Дж. Вандер Плас

Python

для сложных задач
наука о данных:
и машинное обучение

ПИТЕР®
Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск

2018

Python Data Science Handbook

Essential Tools for Working with Data

Jake VanderPlas

O'REILLY®

ББК 32.973.2-018.1
УДК 004.43
П37

Плас Дж. Вандер

П37 Python для сложных задач: наука о данных и машинное обучение. — СПб.: Питер, 2018. — 576 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-496-03068-7

Книга «Python для сложных задач: наука о данных и машинное обучение» — это подробное руководство по самым разным вычислительным и статистическим методам, без которых немыслима любая интенсивная обработка данных, научные исследования и передовые разработки. Читатели, уже имеющие опыт программирования и желающие эффективно использовать Python в сфере Data Science, найдут в этой книге ответы на всевозможные вопросы, например: как считать этот формат данных в скрипт? как преобразовать, очистить эти данные и манипулировать ими? как визуализировать данные такого типа? как при помощи этих данных разобраться в ситуации, получить ответы на вопросы, построить статистические модели или реализовать машинное обучение?

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.1
УДК 004.43

ISBN 978-1491912058 англ.
ISBN 978-5-496-03068-7

Содержание

Предисловие

Что такое наука о данных

Эта книга посвящена исследованию данных с помощью языка программирования Python. Сразу же возникает вопрос: что же такое *наука о данных* (data science)? Ответ на него дать непросто — настолько данный термин многозначен.

Долгое время активные критики отказывали термину «наука о данных» в праве на существование либо по причине его избыточности (в конце концов, какая наука *не* имеет дела с данными?), либо расценивая этот термин как «модное словечко» для придания красоты резюме и привлечения внимания агентов по найму кадров.

На мой взгляд, в подобных высказываниях критики упускали нечто очень важное. Лучшее из возможных определений науки о данных приведено в диаграмме Венна в науке о данных, впервые опубликованной Дрю Конвеем в его блоге в сентябре 2010 года (рис. 0.1). Междисциплинарность — ключ к ее пониманию.

Рис. 0.1. Диаграмма Венна в науке о данных Дрю Конвея

Хотя некоторые названия немного ироничны, эта диаграмма улавливает суть того, что, как мне кажется, понимается под наукой о данных: она является *междисциплинарным* предметом.

Наука о данных охватывает три отдельные, но пересекающиеся сферы:

- навыки специалиста по математической статистике, умеющего моделировать наборы данных и извлекать из них основное;
- навыки специалиста в области компьютерных наук, умеющего проектировать и использовать алгоритмы для эффективного хранения, обработки и визуализации этих данных;
- экспертные знания предметной области, полученные в ходе традиционного изучения предмета, — умение как формулировать правильные вопросы, так и рассматривать ответы на них в соответствующем контексте.

С учетом этого я рекомендовал бы рассматривать науку о данных не как новую область знаний, которую нужно изучить, а как новый набор навыков, который вы можете использовать в рамках хорошо знакомой вам предметной области. Извещаете ли вы о результатах выборов, прогнозируете ли прибыльность ценных бумаг, занимаетесь ли оптимизацией контекстной рекламы в Интернете или распознаванием микроорганизмов на сделанных с помощью микроскопа фото, ищете ли новые классы астрономических объектов или же работаете с данными в любой другой сфере, цель этой книги — научить задавать новые вопросы о вашей предметной области и отвечать на них.

Для кого предназначена эта книга

«Как именно следует изучать Python?» — один из наиболее часто задаваемых мне вопросов на различных технологических конференциях и встречах. Задают его заинтересованные в технологиях студенты, разработчики или исследователи, часто уже со значительным опытом написания кода и использования вычислительного и цифрового инструментария. Большинству из них не нужен язык программирования Python в чистом виде, они хотели бы изучать его, чтобы применять в качестве инструмента для решения задач, требующих вычислений с обработкой больших объемов данных.

Эта книга не планировалась в качестве введения в язык Python или в программирование вообще. Я предполагаю, что читатель знаком с языком Python, включая описание функций, присваивание переменных, вызов методов объектов, управление потоком выполнения программы и решение других простейших задач. Она должна помочь пользователям языка Python научиться применять стек инструментов исследования данных языка Python — такие библиотеки, как IPython, NumPy, Pandas,

Matplotlib, Scikit-Learn и соответствующие инструменты, — для эффективного хранения, манипуляции и понимания данных.

Почему Python

За последние несколько десятилетий язык программирования Python превратился в первоклассный инструмент для научных вычислений, включая анализ и визуализацию больших наборов данных. Это может удивить давних поклонников Python: сам по себе этот язык не был создан в расчете на анализ данных или научные вычисления.

Язык программирования Python пригоден для науки о данных в основном благодаря большой и активно развивающейся экосистеме пакетов, созданных сторонними разработчиками:

- библиотеки NumPy — для работы с однородными данными в виде массивов;
- библиотеки Pandas — для работы с неоднородными и поименованными данными;
- SciPy — для общих научных вычислительных задач;
- библиотеки Matplotlib — для визуализаций типографского качества;
- оболочки IPython — для интерактивного выполнения и совместного использования кода;
- библиотеки Scikit-Learn — для машинного обучения и множества других инструментов, которые будут упомянуты в дальнейшем.

Если вы ищете руководство по самому языку программирования Python, рекомендую обратить ваше внимание на проект «Краткая экскурсия по языку программирования Python» (https://github.com/jakevdp/WhirlwindTourOfPython). Он знакомит с важнейшими возможностями языка Python и рассчитан на исследователей данных, уже знакомых с одним или несколькими языками программирования.

Языки программирования Python 2 и Python 3. В книге используется синтаксис Python 3, содержащий несовместимые с выпусками 2.x языка программирования Python расширения. Хотя Python 3.0 был впервые выпущен в 2008 году, он внедрялся довольно медленно, особенно в научном сообществе и сообществе веб-разработчиков. Главная причина заключалась в том, что многие важные сторонние пакеты и наборы программ только через некоторое время стали совместимы с новым языком.

В начале 2014 года стабильные выпуски важнейших инструментов экосистемы науки о данных были полностью совместимы как с языком Python 2, так и Python 3, поэтому данная книга использует более новый синтаксис языка Python 3. Однако абсолютное большинство фрагментов кода в этой книге будет также работать без всяких модификаций на языке Python 2. Случаи, когда применяется несовместимый с Python 2 синтаксис, я буду указывать.

Общая структура книги

Каждая глава книги посвящена конкретному пакету или инструменту, составляющему существенную часть инструментария Python для исследования данных.

- *IPython и Jupyter (глава 1)* — предоставляют вычислительную среду, в которой работают многие использующие Python исследователи данных.
- *NumPy (глава 2)* — предоставляет объект `ndarray` для эффективного хранения и работы с плотными массивами данных в Python.
- *Pandas (глава 3)* — предоставляет объект `DataFrame` для эффективного хранения и работы с поименованными/столбчатыми данными в Python.
- *Matplotlib (глава 4)* — предоставляет возможности для разнообразной гибкой визуализации данных в Python.
- *Scikit-Learn (глава 5)* — предоставляет эффективные реализации на Python большинства важных и широко известных алгоритмов машинного обучения.

Мир PyData гораздо шире представленных пакетов, и он растет день ото дня. С учетом этого я использую каждую возможность в книге, чтобы сослаться на другие интересные работы, проекты и пакеты, расширяющие пределы того, что можно сделать на языке Python. Тем не менее сегодня эти пять пакетов являются основополагающими для многого из того, что можно сделать в области применения языка программирования Python к исследованию данных. Я полагаю, что они будут сохранять свое значение и при росте окружающей их экосистемы.

Использование примеров кода

Дополнительные материалы (примеры кода, упражнения и т. п.) доступны для скачивания по адресу https://github.com/jakevdp/PythonDataScienceHandbook.

Задача этой книги — помочь вам делать вашу работу. Вы можете использовать любой пример кода из книги в ваших программах и документации. Обращаться к нам за разрешением нет необходимости, разве что вы копируете значительную часть кода. Например, написание программы, использующей несколько фрагментов кода из этой книги, не требует отдельного разрешения. Однако для продажи или распространения компакт-диска с примерами из книг O'Reilly разрешение требуется. Ответ на вопрос путем цитирования этой книги и цитирования примеров кода не требует разрешения. Но включение значительного количества примеров кода из книги в документацию к вашему продукту потребует разрешения.

Мы ценим, хотя и не требуем ссылки на первоисточник. Она включает название, автора, издательство и ISBN. Например, «Python для сложных задач. Наука

о данных и машинное обучение (“Питер”). Copyright Джейк Вандер Пласс, 2017, 978-1-491-91205-8».

Если вам кажется, что вы выходите за рамки правомерного использования примеров кода, не стесняясь, связывайтесь с нами по адресу permissions@oreilly.com.

Вопросы установки

Инсталляция Python и набора библиотек, обеспечивающих возможность научных вычислений, не представляет сложности. В данном разделе будут рассмотрены особенности, которые следует принимать во внимание при настройке.

Существует множество вариантов установки Python, но я предложил бы воспользоваться дистрибутивом Anaconda, одинаково работающим в операционных системах Windows, Linux и Mac OS X. Дистрибутив Anaconda существует в двух вариантах.

- *Miniconda* (http://conda.pydata.org/miniconda.html) содержит сам интерпретатор языка программирования Python, а также утилиту командной строки conda, функционирующую в качестве межплатформенной системы управления пакетами, ориентированной на работу с пакетами Python и аналогичной по духу утилитам apt и yum, хорошо знакомым пользователям операционной системы Linux.
- *Anaconda* (https://www.continuum.io/downloads) включает интерпретатор Python и утилиту conda, а также набор предустановленных пакетов, ориентированных на научные вычисления. Приготовьтесь к тому, что установка займет несколько гигабайт дискового пространства.

Все включаемые в Anaconda пакеты можно также установить вручную поверх Miniconda, именно поэтому я рекомендую вам начать с Miniconda.

Для начала скачайте и установите пакет Miniconda (не забудьте выбрать версию с языком Python 3), после чего установите базовые пакеты, используемые в данной книге:

```
[~]$ conda install numpy pandas scikit-learn matplotlib seaborn ipython-notebook
```

На протяжении всей книги мы будем применять и другие, более специализированные утилиты из научной экосистемы Python, установка которых сводится к выполнению команды `conda install` *`название_пакета`*. За дополнительной информацией об утилите conda, включая информацию о создании и использовании сред разработки conda (что я бы *крайне* рекомендовал), обратитесь к онлайн-документации утилиты conda (http://conda.pydata.org/docs/).

Условные обозначения

В этой книге используются следующие условные обозначения.

Курсив

Курсивом выделены новые термины.

`Моноширинный шрифт`

Используется для листингов программ, а также внутри абзацев, чтобы обратиться к элементам программы вроде переменных, функций и типов данных. Им также выделены имена и расширения файлов.

`Полужирный моноширинный шрифт`

Показывает команды или другой текст, который пользователь должен ввести самостоятельно.

`Курсивный моноширинный шрифт`

Показывает текст, который должен быть заменен значениями, введенными пользователем, или значениями, определяемыми контекстом.

1 IPython: за пределами обычного Python

Меня часто спрашивают, какой из множества вариантов среды разработки для Python я использую в своей работе. Мой ответ иногда удивляет спрашивающих: моя излюбленная среда представляет собой оболочку IPython (http://ipython.org/) плюс текстовый редактор (в моем случае редактор Emacs или Atom, в зависимости от настроения). IPython (сокращение от *«интерактивный Python»*) был основан в 2001 году Фернандо Пересом в качестве продвинутого интерпретатора Python и с тех пор вырос в проект, призванный обеспечить, по словам Переса, «утилиты для всего жизненного цикла исследовательских расчетов». Если язык Python — механизм решения нашей задачи в области науки о данных, то оболочку IPython можно рассматривать как интерактивную панель управления.

Оболочка IPython является полезным интерактивным интерфейсом для языка Python и имеет несколько удобных синтаксических дополнений к нему. Большинство из них мы рассмотрим. Кроме этого, оболочка IPython тесно связана с проектом Jupiter (http://jupyter.org/), предоставляющим своеобразный блокнот (текстовый редактор) для браузера, удобный для разработки, совместной работы и использования ресурсов, а также для публикации научных результатов. Блокнот оболочки IPython, по сути, частный случай более общей структуры блокнота Jupiter, включающего блокноты для Julia, R и других языков программирования. Не стоит далеко ходить за примером, показывающим удобства формата блокнота, им служит страница, которую вы сейчас читаете: вся рукопись данной книги была составлена из набора блокнотов IPython.

Оболочка IPython позволяет эффективно использовать язык Python для интерактивных научных вычислений, требующих обработки большого количества данных. В этой главе мы рассмотрим возможности оболочки IPython, полезные

при исследовании данных. Сосредоточим свое внимание на тех предоставляемых IPython синтаксических возможностях, которые выходят за пределы стандартных возможностей языка Python. Немного углубимся в «магические» команды, позволяющие ускорить выполнение стандартных задач при создании и использовании предназначенного для исследования данных кода. И наконец, затронем возможности блокнота, полезные для понимания смысла данных и совместного использования результатов.

Командная строка или блокнот?

Существует два основных способа использования оболочки IPython: командная строка IPython и блокнот IPython. Большая часть изложенного в этой главе материала относится к обоим, а в примерах будет применяться более удобный в конкретном случае вариант. Я буду отмечать немногие разделы, относящиеся только к одному из них. Прежде чем начать, вкратце расскажу, как запускать командную оболочку IPython и блокнот IPython.

Запуск командной оболочки IPython

Данная глава, как и большая часть книги, не предназначена для пассивного чтения. Я рекомендую вам экспериментировать с описываемыми инструментами и синтаксисом: формируемая при этом мышечная память принесет намного больше пользы, чем простое чтение. Начнем с запуска интерпретатора оболочки IPython путем ввода команды `IPython` в командной строке. Если же вы установили один из таких дистрибутивов, как Anaconda или EPD, вероятно, у вас есть запускающий модуль для вашей операционной системы (мы обсудим это подробнее в разделе «Справка и документация в оболочке Python» данной главы).

После этого вы должны увидеть приглашение к вводу:

```
IPython 4.0.1 – An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
Help      -> Python's own help system.
Object?   -> Details about 'object', use 'object??' for extra details.
In [1]:
```

Далее вы можете активно следить за происходящим в книге.

Запуск блокнота Jupiter

Блокнот Jupiter — браузерный графический интерфейс для командной оболочки IPython и богатый набор основанных на нем возможностей динамической

визуализации. Помимо выполнения операторов Python/IPython, блокнот позволяет пользователю вставлять форматированный текст, статические и динамические визуализации, математические уравнения, виджеты JavaScript и многое другое. Более того, эти документы можно сохранять так, что другие люди смогут открывать и выполнять их в своих системах.

Хотя просмотр и редактирование блокнота IPython осуществляется через окно браузера, он должен подключаться к запущенному процессу Python для выполнения кода. Для запуска этого процесса (называемого *ядром* (kernel)) выполните следующую команду в командной строке вашей операционной системы:

```
$ jupyter notebook
```

Эта команда запустит локальный веб-сервер, видимый браузеру. Она сразу же начнет журналировать выполняемые действия. Журнал будет выглядеть следующим образом:

```
$ jupyter notebook
[NotebookApp] Serving notebooks from local directory: /Users/jakevdp/…
[NotebookApp] 0 active kernels
[NotebookApp] The IPython Notebook is running at: http://localhost:8888/
[NotebookApp] Use Control-C to stop this server and shut down all kernels…
```

После выполнения этой команды ваш браузер по умолчанию должен автоматически запуститься и перейти по указанному локальному URL; точный адрес зависит от вашей системы. Если браузер не открывается автоматически, можете запустить его вручную и перейти по указанному адресу (в данном примере http://localhost:8888/).

Справка и документация в оболочке IPython

Если вы не читали остальных разделов в данной главе, прочитайте хотя бы этот. Обсуждаемые здесь утилиты внесли наибольший (из IPython) вклад в мой ежедневный процесс разработки.

Когда человека с техническим складом ума просят помочь другу, родственнику или коллеге решить проблему с компьютером, чаще всего речь идет об умении быстро найти неизвестное решение. В науке о данных все точно так же: допускающие поиск веб-ресурсы, такие как онлайн-документация, дискуссии в почтовых рассылках и ответы на сайте Stack Overflow, содержат массу информации, даже (особенно?) если речь идет о теме, информацию по которой вы уже искали. Уметь эффективно исследовать данные означает скорее не запоминание утилит или команд, которые нужно использовать в каждой из возможных ситуаций, а знание того, как эффек-

тивно искать неизвестную пока информацию: посредством ли поиска в Интернете или с помощью других средств.

Одна из самых полезных возможностей IPython/Jupiter заключается в сокращении разрыва между пользователями и типом документации и поиска, что должно помочь им эффективнее выполнять свою работу. Хотя поиск в Интернете все еще играет важную роль в ответе на сложные вопросы, большое количество информации можно найти, используя саму оболочку IPython. Вот несколько примеров вопросов, на которые IPython может помочь ответить буквально с помощью нескольких нажатий клавиш.

- Как вызвать эту функцию? Какие аргументы и параметры есть у нее?
- Как выглядит исходный код этого объекта Python?
- Что имеется в импортированном мной пакете? Какие атрибуты или методы есть у этого объекта?

Мы обсудим инструменты IPython для быстрого доступа к этой информации, а именно символ `?` для просмотра документации, символы `??` для просмотра исходного кода и клавишу Tab для автодополнения.

Доступ к документации с помощью символа ?

Язык программирования Python и его экосистема для исследования данных являются клиентоориентированными, и в значительной степени это проявляется в доступе к документации. Каждый объект Python содержит ссылку на строку, именуемую *docstring* (сокращение от documentation string — «строка документации»), которая в большинстве случаев будет содержать краткое описание объекта и способ его использования. В языке Python имеется встроенная функция `help()`, позволяющая обращаться к этой информации и выводить результат. Например, чтобы посмотреть документацию по встроенной функции `len`, можно сделать следующее:

```
In [1]: help(len)
Help on built-in function len in module builtins:

len(…)
    len(object) -> integer

    Return the number of items of a sequence or mapping[1].
```

В зависимости от интерпретатора информация будет отображена в виде встраиваемого текста или в отдельном всплывающем окне.

---

[1] Возвращает количество элементов в последовательности или словаре. — *Здесь и далее примеч. пер.*

Поскольку поиск справочной информации по объекту — очень распространенное и удобное действие, оболочка IPython предоставляет символ ? для быстрого доступа к документации и другой соответствующей информации:

```
In [2]: len?
Type:        builtin_function_or_method
String form: <built-in function len>
Namespace:   Python builtin
Docstring:
len(object) -> integer

Return the number of items of a sequence or mapping[1].
```

Данная нотация подходит практически для чего угодно, включая методы объектов:

```
In [3]: L = [1, 2, 3]
In [4]: L.insert?
Type:        builtin_function_or_method
String form: <built-in method insert of list object at 0x1024b8ea8>
Docstring:   L.insert(index, object) – insert object before index[2]
```

или даже сами объекты с документацией по их типу:

```
In [5]: L?
Type:        list
String form: [1, 2, 3]
Length:      3
Docstring:
list() -> new empty list[3]
list(iterable) -> new list initialized from iterable's items[4]
```

Это будет работать даже для созданных пользователем функций и других объектов! В следующем фрагменте кода мы опишем маленькую функцию с docstring:

```
In [6]: def square(a):
  ....:     """Return the square of a."""
  ....:     return a ** 2
  ....:
```

Обратите внимание, что при создании docstring для нашей функции мы просто вставили в первую строку строковый литерал. Поскольку docstring обычно занимает несколько строк, в соответствии с условными соглашениями мы использовали для многострочных docstring нотацию языка Python с тройными кавычками.

---

[1] Возвращает количество элементов в последовательности или словаре.

[2] Вставляет object перед index.

[3] Новый пустой список.

[4] Новый список, инициализированный элементами списка iterable.

Теперь воспользуемся знаком ? для нахождения этой docstring:

```
In [7]: square?
Type:        function
String form: <function square at 0x103713cb0>
Definition:  square(a)
Docstring:   Return the square a.
```

Быстрый доступ к документации через элементы docstring — одна из причин, по которым желательно приучить себя добавлять подобную встроенную документацию в создаваемый код!

Доступ к исходному коду с помощью символов ??

Поскольку текст на языке Python читается очень легко, чтение исходного кода интересующего вас объекта может обеспечить более глубокое его понимание. Оболочка IPython предоставляет сокращенную форму обращения к исходному коду — двойной знак вопроса (??):

```
In [8]: square??
Type:        function
String form: <function square at 0x103713cb0>
Definition:  square(a)
Source:
def square(a):
    "Return the square of a"
    return a ** 2
```

Для подобных простых функций двойной знак вопроса позволяет быстро проникнуть в особенности внутренней реализации.

Поэкспериментировав с ним немного, вы можете обратить внимание, что иногда добавление в конце ?? не приводит к отображению никакого исходного кода: обычно это происходит потому, что объект, о котором идет речь, реализован не на языке Python, а на C или каком-либо другом транслируемом языке расширений. В подобном случае добавление ?? приводит к такому же результату, что и добавление ?. Вы столкнетесь с этим в отношении многих встроенных объектов и типов Python, например для упомянутой выше функции len:

```
In [9]: len??
Type:        builtin_function_or_method
String form: <built-in function len>
Namespace:   Python builtin
Docstring:
len(object) -> integer

Return the number of items of a sequence or mapping[1].
```

[1] Возвращает количество элементов в последовательности или словаре.

Использование `?` и/или `??` — простой способ для быстрого поиска информации о работе любой функции или модуля языка Python.

Просмотр содержимого модулей с помощью Tab-автодополнения

Другой удобный интерфейс оболочки IPython — использование клавиши Tab для автодополнения и просмотра содержимого объектов, модулей и пространств имен. В следующих примерах мы будем применять обозначение `<TAB>` там, где необходимо нажать клавишу Tab.

Автодополнение названий содержимого объектов с помощью клавиши Tab

С каждым объектом Python связано множество различных атрибутов и методов. Аналогично обсуждавшейся выше функции `help` в языке Python есть встроенная функция `dir`, возвращающая их список, но на практике использовать интерфейс Tab-автодополнения гораздо удобнее. Чтобы просмотреть список всех доступных атрибутов объекта, необходимо набрать имя объекта с последующим символом точки (`.`) и нажать клавишу Tab:

```
In [10]: L.<TAB>
L.append   L.copy     L.extend   L.insert   L.remove   L.sort
L.clear    L.count    L.index    L.pop      L.reverse
```

Чтобы сократить этот список, можно набрать первый символ или несколько символов нужного имени и нажать клавишу Tab, после чего будут отображены соответствующие атрибуты и методы:

```
In [10]: L.c<TAB>
L.clear  L.copy   L.count

In [10]: L.co<TAB>
L.copy   L.count
```

Если имеется только один вариант, нажатие клавиши Tab приведет к автодополнению строки. Например, следующее будет немедленно заменено на `L.count`:

```
In [10]: L.cou<TAB>
```

Хотя в языке Python отсутствует четкое разграничение между открытыми/внешними и закрытыми/внутренними атрибутами, по соглашениям, для обозначения подобных методов принято использовать знак подчеркивания. Для ясности эти закрытые методы, а также специальные методы по умолчанию исключаются из списка, но можно вывести их список, набрав знак подчеркивания:

```
In [10]: L._<TAB>
L.__add__           L.__gt__            L.__reduce__
L.__class__         L.__hash__          L.__reduce_ex__
```

Для краткости я показал только несколько первых строк вывода. Большинство этих методов — методы специального назначения языка Python, отмеченные двойным подчеркиванием в названии (на сленге называемые dunder[1]-методами).

Автодополнение с помощью клавиши Tab во время импорта

Tab-автодополнение удобно также при импорте объектов из пакетов. Воспользуемся им для поиска всех возможных вариантов импорта в пакете `itertools`, начинающихся с `co`:

```
In [10]: from itertools import co<TAB>
combinations                  compress
combinations_with_replacement  count
```

Аналогично можно использовать Tab-автодополнение для просмотра доступных в системе вариантов импорта (полученное вами может отличаться от нижеприведенного листинга в зависимости от того, какие сторонние сценарии и модули являются видимыми данному сеансу Python):

```
In [10]: import <TAB>
Display all 399 possibilities? (y or n)
Crypto              dis                 py_compile
Cython              distutils           pyclbr
…                   …                   …
difflib             pwd                 zmq

In [10]: import h<TAB>
hashlib             hmac                http
heapq               html                husl
```

Отмечу, что для краткости я не привожу здесь все 399 пакетов и модулей, доступных в моей системе для импорта.

Помимо автодополнения табуляцией, подбор по джокерному символу

Автодополнение табуляцией удобно в тех случаях, когда вам известны первые несколько символов искомого объекта или атрибута. Однако оно малопригодно, когда необходимо найти соответствие по символам, находящимся в середине или конце

[1] Игра слов: одновременно сокращение от double underscore — «двойное подчеркивание» и dunderhead — «тупица», «болван».

слова. На этот случай оболочка IPython позволяет искать соответствие названий по джокерному символу *.

Например, можно использовать следующий код для вывода списка всех объектов в пространстве имен, заканчивающихся словом `Warning`:

```
In [10]: *Warning?
BytesWarning                  RuntimeWarning
DeprecationWarning            SyntaxWarning
FutureWarning                 UnicodeWarning
ImportWarning                 UserWarning
PendingDeprecationWarning     Warning
ResourceWarning
```

Обратите внимание, что символ * соответствует любой строке, включая пустую.

Аналогично предположим, что мы ищем строковый метод, содержащий где-то в названии слово `find`. Отыскать его можно следующим образом:

```
In [10]: str.*find*?
Str.find
str.rfind
```

Я обнаружил, что подобный гибкий поиск с помощью джокерных символов очень удобен для поиска нужной команды при знакомстве с новым пакетом или обращении после перерыва к уже знакомому.

Сочетания горячих клавиш в командной оболочке IPython

Вероятно, все, кто проводит время за компьютером, используют в своей работе сочетания горячих клавиш. Наиболее известные — Cmd+C и Cmd+V (или Ctrl+C и Ctrl+V), применяемые для копирования и вставки в различных программах и системах. Опытные пользователи выбирают популярные текстовые редакторы, такие как Emacs, Vim и другие, позволяющие выполнять множество операций посредством замысловатых сочетаний клавиш.

В командной оболочке IPython также имеются сочетания горячих клавиш для быстрой навигации при наборе команд. Эти сочетания горячих клавиш предоставляются не самой оболочкой IPython, а через ее зависимости от библиотеки GNU Readline: таким образом определенные сочетания горячих клавиш могут различаться в зависимости от конфигурации вашей системы. Хотя некоторые из них работают в блокноте для браузера, данный раздел в основном касается сочетаний горячих клавиш именно в командной оболочке IPython.

После того как вы привыкнете к сочетаниям горячих клавиш, вы сможете их использовать для быстрого выполнения команд без изменения исходного положения рук на клавиатуре. Если вы пользователь Emacs или имеете опыт работы с Linux-подобными командными оболочками, некоторые сочетания горячих клавиш покажутся вам знакомыми. Мы сгруппируем их в несколько категорий: *навигационные горячие клавиши*, *горячие клавиши ввода текста*, *горячие клавиши для истории команд* и *прочие горячие клавиши*.

Навигационные горячие клавиши

Использовать стрелки «влево» (←) и «вправо» (→) для перемещения назад и вперед по строке вполне естественно, но есть и другие возможности, не требующие изменения исходного положения рук на клавиатуре (табл. 1.1).

Таблица 1.1. Горячие клавиши для навигации

Комбинация клавиш	Действие
Ctrl+A	Перемещает курсор в начало строки
Ctrl+E	Перемещает курсор в конец строки
Ctrl+B (или стрелка «влево»)	Перемещает курсор назад на один символ
Ctrl+F (или стрелка «вправо»)	Перемещает курсор вперед на один символ

Горячие клавиши ввода текста

Для удаления предыдущего символа привычно использовать клавишу Backspace, несмотря на то что требуется небольшая гимнастика для пальцев, чтобы до нее дотянуться. Эта клавиша удаляет только один символ за раз. В оболочке IPython имеется несколько сочетаний горячих клавиш для удаления различных частей набираемого текста. Наиболее полезные из них — команды для удаления сразу целых строк текста (табл. 1.2). Вы поймете, что привыкли к ним, когда поймаете себя на использовании сочетания Ctrl+B и Ctrl+D вместо клавиши Backspace для удаления предыдущего символа!

Таблица 1.2. Горячие клавиши для ввода текста

Комбинация клавиш	Действие
Backspace	Удаляет предыдущий символ в строке
Ctrl+D	Удаляет следующий символ в строке
Ctrl+K	Вырезает текст, начиная от курсора и до конца строки
Ctrl+U	Вырезает текст с начала строки до курсора
Ctrl+Y	Вставляет предварительно вырезанный текст
Ctrl+T	Меняет местами предыдущие два символа

Горячие клавиши для истории команд

Вероятно, наиболее важные из обсуждаемых здесь сочетаний горячих клавиш — сочетания для навигации по истории команд. Данная история команд распространяется за пределы текущего сеанса оболочки IPython: полная история команд хранится в базе данных SQLite в каталоге с профилем IPython. Простейший способ получить к ним доступ — с помощью стрелок «вверх» (↑) и «вниз» (↓) для пошагового перемещения по истории, но есть и другие варианты (табл. 1.3).

Таблица 1.3. Горячие клавиши для истории команд

Комбинация клавиш	Действие
Ctrl+P (или стрелка «вверх»)	Доступ к предыдущей команде в истории
Ctrl+N (или стрелка «вниз»)	Доступ к следующей команде в истории
Ctrl+R	Поиск в обратном направлении по истории команд

Особенно полезным может оказаться поиск в обратном направлении. Как вы помните, в предыдущем разделе мы описали функцию `square`. Выполним поиск в обратном направлении по нашей истории команд Python в новом окне оболочки IPython и найдем это описание снова. После нажатия Ctrl+R в терминале IPython вы должны увидеть следующее приглашение командной строки:

```
In [1]:
(reverse-i-search)`':
```

Если начать вводить символы в этом приглашении, IPython автоматически будет дополнять их, подставляя недавние команды, соответствующие этим символам, если такие существуют:

```
In [1]:
(reverse-i-search)`sqa': square??
```

Вы можете в любой момент добавить символы для уточнения поискового запроса или снова нажать Ctrl+R для поиска следующей соответствующей запросу команды.

Если вы выполняли все действия предыдущего раздела по ходу чтения книги, нажмите Ctrl+R еще два раза, и вы получите следующее:

```
In [1]:
(reverse-i-search)`sqa': def square(a):
    """Return the square of a"""
    return a ** 2
```

Найдя искомую команду, нажмите Enter и поиск завершится. После этого можно использовать найденную команду и продолжить работу в нашем сеансе:

```
In [1]: def square(a):
    """Return the square of a"""
    return a ** 2

In [2]: square(2)
Out[2]: 4
```

Обратите внимание, что также можно использовать сочетания клавиш Ctrl+P/Ctrl+N или стрелки вверх/вниз для поиска по истории команд, но только по совпадающим символам в начале строки. Если вы введете `def` и нажмете Ctrl+P, будет найдена при наличии таковой последняя команда в истории, начинающаяся с символов def.

Прочие горячие клавиши

Имеется еще несколько сочетаний клавиш, не относящихся ни к одной из предыдущих категорий, но заслуживающих упоминания (табл. 1.4).

Таблица 1.4. Дополнительные горячие клавиши

Комбинация клавиш	Действие
Ctrl+L	Очистить экран терминала
Ctrl+C (или стрелка «вниз»)	Прервать выполнение текущей команды Python
Ctrl+D	Выйти из сеанса IPython[1]

Сочетание горячих клавиш Ctrl+C особенно удобно при случайном запуске очень долго работающего задания.

Хотя использование некоторых сочетаний горячих клавиш может показаться утомительным, вскоре у вас появится соответствующая мышечная память и вы будете жалеть, что эти команды недоступны в других программах.

«Магические» команды IPython

Предыдущие два раздела продемонстрировали возможности эффективного использования и изучения языка Python с помощью оболочки IPython. Здесь же мы начнем обсуждать некоторые предоставляемые IPython расширения обычного синтаксиса языка Python. В IPython они известны как *«магические» команды* (magic commands) и отличаются указанием перед названием команды символа %. Эти «магические» команды предназначены для быстрого решения различных распространенных задач стандартного анализа данных.

[1] В отличие от упомянутого выше идентичного сочетания горячих клавиш это сочетание работает при пустой строке приглашения ко вводу.

Существует два вида «магических» команд: *строчные «магические» команды*, обозначаемые одним знаком `%` и работающие с одной строкой ввода, и *блочные «магические» команды*, обозначаемые удвоенным префиксом (`%%`) и работающие с несколькими строками ввода. Мы обсудим несколько коротких примеров и вернемся к более подробному обсуждению некоторых «магических» команд далее в данной главе.

Вставка блоков кода: %paste и %cpaste

При работе с интерпретатором IPython одна из распространенных проблем заключается в том, что вставка многострочных блоков кода может привести к неожиданным ошибкам, особенно при использовании отступов и меток интерпретатора. Довольно часто случается, что вы находите пример кода на сайте и хотите вставить его в интерпретатор. Рассмотрим следующую простую функцию:

```
>>> def donothing(x):
…     return x
```

Код отформатирован так, как он будет отображаться в интерпретаторе языка Python, и если вы скопируете и вставите его непосредственно в оболочку IPython, то вам будет возвращена ошибка:

```
In [2]: >>> def donothing(x):
   …:     …     return x
   …:
  File "<ipython-input-20-5a66c8964687>", line 2
    …     return x
               ^
SyntaxError: invalid syntax
```

При такой непосредственной вставке интерпретатор сбивают с толку лишние символы приглашения к вводу. Но не волнуйтесь: «магическая» функция оболочки IPython `%paste` разработана специально для корректной обработки данного типа многострочного ввода:

```
In [3]: %paste
>>> def donothing(x):
…     return x
## -- Конец вставленного текста –
```

Команда `%paste` не только вводит, но и выполняет код, так что теперь наша функция готова к использованию:

```
In [4]: donothing(10)
Out[4]: 10
```

Аналогично назначение команды `%cpaste`, открывающей интерактивную многострочную командную строку, в которую можно вставлять один или несколько фрагментов кода для пакетного выполнения:

```
In [5]: %cpaste
Pasting code; enter '--' alone on the line to stop or use Ctrl-D.
:>>> def donothing(x):
:…     return x
:--
```

Эти «магические» команды, подобно другим, предоставляют функциональность, которая была бы сложна или вообще невозможна при использовании обычного интерпретатора языка Python.

Выполнение внешнего кода: %run

Начав писать более обширный код, вы, вероятно, будете работать как в оболочке IPython для интерактивного исследования, так и в текстовом редакторе для сохранения кода, который вам хотелось бы использовать неоднократно. Будет удобно выполнять этот код не в отдельном окне, а непосредственно в сеансе оболочки IPython. Для этого можно применять «магическую» функцию `%run`.

Например, допустим, что вы создали файл `myscript.py`, содержащий:

```
#-------------------------------------
# Файл: myscript.py

def square(x):
    """square a number"""
    return x ** 2

for N in range(1, 4):
    print(N, "squared is", square(N))
```

Выполнить это в сеансе IPython можно следующим образом:

```
In [6]: %run myscript.py
1 squared is 1
2 squared is 4
3 squared is 9
```

Обратите внимание, что после выполнения этого сценария все описанные в нем функции становятся доступными для использования в данном сеансе оболочки IPython:

```
In [7]: square(5)
Out[7]: 25
```

Существует несколько параметров для точной настройки метода выполнения кода. Посмотреть относящуюся к этому документацию можно обычным способом, набрав команду `%run?` в интерпретаторе IPython.

Длительность выполнения кода: %timeit

Еще один пример полезной «магической» функции — `%timeit`, автоматически определяющая время выполнения следующего за ней однострочного оператора языка Python. Например, если нам нужно определить производительность спискового включения:

```
In [8]: %timeit L = [n ** 2 for n in range(1000)]
1000 loops, best of 3: 325 µs per loop
```

Преимущество использования функции `%timeit` в том, что для коротких команд она автоматически будет выполнять множественные запуски с целью получения максимально надежных результатов. Для многострочных операторов добавление второго знака `%` превратит ее в блочную «магическую» функцию, которая может работать с многострочным вводом. Например, вот эквивалентная конструкция для цикла `for`:

```
In [9]: %%timeit
   …: L = []
   …: for n in range(1000):
   …:     L.append(n ** 2)
   …:
1000 loops, best of 3: 373 µs per loop
```

Сразу же видно, что в данном случае списковое включение происходит примерно на 10% быстрее, чем эквивалентная конструкция для цикла `for`. Мы рассмотрим `%timeit` и другие подходы к мониторингу скорости выполнения и профилированию кода в разделе «Профилирование и мониторинг скорости выполнения кода» этой главы.

Справка по «магическим» функциям: ?, %magic и %lsmagic

Подобно обычным функциям языка программирования Python, у «магических» функций оболочки IPython есть свои инструкции (docstring), и к этой документации можно обратиться обычным способом. Например, чтобы просмотреть документацию по «магической» функции `%timeit`, просто введите следующее:

```
In [10]: %timeit?
```

К документации по другим функциям можно получить доступ аналогичным образом. Для доступа к общему описанию доступных «магических» функций введите команду:

```
In [11]: %magic
```

Для быстрого получения простого списка всех доступных «магических» функций наберите:

```
In [12]: %lsmagic
```

При желании можно описать свою собственную «магическую» функцию. Если вас это интересует, загляните в приведенные в разделе «Дополнительные источники информации об оболочке IPython» данной главы.

История ввода и вывода

Командная оболочка IPython позволяет получать доступ к предыдущим командам с помощью стрелок «вверх» (↑) и «вниз» (↓) или сочетаний клавиш Ctrl+P/Ctrl+N. Дополнительно как в командной оболочке, так и в блокноте IPython дает возможность получать вывод предыдущих команд, а также строковые версии самих этих команд.

Объекты In и Out оболочки IPython

Полагаю, вы уже хорошо знакомы с приглашениями `In[1]:/Out[1]:`, используемыми оболочкой IPython. Они представляют собой не просто изящные украшения, а подсказывают вам способ обратиться к предыдущим вводам и выводам в текущем сеансе. Допустим, вы начали сеанс, который выглядит следующим образом:

```
In [1]: import math

In [2]: math.sin(2)
Out[2]: 0.9092974268256817

In [3]: math.cos(2)
Out[3]: -0.4161468365471424
```

Мы импортировали встроенный пакет `math`, после чего вычислили значение синуса и косинуса числа 2. Вводы и выводы отображаются в командной оболочке с метками `In/Out`, но за этим кроется нечто большее: оболочка IPython на самом деле создает переменные языка Python с именами `In` и `Out`, автоматически обновляемые так, что они отражают историю:

```
In [4]: print(In)
['', 'import math', 'math.sin(2)', 'math.cos(2)', 'print(In)']

In [5]: Out
Out[5]: {2: 0.9092974268256817, 3: -0.4161468365471424}
```

Объект `In` представляет собой список, отслеживающий очередность команд (первый элемент этого списка — «заглушка», чтобы `In[1]` ссылался на первую команду):

```
In [6]: print(In[1])
import math
```

Объект `Out` — не список, а словарь, связывающий ввод с выводом (если таковой есть).

Обратите внимание, что не все операции генерируют вывод: например, операторы `import` и `print` на вывод не влияют. Последнее обстоятельство может показаться странным, но смысл его становится понятен, если знать, что функция `print` возвращает `None`; для краткости, возвращающие `None` команды не вносят вклада в объект `Out`.

Это может оказаться полезным при необходимости применять ранее полученные результаты. Например, вычислим сумму `sin(2) ** 2` и `cos(2) ** 2`, используя найденные ранее значения:

```
In [8]: Out[2] ** 2 + Out[3] ** 2
Out[8]: 1.0
```

Результат равен `1.0`, как и можно было ожидать из хорошо известного тригонометрического тождества. В данном случае использовать ранее полученные результаты, вероятно, не было необходимости, но эта возможность может оказаться очень полезной, когда после выполнения ресурсоемких вычислений следует применить их результат повторно!

Быстрый доступ к предыдущим выводам с помощью знака подчеркивания

В обычной командной оболочке Python имеется лишь одна функция быстрого доступа к предыдущему выводу: значение переменной _ (одиночный символ подчеркивания) соответствует предыдущему выводу; это работает и в оболочке IPython:

```
In [9]: print(_)
1.0
```

Но IPython несколько более продвинут в этом смысле: можно использовать двойной символ подчеркивания для доступа к выводу на шаг ранее и тройной — для предшествовавшего ему (не считая команд, не генерировавших никакого вывода):

```
In [10]: print(__)
-0.4161468365471424
```

```
In [11]: print(___)
0.9092974268256817
```

На этом оболочка IPython останавливается: более чем три подчеркивания уже немного сложно отсчитывать и на этом этапе проще ссылаться на вывод по номеру строки.

Однако существует еще одна функция быстрого доступа, заслуживающая упоминания: сокращенная форма записи для `Out[X]` выглядит как `_X` (отдельное подчеркивание с последующим номером строки):

```
In [12]: Out[2]
Out[12]: 0.9092974268256817

In [13]: _2
Out[13]: 0.9092974268256817
```

Подавление вывода

Иногда может понадобиться подавить вывод оператора (чаще всего это случается с командами рисования графиков, которые мы рассмотрим в главе 4). Бывает, что выполняемая команда выводит результат, который вам не хотелось бы сохранять в истории команд, например, чтобы соответствующий ресурс можно было освободить после удаления других ссылок. Простейший способ подавления вывода команды — добавить точку с запятой в конце строки:

```
In [14]: math.sin(2) + math.cos(2);
```

Обратите внимание, что результат при этом вычисляется «молча» и вывод не отображается на экране и не сохраняется в словаре `Out`:

```
In [15]: 14 in Out
Out[15]: False
```

Соответствующие «магические» команды

Для доступа сразу к нескольким предыдущим вводам весьма удобно использовать «магическую» команду `%history`. Так можно вывести первые четыре ввода:

```
In [16]: %history -n 1-4
   1: import math
   2: math.sin(2)
   3: math.cos(2)
   4: print(In)
```

Как обычно, вы можете набрать команду `%history?` для получения дополнительной информации об этой команде и описания доступных параметров. Другие аналогичные «магические» команды — `%rerun` (выполняющая повторно какую-либо

часть истории команд) и `%save` (сохраняющая какую-либо часть истории команд в файле). Для получения дополнительной информации рекомендую изучить их с помощью справочной функциональности `?`, обсуждавшейся в разделе «Справки и документация в Python» данной главы.

Оболочка IPython и использование системного командного процессора

При интерактивной работе со стандартным интерпретатором языка Python вы столкнетесь с досадным неудобством в виде необходимости переключаться между несколькими окнами для обращения к инструментам Python и системным утилитам командной строки. Оболочка IPython исправляет эту ситуацию, предоставляя пользователям синтаксис для выполнения инструкций системного командного процессора непосредственно из терминала IPython. Для этого используется восклицательный знак: все, что находится после `!` в строке, будет выполняться не ядром языка Python, а системной командной строкой.

Далее в этом разделе предполагается, что вы работаете в Unix-подобной операционной системе, например Linux или Mac OS X. Некоторые из следующих примеров не будут работать в операционной системе Windows, использующей по умолчанию другой тип командного процессора (хотя после анонса в 2016 году нативной командной оболочки Bash на Windows в ближайшем будущем это может перестать быть проблемой!). Если инструкции системного командного процессора вам не знакомы, рекомендую просмотреть руководство по нему (http://swcarpentry.github.io/shell-novice/), составленное фондом Software Carpentry.

Краткое введение в использование командного процессора

Полный вводный курс использования командного процессора/терминала/командной строки выходит за пределы данной главы, но непосвященных мы кратко познакомим с ним. Командный процессор — способ текстового взаимодействия с компьютером. Начиная с середины 1980-х годов, когда корпорации Microsoft и Apple представили первые версии их графических операционных систем, большинство пользователей компьютеров взаимодействуют со своей операционной системой посредством привычных щелчков кнопкой мыши на меню и движений «перетаскивания». Но операционные системы существовали задолго до этих графических интерфейсов пользователя и управлялись в основном посредством последовательностей текстового ввода: в приглашении командной строки пользователь вводил команду, а компьютер выполнял указанное пользователем действие. Эти первые системы командной строки были предшественниками командных процессоров

и терминалов, используемых до сих пор наиболее деятельными специалистами по науке о данных.

Человек, не знакомый с командными процессорами, мог бы задать вопрос: зачем вообще тратить на это время, если можно многого добиться с помощью простых нажатий на пиктограммы и меню? Пользователь командного процессора мог бы ответить на этот вопрос другим вопросом: зачем гоняться за пиктограммами и щелкать по меню, если можно добиться того же гораздо проще, с помощью ввода команд? Хотя это может показаться типичным вопросом предпочтений, при выходе за пределы простых задач быстро становится понятно, что командный процессор предоставляет неизмеримо больше возможностей управления для сложных задач.

В качестве примера приведу фрагмент сеанса командного процессора операционной системы Linux/OS X, в котором пользователь просматривает, создает и меняет каталоги и файлы в своей системе (`osx:~ $` представляет собой приглашение ко вводу, а все после знака `$` — набираемая команда; текст, перед которым указан символ `#`, представляет собой просто описание, а не действительно вводимый вами текст):

```
osx:~ $ echo "hello world"              # echo аналогично функции print
                                        # языка Python
hello world

osx:~ $ pwd                             # pwd = вывести рабочий каталог
/home/jake                              # это «путь», где мы находимся

osx:~ $ ls                              # ls = вывести содержимое
                                        # рабочего каталога
notebooks  projects

osx:~ $ cd projects/                    # cd = сменить каталог

osx:projects $ pwd
/home/jake/projects

osx:projects $ ls
datasci_book   mpld3   myproject.txt

osx:projects $ mkdir myproject          # mkdir = создать новый каталог

osx:projects $ cd myproject/

osx:myproject $ mv ../myproject.txt ./  # mv = переместить файл.
                                        # В данном случае мы перемещаем
                                        # файл myproject.txt, находящийся
                                        # в каталоге на уровень выше (../),
                                        # в текущий каталог (./)
osx:myproject $ ls
```

```
myproject.txt
```

Обратите внимание, что все это всего лишь краткий способ выполнения привычных операций (навигации по дереву каталогов, создания каталога, перемещения файла и т. д.) путем набора команд вместо щелчков по пиктограммам и меню. Кроме того, с помощью всего нескольких команд (`pwd`, `ls`, `cd`, `mkdir` и `cp`) можно выполнить большинство распространенных операций с файлами. А уж когда вы выходите за эти простейшие операции, подход с использованием командного процессора открывает по-настоящему широкие возможности.

Инструкции командного процессора в оболочке IPython

Вы можете использовать любую работающую в командной строке команду в оболочке IPython, просто поставив перед ней символ `!`. Например, команды `ls`, `pwd` и `echo` можно выполнить следующим образом:

```
In [1]: !ls
myproject.txt

In [2]: !pwd
/home/jake/projects/myproject

In [3]: !echo "printing from the shell"
printing from the shell
```

Передача значений в командный процессор и из него

Инструкции командного процессора можно не только выполнять из оболочки IPython, они могут также взаимодействовать с пространством имен IPython. Например, можно сохранить вывод любой инструкции командного процессора с помощью оператора присваивания:

```
In [4]: contents = !ls

In [5]: print(contents)
['myproject.txt']

In [6]: directory = !pwd

In [7]: print(directory)
['/Users/jakevdp/notebooks/tmp/myproject']
```

Обратите внимание, что эти результаты возвращаются не в виде списков, а как специальный определенный в IPython для командного процессора тип возвращаемого значения:

```
In [8]: type(directory)
IPython.utils.text.Slist
```

Этот тип выглядит и работает во многом подобно спискам языка Python, но у него есть и дополнительная функциональность, в частности методы `grep` и `fields`, а также свойства `s`, `n` и `p`, позволяющие выполнять поиск, фильтрацию и отображение результатов удобным способом. Чтобы узнать об этом больше, воспользуйтесь встроенными справочными возможностями оболочки IPython.

Отправка информации в обратную сторону — передача переменных Python в командный процессор — возможна посредством синтаксиса `{varname}`:

```
In [9]: message = "Hello from Python"

In [10]: !echo {message}
Hello from Python
```

Фигурные скобки содержат имя переменной, заменяемое в инструкции командного процессора ее значением.

«Магические» команды для командного процессора

Поэкспериментировав немного с инструкциями командного процессора, вы можете обратить внимание на то, что использовать команду `!cd` для навигации по файловой системе не получается:

```
In [11]: !pwd
/home/jake/projects/myproject

In [12]: !cd ..

In [13]: !pwd
/home/jake/projects/myproject
```

Причина заключается в том, что инструкции командного процессора в блокноте оболочки IPython выполняются во временном командном подпроцессоре. Если вам нужно поменять рабочий каталог на постоянной основе, можно воспользоваться «магической» командой `%cd`:

```
In [14]: %cd ..
/home/jake/projects
```

На самом деле по умолчанию ее можно использовать даже без символа %:

```
In [15]: cd myproject
/home/jake/projects/myproject
```

Такое поведение носит название *«автомагических»* (automagic) функций, его можно настраивать с помощью «магической» функции `%automagic`.

Кроме `%cd`, доступны и другие «автомагические» функции: `%cat`, `%cp`, `%env`, `%ls`, `%man`, `%mkdir`, `%more`, `%mv`, `%pwd`, `%rm` и `%rmdir`, каждую из которых можно применять без знака %, если активизировано поведение `automagic`. При этом командную строку оболочки IPython можно использовать практически как обычный командный процессор:

```
In [16]: mkdir tmp

In [17]: ls
myproject.txt  tmp/

In [18]: cp myproject.txt tmp/

In [19]: ls tmp
myproject.txt

In [20]: rm –r tmp
```

Доступ к командному процессору из того же окна терминала, в котором происходит сеанс работы с языком Python, означает резкое снижение числа необходимых переключений между интерпретатором и командным процессором при написании кода на языке Python.

Ошибки и отладка

Разработка кода и анализ данных всегда требуют некоторого количества проб и ошибок, и в оболочке IPython есть инструменты для упрощения этого процесса. В данном разделе будут вкратце рассмотрены некоторые возможности по управлению оповещением об ошибках Python, а также утилиты для отладки ошибок в коде.

Управление исключениями: %xmode

Почти всегда при сбое сценария на языке Python генерируется исключение. В случае столкновения интерпретатора с одним из этих исключений, информацию о его причине можно найти в *трассировке* (traceback), к которой можно обратиться из Python. С помощью «магической» функции `%xmode` оболочка IPython дает вам

возможность управлять количеством выводимой при генерации исключения информации. Рассмотрим следующий код:

```
In[1]: def func1(a, b):
           return a / b
       def func2(x):
           a = x
           b = x - 1
           return func1(a, b)

In[2]: func2(1)
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-2-b2e110f6fc8f^gt; in <module>()
----> 1 func2(1)

<ipython-input-1-d849e34d61fb> in func2(x)
      5     a = x
      6     b = x - 1
----> 7     return func1(a, b)

<ipython-input-1-d849e34d61fb> in func1(a, b)
      1 def func1(a, b):
----> 2     return a / b
      3
      4 def func2(x):
      5     a = x

ZeroDivisionError: division by zero
```

Вызов функции `func2` приводит к ошибке, и чтение выведенной трассы позволяет нам в точности понять, что произошло. По умолчанию эта трасса включает несколько строк, описывающих контекст каждого из приведших к ошибке шагов. С помощью «магической» функции `%xmode` (сокращение от exception mode — *режим отображения исключений*) мы можем управлять тем, какая информация будет выведена.

Функция `%xmode` принимает на входе один аргумент, режим, для которого есть три значения: `Plain` (Простой), `Context` (По контексту) и `Verbose` (Расширенный). Режим по умолчанию — `Context`, вывод при котором показан выше. Режим `Plain` дает более сжатый вывод и меньше информации:

```
In[3]: %xmode Plain

Exception reporting mode: Plain

In[4]: func2(1)
```

```
---------------------------------------------------------------------------

Traceback (most recent call last):


  File "<ipython-input-4-b2e110f6fc8f>", line 1, in <module>
    func2(1)


  File "<ipython-input-1-d849e34d61fb>", line 7, in func2
    return func1(a, b)


  File "<ipython-input-1-d849e34d61fb>", line 2, in func1
    return a / b

ZeroDivisionError: division by zero
```

Режим `Verbose` добавляет еще некоторую информацию, включая аргументы для всех вызываемых функций:

```
In[5]: %xmode Verbose

Exception reporting mode: Verbose

In[6]: func2(1)
---------------------------------------------------------------------------

ZeroDivisionError                         Traceback (most recent call last)

<ipython-input-6-b2e110f6fc8f> in <module>()
----> 1 func2(1)
        global func2 = <function func2 at 0x103729320>

<ipython-input-1-d849e34d61fb> in func2(x=1)
      5     a = x
      6     b = x - 1
----> 7     return func1(a, b)
        global func1 = <function func1 at 0x1037294d0>
        a = 1
        b = 0

<ipython-input-1-d849e34d61fb> in func1(a=1, b=0)
      1 def func1(a, b):
----> 2     return a / b
        a = 1
        b = 0
      3
      4 def func2(x):
      5     a = x
ZeroDivisionError: division by zero
```

Эта дополнительная информация может помочь сузить круг возможных причин генерации исключения. Почему бы тогда не использовать режим `Verbose` всегда? Дело в том, что при усложнении кода такой вид трассировки может стать чрезвычайно длинным. В зависимости от контекста иногда проще работать с более кратким выводом режима по умолчанию.

Отладка: что делать, если чтения трассировок недостаточно

Стандартная утилита языка Python для интерактивной отладки называется `pdb` (сокращение от Python Debugger — «отладчик Python»). Этот отладчик предоставляет пользователю возможность выполнять код строка за строкой, чтобы выяснить, что могло стать причиной какой-либо замысловатой ошибки. Расширенная версия этого отладчика оболочки IPython называется `ipdb` (сокращение от IPython Debugger — «отладчик IPython»).

Существует множество способов запуска и использования этих отладчиков; мы не станем описывать их все. Для более полной информации по данным утилитам обратитесь к онлайн-документации.

Вероятно, наиболее удобный интерфейс для отладки в IPython — «магическая» команда `%debug`. Если ее вызвать после столкновения с исключением, она автоматически откроет интерактивную командную строку откладки в точке возникновения исключения. Командная строка `ipdb` позволяет изучать текущее состояние стека, доступные переменные и даже выполнять команды Python!

Посмотрим на самые недавние исключения, затем выполним несколько простейших действий — выведем значения `a` и `b`, после чего наберем `quit` для выхода из сеанса отладки:

```
In[7]: %debug

> <ipython-input-1-d849e34d61fb>(2)func1()
      1 def func1(a, b):
----> 2     return a / b
      3

ipdb> print(a)
1
ipdb> print(b)
0
ipdb> quit
```

Однако интерактивный отладчик позволяет делать не только это — можно пошагово двигаться вверх и вниз по стеку, изучая значения переменных:

```
In[8]: %debug

> <ipython-input-1-d849e34d61fb>(2)func1()
      1 def func1(a, b):
----> 2     return a / b
      3

ipdb> up
> <ipython-input-1-d849e34d61fb>(7)func2()
      5     a = x
      6     b = x - 1
----> 7     return func1(a, b)

ipdb> print(x)
1
ipdb> up
> <ipython-input-6-b2e110f6fc8f>(1)<module>()
----> 1 func2(1)

ipdb> down
> <ipython-input-1-d849e34d61fb>(7)func2()
      5     a = x
      6     b = x - 1
----> 7     return func1(a, b)

ipdb> quit
```

Это позволяет быстро находить не только *что* вызвало ошибку, но и какие вызовы функций привели к ней.

Если вам необходимо, чтобы отладчик запускался автоматически при генерации исключения, можно воспользоваться «магической» функцией `%pdb` для включения такого автоматического поведения:

```
In[9]: %xmode Plain
       %pdb on
       func2(1)

Exception reporting mode: Plain
Automatic pdb calling has been turned ON


Traceback (most recent call last):


  File "<ipython-input-9-569a67d2d312>", line 3, in <module>
    func2(1)


  File "<ipython-input-1-d849e34d61fb>", line 7, in func2
    return func1(a, b)
```

```
  File "<ipython-input-1-d849e34d61fb>", line 2, in func1
    return a / b

ZeroDivisionError: division by zero


> <Ipython-input-1-d849e34d61fb>(2)func1()
      1 def func1(a, b):
----> 2     return a / b
      3
ipdb> print(b)
0
ipdb> quit
```

Наконец, если у вас есть сценарий, который вы хотели бы запускать в начале работы в интерактивном режиме, можно запустить его с помощью команды `%run -d` и использовать команду `next` для пошагового интерактивного перемещения по строкам кода.

Неполный список команд отладки. Для интерактивной отладки доступно намного больше команд, чем мы перечислили (табл. 1.5).

Таблица 1.5. Наиболее часто используемые команды

Команда	Описание
list	Отображает текущее место в файле
h(elp)	Отображает список команд или справку по конкретной команде
q(uit)	Выход из отладчика и программы
c(ontinue)	Выход из отладчика, продолжение выполнения программы
n(ext)	Переход к следующему шагу программы
<enter>	Повтор предыдущей команды
p(rint)	Вывод значений переменных
s(tep)	Вход в подпрограмму
r(eturn)	Возврат из подпрограммы

Для дальнейшей информации воспользуйтесь командой `help` в отладчике или загляните в онлайн-документацию по `ipdb` (https://github.com/gotcha/ipdb).

Профилирование и мониторинг скорости выполнения кода

В процессе разработки кода и создания конвейеров обработки данных всегда присутствуют компромиссы между различными реализациями. В начале создания алгоритма забота о подобных вещах может оказаться контрпродуктивной. Согласно

знаменитому афоризму Дональда Кнута: «Лучше не держать в голове подобные "малые" вопросы производительности, скажем, 97 % времени: преждевременная оптимизация — корень всех зол».

Однако, как только ваш код начинает работать, будет полезно немного заняться его производительностью. Иногда бывает удобно проверить время выполнения заданной команды или набора команд, а иногда — покопаться в состоящем из множества строк процессе и выяснить, где находится узкое место какого-либо сложного набора операций. Оболочка IPython предоставляет широкий выбор функциональности для выполнения подобного мониторинга скорости выполнения кода и его профилирования. Здесь мы обсудим следующие «магические» команды оболочки IPython:

- `%time` — длительность выполнения отдельного оператора;
- `%timeit` — длительность выполнения отдельного оператора при неоднократном повторе, для большей точности;
- `%prun` — выполнение кода с использованием профилировщика;
- `%lprun` — пошаговое выполнение кода с применением профилировщика;
- `%memit` — оценка использования оперативной памяти для отдельного оператора;
- `%mprun` — пошаговое выполнение кода с применением профилировщика памяти.

Последние четыре команды не включены в пакет IPython — для их использования необходимо установить расширения `line_profiler` и `memory_profiler`, которые мы обсудим в следующих разделах.

Оценка времени выполнения фрагментов кода: %timeit и %time

Мы уже встречались со строчной «магической» командой `%timeit` и блочной «магической» командой `%%timeit` во введении в «магические» функции в разделе «"Магические" команды IPython» данной главы; команду `%%timeit` можно использовать для оценки времени многократного выполнения фрагментов кода:

```
In[1]: %timeit sum(range(100))

100000 loops, best of 3: 1.54 µs per loop
```

Обратите внимание, что, поскольку данная операция должна выполняться очень быстро, команда `%timeit` автоматически выполняет ее большое количество раз. В случае более медленных команд команда `%timeit` автоматически подстроится и будет выполнять их меньшее количество раз:

```
In[2]: %%timeit
       total = 0
       for i in range(1000):
           for j in range(1000):
               total += i * (-1) ** j

1 loops, best of 3: 407 ms per loop
```

Иногда повторное выполнение операции — не лучший вариант. Например, в случае необходимости отсортировать список повтор операции мог бы ввести нас в заблуждение. Сортировка предварительно отсортированного списка происходит намного быстрее, чем сортировка неотсортированного, так что повторное выполнение исказит результат:

```
In[3]: import random
       L = [random.random() for i in range(100000)]
       %timeit L.sort()

100 loops, best of 3: 1.9 ms per loop
```

В этом случае лучшим выбором будет «магическая» функция `%timeit`. Она также подойдет для команд с более длительным временем выполнения, в случае которых короткие системно обусловленные задержки вряд ли существенно повлияют на результат. Оценим время сортировки неотсортированного и предварительно отсортированного списков:

```
In[4]: import random
       L = [random.random() for i in range(100000)]
       print("sorting an unsorted list:")
       %time L.sort()

sorting an unsorted list:
CPU times: user 40.6 ms, sys: 896 µs, total: 41.5 ms
Wall time: 41.5 ms

In[5]: print("sorting an already sorted list:")
       %time L.sort()
sorting an already sorted list:
CPU times: user 8.18 ms, sys: 10 µs, total: 8.19 ms
Wall time: 8.24 ms
```

Обратите внимание на то, насколько быстрее сортируется предварительно отсортированный список, а также насколько больше времени занимает оценка с помощью функции `%time` по сравнению с функцией `%timeit`, даже в случае предварительно отсортированного списка! Это происходит в результате того, что «магическая» функция `%timeit` незаметно для нас осуществляет некоторые хитрые трюки, чтобы системные вызовы не мешали оценке времени. Например, она не допускает удаления неиспользуемых объектов Python (известного как *сбор мусора*), которое могло бы повлиять на оценку времени. Именно поэтому выдаваемое функцией `%timeit` время обычно заметно короче выдаваемого функцией `%time`.

В случае как `%time`, так и `%timeit` использование синтаксиса с двойным знаком процента блочной «магической» функции дает возможность оценивать время выполнения многострочных сценариев:

```
In[6]: %%time
       total = 0
       for i in range(1000):
           for j in range(1000):
               total += i * (-1) ** j

CPU times: user 504 ms, sys: 979 µs, total: 505 ms
Wall time: 505 ms
```

Для получения дальнейшей информации по «магическим» функциям `%time` и `%timeit`, а также их параметрам воспользуйтесь справочными функциями оболочки IPython (то есть наберите `%time?` в командной строке IPython).

Профилирование сценариев целиком: %prun

Программы состоят из множества отдельных операторов, и иногда оценка времени их выполнения в контексте важнее, чем по отдельности. В языке Python имеется встроенный профилировщик кода (о котором можно прочитать в документации языка Python), но оболочка IPython предоставляет намного более удобный способ его использования в виде «магической» функции `%prun`.

В качестве примера я опишу простую функцию, выполняющую определенные вычисления:

```
In[7]: def sum_of_lists(N):
           total = 0
           for i in range(5):
               L = [j ^ (j >> i) for j in range(N)]
               total += sum(L)
           return total
```

Теперь мы можем обратиться к «магической» функции `%prun` с указанием вызова функции, чтобы увидеть результаты профилирования:

```
In[8]: %prun sum_of_lists(1000000)
```

В блокноте результат будет выведен в пейджер[1] и будет выглядеть следующим образом:

```
14 function calls in 0.714 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
```

[1] Область вывода в нижней части окна блокнота.

```
     5    0.599    0.120    0.599    0.120 <ipython-input-
                                            19>:4(<listcomp>)
     5    0.064    0.013    0.064    0.013 {built-in method sum}
     1    0.036    0.036    0.699    0.699 <ipython-input-
                                            19>:1(sum_of_lists)
     1    0.014    0.014    0.714    0.714 <string>:1(<module>)
     1    0.000    0.000    0.714    0.714 {built-in method exec}
```

Результат представляет собой таблицу, указывающую в порядке общего затраченного на каждый вызов функции времени, в каких местах выполнение занимает больше всего времени. В данном случае большую часть времени занимает списочное включение внутри функции `sum_of_lists`. Зная это, можно начинать обдумывать возможные изменения в алгоритме для улучшения производительности.

Для получения дополнительной информации по «магической» функции `%prun`, а также доступным для нее параметрам воспользуйтесь справочной функциональностью оболочки IPython (то есть наберите `%prun?` В командной строке IPython).

Пошаговое профилирование с помощью %lprun

Профилирование по функциям с помощью функции `%prun` довольно удобно, но иногда больше пользы может принести построчный отчет профилировщика. Такая функциональность не встроена в язык Python или оболочку IPython, но можно установить пакет `line_profiler`, обладающий такой возможностью. Начнем с использования утилиты языка Python для работы с пакетами, pip, для установки пакета `line_profiler`:

```
$ pip install line_profiler
```

Далее можно воспользоваться IPython для загрузки расширения `line_profiler` оболочки IPython, предоставляемой в виде части указанного пакета:

```
In[9]: %load_ext line_profiler
```

Теперь команда `%lprun` может выполнить построчное профилирование любой функции. В нашем случае необходимо указать ей явным образом, какие функции мы хотели быть профилировать:

```
In[10]: %lprun –f sum_of_lists sum_of_lists(5000)
```

Как и ранее, блокнот отправляет результаты в пейджер, но выглядят они так:

```
Timer unit: 1e-06 s

Total time: 0.009382 s File: <ipython-input-19-fa2be176cc3e>
Function: sum_of_lists at line 1

Line #      Hits       Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def sum_of_lists(N):
     2         1          2      2.0      0.0      total = 0
```

```
    3         6           8      1.3      0.1      for i in range(5):
    4         5        9001   1800.2     95.9          L = [j ^ (j >> i)...
    5         5         371     74.2      4.0          total += sum(L)
    6         1           0      0.0      0.0      return total
```

Информация в заголовке дает нам ключ к чтению результатов: время указывается в микросекундах, и мы можем увидеть, в каком месте выполнение программы занимает наибольшее количество времени. На этой стадии мы получаем возможность применить эту информацию для модификации кода и улучшения его производительности для желаемого сценария использования.

Для получения дополнительной информации о «магической» функции `%lprun`, а также о доступных для нее параметрах воспользуйтесь справочной функциональностью оболочки IPython (то есть наберите `%lprun?` в командной строке IPython).

Профилирование использования памяти: %memit и %mprun

Другой аспект профилирования — количество используемой операциями памяти. Это количество можно оценить с помощью еще одного расширения оболочки IPython — `memory_profiler`. Как и в случае с утилитой `line_profiler`, мы начнем с установки расширения с помощью утилиты `pip`:

```
$ pip install memory_profiler
```

Затем можно воспользоваться оболочкой IPython для загрузки этого расширения:

```
In[12]: %load_ext memory_profiler
```

Расширение профилировщика памяти содержит две удобные «магические» функции: `%memit` (аналог `%timeit` для измерения количества памяти) и `%mprun` (аналог `%lprun` для измерения количества памяти). Применять функцию `%memit` несложно:

```
In[13]: %memit sum_of_lists(1000000)

peak memory: 100.08 MiB, increment: 61.36 MiB
```

Мы видим, что данная функция использует около 100 Мбайт памяти.

Для построчного описания применения памяти можно использовать «магическую» функцию `%mprun`. К сожалению, она работает только для функций, описанных в отдельных модулях, а не в самом блокноте, так что начнем с применения «магической» функции `%%file` для создания простого модуля под названием `mprun_demo.py`, содержащего нашу функцию `sum_of_lists`, с одним дополнением, которое немного прояснит нам результаты профилирования памяти:

```
In[14]: %%file mprun_demo.py
        def sum_of_lists(N):
            total = 0
            for i in range(5):
            L = [j ^ (j >> i) for j in range(N)]
            total += sum(L)
            del L # Удалить ссылку на L
            return total

Overwriting mprun_demo.py
```

Теперь мы можем импортировать новую версию нашей функции и запустить построчный профилировщик памяти:

```
In[15]: from mprun_demo import sum_of_lists
        %mprun -f sum_of_lists sum_of_lists(1000000)
```

Результат, выведенный в пейджер, представляет собой отчет об использовании памяти этой функцией и выглядит следующим образом:

```
Filename: ./mprun_demo.py

Line #    Mem usage    Increment   Line Contents
================================================================
     4     71.9 MiB      0.0 MiB   L = [j ^ (j >> i) for j in range(N)]

Filename: ./mprun_demo.py

Line #    Mem usage    Increment   Line Contents
================================================================
     1     39.0 MiB      0.0 MiB   def sum_of_lists(N):
     2     39.0 MiB      0.0 MiB       total = 0
     3     46.5 MiB      7.5 MiB       for i in range(5):
     4     71.9 MiB     25.4 MiB           L = [j ^ (j >> i)
                                           for j in range(N)]
     5     71.9 MiB      0.0 MiB           total += sum(L)
     6     46.5 MiB    -25.4 MiB           del L # Удалить ссылку на L
     7     39.1 MiB     -7.4 MiB       return total
```

Здесь столбец `Increment` сообщает нам, насколько каждая строка отражается в общем объеме памяти. Обратите внимание, что при создании и удалении списка `L`, помимо фонового использования памяти самим интерпретатором языка Python, использование памяти увеличивается примерно на 25 Мбайт.

Для получения дополнительной информации о «магических» функциях `%memit` и `%mprun`, а также о доступных для них параметрах воспользуйтесь справочной функциональностью оболочки IPython (то есть наберите `%memit?` в командной строке IPython).

Дополнительные источники информации об оболочке IPython

В данной главе мы захватили лишь верхушку айсберга по использованию языка Python для задач науки о данных. Гораздо больше информации доступно как в печатном виде, так и в Интернете. Здесь мы приведем некоторые дополнительные ресурсы, которые могут вам пригодиться.

Веб-ресурсы

- *Сайт IPython* (http://ipython.org/). Сайт IPython содержит ссылки на документацию, примеры, руководства и множество других ресурсов.
- *Сайт nbviewer* (http://nbviewer.ipython.org/). Этот сайт демонстрирует статические визуализации всех доступных в Интернете блокнотов оболочки IPython. Главная его страница показывает несколько примеров блокнотов, которые можно пролистать, чтобы увидеть, для чего другие разработчики используют язык Python!
- *Галерея интересных блокнотов оболочки IPython* (http://github.com/ipython/ipython/wiki/A-gallery-of-interesting-ipython-Notebooks/). Этот непрерывно растущий список блокнотов, поддерживаемый `nbviewer`, демонстрирует глубину и размах численного анализа, возможного с помощью оболочки IPython. Он включает все, начиная от коротких примеров и руководств и заканчивая полноразмерными курсами и книгами, составленными в формате блокнотов!
- *Видеоруководства*. В Интернете вы найдете немало видеоруководств по оболочке IPython. Особенно рекомендую руководства Фернандо Переса и Брайана Грейнджера — двух основных разработчиков, создавших и поддерживающих оболочку IPython и проект Jupiter, — с конференций PyCon, SciPy and PyData.

Книги

- *Python for Data Analysis*[1] (http://bit.ly/python-for-data-analysis). Эта книга Уэса Маккинли включает главу, описывающую использование оболочки IPython с точки зрения исследователя данных. Хотя многое из ее материала пересекается с тем, что мы тут обсуждали, вторая точка зрения никогда не помешает.
- *Learning IPython for Interactive Computing and Data Visualization* (*«Изучаем оболочку IPython для целей интерактивных вычислений и визуализации данных»*, http://bit.ly/2eLCBB7). Эта книга Цириллы Россан предлагает неплохое введение в использование оболочки IPython для анализа данных.

[1] *Маккинли У.* Python и анализ данных. — М.: ДМК-Пресс, 2015.

- *IPython Interactive Computing and Visualization Cookbook* (*«Справочник по интерактивным вычислениям и визуализации с помощью языка IPython»*, http://bit.ly/2fCEtNE). Данная книга также написана Цириллой Россан и представляет собой более длинное и продвинутое руководство по использованию оболочки IPython для науки о данных. Несмотря на название, она посвящена не только оболочке IPython, в книге рассмотрены некоторые детали широкого спектра проблем науки о данных.

Вы можете и сами найти справочные материалы: основанная на символе `?` справочная функциональность оболочки IPython (обсуждавшаяся в разделе «Справка и документация в оболочке Python» этой главы) может оказаться чрезвычайно полезной, если ее использовать правильно. При работе с примерами из данной книги и другими применяйте ее для знакомства со всеми утилитами, которые предоставляет IPython.

2 Введение в библиотеку NumPy

В этой главе мы рассмотрим методы эффективной загрузки, хранения и управления данными в языке Python, находящимися в оперативной памяти. Тематика очень широка, ведь наборы данных могут поступать из разных источников и быть различных форматов (наборы документов, изображений, аудиоклипов, наборы численных измерений и др.). Несмотря на столь очевидную разнородность, все эти данные удобно рассматривать как массивы числовых значений.

Изображения (особенно цифровые) можно представить в виде простых двумерных массивов чисел, отражающих яркость пикселов соответствующей области; аудиоклипы — как одномерные массивы интенсивности звука в соответствующие моменты времени. Текст можно преобразовать в числовое представление различными путями, например с двоичными числами, представляющими частоту определенных слов или пар слов. Вне зависимости от характера данных первым шагом к их анализу является преобразование в числовые массивы (мы обсудим некоторые примеры этого процесса в разделе «Проектирование признаков» главы 5).

Эффективное хранение и работа с числовыми массивами очень важны для процесса исследования данных. Мы изучим специализированные инструменты языка Python, предназначенные для обработки подобных массивов, — пакет NumPy и пакет Pandas (см. главу 3).

Библиотека NumPy (сокращение от Numerical Python — «числовой Python») обеспечивает эффективный интерфейс для хранения и работы с плотными буферами данных. Массивы библиотеки NumPy похожи на встроенный тип данных языка Python `list`, но обеспечивают гораздо более эффективное хранение и операции с данными при росте размера массивов. Массивы библиотеки NumPy формируют ядро практически всей экосистемы утилит для исследования данных Python. Время, проведенное за изучением способов эффективного использования пакета

NumPy, не будет потрачено впустую вне зависимости от интересующего вас аспекта науки о данных.

Если вы последовали приведенному в предисловии совету и установили стек утилит Anaconda, то пакет NumPy уже готов к использованию. Если же вы относитесь к числу тех, кто любит все делать своими руками, то перейдите на сайт NumPy и следуйте имеющимся там указаниям. После этого импортируйте NumPy и тщательно проверьте его версию:

```
In[1]: import numpy
       numpy.__version__


Out[1]: '1.11.1'
```

Для используемых здесь частей этого пакета я рекомендовал бы применять NumPy версии 1.8 или более поздней. По традиции, большинство людей в мире SciPy/PyData импортируют пакет NumPy, используя в качестве его псевдонима `np`:

```
In[2]: import numpy as np
```

На протяжении книги мы будем именно так импортировать и применять NumPy.

Напоминание о встроенной документации

Читая данную главу, не забывайте, что оболочка IPython позволяет быстро просматривать содержимое пакетов (посредством автодополнения при нажатии клавиши Tab), а также документацию по различным функциям (используя символ `?`). Загляните в раздел «Справка и документация в оболочке Python» главы 1, если вам нужно освежить в памяти эти возможности.

Например, для отображения всего содержимого пространства имен `numpy` можете ввести команду:

```
In [3]: np.<TAB>
```

Для отображения встроенной документации пакета NumPy воспользуйтесь командой:

```
In [4]: np?
```

Более подробную документацию, а также руководства и другие источники информации можно найти на сайте http://www.numpy.org.

Работа с типами данных в языке Python

Чтобы достичь эффективности научных вычислений, ориентированных на работу с данными, следует знать, как хранятся и обрабатываются данные. В этом разделе описываются и сравниваются способы обработки массивов данных в языке Python, а также вносимые в этот процесс библиотекой NumPy усовершенствования. Знание различий очень важно для понимания большей части материала в книге.

Пользователей языка Python зачастую привлекает его простота, немаловажной частью которой является динамическая типизация. В то время как в языках со статической типизацией, таких как C и Java, необходимо явным образом объявлять все переменные, языки с динамической типизацией, например Python, этого не требуют. Например, в языке C можно описать операцию следующим образом:

```
/* Код на языке C */
int result = 0;
for(int i=0; i<100; i++){
    result += i;
}
```

На языке Python соответствующую операцию можно описать так:

```
# Код на языке Python
result = 0
for i in range(100):
    result += i
```

Обратите внимание на главное отличие: в языке C типы данных каждой переменной объявлены явным образом, а в Python они определяются динамически. Это значит, что мы можем присвоить любой переменной данные любого типа:

```
# Код на языке Python
x = 4
x = "four"
```

Здесь мы переключили контекст переменной `x` с целого числа на строку. Подобное действие в языке C могло бы привести к ошибке компиляции или другим неожиданным последствиям в зависимости от настроек компилятора:

```
/* Код на языке C */
int x = 4;
x = "four";  // СБОЙ
```

Подобная гибкость делает Python и другие языки с динамической типизацией удобными и простыми в использовании. Понимать, как это работает, очень важно, чтобы научиться эффективно анализировать данные с помощью языка Python. Однако такая гибкость при работе с типами указывает на то, что переменные Python представляют собой нечто большее, чем просто значение, они содержат также дополнительную информацию о типе значения. Мы рассмотрим это подробнее в следующих разделах.

Целое число в языке Python — больше, чем просто целое число

Стандартная реализация языка Python написана на языке C. Это значит, что каждый объект Python — просто искусно замаскированная структура языка C, со-

держащая не только значение, но и другую информацию. Например, при описании целочисленной переменной на языке Python, такой как `x = 10000`, `x` представляет собой не просто «чистое» целое число. На самом деле это указатель на составную структуру языка C, содержащую несколько значений. Посмотрев в исходный код Python 3.4, можно узнать, что описание целочисленного типа (типа long) фактически выглядит следующим образом (после разворачивания макросов языка C):

```
struct _longobject {
    long ob_refcnt;
    PyTypeObject *ob_type;
    size_t ob_size;
    long ob_digit[1];
};
```

Отдельное целое число в языке Python 3.4 фактически состоит из четырех частей:

- `ob_refcnt` — счетчика ссылок, с помощью которого Python незаметно выполняет выделение и освобождение памяти;
- `ob_type` — кодирующей тип переменной;
- `ob_size` — задающей размер следующих элементов данных;
- `ob_digit` — содержащей фактическое целочисленное значение, которое представляет переменная языка Python.

Это значит, что существует некоторая избыточность при хранении целого числа в языке Python по сравнению с целым числом в компилируемых языках, таких как C (рис. 2.1).

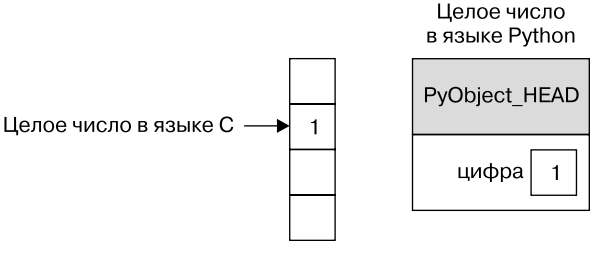

Рис. 2.1. Разница между целыми числами в языках C и Python

`PyObject_HEAD` на этом рисунке — часть структуры, содержащая счетчик ссылок, код типа и другие упомянутые выше элементы.

Еще раз акцентируем внимание на основном отличии: целое число в языке C представляет собой ярлык для места в памяти, байты в котором кодируют целочисленное значение. Целое число в Python — указатель на место в памяти, где хранится вся информация об объекте языка Python, включая байты, содержащие целочисленное значение. Именно эта дополнительная информация в структуре Python для

целого числа и является тем, что позволяет так свободно программировать на языке Python с использованием динамической типизации. Однако эта дополнительная информация в типах Python влечет и накладные расходы, что становится особенно заметно в структурах, объединяющих значительное количество таких объектов.

Список в языке Python — больше, чем просто список

Теперь рассмотрим, что происходит при использовании структуры языка Python, содержащей много объектов. Стандартным изменяемым многоэлементным контейнером в Python является список. Создать список целочисленных значений можно следующим образом:

```
In[1]: L = list(range(10))
       L

Out[1]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In[2]: type(L[0])

Out[2]: int
```

Или аналогичным образом — список строк:

```
In[3]: L2 = [strI for c in L]
       L2

Out[3]: ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']

In[4]: type(L2[0])

Out[4]: str
```

В силу динамической типизации языка Python можно создавать даже неоднородные списки:

```
In[5]: L3 = [True, "2", 3.0, 4]
       [type(item) for item in L3]
```

Out[5]: [bool, str, float, int] Однако подобная гибкость имеет свою цену: для использования гибких типов данных каждый элемент списка должен содержать информацию о типе, счетчик ссылок и другую информацию, то есть каждый элемент представляет собой целый объект языка Python. В частном случае совпадения типа всех переменных большая часть этой информации избыточна: намного рациональнее хранить данные в массиве с фиксированным типом значений. Различие между списком с динамическим типом значений и списком с фиксированным типом (в стиле библиотеки NumPy) проиллюстрировано на рис. 2.2.

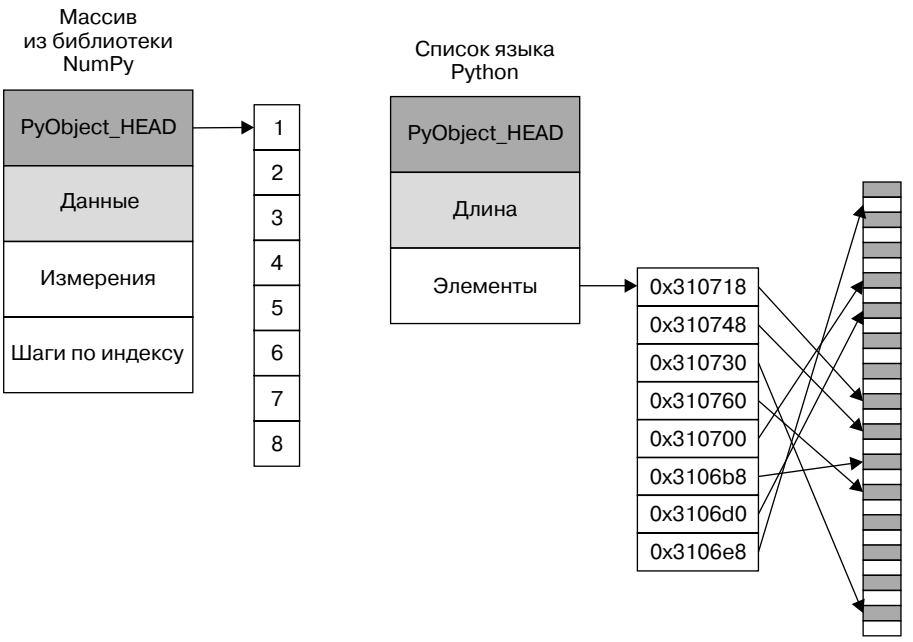

Рис. 2.2. Различие между списками в языках C и Python

На уровне реализации массив фактически содержит один указатель на непрерывный блок данных. Список в языке Python же содержит указатель на блок указателей, каждый из которых, в свою очередь, указывает на целый объект языка Python, например, целое число. Преимущество такого списка состоит в его гибкости: раз каждый элемент списка — полномасштабная структура, содержащая как данные, так и информацию о типе, список можно заполнить данными любого требуемого типа. Массивам с фиксированным типом из библиотеки NumPy недостает этой гибкости, но они гораздо эффективнее хранят данные и работают с ними.

Массивы с фиксированным типом в языке Python

Язык Python предоставляет несколько возможностей для хранения данных в эффективно работающих буферах с фиксированным типом значений. Встроенный модуль `array` (доступен начиная с версии 3.3 языка Python) можно использовать для создания плотных массивов данных одного типа:

```
In[6]: import array
       L = list(range(10))
       A = array.array('i', L)
       A


Out[6]: array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Здесь `'i'` — код типа, указывающий, что содержимое является целыми числами.

Намного удобнее объект `ndarray` из библиотеки NumPy. В то время как объект `array` языка Python обеспечивает эффективное хранение данных в формате массива, библиотека NumPy добавляет еще и возможность выполнения эффективных *операций* над этими данными. Мы рассмотрим такие операции в следующих разделах, а здесь продемонстрируем несколько способов создания NumPy-массива.

Начнем с обычного импорта пакета NumPy под псевдонимом `np`:

```
In[7]: import numpy as np
```

Создание массивов из списков языка Python

Для создания массивов из списков языка Python можно воспользоваться функцией `np.array`:

```
In[8]: # массив целочисленных значений:
       np.array([1, 4, 2, 5, 3])

Out[8]: array([1, 4, 2, 5, 3])
```

В отличие от списков языка Python библиотека NumPy ограничивается массивами, содержащими элементы одного типа. Если типы элементов не совпадают, NumPy попытается выполнить повышающее приведение типов (в данном случае целочисленные значения приводятся к числам с плавающей точкой):

```
In[9]: np.array([3.14, 4, 2, 3])

Out[9]: array([ 3.14,  4.  ,  2.  ,  3.  ])
```

Если же необходимо явным образом задать тип данных для итогового массива, можно воспользоваться ключевым словом `dtype`:

```
In[10]: np.array([1, 2, 3, 4], dtype='float32')

Out[10]: array([ 1.,  2.,  3.,  4.], dtype=float32)
```

Наконец, в отличие от списков в языке Python массивы библиотеки NumPy могут явным образом описываться как многомерные. Вот один из способов задания значений многомерного массива с помощью списка списков:

```
In[11]: # Вложенные списки преобразуются в многомерный массив
        np.array([range(i, i + 3) for i in [2, 4, 6]])
Out[11]: array([[2, 3, 4],
                [4, 5, 6],
                [6, 7, 8]])
```

Вложенные списки рассматриваются как строки в итоговом двумерном массиве.

Создание массивов с нуля

Большие массивы эффективнее создавать с нуля с помощью имеющихся в пакете NumPy методов. Вот несколько примеров:

```
In[12]: # Создаем массив целых чисел длины 10, заполненный нулями
        np.zeros(10, dtype=int)

Out[12]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

In[13]: # Создаем массив размером 3 x 5 значений с плавающей точкой,
        # заполненный единицами
        np.ones((3, 5), dtype=float)

Out[13]: array([[ 1.,  1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  1.,  1.]])

In[14]: # Создаем массив размером 3 x 5, заполненный значением 3.14
        np.full((3, 5), 3.14)

Out[14]: array([[ 3.14,  3.14,  3.14,  3.14,  3.14],
                [ 3.14,  3.14,  3.14,  3.14,  3.14],
                [ 3.14,  3.14,  3.14,  3.14,  3.14]])

In[15]: # Создаем массив, заполненный линейной последовательностью,
        # начинающейся с 0 и заканчивающейся 20, с шагом 2
        # (аналогично встроенной функции range())
        np.arange(0, 20, 2)

Out[15]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])

In[16]: # Создаем массив из пяти значений,
        # равномерно располагающихся между 0 и 1
        np.linspace(0, 1, 5)

Out[16]: array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])

In[17]: # Создаем массив размером 3 x 3 равномерно распределенных
        # случайных значения от 0 до 1
        np.random.random((3, 3))

Out[17]: array([[ 0.99844933,  0.52183819,  0.22421193],
                [ 0.08007488,  0.45429293,  0.20941444],
                [ 0.14360941,  0.96910973,  0.946117  ]])

In[18]: # Создаем массив размером 3 x 3 нормально распределенных
        # случайных значения с медианой 0 и стандартным отклонением 1
        np.random.normal(0, 1, (3, 3))

Out[18]: array([[ 1.51772646,  0.39614948, -0.10634696],
```

```
                [ 0.25671348,  0.00732722,  0.37783601],
                [ 0.68446945,  0.15926039, -0.70744073]])

In[19]: # Создаем массив размером 3 x 3 случайных целых числа
        # в промежутке [0, 10)
        np.random.randint(0, 10, (3, 3))

Out[19]: array([[2, 3, 4],
                [5, 7, 8],
                [0, 5, 0]])

In[20]: # Создаем единичную матрицу размером 3 x 3
        np.eye(3)

Out[20]: array([[ 1.,  0.,  0.],
                [ 0.,  1.,  0.],
                [ 0.,  0.,  1.]])

In[21]: # Создаем неинициализированный массив из трех целочисленных
        # значений. Значениями будут произвольные, случайно оказавшиеся
        # в соответствующих ячейках памяти данные
        np.empty(3)

Out[21]: array([ 1.,  1.,  1.])
```

Стандартные типы данных библиотеки NumPy

Массивы библиотеки NumPy содержат значения простых типов, поэтому важно знать все подробности об этих типах и их ограничениях. Поскольку библиотека NumPy написана[1] на языке C, эти типы будут знакомы пользователям языков C, Fortran и др.

Стандартные типы данных библиотеки NumPy перечислены в табл. 2.1. Обратите внимание, что при создании массива можно указывать их с помощью строк:

```
np.zeros(10, dtype='int16')
```

или соответствующего объекта из библиотеки NumPy:

```
Np.zeros(10, dtype=np.int16)
```

Таблица 2.1. Стандартные типы данных библиотеки NumPy

Тип данных	Описание
bool_	Булев тип (True или False), хранящийся в виде 1 байта
int_	Тип целочисленного значения по умолчанию (аналогичен типу long языка C; обычно int64 или int32)
intc	Идентичен типу int языка C (обычно int32 или int64)

[1] Большей частью.

Тип данных	Описание
intp	Целочисленное значение, используемое для индексов (аналогично типу ssize_t языка C; обычно int32 или int64)
int8	Байтовый тип (от –128 до 127)
int16	Целое число (от –32 768 до 32 767)
int32	Целое число (от –2 147 483 648 до 2 147 483 647)
int64	Целое число (от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807)
uint8	Беззнаковое целое число (от 0 до 255)
uint16	Беззнаковое целое число (от 0 до 65 535)
uint32	Беззнаковое целое число (от 0 до 4 294 967 295)
uint64	Беззнаковое целое число (от 0 до 18 446 744 073 709 551 615)
float_	Сокращение для названия типа float64
float16	Число с плавающей точкой с половинной точностью: 1 бит знак, 5 бит порядок, 10 бит мантисса
float32	Число с плавающей точкой с одинарной точностью: 1 бит знак, 8 бит порядок, 23 бита мантисса
float64	Число с плавающей точкой с удвоенной точностью: 1 бит знак, 11 бит порядок, 52 бита мантисса
complex_	Сокращение для названия типа complex128
complex64	Комплексное число, представленное двумя 32-битными числами
complex128	Комплексное число, представленное двумя 64-битными числами

Возможно задание и более сложных типов, например задание чисел с прямым или обратным порядком байтов. Для получения более подробной информации загляните в документацию по пакету NumPy (http://numpy.org/). Библиотека NumPy также поддерживает составные типы данных, которые мы рассмотрим в разделе «Структурированные данные: структурированные массивы библиотеки NumPy» данной главы.

Введение в массивы библиотеки NumPy

Работа с данными на языке Python — практически синоним работы с массивами библиотеки NumPy: даже более новые утилиты, например библиотека Pandas (см. главу 3), основаны на массивах NumPy. В этом разделе мы рассмотрим несколько примеров использования массивов библиотеки NumPy для доступа к данным и подмассивам, а также срезы, изменения формы и объединения массивов. Хотя демонстрируемые типы операций могут показаться несколько скучными и педантичными, они являются своеобразными «кирпичиками» для множества других примеров из этой книги. Изучите их хорошенько!

Мы рассмотрим несколько категорий простейших операций с массивами:

- *атрибуты массивов* — определение размера, формы, объема занимаемой памяти и типов данных массивов;
- *индексацию массивов* — получение и задание значений отдельных элементов массива;
- *срезы массивов* — получение и задание значений подмассивов внутри большого массива;
- *изменение формы массивов* — изменение формы заданного массива;
- *слияние и разбиение массивов* — объединение нескольких массивов в один и разделение одного массива на несколько.

Атрибуты массивов библиотеки NumPy

Обсудим некоторые атрибуты массивов. Начнем с описания трех массивов случайных чисел: одномерного, двумерного и трехмерного. Воспользуемся генератором случайных чисел библиотеки NumPy, задав для него начальное значение, чтобы гарантировать генерацию одних и тех же массивов при каждом выполнении кода:

```
In[1]: import numpy as np
       np.random.seed(0)  # начальное значение для целей воспроизводимости

       x1 = np.random.randint(10, size=6)          # одномерный массив
       x2 = np.random.randint(10, size=(3, 4))     # двумерный массив
       x3 = np.random.randint(10, size=(3, 4, 5))  # трехмерный массив
```

У каждого из массивов есть атрибуты `ndim` (размерность), `shape` (размер каждого измерения) и `size` (общий размер массива):

```
In[2]: print("x3 ndim: ", x3.ndim)
       print("x3 shape:", x3.shape)
       print("x3 size: ", x3.size)

x3 ndim:  3
x3 shape: (3, 4, 5)
x3 size:  60
```

Еще один полезный атрибут — `dtype`, тип данных массива (который мы уже ранее обсуждали в разделе «Работа с типами данных в языке Python» этой главы):

```
In[3]: print("dtype:", x3.dtype)

dtype: int64
```

Другие атрибуты включают `itemsize`, выводящий размер (в байтах) каждого элемента массива, и `nbytes`, выводящий полный размер массива (в байтах):

```
In[4]: print("itemsize:", x3.itemsize, "bytes")
```

```
print("nbytes:", x3.nbytes, "bytes")
```

```
itemsize: 8 bytes
nbytes: 480 bytes
```

В общем значение атрибута `nbytes` должно быть равно значению атрибута `itemsize`, умноженному на `size`.

Индексация массива: доступ к отдельным элементам

Если вы знакомы с индексацией стандартных списков языка Python, то индексация библиотеки NumPy будет для вас привычной. В одномерном массиве обратиться к *i*-му (считая с 0) значению можно, указав требуемый индекс в квадратных скобках, точно так же, как при работе со списками языка Python:

```
In[5]: x1

Out[5]: array([5, 0, 3, 3, 7, 9])

In[6]: x1[0]

Out[6]: 5

In[7]: x1[4]

Out[7]: 7
```

Для индексации с конца массива можно использовать отрицательные индексы:

```
In[8]: x1[-1]

Out[8]: 9

In[9]: x1[-2]

Out[9]: 7
```

Обращаться к элементам в многомерном массиве можно с помощью разделенных запятыми кортежей индексов:

```
In[10]: x2

Out[10]: array([[3, 5, 2, 4],
                [7, 6, 8, 8],
                [1, 6, 7, 7]])

In[11]: x2[0, 0]

Out[11]: 3
```

```
In[12]: x2[2, 0]

Out[12]: 1

In[13]: x2[2, -1]

Out[13]: 7
```

Вы также можете изменить значения, используя любую из перечисленных выше индексных нотаций.

```
In[14]: x2[0, 0] = 12
        x2


Out[14]: array([[12, 5, 2, 4],
                [ 7, 6, 8, 8],
                [ 1, 6, 7, 7]])
```

Не забывайте, что, в отличие от списков языка Python, у массивов NumPy фиксированный тип данных. При попытке вставить в массив целых чисел значение с плавающей точкой это значение будет незаметно усечено. Не попадитесь в ловушку!

