1.6	NaN	GRD	КРАСНОДАРСКИЙ КРАЙ	30000.0	1.0
1	F	NaN	MAR	UMN	10889,00	6	1.1	NaN	NaN	МОСКВА	NaN	2.0
2	M	32.0	MAR	SPC	10728,00	12	1.1	NaN	NaN	ОБЛ САРАТОВСКАЯ	NaN	5.0
3	F	27.0	NaN	SPC	12009,09	12	1.1	NaN	NaN	ОБЛ ВОЛГОГРАДСКАЯ	NaN	2.0
4	M	45.0	NaN	SPC	NaN	10	1.1	0,421385	SCH	ЧЕЛЯБИНСКАЯ ОБЛАСТЬ	NaN	1.0

Преобразование типов переменных и нормализация строковых значений

Преобразование типов переменных – один из первых этапов предварительной подготовки данных. Например, количественная переменная может быть ошибочно записана как категориальная или наоборот. Часто этот этап сочетается с этапом нормализации строковых значений.

Обратите внимание на странное строковое значение `ОБЛ САРАТОВСКАЯ` переменной *living_region*. Давайте с помощью метода `.unique()` выведем уникальные значения этой переменной.

In[6]:
```
# смотрим уникальные значения
# по переменной living_region
data['living_region'].unique()
```

Out[6]:
```
array(['КРАСНОДАРСКИЙ КРАЙ', 'МОСКВА', 'ОБЛ САРАТОВСКАЯ',
       'ОБЛ ВОЛГОГРАДСКАЯ', 'ЧЕЛЯБИНСКАЯ ОБЛАСТЬ', 'СТАВРОПОЛЬСКИЙ КРАЙ',
       'ОБЛ НИЖЕГОРОДСКАЯ', 'МОСКОВСКАЯ ОБЛ',
       'ХАНТЫ-МАНСИЙСКИЙ АВТОНОМНЫЙ ОКРУГ - ЮГРА', 'КРАЙ СТАВРОПОЛЬСКИЙ',
       'САНКТ-ПЕТЕРБУРГ', 'РЕСП. БАШКОРТОСТАН', 'ОБЛ АРХАНГЕЛЬСКАЯ',
       'ХАНТЫ-МАНСИЙСКИЙ АО', 'РЕСП БАШКОРТОСТАН', 'ПЕРМСКИЙ КРАЙ',
       'РЕСП КАРАЧАЕВО-ЧЕРКЕССКАЯ', 'САРАТОВСКАЯ ОБЛ', 'ОБЛ КАЛУЖСКАЯ',
       'ОБЛ ВОЛОГОДСКАЯ', 'РОСТОВСКАЯ ОБЛ', 'УДМУРТСКАЯ РЕСП',
       'ОБЛ ИРКУТСКАЯ', 'ПРИВОЛЖСКИЙ ФЕДЕРАЛЬНЫЙ ОКРУГ', 'ОБЛ МОСКОВСКАЯ',
       'ОБЛ ТЮМЕНСКАЯ', 'ОБЛ БЕЛГОРОДСКАЯ', 'РОСТОВСКАЯ ОБЛАСТЬ',
       'ОБЛ КОСТРОМСКАЯ', 'РЕСП ХАКАСИЯ', 'РЕСПУБЛИКА ТАТАРСТАН',
       'ИРКУТСКАЯ ОБЛАСТЬ', 'ОБЛ СВЕРДЛОВСКАЯ', 'ОБЛ ПСКОВСКАЯ',
       'КРАЙ ЗАБАЙКАЛЬСКИЙ', 'СВЕРДЛОВСКАЯ ОБЛ', 'ОБЛ ОРЕНБУРГСКАЯ',
       'ОБЛ ВОРОНЕЖСКАЯ', 'ОБЛ АСТРАХАНСКАЯ', 'ОБЛ НОВОСИБИРСКАЯ',
       'ОБЛ ЧЕЛЯБИНСКАЯ', 'ОРЕНБУРГСКАЯ ОБЛ', 'СВЕРДЛОВСКАЯ ОБЛАСТЬ',
       'ОБЛ КУРГАНСКАЯ', 'ЧЕЛЯБИНСКАЯ ОБЛ', 'НИЖЕГОРОДСКАЯ ОБЛАСТЬ',
       'ТАТАРСТАН РЕСП', 'УЛЬЯНОВСКАЯ ОБЛ', 'МОСКВА Г', 'ОБЛ МУРМАНСКАЯ',
       'КРАСНОЯРСКИЙ КРАЙ', 'РЕСП БУРЯТИЯ', 'РЕСП. САХА (ЯКУТИЯ)',
       'ОБЛ АМУРСКАЯ', 'ХАБАРОВСКИЙ КРАЙ', 'САНКТ-ПЕТЕРБУРГ Г',
       'ЯМАЛО-НЕНЕЦКИЙ АО', 'ОБЛ САМАРСКАЯ', 'ТЮМЕНСКАЯ ОБЛАСТЬ',
       'ТВЕРСКАЯ ОБЛАСТЬ', 'ЯРОСЛАВСКАЯ ОБЛАСТЬ', 'ОБЛ ВЛАДИМИРСКАЯ',
       'ОБЛ ЛЕНИНГРАДСКАЯ', 'ОРЛОВСКАЯ ОБЛ', 'ОБЛ КЕМЕРОВСКАЯ',
       'ОМСКАЯ ОБЛ', 'РЕСП ЧЕЧЕНСКАЯ', 'ОБЛ КУРСКАЯ', 'ТУЛЬСКАЯ ОБЛ',
       'РЕСП АДЫГЕЯ', 'ТУЛЬСКАЯ ОБЛАСТЬ', 'РЕСПУБЛИКА КОМИ',
       'ПРИМОРСКИЙ КРАЙ', 'САМАРСКАЯ ОБЛ', 'СМОЛЕНСКАЯ ОБЛАСТЬ',
       'ОБЛ КИРОВСКАЯ', 'САМАРСКАЯ ОБЛАСТЬ', 'РЕСП ДАГЕСТАН',
       'ПЕНЗЕНСКАЯ ОБЛ', 'ТВЕРСКАЯ ОБЛ', 'УДМУРТСКАЯ РЕСПУБЛИКА',
       'РЕСП КАРЕЛИЯ', 'ОБЛ ТОМСКАЯ', 'РЕСПУБЛИКА БУРЯТИЯ',
       'ОБЛ МАГАДАНСКАЯ', 'РЕСП КОМИ', 'ЯРОСЛАВСКАЯ ОБЛ',
       'ОРЛОВСКАЯ ОБЛАСТЬ', 'ТОМСКАЯ ОБЛАСТЬ', 'РЕСП МАРИЙ ЭЛ',
       'ОБЛ ИВАНОВСКАЯ', 'КРАЙ КРАСНОДАРСКИЙ', 'РЕСПУБЛИКА АДЫГЕЯ',
       'САРАТОВСКАЯ ОБЛАСТЬ', 'ЕВРЕЙСКАЯ АОБЛ', 'ХАКАСИЯ РЕСП',
       'ПСКОВСКАЯ ОБЛ', 'КРАЙ АЛТАЙСКИЙ', 'РЕСП КАБАРДИНО-БАЛКАРСКАЯ',
       'ТЮМЕНСКАЯ ОБЛ', 'КРАЙ ПЕРМСКИЙ',
       'АО ХАНТЫ-МАНСИЙСКИЙ АВТОНОМНЫЙ ОКРУГ - Ю', 'БАШКОРТОСТАН',
       'ОБЛ ТАМБОВСКАЯ', 'ТЫВА РЕСП', 'ОБЛ НОВГОРОДСКАЯ', 'ОБЛ ЛИПЕЦКАЯ',
       'ОБЛ ТУЛЬСКАЯ', 'ПСКОВСКАЯ ОБЛАСТЬ', 'ВОЛОГОДСКАЯ ОБЛАСТЬ',
       'САХА /ЯКУТИЯ/ РЕСП', 'ОМСКАЯ ОБЛАСТЬ', 'АРХАНГЕЛЬСКАЯ ОБЛАСТЬ',
       'ТАМБОВСКАЯ ОБЛ', 'РЕСП СЕВЕРНАЯ ОСЕТИЯ - АЛАНИЯ',
       'ЛЕНИНГРАДСКАЯ ОБЛ', 'ОБЛ ТВЕРСКАЯ', 'ПЕНЗЕНСКАЯ ОБЛАСТЬ',
       'УЛЬЯНОВСКАЯ ОБЛАСТЬ', 'СМОЛЕНСКАЯ ОБЛ', 'Г. САНКТ-ПЕТЕРБУРГ',
       'ТОМСКАЯ ОБЛ', 'РЯЗАНСКАЯ ОБЛ', 'САХАЛИНСКАЯ ОБЛАСТЬ',
       'ЧУВАШСКАЯ РЕСПУБЛИКА - ЧУВАШИЯ', nan, 'БАШКОРТОСТАН РЕСП',
       'КРАЙ КАМЧАТСКИЙ', 'РЕСП МОРДОВИЯ', 'РЕСПУБЛИКА КАЛМЫКИЯ',
       'РЕСП АЛТАЙ', 'ОБЛ БРЯНСКАЯ', 'РЕСП ТАТАРСТАН',
       'МОСКОВСКАЯ ОБЛАСТЬ', 'КАБАРДИНО-БАЛКАРСКАЯ РЕСП',
       'ЧУВАШСКАЯ РЕСПУБЛИКА', 'АО ЯМАЛО-НЕНЕЦКИЙ', 'ОБЛ ПЕНЗЕНСКАЯ',
       'КРАЙ ПРИМОРСКИЙ', 'САХАЛИНСКАЯ ОБЛ', 'КРАЙ КРАСНОЯРСКИЙ',
```

```
'ОРЕНБУРГСКАЯ ОБЛАСТЬ', 'РЯЗАНСКАЯ ОБЛАСТЬ', 'РЕСПУБЛИКА КАРЕЛИЯ',
'ОБЛ СМОЛЕНСКАЯ', 'ОБЛ КАЛИНИНГРАДСКАЯ', 'РЕСПУБЛИКА ТЫВА',
'МУРМАНСКАЯ ОБЛ', 'ТАМБОВСКАЯ ОБЛАСТЬ', 'ОБЛ РОСТОВСКАЯ',
'РЕСПУБЛИКА МАРИЙ ЭЛ', 'АСТРАХАНСКАЯ ОБЛАСТЬ', 'КУРСКАЯ ОБЛ',
'СЕВЕРНАЯ ОСЕТИЯ - АЛАНИЯ РЕСП', 'НИЖЕГОРОДСКАЯ ОБЛ',
'РЕСПУБЛИКА МОРДОВИЯ', 'НОВОСИБИРСКАЯ ОБЛ',
'ЧУВАШСКАЯ - ЧУВАШИЯ РЕСП', 'РЕСП САХА /ЯКУТИЯ/', 'РЕСП КАЛМЫКИЯ',
'НЕНЕЦКИЙ АО', 'ЛЕНИНГРАДСКАЯ ОБЛАСТЬ', 'ЗАБАЙКАЛЬСКИЙ КРАЙ',
'ВОЛОГОДСКАЯ', 'АРХАНГЕЛЬСКАЯ', 'РЕСП УДМУРТСКАЯ', 'Г МОСКВА',
'БУРЯТИЯ РЕСП', 'РЕСП ТЫВА', 'ПЕРМСКАЯ ОБЛ', 'ОБЛ РЯЗАНСКАЯ',
'КАЛМЫКИЯ РЕСП', 'Г. МОСКВА', 'КЕМЕРОВСКАЯ ОБЛ',
'КАБАРДИНО-БАЛКАРСКАЯ', 'КРАЙ ХАБАРОВСКИЙ', 'ОБЛ ЯРОСЛАВСКАЯ',
'МУРМАНСКАЯ ОБЛАСТЬ', 'ВОЛОГОДСКАЯ ОБЛ', 'ОБЛ УЛЬЯНОВСКАЯ',
'КЕМЕРОВСКАЯ ОБЛАСТЬ', 'ИРКУТСКАЯ ОБЛ', 'ВЛАДИМИРСКАЯ ОБЛ',
'АСТРАХАНСКАЯ ОБЛ', 'КУРСКАЯ ОБЛАСТЬ', 'КУРГАНСКАЯ ОБЛ',
'КУРГАНСКАЯ ОБЛАСТЬ', 'ВОРОНЕЖСКАЯ ОБЛ', 'ОБЛ САХАЛИНСКАЯ',
'БЕЛГОРОДСКАЯ ОБЛ', 'РОСТОВСКАЯ', 'ВОЛГОГРАДСКАЯ ОБЛАСТЬ',
'АСТРАХАНСКАЯ', 'ЧУКОТСКИЙ АО', 'ГОРЬКОВСКАЯ ОБЛ',
'НОВГОРОДСКАЯ ОБЛ', 'КАЛИНИНГРАДСКАЯ ОБЛ', 'ОМСКАЯ', 'КОМИ РЕСП',
'СЕВ. ОСЕТИЯ - АЛАНИЯ', 'САМАРСКАЯ', 'ВОЛГОГРАДСКАЯ ОБЛ',
'ОБЛ ОРЛОВСКАЯ', 'БЕЛГОРОДСКАЯ ОБЛАСТЬ', 'БУРЯТИЯ',
'ВЛАДИМИРСКАЯ ОБЛАСТЬ', 'ЧУВАШИЯ ЧУВАШСКАЯ РЕСПУБЛИКА -',
'РЕСП ИНГУШЕТИЯ', 'КРАСНОДАРСКИЙ', 'АМУРСКАЯ ОБЛАСТЬ',
'ЛЕНИНГРАДСКАЯ', 'БРЯНСКАЯ ОБЛ', 'ЧЕЧЕНСКАЯ РЕСП',
'ЧЕЧЕНСКАЯ РЕСПУБЛИКА', 'РЕСПУБЛИКАТАТАРСТАН', 'АДЫГЕЯ РЕСП',
'ОБЛ ОМСКАЯ', 'ТОМСКАЯ', 'РОССИЯ', 'МАРИЙ ЭЛ РЕСП',
'КИРОВСКАЯ ОБЛ', 'ВОРОНЕЖСКАЯ ОБЛАСТЬ', 'МОРДОВИЯ РЕСП',
'ЛИПЕЦКАЯ ОБЛ', 'ХАКАСИЯ', 'КОСТРОМСКАЯ ОБЛ', 'КОМИ',
'КАЛУЖСКАЯ ОБЛ', 'ЧИТИНСКАЯ ОБЛ', 'БРЯНСКАЯ ОБЛАСТЬ',
'АМУРСКАЯ ОБЛ', 'ЧУВАШСКАЯ РЕСП', 'МОСКОВСКАЯ',
'КАРАЧАЕВО-ЧЕРКЕССКАЯ РЕСП', '98', 'МЫТИЩИНСКИЙ Р-Н',
'АРХАНГЕЛЬСКАЯ ОБЛ', 'СТАВРОПОЛЬСКИЙ', 'ИВАНОВСКАЯ ОБЛАСТЬ',
'КАЛУЖСКАЯ ОБЛАСТЬ', 'КАМЧАТСКИЙ КРАЙ', 'ОБЛ. ВЛАДИМИРСКАЯ',
'РЕСПУБЛИКА ДАГЕСТАН', 'ИВАНОВСКАЯ ОБЛ', 'КРАЙ.ПЕРМСКИЙ',
'КАЛУЖСКАЯ', 'КРАЙ. СТАВРОПОЛЬСКИЙ', 'РЕСПУБЛИКА ХАКАСИЯ',
'РЕСПУБЛИКА САХА', 'КОСТРОМСКАЯ ОБЛАСТЬ', 'АО НЕНЕЦКИЙ',
'МОСКВОСКАЯ ОБЛ', 'ДАГЕСТАН РЕСП', 'КРАЙ. КРАСНОЯРСКИЙ',
'ОБЛ.РОСТОВСКАЯ', 'ЛИПЕЦКАЯ ОБЛАСТЬ', 'ЕВРЕЙСКАЯ АВТОНОМНАЯ',
'ЭВЕНКИЙСКИЙ АО', 'КАМЧАТСКАЯ ОБЛАСТЬ', 'РЕСПУБЛИКА АЛТАЙ',
'АЛТАЙСКИЙ КРАЙ', 'НОВОСИБИРСКАЯ ОБЛАСТЬ', 'НОВГОРОДСКАЯ ОБЛАСТЬ',
'КАРЕЛИЯ РЕСП', 'ГУСЬ-ХРУСТАЛЬНЫЙ Р-Н', 'КАЛМЫКИЯ',
'ИНГУШЕТИЯ РЕСП', 'АЛТАЙСКИЙ', 'КИРОВСКАЯ ОБЛАСТЬ',
'ОБЛ. МУРМАНСКАЯ', 'КАРЕЛИЯ', 'РЕСП. КОМИ', 'ОБЛ. ЛИПЕЦКАЯ',
'БРЯНСКИЙ', 'ОБЛ. СВЕРДЛОВСКАЯ', 'ПЕРМСКИЙ', 'ОРЁЛ',
'ОБЛ.НИЖЕГОРОДСКАЯ', 'АОБЛ ЕВРЕЙСКАЯ', 'СВЕРДЛОВСКАЯ',
'ОБЛ.МОСКОВСКАЯ', 'ОБЛ.САРАТОВСКАЯ', 'КЕМЕРОВСКАЯ',
'ДАЛЬНИЙ ВОСТОК', 'ОБЛ. НОВОСИБИРСКАЯ', 'ОБЛ. КУРГАНСКАЯ', '74',
'ОБЛ. БЕЛГОРОДСКАЯ', 'МАГАДАНСКАЯ ОБЛАСТЬ', 'ВОЛОГОДСКАЯ ОБЛ.',
'ТЮМЕНСКАЯ', 'КАРАЧАЕВО-ЧЕРКЕССКАЯ', 'ОБЛ. ЧЕЛЯБИНСКАЯ',
'САХА /ЯКУТИЯ/', 'Г.МОСКВА', 'Г.ОДИНЦОВО МОСКОВСКАЯ ОБЛ',
'НОВОСИБИРСКАЯ', 'РЕСП.БАШКОРТОСТАН', 'КРАЙ. ПЕРМСКИЙ',
'РЕСП ЧУВАШСКАЯ - ЧУВАШИЯ', 'ОБЛ. КИРОВСКАЯ',
'КАЛИНИНГРАДСКАЯ ОБЛ.'], dtype=object)
```

Теперь с помощью метода `.nunique()` выведем количество уникальных значений переменной *living_region*.

In[7]:
```
# смотрим количество уникальных значений
# переменной living_region
data['living_region'].nunique()
```

Out[7]:
```
301
```

Мы видим, что один и тот же регион может быть по-разному записан (например, Архангельская область записана как АРХАНГЕЛЬСКАЯ, АРХАНГЕЛЬСКАЯ ОБЛ, АРХАНГЕЛЬСКАЯ ОБЛАСТЬ, ОБЛ АРХАНГЕЛЬСКАЯ),

некоторые регионы представлены числами типа 74 и 98 (скорее всего это автомобильные коды регионов: 74 соответствует Челябинской области, а 98 – Санкт-Петербургу), также присутствуют и вовсе странные категории (ГОРЬКОВСКАЯ ОБЛ, ГУСЬ-ХРУСТАЛЬНЫЙ Р-Н, ДАЛЬНИЙ ВОСТОК, ПРИВОЛЖСКИЙ ФЕДЕРАЛЬНЫЙ ОКРУГ, РОССИЯ). Давайте приведем названия регионов к единому стандарту.
Увеличим количество отображаемых строк в выводе.

In[8]:
```
# увеличиваем максимальное количество
# отображаемых строк
pd.options.display.max_rows = 310
```

Теперь уникальные значения переменной *living_region* записываем в отдельный объект *regions*.

In[9]:
```
# уникальные значения переменной living_region
# записываем в отдельный объект regions
regions = data['living_region'].unique()
```

Теперь мы создаем серию, у которой в качестве индексных меток и значений будут выступать уникальные значения переменной *living_region*, записанные в `regions`.

In[10]:
```
# создаем серию, у которой в качестве значений и индексных
# меток будут выступать уникальные значения переменной
# living_region, записанные в regions
regions = pd.Series(data=regions, index=regions, name='regions')
regions
```

Out[10]:
```
КРАСНОДАРСКИЙ КРАЙ                                КРАСНОДАРСКИЙ КРАЙ
МОСКВА                                                    МОСКВА
ОБЛ САРАТОВСКАЯ                                  ОБЛ САРАТОВСКАЯ
ОБЛ ВОЛГОГРАДСКАЯ                               ОБЛ ВОЛГОГРАДСКАЯ
ЧЕЛЯБИНСКАЯ ОБЛАСТЬ                             ЧЕЛЯБИНСКАЯ ОБЛАСТЬ
СТАВРОПОЛЬСКИЙ КРАЙ                             СТАВРОПОЛЬСКИЙ КРАЙ
ОБЛ НИЖЕГОРОДСКАЯ                               ОБЛ НИЖЕГОРОДСКАЯ
МОСКОВСКАЯ ОБЛ                                   МОСКОВСКАЯ ОБЛ
ХАНТЫ-МАНСИЙСКИЙ АВТОНОМНЫЙ ОКРУГ - ЮГРА    ХАНТЫ-МАНСИЙСКИЙ АВТОНОМНЫЙ ОКРУГ - ЮГРА
КРАЙ СТАВРОПОЛЬСКИЙ                             КРАЙ СТАВРОПОЛЬСКИЙ
САНКТ-ПЕТЕРБУРГ                                     САНКТ-ПЕТЕРБУРГ
РЕСП. БАШКОРТОСТАН                               РЕСП. БАШКОРТОСТАН
ОБЛ АРХАНГЕЛЬСКАЯ                               ОБЛ АРХАНГЕЛЬСКАЯ
ХАНТЫ-МАНСИЙСКИЙ АО                             ХАНТЫ-МАНСИЙСКИЙ АО
РЕСП БАШКОРТОСТАН                               РЕСП БАШКОРТОСТАН
ПЕРМСКИЙ КРАЙ                                       ПЕРМСКИЙ КРАЙ
РЕСП КАРАЧАЕВО-ЧЕРКЕССКАЯ                   РЕСП КАРАЧАЕВО-ЧЕРКЕССКАЯ
САРАТОВСКАЯ ОБЛ                                 САРАТОВСКАЯ ОБЛ
ОБЛ КАЛУЖСКАЯ                                     ОБЛ КАЛУЖСКАЯ
ОБЛ ВОЛОГОДСКАЯ                                 ОБЛ ВОЛОГОДСКАЯ
РОСТОВСКАЯ ОБЛ                                   РОСТОВСКАЯ ОБЛ
УДМУРТСКАЯ РЕСП                                 УДМУРТСКАЯ РЕСП
ОБЛ ИРКУТСКАЯ                                     ОБЛ ИРКУТСКАЯ
ПРИВОЛЖСКИЙ ФЕДЕРАЛЬНЫЙ ОКРУГ           ПРИВОЛЖСКИЙ ФЕДЕРАЛЬНЫЙ ОКРУГ
ОБЛ МОСКОВСКАЯ                                   ОБЛ МОСКОВСКАЯ
ОБЛ ТЮМЕНСКАЯ                                     ОБЛ ТЮМЕНСКАЯ
ОБЛ БЕЛГОРОДСКАЯ                               ОБЛ БЕЛГОРОДСКАЯ
РОСТОВСКАЯ ОБЛАСТЬ                           РОСТОВСКАЯ ОБЛАСТЬ
ОБЛ КОСТРОМСКАЯ                                 ОБЛ КОСТРОМСКАЯ
РЕСП ХАКАСИЯ                                       РЕСП ХАКАСИЯ
РЕСПУБЛИКА ТАТАРСТАН                       РЕСПУБЛИКА ТАТАРСТАН
ИРКУТСКАЯ ОБЛАСТЬ                             ИРКУТСКАЯ ОБЛАСТЬ
```

ОБЛ СВЕРДЛОВСКАЯ	ОБЛ СВЕРДЛОВСКАЯ
ОБЛ ПСКОВСКАЯ	ОБЛ ПСКОВСКАЯ
КРАЙ ЗАБАЙКАЛЬСКИЙ	КРАЙ ЗАБАЙКАЛЬСКИЙ
СВЕРДЛОВСКАЯ ОБЛ	СВЕРДЛОВСКАЯ ОБЛ
ОБЛ ОРЕНБУРГСКАЯ	ОБЛ ОРЕНБУРГСКАЯ
ОБЛ ВОРОНЕЖСКАЯ	ОБЛ ВОРОНЕЖСКАЯ
ОБЛ АСТРАХАНСКАЯ	ОБЛ АСТРАХАНСКАЯ
ОБЛ НОВОСИБИРСКАЯ	ОБЛ НОВОСИБИРСКАЯ
ОБЛ ЧЕЛЯБИНСКАЯ	ОБЛ ЧЕЛЯБИНСКАЯ
ОРЕНБУРГСКАЯ ОБЛ	ОРЕНБУРГСКАЯ ОБЛ
СВЕРДЛОВСКАЯ ОБЛАСТЬ	СВЕРДЛОВСКАЯ ОБЛАСТЬ
ОБЛ КУРГАНСКАЯ	ОБЛ КУРГАНСКАЯ
ЧЕЛЯБИНСКАЯ ОБЛ	ЧЕЛЯБИНСКАЯ ОБЛ
НИЖЕГОРОДСКАЯ ОБЛАСТЬ	НИЖЕГОРОДСКАЯ ОБЛАСТЬ
ТАТАРСТАН РЕСП	ТАТАРСТАН РЕСП
УЛЬЯНОВСКАЯ ОБЛ	УЛЬЯНОВСКАЯ ОБЛ
МОСКВА Г	МОСКВА Г
ОБЛ МУРМАНСКАЯ	ОБЛ МУРМАНСКАЯ
КРАСНОЯРСКИЙ КРАЙ	КРАСНОЯРСКИЙ КРАЙ
РЕСП БУРЯТИЯ	РЕСП БУРЯТИЯ
РЕСП. САХА (ЯКУТИЯ)	РЕСП. САХА (ЯКУТИЯ)
ОБЛ АМУРСКАЯ	ОБЛ АМУРСКАЯ
ХАБАРОВСКИЙ КРАЙ	ХАБАРОВСКИЙ КРАЙ
САНКТ-ПЕТЕРБУРГ Г	САНКТ-ПЕТЕРБУРГ Г
ЯМАЛО-НЕНЕЦКИЙ АО	ЯМАЛО-НЕНЕЦКИЙ АО
ОБЛ САМАРСКАЯ	ОБЛ САМАРСКАЯ
ТЮМЕНСКАЯ ОБЛАСТЬ	ТЮМЕНСКАЯ ОБЛАСТЬ
ТВЕРСКАЯ ОБЛАСТЬ	ТВЕРСКАЯ ОБЛАСТЬ
ЯРОСЛАВСКАЯ ОБЛАСТЬ	ЯРОСЛАВСКАЯ ОБЛАСТЬ
ОБЛ ВЛАДИМИРСКАЯ	ОБЛ ВЛАДИМИРСКАЯ
ОБЛ ЛЕНИНГРАДСКАЯ	ОБЛ ЛЕНИНГРАДСКАЯ
ОРЛОВСКАЯ ОБЛ	ОРЛОВСКАЯ ОБЛ
ОБЛ КЕМЕРОВСКАЯ	ОБЛ КЕМЕРОВСКАЯ
ОМСКАЯ ОБЛ	ОМСКАЯ ОБЛ
РЕСП ЧЕЧЕНСКАЯ	РЕСП ЧЕЧЕНСКАЯ
ОБЛ КУРСКАЯ	ОБЛ КУРСКАЯ
ТУЛЬСКАЯ ОБЛ	ТУЛЬСКАЯ ОБЛ
РЕСП АДЫГЕЯ	РЕСП АДЫГЕЯ
ТУЛЬСКАЯ ОБЛАСТЬ	ТУЛЬСКАЯ ОБЛАСТЬ
РЕСПУБЛИКА КОМИ	РЕСПУБЛИКА КОМИ
ПРИМОРСКИЙ КРАЙ	ПРИМОРСКИЙ КРАЙ
САМАРСКАЯ ОБЛ	САМАРСКАЯ ОБЛ
СМОЛЕНСКАЯ ОБЛАСТЬ	СМОЛЕНСКАЯ ОБЛАСТЬ
ОБЛ КИРОВСКАЯ	ОБЛ КИРОВСКАЯ
САМАРСКАЯ ОБЛАСТЬ	САМАРСКАЯ ОБЛАСТЬ
РЕСП ДАГЕСТАН	РЕСП ДАГЕСТАН
ПЕНЗЕНСКАЯ ОБЛ	ПЕНЗЕНСКАЯ ОБЛ
ТВЕРСКАЯ ОБЛ	ТВЕРСКАЯ ОБЛ
УДМУРТСКАЯ РЕСПУБЛИКА	УДМУРТСКАЯ РЕСПУБЛИКА
РЕСП КАРЕЛИЯ	РЕСП КАРЕЛИЯ
ОБЛ ТОМСКАЯ	ОБЛ ТОМСКАЯ
РЕСПУБЛИКА БУРЯТИЯ	РЕСПУБЛИКА БУРЯТИЯ
ОБЛ МАГАДАНСКАЯ	ОБЛ МАГАДАНСКАЯ
РЕСП КОМИ	РЕСП КОМИ
ЯРОСЛАВСКАЯ ОБЛ	ЯРОСЛАВСКАЯ ОБЛ
ОРЛОВСКАЯ ОБЛАСТЬ	ОРЛОВСКАЯ ОБЛАСТЬ
ТОМСКАЯ ОБЛАСТЬ	ТОМСКАЯ ОБЛАСТЬ
РЕСП МАРИЙ ЭЛ	РЕСП МАРИЙ ЭЛ
ОБЛ ИВАНОВСКАЯ	ОБЛ ИВАНОВСКАЯ
КРАЙ КРАСНОДАРСКИЙ	КРАЙ КРАСНОДАРСКИЙ
РЕСПУБЛИКА АДЫГЕЯ	РЕСПУБЛИКА АДЫГЕЯ
САРАТОВСКАЯ ОБЛАСТЬ	САРАТОВСКАЯ ОБЛАСТЬ
ЕВРЕЙСКАЯ АОБЛ	ЕВРЕЙСКАЯ АОБЛ
ХАКАСИЯ РЕСП	ХАКАСИЯ РЕСП
ПСКОВСКАЯ ОБЛ	ПСКОВСКАЯ ОБЛ
КРАЙ АЛТАЙСКИЙ	КРАЙ АЛТАЙСКИЙ
РЕСП КАБАРДИНО-БАЛКАРСКАЯ	РЕСП КАБАРДИНО-БАЛКАРСКАЯ
ТЮМЕНСКАЯ ОБЛ	ТЮМЕНСКАЯ ОБЛ
КРАЙ ПЕРМСКИЙ	КРАЙ ПЕРМСКИЙ
АО ХАНТЫ-МАНСИЙСКИЙ АВТОНОМНЫЙ ОКРУГ - Ю	АО ХАНТЫ-МАНСИЙСКИЙ АВТОНОМНЫЙ ОКРУГ - Ю
БАШКОРТОСТАН	БАШКОРТОСТАН
ОБЛ ТАМБОВСКАЯ	ОБЛ ТАМБОВСКАЯ

ТЫВА РЕСП	ТЫВА РЕСП
ОБЛ НОВГОРОДСКАЯ	ОБЛ НОВГОРОДСКАЯ
ОБЛ ЛИПЕЦКАЯ	ОБЛ ЛИПЕЦКАЯ
ОБЛ ТУЛЬСКАЯ	ОБЛ ТУЛЬСКАЯ
ПСКОВСКАЯ ОБЛАСТЬ	ПСКОВСКАЯ ОБЛАСТЬ
ВОЛОГОДСКАЯ ОБЛАСТЬ	ВОЛОГОДСКАЯ ОБЛАСТЬ
САХА /ЯКУТИЯ/ РЕСП	САХА /ЯКУТИЯ/ РЕСП
ОМСКАЯ ОБЛАСТЬ	ОМСКАЯ ОБЛАСТЬ
АРХАНГЕЛЬСКАЯ ОБЛАСТЬ	АРХАНГЕЛЬСКАЯ ОБЛАСТЬ
ТАМБОВСКАЯ ОБЛ	ТАМБОВСКАЯ ОБЛ
РЕСП СЕВЕРНАЯ ОСЕТИЯ - АЛАНИЯ	РЕСП СЕВЕРНАЯ ОСЕТИЯ - АЛАНИЯ
ЛЕНИНГРАДСКАЯ ОБЛ	ЛЕНИНГРАДСКАЯ ОБЛ
ОБЛ ТВЕРСКАЯ	ОБЛ ТВЕРСКАЯ
ПЕНЗЕНСКАЯ ОБЛАСТЬ	ПЕНЗЕНСКАЯ ОБЛАСТЬ
УЛЬЯНОВСКАЯ ОБЛАСТЬ	УЛЬЯНОВСКАЯ ОБЛАСТЬ
СМОЛЕНСКАЯ ОБЛ	СМОЛЕНСКАЯ ОБЛ
Г. САНКТ-ПЕТЕРБУРГ	Г. САНКТ-ПЕТЕРБУРГ
ТОМСКАЯ ОБЛ	ТОМСКАЯ ОБЛ
РЯЗАНСКАЯ ОБЛ	РЯЗАНСКАЯ ОБЛ
САХАЛИНСКАЯ ОБЛАСТЬ	САХАЛИНСКАЯ ОБЛАСТЬ
ЧУВАШСКАЯ РЕСПУБЛИКА - ЧУВАШИЯ	ЧУВАШСКАЯ РЕСПУБЛИКА - ЧУВАШИЯ
NaN	NaN
БАШКОРТОСТАН РЕСП	БАШКОРТОСТАН РЕСП
КРАЙ КАМЧАТСКИЙ	КРАЙ КАМЧАТСКИЙ
РЕСП МОРДОВИЯ	РЕСП МОРДОВИЯ
РЕСПУБЛИКА КАЛМЫКИЯ	РЕСПУБЛИКА КАЛМЫКИЯ
РЕСП АЛТАЙ	РЕСП АЛТАЙ
ОБЛ БРЯНСКАЯ	ОБЛ БРЯНСКАЯ
РЕСП ТАТАРСТАН	РЕСП ТАТАРСТАН
МОСКОВСКАЯ ОБЛАСТЬ	МОСКОВСКАЯ ОБЛАСТЬ
КАБАРДИНО-БАЛКАРСКАЯ РЕСП	КАБАРДИНО-БАЛКАРСКАЯ РЕСП
ЧУВАШСКАЯ РЕСПУБЛИКА	ЧУВАШСКАЯ РЕСПУБЛИКА
АО ЯМАЛО-НЕНЕЦКИЙ	АО ЯМАЛО-НЕНЕЦКИЙ
ОБЛ ПЕНЗЕНСКАЯ	ОБЛ ПЕНЗЕНСКАЯ
КРАЙ ПРИМОРСКИЙ	КРАЙ ПРИМОРСКИЙ
САХАЛИНСКАЯ ОБЛ	САХАЛИНСКАЯ ОБЛ
КРАЙ КРАСНОЯРСКИЙ	КРАЙ КРАСНОЯРСКИЙ
ОРЕНБУРГСКАЯ ОБЛАСТЬ	ОРЕНБУРГСКАЯ ОБЛАСТЬ
РЯЗАНСКАЯ ОБЛАСТЬ	РЯЗАНСКАЯ ОБЛАСТЬ
РЕСПУБЛИКА КАРЕЛИЯ	РЕСПУБЛИКА КАРЕЛИЯ
ОБЛ СМОЛЕНСКАЯ	ОБЛ СМОЛЕНСКАЯ
ОБЛ КАЛИНИНГРАДСКАЯ	ОБЛ КАЛИНИНГРАДСКАЯ
РЕСПУБЛИКА ТЫВА	РЕСПУБЛИКА ТЫВА
МУРМАНСКАЯ ОБЛ	МУРМАНСКАЯ ОБЛ
ТАМБОВСКАЯ ОБЛАСТЬ	ТАМБОВСКАЯ ОБЛАСТЬ
ОБЛ РОСТОВСКАЯ	ОБЛ РОСТОВСКАЯ
РЕСПУБЛИКА МАРИЙ ЭЛ	РЕСПУБЛИКА МАРИЙ ЭЛ
АСТРАХАНСКАЯ ОБЛАСТЬ	АСТРАХАНСКАЯ ОБЛАСТЬ
КУРСКАЯ ОБЛ	КУРСКАЯ ОБЛ
СЕВЕРНАЯ ОСЕТИЯ - АЛАНИЯ РЕСП	СЕВЕРНАЯ ОСЕТИЯ - АЛАНИЯ РЕСП
НИЖЕГОРОДСКАЯ ОБЛ	НИЖЕГОРОДСКАЯ ОБЛ
РЕСПУБЛИКА МОРДОВИЯ	РЕСПУБЛИКА МОРДОВИЯ
НОВОСИБИРСКАЯ ОБЛ	НОВОСИБИРСКАЯ ОБЛ
ЧУВАШСКАЯ - ЧУВАШИЯ РЕСП	ЧУВАШСКАЯ - ЧУВАШИЯ РЕСП
РЕСП САХА /ЯКУТИЯ/	РЕСП САХА /ЯКУТИЯ/
РЕСП КАЛМЫКИЯ	РЕСП КАЛМЫКИЯ
НЕНЕЦКИЙ АО	НЕНЕЦКИЙ АО
ЛЕНИНГРАДСКАЯ ОБЛАСТЬ	ЛЕНИНГРАДСКАЯ ОБЛАСТЬ
ЗАБАЙКАЛЬСКИЙ КРАЙ	ЗАБАЙКАЛЬСКИЙ КРАЙ
ВОЛОГОДСКАЯ	ВОЛОГОДСКАЯ
АРХАНГЕЛЬСКАЯ	АРХАНГЕЛЬСКАЯ
РЕСП УДМУРТСКАЯ	РЕСП УДМУРТСКАЯ
Г МОСКВА	Г МОСКВА
БУРЯТИЯ РЕСП	БУРЯТИЯ РЕСП
РЕСП ТЫВА	РЕСП ТЫВА
ПЕРМСКАЯ ОБЛ	ПЕРМСКАЯ ОБЛ
ОБЛ РЯЗАНСКАЯ	ОБЛ РЯЗАНСКАЯ
КАЛМЫКИЯ РЕСП	КАЛМЫКИЯ РЕСП
Г. МОСКВА	Г. МОСКВА
КЕМЕРОВСКАЯ ОБЛ	КЕМЕРОВСКАЯ ОБЛ
КАБАРДИНО-БАЛКАРСКАЯ	КАБАРДИНО-БАЛКАРСКАЯ
КРАЙ ХАБАРОВСКИЙ	КРАЙ ХАБАРОВСКИЙ

ОБЛ ЯРОСЛАВСКАЯ	ОБЛ ЯРОСЛАВСКАЯ
МУРМАНСКАЯ ОБЛАСТЬ	МУРМАНСКАЯ ОБЛАСТЬ
ВОЛОГОДСКАЯ ОБЛ	ВОЛОГОДСКАЯ ОБЛ
ОБЛ УЛЬЯНОВСКАЯ	ОБЛ УЛЬЯНОВСКАЯ
КЕМЕРОВСКАЯ ОБЛАСТЬ	КЕМЕРОВСКАЯ ОБЛАСТЬ
ИРКУТСКАЯ ОБЛ	ИРКУТСКАЯ ОБЛ
ВЛАДИМИРСКАЯ ОБЛ	ВЛАДИМИРСКАЯ ОБЛ
АСТРАХАНСКАЯ ОБЛ	АСТРАХАНСКАЯ ОБЛ
КУРСКАЯ ОБЛАСТЬ	КУРСКАЯ ОБЛАСТЬ
КУРГАНСКАЯ ОБЛ	КУРГАНСКАЯ ОБЛ
КУРГАНСКАЯ ОБЛАСТЬ	КУРГАНСКАЯ ОБЛАСТЬ
ВОРОНЕЖСКАЯ ОБЛ	ВОРОНЕЖСКАЯ ОБЛ
ОБЛ САХАЛИНСКАЯ	ОБЛ САХАЛИНСКАЯ
БЕЛГОРОДСКАЯ ОБЛ	БЕЛГОРОДСКАЯ ОБЛ
РОСТОВСКАЯ	РОСТОВСКАЯ
ВОЛГОГРАДСКАЯ ОБЛАСТЬ	ВОЛГОГРАДСКАЯ ОБЛАСТЬ
АСТРАХАНСКАЯ	АСТРАХАНСКАЯ
ЧУКОТСКИЙ АО	ЧУКОТСКИЙ АО
ГОРЬКОВСКАЯ ОБЛ	ГОРЬКОВСКАЯ ОБЛ
НОВГОРОДСКАЯ ОБЛ	НОВГОРОДСКАЯ ОБЛ
КАЛИНИНГРАДСКАЯ ОБЛ	КАЛИНИНГРАДСКАЯ ОБЛ
ОМСКАЯ	ОМСКАЯ
КОМИ РЕСП	КОМИ РЕСП
СЕВ. ОСЕТИЯ - АЛАНИЯ	СЕВ. ОСЕТИЯ - АЛАНИЯ
САМАРСКАЯ	САМАРСКАЯ
ВОЛГОГРАДСКАЯ ОБЛ	ВОЛГОГРАДСКАЯ ОБЛ
ОБЛ ОРЛОВСКАЯ	ОБЛ ОРЛОВСКАЯ
БЕЛГОРОДСКАЯ ОБЛАСТЬ	БЕЛГОРОДСКАЯ ОБЛАСТЬ
БУРЯТИЯ	БУРЯТИЯ
ВЛАДИМИРСКАЯ ОБЛАСТЬ	ВЛАДИМИРСКАЯ ОБЛАСТЬ
ЧУВАШИЯ ЧУВАШСКАЯ РЕСПУБЛИКА -	ЧУВАШИЯ ЧУВАШСКАЯ РЕСПУБЛИКА -
РЕСП ИНГУШЕТИЯ	РЕСП ИНГУШЕТИЯ
КРАСНОДАРСКИЙ	КРАСНОДАРСКИЙ
АМУРСКАЯ ОБЛАСТЬ	АМУРСКАЯ ОБЛАСТЬ
ЛЕНИНГРАДСКАЯ	ЛЕНИНГРАДСКАЯ
БРЯНСКАЯ ОБЛ	БРЯНСКАЯ ОБЛ
ЧЕЧЕНСКАЯ РЕСП	ЧЕЧЕНСКАЯ РЕСП
ЧЕЧЕНСКАЯ РЕСПУБЛИКА	ЧЕЧЕНСКАЯ РЕСПУБЛИКА
РЕСПУБЛИКАТАТАРСТАН	РЕСПУБЛИКАТАТАРСТАН
АДЫГЕЯ РЕСП	АДЫГЕЯ РЕСП
ОБЛ ОМСКАЯ	ОБЛ ОМСКАЯ
ТОМСКАЯ	ТОМСКАЯ
РОССИЯ	РОССИЯ
МАРИЙ ЭЛ РЕСП	МАРИЙ ЭЛ РЕСП
КИРОВСКАЯ ОБЛ	КИРОВСКАЯ ОБЛ
ВОРОНЕЖСКАЯ ОБЛАСТЬ	ВОРОНЕЖСКАЯ ОБЛАСТЬ
МОРДОВИЯ РЕСП	МОРДОВИЯ РЕСП
ЛИПЕЦКАЯ ОБЛ	ЛИПЕЦКАЯ ОБЛ
ХАКАСИЯ	ХАКАСИЯ
КОСТРОМСКАЯ ОБЛ	КОСТРОМСКАЯ ОБЛ
КОМИ	КОМИ
КАЛУЖСКАЯ ОБЛ	КАЛУЖСКАЯ ОБЛ
ЧИТИНСКАЯ ОБЛ	ЧИТИНСКАЯ ОБЛ
БРЯНСКАЯ ОБЛАСТЬ	БРЯНСКАЯ ОБЛАСТЬ
АМУРСКАЯ ОБЛ	АМУРСКАЯ ОБЛ
ЧУВАШСКАЯ РЕСП	ЧУВАШСКАЯ РЕСП
МОСКОВСКАЯ	МОСКОВСКАЯ
КАРАЧАЕВО-ЧЕРКЕССКАЯ РЕСП	КАРАЧАЕВО-ЧЕРКЕССКАЯ РЕСП
98	98
МЫТИЩИНСКИЙ Р-Н	МЫТИЩИНСКИЙ Р-Н
АРХАНГЕЛЬСКАЯ ОБЛ	АРХАНГЕЛЬСКАЯ ОБЛ
СТАВРОПОЛЬСКИЙ	СТАВРОПОЛЬСКИЙ
ИВАНОВСКАЯ ОБЛАСТЬ	ИВАНОВСКАЯ ОБЛАСТЬ
КАЛУЖСКАЯ ОБЛАСТЬ	КАЛУЖСКАЯ ОБЛАСТЬ
КАМЧАТСКИЙ КРАЙ	КАМЧАТСКИЙ КРАЙ
ОБЛ. ВЛАДИМИРСКАЯ	ОБЛ. ВЛАДИМИРСКАЯ
РЕСПУБЛИКА ДАГЕСТАН	РЕСПУБЛИКА ДАГЕСТАН
ИВАНОВСКАЯ ОБЛ	ИВАНОВСКАЯ ОБЛ
КРАЙ.ПЕРМСКИЙ	КРАЙ.ПЕРМСКИЙ
КАЛУЖСКАЯ	КАЛУЖСКАЯ
КРАЙ. СТАВРОПОЛЬСКИЙ	КРАЙ. СТАВРОПОЛЬСКИЙ
РЕСПУБЛИКА ХАКАСИЯ	РЕСПУБЛИКА ХАКАСИЯ

```
РЕСПУБЛИКА САХА                                 РЕСПУБЛИКА САХА
КОСТРОМСКАЯ ОБЛАСТЬ                          КОСТРОМСКАЯ ОБЛАСТЬ
АО НЕНЕЦКИЙ                                         АО НЕНЕЦКИЙ
МОСКВОСКАЯ ОБЛ                                   МОСКВОСКАЯ ОБЛ
ДАГЕСТАН РЕСП                                     ДАГЕСТАН РЕСП
КРАЙ. КРАСНОЯРСКИЙ                           КРАЙ. КРАСНОЯРСКИЙ
ОБЛ.РОСТОВСКАЯ                                   ОБЛ.РОСТОВСКАЯ
ЛИПЕЦКАЯ ОБЛАСТЬ                               ЛИПЕЦКАЯ ОБЛАСТЬ
ЕВРЕЙСКАЯ АВТОНОМНАЯ                       ЕВРЕЙСКАЯ АВТОНОМНАЯ
ЭВЕНКИЙСКИЙ АО                                   ЭВЕНКИЙСКИЙ АО
КАМЧАТСКАЯ ОБЛАСТЬ                         КАМЧАТСКАЯ ОБЛАСТЬ
РЕСПУБЛИКА АЛТАЙ                             РЕСПУБЛИКА АЛТАЙ
АЛТАЙСКИЙ КРАЙ                                 АЛТАЙСКИЙ КРАЙ
НОВОСИБИРСКАЯ ОБЛАСТЬ                   НОВОСИБИРСКАЯ ОБЛАСТЬ
НОВГОРОДСКАЯ ОБЛАСТЬ                     НОВГОРОДСКАЯ ОБЛАСТЬ
КАРЕЛИЯ РЕСП                                       КАРЕЛИЯ РЕСП
ГУСЬ-ХРУСТАЛЬНЫЙ Р-Н                     ГУСЬ-ХРУСТАЛЬНЫЙ Р-Н
КАЛМЫКИЯ                                               КАЛМЫКИЯ
ИНГУШЕТИЯ РЕСП                                 ИНГУШЕТИЯ РЕСП
АЛТАЙСКИЙ                                             АЛТАЙСКИЙ
КИРОВСКАЯ ОБЛАСТЬ                           КИРОВСКАЯ ОБЛАСТЬ
ОБЛ. МУРМАНСКАЯ                               ОБЛ. МУРМАНСКАЯ
КАРЕЛИЯ                                                 КАРЕЛИЯ
РЕСП. КОМИ                                           РЕСП. КОМИ
ОБЛ. ЛИПЕЦКАЯ                                   ОБЛ. ЛИПЕЦКАЯ
БРЯНСКИЙ                                               БРЯНСКИЙ
ОБЛ. СВЕРДЛОВСКАЯ                           ОБЛ. СВЕРДЛОВСКАЯ
ПЕРМСКИЙ                                               ПЕРМСКИЙ
ОРЁЛ                                                       ОРЁЛ
ОБЛ.НИЖЕГОРОДСКАЯ                         ОБЛ.НИЖЕГОРОДСКАЯ
АОБЛ ЕВРЕЙСКАЯ                                 АОБЛ ЕВРЕЙСКАЯ
СВЕРДЛОВСКАЯ                                     СВЕРДЛОВСКАЯ
ОБЛ.МОСКОВСКАЯ                                 ОБЛ.МОСКОВСКАЯ
ОБЛ.САРАТОВСКАЯ                               ОБЛ.САРАТОВСКАЯ
КЕМЕРОВСКАЯ                                         КЕМЕРОВСКАЯ
ДАЛЬНИЙ ВОСТОК                                 ДАЛЬНИЙ ВОСТОК
ОБЛ. НОВОСИБИРСКАЯ                         ОБЛ. НОВОСИБИРСКАЯ
ОБЛ. КУРГАНСКАЯ                               ОБЛ. КУРГАНСКАЯ
74                                                           74
ОБЛ. БЕЛГОРОДСКАЯ                           ОБЛ. БЕЛГОРОДСКАЯ
МАГАДАНСКАЯ ОБЛАСТЬ                     МАГАДАНСКАЯ ОБЛАСТЬ
ВОЛОГОДСКАЯ ОБЛ.                             ВОЛОГОДСКАЯ ОБЛ.
ТЮМЕНСКАЯ                                           ТЮМЕНСКАЯ
КАРАЧАЕВО-ЧЕРКЕССКАЯ                   КАРАЧАЕВО-ЧЕРКЕССКАЯ
ОБЛ. ЧЕЛЯБИНСКАЯ                             ОБЛ. ЧЕЛЯБИНСКАЯ
САХА /ЯКУТИЯ/                                   САХА /ЯКУТИЯ/
Г.МОСКВА                                                 Г.МОСКВА
Г.ОДИНЦОВО МОСКОВСКАЯ ОБЛ           Г.ОДИНЦОВО МОСКОВСКАЯ ОБЛ
НОВОСИБИРСКАЯ                                       НОВОСИБИРСКАЯ
РЕСП.БАШКОРТОСТАН                           РЕСП.БАШКОРТОСТАН
КРАЙ. ПЕРМСКИЙ                                   КРАЙ. ПЕРМСКИЙ
РЕСП ЧУВАШСКАЯ - ЧУВАШИЯ                 РЕСП ЧУВАШСКАЯ - ЧУВАШИЯ
ОБЛ. КИРОВСКАЯ                                   ОБЛ. КИРОВСКАЯ
КАЛИНИНГРАДСКАЯ ОБЛ.                       КАЛИНИНГРАДСКАЯ ОБЛ.
Name: regions, dtype: object
```

Теперь задаем список стоп-слов – слов, которые не несут никакой смысловой нагрузки и их можно проигнорировать. Их еще называют шумовыми словами.

In[11]:
```
# задаем стоп-слова
stopwrds = set(['ОБЛ', 'ОБЛАСТЬ', 'РЕСП', 'РЕСПУБЛИКА',
                'КРАЙ', 'Г', 'АО', 'АОБЛ', 'АВТОНОМНАЯ'])
```

Теперь мы напишем функцию для предобработки значений нашей серии `regions` и применим ее.

In[12]:

```
# пишем функцию для предобработки значений серии
def clean_region(x):
    x = re.sub('[.,]+', ' ', str(x))
    wrds = x.split(' ')
    wrds_new = []
    for w in wrds:
        if not w in stopwrds:
            wrds_new.append(w)
    x = ''.join(wrds_new)
    return x

# применяем функцию к нашей серии
regions = regions.map(clean_region)
```

Давайте взглянем на результаты применения функции.

In[13]:

```
# смотрим результаты применения функции
regions
```

Out[13]:

```
КРАСНОДАРСКИЙ КРАЙ                                КРАСНОДАРСКИЙ
МОСКВА                                                МОСКВА
ОБЛ САРАТОВСКАЯ                                   САРАТОВСКАЯ
ОБЛ ВОЛГОГРАДСКАЯ                                 ВОЛГОГРАДСКАЯ
ЧЕЛЯБИНСКАЯ ОБЛАСТЬ                               ЧЕЛЯБИНСКАЯ
СТАВРОПОЛЬСКИЙ КРАЙ                               СТАВРОПОЛЬСКИЙ
ОБЛ НИЖЕГОРОДСКАЯ                                 НИЖЕГОРОДСКАЯ
МОСКОВСКАЯ ОБЛ                                    МОСКОВСКАЯ
ХАНТЫ-МАНСИЙСКИЙ АВТОНОМНЫЙ ОКРУГ - ЮГРА    ХАНТЫ-МАНСИЙСКИЙАВТОНОМНЫЙОКРУГ-ЮГРА
КРАЙ СТАВРОПОЛЬСКИЙ                               СТАВРОПОЛЬСКИЙ
САНКТ-ПЕТЕРБУРГ                                   САНКТ-ПЕТЕРБУРГ
РЕСП. БАШКОРТОСТАН                                БАШКОРТОСТАН
ОБЛ АРХАНГЕЛЬСКАЯ                                 АРХАНГЕЛЬСКАЯ
ХАНТЫ-МАНСИЙСКИЙ АО                               ХАНТЫ-МАНСИЙСКИЙ
РЕСП БАШКОРТОСТАН                                 БАШКОРТОСТАН
ПЕРМСКИЙ КРАЙ                                         ПЕРМСКИЙ
РЕСП КАРАЧАЕВО-ЧЕРКЕССКАЯ                     КАРАЧАЕВО-ЧЕРКЕССКАЯ
САРАТОВСКАЯ ОБЛ                                   САРАТОВСКАЯ
ОБЛ КАЛУЖСКАЯ                                     КАЛУЖСКАЯ
ОБЛ ВОЛОГОДСКАЯ                                   ВОЛОГОДСКАЯ
РОСТОВСКАЯ ОБЛ                                    РОСТОВСКАЯ
УДМУРТСКАЯ РЕСП                                   УДМУРТСКАЯ
ОБЛ ИРКУТСКАЯ                                     ИРКУТСКАЯ
ПРИВОЛЖСКИЙ ФЕДЕРАЛЬНЫЙ ОКРУГ             ПРИВОЛЖСКИЙФЕДЕРАЛЬНЫЙОКРУГ
ОБЛ МОСКОВСКАЯ                                    МОСКОВСКАЯ
ОБЛ ТЮМЕНСКАЯ                                     ТЮМЕНСКАЯ
ОБЛ БЕЛГОРОДСКАЯ                                  БЕЛГОРОДСКАЯ
РОСТОВСКАЯ ОБЛАСТЬ                                РОСТОВСКАЯ
ОБЛ КОСТРОМСКАЯ                                   КОСТРОМСКАЯ
РЕСП ХАКАСИЯ                                          ХАКАСИЯ
РЕСПУБЛИКА ТАТАРСТАН                              ТАТАРСТАН
ИРКУТСКАЯ ОБЛАСТЬ                                 ИРКУТСКАЯ
ОБЛ СВЕРДЛОВСКАЯ                                  СВЕРДЛОВСКАЯ
ОБЛ ПСКОВСКАЯ                                     ПСКОВСКАЯ
КРАЙ ЗАБАЙКАЛЬСКИЙ                                ЗАБАЙКАЛЬСКИЙ
СВЕРДЛОВСКАЯ ОБЛ                                  СВЕРДЛОВСКАЯ
ОБЛ ОРЕНБУРГСКАЯ                                  ОРЕНБУРГСКАЯ
ОБЛ ВОРОНЕЖСКАЯ                                   ВОРОНЕЖСКАЯ
ОБЛ АСТРАХАНСКАЯ                                  АСТРАХАНСКАЯ
ОБЛ НОВОСИБИРСКАЯ                                 НОВОСИБИРСКАЯ
ОБЛ ЧЕЛЯБИНСКАЯ                                   ЧЕЛЯБИНСКАЯ
ОРЕНБУРГСКАЯ ОБЛ                                  ОРЕНБУРГСКАЯ
СВЕРДЛОВСКАЯ ОБЛАСТЬ                              СВЕРДЛОВСКАЯ
ОБЛ КУРГАНСКАЯ                                    КУРГАНСКАЯ
ЧЕЛЯБИНСКАЯ ОБЛ                                   ЧЕЛЯБИНСКАЯ
НИЖЕГОРОДСКАЯ ОБЛАСТЬ                             НИЖЕГОРОДСКАЯ
ТАТАРСТАН РЕСП                                        ТАТАРСТАН
УЛЬЯНОВСКАЯ ОБЛ                                   УЛЬЯНОВСКАЯ
МОСКВА Г                                              МОСКВА
ОБЛ МУРМАНСКАЯ                                    МУРМАНСКАЯ
```

КРАСНОЯРСКИЙ КРАЙ	КРАСНОЯРСКИЙ
РЕСП БУРЯТИЯ	БУРЯТИЯ
РЕСП. САХА (ЯКУТИЯ)	САХА(ЯКУТИЯ)
ОБЛ АМУРСКАЯ	АМУРСКАЯ
ХАБАРОВСКИЙ КРАЙ	ХАБАРОВСКИЙ
САНКТ-ПЕТЕРБУРГ Г	САНКТ-ПЕТЕРБУРГ
ЯМАЛО-НЕНЕЦКИЙ АО	ЯМАЛО-НЕНЕЦКИЙ
ОБЛ САМАРСКАЯ	САМАРСКАЯ
ТЮМЕНСКАЯ ОБЛАСТЬ	ТЮМЕНСКАЯ
ТВЕРСКАЯ ОБЛАСТЬ	ТВЕРСКАЯ
ЯРОСЛАВСКАЯ ОБЛАСТЬ	ЯРОСЛАВСКАЯ
ОБЛ ВЛАДИМИРСКАЯ	ВЛАДИМИРСКАЯ
ОБЛ ЛЕНИНГРАДСКАЯ	ЛЕНИНГРАДСКАЯ
ОРЛОВСКАЯ ОБЛ	ОРЛОВСКАЯ
ОБЛ КЕМЕРОВСКАЯ	КЕМЕРОВСКАЯ
ОМСКАЯ ОБЛ	ОМСКАЯ
РЕСП ЧЕЧЕНСКАЯ	ЧЕЧЕНСКАЯ
ОБЛ КУРСКАЯ	КУРСКАЯ
ТУЛЬСКАЯ ОБЛ	ТУЛЬСКАЯ
РЕСП АДЫГЕЯ	АДЫГЕЯ
ТУЛЬСКАЯ ОБЛАСТЬ	ТУЛЬСКАЯ
РЕСПУБЛИКА КОМИ	КОМИ
ПРИМОРСКИЙ КРАЙ	ПРИМОРСКИЙ
САМАРСКАЯ ОБЛ	САМАРСКАЯ
СМОЛЕНСКАЯ ОБЛАСТЬ	СМОЛЕНСКАЯ
ОБЛ КИРОВСКАЯ	КИРОВСКАЯ
САМАРСКАЯ ОБЛАСТЬ	САМАРСКАЯ
РЕСП ДАГЕСТАН	ДАГЕСТАН
ПЕНЗЕНСКАЯ ОБЛ	ПЕНЗЕНСКАЯ
ТВЕРСКАЯ ОБЛ	ТВЕРСКАЯ
УДМУРТСКАЯ РЕСПУБЛИКА	УДМУРТСКАЯ
РЕСП КАРЕЛИЯ	КАРЕЛИЯ
ОБЛ ТОМСКАЯ	ТОМСКАЯ
РЕСПУБЛИКА БУРЯТИЯ	БУРЯТИЯ
ОБЛ МАГАДАНСКАЯ	МАГАДАНСКАЯ
РЕСП КОМИ	КОМИ
ЯРОСЛАВСКАЯ ОБЛ	ЯРОСЛАВСКАЯ
ОРЛОВСКАЯ ОБЛАСТЬ	ОРЛОВСКАЯ
ТОМСКАЯ ОБЛАСТЬ	ТОМСКАЯ
РЕСП МАРИЙ ЭЛ	МАРИЙЭЛ
ОБЛ ИВАНОВСКАЯ	ИВАНОВСКАЯ
КРАЙ КРАСНОДАРСКИЙ	КРАСНОДАРСКИЙ
РЕСПУБЛИКА АДЫГЕЯ	АДЫГЕЯ
САРАТОВСКАЯ ОБЛАСТЬ	САРАТОВСКАЯ
ЕВРЕЙСКАЯ АОБЛ	ЕВРЕЙСКАЯ
ХАКАСИЯ РЕСП	ХАКАСИЯ
ПСКОВСКАЯ ОБЛ	ПСКОВСКАЯ
КРАЙ АЛТАЙСКИЙ	АЛТАЙСКИЙ
РЕСП КАБАРДИНО-БАЛКАРСКАЯ	КАБАРДИНО-БАЛКАРСКАЯ
ТЮМЕНСКАЯ ОБЛ	ТЮМЕНСКАЯ
КРАЙ ПЕРМСКИЙ	ПЕРМСКИЙ
АО ХАНТЫ-МАНСИЙСКИЙ АВТОНОМНЫЙ ОКРУГ - Ю	ХАНТЫ-МАНСИЙСКИЙАВТОНОМНЫЙОКРУГ-Ю
БАШКОРТОСТАН	БАШКОРТОСТАН
ОБЛ ТАМБОВСКАЯ	ТАМБОВСКАЯ
ТЫВА РЕСП	ТЫВА
ОБЛ НОВГОРОДСКАЯ	НОВГОРОДСКАЯ
ОБЛ ЛИПЕЦКАЯ	ЛИПЕЦКАЯ
ОБЛ ТУЛЬСКАЯ	ТУЛЬСКАЯ
ПСКОВСКАЯ ОБЛАСТЬ	ПСКОВСКАЯ
ВОЛОГОДСКАЯ ОБЛАСТЬ	ВОЛОГОДСКАЯ
САХА /ЯКУТИЯ/ РЕСП	САХА/ЯКУТИЯ/
ОМСКАЯ ОБЛАСТЬ	ОМСКАЯ
АРХАНГЕЛЬСКАЯ ОБЛАСТЬ	АРХАНГЕЛЬСКАЯ
ТАМБОВСКАЯ ОБЛ	ТАМБОВСКАЯ
РЕСП СЕВЕРНАЯ ОСЕТИЯ - АЛАНИЯ	СЕВЕРНАЯОСЕТИЯ-АЛАНИЯ
ЛЕНИНГРАДСКАЯ ОБЛ	ЛЕНИНГРАДСКАЯ
ОБЛ ТВЕРСКАЯ	ТВЕРСКАЯ
ПЕНЗЕНСКАЯ ОБЛАСТЬ	ПЕНЗЕНСКАЯ
УЛЬЯНОВСКАЯ ОБЛАСТЬ	УЛЬЯНОВСКАЯ
СМОЛЕНСКАЯ ОБЛ	СМОЛЕНСКАЯ
Г. САНКТ-ПЕТЕРБУРГ	САНКТ-ПЕТЕРБУРГ
ТОМСКАЯ ОБЛ	ТОМСКАЯ

РЯЗАНСКАЯ ОБЛ	РЯЗАНСКАЯ
САХАЛИНСКАЯ ОБЛАСТЬ	САХАЛИНСКАЯ
ЧУВАШСКАЯ РЕСПУБЛИКА - ЧУВАШИЯ	ЧУВАШСКАЯ-ЧУВАШИЯ
NaN	nan
БАШКОРТОСТАН РЕСП	БАШКОРТОСТАН
КРАЙ КАМЧАТСКИЙ	КАМЧАТСКИЙ
РЕСП МОРДОВИЯ	МОРДОВИЯ
РЕСПУБЛИКА КАЛМЫКИЯ	КАЛМЫКИЯ
РЕСП АЛТАЙ	АЛТАЙ
ОБЛ БРЯНСКАЯ	БРЯНСКАЯ
РЕСП ТАТАРСТАН	ТАТАРСТАН
МОСКОВСКАЯ ОБЛАСТЬ	МОСКОВСКАЯ
КАБАРДИНО-БАЛКАРСКАЯ РЕСП	КАБАРДИНО-БАЛКАРСКАЯ
ЧУВАШСКАЯ РЕСПУБЛИКА	ЧУВАШСКАЯ
АО ЯМАЛО-НЕНЕЦКИЙ	ЯМАЛО-НЕНЕЦКИЙ
ОБЛ ПЕНЗЕНСКАЯ	ПЕНЗЕНСКАЯ
КРАЙ ПРИМОРСКИЙ	ПРИМОРСКИЙ
САХАЛИНСКАЯ ОБЛ	САХАЛИНСКАЯ
КРАЙ КРАСНОЯРСКИЙ	КРАСНОЯРСКИЙ
ОРЕНБУРГСКАЯ ОБЛАСТЬ	ОРЕНБУРГСКАЯ
РЯЗАНСКАЯ ОБЛАСТЬ	РЯЗАНСКАЯ
РЕСПУБЛИКА КАРЕЛИЯ	КАРЕЛИЯ
ОБЛ СМОЛЕНСКАЯ	СМОЛЕНСКАЯ
ОБЛ КАЛИНИНГРАДСКАЯ	КАЛИНИНГРАДСКАЯ
РЕСПУБЛИКА ТЫВА	ТЫВА
МУРМАНСКАЯ ОБЛ	МУРМАНСКАЯ
ТАМБОВСКАЯ ОБЛАСТЬ	ТАМБОВСКАЯ
ОБЛ РОСТОВСКАЯ	РОСТОВСКАЯ
РЕСПУБЛИКА МАРИЙ ЭЛ	МАРИЙЭЛ
АСТРАХАНСКАЯ ОБЛАСТЬ	АСТРАХАНСКАЯ
КУРСКАЯ ОБЛ	КУРСКАЯ
СЕВЕРНАЯ ОСЕТИЯ - АЛАНИЯ РЕСП	СЕВЕРНАЯОСЕТИЯ-АЛАНИЯ
НИЖЕГОРОДСКАЯ ОБЛ	НИЖЕГОРОДСКАЯ
РЕСПУБЛИКА МОРДОВИЯ	МОРДОВИЯ
НОВОСИБИРСКАЯ ОБЛ	НОВОСИБИРСКАЯ
ЧУВАШСКАЯ - ЧУВАШИЯ РЕСП	ЧУВАШСКАЯ-ЧУВАШИЯ
РЕСП САХА /ЯКУТИЯ/	САХА/ЯКУТИЯ/
РЕСП КАЛМЫКИЯ	КАЛМЫКИЯ
НЕНЕЦКИЙ АО	НЕНЕЦКИЙ
ЛЕНИНГРАДСКАЯ ОБЛАСТЬ	ЛЕНИНГРАДСКАЯ
ЗАБАЙКАЛЬСКИЙ КРАЙ	ЗАБАЙКАЛЬСКИЙ
ВОЛОГОДСКАЯ	ВОЛОГОДСКАЯ
АРХАНГЕЛЬСКАЯ	АРХАНГЕЛЬСКАЯ
РЕСП УДМУРТСКАЯ	УДМУРТСКАЯ
Г МОСКВА	МОСКВА
БУРЯТИЯ РЕСП	БУРЯТИЯ
РЕСП ТЫВА	ТЫВА
ПЕРМСКАЯ ОБЛ	ПЕРМСКАЯ
ОБЛ РЯЗАНСКАЯ	РЯЗАНСКАЯ
КАЛМЫКИЯ РЕСП	КАЛМЫКИЯ
Г. МОСКВА	МОСКВА
КЕМЕРОВСКАЯ ОБЛ	КЕМЕРОВСКАЯ
КАБАРДИНО-БАЛКАРСКАЯ	КАБАРДИНО-БАЛКАРСКАЯ
КРАЙ ХАБАРОВСКИЙ	ХАБАРОВСКИЙ
ОБЛ ЯРОСЛАВСКАЯ	ЯРОСЛАВСКАЯ
МУРМАНСКАЯ ОБЛАСТЬ	МУРМАНСКАЯ
ВОЛОГОДСКАЯ ОБЛ	ВОЛОГОДСКАЯ
ОБЛ УЛЬЯНОВСКАЯ	УЛЬЯНОВСКАЯ
КЕМЕРОВСКАЯ ОБЛАСТЬ	КЕМЕРОВСКАЯ
ИРКУТСКАЯ ОБЛ	ИРКУТСКАЯ
ВЛАДИМИРСКАЯ ОБЛ	ВЛАДИМИРСКАЯ
АСТРАХАНСКАЯ ОБЛ	АСТРАХАНСКАЯ
КУРСКАЯ ОБЛАСТЬ	КУРСКАЯ
КУРГАНСКАЯ ОБЛ	КУРГАНСКАЯ
КУРГАНСКАЯ ОБЛАСТЬ	КУРГАНСКАЯ
ВОРОНЕЖСКАЯ ОБЛ	ВОРОНЕЖСКАЯ
ОБЛ САХАЛИНСКАЯ	САХАЛИНСКАЯ
БЕЛГОРОДСКАЯ ОБЛ	БЕЛГОРОДСКАЯ
РОСТОВСКАЯ	РОСТОВСКАЯ
ВОЛГОГРАДСКАЯ ОБЛАСТЬ	ВОЛГОГРАДСКАЯ
АСТРАХАНСКАЯ	АСТРАХАНСКАЯ
ЧУКОТСКИЙ АО	ЧУКОТСКИЙ

ГОРЬКОВСКАЯ ОБЛ	ГОРЬКОВСКАЯ
НОВГОРОДСКАЯ ОБЛ	НОВГОРОДСКАЯ
КАЛИНИНГРАДСКАЯ ОБЛ	КАЛИНИНГРАДСКАЯ
ОМСКАЯ	ОМСКАЯ
КОМИ РЕСП	КОМИ
СЕВ. ОСЕТИЯ - АЛАНИЯ	СЕВОСЕТИЯ-АЛАНИЯ
САМАРСКАЯ	САМАРСКАЯ
ВОЛГОГРАДСКАЯ ОБЛ	ВОЛГОГРАДСКАЯ
ОБЛ ОРЛОВСКАЯ	ОРЛОВСКАЯ
БЕЛГОРОДСКАЯ ОБЛАСТЬ	БЕЛГОРОДСКАЯ
БУРЯТИЯ	БУРЯТИЯ
ВЛАДИМИРСКАЯ ОБЛАСТЬ	ВЛАДИМИРСКАЯ
ЧУВАШИЯ ЧУВАШСКАЯ РЕСПУБЛИКА -	ЧУВАШИЯЧУВАШСКАЯ-
РЕСП ИНГУШЕТИЯ	ИНГУШЕТИЯ
КРАСНОДАРСКИЙ	КРАСНОДАРСКИЙ
АМУРСКАЯ ОБЛАСТЬ	АМУРСКАЯ
ЛЕНИНГРАДСКАЯ	ЛЕНИНГРАДСКАЯ
БРЯНСКАЯ ОБЛ	БРЯНСКАЯ
ЧЕЧЕНСКАЯ РЕСП	ЧЕЧЕНСКАЯ
ЧЕЧЕНСКАЯ РЕСПУБЛИКА	ЧЕЧЕНСКАЯ
РЕСПУБЛИКАТАТАРСТАН	РЕСПУБЛИКАТАТАРСТАН
АДЫГЕЯ РЕСП	АДЫГЕЯ
ОБЛ ОМСКАЯ	ОМСКАЯ
ТОМСКАЯ	ТОМСКАЯ
РОССИЯ	РОССИЯ
МАРИЙ ЭЛ РЕСП	МАРИЙЭЛ
КИРОВСКАЯ ОБЛ	КИРОВСКАЯ
ВОРОНЕЖСКАЯ ОБЛАСТЬ	ВОРОНЕЖСКАЯ
МОРДОВИЯ РЕСП	МОРДОВИЯ
ЛИПЕЦКАЯ ОБЛ	ЛИПЕЦКАЯ
ХАКАСИЯ	ХАКАСИЯ
КОСТРОМСКАЯ ОБЛ	КОСТРОМСКАЯ
КОМИ	КОМИ
КАЛУЖСКАЯ ОБЛ	КАЛУЖСКАЯ
ЧИТИНСКАЯ ОБЛ	ЧИТИНСКАЯ
БРЯНСКАЯ ОБЛАСТЬ	БРЯНСКАЯ
АМУРСКАЯ ОБЛ	АМУРСКАЯ
ЧУВАШСКАЯ РЕСП	ЧУВАШСКАЯ
МОСКОВСКАЯ	МОСКОВСКАЯ
КАРАЧАЕВО-ЧЕРКЕССКАЯ РЕСП	КАРАЧАЕВО-ЧЕРКЕССКАЯ
98	98
МЫТИЩИНСКИЙ Р-Н	МЫТИЩИНСКИЙР-Н
АРХАНГЕЛЬСКАЯ ОБЛ	АРХАНГЕЛЬСКАЯ
СТАВРОПОЛЬСКИЙ	СТАВРОПОЛЬСКИЙ
ИВАНОВСКАЯ ОБЛАСТЬ	ИВАНОВСКАЯ
КАЛУЖСКАЯ ОБЛАСТЬ	КАЛУЖСКАЯ
КАМЧАТСКИЙ КРАЙ	КАМЧАТСКИЙ
ОБЛ. ВЛАДИМИРСКАЯ	ВЛАДИМИРСКАЯ
РЕСПУБЛИКА ДАГЕСТАН	ДАГЕСТАН
ИВАНОВСКАЯ ОБЛ	ИВАНОВСКАЯ
КРАЙ.ПЕРМСКИЙ	ПЕРМСКИЙ
КАЛУЖСКАЯ	КАЛУЖСКАЯ
КРАЙ. СТАВРОПОЛЬСКИЙ	СТАВРОПОЛЬСКИЙ
РЕСПУБЛИКА ХАКАСИЯ	ХАКАСИЯ
РЕСПУБЛИКА САХА	САХА
КОСТРОМСКАЯ ОБЛАСТЬ	КОСТРОМСКАЯ
АО НЕНЕЦКИЙ	НЕНЕЦКИЙ
МОСКВОСКАЯ ОБЛ	МОСКВОСКАЯ
ДАГЕСТАН РЕСП	ДАГЕСТАН
КРАЙ. КРАСНОЯРСКИЙ	КРАСНОЯРСКИЙ
ОБЛ.РОСТОВСКАЯ	РОСТОВСКАЯ
ЛИПЕЦКАЯ ОБЛАСТЬ	ЛИПЕЦКАЯ
ЕВРЕЙСКАЯ АВТОНОМНАЯ	ЕВРЕЙСКАЯ
ЭВЕНКИЙСКИЙ АО	ЭВЕНКИЙСКИЙ
КАМЧАТСКАЯ ОБЛАСТЬ	КАМЧАТСКАЯ
РЕСПУБЛИКА АЛТАЙ	АЛТАЙ
АЛТАЙСКИЙ КРАЙ	АЛТАЙСКИЙ
НОВОСИБИРСКАЯ ОБЛАСТЬ	НОВОСИБИРСКАЯ
НОВГОРОДСКАЯ ОБЛАСТЬ	НОВГОРОДСКАЯ
КАРЕЛИЯ РЕСП	КАРЕЛИЯ
ГУСЬ-ХРУСТАЛЬНЫЙ Р-Н	ГУСЬ-ХРУСТАЛЬНЫЙР-Н
КАЛМЫКИЯ	КАЛМЫКИЯ

```
ИНГУШЕТИЯ РЕСП                                ИНГУШЕТИЯ
АЛТАЙСКИЙ                                     АЛТАЙСКИЙ
КИРОВСКАЯ ОБЛАСТЬ                             КИРОВСКАЯ
ОБЛ. МУРМАНСКАЯ                               МУРМАНСКАЯ
КАРЕЛИЯ                                         КАРЕЛИЯ
РЕСП. КОМИ                                        КОМИ
ОБЛ. ЛИПЕЦКАЯ                                 ЛИПЕЦКАЯ
БРЯНСКИЙ                                       БРЯНСКИЙ
ОБЛ. СВЕРДЛОВСКАЯ                           СВЕРДЛОВСКАЯ
ПЕРМСКИЙ                                       ПЕРМСКИЙ
ОРЁЛ                                             ОРЁЛ
ОБЛ.НИЖЕГОРОДСКАЯ                         НИЖЕГОРОДСКАЯ
АОБЛ ЕВРЕЙСКАЯ                                 ЕВРЕЙСКАЯ
СВЕРДЛОВСКАЯ                                 СВЕРДЛОВСКАЯ
ОБЛ.МОСКОВСКАЯ                                МОСКОВСКАЯ
ОБЛ.САРАТОВСКАЯ                               САРАТОВСКАЯ
КЕМЕРОВСКАЯ                                   КЕМЕРОВСКАЯ
ДАЛЬНИЙ ВОСТОК                             ДАЛЬНИЙВОСТОК
ОБЛ. НОВОСИБИРСКАЯ                         НОВОСИБИРСКАЯ
ОБЛ. КУРГАНСКАЯ                               КУРГАНСКАЯ
74                                                 74
ОБЛ. БЕЛГОРОДСКАЯ                           БЕЛГОРОДСКАЯ
МАГАДАНСКАЯ ОБЛАСТЬ                         МАГАДАНСКАЯ
ВОЛОГОДСКАЯ ОБЛ.                            ВОЛОГОДСКАЯ
ТЮМЕНСКАЯ                                     ТЮМЕНСКАЯ
КАРАЧАЕВО-ЧЕРКЕССКАЯ                 КАРАЧАЕВО-ЧЕРКЕССКАЯ
ОБЛ. ЧЕЛЯБИНСКАЯ                             ЧЕЛЯБИНСКАЯ
САХА /ЯКУТИЯ/                               САХА/ЯКУТИЯ/
Г.МОСКВА                                         МОСКВА
Г.ОДИНЦОВО МОСКОВСКАЯ ОБЛ                ОДИНЦОВОМОСКОВСКАЯ
НОВОСИБИРСКАЯ                                 НОВОСИБИРСКАЯ
РЕСП.БАШКОРТОСТАН                             БАШКОРТОСТАН
КРАЙ. ПЕРМСКИЙ                                 ПЕРМСКИЙ
РЕСП ЧУВАШСКАЯ - ЧУВАШИЯ                  ЧУВАШСКАЯ-ЧУВАШИЯ
ОБЛ. КИРОВСКАЯ                                 КИРОВСКАЯ
КАЛИНИНГРАДСКАЯ ОБЛ.                       КАЛИНИНГРАДСКАЯ
Name: regions, dtype: object
```

Обратите внимание, теперь пропускам (значениям `NaN`) соответствует отдельная категория `nan`.
Вносим итоговые правки.

In[14]:

```
# вносим финальные корректировки в regions
regions['ЧУКОТСКИЙ АО'] = 'ЧУКОТСКИЙ'
regions['ЧУВАШСКАЯ РЕСПУБЛИКА - ЧУВАШИЯ'] = 'ЧУВАШСКАЯ'
regions['ЧУВАШИЯ ЧУВАШСКАЯ РЕСПУБЛИКА -'] = 'ЧУВАШСКАЯ'
regions['ЧУВАШСКАЯ - ЧУВАШИЯ РЕСП'] = 'ЧУВАШСКАЯ'
regions['РЕСП ЧУВАШСКАЯ - ЧУВАШИЯ'] = 'ЧУВАШСКАЯ'
regions['ЧУВАШСКАЯ - ЧУВАШИЯ РЕСП'] = 'ЧУВАШСКАЯ'
regions['РЕСПУБЛИКАТАТАРСТАН'] = 'ТАТАРСТАН'
regions['ПРИВОЛЖСКИЙ ФЕДЕРАЛЬНЫЙ ОКРУГ'] = 'МОСКОВСКАЯ'
regions['ПЕРМСКАЯ ОБЛ'] = 'ПЕРМСКИЙ'
regions['ОРЁЛ'] = 'ОРЛОВСКАЯ'
regions['Г.ОДИНЦОВО МОСКОВСКАЯ ОБЛ'] = 'МОСКОВСКАЯ'
regions['МЫТИЩИНСКИЙ Р-Н'] = 'МОСКОВСКАЯ'
regions['МОСКОВСКИЙ П'] = 'МОСКОВСКАЯ'
regions['КАМЧАТСКАЯ ОБЛАСТЬ'] = 'КАМЧАТСКИЙ'
regions['ДАЛЬНИЙ ВОСТОК'] = 'МОСКОВСКАЯ'
regions['ДАЛЬНИЙВОСТОК'] = 'МОСКОВСКАЯ'
regions['ГУСЬ-ХРУСТАЛЬНЫЙ Р-Н'] = 'ВЛАДИМИРСКАЯ'
regions['ГОРЬКОВСКАЯ ОБЛ'] = 'НИЖЕГОРОДСКАЯ'
regions['ЭВЕНКИЙСКИЙ АО'] = 'КРАСНОЯРСКИЙ'
regions['ХАНТЫ-МАНСИЙСКИЙ АВТОНОМНЫЙ ОКРУГ - ЮГРА'] = 'ХАНТЫ-МАНСИЙСКИЙ'
regions['АО ХАНТЫ-МАНСИЙСКИЙ АВТОНОМНЫЙ ОКРУГ - Ю'] = 'ХАНТЫ-МАНСИЙСКИЙ'
regions['АО ХАНТЫ-МАНСИЙСКИЙ-ЮГРА'] = 'ХАНТЫ-МАНСИЙСКИЙ'
regions['СЕВ. ОСЕТИЯ - АЛАНИЯ'] = 'СЕВЕРНАЯОСЕТИЯ-АЛАНИЯ'
regions['РЕСП. САХА (ЯКУТИЯ)'] = 'САХА/ЯКУТИЯ/'
regions['РЕСПУБЛИКА САХА'] = 'САХА/ЯКУТИЯ/'
regions['ДАЛЬНИЙВОСТОК'] = 'МОСКОВСКАЯ'
regions['САХА'] = 'САХА/ЯКУТИЯ/'
```

```
regions['98'] = 'САНКТ-ПЕТЕРБУРГ'
regions['74'] = 'ЧЕЛЯБИНСКАЯ'
regions['РОССИЯ'] = 'МОСКОВСКАЯ'
regions['МОСКВОСКАЯ'] = 'МОСКОВСКАЯ'
regions['МОСКВОСКАЯ ОБЛ'] = 'МОСКОВСКАЯ'
regions['ЧЕЛЯБИНСК'] = 'ЧЕЛЯБИНСКАЯ'
regions['Г. ЧЕЛЯБИНСК'] = 'ЧЕЛЯБИНСКАЯ'
regions['БРЯНСКИЙ'] = 'БРЯНСКАЯ'
```

Давайте снова взглянем на нашу серию.

In[15]:

```
# вновь смотрим серию
regions
```

Out[15]:

```
КРАСНОДАРСКИЙ КРАЙ                          КРАСНОДАРСКИЙ
МОСКВА                                             МОСКВА
ОБЛ САРАТОВСКАЯ                              САРАТОВСКАЯ
ОБЛ ВОЛГОГРАДСКАЯ                           ВОЛГОГРАДСКАЯ
ЧЕЛЯБИНСКАЯ ОБЛАСТЬ                          ЧЕЛЯБИНСКАЯ
СТАВРОПОЛЬСКИЙ КРАЙ                         СТАВРОПОЛЬСКИЙ
ОБЛ НИЖЕГОРОДСКАЯ                           НИЖЕГОРОДСКАЯ
МОСКОВСКАЯ ОБЛ                               МОСКОВСКАЯ
ХАНТЫ-МАНСИЙСКИЙ АВТОНОМНЫЙ ОКРУГ - ЮГРА    ХАНТЫ-МАНСИЙСКИЙ
КРАЙ СТАВРОПОЛЬСКИЙ                          СТАВРОПОЛЬСКИЙ
САНКТ-ПЕТЕРБУРГ                              САНКТ-ПЕТЕРБУРГ
РЕСП. БАШКОРТОСТАН                           БАШКОРТОСТАН
ОБЛ АРХАНГЕЛЬСКАЯ                            АРХАНГЕЛЬСКАЯ
ХАНТЫ-МАНСИЙСКИЙ АО                          ХАНТЫ-МАНСИЙСКИЙ
РЕСП БАШКОРТОСТАН                            БАШКОРТОСТАН
ПЕРМСКИЙ КРАЙ                                    ПЕРМСКИЙ
РЕСП КАРАЧАЕВО-ЧЕРКЕССКАЯ               КАРАЧАЕВО-ЧЕРКЕССКАЯ
САРАТОВСКАЯ ОБЛ                              САРАТОВСКАЯ
ОБЛ КАЛУЖСКАЯ                                КАЛУЖСКАЯ
ОБЛ ВОЛОГОДСКАЯ                              ВОЛОГОДСКАЯ
РОСТОВСКАЯ ОБЛ                               РОСТОВСКАЯ
УДМУРТСКАЯ РЕСП                              УДМУРТСКАЯ
ОБЛ ИРКУТСКАЯ                                ИРКУТСКАЯ
ПРИВОЛЖСКИЙ ФЕДЕРАЛЬНЫЙ ОКРУГ                МОСКОВСКАЯ
ОБЛ МОСКОВСКАЯ                               МОСКОВСКАЯ
ОБЛ ТЮМЕНСКАЯ                                ТЮМЕНСКАЯ
ОБЛ БЕЛГОРОДСКАЯ                            БЕЛГОРОДСКАЯ
РОСТОВСКАЯ ОБЛАСТЬ                           РОСТОВСКАЯ
ОБЛ КОСТРОМСКАЯ                              КОСТРОМСКАЯ
РЕСП ХАКАСИЯ                                     ХАКАСИЯ
РЕСПУБЛИКА ТАТАРСТАН                          ТАТАРСТАН
ИРКУТСКАЯ ОБЛАСТЬ                             ИРКУТСКАЯ
ОБЛ СВЕРДЛОВСКАЯ                             СВЕРДЛОВСКАЯ
ОБЛ ПСКОВСКАЯ                                ПСКОВСКАЯ
КРАЙ ЗАБАЙКАЛЬСКИЙ                           ЗАБАЙКАЛЬСКИЙ
СВЕРДЛОВСКАЯ ОБЛ                             СВЕРДЛОВСКАЯ
ОБЛ ОРЕНБУРГСКАЯ                             ОРЕНБУРГСКАЯ
ОБЛ ВОРОНЕЖСКАЯ                              ВОРОНЕЖСКАЯ
ОБЛ АСТРАХАНСКАЯ                             АСТРАХАНСКАЯ
ОБЛ НОВОСИБИРСКАЯ                            НОВОСИБИРСКАЯ
ОБЛ ЧЕЛЯБИНСКАЯ                              ЧЕЛЯБИНСКАЯ
ОРЕНБУРГСКАЯ ОБЛ                             ОРЕНБУРГСКАЯ
СВЕРДЛОВСКАЯ ОБЛАСТЬ                         СВЕРДЛОВСКАЯ
ОБЛ КУРГАНСКАЯ                               КУРГАНСКАЯ
ЧЕЛЯБИНСКАЯ ОБЛ                              ЧЕЛЯБИНСКАЯ
НИЖЕГОРОДСКАЯ ОБЛАСТЬ                        НИЖЕГОРОДСКАЯ
ТАТАРСТАН РЕСП                                   ТАТАРСТАН
УЛЬЯНОВСКАЯ ОБЛ                              УЛЬЯНОВСКАЯ
МОСКВА Г                                           МОСКВА
ОБЛ МУРМАНСКАЯ                               МУРМАНСКАЯ
КРАСНОЯРСКИЙ КРАЙ                            КРАСНОЯРСКИЙ
РЕСП БУРЯТИЯ                                     БУРЯТИЯ
РЕСП. САХА (ЯКУТИЯ)                          САХА/ЯКУТИЯ/
ОБЛ АМУРСКАЯ                                     АМУРСКАЯ
ХАБАРОВСКИЙ КРАЙ                             ХАБАРОВСКИЙ
```

САНКТ-ПЕТЕРБУРГ Г	САНКТ-ПЕТЕРБУРГ
ЯМАЛО-НЕНЕЦКИЙ АО	ЯМАЛО-НЕНЕЦКИЙ
ОБЛ САМАРСКАЯ	САМАРСКАЯ
ТЮМЕНСКАЯ ОБЛАСТЬ	ТЮМЕНСКАЯ
ТВЕРСКАЯ ОБЛАСТЬ	ТВЕРСКАЯ
ЯРОСЛАВСКАЯ ОБЛАСТЬ	ЯРОСЛАВСКАЯ
ОБЛ ВЛАДИМИРСКАЯ	ВЛАДИМИРСКАЯ
ОБЛ ЛЕНИНГРАДСКАЯ	ЛЕНИНГРАДСКАЯ
ОРЛОВСКАЯ ОБЛ	ОРЛОВСКАЯ
ОБЛ КЕМЕРОВСКАЯ	КЕМЕРОВСКАЯ
ОМСКАЯ ОБЛ	ОМСКАЯ
РЕСП ЧЕЧЕНСКАЯ	ЧЕЧЕНСКАЯ
ОБЛ КУРСКАЯ	КУРСКАЯ
ТУЛЬСКАЯ ОБЛ	ТУЛЬСКАЯ
РЕСП АДЫГЕЯ	АДЫГЕЯ
ТУЛЬСКАЯ ОБЛАСТЬ	ТУЛЬСКАЯ
РЕСПУБЛИКА КОМИ	КОМИ
ПРИМОРСКИЙ КРАЙ	ПРИМОРСКИЙ
САМАРСКАЯ ОБЛ	САМАРСКАЯ
СМОЛЕНСКАЯ ОБЛАСТЬ	СМОЛЕНСКАЯ
ОБЛ КИРОВСКАЯ	КИРОВСКАЯ
САМАРСКАЯ ОБЛАСТЬ	САМАРСКАЯ
РЕСП ДАГЕСТАН	ДАГЕСТАН
ПЕНЗЕНСКАЯ ОБЛ	ПЕНЗЕНСКАЯ
ТВЕРСКАЯ ОБЛ	ТВЕРСКАЯ
УДМУРТСКАЯ РЕСПУБЛИКА	УДМУРТСКАЯ
РЕСП КАРЕЛИЯ	КАРЕЛИЯ
ОБЛ ТОМСКАЯ	ТОМСКАЯ
РЕСПУБЛИКА БУРЯТИЯ	БУРЯТИЯ
ОБЛ МАГАДАНСКАЯ	МАГАДАНСКАЯ
РЕСП КОМИ	КОМИ
ЯРОСЛАВСКАЯ ОБЛ	ЯРОСЛАВСКАЯ
ОРЛОВСКАЯ ОБЛАСТЬ	ОРЛОВСКАЯ
ТОМСКАЯ ОБЛАСТЬ	ТОМСКАЯ
РЕСП МАРИЙ ЭЛ	МАРИЙЭЛ
ОБЛ ИВАНОВСКАЯ	ИВАНОВСКАЯ
КРАЙ КРАСНОДАРСКИЙ	КРАСНОДАРСКИЙ
РЕСПУБЛИКА АДЫГЕЯ	АДЫГЕЯ
САРАТОВСКАЯ ОБЛАСТЬ	САРАТОВСКАЯ
ЕВРЕЙСКАЯ АОБЛ	ЕВРЕЙСКАЯ
ХАКАСИЯ РЕСП	ХАКАСИЯ
ПСКОВСКАЯ ОБЛ	ПСКОВСКАЯ
КРАЙ АЛТАЙСКИЙ	АЛТАЙСКИЙ
РЕСП КАБАРДИНО-БАЛКАРСКАЯ	КАБАРДИНО-БАЛКАРСКАЯ
ТЮМЕНСКАЯ ОБЛ	ТЮМЕНСКАЯ
КРАЙ ПЕРМСКИЙ	ПЕРМСКИЙ
АО ХАНТЫ-МАНСИЙСКИЙ АВТОНОМНЫЙ ОКРУГ - Ю	ХАНТЫ-МАНСИЙСКИЙ
БАШКОРТОСТАН	БАШКОРТОСТАН
ОБЛ ТАМБОВСКАЯ	ТАМБОВСКАЯ
ТЫВА РЕСП	ТЫВА
ОБЛ НОВГОРОДСКАЯ	НОВГОРОДСКАЯ
ОБЛ ЛИПЕЦКАЯ	ЛИПЕЦКАЯ
ОБЛ ТУЛЬСКАЯ	ТУЛЬСКАЯ
ПСКОВСКАЯ ОБЛАСТЬ	ПСКОВСКАЯ
ВОЛОГОДСКАЯ ОБЛАСТЬ	ВОЛОГОДСКАЯ
САХА /ЯКУТИЯ/ РЕСП	САХА/ЯКУТИЯ/
ОМСКАЯ ОБЛАСТЬ	ОМСКАЯ
АРХАНГЕЛЬСКАЯ ОБЛАСТЬ	АРХАНГЕЛЬСКАЯ
ТАМБОВСКАЯ ОБЛ	ТАМБОВСКАЯ
РЕСП СЕВЕРНАЯ ОСЕТИЯ - АЛАНИЯ	СЕВЕРНАЯОСЕТИЯ-АЛАНИЯ
ЛЕНИНГРАДСКАЯ ОБЛ	ЛЕНИНГРАДСКАЯ
ОБЛ ТВЕРСКАЯ	ТВЕРСКАЯ
ПЕНЗЕНСКАЯ ОБЛАСТЬ	ПЕНЗЕНСКАЯ
УЛЬЯНОВСКАЯ ОБЛАСТЬ	УЛЬЯНОВСКАЯ
СМОЛЕНСКАЯ ОБЛ	СМОЛЕНСКАЯ
Г. САНКТ-ПЕТЕРБУРГ	САНКТ-ПЕТЕРБУРГ
ТОМСКАЯ ОБЛ	ТОМСКАЯ
РЯЗАНСКАЯ ОБЛ	РЯЗАНСКАЯ
САХАЛИНСКАЯ ОБЛАСТЬ	САХАЛИНСКАЯ
ЧУВАШСКАЯ РЕСПУБЛИКА - ЧУВАШИЯ	ЧУВАШСКАЯ
NaN	nan
БАШКОРТОСТАН РЕСП	БАШКОРТОСТАН

КРАЙ КАМЧАТСКИЙ	КАМЧАТСКИЙ
РЕСП МОРДОВИЯ	МОРДОВИЯ
РЕСПУБЛИКА КАЛМЫКИЯ	КАЛМЫКИЯ
РЕСП АЛТАЙ	АЛТАЙ
ОБЛ БРЯНСКАЯ	БРЯНСКАЯ
РЕСП ТАТАРСТАН	ТАТАРСТАН
МОСКОВСКАЯ ОБЛАСТЬ	МОСКОВСКАЯ
КАБАРДИНО-БАЛКАРСКАЯ РЕСП	КАБАРДИНО-БАЛКАРСКАЯ
ЧУВАШСКАЯ РЕСПУБЛИКА	ЧУВАШСКАЯ
АО ЯМАЛО-НЕНЕЦКИЙ	ЯМАЛО-НЕНЕЦКИЙ
ОБЛ ПЕНЗЕНСКАЯ	ПЕНЗЕНСКАЯ
КРАЙ ПРИМОРСКИЙ	ПРИМОРСКИЙ
САХАЛИНСКАЯ ОБЛ	САХАЛИНСКАЯ
КРАЙ КРАСНОЯРСКИЙ	КРАСНОЯРСКИЙ
ОРЕНБУРГСКАЯ ОБЛАСТЬ	ОРЕНБУРГСКАЯ
РЯЗАНСКАЯ ОБЛАСТЬ	РЯЗАНСКАЯ
РЕСПУБЛИКА КАРЕЛИЯ	КАРЕЛИЯ
ОБЛ СМОЛЕНСКАЯ	СМОЛЕНСКАЯ
ОБЛ КАЛИНИНГРАДСКАЯ	КАЛИНИНГРАДСКАЯ
РЕСПУБЛИКА ТЫВА	ТЫВА
МУРМАНСКАЯ ОБЛ	МУРМАНСКАЯ
ТАМБОВСКАЯ ОБЛАСТЬ	ТАМБОВСКАЯ
ОБЛ РОСТОВСКАЯ	РОСТОВСКАЯ
РЕСПУБЛИКА МАРИЙ ЭЛ	МАРИЙЭЛ
АСТРАХАНСКАЯ ОБЛАСТЬ	АСТРАХАНСКАЯ
КУРСКАЯ ОБЛ	КУРСКАЯ
СЕВЕРНАЯ ОСЕТИЯ - АЛАНИЯ РЕСП	СЕВЕРНАЯОСЕТИЯ-АЛАНИЯ
НИЖЕГОРОДСКАЯ ОБЛ	НИЖЕГОРОДСКАЯ
РЕСПУБЛИКА МОРДОВИЯ	МОРДОВИЯ
НОВОСИБИРСКАЯ ОБЛ	НОВОСИБИРСКАЯ
ЧУВАШСКАЯ - ЧУВАШИЯ РЕСП	ЧУВАШСКАЯ
РЕСП САХА /ЯКУТИЯ/	САХА/ЯКУТИЯ/
РЕСП КАЛМЫКИЯ	КАЛМЫКИЯ
НЕНЕЦКИЙ АО	НЕНЕЦКИЙ
ЛЕНИНГРАДСКАЯ ОБЛАСТЬ	ЛЕНИНГРАДСКАЯ
ЗАБАЙКАЛЬСКИЙ КРАЙ	ЗАБАЙКАЛЬСКИЙ
ВОЛОГОДСКАЯ	ВОЛОГОДСКАЯ
АРХАНГЕЛЬСКАЯ	АРХАНГЕЛЬСКАЯ
РЕСП УДМУРТСКАЯ	УДМУРТСКАЯ
Г МОСКВА	МОСКВА
БУРЯТИЯ РЕСП	БУРЯТИЯ
РЕСП ТЫВА	ТЫВА
ПЕРМСКАЯ ОБЛ	ПЕРМСКИЙ
ОБЛ РЯЗАНСКАЯ	РЯЗАНСКАЯ
КАЛМЫКИЯ РЕСП	КАЛМЫКИЯ
Г. МОСКВА	МОСКВА
КЕМЕРОВСКАЯ ОБЛ	КЕМЕРОВСКАЯ
КАБАРДИНО-БАЛКАРСКАЯ	КАБАРДИНО-БАЛКАРСКАЯ
КРАЙ ХАБАРОВСКИЙ	ХАБАРОВСКИЙ
ОБЛ ЯРОСЛАВСКАЯ	ЯРОСЛАВСКАЯ
МУРМАНСКАЯ ОБЛАСТЬ	МУРМАНСКАЯ
ВОЛОГОДСКАЯ ОБЛ	ВОЛОГОДСКАЯ
ОБЛ УЛЬЯНОВСКАЯ	УЛЬЯНОВСКАЯ
КЕМЕРОВСКАЯ ОБЛАСТЬ	КЕМЕРОВСКАЯ
ИРКУТСКАЯ ОБЛ	ИРКУТСКАЯ
ВЛАДИМИРСКАЯ ОБЛ	ВЛАДИМИРСКАЯ
АСТРАХАНСКАЯ ОБЛ	АСТРАХАНСКАЯ
КУРСКАЯ ОБЛАСТЬ	КУРСКАЯ
КУРГАНСКАЯ ОБЛ	КУРГАНСКАЯ
КУРГАНСКАЯ ОБЛАСТЬ	КУРГАНСКАЯ
ВОРОНЕЖСКАЯ ОБЛ	ВОРОНЕЖСКАЯ
ОБЛ САХАЛИНСКАЯ	САХАЛИНСКАЯ
БЕЛГОРОДСКАЯ ОБЛ	БЕЛГОРОДСКАЯ
РОСТОВСКАЯ	РОСТОВСКАЯ
ВОЛГОГРАДСКАЯ ОБЛАСТЬ	ВОЛГОГРАДСКАЯ
АСТРАХАНСКАЯ	АСТРАХАНСКАЯ
ЧУКОТСКИЙ АО	ЧУКОТСКИЙ
ГОРЬКОВСКАЯ ОБЛ	НИЖЕГОРОДСКАЯ
НОВГОРОДСКАЯ ОБЛ	НОВГОРОДСКАЯ
КАЛИНИНГРАДСКАЯ ОБЛ	КАЛИНИНГРАДСКАЯ
ОМСКАЯ	ОМСКАЯ
КОМИ РЕСП	КОМИ

СЕВ. ОСЕТИЯ - АЛАНИЯ	СЕВЕРНАЯОСЕТИЯ-АЛАНИЯ
САМАРСКАЯ	САМАРСКАЯ
ВОЛГОГРАДСКАЯ ОБЛ	ВОЛГОГРАДСКАЯ
ОБЛ ОРЛОВСКАЯ	ОРЛОВСКАЯ
БЕЛГОРОДСКАЯ ОБЛАСТЬ	БЕЛГОРОДСКАЯ
БУРЯТИЯ	БУРЯТИЯ
ВЛАДИМИРСКАЯ ОБЛАСТЬ	ВЛАДИМИРСКАЯ
ЧУВАШИЯ ЧУВАШСКАЯ РЕСПУБЛИКА -	ЧУВАШСКАЯ
РЕСП ИНГУШЕТИЯ	ИНГУШЕТИЯ
КРАСНОДАРСКИЙ	КРАСНОДАРСКИЙ
АМУРСКАЯ ОБЛАСТЬ	АМУРСКАЯ
ЛЕНИНГРАДСКАЯ	ЛЕНИНГРАДСКАЯ
БРЯНСКАЯ ОБЛ	БРЯНСКАЯ
ЧЕЧЕНСКАЯ РЕСП	ЧЕЧЕНСКАЯ
ЧЕЧЕНСКАЯ РЕСПУБЛИКА	ЧЕЧЕНСКАЯ
РЕСПУБЛИКАТАТАРСТАН	ТАТАРСТАН
АДЫГЕЯ РЕСП	АДЫГЕЯ
ОБЛ ОМСКАЯ	ОМСКАЯ
ТОМСКАЯ	ТОМСКАЯ
РОССИЯ	МОСКОВСКАЯ
МАРИЙ ЭЛ РЕСП	МАРИЙЭЛ
КИРОВСКАЯ ОБЛ	КИРОВСКАЯ
ВОРОНЕЖСКАЯ ОБЛАСТЬ	ВОРОНЕЖСКАЯ
МОРДОВИЯ РЕСП	МОРДОВИЯ
ЛИПЕЦКАЯ ОБЛ	ЛИПЕЦКАЯ
ХАКАСИЯ	ХАКАСИЯ
КОСТРОМСКАЯ ОБЛ	КОСТРОМСКАЯ
КОМИ	КОМИ
КАЛУЖСКАЯ ОБЛ	КАЛУЖСКАЯ
ЧИТИНСКАЯ ОБЛ	ЧИТИНСКАЯ
БРЯНСКАЯ ОБЛАСТЬ	БРЯНСКАЯ
АМУРСКАЯ ОБЛ	АМУРСКАЯ
ЧУВАШСКАЯ РЕСП	ЧУВАШСКАЯ
МОСКОВСКАЯ	МОСКОВСКАЯ
КАРАЧАЕВО-ЧЕРКЕССКАЯ РЕСП	КАРАЧАЕВО-ЧЕРКЕССКАЯ
98	САНКТ-ПЕТЕРБУРГ
МЫТИЩИНСКИЙ Р-Н	МОСКОВСКАЯ
АРХАНГЕЛЬСКАЯ ОБЛ	АРХАНГЕЛЬСКАЯ
СТАВРОПОЛЬСКИЙ	СТАВРОПОЛЬСКИЙ
ИВАНОВСКАЯ ОБЛАСТЬ	ИВАНОВСКАЯ
КАЛУЖСКАЯ ОБЛАСТЬ	КАЛУЖСКАЯ
КАМЧАТСКИЙ КРАЙ	КАМЧАТСКИЙ
ОБЛ. ВЛАДИМИРСКАЯ	ВЛАДИМИРСКАЯ
РЕСПУБЛИКА ДАГЕСТАН	ДАГЕСТАН
ИВАНОВСКАЯ ОБЛ	ИВАНОВСКАЯ
КРАЙ.ПЕРМСКИЙ	ПЕРМСКИЙ
КАЛУЖСКАЯ	КАЛУЖСКАЯ
КРАЙ. СТАВРОПОЛЬСКИЙ	СТАВРОПОЛЬСКИЙ
РЕСПУБЛИКА ХАКАСИЯ	ХАКАСИЯ
РЕСПУБЛИКА САХА	САХА/ЯКУТИЯ/
КОСТРОМСКАЯ ОБЛАСТЬ	КОСТРОМСКАЯ
АО НЕНЕЦКИЙ	НЕНЕЦКИЙ
МОСКВОСКАЯ ОБЛ	МОСКОВСКАЯ
ДАГЕСТАН РЕСП	ДАГЕСТАН
КРАЙ. КРАСНОЯРСКИЙ	КРАСНОЯРСКИЙ
ОБЛ.РОСТОВСКАЯ	РОСТОВСКАЯ
ЛИПЕЦКАЯ ОБЛАСТЬ	ЛИПЕЦКАЯ
ЕВРЕЙСКАЯ АВТОНОМНАЯ	ЕВРЕЙСКАЯ
ЭВЕНКИЙСКИЙ АО	КРАСНОЯРСКИЙ
КАМЧАТСКАЯ ОБЛАСТЬ	КАМЧАТСКИЙ
РЕСПУБЛИКА АЛТАЙ	АЛТАЙ
АЛТАЙСКИЙ КРАЙ	АЛТАЙСКИЙ
НОВОСИБИРСКАЯ ОБЛАСТЬ	НОВОСИБИРСКАЯ
НОВГОРОДСКАЯ ОБЛАСТЬ	НОВГОРОДСКАЯ
КАРЕЛИЯ РЕСП	КАРЕЛИЯ
ГУСЬ-ХРУСТАЛЬНЫЙ Р-Н	ВЛАДИМИРСКАЯ
КАЛМЫКИЯ	КАЛМЫКИЯ
ИНГУШЕТИЯ РЕСП	ИНГУШЕТИЯ
АЛТАЙСКИЙ	АЛТАЙСКИЙ
КИРОВСКАЯ ОБЛАСТЬ	КИРОВСКАЯ
ОБЛ. МУРМАНСКАЯ	МУРМАНСКАЯ
КАРЕЛИЯ	КАРЕЛИЯ

```
РЕСП. КОМИ                                  КОМИ
ОБЛ. ЛИПЕЦКАЯ                           ЛИПЕЦКАЯ
БРЯНСКИЙ                                 БРЯНСКАЯ
ОБЛ. СВЕРДЛОВСКАЯ                     СВЕРДЛОВСКАЯ
ПЕРМСКИЙ                                 ПЕРМСКИЙ
ОРЁЛ                                     ОРЛОВСКАЯ
ОБЛ.НИЖЕГОРОДСКАЯ                    НИЖЕГОРОДСКАЯ
АОБЛ ЕВРЕЙСКАЯ                           ЕВРЕЙСКАЯ
СВЕРДЛОВСКАЯ                           СВЕРДЛОВСКАЯ
ОБЛ.МОСКОВСКАЯ                         МОСКОВСКАЯ
ОБЛ.САРАТОВСКАЯ                       САРАТОВСКАЯ
КЕМЕРОВСКАЯ                           КЕМЕРОВСКАЯ
ДАЛЬНИЙ ВОСТОК                         МОСКОВСКАЯ
ОБЛ. НОВОСИБИРСКАЯ                    НОВОСИБИРСКАЯ
ОБЛ. КУРГАНСКАЯ                         КУРГАНСКАЯ
74                                     ЧЕЛЯБИНСКАЯ
ОБЛ. БЕЛГОРОДСКАЯ                     БЕЛГОРОДСКАЯ
МАГАДАНСКАЯ ОБЛАСТЬ                   МАГАДАНСКАЯ
ВОЛОГОДСКАЯ ОБЛ.                       ВОЛОГОДСКАЯ
ТЮМЕНСКАЯ                               ТЮМЕНСКАЯ
КАРАЧАЕВО-ЧЕРКЕССКАЯ              КАРАЧАЕВО-ЧЕРКЕССКАЯ
ОБЛ. ЧЕЛЯБИНСКАЯ                       ЧЕЛЯБИНСКАЯ
САХА /ЯКУТИЯ/                        САХА/ЯКУТИЯ/
Г.МОСКВА                                  МОСКВА
Г.ОДИНЦОВО МОСКОВСКАЯ ОБЛ              МОСКОВСКАЯ
НОВОСИБИРСКАЯ                         НОВОСИБИРСКАЯ
РЕСП.БАШКОРТОСТАН                     БАШКОРТОСТАН
КРАЙ. ПЕРМСКИЙ                           ПЕРМСКИЙ
РЕСП ЧУВАШСКАЯ - ЧУВАШИЯ                ЧУВАШСКАЯ
ОБЛ. КИРОВСКАЯ                          КИРОВСКАЯ
КАЛИНИНГРАДСКАЯ ОБЛ.                КАЛИНИНГРАДСКАЯ
ЧУКОТСКИЙ АО                            ЧУКОТСКИЙ
МОСКОВСКИЙ П                           МОСКОВСКАЯ
ДАЛЬНИЙВОСТОК                          МОСКОВСКАЯ
АО ХАНТЫ-МАНСИЙСКИЙ-ЮГРА            ХАНТЫ-МАНСИЙСКИЙ
САХА                                 САХА/ЯКУТИЯ/
МОСКВОСКАЯ                             МОСКОВСКАЯ
ЧЕЛЯБИНСК                              ЧЕЛЯБИНСКАЯ
Г. ЧЕЛЯБИНСК                           ЧЕЛЯБИНСКАЯ
Name: regions, dtype: object
```

Теперь заменяем исходные категории переменной *living_region* на новые с помощью нашей серии.

In[16]:
```
# заменяем исходные категории переменной
# living_region на новые
data['living_region'] = data['living_region'].map(regions)
```

Взглянем на уникальные значения преобразованной переменной *living_region*.

In[17]:
```
# смотрим уникальные значения
# по переменной living_region
data['living_region'].unique()
```

Out[17]:
```
array(['КРАСНОДАРСКИЙ', 'МОСКВА', 'САРАТОВСКАЯ', 'ВОЛГОГРАДСКАЯ',
       'ЧЕЛЯБИНСКАЯ', 'СТАВРОПОЛЬСКИЙ', 'НИЖЕГОРОДСКАЯ', 'МОСКОВСКАЯ',
       'ХАНТЫ-МАНСИЙСКИЙ', 'САНКТ-ПЕТЕРБУРГ', 'БАШКОРТОСТАН',
       'АРХАНГЕЛЬСКАЯ', 'ПЕРМСКИЙ', 'КАРАЧАЕВО-ЧЕРКЕССКАЯ', 'КАЛУЖСКАЯ',
       'ВОЛОГОДСКАЯ', 'РОСТОВСКАЯ', 'УДМУРТСКАЯ', 'ИРКУТСКАЯ',
       'ТЮМЕНСКАЯ', 'БЕЛГОРОДСКАЯ', 'КОСТРОМСКАЯ', 'ХАКАСИЯ', 'ТАТАРСТАН',
       'СВЕРДЛОВСКАЯ', 'ПСКОВСКАЯ', 'ЗАБАЙКАЛЬСКИЙ', 'ОРЕНБУРГСКАЯ',
       'ВОРОНЕЖСКАЯ', 'АСТРАХАНСКАЯ', 'НОВОСИБИРСКАЯ', 'КУРГАНСКАЯ',
       'УЛЬЯНОВСКАЯ', 'МУРМАНСКАЯ', 'КРАСНОЯРСКИЙ', 'БУРЯТИЯ',
       'САХА/ЯКУТИЯ/', 'АМУРСКАЯ', 'ХАБАРОВСКИЙ', 'ЯМАЛО-НЕНЕЦКИЙ',
```

```
'САМАРСКАЯ', 'ТВЕРСКАЯ', 'ЯРОСЛАВСКАЯ', 'ВЛАДИМИРСКАЯ',
'ЛЕНИНГРАДСКАЯ', 'ОРЛОВСКАЯ', 'КЕМЕРОВСКАЯ', 'ОМСКАЯ', 'ЧЕЧЕНСКАЯ',
'КУРСКАЯ', 'ТУЛЬСКАЯ', 'АДЫГЕЯ', 'КОМИ', 'ПРИМОРСКИЙ',
'СМОЛЕНСКАЯ', 'КИРОВСКАЯ', 'ДАГЕСТАН', 'ПЕНЗЕНСКАЯ', 'КАРЕЛИЯ',
'ТОМСКАЯ', 'МАГАДАНСКАЯ', 'МАРИЙЭЛ', 'ИВАНОВСКАЯ', 'ЕВРЕЙСКАЯ',
'АЛТАЙСКИЙ', 'КАБАРДИНО-БАЛКАРСКАЯ', 'ТАМБОВСКАЯ', 'ТЫВА',
'НОВГОРОДСКАЯ', 'ЛИПЕЦКАЯ', 'СЕВЕРНАЯОСЕТИЯ-АЛАНИЯ', 'РЯЗАНСКАЯ',
'САХАЛИНСКАЯ', 'ЧУВАШСКАЯ', 'nan', 'КАМЧАТСКИЙ', 'МОРДОВИЯ',
'КАЛМЫКИЯ', 'АЛТАЙ', 'БРЯНСКАЯ', 'КАЛИНИНГРАДСКАЯ', 'НЕНЕЦКИЙ',
'ЧУКОТСКИЙ', 'ИНГУШЕТИЯ', 'ЧИТИНСКАЯ'], dtype=object)
```

Вновь выведем информацию о количестве уникальных значений переменной *living_region*.

```
In[18]:
# смотрим количество уникальных значений
# переменной living_region
data['living_region'].nunique()

Out[18]:
85
```

Теперь взглянем на типы переменных с помощью метода `.info()`.

```
In[19]:
# смотрим типы переменных
print(data.info())

Out[19]:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 170746 entries, 0 to 170745
Data columns (total 14 columns):
gender                  170746 non-null object
age                     170743 non-null float64
marital_status          170743 non-null object
job_position            170746 non-null object
credit_sum              170744 non-null object
credit_month            170746 non-null int64
tariff_id               170746 non-null float64
score_shk               170739 non-null object
education               170741 non-null object
living_region           170746 non-null object
monthly_income          170741 non-null float64
credit_count            161516 non-null float64
overdue_credit_count    161516 non-null float64
open_account_flg        170746 non-null int64
dtypes: float64(5), int64(2), object(7)
memory usage: 18.2+ MB
None
```

Видно, что категориальные переменные *tariff_id* и *open_account_flg* неверно записаны как количественные переменные: первой присвоен тип `float` (используется для представления чисел с плавающей точкой), а второй – тип `int` (используется для представления целых чисел). Количественные переменные *credit_sum* и *score_shk*, наоборот, неверно записаны как категориальные переменные, им присвоен тип `object`. Это обусловлено тем, что вместо точки в качестве десятичного разделителя использовалась запятая. Кроме того, мы видим, что многие переменные имеют пропуски.

Преобразуем категориальные переменные *tariff_id* и *open_account_flg* в тип `object`. Можно было преобразовать в тип `str`, однако разница заключается в том, что если у переменной есть пропуски, то при

преобразовании в тип `object` пропуски так и остаются пропусками и нуждаются в импутации, а по итогам преобразования в тип `str` пропуски сформируют отдельную категорию `nan`. В ряде случаев это очень удобно, потому что часто пропуски для категориальных переменных выделяют в отдельную категорию. Для компактности программного кода воспользуемся циклом `for`.

In[20]:
```
# преобразуем указанные переменные в тип object
for i in ['tariff_id', 'open_account_flg']:
    data[i] = data[i].astype('object')
```

ДОПОЛНИТЕЛЬНАЯ ИНФОРМАЦИЯ

Если необходимо изменить тип для одной переменной, можно воспользоваться следующим программным кодом:

```
data['gender'] = data['gender'].astype('str')
```

Теперь заменим запятые на точки в переменных *credit_sum* и *score_shk* и присвоим переменным тип `float`. Вновь воспользуемся циклом `for` и связыванием методов (method chaining).

In[21]:
```
# в указанных переменных заменяем запятые на точки и
# преобразуем в тип float
for i in ['credit_sum', 'score_shk']:
    data[i] = data[i].str.replace(',', '.').astype('float')
```

Снова взглянем на типы переменных.