```
In[15]: x1[0] = 3.14159  # это значение будет усечено!
        x1

Out[15]: array([3, 0, 3, 3, 7, 9])
```

Срезы массивов: доступ к подмассивам

Аналогично доступу к отдельным элементам массива можно использовать квадратные скобки для доступа к подмассивам с помощью *срезов* (slicing), обозначаемых знаком двоеточия (`:`). Синтаксис срезов библиотеки NumPy соответствует аналогичному синтаксису для стандартных списков языка Python. Для доступа к срезу массива `x` используйте синтаксис:

```
x[начало:конец:шаг]
```

Если какие-либо из этих значений не указаны, значения применяются по умолчанию: `начало = 0`, `конец =` *`размер соответствующего измерения`*, `шаг = 1`. Мы рассмотрим доступ к массивам в одном и нескольких измерениях.

Одномерные подмассивы

```
In[16]: x = np.arange(10)
        x

Out[16]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In[17]: x[:5]  # первые пять элементов

Out[17]: array([0, 1, 2, 3, 4])

In[18]: x[5:]  # элементы после индекса = 5

Out[18]: array([5, 6, 7, 8, 9])

In[19]: x[4:7]  # подмассив из середины

Out[19]: array([4, 5, 6])

In[20]: x[::2]  # каждый второй элемент

Out[20]: array([0, 2, 4, 6, 8])

In[21]: x[1::2]  # каждый второй элемент, начиная с индекса 1

Out[21]: array([1, 3, 5, 7, 9])
```

Потенциально к небольшой путанице может привести отрицательное значение параметра `шаг`. В этом случае значения по умолчанию для `начало` и `конец` меняются местами. Это удобный способ «перевернуть» массив:

```
In[22]: x[::-1]  # все элементы в обратном порядке

Out[22]: array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])

In[23]: x[5::-2]  # каждый второй элемент в обратном порядке,
                  # начиная с индекса 5

Out[23]: array([5, 3, 1])
```

Многомерные подмассивы

Многомерные срезы задаются схожим образом, с разделением срезов запятыми. Например:

```
In[24]: x2

Out[24]: array([[12,  5,  2,  4],
                [ 7,  6,  8,  8],
                [ 1,  6,  7,  7]])

In[25]: x2[:2, :3]  # две строки, три столбца

Out[25]: array([[12,  5,  2],
                [ 7,  6,  8]])

In[26]: x2[:3, ::2]  # все строки, каждый второй столбец
```

```
Out[26]: array([[12,  2],
                [ 7,  8],
                [ 1,  7]])
```

Измерения подмассивов также можно «переворачивать»:

```
In[27]: x2[::-1, ::-1]

Out[27]: array([[ 7,  7,  6,  1],
                [ 8,  8,  6,  7],
                [ 4,  2,  5, 12]])
```

Доступ к строкам и столбцам массива

Часто возникает необходимость в доступе к отдельным строкам или столбцам массива. Предоставить доступ можно путем комбинации индексации и среза, с помощью пустого среза, задаваемого двоеточием (`:`):

```
In[28]: print(x2[:, 0])  # первый столбец массива x2

[12  7  1]

In[29]: print(x2[0, :])  # первая строка массива x2

[12  5  2  4]
```

В случае предоставления доступа к строке пустой срез можно опустить ради более лаконичного синтаксиса:

```
In[30]: print(x2[0])  # эквивалентно x2[0, :]

[12  5  2  4]
```

Подмассивы как предназначенные только для чтения представления

Срезы массивов возвращают *представления* (views), а не *копии* (copies) данных массива. Этим срезы массивов библиотеки NumPy отличаются от срезов списков языка Python (в списках срезы являются копиями). Рассмотрим уже знакомый нам двумерный массив:

```
In[31]: print(x2)

[[12  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

Извлечем из него двумерный подмассив 2 × 2:

```
In[32]: x2_sub = x2[:2, :2]
        print(x2_sub)
[[12  5]
 [ 7  6]]
```

Теперь, если мы изменим этот подмассив, увидим, что исходный массив также поменялся! Смотрите:

```
In[33]: x2_sub[0, 0] = 99
        print(x2_sub)

[[99  5]
 [ 7  6]]

In[34]: print(x2)

[[99  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

Такое поведение по умолчанию действительно очень удобно: при работе с большими наборами данных не требуется копировать базовый буфер данных для обращения к их частям и обработки этих частей.

Создание копий массивов

Несмотря на все замечательные возможности представлений массивов, иногда бывает удобно явным образом скопировать данные из массива или подмассива. Это легко можно сделать с помощью метода `copy()`:

```
In[35]: x2_sub_copy = x2[:2, :2].copy()
        print(x2_sub_copy)

[[99  5]
 [ 7  6]]
```

Если мы теперь поменяем этот подмассив, исходный массив останется неизменным:

```
In[36]: x2_sub_copy[0, 0] = 42
        print(x2_sub_copy)

[[42  5]
 [ 7  6]]

In[37]: print(x2)

[[99  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

Изменение формы массивов

Еще одна удобная операция — изменение формы массивов методом `reshape()`. Например, если вам требуется поместить числа от 1 до 9 в таблицу 3 × 3, сделать это можно следующим образом:

```
In[38]: grid = np.arange(1, 10).reshape((3, 3))
        print(grid)

[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Обратите внимание, что размер исходного массива должен соответствовать размеру измененного. По возможности метод `reshape` будет использовать предназначенные только для чтения представления, но непрерывные буферы памяти найти удается не всегда.

Другой часто используемый паттерн изменения формы — преобразование одномерного массива в двумерную матрицу-строку или матрицу-столбец. Для этого можно применить метод `reshape`, но лучше воспользоваться ключевым словом `newaxis` при выполнении операции среза:

```
In[39]: x = np.array([1, 2, 3])

        # Преобразование в вектор-строку с помощью reshape
        x.reshape((1, 3))

Out[39]: array([[1, 2, 3]])

In[40]: # Преобразование в вектор-строку посредством newaxis
        x[np.newaxis, :]

Out[40]: array([[1, 2, 3]])

In[41]: # Преобразование в вектор-столбец с помощью reshape
        x.reshape((3, 1))

Out[41]: array([[1],
                [2],
                [3]])

In[42]: # Преобразование в вектор-столбец посредством newaxis
        x[:, np.newaxis]

Out[42]: array([[1],
                [2],
                [3]])
```

В нашей книге мы будем часто встречать подобное преобразование.

Слияние и разбиение массивов

Все предыдущие операции работали с одним массивом. Но можно объединить несколько массивов в один и, наоборот, разбить единый массив на несколько подмассивов. Эти операции мы рассмотрим далее.

Слияние массивов

Слияние, или объединение, двух массивов в библиотеке NumPy выполняется в основном с помощью методов `np.concatenate`, `np.vstack` и `np.hstack`. Метод `np.concatenate` принимает на входе кортеж или список массивов в качестве первого аргумента:

```
In[43]: x = np.array([1, 2, 3])
        y = np.array([3, 2, 1])
        np.concatenate([x, y])

Out[43]: array([1, 2, 3, 3, 2, 1])
```

Можно объединять более двух массивов одновременно:

```
In[44]: z = [99, 99, 99]
        print(np.concatenate([x, y, z]))

[ 1  2  3  3  2  1999999]
```

Для объединения двумерных массивов можно также использовать `np.concatenate`:

```
In[45]: grid = np.array([[1, 2, 3],
                         [4, 5, 6]])

In[46]: # слияние по первой оси координат
        np.concatenate([grid, grid])

Out[46]: array([[1, 2, 3],
                [4, 5, 6],
                [1, 2, 3],
                [4, 5, 6]])

In[47]: # слияние по второй оси координат (с индексом 0)
        np.concatenate([grid, grid], axis=1)

Out[47]: array([[1, 2, 3, 1, 2, 3],
                [4, 5, 6, 4, 5, 6]])
```

Для работы с массивами с различающимися измерениями удобнее и понятнее использовать функции `np.vstack` (вертикальное объединение) и `np.hstack` (горизонтальное объединение):

```
In[48]: x = np.array([1, 2, 3])
        grid = np.array([[9, 8, 7],
                         [6, 5, 4]])

        # Объединяет массивы по вертикали
        np.vstack([x, grid])

Out[48]: array([[1, 2, 3],
                [9, 8, 7],
                [6, 5, 4]])

In[49]: # Объединяет массивы по горизонтали
        y = np.array([[99],
                      [99]])
        np.hstack([grid, y])

Out[49]: array([[ 9,  8,  7, 99],
                [ 6,  5,  4, 99]])
```

Функция `np.dstack` аналогично объединяет массивы по третьей оси.

Разбиение массивов

Противоположностью слияния является разбиение, выполняемое с помощью функций `np.split`, `np.hsplit` и `np.vsplit`. Каждой из них необходимо передавать список индексов, задающих точки раздела:

```
In[50]: x = [1, 2, 3, 99, 99, 3, 2, 1]
        x1, x2, x3 = np.split(x, [3, 5])
        print(x1, x2, x3)

[1 2 3] [99 99] [3 2 1]
```

Обратите внимание, что N точек раздела означают $N + 1$ подмассив. Соответствующие функции `np.hsplit` и `np.vsplit` действуют аналогично:

```
In[51]: grid = np.arange(16).reshape((4, 4))
        grid

Out[51]: array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11],
                [12, 13, 14, 15]])

In[52]: upper, lower = np.vsplit(grid, [2])
        print(upper)
        print(lower)
[[0 1 2 3]
 [4 5 6 7]]
[[ 8  9 10 11]
 [12 13 14 15]]
```

```
In[53]: left, right = np.hsplit(grid, [2])
        print(left)
        print(right)
[[ 0  1]
 [ 4  5]
 [ 8  9]
 [12 13]]
[[ 2  3]
 [ 6  7]
 [10 11]
 [14 15]]
```

Функция `np.dsplit` аналогично разделяет массивы по третьей оси.

Выполнение вычислений над массивами библиотеки NumPy: универсальные функции

Библиотека NumPy предоставляет удобный и гибкий интерфейс для оптимизированных вычислений над массивами данных.

Выполнение вычислений над массивами библиотеки NumPy может быть очень быстрым и очень медленным. Ключ к их ускорению — использование векторизованных операций, обычно реализуемых посредством *универсальных функций* (universal functions, ufuncs) языка Python. В данном разделе будет обосновано, почему универсальные функции библиотеки NumPy необходимы и почему они могут намного ускорить выполнение повторяющихся вычислений над элементами массивов, а также познакомим вас с множеством наиболее распространенных и полезных универсальных арифметических функций из библиотеки NumPy.

Медлительность циклов

Реализация языка Python по умолчанию (известная под названием CPython) выполняет некоторые операции очень медленно. Частично это происходит из-за динамической, интерпретируемой природы языка. Гибкость типов означает, что последовательности операций нельзя скомпилировать в столь же производительный машинный код, как в случае языков C и Fortran. В последнее время было предпринято несколько попыток справиться с этой проблемой:

- проект PyPy (http://pypy.org), реализация языка Python с динамической компиляцией;
- проект Cython (http://cython.org), преобразующий код на языке Python в компилируемый код на языке C;

- проект Numba (http://numba.pydata.org), преобразующий фрагменты кода на языке Python в быстрый LLVM-байткод.

У каждого проекта есть свои сильные и слабые стороны, но ни один из них пока не обошел стандартный механизм CPython по популярности.

Относительная медлительность Python обычно обнаруживается при повторении множества мелких операций, например при выполнении обработки всех элементов массива в цикле. Пусть у нас имеется массив значений и необходимо вычислить обратную величину каждого из них. Очевидное решение могло бы выглядеть следующим образом:

```
In[1]: import numpy as np
       np.random.seed(0)

       def compute_reciprocals(values):
           output = np.empty(len(values))
           for i in range(len(values)):
               output[i] = 1.0 / values[i]
           return output

       values = np.random.randint(1, 10, size=5)
       compute_reciprocals(values)

Out[1]: array([ 0.16666667,  1.        ,  0.25    ,  0.25    ,  0.125    ])
```

Такая реализация, вероятно, кажется вполне естественной разработчикам с опытом работы на языках программирования C или Java. Однако, оценив время выполнения этого кода для большого объема данных, мы обнаружим, что данная операция выполняется крайне медленно. Оценим это время с помощью «магической» функции `%timeit` оболочки IPython (обсуждавшейся в разделе «Профилирование и мониторинг скорости выполнения кода» главы 1):

```
In[2]: big_array = np.random.randint(1, 100, size=1000000)
       %timeit compute_reciprocals(big_array)

1 loop, best of 3: 2.91 s per loop
```

Выполнение миллионов операций и сохранение результата заняло несколько секунд! В наши дни, когда даже у смартфонов быстродействие измеряется в гигафлопсах (то есть миллиардах операций с плавающей точкой в секунду), это представляется медленным практически до абсурда. Оказывается, проблема не в самих операциях, а в проверке типов и диспетчеризации функций, выполняемых CPython при каждом проходе цикла. Всякий раз, когда вычисляется обратная величина, Python сначала проверяет тип объекта и выполняет динамический поиск подходящей для этого типа функции. Если бы мы работали с компилируемым кодом, сведения о типе были бы известны до выполнения кода, а значит, результат вычислялся бы намного эффективнее.

Введение в универсальные функции

Библиотека NumPy предоставляет для многих типов операций удобный интерфейс для компилируемой процедуры со статической типизацией. Он известен под названием *векторизованной* операции. Для этого достаточно просто выполнить операцию с массивом, которая затем будет применена для каждого из его элементов. Векторизованный подход спроектирован так, чтобы переносить цикл в скомпилированный слой, лежащий в основе библиотеки NumPy, что обеспечивает гораздо более высокую производительность.

Сравните результаты следующих двух фрагментов кода:

```
In[3]: print(compute_reciprocals(values))
       print(1.0 / values)
```

```
[ 0.16666667  1.     0.25     0.25     0.125     ]
[ 0.16666667  1.     0.25     0.25     0.125     ]
```

Можем отметить, что векторизованная операция выполняется на несколько порядков быстрее, чем стандартный цикл Python:

```
In[4]: %timeit (1.0 / big_array)
100 loops, best of 3: 4.6 ms per loop
```

Векторизованные операции в библиотеке NumPy реализованы посредством *универсальных функций* (ufuncs), главная задача которых состоит в быстром выполнении повторяющихся операций над значениями из массивов библиотеки NumPy. Универсальные функции исключительно гибки. Выше была показана операция между скалярным значением и массивом, но можно также выполнять операции над двумя массивами:

```
In[5]: np.arange(5) / np.arange(1, 6)
```

```
Out[5]: array([ 0.      ,  0.5     ,  0.66666667,  0.75    ,  0.8     ])
```

Операции с универсальными функциями не ограничиваются одномерными массивами, они также могут работать и с многомерными:

```
In[6]: x = np.arange(9).reshape((3, 3))
       2 ** x
Out[6]: array([[  1,   2,   4],
               [  8,  16,  32],
               [ 64, 128, 256]])
```

Вычисления с применением векторизации посредством универсальных функций практически всегда более эффективны, чем их эквиваленты, реализованные с помощью циклов Python, особенно при росте размера массивов. Столкнувшись с подобным циклом в сценарии на языке Python, следует обдумать, не стоит ли заменить его векторизованным выражением.

Обзор универсальных функций библиотеки NumPy

Существует два вида универсальных функций: *унарные* универсальные функции, с одним аргументом, и *бинарные*, с двумя аргументами. Мы рассмотрим примеры обоих типов функций.

Арифметические функции над массивами

Универсальные функции библиотеки NumPy очень просты в использовании, поскольку применяют нативные арифметические операторы языка Python. Можно выполнять обычные сложение, вычитание, умножение и деление:

```
In[7]: x = np.arange(4)
       print("x     =", x)
       print("x + 5 =", x + 5)
       print("x - 5 =", x - 5)
       print("x * 2 =", x * 2)
       print("x / 2 =", x / 2)
       print("x // 2 =", x // 2)  # деление с округлением в меньшую сторону

x     = [0 1 2 3]
x + 5 = [5 6 7 8]
x - 5 = [-5 -4 -3 -2]
x * 2 = [0 2 4 6]
x / 2 = [ 0.   0.5  1.   1.5]
x // 2 = [0 0 1 1]
```

Существует также унарная универсальная функция для операции изменения знака, оператор ** для возведения в степень и оператор % для деления по модулю:

```
In[8]: print("-x     = ", -x)
       print("x ** 2 = ", x ** 2)
       print("x % 2  = ", x % 2)

-x     =  [ 0 -1 -2 -3]
x ** 2 =  [0 1 4 9]
x % 2  =  [0 1 0 1]
```

Дополнительно эти операции можно комбинировать любыми способами с соблюдением стандартного приоритета выполнения операций:

```
In[9]: -(0.5*x + 1) ** 2

Out[9]: array([-1.  , -2.25, -4.  , -6.25])
```

Все арифметические операции — удобные адаптеры для встроенных функций библиотеки NumPy. Например, оператор + является адаптером для функции add:

```
In[10]: np.add(x, 2)

Out[10]: array([2, 3, 4, 5])
```

В табл. 2.2 перечислены реализованные в библиотеке NumPy арифметические операторы.

Таблица 2.2. Реализованные в библиотеке NumPy арифметические операторы

Оператор	Эквивалентная универсальная функция	Описание
+	np.add	Сложение (например, 1 + 1 = 2)
–	np.subtract	Вычитание (например, 3 – 2 = 1)
–	np.negative	Унарная операция изменения знака (например, –2)
*	np.multiply	Умножение (например, 2 * 3 = 6)
/	np.divide	Деление (например, 3 / 2 = 1.5)
//	np.floor_divide	Деление с округлением в меньшую сторону (например, 3 // 2 = 1)
**	np.power	Возведение в степень (например, 2 ** 3 = 8)
%	np.mod	Модуль/остаток (например, 9 % 4 = 1)

Помимо этого, существуют еще логические/побитовые операции, которые мы рассмотрим в разделе «Сравнения, маски и булева логика» данной главы.

Абсолютное значение

Аналогично тому, что библиотека NumPy понимает встроенные арифметические операторы, она также понимает встроенную функцию абсолютного значения языка Python:

```
In[11]: x = np.array([-2, -1, 0, 1, 2])
        abs(x)

Out[11]: array([2, 1, 0, 1, 2])
```

Соответствующая универсальная функция библиотеки NumPy — `np.absolute`, доступная также под псевдонимом `np.abs`:

```
In[12]: np.absolute(x)

Out[12]: array([2, 1, 0, 1, 2])

In[13]: np.abs(x)

Out[13]: array([2, 1, 0, 1, 2])
```

Эта универсальная функция может также обрабатывать комплексные значения, возвращая их модуль:

```
In[14]: x = np.array([3 - 4j, 4 - 3j, 2 + 0j, 0 + 1j])
```

```
        np.abs(x)

Out[14]: array([ 5.,  5.,  2.,  1.])
```

Тригонометрические функции

Библиотека NumPy предоставляет множество универсальных функций, одни из наиболее важных — тригонометрические функции. Начнем с описания массива углов:

```
In[15]: theta = np.linspace(0, np.pi, 3)
```

Теперь мы можем вычислить некоторые тригонометрические функции от этих значений:

```
In[16]: print("theta      = ", theta)
        print("sin(theta) = ", np.sin(theta))
        print("cos(theta) = ", np.cos(theta))
        print("tan(theta) = ", np.tan(theta))

theta      =  [ 0.          1.57079633  3.14159265]
sin(theta) =  [  0.00000000e+00   1.00000000e+00   1.22464680e-16]
cos(theta) =  [  1.00000000e+00   6.12323400e-17  -1.00000000e+00]
tan(theta) =  [  0.00000000e+00   1.63312394e+16  -1.22464680e-16]
```

Значения вычислены в пределах точности конкретной вычислительной машины, поэтому те из них, которые должны быть нулевыми, не всегда в точности равны нулю. Доступны для использования также обратные тригонометрические функции:

```
In[17]: x = [-1, 0, 1]
        print("x         = ", x)
        print("arcsin(x) = ", np.arcsin(x))
        print("arccos(x) = ", np.arccos(x))
        print("arctan(x) = ", np.arctan(x))

x         =  [-1, 0, 1]
arcsin(x) =  [-1.57079633  0.          1.57079633]
arccos(x) =  [ 3.14159265  1.57079633  0.        ]
arctan(x) =  [-0.78539816  0.          0.78539816]
```

Показательные функции и логарифмы

Показательные функции — еще один распространенный тип операций, доступный в библиотеке NumPy:

```
In[18]: x = [1, 2, 3]
        print("x     =", x)
        print("e^x   =", np.exp(x))
        print("2^x   =", np.exp2(x))
        print("3^x   =", np.power(3, x))
```

```
x     = [1, 2, 3]
e^x   = [  2.71828183   7.3890561   20.08553692]
2^x   = [ 2.  4.  8.]
3^x   = [ 3  9 27]
```

Функции, обратные к показательным, и логарифмы также имеются в библиотеке. Простейшая функция `np.log` возвращает натуральный логарифм числа. Если вам требуется логарифм по основанию 2 или 10, они также доступны:

```
In[19]: x = [1, 2, 4, 10]
        print("x        =", x)
        print("ln(x)    =", np.log(x))
        print("log2(x)  =", np.log2(x))
        print("log10(x) =", np.log10(x))

x        = [1, 2, 4, 10]
ln(x)    = [ 0.          0.69314718  1.38629436  2.30258509]
log2(x)  = [ 0.          1.          2.          3.32192809]
log10(x) = [ 0.          0.30103     0.60205999  1.        ]
```

Имеются некоторые специальные версии функций, удобные для сохранения точности при очень малых вводимых значениях:

```
In[20]: x = [0, 0.001, 0.01, 0.1]
        print("exp(x) - 1 =", np.expm1(x))
        print("log(1 + x) =", np.log1p(x))

exp(x) - 1 = [ 0.          0.0010005   0.01005017  0.10517092]
log(1 + x) = [ 0.          0.0009995   0.00995033  0.09531018]
```

При очень малых значениях элементов вектора `x` данные функции возвращают намного более точные результаты, чем обычные функции `np.log` и `np.exp`.

Специализированные универсальные функции

В библиотеке NumPy имеется немало других универсальных функций, включая гиперболические тригонометрические функции, поразрядную арифметику, операторы сравнения, преобразования из радианов в градусы, округление и остатки от деления, а также многое другое. Если вы заглянете в документацию по библиотеке NumPy, то откроете для себя немало интересной функциональности.

Если один замечательный источник специализированных и сложных универсальных функций — подмодуль `scipy.special`. Если вам необходимо вычислить значение какой-то хитрой математической функции на ваших данных, очень возможно, что эта функциональность уже реализована в `scipy.special`. Следующий фрагмент кода демонстрирует несколько функций, которые могут пригодиться для статистических вычислений:

```
In[21]: from scipy import special

In[22]: # Гамма-функции (обобщенные факториалы) и тому подобные функции
        x = [1, 5, 10]
        print("gamma(x)     =", special.gamma(x))
        print("ln|gamma(x)| =", special.gammaln(x))
        print("beta(x, 2)   =", special.beta(x, 2))

gamma(x)     = [  1.00000000e+00   2.40000000e+01   3.62880000e+05]
ln|gamma(x)| = [  0.           3.17805383  12.80182748]
beta(x, 2)   = [ 0.5         0.03333333  0.00909091]

In[23]: # Функция ошибок (интеграл от Гауссовой функции),
        # дополнительная и обратная к ней функции
        x = np.array([0, 0.3, 0.7, 1.0])
        print("erf(x)  =", special.erf(x))
        print("erfc(x) =", special.erfc(x))
        print("erfinv(x) =", special.erfinv(x))

erf(x)  = [ 0.          0.32862676  0.67780119  0.84270079]
erfc(x) = [ 1.          0.67137324  0.32219881  0.15729921]
erfinv(x) = [ 0.          0.27246271  0.73286908         inf]
```

В библиотеках NumPy и `scipy.special` имеется много универсальных функций. Документация по ним доступна в Интернете.

Продвинутые возможности универсальных функций

Многие пользователи пакета NumPy работают с универсальными функциями, даже не подозревая обо всех их возможностях. Мы вкратце рассмотрим некоторые специализированные возможности универсальных функций.

Указание массива для вывода результата

При больших вычислениях удобно задать массив, в котором будет сохранен результат вычисления. Вместо того чтобы создавать временный массив, можно воспользоваться этой возможностью для записи результатов вычислений непосредственно в нужное вам место памяти. Сделать это для любой универсальной функции можно с помощью аргумента `out`:

```
In[24]: x = np.arange(5)
        y = np.empty(5)
        np.multiply(x, 10, out=y)
        print(y)

[  0.  10.  20.  30.  40.]
```

Эту возможность можно использовать даже вместе с представлениями массивов. Например, можно записать результаты вычислений в каждый второй элемент заданного массива:

```
In[25]: y = np.zeros(10)
        np.power(2, x, out=y[::2])
        print(y)

[  1.   0.   2.   0.   4.   0.   8.   0.  16.   0.]
```

Если бы мы вместо этого написали `y[::2] = 2 ** x`, был бы создан временный массив для хранения результатов операции `2 ** x` с последующим копированием этих значений в массив `y`. Для столь незначительных объемов вычислений особой разницы нет, но для очень больших массивов экономия памяти за счет аккуратного использования аргумента `out` может оказаться значительной.

Сводные показатели

У бинарных универсальных функций есть возможность вычислять непосредственно на основе объекта некоторые сводные данные. Например, если нам нужно редуцировать массив с помощью данной конкретной операции, можно воспользоваться методом `reduce` соответствующей универсальной функции. Операция `reduce` многократно применяет заданную операцию к элементам массива до тех пор, пока не останется только один результат.

Например, вызов метода `reduce` для универсальной функции `add` возвращает сумму всех элементов массива:

```
In[26]: x = np.arange(1, 6)
        np.add.reduce(x)

Out[26]: 15
```

Аналогично вызов метода `reduce` для универсальной функции `multiply` возвращает произведение всех элементов массива:

```
In[27]: np.multiply.reduce(x)

Out[27]: 120
```

Если же мы хотим сохранить все промежуточные результаты вычислений, можно вместо `reduce` воспользоваться функцией `accumulate`:

```
In[28]: np.add.accumulate(x)

Out[28]: array([ 1,  3,  6, 10, 15])

In[29]: np.multiply.accumulate(x)

Out[29]: array([  1,   2,   6,  24, 120])
```

Обратите внимание, что в данных конкретных случаях для вычисления этих значений существуют и специализированные функции библиотеки NumPy (`np.sum`,

`np.prod`, `np.cumsum`, `np.cumprod`), которые мы рассмотрим в разделе «Агрегирование: минимум, максимум и все, что посередине» данной главы.

Векторные произведения

Все универсальные функции могут выводить результат применения соответствующей операции ко всем парам двух аргументов с помощью метода `outer`. Это дает возможность одной строкой кода создавать, например, таблицу умножения:

```
In[30]: x = np.arange(1, 6)
        np.multiply.outer(x, x)

Out[30]: array([[ 1,  2,  3,  4,  5],
                [ 2,  4,  6,  8, 10],
                [ 3,  6,  9, 12, 15],
                [ 4,  8, 12, 16, 20],
                [ 5, 10, 15, 20, 25]])
```

Очень удобны методы `ufunc.at` и `ufunc.reduceat`, которые мы рассмотрим в разделе «"Прихотливая" индексация» данной главы.

Универсальные функции дают возможность работать с массивами различных размеров и форм, используя набор операций под названием *транслирование* (broadcasting). Эта тема достаточно важна, так что ей будет посвящен целый раздел (см. «Операции над массивами. Транслирование» данной главы).

Универсальные функции: дальнейшая информация

На сайтах документации библиотек NumPy и SciPy можно найти дополнительную информацию об универсальных функциях (включая полный список имеющихся функций).

Получить доступ к этой информации можно непосредственно из оболочки IPython путем импорта этих пакетов и использования автодополнения табуляцией (клавиша `Tab`) и справочной функциональности (`?`), как описано в разделе «Справка и документация в оболочке Python» главы 1.

Агрегирование: минимум, максимум и все, что посередине

Очень часто при работе с большими объемами данных первый шаг заключается в вычислении сводных статистических показателей по этим данным. Среднее значение и стандартное отклонение, позволяющие выявить «типичные» значения в наборе данных, — наиболее распространенные сводные статистические показате-

ли, но и другие сводные показатели также полезны (сумма, произведение, медиана, минимум и максимум, квантили и т. д.).

В библиотеке NumPy имеются быстрые функции агрегирования для работы с массивами. Продемонстрирую некоторые из них.

Суммирование значений из массива

В качестве примера рассмотрим вычисление суммы значений массива. В «чистом» языке Python это можно сделать с помощью встроенной функции `sum`:

```
In[1]: import numpy as np

In[2]: L = np.random.random(100)
       sum(L)

Out[2]: 55.61209116604941
```

Синтаксис очень похож на функцию `sum` библиотеки NumPy, и результат в простейшем случае тот же:

```
In[3]: np.sum(L)

Out[3]: 55.612091166049424
```

Однако, поскольку функция `sum` выполняет операцию в скомпилированном коде, версия библиотеки NumPy данной операции работает намного быстрее:

```
In[4]: big_array = np.random.rand(1000000)
       %timeit sum(big_array)
       %timeit np.sum(big_array)

10 loops, best of 3: 104 ms per loop
1000 loops, best of 3: 442 µs per loop
```

Будьте осторожны: функции `sum` и `np.sum` не идентичны. Например, смысл их необязательных аргументов различен и функция `np.sum` умеет работать с многомерными массивами.

Минимум и максимум

В «чистом» языке Python имеются встроенные функции `min` и `max`, используемые для вычисления минимального и максимального значений любого заданного массива:

```
In[5]: min(big_array), max(big_array)

Out[5]: (1.1717128136634614e-06, 0.9999976784968716)
```

Синтаксис соответствующих функций из библиотеки NumPy аналогичен, причем они также работают намного быстрее:

```
In[6]: np.min(big_array), np.max(big_array)

Out[6]: (1.1717128136634614e-06, 0.9999976784968716)

In[7]: %timeit min(big_array)
       %timeit np.min(big_array)

10 loops, best of 3: 82.3 ms per loop
1000 loops, best of 3: 497 µs per loop
```

Для `min`, `max`, `sum` и еще нескольких функций вычисления сводных показателей библиотеки NumPy существует сокращенная запись операции путем применения методов самого объекта массива:

```
In[8]: print(big_array.min(), big_array.max(), big_array.sum())

1.17171281366e-060.999997678497499911.628197
```

При работе с массивами библиотеки NumPy обязательно проверяйте, используете ли вы NumPy-версию функций для вычисления сводных показателей!

Многомерные сводные показатели

Агрегирование по столбцу или строке — один из часто применяемых видов операций агрегирования. Пусть имеются какие-либо данные, находящиеся в двумерном массиве:

```
In[9]: M = np.random.random((3, 4))
       print(M)

[[ 0.8967576   0.03783739  0.75952519  0.06682827]
 [ 0.8354065   0.99196818  0.19544769  0.43447084]
 [ 0.66859307  0.15038721  0.37911423  0.6687194 ]]
```

По умолчанию все функции агрегирования библиотеки NumPy возвращают сводный показатель по всему массиву:

```
In[10]: M.sum()

Out[10]: 6.0850555667307118
```

Но функции агрегирования принимают на входе дополнительный аргумент, позволяющий указать *ось*, по которой вычисляется сводный показатель. Например, можно найти минимальное значение каждого из столбцов, указав `axis=0`:

```
In[11]: M.min(axis=0)

Out[11]: array([ 0.66859307,  0.03783739,  0.19544769,  0.06682827])
```

Функция возвращает четыре значения, соответствующие четырем столбцам чисел.

Аналогично можно вычислить максимальное значение в каждой из строк:

```
In[12]: M.max(axis=1)

Out[12]: array([ 0.8967576 ,  0.99196818,  0.6687194 ])
```

Способ задания оси в этих примерах может вызвать затруднения у пользователей, работавших ранее с другими языками программирования. Ключевое слово `axis` задает *измерение массива, которое будет «схлопнуто»*, а не возвращаемое измерение. Так что указание `axis=0` означает, что первая ось будет «схлопнута»: для двумерных массивов значения в каждом из столбцов будут агрегированы.

Другие функции агрегирования

Библиотека NumPy предоставляет много других агрегирующих функций. У большинства есть `NaN`-безопасный эквивалент, вычисляющий результат с игнорированием отсутствующих значений, помеченных специально определенным организацией IEEE значением с плавающей точкой `NaN` (см. в разделе «Обработка отсутствующих данных» главы 3 более подробное обсуждение этого вопроса). Некоторые из `NaN`-безопасных функций были добавлены только в версии 1.8 библиотеки NumPy, поэтому они недоступны в ранних версиях.

В табл. 2.3 приведен список полезных агрегирующих функций, доступных в библиотеке NumPy.

Таблица 2.3. Доступные в библиотеке NumPy функции агрегирования

Имя функции	NaN-безопасная версия	Описание
np.sum	np.nansum	Вычисляет сумму элементов
np.prod	np.nanprod	Вычисляет произведение элементов
np.mean	np.nanmean	Вычисляет среднее значение элементов
np.std	np.nanstd	Вычисляет стандартное отклонение
np.var	np.nanvar	Вычисляет дисперсию
np.min	np.nanmin	Вычисляет минимальное значение
np.max	np.nanmax	Вычисляет максимальное значение
np.argmin	np.nanargmin	Возвращает индекс минимального значения
np.argmax	np.nanargmax	Возвращает индекс максимального значения
np.median	np.nanmedian	Вычисляет медиану элементов
np.percentile	np.nanpercentile	Вычисляет квантили элементов
np.any	N/A	Проверяет, существуют ли элементы со значением true
np.all	N/A	Проверяет, все ли элементы имеют значение true

Мы часто будем встречаться с этими агрегирующими функциями в дальнейшем.

Пример: чему равен средний рост президентов США

Имеющиеся в библиотеке NumPy функции агрегирования могут очень пригодиться для обобщения набора значений. В качестве простого примера рассмотрим рост всех президентов США. Эти данные доступны в файле `president_heights.csv`, представляющем собой простой разделенный запятыми список меток и значений:

```
In[13]: !head -4 data/president_heights.csv

order,name,height(cm)
1,George Washington,189
2,John Adams,170
3,Thomas Jefferson,189
```

Мы воспользуемся пакетом Pandas, который изучим более детально в главе 3, для чтения файла и извлечения данной информации (обратите внимание, что рост указан в сантиметрах):

```
In[14]: import pandas as pd
        data = pd.read_csv('data/president_heights.csv')
        heights = np.array(data['height(cm)'])
        print(heights)

[189 170 189 163 183 171 185 168 173 183 173 173 175 178 183 193 178 173
 174 183 183 168 170 178 182 180 183 178 182 188 175 179 183 193 182 183
 177 185 188 188 182 185]
```

Теперь, получив такой массив данных, мы можем вычислить множество сводных статистических показателей:

```
In[15]: print("Mean height:       ",    heights.mean())
        print("Standard deviation:", heights.std())
        print("Minimum height:    ",   heights.min())
        print("Maximum height:    ",  heights.max())

Mean height:           179.738095238
Standard deviation:    6.93184344275
Minimum height:        163
Maximum height:        193
```

Обратите внимание, что в каждом случае операция агрегирования редуцирует весь массив к одному итоговому значению, дающему нам информацию о распределении значений. Возможно, нам захочется также вычислить квантили:

```
In[16]: print("25th percentile:   ", np.percentile(heights, 25))
        print("Median:            ", np.median(heights))
        print("75th percentile:   ", np.percentile(heights, 75))

25th percentile:    174.25
Median:             182.0
75th percentile:    183.0
```

Видим, что медиана роста президентов США составляет 182 систем, то есть чуть-чуть не дотягивает до шести футов.

Иногда полезнее видеть графическое представление подобных данных, которое можно получить с помощью утилит из Matplotlib (подробнее мы рассмотрим Matplotlib в главе 4). Например, следующий код генерирует график, показанный на рис. 2.3:

```
In[17]: %matplotlib inline
        import matplotlib.pyplot as plt
        import seaborn; seaborn.set()  # задает стиль графика

In[18]: plt.hist(heights)
        plt.title('Height Distribution of US Presidents') # Распределение роста
                                                          # президентов США
        plt.xlabel('height (cm)')                         # Рост, см
        plt.ylabel('number');                             # Количество
```

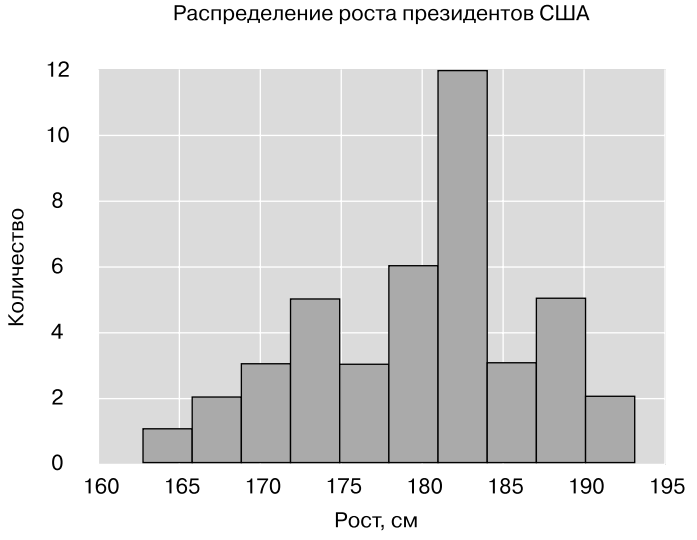

Рис. 2.3. Гистограмма роста президентов

Эти сводные показатели — базовые элементы так называемого разведочного анализа данных (exploratory data analysis), который мы рассмотрим подробнее в следующих главах книги.

Операции над массивами. Транслирование

В предыдущем разделе мы рассмотрели, каким образом можно использовать универсальные функции библиотеки NumPy для *векторизации* операций, а следовательно, устранения медленных стандартных циклов языка Python. Еще один способ применения операций векторизации — использовать имеющиеся в библиотеке

NumPy возможности транслирования (broadcasting). Транслирование представляет собой набор правил по применению бинарных универсальных функций (сложение, вычитание, умножение и т. д.) к массивам различного размера.

Введение в транслирование

Для массивов одного размера бинарные операции выполняются поэлементно:

```
In[1]: import numpy as np

In[2]: a = np.array([0, 1, 2])
       b = np.array([5, 5, 5])
       a + b

Out[2]: array([5, 6, 7])
```

Транслирование дает возможность выполнять подобные виды бинарных операций над массивами различных размеров, например, можно легко прибавить скалярное значение (рассматривая его как нульмерный массив) к массиву:

```
In[3]: a + 5

Out[3]: array([5, 6, 7])
```

Можно рассматривать транслирование как операцию, превращающую путем растягивания (или дублирования) значение 5 в массив [5, 5, 5], после чего складывающую полученный результат с массивом `a`. Преимущество NumPy-транслирования заключается в том, что дублирование значений на самом деле не выполняется, это лишь удобная нам умозрительная модель.

Аналогично можно распространить транслирование на массивы большей размерности. Посмотрите на результат сложения одномерного и двумерного массивов:

```
In[4]: M = np.ones((3, 3))
       M

Out[4]: array([[ 1.,  1.,  1.],
               [ 1.,  1.,  1.],
               [ 1.,  1.,  1.]])

In[5]: M + a

Out[5]: array([[ 1.,  2.,  3.],
               [ 1.,  2.,  3.],
               [ 1.,  2.,  3.]])
```

Здесь одномерный массив `a` растягивается (транслируется) на второе измерение, чтобы соответствовать форме массива `M`.

Эти примеры просты и понятны. Более сложные случаи могут включать транслирование обоих массивов. Рассмотрим следующий пример:

```
In[6]: a = np.arange(3)
       b = np.arange(3)[:, np.newaxis]

       print(a)
       print(b)

[0 1 2]
[[0]
 [1]
 [2]]

In[7]: a + b

Out[7]: array([[0, 1, 2],
               [1, 2, 3],
               [2, 3, 4]])
```

Аналогично тому, как мы раньше растягивали (транслировали) один массив, чтобы он соответствовал форме другого, здесь мы растягиваем *оба* массива `a` и `b`, чтобы привести их к общей форме. В результате мы получаем двумерный массив! Геометрия этих примеров наглядно показана на рис. 2.4[1].

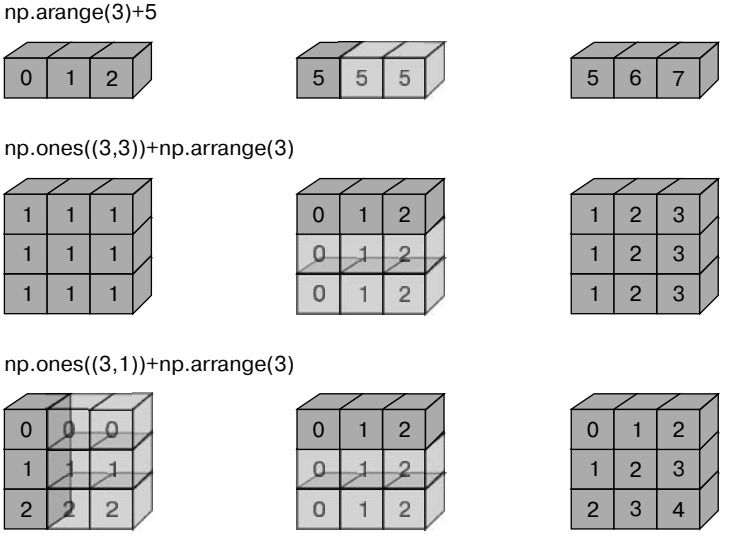

Рис. 2.4. Визуализация транслирования массивов библиотекой NumPy

[1] Код, который был применен для создания этой диаграммы, можно найти в онлайн-приложении. Он был адаптирован из источника, опубликованного в документации к модулю astroML. Используется с разрешения. — *Примеч. авт.*

Полупрозрачные кубики представляют собой транслируемые значения: соответствующая дополнительная память фактически не выделяется в ходе операции, но мысленно удобно представлять себе, что так и происходит.

Правила транслирования

Транслирование в библиотеке NumPy следует строгому набору правил, определяющему взаимодействие двух массивов.

- *Правило 1:* если размерность двух массивов отличается, форма массива с меньшей размерностью *дополняется* единицами с ведущей (левой) стороны.
- *Правило 2*: если форма двух массивов не совпадает в каком-то измерении, массив с формой, равной 1 в данном измерении, растягивается вплоть до соответствия форме другого массива.
- *Правило 3:* если в каком-либо измерении размеры массивов различаются и ни один не равен 1, генерируется ошибка.

Для разъяснения этих правил рассмотрим несколько примеров.

Транслирование. Пример 1

Рассмотрим сложение двумерного массива с одномерным:

```
In[8]: M = np.ones((2, 3))
       a = np.arange(3)
```

Рассмотрим эту операцию подробнее. Формы массивов следующие:

```
M.shape = (2, 3)
a.shape = (3,)
```

По правилу 1, поскольку размерность массива `a` меньше, мы дополняем его измерениями слева:

```
M.shape -> (2, 3)
a.shape -> (1, 3)
```

По правилу 2 мы видим, что первое измерение массивов различается, так что мы растягиваем его вплоть до совпадения:

```
M.shape -> (2, 3)
a.shape -> (2, 3)
```

Формы совпадают, и мы видим, что итоговая форма будет (2, 3):

```
In[9]: M + a

Out[9]: array([[ 1.,  2.,  3.],
               [ 1.,  2.,  3.]])
```

Транслирование. Пример 2

Рассмотрим пример, в котором необходимо транслировать оба массива:

```
In[10]: a = np.arange(3).reshape((3, 1))
        b = np.arange(3)
```

Начнем с записи формы наших массивов:

```
a.shape = (3, 1)
b.shape = (3,)
```

Правило 1 гласит, что мы должны дополнить форму массива `b` единицами:

```
a.shape -> (3, 1)
b.shape -> (1, 3)
```

Правило 2 говорит, что нужно увеличивать эти единицы вплоть до совпадения с размером другого массива:

```
a.shape -> (3, 3)
b.shape -> (3, 3)
```

Поскольку результаты совпадают, формы совместимы. Получаем:

```
In[11]: a + b

Out[11]: array([[0, 1, 2],
                [1, 2, 3],
                [2, 3, 4]])
```

Транслирование. Пример 3

Рассмотрим пример, в котором два массива несовместимы:

```
In[12]: M = np.ones((3, 2))
        a = np.arange(3)
```

Эта ситуация лишь немного отличается от примера 1: матрица `M` транспонирована. Какое же влияние это окажет на вычисления? Формы массивов следующие:

```
M.shape = (3, 2)
a.shape = (3,)
```

Правило 1 требует от нас дополнить форму массива `a` единицами:

```
M.shape -> (3, 2)
a.shape -> (1, 3)
```

Согласно правилу 2 первое измерение массива `a` растягивается, чтобы соответствовать таковому массива `M`:

```
M.shape -> (3, 2)
a.shape -> (3, 3)
```

Теперь вступает в действие правило 3 — итоговые формы не совпадают, так что массивы несовместимы, что мы и видим, попытавшись выполнить данную операцию:

```
In[13]: M + a

---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-13-9e16e9f98da6> in <module>()
----> 1 M + a

ValueError: operands could not be broadcast together with shapes (3,2) (3,)
```

Обратите внимание на имеющийся потенциальный источник ошибки: можно было бы сделать массивы `a` и `M` совместимыми, скажем путем дополнения формы `a` единицами справа, а не слева. Но правила транслирования работают не так! Если вам хочется применить правостороннее дополнение, можете сделать это явным образом, поменяв форму массива (мы воспользуемся ключевым словом `np.newaxis`, описанным в разделе «Введение в массивы библиотеки NumPy» данной главы):

```
In[14]: a[:, np.newaxis].shape

Out[14]: (3, 1)

In[15]: M + a[:, np.newaxis]

Out[15]: array([[ 1.,  1.],
                [ 2.,  2.],
                [ 3.,  3.]])
```

Хотя мы сосредоточили внимание на операторе `+`, данные правила транслирования применимы ко *всем* бинарным универсальным функциям. Например, рассмотрим функцию `logaddexp(a, b)`, вычисляющую $\log(\exp(a) + \exp(b))$ с большей точностью, чем при стандартном подходе:

```
In[16]: np.logaddexp(M, a[:, np.newaxis])

Out[16]: array([[ 1.31326169,  1.31326169],
                [ 1.69314718,  1.69314718],
                [ 2.31326169,  2.31326169]])
```

Для получения дальнейшей информации по множеству доступных универсальных функций см. раздел «Выполнение вычислений над массивами библиотеки NumPy: универсальные функции» данной главы.

Транслирование на практике

Операции транслирования — основное ядро множества примеров, приводимых в дальнейшем в нашей книге. Рассмотрим несколько простых примеров сфер их возможного применения.

Центрирование массива

Благодаря универсальным функциям пользователи библиотеки NumPy избавляются от необходимости писать явным образом медленно работающие циклы языка Python. Транслирование расширяет эти возможности. Один из часто встречающихся примеров — центрирование массива. Пускай у вас есть массив из десяти наблюдений, каждое состоит из трех значений. Используя стандартные соглашения (см. «Представление данных в Scikit-Learn» главы 5), сохраним эти данные в массиве 10×3:

```
In[17]: X = np.random.random((10, 3))
```

Вычислить среднее значение каждого признака можно, применив функцию агрегирования `mean` по первому измерению:

```
In[18]: Xmean = X.mean(0)
        Xmean

Out[18]: array([ 0.53514715,  0.66567217,  0.44385899])
```

Теперь можно центрировать массив `X` путем вычитания среднего значения (это операция транслирования):

```
In[19]: X_centered = X - Xmean
```

Чтобы убедиться в правильности выполнения данной операции, проверьте, чтобы среднее значение центрированного массива было близко к 0:

```
In[20]: X_centered.mean(0)

Out[20]: array([  2.22044605e-17,  -7.77156117e-17,  -1.66533454e-17])
```

В пределах машинной точности среднее значение массива теперь равно 0.

Построение графика двумерной функции

Одна из частых сфер применения транслирования — визуализация изображений на основе двумерных функций. Если нам требуется описать функцию $z = f(x, y)$, можно воспользоваться транслированием для вычисления значений этой функции на координатной сетке:

```
In[21]: # Задаем для x и y 50 шагов от 0 до 5
        x = np.linspace(0, 5, 50)
        y = np.linspace(0, 5, 50)[:, np.newaxis]

        z = np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
```

Воспользуемся библиотекой Matplotlib для построения графика двумерного массива (мы обсудим эти инструменты подробно в разделе «Графики плотности и контурные графики» главы 4):

```
In[22]: %matplotlib inline[1]
        import matplotlib.pyplot as plt

In[23]: plt.imshow(z, origin='lower', extent=[0, 5, 0, 5], cmap='viridis')
         plt.colorbar();[2]
```

Результат, показанный на рис. 2.5, представляет собой великолепную визуализацию двумерной функции.

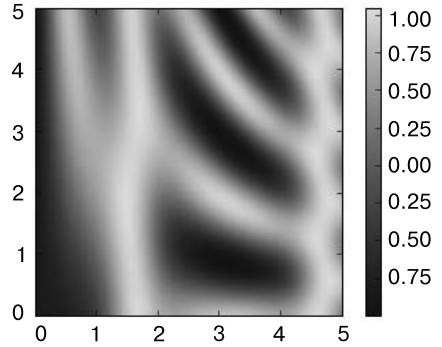

Рис. 2.5. Визуализация двумерного массива

Сравнения, маски и булева логика

В этом разделе показано использование булевых масок для просмотра и изменения значений в NumPy-массивах. Маскирование удобно для извлечения, модификации, подсчета или других манипуляций со значениями в массиве по какому-либо критерию. Например, вам может понадобиться подсчитать все значения, превышающие определенное число, или, возможно, удалить все аномальные значения, превышающие какую-либо пороговую величину. В библиотеке NumPy булевы маски зачастую самый эффективный способ решения подобных задач.

Пример: подсчет количества дождливых дней

Пускай у вас есть последовательности данных, отражающие количество осадков в каждый день года для конкретного города. Например, с помощью библиотеки

---

[1] Данная команда работает не в командной строке IPython, а только в блокноте.

[2] Обратите внимание, что эти две команды необходимо выполнять вместе, а не по отдельности, чтобы не получить ошибку.

Pandas (рассматриваемой подробнее в главе 3) мы загрузили ежедневную статистику по осадкам для Сиэтла за 2014 год:

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

       # Используем Pandas для извлечения количества осадков в дюймах
       # в виде NumPy-массива
       rainfall = pd.read_csv('data/Seattle2014.csv')['PRCP'].values
       inches = rainfall / 254  # 1/10mm -> inches
       inches.shape
```

```
Out[1]: (365,)
```

Этот массив содержит 365 значений, соответствующих ежедневным осадкам в дюймах, с 1 января по 31 декабря 2014 года.

В качестве первой простейшей визуализации рассмотрим гистограмму дождливых дней, показанную на рис. 2.6, сгенерированную с помощью библиотеки Matplotlib (подробнее этот инструмент мы изучим в главе 4):

```
In[2]: %matplotlib inline
       import matplotlib.pyplot as plt
       import seaborn; seaborn.set()  # задаем стили
```

```
In[3]: plt.hist(inches, 40);
```

Эта гистограмма дает общее представление о том, что такое наши данные: несмотря на репутацию, измеренное количество осадков в Сиэтле в абсолютное большинство дней в 2014 году близко к нулю. Но она плохо отражает нужную информацию: например, сколько было дождливых дней? Каково было среднее количество осадков в эти дождливые дни? Сколько было дней с более чем половиной дюйма осадков?

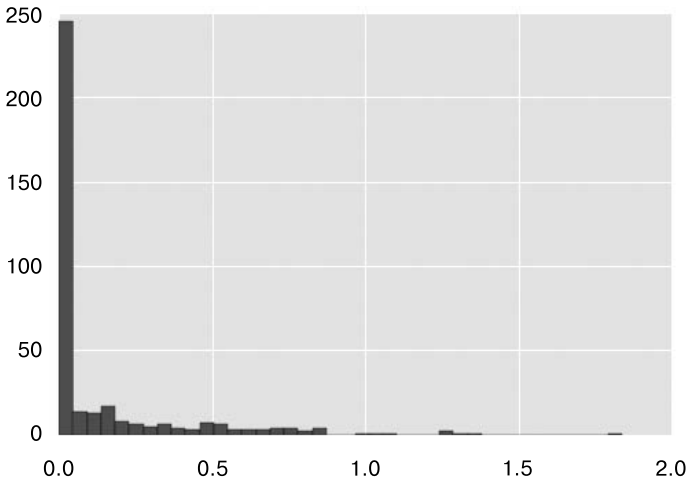

Рис. 2.6. Гистограмма осадков в 2014 году в Сиэтле

Углубляемся в изучение данных. Один из возможных подходов состоит в ответе на эти вопросы «вручную»: организовать цикл по данным, увеличивая счетчик на единицу всякий раз при попадании на находящиеся в нужном диапазоне значения. По уже обсуждавшимся в этой главе причинам такой подход окажется очень неэффективным как с точки зрения времени, затраченного на написание кода, так и времени на вычисление результата. Как мы видели в разделе «Выполнение вычислений над массивами библиотеки NumPy: универсальные функции» данной главы, можно воспользоваться универсальными функциями библиотеки NumPy вместо циклов для выполнения быстрых поэлементных арифметических операций над массивами. В то же время можно использовать другие универсальные функции для поэлементных *сравнений* в массивах, после чего применять результат для ответа на интересующие нас вопросы. Мы ненадолго отложим наши данные в сторону и обсудим некоторые общие инструменты библиотеки NumPy, позволяющие применять *маскирование* (masking) для быстрого ответа на подобные вопросы.

Операторы сравнения как универсальные функции

В разделе «Выполнение вычислений над массивами библиотеки NumPy: универсальные функции» данной главы вы познакомились с универсальными функциями, сосредоточив внимание на арифметических операторах. Вы увидели, что использование операторов +, -, *, / и других для операций над массивами приводит к поэлементным операциям. В библиотеке NumPy также реализованы операторы сравнения, такие как < («меньше») и > («больше») в виде поэлементных универсальных функций. Результат этих операторов сравнения всегда представляет собой массив с булевым типом данных. Доступны для использования все шесть стандартных операторов сравнения:

```
In[4]: x = np.array([1, 2, 3, 4, 5])

In[5]: x < 3  # меньше

Out[5]: array([ True,  True, False, False, False], dtype=bool)

In[6]: x > 3  # больше

Out[6]: array([False, False, False,  True,  True], dtype=bool)

In[7]: x <= 3  # меньше или равно

Out[7]: array([ True,  True,  True, False, False], dtype=bool)

In[8]: x >= 3  # больше или равно

Out[8]: array([False, False,  True,  True,  True], dtype=bool)

In[9]: x != 3  # не равно
```

```
Out[9]: array([ True,  True, False,  True,  True], dtype=bool)

In[10]: x == 3  # равно

Out[10]: array([False, False,  True, False, False], dtype=bool)
```

Можно также выполнять поэлементное сравнение двух массивов и использовать составные выражения:

```
In[11]: (2 * x) == (x ** 2)

Out[11]: array([False,  True, False, False, False], dtype=bool)
```

Как и в случае арифметических операторов, операторы сравнения реализованы в библиотеке NumPy как универсальные функции (табл. 2.4). Например, когда вы пишете `x < 3`, библиотека NumPy на самом деле использует внутри функцию `np.less(x, 3)`.

Таблица 2.4. Краткий список операторов сравнения и эквивалентных им универсальных функций

Оператор	Эквивалентная универсальная функция
==	np.equal
!=	np.not_equal
<	np.less
<=	np.less_equal
>	np.greater
>=	np.greater_equal

Как и в случае с арифметическими универсальными функциями, они могут работать с массивами любого размера и формы. Вот пример для двумерного массива:

```
In[12]: rng = np.random.RandomState(0)
        x = rng.randint(10, size=(3, 4))
        x

Out[12]: array([[5, 0, 3, 3],
                [7, 9, 3, 5],
                [2, 4, 7, 6]])

In[13]: x < 6

Out[13]: array([[ True,  True,  True,  True],
                [False, False,  True,  True],
                [ True,  True, False, False]], dtype=bool)
```

Во всех случаях результат представляет собой булев массив, и в библиотеке NumPy имеется набор простых паттернов для работы с этими булевыми результатами.

Работа с булевыми массивами

В случае булева массива существует множество доступных и удобных операций. Мы будем работать с `x` — созданным нами ранее двумерным массивом.

Подсчет количества элементов

Для подсчета количества элементов `True` в булевом массиве удобно использовать функцию `np.count_nonzero`:

```
In[15]: # Сколько значений массива меньше 6?
        np.count_nonzero(x < 6)

Out[15]: 8
```

Можно увидеть, что в массиве есть восемь элементов, чье значение меньше 6. Другой способ получить эту информацию — воспользоваться функцией `np.sum`. В этом случае `False` интерпретируется как `0`, а `True` — как `1`:

```
In[16]: np.sum(x < 6)

Out[16]: 8
```

Преимущество функции `np.sum` заключается в том, что, подобно другим функциям агрегирования библиотеки NumPy, это суммирование можно выполнять также по столбцам или строкам:

```
In[17]: # Сколько значений меньше 6 содержится в каждой строке?
        np.sum(x < 6, axis=1)

Out[17]: array([4, 2, 2])
```

Этот оператор подсчитывает количество значений меньше 6 в каждой строке матрицы.

Если вам необходимо быстро проверить, существует ли хоть одно истинное значение, или все ли значения истинны, можно воспользоваться (как вы наверняка догадались) функциями `np.any()` и `np.all()`:

```
In[18]: # Имеются ли в массиве какие-либо значения, превышающие 8?
        np.any(x > 8)

Out[18]: True

In[19]: # Имеются ли в массиве какие-либо значения меньше 0?
        np.any(x < 0)

Out[19]: False
```

```
In[20]: # Все ли значения меньше 10?
        np.all(x < 10)

Out[20]: True

In[21]: # Все ли значения равны 6?
        np.all(x == 6)

Out[21]: False
```

Функции `np.any()` и `np.all()` также можно использовать по конкретным осям. Например:

```
In[22]: # Все ли значения в каждой строке меньше 8?
        np.all(x < 8, axis=1)

Out[22]: array([ True, False,  True], dtype=bool)
```

В первой и третьей строках имеются значения меньше 8, а во второй — нет.

Наконец, небольшое предупреждение: как упоминалось в разделе «Агрегирование: минимум, максимум и все, что посередине» данной главы, в языке Python имеются встроенные функции `sum()`, `any()` и `all()`. Их синтаксис отличается от аналогичных функций библиотеки NumPy. В частности, они будут выдавать ошибку или неожиданные результаты при использовании для работы с многомерными массивами. Убедитесь, что вы применяете для данных примеров функции `np.sum()`, `np.any()` и `np.all()`.

Булевы операторы

Вы уже знаете, как можно подсчитать все дни с осадками менее четырех дюймов или все дни с осадками более двух дюймов. Но что, если нужна информация обо всех днях с толщиной слоя осадков менее четырех дюймов *и* более одного дюйма? Это можно сделать с помощью *побитовых логических операторов* (bitwise logic operators) языка Python: `&`, `|`, `^` и `~`. Аналогично обычным арифметическим операторам библиотека NumPy перегружает их как универсальные функции, поэлементно работающие с (обычно булевыми) массивами.

Например, можно решить подобную составную задачу следующим образом:

```
In[23]: np.sum((inches > 0.5) & (inches < 1))

Out[23]: 29
```

Видим, что в 2014 году в Сиэтле было 29 дней с толщиной слоя осадков от 0.5 до 1 дюйма.

Обратите внимание, что скобки здесь важны в силу правил приоритета операторов, без скобок это выражение вычислялось бы следующим образом, что привело бы к ошибке:

```
inches > (0.5 & inches) < 1
```

Воспользовавшись эквивалентностью выражений `А И В` и `НЕ(А ИЛИ В)`, о которой вы, возможно, помните из вводного курса математической логики, можно вычислить тот же результат и другим образом:

```
In[24]: np.sum(~( (inches <= 0.5) | (inches >= 1) ))

Out[24]: 29
```

Объединив операторы сравнения и булевы операторы при работе с массивами, можно получить целый диапазон эффективных логических операций (табл. 2.5).

Таблица 2.5. Побитовые булевы операторы и эквивалентные им универсальные функции

Оператор	Эквивалентная универсальная функция
&	np.bitwise_and
|	np.bitwise_or
^	np.bitwise_xor
~	np.bitwise_not

С помощью этих инструментов можно начать отвечать на различные типы вопросов относительно наших данных по осадкам. Вот примеры результатов, которые можно вычислить путем сочетания маскирования с агрегированием:

```
In[25]: print("Number days without rain: ",       np.sum(inches == 0))
        print("Number days with rain: ",          np.sum(inches != 0))
        print("Days with more than 0.5 inches:", np.sum(inches > 0.5))
        print("Rainy days with < 0.1 inches :",   np.sum((inches > 0) &
                                                         (inches < 0.2)))
Number days without rain:       215
Number days with rain:          150
Days with more than 0.5 inches: 37
Rainy days with < 0.1 inches :  75
```

Булевы массивы как маски

В предыдущем разделе мы рассматривали сводные показатели, вычисленные непосредственно из булевых массивов. Гораздо больше возможностей дает использование булевых массивов в качестве масок, для выбора нужных подмножеств самих данных. Вернемся к массиву `x` и допустим, что нам необходим массив всех значений из массива `x`, меньше чем, скажем, `5`:

```
In[26]: x

Out[26]: array([[5, 0, 3, 3],
                [7, 9, 3, 5],
                [2, 4, 7, 6]])
```

Как мы уже видели, можно легко получить булев массив для этого условия:

```
In[27]: x < 5

Out[27]: array([[False,  True,  True,  True],
                [False, False,  True, False],
                [ True,  True, False, False]], dtype=bool)
```

Чтобы *выбрать* нужные значения из массива, достаточно просто проиндексировать исходный массив `x` по этому булеву массиву. Такое действие носит название операции *наложения маски* или *маскирования*:

```
In[28]: x[x < 5]

Out[28]: array([0, 3, 3, 3, 2, 4])
```

При этом был возвращен одномерный массив, заполненный всеми значениями, удовлетворяющими нашему условию. Другими словами, все значения, находящиеся в массиве `x` на позициях, на которых в массиве-маске находятся значения `True`.

После этого можно поступать с этими значениями так, как нам будет нужно. Например, можно вычислить какой-нибудь соответствующий статистический показатель на наших данных о дождях в Сиэтле:

```
In[29]:
# создаем маску для всех дождливых дней
rainy = (inches > 0)
# создаем маску для всех летних дней (21 июня[1] – 172-й день)
summer = (np.arange(365) - 172 < 90) & (np.arange(365) - 172 > 0)

print("Median precip on rainy days in 2014 (inches): ",
      np.median(inches[rainy]))
print("Median precip on summer days in 2014 (inches): ",
      np.median(inches[summer]))
print("Maximum precip on summer days in 2014 (inches): ",
      np.max(inches[summer]))
print("Median precip on non-summer rainy days (inches):",
      np.median(inches[rainy & ~summer]))

Median precip on rainy days in 2014 (inches): 0.194881889764
Median precip on summer days in 2014 (inches): 0.0
Maximum precip on summer days in 2014 (inches): 0.850393700787
Median precip on non-summer rainy days (inches): 0.200787401575
```

[1] Астрономическое лето в Северном полушарии начинается 21 июня, в день летнего сонцестояния.

Путем сочетания булевых операций, операций маскирования и агрегирования можно очень быстро и легко отвечать на подобные вопросы относительно нашего набора данных.

Использование ключевых слов and/or по сравнению с использованием операторов &\|

Разница между ключевыми словами `AND` и `OR` и операторами `&` и `|` — распространенный источник путаницы. Какие из них и когда следует использовать?

Различие заключается в следующем: ключевые слова `AND` и `OR` определяют истинность или ложность всего объекта, операторы `&` и `|` оперируют *отдельными битами внутри каждого из объектов*.

Использование ключевых слов `and` и `or` приводит к тому, что язык Python будет рассматривать объект как одну булеву сущность. В Python все ненулевые целые числа будут рассматриваться как `True`. Таким образом:

```
In[30]: bool(42), bool(0)
Out[30]: (True, False)
In[31]: bool(42 and 0)

Out[31]: False

In[32]: bool(42 or 0)

Out[32]: True
```

При использовании операторов `&` и `|` для работы с целыми числами выражение оперирует разрядами элемента, фактически применяя операции `and` и `or` к составляющим число отдельным битам:

```
In[33]: bin(42)

Out[33]: '0b101010'

In[34]: bin(59)

Out[34]: '0b111011'

In[35]: bin(42 & 59)

Out[35]: '0b101010'
In[36]: bin(42 | 59)

Out[36]: '0b111011'
```

Обратите внимание, что для получения результата сравниваются соответствующие биты двоичного представления чисел.

Массив булевых значений в библиотеке NumPy можно рассматривать как строку битов, где `1 = True` и `0 = False`, а результат применения операторов `&` и `|` аналогичен вышеприведенному:

```
In[37]: A = np.array([1, 0, 1, 0, 1, 0], dtype=bool)
        B = np.array([1, 1, 1, 0, 1, 1], dtype=bool)
        A | B

Out[37]: array([ True,  True,  True, False,  True,  True],
        dtype=bool)
```

Использование же ключевого слова `or` для этих массивов приведет к вычислению истинности или ложности всего объекта массива — не определенного формально значения:

```
In[38]: A or B

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)

<ipython-input-38-5d8e4f2e21c0> in <module>()
----> 1 A or B

ValueError: The truth value of an array with more than one element is...
```

При создании булева выражения с заданным массивом следует использовать операторы `&` и `|`, а не операции `and` или `or`:

```
In[39]: x = np.arange(10)
        (x > 4) & (x < 8)

Out[39]: array([False, False, ...,  True,  True, False, False],
               dtype=bool)
```

Попытка же вычислить истинность или ложность всего массива приведет к уже наблюдавшейся нами выше ошибке `ValueError`:

```
In[40]: (x > 4) and (x < 8)

---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-40-3d24f1ffd63d> in <module>()
----> 1 (x > 4) and (x < 8)

ValueError: The truth value of an array with more than one element is...
```

Итак, запомните: операции `and` и `or` вычисляют единое булево значение для всего объекта, в то время как операторы `&` и `|` вычисляют много булевых значений для содержимого (отдельных битов или байтов) объекта. Второй из этих вариантов практически всегда будет именно той операцией, которая будет вам нужна при работе с булевыми массивами библиотеки NumPy.

«Прихотливая» индексация

В предыдущих разделах мы рассмотрели доступ и изменение частей массива с помощью простых индексов (например, `arr[0]`), срезов (например, `arr[:5]`) и булевых масок (например, `arr[arr > 0]`). В этом разделе мы изучим другую разновидность индексации массивов, так называемую *прихотливую индексацию* (fancy indexing). «Прихотливая» индексация похожа на уже рассмотренную нами простую индексацию, но вместо скалярных значений передаются массивы индексов. Это дает возможность очень быстрого доступа и модификации сложных подмножеств значений массива.

Исследуем возможности «прихотливой» индексации

Суть «прихотливой» индексации проста: она заключается в передаче массива индексов с целью одновременного доступа к нескольким элементам массива. Например, рассмотрим следующий массив:

```
In[1]: import numpy as np
       rand = np.random.RandomState(42)

       x = rand.randint(100, size=10)
       print(x)

[51 92 14 71 60 20 82 86 74 74]
```

Допустим, нам требуется обратиться к трем различным элементам. Можно сделать это следующим образом:

```
In[2]: [x[3], x[7], x[2]]

Out[2]: [71, 86, 14]
```

С другой стороны, можно передать единый список индексов и получить тот же результат:

```
In[3]: ind = [3, 7, 4]
       x[ind]

Out[3]: array([71, 86, 60])
```

В случае «прихотливой» индексации форма результата отражает форму *массивов индексов* (index arrays), а не форму *индексируемого массива*:

```
In[4]: ind = np.array([[3, 7],
                       [4, 5]])
       x[ind]
```

```
Out[4]: array([[71, 86],
               [60, 20]])
```

«Прихотливая» индексация работает и в случае многомерных массивов. Рассмотрим следующий массив:

```
In[5]: X = np.arange(12).reshape((3, 4))
       X

Out[5]: array([[ 0,  1,  2,  3],
               [ 4,  5,  6,  7],
               [ 8,  9, 10, 11]])
```

Аналогично обычной индексации первый индекс относится к строкам, а второй — к столбцам:

```
In[6]: row = np.array([0, 1, 2])
       col = np.array([2, 1, 3])
       X[row, col]

Out[6]: array([ 2,  5, 11])
```

Первое значение в результате — `X[0, 2]`, второе — `X[1, 1]`, и третье — `X[2, 3]`. Составление пар индексов при «прихотливой» индексации подчиняется всем правилам транслирования, описанным в разделе «Операции над массивами. Транслирование» данной главы. Так, например, если мы скомбинируем вектор-столбец и вектор-строку в индексах, то получим двумерный результат:

```
In[7]: X[row[:, np.newaxis], col]

Out[7]: array([[ 2,  1,  3],
               [ 6,  5,  7],
               [10,  9, 11]])
```

Каждое строчное значение соединяется с каждым вектором-столбцом точно так же, как при транслировании арифметических операций. Например:

```
In[8]: row[:, np.newaxis] * col

Out[8]: array([[0, 0, 0],
               [2, 1, 3],
               [4, 2, 6]])
```

При работе с «прихотливой» индексацией важно никогда не забывать, что возвращаемое значение отражает *транслируемую форму индексов*, а не форму индексируемого массива.

Комбинированная индексация

Для реализации еще более сложных операций «прихотливую» индексацию можно использовать совместно с другими схемами индексации:

```
In[9]: print(X)

[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

Можно применять совместно «прихотливые» и простые индексы:

```
In[10]: X[2, [2, 0, 1]]

Out[10]: array([10,  8,  9])
```

Можно также использовать совместно «прихотливые» индексы и срезы:

```
In[11]: X[1:, [2, 0, 1]]

Out[11]: array([[ 6,  4,  5],
                [10,  8,  9]])
```

И можно применять совместно «прихотливую» индексацию и маскирование:

```
In[12]: mask = np.array([1, 0, 1, 0], dtype=bool)
        X[row[:, np.newaxis], mask]

Out[12]: array([[ 0,  2],
                [ 4,  6],
                [ 8, 10]])
```

Все эти варианты индексации вместе обеспечивают набор чрезвычайно гибких операций по доступу к значениям массивов и их изменению.

Пример: выборка случайных точек

Частая сфера применения «прихотливой» индексации — выборка подмножеств строк из матрицы. Пусть у нас имеется матрица размером `N` на `D`, представляющая `N` точек в `D` измерениях, например следующие точки, полученные из двумерного нормального распределения:

```
In[13]: mean = [0, 0]
        cov = [[1, 2],
               [2, 5]]
        X = rand.multivariate_normal(mean, cov, 100)
        X.shape

Out[13]: (100, 2)
```

С помощью инструментов рисования графиков, которые мы обсудим в главе 4, можно визуализировать эти точки в виде диаграммы рассеяния (рис. 2.7):

```
In[14]: %matplotlib inline
        import matplotlib.pyplot as plt
        import seaborn; seaborn.set()  # for plot styling

        plt.scatter(X[:, 0], X[:, 1]);
```

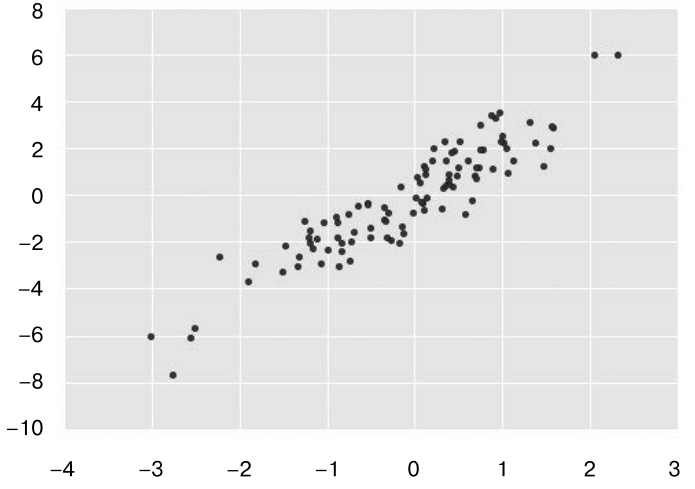

Рис. 2.7. Нормально распределенные точки

Воспользуемся «прихотливой» индексацией для выборки 20 случайных точек. Мы сделаем это с помощью выбора предварительно 20 случайных индексов без повторов и воспользуемся этими индексами для выбора части исходного массива:

```
In[15]: indices = np.random.choice(X.shape[0], 20, replace=False)
        indices

Out[15]: array([93, 45, 73, 81, 50, 10, 98, 94,  4, 64, 65, 89, 47, 84, 82,
                80, 25, 90, 63, 20])

In[16]: selection = X[indices]  # Тут используется «прихотливая» индексация
        selection.shape

Out[16]: (20, 2)
```

Чтобы посмотреть, какие точки были выбраны, нарисуем поверх первой диаграммы большие круги в местах расположения выбранных точек (рис. 2.8).

```
In[17]: plt.scatter(X[:, 0], X[:, 1], alpha=0.3)
        plt.scatter(selection[:, 0], selection[:, 1],
                    facecolor='none', s=200);
```

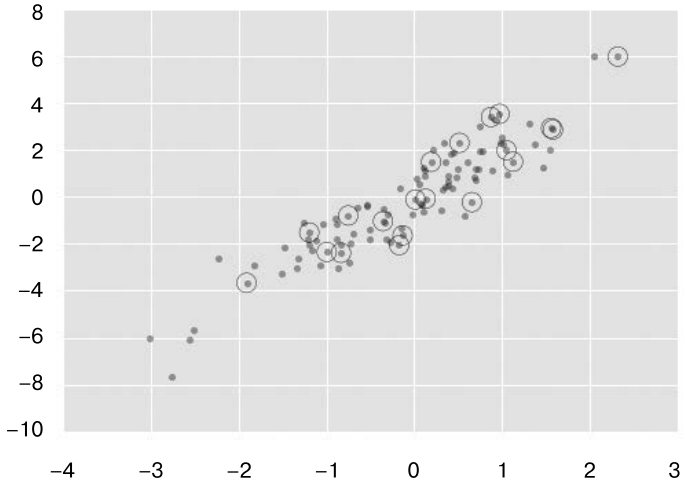

Рис. 2.8. Случайно выбранные точки

Подобная стратегия часто используется для быстрого секционирования наборов данных, часто требуемого при разделении на обучающую/тестовую последовательности для проверки статистических моделей (см. раздел «Гиперпараметры и проверка модели» главы 5), а также в выборочных методах ответа на статистические вопросы.

Изменение значений с помощью прихотливой индексации

Аналогично тому, как «прихотливую» индексацию можно использовать для доступа к частям массива, ее можно применять и для модификации частей массива. Например, допустим, что у нас есть массив индексов и нам нужно присвоить соответствующим элементам массива какие-то значения:

```
In[18]: x = np.arange(10)
        i = np.array([2, 1, 8, 4])

        x[i] = 99
        print(x)

[ 0 99 99  3 99  5  6  7 99  9]
```

Для этого можно использовать любой из операторов присваивания. Например:

```
In[19]: x[i] -= 10
        print(x)


[ 0 89 89  3 89  5  6  7 89  9]
```

Замечу, однако, что повторяющиеся индексы при подобных операциях могут привести к некоторым потенциально неожиданным результатам. Рассмотрим следующий пример:

```
In[20]: x = np.zeros(10)
        x[[0, 0]] = [4, 6]
        print(x)

[ 6.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
```

Куда пропало 4? В результате этой операции сначала выполняется присваивание `x[0] = 4` с последующим присваиванием `x[0] = 6`. В итоге `x[0]` содержит значение `6`.

Довольно логично, но рассмотрим такую операцию:

```
In[21]: i = [2, 3, 3, 4, 4, 4]
        x[i] += 1
        x

Out[21]: array([ 6.,  0.,  1.,  1.,  1.,  0.,  0.,  0.,  0.,  0.])
```

Можно было ожидать, что `x[3]` будет содержать значение `2`, а `x[4]` — значение `3`, так как именно столько раз повторяется каждый из этих индексов. Почему же это не так? По сути, не из-за того, что выражение `x[i] += 1` задумывалось как сокращенная форма записи для `x[i] = x[i] + 1`. Вычисляется выражение `x[i] + 1`, после чего результат присваивается соответствующим индексам элементам в массиве `x`. Получается, что это не выполняемый несколько раз инкремент, а присваивание, приводящее к интуитивно не очевидным результатам.

Что же делать, если требуется другое поведение при повторяющейся операции? В этом случае можно воспользоваться методом `at()` универсальных функций (доступен начиная с версии 1.8 библиотеки NumPy) и сделать следующее:

```
In[22]: x = np.zeros(10)
        np.add.at(x, i, 1)
        print(x)

[ 0.  0.  1.  2.  3.  0.  0.  0.  0.  0.]
```

Метод `at()` применяет соответствующий оператор к элементам с заданными индексами (в данном случае `i`) с использованием заданного значения (в данном случае `1`). Аналогичный по духу метод универсальных функций `reduceat()`, о котором можно прочитать в документации библиотеки NumPy.

Пример: разбиение данных на интервалы

Можно использовать эти идеи для эффективного разбиения данных с целью построения гистограммы вручную. Например, пусть у нас есть 1000 значений и нам

необходимо быстро выяснить, как они распределяются по массиву интервалов. Это можно сделать с помощью метода `at()` универсальных функций следующим образом:

```
In[23]: np.random.seed(42)
        x = np.random.randn(100)

        # Рассчитываем гистограмму вручную
        bins = np.linspace(-5, 5, 20)
        counts = np.zeros_like(bins)

        # Ищем подходящий интервал для каждого x
        i = np.searchsorted(bins, x)

        # Добавляем 1 к каждому из интервалов
        np.add.at(counts, i, 1)
```

Полученные числа отражают количество точек в каждом из интервалов, другими словами, гистограмму (рис. 2.9):

```
In[24]: # Визуализируем результаты
        plt.plot(bins, counts, linestyle='steps');
```

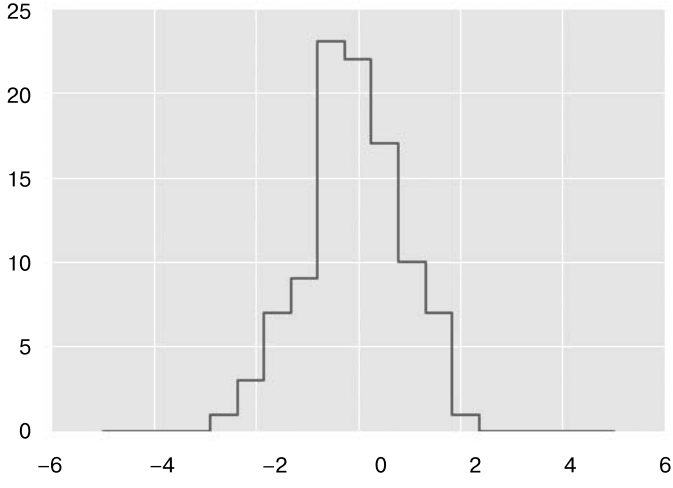

Рис. 2.9. Вычисленная вручную гистограмма

Получать каждый раз гистограмму таким образом нет необходимости. Библиотека Matplotlib предоставляет процедуру `plt.hist()`, которая делает то же самое одной строкой кода:

```
plt.hist(x, bins, histtype='step');
```

Эта функция создаст практически точно такую же диаграмму, как на рис. 2.9. Для расчета разбиения по интервалам библиотека Matplotlib использует функцию `np.histogram`, выполняющую вычисления, очень похожие на сделанные нами. Давайте сравним их:

```
In[25]: print("NumPy routine:")
        %timeit counts, edges = np.histogram(x, bins)
        print("Custom routine:")
        %timeit np.add.at(counts, np.searchsorted(bins, x), 1)

NumPy routine:
10000 loops, best of 3: 97.6 µs per loop
Custom routine:
10000 loops, best of 3: 19.5 µs per loop
```

Наш собственный однострочный алгоритм работает в несколько раз быстрее, чем оптимизированный алгоритм из библиотеки NumPy! Как это возможно? Если мы заглянем в исходный код процедуры `np.histogram` (в оболочке IPython это можно сделать, введя команду `np.histogram??`), то увидим, что она гораздо сложнее простого поиска-и-подсчета, выполненного нами. Дело в том, что алгоритм из библиотеки NumPy более гибок, потому что разработан с ориентацией на более высокую производительность при значительном увеличении количества точек данных:

```
In[26]: x = np.random.randn(1000000)
        print("NumPy routine:")
        %timeit counts, edges = np.histogram(x, bins)

        print("Custom routine:")
        %timeit np.add.at(counts, np.searchsorted(bins, x), 1)

NumPy routine:
10 loops, best of 3: 68.7 ms per loop
Custom routine:
10 loops, best of 3: 135 ms per loop
```

Это сравнение демонстрирует нам, что эффективность алгоритма почти всегда непростой вопрос. Эффективный для больших наборов данных алгоритм не всегда окажется оптимальным вариантом для маленьких, и наоборот (см. врезку «Нотация “О-большого”» далее). Но преимущество самостоятельного программирования этого алгоритма заключается в том, что, получив понимание работы подобных простых методов, вы сможете «строить» из этих «кирпичиков» очень интересные варианты пользовательского поведения. Ключ к эффективному использованию языка Python в приложениях, требующих обработки больших объемов данных, заключается в том, чтобы знать о существовании удобных процедур, таких как `np.histogram`, и сферах их использования. Кроме того, нужно знать, как применять низкоуровневую функциональность при необходимости в узконаправленном поведении.

Сортировка массивов

До сих пор мы занимались в основном инструментами доступа и изменения данных массивов с помощью библиотеки NumPy. Этот раздел охватывает алгоритмы, связанные с сортировкой значений в массивах библиотеки NumPy. Данные алгоритмы — излюбленная тема вводных курсов по вычислительной технике. Если вы когда-либо были слушателем такого курса, то вам, наверное, снились сны (в зависимости от вашего темперамента, возможно, кошмары) о сортировке вставкой, сортировке методом выбора, сортировке слиянием, быстрой сортировке, пузырьковой сортировке и многих других. Все они представляют собой средства для решения одной и той же задачи — сортировки значений в списке или массиве.

Например, простая *сортировка вставкой* (insertion sort) многократно находит минимальное значение из списка и выполняет перестановки до тех пор, пока список не будет отсортирован. Это можно запрограммировать с помощью всего нескольких строк кода на языке Python:

```
In[1]: import numpy as np

       def selection_sort(x):
           for i in range(len(x)):
               swap = i + np.argmin(x[i:])
               (x[i], x[swap]) = (x[swap], x[i])
           return x

In[2]: x = np.array([2, 1, 4, 3, 5])
       selection_sort(x)

Out[2]: array([1, 2, 3, 4, 5])
```

Сортировка вставкой удобна из-за своей простоты, но слишком медлительна, чтобы подходить для массивов большего размера. Для списка из N значений она потребует N циклов, каждый из них выполняет порядка $\sim N$ сравнений для поиска значения, которое нужно переставить. На языке нотации «О-большого», часто используемой для описания характеристик этих алгоритмов (см. врезку «Нотация “О-большого”» далее), временная сложность сортировки вставкой в среднем имеет порядок $O[N^2]$: при удвоении количества элементов списка время выполнения вырастет примерно в четыре раза.

Даже сортировка выбором гораздо лучше моего фаворита среди всех алгоритмов сортировки — *случайной сортировки* (bogosort):

```
In[3]: def bogosort(x):
           while np.any(x[:-1] > x[1:]):
               np.random.shuffle(x)
           return x
```

```
In[4]: x = np.array([2, 1, 4, 3, 5])
       bogosort(x)

Out[4]: array([1, 2, 3, 4, 5])
```

Этот алгоритм сортировки опирается в своей работе на чистое везение: он многократно перетасовывает массив случайным образом до тех пор, пока результат не окажется отсортированным. При средней сложности порядка $O[N \times N!]$: (это N умножить на N факториал) его не стоит использовать ни для каких реальных расчетов.

В Python имеются *намного* более эффективные встроенные алгоритмы сортировки. Начнем с изучения встроенных алгоритмов языка Python, после чего рассмотрим утилиты, включенные в библиотеку NumPy и оптимизированные под NumPy-массивы.

Быстрая сортировка в библиотеке NumPy: функции np.sort и np.argsort

Хотя в языке Python имеются встроенные функции `sort` и `sorted` для работы со списками, мы не будем их рассматривать, поскольку функция библиотеки NumPy `np.sort` оказывается намного более эффективной и подходящей для наших целей. По умолчанию функция `np.sort` использует имеющий сложность $O[N \log N]$: алгоритм *быстрой сортировки* (quicksort), хотя доступны для использования также алгоритмы сортировки *слиянием* (mergesort) и *пирамидальной сортировки* (heapsort). Для большинства приложений используемой по умолчанию быстрой сортировки более чем достаточно.

Чтобы получить отсортированную версию входного массива без его изменения, можно использовать функцию `np.sort`:

```
In[5]: x = np.array([2, 1, 4, 3, 5])
       np.sort(x)

Out[5]: array([1, 2, 3, 4, 5])
```

Если же вы предпочитаете отсортировать имеющийся массив, то можно вместо этого применять метод `sort` массивов:

```
In[6]: x.sort()
       print(x)

[1 2 3 4 5]
```

Имеется также родственная функция `argsort`, возвращающая *индексы* отсортированных элементов:

```
In[7]: x = np.array([2, 1, 4, 3, 5])
       i = np.argsort(x)
       print(i)

[1 0 3 2 4]
```

Первый элемент этого результата соответствует индексу минимального элемента, второй — индексу второго по величине и т. д. В дальнейшем эти индексы при желании можно будет использовать для построения (посредством «прихотливой» индексации) отсортированного массива:

```
In[8]: x[i]

Out[8]: array([1, 2, 3, 4, 5])
```

Сортировка по строкам и столбцам. У алгоритмов сортировки библиотеки NumPy имеется удобная возможность выполнять сортировку по конкретным строкам или столбцам многомерного массива путем задания аргумента `axis`. Например:

```
In[9]: rand = np.random.RandomState(42)
       X = rand.randint(0, 10, (4, 6))
       print(X)

[[6 3 7 4 6 9]
 [2 6 7 4 3 7]
 [7 2 5 4 1 7]
 [5 1 4 0 9 5]]

In[10]: # Сортируем все столбцы массива X
        np.sort(X, axis=0)

Out[10]: array([[2, 1, 4, 0, 1, 5],
                [5, 2, 5, 4, 3, 7],
                [6, 3, 7, 4, 6, 7],
                [7, 6, 7, 4, 9, 9]])

In[11]: # Сортируем все строки массива X
        np.sort(X, axis=1)

Out[11]: array([[3, 4, 6, 6, 7, 9],
                [2, 3, 4, 6, 7, 7],
                [1, 2, 4, 5, 7, 7],
                [0, 1, 4, 5, 5, 9]])
```

Не забывайте, что при этом все строки или столбцы рассматриваются как отдельные массивы, так что любые возможные взаимосвязи между значениями строк или столбцов будут утеряны.

Частичные сортировки: секционирование

Иногда нам не требуется сортировать весь массив, а просто нужно найти `K` наименьших значений в нем. Библиотека NumPy предоставляет для этой цели функцию `np.partition`. Функция `np.partition` принимает на входе массив и число `K`. Результат представляет собой новый массив с `K` наименьшими значениями слева от точки разбиения и остальные значения справа от нее в произвольном порядке:

```
In[12]: x = np.array([7, 2, 3, 1, 6, 5, 4])
        np.partition(x, 3)

Out[12]: array([2, 1, 3, 4, 6, 5, 7])
```

Первые три значения в итоговом массиве — три наименьших значения в нем, а на остальных позициях массива располагаются все прочие значения. Внутри каждой из двух секций элементы располагаются в произвольном порядке.

Аналогично сортировке можно секционировать по произвольной оси многомерного массива:

```
In[13]: np.partition(X, 2, axis=1)

Out[13]: array([[3, 4, 6, 7, 6, 9],
                [2, 3, 4, 7, 6, 7],
                [1, 2, 4, 5, 7, 7],
                [0, 1, 4, 5, 9, 5]])
```

Результат представляет собой массив, в котором на первых двух позициях в каждой строке находятся наименьшие значения из этой строки, а остальные значения заполняют прочие места.

Наконец, аналогично функции `np.argsort`, вычисляющей индексы для сортировки, существует функция `np.argpartition`, вычисляющая индексы для секции. Мы увидим ее в действии в следующем разделе.

Пример: K ближайших соседей

Давайте вкратце рассмотрим, как можно использовать функцию `np.argpartition` по нескольким осям для поиска ближайших соседей каждой точки из определенного набора. Начнем с создания случайного набора из десяти точек на двумерной плоскости. По стандартным соглашениям образуем из них массив 10×2:

```
In[14]: X = rand.rand(10, 2)
```

Чтобы наглядно представить себе расположение этих точек, нарисуем для них диаграмму рассеяния (рис. 2.10):

```
In[15]: %matplotlib inline
        import matplotlib.pyplot as plt
        import seaborn; seaborn.set() # Plot styling
        plt.scatter(X[:, 0], X[:, 1], s=100);
```

Теперь можно вычислить расстояние между всеми парами точек. Вспоминаем, что квадрат расстояния между двумя точками равен сумме квадратов расстояний между ними по каждой из координат. Воспользовавшись возможностями эффективного транслирования (см. «Операции над массивами. Транслирование» этой главы) и агрегирования (см. «Агрегирование: минимум, максимум и все, что

посередине» данной главы), предоставляемыми библиотекой NumPy, мы можем вычислить матрицу квадратов расстояний с помощью одной строки кода:

```
In[16]: dist_sq = np.sum((X[:,np.newaxis,:] - X[np.newaxis,:,:])
                         ** 2, axis=-1)
```

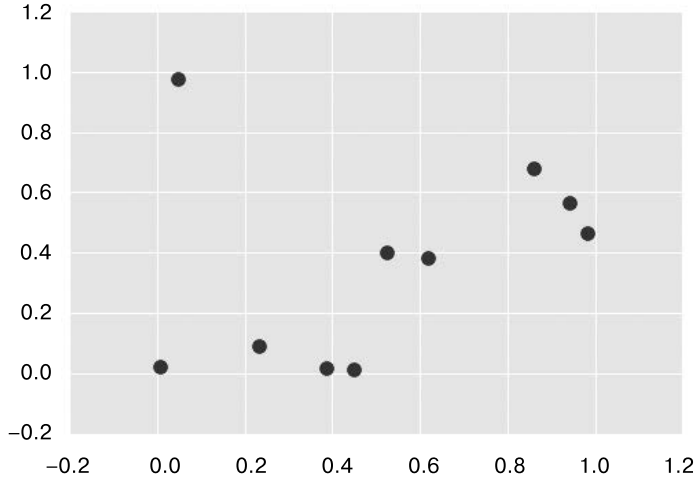

Рис. 2.10. Визуализация точек в примере K соседей

Эта операция довольно сложна по синтаксису и может привести вас в замешательство, если вы плохо знакомы с правилами транслирования библиотеки NumPy. В подобных случаях полезно разбить операцию на отдельные шаги:

```
In[17]: # Для каждой пары точек вычисляем разности их координат
        differences = X[:, np.newaxis, :] - X[np.newaxis, :, :]
        differences.shape

Out[17]: (10, 10, 2)

In[18]: # Возводим разности координат в квадрат
        sq_differences = differences ** 2
        sq_differences.shape

Out[18]: (10, 10, 2)

In[19]: # Суммируем квадраты разностей координат
        # для получения квадрата расстояния
        dist_sq = sq_differences.sum(-1)
        dist_sq.shape

Out[19]: (10, 10)
```

На всякий случай для контроля проверим, что диагональ матрицы (то есть набор расстояний между каждой точкой и ей самой) состоит из нулей:

```
In[20]: dist_sq.diagonal()

Out[20]: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
```

Проверка пройдена! Теперь, получив матрицу квадратов расстояний между взятыми попарно точками, мы можем воспользоваться функцией `np.argsort` для сортировки по каждой строке. Крайние слева столбцы будут представлять собой индексы ближайших соседей:

```
In[21]: nearest = np.argsort(dist_sq, axis=1)
        print(nearest)


[[0 3 9 7 1 4 2 5 6 8]
 [1 4 7 9 3 6 8 5 0 2]
 [2 1 4 6 3 0 8 9 7 5]
 [3 9 7 0 1 4 5 8 6 2]
 [4 1 8 5 6 7 9 3 0 2]
 [5 8 6 4 1 7 9 3 2 0]
 [6 8 5 4 1 7 9 3 2 0]
 [7 9 3 1 4 0 5 8 6 2]
 [8 5 6 4 1 7 9 3 2 0]
 [9 7 3 0 1 4 5 8 6 2]]
```

Обратите внимание, что первый столбец представляет собой числа с 0 до 9 в порядке возрастания: это происходит из-за того, что ближайший сосед каждой точки — она сама, как и можно было ожидать.

Выполнив полную сортировку, мы проделали лишнюю работу. Если нас интересовали `K` ближайших соседей, было достаточно секционировать все строки так, чтобы сначала шли `K+1` минимальных квадратов расстояний, а бóльшие расстояния заполняли оставшиеся позиции массива. Сделать это можно с помощью функции `np.argpartition`:

```
In[22]: K = 2
        nearest_partition = np.argpartition(dist_sq, K + 1, axis=1)
```

Чтобы визуализировать эту сетку соседей, выведем на диаграмму точки вдоль линий, связывающих каждую точку с ее ближайшими двумя соседями (рис. 2.11):

```
In[23]: plt.scatter(X[:, 0], X[:, 1], s=100)

        # Рисуем линии из каждой точки к ее двум ближайшим соседям
        K = 2

        for i in range(X.shape[0]):
            for j in nearest_partition[i, :K+1]:
```

```
# чертим линию от X[i] до X[j]
# Используем для этого «магическую» функцию zip:
plt.plot(*zip(X[j], X[i]), color='black')
```

От каждой нарисованной на диаграмме точки ведут линии к двум ее ближайшим соседям. На первый взгляд может показаться странным, что из некоторых точек отходит более двух линий. Дело в том, что, если точка A — один из двух ближайших соседей точки B, вовсе не обязательно, что точка B — один из двух ближайших соседей точки A.

Хотя применяемые при этом транслирование и построчная сортировка могут показаться более запутанным подходом, чем написание цикла, оказывается, что такой способ работы с подобными данными на языке Python весьма эффективен. Как бы ни было заманчиво сделать то же самое, вручную организовав цикл по данным и сортировку каждого набора соседей отдельно, получившийся в итоге алгоритм почти наверняка будет работать медленнее, чем рассмотренная выше векторизованная версия. Красота такого подхода — в его независимости от размера входных данных: можно с одинаковой легкостью вычислить соседей среди 100 или 1 000 000 точек в любом количестве измерений, и код будет выглядеть точно так же.

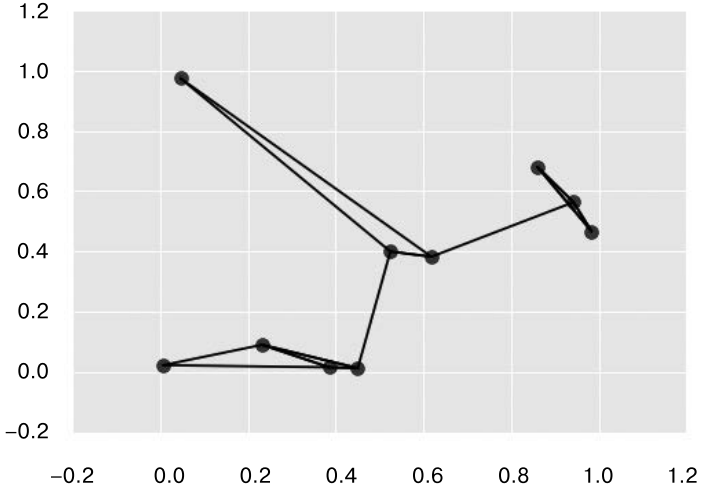

Рис. 2.11. Визуализация соседей каждой точки

Наконец, отмечу, что для выполнения поисков соседей в очень больших массивах данных существуют основанные на деревьях и/или аппроксимационные алгоритмы, масштабирующиеся как $O[N \log N]$: или даже лучше, в отличие от грубого подхода $O[N^2]$:. Один из примеров таких алгоритмов — K-мерное дерево (KD-tree), реализованное в библиотеке Scikit-Learn.

Нотация «О-большого»

Нотация «О-большого» — средство, позволяющее описывать рост числа операций, необходимых для выполнения алгоритма, по мере роста объема входных данных. Для правильного использования нужно немного углубиться в теорию вычислительной техники и уметь отличать данную нотацию от родственных нотаций «о-маленького», «θ-большого», «Ω-большого» и, вероятно, множества их гибридов. Хотя эти варианты позволяют с большей точностью выражать информацию о масштабируемости алгоритмов, помимо экзаменов по теории вычислительной техники и замечаний педантичных комментаторов в блогах, их редко где можно увидеть. Намного более распространенной в мире науки о данных является более гибкая нотация «О-большого»: общее (хотя и не такое точное) описание масштабируемости алгоритма. Простите нас, теоретики и педанты, но именно эту интерпретацию мы и будем использовать в данной книге.

Нотация «О-большого» показывает, во сколько раз больше времени будет занимать выполнение алгоритма при росте количества данных. Если ваш алгоритм $O[N]$ (читается «порядка N») выполняется 1 секунду при работе со списком длины $N = 1000$, то следует ожидать, что его выполнение займет примерно 5 секунд для списка длиной $N = 5000$. Если же у вас алгоритм $O[N^2]$ (читается «порядка N квадрат») выполняется 1 секунду при работе со списком длины $N = 1000$, то следует ожидать, что его выполнение займет примерно 25 секунд для списка длиной $N = 5000$.

Для наших целей N будет обычно обозначать какой-либо аспект размера набора данных (количество точек, количество измерений и т. п.). При попытке анализа миллиардов или триллионов выборок разница между сложностью $O[N]$ и $O[N^2]$ может быть более чем существенной!

Обратите внимание, что нотация «О-большого» сама по себе ничего не говорит о фактическом времени выполнения вычислений, а только о его масштабировании при изменении N. Обычно, например, алгоритм со сложностью $O[N]$ считается лучше масштабируемым, чем алгоритм с $O[N^2]$, и на то есть веские причины. Но, в частности, для маленьких наборов данных лучше масштабируемый алгоритм не обязательно будет более быстрым. Например, при работе с конкретной задачей алгоритм с $O[N^2]$ может выполняться 0,01 секунды, а «лучший» алгоритм $O[N]$ — 1 секунду. Увеличьте, однако, N на три порядка, и алгоритм $O[N]$ окажется победителем.

Даже такая упрощенная версия нотации «О-большого» может оказаться очень удобной для сравнения производительности алгоритмов, и мы будем использовать эту нотацию в нашей книге, где будет идти речь о масштабируемости алгоритмов.

Структурированные данные: структурированные массивы библиотеки NumPy

Часто данные можно представить с помощью однородного массива значений, но иногда это не удается. В этом разделе демонстрируется использование таких возможностей библиотеки NumPy, как *структурированные массивы* (structured arrays)

и *массивы записей* (record arrays), обеспечивающих эффективное хранилище для сложных неоднородных данных. Хотя демонстрируемые паттерны удобны для простых операций, подобные сценарии часто применяются и при работе со структурами данных `DataFrame` из библиотеки Pandas, которые мы рассмотрим в главе 3.

Пускай у нас имеется несколько категорий данных (например, имя, возраст и вес) о нескольких людях и мы хотели бы хранить эти значения для использования в программе на языке Python. Можно сохранить их в отдельных массивах:

```
In[2]: name = ['Alice', 'Bob', 'Cathy', 'Doug']
       age = [25, 45, 37, 19]
       weight = [55.0, 85.5, 68.0, 61.5]
```

Однако это не очень правильное решение. Ничто не указывает на то, что массивы как-то связаны, лучше было бы использовать для хранения этих данных единую структуру. Библиотека NumPy может это осуществить посредством применения структурированных массивов, представляющих собой массивы с составным типом данных.

Вспомните, что ранее мы создавали простой массив с помощью следующего выражения:

```
In[3]: x = np.zeros(4, dtype=int)
```

Аналогично можно создать структурированный массив, применяя спецификацию сложного типа данных:

```
In[4]: # Используем для структурированного массива составной тип данных
       data = np.zeros(4, dtype={'names':('name', 'age', 'weight'),
                                 'formats':('U10', 'i4', 'f8')})
       print(data.dtype)
```

```
[('name', '<U10'), ('age', '<i4'), ('weight', '<f8')]
```

`'U10'` означает «строку в кодировке Unicode максимальной длины 10», `'i4'` — «4-байтное (то есть 32-битное) целое число», а `'f8'` — «8-байтное (то есть 64-битное) число с плавающей точкой». Мы обсудим другие варианты подобного кодирования типов в следующем разделе.

Создав пустой массив-контейнер, мы можем заполнить его нашим списком значений:

```
In[5]: data['name'] = name
       data['age'] = age
       data['weight'] = weight
       print(data)
```

```
[('Alice', 25, 55.0) ('Bob', 45, 85.5) ('Cathy', 37, 68.0)
 ('Doug', 19, 61.5)]
```

Как мы и хотели, данные теперь располагаются все вместе в одном удобном блоке памяти.

В структурированных массивах удобно то, что можно ссылаться на значения и по имени, и по индексу:

```
In[6]: # Извлечь все имена
       data['name']

Out[6]: array(['Alice', 'Bob', 'Cathy', 'Doug'],
              dtype='<U10')

In[7]: # Извлечь первую строку данных
       data[0]

Out[7]: ('Alice', 25, 55.0)

In[8]: # Извлечь имя из последней строки
       data[-1]['name']

Out[8]: 'Doug'
```

Появляется возможность с помощью булева маскирования выполнять и более сложные операции, такие как фильтрация по возрасту:

```
In[9]: # Извлечь имена людей с возрастом менее 30
       data[data['age'] < 30]['name']

Out[9]: array(['Alice', 'Doug'],
              dtype='<U10')
```

Для выполнения более сложных операций лучше использовать пакет Pandas, который будет рассмотрен в главе 3. Библиотека Pandas предоставляет объект `DataFrame` — основанную на массивах библиотеки NumPy структуру, обладающую массой полезной функциональности по работе с данными.

Создание структурированных массивов

Типы данных для структурированных массивов можно задавать несколькими способами. Ранее мы рассмотрели метод с использованием словаря:

```
In[10]: np.dtype({'names':('name', 'age', 'weight'),
                  'formats':('U10', 'i4', 'f8')})

Out[10]: dtype([('name', '<U10'), ('age', '<i4'), ('weight', '<f8')])
```

Для ясности можно задавать числовые типы как с применением типов данных языка Python, так и типов `dtype` библиотеки NumPy:

```
In[11]: np.dtype({'names':('name', 'age', 'weight'),
                  'formats':((np.str_, 10), int, np.float32)})

Out[11]: dtype([('name', '<U10'), ('age', '<i8'), ('weight', '<f4')])
```

Составные типы данных можно задавать в виде списка кортежей:

```
In[12]: np.dtype([('name', 'S10'), ('age', 'i4'), ('weight', 'f8')])

Out[12]: dtype([('name', 'S10'), ('age', '<i4'), ('weight', '<f8')])
```

Если названия типов для вас не важны, можете задать только сами типы данных в разделенной запятыми строке:

```
In[13]: np.dtype('S10,i4,f8')

Out[13]: dtype([('f0', 'S10'), ('f1', '<i4'), ('f2', '<f8')])
```

Сокращенные строковые коды форматов могут показаться запутанными, но они основаны на простых принципах. Первый (необязательный) символ — < или >, означает «число с прямым порядком байтов» или «число с обратным порядком байтов» соответственно и задает порядок значащих битов. Следующий символ задает тип данных: символы, байтовый тип, целые числа, числа с плавающей точкой и т. д. (табл. 2.6). Последний символ или символы отражают размер объекта в байтах.

Таблица 2.6. Типы данных библиотеки NumPy

Символ	Описание	Пример
'b'	Байтовый тип	np.dtype('b')
'i'	Знаковое целое число	np.dtype('i4') == np.int32
'u'	Беззнаковое целое число	np.dtype('u1') == np.uint8
'f'	Число с плавающей точкой	np.dtype('f8') == np.int64
'c'	Комплексное число с плавающей точкой	np.dtype('c16') == np.complex128
'S', 'a'	Строка	np.dtype('S5')
'U'	Строка в кодировке Unicode	np.dtype('U') == np.str_
'V'	Неформатированные данные (тип void)	np.dtype('V') == np.void

Более продвинутые типы данных

Можно описывать и еще более продвинутые типы данных. Например, можно создать тип, в котором каждый элемент содержит массив или матрицу значений. В следующем примере мы создаем тип данных, содержащий компонент `mat`, состоящий из матрицы 3 × 3 значения с плавающей точкой:

```
In[14]: tp = np.dtype([('id', 'i8'), ('mat', 'f8', (3, 3))])
        X = np.zeros(1, dtype=tp)
```

```
        print(X[0])
        print(X['mat'][0])
```

```
(0, [[0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]])
[[ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]]
```

Теперь каждый элемент массива `X` состоит из целого числа `id` и матрицы 3×3. Почему такой массив может оказаться предпочтительнее, чем простой многомерный массив или, возможно, словарь языка Python? Дело в том, что `dtype` библиотеки NumPy напрямую соответствует описанию структуры из языка C, так что можно обращаться к содержащему этот массив буферу памяти непосредственно из соответствующим образом написанной программы на языке C. Если вам понадобится написать на языке Python интерфейс к уже существующей библиотеке на языке C или Fortran, которая работает со структурированными данными, вероятно, структурированные массивы будут вам весьма полезны!

Массивы записей: структурированные массивы с дополнительными возможностями

Библиотека NumPy предоставляет класс `np.recarray`, практически идентичный только что описанным структурированным массивам, но с одной дополнительной возможностью: доступ к полям можно осуществлять как к атрибутам, а не только как к ключам словаря. Как вы помните, ранее мы обращались к значениям возраста путем написания следующей строки кода:

```
In[15]: data['age']
```

```
Out[15]: array([25, 45, 37, 19], dtype=int32)
```

Если же представить наши данные как массив записей, то можно обращаться к этим данным с помощью чуть более короткого синтаксиса:

```
In[16]: data_rec = data.view(np.recarray)
        data_rec.age
```

```
Out[16]: array([25, 45, 37, 19], dtype=int32)
```

Недостаток такого подхода состоит в том, что при доступе к полям массивов записей неизбежны дополнительные накладные расходы, даже при использовании того же синтаксиса. Это видно из следующего:

```
In[17]: %timeit data['age']
        %timeit data_rec['age']
        %timeit data_rec.age
```

```
1000000 loops, best of 3: 241 ns per loop
100000 loops, best of 3: 4.61 µs per loop
100000 loops, best of 3: 7.27 µs per loop
```

Имеет ли смысл жертвовать дополнительным временем ради более удобного синтаксиса — зависит от вашего приложения.

Вперед, к Pandas

Этот раздел по структурированным массивам и массивам записей не случайно стоит в конце этой главы, поскольку он удачно подводит нас к теме следующего рассматриваемого пакета — библиотеки Pandas. В определенных случаях не помешает знать о существовании обсуждавшихся здесь структурированных массивов, особенно если вам нужно, чтобы массивы библиотеки NumPy соответствовали двоичным форматам данных в C, Fortran или другом языке программирования. Для регулярной работы со структурированными данными намного удобнее использовать пакет Pandas, который мы подробно рассмотрим в следующей главе.

3 Манипуляции над данными с помощью пакета Pandas

В предыдущей главе мы рассмотрели библиотеку NumPy и ее объект `ndarray`, обеспечивающий эффективное хранение плотных массивов и манипуляции над ними в Python. В этой главе мы, основываясь на этих знаниях, детально ознакомимся со структурами данных библиотеки Pandas.

Pandas — более новый пакет, надстройка над библиотекой NumPy, обеспечивающий эффективную реализацию класса `DataFrame`. Объекты `DataFrame` — многомерные массивы с метками для строк и столбцов, а также зачастую с неоднородным типом данных и/или пропущенными данными. Помимо удобного интерфейса для хранения маркированных данных, библиотека Pandas реализует множество операций для работы с данными хорошо знакомых пользователям фреймворков баз данных и электронных таблиц.

Структура данных `ndarray` библиотеки NumPy предоставляет все необходимые возможности для работы с хорошо упорядоченными данными в задачах численных вычислений. Для этой цели библиотека NumPy отлично подходит, однако имеет свои ограничения, которые становятся заметными, чуть только нам потребуется немного больше гибкости (маркирование данных, работа с пропущенными данными и т. д.). Эти ограничения проявляются также при попытках выполнения операций, неподходящих для поэлементного транслирования (группировки, создание сводных таблиц и т. д.). Такие операции являются важной частью анализа данных с меньшей степенью структурированности, содержащихся во многих формах окружающего мира. Библиотека Pandas, особенно ее объекты `Series` и `DataFrame`, основана на структурах массивов библиотеки NumPy и обеспечивает эффективную работу над подобными задачами «очистки данных».

В этой главе мы сосредоточимся на стандартных приемах использования объектов `Series`, `DataFrame` и связанных с ними структур. По мере возможности мы будем применять взятые из реальных наборов данных примеры, но они не являются нашей целью.

Установка и использование библиотеки Pandas

Для установки пакета Pandas необходимо наличие в вашей системе пакета NumPy, а если вы выполняете сборку библиотеки из исходного кода, то и соответствующих утилит для компиляции исходных кодов на языках C и Cython, из которых состоит Pandas. Подробные инструкции по установке можно найти в документации пакета Pandas (http://pandas.pydata.org/). Если же вы последовали совету из предисловия и воспользовались стеком Anaconda, то пакет Pandas у вас уже имеется.

После установки пакета Pandas можно импортировать его и проверить версию:

```
In[1]: import pandas
       pandas.__version__

Out[1]: '0.18.1'
```

Аналогично тому, как мы импортировали пакет NumPy под псевдонимом `np`, пакет Pandas импортируем под псевдонимом `pd`:

```
In[2]: import pandas as pd
```

Мы будем использовать эти условные обозначения для импорта далее в книге.

Напоминание о встроенной документации

Оболочка IPython предоставляет возможность быстро просматривать содержимое пакетов (с помощью клавиши Tab), а также документацию по различным функциям (используя символ `?`). Загляните в раздел «Справка и документация в оболочке Python» главы 1, если вам нужно освежить в памяти эти возможности.

Для отображения всего содержимого пространства имен `numpy` можете ввести следующую команду:

```
In [3]: np.<TAB>
```

Для отображения встроенной документации пакета NumPy используйте команду:

```
In [4]: np?
```

Более подробную документацию, а также руководства и другие источники информации можно найти на сайте http://pandas.pydata.org.

Знакомство с объектами библиотеки Pandas

На самом примитивном уровне объекты библиотеки Pandas можно считать расширенной версией структурированных массивов библиотеки NumPy, в которых строки и столбцы идентифицируются метками, а не простыми числовыми индексами. Библиотека Pandas предоставляет множество полезных утилит, методов и функциональности в дополнение к базовым структурам данных, но все последующее изложение потребует понимания этих базовых структур. Позвольте познакомить вас с тремя фундаментальными структурами данных библиотеки Pandas: классами `Series`, `DataFrame` и `Index`.

Начнем наш сеанс программирования с обычных импортов библиотек NumPy и Pandas:

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

Объект Series библиотеки Pandas

Объект `Series` библиотеки Pandas — одномерный массив индексированных данных. Его можно создать из списка или массива следующим образом:

```
In[2]: data = pd.Series([0.25, 0.5, 0.75, 1.0])
       data

Out[2]: 0    0.25
        1    0.50
        2    0.75
        3    1.00
        dtype: float64
```

Как мы видели из предыдущего результата, объект `Series` служит адаптером как для последовательности значений, так и последовательности индексов, к которым можно получить доступ посредством атрибутов `values` и `index`. Атрибут `values` представляет собой уже знакомый нам массив NumPy:

```
In[3]: data.values

Out[3]: array([ 0.25,  0.5 ,  0.75,  1.  ])
```

`index` — массивоподобный объект типа `pd.Index`, который мы рассмотрим подробнее далее:

```
In[4]: data.index

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

[1] См. параметры «начало» (start), «конец» (stop) и «шаг» (step), рассмотренные в главе 2.

Аналогично массивам библиотеки NumPy, к данным можно обращаться по соответствующему им индексу посредством нотации с использованием квадратных скобок языка Python:

```
In[5]: data[1]

Out[5]: 0.5

In[6]: data[1:3]

Out[6]: 1    0.50
        2    0.75
        dtype: float64
```

Однако объект `Series` библиотеки Pandas намного универсальнее и гибче, чем эмулируемый им одномерный массив библиотеки NumPy.

Объект Series как обобщенный массив NumPy

Может показаться, что объект `Series` и одномерный массив библиотеки NumPy взаимозаменяемы. Основное различие между ними — индекс. В то время как индекс массива NumPy, используемый для доступа к значениям, — целочисленный и *описывается неявно*, индекс объекта `Series` библиотеки Pandas *описывается явно* и связывается со значениями.

Явное описание индекса расширяет возможности объекта `Series`. Такой индекс не должен быть целым числом, а может состоять из значений любого нужного типа. Например, при желании мы можем использовать в качестве индекса строковые значения:

```
In[7]: data = pd.Series([0.25, 0.5, 0.75, 1.0],
                        index=['a', 'b', 'c', 'd'])
       data

Out[7]: a    0.25
        b    0.50
        c    0.75
        d    1.00
        dtype: float64
```

При этом доступ к элементам работает обычным образом:

```
In[8]: data['b']

Out[8]: 0.5
```

Можно применять даже индексы, состоящие из несмежных или непоследовательных значений:

```
In[9]: data = pd.Series([0.25, 0.5, 0.75, 1.0],
                        index=[2, 5, 3, 7])
       data

Out[9]: 2    0.25
        5    0.50
        3    0.75
        7    1.00
        dtype: float64

In[10]: data[5]

Out[10]: 0.5
```

Объект Series как специализированный словарь

Объект `Series` библиотеки Pandas можно рассматривать как специализированную разновидность словаря языка Python. Словарь — структура, задающая соответствие произвольных ключей набору произвольных значений, а объект `Series` — структура, задающая соответствие типизированных ключей набору типизированных значений. Типизация важна: точно так же, как соответствующий типу специализированный код для массива библиотеки NumPy при выполнении определенных операций делает его эффективнее, чем стандартный список Python, информация о типе в объекте `Series` библиотеки Pandas делает его намного более эффективным для определенных операций, чем словари Python.

Можно сделать аналогию «объект `Series` — словарь» еще более наглядной, сконструировав объект `Series` непосредственно из словаря Python:

```
In[11]: population_dict = {'California': 38332521,
                           'Texas': 26448193,
                           'New York': 19651127,
                           'Florida': 19552860,
                           'Illinois': 12882135}
        population = pd.Series(population_dict)
        population

Out[11]: California    38332521
         Florida       19552860
         Illinois      12882135
         New York      19651127
         Texas         26448193
         dtype: int64
```

По умолчанию при этом будет создан объект `Series` с полученным из отсортированных ключей индексом. Следовательно, для него возможен обычный доступ к элементам, такой же, как для словаря:

```
In[12]: population['California']

Out[12]: 383 32521
```

Однако, в отличие от словаря, объект `Series` поддерживает характерные для массивов операции, такие как срезы:

```
In[13]: population['California':'Illinois']

Out[13]: California    38332521
         Florida       19552860
         Illinois      12882135
         dtype: int64
```

Мы рассмотрим некоторые нюансы индексации и срезов в библиотеке Pandas в разделе «Индексация и выборка данных» этой главы.

Создание объектов Series

Мы уже изучили несколько способов создания объектов `Series` библиотеки Pandas с нуля. Все они представляют собой различные варианты следующего синтаксиса:

```
>>> pd.Series(data, index=index)
```

где `index` — необязательный аргумент, а `data` может быть одной из множества сущностей.

Например, аргумент `data` может быть списком или массивом NumPy. В этом случае `index` по умолчанию будет целочисленной последовательностью:

```
In[14]: pd.Series([2, 4, 6])

Out[14]: 0    2
         1    4
         2    6
         dtype: int64
```

Аргумент `data` может быть скалярным значением, которое будет повторено нужное количество раз для заполнения заданного индекса:

```
In[15]: pd.Series(5, index=[100, 200, 300])

Out[15]: 100    5
         200    5
         300    5
         dtype: int64
```

Аргумент `data` может быть словарем, в котором `index` по умолчанию является отсортированными ключами этого словаря:

```
In[16]: pd.Series({2:'a', 1:'b', 3:'c'})

Out[16]: 1    b
         2    a
         3    c
         dtype: object
```

В каждом случае индекс можно указать вручную, если необходимо получить другой результат:

```
In[17]: pd.Series({2:'a', 1:'b', 3:'c'}, index=[3, 2])

Out[17]: 3    c
         2    a
         dtype: object
```

Обратите внимание, что объект `Series` заполняется только заданными явным образом ключами.

Объект DataFrame библиотеки Pandas

Следующая базовая структура библиотеки Pandas — объект `DataFrame`. Как и объект `Series`, обсуждавшийся в предыдущем разделе, объект `DataFrame` можно рассматривать или как обобщение массива NumPy, или как специализированную версию словаря Python. Изучим оба варианта.

DataFrame как обобщенный массив NumPy

Если объект `Series` — аналог одномерного массива с гибкими индексами, объект `DataFrame` — аналог двумерного массива с гибкими индексами строк и гибкими именами столбцов. Аналогично тому, что двумерный массив можно рассматривать как упорядоченную последовательность выровненных столбцов, объект `DataFrame` можно рассматривать как упорядоченную последовательность выровненных объектов `Series`. Под «выровненными» имеется в виду то, что они используют один и тот же индекс.

Чтобы продемонстрировать это, сначала создадим новый объект `Series`, содержащий площадь каждого из пяти упомянутых в предыдущем разделе штатов:

```
In[18]:
area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
             'Florida': 170312, 'Illinois': 149995}
area = pd.Series(area_dict)
area

Out[18]: California    423967
         Florida       170312
         Illinois      149995
         New York      141297
         Texas         695662
         dtype: int64
```

Воспользовавшись объектом `population` класса `Series`, сконструируем на основе словаря единый двумерный объект, содержащий всю эту информацию:

```
In[19]: states = pd.DataFrame({'population': population,
                               'area': area})
        states

Out[19]:             area     population
         California  423967   38332521
         Florida     170312   19552860
         Illinois    149995   12882135
         New York    141297   19651127
         Texas       695662   26448193
```

Аналогично объекту `Series` у объекта `DataFrame` имеется атрибут `index`, обеспечивающий доступ к меткам индекса:

```
In[20]: states.index

Out[20]:
Index(['California', 'Florida', 'Illinois', 'New York', 'Texas'],
      dtype='object')
```

Помимо этого, у объекта `DataFrame` есть атрибут `columns`, представляющий собой содержащий метки столбцов объект `Index`:

```
In[21]: states.columns

Out[21]: Index(['area', 'population'], dtype='object')
```

Таким образом, объект `DataFrame` можно рассматривать как обобщение двумерного массива NumPy, где как у строк, так и у столбцов есть обобщенные индексы для доступа к данным.

Объект DataFrame как специализированный словарь

`DataFrame` можно рассматривать как специализированный словарь. Если словарь задает соответствие ключей значениям, то `DataFrame` задает соответствие имени столбца объекту `Series` с данными этого столбца. Например, запрос данных по атрибуту `'area'` приведет к тому, что будет возвращен объект `Series`, содержащий уже виденные нами ранее площади штатов:

```
In[22]: states['area']

Out[22]: California    423967
         Florida       170312
         Illinois      149995
         New York      141297
         Texas         695662
         Name: area, dtype: int64
```

Обратите внимание на возможный источник путаницы: в двумерном массиве NumPy `data[0]` возвращает первую *строку*. По этой причине объекты `DataFrame`

лучше рассматривать как обобщенные словари, а не обобщенные массивы, хотя обе точки зрения имеют право на жизнь. Мы изучим более гибкие средства индексации объектов `DataFrame` в разделе «Индексация и выборка данных» этой главы.

Создание объектов DataFrame

Существует множество способов создания объектов `DataFrame` библиотеки Pandas. Вот несколько примеров.

Из одного объекта Series. Объект `DataFrame` — набор объектов `Series`. `DataFrame`, состоящий из одного столбца, можно создать на основе одного объекта `Series`:

```
In[23]: pd.DataFrame(population, columns=['population'])

Out[23]:              population
         California    38332521
         Florida       19552860
         Illinois      12882135
         New York      19651127
         Texas         26448193
```

Из списка словарей. Любой список словарей можно преобразовать в объект `DataFrame`. Мы воспользуемся простым списковым включением для создания данных:

```
In[24]: data = [{'a': i, 'b': 2 * i}
                for i in range(3)]
        pd.DataFrame(data)

Out[24]:    a  b
         0  0  0
         1  1  2
         2  2  4
```

Даже если некоторые ключи в словаре отсутствуют, библиотека Pandas просто заполнит их значениями `NaN` (то есть Not a number — «не является числом»):

```
In[25]: pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}])

Out[25]:    a    b  c
         0  1.0  2  NaN
         1  NaN  3  4.0
```

Из словаря объектов Series. Объект `DataFrame` также можно создать на основе словаря объектов `Series`:

```
In[26]: pd.DataFrame({'population': population,
                      'area': area})

Out[26]:             area     population
         California  423967   38332521
```

```
Florida     170312   19552860
Illinois    149995   12882135
New York    141297   19651127
Texas       695662   26448193
```

Из двумерного массива NumPy. Если у нас есть двумерный массив данных, мы можем создать объект `DataFrame` с любыми заданными именами столбцов и индексов. Для каждого из пропущенных значений будет использоваться целочисленный индекс:

```
In[27]: pd.DataFrame(np.random.rand(3, 2),
                     columns=['foo', 'bar'],
                     index=['a', 'b', 'c'])

Out[27]:    foo       bar
         a  0.865257  0.213169
         b  0.442759  0.108267
         c  0.047110  0.905718
```

Из структурированного массива NumPy. Мы рассматривали структурированные массивы в разделе «Структурированные данные: структурированные массивы библиотеки NumPy» главы 2. Объект `DataFrame` библиотеки Pandas ведет себя во многом аналогично структурированному массиву и может быть создан непосредственно из него:

```
In[28]: A = np.zeros(3, dtype=[('A', 'i8'), ('B', 'f8')])
        A
Out[28]: array([(0, 0.0), (0, 0.0), (0, 0.0)],
               dtype=[('A', '<i8'), ('B', '<f8')])

In[29]: pd.DataFrame(A)

Out[29]:    A  B
         0  0  0.0
         1  0  0.0
         2  0  0.0
```

Объект Index библиотеки Pandas

Как объект `Series`, так и объект `DataFrame` содержат явный *индекс*, обеспечивающий возможность ссылаться на данные и модифицировать их. Объект `Index` можно рассматривать или как *неизменяемый массив* (immutable array), или как *упорядоченное множество* (ordered set) (формально мультимножество, так как объекты `Index` могут содержать повторяющиеся значения). Из этих способов его представления следуют некоторые интересные возможности операций над объектами `Index`. В качестве простого примера создадим `Index` из списка целых чисел:

```
In[30]: ind = pd.Index([2, 3, 5, 7, 11])
        ind

Out[30]: Int64Index([2, 3, 5, 7, 11], dtype='int64')
```

Объект Index как неизменяемый массив

Объект `Index` во многом ведет себя аналогично массиву. Например, для извлечения из него значений или срезов можно использовать стандартную нотацию индексации языка Python:

```
In[31]: ind[1]

Out[31]: 3

In[32]: ind[::2]

Out[32]: Int64Index([2, 5, 11], dtype='int64')
```

У объектов `Index` есть много атрибутов, знакомых нам по массивам NumPy:

```
In[33]: print(ind.size, ind.shape, ind.ndim, ind.dtype)


5 (5,) 1 int64
```

Одно из различий между объектами `Index` и массивами NumPy — неизменяемость индексов, то есть их нельзя модифицировать стандартными средствами:

```
In[34]: ind[1] = 0

---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-34-40e631c82e8a> in <module>()
----> 1 ind[1] = 0

/Users/jakevdp/anaconda/lib/python3.5/site-packages/pandas/indexes/base.py ...
   1243
   1244     def __setitem__(self, key, value):
-> 1245         raise TypeError("Index does not support mutable operations")
   1246
   1247     def __getitem__(self, key):

TypeError: Index does not support mutable operations
```

Неизменяемость делает безопаснее совместное использование индексов несколькими объектами `DataFrame` и массивами, исключая возможность побочных эффектов в виде случайной модификации индекса по неосторожности.

Index как упорядоченное множество

Объекты библиотеки Pandas спроектированы с прицелом на упрощение таких операций, как соединения наборов данных, зависящие от многих аспектов арифметики множеств. Объект `Index` следует большинству соглашений, используемых встроенной структурой данных `set` языка Python, так что объединения, пересечения, разности и другие операции над множествами можно выполнять привычным образом:

```
In[35]: indA = pd.Index([1, 3, 5, 7, 9])
        indB = pd.Index([2, 3, 5, 7, 11])

In[36]: indA & indB  # пересечение

Out[36]: Int64Index([3, 5, 7], dtype='int64')

In[37]: indA | indB  # объединение

Out[37]: Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')

In[38]: indA ^ indB  # симметричная разность

Out[38]: Int64Index([1, 2, 9, 11], dtype='int64')
```

Эти операции можно выполнять также методами объектов, например `indA.intersection(indB)`.

Индексация и выборка данных

В главе 2 мы подробно рассмотрели методы и инструменты доступа, задания и изменения значений в массивах библиотеки NumPy: индексацию (`arr[2, 1]`), срезы массивов (`arr[:,1:5]`), маскирование (`arr[arr > 0]`), «прихотливую» индексацию (`arr[0,[1,5]]`), а также их комбинации (`arr[:,[1,5]]`). Здесь мы изучим аналогичные средства доступа и изменения значений в объектах `Series` и `DataFrame` библиотеки Pandas. Если вы использовали паттерны библиотеки NumPy, то соответствующие паттерны библиотеки Pandas будут для вас привычны.

Начнем с простого случая одномерного объекта `Series`, после чего перейдем к более сложному двумерному объекту `DataFrame`.

Выборка данных из объекта Series

Объект `Series` во многом ведет себя подобно одномерному массиву библиотеки NumPy и стандартному словарю языка Python. Это поможет нам лучше понимать паттерны индексации и выборки данных из этих массивов.

Объект Series как словарь

Объект `Series` задает соответствие набора ключей набору значений аналогично словарю:

```
In[1]: import pandas as pd
       data = pd.Series([0.25, 0.5, 0.75, 1.0],
                        index=['a', 'b', 'c', 'd'])
       data

Out[1]: a    0.25
        b    0.50
        c    0.75
        d    1.00
        dtype: float64

In[2]: data['b']

Out[2]: 0.5
```

Для просмотра ключей/индексов и значений выражения можно также использовать методы языка Python, аналогичные таковым для словарей:

```
In[3]: 'a' in data

Out[3]: True

In[4]: data.keys()

Out[4]: Index(['a', 'b', 'c', 'd'], dtype='object')

In[5]: list(data.items())

Out[5]: [('a', 0.25), ('b', 0.5), ('c', 0.75), ('d', 1.0)]
```

Объекты `Series` можно модифицировать с помощью синтаксиса, подобного синтаксису для словарей. Аналогично расширению словаря путем присваивания значения для нового ключа можно расширить объект `Series`, присвоив значение для нового значения индекса:

```
In[6]: data['e'] = 1.25
       data

Out[6]: a    0.25
        b    0.50
        c    0.75
        d    1.00
        e    1.25
        dtype: float64
```

Такая легкая изменяемость объектов — удобная возможность: библиотека Pandas сама, незаметно для нас, принимает решения о размещении в памяти и необходимости

копирования данных. Пользователю, как правило, не приходится заботиться о подобных вопросах.

Объект Series как одномерный массив

Объект `Series`, основываясь на интерфейсе, напоминающем словарь, предоставляет возможность выборки элементов с помощью тех же базовых механизмов, что и для массивов NumPy, то есть *срезов*, *маскирования* и *«прихотливой» индексации*. Приведу несколько примеров:

```
In[7]: # срез посредством явного индекса
       data['a':'c']

Out[7]: a    0.25
        b    0.50
        c    0.75
        dtype: float64

In[8]: # срез посредством неявного целочисленного индекса
       data[0:2]

Out[8]: a    0.25
        b    0.50
        dtype: float64

In[9]: # маскирование
       data[(data > 0.3) & (data < 0.8)]

Out[9]: b    0.50
        c    0.75
        dtype: float64

In[10]: # «прихотливая» индексация
        data[['a', 'e']]

Out[10]: a    0.25
         e    1.25
         dtype: float64
```

Наибольшие затруднения среди них могут вызвать срезы. Обратите внимание, что при выполнении среза с помощью явного индекса (`data['a':'c']`) значение, соответствующее последнему индексу, *включается* в срез, а при срезе неявным индексом (`data[0:2]`) — *не включается*.

Индексаторы: loc, iloc и ix

Подобные обозначения для срезов и индексации могут привести к путанице. Например, при наличии у объекта `Series` явного целочисленного индекса опера-

ция индексации (`data[1]`) будет использовать явные индексы, а операция среза (`data[1:3]`) — неявный индекс в стиле языка Python.

```
In[11]: data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
        data

Out[11]: 1    a
         3    b
         5    c
         dtype: object

In[12]: # Использование явного индекса при индексации
        data[1]

Out[12]: 'a'

In[13]: # Использование неявного индекса при срезе
        data[1:3]

Out[13]: 3    b
         5    c
         dtype: object
```

Из-за этой потенциальной путаницы в случае целочисленных индексов в библиотеке Pandas предусмотрены специальные атрибуты-*индексаторы*, позволяющие явным образом применять определенные схемы индексации. Они являются не функциональными методами, а именно атрибутами, предоставляющими для данных из объекта `Series` определенный интерфейс для выполнения срезов.

Во-первых, атрибут `loc` позволяет выполнить индексацию и срезы с использованием явного индекса:

```
In[14]: data.loc[1]

Out[14]: 'a'

In[15]: data.loc[1:3]

Out[15]: 1    a
         3    b
         dtype: object
```

Атрибут `iloc` дает возможность выполнить индексацию и срезы, применяя неявный индекс в стиле языка Python:

```
In[16]: data.iloc[1]

Out[16]: 'b'

In[17]: data.iloc[1:3]

Out[17]: 3    b
```

```
5    c
dtype: object
```

Третий атрибут-индексатор `ix` представляет собой гибрид первых двух и для объектов `Series` эквивалентен обычной индексации с помощью `[]`. Назначение индексатора `ix` станет понятнее в контексте объектов `DataFrame`, которые мы рассмотрим далее.

Один из руководящих принципов написания кода на языке Python — «лучше явно, чем неявно». То, что атрибуты `loc` и `iloc` по своей природе явные, делает их очень удобными для обеспечения «чистоты» и удобочитаемости кода. Я рекомендую использовать оба, особенно в случае целочисленных индексов, чтобы сделать код более простым для чтения и понимания и избежать случайных малозаметных ошибок при обозначении индексации и срезов.

Выборка данных из объекта DataFrame

Объект `DataFrame` во многом ведет себя аналогично двумерному или структурированному массиву, а также словарю объектов `Series` с общим индексом. Эти аналогии следует иметь в виду во время изучения способов выборки данных из объекта.

Объект DataFrame как словарь

Первая аналогия, которую мы будем обсуждать, — объект `DataFrame` как словарь схожих между собой объектов `Series`. Вернемся к примеру про площадь и численность населения штатов:

```
In[18]: area = pd.Series({'California': 423967, 'Texas': 695662,
                          'New York': 141297, 'Florida': 170312,
                          'Illinois': 149995})
        pop = pd.Series({'California': 38332521, 'Texas': 26448193,
                         'New York': 19651127, 'Florida': 19552860,
                         'Illinois': 12882135})
        data = pd.DataFrame({'area':area, 'pop':pop})
        data

Out[18]:             area    pop
         California  423967  38332521
         Florida     170312  19552860
         Illinois    149995  12882135
         New York    141297  19651127
         Texas       695662  26448193
```

К отдельным объектам `Series`, составляющим столбцы объекта `DataFrame`, можно обращаться посредством такой же индексации, как и для словарей, по имени столбца:

```
In[19]: data['area']

Out[19]: California    423967
         Florida       170312
         Illinois      149995
         New York      141297
         Texas         695662
         Name: area, dtype: int64
```

Можно обращаться к данным и с помощью атрибутов, используя в их качестве строковые имена столбцов:

```
In[20]: data.area

Out[20]: California    423967
         Florida       170312
         Illinois      149995
         New York      141297
         Texas         695662
         Name: area, dtype: int64
```

При доступе по имени атрибута-столбца фактически происходит обращение к тому же объекту, что и при словарном варианте доступа:

```
In[21]: data.area is data['area']

Out[21]: True
```

Хотя это и удобное сокращенное написание, не забывайте, что оно работает не всегда! Например, если имена столбцов — не строки или имена столбцов конфликтуют с методами объекта `DataFrame`, доступ по именам атрибутов невозможен. Например, у объекта `DataFrame` есть метод `pop()`, так что выражение `data.pop` будет обозначать его, а не столбец `"pop"`:

```
In[22]: data.pop is data['pop']

Out[22]: False
```

Не поддавайтесь искушению присваивать значения столбцов посредством атрибутов. Лучше использовать выражение `data['pop'] = z` вместо `data.pop = z`.

Как и в случае с обсуждавшимися ранее объектами `Series`, такой «словарный» синтаксис можно применять для модификации объекта, например добавления еще одного столбца:

```
In[23]: data['density'] = data['pop'] / data['area']
        data

Out[23]:             area    pop       density
         California  423967  38332521   90.413926
         Florida     170312  19552860  114.806121
```

```
Illinois    149995  12882135   85.883763
New York    141297  19651127  139.076746
Texas       695662  26448193   38.018740
```

Приведенный пример демонстрирует простоту синтаксиса поэлементных операций над объектами `Series`. Этот вопрос мы изучим подробнее в разделе «Операции над данными в библиотеке Pandas» текущей главы.

Объект DataFrame как двумерный массив

Объект `DataFrame` можно рассматривать как двумерный массив с расширенными возможностями. Взглянем на исходный массив данных с помощью атрибута `values`:

```
In[24]: data.values

Out[24]: array([[  4.23967000e+05,   3.83325210e+07,   9.04139261e+01],
                [  1.70312000e+05,   1.95528600e+07,   1.14806121e+02],
                [  1.49995000e+05,   1.28821350e+07,   8.58837628e+01],
                [  1.41297000e+05,   1.96511270e+07,   1.39076746e+02],
                [  6.95662000e+05,   2.64481930e+07,   3.80187404e+01]])
```

Мы можем выполнить множество привычных для массивов действий над объектом `DataFrame`. Например, транспонировать весь `DataFrame`, поменяв местами строки и столбцы:

```
In[25]: data.T

Out[25]:
         California    Florida       Illinois      New York      Texas
area     4.239670e+05  1.703120e+05  1.499950e+05  1.412970e+05  6.956620e+05
pop      3.833252e+07  1.955286e+07  1.288214e+07  1.965113e+07  2.644819e+07
density  9.041393e+01  1.148061e+02  8.588376e+01  1.390767e+02  3.801874e+01
```

Однако, когда речь заходит об индексации объектов `DataFrame`, становится ясно, что словарная индексация мешает нам рассматривать их просто как массивы NumPy. В частности, указание отдельного индекса для массива означает доступ к строке:

```
In[26]: data.values[0]

Out[26]: array([  4.23967000e+05,   3.83325210e+07,   9.04139261e+01])
```

а указание отдельного «индекса» для объекта `DataFrame` — доступ к столбцу:

```
In[27]: data['area']

Out[27]: California    423967
         Florida       170312
```

```
Illinois      149995
New York      141297
Texas         695662
Name: area, dtype: int64
```

Таким образом, нам необходим еще один тип синтаксиса для индексации, аналогичной по стилю индексации массивов. Библиотека Pandas применяет упомянутые ранее индексаторы `loc`, `iloc` и `ix`. С помощью индексатора `iloc` можно индексировать исходный массив, как будто это простой массив NumPy (используя неявный синтаксис языка Python), но с сохранением в результирующих данных меток объекта `DataFrame` для индекса и столбцов:

```
In[28]: data.iloc[:3, :2]

Out[28]:             area    pop
         California  423967  38332521
         Florida     170312  19552860
         Illinois    149995  12882135

In[29]: data.loc[:'Illinois', :'pop']

Out[29]:             area    pop
         California  423967  38332521
         Florida     170312  19552860
         Illinois    149995  12882135
```

Индексатор `ix` позволяет комбинировать эти два подхода:

```
In[30]: data.ix[:3, :'pop']

Out[30]:             area    pop
         California  423967  38332521
         Florida     170312  19552860
         Illinois    149995  12882135
```

Не забывайте, что в случае целочисленных индексов индексатор `ix` может быть источником тех же проблем, что и при целочисленной индексации объектов `Series`.

В этих индексаторах можно использовать все уже знакомые вам паттерны доступа к данным в стиле библиотеки NumPy. Например, в индексаторе `loc` можно сочетать маскирование и «прихотливую» индексацию:

```
In[31]: data.loc[data.density > 100, ['pop', 'density']]

Out[31]:           pop       density
         Florida   19552860  114.806121
         New York  19651127  139.076746
```

Любой такой синтаксис индексации можно применять для задания или изменения значений. Это выполняется обычным, уже привычным вам по работе с библиотекой NumPy, способом:

```
In[32]: data.iloc[0, 2] = 90
        data

Out[32]:             area    pop       density
         California  423967  38332521   90.000000
         Florida     170312  19552860  114.806121
         Illinois    149995  12882135   85.883763
         New York    141297  19651127  139.076746
         Texas       695662  26448193   38.018740
```

Чтобы достичь уверенности при манипуляции данными с помощью библиотеки Pandas, рекомендую потратить немного времени на эксперименты над простым объектом `DataFrame` и пробы типов индексации, срезов, маскирования и «прихотливой» индексации.

Дополнительный синтаксис для индексации

Существует еще несколько вариантов синтаксиса для индексации, казалось бы, плохо согласующихся с обсуждавшимся ранее, но очень удобных на практике. Во-первых, если *индексация* относится к столбцам, *срезы* относятся к строкам:

```
In[33]: data['Florida':'Illinois']

Out[33]:           area    pop       density
         Florida   170312  19552860  114.806121
         Illinois  149995  12882135   85.883763
```

При подобных срезах можно также ссылаться на строки по номеру, а не по индексу:

```
In[34]: data[1:3]

Out[34]:           area    pop       density
         Florida   170312  19552860  114.806121
         Illinois  149995  12882135   85.883763
```

Непосредственные операции маскирования также интерпретируются построчно, а не по столбцам:

```
In[35]: data[data.density > 100]

Out[35]:           area    pop       density
         Florida   170312  19552860  114.806121
         New York  141297  19651127  139.076746
```

Эти два варианта обозначений синтаксически подобны таковым для массивов библиотеки NumPy, и они, возможно, хоть и не вполне вписываются в шаблоны синтаксиса библиотеки Pandas, но весьма удобны на практике.

Операции над данными в библиотеке Pandas

Одна из важнейших составляющих библиотеки NumPy — способность выполнять быстрые поэлементные операции — как простейшие арифметические (сложение, вычитание, умножение и т. д.), так и более сложные (тригонометрические, показательные и логарифмические функции и т. п.). Библиотека Pandas наследует от NumPy немалую часть этой функциональности, и ключ к ее использованию — универсальные функции, с которыми мы познакомились в разделе «Выполнение вычислений над массивами библиотеки NumPy: универсальные функции» главы 2.

Однако библиотека Pandas включает несколько полезных трюков: для унарных операций, например изменения знака и тригонометрических функций, при использовании ее универсальных функций в выводе будут *сохранены индекс и метки столбцов*, а для бинарных операций, например сложения и умножения, библиотека Pandas будет автоматически *выравнивать индексы* при передаче объектов универсальной функции. Это значит, что сохранение контекста данных и объединение данных из различных источников — две задачи, потенциально чреватые ошибками при работе с исходными массивами библиотеки NumPy, — становятся надежно защищенными от ошибок благодаря библиотеке Pandas. Кроме того, в библиотеке заданы операции между одномерными структурами объектов `Series` и двумерными структурами объектов `DataFrame`.

Универсальные функции: сохранение индекса

В силу того что библиотека Pandas предназначена для работы с библиотекой NumPy, все универсальные функции библиотеки NumPy будут работать с объектами `Series` и `DataFrame` библиотеки Pandas. Начнем с описания простых объектов `Series` и `DataFrame` для демонстрации этого:

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

In[2]: rng = np.random.RandomState(42)
       ser = pd.Series(rng.randint(0, 10, 4))
       ser

Out[2]: 0    6
        1    3
        2    7
        3    4
        dtype: int64

In[3]: df = pd.DataFrame(rng.randint(0, 10, (3, 4)),
                         columns=['A', 'B', 'C', 'D'])
       df
```

```
Out[3]:    A  B  C  D
        0  6  9  2  6
        1  7  4  3  7
        2  7  2  5  4
```

Если применить универсальную функцию NumPy к любому из этих объектов, результатом будет другой объект библиотеки Pandas *с сохранением индексов*:

```
In[4]: np.exp(ser)

Out[4]: 0     403.428793
        1      20.085537
        2    1096.633158
        3      54.598150
        dtype: float64
```

Или в случае немного более сложных вычислений:

```
In[5]: np.sin(df * np.pi / 4)

Out[5]:           A             B         C             D
        0 -1.000000  7.071068e-01  1.000000 -1.000000e+00
        1 -0.707107  1.224647e-16  0.707107 -7.071068e-01
        2 -0.707107  1.000000e+00 -0.707107  1.224647e-16
```

Все описанные в разделе «Выполнение вычислений над массивами библиотеки NumPy: универсальные функции» главы 2 универсальные функции можно использовать аналогично вышеприведенным.

Универсальные функции: выравнивание индексов

При бинарных операциях над двумя объектами `Series` или `DataFrame` библиотека Pandas будет выравнивать индексы в процессе выполнения операции. Это очень удобно при работе с неполными данными.

Выравнивание индексов в объектах Series

Допустим, мы объединили два различных источника данных, чтобы найти три штата США с наибольшей *площадью* и три штата США с наибольшим количеством *населения*:

```
In[6]: area = pd.Series({'Alaska': 1723337, 'Texas': 695662,
                         'California': 423967}, name='area')
       population = pd.Series({'California': 38332521, 'Texas': 26448193,
                               'New York': 19651127}, name='population')
```

Посмотрим, что получится, если разделить второй результат на первый для вычисления плотности населения:

```
In[7]: population / area

Out[7]: Alaska              NaN
        California    90.413926
        New York            NaN
        Texas         38.018740
        dtype: float64
```

Получившийся в итоге массив содержит *объединение* индексов двух исходных массивов, которое можно определить посредством стандартной арифметики множеств языка Python для этих индексов:

```
In[8]: area.index | population.index

Out[8]: Index(['Alaska', 'California', 'New York', 'Texas'], dtype='object')
```

Ни один из относящихся к ним обоим элементов не содержит значения `NaN` («нечисловое значение»), с помощью которого библиотека Pandas отмечает пропущенные данные (см. дальнейшее обсуждение вопроса отсутствующих данных в разделе «Обработка отсутствующих данных» этой главы). Аналогичным образом реализовано сопоставление индексов для всех встроенных арифметических выражений языка Python: все отсутствующие значения заполняются по умолчанию значением `NaN`:

```
In[9]: A = pd.Series([2, 4, 6], index=[0, 1, 2])
       B = pd.Series([1, 3, 5], index=[1, 2, 3])
       A + B

Out[9]: 0    NaN
        1    5.0
        2    9.0
        3    NaN
        dtype: float64
```

Если использование значений `NaN` нежелательно, можно заменить заполняющее значение другим, воспользовавшись соответствующими методами объекта вместо операторов. Например, вызов метода `A.add(B)` эквивалентен вызову `A + B`, но предоставляет возможность по желанию задать явным образом значения заполнителей для любых потенциально отсутствующих элементов в объектах `A` или `B`:

```
In[10]: A.add(B, fill_value=0)

Out[10]: 0    2.0
         1    5.0
         2    9.0
         3    5.0
         dtype: float64
```

Выравнивание индексов в объектах DataFrame

При выполнении операций над объектами `DataFrame` происходит аналогичное выравнивание *как* для столбцов, *так* и для индексов:

```
In[11]: A = pd.DataFrame(rng.randint(0, 20, (2, 2)),
                         columns=list('AB'))
        A
Out[11]:    A   B
         0  1  11
         1  5   1

In[12]: B = pd.DataFrame(rng.randint(0, 10, (3, 3)),
                         columns=list('BAC'))
        B

Out[12]:    B  A  C
         0  4  0  9
         1  5  8  0
         2  9  2  6

In[13]: A + B
Out[13]:       A     B   C
         0   1.0  15.0 NaN
         1  13.0   6.0 NaN
         2   NaN   NaN NaN
```

Обратите внимание, что индексы выравниваются правильно независимо от их расположения в двух объектах и индексы в полученном результате отсортированы. Как и в случае объектов `Series`, можно использовать соответствующие арифметические методы объектов и передавать для использования вместо отсутствующих значений любое нужное значение `fill_value`. В следующем примере мы заполним отсутствующие значения средним значением всех элементов объекта `A` (которое вычислим, выстроив сначала значения объекта `A` в один столбец с помощью функции `stack`[1]):

```
In[14]: fill = A.stack().mean()
        A.add(B, fill_value=fill)

Out[14]:       A     B     C
         0   1.0  15.0  13.5
         1  13.0   6.0   4.5
         2   6.5  13.5  10.5
```

---

[1] Результат применения к объекту A функции выглядит следующим образом:
In [45]: A.stack()
Out[45]:
0 A 1
  B 11
1 A 5
  B 1
dtype: int32
См. также подраздел «Мультииндекс как дополнительное измерение» раздела «Мультииндексированный объект Series» текущей главы.

В табл. 3.1 приведен перечень операторов языка Python и эквивалентных им методов объектов библиотеки Pandas.

Таблица 3.1. Соответствие между операторами языка Python и методами библиотеки Pandas

Оператор языка Python	Метод (-ы) библиотеки Pandas
+	add()
–	sub(), subtract()
*	mul(), multiply()
/	truediv(), div(), divide()
//	floordiv()
%	mod()
**	pow()

Универсальные функции: выполнение операции между объектами DataFrame и Series

При выполнении операций между объектами `DataFrame` и `Series` выравнивание столбцов и индексов осуществляется аналогичным образом. Операции между объектами `DataFrame` и `Series` подобны операциям между двумерным и одномерным массивами библиотеки NumPy. Рассмотрим одну из часто встречающихся операций — вычисление разности двумерного массива и одной из его строк:

```
In[15]: A = rng.randint(10, size=(3, 4))
        A

Out[15]: array([[3, 8, 2, 4],
                [2, 6, 4, 8],
                [6, 1, 3, 8]])

In[16]: A - A[0]

Out[16]: array([[ 0,  0,  0,  0],
                [-1, -2,  2,  4],
                [ 3, -7,  1,  4]])
```

В соответствии с правилами транслирования библиотеки NumPy («Операции над массивами. Транслирование» главы 2), вычитание из двумерного массива одной из его строк выполняется построчно.

В библиотеке Pandas вычитание по умолчанию также происходит построчно:

```
In[17]: df = pd.DataFrame(A, columns=list('QRST'))
        df - df.iloc[0]

Out[17]:    Q  R  S  T
         0  0  0  0  0
```

```
        1 -1 -2  2  4
        2  3 -7  1  4
```

Если же вы хотите выполнить эту операцию по столбцам, то можете воспользоваться упомянутыми выше методами объектов, указав ключевое слово `axis`:

```
In[18]: df.subtract(df['R'], axis=0)

Out[18]:    Q  R  S  T
        0 -5  0 -6 -4
        1 -4  0 -2  2
        2  5  0  2  7
```

Обратите внимание, что операции `DataFrame/Series`, аналогично обсуждавшимся ранее операциям, будут автоматически выполнять выравнивание индексов между двумя их элементами:

```
In[19]: halfrow = df.iloc[0, ::2]
        halfrow

Out[19]: Q    3
         S    2
         Name: 0, dtype: int64

In[20]: df - halfrow

Out[20]:      Q   R    S   T
        0  0.0 NaN  0.0 NaN
        1 -1.0 NaN  2.0 NaN
        2  3.0 NaN  1.0 NaN
```

Подобное сохранение и выравнивание индексов и столбцов означает, что операции над данными в библиотеке Pandas всегда сохраняют контекст данных, предотвращая возможные ошибки при работе с неоднородными и/или неправильно/неодинаково выровненными данными в исходных массивах NumPy.

Обработка отсутствующих данных

Реальные данные редко бывают очищенными и однородными. В частности, во многих интересных наборах данных некоторое количество данных отсутствует. Еще более затрудняет работу то, что в различных источниках данных отсутствующие данные могут быть помечены различным образом.

В этом разделе мы обсудим общие соображения, касающиеся отсутствующих данных, обсудим способы представления их библиотекой Pandas и продемонстрируем встроенные инструменты библиотеки Pandas для обработки отсутствующих данных в языке Python. Здесь и далее мы будем называть отсутствующие данные *null*, *NaN* или *NA*-значениями.

Компромиссы при обозначении отсутствующих данных

Разработано несколько схем для обозначения наличия пропущенных данных в таблицах или объектах `DataFrame`. Они основываются на одной из двух стратегий: использование *маски*, отмечающей глобально отсутствующие значения, или выбор специального *значения-индикатора* (sentinel value), обозначающего пропущенное значение.

Маска может быть или совершенно отдельным булевым массивом или может включать выделение одного бита представления данных на локальную индикацию отсутствия значения.

Значение-индикатор может быть особым условным обозначением, подходящим для конкретных данных, например указывать на отсутствующее целое число с помощью значения –9999 или какой-то редкой комбинации битов. Или же оно может быть более глобальным обозначением, например обозначать отсутствующее значение с плавающей точкой с помощью *NaN* — специального значения, включенного в спецификации IEEE для чисел с плавающей точкой.

В каждом из подходов есть свои компромиссы: использование отдельного массива-маски требует выделения памяти под дополнительный булев массив, приводящего к накладным расходам в смысле как оперативной памяти, так и процессорного времени. Значение-индикатор сокращает диапазон доступных для представления допустимых значений и может потребовать выполнения дополнительной (зачастую неоптимизированной) логики при арифметических операциях на CPU и GPU. Общие специальные значения, такие как *NaN*, доступны не для всех типов данных.

Как и в большинстве случаев, где нет универсального оптимального варианта, различные языки программирования и системы используют различные обозначения. Например, язык R применяет зарезервированные комбинации битов для каждого типа данных в качестве значений-индикаторов для отсутствующих данных. А система SciDB использует для индикации состояния *NA* дополнительный байт, присоединяемый к каждой ячейке.

Отсутствующие данные в библиотеке Pandas

Способ обработки отсутствующих данных библиотекой Pandas определяется тем, что она основана на пакете NumPy, в котором отсутствует встроенное понятие *NA*-значений для всех типов данных, кроме данных с плавающей точкой.

Библиотека Pandas могла бы последовать примеру языка R и задавать комбинации битов для каждого конкретного типа данных для индикации отсутствия значения, но этот подход оказывается довольно громоздким. Ведь если в языке R насчитывается всего четыре базовых типа данных, то NumPy поддерживает *намного* больше. Например, в языке R есть только один целочисленный тип, а библиотека NumPy

поддерживает *четырнадцать* простых целочисленных типов, с учетом различной точности, знаковости/беззнаковости и порядка байтов числа. Резервирование специальной комбинации битов во всех доступных в библиотеке NumPy типах данных привело бы к громадным накладным расходам в разнообразных частных случаях операций для различных типов и, вероятно, потребовало бы даже отдельной ветви пакета NumPy. Кроме того, для небольших типов данных (например, 8-битных целых чисел) потеря одного бита на маску существенно сузит диапазон доступных для представления этим типом значений.

Библиотека NumPy поддерживает использование маскированных массивов, то есть массивов, к которым присоединены отдельные булевы массивы-маски, предназначенные для маркирования как «плохих» или «хороших» данных. Библиотека Pandas могла унаследовать такую возможность, но накладные расходы на хранение, вычисления и поддержку кода сделали этот вариант малопривлекательным.

По этой причине в библиотеке Pandas было решено использовать для отсутствующих данных индикаторы, а также два уже существующих в Python пустых значения: специальное значение `NaN` с плавающей точкой и объект `None` языка Python. У этого решения есть свои недостатки, но на практике в большинстве случаев оно представляет собой удачный компромисс.

None: отсутствующие данные в языке Python

Первое из используемых библиотекой Pandas значений-индикаторов — `None`, объект-одиночка Python, часто применяемый для обозначения отсутствующих данных в коде на языке Python. В силу того что `None` — объект Python, его нельзя использовать в произвольных массивах библиотек NumPy/Pandas, а только в массивах с типом данных `'object'` (то есть массивах объектов языка Python):

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

In[2]: vals1 = np.array([1, None, 3, 4])
       vals1

Out[2]: array([1, None, 3, 4], dtype=object)
```

Выражение `dtype=object` означает, что наилучший возможный вывод об общем типе элементов содержимого данного массива, который только смогла сделать библиотека NumPy, — то, что они все являются объектами Python. Хотя такая разновидность массивов полезна для определенных целей, все операции над ними будут выполняться на уровне языка Python, с накладными расходами, значительно превышающими расходы на выполнение быстрых операций над массивами с нативными типами данных:

```
In[3]: for dtype in ['object', 'int']:
```

```
            print("dtype =", dtype)
            %timeit np.arange(1E6, dtype=dtype).sum()
            print()

dtype = object
10 loops, best of 3: 78.2 ms per loop

dtype = int
100 loops, best of 3: 3.06 ms per loop
```

Использование объектов языка Python в массивах означает также, что при выполнении функций агрегирования по массиву со значениями `None`, например `sum()` или `min()`, вам будет возвращена ошибка:

```
In[4]: vals1.sum()

TypeError                                 Traceback (most recent call last)

<ipython-input-4-749fd8ae6030> in <module>()
----> 1 vals1.sum()

/Users/jakevdp/anaconda/lib/python3.5/site-packages/numpy/core/_methods.py ...
     30
     31 def _sum(a, axis=None, dtype=None, out=None, keepdims=False):
---> 32     return umr_sum(a, axis, dtype, out, keepdims)
     33
     34 def _prod(a, axis=None, dtype=None, out=None, keepdims=False):

TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

Эта ошибка отражает тот факт, что операция между целочисленным значением и значением `None` не определена.

NaN: отсутствующие числовые данные

Еще одно представление отсутствующих данных, `NaN`, представляет собой специальное значение с плавающей точкой, распознаваемое всеми системами, использующими стандарное IEEE-представление чисел с плавающей точкой:

```
In[5]: vals2 = np.array([1, np.nan, 3, 4])
       vals2.dtype

Out[5]: dtype('float64')
```

Обратите внимание, что библиотека NumPy выбрала для этого массива нативный тип с плавающей точкой: это значит, что, в отличие от вышеупомянутого массива объектов, этот массив поддерживает быстрые операции, передаваемые на выполнение скомпилированному коду. Вы должны отдавать себе отчет, что значение

NaN в чем-то подобно «вирусу данных»: оно «заражает» любой объект, к которому «прикасается». Вне зависимости от операции результат арифметического действия с участием NaN будет равен NaN:

```
In[6]: 1 + np.nan

Out[6]: nan

In[7]: 0 *  np.nan

Out[7]: nan
```

Это значит, что операция агрегирования значений определена (то есть ее результатом не будет ошибка), но не всегда приносит пользу:

```
In[8]: vals2.sum(), vals2.min(), vals2.max()

Out[8]: (nan, nan, nan)
```

Библиотека NumPy предоставляет специализированные агрегирующие функции, игнорирующие эти пропущенные значения:

```
In[9]: np.nansum(vals2), np.nanmin(vals2), np.nanmax(vals2)

Out[9]: (8.0, 1.0, 4.0)
```

Не забывайте, что NaN — именно значение с плавающей точкой, аналога значения NaN для целочисленных значений, строковых и других типов не существует.

Значения NaN и None в библиотеке Pandas

Как у значения NaN, так и у None есть свое назначение, и библиотека Pandas делает их практически взаимозаменяемыми путем преобразования одного в другое в определенных случаях:

```
In[10]: pd.Series([1, np.nan, 2, None])

Out[10]: 0    1.0
         1    NaN
         2    2.0
         3    NaN
         dtype: float64
```

Библиотека Pandas автоматически выполняет преобразование при обнаружении *NA*-значений для тех типов, у которых отсутствует значение-индикатор. Например, если задать значение элемента целочисленного массива равным np.nan, для соответствия типу отсутствующего значения будет автоматически выполнено повышающее приведение типа этого *массива* к типу с плавающей точкой:

```
In[11]: x = pd.Series(range(2), dtype=int)
        x

Out[11]: 0    0
         1    1
         dtype: int64

In[12]: x[0] = None
        x

Out[12]: 0    NaN
         1    1.0
         dtype: float64
```

Обратите внимание, что, помимо приведения типа целочисленного массива к массиву значений с плавающей точкой, библиотека Pandas автоматически преобразует значение `None` в `NaN`. Замечу, что существует план по внесению в будущем нативного целочисленного *NA* в библиотеку Pandas, но на момент написания данной книги оно еще не было включено.

Хотя подобный подход со значениями-индикаторами/приведением типов библиотеки Pandas может показаться несколько вычурным по сравнению с более унифицированным подходом к *NA*-значениям в таких предметно-ориентированных языках, как R, на практике он прекрасно работает и на моей памяти лишь изредка вызывал проблемы.

В табл. 3.2 перечислены правила повышающего приведения типов в библиотеке Pandas в случае наличия *NA*-значений.

Таблица 3.2. Правила повышающего приведения типов в библиотеке Pandas

Класс типов	Преобразование при хранении NA-значений	Значение-индикатор NA
С плавающей точкой	Без изменений	np.nan
Объект (object)	Без изменений	None или np.nan
Целое число	Приводится к float64	np.nan
Булево значение	Приводится к object	None или np.nan

Имейте в виду, что строковые данные в библиотеке Pandas всегда хранятся с типом данных (dtype) `object`.

Операции над пустыми значениями

Библиотека Pandas рассматривает значения `None` и `NaN` как взаимозаменяемые средства указания на отсутствующие или пустые значения. Существует несколько удобных методов для обнаружения, удаления и замены пустых значений в структурах данных библиотеки Pandas, призванных упростить работу с ними.

- `isnull()` — генерирует булеву маску для отсутствующих значений.
- `notnull()` — противоположность метода `isnull()`.
- `dropna()` — возвращает отфильтрованный вариант данных.
- `fillna()` — возвращает копию данных, в которой пропущенные значения заполнены или восстановлены.

Завершим раздел кратким рассмотрением и демонстрацией этих методов.

Выявление пустых значений

У структур данных библиотеки Pandas имеются два удобных метода для выявления пустых значений: `isnull()` и `notnull()`. Каждый из них возвращает булеву маску для данных. Например:

```
In[13]: data = pd.Series([1, np.nan, 'hello', None])

In[14]: data.isnull()

Out[14]: 0    False
         1     True
         2    False
         3     True
         dtype: bool
```

Как уже упоминалось в разделе «Индексация и выборка данных» этой главы, булевы маски можно использовать непосредственно в качестве индекса объектов `Series` или `DataFrame`:

```
In[15]: data[data.notnull()]

Out[15]: 0        1
         2    hello
         dtype: object
```

Аналогичные булевы результаты дает использование методов `isnull()` и `notnull()` для объектов `DataFrame`.

Удаление пустых значений

Помимо продемонстрированного выше маскирования, существуют удобные методы: `dropna()` (отбрасывающий *NA*-значения) и `fillna()` (заполняющий *NA*-значения). Для объекта `Series` результат вполне однозначен:

```
In[16]: data.dropna()

Out[16]: 0        1
         2    hello
         dtype: object
```

В случае объектов DataFrame существует несколько параметров. Рассмотрим следующий объект DataFrame:

```
In[17]: df = pd.DataFrame([[1,      np.nan, 2],
                           [2,      3,      5],
                           [np.nan, 4,      6]])
        df

Out[17]:      0    1  2
         0  1.0  NaN  2
         1  2.0  3.0  5
         2  NaN  4.0  6
```

Нельзя выбросить из DataFrame отдельные значения, только целые строки или столбцы. В зависимости от приложения может понадобиться тот или иной вариант, так что функция dropna() предоставляет для случая объектов DataFrame несколько параметров.

По умолчанию dropna() отбрасывает все строки, в которых присутствует *хотя бы одно* пустое значение:

```
In[18]: df.dropna()

Out[18]:      0    1  2
         1  2.0  3.0  5
```

В качестве альтернативы можно отбрасывать *NA*-значения по разным осям: задание параметра axis=1 отбрасывает все столбцы, содержащие хотя бы одно пустое значение:

```
In[19]: df.dropna(axis='columns')

Out[19]:    2
         0  2
         1  5
         2  6
```

Однако при этом отбрасываются также и некоторые «хорошие» данные. Возможно, вам захочется отбросить строки или столбцы, *все* значения (или большинство) в которых представляют собой *NA*. Такое поведение можно задать с помощью параметров how и thresh, обеспечивающих точный контроль допустимого количества пустых значений.

По умолчанию how='any', то есть отбрасываются все строки или столбцы (в зависимости от ключевого слова axis), содержащие хоть одно пустое значение. Можно также указать значение how='all', при нем будут отбрасываться только строки/столбцы, *все* значения в которых пустые:

```
In[20]: df[3] = np.nan
        df

Out[20]:      0    1  2   3
```

```
         0  1.0  NaN  2 NaN
         1  2.0  3.0  5 NaN
         2  NaN  4.0  6 NaN

In[21]: df.dropna(axis='columns', how='all')

Out[21]:      0    1  2
         0  1.0  NaN  2
         1  2.0  3.0  5
         2  NaN  4.0  6
```

Для более точного контроля можно задать с помощью параметра `thresh` минимальное количество непустых значений для строки/столбца, при котором он не отбрасывается:

```
In[22]: df.dropna(axis='rows', thresh=3)

Out[22]:      0    1  2   3
         1  2.0  3.0  5 NaN
```

В данном случае отбрасываются первая и последняя строки, поскольку в них содержится только по два непустых значения.

Заполнение пустых значений

Иногда предпочтительнее вместо отбрасывания пустых значений заполнить их каким-то допустимым значением. Это значение может быть фиксированным, например нулем, или интерполированным или восстановленным на основе «хороших» данных значением. Это можно сделать путем замены в исходных данных, используя результат метода `isnull()` в качестве маски. Но это настолько распространенная операция, что библиотека Pandas предоставляет метод `fillna()`, возвращающий копию массива с замененными пустыми значениями.

Рассмотрим следующий объект `Series`:

```
In[23]: data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))
        data

Out[23]: a    1.0
         b    NaN
         c    2.0
         d    NaN
         e    3.0
         dtype: float64
```

Можно заполнить *NA*-элементы одним фиксированным значением, например, нулями:

```
In[24]: data.fillna(0)

Out[24]: a    1.0
```

```
b    0.0
c    2.0
d    0.0
e    3.0
dtype: float64
```

Можно задать параметр заполнения по направлению «вперед», копируя предыдущее значение в следующую ячейку:

```
In[25]: # заполнение по направлению «вперед»
        data.fillna(method='ffill')

Out[25]: a    1.0
         b    1.0
         c    2.0
         d    2.0
         e    3.0
         dtype: float64
```

Или можно задать параметр заполнения по направлению «назад», копируя следующее значение в предыдущую ячейку:

```
In[26]: # заполнение по направлению «назад»
        data.fillna(method='bfill')

Out[26]: a    1.0
         b    2.0
         c    2.0
         d    3.0
         e    3.0
         dtype: float64
```

Для объектов `DataFrame` опции аналогичны, но в дополнение можно задать ось, по которой будет выполняться заполнение:

```
In[27]: df

Out[27]:      0    1  2   3
         0  1.0  NaN  2 NaN
         1  2.0  3.0  5 NaN
         2  NaN  4.0  6 NaN

In[28]: df.fillna(method='ffill', axis=1)

Out[28]:      0    1    2    3
         0  1.0  1.0  2.0  2.0
         1  2.0  3.0  5.0  5.0
         2  NaN  4.0  6.0  6.0
```

Обратите внимание, что если при заполнении по направлению «вперед» предыдущего значения нет, то *NA*-значение остается незаполненным.

Иерархическая индексация

До сих пор мы рассматривали главным образом одномерные и двумерные данные, находящиеся в объектах `Series` и `DataFrame` библиотеки Pandas. Часто бывает удобно выйти за пределы двух измерений и хранить многомерные данные, то есть данные, индексированные по более чем двум ключам. Хотя библиотека Pandas предоставляет объекты `Panel` и `Panel4D`, позволяющие нативным образом хранить трехмерные и четырехмерные данные (см. врезку «Данные объектов Panel» на с. 178), на практике намного чаще используется *иерархическая индексация* (hierarchical indexing), или *мультииндексация* (multi-indexing), для включения в один индекс нескольких *уровней*. При этом многомерные данные могут быть компактно представлены в уже привычных нам одномерных объектах `Series` и двумерных объектах `DataFrame`.

В этом разделе мы рассмотрим создание объектов `MultiIndex` напрямую, приведем соображения относительно индексации, срезов и вычисления статистических показателей по мультииндексированным данным, а также полезные методы для преобразования между простым и иерархически индексированным представлением данных.

Начнем с обычных импортов:

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

Мультииндексированный объект Series

Рассмотрим, как можно представить двумерные данные в одномерном объекте `Series`. Для конкретики изучим ряд данных, в котором у каждой точки имеются символьный и числовой ключи.

Плохой способ

Пускай нам требуется проанализировать данные о штатах за два разных года. Вам может показаться соблазнительным, воспользовавшись утилитами библиотеки Pandas, применить в качестве ключей кортежи языка Python:

```
In[2]: index = [('California', 2000), ('California', 2010),
                ('New York', 2000), ('New York', 2010),
                ('Texas', 2000), ('Texas', 2010)]
       populations = [33871648, 37253956,
                      18976457, 19378102,
                      20851820, 25145561]
       pop = pd.Series(populations, index=index)
       pop

Out[2]: (California, 2000)    33871648
```

```
        (California, 2010)    37253956
        (New York, 2000)      18976457
        (New York, 2010)      19378102
        (Texas, 2000)         20851820
        (Texas, 2010)         25145561
        dtype: int64
```

При подобной схеме индексации появляется возможность непосредственно индексировать или выполнять срез ряда данных на основе такого мультииндекса:

```
In[3]: pop[('California', 2010):('Texas', 2000)]

Out[3]: (California, 2010)    37253956
        (New York, 2000)      18976457
        (New York, 2010)      19378102
        (Texas, 2000)         20851820
        dtype: int64
```

Однако на этом удобство заканчивается. Например, при необходимости выбрать все значения из 2010 года придется проделать громоздкую (и потенциально медленную) очистку данных:

```
In[4]: pop[[i for i in pop.index if i[1] == 2010]]

Out[4]: (California, 2010)    37253956
        (New York, 2010)      19378102
        (Texas, 2010)         25145561
        dtype: int64
```

Это хоть и приводит к желаемому результату, но гораздо менее изящно (и далеко не так эффективно), как использование синтаксиса срезов, столь полюбившегося нам в библиотеке Pandas.

Лучший способ

В библиотеке Pandas есть лучший способ выполнения таких операций. Наша индексация, основанная на кортежах, по сути, является примитивным мультииндексом, и тип `MultiIndex` библиотеки Pandas как раз обеспечивает необходимые нам операции. Создать мультииндекс из кортежей можно следующим образом:

```
In[5]: index = pd.MultiIndex.from_tuples(index)
       index

Out[5]: MultiIndex(levels=[['California', 'New York', 'Texas'], [2000, 2010]],
                   labels=[[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]])
```

Обратите внимание, что `MultiIndex` содержит несколько *уровней* (levels) индексации. В данном случае названия штатов и годы, а также несколько кодирующих эти уровни *меток* (labels) для каждой точки данных.

Проиндексировав заново наши ряды данных с помощью `MultiIndex`, мы увидим иерархическое представление данных:

```
In[6]: pop = pop.reindex(index)
       pop

Out[6]: California  2000    33871648
                    2010    37253956
        New York    2000    18976457
                    2010    19378102
        Texas       2000    20851820
                    2010    25145561
        dtype: int64
```

Здесь первые два столбца представления объекта `Series` отражают значения мультииндекса, а третий столбец — данные. Обратите внимание, что в первом столбце отсутствуют некоторые элементы: в этом мультииндексном представлении все пропущенные элементы означают то же значение, что и строкой выше.

Теперь для выбора всех данных, второй индекс которых равен 2010, можно просто воспользоваться синтаксисом срезов библиотеки Pandas:

```
In[7]: pop[:, 2010]

Out[7]: California    37253956
        New York      19378102
        Texas         25145561
        dtype: int64
```

Результат представляет собой массив с одиночным индексом и только теми ключами, которые нас интересуют. Такой синтаксис намного удобнее (а операция выполняется гораздо быстрее!), чем мультииндексное решение на основе кортежей, с которого мы начали. Сейчас мы обсудим подробнее подобные операции индексации над иерархически индексированными данными.

Мультииндекс как дополнительное измерение

Мы могли с легкостью хранить те же самые данные с помощью простого объекта `DataFrame` с индексом и метками столбцов. На самом деле библиотека Pandas создана с учетом этой равнозначности. Метод `unstack()` может быстро преобразовать мультииндексный объект `Series` в индексированный обычным образом объект `DataFrame`:

```
In[8]: pop_df = pop.unstack()
       pop_df

Out[8]:                 2000      2010
        California  33871648  37253956
        New York    18976457  19378102
        Texas       20851820  25145561
```

Как и можно ожидать, метод `stack()` выполняет противоположную операцию:

```
In[9]: pop_df.stack()

Out[9]:  California  2000    33871648
                    2010    37253956
         New York    2000    18976457
                    2010    19378102
         Texas       2000    20851820
                    2010    25145561
         dtype: int64
```

Почему вообще имеет смысл возиться с иерархической индексацией? Причина проста: аналогично тому, как мы использовали мультииндексацию для представления двумерных данных в одномерном объекте `Series`, можно использовать ее для представления данных с тремя или более измерениями в объектах `Series` или `DataFrame`. Каждый новый уровень в мультииндексе представляет дополнительное измерение данных. Благодаря использованию этого свойства мы получаем намного больше свободы в представлении типов данных. Например, нам может понадобиться добавить в демографические данные по каждому штату за каждый год еще один столбец (допустим, количество населения младше 18 лет). Благодаря типу `MultiIndex` это сводится к добавлению еще одного столбца в объект `DataFrame`:

```
In[10]: pop_df = pd.DataFrame({'total': pop,
                               'under18': [9267089, 9284094,
                                           4687374, 4318033,
                                           5906301, 6879014]})
        pop_df

Out[10]:                    total  under18
         California 2000  33871648  9267089
                    2010  37253956  9284094
         New York   2000  18976457  4687374
                    2010  19378102  4318033
         Texas      2000  20851820  5906301
                    2010  25145561  6879014
```

Помимо этого, все универсальные функции и остальная функциональность, обсуждавшаяся в разделе «Операции над данными в библиотеке Pandas» этой главы, также прекрасно работают с иерархическими индексами. В следующем фрагменте кода мы вычисляем по годам долю населения младше 18 лет на основе вышеприведенных данных:

```
In[11]: f_u18 = pop_df['under18'] / pop_df['total']
        f_u18.unstack()

Out[11]:                 2000      2010
         California  0.273594  0.249211
         New York    0.247010  0.222831
         Texas       0.283251  0.273568
```

Это дает нам возможность легко и быстро манипулировать даже многомерными данными и исследовать их.

Методы создания мультииндексов

Наиболее простой метод создания мультииндексированного объекта `Series` или `DataFrame` — передать в конструктор список из двух или более индексных массивов. Например:

```
In[12]: df = pd.DataFrame(np.random.rand(4, 2),
                          index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
                          columns=['data1', 'data2'])
        df

Out[12]:         data1     data2
         a 1  0.554233  0.356072
           2  0.925244  0.219474
         b 1  0.441759  0.610054
           2  0.171495  0.886688
```

Вся работа по созданию мультииндекса выполняется в фоновом режиме.

Если передать словарь с соответствующими кортежами в качестве ключей, библиотека Pandas автоматически распознает такой синтаксис и будет по умолчанию использовать мультииндекс:

```
In[13]: data = {('California', 2000): 33871648,
                ('California', 2010): 37253956,
                ('Texas', 2000): 20851820,
                ('Texas', 2010): 25145561,
                ('New York', 2000): 18976457,
                ('New York', 2010): 19378102}
        pd.Series(data)

Out[13]: California  2000    33871648
                     2010    37253956
         New York    2000    18976457
                     2010    19378102
         Texas       2000    20851820
                     2010    25145561
         dtype: int64
```

Тем не менее иногда бывает удобно создавать объекты `MultiIndex` явным образом. Далее мы рассмотрим несколько методов для этого.

Явные конструкторы для объектов MultiIndex

При формировании индекса для большей гибкости можно воспользоваться имеющимися в классе `pd.MultiIndex` конструкторами-методами класса. Например, можно сформировать объект `MultiIndex` из простого списка массивов, задающих значения индекса в каждом из уровней:

```
In[14]: pd.MultiIndex.from_arrays([['a', 'a', 'b', 'b'], [1, 2, 1, 2]])

Out[14]: MultiIndex(levels=[['a', 'b'], [1, 2]],
                    labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

Или из списка кортежей, задающих все значения индекса в каждой из точек:

```
In[15]: pd.MultiIndex.from_tuples([('a', 1), ('a', 2), ('b', 1), ('b', 2)])

Out[15]: MultiIndex(levels=[['a', 'b'], [1, 2]],
                    labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

Или из декартова произведения обычных индексов:

```
In[16]: pd.MultiIndex.from_product([['a', 'b'], [1, 2]])

Out[16]: MultiIndex(levels=[['a', 'b'], [1, 2]],
                    labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

Можно сформировать объект `MultiIndex` непосредственно с помощью его внутреннего представления, передав в конструктор `levels` (список списков, содержащих имеющиеся значения индекса для каждого уровня) и `labels` (список списков меток):

```
In[17]: pd.MultiIndex(levels=[['a', 'b'], [1, 2]],
                      labels=[[0, 0, 1, 1], [0, 1, 0, 1]])

Out[17]: MultiIndex(levels=[['a', 'b'], [1, 2]],
                    labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

Любой из этих объектов можно передать в качестве аргумента метода `index` при создании объектов `Series` или `DataFrame` или методу `reindex` уже существующих объектов `Series` или `DataFrame`.

Названия уровней мультииндексов

Иногда бывает удобно задать названия для уровней объекта `MultiIndex`. Сделать это можно, передав аргумент `names` любому из вышеперечисленных конструкторов класса `MultiIndex` или задав значения атрибута `names` постфактум:

```
In[18]: pop.index.names = ['state', 'year']
        pop

Out[18]: state       year
         California  2000    33871648
                     2010    37253956
         New York    2000    18976457
                     2010    19378102
         Texas       2000    20851820
                     2010    25145561
         dtype: int64
```

В случае более сложных наборов данных такой способ дает возможность не терять из виду, что означают различные значения индекса.

Мультииндекс для столбцов

В объектах `DataFrame` строки и столбцы полностью симметричны, и у столбцов, точно так же, как и у строк, может быть несколько уровней индексов. Рассмотрим следующий пример, представляющий собой имитацию неких (в чем-то достаточно реалистичных) медицинских данных:

```
In[19]:
# Иерархические индексы и столбцы
index = pd.MultiIndex.from_product([[2013, 2014], [1, 2]],
                                   names=['year', 'visit'])
columns = pd.MultiIndex.from_product([['Bob', 'Guido', 'Sue'],
                                      ['HR', 'Temp']],
                                     names=['subject', 'type'])

# Создаем имитационные данные
data = np.round(np.random.randn(4, 6), 1)
data[:, ::2] *= 10
data += 37

# Создаем объект DataFrame
health_data = pd.DataFrame(data, index=index, columns=columns)
health_data
```

```
Out[19]: subject      Bob       Guido         Sue
         type          HR  Temp    HR  Temp    HR  Temp
         year visit
         2013 1      31.0  38.7  32.0  36.7  35.0  37.2
              2      44.0  37.7  50.0  35.0  29.0  36.7
         2014 1      30.0  37.4  39.0  37.8  61.0  36.9
              2      47.0  37.8  48.0  37.3  51.0  36.5
```

Как видим, мультииндексация как строк, так и столбцов может оказаться *чрезвычайно* удобной. По сути дела, это четырехмерные данные со следующими измерениями: субъект, измеряемый параметр[1], год и номер посещения. При наличии этого мы можем, например, индексировать столбец верхнего уровня по имени человека и получить объект `DataFrame`, содержащий информацию только об этом человеке:

```
In[20]: health_data['Guido']
Out[20]: type          HR  Temp
         year visit
         2013 1      32.0  36.7
              2      50.0  35.0
         2014 1      39.0  37.8
              2      48.0  37.3
```

[1] Пульс (HR, от англ. heart rate — «частота сердцебиений») и температура (temp).

Для сложных записей, содержащих несколько маркированных неоднократно измеряемых параметров для многих субъектов (людей, стран, городов и т. д.), будет исключительно удобно использовать иерархические строки и столбцы!

Индексация и срезы по мультииндексу

Объект `MultiIndex` спроектирован так, чтобы индексация и срезы по мультииндексу были интуитивно понятны, особенно если думать об индексах как о дополнительных измерениях. Изучим сначала индексацию мультииндексированного объекта `Series`, а затем мультииндексированного объекта `DataFrame`.

Мультииндексация объектов Series

Рассмотрим мультииндексированный объект `Series`, содержащий количество населения по штатам:

```
In[21]: pop

Out[21]: state       year
         California  2000    33871648
                     2010    37253956
         New York    2000    18976457
                     2010    19378102
         Texas       2000    20851820
                     2010    25145561
         dtype: int64
```

Обращаться к отдельным элементам можно путем индексации с помощью нескольких термов:

```
In[22]: pop['California', 2000]

Out[22]: 33871648
```

Объект `MultiIndex` поддерживает также *частичную индексацию* (partial indexing), то есть индексацию только по одному из уровней индекса. Результат — тоже объект `Series`, с более низкоуровневыми индексами:

```
In[23]: pop['California']

Out[23]: year
         2000    33871648
         2010    37253956
         dtype: int64
```

Возможно также выполнение частичных срезов, если мультииндекс отсортирован (см. обсуждение в пункте «Отсортированные и неотсортированные индексы» подраздела «Перегруппировка мультииндексов» данного раздела):

```
In[24]: pop.loc['California':'New York']

Out[24]: state       year
         California  2000    33871648
                     2010    37253956
         New York    2000    18976457
                     2010    19378102
         dtype: int64
```

С помощью отсортированных индексов можно выполнять частичную индексацию по нижним уровням, указав пустой срез в первом индексе:

```
In[25]: pop[:, 2000]

Out[25]: state
         California    33871648
         New York      18976457
         Texas         20851820
         dtype: int64
```

Другие типы индексации и выборки (обсуждаемые в разделе «Индексация и выборка данных» этой главы) тоже работают. Выборка данных на основе булевой маски:

```
In[26]: pop[pop > 22000000]

Out[26]: state       year
         California  2000    33871648
                     2010    37253956
         Texas       2010    25145561
         dtype: int64
```

Выборка на основе «прихотливой» индексации:

```
In[27]: pop[['California', 'Texas']]

Out[27]: state       year
         California  2000    33871648
                     2010    37253956
         Texas       2000    20851820
                     2010    25145561
         dtype: int64
```

Мультииндексация объектов DataFrame

Мультииндексированные объекты `DataFrame` ведут себя аналогичным образом. Рассмотрим наш модельный медицинский объект `DataFrame`:

```
In[28]: health_data

Out[28]: subject      Bob       Guido         Sue
         type          HR  Temp    HR  Temp    HR  Temp
         year visit
```

```
         2013 1      31.0  38.7  32.0  36.7  35.0  37.2
              2      44.0  37.7  50.0  35.0  29.0  36.7
         2014 1      30.0  37.4  39.0  37.8  61.0  36.9
              2      47.0  37.8  48.0  37.3  51.0  36.5
```

Не забывайте, что в объектах `DataFrame` основными являются столбцы, и используемый для мультииндексированных `Series` синтаксис применяется тоже к столбцам. Например, можно узнать пульс Гвидо с помощью простой операции:

```
In[29]: health_data['Guido', 'HR']

Out[29]: year  visit
         2013  1        32.0
               2        50.0
         2014  1        39.0
               2        48.0
         Name: (Guido, HR), dtype: float64
```

Как и в случае с одиночным индексом, можно использовать индексаторы `loc`, `iloc` и `ix`, описанные в разделе «Индексация и выборка данных» этой главы. Например:

```
In[30]: health_data.iloc[:2, :2]

Out[30]: subject      Bob
         type          HR  Temp
         year visit
         2013 1      31.0  38.7
              2      44.0  37.7
```

Эти индексаторы возвращают массивоподобное представление исходных двумерных данных, но в каждом отдельном индексе в `loc` и `iloc` можно указать кортеж из нескольких индексов. Например:

```
In[31]: health_data.loc[:, ('Bob', 'HR')]

Out[31]: year  visit
         2013  1        31.0
               2        44.0
         2014  1        30.0
               2        47.0
         Name: (Bob, HR), dtype: float64
```

Работать со срезами в подобных кортежах индексов не очень удобно: попытка создать срез в кортеже может привести к синтаксической ошибке:

```
In[32]: health_data.loc[(:, 1), (:, 'HR')]

  File "<IPython-input-32-8e3cc151e316>", line 1
    health_data.loc[(:, 1), (:, 'HR')]
                    ^
SyntaxError: invalid syntax
```

Избежать этого можно, сформировав срез явным образом с помощью встроенной функции Python `slice()`, но лучше в данном случае использовать объект `IndexSlice`, предназначенный библиотекой Pandas как раз для подобной ситуации. Например:

```
In[33]: idx = pd.IndexSlice
        health_data.loc[idx[:, 1], idx[:, 'HR']]

Out[33]: subject      Bob Guido   Sue
         type          HR    HR    HR
         year visit
         2013 1      31.0  32.0  35.0
         2014 1      30.0  39.0  61.0
```

Существует множество способов взаимодействия с данными в мультииндексированных объектах `Series` и `DataFrame`, и лучший способ привыкнуть к ним — начать с ними экспериментировать!

Перегруппировка мультииндексов

Один из ключей к эффективной работе с мультииндексированными данными — умение эффективно преобразовывать данные. Существует немало операций, сохраняющих всю информацию из набора данных, но преобразующих ее ради удобства проведения различных вычислений. Мы рассмотрели небольшой пример этого с методами `stack()` и `unstack()`, но есть гораздо больше способов точного контроля над перегруппировкой данных между иерархическими индексами и столбцами.

Отсортированные и неотсортированные индексы

Большинство операций срезов с мультииндексами завершится ошибкой, если индекс не отсортирован. Рассмотрим этот вопрос.

Начнем с создания простых мультииндексированных данных, *индексы в которых не отсортированы лексикографически*:

```
In[34]: index = pd.MultiIndex.from_product([['a', 'c', 'b'], [1, 2]])
        data = pd.Series(np.random.rand(6), index=index)
        data.index.names = ['char', 'int']
        data

Out[34]: char  int
         a     1      0.003001
               2      0.164974
         c     1      0.741650
               2      0.569264
         b     1      0.001693
               2      0.526226
         dtype: float64
```

Если мы попытаемся выполнить частичный срез этого индекса, то получим ошибку:

```
In[35]: try:
            data['a':'b']
        except KeyError as e:
            print(type(e))
            print(e)

<class 'KeyError'>
'Key length (1) was greater than MultiIndex lexsort depth (0)'
```

Хотя из сообщения об ошибке[1] это не вполне понятно, ошибка генерируется, потому что объект `MultiIndex` не отсортирован. По различным причинам частичные срезы и другие подобные операции требуют, чтобы уровни мультииндекса были отсортированы (лексикографически упорядочены). Библиотека Pandas предоставляет множество удобных инструментов для выполнения подобной сортировки. В качестве примеров можем указать методы `sort_index()` и `sortlevel()` объекта `DataFrame`. Мы воспользуемся простейшим из них — методом `sort_index()`:

```
In[36]: data = data.sort_index()
        data

Out[36]: char  int
         a     1      0.003001
               2      0.164974
         b     1      0.001693
               2      0.526226
         c     1      0.741650
               2      0.569264
         dtype: float64
```

После подобной сортировки индекса частичный срез будет выполняться как положено:

```
In[37]: data['a':'b']

Out[37]: char  int
         a     1      0.003001
               2      0.164974
         b     1      0.001693
               2      0.526226
         dtype: float64
```

Выполнение над индексами операций stack и unstack

Существует возможность преобразовывать набор данных из вертикального мультииндексированного в простое двумерное представление, при необходимости указывая требуемый уровень:

---

[1] «Длина ключа была больше, чем глубина лексикографической сортировки объекта MultiIndex».

```
In[38]: pop.unstack(level=0)

Out[38]:  state  California   New York     Texas
          year
          2000     33871648   18976457  20851820
          2010     37253956   19378102  25145561

In[39]: pop.unstack(level=1)

Out[39]: year            2000      2010
         state
         California  33871648  37253956
         New York    18976457  19378102
         Texas       20851820  25145561
```

Методу `unstack()` противоположен по действию метод `stack()`, которым можно воспользоваться, чтобы получить обратно исходный ряд данных:

```
In[40]: pop.unstack().stack()

Out[40]: state       year
         California  2000    33871648
                     2010    37253956
         New York    2000    18976457
                     2010    19378102
         Texas       2000    20851820
                     2010    25145561
         dtype: int64
```

Создание и перестройка индексов

Еще один способ перегруппировки иерархических данных — преобразовать метки индекса в столбцы с помощью метода `reset_index`. Результатом вызова этого метода для нашего ассоциативного словаря населения будет объект `DataFrame` со столбцами *state* (штат) и *year* (год), содержащими информацию, ранее находившуюся в индексе. Для большей ясности можно при желании задать название для представленных в виде столбцов данных:

```
In[41]: pop_flat = pop.reset_index(name='population')
        pop_flat

Out[41]:         state  year  population
         0  California  2000    33871648
         1  California  2010    37253956
         2    New York  2000    18976457
         3    New York  2010    19378102
         4       Texas  2000    20851820
         5       Texas  2010    25145561
```

При работе с реальными данными исходные входные данные очень часто выглядят подобным образом, поэтому удобно создать объект `MultiIndex` из значений

столбцов. Это можно сделать с помощью метода `set_index` объекта `DataFrame`, возвращающего мультииндексированный объект `DataFrame`:

```
In[42]: pop_flat.set_index(['state', 'year'])

Out[42]:                  population
        state      year
        California 2000    33871648
                   2010    37253956
        New York   2000    18976457
                   2010    19378102
        Texas      2000    20851820
                   2010    25145561
```

На практике, я полагаю, этот тип перестройки индекса — один из самых удобных паттернов при работе с реальными наборами данных.

Агрегирование по мультииндексам

В библиотеке Pandas имеются встроенные методы для агрегирования данных, например `mean()`, `sum()` и `max()`. В случае иерархически индексированных данных им можно передать параметр `level` для указания подмножества данных, на котором будет вычисляться сводный показатель.

Например, вернемся к нашим медицинским данным:

```
In[43]: health_data

Out[43]:  subject      Bob       Guido         Sue
          type          HR  Temp    HR  Temp    HR  Temp
          year visit
          2013 1      31.0  38.7  32.0  36.7  35.0  37.2
               2      44.0  37.7  50.0  35.0  29.0  36.7
          2014 1      30.0  37.4  39.0  37.8  61.0  36.9
               2      47.0  37.8  48.0  37.3  51.0  36.5
```

Допустим, нужно усреднить результаты измерений показателей по двум визитам в течение года. Сделать это можно путем указания уровня индекса, который мы хотели бы исследовать, в данном случае года (year):

```
In[44]: data_mean = health_data.mean(level='year')
        data_mean

Out[44]: subject   Bob       Guido          Sue
         type       HR  Temp    HR   Temp    HR   Temp
         year
         2013     37.5  38.2  41.0  35.85  32.0  36.95
         2014     38.5  37.6  43.5  37.55  56.0  36.70
```

Далее, воспользовавшись ключевым словом `axis`, можно получить и среднее значение по уровням по столбцам:

```
In[45]: data_mean.mean(axis=1, level='type')

Out[45]: type         HR       Temp
         year
         2013  36.833333  37.000000
         2014  46.000000  37.283333
```

Так, всего двумя строками кода мы смогли найти средний пульс и температуру по всем субъектам и визитам за каждый год. Такой синтаксис представляет собой сокращенную форму функциональности `GroupBy`, о которой мы поговорим в разделе «Агрегирование и группировка» главы 3. Хотя наш пример — всего лишь модель, структура многих реальных наборов данных аналогичным образом иерархична.

Данные объектов Panel

В библиотеке Pandas есть еще несколько пока не охваченных нами структур данных, а именно объекты `pd.Panel` и `pd.Panel4D`. Их можно рассматривать как соответственно трехмерное и четырехмерное обобщение (одномерной структуры) объекта `Series` и (двумерной структуры) объекта `DataFrame`. Раз вы уже знакомы с индексацией данных в объектах `Series` и `DataFrame` и манипуляциями над ними, то использование объектов `Panel` и `Panel4D` не должно вызвать у вас затруднений. В частности, возможности индексаторов `loc`, `iloc` и `ix`, обсуждавшихся в разделе «Индексация и выборка данных» текущей главы, с легкостью распространяются на эти структуры более высоких размерностей.

Мы не будем рассматривать многомерные структуры далее в данном тексте, поскольку, как я обнаружил, в большинстве случаев мультииндексация — удобное и концептуально простое представление для данных более высоких размерностей. Помимо этого, многомерные данные по существу — плотное представление данных, в то время как мультииндексация — разреженное представление данных. По мере увеличения размерности плотное представление становится все менее эффективным для большинства реальных наборов данных. В некоторых специализированных приложениях, однако, эти структуры данных могут быть полезны. Если вы захотите узнать больше о структурах `Panel` и `Panel4D`, загляните в ссылки, приведенные в разделе «Дополнительные источники информации» данной главы.

Объединение наборов данных: конкатенация и добавление в конец

Некоторые наиболее интересные исследования выполняются благодаря объединению различных источников данных. Эти операции могут включать в себя что угодно, начиная с простейшей конкатенации двух различных наборов данных до

более сложных соединений и слияний в стиле баз данных, корректно обрабатывающих все возможные частичные совпадения наборов. Объекты `Series` и `DataFrame` созданы в расчете на подобные операции, и библиотека Pandas содержит функции и методы для быстрого и удобного выполнения таких манипуляций.

Мы рассмотрим простую конкатенацию объектов `Series` и `DataFrame` с помощью функции `pd.concat`, углубимся в реализованные в библиотеке Pandas более запутанные слияния и соединения, выполняемые в оперативной памяти.

Начнем с обычных импортов:

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

Для удобства опишем следующую функцию, создающую объект `DataFrame` определенной формы, которая нам пригодится в дальнейшем:

```
In[2]: def make_df(cols, ind):
           """Быстро создаем объект DataFrame"""
           data = {c: [str(c) + str(i) for i in ind]
                   for c in cols}
           return pd.DataFrame(data, ind)

       # Экземпляр DataFrame
       make_df('ABC', range(3))
```

```
Out[2]:     A   B   C
        0  A0  B0  C0
        1  A1  B1  C1
        2  A2  B2  C2
```

Напоминание: конкатенация массивов NumPy

Конкатенация объектов `Series` и `DataFrame` очень похожа на конкатенацию массивов библиотеки NumPy, которую можно осуществить посредством функции `np.concatenate`, обсуждавшейся в разделе «Введение в массивы библиотеки NumPy» главы 2. Напомним, что таким образом можно объединять содержимое двух или более массивов в один:

```
In[4]: x = [1, 2, 3]
       y = [4, 5, 6]
       z = [7, 8, 9]
       np.concatenate([x, y, z])
```

```
Out[4]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Первый аргумент данной функции — список или кортеж объединяемых массивов. Кроме того, она принимает на входе ключевое слово `axis`, дающее возможность задавать ось, по которой будет выполняться конкатенация:

```
In[5]: x = [[1, 2],
            [3, 4]]
       np.concatenate([x, x], axis=1)

Out[5]: array([[1, 2, 1, 2],
               [3, 4, 3, 4]])
```

Простая конкатенация с помощью метода pd.concat

В библиотеке Pandas имеется функция, `pd.concat()`, синтаксис которой аналогичен функции `np.concatenate`, однако она содержит множество параметров, которые мы вскоре обсудим:

```
# Сигнатура функции pd.concat в библиотеке Pandas v0.18
pd.concat(objs, axis=0, join='outer', join_axes=None, ignore_index=False,
          keys=None, levels=None, names=None, verify_integrity=False,
          copy=True)
```

Функцию `pd.concat` можно использовать для простой конкатенации объектов `Series` или `DataFrame` аналогично тому, как функцию `np.concatenate()` можно применять для простой конкатенации массивов:

```
In[6]: ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
       ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
       pd.concat([ser1, ser2])

Out[6]: 1    A
        2    B
        3    C
        4    D
        5    E
        6    F
        dtype: object
```

Она также подходит для конкатенации объектов более высокой размерности, таких как `DataFrame`:

```
In[7]: df1 = make_df('AB', [1, 2])
       df2 = make_df('AB', [3, 4])
       print(df1); print(df2); print(pd.concat([df1, df2]))
df1             df2           pd.concat([df1, df2])
     A   B          A   B          A   B
 1  A1  B1      3  A3  B3      1  A1  B1
 2  A2  B2      4  A4  B4      2  A2  B2
                               3  A3  B3
                               4  A4  B4
```

По умолчанию конкатенация происходит в объекте `DataFrame` построчно, то есть `axis=0`. Аналогично функции `np.concatenate()` функция `pd.concat()` позволяет указывать ось, по которой будет выполняться конкатенация. Рассмотрим следующий пример:

```
In[8]: df3 = make_df('AB', [0, 1])
       df4 = make_df('CD', [0, 1])
       print(df3); print(df4); print(pd.concat([df3, df4], axis='col'))
df3             df4           pd.concat([df3, df4], axis='col')
     A   B          C   D          A   B   C   D
 0  A0  B0       0  C0  D0      0  A0  B0  C0  D0
 1  A1  B1       1  C1  D1      1  A1  B1  C1  D1
```

Мы могли задать ось и эквивалентным образом: `axis=1`, здесь же мы использовали более интуитивно понятный вариант `axis='col'`[1].

Дублирование индексов

Важное различие между функциями `np.concatenate()` и `pd.concat()` состоит в том, что конкатенация из библиотеки Pandas *сохраняет индексы*, даже если в результате некоторые индексы будут дублироваться. Рассмотрим следующий пример:

```
In[9]: x = make_df('AB', [0, 1])
       y = make_df('AB', [2, 3])
       y.index = x.index  # Дублируем индексы!
       print(x); print(y); print(pd.concat([x, y]))
x              y              pd.concat([x, y])
     A   B          A   B          A   B
 0  A0  B0       0  A2  B2       0  A0  B0
 1  A1  B1       1  A3  B3       1  A1  B1
                                  0  A2  B2
                                  1  A3  B3
```

Обратите внимание на повторяющиеся индексы. Хотя в объектах `DataFrame` это допустимо, подобный результат часто может быть нежелателен. Функция `pd.concat()` предоставляет нам несколько способов решения этой проблемы.

Перехват повторов как ошибки. Если вам нужно просто гарантировать, что индексы в возвращаемом функцией `pd.concat()` результате не перекрываются, можно задать флаг `verify_integrity`. В случае равного `True` значения этого флага конкатенация приведет к генерации ошибки при наличии дублирующихся индексов. Вот пример, в котором мы для большей ясности перехватываем и выводим в консоль сообщение об ошибке:

---

[1] В текущей (0.19.2) версии библиотеки Pandas допустимы следующие варианты синтаксиса:
axis='columns'
или
axis=1
Указанный вариант синтаксиса приведет к ошибке. Впрочем, в версии 0.18.1 библиотеки Pandas, используемой автором книги, документированный синтаксис для этой функции допускает только применение числовых значений для параметра axis.

```
In[10]: try:
            pd.concat([x, y], verify_integrity=True)
        except ValueError as e:
            print("ValueError:", e)

ValueError: Indexes have overlapping values: [0, 1]
```

Игнорирование индекса. Иногда индекс сам по себе не имеет значения и лучше его просто проигнорировать. Для этого достаточно установить флаг `ignore_index`. В случае равного `True` значения этого флага конкатенация приведет к созданию нового целочисленного индекса для итогового объекта `Series`:

```
In[11]: print(x); print(y); print(pd.concat([x, y], ignore_index=True))

x             y             pd.concat([x, y], ignore_index=True)
    A   B         A   B         A   B
0  A0  B0     0  A2  B2     0  A0  B0
1  A1  B1     1  A3  B3     1  A1  B1
                            2  A2  B2
                            3  A3  B3
```

Добавление ключей мультииндекса. Еще один вариант — воспользоваться параметром `keys` для задания меток для источников данных. Результатом будут иерархически индексированные ряды, содержащие данные:

```
In[12]: print(x); print(y); print(pd.concat([x, y], keys=['x', 'y']))

x               y               pd.concat([x, y], keys=['x', 'y'])
    A   B           A   B                 A   B
0  A0  B0       0  A2  B2       x  0  A0  B0
1  A1  B1       1  A3  B3          1  A1  B1
                                y  0  A2  B2
                                   1  A3  B3
```

Результат представляет собой мультииндексированный объект `DataFrame`, так что мы сможем воспользоваться описанным в разделе «Иерархическая индексация» этой главы, чтобы преобразовать эти данные в требуемое нам представление.

Конкатенация с использованием соединений

В рассматриваемых примерах в основном производится конкатенация объектов `DataFrame` с общими названиями столбцов. На практике у данных из разных источников могут быть различные наборы имен столбцов. На этот случай у функции `pd.concat()` имеется несколько опций. Изучим объединение следующих двух объектов `DataFrame`, у которых столбцы (но не все!) называются одинаково:

```
In[13]: df5 = make_df('ABC', [1, 2])
        df6 = make_df('BCD', [3, 4])
        print(df5); print(df6); print(pd.concat([df5, df6]))
```

```
df5                df6                pd.concat([df5, df6])
     A   B   C          B   C   D          A   B   C    D
 1  A1  B1  C1     3  B3  C3  D3     1   A1  B1  C1  NaN
 2  A2  B2  C2     4  B4  C4  D4     2   A2  B2  C2  NaN
                                     3  NaN  B3  C3   D3
                                     4  NaN  B4  C4   D4
```

По умолчанию элементы, данные для которых отсутствуют, заполняются *NA*-значениями. Чтобы поменять это поведение, можно указать одну из нескольких опций для параметров `join` и `join_axes` функции конкатенации. По умолчанию соединение — объединение входных столбцов (`join='outer'`), но есть возможность поменять это поведение на пересечение столбцов с помощью опции `join='inner'`:

```
In[14]: print(df5); print(df6);
        print(pd.concat([df5, df6], join='inner'))
```

```
df5                df6                pd.concat([df5, df6], join='inner')
     A   B   C          B   C   D          B   C
 1  A1  B1  C1     3  B3  C3  D3     1  B1  C1
 2  A2  B2  C2     4  B4  C4  D4     2  B2  C2
                                     3  B3  C3
                                     4  B4  C4
```

Еще одна опция предназначена для указания явным образом индекса оставшихся столбцов с помощью аргумента `join_axes`, которому присваивается список объектов индекса. В данном случае мы указываем, что возвращаемые столбцы должны совпадать со столбцами первого из конкатенируемых объектов `DataFrame`:

```
In[15]: print(df5); print(df6);
        print(pd.concat([df5, df6], join_axes=[df5.columns]))
```

```
df5                df6          pd.concat([df5, df6], join_axes=[df5.columns])
     A   B   C          B   C   D          A   B   C
 1  A1  B1  C1     3  B3  C3  D3     1   A1  B1  C1
 2  A2  B2  C2     4  B4  C4  D4     2   A2  B2  C2
                                     3  NaN  B3  C3
                                     4  NaN  B4  C4
```

Различные сочетания опций функции `pd.concat()` обеспечивают широкий диапазон возможных поведений при соединении двух наборов данных. Не забывайте об этом при использовании ее для ваших собственных данных.

Метод append()

Непосредственная конкатенация массивов настолько распространена, что в объекты `Series` и `DataFrame` был включен метод `append()`, позволяющий выполнить то же самое с меньшими усилиями. Например, вместо вызова `pd.concat([df1, df2])` можно вызвать `df1.append(df2)`:

```
In[16]: print(df1); print(df2); print(df1.append(df2))

df1              df2              df1.append(df2)
     A   B            A   B            A   B
 1  A1  B1       3  A3  B3       1   A1  B1
 2  A2  B2       4  A4  B4       2   A2  B2
                                  3   A3  B3
                                  4   A4  B4
```

Не забывайте, что, в отличие от методов `append()` и `extend()` списков языка Python, метод `append()` в библиотеке Pandas не изменяет исходный объект. Вместо этого он создает новый объект с объединенными данными, что делает этот метод не слишком эффективным, поскольку означает создание нового индекса *и* буфера данных. Следовательно, если вам необходимо выполнить несколько операций `append`, лучше создать список объектов `DataFrame` и передать их все сразу функции `concat()`.

В следующем разделе мы рассмотрим другой подход к объединению данных из нескольких источников, обладающий еще более широкими возможностями: выполнение слияний/объединений в стиле баз данных с помощью функции `pd.merge`. Для получения дополнительной информации о методах `concat()`, `append()` и относящейся к ним функциональности см. раздел Merge, Join and Concatenate («Слияние, соединение и конкатенация», http://pandas.pydata.org/pandas-docs/stable/merging.html) документации библиотеки Pandas.

Объединение наборов данных: слияние и соединение

Одно из важных свойств библиотеки Pandas — ее высокопроизводительные, выполняемые в оперативной памяти операции соединения и слияния. Если вы когда-либо работали с базами данных, вам должен быть знаком такой вид взаимодействия с данными. Основной интерфейс для них — функция `pd.merge`. Несколько примеров ее работы на практике мы рассмотрим далее.

Реляционная алгебра

Реализованное в методе `pd.merge` поведение представляет собой подмножество того, что известно под названием *«реляционная алгебра»* (relational algebra). Реляционная алгебра — формальный набор правил манипуляции реляционными данными, формирующий теоретические основания для имеющихся в большинстве баз данных операций. Сила реляционного подхода состоит в предоставлении им нескольких простейших операций — своеобразных «кирпичиков» для построения более сложных операций над любым набором данных. При наличии эффективно

реализованного в базе данных или другой программе подобного базового набора операций можно выполнять широкий диапазон весьма сложных составных операций.

Библиотека Pandas реализует несколько из этих базовых «кирпичиков» в функции `pd.merge()` и родственном ей методе `join()` объектов `Series` и `DataFrame`. Они обеспечивают возможность эффективно связывать данные из различных источников.

Виды соединений

Функция `pd.merge()` реализует множество типов соединений: *«один-к-одному»*, *«многие-к-одному»* и *«многие-ко-многим»*. Все эти три типа соединений доступны через один и тот же вызов `pd.merge()`, тип выполняемого соединения зависит от формы входных данных. Ниже мы рассмотрим примеры этих трех типов слияний и обсудим более подробно имеющиеся параметры.

Соединения «один-к-одному»

Простейший тип выражения слияния — соединение «один-к-одному», во многом напоминающее конкатенацию по столбцам, которую мы изучили в разделе «Объединение наборов данных: конкатенация и добавление в конец» этой главы. В качестве примера рассмотрим следующие два объекта `DataFrame`, содержащие информацию о нескольких служащих компании:

```
In[2]:
df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'group': ['Accounting', 'Engineering', 'Engineering',
                              'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                    'hire_date': [2004, 2008, 2012, 2014]})
print(df1); print(df2)
```

```
df1                         df2
  employee        group       employee  hire_date
0      Bob   Accounting    0     Lisa       2004
1     Jake  Engineering    1      Bob       2008
2     Lisa  Engineering    2     Jake       2012
3      Sue           HR    3      Sue       2014
```

Чтобы объединить эту информацию в один объект `DataFrame`, воспользуемся функцией `pd.merge()`:

```
In[3]: df3 = pd.merge(df1, df2)
       df3

Out[3]:   employee        group  hire_date
        0      Bob   Accounting       2008
```

```
1     Jake  Engineering       2012
2     Lisa  Engineering       2004
3      Sue           HR       2014
```

Функция `pd.merge()` распознает, что в обоих объектах `DataFrame` имеется столбец `employee`, и автоматически выполняет соединение, используя этот столбец в качестве ключа. Результатом слияния становится новый объект `DataFrame`, объединяющий информацию из двух входных объектов. Обратите внимание, что порядок записей в столбцах не обязательно сохраняется: в данном случае сортировка столбца `employee` различна в объектах `df1` и `df2` и функция `pd.merge()` обрабатывает эту ситуацию корректным образом. Кроме того, не забывайте, что слияние игнорирует индекс, за исключением особого случая слияния по индексу (см. пункт «Ключевые слова left_index и right_index» подраздела «Задание ключа слияния» данного раздела).

Соединения «многие-к-одному»

«Многие-к-одному» — соединения, при которых один из двух ключевых столбцов содержит дублирующиеся значения. В случае соединения «многие-к-одному» в итоговом объекте `DataFrame` эти дублирующиеся записи будут сохранены. Рассмотрим следующий пример соединения «многие-к-одному»:

```
In[4]: df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                           'supervisor': ['Carly', 'Guido', 'Steve']})
       print(df3); print(df4); print(pd.merge(df3, df4))

df3                                   df4
  employee        group  hire_date             group supervisor
0      Bob   Accounting       2008    0   Accounting      Carly
1     Jake  Engineering       2012    1  Engineering      Guido
2     Lisa  Engineering       2004    2           HR      Steve
3      Sue           HR       2014

pd.merge(df3, df4)
  employee        group  hire_date supervisor
0      Bob   Accounting       2008      Carly
1     Jake  Engineering       2012      Guido
2     Lisa  Engineering       2004      Guido
3      Sue           HR       2014      Steve
```

В итоговом объекте `DataFrame` имеется дополнительный столбец с информацией о руководителе (supervisor) с повторением информации в одном или нескольких местах в соответствии с вводимыми данными.

Соединения «многие-ко-многим»

Соединения «многие-ко-многим» семантически несколько более сложны, но тем не менее четко определены. Если столбец ключа как в левом, так и в правом массивах

содержит повторяющиеся значения, результат окажется слиянием типа «многие-ко-многим». Рассмотрим следующий пример, в котором объект `DataFrame` отражает один или несколько навыков, соответствующих конкретной группе.

Выполнив соединение «многие-ко-многим», можно выяснить навыки каждого конкретного человека:

```
In[5]: df5 = pd.DataFrame({'group': ['Accounting', 'Accounting',
                                     'Engineering', 'Engineering',
                                     'HR', 'HR'],
                           'skills': ['math', 'spreadsheets', 'coding',
                                      'linux',
                                      'spreadsheets', 'organization']})
print(df1); print(df5); print(pd.merge(df1, df5))
```

```
df1                           df5
  employee        group                 group        skills
0      Bob   Accounting      0   Accounting          math
1     Jake  Engineering      1   Accounting  spreadsheets
2     Lisa  Engineering      2  Engineering        coding
3      Sue           HR      3  Engineering         linux
                              4           HR  spreadsheets
                              5           HR  organization

pd.merge(df1, df5)
  employee        group        skills
0      Bob   Accounting          math
1      Bob   Accounting  spreadsheets
2     Jake  Engineering        coding
3     Jake  Engineering         linux
4     Lisa  Engineering        coding
5     Lisa  Engineering         linux
6      Sue           HR  spreadsheets
7      Sue           HR  organization
```

Эти три типа соединений можно использовать и в других инструментах библиотеки Pandas, что дает возможность реализовать широкий диапазон функциональности. Однако на практике наборы данных редко оказываются такими же «чистыми», как те, с которыми мы имели дело. В следующем разделе мы рассмотрим параметры метода `pd.merge()`, позволяющие более тонко описывать желаемое поведение операций соединения.

Задание ключа слияния

Мы рассмотрели поведение метода `pd.merge()` по умолчанию: он выполняет поиск в двух входных объектах соответствующих названий столбцов и использует найденное в качестве ключа. Однако зачастую имена столбцов не совпадают добуквенно точно, и в методе `pd.merge()` имеется немало параметров для такой ситуации.

Ключевое слово on

Проще всего указать название ключевого столбца с помощью ключевого слова on, в котором указывается название или список названий столбцов:

```
In[6]: print(df1); print(df2); print(pd.merge(df1, df2, on='employee'))

df1                          df2
  employee        group          employee  hire_date
0      Bob   Accounting        0     Lisa       2004
1     Jake  Engineering        1      Bob       2008
2     Lisa  Engineering        2     Jake       2012
3      Sue           HR        3      Sue       2014

pd.merge(df1, df2, on='employee')
  employee        group  hire_date
0      Bob   Accounting       2008
1     Jake  Engineering       2012
2     Lisa  Engineering       2004
3      Sue           HR       2014
```

Этот параметр работает только в том случае, когда в левом и правом объектах `DataFrame` имеется указанное название столбца.

Ключевые слова left_on и right_on

Иногда приходится выполнять слияние двух наборов данных с различными именами столбцов. Например, у нас может быть набор данных, в котором столбец для имени служащего называется `Name`, а не `Employee`. В этом случае можно воспользоваться ключевыми словами `left_on` и `right_on` для указания названий двух нужных столбцов:

```
In[7]:
df3 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'salary': [70000, 80000, 120000, 90000]})
print(df1); print(df3);
print(pd.merge(df1, df3, left_on="employee", right_on="name"))

df1                          df3
  employee        group          name  salary
0      Bob   Accounting        0   Bob   70000
1     Jake  Engineering        1  Jake   80000
2     Lisa  Engineering        2  Lisa  120000
3      Sue           HR        3   Sue   90000

pd.merge(df1, df3, left_on="employee", right_on="name")
  employee        group  name  salary
0      Bob   Accounting   Bob   70000
1     Jake  Engineering  Jake   80000
2     Lisa  Engineering  Lisa  120000
3      Sue           HR   Sue   90000
```

Результат этой операции содержит избыточный столбец, который можно при желании удалить. Например, с помощью имеющегося в объектах `DataFrame` метода `drop()`:

```
In[8]:
pd.merge(df1, df3, left_on="employee", right_on="name").drop('name', axis=1)

Out[8]:   employee        group  salary
        0      Bob   Accounting   70000
        1     Jake  Engineering   80000
        2     Lisa  Engineering  120000
        3      Sue           HR   90000
```

Ключевые слова left_index и right_index

Иногда удобнее вместо слияния по столбцу выполнить слияние по индексу. Допустим, у нас имеются следующие данные:

```
In[9]: df1a = df1.set_index('employee')
       df2a = df2.set_index('employee')
       print(df1a); print(df2a)

df1a                          df2a
                group                      hire_date
employee                       employee
Bob          Accounting        Lisa            2004
Jake       Engineering         Bob             2008
Lisa       Engineering         Jake            2012
Sue                HR          Sue             2014
```

Можно использовать индекс в качестве ключа слияния путем указания в методе `pd.merge()` флагов `left_index` и/или `right_index`:

```
In[10]:
print(df1a); print(df2a);
print(pd.merge(df1a, df2a, left_index=True, right_index=True))

df1a                          df2a
                group                      hire_date
employee                       employee
Bob          Accounting        Lisa            2004
Jake       Engineering         Bob             2008
Lisa       Engineering         Jake            2012
Sue                HR          Sue             2014

pd.merge(df1a, df2a, left_index=True, right_index=True)
                 group  hire_date
employee
Lisa       Engineering       2004
Bob          Accounting      2008
Jake       Engineering       2012
Sue                HR        2014
```

Для удобства в объектах `DataFrame` реализован метод `join()`, выполняющий по умолчанию слияние по индексам:

```
In[11]: print(df1a); print(df2a); print(df1a.join(df2a))

df1a                      df2a
                group                hire_date
employee                  employee
Bob        Accounting     Lisa           2004
Jake      Engineering     Bob            2008
Lisa      Engineering     Jake           2012
Sue                HR     Sue            2014

df1a.join(df2a)
                group  hire_date
employee
Bob        Accounting       2008
Jake      Engineering       2012
Lisa      Engineering       2004
Sue                HR       2014
```

Если требуется комбинация слияния по столбцам и индексам, можно для достижения нужного поведения воспользоваться сочетанием флага `left_index` с параметром `right_on` или параметра `left_on` с флагом `right_index`:

```
In[12]:
print(df1a); print(df3);
print(pd.merge(df1a, df3, left_index=True, right_on='name'))

df1a                        df3
                group
employee                    name  salary
Bob        Accounting    0   Bob   70000
Jake      Engineering    1  Jake   80000
Lisa      Engineering    2  Lisa  120000
Sue                HR    3   Sue   90000

pd.merge(df1a, df3, left_index=True, right_on='name')
          group  name  salary
0   Accounting   Bob   70000
1  Engineering  Jake   80000
2  Engineering  Lisa  120000
3           HR   Sue   90000
```

Все эти параметры работают и в случае нескольких индексов и/или столбцов, синтаксис для этого интуитивно понятен. Более подробную информацию по этому вопросу см. в разделе Merge, Join and Concatenate («Слияние, соединение и конкатенация», http://pandas.pydata.org/pandas-docs/stable/merging.html) документации библиотеки Pandas.

Задание операций над множествами для соединений

Во всех предыдущих примерах мы игнорировали один важный нюанс выполнения соединения — вид используемой при соединении операции алгебры множеств. Это играет важную роль в случаях, когда какое-либо значение есть в одном ключевом столбце, но отсутствует в другом. Рассмотрим следующий пример:

```
In[13]: df6 = pd.DataFrame({'name': ['Peter', 'Paul', 'Mary'],
                            'food': ['fish', 'beans', 'bread']},
                           columns=['name', 'food'])
        df7 = pd.DataFrame({'name': ['Mary', 'Joseph'],
                            'drink': ['wine', 'beer']},
                           columns=['name', 'drink'])
        print(df6); print(df7); print(pd.merge(df6, df7))
```

```
df6                 df7                pd.merge(df6, df7)
    name   food          name drink        name   food  drink
0  Peter   fish     0    Mary  wine   0   Mary  bread   wine
1   Paul  beans     1  Joseph  beer
2   Mary  bread
```

Здесь мы слили воедино два набора данных, у которых совпадает только одна запись `name`: `Mary`. По умолчанию результат будет содержать *пересечение* двух входных множеств — *внутреннее соединение* (inner join). Можно указать это явным образом, с помощью ключевого слова `how`, имеющего по умолчанию значение `'inner'`:

```
In[14]: pd.merge(df6, df7, how='inner')

Out[14]:    name   food drink
         0  Mary  bread  wine
```

Другие возможные значения ключевого слова `how`: `'outer'`, `'left'` и `'right'`. *Внешнее соединение* (outer join) означает соединение по объединению входных столбцов и заполняет значениями *NA* все пропуски значений:

```
In[15]: print(df6); print(df7); print(pd.merge(df6, df7, how='outer'))
```

```
df6                 df7                pd.merge(df6, df7, how='outer')
     name   food          name drink         name   food drink
0  Peter   fish     0    Mary  wine    0   Peter   fish   NaN
1   Paul  beans     1  Joseph  beer    1    Paul  beans   NaN
2   Mary  bread                        2    Mary  bread  wine
                                       3  Joseph    NaN  beer
```

Левое соединение (left join) и *правое соединение* (right join) выполняют соединение по записям слева и справа соответственно. Например:

```
In[16]: print(df6); print(df7); print(pd.merge(df6, df7, how='left'))
```

```
df6                 df7                pd.merge(df6, df7, how='left')
     name   food          name drink         name   food drink
```

```
0   Peter   fish     0    Mary  wine    0   Peter   fish   NaN
1    Paul  beans     1  Joseph  beer    1    Paul  beans   NaN
2    Mary  bread                        2    Mary  bread  wine
```

Строки результата теперь соответствуют записям в левом из входных объектов. Опция `how='right'` работает аналогичным образом.

Все эти опции можно непосредственно применять ко всем вышеописанным типам соединений.

Пересекающиеся названия столбцов: ключевое слово suffixes

Вам может встретиться случай, когда в двух входных объектах присутствуют конфликтующие названия столбцов. Рассмотрим следующий пример:

```
In[17]: df8 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                            'rank': [1, 2, 3, 4]})
        df9 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                            'rank': [3, 1, 4, 2]})
        print(df8); print(df9); print(pd.merge(df8, df9, on="name"))
```

```
df8                df9                pd.merge(df8, df9, on="name")
    name  rank         name  rank        name  rank_x  rank_y
0   Bob      1      0   Bob     3     0   Bob       1       3
1  Jake      2      1  Jake     1     1  Jake       2       1
2  Lisa      3      2  Lisa     4     2  Lisa       3       4
3   Sue      4      3   Sue     2     3   Sue       4       2
```

Поскольку в результате должны были быть два конфликтующих названия столбцов, функция слияния автоматически добавила в названия суффиксы `_x` и `_y`, чтобы обеспечить уникальность названий столбцов результата. Если подобное поведение, принятое по умолчанию, неуместно, можно задать пользовательские суффиксы с помощью ключевого слова `suffixes`:

```
In[18]:
print(df8); print(df9);
print(pd.merge(df8, df9, on="name", suffixes=["_L", "_R"]))
```

```
df8                df9
    name  rank         name  rank
0   Bob      1      0   Bob     3
1  Jake      2      1  Jake     1
2  Lisa      3      2  Lisa     4
3   Sue      4      3   Sue     2

pd.merge(df8, df9, on="name", suffixes=["_L", "_R"])
   name  rank_L  rank_R
0   Bob       1       3
```

```
1  Jake       2       1
2  Lisa       3       4
3   Sue       4       2
```

Эти суффиксы будут работать для всех возможных вариантов соединений, в том числе и в случае нескольких пересекающихся по названию столбцов.

За более подробной информацией об этих вариантах загляните в раздел «Агрегирование и группировка» данной главы, в котором мы подробнее изучим реляционную алгебру, а также в раздел Merge, Join and Concatenate («Слияние, соединение и конкатенация», http://pandas.pydata.org/pandas-docs/stable/merging.html) документации библиотеки Pandas.

Пример: данные по штатам США

Операции слияния и соединения чаще всего оказываются нужны при объединении данных из различных источников. Здесь мы рассмотрим пример с определенными данными по штатам США и их населению. Файлы данных можно найти по адресу http://github.com/jakevdp/data-USstates/:

```
In[19]:
# Инструкции системного командного процессора для скачивания данных:
# !curl -O https://raw.githubusercontent.com/jakevdp/
#    data-USstates/master/state-population.csv
# !curl -O https://raw.githubusercontent.com/jakevdp/
#    data-USstates/master/state-areas.csv
# !curl -O https://raw.githubusercontent.com/jakevdp/
#    data-USstates/master/state-abbrevs.csv
```

Посмотрим на эти наборы данных с помощью функции `read_csv()` библиотеки Pandas:

```
In[20]: pop = pd.read_csv('state-population.csv')
        areas = pd.read_csv('state-areas.csv')
        abbrevs = pd.read_csv('state-abbrevs.csv')

        print(pop.head()); print(areas.head()); print(abbrevs.head())
```

```
pop.head()
  state/region     ages  year  population
0           AL  under18  2012   1117489.0
1           AL    total  2012   4817528.0
2           AL  under18  2010   1130966.0
3           AL    total  2010   4785570.0
4           AL  under18  2011   1125763.0

areas.head()
        state  area (sq. mi)
0     Alabama          52423
1      Alaska         656425
2     Arizona         114006
3    Arkansas          53182
3    Arkansas          53182
4  California         163707

abbrevs.head()
        state abbreviation
0     Alabama           AL
```

```
1      Alaska           AK
2     Arizona           AZ
3    Arkansas           AR
4  California           CA[1]
```

Допустим, нам нужно на основе этой информации отсортировать штаты и территорию США по плотности населения в 2010 году. Информации для этого у нас достаточно, но для достижения цели придется объединить наборы данных.

Начнем со слияния «многие-ко-многим», которое позволит получить полное имя штата в объекте `DataFrame` для населения. Выполнить слияние нужно на основе столбца `state/region` объекта `pop` и столбца `abbreviation` объекта `abbrevs`. Мы воспользуемся опцией `how='outer'`, чтобы гарантировать, что не упустим никаких данных из-за несовпадения меток.

```
In[21]: merged = pd.merge(pop, abbrevs, how='outer',
                          left_on='state/region', right_on='abbreviation')
        merged = merged.drop('abbreviation', 1) # Удаляем дублирующуюся
                                                # информацию
        merged.head()
Out[21]:   state/region     ages  year  population    state
         0           AL  under18  2012   1117489.0  Alabama
         1           AL    total  2012   4817528.0  Alabama
         2           AL  under18  2010   1130966.0  Alabama
         3           AL    total  2010   4785570.0  Alabama
         4           AL  under18  2011   1125763.0  Alabama
```

Следует проверить, не было ли каких-то несовпадений. Сделать это можно путем поиска строк с пустыми значениями:

```
In[22]: merged.isnull().any()

Out[22]: state/region    False
         ages            False
         year            False
         population       True
         state            True
         dtype: bool
```

Часть информации о населении отсутствует, выясним, какая именно:

```
In[23]: merged[merged['population'].isnull()].head()

Out[23]:       state/region     ages  year  population state
         2448           PR  under18  1990         NaN   NaN
         2449           PR    total  1990         NaN   NaN
         2450           PR    total  1991         NaN   NaN
         2451           PR  under18  1991         NaN   NaN
         2452           PR    total  1993         NaN   NaN
```

[1] Функция head() возвращает первые n строк набора данных. По умолчанию n = 5.

Похоже, что источник пустых значений по населению — Пуэрто-Рико, до 2000 года. Вероятно, это произошло из-за того, что необходимых данных не было в первоисточнике.

Мы видим, что некоторые из новых значений столбца `state` тоже пусты, а значит, в ключе объекта `abbrevs` отсутствовали соответствующие записи! Выясним, для каких территорий отсутствуют соответствующие значения:

```
In[24]: merged.loc[merged['state'].isnull(), 'state/region'].unique()

Out[24]: array(['PR', 'USA'], dtype=object)
```

Все понятно: наши данные по населению включают записи для Пуэрто-Рико (PR) и США в целом (USA), отсутствующие в ключе аббревиатур штатов. Исправим это, вставив соответствующие записи:

```
In[25]: merged.loc[merged['state/region'] == 'PR', 'state'] = 'Puerto Rico'
        merged.loc[merged['state/region'] == 'USA', 'state'] = 'United States'
        merged.isnull().any()

Out[25]: state/region    False
         ages            False
         year            False
         population       True
         state           False
         dtype: bool
```

В столбце `state` больше нет пустых значений. Готово!

Теперь можно слить результат с данными по площади штатов с помощью аналогичной процедуры. После изучения имеющихся результатов становится понятно, что нужно выполнить соединение по столбцу `state` в обоих объектах:

```
In[26]: final = pd.merge(merged, areas, on='state', how='left')
        final.head()

Out[26]:   state/region     ages  year  population    state  area (sq. mi)
         0           AL  under18  2012   1117489.0  Alabama        52423.0
         1           AL    total  2012   4817528.0  Alabama        52423.0
         2           AL  under18  2010   1130966.0  Alabama        52423.0
         3           AL    total  2010   4785570.0  Alabama        52423.0
         4           AL  under18  2011   1125763.0  Alabama        52423.0
```

Выполним снова проверку на пустые значения, чтобы узнать, были ли какие-то несовпадения:

```
In[27]: final.isnull().any()

Out[27]: state/region     False
         ages             False
         year             False
```

```
population        True
state            False
area (sq. mi)     True
dtype: bool
```

В столбце `area` имеются пустые значения. Посмотрим, какие территории не были учтены:

```
In[28]: final['state'][final['area (sq. mi)'].isnull()].unique()

Out[28]: array(['United States'], dtype=object)
```

Видим, что наш `DataFrame`-объект `areas` не содержит площадь США в целом. Мы могли бы вставить соответствующее значение (например, воспользовавшись суммой площадей всех штатов), но в данном случае мы просто удалим пустые значения, поскольку плотность населения США в целом нас сейчас не интересует:

```
In[29]: final.dropna(inplace=True)
        final.head()

Out[29]:   state/region     ages  year  population    state  area (sq. mi)
         0           AL  under18  2012   1117489.0  Alabama        52423.0
         1           AL    total  2012   4817528.0  Alabama        52423.0
         2           AL  under18  2010   1130966.0  Alabama        52423.0
         3           AL    total  2010   4785570.0  Alabama        52423.0
         4           AL  under18  2011   1125763.0  Alabama        52423.0
```

Теперь у нас есть все необходимые нам данные. Чтобы ответить на интересующий вопрос, сначала выберем часть данных, соответствующих 2010 году и всему населению. Воспользуемся функцией `query()` (для этого должен быть установлен пакет `numexpr`, см. раздел «Увеличение производительности библиотеки Pandas: eval() и query()» данной главы):

```
In[30]: data2010 = final.query("year == 2010 & ages == 'total'")
        data2010.head()

Out[30]:     state/region   ages  year  population       state  area (sq. mi)
         3              AL  total  2010   4785570.0     Alabama        52423.0
         91             AK  total  2010    713868.0      Alaska       656425.0
         101            AZ  total  2010   6408790.0     Arizona       114006.0
         189            AR  total  2010   2922280.0    Arkansas        53182.0
         197            CA  total  2010  37333601.0  California       163707.0
```

Теперь вычислим плотность населения и выведем данные в соответствующем порядке. Начнем с переиндексации наших данных по штату, после чего вычислим результат:

```
In[31]: data2010.set_index('state', inplace=True)
        density = data2010['population'] / data2010['area (sq. mi)']
```

```
In[32]: density.sort_values(ascending=False, inplace=True)
        density.head()

Out[32]: state
         District of Columbia    8898.897059
         Puerto Rico             1058.665149
         New Jersey              1009.253268
         Rhode Island             681.339159
         Connecticut              645.600649
         dtype: float64
```

Результат — список штатов США плюс Вашингтон (округ Колумбия) и Пуэрто-Рико, упорядоченный по плотности населения в 2010 году, в жителях на квадратную милю. Как видим, самая густонаселенная территория в этом наборе данных — Вашингтон (округ Колумбия); среди штатов же самый густонаселенный — Нью-Джерси.

Можно также вывести окончание списка:

```
In[33]: density.tail()

Out[33]: state
         South Dakota    10.583512
         North Dakota     9.537565
         Montana          6.736171
         Wyoming          5.768079
         Alaska           1.087509
         dtype: float64
```

Как видим, штатом с наименьшей плотностью населения, причем с большим отрывом от остальных, оказалась Аляска, насчитывающая в среднем одного жителя на квадратную милю.

Подобное громоздкое слияние данных — распространенная задача при ответе на вопросы, связанные с реальными источниками данных. Надеюсь, что этот пример дал вам представление, какими способами можно комбинировать вышеописанные инструменты, чтобы почерпнуть полезную информацию из данных!

Агрегирование и группировка

Важная часть анализа больших данных — их эффективное обобщение: вычисление сводных показателей, например `sum()`, `mean()`, `median()`, `min()` и `max()`, в которых одно число позволяет понять природу, возможно, огромного набора данных. В этом разделе мы займемся изучением сводных показателей в библиотеке Pandas, начиная с простых операций, подобных тем, с которыми мы уже имели дело при работе с массивами NumPy, и заканчивая более сложными операциями на основе понятия `groupby`.

Данные о планетах

Воспользуемся набором данных «Планеты» (Planets), доступным через пакет Seaborn (см. раздел «Визуализация с помощью библиотеки Seaborn» главы 4). Он включает информацию об открытых астрономами планетах, вращающихся вокруг других звезд, известных под названием *внесолнечных планет* или *экзопланет* (exoplanets). Скачать его можно с помощью команды пакета Seaborn:

```
In[2]: import seaborn as sns
       planets = sns.load_dataset('planets')
       planets.shape

Out[2]: (1035, 6)

In[3]: planets.head()

Out[3]:    method           number  orbital_period  mass   distance  year
        0  Radial Velocity  1       269.300         7.10   77.40     2006
        1  Radial Velocity  1       874.774         2.21   56.95     2008
        2  Radial Velocity  1       763.000         2.60   19.84     2011
        3  Radial Velocity  1       326.030         19.40  110.62    2007
        4  Radial Velocity  1       516.220         10.50  119.47    2009
```

Этот набор данных содержит определенную информацию о более чем 1000 экзопланет, открытых до 2014 года.