In[22]:
```
# смотрим типы переменных
print(data.info())
```

Out[22]:
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 170746 entries, 0 to 170745
Data columns (total 14 columns):
gender                  170746 non-null object
age                     170743 non-null float64
marital_status          170743 non-null object
job_position            170746 non-null object
credit_sum              170744 non-null float64
credit_month            170746 non-null int64
tariff_id               170746 non-null object
score_shk               170739 non-null float64
education               170741 non-null object
living_region           170746 non-null object
monthly_income          170741 non-null float64
credit_count            161516 non-null float64
overdue_credit_count    161516 non-null float64
open_account_flg        170746 non-null object
dtypes: float64(6), int64(1), object(7)
memory usage: 18.2+ MB
None
```

Теперь мы видим, что категориальные переменные *tariff_id* и *open_account_flg* верно записаны как категориальные переменные: им присвоен тип `object`. Количественные переменные *credit_sum* и *score_shk* верно записаны как количественные, им присвоен тип `float`.

Давайте посмотрим наблюдения.

In[23]:
```
# выводим первые 5 наблюдений датафрейма
data.head()
```

Out[23]:

	gender	age	marital_status	job_position	credit_sum	credit_month	tariff_id	score_shk	education	living_region	monthly_income
0	M	NaN	NaN	UMN	59998.00	10	1.6	NaN	GRD	КРАСНОДАРСКИЙ	30000.0
1	F	NaN	MAR	UMN	10889.00	6	1.1	NaN	NaN	МОСКВА	NaN
2	M	32.0	MAR	SPC	10728.00	12	1.1	NaN	NaN	САРАТОВСКАЯ	NaN
3	F	27.0	NaN	SPC	12009.09	12	1.1	NaN	NaN	ВОЛГОГРАДСКАЯ	NaN
4	M	45.0	NaN	SPC	NaN	10	1.1	0.421385	SCH	ЧЕЛЯБИНСКАЯ	NaN

Видим, что теперь в переменных *credit_sum* и *score_shk* в качестве десятичного разделителя используется точка, а не запятая.

Переименование категорий переменных

Иногда бывают ситуации, когда нужно переименовать категории переменной. Для этого можно использовать метод `map`, передав ему в качестве аргумента словарь вида `{старое название категории: новое название категории}`.
Давайте изменим значения переменной *gender*.

In[24]:
```
# создаем словарь, в котором ключом является старое
# название категории, значением - новое название
# категории
d = {'M':'Male', 'F':'Female'}

# передаем словарь в метод map
data['gender'] = data['gender'].map(d)
data.head()
```

Out[24]:

	gender	age	marital_status	job_position	credit_sum	credit_month	tariff_id	score_shk	education	living_region	monthly_income
0	Male	NaN	NaN	UMN	59998.00	10	1.6	NaN	GRD	КРАСНОДАРСКИЙ	30000.0
1	Female	NaN	MAR	UMN	10889.00	6	1.1	NaN	NaN	МОСКВА	NaN
2	Male	32.0	MAR	SPC	10728.00	12	1.1	NaN	NaN	САРАТОВСКАЯ	NaN
3	Female	27.0	NaN	SPC	12009.09	12	1.1	NaN	NaN	ВОЛГОГРАДСКАЯ	NaN
4	Male	45.0	NaN	SPC	NaN	10	1.1	0.421385	SCH	ЧЕЛЯБИНСКАЯ	NaN

Аналогичную операцию можно выполнить с помощью метода `replace`.

In[25]:
```
# создаем словарь, в котором ключом является старое
# название категории, значением - новое название
# категории
f = {'Male':'M', 'Female':'F'}

# передаем в метод replace словарь, в котором ключом
# будет название переменной, а значением - словарь со
# старыми и новыми названиями категорий
data = data.replace({'gender': f})
data.head()
```

Out[25]:

	gender	age	marital_status	job_position	credit_sum	credit_month	tariff_id	score_shk	education	living_region	monthly_income
0	M	NaN	NaN	UMN	59998.00	10	1.6	NaN	GRD	КРАСНОДАРСКИЙ	30000.0
1	F	NaN	MAR	UMN	10889.00	6	1.1	NaN	NaN	МОСКВА	NaN
2	M	32.0	MAR	SPC	10728.00	12	1.1	NaN	NaN	САРАТОВСКАЯ	NaN
3	F	27.0	NaN	SPC	12009.09	12	1.1	NaN	NaN	ВОЛГОГРАДСКАЯ	NaN
4	M	45.0	NaN	SPC	NaN	10	1.1	0.421385	SCH	ЧЕЛЯБИНСКАЯ	NaN

Обработка редких категорий

Часто бывает, что наши переменные содержат редкие категории. Редкие категории являются источником шума в данных, который негативно повлияет на качество модели. Кроме того, при разбиении набора данных на обучающую и контрольную выборки, может оказаться, что данная категория отсутствует в обучающей выборке, но присутствует в контрольной выборке. Это вызовет проблемы при моделировании. Например, регрессионная модель, встретив в новых данных наблюдение с неизвестной категорией предиктора, не сможет вычислить прогноз, потому что необходимый для прогноза регрессионный коэффициент по этой категории предиктора будет отсутствовать.

Обработка редких категорий выполняется либо до разбиения на обучающую и контрольную выборки, либо после него в зависимости от причин, обусловивших появление таких категорий.

Если переменная содержит 2-3 редких категорий небольшой частоты, скорее всего, такие категории случайны и могли быть обусловлены ошибками ввода. В таком случае эти категории, как правило, объединяют с самой часто встречающейся категорией или объединяют по смыслу (например, у нас есть категории AC, BD, KD и редкая категория KF, последняя категория является скорее всего результатом ошибки ввода, клавиши D и F находятся рядом, и ее можно заменить на KD). Это можно сделать как до разбиения на обучение и контроль, так и после него.

Если переменная содержит множество категорий небольшой частоты, нам необходимо задать порог укрупнения – минимальное количество наблюдений в категории, ниже которого категория объявляется редкой. Поэтому для объективности решение о выборе такого порога должно приниматься уже после разбиения на обучающую и контрольную выборки. В противном случае получится, что решение о выборе порога мы принимали с учетом информации «из будущего». Однако на практике из соображений удобства часто такое укрупнение делают до разбиения.

Множественные редкие категории часто объединяют в одну отдельную категорию, если подтверждается о гипотеза о том, что редкие категории описывают определенный паттерн. Например, в кредитном скоринге укрупнение редких категорий в отдельные категории нередко улучшает результат. Редкие типы кредитов могут соответствовать кредитам, выданным на эксклюзивных условиях, такие кредиты выдаются людям

с хорошей кредитной историей и, таким образом, объединив редкие категорию в отдельную группу, мы выделяем группу заемщиков с лучшим кредитным статусом.
Часто редкие категории объединяют с уже существующими категориями по результатам, полученным с помощью метода CHAID. Кроме того, применяется случайное присвоение редких категорий уже существующим категориям.
Давайте посмотрим распределение значений по всем категориальным переменным.

```
In[26]:
# выводим частоты категорий по каждой
# категориальной переменной
categorical_columns = [c for c in data.columns if data[c].dtype.name == 'object']
for c in categorical_columns:
    print(data[c].value_counts())
```

Внимательный анализ показывает, что переменные `job_position`, `tariff_id` и `living_region` содержат множественные редкие категории.
Выведем частоты категорий по переменной *job_position* с помощью метода `.value_counts()`. Обратите внимание, по умолчанию пропуски не выводятся и чтобы их вывести, необходимо для параметра `dropna` метода `.value_counts()` задать значение `False`.

```
In[27]:
# выводим частоты категорий для переменной job_position,
# dropna=False выведет частоту пропусков, если они есть
print(data['job_position'].value_counts(dropna=False))

Out[27]:
SPC    134680
UMN     17674
BIS      5591
PNA      4107
DIR      3750
ATP      2791
WRK       656
NOR       537
WOI       352
INP       241
BIU       126
WRP       110
PNI        65
PNV        40
PNS        12
HSK         8
INV         5
ONB         1
Name: job_position, dtype: int64
```

Видно, что переменная *job_position* не содержит пропусков. Все категории переменной *job_position* с частотой менее 55 наблюдений (это категории `PNV`, `PNS`, `HSK`, `INV` и `ONB`) мы объединим в отдельную категорию `OTHER`.

In[28]:
```
# записываем указанные категории переменной
# job_position в отдельную категорию OTHER
data.at[data['job_position'] == 'PNV', 'job_position'] = 'OTHER'
data.at[data['job_position'] == 'PNS', 'job_position'] = 'OTHER'
data.at[data['job_position'] == 'HSK', 'job_position'] = 'OTHER'
data.at[data['job_position'] == 'INV', 'job_position'] = 'OTHER'
data.at[data['job_position'] == 'ONB', 'job_position'] = 'OTHER'
```

Снова выведем частоты категорий по переменной *job_position*.

In[29]:
```
# выводим частоты категорий для переменной job_position
print(data['job_position'].value_counts(dropna=False))
```

```
SPC      134680
UMN       17674
BIS        5591
PNA        4107
DIR        3750
ATP        2791
WRK         656
NOR         537
WOI         352
INP         241
BIU         126
WRP         110
OTHER        66
PNI          65
Name: job_position, dtype: int64
```

Теперь выведем частоты категорий по переменной *tariff_id*.

In[30]:
```
# выводим частоты категорий для переменной tariff_id
print(data['tariff_id'].value_counts(dropna=False))
```

Out[30]:

```
 1.10    69355
 1.60    39117
 1.32    15537
 1.40    10970
 1.50     7497
 1.90     5538
 1.43     3930
 1.30     3339
 1.16     3232
 1.00     2245
 1.44     2228
 1.19     2102
 1.20     1306
 1.70     1007
 1.17      717
 1.21      579
 1.94      414
 1.22      376
 1.23      370
 1.91      317
 1.24      303
 1.41      132
 1.25       56
 1.18       36
 1.26       11
 1.28       10
 1.52        7
 1.27        6
 1.48        5
 1.56        2
 1.96        1
 1.29        1
Name: tariff_id, dtype: int64
```

Как и переменная *job_position*, переменная *tariff_id* не содержит пропусков. Все категории переменной *tariff_id* с частотой менее 55 наблюдений (категории `1.29`, `1.96`, `1.56`, `1.48`, `1.27`, `1.52`, `1.28`, `1.26` и `1.18`) объединим в отдельную категорию `1.99`. Кроме того на основе категориальной переменной *tariff_id* создадим количественную переменную *tariff*. Затем переменной *tariff_id* присвоим тип `str` и заменим в ее значениях точки на символы нижнего подчеркивания. Это

обусловлено тем, что потом мы воспользуемся библиотекой catboost и мы должны указать алгоритму CatBoost индексы наших категориальных признаков, но из-за значений вида `1.22`, `1.23` CatBoost, ожидая увидеть целочисленные или строковые переменные, примет нашу переменную за вещественную и выдаст ошибку `cat_features must be integer or string, real number values and NaN values should be converted to string`.

In[31]:
```
# все категории переменной tariff_id с частотой
# менее 55 наблюдений записываем в отдельную
# категорию 1.99
data.loc[data['tariff_id'].value_counts()[data['tariff_id']].values < 55,
         'tariff_id'] = 1.99

# на основе категориальной переменной tariff_id создаем
# количественную переменную tariff
data['tariff'] = data['tariff_id'].astype('float')

# заменим точки на символы подчеркивания
data['tariff_id'] = data['tariff_id'].astype('str').str.replace('.', '_')
```

Обратите внимание, что этот программный код, выполняющий укрупнение категорий, сработает только в том случае, когда у вас отсутствуют значения `NaN`, в противном случае будет выдана ошибка, поскольку нельзя выполнить индексацию с помощью вектора, содержащего значения `NA/NaN (cannot index with vector containing NA/NaN values)`.
Взглянем на распределение частот переменной *tariff_id* после укрупнения категорий.

In[32]:
```
# выводим частоты категорий для переменной tariff_id
print(data['tariff_id'].value_counts(dropna=False))
```

Out[32]:
```
1_1     69355
1_6     39117
1_32    15537
1_4     10970
1_5      7497
1_9      5538
1_43     3930
1_3      3339
1_16     3232
1_0      2245
1_44     2228
1_19     2102
1_2      1306
1_7      1007
1_17      717
1_21      579
1_94      414
1_22      376
1_23      370
1_91      317
1_24      303
1_41      132
1_99       79
1_25       56
Name: tariff_id, dtype: int64
```

Выведем частоты категорий по переменной *living_region*. Поскольку категорий очень много, выведем лишь последние 10 категорий.

In[33]:
```
# выводим частоты для последних 10 категорий
# переменной living_region
print(data['living_region'].value_counts(dropna=False).tail(10))
```

```
Out[33]:
ЕВРЕЙСКАЯ      203
nan            192
НЕНЕЦКИЙ       172
МАГАДАНСКАЯ    159
ДАГЕСТАН        69
АЛТАЙ           54
ЧУКОТСКИЙ       32
ЧЕЧЕНСКАЯ       31
ИНГУШЕТИЯ       19
ЧИТИНСКАЯ       17
Name: living_region, dtype: int64
```

Все категории переменной *living_region* с частотой <= 50 наблюдений (категории ЧУКОТСКИЙ, ЧЕЧЕНСКАЯ, ИНГУШЕТИЯ, ЧИТИНСКАЯ) объединим в отдельную категорию OTHER.
Сначала создаем серию, у которой значениями будут частоты категорий переменной *living_region*.

```
In[34]:
# создаем серию, у которой значениями будут частоты
# категорий переменной living_region
region_series = data['living_region'].value_counts()
```

Теперь нам надо определится с пороговой относительной частотой. В данном случае мы хотим объединить категории с частотой 50 наблюдений и меньше в категорию OTHER. Мы делим 50 наблюдений на общее количество наблюдений в наборе данных (170746 наблюдений), умножаем на 100 и получаем пороговую относительную частоту 0,029. С помощью программного кода, приведенного ниже, мы делим частоту каждой категории на общее количество наблюдений, умножаем на 100, получаем относительную частоту и сравниваем ее с пороговым значением 0,029. Если относительная частота категории меньше 0,029, возвращаем значение TRUE, если больше, то возвращаем значение FALSE.

```
In[35]:
# создаем булеву маску, если частота категории меньше 0.029,
# будет возвращено значение TRUE, в противном случае
# будет возвращено значение FALSE
mask = (region_series/region_series.sum() * 100).lt(0.029)

# выводим последние 10 категорий
mask.tail(10)
```

```
Out[35]:
ЕВРЕЙСКАЯ      False
nan            False
НЕНЕЦКИЙ       False
МАГАДАНСКАЯ    False
ДАГЕСТАН       False
АЛТАЙ          False
ЧУКОТСКИЙ       True
ЧЕЧЕНСКАЯ       True
ИНГУШЕТИЯ       True
ЧИТИНСКАЯ       True
Name: living_region, dtype: bool
```

Теперь с помощью функции `np.where()` мы все категории, по которым возвращено значение TRUE, заменим на категорию OTHER, в противном случае оставим категории неизменными.

```
In[36]:
# с помощью функции np.where мы все категории, по которым возвращено
# значение TRUE, заменим на категорию OTHER, в противном случае
# оставим категории неизменными
data['living_region'] = np.where(data['living_region'].isin(region_series[mask].index),
                                 'OTHER', data['living_region'])
```

Взглянем на распределение частот переменной *living_region* после укрупнения категорий.

```
In[37]:
# выводим частоты для последних 10 категорий
# переменной living_region
print(data['living_region'].value_counts(dropna=False).tail(10))

Out[37]:
КАМЧАТСКИЙ               412
СЕВЕРНАЯОСЕТИЯ-АЛАНИЯ    379
КАЛМЫКИЯ                 305
ЕВРЕЙСКАЯ                203
nan                      192
НЕНЕЦКИЙ                 172
МАГАДАНСКАЯ              159
OTHER                     99
ДАГЕСТАН                  69
АЛТАЙ                     54
Name: living_region, dtype: int64
```

Проблема появления новых категорий в новых данных

Существует еще проблема появления новых категорий в новых данных. Например, мы разработали и внедрили скоринговую модель. К моменту внедрения модели маркетинговая или кредитная политика банка поменялась, и у нас в переменной *Сфера занятости* появилась новая категория Няни, воспитательницы. В банках часто применяется консервативный подход: новая категория приравнивается к категории, демонстрирующей наибольший уровень риска, потому что мы ничего не знаем об этой категории клиентов и их возможном кредитном статусе.
В соревнованиях новую категорию можно приравнять к самой часто встречающейся категории или приравнять к пропуску и обрабатывать так, как было условлено обрабатывать пропуски.
Допустим, у нас в обучающих данных есть переменная *pay*. У нее есть редкая категория CD.

```
CC      2561
CH       977
Auto     889
CD         2
Name: pay, dtype: int64
```

Заменяем редкую категорию модой.

```
# заменяем редкую категорию модой
data.at[data['pay'] == 'CD', 'pay'] = 'CC'
data['pay'].value_counts(dropna=False)
```

В функцию предобработки, которую мы будем применять к новым данным, добавим программный код, заменяющий все новые категории модой.

```
…
# все новые категории переменной pay заменяем модой
replace_new_values = lambda s: 'CC' if s not in ['CC', 'Auto', 'CH'] else s
df['pay'] = df['pay'].map(replace_new_values)
…
```

Разбиение набора данных на обучающую и контрольную

В прогнозном моделировании нам важно построить модель на обучающих данных, а затем получить точные прогнозы для новых, еще не встречавшихся данных, состоящих из тех же самых предикторов, что и использованный нами обучающий набор. Если модель может выдавать точные прогнозы на ранее не встречавшихся данных, можно сказать, что модель обладает способностью *обобщать* результат на новые данные. Нам требуется построить модель с максимальной *обобщающей способностью* (*generalization*). Для задач классификации в качестве метрики обобщающей способности обычно используется правильность – количество верно классифицированных наблюдений от общего количества наблюдений или площадь под ROC-кривой (Area Under Curve - AUC), для задач регрессии – среднеквадратическая ошибка, R-квадрат.

Обычно цель специалиста по машинному обучению сводится к тому, чтобы модель давала точные прогнозы на обучающем наборе. Если обучающий набор и новые данные имеют много общего между собой, можно ожидать, что модель будет точно прогнозировать новые данные.

Однако в ряде случаев на новых данных модель работает существенно хуже. Почему так происходит?

Проблема заключается в том, что на этапе подготовки данных часто отсутствует априорная информация о полезности тех или иных предикторов. Избыточное включение предикторов, не несущих новой информации, ведет к тому, что модель становится слишком сложной. Она слишком точно подстраивается под особенности обучающего набора, улавливает не только фактические взаимосвязи, но и случайные возмущения обучающих данных. По сути такая модель восстанавливает не только искомую зависимость, но и выполняет подгонку конкретных наблюдений. В итоге мы получаем модель, которая идеально работает на обучающем наборе, но плохо обобщает результат на новые данные, поскольку описывает случайный шум в данных, не имеющий ничего общего с истинной формой связи между зависимой переменной и предикторами. Такую ситуацию называют *переобучением* (*overfitting*). С другой стороны, включение недостаточного числа полезных

признаков, наоборот, приводит к тому, что модель не может в достаточной мере уловить фактические зависимости, и качество модели даже на обучающей выборке остается довольно низким. Такую ситуацию называют *недообучением* (*underfitting*).
Для борьбы с описанными ситуациями необходимо настраивать сложность модели и проверять обобщающую способность, используя отложенную выборку. Существует оптимальная точка, которая позволяет получить наилучшую обобщающую способность. Собственно это и есть модель, которую нам нужно найти.

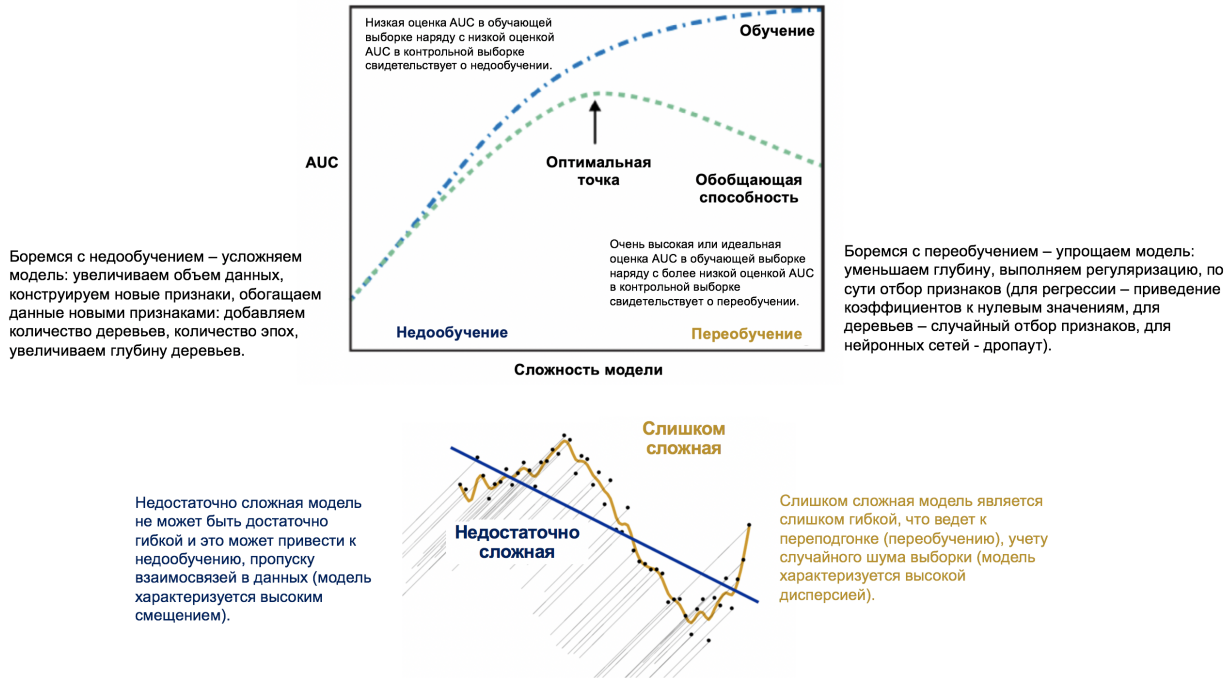


Рис. 4 Компромисс между сложностью модели и правильностью на обучающей и контрольной выборках

Ошибка прогноза любой модели вида $y = f(\vec{x}) + \epsilon$ складывается из смещения, дисперсии и неустранимой ошибки (шума).

$$\begin{aligned}
\mathrm{Err}\left(\vec{x}\right) &= \mathbb{E}\left[\left(y - \hat{f}\left(\vec{x}\right)\right)^2\right] \\
&= \sigma^2 + f^2 + \mathrm{Var}\left(\hat{f}\right) + \mathbb{E}\left[\hat{f}\right]^2 - 2f\mathbb{E}\left[\hat{f}\right] \\
&= \left(f - \mathbb{E}\left[\hat{f}\right]\right)^2 + \mathrm{Var}\left(\hat{f}\right) + \sigma^2 \\
&= \mathrm{Bias}\left(\hat{f}\right)^2 + \mathrm{Var}\left(\hat{f}\right) + \sigma^2
\end{aligned}$$

Смещение (bias) – это отклонение среднего ответа обученного алгоритма от ответа идеального алгоритма. Дисперсия (variance) – это разброс ответов обученных алгоритмов относительно среднего ответа.
Смещение показывает, насколько хорошо с помощью данных и выбранного метода можно приблизить оптимальный алгоритм. Дисперсия показывает, насколько сильно может изменяться ответ обученного алгоритма в зависимости от выборки — иными словами, она характеризует чувствительность метода обучения к изменениям в обучающей выборке. Например, одинокое глубокое дерево решений имеет низкое смещение (мы можем сделать листья настолько мелкими, что они будут содержать по одному наблюдению и мы получим однозначный прогноз) и высокую дисперсию (пермутировав несколько наблюдений, мы можем получить совершенно другое дерево).
Выбор сложности модели – это компромисс между смещением и дисперсией. Недостаточно сложная модель не может быть достаточно гибкой и это может привести к недообучению, пропуску взаимосвязей в данных (модель характеризуется высоким смещением). Неопытный исследователь может предположить, что наиболее сложная модель должна всегда лучше предсказывать, но это не так. Слишком сложная модель является слишком гибкой, что ведет к переобучению, учету случайного шума выборки (модель характеризуется высокой дисперсией). У моделей с низким смещением высока дисперсия и наоборот.
В современных методах машинного обучения мы чаще всего регулируем сложность через отбор признаков: обращение коэффициентов предикторов точно в ноль с помощью регуляризации в регрессионных моделях, случайный отбор признаков и прунинг в случайном лесе и градиентном бустинге, дропаут в нейронных сетях.
Наиболее распространенными методами проверки являются: однократное случайное разбиение набора данных на обучающую и контрольную выборки, однократное случайное разбиение набора данных на обучающую, проверочную/валидационную и контрольную/тестовую выборки, различные варианты перекрестной проверки, обычно также с резервированием тестовой выборки для итоговой оценки качества (кросс-валидация).
Мы применим случайное разбиение на обучающую и контрольную выборки. При таком способе проверки модель строится на обучающей выборке, а ее качество проверяется на контрольной выборке.

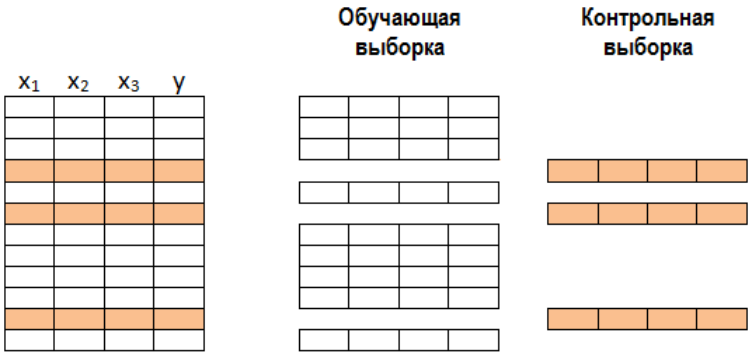


Рис. 5 Схема работы метода разделения выборки на обучающую и контрольную

Очевидно, что описанный подход является корректным только в том случае, если на основании оценки качества на контрольной выборке не производится выбор лучшей модели из множества альтернатив (т.е. не осуществляется настройка параметров). В противном случае можно получить завышенные оценки качества, т.е. может произойти косвенное переобучение на контрольную выборку.

Давайте с помощью метода `.sample()` случайным образом отберем 70% наблюдений из датафрейма `data` в обучающий датафрейм `train`. В исходном датафрейме `data` оставляем только те наблюдения, индексные метки которых отличаются от индексных меток наблюдений, попавших в обучающий датафрейм `train`, и записываем контрольный датафрейм `test`. В итоге 30% наблюдений сформируют контрольную выборку. Для получения воспроизводимых результатов разбиения с помощью параметра `random_state` задаем стартовое значение генератора случайных чисел.

```
In[38]:
# с помощью метода .sample из исходного
# датафрейма data случайно отбираем
# 70% наблюдений в обучающий датафрейм train
train = data.sample(frac=0.7,random_state=200)

# в исходном датафрейме data оставляем только те
# наблюдения, индексные метки которых отличаются
# от индекса наблюдений, попавших в обучающий датафрейм
# train, и записываем контрольный датафрейм test
test = data.drop(train.index)
```

Импутация пропусков

Выполнив разбиение набора данных на обучающую и контрольную выборки, можно приступать к импутации пропусков.

Выделяют 3 типа возникновения пропусков: MCAR, MAR, MNAR.

MCAR («совершенно случайно пропущенные» – Missing Completely At Random) – тип возникновения пропусков, при котором вероятность пропуска для каждого наблюдения набора одинакова. Вероятность пропуска значения для переменной X не связана ни со значением самой переменной X, ни со значениями других переменных в наборе данных.

Например, переменная *Доход* подчиняется условию MCAR, если респонденты, которые не сообщают о своем доходе, имеют в среднем такой же размер дохода, что и респонденты, которые указывают свой доход.

MAR («случайно пропущенные» – Missing At Random) — тип возникновения пропусков, когда данные пропущены не случайно, а ввиду некоторых закономерностей. Вероятность пропуска значения для переменной X может быть объяснена другими имеющимися переменными, не содержащими пропуски. Например, переменная *Доход* подчиняется условию MAR, если вероятность пропуска данных в переменной *Доход* зависит от наблюдаемой переменной, например от переменной *Образование.* Например, респонденты с низким уровнем образования могут иметь большее количество пропущенных значений дохода. Необходимо проанализировать взаимосвязь между переменной *Доход* и переменной *Образование*.

MNAR («не случайно пропущенные» – Missing Not At Random) — тип пропущенных данных, когда пропуск значения не является совершенно случайным и не может быть полностью объяснен другими переменными в наборе. Пропущенные значения остаются зависимыми от неизвестных нам факторов, необходимо провести дополнительные исследования. Здесь можно привести вышеописанный случай с пропусками в переменной *Доход*, но только теперь переменная *Образование* у нас отсутствует.

Давайте выведем сводку о количестве пропусков по каждой переменной в обучающей и контрольной выборках, воспользовавшись цепочкой методов `.isnull()` и `.sum()`:

In[39]:
```
# выводим информацию о количестве пропусков
# по каждой переменной в обучающей выборке
train.isnull().sum()
```

Out[39]:
```
gender                    0
age                       1
marital_status            1
job_position              0
credit_sum                1
credit_month              0
tariff_id                 0
score_shk                 4
education                 2
living_region             0
monthly_income            2
credit_count           6477
overdue_credit_count   6477
open_account_flg          0
dtype: int64
```

In[40]:
```
# выводим информацию о количестве пропусков
# по каждой переменной в контрольной выборке
test.isnull().sum()
```

Out[40]:
```
gender                    0
age                       2
marital_status            2
job_position              0
credit_sum                1
credit_month              0
tariff_id                 0
score_shk                 3
education                 3
living_region             0
monthly_income            3
credit_count           2753
overdue_credit_count   2753
open_account_flg          0
dtype: int64
```

Видно, что пропуски есть не только в количественных, но и в категориальных переменных. И здесь возникает вопрос, как осуществлять импутацию пропущенных значений.

Пропуски в количественных переменных заменяются значениями вычисленных статистик, обычно используется среднее или медиана. В случае данных, имеющих асимметричное распределение, предпочитают использовать медиану, а не среднее, так как на нее не влияет небольшое число наблюдений с очень большими или очень маленькими значениями.

Обратите внимание, что импутацию средним, медианой и прочими статистиками необходимо выполнять после разбиения набора данных на обучающую и контрольную выборки. Если выполнить импутацию на всем наборе, а потом разбить его на обучающую и контрольную выборки, получится, что при вычислении статистик для импутации использовались все наблюдения набора, часть из которых потом у нас вошла в контрольную выборку (по сути выборку новых данных). Поэтому получается, что статистики для импутации, которые мы получили на всем наборе, пришли к нам частично из «будущего» (из новой, контрольной выборки, которой по факту еще нет). Однако мы должны смоделировать наиболее близкую к реальности ситуацию, когда у нас есть только обучающая выборка, а никаких новых данных еще нет. Статистики, вычисленные на обучающей выборке, применяются для импутации пропусков как в обучающей, так и в контрольной выборках.

Помимо импутации средним или медианой пропуски можно заменить значениями-константами. Для древовидных алгоритмов эффективным может быть импутация значением, лежащим вне диапазона имеющихся данных. Например, пропуски признака можно закодировать большим отрицательным значением (-999). В этом случае в дереве можно будет выбрать такое разбиение по этому признаку, что все наблюдения с известными значениями пойдут в левый узел, а все наблюдения с пропусками — в правый.

Выполнить импутацию константами и создать индикаторы пропусков можно (и нужно, чтобы не повторять одни и те же операции для двух наборов данных) до разбиения на обучение и контроль, потому что в рамках этой операции мы не делаем вычислений, охватывающих все наблюдения исходного набора.
Бинарные переменные, у которых есть пропуски, можно превратить в категориальные с тремя категориями, где первую категорию можно закодировать как -1, вторую категорию – как 1, а пропуски – как 0.

Количественная переменная	Бинарная переменная
NA	1
6	0
5	1
8	NA
0	0
NA	0
9	1
3	NA
5	0

Количественная переменная				Бинарная переменная
медиана	среднее	-999	индикатор пропусков	
5	5.14	-999	1	1
6	6	6	0	-1
5	5	5	0	1
8	8	8	0	0
0	0	0	0	-1
5	5.14	-999	1	-1
9	9	9	0	1
3	3	3	0	0
5	5	5	0	-1

данные способы импутации можно применить только после разбиения на обучение и контроль

данные способы импутации можно применять как до, так и после разбиения на обучение и контроль

Рис. 6 Способы импутации пропусков для количественных и бинарных переменных (импутированные значения выделены желтым фоном)

Пропуски в категориальных переменных можно заменить самой часто встречающейся категорией – модой. Вновь обратите внимание, что, поскольку мы используем вычисления, импутацию модой можно осуществлять только после разбиения на обучение и контроль.
Пропуски в категориальных переменных часто кодируют отдельной категорией для пропусков, а также, как и в случае с количественными переменными, создают индикаторы пропусков (кроме случаев, когда категориальная переменная преобразуется в набор дихотомических с помощью one-hot-кодирования).

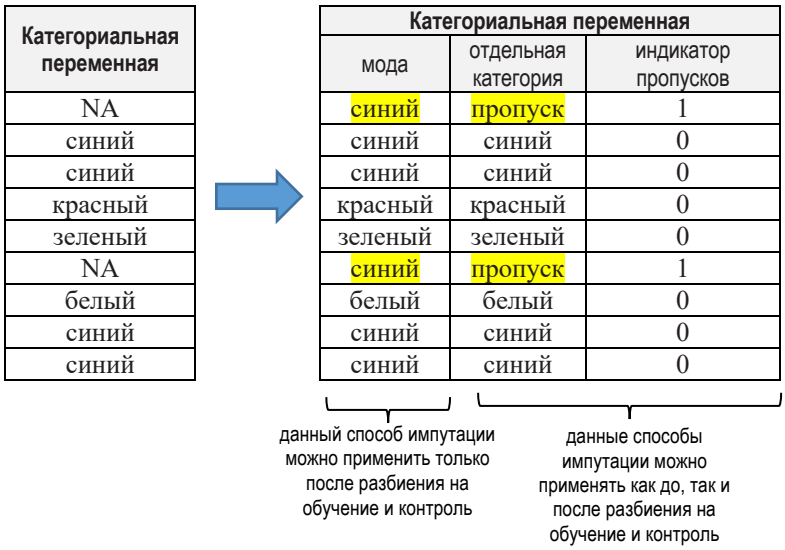


Рис. 7 Способы импутации пропусков для категориальных переменных (импутированные значения выделены желтым фоном)

Необходимо заметить, что в практике построения бизнес-моделей индикаторы пропусков носят временный характер. Чаще всего, когда необходимо строить регрессионную модель и имеются пропуски в переменной, создают индикатор пропуска, строят уравнение регрессии и смотрят, значимо ли отличается от 0 коэффициент при данном индикаторе пропусков. Если коэффициент значим, то способ импутации пропусков в этой переменной важен и будет влиять на качество модели, нужно детальнее проработать стратегию импутации пропусков, в противном случае можно воспользоваться медианой и средним (в случае количественной переменной) или модой (в случае категориальной переменной).

Давайте заменим пропуски по количественной переменной *age* в обучающей выборке медианой. Для этого необходимо воспользоваться методом `.fillna()`, задав с помощью параметра `inplace` значение `True`, чтобы выполнить импутацию на месте. Пропуски по количественной переменной *age* в контрольной выборке заменяем медианой, вычисленной на обучающей выборке.

In[41]:

```
# заполняем пропуски в переменной age медианой
train['age'].fillna(train['age'].median(), inplace=True)
test['age'].fillna(train['age'].median(), inplace=True)
```

ДОПОЛНИТЕЛЬНАЯ ИНФОРМАЦИЯ

Если бы нам понадобилось заменить пропуски средним, можно было воспользоваться следующим программным кодом:

```
train['age'].fillna(train['age'].mean(), inplace=True)
test['age'].fillna(train['age'].mean(), inplace=True)
```

Теперь выполним импутацию количественных переменных *credit_sum*, *score_shk* и *monthly_income* медианами, воспользовавшись циклом `for`:

```
In[42]:
# заполняем пропуски в переменных credit_sum, score_shk
# и monthly_income медианами
for i in ['credit_sum', 'score_shk', 'monthly_income']:
    train[i].fillna(train[i].median(), inplace=True)
    test[i].fillna(train[i].median(), inplace=True)
```

Вновь обратите внимание, что пропуски в обучающей и контрольной выборках импутированы медианами, вычисленными на обучающей выборке.

Пропуски количественных переменных *credit_count* и *overdue_credit_count* заменим значением −1:

```
In[43]:
# заполняем пропуски в переменных credit_count,
# overdue_credit_count -1
for i in ['credit_count', 'overdue_credit_count']:
    train[i].fillna(-1, inplace=True)

for i in ['credit_count', 'overdue_credit_count']:
    test[i].fillna(-1, inplace=True)
```

Теперь приступаем к импутации пропусков в категориальных переменных. Выведем частоты категорий переменной *marital_status* в обучающей и контрольной выборках с помощью метода `.value_counts()`.

```
In[44]:
# выводим частоты категорий
# для переменной marital_status
print(train['marital_status'].value_counts(dropna=False))
print(test['marital_status'].value_counts(dropna=False))

Out[44]:
MAR    65652
UNM    36581
DIV    11918
CIV     2942
WID     2428
NaN        1
Name: marital_status, dtype: int64
MAR    28302
UNM    15568
DIV     5051
CIV     1254
WID     1047
NaN        2
Name: marital_status, dtype: int64
```

Заменим пропуски модой – самой часто встречающейся категорией, в данном случае категорией `MAR`. Моду мы еще можем вычислить с помощью метода `.mode()`.

In[45]:
```
# вычисляем моду для переменной
# education
train['marital_status'].mode()
```

Out[45]:
```
0    MAR
dtype: object
```

In[46]:
```
# выполняем импутацию пропусков модой
train['marital_status'] = train['marital_status'].fillna('MAR')
test['marital_status'] = test['marital_status'].fillna('MAR')
```

Снова выведем частоты категорий переменной *marital_status*.

In[47]:
```
# выводим частоты категорий
# для переменной marital_status
print(train['marital_status'].value_counts(dropna=False))
print(test['marital_status'].value_counts(dropna=False))
```

Out[47]:
```
MAR    65653
UNM    36581
DIV    11918
CIV     2942
WID     2428
Name: marital_status, dtype: int64
MAR    28304
UNM    15568
DIV     5051
CIV     1254
WID     1047
Name: marital_status, dtype: int64
```

Выведем частоты категорий переменной *education*.

In[48]:
```
# выводим частоты категорий
# для переменной education
print(train['education'].value_counts(dropna=False))
print(test['education'].value_counts(dropna=False))
```

Out[48]:
```
SCH    61126
GRD    50928
UGR     6973
PGR      418
ACD       75
NaN        2
Name: education, dtype: int64
SCH    26411
GRD    21663
UGR     2968
PGR      147
ACD       32
NaN        3
Name: education, dtype: int64
```

Мы вновь заменим пропуски модой, но сделаем это с помощью цепочки методов `.fillna()` и `.value_counts()`:

In[49]:
```
# выполняем импутацию пропусков модой
train['education'].fillna(train['education'].value_counts().index[0], inplace=True)
test['education'].fillna(train['education'].value_counts().index[0], inplace=True)
```

Частота с индексом `0` в распределении частот, вычисляемом методом `.value_counts()`, и будет как раз модой.

Конструирование новых признаков

Теперь приступим к конструированию новых признаков (feature engineering). Конструирование новых признаков – неотъемлемый этап предварительной подготовки данных. Выделяют два типа конструирования признаков – статическое конструирование признаков (static feature engineering) и динамическое конструирование признаков (dynamic feature engineering). Статическое конструирование признаков подразумевает, что мы создаем признаки вручную. Мы создаем фактические признаки (они присутствуют в нашем наборе данных) и подаем на вход модели. Динамическое конструирование признаков подразумевает, что мы создаем признаки в ходе построения модели «на лету», эти признаки создаются только в процессе обучения модели и в наш набор данных не записываются. Каждый тип можно разделить еще на два подтипа: конструирование признаков, исходя из предметной области, опыта, бизнес-логики (domain-based feature engineering) и конструирование признаков, исходя из особенностей алгоритма (algorithm-based feature engineering).

Рис. 8 Типы конструирования признаков

Мы начнем со статического конструирования признаков, исходя из предметной области, опыта, бизнес-логики. Чаще всего в рамках такого способа конструирования признаков создают агрегаты. Агрегатная переменная – переменная, полученная в результате математических операций над двумя и более независимыми переменными.
Давайте создадим переменную *paym*, которая является отношением выданной суммы кредита к сроку кредита, то есть ежемесячной суммой кредита.

```
In[50]:
# создаем переменную paym, которая
# является отношением выданной суммы кредита
# (credit_sum) к сроку кредита (credit_month),
# то есть ежемесячной суммой кредита
train['paym'] = train['credit_sum'] / train['credit_month']
test['paym'] = test['credit_sum'] / test['credit_month']
```

Кроме того, создадим переменную *pti*, которая является отношением ежемесячной суммы кредита к ежемесячному заработку. При

вычислении переменных, которые являются отношениями количественных признаков, в случае когда происходит деление на ноль, могут быть получены бесконечные значения (значения `Inf` и `-Inf`). Их нужно заменить на конкретное значение. В данном случае заменим их на 1.

In[51]:
```
# создаем переменную pti, которая является
# отношением ежемесячной суммы кредита
# (paym) к ежемесячному заработку
# (monthly_income)
train['pti'] = train['paym'] / train['monthly_income']
test['pti'] = test['paym'] / test['monthly_income']

# заменяем бесконечные значения на 1
train['pti'].replace([np.inf, -np.inf], 1, inplace=True)
test['pti'].replace([np.inf, -np.inf], 1, inplace=True)
```

Вышеперечисленные признаки мы создали, исходя из знания предметной области. По опыту такие переменные являются довольно эффективными предикторами в кредитном скоринге.

Создание переменной, у которой значения основаны на значениях исходной переменной

Часто при конструировании признаков, исходя из предметной области, нужно проявить фантазию и вы должны понимать, что сам признак – «шкатулка с секретом», часто просто указатель, в какую сторону нужно двигаться, чтобы извлечь ценную информацию. Например, нередко приходится работать с таким предиктором, как сфера занятости, который обычно состоит из множества категорий. Сама по себе сфера занятости может быть слабо связана с кредитоспособностью, однако если мы заменим сферу занятости средней заработной платой в этой сфере, количеством банкротств предприятий в соответствующей сфере, количеством месяцев, в течение которого уволенный сотрудник, занятый в этой сфере, находит работу, мы можем получить более полезный признак. Регионы мы можем заменить средней заработной платой в регионе, размером и глубиной просрочки в регионе, криминогенностью региона. Чаще всего мы подобные операции выполняем через создание переменной, у которой значения основаны на значениях исходной переменной. Давайте применительно к нашему набору данных создадим такую переменную.

Сначала выведем уникальные значения исходной переменной, в данном случае – значения переменной *job_position*.

In[52]:
```
# выводим уникальные значения исходной переменной,
# в данном случае - значения переменной job_position
print(train['job_position'].unique())
```

Out[52]:
```
['SPC' 'UMN' 'BIS' 'DIR' 'PNA' 'ATP' 'OTHER' 'WRK' 'BIU' 'NOR' 'WOI' 'INP'
 'PNI' 'WRP']
```

Допустим, мы получили из какого-то внешнего источника информацию о средней заработной плате клиентов с различными профессиями. Тогда создаем словарь, в котором ключом будет значение исходной переменной *job_position*, а значением – значение нашей будущей переменной *avrzarplata*.

In[53]:
```
# затем создаем словарь, в котором ключом будет значение
# исходной переменной job_position, а значением - значение
# будущей переменной avrzarplata
dct = {'UMN': 51000, 'SPC': 63000, 'INP': 55000, 'DIR': 60000,
       'ATP': 46000, 'PNA': 71000, 'BIS': 86000, 'WOI': 76000,
       'NOR': 54000,'WRK': 77000, 'WRP': 75000, 'PNV': 67000,
       'BIU': 43000, 'PNI': 69000, 'HSK': 74000, 'PNS': 44000,
       'INV': 88000, 'ONB': 62000, 'OTHER': 20000}

# создаем новую переменную avrzarplata, у которой значения
# сопоставлены значениям переменной job_position
train['avrzarplata'] = train['job_position'].map(dct)
train.head()
```

Out[53]:

credit_count	overdue_credit_count	open_account_flg	tariff	paym	pti	avrzarplata
2.0	0.0	0	1.40	3357.900000	0.093275	63000
2.0	0.0	0	1.32	2351.100000	0.052247	63000
6.0	0.0	0	1.50	1666.250000	0.033325	63000
4.0	0.0	0	1.30	581.666667	0.016619	51000
2.0	0.0	0	1.60	3635.800000	0.072716	86000

Давайте удалим созданную нами переменную *avrzarplata*.

In[54]:
```
# удалим переменную avrzarplata
train.drop('avrzarplata', axis=1, inplace=True)
```

Создание бинарной переменной на основе значений количественных переменных

Большую важность имеют бинарные переменные (их еще называют переменными-флагами). Возьмем такую сферу, как рынок недвижимости. При прогнозировании стоимости квадратного метра жилья нужно учитывать, что квартиры на первом, на последнем этаже, без балкона стоят дешевле и поэтому можно создать переменные-флаги *Квартира находится на первом этаже*, *Квартира находится на последнем этаже*, *Квартира без балкона*. При этом при создании переменных *Квартира находится на первом этаже*, *Квартира находится на последнем этаже* нужно учитывать, что скорее всего для таунхаусов, 2-3 этажных шлакоблочных домов вряд ли этот индикатор будет работать, поэтому данный факт нужно оговорить специальным условием. Квартиры на втором и третьем этажах в целом привлекательны, но если на первом

этаже находится бар, ресторан или магазин, это может стать фактором, понижающим стоимость. Можно сделать индикатор, который в случае наличия магазина на первом этаже для квартир, расположенных на первых трех этажах, принимает значение 1 или значение 0 в противном случае. Квартиры, которые продаются по ипотеке, как правило, стоят дороже, можно создать индикатор *Квартира была реализована по ипотеке*.

Давайте применительно к нашему набору данных создадим бинарную переменную, основываясь на значениях одной или нескольких количественных переменных. Сначала создадим переменную *retired*, которая принимает значение `Yes`, если значение переменной *age* больше 60, и значение `No` в противном случае.

In[55]:
```
# создаем новую переменную retired, которая принимает
# значение "Yes", если значение переменной age больше 60,
# и значение "No" в противном случае
train['retired'] = np.where(train['age']>=60, 'Yes', 'No')
train.head()
```

Out[55]:

credit_count	overdue_credit_count	open_account_flg	tariff	paym	pti	retired
2.0	0.0	0	1.40	3357.900000	0.093275	No
2.0	0.0	0	1.32	2351.100000	0.052247	No
6.0	0.0	0	1.50	1666.250000	0.033325	No
4.0	0.0	0	1.30	581.666667	0.016619	Yes
2.0	0.0	0	1.60	3635.800000	0.072716	No

А теперь создадим переменную *age_inc*, которая принимает значение `Yes`, если речь идет о клиентах старше 35 лет и с суммой кредита свыше 10000, и значение `No` в противном случае.

In[56]:
```
# создаем новую переменную age_inc, которая принимает значение "Yes",
# если речь идет о клиентах старше 35 лет И с суммой кредита свыше 10000,
# и значение "No" в противном случае
train['age_inc'] = np.where((train['age'] > 35) & (train['credit_sum'] > 10000), 'Yes', 'No')
train.head()
```

Out[56]:

credit_count	overdue_credit_count	open_account_flg	tariff	paym	pti	retired	age_inc
2.0	0.0	0	1.40	3357.900000	0.093275	No	No
2.0	0.0	0	1.32	2351.100000	0.052247	No	Yes
6.0	0.0	0	1.50	1666.250000	0.033325	No	No
4.0	0.0	0	1.30	581.666667	0.016619	Yes	No
2.0	0.0	0	1.60	3635.800000	0.072716	No	Yes

Давайте удалим результаты наших «экспериментов» – переменные *retired* и *age_inc*.

In[57]:
```
# удалим переменные retired и age_inc
train.drop(['retired', 'age_inc'], axis=1, inplace=True)
```

Создание переменной, у которой каждое значение - среднее значение количественной переменной, взятое по уровню категориальной переменной

При решении задач часто повысить качество модели позволяет переменная, у которой каждое значение – это среднее значение количественной переменной, взятое по уровню категориальной переменной.

In[58]:
```
# пишем функцию, создающую переменную, у которой каждое
# значение будет средним значением количественной
# переменной (real_feature), взятым по уровню
# категориальной переменной (cat_feature)
def code_mean(df, cat_feature, real_feature):
    return (df[cat_feature].map(df.groupby(cat_feature)[real_feature].mean()))

# создаем переменную, у которой каждое значение -
# среднее значение monthly_income в
# категории переменной living_region
train['region_mean_income'] = code_mean(train, 'living_region', 'monthly_income')
train.head()
```

Out[58]:

overdue_credit_count	open_account_flg	tariff	paym	pti	region_mean_income
0.0	0	1.40	3357.900000	0.093275	38026.071197
0.0	0	1.32	2351.100000	0.052247	32589.184487
0.0	0	1.50	1666.250000	0.033325	34645.748347
0.0	0	1.30	581.666667	0.016619	33443.197674
0.0	0	1.60	3635.800000	0.072716	34010.612707

Поскольку здесь мы используем вычисления по набору данных (вычисляем среднее), то такие переменные нужно создавать после разбиения на обучающую и контрольную выборку.
Давайте удалим созданную нами переменную *region_mean_income*.

In[59]:
```
# удалим переменную region_mean_income
train.drop('region_mean_income', axis=1, inplace=True)
```

Возведение в квадрат

Часто новые переменные создают, просто за счет возведения исходных количественных переменных в степень. Например, создадим новые переменные, возведя значения количественных переменных *tariff*, *age*, *credit_sum*, *score_shk*, *monthly_income*, *credit_month* и *credit_count* в квадрат. Из-за возведения в квадрат у нас могут получиться очень

большие значения, а для представления очень больших значений в некоторых переменных используется экспоненциальная запись. Поэтому отключим ее.

In[60]:
```
# отключаем экспоненциальную запись
pd.set_option('display.float_format', lambda x: '%.3f' % x)
train.head()

# создаем новые переменные, возведя некоторые
# количественные переменные в квадрат
train['tariff_sq'] = train['tariff']**2
test['tariff_sq'] = test['tariff']**2

train['age_sq'] = train['age']**2
test['age_sq'] = test['age']**2

train['credit_sum_sq'] = train['credit_sum']**2
test['credit_sum_sq'] = test['credit_sum']**2

train['score_sq'] = train['score_shk']**2
test['score_sq'] = test['score_shk']**2

train['income_sq'] = train['monthly_income']**2
test['income_sq'] = test['monthly_income']**2

train['credit_month_sq'] = train['credit_month']**2
test['credit_month_sq'] = test['credit_month']**2

train['credit_count_sq'] = train['credit_count']**2
test['credit_count_sq'] = test['credit_count']**2
```

Теперь рассмотрим примеры конструирования признаков, исходя из особенностей алгоритма.

Регрессионные модели, метод опорных векторов и нейронные сети не умеют работать с категориальными предикторами напрямую, поэтому при работе с этими методами мы должны представить каждую категорию предиктора в виде бинарного признака, для этого выполняем дамми-кодирование.

В отличие от деревьев решений и их более продвинутых потомков – ансамблей на основе деревьев решений регрессионные модели не способны уловить сложные нелинейные взаимосвязи между предикторами и зависимой переменной, для этого мы выполняем биннинг. Кроме того, в отличие от древовидных алгоритмов регрессионные модели не могут описывать сложные взаимодействия и мы должны их создать самостоятельно для улучшения качества.

Для ансамблей на основе деревьев решений (случайный лес, градиентный бустинг) часто бывает полезно преобразовать категориальный предиктор в количественный, представив каждую категорию в виде числа. Это обусловлено проблемой множественных сравнений, актуальной для древовидных алгоритмов: в качестве предиктора расщепления чаще всего выбирается тот, по которому может быть рассмотрено наиболее количество вариантов расщепления (при этом нет гарантии, что этот предиктор является действительно полезным

с точки зрения взаимосвязи с зависимой переменной или информативности).

Например, в дереве CART, лежащем в основе случайного леса, для категориального предиктора с k категориями будет рассмотрено $2^{k-1} - 1$ вариантов разбиений[10]. Количество возможных разбиений для случайного леса возрастает лавинообразно с увеличением количества категорий. Например, при k=33 будет рассмотрено четыре миллиона возможных вариантов расщепления. Случайный лес при выборе разбиений будет склоняться в пользу категориальных признаков с большим количеством уровней. Как вариант, можно перекодировать категориальный предиктор с большим количеством уровней в количественный предиктор, представим категории в виде частот, и, таким образом, перейти к $k-1$ вариантам разбиения.

В питоновской библиотеке scikit-learn каждый уровень категориальной переменной должен быть представлен дамми-переменной. Для этого нужно выполнить дамми-кодирование. Помимо того, что дамми-кодирование может привести к огромному увеличению размерности пространства признаков, применительно к ансамблям на основе деревьев решений, способным обрабатывать категориальные переменные по принципу «как есть», оно стирает важную информацию о структуре категориального признака, по сути разбив один цельный признак на множество отдельных бинарных признаков. Бинарный признак может быть разбит только одним способом, а категориальный признак с k уровнями может быть разбит $2^{k-1} - 1$ способами. Таким образом, в полученном пространстве признаков количественные переменные получат большую важность, чем категориальные переменные, представленные бинарными признаками. Поэтому опять-таки категориальную переменную лучше обработать как количественную, не «расплавляя» ее на бинарные признаки.

Чаще всего для превращения категориального предиктора в количественный используют Label Encoding, когда категориям предиктора в лексикографическом порядке присваивают целые числа (имеет смысл для порядковых переменных), Frequency Encoding, когда каждую категорию предиктора представляем как относительную частоту наблюдений в данной категории (по сути получаем вероятность появления категории в наборе данных), Likelihood Encoding, когда каждую категорию предиктора заменяем средним значением зависимой переменной в данной категории (для бинарной классификации это будет соответствовать вероятности положительного класса зависимой переменной в данной категории предиктора), Ordinal Encoding, способ, похожий на Label Encoding с той только разницей, что мы присваиваем целые числа категориям в зависимости от порядка их появления в наборе данных.

---

[10] При условии, что категориальный предиктор обрабатывается по принципу «как есть».

Также обратите внимание, многие методы кластерного анализа не умеют работать с категориальными признаками и нужно представить категориальный признак как количественный.

Дамми-кодирование (One-hot Encoding)

Итак, мы выяснили, что некоторые методы машинного обучения не умеет напрямую обрабатывать категориальные переменные. Кроме того, библиотека scikit-learn под капотом преобразовывает датафреймы pandas в массивы NumPy, в которых каждый столбец должен быть количественным признаком. Поэтому мы должны представить категориальную переменную в виде набора количественных переменных. На сегодняшний момент наиболее распространенным способом такого представления является *дамми-кодирование* (*one-hot-encoding* или *one-out-of-N encoding*), если перевести дословно, *кодирование с одним горячим состоянием*.

Дамми-кодирование заключается в том, чтобы представить уровни категориальной переменной в виде новых признаков, которые могут принимать значения 0 и 1. Выделяют дамми-кодирование по методу неполного ранга и по методу полного ранга.

При выполнении дамми-кодирования по методу неполного ранга категориальная переменная с k уровнями будет заменена k дамми-переменными. Например, категориальная переменная *State* имеет возможные уровни `Washington`, `Arizona`, `Nevada`, `California` и `Oregon`. Для того, чтобы закодировать эти пять возможных уровней, мы создаем пять новых признаков: `Washington`, `Arizona`, `Nevada`, `California` и `Oregon`. Мы словно задаем вопросы: «данный штат является Вашингтоном?», «данный штат является Аризоной?», «данный штат является Невадой?», «данный штат является Калифорнией?», «данный штат является Орегоном?». Признак равен 1, если *State* имеет соответствующую категорию, или равен 0 в противном случае. Таким образом, для каждого наблюдения только *один* из пяти новых признаков будет равен 1. Именно поэтому данная операция называется one-hot-кодированием, дословно *кодированием с одним горячим (активным) состоянием*, и показана на рис. 9.

State
Washington
Arizona
Nevada
California
Oregon

Washington	Arizona	Nevada	California	Oregon
1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1

Рис. 9 Дамми-кодирование по методу неполного ранга

На рис. 9 мы видим, что одна переменная кодируется с помощью пяти новых признаков. Включив эту информацию в модель, мы не используем

переменную *State*, а работаем только с этими пятью признаками, принимающими значение 0 или 1.

При выполнении дамми-кодирования по методу полного ранга мы заменяем категориальную переменную с k уровнями $k - 1$ дамми-переменными. Теперь, чтобы закодировать эти пять возможных уровней, мы создаем четыре новых признака `Washington`, `Arizona`, `Nevada`, `California`, а последний уровень `Oregon` объявляем базовым (опорным): с ним будем сравнивать все остальные категории. Матрица, кодирующая дамми-переменные по методу полного ранга, показана на рис. 10.

State
Washington
Arizona
Nevada
California
Oregon

Washington	Arizona	Nevada	California
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1
0	0	0	0

Опорный уровень

Рис. 10 Дамми-кодирование по методу полного ранга

Библиотека `pandas` предлагает очень простой способ дамми-кодирования с помощью функции `pd.get_dummies()`. Параметр `drop_first` задает тип дамми-кодирования. По умолчанию для этого параметра задано значение `False` и выполняется дамми-кодирование по методу неполного ранга, в противном случае будет выполнено дамми-кодирование по методу полного ранга. Функция `pd.get_dummies()` автоматически преобразует заданную категориальную переменную в набор дамми-переменных. Давайте преобразуем переменную *marital_status* в набор дамми-переменных по методу неполного ранга.

In[61]:
```
# выполняем дамми-кодирование переменной
# marital_status по методу неполного ранга
dummies_unfull_rank_marital_status = pd.get_dummies(train['marital_status'])
# выводим первые 5 наблюдений
dummies_unfull_rank_marital_status.head()
```

Out[61]:

	CIV	DIV	MAR	UNM	WID
53397	0	0	0	1	0
143962	0	0	0	1	0
146922	0	0	1	0	0
63697	0	0	0	1	0
54503	0	0	1	0	0

In[62]:
```
# выполняем дамми-кодирование переменной
# marital_status по методу полного ранга
dummies_full_rank_marital_status = pd.get_dummies(train['marital_status'],
                                                  drop_first=True)
# выводим первые 5 наблюдений
dummies_full_rank_marital_status.head()
```

Out[62]:

	DIV	MAR	UNM	WID
53397	0	0	1	0
143962	0	0	1	0
146922	0	1	0	0
63697	0	0	1	0
54503	0	1	0	0

Кодирование контрастами (Effect Coding)

При выполнении дамми-кодирования по методу полного ранга коэффициенты при дамми-переменных выражают влияние каждой категории по сравнению с опорной. Кодирование контрастами применяется, когда нужно сравнить вклад каждой категории со средним вкладом по всем категориям. Кодирование контрастами похоже на дамми-кодирование по методу полного ранга с той только разницей, что в опорном уровне, представленном нулями, нули заменяются на -1.

State
Washington
Arizona
Nevada
California
Oregon

Washington	Arizona	Nevada	California
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1
-1	-1	-1	-1

Опорный уровень

Рис. 11 Кодирование контрастами

In[63]:

```
# выполняем кодирование контрастами
effects_marital_status = pd.get_dummies(train['marital_status'])
effects_marital_status = effects_marital_status.iloc[:,:-1]
effects_marital_status.loc[np.all(effects_marital_status == 0, axis=1)] = -1.

# выводим первые 10 наблюдений
effects_marital_status.head(10)
```

Out[63]:

	CIV	DIV	MAR	UNM
53397	0.000	0.000	0.000	1.000
143962	0.000	0.000	0.000	1.000
146922	0.000	0.000	1.000	0.000
63697	0.000	0.000	0.000	1.000
54503	0.000	0.000	1.000	0.000
115316	0.000	0.000	0.000	1.000
16354	0.000	0.000	0.000	1.000
98227	-1.000	-1.000	-1.000	-1.000
69607	0.000	0.000	1.000	0.000
99745	0.000	0.000	0.000	1.000

Присвоение категориям в лексикографическом порядке целочисленных значений, начиная с 0 (Label Encoding)

Категориям переменной в лексикографическом порядке можно присвоить целочисленные значения (начиная с 0) и в итоге категориальная переменная теряет свою категориальную природу и превращается в количественную.

Допустим, у нас есть переменная с 4 категориями A, B, C и D. Тогда мы присвоим A – 0, B – 1, C – 2, D – 3.

Переменная Class	
A	0
B	1
C	2
D	3

Переменная Class	Кодировка Label Encoding для переменной Class
B	1
A	0
C	2
A	0
A	0
D	3
D	3
B	1

Рис. 12 Пример кодировки Label Encoding

Эту процедуру можно выполнить с помощью класса `LabelEncoder`.

```
In[64]:
# импортируем класс LabelEncoder
from sklearn.preprocessing import LabelEncoder
# создаем экземпляр класса LabelEncoder
label_encoder = LabelEncoder().fit(train['job_position'])
# выполняем кодировку
train['job_position2'] = label_encoder.transform(train['job_position'])
train.head()
```

Out[64]:

credit_sum_sq	score_sq	income_sq	credit_month_sq	credit_count_sq	job_position2
1127549241.000	0.186	1296000000.000	100	4.000	9
552767121.000	0.129	2025000000.000	100	4.000	9
1599200100.000	0.376	2500000000.000	576	36.000	9
12180100.000	0.138	1225000000.000	36	16.000	10
1321904164.000	0.416	2500000000.000	100	4.000	1

Мы можем выполнить Label Encoding как до разбиения на обучающую и контрольную выборки, так и после разбиения, поскольку не делаем никаких вычислений (разумеется, лучше выполнить Label Encoding до разбиения, чтобы не писать лишний программный код).

На практике Label Encoding делают вручную и чаще всего для порядковых переменных. Для номинальных переменных кодировка Label Encoding редко бывает эффективна. Например, у нас есть переменная *Цвет автомобиля* с категориями *Синий*, *Желтый*, *Красный*,

мы можем закодировать их как 2, 0 и 1, но это не будет иметь большого смысла, ведь выполнив такую кодировку, мы предполагаем определенный порядок $0 < 1 < 2$, который здесь не выполняется. Давайте удалим созданную нами переменную *job_position2*.

In[65]:
```
# удалим переменную job_position2
train.drop('job_position2', axis=1, inplace=True)
```

Создание переменной, у которой каждое значение – частота наблюдений в категории переменных (Frequency Encoding)

Категориям переменной присваиваем абсолютные или относительные частоты.

Переменная Class

A	0,44 (4 из 9)
B	0,33 (3 из 9)
C	0,22 (2 из 9)

Переменная Class	Кодировка Frequency Encoding для переменной Class	
	относительная частота	абсолютная частота
A	0,44	4
B	0,33	3
A	0,44	4
C	0,22	2
B	0,33	3
B	0,33	3
A	0,44	4
A	0,44	4
C	0,22	2

Рис. 13 Пример кодировки Frequency Encoding

Сначала закодируем категории абсолютными частотами.

In[66]:
```
# создаем переменную region_abs_freq, у которой каждое значение - абсолютная
# частота наблюдений в категории переменной living_region
abs_freq = train['living_region'].value_counts()
train['region_abs_freq'] = train['living_region'].map(abs_freq)
test['region_abs_freq'] = test['living_region'].map(abs_freq)
train.head()
```

credit_sum_sq	score_sq	income_sq	credit_month_sq	credit_count_sq	region_abs_freq
1127549241.000	0.186	1296000000.000	100	4.000	618
552767121.000	0.129	2025000000.000	100	4.000	3236
1599200100.000	0.376	2500000000.000	576	36.000	3024
12180100.000	0.138	1225000000.000	36	16.000	344
1321904164.000	0.416	2500000000.000	100	4.000	3620

Теперь закодируем категории относительными частотами.

In[67]:
```
# создаем переменную region_rel_freq, у которой каждое значение - относительная
# частота наблюдений в категории переменной living_region
rel_freq = train['living_region'].value_counts() / len(train['living_region'])
train['region_rel_freq'] = train['living_region'].map(rel_freq)
test['region_rel_freq'] = test['living_region'].map(rel_freq)
train.head()
```

score_sq	income_sq	credit_month_sq	credit_count_sq	region_abs_freq	region_rel_freq
0.186	1296000000.000	100	4.000	618	0.005
0.129	2025000000.000	100	4.000	3236	0.027
0.376	2500000000.000	576	36.000	3024	0.025
0.138	1225000000.000	36	16.000	344	0.003
0.416	2500000000.000	100	4.000	3620	0.030

Обратите внимание, что поскольку мы используем вычисления, то кодировку частотами нужно выполнять строго после разбиения на обучающую и контрольную выборки. Частотами, вычисленными для категорий переменной в обучающей выборке, заменяем категории переменной в обучающей и контрольной выборках. Следует помнить, что в обучающем наборе могут встречаться не все возможные категории предиктора, некоторые категории могут присутствовать только в тестовом наборе или появятся только тогда, когда модель будет применяться к новым данным. Поэтому схема кодировки должна предусмотреть возможность обработки новых категорий.

Кроме того, когда мы найдем комбинацию оптимальных значений гиперпараметров на контрольной выборке или тестовой выборке и нам нужно будет построить модель с этой комбинацией оптимальных значений гиперпараметров на всей исторической выборке, мы должны заново вычислить частоты для категорий.

На практике чаще используют кодирование относительными частотами, потому что абсолютные частоты сильно зависят от размера набора данных. Кодирование частотами может приводить к коллизиям, когда разные категории встречаются одинаковое количество раз и мы должны присвоить им одинаковые частоты. В таких случаях можно попробовать добавить случайный шум, добавить константу или объединить, если исходные категории не отличаются по зависимой переменной (можно использовать тест хи-квадрат), и посмотреть, какой вариант дает наилучший результат. Для линейных моделей часто бывает полезным выполнить для полученной переменной преобразование, максимизирующее нормальность.

Давайте удалим созданные нами переменные *region_abs_freq* и *region_rel_freq*.

```
In[68]:
# удалим переменные region_abs_freq и region_rel_freq
train.drop(['region_abs_freq', 'region_rel_freq'], axis=1, inplace=True)
test.drop(['region_abs_freq', 'region_rel_freq'], axis=1, inplace=True)
```

Кодирование вероятностями зависимой переменной (Likelihood Encoding)

Участники соревнований Kaggle часто используют кодирование вероятностями зависимой переменной (Likelihood Encoding). Существует множество способов такой кодировки. Начнем с наиболее распространенного способа, который предложил в 2001 году Дэниэл Микки-Баррека в своей статье «A Preprocessing Scheme for High-Cardinality Categorical Attributes in Classification and Prediction Problems» («Схема предварительной обработки категориальных признаков с большим количеством уровней для задач классификации и прогнозирования»).

Когда зависимая переменная Y является бинарной, мы можем сопоставить каждое индивидуальное значение X_i категориальной переменной X скаляру S_i, представляющему собой оценку вероятности $Y = 1$ для $X = X_i$:

$$X_i \to S_i \cong P(Y|X = X_i) \qquad (1)$$

Проще говоря, мы можем представить каждую категорию предиктора как вероятность появления значения 1 зависимой переменной в этой категории (отсюда и название «кодирование вероятностями» или Likelihood Encoding). Обратите внимание, что поскольку мы используем вычисления, то кодирование вероятностями нужно выполнять строго после разбиения на обучающую и контрольную выборки. Значениями, вычисленными для категорий переменной в обучающей выборке, заменяем категории переменной в обучающей и контрольной выборках. Как и в случае кодировки частотами, при кодировке вероятностями следует позаботиться о том, чтобы схема кодировки предусматривала возможность обработки новых категорий.

Чтобы вычислить вероятность для каждого уровня, мы по каждой категории смотрим количество наблюдений, в которых зависимая переменная приняла значение 1, делим это количество на общее количество наблюдений в данной категории.

$$S_i = \frac{n_{iY}}{n_i} \qquad (2)$$

где:

n_{iY} – количество наблюдений i-той категории, в которых зависимая переменная приняла значение 1;

n_i – общее количество наблюдений i-той категории.

Например, у нас есть предиктор *Class*. Он имеет категории A, B и C. Категория A встречается в 4 наблюдениях. В 3 из 4 наблюдений зависимая переменная принимает значение 1. 3 делим на 4, получаем 0,75. Вероятность для категории A будет равна 0,75. По сути мы заменяем категорию предиктора средним значением зависимой переменной в этой категории, отсюда второе название данной кодировки – кодирование обычным средним значением зависимой переменной (Mean Target Encoding). Пример такой кодировки показан на рис. 14.

Переменная Class	
A	0,75 (3 из 4)
B	0,66 (2 из 3)
C	1,00 (2 из 2)

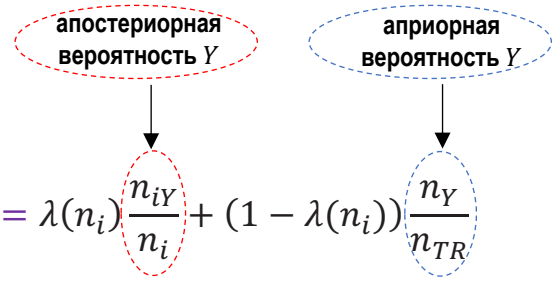
Переменная Class	Зависимая переменная	Кодировка Simple Mean Target Encoding для переменной Class
A	1	0,75
A	0	0,75
A	1	0,75
A	1	0,75
B	1	0,66
B	1	0,66
B	0	0,66
C	1	1,00
C	1	1,00

Рис. 14 Пример кодировки обычным средним значением зависимой переменной

К сожалению, часто категориальные переменные имеют редкие уровни и, следовательно, оценка $P(Y|X = X_i)$ в уравнении 2 будет довольно ненадежной. Допустим, некоторый уровень встретился всего несколько раз, и в соответствующих наблюдениях зависимая переменная принимает значение 1. Тогда среднее значение зависимой переменной тоже будет единицей. При этом на тестовом наборе может возникнуть совсем другая ситуация. В нашем примере, изображенном на рис. 8, такой проблемной категорией является категория C, встречается она меньше остальных категорий, но при этом вероятность ее появления равна 1.

Кодировка средним значением зависимой переменной, сглаженным через сигмоидальную функцию

Чтобы сгладить эффект редких уровней, оценка вероятности S_i для каждого уровня рассчитывается как смесь двух вероятностей: апостериорной вероятности Y для $X = X_i$, вычисляемой с помощью уравнения 2, и априорной вероятности Y (базовой вероятности, получаемой по всему обучающему набору). Эти две вероятности «смешивают» с помощью весового коэффициента, который является функцией от размера категории предиктора. Вычисляемое среднее значение называют средним значением, сглаженным через сигмоидальную функцию.



$$S_i = \lambda(n_i)\frac{n_{iY}}{n_i} + (1 - \lambda(n_i))\frac{n_Y}{n_{TR}} \qquad (3)$$

где:

n_{iY} – количество наблюдений i-той категории предиктора, в которых зависимая переменная приняла значение 1;

n_i – общее количество наблюдений i-той категории предиктора;

n_Y – количество наблюдений обучающего набора данных, в которых зависимая переменная приняла значение 1;

n_{TR} – общее количество наблюдений обучающего набора данных;

$\lambda(n_i)$ – весовой коэффициент, представляет собой монотонно возрастающую функцию от n_i, ограниченную диапазоном значений от 0 до 1.

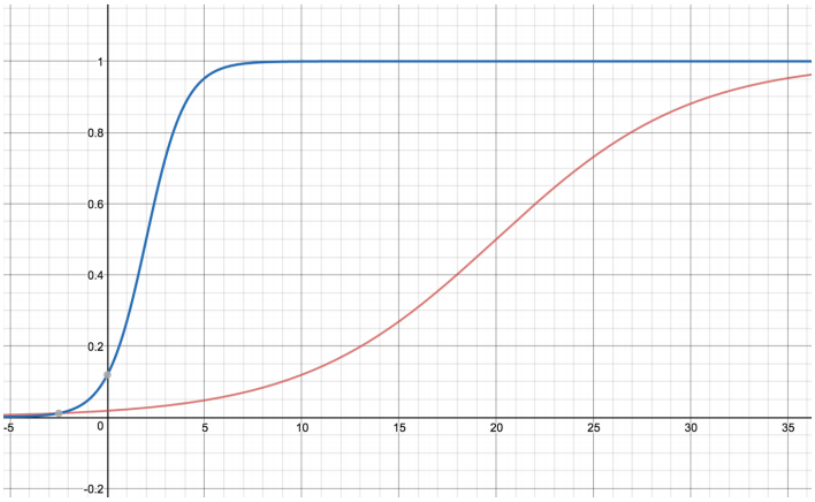
Эту формулу часто сводят к более простой формуле:

$$S_{level} = \lambda(n_{level}) \times mean(level) + \big(1 - \lambda(n_{level})\big) \times mean(dataset) \qquad (3a)$$

Обоснование формулы (3) сводится к тому, что если размер категории является большим ($\lambda \cong 1$), мы должны больше доверять оценке апостериорной вероятности. Закодированное значение уровня будет ближе к среднему значению зависимой переменной в данном уровне. Однако, если размер категории ($\lambda \cong 0$) мал, мы должны ориентироваться на априорную вероятность. Закодированное значение уровня будет ближе к среднему значению зависимой переменной в обучающем наборе. Обычно весовой коэффициент $\lambda(n)$ задается как функция с одним или несколькими настраиваемыми параметрами и может быть оптимизирован на основе характеристик данных. Например, коэффициент $\lambda(n)$ может быть определен как:

$$\lambda(n) = \frac{1}{1 + e^{-\frac{(n-k)}{f}}} \qquad (4)$$

где $\lambda(n)$ является сигмоидальной функцией, которая предполагает значение 0,5 для $n = k$. Примеры таких функций приведены на рис. 15.



$$\lambda(n) = \frac{1}{1 + e^{-\frac{(n-k)}{f}}}$$

Рис. 15 Примеры сигмоидальных функций $\lambda(n)$

Здесь мы настраиваем три параметра: n – частоту категории (например, частота для категории A будет равна 4, для категории B – 3, для категории C – 2), k – половину минимально допустимого размера категории, при которой мы полностью «доверяем» апостериорной вероятности, и f – угол наклона сигмоиды (определяет скорость перехода от апостериорной вероятности к априорной).

Обратите внимание, что сглаживание мы применяем только для обучающей выборки, для контрольной выборки мы используем обычные средние значения зависимой переменной в соответствующих категориях предиктора. Вспомним, что вычисление сглаженных средних значений зависимой переменной – это тоже модель, которую мы строим на обучающей выборке и проверяем на контрольной выборке.

Кроме того, когда мы найдем комбинацию оптимальных значений гиперпараметров на контрольной выборке или тестовой выборке и нам нужно будет построить модель с этой комбинацией оптимальных значений гиперпараметров на всей исторической выборке, мы должны заново вычислить сглаженные средние значения зависимой переменной для категорий предиктора на всей исторической выборке.

Для категорий предиктора в новых данных мы используем обычные средние значения зависимой переменной в соответствующих категориях предиктора, вычисленные на всей исторической выборке.

Давайте выполним кодирование для наших категорий, при этом сохраним значение параметра f неизменным (пусть оно будет равно 0,25), значение параметра k будем увеличивать с 1 до 3.

$$\lambda = \frac{1}{1 + exp^{-\frac{(n-1)}{0,25}}}$$

	n	$mean(level)$	$mean(dataset)$	λ	$1-\lambda$	Результат кодировки
A	4	0,75	0,77	0,99	0,01	0,99*0,75 + 0,01*0,77 = 0,7502
B	3	0,66	0,77	0,99	0,01	0,99*0,66 + 0,01*0,77 = 0,6611
C	2	1,00	0,77	0,98	0,02	0,98*1,0 + 0,02*0,77 = 0,9954

$$\lambda = \frac{1}{1 + exp^{-\frac{(n-2)}{0,25}}}$$

	n	$mean(level)$	$mean(dataset)$	λ	$1-\lambda$	Результат кодировки
A	4	0,75	0,77	0,99	0,01	0,99*0,75 + 0,01*0,77 = 0,7502
B	3	0,66	0,77	0,98	0,02	0,98*0,66 + 0,02*0,77 = 0,6622
C	2	1,00	0,77	0,5	0,5	0,5*1.0 + 0,5*0,77 = 0,885

$$\lambda = \frac{1}{1 + exp^{-\frac{(n-3)}{0,25}}}$$

	n	$mean(level)$	$mean(dataset)$	λ	$1-\lambda$	Результат кодировки
A	4	0,75	0,77	0,98	0,02	0,98*0,75 + 0,02*0,77 = 0,7504
B	3	0,66	0,77	0,5	0,5	0,5*0,66 + 0,5*0,77 = 0,715
C	2	1,00	0,77	0,017	0,983	0,017*1,0 + 0,983*0,77 = 0,773

Рис. 16 Пример кодировки средним значением зависимой переменной, сглаженным через сигмоидальную функцию

Обратите внимание, что при первом варианте кодировки для категории C вес априорной вероятности резко падает и мы больше ориентируемся на апостериорную вероятность, т.е. на среднее значение зависимой переменной в данной категории предиктора. Как результат, закодированное значение предиктора будет находиться ближе к среднему значению зависимой переменной в данной категории предиктора. При втором варианте вес апостериорной вероятности и вес априорной вероятности будет одинаковым. Здесь закодированное значение предиктора представляет собой среднее значение двух средних – среднего значения зависимой переменной в данной категории предиктора и среднего значения зависимой переменной в обучающем наборе. При третьем варианте кодировки вес апостериорной вероятности резко падает, и мы больше ориентируемся на априорную вероятность, т.е. на среднее значение зависимой переменной в обучающем наборе. Закодированное значение предиктора будет находиться ближе к среднему значению зависимой переменной в обучающем наборе.

Часто бывает, что категориальная переменная содержит пропущенные значения. В таком случае можно ввести дополнительное значение X – *нулевое* значение и оценивать вероятность Y для $X = X_0$ с помощью стандартной формулы:

$$S_0 = \lambda(n_0)\frac{n_{0Y}}{n_0} + (1 - \lambda(n_0))\frac{n_Y}{n_{TR}} \qquad (5)$$

Преимущество данного подхода заключается в том, что если пропуск действительно влияет на зависимую переменную, то оценка учтет это. Если же наличие пропусков не влияет на значение зависимой переменной, то оценка будет стремится к априорной вероятности зависимой переменной, и это будет соответствовать «нейтральности» пропущенного значения.

Кодировка средним значением зависимой переменной, сглаженным через параметр регуляризации

Теперь рассмотрим более простую схему кодировки средним значением зависимой переменной, сглаженным через параметр регуляризации. Она выполняется по формуле:

$$S_{level} = \frac{mean(level) \times nrows + mean(dataset) \times alpha}{nrows \times alpha} \qquad (6)$$

где:
$nrows$ – количество наблюдений, относящихся к категории предиктора;
$alpha$ – параметр регуляризации.

Кодировка простым средним значением зависимой переменной по схеме leave-one-out

Еще один способ – кодирование простым средним значением зависимой переменной с исключением по одному (Leave-One-Out Mean Target Encoding). По каждому наблюдению вычисляем среднее значение зависимой переменной, используя все оставшиеся наблюдения данной категории, кроме него самого.

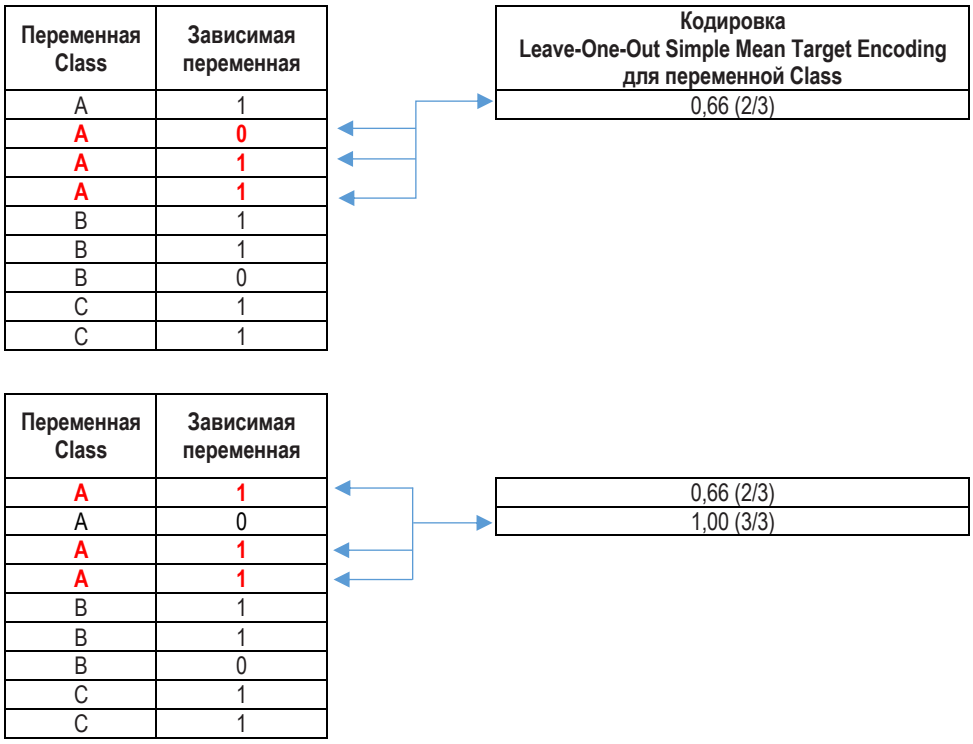


Рис. 17 Пример кодировки простым средним значением зависимой переменной по схеме leave-one-out

Кодировка простым средним значением зависимой переменной по схеме K-fold

Кроме того, применяют кодирование простым средним значением зависимой переменной с помощью *k*-блочной перекрестной проверки (K-fold Mean Target Encoding). Например, при 5-блочной перекрестной проверке мы разбиваем набор на 5 блоков: блоки 1, 2, 3, 4, 5. Вычисляем среднее значение зависимой переменной в каждой категории предиктора по каждому блоку перекрестной проверки. Например, вычисляем среднее значение зависимой переменной в категории B в блоке 1. Для вычислений используем только наблюдения категории B в блоках с 2 по 5.

Переменная Class	Зависимая переменная	Блок	Кодировка K-fold Simple Mean Target Encoding для переменной Class
A	1	5	
A	1	3	
B	0	2	
B	0	1	0,4 (2 из 5)
A	1	1	
B	0	2	
B	0	1	0,4 (2 из 5)
A	1	4	
A	0	4	
C	1	4	
C	0	4	
B	1	4	
A	1	2	
A	0	5	
B	1	4	
A	1	5	
C	0	3	
A	1	4	
B	0	2	
B	1	1	0,4 (2 из 5)

Рис. 18 Пример кодировки простым средним значением зависимой переменной по схеме K-fold

Кодировка средним значением зависимой переменной, сглаженным через сигмоидальную функцию, по схеме K-fold

В соревнованиях Kaggle чаще всего применяют кодирование средним значением зависимой переменной, сглаженным через сигмоидальную функцию, по схеме K-fold. Допустим, у нас есть переменная *Class*, которая имеет категории A, B, C, и в крайнем правом столбце приведены результаты кодировки средним значением зависимой переменной, сглаженным через сигмоидальную функцию, по схеме K-fold (рис. 19).

	Переменная Class	Номер контрольного блока	Зависимая переменная	Кодировка Kfold Blended Mean Target Encoding для переменной Class
1	A	0	1	**0,568**
2	A	0	1	**0,568**
3	A	0	1	**0,568**
4	A	0	1	**0,568**
5	A	0	0	**0,568**
6	A	0	1	**0,568**
7	**A**	**1**	**0**	0,616
8	**A**	**1**	**1**	0,616
9	**A**	**2**	**1**	0,596
10	B	0	0	0,507
11	B	0	1	0,507
12	B	1	0	0,517
13	B	1	0	0,517
14	B	1	1	0,517
15	B	1	1	0,517
16	B	1	0	0,517
17	B	2	0	0,524
18	C	0	0	0,543
19	C	1	0	0,543
20	C	2	1	0,472

Рис. 19 Пример кодировки средним значением зависимой переменной, сглаженным через сигмоидальную функцию, по схеме K-fold

Выясним, как было получено сглаженное среднее значение зависимой переменной для категории A предиктора *Class* в блоке 0 (выделены синим жирным шрифтом). Сначала вычисляем среднее значение зависимой переменной для категории A в блоке 0. Для вычисления этого среднего значения используются лишь наблюдения в категории A в обучающих блоках 1 и 2 (выделены красным жирным шрифтом), 2 / 3 = 0,67. Затем вычисляем среднее значение зависимой переменной в нашем наборе данных, 11 / 20 = 0,55. Теперь вычисляем весовой коэффициент $\lambda(n_i)$. Задаем k равным 20, а f равным 10. Весовой коэффициент равен 1 / (1 + exp(-((3 - 20)/10))) = 0,154. Наконец, вычисляем сглаженное среднее, 0,154 * 0,67 + (1 - 0,154) * 0,55 = 0,10 + 0,47 = 0,57.

Разберем несколько программных реализаций вышеописанной кодировки. Начнем с самой простой, в которой мы вычисляем среднее значение зависимой переменной в каждой категории в каждом блоке перекрестной проверки, используя данные вне этого блока, при этом сглаживание через сигмоидальную функцию не применяется.

Воспользуемся классом `StratifiedKFold` для применения перекрестной проверки.

In[69]:
```
# импортируем класс StratifiedKFold
from sklearn.model_selection import StratifiedKFold
# создаем экземпляр класса StratifiedKFold
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
```

Теперь определимся с переменными, для которых нужно выполнить кодирование.

In[70]:
```
# создаем список из двух признаков и зависимой переменной
cat_cols = ['living_region', 'job_position', 'open_account_flg']
# создаем обучающий массив со значениями зависимой переменной
y_train = train.loc[:, 'open_account_flg'].astype('int')
# создаем обучающий массив признаков
X_train = train[cat_cols].drop('open_account_flg', axis=1)
# создаем контрольный массив признаков
X_valid = test[cat_cols].drop('open_account_flg', axis=1)
```

Пишем функцию, которая выполняет кодирование. У нее – четыре аргумента: обучающий массив признаков, обучающий массив со значениями зависимой переменной, контрольный массив признаков, модель перекрестной проверки.

In[71]:
```
# пишем функцию, выполняющую кодирование средними
# значениями зависимой переменной
def mean_target_enc(X_train, y_train, X_valid, skf):
    # отключаем предупреждения Anaconda
    import warnings
    warnings.filterwarnings('ignore')

    # вычисляем глобальное среднее - среднее значение
    # зависимой переменной в обучающем наборе
    glob_mean = y_train.mean()

    # конкатенируем обучающий массив с признаками (задается первым аргументом)
    # и обучающий массив с метками зависимой переменной (задается вторым
    # аргументом) по оси столбцов
    X_train = pd.concat([X_train, pd.Series(y_train, name='open_account_flg')], axis=1)
    # создаем копию массива признаков, получившегося в результате конкатенации
    new_X_train = X_train.copy()

    # создаем список с именами категориальных признаков,
    # который мы будем использовать ниже в циклах for
    cat_features = X_train.columns[X_train.dtypes == 'object'].tolist()

    # для каждого категориального признака создаем столбец, каждое
    # значение которого - глобальное среднее
    for col in cat_features:
        new_X_train[col + '_mean_target'] = [glob_mean for _ in range(new_X_train.shape[0])]

    # вычисляем среднее значение зависимой переменной в категории признака
    # по каждому блоку перекрестной проверки, используя данные вне этого блока
    # например, мы используем 5-блочную перекрестную проверку и нам нужно
    # вычислить среднее значение зависимой переменной для категории A в блоке 0,
    # для вычисления этого среднего значения используются лишь наблюдения в категории A
    # в обучающих блоках 1, 2, 3, 4, если вместо категорий у нас
    # значения NaN, заменяем глобальным средним, в итоге
    # получаем новый обучающий набор
    for train_idx, valid_idx in skf.split(X_train, y_train):
        X_train_cv, X_valid_cv = X_train.iloc[train_idx, :], X_train.iloc[valid_idx, :]

        for col in cat_features:
            means = X_valid_cv[col].map(X_train_cv.groupby(col)['open_account_flg'].mean())
            X_valid_cv[col + '_mean_target'] = means.fillna(glob_mean)

        new_X_train.iloc[valid_idx] = X_valid_cv

    # удаляем из нового обучающего набора категориальные признаки и зависимую переменную
    new_X_train.drop(cat_features + ['open_account_flg'], axis=1, inplace=True)

    # создаем копию контрольного массива признаков
    new_X_valid = X_valid.copy()

    # каждую категорию категориального признака в контрольном наборе
    # заменяем средним значением зависимой переменной в этой же категории
    # признака, вычисленным на обучающем наборе, значения NaN
```

```
    # заменяем глобальным средним
    for col in cat_features:
        means = new_X_valid[col].map(X_train.groupby(col)['open_account_flg'].mean())
        new_X_valid[col + '_mean_target'] = means.fillna(glob_mean)

    # удаляем из контрольного набора категориальные признаки
    new_X_valid.drop(X_train.columns[X_train.dtypes == 'object'], axis=1, inplace=True)

    # возвращаем новые датафреймы
    return new_X_train, new_X_valid
```

Применяем написанную функцию.

In[72]:

```
# выполняем кодирование средними значениями
# зависимой переменной для переменных
# living_region и job_position
# в обучающем и контрольном наборах
train_mean_target, valid_mean_target = mean_target_enc(X_train, y_train, X_valid, skf)
```

Давайте взглянем на результаты кодировки в обучающем массиве признаков.

In[73]:

```
# взглянем на результаты кодировки в обучающем массиве признаков
train_mean_target.head()
```

Out[73]:

	living_region_mean_target	job_position_mean_target
53397	0.199	0.166
143962	0.099	0.166
146922	0.200	0.166
63697	0.172	0.188
54503	0.181	0.154

Класс `TargetEncoder` пакета `category_encoders` позволяет выполнить кодирование средними значениями зависимой переменной с использованием сглаживания через сигмоидальную функцию.
Если вы используете Anaconda, пакет `category_encoders` можно установить в Anaconda Prompt с помощью команды `conda install -c conda-forge category_encoders`.

In[74]:

```
# импортируем кодировщики из пакета category_encoders, предварительно
# установив его в Anaconda Prompt с помощью
# команды conda install -c conda-forge category_encoders
from category_encoders import *
```

Теперь создаем экземпляр класса `TargetEncoder` и строим модель. Обратите внимание на список параметров и гиперпараметров для класса `TargetEncoder`.

Параметр/гиперпараметр	Предназначение
`cols`	Задает список предикторов для кодировки.
`min_samples_leaf`	Задает частоту категории, которую надо учитывать при вычислении среднего значения зависимой переменной в категории предиктора.
`smoothing`	Задает сглаживание (баланс между апостериорной и априорной вероятностями).

In[75]:
```
# создаем экземпляр класса TargetEncoder (модель)
# для обучающего массива признаков
target_enc = TargetEncoder(cols=['living_region', 'job_position'],
                           smoothing=2,
                           min_samples_leaf=4)
```

Обратите внимание, что в машинном обучении есть четкая разница между понятиями «параметр» и «гиперпараметр». Параметры мы находим в ходе обучения модели. Например, сейчас мы будем строить модель, вычисляющую средние значения зависимой переменной для соответствующих категорий предикторов. Эти средние значения будут параметрами. В регрессионных моделях параметрами будут веса признаков (их называют регрессионными коэффициентами). В деревьях решений в качестве параметров будут выступать правила разбиения. На практике параметрами называют не только параметры модели, но и самые обычные настройки. Например, `cols` – это пример такого параметра. А вот гиперпараметры нельзя «выучить» в процессе обучения, их задают перед обучением модели. Например, наша модель, вычисляющая средние значения зависимой переменной, не может самостоятельно выяснить оптимальную частоту категории, которую надо учитывать при вычислении среднего значения зависимой переменной в категории предиктора. Поэтому `min_samples_leaf` – это гиперпараметр, который мы настраиваем на отложенной выборке. Логистическая регрессия не может самостоятельно выяснить оптимальное значение силы регуляризации `C`. Поэтому сила регуляризации – это гиперпараметр модели, который позволяет улучшить качество модели и опять-таки настраивается на отложенной выборке. Случайный лес не может самостоятельно выяснить оптимальное значение максимальной глубины `max_depth`. Поэтому `max_depth` – это тоже гиперпараметр, который мы настраиваем на отложенной выборке.

Выполняем кодирование и смотрим результаты.

In[76]:

```
# обучаем модель, т.е. создаем таблицу, в соответствии с которой
# категориям предиктора в обучающей выборке будут сопоставлены
# сглаженные средние значения зависимой переменной
target_enc.fit(X_train, y_train)

# применяем модель к обучающему массиву признаков, категории предиктора
# в обучающей выборке заменяются на сглаженные средние значения зависимой
# переменной
target_encoded_train = target_enc.transform(X_train)

# создаем экземпляр класса TargetEncoder (модель)
# для контрольной выборки
target_enc_valid = TargetEncoder(cols=['living_region', 'job_position'],
                                 smoothing=False)

# обучаем модель, т.е. создаем таблицу, в соответствии с которой
# категориям предиктора в контрольной выборке будут сопоставлены
# обычные средние значения зависимой переменной в этих категориях,
# вычисленные на обучающей выборке
target_enc_valid.fit(X_train, y_train)

# применяем модель к контрольной выборке,
# категории предиктора в контрольной выборке заменяются на обычные
# средние значения зависимой переменной в этих категориях,
# вычисленные на обучающей выборке
target_encoded_valid = target_enc_valid.transform(X_valid)
```

In[77]:

```
# взглянем на результаты кодировки в обучающем массиве признаков
target_encoded_train.head()
```

Out[77]:

	living_region	job_position
53397	0.204	0.167
143962	0.095	0.167
146922	0.201	0.167
63697	0.177	0.187
54503	0.183	0.159

Присвоение категориям в зависимости от порядка их появления целочисленных значений, начиная с 1 (Ordinal Encoding)

Категориям переменной в зависимости от порядка их появления в наборе можно присвоить целочисленные значения (начиная с 1) и в итоге категориальная переменная теряет свою категориальную природу и превращается в количественную.

Допустим, у нас есть переменная с 4 категориями A, B, C и D. Первой категорией в нашем наборе будет категория B, второй категорией – категория A, третьей категорией – категория C, четвертой категорией – категория D. Тогда мы присвоим A – 2, B – 1, C – 3, D – 4.

Переменная Class	
A	2
B	1
C	3
D	4

Переменная Class	Кодировка Ordinal Encoding для переменной Class
B	1
A	2
C	3
A	2
A	2
D	4
D	4
B	1

Рис. 20 Пример кодировки Ordinal Encoding

```
In[78]:
# создаем экземпляр класса OrdinalEncoder
ordinal_enc = OrdinalEncoder(cols=['living_region', 'job_position']).fit(X_train, y_train)

# выполняем кодирование переменных
# living_region и job_position
# в обучающем наборе
ordinal_encoded_train = ordinal_enc.transform(X_train)

# взглянем на результаты кодировки
# в обучающем массиве признаков
ordinal_encoded_train.head()
```

```
Out[78]:
```

	living_region	job_position
53397	1	1
143962	2	1
146922	3	1
63697	4	2
54503	5	3

Мы можем выполнить Ordinal Encoding с помощью класса `OrdinalEncoder` пакета `category_encoders` как до разбиения на обучающую и контрольную выборки, так и после разбиения, поскольку не делаем никаких вычислений (разумеется, лучше выполнить Ordinal Encoding до разбиения, чтобы не писать лишний программный код).

Присвоение категориям, отсортированным по процентной доле наблюдений положительного класса зависимой переменной, целочисленных значений, начиная с 0 (еще одна схема Ordinal Encoding)

Для задачи бинарной классификации каждой категории предиктора, отсортированной по процентной доле наблюдений положительного класса зависимой переменной, сопоставляется целое число, начиная с 0. Для задачи регрессии аналогично каждой категории предиктора, отсортированной по среднему значению зависимой переменной в этой категории, сопоставляется целое число, начиная с 0. Допустим, у нас есть бинарная зависимая переменная и предиктор с 4 категориями A, B,

C и D. Сначала посчитаем для каждой категории процентную долю наблюдений, попавших во второй класс зависимой переменной: A – 0,25, B – 0,5, C – 0,33, D – 0,66. Потом отсортируем категории по возрастанию: A – 0,25, C – 0,33, B – 0,5, D – 0,66. Затем каждой категории присваиваем порядковый номер, начиная с 0: A – 0, C – 1, B – 2, D – 3.

Переменная Class	Зависимая переменная	Кодировка SortByResponse для переменной Class
A	0	0
A	1	0
C	1	1
D	1	3
A	0	0
A	0	0
B	0	2
B	1	2
C	0	1
D	1	3
C	0	1
D	0	3

Рис. 21 Пример еще одной кодировки Ordinal Encoding

Данная кодировка реализована в H2O под названием SortByResponse.

Бинарное кодирование (Binary Encoding)

В ходе бинарного кодирования мы выполняем Ordinal Encoding, т.е. каждую категорию представляем в виде целочисленного значения, используем двоичное представление этого числа (например, значение 5 будет представлено как 101, а значение 10 – как 1010) и создаем столбцы в количестве, соответствующем количеству бит, необходимому для этого двоичного представления.

In[79]:
```
# создаем экземпляр класса BinaryEncoder
binary_enc = BinaryEncoder(cols=['living_region', 'job_position']).fit(X_train, y_train)

# выполняем бинарное кодирование
binary_encoded_train = binary_enc.transform(X_train)

# смотрим результаты кодировки в обучающем фрейме
binary_encoded_train.head()
```

Out[79]:

	living_region_0	living_region_1	living_region_2	living_region_3	living_region_4	living_region_5	living_region_6	living_region_7	job_position_0
53397	0	0	0	0	0	0	0	1	0
143962	0	0	0	0	0	0	1	0	0
146922	0	0	0	0	0	0	1	1	0
63697	0	0	0	0	0	1	0	0	0
54503	0	0	0	0	0	1	0	1	0

Наиболее целесообразно применять бинарное кодирование для переменных с очень большим количеством категорий.

Мы можем выполнить Binary Encoding с помощью класса `BinaryEncoder` пакета `category_encoders` как до разбиения на обучающую и контрольную выборки, так и после разбиения, поскольку не делаем никаких вычислений (оптимально выполнить Binary Encoding до разбиения, чтобы не писать лишний программный код).

Создание переменных-взаимодействий

Выше мы говорили, что качество регрессионных моделей можно улучшить за счет создания взаимодействий (мы осуществляем т.н. «инъекцию гибкости»). Давайте создадим такую переменную.

In[80]:

```
# пишем функцию, которая создает взаимодействие
# в результате конъюнкции переменных
# feature1 и feature2
def make_conj(df, feature1, feature2):
    df[feature1 + ' + ' + feature2] = df[feature1].astype(str) + ' + ' + df[feature2].astype(str)
make_conj(train, 'education', 'marital_status')
train.head()
```

Out[80]:

credit_sum_sq	score_sq	income_sq	credit_month_sq	credit_count_sq	education + marital_status
1127549241.000	0.186	1296000000.000	100	4.000	SCH + UNM
552767121.000	0.129	2025000000.000	100	4.000	GRD + UNM
1599200100.000	0.376	2500000000.000	576	36.000	GRD + MAR
12180100.000	0.138	1225000000.000	36	16.000	GRD + UNM
1321904164.000	0.416	2500000000.000	100	4.000	SCH + MAR

Теперь удалим созданную переменную *education + marital_status*.

In[81]:

```
# удалим переменную education + marital_status
train.drop('education + marital_status', axis=1, inplace=True)
```

Категоризация (биннинг) количественной переменной

Для количественных независимых переменных биннинг – это разбивка диапазона значений переменной на интервалы (бины). Например, есть переменная *Возраст* с диапазоном значений от 20 до 70 лет, можно разбить на интервалы: от 18 до 30 лет, от 31 года до 50 лет, от 51 года до 70 лет. В итоге получим категориальную переменную, в которой заданные нами интервалы являются категориями. Для категориальных независимых переменных биннинг – это переназначение (группировка) исходных категорий переменной. Например, есть переменная *Возраст* с категориями от 18 до 25 лет, от 26 до 35 лет, от 36 до 45 лет, от 46 до 55 лет, от 56 до 65 лет. Категории можно укрупнить, из пяти категорий сделать три: от 18 до 35 лет, от 36 до 55 лет, 56 лет и старше.

Основная причина проведения биннинга – это борьба с нелинейностью при построении скоринговых моделей на основе логистической

регрессии. Часто взаимосвязь между непрерывной переменной и событием является нелинейной. Уравнение логистической регрессии, несмотря на то что ее выходное значение подвергается нелинейному преобразованию путем логита, все равно моделирует линейные зависимости между предикторами и зависимой переменной.

Для иллюстрации можно взять пример с нелинейной зависимостью между возрастом и событием (например, откликом). Допустим, рассчитанный регрессионный коэффициент в уравнении логистической регрессии получился отрицательным. Это значит, что вероятность отклика с возрастом уменьшается. После проведенного биннинга, когда были выделены категории от 18 до 25 лет, от 26 до 35 лет, от 36 до 45 лет и старше 45 лет, оказалось, что зависимость между возрастом и событием нелинейная. Первая (молодые) и последняя (старший возраст) категории склонны к отклику, а промежуточные сегменты, наоборот, не склонны к отклику.

Однако у биннинга имеются и серьезные недостатки. Авторитетный статистик Фрэнк Харрелл приводит ряд причин, по которым не следует проводить биннинг количественных независимых переменных:

- потеря прогнозной силы переменной в силу снижение ее информативности (вспомним, что наиболее полную информацию несет количественная шкала);
- в основе биннинга лежит некорректное предположение о том, что зависимость между предиктором и откликом внутри интервалов является монотонной (это предположение еще менее разумно, чем предположение о линейности)
- при разбиении всего диапазона значений переменной на интервалы с равным количеством наблюдений, первый и последний интервалы могут оказаться очень широкими, потому что плотность распределения в них может быть низкой (если взять, например, нормально распределенную величину), если же выполнять разделение на интервалы равной ширины, то количество наблюдений в каждом из них может очень сильно различаться;
- очевидный субъективизм категоризации, выражающийся в том, что если нескольким исследователям предложить категоризировать переменную, они выберут разные границы интервалов.

В силу недостатков, изложенных ниже, биннинг как инструмент борьбы с нелинейностью используется все реже и уступает место преобразованиям на основе ограниченных кубических сплайнов (регрессионных сплайнов, кусочных кубических полиномов), логарифма, корней второй и третьей степени.

В то же время биннинг можно с успехом использовать для создания новых переменных, способных улучшить качество модели.

Категоризация на основе интервалов, заданных вручную

Самый простой вариант биннинга – разбить количественную переменную на определенное количество интервалов, заданных вручную. В питоновской библиотеке pandas биннинг на основе интервалов одинаковой ширины или интервалов, заданных вручную, выполняется с помощью функции `cut()`. Функция `cut()` имеет общий вид:

```
pandas.cut(x, bins, right=True, labels=None,
           precision=3, include_lowest=False)
```

где

`x`	Задает 1-мерный входной массив для биннинга.
`bins`	Задает правило биннинга: • целочисленное значение – определяет количество бинов одинаковой ширины; • последовательность скаляров – определяет точки разбиения, допускается разная ширина бинов; • `IntervalIndex` – определяет точные границы бинов;
`right`	Закрывает интервалы справа, если задано значение `True` (по умолчанию), либо закрывает интервалы слева, если задано значение `False`.
`labels`	Задает метки бинов. Должен иметь длину, совпадающую с количеством бинов. Если задано значение `False` (по умолчанию), возвращает числовые метки бинов.
`precision`	Задает точность, используемую для хранения и отображения числовых меток бинов.
`include_lowest`	Если задано значение `True`, включает самое нижнее значение точек разбиения.

Давайте категоризируем переменную *monthly_income*.

In[82]:

```
# задаем точки, в которых будут находится границы категорий
# (до 50000, от 50000 до 200000, от 200000 и выше)
bins = [-np.inf, 50000, 200000, np.inf]
# задаем метки для категорий будущей переменной
group_names = ['Low', 'Average', 'High']
# осуществляем биннинг переменной monthly_income
# и записываем результаты в новую переменную incomecat
train['incomecat'] = pd.cut(train['monthly_income'], bins, labels=group_names)
train.head()
```

Out[82]:

age_sq	credit_sum_sq	score_sq	income_sq	credit_month_sq	credit_count_sq	incomecat
784.000	1127549241.000	0.186	1296000000.000	100	4.000	Low
1444.000	552767121.000	0.129	2025000000.000	100	4.000	Low
625.000	1599200100.000	0.376	2500000000.000	576	36.000	Low
4225.000	12180100.000	0.138	1225000000.000	36	16.000	Low
2401.000	1321904164.000	0.416	2500000000.000	100	4.000	Low

Удалим созданную переменную *incomecat*.

In[83]:
```
# удалим переменную incomecat
train.drop('incomecat', axis=1, inplace=True)
```

Категоризация на основе квантилей

Количественную переменную можно разбить на квантили. Квантиль – это слово мужского рода с ударением на последнем слоге. По сути это значение, которое заданная случайная величина не превышает с фиксированной вероятностью. Самым известным квантилем является 0,5-квантиль (говорят *квантиль уровня 0,5*) или медиана – значение, которое делит упорядоченный числовой ряд пополам, то есть ровно половина остальных значений больше него, а другая половина меньше его. 0,25-квантиль – это значение, ниже которого будет лежать 25% значений числового ряда, а выше – 75% значений числового ряда. Среди всех возможных квантилей обычно выделяют определенные семейства. Квантили одного семейства делят диапазон изменения признака на заданное число равнонаполненных частей. Семейство определяется тем, сколько частей получается. Наиболее популярными квантилями являются квартили, разбивающие диапазон изменения признака на 4 равнонаполненные части; децили – на 10 равнонаполненных частей; процентили – на 100 частей.

В питоновской библиотеке pandas биннинг на основе квантилей выполняется с помощью функции `qcut()`. Функция `qcut()` имеет общий вид:

```
pandas.qcut(x, q, labels=None, retbins=False,
            precision=3, duplicates='raise')
```

где

`x`	Задает 1-мерный входной массив или серию для биннинга.
`q`	Задает количество квантилей (целое число или массив квантилей)
`labels`	Задает метки бинов. Должен иметь длину, совпадающую с количеством бинов. Если задано значение `False` (по умолчанию), возвращает числовые метки бинов.
`retbins`	Возвращает бины или метки.
`precision`	Задает точность, используемую для хранения и отображения числовых меток бинов.
`duplicates`	Если границы бинов не уникальны, выдает `ValueError` или удаляет их.

In[84]:
```
# осуществляем биннинг переменной monthly_income
# на основе децилей и записываем результаты
# в новую переменную income_decile
train['income_decile'] = pd.qcut(train['monthly_income'], 10)
train.head()
```

Out[84]:

credit_sum_sq	score_sq	income_sq	credit_month_sq	credit_count_sq	income_decile
1127549241.000	0.186	1296000000.000	100	4.000	(35000.0, 40000.0]
552767121.000	0.129	2025000000.000	100	4.000	(40000.0, 45000.0]
1599200100.000	0.376	2500000000.000	576	36.000	(45000.0, 50000.0]
12180100.000	0.138	1225000000.000	36	16.000	(30000.0, 35000.0]
1321904164.000	0.416	2500000000.000	100	4.000	(45000.0, 50000.0]

Обратите внимание на квадратные и круглые скобки интервалов. Интервал закрывается либо слева, либо справа, то есть соответствующий конец включается в данный интервал. Согласно принятой в математике нотации интервалов круглая скобка означает, что соответствующий конец не включается (открыт), а квадратная – что включается (закрыт). По умолчанию интервалы открыты слева и закрыты справа. Интервал `(35000.0,40000.0]` содержит наблюдения, в которых значение переменной больше 35000, но при этом меньше или равно 40000.

Обратите внимание, что, нельзя создать с помощью биннинга новую переменную на общем наборе данных, а потом разбить набор на обучение и контроль и работать с такой переменной в соответствующей выборке, как с обычной исторической переменной. Это обусловлено тем, что для биннинга используется информация о распределении значений переменной по всему набору данных. В результате получится, что в контрольной выборке мы будем использовать переменную, категории которой были получены, исходя из информации всего набора данных. Здесь важно понять, что с помощью биннинга мы на обучающей выборке получаем правила дискретизации (для количественных переменных) и правила перегруппировки (для категориальных переменных), которые применяются к соответствующей переменной в обучающей и контрольной выборках.

Давайте удалим недавно созданную переменную *income_decile*.

In[85]:
```
# удалим переменную income_decile
train.drop('income_decile', axis=1, inplace=True)
```

Теперь выясним, что представляет из себя динамическое конструирование признаков (dynamic feature engineering). Напомним, динамическое конструирование признаков подразумевает, что мы создаем признаки в ходе построения модели «на лету», эти признаки создаются только в процессе обучения модели и в наш набор данных не записываются.

Ранее мы говорили, что для ансамблей на основе деревьев решений часто бывает полезно преобразовать категориальный предиктор в

количественный, представив каждую категорию в виде числа. Однако это преобразование мы выполняли вручную. В новой библиотеке градиентного бустинга `catboost` категориальные признаки превращаются в количественные автоматически, об этом будет рассказано подробнее в следующих разделах.

В ситуации, когда мы работаем с высокоразмерными разреженными признаков, хорошие результаты позволяет получить библиотека градиентного бустинга `lightGBM`, в которой используется техника под названием *связывание взаимоисключающих признаков* (*Exclusive Feature Bundling* или *EFB*). Используя EFB, мы связываем взаимоисключающие предикторы (т.е. берем те, которые редко принимают ненулевые значения одновременно) для уменьшения количества используемых предикторов.

В библиотеке машинного обучения `h2o` вместо превращения категориального предиктора с большим числом категорий в количественный мы можем укрупнить категории прямо в ходе построения модели случайного леса или градиентного бустинга. Для этого используется гиперпараметр `nbins_cats`. Для гиперпараметра `nbins_cats` по умолчанию используется значение 1024, т. е. для категориального предиктора создается не менее 1024 бинов. Если количество категорий меньше значения гиперпараметра `nbins_cats`, каждая категория получает свой бин.

Допустим, у нас есть переменная *Class*. Если у нее есть уровни A, B, C, D, E, F, G и мы зададим `nbins_cats`=8, то будут сформировано 7 бинов: {A}, {B}, {C}, {D}, {E}, {F} и {G}. Каждая категория получает свой бин. Будет рассмотрено $2^6 - 1 = 63$ точки расщепления. Если мы зададим `nbins_cats`=10, то все равно будут получены те же самые бины, потому что у нас всего 7 категорий. Если количество категорий больше значения гиперпараметра `nbins_cats`, категории будут сгруппированы в бины в лексикографическом порядке. Например, если мы зададим `nbins_cats`=2, то будет сформировано 2 бина: {A, B, C, D} и {E, F, G}. У нас будет одна точка расщепления.

Для предикторов с большим количеством категорий небольшое значение гиперпараметра `nbins_cats` может внести в процесс создания точек расщепления дополнительную случайность (поскольку категории группируется в определенном смысле произвольным образом), в то время как большие значения гиперпараметра `nbins_cats` (например, значение гиперпараметра `nbins_cats`, совпадающее с количеством категорий), наоборот, снижают эту случайность, каждая отдельная категория может быть рассмотрена при формировании точки разбиения, что приводит к переобучению на обучающем наборе.

Очевидно, что гиперпараметр `nbins_cats` может быть полезен при наличии в наборе данных одного или нескольких предикторов с очень большим количеством категорий, поскольку в такой ситуации случайный лес при поиске оптимальных расщепляющих значений будет

склоняться в пользу именно этих предикторов, даже если они не обладают высокой прогнозной силой. Снизив количество категорий для таких переменных за счет выбора меньшего значения гиперпараметра `nbins_cats`, можно скорректировать тенденцию выбирать преимущественно предикторы с большим числом категорий и тем самым улучшить качество модели. В то же время следует помнить, если вы будете таким образом укрупнять категории действительно важного предиктора, вы можете ухудшить качество модели.

Давайте завершим тему с конструированием признаков и будем двигаться дальше. В предыдущих разделах мы заменяли пропуски, создавали новые переменные, давайте еще раз убедимся в отсутствии пропусков в переменных, а также в однаковом количестве переменных в обучающей и контрольной выборках.

In[86]:

```
# убеждаемся в отсутствии пропусков в переменных, а также
# в одинаковом количестве переменных в обучающей
# и контрольной выборках
print(train.info())
print(test.info())
```

Out[86]:

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 119522 entries, 53397 to 149270
Data columns (total 24 columns):
gender                  119522 non-null object
age                     119522 non-null float64
marital_status          119522 non-null object
job_position            119522 non-null object
credit_sum              119522 non-null float64
credit_month            119522 non-null int64
tariff_id               119522 non-null object
score_shk               119522 non-null float64
education               119522 non-null object
living_region           119522 non-null object
monthly_income          119522 non-null float64
credit_count            119522 non-null float64
overdue_credit_count    119522 non-null float64
open_account_flg        119522 non-null object
tariff                  119522 non-null float64
paym                    119522 non-null float64
pti                     119522 non-null float64
tariff_sq               119522 non-null float64
age_sq                  119522 non-null float64
credit_sum_sq           119522 non-null float64
score_sq                119522 non-null float64
income_sq               119522 non-null float64
credit_month_sq         119522 non-null int64
credit_count_sq         119522 non-null float64
dtypes: float64(15), int64(2), object(7)
memory usage: 22.8+ MB
None

<class 'pandas.core.frame.DataFrame'>
Int64Index: 51224 entries, 0 to 170745
Data columns (total 24 columns):
gender                  51224 non-null object
age                     51224 non-null float64
marital_status          51224 non-null object
job_position            51224 non-null object
credit_sum              51224 non-null float64
credit_month            51224 non-null int64
tariff_id               51224 non-null object
score_shk               51224 non-null float64
education               51224 non-null object
```

```
living_region           51224 non-null object
monthly_income          51224 non-null float64
credit_count            51224 non-null float64
overdue_credit_count    51224 non-null float64
open_account_flg        51224 non-null object
tariff                  51224 non-null float64
paym                    51224 non-null float64
pti                     51224 non-null float64
tariff_sq               51224 non-null float64
age_sq                  51224 non-null float64
credit_sum_sq           51224 non-null float64
score_sq                51224 non-null float64
income_sq               51224 non-null float64
credit_month_sq         51224 non-null int64
credit_count_sq         51224 non-null float64
dtypes: float64(15), int64(2), object(7)
memory usage: 9.8+ MB
None
```

Убедившись, что переменные не имеют пропусков, обучающий и датафреймы содержат одинаковое количество переменных, приступаем к следующему этапу предварительной подготовки данных – дамми-кодированию.

Дамми-кодирование и подготовка массивов для обучения и проверки

Функция `pd.get_dummies()` автоматически преобразует все столбцы, которые являются категориальными переменными, в дамми-переменные. Применим ее к нашему датафрейму, при этом напечатаем названия столбцов, которые у нас были до использования прямого кодирования, и названия столбцов, которые появились после использования прямого кодирования.

In[87]:
```
# печатаем названия столбцов до и после
# дамми-кодирования
print("Исходные переменные:\n", list(train.columns), "\n")
train_dummies = pd.get_dummies(train)
print("Переменные после get_dummies:\n", list(train_dummies.columns))

print("Исходные переменные:\n", list(test.columns), "\n")
test_dummies = pd.get_dummies(test)
print("Переменные после get_dummies:\n", list(test_dummies.columns))
```

Для удобства выведем первые 5 наблюдений обучающего датафрейма после выполнения дамми-кодирования. Чтобы полностью просмотреть все полученные переменные, нужно увеличить количество выводимых столбцов.

In[88]:
```
# увеличиваем количество выводимых столбцов
pd.set_option('display.max_columns', 150)
train_dummies.head()
```

Out[88]:

	age	credit_sum	credit_month	score_shk	monthly_income	credit_count	overdue_credit_count	tariff	paym
53397	28.000	33579.000	10	0.431	36000.000	2.000	0.000	1.400	3357.900
143962	38.000	23511.000	10	0.358	45000.000	2.000	0.000	1.320	2351.100
146922	25.000	39990.000	24	0.613	50000.000	6.000	0.000	1.500	1666.250
63697	65.000	3490.000	6	0.371	35000.000	4.000	0.000	1.300	581.667
54503	49.000	36358.000	10	0.645	50000.000	2.000	0.000	1.600	3635.800

Видим, что количественные переменные остались без изменения (они приводятся первыми), а категориальные переменные были преобразованы в количественные переменные. Обратите внимание, что категориальная зависимая переменная *open_account_flg* теперь представлена двумя количественными переменными *open_account_flg_0* и *open_account_flg_1*.

living_region_ЧУВАШСКАЯ	living_region_ЯМАЛО-НЕНЕЦКИЙ	living_region_ЯРОСЛАВСКАЯ	open_account_flg_0	open_account_flg_1
0	0	0	1	0
0	0	0	1	0
0	0	0	1	0
0	0	0	1	0
0	0	0	1	0

Рис. 22 Переменная open_account_flg после выполнения дамми-кодирования

На базе переменной *open_account_flg_1* создаем обучающий и контрольный массивы значений зависимой переменной (обучающий и контрольный массивы меток `y_train` и `y_test`), которые будут использоваться для построения модели и ее проверки соответственно. Согласно некоторым математическим соглашениям в `scikit-learn` для массива меток используется строчная `y`.

In[89]:
```
# создаем обучающий и контрольный массивы значений
# зависимой переменной
y_train = train_dummies.loc[:, 'open_account_flg_1']
y_test = test_dummies.loc[:, 'open_account_flg_1']
```

Теперь на основе датафреймов `train_dummies` и `test_dummies` нам нужно сформировать обучающий и контрольный массивы признаков, которые будет использоваться для построения модели и ее проверки соответственно. В `scikit-learn` для массива данных используется заглавная `X`. Переменные *open_account_flg_0* и *open_account_flg_1* расположились в конце соответствующих датафреймов, поэтому мы можем просто указать диапазон нужных нам столбцов.

In[90]:
```
# создаем обучающий и контрольный массивы
# значений признаков
X_train = train_dummies.loc[:, 'age':'living_region_ЯРОСЛАВСКАЯ']
X_test = test_dummies.loc[:, 'age':'living_region_ЯРОСЛАВСКАЯ']
```

Выбор метрики качества

Теперь все готово для моделирования и нам нужно определиться с метрикой качества. Для моделей классификации тремя важнейшими метриками качества являются правильность, AUC и F1-мера.

При решении бизнес-задач, сводящихся к бинарной классификации, все наблюдения делят на два класса: класс с отрицательными исходами (первый уровень зависимой переменной) и класс с положительными исходами (второй уровень зависимой переменной). Обычно положительный класс обозначает наступление какого-то важного для бизнеса события (оттока, отклика, дефолта) и является интересующим нас классом. Допустим, важным событием является отток, поэтому положительным (интересующим) классом станет класс «Уходит» (соответствует ушедшим клиентам), а отрицательным классом – класс «Остается» (соответствует оставшимся клиентам).

Разбиение на два спрогнозированных класса получают с помощью варьирования порога отсечения – порогового значения спрогнозированной вероятности положительного класса, меняющегося в интервале 0 до 1. По умолчанию используется пороговое значение 0,5. Если вероятность положительного класса больше 0.5, то прогнозируется положительный класс, если она меньше этого порогового значения, прогнозируется отрицательный класс. Возьмем наблюдение, для которого были спрогнозированы вероятности классов 0,44 и 0,56. Нашим положительным классом является класс *Уходит*. В данном случае вероятность класса *Уходит* равна 0,56 и превышает пороговое значение 0,5, поэтому прогнозируется класс *Уходит*.

На основе фактической и спрогнозированной принадлежности наблюдений к отрицательному или положительному классу возможны четыре типа случаев.

TP (*True Positives*) – верно классифицированные положительные примеры, или истинно положительные случаи. Пример истинно положительного случая – ушедший клиент верно классифицирован как ушедший.

TN (*True Negatives*) – верно классифицированные отрицательные примеры, или истинно отрицательные случаи. Пример истинно отрицательного случая – оставшийся клиент верно классифицирован как оставшийся.

FN (*False Negatives*) – положительные примеры, неверно классифицированные как отрицательные (ошибка I рода). Это так называемый «ложный пропуск», когда интересующее нас событие ошибочно не обнаруживается (ложноотрицательные случаи). Пример ложноотрицательного случая – ушедший клиент ошибочно классифицирован как оставшийся.

FP (*False Positives*) – отрицательные примеры, неверно классифицированные как положительные (ошибка II рода). Это «ложная тревога», когда при отсутствии события ошибочно выносится

решение о его присутствии (ложноположительные случаи). Пример ложноположительного случая – оставшийся клиент ошибочно классифицирован как ушедший.
Эти четыре типа случаев образуют следующую матрицу ошибок (рис. 23).