Простое агрегирование в библиотеке Pandas

Ранее мы рассмотрели некоторые доступные для массивов NumPy возможности по агрегированию данных (см. раздел «Агрегирование: минимум, максимум и все, что посередине» главы 2). Как и в случае одномерных массивов библиотеки NumPy, для объектов `Series` библиотеки Pandas агрегирующие функции возвращают скалярное значение:

```
In[4]: rng = np.random.RandomState(42)
       ser = pd.Series(rng.rand(5))
       ser

Out[4]: 0    0.374540
        1    0.950714
        2    0.731994
        3    0.598658
        4    0.156019
        dtype: float64

In[5]: ser.sum()

Out[5]: 2.8119254917081569
```

```
In[6]: ser.mean()

Out[6]: 0.56238509834163142
```

В случае объекта `DataFrame` по умолчанию агрегирующие функции возвращают сводные показатели по каждому столбцу:

```
In[7]: df = pd.DataFrame({'A': rng.rand(5),
                          'B': rng.rand(5)})
       df

Out[7]:           A         B
        0  0.155995  0.020584
        1  0.058084  0.969910
        2  0.866176  0.832443
        3  0.601115  0.212339
        4  0.708073  0.181825

In[8]: df.mean()

Out[8]: A    0.477888
        B    0.443420
        dtype: float64
```

Можно вместо этого агрегировать и по строкам, задав аргумент `axis`:

```
In[9]: df.mean(axis='columns')

Out[9]: 0    0.088290
        1    0.513997
        2    0.849309
        3    0.406727
        4    0.444949
        dtype: float64
```

Объекты `Series` и `DataFrame` библиотеки Pandas содержат методы, соответствующие всем упомянутым в разделе «Агрегирование: минимум, максимум и все, что посередине» главы 2 распространенным агрегирующим функциям. В них есть удобный метод `describe()`, вычисляющий сразу несколько самых распространенных сводных показателей для каждого столбца и возвращающий результат. Опробуем его на наборе данных «Планеты», пока удалив строки с отсутствующими значениями:

```
In[10]: planets.dropna().describe()

Out[10]:           number  orbital_period        mass    distance         year
         count  498.00000      498.000000  498.000000  498.000000   498.000000
         mean     1.73494      835.778671    2.509320   52.068213  2007.377510
         std      1.17572     1469.128259    3.636274   46.596041     4.167284
         min      1.00000        1.328300    0.003600    1.350000  1989.000000
         25%      1.00000       38.272250    0.212500   24.497500  2005.000000
```

```
50%      1.00000      357.000000    1.245000    39.940000   2009.000000
75%      2.00000      999.600000    2.867500    59.332500   2011.000000
max      6.00000    17337.500000   25.000000   354.000000   2014.000000
```

Эта возможность очень удобна для первоначального знакомства с общими характеристиками нашего набора данных. Например, мы видим в столбце `year`, что, хотя первая экзопланета была открыта еще в 1989 году, половина всех известных экзопланет открыта не ранее 2010 года. В значительной степени мы обязаны этим миссии *«Кеплер»*, представляющей собой космический телескоп, специально разработанный для поиска затмений от планет, вращающих вокруг других звезд.

В табл. 3.3 перечислены основные встроенные агрегирующие методы библиотеки Pandas.

Таблица 3.3. Список агрегирующих методов библиотеки Pandas

Агрегирующая функция	Описание
count()	Общее количество элементов
first(), last()	Первый и последний элементы
mean(), median()	Среднее значение и медиана
min(), max()	Минимум и максимум
std(), var()	Стандартное отклонение и дисперсия
mad()	Среднее абсолютное отклонение
prod()	Произведение всех элементов
sum()	Сумма всех элементов

Это все методы объектов `DataFrame` и `Series`.

Для более глубокого исследования данных простых сводных показателей часто недостаточно. Следующий уровень обобщения данных — операция `groupby`, позволяющая быстро и эффективно вычислять сводные показатели по подмножествам данных.

GroupBy: разбиение, применение, объединение

Простые агрегирующие функции дают возможность «прочувствовать» набор данных, но зачастую бывает нужно выполнить условное агрегирование по какой-либо метке или индексу. Это действие реализовано в так называемой операции `GroupBy`. Название group by («сгруппировать по») ведет начало от одноименной команды в языке SQL баз данных, но, возможно, будет понятнее говорить о ней в терминах, придуманных Хэдли Викхэмом, более известным своими библиотеками для языка R: *разбиение*, *применение* и *объединение*.

Разбиение, применение и объединение

Канонический пример операции «разбить, применить, объединить», в которой «применить» — обобщающее агрегирование, показан на рис. 3.1.

Рисунок 3.1 демонстрирует, что именно делает операция `GroupBy`.

- Шаг *разбиения* включает разделение на части и группировку объекта `DataFrame` на основе значений заданного ключа.
- Шаг *применения* включает вычисление какой-либо функции, обычно агрегирующей, преобразование или фильтрацию в пределах отдельных групп.
- На шаге *объединения* выполняется слияние результатов этих операций в выходной массив.

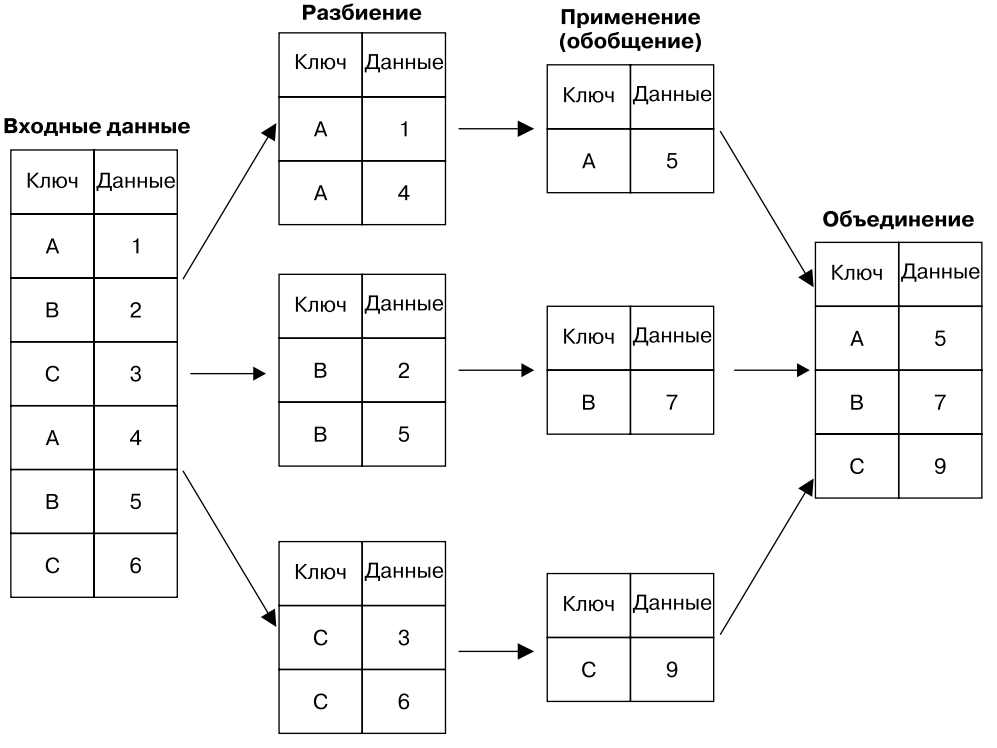

Рис. 3.1. Визуальное представление операции GroupBy

Хотя мы, конечно, могли бы сделать это вручную с помощью какого-либо сочетания описанных выше команд маскирования, агрегирования и слияния, важно понимать, что *не обязательно создавать объекты для промежуточных разбиений*. Операция `GroupBy` может проделать все это за один проход по данным, вычисляя сумму, среднее значение, количество, минимум и другие сводные показатели для

каждой группы. Мощь операции `GroupBy` состоит в абстрагировании этих шагов: пользователю не нужно заботиться о том, *как* фактически выполняются вычисления, а можно вместо этого думать об *операции в целом*.

В качестве примера рассмотрим использование библиотеки Pandas для выполнения показанных на рис. 3.1 вычислений. Начнем с создания входного объекта `DataFrame`:

```
In[11]: df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                           'data': range(6)}, columns=['key', 'data'])
        df

Out[11]:   key  data
         0   A     0
         1   B     1
         2   C     2
         3   A     3
         4   B     4
         5   C     5
```

Простейшую операцию «разбить, применить, объединить» можно реализовать с помощью метода `groupby()` объекта `DataFrame`, передав в него имя желаемого ключевого столбца:

```
In[12]: df.groupby('key')

Out[12]: <pandas.core.groupby.DataFrameGroupBy object at 0x117272160>
```

Обратите внимание, что возвращаемое — не набор объектов `DataFrame`, а объект `DataFrameGroupBy`. Этот объект особенный, его можно рассматривать как специальное представление объекта `DataFrame`, готовое к группировке, но не выполняющее никаких фактических вычислений до этапа применения агрегирования. Подобный метод «отложенного вычисления» означает возможность очень эффективной реализации распространенных агрегирующих функций, причем практически прозрачным для пользователя образом.

Для получения результата можно вызвать один из агрегирующих методов этого объекта `DataFrameGroupBy`, что приведет к выполнению соответствующих шагов применения/объединения:

```
In[13]: df.groupby('key').sum()

Out[13]:      data
         key
         A       3
         B       5
         C       7
```

Метод `sum()` — лишь один из возможных вариантов в этой команде. Здесь можно использовать практически любую распространенную агрегирующую функцию библиотек Pandas или NumPy, равно как и практически любую корректную операцию объекта `DataFrame`.

Объект GroupBy

Объект `GroupBy` — очень гибкая абстракция. Во многом с ним можно обращаться как с коллекцией объектов `DataFrame`, и вся сложность будет скрыта от пользователя. Рассмотрим примеры на основе набора данных «Планеты».

Вероятно, самые важные из доступных благодаря объекту `GroupBy` операций — *агрегирование*, *фильтрация*, *преобразование* и *применение*. Мы обсудим каждую из них более подробно в пункте «Агрегирование, фильтрация, преобразование, применение» данного подраздела, но сначала познакомимся с другой функциональностью, которую можно использовать вместе с базовой операцией `GroupBy`.

Индексация по столбцам. Объект `GroupBy` поддерживает индексацию по столбцам аналогично объекту `DataFrame`, с возвратом модифицированного объекта `GroupBy`. Например:

```
In[14]: planets.groupby('method')

Out[14]: <pandas.core.groupby.DataFrameGroupBy object at 0x1172727b8>

In[15]: planets.groupby('method')['orbital_period']

Out[15]: <pandas.core.groupby.SeriesGroupBy object at 0x117272da0>
```

Здесь мы выбрали конкретную группу `Series` из исходной группы `DataFrame`, сославшись на соответствующее имя столбца. Как и в случае с объектом `GroupBy`, никаких вычислений не происходит до вызова для этого объекта какого-нибудь агрегирующего метода:

```
In[16]: planets.groupby('method')['orbital_period'].median()

Out[16]: method
         Astrometry                     631.180000
         Eclipse Timing Variations     4343.500000
         Imaging                      27500.000000
         Microlensing                  3300.000000
         Orbital Brightness Modulation    0.342887
         Pulsar Timing                   66.541900
         Pulsation Timing Variations   1170.000000
         Radial Velocity                360.200000
         Transit                          5.714932
         Transit Timing Variations       57.011000
         Name: orbital_period, dtype: float64
```

Результат дает нам общее представление о масштабе чувствительности каждого из методов к периодам обращения (в днях).

Цикл по группам. Объект `GroupBy` поддерживает непосредственное выполнение циклов по группам с возвратом каждой группы в виде объекта `Series` или `DataFrame`:

```
In[17]: for (method, group) in planets.groupby('method'):
            print("{0:30s} shape={1}".format(method, group.shape))

Astrometry                     shape=(2, 6)
Eclipse Timing Variations      shape=(9, 6)
Imaging                        shape=(38, 6)
Microlensing                   shape=(23, 6)
Orbital Brightness Modulation  shape=(3, 6)
Pulsar Timing                  shape=(5, 6)
Pulsation Timing Variations    shape=(1, 6)
Radial Velocity                shape=(553, 6)
Transit                        shape=(397, 6)
Transit Timing Variations      shape=(4, 6)
```

Это может пригодиться для выполнения некоторых вещей вручную, хотя обычно быстрее воспользоваться встроенной функциональностью `apply`.

Методы диспетчеризации. Благодаря определенной магии классов языка Python все методы, не реализованные явным образом объектом `GroupBy`, будут передаваться далее и выполняться для групп, вне зависимости от того, являются ли они объектами `Series` или `DataFrame`. Например, можно использовать метод `describe()` объекта `DataFrame` для вычисления набора сводных показателей, описывающих каждую группу в данных:

```
In[18]: planets.groupby('method')['year'].describe().unstack()

Out[18]:
                              count         mean       std     min      25%
\\
method
Astrometry                     2.0  2011.500000  2.121320  2010.0  2010.75
Eclipse Timing Variations      9.0  2010.000000  1.414214  2008.0  2009.00
Imaging                       38.0  2009.131579  2.781901  2004.0  2008.00
Microlensing                  23.0  2009.782609  2.859697  2004.0  2008.00
Orbital Brightness Modulation  3.0  2011.666667  1.154701  2011.0  2011.00
Pulsar Timing                  5.0  1998.400000  8.384510  1992.0  1992.00
Pulsation Timing Variations    1.0  2007.000000       NaN  2007.0  2007.00
Radial Velocity              553.0  2007.518987  4.249052  1989.0  2005.00
Transit                      397.0  2011.236776  2.077867  2002.0  2010.00
Transit Timing Variations      4.0  2012.500000  1.290994  2011.0  2011.75
                                50%      75%     max
method
Astrometry                   2011.5  2012.25  2013.0
Eclipse Timing Variations    2010.0  2011.00  2012.0
Imaging                      2009.0  2011.00  2013.0
Microlensing                 2010.0  2012.00  2013.0
Orbital Brightness Modulation 2011.0  2012.00  2013.0
Pulsar Timing                1994.0  2003.00  2011.0
Pulsation Timing Variations  2007.0  2007.00  2007.0
Radial Velocity              2009.0  2011.00  2014.0
Transit                      2012.0  2013.00  2014.0
Transit Timing Variations    2012.5  2013.25  2014.0
```

Эта таблица позволяет получить лучшее представление о наших данных. Например, большинство планет было открыто методом измерения лучевой скорости (radial velocity method) и транзитным методом (transit method), хотя последний стал распространенным благодаря новым более точным телескопам только в последнее десятилетие. Похоже, что новейшими методами являются метод вариации времени транзитов (transit timing variation method) и метод модуляции орбитальной яркости (orbital brightness modulation method), которые до 2011 года не использовались для открытия новых планет.

Это всего лишь один пример полезности методов диспетчеризации. Обратите внимание, что они применяются к *каждой отдельной группе*, после чего результаты объединяются в объект `GroupBy` и возвращаются. Можно использовать для соответствующего объекта `GroupBy` любой допустимый метод объектов `Series`/`DataFrame`, что позволяет выполнять многие весьма гибкие и мощные операции!

Агрегирование, фильтрация, преобразование, применение

Предыдущее обсуждение касалось агрегирования применительно к операции объединения, но доступны и другие возможности. В частности, у объектов `GroupBy` имеются методы `aggregate()`, `filter()`, `transform()` и `apply()`, эффективно выполняющие множество полезных операций до объединения сгруппированных данных.

В следующих подразделах мы будем использовать объект `DataFrame`:

```
In[19]: rng = np.random.RandomState(0)
        df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                           'data1': range(6),
                           'data2': rng.randint(0, 10, 6)},
                           columns = ['key', 'data1', 'data2'])
        df

Out[19]:   key  data1  data2
         0   A      0      5
         1   B      1      0
         2   C      2      3
         3   A      3      3
         4   B      4      7
         5   C      5      9
```

Агрегирование. Мы уже знакомы со сводными показателями объекта `GroupBy`, вычисляемыми с помощью методов `sum()`, `median()` и т. п., но метод `aggregate()` обеспечивает еще большую гибкость. Он может принимать на входе строку, функцию или список и вычислять все сводные показатели сразу. Вот пример, включающий все вышеупомянутое:

```
In[20]: df.groupby('key').aggregate(['min', np.median, max])

Out[20]:       data1            data2
```

```
        min median max   min median max
key
A         0    1.5   3     3    4.0   5
B         1    2.5   4     0    3.5   7
C         2    3.5   5     3    6.0   9
```

Еще один удобный паттерн — передача в него словаря, связывающего имена столбцов с операциями, которые должны быть применены к этим столбцам:

```
In[21]: df.groupby('key').aggregate({'data1': 'min',
                                     'data2': 'max'})
Out[21]:      data1  data2
        key
        A         0      5
        B         1      7
        C         2      9
```

Фильтрация. Операция фильтрации дает возможность опускать данные в зависимости от свойств группы. Например, нам может понадобиться оставить в результате все группы, в которых стандартное отклонение превышает какое-либо критическое значение:

```
In[22]:
def filter_func(x):
    return x['data2'].std() > 4

print(df); print(df.groupby('key').std());
print(df.groupby('key').filter(filter_func))
```

```
df                      df.groupby('key').std()
   key  data1  data2    key      data1     data2
0   A      0      5     A      2.12132  1.414214
1   B      1      0     B      2.12132  4.949747
2   C      2      3     C      2.12132  4.242641
3   A      3      3
4   B      4      7
5   C      5      9

df.groupby('key').filter(filter_func)
  key  data1  data2
1   B      1      0
2   C      2      3
4   B      4      7
5   C      5      9
```

Функция `filter()` возвращает булево значение, определяющее, прошла ли группа фильтрацию. В данном случае, поскольку стандартное отклонение группы A превышает 4, она удаляется из итогового результата.

Преобразование. В то время как агрегирующая функция должна возвращать сокращенную версию данных, преобразование может вернуть версию полного

набора данных, преобразованную ради дальнейшей их перекомпоновки. При подобном преобразовании форма выходных данных совпадает с формой входных. Распространенный пример — центрирование данных путем вычитания среднего значения по группам:

```
In[23]: df.groupby('key').transform(lambda x: x - x.mean())

Out[23]:   data1  data2
         0   -1.5    1.0
         1   -1.5   -3.5
         2   -1.5   -3.0
         3    1.5   -1.0
         4    1.5    3.5
         5    1.5    3.0
```

Метод apply(). Метод `apply()` позволяет применять произвольную функцию к результатам группировки. В качестве параметра эта функция должна получать объект `DataFrame`, а возвращать или объект библиотеки Pandas (например, `DataFrame`, `Series`), или скалярное значение, в зависимости от возвращаемого значения будет вызвана соответствующая операция объединения.

Например, функция `apply()`, нормирующая первый столбец на сумму значений второго:

```
In[24]: def norm_by_data2(x):
            # x – объект DataFrame сгруппированных значений
            x['data1'] /= x['data2'].sum()
            return x
        print(df); print(df.groupby('key').apply(norm_by_data2))
```

```
df                    df.groupby('key').apply(norm_by_data2)
  key  data1  data2       key     data1  data2
0   A      0      5     0   A  0.000000      5
1   B      1      0     1   B  0.142857      0
2   C      2      3     2   C  0.166667      3
3   A      3      3     3   A  0.375000      3
4   B      4      7     4   B  0.571429      7
5   C      5      9     5   C  0.416667      9
```

Функция `apply()` в `GroupBy` достаточно гибка. Единственное требование, чтобы она принимала на входе объект `DataFrame` и возвращала объект библиотеки Pandas или скалярное значение; что вы делаете внутри, остается на ваше усмотрение!

Задание ключа разбиения

В представленных ранее простых примерах мы разбивали объект `DataFrame` по одному столбцу. Это лишь один из многих вариантов задания принципа формирования групп, и мы сейчас рассмотрим некоторые другие возможности.

Список, массив, объект Series и индекс как ключи группировки. Ключ может быть любым рядом или списком такой же длины, как и у объекта `DataFrame`. Например:

```
In[25]: L = [0, 1, 0, 1, 2, 0]
print(df); print(df.groupby(L).sum())
```

```
df                    df.groupby(L).sum()
  key  data1  data2         data1  data2
0   A      0      5     0      7     17
1   B      1      0     1      4      3
2   C      2      3     2      4      7
3   A      3      3
4   B      4      7
5   C      5      9
```

Разумеется, это значит, что есть еще один, несколько более длинный способ выполнить вышеприведенную операцию `df.groupby('key')`:

```
In[26]: print(df); print(df.groupby(df['key']).sum())
```

```
df                        df.groupby(df['key']).sum()
  key  data1  data2             data1  data2
0   A      0      5       A         3      8
1   B      1      0       B         5      7
2   C      2      3       C         7     12
3   A      3      3
4   B      4      7
5   C      5      9
```

Словарь или объект Series, связывающий индекс и группу. Еще один метод: указать словарь, задающий соответствие значений индеякса и ключей группировки:

```
In[27]: df2 = df.set_index('key')
        mapping = {'A': 'vowel', 'B': 'consonant', 'C': 'consonant'}
        print(df2); print(df2.groupby(mapping).sum())
```

```
df2                       df2.groupby(mapping).sum()
key  data1  data2                    data1  data2
A        0      5         consonant     12     19
B        1      0         vowel          3      8
C        2      3
A        3      3
B        4      7
C        5      9
```

Любая функция языка Python. Аналогично заданию соответствия можно передать функции `groupby` любую функцию, принимающую на входе значение индекса и возвращающую группу:

```
In[28]: print(df2); print(df2.groupby(str.lower).mean())

df2                      df2.groupby(str.lower).mean()
key  data1  data2            data1  data2
A        0      5         a    1.5    4.0
B        1      0         b    2.5    3.5
C        2      3         c    3.5    6.0
A        3      3
B        4      7
C        5      9
```

Список допустимых ключей. Можно комбинировать любые из предыдущих вариантов ключей для группировки по мультииндексу:

```
In[29]: df2.groupby([str.lower, mapping]).mean()

Out[29]:              data1  data2
         a vowel        1.5    4.0
         b consonant    2.5    3.5
         c consonant    3.5    6.0
```

Пример группировки

В качестве примера соберем все это вместе в нескольких строках кода на языке Python и подсчитаем количество открытых планет по методу открытия и десятилетию:

```
In[30]: decade = 10 * (planets['year'] // 10)
        decade = decade.astype(str) + 's'
        decade.name = 'decade'
        planets.groupby(['method', decade])
                        ['number'].sum().unstack().fillna(0)

Out[30]: decade                          1980s  1990s  2000s  2010s
         method
         Astrometry                        0.0    0.0    0.0    2.0
         Eclipse Timing Variations         0.0    0.0    5.0   10.0
         Imaging                           0.0    0.0   29.0   21.0
         Microlensing                      0.0    0.0   12.0   15.0
         Orbital Brightness Modulation     0.0    0.0    0.0    5.0
         Pulsar Timing                     0.0    9.0    1.0    1.0
         Pulsation Timing Variations       0.0    0.0    1.0    0.0
         Radial Velocity                   1.0   52.0  475.0  424.0
         Transit                           0.0    0.0   64.0  712.0
         Transit Timing Variations         0.0    0.0    0.0    9.0
```

Это демонстрирует возможности комбинирования нескольких из вышеописанных операций применительно к реальным наборам данных. Мы мгновенно получили представление о том, когда и как открывались экзопланеты в последние несколько десятилетий!

Теперь же я предложил бы углубиться в эти несколько строк кода и выполнить их пошагово, чтобы убедиться, что вы действительно понимаете, какой вклад в результат они вносят. Это в чем-то непростой пример, но благодаря хорошему пониманию элементов кода у вас появятся средства для исследования ваших собственных данных.

Сводные таблицы

Мы уже видели возможности по исследованию отношений в наборе данных, предоставляемые абстракцией `GroupBy`. *Сводная таблица* (pivot table) — схожая операция, часто встречающаяся в электронных таблицах и других программах, работающих с табличными данными. Сводная таблица получает на входе простые данные в виде столбцов и группирует записи в двумерную таблицу, обеспечивающую многомерное представление данных. Различие между сводными таблицами и операцией `GroupBy` иногда неочевидно. Лично мне помогает представлять сводные таблицы как многомерную версию агрегирующей функции `GroupBy`. То есть вы выполняете операцию «разбить, применить, объединить», но как разбиение, так и объединение происходят не на одномерном индексе, а на двумерной координатной сетке.

Данные для примеров работы со сводными таблицами

Для примеров из этого раздела мы воспользуемся базой данных пассажиров парохода *«Титаник»*, доступной через библиотеку Seaborn (см. раздел «Визуализация с помощью библиотеки Seaborn» главы 4):

```
In[1]: import numpy as np
       import pandas as pd
       import seaborn as sns
       titanic = sns.load_dataset('titanic')

In[2]: titanic.head()

Out[2]:
   survived  pclass     sex   age  sibsp  parch     fare embarked  class  \\
0         0       3    male  22.0      1      0   7.2500        S  Third
1         1       1  female  38.0      1      0  71.2833        C  First
2         1       3  female  26.0      0      0   7.9250        S  Third
3         1       1  female  35.0      1      0  53.1000        S  First
4         0       3    male  35.0      0      0   8.0500        S  Third

     who adult_male deck  embark_town alive  alone
0    man       True  NaN  Southampton    no  False
1  woman      False    C    Cherbourg   yes  False
2  woman      False  NaN  Southampton   yes   True
```

```
3  woman       False    C  Southampton   yes  False
4    man        True  NaN  Southampton    no   True
```

Этот набор данных содержит информацию о каждом пассажире злополучного рейса, включая пол, возраст, класс, стоимость билета и многое другое.

Сводные таблицы «вручную»

Чтобы узнать о данных больше, можно начать с группировки пассажиров по полу, информации о том, выжил ли пассажир, или какой-то их комбинации. Если вы читали предыдущий раздел, то можете воспользоваться операцией `GroupBy`. Например, посмотрим на коэффициент выживаемости в зависимости от пола:

```
In[3]: titanic.groupby('sex')[['survived']].mean()

Out[3]:        survived
       sex
       female  0.742038
       male    0.188908
```

Это сразу же дает нам некоторое представление о наборе данных: в целом, три четверти находившихся на борту женщин выжило, в то время как из мужчин выжил только каждый пятый!

Однако хотелось бы заглянуть немного глубже и увидеть распределение выживших по полу и классу. Говоря языком `GroupBy`, можно было бы идти следующим путем: *сгруппировать по* классу и полу, *выбрать* выживших, *применить* агрегирующую функцию среднего значения, *объединить* получившиеся группы, после чего *выполнить операцию unstack* иерархического индекса, чтобы обнажить скрытую многомерность. В виде кода:

```
In[4]: titanic.groupby(['sex', 'class'])
                      ['survived'].aggregate('mean').unstack()

Out[4]: class      First    Second     Third
        sex
        female  0.968085  0.921053  0.500000
        male    0.368852  0.157407  0.135447
```

Это дает нам лучшее представление о том, как пол и класс влияли на выживаемость, но код начинает выглядеть несколько запутанным. Хотя каждый шаг этого конвейера представляется вполне осмысленным в свете ранее рассмотренных инструментов, такая длинная строка кода не особо удобна для чтения или использования. Двумерный `GroupBy` встречается настолько часто, что в состав библиотеки Pandas был включен удобный метод, `pivot_table`, позволяющий описывать более кратко данную разновидность многомерного агрегирования.

Синтаксис сводных таблиц

Вот эквивалентный вышеприведенной операции код, использующий метод `pivot_table` объекта `DataFrame`:

```
In[5]: titanic.pivot_table('survived', index='sex', columns='class')

Out[5]: class      First    Second     Third
        sex
        female  0.968085  0.921053  0.500000
        male    0.368852  0.157407  0.135447
```

Такая запись несравненно удобнее для чтения, чем подход с `GroupBy`, при том же результате. Как и можно было ожидать от трансатлантического круиза начала XX века, судьба благоприятствовала женщинам и первому классу. Женщины из первого класса выжили практически все (привет, Роуз!), из мужчин третьего класса выжила только десятая часть (извини, Джек!).

Многоуровневые сводные таблицы

Группировку в сводных таблицах, как и при операции `GroupBy`, можно задавать на нескольких уровнях и с множеством параметров. Например, интересно взглянуть на возраст в качестве третьего измерения. Разобьем данные на интервалы по возрасту с помощью функции `pd.cut`:

```
In[6]: age = pd.cut(titanic['age'], [0, 18, 80])
       titanic.pivot_table('survived', ['sex', age], 'class')

Out[6]:   class               First    Second     Third
          sex    age
          female (0, 18]   0.909091  1.000000  0.511628
                 (18, 80]  0.972973  0.900000  0.423729
          male   (0, 18]   0.800000  0.600000  0.215686
                 (18, 80]  0.375000  0.071429  0.133663
```

Мы можем применить ту же стратегию при работе со столбцами. Добавим сюда информацию о стоимости билета, воспользовавшись функцией `pd.qcut` для автоматического вычисления квантилей:

```
In[7]: fare = pd.qcut(titanic['fare'], 2)
       titanic.pivot_table('survived', ['sex', age], [fare, 'class'])

Out[7]:
fare            [0, 14.454]
class                 First    Second     Third      \\
sex    age
female (0, 18]          NaN  1.000000  0.714286
       (18, 80]         NaN  0.880000  0.444444
male   (0, 18]          NaN  0.000000  0.260870
       (18, 80]         0.0  0.098039  0.125000
```

```
fare            (14.454, 512.329]
class                   First    Second     Third
sex    age
female (0, 18]      0.909091  1.000000  0.318182
       (18, 80]     0.972973  0.914286  0.391304
male   (0, 18]      0.800000  0.818182  0.178571
       (18, 80]     0.391304  0.030303  0.192308
```

Результат представляет собой четырехмерную сводную таблицу с иерархическими индексами (см. раздел «Иерархическая индексация» данной главы), выведенную в демонстрирующей отношения между значениями сетке.

Дополнительные параметры сводных таблиц

Полная сигнатура вызова метода `pivot_table` объектов `DataFrame` выглядит следующим образом:

```
# сигнатура вызова в версии 0.18[1] библиотеки Pandas
DataFrame.pivot_table(data, values=None, index=None, columns=None,
                      aggfunc='mean', fill_value=None, margins=False,
                      dropna=True, margins_name='All')
```

Мы уже видели примеры первых трех аргументов, в данном подразделе рассмотрим остальные. Два из параметров, `fill_value` и `dropna`, относятся к пропущенным значениям и интуитивно понятны, примеры их использования мы приводить не будем.

Ключевое слово `aggfunc` управляет тем, какой тип агрегирования применяется, по умолчанию это среднее значение. Как и в `GroupBy`, спецификация агрегирующей функции может быть строкой с одним из нескольких обычных вариантов (`'sum'`, `'mean'`, `'count'`, `'min'`, `'max'` и т. д.) или функцией, реализующей агрегирование (`np.sum()`, `min()`, `sum()` и т. п.). Кроме того, агрегирование может быть задано в виде словаря, связывающего столбец с любым из вышеперечисленных вариантов:

```
In[8]: titanic.pivot_table(index='sex', columns='class',
                           aggfunc={'survived':sum, 'fare':'mean'})
Out[8]:              fare                            survived
       class         First     Second      Third    First Second Third
       sex
       female  106.125798  21.970121  16.118810     91.0   70.0  72.0
       male     67.226127  19.741782  12.661633     45.0   17.0  47.0
```

Обратите внимание, что мы опустили ключевое слово `values`, при задании `aggfunc` происходит автоматическое определение.

---

[1] На момент выпуска версии 0.19.2 данная сигнатура осталась прежней.

Иногда бывает полезно вычислять итоги по каждой группе. Это можно сделать с помощью ключевого слова `margins`:

```
In[9]: titanic.pivot_table('survived', index='sex', columns='class',
                           margins=True)

Out[9]: class      First    Second     Third       All
        sex
        female  0.968085  0.921053  0.500000  0.742038
        male    0.368852  0.157407  0.135447  0.188908
        All     0.629630  0.472826  0.242363  0.383838
```

Такие итоги автоматически дают нам информацию о выживаемости вне зависимости от класса, коэффициенте выживаемости по классу вне зависимости от пола и общем коэффициенте выживаемости 38 %. Метки для этих итогов можно задать с помощью ключевого слова `margins_name`, по умолчанию имеющего значение `"All"`.

Пример: данные о рождаемости

В качестве примера взглянем на находящиеся в открытом доступе данные о рождаемости в США, предоставляемые центрами по контролю заболеваний (Centers for Disease Control, CDC). Данные можно найти по адресу https://raw.githubusercontent.com/jakevdp/data-CDCbirths/master/births.csv (этот набор данных довольно широко исследовался Эндрю Гелманом и его группой (см., например, сообщение в блоге http://bit.ly/2fZzW8K)):

```
In[10]:
# Инструкция системного командного процессора для скачивания данных:
# !curl -O https://raw.githubusercontent.com/jakevdp/data-CDCbirths/
# master/births.csv

In[11]: births = pd.read_csv('births.csv')
```

Посмотрев на эти данные, мы обнаружим их относительную простоту — они содержат количество новорожденных, сгруппированных по дате и полу:

```
In[12]: births.head()

Out[12]:    year  month day gender  births
         0  1969      1   1      F    4046
         1  1969      1   1      M    4440
         2  1969      1   2      F    4454
         3  1969      1   2      M    4548
         4  1969      1   3      F    4548
```

Мы начнем понимать эти данные немного лучше, воспользовавшись сводной таблицей. Добавим в них столбец для десятилетия и взглянем на рождения девочек и мальчиков как функцию от десятилетия:

```
In[13]:
births['decade'] = 10 * (births['year'] // 10)
births.pivot_table('births', index='decade', columns='gender', aggfunc='sum')

Out[13]: gender         F         M          decade
         1960     1753634   1846572
         1970    16263075  17121550
         1980    18310351  19243452
         1990    19479454  20420553
         2000    18229309  19106428
```

Сразу же видим, что в каждом десятилетии мальчиков рождается больше, чем девочек. Воспользуемся встроенными средствами построения графиков библиотеки Pandas для визуализации общего количества новорожденных в зависимости от года (рис. 3.2; см. обсуждение построения графиков с помощью библиотеки Matplotlib в главе 4):

```
In[14]:
%matplotlib inline
import matplotlib.pyplot as plt
sns.set()  # Используем стили библиотеки Seaborn
births.pivot_table('births', index='year', columns='gender',
                   aggfunc='sum').plot()
plt.ylabel('total births per year');     # общее количество новорожденных
                                         # в течение года
```

Благодаря сводной таблице и методу `plot()` мы можем сразу же увидеть ежегодный тренд новорожденных по полу. В последние 50 с лишним лет мальчиков рождалось больше, чем девочек, примерно на 5%.

Дальнейшее исследование данных. Хотя это, возможно, и не имеет отношения к сводным таблицам, есть еще несколько интересных вещей, которые можно извлечь из этого набора данных с помощью уже рассмотренных инструментов библиотеки Pandas. Нам придется начать с небольшой очистки данных, удалив аномальные значения, возникшие из-за неправильно набранных дат (например, 31 июня) или отсутствующих значений (например, 99 июня). Простой способ убрать сразу их все — отсечь аномальные значения. Мы сделаем это с помощью надежного алгоритма *сигма-отсечения* (sigma-clipping)[1]:

```
In[15]: quartiles = np.percentile(births['births'], [25, 50, 75])
        mu = quartiles[1]
        sig = 0.74 * (quartiles[2] - quartiles[0])
```

[1] Узнать больше об операциях сигма-отсечения вы можете из книги, которую я написал совместно с Желько Ивечичем, Эндрю Дж. Конолли и Александром Греем: Statistics, Data Mining and Machine Learning in Astronomy: A Practical Python Guide for the Analysis of Survey Data (Princeton University Press, 2014).

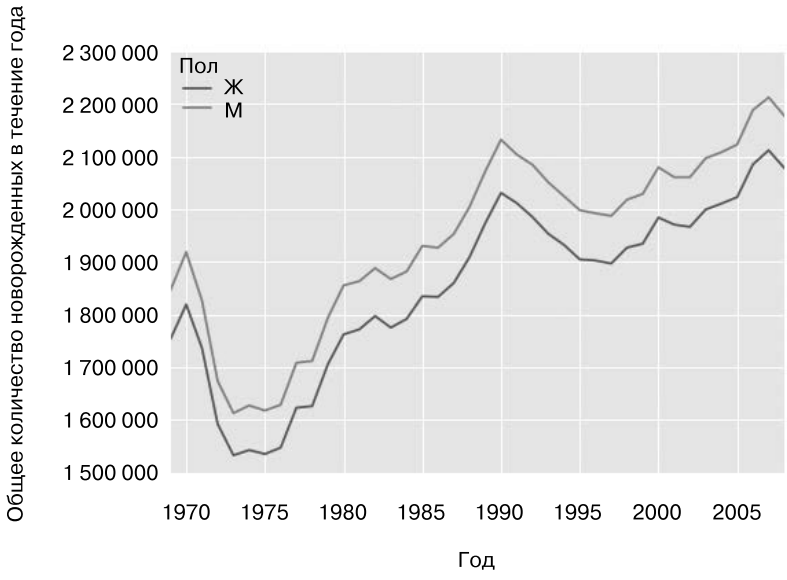

Рис. 3.2. Общее количество новорожденных в США в зависимости от года и пола

Последняя строка представляет собой грубую оценку среднего значения выборки, в котором 0.74 — межквартильный размах Гауссового распределения. Теперь можно воспользоваться методом `query()` (обсуждаемым далее в разделе «Увеличение производительности библиотеки Pandas: eval() и query()» этой главы) для фильтрации строк, в которых количество новорожденных выходит за пределы этих значений:

```
In[16]:
births = births.query('(births > @mu - 5 * @sig) & (births < @mu + 5 * @sig)')
```

Далее мы устанавливаем целочисленный тип столбца для `day`. Ранее он был строчным, поскольку некоторые столбцы в наборе данных содержат значение `'null'`:

```
In[17]: # делаем тип столбца 'day' целочисленным;
# изначально он был строчным из-за пустых значений
        births['day'] = births['day'].astype(int)
```

Наконец, мы можем создать индекс для даты, объединив день, месяц и год (см. «Работа с временными рядами» этой главы). Это даст нам возможность быстро вычислять день недели для каждой строки:

```
In[18]: # создаем индекс для даты из года, месяца и дня
        births.index = pd.to_datetime(10000 * births.year +
                                      100 * births.month +
                                      births.day, format='%Y%m%d')

        births['dayofweek'] = births.index.dayofweek  # День недели
```

С помощью этого можно построить график дней рождения в зависимости от дня недели за несколько десятилетий (рис. 3.3):

```
In[19]:
import matplotlib.pyplot as plt
import matplotlib as mpl

births.pivot_table('births', index='dayofweek',
                   columns='decade', aggfunc='mean').plot()
plt.gca().set_xticklabels(['Mon', 'Tues', 'Wed', 'Thurs', 'Fri', 'Sat',
'Sun'])
plt.ylabel('mean births by day'); # среднее количество новорожденных в день
```

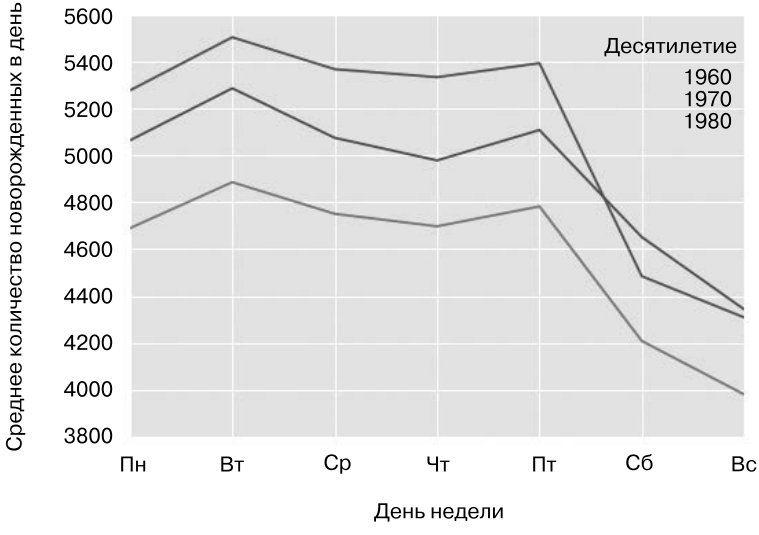

Рис. 3.3. Среднее количество новорожденных за день в зависимости от дня недели и десятилетия

Становится очевидно, что в выходные происходит меньше рождений, чем в будние дни! Обратите внимание, что 1990-е и 2000-е годы отсутствуют на графике, поскольку начиная с 1989 года данные CDC содержат только месяц рождения.

Еще одно интересное представление этих данных можно получить, построив график рождений в зависимости от дня *года*. Сначала сгруппируем данные отдельно по месяцу и дню:

```
In[20]:
births_by_date = births.pivot_table('births',
                                    [births.index.month, births.index.day])
births_by_date.head()

Out[20]: 1  1    4009.225
            2    4247.400
```

```
3    4500.900
4    4571.350
5    4603.625
Name: births, dtype: float64
```

Результат представляет собой мультииндекс по месяцам и дням. Чтобы упростить построение графика, преобразуем эти месяцы и дни в даты путем связывания их с фиктивным годом (обязательно выберите високосный год, чтобы обработать 29 февраля корректным образом!)

```
In[21]: births_by_date.index = [pd.datetime(2012, month, day)
                                for (month, day) in births_by_date.index]
        births_by_date.head()

Out[21]: 2012-01-01    4009.225
         2012-01-02    4247.400
         2012-01-03    4500.900
         2012-01-04    4571.350
         2012-01-05    4603.625
         Name: births, dtype: float64
```

Если смотреть только на месяц и день, то мы получаем временной ряд, отражающий среднее количество новорожденных в зависимости от дня года. Исходя из этого, мы можем построить с помощью метода `plot` график данных (рис. 3.4). В нем мы обнаруживаем некоторые любопытные тренды:

```
In[22]: # Строим график результатов
        fig, ax = plt.subplots(figsize=(12, 4))
        births_by_date.plot(ax=ax);
```

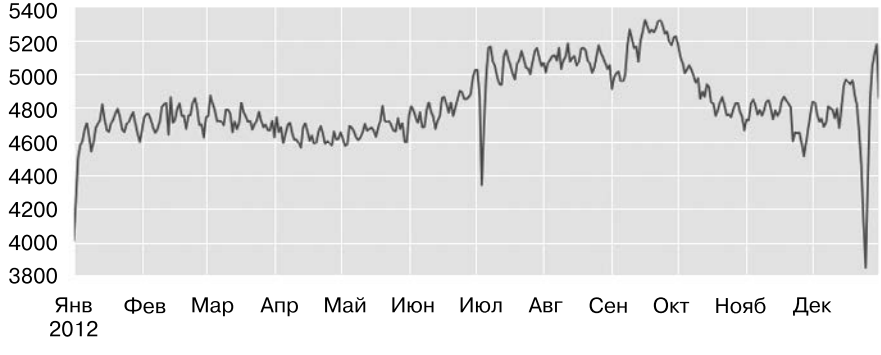

Рис. 3.4. Среднее количество новорожденных в зависимости от месяца

В частности, на графике удивляет резкое падение количества рождений в государственные праздники США (например, День независимости, День труда, День благодарения, Рождество, Новый год). Хотя оно отражает скорее тенденции, относящиеся к заранее запланированным/искусственным родам, а не глубокое

психосоматическое влияние на естественные роды. Дальнейшее обсуждение данной тенденции, ее анализ и ссылки на эту тему смотрите в сообщении по адресу http://bit.ly/2fZzW8K из блога Эндрю Гелмана. Мы вернемся к этому графику в подразделе «Пример: влияние выходных на рождения детей в США» раздела «Текст и поясняющие надписи» главы 4, в котором воспользуемся инструментами библиотеки Matplotlib для добавления меток на график.

Глядя на этот краткий пример, вы могли заметить, что многие из рассмотренных нами инструментов языка Python и библиотеки Pandas можно комбинировать между собой и использовать, чтобы почерпнуть полезную информацию из множества наборов данных. Более сложные манипуляции над данными мы увидим в следующих разделах!

Векторизованные операции над строками

Одна из сильных сторон языка Python — относительное удобство работы в нем со строковыми данными и манипуляций ими. Библиотека Pandas вносит в это свою лепту и предоставляет набор *векторизованных операций над строками*, ставших существенной частью очистки данных, необходимой при работе с реальными данными. В этом разделе мы изучим некоторые строковые операции библиотеки Pandas, после чего рассмотрим их использование для частичной очистки очень зашумленного набора данных рецептов, собранных в Интернете.

Знакомство со строковыми операциями библиотеки Pandas

В предыдущем разделе мы видели, как обобщают арифметические операции такие инструменты, как библиотека NumPy и библиотека Pandas, позволяя легко и быстро выполнять одну и ту же операцию над множеством элементов массива. Например:

```
In[1]: import numpy as np
       x = np.array([2, 3, 5, 7, 11, 13])
       x * 2

Out[1]: array([ 4,  6, 10, 14, 22, 26])
```

Векторизация операций упрощает синтаксис работы с массивами данных: больше нет необходимости беспокоиться о размере или форме массива, а только о нужной нам операции. Библиотека NumPy не предоставляет такого простого способа доступа для массивов строк, так что приходится использовать более длинный синтаксис циклов:

```
In[2]: data = ['peter', 'Paul', 'MARY', 'gUIDO']
       [s.capitalize() for s in data]

Out[2]: ['Peter', 'Paul', 'Mary', 'Guido']
```

Вероятно, для работы с некоторыми данными этого достаточно, но при наличии отсутствующих значений все портится. Например:

```
In[3]: data = ['peter', 'Paul', None, 'MARY', 'gUIDO']
       [s.capitalize() for s in data]

---------------------------------------------------------------------------
---------------------------------------------------------------------------

AttributeError                            Traceback (most recent call last)

<ipython-input-3-fc1d891ab539> in <module>()
      1 data = ['peter', 'Paul', None, 'MARY', 'gUIDO']
----> 2 [s.capitalize() for s in data]


<ipython-input-3-fc1d891ab539> in <listcomp>(.0)
      1 data = ['peter', 'Paul', None, 'MARY', 'gUIDO']
----> 2 [s.capitalize() for s in data]


AttributeError: 'NoneType' object has no attribute 'capitalize'
---------------------------------------------------------------------------
```

Библиотека Pandas включает средства как для работы с векторизованными строковыми операциями, так и для корректной обработки отсутствующих значений посредством атрибута `str` объектов `Series` библиотеки Pandas и содержащих строки объектов `Index`. Так, допустим, мы создали объект `Series` библиотеки Pandas с теми же данными:

```
In[4]: import pandas as pd
       names = pd.Series(data)
       names

Out[4]: 0    peter
        1     Paul
        2     None
        3     MARY
        4    gUIDO
        dtype: object
```

Теперь можно вызвать один-единственный метод для преобразования строчных букв в заглавные, который будет игнорировать любые отсутствующие значения:

```
In[5]: names.str.capitalize()
```

```
Out[5]: 0    Peter
        1     Paul
        2     None
        3     Mary
        4    Guido
        dtype: object
```

С помощью Tab-автодополнения для этого атрибута `str` можно получить список всех векторизованных строковых методов, доступных в библиотеке Pandas.

Таблицы методов работы со строками библиотеки Pandas

Если вы хорошо разбираетесь в манипуляции строковыми данными в языке Python, львиная доля синтаксиса работы со строками библиотеки Pandas будет вам интуитивно понятна настолько, что достаточно, наверное, просто привести таблицу имеющихся методов. С этого и начнем, прежде чем углубимся в некоторые нюансы. Примеры в этом разделе используют следующий ряд имен:

```
In[6]: monte = pd.Series(['Graham Chapman', 'John Cleese', 'Terry Gilliam',
                          'Eric Idle', 'Terry Jones', 'Michael Palin'])
```

Методы, аналогичные строковым методам языка Python

Практически для всех встроенных строковых методов Python есть соответствующий векторизованный строковый метод библиотеки Pandas. Вот список методов атрибута `str` библиотеки Pandas, дублирующий строковые методы языка Python:

```
len()      lower()        translate()    islower()
ljust()    upper()        startswith()   isupper()
rjust()    find()         endswith()     isnumeric()
center()   rfind()        isalnum()      isdecimal()
zfill()    index()        isalpha()      split()
strip()    rindex()       isdigit()      rsplit()
rstrip()   capitalize()   isspace()      partition()
lstrip()   swapcase()     istitle()      rpartition()
```

Обратите внимание, что возвращаемые значения у них отличаются. Некоторые, например `lower()`, возвращают `Series` строк:

```
In[7]: monte.str.lower()

Out[7]: 0    graham chapman
        1       john cleese
        2     terry gilliam
```

```
3         eric idle
4       terry jones
5     michael palin
dtype: object
```

Часть других возвращает числовые значения:

```
In[8]: monte.str.len()

Out[8]: 0    14
        1    11
        2    13
        3     9
        4    11
        5    13
        dtype: int64
```

Или булевы значения:

```
In[9]: monte.str.startswith('T')

Out[9]: 0    False
        1    False
        2     True
        3    False
        4     True
        5    False
        dtype: bool
```

Или списки и другие составные значения для каждого элемента:

```
In[10]: monte.str.split()

Out[10]: 0    [Graham, Chapman]
         1       [John, Cleese]
         2     [Terry, Gilliam]
         3         [Eric, Idle]
         4       [Terry, Jones]
         5     [Michael, Palin]
         dtype: object
```

Мы увидим манипуляции над подобными объектами типа «ряды списков», когда продолжим обсуждение.

Методы, использующие регулярные выражения

Помимо этого, существует и несколько методов, принимающих на входе регулярные выражения для проверки содержимого каждого из строковых элементов и следующих некоторым соглашениям по API встроенного модуля re языка Python (табл. 3.4).

Таблица 3.4. Соответствие между методами библиотеки Pandas и функциями модуля re языка Python

Метод	Описание
match()	Вызывает функцию re.match() для каждого элемента, возвращая булево значение
extract()	Вызывает функцию re.match() для каждого элемента, возвращая подходящие группы в виде строк
findall()	Вызывает функцию re.findall() для каждого элемента
replace()	Заменяет вхождения шаблона какой-либо другой строкой
contains()	Вызывает функцию re.search() для каждого элемента, возвращая булево значение
count()	Подсчитывает вхождения шаблона
split()	Эквивалент функции str.split(), но принимающий на входе регулярные выражения
rsplit()	Эквивалент функции str.rsplit(), но принимающий на входе регулярные выражения

С помощью этих функций можно выполнять массу интересных операций. Например, можно извлечь имя из каждого элемента, выполнив поиск непрерывной группы символов в начале каждого из них:

```
In[11]: monte.str.extract('([A-Za-z]+)')

Out[11]: 0     Graham
         1       John
         2      Terry
         3       Eric
         4      Terry
         5    Michael
         dtype: object
```

Или, например, найти все имена, начинающиеся и заканчивающиеся согласным звуком, воспользовавшись символами регулярных выражений «начало строки» (^) и «конец строки» ($):

```
In[12]: monte.str.findall(r'^[^AEIOU].*[^aeiou]$')

Out[12]: 0    [Graham Chapman]
         1                  []
         2     [Terry Gilliam]
         3                  []
         4       [Terry Jones]
         5     [Michael Palin]
         dtype: object
```

Такой сжатый синтаксис регулярных выражений для записей объектов `Series` и `DataFrame` открывает массу возможностей для анализа и очистки данных.

Прочие методы

Наконец, существуют и прочие методы, пригодные для разных удобных операций (табл. 3.5).

Таблица 3.5. Прочие методы для работы со строками библиотеки Pandas

Метод	Описание
get()	Индексирует все элементы
slice()	Вырезает подстроку из каждого элемента
slice_replace()	Заменяет в каждом элементе вырезанную подстроку заданным значением
cat()	Конкатенация строк
repeat()	Повторяет значения (указанное число раз)
normalize()	Возвращает версию строки в кодировке Unicode
pad()	Добавляет пробелы слева, справа или с обеих сторон строки
wrap()	Разбивает длинные строковые значения на строки длины, не превышающей заданную
join()[1]	Объединяет строки из всех элементов с использованием заданного разделителя
get_dummies()	Извлекает значения переменных-индикаторов в виде объекта DataFrame

Векторизованный доступ к элементам и вырезание подстрок. Операции `get()` и `slice()`, в частности, предоставляют возможность векторизованного доступа к элементам из каждого массива. Например, можно вырезать первые три символа из каждого массива посредством выражения `str.slice(0, 3)`. Обратите внимание, что такая возможность доступна и с помощью обычного синтаксиса индексации языка Python, например, `df.str.slice(0, 3)` эквивалентно `df.str[0:3]`:

```
In[13]: monte.str[0:3]

Out[13]: 0    Gra
         1    Joh
         2    Ter
         3    Eri
         4    Ter
         5    Mic
         dtype: object
```

Индексация посредством `df.str.get(i)` и `df.str[i]` происходит аналогично.

---

[1] Ввиду некоторой неочевидности приведу синтаксис использования этой функции для нашего примера:
";".join(monte)
где ; — разделитель.

Эти методы `get()` и `slice()` также дают возможность обращаться к элементам возвращаемых методом `split()` массивов. Например, для извлечения фамилии из каждой записи можно использовать вместе методы `split()` и `get()`:

```
In[14]: monte.str.split().str.get(-1)

Out[14]: 0    Chapman
         1     Cleese
         2    Gilliam
         3       Idle
         4      Jones
         5      Palin
         dtype: object
```

Индикаторные переменные. Еще один метод, требующий некоторых дополнительных пояснений, — `get_dummies()`. Удобно, когда в данных имеется столбец, содержащий кодированный индикатор. Например, у нас есть набор данных, содержащий информацию в виде кодов, таких как `A="родился в США"`, `B="родился в Великобритании"`, `C="любит сыр"`, `D="любит мясные консервы"`:

```
In[15]:
full_monte = pd.DataFrame({'name': monte,
                           'info': ['B|C|D', 'B|D', 'A|C', 'B|D', 'B|C',
                           'B|C|D']})
full_monte

Out[15]:     info            name
         0  B|C|D  Graham Chapman
         1    B|D     John Cleese
         2    A|C   Terry Gilliam
         3    B|D       Eric Idle
         4    B|C     Terry Jones
         5  B|C|D   Michael Palin
```

Метод `get_dummies()` дает возможность быстро разбить все индикаторные переменные, преобразовав их в объект `DataFrame`:

```
In[16]: full_monte['info'].str.get_dummies('|')

Out[16]:    A  B  C  D
         0  0  1  1  1
         1  0  1  0  1
         2  1  0  1  0
         3  0  1  0  1
         4  0  1  1  0
         5  0  1  1  1
```

Используя эти операции как «строительные блоки», можно создать бесчисленное множество обрабатывающих строки процедур для очистки данных.

Мы не будем углубляться в эти методы, но я рекомендую прочитать раздел Working with Text Data («Работа с текстовыми данными») из онлайн-документации

библиотеки Pandas (http://pandas.pydata.org/pandas-docs/stable/text.html) или заглянуть в раздел «Дополнительные источники информации» данной главы.

Пример: база данных рецептов

Описанные векторизованные строковые операции оказываются наиболее полезными при очистке сильно зашумленных реальных данных. Здесь мы рассмотрим пример такой очистки, воспользовавшись полученной из множества различных интернет-источников базой данных рецептов. Наша цель — разбор рецептов на списки ингредиентов, чтобы можно было быстро найти рецепт, исходя из имеющихся в распоряжении ингредиентов.

Используемые для компиляции сценарии можно найти по адресу https://github.com/fictivekin/openrecipes, как и ссылку на актуальную версию базы.

По состоянию на весну 2016 года размер базы данных составляет около 30 Мбайт, ее можно скачать и разархивировать с помощью следующих команд:

```
In[17]: # !curl -O
        # http://openrecipes.s3.amazonaws.com/20131812-recipeitems.json.gz
        # !gunzip 20131812-recipeitems.json.gz
```

База данных находится в формате JSON, так что можно попробовать воспользоваться функцией `pd.read_json` для ее чтения:

```
In[18]: try:
            recipes = pd.read_json('recipeitems-latest.json')
        except ValueError as e:
            print("ValueError:", e)
```

```
ValueError: Trailing data
```

Упс! Мы получили ошибку `ValueError` с упоминанием наличия «хвостовых данных». Если поискать эту ошибку в Интернете, складывается впечатление, что она появляется из-за использования файла, в котором *каждая строка* сама по себе является корректным JSON, а весь файл в целом — нет. Рассмотрим, справедливо ли это объяснение:

```
In[19]: with open('recipeitems-latest.json') as f:
            line = f.readline()
        pd.read_json(line).shape
```

```
Out[19]: (2, 12)
```

Да, очевидно, каждая строка — корректный JSON, так что нам нужно соединить их все воедино. Один из способов сделать это — фактически сформировать строковое представление, содержащее все записи JSON, после чего загрузить все с помощью `pd.read_json`:

```
In[20]: # Читаем весь файл в массив Python
        with open('recipeitems-latest.json', 'r') as f:
            # Извлекаем каждую строку
            data = (line.strip() for line in f)
            # Преобразуем так, чтобы каждая строка была элементом списка
            data_json = "[{0}]".format(','.join(data))
        # Читаем результат в виде JSON
        recipes = pd.read_json(data_json)

In[21]: recipes.shape

Out[21]: (173278, 17)
```

Видим, что здесь почти 200 тысяч рецептов и 17 столбцов. Посмотрим на одну из строк, чтобы понять, что мы получили:

```
In[22]: recipes.iloc[0]

Out[22]:
_id                                {'$oid': '5160756b96cc62079cc2db15'}
cookTime                                                          PT30M
creator                                                             NaN
dateModified                                                        NaN
datePublished                                                2013-03-11
description           Late Saturday afternoon, after Marlboro Man ha...
image                 http://static.thepioneerwoman.com/cooking/file...
ingredients           Biscuits\n3 cups All-purpose Flour\n2 Tablespo...
name                                  Drop Biscuits and Sausage Gravy
prepTime                                                          PT10M
recipeCategory                                                      NaN
recipeInstructions                                                  NaN
recipeYield                                                          12
source                                                   thepioneerwoman
totalTime                                                           NaN
ts                                            {'$date': 1365276011104}
url                   http://thepioneerwoman.com/cooking/2013/03/dro...
Name: 0, dtype: object
```

Здесь содержится немало информации, но *б*ольшая ее часть находится в беспорядочном виде, как это обычно и бывает со взятыми из Интернета данными. В частности, список ингредиентов находится в строковом формате и нам придется аккуратно извлечь оттуда интересующую нас информацию. Начнем с того, что взглянем поближе на ингредиенты:

```
In[23]: recipes.ingredients.str.len().describe()

Out[23]: count    173278.000000
         mean        244.617926
         std         146.705285
         min           0.000000
         25%         147.000000
```

```
50%          221.000000
75%          314.000000
max         9067.000000
Name: ingredients, dtype: float64
```

Средняя длина списка ингредиентов составляет 250 символов при минимальной длине 0 и максимальной — почти 10 000 символов!

Из любопытства посмотрим, у какого рецепта самый длинный список ингредиентов:

```
In[24]: recipes.name[np.argmax(recipes.ingredients.str.len())]

Out[24]: 'Carrot Pineapple Spice &amp;amp; Brownie Layer Cake with Whipped Cream
&amp;amp; Cream Cheese Frosting and Marzipan Carrots'
```

Этот рецепт явно выглядит не очень простым.

Можно сделать и другие открытия на основе сводных показателей. Например, посмотрим, сколько рецептов описывают еду, предназначенную для завтрака:

```
In[33]: recipes.description.str.contains('[Bb]reakfast').sum()

Out[33]: 3524
```

Или сколько рецептов содержат корицу (cinnamon) в списке ингредиентов:

```
In[34]: recipes.ingredients.str.contains('[Cc]innamon').sum()

Out[34]: 10526
```

Можно даже посмотреть, есть ли рецепты, в которых название этого ингредиента написано с орфографической ошибкой, как cinamon:

```
In[27]: recipes.ingredients.str.contains('[Cc]inamon').sum()

Out[27]: 11
```

Такая разновидность обязательного предварительного изучения данных возможна благодаря инструментам по работе со строками библиотеки Pandas. Именно в сфере такой очистки данных Python действительно силен.

Простая рекомендательная система для рецептов

Немного углубимся в этот пример и начнем работу над простой рекомендательной системой для рецептов: по заданному списку ингредиентов необходимо найти рецепт, использующий их все. Концептуально простая, эта задача усложняется неоднородностью данных: не существует удобной операции, которая позволила бы извлечь из каждой строки очищенный список ингредиентов. Так что мы немного сжульничаем: начнем со списка распространенных ингредиентов и будем искать,

в каждом ли рецепте они содержатся в списке ингредиентов. Для упрощения ограничимся травами и специями:

```
In[28]: spice_list = ['salt', 'pepper', 'oregano', 'sage', 'parsley',
                      'rosemary', 'tarragon', 'thyme', 'paprika', 'cumin']
```

Мы можем создать булев объект `DataFrame`, состоящий из значений `True` и `False`, указывающих, содержится ли данный ингредиент в списке:

```
In[29]:
import re
spice_df = pd.DataFrame(
           dict((spice, recipes.ingredients.str.contains(spice, re.IGNORECASE))
                                                for spice in spice_list))
spice_df.head()
```

```
Out[29]:
   cumin oregano paprika parsley pepper rosemary   sage   salt tarragon  thyme
0  False   False   False   False  False    False   True  False    False  False
1  False   False   False   False  False    False  False  False    False  False
2   True   False   False   False   True    False  False   True    False  False
3  False   False   False   False  False    False  False  False    False  False
4  False   False   False   False  False    False  False  False    False  False
```

Теперь в качестве примера допустим, что мы хотели бы найти рецепт, в котором используются петрушка (parsley), паприка (paprika) и эстрагон (tarragon). Это можно сделать очень быстро, воспользовавшись методом `query()` объекта `DataFrame`, который мы обсудим подробнее в разделе «Увеличение производительности библиотеки Pandas: eval() и query()» данной главы:

```
In[30]: selection = spice_df.query('parsley & paprika & tarragon')
        len(selection)

Out[30]: 10
```

Мы нашли всего десять рецептов с таким сочетанием ингредиентов. Воспользуемся возвращаемым этой выборкой индексом, чтобы выяснить названия этих рецептов:

```
In[31]: recipes.name[selection.index]

Out[31]: 2069      All cremat with a Little Gem, dandelion and wa...
         74964                         Lobster with Thermidor butter
         93768      Burton's Southern Fried Chicken with White Gravy
         113926                    Mijo's Slow Cooker Shredded Beef
         137686                    Asparagus Soup with Poached Eggs
         140530                                 Fried Oyster Po'boys
         158475                Lamb shank tagine with herb tabbouleh
         158486                 Southern fried chicken in buttermilk
         163175            Fried Chicken Sliders with Pickles + Slaw
         165243                         Bar Tartine Cauliflower Salad
         Name: name, dtype: object
```

Теперь, сократив наш список рецептов почти в 20 000 раз, мы можем принять более взвешенное решение: что готовить на обед.

Дальнейшая работа с рецептами

Надеемся, что этот пример позволил вам попробовать на вкус (па-ба-ба-бам!) операции по очистке данных, которые хорошо выполняются с помощью строковых методов библиотеки Pandas. Создание надежной рекомендательной системы для рецептов потребовало бы *намного* больше труда! Важной частью этой задачи было бы извлечение из каждого рецепта полного списка ингредиентов. К сожалению, разнообразие используемых форматов делает этот процесс весьма трудоемким. Как подсказывает нам Капитан Очевидность, в науке о данных очистка поступивших из реального мира данных часто представляет собой основную часть работы, и библиотека Pandas предоставляет инструменты для эффективного выполнения этой задачи.

Работа с временными рядами

Библиотека Pandas была разработана в расчете на построение финансовых моделей, так что, как вы могли и ожидать, она содержит весьма широкий набор инструментов для работы с датой, временем и индексированными по времени данными. Данные о дате и времени могут находиться в нескольких видах, которые мы сейчас обсудим.

- *Метки даты/времени* ссылаются на конкретные моменты времени (например, 4 июля 2015 года в 07:00 утра).
- *Временные интервалы* и *периоды* ссылаются на отрезки времени между конкретными начальной и конечной точками (например, 2015 год). Периоды обычно представляют собой особый случай интервалов, с непересекающимися интервалами одинаковой длительности (например, 24-часовые периоды времени, составляющие сутки).
- *Временная дельта* (она же *продолжительность*) относится к отрезку времени конкретной длительности (например, 22,56 с).

В данном разделе мы расскажем, как работать с каждым из этих типов временных данных в библиотеке Pandas. Короткий раздел никоим образом не претендует на звание исчерпывающего руководства по имеющимся в Python или библиотеке Pandas инструментам работы с временными рядами. Он представляет собой обзор работы с временными рядами в общих чертах. Мы начнем с краткого обсуждения инструментов для работы с датой и временем в языке Python, прежде чем перейти непосредственно к обсуждению инструментов библиотеки Pandas. После перечисления источников углубленной информации мы рассмотрим несколько кратких примеров работы с данными временных рядов в библиотеке Pandas.

Дата и время в языке Python

В мире языка Python существует немало представлений дат, времени, временных дельт и интервалов времени. Хотя для приложений науки о данных наиболее удобны инструменты работы с временными рядами библиотеки Pandas, не помешает посмотреть на другие используемые в Python пакеты.

Нативные даты и время языка Python: пакеты datetime и dateutil

Базовые объекты Python для работы с датами и временем располагаются во встроенном пакете `datetime`. Его, вместе со сторонним модулем `dateutil`, можно использовать для быстрого выполнения множества удобных операций над датами и временем. Например, можно вручную сформировать дату с помощью типа `datetime`:

```
In[1]: from datetime import datetime
       datetime(year=2015, month=7, day=4)

Out[1]: datetime.datetime(2015, 7, 4, 0, 0)
```

Или, воспользовавшись модулем `dateutil`, можно выполнять синтаксический разбор дат, находящихся во множестве строковых форматов:

```
In[2]: from dateutil import parser
       date = parser.parse("4th of July, 2015")
       date

Out[2]: datetime.datetime(2015, 7, 4, 0, 0)
```

При наличии объекта `datetime` можно делать вывод дня недели:

```
In[3]: date.strftime('%A')

Out[3]: 'Saturday'
```

В этой команде мы использовали для вывода даты один из стандартных кодов форматирования строк (`"%A"`), о котором можно прочитать в разделе `strftime` (https://docs.python.org/3/library/datetime.html#strftime-and-strptime-behavior) документации по пакету `datetime` (https://docs.python.org/3/library/datetime.html) языка Python. Документацию по другим полезным утилитам для работы с датой и временем можно найти в онлайн-документации пакета `dateutil` (http://labix.org/python-dateutil). Не помешает также быть в курсе связанного с ними пакета `pytz` (http://pytz.sourceforge.net), содержащего инструменты для работы с частью данных временных рядов — часовыми поясами.

Сила пакетов `datetime` и `dateutil` заключается в их гибкости и удобном синтаксисе: эти объекты и их встроенные методы можно использовать для выполнения практически любой интересующей вас операции. Единственное, в чем они работают плохо, это работа с большими массивами дат и времени: подобно спискам числовых

переменных языка Python, работающим неоптимально по сравнению с типизированными числовыми массивами в стиле библиотеки NumPy, списки объектов даты/времени Python работают с меньшей производительностью, чем типизированные массивы кодированных дат.

Типизированные массивы значений времени: тип datetime64 библиотеки NumPy

Указанная слабая сторона формата даты/времени языка Python побудила команду разработчиков библиотеки NumPy добавить набор нативных типов данных временных рядов. Тип (dtype) `datetime64` кодирует даты как 64-битные целые числа, так что представление массивов дат оказывается очень компактным. Для типа `datetime64` требуется очень точно заданный формат входных данных:

```
In[4]: import numpy as np
       date = np.array('2015-07-04', dtype=np.datetime64)
       date

Out[4]: array(datetime.date(2015, 7, 4), dtype='datetime64[D]')
```

Но как только дата отформатирована, можно быстро выполнять над ней различные векторизованные операции:

```
In[5]: date + np.arange(12)

Out[5]:
array(['2015-07-04', '2015-07-05', '2015-07-06', '2015-07-07',
       '2015-07-08', '2015-07-09', '2015-07-10', '2015-07-11',
       '2015-07-12', '2015-07-13', '2015-07-14', '2015-07-15'],
       dtype='datetime64[D]')
```

Поскольку `datetime64`-массивы библиотеки NumPy содержат данные одного типа, подобные операции выполняются намного быстрее, чем если работать непосредственно с объектами `datetime` языка Python, особенно если речь идет о больших массивах (мы рассматривали эту разновидность векторизации в разделе «Выполнение вычислений над массивами библиотеки NumPy: универсальные функции» главы 2).

Важный нюанс относительно объектов `datetime64` и `timedelta64`: они основаны на *базовой единице времени* (fundamental time unit). Поскольку объект `datetime64` ограничен точностью 64 бита, кодируемый им диапазон времени составляет эту базовую единицу, умноженную на 2^{64}. Другими словами, `datetime64` навязывает компромисс между *разрешающей способностью по времени* и *максимальным промежутком времени*.

Например, если нам требуется разрешающая способность 1 наносекунда, то у нас будет информация, достаточная для кодирования только интервала 2^{64} наносекунды, или чуть более 600 лет. Библиотека NumPy определяет требуемую единицу

на основе входной информации; например, вот дата/время на основе единицы в один день:

```
In[6]: np.datetime64('2015-07-04')

Out[6]: numpy.datetime64('2015-07-04')
```

Вот дата/время на основе единицы в одну минуту:

```
In[7]: np.datetime64('2015-07-0412:00')

Out[7]: numpy.datetime64('2015-07-04T12:00')
```

Обратите внимание, что часовой пояс автоматически задается в соответствии с местным временем выполняющего код компьютера. Можно обеспечить принудительное использование любой требуемой базовой единицы с помощью одного из множества кодов форматирования; например, вот дата/время на основе единицы в одну наносекунду:

```
In[8]: np.datetime64('2015-07-0412:59:59.50', 'ns')

Out[8]: numpy.datetime64('2015-07-04T12:59:59.500000000')
```

В табл. 3.6, взятой из документации по типу `datetime64` библиотеки NumPy, перечислены доступные для использования коды форматирования, а также относительные и абсолютные промежутки времени, которые можно кодировать с их помощью.

Таблица 3.6. Описание кодов форматирования даты и времени

Код	Значение	Промежуток времени (относительный)	Промежуток времени (абсолютный)
Y	Год	±9.2e18 лет	[9.2e18 до н. э., 9.2e18 н. э.]
M	Месяц	±7.6e17 лет	[7.6e17 до н. э., 7.6e17 н. э.]
W	Неделя	±1.7e17 лет	[1.7e17 до н. э., 1.7e17 н. э.]
D	День	±2.5e16 лет	[2.5e16 до н. э., 2.5e16 н. э.]
h	Час	±1.0e15 лет	[1.0e15 до н. э., 1.0e15 н. э.]
m	Минута	±1.7e13 лет	[1.7e13 до н. э., 1.7e13 н. э.]
s	Секунда	±2.9e12 лет	[2.9e9 до н. э., 2.9e9 н. э.]
ms	Миллисекунда	±2.9e9 лет	[2.9e6 до н. э., 2.9e6 н. э.]
us	Микросекунда	±2.9e6 лет	[290301 до н. э., 294241 н. э.]
ns	Наносекунда	±292 лет	[1678 до н. э., 2262 н. э.]
ps	Пикосекунда	±106 дней	[1969 до н. э., 1970 н. э.]
fs	Фемтосекунда	±2.6 часов	[1969 до н. э., 1970 н. э.]
as	Аттосекунда	±9.2 секунды	[1969 до н. э., 1970 н. э.]

Удобное значение по умолчанию для типов данных, встречающихся в реальном мире, — `datetime64[ns]`, позволяющее кодировать достаточный диапазон современных дат с высокой точностью.

Наконец, отметим, что, хотя тип данных `datetime64` лишен некоторых недостатков встроенного типа данных `datetime` языка Python, ему недостает многих предоставляемых `datetime` и особенно `dateutil` удобных методов и функций. Больше информации можно найти в документации по типу `datetime64` библиотеки NumPy (http://docs.scipy.org/doc/numpy/reference/arrays.datetime.html).

Даты и время в библиотеке Pandas: избранное из лучшего

Библиотека Pandas предоставляет, основываясь на всех только что обсуждавшихся инструментах, объект `Timestamp`, сочетающий удобство использования `datetime` и `dateutil` с эффективным хранением и векторизованным интерфейсом типа `numpy.datetime64`. Библиотека Pandas умеет создавать из нескольких таких объектов `Timestamp` объект класса `DatetimeIndex`, который можно использовать для индексации данных в объектах `Series` или `DataFrame`. Можно применить инструменты библиотеки Pandas для воспроизведения вышеприведенной наглядной демонстрации. Можно выполнить синтаксический разбор строки с датой в гибком формате и воспользоваться кодами форматирования, чтобы вывести день недели:

```
In[9]: import pandas as pd
       date = pd.to_datetime("4th of July, 2015")
       date

Out[9]: Timestamp('2015-07-0400:00:00')

In[10]: date.strftime('%A')

Out[10]: 'Saturday'
```

Кроме этого, можно выполнять векторизованные операции в стиле библиотеки NumPy непосредственно над этим же объектом:

```
In[11]: date + pd.to_timedelta(np.arange(12), 'D')

Out[11]: DatetimeIndex(['2015-07-04', '2015-07-05', '2015-07-06', '2015-07-07',
                        '2015-07-08', '2015-07-09', '2015-07-10', '2015-07-11',
                        '2015-07-12', '2015-07-13', '2015-07-14', '2015-07-15'],
                       dtype='datetime64[ns]', freq=None)
```

В следующем разделе мы подробнее рассмотрим манипуляции над данными временных рядов с помощью предоставляемых библиотекой Pandas инструментов.

Временные ряды библиотеки Pandas: индексация по времени

Инструменты для работы с временными рядами библиотеки Pandas особенно удобны при необходимости *индексации данных по меткам даты/времени*. Например, создадим объект `Series` с индексированными по времени данными:

```
In[12]: index = pd.DatetimeIndex(['2014-07-04', '2014-08-04',
                                  '2015-07-04', '2015-08-04'])
        data = pd.Series([0, 1, 2, 3], index=index)
        data

Out[12]: 2014-07-04    0
         2014-08-04    1
         2015-07-04    2
         2015-08-04    3
         dtype: int64
```

Теперь, когда эти данные находятся в объекте `Series`, можно использовать для них любые из обсуждавшихся в предыдущих разделах паттернов индексации `Series`, передавая значения, которые допускают приведение к типу даты:

```
In[13]: data['2014-07-04':'2015-07-04']

Out[13]: 2014-07-04    0
         2014-08-04    1
         2015-07-04    2
         dtype: int64
```

Имеются также дополнительные специальные операции индексации, предназначенные только для дат. Например, можно указать год, чтобы получить срез всех данных за этот год:

```
In[14]: data['2015']

Out[14]: 2015-07-04    2
         2015-08-04    3
         dtype: int64
```

Позднее мы рассмотрим еще примеры удобства индексации по датам. Но сначала изучим имеющиеся структуры данных для временных рядов.

Структуры данных для временных рядов библиотеки Pandas

В этом разделе мы рассмотрим основные структуры данных, предназначенные для работы с временными рядами.

- Для *меток даты/времени* библиотека Pandas предоставляет тип данных `Timestamp`. Этот тип является заменой для нативного типа данных `datetime` языка Python, он основан на более эффективном типе данных `numpy.datetime64`. Соответствующая индексная конструкция — `DatetimeIndex`.
- Для *периодов времени* библиотека Pandas предоставляет тип данных `Period`. Этот тип на основе типа данных `numpy.datetime64` кодирует интервал времени фиксированной периодичности. Соответствующая индексная конструкция — `PeriodIndex`.
- Для *временных дельт* (*продолжительностей*) библиотека Pandas предоставляет тип данных `Timedelta`. `Timedelta` — основанная на типе `numpy.timedelta64` более эффективная замена нативного типа данных `datetime.timedelta` языка Python. Соответствующая индексная конструкция — `TimedeltaIndex`.

Самые базовые из этих объектов даты/времени — объекты `Timestamp` и `DatetimeIndex`. Хотя к ним и можно обращаться непосредственно, чаще используют функцию `pd.to_datetime()`, умеющую выполнять синтаксический разбор широкого диапазона форматов. При передаче в функцию `pd.to_datetime()` отдельной даты она возвращает `Timestamp`, при передаче ряда дат по умолчанию возвращает `DatetimeIndex`:

```
In[15]: dates = pd.to_datetime([datetime(2015, 7, 3), '4th of July, 2015',
                                '2015-Jul-6', '07-07-2015', '20150708'])
        dates

Out[15]: DatetimeIndex(['2015-07-03', '2015-07-04', '2015-07-06', '2015-07-07',
                        '2015-07-08'],
                       dtype='datetime64[ns]', freq=None)
```

Любой объект `DatetimeIndex` можно с помощью функции `to_period()` преобразовать в объект `PeriodIndex`, указав код для периодичности интервала. В данном случае мы использовали код `'D'`, означающий, что периодичность интервала — один день:

```
In[16]: dates.to_period('D')

Out[16]: PeriodIndex(['2015-07-03', '2015-07-04', '2015-07-06', '2015-07-07',
                      '2015-07-08'],
                     dtype='int64', freq='D')
```

Объект `TimedeltaIndex` создается, например, при вычитании одной даты из другой:

```
In[17]: dates - dates[0]

Out[17]:
TimedeltaIndex(['0 days', '1 days', '3 days', '4 days', '5 days'],
               dtype='timedelta64[ns]', freq=None)
```

Регулярные последовательности: функция pd.date_range(). Чтобы облегчить создание регулярных последовательностей, библиотека Pandas предоставляет несколько функций: `pd.date_range()` — для меток даты/времени, `pd.period_range()` — для периодов времени и `pd.timedelta_range()` — для временных дельт. Мы уже видели, что функции `range()` языка Python и `np.arange()` библиотеки NumPy преобразуют начальную точку, конечную точку и (необязательную) величину шага в последовательность. Аналогично функция `pd.date_range()` создает регулярную последовательность дат, принимая на входе начальную дату, конечную дату и необязательный код периодичности. По умолчанию период равен одному дню:

```
In[18]: pd.date_range('2015-07-03', '2015-07-10')

Out[18]: DatetimeIndex(['2015-07-03', '2015-07-04', '2015-07-05', '2015-07-06',
                        '2015-07-07', '2015-07-08', '2015-07-09', '2015-07-10'],
                       dtype='datetime64[ns]', freq='D')
```

В качестве альтернативы можно также задать диапазон дат с помощью не начальной и конечной точек, а посредством начальной точки и количества периодов времени:

```
In[19]: pd.date_range('2015-07-03', periods=8)

Out[19]: DatetimeIndex(['2015-07-03', '2015-07-04', '2015-07-05', '2015-07-06',
                        '2015-07-07', '2015-07-08', '2015-07-09', '2015-07-10'],
                       dtype='datetime64[ns]', freq='D')
```

Можно изменить интервал времени, поменяв аргумент `freq`, имеющий по умолчанию значение `'D'`. Например, в следующем фрагменте мы создаем диапазон часовых меток даты/времени:

```
In[20]: pd.date_range('2015-07-03', periods=8, freq='H')

Out[20]: DatetimeIndex(['2015-07-0300:00:00', '2015-07-0301:00:00',
                        '2015-07-0302:00:00', '2015-07-0303:00:00',
                        '2015-07-0304:00:00', '2015-07-0305:00:00',
                        '2015-07-0306:00:00', '2015-07-0307:00:00'],
                       dtype='datetime64[ns]', freq='H')
```

Для создания регулярных последовательностей значений периодов или временных дельт можно воспользоваться функциями `pd.period_range()` и `pd.timedelta_range()`, напоминающими функцию `date_range()`. Вот несколько периодов времени длительностью в месяц:

```
In[21]: pd.period_range('2015-07', periods=8, freq='M')

Out[21]:
PeriodIndex(['2015-07', '2015-08', '2015-09', '2015-10', '2015-11', '2015-12',
             '2016-01', '2016-02'],
            dtype='int64', freq='M')
```

Вот последовательность продолжительностей, увеличивающихся на час:

```
In[22]: pd.timedelta_range(0, periods=10, freq='H')

Out[22]:
TimedeltaIndex(['00:00:00', '01:00:00', '02:00:00', '03:00:00', '04:00:00',
                '05:00:00', '06:00:00', '07:00:00', '08:00:00', '09:00:00'],
               dtype='timedelta64[ns]', freq='H')
```

Все эти операции требуют понимания кодов периодичности, приведенных в следующем разделе.

Периодичность и смещения дат

Периодичность или смещение даты — базовое понятие для инструментов библиотеки Pandas, необходимых для работы с временными рядами. Аналогично уже продемонстрированным кодам `D` (день) и `H` (час) можно использовать коды для задания любой требуемой периодичности. В табл. 3.7 описаны основные существующие коды.

Таблица 3.7. Список кодов периодичности библиотеки Pandas

Код	Описание	Код	Описание
D	Календарный день	B	Рабочий день
W	Неделя		
M	Конец месяца	BM	Конец отчетного месяца
Q	Конец квартала	BQ	Конец отчетного квартала
A	Конец года	BA	Конец финансового года
H	Час	BH	Рабочие часы
T	Минута		
S	Секунда		
L	Миллисекунда		
U	Микросекунда		
N	Наносекунда		

Периодичность в месяц, квартал и год определяется на конец соответствующего периода. Добавление к любому из кодов суффикса S приводит к определению начала периода (табл. 3.8).

Таблица 3.8. Список стартовых кодов периодичности

Код	Описание
MS	Начало месяца
BMS	Начало отчетного месяца
QS	Начало квартала

Код	Описание
BQS	Начало отчетного квартала
AS	Начало года
BAS	Начало финансового года

Кроме этого, можно изменить используемый для определения квартала или года месяц с помощью добавления в конец кода месяца, состоящего из трех букв:

- `Q-JAN`, `BQ-FEB`, `QS-MAR`, `BQS-APR` и т. д.
- `A-JAN`, `BA-FEB`, `AS-MAR`, `BAS-APR` и т. д.

Аналогичным образом можно изменить точку разбиения для недельной периодичности, добавив состоящий из трех букв код дня недели:

`W-SUN`, `W-MON`, `W-TUE`, `W-WED` и т. д.

Для указания иной периодичности можно сочетать коды с числами. Например, для периодичности 2 часа 30 минут можно скомбинировать коды для часа (`H`) и минуты (`T`):

```
In[23]: pd.timedelta_range(0, periods=9, freq="2H30T")

Out[23]:
TimedeltaIndex(['00:00:00', '02:30:00', '05:00:00', '07:30:00', '10:00:00',
                '12:30:00', '15:00:00', '17:30:00', '20:00:00'],
               dtype='timedelta64[ns]', freq='150T')
```

Все эти короткие коды ссылаются на соответствующие экземпляры смещений даты/времени временных рядов библиотеки Pandas, которые можно найти в модуле `pd.tseries.offsets`. Например, можно непосредственно создать смещение в один рабочий день следующим образом:

```
In[24]: from pandas.tseries.offsets import BDay
        pd.date_range('2015-07-01', periods=5, freq=BDay())

Out[24]: DatetimeIndex(['2015-07-01', '2015-07-02', '2015-07-03',
                        '2015-07-06','2015-07-07'],
                       dtype='datetime64[ns]', freq='B')
```

Дальнейшее обсуждение периодичности и смещений времени можно найти в разделе «Объекты DateOffset» (http://pandas.pydata.org/pandas-docs/stable/timeseries.html#dateoffset-objects) онлайн-документации библиотеки Pandas.

Передискретизация, временные сдвиги и окна

Возможность использовать дату/время в качестве индексов для интуитивно понятной организации данных и доступа к ним — немаловажная часть инструментария

библиотеки Pandas по работе с временными рядами. При этом сохраняются общие преимущества использования индексированных данных (автоматическое выравнивание во время операций, интуитивно понятные срезы и доступ к данным и т. д.), но библиотека Pandas предоставляет еще несколько дополнительных операций специально для временных рядов.

Мы рассмотрим некоторые из них, воспользовавшись в качестве примера данными по курсам акций. Библиотека Pandas, будучи разработанной в значительной степени для работы с финансовыми данными, имеет для этой цели несколько весьма специфических инструментов. Например, сопутствующий Pandas пакет `pandas-datareader` (который можно установить с помощью команды `conda install pandas-datareader`) умеет импортировать финансовые данные из множества источников, включая Yahoo! Finance, Google Finance и другие.

В следующем примере мы импортируем историю цен акций для Google:

```
In[25]: from pandas_datareader import data
        goog = data.DataReader('GOOG', start='2004', end='2016',
                               data_source='google')
        goog.head()
```

```
Out[25]:              Open   High    Low  Close  Volume
         Date
         2004-08-19  49.96  51.98  47.93  50.12     NaN
         2004-08-20  50.69  54.49  50.20  54.10     NaN
         2004-08-23  55.32  56.68  54.47  54.65     NaN
         2004-08-24  55.56  55.74  51.73  52.38     NaN
         2004-08-25  52.43  53.95  51.89  52.95     NaN
```

Для простоты будем использовать только окончательную цену:

```
In[26]: goog = goog['Close']
```

Визуализировать это можно с помощью метода `plot()` после обычных команд настройки Matplotlib (рис. 3.5):

```
In[27]: %matplotlib inline
        import matplotlib.pyplot as plt
        import seaborn; seaborn.set()
```

```
In[28]: goog.plot();
```

Передискретизация и изменение периодичности интервалов

При работе с данными временных рядов часто бывает необходимо переразбить их с использованием интервалов другой периодичности. Сделать это можно с помощью метода `resample()` или гораздо более простого метода `asfreq()`. Основная

разница между ними заключается в том, что `resample()` выполняет *агрегирование данных*, а `asfreq()` — *выборку данных*.

Рассмотрим, что возвращают эти два метода для данных по ценам закрытия Google при понижающей дискретизации данных. Здесь мы выполняем передискретизацию данных на конец финансового года (рис. 3.6).

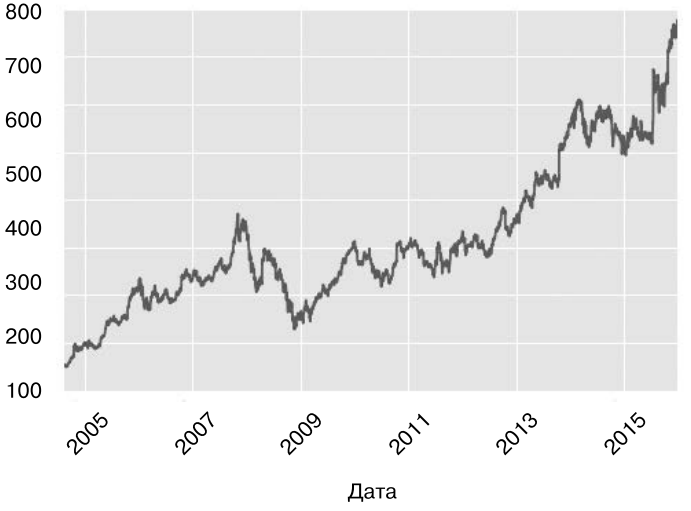

Рис. 3.5. График изменения цен акций Google с течением времени

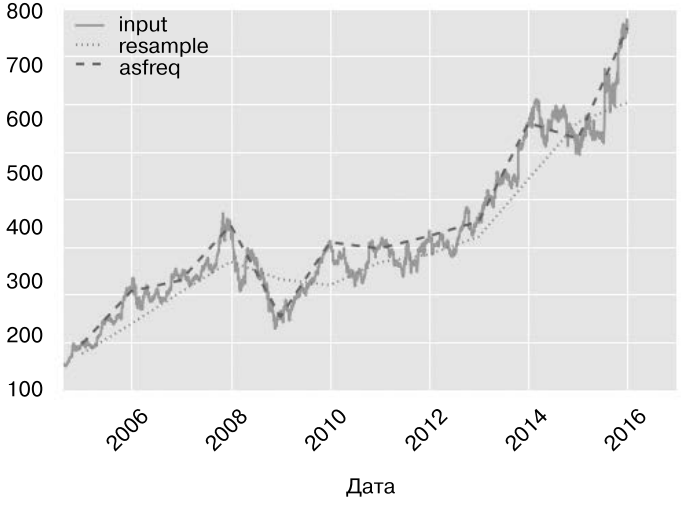

Рис. 3.6. Передискретизация цен акций Google

```
In[29]: goog.plot(alpha=0.5, style='-')
        goog.resample('BA').mean().plot(style=':')
```

```
        goog.asfreq('BA').plot(style='--');
        plt.legend(['input', 'resample', 'asfreq'],
                   loc='upper left');
```

Обратите внимание на различие: в каждой точке `resample` выдает *среднее значение за предыдущий год*, а `asfreq` — *значение на конец года*.

В случае повышающей дискретизации методы `resample()` и `asfreq()` в значительной степени идентичны, хотя доступных для использования параметров у `resample()` гораздо больше. В данном случае оба этих метода по умолчанию оставляют значения интерполированных точек пустыми, то есть заполненными значениями *NA*. Аналогично обсуждавшейся выше функции `pd.fillna()` метод `asfreq()` принимает аргумент `method`, определяющий, откуда будут браться значения для таких точек. Здесь мы передискретизируем данные по рабочим дням с периодичностью обычного дня, то есть включая выходные дни (рис. 3.7):

```
In[30]: fig, ax = plt.subplots(2, sharex=True)
        data = goog.iloc[:10]

        data.asfreq('D').plot(ax=ax[0], marker='o')

        data.asfreq('D', method='bfill').plot(ax=ax[1], style='-o')
        data.asfreq('D', method='ffill').plot(ax=ax[1], style='--o')
        ax[1].legend(["back-fill", "forward-fill"]);
```

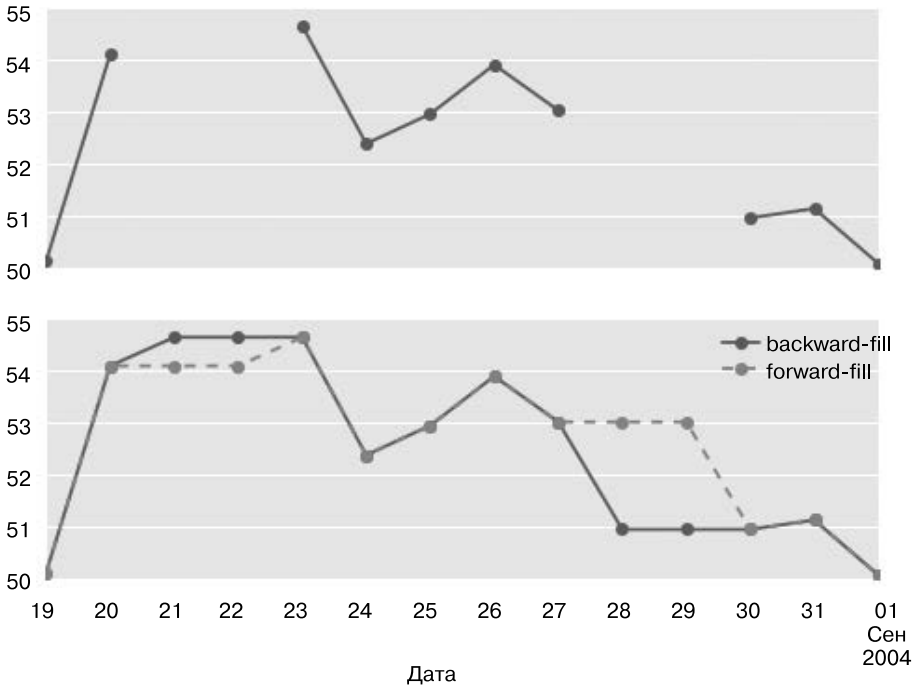

Рис. 3.7. Сравнение интерполяции вперед (forward-fill interpolation) и назад (backward-fill interpolation)

Верхний график представляет поведение по умолчанию: в выходные дни[1] значения равны *NA* и отсутствуют на графике. Нижний график демонстрирует различия между двумя методиками заполнения пропусков: интерполяцией вперед (forward-fill interpolation) и интерполяцией назад (back-fill interpolation).

Временные сдвиги

Еще одна распространенная операция с временными рядами — сдвиг данных во времени. В библиотеке Pandas есть два родственных метода для подобных вычислений: `shift()` и `tshift()`. Разница между ними заключается в том, что `shift()` выполняет *сдвиг данных*, а `tshift()` — *сдвиг индекса*. В обоих случаях сдвиг задается кратным периоду.

В следующем фрагменте кода мы сдвигаем как данные, так и индекс на 900 дней (рис. 3.8):

```
In[31]: fig, ax = plt.subplots(3, sharey=True)

        # задаем периодичность данных
        goog = goog.asfreq('D', method='pad')

        goog.plot(ax=ax[0])
        goog.shift(900).plot(ax=ax[1])
        goog.tshift(900).plot(ax=ax[2])

        # Легенды и пояснения
        local_max = pd.to_datetime('2007-11-05')
        offset = pd.Timedelta(900, 'D')

        ax[0].legend(['input'], loc=2)
        ax[0].get_xticklabels()[4].set(weight='heavy', color='red')
        ax[0].axvline(local_max, alpha=0.3, color='red')

        ax[1].legend(['shift(900)'], loc=2)
        ax[1].get_xticklabels()[4].set(weight='heavy', color='red')
        ax[1].axvline(local_max + offset, alpha=0.3, color='red')

        ax[2].legend(['tshift(900)'], loc=2)
        ax[2].get_xticklabels()[1].set(weight='heavy', color='red')
        ax[2].axvline(local_max + offset, alpha=0.3, color='red');
```

Видим, что `shift(900)` сдвигает *данные* на 900 дней, перемещая часть из них за пределы графика (и оставляя *NA*-значения с другой стороны), в то время как `tshift(900)` сдвигает на 900 дней *значения индекса*.

Такую разновидность сдвигов часто используют для вычисления изменений с течением времени. Например, мы воспользовались сдвинутыми значениями, чтобы вычислить прибыль за год от вложений в акции Google по всему набору данных (рис. 3.9):

[1] В данном случае имеются в виду конкретно так называемые банковские выходные дни, когда банки не работают.

```
In[32]: ROI = 100 * (goog.tshift(-365) / goog - 1)
        ROI.plot()
        plt.ylabel('% Return on Investment'); # Прибыль от вложений
```

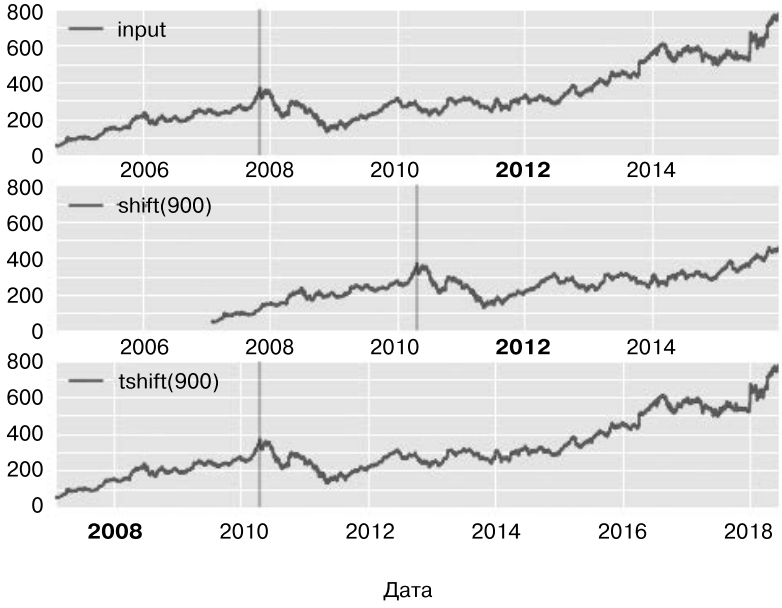

Рис. 3.8. Сравнение shift и tshift

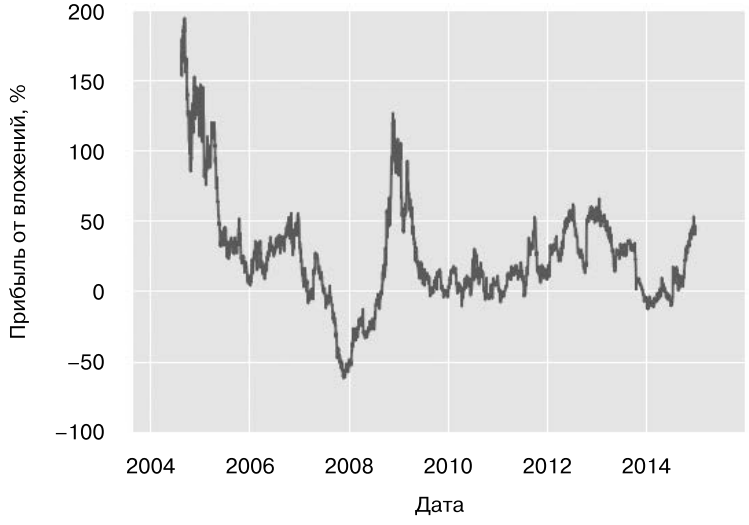

Рис. 3.9. Прибыль на текущий день от вложений в акции Google

Это помогает увидеть общие тренды акций Google: до сих пор наиболее благоприятным для инвестиций в акции Google был (что неудивительно) момент вскоре после первоначального их размещения на рынке, а также в середине экономической рецессии 2009 года.

Скользящие окна

Скользящие статистические показатели — третья из реализованных в библиотеке Pandas разновидностей операций, предназначенных для временных рядов. Работать с ними можно с помощью атрибута `rolling()` объектов `Series` и `DataFrame`, возвращающего представление, подобное тому, с которым мы сталкивались при выполнении операции `groupby` (см. раздел «Агрегирование и группировка» данной главы). Это скользящее представление предоставляет по умолчанию несколько операций агрегирования.

Например, вот одногодичное скользящее среднее значение и стандартное отклонение цен на акции Google (рис. 3.10):

```
In[33]: rolling = goog.rolling(365, center=True)
        data = pd.DataFrame({'input': goog,
                             'one-year rolling_mean': rolling.mean(),
                             'one-year rolling_std': rolling.std()})
        ax = data.plot(style=['-', '--', ':'])
        ax.lines[0].set_alpha(0.3)
```

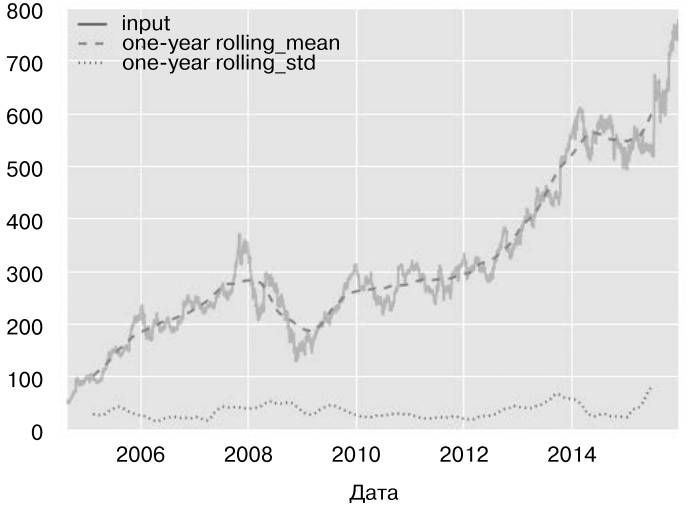

Рис. 3.10. Скользящие статистические показатели для цен на акции Google

Как и в случае операций `groupby`, можно использовать методы `aggregate()` и `apply()` для вычисления пользовательских скользящих показателей.

Где найти дополнительную информацию

В данном разделе приведена лишь краткая сводка некоторых наиболее важных возможностей инструментов для работы с временными рядами библиотеки Pandas. Более развернутое обсуждение этой темы можно найти в разделе Time Series/Date («Временные ряды/даты») онлайн-документации библиотеки Pandas (http://pandas.pydata.org/pandas-docs/stable/timeseries.html).

Еще один великолепный источник информации — руководство *Python for Data Analysis* издательства O'Reilly (http://shop.oreilly.com/product/0636920023784.do). Хотя ему уже несколько лет, это бесценный источник информации по использованию библиотеки Pandas. В частности, в книге сделан особый акцент на применении инструментов временных рядов в контексте бизнеса и финансов и уделено больше внимания конкретным деталям бизнес-календаря, работе с часовыми поясами и связанным с этим вопросам.

Вы также можете воспользоваться справочной функциональностью оболочки IPython для изучения и экспериментов с другими параметрами, имеющимися у обсуждавшихся здесь функций и методов. Я считаю, что это оптимальный способ изучения какого-либо нового инструмента языка Python.

Пример: визуализация количества велосипедов в Сиэтле

В качестве более сложного примера работы с данными временных рядов рассмотрим подсчет количества велосипедов на Фримонтском мосту в Сиэтле. Эти данные поступают из автоматического счетчика велосипедов, установленного в конце 2012 года с индуктивными датчиками на восточной и западной боковых дорожках моста. Сведения о почасовом количестве велосипедов можно скачать по адресу http://data.seattle.gov/; вот прямая ссылка на набор данных: https://data.seattle.gov/Transportation/Fremont-Bridge-Hourly-Bicycle-Counts-by-Month-Octo/65db-xm6k.

По состоянию на лето 2016 года CSV-файл можно скачать следующим образом:

```
In[34]:
# !curl -o FremontBridge.csv
# https://data.seattle.gov/api/views/65db-xm6k/rows.csv?accessType=DOWNLOAD
```

После скачивания набора данных можно воспользоваться библиотекой Pandas для чтения CSV-файла в объект `DataFrame`. Можно указать, что в качестве индекса мы хотим видеть объекты `Date` и чтобы выполнялся автоматический синтаксический разбор этих дат:

```
In[35]:
data = pd.read_csv('FremontBridge.csv', index_col='Date', parse_dates=True)
data.head()
```

```
Out[35]:                     Fremont Bridge West Sidewalk  \\
```

```
Date
2012-10-0300:00:00                          4.0
2012-10-0301:00:00                          4.0
2012-10-0302:00:00                          1.0
2012-10-0303:00:00                          2.0
2012-10-0304:00:00                          6.0

                     Fremont Bridge East Sidewalk
Date
2012-10-0300:00:00                          9.0
2012-10-0301:00:00                          6.0
2012-10-0302:00:00                          1.0
2012-10-0303:00:00                          3.0
2012-10-0304:00:00                          1.0
```

Для удобства мы подвергнем этот набор данных дальнейшей обработке, сократив названия столбцов и добавив столбец Total («Итого»):

```
In[36]: data.columns = ['West', 'East']
        data['Total'] = data.eval('West + East')
```

Теперь рассмотрим сводные статистические показатели для этих данных:

```
In[37]: data.dropna().describe()

Out[37]:                West          East         Total
         count  33544.000000  33544.000000  33544.000000
         mean      61.726568     53.541706    115.268275
         std       83.210813     76.380678    144.773983
         min        0.000000      0.000000      0.000000
         25%        8.000000      7.000000     16.000000
         50%       33.000000     28.000000     64.000000
         75%       80.000000     66.000000    151.000000
         max      825.000000    717.000000   1186.000000
```

Визуализация данных

Мы можем почерпнуть полезную информацию из этого набора данных, визуализировав его. Начнем с построения графика исходных данных (рис. 3.11).

```
In[38]: %matplotlib inline
        import seaborn; seaborn.set()

In[39]: data.plot()
        plt.ylabel('Hourly Bicycle Count'); # Количество велосипедов по часам
```

Примерно 25 000 почасовых выборок — слишком плотная дискретизация, чтобы можно было понять хоть что-то. Можно почерпнуть больше информации, если выполнить передискретизацию этих данных на сетке с более крупным шагом. Выполним передискретизацию с шагом одна неделя (рис. 3.12):

```
In[40]: weekly = data.resample('W').sum()
        weekly.plot(style=[':', '--', '-'])
        plt.ylabel('Weekly bicycle count'); # Количество велосипедов еженедельно
```

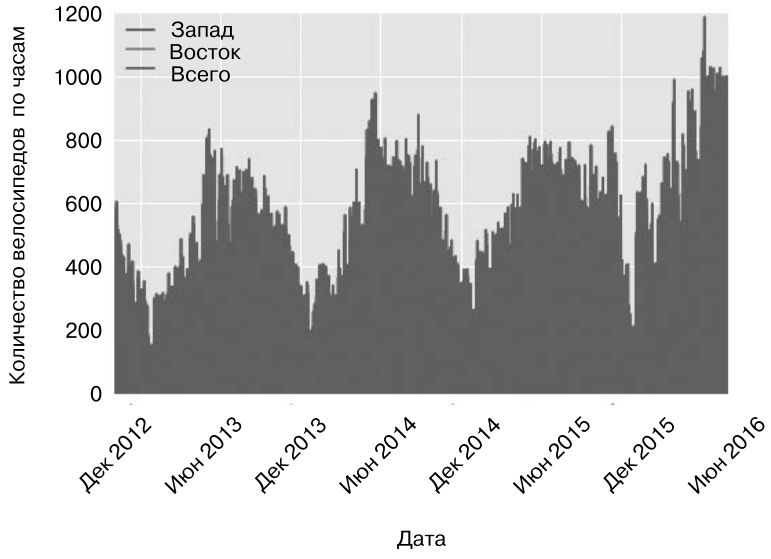

Рис. 3.11. Количество велосипедов за каждый час на Фримонтском мосту в Сиэтле

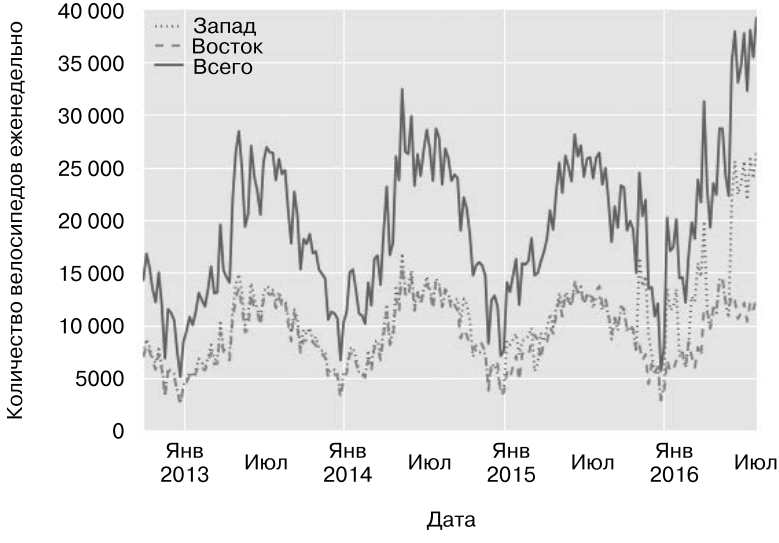

Рис. 3.12. Количество велосипедов, пересекающих Фримонтский мост в Сиэтле, с шагом одна неделя

Это демонстрирует нам некоторые интересные сезонные тренды: как и следовало ожидать, летом люди ездят на велосипедах больше, чем зимой, и даже в пределах каждого из сезонов велосипеды используются с разной интенсивностью в разные недели (вероятно, в зависимости от погоды; см. раздел «Заглянем глубже: линейная регрессия» главы 5, в котором будем рассматривать этот вопрос).

Еще один удобный способ агрегирования данных — вычисление скользящего среднего с помощью функции `pd.rolling_mean()`. Здесь мы вычисляем для наших данных скользящее среднее за 30 дней, центрируя при этом окно (рис. 3.13):

```
In[41]: daily = data.resample('D').sum()
        daily.rolling(30, center=True).sum().plot(style=[':', '--', '-'])
        plt.ylabel('mean hourly count'); # Среднее количество по часам
```

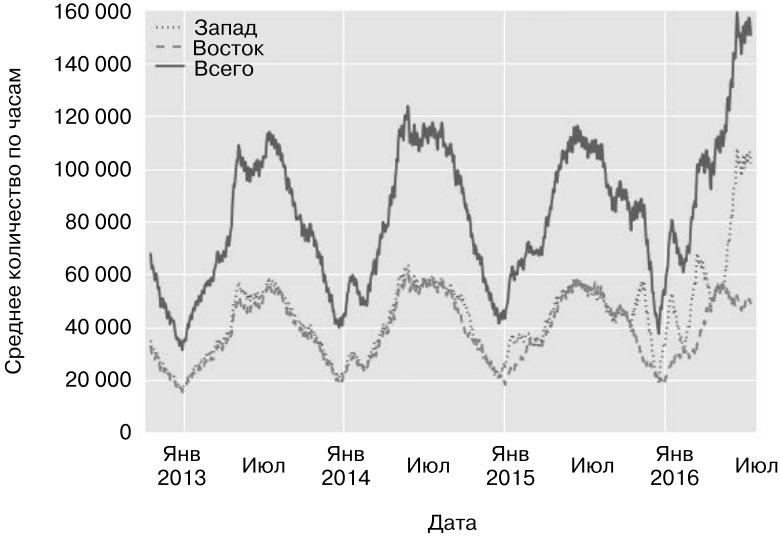

Рис. 3.13. Скользящее среднее значение еженедельного количества велосипедов

Причина зубчатости получившего изображения — в резкой границе окна. Более гладкую версию скользящего среднего можно получить, воспользовавшись оконной функцией, например Гауссовым окном. Следующий код (визуализированный на рис. 3.14) задает как ширину окна (в нашем случае 50 дней), так и ширину Гауссовой функции внутри окна (в нашем случае 10 дней):

```
In[42]:
daily.rolling(50, center=True,
              win_type='gaussian').sum(std=10).plot(style=[':', '--', '-']);
```

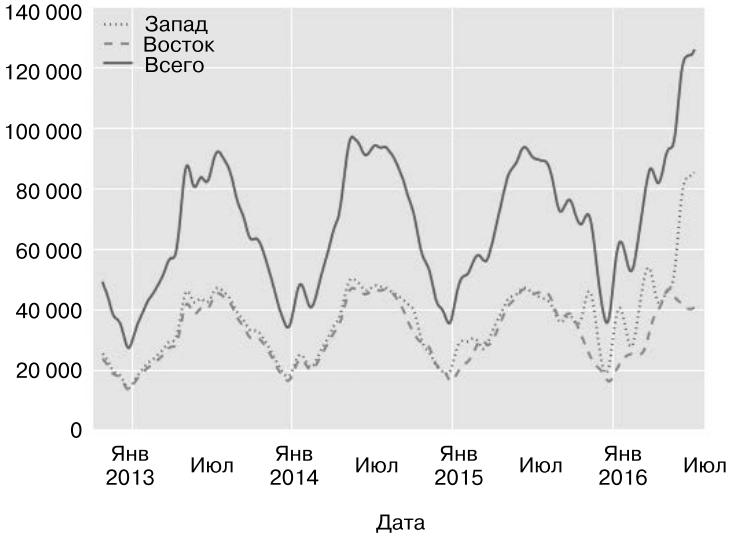

Рис. 3.14. Сглаженная Гауссова функция еженедельного количества велосипедов

Углубляемся в изучение данных

Хотя с помощью сглаженных представлений данных на рис. 3.14 можно получить общее представление о трендах данных, они скрывают от нас многие интересные нюансы их структуры. Например, нам может понадобиться увидеть усредненное движение велосипедного транспорта как функцию от времени суток. Это можно сделать с помощью функциональности `GroupBy`, обсуждавшейся в разделе «Агрегирование и группировка» данной главы (рис. 3.15):

```
In[43]: by_time = data.groupby(data.index.time).mean()
        hourly_ticks = 4 * 60 * 60 * np.arange(6)
        by_time.plot(xticks=hourly_ticks, style=[':', '--', '-']);
```

Почасовое движение транспорта представляет собой строго бимодальное распределение с максимумами в 08:00 утра и 05:00 вечера. Вероятно, это свидетельствует о существенном вкладе маятниковой миграции[1] через мост. В пользу этого говорят и различия между значениями с западной боковой дорожки (обычно используемой при движении в деловой центр Сиэтла) с более выраженными утренними максимумами и значениями с восточной боковой дорожки (обычно используемой при движении из делового центра Сиэтла) с более выраженными вечерними максимумами.

[1] См. https://ru.wikipedia.org/wiki/Маятниковая_миграция.

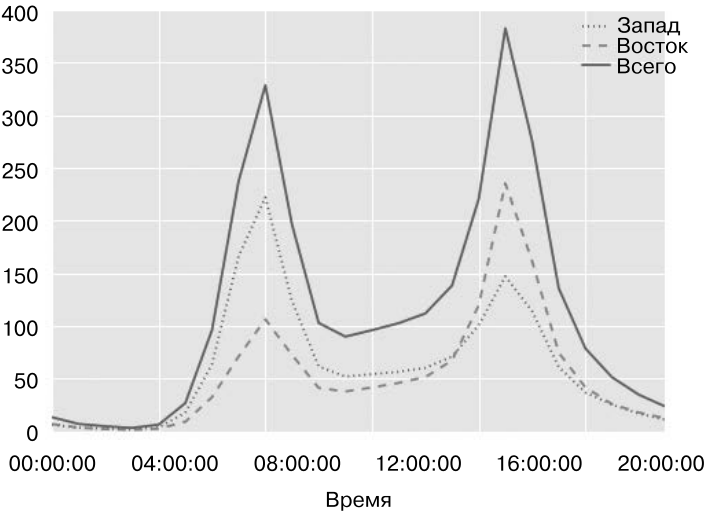

Рис. 3.15. Среднее почасовое количество велосипедов

Нас могут также интересовать изменения ситуации по дням недели. Это можно выяснить с помощью операции `groupby` (рис. 3.16):

```
In[44]: by_weekday = data.groupby(data.index.dayofweek).mean()
        by_weekday.index = ['Mon', 'Tues', 'Wed', 'Thurs',
                            'Fri', 'Sat', 'Sun']
        by_weekday.plot(style=[':', '--', '-']);
```

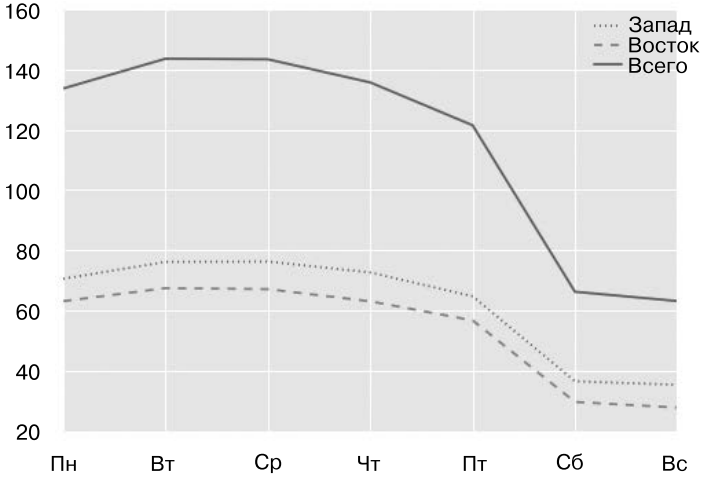

Рис. 3.16. Среднее количество велосипедов по дням

Этот график демонстрирует существенное различие между количеством велосипедов в будние и выходные дни: с понедельника по пятницу мост пересекает в среднем вдвое больше велосипедистов, чем в субботу и воскресенье.

С учетом этого выполним сложную операцию `groupby` и посмотрим на почасовой тренд в будни по сравнению с выходными. Начнем с группировки как по признаку выходного дня, так и по времени суток:

```
In[45]: weekend = np.where(data.index.weekday < 5, 'Weekday', 'Weekend')
        by_time = data.groupby([weekend, data.index.time]).mean()
```

Теперь воспользуемся некоторыми инструментами из раздела «Множественные субграфики» главы 4, чтобы нарисовать два графика бок о бок (рис. 3.17):

```
In[46]: import matplotlib.pyplot as plt
        fig, ax = plt.subplots(1, 2, figsize=(14, 5))
        by_time.ix['Weekday'].plot(ax=ax[0], title='Weekdays',
                                   xticks=hourly_ticks, style=[':', '--', '-'])
        by_time.ix['Weekend'].plot(ax=ax[1], title='Weekends',
                                   xticks=hourly_ticks, style=[':', '--', '-']);
```

Результат оказался очень интересным: мы видим бимодальный паттерн, связанный с поездками на работу в город на протяжении рабочей недели, и унимодальный паттерн, связанный с досугом/отдыхом во время выходных. Было бы интересно дальше покопаться в этих данных и изучить влияние погоды, температуры, времени года и других факторов на паттерны поездок в город на велосипедах. Дальнейшее обсуждение этих вопросов см. в сообщении «Действительно ли в Сиэтле наблюдается оживление в сфере поездок на велосипедах?» (https://jakevdp.github.io/blog/2014/06/10/is-seattle-really-seeing-an-uptick-in-cycling/) из моего блога, в котором используется подмножество этих данных. Мы также вернемся к этому набору данных в контексте моделирования в разделе «Заглянем глубже: линейная регрессия» главы 5.

Увеличение производительности библиотеки Pandas: eval() и query()

Основные возможности стека PyData основываются на умении библиотек NumPy и Pandas передавать простые операции на выполнение программам на языке C посредством интуитивно понятного синтаксиса: примерами могут послужить векторизованные/транслируемые операции в библиотеке NumPy, а также операции группировки в библиотеке Pandas. Хотя эти абстракции весьма производительны и эффективно работают для многих распространенных сценариев использования, они зачастую требуют создания вре́менных вспомогательных объектов, что приводит к чрезмерным накладным расходам как процессорного времени, так и оперативной памяти.

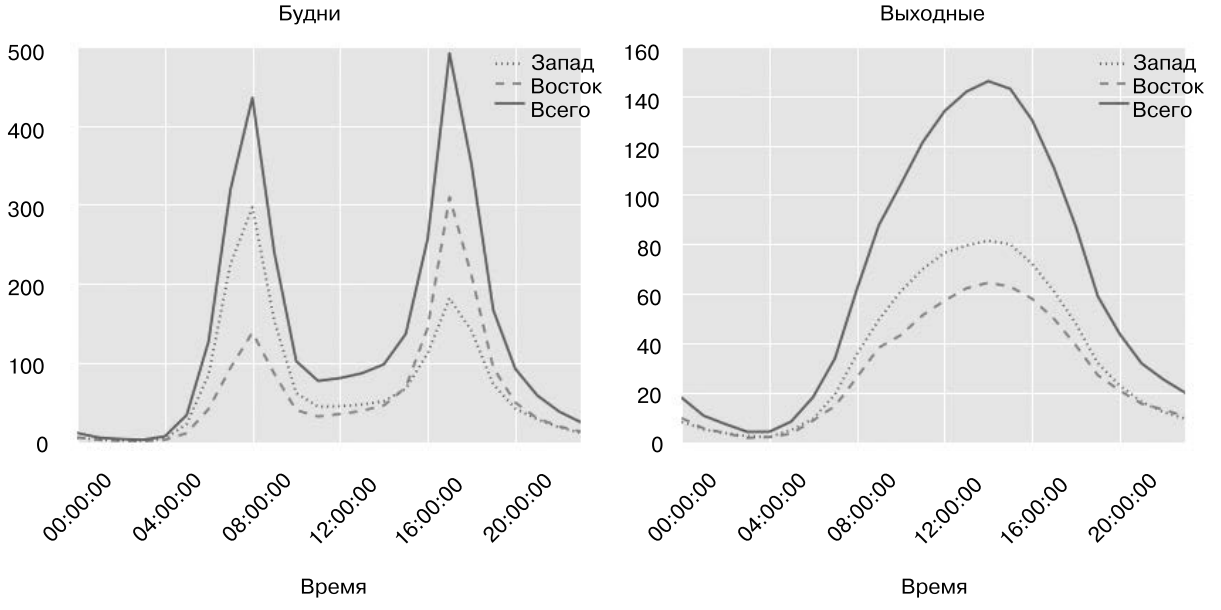

Рис. 3.17. Среднее количество велосипедов по часам, в рабочие и выходные дни

По состоянию на версию 0.13 (выпущенную в январе 2014 года) библиотека Pandas включает некоторые экспериментальные инструменты, позволяющие обращаться к работающим со скоростью написанных на языке C операциям без выделения существенных объемов памяти на промежуточные массивы. Эти утилиты — функции `eval()` и `query()`, основанные на пакете Numexpr (https://github.com/pydata/numexpr). Мы рассмотрим их использование и приведем некоторые эмпирические правила, позволяющие решить, имеет ли смысл их применять.

Основания для использования функций query() и eval(): составные выражения

Библиотеки NumPy и Pandas поддерживают выполнение быстрых векторизованных операций; например, при сложении элементов двух массивов:

```
In[1]: import numpy as np
       rng = np.random.RandomState(42)
       x = rng.rand(1E6)
       y = rng.rand(1E6)
       %timeit x + y 100 loops, best of 3: 3.39 ms per loop
```

Как уже обсуждалось в разделе «Выполнение вычислений над массивами библиотеки NumPy: универсальные функции» главы 2, такая операция выполняется гораздо быстрее, чем сложение с помощью цикла или спискового включения языка Python:

```
In[2]:
%timeit np.fromiter((xi + yi for xi, yi in zip(x, y)),
                    dtype=x.dtype, count=len(x))


1 loop, best of 3: 266 ms per loop
```

Однако данная абстракция оказывается менее эффективной при вычислении составных выражений. Например, рассмотрим выражение:

```
In[3]: mask = (x > 0.5) & (y < 0.5)
```

Поскольку библиотека NumPy вычисляет каждое подвыражение, оно эквивалентно следующему:

```
In[4]: tmp1 = (x > 0.5)
       tmp2 = (y < 0.5)
       mask = tmp1 & tmp2
```

Другими словами, *для каждого промежуточного шага явным образом выделяется оперативная память*. Если массивы `x` и `y` очень велики, это может привести к значительным накладным расходам оперативной памяти и процессорного времени.

Библиотека Numexpr позволяет вычислять подобные составные выражения поэлементно, не требуя выделения памяти под промежуточные массивы целиком. В документации библиотеки Numexpr (https://github.com/pydata/numexpr) приведено больше подробностей, но пока достаточно будет сказать, что функции этой библиотеки принимают на входе *строку*, содержащую выражение в стиле библиотеки NumPy, которое требуется вычислить:

```
In[5]: import numexpr
       mask_numexpr = numexpr.evaluate('(x > 0.5) & (y < 0.5)')
       np.allclose(mask, mask_numexpr)


Out[5]: True
```

Преимущество заключается в том, что библиотека Numexpr вычисляет выражение, не используя полноразмерных вре́менных массивов, а потому оказывается намного более эффективной, чем NumPy, особенно в случае больших массивов. Инструменты `query()` и `eval()`, которые мы будем обсуждать, идеологически схожи и используют пакет Numexpr.

Использование функции pandas.eval() для эффективных операций

Функция `eval()` библиотеки Pandas применяет строковые выражения для эффективных вычислительных операций с объектами `DataFrame`. Например, рассмотрим следующие объекты `DataFrame`:

```
In[6]: import pandas as pd
       nrows, ncols = 100000, 100
       rng = np.random.RandomState(42)
       df1, df2, df3, df4 = (pd.DataFrame(rng.rand(nrows, ncols))
                             for i in range(4))
```

Для вычисления суммы всех четырех объектов `DataFrame` при стандартном подходе библиотеки Pandas можно написать сумму:

```
In[7]: %timeit df1 + df2 + df3 + df4

10 loops, best of 3: 87.1 ms per loop
```

Можно вычислить тот же результат с помощью функции `pd.eval()`, задав выражение в виде строки:

```
In[8]: %timeit pd.eval('df1 + df2 + df3 + df4')

10 loops, best of 3: 42.2 ms per loop
```

Версия этого выражения с функцией `eval()` работает на 50 % быстрее (и использует намного меньше памяти), возвращая тот же самый результат:

```
In[9]: np.allclose(df1 + df2 + df3 + df4,
                   pd.eval('df1 + df2 + df3 + df4'))

Out[9]: True
```

Поддерживаемые функцией pd.eval() операции. На момент выпуска версии 0.16 библиотеки Pandas функция `pd.eval()` поддерживает широкий спектр операций. Для их демонстрации мы будем использовать следующие целочисленные объекты `DataFrame`:

```
In[10]: df1, df2, df3, df4, df5 = (pd.DataFrame(rng.randint(0, 1000, (100,
3)))
                                   for i in range(5))
```

Арифметические операторы. Функция `pd.eval()` поддерживает все арифметические операторы. Например:

```
In[11]: result1 = -df1 * df2 / (df3 + df4) - df5
        result2 = pd.eval('-df1 * df2 / (df3 + df4) - df5')
        np.allclose(result1, result2)

Out[11]: True
```

Операторы сравнения. Функция `pd.eval()` поддерживает все операторы сравнения, включая выражения, организованные цепочкой:

```
In[12]: result1 = (df1 < df2) & (df2 <= df3) & (df3 != df4)
        result2 = pd.eval('df1 < df2 <= df3 != df4')
        np.allclose(result1, result2)

Out[12]: True
```

Побитовые операторы. Функция `pd.eval()` поддерживает побитовые операторы `&` и `|`:

```
In[13]: result1 = (df1 < 0.5) & (df2 < 0.5) | (df3 < df4)
        result2 = pd.eval('(df1 < 0.5) & (df2 < 0.5) | (df3 < df4)')
        np.allclose(result1, result2)

Out[13]: True
```

Кроме того, она допускает использование литералов `and` и `or` в булевых выражениях:

```
In[14]: result3 = pd.eval('(df1 < 0.5) and (df2 < 0.5) or (df3 < df4)')
        np.allclose(result1, result3)

Out[14]: True
```

Атрибуты объектов и индексы. Функция `pd.eval()` поддерживает доступ к атрибутам объектов с помощью синтаксиса `obj.attr` и к индексам посредством синтаксиса `obj[index]`:

```
In[15]: result1 = df2.T[0] + df3.iloc[1]
        result2 = pd.eval('df2.T[0] + df3.iloc[1]')
        np.allclose(result1, result2)

Out[15]: True
```

Другие операции. Другие операции, например вызовы функций, условные выражения, циклы и другие, более сложные конструкции, пока в функции `pd.eval()` не реализованы. При необходимости выполнения подобных сложных видов выражений можно воспользоваться самой библиотекой Numexpr.

Использование метода DataFrame.eval() для выполнения операций по столбцам

У объектов `DataFrame` существует метод `eval()`, работающий схожим образом с высокоуровневой функцией `pd.eval()` из библиотеки Pandas. Преимущество метода `eval()` заключается в возможности ссылаться на столбцы *по имени*. Возьмем для примера следующий маркированный массив:

```
In[16]: df = pd.DataFrame(rng.rand(1000, 3), columns=['A', 'B', 'C'])
        df.head()

Out[16]:          A         B         C
         0  0.375506  0.406939  0.069938
         1  0.069087  0.235615  0.154374
         2  0.677945  0.433839  0.652324
         3  0.264038  0.808055  0.347197
         4  0.589161  0.252418  0.557789
```

Воспользовавшись функцией `pd.eval()` так, как показано выше, можно вычислять выражения с этими тремя столбцами следующим образом:

```
In[17]: result1 = (df['A'] + df['B']) / (df['C'] - 1)
        result2 = pd.eval("(df.A + df.B) / (df.C - 1)")
        np.allclose(result1, result2)

Out[17]: True
```

Метод `DataFrame.eval()` позволяет описывать вычисления со столбцами гораздо лаконичнее:

```
In[18]: result3 = df.eval('(A + B) / (C - 1)')
        np.allclose(result1, result3)

Out[18]: True
```

Обратите внимание, что мы *обращаемся с названиями столбцов* в вычисляемом выражении *как с переменными* и получаем желаемый результат.

Присваивание в методе DataFrame.eval()

Метод `DataFrame.eval()` позволяет выполнять присваивание значения любому из столбцов. Воспользуемся предыдущим объектом `DataFrame` со столбцами `'A'`, `'B'` и `'C'`:

```
In[19]: df.head()

Out[19]:          A         B         C
         0  0.375506  0.406939  0.069938
         1  0.069087  0.235615  0.154374
         2  0.677945  0.433839  0.652324
         3  0.264038  0.808055  0.347197
         4  0.589161  0.252418  0.557789
```

Метод `DataFrame.eval()` можно использовать, например, для создания нового столбца `'D'` и присваивания ему значения, вычисленного на основе других столбцов:

```
In[20]: df.eval('D = (A + B) / C', inplace=True)
        df.head()
Out[20]:          A         B         C          D
         0  0.375506  0.406939  0.069938  11.187620
         1  0.069087  0.235615  0.154374   1.973796
         2  0.677945  0.433839  0.652324   1.704344
         3  0.264038  0.808055  0.347197   3.087857
         4  0.589161  0.252418  0.557789   1.508776
```

Аналогично можно модифицировать значения любого уже существующего столбца:

```
In[21]: df.eval('D = (A - B) / C', inplace=True)
        df.head()

Out[21]:          A         B         C         D
         0  0.375506  0.406939  0.069938 -0.449425
         1  0.069087  0.235615  0.154374 -1.078728
         2  0.677945  0.433839  0.652324  0.374209
         3  0.264038  0.808055  0.347197 -1.566886
         4  0.589161  0.252418  0.557789  0.603708
```

Локальные переменные в методе DataFrame.eval()

Метод `DataFrame.eval()` поддерживает дополнительный синтаксис для работы с локальными переменными языка Python. Взгляните на следующий фрагмент кода:

```
In[22]: column_mean = df.mean(1)
        result1 = df['A'] + column_mean
        result2 = df.eval('A + @column_mean')
        np.allclose(result1, result2)

Out[22]: True
```

Символ `@` отмечает *имя переменной*, а не *имя столбца*, позволяя тем самым эффективно вычислять значение выражений с использованием двух пространств имен: пространства имен столбцов и пространства имен объектов Python. Обратите внимание, что этот символ `@` поддерживается лишь *методом* `DataFrame.eval()`, но не *функцией* `pandas.eval()`, поскольку у функции `pandas.eval()` есть доступ только к одному пространству имен (языка Python).

Метод DataFrame.query()

У объектов `DataFrame` есть еще один метод, основанный на вычислении строк, именуемый `query()`. Рассмотрим следующее:

```
In[23]: result1 = df[(df.A < 0.5) & (df.B < 0.5)]
        result2 = pd.eval('df[(df.A < 0.5) & (df.B < 0.5)]')
        np.allclose(result1, result2)

Out[23]: True
```

Как и в случае с примером, который мы использовали при обсуждении метода `DataFrame.eval()`, перед нами выражение, содержащее столбцы объекта `DataFrame`. Однако его нельзя выразить с помощью синтаксиса метода `DataFrame.eval()`! Вместо него для подобных операций фильтрации можно воспользоваться методом `query()`:

```
In[24]: result2 = df.query('A < 0.5 and B < 0.5')
        np.allclose(result1, result2)

Out[24]: True
```

Он не только обеспечивает, по сравнению с выражениями маскирования, более эффективные вычисления, но и намного понятнее и легче читается. Обратите внимание, что метод `query()` также позволяет использовать флаг `@` для обозначения локальных переменных:

```
In[25]: Cmean = df['C'].mean()
        result1 = df[(df.A < Cmean) & (df.B < Cmean)]
        result2 = df.query('A < @Cmean and B < @Cmean')
        np.allclose(result1, result2)

Out[25]: True
```

Производительность: когда следует использовать эти функции

В процессе принятия решения применять ли эти функции обратите внимание на два момента: *процессорное время* и *объем используемой памяти*. Предсказать объем используемой памяти намного проще. Как уже упоминалось, все составные

выражения с применением массивов NumPy или объектов `DataFrame` библиотеки Pandas приводят к неявному созданию вре́менных массивов. Например, вот это:

```
In[26]: x = df[(df.A < 0.5) & (df.B < 0.5)]
```

приблизительно соответствует следующему:

```
In[27]: tmp1 = df.A < 0.5
        tmp2 = df.B < 0.5
        tmp3 = tmp1 & tmp2
        x = df[tmp3]
```

Если размер вре́менных объектов `DataFrame` существенен по сравнению с доступной оперативной памятью вашей системы (обычно несколько гигабайтов), то будет разумно воспользоваться выражениями `eval()` или `query()`. Выяснить приблизительный размер массива в байтах можно с помощью следующего оператора:

```
In[28]: df.values.nbytes

Out[28]: 32000
```

`eval()` будет работать быстрее, если вы не используете всю доступную в системе оперативную память. Основную роль играет отношение размера вре́менных объектов `DataFrame` по сравнению с размером L1 или L2 кэша процессора в системе (в 2016 году он составляет несколько мегабайтов). `eval()` позволяет избежать потенциально медленного перемещения значений между различными кэшами памяти в том случае, когда это отношение намного больше 1. Я обнаружил, что на практике различие в скорости вычислений между традиционными методами и методом `eval/query` обычно довольно незначительно. Напротив, традиционный метод работает быстрее для маленьких массивов! Преимущество метода `eval/query` заключается в экономии оперативной памяти и иногда — в более понятном синтаксисе.

Мы рассмотрели большинство нюансов работы с методами `eval()` и `query()`, дополнительную информацию можно найти в документации по библиотеке Pandas. В частности, можно задавать для работы этих запросов различные синтаксические анализаторы и механизмы. Подробности — в разделе Enhancing Performance («Повышение производительности», http://pandas.pydata.org/pandas-docs/dev/enhancingperf.html).

Дополнительные источники информации

В этой главе мы охватили большую часть основ эффективного использования библиотеки Pandas для анализа данных. Чтобы изучить библиотеку Pandas глубже, я бы рекомендовал обратиться к следующим источникам информации.

- *Онлайн-документация библиотеки Pandas* (http://pandas.pydata.org/). Это всесторонний источник документации по данному пакету. Хотя примеры в документации обычно представляют собой небольшие сгенерированные наборы

данных, параметры описываются во всей полноте, что обычно очень удобно для понимания того, как использовать различные функции.

- *Python for Data Analysis*[1] (http://bit.ly/python-for-data-analysis). В книге, написанной Уэсом Маккинни, углубленно рассматриваются инструменты для работы с временными рядами, обеспечившие его как финансового аналитика средствами к существованию. В книге также приведено множество интересных примеров применения Python для получения полезной информации из реальных наборов данных. Не забывайте, что этой книге уже несколько лет и с того времени в пакете Pandas появилось немало новых возможностей, не охваченных ею (впрочем, в 2017 году[2] ожидается новое издание).
- *Обсуждение библиотеки Pandas на форуме Stack Overflow*. У библиотеки Pandas столько пользователей, что любой вопрос, который только может у вас возникнуть, вероятно, уже был задан на форуме Stack Overflow и на него получен ответ. При использовании библиотеки Pandas вашими лучшими друзьями станут поисковые системы. Просто введите свой вопрос, сформулируйте проблему или ошибку, с которой вы столкнулись, — более чем вероятно, что вы найдете решение на одной из страниц сайта Stack Overflow.
- *Библиотека Pandas на сайте PyVideo* (http://pyvideo.org/tag/pandas/). Многие форумы, начиная от PyCon, SciPy и до PyData, выпускали руководства от разработчиков и опытных пользователей библиотеки Pandas. Презентаторы, создающие руководства PyCon, наиболее опытные и профессиональные.

Надеюсь, что с этими источниками информации в сочетании с полученной в данной главе демонстрацией возможностей вы будете готовы справиться с помощью библиотеки Pandas с любой задачей по анализу данных, какая вам только встретится!

---

[1] *Маккинни У.* Python и анализ данных. — М.: ДМК-Пресс, 2015. — 482 с.

[2] По предварительным данным, она должна выйти 25 августа 2017 года. В настоящее время (февраль 2017 года) на сайте издательства O'Reilly доступна предварительная версия в виде электронной книги.

4 Визуализация с помощью библиотеки Matplotlib

Matplotlib — мультиплатформенная библиотека для визуализации данных, основанная на массивах библиотеки NumPy и спроектированная в расчете на работу с обширным стеком SciPy. Она была задумана Джоном Хантером в 2002 году и изначально представляла собой патч к оболочке IPython, предназначенный для реализации возможности интерактивного построения с помощью утилиты gnuplot графиков в стиле MATLAB из командной строки IPython. Создатель оболочки IPython Фернандо Перес в этот момент был занят завершением написания диссертации, он сообщил Джону, что в ближайшие несколько месяцев у него не будет времени на анализ патча. Хантер принял это как благословение на самостоятельную разработку — так родился пакет Matplotlib, версия 0.1 которого была выпущена в 2003 году. Институт исследований космоса с помощью космического телескопа (Space Telescope Science Institute, занимающийся управлением телескопом «Хаббл») финансово поддержал разработку пакета Matplotlib и обеспечил расширение его возможностей, избрав в качестве пакета для формирования графических изображений.

Одна из важнейших возможностей пакета Matplotlib — хорошая совместимость с множеством операционных систем и графических прикладных частей. Matplotlib поддерживает десятки прикладных частей и типов вывода, а значит, можно полагаться на него независимо от используемой операционной системы или требуемого формата вывода. Самая сильная сторона пакета Matplotlib — кросс-платформенный подход типа «все для всех», который привел к росту пользователей, что, в свою очередь, стало причиной появления большого числа активных разработчиков, увеличения возможностей инструментов пакета Matplotlib и его распространенности в мире научных вычислений на языке Python.

В последние годы интерфейс и стиль библиотеки Matplotlib начали несколько устаревать. На фоне новых утилит, таких как ggplot и ggvis в языке R, а также наборов веб-инструментов визуализации, основанных на холстах D3js и HTML5, она кажется

неуклюжей и старомодной. Тем не менее, я полагаю, нельзя игнорировать возможности библиотеки Matplotlib — надежного кросс-платформенного графического механизма. Свежие версии Matplotlib упрощают настройку новых глобальных стилей вывода графики (см. раздел «Пользовательские настройки Matplotlib: конфигурации и таблицы стилей» данной главы). Разрабатываются новые пакеты, предназначенные для работы с ней через более современные и «чистые» API, например Seaborn (см. раздел «Визуализация с помощью библиотеки Seaborn» этой главы), ggplot (http://yhat.github.io/ggplot), HoloViews (http://holoviews.org/), Altair (http://altair-viz.github.io/) и даже саму библиотеку Pandas можно использовать в качестве адаптеров для API Matplotlib. Однако даже при наличии подобных адаптеров полезно разобраться в синтаксисе Matplotlib для настройки вывода итогового графика. Исходя из этого, я считаю, что сама библиотека Matplotlib остается жизненно важной частью стека визуализации данных, даже если новые инструменты приводят к тому, что сообщество пользователей постепенно отходит от непосредственного использования API Matplotlib.

Общие советы по библиотеке Matplotlib

Прежде чем погрузиться в подробности создания визуализаций с помощью Matplotlib, расскажу несколько полезных вещей про этот пакет.

Импорт matplotlib

Аналогично тому, как мы использовали сокращение `np` для библиотеки NumPy и сокращение `pd` для библиотеки Pandas, мы будем применять стандартные сокращения для импортов библиотеки Matplotlib:

```
In[1]: import matplotlib as mpl
       import matplotlib.pyplot as plt
```

Чаще всего мы будем использовать интерфейс `plt`.

Настройка стилей

Для выбора подходящих стилей для наших графиков мы будем применять директиву `plt.style`. В следующем фрагменте кода мы задаем директиву `classic`, которая обеспечит в создаваемых нами графиках классический стиль библиотеки Matplotlib:

```
In[2]: plt.style.use('classic')
```

В данном разделе мы будем настраивать этот стиль по мере необходимости. Обратите внимание, что таблицы стилей поддерживаются версией 1.5 библиотеки Matplotlib. В более ранних версиях доступен только стиль по умолчанию. Дальнейшую информацию о таблицах стилей см. в разделе «Пользовательские настройки Matplotlib: конфигурации и таблицы стилей» этой главы.

Использовать show() или не использовать? Как отображать свои графики

Визуализация, которую не видно, особой пользы не несет, но то, в каком виде вы увидите графики библиотеки Matplotlib, зависит от контекста. Имеется три возможных контекста:

- использование Matplotlib в сценарии;
- в терминале оболочки IPython;
- в блокноте IPython.

Построение графиков из сценария

Функция `plt.show()`будет полезна при использовании библиотеки Matplotlib изнутри сценария. Она запускает цикл ожидания события, ищет все активные в настоящий момент объекты графиков и открывает одно или несколько интерактивных окон для отображения вашего графика (графиков).

Допустим, у вас имеется файл `myplot.py`, содержащий следующее:

```
# ------- файл: myplot.py ------
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 10, 100)

plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))

plt.show()
```

Далее можно запустить этот сценарий из командной строки, что приведет к открытию окна с вашим графиком:

```
$ python myplot.py
```

Команда `plt.show()` выполняет «под капотом» много разной работы, так как ей необходимо взаимодействовать с интерактивной графической прикладной частью вашей системы. Детали этой операции различаются в зависимости от операционной системы и конкретной версии, но библиотека Matplotlib делает все возможное, чтобы скрыть от вас эти детали.

Одно важное замечание: команду `plt.show()` следует использовать только *один раз* за сеанс работы с Python, и чаще всего ее можно увидеть в самом конце сценария. Выполнение нескольких команд `show()` может привести к непредсказуемому поведению в зависимости от прикладной части, так что лучше избегать этого.

Построение графиков из командной оболочки IPython

Очень удобно использовать Matplotlib интерактивно из командной оболочки IPython (см. главу 1). Оболочка IPython будет отлично работать с библиотекой Matplotlib, если перевести ее в режим Matplotlib. Для активизации этого режима после запуска `IPython` можно воспользоваться «магической» командой `%matplotlib`:

```
In [1]: %matplotlib
Using matplotlib backend: TkAgg

In [2]: import matplotlib.pyplot as plt
```

После любая команда `plot` приведет к открытию окна графика с возможностью выполнения дальнейших команд для его изменения. Некоторые изменения (например, модификация свойств уже нарисованных линий) не будут отрисовываться автоматически. Чтобы добиться этого, воспользуйтесь командой `plt.draw()`. Выполнять команду `plt.show()` в режиме Matplotlib не обязательно.

Построение графиков из блокнота IPython

Блокнот IPython — браузерный интерактивный инструмент для анализа данных, допускающий совмещение комментариев, кода, графики, элементов HTML и многого другого в единый исполняемый документ (см. главу 1).

Интерактивное построение графиков в блокноте IPython возможно с помощью команды `%matplotlib`, работает аналогично командной оболочке IPython. В блокноте IPython у вас появляется возможность включения графики непосредственно в блокнот с двумя возможными альтернативами:

- использование команды `%matplotlib notebook` приведет к включению в блокнот *интерактивных* графиков;
- выполнение команды `%matplotlib inline` приведет к включению в блокнот статических изображений графиков.

В книге мы будем использовать команду `%matplotlib inline`:

```
In[3]: %matplotlib inline
```

После выполнения этой команды (которое нужно произвести только один раз за сеанс/для одного ядра Python) все создающие графики блоки в блокноте будут включать PNG-изображения итогового графика (рис. 4.1):

```
In[4]: import numpy as np
       x = np.linspace(0, 10, 100)

       fig = plt.figure()
       plt.plot(x, np.sin(x), '-')
       plt.plot(x, np.cos(x), '--');
```

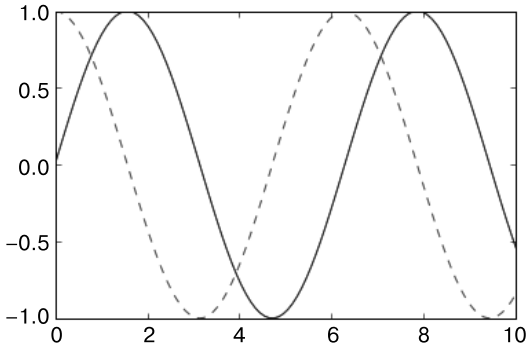

Рис. 4.1. Простейший пример построения графиков

Сохранение рисунков в файл

Умение сохранять рисунки в файлы различных форматов — одна из возможностей библиотеки Matplotlib. Например, сохранить предыдущий рисунок в файл PNG можно с помощью команды `savefig()`:

```
In[5]: fig.savefig('my_figure.png')
```

В текущем рабочем каталоге появился файл с названием `my_figure.png`:

```
In[6]: !ls -lh my_figure.png
```

```
-rw-r--r--  1 jakevdp  staff    16K Aug 1110:59 my_figure.png
```

Чтобы убедиться, что содержимое этого файла соответствует нашим ожиданиям, воспользуемся объектом `Image` оболочки IPython для отображения его содержимого (рис. 4.2):

```
In[7]: from IPython.display import Image
       Image('my_figure.png')
```

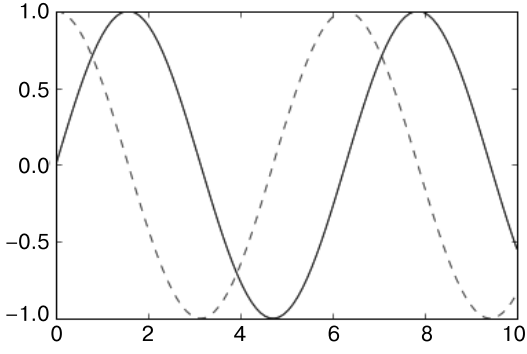

Рис. 4.2. Простой график в виде PNG

Команда `savefig()` определяет формат файла, исходя из расширения заданного имени файла. В зависимости от установленной в вашей системе прикладной части может поддерживаться множество различных форматов файлов. Вывести список поддерживаемых форматов файлов для вашей системы вы можете с помощью следующего метода объекта `canvas` рисунка:

```
In[8]: fig.canvas.get_supported_filetypes()

Out[8]: {'eps': 'Encapsulated Postscript',
         'jpeg': 'Joint Photographic Experts Group',
         'jpg': 'Joint Photographic Experts Group',
         'pdf': 'Portable Document Format',
         'pgf': 'PGF code for LaTeX',
         'png': 'Portable Network Graphics',
         'ps': 'Postscript',
         'raw': 'Raw RGBA bitmap',
         'rgba': 'Raw RGBA bitmap',
         'svg': 'Scalable Vector Graphics',
         'svgz': 'Scalable Vector Graphics',
         'tif': 'Tagged Image File Format',
         'tiff': 'Tagged Image File Format'}
```

Обратите внимание, что при сохранении рисунка не обязательно использовать команду `plt.show()` или другие команды, обсуждавшиеся ранее.

Два интерфейса по цене одного

Два интерфейса библиотеки Matplotlib (удобный MATLAB-подобный интерфейс, основанный на сохранении состояния, и обладающий большими возможностями объектно-ориентированный интерфейс) — свойство, которое потенциально может привести к путанице. Рассмотрим вкратце различия между ними.

Интерфейс в стиле MATLAB

Библиотека Matplotlib изначально была написана как альтернативный вариант (на языке Python) для пользователей пакета MATLAB, и значительная часть ее синтаксиса отражает этот факт. MATLAB-подобные инструменты содержатся в интерфейсе `pyplot` (`plt`). Например, следующий код, вероятно, выглядит довольно знакомо пользователям MATLAB (рис. 4.3):

```
In[9]: plt.figure()  # Создаем рисунок для графика

       # Создаем первую из двух областей графика и задаем текущую ось
       plt.subplot(2, 1, 1) # (rows, columns, panel number)
       plt.plot(x, np.sin(x))
```

```
        # Создаем вторую область и задаем текущую ось
        plt.subplot(2, 1, 2)
        plt.plot(x, np.cos(x));
```

Важно отметить, что этот интерфейс *сохраняет состояние*: он отслеживает текущий рисунок и его оси координат и для него выполняет все команды `plt`. Получить на них ссылки можно с помощью команд `plt.gcf()` (от англ. get current figure — «получить текущий рисунок») и `plt.gca()` (от англ. get current axes — «получить текущие оси координат»).

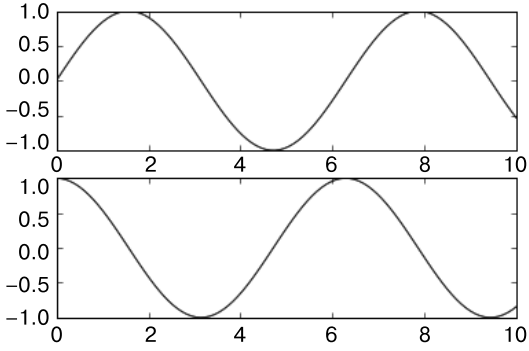

Рис. 4.3. Субграфики, созданные с помощью MATLAB-подобного интерфейса

Хотя в случае простых графиков этот интерфейс с сохранением состояния быстр и удобен, его использование может привести к проблемам. Например, как после создания второй области рисунка вернуться в первую и добавить что-либо в ней. Сделать это в MATLAB-подобном интерфейсе можно, но довольно громоздким способом. Существует лучший вариант.

Объектно-ориентированный интерфейс

Объектно-ориентированный интерфейс подходит для более сложных ситуаций, когда вам требуется больше возможностей управления рисунком. В объектно-ориентированном интерфейсе функции рисования не полагаются на понятие текущего рисунка или осей, а являются *методами* явным образом определяемых объектов `Figure` и `Axes`. Чтобы перерисовать предыдущий рисунок с его помощью, можно сделать следующее (рис. 4.4):

```
In[10]: # Сначала создаем сетку графиков
        # ax будет массивом из двух объектов Axes
        fig, ax = plt.subplots(2)

        # Вызываем метод plot() соответствующего объекта
        ax[0].plot(x, np.sin(x))
        ax[1].plot(x, np.cos(x));
```

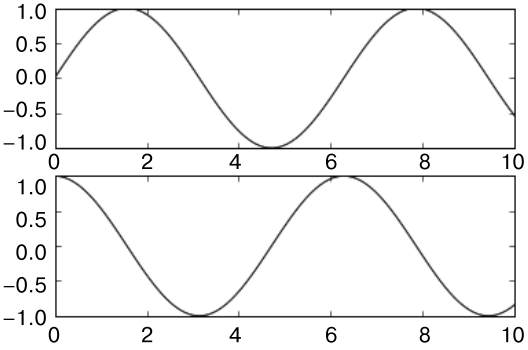

Рис. 4.4. Субграфики, созданные с помощью объектно-ориентированного интерфейса

В случае более простых графиков выбор интерфейса в основном вопрос личных предпочтений, но по мере усложнения графиков объектно-ориентированный подход становится необходимостью. В этой главе мы будем переключаться между MATLAB-подобным и объектно-ориентированным интерфейсами в зависимости от того, какой из них удобнее для конкретной задачи. В большинстве случаев в коде приходится всего лишь заменить `plt.plot()` на `ax.plot()` и не более того, но есть несколько нюансов, на которые мы будем обращать внимание в следующих разделах.

Простые линейные графики

Вероятно, простейшим из всех графиков является визуализация отдельной функции $y = f(x)$. В этом разделе мы рассмотрим создание простого графика такого типа. Как и во всех следующих разделах, начнем с настройки блокнота для построения графиков и импорта функций, которые будем использовать:

```
In[1]: %matplotlib inline
       import matplotlib.pyplot as plt
       plt.style.use('seaborn-whitegrid')
       import numpy as np
```

Во всех графиках Matplotlib мы начинаем с создания рисунка и системы координат. В простейшем случае рисунок и систему координат можно создать следующим образом (рис. 4.5):

```
In[2]: fig = plt.figure()
       ax = plt.axes()
```

В библиотеке Matplotlib можно рассматривать *рисунок* (экземпляр класса `plt.Figure`) как единый контейнер, содержащий все объекты, представляющие систему координат, графику, текст и метки. Система координат (она же — *оси координат*, экземпляры класса `plt.Axes`) — то, что вы видите выше: ограничивающий

прямоугольник с делениями и метками, в котором потом будут находиться составляющие нашу визуализацию элементы графика. В этой книге мы будем использовать имя переменной `fig` для экземпляра рисунка и `ax` для экземпляров системы координат или группы экземпляров систем координат.

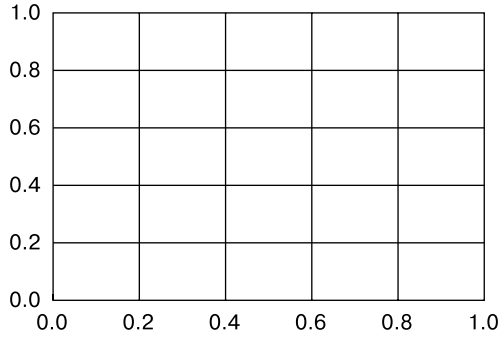

Рис. 4.5. Пустая система координат с координатными осями

После создания осей можно применить функцию `ax.plot` для построения графика данных. Начнем с простой синусоиды (рис. 4.6):

```
In[3]: fig = plt.figure()
       ax = plt.axes()

       x = np.linspace(0, 10, 1000)
       ax.plot(x, np.sin(x));
```

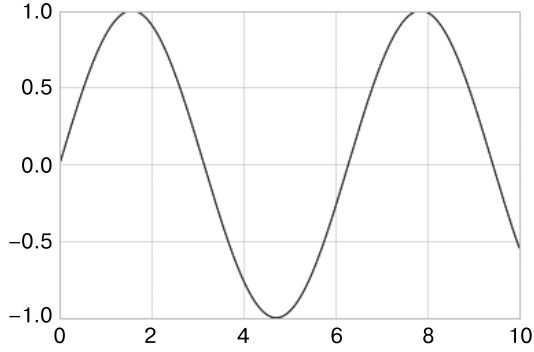

Рис. 4.6. Простая синусоида

Мы могли бы воспользоваться и интерфейсом `pylab`, при этом рисунок и система координат были бы созданы в фоновом режиме (рис. 4.7, см. раздел «Два интерфейса по цене одного» данной главы, в котором обсуждаются эти два интерфейса):

```
In[4]: plt.plot(x, np.sin(x));
```

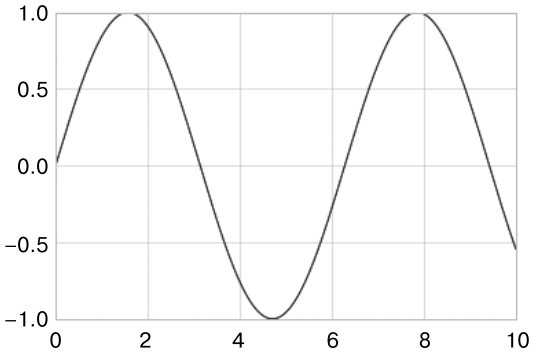

Рис. 4.7. Построение графика простой синусоиды с помощью объектно-ориентированного интерфейса

Если нужно создать простой рисунок с несколькими линиями, можно вызвать функцию `plot` несколько раз (рис. 4.8):

```
In[5]: plt.plot(x, np.sin(x))
       plt.plot(x, np.cos(x));
```

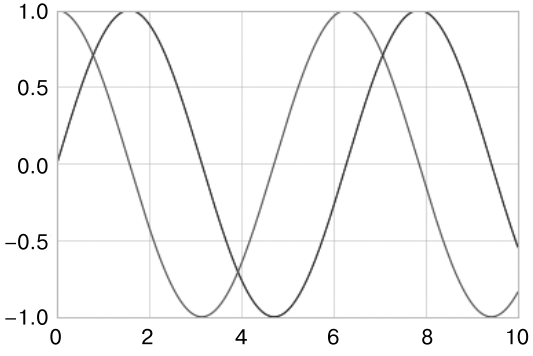

Рис. 4.8. Рисуем несколько линий

Вот и все, что касается построения графиков простых функций в библиотеке Matplotlib! Теперь углубимся в некоторые подробности управления внешним видом осей и линий.

Настройка графика: цвета и стили линий

Первое, что вы можете захотеть сделать с графиком, — научиться управлять цветами и стилями линий. Функция `plt.plot` принимает дополнительные аргументы, которыми можно воспользоваться для этой цели. Для настройки цвета используйте

ключевое слово `color`, которому ставится в соответствие строковый аргумент, задающий практически любой цвет. Задать цвет можно разными способами (рис. 4.9):

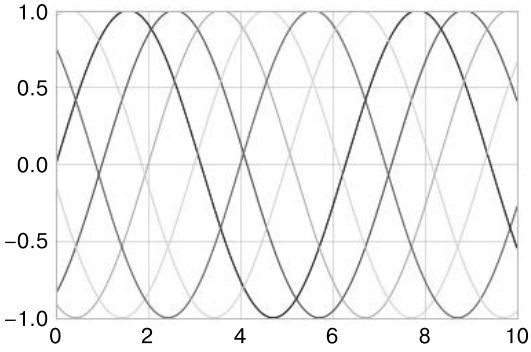

Рис. 4.9. Управление цветом элементов графика

```
In[6]:
plt.plot(x, np.sin(x - 0), color='blue')    # Задаем цвет по названию
plt.plot(x, np.sin(x - 1), color='g')       # Краткий код цвета (rgbcmyk)
plt.plot(x, np.sin(x - 2), color='0.75')    # Шкала оттенков серого цвета,
                                            # значения в диапазоне от 0 до 1
plt.plot(x, np.sin(x - 3), color='#FFDD44')     # 16-ричный код
                                                # (RRGGBB от 00 до FF)
plt.plot(x, np.sin(x - 4), color=(1.0,0.2,0.3)) # Кортеж RGB, значения 0 и 1
plt.plot(x, np.sin(x - 5), color='chartreuse'); # Поддерживаются все названия
                                                # цветов HTML
```

Если цвет не задан, библиотека Matplotlib будет автоматически перебирать по циклу набор цветов по умолчанию при наличии на графике нескольких линий.

Стиль линий можно настраивать и с помощью ключевого слова `linestyle` (рис. 4.10):

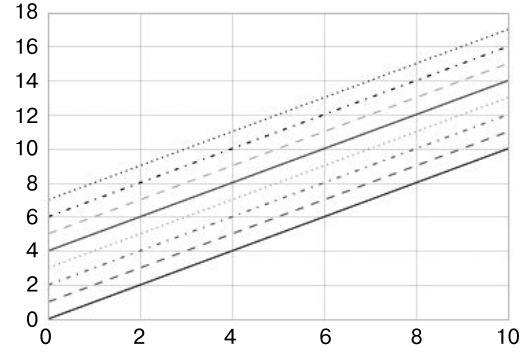

Рис. 4.10. Примеры различных стилей линий

```
In[7]: plt.plot(x, x + 0, linestyle='solid')
       plt.plot(x, x + 1, linestyle='dashed')
       plt.plot(x, x + 2, linestyle='dashdot')
       plt.plot(x, x + 3, linestyle='dotted');

       # Можно использовать и следующие сокращенные коды:
       plt.plot(x, x + 4, linestyle='-')  # сплошная линия
       plt.plot(x, x + 5, linestyle='--') # штриховая линия
       plt.plot(x, x + 6, linestyle='-.') # штрихпунктирная линия
       plt.plot(x, x + 7, linestyle=':'); # пунктирная линия
```

Если вы предпочитаете максимально сжатый синтаксис, можно объединить задание кодов `linestyle` и `color` в одном неключевом аргументе функции `plt.plot` (рис. 4.11):

```
In[8]: plt.plot(x, x + 0, '-g')  # сплошная линия зеленого цвета
       plt.plot(x, x + 1, '--c') # штриховая линия голубого цвета
       plt.plot(x, x + 2, '-.k') # штрихпунктирная линия черного цвета
       plt.plot(x, x + 3, ':r'); # пунктирная линия красного цвета
```

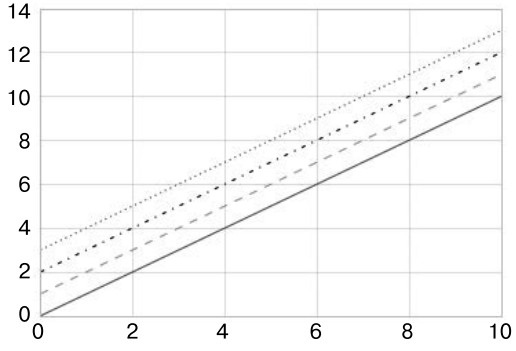

Рис. 4.11. Сокращенный синтаксис для управления цветами и стилями

Эти односимвольные коды цветов отражают стандартные сокращения, принятые в широко используемых для цифровой цветной графики цветовых моделях RGB (Red/Green/Blue — «красный/зеленый/синий») и CMYK (Cyan/Magenta/Yellow/blacK — «голубой/пурпурный/желтый/черный»).

Существует множество других ключевых аргументов, позволяющих выполнять более тонкую настройку внешнего вида графика. Чтобы узнать больше, рекомендую посмотреть docstring функции `plt.plot()` с помощью справочных инструментов оболочки IPython (см. раздел «Справка и документация в оболочке Python» главы 1).

Настройка графика: пределы осей координат

Библиотека Matplotlib достаточно хорошо подбирает пределы осей координат по умолчанию, но иногда требуется более точная настройка. Простейший

способ настройки пределов осей координат — методы `plt.xlim()` и `plt.ylim()` (рис. 4.12):

```
In[9]: plt.plot(x, np.sin(x))

       plt.xlim(-1, 11)
       plt.ylim(-1.5, 1.5);
```

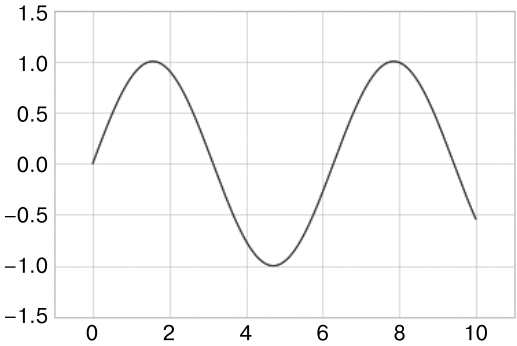

Рис. 4.12. Пример задания пределов осей координат

Если вам нужно, чтобы оси отображались зеркально, можно указать аргументы в обратном порядке (рис. 4.13):

```
In[10]: plt.plot(x, np.sin(x))

        plt.xlim(10, 0)
        plt.ylim(1.2, -1.2);
```

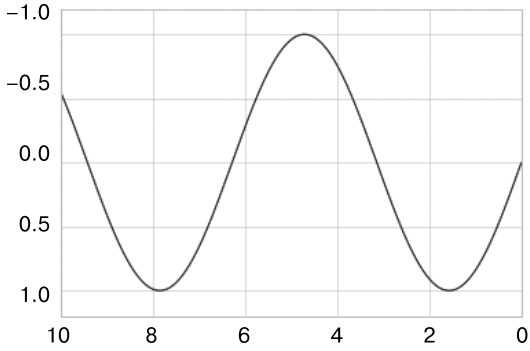

Рис. 4.13. Пример зеркального отображения оси *Y*

Удобный метод для этих действий — `plt.axis()` (не перепутайте метод `plt.axis()` с методом `plt.axes()`!). Метод `plt.axis()` предоставляет возможность задавать

пределы осей X и Y с помощью одного вызова путем передачи списка, в котором указываются `[xmin, xmax, ymin, ymax]` (рис. 4.14):

```
In[11]: plt.plot(x, np.sin(x))
        plt.axis([-1, 11, -1.5, 1.5]);
```

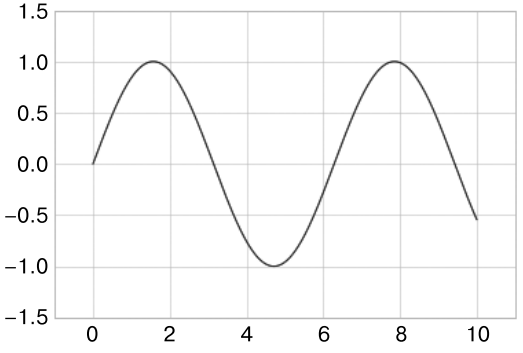

Рис. 4.14. Задание пределов осей координат с помощью метода plt.axis()

Метод `plt.axis()` умеет даже больше, позволяя вам, например, автоматически подгонять границы к текущему графику (рис. 4.15):

```
In[12]: plt.plot(x, np.sin(x))
        plt.axis('tight');
```

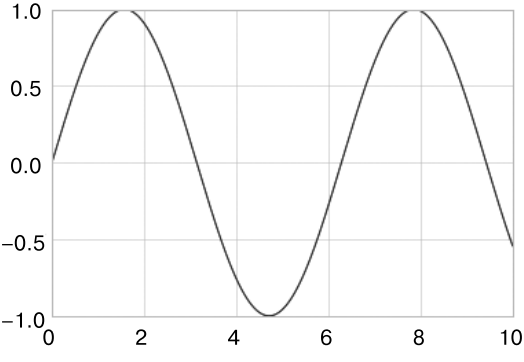

Рис. 4.15. Пример «компактного» графика

С помощью «компактного» графика возможно указание спецификаций и еще более высокого уровня. Например, можно выровнять соотношение сторон графика так, чтобы на вашем экране длина равных приращений по осям X и Y выглядела одинаковой (рис. 4.16):

```
In[13]: plt.plot(x, np.sin(x))
        plt.axis('equal');
```

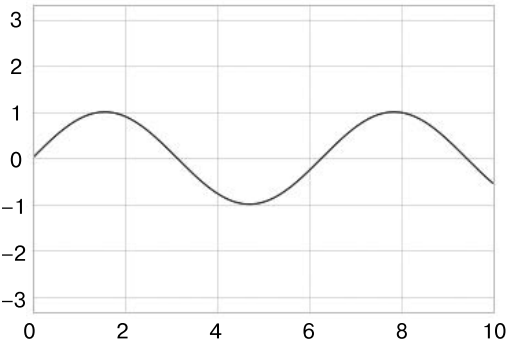

Рис. 4.16. Пример «равностороннего» графика, в котором приращения по осям соответствуют выходному разрешению

Дополнительную информацию по пределам осей координат и другим возможностям метода `plt.axis()` можно найти в документации по этому методу.

Метки на графиках

В завершение этого раздела рассмотрим маркирование графиков: названия, метки осей координат и простые легенды.

Названия и метки осей — простейшие из подобных меток. Существуют методы, позволяющие быстро задать их значения (рис. 4.17):

```
In[14]: plt.plot(x, np.sin(x))
        plt.title("A Sine Curve") # Синусоидальная кривая

        plt.xlabel("x")
        plt.ylabel("sin(x)");
```

Рис. 4.17. Пример меток осей координат и названия графика

С помощью необязательных аргументов функций можно настраивать расположение, размер и стиль этих меток. Подробная информация представлена в документации библиотеки Matplotlib и в разделе docstring для каждой из функций.

В случае отображения нескольких линий в одной координатной сетке удобно создать легенду для графика, на которой бы отмечался каждый тип линии. В библиотеке Matplotlib для быстрого создания такой легенды имеется встроенный метод `plt.legend()`. Хотя существует несколько возможных способов, проще всего, как мне кажется, задать метку каждой линии с помощью ключевого слова `label` функции `plot` (рис. 4.18):

```
In[15]: plt.plot(x, np.sin(x), '-g', label='sin(x)')
        plt.plot(x, np.cos(x), ':b', label='cos(x)')
        plt.axis('equal')

        plt.legend();
```

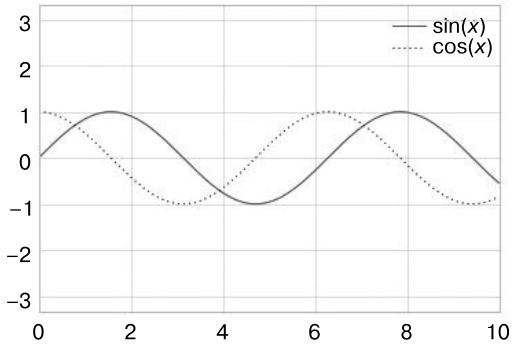

Рис. 4.18. Пример легенды графика

Функция `plt.legend()` отслеживает стиль и цвет линии и устанавливает их соответствие с нужной меткой. Больше информации по заданию и форматированию легенд графиков можно найти в docstring метода `plt.legend()`. Кроме того, мы рассмотрим некоторые продвинутые параметры задания легенд в разделе «Пользовательские настройки легенд на графиках» этой главы.

Нюансы использования Matplotlib

Хотя для большинства функций интерфейса `plt` соответствующие методы интерфейса `ax` носят такое же название (например, `plt.plot()` → `ax.plot()`, `plt.legend()` → `ax.legend()` и т. д.), это касается не всех команд. В частности, функции для задания пределов, меток и названий графиков называются несколько иначе. Вот список соответствий между MATLAB-подобным функциями и объектно-ориентированным методами:

- `plt.xlabel()` → `ax.set_xlabel()`;
- `plt.ylabel()` → `ax.set_ylabel()`;
- `plt.xlim()` → `ax.set_xlim()`;
- `plt.ylim()` → `ax.set_ylim()`;
- `plt.title()` → `ax.set_title()`.

В объектно-ориентированном интерфейсе построения графиков вместо того, чтобы вызывать эти функции по отдельности, удобнее воспользоваться методом `ax.set()`, чтобы задать значения всех этих параметров за один раз (рис. 4.19):

```
In[16]: ax = plt.axes()
        ax.plot(x, np.sin(x))
        ax.set(xlim=(0, 10), ylim=(-2, 2),
               xlabel='x', ylabel='sin(x)',
               title='A Simple Peot'); # Простая диаграмма
```

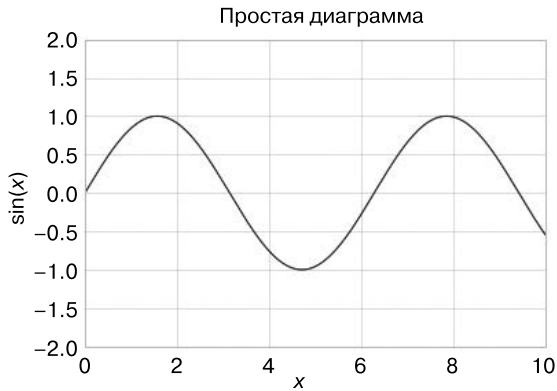

Рис. 4.19. Пример использования метода ax.set() для задания значения нескольких параметров за один вызов

Простые диаграммы рассеяния

Еще один часто используемый тип графиков — диаграммы рассеяния, родственные линейным графикам. В них точки не соединяются отрезками линий, а представлены по отдельности точками, кругами или другими фигурами на графике. Начнем с настройки блокнота для построения графиков и импорта нужных нам функций:

```
In[1]: %matplotlib inline
       import matplotlib.pyplot as plt
       plt.style.use('seaborn-whitegrid')
       import numpy as np
```

Построение диаграмм рассеяния с помощью функции plt.plot

В предыдущем разделе мы рассмотрели построение линейных графиков посредством функции `plt.plot/ax.plot`. С ее помощью можно строить и диаграммы рассеяния (рис. 4.20):

```
In[2]: x = np.linspace(0, 10, 30)
       y = np.sin(x)

       plt.plot(x, y, 'o', color='black');
```

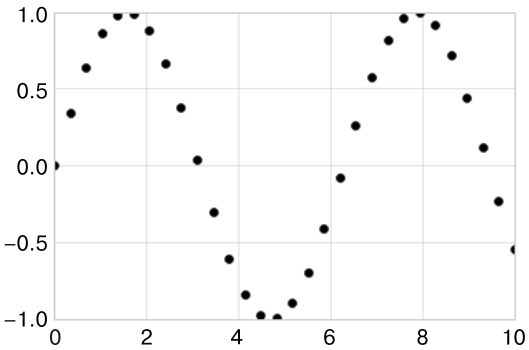

Рис. 4.20. Пример диаграммы рассеяния

Третий аргумент в вызове этой функции описывает тип символа, применяемого для графика. Для управления стилем линии можно использовать такие опции, как `'-'` и `'--'`. Для стиля маркера существует свой набор кратких строковых кодов. Полный список можно найти в документации по функции `plt.plot` или в онлайн-документации по библиотеке Matplotlib. Большинство вариантов интуитивно понятны, мы продемонстрируем часто используемые (рис. 4.21):

```
In[3]: rng = np.random.RandomState(0)
       for marker in ['o', '.', ',', 'x', '+', 'v', '^', '<', '>', 's', 'd']:
           plt.plot(rng.rand(5), rng.rand(5), marker,
                    label="marker='{0}'".format(marker))
       plt.legend(numpoints=1)
       plt.xlim(0, 1.8);
```

Эти символьные коды можно использовать совместно с кодами линий и цветов, рисуя точки вместе с соединяющей их линией (рис. 4.22):

```
In[4]: plt.plot(x, y, '-ok');   # линия (-), маркер круга (o), черный цвет (k)
```

С помощью дополнительных ключевых аргументов функции `plt.plot` можно задавать множество свойств линий и маркеров (рис. 4.23):

```
In[5]: plt.plot(x, y, '-p', color='gray',
                markersize=15, linewidth=4,
```

```
        markerfacecolor='white',
        markeredgecolor='gray',
        markeredgewidth=2)
plt.ylim(-1.2, 1.2);
```

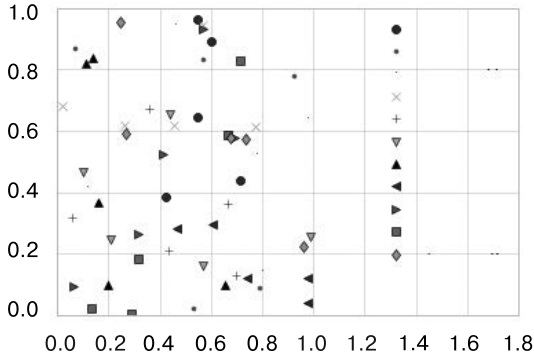

Рис. 4.21. Демонстрация различных маркеров

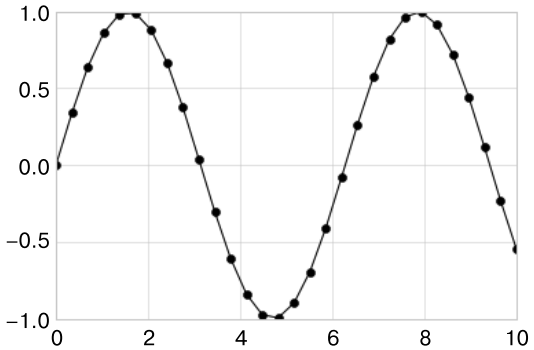

Рис. 4.22. Сочетание линий и маркеров точек

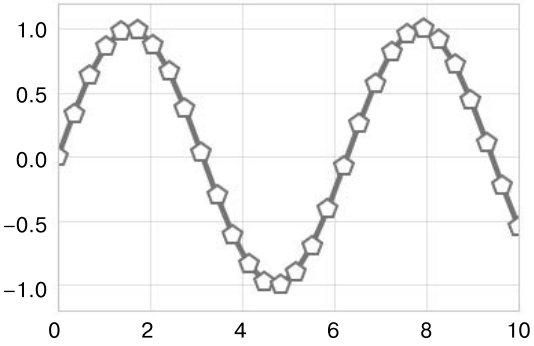

Рис. 4.23. Индивидуальная настройка вида линий и маркеров точек

Подобная гибкость функции `plt.plot` позволяет использовать широкий диапазон настроек визуализации. Полное описание имеющихся настроек можно найти в документации по функции `plt.plot`.

Построение диаграмм рассеяния с помощью функции plt.scatter

Еще большими возможностями обладает метод построения диаграмм рассеяния с помощью функции `plt.scatter`, во многом напоминающей функцию `plt.plot` (рис. 4.24):

```
In[6]: plt.scatter(x, y, marker='o');
```

Основное различие между функциями `plt.scatter` и `plt.plot` состоит в том, что с помощью первой можно создавать диаграммы рассеяния с индивидуально задаваемыми (или выбираемыми в соответствии с данными) свойствами каждой точки (размер, цвет заливки, цвет рамки и т. д.).

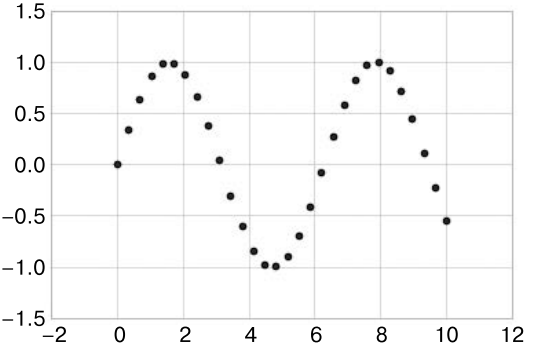

Рис. 4.24. Простая диаграмма рассеяния

Продемонстрируем это, создав случайную диаграмму рассеяния с точками различных цветов и размеров. Чтобы лучше видеть перекрывающиеся результаты, воспользуемся ключевым словом `alpha` для настройки уровня прозрачности (рис. 4.25):

```
In[7]: rng = np.random.RandomState(0)
       x = rng.randn(100)
       y = rng.randn(100)
       colors = rng.rand(100)
       sizes = 1000 * rng.rand(100)

       plt.scatter(x, y, c=colors, s=sizes, alpha=0.3,
                   cmap='viridis')
       plt.colorbar();  # Отображаем цветовую шкалу
```

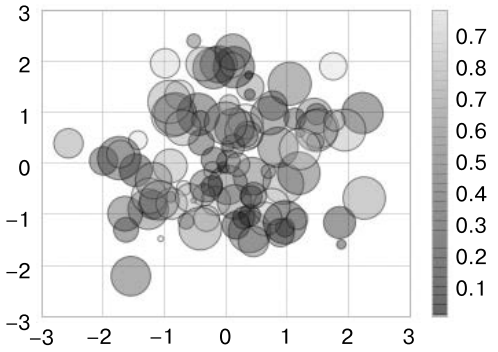

Рис. 4.25. Изменение размера, цвета и прозрачности точек на диаграмме рассеяния

Обратите внимание, что цвета автоматически привязываются к цветовой шкале (которую мы отобразили с помощью команды `colorbar()`), а размеры указываются в пикселах. Благодаря всему этому можно использовать цвет и размер точек для передачи информации на графике с целью иллюстрации многомерных данных.

Для примера возьмем набор данных Iris из библиотеки Scikit-Learn, каждая выборка представляет собой один из трех типов цветов с тщательно измеренными лепестками и чашелистиками (рис. 4.26):

```
In[8]: from sklearn.datasets import load_iris
       iris = load_iris()

       features = iris.data.T
       plt.scatter(features[0], features[1], alpha=0.2,
                   s=100*features[3], c=iris.target, cmap='viridis')
       plt.xlabel(iris.feature_names[0])
       plt.ylabel(iris.feature_names[1]);
```

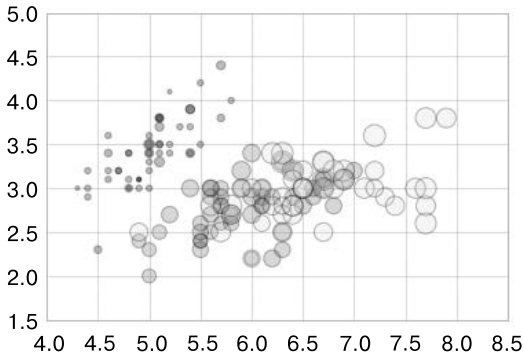

Рис. 4.26. Использование свойств точек для кодирования признаков данных набора Iris

Как видим, эта диаграмма рассеяния позволила нам одновременно исследовать четыре различных измерения данных: (x, y)-координаты каждой точки соответствуют длине и ширине чашелистика, размер точки — ширине лепестков, цвет — конкретной разновидности цветка. Подобные многоцветные диаграммы рассеяния с несколькими признаками для каждой точки будут очень полезны как для исследования, так и для представления данных.

Сравнение функций plot и scatter: примечание относительно производительности

При небольших объемах данных это не играет роли, но при наборах данных, превышающих несколько тысяч точек, функция `plt.plot` может оказаться намного эффективнее `plt.scatter`. Поскольку `plt.scatter` умеет визуализировать различные размеры и цвета каждой точки, визуализатору приходится выполнять дополнительную работу по формированию каждой точки в отдельности. В случае же функции `plt.plot`, напротив, точки всегда двойники друг друга, поэтому работа по определению внешнего вида точек выполняется только один раз для всего набора данных. Для больших наборов данных это различие может приводить к коренным различиям в производительности, поэтому в таком случае следует использовать функцию `plt.plot`, а не `plt.scatter`.

Визуализация погрешностей

Точный учет погрешностей, как и точная информация о самих значениях важен для любых научных измерений. Например, представьте, что я использую некоторые астрофизические наблюдения для оценки постоянной Хаббла, то есть измеряю скорость расширения Вселенной в данной точке. Мне известно, что в современных источниках по этому вопросу указывается значение около 71 (км/с)/Мпк, а я с помощью моего метода получил значение 74 (км/с)/Мпк. Не противоречат ли значения друг другу? По вышеприведенной информации ответить на этот вопрос невозможно.

Теперь допустим, что я дополнил эту информацию погрешностями: современные источники указывают значение около 71 ± 2,5 (км/с)/Мпк, а мой метод дал значение 74 ± 5 (км/с)/Мпк. Не противоречат ли теперь значения друг другу? Это вопрос, на который вполне можно дать количественный ответ.

При визуализации данных и результатов эффективное отображение погрешностей позволяет передавать с помощью графика намного более полную информацию.

Простые планки погрешностей

Простые планки погрешностей можно создать с помощью вызова всего одной функции библиотеки Matplotlib (рис. 4.27):

```
In[1]: %matplotlib inline
       import matplotlib.pyplot as plt
       plt.style.use('seaborn-whitegrid')
```

```
        import numpy as np
In[2]: x = np.linspace(0, 10, 50)
        dy = 0.8
        y = np.sin(x) + dy * np.random.randn(50)

        plt.errorbar(x, y, yerr=dy, fmt='.k');
```

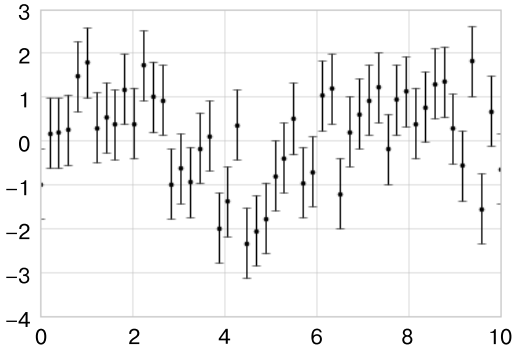

Рис. 4.27. Пример планок погрешностей

Здесь `fmt` — код форматирования, управляющий внешним видом линий и точек, его синтаксис совпадает с сокращенным синтаксисом, используемым в функции `plt.plot`, описываемой в разделах «Простые линейные графики» и «Простые диаграммы рассеяния» данной главы.

Помимо этих простейших, у функции `errorbar` есть множество параметров для более тонкой настройки выводимых данных. С помощью этих дополнительных параметров вы можете легко настроить в соответствии со своими требованиями внешний вид графика планок погрешностей. Планки погрешностей удобно делать более светлыми, чем точки, особенно на насыщенных графиках (рис. 4.28):

```
In[3]: plt.errorbar(x, y, yerr=dy, fmt='o', color='black',
                    ecolor='lightgray', elinewidth=3, capsize=0);
```

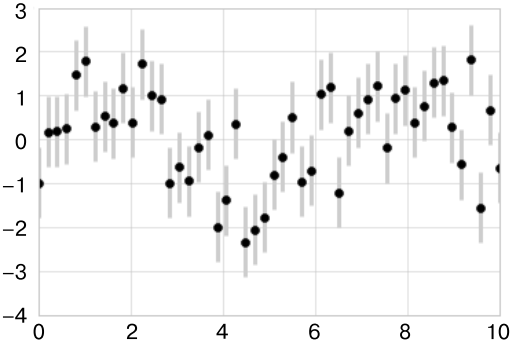

Рис. 4.28. Пользовательские настройки планок погрешностей

В дополнение к этим опциям можно также создавать горизонтальные планки погрешностей (`xerr`), односторонние планки погрешностей и много других вариантов. Чтобы узнать больше об имеющихся опциях, обратитесь к docstring функции `plt.errorbar`.

Непрерывные погрешности

В некоторых случаях желательно отображать планки погрешностей для непрерывных величин. Хотя в библиотеке Matplotlib отсутствует встроенная удобная утилита для решения данной задачи, не составит особого труда скомбинировать такие примитивы, как `plt.plot` и `plt.fill_between`, для получения искомого результата.

Выполним с помощью API пакета Scikit-Learn (см. подробности в разделе «Знакомство с библиотекой Scikit-Learn» главы 5) простую *регрессию на основе Гауссова процесса* (Gaussian process regression, GPR). Она представляет собой метод подбора по имеющимся данным очень гибкой непараметрической функции с непрерывной мерой неопределенности измерения. Мы не будем углубляться в детали регрессии на основе Гауссова процесса, а сконцентрируемся на визуализации подобной непрерывной погрешности измерения:

```
In[4]: from sklearn.gaussian_process import GaussianProcess

       # Описываем модель и отрисовываем некоторые данные
       model = lambda x: x * np.sin(x)
       xdata = np.array([1, 3, 5, 6, 8])
       ydata = model(xdata)

       # Выполняем подгонку Гауссова процесса
       gp = GaussianProcess[1](corr='cubic', theta0=1e-2, thetaL=1e-4,
                               thetaU=1E-1, random_start=100)
       gp.fit(xdata[:, np.newaxis], ydata)

       xfit = np.linspace(0, 10, 1000)
       yfit, MSE = gp.predict(xfit[:, np.newaxis], eval_MSE=True)
       dyfit = 2 * np.sqrt(MSE)  # 2*сигма ~ область с уровнем доверия 95%
```

Теперь у нас имеются значения `xfit`, `yfit` и `dyfit`, представляющие выборку непрерывных приближений к нашим данным. Мы можем передать их, как и выше, функции `plt.errorbar`, но рисовать 1000 точек с помощью 1000 планок погрешностей не хотелось бы. Вместо этого можно воспользоваться функцией `plt.fill_between` и визуализировать эту непрерывную погрешность с помощью светлого цвета (рис. 4.29):

```
In[5]: # Визуализируем результат
       plt.plot(xdata, ydata, 'or')
       plt.plot(xfit, yfit, '-', color='gray')
```

[1] Этот класс является устаревшим и планируется к удалению в версии 0.20 библиотеки SciKit-Learn. Вместо него рекомендуется использовать класс Gaussian Process Regressor.

```
plt.fill_between(xfit, yfit - dyfit, yfit + dyfit,
                 color='gray', alpha=0.2)
plt.xlim(0, 10);
```

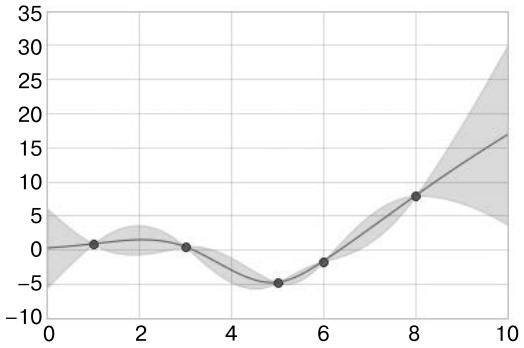

Рис. 4.29. Представляем непрерывную погрешность с помощью закрашенных областей

Обратите внимание на передаваемые функции `plt.fill_between` параметры: мы передали в нее значение `x`, затем нижнюю границу по `y`, затем верхнюю границу по `y`, в результате получили заполненную область между ними.

Получившийся рисунок отлично поясняет, что делает алгоритм регрессии на основе Гауссова процесса. В областях возле измеренной точки данных модель жестко ограничена, что и отражается в малых ошибках модели. В удаленных же от измеренной точки данных областях модель жестко не ограничивается и ошибки модели растут.

Чтобы узнать больше о возможностях функции `plt.fill_between()` (и родственной ей функции `plt.fill()`), см. ее docstring или документацию библиотеки Matplotlib.

Наконец, если описанное вам не по вкусу, загляните в раздел «Визуализация с помощью библиотеки Seaborn» данной главы, в котором мы обсудим пакет Seaborn с продвинутым API для визуализации подобных непрерывных погрешностей.

Графики плотности и контурные графики

Иногда удобно отображать трехмерные данные в двумерной плоскости с помощью контуров или областей, кодированных различными цветами. Существует три предназначенные для этой цели функции библиотеки Matplotlib:

- `plt.contour` — для контурных графиков;
- `plt.contourf` — для контурных графиков с заполнением;
- `plt.imshow` — для отображения изображений. В этом разделе мы рассмотрим несколько примеров использования данных функций. Начнем с настройки блокнота для построения графиков и нужных нам импортов:

```
In[1]: %matplotlib inline
       import matplotlib.pyplot as plt
       plt.style.use('seaborn-white')
       import numpy as np
```

Визуализация трехмерной функции. Начнем с демонстрации контурного графика для функции вида `z = f (x, y)`, воспользовавшись для этой цели функцией `f` (мы уже встречались с ней в разделе «Операции над массивами. Транслирование» главы 2, где использовали ее в примере транслирования массивов):

```
In[2]: def f(x, y):
           return np.sin(x) ** 10 + np.cos(10 +  * x) * np.cos(x)
```

Создать контурный график можно с помощью функции `plt.contour`. У нее имеется три аргумента: координатная сетка значений *x*, координатная сетка значений *y* и координатная сетка значений *z*. Значения *x* и *y* представлены точками на графике, а значения *z* будут представлены контурами уровней. Вероятно, наиболее простой способ подготовить такие данные — воспользоваться функцией `np.meshgrid`, формирующей двумерные координатные сетки из одномерных массивов:

```
In[3]: x = np.linspace(0, 5, 50)
       y = np.linspace(0, 5, 40)

       X, Y = np.meshgrid(x, y)
       Z = f(X, Y)
```

Теперь посмотрим, как это отображается на обычном линейном контурном графике (рис. 4.30):

```
In[4]: plt.contour(X, Y, Z, colors='black');
```

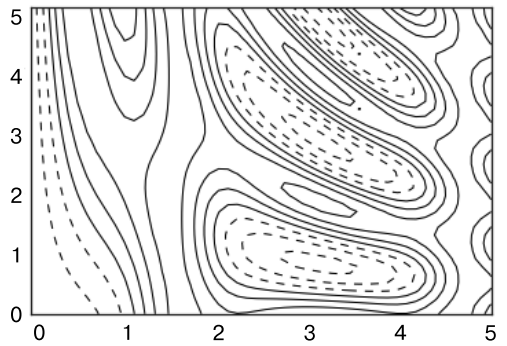

Рис. 4.30. Визуализация трехмерных данных с помощью контуров

Обратите внимание, что по умолчанию при использовании одного цвета отрицательные значения обозначаются штриховыми линиями, а положительные — сплошными. Можно также кодировать линии различными цветами, задав карты цветов с помощью аргумента `cmap`. В данном случае мы также указали, что хотели бы нарисовать больше линий — разделить данные на 20 интервалов с равными промежутками (рис. 4.31):

```
In[5]: plt.contour(X, Y, Z, 20, cmap='RdGy');
```

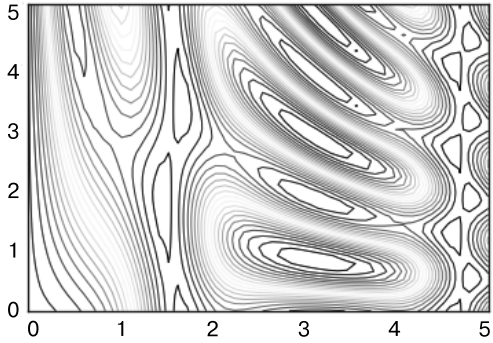

Рис. 4.31. Визуализация трехмерных данных с помощью разноцветных контуров

Мы выбрали здесь карту цветов `RdGy` (сокращение для Red — Gray — «красный — серый») — отличный выбор для случая центрированных данных. В библиотеке Matplotlib доступен широкий диапазон карт цветов, которые можно просмотреть в IPython путем TAB-автодополнения названия модуля `plt.cm`:

```
plt.cm.<TAB>
```

Наш график приобрел более приятный глазу вид, но промежутки между линиями несколько отвлекают внимание. Изменить это можно, воспользовавшись контурным графиком с заполнением, доступным посредством функции `plt.contourf()` (обратите внимание на букву `f` в конце ее названия), синтаксис которой не отличается от синтаксиса функции `plt.contour()`.

Вдобавок мы воспользуемся командой `plt.colorbar()`, автоматически создающей для графика дополнительную ось с маркированной информацией о цвете (рис. 4.32):

```
In[6]: plt.contourf(X, Y, Z, 20, cmap='RdGy')
       plt.colorbar();
```

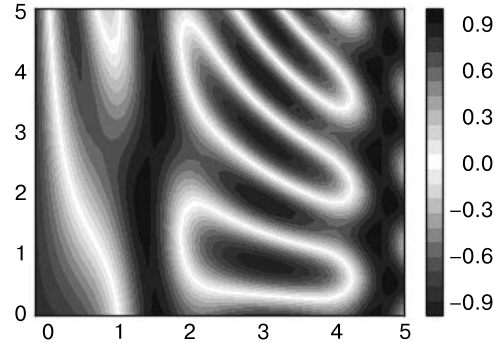

Рис. 4.32. Визуализация трехмерных данных с помощью заполненных контуров

Шкала цветов наглядно демонстрирует, что черные области — точки максимума, а красные — точки минимума.

Потенциальная проблема этого графика — его «пятнистость». Дело в том, что градации цветов здесь дискретны, а не непрерывны, что не всегда удобно. Исправить это можно путем задания очень большого количества контуров, что приведет к низкой производительности: библиотеке Matplotlib придется визуализировать новый полигон для каждого шага уровня. Лучшее решение — воспользоваться функцией `plt.imshow()`, интерпретирующей двумерную сетку данных как изображение.

На рис. 4.33 показан результат выполнения следующего кода:

```
In[7]: plt.imshow(Z, extent=[0, 5, 0, 5], origin='lower', cmap='RdGy')
       plt.colorbar()
       plt.axis(aspect='image');
```

Однако есть несколько потенциальных проблем и с функцией `imshow()`.

- Функция `plt.imshow()` не принимает в качестве параметров координатные сетки *x* и *y*, так что вам придется вручную задать размеры изображения на графике: `extent [xmin, xmax, ymin, ymax]`.
- По умолчанию функция `plt.imshow()` следует стандартному определению массива для изображения, в котором начало координат находится в верхнем левом, а не в нижнем левом углу, как на большинстве контурных графиков. Это поведение можно изменить в случае отображения данных с привязкой к сетке.
- Функция `plt.imshow()` автоматически настраивает соотношение сторон графика в соответствии с входными данными. Это поведение можно изменить, задав, например, `plt.axis(aspect='image')`, чтобы отрезки по осям *X* и *Y* были одинаковыми.

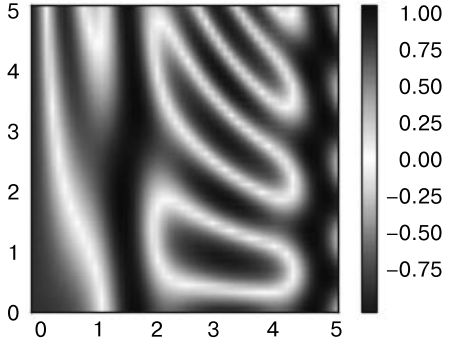

Рис. 4.33. Представляем трехмерные данные в виде изображения

Иногда удобно комбинировать контурный график с графиком-изображением. Например, для создания показанного на рис. 4.34 эффекта мы воспользуемся

частично прозрачным фоновым изображением (задав прозрачность посредством параметра `alpha`) и нарисуем на самих контурах метки (с помощью функции `plt.clabel()`):

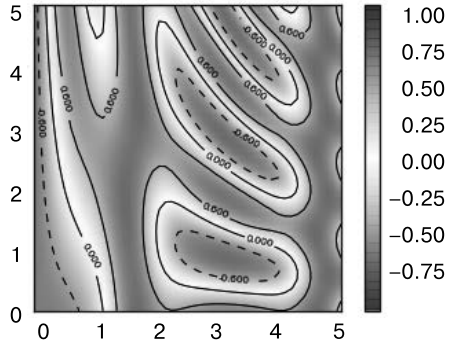

Рис. 4.34. Маркированные контуры поверх изображения

```
In[8]: contours = plt.contour(X, Y, Z, 3, colors='black')
       plt.clabel(contours, inline=True, fontsize=8)

       plt.imshow(Z, extent=[0, 5, 0, 5], origin='lower',
                  cmap='RdGy', alpha=0.5)
       plt.colorbar();
```

Сочетание этих трех функций — `plt.contour`, `plt.contourf` и `plt.imshow` — предоставляет практически неограниченные возможности по отображению подобных трехмерных данных на двумерных графиках. Дальнейшую информацию относительно имеющихся у этих функций параметров вы можете найти в их docstring. Если вас интересует трехмерная визуализация таких данных, загляните в раздел «Построение трехмерных графиков в библиотеке Matplotlib» данной главы.

Гистограммы, разбиения по интервалам и плотность

Простая гистограмма может принести огромную пользу при первичном анализе набора данных. Ранее мы видели пример использования функции библиотеки Matplotlib (см. раздел «Сравнения, маски и булева логика» главы 2) для создания простой гистограммы в одну строку после выполнения всех обычных импортов (рис. 4.35):

```
In[1]: %matplotlib inline
       import numpy as np
       import matplotlib.pyplot as plt
       plt.style.use('seaborn-white')
```

```
        data = np.random.randn(1000)

In[2]: plt.hist(data);
```

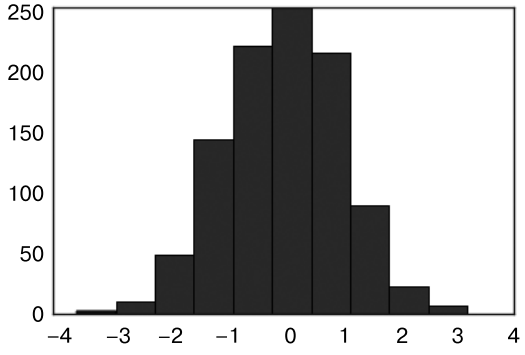

Рис. 4.35. Простая гистограмма

У функции `hist()` имеется множество параметров для настройки как вычисления, так и отображения. Вот пример гистограммы с детальными пользовательскими настройками (рис. 4.36):

```
In[3]: plt.hist(data, bins=30, normed=True, alpha=0.5,
                histtype='stepfilled', color='steelblue',
                edgecolor='none');
```

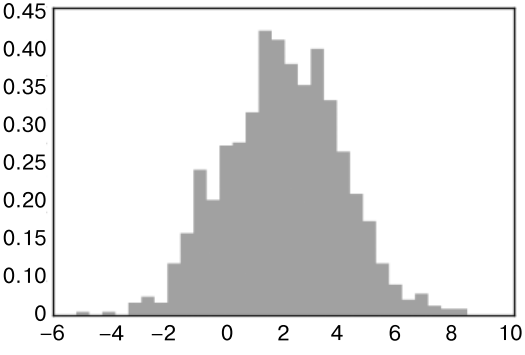

Рис. 4.36. Гистограмма с пользовательскими настройками

Docstring функции `plt.hist` содержит более подробную информацию о других доступных возможностях пользовательской настройки. Сочетание опции `histtype='stepfilled'` с заданной прозрачностью `alpha` представляется мне очень удобным для сравнения гистограмм нескольких распределений (рис. 4.37):

```
In[4]: x1 = np.random.normal(0, 0.8, 1000)
       x2 = np.random.normal(-2, 1, 1000)
       x3 = np.random.normal(3, 2, 1000)
```

```
kwargs = dict(histtype='stepfilled', alpha=0.3, normed=True, bins=40)

plt.hist(x1, **kwargs)
plt.hist(x2, **kwargs)
plt.hist(x3, **kwargs);
```

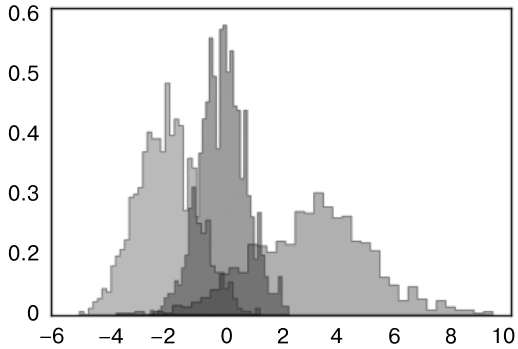

Рис. 4.37. Рисуем несколько гистограмм поверх друг друга

Если же вам нужно вычислить гистограмму (то есть подсчитать количество точек в заданном интервале) и не отображать ее, к вашим услугам функция `np.histogram()`:

```
In[5]: counts, bin_edges = np.histogram(data, bins=5)
       print(counts)

[ 12 190 468 301  29]
```

Двумерные гистограммы и разбиения по интервалам

Аналогично тому, как мы создавали одномерные гистограммы, разбивая последовательность чисел по интервалам, можно создавать и двумерные гистограммы, распределяя точки по двумерным интервалам. Рассмотрим несколько способов выполнения. Начнем с описания данных массивов `x` и `y`, полученных из многомерного Гауссова распределения:

```
In[6]: mean = [0, 0]
       cov = [[1, 1], [1, 2]]
       x, y = np.random.multivariate_normal(mean, cov, 10000).T
```

Функция plt.hist2d: двумерная гистограмма

Один из простых способов нарисовать двумерную гистограмму — воспользоваться функцией `plt.hist2d` библиотеки Matplotlib (рис. 4.38):

```
In[12]: plt.hist2d(x, y, bins=30, cmap='Blues')
        cb = plt.colorbar()
        cb.set_label('counts in bin') # Количествво в интервале
```

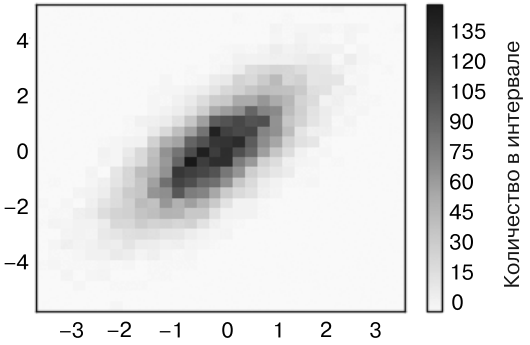

Рис. 4.38. Двумерная гистограмма, построенная с помощью функции plt.hist2d

У функции `plt.hist2d`, как и у функции `plt.hist`, имеется немало дополнительных параметров для тонкой настройки графика и разбиения по интервалам, подробно описанных в ее docstring. Аналогично тому, как у функции `plt.hist` есть эквивалент `np.histogram`, так и у функции `plt.hist2d` имеется эквивалент `np.histogram2d`, который используется следующим образом:

```
In[8]: counts, xedges, yedges = np.histogram2d(x, y, bins=30)
```

Для обобщения разбиения по интервалам для гистограммы на число измерений, превышающее 2, см. функцию `np.histogramdd`.

Функция plt.hexbin: гексагональное разбиение по интервалам

Двумерная гистограмма создает мозаичное представление квадратами вдоль координатных осей. Другая геометрическая фигура для подобного мозаичного представления — правильный шестиугольник. Для этих целей библиотека Matplotlib предоставляет функцию `plt.hexbin` — двумерный набор данных, разбитых по интервалам на сетке из шестиугольников (рис. 4.39):

```
In[9]: plt.hexbin(x, y, gridsize=30, cmap='Blues')
       cb = plt.colorbar(label='count in bin') # Количество в интервале
```

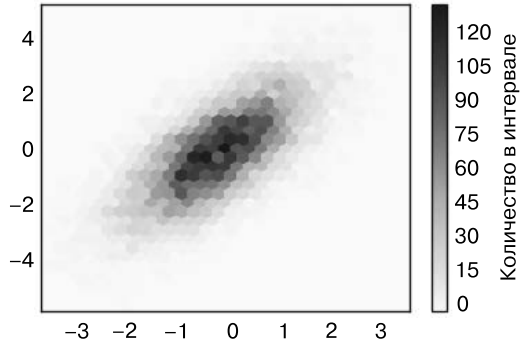

Рис. 4.39. Создание двумерной гистограммы с помощью функции plt.hexbin

У функции `plt.hexbin` имеется множество интересных параметров, включая возможность задавать вес для каждой точки и менять выводимое значение для каждого интервала на любой сводный показатель библиотеки NumPy (среднее значение весов, стандартное отклонение весов и т. д.).

Ядерная оценка плотности распределения

Еще один часто используемый метод оценки плотностей в многомерном пространстве — *ядерная оценка плотности распределения* (kernel density estimation, KDE). Более подробно мы рассмотрим ее в разделе «Заглянем глубже: ядерная оценка плотности распределения» главы 5, а пока отметим, что KDE можно представлять как способ «размазать» точки в пространстве и сложить результаты для получения гладкой функции. В пакете `scipy.stats` имеется исключительно быстрая и простая реализация KDE. Вот короткий пример использования KDE на вышеуказанных данных (рис. 4.40):

```
In[10]: from scipy.stats import gaussian_kde

        # Выполняем подбор на массиве размера [Ndim, Nsamples]
        data = np.vstack([x, y])
        kde = gaussian_kde(data)

        # Вычисляем на регулярной координатной сетке
        xgrid = np.linspace(-3.5, 3.5, 40)
        ygrid = np.linspace(-6, 6, 40)
        Xgrid, Ygrid = np.meshgrid(xgrid, ygrid)
        Z = kde.evaluate(np.vstack([Xgrid.ravel(), Ygrid.ravel()]))

        # Выводим график результата в виде изображения
        plt.imshow(Z.reshape(Xgrid.shape),
                   origin='lower', aspect='auto',
                   extent=[-3.5, 3.5, -6, 6],
                   cmap='Blues')
        cb = plt.colorbar()
        cb.set_label("density") # Плотность
```

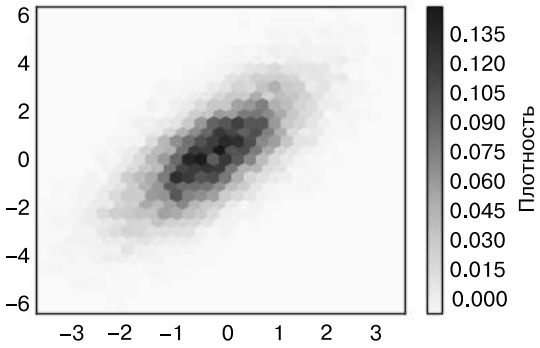

Рис. 4.40. Ядерная оценка плотности распределения

Длина сглаживания метода KDE позволяет эффективно выбирать компромисс между гладкостью и детализацией (один из примеров вездесущих компромиссов между смещением и дисперсией). Существует обширная литература, посвященная выбору подходящей длины сглаживания: в функции `gaussian_kde` используется эмпирическое правило для поиска квазиоптимальной длины сглаживания для входных данных.

В экосистеме SciPy имеются и другие реализации метода KDE, каждая со своими сильными и слабыми сторонами, например методы `sklearn.neighbors.KernelDensity` и `statsmodels.nonparametric.kernel_density.KDEMultivariate`. Использование библиотеки Matplotlib для основанных на методе KDE визуализаций требует написания излишнего кода. Библиотека Seaborn, которую мы будем обсуждать в разделе «Визуализация с помощью библиотеки Seaborn» данной главы, предлагает для создания таких визуализаций API с намного более сжатым синтаксисом.

Пользовательские настройки легенд на графиках

Бо́льшая понятность графика обеспечивается заданием меток для различных элементов графика. Мы ранее уже рассматривали создание простой легенды, здесь продемонстрируем возможности пользовательской настройки расположения и внешнего вида легенд в Matplotlib.

С помощью команды `plt.legend()` можно автоматически создать простейшую легенду для любых маркированных элементов графика (рис. 4.41):

```
In[1]: import matplotlib.pyplot as plt
       plt.style.use('classic')

In[2]: %matplotlib inline
       import numpy as np

In[3]: x = np.linspace(0, 10, 1000)
       fig, ax = plt.subplots()
       ax.plot(x, np.sin(x), '-b', label='Sine')    # Синус
       ax.plot(x, np.cos(x), '--r', label='Cosine') # Косинус
       ax.axis('equal')
       leg = ax.legend();
```

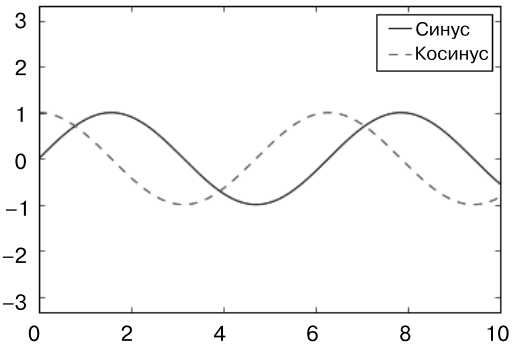

Рис. 4.41. Легенда графика по умолчанию

Существует множество вариантов пользовательских настроек такого графика, которые могут нам понадобиться. Например, можно задать местоположение легенды и отключить рамку (рис. 4.42):

```
In[4]: ax.legend(loc='upper left', frameon=False)
       fig
```

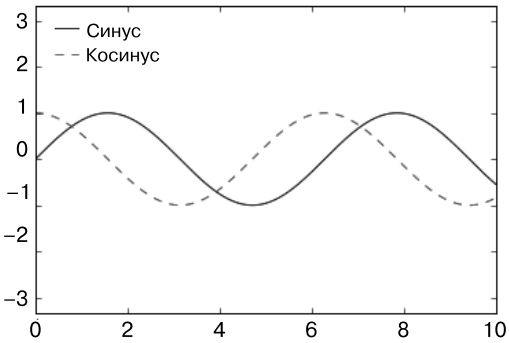

Рис. 4.42. Легенда графика с пользовательскими настройками

Можно также воспользоваться командой `ncol`, чтобы задать количество столбцов в легенде (рис. 4.43):

```
In[5]: ax.legend(frameon=False, loc='lower center', ncol=2)
       fig
```

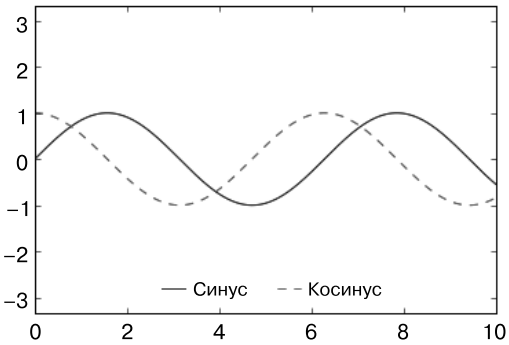

Рис. 4.43. Легенда графика в два столбца

Можно использовать для легенды скругленную прямоугольную рамку (`fancybox`) или добавить тень, поменять прозрачность (альфа-фактор) рамки или поля около текста (рис. 4.44):

```
In[6]: ax.legend(fancybox=True, framealpha=1, shadow=True, borderpad=1)
       fig
```

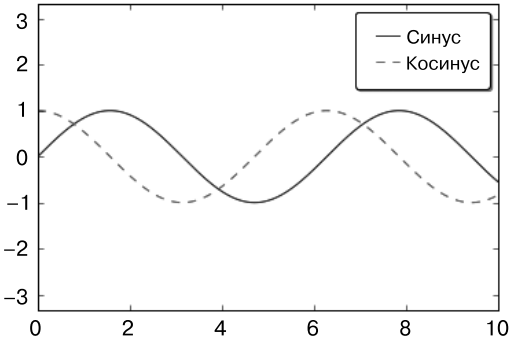

Рис. 4.44. Легенда графика со скругленной прямоугольной рамкой

Дополнительную информацию об имеющихся настройках для легенд можно получить в docstring функции `plt.legend`.

Выбор элементов для легенды

По умолчанию легенда включает все маркированные элементы. Если нам этого не нужно, можно указать, какие элементы и метки должны присутствовать в легенде, воспользовавшись объектами, возвращаемыми командами построения графика. Команда `plt.plot()` умеет рисовать за один вызов несколько линий и возвращать список созданных экземпляров линий. Для указания, какие элементы использовать, достаточно передать какие-либо из них функции `plt.legend()` вместе с задаваемыми метками (рис. 4.45):

```
In[7]: y = np.sin(x[:, np.newaxis] + np.pi * np.arange(0, 2, 0.5))
       lines = plt.plot(x, y)

       # lines представляет собой список экземпляров класса plt.Line2D
       plt.legend(lines[:2], ['first', 'second']); # Первый, второй
```

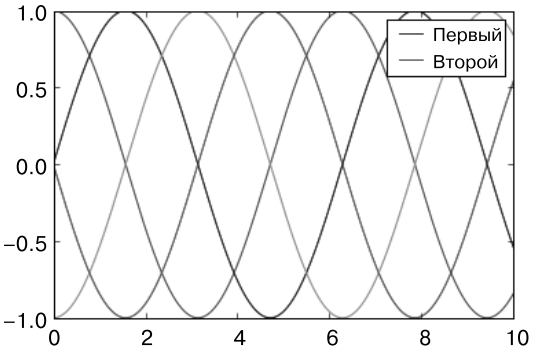

Рис. 4.45. Пользовательские настройки элементов легенды

Обычно на практике мне удобнее использовать первый способ, указывая метки непосредственно для элементов, которые нужно отображать в легенде (рис. 4.46):

```
In[8]: plt.plot(x, y[:, 0], label='first')
       plt.plot(x, y[:, 1], label='second')
       plt.plot(x, y[:, 2:])
       plt.legend(framealpha=1, frameon=True);
```

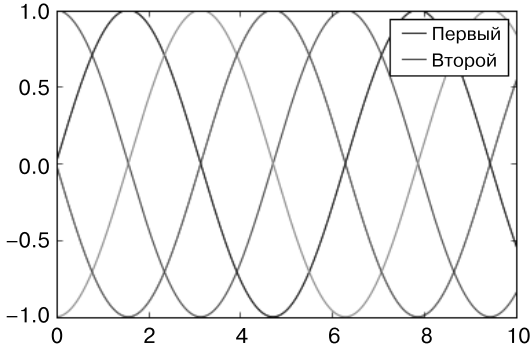

Рис. 4.46. Альтернативный способ пользовательской настройки элементов легенды

Обратите внимание, что по умолчанию в легенде игнорируются все элементы, для которых не установлен атрибут `label`.

Задание легенды для точек различного размера

Иногда возможностей легенды по умолчанию недостаточно для нашего графика. Допустим, вы используете точки различного размера для визуализации определенных признаков данных и хотели бы создать отражающую это легенду. Вот пример, в котором мы будем отражать население городов Калифорнии с помощью размера точек. Нам нужна легенда со шкалой размеров точек, и мы создадим ее путем вывода на графике маркированных данных без самих меток (рис. 4.47):

```
In[9]: import pandas as pd
       cities = pd.read_csv('data/california_cities.csv')

       # Извлекаем интересующие нас данные
       lat, lon = cities['latd'], cities['longd']
       population, area = cities['population_total'], cities['area_total_km2']

       # Распределяем точки по нужным местам,
       # с использованием размера и цвета, но без меток
       plt.scatter(lon, lat, label=None,
                   c=np.log10(population), cmap='viridis',
                   s=area, linewidth=0, alpha=0.5)
       plt.axis(aspect='equal')
```

```
plt.xlabel('longitude')
plt.ylabel('latitude')
plt.colorbar(label='log$_{10}$(population)')
plt.clim(3, 7)

# Создаем легенду:
# выводим на график пустые списки с нужным размером и меткой
for area in [100, 300, 500]:
    plt.scatter([], [], c='k', alpha=0.3, s=area,
                label=str(area) + ' km$^2$')
plt.legend(scatterpoints=1, frameon=False,
           labelspacing=1, title='City Area') # Города

plt.title('California Cities: Area and Population');
# Города Калифорнии: местоположение и население
```

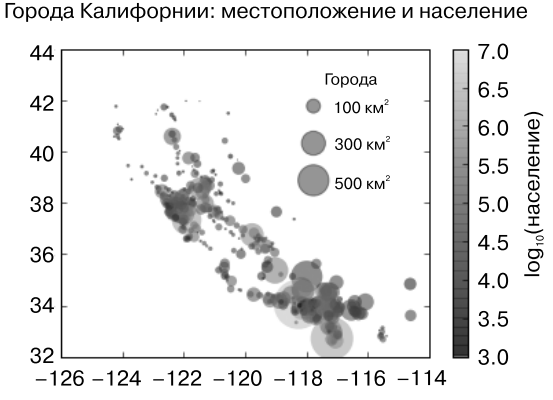

Рис. 4.47. Местоположение, размер и население городов штата Калифорния

Легенда всегда относится к какому-либо находящемуся на графике объекту, поэтому, если нам нужно отобразить объект конкретного вида, необходимо сначала его нарисовать на графике. В данном случае нужных нам объектов (кругов серого цвета) на графике нет, поэтому идем на хитрость и выводим на график пустые списки. Обратите внимание, что в легенде перечислены только те элементы графика, для которых задана метка.

Мы создали посредством вывода на график пустых списков маркированные объекты, которые затем собираются в легенде. Теперь легенда дает нам полезную информацию. Эту стратегию можно использовать для создания и более сложных визуализаций.

Обратите внимание, что в случае подобных географических данных график стал бы понятнее при отображении на нем границ штата и других картографических элементов. Отличный инструмент для этой цели — дополнительный набор утилит Basemap для библиотеки Matplotlib, который мы рассмотрим в разделе «Отображение географических данных с помощью Basemap» данной главы.

Отображение нескольких легенд

Иногда при построении графика необходимо добавить на него несколько легенд для одной и той же системы координат. К сожалению, библиотека Matplotlib не сильно упрощает эту задачу: используя стандартный интерфейс `legend`, можно создавать только одну легенду для всего графика. Если попытаться создать вторую легенду с помощью функций `plt.legend()` и `ax.legend()`, она просто перекроет первую. Решить эту проблему можно, создав изначально для легенды новый рисователь (artist), после чего добавить вручную второй рисователь на график с помощью низкоуровневого метода `ax.add_artist()` (рис. 4.48):

```
In[10]: fig, ax = plt.subplots()

        lines = []
        styles = ['-', '--', '-.', ':']
        x = np.linspace(0, 10, 1000)

        for i in range(4):
            lines += ax.plot(x, np.sin(x - i * np.pi / 2),
                             styles[i], color='black')
        ax.axis('equal')

        # Задаем линии и метки первой легенды
        ax.legend(lines[:2], ['line A', 'line B'],        # Линия A, линия B
                  loc='upper right', frameon=False)

        # Создаем вторую легенду и добавляем рисователь вручную
        from matplotlib.legend import Legend
        leg = Legend(ax, lines[2:], ['line C', 'line D'], # Линия C, линия D
                     loc='lower right', frameon=False)
        ax.add_artist(leg);
```

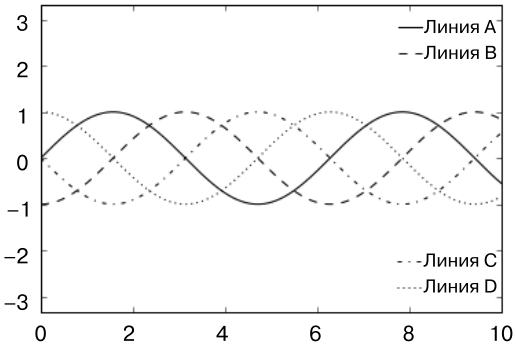

Рис. 4.48. Разделенная на части легенда

Мы мельком рассмотрели низкоуровневые объекты рисования, из которых состоит любой график библиотеки Matplotlib. Если вы заглянете в исходный код метода

`ax.legend()` (напомню, что сделать это можно в блокноте оболочки IPython с помощью команды `legend??`), то увидите, что эта функция состоит просто из логики создания подходящего рисователя `Legend`, сохраняемого затем в атрибуте `legend_` и добавляемого к рисунку при отрисовке графика.