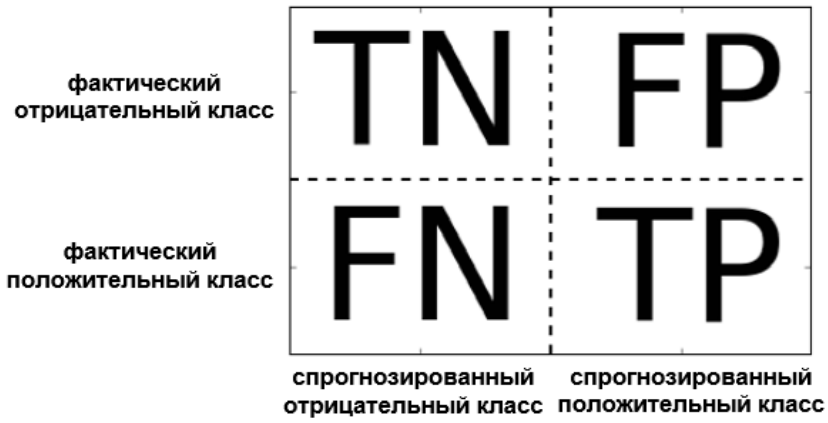


Рис. 23 Матрица ошибок

Давайте построим матрицу ошибок для каких-нибудь произвольных результатов классификации (рис. 24).

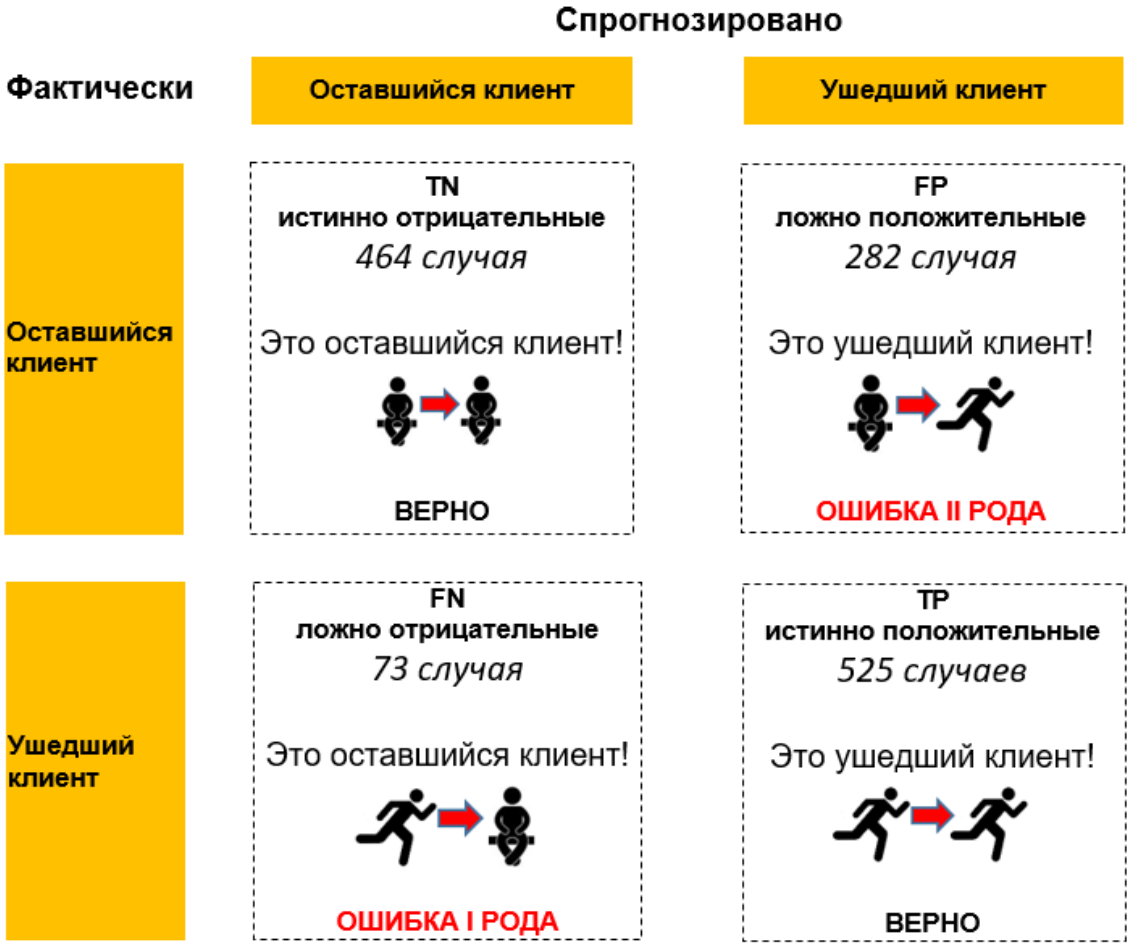


Рис. 24 Четыре типа случаев для нашей таблицы классификации

В первую очередь мы можем вычислить *правильность* (*accuracy*). Ее можно выразить в виде следующей формулы:

$$Acc = \frac{TP + TN}{TP + TN + FP + FN} = \frac{525 + 464}{525 + 464 + 282 + 73} = 0{,}74$$

Рис. 25 Правильность классификации

Правильность – это количество правильно классифицированных случаев (TP+TN, показано на рис. 25 в виде областей красного цвета), поделенное на общее количество случаев (TP+TN+FP+FN, показано на рис. 25 в виде области синего цвета). Правильность еще называют долей правильных ответов.

Следует помнить, что у правильности есть серьезный недостаток, она не может служить достоверной метрикой качества при работе с несбалансированными наборами данных. Представьте, что вам требуется выяснить отклик клиентов на маркетинговое предложение. У вас есть набор данных, в котором 13411 наблюдений соответствуют ситуации «не откликнулся» и 1812 наблюдений – «откликнулся». Другими словами, 88% примеров относятся к классу «не откликнулся». Такие наборы данных, в которых один класс встречается гораздо чаще, чем остальные, часто называют *несбалансированными наборами данных* (*imbalanced datasets*) или *наборами данных с несбалансированными классами* (*datasets with imbalanced classes*). Теперь предположим, что вы

построили модель дерева решений и получили следующую таблицу классификации.

	Не откликнулся	Откликнулся
Не откликнулся	13411	0
Откликнулся	1812	0

В итоге получаем правильность, равную 88%, или (13411+0)/13411+0+1812+0), просто всегда прогнозируя класс «не откликнулся». Правильность по классу *Не откликнулся* у нас будет равна 100%, или 13411/(13411+0), а правильность по классу *Откликнулся* будет равна 0%, или 0/(0+1812). Таким образом, получаем высокое значение правильности при нулевом качестве прогнозирования класса *Откликнулся*.

Кроме правильности есть еще три показателя, с помощью которых можно подытожить информацию матрицы ошибок и тем самым оценить качество классификации − чувствительность (Sensitivity), специфичность (Specificity) и 1 − специфичность (1 − Specificity).

Чувствительность − это количество истинно положительных случаев (True Positive Rate), поделенное на общее количество положительных случаев в выборке. Она измеряется по формуле:

$$Se = TPR = \frac{TP}{TP + FN} = \frac{525}{525 + 73} = 0{,}88$$

В нашем примере чувствительность − это способность модели правильно определять ушедших клиентов. Чувствительность − это правильность классификации для класса *Уходит*. Модель с высокой чувствительностью максимизирует долю правильно классифицированных ушедших клиентов. Чувствительность минимизирует вероятность совершения ошибки I рода, при этом увеличивая вероятность совершения ошибки II рода. Повышая чувствительность, мы минимизируем риск классифицировать ушедшего клиента как оставшегося, но при этом увеличиваем риск классифицировать оставшегося клиента как ушедшего. Образно говоря, увеличивая чувствительность, повышаем «пессимизм» модели. Наша модель демонстрирует высокую чувствительность. Финансовые риски высокочувствительной модели заключаются в том, что мы можем впустую затратить средства на удержание какой-то части лояльных клиентов, ошибочно классифицировав как клиентов, склонных к оттоку.

Обратите внимание, у чувствительности много синонимов. Чувствительность еще называют *полнотой* (*recall*), *процентом результативных ответов* или *хит-рейтом* (*hit rate*).

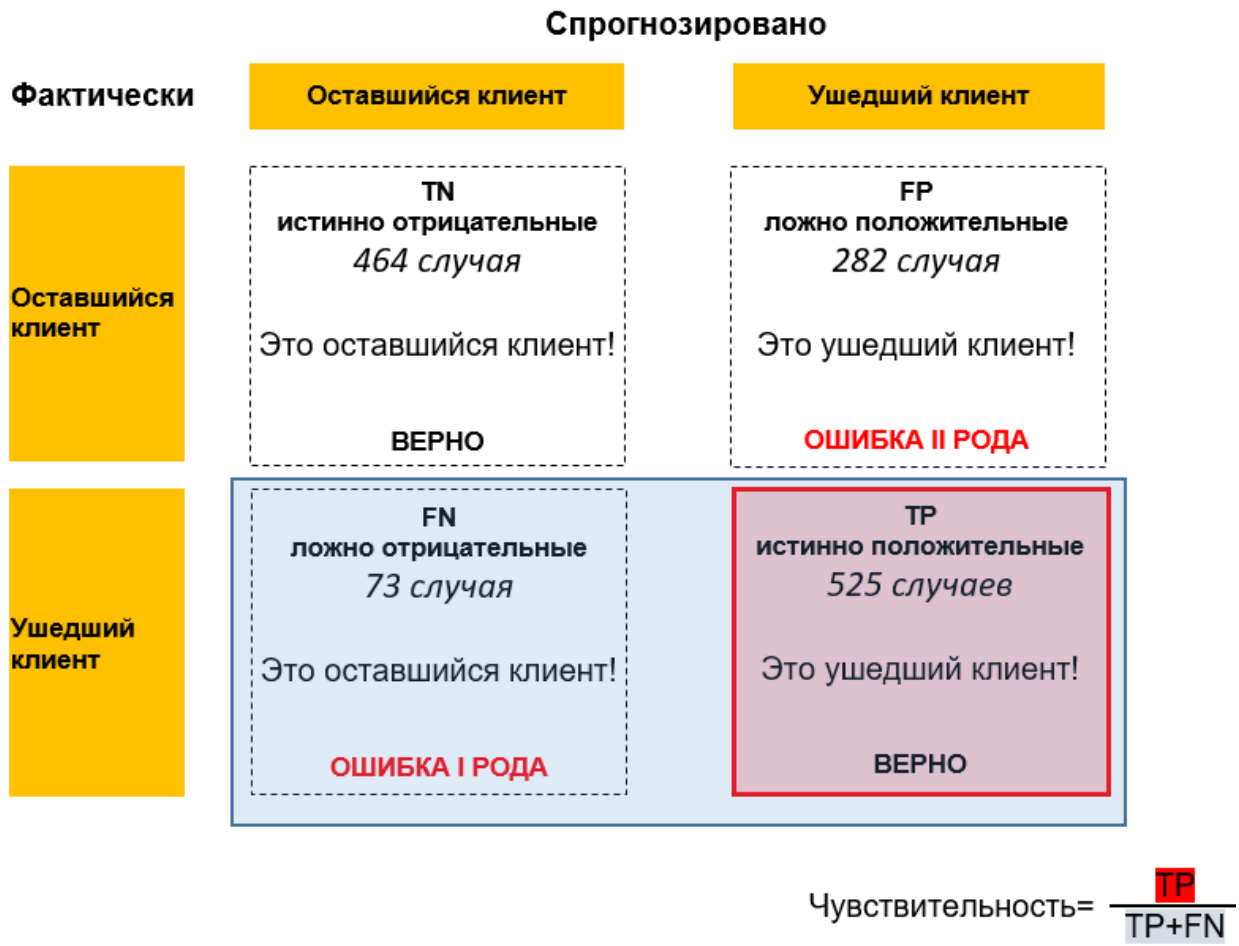


Рис. 26 Чувствительность

Специфичность – это количество истинно отрицательных случаев (True Negatives Rate), поделенное на общее количество отрицательных случаев в выборке. Она измеряется по формуле:

$$Sp = TNR = \frac{TN}{TN + FP} = \frac{464}{464 + 282} = 0{,}62$$

В нашем примере специфичность – это способность модели правильно определять оставшихся клиентов. Специфичность – это правильность классификации для класса *Остается*. Модель с высокой специфичностью максимизирует долю правильно классифицированных оставшихся клиентов. Специфичность минимизирует вероятность совершения ошибки II рода, при этом увеличивая вероятность совершения ошибки I рода. Повышая специфичность, мы минимизируем риск классифицировать оставшегося клиента как ушедшего, но при этом увеличиваем риск классифицировать ушедшего клиента как оставшегося. Образно говоря, увеличивая специфичность, повышаем «оптимизм» модели. Наша модель показывает высокий уровень специфичности. Финансовые риски высокоспецифичной модели заключаются в том, что мы можем потерять какую-то часть клиентов,

которые собирались покинуть компанию, а мы их ошибочно классифицировали как лояльных.

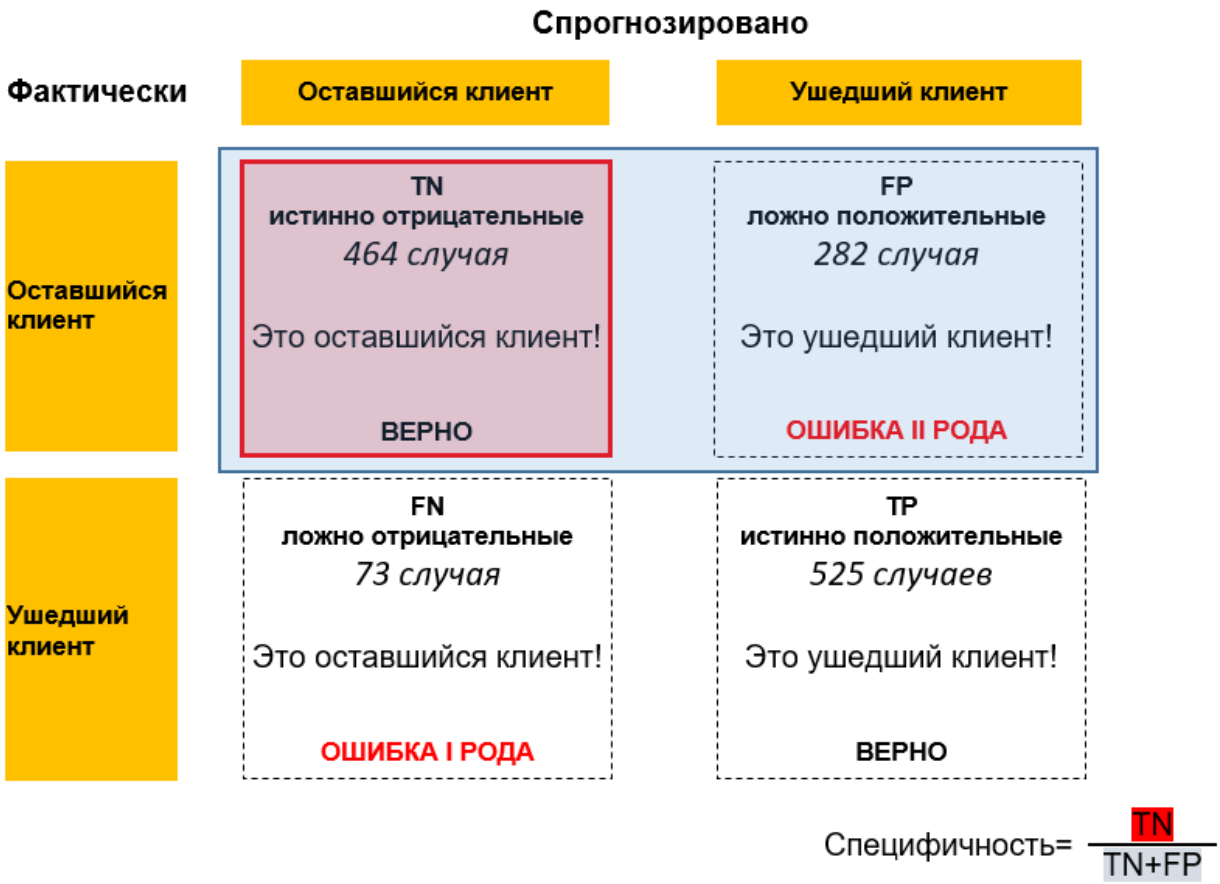


Рис. 27 Специфичность

1 – специфичность (единица минус специфичность) – это количество ложно положительных случаев (False Positives Rate), поделенное на общее количество отрицательных случаев в выборке и вычисляется по формуле:

$$FPR = 1 - Sp = \frac{FP}{FP + TN} = 1 - 0{,}62 = \frac{282}{282 + 464} = 0{,}38$$

В нашем примере 1 – специфичность характеризует уровень «ложных срабатываний» модели, когда оставшийся клиент классифицируется как ушедший.

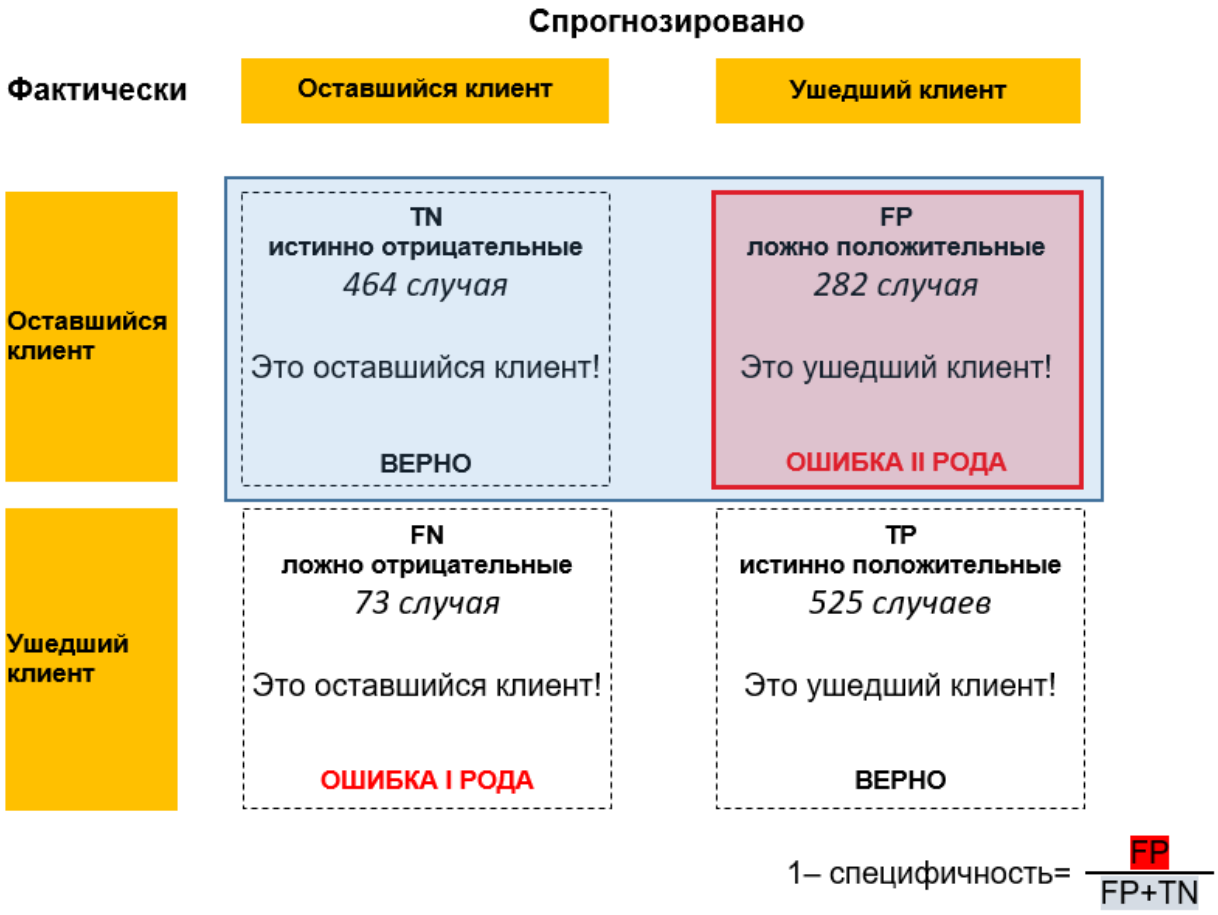


Рис. 28 1 – специфичность

Меняя пороговое значение, с которым сравнивается спрогнозированная вероятность положительного класса, мы будем получать разные результаты классификации, и, соответственно, разные значения вышеприведенных показателей.

Оптимальная модель должна обладать 100%-ной чувствительностью и 100%-ной специфичностью, но добиться этого, как правило, невозможно. Повышая чувствительность, неизбежно снижаем специфичность и, наоборот, понижая чувствительность, неизбежно повышаем специфичность. Поэтому на практике строится ROC-кривая – кривая соотношений истинно положительных случаев (чувствительности) и ложно положительных случаев (1 – специфичности)[11] для различных порогов отсечения и выбирается такой порог отсечения, который дает оптимальное соотношение чувствительности и 1 – специфичности. Понятие «оптимальности» зависит от того, какая задача стоит перед моделером. Можно найти порог, при котором максимизируем долю правильно классифицированных оставшихся клиентов (специфичность), максимизируем долю правильно классифицированных ушедших клиентов (чувствительность) или достигаем баланса между специфичностью и чувствительностью. Давайте подробнее поговорим о ROC-кривой.

[11] Вместо 1-специфичности можно использовать специфичность.

ROC-кривая (англ. receiver operating characteristic – рабочая характеристика приёмника) характеризует способность бинарного классификатора отличать отрицательный класс от положительного при разных порогах отсечения. Эту способность еще называют дискриминирующей способностью. Более строго, ROC-кривая – это кривая соотношений истинно положительных случаев (чувствительности) и ложно положительных случаев (1 – специфичности) для *различных* пороговых значений спрогнозированной вероятности интересующего класса. Используя ROC-кривую, мы не привязаны к конкретному пороговому значению, как в случае с правильностью.

Сам термин «receiver operating characteristic» пришел из теории обработки сигналов времен Второй мировой войны. После атаки на Перл Харбор в 1941 году, когда самолеты японцев были сначала ошибочно приняты за стаю перелетных птиц, а потом за грузовой конвой транспортных самолетов, и перед инженерами-электротехниками и инженерами по радиолокации была поставлена задача увеличить точность распознавания вражеских объектов по радиолокационному сигналу.

Допустим, у нас есть 20 наблюдений, из которых 12 наблюдений относятся к отрицательному классу, а 8 наблюдений – к положительному классу. С помощью бинарного классификатора мы получили следующие спрогнозированные вероятности положительного класса:

№	фактический класс	спрогнозированная вероятность положительного класса
1	N	0,18
2	N	0,24
3	N	0,32
4	N	0,33
5	N	0,4
6	N	0,53
7	N	0,58
8	N	0,59
9	N	0,6
10	N	0,7
11	N	0,75
12	N	0,85
13	P	0,52
14	P	0,72
15	P	0,73
16	P	0,79
17	P	0,82
18	P	0,88
19	P	0,9
20	P	0,92

Рис. 29 Исходные спрогнозированные вероятности положительного класса

Построение ROC-кривой происходит следующим образом.

1. Сначала сортируем все наблюдения в порядке убывания спрогнозированной вероятности положительного класса.
2. Затем создаем график, у которого значения оси абсцисс будут значениями 1 − специфичности (цена деления оси задается значением `1/neg`), а значения оси ординат будут значениями чувствительности (цена деления оси задается значением `1/pos`). При этом `pos` − это количество наблюдений положительного класса, а `neg` − количество наблюдений отрицательного класса.
3. На графике задаем точку с координатами (0, 0) и для каждого отсортированного наблюдения *x*:
   - если *x* принадлежит положительному классу, двигаемся на `1/pos` вверх;
   - если *x* принадлежит отрицательному классу, двигаемся на `1/neg` вправо.

Итак, давайте отсортируем данные в порядке убывания спрогнозированных вероятностей положительного класса.

№	фактический класс	спрогнозированная вероятность положительного класса
20	P	0,92
19	P	0,9
18	P	0,88
12	N	0,85
17	P	0,82
16	P	0,79
11	N	0,75
15	P	0,73
14	P	0,72
10	N	0,7
9	N	0,6
8	N	0,59
7	N	0,58
6	N	0,53
13	P	0,52
5	N	0,4
4	N	0,33
3	N	0,32
2	N	0,24
1	N	0,18

Рис. 30 Отсортированные спрогнозированные вероятности положительного класса

Теперь строим ROC-кривую. Цена деления оси ординат (оси чувствительности) будет равна 1/8, поскольку у нас 8 наблюдений положительного класса. Цена деления оси абсцисс (оси 1 – специфичности) будет равна 1/12, поскольку у нас 12 наблюдений отрицательного класса. Первое наблюдение принадлежит положительному классу, значит двигаемся на 1/8 вверх. Второе наблюдение тоже принадлежит положительному классу, значит снова двигаемся на 1/8 вверх. Третье наблюдение вновь принадлежит положительному классу, значит опять двигаемся на 1/8 вверх. Четвертое наблюдение принадлежит отрицательному классу, двигаемся на 1/12 вправо и так далее.

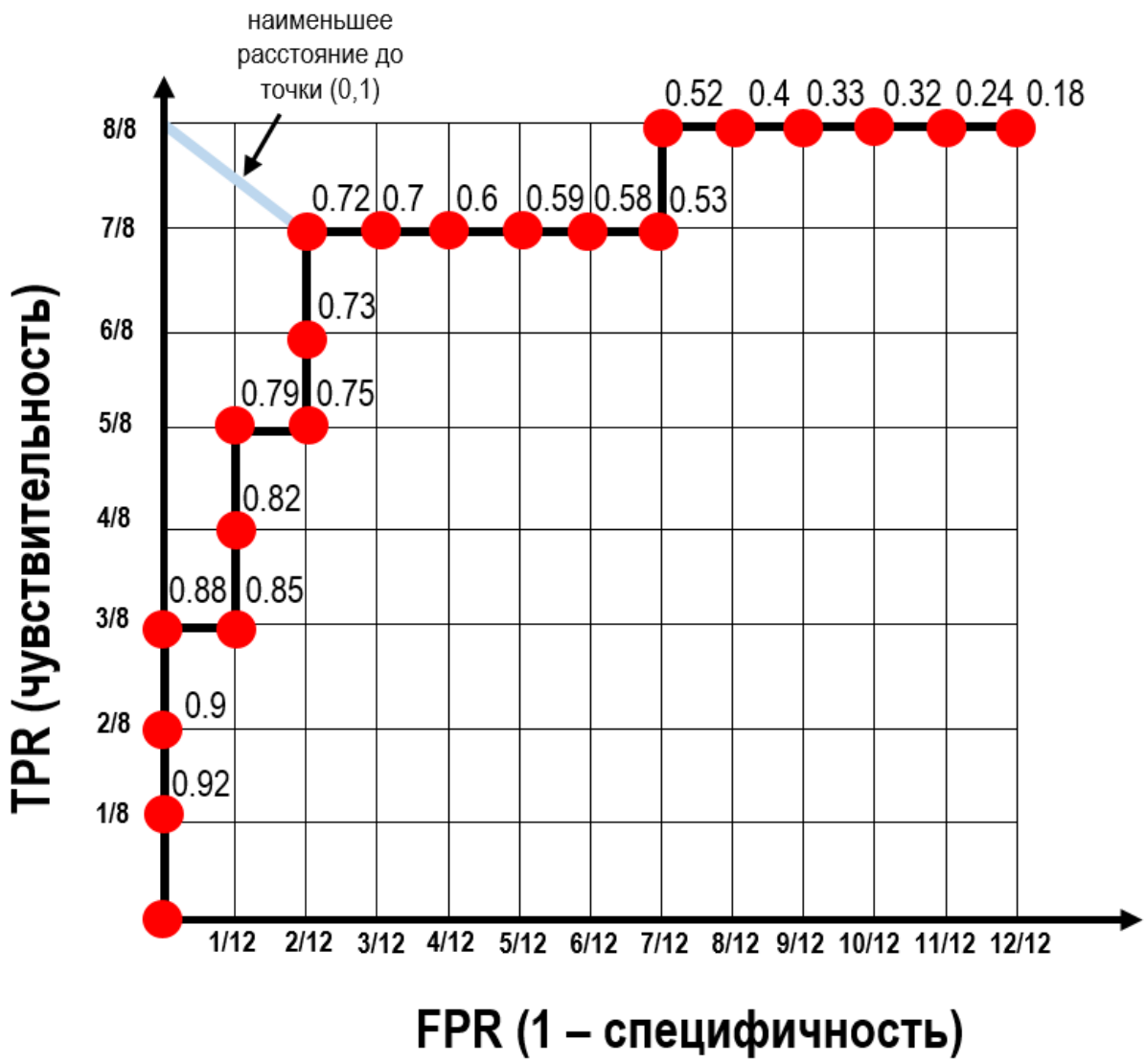


Рис. 31 Построение ROC-кривой вручную

По мере построения ROC-кривой для каждого значения вероятности положительного класса записываем соответствующие ей пары значений 1 − специфичности и чувствительности (координаты).

№	фактический класс	спрогнозированная вероятность положительного класса	1 – специфичность или FPR (при данном пороге вероятности из 12 отрицательных наблюдений n наблюдений будут неверно классифицированы как положительные)	Чувствительность или TPR (при данном пороге вероятности из 8 положительных наблюдений n наблюдений будут верно классифицированы как положительные)
20	P	0,92	0	1/8
19	P	0,9	0	2/8
18	P	0,88	0	3/8
12	N	0,85	1/12	3/8
17	P	0,82	1/12	4/8
16	P	0,79	1/12	5/8
11	N	0,75	2/12	5/8
15	P	0,73	2/12	6/8
14	P	0,72	2/12	7/8
10	N	0,7	3/12	7/8
9	N	0,6	4/12	7/8
8	N	0,59	5/12	7/8
7	N	0,58	6/12	7/8
6	N	0,53	7/12	7/8
13	P	0,52	7/12	8/8
5	N	0,4	8/12	8/8
4	N	0,33	9/12	8/8
3	N	0,32	10/12	8/8
2	N	0,24	11/12	8/8
1	N	0,18	12/12	8/8

Рис. 32 Отсортированные спрогнозированные вероятности положительного класса и соответствующие значения FPR и TPR

Значение вероятности положительного класса, при котором ROC-кривая находится на минимальном расстоянии от верхнего левого угла – точки с координатами (0,1), дает наибольшую правильность классификации. В данном случае таким значением будет значение 0,72. Из таблицы видно, что при пороге 0,72 мы правильно классифицируем 83,3% отрицательных (10 из 12 отрицательных, потому что согласно столбцу «FPR» только 2 из 12 отрицательных будут неверно классифицированы) и 87,5% (7 из 8 положительных согласно столбцу «TPR»). Таким образом, мы правильно классифицируем 17 человек из 20, т.е. правильность составляет 85%. ROC-кривая позволила нам найти такой порог отсечения, при котором чувствительность и 1 – специфичность практически сбалансированы.

При идеальной классификации график ROC-кривой проходит через верхний левый угол. В этом случае доля истинно положительных

примеров составляет 100%, а доля ложно положительных примеров равна 0%. Поэтому, чем ближе кривая к верхнему левому углу, тем выше дискриминирующая способность модели.

Визуальное сравнение двух и более ROC-кривых не всегда позволяет выявить наиболее эффективную модель. Для сравнения двух и более ROC-кривых сравниваются площади под кривыми. Площадь под ROC-кривой часто обозначают как AUC или AUC-ROC (Area Under ROC Curve). Она меняется от 0,5 до 1. Чем больше значение AUC, тем выше качество модели. Наряду с правильностью, AUC – один из важнейших показателей обобщающей способности для моделей бинарной классификации. Обычно считают, что значение AUC от 0,9 до 1 соответствует отличной дискриминирующей способности модели, 0,8–0,9 – очень хорошей, 0,7–0,8 – хорошей, 0,6–0,7 – средней, 0,5–0,6 – неудовлетворительной. AUC можно вычислить с помощью численного метода трапеций, когда площадь под ROC-кривой аппроксимируется суммой площадей трапеций под кривой. Однако есть и другие способы вычислить AUC.

AUC классификатора C – это вероятность того, что классификатор C присвоит случайно отобранному положительному примеру более высокий ранг, чем случайно отобранному отрицательному примеру. Таким образом, $AUC(C) = P[C(x^+) > C(x^-)]$. Таким образом, можно просто подсчитать долю случаев, когда случайно отобранный положительный пример был проранжирован выше, чем случайно отобранный отрицательный пример.

Кроме того, есть еще один способ вычислить AUC, не прибегая к методу трапеций. В рамках этого способа AUC вычисляется по формуле:

$$AUC = \frac{1}{PN}\sum_{j=1}^{N}(s_j - j) = \frac{1}{PN}\sum_{j=1}^{N}\sum_{t=1}^{s_j - j} 1$$

где:

P – количество наблюдений положительного класса;

N – количество наблюдений отрицательного класса;

s_j – ранг j-го наблюдения с отрицательным классом в последовательности, отсортированной по мере убывания вероятности положительного класса;

$s_j - j$ – количество наблюдений положительного класса, лежащих выше j-го наблюдения отрицательного класса.

Итак, для каждого наблюдения с отрицательным классом в отсортированном наборе мы, двигаясь снизу вверх, вычисляем количество наблюдений с положительным классом, которые имеют большее значение вероятности.

№	фактический класс	спрогнозированная вероятность положительного класса	$s_j - j$
20	P	0,92	
19	P	0,9	
18	P	0,88	
12	N	0,85	3
17	P	0,82	
16	P	0,79	
11	N	0,75	5
15	P	0,73	
14	P	0,72	
10	N	0,7	7
9	N	0,6	7
8	N	0,59	7
7	N	0,58	7
6	N	0,53	7
13	P	0,52	
5	N	0,4	8
4	N	0,33	8
3	N	0,32	8
2	N	0,24	8
1	N	0,18	8

Рис. 33 Отсортированные спрогнозированные вероятности положительного класса (подсчет количества наблюдений положительного класса, лежащих выше соответствующего наблюдения отрицательного класса)

Берем наблюдение отрицательного класса 1 (самая нижняя строка таблицы). Подсчитываем количество наблюдений положительного класса, лежащих выше его. Такими наблюдениями будут наблюдения 13, 14, 15, 16, 17, 18, 19 и 20. Всего 8 наблюдений. Записываем это число напротив наблюдения 1. Аналогичный процесс повторяем для каждого наблюдения отрицательного класса, поднимаясь вверх. Затем суммируем полученные значения и сумму делим на произведение количества положительных примеров и количества отрицательных примеров.

$$AUC = \frac{1}{8 \times 12}(8 + 8 + 8 + 8 + 8 + 7 + 7 + 7 + 7 + 7 + 5 + 3) = \frac{83}{96} = 0{,}865$$

На практике бывают ситуации, когда при упорядочении наблюдений по убыванию вероятности положительного класса два наблюдения, принадлежащие разным классам, получают одинаковые вероятности.

№	фактический класс	спрогнозированная вероятность положительного класса
20	P	0,92
19	P	0,9
18	P	0,88
12	N	0,85
17	P	0,82
16	P	0,79
11	N	0,75
15	P	0,75
14	P	0,72
10	N	0,7
9	N	0,6
8	N	0,59
7	N	0,58
6	N	0,53
13	P	0,52
5	N	0,4
4	N	0,33
3	N	0,32
2	N	0,24
1	N	0,18

Рис. 34 Отсортированные спрогнозированные вероятности положительного класса, двум наблюдениям разных классов присвоены одинаковые вероятности

Для таких наблюдений построение ROC-кривой осуществляется иначе. Вообще говоря, здесь могут быть две стратегии. Первая стратегия, которую называют «пессимистичной», заключается в том, чтобы поместить в начало такой последовательности сначала отрицательный пример, а затем положительный (двигаемся вправо и затем вверх, получаем нижний L-образный сегмент). Вторая стратегия, которую называют «оптимистичной», заключается в том, чтобы поместить в начало такой последовательности сначала положительный пример, а затем отрицательный (двигаемся вверх и затем вправо, получаем верхний L-образный сегмент). Компромиссная стратегия, применяющаяся на практике, заключается в усреднении пессимистичного и оптимистичного сегментов. Усреднением будет диагональ, проведенная в прямоугольнике, образованном этими двумя наблюдениями.

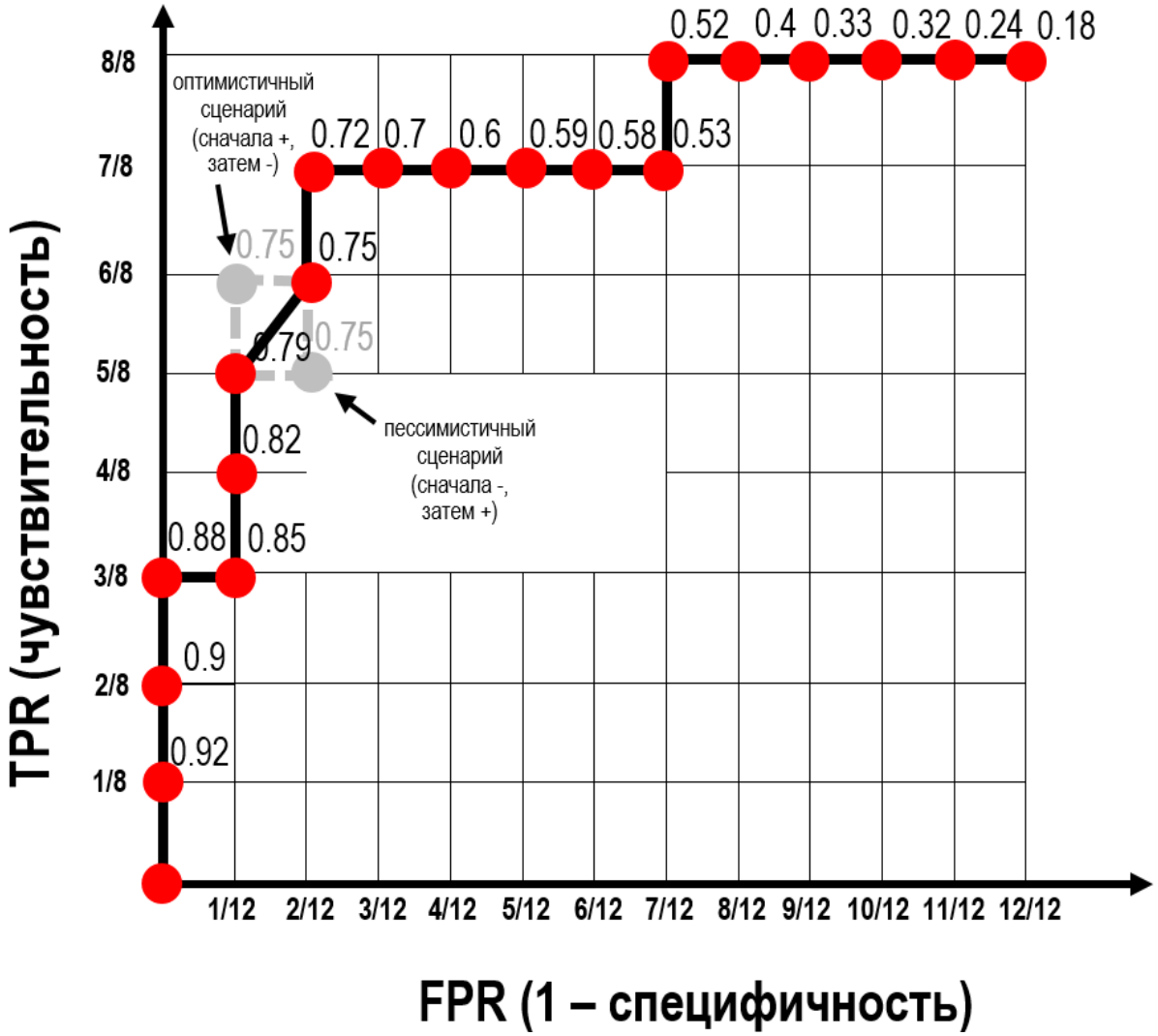


Рис. 35 Построение ROC-кривой вручную в том случае, когда двум наблюдениям, относящимся к разным классам, присвоены одинаковые вероятности

Обратите внимание, что, в отличие от правильности, оценка AUC устойчива к дисбалансу классов, потому что она не использует информацию об исходных распределениях классов и их размерах, для нее важен лишь порядок ранжирования объектов. AUC не зависит от преобразования спрогнозированных вероятностей (например, возведения в квадрат или деления на 2), поскольку зависит не от самих спрогнозированных вероятностей, а от меток классов наблюдений при упорядочивании по этим вероятностям.

Давайте вернемся к нашей матрице ошибок. Еще один показатель, который вычисляют с помощью матрицы ошибок – это *точность* (*precision*).

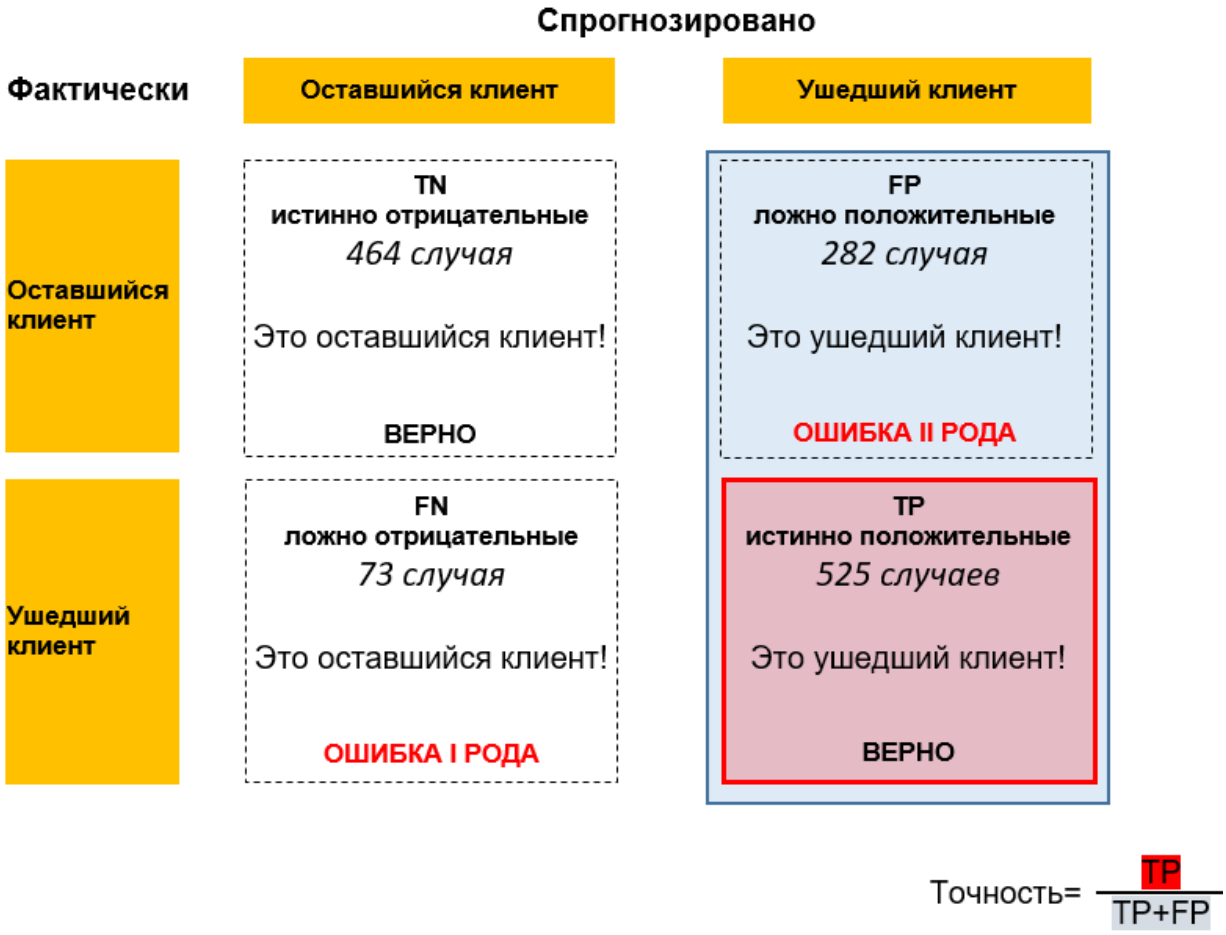


Рис. 36 Точность

Точность – это количество истинно положительных случаев, поделенное на общее количество предсказанных положительных случаев. Для наших данных она измеряется по формуле:

$$P = \frac{TP}{TP + FP} = \frac{525}{525 + 282} = 0{,}65$$

Точность и чувствительность (полнота) похожи тем, что обе метрики интересуют истинно положительные случаи, однако задача точности – определить долю таких случаев в выборке предсказанных положительных случаев, а задачи полноты – определить долю таких случаев в выборке фактических положительных случаев.

Точность важнее правильности, когда вам нужно уменьшить количество ложно положительных случаев, увеличив количество ложно отрицательных случаев. Смысл в том, что ложно положительные случаи имеют более высокую цену ошибки, чем ложно отрицательные случаи.

Для простоты возьмем более образный пример из фантастического фильма. Пусть в условиях зомби-апокалипсиса отрицательным случаем будет человек, превратившийся в зомби, а положительным случаем – здоровый человек. Вы, конечно, постараетесь переместить в безопасную зону максимально возможное количество здоровых людей из общей массы, но вы ведь не хотите ошибочно поместить зомби в безопасную

зону. Поэтому вы стараетесь уменьшить число ложно положительных случаев (когда зомби ошибочно были приняты за здоровых и попали в безопасную зону), увеличив количество ложно отрицательных случаев (когда здоровые люди ошибочно приняты за зомби и не попали в безопасную зону). Если по примерным расчетам мы знаем, что у нас 1000 здоровых и 300 зомби, при этом 950 фактически здоровых находятся в безопасной зоне, а 50 фактически здоровых находятся вне зоны, цена ложно положительного случая будет выше.

TN 300	**FP** 0
FN 50	**TP** 950

Допустим, у нас есть прибор, определяющий, является ли человек зомби. Если прибор имеет идеальную точность, то каждый раз, когда он сообщает «пациент здоров», мы можем доверять ему: действительно пациент здоров. Однако точность не дает никакой информации о том, можем ли мы доверять ему, когда он сообщает «пациент − зомби». Мы должны выяснить, сколько здоровых наш прибор ошибочно принял за зомби (отрицательный класс).
К примеру, у нас есть набор данных из 205 людей, 100 зомби и 105 здоровых. Наш прибор классифицирует пять пациентов как здоровых (и эти пять пациентов действительно здоровы), а всех остальных относит к зомби (отрицательному классу).

TN 100	**FP** 0
FN 100	**TP** 5

Вычисляем точность и полноту. Точность будет идеальной, а полнота − очень низкой.

$$P = \frac{TP}{TP + FP} = \frac{5}{5 + 0} = 1$$

$$Se = \frac{TP}{TP + FN} = \frac{5}{5 + 100} = 0{,}048$$

5 здоровых человек попадут в безопасную зону, а 100 здоровых человек будут обречены.
Теперь представим, наш глупый прибор ВСЕГДА утверждает, что «пациент − здоров».

TN 0	**FP** 100
FN 0	**TP** 105

Мы получаем идеальную полноту!

$$Se = \frac{TP}{TP + FN} = \frac{105}{105 + 0} = 1$$

Должны ли мы заключить, что это идеальное устройство? Нет, мы должны обратиться к точности. Вычисляем точность.

$$P = \frac{TP}{TP + FP} = \frac{105}{105 + 100} = 0{,}51$$

205 человек попадут в безопасную зону, из которых 105 будут здоровыми, а 100 окажутся зомби.
Хотя точность и полнота являются очень важными метриками, сами по себе они не дадут вам полной картины. Одним из способов подытожить их является F-мера (F-measure), которая представляет собой
гармоническое среднее точности и полноты:

$$F = 2 \times \frac{\text{точность} \times \text{полнота}}{\text{точность} + \text{полнота}}$$

Этот вариант вычисления F-меры еще известен как F1-мера. Поскольку F1-мера учитывает точность и полноту, то для бинарной классификации несбалансированных данных она может быть лучшей метрикой, чем правильность.
Ранее был приведен пример, когда нам требовалось выяснить отклик клиентов на маркетинговое предложение. У нас имеется набор данных, в котором 13411 наблюдений соответствуют ситуации «не откликнулся» и 1812 наблюдений – «откликнулся». Мы построили модель дерева и получили следующую таблицу классификации.

	Не откликнулся	Откликнулся
Не откликнулся	13411	0
Откликнулся	1812	0

В итоге получаем правильность, равную 88%, или (13411 + 0) / (13411 + 0 + 1812 + 0), просто всегда прогнозируя класс «не откликнулся». Правильность по классу *Не откликнулся* у нас будет равна 100%, или 13411 / (13411 + 0), а правильность по классу *Откликнулся* (т.е. полнота) будет равна 0%, или 0 / (0 + 1812). Точность будет также равна 0%, или 0 / (0 + 0). Таким образом, получаем высокое значение правильности при нулевом качестве прогнозирования класса *Откликнулся* (при нулевой полноте) и нулевой точности.

Теперь вычислим F1-меру. Нетрудно догадаться, что она будет равна 0. F1-мера действительно дает более лучшее представление о качестве модели, чем правильность в условиях дисбаланса классов. Вместе с тем недостаток F1-меры заключается в том, что, в отличие от правильности, ее труднее интерпретировать и объяснить.
Помимо F1-меры, часто используются F2-мера и F0.5-мера. F1-мера присваивает одинаковый вес точности и полноте. F2-мера используется, когда полноте нужно присвоить вес, в 2 раза больший веса точности. F2-мера вычисляется по формуле:

$$F2 = 5 \times \frac{\text{точность} \times \text{полнота}}{4 \times \text{точность} + \text{полнота}}$$

F0.5-мера используется, когда точности нужно присвоить вес, в 2 раза больший веса полноты.

$$F0.5 = 1{,}25 \times \frac{\text{точность} \times \text{полнота}}{0{,}25 \times \text{точность} + \text{полнота}}$$

По аналогии с ROC-кривой (кривой чувствительности и 1-специфичности) можно построить кривую точности-полноты (PR-кривую) для различных пороговых значений спрогнозированной вероятности положительного класса. При построении ROC-кривой мы по оси *x* откладывали 1 − специфичность, а по оси *y* − чувствительность. При построении PR-кривой по оси *x* откладывают полноту, а по оси *y* − точность.

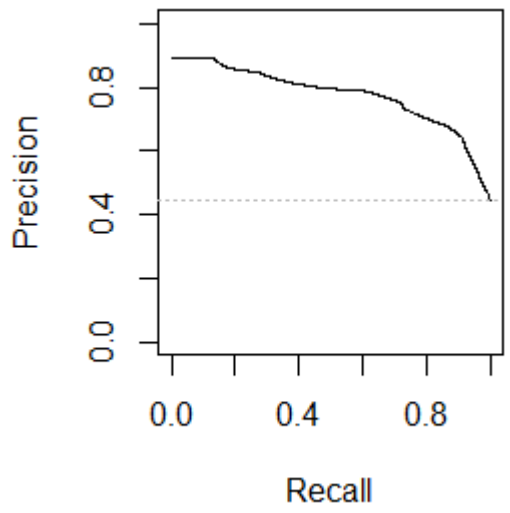


Рис. 37 Пример кривой точности-полноты

Мы помним: чем ближе ROC-кривая подходит к верхнему левому углу, тем лучше классификатор. С PR-кривой все обстоит иначе. Чем ближе PR-кривая подходит к верхнему правому углу, тем лучше классификатор. Точка в верхнем правом углу означает высокое значение

точности и высокое значение полноты для соответствующего порога. Чем больше модель сохраняет высокое значение точности при одновременном увеличении полноты, тем лучше. Мы также можем вычислить площадь под PR-кривой, этот показатель называют AUC-PR. Как взаимосвязаны ROC-кривая и PR-кривая? Исследователи Джесси Дэвис и Марк Гоадрич в своей работе вывели, что если ROC-кривая одного алгоритма лежит полностью над ROC-кривой другого алгоритма, то и PR-кривая одного лежит над PR-кривой другого. Однако, как удалось показать исследователям, алгоритм, который оптимизирует AUC-ROC, не гарантирует оптимизацию AUC-PR.
Импортируем функцию `roc_auc_score()`, которая будет вычислять AUC.

In[91]:
```
# импортируем функцию roc_auc_score
from sklearn.metrics import roc_auc_score
```

Построение моделей случайного леса, градиентного бустинга и логистической регрессии

Сначала построим модель случайного леса. Однако для этого нам придется познакомиться с математическим аппаратом случайного леса.
Основной недостаток деревьев решений – их склонность к переобучению и нестабильность результатов, когда небольшие изменения в наборе данных могут приводить к построению совершенно другого дерева (особенно это актуально для метода CART). Случайный лес стал одним из способов решения этой проблемы. По сути случайный лес – это набор деревьев решений, где каждое дерево немного отличается от остальных. Идея случайного леса заключается в том, что каждое дерево может довольно хорошо прогнозировать, но скорее всего переобучается на определенной части данных. Если мы построим много деревьев, которые хорошо работают и переобучаются с разной степенью, мы можем уменьшить переобучение путем усреднения их результатов.
Для реализации вышеизложенной стратегии нам нужно построить большое количество деревьев решений, то есть ансамбль деревьев. Каждое дерево должно на приемлемом уровне прогнозировать зависимую переменную и должно отличаться от других деревьев (условие декоррелированности деревьев). Для этого в процесс построения деревьев вносим случайность, которая призвана обеспечить уникальность каждого дерева (отсюда случайный лес и получил свое название). Для получения рандомизированных деревьев в случайном лесе последовательно применяются две техники: сначала случайным образом отбираем наблюдения, которые будут использоваться для построения дерева, а затем для каждого узла дерева осуществляем случайный отбор фиксированного количества предикторов для поиска наилучшего расщепления.

Теперь рассмотрим применение этих двух техник в разрезе дилеммы смещения-дисперсии. Вспомним, что смещение − это ошибка, которая характеризует разницу между фактическим и спрогнозированным значением зависимой переменной. Дисперсия − это ошибка, характеризующая чувствительность модели к случайным флуктуациям в обучающем наборе данных. Само по себе одиночное дерево решений имеет низкое смещение (мы можем сделать листья настолько мелкими, что они будут содержать по одному наблюдению и мы получим однозначный прогноз) и высокую дисперсию (изменив всего один признак расщепления в корневом узле, получим совершенно другое дерево из-за иерархической структуры). Случайный лес позволяет значительно уменьшить дисперсию итоговой модели, лишь немного увеличив смещение. Бутстреп и случайный отбор предикторов, призванные уменьшить коррелированность между деревьями, увеличивают смещение модели, однако усреднение по ансамблю уменьшает дисперсию итоговой модели по сравнению с дисперсией отдельного дерева, при этом уменьшение дисперсии позволяет скомпенсировать увеличение смещения и тем самым получить в целом хорошую модель. Кроме того, регулируя глубину, мы также можем регулировать смещение (более глубокие деревья снижают смещение, вспомним, что увеличивая глубину, листья становятся все более мелкими).

Чтобы построить случайный лес, сначала необходимо определиться с количеством деревьев. Допустим, мы хотим построить ансамбль из 5 деревьев. Для построения каждого дерева мы сначала сформируем *бутстреп-выборку* (*bootstrap sample*) наших данных. То есть из набора данных объемом n наблюдений мы случайным образом выбираем наблюдение с возвращением n раз (поскольку отбор с возвращением, то одно и то же наблюдение может быть выбрано несколько раз). Мы получаем выборку, которая имеет такой же размер, что и исходный набор данных, однако некоторые наблюдения будут отсутствовать в нем (примерно 37% наблюдений исходного набора), а некоторые попадут в него несколько раз.

Чтобы проиллюстрировать это, предположим, что мы хотим создать бутстреп-выборку для списка из 10 наблюдений ['1', '2', '3', '4', '5', '6', '7', '8', '9', '10']. Возможная бутстреп-выборка может выглядеть как ['10', '9', '7', '8', '1', '3', '9', '10', '10', '7']. Другой возможной бутстреп-выборкой может быть ['4', '8', '5', '8', '3', '9', '2', '6', '1', '6']. В нашем случае нам нужно построить 5 деревьев, поэтому будет сформировано 5 бутстреп-выборок. Наглядно механизм бутстрепа показан на рис. 38.

Рис. 38 Механизм работы бутстрепа

На основе каждой сформированной бутстреп-выборки строится полное бинарное дерево решений CART[12], то есть разбиения узлов будут продолжаться до тех пор, пока не будет достигнуто минимальное количество наблюдений в терминальных узлах (изначально в качестве минимального количества наблюдений в терминальном узле Лео Брейман, один из авторов случайного леса, предложил для дерева классификации брать значение 1, а для дерева регрессии – значение 5). Однако если в классическом алгоритме дерева решений CART мы для разбиения каждого узла находим наилучшую точку расщепления среди наилучших точек, найденных по *всем* предикторам, то в алгоритм дерева решений CART для случайного леса внесены изменения: для разбиения каждого узла дерева случайным образом отбираем фиксированное подмножество предикторов и затем находим наилучшую точку расщепления среди наилучших точек, найденных по каждому из случайно отобранных предикторов. Давайте выясним, что из себя представляет наилучшая точка расщепления и за одним получше разберемся в работе классического дерева CART.

При построении классического дерева CART узел разбивается на два дочерних узла по наилучшей точке расщепления, которая обеспечивает наибольшее уменьшение неоднородности (улучшение), для этого неоднородность родительского узла сравнивается со взвешенным средним значением неоднородностей дочерних узлов:

[12] В оригинальном подходе Лео Бреймана и в большинстве пакетов используется деревья решений CART, однако существуют реализации, где в качестве деревьев ансамбля используются деревья QUEST или деревья C4.5.

$$\Delta i = i_P - \left(\frac{n_L}{n_P} i_L + \frac{n_R}{n_P} i_R\right)$$

где:

Δi – уменьшение неоднородности (улучшение);
i_P – неоднородность родительского узла;
$\left(\frac{n_L}{n_P} i_L + \frac{n_R}{n_P} i_R\right)$ – взвешенное среднее значение неоднородностей дочерних узлов:
n_L – количество наблюдений в левом дочернем узле;
n_R – количество наблюдений в правом дочернем узле;
n_P – количество наблюдений в родительском узле;
i_L – неоднородность левого дочернего узла;
i_R – неоднородность правого дочернего узла.

Применительно к дереву классификации CART под неоднородностью понимается неоднородность распределения классов зависимой переменной в узле. Однородным узлом является тот, в котором все наблюдения относятся к одному и тому же классу зависимой переменной, в то время как узел с максимальной неоднородностью содержит равное количество наблюдений во всех классах зависимой переменной. Допустим, есть узел, он содержит 6 наблюдений, относящимся к одному из двух классов. Максимальная неоднородность в узле будет достигнута при разбиении его на два класса по 3 наблюдения в каждом, а минимальная неоднородность – при разбиении на 6 наблюдений одного класса и 0 наблюдений другого класса.

Рис. 39 Примеры, иллюстрирующие минимальную и максимальную неоднородность

Наиболее популярная мера неоднородности для деревьев классификации – **мера Джини**. В основе меры Джини лежат возведенные в квадрат вероятности, с которыми наблюдения будут отнесены к каждому классу зависимой переменной.
Общая формула для вычисления меры Джини выглядит так:

$$Gini(t) = 1 - \sum_{k=1}^{K} p_k^2$$

где:
K – количество классов зависимой переменной;
k – класс зависимой переменной;
p_k – вероятность k-того класса зависимой переменной в t-ом узле.

Для бинарной зависимой переменной мера Джини принимает вид:

$$Gini(t) = 1 - p_1^2 - p_0^2$$

где:
p_1^2 – вероятность класса 1 (положительного класса) в t-ом узле;
p_0^2 – вероятность класса 0 (отрицательного класса) в t-ом узле.

Когда наблюдения в узле равномерно распределены по категориям, мера Джини принимает свое максимальное значение (для бинарной зависимой переменной максимальное значение меры Джини равно 0,5). Когда все наблюдения в узле принадлежат к одному и тому же классу, мера Джини равна 0.

Узел с распределением (1, 0)	Мера Джини $= 1 - 1^2 - 0^2 = 0$
Узел с распределением (0,5, 0,5)	Мера Джини $= 1 - 0{,}5^2 - 0{,}5^2 = 0{,}5$
Узел с распределением (0,7, 0,3)	Мера Джини $= 1 - 0{,}7^2 - 0{,}3^2 = 0{,}42$

Предположим, есть данные по клиентам микрофинансовой организации и известно, выплатили они займ или нет (категориальная зависимая переменная *Наличие просрочки*). Для удобства расчетов представим, что наш набор данных состоит всего из 5 наблюдений. В качестве потенциальных предикторов фигурируют две переменные: *Возраст* и *Класс дохода*. Переменная *Возраст* является количественной, переменная *Класс дохода* является категориальной. Необходимо классифицировать клиентов на тех, кто не уйдет в просрочку, и тех, кто уйдет в нее. Схематично наши исходные данные представлены на рис. 40.

Имеется набор данных (корневой узел)

Возраст	Пол	Класс дохода	Наличие просрочки
70	Мужской	A	Да
64	Мужской	B	Да
69	Женский	A	Да
68	Мужской	C	Нет
65	Женский	D	Нет

УЗЕЛ 0

Просрочка		
Нет	2	40%
Да	3	60%

Рис. 40 Исходные данные перед построением дерева классификации CART (корневой узел)

Для количественного предиктора с k уникальными значениями мера Джини будет вычислена для $k - 1$ точек расщепления. Мы видим, что у нас есть значения количественного предиктора *Возраст* 70, 64, 69, 68, 65. Им соответствуют значения категориальной зависимой переменной *Наличие просрочки*: Да, Да, Да, Нет, Нет. Значения предиктора будут упорядочены: 64, 65, 68, 69, 70. В качестве возможных точек расщепления будут рассмотрены средние по каждой паре упорядоченных смежных значений[13]. Будет рассмотрено четыре расщепляющих значения:

Предиктор *Возраст*

Значения	64	65	68	69	70
Точки расщепления		64,5	66,5	68,5	69,5	

Рис. 41 Формирование точек расщепления

В каждой рассматриваемой точке расщепления родительский узел гипотетически разбивается на два дочерних узла (в левый записываются наблюдения со значениями, которые меньше точки расщепления, в правый – наблюдения со значениями, которые больше или равны точке расщепления). Для каждой точки расщепления вычисляется уменьшение Джини – разность между мерой Джини для родительского узла и взвешенным средним значением мер Джини для дочерних узлов. Взвешенное среднее значение мер Джини для дочерних узлов рассчитывается следующим образом. Сначала вычисляются меры Джини для дочерних узлов, полученных при расщеплении. Затем мера Джини для левого дочернего узла умножается на вес левого дочернего узла (долю наблюдений в дочернем левом узле, взятую от общего количества наблюдений в родительском узле). Потом мера Джини для правого дочернего узла умножается на вес правого дочернего узла (долю наблюдений в дочернем правом узле, взятую от общего количества наблюдений в родительском узле). Наконец, произведения суммируются и получается взвешенное среднее значение мер Джини для дочерних узлов.

Наилучшей точкой расщепления для каждого предиктора будет такая точка, которая дает наибольшее уменьшение Джини, т.е. обеспечивает максимальную разность между мерой Джини для родительского узла и взвешенным средним значением мер Джини для дочерних узлов. Для предиктора *Возраст* наилучшей точкой расщепления становится точка 68,5, которая дает наибольшее уменьшение Джини, равное 0,213.

[13] Существуют и другие подходы к формированию точек расщепления.

Предиктор *Возраст*

Значения	64	65	68	69	70
Точки расщепления		64,5	66,5	68,5	69,5

Точки расщепления	<64,5	≥ 64,5	<66,5	≥ 66,5	<68,5	≥ 68,5	<69,5	≥ 69,5
Категория *Нет*	0	2	1	1	2	0	2	0
Категория *Да*	1	2	1	2	1	2	2	1
Уменьшение Джини	0,08		0,013		**0,213**		0,08	

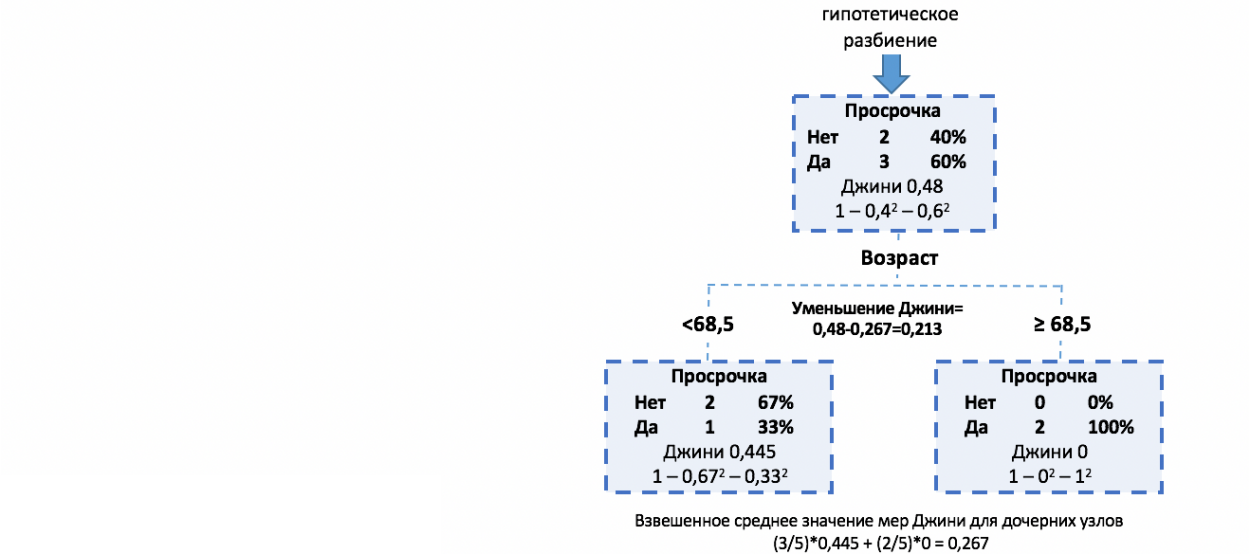


Рис. 42 Поиск наилучшей точки расщепления для количественного предиктора в дереве классификации CART

Для категориального предиктора будут рассмотрены все варианты разбиения категорий на две группы, т.е. для категориального предиктора с k уникальными значениями мера Джини будет вычислена для $2^{k-1} - 1$ точек расщепления и в итоге будет выбрана точка расщепления, дающая наибольшее уменьшение Джини. Например, для предиктора *Класс дохода*, принимающего значения A, B, C, D, будут рассмотрены точки расщепления A и BCD, B и ACD, C и ABD, D и ABC, AB и CD, AC и BD, AD и BC.

Для разбиения узла будет выбрана наилучшая точка из наилучших точек, найденных по предикторам *Возраст* и *Класс дохода*. Например, в нашем случае такой точкой расщепления стала точка расщепления 68,5 по предиктору *Возраст*.

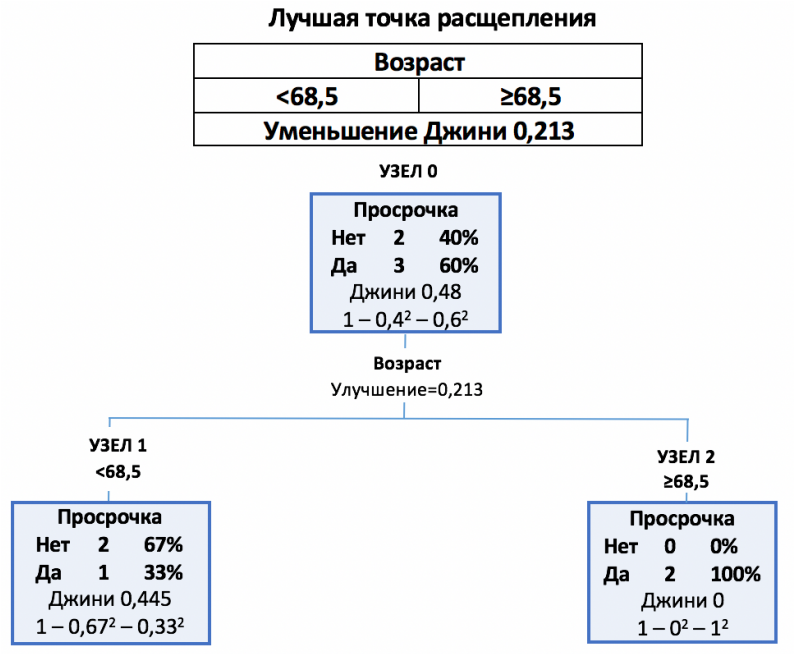


Рис. 43 Выбор наилучшей точки расщепления для разбиения корневого узла дерева классификации CART

Затем для каждого из полученных узлов мы снова будем искать наилучшую точку расщепления по всем предикторам и выбирать наилучшую из лучших.
Кроме меры Джини в качестве показателя неоднородности часто используется энтропия. Она вычисляется по формуле:

$$E(t) = -\sum_{k=1}^{K} p_k \times log_2 p_k$$

Обе меры показаны на рис. 44, где можно четко увидеть, что энтропия (мера Джини) минимальна, когда все наблюдения либо принадлежат отрицательному классу, либо принадлежат положительному классу, и максимальна в случае одинакового количества наблюдений каждого класса.

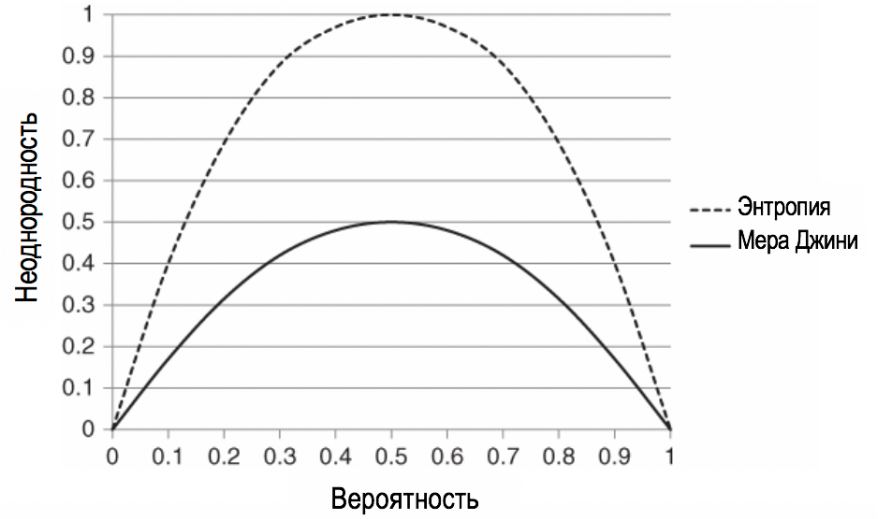


Рис. 44 Сравнение меры Джини и энтропии для двух классов

Применительно к дереву регрессии CART под неоднородностью понимается степень разброса значений количественной зависимой переменной вокруг среднего значения в узле. Здесь мерой неоднородности будет **среднеквадратичная ошибка** – сумма квадратов отклонений (разностей между фактическими значениями зависимой переменной и ее средним значением) в конкретном узле, поделенной на количество наблюдений в этом узле.

$$MSE(t) = \frac{1}{n_t}\sum_i (y_{it} - \bar{y}_t)^2$$

где:

y_{it} – фактическое значение зависимой переменной для *i*-того наблюдения в *t*-ом узле;

$\bar{y}_t$ – среднее значение зависимой переменной для в *t*-ом узле;

n_t – количество наблюдений в *t*-ом узле.

Предположим, есть данные по клиентам микрофинансовой организации и известно количество дней просрочки по каждому (количественная зависимая переменная *Количество дней просрочки*). Для удобства расчетов представим, что наш набор данных состоит всего из 5 наблюдений. В качестве потенциальных предикторов фигурируют две переменные: *Возраст* и *Категория дохода*. Переменная *Возраст* является количественной, переменная *Категория дохода* является категориальной. Необходимо спрогнозировать глубину просрочки по каждому клиенту. Схематично наши исходные данные представлены на рис. 45.

Имеется набор данных (корневой узел)

Возраст	Класс дохода	Количество дней просрочки
70	A	20
64	B	23
69	A	12
68	C	9
65	D	15

УЗЕЛ 0

Количество дней просрочки	
Среднее	15,8
N	5

Рис. 45 Исходные данные перед построением дерева регрессии CART (корневой узел)

Вновь формируем точки расщепления. В каждой рассматриваемой точке расщепления родительский узел гипотетически разбивается на два дочерних узла (в левый записываются наблюдения со значениями, которые меньше точки расщепления, в правый – наблюдения со значениями, которые больше или равны точке расщепления). Для каждой точки расщепления вычисляется уменьшение среднеквадратичной ошибки – разность между среднеквадратичной ошибкой для родительского узла и взвешенным средним значением среднеквадратичных ошибок для дочерних узлов. Взвешенное среднее значение среднеквадратичных ошибок для дочерних узлов рассчитывается следующим образом. Сначала вычисляются среднеквадратичные ошибки для дочерних узлов, полученных при расщеплении. Затем среднеквадратичная ошибка для левого дочернего узла умножается на вес левого дочернего узла (долю наблюдений в дочернем левом узле, взятую от общего количества наблюдений в родительском узле). Потом среднеквадратичная ошибка для правого дочернего узла умножается на вес правого дочернего узла (долю наблюдений в дочернем правом узле, взятую от общего количества наблюдений в родительском узле). Наконец, произведения суммируются и получается взвешенное среднее значение среднеквадратичных ошибок для дочерних узлов.

Наилучшей точкой расщепления для каждого предиктора будет такая точка, которая дает наибольшее уменьшение среднеквадратичной ошибки, т.е. обеспечивает максимальную разность между среднеквадратичной ошибкой для родительского узла и взвешенным средним значением среднеквадратичных ошибок для дочерних узлов. На рис. 46 показано, как была вычислена наилучшая точка расщепления по предиктору *Возраст*.

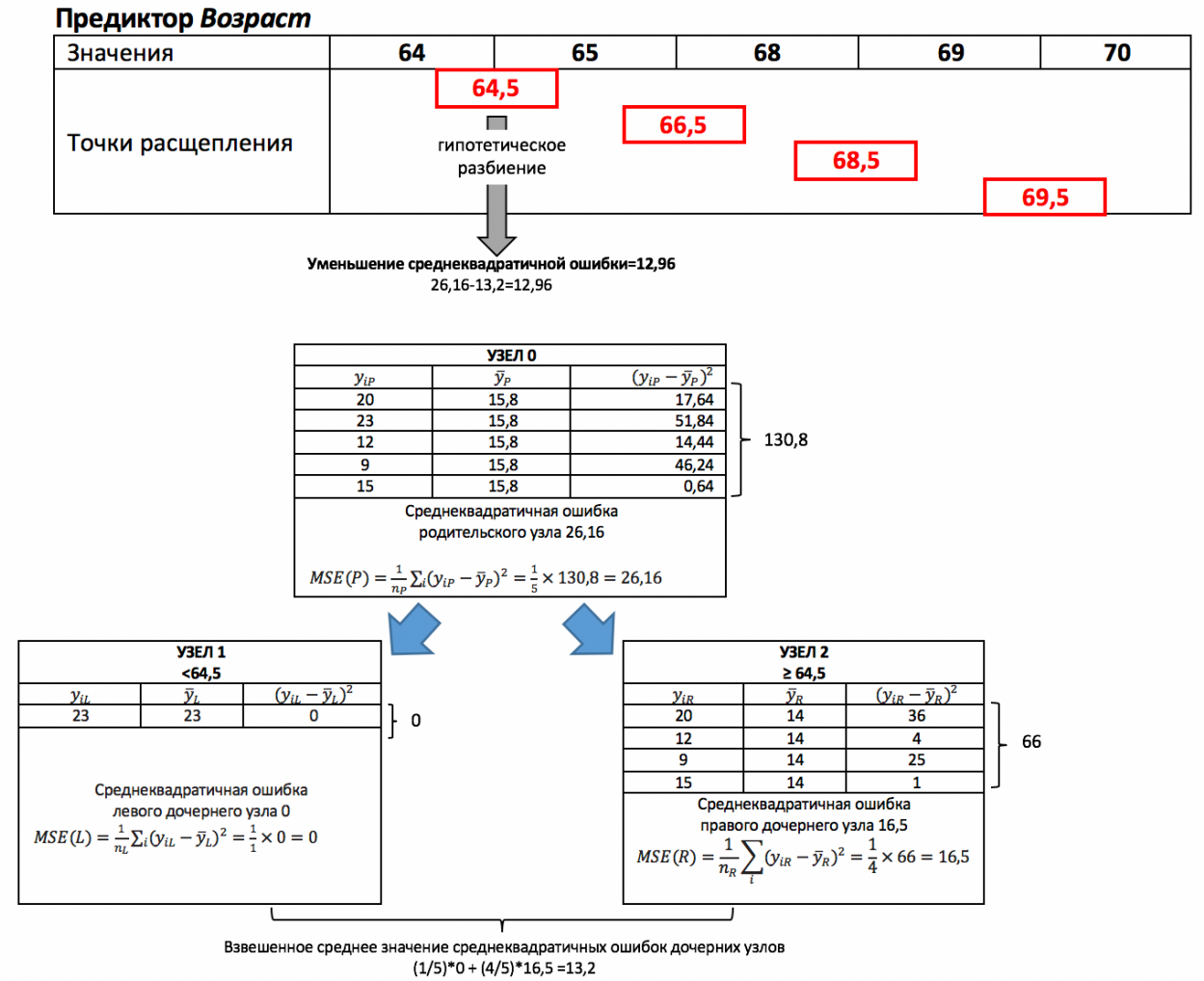


Рис. 46 Вычисление наилучшей точки расщепления для количественного предиктора в дереве регрессии CART

Для разбиения узла будет выбрана наилучшая точка из наилучших точек, найденных по предикторам *Возраст* и *Класс дохода*. В данном случае такой точкой будет точка 64,5, полученная для предиктора *Возраст*.

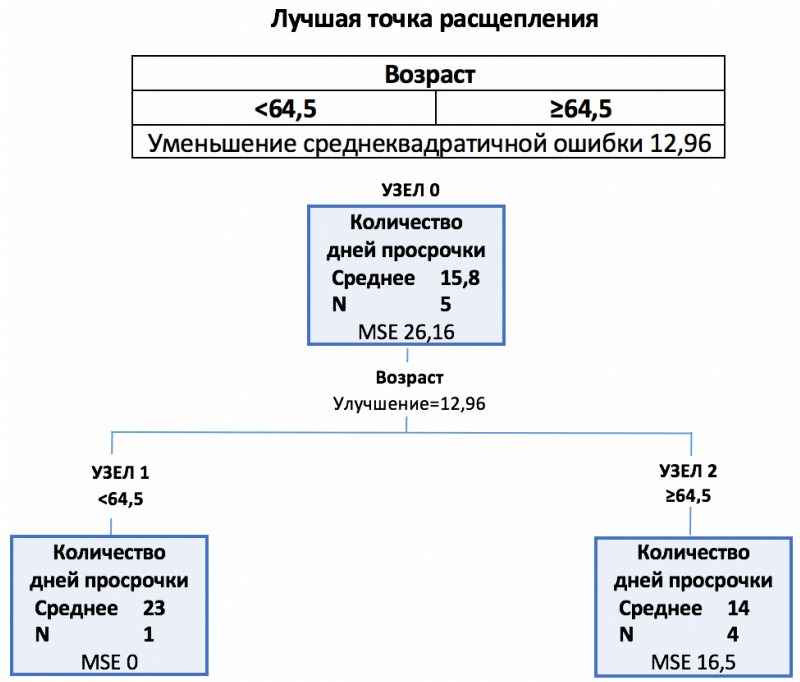


Рис. 47 Выбор наилучшей точки расщепления для разбиения корневого узла дерева регрессии CART

Помимо среднеквадратичной ошибки может использоваться средняя абсолютная ошибка – средний модуль отклонений.

$$MAE = \frac{1}{n_t}\sum_i |y_{it} - \bar{y}_t|$$

Итак, в случайном лесе отбор подмножества предикторов повторяется отдельно для каждого узла, поэтому для разбиения каждого узла дерева может быть использована наилучшая точка расщепления, найденная в «своем» подмножестве случайно отобранных предикторов.

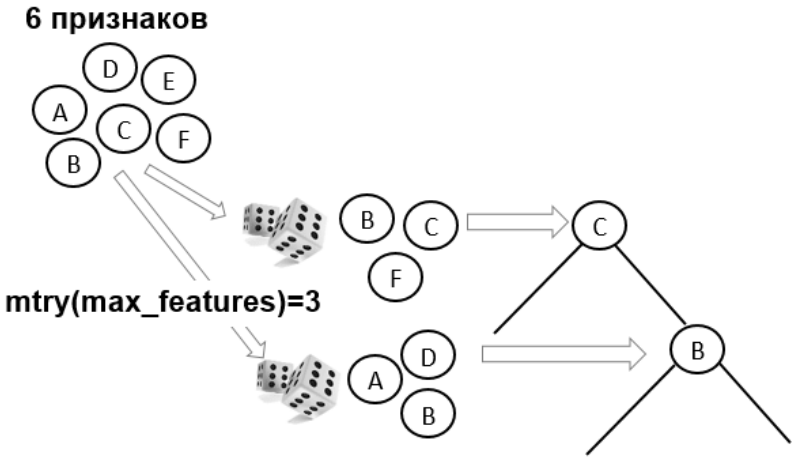


Рис. 48 Случайный отбор предикторов (в данном случае каждый раз происходит случайный отбор 3 предикторов)

Необходимо отметить, что идея случайного отбора определенного количества предикторов в каждом узле дерева появилась не сразу. Сначала Лео Брейман в 1994 году предложил метод бэггинга или бутстреп-агрегирования, когда на основе исходного набора данных мы генерируем бутстреп-выборки, по ним строим полные бинарные деревья и затем агрегируем их результаты путем голосования или простого усреднения. Бэггинг стал предшественником случайного леса. Примерно в это же время Тин Кам Хо предложил идею ансамбля деревьев, в рамках которого для построения каждого дерева используется фиксированное количество случайно отобранных признаков. Ансамбли построенных деревьев Хо назвал случайными лесами решений (Random Decision Forests).

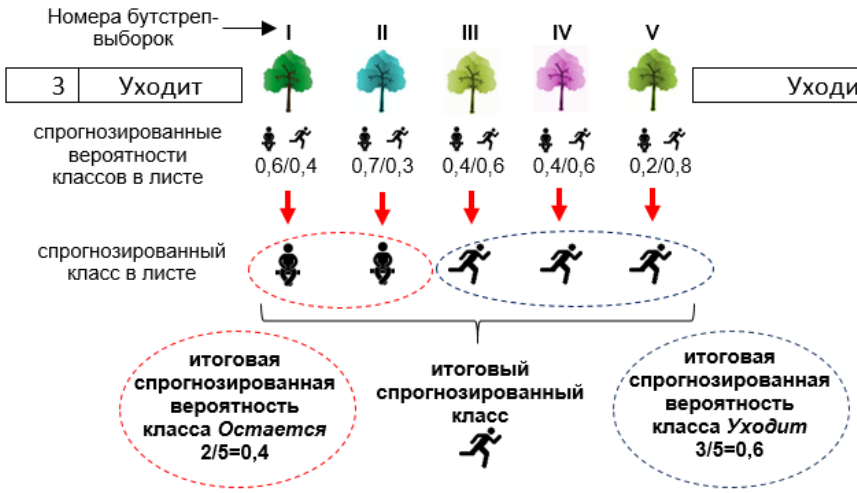
Использование бутстрепа приводит к тому, что деревья решений в случайном лесе строятся на немного отличающихся между собой бутстреп-выборках. Из-за случайного отбора переменных в каждом узле все расщепления в деревьях будут основаны на отличающихся подмножествах предикторов. Вместе эти два механизма приводят к тому, что все деревья в случайном лесе будут отличаться друг от друга.

Решая задачу классификации, каждое дерево сначала вычисляет для наблюдения листовые вероятности классов зависимой переменной. Листовая вероятность класса – это доля объектов класса в листе (терминальном узле) дерева, в который попало классифицируемое наблюдение. Каждое дерево голосует за класс с наибольшей вероятностью в листе. В итоге побеждает класс, за который проголосовало большинство деревьев. Итоговыми вероятностями классов для наблюдения будут доли голосов деревьев, поданных за данный класс.

Данный подход реализован в оригинальном программном коде Лео Бреймана и Адель Катлер, на базе которого написан пакет R `randomForest`. Для каждого дерева фиксируется только «победивший класс», листовые вероятности классов, полученные для отдельного дерева, отбрасываются и в дальнейших расчетах не участвуют. Поэтому в пакете `randomForest` итоговые вероятности классов для наблюдения – это доли голосов деревьев, поданных за данный класс.

№	фактический класс
1	Остается
2	Уходит

спрогнозированный класс (итог голосования деревьев, построенных по всем бутстреп-выборкам)	итоговые вероятности классов (доли голосов деревьев, поданных за соответствующие классы)
Уходит	0,3 0,7
Уходит	0,2 0,8



3	Уходит	Уходит	0,4 0,6

Рис. 49 Получение прогнозов для задачи классификации (оригинальный подход «усреднение прогнозов», реализован в пакете R randomForest)

В питоновской библиотеке `scikit-learn` для моделей `RandomForestClassifier` и `RandomForestRegressor` используется другой подход. Каждое дерево вычисляет для наблюдения листовые вероятности классов. Эти листовые вероятности усредняются по всем деревьям, и в итоге прогнозируется класс с наибольшей усредненной листовой вероятностью. Поэтому для классов `RandomForestClassifier` и `RandomForestRegressor` итоговыми вероятностями классов будут листовые вероятности классов, усредненные по всем деревьям. Аналогичный подход используется для оценки итоговых вероятностей классов в пакете R `ranger`.

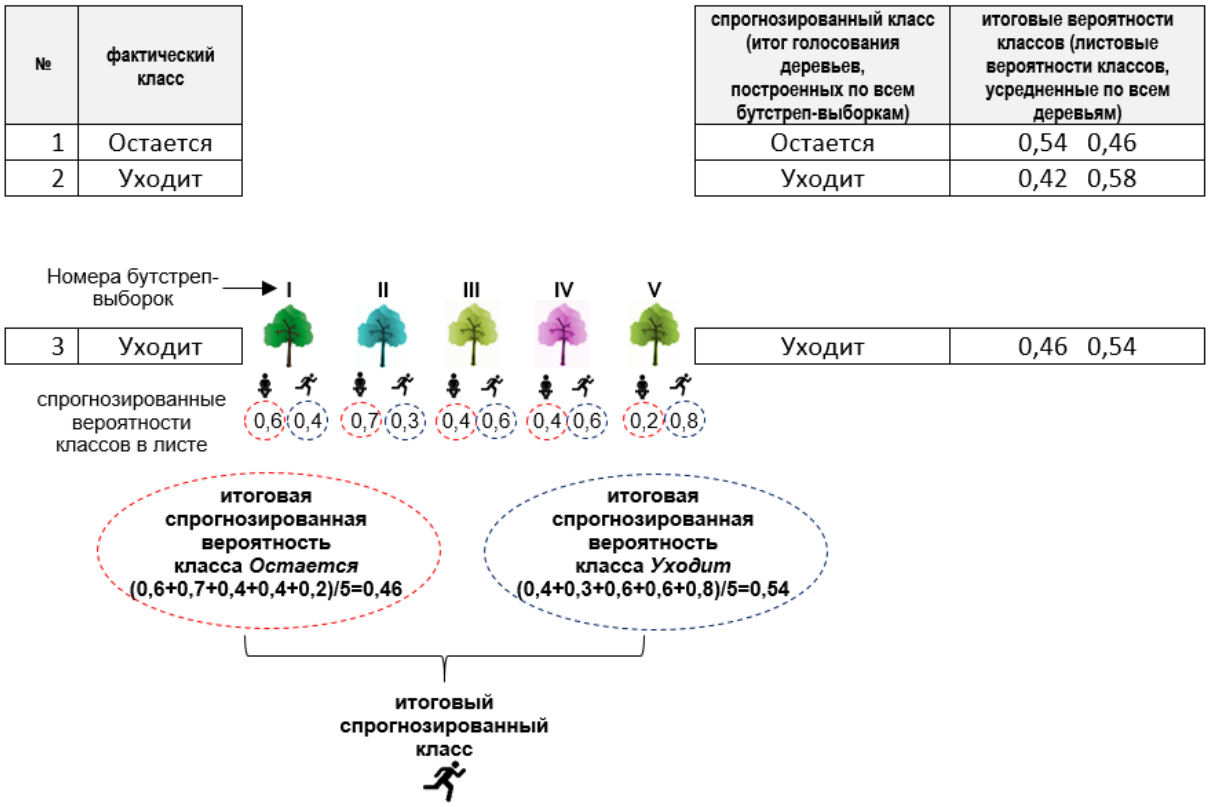


Рис. 50 Получение прогнозов для задачи классификации (подход «усреднение листовых вероятностей», реализован в классе RandomForestClassifier питоновской библиотеки scikit-learn)

Решая задачу регрессии, каждое дерево прогнозирует для наблюдения среднее значение зависимой переменной в листе (терминальном узле), в который это наблюдение попало, и в результате происходит усреднение полученных средних значений по всем деревьям.

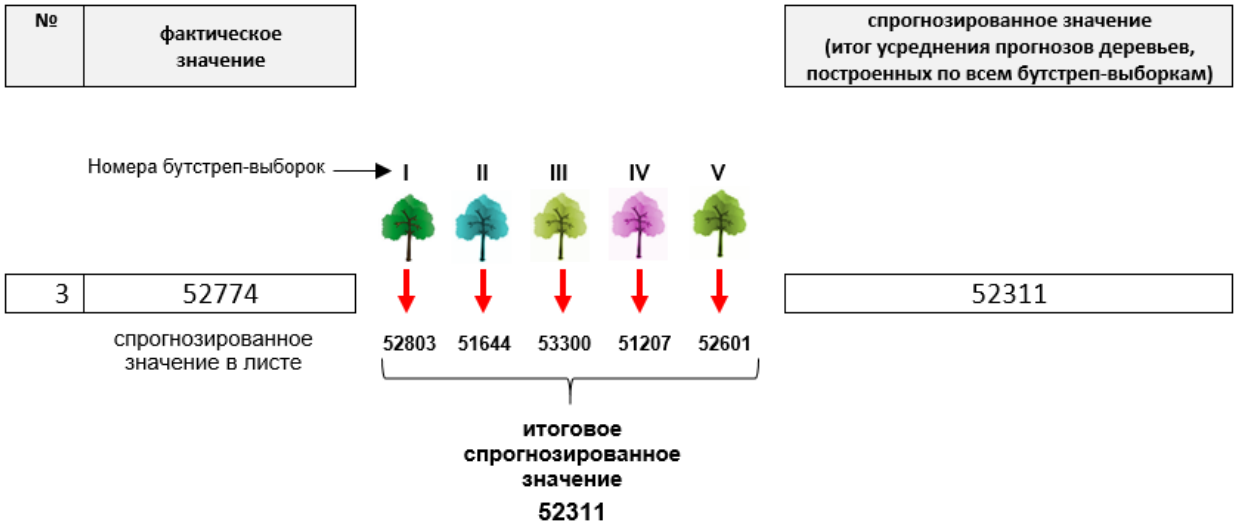


Рис. 51 Получение прогнозов для задачи регрессии

В итоге алгоритм случайного леса можно описать следующим образом:

1. Для b = 1, 2… B (где B – количество деревьев в ансамбле):
   (a) извлечь бутстреп-выборку S размера N из обучающих данных;
   (b) по бутстреп-выборке S построить полное дерево T_b, рекурсивно повторяя следующие шаги для каждого терминального узла, пока не будет достигнуто минимальное количество наблюдений в нем (для классификации – 1 наблюдение, для регрессии – 5 наблюдений):
      i. из первоначального набора M предикторов случайно выбрать m предикторов;
      ii. из m предикторов выбрать предиктор, который обеспечивает наилучшее расщепление;
      iii. расщепить узел на два узла-потомка.
2. В результате получаем ансамбль деревьев решений $\{T_b\}_{b=1}^{B}$.
3. Предсказание новых наблюдений осуществлять следующим образом:

для регрессии: $\hat{f}_{rf}^{B}(x) = \frac{1}{B}\sum_{b=1}^{B} T_b(x)$;

для классификации: пусть $\hat{C}_b(x)$ – класс, спрогнозированный деревом решений T_b, то есть $T_b(x) = \hat{C}_b(x)$; тогда $\hat{C}_{rf}^{B}(x)$ – это класс, наиболее часто встречающийся в множестве $\{\hat{C}_b(x)\}_{b=1}^{B}$

Качество модели случайного леса может быть оценено обычным способом и с помощью метода OOB.
В рамках обычного способа мы берем каждое наблюдение и используем для прогноза деревья, построенные по всем бутстреп-выборкам. Для задачи классификации применяем голосование, подсчитываем количество неправильно классифицированных наблюдений (спрогнозированный класс для наблюдения – это класс, за который проголосовало большинство деревьев, построенных по всем бутстреп-выборкам). Затем делим количество неправильно классифицированных наблюдений на общее количество наблюдений и получаем ошибку классфикации. Для задачи регрессии подсчитываем сумму квадратов остатков – разностей между фактическим и спрогнозированным значением зависимой переменной (спрогнозированное значение для наблюдения – это результат усреднения средних значений, которые вычислили деревья, построенные по всем бутстреп-выборкам). Затем эту сумму делим на количество всех наблюдений и получаем среднеквадратичную ошибку.
В рамках метода OOB мы берем каждое наблюдение и используем для прогноза только те деревья, которые строились по бутстреп-выборкам, не содержащим данное наблюдение (т.е. наблюдение «выпало» из бутстреп-выборки и данную бутстреп-выборку для этого наблюдения можно назвать out-of-bag выборкой, отсюда и название метода). Для задачи классификации применяем голосование, подсчитываем

количество неправильно классифицированных наблюдений (спрогнозированный класс для наблюдения – это класс, за который проголосовало большинство деревьев, построенных по out-of-bag выборкам). Затем делим количество неправильно классифицированных наблюдений на общее количество наблюдений и получаем ошибку классификации по методу OOB. Для задачи регрессии подсчитываем сумму квадратов остатков – разностей между фактическим и спрогнозированным значением зависимой переменной (спрогнозированное значение для наблюдения – это результат усреднения средних значений, которые вычислили деревья, построенные по out-of-bag выборкам). Затем эту сумму делим на количество всех наблюдений и получаем среднеквадратичную ошибку по методу OOB.

Метод OOB используется только для оценки качества модели на обучающей выборке. К каждому наблюдению контрольной выборки мы просто применяем правила, сформулированные каждым деревом леса в ходе обучения. Затем осуществляем голосование или усреднение и на основе полученных прогнозов обычным способом вычисляем метрику качества.

Рассмотрим процесс оценки качества модели с помощью метода OOB наглядно (рис. 52). Допустим, у нас есть исходный набор из 10 наблюдений, на его основе мы сгенерировали 5 бутстреп-выборок. Для каждого наблюдения мы должны зафиксировать бутстреп-выборки, в которых оно отсутствует. Например, на рисунке видно, что наблюдение 2 отсутствует в бутстреп-выборках I и V. Эти выборки будут для наблюдения 2 out-of-bag выборками. Для классификации или вынесения прогноза по наблюдению 2 нас как раз будут интересовать голоса или прогнозы деревьев, построенных по этим двум out-of-bag выборкам.

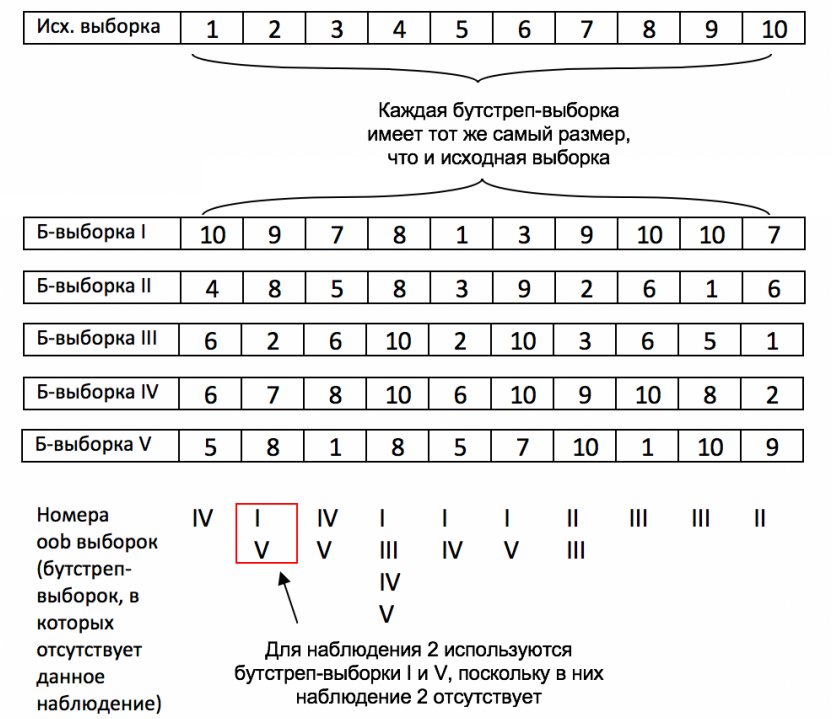


Рис. 52 Out-of-bag выборки для оценки качества модели

Возьмем задачу классификации (рис. 53). Допустим, необходимо отнести наблюдение 4 к тому или иному классу зависимой переменной *Статус клиента [status]*. Фактически оно принадлежит классу *Уходит*. Для оценки качества модели используются только те деревья решений, которые строились по бутстреп-выборкам, не содержащим наблюдение 4, и затем проводится голосование деревьев. Наблюдение 4 отсутствует в 4 бутстреп-выборках: I, III, IV, V. Для данного наблюдения эти бустреп-выборки будут out-of-bag выборками.

В голосовании участвуют 4 дерева, построенных по этим 4 out-of-bag выборкам. Мы предъявляем наше наблюдение каждому дереву, оно проверяет наблюдение на соответствие своим правилам классификации, вычисляет листовые вероятности классов и соответствующий класс. Например, деревья классифицировали наблюдение 4 так: *Остается*, *Остается*, *Остается* и *Уходит*. В итоге побеждает класс *Остается*. Случайный лес ошибочно относит наблюдение 4 к классу *Остается*.

В итоге мы подсчитываем количество таких неверно классифицированных наблюдений, делим на общее количество наблюдений и получаем OOB ошибку для классификации.

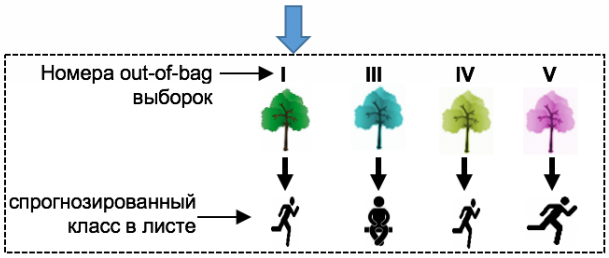
$$ER^{OOB} = \frac{1}{n}\sum_{i=1}^{n} 1\left(\hat{Y}^{OOB}(X_i) \neq Y_i\right)$$

где:

Y_i – фактический класс зависимой переменной;

$\hat{Y}^{OOB}(X_i)$ – спрогнозированный класс зависимой переменной (класс, за который проголосовало большинство деревьев, построенных по выборкам, не содержащим X_i)

№	номера out-of-bag выборок, участвующих в голосовании	фактический класс	спрогнозированный класс (итог голосования деревьев, построенных по out-bag выборкам)	результат классификации
1	IV	Остается	Остается	ВЕРНО
2	I, V	Уходит	Уходит	ВЕРНО
3	IV, V	Уходит	Уходит	ВЕРНО
4	I, III, IV, V	Уходит	Остается	НЕВЕРНО



5	I, IV	Остается	Остается	ВЕРНО
6	I, V	Уходит	Уходит	ВЕРНО
7	II, III	Уходит	Уходит	ВЕРНО
8	III	Остается	Уходит	НЕВЕРНО
9	III	Остается	Остается	ВЕРНО
10	II	Уходит	Уходит	ВЕРНО

количество неверных ответов=2

Ошибка классификации = количество неверно классифицированных наблюдений/общее количество наблюдений = 2/10=0,2

Рис. 53 Вычисление OOB ошибки для задачи классификации

Теперь возьмем задачу регрессии (рис. 54). Допустим, необходимо спрогнозировать значение зависимой переменной *Оценка дохода [income]* для наблюдения 4. Для оценки качества модели используются только те деревья решений, которые строились по бутстреп-выборкам, не содержащим наблюдение 4, и затем проводится усреднение прогнозов, выданных деревьями.

Наблюдение 4, имеющее фактическое значение *45304*, отсутствует в 4 бутстреп-выборках: I, III, IV, V. Таким образом, в усреднении участвуют 4 дерева, построенных по этим 4 бутстреп-выборкам. Мы предъявляем наше наблюдение каждому дереву, оно проверяет наблюдение на соответствие своим правилам прогнозирования, вычисляет среднее значение. Допустим, деревья прогнозируют для наблюдения 4 следующие значения: *44470*, *45112*, *46790* и *47230*. На этот раз нас интересует квадрат остатка – разницы между фактическим значением зависимой переменной *45304* и ее спрогнозированным значением *45901* (результатом усреднения средних значений, вычисленных деревьями).

В итоге по каждому наблюдению вычисляем квадрат остатка, суммируем и полученную сумму квадратов остатков делим на общее количество

наблюдений. Сумма квадратов остатков, поделенная на общее количество наблюдений, становится оценкой качества случайного леса для регрессии. Ее еще называют среднеквадратичной ошибкой по методу ООВ или ООВ ошибкой для регрессии.

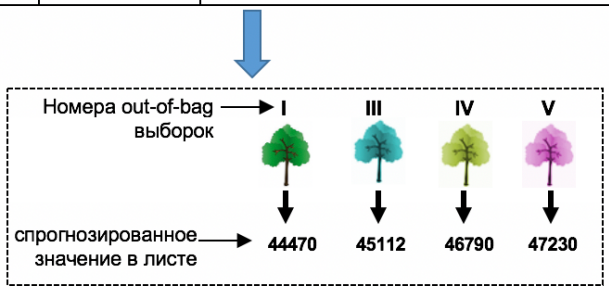
$$MSE^{OOB} = \frac{1}{n}\sum_{i=1}^{n}\left(\hat{Y}^{OOB}(X_i) - Y_i\right)^2$$

где:

Y_i – фактическое значение зависимой переменной;

$\hat{Y}^{OOB}(X_i)$ – спрогнозированное значение зависимой переменной (результат усреднения средних значений, которые вычислили деревья, построенные по выборкам, не содержащим X_i)

№	номера out-of-bag выборок, участвующих в прогнозе	фактическое значение	спрогнозированное значение (результат усреднения прогнозов деревьев, построенных по out-bag выборкам)	квадрат остатка (факт. знач. – спрогн. знач.)²
1	IV	50451	50037	171396
2	I, V	52700	52127	328329
3	IV, V	32704	32028	456976
4	I, III, IV, V	45304	45901	356409



5	I, IV	29518	29067	203401
6	I, V	29508	29029	229441
7	II, III	24018	23501	267289
8	III	26369	26938	323761
9	III	26109	26905	633616
10	II	19369	19857	238144

сумма квадратов остатков=3208762

Среднеквадратичная ошибка = сумма квадратов остатков/общее количество наблюдений = 3208762/10=320876,2

Рис. 54 Вычисление ООВ ошибки для задачи регрессии

На практике оценка ООВ ошибки является адекватной, когда количество деревьев в ансамбле достаточно велико. В ситуации, когда вы используете мало деревьев, то есть мало бутстреп-выборок, высока вероятность того, что наблюдение встретится во всех бутстреп-выборках (иными словами, ни разу не выпадет из бутстреп-выборок) и для него не будет получена ООВ оценка. Необходимо настраивать качество

модели, увеличивая количество деревьев до достаточно большого числа, пока OOB ошибка не перестанет уменьшаться.

В питоновской библиотеке `scikit-learn` случайный лес можно построить с помощью классов `RandomForestClassifier` и `RandomForestRegressor`. Для экземпляра класса `RandomForestClassifier` по умолчанию вычисляется обычная правильность, а для экземпляра класса `RandomForestRegressor` – обычный R-квадрат. Значения (классы) зависимой переменной и вероятности классов вычисляются также обычным способом с помощью методов `predict` и `predict_proba` соответственно. Однако, если задать значение параметра `oob_score = True`, то:

- для экземпляра класса `RandomForestClassifier` в атрибут `oob_score_` будет записана правильность, вычисленная по методу OOB, а в атрибут `oob_decision_function_` – вероятности классов, вычисленные по методу OOB;
- для экземпляра класса `RandomForestRegressor` в атрибут `oob_score_` будет записан R-квадрат, вычисленный по методу OOB, а в атрибут `oob_prediction_` – спрогнозированные значения, вычисленные по методу OOB.

Кроме того, с помощью атрибута `estimators_` можно получить листовые вероятности классов по каждому дереву.

Теперь давайте подробнее рассмотрим параметры и гиперпараметры классов `RandomForestClassifier` и `RandomForestRegressor`.

Параметр/Гиперпараметр	Предназначение
`estimators`	Задает количество деревьев в ансамбле. Значение по умолчанию равно 10. В `scikit-learn 0.22` будет увеличено до 100.
`criterion`	Задает критерий расщепления узлов. Для экземпляра класса `RandomForestClassifier` по умолчанию используется мера Джини (`'gini'`), также можно выбрать критерий энтропии (`'entropy'`). Для экземпляра класса `RandomForestRegressor` по умолчанию используется среднеквадратичная ошибка (`'mse'`), также можно использовать среднюю абсолютную ошибку (`'mae'`).
`max_features`	Задает m – количество случайно отбираемых предикторов для разбиения. Значение `'auto'`, использующееся по умолчанию, для экземпляра класса `RandomForestClassifier` равно $\sqrt{M}$, для экземпляра класса `RandomForestRegressor` равно M, где M – общее количество предикторов в наборе. `None` – в каждом разбиении могут участвовать все предикторы набора данных, и в отбор предикторов не будет привнесена случайность (впрочем, остается случайность, обусловленная бутстрепом). `'sqrt'` – в каждом разбиении участвует $\sqrt{M}$ предикторов.

max_depth	Задает глубину деревьев в случайном лесе. По умолчанию установлено значение None, то есть разбиения продолжаются до тех пор, пока все терминальные узлы не станут чистыми (ограничением здесь служит лишь значение min_samples_leaf).
min_samples_split	Задает минимальное количество наблюдений, необходимое для разбиения внутреннего узла. Значение по умолчанию равно 2.
min_samples_leaf	Задает минимальное количество наблюдений в терминальном узле. Значение по умолчанию равно 1.
max_leaf_nodes	Задает максимальное количество терминальных узлов. По умолчанию используется значение None, деревья строятся по принципу «лучшие узлы рассматриваются раньше». Под лучшими узлами понимаются те, которые дают наибольшее относительное уменьшение неоднородности. Если задано значение None, то количество терминальных узлов не ограничено, в противном случае рост дерева останавливается по достижении количества узлов, равного значению max_leaf_nodes.
bootstrap	Задает генерирование бутстреп-выборок для построения случайного леса. По умолчанию используется значение True.
oob_score	Задает использование out-of-bag-выборок для оценки качества модели на обучающей выборке. По умолчанию используется значение False.
n_jobs	Задает количество ядер процессора, используемых для вычислений. Значение по умолчанию равно 1. Вы можете установить n_jobs=-1, чтобы использовать все ядра вашего процессора
random_state	Задает стартовое значение генератора случайных чисел для получения воспроизводимых результатов. По умолчанию задано значение None, то есть используется экземпляр класса RandomState.
verbose	Задает детализацию процесса построения модели. По умолчанию используется значение 0.
warm_start	Задает «теплый старт». Если задано значение True, алгоритм использует модель, полученную в результате предыдущего вызова, и добавляет большее количество деревьев в ансамбль, в противном случае строится совершенно новая модель. Это очень удобно, когда необходимо посмотреть, как меняется OOB-ошибка в зависимости от роста количества деревьев в ансамбле.
class_weight	Задает веса классов (используется только для экземпляра класса RandomForestClassifier). В качестве значения могут выступать словарь, список из словарей, значение 'balanced', 'balanced_subsample' или None. По умолчанию используется значение None. Вес класса берется из словаря {class_label: weight}. Если словарь не задан, для всех классов устанавливается вес 1. Для мультиклассовых задач список словарей организован в том же порядке, в каком следуют столбцы зависимой переменной y.

	Режим `'balanced'` использует значения зависимой переменной y, чтобы автоматически настроить веса, обратно пропорциональные частотам встречаемости класса, то есть `n_samples / (n_classes * np.bincount(y))`. Режим `'balanced_subsample'` идентичен режиму `'balanced'`, за исключением того, что веса вычисляются для каждой бутстреп-выборки, по которой строилось дерево

Главный гиперпараметр случайного леса – это количество деревьев в ансамбле `n_estimators`. Как правило, большее количество деревьев дает лучший результат. Следует отметить, что если при увеличении числа деревьев улучшения качества не происходит (или даже наблюдается уменьшение качества прогноза), то это может говорить о плохом качестве выборки, о присутствии значительного шума в данных. Подобное явление также часто наблюдается на небольших выборках. Количество деревьев в ансамбле, необходимое для хорошего качества модели, возрастает с числом предикторов и ростом объема данных. Помните, что с ростом количества деревьев требуется больше памяти и больше времени для обучения.

Наилучшим способом определить оптимальное число деревьев является сравнение метрик качества для нескольких вариантов (например, последовательно построив лес из 100, 200, 300, 400 и 500 деревьев) на контрольной выборке. Также можно воспользоваться перекрестной проверкой и сравнить метрики качества, усредненные по контрольным блокам. Обычно по мере увеличения числа деревьев качество модели на обучающей выборке увеличивается (можно добиться даже 100%-ной правильности или значения AUC, равного 1), а на контрольной выборке качество увеличивается и затем стабилизируется на определенном значении.

Итак, большее количество деревьев обычно дает лучшее качество, однако вы должны регулировать стоимость улучшения качества с вычислительной точки зрения. Например, мы можем использовать ансамбль из 300 деревьев и получить на контрольной выборке AUC 0,811, затем дополнительно натренировать еще 100 деревьев и получить AUC 0,83, в этом случае идея увеличить количество деревьев имеет смысл, если же дополнительная тренировка 100 деревьев дает в итоге AUC 0,812, то целесообразность такого улучшения будет сомнительной. Определив такое количество деревьев в ансамбле, которое дает приемлемое качество модели на контрольной выборке и обучается за приемлемое время, мы задаем количество случайно отбираемых предикторов. Лео Брейман называл этот гиперпараметр `mtry`. Разработчики библиотеки `scikit-learn` его переименовали в `max_features`. Поясним, как работает `max_features`. Количество случайно отбираемых предикторов, равное 1, означает, что при разбиении будет выполнен поиск точек расщепления для одного случайно выбранного признака. Если количество случайно отбираемых предикторов задать

равным общему количеству предикторов в наборе данных, это будет обозначать, что в каждом разбиении смогут участвовать все предикторы набора данных, в отбор признаков не будет привнесена случайность (останется лишь случайность, обусловленная бутстрепом). При таком варианте все деревья в случайном лесе будут в большей степени схожи между собой, нежели при более низких значениях `max_features`, повысится декоррелированность деревьев и качество модели может ухудшиться. На практике подбирают значения `max_features`, которые составляют примерно 20–40% от общего числа предикторов. Эти значения были сформулированы на основе дополненных правил Лео Бреймана, приведенных на рис. 55.

Для классификации	**Для регрессии**
$0{,}5 \times \sqrt{M}$	$0{,}5 \times (M/3)$
$\sqrt{M}$	$M/3$
$2 \times \sqrt{M}$	$2 \times (M/3)$
	M*
$2 \times log_2 M + 1$ или $3 \times log_2 M + 1$**	

M – общее число предикторов в наборе

Примечания:
Полученные значения `max_features` округляем в меньшую сторону
* Правило было сформулировано по итогам экспериментов уже после смерти Лео Бреймана
** Данное правило является рекомендацией Лео Бреймана в тех случаях, когда используется one-hot-кодирование или много переменных со слабой прогнозной силой

Рис. 55 Правила для определения оптимального количества случайно отбираемых предикторов

Если есть несколько переменных с сильной прогнозной силой, меньшие значения `max_features` могут дать лучшее качество. Если данные содержат много переменных со слабой прогнозной силой, нужно попробовать б*о*льшие значения `max_features`. При построении случайного леса с помощью классов `RandomForestClassifier` и `RandomForestRegressor` библиотеки `scikit-learn` также помимо правил Бреймана необходимо попробовать б*о*льшие значения `max_features`, поскольку выбор небольшого значения `max_features` уменьшит вероятность отбора полезного предиктора. По мнению ряда специалистов[14], на практике варьирование значений `max_features` не оказывает существенного влияния на качество модели.

[14] *D. Richard Cutler, Thomas C. Edwards, Jr., Karen H. Beard, Adele Cutler, Kyle T. Hess, Jacob Gibson, and Joshua J. Lawler*. 2007. Random forests for classification in ecology. Ecology 88:2783–2792.

Обрезка деревьев, как правило, требует корректировки `max_features`.
В итоге при фиксированном количестве деревьев необходимо выбрать такое количество случайно отбираемых предикторов, которое дает максимальное качество модели на контрольной выборке.
Еще один гиперпараметр, который стоит учитывать при построении модели случайного леса – это глубина деревьев. Как правило, при увеличении глубины возрастает время обучения, но при этом увеличивается качество модели на обучающей и контрольной выборках. В ряде случаев, например, при работе с зашумленными данными, построение деревьев с максимальной глубиной не дает хорошего качества модели. Это обусловлено тем, что при очень высоких значениях глубины деревья становятся излишне сложными и чувствительными к случайным возмущениям данных. В случае большого количества шумовых объектов рандомизация и усреднение по ансамблю не позволяют скомпенсировать возникшее переобучение. В таких случаях нужно выбрать меньшее значение глубины. Вместе с тем нужно избегать и слишком низкого значения глубины, при котором деревья не смогут в достаточной степени обучиться, что приведет к недообучению.
Кроме того, максимальную глубину можно настроить косвенно, изменив количество наблюдений в терминальном узле. Соответствующий гиперпараметр в классах `RandomForestClassifier` и `RandomForestRegressor` питоновской библиотеки `scikit-learn` называется `min_samples_leaf`.
Обратите внимание, что в реализациях случайного леса, где вместо усреднения прогнозов используется усреднение листовых вероятностей классов, увеличение размера терминального узла может улучшить качество. Это обусловлено тем, что при выращивании полных деревьев мы можем получить чистые листы с одним наблюдением и наши оценки листовых вероятностей будут основаны на одном наблюдении, таким образом, усреднение листовых вероятностей классов может привести к получению не совсем корректных итоговых вероятностей классов[15].
Давайте построим модель случайного леса, уменьшив глубину деревьев. Мы будем вычислять AUC по обычному методу и методу OOB.

In[92]:
```
# импортируем класс RandomForestClassifier
from sklearn.ensemble import RandomForestClassifier
# создаем экземпляр класса RandomForestClassifier
forest = RandomForestClassifier(n_estimators=800, max_depth=17,
                                random_state=152, n_jobs=-1)
# строим модель
forest.fit(X_train, y_train)
# оцениваем дискриминирующую способность
# модели случайного леса
print("AUC на обучающей выборке по обычному методу: {:.3f}".format(
    roc_auc_score(y_train, forest.predict_proba(X_train)[:, 1])))
print("AUC на обучающей выборке по методу OOB: {:.3f}".format(
```

[15] *Malley, J. D., Kruppa, J., Dasgupta, A., Malley, K. G., Ziegler, A.* 2012. Probability machines: consistent probability estimation using nonparametric learning machines. Methods Inf Med, vol. 51, no. 1, 74. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3250568/

```
    roc_auc_score(y_train, forest.oob_decision_function_[:, 1])))
print("AUC на контрольной выборке: {:.3f}".format(
    roc_auc_score(y_test, forest.predict_proba(X_test)[:, 1])))
```

Out[92]:

```
AUC на обучающей выборке по обычному методу: 0.895
AUC на обучающей выборке по методу OOB: 0.761
AUC на контрольной выборке: 0.766
```

Теперь вместо случайного леса построим модель бустинга.
Идея бустинга заключается в том, чтобы строить композицию из базовых алгоритмов последовательно: каждый новый базовый алгоритм строится с использованием информации предыдущих алгоритмов и стремится исправить их ошибки. В качестве базового алгоритма обычно берут дерево решений из-за его способности учитывать сложные нелинейные зависимости, дисперсии – чувствительности к изменению выборки наблюдений и количества используемых признаков (будем получать сильно различающиеся деревья и от разнообразия композиция выиграет) и по причине того, что деревья – слабые алгоритмы (в бустинге не имеет смысла использовать сильные алгоритмы, они будут быстро переобучаться, выдавая идеальное значение функции потерь на обучающей выборке, и следующие итерации будут бесполезны).
Давайте рассмотрим идею бустинга на примере задачи регрессии. Допустим, нам нужно спрогнозировать ежемесячный заработок (в долларах) в зависимости от того, владеет ли клиент домом, есть ли у него собственный автомобиль, есть ли у него семья.

	Есть собственный дом?	Есть собственный автомобиль?	Есть семья?	Ежемесячный заработок
1	Да	Да	Да	10000
2	Нет	Нет	Нет	25
3	Да	Нет	Нет	5000

Инициализируем нашу композицию «наивной» константной моделью, просто средним значением зависимой переменной. Для каждого наблюдения спрогнозированное значение зависимой переменной будет представлять собой среднее значение зависимой переменной $\frac{10000+25+5000}{3} = 5008{,}3$.

	Есть собственный дом?	Есть собственный автомобиль?	Есть семья?	Ежемесячный заработок
1	Да	Да	Да	10000
2	Нет	Нет	Нет	25
3	Да	Нет	Нет	5000

Константный прогноз
5008,3
5008,3
5008,3

Затем мы вычисляем остатки (разности между фактическими значениями зависимой переменной и спрогнозированными значениями зависимой переменной) и строим базовый алгоритм, который будет

представлять собой неглубокое регрессионное дерево решений, пытающееся предсказать эти остатки.

	Есть собственный дом?	Есть собственный автомобиль?	Есть семья?	Остаток
1	Да	Да	Да	4991,7 (10000 - 5008,3)
2	Нет	Нет	Нет	-4983,3 (25 – 5008,3)
3	Да	Нет	Нет	-8,3 (5000 – 5008,3)

Зависимая переменная, которую предсказываем

Этот процесс может быть итеративным: вычисляем остатки последнего дерева и обучаем следующее дерево, которое пытается исправить ошибки предыдущих деревьев.
Допустим, мы построили два дерева решений и для наблюдения 1 (Да, Да, Да) первое дерево дает прогноз 3510,3, а второе дерево – прогноз 2120,4, тогда итоговый прогноз для наблюдения 1 будет равен 5008,3 + 3510,3 + 2120,4 = 10639.