Пользовательские настройки шкал цветов

Легенды графика отображают соответствие дискретных меток дискретным точкам. В случае непрерывных меток, базирующихся на цвете точек, линий или областей, отлично подойдет такой инструмент, как шкала цветов. В библиотеке Matplotlib шкала цветов — отдельная система координат, предоставляющая ключ к значению цветов на графике. Поскольку эта книга напечатана в черно-белом исполнении, для данного раздела имеется дополнительное онлайн-приложение, в котором вы можете посмотреть на оригинальные графики в цвете (https://github.com/jakevdp/PythonDataScienceHandbook). Начнем с настройки блокнота для построения графиков и импорта нужных функций:

```
In[1]: import matplotlib.pyplot as plt
       plt.style.use('classic')

In[2]: %matplotlib inline
       import numpy as np
```

Простейшую шкалу цветов можно создать с помощью функции `plt.colorbar` (рис. 4.49):

```
In[3]: x = np.linspace(0, 10, 1000)
       I = np.sin(x) * np.cos(x[:, np.newaxis])

       plt.imshow(I)
       plt.colorbar();
```

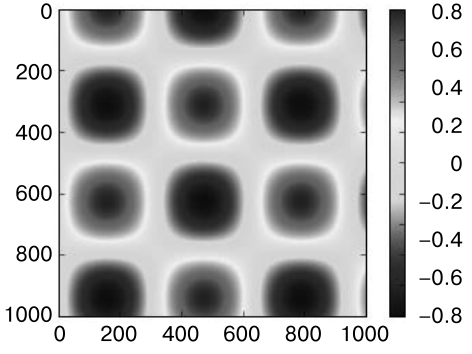

Рис. 4.49. Простая легенда-шкала цветов

Далее мы рассмотрим несколько идей по пользовательской настройке шкалы цветов и эффективному их использованию в разных ситуациях.

Задать карту цветов можно с помощью аргумента `cmap` функции создания визуализации (рис. 4.50):

```
In[4]: plt.imshow(I, cmap='gray');
```

Все доступные для использования карты цветов содержатся в пространстве имен `plt.cm`. Вы можете получить полный список встроенных опций с помощью TAB-автодополнения в оболочке IPython:

```
plt.cm.<TAB>
```

Но *возможность* выбора карты цветов — лишь первый шаг, гораздо важнее *выбрать* среди имеющихся вариантов! Выбор оказывается гораздо более тонким, чем вы могли бы ожидать.

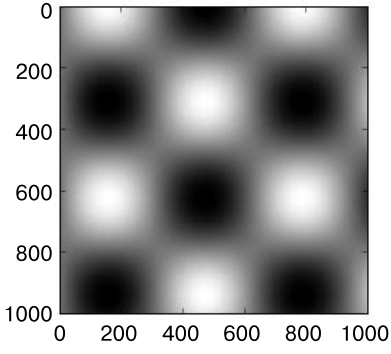

Рис. 4.50. Карта цветов на основе оттенков серого

Выбор карты цветов

Всестороннее рассмотрение вопроса выбора цветов в визуализации выходит за пределы данной книги, но по этому вопросу вы можете почитать статью Ten Simple Rules for Better Figures («Десять простых правил для улучшения рисунков», http://bit.ly/2fDJn9J). Онлайн-документация библиотеки Matplotlib также содержит интересную информацию по вопросу выбора карты цветов (http://matplotlib.org/1.4.1/users/colormaps.html).

Вам следует знать, что существует три различные категории карт цветов:

- последовательные карты цветов. Состоят из одной непрерывной последовательности цветов (например, `binary` или `viridis`);
- дивергентные карты цветов. Обычно содержат два хорошо различимых цвета, отражающих положительные и отрицательные отклонения от среднего значения (например, `RdBu` или `PuOr`);
- качественные карты цветов. В них цвета смешиваются без какого-либо четкого порядка (например, `rainbow` или `jet`).

Карта цветов `jet`, использовавшаяся по умолчанию в библиотеке Matplotlib до версии 2.0, представляет собой пример качественной карты цветов. Ее выбор в качестве карты цветов по умолчанию был весьма неудачен, поскольку качественные карты цветов плохо подходят для отражения количественных данных: обычно они не отражают равномерного роста яркости при продвижении по шкале.

Продемонстрировать это можно путем преобразования шкалы цветов `jet` в черно-белое представление (рис. 4.51):

```
In[5]:
from matplotlib.colors import LinearSegmentedColormap

def grayscale_cmap(cmap):
    """Возвращает версию в оттенках серого заданной карты цветов"""
    cmap = plt.cm.get_cmap(cmap)
    colors = cmap(np.arange(cmap.N))

    # Преобразуем RGBA в воспринимаемую глазом светимость серого цвета
    # ср. http://alienryderflex.com/hsp.html
    RGB_weight = [0.299, 0.587, 0.114]
    luminance = np.sqrt(np.dot(colors[:, :3] ** 2, RGB_weight))
    colors[:, :3] = luminance[:, np.newaxis]

    return LinearSegmentedColormap.from_list(cmap.name + "_gray",
    colors, cmap.N)

def view_colormap(cmap):
    """Рисует карту цветов в эквивалентных оттенках серого"""
    cmap = plt.cm.get_cmap(cmap)
    colors = cmap(np.arange(cmap.N))

    cmap = grayscale_cmap(cmap)
    grayscale = cmap(np.arange(cmap.N))
    fig, ax = plt.subplots(2, figsize=(6, 2),
                           subplot_kw=dict(xticks=[], yticks=[]))
    ax[0].imshow([colors], extent=[0, 10, 0, 1])
    ax[1].imshow([grayscale], extent=[0, 10, 0, 1])

In[6]: view_colormap('jet')
```

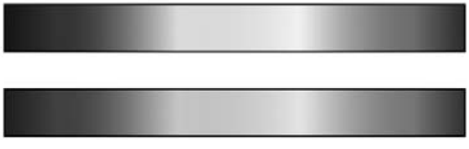

Рис. 4.51. Карта цветов jet и ее неравномерная шкала светимости

Отметим яркие полосы в ахроматическом изображении. Даже в полном цвете эта неравномерная яркость означает, что определенные части диапазона цветов будут притягивать внимание, что потенциально приведет к акцентированию

несущественных частей набора данных. Лучше применять такие карты цветов, как `viridis` (используется по умолчанию, начиная с версии 2.0 библиотеки Matplotlib), специально сконструированные для равномерного изменения яркости по диапазону. Таким образом, они не только согласуются с нашим цветовым восприятием, но и преобразуются для целей печати в оттенках серого (рис. 4.52):

```
In[7]: view_colormap('viridis')
```

Рис. 4.52. Карта цветов viridis и ее равномерная шкала светимости

Если вы предпочитаете радужные цветовые схемы, хорошим вариантом для непрерывных данных будет карта цветов `cubehelix` (рис. 4.53):

```
In[8]: view_colormap('cubehelix')
```

Рис. 4.53. Карта цветов cubehelix и ее светимость

В других случаях, например для отображения положительных и отрицательных отклонений от среднего значения, могут оказаться удобны такие двуцветные карты шкалы цветов, как `RdBu` (сокращение от Red — Blue — «красный — синий»). Однако, как вы можете видеть на рис. 4.54, такая информация будет потеряна при переходе к оттенкам серого!

```
In[9]: view_colormap('RdBu')
```

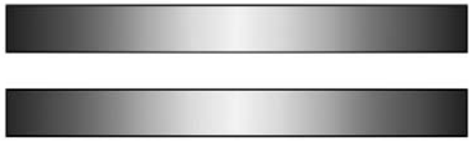

Рис. 4.54. RdBu (красно-синяя) карта цветов и ее светимость

Далее мы увидим примеры использования некоторых из этих карт цветов.

В библиотеке Matplotlib существует множество карт цветов, для просмотра их списка вы можете воспользоваться оболочкой IPython для просмотра содержимого

подмодуля `plt.cm`. Более принципиальный подход к использованию цветов в языке Python можно найти в инструментах и документации по библиотеке Seaborn (см. раздел «Визуализация с помощью библиотеки Seaborn» этой главы).

Ограничения и расширенные возможности по использованию цветов

Библиотека Matplotlib предоставляет возможность разнообразных пользовательских настроек шкал цветов. Сами по себе шкалы цветов — просто экземпляры класса `plt.Axes`, поэтому для них можно использовать все уже изученные нами трюки, связанные с форматированием осей координат и делений на них. Шкалы цветов обладают достаточной гибкостью: например, можно сузить границы диапазона цветов, обозначив выходящие за пределы этого диапазона значения с помощью треугольных стрелок вверху и внизу путем задания значения свойства `extend`. Это может оказаться удобно, например, при выводе зашумленного изображения (рис. 4.55):

```
In[10]: # создаем шум размером 1% от пикселов изображения
        speckles = (np.random.random(I.shape) < 0.01)
        I[speckles] = np.random.normal(0, 3, np.count_nonzero(speckles))

        plt.figure(figsize=(10, 3.5))

        plt.subplot(1, 2, 1)
        plt.imshow(I, cmap='RdBu')
        plt.colorbar()

        plt.subplot(1, 2, 2)
        plt.imshow(I, cmap='RdBu')
        plt.colorbar(extend='both')
        plt.clim(-1, 1);
```

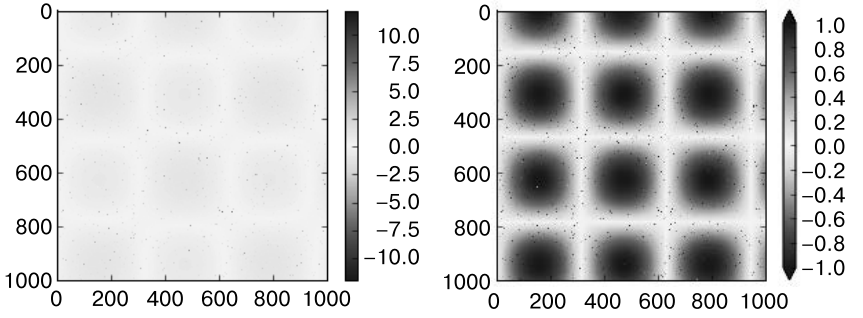

Рис. 4.55. Задаем расширение карты цветов

Обратите внимание, что на левом рисунке зашумленные пикселы влияют на пределы диапазона цветов, по этой причине диапазон шума делает совершенно

неразличимым интересующий нас паттерн. На правом рисунке мы задаем пределы диапазона цветов вручную и добавляем стрелки, указывающие на значения, выходящие за эти пределы. В результате мы получаем намного более удобную визуализацию наших данных.

Дискретные шкалы цветов

Карты цветов по умолчанию непрерывны, но иногда нужно обеспечить отражение дискретных значений. Простейший способ добиться этого — воспользоваться функцией `plt.cm.get_cmap()`, передав в нее название подходящей карты цветов вместе с нужным количеством диапазонов (рис. 4.56):

```
In[11]: plt.imshow(I, cmap=plt.cm.get_cmap('Blues', 6))
        plt.colorbar()
        plt.clim(-1, 1);
```

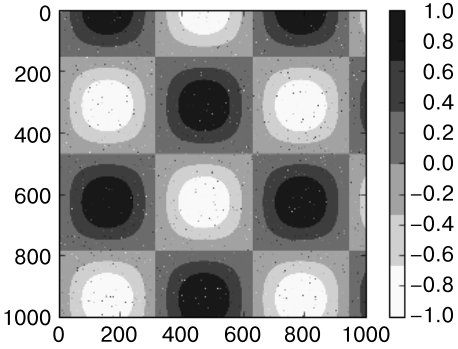

Рис. 4.56. Дискретизированная карта цветов

Использовать дискретный вариант карты цветов можно совершенно так же, как и любую другую карту цветов.

Пример: рукописные цифры

В качестве примера рассмотрим интересную визуализацию данных с рукописными цифрами. Они включены в библиотеку Scikit-Learn и состоят почти из 2000 миниатюр размером 8 × 8 с рукописными цифрами.

Начнем со скачивания содержащих цифры данных и визуализации нескольких примеров изображений с помощью функции `plt.imshow()` (рис. 4.57):

```
In[12]: # Загружаем изображения цифр от 0 до 5 и визуализируем некоторые из них
        from sklearn.datasets import load_digits
        digits = load_digits(n_class=6)
```

```
fig, ax = plt.subplots(8, 8, figsize=(6, 6))
for i, axi in enumerate(ax.flat):
    axi.imshow(digits.images[i], cmap='binary')
    axi.set(xticks=[], yticks=[])
```

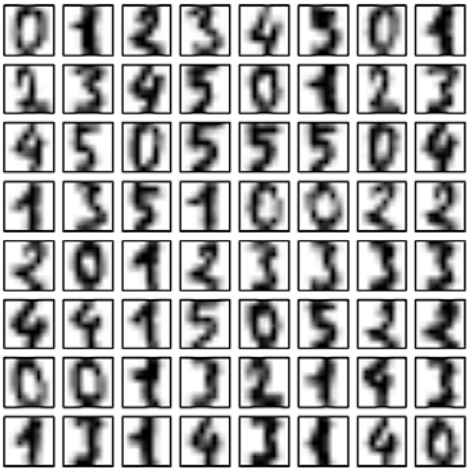

Рис. 4.57. Пример данных с рукописными цифрами

В силу того что каждая цифра определяется оттенком ее 64 пикселов, можно считать ее точкой в 64-мерном пространстве: каждое измерение отражает яркость одного пиксела. Однако визуализация зависимостей в настолько многомерном пространстве представляет собой исключительно непростую задачу. Один из способов ее решения — воспользоваться каким-либо из методов *понижения размерности* (dimensionality reduction), например *обучением на базе многообразий* (manifold learning) с целью снижения размерности данных с сохранением интересующих нас зависимостей. Понижение размерности — пример машинного обучения без учителя (unsupervised machine learning). Мы обсудим его подробнее в разделе «Что такое машинное обучение» главы 5.

Рассмотрим отображение с помощью обучения на базе многообразий наших данных на двумерное пространство (см. подробности в разделе «Заглянем глубже: обучение на базе многообразий» главы 5):

```
In[13]: # Отображаем цифры на двумерное пространство с помощью функции IsoMap
        from sklearn.manifold import Isomap
        iso = Isomap(n_components=2)
        projection = iso.fit_transform(digits.data)
```

Воспользуемся нашей дискретной картой цветов для просмотра результатов, задав параметры `ticks` и `clim` для улучшения внешнего вида итоговой карты цветов (рис. 4.58):

```
In[14]: # Выводим результаты на график
        plt.scatter(projection[:, 0], projection[:, 1], lw=0.1,
                    c=digits.target, cmap=plt.cm.get_cmap('cubehelix', 6))
        plt.colorbar(ticks=range(6), label='digit value')
        # Цифровые значения
        plt.clim(-0.5, 5.5)
```

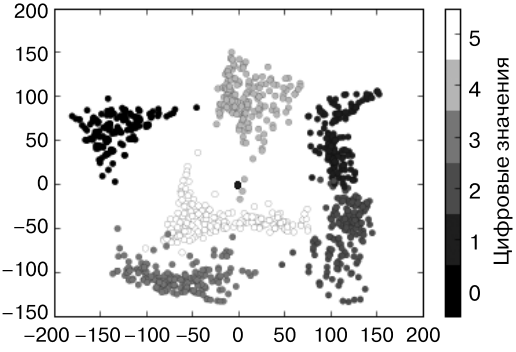

Рис. 4.58. Многообразие рукописных цифр

Это отображение также предоставляет нам полезную информацию о зависимостях внутри набора данных. Например, диапазоны цифр 5 и 3 в проекции практически пересекаются, то есть некоторые рукописные пятерки и тройки отличить друг от друга довольно непросто, и, следовательно, выше вероятность, что автоматический алгоритм классификации будет их путать. Другие значения, например 0 и 1, разделены более отчетливо, значит, вероятность путаницы намного меньше. Это наблюдение хорошо согласуется с нашей интуицией, поскольку цифры 5 и 3 больше похожи друга на друга, чем 0 и 1.

Мы вернемся к обучению на базе многообразий и классификации цифр в главе 5.

Множественные субграфики

Иногда удобно сравнить различные представления данных, разместив их бок о бок. В библиотеке Matplotlib на такой случай предусмотрено понятие *субграфиков* (subplots): несколько маленьких систем координат могут сосуществовать на одном рисунке. Эти субграфики могут представлять собой вставки, сетки графиков или еще более сложные схемы размещения. В данном разделе мы рассмотрим четыре функции для создания субграфиков в библиотеке Matplotlib. Начнем с настройки блокнота для построения графиков и импорта функций, которые нам понадобятся:

```
In[1]: %matplotlib inline
       import matplotlib.pyplot as plt
       plt.style.use('seaborn-white')
       import numpy as np
```

plt.axes: создание субграфиков вручную

Использование функции `plt.axes` — простейший метод создания систем координат, по умолчанию стандартного объекта для системы координат, заполняющего весь график. `plt.axes` также принимает на входе необязательный аргумент, представляющий собой список из четырех чисел в системе координат рисунка. Эти числа означают `[низ, левый угол, ширина, высота]` в системе координат рисунка, отсчет которых начинается с 0 в нижнем левом и заканчивается 1 в верхнем правом углу рисунка.

Например, мы можем создать «вставную» систему координат в верхнем правом углу другой системы координат, задав координаты x и y ее местоположения равными 0.65 (то есть начинающимися на 65% ширины и 65% высоты рисунка), а ее размеры по осям X и Y равными 0.2 (то есть размер этой системы координат составляет 20% ширины и 20% высоты рисунка). Рисунок 4.59 демонстрирует результат следующего кода:

```
In[2]: ax1 = plt.axes()  # обычные оси координат
       ax2 = plt.axes([0.65, 0.65, 0.2, 0.2])
```

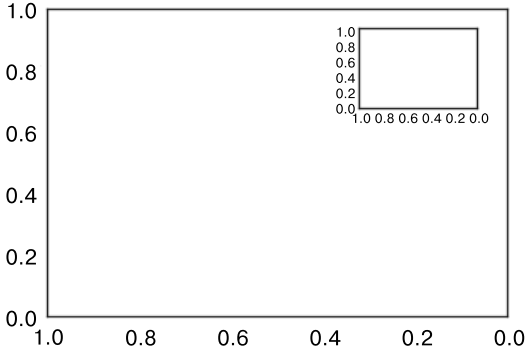

Рис. 4.59. Пример вставной системы координат

Аналог этой команды в объектно-ориентированном интерфейсе — функция `fig.add_axes()`. Воспользуемся ею для создания двух расположенных друг над другом систем координат (рис. 4.60):

```
In[3]: fig = plt.figure()
       ax1 = fig.add_axes([0.1, 0.5, 0.8, 0.4],
                          xticklabels=[], ylim=(-1.2, 1.2))
       ax2 = fig.add_axes([0.1, 0.1, 0.8, 0.4],
                          ylim=(-1.2, 1.2))

       x = np.linspace(0, 10)
       ax1.plot(np.sin(x))
       ax2.plot(np.cos(x));
```

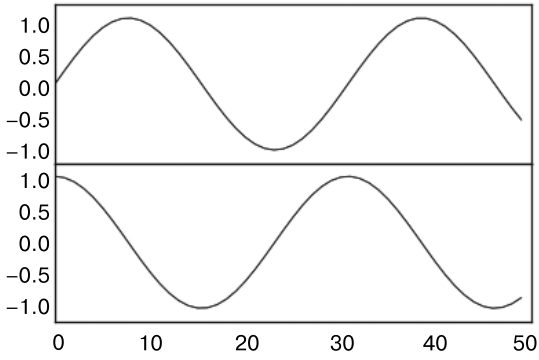

Рис. 4.60. Пример расположенных друг над другом систем координат

Мы получили две соприкасающиеся системы координат (верхняя — без делений): низ верхней области (находящийся на 50 % от размера рисунка) соответствует верху нижней области (находится на 10 + 40 % от размера рисунка).

plt.subplot: простые сетки субграфиков

Выровненные столбцы или строки субграфиков бывают нужны достаточно часто для того, чтобы в библиотеку Matplotlib было включено несколько удобных утилит, облегчающих их создание. Самая низкоуровневая из них — функция `plt.subplot`, создающая отдельный субграфик внутри сетки. Эта команда принимает на входе три целочисленных аргумента — количество строк, количество столбцов и индекс создаваемого по такой схеме графика, отсчет которого начинается в верхнем левом углу и заканчивается в правом нижнем (рис. 4.61):

```
In[4]: for i in range(1, 7):
           plt.subplot(2, 3, i)
           plt.text(0.5, 0.5, str((2, 3, i)),
                    fontsize=18, ha='center')
```

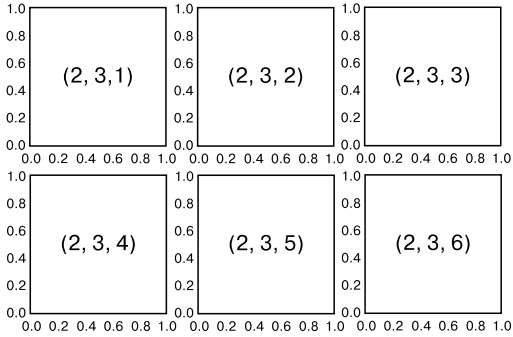

Рис. 4.61. Пример использования функции plt.subplot

Для настройки размеров полей между этими графиками можно выполнить команду `plt.subplots_adjust`. Следующий код (результат которого показан на рис. 4.62) использует эквивалентную объектно-ориентированную команду `fig.add_subplot()`:

```
In[5]: fig = plt.figure()
       fig.subplots_adjust(hspace=0.4, wspace=0.4)
       for i in range(1, 7):
           ax = fig.add_subplot(2, 3, i)
           ax.text(0.5, 0.5, str((2, 3, i)),
                   fontsize=18, ha='center')
```

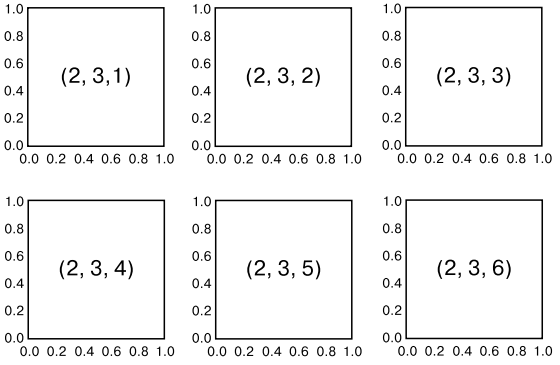

Рис. 4.62. plt.subplot() с выровненными полями

Мы воспользовались аргументами `hspace` и `wspace` функции `plt.subplots_adjust()`, позволяющими задать поля по высоте и ширине рисунка в единицах высоты субграфика (в данном случае поля составляют 40 % от ширины и высоты субграфика).

Функция plt.subplots: создание всей сетки за один раз

Только что описанный подход может оказаться довольно трудоемким при создании большой сетки субграфиков, особенно если нужно скрыть метки осей *X* и *Y* на внутренних графиках. В этом случае удобнее использовать функцию `plt.subplots()` (обратите внимание на букву `s` в конце `subplots`). Вместо отдельного субграфика эта функция создает целую сетку субграфиков одной строкой кода и возвращает их в массиве NumPy. Ее аргументы: количество строк и столбцов, а также необязательные ключевые слова `sharex` и `sharey`, позволяющие задавать связи между различными системами координат.

Здесь мы создаем сетку 2 × 3 субграфиков, в которой у всех систем координат в одной строке одинаковая шкала по оси *Y*, а у всех систем координат в одном столбце — одинаковая шкала по оси *X* (рис. 4.63):

```
In[6]: fig, ax = plt.subplots(2, 3, sharex='col', sharey='row')
```

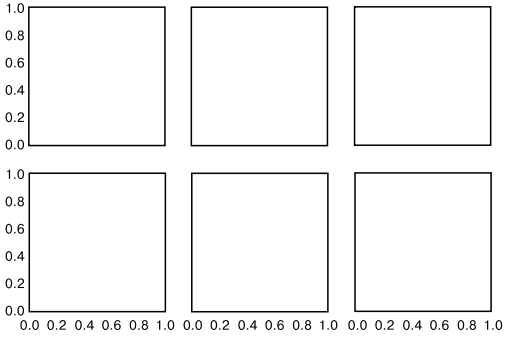

Рис. 4.63. Общие оси координат *X* и *Y* при использовании plt.subplots()

Обратите внимание, что указание ключевых слов `sharex` и `sharey` приводит к автоматическому удалению внутренних меток с сетки в целях очистки пространства графика. Итоговая сетка систем координат возвращается в массиве NumPy, что дает возможность легко ссылаться на требуемую систему координат с помощью обычной индексации, используемой для массивов (рис. 4.64):

```
In[7]: # Системы координат располагаются в двумерном массиве,
       # индексируемом по [строка, столбец]
       for i in range(2):
           for j in range(3):
               ax[i, j].text(0.5, 0.5, str((i, j)),
                             fontsize=18, ha='center')
       fig
```

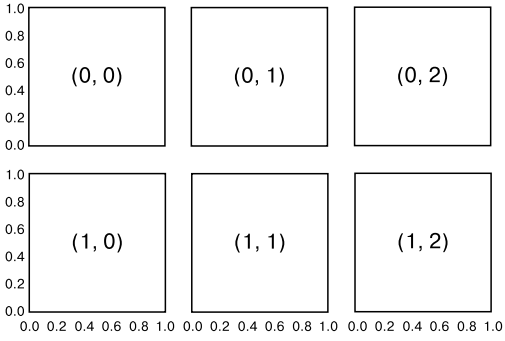

Рис. 4.64. Нумерация графиков в сетке субграфиков

По сравнению с `plt.subplot()` функция `plt.subplots()` намного лучше согласуется с принятой в языке Python индексацией, начинающейся с 0.

Функция plt.GridSpec: более сложные конфигурации

При выходе за пределы обычной сетки графиков к субграфикам, занимающим много строк и столбцов, наилучшим инструментом считается `plt.GridSpec`. Сам по себе объект `plt.GridSpec` не создает графиков, это просто удобный интерфейс, понятный команде `plt.subplot()`. Например, вызов `GridSpec` для сетки из двух строк и трех столбцов с заданными значениями ширины и высоты будет выглядеть следующим образом:

```
In[8]: grid = plt.GridSpec(2, 3, wspace=0.4, hspace=0.3)
```

Затем мы можем задать местоположение и размеры субграфиков с помощью обычного синтаксиса срезов языка Python (рис. 4.65):

```
In[9]: plt.subplot(grid[0, 0])
       plt.subplot(grid[0, 1:])
       plt.subplot(grid[1, :2])
       plt.subplot(grid[1, 2]);
```

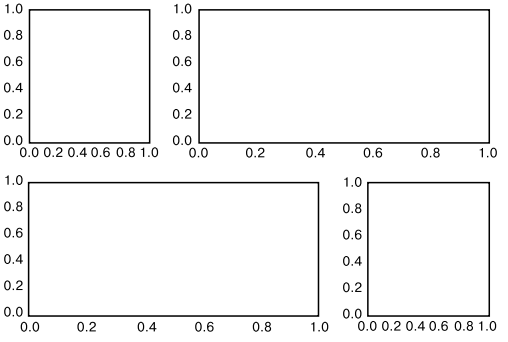

Рис. 4.65. Создание субграфиков неодинаковой формы с помощью функции plt.GridSpec

Подобное гибкое выравнивание сетки находит множество различных применений. Я чаще всего использую его при создании графиков гистограмм с несколькими системами координат (рис. 4.66):

```
In[10]: # Создаем нормально распределенные данные
        mean = [0, 0]
        cov = [[1, 1], [1, 2]]
        x, y = np.random.multivariate_normal(mean, cov, 3000).T

        # Задаем системы координат с помощью функции GridSpec
        fig = plt.figure(figsize=(6, 6))
        grid = plt.GridSpec(4, 4, hspace=0.2, wspace=0.2)
        main_ax = fig.add_subplot(grid[:-1, 1:])
        y_hist = fig.add_subplot(grid[:-1, 0], xticklabels=[], sharey=main_ax)
        x_hist = fig.add_subplot(grid[-1, 1:], yticklabels=[], sharex=main_ax)

        # Распределяем точки по основной системе координат
```

```
main_ax.plot(x, y, 'ok', markersize=3, alpha=0.2)

# Рисуем гистограммы на дополнительных системах координат
x_hist.hist(x, 40, histtype='stepfilled',
            orientation='vertical', color='gray')
x_hist.invert_yaxis()
y_hist.hist(y, 40, histtype='stepfilled',
            orientation='horizontal', color='gray')
y_hist.invert_xaxis()
```

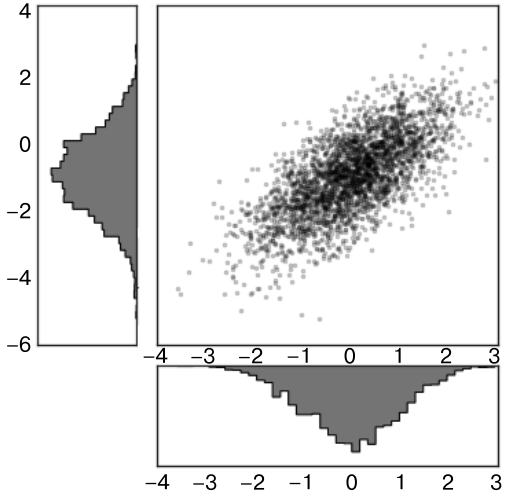

Рис. 4.66. Визуализируем многомерные распределения с помощью функции plt.GridSpec

Такое распределение, выводимое на отдельных графиках по бокам, настолько распространено, что в пакете Seaborn для построения его графиков предусмотрено отдельное API. Подробную информацию см. в разделе «Визуализация с помощью библиотеки Seaborn» данной главы.

Текст и поясняющие надписи

Хорошая визуализация должна рассказывать читателю историю. В некоторых случаях это можно сделать только визуальными средствами, без дополнительного текста, но иногда небольшие текстовые подсказки и метки необходимы. Вероятно, простейший вид поясняющих надписей — метки на осях координат и их названия, но имеющиеся возможности гораздо шире. Рассмотрим на примере каких-нибудь данных, как можно их визуализировать и добавить поясняющие надписи, чтобы облегчить донесение до читателя полезной информации. Начнем с настройки блокнота для построения графиков и импорта необходимых нам функций:

```
In[1]: %matplotlib inline
       import matplotlib.pyplot as plt
```

```
import matplotlib as mpl
plt.style.use('seaborn-whitegrid')
import numpy as np
import pandas as pd
```

Пример: влияние выходных дней на рождение детей в США

Вернемся к данным, с которыми мы работали ранее в разделе «Пример: данные о рождаемости» главы 3, где мы сгенерировали график среднего количества рождений детей в зависимости от дня календарного года. Эти данные можно скачать по адресу https://raw.githubusercontent.com/jakevdp/data-CDCbirths/master/births.csv.

Начнем с той же процедуры очистки данных, которую мы использовали ранее, и построим график результатов (рис. 4.67):

```
In[2]:
births = pd.read_csv('births.csv')

quartiles = np.percentile(births['births'], [25, 50, 75])
mu, sig = quartiles[1], 0.74 * (quartiles[2] - quartiles[0])
births = births.query('(births > @mu - 5 * @sig) & (births < @mu + 5 * @sig)')

births['day'] = births['day'].astype(int)

births.index = pd.to_datetime(10000 * births.year +
                              100 * births.month +
                              births.day, format='%Y%m%d')
births_by_date = births.pivot_table('births',
                                    [births.index.month, births.index.day])
births_by_date.index = [pd.datetime(2012, month, day)
                        for (month, day) in births_by_date.index]

In[3]: fig, ax = plt.subplots(figsize=(12, 4))
       births_by_date.plot(ax=ax);
```

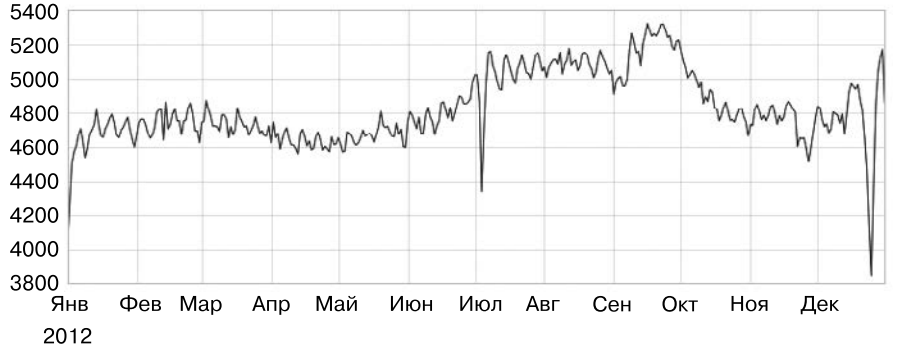

Рис. 4.67. Среднее ежедневное количество новорожденных в зависимости от даты

При работе с данными подобным образом часто бывает полезно снабдить элементы графика пояснениями для привлечения к ним внимания читателя. Это можно сделать вручную с помощью команды `plt.text/ax.text`, которая поместит текст в месте, соответствующем конкретным значениям координат *(x, y)* (рис. 4.68):

```
In[4]: fig, ax = plt.subplots(figsize=(12, 4))
       births_by_date.plot(ax=ax)

       # Добавляем метки на график
       style = dict(size=10, color='gray')

       ax.text('2012-1-1', 3950, "New Year's Day", **style)
       ax.text('2012-7-4', 4250, "Independence Day", ha='center', **style)
       ax.text('2012-9-4', 4850, "Labor Day", ha='center', **style)
       ax.text('2012-10-31', 4600, "Halloween", ha='right', **style)
       ax.text('2012-11-25', 4450, "Thanksgiving", ha='center', **style)
       ax.text('2012-12-25', 3850, "Christmas ", ha='right', **style)

       # Добавляем метки для осей координат
       ax.set(title='USA births by day of year (1969-1988)',
              ylabel='average daily births')

       # Размечаем ось X центрированными метками для месяцев
       ax.xaxis.set_major_locator(mpl.dates.MonthLocator())
       ax.xaxis.set_minor_locator(mpl.dates.MonthLocator(bymonthday=15))
       ax.xaxis.set_major_formatter(plt.NullFormatter())
       ax.xaxis.set_minor_formatter(mpl.dates.DateFormatter('%h'));
```

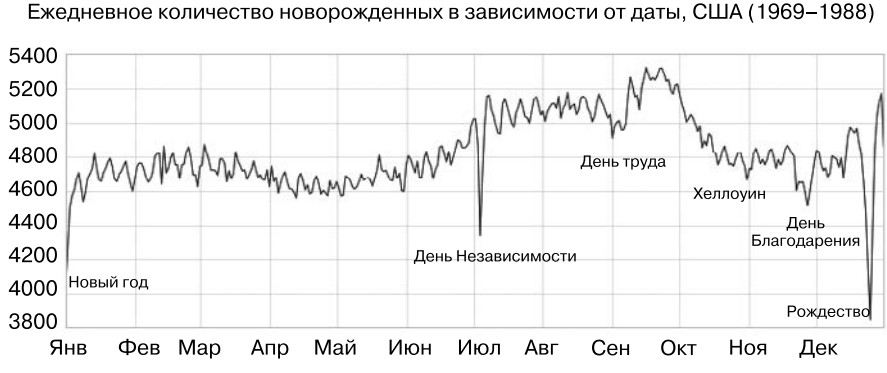

Рис. 4.68. Ежедневное количество рождаемых детей в зависимости от даты, с комментариями

Метод `ax.text` принимает на входе координату *x*, координату *y*, строковое значение и необязательные ключевые слова, задающие цвет, размер, стиль, выравнивание и другие свойства текста. В данном случае мы использовали значения `ha='right'` и `ha='center'`, где `ha` — сокращение от horizontal alignment («выравнивание по горизонтали»). См. дальнейшую информацию об имеющихся настройках в docstring функций `plt.text()` и `mpl.text.Text()`.

Преобразования и координаты текста

В предыдущем примере мы привязали наши текстовые пояснения к конкретным значениям данных. Иногда бывает удобнее привязать текст к координатам на осях рисунка, независимо от данных. В библиотеке Matplotlib это осуществляется путем модификации *преобразования* (transform).

Всем фреймворкам отображения графики необходимы схемы преобразования между разными системами координат. Например, точку данных с координатами *(x, y) = (1, 1)* следует представить в виде точки в определенном месте на рисунке, который, в свою очередь, необходимо представить в виде пикселов на экране. С математической точки зрения подобные преобразования несложны, и сама библиотека Matplotlib внутри использует для их выполнения имеющийся в ней набор инструментов (эти инструменты можно найти в подмодуле `matplotlib.transforms`).

Среднестатистический пользователь редко задумывается о деталях этих преобразований, но если речь идет о размещении текста на рисунке, не помешает иметь о них определенное представление. Существует три предопределенных преобразования, которые могут оказаться полезными в подобной ситуации:

- `ax.transData` — преобразование из системы координат данных;
- `ax.transAxes` — преобразование из системы координат объекта `Axes` (в единицах размеров рисунка);
- `fig.transFigure` — преобразование из системы координат объекта `Figure` (в единицах размеров рисунка)

Рассмотрим пример вывода текста в различных местах рисунка с помощью этих преобразований (рис. 4.69):

```
In[5]: fig, ax = plt.subplots(facecolor='lightgray')
       ax.axis([0, 10, 0, 10])

       # transform=ax.transData – значение по умолчанию,
       # но мы все равно указываем его
       ax.text(1, 5, ". Data: (1, 5)", transform=ax.transData)
       ax.text(0.5, 0.1, ". Axes: (0.5, 0.1)", transform=ax.transAxes)
       ax.text(0.2, 0.2, ". Figure: (0.2, 0.2)", transform=fig.transFigure);
```

Обратите внимание, что по умолчанию текст выравнивается по базовой линии и левому краю указанных координат, поэтому `"."` в начале каждой строки приблизительно отмечает здесь заданные координаты.

Координаты `transData` задают обычные координаты данных, соответствующие меткам на осях *X* и *Y*. Координаты `transAxes` задают местоположение, считая от нижнего левого угла системы координат (здесь — белый прямоугольник), в виде

доли от размера системы координат. Координаты `transFigure` схожи с `transAxes`, но задают местоположение, считая от нижнего левого угла рисунка (здесь — серый прямоугольник) в виде доли от размера рисунка.

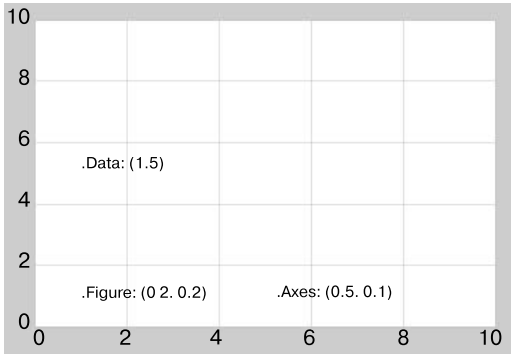

Рис. 4.69. Сравнение различных систем координат библиотеки Matplotlib

Отмечу, что, если поменять пределы осей координат, это повлияет только на координаты `transData`, а другие останутся неизменными (рис. 4.70):

```
In[6]: ax.set_xlim(0, 2)
       ax.set_ylim(-6, 6)
       fig
```

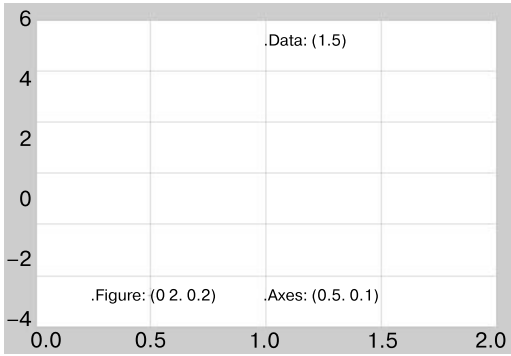

Рис. 4.70. Сравнение различных систем координат библиотеки Matplotlib

Наблюдать это поведение более наглядно можно путем интерактивного изменения пределов осей координат. При выполнении кода в блокноте этого можно добиться, заменив `%matplotlib inline` на `%matplotlib notebook` и воспользовавшись меню каждого из графиков для работы с ним.

Стрелки и поясняющие надписи

Наряду с отметками делений и текстом удобной поясняющей меткой является простая стрелка.

Рисование стрелок в Matplotlib зачастую оказывается более сложной задачей, чем вы могли бы предполагать. Несмотря на существование функции `plt.arrow()`, использовать ее я бы не советовал: создаваемые ею стрелки представляют собой SVG-объекты, подверженные изменениям в зависимости от соотношения сторон графиков, поэтому результат редко оказывается соответствующим ожиданиям. Вместо этого я предложил бы воспользоваться функцией `plt.annotate()`. Она создает текст и стрелку, причем позволяет очень гибко задавать настройки для стрелки.

В следующем фрагменте кода мы используем функцию `annotate` с несколькими параметрами (рис. 4.71):

```
In[7]: %matplotlib inline

       fig, ax = plt.subplots()

       x = np.linspace(0, 20, 1000)
       ax.plot(x, np.cos(x))
       ax.axis('equal')

       ax.annotate('local maximum', xy=(6.28, 1), xytext=(10, 4),
                   arrowprops=dict(facecolor='black', shrink=0.05))

       ax.annotate('local minimum', xy=(5 * np.pi, -1), xytext=(2, -6),
                   arrowprops=dict(arrowstyle="->",
                   connectionstyle="angle3,angleA=0,angleB=-90"));
```

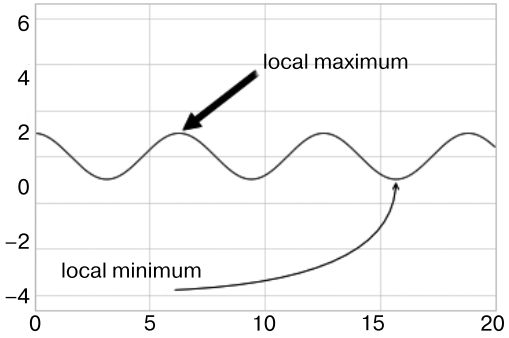

Рис. 4.71. Примеры поясняющих надписей

Стилем стрелки можно управлять с помощью словаря `arrowprops` с множеством параметров. Эти параметры отлично описаны в онлайн-документации библиотеки

Matplotlib, поэтому вместо перечисления их я просто покажу несколько возможностей. Продемонстрируем часть имеющихся параметров на уже знакомом вам графике рождаемости (рис. 4.72):

```
In[8]:
fig, ax = plt.subplots(figsize=(12, 4))
births_by_date.plot(ax=ax)

# Добавляем на график метки
ax.annotate("New Year's Day", xy=('2012-1-1', 4100),  xycoords='data',
            xytext=(50, -30), textcoords='offset points',
            arrowprops=dict(arrowstyle="->",
                            connectionstyle="arc3,rad=-0.2"))

ax.annotate("Independence Day", xy=('2012-7-4', 4250),  xycoords='data',
            bbox=dict(boxstyle="round", fc="none", ec="gray"),
            xytext=(10, -40), textcoords='offset points', ha='center',
            arrowprops=dict(arrowstyle="->"))

ax.annotate('Labor Day', xy=('2012-9-4', 4850), xycoords='data', ha='center',
            xytext=(0, -20), textcoords='offset points')
ax.annotate('', xy=('2012-9-1', 4850), xytext=('2012-9-7', 4850),
            xycoords='data', textcoords='data',
            arrowprops={'arrowstyle': '|-|,widthA=0.2,widthB=0.2', })


ax.annotate('Halloween', xy=('2012-10-31', 4600),  xycoords='data',
            xytext=(-80, -40), textcoords='offset points',
            arrowprops=dict(arrowstyle="fancy",
                            fc="0.6", ec="none",
                            connectionstyle="angle3,angleA=0,angleB=-90"))

ax.annotate('Thanksgiving', xy=('2012-11-25', 4500),  xycoords='data',
            xytext=(-120, -60), textcoords='offset points',
            bbox=dict(boxstyle="round4,pad=.5", fc="0.9"),
            arrowprops=dict(arrowstyle="->",
                  connectionstyle="angle,angleA=0,angleB=80,rad=20"))
ax.annotate('Christmas', xy=('2012-12-25', 3850),  xycoords='data',
             xytext=(-30, 0), textcoords='offset points',
             size=13, ha='right', va="center",
             bbox=dict(boxstyle="round", alpha=0.1),
             arrowprops=dict(arrowstyle="wedge, tail_width=0.5",
                                          alpha=0.1));

# Задаем метки для осей координат
ax.set(title='USA births by day of year (1969-1988)',
       ylabel='average daily births')

# Размечаем ось X центрированными метками для месяцев
ax.xaxis.set_major_locator(mpl.dates.MonthLocator())
ax.xaxis.set_minor_locator(mpl.dates.MonthLocator(bymonthday=15))
ax.xaxis.set_major_formatter(plt.NullFormatter())
```

```
ax.xaxis.set_minor_formatter(mpl.dates.DateFormatter('%h'));

ax.set_ylim(3600, 5400);
```

Вы видите, что спецификации стрелок и текстовых полей очень подробны. Благодаря этому мы можем создавать стрелки практически любого нужного нам вида. К сожалению, это также означает, что подобные элементы требуют отладки вручную — процесс, занимающий немало времени, если речь идет о создании графики типографского уровня качества! Наконец, отмечу, что использовать для представления данных продемонстрированную выше смесь стилей я отнюдь не рекомендую, она дана в качестве примера возможностей.

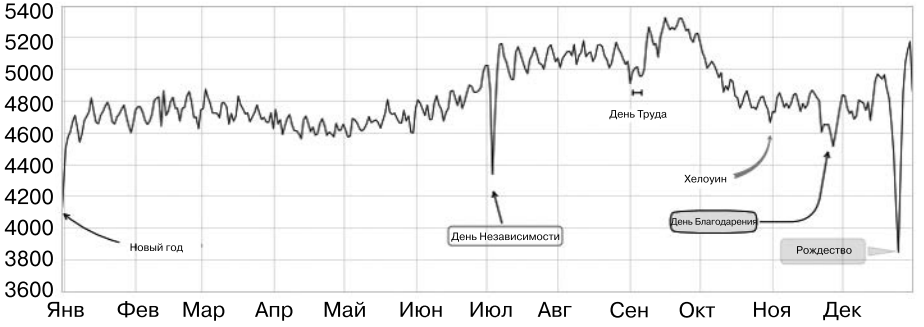

Рис. 4.72. Средняя рождаемость по дням с пояснениями

Дальнейшее обсуждение и примеры стилей стрелок и поясняющих надписей можно найти в галерее библиотеки Matplotlib по адресу http://matplotlib.org/examples/pylab_examples/annotation_demo2.html.

Пользовательские настройки делений на осях координат

Локаторы и форматеры делений на осях, используемые по умолчанию в библиотеке Matplotlib, спроектированы так, что в большинстве обычных ситуаций их вполне достаточно, хотя они отнюдь не оптимальны для всех графиков. В этом разделе мы рассмотрим несколько примеров настройки расположения делений и их форматирования для конкретных интересующих нас видов графиков.

Прежде чем перейти к примерам, следует разобраться в объектной иерархии графиков библиотеки Matplotlib. Matplotlib старается делать объектами языка Python все элементы на графике, например объект `figure` — ограничивающий снаружи все элементы графика прямоугольник. Каждый объект библиотеки Matplotlib также служит контейнером подобъектов. Например, любой объект `figure` может

содержать один объект `axes` или более, каждый из которых, в свою очередь, содержит другие объекты, отражающие содержимое графика.

Метки делений не исключение. У каждого объекта `axes` имеются атрибуты `xaxis` и `yaxis`, которые, в свою очередь, содержат все свойства линий, делений и меток оси координат.

Основные и промежуточные деления осей координат

На каждой оси координат имеются *основные* и *промежуточные* метки делений. Основные деления обычно больше или более заметны, а промежуточные — меньше. По умолчанию библиотека Matplotlib редко использует промежуточные деления, но одно из мест, где их можно увидеть, — логарифмические графики (рис. 4.73):

```
In[1]: %matplotlib inline
       import matplotlib.pyplot as plt
       plt.style.use('seaborn-whitegrid')
       import numpy as np

In[2]: ax = plt.axes(xscale='log', yscale='log')
```

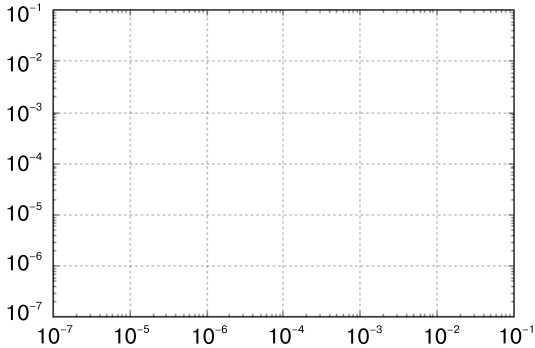

Рис. 4.73. Пример логарифмических шкал и меток

На этом графике мы видим, что каждое основное деление состоит из большого деления и метки, а промежуточное — из маленького деления без метки.

Можно задать пользовательские настройки для этих свойств делений (расположений и меток), задав значения объектов `formatter` и `locator` каждой из осей. Рассмотрим их значения для оси *X* на графике (см. рис. 4.73):

```
In[3]: print(ax.xaxis.get_major_locator())
       print(ax.xaxis.get_minor_locator())

<matplotlib.ticker.LogLocator object at 0x107530cc0>
<matplotlib.ticker.LogLocator object at 0x107530198>
```

```
In[4]: print(ax.xaxis.get_major_formatter())
       print(ax.xaxis.get_minor_formatter())

<matplotlib.ticker.LogFormatterMathtext object at 0x107512780>
<matplotlib.ticker.NullFormatter object at 0x10752dc18>
```

Мы видим, что расположение меток как основных, так и промежуточных делений задает локатор `LogLocator` (что логично для логарифмического графика). Метки промежуточных делений форматируются форматером `NullFormatter` (это означает, что метки отображаться не будут).

Продемонстрируем несколько примеров настройки этих локаторов и форматеров для различных графиков.

Прячем деления и/или метки

Наиболее частая операция с делениями/метками — скрывание делений или меток с помощью классов `plt.NullLocator()` и `plt.NullFormatter()`, как показано на рис. 4.74:

```
In[5]: ax = plt.axes()
       ax.plot(np.random.rand(50))

       ax.yaxis.set_major_locator(plt.NullLocator())
       ax.xaxis.set_major_formatter(plt.NullFormatter())
```

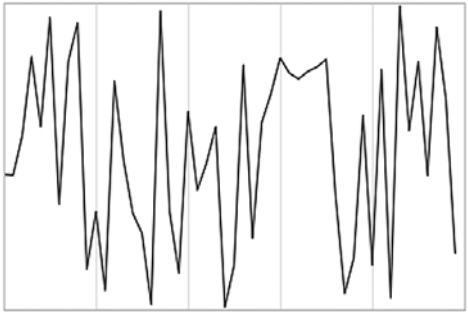

Рис. 4.74. График со скрытыми метками делений (ось *X*) и скрытыми делениями (ось *Y*)

Обратите внимание, что мы убрали метки (но оставили деления/линии координатной сетки) с оси *X*, и убрали деления (а следовательно, и метки) с оси *Y*. Отсутствие делений может быть полезно во многих случаях, например, если нужно отобразить сетку изображений. Например, рассмотрим рис. 4.75, содержащий

изображения лиц людей, — пример, часто используемый в задачах машинного обучения с учителем (более подробную информацию вы можете найти в разделе «Заглянем глубже: метод опорных векторов» главы 5):

```
In[6]: fig, ax = plt.subplots(5, 5, figsize=(5, 5))
       fig.subplots_adjust(hspace=0, wspace=0)

       # Получаем данные по лицам людей из библиотеки scikit-learn
       from sklearn.datasets import fetch_olivetti_faces
       faces = fetch_olivetti_faces().images

       for i in range(5):
           for j in range(5):
               ax[i, j].xaxis.set_major_locator(plt.NullLocator())
               ax[i, j].yaxis.set_major_locator(plt.NullLocator())
               ax[i, j].imshow(faces[10 * i + j], cmap="bone")
```

Рис. 4.75. Скрываем деления на графиках с изображениями

Обратите внимание, что у каждого изображения — отдельная система координат и мы сделали локаторы пустыми, поскольку значения делений (в данном случае количество пикселов) не несут никакой относящейся к делу информации.

Уменьшение или увеличение количества делений

Распространенная проблема с настройками по умолчанию — то, что метки на маленьких субграфиках могут оказаться расположенными слишком близко друг к другу. Это заметно на сетке графиков, показанной на рис. 4.76:

```
In[7]: fig, ax = plt.subplots(4, 4, sharex=True, sharey=True)
```

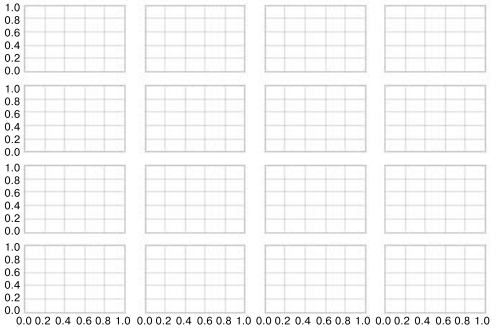

Рис. 4.76. Вид рисунка по умолчанию со слишком плотно расположенными делениями

Числа практически накладываются друг на друга, особенно в делениях на оси *X*, из-за чего их очень сложно разобрать. Исправить это можно с помощью класса `plt.MaxNLocator()`, который дает возможность задавать максимальное отображаемое количество делений. При задании этого числа конкретные местоположения делений выберет внутренняя логика библиотеки Matplotlib (рис. 4.77):

```
In[8]: # Задаем, для всех систем координат, локаторы основных делений осей X и Y
           for axi in ax.flat:
           axi.xaxis.set_major_locator(plt.MaxNLocator(3))
           axi.yaxis.set_major_locator(plt.MaxNLocator(3))
       fig
```

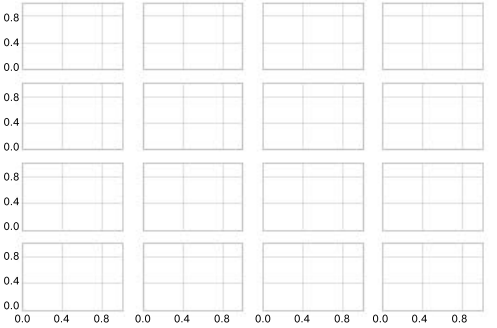

Рис. 4.77. Пользовательские настройки количества делений

Благодаря этому рисунок становится гораздо понятнее. При необходимости еще более точного контроля расположения делений с равными интервалами можно воспользоваться классом `plt.MultipleLocator`, который мы обсудим в следующем разделе.

Более экзотические форматы делений

Форматирование делений, используемое по умолчанию в библиотеке Matplotlib, оставляет желать лучшего. В качестве варианта по умолчанию, который бы

подходил для широкого спектра ситуаций, оно работает неплохо, но иногда требуется нечто большее. Рассмотрим показанный на рис. 4.78 график синуса и косинуса:

```
In[9]: # Строим графики синуса и косинуса
       fig, ax = plt.subplots()
       x = np.linspace(0, 3 * np.pi, 1000)
       ax.plot(x, np.sin(x), lw=3, label='Sine')
       ax.plot(x, np.cos(x), lw=3, label='Cosine')

       # Настраиваем сетку, легенду и задаем пределы осей координат
       ax.grid(True)
       ax.legend(frameon=False)
       ax.axis('equal')
       ax.set_xlim(0, 3 * np.pi);
```

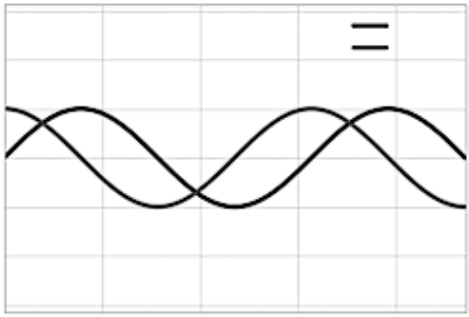

Рис. 4.78. График по умолчанию с целочисленными делениями

Хотелось бы внести несколько изменений. Во-первых, для такого рода данных лучше располагать деления и линии сетки по кратным числу π точках. Сделать это можно путем задания локатора `MultipleLocator`, располагающего деления в точках, кратных переданному ему числу. В дополнение добавим промежуточные деления в точках, кратных $\pi/4$ (рис. 4.79):

```
In[10]: ax.xaxis.set_major_locator(plt.MultipleLocator(np.pi / 2))
        ax.xaxis.set_minor_locator(plt.MultipleLocator(np.pi / 4))
        fig
```

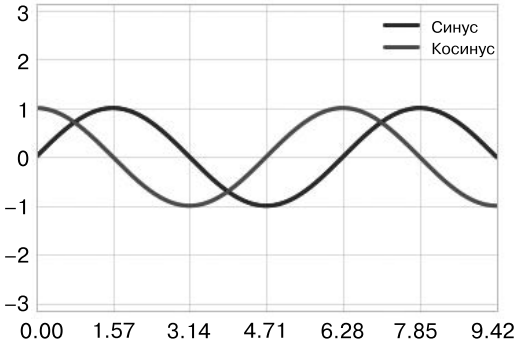

Рис. 4.79. Деления в точках, кратных $\pi/2$

Но теперь эти метки делений выглядят несколько по-дурацки: можно догадаться, что они кратны ϖ, но из десятичного представления сразу это непонятно. Необходимо модифицировать форматер деления. Встроенного форматера для нашей задачи нет, поэтому мы воспользуемся форматером `plt.FuncFormatter`, принимающим на входе пользовательскую функцию, обеспечивающую более точный контроль за форматом вывода делений (рис. 4.80):

```
In[11]: def format_func(value, tick_number):
            # Определяем количество кратных пи/2 значений
            # [в требуемом диапазоне]
            N = int(np.round(2 * value / np.pi))
            if N == 0:
                return "0"
            elif N == 1:
                return r"$\pi/2$"
            elif N == 2:
                return r"$\pi$"
            elif N % 2 > 0:
                return r"${0}\pi/2$".format(N)
            else:
                return r"${0}\pi$".format(N // 2)

        ax.xaxis.set_major_formatter(plt.FuncFormatter(format_func))
        fig
```

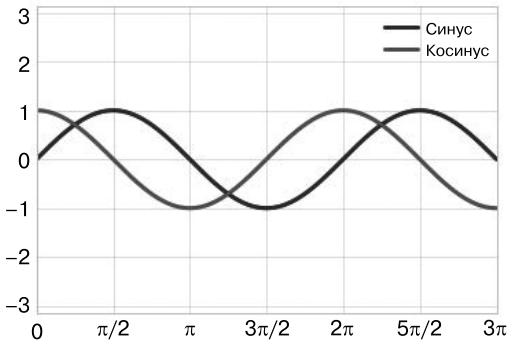

Рис. 4.80. Деления с пользовательскими метками

Намного лучше! Обратите внимание: мы воспользовались тем, что библиотека Matplotlib поддерживает систему верстки LaTeX. Для ее использования необходимо заключить нужную строку в знаки доллара с двух сторон. Это облегчает отображение математических символов и формул. В нашем случае `"$\pi$"` визуализируется в виде греческой буквы π.

Форматер `plt.FuncFormatter` обеспечивает возможность чрезвычайно тонкого контроля внешнего вида делений графика и очень удобен при подготовке графиков для презентаций или публикации.

Краткая сводка локаторов и форматеров

Мы уже упоминали несколько имеющихся форматеров и локаторов. В заключение этого раздела мы перечислим все встроенные локаторы и форматеры. Дальнейшую информацию о них вы можете найти в их docstring или в онлайн-документации библиотеки Matplotlib. Все форматеры и локаторы доступны в пространстве имен `plt` (табл. 4.1, 4.2).

Таблица 4.1. Локаторы

Класс локатора	Описание
NullLocator	Без делений
FixedLocator	Положения делений фиксированы
IndexLocator	Локатор для графика индексированной переменной (например, для x = range(len(y)))
LinearLocator	Равномерно распределенные деления от min до max
LogLocator	Логарифмически распределенные деления от min до max
MultipleLocator	Деления и диапазон значений кратны заданному основанию
MaxNLocator	Находит удачные местоположения для делений в количестве, не превышающем заданного максимального числа
AutoLocator	(По умолчанию.) MaxNLocator с простейшими значениями по умолчанию
AutoMinorLocator	Локатор для промежуточных делений

Таблица 4.2. Форматеры

Класс форматера	Описание
NullFormatter	Деления без меток
IndexFormatter	Задает строковые значения для меток на основе списка меток
FixedFormatter	Позволяет задавать строковые значения для меток вручную
FuncFormatter	Значения меток задаются с помощью пользовательской функции
FormatStrFormatter	Для всех значений используется строка формата
ScalarFormatter	(По умолчанию.) Форматер для скалярных значений
LogFormatter	Форматер по умолчанию для логарифмических систем координат

Далее мы рассмотрим дополнительные примеры этих классов.

Пользовательские настройки Matplotlib: конфигурации и таблицы стилей

Пользователи часто жалуются на настройки графиков по умолчанию в библиотеке Matplotlib. Хотя в выпуске 2.0 библиотеки Matplotlib многое должно измениться, умение настраивать значения по умолчанию поможет вам привести этот пакет в соответствие с вашими собственными эстетическими предпочтениями.

Здесь мы рассмотрим некоторые параметры конфигурации среды (rc) библиотеки Matplotlib и новую возможность — *таблицы стилей* (stylesheets), содержащие неплохие наборы конфигураций по умолчанию.

Выполнение пользовательских настроек графиков вручную

Ранее в этой главе мы видели, что можно менять отдельные настройки графиков, получая в итоге нечто более приятное глазу, чем настройки по умолчанию. Эти настройки можно выполнять и для каждого графика отдельно. Например, вот довольно скучная гистограмма по умолчанию (рис. 4.81):

```
In[1]: import matplotlib.pyplot as plt
       plt.style.use('classic')
       import numpy as np

       %matplotlib inline

In[2]: x = np.random.randn(1000)
       plt.hist(x);
```

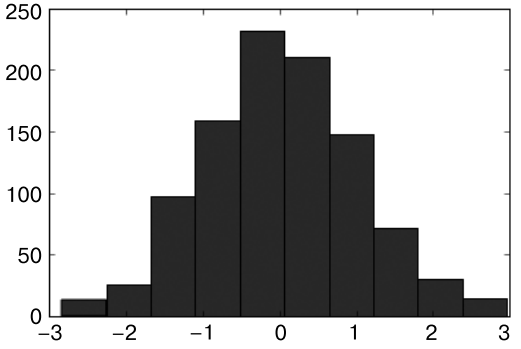

Рис. 4.81. Гистограмма в стиле библиотеки Matplotlib по умолчанию

Мы можем настроить ее вид вручную, превратив эту гистограмму в намного более приятный глазу график, показанный на рис. 4.82:

```
In[3]: # Используем серый фон
       ax = plt.axes(axisbg='#E6E6E6')
       ax.set_axisbelow(True)

       # Рисуем сплошные белые линии сетки
       plt.grid(color='w', linestyle='solid')

       # Скрываем основные линии осей координат
       for spine in ax.spines.values():
           spine.set_visible(False)

       # Скрываем деления сверху и справа
```

```
ax.xaxis.tick_bottom()
ax.yaxis.tick_left()

# Осветляем цвет делений и меток
ax.tick_params(colors='gray', direction='out')
for tick in ax.get_xticklabels():
    tick.set_color('gray')
for tick in ax.get_yticklabels():
    tick.set_color('gray')

# Задаем цвет заливки и границ гистограммы
ax.hist(x, edgecolor='#E6E6E6', color='#EE6666');
```

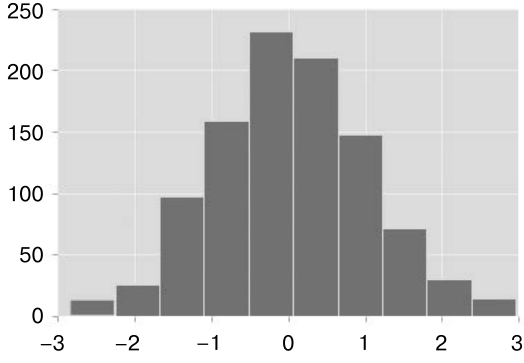

Рис. 4.82. Гистограмма с заданными вручную пользовательскими настройками

Выглядит намного лучше, и можно заметить, что этот вид был вдохновлен пакетом визуализации `ggplot` языка R. Для таких настроек потребовалось немало труда и не хотелось бы снова проделывать их все при каждом создании графика. К счастью, существует способ задать эти настройки один раз для всех графиков.

Изменяем значения по умолчанию: rcParams

Каждый раз при загрузке библиотеки Matplotlib она описывает конфигурацию среды (`rc`), содержащую стили по умолчанию для всех создаваемых вами элементов графиков. Эту конфигурацию можно настроить в любой момент, воспользовавшись удобной утилитой `plt.rc`. Рассмотрим, как можно модифицировать параметры `rc` таким образом, чтобы график по умолчанию выглядел схоже с вышеприведенным.

Начнем с сохранения копии текущего словаря `rcParams`, чтобы можно было без опасений восстановить эти значения в текущем сеансе:

```
In[4]: IPython_default = plt.rcParams.copy()
```

Теперь можно воспользоваться функцией `plt.rc` и изменить некоторые из настроек:

```
In[5]: from matplotlib import cycler
       colors = cycler('color',
                       ['#EE6666', '#3388BB', '#9988DD',
                        '#EECC55', '#88BB44', '#FFBBBB'])
       plt.rc('axes', facecolor='#E6E6E6', edgecolor='none',
              axisbelow=True, grid=True, prop_cycle=colors)
       plt.rc('grid', color='w', linestyle='solid')
       plt.rc('xtick', direction='out', color='gray')
       plt.rc('ytick', direction='out', color='gray')
       plt.rc('patch', edgecolor='#E6E6E6')
       plt.rc('lines', linewidth=2)
```

Описав эти настройки, мы можем создать график и посмотреть на них в действии (рис. 4.83):

```
In[6]: plt.hist(x);
```

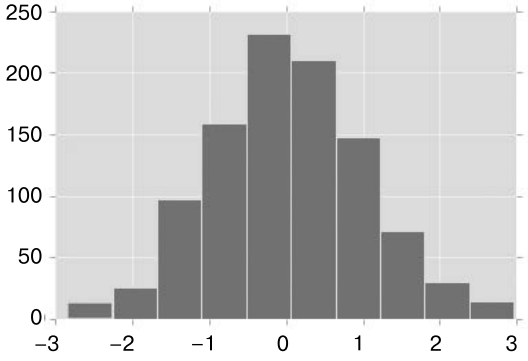

Рис. 4.83. Пользовательская гистограмма с настройками rc

Посмотрим, как выглядят с этими параметрами `rc` простые графики (рис. 4.84):

```
In[7]: for i in range(4):
           plt.plot(np.random.rand(10))
```

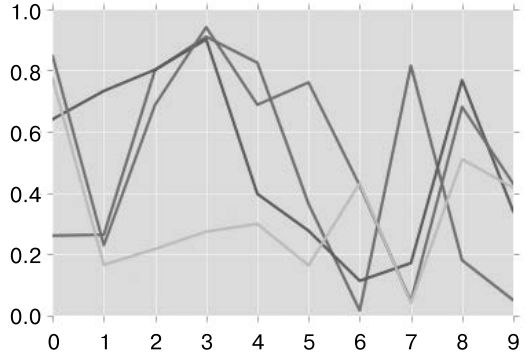

Рис. 4.84. Линейный график с пользовательскими стилями

Такие стили представляются мне гораздо более эстетически приятными, чем стили по умолчанию. Если мое чувство прекрасного расходится с вашим, то у меня есть для вас хорошая новость: вы можете настроить параметры `rc` под свой вкус! Эти настройки можно затем сохранить в файле `.matplotlibrc`, о котором можно прочитать в документации библиотеки Matplotlib. Но даже несмотря на это, я предпочитаю выполнять пользовательские настройки библиотеки Matplotlib с помощью ее таблиц стилей.

Таблицы стилей

В выпущенной в августе 2014 года версии 1.4 библиотеки Matplotlib был добавлен удобный модуль `style`, включающий немало новых таблиц стилей по умолчанию, а также возможность создавать и компоновать пользовательские стили. Формат этих таблиц стилей аналогичен упомянутому выше файлу `.matplotlibrc`, но расширение имен их файлов должно быть `.mplstyle`.

Даже если вы и не хотите создать свой собственный стиль, включенные по умолчанию таблицы стилей очень удобны. Имеющиеся стили перечислены в `plt.style.available` — для краткости привожу только первые пять:

```
In[8]: plt.style.available[:5]

Out[8]: ['fivethirtyeight',
         'seaborn-pastel',
         'seaborn-whitegrid',
         'ggplot',
         'grayscale']
```

Простейший способ переключения таблиц стилей — вызвать функцию:

```
plt.style.use('stylename')
```

Но не забывайте, что это приведет к изменению стиля на весь остаток сеанса! В качестве альтернативы можно воспользоваться контекстным менеджером стилей, устанавливающим стиль временно:

```
with plt.style.context('stylename'):
    make_a_plot()
```

Создадим функцию, рисующую два простейших вида графиков:

```
In[9]: def hist_and_lines():
           np.random.seed(0)
           fig, ax = plt.subplots(1, 2, figsize=(11, 4))
           ax[0].hist(np.random.randn(1000))
           for i in range(3):
               ax[1].plot(np.random.rand(10))
           ax[1].legend(['a', 'b', 'c'], loc='lower left')
```

Мы воспользуемся ею, чтобы увидеть вид этих графиков при использовании различных встроенных стилей.

Стиль по умолчанию

Стиль по умолчанию — тот, который мы видели до сих пор во всей книге. Начнем с него. Во-первых, восстановим нашу конфигурацию среды к имевшимся в блокноте значениям по умолчанию:

```
In[10]: # Восстанавливаем rcParams
        plt.rcParams.update(IPython_default);
```

Теперь посмотрим, как выглядят наши графики (рис. 4.85):

```
In[11]: hist_and_lines()
```

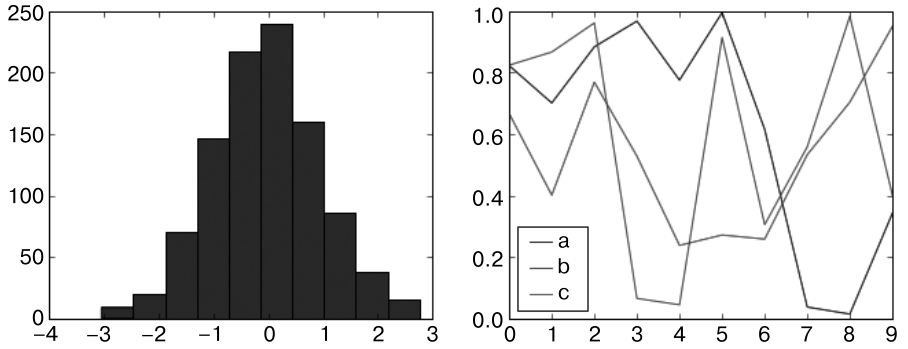

Рис. 4.85. Стиль библиотеки Matplotlib по умолчанию

Стиль FiveThirtyEight

Стиль FiveThirtyEight подражает оформлению популярного сайта FiveThirtyEight (http://fivethirtyeight.com/). Как вы можете видеть на рис. 4.86, он включает жирные шрифты, толстые линии и прозрачные оси координат.

```
In[12]: with plt.style.context('fivethirtyeight'):
            hist_and_lines()
```

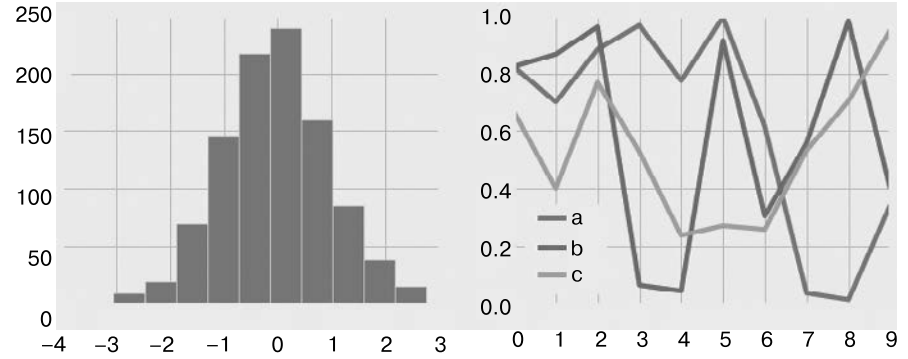

Рис. 4.86. Стиль FiveThirtyEight

ggplot

В языке программирования R пакет `ggplot` — очень популярное средство визуализации. Стиль `ggplot` библиотеки Matplotlib подражает стилям по умолчанию из этого пакета (рис. 4.87):

```
In[13]: with plt.style.context('ggplot'):
            hist_and_lines()
```

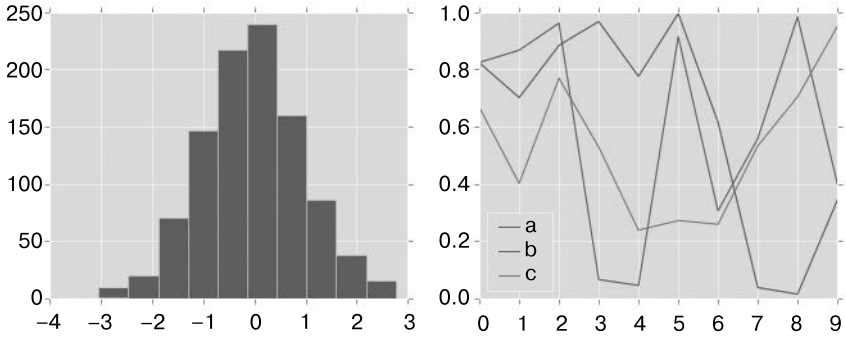

Рис. 4.87. Стиль ggplot

Стиль «байесовские методы для хакеров»

Существует замечательная онлайн-книга «Вероятностное программирование и байесовские методы для хакеров» (Probabilistic Programming and Bayesian Methods for Hackers, http://bit.ly/2fDJsKC). Она содержит рисунки, созданные с помощью библиотеки Matplotlib, и использует в книге для создания единообразного и приятного внешне стиля набор параметров `rc`. Этот стиль воспроизведен в таблице стилей `bmh` (рис. 4.88):

```
In[14]: with plt.style.context('bmh'):
            hist_and_lines()
```

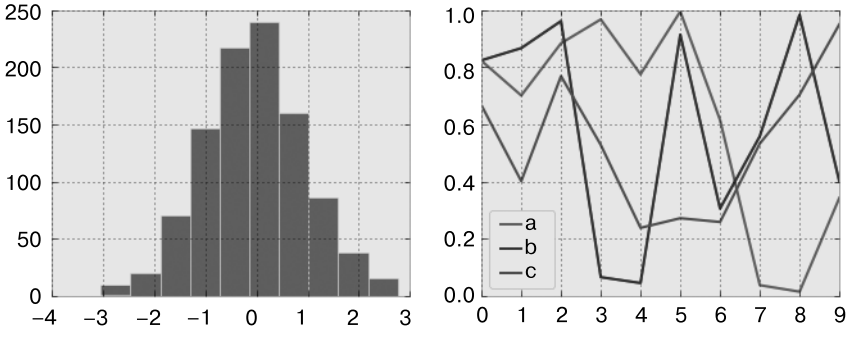

Рис. 4.88. Стиль bmh

Темный фон

Для используемых в презентациях рисунков часто удобнее темный, а не светлый фон. Эту возможность предоставляет стиль `dark_background` (рис. 4.89):

```
In[15]: with plt.style.context('dark_background'):
            hist_and_lines()
```

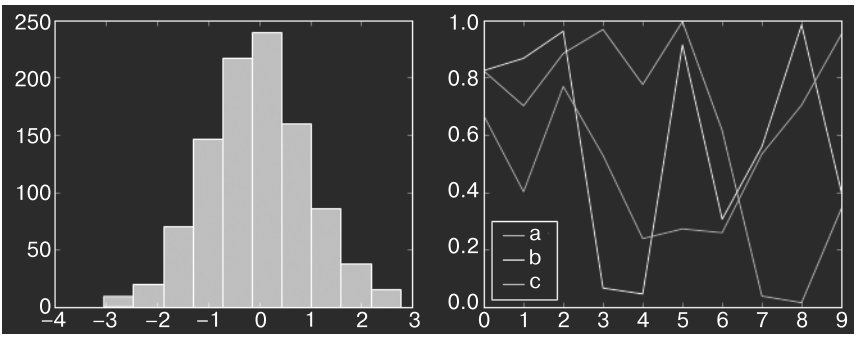

Рис. 4.89. Стиль dark_background

Оттенки серого

Иногда приходится готовить для печатного издания черно-белые рисунки. Для этого может пригодиться стиль `grayscale`, продемонстрированный на рис. 4.90:

```
In[16]: with plt.style.context('grayscale'):
            hist_and_lines()
```

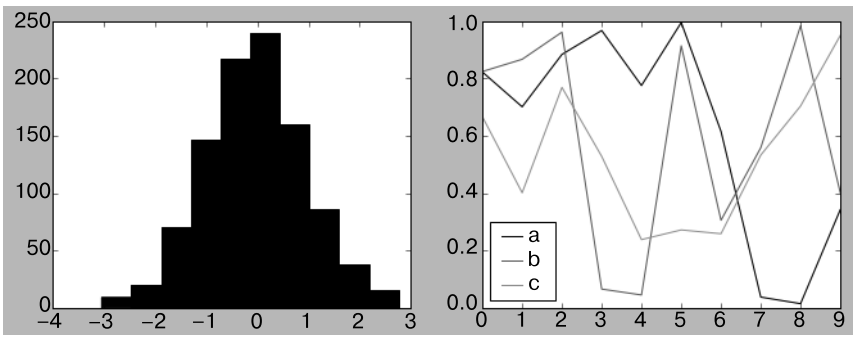

Рис. 4.90. Стиль grayscale

Стиль Seaborn

В библиотеке Matplotlib есть также стили, источником вдохновения для которых послужила библиотека Seaborn (обсуждаемая подробнее в разделе «Визуализация

с помощью библиотеки Seaborn» данной главы). Как мы увидим, эти стили загружаются автоматически при импорте пакета Seaborn в блокнот. Мне эти настройки очень нравятся, я склонен использовать их как настройки по умолчанию в моих собственных исследованиях данных (рис. 4.91):

```
In[17]: import seaborn
        hist_and_lines()
```

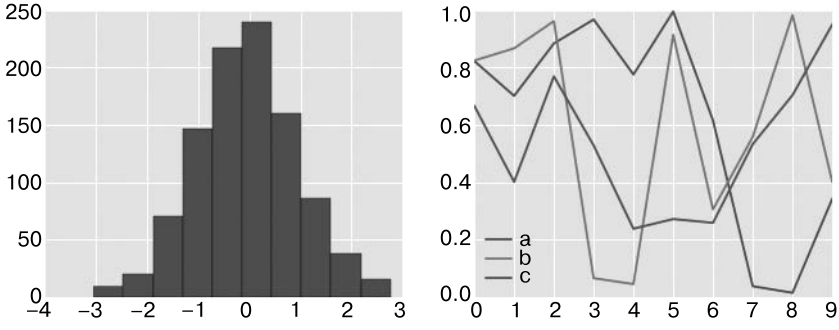

Рис. 4.91. Стили построения графиков библиотеки Seaborn

Благодаря всем этим встроенным вариантам различных стилей построения графиков библиотека Matplotlib становится гораздо более удобной как для интерактивной визуализации, так и создания рисунков для публикации. В этой книге я обычно использую при создании графиков один или несколько таких стилей.

Построение трехмерных графиков в библиотеке Matplotlib

Первоначально библиотека Matplotlib была создана в расчете на построение только двумерных графиков. В период выпуска версии 1.0 на основе имевшегося в библиотеке Matplotlib двумерного отображения графиков были созданы некоторые трехмерные утилиты построения графиков, что привело к удобному (хотя и несколько ограниченному в возможностях) набору инструментов для трехмерной визуализации данных. Мы активизируем возможность построения трехмерных графиков путем импорта набора инструментов `mplot3d`, включенного в основной установочный пакет библиотеки Matplotlib (рис. 4.92):

```
In[1]: from mpl_toolkits import mplot3d
```

После импорта этого подмодуля появляется возможность создавать трехмерные системы координат путем передачи ключевого слова `projection='3d'` любой из обычных функций создания систем координат:

```
In[2]: %matplotlib inline
       import numpy as np
       import matplotlib.pyplot as plt

In[3]: fig = plt.figure()
       ax = plt.axes(projection='3d')
```

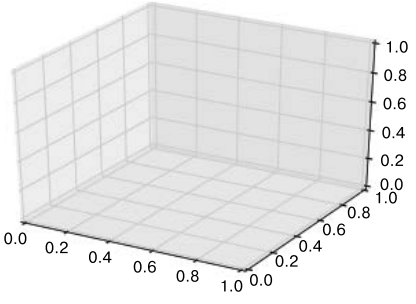

Рис. 4.92. Пустая трехмерная система координат

С помощью такой трехмерной системы координат можно строить различные виды трехмерных графиков. Построение трехмерных графиков — один из видов функциональности, для которого очень полезен интерактивный, а не статический просмотр рисунков в блокноте. Напомню, что для работы с интерактивными рисунками необходимо вместо команды `%matplotlib inline` использовать команду `%matplotlib notebook`.

Трехмерные точки и линии

Линейный график и диаграмма рассеяния — простейшие трехмерные графики, создаваемые на основе множеств кортежей из трех элементов (x, y, z). По аналогии с обсуждавшимися ранее более распространенными двумерными графиками их можно создать с помощью функций `ax.plot3D` и `ax.scatter3D`. Сигнатуры вызовов этих функций практически совпадают с их двумерными аналогами, поэтому вы можете обратиться к разделам «Простые линейные графики» и «Простые диаграммы рассеяния» этой главы за более подробной информацией по настройке вывода ими данных. Мы построим график тригонометрической спирали, а также нарисуем рядом с кривой несколько точек (рис. 4.93):

```
In[4]: ax = plt.axes(projection='3d')

       # Данные для трехмерной кривой
       zline = np.linspace(0, 15, 1000)
       xline = np.sin(zline)
```

```
yline = np.cos(zline)
ax.plot3D(xline, yline, zline, 'gray')

# Данные для трехмерных точек
zdata = 15 * np.random.random(100)
xdata = np.sin(zdata) + 0.1 * np.random.randn(100)
ydata = np.cos(zdata) + 0.1 * np.random.randn(100)
ax.scatter3D(xdata, ydata, zdata, c=zdata, cmap='Greens');
```

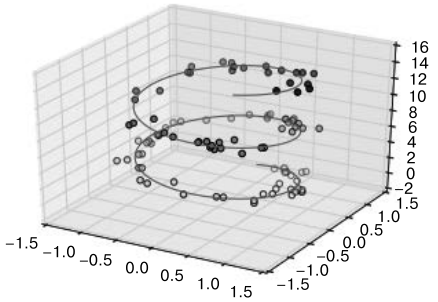

Рис. 4.93. Точки и линии в трех измерениях

Обратите внимание, что по умолчанию степень прозрачности рассеянных по графику точек настраивается таким образом, чтобы придать графику эффект глубины. Хотя в статическом изображении этот трехмерный эффект иногда незаметен, динамическое представление может дать информацию о топологии точек.

Трехмерные контурные графики

Аналогично контурным графикам, рассмотренным нами в разделе «Графики плотности и контурные графики» этой главы, `mplot3d` содержит инструменты для создания трехмерных рельефных графиков на основе тех же входных данных. Подобно двумерным графикам, создаваемым с помощью функции `ax.contour`, для функции `ax.contour3D` необходимо, чтобы все входные данные находились в форме двумерных регулярных сеток, с вычислением данных по оси Z в каждой точке. Мы продемонстрируем трехмерную контурную диаграмму трехмерной синусоиды (рис. 4.94):

```
In[5]: def f(x, y):
           return np.sin(np.sqrt(x ** 2 + y ** 2))

       x = np.linspace(-6, 6, 30)
       y = np.linspace(-6, 6, 30)

       X, Y = np.meshgrid(x, y)
```

```
        Z = f(X, Y)

In[6]: fig = plt.figure()
       ax = plt.axes(projection='3d')
       ax.contour3D(X, Y, Z, 50, cmap='binary')
       ax.set_xlabel('x')
       ax.set_ylabel('y')
       ax.set_zlabel('z');
```

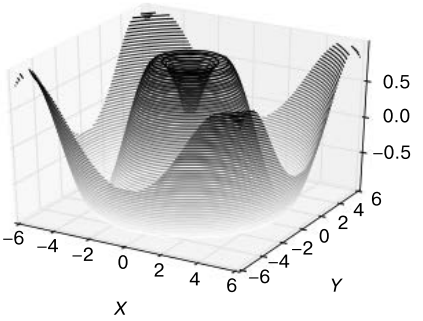

Рис. 4.94. Трехмерный контурный график

Иногда используемый по умолчанию угол зрения неидеален. В этом случае можно воспользоваться методом `view_init` для задания угла возвышения и азимутального угла. В нашем примере (результат которого показан на рис. 4.95) используется угол возвышения 60 градусов (то есть 60 градусов над плоскостью *X*-*Y*) и азимут 35 градусов (то есть график повернут на 35 градусов против часовой стрелки вокруг оси *Z*):

```
In[7]: ax.view_init(60, 35)
       fig
```

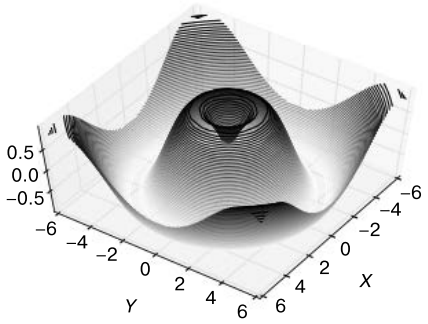

Рис. 4.95. Настройка угла зрения для трехмерного графика

Выполнить такое вращение можно и интерактивно, щелчком кнопкой мыши и перетаскиванием, при использовании одной из интерактивных прикладных частей библиотеки Matplotlib.

Каркасы и поверхностные графики

Каркасы и поверхностные графики — еще два типа трехмерных графиков, подходящих для данных с привязкой к координатам. Они выполняют проекцию координатных значений на заданную трехмерную поверхность, облегчая наглядное представление полученных трехмерных фигур. Вот пример с каркасом (рис. 4.96):

```
In[8]: fig = plt.figure()
       ax = plt.axes(projection='3d')
       ax.plot_wireframe(X, Y, Z, color='black')
       ax.set_title('wireframe'); # Каркас
```

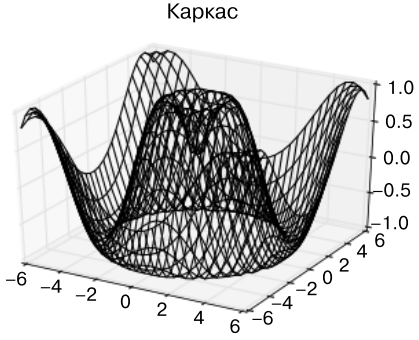

Рис. 4.96. График каркаса

Поверхностный график аналогичен графику каркаса, но каждая грань представляет собой заполненный многоугольник. Добавление карты цветов для заполненных многоугольников помогает лучше прочувствовать топологию визуализируемой поверхности (рис. 4.97):

```
In[9]: ax = plt.axes(projection='3d')
       ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                       cmap='viridis', edgecolor='none')
       ax.set_title('surface'); # Поверхность
```

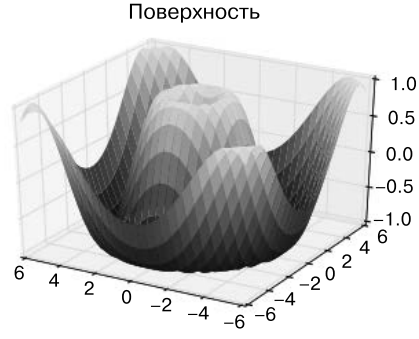

Рис. 4.97. Трехмерный поверхностный график

Обратите внимание, что, хотя координатные значения для поверхностного графика должны быть двумерными, он не обязан быть прямолинейным. Вот пример создания неполной сетки в полярной системе координат, которая при использовании для построения графика функции `surface3D` дает нам срез визуализируемой функции (рис. 4.98):

```
In[10]: r = np.linspace(0, 6, 20)
        theta = np.linspace(-0.9 * np.pi, 0.8 * np.pi, 40)
        r, theta = np.meshgrid(r, theta)

        X = r * np.sin(theta)
        Y = r * np.cos(theta)
        Z = f(X, Y)

        ax = plt.axes(projection='3d')
        ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                        cmap='viridis', edgecolor='none');
```

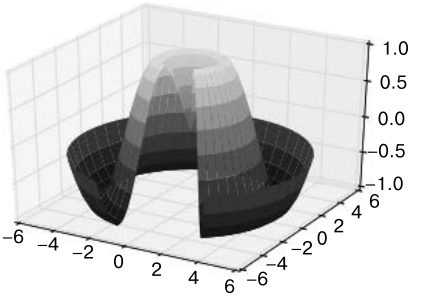

Рис. 4.98. Поверхностный график в полярной системе координат

Триангуляция поверхностей

В некоторых приложениях необходимые для предыдущих операций равномерно дискретизированные координатные сетки неудобны и накладывают слишком много ограничений. В таких случаях могут быть удобны графики, основанные на координатной сетке из треугольников. Что, если вместо равномерно выбранных значений из декартовой или полярной систем координат, мы имеем дело с набором случайно выбранных значений?

```
In[11]: theta = 2 * np.pi * np.random.random(1000)
        r = 6 * np.random.random(1000)
        x = np.ravel(r * np.sin(theta))
        y = np.ravel(r * np.cos(theta))
        z = f(x, y)
```

Нарисуем диаграмму рассеяния точек, чтобы представить себе вид поверхности, которую мы дискретизируем (рис. 4.99):

```
In[12]: ax = plt.axes(projection='3d')
        ax.scatter(x, y, z, c=z, cmap='viridis', linewidth=0.5);
```

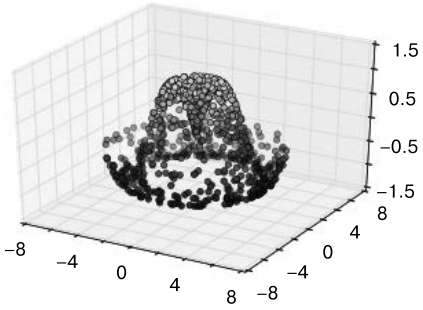

Рис. 4.99. Трехмерная дискретизированная поверхность

Полученное оставляет желать лучшего. В подобном случае нам поможет функция `ax.plot_trisurf`, предназначенная для создания поверхности путем поиска сначала набора треугольников, расположенных между смежными точками (результат показан на рис. 4.100. Напоминаем, что `x`, `y` и `z` — одномерные массивы):

```
In[13]: ax = plt.axes(projection='3d')
        ax.plot_trisurf(x, y, z, cmap='viridis', edgecolor='none');
```

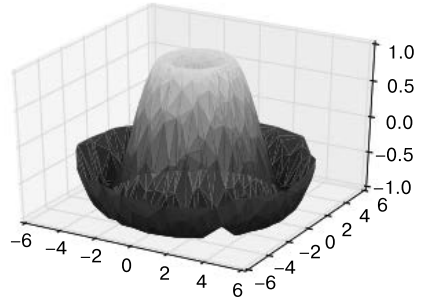

Рис. 4.100. Триангулированный участок поверхности

Результат не такой красивый, как при построении графика с помощью координатной сетки, но гибкость подобной триангуляции дает возможность создавать интересные трехмерные графики. Например, с помощью этого метода можно даже нарисовать трехмерную ленту Мебиуса.

Пример: визуализация ленты Мебиуса. Лента Мебиуса представляет собой полоску бумаги, склеенную в кольцо концами, перевернутыми на 180 градусов. Она весьма интересна топологически, поскольку у нее, несмотря на внешний вид, только одна сторона! В этом разделе мы визуализируем ее с помощью трехмерных инструментов библиотеки Matplotlib. Ключ к созданию ленты Мебиуса — ее параметризация. Это двумерная лента, поэтому нам понадобятся для нее две собственные координаты. Назовем одну из них θ (ее диапазон значений — от 0 до 2π), а вторую — w, с диапазоном значений от -1 на одном краю ленты (по ширине) до 1 на другом:

```
In[14]: theta = np.linspace(0, 2 * np.pi, 30)
        w = np.linspace(-0.25, 0.25, 8)
        w, theta = np.meshgrid(w, theta)
```

Теперь нам нужно на основе этой параметризации вычислить координаты *(x, y, z)* ленты.

Размышляя, можно понять, что в данном случае происходят два вращательных движения: одно — изменение расположения кольца относительно его центра (координата, которую мы назвали θ), а второе — скручивание полоски относительно ее оси координат (назовем эту координату φ). Чтобы получилась лента Мебиуса, полоска должна выполнить половину скручивания за время полного сворачивания в кольцо, то есть $\Delta\varphi = \Delta\theta / 2$.

```
In[15]: phi = 0.5 * theta
```

Теперь вспомним тригонометрию, чтобы выполнить трехмерное наложение. Определим переменную r — расстояние каждой точки от центра и воспользуемся ею для нахождения внутренних координат *(x, y, z)*:

```
In[16]: # радиус в плоскости X-Y
        r = 1 + w * np.cos(phi)

        x = np.ravel(r * np.cos(theta))
        y = np.ravel(r * np.sin(theta))
        z = np.ravel(w * np.sin(phi))
```

Для построения графика этого объекта нужно убедиться, что триангуляция выполнена правильно. Лучший способ сделать это — описать триангуляцию *в координатах базовой параметризации*, после чего позволить библиотеке Matplotlib выполнить проекцию полученной триангуляции в трехмерное пространство ленты Мебиуса. Это можно сделать следующим образом (рис. 4.101):

```
In[17]: # Выполняем триангуляцию в координатах базовой параметризации
        from matplotlib.tri import Triangulation
        tri = Triangulation(np.ravel(w), np.ravel(theta))

        ax = plt.axes(projection='3d')
        ax.plot_trisurf(x, y, z, triangles=tri.triangles,
                        cmap='viridis', linewidths=0.2);

        ax.set_xlim(-1, 1); ax.set_ylim(-1, 1); ax.set_zlim(-1, 1);
```

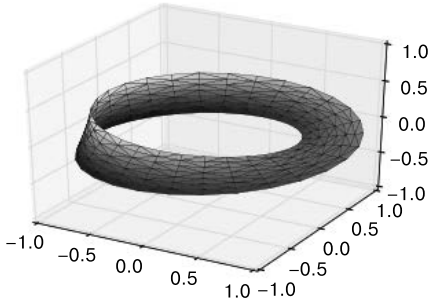

Рис. 4.101. Визуализация ленты Мебиуса

Сочетая все эти методы, можно создавать и отображать в Matplotlib широкий диапазон трехмерных объектов и пространственных моделей.

Отображение географических данных с помощью Basemap

Визуализация географических данных — распространенный вид визуализации в науке о данных. Для этой цели предназначена утилита библиотеки Matplotlib — набор программ Basemap, расположенных в пространстве имен `mpl_toolkits`. Правда, Basemap несколько неудобен в использовании, и даже простые изображения визуализируются дольше, чем хотелось бы. Для более ресурсоемких визуализаций карт, возможно, подойдут более современные решения, такие как библиотека Leaflet или картографический API Google. Тем не менее Basemap — инструмент, который не помешает иметь в запасе пользователям языка Python. В этом разделе мы продемонстрируем несколько примеров, возможных благодаря этому набору инструментов визуализаций карт.

Установка набора программ Basemap не представляет сложности. Если вы используете `conda`, для скачивания и установки пакета вам достаточно набрать команду:

```
$ conda install basemap[1]
```

Необходимо лишь добавить в наш обычный шаблон импортов еще одну строку:

---

[1] Обращаем ваше внимание на тот факт, что на момент выхода русского издания данной книги текущая официальная версия Basemap (1.0.7) требует для своей работы Python 2 или же Python 3 с младшей версией не выше 3.3. При необходимости вы можете установить на свою машину неофициальный выпуск пакета Basemap с помощью следующей команды: conda install -c conda-forge basemap=1.0.8.dev0.

```
In[1]: %matplotlib inline
       import numpy as np
       import matplotlib.pyplot as plt
       from mpl_toolkits.basemap import Basemap
```

Лишь несколько строк кода отделяет установку и импорт набора инструментов Basemap от построения географических графиков (построение графика на рис. 4.102 требует также наличия пакета `PIL` в Python 2 или пакета `pillow` в Python 3):

```
In[2]: plt.figure(figsize=(8, 8))
       m = Basemap(projection='ortho', resolution=None, lat_0=50, lon_0=-100)
       m.bluemarble(scale=0.5);
```

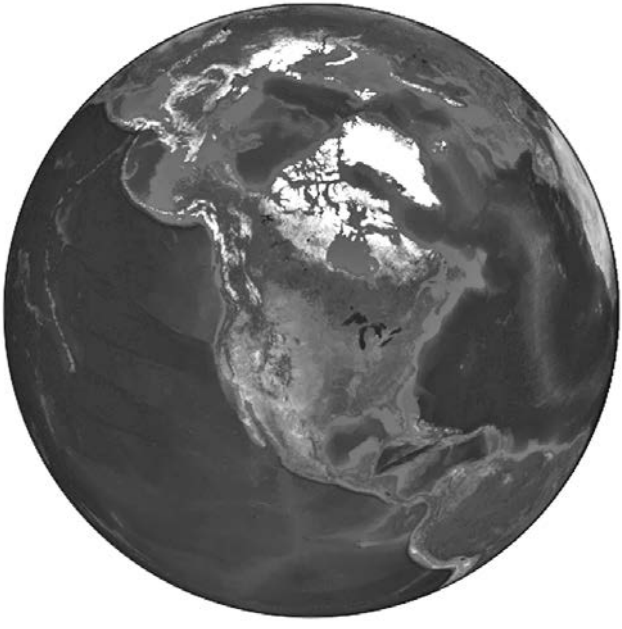

Рис. 4.102. Проекция Земли с помощью метода bluemarble

Смысл передаваемых Basemap атрибутов мы обсудим далее.

Удобнее всего то, что отображаемая на рисунке сфера — не просто изображение, это полнофункциональная система координат Matplotlib, понимающая сферические координаты и позволяющая легко дорисовывать данные на карту! Например, можно взять другую картографическую проекцию, посмотреть на крупный план Северной Америки и нарисовать местоположение Сиэтла. Мы воспользуемся изображением на основе набора данных etopo (отражающим топографические элементы как на поверхности земли, так и находящиеся под океаном) в качестве фона карты (рис. 4.103):

```
In[3]: fig = plt.figure(figsize=(8, 8))
       m = Basemap(projection='lcc', resolution=None,
                   width=8E6, height=8E6,
                   lat_0=45, lon_0=-100,)
       m.etopo(scale=0.5, alpha=0.5)

       # Проецируем кортеж (долгота, широта) на координаты (x, y)
       # для построения графика
       x, y = m(-122.3, 47.6)
       plt.plot(x, y, 'ok', markersize=5)
       plt.text(x, y, ' Seattle', fontsize=12);
```

Рис. 4.103. Наносим на карту данные и метки

Приведенный пример позволил нам взглянуть на то, какие географические визуализации возможны с помощью всего нескольких строк кода на языке Python. Теперь мы обсудим возможности набора инструментов Basemap более детально и рассмотрим несколько примеров визуализации картографических данных. Пользуясь этими примерами как готовыми блоками, вы сможете создавать практически любые картографические визуализации.

Картографические проекции

Первое, с чем нужно определиться при использовании карт, — какую проекцию использовать. Вероятно, вы знаете, что сферическую карту, например карту Земли, невозможно отобразить на плоскости без некоторого ее искажения или нарушения

связности. На протяжении истории человечества было разработано много таких проекций, так что у вас есть из чего выбирать! В зависимости от предполагаемого применения конкретной картографической проекции бывает удобно сохранить определенные свойства карты (например, пространственные направления, площадь, расстояние, форму и т. д.).

Пакет Basemap реализует несколько десятков таких проекций, на которые можно сослаться с помощью короткого кода формата. Мы продемонстрируем наиболее часто используемые среди них.

Начнем с описания удобных утилит, предназначенных для отрисовки нашей карты мира с линиями долготы и широты:

```
In[4]: from itertools import chain
       def draw_map(m, scale=0.2):
           # Отрисовываем изображение с оттененным рельефом
           m.shadedrelief(scale=scale)

           # Значения широты и долготы возвращаются в виде словаря
           lats = m.drawparallels(np.linspace(-90, 90, 13))
           lons = m.drawmeridians(np.linspace(-180, 180, 13))

           # Ключи, содержащие экземпляры класса plt.Line2D
           lat_lines = chain(*(tup[1][0] for tup in lats.items()))
           lon_lines = chain(*(tup[1][0] for tup in lons.items()))
           all_lines = chain(lat_lines, lon_lines)

           # Выполняем цикл по этим линиям, устанавливая нужный стиль
           for line in all_lines:
               line.set(linestyle='-', alpha=0.3, color='w')
```

Цилиндрические проекции

Простейшие из картографических проекций — цилиндрические, в которых линии одинаковой широты и долготы проецируются на горизонтальные и вертикальные линии соответственно. Такой тип проекции хорошо подходит для отображения областей у экватора, но приводит к сильной дисторсии возле полюсов. В различных цилиндрических проекциях расстояния между линиями широты различаются, что приводит к различной степени сохранения пропорций и дисторсии в районе полюсов. На рис. 4.104 показан пример *равнопромежуточной цилиндрической проекции* (equidistant cylindrical projection), характеризующейся выбором такого масштабирования широты, при котором расстояния вдоль меридианов остаются неизменными. Среди других цилиндрических проекций — проекция Меркатора (`projection='merc'`) и равновеликая цилиндрическая проекция (equal-area cylindrical projection).

```
In[5]: fig = plt.figure(figsize=(8, 6), edgecolor='w')
       m = Basemap(projection='cyl', resolution=None,
                   llcrnrlat=-90, urcrnrlat=90,
                   llcrnrlon=-180, urcrnrlon=180, )
       draw_map(m)
```

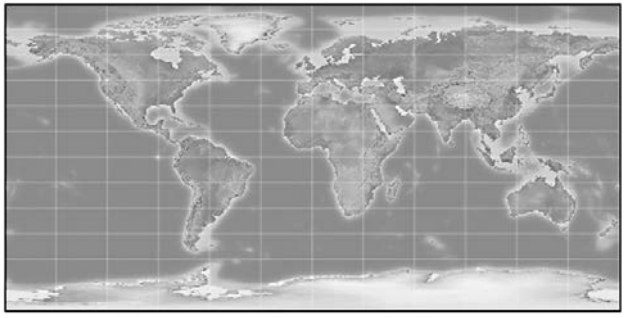

Рис. 4.104. Равновеликая цилиндрическая проекция

Передаваемые в Basemap дополнительные аргументы для этого представления задают широту (`lat`) и долготу (`lon`) нижнего левого (`llcrnr`) и верхнего правого (`urcrnr`) углов карты в градусах.

Псевдоцилиндрические проекции

В псевдоцилиндрических проекциях отсутствует требование вертикальности меридианов (линии одинаковой долготы), это позволяет улучшить показатели возле полюсов проекции. Проекция Мольвейде (`projection='moll'`) — распространенный пример псевдоцилиндрической проекции, в которой все меридианы представляют собой эллиптические дуги (рис. 4.105). Она разработана с целью сохранения пропорций на всей площади карты: хотя имеются некоторые дисторсии около полюсов, мелкие участки отображаются правдоподобно. В числе других псевдоцилиндрических проекций — синусоидальная проекция (`projection='sinu'`) и проекция Робинсона (`projection='robin'`).

```
In[6]: fig = plt.figure(figsize=(8, 6), edgecolor='w')
       m = Basemap(projection='moll', resolution=None,
                   lat_0=0, lon_0=0)
       draw_map(m)
```

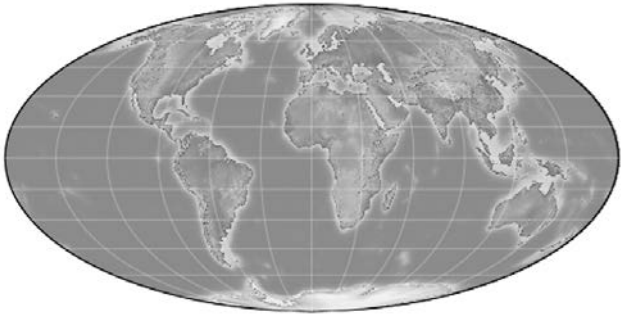

Рис. 4.105. Проекция Мольвейде

Передаваемые в Basemap дополнительные аргументы относятся к широте (`lat_0`) и долготе (`lon_0`) центра карты.

Перспективные проекции

Перспективные проекции создаются путем выбора конкретной главной точки, аналогично фотографированию Земли из конкретной точки пространства (в некоторых проекциях эта точка формально находится внутри Земли!). Распространенный пример — ортографическая проекция (`projection='ortho'`), показывающая одну сторону земного шара так, как его бы видел наблюдатель с очень большого расстояния. Таким образом, при ней можно видеть одновременно только половину земного шара. Среди других перспективных проекций — гномоническая (`projection='gnom'`) и стереографическая (`projection='stere'`) проекции. Они лучше всего подходят для отображения небольших участков карты.

Вот пример ортографической проекции (рис. 4.106):

```
In[7]: fig = plt.figure(figsize=(8, 8))
       m = Basemap(projection='ortho', resolution=None,
                   lat_0=50, lon_0=0)
       draw_map(m);
```

Рис. 4.106. Ортографическая проекция

Конические проекции

При конических проекциях карта проецируется на конус, который затем разворачивается. Такой способ позволяет получить отличные локальные характеристики, но удаленные от точки фокуса конуса области могут оказаться сильно искаженными. Один из примеров таких проекций — равноугольная коническая проекция Ламберта (projection='lcc'), которую мы уже видели в карте Северной Америки. При ее использовании карта проецируется на конус, устроенный таким образом, чтобы сохранялись расстояния на двух стандартных параллелях (задаваемых в Basemap с помощью аргументов `lat_1` и `lat_2`), в то время как между ними масштаб был меньше реального, а за их пределами — больше реального. Другие удобные конические проекции — равнопромежуточная коническая проекция (`projection='eqdc'`) и равновеликая коническая проекция Альберса (`projection='aea'`) (рис. 4.107). Конические проекции, как и перспективные проекции, хорошо подходят для отражения маленьких и средних кусков земного шара.

```
In[8]: fig = plt.figure(figsize=(8, 8))
       m = Basemap(projection='lcc', resolution=None,
                   lon_0=0, lat_0=50, lat_1=45, lat_2=55,
                   width=1.6E7, height=1.2E7)
       draw_map(m)
```

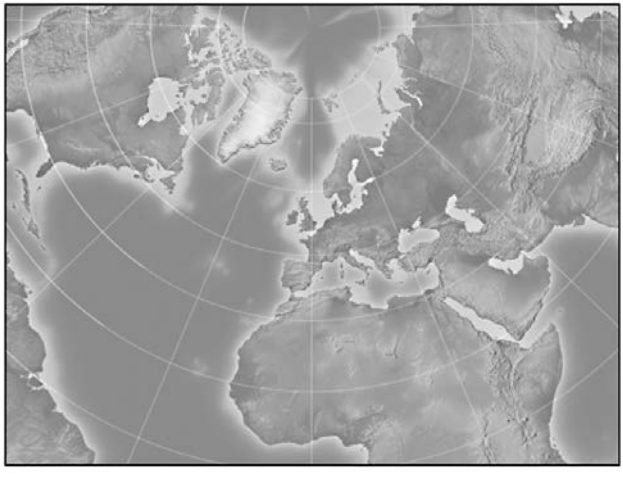

Рис. 4.107. Равновеликая коническая проекция Альберса

Другие проекции

Если вы собираетесь часто иметь дело с картографическими визуализациями, рекомендую почитать и о других проекциях, их свойствах, преимуществах и недостатках. Скорее всего, они имеются в пакете Basemap (http://matplotlib.org/basemap/

users/mapsetup.html). Окунувшись в этот вопрос, вы обнаружите поразительную субкультуру фанатов геовизуализаций, яростно отстаивающих преимущество их излюбленной проекции для любого приложения!

Отрисовка фона карты

Ранее мы рассмотрели методы `bluemarble()` и `shadedrelief()`, предназначенные для проекций изображений всего земного шара на карту, а также методы `drawparallels()` и `drawmeridians()`для рисования линий с постоянной широтой или долготой. Пакет Basemap содержит множество удобных функций для рисования границ физических объектов, например континентов, океанов, озер и рек, а также политических границ — границ стран или штатов/округов США. Далее приведены некоторые из имеющихся функций рисования, возможно, вы захотите изучить их подробнее с помощью справочных средств оболочки IPython.

- Физические границы и водоемы:
  - `drawcoastlines()` — рисует континентальные береговые линии;
  - `drawlsmask()` — рисует маску «земля/море» с целью проекции изображений на то или другое;
  - `drawmapboundary()` — рисует границы на карте, включая заливку цветом океанов;
  - `drawrivers()` — рисует реки на карте;
  - `fillcontinents()` — заливает пространство континентов заданным цветом; в качестве дополнительной настройки может залить озера другим цветом.
- Политические границы:
  - `drawcountries()` — рисует границы стран;
  - `drawstates()` — рисует границы штатов США;
  - `drawcounties()` — рисует границы округов США.
- Свойства карты:
  - `drawgreatcircle()` — рисует большой круг между двумя точками;
  - `drawparallels()` — рисует линии с постоянной широтой (меридианы);
  - `drawmeridians()` — рисует линии с постоянной долготой (параллели);
  - `drawmapscale()` — рисует на карте линейную шкалу масштаба.
- Изображения всего земного шара:
  - `bluemarble()` — проецирует сделанную NASA фотографию «голубого шарика» на карту;
  - `shadedrelief()` — проецирует на карту изображение с оттененным рельефом;
  - `etopo()` — рисует на карте изображение рельефа на основе набора данных etopo;
  - `warpimage()` — проецирует на карту пользовательское изображение.

Для объектов с границами необходимо при создании изображения Basemap задать желаемое разрешение. Аргумент `resolution` класса `Basemap` задает уровень детализации границ[1] (`'c'` (от англ. crude) — грубая детализация, `'l'` (от англ. low) — низкая детализация, `'i'` (от англ. intermediate) — средняя детализация, `'h'` (от англ. high) — высокая детализация, `'f'` (от англ. full) — полная детализация, или `None` — если границы не используются). Выбор значения этого параметра очень важен. Например, отрисовка границ с высоким разрешением на карте земного шара может происходить *очень* медленно.

Вот пример отрисовки границ «земля/море» и влияние на нее параметра разрешения. Мы создадим карту шотландского острова Скай как с низким, так и с высоким разрешением. Остров расположен в точке с координатами 57,3 °N, 6,2 °W, карта размером 90 000 × 120 000 км прекрасно его демонстрирует (рис. 4.108):

Рис. 4.108. Границы на карте при низком и высоком разрешении

```
In[9]: fig, ax = plt.subplots(1, 2, figsize=(12, 8))

       for i, res in enumerate(['l', 'h']):
           m = Basemap(projection='gnom', lat_0=57.3, lon_0=-6.2,
                       width=90000, height=120000, resolution=res, ax=ax[i])
           m.fillcontinents(color="#FFDDCC", lake_color='#DDEEFF')
           m.drawmapboundary(fill_color="#DDEEFF")
```

[1] Обратите внимание, что по умолчанию устанавливаются (см. предыдущее примечание) наборы данных только для грубого и низкого уровней детализации. Для установки наборов данных для высокого разрешения необходимо выполнить следующую команду: conda install -c conda-forge basemap-data-hires

```
    m.drawcoastlines()
    ax[i].set_title("resolution='{0}'".format(res));
```

Обратите внимание, что при низком разрешении береговые линии на этом уровне масштабирования отображаются некорректно, а при высоком — достаточно хорошо. Однако низкое разрешение отлично подходит для глобального представления и работает *гораздо* быстрее, чем загрузка данных по границам в высоком разрешении для всего земного шара. Может потребоваться немного поэкспериментировать с разрешением для конкретного представления, чтобы найти нужное, лучше всего начать с быстро отрисовываемого графика с низким разрешением и наращивать разрешение по мере необходимости.

Нанесение данных на карты

Вероятно, самая полезная на практике возможность набора инструментов Basemap — умение наносить разнообразные данные поверх фонового изображения карты. Для построения простых диаграмм и текста на картах подойдет любая из функций `plt`. Чтобы нанести данные на карту с помощью `plt`, можно воспользоваться для проекции координат широты и долготы на координаты (X, Y) экземпляром класса `Basemap`, как мы уже делали ранее в примере с Сиэтлом.

Помимо этого, среди методов экземпляра Basemap имеется множество функций, специально предназначенных для работы с картами. Они работают очень схоже со своими аналогами из библиотеки Matplotlib, но принимают дополнительный булев аргумент `latlon`, позволяющий (при равном `True` значении) передавать им исходные значения широты и долготы, а не их проекции на координаты (X, Y) .

Вот некоторые из методов, специально предназначенных для работы с картами:

- `contour()/contourf()` — рисует контурные линии или заполненные контуры;
- `imshow()` — отображает изображение;
- `pcolor() / pcolormesh()` — рисует псевдоцветной график для нерегулярных и регулярных сеток;
- `plot()` — рисует линии и/или маркеры;
- `scatter()` — рисует точки с маркерами;
- `quiver()` — рисует вектора;
- `barbs()` — рисует стрелки ветра;
- `drawgreatcircle()` — рисует большой круг[1].

Мы рассмотрим примеры некоторых из этих функций далее. Дальнейшую информацию о них, включая примеры графиков, можно найти в онлайн-документации Basemap.

---

[1] См. https://ru.wikipedia.org/wiki/Большой_круг.

Пример: города Калифорнии

В разделе «Пользовательские настройки легенд на графиках» данной главы было продемонстрировано использование размера и цвета точек на диаграмме рассеяния при передаче информации о расположении, размере и населении городов штата Калифорния. Здесь же мы воссоздадим этот график, но с использованием Basemap для включения данных в соответствующий картографический контекст.

Начнем, как и раньше, с загрузки данных:

```
In[10]: import pandas as pd
        cities = pd.read_csv('data/california_cities.csv')

        # Извлекаем интересующие нас данные
        lat = cities['latd'].values
        lon = cities['longd'].values
        population = cities['population_total'].values
        area = cities['area_total_km2'].values
```

Настраиваем проекцию карты, наносим данные, после чего создаем шкалу цветов и легенду (рис. 4.109):

```
In[11]: # 1. Рисуем фон карты
        fig = plt.figure(figsize=(8, 8))
        m = Basemap(projection='lcc', resolution='h',
                    lat_0=37.5, lon_0=-119,
                    width=1E6, height=1.2E6)
        m.shadedrelief()
        m.drawcoastlines(color='gray')
        m.drawcountries(color='gray')
        m.drawstates(color='gray')

        # 2. Наносим данные по городам, отражая население разными цветами,
        # а площадь – разными размерами точек
        m.scatter(lon, lat, latlon=True,
                  c=np.log10(population), s=area,
                  cmap='Reds', alpha=0.5)

        # 3. Создаем шкалу цветов и легенду
        plt.colorbar(label=r'$\log_{10}({\rm population})$')
        plt.clim(3, 7)

        # Делаем легенду с фиктивными точками
        for a in [100, 300, 500]:
            plt.scatter([], [], c='k', alpha=0.5, s=a,
                        label=str(a) + ' km$^2$')
        plt.legend(scatterpoints=1, frameon=False,
                   labelspacing=1, loc='lower left');
```

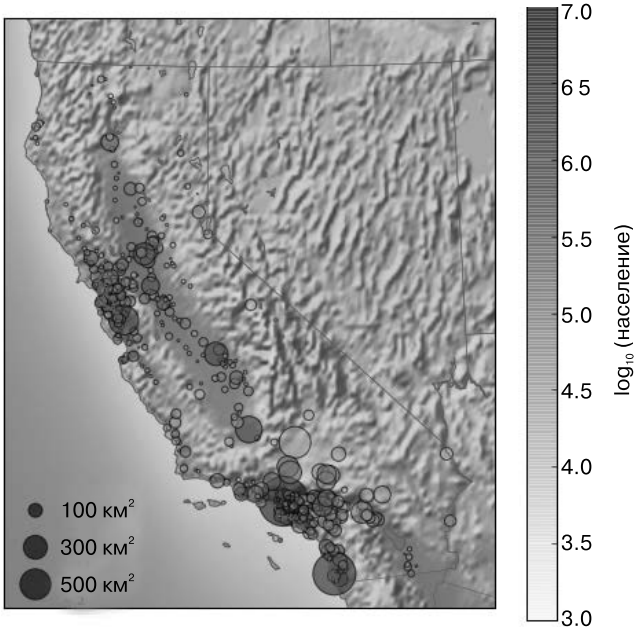

Рис. 4.109. Диаграмма рассеяния поверх карты в качестве фона

Этот график демонстрирует приблизительно, в каких местах Калифорнии живет большое количество людей: они сосредоточены на побережье в районе Лос-Анджелеса и Сан-Франциско, по бокам шоссе в Центральной долине, избегая практически полностью гористых районов на границах штата.

Пример: данные о температуре на поверхности Земли

В качестве примера визуализации непрерывных географических данных рассмотрим «полярный вихрь», охвативший западную половину США в январе 2014 года. Замечательный источник разнообразных метеорологических данных — подразделение NASA Институт изучения космоса имени Годдарда (https://data.giss.nasa.gov/). В этом случае мы воспользуемся температурными данными GIS 250, которые можно скачать с помощью командной оболочки (возможно, на работающих под управлением операционной системы Windows эти команды придется несколько изменить). Используемые здесь данные скачивались 12 июня 2016 года, и их размер тогда составлял примерно 9 Мбайт:

```
In[12]: # !curl -O https://data.giss.nasa.gov/pub/gistemp/gistemp250.nc.gz
        # !gunzip gistemp250.nc.gz
```

Данные находятся в формате NetCDF, который в языке Python можно прочитать с помощью библиотеки `netCDF4`. Установить эту библиотеку можно следующим образом:

```
$ conda install netcdf4
```

Прочитаем данные с помощью команд:

```
In[13]: from netCDF4 import Dataset
        data = Dataset('gistemp250.nc')
```

Файл содержит множество глобальных показаний температуры на различные даты, нам нужно выбрать индекс, соответствующий интересующей нас дате — 15 января 2014 года:

```
In[14]: from netCDF4 import date2index
        from datetime import datetime
        timeindex = date2index(datetime(2014, 1, 15),
                               data.variables['time'])
```

Теперь можно загрузить данные по широте и долготе, а также температурным аномалиям для этого значения индекса:

```
In[15]: lat = data.variables['lat'][:]
        lon = data.variables['lon'][:]
        lon, lat = np.meshgrid(lon, lat)
        temp_anomaly = data.variables['tempanomaly'][timeindex]
```

Воспользуемся методом `pcolormesh()` для отрисовки цветовой сетки наших данных. Нас интересует Северная Америка, и в качестве фона мы будем использовать карту с оттененным рельефом. Обратите внимание, что для этих данных мы специально выбрали дивергентную карту цветов, с нейтральным цветом для 0 и двумя контрастными цветами для отрицательных и положительных значений (рис. 4.110).

В справочных целях мы также нарисуем светлым цветом поверх цветов береговые линии:

```
In[16]: fig = plt.figure(figsize=(10, 8))
        m = Basemap(projection='lcc', resolution='c',
                    width=8E6, height=8E6,
                    lat_0=45, lon_0=-100,)
        m.shadedrelief(scale=0.5)
        m.pcolormesh(lon, lat, temp_anomaly,
                     latlon=True, cmap='RdBu_r')
        plt.clim(-8, 8)
        m.drawcoastlines(color='lightgray')
        plt.title('January 2014 Temperature Anomaly')
        plt.colorbar(label='temperature anomaly (°C)');
        # Температурные аномалии
```

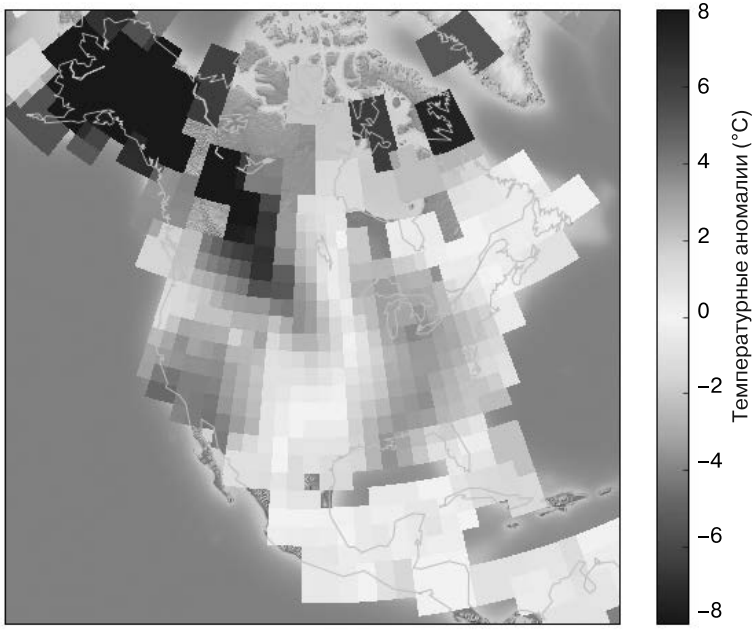

Рис. 4.110. Температурная аномалия в январе 2014 года

Диаграмма демонстрирует картину экстремальных температурных аномалий, имевших место на протяжении этого месяца. В восточной части США температура была гораздо ниже обычного, в то время как на западе и Аляске было намного теплее. В местах, где температура не регистрировалась, мы видим фон карты.

Визуализация с помощью библиотеки Seaborn

Библиотека Matplotlib зарекомендовала себя как невероятно удобный и популярный инструмент визуализации, но даже заядлые ее пользователи признают, что она зачастую оставляет желать лучшего. Существует несколько часто возникающих жалоб на Matplotlib.

- До версии 2.0 параметры по умолчанию библиотеки Matplotlib были далеко не идеальными. Они вели свое происхождение от MATLAB-версии примерно 1999 года, и это часто ощущалось.
- API библиотеки Matplotlib — относительно низкоуровневый. С его помощью можно создавать сложные статистические визуализации, но это требует немало шаблонного кода.
- Matplotlib была выпущена на десятилетие раньше, чем библиотека Pandas, и потому не ориентирована на работу с объектами `DataFrame` библиотеки Pandas.

Для визуализации данных из объекта `DataFrame` библиотеки Pandas приходится извлекать все объекты `Series` и зачастую конкатенировать их в нужный формат. Хорошо было бы иметь библиотеку для построения графиков, в которой присутствовали бы возможности по интеллектуальному использованию меток `DataFrame` на графиках.

Библиотека Seaborn — решение этих проблем. Seaborn предоставляет API поверх библиотеки Matplotlib, обеспечивающий разумные варианты стилей графиков и цветов по умолчанию, определяющий простые высокоуровневые функции для часто встречающихся типов графиков и хорошо интегрирующийся с функциональностью, предоставляемой объектами `DataFrame` библиотеки Pandas.

Справедливости ради упомяну, что команда разработчиков библиотеки Matplotlib тоже пытается решить эти проблемы: они добавили утилиты `plt.style` (которые мы обсуждали в разделе «Пользовательские настройки Matplotlib: конфигурации и таблицы стилей» этой главы) и принимают меры к более органичной обработке данных Pandas. Версия 2.0 библиотеки Matplotlib включает новую таблицу стилей по умолчанию, исправляющую ситуацию. Но по вышеизложенным причинам библиотека Seaborn остается исключительно удобным дополнением.

Seaborn по сравнению с Matplotlib

Вот пример простого графика случайных блужданий с использованием стиля `classic` для форматирования и цветов графика. Начнем с обычных импортов:

```
In[1]: import matplotlib.pyplot as plt
       plt.style.use('classic')
       %matplotlib inline
       import numpy as np
       import pandas as pd
```

Создаем данные случайных блужданий:

```
In[2]: # Создаем данные
       rng = np.random.RandomState(0)
       x = np.linspace(0, 10, 500)
       y = np.cumsum(rng.randn(500, 6), 0)
```

Рисуем простой график (рис. 4.111):

```
In[3]: # Рисуем график, используя параметры Matplotlib по умолчанию
       plt.plot(x, y)
       plt.legend('ABCDEF', ncol=2, loc='upper left');
```

Хотя результат содержит всю информацию, которую нам требуется донести до читателя, это происходит не слишком приятным глазу образом, и даже выглядит слегка старомодным в свете современных визуализаций данных.

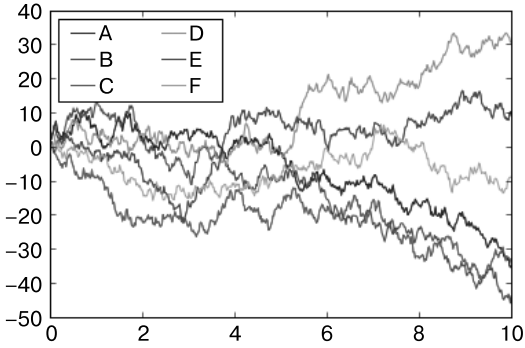

Рис. 4.111. Данные в стиле библиотеки Matplotlib по умолчанию

Теперь посмотрим, как можно сделать это с помощью Seaborn. Помимо множества собственных высокоуровневых функций построения графиков библиотеки Seaborn, она может также перекрывать параметры по умолчанию библиотеки Matplotlib, благодаря чему применение даже более простых сценариев Matplotlib приводит к намного лучшему результату. Задать стиль можно с помощью метода `set()` библиотеки Seaborn. По принятым соглашениям Seaborn импортируется под именем `sns`:

```
In[4]: import seaborn as sns
       sns.set()
```

Теперь выполним еще раз те же две строки кода, что и раньше (рис. 4.112):

```
In[5]: # Тот же самый код для построения графика, что и выше!
       plt.plot(x, y)
       plt.legend('ABCDEF', ncol=2, loc='upper left');
```

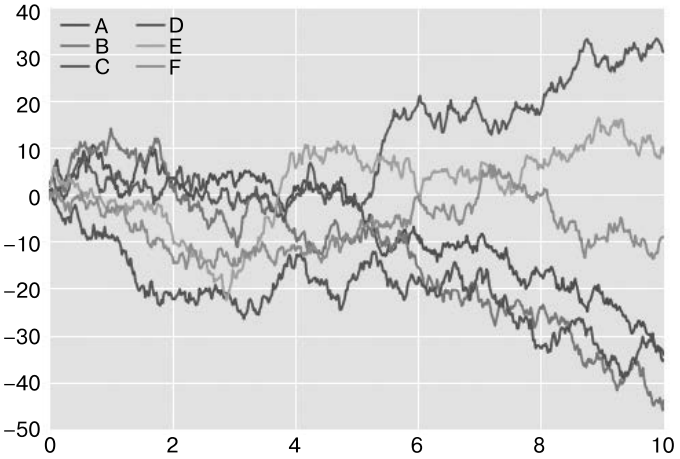

Рис. 4.112. Данные в используемом по умолчанию стиле библиотеки Seaborn

О, намного лучше!

Анализируем графики библиотеки Seaborn

Основная идея библиотеки Seaborn — предоставление высокоуровневых команд для создания множества различных типов графиков, удобных для исследования статистических данных и даже подгонки статистических моделей.

Рассмотрим некоторые из имеющихся в Seaяborn наборов данных и типов графиков. Обратите внимание, что все изложенное далее *можно* выполнить и с помощью обычных команд библиотеки Matplotlib, но API Seaborn намного более удобен.

Гистограммы, KDE и плотности

Зачастую все, что нужно сделать при визуализации статистических данных, — это построить гистограмму и график совместного распределения переменных. Мы уже видели, что в библиотеке Matplotlib сделать это относительно несложно (рис. 4.113):

```
In[6]: data = np.random.multivariate_normal([0, 0], [[5, 2], [2, 2]],
                                            size=2000)
       data = pd.DataFrame(data, columns=['x', 'y'])

       for col in 'xy':
           plt.hist(data[col], normed=True, alpha=0.5)
```

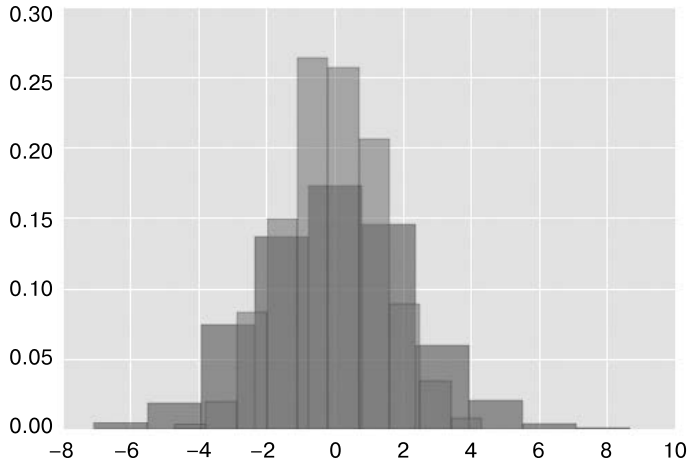

Рис. 4.113. Гистограммы для визуализации распределений

Вместо гистограммы можно получить гладкую оценку распределения путем ядерной оценки плотности распределения, которую Seaborn выполняет с помощью функции `sns.kdeplot` (рис. 4.114):

```
In[7]: for col in 'xy':
           sns.kdeplot(data[col], shade=True)
```

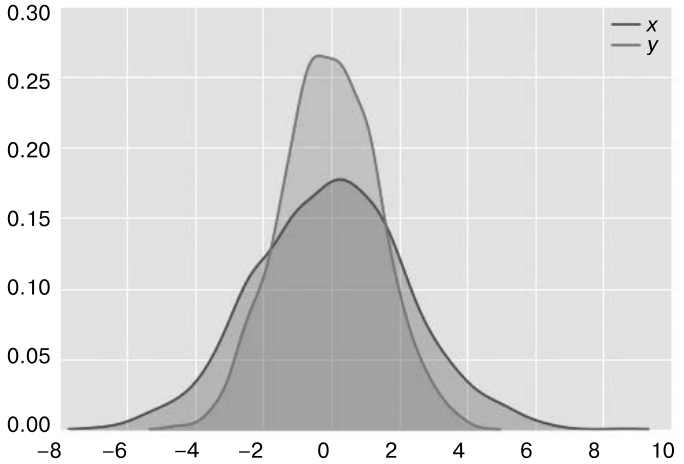

Рис. 4.114. Использование ядерной оценки плотности для визуализации распределений

С помощью функции `distplot` можно сочетать гистограммы и KDE (рис. 4.115):

```
In[8]: sns.distplot(data['x'])
       sns.distplot(data['y']);
```

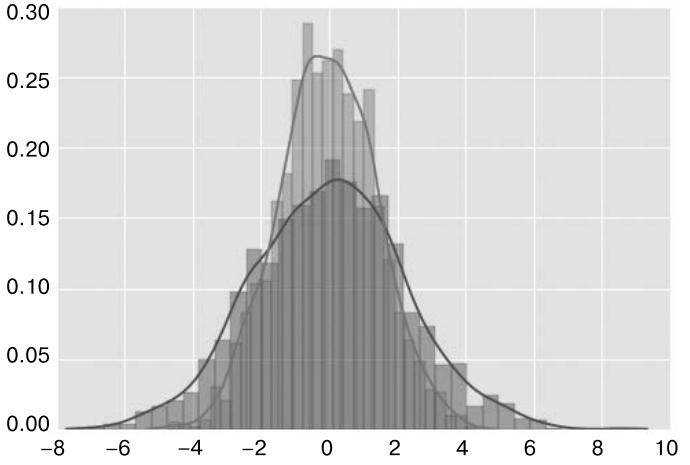

Рис. 4.115. Совместный график ядерной оценки плотности и гистограммы

Если передать функции `kdeplot` весь двумерный набор данных, можно получить двумерную визуализацию данных (рис. 4.116):

```
In[9]: sns.kdeplot(data);
```

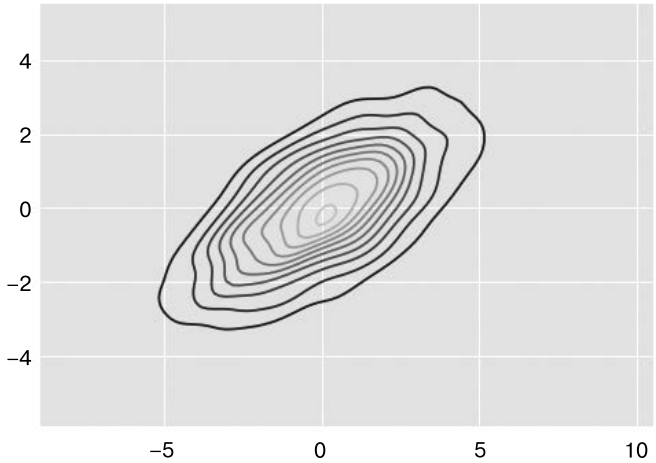

Рис. 4.116. Двумерный график ядерной оценки плотности

Посмотреть на совместное распределение и частные распределения можно, воспользовавшись функцией `sns.jointplot`. Для этого графика мы зададим стиль с белым фоном (рис. 4.117):

```
In[10]: with sns.axes_style('white'):
            sns.jointplot("x", "y", data, kind='kde');
```

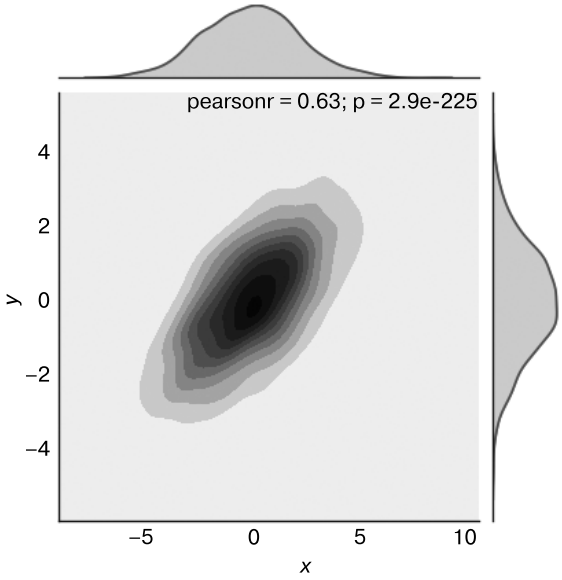

Рис. 4.117. График совместного распределения с двумерным графиком ядерной оценки плотности

Функции `jointplot` можно передавать и другие параметры, например можно воспользоваться гистограммой на базе шестиугольников (рис. 4.118):

```
In[11]: with sns.axes_style('white'):
            sns.jointplot("x", "y", data, kind='hex')
```

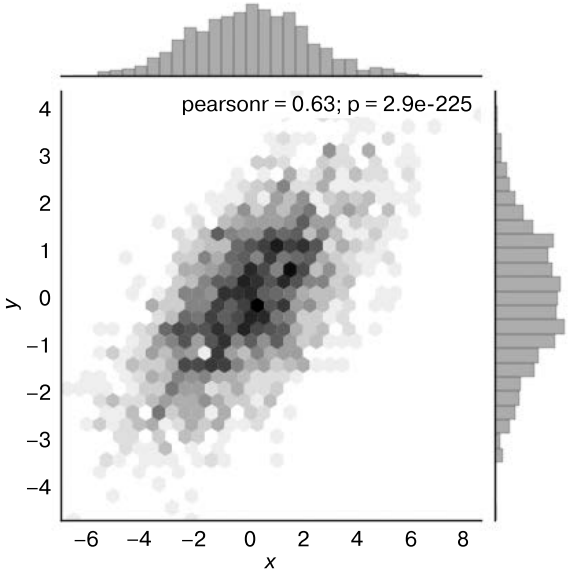

Рис. 4.118. График совместного распределения с гексагональным разбиением по интервалам

При обобщении графиков совместных распределений на наборы данных более высоких размерностей мы постепенно приходим к *графикам пар* (pair plots). Они очень удобны для изучения зависимостей между многомерными данными, когда необходимо построить график всех пар значений.

Мы продемонстрируем это на уже знакомом вам наборе данных Iris[1], содержащем измерения лепестков и чашелистиков трех видов ирисов:

```
In[12]: iris = sns.load_dataset("iris")
        iris.head()

Out[12]:    sepal_length  sepal_width  petal_length  petal_width species
         0           5.1          3.5           1.4          0.2  setosa
         1           4.9          3.0           1.4          0.2  setosa
```

[1] Если вы работаете в операционной системе Windows и ваша версия Python — 3.6, при выполнении этих команд вы можете столкнуться с известной ошибкой, связанной с изменением кодировки имен файлов по умолчанию в Python 3.6. Простейшим решением проблемы будет изменение кодировки на использовавшуюся в предыдущих версиях:
import sys
sys._enablelegacywindowsfsencoding()

```
2          4.7          3.2          1.3          0.2  setosa
3          4.6          3.1          1.5          0.2  setosa
4          5.0          3.6          1.4          0.2  setosa
```

Визуализация многомерных зависимостей между выборками сводится к вызову функции `sns.pairplot` (рис. 4.119):

```
In[13]: sns.pairplot(iris, hue='species', size=2.5);
```

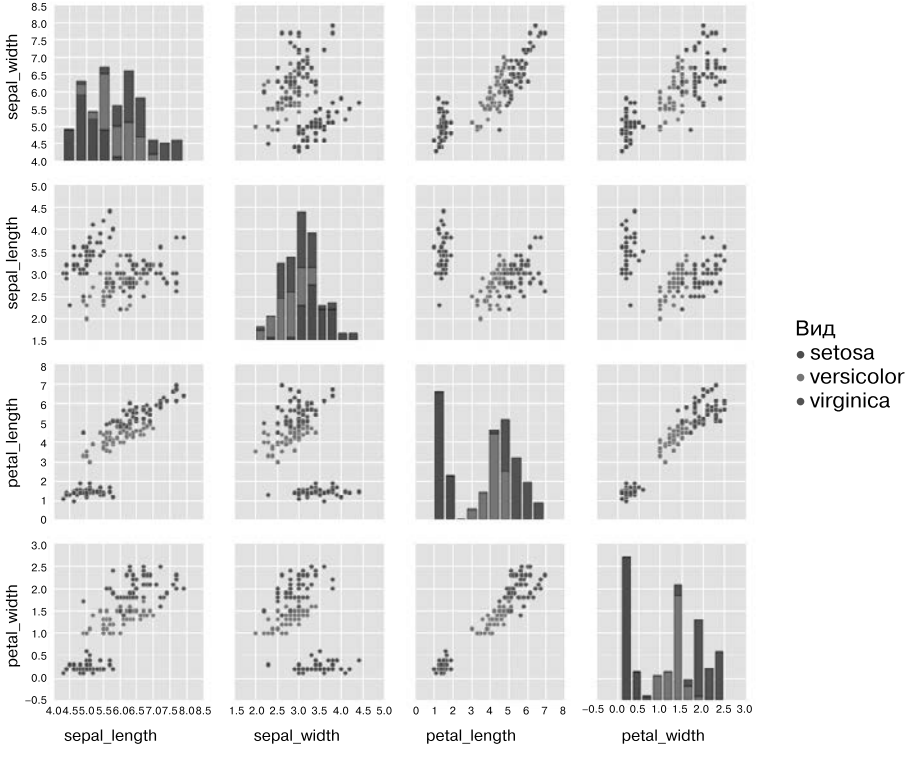

Рис. 4.119. График пар, демонстрирующий зависимости между четырьмя переменными

Фасетные гистограммы

Иногда оптимальный способ представления данных — гистограммы подмножеств. Функция `FacetGrid` библиотеки Seaborn делает эту задачу элементарной. Рассмотрим данные, отображающие суммы, которые персонал ресторана получает в качестве чаевых, в зависимости от данных различных индикаторов (рис. 4.120):

```
In[14]: tips = sns.load_dataset('tips')
        tips.head()
Out[14]:    total_bill   tip     sex smoker  day    time  size
         0       16.99  1.01  Female     No  Sun  Dinner     2
```

```
         1       10.34  1.66    Male     No  Sun  Dinner     3
         2       21.01  3.50    Male     No  Sun  Dinner     3
         3       23.68  3.31    Male     No  Sun  Dinner     2
         4       24.59  3.61  Female     No  Sun  Dinner     4
In[15]: tips['tip_pct'] = 100 * tips['tip'] / tips['total_bill']

        grid = sns.FacetGrid(tips, row="sex", col="time", margin_titles=True)
        grid.map(plt.hist, "tip_pct", bins=np.linspace(0, 40, 15));
```

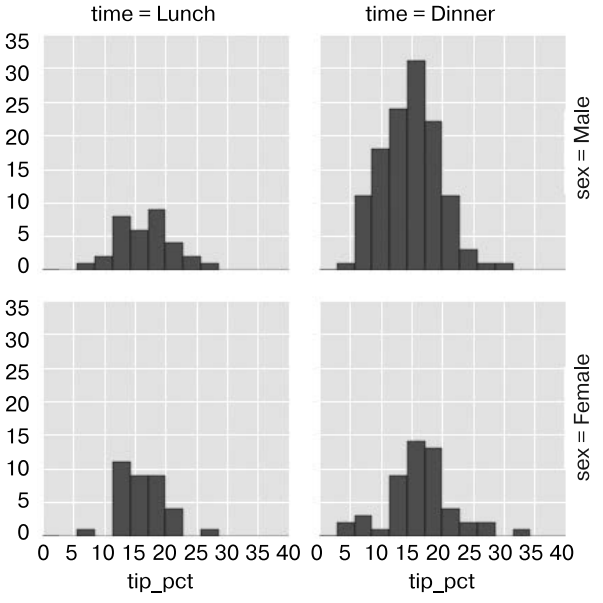

Рис. 4.120. Пример фасетной гистограммы

Графики факторов

Графики факторов тоже подходят для подобных визуализаций. Они позволяют просматривать распределение параметра по интервалам, задаваемым посредством любого другого параметра (рис. 4.121):

```
In[16]: with sns.axes_style(style='ticks'):
            g = sns.factorplot("day", "total_bill", "sex", data=tips,
                               kind="box")
            g.set_axis_labels("Day", "Total Bill"); # День; Итого
```

Совместные распределения

Аналогично графикам пар, которые мы рассматривали ранее, мы можем воспользоваться функцией `sns.jointplot` для отображения совместного распределения между различными наборами данных, а также соответствующих частных распределений (рис. 4.122):

```
In[17]: with sns.axes_style('white'):
            sns.jointplot("total_bill", "tip", data=tips, kind='hex')
```

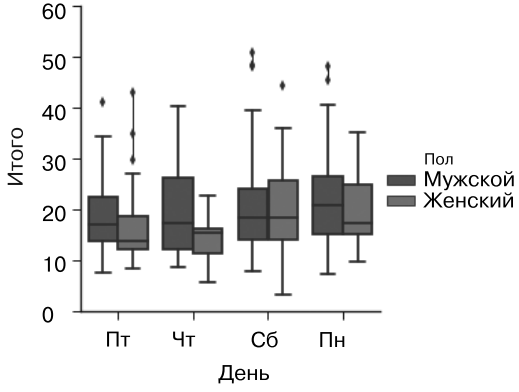

Рис. 4.121. Пример графика факторов со сравнением распределений при различных дискретных факторах

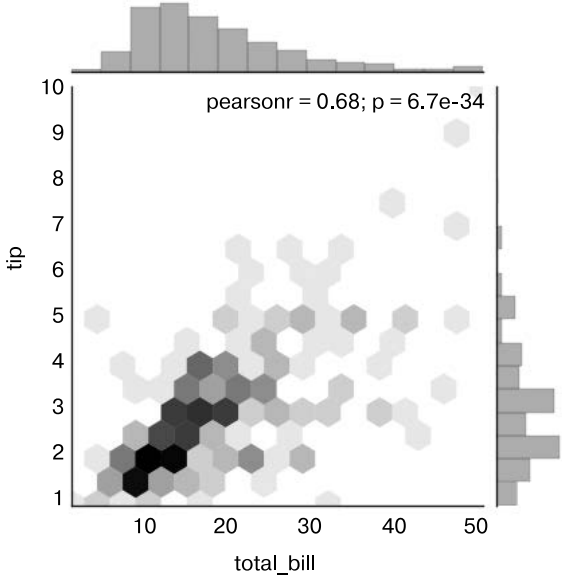

Рис. 4.122. График совместного распределения

График совместного распределения позволяет даже выполнять автоматическую ядерную оценку плотности распределения и регрессию (рис. 4.123):

```
In[18]: sns.jointplot("total_bill", "tip", data=tips, kind='reg');
```

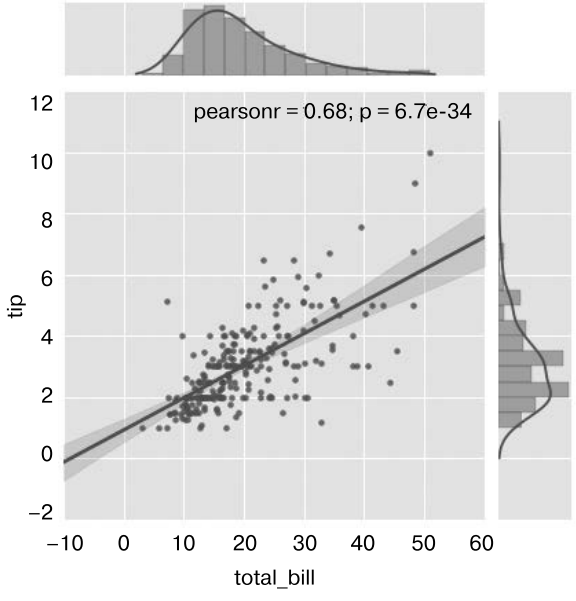

Рис. 4.123. График совместного распределения с подбором регрессии

Столбчатые диаграммы

Графики временных рядов можно строить с помощью функции `sns.factorplot`. В следующем примере, показанном на рис. 4.124, мы воспользуемся данными из набора Planets («Планеты»), которые мы уже видели в разделе «Агрегирование и группировка» главы 2:

```
In[19]: planets = sns.load_dataset('planets')
        planets.head()

Out[19]:            method  number  orbital_period   mass  distance  year
         0  Radial Velocity       1         269.300   7.10     77.40  2006
         1  Radial Velocity       1         874.774   2.21     56.95  2008
         2  Radial Velocity       1         763.000   2.60     19.84  2011
         3  Radial Velocity       1         326.030  19.40    110.62  2007
         4  Radial Velocity       1         516.220  10.50    119.47  2009

In[20]: with sns.axes_style('white'):
            g = sns.factorplot("year", data=planets, aspect=2,  # Год
                               kind="count", color='steelblue') # Количество
            g.set_xticklabels(step=5)
```

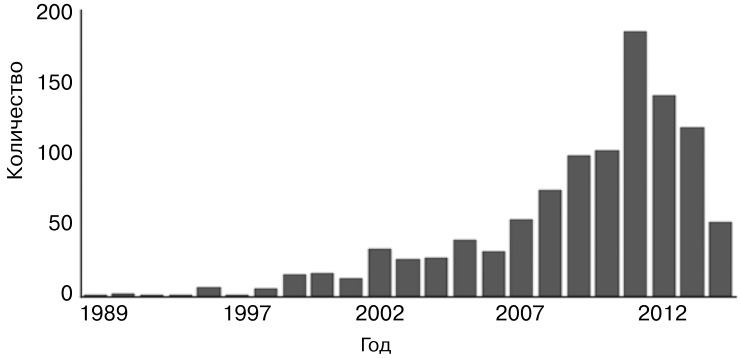

Рис. 4.124. Гистограмма как частный случай графика факторов

Мы можем узнать больше, если посмотрим на *метод*, с помощью которого была открыта каждая из этих планет, как показано на рис. 4.125:

```
In[21]: with sns.axes_style('white'):
            g = sns.factorplot("year", data=planets, aspect=4.0, kind='count',
                               hue='method', order=range(2001, 2015))
            g.set_ylabels('Number of Planets Discovered')
            # Количество обнаруженных планет
```

Дополнительную информацию о построении графиков с помощью библиотеки Seaborn можно найти в документации, справочном руководстве и галерее Seaborn.

Пример: время прохождения марафона

В этом разделе мы рассмотрим использование библиотеки Seaborn для визуализации и анализа данных по времени прохождения марафонской дистанции. Эти данные я собрал из различных интернет-источников, агрегировал, убрал все идентифицирующие данные и поместил на GitHub, откуда их можно скачать (если вас интересует использование языка Python для веб-скрапинга, рекомендую книгу *Web Scraping with Python*[1] (http://shop.oreilly.com/product/063 69200 34391.do) Райана Митчелла. Начнем со скачивания данных из Интернета и загрузки их в Pandas:

```
In[22]: # !curl -O https://raw.githubusercontent.com/jakevdp/marathon-data/
# master/marathon-data.csv

In[23]: data = pd.read_csv('marathon-data.csv')
        data.head()

Out[23]:    age gender     split     final
         0   33      M  01:05:38  02:08:51
         1   32      M  01:06:26  02:09:28
         2   31      M  01:06:49  02:10:42
         3   38      M  01:06:16  02:13:45
         4   31      M  01:06:32  02:13:59
```

[1] *Райан М.* Скрапинг сайтов с помощью Python. — М.: ДМК-Пресс, 2016.

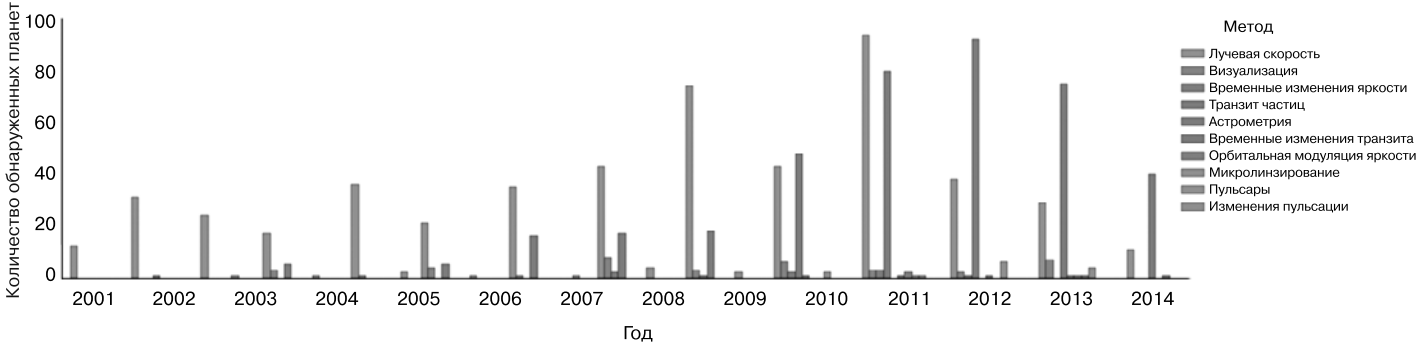

Рис. 4.125. Количесство открытых планет по типу и году открытия (см. полноцветный график в онлайн-режиме (https://github.com/jakevdp/PythonDataScienceHandbook))

По умолчанию библиотека Pandas загружает столбцы с временем как строки Python (тип `object`), убедиться в этом можно, посмотрев значение атрибута `dtypes` объекта `DataFrame`:

```
In[24]: data.dtypes

Out[24]: age        int64
         gender    object
         split     object
         final     object
         dtype: object
```

Исправим это, создав функцию для преобразования значений времени:

```
In[25]: def convert_time(s):
            h, m, s = map(int, s.split(':'))
            return pd.datetools.timedelta(hours=h, minutes=m, seconds=s)
        data = pd.read_csv('marathon-data.csv',
                           converters={'split':convert_time,
                          'final':convert_time})
        data.head()

Out[25]:    age gender    split    final
         0   33      M 01:05:3802:08:51
         1   32      M 01:06:2602:09:28
         2   31      M 01:06:4902:10:42
         3   38      M 01:06:1602:13:45
         4   31      M 01:06:3202:13:59

In[26]: data.dtypes

Out[26]: age                int64
         gender            object
         split     timedelta64[ns]
         final     timedelta64[ns]
         dtype: object
```

Выглядит намного лучше. Добавим для использования при построении графиков столбцы с временем в секундах[1]:

```
In[27]: data['split_sec'] = data['split'].astype(int) / 1E9
        data['final_sec'] = data['final'].astype(int) / 1E9
        data.head()

Out[27]:    age gender    split    final  split_sec  final_sec
         0   33      M 01:05:3802:08:51     3938.0     7731.0
         1   32      M 01:06:2602:09:28     3986.0     7768.0
         2   31      M 01:06:4902:10:42     4009.0     7842.0
         3   38      M 01:06:1602:13:45     3976.0     8025.0
         4   31      M 01:06:3202:13:59     3992.0     8039.0
```

[1] При работе в командной оболочке IPython в 64-битной операционной системе Windows может возникнуть ошибка преобразования типа. Один из путей решения этой проблемы — использовать в следующем коде тип np.int64 вместо int.

Чтобы понять, что представляют собой данные, можно нарисовать для них график `jointplot` (рис. 4.126):

```
In[28]: with sns.axes_style('white'):
            g = sns.jointplot("split_sec", "final_sec", data, kind='hex')
            g.ax_joint.plot(np.linspace(4000, 16000),
                            np.linspace(8000, 32000), ':k')
```

Пунктирная линия показывает, каким было бы для бегунов время прохождения всего марафона, если бы они бежали его с неизменной скоростью. Распределение лежит выше этой прямой, и это значит (как и можно было ожидать), что большинство людей снижает скорость по мере прохождения дистанции. Если вы участвовали в марафонах, то знаете, что в случае бегунов, поступающих наоборот — ускоряющихся во время второй части дистанции, — говорят об *обратном распределении сил*.

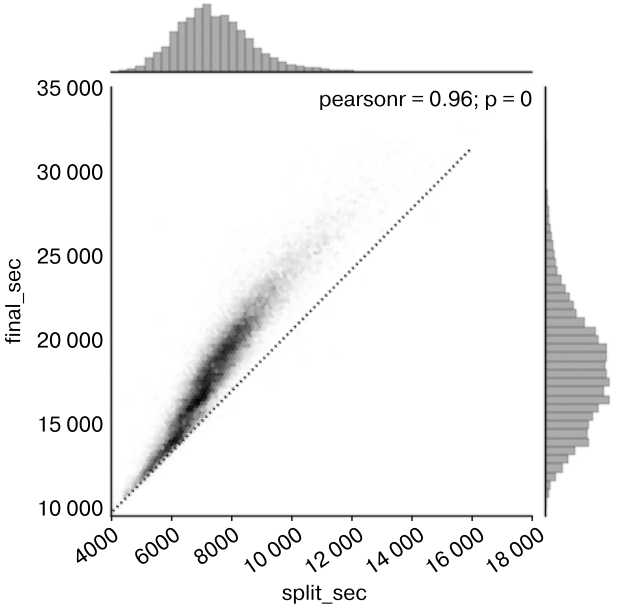

Рис. 4.126. Зависимости между временем прохождения первой половины марафона и всего марафона целиком

Создадим в данных еще один столбец, коэффициент распределения, показывающий, степень прямого и обратного распределения сил каждым бегуном:

```
In[29]: data['split_frac'] = 1 - 2 * data['split_sec'] / data['final_sec']
        data.head()
Out[29]:    age gender    split    final  split_sec  final_sec  split_frac
         0   33      M 01:05:3802:08:51     3938.0     7731.0   -0.018756
         1   32      M 01:06:2602:09:28     3986.0     7768.0   -0.026262
         2   31      M 01:06:4902:10:42     4009.0     7842.0   -0.022443
         3   38      M 01:06:1602:13:45     3976.0     8025.0    0.009097
         4   31      M 01:06:3202:13:59     3992.0     8039.0    0.006842
```

Если этот коэффициент меньше нуля, значит, соответствующий спортсмен распределяет свои силы в обратной пропорции на соответствующую долю. Построим график распределения этого коэффициента (рис. 4.127):

```
In[30]: sns.distplot(data['split_frac'], kde=False);
        plt.axvline(0, color="k", linestyle="--");

In[31]: sum(data.split_frac < 0)

Out[31]: 251
```

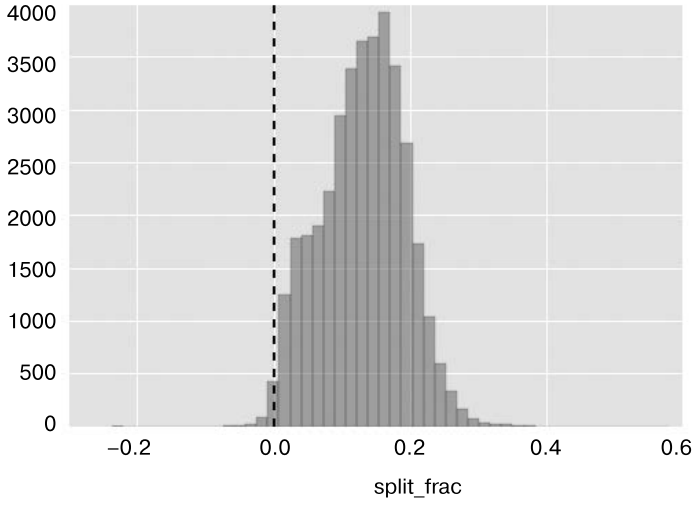

Рис. 4.127. Распределение коэффициентов для всех бегунов; 0.0 соответствует бегуну, пробежавшему первую и вторую половины марафона за одинаковое время

Из почти 40 000 участников только 250 человек распределяют свои силы на марафонской дистанции в обратной пропорции.

Выясним, существует ли какая-либо корреляция между коэффициентом распределения сил и другими переменными. Для построения графиков всех этих корреляций мы воспользуемся функцией `pairgrid` (рис. 4.128):

```
In[32]:
g = sns.PairGrid(data, vars=['age', 'split_sec', 'final_sec', 'split_frac'],
                 hue='gender', palette='RdBu_r')
g.map(plt.scatter, alpha=0.8)
g.add_legend();
```

Похоже, что коэффициент распределения сил никак не коррелирует с возрастом, но коррелирует с итоговым временем забега: более быстрые бегуны склонны распределять свои силы поровну. Как мы видим на этом графике, библиотека Seaborn — не панацея от «недугов» библиотеки Matplotlib, если речь идет о стилях графиков:

в частности, метки на оси *X* перекрываются. Однако, поскольку результат — простой график Matplotlib, можно воспользоваться методами из раздела «Пользовательские настройки делений на осях координат» данной главы для настройки подобных вещей.

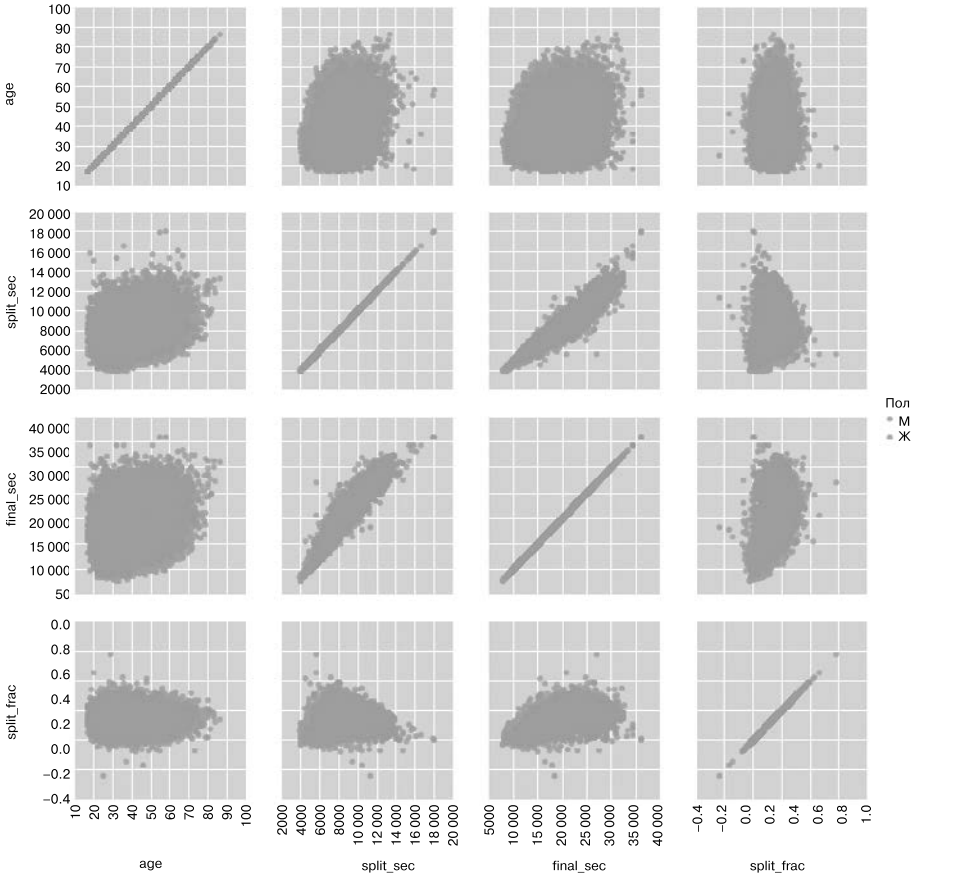

Рис. 4.128. Зависимости между величинами в «марафонском» наборе данных

Кроме того, представляет интерес различие между мужчинами и женщинами. Рассмотрим гистограмму коэффициентов распределения сил для этих двух групп (рис. 4.129):

```
In[33]: sns.kdeplot(data.split_frac[data.gender=='M'],
                    label='men', shade=True)
        sns.kdeplot(data.split_frac[data.gender=='W'],
                    label='women', shade=True)
        plt.xlabel('split_frac');
```

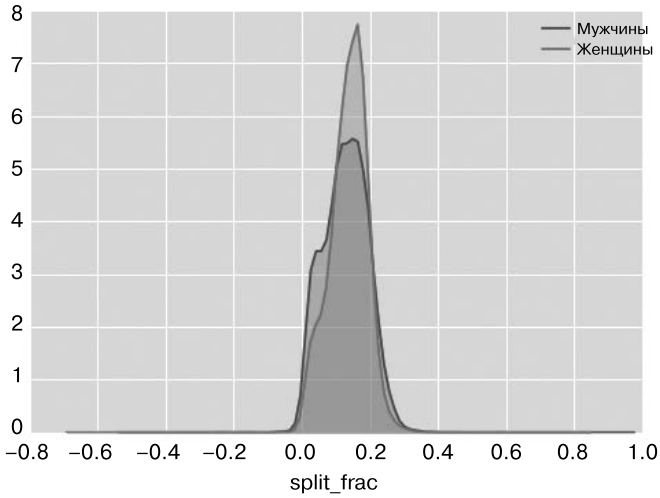

Рис. 4.129. Распределение коэффициентов для бегунов в зависимости от пола

Интересно здесь то, что мужчин, чьи силы распределены практически поровну, намного больше, чем женщин! Это выглядит практически как какое-то бимодальное распределение по мужчинам и женщинам. Удастся ли нам разобраться, в чем дело, взглянув на эти распределения как на функцию возраста?

Удобный способ сравнения распределений — использование так называемой скрипичной диаграммы (рис. 4.130):

```
In[34]:
sns.violinplot("gender", "split_frac", data=data,
               palette=["lightblue", "lightpink"]);
```

Есть и еще один способ сравнить распределения для мужчин и женщин.

Заглянем чуть глубже и сравним эти «скрипичные» диаграммы как функцию возраста. Начнем с создания в массиве нового столбца, отражающего возраст бегуна с точностью до десятилетия (рис. 4.131):

```
In[35]: data['age_dec'] = data.age.map(lambda age: 10 * (age // 10))
data.head()

Out[35]:
   age gender    split    final  split_sec  final_sec  split_frac  age_dec
0   33      M 01:05:3802:08:51     3938.0     7731.0   -0.018756       30
1   32      M 01:06:2602:09:28     3986.0     7768.0   -0.026262       30
2   31      M 01:06:4902:10:42     4009.0     7842.0   -0.022443       30
3   38      M 01:06:1602:13:45     3976.0     8025.0    0.009097       30
4   31      M 01:06:3202:13:59     3992.0     8039.0    0.006842       30

In[36]:
men = (data.gender == 'M')
```

```
women = (data.gender == 'W')
with sns.axes_style(style=None):
    sns.violinplot("age_dec", "split_frac", hue="gender", data=data,
                   split=True, inner="quartile",
                   palette=["lightblue", "lightpink"]);
```

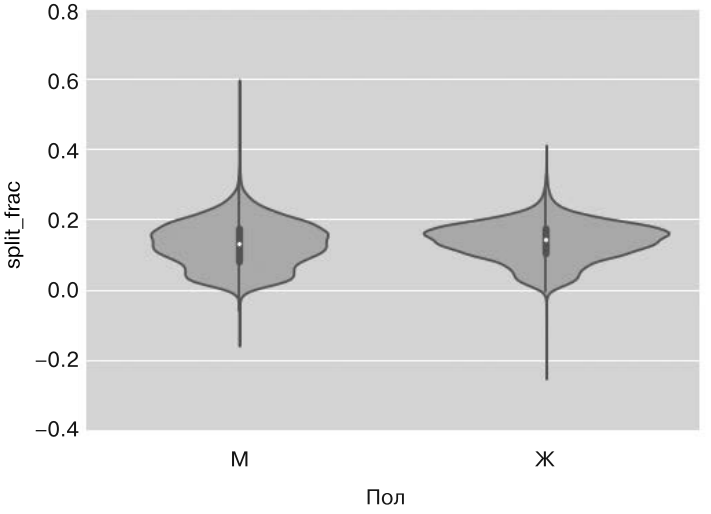

Рис. 4.130. «Скрипичная» диаграмма, показывающая зависимость коэффициента распределения от пола

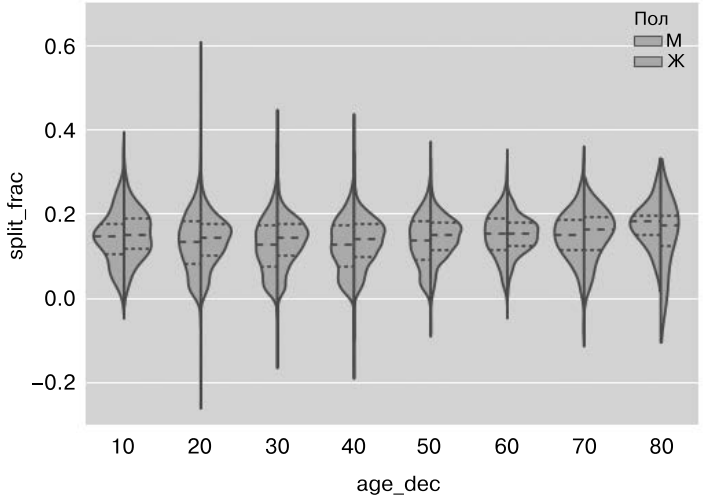

Рис. 4.131. «Скрипичная» диаграмма, отображающая зависимость коэффициента распределения от пола и возраста

При взгляде на этот график заметно, в чем именно различаются распределения для мужчин и женщин. Распределения для мужчин возраста от 20 до 50 лет демонстрируют выраженную склонность к более низким значениям коэффициента разбиения по сравнению с женщинами того же возраста (или вообще любого возраста).

И на удивление, 80-летняя женщина, похоже, обошла *всех* в смысле распределения сил. Вероятно, дело в том, что мы оцениваем распределение на малых количествах, ведь бегунов такого возраста всего несколько:

```
In[38]: (data.age > 80).sum()

Out[38]: 7
```

Возвращаясь к мужчинам с обратным распределением сил: кто эти бегуны? Существует ли корреляция между этим обратным распределением сил и быстрым прохождением марафона в целом? Мы можем легко построить соответствующий график. Воспользуемся функцией `regplot`, автоматически выполняющей подбор параметров линейной регрессии для имеющихся данных (рис. 4.132):

```
In[37]: g = sns.lmplot('final_sec', 'split_frac', col='gender', data=data,
                       markers=".", scatter_kws=dict(color='c'))
        g.map(plt.axhline, y=0.1, color="k", ls=":");
```

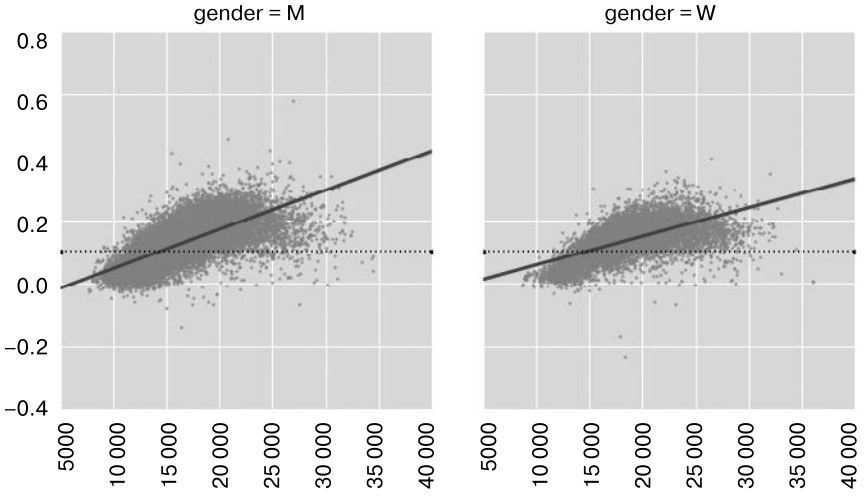

Рис. 4.132. Коэффициенты распределения сил по полу в зависимости от времени пробега

Как видим, люди с низким значением коэффициента распределения сил — элитные бегуны, финиширующие в пределах 15 000 секунд (примерно 4 часов). Вероятность подобного распределения сил для более медленных бегунов невелика.

Дополнительные источники информации

Источники информации о библиотеке Matplotlib

Бессмысленно надеяться охватить в одной главе все имеющиеся в Matplotlib возможности и типы графиков. Как и для других рассмотренных нами пакетов, разумное использование TAB-автодополнения и справочных возможностей оболочки IPython (см. раздел «Справка и документация в оболочке Python» главы 1) может принести немалую пользу при изучении API библиотеки Matplotlib. Кроме того, полезным источником информации может стать онлайн-документация Matplotlib (http://matplotlib.org/), в частности галерея Matplotlib (http://matplotlib.org/gallery.html). В ней представлены миниатюры сотен различных типов графиков, каждая из которых представляет собой ссылку на страницу с фрагментом кода на языке Python, используемым для его генерации. Таким образом, вы можете визуально изучить широкий диапазон различных стилей построения графиков и методик визуализации.

В качестве более обширного обзора библиотеки Matplotlib я бы рекомендовал вам обратить внимание на книгу *Interactive Applications Using Matplotlib* (http://bit.ly/2fSqswQ), написанную разработчиком ядра Matplotlib Беном Рутом.

Другие графические библиотеки языка Python

Хотя Matplotlib — наиболее значительная из предназначенных для визуализации библиотек языка Python, существуют и другие, более современные инструменты, заслуживающие пристального внимания. Я перечислю некоторые из них.

- Bokeh (http://bokeh.pydata.org/) — JavaScript-библиотека визуализации с клиентской частью для языка Python, предназначенная для создания высокоинтерактивных визуализаций с возможностью обработки очень больших и/или потоковых наборов данных. Клиентская часть Python возвращает структуры данных в формате JSON, интерпретируемые затем JavaScript-движком библиотеки Bokeh.
- Plotly (http://plot.ly/) — продукт с открытым исходным кодом одноименной компании, аналогичный по духу библиотеке Bokeh. Поскольку Plotly — основной продукт этого стартапа, разработчики прилагают максимум усилий к его разработке. Использовать эту библиотеку можно совершенно бесплатно.
- Vispy (http://vispy.org/) — активно разрабатываемый программный продукт, ориентированный на динамические визуализации очень больших наборов данных. В силу его ориентации на OpenGL и эффективное использование графических процессоров он способен формировать очень большие и впечатляющие визуализации.

- Vega (https://vega.github.io/) и Vega-Lite (https://vega.github.io/vega-lite) — описательные графические форматы, представляющие собой результат многих лет исследований в области фундаментального языка визуализации данных. Стандартная реализация — на языке JavaScript, но их API от языка не зависит. API для языка Python находится в процессе разработки, в пакете Altair (http://altair-viz.github.io/). Хотя он еще не вполне готов, меня радует сама возможность, что этот проект послужит общей точкой отсчета для визуализаций на языке Python и других языках программирования.

Сфера визуализации в Python-сообществе меняется очень динамично, и я уверен, что этот список устареет сразу после публикации. Внимательно следите за новостями в данной области!

5 Машинное обучение

Машинное обучение — основной способ демонстрации науки о данных широкой общественности. В машинном обучении вычислительные и алгоритмические возможности науки о данных соединяются со статистическим образом мышления, в результате возникает набор подходов к исследованию данных, связанных в основном с эффективностью не теории, а вычислений.

Термин «машинное обучение» иногда рассматривают как некую волшебную таблетку: *воспользуйся для своих данных машинным обучением — и все твои проблемы будут решены*! Однако реальность редко бывает столь проста. Хотя возможности этих методов огромны, для эффективного их использования необходимо хорошо разбираться в сильных и слабых сторонах каждого метода, равно как и в общих понятиях, таких как систематическая ошибки (bias) и дисперсия (variance), переобучение (overfitting) и недообучение (underfitting) и т. д.

В этой главе мы рассмотрим практические аспекты машинного обучения с помощью пакета Scikit-Learn (http://scikit-learn.org/) языка Python. Здесь не планируется всестороннее введение в сферу машинного обучения, поскольку это обширная тема, требующая более формализованного подхода. Не планируется и всестороннее руководство по пакету Scikit-Learn (такие руководства вы можете найти в разделе «Дополнительные источники информации по машинному обучению» этой главы).

Задачи данной главы — познакомить читателя:

- с базовой терминологией и понятиями машинного обучения;
- с API библиотеки Scikit-Learn и некоторыми примерами его использования;
- с подробностями нескольких наиболее важных методов машинного обучения, помочь разобраться в том, как они работают, а также где и когда применимы.

Большая часть материала взята из учебных курсов по Scikit-Learn, а также семинаров, проводившихся мной на PyCon, SciPy, PyData и других конференциях. Многолетние отзывы участников и других докладчиков семинаров позволили сделать изложение материала более доходчивым!

Если вам требуется всесторонний или более формализованный подход к любой из этих тем, я перечислил несколько источников информации и справочников в разделе «Дополнительные источники информации по машинному обучению» текущей главы.

Что такое машинное обучение

Прежде чем углубиться в подробности различных методов машинного обучения, разберемся, что представляет собой машинное обучение. Машинное обучение часто рассматривается как часть сферы искусственного интеллекта. Однако такая классификация, как мне кажется, нередко вводит в заблуждение. Исследования в области машинного обучения возникли на основе научных исследований в этой области, но в контексте приложения методов машинного обучения к науке о данных полезнее рассматривать машинное обучение как средство *создания моделей данных*.

Машинное обучение занимается построением математических моделей для исследования данных. Задачи «обучения» начинаются с появлением у этих моделей *настраиваемых параметров*, которые можно приспособить для отражения наблюдаемых данных, таким образом, программа как бы обучается на данных. Как только эти модели обучатся на имеющихся данных наблюдений, их можно будет использовать для предсказания и понимания различных аспектов данных новых наблюдений. Оставлю читателю в качестве самостоятельного задания обдумать философский вопрос о том, насколько подобное математическое, основанное на моделях обучение схоже с обучением человеческого мозга.

Для эффективного использования этих инструментов необходимо понимать общую формулировку задачи машинного обучения, поэтому начнем с широкой классификации типов подходов, которые мы будем обсуждать.

Категории машинного обучения

На базовом уровне машинное обучение можно разделить на два основных типа.

- *Машинное обучение с учителем* (supervised learning) — включает моделирование признаков данных и соответствующих данным меток. После выбора модели ее можно использовать для присвоения меток новым, неизвестным ранее данным. Оно разделяется далее на *задачи классификации* и *задачи регрессии*. При классификации метки представляют собой дискретные категории, а при регрессии они являются непрерывными величинами. Мы рассмотрим примеры обоих типов машинного обучения с учителем в следующем разделе.
- *Машинное обучение без учителя* (unsupervised learning) — включает моделирование признаков набора данных без каких-либо меток и описывается фразой «Пусть набор данных говорит сам за себя». Эти модели включают такие задачи, как *кластеризация* (clustering) и *понижение размерности* (dimensionality

reduction). Алгоритмы кластеризации служат для выделения отдельных групп данных, в то время как алгоритмы понижения размерности предназначены для поиска более сжатых представлений данных. Мы рассмотрим примеры обоих типов машинного обучения без учителя в следующем разделе.

Кроме того, существуют так называемые методы частичного обучения (semi-supervised learning), располагающиеся примерно посередине между машинным обучением с учителем и машинным обучением без учителя. Методы частичного обучения бывают полезны в случае наличия лишь неполных меток.

Качественные примеры прикладных задач машинного обучения

Чтобы конкретизировать вышеописанные понятия, рассмотрим несколько простых примеров задач машинного обучения. Цель этих примеров — дать интуитивно понятный обзор тех разновидностей машинного обучения, с которыми мы столкнемся в этой главе. В следующих разделах мы рассмотрим подробнее соответствующие модели и их использование. Чтобы получить представление о более технических аспектах, вы можете заглянуть в онлайн-приложение (https://github.com/jakevdp/PythonDataScienceHandbook), а также в генерирующие соответствующие рисунки исходные коды на языке Python.

Классификация: предсказание дискретных меток

Рассмотрим простую задачу классификации: имеется набор точек с метками и требуется классифицировать некоторое количество точек без меток.

Допустим, у нас есть показанные на рис. 5.1 данные (код, использовавшийся для генерации этого, как и всех остальных в этом разделе, рисунка, приведен в онлайн-приложении).

Наши данные двумерны, то есть для каждой точки имеются два *признака* (feature), которым соответствуют координаты *(x, y)* точки на плоскости. В дополнение каждой точке поставлена в соответствие одна из двух *меток класса* (class label), представленная определенным цветом точки. Нам требуется на основе этих признаков и меток создать модель, с помощью которой мы смогли бы определить, должна ли новая точка быть «синей» или «красной».

Существует множество возможных моделей для решения подобной задачи классификации, но мы воспользуемся исключительно простой моделью. Будем исходить из допущения, что наши две группы можно разделить прямой линией на плоскости, так что точки с одной стороны прямой будут принадлежать к одной группе. Данная модель представляет собой количественное выражение утверждения «прямая линия разделяет классы», в то время как *параметры модели* представляют собой конкретные числа, описывающие местоположение и направленность

этой прямой для наших данных. Оптимальные значения этих параметров модели получаются с помощью обучения на имеющихся данных (это и есть обучение в смысле машинного обучения), часто называемого *обучением модели* (training the model). Рисунок 5.2 демонстрирует вид обученной модели для наших данных.

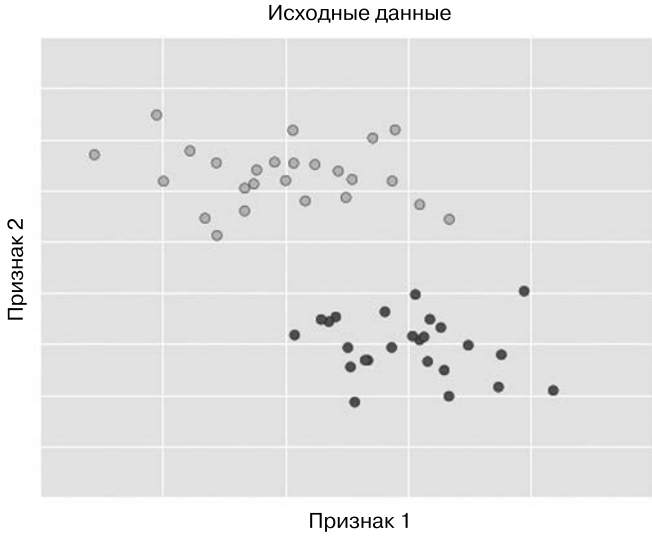

Рис. 5.1. Простой набор данных для классификации

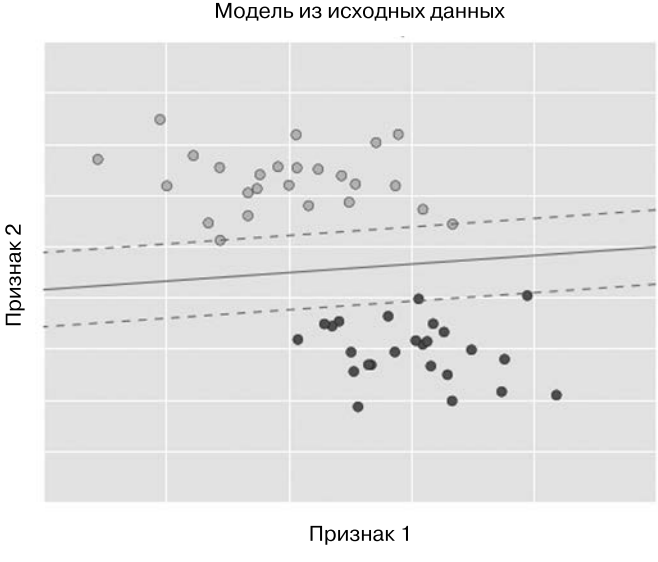

Рис. 5.2. Простая модель классификации

После обучения модели ее можно обобщить на новые, немаркированные данные. Другими словами, можно взять другой набор данных, провести прямую модели через них, после чего на основе этой прямой присвоить метки новым точкам. Этот этап обычно называют *предсказанием* (prediction) (рис. 5.3).

Такова основная идея задачи классификации в машинном обучении, причем слово «классификация» означает, что метки классов у данных дискретны. На первый взгляд это может показаться довольно простым: нет ничего сложного в том, чтобы просто посмотреть на данные и провести подобную разделяющую прямую для классификации данных. Достоинство подхода машинного обучения состоит в том, что его можно обобщить на большие наборы данных и большее количество измерений.

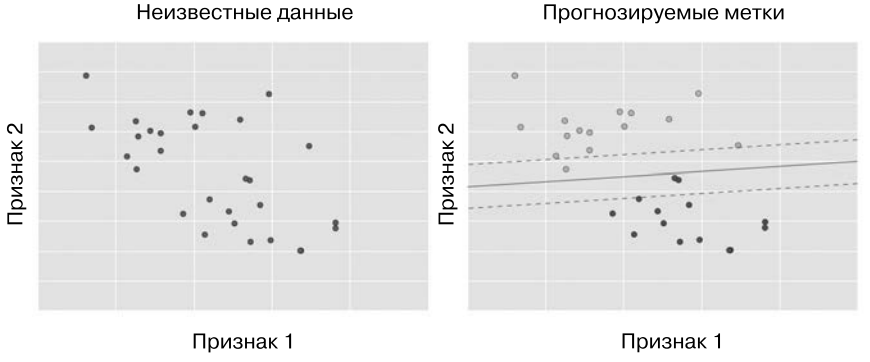

Рис. 5.3. Использование модели классификации для новых данных

Например, вышеописанное аналогично задаче автоматического обнаружения спама в электронной почте. В этом случае можно использовать следующие признаки и метки:

- *признак 1*, *признак 2* и т. д. → нормированные количества ключевых слов или фраз («Виагра», «Нигерийский принц» и т. д.);
- *метка* → «спам» или «не спам».

Для обучающей последовательности эти метки определяются путем индивидуального осмотра небольшой репрезентативной выборки сообщений электронной почты, для остальных сообщений электронной почты метка будет определяться с помощью модели. При обученном соответствующим образом алгоритме классификации с достаточно хорошо сконструированными признаками (обычно тысячи или миллионы слов/фраз) этот тип классификации может оказаться весьма эффективным. Мы рассмотрим пример подобной текстовой классификации в разделе «Заглянем глубже: наивная байесовская классификация» данной главы.

Самые важные алгоритмы классификации, которые мы обсудим в данной главе, — это Гауссов наивный байесовский классификатор (см. раздел «Заглянем глубже: наивная байесовская классификация» данной главы), метод опорных векторов (см. раздел «Заглянем глубже: метод опорных векторов» далее) и классификация на основе случайных лесов (см. раздел «Заглянем глубже: деревья принятия решений и случайные леса» этой главы).

Регрессия: предсказание непрерывных меток

В отличие от дискретных меток, с которыми мы имели дело в алгоритмах классификации, сейчас мы рассмотрим простую задачу *регрессии*, где метки представляют собой непрерывные величины.

Изучим показанные на рис. 5.4 данные, состоящие из набора точек с непрерывными метками.

Как и в примере классификации, наши данные двумерны, то есть каждая точка описывается двумя признаками. Непрерывные метки точек представлены их цветом.

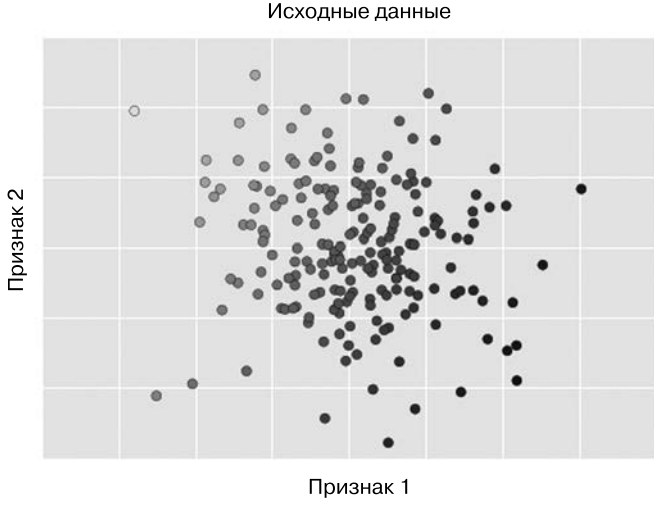

Рис. 5.4. Простой набор данных для регрессии

Существует много возможных моделей регрессии, подходящих для таких данных, но мы для предсказания меток для точек воспользуемся простой линейной регрессией. Модель простой линейной регрессии основана на допущении, что, если рассматривать метки как третье пространственное измерение, можно подобрать для этих данных разделяющую плоскость. Это высокоуровневое обобщение хорошо известной задачи подбора разделяющей прямой для данных с двумя координатами. Визуализировать это можно так, как показано на рис. 5.5.

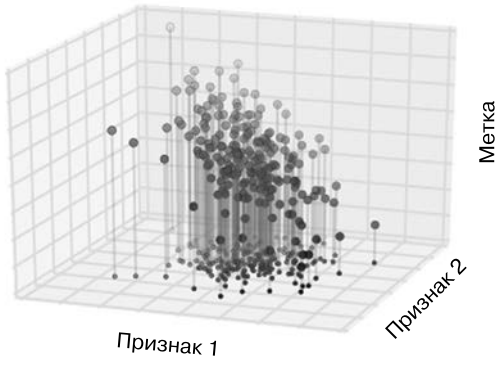

Рис. 5.5. Трехмерное представление данных регрессии

Обратите внимание, что плоскость *«признак 1 — признак 2»* здесь аналогична вышеприведенному двумерному графику. Однако в данном случае метки представлены как цветом, так и положением по третьей оси координат. Логично, что подбор разделяющей плоскости для этих трехмерных данных позволит нам предсказывать будущие метки для любых входных параметров. Возвращаясь к двумерной проекции, после подбора подобной плоскости мы получим результат, показанный на рис. 5.6.

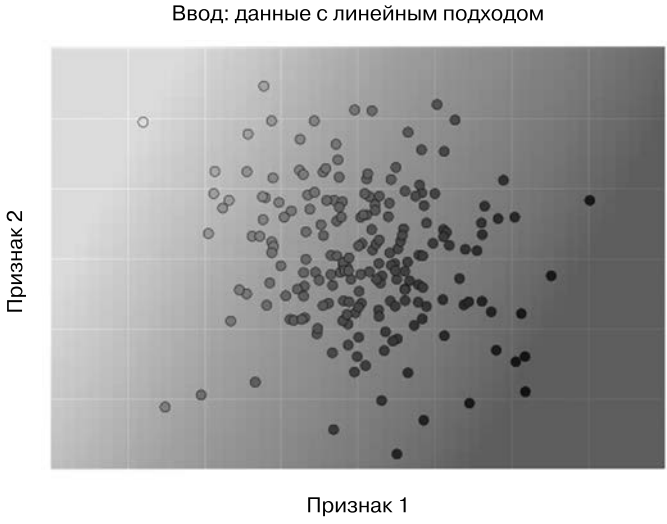

Рис. 5.6. Визуальное представление модели регрессии

С этой подобранной плоскостью у нас будет все нужное для предсказания меток для новых точек. Графически наши результаты будут выглядеть так, как показано на рис. 5.7.

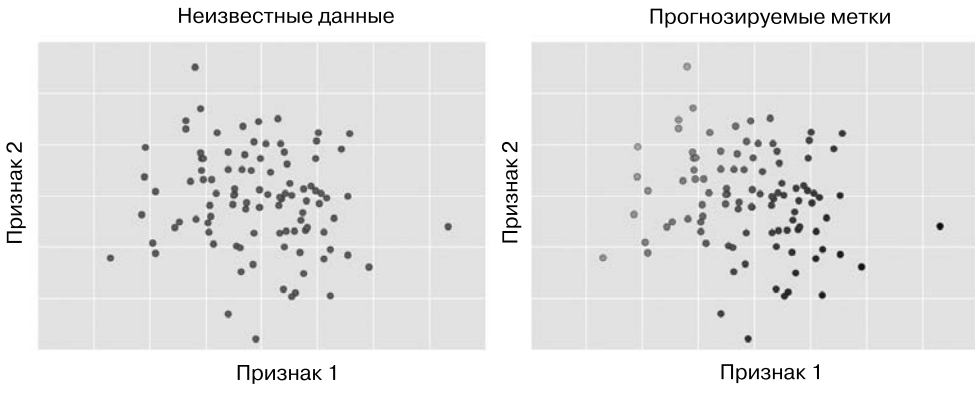

Рис. 5.7. Применение модели регрессии к новым данным

Как и в случае примера классификации, при небольшом количестве измерений эта задача может показаться вполне тривиальной. Однако сила этих методов заключается как раз в том, что их можно использовать и для данных с множеством признаков.

Например, вышеприведенное аналогично задаче вычисления расстояния до наблюдаемых в телескоп галактик — в данном случае можно использовать следующие признаки и метки:

- *признак 1*, *признак 2* и т. д. → яркость каждой галактики на одной из нескольких длин волн (цветов);
- *метка* → расстояние до галактики или ее красное смещение.

Можно вычислить расстояния до небольшого числа этих галактик на основании независимого набора (обычно более дорогостоящих) наблюдений. После этого можно оценить расстояния до оставшихся галактик с помощью подходящей модели регрессии, вместо того чтобы использовать более дорогостоящие наблюдения для всего нужного набора галактик. В астрономических кругах эта задача известна под названием «задачи фотометрического красного смещения».

Далее в этой главе мы будем обсуждать и такие важные регрессионные алгоритмы, как линейная регрессия (см. раздел «Заглянем глубже: линейная регрессия» этой главы), метод опорных векторов (см. раздел «Заглянем глубже: метод опорных векторов» этой главы) и регрессия на основе случайных лесов (см. раздел «Заглянем глубже: деревья принятия решений и случайные леса» далее).

Кластеризация: определение меток для немаркированных данных

Классификация и регрессия, которые мы только что рассмотрели, представляют собой примеры алгоритмов обучения с учителем, в которых создается модель с целью предсказания меток для новых данных. Обучение без учителя касается моделей для описания данных безотносительно каких-либо известных меток.

Часто встречающийся случай машинного обучения без учителя — кластеризация, при которой данные автоматически распределяются по некоторому количеству отдельных групп. Например, наши двумерные данные могли бы оказаться такими, как показаны на рис. 5.8.

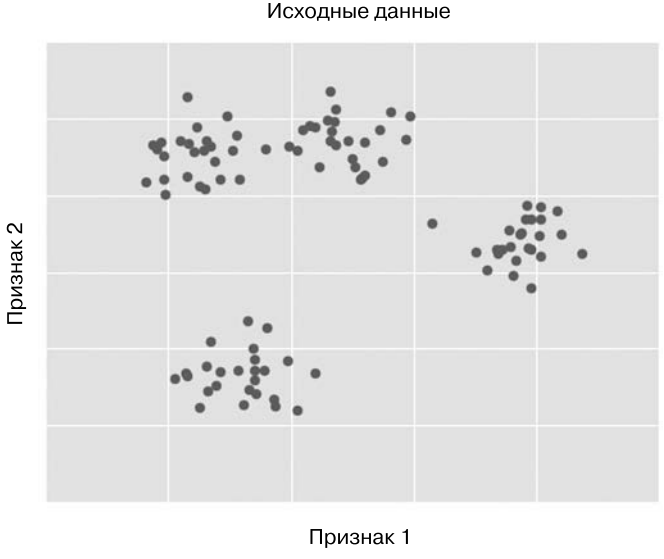

Рис. 5.8. Пример данных для кластеризации

Визуально очевидно, что каждая из этих точек относится к одной из нескольких групп. При таких входных данных модель кластеризации определит на основе внутренней их структуры, какие точки объединены одной группой. Воспользовавшись очень быстрым и интуитивно понятным алгоритмом кластеризации методом *k*-средних (см. раздел «Заглянем глубже: кластеризация методом k-средних» данной главы), мы получаем показанные на рис. 5.9 кластеры.

Метод *k*-средних выполняет обучение модели, состоящей из *k*-центров кластеров. Наилучшими считаются те центры, для которых расстояние от точек до соответствующих им центров минимально. В случае двух измерений эта задача может показаться тривиальной, но по мере усложнения данных и увеличения их объема подобные алгоритмы кластеризации можно использовать для извлечения из набора данных полезной информации.

Среди других важных алгоритмов кластеризации — смесь Гауссовых распределений (см. раздел «Заглянем глубже: смеси Гауссовых распределений» этой главы) и спектральная кластеризация (см. документацию по кластеризации библиотеки Scikit-Learn, http://scikit-learn.org/stable/modules/clustering.html).

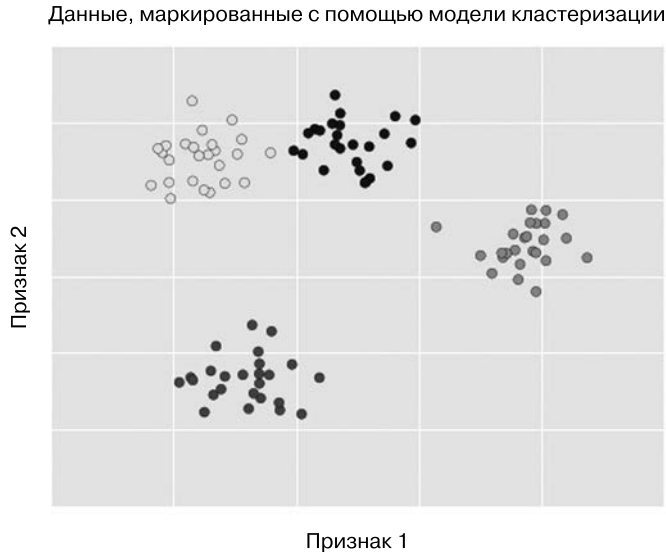

Рис. 5.9. Данные, маркированные с помощью модели кластеризации методом *k*-средних

Понижение размерности

Понижение размерности — еще один пример алгоритма обучения без учителя, в котором метки или другая информация определяются исходя из структуры самого набора данных. Алгоритм понижения размерности несколько труднее для понимания, чем рассмотренные нами ранее примеры, но он заключается в попытке получения представления низкой размерности, которое бы в какой-то мере сохраняло существенные качества полного набора данных. Различные алгоритмы понижения размерности оценивают существенность этих качеств по-разному, как мы увидим далее в разделе «Заглянем глубже: обучение на базе многообразий» этой главы.

В качестве примера рассмотрим данные, показанные на рис. 5.10. Зрительно очевидно, что у таких данных есть внутренняя структура: они получены из одномерной прямой, расположенной в двумерном пространстве в виде спирали. В некотором смысле можно сказать, что эти данные по своей внутренней сути одномерны, но вложены в пространство большей размерности. Подходящая модель понижения размерности в таком случае должна учитывать эту нелинейную

вложенную структуру, чтобы суметь получить из нее представление более низкой размерности.

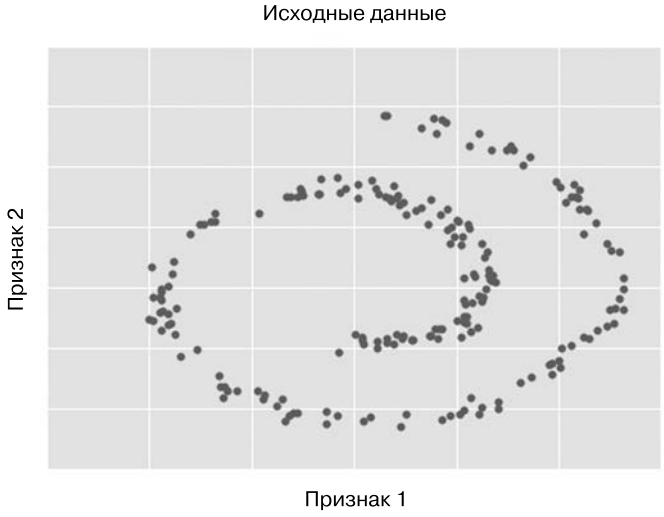

Рис. 5.10. Пример данных для понижения размерности

На рис. 5.11 представлена визуализация результатов работы алгоритма обучения на базе многообразий Isomap, который именно это и делает.

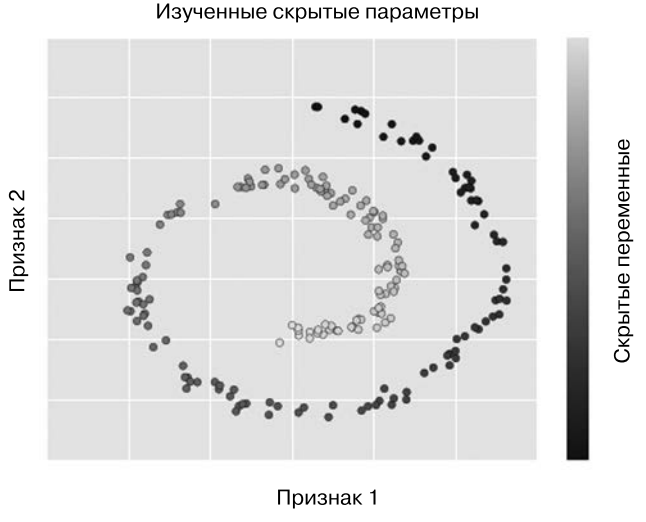

Рис. 5.11. Данные с метками, полученными с помощью алгоритма понижения размерности

Обратите внимание, что цвета, отражающие значения полученной одномерной скрытой переменной, меняются равномерно по ходу спирали, а значит, алгоритм действительно распознает видимую на глаз структуру. Масштаб возможностей алгоритмов понижения размерности становится очевиднее в случаях более высоких размерностей. Например, нам могло понадобиться визуализировать важные зависимости в наборе данных, у которого от 100 до 1000 признаков. Визуализация 1000-мерных данных представляет собой непростую задачу, один из способов решения которой — использование методик понижения размерности, чтобы свести данные к двум или трем измерениям.

Среди других важных алгоритмов понижения размерности — метод главных компонент (см. раздел «Заглянем глубже: метод главных компонент» данной главы) и различные методы обучения на базе многообразий, включая Isomap и локально линейное вложение (см. раздел «Заглянем глубже: обучение на базе многообразий» этой главы).

Резюме

Мы рассмотрели несколько простых примеров основных видов методов машинного обучения. Существует множество важных на практике нюансов, которые мы обошли вниманием, но я надеюсь, что вам хватило этого раздела для получения общего представления о том, какие виды задач позволяют решать методы машинного обучения.

Мы рассмотрели:

- *обучение с учителем* — модели для предсказания меток на основе маркированных обучающих данных;
- *классификацию* — модели для предсказания меток из двух или более отдельных категорий;
- *регрессию* — модели для предсказания непрерывных меток;
- *обучение без учителя* — модели для распознания структуры немаркированных данных;
- *кластеризацию* — модели для выявления и распознания в данных отдельных групп;
- *понижение размерности* — модели для выявления и распознания низкоразмерной структуры в высокоразмерных данных.

В следующих разделах мы изучим данные категории подробнее и увидим более интересные примеры использования этих принципов.

Все вышеприведенные рисунки были сгенерированы на основе выполнения настоящего машинного обучения, применявшийся для этого код вы можете найти в онлайн-приложении (http://github.com/jakevdp/PythonDataScienceHandbook).

Знакомство с библиотекой Scikit-Learn

Существует несколько библиотек языка Python с надежными реализациями широкого диапазона алгоритмов машинного обучения. Одна из самых известных — Scikit-Learn, пакет, предоставляющий эффективные версии множества распространенных алгоритмов. Пакет Scikit-Learn отличает аккуратный, единообразный и продвинутый API, а также удобная и всеохватывающая онлайн-документация. Преимущество этого единообразия в том, что, разобравшись в основах использования и синтаксисе Scikit-Learn для одного типа моделей, вы сможете легко перейти к другой модели или алгоритму.

В этом разделе вы найдете обзор API библиотеки Scikit-Learn. Ясное понимание элементов API — основа для более углубленного практического обсуждения алгоритмов и методов машинного обучения в следующих разделах.

Начнем с *представления данных* (data representation) в библиотеке Scikit-Learn, затем рассмотрим API Estimator (API статистического оценивания) и, наконец, взглянем на интересный пример использования этих инструментов для исследования набора изображений рукописных цифр.

Представление данных в Scikit-Learn

Машинное обучение связано с созданием моделей на основе данных, поэтому начнем с обсуждения понятного компьютеру представления данных. Лучше всего представлять используемые в библиотеке Scikit-Learn данные в виде таблиц.

Данные как таблица

Простейшая таблица — двумерная сетка данных, в которой строки представляют отдельные элементы набора данных, а столбцы — атрибуты, связанные с каждым из этих элементов. Например, рассмотрим набор данных Iris (https://en.wikipedia.org/wiki/Iris_flower_data_set), проанализированный Рональдом Фишером в 1936 году. Скачаем его в виде объекта `DataFrame` библиотеки Pandas с помощью библиотеки Seaborn:

```
In[1]: import seaborn as sns
       iris = sns.load_dataset('iris')
       iris.head()

Out[1]:    sepal_length  sepal_width  petal_length  petal_width species
        0           5.1          3.5           1.4          0.2  setosa
        1           4.9          3.0           1.4          0.2  setosa
        2           4.7          3.2           1.3          0.2  setosa
        3           4.6          3.1           1.5          0.2  setosa
        4           5.0          3.6           1.4          0.2  setosa
```

Каждая строка данных относится к одному из измеренных цветков, а количество строк равно полному количеству цветков в наборе данных. Мы будем называть столбцы этой матрицы *выборками* (samples), а количество строк полагать равным `n_samples`.

Каждый столбец данных относится к конкретному количественному показателю, описывающему данную выборку. Мы будем называть столбцы матрицы *признаками* (features), а количество столбцов полагать равным `n_features`.

Матрица признаков

Из устройства таблицы очевидно, что информацию можно рассматривать как двумерный числовой массив или матрицу, которую мы будем называть *матрицей признаков* (features matrix). По традиции матрицу признаков часто хранят в переменной `X`. Предполагается, что матрица признаков — двумерная, с формой `[n_samples, n_features]`, и хранят ее чаще всего в массиве NumPy или объекте `DataFrame` библиотеки Pandas, хотя некоторые модели библиотеки Scikit-Learn допускают использование также разреженных матриц из библиотеки SciPy.

Выборки (то есть строки) всегда соответствуют отдельным объектам, описываемым набором данных. Например, выборка может быть цветком, человеком, документом, изображением, звуковым файлом, видеофайлом, астрономическим объектом или чем-то еще, что можно описать с помощью набора количественных измерений.

Признаки (то есть столбцы) всегда соответствуют конкретным наблюдениям, описывающим каждую из выборок количественным образом. Значения признаков обычно представляют собой вещественные числа, но в некоторых случаях они могут быть булевыми или иметь дискретные[1] значения.

Целевой массив

Помимо матрицы признаков `x`, обычно мы имеем дело с *целевым массивом* (массивом *меток*), который принято обозначать `y`. Целевой массив обычно одномерен, длиной `n_samples`. Его хранят в массиве NumPy или объекте `Series` библиотеки Pandas. Значения целевого массива могут быть непрерывными числовыми или дискретными классами/метками. Хотя некоторые оцениватели библиотеки Scikit-Learn умеют работать с несколькими целевыми величинами в виде двумерного целевого массива `[n_samples, n_targets]`, мы в основном будем работать с более простым случаем одномерного целевого массива.

Отличительная черта целевого массива от остальных столбцов признаков в том, что он представляет собой величину, значения которой мы хотим *предсказать на*

[1] Перечислимые.

основе имеющихся данных. Говоря статистическим языком, это зависимая переменная (dependent variable). Например, для предыдущих данных это могла оказаться модель для предсказания вида цветка на основе остальных измерений. В таком случае столбец `species` рассматривался бы как целевой массив.

С учетом вышесказанного можно воспользоваться библиотекой Seaborn (которую мы рассматривали в разделе «Визуализация с помощью библиотеки Seaborn» главы 4), чтобы без труда визуализировать данные (рис. 5.12):

```
In[2]: %matplotlib inline
       import seaborn as sns; sns.set()
       sns.pairplot(iris, hue='species', size=1.5);
```

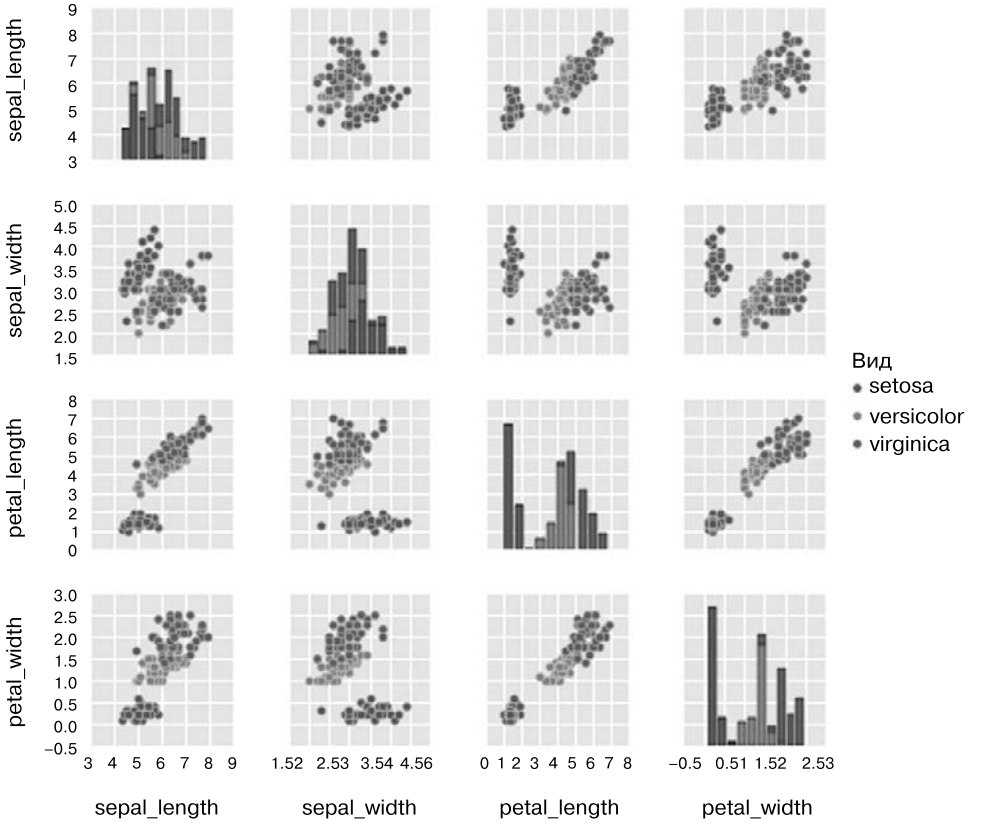

Рис. 5.12. Визуализация набора данных Iris

Для использования набора данных Iris в Scikit-Learn мы извлечем матрицу признаков и целевой массив из объекта `DataFrame`. Сделать это можно с помощью обсуждавшихся в главе 3 операций объекта `DataFrame` библиотеки Pandas:

```
In[3]: X_iris = iris.drop('species', axis=1)
       X_iris.shape

Out[3]: (150, 4)

In[4]: y_iris = iris['species']
       y_iris.shape

Out[4]: (150,)
```

В итоге признаки и целевые величины должны иметь следующий вид (рис. 5.13):

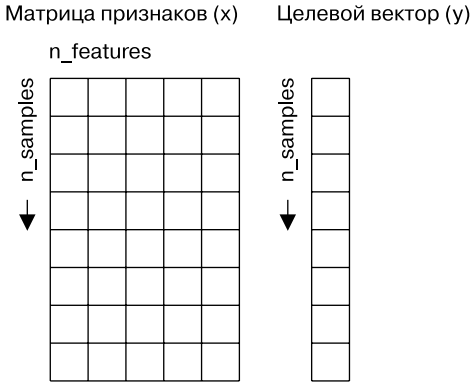

Рис. 5.13. Структура данных Scikit-Learn

Теперь, отформатировав наши данные нужным образом, мы можем перейти к рассмотрению API *статистических оценок* библиотеки Scikit-Learn.

API статистического оценивания библиотеки Scikit-Learn

В документации по API Scikit-Learn говорится, что он основывается на следующих принципах:

- *единообразие* — интерфейс всех объектов идентичен и основан на ограниченном наборе методов, причем документация тоже единообразна;
- *контроль* — видимость всех задаваемых значений параметров как открытых атрибутов;
- *ограниченная иерархия объектов* — классы языка Python используются только для алгоритмов; наборы данных представлены в стандартных форматах (массивы NumPy, объекты `DataFrame` библиотеки Pandas, разреженные матрицы библиотеки SciPy), а для имен параметров используются стандартные строки языка Python;
- *объединение* — многие из задач машинного обучения можно выразить в виде последовательностей алгоритмов более низкого уровня, и библиотека Scikit-Learn пользуется этим фактом при любой возможности;

- *разумные значения по умолчанию* — библиотека задает для необходимых моделей пользовательских параметров соответствующие значения по умолчанию.

На практике эти принципы очень облегчают изучение библиотеки Scikit-Learn. Все алгоритмы машинного обучения в библиотеке Scikit-Learn реализуются через API статистического оценивания, предоставляющий единообразный интерфейс для широкого диапазона прикладных задач машинного обучения.

Основы API статистического оценивания

Чаще всего использование API статистического оценивания библиотеки Scikit-Learn включает следующие шаги (далее мы рассмотрим несколько подробных примеров).

1. Выбор класса модели с помощью импорта соответствующего класса оценивателя из библиотеки Scikit-Learn.
2. Выбор гиперпараметров модели путем создания экземпляра этого класса с соответствующими значениями.
3. Компоновка данных в матрицу признаков и целевой вектор в соответствии с описанным выше.
4. Обучение модели на своих данных посредством вызова метода `fit()` экземпляра модели.
5. Применение модели к новым данным:
   - в случае машинного обучения с учителем метки для неизвестных данных обычно предсказывают с помощью метода `predict()`;
   - в случае машинного обучения без учителя выполняется преобразование свойств данных или вывод их значений посредством методов `transform()` или `predict()`.

Рассмотрим несколько простых примеров применения методов обучения без учителя и с учителем.

Пример обучения с учителем: простая линейная регрессия

В качестве примера этого процесса возьмем простую линейную регрессию, то есть часто встречающийся случай подбора разделяющей прямой для данных вида *(x, y)*. Для этого примера возьмем следующие простые данные (рис. 5.14):

```
In[5]: import matplotlib.pyplot as plt
       import numpy as np

       rng = np.random.RandomState(42)
       x = 10 * rng.rand(50)
       y = 2 * x - 1 + rng.randn(50)
       plt.scatter(x, y);
```

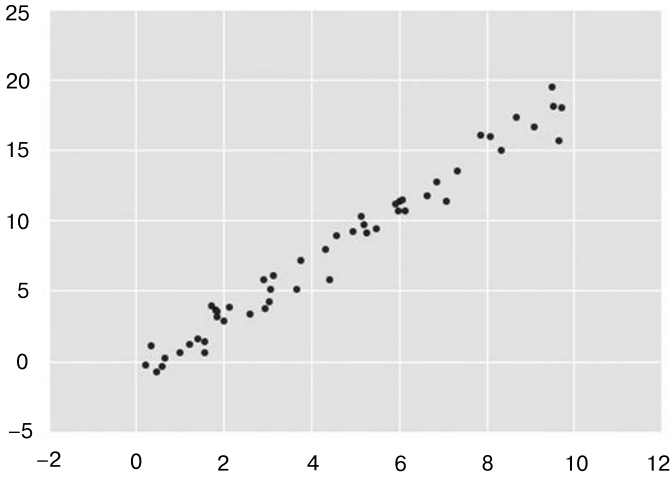

Рис. 5.14. Данные для линейной регрессии

Затем мы можем воспользоваться описанным выше рецептом. Пройдемся по всем шагам этого процесса.

1. Выбор класса модели.

   Каждый класс модели в библиотеке Scikit-Learn представлен соответствующим классом языка Python. Так, например, для расчета модели простой линейной регрессии можно импортировать класс линейной регрессии:

   ```
   In[6]: from sklearn.linear_model import LinearRegression
   ```

   Обратите внимание, что существуют и другие, более общие модели линейной регрессии, прочитать о них подробнее вы можете в документации модуля `sklearn.linear_model` (http://scikit-learn.org/stable/modules/linear_model.html).

2. Выбор гиперпараметров модели.

   Подчеркнем важный момент: *класс модели — не то же самое, что экземпляр модели.*

   После выбора класса модели у нас все еще остаются некоторые возможности для выбора. В зависимости от нашего класса модели может понадобиться ответить на один или несколько следующих вопросов.

   - Хотим ли мы выполнить подбор сдвига прямой (то есть точки пересечения с осью координат)?
   - Хотим ли мы нормализовать модель?
   - Хотим ли мы сделать модель более гибкой, выполнив предварительную обработку признаков?
   - Какая степень регуляризации должна быть у нашей модели?
   - Сколько компонент модели мы хотели бы использовать?

Это примеры тех важных решений, которые нам придется принять *после выбора класса модели*. Результаты этих решений часто называют *гиперпараметрами*, то есть параметрами, задаваемыми до обучения модели на данных. Выбор гиперпараметров в библиотеке Scikit-Learn осуществляется путем передачи значений при создании экземпляра модели. Мы рассмотрим количественные обоснования выбора гиперпараметров в разделе «Гиперпараметры и проверка модели» данной главы.

Создадим экземпляр класса `LinearRegression` и укажем с помощью гиперпараметра `fit_intercept`, что нам бы хотелось выполнить подбор точки пересечения с осью координат:

```
In[7]: model = LinearRegression(fit_intercept=True)
       model

Out[7]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1,
                         normalize=False)
```

Помните, что при создании экземпляра модели выполняется только сохранение значений этих гиперпараметров. В частности, мы все еще не применили модель ни к каким данным: API библиотеки Scikit-Learn очень четко разделяет *выбор модели* и *применение модели к данным*.

3. Формирование из данных матриц признаков и целевого вектора.

   Ранее мы подробно рассмотрели представление данных в библиотеке Scikit-Learn, для которого необходимы двумерная матрица признаков и одномерный целевой вектор. Наша целевая переменная `y` уже имеет нужный вид (массив длиной `n_samples`), но нам придется проделать небольшие манипуляции с данными `x`, чтобы сделать из них матрицу размера `[n_samples, n_features]`. В данном случае манипуляции сводятся просто к изменению формы одномерного массива:

```
In[8]: X = x[:, np.newaxis]
       X.shape

Out[8]: (50, 1)
```

4. Обучение модели на наших данных.