Описание бустинга приведено ниже:

Алгоритм 1. *Алгоритм простого бустинга для регрессии*

пусть h_0 – это «наивная» константная модель
пусть F_0 – это композиция, состоящая только из h_0
для $m = 1,2,\dots,M$ повторять:
    для каждой пары (x_i, y_i) в обучающей выборке вычислить остаток $R\big(y_i, F_{m-1}(x_i)\big) = y_i - F_{m-1}(x_i)$
    обучить базовый алгоритм h_m по остаткам
    добавить h_m в композицию: $F_m(x) = F_{m-1}(x) + h_m(x)$
вернуть композицию F_M

В данном случае мы оптимизировали композицию в соответствии с определенной функцией потерь: мы пытались минимизировать квадратичную ошибку, которая имеет смысл для задачи регрессии. А что если бы мы хотели минимизировать другую функцию потерь, например, логистическую функцию потерь или кросс-энтропию для задачи классификации или другую функцию потерь для задачи регрессии (например, абсолютную ошибку)? Здесь нам на помощь приходит градиентный бустинг.
Давайте вспомним, что такое «градиент». Градиент – это вектор, который показывает направление наибольшего возрастания функции. Соответственно антиградиент – это вектор, который показывает направление наибольшего убывания функции. Проекциями этого вектора на оси координат будут значения частных производных этой функции в соответствующей точке.
Обычная производная дает нам скорость изменения функции одной переменной, обычно x. Например, dF/dx сообщает нам, как изменяется функция F при изменении x. Но если речь идет о функции нескольких переменных, таких как x и y, она будет иметь несколько частных производных: значение функции будет меняться, когда мы меняем x (dF/dx) и когда мы меняем y (dF/dy).

Мы можем представить эти скорости изменения в виде вектора с одним компонентом для каждой частной производной. Таким образом, функция 3 переменных будет иметь градиент с тремя компонентами:
$F(x)$ имеет одну переменную и одну производную: dF/dx;
$F(x, y, z)$ имеет три переменные и три частные производные: $(dF/dx, dF/dy, dF/dz)$.
У градиента функции нескольких переменных будет по одному компоненту для каждого направления. И точно так же, как и обычная производная, градиент указывает направление наибольшего возрастания. Однако теперь, когда у нас есть несколько направлений (x, y и z), направление наибольшего возрастания уже не просто движение «вперед» или «назад» вдоль оси x, как это происходит с функциями одной переменной. Когда у нас есть две переменные, то наш двухкомпонентный градиент может указать любое направление на плоскости. Аналогично, с тремя переменными, градиент может указать такое направление в трехмерном пространстве, которое дает наибольшее возрастание нашей функции.
С понятием градиента связан метод градиентного спуска. Основная идея градиентного спуска состоит в том, чтобы двигаться к локальному минимуму в направлении наиболее быстрого убывания функции, которое определяется антиградиентом. В ходе градиентного спуска мы итеративно применяем следующее правило обновления:

$$x = x - \eta \nabla_{Loss}(x),$$

где $\nabla_{Loss}(x)$ – это градиент функции потерь, который пытаемся минимизировать, а η – *размер шага* (*темп обучения* или *learning rate*).
Мы выбираем каким-либо способом начальную точку, вычисляем в ней градиент рассматриваемой функции и делаем небольшой шаг в обратном, антиградиентном направлении. В результате мы приходим в точку, в которой значение функции будет меньше первоначального. В новой точке повторяем процедуру: снова вычислим градиент функции и делаем шаг в обратном направлении. Продолжая этот процесс, мы будем двигаться в сторону убывания функции.
Если шаг будет слишком большой, мы, возможно, не достигнем локального минимума, поскольку можем получить расходящийся итерационный процесс, как показано в левой части рис. 56. Если же шаг будет очень небольшим, мы постепенно достигнем локального минимума, но это может занять гораздо больше времени, как это видно в правой части рис. 56.

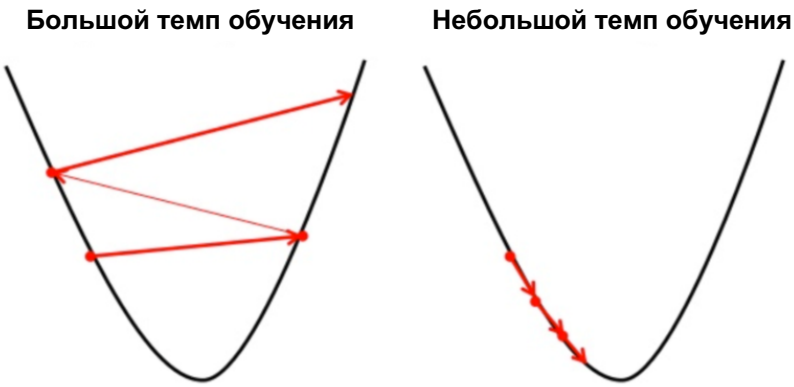


Рис. 56 Высокий и низкий темпы обучения

Так вот процедуру бустинга, описанную в алгоритме 1, можно представить в виде градиентного спуска. Вы спросите, как же связан градиентный спуск с алгоритмом бустинга? Ответ заключается в том, что отрицательный градиент (антиградиент) нашей квадратичной функции потерь, как раз и является остатком (умноженным на 2):

$$Loss(y_i, \hat{y}) = (y_i - \hat{y})^2 \qquad -\nabla_{Loss}(\hat{y}) = 2 \cdot (y_i - \hat{y})$$

Таким образом, алгоритм бустинга можно представить в виде градиентного спуска, оптимизирующего квадратичную функцию потерь, поскольку на каждой итерации он добавляет базовый алгоритм, который пытается предсказать антиградиент функции потерь. На рис. 57 показано, как алгоритм градиентного бустинга добавляет базовые алгоритмы в композицию, постепенно минимизируя функцию потерь (в нашем случае – квадратичную функцию потерь).

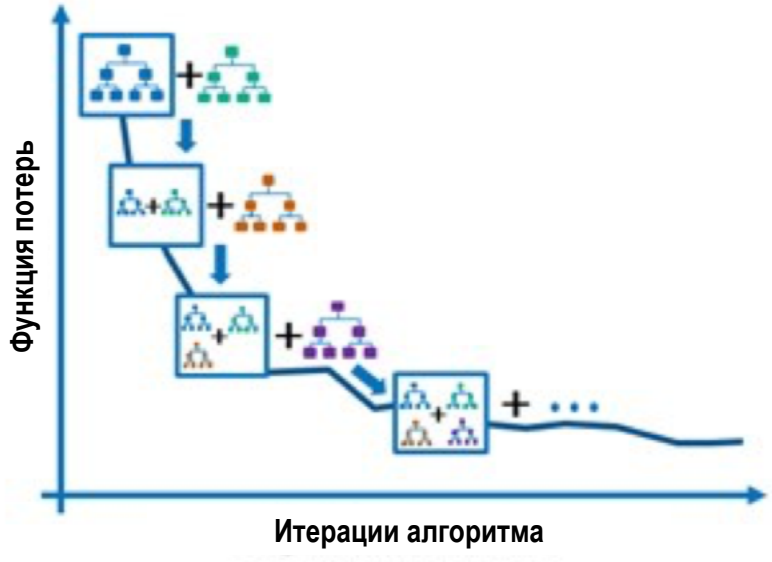


Рис. 57 Добавление базовых алгоритмов в композицию для постепенного уменьшения функции потерь

Итак, как быть, если мы хотим оптимизировать какую-нибудь другую функцию потерь? В таком случае мы делаем следующее: для каждого

наблюдения обучающей выборки вычисляем антиградиент нужной функции потерь, затем обучаем базовый алгоритм, который пытается предсказать этот антиградиент. Отличие от алгоритма 1 заключается в том, что эти антиградиенты больше не интерпретируются как остатки, теперь мы их называем *псевдо-остатками* (*pseudo-residuals*).

Алгоритм 2. *Алгоритм градиентного бустинга*

пусть F_0 – это «наивная» константная модель
для $m = 1, 2, \dots, M$ повторять:
    для каждой пары (x_i, y_i) в обучающей выборке вычислить *псевдо-остаток* $R\big(y_i, F_{m-1}(x_i)\big)$ = антиградиент функции потерь
    обучить базовый алгоритм h_m по псевдо-остаткам
    добавить h_m в композицию: $F_m(x) = F_{m-1}(x) + \eta \cdot h_m(x)$
вернуть композицию F_M

Вернемся к примеру, когда мы вычисляли ежемесячный заработок. Допустим, мы задали темп обучения равным 0,5, в качестве константной модели взяли среднее (5008,3), построили два дерева решений и для наблюдения 1 (Да, Да, Да) первое дерево дало прогноз 4150,8, а второе дерево − прогноз 3560,7, тогда итоговый прогноз для наблюдения 1 будет равен 5008,3 + 0,5 * 4150,8 + 0,5 * 3560,7 = 8864,05.
Обратите внимание на несколько важных моментов.
Если в качестве базового алгоритма выбрано дерево решений, то градиентный бустинг всегда будет использовать регрессионные деревья, даже если мы решаем задачу классификации. Для классификации мы преобразуем полученные значения в вероятности, используя сигмоиду или софтмакс.
Чаще всего в бустинге используются деревья довольно небольшой глубины, всего с 3-6 уровнями ниже корневого узла. Такие деревья характеризуются высоким смещением и более низкой дисперсией в сравнении с глубокими деревьями. Это делается, чтобы ослабить базовые алгоритмы: благодаря построению неглубоких слабых деревьев мы медленно минимизируем нашу функцию потерь.
Темп обучения еще сильнее замедляет этот процесс, позволяя создать более сложную композицию для лучшей минимизации функции потерь. Обычно маленький темп обучения обеспечивает более высокое качество модели, однако меньшее значение темпа обучения требует большего количества итераций для получения хорошего качества модели. На практике строят график изменения функции потерь по мере выполнения градиентного спуска. По оси абсцисс откладываем количество итераций, по оси ординат откладываем значения функции потерь или оптимизируемой метрики. Пробуют разные темпы обучения и смотрят получившиеся графики. График, приведенный ниже, иллюстрирует разницу между хорошим и плохим темпом обучения.

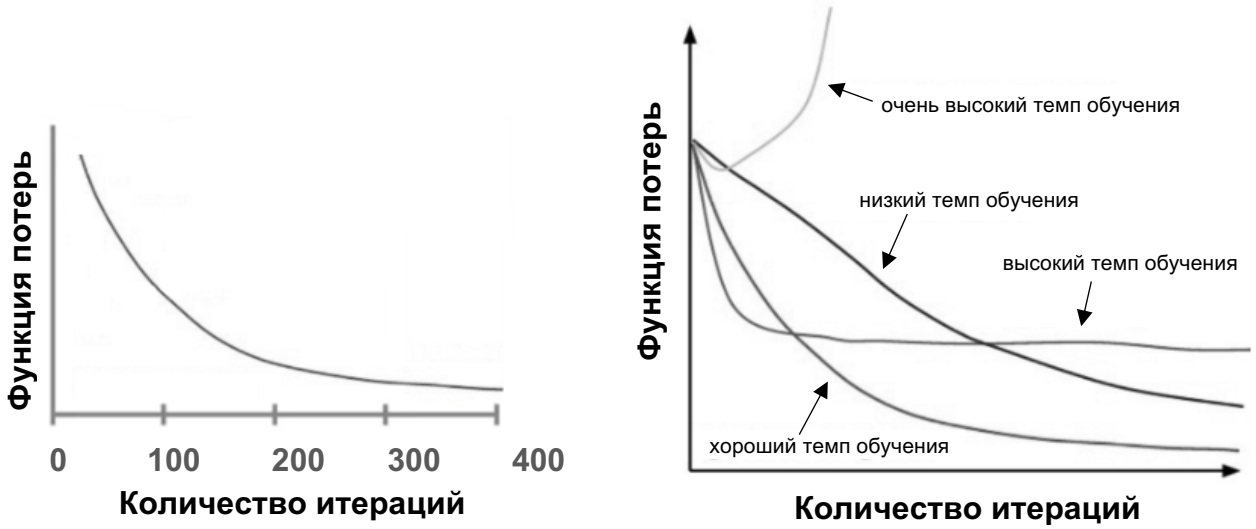


Рис. 58 График зависимости функции потерь от темпа обучения

При хорошем темпе обучения функция потерь должна уменьшаться после каждой итерации.

Бутстреп-выборки в градиентном бустинге не создаются – вместо этого каждое дерево строится по исходному набору данных, модифицированному определенным образом. Чаще всего для построения каждого отдельного дерева используется случайная подвыборка данных и случайное подпространство признаков, таким образом, строя деревья, сильно отличающиеся друг от друга за счет привнесения случайности, мы усиливаем композицию.

В июле 2017 года компания Yandex представила CatBoost, новую библиотеку градиентного бустинга. Двумя критически важными алгоритмическими новшествами, внедренными в CatBoost, стали реализация динамического бустинга, представляющая собой альтернативу классическому алгоритму бустинга, а также инновационный алгоритм обработки категориальных признаков. Оба нововведения предназначены для борьбы со смещением прогноза, который вызван утечкой информации о зависимой переменной – проблемой, актуальной для всех существующих в настоящее время реализаций алгоритма градиентного бустинга.

Как мы уже знаем, в градиентном бустинге мы последовательно строим деревья. Построили дерево, вычисляем значения в листьях. Когда мы вычисляем значение в листе (этим значением будет среднее значение градиента по всем наблюдениям, попавшим в данный лист), то оценку делаем по тем же самым наблюдениям, по которым строилась модель. Данный факт в конечном счете приведет к смещению прогнозов обученной модели. Эту проблему можно определить как особый тип утечки информации о зависимой переменной.

Аналогичная проблема возникает в стандартных алгоритмах предварительной обработки категориальных признаков. Одним из

наиболее эффективных способов использования категориальных признаков в градиентном бустинге является преобразование категориальных признаков в количественные, используя статистики на основе зависимой переменной (например, категории предиктора можно заменить средними значениями зависимой переменной в этих категориях). Статистика на основе зависимой переменной представляет собой простую статистическую модель, и она также может вызвать утечку информации о зависимой переменной (вспомним про редкие категории) и привести к смещению прогноза.

Авторы CatBoost предложили принцип исторической последовательности для решения обеих проблем. Используя его, можно получить новый алгоритм обработки категориальных признаков (вычисление статистик зависимой переменной по «прошлому») и динамический бустинг, модификацию стандартного алгоритма градиентного бустинга, которая позволяет избежать утечки информации о зависимой переменной.

При вычислении статистики зависимой переменной для каждого наблюдения CatBoost опираются лишь на имеющуюся историю. На обучающем наборе выполняем случайную перестановку и теперь для интересующего наблюдения считаем статистику зависимой переменной на основе всех наблюдений, которые в данной перестановке расположились выше нашего наблюдения, т.е. встретились «раньше» нашего наблюдения и по отношению к нему являются «историей». Обратите внимание, если использовать только одну случайную перестановку, то у предшествующих обучающих наблюдений статистика зависимой переменной будет иметь гораздо большую дисперсию, чем у последующих обучающих наблюдений. Поэтому CatBoost выполняет несколько перестановок.

При вычислении значений в листьях можно использовать те же самые случайные перестановки обучающего набора, которые применялись для вычисления статистик зависимой переменной. Для каждого наблюдения мы будем вычислять значение в листе как среднее значение градиента по всем наблюдениям, попавшим в лист раньше рассматриваемого наблюдения в данной случайной перестановке.

Еще одна особенность CatBoost заключается в том, что здесь используются бинарные забывчивые деревья решений (их еще называют «небрежными деревьями» или oblivious trees), в которых для всех узлов определенного уровня применяется одно и то же правило разбиения. По мнению авторов CatBoost, такие деревья сбалансированы, в меньшей степени склонны к переобучению и позволяют быстрее вычислить прогнозы.

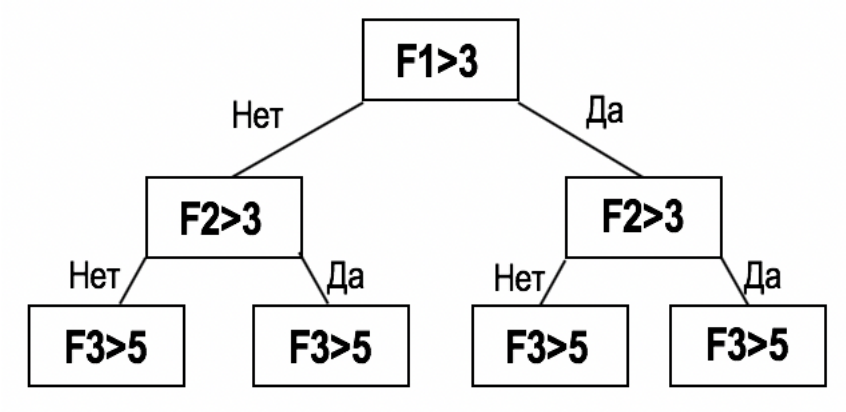


Рис. 59 Пример забывчивого дерева решений

Теперь давайте подробнее разберем процесс преобразования категориальных признаков в количественные. Это происходит перед выбором точки расщепления в процессе создания дерева. Преобразования основываются на различных статистиках для комбинаций категориальных признаков с количественными и друг с другом.

Метод преобразования категориальных признаков в числовые обычно включает следующие этапы:

1) Случайная перестановка набора объектов.
2) Превращение значения метки (зависимой переменной) из числа с плавающей запятой в целочисленное значение.

Сам метод зависит от решаемой задачи машинного обучения, которая определяется выбором функции потерь.

Задача	Способ выполнения преобразования
Регрессия	Выполняется бинаризация значений зависимой переменной. Режим преобразования и количество интервалов (k+1) предварительно задаются в качестве параметров. Всем значениям внутри одного интервала присваивается метка класса – целое число в диапазоне [0; k], определяемое по формуле *<bucket ID – 1>*.
Двухклассовая классификация	Возможные значения зависимой переменной: 0 (объект не принадлежит к положительному классу) или 1 (объект принадлежит к положительному классу).
Многоклассовая классификация	Значения зависимой переменной являются целочисленными идентификаторами классов (нумерация начинается с 0).

3) Преобразование категориальных признаков в числовые. Метод определяется предварительно заданными параметрами.

Метод	Формула
Borders	Расчет для i-го диапазона ($i \in [0; k-1]$): $ctr_i = \frac{countInClass+prior}{totalCount+1}$, где *countInClass* – сколько раз значение зависимой переменной превышает *i* для наблюдений с данным значением категориального признака. Учитываются только те наблюдения, для которых это значение уже рассчитано (расчеты выполняются в порядке следования объектов после перемешивания). *totalCount* – общее количество наблюдений (до текущего), для которых значение признака соответствует значению для текущего наблюдения. *prior* – исходное значение числителя, определенное начальными параметрами.
Buckets	Расчет для *i*-го диапазона ($i \in [0; k]$, создается $k+1$ диапазонов): $ctr_i = \frac{countInClass+prior}{totalCount+1}$, где *countInClass* – сколько раз значение зависимой переменной равно *i* для наблюдений с данным значением категориального признака. Учитываются только те наблюдения, для которых это значение уже рассчитано (расчеты выполняются в порядке следования наблюдений после перемешивания). *totalCount* – общее количество наблюдений, для которых значение признака соответствует значению для текущего наблюдения. *prior* – исходное значение числителя, определенное начальными параметрами.
BinarizedTargetMeanValue	Расчет выполняется по формуле: $ctr_i = \frac{countInClass+prior}{totalCount+1}$, где *countInClass* – отношение суммы целочисленных значений зависимой переменной для этого категориального признака к максимальному значению зависимой переменной (k). *totalCount* – общее количество наблюдений, для которых значение признака соответствует значению для текущего наблюдения. *prior* – исходное значение числителя, определенное начальными параметрами.
Counter	Расчет для обучающей выборки выполняется по формуле: $ctr_i = \frac{curCount+prior}{maxCount+1}$, где

	curCount – общее количество наблюдений с данным значением категориального признака в обучающей выборке. *maxCount* – количество наблюдений в обучающей выборке с наиболее частым значением категориального признака. *prior* – исходное значение числителя, определенное начальными параметрами. Расчет для тестовой выборки выполняется по формуле: $ctr_i = \frac{curCount+prior}{maxCount+1}$, где *curCount* – вычисляется одним из методов: - FullTest – сумма общего количества наблюдений с данным значением категориального признака в обучающей и в тестовой выборке. - SkipTest – общее число наблюдений с данным значением категориального признака в обучающей выборке. *maxCount* – количество наблюдений с наиболее частым значением категориального признака в комбинации выборок в зависимости от метода расчета: - FullTest – обучающая и тестовая выборки. - SkipTest – обучающая выборка. *prior* – исходное значение числителя, определенное начальными параметрами. Примечание: ctr не зависит от значения зависимой переменной.

В результате каждому категориальному признаку или комбинации признаков сопоставляется количественная переменная.

CatBoost позволяет агрегировать несколько признаков. Предположим, что объекты обучающей выборки относятся к категориям двух категориальных признаков: музыкальный жанр («рок», «инди») и стиль музыки («танцевальная», «классическая»). Эти признаки могут встречаться в различных комбинациях. Библиотека CatBoost позволяет создать признак, являющийся комбинациями этих двух (со значениями «танцевальный рок», «классический рок», «танцевальная инди» и «классическая инди»). Комбинировать можно любое количество признаков.

Теперь рассмотрим, как происходит преобразование категориальных признаков в количественные при решении задачи классификации. Оно происходит в три этапа.

1) CatBoost принимает на вход набор значений признаков и значений зависимой переменной (т.е. моделируемых значений). Рис. 60 ниже иллюстрирует, как выглядят результаты на данном этапе.

	Признак$_1$	Признак$_2$	Признак$_3$	Зависимая переменная
1	2	40	Рок	1
2	3	55	Инди	0
3	5	34	Поп	1
4	2	45	Рок	0
5	4	53	Рок	0
6	2	48	Инди	1
7	5	42	Рок	1
…	…	…	…	…

Рис. 60 Исходный набор значений признаков и зависимой переменной

2) Наблюдения несколько раз случайным образом перемешиваются, и создается несколько случайных перестановок набора данных. Рис. 61 показывает, как выглядят результаты на данном этапе.

	Признак$_1$	Признак$_2$	Признак$_3$	Зависимая переменная
1	4	53	Рок	0
2	3	55	Инди	0
3	2	40	Рок	1
4	5	42	Рок	1
5	5	34	Поп	1
6	2	48	Инди	1
7	2	45	Рок	0
…	…	…	…	…

Рис. 61 Набор значений признаков и зависимой переменной после случайной перестановки

3) Все категориальные признаки преобразовываются в количественные по формуле:

$$avg\_target = \frac{countInClass + prior}{totalCount + 1}$$

countInClass – сколько раз значение зависимой переменной равно 1 для объектов с данным значением категориального признака. Учитываются только наблюдения, для которых это значение уже рассчитано (расчеты выполняются в порядке следования наблюдений после перемешивания).
prior – исходное значение числителя, определенное начальными параметрами.
totalCount – общее количество наблюдений (до текущего), для которых значение признака совпадает со значением для текущего наблюдения.
Обратите внимание, что эти значения рассчитываются для каждого наблюдения по отдельности с использованием данных предыдущих наблюдений. В примере с музыкальными жанрами $j \in [1; 3]$, допустимые значения: «Рок», «Поп» и «Инди», исходное значение числителя (prior) установлено равным 0,05.
Допустим, мы хотим вычислить значение для Признака$_3$ в наблюдении 4. Мы видим, что у нас только одно наблюдение со значением «Рок», в

котором зависимая переменная принимает значение 1 (не считая рассматриваемое наблюдение). Поэтому *countInClass* равен 1. При этом у нас два наблюдения со значением «Рок» до текущего. Поэтому *totalCount* равен 2.
На рис. 62 подробно приводятся вычисления.

	Признак$_1$	Признак$_2$	Признак$_3$	Зависимая переменная	Вычисление значений
1	4	53	Рок	0	(0 + prior) / (0 + 1) = 0,05 / 1 = 0,05
2	3	55	Инди	0	(0 + prior) / (0 + 1) = 0,05 / 1 = 0,05
3	2	40	Рок	1	(0 + prior) / (1 + 1) = 0,05 / 2 = 0,025
4	5	42	Рок	1	(1 + prior) / (2 + 1) = 1,05 / 3 = 0,35
5	5	34	Поп	1	(0 + prior) / (0 + 1) = 0,05 / 1 = 0,05
6	2	48	Инди	1	(0 + prior) / (1 + 1) = 0,05 / 2 = 0,025
7	2	45	Рок	0	(2 + prior) / (3 + 1) = 2,05 / 4 = 0,5125
...	...	...	...	...	...

Рис. 62 Вычисление значений будущей количественной переменной для категориального признака

Рис. 63 показывает, как выглядят итоговые результаты.

	Признак$_1$	Признак$_2$	Признак$_3$	Зависимая переменная
1	4	53	0,05	0
2	3	55	0,05	0
3	2	40	0,025	1
4	5	42	0,35	1
5	5	34	0,05	1
6	2	48	0,025	1
7	2	45	0,5125	0
...	...	...	...	...

Рис. 63 Замена категорий признака количественными значениями

Кроме того, CatBoost поддерживает one-hot-кодирование. Включить его можно при помощи гиперпараметра `one_hot_max_size`, с помощью него мы просто задаем целочисленное значение – количество уровней категориального признака. Все категориальные признаки с количеством уровней, равным или меньшим, чем это значение, подвергаются one-hot-кодированию.
Установить библиотеку `catboost` можно в Anaconda Prompt с помощью команды `pip install catboost`.
Кроме того, если вы хотите строить графики обучения, вы должны установить пакет `ipywidgets` и запустить специальную команду перед запуском тетрадки Jupiter:

```
$ pip install ipywidgets
$ jupyter nbextension enable --py widgetsnbextension
```

Теперь импортируем необходимые классы `CatBoostClassifier` и `Pool`.

In[93]:

```
# импортируем классы CatBoostClassifier, Pool, cv
from catboost import CatBoostClassifier, Pool, cv
```

Использовать текущие массивы признаков и меток мы не можем, зависимая переменная должна иметь тип `int`, а в массиве признаков категориальные предикторы должны быть представлены «как есть» без выполнения дамми-кодирования (разумеется, мы можем передать алгоритму массивы признаков после выполнения дамми-кодирования, однако в таком случае игнорируется возможность CatBoost эффективно обрабатывать категориальные признаки).

In[94]:
```
# подготавливаем массив меток для catboost
y_train_catboost = train['open_account_flg'].astype('int')
y_test_catboost = test['open_account_flg'].astype('int')

# подготавливаем массив признаков для catboost
X_train_catboost = train.drop('open_account_flg', axis=1)
X_test_catboost = test.drop('open_account_flg', axis=1)
```

In[95]:
```
# записываем список индексов категориальных предикторов
cat_features_idx = np.where(X_train_catboost.dtypes == 'object')[0].tolist()
# выводим этот список
cat_features_idx
```

Out[95]:
```
[0, 2, 3, 6, 8, 9]
```

Теперь давайте создадим саму модель: здесь мы можем задать значения параметров и гиперпараметров. По умолчанию для задачи классификации оптимизируется логистическая функция потерь (`'loss_function': 'Logloss'`). С помощью параметра `eval_metric` мы можем оптимизировать нашу модель с точки зрения конкретной метрики (например, с точки зрения AUC: `'eval_metric': 'AUC'`), для этого с помощью параметра `eval_set` нужно указать контрольный массив признаков и контрольный массив меток, на которых будет оптимизироваться метрика. Кроме того, с помощью параметра `custom_metric` можно справочно посмотреть, как обучается модель с точки зрения конкретной метрики. Например, мы оптимизируем нашу модель по AUC, но неплохо было бы посмотреть, как меняется F1-мера в зависимости от темпа обучения и количества итераций (`'custom_metric': 'F1'`).
Еще у нас есть интересный параметр `use_best_model`. Если задать этот параметр равным `True`, количество деревьев в итоговой модели определяется следующим образом:
1. Строим количество деревьев в соответствии с выбранными параметрами и гиперпараметрами обучения.
2. Используем проверочный набор данных для поиска итерации, на которой достигается оптимальное значение метрики, заданной с помощью параметра `eval_metric`.
3. Возвращаем модель, при этом деревья, построенные уже после найденной итерации, не будут сохранены.
По умолчанию задано значение `True`, если с помощью параметра `eval_set` задан проверочный набор, в противном случае `False`.

```
params = {
    'loss_function': 'Logloss',
    'eval_metric': 'AUC',
    'custom_metric': 'F1',
    'random_seed': 42,
    'logging_level': 'Silent',
    'use_best_model': False
}
```

Функция потерь

Оптимимизируемая метрика качества, требует, чтобы с помощью настройки `eval_set` был задан контрольный массив признаков и контрольный массив меток

Метрика качества, отслеживаемая справочно

Стартовое значение генератора случайных чисел для воспроизводимости результатов

Уровень детализации вывода, возможные значения:

- `Silent` – ничего не выводить;
- `Verbose` – выводить информацию о значении оптимизируемой метрики, прошедшего времени и ожидаемого времени до окончания обучения;
- `Info` – выводить дополнительную информацию и количество деревьев;
- `Debug` – выводить информацию для отладки.

Для большего удобства работы с массивом признаков и массивом меток библиотека `catboost` предлагает класс `Pool`. В нем хранится вся информация о наборе данных (признаки, метки, индексы категориальных признаков, веса и многое другое).

```
Pool(data,
     label=None,
     cat_features=None,
     column_description=None,
     pairs=None,
     delimiter='\t')
```

2-мерный массив признаков (список, массив NumPy, объект DataFrame, объект Series), набор данных или путь к файлу с описанием данных (строка)

1-мерный массив чисел с плавающей точкой (список, массив NumPy, объект DataFrame, объект Series)

1-мерный массив с индексами категориальных признаков (список, массив NumPy)

путь к файлу с описаниями столбцов (строка)

2-мерная матрица размерностью Nx2 с описанием пар объектов для решения задачи ранжирования

символ, разделяющий столбцы в файле с описанием данных

Итак, давайте зададим параметры и гиперпараметры по умолчанию, при этом будем оптимизировать AUC, справочно выводить информацию об F1-мере, зададим стартовое значение генератора случайных чисел для воспроизводимости результатов и откажемся от выбора наилучшей модели.

In[96]:
```
# задаем значения параметров и гиперпараметров
params = {
    'loss_function': 'Logloss',
    'eval_metric': 'AUC',
    'custom_metric': 'F1',
    'random_seed': 42,
    'logging_level': 'Silent',
    'use_best_model': False
}
```

Теперь создадим обучающий и контрольный пулы, передав массивы признаков, массивы меток и индексы категориальных признаков.

In[97]:
```
# создаем обучающий и контрольный пулы
train_pool = Pool(X_train_catboost, y_train_catboost, cat_features=cat_features_idx)
validate_pool = Pool(X_test_catboost, y_test_catboost, cat_features=cat_features_idx)
```

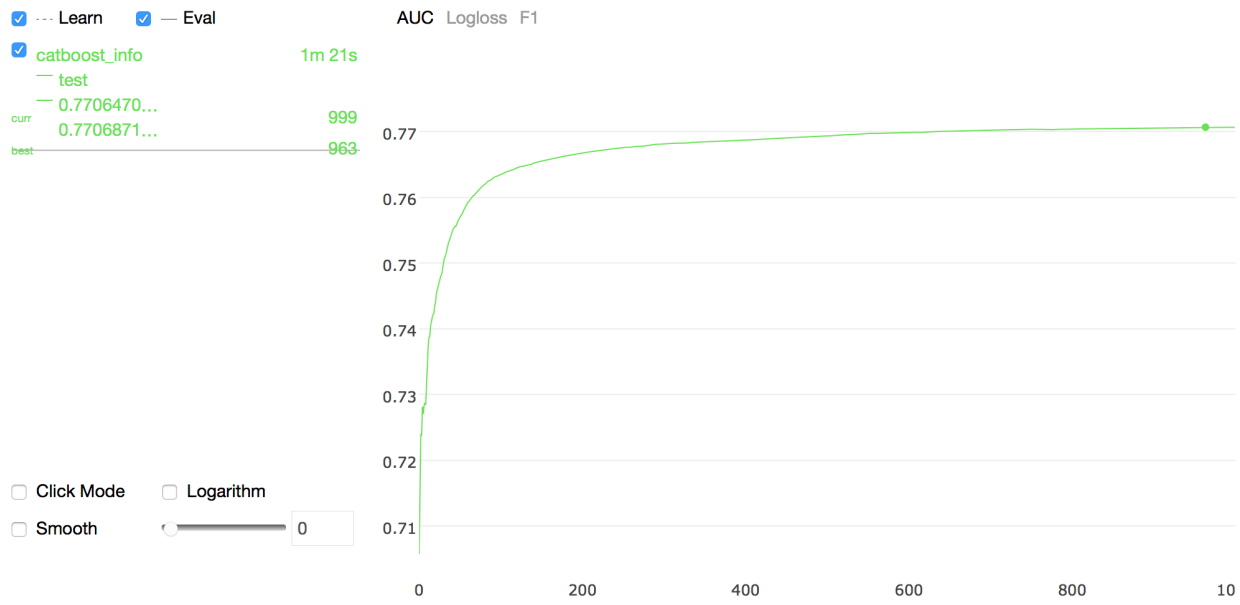
Теперь создаем экземпляр класса `CatBoostClassifier` и обучаем модель.

In[98]:

```
# создаем экземпляр класса CatBoostClassifier
model = CatBoostClassifier(**params)
# обучаем модель на обучающем пуле, с помощью
# контрольного пула оптимизируем AUC и
# печатаем графики обучения и валидации
model.fit(train_pool, eval_set=validate_pool, plot=True)
```

Out[98]:

```
Learning rate set to 0.06487
```



Первым в выводе приводится информация о темпе обучения. Значение по умолчанию определяется автоматически, исходя из свойств набора данных и параметров/гиперпараметров обучения, если удовлетворяются все нижеперечисленные условия:

- решается задача бинарной классификации;
- не изменены значения по умолчанию для гиперпараметров `leaf_estimation_iterations`, `leaf_estimation_method` и `l2_leaf_reg`.

В противном случае значение устанавливается равным 0,03. При выборе `use_best_model=True` темп обучения увеличивается, поэтому нужно брать большее количество итераций или принудительно задавать меньший темп обучения.

Вывод состоит из трех вкладок – `AUC`, `Logloss` и `F1`.

Первая вкладка `AUC` задается оптимизируемой метрикой качества. На рис. 64 подробно разбираются основные компоненты этой вкладки.

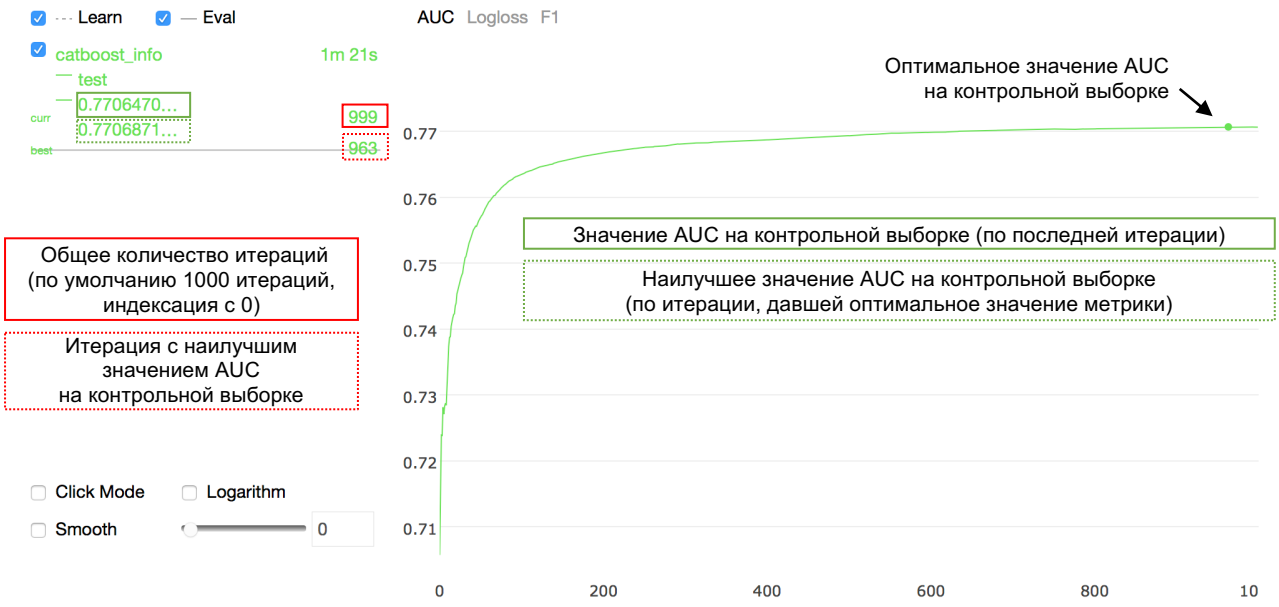


Рис. 64 Компоненты вкладки AUC

Аналогичную информацию мы можем получить и по остальным вкладкам. На рис. 65 показаны компоненты вкладки `Logloss`.

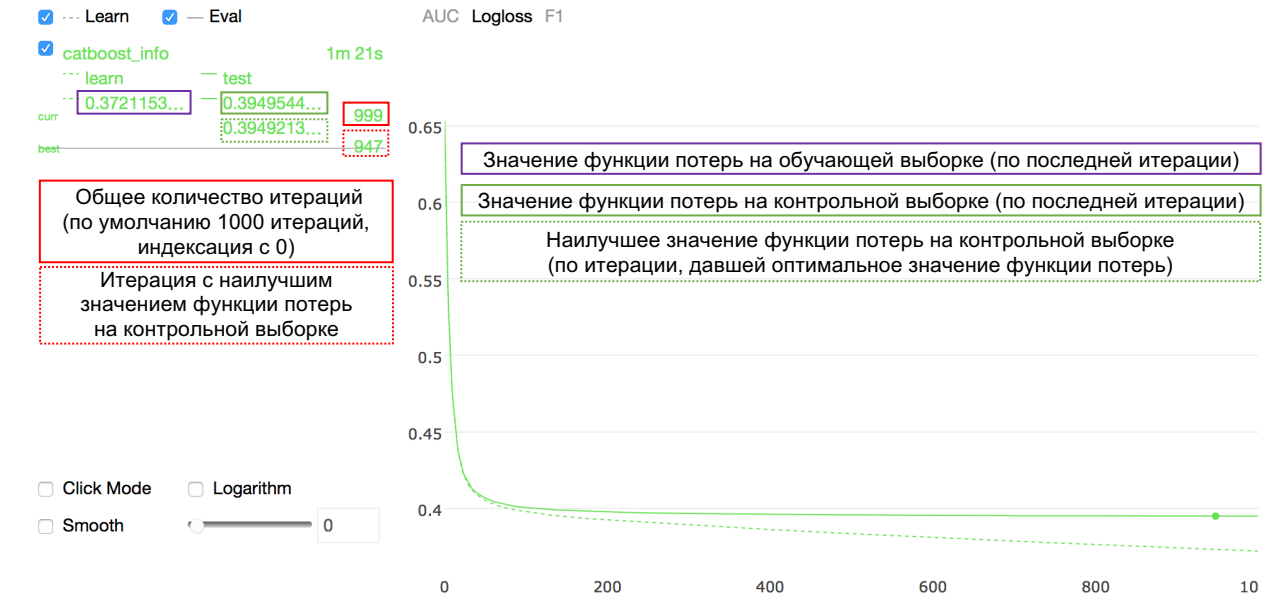


Рис. 65 Компоненты вкладки Logloss

Теперь давайте напечатаем значение AUC на контрольной выборке.

In[99]:
```
# печатаем значение AUC для контрольной выборки
print('AUC модели на контрольной выборке: {:.3}'.format(
    roc_auc_score(y_test_catboost, model.predict_proba(X_test_catboost)[:, 1])))
```

Out[99]:
```
AUC модели на контрольной выборке: 0.771
```

При этом создавать обучающий и контрольный пулы необязательно, мы можем передать в метод `.fit()` исходные массивы признаков и массивы меток и получить тот же самый вывод.

```
In[100]:
# строим модель, передав в метод .fit() исходные
# массивы признаков и массивы меток
model = CatBoostClassifier(
    loss_function='Logloss',
    eval_metric='AUC',
    custom_metric='F1',
    random_seed=42,
    logging_level='Silent',
    use_best_model=False
)

model.fit(
    X_train_catboost, y_train_catboost,
    cat_features=cat_features_idx,
    eval_set=(X_test_catboost, y_test_catboost),
    plot=True
)
```

Обзор параметров и гиперпараметров CatBoost приведен в таблице ниже (важнейшие гиперпараметры выделены красным цветом).

Параметры, отвечающие за выбор оптимизируемой функции потерь/метрики качества	
`loss_function (objective)`	Задает функцию потерь, используемую при обучении (по умолчанию используется значение `Logloss`). Доступные значения: `RMSE`, `Logloss`, `MAE`, `CrossEntropy`, `Quantile`, `LogLinQuantile`, `SMAPE`, `MultiClass` (не поддерживается при использовании GPU), `MultiClassOneVsAll` (не поддерживается при использовании GPU), `MAPE`, `Poisson`, `PairLogit`, `QueryRMSE`, `QuerySoftMax`, `YetiRank`. Для `Quantile` и `LogLinQuantile` можно задать параметр `alpha` (по умолчанию равен 0,5); для `Logloss` – параметр `border`, то есть граничное значение для отнесения наблюдения к положительному классу (по умолчанию равен 0,5); для `YetiRank` – параметры `decay` и `permutations`.
`eval_metric`	Задает метрику качества, используемую для обнаружения переобучения (если используется) и отбора лучшей модели (если используется). Доступные значения: `RMSE`, `Logloss`, `MAE`, `CrossEntropy`, `Quantile`, `LogLinQuantile`, `SMAPE`, `MultiClass` (не поддерживается при использовании GPU), `MultiClassOneVsAll` (не поддерживается при использовании GPU), `MAPE`, `Poisson`, `PairLogit`, `QueryRMSE`, `QuerySoftMax`, `Recall`, `Precision`, `F1`, `TotalF1`, `Accuracy`, `AUC`, `R2`, `MCC`, `PairAccuracy`, `QueryAverage`, `PFound`. Если метрика не задана, то по умолчанию используется оптимизируемая функция потерь.
`custom_metric`	Задает вычисляемую метрику качества для справки. Она не оптимизируется в ходе обучения и отображается только в информационных целях. Доступные значения: `RMSE`, `Logloss`, `MAE`, `CrossEntropy`, `Quantile`, `LogLinQuantile`, `SMAPE`, `MultiClass` (не поддерживается при использовании GPU), `MultiClassOneVsAll` (не поддерживается при использовании GPU), `MAPE`, `Poisson`, `PairLogit`, `QueryRMSE`, `QuerySoftMax`,

	Recall, Precision, F1, TotalF1, Accuracy, AUC, R2, MCC, PairAccuracy, QueryAverage, PFound, CtrFactor. Значения этой метрики для обучающей и контрольной выборки записываются в файлы *learn_error.tsv* и *test_error.tsv*.
Общие параметры и гиперпараметры обучения	
iterations (num_boost_round, n_estimators, num_trees)	Задает максимальное количество деревьев (по умолчанию 1000). Настраивается в первую очередь вместе с гиперпараметром learning_rate. Уменьшая значение learning_rate, выбирайте большее значение iterations. Обратите внимание, после настройки других гиперпараметров данный гиперпараметр вместе с параметром learning_rate нужно настраивать заново.
learning_rate (eta)	Задает скорость обучения. Настраивается в первую очередь вместе с гиперпараметром iterations. Чем меньше значение learning_rate, тем больше итераций требуется для обучения (т.е. необходимо задать большее значение iterations). Значение по умолчанию определяется автоматически, исходя из свойств набора данных и гиперпараметров обучения, если удовлетворяются все нижеперечисленные условия: • решается задача бинарной классификации; • не изменены значения по умолчанию для гиперпараметров leaf_estimation_iterations, leaf_estimation_method и l2_leaf_reg. В противном случае значение устанавливается равным 0,03. Обратите внимание, после настройки других гиперпараметров данный гиперпараметр вместе с параметром iterations нужно настраивать заново.
random_seed (random_state)	Задает стартовое значение генератора случайных чисел для воспроизводимости результатов
l2_leaf_reg (reg_lambda)	Задает коэффициент L2-регуляризации (по умолчанию 3). Можно задать любое положительное значение.
bootstrap_type	Задает тип бутстрепа – метод отбора весов наблюдений. Доступные значения: Poisson (только при использовании GPU), Bayesian (по умолчанию), Bernoulli, No. Обратите внимание, бутстреп применяется на этапе определения структуры дерева, при вычислении значений в листьях используется вся выборка.
bagging_ temperature	Регулирует интенсивность байесовского бутстрепа. Байесовский бутстреп позволяет присваивать случайные веса наблюдениям по формуле: $w(obs) = w(obs) * (-\log(rand(0,1))^{bagging\_temperature}$ Веса порождаются из экспоненциального распределения, если задать значение этого параметра равным 1. Все веса равны 1, если задать значение этого параметра равным 0. Параметр может принимать значение от 0 до бесконечности, но на практике обычно используются значения от 0 (без бэггинга) до 1 (по умолчанию).
subsample	Задает долю отбираемых наблюдений при бэггинге (значение по умолчанию 0,66). Этот параметр можно использовать, если для параметра bootstrap_type задано значение Poisson или Bernoulli.
sampling_ frequency	Задает частоту отбора наблюдений. Возможные значения: PerTree и PerTreeLevel (по умолчанию).

random_strength	Задает множитель стандартного отклонения (по умолчанию 1). Данный параметр позволяет бороться с переобучением модели. Значение этого параметра используется в ходе отбора расщеплений. На каждой итерации каждое возможное расщепление получает оценку с точки зрения того, насколько добавление этого расщепления минимизирует функцию потерь для обучающего набора данных. Выбирается разбиение с наибольшей оценкой. Изначально в этих оценках нет элемента случайности. Затем к оценке добавляется нормально распределенная случайная величина, которая будет зависеть от количества прошедших итераций и длины вектора градиента (ближе к концу обучения уменьшаем случайность): $\text{score(f)} = \text{score(f)} + \text{random\_strength} * \text{N}(0, \text{RndMult} * \text{Sko})$ где: Sko – длина вектора градиента; RndMult – множитель, уменьшающийся при увеличении итерации. Если задать множитель стандартного отклонения равным 0, то в оценку расщеплений не будет привнесена случайность.
use_best_model	Если задать этот параметр равным `True`, количество деревьев в итоговой модели определяется следующим образом: 1. Строим количество деревьев в соответствии с выбранными параметрами обучения. 2. Используем проверочный набор данных для поиска итерации, на которой достигается оптимальное значение метрики, заданной с помощью параметра `eval_metric`. Обратите внимание, деревья, построенные уже после найденной итерации, не будут сохранены. По умолчанию задано значение `True`, если с помощью параметра `eval_set` задан проверочный набор, в противном случае `False`.
best_model_min_trees	Задает минимальное количество деревьев, которое должна иметь наилучшая модель. Если задано определенное целочисленное значение, то итоговая модель будет содержать по крайней мере заданное количество деревьев, даже если наилучшая модель требует меньшее количество деревьев. По умолчанию значение не задано (т.е. минимальное количество деревьев, которое должна содержать наилучшая модель, не задано).
depth	Задает глубину дерева (по умолчанию 6). Можно задать целочисленное значение вплоть до 16. Рекомендуется задавать значения в диапазоне от 1 до 10.
ignored_features	Задает индексы переменных, не используемых при обучении модели. Поддерживаемые операторы: `":"`– разделитель значений индексов, `"-"` – диапазон значений (левый и правый концы включаются). Например, если в нашем обучающем наборе нужно проигнорировать признаки с индексами 1, 2, 7, 42, 43, 44, 45, можно воспользоваться следующей конструкцией: `1:2:7:42-45`. По умолчанию используются все признаки.
one_hot_max_size	Задает one-hot-кодирование для всех категориальных признаков с числом уровней, меньшим или равным значению данного параметра (по умолчанию 2). Лучше работает для категориальных признаков с небольшим количеством уровней.

	Не рекомендуется выполнять one-hot-кодирование самостоятельно в ходе предварительной подготовки данных.
has_time	Если задать этот параметр равным True, то алгоритм будет использовать порядок объектов в обучающих данных (случайные перестановки на этапах преобразования категориальных признаков в количественные и выбора структуры дерева не будут выполняться). По умолчанию задано значение False, т.е. порядок объектов не учитывается, алгоритм генерирует случайные перестановки.
rsm	Задает долю признаков, используемых при создании очередного разбиения узла в процессе построения дерева. Принимает значение от 0 до 1 (по умолчанию, т.е. используются все признаки). Реализовано только для CPU.
nan_mode	Задает метод обработки пропусков. Доступные значения: Forbidden – наличие пропусков не допускается, их наличие приводит к ошибке; Min (по умолчанию) – для пропуска в количественном признаке используется минимальное значение признака; Max – для пропуска в количественном признаке используется максимальное значение признака. Пропуски в категориальном признаке обрабатываются как отдельная категория. Обратите внимание, для древовидных алгоритмов эффективным может быть импутация значением, лежащим вне диапазона имеющихся данных. Например, на этапе предварительной подготовки данных пропуски можно закодировать большим отрицательным значением (-9999).
fold_permutation_ block_size	Перед выполнением случайных пермутаций наблюдения обучающего набора данных группируются в блоки. Данный параметр задает размер блоков. Чем меньше значение, тем медленнее обучение. Большие значения могут привести к ухудшению качества. Параметр может принимать значение от 1 до бесконечности, но на практике обычно используются значения от 1 (по умолчанию) до 256. Для небольших выборок рекомендуется задавать значение 1.
leaf_estimation_ iterations	Задает количество шагов градиентного спуска при расчетах значений в листьях (по умолчанию 1).
leaf_estimation_ method	Задает метод, используемый для расчета значений в листьях. Доступные значения: Newton и Gradient (по умолчанию).
fold_len_ multiplier	Задает коэффициент для изменения размера блоков. Должен быть больше 1 (по умолчанию 2), значения, близкие к 1, приводят к квадратичному относительно числа объектов потреблению памяти и времени вычислений.
approx_on_full_ history	Задает способ вычисления приближенных значений. Если задано значение False (по умолчанию), для вычисления приближенных значений используется только определенная доля наблюдений в блоке. Размер этой доли определяется по формуле $\frac{1}{X}$, где X – это значение параметра fold_len_multiplier. Этот режим более быстрый и в редких случаях дает чуть меньшую правильность. Если задано значение True, то для вычисления приближенных значений будут использованы все предыдущие наблюдения в блоке. Этот

	режим является более медленным и в редких случаях дает чуть большую правильность.
`class_weights`	Задает веса классов зависимой переменной. Значения используются как множители для весов наблюдений соответствующих классов. Для несбалансированного набора при решении задачи бинарной классификации для класса 0 (отрицательного класса) можно задать вес 1, а для класса 1 (положительного класса) задать вес, равный $\frac{\text{количество отрицательных примеров}}{\text{количество положительных примеров}}$. Например, `class_weights=[0.1, 4]` умножает веса наблюдений из класса 0 на 0,1, а веса наблюдений из класса 1 на 4. По умолчанию вес для всех классов устанавливается равным 1.
`boosting_type`	Задает тип бустинга. Доступные значения: `Ordered` и `Plain`. Значение `Ordered` обычно дает лучшее качество на небольших наборах, но работает медленнее, чем `Plain`. Значение `Plain` задает классическую схему градиентного бустинга. Значение по умолчанию зависит от количества наблюдений в обучающем наборе и выбранного режима обучения.
Гиперпараметры для контроля переобучения	
`od_type`	Задает тип используемого детектора переобучения. Доступные значения: `IncToDec` (по умолчанию) и `Iter`. Принцип работы детектора переобучения `IncToDec` выглядит следующим образом. Перед построением каждого нового дерева CatBoost проверяет итоговое изменение функции потерь или метрики, заданной с помощью параметра `eval_metric`, на проверочном наборе данных. Детектор обучения срабатывает, если *текущее p-значение* становится меньше *порога*, заданного с помощью гиперпараметра `od_pval`: *текущее p-значение* < *порог*. Вычисление *текущего p-значения* происходит на основе оценок максимизируемой метрики: 1. Сначала вычисляется *ожидаемый прирост*: *Ожидаемый прирост* $= max_{i_1 \le i_2 \le i} 0{,}99^{i-i_1}(\text{оценка}[i_2] - \text{оценка}[i_1])$ 2. Затем вычисляется x: $$x = \frac{\text{Ожидаемый прирост}[i]}{max_{j \le i}\text{оценка}[j] - \text{оценка}[i]}$$ 3. Вычисляется *текущее p-значение*: *Текущее p-значение* $= exp\left(-\frac{0{,}5}{x}\right)$ Принцип работы детектора переобучения `Iter` выглядит иначе. Перед построением каждого нового дерева CatBoost проверяет количество итераций, прошедших после итерации, на которой было получено оптимальное значение функции потерь или метрики, заданной с помощью параметра `eval_metric`. Модель считается переобученной, если количество итераций превышает значение гиперпараметра `od_wait`.
`od_pval`	Задает порог для детектора переобучения `IncToDec`. Обучение останавливается, когда достигается указанное значение. Чтобы этот гиперпараметр работал, требуется задать проверочный набор данных. Для достижения наилучших результатов рекомендуется использовать значения из диапазона от 10^{-10} до

<table>
<tr><td></td><td>10^{-2}. Чем больше значение, тем раньше сработает остановка. По умолчанию этот гиперпараметр равен 0, т.е. определение переобучения отключено. Не используйте этот гиперпараметр для детектора переобучения <code>Iter</code>.</td></tr>
<tr><td><code>od_wait</code></td><td>Задает количество итераций, в течение которого после достижения оптимального значения функции потерь или метрики, заданной с помощью параметра <code>eval_metric</code>, будет продолжаться обучение. Предназначение этого гиперпараметра зависит от выбранного типа детектора переобучения:
если для гиперпараметра <code>od_type</code> (определяет тип детектора переобучения) задано значение <code>IntToDec</code>, то при достижении порога детектор переобучения игнорируется и обучение будет продолжено в течение указанного числа итераций после достижения оптимального значения функции потерь или метрики, заданной с помощью параметра <code>eval_metric</code>;
если для гиперпараметра <code>od_type</code> (определяет тип детектора переобучения) задано значение <code>Iter</code>, то модель будет оценена как переобученная после достижения оптимального значения функции потерь или метрики, заданной с помощью параметра <code>eval_metric</code>, но ее обучение будет продолжено в течение указанного числа итераций.
По умолчанию значение этого гиперпараметра равно 20.</td></tr>
<tr><td colspan="2">Гиперпараметры бинаризации</td></tr>
<tr><td><code>border_count</code>
(<code>max_bin</code>)</td><td>Задает количество разбиений для количественных признаков. Принимает целочисленные значения от 1 до 255 (по умолчанию 128).</td></tr>
<tr><td><code>feature_border_type</code></td><td>Задает режим бинаризации для количественных признаков. Доступные значения: <code>Median</code>, <code>Uniform</code>, <code>UniformAndQuantiles</code>, <code>MaxLogSum</code>, <code>MinEntropy</code>, <code>GreedyLogSum</code> (по умолчанию).
Перед обучением все возможные значения количественной переменной сортируются по возрастанию и разбиваются на непересекающиеся интервалы (бины), разделенные пороговыми значениями (разбиениями). Размер бинаризации (количество разбиений) определяется начальными параметрами (отдельно вычисляется количество разбиений для количественных признаков и количество разбиений для количественных признаков, полученных в результате преобразования категориальных признаков).
Бинаризация также используется для разбиения категорий на бины при работе с категориальными признаками. Для этого на больших наборах данных используется случайное подмножество набора данных.
В таблице, приведенной ниже, показаны режимы бинаризации, используемые в CatBoost.
<table>
<tr><th>Режим</th><th>Способ определения разбиений</th></tr>
<tr><td><code>Median</code></td><td>Формирует бины с одинаковым количеством наблюдений</td></tr>
<tr><td><code>Uniform</code></td><td>Формирует бины одинаковой ширины</td></tr>
<tr><td><code>UniformAndQuantiles</code></td><td>Комбинирует бины, полученные с помощью режимов Median и Uniform, после того, как уменьшает вдвое размер бинаризации для каждого из этих режимов.</td></tr>
<tr><td><code>MaxLogSum</code></td><td>Максимизирует значение следующего выражения внутри каждого бина:</td></tr>
</table>
</td></tr>
</table>

	$\sum_{i=1}^{n} \log(weight)$, где • n – количество различных значений в бине; • $weight$– частота появления отдельного значения в бине.
MinEntropy	Минимизирует значение следующего выражения внутри каждого бина: $\sum_{i=1}^{n} weight \cdot \log(weight)$, где • n – количество различных значений в бине; • $weight$– частота появления отдельного значения в бине.
GreedyLogSum	Максимизирует жадную аппроксимацию следующего выражения внутри каждого бина: $\sum_{i=1}^{n} \log(weight)$, где • n – количество различных значений в бине; • $weight$– частота появления отдельного значения в бине.

Гиперпараметры преобразования категориальных признаков в количественные

simple_ctr

Задает настройки бинаризации для простых категориальных признаков.

Настройки:

CtrType – метод преобразования категориальных признаков в количественные.

Значения, которые можно задать при обучении на CPU:

Borders	(например, simple_ctr='Borders' или 'simple_ctr': 'Borders')
Buckets	
BinarizedTargetMeanValue	
Counter	

Значения, которые можно задать при обучении на GPU:

- Borders
- Buckets
- FeatureFreq
- FloatTargetMeanValue

TargetBorderCount – количество граничных значений при бинаризации зависимой переменной (только для задачи регрессии и только при обучении на CPU). Допускаются значения от 1 (по умолчанию) до 255. Для CtrType необходимо указать значение BinarizedTargetMeanValue:

```
'simple_ctr': 'BinarizedTargetMeanValue:TargetBorderCount=10'
```

ИЛИ

```
simple_ctr='BinarizedTargetMeanValue:TargetBorderCount=10'
```

TargetBorderType – тип бинаризации зависимой переменной (только для задачи регрессии и только при обучении на CPU). Для CtrType необходимо указать значение BinarizedTargetMeanValue:

```
'simple_ctr': 'BinarizedTargetMeanValue:TargetBorderCount=10:
TargetBorderType=Median'
```

ИЛИ

```
simple_ctr='BinarizedTargetMeanValue:TargetBorderCount=10:
TargetBorderType=Median'
```

Значения, которые можно задать при обучении на CPU:

	• Median • Uniform • UniformAndQuantiles • MaxLogSum • MinEntropy (по умолчанию) • GreedyLogSum CtrBorderCount – количество разбиений для категориальных признаков (от 1 до 255). Необходимо задать значение для CtrType: 'simple_ctr': 'Borders:CtrBorderCount=10' *или* simple_ctr='Borders:CtrBorderCount=10' CtrBorderType – тип бинаризации категориальных признаков. Значения, которые можно задать при обучении на CPU: Uniform Значения, которые можно задать при обучении на GPU: • Median • Uniform • UniformAndQuantiles • MaxLogSum • MinEntropy • GreedyLogSum Prior – задает априорные вероятности. Если указать одно число, оно будет прибавляться в числителю. Если указать пару (только при использовании GPU), первое число будет прибавляться к числителю, а второе – к знаменателю. Необходимо задать значение для CtrType: 'simple_ctr': 'Borders:CtrBorderCount=10:Prior=10' *или* simple_ctr='Borders:CtrBorderCount=10:Prior=10'
combinations_ctr	Задает настройки бинаризации для комбинаций категориальных признаков. Настройки аналогичны настройкам для simple_ctr.
Параметры для многоклассовой классификации	
classes_count	Задает количество классов для многоклассовой классификации. Значение вычисляется следующим образом: наибольшее значение метки класса + 1.
Параметры для настройки производительности	
thread_count	Задает количество используемых ядер/потоков. По умолчанию задано значение -1, что соответствует использованию всех доступных ядер.
Параметры вывода результатов	
logging_level	Задает уровень детализации вывода. Возможные значения: • Silent – ничего не выводить; • Verbose – выводить информацию о значении оптимизируемой метрики, прошедшего времени и ожидаемого времени до окончания обучения; • Info – выводить дополнительную информацию и количество деревьев; • Debug – выводить информацию для отладки.
metric_period	Задает частоту вывода информации (количество итераций между выводом информации). Значение должно быть целым положительным числом (по умолчанию 1 – информация выводится на каждой итерации).
train_dir	Задает папку для хранения создаваемых в процессе обучения файлов (по умолчанию catboost_info).

model_size_reg	Задает коэффициент регуляризации, определяющий размер модели. Чем больше значение (от 0 до бесконечности, по умолчанию 0,5), тем меньше размер модели: происходит уменьшение числа комбинаций признаков в модели, что может повлиять на ее качество.
allow_writing_files	Позволяет записывать файлы с информацией о метриках и контрольными точками в процессе обучения. Если задать равным False, возможность записи будет отключена.
save_snapshot	Позволяет записывать файлы в процессе обучения. Если задать равным False, возможность записи будет отключена.
snapshot_file	Задает имя файла, в котором сохраняется информация о прогрессе обучения. Если файл не существует, то он будет создан, и в него будет записана соответствующая информация. Если же файл уже существует, данные из него будут загружены, и обучение будет продолжено.

Теперь построим модель логистической регрессии.

In[101]:
```
# импортируем класс LogisticRegression
from sklearn.linear_model import LogisticRegression
# создаем экземпляр класса LogisticRegression
# и подгоняем модель
logreg = LogisticRegression().fit(X_train, y_train)
# оцениваем дискриминирующую способность
# модели логистической регрессии
print("AUC на обучающей выборке: {:.3f}".
      format(roc_auc_score(y_train, logreg.predict_proba(X_train)[:, 1])))
print("AUC на контрольной выборке: {:.3f}".
      format(roc_auc_score(y_test, logreg.predict_proba(X_test)[:, 1])))
```

Out[101]:
```
AUC на обучающей выборке: 0.546
AUC на контрольной выборке: 0.545
```

Дискриминирующая способность модели логистической регрессии намного хуже дискриминирующей способности модели случайного леса и модели градиентного бустинга. Это обусловлено тем, что предварительная подготовка данных была оптимизирована под древовидные алгоритмы (одиночное дерево решений, случайный лес, градиентный бустинг), а для логистической регрессии данные нужно подготавливать иначе, с учетом важнейших предпосылок регрессионного анализа. Такой низкий результат – частое явление у начинающего моделера, который пытается «натравить» инструмент без должного понимания математического аппарата того или иного метода машинного обучения. Давайте улучшим результат логистической регрессии, разобравшись в самом методе и выполнив отдельную предварительную подготовку данных, оптимизированную под данный метод.

Математический аппарат логистической регрессии

Модель логистической регрессии представляет собой регрессионную модель, в которой зависимая переменная является бинарной. Такую модель еще называют регрессионной моделью бинарного выбора. Обратите внимание, что строить обычную линейную регрессионную

модель с бинарными зависимыми переменными нельзя. В этом случае невозможно будет интерпретировать предсказанные по регрессии в непрерывной количественной шкале значения зависимой переменной. Значения предикторов в модели бинарного выбора должны быть измерены в количественной шкале. Также в модель бинарного выбора можно включать в качестве предикторов категориальные переменные, которые должны быть предварительно преобразованы в дамми-переменные.
В модели бинарного выбора мы строим регрессионную модель зависимости вероятности того, что бинарная зависимая переменная примет значение 1 при заданных значениях независимых переменных. Для моделирования вероятности бинарной зависимой переменной подбирают специальную монотонно возрастающую логистическую функцию, которая может принимать значения только от 0 до 1.

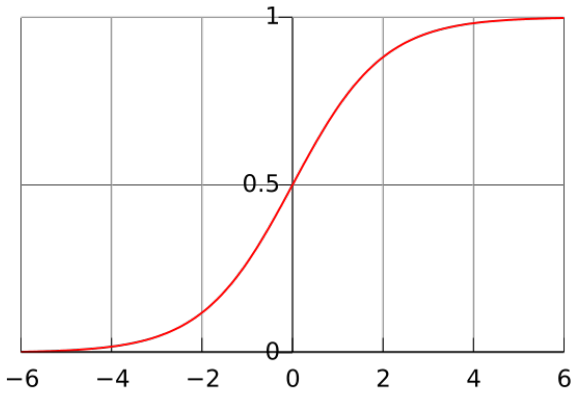


Рис. 66 Логистическая функция (сигмоида)

Ключевыми понятиями для понимания работы метода логистической регрессии являются вероятность и шанс.
Вероятность – это объективная мера появления некоторого события, измеряемая от 0 до 1. На практике оценкой вероятности служит относительная частота появления события. Значение вероятности 0 означает невозможность появления события. Значение вероятности 1 означает, что событие непременно произойдет.

Вероятность $(probability)$: p_i

Шансы – это отношение вероятности того, что событие произойдет (вероятности «успеха»), к вероятности того, что событие не произойдет (вероятности «неудачи»). С ростом вероятности растут шансы, и наоборот. Значение шансов 1 соответствует ситуации, когда вероятности события и отсутствия события равны.

Шанс $(odds)$: $\dfrac{p_i}{1-p_i}$

Например, если вероятность просрочки 90+ (событие произошло) равна 0,8, тогда вероятность отсутствия просрочки 90+ (событие не произошло) равна 1 − 0,8 = 0,2, шансы наличия просрочки 90+ равны 0,8 / 0,2 = 4. Это означает, что шансы наличия просрочки 90+ составляют 4 к 1. Шансы отсутствия просрочки 90+ будут равны 0,2 / 0,8 = 0,25. Выглядит немного странно, но действительно шансы отсутствия просрочки 90+ будут равны 1 к 4. Шансы наличия просрочки 90+ и шансы отсутствия просрочки 90+ являются обратными величинами по отношению друг к другу, т. е. 1 / 4 = 0,25 и 1 / 0,25 = 4. Зная шансы $\frac{p}{1-p} = \frac{0,8}{0,2} = 4$, легко получить вероятность по формуле $\frac{\text{шансы}}{1+\text{шансы}} = \frac{4}{1+4} = 0,8$.

Отношение шансов (odds ratio) − показатель, позволяющий оценить влияние фактора на вероятность события.

Предположим, что из 10 мужчин семеро поступили в университет, тогда как из 10 женщин поступили лишь три. Событием у нас является поступление в университет, а фактором − пол.

Для мужчины вероятность поступить составляет p = 7/10 = 0,7, а вероятность не поступить составляет q = 1 − 0,7 = 0,3.

Для женщины вероятность поступить составляет p = 3/10 = 0,3, а вероятность не поступить составляет q = 1 − 0,3 = 0,7.

Теперь мы можем воспользоваться вероятностями для вычисления шансов на поступление как для мужчин, так и для женщин:

шансы (мужчины) = 0,7 / 0,3 = 2,33333;

шансы (женщины) = 0,3 / 0,7 = 0,42857.

Теперь вычислим отношение шансов ОШ = 2,3333 / 0,42857 = 5,44.

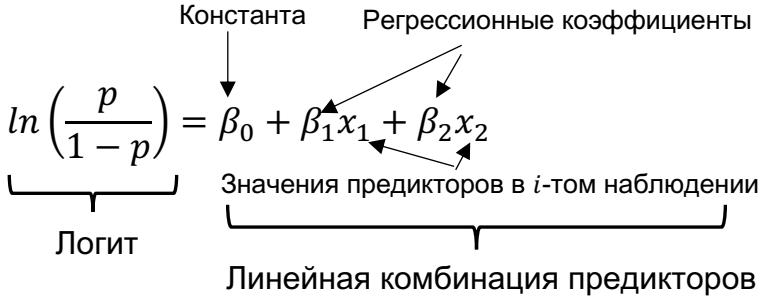
Таким образом, шансы поступления мужчины в 5,44 раза выше шансов поступления женщины.

Ядром математического аппарата логистической регрессии является логит. Логит − это натуральный логарифм шанса.

Логит ($logit$) или логарифм шансов ($\log odds$): $\ln\left(\frac{p_i}{1-p_i}\right)$

Математически модель логистической регрессии выражает зависимость логита от линейной комбинации предикторов.

Модель логистической регрессии, выраженная через логит:



Линейная комбинация предикторов состоит из константы (β_0), регрессионных коэффициентов (β_1, β_2) и предикторов (x_1, x_2). Константа – это натуральный логарифм шансов, когда все предикторы равны 0.

Допустим, мы строим с помощью логистической регрессии скоринговую модель, и событием является просрочка 90+ дней. Модель будет представлять собой зависимость натурального логарифма шансов наступления просрочки 90+ от линейной комбинации значений характеристик заемщика (предикторов), взятых с определенными весами (коэффициентами). Коэффициенты еще называют параметрами регрессионной модели.

Логит обладает интересными свойствами. Он в отличие от вероятности не имеет нижней и верхней границы. Шансы исключают верхнюю границу вероятности, а логарифм шансов исключает нижнюю границу вероятности. При p=1 логит неопределен, потому что шансов 1/0 не существует. При p=0 логит также неопределен, потому что логарифм шансов 0/1 или 0 не существует. По мере приближения вероятности к 1, логит стремится к + ∞. По мере приближения вероятности к 0, логит стремится к - ∞. Таким образом, логит варьирует от - ∞ к + ∞. Проблема нижней и верхней границ вероятности исчезает, поэтому в логистической регрессии в отличие от деревьев решений спрогнозированных вероятностей 0 и 1 не существует.

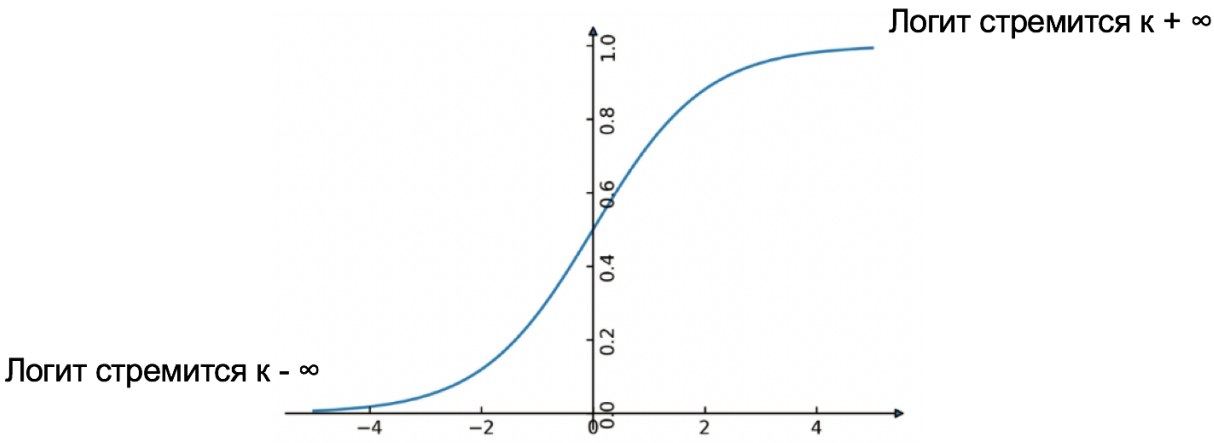


Рис. 67 Свойства логита

Проэкспоненцировав обе части уравнения $ln\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x_1 + \beta_2 x_2$, получим шансы:

$$\frac{p}{1-p} = e^{b_0} \times e^{b_1 x_1} \times e^{b_2 x_2} = e^{b_0 + b_1 x_1 + b_2 x_2}$$

В бизнесе модель логистической регрессии часто выражают не через логит, а через вероятность, используя следующее уравнение:

Модель логистической регрессии, выраженная через вероятность:

$$p = \frac{\text{шансы}}{1 + \text{шансы}} = \frac{e^{\beta_0+\beta_1 x_1+\beta_2 x_2}}{e^{\beta_0+\beta_1 x_1+\beta_2 x_2}+1} = \frac{1}{1+e^{-(\beta_0+\beta_1 x_1+\beta_2 x_2)}} = \frac{1}{1+e^{-y}}$$

В линейной регрессии мы оцениваем параметры модели при помощи метода наименьших квадратов. Иными словами, мы выбираем параметры так, чтобы минимизировать сумму квадратов расстояний между наблюдаемыми и предсказанными значениями зависимой переменной. В логистической регрессии для оценки параметров используется метод максимального правдоподобия.

Цель метода максимального правдоподобия – поиск параметров, максимизирующих правдоподобие наблюдаемых данных.

Если бы мы сейчас занимались непосредственно прогнозами, основанными на ряде серьезных предположений, нас интересовали бы вероятности, например, вероятность того, что дефолт произойдет или не произойдет. Однако на этапе анализа данных у нас есть лишь собранные данные, они зафиксированы и здесь нас интересуют такие параметры модели, которые наилучшим образом описывают наблюдаемые данные. Чем лучше эти параметры описывают данные, тем более правдоподобными они являются.

Давайте возьмем тривиальный пример и найдем значения параметров, при которых наблюдаемые данные являются наиболее правдоподобными.

Предположим, мы бросаем монету 100 раз, наблюдаем 56 случаев выпадения «решки» и 44 случая выпадения «орла». Вместо предположения о том, что p равно 0,5, мы хотим получить оценку максимального правдоподобия для p, исходя из конкретного набора данных.

Как это реализовать? Найти значение p, которое делает наблюдаемые данные наиболее правдоподобными. Наши данные зафиксированы: $n = 100$ (общее количество бросков), $h = 56$ (общее количество выпадений «решки»). Представим, что p равно 0,5. Формула функции правдоподобия для биномиального распределения выглядит так:

$$\frac{n!}{h!\,(n-h)!}p^h(1-p)^{n-h}$$

Включение значения 0,5 в нашу биномиальную вероятностную модель приводит к следующему результату:

$$L(p = 0{,}5|\text{данные}) = \frac{100!}{56!\,44!}0{,}5^{56}0{,}5^{44} = 0{,}0389$$

А что если бы p было равно 0,52?

$$L(p = 0{,}52|\text{данные}) = \frac{100!}{56!\,44!}0{,}52^{56}0{,}48^{44} = 0{,}0581$$

Таким образом, можно сделать вывод, что p, равное 0,52, является более правдоподобным, чем p, равное 0,5. Мы можем вывести таблицу значений правдоподобия для различных значений параметра p, чтобы найти оценку максимального правдоподобия для p (рис. 68).

p	L
0,48	0,0222
0,50	0,0389
0,52	0,0581
0,54	0,0739
0,56	0,0801
0,58	0,0738
0,60	0,0576
0,62	0,0378

Рис. 68 Таблица значений параметра *p* и значений максимального правдоподобия

Если мы визуализируем эти данные, то получим следующий график (рис. 69).

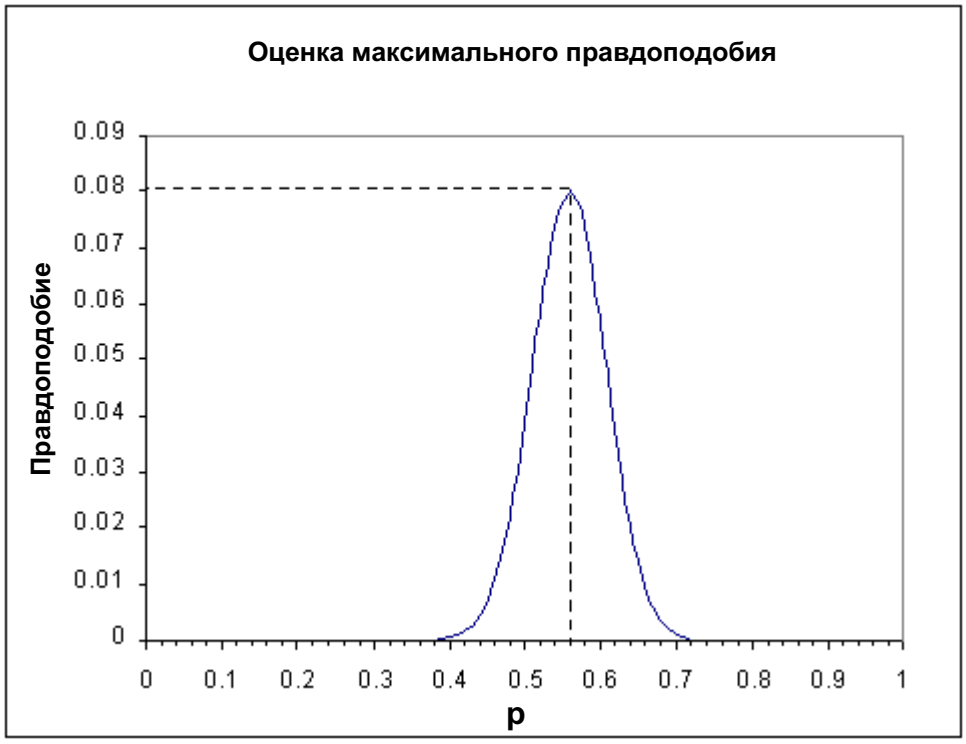


Рис. 69 График функции максимального правдоподобия

Мы видим, что максимальное правдоподобие достигается при p, равном 0,56. Фактически оно в точности равно 0,56, и благодаря этому

тривиальному примеру легко понять, почему данная оценка так важна. Очевидно, что наилучшей оценкой параметра p для любой взятой выборки будет фактическая доля выпадений «решки» (0,56). Конечно, в таком простом примере никто не будет использовать оценку максимального правдоподобия для параметра p. Но не все примеры такие простые. Чем сложнее модель, чем больше у нее параметров, тем сложнее строить предположения, при каких значениях параметров мы сможем получить максимальное правдоподобие.

В логистической регрессии мы ищем такой вектор регрессионных коэффициентов β_0, β, который максимизирует значение функции правдоподобия на обучающей выборке. Для этого используется следующая формула:

$$L(\beta_0, \beta) = \prod_{i=1}^{n} p_i^{y_i} (1 - p_i)^{1-y_i}$$

где:

p_i – вероятность того, что при значениях предикторов в i-том наблюдении переменная y примет значение y_i, $i = 1, \dots, n$ (для наблюдений, в которых $y_i = 1$, i-тый сомножитель в формуле равен p_i, а для наблюдений, в которых $y_i = 0$, i-тый сомножитель в формуле равен $1-p_i$).

На практике вместо максимизации функции правдоподобия обычно решают задачу максимизации ее логарифма:

$$\ln (L(\beta_0, \beta)) = \sum_{i=1}^{n} (y_i \ln p_i + (1 - y_i) \ln(1 - p_i))$$

Данная функция является дифференцируемой по параметрам β_0, β. Однако в отличие от функции, минимизируемой методом наименьших квадратов в задаче построений линейной регрессии, искомые значения, максимизирующие вышеприведенную функцию, не могут быть найдены по явным формулам. Для максимизации функции используются приближенные итеративные методы, например, метод градиентного спуска или метод Ньютона.

Теперь выясним, как следует интерпретировать регрессионные коэффициенты логистической регрессии. Допустим, у нас есть предикторы *Стаж работы*, *Стаж проживания*, *Процент долговых обязательств от дохода*, *Задолженность по кредитной карте*, и мы учимся по ним прогнозировать просрочку 90+. По итогам регрессионного оценивания мы обычно получаем следующую таблицу (рис. 70).

	Регрессионный коэффициент	Значимость (p-значение)	Регрессионный коэффициент, записанный через экспоненту
Стаж работы	-0,243	0,0001	0,785
Стаж проживания	-0,081	0,0001	0,922
Процент долговых обязательств от дохода	0,088	0,0001	1,092
Задолженность по кредитной карте	0,573	0,0001	1,774
Константа	-0,791	0,002	0,453

Рис. 70 Регрессионные коэффициенты логистической регрессии

Коэффициент b_i логистической регрессии показывает, насколько в среднем изменится натуральный логарифм шанса события при изменении предиктора на единицу своего измерения при том, что остальные предикторы фиксированы. Если коэффициент b_i является положительным, то с увеличением значения предиктора на единицу измерения натуральный логарифм шанса события возрастает. Если коэффициент b_i является отрицательным, то с увеличением значения предиктора на единицу измерения натуральный логарифм шанса события убывает. Альтернативный способ интерпретации параметров логистической регрессии заключается в записи регрессионных коэффициентов через экспоненту $\exp(b_i)$ (такие коэффициенты называют экспоненциальными). Экспоненциальный коэффициент $\exp(b_i)$ показывает, во сколько раз в среднем изменится шанс события при изменении независимой переменной на единицу своего измерения при том, что остальные предикторы фиксированы. Если коэффициент $\exp(b_i)$ больше единицы, то с увеличением значения предиктора на единицу измерения шанс события в определенное количество раз возрастает. Если коэффициент $\exp(b_i)$ меньше единицы, то с увеличением значения предиктора на единицу измерения шанс события в определенное количество раз убывает. Если коэффициент b_i положительный, то коэффициент $\exp(b_i)$ всегда будет больше единицы, и, наоборот, если коэффициент b_i отрицательный, то коэффициент $\exp(b_i)$ всегда будет меньше единицы. Давайте потренируемся интерпретировать регрессионные коэффициенты.

Регрессионный коэффициент при предикторе *Стаж работы* отрицателен и равен $-0{,}243$. Это обозначает, что при увеличении стажа работы на единицу измерения (при том, что остальные предикторы остаются неизменными) натуральный логарифм шанса просрочки 90+ уменьшается на 0,243. Поскольку регрессионный коэффициент при предикторе *Стаж работы* является отрицательным, то экспоненциальный коэффициент меньше 0 и равен $\exp(-0{,}243) = 0{,}785$. Это обозначает, что при увеличении стажа работы на единицу измерения (при том, что остальные предикторы остаются неизменными) шанс просрочки 90+ уменьшается в 0,785 раза.

Допустим, у нас есть клиент со стажем работы 1 год и нулевыми значениями остальных предикторов. Вычислим для него натуральный

логарифм шанса просрочки 90+: −0,791 − 0,243*1 − 0,081*0 + 0,088*0 + 0,573*0 = −1,034. Экспоненциальный коэффициент равен exp(−1,034) = 0,36. А теперь вычислим натуральный логарифм шанса просрочки 90+ для клиента со стажем работы 2 года и нулевыми значениями остальных предикторов: −0,791 − 0,243*2 − 0,081*0 + 0,088*0 + 0,573*0 = −1,277. Экспоненциальный коэффициент равен exp(−1,277) = 0,28. Видим, что произошло уменьшение натурального логарифма шанса дефолта с −1,034 до −1,277, т. е. −1,277 + 1,034 = −0,243, т.е. получаем наш регрессионный коэффициент. Шанс просрочки 90+ для заемщика, у которого стаж работы на 1 год больше, уменьшается с 0,36 до 0,28, т.е. в 0,78 раз ($\frac{0,28}{0,36} = 0,78$), т.е. получаем наш экспоненциальный коэффициент.

Натуральный логарифм шансов (логит)

$$ln\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_4 = -0,791 - 0,243 * 1 + 0 + 0 + 0 = -1,034$$

Шансы

$$\frac{p}{1-p} = e^{b_0} \times e^{b_1 x_1} \times e^{b_2 x_2} \times e^{b_3 x_3} \times e^{b_4 x_4} = 0,453 \times 0,784 \times 1 \times 1 \times 1 \times 1 = 0,36$$

$$\frac{p}{1-p} = e^{b_0 + b_1 x_1 + b_2 x_2 + b_3 x_3 + b_4 x_4} = e^{-1,034} = 0,36$$

Рис. 71 Иллюстрация вычисления логита и шансов для клиента со стажем работы 1 год и нулевыми значениями остальных предикторов

Высокие *p*-значения говорят о том, что регрессионный коэффициент не является статистически значимым, а низкие *p*-значения, наоборот, свидетельствуют о статистической значимости регрессионного коэффициента. Мы выдвигаем нулевую гипотезу о том, что данный регрессионный коэффициент не отличается значимо от 0. Если регрессионный коэффициент значимо отличается от 0, мы можем предположить, что предиктор вносит значимый вклад в прогнозирование зависимой переменной. В данном случае для каждого регрессионного коэффициента мы можем отвергнуть нулевую гипотезу (все регрессионные коэффициенты значимо отличаются от 0).

Часто пытаются судить о важности предиктора на основе значимости коэффициента при предикторе, например, используют статистику Вальда. Статистика Вальда равна отношению квадрата регрессионного коэффициента к квадрату стандартной ошибки и подчиняется распределению хи-квадрат.

К сожалению, у статистики Вальда есть одно очень нежелательное свойство. Когда модуль регрессионного коэффициента увеличивается, оценка стандартной ошибки также растет. Это приводит к слишком маленькому значению статистики Вальда, и в результате нулевая гипотеза о равенстве нулю коэффициента не отвергается даже в тех случаях, когда она не верна (ошибка II рода). Следовательно, при больших коэффициентах не следует полагаться на статистику Вальда. Лучше построить две модели − с изучаемой переменной и без нее − и

проверять гипотезы, основываясь на изменении логарифма правдоподобия. Оценка статистики Вальда имеет тенденцию быть смещенной в условиях разреженных данных.
Как вариант, можно строить модель логистической регрессии с одним предиктором и оценивать нужную метрику (правильность или AUC) на контрольном наборе.
С учетом найденных регрессионных коэффициентов уравнение логистической регрессии для вероятности просрочки 90+ можно записать следующим образом:

$$p_i = \frac{1}{1 + e^{-\left(b_0 + b_1 x_i^{(1)} + b_2 x_i^{(2)} + \cdots + b_k x_i^{(k)}\right)}} =$$

$$= \frac{1}{1 + e^{-(-0{,}791 - 0{,}243 \times \text{Работа} - 0{,}081 \times \text{Проживание} + 0{,}088 \times \text{Процент} + 0{,}573 \times \text{Задолженность})}}$$

Это уравнение наглядно иллюстрирует преимущество логистической регрессии: мы можем объяснить клиенту, как была получена вероятность просрочки 90+, и какой вклад внес в эту вероятность каждый признак.
Перед построением логистической регрессии вы должны убедиться, что выполняются некоторые предпосылки: нормальное распределение, единый масштаб измерения переменных и отсутствие взаимосвязи между предикторами. Однако выполнение предпосылок мы проиллюстрируем непосредственно в ходе предварительной подготовки данных, оптимизированной под логистическую регрессию. Давайте приступим к этой подготовке.

Отдельная предварительная подготовка данных для логистической регрессии

Заново загружаем CSV-файл.

In[102]:
```
# записываем CSV-файл в объект DataFrame
data = pd.read_csv('C:/data/credit_train.csv', encoding='cp1251', sep=';')
```

Удаляем идентификационную переменную.

In[103]:
```
# удаляем переменную client_id
data.drop('client_id', axis=1, inplace=True)
```

Выполняем нормализацию категорий переменной *living_region*.

In[104]:
```
# заменяем исходные категории переменной
# living_region на новые
data['living_region'] = data['living_region'].map(regions)
```

Выполняем преобразования типов переменных.

In[105]:
```
# преобразуем указанные переменные в тип object
for i in ['tariff_id', 'open_account_flg']:
    data[i] = data[i].astype('object')

# в указанных переменных заменяем запятые на точки и
# преобразуем в тип float
for i in ['credit_sum', 'score_shk']:
    data[i]=data[i].str.replace(',', '.').astype('float')
```

Теперь укрупняем редкие категории, немного изменив порог для укрупнения для переменных *job_position* и *tariff_id*.

In[106]:
```
# укрупняем редкие категории job_position и tariff_id
data.loc[data['job_position'].value_counts()[data['job_position']].values < 50,
         'job_position'] = 'OTHER'

data.loc[data['tariff_id'].value_counts()[data['tariff_id']].values < 30,
         'tariff_id'] = 1.99

data['living_region'] = np.where(data['living_region'].isin(region_series[mask].index),
                                 'OTHER', data['living_region'])
```

На основе категориальной переменной *tariff_id* создадим количественную переменную *tariff*. Затем переменной *tariff_id* присвоим тип `str` и заменим в ее значениях точки на символы нижнего подчеркивания.

In[107]:
```
# на основе категориальной переменной tariff_id создаем
# количественную переменную tariff
data['tariff'] = data['tariff_id'].astype('float')

# заменим точки на символы подчеркивания
data['tariff_id'] = data['tariff_id'].astype('str').str.replace('.', '_')
```

Теперь давайте создадим индикатор пропусков *ind* для переменной *overdue_credit_count*. Если переменная *overdue_credit_count* содержит пропуск, индикатор принимает значение 1, если не содержит, индикатор принимает значение 0.

In[108]:
```
# создаем индикатор пропусков для переменной overdue_credit_count,
# если переменная содержит пропуск, индикатор принимает значение 1,
# если не содержит, индикатор принимает значение 0
data['ind'] = np.where(
    data['overdue_credit_count'].isnull(), 1, 0).astype('object')
```

Формируем обучающую и контрольную выборки в тех же пропорциях, что и ранее, и с тем же стартовым значением генератора случайных чисел.

In[109]:
```
# разбиваем данные на обучающую и контрольную выборки
train=data.sample(frac=0.7,random_state=200)
test=data.drop(train.index)
```

Импутируем пропуски в переменных *age*, *credit_sum*, *score_shk*, *monthly_income*, *credit_count* и *overdue_credit_count* медианами.

In[110]:
```
# импутируем пропуски в указанных переменных медианами
for i in ['age', 'credit_sum', 'score_shk', 'monthly_income',
          'credit_count', 'overdue_credit_count']:
    train[i].fillna(train[i].median(), inplace=True)
    test[i].fillna(train[i].median(), inplace=True)
```

Пропуски в переменных *marital_status* и *education* заменим на самую часто встречающуюся категорию.

In[111]:
```
# пропуски в переменных marital_status и education
# заменяем на самую часто встречающуюся категорию
for i in ['marital_status', 'education']:
    train[i].fillna(train[i].value_counts().index[0], inplace=True)
    test[i].fillna(train[i].value_counts().index[0], inplace=True)
```

Как мы уже говорили выше, перед построением логистической регрессии вы должны убедиться, что выполняются некоторые предпосылки.
В отличие от линейной регрессии, логистическая регрессия не полагается на предположение о нормальном распределении. Однако ваше решение может быть более стабильным, если распределение предикторов будет многомерным нормальным. Прежде всего необходимо проанализировать распределение переменных. Для этого строят гистограмму распределения, а также график квантиль-квантиль, на котором показана связь между наблюдаемыми значениями переменной и квантилями теоретического распределения (по умолчанию используется нормальное распределение). Если наблюдаемые значения попадают на прямую линию, то теоретическое распределение хорошо подходит к наблюдаемым данным, в противном случае применяют преобразования, максимизирующие нормальность распределения.
Разберем основные преобразования.

Логарифм

$\log_{10}(x)$, $\log_e(x)$, $\log_2(x)$

Строгое преобразование, которое применяется для распределения, сильно скошенного вправо. Применяют логарифм по основанию 2, 10 и *e*. Поскольку логарифм нуля, а равно и любого отрицательного числа, не определен, перед использованием логарифмического преобразования ко всем значениям нужно добавить константу, чтобы сделать их положительными. Например, можно добавить 1 и получить log(x + 1). Тонкий момент заключается в том, что получаемое распределение будет зависеть от того, насколько значение x велико по сравнению с константой 1. Устраняют проблему с помощью формулы log(x/mean(x)+k), где k – маленькое значение-константа (меньше 1). В этом преобразовании k будет работать как фактор, определяющий форму распределения (маленькие значения k делают данные более скошенными влево, а большие значения k делают данные менее скошенными).

Обратите внимание, что выбор основания также важен. Более высокие основания сжимают значения сильнее.

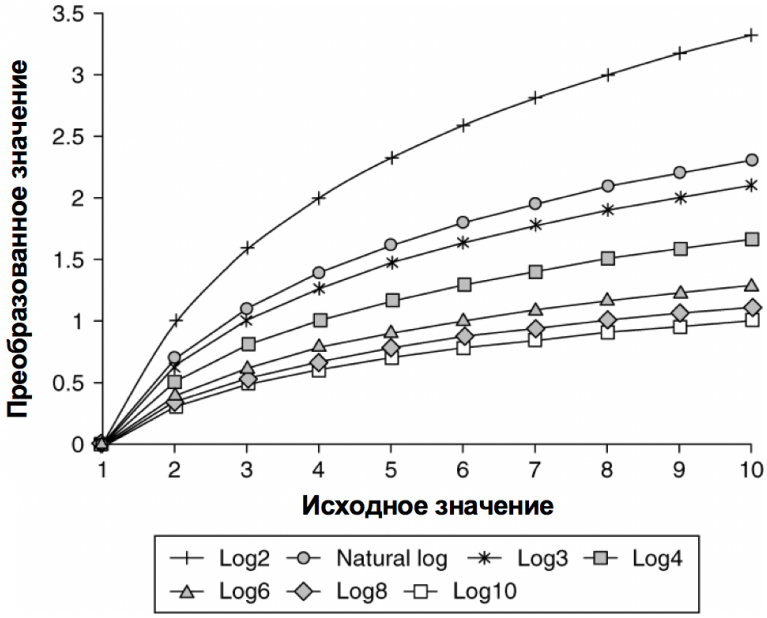


Рис. 72 Зависимость эффективности преобразования от основания логарифма

Корень четвертой степени, кубический корень, квадратный корень

$\sqrt[4]{x},\ \sqrt[3]{x}, \sqrt{x}$

Корень четвертой степени и кубический корень являются менее строгими преобразованиями, чем логарифм. Квадратный корень – менее строгое преобразование, чем корень четвертой степени и кубический корень. В целом корни применяются для распределения, умеренно скошенного вправо. Квадратный корень часто дает хорошее качество для переменных с умеренной правосторонней асимметрией, значения которых представляют собой частоты. Например, речь может идти о распределении частоты редких случайных событий, произошедших за определенный период, которое подчиняется распределению Пуассона. Также квадратный корень дает хорошее качество для переменных с умеренной правосторонней асимметрией, содержащих большое количество очень небольших или нулевых значений. На практике при использовании корней обычно корень берут от модуля числа (чтобы не вычислять корни отрицательных чисел) и затем учитывают знак числа: sign(x) * (abs(x) ^ 1/3), sign(x) * (abs(x) ^ 1/2).

Обратное преобразование, отрицательное обратное преобразование

1/x, -1/x

Строгое преобразование, которое применяется для распределения, очень сильно скошенного вправо. В ситуации распределения, очень сильно скошенного вправо, это преобразование может сработать лучше, чем логарифм. Используют обычно при работе с переменными –

результатами деления одной переменной на другую (т.е. разные коэффициенты, отношения). Его нельзя применить к нулевым значениям, поэтому если есть нулевые значения, нужно добавить константу. Применение данного преобразования к отрицательным значениям не дает содержательной интерпретации, поэтому оно обычно применяется к положительным значениям. Обратное преобразование отношений легко интерпретируется: плотность населения, выраженная как количество человек на единицу площади, станет размером площади на человека; количество пациентов на одного врача станет количеством врачей на одного пациента.

На практике результаты обратного преобразования умножают или делят на определенную константу, например на 1000 или 10000, для удобства интерпретации, но сама по себе эта операция не влияет на асимметрию или линейность.

Положительное обратное преобразование меняет порядок значений, имеющих один и тот же знак, на обратный: наибольшее значение становится наименьшим и т.д. Отрицательное обратное преобразование сохраняет порядок значений, имеющих один и тот же знак.

Также применяется положительное/отрицательное обратное преобразование квадратного корня ($1/\sqrt{x}$ или $-1/\sqrt{x}$), положительное/отрицательное обратное возведение в квадрат ($1/x^2$ или $-1/x^2$), положительное/отрицательное обратное возведение в куб ($1/x^3$ или $-1/x^3$).

Экспоненциальное преобразование

$\exp(x)$, 2^x

Преобразование применяется для распределения, скошенного влево. Может хорошо сработать, если данные содержат логарифмический тренд (отток, выживаемость). Это преобразование можно применять к отрицательным числам. Преобразование зависит от диапазона значений и является одним из самых мощным при работе с данными, скошенными влево. Подбор основания не всегда является простой задачей.

Квадратный корень разности между константой и исходным значением переменной

$\sqrt{k - x}$

Преобразование применяется для распределения, умеренно скошенного влево. Константа k подбирается так, чтобы при вычитании из нее исходного значения переменной итоговое наименьшее значение было равно 1. Обычно в качестве константы берут максимальное значение переменной + 1.

Логарифм разности между константой и исходным значением переменной

$\log(k - x)$

Преобразование применяется для распределения, сильно скошенного влево. Константа подбирается так, чтобы при вычитании из нее исходного значения переменной итоговое наименьшее значение было равно 1. Обычно в качестве константы берут максимальное значение переменной + 1.

Возведение в степень

x^2, x^3

Преобразование применяется для распределения, скошенного влево. Подбор степени – нетривиальная задача, часто зависит от характеристики переменной. На практике обычно используют вторую или третью степень. Возведение в куб применяется для работы с распределением, сильно скошенным влево, возведение в квадрат – для работы с распределением, умеренно скошенным влево.

Арксинус

$\arcsin(x)$, $\arcsin(x/100)$, $\arcsin\sqrt{x}$

Преобразование применяется к данным, которые представляют собой пропорции от 0 до 1 или проценты от 0 до 100. На выходе получаем радианы (или градусы), распределение которых ближе к нормальному. На рис. 73 ниже приведен пример подбора оптимального преобразования для переменной с правосторонней асимметрией.

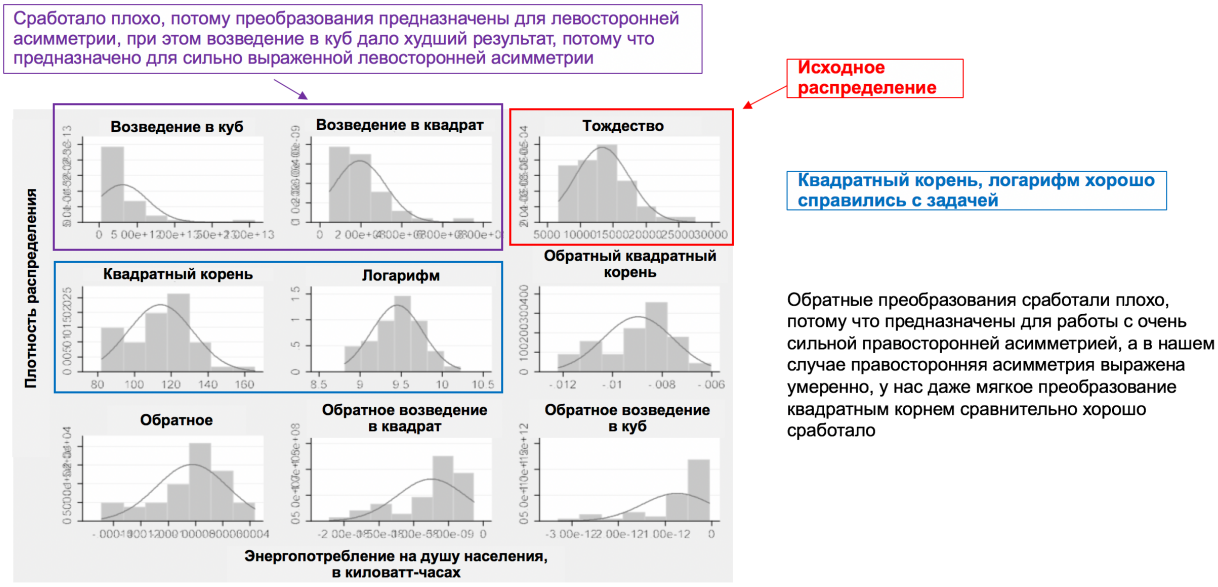


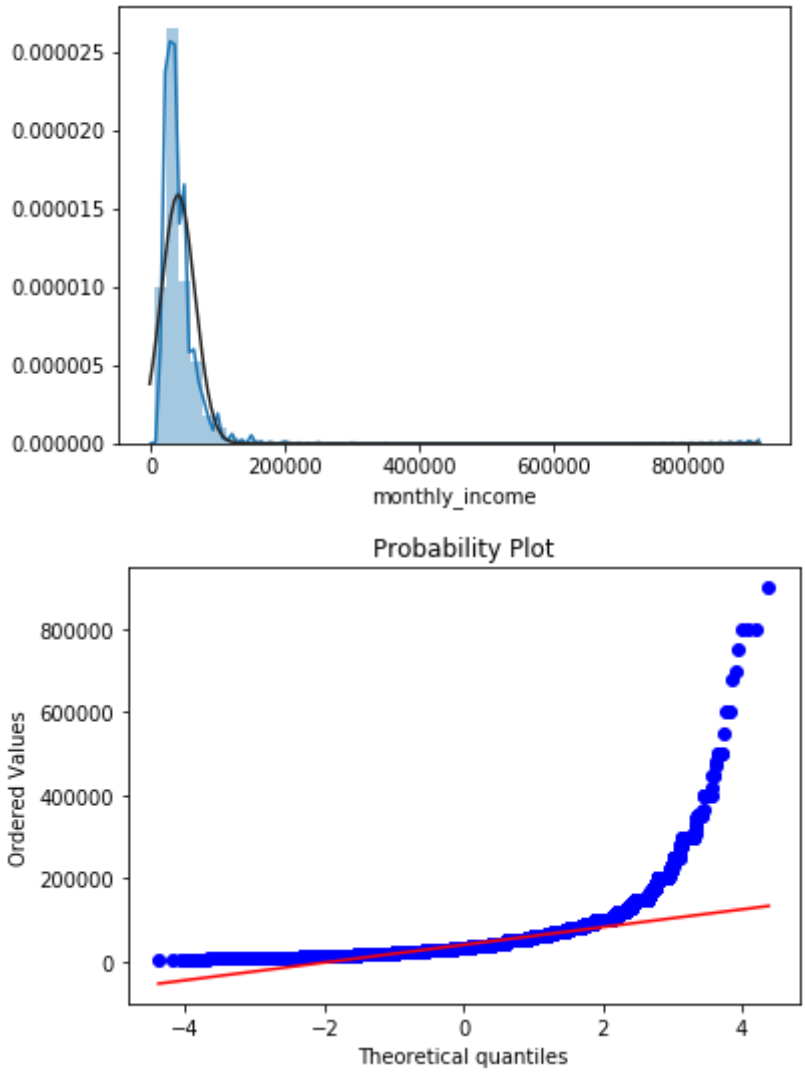
Рис. 73 Пример подбора оптимального преобразования для переменной с правосторонней асимметрией

Давайте попробуем приблизить распределение некоторых количественных переменных к нормальному. Начнем с распределения

переменной *monthly_income*. Давайте построим гистограмму распределения, а также график квантиль-квантиль.

In[112]:
```
# импортируем библиотеку seaborn, предварительно
# установив ее в Anaconda Prompt с помощью команды
# conda install -c anaconda seaborn
import seaborn as sns

# импортируем norm и stats
from scipy import stats
from scipy.stats import norm

# строим гистограмму распределения и график
# квантиль-квантиль для переменной monthly_income
sns.distplot(train['monthly_income'], fit=norm)
fig = plt.figure()
res = stats.probplot(train['monthly_income'], plot=plt)
```

Out[112]:



Исходя из визуального анализа гистограммы, мы видим, что распределение скошено вправо и при этом имеет острую вершину. График квантиль-квантиль показывает, что не все точки лежат на прямой.

О нормальности распределения еще можно судить по коэффициенту асимметрии и коэффициенту эксцесса. В библиотеке pandas

коэффициент асимметрии и коэффициент эксцесса вычисляют с помощью методов `.skew()` и `.kurtosis()` соответственно. Если коэффициент асимметрии положителен, то распределение будет скошено вправо, если отрицателен, распределение будет скошено влево. Если коэффициент эксцесса положителен, то распределение будет иметь острую вершину, если отрицателен, распределение будет иметь пологую вершину. Для нормального распределения коэффициент асимметрии и коэффициент эксцесса равны нулю. В нашем случае мы должны получить положительные коэффициенты асимметрии и эксцесса. Давайте проверим это.

In[113]:
```
# вычисляем коэффициент асимметрии
train['monthly_income'].skew()
```

Out[113]:
```
5.170662523993356
```

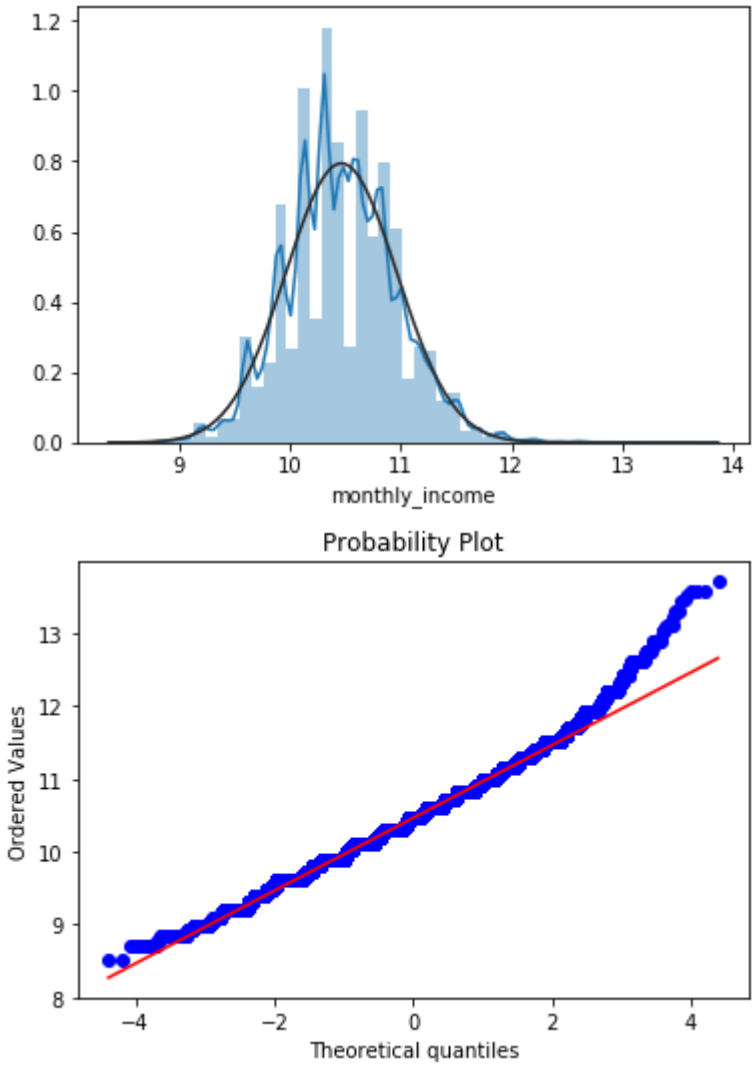
In[114]:
```
# вычисляем коэффициент эксцесса
train['monthly_income'].kurtosis()
```

Out[114]:
```
81.08485953358395
```

Теперь попробуем выполнить преобразование переменной, чтобы приблизить распределение к нормальному. Сначала применим к переменной *monthly_income* логарифмическое преобразование.

In[115]:
```
# строим гистограмму распределения и график
# квантиль-квантиль, применив логарифмическое
# преобразование для переменной monthly_income,
# используем константу a, чтобы не брать
# логарифм нуля
a = 0.001
sns.distplot(np.log(train['monthly_income'] + a), fit=norm)
fig = plt.figure()
res = stats.probplot(np.log(train['monthly_income'] + a), plot=plt)
```

Out[115]:

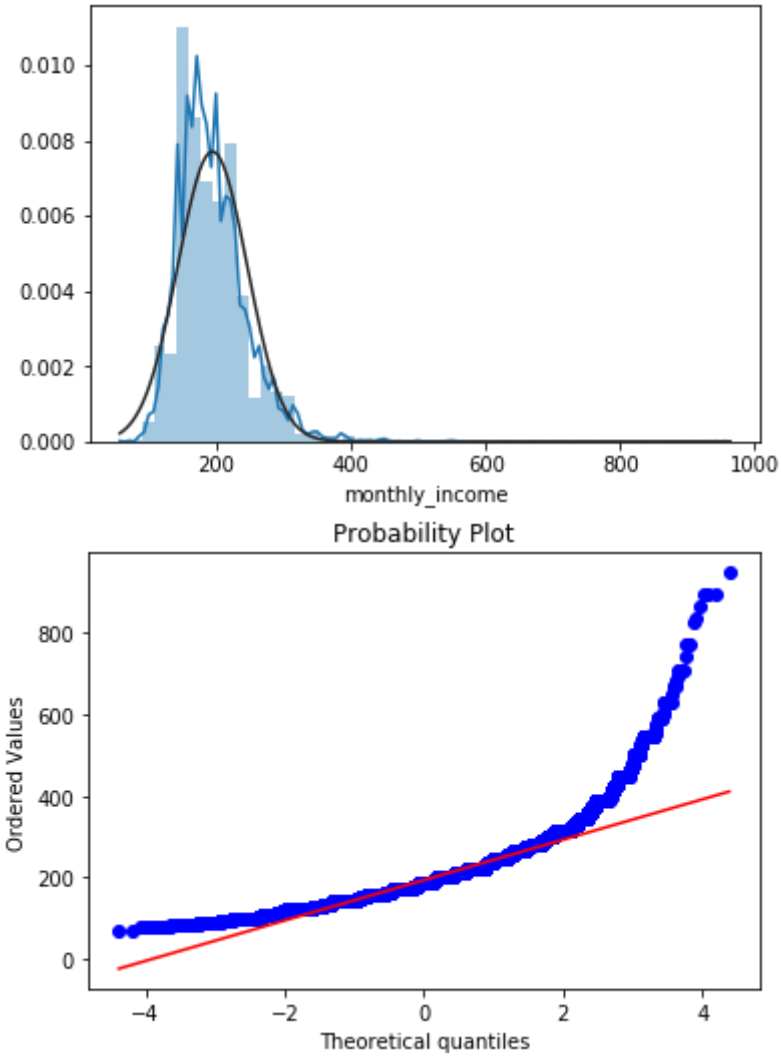


Исходя из гистограммы и графика квантиль-квантиль видим, что по итогам преобразования наше распределение стало больше походить на нормальное. Теперь применим квадратный корень.

In[116]:

```
# строим гистограмму распределения и график
# квантиль-квантиль, применив преобразование
# квадратным корнем для переменной monthly_income,
# используем модуль, чтобы не вычислять корни
# отрицательных чисел, и затем учитываем знак числа

sns.distplot(np.sign(train['monthly_income']) *
             (train['monthly_income'].abs() ** (1/2)), fit=norm)
fig = plt.figure()
res = stats.probplot(np.sign(train['monthly_income']) *
                     (train['monthly_income'].abs() ** (1/2)), plot=plt)
```

Out[116]:

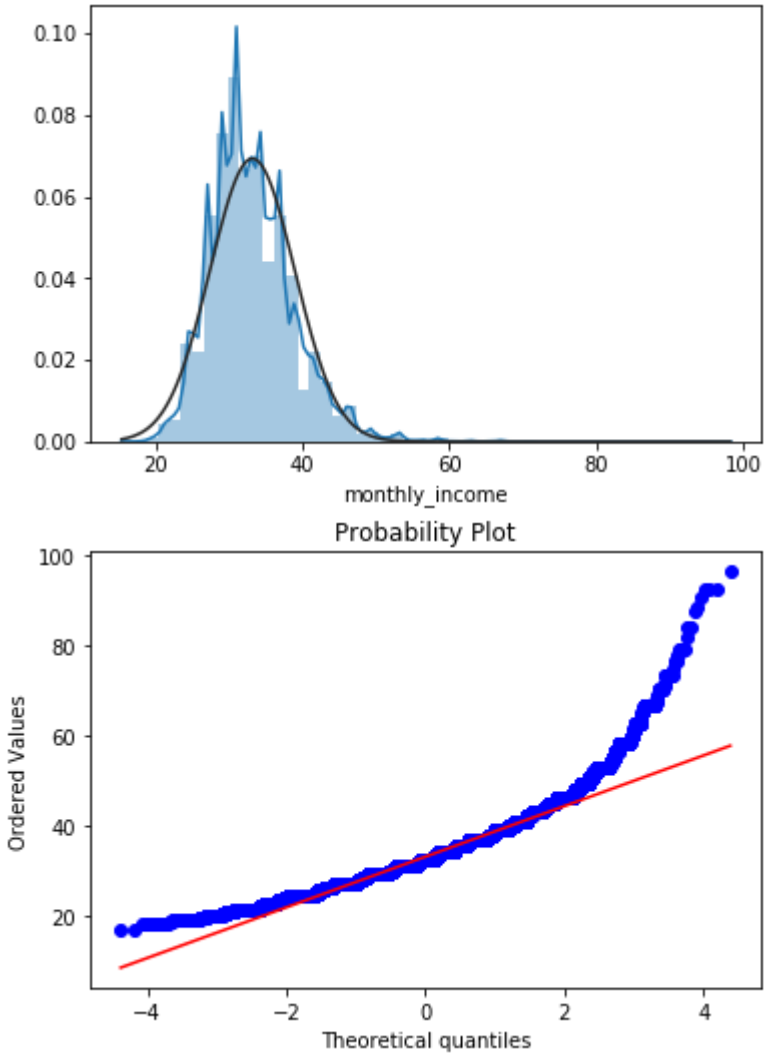


Визуальный анализ гистограммы и графика квантиль-квантиль показывает, что квадратный корень сработал хуже, чем логарифм. Теперь попробуем кубический корень, хотя скорее всего он даст тот же результат, что и квадратный корень.

In[117]:

```
# строим гистограмму распределения и график
# квантиль-квантиль, применив преобразование
# кубическим корнем для переменной monthly_income,
# используем модуль, чтобы не вычислять корни
# отрицательных чисел, и затем учитываем знак числа

sns.distplot(np.sign(train['monthly_income']) *
             (train['monthly_income'].abs() ** (1/3)), fit=norm)
fig = plt.figure()
res = stats.probplot(np.sign(train['monthly_income']) *
                     (train['monthly_income'].abs() ** (1/3)), plot=plt)
```

Out[117]:

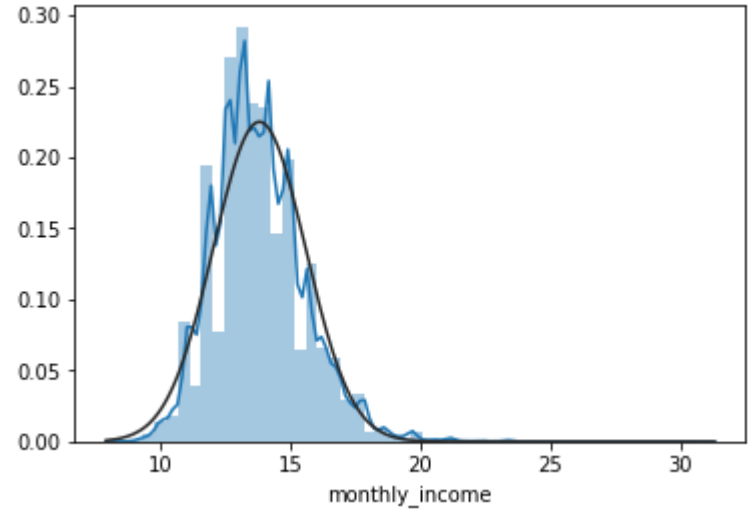


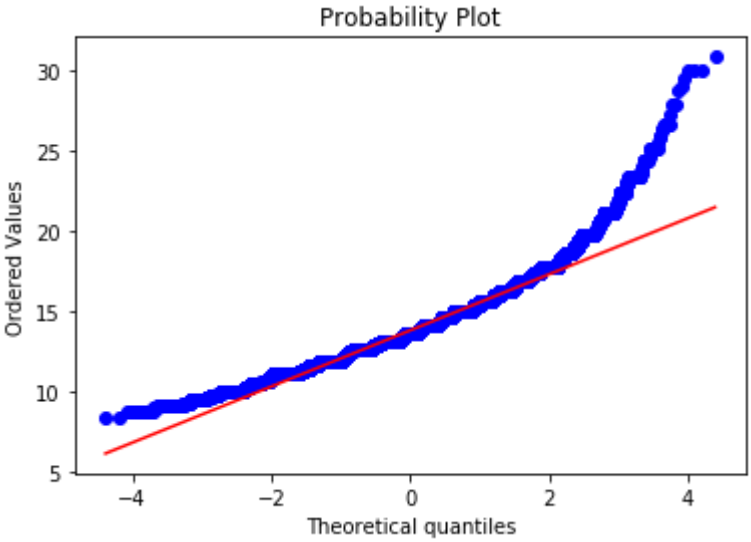
Наш прогноз был верен: кубический корень сработал примерно так же, как квадратный корень.
Теперь применим корень четвертой степени.

In[118]:

```
# строим гистограмму распределения и график
# квантиль-квантиль, применив преобразование
# корнем четвертой степени для переменной monthly_income,
# используем модуль, чтобы не вычислять корни
# отрицательных чисел, и затем учитываем знак числа
sns.distplot(np.sign(train['monthly_income']) *
             (train['monthly_income'].abs() ** (1/4)), fit=norm)
fig = plt.figure()
res = stats.probplot(np.sign(train['monthly_income']) *
                     (train['monthly_income'].abs() ** (1/4)), plot=plt)
```

Out[118]:





Видим, что корень четвертой степени сработал примерно так же, как квадратный и кубический корни.

Делаем вывод, что наилучшим преобразованием для переменной *monhly_income* будет логарифм. Аналогичную процедуру построения гистограмм и графиков квантиль-квантиль с целью подбора наиболее подходящего преобразования нужно повторить для остальных количественных переменных.

В качестве альтернативы выполненным преобразованиям можно применить преобразование Бокса-Кокса с параметром λ, которое выражается следующим образом:

$$y_i^{\lambda} = \begin{cases} \dfrac{y_i^{\lambda} - 1}{\lambda}, \text{если } \lambda \neq 0, \\ \log(y_i), \text{если } \lambda = 0 \end{cases}$$

Параметр λ выбирают, максимизируя логарифм правдоподобия. Кроме того, оптимальное значение параметра подбирают, найдя максимальное значение коэффициента корреляции между квантилями функции нормального распределения и отсортированной преобразованной последовательностью.

Преобразование Бокса-Кокса представляет собой целое семейство преобразований, которое позволяет автоматически находить оптимальную трансформацию для той или иной переменной. При λ = -1 выполняется обратное преобразование При λ = -0,5 – преобразование обратного квадратного корня. При λ = 0,0 – логарифмическое преобразование. При λ = 0,5 – преобразование квадратного корня. При λ = 1 преобразование не выполняется.

Значение параметра λ	Преобразование
λ = -1,0	$x_i(\lambda) = \frac{1}{x_i}$
λ = -0,5	$x_i(\lambda) = \frac{1}{\sqrt{x_i}}$
λ = 0,0	$x_i(\lambda) = \ln(x_i)$
λ = 0,5	$x_i(\lambda) = \sqrt{x_i}$
λ = 2,0	$x_i(\lambda) = x_i^2$

Рис. 74 Таблица преобразований в зависимости от значения параметра λ

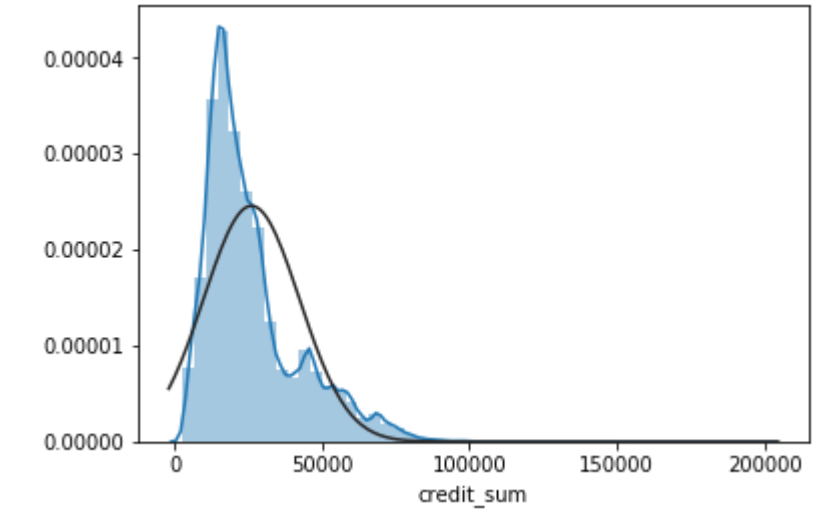
Обратите внимание, что данные, которые мы подвергаем преобразованию Бокса-Кокса, должны быть положительными.

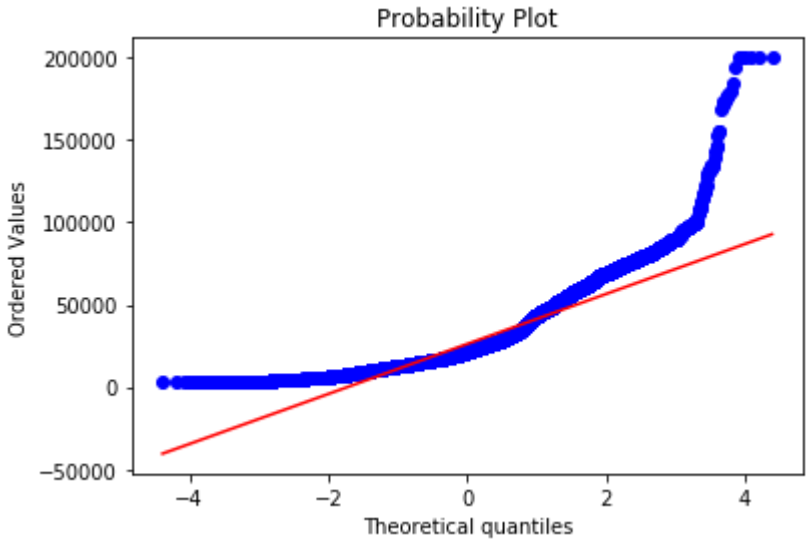
Возьмем переменную *credit_sum*, построим для нее гистограмму распределения и график квантиль-квантиль.

In[119]:

```
# строим гистограмму распределения и график
# квантиль-квантиль для переменной monthly_income
sns.distplot(train['credit_sum'], fit=norm)
fig = plt.figure()
res = stats.probplot(train['credit_sum'], plot=plt)
```

Out[119]:





Давайте найдем значение параметра λ для переменной *credit_sum* с помощью функции `boxcox()` библиотеки SciPy и выполним преобразование этой переменной.

In[120]:

```
# импортируем функцию boxcox
from scipy.stats import boxcox
# выполняем преобразование Бокса-Кокса
transformed, lam = boxcox(train['credit_sum'])
print('Lambda: %f' % lam)
```

Out[120]:

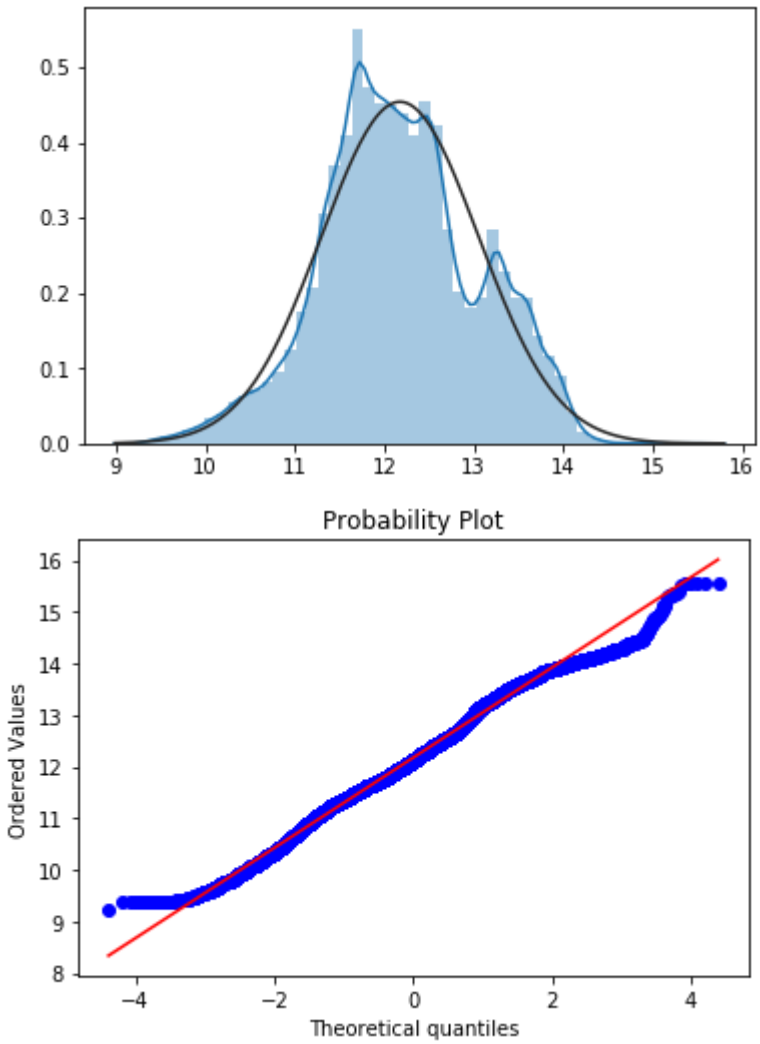
```
Lambda: 0.038149
```

В нашем случае значение параметра равно 0,04, это близко к 0 и будет выполнено логарифмическое преобразование. Давайте построим гистограмму и график Q-Q для переменной *credit_sum*, к которой было применено преобразование Бокса-Кокса.

In[121]:

```
# строим гистограмму распределения и график
# квантиль-квантиль для переменной credit_sum,
# преобразованной с помощью Бокса-Кокса
sns.distplot(transformed, fit=norm)
fig = plt.figure()
res = stats.probplot(transformed, plot=plt)
```

Out[121]:



Видим, что после преобразования распределение переменной *credit_sum* стало в большей степени похожим на нормальное.

ДОПОЛНИТЕЛЬНАЯ ИНФОРМАЦИЯ

Обратите внимание, мы всегда применяем одинаковое значение параметра λ к обучающему и контрольному наборам. По сути, вычисление значение параметра λ – это минимодель, которую мы строим на обучающем наборе и применяем ее к переменным обучающего и контрольного наборов. Нельзя отдельно вычислить значение параметра λ на обучающем наборе, отдельно вычислить значение параметра λ на контрольном наборе и затем использовать это значение для преобразования переменной в соответствующем наборе. Ваш программный код должен выглядеть так:

```
train['credit_sum'], fitted_lambda = boxcox['credit_sum'])
test['credit_sum'] = boxcox(test['credit_sum'], fitted_lambda)
```

Для краткости этап подбора преобразований мы опустим и по его результатам выполним логарифмическое преобразование переменных *monthly_income*, *credit_sum* и *age*.

In[122]:
```
# выполняем логарифмическое преобразование
# указанных переменных
train['credit_sum'] = np.log(train['credit_sum'] + a)
train['monthly_income'] = np.log(train['monthly_income'] + a)
train['age'] = np.log(train['age'] + a)

test['credit_sum'] = np.log(test['credit_sum'] + a)
test['monthly_income'] = np.log(test['monthly_income'] + a)
test['age'] = np.log(test['age'] + a)
```

Приступаем к конструированию новых признаков. Ранее мы говорили о биннинге на основе интервалов одинаковой ширины и квантилей, теперь рассмотрим биннинг по зависимой переменной (оптимальный биннинг). В ходе биннинга нам нужно создать не просто переменные, а переменные, обладающие высокой прогнозной силой. Для задачи бинарной классификации эта прогнозная сила выражается в высокой способности отличать отрицательный класс зависимой переменной от положительного класса. Поэтому при выполнении биннинга ориентируются на два показателя – WoE и IV, которые позволяют судить об этой способности.

Представьте себе, у нас есть переменная *credit_sum* и на ее основе мы хотим создать новую категориальную переменную *credsumcat*. Давайте выясним минимальное и максимальное значения.

In[123]:
```
# взглянем на минимальное и максимальное значения
print(train['credit_sum'].min())
print(train['credit_sum'].max())
```

Out[123]:
```
7.9142526442394505
12.206072650530174
```

Зададим границы категорий и выполним биннинг.

In[124]:
```
# задаем точки, в которых будут находится границы категорий
# будущей переменной credsumcat
bins = [-np.inf, 9.5, 10, 11, np.inf]
# осуществляем биннинг переменной credit_sum и записываем
# результаты в новую переменную credsumcat
train['credsumcat'] = pd.cut(train['credit_sum'], bins).astype('object')
```

Теперь построим простую таблицу сопряженности между переменной *credsumcat* и зависимой переменной *open_account_flg*.

In[125]:
```
# строим таблицу сопряженности credsumcat * open_account_flg
biv = pd.crosstab(train['credsumcat'], train['open_account_flg'])
biv
```

Out[125]:

open_account_flg	0	1
credsumcat		
(-inf, 9.5]	17864	4950
(9.5, 10.0]	32588	7207
(10.0, 11.0]	42824	8112
(11.0, inf]	5227	750

Исходя из этой таблицы, мы можем вычислить «вес» каждой категории. WoE (от weight of evidence) или вес категории вычисляется по формуле:

$$WoE_i = \ln\left(\frac{F_i^0}{F_i^1}\right)$$

где:
i – категория переменной;
ln – натуральный логарифм;
F_i^0 – относительная частота класса 0 (отрицательного класса) в категории;
F_i^1 – относительная частота класса 1 (положительного класса) в категории.
Наблюдения отрицательного класса часто называют не-событиями, а наблюдения положительного класса – событиями.
Обратите внимание, в этой формуле мы используем натуральный логарифм, чтобы разместить полученные категории в логарифмической шкале, которая, как мы уже узнали из предыдущего раздела, является естественной для логистической регрессии.
Вычислим WoE для категории `(-inf, 9.5]`. Относительная частота класса 0 в этой категории равна 17864/(17864+32588+42824+5227) или 17864/98503=0,181. Относительная частота класса 1 равна 4950/(4950+7207+8112+750) или 4950/21019=0,236. Отношение частот равно 0,181/0,236=0,767. Натуральный логарифм этого отношения ln(0,767)=-0,265. WoE измеряет предсказательную силу каждой категории или сгруппированной категории с точки зрения способности отличать класс 0 от класса 1. Отрицательные числа обозначают, что

отдельно взятая категория выделяет бо́льшую пропорцию представителей класса 1, чем представителей класса 0 (что так и есть, 0,236 против 0,181). При работе с WoE, нужно придерживаться четырех правил:

- количество категорий не должно превышать 10;
- каждая категория должна содержать не менее 5% наблюдений;
- категории не должны содержать нулевого количества событий или не-событий;
- пропущенные значения группируются в отдельную категорию.

WoE, как и процент «плохих», должны в достаточной мере отличаться по группам. Группировка выполняется так, чтобы максимизировать разницу между представителями класса 0 или представителями класса 1. Одна из целей работы с WoE – выявить и отделить категории, которые хорошо дифференцируют клиентов. Категории со схожими значениями WoE объединяют, потому что такие категории содержат практически одинаковое количество представителей класса 0 и представителей класса 1 и демонстрируют одинаковое «поведение». Несмотря на то, что абсолютное значение WoE важно, разница между WoE групп играет ключевую роль. Чем больше разница между последующими категориями, тем выше прогнозная сила данной переменной.

В идеале WoE непропущенных значений должно быть монотонным, восходя от отрицательных значений к положительным или наоборот, без смены зависимости на обратную.

Для проверки качества биннинга с помощью WoE строится график значений WoE. Также можно запустить логистическую регрессию с 1 независимой переменной со значениями WoE. Если тангенс угла наклона линии регрессии (попросту говоря, регрессионный коэффициент) не равен 1 или свободный член, определяющий точку пересечения линии регрессии с осью ординат, не равен ln (процент «хороших»/процент «плохих»), то биннинг выполнен некорректно.

Давайте напишем функцию, которая будет вычислять WoE для каждой категории выбранной переменной.

In[126]:
```
# пишем функцию, которая вычисляет WoE для
# каждой категории выбранной переменной,
# при этом используем умножение на 1.0,
# чтобы преобразовать во float и добавляем
# a=0.0001, чтобы избежать деления на 0
def WoE(df, feature, target):
    biv = pd.crosstab(df[feature], df[target].astype('str'))
    a = 0.0001
    WoE = np.log((1.0 * biv['0'] / sum(biv['0']) + a) / (1.0 * biv['1'] / sum(biv['1']) + a))
    return WoE
```

Вычисляем WoE для каждой категории переменной *credsumcat*.

In[127]:
```
# вычисляем WoE для каждой категории переменной credsumcat
WoE(train, 'credsumcat', 'open_account_flg')
```

Out[127]:
```
credsumcat
(-inf, 9.5]    -0.261
(9.5, 10.0]    -0.036
(10.0, 11.0]    0.119
(11.0, inf]     0.396
dtype: float64
```

Итак, мы категоризировали переменную так, чтобы она максимально эффективно отличала один класс от другого. Можно было попробовать увеличить количество категорий. Очень удобно выполнять биннинг по WoE с помощью пакета PyWoE, написанного Денисом Суржко (https://github.com/Densur/PyWoE). Давайте импортируем необходимые классы.

In[128]:
```
# импортируем необходимые классы
from woe import *
```

Сейчас мы должны создать модель – экземпляр класса WoE, задав максимально возможное количество бинов (гиперпараметр `qnt_num`), минимальное количество наблюдений в бине (гиперпараметр `min_block_size`), тип предиктора (параметр `v_type`, значение `'c'` задается для количественного предиктора, значение `'d'` задается для дискретного предиктора), тип зависимой переменной (параметр `t_type`, значение `'c'` задается для количественной зависимой переменной, значение `'b'` задается для бинарной зависимой переменной).

In[129]:
```
# создаем модель - экземпляр класса WoE, задаем максимально возможное
# количество бинов, минимальное количество наблюдений в
# бине, тип предиктора, тип зависимой переменной
woe = WoE(qnt_num=10, min_block_size=10, v_type='c', t_type='b')
```

Обучаем созданную модель, т.е. вычисляем WoE.

In[130]:
```
# обучаем модель - вычисляем WoE
woe.fit(train['credit_sum'], train['open_account_flg'].astype('int'))
```

Теперь применяем обученную модель – выполняем WoE-трансформацию переменной *credit_sum* и выводим информацию о бинах.

In[131]:
```
# выполняем WoE-трансформацию переменной credit_sum
woe.transform(train['credit_sum'])
# выводим информацию о бинах
print(woe.bins)
```

Out[131]:

	mean	bad	obs	good	woe	bins	labels
0	0.274	2976	10871	7895	-0.569	-inf	0
1	0.168	1826	10879	9053	0.056	9.239	1
2	0.151	1639	10849	9210	0.182	9.481	2
3	0.177	1928	10866	8938	-0.011	9.639	3
4	0.200	2168	10866	8698	-0.155	9.758	4
5	0.196	2125	10864	8739	-0.131	9.895	5
6	0.182	1974	10871	8897	-0.039	10.038	6
7	0.162	1759	10878	9119	0.101	10.180	7
8	0.181	1965	10846	8881	-0.036	10.324	8
9	0.127	1380	10866	9486	0.383	10.602	9
10	0.118	1279	10866	9587	0.470	10.869	10

В выводе по каждой полученной категории приводятся следующие показатели:

- `mean` – отношение количество наблюдений положительного класса (`bad`) к общему количеству наблюдений (`obs`);
- `bad` – количество наблюдений положительного класса;
- `obs` – общее количество наблюдений;
- `good` – количество наблюдений отрицательного класса;
- `WoE` – WoE;
- `bins` – нижняя граница категории.

Здесь мы видим нарушение монотонности. С помощью метода `.force_monotonic()` можно улучшить монотонность. Параметр `hypothesis` метода `.force_monotonic()` задает гипотезу о взаимосвязи между предиктором и зависимой переменной (0 – прямая, 1 – обратная).

In[132]:

```
# улучшаем монотонность
woe_monotonic = woe.force_monotonic(hypothesis=0)
print(woe_monotonic.bins)
```

Out[132]:

	mean	bad	obs	good	woe	bins	labels
0	0.274	2976	10871	7895	-0.569	-inf	0
1	0.179	11660	65195	53535	-0.020	9.239	1
2	0.171	3724	21724	18000	0.031	10.180	2
3	0.127	1380	10866	9486	0.383	10.602	3
4	0.118	1279	10866	9587	0.470	10.869	4

С помощью метода `.optimize()` мы можем применить метод деревьев решений CART для биннинга переменной по WoE, максимизирующего определенную метрику качества (для валидации по умолчанию используется 3-блочная перекрестная проверка). Для метода `.optimize()` есть следующие параметры: параметр `criterion` задает критерий разбиения узлов в дереве, параметр `fix_depth` задает дерево фиксированной глубины ($2^{\texttt{fix\_depth}}$ бинов), параметр `cv` задает количество блоков перекрестной проверки, параметр `scoring` задает оптимизируемую метрику, параметр `min_samples_leaf` задает

минимальное количество наблюдений в каждом из оптимизирумых бинов.

In[133]:
```
# выполняем биннинг по WoE с оптимизацией по AUC
# (используется дерево CART)
woe2 = woe.optimize(max_depth=3, scoring='roc_auc', cv=5)
print(woe2.bins)
```

Out[133]:
```
   mean    bad    obs   good    woe   bins labels
0 0.317   1460   4605   3145 -0.777   -inf      0
1 0.243   1509   6206   4697 -0.409  8.893      1
2 0.177  15652  88557  72905 -0.006  9.237      2
3 0.119   2398  20154  17756  0.457 10.652      3
```

Искать оптимальные варианты биннинга можно бесконечно, но давайте вернемся к нашей переменной *credsumcat*. Мы могли бы ее подать на вход модели, но надо убедиться в том, насколько она будет полезна по сравнению с остальными переменными. Для этого используется IV (от information value) или информационное значение. Его можно вычислить для отдельной категории и для всей переменной. Информационное значение для категории вычисляется как разность между относительной частотой класса 0 и относительной частотой класса 1 в данной категории, умноженная на натуральный логарифм отношения этих частот.

$$IV_i = (F_i^0 - F_i^1) \times \ln\left(\frac{F_i^0}{F_i^1}\right)$$

Вычислим информационное значение для категории `(-inf, 9.5]`. Разность между относительными частотами равна 0,181−0,236 = -0,055. Информационное значение равно -0,055 x In(0,767) = -0,055 x (-0,265) = 0,014.

Давайте напишем функцию, которая будет вычислять IV для каждой категории выбранной переменной.

In[134]:
```
# пишем функцию, которая вычисляет IV для
# каждой категории выбранной переменной,
# при этом используем умножение на 1.0,
# чтобы преобразовать во float и добавляем a=0.0001,
# чтобы избежать деления на 0
def IV_cat(df, feature, target):
    biv = pd.crosstab(df[feature], df[target].astype('str'))
    a = 0.0001
    IV_cat = ((1.0 * biv['0'] / sum(biv['0']) + a) -
              (1.0 * biv['1'] / sum(biv['1']) + a)) * np.log(
        (1.0 * biv['0'] / sum(biv['0']) + a) / (1.0 * biv['1'] / sum(biv['1']) + a))
    return IV_cat
```

Теперь вычислим IV для каждой категории переменной *credsumcat*.

In[135]:
```
# вычисляем IV для каждой категории переменной credsumcat
IV_cat(train, 'credsumcat', 'open_account_flg')
```

Out[135]:
```
credsumcat
(-inf, 9.5]    0.014
(9.5, 10.0]    0.000
(10.0, 11.0]   0.006
(11.0, inf]    0.007
dtype: float64
```

Итоговое информационное значение используется для измерения прогнозной силы переменной в целом, для этого информационные значения, вычисленные по каждой категории, складываются.

$$IV = \sum_{i=1}^{k} (F_i^0 - F_i^1) \times \ln\left(\frac{F_i^0}{F_i^1}\right)$$

Информационное значение всегда является положительной величиной. При интерпретации итоговых значений IV руководствуются правилом:

- меньше 0,02 – характеристика не обладает предсказательной способностью;
- от 0,02 до 0,1 – слабая предсказательная способность;
- от 0,1 до 0,3 – средняя предсказательная способность;
- 0,3 и выше – высокая предсказательная способность.

Значения IV более 0,5 обычно вызывают подозрения («слишком хорошо, чтобы быть правдой») и нуждаются в дополнительной проверке.
Сейчас мы напишем функцию, которая будет вычислять итоговое IV для выбранной переменной.

In[136]:
```
# пишем функцию, которая вычисляет итоговое
# IV для выбранной переменной,
# при этом используем умножение на 1.0,
# чтобы преобразовать во float и добавляем a=0.0001,
# чтобы избежать деления на 0
def IV(df, feature, target):
    biv = pd.crosstab(df[feature], df[target].astype('str'))
    a = 0.0001
    IV = sum(((1.0 * biv['0'] / sum(biv['0']) + a) -
              (1.0 * biv['1'] / sum(biv['1']) + a)) * np.log(
        (1.0 * biv['0'] / sum(biv['0']) + a) / (1.0 * biv['1'] / sum(biv['1']) + a)))
    return IV
```

Давайте вычислим итоговое IV для переменной *credsumcat*.

In[137]:
```
# вычисляем итоговое IV для переменной credsumcat
IV(train, 'credsumcat', 'open_account_flg')
```

Out[137]:
```
0.027264421039516484
```

В данном случае переменная *credsumcat* обладает слабой предсказательной способностью.

Давайте удалим переменную *credsumcat*.

In[138]:

```
# удаляем переменную credsumcat
train.drop('credsumcat', axis=1, inplace=True)
```

Однако всецело полагаться на итоговое информационное значение для оценки прогнозной силы переменной не стоит. Итоговое информационное значение зависит от количества категорий/ уникальных значений (чем больше категорий/уникальных значений, тем больше будет IV) и поэтому может быть произвольно высоким. При этом, если WoE сохраняет монотонность как для небольших, так и для крупных категорий, выбирайте более крупные категории. Распространенная ошибка заключается в создании переменной с большим количеством небольших по размеру категорий. Это позволяет получить высокое информационное значение, на основании чего делается вывод (ошибочный), что переменная обладает высокой прогнозной силой. Если использовать много категорий, итоговое информационное значение возрастет, но с практической точки зрения будет бесполезным, потому что будет измерять шум. Поэтому наша задача – получить максимальное информационное значение, при этом выполнив четыре вышеперечисленных правила.

Переменную *credsumcat* мы создавали на основе переменной *credit_month*. Здесь мы взяли переменную *credit_sum* без использования какой-то априорной информации о ее прогнозной силе. Следует помнить, что целесообразно выполнять биннинг сильных переменных, если категоризировать слабую переменную, то и категоризированная переменная будет слабой. Поэтому IV часто используется для сравнительной оценки прогнозной силы переменных. Просто взять количественные переменные и по ним вычислить IV мы не можем, выше мы уже говорили, что IV зависит от количества категорий. Поэтому на практике поступают так: каждую переменную делят на 10 квантилей – групп с примерно одинаковым количеством наблюдений и уже по такой переменной измеряют IV.

Давайте напишем функцию, которая автоматически вычислит IV по всем количественным переменным, у которых больше 10 уникальных значений.

In[139]:

```
# пишем функцию, вычисляющую IV по всем
# количественным предикторам
def numeric_IV(df):
    # создаем список, в который будем записывать IV
    iv_list = []
    # создаем копию датафрейма
    df = df.copy()
    # записываем константу, которую будем добавлять,
    # чтобы избежать деления на 0
    a = 0.0001
    # задаем зависимую переменную
    target = df['open_account_flg'].astype('str')
    # отбираем столбцы, у которых больше 10 уникальных значений
    df = df.loc[:, df.apply(pd.Series.nunique) > 10]
    # из этих столбцов отбираем только количественные
    numerical_columns = df.select_dtypes(include=['number']).columns
    # запускаем цикл, который вычисляет IV по каждой
    # выбранной переменной
    for var_name in numerical_columns:
        # разбиваем переменную на 10 квантилей
        df[var_name] = pd.qcut(df[var_name].values, 10, duplicates='drop').codes
        # строим таблицу сопряженности между категоризированной
        # переменной и зависимой переменной
        biv = pd.crosstab(df[var_name], target)
        # вычисляем IV на основе таблицы сопряженности
        IV = sum(((1.0 * biv['0'] / sum(biv['0']) + a) -
                  (1.0 * biv['1'] / sum(biv['1']) + a)) * np.log(
            (1.0 * biv['0'] / sum(biv['0']) + a) / (1.0 * biv['1'] / sum(biv['1']) + a)))
        # добавляем вычисленное IV в список, где хранятся IV
        iv_list.append(IV)
    # создаем список с названиями столбцов
    col_list = list(numerical_columns)
    # создаем датафрейм с двумя столбцами, в одном - названия переменных,
    # в другом - IV этих переменных
    result = pd.DataFrame({'Название переменной': col_list,'IV': iv_list})
    # добавляем дополнительный столбец "Полезность", задаем строковые значения,
    # которые будут выводится в зависимости от величины IV
    result['Полезность'] = ['Подозрительно высокая' if x > 0.5 else 'Сильная'
                            if x <= 0.5 and x > 0.3 else 'Средняя'
                            if x <= 0.3 and x > 0.1 else 'Слабая'
                            if x <= 0.1 and x > 0.02 else 'Бесполезная'
                            for x in result['IV']]  # по Науму Сиддики
    # возвращаем датафрейм, отсортированный по убыванию IV
    return(result.sort_values(by = 'IV', ascending = False))
```

Применяем функцию `numeric_IV()` к нашему обучающему датафрейму.

In[140]:

```
numeric_IV(train)
```

Out[140]:

	Название переменной	IV	Полезность
6	tariff	0.143	Средняя
1	credit_sum	0.066	Слабая
0	age	0.062	Слабая
2	credit_month	0.038	Слабая
3	score_shk	0.021	Слабая
5	credit_count	0.015	Бесполезная
4	monthly_income	0.006	Бесполезная

Видим, что наиболее сильной является переменная *tariff*, а наиболее слабой – переменная *monthly_income*.
Необходимо отметить, что фильтрация переменных по итоговым значениям IV обычно используется для удаления наиболее слабых переменных. Однако для отбора наиболее сильных переменных в модель логистической регрессии информационные значения использовать не стоит, потому что выбирать переменные нужно на основании того, как они работают вместе, а не на основании того, как они работают по отдельности. Кроме того, следует помнить, информационное значение зависит от объема категорий (по мере увеличения размеров категорий возрастает и информационное значение). Если WoE сохраняет монотонность как для небольших, так и для крупных категорий, выбирайте более крупные категории.
Кроме того, мы можем выполнить интерактивный биннинг с учетом WoE и IV с помощью обработчика **Конечные классы** бесплатной версии программы **Deductor** от компании BaseGroup Labs https://basegroup.ru/deductor/download.

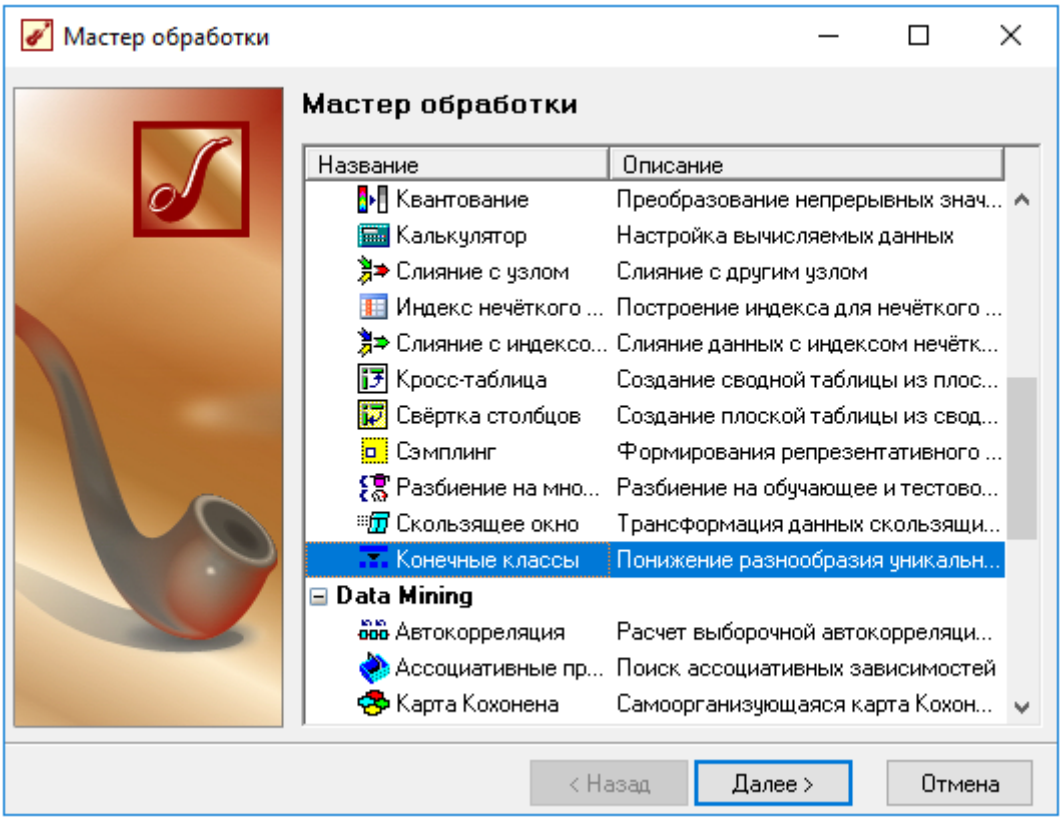
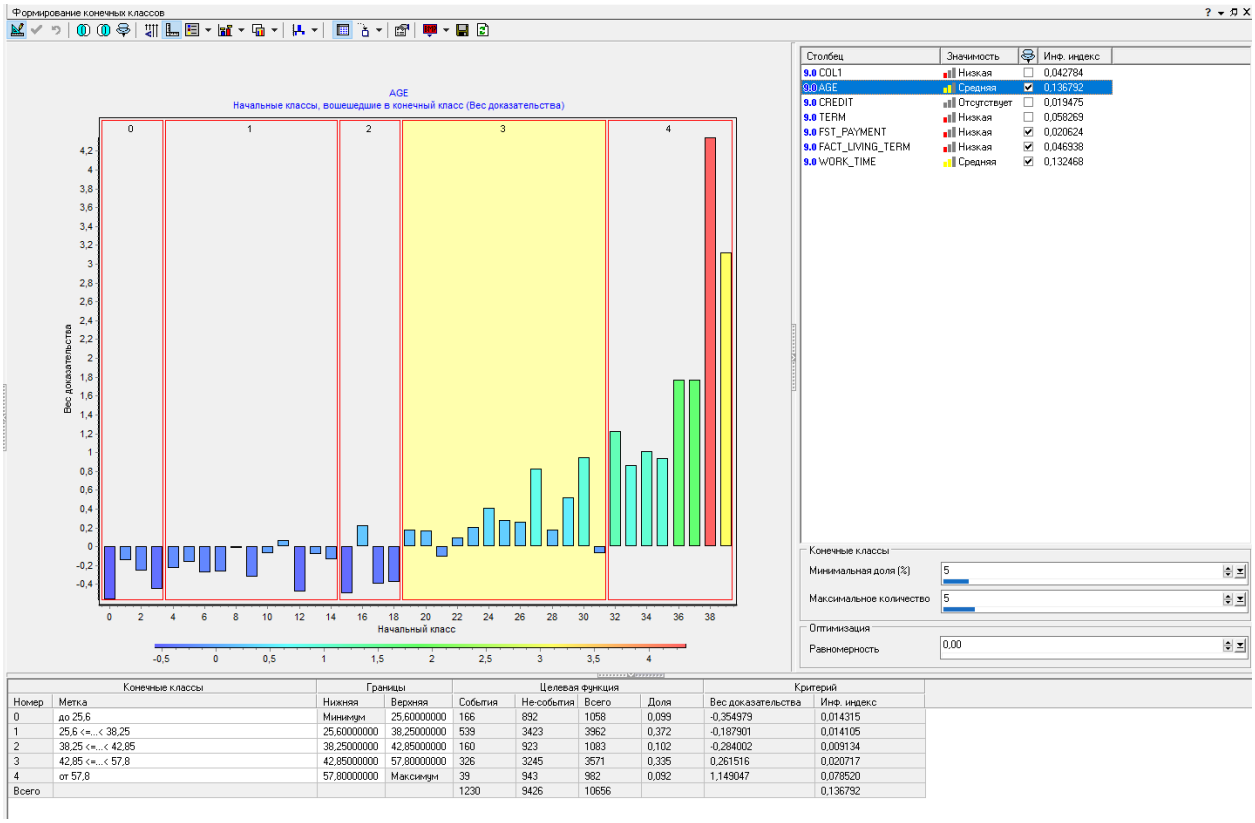


Рис. 75 Обработчик **Конечные классы** программы **Deductor**

В нижеприведенном окне (рис. 76) осуществляем биннинг, включив интерактивный режим (кнопка с изображением линейки-треугольника).



Конечные классы		Границы		Целевая функция				Критерий	
Номер	Метка	Нижняя	Верхняя	События	Не-события	Всего	Доля	Вес доказательства	Инф. индекс
0	до 25,6	Минимум	25,60000000	166	892	1058	0,099	-0,354979	0,014315
1	25,6 <=...< 38,25	25,60000000	38,25000000	539	3423	3962	0,372	-0,187901	0,014105
2	38,25 <=...< 42,85	38,25000000	42,85000000	160	923	1083	0,102	-0,284002	0,009134
3	42,85 <=...< 57,8	42,85000000	57,80000000	326	3245	3571	0,335	0,261516	0,020717
4	от 57,8	57,80000000	Максимум	39	943	982	0,092	1,149047	0,078520
Всего				1230	9426	10656			0,136792

Рис. 76 Режим интерактивного биннинга в программе **Deductor**

Для отбора наиболее сильных переменных можно воспользоваться важностями предикторов, вычисляемыми с помощью случайного леса. Речь идет о важности на основе усредненного уменьшения неоднородности, которую предложил вычислять Лео Брейман.

Алгоритм вычисления важности на основе усредненного уменьшения неоднородности будет выглядеть так:

1. Для каждого дерева случайного леса вычисляем важность – сумму уменьшений неоднородности узлов (улучшений) на всех ветвлениях, связанных с данным предиктором. На рис. 77 показан пример вычисления важности.

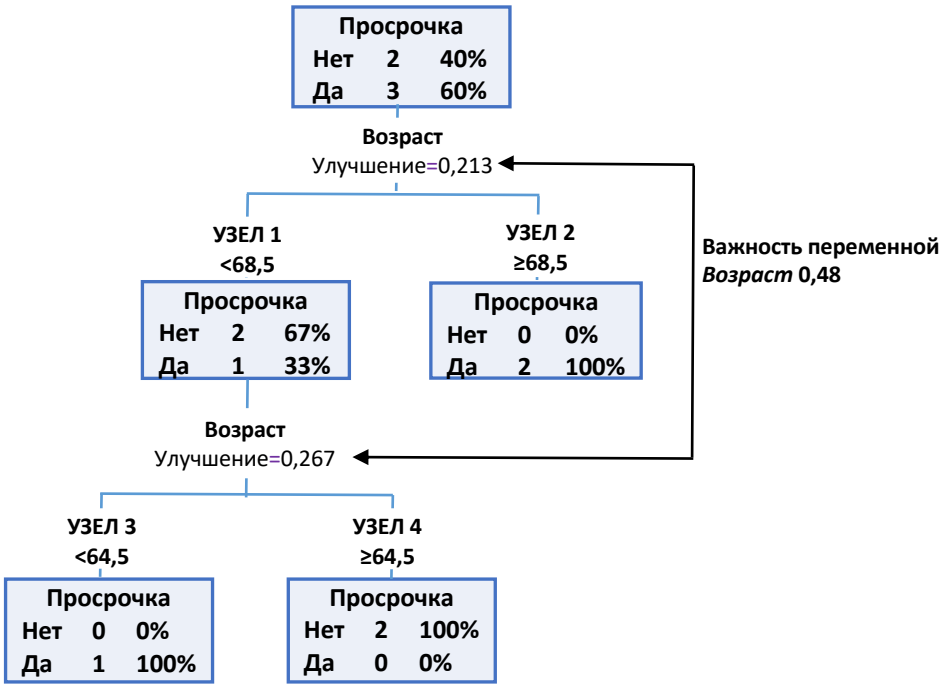


Рис. 77 Вычисление важности для отдельного дерева

2. Итоговую сумму уменьшений неоднородности, полученную по всем деревьям, усредняем путем деления на общее количество деревьев.
3. Вышеописанные шаги повторяем для всех остальных предикторов.
Наиболее важный предиктор – тот, который дает наибольшее усредненное уменьшение неоднородности (для деревьев классификации – уменьшение меры Джини, для деревьев регрессии – уменьшение суммы квадратов остатков).

Нетрудно увидеть недостаток подхода. По сути важность складывается из частоты использования переменной в качестве предиктора разбиения, то есть чаще наиболее важными будут переменные, по которым можем быть рассмотрено больше вариантов разбиения и у них больше шансов стать предиктором разбиения. Поэтому наиболее важными переменными чаще будут переменные с большим количеством уникальных значений, традиционно это количественные переменные и категориальные переменные с большим количеством уровней.

Итак, давайте вычислим важности на основе усредненного уменьшения неоднородности, построив модель случайного леса.

In[141]:

```
# выполняем дамми-кодирование
train_dum = pd.get_dummies(train)
# создаем обучающий массив меток
y_training = train_dum.loc[:, 'open_account_flg_1']
# удаляем из будущего массива признаков результаты
# дамми-кодирования зависимой переменной -
# open_account_flg_0 и open_account_flg_1
train_dum.drop(['open_account_flg_0', 'open_account_flg_1'], axis=1, inplace=True)
# создаем обучающий массив признаков
X_training = train_dum.loc[:, 'age':'ind_1']
```

```
# создаем экземпляр класса RandomForestClassifier
forest = RandomForestClassifier(n_estimators=50, max_depth=17, random_state=152, n_jobs=-1)
# строим модель
forest.fit(X_training, y_training)
# создаем объект с названиями предикторов
feat_labels = X_training.columns
# создаем объект со значениями важностей, вычисленными моделью forest
importances = forest.feature_importances_
# задаем сортировку значений важности и сопоставляем названия предикторов важностям
indices = np.argsort(importances)[::-1]
for f in range(X_training.shape[1]):
    print('%2d) %-*s %f' % (f + 1, 35,
                            feat_labels[indices[f]],
                            importances[indices[f]]))
```

Out[141]:

```
1) credit_sum                         0.098495
2) score_shk                          0.092383
3) age                                0.077462
. . . . . . . . . . . . . . . . . . . . . . . .
```

Согласно случайному лесу среди количественных переменных наиболее важными предикторами стали *credit_sum*, *score_shk* и *age*. Однако обратите внимание, для экономии времени вычислений мы задали ансамбль из 50 деревьев, большее количество деревьев может дать более надежные оценки важности переменных.

Принимая во внимание недостатки важности на основе усредненного уменьшения неоднородности, Лео Брейман предложил алгоритм вычисления важности предиктора на основе усредненного уменьшения качества прогнозирования. Для задачи классификации вычисляем усредненное уменьшении правильности – количестве правильно классифицированных наблюдений от общего количества наблюдений. Для задачи регрессии вычисляется усредненное увеличение среднеквадратичной ошибки. Рассмотрим подробнее вычисление важности на основе усредненного уменьшения правильности.

1. Для каждого дерева классификации случайного леса берем out-of-bag выборку (наблюдения, не попавшие в бутстреп-выборку, по которой строилось данное дерево).

Допустим, у нас есть набор данных из 10 наблюдений. Наблюдение может принадлежать либо классу N, либо классу P. Мы построили ансамбль из 5 деревьев. По каждому из 5 деревьев получаем out-of-bag выборку. Например, для дерева I out-of-bag выборкой будут наблюдения 2, 4, 5, 6.

Исх. выборка	1	2	3	4	5	6	7	8	9	10
Фактический класс	**N**	**P**	**P**	**N**	**N**	**P**	**N**	**N**	**N**	**P**

											Out-of-bag выборки
Дерево I											
Б-выборка I	10	9	7	8	1	3	9	10	10	7	2 4 5 6
Дерево II											
Б-выборка II	4	8	5	8	3	9	2	6	1	6	7 10
Дерево III											
Б-выборка III	6	2	6	10	2	10	3	6	5	1	4 7 8 9
Дерево IV											
Б-выборка IV	6	7	8	10	6	10	9	10	8	2	1 3 4 5
Дерево V											
Б-выборка V	5	8	1	8	5	7	10	1	10	9	2 3 4 6

Рис. 78 Определение out-of-bag выборок

2. Для каждой out-of-bag выборки вычисляем правильность. Считаем количество раз, когда спрогнозированный класс для наблюдения out-of-bag выборки совпал с фактическим, и делим на размер out-of-bag выборки.
Например, для out-of-bag выборки 1 мы получаем 3 верных ответа и делим на 4 наблюдения, получаем правильность 3 / 4 = 0,75.

Out-of-bag выборка 1

	Предиктор 1	Предиктор 2	Предиктор 3	Предиктор 4	Фактический класс	Спрогнозированный класс до пермутации	Верные ответы до пермутации (отмечены X)
2	1	2	11	101	**P**	**P**	**X**
4	2	3	12	102	**N**	**P**	
5	3	5	13	103	**N**	**N**	**X**
6	4	7	14	104	**P**	**P**	**X**
							правильность 3/4=0,75

. . . . . .

Рис. 79 Вычисление правильности для каждой out-of-bag выборки

3. В каждой out-of-bag выборке осуществляем случайную перестановку (пермутацию) значений интересующего нас предиктора и вычисляем правильность в каждой out-of-bag выборке с перестановленными значениями предиктора.
Например, для out-of-bag выборки 1 после пермутации предиктора 2 мы получаем 2 верных ответа и делим на 4 наблюдения, получаем правильность 2 / 4 = 0,5.

Out-of-bag выборка 1

	Предиктор 1	Предиктор 2	Предиктор 3	Предиктор 4	Фактический класс	Спрогнозированный класс после пермутации	Верные ответы после пермутации (отмечены X)
2	1	5	11	101	**P**	**N**	
4	2	7	12	102	**N**	**P**	
5	3	2	13	103	**N**	**N**	**X**
6	4	3	14	104	**P**	**P**	**X**
							правильность 2/4=0.5

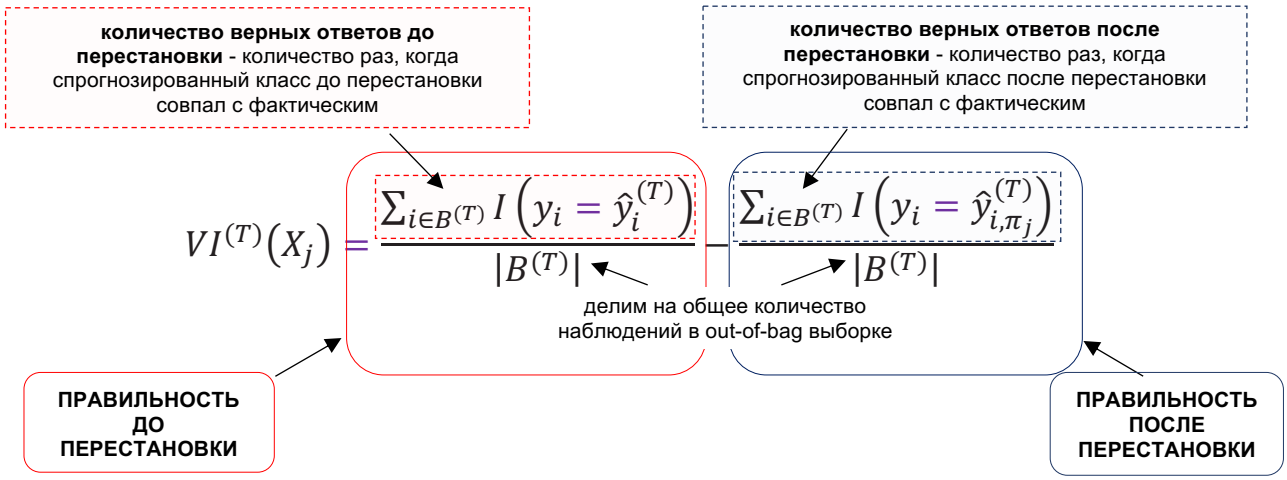
. . . . . .

Рис. 80 Вычисление правильности для каждой out-of-bag выборки после пермутации

4. Вычисляем разность между правильностью с исходными значениями предиктора и правильностью с перестановленными значениями предиктора в каждой out-of-bag выборке.
5. Суммируем разности по out-of-bag выборкам и делим на количество деревьев. Получаем сырое значение важности переменной.
6. Сырое значение важности переменной нормализуем путем деления на стандартную ошибку.
7. Повторяем шаги 2-5 для всех остальных предикторов.

На рис. 81 приводится математический аппарат вычисления важности на основе усредненного уменьшения правильности.

1. Вычисляем важность переменной как уменьшение правильности после перестановки в каждой out-of-bag выборке



где:
$VI^{(T)}(X_j)$ – это важность переменной X_j для дерева T;
$|B^{(T)}|$ – это out-of-bag выборка для дерева T;
y_i – фактический класс зависимой переменной;
$\hat{y}_i^{(T)}$ – спрогнозированный класс зависимой переменной перед перестановкой значений предиктора;
$\hat{y}_{i,\pi_j}^{(T)}$– спрогнозированный класс зависимой переменной после перестановки значений предиктора.
Обратите внимание, что $VI^{(T)}(X_j) = 0$, если X_j не находится в дереве T.

2. Вычисляем сырую важность переменной по ансамблю, просуммировав важности по всем out-of-bag выборкам и усреднив по всем деревьям

$$VI(X_j) = \frac{\sum_{T=1}^{N} VI^{(T)}(X_j)}{N}$$

3. Вычисляем нормализованную важность переменной по ансамблю

$$z_j = \frac{VI(X_j)}{\frac{\sigma}{\sqrt{N}}}$$

Рис. 81 Математический аппарат вычисления сырой и нормализованной важности предиктора на основе усредненного уменьшения правильности

Логическое объяснение алгоритма перестановки состоит в следующем: случайно переставляя значения предиктора, мы разрушаем взаимосвязь между ним и зависимой переменной. При использовании перестановленных значений предиктора (вместе с неперестановленными значениями остальных предикторов) для прогнозирования/ классификации по out-of-bag данным правильность существенно уменьшается (а среднеквадратичная ошибка существенно увеличивается), если между исходным предиктором и зависимой переменной была взаимосвязь. Поэтому чем больше уменьшение

правильности (увеличение среднеквадратичной ошибки), тем важнее предиктор. Важность на основе усредненного уменьшения правильности/увеличения среднеквадратичной ошибки в результате перестановок еще называют пермутированной важностью.

Поскольку предполагается, что после пермутации оценка качества уменьшается, мы ожидаем, что разница между оценкой качества до пермутации и оценкой качества после пермутации будет положительным числом. Однако если предиктор не обладает прогнозной силой, то можно получить отрицательную разницу, когда после случайной пермутации значений предиктора модель показывает более высокое качество. Поэтому отрицательные пермутированные важности будут говорят о том, что соответствующие предикторы не обладают прогнозной силой. Однако обнаружив предикторы с небольшими или отрицательными значениями важностей, не спешите их удалять, посмотрите, как они будут работать, если включить дополнительные переменные. Необходимо понимать, что вычисляемая важность показывает, как данный предиктор работает не по отдельности, а в сочетании с другими переменными. Может оказаться, что предиктор, который является маловажным при совместном использовании с одними переменными, в сочетании с другими переменными станет важным. Кроме того, обратите внимание, что при наличии высоко коррелированных предикторов обе метрики важности не способны определить релевантные переменные.

Мы можем вычислить пермутированные важности предикторов с помощью функции `feature_importance_permutation()` библиотеки `mlxtend`, созданной Себастьяном Рашкой – автором книги-бестселлера «Python и машинное обучение». Обратите внимание, что в данной библиотеке идея Лео Бреймана немного модифицирована, вместо OOB-выборок используется отложенная выборка:

1. Берем модель, которая была построена на обучающем наборе данных.
2. Оцениваем качество модели на независимом наборе данных (например, на контрольном наборе данных) и записываем полученную оценку качества в качестве базовой.
3. Для каждого предиктора i:
   - случайным образом выполняем пермутацию значений предиктора i в исходном наборе данных;
   - записываем оценку качества модели, построенной на наборе с пермутированным предиктором;
   - вычисляем важность признака как разность между базовой оценкой качества (полученной на этапе 2) и оценкой качества, полученной на наборе с пермутированным предиктором.

In[142]:
```
# воспользуемся функцией feature_importance_permutation
# библиотеки mlxtend от Себастьяна Рашки
from mlxtend.evaluate import feature_importance_permutation
```

Теперь подготовим контрольный набор данных и вычислим пермутированные важности.

In[143]:
```
# выполняем дамми-кодирование контрольного набора
test_dum = pd.get_dummies(test)
# создаем контрольный массив меток
y_testing = test_dum.loc[:, 'open_account_flg_1']
# удаляем из будущего массива признаков результаты
# дамми-кодирования зависимой переменной -
# open_account_flg_0 и open_account_flg_1
test_dum.drop(['open_account_flg_0', 'open_account_flg_1'], axis=1, inplace=True)
# создаем контрольный массив признаков
X_testing = test_dum.loc[:, 'age':'ind_1']
```

In[144]:
```
# вычисляем пермутированные важности
imp_vals, _ = feature_importance_permutation(
    predict_method=forest.predict,
    X=X_testing.values,
    y=y_testing.values,
    metric='accuracy',
    num_rounds=10,
    seed=1)
```

In[145]:
```
# выводим пермутированные важности
indices = np.argsort(imp_vals)[::-1]
for f in range(X_training.shape[1]):
    print('%2d) %-*s %f' % (f + 1, 60,
                            feat_labels[indices[f]],
                            imp_vals[indices[f]]))
```

```
 1) ind_0                     0.002001
 2) ind_1                     0.001579
 3) age                       0.001341
 4) credit_month              0.001324
 5) credit_sum                0.001150
 6) tariff                    0.00111
. . . . . .
```

Мы видим, что наиболее важными предикторами стали *ind_0, ind_1*, *age*, *credit_sum* и *tariff*.

До этого момента мы выполняли биннинг количественных предикторов для создания новых признаков, однако новые признаки можно получить на основе биннинга категориальных предикторов. Очень часто для этих целей использует метод деревьев решений CHAID. Мы определяем зависимую переменную, строим дерево CHAID с одним интересующим нас предиктором, полученные узлы становятся новыми категориями предиктора.

Перед началом работы CHAID необходимо преобразовать все имеющиеся количественные предикторы в категориальные переменные. Обычно их разбивают на 10 категорий одинакового объема.

CHAID приступает к построению дерева, итеративно применяя к каждому узлу, начиная с корневого, процедуры объединения категорий, расщепления узла и проверки правил остановки.

Этап 1. Объединение категорий

1. Для каждого предиктора с числом категорий больше двух[16] алгоритм ищет пару категорий с наименее значимыми различиями по зависимой переменной, т.е. пару категорий, для которых после применения соответствующего статистического критерия получено наибольшее *p*-значение. Выбор статистического критерия определяется типом шкалы зависимой переменной. Для номинальной зависимой переменной используется критерий хи-квадрат Пирсона. Алгоритм строит двухвходовую таблицу сопряженности с категориями предиктора в качестве строк и категориями зависимой переменной в качестве столбцов. Он проверяет нулевую гипотезу о том, что категории предиктора не отличаются друг от друга с точки зрения распределения категорий зависимой переменной. Для количественной зависимой переменной используется F-критерий. Алгоритм осуществляет однофакторный дисперсионный анализ и проверяет нулевую гипотезу о том, что средние значения зависимой переменной для различных категорий предиктора не различаются между собой.
2. Найдя наибольшее *p*-значение для пары категорий, алгоритм сравнивает его с заданным уровнем значимости для объединения категорий.

Если *p*-значение:

- меньше или равно заданному уровню значимости для объединения категорий – алгоритм переходит к вычислению скорректированных *p*-значений для полученного набора категорий (шаг 3);
- больше уровня значимости для объединения категорий – эта пара объединяется в отдельную составную категорию, в результате формируется новый набор категорий предиктора и процесс начинается заново с поиска пары категорий с наибольшим *p*-значением.

(Опциональный шаг) Если новая составная категория состоит из трех и более исходных категорий, алгоритм находит внутри этой составной категории наилучшее бинарное расщепление, которое дает наименьшее *p*-значение. Алгоритм выполняет бинарное расщепление, если его *p*-значение не превышает уровня значимости для разбиения объединенных категорий.

---

[16] Если предиктор имеет одну категорию, он исключается из анализа. Если предиктор имеет две категории, происходит переход к шагу 3.

3. Получив сформированный набор категорий предиктора, алгоритм для категориальной зависимой переменной вновь строит двухвходовую таблицу сопряженности с категориями предиктора в качестве строк и категориями зависимой переменной в качестве столбцов, а для количественной зависимой переменной вновь выполняет однофакторный дисперсионный анализ. В результате алгоритм вычисляет скорректированное *p*-значение критерия хи-квадрат или F-критерия как исходное *p*-значение, умноженное на поправку Бонферонни. Поправка Бонферрони представляет собой корректировку уровня значимости в зависимости от числа возможных способов, с помощью которых исходные категории предиктора могут быть объединены в итоговые категории.

Этап 2. Расщепление узла

После вычисления скорректированных *p*-значений для итоговых наборов категорий по всем предикторам алгоритм переходит к этапу расщепления узла.

1. На этапе расщепления алгоритм выбирает, какой предиктор обеспечит наилучшее разбиение узла. Для этого предиктор должен иметь наименьшее скорректированное *p*-значение (т.е. должен являться наиболее статистически значимым).
2. Найдя предиктор с наименьшим скорректированным *p*-значением, алгоритм сравнивает его с заданным уровнем значимости для расщепления.

Если *p*-значение:

- меньше или равно заданному уровню значимости для расщепления – алгоритм разбивает узел с использованием данного предиктора;
- больше заданного уровня значимости для расщепления, то алгоритм не расщепляет узел и узел рассматривается как терминальный.

Этап 3. Остановка

Алгоритм проверяет, нужно ли прекратить построение дерева, в соответствии со следующими правилами остановки.

1. Если узел стал однородным, то есть все наблюдения в узле имеют одинаковые значения зависимой переменной, узел не разбивается.
2. Если текущая глубина дерева достигает заданной пользователем максимальной глубины дерева, процесс построения дерева останавливается.
3. Если количество наблюдений в родительском узле меньше заданного пользователем минимума наблюдений в родительском узле, узел не разбивается.

4. Если минимальное абсолютное количество наблюдений в терминальном узле меньше заданного пользователем минимума наблюдений в терминальном узле, узел не разбивается.
5. Если минимальная относительная частота наблюдений в терминальном узле меньше заданной пользователем минимальной относительной частоты наблюдений в терминальном узле, узел не разбивается.

При выполнении биннинга одного категориального предиктора у нас будет выполняться только первый этап – этап объединения категорий.

В среде Python биннинг на основе метода CHAID можно выполнить с помощью пакета `CHAID`. Этот пакет можно установить в Anaconda Promt с помощью команды `pip install CHAID`.

In[146]:
```
# воспользуемся пакетом CHAID для биннинга
# категориальной переменной tariff_id,
# предварительно установив его в Anaconda Promt
# с помощью команды pip install CHAID
from CHAID import Tree
# задаем название предиктора
independent_variable = 'tariff_id'
# задаем название зависимой переменной
dep_variable = 'open_account_flg'
# создаем словарь, где ключом будет название
# предиктора, а значением - тип переменной
dct = {independent_variable: 'nominal'}
```

In[147]:
```
# строим дерево CHAID и выводим его
tree = Tree.from_pandas_df(train, dct, dep_variable, max_depth=1)
tree.print_tree()
```

Out[147]:
```
([], {0: 98503.0, 1: 21019.0}, (tariff_id, p=0.0, score=4912.276428995799, groups=[['1_0', '1_9'], ['1_1', '1_17'],
['1_16', '1_5', '1_2', '1_7', '1_94'], ['1_18'], ['1_19', '1_4'], ['1_21', '1_22', '1_23', '1_99'], ['1_24', '1_25',
'1_6'], ['1_3', '1_41', '1_91'], ['1_32'], ['1_43', '1_44']]), dof=9))
|-- (['1_0', '1_9'], {0: 5069.0, 1: 359.0}, <Invalid Chaid Split> - the max depth has been reached)
|-- (['1_1', '1_17'], {0: 39687.0, 1: 9244.0}, <Invalid Chaid Split> - the max depth has been reached)
|-- (['1_16', '1_5', '1_2', '1_7', '1_94'], {0: 8504.0, 1: 940.0}, <Invalid Chaid Split> - the max depth has been re
ached)
|-- (['1_18'], {0: 2.0, 1: 21.0}, <Invalid Chaid Split> - the minimum parent node size threshold has been reached)
|-- (['1_19', '1_4'], {0: 8060.0, 1: 1053.0}, <Invalid Chaid Split> - the max depth has been reached)
|-- (['1_21', '1_22', '1_23', '1_99'], {0: 928.0, 1: 36.0}, <Invalid Chaid Split> - the max depth has been reached)
|-- (['1_24', '1_25', '1_6'], {0: 23847.0, 1: 3934.0}, <Invalid Chaid Split> - the max depth has been reached)
|-- (['1_3', '1_41', '1_91'], {0: 1879.0, 1: 774.0}, <Invalid Chaid Split> - the max depth has been reached)
|-- (['1_32'], {0: 6792.0, 1: 4110.0}, <Invalid Chaid Split> - the max depth has been reached)
+-- (['1_43', '1_44'], {0: 3735.0, 1: 548.0}, <Invalid Chaid Split> - the max depth has been reached)
```

Первая строка вывода начинается с информации о частотах классов зависимой переменной `{0: 98503.0, 1: 21019.0}`, значение `p` показывает статистическую значимость, `score` показывает значение хи-квадрат, `groups` показывает полученные категории. Далее приводятся узлы - укрупненные категории и распределение классов зависимой переменной в каждом узле. Также выводятся предупреждения, сообщающие, какое из правил остановки сработало: для большинства узлов приводится предупреждение `the max depth has been reached` – `достигнута максимальная глубина`, это неудивительно, ведь мы выбрали глубину 1 (дерево будет иметь один уровень, лежащий ниже корневого узла). Для одного из узлов выведено предупреждение `the minimum parent node size threshold has been reached` – `достигнут порог по минимальному размеру`

`родительского узла`, это обусловлено тем, что по умолчанию минимальное количество наблюдений в разбиваемом узле (регулируется параметром `min_parent_node_size`) должно быть не менее 30, а наш узел содержит всего 23 наблюдения. Укрупненные категории необходимо создавать, придерживаясь тех же самых правил, что и при биннинге количественных переменных:

- количество категорий не должно превышать 10;
- каждая категория должна содержать не менее 5% наблюдений;
- категории не должны содержать нулевого количества событий или не-событий.

Для оценки прогнозной силы укрупненных категорий и новой переменной можно также воспользоваться WoE и IV.

Теперь создаем уже знакомые нам переменные *paym* и *pti*, а также квадраты некоторых количественных признаков.

In[148]:

```
# создаем переменную paym, которая
# является отношением выданной суммы кредита
# (credit_sum) к сроку кредита (credit_month),
# то есть ежемесячной суммой кредита
train['paym'] = train['credit_sum'] / train['credit_month']
test['paym'] = test['credit_sum'] / test['credit_month']

# заменяем бесконечные значения на 1
train['paym'].replace([np.inf, -np.inf], 1, inplace=True)
test['paym'].replace([np.inf, -np.inf], 1, inplace=True)

# создаем переменную pti, которая является
# отношением ежемесячной суммы кредита
# (paym) к ежемесячному заработку
# (monthly_income)
train['pti'] = train['paym'] / train['monthly_income']
test['pti'] = test['paym'] / test['monthly_income']

# заменяем бесконечные значения на 1
train['pti'].replace([np.inf, -np.inf], 1, inplace=True)
test['pti'].replace([np.inf, -np.inf], 1, inplace=True)

# создаем новые переменные, возведя некоторые
# количественные переменные в квадрат
train['tariff_sq'] = train['tariff']**2
test['tariff_sq'] = test['tariff']**2

train['age_sq'] = train['age']**2
test['age_sq'] = test['age']**2

train['credit_sum_sq'] = train['credit_sum']**2
test['credit_sum_sq'] = test['credit_sum']**2

train['score_sq'] = train['score_shk']**2
test['score_sq'] = test['score_shk']**2

train['income_sq'] = train['monthly_income']**2
test['income_sq'] = test['monthly_income']**2

train['credit_month_sq'] = train['credit_month']**2
test['credit_month_sq'] = test['credit_month']**2

train['credit_count_sq'] = train['credit_count']**2
test['credit_count_sq'] = test['credit_count']**2
```

Напишем функцию, которая автоматически разобьет количественные переменные на нужное количество бинов и вычислит IV.

In[149]:

```
# пишем функцию, выполняющую биннинг
def user_bin(df, number):
    # увеличиваем максимальную ширину столбца
    pd.set_option('max_colwidth', 800)
    # задаем список, где будут хранится IV
    iv_list = []
    # задаем список, где будет хранится информация о количестве бинов
    bins_list = []
    # задаем список, где будет хранится информация о бинах
    groups_list = []
    # записываем константу, которую будем добавлять,
    # чтобы избежать деления на 0
    a = 0.0001
    # задаем зависимую переменную
    target = df['open_account_flg'].astype('str')
    # отбираем столбцы, у которых больше 10 уникальных значений
    df = df.loc[:, df.apply(pd.Series.nunique) > 10]
    # из этих столбцов отбираем только количественные
    numerical_columns = df.select_dtypes(include=['number']).columns
    # запускаем цикл, который вычисляет IV по каждой
    # выбранной переменной
    for var_name in numerical_columns:
        # фиксируем количество бинов
        num = number
        # создаем точки разбиения
        bins = np.linspace(df[var_name].min(), df[var_name].max(), num)
        # округляем значения точек разбиения
        rounded_bins = np.round(bins, 2)
        # создаем бины
        groups = np.digitize(df[var_name], rounded_bins)
        # строим таблицу сопряженности между категоризированной
        # переменной и зависимой переменной
        biv = pd.crosstab(groups, target)
        # вычисляем IV на основе таблицы сопряженности
        IV = sum(((1.0 * biv['0'] / sum(biv['0']) + a) -
                  (1.0 * biv['1'] / sum(biv['1']) + a)) * np.log(
            (1.0 * biv['0'] / sum(biv['0']) + a) / (1.0 * biv['1'] / sum(biv['1']) + a)))
        # добавляем вычисленное IV в список, где хранятся IV
        iv_list.append(IV)
        # добавляем информацию о количестве бинов в список, где хранится
        # информация о количестве бинов
        bins_list.append(num)
        # добавляем бины в список, где хранится
        # информация о бинах
        groups_list.append(rounded_bins)
    # создаем список с названиями столбцов
    col_list = list(numerical_columns)
    # создаем датафрейм с четырьмя столбцами, в первом - названия переменных,
    # во втором - бины, в третьем - IV, в четвертом - количество бинов
    result = pd.DataFrame({'Переменная' : col_list,
                           'Бины': groups_list,
                           'IV' : iv_list,
                           'Количество_бинов' : bins_list})
    # возвращаем датафрейм, отсортированный по убыванию IV
    return(result.sort_values(by = 'IV', ascending = False))
```

Давайте воспользуемся нашей функцией.

In[150]:
```
# разбиваем все количественные переменные
# на 10 категорий и смотрим IV
user_bin(train, 10)
```

Out[150]:

	Переменная	Бины	IV	Количество_бинов
6	tariff	[1.0, 1.11, 1.22, 1.33, 1.44, 1.55, 1.66, 1.77, 1.88, 1.99]	0.211	10
9	tariff_sq	[1.0, 1.33, 1.66, 1.99, 2.32, 2.64, 2.97, 3.3, 3.63, 3.96]	0.146	10
2	credit_month	[3.0, 6.67, 10.33, 14.0, 17.67, 21.33, 25.0, 28.67, 32.33, 36.0]	0.093	10
10	age_sq	[8.35, 9.45, 10.54, 11.63, 12.72, 13.81, 14.9, 15.99, 17.08, 18.17]	0.080	10
0	age	[2.89, 3.04, 3.2, 3.35, 3.5, 3.65, 3.81, 3.96, 4.11, 4.26]	0.076	10
1	credit_sum	[7.91, 8.39, 8.87, 9.34, 9.82, 10.3, 10.78, 11.25, 11.73, 12.21]	0.065	10
7	paym	[0.24, 0.65, 1.05, 1.46, 1.87, 2.27, 2.68, 3.09, 3.49, 3.9]	0.062	10
11	credit_sum_sq	[62.64, 72.23, 81.82, 91.42, 101.01, 110.61, 120.2, 129.8, 139.39, 148.99]	0.060	10
8	pti	[0.02, 0.06, 0.1, 0.15, 0.19, 0.23, 0.27, 0.31, 0.35, 0.39]	0.037	10
12	score_sq	[0.0, 0.14, 0.28, 0.42, 0.57, 0.71, 0.85, 0.99, 1.13, 1.27]	0.023	10
14	credit_month_sq	[9.0, 152.0, 295.0, 438.0, 581.0, 724.0, 867.0, 1010.0, 1153.0, 1296.0]	0.021	10
3	score_shk	[0.0, 0.13, 0.25, 0.38, 0.5, 0.63, 0.75, 0.88, 1.0, 1.13]	0.021	10
4	monthly_income	[8.52, 9.09, 9.67, 10.25, 10.83, 11.4, 11.98, 12.56, 13.13, 13.71]	0.007	10
13	income_sq	[72.54, 85.37, 98.19, 111.02, 123.84, 136.67, 149.49, 162.32, 175.14, 187.97]	0.006	10
5	credit_count	[0.0, 2.33, 4.67, 7.0, 9.33, 11.67, 14.0, 16.33, 18.67, 21.0]	0.003	10
15	credit_count_sq	[0.0, 49.0, 98.0, 147.0, 196.0, 245.0, 294.0, 343.0, 392.0, 441.0]	0.002	10

Ниже мы создадим новые переменные, поэкспериментировав с нашей функцией.

In[151]:
```
# задаем точки, в которых будут находится границы категорий
# будущей переменной monthcat
bins = [-np.inf, 7.95, 9.6, 11.25, 13.725, 14.55,
        17.85, 20.325, 23.625, 24.45, 26.925, np.inf]
# осуществляем биннинг переменной credit_month и записываем
# результаты в новую переменную monthcat
train['monthcat'] = pd.cut(train['credit_month'], bins).astype('object')
test['monthcat'] = pd.cut(test['credit_month'], bins).astype('object')
```

In[152]:
```
# задаем точки, в которых будут находится границы категорий
# будущей переменной agecat
bins = [-np.inf, 3.09, 3.28, 3.48, 3.67, 3.87, 4.07, np.inf]
# осуществляем биннинг переменной age и записываем
# результаты в новую переменную agecat
train['agecat'] = pd.cut(train['age'], bins).astype('object')
test['agecat'] = pd.cut(test['age'], bins).astype('object')
```

In[153]:
```
# задаем точки, в которых будут находится границы категорий
# будущей переменной credsumcat
bins = [-np.inf, 8.3, 8.69, 9.08, 9.47, 9.87, 10.26,
        10.65, 11.04, 11.43, 11.82, np.inf]
# осуществляем биннинг переменной credit_sum и записываем
# результаты в новую переменную credsumcat
train['credsumcat'] = pd.cut(train['credit_sum'], bins).astype('object')
test['credsumcat'] = pd.cut(test['credit_sum'], bins).astype('object')
```

In[154]:

```
# задаем точки, в которых будут находится границы категорий
# будущей переменной paymcat
bins = [-np.inf, 0.57, 0.91, 1.24, 1.57, 1.9, 2.24,
        2.57, 2.9, 3.23, 3.57, np.inf]
# осуществляем биннинг переменной paym и записываем
# результаты в новую переменную paymcat
train['paymcat'] = pd.cut(train['paym'], bins).astype('object')
test['paymcat'] = pd.cut(test['paym'], bins).astype('object')
```

Кроме того, создадим переменную *match*, если количество просроченных кредитов (*overdue_credit_count*) совпадает с количеством кредитов (*credit_count*), переменная принимает значение 1, если не совпадает, принимает значение 0.

In[155]:

```
# создаем переменную match, если количество просроченных
# кредитов совпадает с количеством кредитов, переменная
# принимает значение 1, если не совпадает, принимает значение 0
train['match'] = np.where(
    train['overdue_credit_count'] == train['credit_count'], 1, 0).astype('object')
test['match'] = np.where(
    test['overdue_credit_count'] == test['credit_count'], 1, 0).astype('object')
```

Теперь пришло время вспомнить о второй важной предпосылке – едином масштабе измерения переменных. Если не привести переменные к единому масштабу, то прогноз будут определять переменные, имеющие наибольшую дисперсию. Допустим, у нас есть доход в долларах и возраст. Поскольку дисперсия у дохода в долларах обычно намного больше, чем у возраста, доход будет доминировать в решении. Взгляните на рисунок. На вертикальной (доход) и горизонтальной (возраст) оси установлен одинаковый масштаб (единичный). Из рисунка ясно, что при измерении дохода в долларах (а не в тысячах или десятках тысяч долларов) решения будут почти полностью определяться доходом.

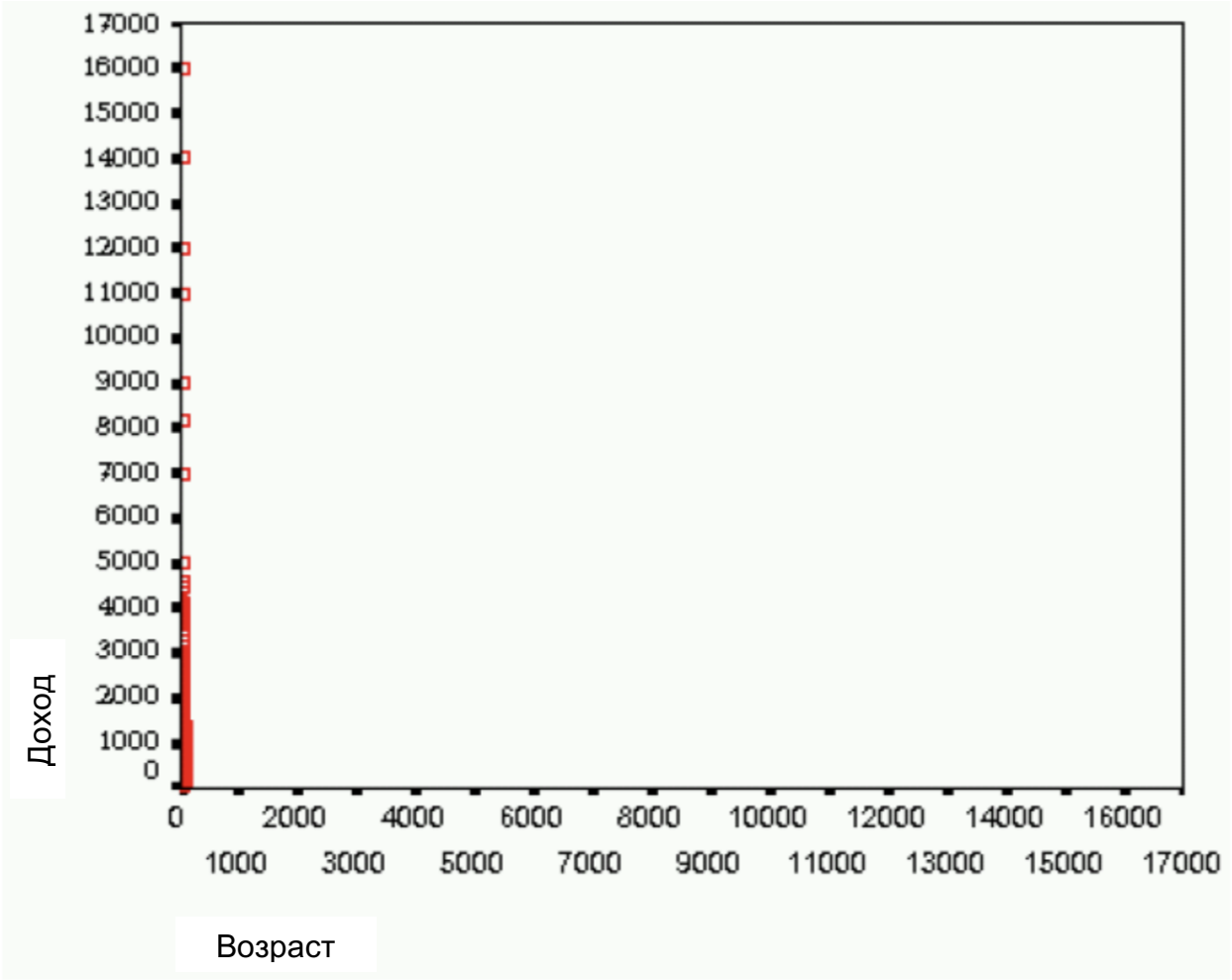


Рис. 82 Вычисления в условиях разных масштабов

Единый масштаб позволяет сравнивать коэффициенты при предикторах и корректно применять регуляризацию (стандартизация обеспечивает равные условия коэффициентам при сжатии). Разумеется, в масштабировании нуждаются лишь количественные переменные, потому что категориальные переменные будут записаны в виде дамми-переменных со значениями 0 или 1.

Самое простое масштабирование подразумевает, что из каждого значения переменной мы вычтем среднее значение и полученный результат разделим на стандартное отклонение. Обратите внимание, процедуру масштабирования также часто называют стандартизацией.

$$\frac{x_i - mean(x)}{stdev(x)}$$

В итоге мы получаем распределение со средним 0 и стандартным отклонением 1. Именно это и делает класс `StandardScaler`.

Для этого нужно импортировать класс `StandardScaler`, который осуществляет предварительную обработку, а затем создать его экземпляр:

```
# импортируем класс StandardScaler
from sklearn.preprocessing import StandardScaler
# создаем экземпляр класса StandardScaler
scaler = StandardScaler()
```

Затем с помощью метода .fit() мы строим модель scaler на обучающих данных. В отличие от обычных моделей машинного обучения, при вызове метода .fit() scaler работает с данными (X_train), а ответы (y_train) не используются:

```
# обучаем модель
scaler.fit(X_train)
```

В данном случае модель подразумевает под собой вычисление среднего значения и стандартного отклонения для каждой переменной обучающего набора.
Чтобы применить модель, которую мы только что подогнали, к нашим данным, то есть фактически *отмасштабировать* (*scale*) обучающие и контрольные данные, мы воспользуемся методом .transform(). Метод .transform() используется в scikit-learn, когда модель возвращает новое представление данных.

```
# преобразовываем данные
X_tr_scaled = scaler.transform(X_train)
X_tst_scaled = scaler.transform(X_test)
```

Фактически метод .transform() из каждого значения переменной обучающего и контрольного наборов вычитает среднее значение соответствующей переменной в обучающем наборе и делит на стандартное отклонение этой переменной, также взятое по обучающему набору.
Обратите внимание, что, масштабируя переменную в обучающем и контрольном наборах, мы всегда используем среднее и стандартное отклонение переменной в обучающем наборе. Нельзя отдельно вычислить среднее значение и стандартное отклонение переменной на обучающем наборе, а затем отдельно вычислить среднее значение и стандартное отклонение переменной на контрольном наборе и потом использовать эти значения для преобразования переменной в соответствующем наборе.
StandardScaler хорошо работает, когда данные подчиняются нормальному распределению.
Еще можно воспользоваться классом MinMaxScaler. Из каждого значения переменной мы вычитаем минимальное значение и полученный результат делим на ширину диапазона (разницу между минимальным и максимальным значениями).

$$\frac{x_i - \min(x)}{\max(x) - \min(x)}$$

В итоге мы сжимаем значения переменных в диапазон от 0 до 1 (или от -1 до 1, если есть отрицательные значения). Этот способ работает лучше в тех случаях, когда `StandardScaler` дает не очень хороший результат. Если распределение не является нормальным или стандартное отклонение является очень маленьким, `MinMaxScaler` сработает лучше.
Однако обратите внимание, что `MinMaxScaler` чувствителен к выбросам, поэтому если данные содержат выбросы, класс `RobustScaler` может дать лучший результат.
Класс `RobustScaler` похож на класс `MinMaxScaler`, но вместо ширины диапазона использует межквартильный размах и поэтому устойчив к выбросам. В основе выполняемого преобразования лежит следующая формула:

$$\frac{x_i - Q_1(x)}{Q_3(x) - Q_1(x)}$$

В данном случае мы выполним обычную стандартизацию вручную.

In[156]:
```
# выделяем количественные переменные в отдельный список
num_cols = [c for c in train.columns if train[c].dtype.name != 'object']
# создаем копию обучающего набора
train_copy = train.copy()
# выполняем стандартизацию
for i in num_cols:
    train[i] = (train[i] - train[i].mean()) / train[i].std()
    test[i] = (test[i] - train_copy[i].mean()) / train_copy[i].std()
```

Перед выполнением дамми-кодирования вновь убеждаемся в отсутствии пропусков и одинаковом количестве переменных в датафреймах.

In[157]:
```
# убеждаемся в отсутствии пропусков в переменных, а также
# в одинаковом количестве переменных в обучающей
# и контрольной выборках
print(train.info())
print(test.info())
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 119522 entries, 53397 to 149270
Data columns (total 30 columns):
gender                  119522 non-null object
age                     119522 non-null float64
marital_status          119522 non-null object
job_position            119522 non-null object
credit_sum              119522 non-null float64
credit_month            119522 non-null float64
tariff_id               119522 non-null object
score_shk               119522 non-null float64
education               119522 non-null object
living_region           119522 non-null object
monthly_income          119522 non-null float64
credit_count            119522 non-null float64
overdue_credit_count    119522 non-null float64
open_account_flg        119522 non-null object
tariff                  119522 non-null float64
ind                     119522 non-null object
```

```
paym                    119522 non-null float64
pti                     119522 non-null float64
tariff_sq               119522 non-null float64
age_sq                  119522 non-null float64
credit_sum_sq           119522 non-null float64
score_sq                119522 non-null float64
income_sq               119522 non-null float64
credit_month_sq         119522 non-null float64
credit_count_sq         119522 non-null float64
monthcat                119522 non-null object
agecat                  119522 non-null object
credsumcat              119522 non-null object
paymcat                 119522 non-null object
match                   119522 non-null object
dtypes: float64(17), object(13)
memory usage: 28.3+ MB
None
<class 'pandas.core.frame.DataFrame'>
Int64Index: 51224 entries, 0 to 170745
Data columns (total 30 columns):
gender                  51224 non-null object
age                     51224 non-null float64
marital_status          51224 non-null object
job_position            51224 non-null object
credit_sum              51224 non-null float64
credit_month            51224 non-null float64
tariff_id               51224 non-null object
score_shk               51224 non-null float64
education               51224 non-null object
living_region           51224 non-null object
monthly_income          51224 non-null float64
credit_count            51224 non-null float64
overdue_credit_count    51224 non-null float64
open_account_flg        51224 non-null object
tariff                  51224 non-null float64
ind                     51224 non-null object
paym                    51224 non-null float64
pti                     51224 non-null float64
tariff_sq               51224 non-null float64
age_sq                  51224 non-null float64
credit_sum_sq           51224 non-null float64
score_sq                51224 non-null float64
income_sq               51224 non-null float64
credit_month_sq         51224 non-null float64
credit_count_sq         51224 non-null float64
monthcat                51224 non-null object
agecat                  51224 non-null object
credsumcat              51224 non-null object
paymcat                 51224 non-null object
match                   51224 non-null object
dtypes: float64(17), object(13)
memory usage: 12.1+ MB
None
```

Убедившись, что все в порядке, выполняем дамми-кодирование.

In[158]:

```
# печатаем названия столбцов до и после
# дамми-кодирования
print("Исходные переменные:\n", list(train.columns), "\n")
train_dumm = pd.get_dummies(train)
print("Переменные после get_dummies:\n", list(train_dumm.columns))

print("Исходные переменные:\n", list(test.columns), "\n")
test_dumm = pd.get_dummies(test)
print("Переменные после get_dummies:\n", list(test_dumm.columns))
```

Затем создаем массивы для моделирования.

In[159]:
```
# создаем обучающий и контрольный массивы меток
y_tr = train_dumm.loc[:, 'open_account_flg_1']
y_tst = test_dumm.loc[:, 'open_account_flg_1']
# удаляем из будущих массивов признаков результаты
# дамми-кодирования зависимой переменной -
# open_account_flg_0 и open_account_flg_1
train_dumm.drop(['open_account_flg_0', 'open_account_flg_1'], axis=1, inplace=True)
test_dumm.drop(['open_account_flg_0', 'open_account_flg_1'], axis=1, inplace=True)
# создаем обучающий и контрольный массивы признаков
X_tr = train_dumm.loc[:, 'age':'match_1']
X_tst = test_dumm.loc[:, 'age':'match_1']
```

Строим модель логистической регрессии.

In[160]:
```
# строим модель логистической регрессии
logreg = LogisticRegression().fit(X_tr, y_tr)
print("AUC на обучающей выборке: {:.3f}".
      format(roc_auc_score(y_tr, logreg.predict_proba(X_tr)[:, 1])))
print("AUC на контрольной выборке: {:.3f}".
      format(roc_auc_score(y_tst, logreg.predict_proba(X_tst)[:, 1])))
```

Out[160]:
```
AUC на обучающей выборке: 0.758
AUC на контрольной выборке: 0.760
```

При желании с помощью атрибута `coef_` и `intercept_` можно взглянуть на константу и регрессионные коэффициенты.

In[161]:
```
# запишем коэффициенты, предварительно округлив до 3-го знака,
# и названия предикторов в отдельные объекты
coef = np.round(logreg.coef_, 3)
feat_labels = X_tr.columns

# запишем свободный член (константу) в отдельный объект, при этом
# переводим массив, полученный с помощью атрибута intercept_,
# в скаляр с округлением до 3-го знака
intercept = np.round(np.asscalar(logreg.intercept_), 3)

# печатаем название "Константа"
print("Константа:", intercept)
# печатаем название "Регрессионные коэффициенты"
print("Регрессионные коэффициенты:")
# для удобства сопоставим каждому названию
# предиктора соответствующий коэффициент
for c, feature in zip(coef[0], feat_labels):
    print(feature, c)
```

Out[161]:
```
Константа: -0.148
Регрессионные коэффициенты:
age -2.081
credit_sum -1.701
credit_month 0.489
score_shk -0.41
monthly_income -0.407
credit_count 0.042
overdue_credit_count 0.046
. . . . . .
```

Теперь давайте вычислим экспоненциальные коэффициенты.

```
In[162]:
# вычислим экспоненциальные коэффициенты
# и запишем их в отдельный объект
exp_coef = np.round(np.exp(coef), 3)

# печатаем название "Константа"
print("Константа:", intercept)
# печатаем название "Экспоненциальные коэффициенты"
print("Экспоненциальные коэффициенты:")
# для удобства сопоставим каждому названию
# предиктора соответствующий коэффициент
for c, feature in zip(exp_coef[0], feat_labels):
    print(feature, c)

Out[162]:
Константа: -0.148
Экспоненциальные коэффициенты:
age 0.125
credit_sum 0.183
credit_month 1.631
score_shk 0.664
monthly_income 0.666
credit_count 1.043
overdue_credit_count 1.047
. . . . . .
```

Третьей предпосылкой построения логистической регрессии является отсутствие взаимосвязи между предикторами. Наличие сильной корреляционной взаимосвязи между предикторами называется мультиколлинеарностью. Задача логистической регрессии – аккуратно измерить вклад каждого признака в прогноз. Вспомним, что регрессионный коэффициент показывает изменение натурального логарифма шанса события при изменении предиктора на единицу своего измерения при том, что все остальные предикторы фиксированы. Последнее дополнение очень важно, ведь если интересующий нас предиктор будет сильно скоррелирован с другим, то при его изменении на единицу измерения будет меняться и другой предиктор, т.е. будет привнесено влияние другого предиктора.

Мультиколлинеарность в модели будет проявляться в следующем:

- сильный разброс оценок коэффициентов регрессии (большие положительные и большие отрицательные оценки коэффициентов регрессии, значительно выше 1,0 по модулю);
- резкое изменение оценок коэффициентов регрессии при добавлении или удалении предиктора;
- неправильный знак перед коэффициентом регрессии;
- присутствие в модели большого количества статистически незначимых оценок коэффициентов регрессии.

Вариабельность оценок коэффициентов регрессии не позволяет судить о степени влияния предиктора на зависимую переменную и построить статистически устойчивую и точную модель.

Самый эффективный способ борьбы с мультиколлинеарностью – увеличение размера выборки. Увеличивая размер выборки, мы уменьшаем дисперсию регрессионных коэффициентов и увеличиваем их статистическую значимость (стандартные отклонения регрессионных

коэффициентов обратно пропорциональны $\sqrt{n}$, где n – это размер выборки). Вообще говоря, проблема мультиколлинеарности характерна для небольших выборок и во многом снимается при работе с выборками достаточного объема. Точнее, увеличение объема выборки помогает бороться с последствиями наличия коллинеарных предикторов: ранее незначимые коэффициенты могут стать значимыми, хотя корреляции между предикторами останутся. Более того, ранее незначимые корреляции между предикторами с ростом размера выборки тоже могут становиться значимыми.

Второй способ борьбы с мультиколлинеарностью заключается в том, чтобы объединить высоко коррелированные переменные в одну составную переменную. Составная (инструментальная, агрегатная) переменная – переменная, полученная в результате математических операций над двумя и более независимыми переменными. Например, между двумя переменными *Сумма кредитных обязательств* и *Доход* может наблюдаться сильная корреляция, но при этом они несут ценность для модели и удалить их нельзя. Можно сформировать новую переменную, сформулировав ее как (*Сумма кредитных обязательств / Доход*) * 100%. Другой пример составной переменной – отношение суммы платежей к сумме кредитов.

Третий способ – преобразование переменных с помощью метода главных компонент. Метод главных компонент преобразует набор коррелированных исходных переменных в набор некоррелированных переменных (компонент).

Четвертый и наиболее часто используемый способ – это снижение разброса оценок коэффициентов регрессии с помощью методов регуляризации.

Как и в обычной логистической регрессии, при использовании регуляризации максимизируется правдоподобие, но с дополнительным ограничением на оценки регрессионных коэффициентов (вводится штраф за слишком большие оценки коэффициентов). В методе лассо штрафной член представляет собой произведение штрафного коэффициента λ на сумму модулей регрессионных коэффициентов. Метод лассо называют L1-регуляризацией.

$$\|w\|_1 = \sum_i |w_i|$$

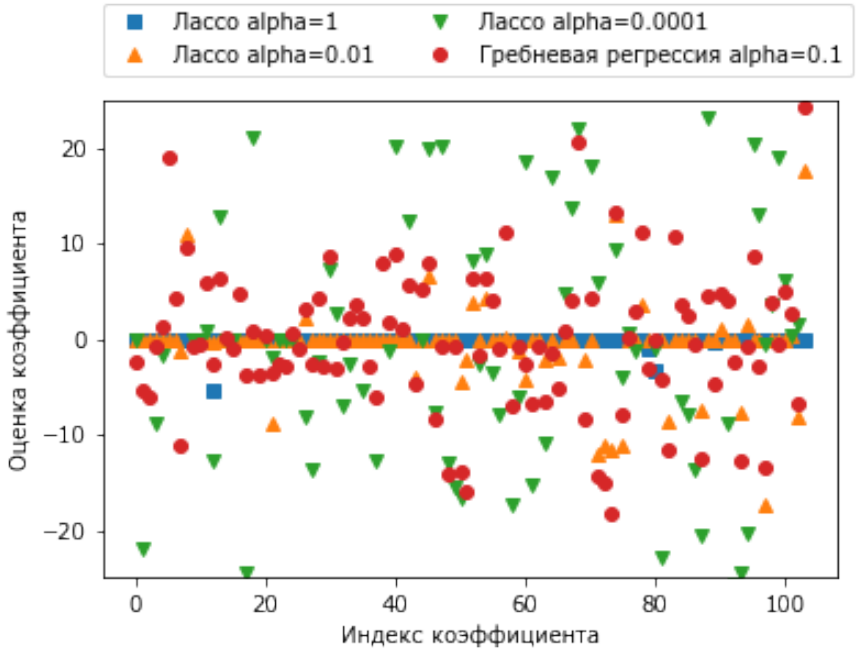
$$max_w l(w) - \lambda \|w\|_1$$

Метод лассо устанавливает некоторые регрессионные коэффициенты точно равными нулю и тем самым осуществляет отбор переменных. Исключение малоинформативных признаков позволяет снизить переобучение и тем самым может повысить качество модели.

В методе гребневой регрессии штрафной член представляет собой произведение штрафного коэффициента λ на сумму квадратов регрессионных коэффициентов. Метод гребневой регрессии называют также L2-регуляризацией.

$$\|w\|_2 = \sqrt{\sum_i w_i^2}$$

$$max_w l(w) - \lambda\|w\|_2^2$$



Лассо обращает некоторые коэффициенты точно в ноль.

Рис. 83 Сравнение лассо-регрессии и гребневой регрессии

Лассо обычно дает хороший результат, если у нас имеется небольшое количество значимых параметров, а остальные близки к 0 (т.е. когда у нас лишь небольшое количество предикторов влияют на зависимую переменную). Гребневая регрессия хорошо работает, когда у нас много значимых регрессионных коэффициентов с одинаковыми значениями (т.е. большинство предикторов влияют на зависимую переменную).
Чтобы ответить на вопрос, почему метод лассо устанавливает некоторые регрессионные коэффициенты точно равными нулю, достаточно посмотреть на рис. 84.

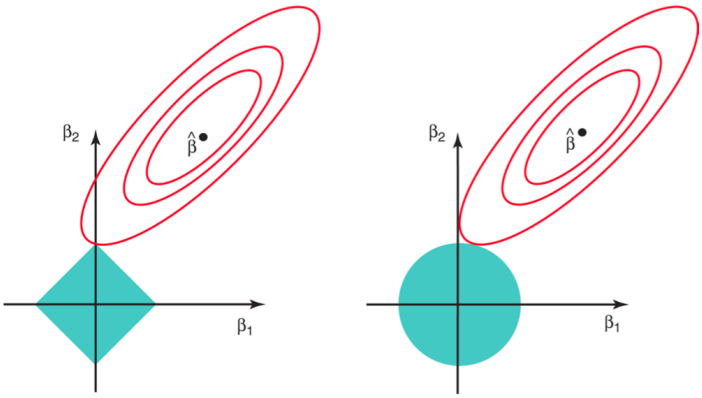


Рис. 84 Контурные диаграммы функций потерь и ограничений для лассо-регрессии (слева) и гребневой регрессии (справа)

Оценки коэффициентов, полученные методами лассо и гребневой регрессии, соответствуют первой точке, в которой эллипс касается области ограничений (залитые зеленым цветом ромб и круг показывают области ограничений лассо и гребневой регрессии соответственно). Поскольку область ограничений у гребневой регрессии имеет круглую форму без острых углов, эта точка касания обычно не приходится на координатную ось, и поэтому оценки коэффициентов гребневой регрессии всегда будут отличными от нуля. Однако область ограничений лассо имеет углы на каждой из осей, в связи с чем эллипс часто будет пересекать область ограничений на той или иной оси. Когда это происходит, один из коэффициентов принимает нулевое значение. В задачах более высокой размерности нулю могут быть равны одновременно несколько оценок коэффициентов. На рисунке пересечение происходит при $\beta_1 = 0$, и поэтому итоговая модель включает только β_2.

Мы рассмотрели простой случай с $p = 2$. При $p = 3$ область ограничений гребневой регрессии превращается в сферу, а область ограничений лассо становится полиэдром. Когда $p > 3$, область ограничений гребневой регрессии становится гиперсферой, а область ограничений лассо-регрессии – политопом. Однако ключевая идея, изображенная на рисунке, остается той же. В частности, метод лассо выполняет отбор переменных при $p > 2$ благодаря наличию острых углов у полиэдра или политопа.

В некоторых сферах, где интерпретация модели является главным требованием (медицина, кредитный скоринг), регуляризация может быть запрещена, поскольку не позволяет однозначно интерпретировать регрессионные коэффициенты.

Допустим, у нас есть набор данных Breast Cancer. Нужно по значениям предикторов предсказать тип опухоли (доброкачественная или злокачественная). На рисунке по оси абсцисс отложены названия предикторов, а по оси ординат – оценки коэффициентов.

Изучив график более внимательно, можно увидеть интересный эффект, произошедший с третьим коэффициентом, коэффициентом признака «mean perimeter». При C=100 и C=1 коэффициент отрицателен, тогда как при C=0,001 коэффициент положителен, при этом его оценка больше, чем оценка коэффициента при C=1.
Однако изменение знака коэффициента для признака «mean perimeter» означает, что в зависимости от рассматриваемого штрафа увеличение значения «mean perimeter» может увеличивать или уменьшать вероятность диагностирования злокачественной опухоли.

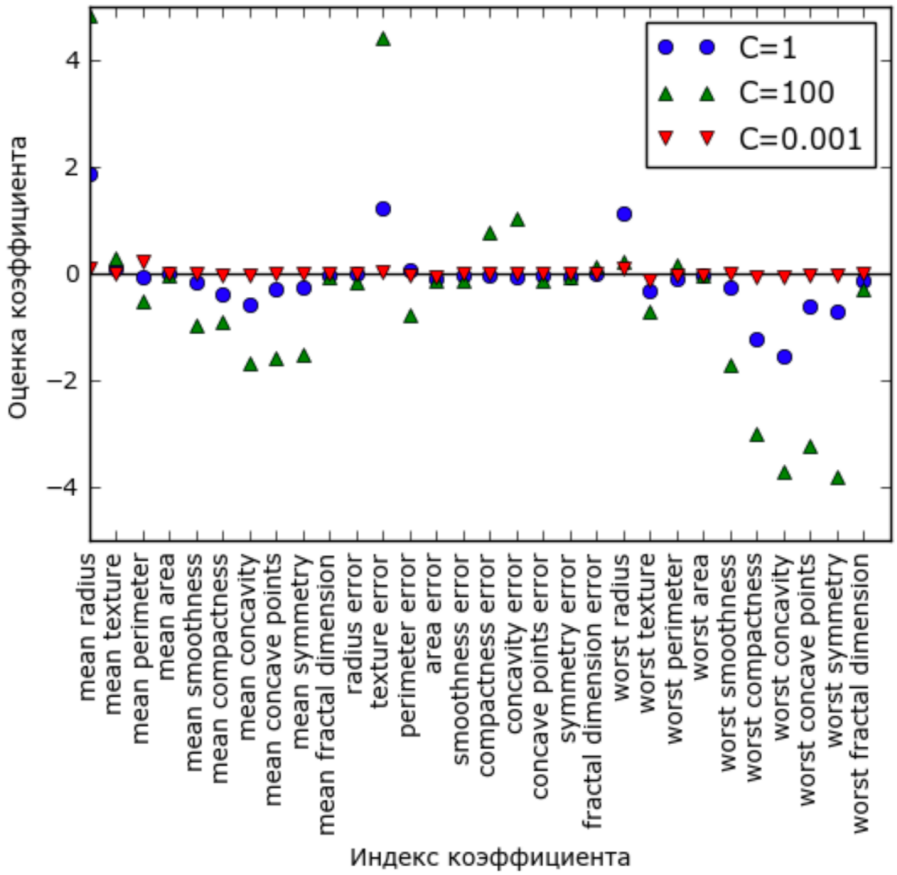


Рис. 85 Коэффициенты, полученные с помощью логистической регрессии с разными значениями регуляризации для набора данных Breast Cancer

Теперь давайте подробнее рассмотрим параметры и гиперпараметры класса `LogisticRegression`. Они приводятся в таблице ниже.

Параметр/Гиперпараметр	Предназначение
`penalty`	Задает тип регуляризации. Значение `l1` соответствует L1-регуляризации (лассо), значение `l2` соответствует L2-регуляризации (гребневой регрессии). Оптимизаторы `newton-cg`, `sag` и `lbfgs` поддерживают только `l2`. По умолчанию используется значение `l2`.

C	Задает силу регуляризации (чем больше значение, тем меньше сила регуляризации). По умолчанию используется значение 1.
tol	Задает допуск сходимости. По умолчанию используется значение 1e-4.
solver	Задает оптимизатор. Возможные значения: newton-cg, lbfgs, liblinear, sag, saga. По умолчанию используется значение liblinear. При работе с небольшими наборами данными хорошим выбором может быть значение liblinear, тогда как при работе с большими наборами данных более быстрыми будут sag и saga. Оптимизаторы newton-cg, lbfgs и sag работают только с l2-регуляризацией, тогда как liblinear и saga работают только с l1-регуляризацией.
max_iter	Задает максимальное количество итераций для оптимизации. По умолчанию используется значение 100.
class_weight	Задает веса классов, например, {0:0.67, 1:0.33}. По умолчанию все классы имеют вес 1.

Мы можем строить модели с разными значениями гиперпараметра C на обучающей выборке, а проверять их качество на контрольной выборке. Проблема такого подхода заключается в том, что мы используем контрольную выборку и для настройки параметров (в данном случае – значений гиперпараметра C) и для оценки качества модели. Мы просто будем настраивать нашу модель под контрольную выборку, ведь любой выбор, сделанный, исходя из метрики на контрольном наборе, «сливает» модели информацию контрольного набора. В итоге мы можем получить оптимистичные результаты. Поэтому важно иметь отдельный тестовый набор, который не использовался для обучения, настройки гиперпараметров и применяется однократно лишь для итоговой оценки. Один из способов решения этой проблемы заключается в том, чтобы разбить данные на три набора: обучающий набор для построения модели, контрольный (валидационный) набор для выбора гиперпараметров модели, а также тестовый набор для оценки качества работы выбранных гиперпараметров. Кроме того, можно воспользоваться комбинированной проверкой, которая сочетает случайное разбиение на обучающий и тестовый наборы и перекрестную проверку.

В рамках этого метода набор данных сначала разбивается на обучающую и тестовую выборки. В нашем случае уже есть обучающая и тестовая выборки. На обучающей выборке запускается перекрестная проверка. Например, при 5-блочной перекрестной проверке исходная обучающая выборка будет разбита на 5 блоков приблизительно равного объема, а затем 5 раз на четырех блоках будет выполнено обучение модели, а пятый, контрольный блок будет использован для проверки. Например, на первом проходе модель будет обучаться на блоках 1-4, а проверяться на блоке 5. На втором проходе модель будет обучаться на блоках 1-3 и 5, а проверяться на блоке 4. Затем вычисляется среднее значение

метрики качества, полученное на 5 контрольных блоках перекрестной проверки.

Допустим, метрикой качества является AUC, у нас 10 значений гиперпараметра `C` и мы задали 5-блочную перекрестную проверку. Для каждого значения гиперпараметра `C` будет построено 5 моделей и среднее значение AUC будет рассчитано на основе 5 контрольных блоков. В итоге выбирается такое значение гиперпараметра, которое дает наибольшее среднее значение AUC. Модель с этим значением гиперпараметра строится на исходной обучающей выборке и проверяется на тестовой выборке. Все вышесказанное можно записать в виде схемы:

1. Разбить набор данных на обучающую и тестовую выборки
2. Задать набор значений гиперпараметров
3. **for** каждого значения гиперпараметра (комбинации значений гиперпараметров) **do**
4. ¦ **for** для каждой итерации перекрестной проверки **do**
5. ¦ ¦ Сформировать обучающие блоки и контрольный блок перекрестной проверки
6. ¦ ¦ Обучить модель на обучающих блоках перекрестной проверки
7. ¦ ¦ Вычислить метрику качества для контрольного блока перекрестной проверки
8. ¦ **end**
9. ¦ Вычислить метрику качества модели, усредненную по контрольным блокам
10. **end**
11. Определить оптимальное значение гиперпараметра (оптимальную комбинацию значений гиперпараметров), дающее наилучшее качество
12. Обучить модель на обучающей выборке с использованием оптимального значения гиперпараметра (оптимальной комбинации значений гиперпараметров)
13. Проверить качество обученной модели на тестовой выборке

Также приводится рисунок, наглядно иллюстрирующий комбинированную проверку (рис. 86).

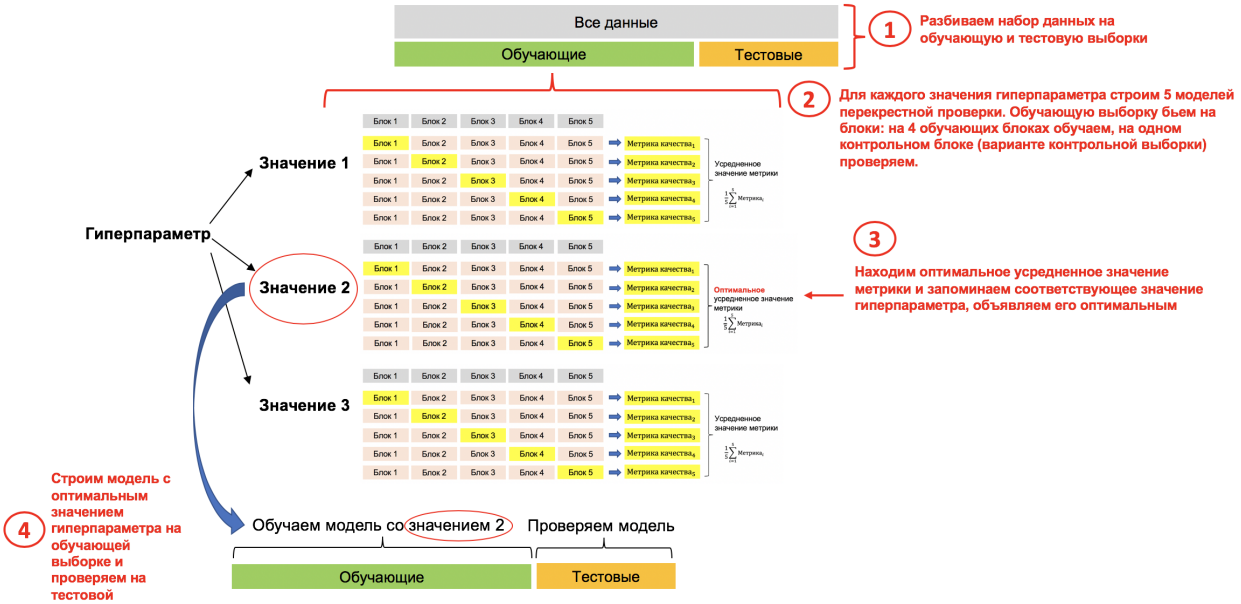


Рис. 86 Комбинированная проверка

В библиотеке `scikit-learn` выполнить такой вид проверки можно с помощью класса `GridSearchCV`.
Давайте подробнее рассмотрим параметры и атрибуты класса `GridSearchCV`. Они приводятся в таблицах ниже (рис. 87).

Параметр	Предназначение
`estimator`	Задает объект-модель машинного обучения, т. е. экземпляр класса, в котором реализован соответствующий метод машинного обучения.

`param_grid`	Задает словарь или список словарей.
	Словарь, в котором ключами будут названия гиперпараметров (строки) и значениями будут списки значений гиперпараметров.
	`param_grid = {'max_depth': [4, 6, 8, 10],`
	`'max_features': [3, 6, 9, 12]}`
	Список словарей, когда каждый словарь в списке выделяется в самостоятельную сетку параметров.
	`param_grid = [{'kernel': ['rbf'],`
	`'C': [0.001, 0.01, 0.1, 1, 10, 100],`
	`'gamma': [0.001, 0.01, 0.1, 1, 10, 100]},`
	`{'kernel': ['linear'],`
	`'C': [0.001, 0.01, 0.1, 1, 10, 100]}]`

`scoring`	Задает оптимизируемую метрику.
`n_jobs`	Задает количество используемых ядер процессора.
`cv`	Задает стратегию перекрестной проверки.
`refit`	Заново обучает модель с наилучшими значениями гиперпараметров на всем наборе (по умолчанию задан).

Атрибут	Предназначение
`cv_results_`	Возвращает результаты перекрестной проверки, которые представляют собой словарь массивов NumPy – словарь, в котором ключами являются имена столбцов, а значениями – значения столбцов, его можно перевести в объект DataFrame библиотеки pandas.
`best_estimator_`	Возвращает модель, которая дала наилучшее значение метрики на той части данных, которая не участвовала в обучении.
`best_score_`	Возвращает наилучшее значение метрики по итогам перекрестной проверки, полученное с помощью наилучшей модели.
`best_params_`	Возвращает значения гиперпараметров, давших наилучшие результаты на той части данных, которая не участвовала в обучении.

Рис. 87 Параметры и атрибуты класса GridSearchCV

Воспользуемся классом `GridSearchCV` для поиска оптимального значения параметра `C`, при этом выберем L1-регуляризацию, при которой некоторые коэффициенты обращаются в ноль и происходит отбор предикторов. Для экономии вычислений мы выбрали 3-блочную перекрестную проверку (большее количество блоков может дать более надежную оценку качества) и три значения параметра `C`.

In[163]:
```
# импортируем класс GridSearchCV
from sklearn.model_selection import GridSearchCV
# создаем экземпляр класса LogisticRegression,
# логистическую регрессию с L1-регуляризацией
logreg_grid = LogisticRegression(penalty = 'l1')
# задаем сетку параметров, будем перебирать
# разные значения штрафа
param_grid = {'C': [0.6, 0.7, 0.8]}
# задаем стратегию перекрестной проверки
stratcv = StratifiedKFold(n_splits=3)
# создаем экземпляр класса GridSearchCV
grid_search = GridSearchCV(logreg_grid, param_grid,
                           scoring = 'roc_auc',
                           n_jobs = -1, cv = stratcv)
# запускаем решетчатый поиск
grid_search.fit(X_tr_scaled3, y_tr)
# проверяем модель со значением гиперпараметра C, дающим наибольшее
# значение AUC (усредненное по контрольным блокам перекрестной
```

```
# проверки), на тестовой выборке
test_score = roc_auc_score(y_tst, grid_search.predict_proba(X_tst_scaled3)[:, 1])
# смотрим результаты решетчатого поиска
print("AUC на тестовой выборке: {:.3f}".format(test_score))
print("Наилучшее значение гиперпараметра C: {}".format(grid_search.best_params_))
print("Наилучшее значение AUC: {:.3f}".format(grid_search.best_score_))
```

```
Out[163]:
AUC на тестовой выборке: 0.760
Наилучшее значение гиперпараметра C: {'C': 0.8}
Наилучшее значение AUC: 0.755
```

Здесь мы видим, что применение регуляризации не позволило улучшить модель, возможно, следует расширить пространство поиска. Вместе с тем при выборе оптимальной модели лучше ориентироваться на результаты комбинированной проверки, а не на результаты, полученные на контрольной выборке при однократном случайном разбиении на обучение и контроль.

Построение логистической регрессии в библиотеке H2O

Библиотека `h2o` позволяет использовать в среде Python возможности платформы H2O от одноименной компании. H2O – платформа с открытым исходным кодом, которая предназначена для быстрого, распределенного и масштабируемого машинного обучения. В США она является одной из самых популярных платформ, использующихся при работе с большими данными. H2O написана на Java, поэтому для работы пакеты `h2o` необходимо зайти на страницу загрузок Java http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html и установить Java SE Development Kit 8. Обратите внимание, вы должны установить именно 8-ю версию, в настоящий момент H2O не поддерживает Java 9.

После установки необходимо установить саму библиотеку. Если вы используете Anaconda, вы можете установить ее в Anaconda Promt с помощью команды `pip install h2o`.

Для построения логистической регрессии нам потребуется класс `H2OGeneralizedLinearEstimator`. Класс `H2OGeneralizedLinearEstimator` библиотеки `h2o` позволяет строить обобщенные линейные модели (generalized linear models – GLM).

Метод обобщенных линейных моделей оценивает регрессионные модели с зависимой переменной, принадлежащей к экспоненциальному семейству распределений.

К семейству GLM-моделей относятся:

- гауссовская регрессия (линейная регрессия);
- пуассоновская регрессия;
- биномиальная регрессия (бинарная логистическая регрессия);
- мультиномиальная регрессия;
- гамма-регрессия;

- порядковая регрессия.

Обобщенная линейная модель состоит из трех компонент: случайной компоненты, систематической компоненты и функции связи.

- *Случайная компонента* задает распределение зависимой переменной Y_i (для i-того наблюдения из n независимых наблюдений) из экспоненциального семейства, например, речь может идти о нормальном распределении для линейной регрессии, биномиальном распределении для биномиальной регрессии. Ее еще называют компонентой шума или компонентой ошибки, потому что она определяет, как необходимо добавить случайную ошибку к прогнозу, сделанному в соответствии с заданной функцией связи.
- *Систематическая компонента* задает линейную комбинацию (взвешенную сумму) предикторов:
  $\eta_i = w_0 + w_1 X_{i1} + w_2 X_{i2} + \cdots + w_k X_{ik}$.
- *Функция связи* задает нелинейное преобразование взвешенной суммы предикторов:
  $\mu_i = g^{-1}(\eta_i) = g^{-1}(w_0 + w_1 X_{i1} + w_2 X_{i2} + \cdots + w_k X_{ik})$.

Функция связи показывает, как ожидаемое значение зависимой переменной μ_i должно быть связано с линейной комбинацией предикторов η_i. Функция связи может быть любой дифференцируемой функцией. Это позволяет использовать предположение об аддитивности влияния ковариат на зависимую переменную (т.е. признаки могут влиять на зависимую переменную не только по отдельности, но и совместно) и определенным образом ограничивать диапазон значений зависимой переменной в зависимости от выбранного преобразования g. В линейной регрессии функция связи будет тождественной функцией, в биномиальной регрессии – логистической функцией. Поскольку функция связи является обратимой, мы записываем ее именно в таком виде:

$\mu_i = g^{-1}(\eta_i) = g^{-1}(w_0 + w_1 X_{i1} + w_2 X_{i2} + \cdots + w_k X_{ik})$.

Функция связи	$\eta_i = g(\mu_i)$	$\mu_i = g^{-1}(\eta_i)$
Тождество	μ_i	η_i
Логарифм	$\ln \mu_i$	e^{η_i}
Логит	$\ln \frac{\mu_i}{1-\mu_i}$	$\frac{1}{1+e^{-\eta_i}}$

Рис. 88 Функции связи для различных обобщенных линейных моделей

Класс `H2OGeneralizedLinearEstimator` имеет общий вид

```
H2OGeneralizedLinearEstimator(model_id=None, family=None,
                              link=None, solver=None,
                              max_active_predictors=None,
                              compute_p_values=False,
                              ignore_const_cols=None,
                              remove_collinear_columns=False,
                              intercept=None, non_negative=False,
                              prior=None, interactions=None,
                              beta_constraints=None,
                              standardize=None,
                              missing_values_handling=None,
                              seed=None,
                              max_iterations=None
                              beta_epsilon=None,
                              gradient_epsilon=None,
                              objective_epsilon=None,
                              nfolds=None,
                              keep_cross_validation_predictions=None,
                              keep_cross_validation_fold_assignment=None,
                              fold_assignment=None,
                              alpha=None, lambda_=None,
                              lambda_search=None,
                              nlambdas=None, lambda_min_ratio=None,
                              nlambdas=100,
                              balance_classes=False,
                              class_sampling_factors=False,
                              max_after_balance_size=False)
```

где

Общие параметры и гиперпараметры обучения	
`model_id`	Задает идентификатор модели. Если не задан, идентификатор генерируется автоматически.
`family`	Задает случайную компоненту обобщенной линейной модели – вид распределения зависимой переменной. Возможные значения для регрессии: `'gaussian'`, `'poisson'`, `'gamma'`, `'tweedie'`. Для биномиальной регрессии нужно задать значение `'binomial'`, для мультиномиальной регрессии – `'multinomial'`, для порядковой регрессии – `'ordinal'`.
`link`	Задает функцию связи. Возможные значения: `'family_default'`, `'identity'`, `'logit'`, `'log'`, `'inverse'`, `'tweedie'`, `'ologit'`, `'oprobit'`, `'ologlog'`.
`solver`	Задает оптимизатор, возможные значения: `'IRLSM'`, `'L_BFGS'`, `'COORDINATE_DESCENT_NAIVE'`, `'COORDINATE_DESCENT'`, `'GRADIENT_DESCENT_LH'`, `'GRADIENT_DESCENT_SQERR'` и `'AUTO'`. `'IRLSM'` – метод наименьших квадратов с итеративным пересчётом весов (iterative reweighted least squares method). Оптимальный вариант для относительно небольших объемов данных (до 500 переменных, хотя объем в основном ограничен количеством доступной памяти). Для большего количества переменных можно применять IRLSM при использовании регуляризации с большими значениями `alpha`, чтобы получить большое количество нулевых коэффициентов.

	`'L_BFGS'` – L означает «ограниченная память» (limited memory), а BFGS – алгоритм Бройдена–Флетчера–Гольдфарба–Шанно (Broyden–Fletcher–Goldfarb–Shanno algorithm). Википедия (https://en.wikipedia.org/wiki/Limitedmemory_BFGS) описывает его как квазиньютоновский метод, начните с изучения этой статьи, если вас интересуют подробности. Этот метод используют вместо IRLSM при работе с большим количеством переменных (реализация в H2O поддерживает работу с сотнями тысяч переменных). `'COORDINATE_DESCENT_NAIVE'`, `'COORDINATE_DESCENT'` – экспериментальные варианты IRLSM (см. https://en.wikipedia.org/wiki/Coordinate_descent). Хорошо подходят для данных, содержащих до 5000 переменных, и представляющих собой разреженные признаки. Они могут повысить качество, когда данные содержат категориальные переменные с большим количеством уровней. Варианты `'GRADIENT_DESCENT_LH'` и `'GRADIENT_DESCENT_SQERR'` задают только для порядковой регрессии. `'AUTO'` – алгоритм выбирается на основе используемых данных и других параметров: • в случае, когда у вас больше 5000 активных предикторов, используется L_BFGS; • если `family='multinomial'` и `alpha=0` (гребневая регрессия или отсутствие штрафа), используется L_BFGS; • если `lambda_search=True` (т.е. задан поиск оптимального значения штрафного коэффициента), то используется COORDINATE_DESCENT; • если ни одно из вышеперечисленных условий не выполняется, используется IRLSM (это обусловлено тем, что COORDINATE_DESCENT дает хорошее качество при `lambda_search=TRUE`). Обратите внимание на некоторые советы: • L_BFGS дает лучшее качество, когда вы строите мультиномиальную регрессию с L2-регуляризацией и у вас большое количество предикторов; • если вам нужно вычислить *p*-значение, используйте IRLSM; • используйте COORDINATE_DESCENT, если у вас меньше 5000 предикторов и используется L1-регуляризация; • COORDINATE_DESCENT дает лучшее качество при `lambda_search=True`.
`max_active_predictors`	Ограничивает количество активных предикторов (обратите внимание, что фактическое количество ненулевых предикторов будет несколько ниже). Значение по умолчанию равно −1, что означает отсутствие ограничений, но можно задать и другое значение для ранней остановки поиска оптимального значения штрафного коэффициента (lambda_search) при достижении заданного количества ненулевых коэффициентов. Модели, построенные в начале поиска, имеют более высокие значения `lambda_`, рассматривают меньшее количество предикторов и

	требуют небольшого времени вычислений. Модели, построенные в конце поиска, имеют более низкие значения `lambda_`, рассматривают большее количество предикторов и требуют большого времени вычислений. Параметр полезен для получения разреженного решения, чтобы избежать затратных вычислений для моделей со слишком большим количеством предикторов. Если для параметра `solver` заданы значения `'IRLSM'`, `'COORDINATE_DESCENT_NAIVE'` и `'COORDINATE_DESCENT'`, то значение параметра `max_active_predictors` по умолчанию равно 5000. Если для параметра `solver` задано значение `'AUTO'` и у вас изначально меньше 5000 активных предикторов, то оптимизатором будет `'IRLSM'` или `'COORDINATE_DESCENT'` (при `lambda_search=True`), а значением по умолчанию для параметра `max_active_predictors` будет 5000. Если у вас `lambda_search=True`, `alpha` > 0 и `solver='AUTO'`, то оптимизатором будет `'COORDINATE_DESCENT'`, а значение параметра `max_active_predictors` по умолчанию будет равно 5000. Для всех остальных сценариев значение по умолчанию для параметра `max_active_predictors` равно 100000000.
`compute_ p_values`	Задает вычисление *p*-значений. Этот параметр можно задавать только при отключенной регуляризации (`lambda_=0`) и когда используется метод наименьших квадратов с итеративным пересчётом весов (`solver='IRLSM'`). Рекомендуется задать параметр `remove_collinear_columns`. H2O вернет ошибку, если запрошено вычисление *p*-значений, при этом есть коррелирующие столбцы, но параметр `remove_collinear_columns` не задан.
`ignore_const_ cols`	Игнорирует столбцы с одинаковыми значениями. По умолчанию используется значение `True`.
`remove_ collinear_ columns`	Удаляет коррелирующие столбцы в ходе построения модели (не используется по умолчанию). Параметр следует задавать только в тех случаях, когда регуляризация не используется (`lambda_=0`), в противном случае это только приведет к ухудшению качества.
`intercept`	Задает включение константы в модель. Константа корректирует все прогнозы в ту или иную сторону на постоянную величину, т.е. она представляет собой спрогнозированное значение, когда все предикторы точно равны нулю. Чтобы исключить константу (приравнять ее к нулю), необходимо задать `intercept=False`.
`non_negative`	Возвращает только положительные регрессионные коэффициенты. Иногда при работе с данными регрессионные модели могут выдавать противоречивые результаты, например, увеличение значения предиктора ведет к увеличению значения зависимой переменной, хотя между ними отрицательная корреляция. Задав `non_negative=True`, вы ограничиваете диапазон значений коэффициентов только неотрицательными числами.
`prior`	Задает априорную вероятность положительного класса зависимой переменной, когда задано `family='binomial'`. Значение должно быть выбрано из диапазона от 0 до 1 и по умолчанию равно -1 (не используется). Изменение параметра

	просто сдвигает константу. Чаще всего используется в трех случаях: 1) Разбили выборку на обучающую и контрольную, построили логистическую регрессию на обучающей и проверили на контрольной. Однако процент «плохих» в обучающей слегка отличается от процента «плохих» во всей выборке. Чтобы это влияние устранить, можно применить корректировку априорных вероятностей. 2) Внедрили модель, однако процент «плохих» изменился (из-за макроэкономики, маркетинга, кредитной политики). Можно поправить с помощью корректировки. 3) Скоринговые модели содержат традиционно гораздо больше «хороших», чем «плохих». Допустим, мы планируем применять модель для скоринга потенциально высокорисковых клиентов, модель надо скорректировать на уровень дефолта в том сегменте клиентов, где необходимо делать скоринг, иначе занизим вероятности дефолта.
`interactions`	Задает список взаимодействий предикторов.
`beta_constraints`	Задает фрейм, который содержит по одной строке на каждый предиктор и состоит из следующих столбцов: `'names'`, `'lower_bounds'`, `'upper_bounds'`, `'beta_given'`, `'rho'`. `'lower_bounds'` и `'upper_bounds'` задают границы для значений коэффициентов, а `'beta_given'` и `'rho'` содержат начальные значения коэффициентов и величины L2-штрафов для них.
`standardize`	Стандартизирует переменные. По умолчанию задано значение `True`. Не рекомендуется отключать.
`missing_values_handling`	Задает способ обработки пропущенных значений.
`seed`	Задает стартовое значение генератора случайных чисел.
Параметры работы оптимизатора	
`max_iterations`	Задает максимально допустимое количество итераций в ходе обучения модели. Значение должно лежать в диапазоне от 1 до 1e6 включительно.
`beta_epsilon`	Параметр для IRLSM. Если значение бета меняется меньше, чем на заданную величину, алгоритм останавливается.
`gradient_epsilon`	Параметр для L-BFGS. Если значение целевой функции меняется меньше, чем на заданную величину, алгоритм останавливается.
`objective_epsilon`	Если значение целевой функции меняется меньше, чем на заданную величину, алгоритм останавливается.
Параметры перекрестной проверки	
`nfolds`	Задает количество блоков *k*-блочной перекрестной проверки. По умолчанию значение равно 0, то есть перекрестная проверка не используется. Если используется перекрестная проверка, требуется задать значение 2 и более.
`keep_cross_validation_predictions`	Задает сохранение прогнозов кросс-валидационных моделей. По умолчанию используется значение `False`.
`keep_cross_validation_fold_assignment`	Сохраняет результаты назначения данных в блоки перекрестной проверки. По умолчанию используется значение `False`.

fold_ assignment	Задает схему назначения данных в блоки перекрестной проверки. По умолчанию используется значение 'AUTO'. Можно задать значения 'AUTO', 'Random', 'Modulo', 'Stratified'. Схема 'Stratified' контролирует распределение классов зависимой переменной в блоках перекрестной проверки (доступна только для задачи классификации).
Гиперпараметры регуляризации	
alpha	Задает тип регуляризации. Значение 1 соответствует l1-регуляризации (лассо), значение 0 соответствует l2-регуляризации (гребневой регрессии), промежуточное значение соответствует комбинации штрафов l1 и l2 (эластичной сети). Если задано solver='L_BFGS', значение alpha по умолчанию равно 0, в противном случае значение равно 0,5 (т.е. используется эластичная сеть).
lambda_	Задает величину штрафного коэффициента (силу регуляризации). Чем больше значение, тем сильнее коэффициенты сжимаются к нулю. Значение 0 соответствует отсутствию регуляризации, и гиперпараметр alpha игнорируется.
lambda_search	Задает поиск оптимального значения штрафного коэффициента.
lambda_min_ ratio	Задает минимальное значение lambda_, заданное как процент от начального (максимального) значения. Если количество строк больше количества столбцов, то значение по умолчанию будет равно 0,0001. Например, если начальное значение lambda_ составляло 15, то параметр lambda_min_ratio будет равен 0,0015. Если количество строк меньше количества столбцов, то значение по умолчанию будет равно 0,01. Значение 0 никогда не проверяется в ходе автоматического поиска, так что при необходимости такое значение следует задавать вручную.
nlambdas	Задает количество проверяемых значений гиперпараметра lambda_. Когда alpha >0, значение lambda_min_ratio по умолчанию равно 1E-4 и значением nlambdas по умолчанию будет 100. Когда alpha=0, то значением nlambdas по умолчанию будет 30, поскольку для гребневой регрессии требуется меньшее количество значений lambda_. Можно задать меньшее значение для экономии времени.
Гиперпараметры для работы с дисбалансом классов	
balance_ classes	Позволяет менять баланс классов. По умолчанию задано значение False.
class_ sampling_ factors	Задает желаемые пропорции классов (перечисляются в лексикографическом порядке). Для работы параметра требуется значение гиперпараметра balance_classes=True.
max_after_ balance_size	Задает максимальный относительный размер обучающих данных после балансировки пропорций классов (не может быть меньше 1). Для работы параметра требуется значение гиперпараметра balance_classes=True. По умолчанию используется значение 5.

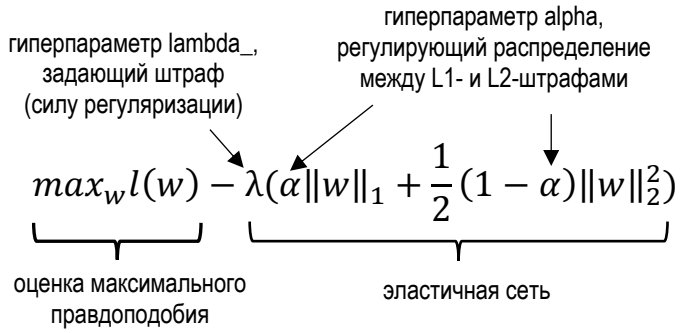
В зависимости от решаемой задачи в первую очередь для обучения обобщенной модели нужно задать случайную компоненту,

определяющую распределение зависимой переменной (параметр `family`), и функцию связи (параметр `link`). Ниже приводится таблица соответствия значений для параметров `family` и `link` (рис. 89).

параметр family	**параметр link**								
	`family_default`	`identity`	`logit`	`log`	`inverse`	`tweedie`	`ologit`	`oprobit`	`ologlog`
`binomial`	x		x						
`quasibinomial`	x		x						
`multinomial`	x								
`ordinal`	x						x	x	x
`gaussian`	x	x		x	x				
`poisson`	x	x		x					
`gamma`	x	x		x	x				
`tweedie`	x					x			

Рис. 89 Таблица соответствия значений для параметров family и link

В библиотеке `h2o` обобщенные линейные модели строятся путем максимизации правдоподобия с ограничением на оценки регрессионных коэффициентов. Ограничение достигается за счет применения регуляризации методом эластичной сети, которая комбинирует штрафы гребневой регрессии и лассо-регрессии. Применительно к биномиальной модели это будет выглядеть так:



В этой формуле мы видим два важнейших гиперпараметра, отвечающих за регуляризацию – `lambda_` и `alpha`. Чтобы получить наилучшую модель, нам нужно найти оптимальные значения этих гиперпараметров. Гиперпараметр `lambda_` задает штрафной коэффициент. Чем больше значение `lambda_`, тем сильнее коэффициенты будут сжиматься к нулю. Если значение равно 0, то регуляризация не используется и строится обычная обобщенная линейная модель.

Гиперпараметр `alpha` задает распределение между L1- и L2-штрафами. Если значение `alpha` равно 0, то применяется L2-регуляризация (гребневая регрессия), если значение `alpha` равно 1, то применяется L1-регуляризация (лассо), промежуточные значения между 0 и 1 задают эластичную сеть. Ниже приводится таблица, в которой приводится тип регуляризации в зависимости от значений `lambda_` и `alpha` (рис. 90).

значение lamba	значение alpha	результат
lambda_ = 0	alpha = любое значение	Нет регуляризации, гиперпараметр alpha игнорируется
lambda_ > 0	alpha = 0	Гребневая регрессия
lambda_ > 0	alpha = 1	Лассо
lambda_ > 0	0 < alpha < 1	Эластичная сеть

Рис. 90 Тип регуляризации в зависимости от значений lambda_ и alpha

Кроме того, с помощью специального параметра `lambda_search` можно задать поиск оптимального значения штрафного коэффициента `lambda_`. Если задан данный параметр, процедура будет вычислять модели по полному пути регуляризации. Путь регуляризации начинается с максимального значения `lambda_` и двигается к минимальному значению, уменьшая силу регуляризации на каждом шаге. Минимальное значение `lambda_` определяется сочетанием значений гиперпараметров `lambda_min_ratio` и `nlambdas`. Значение `lambda_min_ratio` по умолчанию зависит от размера набора данных (соотношения количества строк к количеству столбцов). Если количество строк больше количества столбцов, то значение `lambda_min_ratio` будет равно 0,0001 (в противном случае оно будет равно 0,01). Если максимальное значение `lambda_` равно 2, то минимальное значение `lambda_` будет равно `lambda_min_ratio` × максимальное значение `lambda_`, т.е. $2 \times 0{,}0001 = 0{,}0002$. Гиперпараметр `nlambdas` задает количество значений, проверяемых в ходе поиска оптимального значения штрафного коэффициента.
Возвращенная модель будет иметь коэффициенты, соответствующие оптимальному значению `lambda_`, найденного в ходе обучения. Когда задано значение `alpha > 0`, перебор значений `lambda_` можно использовать для эффективной обработки наборов с большим количеством предикторов, поскольку это позволяет удалять бесполезные (шумовые) предикторы и строить модели лишь для небольшого поднабора предикторов. Возможное применение перебора значений `lambda_` может быть следующим: запускаете его на наборе с большим количеством предикторов, но при этом ограничиваете количество активных предикторов относительно небольшим значением.
Главным показателем качества подгонки модели будет девианс, вычисляемый по формуле:

$$D = -2\sum_{i=1}^{n}(y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)$$

Итак, давайте построим модель логистической регрессии в H2O. Для этого импортируем библиотеку `h2o`, модуль `os`, с помощью функции `h2o.init()` запустим кластер H2O. Параметр `nthreads` задает количество ядер процессора, используемое при вычислениях (по умолчанию

используются все ядра процессора), параметр `max_mem_size` задает максимальный объем памяти для вычислений.

In[164]:
```
# перед импортом библиотеки h2o и модуля os убедитесь, что библиотека h2o установлена
# (сначала установите Java SE Development Kit 8, обратите внимание,
# 9-я версия H2O не поддерживается, а затем после установки Java
# запустите Anaconda Prompt и установите h2o с помощью
# строки pip install h2o)
import h2o
import os
h2o.init(nthreads=-1, max_mem_size=8)
```

Out[164]:
```
Checking whether there is an H2O instance running at http://localhost:54321. connected.
```

H2O cluster uptime:	21 mins 11 secs
H2O cluster timezone:	Europe/Moscow
H2O data parsing timezone:	UTC
H2O cluster version:	3.18.0.11
H2O cluster version age:	2 months and 15 days
H2O cluster name:	H2O_from_python_Gewissta_kt7oqr
H2O cluster total nodes:	1
H2O cluster free memory:	6.892 Gb
H2O cluster total cores:	4
H2O cluster allowed cores:	4
H2O cluster status:	locked, healthy
H2O connection url:	http://localhost:54321
H2O connection proxy:	None
H2O internal security:	False
H2O API Extensions:	Algos, AutoML, Core V3, Core V4
Python version:	3.6.5 final

Теперь наши датафреймы нужно преобразовать во фреймы H2O – специальную структуру данных, которая используется платформой H2O при вычислениях. Это можно сделать с помощью функции `h2o.H2OFrame()`. Однако давайте не будем торопиться с преобразованием во фреймы. Дело в том, что H2O не умеет работать с кириллицей и, если она используется в названиях категорий (в нашем случае речь идет о категориях переменной *living_region*), лучше выполнить транслитерацию латиницей. В этом нам поможет библиотека `cyrtranslit`.

In[165]:
```
# импортируем библиотеку для транслитерации
import cyrtranslit
```

In[166]:
```
# выполняем транслитерацию
train['living_region'] = train['living_region'].apply(lambda x: \
                                                      cyrtranslit.to_latin(x, 'ru'))
test['living_region'] = test['living_region'].apply(lambda x: \
                                                    cyrtranslit.to_latin(x, 'ru'))
```

In[167]:
```
# смотрим результаты транслитерации
print(train['living_region'].unique())
print(test['living_region'].unique())
```

Out[167]:
```
['AMURSKAJA' "STAVROPOL'SKIJ" 'ROSTOVSKAJA' 'HAKASIJA' 'CHELJABINSKAJA'
 'SAHA/JAKUTIJA/' 'TATARSTAN' 'KARELIJA' 'KEMEROVSKAJA' 'NIZHEGORODSKAJA'
 'VOLGOGRADSKAJA' 'PRIMORSKIJ' 'KABARDINO-BALKARSKAJA' 'KRASNOJARSKIJ'
 'BASHKORTOSTAN' 'CHUVASHSKAJA' 'TVERSKAJA' 'RJAZANSKAJA' 'MOSKVA'
 'MOSKOVSKAJA' 'IRKUTSKAJA' 'SAMARSKAJA' 'OMSKAJA' 'VORONEZHSKAJA'
 'JAMALO-NENECKIJ' 'PERMSKIJ' 'SVERDLOVSKAJA' 'LENINGRADSKAJA'
 'KRASNODARSKIJ' 'PENZENSKAJA' 'HABAROVSKIJ' "TUL'SKAJA" 'SANKT-PETERBURG'
 'SARATOVSKAJA' 'HANTY-MANSIJSKIJ' 'KURGANSKAJA' 'KOMI' 'MURMANSKAJA'
 'KALUZHSKAJA' 'ORLOVSKAJA' 'KAMCHATSKIJ' "ZABAJKAL'SKIJ" 'NOVOSIBIRSKAJA'
 'TJUMENSKAJA' 'ORENBURGSKAJA' 'VOLOGODSKAJA' 'BELGORODSKAJA'
 'MAGADANSKAJA' 'KURSKAJA' 'MORDOVIJA' 'ADYGEJA' 'ASTRAHANSKAJA'
 'ALTAJSKIJ' 'JAROSLAVSKAJA' 'VLADIMIRSKAJA' 'MARIJEHL' 'TOMSKAJA'
 "ARHANGEL'SKAJA" 'IVANOVSKAJA' 'SAHALINSKAJA' 'EVREJSKAJA' 'BURJATIJA'
 'TAMBOVSKAJA' "UL'JANOVSKAJA" 'TYVA' 'KALININGRADSKAJA' 'KALMYKIJA'
 'UDMURTSKAJA' 'KOSTROMSKAJA' 'PSKOVSKAJA' 'LIPECKAJA' 'NOVGORODSKAJA'
 'SEVERNAJAOSETIJA-ALANIJA' 'BRJANSKAJA' 'SMOLENSKAJA'
 'KARACHAEVO-CHERKESSKAJA' 'KIROVSKAJA' 'DAGESTAN' 'nan' 'NENECKIJ'
 'OTHER' 'ALTAJ']
['KRASNODARSKIJ' 'MOSKVA' 'SARATOVSKAJA' 'CHELJABINSKAJA' "STAVROPOL'SKIJ"
 'SANKT-PETERBURG' 'BASHKORTOSTAN' 'HANTY-MANSIJSKIJ' 'ROSTOVSKAJA'
 'MOSKOVSKAJA' 'IRKUTSKAJA' "ZABAJKAL'SKIJ" 'KURGANSKAJA' 'PERMSKIJ'
 'ASTRAHANSKAJA' 'NIZHEGORODSKAJA' 'ORENBURGSKAJA' "UL'JANOVSKAJA"
 'TJUMENSKAJA' 'MURMANSKAJA' 'KRASNOJARSKIJ' 'BURJATIJA' 'AMURSKAJA'
 'TATARSTAN' 'JAROSLAVSKAJA' 'SVERDLOVSKAJA' 'SAMARSKAJA' 'OMSKAJA'
 'OTHER' 'ADYGEJA' 'VOLGOGRADSKAJA' 'PRIMORSKIJ' 'SMOLENSKAJA'
 'NOVOSIBIRSKAJA' 'DAGESTAN' 'TVERSKAJA' 'TOMSKAJA' 'LENINGRADSKAJA'
 'KEMEROVSKAJA' 'MARIJEHL' 'VOLOGODSKAJA' 'HABAROVSKIJ' 'NOVGORODSKAJA'
 'KALUZHSKAJA' "TUL'SKAJA" 'EVREJSKAJA' 'LIPECKAJA' 'TAMBOVSKAJA'
 'UDMURTSKAJA' 'PENZENSKAJA' 'SAHA/JAKUTIJA/' 'KOMI' 'KURSKAJA'
 'IVANOVSKAJA' 'KOSTROMSKAJA' 'ALTAJ' 'VORONEZHSKAJA' 'VLADIMIRSKAJA'
 'HAKASIJA' "ARHANGEL'SKAJA" 'TYVA' 'KAMCHATSKIJ' 'JAMALO-NENECKIJ' 'nan'
 'PSKOVSKAJA' 'KARELIJA' 'KALININGRADSKAJA' 'KIROVSKAJA'
 'KABARDINO-BALKARSKAJA' 'SEVERNAJAOSETIJA-ALANIJA' 'RJAZANSKAJA'
 'CHUVASHSKAJA' 'ALTAJSKIJ' 'ORLOVSKAJA' 'KALMYKIJA' 'BRJANSKAJA'
 'BELGORODSKAJA' 'MORDOVIJA' 'KARACHAEVO-CHERKESSKAJA' 'SAHALINSKAJA'
 'MAGADANSKAJA' 'NENECKIJ']
```

Категориальные переменные *open_account_flg*, *match* и *ind* принимают значения 0 и 1 и в ходе преобразования будут прочитаны как целочисленные переменные. Давайте заменим значения 0 и 1 переменных *open_account_flg*, *match* и *ind* заменим на *No* и *Yes* соответственно.

In[168]:
```
# в указанных переменных заменяем
# значения 0 и 1 на No и Yes
for i in ['open_account_flg', 'ind', 'match']:
    train[i] = np.where(train[i] == 0, 'No', 'Yes')
    test[i] = np.where(test[i] == 0, 'No', 'Yes')
```

Вот теперь мы можем преобразовать датафреймы во фреймы H2O.

In[169]:
```
# преобразовываем датафреймы pandas во фреймы h2o -
# специальную структуру данных, используемую h2o
tr = h2o.H2OFrame(train)
valid = h2o.H2OFrame(test)
```

Давайте с помощью метода `.describe()` библиотеки `h2o` взглянем на обучающий фрейм.

In[170]:
```
# взглянем на обучающий фрейм, обратите внимание, сейчас
# метод .describe() - это метод h2o, а не pandas
tr.describe()
```

Out[170]:

```
Rows:119522
Cols:30
```

	gender	age	marital_status	job_position	credit_sum	credit_month
type	enum	real	enum	enum	real	real
mins		-2.3725471873521573			-3.4612087704028807	-2.267902147056745
mean		-1.6141741312845933e-12			6.699963453468024e-14	2.4778217588796186e-16
maxs		2.5131345688897175			3.687277417703042	7.114561232323337
sigma		0.999999999999973			0.9999999999999912	1.0000000000004858
zeros		0			0	0
missing	0	0	0	0	0	0
0	M	-0.7995604182457782	UNM	SPC	0.7151405451638057	-0.2776826423397579
1	M	0.287656587111897	UNM	SPC	0.12146407959376196	-0.2776826423397579
2	F	-1.203029345079497	MAR	SPC	1.0061696655738412	3.702756367094216
3	F	2.198790976654884	UNM	UMN	-3.0557922269231743	-1.4149509307494648
4	M	1.1927855064995807	MAR	BIS	0.8475786153137629	-0.2776826423397579
5	F	0.3801347242703677	UNM	SPC	-0.8388749617927028	-0.2776826423397579
6	M	-0.5539335340878376	UNM	SPC	-0.6059000915820513	0.29095150186509544
7	F	0.8095962462734801	WID	SPC	0.4061883138870192	-0.2776826423397579
8	F	-0.2146114335435804	MAR	SPC	-0.5075135719940791	-0.2776826423397579
9	F	-0.5539335340878376	UNM	SPC	-2.9527303257337056	-1.4149509307494648

В полученном выводе первой приводится информация о количестве строк (наблюдений) и количестве столбцов (переменных). Затем перечисляются показатели:

- `type` – информация о типах переменных (тип `int` – типу `int`, когда значения представляют собой целые числа; тип `real` соответствует типу `float`, когда значения представляют собой числа с плавающей точкой; тип `enum` соответствует типу `object`, когда значения представляют собой категории; тип `string` – это текст, соответствуют типу `str`, его обычно конвертируют в тип `enum`, потому что текстовые данные не могут напрямую использоваться при построении моделей);
- `mins` – минимальные значения переменных;
- `mean` – средние значения переменных;
- `maxs` – максимальные значения переменных;
- `sigma` – стандартные отклонения переменных;
- `zeros` – нулевые значения переменных (для оценки степени разреженности данных);

- `missing` – пропущенные значения переменных;
- `0…9` – первые 10 наблюдений для переменных.

В нашем случае все переменные имеют правильный тип. В тех случаях, когда вам нужно преобразовать целочисленную или строковую переменную в категориальную, можно воспользоваться методом `.asfactor()`. Если нужно преобразовать категориальную переменную в целочисленную, воспользуйтесь методом `.asnumeric()`.
Далее мы задаем `dependent` – название зависимой переменной и `predictors` – список названий предикторов. Чуть позже с помощью `dependent` и `predictors` мы укажем алгоритму логистической регрессии, какая переменная в обучающем и контрольном фреймах будет зависимой переменной, а какие переменные будут предикторами.

In[171]:
```
# задаем название зависимой переменной
dependent = 'open_account_flg'
# задаем список названий предикторов
predictors = list(tr.columns)
# удаляем название зависимой переменной из
# списка названий предикторов
predictors.remove(dependent)
```

Теперь импортируем класс `H2OGeneralizedLinearEstimator`, создаем экземпляр класса и строим модель логистической регрессии.

In[172]:
```
# импортируем класс H2OGeneralizedLinearEstimator
from h2o.estimators.glm import H2OGeneralizedLinearEstimator
```

In[173]:
```
# создаем экземпляр класса H2OGeneralizedLinearEstimator,
# задаем lambda_search - перебор значений штрафного
# коэффициента lambda_
glm_model = H2OGeneralizedLinearEstimator(model_id='logreg_credit', family='binomial',
                                          lambda_search=True)
# обучаем модель
glm_model.train(predictors, dependent,
                training_frame=tr, validation_frame=valid)
```
In[174]:
```
# смотрим модель
glm_model
```

Первыми приводятся метрики качества модели, вычисленные на обучающей выборке (рис. 91).

```
Model Details
=============
H2OGeneralizedLinearEstimator :  Generalized Linear Modeling
Model Key:  logreg_credit


ModelMetricsBinomialGLM: glm
** Reported on train data. **

MSE: 0.1261672411426619
RMSE: 0.35520028313989543
LogLoss: 0.40230488820323834
Null degrees of freedom: 119521
Residual degrees of freedom: 119351
Null deviance: 111168.75088405701
Residual deviance: 96168.56969565489
AIC: 96510.56969565489
AUC: 0.7575812955289656
Gini: 0.5151625910579312
Confusion Matrix (Act/Pred) for max f1 @ threshold = 0.21290383555924716:
```

	No	Yes	Error	Rate
No	74663.0	23840.0	0.242	(23840.0/98503.0)
Yes	8091.0	12928.0	0.3849	(8091.0/21019.0)
Total	82754.0	36768.0	0.2672	(31931.0/119522.0)

```
Maximum Metrics: Maximum metrics at their respective thresholds
```

metric	threshold	value	idx
max f1	0.2129038	0.4474363	229.0
max f2	0.1101217	0.5928520	307.0
max f0point5	0.3080352	0.4099096	169.0
max accuracy	0.5578902	0.8293954	62.0
max precision	0.7984423	0.7207792	13.0
max recall	0.0110493	1.0	398.0
max specificity	0.9582089	0.9999898	0.0
max absolute_mcc	0.2190982	0.3078501	225.0
max min_per_class_accuracy	0.1787889	0.6931348	253.0
max mean_per_class_accuracy	0.1603741	0.6956576	267.0

Рис. 91 Информация о модели: метрики качества, вычисленные на обучающей выборке

MSE – среднеквадратичная ошибка, вычисляется по формуле:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2,$$

где:

y_i – фактическое значение зависимой переменной в i-ном наблюдении;

$\hat{y}_i$ – спрогнозированное значение зависимой переменной в i-ном наблюдении;

N – общее количество наблюдений.

Чем меньше значение метрики, тем лучше качество модели.

RMSE – корень из среднеквадратичной ошибки, вычисляется по формуле:

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2},$$

где
y_i – фактическое значение зависимой переменной в i-ном наблюдении;
$\hat{y}_i$ – спрогнозированное значение зависимой переменной в i-ном наблюдении;
N – общее количество наблюдений.
Чем меньше значение метрики, тем лучше качество модели. RMSE часто используется вместо MSE для того, чтобы получить ошибку такой же размерности, что и у зависимой переменной.
LogLoss – логистическая функция потерь, вычисляется по формуле:
$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\log p_{ij},$$
где:
N – общее количество наблюдений;
M – количество возможных классов;
p_{ij} – спрогнозированная вероятность класса j для наблюдения i.
Для бинарной классификации логистическая функция потерь сводится к формуле
$$logloss = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log p_i + (1 - y_i)\log(1 - p_i)].$$
Чем меньше значение метрики, тем лучше качество модели. У идеального классификатора значение метрики logloss будет равно точно 0. Недостаток logloss заключается в том, что он крайне сильно штрафует за уверенность классификатора в неверном ответе. Ошибка на одном объекте может дать существенное ухудшение общей ошибки на выборке.
AIC – информационный критерий Акаике, применяется только для выбора из нескольких моделей. Чем меньше значение метрики, тем лучше качество модели.
AUC – площадь под ROC-кривой, позволяет оценить дискриминирующую способность модели. Чем выше значение метрики, тем лучше качество модели.
Gini – индекс Джини, еще один показатель дискриминирующей способности модели, вычисляется как 2AUC – 1. Чем выше значение метрики, тем лучше качество модели.
Confusion Matrix – матрица ошибок, которая приводится для порогового значения спрогнозированной вероятности события, оптимального с точки зрения F1-меры. Строки матрицы – фактические классы зависимой переменной, столбцы матрицы – спрогнозированные классы зависимой переменной. При этом для классификации H2O использует

пороговое значение вероятности положительного класса, при котором достигается максимальное значение F1-меры. В данном случае этот порог равен 0,201. Поэтому если вероятность положительного класса меньше порогового значения 0,201, прогнозируется класс 0, а если она больше порогового значения 0,201, прогнозируется класс 1.
`Maximum Metrics` – максимальные значения метрик и соответствующие пороговые значения спрогнозированной вероятности события (вероятности положительного класса).
Рассмотрим наиболее важные метрики.
`max f1` – максимальное значение F1-меры. F1-мера вычисляется по формуле:

$$F1 = 2 \times \frac{\text{точность} \times \text{полнота}}{\text{точность} + \text{полнота}}$$

F1-мера присваивает одинаковый вес точности и полноте.

`max f2` – максимальное значение F2-меры. F2-мера вычисляется по формуле:

$$F2 = 5 \times \frac{\text{точность} \times \text{полнота}}{4 \times \text{точность} + \text{полнота}}$$

F2-мера используется, когда полноте нужно присвоить вес, в 2 раза больший веса точности.

`max f0.5` – максимальное значение F0.5-меры. F0.5-мера вычисляется по формуле:

$$F0.5 = 1{,}25 \times \frac{\text{точность} \times \text{полнота}}{0{,}25 \times \text{точность} + \text{полнота}}$$

F0.5-мера используется, когда точности нужно присвоить вес, в 2 раза больший веса полноты.
`max accuracy` – максимальное значение правильности. Правильность вычисляется по формуле:

$$\text{Правильность} = \frac{TP + TN}{TP + TN + FP + FN}$$

где
TP – истинно положительные примеры;
TN – истинно отрицательные примеры;
FP – ложно положительные примеры;
FN – ложно отрицательные примеры.

`max precision` – максимальное значение точности. Точность показывает, сколько из предсказанных положительных примеров оказались действительно положительными. Таким образом, точность – это количество истинно положительных примеров, поделенное на общее количество предсказанных положительных примеров:

$$\text{Точность} = \frac{TP}{TP + FP}$$

`max recall` – максимальное значение полноты (чувствительности). Полнота показывает, сколько от общего числа фактических положительных примеров было предсказано как положительный класс. Таким образом, полнота – это количество истинно положительных примеров, поделенное на общее количество фактических положительных примеров:

$$\text{Полнота} = \frac{TP}{TP + FN}$$

`max specificity` – максимальное значение специфичности. Специфичность – это количество истинно отрицательных примеров, поделенное на общее количество фактических отрицательных примеров:

$$\text{Специфичность} = \frac{TN}{TN + FP}$$

`max absolute_mcc` – максимальное абсолютное значение коэффициента корреляции Мэтьюса (Matthews correlation coefficient). Коэффициент корреляции Мэтьюса, предложенный в 1975 году Брайаном Мэтьюсом, используется в машинном обучении в качестве критерия качества бинарной классификации. Он вычисляется по формуле:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

Коэффициент корреляции Мэтьюса может рассматриваться в качестве сбалансированной метрики даже в тех случаях, когда классы могут иметь сильно различающиеся размеры. По сути этот коэффициент является коэффициентом корреляции между фактической и спрогнозированной бинарными классификациями. Он принимает значения от -1 до +1. Значение +1 соответствует идеальному прогнозу, значение 0 означает, что прогноз не лучше случайного угадывания, а значение -1 указывает на полную рассогласованность между фактическими и прогнозными значениями.

Вслед за метриками качества, вычисленными на обучающей выборке, приводится таблица выигрышей, вычисленная на обучающей выборке (рис. 92).

```
Gains/Lift Table: Avg response rate: 17,59 %, avg score: 17,59 %
```

	group	cumulative_data_fraction	lower_threshold	lift	cumulative_lift	response_rate	score	cumulative_respo
	1	0.0100065	0.6378976	3.8796699	3.8796699	0.6822742	0.7168867	0.6822742
	2	0.0200047	0.5632932	3.2310053	3.5554733	0.5682008	0.5984885	0.6252614
	3	0.0300028	0.5173209	2.7218484	3.2776758	0.4786611	0.5393525	0.5764083
	4	0.0400010	0.4829285	2.8265348	3.1649141	0.4970711	0.4991265	0.5565781
	5	0.0500075	0.4557817	2.4723387	3.0263295	0.4347826	0.4690611	0.5322068
	6	0.1000067	0.3723818	2.1837751	2.6050875	0.3840361	0.4096924	0.4581277
	7	0.1500059	0.3211259	2.0086924	2.4063003	0.3532463	0.3452040	0.4231692
	8	0.2000050	0.2799216	1.8202883	2.2598034	0.3201138	0.3000152	0.3974064
	9	0.3000033	0.2162468	1.5253122	2.0149798	0.2682396	0.2466348	0.3543520
	10	0.4000017	0.1656031	1.1751469	1.8050260	0.2066600	0.1899420	0.3174298
	11	0.5	0.1281797	0.8573339	1.6154907	0.1507697	0.1459414	0.2840983
	12	0.5999983	0.1012857	0.6798725	1.4595565	0.1195616	0.1138453	0.2566759
	13	0.6999967	0.0808421	0.4771953	1.3192209	0.0839190	0.0906950	0.2319967
	14	0.7999950	0.0637231	0.3644383	1.1998743	0.0640897	0.0721637	0.2110085
	15	0.8999933	0.0465378	0.2592936	1.0953663	0.0455991	0.0552949	0.1926298
	16	1.0	0.0038062	0.1417670	1.0	0.0249310	0.0343014	0.1758588

Рис. 92 Информация о модели: таблица выигрышей, вычисленная на обучающей выборке

Таблица выигрышей, показанная на рис. 92, была создана путем разбиения данных на группы по квантилям спрогнозированной вероятности положительного класса, взятым в качестве пороговых значений. Количество групп по умолчанию равно 20, однако, если количество уникальных пороговых значений спрогнозированной вероятности меньше 20, то количество групп корректируется на количество этих значений.

Для каждой группы вычисляются накопленная доля данных (`cumulative_data_fraction`), пороговое значение спрогнозированной вероятности (`lower_threshold`), прирост (`lift`), накопленный прирост (`cumulative_lift`), процент достигнутого отклика (`response_rate`), накопленный процент достигнутого отклика (`cumulative_response_rate`), процент охвата (`capture_rate`), накопленный процент охвата (`cumulative_capture_rate`), выигрыш (`gain`) и накопленный выигрыш (`cumulative_gain`).

Прирост (lift) по *n*-ной группе – это отношение фактической доли объектов положительного класса в *n*-ной группе к общей фактической доле объектов положительного класса в наборе данных. Объекты положительного класса еще называют событиями. Например, для вычисления прироста в шестой группе мы должны разделить фактическую долю объектов положительного класса в группе (берем ее из столбца `response_rate`) на общую фактическую долю объектов положительного класса в наборе данных (берем ее из строки `Gains/Lift`

`Table: Avg response rate`, предваряющей таблицу выигрышей): 0,384 / 0,176 = 2,18.

Накопленный прирост (cumulative lift) по *n*-ной группе – это отношение фактической доли объектов положительного класса, накопленной по первым *n* группам, к общей фактической доле объектов положительного класса в наборе данных. Попробуем вычислить накопленный прирост для шестой группы. Для вычисления накопленного прироста нужно разделить фактическую долю объектов положительного класса, накопленную по первым 6 группам (берем ее из столбца `cumulative_response_rate`), на общую фактическую долю объектов положительного класса в наборе данных: 0,458 / 0,176 = 2,6.

Процент отклика (response rate) по *n*-ной группе – это доля объектов положительного класса в *n*-ной группе от общего количества объектов в *n*-ной группе.

Процент охвата (capture rate) по *n*-ной группе – это отношение фактического числа объектов положительного класса в *n*-ной группе к общему фактическому числу объектов положительного класса в наборе данных.

Выигрыш (gain) для *n*-ной группы вычисляется по формуле: 100 * (Прирост по *n*-ной группе – 1).

Накопленный выигрыш (cumulative gain) для *n*-ной группы вычисляется по формуле: 100 * (Накопленный прирост по *n*-ной группе – 1).

Положительный выигрыш означает, что фактическая доля событий в группе больше общей фактической доли событий в наборе данных. Отрицательный выигрыш означает, что фактическая доля событий в группе меньше общей фактической доли событий в наборе данных.

После таблицы выигрышей, вычисленной на обучающей выборке, приводятся метрики, вычисленные на контрольной выборке (рис. 93), и таблица выигрышей, вычисленная на контрольной выборке (рис. 94).

```
ModelMetricsBinomialGLM: glm
** Reported on validation data. **

MSE: 0.1266624730424852
RMSE: 0.35589671681891816
LogLoss: 0.4026697617885108
Null degrees of freedom: 51223
Residual degrees of freedom: 51053
Null deviance: 47733.00986177474
Residual deviance: 41252.71175570936
AIC: 41594.71175570936
AUC: 0.7593370322058829
Gini: 0.5186740644117658
Confusion Matrix (Act/Pred) for max f1 @ threshold = 0.2040389818816695:
```

	No	Yes	Error	Rate
No	31286.0	10901.0	0.2584	(10901.0/42187.0)
Yes	3256.0	5781.0	0.3603	(3256.0/9037.0)
Total	34542.0	16682.0	0.2764	(14157.0/51224.0)

```
Maximum Metrics: Maximum metrics at their respective thresholds
```

metric	threshold	value	idx
max f1	0.2040390	0.4495509	237.0
max f2	0.1109271	0.5957430	306.0
max f0point5	0.2900205	0.4088787	182.0
max accuracy	0.5238319	0.8277956	72.0
max precision	0.7369486	0.7314286	20.0
max recall	0.0171879	1.0	395.0
max specificity	0.9490338	0.9999763	0.0
max absolute_mcc	0.1897439	0.3103128	247.0
max min_per_class_accuracy	0.1796965	0.6942423	254.0
max mean_per_class_accuracy	0.1571504	0.6977553	270.0

Рис. 93 Информация о модели: метрики качества, вычисленные на контрольной выборке

Gains/Lift Table: Avg response rate: 17,64 %, avg score: 17,58 %

	group	cumulative_data_fraction	lower_threshold	lift	cumulative_lift	response_rate	score	cumulative_respo
	1	0.0100148	0.6339341	3.6904414	3.6904414	0.6510721	0.7165538	0.6510721
	2	0.0200102	0.5612566	3.0112593	3.3511816	0.53125	0.5941409	0.5912195
	3	0.0300055	0.5127573	2.9337636	3.2121328	0.5175781	0.5348583	0.5666884
	4	0.0400008	0.4794973	2.5794978	3.0540513	0.4550781	0.4949154	0.5387994
	5	0.0500156	0.4532387	2.4971250	2.9425356	0.4405458	0.4658077	0.5191257
	6	0.1000117	0.3708810	2.1756706	2.5591779	0.3838344	0.4074308	0.4514933
	7	0.1500078	0.3211706	2.0141000	2.3775089	0.3553299	0.3451958	0.4194430
	8	0.2000039	0.2812637	1.8658091	2.2495965	0.3291683	0.3006420	0.3968765
	9	0.2999961	0.2172260	1.5482010	2.0158132	0.2731355	0.2475519	0.3556322
	10	0.4000078	0.1668695	1.1993726	1.8116831	0.2115948	0.1912280	0.3196193
	11	0.5	0.1280219	0.8775721	1.6248755	0.1548223	0.1464561	0.2866625
	12	0.5999922	0.1007380	0.6673089	1.4652915	0.1177275	0.1135140	0.2585085
	13	0.7000039	0.0807190	0.4757659	1.3239149	0.0839352	0.0902004	0.2335667
	14	0.7999961	0.0634822	0.3795804	1.2058817	0.0669660	0.0720347	0.2127431
	15	0.8999883	0.0462194	0.2323961	1.0977237	0.0409996	0.0549993	0.1936617
	16	1.0	0.0036716	0.1206011	1.0	0.0212766	0.0342129	0.1764212

Рис. 94 Информация о модели: таблица выигрышей, вычисленная на контрольной выборке

Последний элемент вывода – таблица с историей вычислений (рис. 85), где по каждой итерации приводится информация о временной метке итерации, продолжительности итерации, номере итерации, силе регуляризации (значении параметра `lambda_`), количестве активных предикторов, девиансе на обучающем наборе, девиансе на контрольном наборе.

Scoring History:

	timestamp	duration	iteration	lambda	predictors	deviance_train	deviance_test
	2018-11-05 14:45:58	0.000 sec	1	,63E-1	1	0.9301112	0.9318485
	2018-11-05 14:45:58	0.019 sec	2	,57E-1	3	0.9291239	0.9308659
	2018-11-05 14:45:58	0.039 sec	3	,52E-1	4	0.9280793	0.9298199
	2018-11-05 14:45:58	0.055 sec	4	,48E-1	4	0.9267472	0.9284690
	2018-11-05 14:45:59	0.079 sec	5	,43E-1	4	0.9256036	0.9273080
---	---	---	---	---	---	---	---
	2018-11-05 14:46:02	3.821 sec	102	,4E-4	167	0.8048477	0.8054281
	2018-11-05 14:46:02	3.875 sec	103	,37E-4	168	0.8047872	0.8054063
	2018-11-05 14:46:02	3.924 sec	104	,34E-4	171	0.8047212	0.8053784
	2018-11-05 14:46:02	3.975 sec	105	,31E-4	171	0.8046605	0.8053543
	2018-11-05 14:46:02	4.024 sec	106	,28E-4	171	0.8046098	0.8053395

See the whole table with table.as_data_frame()

Рис. 95 Информация о модели: история вычислений

Видим, что с уменьшением силы регуляризации (если задан поиск оптимального значения параметра `lambda_`) все большее количество предикторов остается в модели и обучение модели занимает все больше времени.
Если нужно вывести таблицу полностью, можно воспользоваться следующим программным кодом.

In[175]:
```
# выводим историю вычислений полностью
glm_model._model_json['output']['scoring_history'].as_data_frame()
```

Out[175]:

	timestamp	duration	iteration	lambda	predictors	deviance_train	deviance_test
0	2018-11-07 09:42:52	0.000 sec	1	,63E-1	1	0.930	0.932
1	2018-11-07 09:42:52	0.018 sec	2	,57E-1	3	0.929	0.931
2	2018-11-07 09:42:52	0.038 sec	3	,52E-1	4	0.928	0.930
3	2018-11-07 09:42:52	0.055 sec	4	,48E-1	4	0.927	0.928
4	2018-11-07 09:42:52	0.079 sec	5	,43E-1	4	0.926	0.927
5	2018-11-07 09:42:52	0.104 sec	6	,39E-1	5	0.924	0.926
6	2018-11-07 09:42:52	0.140 sec	8	,36E-1	9	0.920	0.921
7	2018-11-07 09:42:52	0.170 sec	10	,33E-1	9	0.914	0.915
8	2018-11-07 09:42:52	0.208 sec	12	,3E-1	10	0.908	0.909
9	2018-11-07 09:42:52	0.247 sec	14	,27E-1	11	0.903	0.903

. . . . . .

Давайте взглянем на значение `alpha` и оптимальное значение штрафного коэффициента `lambda_`, найденного в ходе перебора значений.

In[176]:
```
# смотрим значение alpha и оптимальное значение lambda_,
# найденное в ходе перебора значений lambda_
glm_model.summary()['regularization']
```

Out[176]:
```
['Elastic Net (alpha = 0.5, lambda = 2.784E-5 )']
```

С помощью метода `.predict()` можно вычислить спрогнозированные классы зависимой переменной и вероятности классов для контрольной выборки.

In[177]:
```
# вычисляем прогнозы на контрольной выборке
predictions = glm_model.predict(valid)
predictions
```

Out[177]:

predict	No	Yes
No	0.959395	0.0406053
Yes	0.604989	0.395011
Yes	0.688217	0.311783
No	0.913428	0.0865721
No	0.926099	0.0739008
No	0.923063	0.0769366
Yes	0.653976	0.346024
No	0.881133	0.118867
Yes	0.749682	0.250318
No	0.856596	0.143404

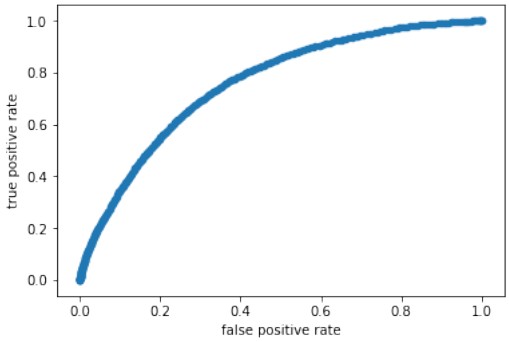
По умолчанию будут выведены результаты для первых 10 наблюдений. Для классификации H2O использует пороговое значение вероятности положительного класса, при котором достигается максимальное значение F1-меры. В данном случае мы вычисляем прогнозы для контрольной выборки, поэтому для классификации будет использоваться порог, при котором достигается максимальное значение F1-меры для контрольной выборки. Этот порог равен 0,204. Поэтому если вероятность положительного класса меньше порогового значения 0,204, прогнозируется класс 0, а если она больше порогового значения 0,204, прогнозируется класс 1.
С помощью функции `h2o.export_file()` можно сохранить прогнозы в отдельный CSV-файл.

In[178]:
```
# сохраняем прогнозы в CSV-файл
h2o.export_file(predictions, path='pred.csv', force=True)
```

Обратите внимание, что значение параметра `force=True` позволяет перезаписать файл, если он уже существует. Если же задано значение `False`, а файл с указанным названием уже есть, будет выдана ошибка.
Кроме того, с помощью метода `.roc()` можно построить ROC-кривую.

In[179]:
```
# выводим ROC-кривую для контрольной выборки
tmp = glm_model.roc(valid=True)
df = pd.DataFrame({'false positive rate': tmp[0], 'true positive rate': tmp[1]})
df.plot(kind='scatter', x='false positive rate', y='true positive rate')
```

Out[175]:



При желании можно вывести таблицу обычных и стандартизированных регрессионных коэффициентов.

In[180]:

```
# записываем таблицу коэффициентов
coeff_table = glm_model._model_json['output']['coefficients_table']

# преобразуем таблицу коэффициентов в датафрейм pandas
coeff_table.as_data_frame()
```

Out[180]:

	names	coefficients	standardized_coefficients
0	Intercept	-1.274	-1.274
1	living_region.ADYGEJA	0.354	0.354
2	living_region.ALTAJ	0.000	0.000
3	living_region.ALTAJSKIJ	-0.101	-0.101
4	living_region.AMURSKAJA	-0.030	-0.030
5	living_region.ARHANGEL'SKAJA	-0.048	-0.048
6	living_region.ASTRAHANSKAJA	-0.262	-0.262
7	living_region.BASHKORTOSTAN	-0.086	-0.086
8	living_region.BELGORODSKAJA	0.000	0.000
9	living_region.BRJANSKAJA	0.154	0.154
10	living_region.BURJATIJA	0.197	0.197

В модель можно добавить парные взаимодействия признаков и за счет этого улучшить качество модели.

In[181]:
```
# еще можно добавлять взаимодействия признаков

# создаем экземпляр класса H2OGeneralizedLinearEstimator,
# задавая lambda_search и список переменных для взаимодействий
glm_model2 = H2OGeneralizedLinearEstimator(model_id='logreg_credit2', family='binomial',
                                           lambda_search=True,
                                           interactions = ['monthcat',
                                                           'credsumcat',
                                                           'paymcat'])
# обучаем модель с взаимодействиями
glm_model2.train(predictors, dependent, training_frame=tr,
                 validation_frame=valid)
```

In[182]:
```
# смотрим модель с взаимодействиями
glm_model2
```

Чтобы вывести *p*-значения регрессионных коэффициентов, нужно выполнить четыре условия:

- задать вычисление *p*-значений (`compute_p_values=True`);
- отключить регуляризацию (`lambda_=0`);
- задать метод наименьших квадратов с итеративным пересчетом весов (`solver='IRLSM'`);
- удалить столбцы, которые коррелируют между собой (`remove_collinear_columns=True`).

In[183]:
```
# чтобы вычислить p-значения коэффициентов, нужно
# задать compute_p_values=True, отключить
# регуляризацию (lambda_=0), задать метод наименьших
# квадратов с итеративным пересчётом весов (solver='IRLSM'),
# рекомендуется задать параметр remove_collinear_columns

# создаем экземпляр класса H2OGeneralizedLinearEstimator
glm_model3 = H2OGeneralizedLinearEstimator(model_id='logreg_credit3', lambda_=0,
                                           family='binomial', solver='IRLSM',
                                           remove_collinear_columns=True,
                                           compute_p_values=True)
# обучаем модель
glm_model3.train(predictors, dependent, training_frame=tr,
                 validation_frame=valid)
```

In[184]:
```
# записываем таблицу коэффициентов
coeff_table = glm_model3._model_json['output']['coefficients_table']

# преобразуем таблицу коэффициентов в датафрейм pandas
coeff_table.as_data_frame()
```

Out[184]:

	names	coefficients	std_error	z_value	p_value	standardized_coefficients
0	Intercept	-0.995	0.556	-1.789	0.074	-0.995
1	living_region.ALTAJ	-0.333	0.451	-0.739	0.460	-0.333
2	living_region.ALTAJSKIJ	-0.538	0.193	-2.789	0.005	-0.538
3	living_region.AMURSKAJA	-0.456	0.170	-2.690	0.007	-0.456
4	living_region.ARHANGEL'SKAJA	-0.447	0.154	-2.898	0.004	-0.447
5	living_region.ASTRAHANSKAJA	-0.665	0.152	-4.382	0.000	-0.665
6	living_region.BASHKORTOSTAN	-0.481	0.139	-3.466	0.001	-0.481
7	living_region.BELGORODSKAJA	-0.390	0.179	-2.172	0.030	-0.390
8	living_region.BRJANSKAJA	-0.216	0.172	-1.261	0.207	-0.216
9	living_region.BURJATIJA	-0.172	0.158	-1.094	0.274	-0.172

Библиотека `h2o` предлагает богатые возможности для конструирования признаков. С помощью класса `TargetEncoder` мы можем выполнить кодирование средними значениями зависимой переменными, сглаженными через сигмоидальную функцию по схеме K-Fold или leave-one-out. Ниже приводится список параметров и гиперпараметров для класса `TargetEncoder`.

Параметр/гиперпараметр	Предназначение
`x`	Задает список предикторов для кодировки.
`blending_avg` (будет заменен на `blended_avg`)	Задает сглаживание среднего через сигмоидальную функцию.
`inflection_point`	Задает половину минимально допустимого размера категории, при которой мы полностью «доверяем» апостериорной вероятности.
`smoothing`	Задает угол наклона сигмоиды (скорость перехода от апостериорной вероятности к априорной).

Итак, давайте выполним для переменной *living_region* кодирование средними значениями зависимой переменными, сглаженными через сигмоидальную функцию по схеме K-Fold.

In[185]:

```
# импортируем класс TargetEncoder библиотеки H2O
from h2o.targetencoder import TargetEncoder

# создаем столбец с индексами блоков
tr['cv_fold_te'] = tr.kfold_column(n_folds=5, seed=42)

# создаем экземпляр класса TargetEncoder (модель)
targetEncoder = TargetEncoder(x=['living_region'],
                              blending_avg=True,
                              inflection_point=3,
                              smoothing=1,
                              y='open_account_flg',
                              fold_column='cv_fold_te')

# обучаем модель, т.е. создаем таблицу, в соответствии с которой
# категориям предиктора living_region будут сопоставлены средние
# значения зависимой переменной
targetEncoder.fit(tr)
```

In[186]:
```
# применяем модель к обучающей выборке, категории предиктора
# в обучающей выборке заменяются на сглаженные средние значения зависимой
# переменной
tr = targetEncoder.transform(frame=tr,
                             holdout_type='kfold',
                             noise=0
                            )

# создаем экземпляр класса TargetEncoder (модель)
# для контрольной выборки
targetEncoder_valid = TargetEncoder(x=['living_region'],
                                    y='open_account_flg',
                                    blending_avg=False)

# обучаем модель, т.е. создаем таблицу, в соответствии с которой
# категориям предиктора в контрольной выборке будут сопоставлены
# обычные средние значения зависимой переменной в этих категориях,
# вычисленные на обучающей выборке
targetEncoder_valid.fit(tr)

# применяем модель к контрольной выборке,
# категории предиктора в контрольной выборке заменяются на обычные
# средние значения зависимой переменной в этих категориях,
# вычисленные на обучающей выборке
valid = targetEncoder_valid.transform(frame=valid,
                                      holdout_type='none',
                                      noise=0
                                     )
```

Посмотрим на результаты кодировки.

In[187]:
```
# взглянем на результаты кодирования
print(tr[['cv_fold_te', 'living_region', 'living_region_te']])
print(valid[['living_region', 'living_region_te']])
```

Out[187]:

cv_fold_te	living_region	living_region_te
0	ADYGEJA	0.255892
0	ADYGEJA	0.255892
0	ADYGEJA	0.255892
0	ADYGEJA	0.255892
0	ADYGEJA	0.255892
0	ADYGEJA	0.255892
0	ADYGEJA	0.255892
0	ADYGEJA	0.255892
0	ADYGEJA	0.255892
0	ADYGEJA	0.255892

living_region	living_region_te
ADYGEJA	0.236702
ADYGEJA	0.236702
ADYGEJA	0.236702
ADYGEJA	0.236702
ADYGEJA	0.236702
ADYGEJA	0.236702
ADYGEJA	0.236702
ADYGEJA	0.236702
ADYGEJA	0.236702
ADYGEJA	0.236702

Заново сформируем список предикторов

In[188]:
```
# заново формируем список предикторов
independents = list(set(tr.col_names) - set(['cv_fold_te', 'open_account_flg', 'living_region']))
independents
```

Out[188]:
```
['credit_month_sq',
 'tariff',
 'tariff_sq',
 'income_sq',
 'match',
 'monthcat',
 'paymcat',
 'tariff_id',
 'paym',
 'credit_sum_sq',
 'gender',
 'marital_status',
 'credit_count_sq',
 'pti',
 'job_position',
 'credit_count',
 'credsumcat',
 'score_shk',
 'overdue_credit_count',
 'age_sq',
 'score_sq',
 'credit_month',
 'living_region_te',
 'education',
 'agecat',
 'age',
 'ind',
 'credit_sum',
 'monthly_income']
```

Теперь давайте обучим модель логистической регрессии на наборе, в котором к переменной *living_region* применено кодирование средними значениями зависимой переменной.

In[189]:
```
# обучаем модель на наборе, в котором к переменной living_region
# применено кодирование средними значениями зависимой переменной
glm_model4 = H2OGeneralizedLinearEstimator(model_id='logreg_credit4', family='binomial',
                                           lambda_search=True)
glm_model4.train(independents, dependent,
                 training_frame=tr, validation_frame=valid)
glm_model4
```

В данном случае этот вид кодировки не позволил улучшить модель.
Теперь перейдем к решетчатому поиску. Обычная практика решетчатого поиска в H2O заключается в том, чтобы включить поиск оптимального значения штрафного коэффициента `lambda_` и перебор значений `alpha` (при этом достаточно передать в сетку параметров значения `alpha`, а для перебора значений `lambda_` достаточно задать `lambda_search=True`).
Для более надежной проверки качества моделей воспользуемся 5-блочной перекрестной проверкой. В ходе *k*-блочной перекрестной проверки для каждой комбинации значений гиперпараметров `alpha` и `lambda_` строится *k*+1 моделей. В нашем случае мы используем 5-блочную перекрестную проверку, поэтому для каждой комбинации параметров будет построено 6 моделей: 5 моделей перекрестной проверки и одна общая модель. Первые 5 моделей (их называют

моделями перекрестной проверки) строятся на 4 обучающих блоках (80% исходного обучающего набора) и проверяются на одном контрольном блоке (20% исходного обучающего набора). Прогнозы, полученные по 5 контрольным блокам перекрестной проверки, объединяются и вычисляются метрики качества на основе объединенных прогнозов (эти метрики будут приведены в отчете о построении модели в разделе `Reported on cross-validation data`). Кроме того, метрики качества, полученные по 5 контрольным блокам, усредняются, и мы получаем усредненные метрики перекрестной проверки (именно эти усредненные метрики наряду с метриками по каждому контрольному блоку будут приведены в отчете о построении модели в разделе `Cross-Validation Metrics Summary`). Потом строится общая модель на 100% обучающих данных и мы получаем метрики качества на всей обучающей выборке (эти метрики будут приведены в отчете о построении модели в разделе `Reported on train data`). Если задана контрольная выборка, то вычисляются метрики качества на контрольной выборке (данные метрики будут приведены в отчете о построении модели в разделе `Reported on validation data`).

In[190]:
```
# импортируем класс H2OGridSearch для выполнения решетчатого поиска
from h2o.grid.grid_search import H2OGridSearch

# задаем сетку параметров, будем перебирать разные значения alpha,
# alpha определяет тип регуляризации: значение 1 соответствует
# l1-регуляризации (лассо), значение 0 соответствует l2-регуляризации
# (гребневой регрессии), промежуточное значение соответствует
# комбинации штрафов l1 и l2 (эластичной сети)
hyper_parameters = {'alpha':[0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]}


# создаем экземпляр класса H2OGridSearch, lambda_search
# задает перебор значений lambda - силы регуляризации
gridsearch = H2OGridSearch(H2OGeneralizedLinearEstimator(family='binomial',
                                                         lambda_search=True),
                           grid_id='gr_glm_outputs', hyper_params=hyper_parameters)

# подгоняем модели решетчатого поиска
gridsearch.train(predictors, dependent,
                 training_frame=tr,
                 nfolds=5,
                 keep_cross_validation_predictions=True,
                 seed=1000000)
```

Давайте выведем информацию о полученных моделях.

In[191]:
```
# выводим результаты решетчатого поиска
gridsearch.show()
```

Out[191]:
```
      alpha     model_ids              logloss
0     [0.8]   res_model_8    0.403783904581411
1     [1.0]  res_model_10   0.4037843646936971
2     [0.0]   res_model_0  0.40378490702836994
3     [0.9]   res_model_9    0.403787337526566
4     [0.7]   res_model_7   0.4038012885552281
5     [0.6]   res_model_6   0.4038192036403213
6     [0.5]   res_model_5  0.40385067347268233
7     [0.4]   res_model_4   0.4038592327919046
8     [0.3]   res_model_3   0.4038774032189474
```

```
9     [0.2]   res_model_2  0.40390495413889543
10    [0.1]   res_model_1   0.4039336134105529
```

В выводе приводится информация по каждой модели, построенной в ходе решетчатого поиска. По умолчанию модели приводятся в порядке увеличения логистической функции потерь, вычисленной по объединенным прогнозам в контрольных блоках. Для удобства отсортируем полученные модели в порядке убывания AUC.

In[192]:
```
# сортируем результаты решетчатого поиска
# по убывания AUC
gridperf = gridsearch.get_grid(sort_by='auc', decreasing=True)
gridperf
```

Out[192]:
```
      alpha     model_ids                  auc
0     [0.0]   res_model_0  0.7549188411754277
1     [0.8]   res_model_8  0.7548821068581111
2     [0.9]   res_model_9  0.7548660442881122
3     [1.0]  res_model_10  0.7548519612542383
4     [0.7]   res_model_7   0.754840981192143
5     [0.6]   res_model_6  0.7548201500579959
6     [0.5]   res_model_5  0.7547736052398202
7     [0.4]   res_model_4  0.7547638118851201
8     [0.3]   res_model_3  0.7547525608654144
9     [0.2]   res_model_2  0.7547258222757727
10    [0.1]   res_model_1  0.7546686697810946
```

Наилучшей моделью стала модель с `alpha=0`, которая имеет значение AUC 0,755. Теперь извлекаем ее, при желании можно взглянуть на нее.

In[193]:
```
# извлекаем наилучшую модель
best_model = gridperf.models[0]
best_model
```

In[194]:
```
# смотрим AUC наилучшей модели
# на контрольной выборке
bestmodel_perf = best_model.model_performance(valid)
print(bestmodel_perf.auc())
```

Out[194]:
```
0.75953443732174
```

Снова выведем оптимальные значения `alpha` и `lambda_`.

In[195]:
```
# смотрим оптимальные значения lambda_ и alpha
best_model.summary()['regularization']
```

Out[195]:
```
['Ridge ( lambda = 5.058E-6 )']
```

В нашем случае оптимальными значениями стали `alpha=0` и `lambda_=5.058E-6`.
Завершаем работу с H2O.

In[196]:
```
# завершаем работу с H2O
h2o.cluster().shutdown()
```

ПРИЛОЖЕНИЕ 5. ПРИМЕР ПРЕДВАРИТЕЛЬНОЙ ПОДГОТОВКИ ДАННЫХ В PANDAS (КОНКУРСНАЯ ЗАДАЧА ПРЕДСКАЗАНИЯ ОТКЛИКА ОТП БАНКА)

Материал подготовлен директором по научной работе исследовательского центра «Гевисста» Артемом Груздевым.

В этом приложении речь пойдет о решении конкурсной задачи предсказания отклика ОТП Банка (обсуждение задачи можно найти на сайте http://www.machinelearning.ru, а обсуждение итогов конкурса – в презентации http://www.forecsys.ru/get_file.php?id=558). Необходимые нам данные записаны в файле *Credit_OTP.csv*. Исходная выборка содержит записи о 15223 клиентах, классифицированных на два класса: 0 – отклика не было (13 411 клиентов) и 1 – отклик был (1812 клиентов). По каждому наблюдению (клиенту) фиксируются 52 исходные переменные.

Список исходных переменных включает в себя:

- категориальный предиктор *Уникальный идентификатор объекта в выборке [AGREEMENT_RK]*;
- категориальная зависимая переменная *Отклик на маркетинговую кампанию [TARGET]*;
- количественный предиктор *Возраст клиента [AGE]*;
- категориальный предиктор *Социальный статус клиента относительно работы [SOCSTATUS_WORK_FL]*;
- категориальный предиктор *Социальный статус клиента относительно пенсии [SOCSTATUS_PENS_FL]*;
- категориальный предиктор *Пол клиента [GENDER]*;
- количественный предиктор *Количество детей клиента [CHILD_TOTAL]*;
- количественный предиктор *Количество иждивенцев клиента [DEPENDANTS]*;
- категориальный предиктор *Образование [EDUCATION]*;
- категориальный предиктор *Семейное положение [MARITAL_STATUS]*;
- категориальный предиктор *Отрасль работы клиента [GEN_INDUSTRY]*;
- категориальный предиктор *Должность [GEN_TITLE]*;
- категориальный предиктор *Форма собственности компании [ORG_TP_STATE]*;
- категориальный предиктор *Отношение к иностранному капиталу [ORG_TP_FCAPITAL]*;

- категориальный предиктор *Направление деятельности внутри компании [JOB_DIR]*;
- категориальный предиктор *Семейный доход [FAMILY_INCOME]*;
- количественный предиктор *Личный доход клиента в рублях [PERSONAL_INCOME]*;
- категориальный предиктор *Область регистрации клиента [REG_ADDRESS_PROVINCE]*;
- категориальный предиктор *Область фактического пребывания клиента [FACT_ADDRESS_PROVINCE]*;
- категориальный предиктор *Почтовый адрес область [POSTAL_ADDRESS_PROVINCE]*;
- категориальный предиктор *Область торговой точки, где клиент брал последний кредит [TP_PROVINCE]*;
- категориальный предиктор *Регион РФ [REGION_NM]*;
- категориальный предиктор *Адрес регистрации и адрес фактического пребывания клиента совпадают [REG_FACT_FL]*;
- категориальный предиктор *Адрес фактического пребывания клиента и его почтовый адрес совпадают [FACT_POST_FL]*;
- категориальный предиктор *Адрес регистрации клиента и его почтовый адрес совпадают [REG_POST_FL]*;
- категориальный предиктор *Почтовый, фактический и адрес регистрации совпадают [REG_FACT_POST_FL]*;
- категориальный предиктор *Область регистрации, фактического пребывания, почтового адреса и область расположения торговой точки, где клиент брал кредит совпадают [REG_FACT_POST_TP_FL]*;
- категориальный предиктор *Наличие в собственности квартиры [FL_PRESENCE_FL]*;
- количественный предиктор *Количество автомобилей в собственности [OWN_AUTO]*;
- категориальный предиктор *Наличие в собственности автомобиля российского производства [AUTO_RUS_FL]*;
- категориальный предиктор *Наличие в собственности загородного дома [HS_PRESENCE_FL]*;
- категориальный предиктор *Наличие в собственности коттеджа [COT_PRESENCE_FL]*;
- категориальный предиктор *Наличие в собственности гаража [GAR_PRESENCE_FL]*;
- категориальный предиктор *Наличие в собственности земельного участка [LAND_PRESENCE_FL]*;
- количественный предиктор *Сумма последнего кредита клиента в рублях [CREDIT]*;
- количественный предиктор *Срок кредита [TERM]*;

- количественный предиктор *Первоначальный взнос в рублях [FST_PAYMENT]*;
- категориальный предиктор *В анкете клиент указал водительское удостоверение [DL_DOCUMENT_FL]*;
- категориальный предиктор *В анкете клиент указал ГПФ [GPF_DOCUMENT_FL]*;
- количественный предиктор *Количество месяцев проживания по месту фактического пребывания [FACT_LIVING_TERM]*;
- количественный предиктор *Время работы на текущем месте в месяцах [WORK_TIME]*;
- категориальный предиктор *Наличие в заявке телефона по фактическому месту пребывания [FACT_PHONE_FL]*;
- категориальный предиктор *Наличие в заявке телефона по месту регистрации [REG_PHONE_FL]*;
- категориальный предиктор *Наличие в заявке рабочего телефона [GEN_PHONE_FL]*;
- количественный предиктор *Количество ссуд клиента [LOAN_NUM_TOTAL]*;
- количественный предиктор *Количество погашенных ссуд клиента [LOAN_NUM_CLOSED]*;
- количественный предиктор *Количество платежей, которые сделал клиент [LOAN_NUM_PAYM]*;
- количественный предиктор *Количество просрочек, допущенных клиентом [LOAN_DLQ_NUM]*;
- количественный предиктор *Номер максимальной просрочки, допущенной клиентом [LOAN_MAX_DLQ]*;
- количественный предиктор *Средняя сумма просрочки в рублях [LOAN_AVG_DLQ_AMT]*;
- количественный предиктор *Максимальная сумма просрочки в рублях [LOAN_MAX_DLQ_AMT]*;
- количественный предиктор *Количество уже утилизированных карт [PREVIOUS_CARD_NUM_UTILIZED].*

Кроме того, в нашем распоряжении имеется файл новых данных *Credit_OTP_new.csv* с известными значениями зависимой переменной.

Задача состоит в том, чтобы с помощью метода логистической регрессии построить модель прогнозирования отклика. Построение модели будет состоять из двух этапов. На первом этапе мы разобьем историческую выборку на обучающую и контрольную. Контрольную выборку используем для подбора наилучших преобразований, конструирования наиболее полезных признаков. Затем для подбора оптимальных значений параметров воспользуемся комбинированной проверкой: на обучающих блоках перекрестной проверки будем строить модели с разными значениями параметров, на контрольных блоках перекрестной проверки настраивать параметры, в результате выберем модель с

комбинацией оптимальных значений параметров, обучим на всей обучающей выборке и проверим на контрольной. В рамках второго этапа мы обучим модель с наилучшими преобразованиями, наиболее полезными признаками и оптимальными значениями параметров на всей исторической выборке и применим модель к выборке новых данных.

Этап I. Построение модели на обучающей выборке - части исторической выборки и ее проверка на контрольной выборке - части исторической выборки

I.1. Считывание CSV-файла, содержащего исторические данные, в объект DataFrame

Сначала импортируем необходимые библиотеки.

```
In[1]:
# отключаем предупреждения Anaconda
import warnings
warnings.simplefilter('ignore')

# импортируем необходимые библиотеки
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
plt.rc('font', family='Verdana')
```

Увеличиваем количество отображаемых столбцов и записываем файл *Credit_OTP.csv* в датафрейм `data` с помощью функции `pd.read_csv()`. У нас в файле используется кириллица, поэтому с помощью параметра `encoding`, определяющего тип кодировки, задаем значение `'cp1251'`. Поскольку в этом CSV-файле в качестве символа-разделителя используется точка с запятой, с помощью параметра `sep`, определяющего тип символа-разделителя, задаем значение `';'`.

```
In[2]:
# увеличиваем количество выводимых столбцов
pd.set_option('display.max_columns', 60)
# загружаем набор данных
data = pd.read_csv('Data/Credit_OTP.csv', encoding='cp1251', sep=';')
# выводим первые 5 наблюдений
data.head()
```

Out[2]:

	AGREEMENT_RK	TARGET	AGE	SOCSTATUS_WORK_FL	SOCSTATUS_PENS_FL	GENDER	CHILD_TOTAL	DEPENDANTS	EDUCATION	MARITAL_STATUS
0	59910150	0	49	1	0	1	2	1	Среднее специальное	Состою в браке
1	59910230	0	32	1	0	1	3	3	Среднее	Состою в браке
2	59910525	0	52	1	0	1	4	0	Неполное среднее	Состою в браке
3	59910803	0	39	1	0	1	1	1	Высшее	Состою в браке
4	59911781	0	30	1	0	0	0	0	Среднее	Состою в браке

I.2. Преобразование типов переменных

Теперь выведем информацию о количестве пропусков и типах переменных.

In[3]:
```
# выводим информацию о количестве непропущенных
# наблюдений в переменных и типах переменных
data.info()
```

Out[3]:
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 15223 entries, 0 to 15222
Data columns (total 52 columns):
AGREEMENT_RK                15223 non-null int64
TARGET                      15223 non-null int64
AGE                         15223 non-null int64
SOCSTATUS_WORK_FL           15223 non-null int64
SOCSTATUS_PENS_FL           15223 non-null int64
GENDER                      15223 non-null int64
CHILD_TOTAL                 15223 non-null int64
DEPENDANTS                  15223 non-null int64
EDUCATION                   15223 non-null object
MARITAL_STATUS              15223 non-null object
GEN_INDUSTRY                13856 non-null object
GEN_TITLE                   13856 non-null object
ORG_TP_STATE                13856 non-null object
ORG_TP_FCAPITAL             13858 non-null object
JOB_DIR                     13856 non-null object
FAMILY_INCOME               15223 non-null object
PERSONAL_INCOME             15223 non-null object
REG_ADDRESS_PROVINCE        15223 non-null object
FACT_ADDRESS_PROVINCE       15223 non-null object
POSTAL_ADDRESS_PROVINCE     15223 non-null object
TP_PROVINCE                 14928 non-null object
REGION_NM                   15222 non-null object
REG_FACT_FL                 15223 non-null int64
FACT_POST_FL                15223 non-null int64
REG_POST_FL                 15223 non-null int64
REG_FACT_POST_FL            15223 non-null int64
REG_FACT_POST_TP_FL         15223 non-null int64
FL_PRESENCE_FL              15223 non-null int64
OWN_AUTO                    15223 non-null int64
AUTO_RUS_FL                 15223 non-null int64
HS_PRESENCE_FL              15223 non-null int64
COT_PRESENCE_FL             15223 non-null int64
GAR_PRESENCE_FL             15223 non-null int64
LAND_PRESENCE_FL            15223 non-null int64
CREDIT                      15223 non-null object
TERM                        15223 non-null int64
FST_PAYMENT                 15223 non-null object
DL_DOCUMENT_FL              15223 non-null int64
GPF_DOCUMENT_FL             15223 non-null int64
FACT_LIVING_TERM            15223 non-null int64
WORK_TIME                   13855 non-null float64
FACT_PHONE_FL               15223 non-null int64
```

```
REG_PHONE_FL                  15223 non-null int64
GEN_PHONE_FL                  15223 non-null int64
LOAN_NUM_TOTAL                15223 non-null int64
LOAN_NUM_CLOSED               15223 non-null int64
LOAN_NUM_PAYM                 15223 non-null int64
LOAN_DLQ_NUM                  15223 non-null int64
LOAN_MAX_DLQ                  15223 non-null int64
LOAN_AVG_DLQ_AMT              15223 non-null object
LOAN_MAX_DLQ_AMT              15223 non-null object
PREVIOUS_CARD_NUM_UTILIZED    288 non-null float64
dtypes: float64(2), int64(32), object(18)
memory usage: 6.0+ MB
```

Переменные *WORK_TIME*, *GEN_INDUSTRY*, *GEN_TITLE*, *ORG_TP_STATE*, *ORG_TP_FCAPITAL* и *JOB_DIR* имеют одинаковое количество пропусков. Такое часто бывает, когда у наблюдений имеются одновременные пропуски сразу по нескольким переменным. Это говорит о наличии определенного паттерна в данных. Видим, что у многих переменных тип определен неправильно, идентификационная переменная *AGREEMENT_RK* неверно записана как количественная, все категориальные переменные-флаги, принимающие значение 0 или 1, были также неверно записаны как количественные. Некоторые количественные переменные (*PERSONAL_INCOME*, *CREDIT*, *FST_PAYMENT*, *LOAN_AVG_DLQ_AMT*, *LOAN_MAX_DLQ_AMT*) были неверно записаны как категориальные из-за того, что в качестве десятичного разделителя вместо точки использовалась запятая. Давайте выполним преобразование типов.

In[4]:
```
# переменные, неверно записанные как количественные,
# преобразуем в тип object
for i in ['AGREEMENT_RK', 'TARGET', 'SOCSTATUS_WORK_FL',
          'SOCSTATUS_PENS_FL', 'GENDER',
          'REG_FACT_FL', 'FACT_POST_FL', 'REG_POST_FL',
          'REG_FACT_POST_FL', 'REG_FACT_POST_TP_FL', 'FL_PRESENCE_FL',
          'AUTO_RUS_FL', 'HS_PRESENCE_FL', 'COT_PRESENCE_FL',
          'GAR_PRESENCE_FL', 'LAND_PRESENCE_FL', 'DL_DOCUMENT_FL',
          'GPF_DOCUMENT_FL', 'FACT_PHONE_FL', 'REG_PHONE_FL', 'GEN_PHONE_FL']:
    data[i] = data[i].astype('object')

# в указанных переменных заменяем запятую на точку в качестве
# десятичного разделителя и преобразуем в тип float
for i in ['PERSONAL_INCOME', 'CREDIT', 'FST_PAYMENT',
          'LOAN_AVG_DLQ_AMT', 'LOAN_MAX_DLQ_AMT']:
    data[i] = data[i].str.replace(',', '.').astype('float')

# вновь смотрим типы переменных
data.info()
```

Out[4]:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 15223 entries, 0 to 15222
Data columns (total 52 columns):
AGREEMENT_RK                    15223 non-null object
TARGET                          15223 non-null object
AGE                             15223 non-null int64
SOCSTATUS_WORK_FL               15223 non-null object
SOCSTATUS_PENS_FL               15223 non-null object
GENDER                          15223 non-null object
CHILD_TOTAL                     15223 non-null int64
DEPENDANTS                      15223 non-null int64
EDUCATION                       15223 non-null object
MARITAL_STATUS                  15223 non-null object
GEN_INDUSTRY                    13856 non-null object
GEN_TITLE                       13856 non-null object
ORG_TP_STATE                    13856 non-null object
ORG_TP_FCAPITAL                 13858 non-null object
JOB_DIR                         13856 non-null object
FAMILY_INCOME                   15223 non-null object
PERSONAL_INCOME                 15223 non-null float64
REG_ADDRESS_PROVINCE            15223 non-null object
FACT_ADDRESS_PROVINCE           15223 non-null object
POSTAL_ADDRESS_PROVINCE         15223 non-null object
TP_PROVINCE                     14928 non-null object
REGION_NM                       15222 non-null object
REG_FACT_FL                     15223 non-null object
FACT_POST_FL                    15223 non-null object
REG_POST_FL                     15223 non-null object
REG_FACT_POST_FL                15223 non-null object
REG_FACT_POST_TP_FL             15223 non-null object
FL_PRESENCE_FL                  15223 non-null object
OWN_AUTO                        15223 non-null int64
AUTO_RUS_FL                     15223 non-null object
HS_PRESENCE_FL                  15223 non-null object
COT_PRESENCE_FL                 15223 non-null object
GAR_PRESENCE_FL                 15223 non-null object
LAND_PRESENCE_FL                15223 non-null object
CREDIT                          15223 non-null float64
TERM                            15223 non-null int64
FST_PAYMENT                     15223 non-null float64
DL_DOCUMENT_FL                  15223 non-null object
GPF_DOCUMENT_FL                 15223 non-null object
FACT_LIVING_TERM                15223 non-null int64
WORK_TIME                       13855 non-null float64
FACT_PHONE_FL                   15223 non-null object
REG_PHONE_FL                    15223 non-null object
GEN_PHONE_FL                    15223 non-null object
LOAN_NUM_TOTAL                  15223 non-null int64
LOAN_NUM_CLOSED                 15223 non-null int64
LOAN_NUM_PAYM                   15223 non-null int64
LOAN_DLQ_NUM                    15223 non-null int64
LOAN_MAX_DLQ                    15223 non-null int64
LOAN_AVG_DLQ_AMT                15223 non-null float64
LOAN_MAX_DLQ_AMT                15223 non-null float64
PREVIOUS_CARD_NUM_UTILIZED      288 non-null float64
dtypes: float64(7), int64(11), object(34)
memory usage: 6.0+ MB
```

Видим, что все переменные получили корректные типы данных.

I.3. Импутация пропусков, не использующая результаты математических вычислений (импутация, которую можно выполнять до/после разбиения на обучение/контроль)

Теперь займемся импутацией пропусков, которая не использует результаты математических вычислений и поэтому ее можно выполнять как до, так и после разбиения на обучающую и контрольную выборки.

Пропуски в переменных *GEN_INDUSTRY*, *GEN_TITLE*, *ORG_TP_STATE*, *ORG_TP_FCAPITAL*, *JOB_DIR* и *WORK_TIME* скорее всего обусловлены неприменимостью вопроса о занятости для клиентов, являющихся пенсионерами. Мы заменим пропуски в переменных *GEN_INDUSTRY*, *GEN_TITLE*, *ORG_TP_STATE* и *ORG_TP_FCAPITAL* меткой «Не указано», если в соответствующей переменной есть пропуск и при этом переменная *SOCSTATUS_PENS_FL* имеет значение 1 (т.е. клиент является пенсионером), в противном случае вернем исходное значение соответствующей переменной. Пропуски в переменных *JOB_DIR*, *TP_PROVINCE* и *REGION_NM* заменяем меткой «Не указано». Если в переменной *PREVIOUS_CARD_NUM_UTILIZED* есть пропуск, заменяем его нулем.

In[5]:

```
# если в интересующей нас переменной есть пропуск
# и при этом переменная SOCSTATUS_PENS_FL имеет значение 1,
# заменяем такие пропуски меткой "Не указано"
data['GEN_INDUSTRY'] = np.where(data['GEN_INDUSTRY'].isnull() \
                                | (data['SOCSTATUS_PENS_FL'] == 1),
                                'Не указано', data['GEN_INDUSTRY'])
data['GEN_TITLE'] = np.where(data['GEN_TITLE'].isnull()  \
                             | (data['SOCSTATUS_PENS_FL'] == 1),
                             'Не указано', data['GEN_TITLE'])
data['ORG_TP_STATE'] = np.where(data['ORG_TP_STATE'].isnull() \
                                | (data['SOCSTATUS_PENS_FL'] == 1),
                                'Не указано', data['ORG_TP_STATE'])
data['ORG_TP_FCAPITAL'] = np.where(data['ORG_TP_FCAPITAL'].isnull() \
                                   | (data['SOCSTATUS_PENS_FL'] == 1),
                                   'Не указано', data['ORG_TP_FCAPITAL'])

# заменяем пропуски в указанных переменных
# меткой "Не указано"
data['JOB_DIR'] = np.where(data['JOB_DIR'].isnull(), 'Не указано', data['JOB_DIR'])
data['REGION_NM'] = np.where(data['REGION_NM'].isnull(), 'Не указано', data['REGION_NM'])

# пропуски в переменной TP_PROVINCE заменим значением
# переменной FACT_ADDRESS_PROVINCE
data['TP_PROVINCE'] = np.where(data['TP_PROVINCE'].isnull(),
                               data['FACT_ADDRESS_PROVINCE'], data['TP_PROVINCE'])

# заменяем пропуски в переменной
# PREVIOUS_CARD_NUM_UTILIZED нулями
data['PREVIOUS_CARD_NUM_UTILIZED'] = np.where(data['PREVIOUS_CARD_NUM_UTILIZED'].isnull(), 0,
                                              data['PREVIOUS_CARD_NUM_UTILIZED'])
# смотрим количество пропусков
data.info()
```

```
Out[5]:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 15223 entries, 0 to 15222
Data columns (total 52 columns):
AGREEMENT_RK                   15223 non-null object
TARGET                         15223 non-null object
AGE                            15223 non-null int64
SOCSTATUS_WORK_FL              15223 non-null object
SOCSTATUS_PENS_FL              15223 non-null object
GENDER                         15223 non-null object
CHILD_TOTAL                    15223 non-null int64
DEPENDANTS                     15223 non-null int64
EDUCATION                      15223 non-null object
MARITAL_STATUS                 15223 non-null object
GEN_INDUSTRY                   15223 non-null object
GEN_TITLE                      15223 non-null object
ORG_TP_STATE                   15223 non-null object
ORG_TP_FCAPITAL                15223 non-null object
JOB_DIR                        15223 non-null object
FAMILY_INCOME                  15223 non-null object
PERSONAL_INCOME                15223 non-null float64
REG_ADDRESS_PROVINCE           15223 non-null object
FACT_ADDRESS_PROVINCE          15223 non-null object
POSTAL_ADDRESS_PROVINCE        15223 non-null object
TP_PROVINCE                    15223 non-null object
REGION_NM                      15223 non-null object
REG_FACT_FL                    15223 non-null object
FACT_POST_FL                   15223 non-null object
REG_POST_FL                    15223 non-null object
REG_FACT_POST_FL               15223 non-null object
REG_FACT_POST_TP_FL            15223 non-null object
FL_PRESENCE_FL                 15223 non-null object
OWN_AUTO                       15223 non-null int64
AUTO_RUS_FL                    15223 non-null object
HS_PRESENCE_FL                 15223 non-null object
COT_PRESENCE_FL                15223 non-null object
GAR_PRESENCE_FL                15223 non-null object
LAND_PRESENCE_FL               15223 non-null object
CREDIT                         15223 non-null float64
TERM                           15223 non-null int64
FST_PAYMENT                    15223 non-null float64
DL_DOCUMENT_FL                 15223 non-null object
GPF_DOCUMENT_FL                15223 non-null object
FACT_LIVING_TERM               15223 non-null int64
WORK_TIME                      13855 non-null float64
FACT_PHONE_FL                  15223 non-null object
REG_PHONE_FL                   15223 non-null object
GEN_PHONE_FL                   15223 non-null object
LOAN_NUM_TOTAL                 15223 non-null int64
LOAN_NUM_CLOSED                15223 non-null int64
LOAN_NUM_PAYM                  15223 non-null int64
LOAN_DLQ_NUM                   15223 non-null int64
LOAN_MAX_DLQ                   15223 non-null int64
LOAN_AVG_DLQ_AMT               15223 non-null float64
LOAN_MAX_DLQ_AMT               15223 non-null float64
PREVIOUS_CARD_NUM_UTILIZED     15223 non-null float64
dtypes: float64(7), int64(11), object(34)
memory usage: 6.0+ MB
```

I.4. Обработка редких категорий

Сейчас мы выполним обработку редких категорий. Выделим категориальные предикторы в отдельную группу и выведем статистики по ним.

In[6]:
```
# выделяем категориальные переменные в группу
categorical_columns = [c for c in data.columns if data[c].dtype.name == 'object']

# выводим статистики по категориальным переменным,
# смотрим unique - количество уникальных значений,
# выявляем бесполезные переменные - переменные, у
# которых уникальных значений столько, сколько
# наблюдений (AGREEMENT_RK), и переменные с одним
# уникальным значением (DL_DOCUMENT_FL)
data[categorical_columns].describe()
```

	AGREEMENT_RK	TARGET	SOCSTATUS_WORK_FL	SOCSTATUS_PENS_FL	GENDER	EDUCATION	MARITAL_STATUS	GEN_INDUSTRY	GEN_TITLE
count	15223	15223	15223	15223	15223	15223	15223	15223	15223
unique	15223	2	2	2	2	7	5	32	13
top	64585727	0	1	0	1	Среднее специальное	Состою в браке	Торговля	Специалист
freq	1	13411	13847	13176	9964	6518	9416	2303	6680

В этой сводке нас будет интересовать строка `unique` – количество уникальных значений. С помощью нее мы выявляем бесполезные переменные - переменные, у которых уникальных значений столько, сколько наблюдений (в нашем случае такой переменной будет *AGREEMENT_RK*), и переменные с одним уникальным значением (переменная *DL_DOCUMENT_FL*). Давайте удалим переменные *AGREEMENT_RK* и *DL_DOCUMENT_FL*.

In[7]:
```
# удаляем идентификационную переменную AGREEMENT_RK,
# потому что у нее количество уникальных значений
# равно количеству наблюдений
data.drop('AGREEMENT_RK', axis=1, inplace=True)

# удаляем переменную DL_DOCUMENT_FL, потому что
# у нее одно уникальное значение
data.drop('DL_DOCUMENT_FL', axis=1, inplace=True)
```

Заново выделяем категориальные переменные в группу и с помощью цикла `for` выводим частоты по каждой категориальной переменной, сейчас нам необходимо выявить редкие категории.

In[8]:
```
# заново выделим категориальные переменные в группу
# и пробежим по ней, выводя частоты категорий
# по каждой категориальной переменной, чтобы
# выявить редкие категории
categorical_columns = [c for c in data.columns if data[c].dtype.name == 'object']
for c in categorical_columns:
    print(data[c].value_counts(dropna=False))
```

В переменной *REGION_NM* заменим категорию `Не указано`, состоящую из 1 наблюдения, на самую часто встречающуюся категорию `ЮЖНЫЙ`. При этом замечаем, что в этой переменной есть некорректная категория `ПОВОЛЖСКИЙ`, которую мы заменим на `ПРИВОЛЖСКИЙ`. В переменных *REG_ADDRESS_PROVINCE*, *POSTAL_ADDRESS_PROVINCE* и *FACT_ADDRESS_PROVINCE* редкие категории `Москва`, `Хакасия`, `Ямало-Ненецкий АО`, `Магаданская область`, `Калмыкия`, `Дагестан`, `Агинский Бурятский АО`, `Усть-Ордынский Бурятский АО`, `Эвенкийский АО`, `Коми-Пермяцкий АО`, `Чечня` запишем в отдельную категорию `ДРУГОЕ`. В

переменной *TP_PROVINCE* в отдельную категорию ДРУГОЕ запишем редкие категории Сахалинская область, Еврейская АО, Магаданская область, Москва, Кабардино-Балкария.

In[9]:
```
# заменяем категорию "Не указано" на категорию "ЮЖНЫЙ"
data['REGION_NM'] = np.where(data['REGION_NM'] == 'Не указано', 'ЮЖНЫЙ',
                             data['REGION_NM'])

# заменяем неверную категорию "ПОВОЛЖСКИЙ" на категорию "ПРИВОЛЖСКИЙ"
data.at[data['REGION_NM'] == 'ПОВОЛЖСКИЙ', 'REGION_NM'] = 'ПРИВОЛЖСКИЙ'

# записываем редкие категории в одну отдельную категорию
for i in ['REG_ADDRESS_PROVINCE', 'POSTAL_ADDRESS_PROVINCE', 'FACT_ADDRESS_PROVINCE']:
    data[i] = np.where((data[i] == 'Москва') \
                       | (data[i] == 'Хакасия') \
                       | (data[i] == 'Ямало-Ненецкий АО') \
                       | (data[i] == 'Магаданская область') \
                       | (data[i] == 'Калмыкия') \
                       | (data[i] == 'Дагестан') \
                       | (data[i] == 'Агинский Бурятский АО') \
                       | (data[i] == 'Усть-Ордынский Бурятский АО') \
                       | (data[i] == 'Эвенкийский АО') \
                       | (data[i] == 'Коми-Пермяцкий АО') \
                       | (data[i] == 'Чечня'),
                      'ДРУГОЕ', data[i])

data['TP_PROVINCE'] = np.where((data['TP_PROVINCE'] == 'Сахалинская область') \
                               | (data['TP_PROVINCE'] == 'Еврейская АО') \
                               | (data['TP_PROVINCE'] == 'Магаданская область') \
                               | (data['TP_PROVINCE'] == 'Дагестан') \
                               | (data['TP_PROVINCE'] == 'Кабардино-Балкария'),
                              'ДРУГОЕ', data['TP_PROVINCE'])
```

Теперь выполним укрупнений категорий для переменных *EDUCATION*, *GEN_INDUSTRY*, *GEN_TITLE*, *ORG_TP_STATE* и *JOB_DIR*.

In[10]:
```
# укрупняем категории переменной EDUCATION
data.at[data['EDUCATION'] == 'Ученая степень', 'EDUCATION'] = 'Высшее'
data.at[data['EDUCATION'] == 'Два и более высших образования', 'EDUCATION'] = 'Высшее'

# смотрим частоты категорий переменной EDUCATION
data['EDUCATION'].value_counts(dropna=False)
```

Out[10]:
```
Среднее специальное    6518
Среднее                4679
Высшее                 3154
Неоконченное высшее     532
Неполное среднее        340
Name: EDUCATION, dtype: int64
```

In[11]:
```
# записываем некоторые категории переменной GEN_INDUSTRY
# в отдельную категорию
data.at[data['GEN_INDUSTRY'] == 'Юридические услуги/нотариальные услуги',
        'GEN_INDUSTRY'] = 'Другие сферы'
data.at[data['GEN_INDUSTRY'] == 'Страхование', 'GEN_INDUSTRY'] = 'Другие сферы'
data.at[data['GEN_INDUSTRY'] == 'Туризм', 'GEN_INDUSTRY'] = 'Другие сферы'
data.at[data['GEN_INDUSTRY'] == 'Недвижимость', 'GEN_INDUSTRY'] = 'Другие сферы'
data.at[data['GEN_INDUSTRY'] == 'Управляющая компания', 'GEN_INDUSTRY'] = 'Другие сферы'
data.at[data['GEN_INDUSTRY'] == 'Логистика', 'GEN_INDUSTRY'] = 'Другие сферы'
data.at[data['GEN_INDUSTRY'] == 'Подбор персонала', 'GEN_INDUSTRY'] = 'Другие сферы'
data.at[data['GEN_INDUSTRY'] == 'Маркетинг', 'GEN_INDUSTRY'] = 'Другие сферы'
# смотрим частоты категорий переменной GEN_INDUSTRY
data['GEN_INDUSTRY'].value_counts(dropna=False)
```

Out[11]:
```
Торговля                                    2303
Не указано                                  2048
Другие сферы                                1776
Металлургия/Промышленность/Машиностроение   1285
Государственная служба                      1223
Здравоохранение                             1087
Образование                                  923
Транспорт                                    763
Сельское хозяйство                           670
Строительство                                556
Коммунальное хоз-во/Дорожные службы          495
Ресторанный бизнес/Общественное питание      393
Наука                                        361
Нефтегазовая промышленность                  220
Банк/Финансы                                 168
Сборочные производства                       167
Энергетика                                   139
Развлечения/Искусство                        129
ЧОП/Детективная д-ть                         128
Информационные услуги                        102
Салоны красоты и здоровья                     96
Информационные технологии                     82
Химия/Парфюмерия/Фармацевтика                 60
СМИ/Реклама/PR-агенства                       49
Name: GEN_INDUSTRY, dtype: int64
```

In[12]:
```
# укрупняем категории переменной GEN_TITLE
data.at[data['GEN_TITLE'] == 'Партнер', 'GEN_TITLE'] = 'Другое'
data.at[data['GEN_TITLE'] == 'Военнослужащий по контракту', 'GEN_TITLE'] = 'Другое'

# смотрим частоты категорий переменной GEN_TITLE
data['GEN_TITLE'].value_counts(dropna=False)
```

Out[12]:
```
Специалист                      6680
Рабочий                         2898
Не указано                      2048
Служащий                         858
Руководитель среднего звена      657
Работник сферы услуг             529
Высококвалифиц. специалист       529
Руководитель высшего звена       409
Другое                           272
Индивидуальный предприниматель   210
Руководитель низшего звена       133
Name: GEN_TITLE, dtype: int64
```

In[13]:
```
# укрупняем категории переменной ORG_TP_STATE
data.at[data['ORG_TP_STATE'] == 'Частная ком. с инос. капиталом',
        'ORG_TP_STATE'] = 'Частная компания'

# смотрим частоты категорий переменной ORG_TP_STATE
data['ORG_TP_STATE'].value_counts(dropna=False)
```

Out[13]:
```
Частная компания                6288
Государственная комп./учреж.    5726
Не указано                      2048
Индивидуальный предприниматель   930
Некоммерческая организация       231
Name: ORG_TP_STATE, dtype: int64
```

In[14]:
```
# укрупняем категории переменной JOB_DIR
data.at[data['JOB_DIR'] == 'Реклама и маркетинг', 'JOB_DIR'] = 'Другое'
data.at[data['JOB_DIR'] == 'Кадровая служба и секретариат', 'JOB_DIR'] = 'Другое'
data.at[data['JOB_DIR'] == 'Пр-техн. обесп. и телеком.', 'JOB_DIR'] = 'Другое'
data.at[data['JOB_DIR'] == 'Юридическая служба', 'JOB_DIR'] = 'Другое'

# смотрим частоты категорий переменной JOB_DIR
data['JOB_DIR'].value_counts(dropna=False)
```

Out[14]:
```
Участие в основ. деятельности    11452
Не указано                        1367
Вспомогательный техперсонал       1025
Бухгалтерия, финансы, планир.      481
Адм-хоз. и трансп. службы          279
Другое                             238
Снабжение и сбыт                   217
Служба безопасности                164
Name: JOB_DIR, dtype: int64
```

I.5. Конструирование новых признаков, не использующее результаты математических вычислений (которое можно выполнять до/после разбиения на обучение/контроль)

Теперь создадим новые переменные, при конструировании которых не нужны те или иные математические вычисления, и поэтому их можно создавать до или после разбиения на обучающую и контрольную выборки. Мы создадим *FACT_TP_FL*, которая принимает значение 1, если область фактического пребывания клиента и область торговой точки, где клиент брал последний кредит, совпадают, или 0 в противном случае. Кроме того, мы создадим переменную *AUTO_FOR_FL*, которая принимает значение 1, если у клиента есть автомобиль импортного производства, или 0 в противном случае. Еще мы создадим парное взаимодействие переменных *GENDER* и *GAR_PRESENCE_FL* и парное взаимодействие переменных *REG_FACT_FL* и *GAR_PRESENCE_FL*.

In[15]:
```
# создаем переменную FACT_TP_FL, которая принимает значение 1, если
# область фактического пребывания клиента и область торговой точки,
# где клиент брал последний кредит, совпадают, или 0
# в противном случае
data['FACT_TP_FL'] = np.where(data['FACT_ADDRESS_PROVINCE'] == data['TP_PROVINCE'],
                              1, 0).astype('object')

# создаем переменную AUTO_FOR_FL, которая принимает значение 1, если
# у клиента есть автомобиль импортного производства, или
# 0 в противном случае
data['AUTO_FOR_FL'] = np.where((data['AUTO_RUS_FL'] == '0') & (data['OWN_AUTO'] > 0),
                               1, 0).astype('object')
```

In[16]:
```
# пишем функцию, которая создает переменную
# в результате конъюнкции переменных
# f1 и f2
def make_conj(df, f1, f2):
    df[f1 + "+" + f2] = df[f1].astype('str') + " + " + df[f2].astype('str')
```

In[17]:
```
# применяем функцию
make_conj(data, 'GENDER', 'GAR_PRESENCE_FL')
make_conj(data, 'REG_FACT_FL', 'GAR_PRESENCE_FL')
```

Теперь выведем статистики по количественным переменным.

In[18]:
```
# отключаем на всякий случай экспоненциальное представление
pd.set_option('display.float_format', lambda x: '%.3f' % x)
# выводим статистики по количественным переменным
data.describe()
```

	AGE	CHILD_TOTAL	DEPENDANTS	PERSONAL_INCOME	OWN_AUTO	CREDIT	TERM	FST_PAYMENT	FACT_LIVING_TERM	WORK_TIME
count	15223.000	15223.000	15223.000	15223.000	15223.000	15223.000	15223.000	15223.000	15223.000	13855.000
mean	40.406	1.099	0.645	13853.836	0.116	14667.959	8.101	3398.563	3039.340	292.212
std	11.601	0.995	0.812	9015.468	0.321	12147.873	4.094	5158.109	262455.437	24364.832
min	21.000	0.000	0.000	24.000	0.000	2000.000	3.000	0.000	-26.000	1.000
25%	30.000	0.000	0.000	8000.000	0.000	6500.000	6.000	1000.000	41.000	24.000
50%	39.000	1.000	0.000	12000.000	0.000	11550.000	6.000	2000.000	108.000	48.000
75%	50.000	2.000	1.000	17000.000	0.000	19170.000	10.000	4000.000	204.000	110.000
max	67.000	10.000	7.000	250000.000	2.000	119700.000	36.000	140000.000	28101997.000	2867959.000

В полученной сводке мы должны обратить внимание на три момента:

- отрицательное минимальное значение *FACT_LIVING_TERM*;
- нулевые минимальные значения переменных *CHILD_TOTAL*, *DEPENDANTS*, *OWN_AUTO*, *FST_PAYMENT*, *LOAN_NUM_CLOSED*, *LOAN_DLQ_NUM*, *LOAN_MAX_DLQ*, *LOAN_AVG_DLQ_AMT*, *LOAN_MAX_DLQ_AMT*, *PREVIOUS_CARD_NUM_UTILIZED*, при конструировании новых признаков на базе таких переменных нужно быть особо внимательным, т.к. при делении на ноль могут появятся бесконечные значения (infinite values);
- аномально большие максимальные значения переменных *FACT_LIVING_TERM* и *WORK_TIME*, такие значения могут снизить качество регрессионной модели.

Значения переменной *FACT_LIVING_TERM* берем по модулю, чтобы избавиться от отрицательных значений.

In[19]:
```
# значения переменной FACT_LIVING_TERM берем по модулю,
# чтобы избавиться от отрицательных значений
data['FACT_LIVING_TERM'] = data['FACT_LIVING_TERM'].abs()
```

I.6. Разбиение на обучающую и контрольную выборки

Разбиваем наши данные на обучающую и контрольную выборки.

In[20]:
```
# разбиваем данные на обучающую и контрольную выборки
train = data.sample(frac=0.7, random_state=200)
test = data.drop(train.index)
```

I.7. Импутация пропусков, использующая статистики – результаты математических вычислений (ее нужно выполнять после разбиения на обучение и контроль)

Давайте выполним импутацию пропусков в переменных *FACT_LIVING_TERM* и *WORK_TIME*.

Начнем с переменной *FACT_LIVING_TERM*. В ней нет пропусков, но здесь необходимо учесть момент, что часто бывает ситуация, когда из-за ошибок ввода данных стаж проживания превышает возраст клиента. Наблюдения, в которых количество лет проживания по месту фактического пребывания превышает возраст, записываем как пропуски, а затем импутируем их медианой переменной, вычисленной на обучающей выборке.

In[21]:

```
# наблюдения, в которых количество лет проживания
# по месту фактического пребывания превышает
# возраст, записываем как пропуски
train['FACT_LIVING_TERM'] = np.where(train['FACT_LIVING_TERM'] / 12 > train['AGE'],
                                     np.NaN, train['FACT_LIVING_TERM'])
test['FACT_LIVING_TERM'] = np.where(test['FACT_LIVING_TERM'] / 12 > test['AGE'],
                                    np.NaN, test['FACT_LIVING_TERM'])

# импутируем пропуски медианой, вычисленной
# на обучающей выборке
train['FACT_LIVING_TERM'].fillna(train['FACT_LIVING_TERM'].median(),
                                 inplace=True)
test['FACT_LIVING_TERM'].fillna(train['FACT_LIVING_TERM'].median(),
                                inplace=True)
```

Пропуски в переменной *WORK_TIME* обусловлены скорее всего тем, что они соответствуют клиентам-пенсионерам. Кроме того, из-за ошибок ввода данных часто встречаются два типа наблюдений:

- наблюдения, в которых время работы в годах превышает возраст (например, человек работает 40 лет, а живет всего 25);
- наблюдения, в которых разница между возрастом и временем работы в годах меньше 16 (например, у 30-летнего время работы в годах составляет 20 лет, получается, он работает с 10 лет).

Такие наблюдения мы тоже можем пометить как пропуски и импутировать их медианой переменной, вычисленной на обучающей выборке.

```
In[22]:
# наблюдения, в которых время работы в годах превышает
# возраст (например, человек работает 40 лет, а живет
# всего 25), записываем как пропуски
train['WORK_TIME'] = np.where(train['WORK_TIME'] / 12 > train['AGE'],
                              np.NaN, train['WORK_TIME'])
test['WORK_TIME'] = np.where(test['WORK_TIME'] / 12 > test['AGE'],
                             np.NaN, test['WORK_TIME'])

# наблюдения, в которых разница между возрастом и временем работы в годах
# меньше 16 (например, у 30-летнего время работы в годах составляет 20 лет,
# получается, он работает с 10 лет), записываем как пропуски
train['WORK_TIME'] = np.where((train['AGE'] - train['WORK_TIME'] / 12) < 16,
                              np.NaN, train['WORK_TIME'])
test['WORK_TIME'] = np.where((test['AGE'] - test['WORK_TIME'] / 12) < 16,
                             np.NaN, test['WORK_TIME'])

# импутируем пропуски медианой, вычисленной
# на обучающей выборке
train['WORK_TIME'].fillna(train['WORK_TIME'].median(), inplace=True)
test['WORK_TIME'].fillna(train['WORK_TIME'].median(), inplace=True)
```

Теперь убедимся, что наши переменные не содержат пропусков.

```
In[23]:
# выводим информацию об общем количестве пропущенных
# наблюдений в обучающей и контрольной выборках
print(train.isnull().sum().sum())
print(test.isnull().sum().sum())

Out[23]:
0
0
```

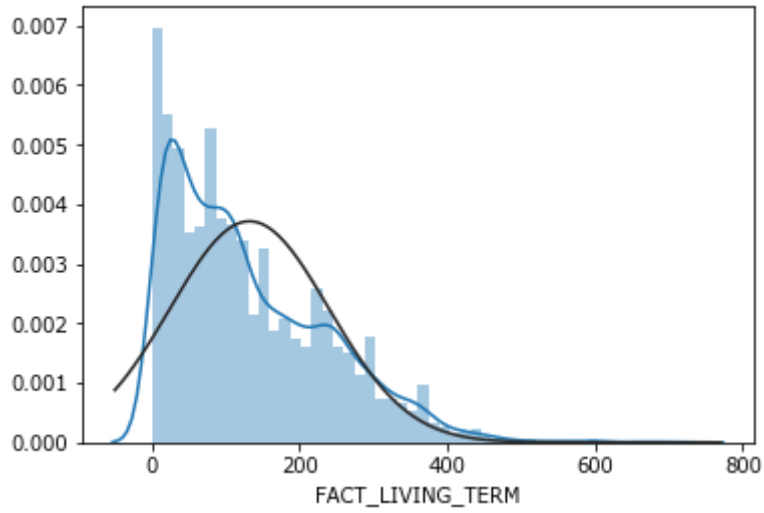
Мы убеждаемся, что наши данные не содержат пропусков и двигаемся дальше.

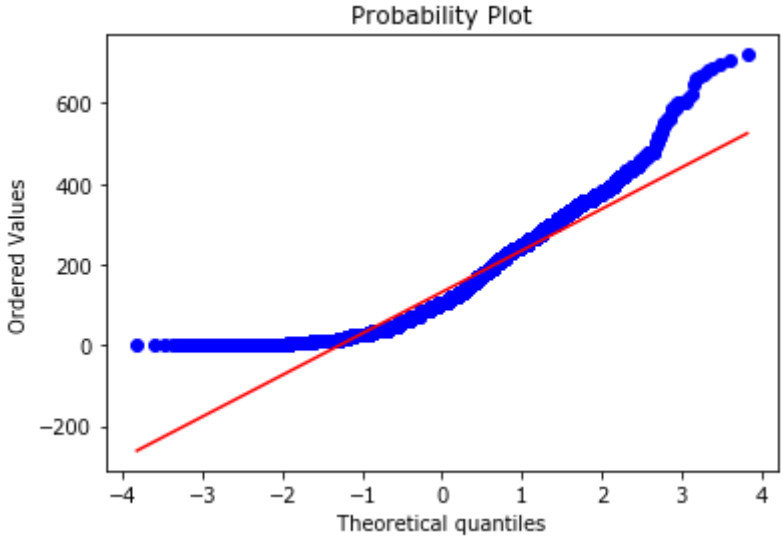
I.8. Поиск преобразований переменных, максимизирующих нормальность распределения (дается в сокращенном виде)

Давайте выполним поиск преобразований переменных, максимизирующих нормальность распределения. Начнем с переменной *FACT_LIVING_TERM*. Построим для нее гистограмму распределения и график квантиль-квантиль, а также дополнительно вычислим коэффициент скоса и коэффициент эксцесса.

```
In[24]:
# импортируем библиотеку seaborn, предварительно
# установив ее в Anaconda Prompt с помощью команды
# conda install -c anaconda seaborn
import seaborn as sns

# импортируем norm и stats
from scipy.stats import norm
from scipy import stats

# строим гистограмму распределения и график
# квантиль-квантиль для переменной FACT_LIVING_TERM
sns.distplot(train['FACT_LIVING_TERM'], fit=norm)
fig = plt.figure()
res = stats.probplot(train['FACT_LIVING_TERM'], plot=plt)
print("Скос", train['FACT_LIVING_TERM'].skew())
print("Эксцесс", train['FACT_LIVING_TERM'].kurtosis())
```

Out[24]:
```
Скос 1.0009463793229856
Эксцесс 0.8475822970599065
```

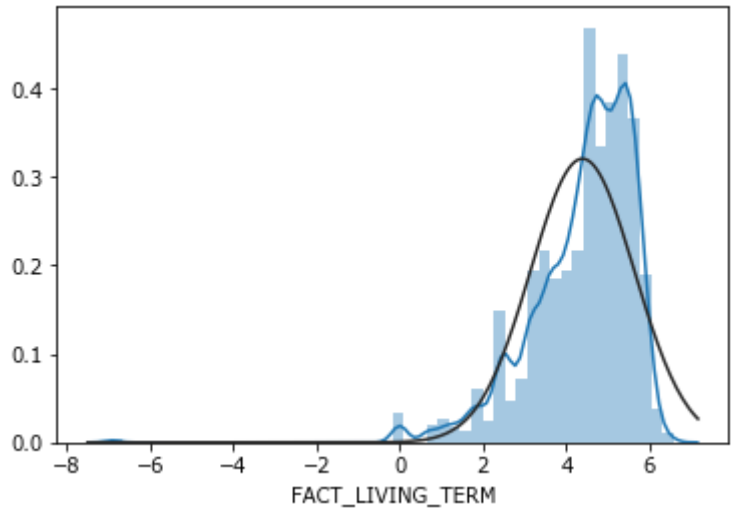


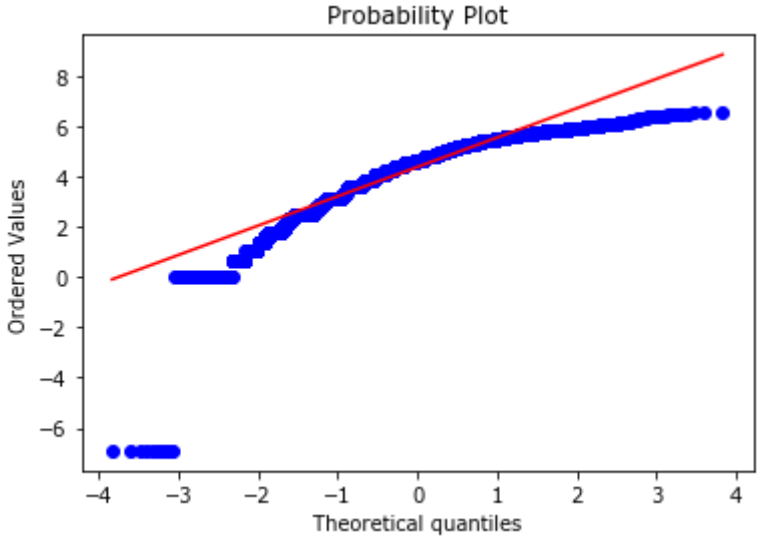


Здесь мы видим правостороннюю асимметрию, попробуем применить логарифмическое преобразование.

In[25]:
```
# строим гистограмму распределения и график
# квантиль-квантиль, применив логарифмическое
# преобразование для переменной FACT_LIVING_TERM,
# используем константу a, чтобы не брать
# логарифм нуля
a = 0.001
sns.distplot(np.log(train['FACT_LIVING_TERM'] + a), fit=norm)
fig = plt.figure()
res = stats.probplot(np.log(train['FACT_LIVING_TERM'] + a), plot=plt)
print("Скос", var.skew())
print("Эксцесс", var.kurtosis())
```

Out[25]:
```
Скос -1.7877641745035981
Эксцесс 8.158332933708543
```





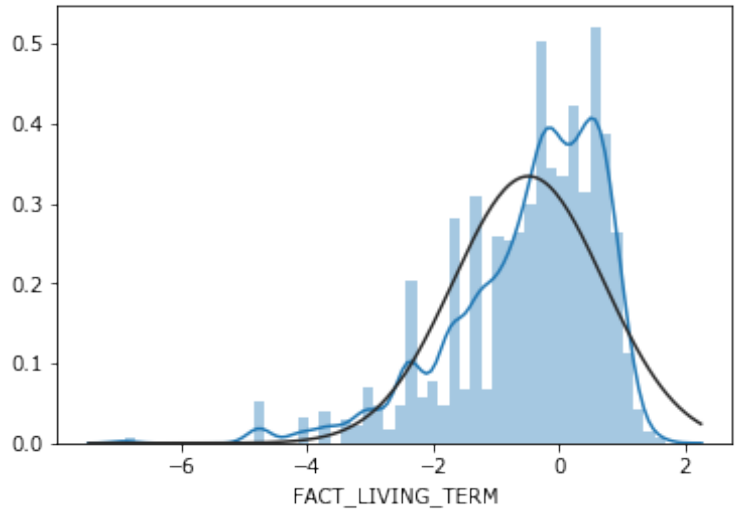
Логарифмическое преобразование по классической схеме не дало улучшение, попробуем воспользоваться формулой log(x/mean(x)+k), где k – маленькое значение-константа (меньше 1). Вспомним, что в этом преобразовании k будет работать как фактор, определяющий форму распределения (маленькие значения k делают данные более скошенными влево, а большие значения k делают данные менее скошенными). Сначала возьмем значение, близкое к 0.

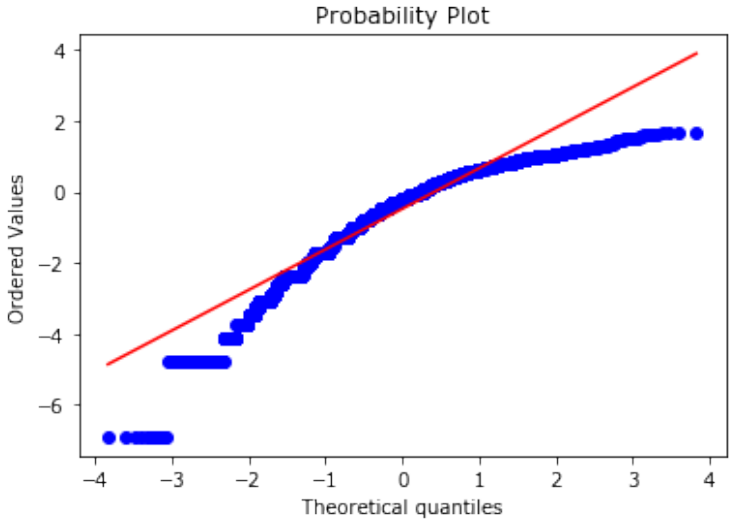
In[26]:
```
# строим гистограмму распределения и график
# квантиль-квантиль, применив логарифмическое
# преобразование по формуле log(x/mean(x)+k)
# для переменной FACT_LIVING_TERM,
# где k - небольшое значение, близкое к 0,
# чтобы сильнее смещать распределение влево
k = 0.001
sns.distplot(
    np.log((train['FACT_LIVING_TERM'] / train['FACT_LIVING_TERM'].mean()) + k),
    fit=norm)
fig = plt.figure()
res = stats.probplot(
    np.log((train['FACT_LIVING_TERM'] / train['FACT_LIVING_TERM'].mean()) + k),
```

```
    plot=plt)
var = np.log(
    (train['FACT_LIVING_TERM'] / train['FACT_LIVING_TERM'].mean()) + k)
print("Скос", var.skew())
print("Эксцесс", var.kurtosis())
```

Out[26]:

```
Скос -1.198753602609067
Эксцесс 1.8060635345088007
```





Видим, что коэффициент скоса уменьшился с 1 (исходное значение коэффициента скоса) до -1,2. Теперь возьмем значение, близкое к 1.

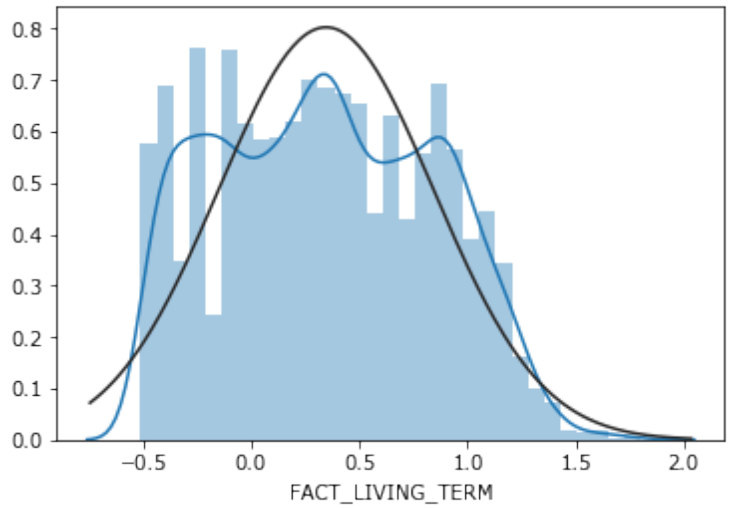
In[27]:

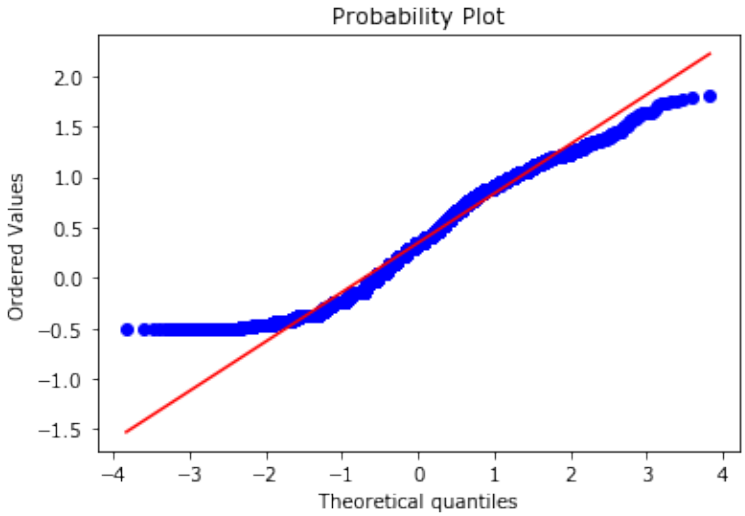
```
# строим гистограмму распределения и график
# квантиль-квантиль, применив логарифмическое
# преобразование по формуле log(x/mean(x)+k)
# для переменной FACT_LIVING_TERM,
# где k - небольшое значение, близкое к 1,
# чтобы слабее смещать распределение влево
k = 0.6
sns.distplot(
    np.log((train['FACT_LIVING_TERM'] / train['FACT_LIVING_TERM'].mean()) + k),
    fit=norm)
fig = plt.figure()
res = stats.probplot(
    np.log((train['FACT_LIVING_TERM'] / train['FACT_LIVING_TERM'].mean()) + k),
    plot=plt)
var = np.log(
    (train['FACT_LIVING_TERM'] / train['FACT_LIVING_TERM'].mean()) + k)
```

```
print("Скос", var.skew())
print("Эксцесс", var.kurtosis())
```

Out[27]:

```
Скос 0.11907859496052849
Эксцесс -0.971513963071875
```





На этот раз коэффициент скоса уменьшился не так значительно – с 1 (исходное значение коэффициента скоса) до 0,12. Таким образом, варьируя k, мы можем более тонко настроить логистическое преобразование.

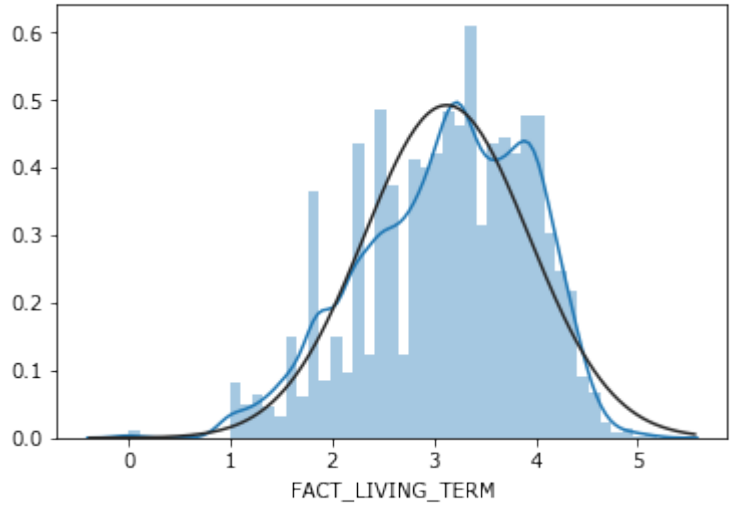
Теперь попробуем корень четвертой степени.

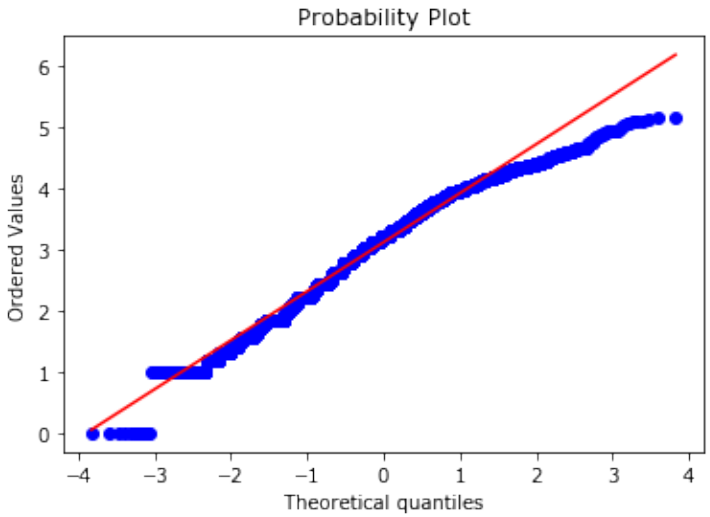
In[28]:

```
# строим гистограмму распределения и график
# квантиль-квантиль, применив преобразование
# корнем четвертой степени для переменной FACT_LIVING_TERM,
# используем модуль, чтобы не вычислять корни
# отрицательных чисел, и затем учитываем знак числа
sns.distplot(np.sign(train['FACT_LIVING_TERM']) * (
    train['FACT_LIVING_TERM'].abs() ** (1/4)), fit=norm)
fig = plt.figure()
res = stats.probplot(np.sign(train['FACT_LIVING_TERM']) * (
    train['FACT_LIVING_TERM'].abs() ** (1/4)), plot=plt)
var = np.sign(train['FACT_LIVING_TERM']) * (
    train['FACT_LIVING_TERM'].abs() ** (1/4))
print("Скос", var.skew())
print("Эксцесс", var.kurtosis())
```

Out[28]:

```
Скос -0.4317392497639781
Эксцесс -0.3100517857049643
```





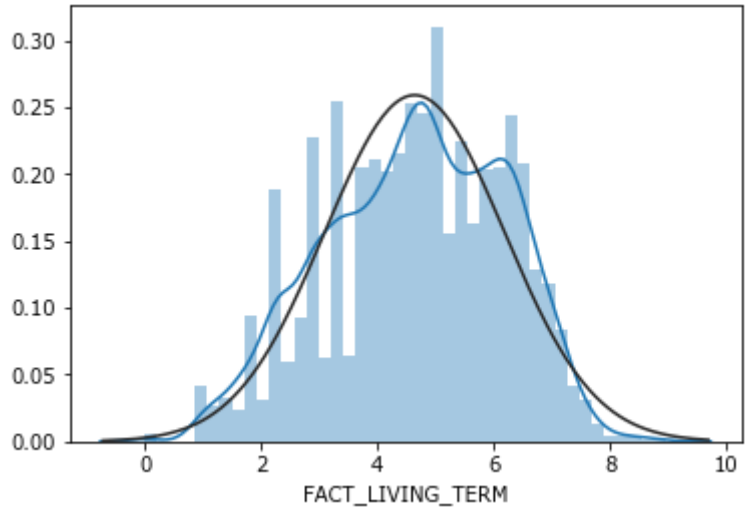
Попробуем кубический корень.

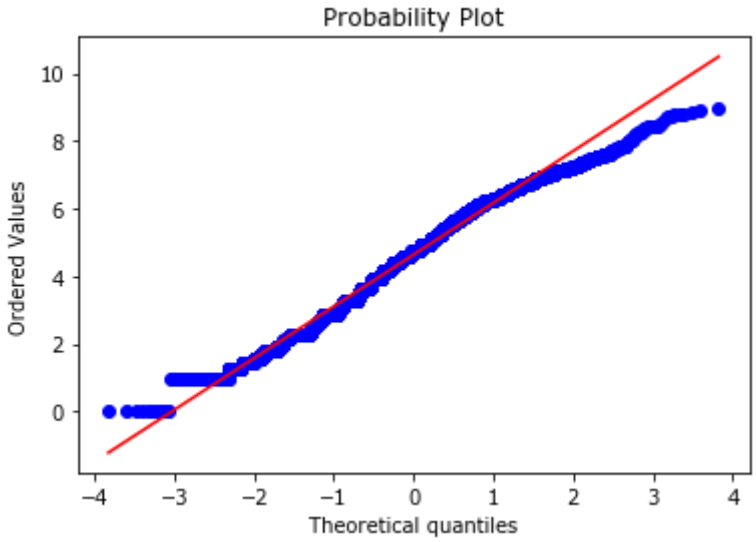
In[29]:

```
# строим гистограмму распределения и график
# квантиль-квантиль, применив преобразование
# кубическим корнем для переменной FACT_LIVING_TERM,
# используем модуль, чтобы не вычислять корни
# отрицательных чисел, и затем учитываем знак числа
sns.distplot(np.sign(train['FACT_LIVING_TERM']) * (
    train['FACT_LIVING_TERM'].abs() ** (1/3)), fit=norm)
fig = plt.figure()
res = stats.probplot(np.sign(train['FACT_LIVING_TERM']) * (
    train['FACT_LIVING_TERM'].abs() ** (1/3)), plot=plt)
var = np.sign(train['FACT_LIVING_TERM']) * (
    train['FACT_LIVING_TERM'].abs() ** (1/3))
print("Скос", var.skew())
print("Эксцесс", var.kurtosis())
```

Out[29]:

```
Скос -0.21552870628926826
Эксцесс -0.6169680965500373
```



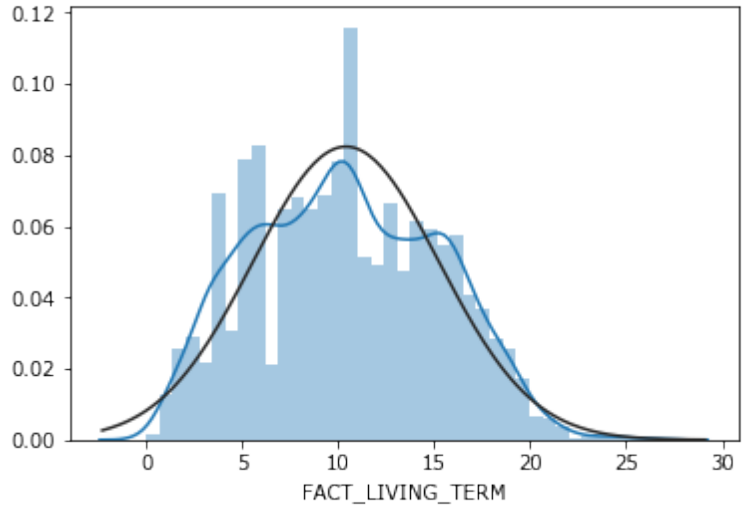


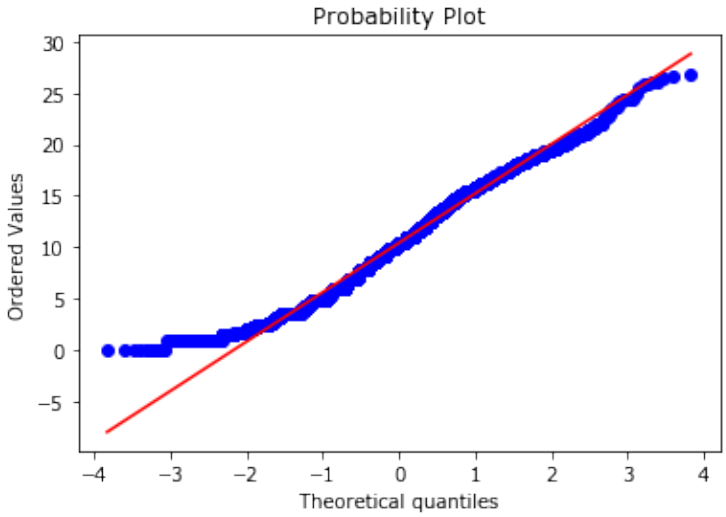
Кубический корень сработал лучше логарифма, теперь попробуем квадратный корень.

In[30]:

```
# строим гистограмму распределения и график
# квантиль-квантиль, применив преобразование
# квадратным корнем для переменной FACT_LIVING_TERM,
# используем модуль, чтобы не вычислять корни
# отрицательных чисел, и затем учитываем знак числа
sns.distplot(np.sign(train['FACT_LIVING_TERM']) * (
    train['FACT_LIVING_TERM'].abs() ** (1/2)), fit=norm)
fig = plt.figure()
res = stats.probplot(np.sign(train['FACT_LIVING_TERM']) * (
    train['FACT_LIVING_TERM'].abs() ** (1/2)), plot=plt)
var = np.sign(train['FACT_LIVING_TERM']) * (
    train['FACT_LIVING_TERM'].abs() ** (1/2))
print("Скос", var.skew())
print("Эксцесс", var.kurtosis())
```

Out[30]:
```
Скос 0.13511106646144974
Эксцесс -0.7168466001317162
```





Квадратный корень сработал чуть хуже, чем кубический корень. Теперь попробуем преобразование Бокса-Кокса.

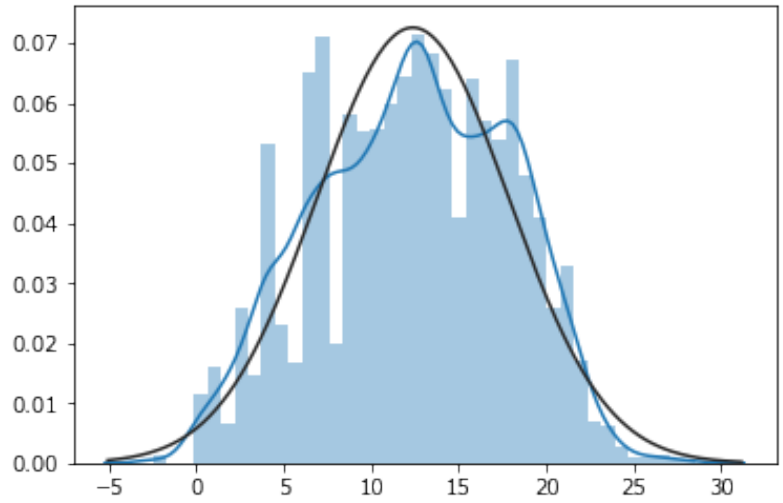
In[31]:
```
# импортируем функцию boxcox
from scipy.stats import boxcox
# выполняем преобразование Бокса-Кокса
box_transformed, lam = boxcox(train['FACT_LIVING_TERM'] + a)
print("Lambda: %f" % lam)
```

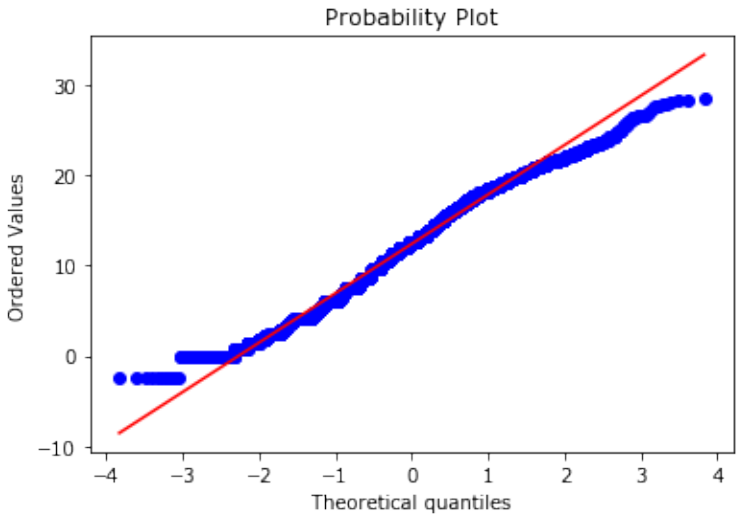
Out[31]:
```
Lambda: 0.373383
```

In[32]:
```
# строим гистограмму распределения и график
# квантиль-квантиль для переменной monthly_income,
# преобразованной с помощью Бокса-Кокса
sns.distplot(box_transformed, fit=norm)
fig = plt.figure()
res = stats.probplot(box_transformed, plot=plt)
box_transformed = pd.Series(box_transformed)
print("Скос", box_transformed.skew())
print("Эксцесс", box_transformed.kurtosis())
```

Out[32]:

```
Скос -0.12353447096945443
Эксцесс -0.6873279108128143
```





Полученное значение `lambda` близко к `lambda` = 0,5, которое соответствует преобразованию квадратного корня.
По-видимому, оптимальным преобразованием для переменной *FACT_LIVING_TERM* будет квадратный или кубический корень.
Подбираем соответствующие преобразования для остальных переменных.

I.9. Биннинг как один способов конструирования новых признаков, использующий результаты математических вычислений (нужно выполнять только после разбиения на обучение и контроль)

Сейчас на основе переменной *PERSONAL_INCOME* мы создадим временную переменную *PERSONAL_INCOME_CAT*, у которой метки категорий будут повторять метки категорий *FAMILY_INCOME*.

In[33]:

```
# на основе переменной PERSONAL_INCOME создаем временную переменную
# PERSONAL_INCOME_CAT, у которой метки категорий будут
# повторять метки категорий FAMILY_INCOME
bins = [-np.inf, 5000, 10000, 20000, 50000, np.inf]
lab = ['до 5000 руб.', 'от 5000 до 10000 руб.', 'от 10000 до 20000 руб.',
       'от 20000 до 50000 руб.', 'свыше 50000 руб.']
train['PERSONAL_INCOME_CAT'] = pd.cut(train['PERSONAL_INCOME'], bins, labels=lab)
test['PERSONAL_INCOME_CAT'] = pd.cut(test['PERSONAL_INCOME'], bins, labels=lab)
```

In[34]:

```
# выводим метки переменной PERSONAL_INCOME_CAT
train['PERSONAL_INCOME_CAT'].unique()
```

Out[34]:

```
[от 10000 до 20000 руб., от 20000 до 50000 руб., от 5000 до 10000 руб., до 5000 руб., свыше
50000 руб.]
Categories (5, object): [до 5000 руб. < от 5000 до 10000 руб. < от 10000 до 20000 руб. < от
20000 до 50000 руб. < свыше 50000 руб.]
```

Теперь мы создадим переменную *PERSONAL_FAMILY_INCOME*, которая принимает значение 1, если категория переменной *PERSONAL_INCOME_CAT* совпадает с категорией переменной *FAMILY_INCOME*, или 0 в противном случае. Затем преобразовываем переменную *PERSONAL_FAMILY_INCOME* в категориальную, а переменную *PERSONAL_INCOME_CAT* удаляем.

In[35]:

```
# создаем переменную PERSONAL_FAMILY_INCOME, которая принимает значение 1, если
# категория переменной PERSONAL_INCOME_CAT совпадает с категорией
# переменной FAMILY_INCOME, или 0 в противном случае, затем
# преобразовываем в тип object
train['PERSONAL_FAMILY_INCOME'] = np.where(train['PERSONAL_INCOME_CAT'] == train['FAMILY_INCOME'],
                                           1, 0).astype('object')
test['PERSONAL_FAMILY_INCOME'] = np.where(test['PERSONAL_INCOME_CAT'] == test['FAMILY_INCOME'],
                                          1, 0).astype('object')

# удаляем переменную PERSONAL_INCOME_CAT
train.drop('PERSONAL_INCOME_CAT', axis=1, inplace=True)
test.drop('PERSONAL_INCOME_CAT', axis=1, inplace=True)
```

Создаем новые переменные на основе биннинга переменных *CREDIT*, *FST_PAYMENT* и *AGECAT*. В данном случае мы выбрали переменные для биннинга с точки зрения бизнес-логики, как правило, сумма кредита, размер первого платежа и возраст являются одними из наиболее важных переменных, характеризующих клиента. Точки разбиения подбирались в программе Deductor.

In[36]:

```
# задаем точки, в которых будут находится границы категорий
# будущей переменной CREDITCAT
bins = [-np.inf, 7292, 9427, 14169, 27449, np.inf]
# осуществляем биннинг переменной CREDIT и записываем
# результаты в новую переменную CREDITCAT
train['CREDITCAT'] = pd.cut(train['CREDIT'], bins).astype('object')
test['CREDITCAT'] = pd.cut(test['CREDIT'], bins).astype('object')
```

In[37]:

```
# задаем точки, в которых будут находится границы категорий
# будущей переменной FSTPAYMENTCAT
bins = [-np.inf, 1500, 4995, np.inf]
# осуществляем биннинг переменной FST_PAYMENT и записываем
# результаты в новую переменную FSTPAYMENTCAT
train['FSTPAYMENTCAT'] = pd.cut(train['FST_PAYMENT'], bins).astype('object')
test['FSTPAYMENTCAT'] = pd.cut(test['FST_PAYMENT'], bins).astype('object')
```

In[38]:
```
# задаем точки, в которых будут находится границы категорий
# будущей переменной AGECAT
bins = [-np.inf, 29, 43, 52, 57, np.inf]
# осуществляем биннинг переменной AGE и записываем
# результаты в новую переменную AGECAT
train['AGECAT'] = pd.cut(train['AGE'], bins).astype('object')
test['AGECAT'] = pd.cut(test['AGE'], bins).astype('object')
```

Теперь давайте вычислим IV по всем количественным переменным, предварительно категоризированным на 10 квантилей. Это даст дополнительную информацию о том, на основе каких переменных можно еще создать новые переменные с помощью биннинга.

In[39]:
```
# пишем функцию, вычисляющую IV по всем
# количественным предикторам
def numeric_IV(df):
    iv_list = []
    a = 0.0001
    numerical_columns = [c for c in df.columns if df[c].dtype.name != 'object']
    for var_name in numerical_columns:
        df[var_name] = pd.qcut(df[var_name].values, 10, duplicates='drop').codes
        biv = pd.crosstab(df[var_name], df['TARGET'])
        IV = sum(((1.0 * biv[0] / sum(biv[0]) + a) - (1.0 * biv[1] / sum(biv[1]) + a)) *
                 np.log((1.0 * biv[0] / sum(biv[0]) + a) / (1.0 * biv[1] / sum(biv[1]) + a)))
        iv_list.append(IV)
    col_list = list(numerical_columns)
    results = pd.DataFrame({'Название переменной': col_list, 'IV': iv_list})
    results['Полезность'] = ['Подозрительно высокая' if x > 0.5 else 'Сильная'
                             if x <= 0.5 and x > 0.3 else 'Средняя'
                             if x <= 0.3 and x > 0.1 else 'Слабая'
                             if x <= 0.1 and x > 0.02 else 'Бесполезная'
                             for x in results['IV']]  # по Наиму Сиддики
    return(results.sort_values(by='IV', ascending=False))
```

In[40]:
```
# применяем нашу функцию к обучающему набору
train_copy = train.copy()
numeric_IV(train_copy)
```

Out[40]:

	Название переменной	IV	Полезность
0	AGE	0.136	Средняя
9	WORK_TIME	0.112	Средняя
3	PERSONAL_INCOME	0.096	Слабая
8	FACT_LIVING_TERM	0.060	Слабая
6	TERM	0.052	Слабая
12	LOAN_NUM_PAYM	0.046	Слабая
15	LOAN_AVG_DLQ_AMT	0.042	Слабая
16	LOAN_MAX_DLQ_AMT	0.041	Слабая
7	FST_PAYMENT	0.029	Слабая
5	CREDIT	0.026	Слабая
2	DEPENDANTS	0.025	Слабая
11	LOAN_NUM_CLOSED	0.021	Слабая
13	LOAN_DLQ_NUM	0.011	Бесполезная
10	LOAN_NUM_TOTAL	0.004	Бесполезная
1	CHILD_TOTAL	0.004	Бесполезная
14	LOAN_MAX_DLQ	0.001	Бесполезная
4	OWN_AUTO	0.000	Бесполезная
17	PREVIOUS_CARD_NUM_UTILIZED	0.000	Бесполезная

Теперь выполним укрупнение категорий *GEN_TITLE*, а на основе переменных *REGION_NM* и *ORG_TP_STATE* создадим новые переменные *REGIONCAT* и *ORGCAT* с укрупненными категориями. Для укрупнения использовался метод CHAID.

In[41]:

```
# пишем функцию, которая создает
# из списка списков словарь
def list_to_dict(input_list):
    output_dict = {}
    for n, sample_list in enumerate(input_list):
        for value in sample_list:
            output_dict[value] = n
    return output_dict

# создаем список списков для
# переменной GEN_TITLE
map_data_list = [
    ['Рабочий',
     'Служащий',
     'Работник сферы услуг',
     'Другое',
     'Индивидуальный предприниматель',
     'Руководитель низшего звена'],
    ['Специалист',
     'Руководитель среднего звена'],
    ['Не указано'],
    ['Высококвалифиц. специалист', 'Руководитель высшего звена']
]

# создаем из списка списков словарь
map_data_dict = list_to_dict(map_data_list)

# укрупняем категории переменной GEN_TITLE
train['GEN_TITLE'] = train['GEN_TITLE'].map(map_data_dict).astype('object')
test['GEN_TITLE'] = test['GEN_TITLE'].map(map_data_dict).astype('object')

# смотрим результат укрупнения на обучающей выборке
train['GEN_TITLE'].value_counts(dropna=False)
```

```
Out[41]:
1    5050
0    3459
2    1486
3     661
Name: GEN_TITLE, dtype: int64

In[42]:
# создаем список списков для
# переменной REGION_NM
map_data_list2 = [
    ['ЮЖНЫЙ', 'ЦЕНТРАЛЬНЫЙ 1', 'ЦЕНТРАЛЬНЫЙ 2'],
    ['ЗАПАДНО-СИБИРСКИЙ', 'ВОСТОЧНО-СИБИРСКИЙ', 'УРАЛЬСКИЙ',
     'ДАЛЬНЕВОСТОЧНЫЙ', 'СЕВЕРО-ЗАПАДНЫЙ', 'ЦЕНТРАЛЬНЫЙ ОФИС'],
    ['ПРИВОЛЖСКИЙ']
]

# создаем из списка списков словарь
map_data_dict2 = list_to_dict(map_data_list2)

# на основе укрупнения категорий переменной REGION_NM
# создадим переменную REGIONCAT
train['REGIONCAT'] = train['REGION_NM'].map(map_data_dict2).astype('object')
test['REGIONCAT'] = test['REGION_NM'].map(map_data_dict2).astype('object')

# смотрим новую переменную в обучающей выборке
train['REGIONCAT'].value_counts(dropna=False)

Out[42]:
1    4735
0    4024
2    1897
Name: REGIONCAT, dtype: int64

In[43]:
# создаем список списков для
# переменной ORG_TP_STATE
map_data_list3 = [
    ['Частная компания', 'Индивидуальный предприниматель', 'Некоммерческая организация'],
    ['Не указано'],
    ['Государственная комп./учреж.']
]

# создаем из списка списков словарь
map_data_dict3 = list_to_dict(map_data_list3)

# на основе укрупнения категорий переменной ORG_TP_STATE
# создаем переменную ORGCAT
train['ORGCAT'] = train['ORG_TP_STATE'].map(map_data_dict3).astype('object')
test['ORGCAT'] = test['ORG_TP_STATE'].map(map_data_dict3).astype('object')

# смотрим новую переменную в обучающей выборке
train['ORGCAT'].value_counts(dropna=False)

Out[43]:
0    5169
2    4001
1    1486
Name: ORGCAT, dtype: int64
```

I.10. Выполнение преобразований, исходя из информации гистограмм распределения и графиков квантиль-квантиль

Теперь применяем оптимальные преобразования переменных, которые мы нашли в ходе построения гистограмм распределения и графиков квантиль-квантиль.

```
In[44]:
# выполняем логарифмическое преобразование
# переменной PERSONAL_INCOME
train['PERSONAL_INCOME'] = np.log(train['PERSONAL_INCOME'] + a)
test['PERSONAL_INCOME'] = np.log(test['PERSONAL_INCOME'] + a)

# выполняем логарифмическое преобразование
# переменной CREDIT
train['CREDIT'] = np.log(train['CREDIT'] + a)
test['CREDIT'] = np.log(test['CREDIT'] + a)

# выполняем логарифмическое преобразование
# переменной WORK_TIME
train['WORK_TIME'] = np.log(train['WORK_TIME'] + a)
test['WORK_TIME'] = np.log(test['WORK_TIME'] + a)

# выполняем преобразование переменной FACT_LIVING_TERM
# кубическим корнем
train['FACT_LIVING_TERM'] = np.sign(train['FACT_LIVING_TERM']) * (
    train['FACT_LIVING_TERM'].abs() ** (1/3))
test['FACT_LIVING_TERM'] = np.sign(test['FACT_LIVING_TERM']) * (
    test['FACT_LIVING_TERM'].abs() ** (1/3))

# выполняем логарифмическое преобразование
# переменной LOAN_AVG_DLQ_AMT
train['LOAN_AVG_DLQ_AMT'] = np.log(train['LOAN_AVG_DLQ_AMT'] + a)
test['LOAN_AVG_DLQ_AMT'] = np.log(test['LOAN_AVG_DLQ_AMT'] + a)
```

1.11. Конструирование новых признаков

Вновь выполним конструирование новых признаков. Мы создадим:

- переменную *CHILD_DEP* – отношение количества детей (*CHILD_TOTAL*) к общему количеству детей и иждивенцев (*CHILD_TOTAL* и *DEPENDANTS*);
- переменную *PAYMENT* – сумму ежемесячного взноса по кредиту, разделив сумму кредита (*CREDIT*) на срок кредита (*TERM*);
- переменную *PTI* – коэффициент долговой нагрузки, разделив сумму ежемесячного взноса по кредиту (*PAYMENT*) на личный доход (*PERSONAL_INCOME*);
- переменную *CLOSED_TO_TOTAL* – коэффициент погашения ссуд, разделив количество погашенных ссуд (*LOAN_NUM_CLOSED*) на общее количество ссуд (*LOAN_NUM_TOTAL*);
- переменную *PAYM_TO_LOAN* – отношение количества платежей (*LOAN_NUM_PAYM*) к общему количеству ссуд (*LOAN_NUM_TOTAL*);
- переменную *DLQ_TO_PAYM* – отношение количества просрочек (*LOAN_DLQ_NUM*) к общему количеству платежей (*LOAN_NUM_PAYM*);
- переменную *FST_SHARE* – отношение суммы первого платежа по кредиту (*FST_PAYMENT*) к объединенной сумме первого платежа (*FST_PAYMENT*) и кредита (*CREDIT*);
- переменную *DLQ_TIME* – индекс времени наступления максимальной просрочки, разделив номер максимальной

просрочки ($LOAN\_MAX\_DLQ$) на количество просрочек ($LOAN\_DLQ\_NUM$).

In[45]:

```
# создаем переменную CHILD_DEP - отношение количества детей (CHILD_TOTAL)
# к общему количеству детей и иждивенцев (CHILD_TOTAL и DEPENDANTS)
train['CHILD_DEP'] = train['CHILD_TOTAL'] / (train['CHILD_TOTAL'] + train['DEPENDANTS'])
test['CHILD_DEP'] = test['CHILD_TOTAL'] / (test['CHILD_TOTAL'] + test['DEPENDANTS'])

# если переменная содержит пропуск, то возвращается значение 0,
# если пропуска нет, то возвращается исходное значение переменной
train['CHILD_DEP'] = np.where(train['CHILD_DEP'].isnull(), 0, train['CHILD_DEP'])
test['CHILD_DEP'] = np.where(test['CHILD_DEP'].isnull(), 0, test['CHILD_DEP'])

# если бы наша переменная содержала бесконечные значения, то можно было бы воспользоваться
# train['CHILD_DEP'] = np.where(np.isfinite(train['CHILD_DEP']), train['CHILD_DEP'], 0)
# т.е. если переменная CHILD_DEP содержит конечное значение,
# возвращаем исходное значение, если она содержит бесконечное
# значение, возвращаем значение 0

# создаем переменную PAYMENT - сумму ежемесячного взноса
# по кредиту, разделив сумму кредита (CREDIT)
# на срок кредита (TERM)
train['PAYMENT'] = np.log((train['CREDIT'] / train['TERM']) + a)
test['PAYMENT'] = np.log((test['CREDIT'] / test['TERM']) + a)

# создаем переменную PTI - коэффициент долговой нагрузки,
# разделив сумму ежемесячного взноса по кредиту (PAYMENT)
# на личный доход (PERSONAL_INCOME)
train['PTI'] = train['PAYMENT'] / train['PERSONAL_INCOME']
test['PTI'] = test['PAYMENT'] / test['PERSONAL_INCOME']
# создаем переменную CLOSED_TO_TOTAL - коэффициент погашения ссуд,
# разделив количество погашенных ссуд (LOAN_NUM_CLOSED)
# на общее количество ссуд (LOAN_NUM_TOTAL)
train['CLOSED_TO_TOTAL'] = train['LOAN_NUM_CLOSED'] / train['LOAN_NUM_TOTAL']
test['CLOSED_TO_TOTAL'] = test['LOAN_NUM_CLOSED'] / test['LOAN_NUM_TOTAL']

# создаем переменную PAYM_TO_LOAN- отношение количества
# платежей (LOAN_NUM_PAYM) к общему количеству ссуд (LOAN_NUM_TOTAL)
train['PAYM_TO_LOAN'] = train['LOAN_NUM_PAYM'] / train['LOAN_NUM_TOTAL']
test['PAYM_TO_LOAN'] = test['LOAN_NUM_PAYM'] / test['LOAN_NUM_TOTAL']

# создаем переменную DLQ_TO_PAYM - отношение количества просрочек
# (LOAN_DLQ_NUM) к общему количеству платежей (LOAN_NUM_PAYM)
train['DLQ_TO_PAYM'] = train['LOAN_DLQ_NUM'] / train['LOAN_NUM_PAYM']
test['DLQ_TO_PAYM'] = test['LOAN_DLQ_NUM'] / test['LOAN_NUM_PAYM']

# создаем переменную FST_SHARE - отношение суммы первого платежа по кредиту
# к объединенной сумме первого платежа (FST_PAYMENT) и кредита (CREDIT)
train['FST_SHARE'] = train['FST_PAYMENT'] / (train['FST_PAYMENT'] + train['CREDIT'])
test['FST_SHARE'] = test['FST_PAYMENT'] / (test['FST_PAYMENT'] + test['CREDIT'])

# создаем DLQ_TIME - индекс времени наступления максимальной просрочки,
# разделив номер максимальной просрочки (LOAN_MAX_DLQ)
# на количество просрочек (LOAN_DLQ_NUM)
train['DLQ_TIME'] = train['LOAN_MAX_DLQ'] / train['LOAN_DLQ_NUM']
test['DLQ_TIME'] = test['LOAN_MAX_DLQ'] / test['LOAN_DLQ_NUM']

# если переменная содержит пропуск, то возвращается значение 0,
# если пропуска нет, то возвращается исходное значение переменной
train['DLQ_TIME'] = np.where(train['DLQ_TIME'].isnull(), 0, train['DLQ_TIME'])
test['DLQ_TIME'] = np.where(test['DLQ_TIME'].isnull(), 0, test['DLQ_TIME'])

# создаем переменную LOAN_MAX_DLQ_RANGE - разницу между максимальной суммой
# просрочки (LOAN_MAX_DLQ_AMT) и средней суммой просрочки (LOAN_AVG_DLQ_AMT)
train['LOAN_MAX_DLQ_RANGE'] = ((train['LOAN_MAX_DLQ_AMT'] - train['LOAN_AVG_DLQ_AMT']) /
                               train['LOAN_AVG_DLQ_AMT']).abs()
test['LOAN_MAX_DLQ_RANGE'] = ((test['LOAN_MAX_DLQ_AMT'] - test['LOAN_AVG_DLQ_AMT']) /
                              test['LOAN_AVG_DLQ_AMT']).abs()

# если переменная содержит пропуск, то возвращается значение 0,
```

```
# если пропуска нет, то возвращается исходное значение переменной
train['LOAN_MAX_DLQ_RANGE'] = np.where(train['LOAN_MAX_DLQ_RANGE'].isnull(), 0,
                                       train['LOAN_MAX_DLQ_RANGE'])
test['LOAN_MAX_DLQ_RANGE'] = np.where(test['LOAN_MAX_DLQ_RANGE'].isnull(), 0,
                                      test['LOAN_MAX_DLQ_RANGE'])
```

I.12. Стандартизация

Теперь выполним стандартизацию количественных переменных.

In[46]:

```
# выполняем стандартизацию количественных переменных
num_cols = [c for c in train.columns if train[c].dtype.name != 'object']
train_copy = train.copy()
for i in num_cols:
    train[i] = (train[i] - train[i].mean()) / train[i].std()
    test[i] = (test[i] - train_copy[i].mean()) / train_copy[i].std()
```

Убедимся, что наши новые переменные не содержат пропусков.

In[47]:

```
# выводим информацию об общем количестве пропущенных
# наблюдений в обучающей и контрольной выборках
print(train.isnull().sum().sum())
print(test.isnull().sum().sum())
```

Out[47]:

```
0
0
```

I.13. Дамми-кодирование

Убедившись, что все в порядке, выполняем дамми-кодирование.

In[48]:

```
# печатаем названия столбцов до и после
# дамми-кодирования
print("Исходные переменные:\n", list(train.columns), "\n")
train_dummies = pd.get_dummies(train)
print("Переменные после get_dummies:\n", list(train_dummies.columns))

print("Исходные переменные:\n", list(test.columns), "\n")
test_dummies = pd.get_dummies(test)
print("Переменные после get_dummies:\n", list(test_dummies.columns))
```

I.14. Подготовка массивов признаков и массивов меток зависимой переменной

Теперь готовим массивы признаков и массивы меток.

In[49]:

```
# создаем обучающий и контрольный массивы меток
y_train = train_dummies.loc[:, 'TARGET_1']
y_test = test_dummies.loc[:, 'TARGET_1']
# удаляем из будущих массивов признаков результаты
# дамми-кодирования зависимой переменной -
# TARGET_0 и TARGET_1
train_dummies.drop(['TARGET_0', 'TARGET_1'], axis=1, inplace=True)
test_dummies.drop(['TARGET_0', 'TARGET_1'], axis=1, inplace=True)
# создаем обучающий и контрольный массивы признаков
X_train = train_dummies.loc[:, 'AGE':'ORGCAT_2']
X_test = test_dummies.loc[:, 'AGE':'ORGCAT_2']
```

У нас все готово для моделирования!

I.15. Построение логистической регрессии с помощью класса LogisticRegression библиотеки scikit-learn

Теперь строим модели логистической регрессии, попробовав различные виды стандартизации и регуляризацию.

In[50]:
```
# импортируем класс LogisticRegression
from sklearn.linear_model import LogisticRegression
```

In[51]:
```
# импортируем функцию roc_auc_score
from sklearn.metrics import roc_auc_score
# строим логистическую регрессию
logreg = LogisticRegression().fit(X_train, y_train)
print("AUC на обучающей выборке: {:.3f}".
      format(roc_auc_score(y_train, logreg.predict_proba(X_train)[:, 1])))
print("AUC на контрольной выборке: {:.3f}".
      format(roc_auc_score(y_test, logreg.predict_proba(X_test)[:, 1])))
```

Out[51]:
```
AUC на обучающей выборке: 0.747
AUC на контрольной выборке: 0.691
```

Теперь воспользуемся L1-регуляризацией (лассо), которая устанавливают некоторые оценки коэффициентов регрессии точно в нуле и тем самым осуществляет отбор переменных.

In[52]:
```
# строим логистическую регрессию,
# используем l1-регуляризацию (лассо), которая
# устанавливают некоторые оценки коэффициентов
# регрессии точно в нуле и тем самым осуществляет
# отбор переменных
logreg = LogisticRegression(C=0.12, penalty='l1').fit(X_train, y_train)
print("AUC на обучающей выборке: {:.3f}".
      format(roc_auc_score(y_train, logreg.predict_proba(X_train)[:, 1])))
print("AUC на контрольной выборке: {:.3f}".
      format(roc_auc_score(y_test, logreg.predict_proba(X_test)[:, 1])))
```

Out[52]:
```
AUC на обучающей выборке: 0.723
AUC на контрольной выборке: 0.701
```

I.16. Настройка гиперпараметров логистической регрессии с помощью класса GridSearchCV

Для получения более надежных оценок качества при подборе различных значений гиперпараметра `C` воспользуемся комбинированной проверкой.

```
In[53]:
# импортируем класс StratifiedKFold для выполнения
# стратифицированной перекрестной проверки (т.е. в
# каждом блоке соблюдаются исходные пропорции классов)
from sklearn.model_selection import StratifiedKFold
# создаем экземпляр класса StratifiedKFold, 10-блочную
# перекрестную проверку со случайным перемешиванием данных
strat = StratifiedKFold(n_splits=10, shuffle=True,
                        random_state=42)
# импортируем класс GridSearchCV
from sklearn.model_selection import GridSearchCV
# создаем экземпляр класса LogisticRegression,
# логистическую регрессию с L1-регуляризацией
logreg_grid = LogisticRegression(penalty='l1', random_state=42)
# задаем сетку параметров, будем перебирать
# разные значения штрафа
param_grid = {'C': [0.18, 0.2, 0.22, 0.24, 0.26, 0.28, 0.3, 0.32]}

In[54]:
# создаем экземпляр класса GridSearchCV
grid_search = GridSearchCV(logreg_grid, param_grid,
                           scoring='roc_auc',
                           return_train_score=True,
                           n_jobs=-1, cv=strat)
# запускаем решетчатый поиск
grid_search.fit(X_train, y_train)
# проверяем модель со значением гиперпараметра C, дающим наибольшее
# значение AUC (усредненное по контрольным блокам перекрестной
# проверки), на тестовой выборке
test_score = roc_auc_score(y_test, grid_search.predict_proba(X_test)[:, 1])
# смотрим результаты решетчатого поиска
print("AUC на тестовой выборке: {:.3f}".format(test_score))
print("Наилучшее значение гиперпараметра C: {}".format(grid_search.best_params_))
print("Наилучшее значение AUC: {:.3f}".format(grid_search.best_score_))

Out[54]:
AUC на тестовой выборке: 0.701
Наилучшее значение гиперпараметра C: {'C': 0.2}
Наилучшее значение AUC: 0.704

In[55]:
# преобразуем результаты поиска в датафрейм
# и выведем их
results = pd.DataFrame(grid_search.cv_results_)
table = results.pivot_table(values = ['mean_test_score'],
                            index = ['param_C'])
print(table)

Out[55]:
         mean_test_score
param_C
0.180              0.704
0.200              0.704
0.220              0.704
0.240              0.704
0.260              0.704
0.280              0.703
0.300              0.703
0.320              0.703
```

I.17. Отбор признаков для логистической регрессии с помощью случайного леса (класса RFE)

Мы можем выполнить отбор признаков не только с помощью L1-регуляризации, но и с помощью случайного леса. Для этого при помощи случайного леса отберем n наиболее важных переменных и затем на редуцированном массиве признаков построим логистическую регрессию.

В этих целях воспользуемся классом RFE. Обратите внимание, чем выше будет качество вашего случайного леса, тем больше вероятность, что вы сможете отобрать наиболее полезные признаки. Поэтому перед тем, как выполнить редукцию признаков с помощью случайного леса, нужно настроить его параметры, в данном случае для экономии времени мы возьмем уже найденные оптимальные значения параметров `n_estimators` и `max_depth`.

1.17.1. Отбор признаков для логистической регрессии с помощью случайного леса (класса RFE)

Мы построим случайный лес из 200 деревьев глубиной 10 и отберем 100 наиболее важных предикторов.

In[56]:
```
# импортируем класс RandomForestClassifier
from sklearn.ensemble import RandomForestClassifier
# импортируем класс RFE
from sklearn.feature_selection import RFE
# создаем экземпляр класса RFE, отберем первые 100 переменных
# с помощью случайного леса
select = RFE(RandomForestClassifier(n_estimators=200, max_depth=10,
                                    random_state=42, n_jobs=-1),
             n_features_to_select=100)
# обучаем модель и применяем к нашим массивам
select.fit(X_train_scaled, y_train)
X_train_rfe = select.transform(X_train_scaled)
X_test_rfe = select.transform(X_test_scaled)
```

В данном случае метод `.fit()` определяет, какие признаки не входят в топ-100 и их нужно исключить, а метод `.transform()` удаляет неважные признаки (т.е. выполняет преобразование массива признаков). Давайте взглянем на форму обучающего и контрольного массивов.

In[57]:
```
# смотрим форму массива
print("Форма исходного обучающего массива: {}".format(str(X_train_rfe.shape)))
print("Форма исходного тестового массива: {}".format(str(X_test_rfe.shape)))
```

Out[57]:
```
Форма исходного обучающего массива: (10656, 100)
Форма исходного тестового массива: (4567, 100)
```

С помощью метода `.get_support()` взглянем, какие переменные были отобраны.

In[58]:
```
# взглянем, какие переменные были отобраны
mask = select.get_support()
feat_labels = X_train.columns
for c, feature in zip(mask, feat_labels):
    print(feature, c)

# можно было еще воспользоваться вот таким программным кодом:
# for i, (a, b) in enumerate(zip(feat_labels, mask)): print(i, a, b)
```

Out[58]:
```
AGE True
CHILD_TOTAL True
DEPENDANTS True
PERSONAL_INCOME True
OWN_AUTO True
```

```
CREDIT True
TERM True
FST_PAYMENT True
FACT_LIVING_TERM True
WORK_TIME True
LOAN_NUM_TOTAL True
LOAN_NUM_CLOSED True
LOAN_NUM_PAYM True
LOAN_DLQ_NUM True
LOAN_MAX_DLQ True
LOAN_AVG_DLQ_AMT True
LOAN_MAX_DLQ_AMT True
. . . . . .
```

I.17.2. Построение логистической регрессии по массиву признаков, отобранных с помощью случайного леса (класса RFE)

Теперь на массиве признаков, отобранных с помощью класса RFE, построим модель логистической регрессии.

In[59]:
```
logreg = LogisticRegression().fit(X_train_rfe, y_train)
print("AUC на обучающей выборке: {:.3f}".
      format(roc_auc_score(y_train, logreg.predict_proba(X_train_rfe)[:, 1])))
print("AUC на контрольной выборке: {:.3f}".
      format(roc_auc_score(y_test, logreg.predict_proba(X_test_rfe)[:, 1])))
```

Out[59]:
```
AUC на обучающей выборке: 0.724
AUC на контрольной выборке: 0.699
```

I.17.3. Построение логистической регрессии по массиву признаков, отобранных с помощью случайного леса (класса RFE)

А теперь попробуем настроить значения параметра C логистической регрессии с помощью решетчатого поиска, использовав массив признаков, отобранных с помощью случайного леса (класса RFE).

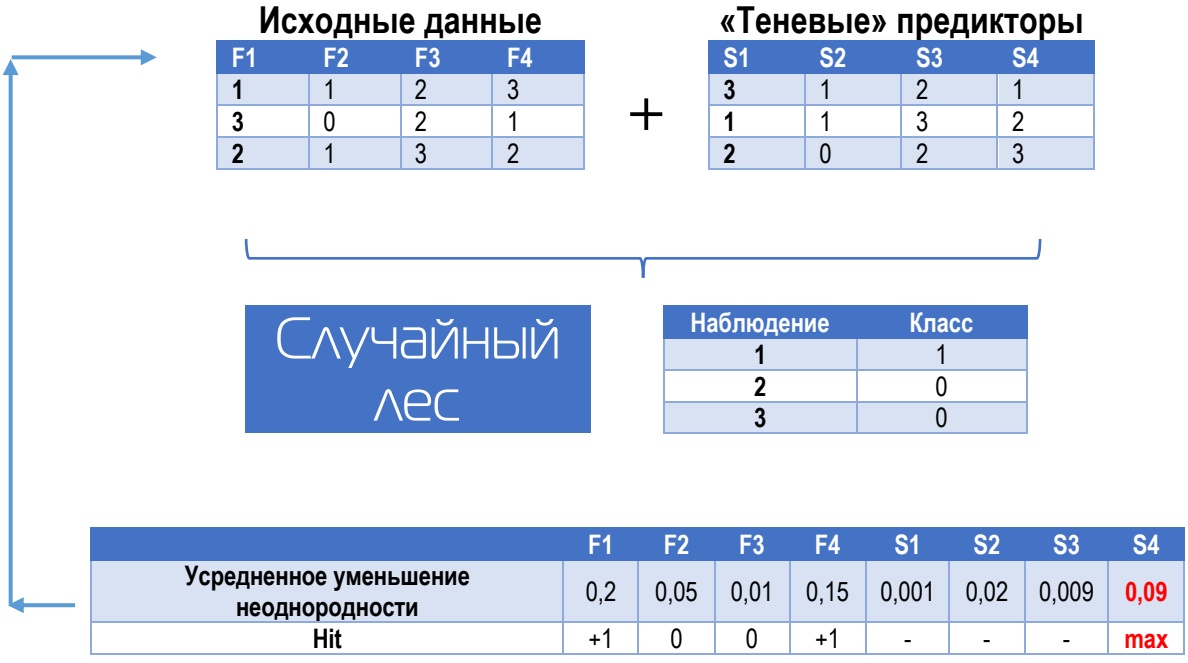
In[60]:
```
# создаем экземпляр класса LogisticRegression,
# логистическую регрессию с L1-регуляризацией
logreg_grid2 = LogisticRegression(penalty='l1', random_state=42)
# задаем сетку параметров, будем перебирать
# разные значения штрафа
param_grid2 = {'C': [0.1, 0.12, 0.14, 0.16, 0.18, 0.20]}
# создаем экземпляр класса GridSearchCV
grid_search2 = GridSearchCV(logreg_grid2, param_grid2,
                            scoring='roc_auc',
                            n_jobs=-1, cv=strat)
# запускаем решетчатый поиск
grid_search2.fit(X_train_rfe, y_train)
# проверяем модель со значением гиперпараметра C, дающим наибольшее
# значение AUC (усредненное по контрольным блокам перекрестной
# проверки), на тестовой выборке
test_score = roc_auc_score(y_test, grid_search2.predict_proba(X_test_rfe)[:, 1])
# смотрим результаты решетчатого поиска
print("AUC на тестовой выборке: {:.3f}".format(test_score))
print("Наилучшее значение гиперпараметра C: {}".format(grid_search2.best_params_))
print("Наилучшее значение AUC: {:.3f}".format(grid_search2.best_score_))
```

Out[60]:
```
AUC на тестовой выборке: 0.702
Наилучшее значение гиперпараметра C: {'C': 0.2}
Наилучшее значение AUC: 0.703
```

I.18. Отбор признаков для логистической регрессии с помощью BorutaPy

Теперь для отбора признаков воспользуемся пакетом `BorutaPy`. В основе алгоритма Boruta лежит идея оценки важности предикторов при помощи случайных перестановок их значений.

Сначала набор данных дополняется копиями всех предикторов – так называемыми «теневыми» предикторами (shadow attributes), при этом значения добавленных предикторов случайным образом перемешиваются, чтобы они перестали быть связанными с зависимой переменной. Мы обучаем модель типа случайный лес (можно использовать и другие модели, возвращающие важности признаков) и получаем значение относительной важности по каждому признаку, более высокие значения указывают на большую важность. Затем мы проверяем каждый исходный предиктор, есть ли среди них предикторы с важностью большей, чем у лучшего из «теневых» предикторов. Если такие предикторы есть, помещаем их в вектор «хитов» и запускаем новую итерацию.



F1	F2	F3	F4
1	1	2	3
3	0	2	1
2	1	3	2

+

S1	S2	S3	S4
3	1	2	1
1	1	3	2
2	0	2	3

Наблюдение	Класс
1	1
2	0
3	0

	F1	F2	F3	F4	S1	S2	S3	S4
Усредненное уменьшение неоднородности	0,2	0,05	0,01	0,15	0,001	0,02	0,009	0,09
Hit	+1	0	0	+1	-	-	-	max

После заранее определенного количества итераций мы получаем таблицу этих «хитов». В каждой итерации мы проверяем, является ли данный предиктор более важным, чем «теневой». Мы просто сравниваем количество раз, когда предиктор стал более важным, чем «теневой» предиктор, используя биномиальное распределение.

	F1	F2	F3	F4	S1	S2	S3	S4
Усредненное уменьшение неоднородности	0,2	0,05	0,01	0,15	0,001	0,02	0,009	**0,09**
Hit	+1	0	0	+1	-	-	-	**max**
Hit	+1	+1	0	0	-	**max**	-	-
Hit	+1	0	0	+1	-	-	-	**max**

Признак F1 в 3 итерациях имеет важность, превышающую важность лучшего из «теневых» признаков, все 3 раза. Итак, мы вычисляем *p*-значение, используя биномиальное распределение, $k = 3$, $n = 3$, $p = 0{,}5$. Поскольку мы делаем это для тысяч признаков, мы должны ввести поправку на множественное тестирование. Исходный метод использует для этого довольно консервативную поправку Бонферрони. Мы говорим, что признак подтверждается как важный, если его скорректированное *p*-значение меньше 0,05.

Затем мы удаляем этот столбец из исходной матрицы данных и запускаем следующую итерацию. И, наоборот, если признак не был записан в качестве «хита», например, в 15 итерациях, мы отвергаем его, а также удаляем его из исходной матрицы данных. Если достигнуто заданное количество итераций (или если все признаки были подтверждены или отвергнуты), мы останавливаемся.

I.18.1. Отбор признаков для логистической регрессии с помощью BorutaPy (на основе случайного леса)

Первым делом мы должны импортировать класс `BorutaPy`.

```
In[61]:
# устанавливаем пакет boruta в Anaconda Promt
# с помощью строки pip install Boruta
# импортируем класс BorutaPy
from boruta import BorutaPy
```

Ниже приводится список гиперпараметров для класса `BorutaPy`.

Гиперпараметр	Предназначение
`estimator`	Задает объект-модель машинного обучения (обычно ансамбль деревьев решений), с помощью которого будут вычислены важности предикторов.
`n_estimators`	Задает количество базовых моделей для ансамбля. По умолчанию задано значение 1000.
`perc`	Задает процентиль в качестве порога для сравнения теневых и исходных предикторов. По умолчанию задано значение 100.
`alpha`	Задает уровень значимости для скорректированных *p*-значений. По умолчанию задано значение 0,05.
`max_iter`	Задает максимальное количество итераций. По умолчанию задано значение 100.

```
In[62]:
# создаем экземпляр класса RandomForestClassifier,
# задаем модель леса с 50 деревьями глубиной 10
rf = RandomForestClassifier(n_estimators=50, max_depth=10,
                            random_state=42, n_jobs=-1)
rf

In[63]:
# создаем экземпляр класса BorutaPy, т.е. задаем модель BorutaPy
boruta_selector = BorutaPy(rf, n_estimators=10,
                           verbose=2, alpha=0.06,
                           perc=10, max_iter=50,
                           random_state=42)
boruta_selector
```

Обратите внимание, что модель BorutaPy принимает на вход только массивы NumPy, поэтому воспользуемся свойством `.values`.

```
In[64]:
# обучаем модель Boruta, отбирая с помощью случайного леса предикторы
boruta_selector.fit(X_train.values, y_train.values)

In[65]:
# преобразуем данные с помощью модели BorutaPy, то есть
# фильтруем предикторы согласно обученной модели BorutaPy
X_train_boruta = boruta_selector.transform(X_train.values)
X_test_boruta = boruta_selector.transform(X_test.values)
# смотрим форму массива
print("Форма исходного обучающего массива: {}".format(str(X_train_boruta.shape)))
print("Форма исходного контрольного массива: {}".format(str(X_test_boruta.shape)))

Out[65]:
Форма исходного обучающего массива: (10656, 172)
Форма исходного контрольного массива: (4567, 172)
```

I.18.2. Построение логистической регрессии по массиву признаков, отобранных с помощью BorutaPy (на основе случайного леса)

Теперь на массиве признаков, отобранных с помощью класса `BorutaPy`, построим модель логистической регрессии.

```
In[66]:
logreg = LogisticRegression().fit(X_train_boruta, y_train)
print("AUC на обучающей выборке: {:.3f}".
      format(roc_auc_score(y_train, logreg.predict_proba(X_train_boruta)[:, 1])))
print("AUC на контрольной выборке: {:.3f}".
      format(roc_auc_score(y_test, logreg.predict_proba(X_test_boruta)[:, 1])))

Out[66]:
AUC на обучающей выборке: 0.731
AUC на контрольной выборке: 0.703
```

I.18.3. Настройка параметров логистической регрессии с помощью решетчатого поиска по массиву признаков, отобранных с помощью BorutaPy (на основе случайного леса)

Попробуем настроить значения гиперпараметра `C` логистической регрессии с помощью решетчатого поиска, использовав массив признаков, отобранных с помощью класса `BorutaPy`.

In[67]:
```
# создаем экземпляр класса LogisticRegression,
# логистическую регрессию с L1-регуляризацией
logreg_grid3 = LogisticRegression(penalty='l1', random_state=42)
# задаем сетку параметров, будем перебирать
# разные значения штрафа
param_grid3 = {'C': [1.0, 0.9, 0.8, 0.7, 0.6]}
# создаем экземпляр класса GridSearchCV
grid_search3 = GridSearchCV(logreg_grid3, param_grid3,
                            scoring='roc_auc',
                            n_jobs=-1, cv=strat)
# запускаем решетчатый поиск
grid_search3.fit(X_train_boruta, y_train)
# проверяем модель со значением гиперпараметра C, дающим наибольшее
# значение AUC (усредненное по контрольным блокам перекрестной
# проверки), на тестовой выборке
test_score = roc_auc_score(y_test, grid_search3.predict_proba(X_test_boruta)[:, 1])
# смотрим результаты решетчатого поиска
print("AUC на тестовой выборке: {:.3f}".format(test_score))
print("Наилучшее значение гиперпараметра C: {}".format(grid_search3.best_params_))
print("Наилучшее значение AUC: {:.3f}".format(grid_search3.best_score_))
```

Out[67]:
```
AUC на тестовой выборке: 0.705
Наилучшее значение гиперпараметра C: {'C': 0.6}
Наилучшее значение AUC: 0.702
```

I.19. Проблема дисбаланса классов

Очень многие практические задачи классификации являются несбалансированными, то есть, по меньшей мере, один из классов содержит очень небольшое количество наблюдений. При работе с такими задачами нас может интересовать именно правильная классификация редко встречающего класса (положительного класса). Примерами таких задач являются обнаружение мошенничества, прогнозирование дефолта и диагностика редких заболеваний. Однако наиболее часто используемые алгоритмы классификации плохо решают такие задачи, потому что они направлены на минимизацию общей ошибки классификации, а не на минимизацию ошибки классификации положительных наблюдений.

Давайте посмотрим пропорции классов в наших данных.

In[68]:
```
# смотрим распределение классов зависимой переменной
# в обучающей выборке (относительные частоты)
print(y_train.value_counts(normalize=True))
```

Out[68]:
```
0   0.885
1   0.115
Name: TARGET_1, dtype: float64
```

In[69]:
```
# смотрим распределение классов зависимой переменной
# в обучающей выборке (абсолютные частоты)
print(y_train.value_counts())
```

Out[69]:
```
0    9426
1    1230
Name: TARGET_1, dtype: int64
```

Наш набор данных можно назвать несбалансированным.
Существует два распространенных подхода к решению проблемы крайне несбалансированных данных: изменение весов классов и семплинг.

1.19.1. Изменение весов классов

Первый подход предлагает использовать в ходе обучения разные стоимости ошибочной классификации. Ошибкам отнесения к классам зависимой переменной мы можем назначить разные цены в зависимости от ценности категории. Например, предположим, что пациент относится к одному из двух классов: *Здоров* (отрицательный класс) и *Болен* (положительный класс). Ошибочное отнесение больного пациента к классу *Здоров*, вероятно, является ошибкой, имеющей более высокую цену, чем ошибочное отнесение здорового пациента к классу *Болен*. Итак, мы определяем, что стоимость ошибочной классификации наблюдений миноритарного класса выше, и пытаемся минимизировать общую стоимость ошибочной классификации.
Давайте применительно к нашим данным попробуем варьировать вес миноритарного класса.

```
In[70]:
from sklearn.model_selection import cross_val_score

best_score = 0

for class_weight in ['balanced', {0:0.70, 1:0.30}, {0:0.75, 1:0.25},
                     {0:0.80, 1:0.20}, {0:0.85, 1:0.15}]:
    # для разных пропорций классов, задаваемых class_weight,
    # обучаем логистическую регрессию
    logreg = LogisticRegression(class_weight=class_weight,
                                random_state=42)
    scores = cross_val_score(logreg, X_train, y_train, scoring='roc_auc', cv=strat)
    # вычисляем среднее значение AUC перекрестной проверки
    auc_score = np.mean(scores)
    # если получаем лучшее значение AUC, сохраняем его и значение параметра
    if auc_score > best_score:
        best_score = auc_score
        best_parameters = {'class_weight': class_weight}
# строим модель со значением гиперпараметра class_weight, давшим наилучшее
# значение AUC, на обучающей выборке
logreg_best = LogisticRegression(**best_parameters, random_state=42)
logreg_best.fit(X_train, y_train)
# проверяем качество модели на контрольной выборке
test_score = roc_auc_score(y_test, logreg_best.predict_proba(X_test)[:, 1])
print("Лучшее усредненное значение AUC cv: {:.2f}".format(best_score))
print("Наилучшее значение гиперпараметра class_weight: ", best_parameters)
print("AUC модели на тестовой выборке: {:.2f}".format(test_score))

Out[70]:
Лучшее усредненное значение AUC cv: 0.70
Наилучшее значение гиперпараметра class_weight:  {'class_weight': {0: 0.85, 1: 0.15}}
AUC модели на тестовой выборке: 0.69
```

1.19.2. Семплинг

Второй подход заключается в использовании семплинга. Мы можем удалить некоторое количество примеров мажоритарного класса (данную технику называют андерсемплингом) или увеличить количество примеров миноритарного класса (эта техника называется оверсемплингом). Удалить примеры мажоритарного класса или

увеличить количество примеров миноритарного класса можно случайным образом или по специальным правилам. Можно также сочетать андерсемплинг и оверсемплинг.
Обратите внимание, что присвоение весов и любой семплинг, будь то андерсемплинг или оверсемплинг, случайный или по определенным правилам, мы осуществляем только на обучающей выборке, а затем смотрим, как модель, обученная по взвешенным/семлированным данным, работает на контрольных данных, не подвергшихся взвешиванию/семплингу. Мы должны помнить, что при работе с новыми данными у нас не будет никакой зависимой переменной, и мы не можем вновь применить взвешивание/семплинг, у нас есть лишь модель, построенная на взвешенных/семплированных обучающих данных.
В питоновской библиотеке `scikit-learn` семплинг выполняется с помощью различных классов библиотеки `imbalanced-learn`. Если вы используете дистрибутив Anaconda, эту библиотеку можно установить в Anaconda Promt с помощью команды `conda install -c conda-forge imbalanced-learn`.

1.19.2.1. Случайное удаление примеров мажоритарного класса (Random Undersampling)

В рамках случайного андерсемплинга мы вычисляем K – количество мажоритарных примеров, которое необходимо удалить для достижения требуемого уровня соотношения различных классов. Затем случайным образом мы отбираем K мажоритарных примеров и удаляем.
Давайте попробуем применить эту стратегию с помощью класса `RandomUnderSampler` библиотеки `imbalanced-learn`.

In[71]:
```
# импортируем класс RandomUnderSampler
from imblearn.under_sampling import RandomUnderSampler
```

Ниже приводится список гиперпараметров для класса `RandomUnderSampler`.

Гиперпараметр	Предназначение
`ratio`	Задает соотношение числа объектов в миноритарном и мажоритарном классах. Если передать словарь, ключами будут классы, значениями – количество наблюдений. Значение `'majority'` выполняет семплинг мажоритарного класса. Однако наилучшее качество обычно дает тонкая настройка с помощью словаря.
`random_state`	Задает стартовое значение генератора случайных чисел для воспроизводимости.

Создаем экземпляр класса `RandomUnderSampler`, настроив гиперпараметры. Например, если задать `ratio={0:9000, 1:1230}`, то применительно к нашему случаю мы оставляем все примеры

миноритарного класса (1230), а из исходных 9426 наблюдений мажоритарного класса удаляем 426 наблюдений, оставив 9000 наблюдений. Также не забываем про гиперпараметр `random_state`, который задает стартовое значение генератора случайных чисел для воспроизводимости результатов.

```
In[72]:
# создаем экземпляр класса RandomUnderSampler
rus = RandomUnderSampler(ratio={0:9000, 1:1230}, random_state=42)
```

Затем обучаем нашу модель, то есть выполняем андерсемплинг с помощью метода `fit_sample`, создав новый массив признаков и новый массив меток для обучения.

```
In[73]:
# обучаем модель RandomUnderSampler, т.е. выполняем семплинг,
# создав новый обучающий массив признаков и обучающий
# массив меток
X_resampled, y_resampled = rus.fit_sample(X_train, y_train)
```

И вот на этих новых массивах мы обучаем модель логистической регрессии и проверяем ее на контрольной выборке, не участвовавшей в семплировании.

```
In[74]:
# на семплированных наборах данных строим модель логистической регрессии
logreg = LogisticRegression().fit(X_resampled, y_resampled)
print("AUC на обучающей выборке: {:.3f}".
      format(roc_auc_score(y_resampled, logreg.predict_proba(X_resampled)[:, 1])))
# проверяем качество модели на контрольной выборке,
# не участвовавшей в семплировании
print("AUC на контрольной выборке: {:.3f}".
      format(roc_auc_score(y_test, logreg.predict_proba(X_test)[:, 1])))

Out[74]:
AUC на обучающей выборке: 0.748
AUC на контрольной выборке: 0.691
```

1.19.2.2. Связи Томека

Примеры мажоритарного класса могут удаляться не только случайным образом, но и по определенным правилам. Одной из таких стратегий являются связи Томека.

Пусть примеры E_i и E_j принадлежат различным классам, $d(E_i, E_j)$ – расстояние между указанными примерами. Пара (E_i, E_j) называется связью Томека, если не найдется ни одного примера E_l такого, что будет справедлива система неравенств:

$$\begin{cases} d(E_i, E_l) < d(E_i, E_j) \\ d(E_j, E_l) < d(E_i, E_j) \end{cases}$$

Согласно данному подходу, все мажоритарные записи, входящие в связи Томека, должны быть удалены из набора данных. Этот способ хорошо удаляет записи, которые можно рассматривать в качестве «зашумляющих».

Давайте применим эту стратегию с помощью класса `TomekLinks` библиотеки `imbalanced-learn`.

In[75]:
```
# импортируем класс TomekLinks
from imblearn.under_sampling import TomekLinks
```

Здесь также есть гиперпараметр `ratio`, задающий соотношение числа объектов в миноритарном и мажоритарном классах. Гиперпараметр `random_state` задает стартовое значение генератора случайных чисел для воспроизводимости результатов. Поскольку вычисление мажоритарных примеров, входящих в связи Томека, является затратной процедурой с вычислительной точки зрения, то с помощью параметра `n_jobs` можно задать количество ядер процессора для распараллеливания вычислений.

In[76]:
```
# создаем экземпляр класса TomekLinks
tomek = TomekLinks(ratio='majority', random_state=42, n_jobs=-1)
# обучаем модель TomekLinks, т.е. выполняем семплинг,
# создав новые наборы данных
X_resampled, y_resampled = tomek.fit_sample(X_train, y_train)
# на семплированных наборах данных строим модель логистической регрессии
logreg = LogisticRegression().fit(X_resampled, y_resampled)
print("AUC на обучающей выборке: {:.3f}".
      format(roc_auc_score(y_resampled, logreg.predict_proba(X_resampled)[:, 1])))
# проверяем качество модели на контрольной выборке,
# не участвовавшей в семплировании
print("AUC на контрольной выборке: {:.3f}".
      format(roc_auc_score(y_test, logreg.predict_proba(X_test)[:, 1])))
```

Out[76]:
```
AUC на обучающей выборке: 0.752
AUC на контрольной выборке: 0.692
```

1.19.2.3. Случайное дублирование примеров миноритарного класса (Random Oversampling)

В рамках случайного оверсемплинга мы случайным образом дублируем примеры миноритарного класса. В зависимости от того, какое соотношение классов требуется, выбирается необходимое количество случайных записей для дублирования.

Для выполнения случайного оверсемплинга нужно воспользоваться классом `RandomOverSampler` библиотеки `imbalanced-learn`.

In[77]:
```
# импортируем класс RandomOverSampler
from imblearn.over_sampling import RandomOverSampler
```

Главным является гиперпараметр `ratio`. Для него задают значение `'minority'`, и тогда выполняется случайное дублирование примеров миноритарного класса. Также можно передать словарь, ключами будут классы, а значениями – количество наблюдений. Например, если задать `ratio = {0:9426, 1:1300}`, то применительно к нашему случаю мы оставляем все примеры мажоритарного класса (9426), а из исходных 1230 наблюдений миноритарного класса создаем дополнительно 70 наблюдений путем дублирования. Наилучшее качество обычно дает

тонкая настройка с помощью словаря. Не забываем про гиперпараметр `random_state` для получения воспроизводимых результатов.

```
In[78]:
# создаем экземпляр класса RandomOverSampler
ros = RandomOverSampler(ratio={0:9426, 1:1300}, random_state=42)
# обучаем модель RandomOverSampler, т.е. выполняем семплинг,
# создав новые наборы данных
X_resampled, y_resampled = ros.fit_sample(X_train, y_train)
# на семплированных наборах данных строим модель логистической регрессии
logreg = LogisticRegression().fit(X_resampled, y_resampled)
print("AUC на обучающей выборке: {:.3f}".
      format(roc_auc_score(y_resampled, logreg.predict_proba(X_resampled)[:, 1])))
# проверяем качество модели на контрольной выборке,
# не участвовавшей в семплировании
print("AUC на контрольной выборке: {:.3f}".
      format(roc_auc_score(y_test, logreg.predict_proba(X_test)[:, 1])))

Out[78]:
AUC на обучающей выборке: 0.749
AUC на контрольной выборке: 0.690
```

1.19.2.4. SMOTE

SMOTE (Syntetic Minority Oversampling Technique – оверсемплинг за счет создания синтетических примеров миноритарного класса) генерирует синтетические примеры миноритарного класса, чтобы увеличить долю миноритарного класса в выборке. Для каждого примера миноритарного класса мы вычисляем его k ближайших соседей (обычно $k = 5$) и затем из них случайным образом выбираем некоторые примеры в соответствии с нашим уровнем оверсемплинга. После этого вдоль линий, соединяющих пример миноритарного класса с его выбранными ближайшими соседями, генерируются новые синтетические примеры.

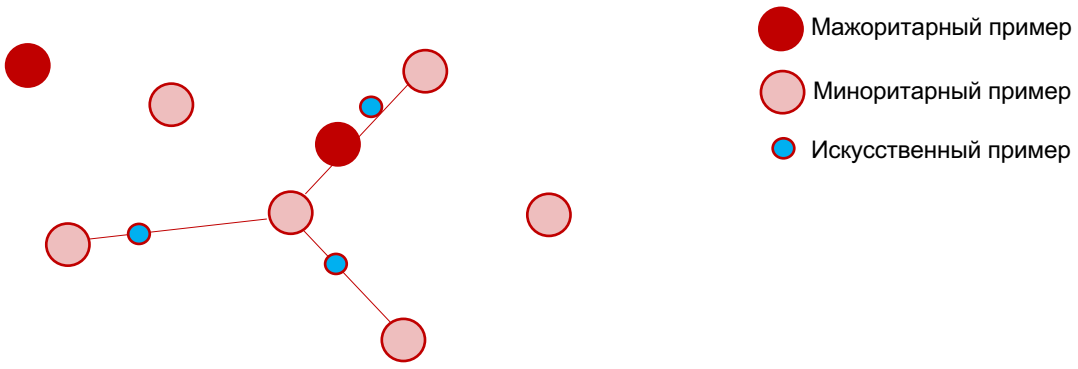


Рис. 1 Генерирование искусственных миноритарных примеров в SMOTE

В отличие от случайного оверсемплинга, реализованного в классическом SMOTE, модифицированный SMOTE (borderline-SMOTE1 и borderline-SMOTE2) выполняет отбор и усиление пограничных примеров миноритарного класса (примеров миноритарного класса, расположенных на границе принятия решений). Сначала мы находим пограничные

примеры миноритарного класса, затем генерируем из них синтетические примеры и затем добавляем в исходную обучающую выборку.
Пусть весь обучающий набор – это T, миноритарный класс – P, а мажоритарный класс – N и тогда

$$P = \{p_1, p_2, \dots, p_{pnum}\}, N = \{n_1, n_2, \dots, n_{nnum}\}$$

где $pnum$ и $nnum$ – это количество примеров миноритарного и мажоритарного классов. Тогда детальная процедура borderline-SMOTE1 выглядит так:
Этап 1. Для каждого примера $p_i\{i = 1,2, \dots, pnum\}$ в миноритарном классе P мы вычисляем m ближайших соседей из всей обучающей выборки T. Количество примеров мажоритарного классов, попавших в m ближайших соседей, помечается как $m'(0 \leq m' \leq m)$.
Этап 2. Если $m' = m$, т.е. все m ближайших соседей примера p_i являются примерами мажоритарного класса, пример p_i рассматривается как зашумленный и в дальнейших этапах не участвует. Если $m / 2 \leq m' < m$, т.е. если для примера p_i среди его ближайших соседей больше половины – это примеры мажоритарного класса, то пример p_i рассматривается как пример, имеющий риск неправильной классификации, и попадает в набор под названием *DANGER* («опасная зона»). Если $0 \leq m' < m / 2$, т.е. если для примера p_i среди его ближайших соседей больше половины – это примеры миноритарного класса, то такой пример считается надежным и не участвует в следующих этапах.

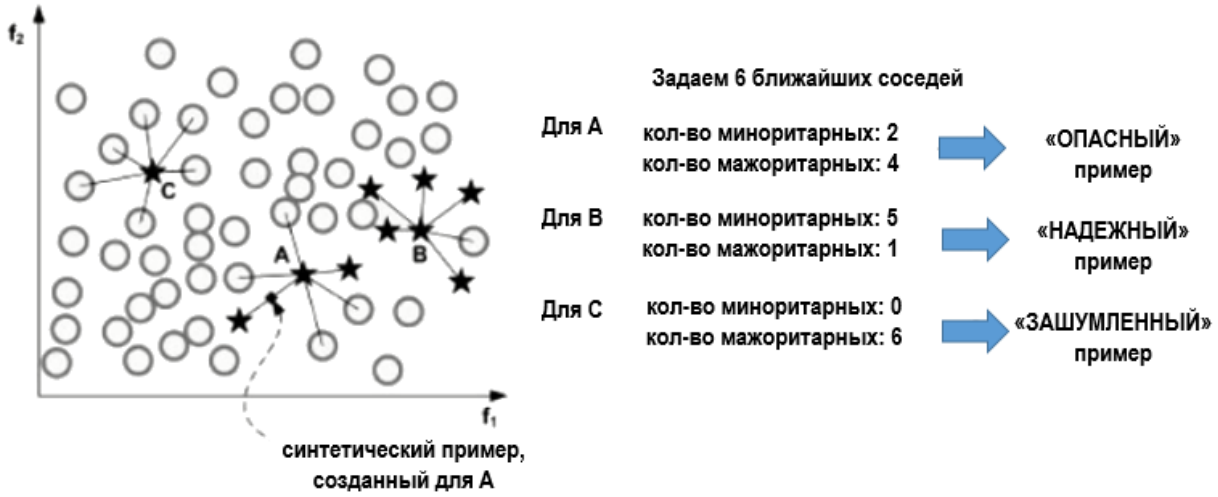


Рис. 2 Три типа примеров в алгоритме SMOTE

Этап 3. Примеры, попавшие в набор *DANGER*, являются пограничными примерами миноритарного класса P, и мы можем увидеть, что каждый пример из набора *DANGER* также является примером миноритарного класса.

Мы задаем

$$DANGER = \{p'_1, p'_2, \dots, p'_{dnum}\}, \qquad 0 \le dnum \le pnum$$

Для каждого примера, входящего в набор *DANGER*, мы вычисляем k ближайших соседей из *P*.

Этап 4. На этом этапе мы генерируем $s \times dnum$ синтетических примеров миноритарного класса из примеров, вошедших в набор *DANGER*, где s – это целое число от 1 до k. Для каждого p'_i мы случайно отбираем s ближайших соседей из k ближайших соседей в *P*. Сначала мы вычисляем разности $diff_j (j = 1,2, \dots, s)$ между p'_i и s ближайшими соседями из *P*, затем умножаем $diff_j$ на случайное число $r_j (j = 1,2, \dots, s)$, лежащее в интервале от 0 до 1, в итоге генерируем s новых синтетических примеров миноритарного класса, расположенных между p'_i и его ближайшими соседями:

$$synthetic_j = p'_i + r_j \times diff_j, \quad j = 1,2, \dots, s$$

Мы повторяем вышеприведенную процедуру для каждого примера p'_i, вошедшего в набор *DANGER*, и получаем $s \times dnum$ синтетических примеров.

В процедуре выше p_i, n_i, p'_i, $diff_j$ и $synthetic_j$ являются векторами. Мы видим, что новые синтетические данные сгенерированы вдоль линий, соединяющих пограничные примеры миноритарного класса с их ближайшими соседями – примерами того же класса, таким образом, происходит усиление этих пограничных примеров.

Borderline-SMOTE2 не только генерирует синтетические примеры на основе каждого примера, вошедшего в набор *DANGER*, и его ближайших соседей – примеров миноритарного класса *P*, но и делает то же самое, используя ближайших соседей – примеры мажоритарного класса *N*. Разность между p'_i и его ближайшим соседом – примером мажоритарного класса умножается на случайное число, лежащее в интервале от 0 до 0,5, таким образом, новые сгенерированные примеры располагаются ближе к миноритарному классу.

Нам понадобится класс `SMOTE` библиотеки `imbalanced-learn`.

In[79]:
```
# импортируем класс SMOTE
from imblearn.over_sampling import SMOTE
```

Здесь помимо гиперпараметров `ratio` и `random_state` используется ряд дополнительных гиперпараметров. Гиперпараметр `kind` определяет тип алгоритма SMOTE. Можно задать значения `'regular'` (по умолчанию), `'borderline1'`, `'borderline2'`, `'svm'`. Чаще всего значения `'borderline1'` и `'borderline2'` дают лучшее качество. Гиперпараметр `k_neighbors` задает количество ближайших соседей для создания синтетических примеров (по умолчанию используется значение 5). Гиперпараметр `m_neighbors` задает количество ближайших

соседей, которое требуется для идентификации миноритарного примера как «опасного» (по умолчанию используется значение 10). Гиперпараметр `m_neighbors` используется только со значениями `kind = {'borderline1', 'borderline2', 'svm'}`. С помощью параметра `n_jobs` можно задать количество ядер процессора для распараллеливания вычислений. Применительно к нашему примеру мы используем Borderline-SMOTE2.

```
In[80]:
# применяем  borderline-SMOTE2
smote = SMOTE(random_state=152, kind='borderline2',
              n_jobs=-1)
X_smote, y_smote = smote.fit_sample(X_train, y_train)
# на семплированных наборах данных строим модель логистической регрессии
logreg = LogisticRegression().fit(X_smote, y_smote)
print("AUC на обучающей выборке: {:.3f}".
      format(roc_auc_score(y_smote, logreg.predict_proba(X_smote)[:, 1])))
# проверяем качество модели на контрольной выборке,
# не участвовавшей в семплировании
print("AUC на контрольной выборке: {:.3f}".
      format(roc_auc_score(y_test, logreg.predict_proba(X_test)[:, 1])))

Out[80]:
AUC на обучающей выборке: 0.775
AUC на контрольной выборке: 0.686
```

Отметим, что изменение пропорций классов с помощью весов и семплинга – это самый последний инструмент в арсенале средств аналитика. Проблему дисбаланса классов нужно попытаться решить сперва за счет конструирования новых признаков, обогащения данными из других источников, настройки параметров и ансамблирования моделей. И только когда все традиционные способы улучшения качества модели уже испробованы, можно применить специальные техники устранения дисбаланса классов. Применительно к нашему примеру ни одна из техник взвешивания и семплирования не дала существенного прироста качества.

Этап II. Построение модели на всей исторической выборке и применение к новым данным

II.1. Считывание CSV-файла, содержащего исторические данные, в объект DataFrame

Давайте заново считаем наши исторические данные в датафрейм `data`.

```
In[81]:
# загружаем набор данных
fulldata = pd.read_csv('Data/Credit_OTP.csv', encoding='cp1251', sep=';')
```

II.2. Предварительная обработка исторических данных

Теперь нам предстоит написать функцию, которая будет выполнять предварительную обработку исторических данных, ее мы применим ко

всей исторической выборке, а позже – к новым данным. Написание единой функции предварительной обработки важно по той причине, что новые данные нужно подготовить точно так же, как мы подготавливали исторические данные (должны совпадать названия и типы переменных, стратегия обработки пропусков и редких категорий, точки разбиения, которые использовались для формирования бинов, преобразования переменных, максимизирующие нормальность, и т.д.), и лучше это сделать в автоматическом режиме. Это позволит избежать расхождений в стратегиях предварительной подготовки на исторических (обучающих) и новых данных.

In[82]:

```
# пишем функцию, выполняющую предварительную обработку
# исторических данных
def preprocessing(df):
    # удаляем идентификационную переменную AGREEMENT_RK,
    # потому что у нее количество уникальных значений
    # равно количеству наблюдений
    df.drop('AGREEMENT_RK', axis=1, inplace=True)

    # удаляем переменную DL_DOCUMENT_FL, потому что
    # у нее одно уникальное значение
    df.drop('DL_DOCUMENT_FL', axis=1, inplace=True)

    # преобразуем указанные переменные в тип object
    for i in ['TARGET', 'SOCSTATUS_WORK_FL', 'SOCSTATUS_PENS_FL', 'GENDER',
              'REG_FACT_FL', 'FACT_POST_FL', 'REG_POST_FL', 'REG_FACT_POST_FL',
              'REG_FACT_POST_TP_FL', 'FL_PRESENCE_FL', 'AUTO_RUS_FL',
              'HS_PRESENCE_FL', 'COT_PRESENCE_FL', 'GAR_PRESENCE_FL',
              'LAND_PRESENCE_FL', 'GPF_DOCUMENT_FL', 'FACT_PHONE_FL',
              'REG_PHONE_FL', 'GEN_PHONE_FL']:
        df[i] = df[i].astype('object')

    # в указанных переменных заменяем запятую на точку в качестве
    # десятичного разделителя и преобразуем в тип float
    for i in ['PERSONAL_INCOME', 'CREDIT', 'FST_PAYMENT',
              'LOAN_AVG_DLQ_AMT', 'LOAN_MAX_DLQ_AMT']:
        df[i] = df[i].str.replace(',', '.').astype('float')


    # если в интересующей нас переменной есть пропуск
    # и при этом переменная SOCSTATUS_PENS_FL имеет значение 1,
    # заменяем такие пропуски меткой "Не указано"
    df['GEN_INDUSTRY'] = np.where(df['GEN_INDUSTRY'].isnull() \
                                  | (df['SOCSTATUS_PENS_FL'] == 1),
                                  'Не указано', df['GEN_INDUSTRY'])
    df['GEN_TITLE'] = np.where(df['GEN_TITLE'].isnull() \
                               | (df['SOCSTATUS_PENS_FL'] == 1),
                               'Не указано', df['GEN_TITLE'])
    df['ORG_TP_STATE'] = np.where(df['ORG_TP_STATE'].isnull() \
                                  | (df['SOCSTATUS_PENS_FL'] == 1),
                                  'Не указано', df['ORG_TP_STATE'])
    df['ORG_TP_FCAPITAL'] = np.where(df['ORG_TP_FCAPITAL'].isnull() \
                                     | (df['SOCSTATUS_PENS_FL'] == 1),
                                     'Не указано', df['ORG_TP_FCAPITAL'])

    # заменяем пропуски в указанных переменных
    # меткой "Не указано"
    df['JOB_DIR'] = np.where(df['JOB_DIR'].isnull(), 'Не указано', df['JOB_DIR'])
    df['REGION_NM'] = np.where(df['REGION_NM'].isnull(), 'Не указано', df['REGION_NM'])
    # пропуски в переменной TP_PROVINCE заменим значением
    # переменной FACT_ADDRESS_PROVINCE
    df['TP_PROVINCE'] = np.where(df['TP_PROVINCE'].isnull(),
                                 df['FACT_ADDRESS_PROVINCE'], df['TP_PROVINCE'])

    # заменяем пропуски в переменной
    # PREVIOUS_CARD_NUM_UTILIZED нулями
```

```
df['PREVIOUS_CARD_NUM_UTILIZED'] = np.where(df['PREVIOUS_CARD_NUM_UTILIZED'].isnull(), 0,
                                            df['PREVIOUS_CARD_NUM_UTILIZED'])

# заменяем категорию "Не указано" на категорию "ЮЖНЫЙ"
df['REGION_NM'] = np.where(df['REGION_NM'] == 'Не указано', 'ЮЖНЫЙ', df['REGION_NM'])

# заменяем неверную категорию "ПОВОЛЖСКИЙ" на категорию "ПРИВОЛЖСКИЙ"
df.at[df['REGION_NM'] == 'ПОВОЛЖСКИЙ', 'REGION_NM'] = 'ПРИВОЛЖСКИЙ'

# записываем редкие категории в одну отдельную категорию
for i in ['REG_ADDRESS_PROVINCE', 'POSTAL_ADDRESS_PROVINCE', 'FACT_ADDRESS_PROVINCE']:
    df[i] = np.where((df[i] == 'Москва') \
                     | (df[i] == 'Хакасия') \
                     | (df[i] == 'Ямало-Ненецкий АО') \
                     | (df[i] == 'Магаданская область') \
                     | (df[i] == 'Калмыкия') \
                     | (df[i] == 'Дагестан') \
                     | (df[i] == 'Агинский Бурятский АО') \
                     | (df[i] == 'Усть-Ордынский Бурятский АО') \
                     | (df[i] == 'Эвенкийский АО') \
                     | (df[i] == 'Коми-Пермяцкий АО') \
                     | (df[i] == 'Чечня'),
                     'ДРУГОЕ', df[i])


df['TP_PROVINCE'] = np.where((df['TP_PROVINCE'] == 'Сахалинская область') \
                             | (df['TP_PROVINCE'] == 'Еврейская АО') \
                             | (df['TP_PROVINCE'] == 'Магаданская область') \
                             | (df['TP_PROVINCE'] == 'Дагестан') \
                             | (df['TP_PROVINCE'] == 'Кабардино-Балкария'),
                             'ДРУГОЕ', df['TP_PROVINCE'])

# укрупняем категории переменной EDUCATION
df.at[df['EDUCATION'] == 'Ученая степень', 'EDUCATION'] = 'Высшее'
df.at[df['EDUCATION'] == 'Два и более высших образования',
      'EDUCATION'] = 'Высшее'

# записываем некоторые категории переменной GEN_INDUSTRY
# в отдельную категорию
df.at[df['GEN_INDUSTRY'] == 'Юридические услуги/нотариальные услуги',
      'GEN_INDUSTRY'] = 'Другие сферы'
df.at[df['GEN_INDUSTRY'] == 'Страхование', 'GEN_INDUSTRY'] = 'Другие сферы'
df.at[df['GEN_INDUSTRY'] == 'Туризм', 'GEN_INDUSTRY'] = 'Другие сферы'
df.at[df['GEN_INDUSTRY'] == 'Недвижимость', 'GEN_INDUSTRY'] = 'Другие сферы'
df.at[df['GEN_INDUSTRY'] == 'Управляющая компания', 'GEN_INDUSTRY'] = 'Другие сферы'
df.at[df['GEN_INDUSTRY'] == 'Логистика', 'GEN_INDUSTRY'] = 'Другие сферы'
df.at[df['GEN_INDUSTRY'] == 'Подбор персонала', 'GEN_INDUSTRY'] = 'Другие сферы'
df.at[df['GEN_INDUSTRY'] == 'Маркетинг', 'GEN_INDUSTRY'] = 'Другие сферы'

# укрупняем категории переменной GEN_TITLE
df.at[df['GEN_TITLE'] == 'Партнер', 'GEN_TITLE'] = 'Другое'
df.at[df['GEN_TITLE'] == 'Военнослужащий по контракту', 'GEN_TITLE'] = 'Другое'

# укрупняем категории переменной ORG_TP_STATE
df.at[df['ORG_TP_STATE'] == 'Частная ком. с инос. капиталом',
      'ORG_TP_STATE'] = 'Частная компания'

# укрупняем категории переменной JOB_DIR
df.at[df['JOB_DIR'] == 'Реклама и маркетинг', 'JOB_DIR'] = 'Другое'
df.at[df['JOB_DIR'] == 'Кадровая служба и секретариат', 'JOB_DIR'] = 'Другое'
df.at[df['JOB_DIR'] == 'Пр-техн. обесп. и телеком.', 'JOB_DIR'] = 'Другое'
df.at[df['JOB_DIR'] == 'Юридическая служба', 'JOB_DIR'] = 'Другое'

# создаем переменную FACT_TP_FL, которая принимает значение 1, если
# область фактического пребывания клиента и область торговой точки,
# где клиент брал последний кредит, совпадают, или 0
# в противном случае
df['FACT_TP_FL'] = np.where(df['FACT_ADDRESS_PROVINCE'] == df['TP_PROVINCE'],
                            1, 0).astype('object')

# создаем переменную AUTO_FOR_FL, которая принимает значение 1,
# если у клиента – импортный автомобиль, или 0 в противном случае
```

```
df['AUTO_FOR_FL'] = np.where((df['AUTO_RUS_FL'] == '0') & (df['OWN_AUTO'] > 0),
                             1, 0).astype('object')

# создаем переменные - результаты конъюнкций
df['GENDER+GAR_PRESENCE_FL'] = df.apply(
    lambda x: f"{x['GENDER']} + {x['GAR_PRESENCE_FL']}",
    axis=1)
df['REG_FACT_FL+GAR_PRESENCE_FL'] = df.apply(
    lambda x: f"{x['REG_FACT_FL']} + {x['GAR_PRESENCE_FL']}",
    axis=1)

# значения переменной FACT_LIVING_TERM берем по модулю,
# чтоб избавиться от отрицательных значений
df['FACT_LIVING_TERM'] = df['FACT_LIVING_TERM'].abs()

# наблюдения, в которых количество лет проживания
# по месту фактического пребывания, превышает
# возраст, записываем как пропуски
df['FACT_LIVING_TERM'] = np.where(df['FACT_LIVING_TERM'] / 12 > df['AGE'],
                                  np.NaN, df['FACT_LIVING_TERM'])

# импутируем пропуски медианой
df['FACT_LIVING_TERM'].fillna(df['FACT_LIVING_TERM'].median(), inplace=True)

# наблюдения, в которых время работы в годах превышает
# возраст (например, человек работает 40 лет, а живет
# всего 25), записываем как пропуски
df['WORK_TIME'] = np.where(df['WORK_TIME'] / 12 > df['AGE'],
                           np.NaN, df['WORK_TIME'])

# наблюдения, в которых разница между возрастом и временем работы в годах
# меньше 16 (например, у 30-летнего время работы в годах составляет 20 лет,
# получается, он работает с 10 лет), записываем как пропуски
df['WORK_TIME'] = np.where((df['AGE'] - df['WORK_TIME'] / 12) < 16,
                           np.NaN, df['WORK_TIME'])

# импутируем пропуски в переменной WORK_TIME медианой
df['WORK_TIME'].fillna(df['WORK_TIME'].median(), inplace=True)

# на основе переменной PERSONAL_INCOME создаем переменную
# PERSONAL_INCOME_CAT, у которой метки категорий будут
# повторять метки категорий FAMILY_INCOME
bins = [-np.inf, 5000, 10000, 20000, 50000, np.inf]
lab = ['до 5000 руб.', 'от 5000 до 10000 руб.', 'от 10000 до 20000 руб.',
       'от 20000 до 50000 руб.', 'свыше 50000 руб.']
df['PERSONAL_INCOME_CAT'] = pd.cut(df['PERSONAL_INCOME'], bins, labels=lab)

# создаем переменную PERSONAL_FAMILY_INCOME, которая принимает значение 1, если
# категория переменной PERSONAL_INCOME_CAT совпадает с категорией
# переменной FAMILY_INCOME, или 0 в противном случае, затем
# преобразовываем в тип object
df['PERSONAL_FAMILY_INCOME'] = np.where(df['PERSONAL_INCOME_CAT'] == df['FAMILY_INCOME'],
                                       1, 0).astype('object')

# удаляем переменную PERSONAL_INCOME_CAT
df.drop('PERSONAL_INCOME_CAT', axis=1, inplace=True)

# задаем точки, в которых будут находится границы категорий
# будущей переменной CREDITCAT
bins = [-np.inf, 7292, 9427, 14169, 27449, np.inf]

# осуществляем биннинг переменной CREDIT и записываем
# результаты в новую переменную CREDITCAT
df['CREDITCAT'] = pd.cut(df['CREDIT'], bins).astype('object')

# задаем точки, в которых будут находится границы категорий
# будущей переменной FSTPAYMENTCAT
bins = [-np.inf, 1500, 4995, np.inf]

# осуществляем биннинг переменной FST_PAYMENT и записываем
# результаты в новую переменную FSTPAYMENTCAT
df['FSTPAYMENTCAT'] = pd.cut(df['FST_PAYMENT'], bins).astype('object')
```

```
# задаем точки, в которых будут находится границы категорий
# будущей переменной AGECAT
bins = [-np.inf, 29, 43, 52, 57, np.inf]

# осуществляем биннинг переменной AGE и записываем
# результаты в новую переменную AGECAT
df['AGECAT'] = pd.cut(df['AGE'], bins).astype('object')

# пишем функцию, которая создает
# из списка списков словарь
def list_to_dict(input_list):
    output_dict = {}
    for n, sample_list in enumerate(input_list):
        for value in sample_list:
            output_dict[value] = n
    return output_dict

# создаем список списков для
# переменной GEN_TITLE
map_df_list = [
    ['Рабочий',
     'Служащий',
     'Работник сферы услуг',
     'Другое',
     'Индивидуальный предприниматель',
     'Руководитель низшего звена'],
    ['Специалист',
     'Руководитель среднего звена'],
    ['Не указано'],
    ['Высококвалифиц. специалист',
     'Руководитель высшего звена']
]

# создаем из списка списков словарь
map_df_dict = list_to_dict(map_df_list)

# укрупняем категории переменной GEN_TITLE
df['GEN_TITLE'] = df['GEN_TITLE'].map(map_df_dict).astype('object')

# создаем список списков для
# переменной REGION_NM
map_df_list2 = [
    ['ЮЖНЫЙ', 'ЦЕНТРАЛЬНЫЙ 1', 'ЦЕНТРАЛЬНЫЙ 2'],
    ['ЗАПАДНО-СИБИРСКИЙ', 'ВОСТОЧНО-СИБИРСКИЙ', 'УРАЛЬСКИЙ',
     'ДАЛЬНЕВОСТОЧНЫЙ', 'СЕВЕРО-ЗАПАДНЫЙ', 'ЦЕНТРАЛЬНЫЙ ОФИС'],
    ['ПРИВОЛЖСКИЙ']
]

# создаем из списка списков словарь
map_df_dict2 = list_to_dict(map_df_list2)

# на основе укрупнения категорий переменной REGION_NM
# создадим переменную REGIONCAT
df['REGIONCAT'] = df['REGION_NM'].map(map_df_dict2).astype('object')

# создаем список списков для
# переменной ORG_TP_STATE
map_df_list3 = [
    ['Частная компания',
     'Индивидуальный предприниматель',
     'Некоммерческая организация'],
    ['Не указано'],
    ['Государственная комп./учреж.']
]

# создаем из списка списков словарь
map_df_dict3 = list_to_dict(map_df_list3)

# на основе укрупнения категорий переменной ORG_TP_STATE
# создаем переменную ORGCAT
df['ORGCAT'] = df['ORG_TP_STATE'].map(map_df_dict3).astype('object')
```

```
# задаем константу
a = 0.01
# выполняем логарифмическое преобразование
# переменной PERSONAL_INCOME
df['PERSONAL_INCOME'] = np.log(df['PERSONAL_INCOME'] + a)

# выполняем логарифмическое преобразование
# переменной CREDIT
df['CREDIT'] = np.log(df['CREDIT'] + a)

# выполняем логарифмическое преобразование
# переменной WORK_TIME
df['WORK_TIME'] = np.log(df['WORK_TIME'] + a)

# выполняем преобразование переменной FACT_LIVING_TERM
# кубическим корнем
df['FACT_LIVING_TERM'] = np.sign(df['FACT_LIVING_TERM']) * (
    df['FACT_LIVING_TERM'].abs() ** (1/3))

# выполняем логарифмическое преобразование
# переменной LOAN_AVG_DLQ_AMT
df['LOAN_AVG_DLQ_AMT'] = np.log(df['LOAN_AVG_DLQ_AMT'] + a)

# создаем переменную CHILD_DEP
df['CHILD_DEP'] = df['CHILD_TOTAL'] / (df['CHILD_TOTAL'] + df['DEPENDANTS'])

# если переменная содержит пропуск, то возвращается значение 0,
# если пропуска нет, то возвращается исходное значение переменной
df['CHILD_DEP'] = np.where(df['CHILD_DEP'].isnull(), 0, df['CHILD_DEP'])

# создаем переменную PAYMENT - сумму ежемесячного взноса
# по кредиту, разделив сумму кредита (CREDIT)
# на срок кредита (TERM)
df['PAYMENT'] = np.log((df['CREDIT'] / df['TERM']) + a)

# создаем переменную PTI - коэффициент долговой нагрузки,
# разделив сумму ежемесячного взноса по кредиту (PAYMENT)
# на личный доход (PERSONAL_INCOME)
df['PTI'] = df['PAYMENT'] / df['PERSONAL_INCOME']

# создаем переменную CLOSED_TO_TOTAL - коэффициент погашения ссуд,
# разделив количество погашенных ссуд (LOAN_NUM_CLOSED)
# на общее количество ссуд (LOAN_NUM_TOTAL)
df['CLOSED_TO_TOTAL'] = df['LOAN_NUM_CLOSED'] / df['LOAN_NUM_TOTAL']

# создаем переменную PAYM_TO_LOAN – отношение количества платежей
# (LOAN_NUM_PAYM) к общему количеству ссуд (LOAN_NUM_TOTAL)
df['PAYM_TO_LOAN'] = df['LOAN_NUM_PAYM'] / df['LOAN_NUM_TOTAL']

# создаем переменную DLQ_TO_PAYM – отношение количества просрочек
# (LOAN_DLQ_NUM) к общему количеству платежей (LOAN_NUM_PAYM)
df['DLQ_TO_PAYM'] = df['LOAN_DLQ_NUM'] / df['LOAN_NUM_PAYM']

# создаем переменную FST_SHARE - отношение суммы первого платежа по кредиту
# к объединенной сумме первого платежа (FST_PAYMENT) и кредита (CREDIT)
df['FST_SHARE'] = df['FST_PAYMENT'] / (df['FST_PAYMENT'] + df['CREDIT'])

# создаем DLQ_TIME - индекс времени наступления максимальной просрочки,
# разделив номер максимальной просрочки (LOAN_MAX_DLQ)
# на количество просрочек (LOAN_DLQ_NUM)
df['DLQ_TIME'] = df['LOAN_MAX_DLQ'] / df['LOAN_DLQ_NUM']
df['DLQ_TIME'] = np.where(df['DLQ_TIME'].isnull(), 0, df['DLQ_TIME'])

# создаем переменную LOAN_MAX_DLQ_RANGE - разницу между максимальной суммой
# просрочки (LOAN_MAX_DLQ_AMT) и средней суммой просрочки (LOAN_AVG_DLQ_AMT)
df['LOAN_MAX_DLQ_RANGE'] = ((df['LOAN_MAX_DLQ_AMT'] - df['LOAN_AVG_DLQ_AMT']) /
                            df['LOAN_AVG_DLQ_AMT']).abs()
df['LOAN_MAX_DLQ_RANGE'] = np.where(df['LOAN_MAX_DLQ_RANGE'].isnull(), 0,
                                    df['LOAN_MAX_DLQ_RANGE'])
```

Давайте применим нашу функцию предварительной обработки к историческим данным.

In[83]:
```
# выполняем предварительную обработку
# исторических данных
preprocessing(fulldata)
```

In[84]:
```
# создаем копию набора с историческими данными
fulldata_copy = fulldata.copy()
# выполняем стандартизацию количественных переменных
numerical_cols = [c for c in fulldata.columns if fulldata[c].dtype.name != 'object']
for i in numerical_cols:
    fulldata[i] = (fulldata[i] - fulldata[i].mean()) / fulldata[i].std()
```

Теперь выполним дамми-кодирование, сформируем массив признаков и массив меток.

In[85]:
```
# выполняем дамми-кодирование
data_dummies = pd.get_dummies(data)
# создаем обучающий массив меток
y = data_dummies.loc[:, 'TARGET_1']
# удаляем из будущего массива признаков результаты
# дамми-кодирования зависимой переменной -
# TARGET_0 и TARGET_1
data_dummies.drop(['TARGET_0', 'TARGET_1'], axis=1, inplace=True)
# создаем обучающий массив признаков
X = data_dummies.loc[:, 'AGE':'ORGCAT_2']
```

II.3. Обучение модели логистической регрессии на всех исторических данных

А сейчас мы обучим на всей исторической выборке модель логистической регрессии с оптимальным значением параметра `C`, найденным по результатам комбинированной проверки.

In[86]:
```
# строим логистическую регрессию
logreg = LogisticRegression(C=0.2, penalty='l1').fit(X, y)
print("AUC на всей исторической выборке: {:.3f}".
      format(roc_auc_score(y, logreg.predict_proba(X)[:, 1])))
```

Out[86]:
```
AUC на всей исторической выборке: 0.727
```

II.4. Считывание CSV-файла, содержащего новые данные, в объект DataFrame

Теперь считываем новые данные, записанные в файле *Credit_OTP_new.csv*, в датафрейм `newdata`.

In[87]:
```
# загружаем новые данные
newdata = pd.read_csv('Data/Credit_OTP_new.csv', encoding='cp1251', sep=';')
```

II.5. Предварительная обработка новых данных

Применяем нашу функцию предварительной обработки к новым данным.

```
In[88]:
# выполняем предварительную обработку
# новых данных
preprocessing(newdata)

In[89]:
# выполняем стандартизацию количественных переменных
for i in numerical_cols:
    newdata[i] = (newdata[i] - fulldata_copy[i].mean()) / fulldata_copy[i].std()
```

Теперь выполним дамми-кодирование, сформируем массив признаков и массив меток, выполним стандартизацию.

```
In[90]:
# выполняем дамми-кодирование
# новых данных
newdata_dummies = pd.get_dummies(newdata)
# создаем массив меток
y_new = newdata_dummies.loc[:, 'TARGET_1']
# удаляем из будущего массива признаков результаты
# дамми-кодирования зависимой переменной -
# TARGET_0 и TARGET_1
newdata_dummies.drop(['TARGET_0', 'TARGET_1'], axis=1, inplace=True)
# создаем массив признаков
X_new = newdata_dummies.loc[:, 'AGE':'ORGCAT_2']
```

II.6. Применение модели логистической регрессии, построенной на всех исторических данных, к новым данным

Применяем модель логистической регрессии, построенной на всех исторических данных, к новым данным.

```
In[91]:
print("AUC на выборке новых данных: {:.3f}".
      format(roc_auc_score(y_new, logreg.predict_proba(X_new)[:, 1])))

Out[91]:
AUC на выборке новых данных: 0.708
```

Теперь, когда вы познакомились с процедурой предварительной подготовки данных для логистической регрессии, знаете, как строить модели логистической регрессии и настраивать параметры в классах `LogisticRegression` и `H2OGeneralizedLinearEstimator`, попробуйте выполнить аналогичные действия для файла данных *Churn_logreg.csv*. Исходная выборка содержит записи о 4431 клиенте, классифицированных на два класса: *Остается* – клиент продолжает пользоваться услугами провайдера (2496 клиентов) и *Уходит* – клиент перестал пользоваться услугами провайдера (1935 клиентов). Необходимо классифицировать клиентов на тех, кто продолжает пользовать услугами компании, и тех, кто перестал ими пользоваться. По каждому наблюдению (клиенту) фиксируются 12 исходных переменных.

Список исходных переменных включает в себя:

- количественный предиктор *Длительность междугородних звонков [longdist]*;
- количественный предиктор *Длительность международных звонков [internat]*;
- количественный предиктор *Длительность местных звонков [local]*;
- категориальный предиктор *Скидка на международные звонки [int_disc]*;
- категориальный предиктор *Тип тарифа [billtype]*;
- категориальный предиктор *Способ оплаты [pay]*;
- количественный предиктор *Возраст [age];*
- количественный предиктор *Пол [gender]*;
- количественный предиктор *Семейный статус [marital]*;
- категориальный предиктор *Количество детей [children]*;
- категориальный предиктор *Ежемесячный доход [income]*;
- категориальная зависимая переменная *Факт оттока клиента [Churn]*.

Критерием качества модели логистической регрессии является оценка AUC, вычисленная на контрольной выборке с применением комбинированной проверки (данные должны быть разбиты на обучающую и контрольную выборки в соотношении 70%/30% и нужно использовать 5-блочную перекрестную проверку). Хорошим результатом является оценка AUC выше 0,9. Затем нужно построить модель с оптимальными значениями параметров на всей исторической выборке и применить к новым данным, размещенным в файле *Churn_new.csv*. Пример решения размещен в тетрадке *Приложение 5._Решение домашнего задания 1.ipynb*.

Кроме того, попробуйте решить конкурсную задачу Give Me Some Credit (описание задачи можно найти на сайте https://www.kaggle.com/c/GiveMeSomeCredit). Необходимые данные записаны в файле *cs_training.csv*. Исходная выборка содержит записи о 150000 клиентах, классифицированных на два класса: 0 − просрочки 90+ нет (139974 клиента) и 1 − просрочка 90+ есть (10026 клиентов).

Список исходных переменных включает в себя:

- категориальный предиктор *Уникальный идентификатор [Unnamed: 0]*;
- категориальная зависимая переменная *Наличие просрочки 90+ по данным банка [SeriousDlqin2yrs]*;
- количественный предиктор *Утилизация [RevolvingUtilizationOfUnsecuredLines]*;
- количественный предиктор *Возраст клиента [age]*;
- количественный предиктор *Количество просрочек 30-59 дней по данным БКИ [NumberOfTime30-59DaysPastDueNotWorse]*;

- количественный предиктор *Соотношение долга к доходу [DebtRatio]*;
- количественный предиктор *Ежемесячный заработок [MonthlyIncome]*;
- количественный предиктор *Количество кредитов [NumberOfOpenCreditLinesAndLoans]*;
- количественный предиктор *Количество просрочек 90+ по данным БКИ [NumberOfTimes90DaysLate]*;
- категориальный предиктор *Количество ипотечных кредитов [NumberRealEstateLoansOrLines]*;
- количественный предиктор *Количество просрочек 60-89 дней по данным БКИ [NumberOfTime60-89DaysPastDueNotWorse]*;
- количественный предиктор *Количество иждивенцев [NumberOfDependents]*.

Критерием качества модели логистической регрессии является оценка AUC, вычисленная на контрольной выборке с применением комбинированной проверки (данные должны быть разбиты на обучающую и контрольную выборки в соотношении 70%/30% и нужно использовать 5-блочную перекрестную проверку). Хорошим результатом является оценка AUC выше 0,86. Затем нужно построить модель с оптимальными значениями параметров на всей исторической выборке и применить к новым данным, размещенным в файле *Churn_new.csv*. Пример решения размещен в тетрадке *Приложение 5._Решение домашнего задания 2.ipynb*.

Наконец, третья задача представляет собой задачу оттока. Имеются исторические данные по пролонгациям безубыточных клиентов страховой компании. Требуется построить модель, прогнозирующую вероятность того, что клиент уйдет из компании по окончании действия договора. Список исходных переменных включает в себя:

- категориальный предиктор *ID полиса [ID]*;
- категориальный предиктор *Марка транспортного средства [Make]*;
- категориальный предиктор *Модель транспортного средства [Model]*;
- количественный предиктор *Мощность транспортного средства в л.с. [Power]*;
- категориальный предиктор *Кредитное транспортное средство или нет [Is_credit]*;
- категориальный предиктор *Месяц начала действия риска [Begin_month]*;
- категориальный предиктор *Месяц окончания действия риска [End_month]*;
- категориальная зависимая переменная *Возобновлен контракт или нет [Is_renewed]*;
- категориальный предиктор *Канал продаж полиса [Sales_channel]*;

- категориальный предиктор *Филиал продаж полиса [Fillial]*;
- количественный предиктор *Минимальный возраст лиц, допущенных к управлению [Min_age]*;
- количественный предиктор *Минимальный стаж вождения лиц, допущенных к управлению [Min_exp]*;
- количественный предиктор *Количество лет пролонгации полиса [Years_renewed]*;
- категориальный предиктор *Пол страхователя [M_F]*;
- категориальный предиктор *Наличие ОСАГО у клиента [OSAGO]*;
- количественный предиктор *Сумма франшизы [Deductible]*;
- категориальный предиктор *Регион регистрации страхователя [M_F]*;
- количественный предиктор *Изменение премии при пролонгации [Price_change]*;
- количественный предиктор *Стоимость транспортного средства (Страховая сумма) [Price_change]*;
- категориальный предиктор *Посредник по полису (Идентификатор агента) [Agent]*.

Критерием качества модели является оценка AUC, вычисленная на контрольной выборке с применением комбинированной проверки (данные должны быть разбиты на обучающую и контрольную выборки в соотношении 70%/30% и нужно использовать 5-блочную перекрестную проверку). Хорошим результатом является оценка AUC выше 0,7. В этой задаче попробуйте применить библиотеки градиентного бустинга. Пример решения размещен в тетрадке *Приложение 5._Решение домашнего задания 3.ipynb*.

ПРИЛОЖЕНИЕ 6. РАБОТА С ДАТАМИ И СТРОКАМИ

Материал подготовлен специалистом исследовательского центра «Гевисста» Александром Аниным.

На практике нам часто приходится работать с данными, содержащими временные ряды, и выполнять нормализацию строковых значений. Давайте посмотрим, как можно с ними работать.

Работа с датами

Загрузим необходимые библиотеки и набор данных, с которым будем работать.

In[1]:
```
# загружаем необходимые библиотеки
import numpy as np
import pandas as pd
```

In[2]:
```
# записываем датафрейм на основе CSV файла, содержащего даты
data = pd.read_csv('Data/MFOcredit.csv', encoding='cp1251', sep=';')
data.head()
```

Out[2]:

	id	date_start	date_end	gender	age	auto	housing	marstatus	regclient	jobtype	region	credits	children	delinq60plus
0	1	03-Jan-2013	12-Jan-2013	Мужской&*	44	Нет	Собственное	Гражданский брак/женат/замужем	Нет	_Официальное	Новосибирская область	Нет	Да	Нет
1	2	03-Jan-2013	17-Jan-2013	Мужской&*	21	Пропуск поля	Живут с родителями	Холост	Нет	_Официальное	Кемеровская область юг	Да	Нет	Нет
2	3	03-Jan-2013	17-Jan-2013	Мужской&*	25	Пропуск поля	Собственное	Холост	Да	_Официальное	Кемеровская область север	Пропуск поля	Нет	Нет
3	4	03-Jan-2013	17-Jan-2013	Женский	47	Пропуск поля	Собственное	Гражданский брак/женат/замужем	Да	Офи&циальное	Кемеровская область север	Нет	Нет	Нет
4	5	03-Jan-2013	17-Jan-2013	Мужской&*	22	Нет	Арендуемое	Гражданский брак/женат/замужем	Нет	Официальное	Кемеровская область север	Да	Да	Нет

Теперь нашим переменным *date_start* и *date_end* нужно присвоить тип `datetime`, чтобы работать с этими переменными как с датами.

In[3]:
```
# переменным date_start и date_end присваиваем тип datetime,
# чтобы работать с этими переменными как с датами
data['date_start'] = pd.to_datetime(data['date_start'])
data['date_end'] = pd.to_datetime(data['date_end'])
data.info()
```

Out[3]:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 35212 entries, 0 to 35211
Data columns (total 14 columns):
id              35212 non-null int64
date_start      35212 non-null datetime64[ns]
date_end        35212 non-null datetime64[ns]
gender          35212 non-null object
age             35212 non-null int64
auto            35212 non-null object
housing         35212 non-null object
marstatus       35212 non-null object
regclient       35212 non-null object
jobtype         35212 non-null object
region          35212 non-null object
credits         35212 non-null object
children        35212 non-null object
delinq60plus    35212 non-null object
dtypes: datetime64[ns](2), int64(2), object(10)
memory usage: 3.8+ MB
```

Мы можем вычислить срок кредита, для этого достаточно вычесть из даты погашения займа дату открытия займа и выразить результат в днях.

In[4]:

```
# вычисляем разницу в днях между датой погашения займа и
# датой открытия займа, т.е. по сути вычисляем срок займа
data['diff'] = (data['date_end'] - data['date_start']).dt.days
data.head()
```

Out[4]:

gender	age	auto	housing	marstatus	regclient	jobtype	region	credits	children	delinq60plus	diff
Мужской&*	44	Нет	Собственное	Гражданский брак/женат/замужем	Нет	_Официальное	Новосибирская область	Нет	Да	Нет	9
Мужской&*	21	Пропуск поля	Живут с родителями	Холост	Нет	_Официальное	Кемеровская область юг	Да	Нет	Нет	14
Мужской&*	25	Пропуск поля	Собственное	Холост	Да	_Официальное	Кемеровская область север	Пропуск поля	Нет	Нет	14
Женский	47	Пропуск поля	Собственное	Гражданский брак/женат/замужем	Да	Офи&циальное	Кемеровская область север	Нет	Нет	Нет	14
Мужской&*	22	Нет	Арендуемое	Гражданский брак/женат/замужем	Нет	Официальное	Кемеровская область север	Да	Да	Нет	14

Теперь давайте из переменной *data_start* извлечем года, кварталы, месяцы, порядковые номера дней, порядковые номера дней недели, названия дней недели, которые по сути будут новыми переменными.

In[5]:

```
# выделяем из переменной data_start года
data['year'] = pd.DatetimeIndex(data['date_start']).year
# а еще можно так data['year'] = data['date_start'].dt.year

# выделяем из переменной data_start кварталы
data['quarter'] = pd.DatetimeIndex(data['date_start']).quarter
# а еще можно так data['quarter'] = data['date_start'].dt.quarter

# выделяем из переменной data_start месяцы, где 1 - январь, 12 - декабрь
data['month'] = pd.DatetimeIndex(data['date_start']).month
# а еще можно так data['month'] = data['date_start'].dt.month

# выделяем из переменной data_start порядковые номера дней от 1 до 31
data['day'] = pd.DatetimeIndex(data['date_start']).day
# а еще можно так data['day'] = data['date_start'].dt.day

# выделяем из переменной data_start дни недели от 0 до 6,
# где 0 - понедельник, 6 - воскресенье
```

```
data['dayofweek'] = pd.DatetimeIndex(data['date_start']).dayofweek
# а еще можно так data['dayofweek'] = data['date_start'].dt.dayofweek

# выделяем из переменной data_start названия дней недели
data['weekday_name'] = pd.DatetimeIndex(data['date_start']).weekday_name
# а еще можно так data['weekday_name'] = data['date_start'].dt.weekday_name
data.head()
```

Out[5]:

credits	children	delinq60plus	diff	year	quarter	month	day	dayofweek	weekday_name
Нет	Да	Нет	9	2013	1	1	3	3	Thursday
Да	Нет	Нет	14	2013	1	1	3	3	Thursday
Пропуск поля	Нет	Нет	14	2013	1	1	3	3	Thursday
Нет	Нет	Нет	14	2013	1	1	3	3	Thursday
Да	Да	Нет	14	2013	1	1	3	3	Thursday

Часто даты в исходных датах могут быть записаны в разных форматах, поэтому важно правильно распарсить их. Это можно сделать с помощью параметра `format` функции `to_datetime` библиотеки pandas.
Давайте запишем датафрейм, содержащий даты в разных форматах.

In[6]:
```
# записываем датафрейм на основе CSV файла,
# содержащего различные форматы даты
data2 = pd.read_csv('Data/DiffDates.csv', encoding='cp1251', sep=';')
data2.head()
```

Out[6]:

	даты в формате dd.mm.yyyy	даты в формате dd.mm.yy	даты в формате dd-Mth-yyyy	даты в формате dd-mm-yy	даты в формате Month dd, yyyy	даты в формате dd.mm.yyyy hh:mm	даты в формате ddMMMyy:hh:mm:ss	даты в формате dd/mm/yyyy
0	01.10.2009	01.10.09	01-Oct-2009	01-10-09	October 1, 2009	01.10.2009 9:30	18APR17:00:00:00	01/10/2009
1	09.12.2008	09.12.08	09-Dec-2008	09-12-08	December 9, 2008	09.12.2008 10:02	10MAR17:00:00:00	09/12/2008
2	01.11.2012	01.11.12	01-Nov-2012	01-11-12	November 01, 2012	01.11.2012 0:05	21FEB17:00:00:00	01/11/2012
3	15.12.2016	15.12.16	15-Dec-2016	15-12-16	December 15, 2016	15.12.2016 16:07	26APR17:00:00:00	15/12/2016
4	11.03.2014	11.03.14	11-Mar-2014	11-03-14	March 11, 2014	11.03.2014 15:08	08APR17:00:00:00	11/03/2014

Теперь присвоим переменным тип `datetime`, задав с помощью параметра `format` паттерн парсинга даты.

In[7]:
```
# присваиваем переменным тип datetime, задав с помощью
# параметра format паттерн парсинга даты
data2['даты в формате dd.mm.yyyy'] = pd.to_datetime(data2['даты в формате dd.mm.yyyy'],
                                                    format='%d.%m.%Y')
data2['даты в формате dd.mm.yy'] = pd.to_datetime(data2['даты в формате dd.mm.yy'],
                                                  format='%d.%m.%y')
data2['даты в формате dd-Mth-yyyy'] = pd.to_datetime(data2['даты в формате dd-Mth-yyyy'],
                                                     format='%d-%b-%Y')
data2['даты в формате dd-mm-yy'] = pd.to_datetime(data2['даты в формате dd-mm-yy'],
                                                  format='%d-%m-%y')
data2['даты в формате Month dd, yyyy'] = pd.to_datetime(data2['даты в формате Month dd, yyyy'],
                                                        format='%B %d, %Y')
data2['даты в формате dd.mm.yyyy hh:mm'] = pd.to_datetime(
    data2['даты в формате dd.mm.yyyy hh:mm'],
    format='%d.%m.%Y %H:%M')
```

```
data2['даты в формате ddMMMyy:hh:mm:ss'] = pd.to_datetime(
    data2['даты в формате ddMMMyy:hh:mm:ss'],
    format='%d%b%y:%H:%M:%S')
data2['даты в формате dd/mm/yyyy'] = pd.to_datetime(
    data2['даты в формате dd/mm/yyyy'],
    format='%d/%m/%Y')
data2.head()
```

Out[7]:

	даты в формате dd.mm.yyyy	даты в формате dd.mm.yy	даты в формате dd-Mth-yyyy	даты в формате dd-mm-yy	даты в формате Month dd, yyyy	даты в формате dd.mm.yyyy hh:mm	даты в формате ddMMMyy:hh:mm:ss	даты в формате dd/mm/yyyy
0	2009-10-01	2009-10-01	2009-10-01	2009-10-01	2009-10-01	2009-10-01 09:30:00	2017-04-18	2009-10-01
1	2008-12-09	2008-12-09	2008-12-09	2008-12-09	2008-12-09	2008-12-09 10:02:00	2017-03-10	2008-12-09
2	2012-11-01	2012-11-01	2012-11-01	2012-11-01	2012-11-01	2012-11-01 00:05:00	2017-02-21	2012-11-01
3	2016-12-15	2016-12-15	2016-12-15	2016-12-15	2016-12-15	2016-12-15 16:07:00	2017-04-26	2016-12-15
4	2014-03-11	2014-03-11	2014-03-11	2014-03-11	2014-03-11	2014-03-11 15:08:00	2017-04-08	2014-03-11

Работа со строками

Изменение регистра строк

Давайте вернемся к ранее прочитанному датафрейму `data`. С помощью метода `.str.lower()` мы можем перевести строковые значения переменной *marstatus* в нижний регистр.

In[8]:

```
# переводим строки (значения переменной marstatus) в нижний регистр
data['marstatus'] = data['marstatus'].str.lower()
data.head()
```

Out[8]:

	id	date_start	date_end	gender	age	auto	housing	marstatus	regclient	jobtype	...	credits	children	delinq60plus
0	1	2013-01-03	2013-01-12	Мужской&*	44	Нет	Собственное	гражданский брак/женат/замужем	Нет	_Официальное	...	Нет	Да	Нет
1	2	2013-01-03	2013-01-17	Мужской&*	21	Пропуск поля	Живут с родителями	холост	Нет	_Официальное	...	Да	Нет	Нет
2	3	2013-01-03	2013-01-17	Мужской&*	25	Пропуск поля	Собственное	холост	Да	_Официальное	...	Пропуск поля	Нет	Нет
3	4	2013-01-03	2013-01-17	Женский	47	Пропуск поля	Собственное	гражданский брак/женат/замужем	Да	Офи&циальное	...	Нет	Нет	Нет
4	5	2013-01-03	2013-01-17	Мужской&*	22	Нет	Арендуемое	гражданский брак/женат/замужем	Нет	Официальное	...	Да	Да	Нет

Теперь, наоборот, переведем строковые значения переменной *marstatus* в верхний регистр.

In[9]:

```
# переводим строки (значения переменной marstatus) в верхний регистр
data['marstatus'] = data['marstatus'].str.upper()
data.head()
```

Out[9]:

	id	date_start	date_end	gender	age	auto	housing	marstatus	regclient	jobtype	...	credits	children	delinq60plus
0	1	2013-01-03	2013-01-12	Мужской&*	44	Нет	Собственное	ГРАЖДАНСКИЙ БРАК/ЖЕНАТ/ ЗАМУЖЕМ	Нет	_Официальное	...	Нет	Да	Нет
1	2	2013-01-03	2013-01-17	Мужской&*	21	Пропуск поля	Живут с родителями	ХОЛОСТ	Нет	_Официальное	...	Да	Нет	Нет
2	3	2013-01-03	2013-01-17	Мужской&*	25	Пропуск поля	Собственное	ХОЛОСТ	Да	_Официальное	...	Пропуск поля	Нет	Нет
3	4	2013-01-03	2013-01-17	Женский	47	Пропуск поля	Собственное	ГРАЖДАНСКИЙ БРАК/ЖЕНАТ/ ЗАМУЖЕМ	Да	Офи&циальное	...	Нет	Нет	Нет
4	5	2013-01-03	2013-01-17	Мужской&*	22	Нет	Арендуемое	ГРАЖДАНСКИЙ БРАК/ЖЕНАТ/ ЗАМУЖЕМ	Нет	Официальное	...	Да	Да	Нет

С помощью метода `.str.replace()` можно поменять одно строковое значение на другое.

In[10]:
```
# меняем одно из строковых значений переменной marstatus
data['marstatus'] = data['marstatus'].str.replace('ХОЛОСТ', 'Не женат/не замужем')
data.head()
```

Out[10]:

	id	date_start	date_end	gender	age	auto	housing	marstatus	regclient	jobtype	...	credits	children	delinq60plus
0	1	2013-01-03	2013-01-12	Мужской&*	44	Нет	Собственное	ГРАЖДАНСКИЙ БРАК/ЖЕНАТ/ ЗАМУЖЕМ	Нет	_Официальное	...	Нет	Да	Нет
1	2	2013-01-03	2013-01-17	Мужской&*	21	Пропуск поля	Живут с родителями	Не женат/не замужем	Нет	_Официальное	...	Да	Нет	Нет
2	3	2013-01-03	2013-01-17	Мужской&*	25	Пропуск поля	Собственное	Не женат/не замужем	Да	_Официальное	...	Пропуск поля	Нет	Нет
3	4	2013-01-03	2013-01-17	Женский	47	Пропуск поля	Собственное	ГРАЖДАНСКИЙ БРАК/ЖЕНАТ/ ЗАМУЖЕМ	Да	Офи&циальное	...	Нет	Нет	Нет
4	5	2013-01-03	2013-01-17	Мужской&*	22	Нет	Арендуемое	ГРАЖДАНСКИЙ БРАК/ЖЕНАТ/ ЗАМУЖЕМ	Нет	Официальное	...	Да	Да	Нет

Изменение строкового значения

Мы видим, что строковые значения переменных *gender* и *jobtype* содержат лишние символы типа &*_, давайте удалим их с помощью метода `.str.replace()`.

In[11]:
```
# удаляем лишние символы из строковых значений
# переменных gender и jobtype
for i in ['gender', 'jobtype']:
    if i in data.columns:
        data[i] = data[i].str.replace('[*&_]', '')
data.head()
```

Out[11]:

	id	date_start	date_end	gender	age	auto	housing	marstatus	regclient	jobtype	...	credits	children	delinq60plus
0	1	2013-01-03	2013-01-12	Мужской	44	Нет	Собственное	ГРАЖДАНСКИЙ БРАК/ЖЕНАТ/ ЗАМУЖЕМ	Нет	Официальное	...	Нет	Да	Нет
1	2	2013-01-03	2013-01-17	Мужской	21	Пропуск поля	Живут с родителями	Не женат/не замужем	Нет	Официальное	...	Да	Нет	Нет
2	3	2013-01-03	2013-01-17	Мужской	25	Пропуск поля	Собственное	Не женат/не замужем	Да	Официальное	...	Пропуск поля	Нет	Нет
3	4	2013-01-03	2013-01-17	Женский	47	Пропуск поля	Собственное	ГРАЖДАНСКИЙ БРАК/ЖЕНАТ/ ЗАМУЖЕМ	Да	Официальное	...	Нет	Нет	Нет
4	5	2013-01-03	2013-01-17	Мужской	22	Нет	Арендуемое	ГРАЖДАНСКИЙ БРАК/ЖЕНАТ/ ЗАМУЖЕМ	Нет	Официальное	...	Да	Да	Нет

Определение пола клиента по отчеству

Часто при работе с базами данных бывает ситуация, когда у нас есть информация об именах, отчествах и фамилиях клиентов, но нет информации о поле. Давайте рассмотрим случай, когда по отчеству каждого клиента можно определить его пол.

In[12]:

```
# загружаем CSV файл с ФИО клиентов, по которым нужно определить пол
data3 = pd.read_csv('Data/Gender_based_on_middle_name.csv', encoding='cp1251', sep=';')
data3.head(20)
```

Out[12]:

	Клиент	Возраст	Регион	Статус
0	_Колесников Вячеслав Анатольевич	33	Красноярск- 2	Вернул(а)
1	_Саймурзанов Михаил Борисович	22	Красноярск- 2	Вернул(а)
2	Абаимов Максим Дмитриевич	43	Москва- 4	Вернул(а)
3	Абакумова Юлия Ивановна	22	Москва- 4	Вернул(а)
4	Абанова Елена Владимировна	54	Санкт-Петербург- 6	Вернул(а)
5	Абдрахимова Юлия Рафиковна	23	Санкт-Петербург- 6	Вернул(а)
6	Абдугалиева Айгуль Максутовна	27	Москва- 4	Не вернул(а)
7	Абдуллаев Ильгар Эльдарович	44	Екатеринбург- 8	Не вернул(а)
8	Абдуллин Евгений Эдуардович	22	Екатеринбург- 8	Не вернул(а)
9	Абдуллина Екатерина Анатольевна	63	Екатеринбург- 8	Вернул(а)
10	Абдуллина Екатерина Анатольевна	63	Екатеринбург- 8	Вернул(а)
11	Абдурасулова Наталья Таджиловна	55&	Екатеринбург- 8	Вернул(а)
12	Абдурахимова Алена Алимовна	57	Екатеринбург- 8	Вернул(а)
13	Абельдина Гульпархия Галимжановна	41	Екатеринбург- 8	Вернул(а)
14	Аблец Юлия Сергеевна	33	Екатеринбург- 8	Вернул(а)
15	Аболмасова Ирина Олеговна	38	Екатеринбург- 8	Вернул(а)
16	Абраев Нурлан Мусайбекович	49	Екатеринбург- 8	Вернул(а)
17	Абраменко Екатерина Владимировна	56	Москва- 4	Вернул(а)
18	Абрамов Дмитрий Владимирович	51	Москва- 4	Вернул(а)
19	Абрамов Никита Валерьевич	45лет	Екатеринбург- 8	Вернул(а)

Теперь мы создадим переменную *Пол*, которая будет иметь значение `True`, если строковое значение переменной *Клиент* содержит паттерн `вна` (Викторовна, Дмитриевна), и `False` в противном случае. Для этого воспользуемся методом `.str.contains()`.

In[13]:

```
# создаем переменную Пол, которая будет иметь значение True, если строковое значение
# переменной Клиент содержит паттерн "вна" (Викторовна, Дмитриевна),
# и False в противном случае
data3['Пол'] = data3['Клиент'].str.contains('вна')
data3.head(20)
```

Out[13]:

	Клиент	Возраст	Регион	Статус	Пол
0	_Колесников Вячеслав Анатольевич	33	Красноярск- 2	Вернул(а)	False
1	_Саймурзанов Михаил Борисович	22	Красноярск- 2	Вернул(а)	False
2	Абаимов Максим Дмитриевич	43	Москва- 4	Вернул(а)	False
3	Абакумова Юлия Ивановна	22	Москва- 4	Вернул(а)	True
4	Абанова Елена Владимировна	54	Санкт-Петербург- 6	Вернул(а)	True
5	Абдрахимова Юлия Рафиковна	23	Санкт-Петербург- 6	Вернул(а)	True
6	Абдугалиева Айгуль Максутовна	27	Москва- 4	Не вернул(а)	True
7	Абдуллаев Ильгар Эльдарович	44	Екатеринбург- 8	Не вернул(а)	False
8	Абдуллин Евгений Эдуардович	22	Екатеринбург- 8	Не вернул(а)	False
9	Абдуллина Екатерина Анатольевна	63	Екатеринбург- 8	Вернул(а)	True
10	Абдуллина Екатерина Анатольевна	63	Екатеринбург- 8	Вернул(а)	True
11	Абдурасулова Наталья Таджиловна	55&	Екатеринбург- 8	Вернул(а)	True
12	Абдурахимова Алена Алимовна	57	Екатеринбург- 8	Вернул(а)	True
13	Абельдина Гульпархия Галимжановна	41	Екатеринбург- 8	Вернул(а)	True
14	Аблец Юлия Сергеевна	33	Екатеринбург- 8	Вернул(а)	True
15	Аболмасова Ирина Олеговна	38	Екатеринбург- 8	Вернул(а)	True
16	Абраев Нурлан Мусайбекович	49	Екатеринбург- 8	Вернул(а)	False
17	Абраменко Екатерина Владимировна	56	Москва- 4	Вернул(а)	True
18	Абрамов Дмитрий Владимирович	51	Москва- 4	Вернул(а)	False
19	Абрамов Никита Валерьевич	45лет	Екатеринбург- 8	Вернул(а)	False

Теперь присвоим переменной *Пол* тип `str` и переименуем категории с помощью словаря.

In[14]:
```
# присвоим переменной Пол тип str
data3['Пол'] = data3['Пол'].astype('str')
```

In[15]:
```
# переименуем категории переменной Пол
d = {'False': 'Мужской', 'True': 'Женский'}
data3['Пол'] = data3['Пол'].map(d)
data3.head(20)
```

Out[15]:

	Клиент	Возраст	Регион	Статус	Пол
0	_Колесников Вячеслав Анатольевич	33	Красноярск- 2	Вернул(а)	Мужской
1	_Саймурзанов Михаил Борисович	22	Красноярск- 2	Вернул(а)	Мужской
2	Абаимов Максим Дмитриевич	43	Москва- 4	Вернул(а)	Мужской
3	Абакумова Юлия Ивановна	22	Москва- 4	Вернул(а)	Женский
4	Абанова Елена Владимировна	54	Санкт-Петербург- 6	Вернул(а)	Женский
5	Абдрахимова Юлия Рафиковна	23	Санкт-Петербург- 6	Вернул(а)	Женский
6	Абдугалиева Айгуль Максутовна	27	Москва- 4	Не вернул(а)	Женский
7	Абдуллаев Ильгар Эльдарович	44	Екатеринбург- 8	Не вернул(а)	Мужской
8	Абдуллин Евгений Эдуардович	22	Екатеринбург- 8	Не вернул(а)	Мужской
9	Абдуллина Екатерина Анатольевна	63	Екатеринбург- 8	Вернул(а)	Женский
10	Абдуллина Екатерина Анатольевна	63	Екатеринбург- 8	Вернул(а)	Женский
11	Абдурасулова Наталья Таджиловна	55&	Екатеринбург- 8	Вернул(а)	Женский
12	Абдурахимова Алена Алимовна	57	Екатеринбург- 8	Вернул(а)	Женский
13	Абельдина Гульпархия Галимжановна	41	Екатеринбург- 8	Вернул(а)	Женский
14	Аблец Юлия Сергеевна	33	Екатеринбург- 8	Вернул(а)	Женский
15	Аболмасова Ирина Олеговна	38	Екатеринбург- 8	Вернул(а)	Женский
16	Абраев Нурлан Мусайбекович	49	Екатеринбург- 8	Вернул(а)	Мужской
17	Абраменко Екатерина Владимировна	56	Москва- 4	Вернул(а)	Женский
18	Абрамов Дмитрий Владимирович	51	Москва- 4	Вернул(а)	Мужской
19	Абрамов Никита Валерьевич	45лет	Екатеринбург- 8	Вернул(а)	Мужской

Еще можно было извлечь последние три символа в каждом строковом значении переменной *Клиент* и затем на основе полученных значений создать новую переменную.

In[16]:

```
# извлекаем последние три символа в каждом строковом значении переменной Клиент
# и затем на основе полученных значений создаем новую переменную
data3['Пол2'] = data3['Клиент'].str[-3:]
# переименуем категории переменной Пол2
d = {'вич': 'Мужской', 'вна': 'Женский'}
data3['Пол2'] = data3['Пол2'].map(d)
data3.head(20)
```

Out[16]:

	Клиент	Возраст	Регион	Статус	Пол	Пол2
0	_Колесников Вячеслав Анатольевич	33	Красноярск- 2	Вернул(а)	Мужской	Мужской
1	_Саймурзанов Михаил Борисович	22	Красноярск- 2	Вернул(а)	Мужской	Мужской
2	Абаимов Максим Дмитриевич	43	Москва- 4	Вернул(а)	Мужской	Мужской
3	Абакумова Юлия Ивановна	22	Москва- 4	Вернул(а)	Женский	Женский
4	Абанова Елена Владимировна	54	Санкт-Петербург- 6	Вернул(а)	Женский	Женский
5	Абдрахимова Юлия Рафиковна	23	Санкт-Петербург- 6	Вернул(а)	Женский	Женский
6	Абдугалиева Айгуль Максутовна	27	Москва- 4	Не вернул(а)	Женский	Женский
7	Абдуллаев Ильгар Эльдарович	44	Екатеринбург- 8	Не вернул(а)	Мужской	Мужской
8	Абдуллин Евгений Эдуардович	22	Екатеринбург- 8	Не вернул(а)	Мужской	Мужской
9	Абдуллина Екатерина Анатольевна	63	Екатеринбург- 8	Вернул(а)	Женский	Женский
10	Абдуллина Екатерина Анатольевна	63	Екатеринбург- 8	Вернул(а)	Женский	Женский
11	Абдурасулова Наталья Таджиловна	55&	Екатеринбург- 8	Вернул(а)	Женский	Женский
12	Абдурахимова Алена Алимовна	57	Екатеринбург- 8	Вернул(а)	Женский	Женский
13	Абельдина Гульпархия Галимжановна	41	Екатеринбург- 8	Вернул(а)	Женский	Женский
14	Аблец Юлия Сергеевна	33	Екатеринбург- 8	Вернул(а)	Женский	Женский
15	Аболмасова Ирина Олеговна	38	Екатеринбург- 8	Вернул(а)	Женский	Женский
16	Абраев Нурлан Мусайбекович	49	Екатеринбург- 8	Вернул(а)	Мужской	Мужской
17	Абраменко Екатерина Владимировна	56	Москва- 4	Вернул(а)	Женский	Женский
18	Абрамов Дмитрий Владимирович	51	Москва- 4	Вернул(а)	Мужской	Мужской
19	Абрамов Никита Валерьевич	45лет	Екатеринбург- 8	Вернул(а)	Мужской	Мужской

Удаление лишних символов из строк

Теперь с помощью метода `.str.lstrip()` удалим ненужный символ подчеркивания, с которого начинаются несколько значений переменной *Клиент*. Метод `.str.lstrip()` возвращает копию указанной строки, с начала которой (т.е. слева l — left) устранены указанные символы.

In[17]:

```
# с помощью метода .str.lstrip() удалим ненужный символ
# подчеркивания, с которого начинаются несколько значений
# переменной Клиент, метод .str.lstrip() возвращает
# копию указанной строки, с начала (слева l – left)
# которой устранены указанные символы
data3['Клиент'] = data3['Клиент'].str.lstrip('_')
data3.head(20)
```

Out[17]:

	Клиент	Возраст	Регион	Статус	Пол	Пол2
0	Колесников Вячеслав Анатольевич	33	Красноярск- 2	Вернул(а)	Мужской	Мужской
1	Саймурзанов Михаил Борисович	22	Красноярск- 2	Вернул(а)	Мужской	Мужской
2	Абаимов Максим Дмитриевич	43	Москва- 4	Вернул(а)	Мужской	Мужской
3	Абакумова Юлия Ивановна	22	Москва- 4	Вернул(а)	Женский	Женский
4	Абанова Елена Владимировна	54	Санкт-Петербург- 6	Вернул(а)	Женский	Женский
5	Абдрахимова Юлия Рафиковна	23	Санкт-Петербург- 6	Вернул(а)	Женский	Женский
6	Абдугалиева Айгуль Максутовна	27	Москва- 4	Не вернул(а)	Женский	Женский
7	Абдуллаев Ильгар Эльдарович	44	Екатеринбург- 8	Не вернул(а)	Мужской	Мужской
8	Абдуллин Евгений Эдуардович	22	Екатеринбург- 8	Не вернул(а)	Мужской	Мужской
9	Абдуллина Екатерина Анатольевна	63	Екатеринбург- 8	Вернул(а)	Женский	Женский
10	Абдуллина Екатерина Анатольевна	63	Екатеринбург- 8	Вернул(а)	Женский	Женский
11	Абдурасулова Наталья Таджиловна	55&	Екатеринбург- 8	Вернул(а)	Женский	Женский
12	Абдурахимова Алена Алимовна	57	Екатеринбург- 8	Вернул(а)	Женский	Женский
13	Абельдина Гульпархия Галимжановна	41	Екатеринбург- 8	Вернул(а)	Женский	Женский
14	Аблец Юлия Сергеевна	33	Екатеринбург- 8	Вернул(а)	Женский	Женский
15	Аболмасова Ирина Олеговна	38	Екатеринбург- 8	Вернул(а)	Женский	Женский
16	Абраев Нурлан Мусайбекович	49	Екатеринбург- 8	Вернул(а)	Мужской	Мужской
17	Абраменко Екатерина Владимировна	56	Москва- 4	Вернул(а)	Женский	Женский
18	Абрамов Дмитрий Владимирович	51	Москва- 4	Вернул(а)	Мужской	Мужской
19	Абрамов Никита Валерьевич	45лет	Екатеринбург- 8	Вернул(а)	Мужской	Мужской

Теперь с помощью метода `.str.rstrip()` удалим ненужные символы, которыми заканчиваются некоторые значения переменной *Возраст*. Метод `.str.rstrip()` возвращает копию указанной строки, с конца которой (справа r — right) устранены указанные символы.

In[18]:

```
# с помощью метода .str.rstrip() удалим ненужные символы, которыми
# заканчиваются некоторые значения переменной Возраст, метод
# .str.rstrip() возвращает копию указанной строки, с конца
# (справа r – right) которой устранены указанные символы
data3['Возраст'] = data3['Возраст'].str.rstrip('&лет')
data3.head(20)
```

Out[18]:

	Клиент	Возраст	Регион	Статус	Пол	Пол2
0	Колесников Вячеслав Анатольевич	33	Красноярск- 2	Вернул(а)	Мужской	Мужской
1	Саймурзанов Михаил Борисович	22	Красноярск- 2	Вернул(а)	Мужской	Мужской
2	Абаимов Максим Дмитриевич	43	Москва- 4	Вернул(а)	Мужской	Мужской
3	Абакумова Юлия Ивановна	22	Москва- 4	Вернул(а)	Женский	Женский
4	Абанова Елена Владимировна	54	Санкт-Петербург- 6	Вернул(а)	Женский	Женский
5	Абдрахимова Юлия Рафиковна	23	Санкт-Петербург- 6	Вернул(а)	Женский	Женский
6	Абдугалиева Айгуль Максутовна	27	Москва- 4	Не вернул(а)	Женский	Женский
7	Абдуллаев Ильгар Эльдарович	44	Екатеринбург- 8	Не вернул(а)	Мужской	Мужской
8	Абдуллин Евгений Эдуардович	22	Екатеринбург- 8	Не вернул(а)	Мужской	Мужской
9	Абдуллина Екатерина Анатольевна	63	Екатеринбург- 8	Вернул(а)	Женский	Женский
10	Абдуллина Екатерина Анатольевна	63	Екатеринбург- 8	Вернул(а)	Женский	Женский
11	Абдурасулова Наталья Таджиловна	55	Екатеринбург- 8	Вернул(а)	Женский	Женский
12	Абдурахимова Алена Алимовна	57	Екатеринбург- 8	Вернул(а)	Женский	Женский
13	Абельдина Гульпархия Галимжановна	41	Екатеринбург- 8	Вернул(а)	Женский	Женский
14	Аблец Юлия Сергеевна	33	Екатеринбург- 8	Вернул(а)	Женский	Женский
15	Аболмасова Ирина Олеговна	38	Екатеринбург- 8	Вернул(а)	Женский	Женский
16	Абраев Нурлан Мусайбекович	49	Екатеринбург- 8	Вернул(а)	Мужской	Мужской
17	Абраменко Екатерина Владимировна	56	Москва- 4	Вернул(а)	Женский	Женский
18	Абрамов Дмитрий Владимирович	51	Москва- 4	Вернул(а)	Мужской	Мужской
19	Абрамов Никита Валерьевич	45	Екатеринбург- 8	Вернул(а)	Мужской	Мужской

Теперь удалим последние 3 символа в каждом строковом значении переменной *Регион*.

In[19]:

```
# удаляем последние 3 символа в каждом строковом
# значении переменной Регион
data3['Регион'] = data3['Регион'].map(lambda x: str(x)[:-3])
data3.head(20)
```

Out[19]:

	Клиент	Возраст	Регион	Статус	Пол	Пол2
0	Колесников Вячеслав Анатольевич	33	Красноярск	Вернул(а)	Мужской	Мужской
1	Саймурзанов Михаил Борисович	22	Красноярск	Вернул(а)	Мужской	Мужской
2	Абаимов Максим Дмитриевич	43	Москва	Вернул(а)	Мужской	Мужской
3	Абакумова Юлия Ивановна	22	Москва	Вернул(а)	Женский	Женский
4	Абанова Елена Владимировна	54	Санкт-Петербург	Вернул(а)	Женский	Женский
5	Абдрахимова Юлия Рафиковна	23	Санкт-Петербург	Вернул(а)	Женский	Женский
6	Абдугалиева Айгуль Максутовна	27	Москва	Не вернул(а)	Женский	Женский
7	Абдуллаев Ильгар Эльдарович	44	Екатеринбург	Не вернул(а)	Мужской	Мужской
8	Абдуллин Евгений Эдуардович	22	Екатеринбург	Не вернул(а)	Мужской	Мужской
9	Абдуллина Екатерина Анатольевна	63	Екатеринбург	Вернул(а)	Женский	Женский
10	Абдуллина Екатерина Анатольевна	63	Екатеринбург	Вернул(а)	Женский	Женский
11	Абдурасулова Наталья Таджиловна	55	Екатеринбург	Вернул(а)	Женский	Женский
12	Абдурахимова Алена Алимовна	57	Екатеринбург	Вернул(а)	Женский	Женский
13	Абельдина Гульпархия Галимжановна	41	Екатеринбург	Вернул(а)	Женский	Женский
14	Аблец Юлия Сергеевна	33	Екатеринбург	Вернул(а)	Женский	Женский
15	Аболмасова Ирина Олеговна	38	Екатеринбург	Вернул(а)	Женский	Женский
16	Абраев Нурлан Мусайбекович	49	Екатеринбург	Вернул(а)	Мужской	Мужской
17	Абраменко Екатерина Владимировна	56	Москва	Вернул(а)	Женский	Женский
18	Абрамов Дмитрий Владимирович	51	Москва	Вернул(а)	Мужской	Мужской
19	Абрамов Никита Валерьевич	45	Екатеринбург	Вернул(а)	Мужской	Мужской

Удаление повторяющихся строк

Если внимательно взглянуть на данные, можно увидеть, что некоторые наблюдения дублируются (например, два раза повторяется информация по клиенту `Абдуллина Екатерина Анатольевна`).

8	Абдуллин Евгений Эдуардович	22	Екатеринбург	Не вернул(а)	Мужской	Мужской
9	Абдуллина Екатерина Анатольевна	63	Екатеринбург	Вернул(а)	Женский	Женский
10	Абдуллина Екатерина Анатольевна	63	Екатеринбург	Вернул(а)	Женский	Женский
11	Абдурасулова Наталья Таджиловна	55	Екатеринбург	Вернул(а)	Женский	Женский
12	Абдурахимова Алена Алимовна	57	Екатеринбург	Вернул(а)	Женский	Женский
13	Абельдина Гульпархия Галимжановна	41	Екатеринбург	Вернул(а)	Женский	Женский

Давайте с помощью метода `.drop_duplicates()` удалим дублирующиеся строки.

In[20]:

```
# удаляем дублирующиеся строки
data3 = data3.drop_duplicates()
data3.head(20)
```

Out[20]:

	Клиент	Возраст	Регион	Статус	Пол	Пол2
0	Колесников Вячеслав Анатольевич	33	Красноярск	Вернул(а)	Мужской	Мужской
1	Саймурзанов Михаил Борисович	22	Красноярск	Вернул(а)	Мужской	Мужской
2	Абаимов Максим Дмитриевич	43	Москва	Вернул(а)	Мужской	Мужской
3	Абакумова Юлия Ивановна	22	Москва	Вернул(а)	Женский	Женский
4	Абанова Елена Владимировна	54	Санкт-Петербург	Вернул(а)	Женский	Женский
5	Абдрахимова Юлия Рафиковна	23	Санкт-Петербург	Вернул(а)	Женский	Женский
6	Абдугалиева Айгуль Максутовна	27	Москва	Не вернул(а)	Женский	Женский
7	Абдуллаев Ильгар Эльдарович	44	Екатеринбург	Не вернул(а)	Мужской	Мужской
8	Абдуллин Евгений Эдуардович	22	Екатеринбург	Не вернул(а)	Мужской	Мужской
9	Абдуллина Екатерина Анатольевна	63	Екатеринбург	Вернул(а)	Женский	Женский
11	Абдурасулова Наталья Таджиловна	55	Екатеринбург	Вернул(а)	Женский	Женский
12	Абдурахимова Алена Алимовна	57	Екатеринбург	Вернул(а)	Женский	Женский
13	Абельдина Гульпархия Галимжановна	41	Екатеринбург	Вернул(а)	Женский	Женский
14	Аблец Юлия Сергеевна	33	Екатеринбург	Вернул(а)	Женский	Женский
15	Аболмасова Ирина Олеговна	38	Екатеринбург	Вернул(а)	Женский	Женский
16	Абраев Нурлан Мусайбекович	49	Екатеринбург	Вернул(а)	Мужской	Мужской
17	Абраменко Екатерина Владимировна	56	Москва	Вернул(а)	Женский	Женский
18	Абрамов Дмитрий Владимирович	51	Москва	Вернул(а)	Мужской	Мужской
19	Абрамов Никита Валерьевич	45	Екатеринбург	Вернул(а)	Мужской	Мужской
20	Абрамов Сергей Сергеевич	32	Москва	Вернул(а)	Мужской	Мужской

Извлечение нужных символов из строк

Часто данные могут быть некорректно записаны, например, несколько полей могут быть записаны в одно, и необходимо извлечь их. Давайте загрузим данные.

In[21]:

```
# загружаем данные
data4 = pd.read_csv('Data/Raw_text.csv', encoding='cp1251')
data4
```

Out[21]:

	raw
0	KDR 1 2014-12-23 3242.0
1	MSK 1 2010-02-23 3453.7
2	KRSK 0 2014-06-20 2123.0
3	SPB 0 2014-03-14 1123.6
4	EKB 1 2013-01-15 2134.0

Видим, что несколько переменных записаны в один столбец. Давайте с помощью метода `.str.extract()` извлечем даты, создав переменную `date`.

In[22]:
```
# с помощью метода .str.extract() извлекаем даты из столбца raw,
# создав переменную date
data4['date'] = data4['raw'].str.extract('(....-..-..)', expand=True)
data4
```

Out[22]:

	raw	date
0	KDR 1 2014-12-23 3242.0	2014-12-23
1	MSK 1 2010-02-23 3453.7	2010-02-23
2	KRSK 0 2014-06-20 2123.0	2014-06-20
3	SPB 0 2014-03-14 1123.6	2014-03-14
4	EKB 1 2013-01-15 2134.0	2013-01-15

Извлекаем одиночные цифры из столбца `raw`, создав переменную *gender*.

In[23]:
```
# извлекаем одиночные цифры из столбца raw, создав переменную gender
data4['gender'] = data4['raw'].str.extract('(\d)', expand=True)
data4
```

Out[23]:

	raw	date	gender
0	KDR 1 2014-12-23 3242.0	2014-12-23	1
1	MSK 1 2010-02-23 3453.7	2010-02-23	1
2	KRSK 0 2014-06-20 2123.0	2014-06-20	0
3	SPB 0 2014-03-14 1123.6	2014-03-14	0
4	EKB 1 2013-01-15 2134.0	2013-01-15	1

Извлекаем числа с плавающей точкой из столбца `raw`, создав переменную *score*.

In[24]:
```
# извлекаем числа с плавающей точкой из столбца raw,
# создав переменную score
data4['score'] = data4['raw'].str.extract('(\d\d\d\d\.\d)', expand=True)
data4
```

Out[24]:

	raw	date	gender	score
0	KDR 1 2014-12-23 3242.0	2014-12-23	1	3242.0
1	MSK 1 2010-02-23 3453.7	2010-02-23	1	3453.7
2	KRSK 0 2014-06-20 2123.0	2014-06-20	0	2123.0
3	SPB 0 2014-03-14 1123.6	2014-03-14	0	1123.6
4	EKB 1 2013-01-15 2134.0	2013-01-15	1	2134.0

Наконец, извлекаем текст из столбца `raw`, создав переменную *city*.

In[25]:
```
# извлекаем текст из столбца raw, создав переменную city
data4['city'] = data4['raw'].str.extract('([A-Z]\w{0,})', expand=True)
data4
```

Out[25]:

	raw	date	gender	score	city
0	KDR 1 2014-12-23 3242.0	2014-12-23	1	3242.0	KDR
1	MSK 1 2010-02-23 3453.7	2010-02-23	1	3453.7	MSK
2	KRSK 0 2014-06-20 2123.0	2014-06-20	0	2123.0	KRSK
3	SPB 0 2014-03-14 1123.6	2014-03-14	0	1123.6	SPB
4	EKB 1 2013-01-15 2134.0	2013-01-15	1	2134.0	EKB

ПРИЛОЖЕНИЕ 7. РАБОТА С ПРЕДУПРЕЖДЕНИЕМ SETTINGWITHCOPYWARNING В БИБЛИОТЕКЕ PANDAS

Материал подготовлен веб-разработчиком на Python Бенджамином Прайком.

`SettingWithCopyWarning` является одним из наиболее распространенных предупреждений, с которыми сталкиваются пользователи при изучении библиотеки pandas. Неудивительно, что многие по-разному реагируют на него. Существует масса способов индексирования структур данных в библиотеке pandas, каждый со своим особым нюансом и даже сама библиотека pandas не гарантирует единого результата для двух строк кода, выглядящих одинаково.

В этом руководстве объясняются причины срабатывания данного предупреждения и демонстрируются способы, как его избежать. Кроме того, здесь мы рассмотрим детали того, что совершается под капотом. Это даст вам лучшее представление о том, что именно происходит, и поможет вам понять, почему все это работает именно так.

Для исследования проблемы срабатывания предупреждения `SettingWithCopyWarning` мы воспользуемся файлом *Auctions.csv*, представляющим собой данные аукционов Ebay. Данные содержат следующие переменные:

- *auctionid* – уникальный идентификатор аукциона;
- *bid* – значение ставки;
- *bidtime* – срок аукциона на момент ставки (в днях);
- *bidder* – пользовательское имя покупателя;
- *bidderrate* – пользовательский рейтинг покупателя;
- *openbid* – начальная ставка, оглашенная продавцом;
- *price* – победившая ставка, закрывающая аукцион.

Давайте взглянем на них:

```
In[1]:
# импортируем библиотеку pandas
import pandas as pd
# загружаем данные
data = pd.read_csv('Data/Auctions.csv')
# выводим первые 5 наблюдений
data.head()
```

Out[1]:

	auctionid	bid	bidtime	bidder	bidderrate	openbid	price
0	8213034705	95.0	2.927373	jake7870	0	95.0	117.5
1	8213034705	115.0	2.943484	davidbresler2	1	95.0	117.5
2	8213034705	100.0	2.951285	gladimacowgirl	58	95.0	117.5
3	8213034705	117.5	2.998947	daysrus	10	95.0	117.5
4	8213060420	2.0	0.065266	donnie4814	5	1.0	120.0

Что представляет из себя предупреждение SettingWithCopyWarning?

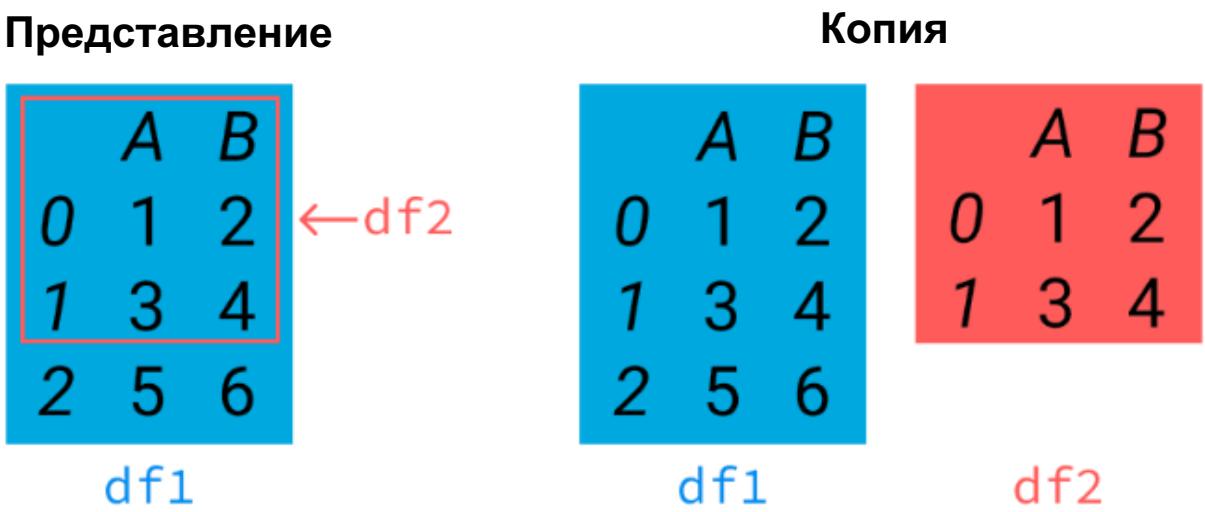
Первое, что нужно понять – это то, что `SettingWithCopyWarning` является предупреждением, а не ошибкой.

Ошибка говорит нам «что-то пошло не так», например, недопустимый синтаксис или попытка ссылаться на переменную, которая была не определена заранее. Предупреждение же служит цели просигнализировать программисту о возможных ошибках или проблемах с кодом, которые, однако, позволяют ему выполнять те или иные операции под капотом. В данном случае предупреждение скорее всего указывает на серьезную, но незаметную ошибку.

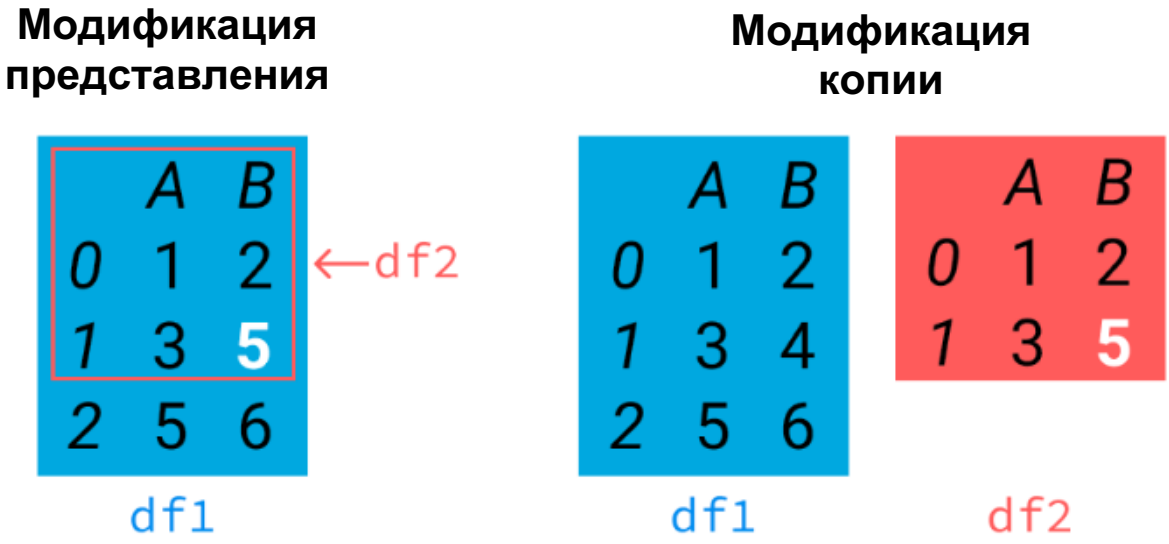
`SettingWithCopyWarning` информирует вас о том, что ваша операция, возможно, не сработала должным образом, поэтому вы должны проверить результат и убедиться, что не допустили ошибку.

Может возникнуть соблазн игнорировать предупреждение, если ваш код все еще работает так, как ожидалось. Это плохая практика, и `SettingWithCopyWarning` никогда не следует игнорировать. Потратьте некоторое время, чтобы понять, почему вы получаете это предупреждение, прежде чем предпринимать действия.

Чтобы прояснить для себя суть предупреждения `SettingWithCopyWarning`, полезно понять, что в библиотеке pandas одни операции могут возвращать представление ваших данных, а другие возвращают копию.



Как видно на рисунке выше, представление `df2` слева – это просто подмножество исходного датафрейма `df1`, тогда как копия справа создает новый, уникальный объект `df2`.
Это может вызвать проблемы при попытке внести изменения:



В зависимости от наших задач нам может потребоваться изменение исходного датафрейма `df1` (слева) или же изменение только датафрейма `df2` (справа). Предупреждение сообщает нам, что наш код, возможно, выполнил совершенно не то, что нам было нужно.
Мы вернемся к этому моменту позже, а пока давайте рассмотрим две основные причины выдачи предупреждения и подумаем, как их устранить.

Присваивание по цепочке (chained assignment)

Библиотека pandas генерирует предупреждение, когда обнаруживает так называемое присваивание по цепочке (chained assignment). Давайте

определим несколько терминов, которые будем в дальнейшем использовать:

- присваивание (assignment) – операции, которые устанавливают значение чего-либо, например, `data = pd.read_csv('Data/Auctions.csv')`, часто их называют операциями `set` (от англ. set - установить);
- доступ (access) – операции, возвращающие значение чего-то, в частности, примеры индексации и цепочки, приведенные ниже, часто такие операции называют операциями `get` (от англ. get - получить);
- индексирование (indexing) – любой метод присваивания или доступа, который ссылается на подмножество данных: например, `data[1:5]`;
- сцепление (chaining) – использование операций индексирования одной за другой: например, `data[1:5][1:3]`.

Присваивание по цепочке – это комбинация цепочки и присваивания. Давайте быстро рассмотрим пример с набором данных, который мы загрузили ранее. Предположим, что нам сказали, что значение рейтинга пользователя `parakeet2004` ошибочно, и мы должны его обновить. Посмотрим на текущие значения.

In[2]:
```
# смотрим значения данных для пользователя parakeet2004
data[data.bidder == 'parakeet2004']
```

Out[2]:

	auctionid	bid	bidtime	bidder	bidderrate	openbid	price
6	8213060420	3.00	0.186539	parakeet2004	5	1.0	120.0
7	8213060420	10.00	0.186690	parakeet2004	5	1.0	120.0
8	8213060420	24.99	0.187049	parakeet2004	5	1.0	120.0

У нас есть три наблюдения, в которых нужно обновить значения переменной *bidderrate*.

In[3]:
```
# обновляем значения переменной bidderrate
# для пользователя parakeet2004
data[data.bidder == 'parakeet2004']['bidderrate'] = 100
```

Out[3]:
```
C:\Anaconda3\lib\site-packages\ipykernel_launcher.py:3: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-
docs/stable/indexing.html#indexing-view-versus-copy
```

О, нет! Мы мистическим образом столкнулись с предупреждением `SetWithCopyWarning`! Если посмотрим на наши значения, то увидим, что в данном случае они не изменились:

In[4]:
```
# вновь смотрим значения данных для пользователя parakeet2004
data[data.bidder == 'parakeet2004']
```

Out[4]:

	auctionid	bid	bidtime	bidder	bidderrate	openbid	price
6	8213060420	3.00	0.186539	parakeet2004	5	1.0	120.0
7	8213060420	10.00	0.186690	parakeet2004	5	1.0	120.0
8	8213060420	24.99	0.187049	parakeet2004	5	1.0	120.0

Предупреждение было сформировано по причине того, что мы связали вместе две операции индексирования. Это легко обнаружить, ведь мы дважды использовали квадратные скобки, но то же самое было бы верно, если бы мы использовали другие методы доступа, такие как `.bidderrate`, `.loc[]`, `.iloc[]`, `.ix[]` и т.д. Наши операции в цепочке выглядят так:

- `data[data.bidder == 'parakeet2004']`
- `['tendrate'] = 100`

Эти операции в цепочке выполняются независимо друг от друга, одна за другой. Первая операция – это метод доступа (операция `get`), который возвращает объект `DataFrame`, содержащий все строки, в которых переменная *bidder* принимает значение `'parakeet2004'`. Вторая операция – операция присваивания (операция `set`), которая совершается над этим новым объектом `DataFrame`. Получается, что мы вообще не работаем с исходным объектом `DataFrame`. Собственно сообщение `A value is trying to be set on a copy of a slice from a DataFrame` об этом нас и предупреждает: происходит попытка изменить значение в копии среза данных датафрейма, но это не изменит исходный датафрейм.

Решение будет простым: объединяем эти две операции в одну операцию, используя свойство `.loc`, чтобы pandas могла убедиться, что задан исходный объект `DataFrame`. Библиотека pandas всегда будет следить за тем, чтобы операции `set` использовались без цепочек.

In[5]:
```
# задаем новое значение
data.loc[data.bidder == 'parakeet2004', 'bidderrate'] = 100
# смотрим результат
data[data.bidder == 'parakeet2004']['bidderrate']
```

Out[5]:
```
6    100
7    100
8    100
Name: bidderrate, dtype: int64
```

В данном случае все прекрасно сработало.

Скрытая цепочка

Переходим ко второй наиболее распространенной причине выдачи предупреждения `SettingWithCopyWarning`. Давайте исследуем выигравшие ставки. Мы создадим новый объект `DataFrame` для работы с ними, заранее позаботившись о том, чтобы использовать свойство `.loc`.

In[6]:
```
# создаем новый датафрейм, в котором ставка
# совпадает с конечной ценой
winners = data.loc[data.bid == data.price]
winners.head()
```

Out[6]:

	auctionid	bid	bidtime	bidder	bidderrate	openbid	price
3	8213034705	117.5	2.998947	daysrus	10	95.00	117.5
25	8213060420	120.0	2.999722	djnoeproductions	17	1.00	120.0
44	8213067838	132.5	2.996632	*champaignbubbles*	202	29.99	132.5
45	8213067838	132.5	2.997789	*champaignbubbles*	202	29.99	132.5
66	8213073509	114.5	2.999236	rr6kids	4	1.00	114.5

Мы могли бы написать несколько следующих строк программного кода, чтобы получить больше информации о наших победителях.

In[7]:
```
mean_win_time = winners.bidtime.mean()
mode_open_bid = winners.openbid.mode()
```

Случайно мы сталкиваемся с другой ошибкой в нашем датафрейме. На этот раз в наблюдении с меткой индекса `304` у переменной *bidder* отсутствует значение.

In[8]:
```
# смотрим значение переменной bidder для
# наблюдения с меткой индекса 304
winners.loc[304, 'bidder']
```

Out[8]:
```
nan
```

Предположим, что мы знаем истинное имя пользователя и вносим изменения.

In[9]:
```
# присваиваем значение переменной bidder
# в наблюдении с меткой индекса 304
winners.loc[304, 'bidder'] = 'therealname'
```

Out[9]:
```
C:\Anaconda3\lib\site-packages\ipykernel_launcher.py:3: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-
docs/stable/indexing.html#indexing-view-versus-copy
```

Еще одно предупреждение `SettingWithCopyWarning`! Но ведь мы использовали свойство `.loc`, почему же это произошло снова? Чтобы понять причину, давайте посмотрим на результат выполнения нашего программного кода:

In[10]:
```
# печатаем значение переменной bidder
# в наблюдении с меткой индекса 304
print(winners.loc[304, 'bidder'])
```

```
Out[10]:
therealname
```

На этот раз программный код сработал, так почему же мы получили предупреждение?
Поскольку датафрейм `winners` был создан как результат операции `get (data.loc[data.bid == data.price])`, он может быть копией исходного объекта `DataFrame`, а может и не быть таковой, но пока мы не проверим, мы этого не узнаем! Когда мы индексировали датафрейм `winners`, мы по сути использовали индексирование по цепочке.
Это означает, что мы, возможно, дополнительно модифицировали датафрейм `data`, когда пытались модифицировать датафрейм `winners`.
В реальности эти строки могут находиться на большом расстоянии друг от друга, поэтому отследить источник проблемы будет сложнее, но ситуация будет аналогичная.
Чтобы предотвратить предупреждение в данном случае, нам нужно явно указать библиотеке pandas, что мы выполняем копирование при создании нового датафрейма:

```
In[11]:
# создаем копию датафрейма, в котором ставка
# совпадает с конечной ценой
winners = data.loc[data.bid == data.price].copy()
# присваиваем значение переменной bidder
# в наблюдении с меткой индекса 304
winners.loc[304, 'bidder'] = 'therealname'
# печатаем значение переменной bidder
# в наблюдении с меткой индекса 304
# в новом датафрейме
print(winners.loc[304, 'bidder'])
# печатаем значение переменной bidder
# в наблюдении с меткой индекса 304
# в исходном датафрейме
print(data.loc[304, 'bidder'])

Out[11]:
therealname
nan
```

Вот и все! Так просто.

Хитрость заключается в том, чтобы научиться выявлять индексирование по цепочке и избегать его любой ценой. Если вы хотите изменить исходный файл, используйте одну операцию присваивания. Если вы хотите получить копию, убедитесь, что вы передаете библиотеке pandas именно эту инструкцию. Это позволит сэкономить время и сделать ваш программный код более надежным.

Советы и рекомендации по работе с предупреждением SettingWithCopyWarning

Теперь подробнее рассмотрим настройки предупреждения `SettingWithCopyWarning`.

Отключение предупреждения

Опция библиотеки `pandas` `mode.chained_assignment` может принимать одно из значений:

- `'raise'` – генерирует исключение вместо предупреждения;
- `'warn'` - генерирует предупреждение (по умолчанию);
- `'None'` – полностью отключает предупреждение.

Например, давайте отключим предупреждение:

```
In[12]:
# отключаем предупреждение SettingWithCopyWarning
pd.set_option('mode.chained_assignment', None)
# задаем новое значение
data[data.bidder == 'parakeet2004']['bidderrate'] = 100
```

Поскольку такой вариант не дает нам никакого предупреждения, он не рекомендуется в тех случаях, когда у вас нет полного понимания того, что вы делаете. Если у вас есть даже мельчайшие сомнения в сомнении, не используйте отключение. Некоторые разработчики очень серьезно относятся к `SettingWithCopyWarning` и предпочитают вместо этого выбирать вариант `'raise'`:

```
In[13]:
# будем генерировать исключение вместо предупреждения
pd.set_option('mode.chained_assignment', 'raise')
# задаем новое значение
data[data.bidder == 'parakeet2004']['bidderrate'] = 100

Out[13]:
---------------------------------------------------------------------------
SettingWithCopyError                      Traceback (most recent call last)
<ipython-input-13-dd83e6d82f77> in <module>()
      2 pd.set_option('mode.chained_assignment', 'raise')
      3 # задаем новое значение
----> 4 data[data.bidder == 'parakeet2004']['bidderrate'] = 100

C:\Anaconda3\lib\site-packages\pandas\core\frame.py in __setitem__(self, key, value)
   3117         else:
   3118             # set column
-> 3119             self._set_item(key, value)
   3120
   3121     def _setitem_slice(self, key, value):

C:\Anaconda3\lib\site-packages\pandas\core\frame.py in _set_item(self, key, value)
   3199         # value exception to occur first
   3200         if len(self):
-> 3201             self._check_setitem_copy()
   3202
   3203     def insert(self, loc, column, value, allow_duplicates=False):

C:\Anaconda3\lib\site-packages\pandas\core\generic.py in _check_setitem_copy(self,
stacklevel, t, force)
   2710
   2711             if value == 'raise':
-> 2712                 raise com.SettingWithCopyError(t)
   2713             elif value == 'warn':
   2714                 warnings.warn(t, com.SettingWithCopyWarning,

SettingWithCopyError:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-
docs/stable/indexing.html#indexing-view-versus-copy
```

Это может быть особенно полезным, если вы работаете над проектом с неопытными разработчиками библиотеки pandas в своей команде или над проектом со строгими требованиями к оформлению программного кода.
Более аккуратный способ использования этого параметра – воспользоваться диспетчером контекста.

In[14]:
```
# сбрасываем опцию, заданную в предыдущем
# блоке программного кода
pd.reset_option('mode.chained_assignment')
# отключаем предупреждение, воспользовавшись
# диспетчером контекста
with pd.option_context('mode.chained_assignment', None):
    data[data.bidder == 'parakeet2004']['bidderrate'] = 100
```

Как видно, такой подход обеспечивает точечное подавление предупреждений, не применяя его ко всему окружению.

Однотипные и многотипные объекты

Еще один момент, который стоит подчеркнуть, - это различие между однотипными и многотипными объектами. Объект `DataFrame` является однотипным, если все его столбцы имеют один и тот же тип, например:

In[15]:
```
# импортируем библиотеку numpy
import numpy as np
# создаем однотипный объект DataFrame
single_dtype_df = pd.DataFrame(np.random.rand(5,2), columns=list('AB'))
# печатаем информацию о типах столбцов
print(single_dtype_df.dtypes)
# выводим данные
single_dtype_df
```

Out[15]:
```
A    float64
B    float64
dtype: object
```

	A	B
0	0.445394	0.062785
1	0.980684	0.967140
2	0.504585	0.491010
3	0.112733	0.699670
4	0.179244	0.717076

И, наоборот, объект `DataFrame` является многотипным, если не все его столбцы имеют одинаковый тип, например:

In[16]:
```
# создаем многотипный объект DataFrame
multiple_dtype_df = pd.DataFrame({'A': np.random.rand(5),'B': list('abcde')})
# печатаем информацию о типах столбцов
print(multiple_dtype_df.dtypes)
# выводим данные
multiple_dtype_df
```

Out[16]:

```
A    float64
B     object
dtype: object
```

	A	B
0	0.689768	a
1	0.503154	b
2	0.187169	c
3	0.953120	d
4	0.659795	e

По некоторым историческим причинам операция обращения по индексу, совершаемая над многотипным объектом, всегда будет возвращать копию. Однако операция обращения по индексу, совершаемая над однотипным объектом, почти всегда возвращает представление, оговорка здесь заключается в том, что все зависит от расположении объекта в памяти, которое не гарантировано.

Ложные срабатывания

Ложные срабатывания или ситуации, когда необоснованно сообщалось о присваивании по цепочке, часто встречались в ранних версиях библиотеки pandas, но с тех пор эти ситуации в основном были исправлены. Для полноты картины полезно рассмотреть некоторые примеры исправленных ложных срабатываний. Если вы столкнулись с ситуациями, описанными ниже в более ранних версиях библиотеки pandas, тогда предупреждение можно смело игнорировать (или вообще избежать его появления, обновившись).

Добавление нового столбца в объект `DataFrame` с использованием значений текущего столбца, как правило, выдавало предупреждение, но это было исправлено.

In[17]:

```
# добавляем новый столбец bidtime_hours
# на основе значений столбца bidtime
data['bidtime_hours'] = data.bidtime.map(lambda x: x * 24)
data.head(2)
```

Out[17]:

	auctionid	bid	bidtime	bidder	bidderrate	openbid	price	bidtime_hours
0	8213034705	95.0	2.927373	jake7870	0	95.0	117.5	70.256952
1	8213034705	115.0	2.943484	davidbresler2	1	95.0	117.5	70.643616

До недавнего времени ложные срабатывания также возникали, когда значения задавались с помощью метода `.apply()` среза объекта `DataFrame`, хотя сейчас это тоже исправлено.

In[18]:
```
# добавляем новый столбец bidtime_hours
# на основе значений столбца bidtime
data.loc[:, 'bidtime_hours'] = data.bidtime.apply(lambda x: x * 24)
data.head(2)
```

Out[18]:

	auctionid	bid	bidtime	bidder	bidderrate	openbid	price	bidtime_hours
0	8213034705	95.0	2.927373	jake7870	0	95.0	117.5	70.256952
1	8213034705	115.0	2.943484	davidbresler2	1	95.0	117.5	70.643616

И, наконец, вплоть до версии 0.17.0 был баг в методе `.sample()` объекта `DataFrame`, который вызывал ложное срабатывание `SettingWithCopy`. Теперь метод `.sample()` возвращает копию.

In[19]:
```
# случайным образом отбираем 2 наблюдения
# из исходного датафрейма
sample = data.sample(2, random_state=200)
# задаем в этих двух наблюдениях значение
# переменной price равным 120
sample.loc[:, 'price'] = 120
# выводим наблюдения
sample.head()
```

Out[19]:

	auctionid	bid	bidtime	bidder	bidderrate	openbid	price	bidtime_hours
467	8215408023	30.0	1.739572	bigreggie101	0	0.99	120	41.749728
233	8213387659	40.0	2.016701	mniffin01	0	9.99	120	48.400824

Подробнее о присваивании по цепочке

Давайте снова воспользуемся нашим предыдущим примером, в котором мы пытались изменить значение переменной *bidderrate* в тех наблюдениях, где переменная *bidder* принимает значение `'parakeet2004'`.

In[20]:
```
# вернемся к предыдущему примеру
data[data.bidder == 'parakeet2004']['bidderrate'] = 100
```

Своим предупреждением `SettingWithCopyWarning` библиотека pandas сообщает, что поведение нашего кода неоднозначно, но чтобы понять причину его появления, полезно будет рассмотреть несколько принципов.

Ранее мы уже кратко упомянули о представлении и копии. Существует два возможных способа доступа к подмножеству объекта `DataFrame`: можно создать ссылку на исходные данные в памяти (представление) или скопировать подмножество в новый, меньший по размеру объект `DataFrame` (копия). Представление – это способ взглянуть на определенную часть **исходных** данных, тогда как копия – это **клон** этих данных в новой области памяти. Как показал наш рисунок, при

модификации представления произойдет модификация исходной переменной, а вот при модификации копии этого не произойдет.
По причинам, которые мы рассмотрим позже, однозначный результат операций `get` в библиотеке pandas не гарантирован. Когда вы индексируете структуры данных библиотеки pandas, может быть возвращено либо представление, либо копия. Это означает, что операции `get`, совершаемые над объектом `DataFrame`, возвращают новый объект `DataFrame`, который может содержать:

- копию данных, содержащихся в исходном объекте;
- ссылку на данные исходного объекта без выполнения копирования.

Поскольку мы не знаем, что произойдет, и каждый случай будет уникальным, игнорировать предупреждение – это все равно что играть с огнем.
Чтобы более четко проиллюстрировать разницу между представлением и копией, возникающую неоднозначность, давайте создадим простой объект `DataFrame` и проиндексируем его:

```
In[21]:
# создаем новый объект DataFrame
df1 = pd.DataFrame(np.arange(6).reshape((3,2)), columns=list('AB'))
df1
```

```
Out[21]:
```

	A	B
0	0	1
1	2	3
2	4	5

И давайте присвоим подмножество датафрейма `df1` датафрейму `df2`:

```
In[22]:
# присваиваем подмножества датафрейма df1
# датафрейму df2
df2 = df1.loc[:1]
df2
```

```
Out[22]:
```

	A	B
0	0	1
1	2	3

Учитывая все вышесказанное, мы уже знаем, что датафрейм `df2` может быть представлением датафрейма `df1` или копией подмножества `df1`.
Прежде чем мы начнем решать нашу проблему, нам нужно еще раз взглянуть на индексирование по цепочке. Ниже мы объединили две операции индексирования:

```
__intermediate__ = data[data.bidder == 'parakeet2004']
__intermediate__['bidderrate'] = 100
```

Здесь `__intermediate__` представляет собой результат первого вызова и полностью скрыт от нас. Помните, что мы получили бы тот же результат, если бы использовали атрибутивный доступ:

```
data[data.bidder == 'parakeet2004'].bidderrate = 100
```

То же самое относится к любой другой форме вызова по цепочке, потому что **мы генерируем этот промежуточный объект**.
Под капотом индексирование по цепочке обозначает создание более одного вызова `__getitem__` или `__setitem__` для выполнения одной операции. Это специальные встроенные методы Python, пример того, что называется синтаксическим сахаром. Метод `__getitem__` отвечает за получение элемента по индексу или ключу. Метод `__setitem__` отвечает за присваивание элемента с данным ключом или индексом. Давайте посмотрим, какие действия совершает интерпретатор Python применительно к нашему примеру.

```
# наш код
data[data.bidder == 'parakeet2004']['bidderrate'] = 100

# а вот так происходит выполнение кода
data.__getitem__(data.__getitem__('bidder') == 'parakeet2004').__setitem__('bidderrate', 100)
```

Как вы, возможно, уже поняли, `SettingWithCopyWarning` генерируется в результате вызова метода `__setitem__` по цепочке. Вы можете попробовать самостоятельно – две строки программного кода, приведенные выше, одинаковы. Для ясности обратите внимание, что второй вызов метода `__getitem__` (для столбца `bidder`) является вложенным и вовсе не относится к проблеме вызова по цепочке.
В целом, как уже обсуждалось, библиотека pandas не гарантирует, вернет ли операция `get` представление или копию данных. Если возвращено представление, то второе выражение в нашем присваивании по цепочке будет вызовом метода `__setitem__` исходного объекта. Однако если будет возвращена копия, то результатом будет копия, модифицированная вместо исходного объекта, сам исходный объект не будет изменен.
Вот в этом и заключается смысл предупреждения `A value is trying to be set on a copy of a slice from a DataFrame`. Происходит попытка изменить значение в копии среза данных датафрейма, но это не изменит исходный датафрейм.
`SettingWithCopyWarning` сообщает нам, что библиотека pandas не может определить, что было возвращено первым вызовом метода `__getitem__` –копия или представление, и поэтому неясно, изменило ли присваивание исходный объект или нет. Еще один способ ответить на вопрос, почему библиотека pandas выдает нам предупреждение – подумать о том, что применительно к рассматриваемому примеру ответ на вопрос «исходный объект модифицирован?» не известен.
Мы хотим изменить исходный объект, и решение, которое предполагает предупреждение, состоит в том, чтобы преобразовать эти две отдельные цепочки в одну операцию присваивания с помощью свойства `.loc`. Это приведет к удалению индексирования по цепочке из нашего кода, и мы

больше не получим предупреждение. Наш исправленный программный код и его расширенная версия будут выглядеть так:

```
# наш код
data.loc[data.bidder == 'parakeet2004', 'bidderrate'] = 100

# а вот так происходит выполнение кода
data.loc.__setitem__((data.__getitem__('bidder') == 'parakeet2004', 'bidderrate'), 100)
```

Свойство `.loc` гарантирует, что мы работаем с исходным объектом `DataFrame`, получив расширенные возможности индексирования.

Ложные пропуски

Использование свойства `.loc` не решает полностью наши проблемы, потому что операции `get` со свойством `.loc` могут возвращать либо представление, либо копию. Давайте быстро рассмотрим один сложный пример.

In[23]:
```
# извлекаем значения переменных bidderrate и bid, когда
# переменная bidder принимает значение parakeet2004
data.loc[data.bidder == 'parakeet2004', ('bidderrate', 'bid')]
```

Out[23]:

	bidderrate	bid
6	100	3.00
7	100	10.00
8	100	24.99

На этот раз мы извлекли два столбца, а не один. Попробуем задать новые значения для переменной *bid*.

In[24]:
```
# пробуем поменять значения переменной bid
data.loc[data.bidder == 'parakeet2004', ('bidderrate', 'bid')]['bid'] = 5.0
data.loc[data.bidder == 'parakeet2004', ('bidderrate', 'bid')]
```

Out[24]:

	bidderrate	bid
6	100	3.00
7	100	10.00
8	100	24.99

Никакого эффекта и никакого предупреждения! Мы задаем значение в копии среза, но библиотека pandas не обнаруживает этого, перед нами – пример ложного пропуска. Сам факт применения свойства `.loc` не означает, что мы можем использовать присваивание по цепочке. Существует старая, нерешенная проблема для конкретно этой ошибки https://github.com/pandas-dev/pandas/issues/9767.

Корректный программный код для этого случая выглядит так:

In[25]:
```
# решением будет следующий программный код
data.loc[data.bidder == 'parakeet2004', 'bid'] = 5.0
data.loc[data.bidder == 'parakeet2004', ('bidderrate', 'bid')]
```

Out[25]:

	bidderrate	bid
6	100	5.0
7	100	5.0
8	100	5.0

Вы можете задаться вопросом, как на практике может возникнуть такая проблема, но это еще цветочки по сравнению с тем, что можно получить, когда мы обращаемся к объектам `DataFrame` и полученные результаты присваиваем переменным, все это будет показано в следующем разделе.

И вновь о скрытой цепочке

Давайте еще раз взглянем на наш пример со скрытой цепочкой, рассмотренный ранее, где мы пытались задать значение переменной *bidder* для наблюдения с меткой `304` датафрейма `winners`.

In[26]:
```
# снова обратимся к примеру со скрытой цепочкой
winners = data.loc[data.bid == data.price]
winners.loc[304, 'bidder'] = 'therealname'
```

Out[26]:
```
C:\Anaconda3\lib\site-packages\ipykernel_launcher.py:3: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-
docs/stable/indexing.html#indexing-view-versus-copy
```

Мы получили предупреждение `SettingWithCopyWarning`, хотя использовали свойство `.loc`. Эта проблема может сбить с толку, но давайте подумаем о переменной `winners`. Чем она является на самом деле? Учитывая, что мы создали экземпляр `data.loc[data.bid == data.price]`, мы не знаем, является ли он представлением или копией нашего исходного датафрейма `data` (поскольку операции `get` возвращают либо представление, либо копию). Объединение операции создания экземпляра со строкой программного кода, породившей предупреждение, делает нашу ошибку очевидной.

In[27]:
```
# объединяем операцию создания экземпляра со строкой
# программного кода, породившей предупреждение
data.loc[data.bid == data.price].loc[304, 'bidder'] = 'therealname'
```

Out[27]:
```
C:\Anaconda3\lib\site-packages\ipykernel_launcher.py:3: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-
docs/stable/indexing.html#indexing-view-versus-copy
```

Мы снова использовали присваивание по цепочке, просто на этот раз оно было разбито на две строки программного кода. Еще один способ задуматься – задать вопрос «У нас изменяется один или два объекта?» В нашем случае ответ неизвестен: если датафрейм `winners` является копией, тогда изменения затрагивают только датафрейм `winners`, но если он является представлением, то и датафрейм `winners`, и датафрейм `data` будут модифицированы. Эта ситуация может возникать между строками, которые очень далеко отстоят друг от друга в скрипте или кодовой базе, что затрудняет отслеживание источника проблемы.
Цель предупреждения здесь – уберечь нас от мыслей о том, будто мы меняем исходный датафрейм, хотя на самом деле мы этого не делаем, или о том, что мы модифицируем копию данных, а не исходные данные. Способ решения этой проблемы во многом будет зависеть от наших собственных намерений. Если мы хотим работать с копией наших исходных данных, решение состоит в том, чтобы заставить библиотеку pandas сделать копию.

```
In[28]:
# работаем с датафреймом как копией
winners = data.loc[data.bid == data.price].copy()
winners.loc[304, 'bidder'] = 'therealname'

print(data.loc[304, 'bidder']) # исходный датафрейм
print(winners.loc[304, 'bidder']) # копия
```

Если, с другой стороны, вам требуется обновить исходный объект `DataFrame`, то нужно работать с исходным объектом `DataFrame`. Наш предыдущий код стал бы выглядеть так:

```
In[29]:
# находим покупателей, ставка которых выиграла
winner_mask = data.bid == data.price
# выведем несколько наблюдений
data.loc[winner_mask].head()
# выполняем анализ
mean_win_time = data.loc[winner_mask, 'bidtime'].mean()
... # 20 строк программного кода
mode_open_bid = data.loc[winner_mask, 'openbid'].mode()
# изменяем имя покупателя
data.loc[304, 'bidder'] = 'therealname'
```

В более сложных обстоятельствах, таких как модификация подмножества подмножества объекта `DataFrame`, вместо использования обращения к индексу по цепочке можно изменять срезы данных, получаемые через `.loc` из исходного датафрейма. Например, вы можете изменить нашу новую переменную `winner_mask`, приведенную выше, или создать новую переменную, которая отбирает подмножество датафрейма `winners`, как показано ниже:

```
In[30]:
# создаем новую переменную, которая отбирает
# подмножество датафрейма winners
high_winner_mask = winner_mask & (data.price > 150)
data.loc[high_winner_mask].head()
```

Out[30]:

	auctionid	bid	bidtime	bidder	bidderrate	openbid	price	bidtime_hours
225	8213387444	152.0	2.919757	uconnbabydoll1975	15	0.99	152.0	70.074168
328	8213935134	207.5	2.983542	toby2492	0	0.10	207.5	71.605008
416	8214430396	199.0	2.990463	volpendesta	4	9.99	199.0	71.771112
531	8215582227	152.5	2.999664	ultimatum_man	2	60.00	152.5	71.991936

Этот прием более надежен с точки зрения будущего обслуживания и масштабирования программного кода.

ПРИЛОЖЕНИЕ 8. ОТ PANDAS К SCIKIT-LEARN – НОВЫЙ ПОДХОД К УПРАВЛЕНИЮ РАБОЧИМИ ПРОЦЕССАМИ

Материал подготовлен основателем компании DunderData Тедом Петроу, специалистом по работе с данными компании «Актион-МЦФЭР» Степаном Соколом, директором исследовательского центра «Гевисста» Артемом Груздевым

Эта статья доступна в виде тетрадки Jupiter по адресу https://colab.research.google.com/drive/1yHnTLJVWDzI7_WqjTlgRMBOTcwt8qIr1 (ее можно открыть и редактировать в режиме «песочницы»), а также на Github https://github.com/DunderData/Machine-Learning-Tutorials организации DunderData.

Новый уровень интеграции Scikit-Learn с Pandas

Scikit-Learn готовит одно из самых больших обновлений за последние несколько лет в виде значительного релиза версии 0.20. Для многих специалистов по анализу данных типичный рабочий процесс состоит в использовании библиотеки pandas для разведочного анализа данных, которое предшествует машинному обучению в библиотеке scikit-learn. Новое обновление сделает этот процесс проще, стабильнее, стандартизует и обогатит функционал.

Краткое резюме и цели статьи

- Эта статья предназначена для тех, кто использует библиотеку scikit-learn в качестве библиотеки машинного обучения, но при этом применяют библиотеку pandas в качестве инструмента исследования и обработки данных.
- Предполагается, что читатель знаком с библиотеками pandas и scikit-Learn.
- Будет рассмотрен новый класс `ColumnTransformer`, позволяющий применять отдельные преобразования над различными частями набора данных одновременно непосредственно перед конкатенацией полученных результатов.
- Основным узким местом для пользователей (и по мнению автора самой уязвимой частью scikit-learn) была подготовка столбцов датафрейма, содержащих строковые значения. Этот процесс должен стать более стандартизованным.
- Класс `OneHotEncoder` обновлен в части кодирования признаков со строчными значениями.
- Для упрощения процедуры дамми-кодирования мы воспользуемся новым классом `SimpleImputer` для заполнения пропущенных значений константами.

- Мы создадим пользовательский класс, который выполнит все «базовые» преобразования над набором данных вместо применения встроенных инструментов библиотеки scikit-learn.
- И, наконец, биннинг количественных переменных будет осуществлен с помощью нового класса `KbinsDiscretizer`.

Те, кто использует pandas в качестве инструмента для разведочного анализа и обработки данных, наверняка знакомы с нестандартными методами обработки столбцов, содержащих строковые значения. Модели машинного обучения scikit-learn требуют данных, представленных в виде двумерной структуры числовых значений. Строковые значения недопустимы. Библиотека scikit-learn никогда не предлагала единого способ обработки столбцов со строковыми значениями, которые часто встречаются в данных.

Это привело к появлению ряда пособий по обработке столбцов со строковыми значениями разными способами. В некоторых решениях предлагается использовать функцию `get_dummies` библиотеки pandas. В других решениях используется класс `LabelBinarizer` библиотеки scikit-learn, который хотя и выполняет дамми-кодирование, предназначался для меток зависимой переменной, а не для признаков. В третьих создаются собственные пользовательские классы. Были созданы даже целые пакеты типа sklearn-pandas, чтобы решить эту задачу. Такой недостаток стандартизации доставлял неудобства для тех, кто обучал модели машинного обучения на данных, содержащих столбцы со строковыми значениями.

Более того, не существовало достаточной поддержки преобразований для отдельных столбцов, а не для всего набора данных. Например, общепринято выполнять стандартизацию только количественных переменных. Теперь это будет намного проще.

Теперь обновим scikit-learn до версии 0.20.

```
conda update scikit-learn
pip install -U scikit-learn
```

Знакомство с классом ColumnTransformer и обновленным классом OneHotEncoder

С обновленной версией 0.20 многие рабочие процессы в pandas и scikit-learn станут унифицированными. Класс `ColumnTransformer` применяет трансформацию над определенным подмножеством признаков имеющегося объекта `DataFrame` или массива.

Класс `OneHotEncoder` уже давно реализован, однако с обновлением он сможет кодировать столбцы со строковыми значениями. До сих пор он мог кодировать лишь столбцы типа `object`.

Далее будет рассмотрено, как эти нововведения обрабатывают столбцы со строковыми значениями в датафрейме pandas.

Задача предсказания цен на недвижимость с Kaggle

Одним из начинающихся соревнований по машинному обучению на Kaggle (на момент написания книги) является соревнование Housing Prices: Advanced Regression Techniques (https://www.kaggle.com/c/house-prices-advanced-regression-techniques). Целью является предсказание цен на жилье по 80 признакам, представленным смесью из количественных и категориальных предикторов. Данные можно скачать с сайта Kaggle или репозитория https://github.com/Kaggle/kaggle-api.

Исследуем данные

Импортируем необходимые библиотеки, считываем данные в объект `DataFrame` и выводим первые пять строк:

In[1]:

```
# импортируем необходимые библиотеки
import pandas as pd
import numpy as np

# записываем CSV-файл в объект DataFrame
train = pd.read_csv('https://raw.githubusercontent.com/'
                    'DunderData/Machine-Learning-Tutorials/'
                    'master/data/housing/train.csv')

# выводим первые 5 наблюдений
train.head()
```

Out[1]:

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities	...	PoolArea	PoolQC	Fence	MiscFeature	MiscVal
0	1	60	RL	65.0	8450	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	NaN	NaN	0
1	2	20	RL	80.0	9600	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	NaN	NaN	0
2	3	60	RL	68.0	11250	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	NaN	0
3	4	70	RL	60.0	9550	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	NaN	0
4	5	60	RL	84.0	14260	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	NaN	0

5 rows × 81 columns

Посмотрим количество наблюдений и количество признаков в обучающем датафрейме.

In[2]:

```
# смотрим форму обучающего датафрейма
train.shape
```

Out[2]:

```
(1460, 81)
```

Удаление зависимой переменной из обучающего набора

Зависимой переменной является *SalesPrice*, которую мы превратим в массив меток. Он будет использован позднее при обучении модели.

In[3]:

```
# создаем массив меток
y = train.pop('SalePrice').values
```

Кодировка отдельного столбца со строковыми значениями

Для начала закодируем отдельный столбец со строковыми значениями *HouseStyle*, который содержит значения, описывающие внешний облик дома. Давайте выведем частоту каждого строкового значения.

```
In[4]:
# выводим частоты категорий переменной HouseStyle
vc = train['HouseStyle'].value_counts()
vc

Out[4]:
1Story    726
2Story    445
1.5Fin    154
SLvl       65
SFoyer     37
1.5Unf     14
2.5Unf     11
2.5Fin      8
Name: HouseStyle, dtype: int64
```

Данный столбец содержит 8 уникальных значений.

Scikit-Learn – только двумерные данные

Большинство моделей scikit-Learn требуют, чтобы данными были строго двумерными. Если мы выбираем столбец `train['HouseStyle']`, технически будет создан объект Series, который является одномерным объектом данных. Можно заставить библиотеку Pandas создать объект `DataFrame` с единственным столбцом, передав список, состоящий из одного элемента, в квадратные скобки:

```
In[5]:
# создаем объект DataFrame с одним столбцом
hs_train = train[['HouseStyle']].copy()
hs_train.ndim

Out[5]:
2
```

Импортируем класс, создаем экземпляр класса - модель, обучаем модель – трехэтапный процесс работы с моделью.

Интерфейс библиотеки scikit-learn совместим со всеми моделями машинного обучения и использует трехэтапный процесс обучения на данных.

- Импортируем нужный нам класс из соответствующего модуля.
- Создаем экземпляр класса – модель, возможно изменив значения гиперпараметров по умолчанию.
- Обучаем модель на данных. Возможно при необходимости мы преобразуем данные в новое пространство.

Ниже мы импортируем класс `OneHotEncoder`, создаем экземпляр класса, убеждаемся в том, что мы получаем плотный (а не разреженный) массив,

наконец, выполняем дамми-кодирование нашего отдельного столбца с помощью метода `.fit_transform()`.

```
In[6]:
# импортируем класс OneHotEncoder
from sklearn.preprocessing import OneHotEncoder
# создаем экземпляр класса OneHotEncoder,
# будем возвращать плотный массив
ohe = OneHotEncoder(sparse=False)
# выполняем дамми-кодирование переменной
# HouseStyle в обучающем наборе
hs_train_transformed = ohe.fit_transform(hs_train)
# смотрим результаты
hs_train_transformed

Out[6]:
array([[0., 0., 0., ..., 1., 0., 0.],
       [0., 0., 1., ..., 0., 0., 0.],
       [0., 0., 0., ..., 1., 0., 0.],
       ...,
       [0., 0., 0., ..., 1., 0., 0.],
       [0., 0., 1., ..., 0., 0., 0.],
       [0., 0., 1., ..., 0., 0., 0.]])
```

Как и ожидалось, каждое уникальное значение становится отдельным бинарным признаком.

```
In[7]:
# убеждаемся, что каждое уникальное значение
# получило свой столбец (8 уникальных значений
# даст 8 столбцов)
hs_train_transformed.shape

Out[7]:
(1460, 8)
```

У нас NumPy массив. Где имена столбцов?

Отметим, что полученный нами объект представляет собой массив NumPy, а не датафрейм pandas. Библиотека scikit-learn изначально не предполагала непосредственной интеграции с библиотекой pandas. Все объекты библиотеки pandas конвертируются в массивы NumPy под капотом и именно они всегда возвращаются после выполнения преобразования.

Однако мы все же можно извлечь имя столбца из объекта `OneHotEncoder` с помощью метода `.get_feature_names()`.

```
In[8]:
# запишем имена столбцов в объект feature_names
feature_names = ohe.get_feature_names()
# смотрим имена столбцов
feature_names

Out[8]:
array(['x0_1.5Fin', 'x0_1.5Unf', 'x0_1Story', 'x0_2.5Fin', 'x0_2.5Unf',
       'x0_2Story', 'x0_SFoyer', 'x0_SLvl'], dtype=object)
```

Проверка корректности первой строки данных

Полезной привычкой будет проверка корректной работы модели. Давайте выведем первую строку преобразованных данных.

In[9]:
```
# выведем первую строку преобразованных данных
row0 = hs_train_transformed[0]
row0
```

Out[9]:
```
array([0., 0., 0., 0., 0., 1., 0., 0.])
```

Шестое значение массива было закодировано единицей. Используя логическое индексирование, можно выяснить имя переменной.

In[10]:
```
# выясним имя переменной, которое
# в первой строке закодировано 1
feature_names[row0 == 1]
```

Out[10]:
```
array(['x0_2Story'], dtype=object)
```

Теперь убедимся, что первое значение в нашем исходном столбце является тем же самым.

In[11]:
```
# смотрим значение исходного столбца
# HouseStyle в строке 0
hs_train.values[0]
```

Out[11]:
```
array(['2Story'], dtype=object)
```

Используем метод .inverse_transform() для автоматизации данной операции

Как и у большинства классов-трансформеров, у объекта `OneHotEncoder` есть метод `.inverse_transform()`, который вернет вам исходные данные. Здесь мы должны обернуть `row0` в список, чтобы получить двумерный массив.

In[12]:
```
# выполним обратное преобразование
ohe.inverse_transform([row0])
```

Out[12]:
```
array([['2Story']], dtype=object)
```

Мы можем проверить все значения сразу, выполнив инвертирование всего преобразованного массива.

In[13]:
```
# выполняем инвертирование всего
# преобразованного массива
hs_inv = ohe.inverse_transform(hs_train_transformed)
hs_inv
```

Out[13]:
```
array([['2Story'],
       ['1Story'],
       ['2Story'],
       ...,
       ['2Story'],
       ['1Story'],
       ['1Story']], dtype=object)
```

In[14]:
```
# проверяем, совпадают ли массивы
np.array_equal(hs_inv, hs_train.values)
```

Out[14]:
```
True
```

Применение преобразования к тестовому набору

Любое преобразование, применяемое к обучающему набору, должно быть применено и к тестовому набору. Давайте прочитаем тестовый набор данных, извлечем тот же самый столбец и применим к нему наше преобразование.

In[15]:
```
# выполняем дамми-кодирование переменной HouseStyle
# в тестовом наборе
test = pd.read_csv('https://raw.githubusercontent.com/'
                   'DunderData/Machine-Learning-Tutorials/'
                   'master/data/housing/test.csv')
hs_test = test[['HouseStyle']].copy()
hs_test_transformed = ohe.transform(hs_test)
hs_test_transformed
```

Out[15]:
```
array([[0., 0., 1., ..., 0., 0., 0.],
       [0., 0., 1., ..., 0., 0., 0.],
       [0., 0., 0., ..., 1., 0., 0.],
       ...,
       [0., 0., 1., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 1., 0.],
       [0., 0., 0., ..., 1., 0., 0.]])
```

По итогам дамми-кодирования должно получиться 8 столбцов, что мы и видим.

In[16]:
```
# убеждаемся, что каждое уникальное значение
# получило свой столбец (8 уникальных значений
# даст 8 столбцов)
hs_test_transformed.shape
```

Out[16]:
```
(1459, 8)
```

Этот пример работает исправно, однако существует множество случаев, когда мы можем столкнуться с проблемами. Они будут рассмотрены ниже.

Проблема №1 – Новые категории в тестовом наборе

Что случится, если в тестовом наборе попадется такая категория переменной *HouseStyle*, уникальна только для него? Например, значение `3Story`. Давайте заменим значение переменной *HouseStyle* в первой строке тестового набора и попытаемся выполнить дамми-кодирование.

In[17]:
```
# копируем переменную HouseStyle
# в тестовом наборе
hs_test = test[['HouseStyle']].copy()
# заменяем значение в переменной HouseStyle
hs_test.iloc[0, 0] = '3Story'
# выводим первые три наблюдения
hs_test.head(3)
```

Out[17]:

	HouseStyle
0	3Story
1	1Story
2	2Story

In[18]:
```
# выполняем дамми-кодирование переменной
# HouseStyle в тестовом наборе
ohe.transform(hs_test)
```

```
ValueError: Found unknown categories ['3Story'] in column 0 during transform
```

Ошибка: Unknown Category

По умолчанию объект `OneHotEncoder` выдаст ошибку. Вероятно, это то, что нужно, ведь нам необходимо знать, есть ли в тестовом наборе новые категории. Если такая проблема возникла, то необходимо более детальное исследование. Сейчас же мы проигнорируем эту проблему и соответствующее наблюдение закодируем строкой, состоящей из нулей, задав для параметра `handle_unknown` значение `'ignore'` при создании экземпляра класса `OneHotEncoder`.

In[19]:
```
# создаем экземпляр класса OneHotEncoder, задав
# для параметра handle_unknown значение ignore
ohe = OneHotEncoder(sparse=False, handle_unknown='ignore')
# обучаем модель
ohe.fit(hs_train)
# выполняем дамми-кодирование переменной
# HouseStyle в тестовом наборе
hs_test_transformed = ohe.transform(hs_test)
hs_test_transformed
```

Out[19]:
```
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 1., ..., 0., 0., 0.],
       [0., 0., 0., ..., 1., 0., 0.],
       ...,
       [0., 0., 1., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 1., 0.],
       [0., 0., 0., ..., 1., 0., 0.]])
```

Теперь проверим, состоит ли первая строка из одних нулей.

In[20]:
```
# проверим, состоит ли первая строка из одних нулей
hs_test_transformed[0]
```

Out[20]:
```
array([0., 0., 0., 0., 0., 0., 0., 0.])
```

Проблема №2 – Пропущенные значения в тестовом наборе

Если тестовый набор содержит пропущенные значения (NaN или None), то они будут проигнорированы при условии, что для параметра `handle_unknown` задано значение `'ignore'`. Давайте заменим первые два значения переменной *HouseStyle* в тестовом наборе на пропуски.

In[21]:
```
# копируем переменную HouseStyle
# в тестовом наборе
hs_test = test[['HouseStyle']].copy()
# заменяем первые два значения
# на пропуски
hs_test.iloc[0, 0] = np.nan
hs_test.iloc[1, 0] = None
# выводим первые четыре наблюдения
hs_test.head(4)
```

Out[21]:

	HouseStyle
0	NaN
1	None
2	2Story
3	2Story

Проблема №3 – Пропущенные значения в обучающем наборе

Пропущенные значения в обучающей выборке являются большей проблемой. На данный момент мы не можем обучить `OneHotEncoder` на данных, содержащих пропуски.

In[22]:
```
# копируем переменную HouseStyle
# в обучающем наборе
hs_train = hs_train.copy()
# заменяем первое значение на пропуск
hs_train.iloc[0, 0] = np.nan
# создаем экземпляр класса OneHotEncoder
ohe = OneHotEncoder(sparse=False, handle_unknown='ignore')
# выполняем дамми-кодирование переменной
# HouseStyle в обучающем наборе
ohe.fit_transform(hs_train)
```

```
ValueError: Input contains NaN
```

В данной ситуации было бы воспользоваться той же самой возможностью проигнорировать пропуски, как в примере с тестовым набором выше. Но на данный момент такой возможности нет, поэтому необходимо их импутировать.

Необходимость импутации пропущенных значений

Теперь необходимо заполнить пропущенные значения. Класс `Imputer` из модуля `preprocessing` объявлен устаревшим. Новый модуль `Impute` пришел ему на замену со своим классом `SimpleImputer` и новой стратегией `'constant'`. По умолчанию при использовании этой стратегии пропуски будут заменены на строку `'missing_value'`. Но есть возможность заполнять пропуски любыми значениями с помощью параметра `fill_value`.

```
In[23]:
# копируем переменную HouseStyle
# в обучающем наборе
hs_train = train[['HouseStyle']].copy()
# заменяем первое значение на пропуск
hs_train.iloc[0, 0] = np.nan
# импортируем класс SimpleImputer
from sklearn.impute import SimpleImputer
# создаем экземпляр класса SimpleImputer
si = SimpleImputer(strategy='constant', fill_value='MISSING')
# выполняем импутацию
hs_train_imputed = si.fit_transform(hs_train)
hs_train_imputed

Out[23]:
array([['MISSING'],
       ['1Story'],
       ['2Story'],
       ...,
       ['2Story'],
       ['1Story'],
       ['1Story']], dtype=object)
```

Вот теперь мы можем выполнить дамми-кодирование, как уже делали это ранее.

```
In[24]:
# выполняем дамми-кодирование переменной HouseStyle
# в импутированном обучающем наборе
hs_train_transformed = ohe.fit_transform(hs_train_imputed)
hs_train_transformed

Out[24]:
array([[0., 0., 0., ..., 1., 0., 0.],
       [0., 0., 1., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 1., ..., 0., 0., 0.],
       [0., 0., 1., ..., 0., 0., 0.]])
```

Стоит отметить, что теперь в наборе данных присутствует дополнительный столбец и дополнительное имя столбца.

Больше о методе .fit_transform()

Для всех моделей метод `.fit_transform()` сначала вызывает метод `.fit()`, а затем вызывает метод `.transform()`. Метод `.fit()` вычисляет то, что будет использовано в ходе преобразования. Например, если бы мы задали для `SimpleImputer` стратегию `'mean'`, то при вызове метода `.fit()` мы бы вычисляли и сохраняли среднее значение по каждому признаку. А метод `.transform()` использует эти средние значения для заполнения пропусков в каждом соответствующем столбце и возвращает преобразованный массив.

OneHotEncoder работает аналогично. Во время выполнения метода `.fit()`, он находит все уникальные значения для каждого столбца и сохраняет их. Когда вызывается метод `.transform()`, сохраненные уникальные значения для каждого столбца используются для создания бинарных признаков.

Применение нескольких преобразований к тестовому набору

Кроме того, мы можем вручную применить каждое из описанных выше преобразований в следующем порядке:

In[25]:
```
# копируем переменную HouseStyle
# в тестовом наборе
hs_test = test[['HouseStyle']].copy()
# заменяем первые два значения на новое
# строковое значение и пропуск
hs_test.iloc[0, 0] = 'unique value to test set'
hs_test.iloc[1, 0] = np.nan
# выполняем импутацию
hs_test_imputed = si.transform(hs_test)
# выполняем дамми-кодирование
hs_test_transformed = ohe.transform(hs_test_imputed)
# убеждаемся, что каждое уникальное значение
# получило свой столбец (8 уникальных значений плюс
# 1 новое unique value to test set даст 9 столбцов)
hs_test_transformed.shape
```

Out[25]:
```
(1459, 9)
```

Применение конвейера

Библиотека scikit-learn предлагает класс `Pipeline`, который принимает список преобразований и выполняет их последовательно. Кроме того, в него можно поместить модель машинного обучения в качестве конечного класса. Ниже приведен пример простой импутации и дамми-кодирования.

In[26]:
```
# импортируем класс Pipeline
from sklearn.pipeline import Pipeline
```

Каждый этап конвейера представляет собой кортеж, состоящий из двух элементов – названия шага и экземпляра класса. Результат предыдущего этапа будет входными данными для последующего этапа.

In[27]:
```
# задаем этап импутации
si_step = ('si', SimpleImputer(strategy='constant',
                               fill_value='MISSING'))
# задаем этап дамми-кодирования
ohe_step = ('ohe', OneHotEncoder(sparse=False,
                                 handle_unknown='ignore'))
# создаем список этапов
steps = [si_step, ohe_step]
# создаем конвейер, передав список этапов
pipe = Pipeline(steps)
# копируем переменную HouseStyle
# в обучающем наборе
hs_train = train[['HouseStyle']].copy()
# заменяем первое значение на пропуск
hs_train.iloc[0, 0] = np.nan
# выполняем импутацию и дамми-кодирование
# переменной HouseStyle в обучающем наборе
# с помощью каждого этапа конвейера
hs_transformed = pipe.fit_transform(hs_train)
# убеждаемся, что каждое уникальное значение
# получило свой столбец (8 уникальных значений
# плюс 1 новое MISSING даст 9 столбцов)
hs_transformed.shape
```

Out[27]:
```
(1460, 9)
```

Преобразования для тестового набора легко выполняются с помощью соответствующего этапа конвейера, для этого нужно просто передать тестовый набор в метод `.transform()`.

In[28]:
```
# копируем переменную HouseStyle
# в тестовом наборе
hs_test = test[['HouseStyle']].copy()
# выполняем импутацию и дамми-кодирование
# переменной HouseStyle в контрольном наборе
# с помощью каждого этапа конвейера
hs_test_transformed = pipe.transform(hs_test)
# убеждаемся, что каждое уникальное значение
# получило свой столбец (8 уникальных значений
# плюс 1 новое MISSING даст 9 столбцов)
hs_test_transformed.shape
```

Out[28]:
```
(1459, 9)
```

Почему для тестового набора мы вызываем только метод .transform()?

Для выполнения преобразований тестового набора нам нужно вызвать метод `.transform()`, а не `.fit_transform()`, поскольку библиотека scikit-learn уже нашла всю необходимую информацию, которая ей понадобиться для преобразования любого другого набора, содержащего те же самые имена столбцов.

Выполнение преобразований для нескольких столбцов со строковыми значениями

Выполнить преобразование нескольких столбцов со строковыми значениями так же просто. Необходимо выбрать нужные столбцы и передать их в тот же самый конвейер.

In[29]:
```
# создаем список столбцов RoofMatl и HouseStyle
# со строковыми значениями
string_cols = ['RoofMatl', 'HouseStyle']
# создаем датафрейм из двух столбцов
# RoofMatl и HouseStyle
string_train = train[string_cols]
# выводим первые три наблюдения
string_train.head(3)
```

Out[29]:

	RoofMatl	HouseStyle
0	CompShg	2Story
1	CompShg	1Story
2	CompShg	2Story

Обращение к отдельным этапам конвейера

При необходимости можно обратиться к отдельному этапу конвейера по его имени с помощью атрибута `named_steps`. В этом примере обратимся к `OneHotEncoder`, чтобы вывести названия признаков.

```
In[30]:
# обращаемся к этапу ohe
ohe = pipe.named_steps['ohe']
# извлекаем названия признаков
ohe.get_feature_names()

Out[30]:
array(['x0_1.5Fin', 'x0_1.5Unf', 'x0_1Story', 'x0_2.5Fin', 'x0_2.5Unf',
       'x0_2Story', 'x0_MISSING', 'x0_SFoyer', 'x0_SLvl'], dtype=object)
```

Использование нового ColumnTransformer для отбора столбцов

Класс `ColumnTransformer` (входящий в новый модуль `compose`) позволяет определить, для каких столбцов нужно выполнить определенные преобразования. Категориальные признаки мы почти всегда преобразовываем иначе, чем количественные.

В настоящее время класс `ColumnTransformer` проходит испытание, а значит его функционал может измениться в будущем.

`ColumnTransformer` принимает список трехэлементных кортежей. Первое значение в кортеже является **именем** самого кортежа, второе – **экземпляр класса**, третье – **список столбцов**, которые нужно преобразовать. Кортеж будет выглядеть следующим образом:

`('name', SomeTransformer(parameters), columns)`

Столбцы необязательно должны быть именами столбцов, вы можете использовать целочисленные индексы столбцов, массив булевых значений и даже функцию, которая принимает в качестве аргумента весь DataFrame и возвращает набор столбцов.

Также предусмотрена возможность использовать массив NumPy вместе с `ColumnTransformer`, но настоящая статья посвящена интеграции с библиотекой pandas, поэтому рассмотрение функционала будет ограничено объектами `DataFrame`.

Передаем конвейер в ColumnTransformer

Мы даже можем передать конвейер с множеством преобразований в `ColumnTransformer`, что мы и сделаем ниже, поскольку нам нужно применить несколько преобразований к столбцам со строковыми значениями.

Ниже мы выполним ту же самую импутацию и дамми-кодирование, что и в предыдущих примерах, но теперь с использованием `ColumnTransformer`. При этом конвейер остается тем же самым, что и прежде, просто к имени каждого этапа добавим префикс `cat`. В следующем разделе мы добавим дополнительный конвейер для количественных столбцов.

In[31]:
```
# импортируем класс ColumnTransformer
from sklearn.compose import ColumnTransformer
# задаем этап импутации
cat_si_step = ('si', SimpleImputer(strategy='constant',
                                   fill_value='MISSING'))
# задаем этап дамми-кодирования
cat_ohe_step = ('ohe', OneHotEncoder(sparse=False,
                                     handle_unknown='ignore'))
# создаем список этапов
cat_steps = [cat_si_step, cat_ohe_step]
# создаем конвейер, передав список этапов
cat_pipe = Pipeline(cat_steps)
# создаем список столбцов RoofMatl и HouseStyle
# со строковыми значениями
cat_cols = ['RoofMatl', 'HouseStyle']
# создаем список с трехэлементным кортежем
cat_transformers = [('cat', cat_pipe, cat_cols)]
# передаем список в ColumnTransformer
ct = ColumnTransformer(transformers=cat_transformers)
```

Передаем весь объект DataFrame в ColumnTransformer

Экземпляр класса `ColumnTransformer` отбирает нужные нам столбцы, поэтому мы можем просто передать весь датафрейм в метод `.fit_transform()`. Нужные столбцы будут выбраны автоматически.

In[32]:
```
# передаем обучающий набор в ColumnTransformer
X_cat_transformed = ct.fit_transform(train)
# убеждаемся, что преобразования выполнены
X_cat_transformed.shape
```

Out[32]:
```
(1460, 16)
```

Тестовый набор можно преобразовать аналогичным образом.

In[33]:
```
# выполняем преобразование тестового набора
X_cat_transformed_test = ct.transform(test)
# убеждаемся, что преобразования выполнены
X_cat_transformed_test.shape
```

Out[33]:
```
(1459, 16)
```

Извлечение названий признаков

Придется немного поковыряться, чтобы достать названия признаков. Все классы, выполняющие преобразования, хранятся в атрибуте `named_transformers_`. Ниже мы выбираем наш трансформер. Пока он только один – конвейер под названием `'cat'`, где `'cat'` – это имя, первый элемент в трехэлементном кортеже.

In[34]:
```
# выбираем трансформер
pl = ct.named_transformers_['cat']
```

Теперь из этого конвейера мы извлекаем объект `OneHotEncoder` и уже из него можно извлечь имена предикторов.

In[35]:
```
# извлекаем объект OneHotEncoder
ohe = pl.named_steps['ohe']
```

```
# извлекаем имена признаков
ohe.get_feature_names()
```

Out[35]:
```
array(['x0_ClyTile', 'x0_CompShg', 'x0_Membran', 'x0_Metal', 'x0_Roll',
       'x0_Tar&Grv', 'x0_WdShake', 'x0_WdShngl', 'x1_1.5Fin', 'x1_1.5Unf',
       'x1_1Story', 'x1_2.5Fin', 'x1_2.5Unf', 'x1_2Story', 'x1_SFoyer',
       'x1_SLvl'], dtype=object)
```

Преобразование количественных переменных

Количественным переменным могут потребоваться совершенно другие преобразования. Вместо импутации пропусков константами чаще используется импутация средним значением или медианой. И вместо дамми-кодирования обычно выполняют стандартизацию путем вычитания из исходного значения признака среднего значения и делением полученного результата на стандартное отклонение. Это позволяет улучшить качество регрессионных моделей.

Работа со всеми количественными признаками

Вместо выбора одного или нескольких столбцов вручную, как мы делали это выше, работая со столбцами, содержащими строковые значения, можно выбрать все количественные переменные. Чтобы это сделать, нужно сперва найти тип каждого признака с помощью атрибута `dtypes`, после чего проверить, имеет ли данный признак тип `object`. Атрибут `dtypes` позволяет выяснить тип каждого столбца. Мы можем использовать его для поиска столбцов с числовыми или строковыми значениями. Библиотека pandas хранит все столбцы со строковыми значениями как столбцы типа `object`.

In[36]:
```
# смотрим типы первых 5 признаков
train.dtypes.head()
```

Out[36]:
```
Id               int64
MSSubClass       int64
MSZoning        object
LotFrontage    float64
LotArea          int64
dtype: object
```

С помощью атрибута `kind` выводим тип каждой переменной в виде одной буквы.

In[37]:
```
# выводим тип каждой переменной в виде одной буквы
kinds = np.array([dt.kind for dt in train.dtypes])
kinds[:5]
```

Out[37]:
```
array(['i', 'i', 'O', 'f', 'i'], dtype='<U1')
```

Предположим, что все количественные предикторы не относятся к типу `object`. Исходя из этого, мы можем извлечь категориальные признаки.

```
In[38]:
# создаем массив из всех признаков
all_columns = train.columns.values
# создаем массив из количественных признаков
is_num = kinds != 'O'
# отбираем только количественные признаки
num_cols = all_columns[is_num]
# выводим первые 5 количественных признаков
num_cols[:5]
Out[38]:
array(['Id', 'MSSubClass', 'LotFrontage', 'LotArea', 'OverallQual'],
      dtype=object)

In[39]:
# отбираем только категориальные признаки
cat_cols = all_columns[~is_num]
# выводим первые 5 категориальных признаков
cat_cols[:5]

Out[39]:
array(['MSZoning', 'Street', 'Alley', 'LotShape', 'LandContour'],
      dtype=object)
```

Как только количественные признаки определены, мы вновь можем воспользоваться классом `ColumnTransformer`.

```
In[40]:
# импортируем класс StandardScaler
from sklearn.preprocessing import StandardScaler
# задаем этап импутации
num_si_step = ('si', SimpleImputer(strategy='median'))
# задаем этап стандартизации
num_ss_step = ('ss', StandardScaler())
# создаем список этапов
num_steps = [num_si_step, num_ss_step]
# создаем конвейер, передав список этапов
num_pipe = Pipeline(num_steps)
# создаем список с трехэлементным кортежем
num_transformers = [('num', num_pipe, num_cols)]
# создаем трансформер, передав список
ct = ColumnTransformer(transformers=num_transformers)
# выполняем преобразования обучающего
# набора с помощью трансформера
X_num_transformed = ct.fit_transform(train)
# убеждаемся, что преобразования выполнены
X_num_transformed.shape

Out[40]:
(1460, 37)
```

Передаем конвейер с преобразованиями для категориальных признаков и конвейер с преобразования для количественных признаков в ColumnTransformer

С помощью `ColumnTransformer` для каждого типа столбцов можно применить отдельные преобразования. Нам нужно определиться с операциями преобразования для категориальных признаков и операциями преобразования для количественных признаков. После этого создаем отдельный конвейер для категориальных признаков, отдельный конвейер для количественных признаков, затем используем `ColumnTransformer` для независимого преобразования признаков.

Преобразования в обоих конвейерах будут выполняться параллельно, а их результаты будут сконкатенированы.

In[41]:
```
# создаем список трехэлементный кортежей, в котором
# первый элемент кортежа - название конвейера
# с преобразованиями для определенного типа признаков
transformers = [('cat', cat_pipe, cat_cols),
                ('num', num_pipe, num_cols)]
# передаем список в ColumnTransformer
ct = ColumnTransformer(transformers=transformers)
# выполняем преобразования для категориальных
# и количественных признаков
X = ct.fit_transform(train)
# убеждаемся, что преобразования выполнены
X.shape
```

Out[41]:
```
(1460, 305)
```

Машинное обучение

Целью всех вышеописанных этапов была подготовка данных для последующего машинного обучения. Мы можем создать итоговый конвейер и добавить в него в качестве последнего этапа модель машинного обучения. Первым этапом этого конвейера будет преобразование данных, описанное выше. В качестве зависимой переменной y, как было принято выше, берется переменная *SalePrice*. Здесь мы просто используем метод `.fit()` вместо `.fit_transform()`, поскольку итоговым этапом конвейера будет модель машинного обучения, которая не производит никаких преобразований.

In[42]:
```
# импортируем класс Ridge
from sklearn.linear_model import Ridge
# добавляем в конвейер новый этап - модель машинного обучения
# (модель гребневой регрессии)
ml_pipe = Pipeline([('transform', ct), ('ridge', Ridge())])
# выполняем преобразования и обучаем модель гребневой регрессии
ml_pipe.fit(train, y)
```

Оценить качество модели можно, например, с помощью метода `.score()`, возвращающего значение R-квадрат.

In[43]:
```
# оцениваем качество модели
ml_pipe.score(train, y)
```

Out[43]:
```
0.9220545988101002
```

Перекрестная проверка

Разумеется, оценка модели на обучающем наборе данных бесполезна. Чтобы лучше понять, как модель поведет себя на неизвестных ей данных, нужно провести *k*-блочную перекрестную проверку. Для воспроизводимости результатов нужно задать значение `random_state`.

In[44]:
```
# импортируем класс KFold, функцию cross_val_score()
from sklearn.model_selection import KFold, cross_val_score
# создаем экземпляр класса KFold
kf = KFold(n_splits=5, shuffle=True, random_state=123)
# выполняем перекрестную проверку, конвейер размещен
# внутри цикла перекрестной проверки
cross_val_score(ml_pipe, train, y, cv=kf).mean()
```

Out[44]:
```
0.813392001956967
```

Отбор наилучших значений гиперпараметров с помощью решетчатого поиска

Решетчатый поиск в библиотеке scikit-learn требует передачи словаря, ключами которого будут названия гиперпараметров, а значениями – списки значений этих гиперпараметров. При работе с конвейером мы должны взять имя этапа с двумя символами нижнего подчеркивания на конце и после этого указать имя гиперпараметра. Если мы работаем с многоуровневым конвейером, как в примере ниже, то тогда двойное нижнее подчеркивание должно разделять имена всех уровней (`transform__num__si__strategy`, здесь мы работаем с уровнями `transform`, `num` и `si`), пока не будет достигнуто название гиперпараметра экземпляра класса (в данном случае название гиперпараметра `strategy` объект `si`, экземпляр класса `SimpleImputer`), по значениям которого должен быть осуществлен поиск (в данном случае, как можно увидеть ниже, поиск должен быть осуществлен по значениям `'mean'` и `'median'`).

In[45]:
```
# импортируем класс GridSearchCV
from sklearn.model_selection import GridSearchCV
# задаем сетку гиперпараметров
param_grid = {
    'transform__num__si__strategy': ['mean', 'median'],
    'ridge__alpha': [.001, 0.1, 1.0, 5, 10, 50, 100, 1000]
}
# передаем конвейер в объект GridSearchCV
gs = GridSearchCV(ml_pipe, param_grid, cv=kf, return_train_score=True)
# выполняем решетчатый поиск
gs.fit(train, y)
# смотрим наилучшие значения гиперпараметров
gs.best_params_
```

Out[45]:
```
{'ridge__alpha': 10, 'transform__num__si__strategy': 'median'}
```

In[46]:
```
# смотрим наилучшее значение R-квадрат
gs.best_score_
```

Out[46]:
```
0.8190367464419682
```

Представление результатов решетчатого поиска в виде датафрейма pandas

Результаты решетчатого поиска хранятся в атрибуте `cv_results_`. Он представляет собой словарь, а значит его можно представить в виде объекта `DataFrame` для удобства чтения.

In[47]:

```
# представляем результаты решетчатого поиска
# в виде датафрейма pandas
pd.DataFrame(gs.cv_results_)
```

Out[47]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_ridge__alpha	param_transform__num__si__strategy	params
0	0.021664	0.003552	0.006165	0.000787	0.001	mean	{'ridge__alpha': 0.001, 'transform__num__si__s...
1	0.021008	0.000238	0.005862	0.000031	0.001	median	{'ridge__alpha': 0.001, 'transform__num__si__s...
2	0.019300	0.000115	0.005716	0.000137	0.1	mean	{'ridge__alpha': 0.1, 'transform__num__si__str...
3	0.021357	0.000452	0.005779	0.000120	0.1	median	{'ridge__alpha': 0.1, 'transform__num__si__str...
4	0.021137	0.001272	0.006105	0.000373	1	mean	{'ridge__alpha': 1.0, 'transform__num__si__str...

Создание пользовательского трансформера, выполняющего основные преобразования

У вышеописанного подхода к обработке данных есть недостатки. Например, было бы удобно, если бы `OneHotEncoder` позволял игнорировать пропущенные данные в ходе применения метода `.fit()`. Ведь можно закодировать пропущенные значения, просто заполнив строку нулями. Но сейчас метод требует, чтобы пропуски были заменены некоторым строковым значением, которое потом будет представлено в виде отдельного бинарного признака.

Редкие категории

Строковые значения признака, которые встретились в обучающем наборе несколько раз, вряд ли будут полезными при валидации модели. Иногда их желательно закодировать так, как если бы эти значения были пропусками.

Написание собственного пользовательского класса

В документации scikit-learn дается справка по написанию собственного класса. Класс `BaseEstimator`, расположенный внутри модуля `base` предлагает методы `get_params` и `set_params`. Метод `set_params` необходим при выполнении решетчатого поиска. Можно написать собственный класс или наследовать из `BaseEstimator`. Существует также класс `TransformerMixin`, однако он позволяет лишь написать метод `.fit_transform()`.

Класс `BasicTransformer`, который мы сейчас создадим, выполнет следующие операции:

- заполняет пропуски в количественных признаках средним значением или медианой;
- стандартизирует все количественные признаки;
- выполняет дамми-кодирование для признаков со строковыми значениями;
- не заполняет пропуски в категориальных признаках, а при выполнении дамми-кодирования заменяет пропущенное значение строкой, состоящей из нулей;
- игнорирует новые строковые значения (категории) в тестовом наборе;
- позволяет устанавливать порог частоты строковых значений, ниже которого строковое значение по итогам дамми-кодирования будет представлено строкой из нулей.
- работает только с датафреймами pandas, находится в разработке и поэтому может некорректно работать с некоторыми наборами данных
- назван `basic`, потому что содержит все основные преобразования, которые, как правило, применяются для большинства наборов данных.

In[48]:

```
# импортируем класс BaseEstimator
from sklearn.base import BaseEstimator

# создаем класс BasicTransformer
class BasicTransformer(BaseEstimator):

    def __init__(self, cat_threshold=None, num_strategy='median', return_df=False):
        # храним параметры как публичные атрибуты
        self.cat_threshold = cat_threshold

        if num_strategy not in ['mean', 'median']:
            raise ValueError('num_strategy must be either "mean" or "median"')
        self.num_strategy = num_strategy
        self.return_df = return_df

    def fit(self, X, y=None):
        # подразумеваем, что X - это объект DataFrame
        self._columns = X.columns.values

        # разбиваем данные на категориальные и количественные признаки
        self._dtypes = X.dtypes.values
        self._kinds = np.array([dt.kind for dt in X.dtypes])
        self._column_dtypes = {}
        is_cat = self._kinds == 'O'
        self._column_dtypes['cat'] = self._columns[is_cat]
        self._column_dtypes['num'] = self._columns[~is_cat]
        self._feature_names = self._column_dtypes['num']

        # создаем словарь на основе категориального признака,
        # где ключом будет уникальное значение выше порога
        self._cat_cols = {}
        for col in self._column_dtypes['cat']:
            vc = X[col].value_counts()
            if self.cat_threshold is not None:
                vc = vc[vc > self.cat_threshold]
            vals = vc.index.values
```

```
                self._cat_cols[col] = vals
                self._feature_names = np.append(self._feature_names, col + '_' + vals)

        # вычисляем общее количество новых категориальных признаков
        self._total_cat_cols = sum([len(v) for col, v in self._cat_cols.items()])

        # вычисляем среднее или медиану
        self._num_fill = X[self._column_dtypes['num']].agg(self.num_strategy)
        return self

    def transform(self, X):
        # проверяем, есть ли у нас объект DataFrame с теми же именами столбцов,
        # что и в том объекте DataFrame, который использовался для обучения
        if set(self._columns) != set(X.columns):
            raise ValueError('Passed DataFrame has diff cols than fit DataFrame')
        elif len(self._columns) != len(X.columns):
            raise ValueError('Passed DataFrame has diff number of cols than fit DataFrame')

        # заполняем пропуски
        X_num = X[self._column_dtypes['num']].fillna(self._num_fill)

        # стандартизируем количественные признаки
        std = X_num.std()
        X_num = (X_num - X_num.mean()) / std
        zero_std = np.where(std == 0)[0]

        # Если стандартное отклонение 0, то все значения идентичны. Задаем их равными 0.
        if len(zero_std) > 0:
            X_num.iloc[:, zero_std] = 0
        X_num = X_num.values

        # создаем отдельный массив для преобразованных категориальных признаков
        X_cat = np.empty((len(X), self._total_cat_cols), dtype='int')
        i = 0
        for col in self._column_dtypes['cat']:
            vals = self._cat_cols[col]
            for val in vals:
                X_cat[:, i] = X[col] == val
                i += 1

        # конкатенируем преобразованные количественные и категориальные признаки
        data = np.column_stack((X_num, X_cat))

        # возвращаем либо DataFrame, либо массив
        if self.return_df:
            return pd.DataFrame(data=data, columns=self._feature_names)
        else:
            return data

    def fit_transform(self, X, y=None):
        return self.fit(X).transform(X)

    def get_feature_names(self):
        return self._feature_names
```

Применение собственного класса BasicTransformer

Пользовательский класс `BasicTransformer` должен быть пригодным для использования точно так же, как и любой другой класс библиотеки scikit-learn. Мы можем создать экземпляр класса `BasicTransformer` и с его помощью преобразовать данные.

In[49]:

```
# создаем экземпляр класса BasicTransformer
bt = BasicTransformer(cat_threshold=3, return_df=True)
# выполняем преобразование обучающего
# набора с помощью BasicTransformer
train_transformed = bt.fit_transform(train)
```

```
# выводим первые 3 наблюдения
train_transformed.head(3)
```

Out[49]:

	Id	MSSubClass	LotFrontage	LotArea	OverallQual	OverallCond	YearBuilt	YearRemodAdd	MasVnrArea	BsmtFinSF1	...	SaleType_ConLD
0	-1.730272	0.073350	-0.220799	-0.207071	0.651256	-0.517023	1.050634	0.878367	0.513928	0.575228	...	0.0
1	-1.727900	-0.872264	0.460162	-0.091855	-0.071812	2.178881	0.156680	-0.429430	-0.570555	1.171591	...	0.0
2	-1.725528	0.073350	-0.084607	0.073455	0.651256	-0.517023	0.984415	0.829930	0.325803	0.092875	...	0.0

3 rows × 246 columns

Использование BasicTransformer в конвейере

Наш трансформер может быть частью конвейера.

In[50]:
```
# передаем наш BasicTransformer в конвейер
basic_pipe = Pipeline([('bt', bt), ('ridge', Ridge())])
# выполняем преобразования и строим модель гребневой регрессии
basic_pipe.fit(train, y)
# оцениваем качество модели
basic_pipe.score(train, y)
```

Out[50]:
```
0.9035633239411639
```

Кроме того, мы можем передать наш конвейер в цикл перекрестной проверки, как это уже делали выше.

In[51]:
```
# передаем наш конвейер в цикл перекрестной проверки
cross_val_score(basic_pipe, train, y, cv=kf).mean()
```

Out[51]:
```
0.8157670129480052
```

Аналогичным образом мы можем разместить конвейер внутри объекта `GridSearchCV`. Оказывается, исключение редких категорий в данном случае не помогает улучшить качество модели. Наилучшее значение немного выросло, возможно в силу несколько иного способа обработки редких категорий.

In[52]:
```
# задаем сетку гиперпараметров
param_grid = {
    'bt__cat_threshold': [0, 1, 2, 3, 5],
    'ridge__alpha': [.1, 1, 10, 100]
}
# передаем наш конвейер в объект GridSearchCV
gs = GridSearchCV(basic_pipe, param_grid, cv=kf)
# выполняем решетчатый поиск
gs.fit(train, y)
# смотрим наилучшие значения гиперпараметров
gs.best_params_
```

Out[52]:
```
{'bt__cat_threshold': 0, 'ridge__alpha': 10}
```
In[53]:
```
# смотрим наилучшее значение R-квадрат
gs.best_score_
```

Out[53]:
```
0.8297473585998102
```

Биннинг и преобразование количественных переменных с помощью нового класса KBinsDiscretizer

У нас есть несколько признаков, содержащих года. Имеет смысл дискретизировать их и обрабатывать как категориальные переменные. Для решения этой задачи библиотека scikit-learn предложила новый класс `KBinsDiscretizer`. Он не только определяет бины, но и выполняет дамми-кодирование, т.е. каждую полученную категорию (бин) представляет в виде бинарного столбца. Раньше в библиотеке pandas это можно было сделать с помощью функций `cut()` и `qcut()`.
Ниже приведен принцип работы класса `KBinsDiscretizer` на примере признака *YearBuilt*.

In[54]:
```
# импортируем класс KBinsDiscretizer
from sklearn.preprocessing import KBinsDiscretizer
# создаем экземпляр класса KBinsDiscretizer
kbd = KBinsDiscretizer(encode='onehot-dense')
# выполняем биннинг
year_built_transformed = kbd.fit_transform(train[['YearBuilt']])
# смотрим результаты
year_built_transformed
```

Out[54]:
```
array([[0., 0., 0., 0., 1.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       ...,
       [1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.]])
```

По умолчанию каждый интервал содержит (приблизительно) равное количество наблюдений. Давайте убедимся в этом.

In[55]:
```
# убедимся, что бины содержать примерно
# одинаковое количество наблюдений
year_built_transformed.sum(axis=0)
```

Out[55]:
```
array([292., 274., 307., 266., 321.])
```

Теперь посмотрим на границы бинов.

In[56]:
```
# посмотрим на границы бинов
kbd.bin_edges_
```

Out[56]:
```
array([array([1872. , 1947.8, 1965. , 1984. , 2003. , 2010. ])],
      dtype=object)
```

Это мы применили метод `'quantile'`, выставленный по умолчанию (регулируется с помощью гиперпараметра `strategy`). Количество создаваемых бинов по умолчанию равно 5 (регулируется с помощью гиперпараметра `n_bins`). Можно применить метод `'uniform'`, чтобы разбить интервалы на строго равное количество наблюдений или метод `'kmeans'`, использующий кластеризацию методом k-средних для определения интервалов.

Отдельная обработка всех столбцов с годами с помощью ColumnTransformer

Теперь мы возьмем еще один набор признаков, которое требует отдельной обработки и мы выполним его с помощью класса `ColumnTransformer`. Мы добавим еще один этап к нашим предыдущим трансформерам. Кроме того, мы отбросим столбец `id`, который просто идентифицирует каждое наблюдение.

In[57]:
```
# выделяем переменные с годами в отдельный список
year_cols = ['YearBuilt', 'YearRemodAdd', 'GarageYrBlt',
             'YrSold']
# создаем булев массив
not_year = ~np.isin(num_cols, year_cols + ['Id'])
# выделяем количественные переменные, исключив
# переменные с годами и Id
num_cols2 = num_cols[not_year]
# создаем этап импутации
year_si_step = ('si', SimpleImputer(strategy='median'))
# создаем этап биннинга
year_kbd_step = ('kbd', KBinsDiscretizer(n_bins=5,
                                         encode='onehot-dense'))
# создаем список этапов
year_steps = [year_si_step, year_kbd_step]
# создаем конвейер, передав список этапов
year_pipe = Pipeline(year_steps)
# создаем список трехэлементный кортежей, в котором
# первый элемент кортежа - название конвейера с
# преобразованиями для определенного типа признаков
transformers = [('cat', cat_pipe, cat_cols),
                ('num', num_pipe, num_cols2),
                ('year', year_pipe, year_cols)]
# создаем трансформер, передав список
ct = ColumnTransformer(transformers=transformers)
# выполняем преобразования обучающего
# набора с помощью трансформера
X = ct.fit_transform(train)
# убеждаемся, что преобразования выполнены
X.shape
```

Out[57]:
```
(1460, 320)
```

После проведения перекрестной проверки и оценки качества модели очевидно, что все проделанные преобразования не принесли улучшений модели.

In[58]:
```
# добавляем в конвейер новый этап - модель машинного обучения
# (модель гребневой регрессии)
ml_pipe = Pipeline([('transform', ct), ('ridge', Ridge())])
# выполняем перекрестную проверку, конвейер размещен
# внутри цикла перекрестной проверки
cross_val_score(ml_pipe, train, y, cv=kf).mean()
```

Out[58]:
```
0.8127854010155339
```

Перебор разного количества бинов может положительно повлиять на качество модели.

In[59]:
```
# задаем сетку гиперпараметров
param_grid = {
    'transform__year__kbd__n_bins': [4, 6, 8, 10],
    'ridge__alpha': [.1, .5, 1, 5, 10, 100]
}
# передаем наш конвейер в объект GridSearchCV
gs = GridSearchCV(ml_pipe, param_grid, cv=kf)
# выполняем решетчатый поиск
gs.fit(train, y)
# смотрим наилучшие значения гиперпараметров
gs.best_params_
```

Out[59]:
```
{'ridge__alpha': 10, 'transform__year__kbd__n_bins': 6}
```

In[60]:
```
# смотрим наилучшее значение R-квадрат
gs.best_score_
```

Out[60]:
```
0.8198372622112804
```

Применение RobustScaler и FunctionTransformer

Теперь попробуем решить задачу по-другому. Заново загрузим данные, выделим признаки с небольшой отрицательной асимметрией, признаки с небольшой и средней положительной асимметрией и признаки с высокой положительной асимметрией и применим для них соответствующее преобразование, максимизирующее нормальность. Кроме того, для переменных с небольшой и средней положительной асимметрией применим другой тип стандартизации, при котором из каждого исходного значения признака вычитается значение, соответствующее первому квартилю, и полученный результат делится на межквартильный размах (эту стандартизацию можно выполнить с помощью класса `RobustScaler`).

Итак, давайте загрузим данные, создадим массив меток и удалим переменную *Id*.

In[61]:
```
# заново записываем CSV-файл в объект DataFrame
train = pd.read_csv('https://raw.githubusercontent.com/'
                    'DunderData/Machine-Learning-Tutorials/'
                    'master/data/housing/train.csv')

# создаем массив меток
y = train.pop('SalePrice').values
```

```
# удаляем переменную Id
train.drop('Id', axis=1, inplace=True)
```

Теперь создадим список количественных признаков и список категориальных признаков.

In[62]:
```
# выделим категориальные и количественные признаки
cat_columns = train.dtypes[train.dtypes == 'object'].index
num_columns = train.dtypes[train.dtypes != 'object'].index
```

Вычислим коэффициент асимметрии для количественных признаков.

In[63]:
```
# вычислим коэффициент асимметрии для
# количественных признаков
for i in num_columns:
    print(i, train[i].skew())
```

Out[63]:
```
MSSubClass 1.4076567471495591
LotFrontage 2.163569142324884
LotArea 12.207687851233496
OverallQual 0.2169439277628693
OverallCond 0.6930674724842182
YearBuilt -0.613461172488183
YearRemodAdd -0.5035620027004709
MasVnrArea 2.669084210182863
BsmtFinSF1 1.685503071910789
BsmtFinSF2 4.255261108933303
BsmtUnfSF 0.9202684528039037
TotalBsmtSF 1.5242545490627664
1stFlrSF 1.3767566220336365
2ndFlrSF 0.8130298163023265
LowQualFinSF 9.011341288465387
GrLivArea 1.3665603560164552
BsmtFullBath 0.596066609663168
BsmtHalfBath 4.103402697955168
FullBath 0.036561558402727165
HalfBath 0.675897448233722
BedroomAbvGr 0.21179009627507137
KitchenAbvGr 4.488396777072859
TotRmsAbvGrd 0.6763408364355531
Fireplaces 0.6495651830548841
GarageYrBlt -0.6494146238714679
GarageCars -0.3425489297486655
GarageArea 0.17998090674623907
WoodDeckSF 1.5413757571931312
OpenPorchSF 2.3643417403694404
EnclosedPorch 3.08987190371177
3SsnPorch 10.304342032693112
ScreenPorch 4.122213743143115
PoolArea 14.828373640750588
MiscVal 24.476794188821916
MoSold 0.21205298505146022
YrSold 0.09626851386568028
```

Теперь импортируем классы `FunctionTransformer` и `RobustScaler`.

In[64]:
```
# импортируем классы FunctionTransformer, RobustScaler
from sklearn.preprocessing import FunctionTransformer, RobustScaler
```

Теперь выделим списки признаков в зависимости от их коэффициента асимметрии, создадим для каждого списка свой конвейер

преобразований и построим модель гребневой регрессии, вновь для проверки качества применив перекрестную проверку.

In[65]:
```
# выделяем список признаков с небольшой отрицательной асимметрией
neg_skew_num_columns = ['YearBuilt', 'YearRemodAdd', 'GarageYrBlt', 'GarageCars']
# выделяем список признаков с высокой положительной асимметрией
high_pos_skew_num_columns = ['MiscVal', 'PoolArea', '3SsnPorch', 'LotArea', 'LowQualFinSF']
# создадим булев массив
not_neg_high_pos_skew_num_columns = ~np.isin(
    num_columns, high_pos_skew_num_columns + neg_skew_num_columns)
# из списка количественных признаков удалим количественные признаки
# с небольшой отрицательной и высокой положительной асимметрией
num_columns = num_columns[not_neg_high_pos_skew_num_columns]
```

In[66]:
```
# создаем конвейер преобразований для количественных признаков
# с небольшой отрицательной асимметрией
num_negskew_pipe = Pipeline([
    ('imputer', SimpleImputer(strategy='mean')),
    ('square', FunctionTransformer(np.square, validate=False)),
    ('scaler', StandardScaler())
])

# создаем конвейер преобразований для количественных признаков
# с небольшой и средней положительной асимметрией
num_pipe = Pipeline([
    ('imputer', SimpleImputer(strategy='mean')),
    ('log', FunctionTransformer(np.log1p, validate=False)),
    ('scaler', RobustScaler())
])

# создаем конвейер преобразований для количественных
# признаков с высокой асимметрией
num_highposskew_pipe = Pipeline([
    ('imputer', SimpleImputer(strategy='mean')),
    ('sqrt', FunctionTransformer(np.sqrt, validate=False)),
    ('kbd', KBinsDiscretizer(n_bins=5, encode='onehot-dense'))
])

# создаем конвейер преобразований для категориальных
# признаков
cat_pipe = Pipeline([
    ('imputer', SimpleImputer(strategy='constant', fill_value='MISSING')),
    ('ohe', OneHotEncoder(sparse=False, handle_unknown='ignore'))
])

transformers = [('num_negskew', num_negskew_pipe, neg_skew_num_columns),
                ('num', num_pipe, num_columns),
                ('num_highposskew', num_highposskew_pipe, high_pos_skew_num_columns),
                ('cat', cat_pipe, cat_columns)]

transformer = ColumnTransformer(transformers=transformers)

# добавляем в конвейер новый этап - модель машинного обучения
# (модель гребневой регрессии)
ml_pipe = Pipeline([('transform', transformer), ('ridge', Ridge())])
# выполняем перекрестную проверку, конвейер размещен
# внутри цикла перекрестной проверки
cross_val_score(ml_pipe, train, y, cv=kf).mean()
```

Out[66]:
```
0.8307940999181337
```

Теперь попробуем подобрать оптимальные значения гиперпараметров `alpha` и `n_bins` с помощью решетчатого поиска.

In[67]:

```
# задаем сетку гиперпараметров
param_grid = {
    'ridge__alpha': [.1, .5, 1, 5, 10, 100],
    'transform__num_highposskew__kbd__n_bins': [3, 4, 5, 6, 7]
}
# передаем наш конвейер в объект GridSearchCV
gs = GridSearchCV(ml_pipe, param_grid, cv=kf)
# выполняем решетчатый поиск
gs.fit(train, y)
# смотрим наилучшие значения гиперпараметров
gs.best_params_
```

Out[67]:

```
{'ridge__alpha': 10, 'transform__num_highposskew__kbd__n_bins': 7}
```

In[68]:

```
# смотрим наилучшее значение R-квадрат
gs.best_score_
```

Out[68]:

```
0.8474547753257715
```

Подбор преобразования в зависимости от асимметрии распределения переменной, применение `RobustScaler` и более тщательный перебор значений гиперпараметров `alpha` и `n_bins` позволил улучшить результат.