   Пришло время применить модель к данным. Сделать это можно с помощью метода `fit()` модели:

```
In[9]: model.fit(X, y)

Out[9]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1,
                         normalize=False)
```

Команда `fit()` вызывает выполнение «под капотом» множества вычислений, в зависимости от модели, и сохранение результатов этих вычислений в атрибутах модели, доступных для просмотра пользователем. В библиотеке Scikit-Learn по

традиции все параметры модели, полученные в процессе выполнения команды `fit()`, содержат в конце названия знак подчеркивания. Например, в данной линейной модели:

```
In[10]: model.coef_

Out[10]: array([ 1.9776566])

In[11]: model.intercept_

Out[11]: -0.90331072553111635
```

Эти два параметра представляют собой угловой коэффициент и точку пересечения с осью координат для простой линейной аппроксимации наших данных. Сравнивая с описанием данных, видим, что они очень близки к исходному угловому коэффициенту, равному 2, и точке пересечения, равной –1.

Часто возникает вопрос относительно погрешностей в подобных внутренних параметрах модели. В целом библиотека Scikit-Learn не предоставляет инструментов, позволяющих делать выводы непосредственно из внутренних параметров модели: интерпретация параметров скорее вопрос *статистического моделирования*, а не машинного обучения. Машинное обучение концентрируется в основном на том, что *предсказывает* модель. Для тех, кто хочет узнать больше о смысле подбираемых параметров модели, существуют другие инструменты, включая пакет StatsModels языка Python (http://statsmodels.sourceforge.net/).

5. Предсказание меток для новых данных.

   После обучения модели главная задача машинного обучения с учителем заключается в вычислении с ее помощью значений для новых данных, не являющихся частью обучающей последовательности. Сделать это в библиотеке Scikit-Learn можно посредством метода `predict()`. В этом примере наши новые данные будут сеткой x-значений и нас будет интересовать, какие y-значения предсказывает модель:

   ```
   In[12]: xfit = np.linspace(-1, 11)
   ```

   Как и ранее, эти x-значения требуется преобразовать в матрицу признаков `[n_samples, n_features]`, после чего можно подать их на вход модели:

   ```
   In[13]: Xfit = xfit[:, np.newaxis]
           yfit = model.predict(Xfit)
   ```

   Наконец, визуализируем результаты, нарисовав сначала график исходных данных, а затем обученную модель (рис. 5.15):

   ```
   In[14]: plt.scatter(x, y)
           plt.plot(xfit, yfit);
   ```

   Обычно эффективность модели оценивают, сравнивая ее результаты с эталоном, как мы увидим в следующем примере.

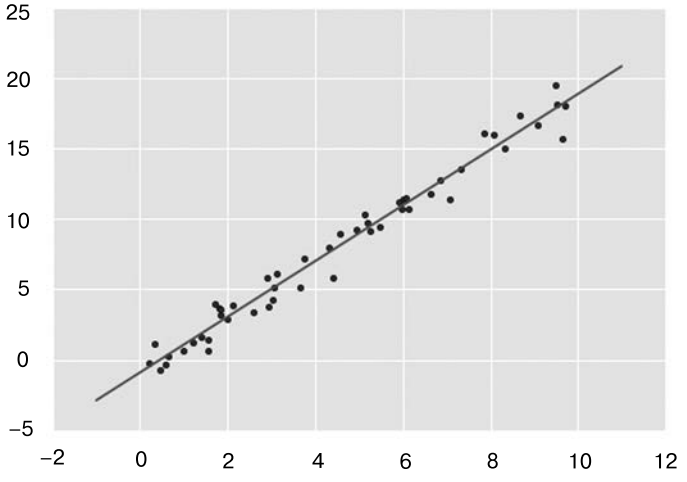

Рис. 5.15. Простая линейная регрессионная аппроксимация наших данных

Пример обучения с учителем: классификация набора данных Iris

Рассмотрим другой пример того же процесса, воспользовавшись обсуждавшимся ранее набором данных Iris. Зададимся вопросом: насколько хорошо мы сможем предсказать метки остальных данных с помощью модели, обученной на некоторой части данных набора Iris?

Для этой задачи мы воспользуемся чрезвычайно простой обобщенной моделью, известной под названием *«Гауссов наивный байесовский классификатор»*, исходящей из допущения, что все классы взяты из выровненного по осям координат Гауссова распределения (см. раздел «Заглянем глубже: наивная байесовская классификация» данной главы). Гауссов наивный байесовский классификатор в силу отсутствия гиперпараметров и высокой производительности — хороший кандидат на роль эталонной классификации. Имеет смысл поэкспериментировать с ним, прежде чем выяснять, можно ли получить лучшие результаты с помощью более сложных моделей.

Мы собираемся проверить работу модели на неизвестных ей данных, так что необходимо разделить данные на *обучающую последовательность* (training set) и *контрольную последовательность* (testing set). Это можно сделать вручную, но удобнее воспользоваться вспомогательной функцией `train_test_split`:

```
In[15]: from sklearn.cross_validation import train_test_split
        Xtrain, Xtest, ytrain, ytest = train_test_split(X_iris, y_iris,
                                                        random_state=1)
```

После упорядочения данных последуем нашему рецепту для предсказания меток:

```
In[16]: from sklearn.naive_bayes import GaussianNB # 1. Выбираем класс модели
        model = GaussianNB()                       # 2. Создаем экземпляр модели
```

```
        model.fit(Xtrain, ytrain)              # 3. Обучаем модель на данных
        y_model = model.predict(Xtest)         # 4. Предсказываем значения
                                               # для новых данных
```

Воспользуемся утилитой `accuracy_score` для выяснения того, какая часть предсказанных меток соответствует истинному значению:

```
In[17]: from sklearn.metrics import accuracy_score
        accuracy_score(ytest, y_model)

Out[17]: 0.97368421052631582
```

Как видим, точность превышает 97 %, поэтому для этого конкретного набора данных даже очень наивный алгоритм классификации оказывается эффективным!

Пример обучения без учителя: понижение размерности набора данных Iris

В качестве примера задачи обучения без учителя рассмотрим задачу понижения размерности набора данных Iris с целью упрощения его визуализации. Напомню, что данные Iris четырехмерны: для каждой выборки зафиксированы четыре признака.

Задача понижения размерности заключается в выяснении, существует ли подходящее представление более низкой размерности, сохраняющее существенные признаки данных. Зачастую понижение размерности используется для облегчения визуализации данных, в конце концов, гораздо проще строить график данных в двух измерениях, чем в четырех или более!

В этом разделе мы будем использовать метод главных компонент (PCA; см. раздел «Заглянем глубже: метод главных компонент» данной главы), представляющий собой быстрый линейный метод понижения размерности. Наша модель должна будет возвращать две компоненты, то есть двумерное представление данных.

Следуя вышеописанной последовательности шагов, получаем:

```
In[18]:
from sklearn.decomposition import PCA  # 1. Выбираем класс модели
model = PCA(n_components=2)      # 2. Создаем экземпляр модели
                                 # с гиперпараметрами
model.fit(X_iris)                # 3. Обучаем модель на данных. Обратите
                                 # внимание, что у мы не указываем!
X_2D = model.transform(X_iris)   # 4. Преобразуем данные в двумерные
```

Построим график полученных результатов. Сделать это быстрее всего можно, вставив результаты в исходный объект `DataFrame` Iris и воспользовавшись функцией `lmplot` для отображения результатов:

```
In[19]: iris['PCA1'] = X_2D[:, 0]
        iris['PCA2'] = X_2D[:, 1]
        sns.lmplot("PCA1", "PCA2", hue='species', data=iris, fit_reg=False);
```

Мы видим, что в двумерном представлении виды цветов четко разделены, хотя алгоритм PCA ничего не знает о метках видов цветов! Это показывает, что, как мы и видели ранее, даже относительно простая классификация на этом наборе данных, вероятно, будет работать достаточно хорошо (рис. 5.16).

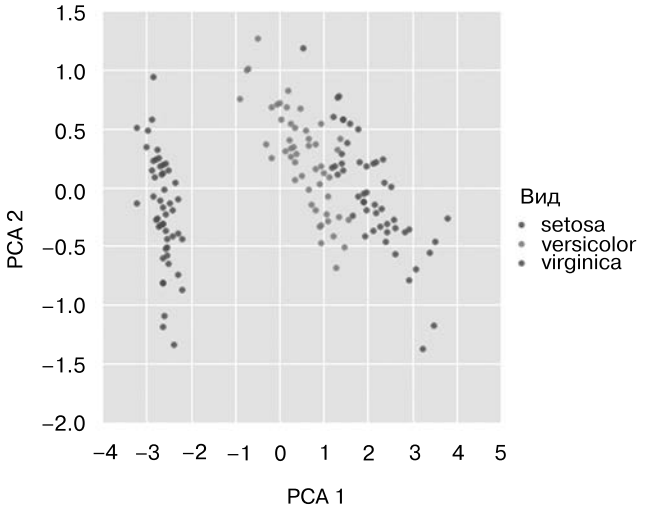

Рис. 5.16. Проекция данных набора Iris на двумерное пространство

Обучение без учителя: кластеризация набора данных Iris

Теперь рассмотрим кластеризацию набора данных Iris. Алгоритм кластеризации пытается выделить группы данных безотносительно к каким-либо меткам. Здесь мы собираемся использовать мощный алгоритм кластеризации под названием смесь Гауссовых распределений (Gaussian mixture model, GMM), которую изучим подробнее в разделе «Заглянем глубже: смеси Гауссовых распределений» этой главы. Метод GMM состоит в попытке моделирования данных в виде набора Гауссовых пятен.

Обучаем смесь Гауссовых распределений следующим образом:

```
In[20]:
from sklearn.mixture import GMM      # 1. Выбираем класс модели
model = GMM(n_components=3, covariance_type='full')  # 2. Создаем экземпляр
                                                     # модели
с гиперпараметрами
model.fit(X_iris)                    # 3. Обучаем модель на данных. Обратите
                                     # внимание, что у мы не указываем!
y_gmm = model.predict(X_iris)        # 4. Определяем метки кластеров
```

Как и ранее, добавим столбец `cluster` в `DataFrame` Iris и воспользуемся библиотекой Seaborn для построения графика результатов (рис. 5.17):

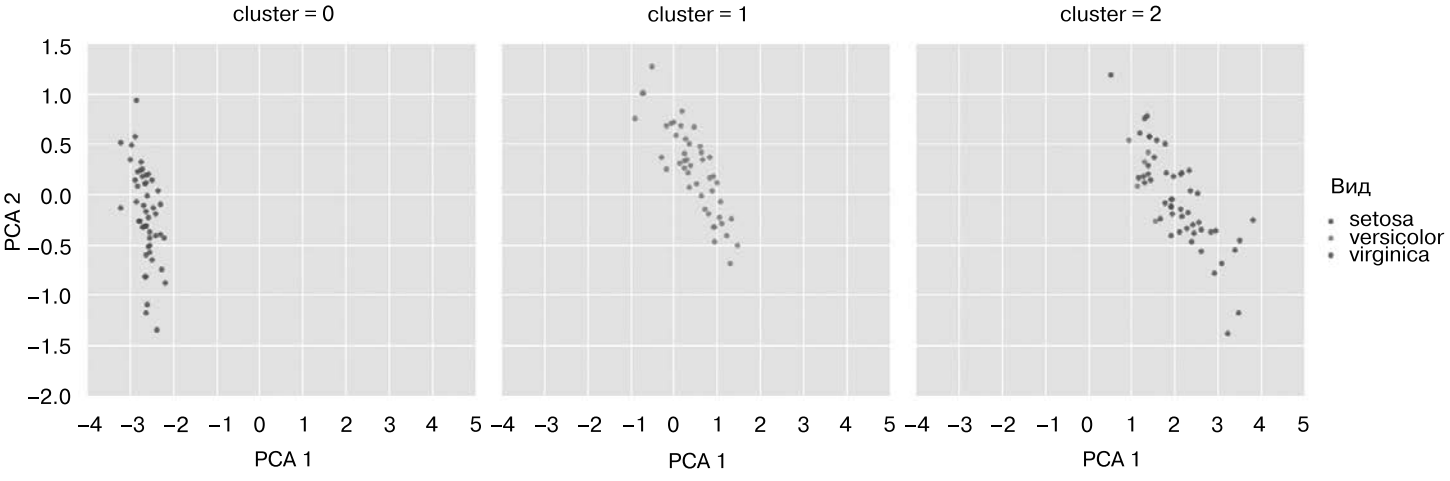

Рис. 5.17. Кластеризация методом k-средних в наборе данных Iris

```
In[21]:
iris['cluster'] = y_gmm
sns.lmplot("PCA1", "PCA2", data=iris, hue='species',
           col='cluster', fit_reg=False);
```

Разбив данные в соответствии с номерами кластеров, мы видим, насколько хорошо алгоритм GMM восстановил требуемые метки: вид *setosa* идеально выделен в кластер 0, правда, небольшое количество экземпляров видов *versicolor* и *virginica* смешались между собой. Следовательно, даже если у нас нет эксперта, который мог бы сообщить нам, к каким видам относятся отдельные цветки, одних измерений вполне достаточно для *автоматического* распознания этих различных разновидностей цветков с помощью простого алгоритма кластеризации! Подобный алгоритм может в дальнейшем помочь специалистам по предметной области выяснить связи между исследуемыми образцами.

Прикладная задача: анализ рукописных цифр

Продемонстрируем эти принципы на более интересной задаче, рассмотрев один из аспектов задачи оптического распознавания символов — распознавание рукописных цифр. Традиционно эта задача включает как определение местоположения на рисунке, так и распознание символов. Мы пойдем самым коротким путем и воспользуемся встроенным в библиотеку Scikit-Learn набором преформатированных цифр.

Загрузка и визуализация цифр

Воспользуемся интерфейсом доступа к данным библиотеки Scikit-Learn и посмотрим на эти данные:

```
In[22]: from sklearn.datasets import load_digits
        digits = load_digits()
        digits.images.shape

Out[22]: (1797, 8, 8)
```

Данные изображений представляют собой трехмерный массив: 1797 выборок, каждая состоит из сетки пикселов размером 8×8. Визуализируем первую их сотню (рис. 5.18):

```
In[23]: import matplotlib.pyplot as plt

        fig, axes = plt.subplots(10, 10, figsize=(8, 8),
                                 subplot_kw={'xticks':[], 'yticks':[]},
                                 gridspec_kw=dict(hspace=0.1, wspace=0.1))

        for i, ax in enumerate(axes.flat):
            ax.imshow(digits.images[i], cmap='binary',
                 interpolation='nearest')
```

```
        ax.text(0.05, 0.05, str(digits.target[i]),
                transform=ax.transAxes, color='green')
```

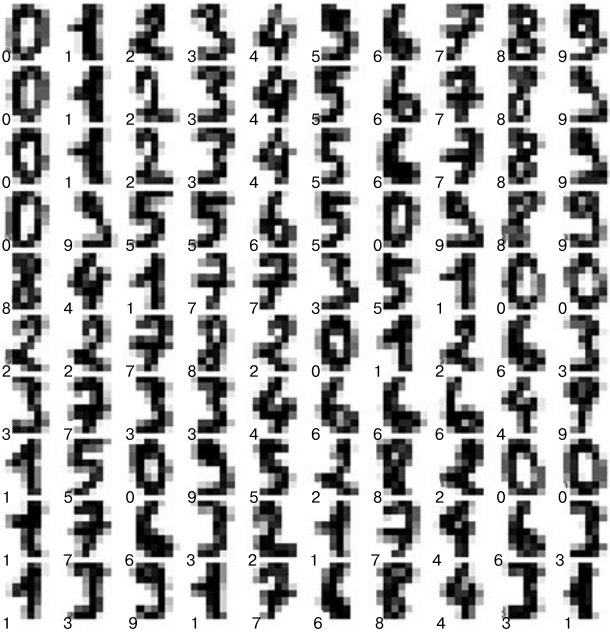

Рис. 5.18. Данные рукописных цифр; каждая выборка представлена сеткой пикселов размером 8 × 8

Для работы с этими данными в библиотеке Scikit-Learn нам нужно получить их двумерное `[n_samples, n_features]` представление. Для этого мы будем трактовать каждый пиксел в изображении как признак, то есть «расплющим» массивы пикселов так, чтобы каждую цифру представлял массив пикселов длиной 64 элемента. Кроме этого, нам понадобится целевой массив, задающий для каждой цифры предопределенную метку. Эти два параметра встроены в набор данных цифр в виде атрибутов `data` и `target`, соответственно:

```
In[24]: X = digits.data
        X.shape

Out[24]: (1797, 64)

In[25]: y = digits.target
        y.shape

Out[25]: (1797,)
```

Итого получаем 1797 выборок и 64 признака.

Обучение без учителя: понижение размерности

Хотелось бы визуализировать наши точки в 64-мерном параметрическом пространстве, но эффективно визуализировать точки в пространстве такой высокой размерности непросто. Понизим вместо этого количество измерений до 2, воспользовавшись методом обучения без учителя. Здесь мы будем применять алгоритм обучения на базе многообразий под названием Isomap (см. раздел «Заглянем глубже: обучение на базе многообразий» этой главы) и преобразуем данные в двумерный вид:

```
In[26]: from sklearn.manifold import Isomap
        iso = Isomap(n_components=2)
        iso.fit(digits.data)
        data_projected = iso.transform(digits.data)
        data_projected.shape

Out[26]: (1797, 2)
```

Теперь наши данные стали двумерными. Построим график этих данных, чтобы увидеть, можно ли что-то понять из их структуры (рис. 5.19):

```
In[27]: plt.scatter(data_projected[:, 0], data_projected[:, 1],
                    c=digits.target, edgecolor='none', alpha=0.5,
                    cmap=plt.cm.get_cmap('spectral', 10))
        plt.colorbar(label='digit label', ticks=range(10))
        plt.clim(-0.5, 9.5);
```

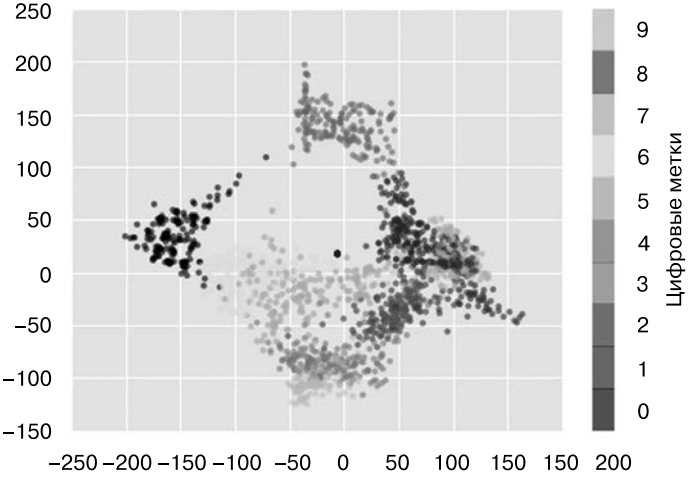

Рис. 5.19. Isomap-вложение набора данных цифр

Этот график дает нам представление о разделении различных цифр в 64-мерном пространстве. Например, нули (отображаемые черным цветом) и единицы

(отображаемые фиолетовым) практически не пересекаются в параметрическом пространстве. На интуитивном уровне это представляется вполне логичным: нули содержат пустое место в середине изображения, а у единиц там, наоборот, чернила. С другой стороны, единицы и четверки на графике располагаются сплошным спектром, что понятно, ведь некоторые люди рисуют единицы со «шляпками», из-за чего они становятся похожи на четверки.

В целом различные группы достаточно хорошо разнесены в параметрическом пространстве. Это значит, что даже довольно простой алгоритм классификации с учителем должен работать на них достаточно хорошо.

Классификация цифр

Применим алгоритм классификации к нашим цифрам. Как и в случае с набором данных Iris, разобьем данные на обучающую и контрольную последовательности, после чего обучим на первой из них Гауссову наивную байесовскую модель таким образом:

```
In[28]: Xtrain, Xtest, ytrain, ytest =
        train_test_split(X, y, random_state=0)

In[29]: from sklearn.naive_bayes import GaussianNB
        model = GaussianNB()
        model.fit(Xtrain, ytrain)
        y_model = model.predict(Xtest)
```

Теперь, осуществив предсказания по нашей модели, мы можем оценить ее точность, сравнив настоящие значения из контрольной последовательности с предсказанными:

```
In[30]: from sklearn.metrics import accuracy_score
        accuracy_score(ytest, y_model)

Out[30]: 0.83333333333333337
```

Даже при такой исключительно простой модели мы получили более чем 80%-ную точность классификации цифр! Однако из одного числа сложно понять, *где* наша модель ошиблась. Для этой цели удобна так называемая *матрица различий* (confusion matrix), вычислить которую можно с помощью библиотеки Scikit-Learn, а нарисовать посредством Seaborn (рис. 5.20):

```
In[31]: from sklearn.metrics import confusion_matrix
        mat = confusion_matrix(ytest, y_model)
        sns.heatmap(mat, square=True, annot=True, cbar=False)
        plt.xlabel('predicted value') # Прогнозируемое значение
        plt.ylabel('true value');     # Настоящее значение
```

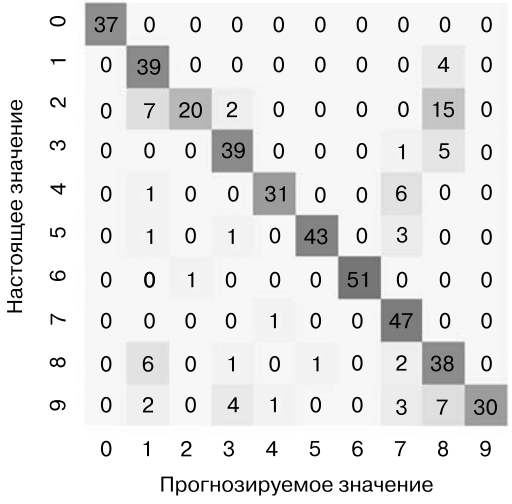

Рис. 5.20. Матрица различий, демонстрирующая частоты ошибочных классификаций нашего классификатора

Этот рисунок демонстрирует нам места, в которых наш классификатор склонен ошибаться, например, значительное количество двоек ошибочно классифицированы как единицы или восьмерки. Другой способ получения информации о характеристиках модели — построить график входных данных еще раз вместе с предсказанными метками. Мы будем использовать зеленый цвет для правильных меток, и красный — для ошибочных (рис. 5.21):

```
In[32]: fig, axes = plt.subplots(10, 10, figsize=(8, 8),
                                 subplot_kw={'xticks':[], 'yticks':[]},
                                 gridspec_kw=dict(hspace=0.1, wspace=0.1))

        for i, ax in enumerate(axes.flat):
            ax.imshow(digits.images[i], cmap='binary',
                      interpolation='nearest')
            ax.text(0.05, 0.05, str(y_model[i]),
                    transform=ax.transAxes,
                    color='green' if (ytest[i] == y_model[i]) else 'red')
```

Из этого подмножества данных можно почерпнуть полезную информацию относительно мест, в которых алгоритм работает неоптимально. Чтобы поднять нашу точность выше 80 %, можно воспользоваться более сложным алгоритмом, таким как метод опорных векторов (см. раздел «Заглянем глубже: метод опорных векторов» этой главы), случайные леса (см. раздел «Заглянем глубже: деревья принятия решений и случайные леса» данной главы) или другим методом классификации.

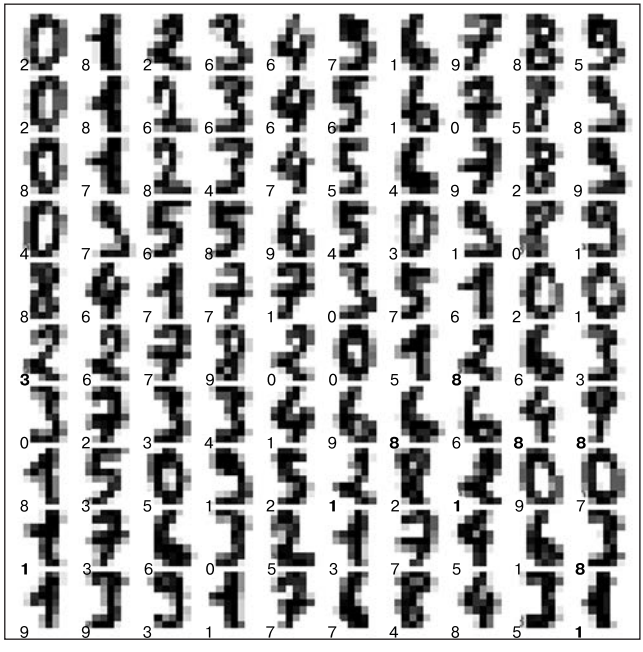

Рис. 5.21. Данные, показывающие правильные (зеленым цветом) и ошибочные (красным) метки. Цветную версию этого графика см. в онлайн-приложении (https://github.com/jakevdp/PythonDataScienceHandbook)

Резюме

В этом разделе мы рассмотрели основные возможности представления данных библиотеки Scikit-Learn, а также API статистического оценивания. Независимо от типа оценивателя применяется одна и та же схема: импорт/создание экземпляра/обучение/предсказание. Вооружившись этой информацией по API статистического оценивания, вы можете, изучив документацию библиотеки Scikit-Learn, начать экспериментировать, используя различные модели для своих данных.

В следующем разделе мы рассмотрим, вероятно, самый важный вопрос машинного обучения: выбор и проверку модели.

Гиперпараметры и проверка модели

В предыдущем разделе описана основная схема использования моделей машинного обучения с учителем.

1. Выбрать класс модели.
2. Выбрать гиперпараметры модели.

3. Обучить модель на данных обучающей последовательности.
4. Использовать модель, чтобы предсказать метки для новых данных.

Первые два пункта — выбор модели и выбор гиперпараметров — вероятно, важнее всего для эффективного использования этих инструментов и методов. Для оптимального их выбора необходим способ *проверки* того, что конкретная модель с конкретными гиперпараметрами хорошо аппроксимирует конкретные данные. Хотя эта задача может показаться несложной, в ней встречаются определенные подводные камни, которые необходимо обойти.

Соображения относительно проверки модели

В принципе, проверка модели очень проста: после выбора модели и гиперпараметров оценить ее эффективность можно, воспользовавшись ею для части обучающей последовательности и сравнив предсказания с известными значениями.

В следующих разделах я сначала продемонстрирую «наивный» подход к проверке модели и покажу, почему он не работает, прежде чем обратиться к анализу использования отложенных наборов данных и перекрестной проверки в целях более надежной оценки модели.

Плохой способ проверки модели

Посмотрим на «наивный» подход к проверке на наборе данных Iris, с которым мы работали в предыдущем разделе. Начнем с загрузки данных:

```
In[1]: from sklearn.datasets import load_iris
       iris = load_iris()
       X = iris.data
       y = iris.target
```

Выберем теперь модель и гиперпараметры. Мы будем использовать в этом примере классификатор на основе метода *k*-средних с `n_neighbors=1`. Это очень простая и интуитивно понятная модель, которую можно описать фразой «Метка для неизвестной точки такая же, как и метка ближайшей к ней обучающей точки»:

```
In[2]: from sklearn.neighbors import KNeighborsClassifier
       model = KNeighborsClassifier(n_neighbors=1)
```

Далее мы обучаем модель и используем ее для предсказания меток для уже известных данных:

```
In[3]: model.fit(X, y)
       y_model = model.predict(X)
```

Наконец, мы вычисляем долю правильно маркированных точек:

```
In[4]: from sklearn.metrics import accuracy_score
       accuracy_score(y, y_model)

Out[4]: 1.0
```

Как видим, показатель точности равен 1.0, то есть наша модель правильно пометила 100 % точек! Но действительно ли это правильная оценка ожидаемой точности? Действительно ли нам попалась модель, которая будет работать правильно в 100 % случаев?

Как вы могли догадаться, ответ на этот вопрос — нет. На самом деле этот подход имеет фундаментальный изъян: *обучение и оценка модели выполняются на одних и тех же данных*! Более того, модель ближайшего соседа — оцениватель, работающий путем *обучения на примерах* (instance-based estimator), попросту сохраняющий обучающие данные и предсказывающий метки путем сравнения новых данных с этими сохраненными точками. За исключением некоторых специально сконструированных случаев его точность будет *всегда* равна 100 %!

Хороший способ проверки модели: отложенные данные

Для более точного выяснения рабочих характеристик модели воспользуемся так называемыми *отложенными наборами данных* (holdout sets), то есть отложим некоторое подмножество данных из обучающей последовательности модели, после чего используем его для проверки качества работы модели. Это разделение в Scikit-Learn можно произвести с помощью утилиты `train_test_split`:

```
In[5]: from sklearn.cross_validation import train_test_split

       # Разделяем данные: по 50% в каждом из наборов
       X1, X2, y1, y2 = train_test_split(X, y, random_state=0,
                                         train_size=0.5)

       # Обучаем модель на одном из наборов данных
       model.fit(X1, y1)

       # Оцениваем работу модели на другом наборе
       y2_model = model.predict(X2)
       accuracy_score(y2, y2_model)

Out[5]: 0.90666666666666662
```

Мы получили более логичный результат: классификатор на основе метода ближайшего соседа демонстрирует точность около 90 % на этом отложенном наборе данных. Отложенный набор данных схож с неизвестными данными, поскольку модель «не видела» их ранее.

Перекрестная проверка модели

Потеря части наших данных для обучения модели — один из недостатков использования отложенного набора данных для проверки модели. В предыдущем случае половина набора данных не участвует в обучении модели! Это неоптимально и может стать причиной проблем, особенно если исходный набор данных невелик.

Один из способов решения этой проблемы — *перекрестная проверка* (cross-validation), то есть выполнение последовательности аппроксимаций, в которых каждое подмножество данных используется как в качестве обучающей последовательности, так и проверочного набора. Наглядно его можно представить в виде рис. 5.22.

Рис. 5.22. Визуализация двухблочной перекрестной проверки

Мы выполняем две попытки проверки, попеременно используя каждую половину данных в качестве отложенного набора данных. Воспользовавшись полученными выше разделенными данными, мы можем реализовать это следующим образом:

```
In[6]: y2_model = model.fit(X1, y1).predict(X2)
       y1_model = model.fit(X2, y2).predict(X1)
       accuracy_score(y1, y1_model), accuracy_score(y2, y2_model)

Out[6]: (0.95999999999999996, 0.90666666666666662)
```

Полученные числа — две оценки точности, которые можно обобщить (допустим, взяв от них среднее значение) для получения лучшей меры общей работы модели. Этот конкретный вид перекрестной проверки, в которой мы разбили данные на два набора и по очереди использовали каждый из них в качестве проверочного набора, называется *двухблочной перекрестной проверкой* (two-fold cross-validation).

Можно распространить эту идею на случай большего числа попыток и большего количества блоков данных, например, на рис. 5.23 представлена пятиблочная перекрестная проверка.

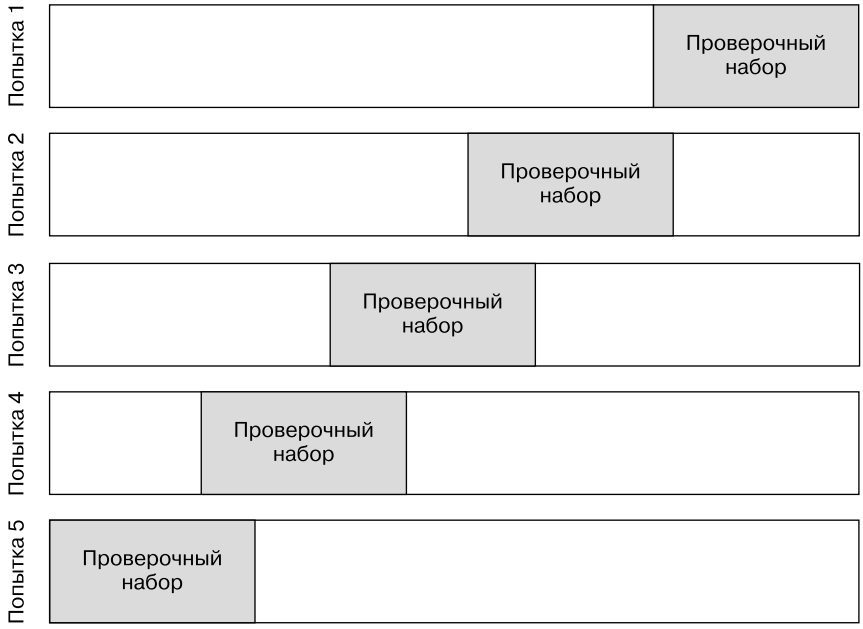

Рис. 5.23. Визуализация пятиблочной перекрестной проверки

В этом случае мы разбиваем данные на пять групп и по очереди используем каждую из них для оценки обучения модели на остальных 4/5 данных. Делать это вручную довольно утомительно, так что воспользуемся удобной утилитой `cross_val_score` библиотеки Scikit-Learn для большей краткости синтаксиса:

```
In[7]: from sklearn.cross_validation import cross_val_score
       cross_val_score(model, X, y, cv=5)

Out[7]: array([ 0.96666667, 0.96666667, 0.93333333, 0.93333333,  1.        ])
```

Повторение проверки по различным подмножествам данных дает нам лучшее представление о качестве работы алгоритма.

Библиотека Scikit-Learn реализует множество схем перекрестной проверки, удобных в определенных конкретных ситуациях. Они реализованы с помощью итераторов в модуле `cross_validation`. Например, мы взяли предельный случай, при котором количество блоков равно количеству точек данных, и обучаем модель в каждой попытке на всех точках, кроме одной. Такой тип перекрестной проверки известен под названием перекрестной проверки *по отдельным объектам* (leave-one-out cross-validation — дословно «перекрестная проверка по всем без одного»), ее можно использовать следующим образом:

```
In[8]: from sklearn.cross_validation import LeaveOneOut
       scores = cross_val_score(model, X, y, cv=LeaveOneOut(len(X)))
       scores

Out[8]: array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
                1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
                1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
                1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
                1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
                1.,  1.,  1.,  1.,  1.,  0.,  1.,  0.,  1.,  1.,  1.,  1.,  1.,
                1.,  1.,  1.,  1.,  1.,  0.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
                1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
                1.,  1.,  0.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
                1.,  1.,  0.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
                1.,  1.,  1.,  0.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
                1.,  1.,  1.,  1.,  1.,  1.,  1.])
```

Перекрестная проверка по отдельным объектам выдает результаты по всем 150 попыткам (ведь у нас 150 выборок), причем каждый результат показывает, правильным (`1.0`) или неправильным (`0.0`) было предсказание. Среднее значение этих результатов дает нам общую оценку частоты ошибок:

```
In[9]: scores.mean()

Out[9]: 0.95999999999999996
```

Аналогичным образом можно выполнять и другие схемы перекрестной проверки. Чтобы узнать, какие еще их варианты есть в библиотеке Scikit-Learn, воспользуйтесь оболочкой IPython для просмотра подмодуля `sklearn.cross_validation` или обратитесь к онлайн-документации по перекрестной проверке библиотеки Scikit-Learn.

Выбор оптимальной модели

Теперь, когда мы познакомились с основными идеями простой и перекрестной проверки, можно немного углубиться в вопросы выбора модели и гиперпараметров. Данная тема представляет собой один из наиболее важных аспектов машинного обучения, причем эти вопросы зачастую рассматриваются лишь вскользь во вводных курсах машинного обучения.

Важнейшим является следующим вопрос: *что делать, если наш оцениватель показывает недостаточно хорошие результаты?* Существует несколько возможных ответов:

- использовать более сложную/гибкую модель;
- применять менее сложную/гибкую модель;

- собрать больше выборок для обучения;
- собрать больше данных для добавления новых признаков к каждой заданной выборке.

Ответ на этот вопрос зачастую парадоксален. В частности, иногда использование более сложной модели приводит к худшим результатам, а добавление новых выборок для обучения не приводит к их улучшению! Успешных специалистов-практиков в области машинного обучения как раз и отличает умение определять, какие действия улучшат характеристики модели.

Компромисс между систематической ошибкой и дисперсией

По существу, выбор «оптимальной модели» состоит в поиске наилучшего компромисса между *систематической ошибкой* (bias) и *дисперсией* (variance). Рассмотрим рис. 5.24, на котором представлены два случая регрессионной аппроксимации одного набора данных.

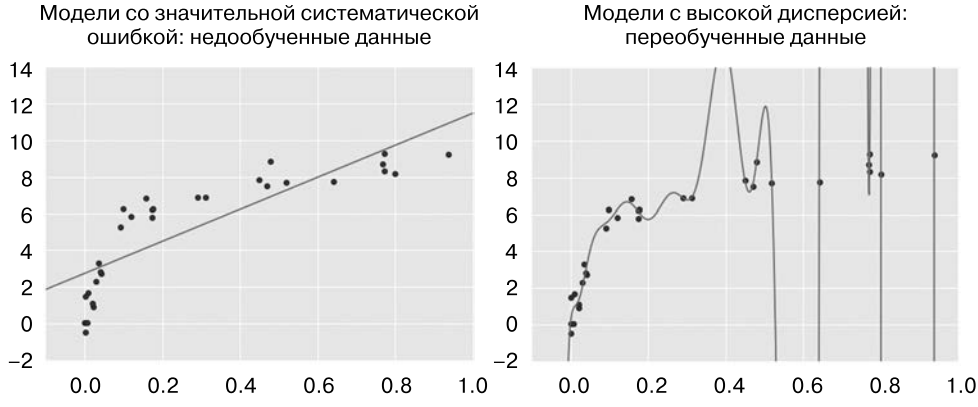

Рис. 5.24. Модели регрессии со значительной систематической ошибкой и высокой дисперсией

Очевидно, что обе модели не слишком хорошо аппроксимируют наши данные, но проблемы с ними различны.

Приведенная слева модель пытается найти прямолинейное приближение к данным. Но в силу того, что внутренняя структура данных сложнее прямой линии, с помощью прямолинейной модели невозможно описать этот набор данных достаточно хорошо. О подобной модели говорят, что она *недообучена* (underfit), то есть гибкость модели недостаточна для удовлетворительного учета всех признаков в данных. Другими словами, у этой модели имеется значительная *систематическая ошибка*.

Приведенная справа модель пытается подобрать для наших данных многочлен высокой степени. В этом случае модель достаточно гибка, чтобы практически идеально соответствовать всем нюансам данных, но хотя она и описывает очень точно обучающую последовательность, конкретная ее форма отражает скорее характеристики шума в данных, чем внутренние свойства процесса, сгенерировавшего данные. О подобной модели говорят, что она *переобучена* (overfit), то есть гибкость модели такова, что в конце концов модель учитывает не только исходное распределение данных, но и случайные ошибки в них. Другими словами, у этой модели имеется высокая дисперсия.

Чтобы взглянуть с другой стороны, посмотрим, что получится, если воспользоваться этими двумя моделями для предсказания y-значения для каких-либо новых данных. В диаграммах на рис. 5.25 красные (более светлые в печатном варианте книги) точки обозначают исключенные из обучающей последовательности данные.

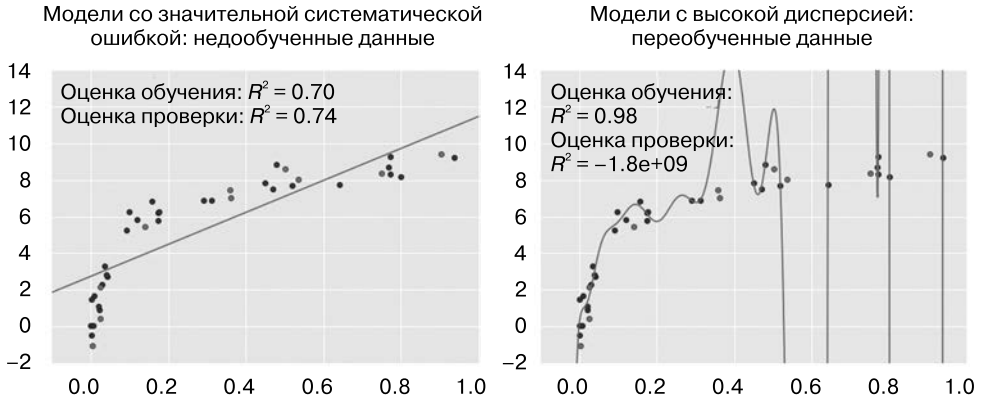

Рис. 5.25. Оценки эффективности модели для обучения и проверки для моделей со значительной систематической ошибкой и высокой дисперсией

В качестве оценки эффективности здесь используется R^2 — *коэффициент детерминации* или *коэффициент смешанной корреляции* (подробнее о нем можно прочитать по адресу https://ru.wikipedia.org/wiki/Коэффициент_детерминации). Он представляет собой меру того, насколько хорошо модель работает по сравнению с простым средним значением целевых величин. $R^2 = 1$ означает идеальное совпадение предсказаний, а $R^2 = 0$ показывает, что модель оказалась ничем не лучше простого среднего значения данных, а отрицательные значения указывают на модели, которые работают еще хуже.

На основе оценок эффективности вышеприведенных двух моделей мы можем сделать следующее обобщенное наблюдение.

- Для моделей со значительной систематической ошибкой эффективность модели на проверочном наборе данных сопоставима с ее эффективностью на обучающей последовательности.
- Для моделей с высокой дисперсией эффективность модели на проверочном наборе данных существенно хуже ее эффективности на обучающей последовательности.

Если бы мы умели управлять сложностью модели, то оценки эффективности для обучения и проверки вели бы себя так, как показано на рис. 5.26.

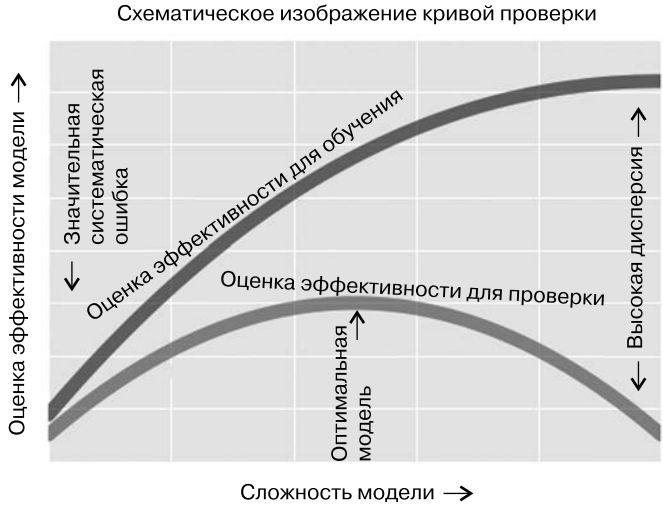

Рис. 5.26. Схематическое изображение зависимости между сложностью модели и оценками эффективности для обучения и проверки

Показанный на рис. 5.26 график часто называют *кривой проверки* (validation curve). На нем можно наблюдать следующие важные особенности.

- Оценка эффективности для обучения всегда превышает оценку эффективности для проверки. Это логично: модель лучше подходит для уже виденных ею данных, чем для тех, которые она еще не видела.
- Модели с очень низкой сложностью (со значительной систематической ошибкой) являются недообученными, то есть эти модели будут плохо предсказывать как данные обучающей последовательности, так и любые ранее не виденные ими данные.
- Модели с очень высокой сложностью (с высокой дисперсией) являются переобученными, то есть будут очень хорошо предсказывать данные обучающей последовательности, но на любых ранее не виденных данных работать очень плохо.

- Кривая проверки достигает максимума в какой-то промежуточной точке. Этот уровень сложности означает приемлемый компромисс между систематической ошибкой и дисперсией.

Средства регулирования сложности модели различаются в зависимости от модели. Как осуществлять подобную регулировку для каждой из моделей, мы увидим в следующих разделах, когда будем обсуждать конкретные модели подробнее.

Кривые проверки в библиотеке Scikit-Learn

Рассмотрим пример перекрестной проверки для расчета кривой проверки для класса моделей. Мы будем использовать *модель полиномиальной регрессии* (polynomial regression model): это обобщенная линейная модель с параметризованной степенью многочлена. Например, многочлен 1-й степени аппроксимирует наши данные прямой линией; при параметрах модели a и b:

$$y = ax + b.$$

Многочлен 3-й степени аппроксимирует наши данные кубической кривой; при параметрах модели a, b, c, d:

$$y = ax^3 + bx^2 + cx + d.$$

Это можно обобщить на любое количество полиномиальных признаков. В библиотеке Scikit-Learn реализовать это можно с помощью простой линейной регрессии в сочетании с полиномиальным препроцессором. Мы воспользуемся *конвейером* (pipeline) для соединения этих операций в единую цепочку (мы обсудим полиномиальные признаки и конвейеры подробнее в разделе «Проектирование признаков» данной главы):

```
In[10]: from sklearn.preprocessing import PolynomialFeatures
        from sklearn.linear_model import LinearRegression
        from sklearn.pipeline import make_pipeline

        def PolynomialRegression(degree=2, **kwargs):
            return make_pipeline(PolynomialFeatures(degree),
                                 LinearRegression(**kwargs))
```

Теперь создадим данные, на которых будем обучать нашу модель:

```
In[11]: import numpy as np

        def make_data(N, err=1.0, rseed=1):
            # Создаем случайные выборки данных
            rng = np.random.RandomState(rseed)
            X = rng.rand(N, 1) ** 2
            y = 10 - 1. / (X.ravel() + 0.1)
            if err > 0:
```

```
            y += err * rng.randn(N)
        return X, y

    X, y = make_data(40)
```

Визуализируем наши данные вместе с несколькими аппроксимациями их многочленами различной степени (рис. 5.27):

```
In[12]: %matplotlib inline
        import matplotlib.pyplot as plt
        import seaborn; seaborn.set()  # plot formatting

        X_test = np.linspace(-0.1, 1.1, 500)[:, None]

        plt.scatter(X.ravel(), y, color='black')
        axis = plt.axis()
        for degree in [1, 3, 5]:
            y_test = PolynomialRegression(degree).fit(X, y).predict(X_test)
            plt.plot(X_test.ravel(), y_test, label='degree={0}'.format(degree))
        plt.xlim(-0.1, 1.0)
        plt.ylim(-2, 12)
        plt.legend(loc='best');
```

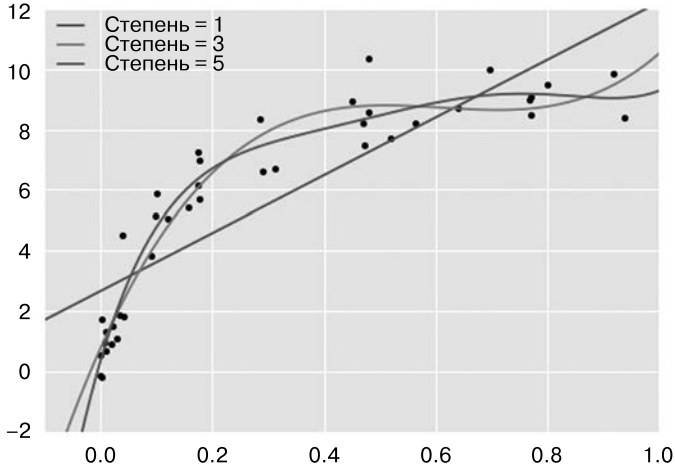

Рис. 5.27. Аппроксимации набора данных тремя различными полиномиальными моделями

Параметром, служащим для управления сложностью модели, в данном случае является степень многочлена, которая может быть любым неотрицательным числом. Не помешает задать себе вопрос: какая степень многочлена обеспечивает подходящий компромисс между систематической ошибкой (недообучение) и дисперсией (переобучение)?

Чтобы решить этот вопрос, визуализируем кривую проверки для этих конкретных данных и моделей. Проще всего сделать это с помощью предоставляемой библиотекой Scikit-Learn удобной утилиты `validation_curve`. Эта функция, получив на входе модель, данные, название параметра и диапазон для анализа, автоматически вычисляет в этом диапазоне значение как оценки эффективности для обучения, так и оценки эффективности для проверки (рис. 5.28):

```
In[13]:
from sklearn.learning_curve import validation_curve
degree = np.arange(0, 21)
train_score, val_score = validation_curve(PolynomialRegression(), X, y,
                                          'polynomialfeatures__degree',
                                          degree, cv=7)

plt.plot(degree, np.median(train_score, 1), color='blue',
                         label='training score')   # Оценка обучения
plt.plot(degree, np.median(val_score, 1), color='red',
                         label='validation score') # Оценка проверки
plt.legend(loc='best')
plt.ylim(0, 1)
plt.xlabel('degree') # Степень
plt.ylabel('score'); # Оценка
```

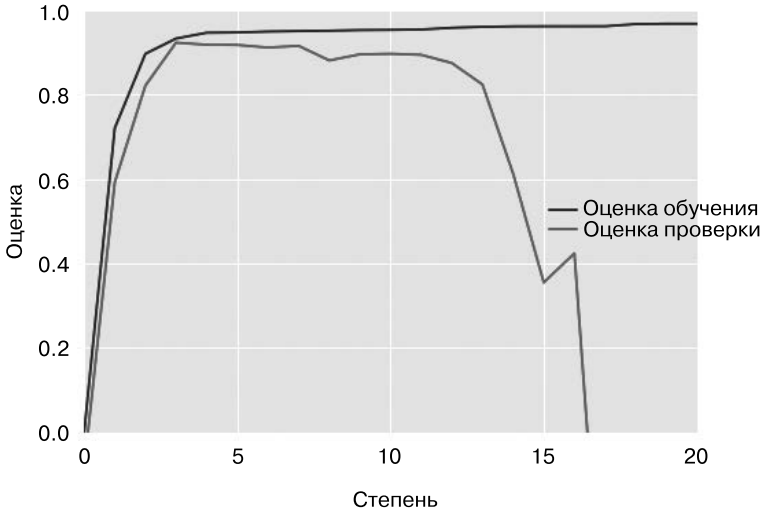

Рис. 5.28. Кривая проверки для приведенных на рис. 5.27 данных (ср.: рис. 5.26)

Этот график в точности демонстрирует ожидаемое нами качественное поведение: оценка эффективности для обучения на всем диапазоне превышает оценку эффективности для проверки; оценка эффективности для обучения монотонно растет с ростом сложности модели, а оценка эффективности для проверки

достигает максимума перед резким спадом в точке, где модель становится переобученной.

Как можно понять из приведенной кривой проверки, оптимальный компромисс между систематической ошибкой и дисперсией достигается для многочлена третьей степени. Вычислить и отобразить на графике эту аппроксимацию на исходных данных можно следующим образом (рис. 5.29):

```
In[14]: plt.scatter(X.ravel(), y)
        lim = plt.axis()
        y_test = PolynomialRegression(3).fit(X, y).predict(X_test)
        plt.plot(X_test.ravel(), y_test);
        plt.axis(lim);
```

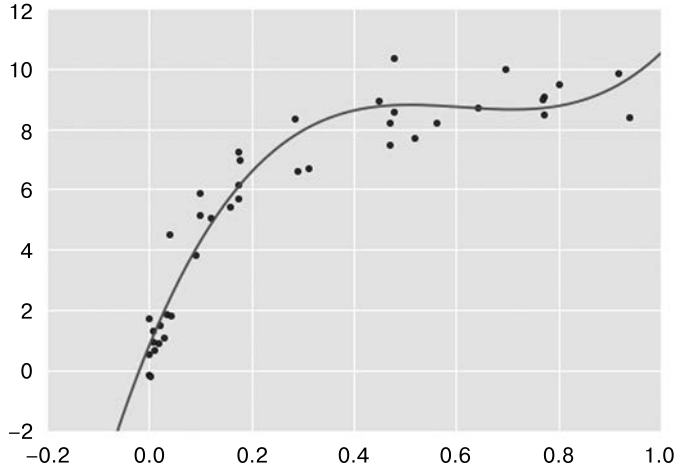

Рис. 5.29. Перекрестно-проверенная оптимальная модель для приведенных на рис. 5.27 данных

Отмечу, что для нахождения оптимальной модели не требуется вычислять оценку эффективности для обучения, но изучение зависимости между оценками эффективности для обучения и проверки дает нам полезную информацию относительно эффективности модели.

Кривые обучения

Важный нюанс сложности моделей состоит в том, что оптимальность модели обычно зависит от размера обучающей последовательности. Например, сгенерируем новый набор данных с количеством точек в пять раз больше (рис. 5.30):

```
In[15]: X2, y2 = make_data(200)
        plt.scatter(X2.ravel(), y2);
```

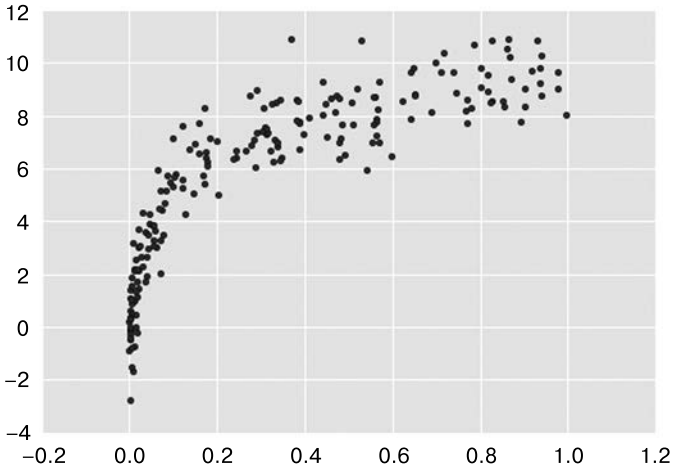

Рис. 5.30. Данные для демонстрации кривых обучения

Повторим вышеприведенный код для построения графика кривой обучения для этого большего набора данных. Для сравнения выведем поверх и предыдущие результаты (рис. 5.31):

```
In[16]:
degree = np.arange(21)
train_score2, val_score2 = validation_curve(PolynomialRegression(), X2, y2,
                                            'polynomialfeatures__degree',
                                            degree, cv=7)

plt.plot(degree, np.median(train_score2, 1), color='blue',
         label='training score')
plt.plot(degree, np.median(val_score2, 1), color='red',
                          label='validation score')
plt.plot(degree, np.median(train_score, 1), color='blue', alpha=0.3,
         linestyle='dashed')
plt.plot(degree, np.median(val_score, 1), color='red', alpha=0.3,
         linestyle='dashed')
plt.legend(loc='lower center')
plt.ylim(0, 1)
plt.xlabel('degree')
plt.ylabel('score');
```

Сплошные линии показывают новые результаты, а более бледные штриховые линии — результаты предыдущего меньшего набора данных. Из кривой проверки ясно, что этот больший набор данных позволяет использовать намного более сложную модель: максимум, вероятно, возле степени 6, но даже модель со степенью 20 не выглядит сильно переобученной — оценки эффективности для проверки и обучения остаются очень близки друг к другу.

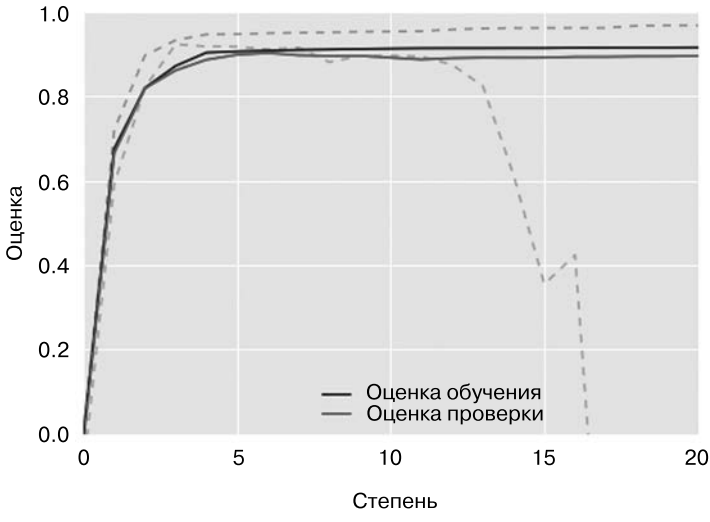

Рис. 5.31. Кривые обучения для аппроксимации полиномиальной моделью приведенных на рис. 5.30 данных

Таким образом, мы видим, что поведение кривой проверки зависит не от одного, а от двух важных факторов: сложности модели и количества точек обучения. Зачастую бывает полезно исследовать поведение модели как функции от количества точек обучения. Сделать это можно путем использования постепенно увеличивающихся подмножеств данных для обучения модели. График оценок для обучения/проверки с учетом размера обучающей последовательности известен под названием *кривой обучения* (learning curve).

Поведение кривой обучения должно быть следующим.

- Модель заданной сложности окажется *переобученной* на слишком маленьком наборе данных. Это значит, что оценка эффективности для обучения будет относительно высокой, а оценка эффективности для проверки — относительно низкой.
- Модель заданной сложности окажется *недообученной* на слишком большом наборе данных. Это значит, что оценка эффективности для обучения будет снижаться, а оценка эффективности для проверки — повышаться по мере роста размера набора данных.
- Модель никогда, разве что случайно, не покажет на проверочном наборе лучший результат, чем на обучающей последовательности. Это значит, что кривые будут сближаться, но никогда не пересекутся.

Учитывая эти особенности, можно ожидать, что кривая обучения будет выглядеть качественно схожей с изображенной на рис. 5.32.

Заметная особенность кривой обучения — сходимость к конкретному значению оценки при росте числа обучающих выборок. В частности, если количество точек достигло значения, при котором данная конкретная модель сошлась, то *добавление новых обучающих данных не поможет*! Единственным способом улучшить качество модели в этом случае будет использование другой (зачастую более сложной) модели.

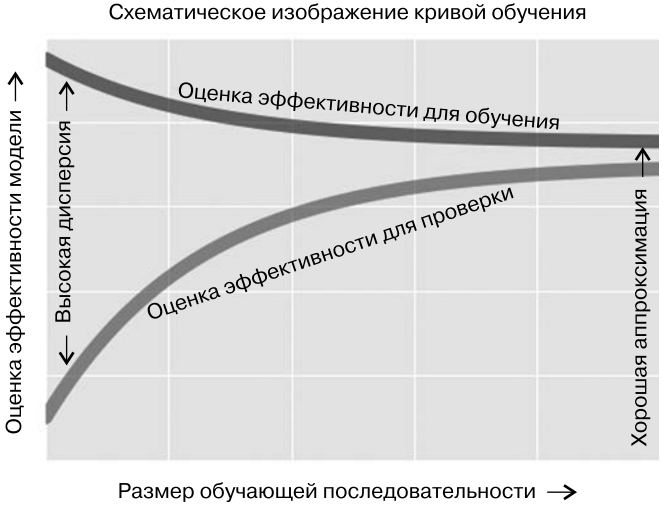

Рис. 5.32. Схематическое изображение типичной кривой обучения

Кривые обучения в библиотеке Scikit-Learn. Библиотека Scikit-Learn предоставляет удобные утилиты для вычисления кривых обучения для моделей. В этом разделе мы вычислим кривую обучения для нашего исходного набора данных с полиномиальными моделями второй и девятой степени (рис. 5.33):

```
In[17]:
from sklearn.learning_curve import learning_curve

fig, ax = plt.subplots(1, 2, figsize=(16, 6))
fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)

for i, degree in enumerate([2, 9]):
    N, train_lc, val_lc = learning_curve(PolynomialRegression(degree),
                                         X, y, cv=7,
                                         train_sizes=np.linspace(0.3, 1, 25))

    ax[i].plot(N, np.mean(train_lc, 1), color='blue', label='training score')
    ax[i].plot(N, np.mean(val_lc, 1), color='red', label='validation score')
    ax[i].hlines(np.mean([train_lc[-1], val_lc[-1]]), N[0], N[-1],
```

```
                color='gray',
                linestyle='dashed')
    ax[i].set_ylim(0, 1)
    ax[i].set_xlim(N[0], N[-1])
    ax[i].set_xlabel('training size') # Размерность обучения
    ax[i].set_ylabel('score')
    ax[i].set_title('degree = {0}'.format(degree), size=14)
    ax[i].legend(loc='best')
```

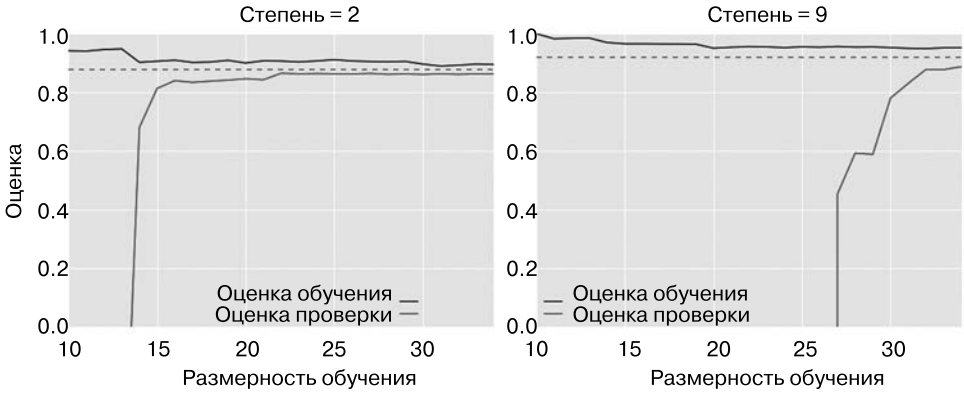

Рис. 5.33. Кривые обучения для модели низкой *(слева)* и высокой сложности *(справа)*

Это ценный показатель, поскольку он наглядно демонстрирует нам реакцию нашей модели на увеличение объема обучающих данных. В частности, после того момента, когда кривая обучения уже сошлась к какому-то значению (то есть когда кривые обучения и проверки уже близки друг к другу), *добавление дополнительных обучающих данных не улучшит аппроксимацию существенно*! Эта ситуация отражена на левом рисунке с кривой обучения для модели второй степени.

Единственный способ улучшения оценки уже сошедшейся кривой — использовать другую (обычно более сложную) модель. Это видно на правом рисунке: перейдя к более сложной модели, мы улучшаем оценку для точки сходимости (отмеченную штриховой линией) за счет более высокой дисперсии модели (соответствующей расстоянию между оценками эффективности для обучения и проверки). Если бы нам пришлось добавить еще больше точек, кривая обучения для более сложной из этих моделей все равно в итоге бы сошлась.

Построение графика кривой обучения для конкретных модели и набора данных облегчает принятие решения о том, как продвинуться еще дальше на пути улучшения анализа данных.

Проверка на практике: поиск по сетке

Из предшествующего обсуждения вы должны были понять смысл компромисса между систематической ошибкой и дисперсией и его зависимость от сложности модели и размера обучающей последовательности. На практике у моделей обычно больше одного параметра, поэтому графики кривых проверки и обучения превращаются из двумерных линий в многомерные поверхности. Выполнение подобных визуализаций в таких случаях представляет собой непростую задачу, поэтому лучше отыскать конкретную модель, при которой оценка эффективности для проверки достигает максимума.

Библиотека Scikit-Learn предоставляет для этой цели специальные автоматические инструменты, содержащиеся в модуле `grid_search`. Рассмотрим трехмерную сетку признаков модели — степени многочлена, флага, указывающего, нужно ли подбирать точку пересечения с осью координат, и флага, указывающего, следует ли выполнять нормализацию. Выполнить эти настройки можно с помощью метаоценивателя `GridSearchCV` библиотеки Scikit-Learn:

```
In[18]: from sklearn.grid_search import GridSearchCV

        param_grid = {'polynomialfeatures__degree': np.arange(21),
                      'linearregression__fit_intercept': [True, False],
                      'linearregression__normalize': [True, False]}

        grid = GridSearchCV(PolynomialRegression(), param_grid, cv=7)
```

Отмечу, что, как и обычный оцениватель, он еще не был применен к каким-либо данным. Обучение модели наряду с отслеживанием промежуточных оценок эффективности в каждой из точек сетки производится путем вызова метода `fit()`:

```
In[19]: grid.fit(X, y);
```

После обучения можно узнать значения оптимальных параметров:

```
In[20]: grid.best_params_


Out[20]: {'linearregression__fit_intercept': False,
          'linearregression__normalize': True,
          'polynomialfeatures__degree': 4}
```

При необходимости можно воспользоваться этой оптимальной моделью и продемонстрировать аппроксимацию с помощью уже виденного вами выше кода (рис. 5.34):

```
In[21]: model = grid.best_estimator_

        plt.scatter(X.ravel(), y)
        lim = plt.axis()
```

```
y_test = model.fit(X, y).predict(X_test)
plt.plot(X_test.ravel(), y_test, hold=True);
plt.axis(lim);
```

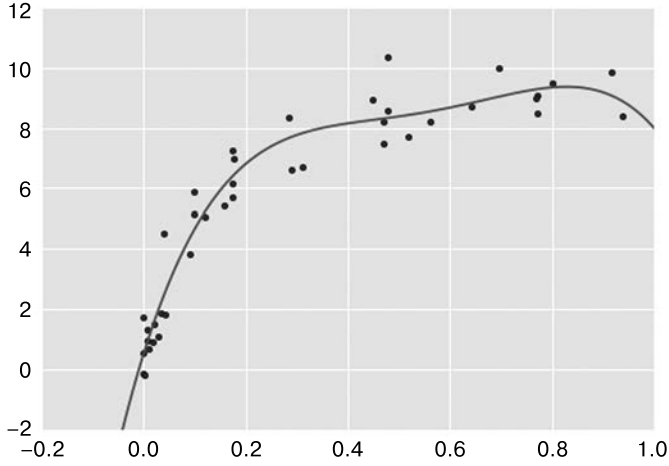

Рис. 5.34. Оптимальная модель, определенная посредством автоматического поиска по сетке

У поиска по сетке имеется множество опций, включая возможности задания пользовательской функции оценки эффективности, распараллеливания вычислений, выполнения случайного поиска и др. Для получения дополнительной информации см. примеры в разделах «Заглянем глубже: ядерная оценка плотности распределения» и «Прикладная задача: конвейер распознавания лиц» данной главы или обратитесь к документации библиотеки Scikit-Learn, посвященной поиску по сетке (http://scikit-learn.org/stable/modules/grid_search.html).

Резюме

В этом разделе мы приступили к изучению понятий проверки модели и оптимизации гиперпараметров, фокусируя внимание на наглядных аспектах компромисса между систематической ошибкой и дисперсией и его работы при подгонке моделей к данным. В частности, мы обнаружили, что *крайне важно* использовать проверочный набор или перекрестную проверку при настройке параметров, чтобы избежать переобучения более сложных/гибких моделей.

В следующих разделах мы детально рассмотрим некоторые особенно удобные модели, попутно обсуждая имеющиеся возможности их настройки и влияние этих свободных параметров на сложность модели. Не забывайте уроки этого раздела при дальнейшем чтении книги и изучении упомянутых методов машинного обучения!

Проектирование признаков

В предыдущих разделах мы обрисовали базовые понятия машинного обучения, но во всех предполагалось, что наши данные находятся в аккуратном формате `[n_samples, n_features]`. На практике же данные редко поступают к нам в подобном виде. Поэтому одним из важнейших этапов использования машинного обучения на практике становится *проектирование признаков* (feature engineering), то есть преобразование всей касающейся задачи информации в числа, пригодные для построения матрицы признаков.

В этом разделе мы рассмотрим несколько часто встречающихся примеров задач проектирования признаков: признаки для представления *категориальных данных* (categorical data), признаки для представления *текста* и признаки для представления *изображений*. Кроме того, мы обсудим использование *производных признаков* (derived features) для повышения сложности модели и заполнение отсутствующих данных. Этот процесс часто называют *векторизацией*, так как он включает преобразование данных в произвольной форме в аккуратные векторы.

Категориальные признаки

Категориальные данные — один из распространенных типов нечисловых данных. Например, допустим, что мы анализируем какие-то данные по ценам на жилье и, помимо числовых признаков, таких как «цена» и «количество комнат», в них имеется информация о микрорайоне (neighborhood). Например, пусть наши данные выглядят следующим образом:

```
In[1]: data = [
           {'price': 850000, 'rooms': 4, 'neighborhood': 'Queen Anne'},
           {'price': 700000, 'rooms': 3, 'neighborhood': 'Fremont'},
           {'price': 650000, 'rooms': 3, 'neighborhood': 'Wallingford'},
           {'price': 600000, 'rooms': 2, 'neighborhood': 'Fremont'}
       ]
```

Вам может показаться соблазнительным кодировать эти данные с помощью задания прямого числового отображения:

```
In[2]: {'Queen Anne': 1, 'Fremont': 2, 'Wallingford': 3};
```

Но оказывается, что в библиотеке Scikit-Learn такой подход не очень удобен: модели данного пакета исходят из базового допущения о том, что числовые признаки отражают алгебраические величины. Следовательно, подобное отображение будет подразумевать, например, что *Queen Anne* < *Fremont* < *Wallingford* или даже что *Wallingford* – *Queen Anne* = *Fremont*, что, не считая сомнительных демографических шуток, не имеет никакого смысла.

Испытанным методом для такого случая является *прямое кодирование* (one-hot encoding), означающее создание дополнительных столбцов-индикаторов наличия/отсутствия категории с помощью значений 1 или 0 соответственно. При наличии данных в виде списка словарей для этой цели можно воспользоваться утилитой `DictVectorizer` библиотеки Scikit-Learn:

```
In[3]: from sklearn.feature_extraction import DictVectorizer
       vec = DictVectorizer(sparse=False, dtype=int)
       vec.fit_transform(data)

Out[3]: array([[     0,      1,      0, 850000,      4],
               [     1,      0,      0, 700000,      3],
               [     0,      0,      1, 650000,      3],
               [     1,      0,      0, 600000,      2]], dtype=int64)
```

Обратите внимание, что столбец *neighborhood* превратился в три отдельных столбца, отражающих три метки микрорайонов, и что в каждой строке стоит 1 в соответствующем ее микрорайону столбце. После подобного кодирования категориальных признаков можно продолжить обучение модели Scikit-Learn обычным образом.

Чтобы узнать, что означает каждый столбец, можно посмотреть названия признаков:

```
In[4]: vec.get_feature_names()

Out[4]: ['neighborhood=Fremont',
         'neighborhood=Queen Anne',
         'neighborhood=Wallingford',
         'price',
         'rooms']
```

У этого подхода имеется один очевидный недостаток: если количество значений категории велико, размер набора данных может *значительно* вырасти. Однако поскольку кодированные данные состоят в основном из нулей, эффективным решением будет разреженный формат вывода:

```
In[5]: vec = DictVectorizer(sparse=True, dtype=int)
       vec.fit_transform(data)

Out[5]: <4x5 sparse matrix of type '<class 'numpy.int64'>'
            with 12 stored elements in Compressed Sparse Row format>
```

Многие (хотя пока не все) оцениватели библиотеки Scikit-Learn допускают передачу им подобных разреженных входных данных при обучении и оценке моделей. Для поддержки подобного кодирования библиотека Scikit-Learn включает две дополнительные утилиты: `sklearn.preprocessing.OneHotEncoder` и `sklearn.feature_extraction.FeatureHasher`.

Текстовые признаки

При проектировании признаков часто требуется преобразовывать текст в набор репрезентативных числовых значений. Например, в основе наиболее автоматических процедур извлечения данных социальных медиа лежит определенный вид кодирования текста числовыми значениями. Один из простейших методов кодирования данных — по *количеству слов*: для каждого фрагмента текста подсчитывается количество вхождений в него каждого из слов, после чего результаты помещаются в таблицу.

Рассмотрим следующий набор из трех фраз:

```
In[6]: sample = ['problem of evil',
                 'evil queen',
                 'horizon problem']
```

Для векторизации этих данных на основе числа слов можно создать столбцы, соответствующие словам problem, evil, horizon и т. д. Хотя это можно сделать вручную, мы избежим нудной работы, воспользовавшись утилитой `CountVectorizer` библиотеки Scikit-Learn:

```
In[7]: from sklearn.feature_extraction.text import CountVectorizer
       vec = CountVectorizer()
       X = vec.fit_transform(sample)
       X

Out[7]: <3x5 sparse matrix of type '<class 'numpy.int64'>'
            with 7 stored elements in Compressed Sparse Row format>
```

Результат представляет собой разреженную матрицу, содержащую количество вхождений каждого из слов. Для удобства мы преобразуем ее в объект `DataFrame` с маркированными столбцами:

```
In[8]: import pandas as pd
       pd.DataFrame(X.toarray(), columns=vec.get_feature_names())

Out[8]:    evil  horizon  of  problem  queen
        0     1        0   1        1      0
        1     1        0   0        0      1
        2     0        1   0        1      0
```

У этого подхода существуют проблемы: использование непосредственно количеств слов ведет к признакам, с которыми встречающимся очень часто словам придается слишком большое значение, а это в некоторых алгоритмах классификации может оказаться субоптимальным. Один из подходов к решению этой проблемы известен под названием *«терма-обратная частотность документа»* (term frequency-inverse document frequency) или TF-IDF. При нем слова получают вес с учетом

частоты их появления во всех документах. Синтаксис вычисления этих признаков аналогичен предыдущему примеру:

```
In[9]: from sklearn.feature_extraction.text import TfidfVectorizer
       vec = TfidfVectorizer()
       X = vec.fit_transform(sample)
       pd.DataFrame(X.toarray(), columns=vec.get_feature_names())

Out[9]:        evil   horizon        of   problem     queen
        0  0.517856  0.000000  0.680919  0.517856  0.000000
        1  0.605349  0.000000  0.000000  0.000000  0.795961
        2  0.000000  0.795961  0.000000  0.605349  0.000000
```

Чтобы увидеть пример использования TF-IDF в задачах классификации, см. раздел «Заглянем глубже: наивная байесовская классификация» данной главы.

Признаки для изображений

Достаточно часто для задач машинного обучения требуется соответствующим образом кодировать *изображения*. Простейший подход — тот, который мы использовали для набора данных рукописных цифр в разделе «Знакомство с библиотекой Scikit-Learn» этой главы, — использовать значения интенсивности самих пикселов. Но подобные подходы могут, в зависимости от прикладной задачи, оказаться неоптимальными.

Всесторонний обзор методов выделения признаков для изображений выходит далеко за рамки данного раздела, но вы можете найти отличные реализации множества стандартных подходов в проекте Scikit-Image (http://scikit-image.org/). Впрочем, один пример совместного использования библиотеки Scikit-Learn и пакета Scikit-Image вы можете найти в разделе «Прикладная задача: конвейер распознавания лиц» данной главы.

Производные признаки

Еще один удобный тип признаков — выведенные математически из каких-либо входных признаков. Мы уже встречались с ними в разделе «Гиперпараметры и проверка модели» этой главы, когда создавали *полиномиальные признаки* из входных данных. Мы видели, что можно преобразовать линейную регрессию в полиномиальную регрессию не путем изменения модели, а преобразования входных данных! Этот метод, известный под названием *регрессии по комбинации базисных функций* (basis function regression), рассматривается подробнее в разделе «Заглянем глубже: линейная регрессия» текущей главы.

Например, очевидно, что следующие данные нельзя адекватно описать с помощью прямой линии (рис. 5.35):

```
In[10]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt

        x = np.array([1, 2, 3, 4, 5])
        y = np.array([4, 2, 1, 3, 7])
        plt.scatter(x, y);
```

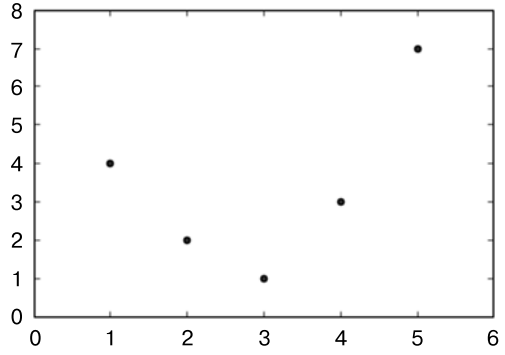

Рис. 5.35. Данные, которые нельзя хорошо описать с помощью прямой линии

Тем не менее мы можем подобрать разделяющую прямую для этих данных с помощью функции `LinearRegression` и получить оптимальный результат (рис. 5.36):

```
In[11]: from sklearn.linear_model import LinearRegression
        X = x[:, np.newaxis]
        model = LinearRegression().fit(X, y)
        yfit = model.predict(X)
        plt.scatter(x, y)
        plt.plot(x, yfit);
```

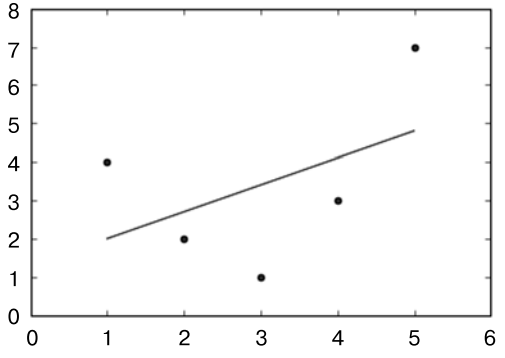

Рис. 5.36. Неудачная прямолинейная аппроксимация

Очевидно, что для описания зависимости между x и y нам требуется использовать более сложную модель. Сделать это можно путем преобразования данных, добавив дополнительные столбцы признаков для увеличения гибкости модели. Добавить в данные полиномиальные признаки можно следующим образом:

```
In[12]: from sklearn.preprocessing import PolynomialFeatures
        poly = PolynomialFeatures(degree=3, include_bias=False)
        X2 = poly.fit_transform(X)
        print(X2)
```

```
[[   1.    1.    1.]
 [   2.    4.    8.]
 [   3.    9.   27.]
 [   4.   16.   64.]
 [   5.   25.  125.]]
```

В матрице производных признаков один столбец соответствует x, второй — x^2, а третий — x^3. Расчет линейной регрессии для этих расширенных входных данных позволяет получить намного лучшую аппроксимацию (рис. 5.37):

```
In[13]: model = LinearRegression().fit(X2, y)
        yfit = model.predict(X2)
        plt.scatter(x, y)
        plt.plot(x, yfit);
```

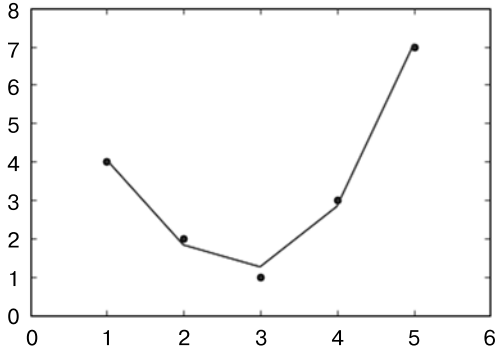

Рис. 5.37. Линейная аппроксимация по производным полиномиальным признакам

Идея улучшения модели путем не изменения самой модели, а преобразования входных данных является базовой для многих более продвинутых методов машинного обучения. Мы обсудим эту идею подробнее в разделе «Заглянем глубже: линейная регрессия» данной главы в контексте регрессии по комбинации базисных функций. В общем случае это путь к набору обладающих огромными возможностями методик, известных под названием *«ядерные методы»* (kernel

methods), которые мы рассмотрим в разделе «Заглянем глубже: метод опорных векторов» текущей главы.

Внесение отсутствующих данных

Еще одна часто встречающаяся задача в проектировании признаков — обработка отсутствующих данных. Мы уже обсуждали этот вопрос в объектах `DataFrame` в разделе «Обработка отсутствующих данных» главы 3 и видели, что отсутствующие данные часто помечаются значением `NaN`. Например, наш набор данных может выглядеть следующим образом:

```
In[14]: from numpy import nan
        X = np.array([[ nan, 0,   3  ],
                      [ 3,   7,   9  ],
                      [ 3,   5,   2  ],
                      [ 4,   nan, 6  ],
                      [ 8,   8,   1  ]])
        y = np.array([14, 16, -1,  8, -5])
```

При использовании для подобных данных типичной модели машинного обучения необходимо сначала заменить отсутствующие данные каким-либо подходящим значением. Это действие называется *заполнением* (imputation) пропущенных значений, и методики его выполнения варьируются от простых (например, замены пропущенных значений средним значением по столбцу) до сложных (например, с использованием восстановления матриц (matrix completion) или ошибкоустойчивого алгоритма для обработки подобных данных).

Сложные подходы, как правило, очень сильно зависят от конкретной прикладной задачи, и мы не станем углубляться в них. Для реализации стандартного подхода к заполнению пропущенных значений (с использованием среднего значения, медианы или часто встречающегося значения) библиотека Scikit-Learn предоставляет класс `Imputer`:

```
In[15]: from sklearn.preprocessing import Imputer
        imp = Imputer(strategy='mean')
        X2 = imp.fit_transform(X)
        X2

Out[15]: array([[ 4.5,  0. ,  3. ],
                [ 3. ,  7. ,  9. ],
                [ 3. ,  5. ,  2. ],
                [ 4. ,  5. ,  6. ],
                [ 8. ,  8. ,  1. ]])
```

В результате мы получили данные, в которых два пропущенные значения заменены на среднее значение остальных элементов соответствующего столбца. Эти данные можно передать, например, непосредственно оценивателю `LinearRegression`:

```
In[16]: model = LinearRegression().fit(X2, y)
        model.predict(X2)

Out[16]:
array([ 13.14869292,  14.3784627 ,  -1.15539732,  10.96606197,  -5.33782027])
```

Конвейеры признаков

Во всех предыдущих примерах может быстро надоесть выполнять преобразования вручную, особенно если нужно связать цепочкой несколько шагов. Например, нам может понадобиться следующий конвейер обработки.

1. Внести вместо отсутствующих данных средние значения.
2. Преобразовать признаки в квадратичные.
3. Обучить модель линейной регрессии.

Для организации потоковой обработки подобного конвейера библиотека Scikit-Learn предоставляет объект конвейера, который можно использовать следующим образом:

```
In[17]: from sklearn.pipeline import make_pipeline

        model = make_pipeline(Imputer(strategy='mean'),
                              PolynomialFeatures(degree=2),
                              LinearRegression())
```

Этот конвейер выглядит и функционирует аналогично обычному объекту библиотеки Scikit-Learn, и выполняет все заданные шаги для любых входных данных.

```
In[18]: model.fit(X, y)  # Вышеприведенный массив X с пропущенными значениями
        print(y)
        print(model.predict(X))

[1416 -1  8 -5]
[ 14.  16.  -1.   8.  -5.]
```

Все шаги этой модели выполняются автоматически. Обратите внимание, что для простоты демонстрации мы применили модель к тем данным, на которых она была обучена, именно поэтому она сумела столь хорошо предсказать результат (более подробное обсуждение этого вопроса можно найти в разделе «Гиперпараметры и проверка модели» данной главы).

Некоторые примеры работы конвейеров библиотеки Scikit-Learn вы увидите в следующем разделе, посвященном наивной байесовской классификации, а также в разделах «Заглянем глубже: линейная регрессия» и «Заглянем глубже: метод опорных векторов» этой главы.

Заглянем глубже: наивная байесовская классификация

Предыдущие четыре раздела были посвящены общему обзору принципов машинного обучения. В этом и следующем разделах мы подробно рассмотрим несколько специализированных алгоритмов для обучения без и с учителем, начиная с наивной байесовской классификации.

Наивные байесовские модели — группа исключительно быстрых и простых алгоритмов классификации, зачастую подходящих для наборов данных очень высоких размерностей. В силу их быстроты и столь небольшого количества настраиваемых параметров они оказываются очень удобны в качестве грубого эталона для задач классификации. Этот раздел мы посвятим наглядному объяснению работы наивных байесовских классификаторов вместе с двумя примерами их работы на некоторых наборах данных.

Байесовская классификация

Наивные байесовские классификаторы основаны на байесовских методах классификации, в основе которых лежит теорема Байеса — уравнение, описывающее связь условных вероятностей статистических величин. В байесовской классификации нас интересует поиск вероятности метки (категории) при определенных заданных признаках, являющихся результатами наблюдений/экспериментов, обозначенной $P(L \mid \text{признаков})$. Теорема Байеса позволяет выразить это в терминах величин, которые мы можем вычислить напрямую:

$$P(L \mid \text{признаков}) = \frac{P(\text{признаков} \mid L)P(L)}{P(\text{признаков})}.$$

Один из способов выбора между двумя метками (L_1 и L_2) — вычислить отношение апостериорных вероятностей для каждой из них:

$$\frac{P(L_1 \mid \text{признаков})}{P(L_2 \mid \text{признаков})} = \frac{P(\text{признаков} \mid L_1)P(L_1)}{P(\text{признаков} \mid L_2)P(L_2)}.$$

Все, что нам теперь нужно, — модель, с помощью которой можно было бы вычислить $P(\text{признаков} \mid L_i)$ для каждой из меток. Подобная модель называется *порождающей моделью* (generative model), поскольку определяет гипотетический случайный процесс генерации данных. Задание порождающей модели для каждой из меток/категорий — основа обучения подобного байесовского классификатора. Обобщенная версия подобного шага обучения — непростая задача, но мы упростим ее, приняв некоторые упрощающие допущения о виде модели.

Именно на этом этапе возникает слово «наивный» в названии «наивный байесовский классификатор»: сделав очень «наивное» допущение относительно порождающей модели для каждой из меток/категорий, можно будет отыскать грубое приближение порождающей модели для каждого класса, после чего перейти к байесовской классификации. Различные виды наивных байесовских классификаторов основываются на различных «наивных» допущениях относительно данных, мы рассмотрим несколько из них в следующих разделах. Начнем с обычных импортов:

```
In[1]: %matplotlib inline
       import numpy as np
       import matplotlib.pyplot as plt
       import seaborn as sns; sns.set()
```

Гауссов наивный байесовский классификатор

Вероятно, самый простой для понимания наивный байесовский классификатор — Гауссов. В этом классификаторе допущение состоит в том, что *данные всех категорий взяты из простого нормального распределения*. Пускай у нас имеются следующие данные (рис. 5.38):

```
In[2]: from sklearn.datasets import make_blobs
       X, y = make_blobs(100, 2, centers=2, random_state=2, cluster_std=1.5)
       plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='RdBu');
```

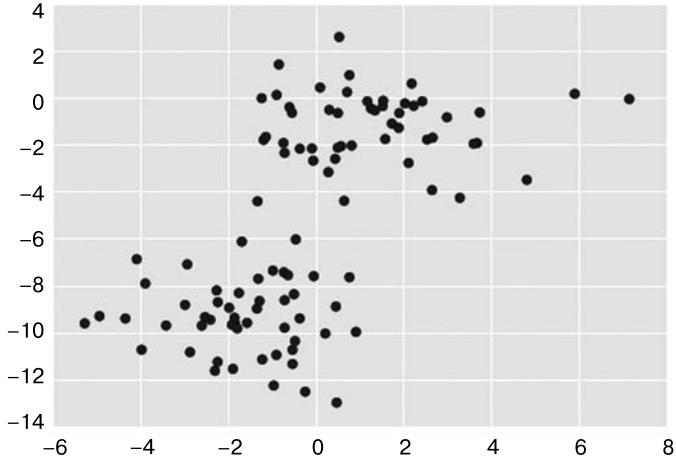

Рис. 5.38. Данные для наивной байесовской классификации

Один из самых быстрых способов создания простой модели — допущение о том, что данные подчиняются нормальному распределению без ковариации между измерениями. Для обучения этой модели достаточно найти среднее значение

и стандартное отклонение точек внутри каждой из категорий — это все, что требуется для описания подобного распределения. Результат этого наивного Гауссова допущения показан на рис. 5.39.

Эллипсы на этом рисунке представляют Гауссову порождающую модель для каждой из меток с ростом вероятности по мере приближении к центру эллипса. С помощью этой порождающей модели для каждого класса мы можем легко вычислить вероятность $P(\text{признаков} \mid L_i)$ для каждой точки данных, а следовательно, быстро рассчитать соотношение для апостериорной вероятности и определить, какая из меток с большей вероятностью соответствует конкретной точке.

Эта процедура реализована в оценивателе `sklearn.naive_bayes.GaussianNB`:

```
In[3]: from sklearn.naive_bayes import GaussianNB
       model = GaussianNB()
       model.fit(X, y);
```

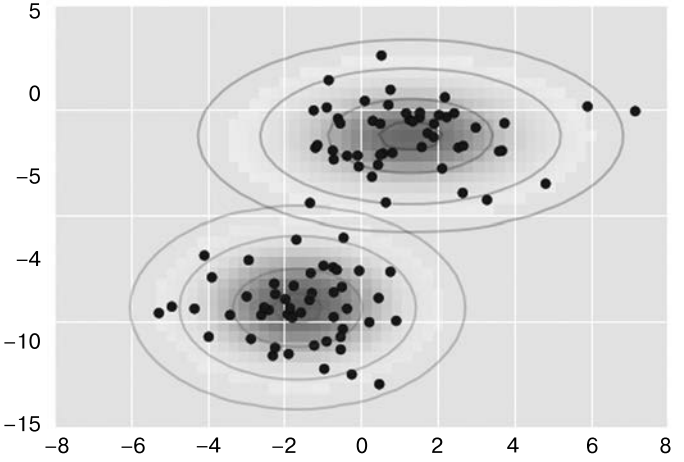

Рис. 5.39. Визуализация Гауссовой наивной байесовской модели

Сгенерируем какие-нибудь новые данные и выполним предсказание метки:

```
In[4]: rng = np.random.RandomState(0)
       Xnew = [-6, -14] + [14, 18] * rng.rand(2000, 2)
       ynew = model.predict(Xnew)
```

Теперь у нас есть возможность построить график этих новых данных и понять, где пролегает граница принятия решений (decision boundary) (рис. 5.40):

```
In[5]: plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='RdBu')
       lim = plt.axis()
       plt.scatter(Xnew[:, 0], Xnew[:, 1], c=ynew, s=20, cmap='RdBu',
                   alpha=0.1)
       plt.axis(lim);
```

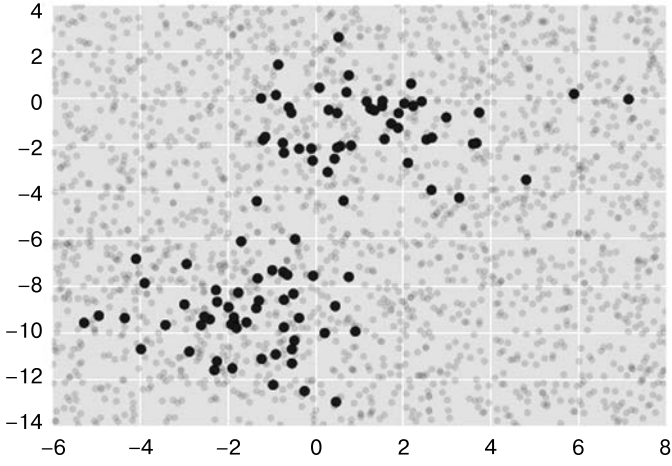

Рис. 5.40. Визуализация Гауссовой наивной байесовской классификации

Мы видим, что граница слегка изогнута, в целом граница при Гауссовом наивном байесовском классификаторе соответствует кривой второго порядка.

Положительная сторона этого байесовского формального представления заключается в возможности естественной вероятностной классификации, рассчитать которую можно с помощью метода `predict_proba`:

```
In[6]: yprob = model.predict_proba(Xnew)
       yprob[-8:].round(2)

Out[6]: array([[ 0.89,  0.11],
               [ 1.  ,  0.  ],
               [ 1.  ,  0.  ],
               [ 1.  ,  0.  ],
               [ 1.  ,  0.  ],
               [ 1.  ,  0.  ],
               [ 0.  ,  1.  ],
               [ 0.15,  0.85]])
```

Столбцы отражают апостериорные вероятности первой и второй меток соответственно. Подобные байесовские методы могут оказаться весьма удобным подходом при необходимости получения оценок погрешностей в классификации.

Качество получаемой в итоге классификации не может превышать качества исходных допущений модели, поэтому Гауссов наивный байесовский классификатор зачастую не демонстрирует слишком хороших результатов. Тем не менее во многих случаях — особенно при значительном количестве признаков — исходные допущения не настолько плохи, чтобы нивелировать удобство Гауссова наивного байесовского классификатора.

Полиномиальный наивный байесовский классификатор

Гауссово допущение — далеко не единственное простое допущение, которое можно использовать для описания порождающего распределения для всех меток. Еще один полезный пример — полиномиальный наивный байесовский классификатор с допущением о том, что признаки сгенерированы на основе простого полиномиального распределения. Полиномиальное распределение описывает вероятность наблюдения количеств вхождений в несколько категорий, таким образом, полиномиальный наивный байесовский классификатор лучше всего подходит для признаков, отражающих количество или частоту вхождения.

Основная идея остается точно такой же, но вместо моделирования распределения данных с оптимальной Гауссовой функцией мы моделируем распределение данных с оптимальным полиномиальным распределением.

Пример: классификация текста. Полиномиальный наивный байесовский классификатор нередко используется при классификации текста, где признаки соответствуют количеству слов или частотам их употребления в классифицируемых документах. Мы уже обсуждали вопрос извлечения подобных признаков из текста в разделе «Проектирование признаков» данной главы. Здесь же, чтобы продемонстрировать классификацию коротких документов по категориям, мы воспользуемся разреженными признаками количеств слов из корпуса текста 20 Newsgroups («20 дискуссионных групп»).

Скачаем данные и изучим целевые названия:

```
In[7]: from sklearn.datasets import fetch_20newsgroups
       data = fetch_20newsgroups()
       data.target_names

Out[7]: ['alt.atheism',
         'comp.graphics',
         'comp.os.ms-windows.misc',
         'comp.sys.ibm.pc.hardware',
         'comp.sys.mac.hardware',
         'comp.windows.x',
         'misc.forsale',
         'rec.autos',
         'rec.motorcycles',
         'rec.sport.baseball',
         'rec.sport.hockey',
         'sci.crypt',
         'sci.electronics',
         'sci.med',
         'sci.space',
         'soc.religion.christian',
         'talk.politics.guns',
         'talk.politics.mideast',
         'talk.politics.misc',
         'talk.religion.misc']
```

Для простоты выберем лишь несколько из этих категорий, после чего скачаем обучающую и контрольную последовательности:

```
In[8]:
categories = ['talk.religion.misc', 'soc.religion.christian', 'sci.space',
              'comp.graphics']
train = fetch_20newsgroups(subset='train', categories=categories)
test = fetch_20newsgroups(subset='test', categories=categories)
```

Вот типичный образец записи из этого набора данных:

```
In[9]: print(train.data[5])

From: dmcgee@uluhe.soest.hawaii.edu (Don McGee)
Subject: Federal Hearing
Originator: dmcgee@uluhe
Organization: School of Ocean and Earth Science and Technology
Distribution: usa
Lines: 10


Fact or rumor....?  Madalyn Murray O'Hare an atheist who eliminated the
use of the bible reading and prayer in public schools 15 years ago is now
going to appear before the FCC with a petition to stop the reading of the
Gospel on the airways of America.  And she is also campaigning to remove
Christmas programs, songs, etc from the public schools.  If it is true
then mail to Federal Communications Commission 1919 H Street Washington DC
20054 expressing your opposition to her request.  Reference Petition number
2493.
```

Чтобы использовать эти данные для машинного обучения, необходимо преобразовать содержимое каждой строки в числовой вектор. Для этого воспользуемся векторизатором TF–IDF (который обсуждали в разделе «Проектирование признаков» текущей главы) и создадим конвейер, присоединяющий его последовательно к полиномиальному наивному байесовскому классификатору:

```
In[10]: from sklearn.feature_extraction.text import TfidfVectorizer
        from sklearn.naive_bayes import MultinomialNB
        from sklearn.pipeline import make_pipeline

        model = make_pipeline(TfidfVectorizer(), MultinomialNB())
```

С помощью этого конвейера мы можем применить модель к обучающей последовательности и предсказать метки для контрольных данных:

```
In[11]: model.fit(train.data, train.target)
        labels = model.predict(test.data)
```

Теперь, предсказав метки для контрольных данных, мы изучим их, чтобы выяснить эффективность работы оценивателя. Например, вот матрица различий между настоящими и предсказанными метками для контрольных данных (рис. 5.41):

```
In[12]:
from sklearn.metrics import confusion_matrix
mat = confusion_matrix(test.target, labels)
sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False,
            xticklabels=train.target_names, yticklabels=train.target_names)
plt.xlabel('true label')
plt.ylabel('predicted label');
```

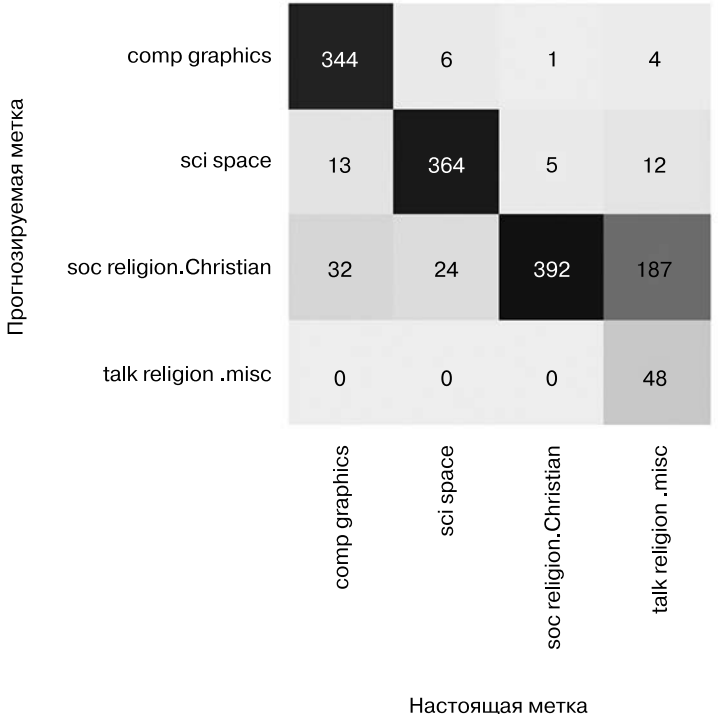

Рис. 5.41. Матрица различий для полиномиального наивного байесовского классификатора текста

Даже этот очень простой классификатор может легко отделять обсуждения космоса от дискуссий о компьютерах, но он путает обсуждения религии вообще и обсуждения христианства. Вероятно, этого следовало ожидать!

Хорошая новость состоит в том, что у нас теперь есть инструмент определения категории для *любой* строки с помощью метода `predict()` нашего конвейера. Следующий фрагмент кода описывает простую вспомогательную функцию, возвращающую предсказание для отдельной строки:

```
In[13]: def predict_category(s, train=train, model=model):
            pred = model.predict([s])
            return train.target_names[pred[0]]
```

Попробуем ее на деле:

```
In[14]: predict_category('sending a payload to the ISS')

Out[14]: 'sci.space'

In[15]: predict_category('discussing islam vs atheism')

Out[15]: 'soc.religion.christian'

In[16]: predict_category('determining the screen resolution')

Out[16]: 'comp.graphics'
```

Это лишь простая вероятностная модель (взвешенной) частоты каждого из слов в строке, тем не менее результат поразителен. Даже очень наивный алгоритм может оказаться удивительно эффективным при разумном использовании и обучении на большом наборе многомерных данных.

Когда имеет смысл использовать наивный байесовский классификатор

В силу столь строгих допущений относительно данных наивные байесовские классификаторы обычно работают хуже, чем более сложные модели. Тем не менее у них есть несколько достоинств:

- они выполняют как обучение, так и предсказание исключительно быстро;
- обеспечивают простое вероятностное предсказание;
- их результаты часто очень легки для интерпретации;
- у них очень мало (если вообще есть) настраиваемых параметров.

Эти достоинства означают, что наивный байесовский классификатор зачастую оказывается удачным кандидатом на роль первоначальной эталонной классификации. Если оказывается, что он демонстрирует удовлетворительные результаты, то поздравляем: вы нашли для своей задачи очень быстрый классификатор, возвращающий очень удобные для интерпретации результаты. Если же нет, то вы всегда можете начать пробовать более сложные модели, уже имея представление о том, насколько хорошо они должны работать.

Наивные байесовские классификаторы склонны демонстрировать особенно хорошие результаты в следующих случаях:

- когда данные действительно соответствуют наивным допущениям (на практике бывает очень редко);
- для очень хорошо разделяемых категорий, когда сложность модели не столь важна;

- для данных с очень большим числом измерений, когда сложность модели не столь важна.

Два последних случая кажутся отдельными, но на самом деле они взаимосвязаны: по мере роста количества измерений у набора данных вероятность близости любых двух точек падает (в конце концов, чтобы находиться рядом, они должны находиться рядом *по каждому из измерений*). Это значит, что кластеры в многомерных случаях имеют склонность к более выраженной изоляции, чем кластеры в случаях с меньшим количеством измерений (конечно, если новые измерения действительно вносят дополнительную информацию). Поэтому упрощенные классификаторы, такие как наивный байесовский классификатор, при росте количества измерений начинают работать не хуже, а то и лучше более сложных: когда данных достаточно много, даже простая модель может оказаться весьма эффективной.

Заглянем глубже: линейная регрессия

Аналогично тому, как наивный байесовский классификатор (который мы обсуждали в разделе: «Заглянем глубже: наивная байесовская классификация» данной главы) — отличная отправная точка для задач классификации, так и линейные регрессионные модели — хорошая отправная точка для задач регрессии. Подобные модели популярны в силу быстрой обучаемости и возврата очень удобных для интерпретации результатов. Вероятно, вы уже знакомы с простейшей формой линейной регрессионной модели (то есть подбора для данных разделяющей прямой линии), но такие модели можно распространить на моделирование и более сложного поведения данных.

В этом разделе мы начнем с быстрого и интуитивно понятного математического описания известной задачи, а потом перейдем к вопросам обобщенных линейных моделей, учитывающих более сложные паттерны в данных. Начнем с обычных импортов:

```
In[1]: %matplotlib inline
       import matplotlib.pyplot as plt
       import seaborn as sns; sns.set()
       import numpy as np
```

Простая линейная регрессия

Начнем с линейной регрессии, прямолинейной аппроксимации наших данных. Прямолинейная аппроксимация представляет собой модель вида $y = ax + b$, в которой a известна как *угловой коэффициент*, а b — как *точка пересечения с осью координат Y*.

Рассмотрим следующие данные, распределенные около прямой с угловым коэффициентом 2 и точкой пересечения –5 (рис. 5.42):

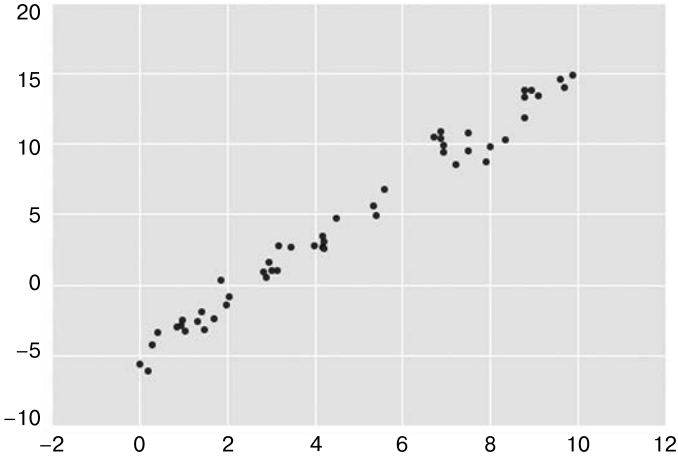

Рис. 5.42. Данные для линейной регрессии

```
In[2]: rng = np.random.RandomState(1)
       x = 10 * rng.rand(50)
       y = 2 * x - 5 + rng.randn(50)
       plt.scatter(x, y);
```

Воспользуемся оценивателем `LinearRegression` из библиотеки Scikit-Learn для обучения на этих данных и поиска оптимальной прямой (рис. 5.43):

```
In[3]: from sklearn.linear_model import LinearRegression
       model = LinearRegression(fit_intercept=True)

       model.fit(x[:, np.newaxis], y)

       xfit = np.linspace(0, 10, 1000)
       yfit = model.predict(xfit[:, np.newaxis])

       plt.scatter(x, y)
       plt.plot(xfit, yfit);
```

Подбираемые параметры модели (в библиотеке Scikit-Learn всегда содержат в конце знак подчеркивания) включают угловой коэффициент и точку пересечения с осью координат. В данном случае соответствующие параметры — `coef_` и `intercept_`:

```
In[4]: print("Model slope:    ", model.coef_[0])
       print("Model intercept:", model.intercept_)

Model slope:     2.02720881036
Model intercept: -4.99857708555
```

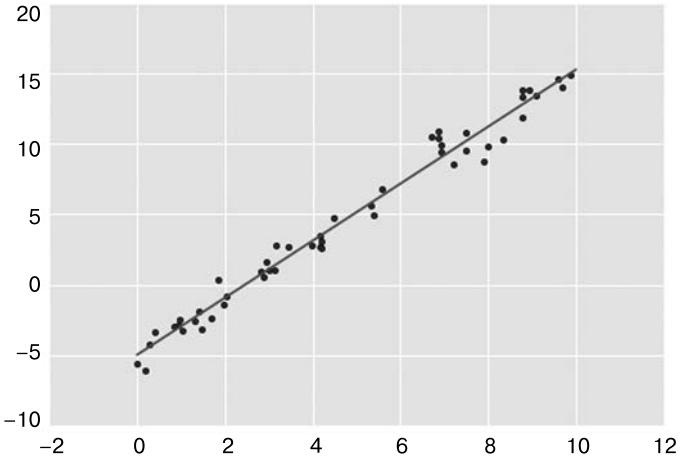

Рис. 5.43. Линейная регрессионная модель

Видим, что результаты очень близки к входным данным, как мы и надеялись.

Однако возможности оценивателя `LinearRegression` намного шире этого: помимо аппроксимации прямыми линиями, он может также работать с многомерными линейными моделями следующего вида:

$$y = a_0 + a_1x_1 + a_2x_2 + \ldots$$

с несколькими величинами x. Геометрически это подобно подбору плоскости для точек в трех измерениях или гиперплоскости для точек в пространстве с еще бóльшим числом измерений.

Многомерная сущность подобных регрессий усложняет их визуализацию, но мы можем посмотреть на одну из этих аппроксимаций в действии, создав данные для нашего примера с помощью оператора матричного умножения из библиотеки NumPy:

```
In[5]: rng = np.random.RandomState(1)
       X = 10 * rng.rand(100, 3)
       y = 0.5 + np.dot(X, [1.5, -2., 1.])

       model.fit(X, y)
       print(model.intercept_)
       print(model.coef_)
```

```
0.5
[ 1.5 -2.   1. ]
```

Здесь данные величины y сформированы из трех случайных значений величины x, а линейная регрессия восстанавливает использовавшиеся для их формирования коэффициенты.

Аналогичным образом можно использовать оцениватель `LinearRegression` для аппроксимации наших данных прямыми, плоскостями и гиперплоскостями. По-прежнему складывается впечатление, что этот подход ограничивается лишь строго линейными отношениями между переменными, но оказывается, что ослабление этого требования также возможно.

Регрессия по комбинации базисных функций

Один из трюков, позволяющих приспособить линейную регрессию к нелинейным отношениям между переменными, — преобразование данных в соответствии с новыми базисными функциями. Один из вариантов этого трюка мы уже встречали в конвейере `PolynomialRegression`, который использовался в разделах «Гиперпараметры и проверка модели» и «Проектирование признаков» данной главы. Идея состоит в том, чтобы взять многомерную линейную модель:

$$y = a_0 + a_1 x_1 + a_2 x^2 + a_3 x^3 + \ldots$$

и построить x_1, x_2, x_3 и т. д. на основе имеющегося одномерного входного значения x. То есть у нас $x_n = f_n(x)$, где $f_n(x)$ — некая функция, выполняющая преобразование данных.

Например, если $f_n(x) = x^n$, наша модель превращается в полиномиальную регрессию:

$$y = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \ldots$$

Обратите внимание, что модель по-прежнему *остается линейной* — линейность относится к тому, что коэффициенты a_n никогда не умножаются и не делятся друг на друга. Фактически мы взяли наши одномерные значения x и выполнили проекцию их на более многомерное пространство, так что с помощью линейной аппроксимации мы можем теперь отражать более сложные зависимости между x и y.

Полиномиальные базисные функции

Данная полиномиальная проекция настолько удобна, что была встроена в библиотеку Scikit-Learn в виде преобразователя `PolynomialFeatures`:

```
In[6]: from sklearn.preprocessing import PolynomialFeatures
       x = np.array([2, 3, 4])
       poly = PolynomialFeatures(3, include_bias=False)
       poly.fit_transform(x[:, None])

Out[6]: array([[  2.,   4.,   8.],
               [  3.,   9.,  27.],
               [  4.,  16.,  64.]])
```

Как видим, преобразователь превратил наш одномерный массив в трехмерный путем возведения каждого из значений в степень. Это новое, более многомерное представление данных можно далее использовать для линейной регрессии.

Как мы уже видели в разделе «Проектирование признаков» данной главы, самый изящный способ выполнения этого — воспользоваться конвейером. Создадим, указанным образом полиномиальную модель седьмого порядка:

```
In[7]: from sklearn.pipeline import make_pipeline
       poly_model = make_pipeline(PolynomialFeatures(7), LinearRegression())
```

После такого преобразования можно воспользоваться линейной моделью для подбора намного более сложных зависимостей между величинами x и y. Например, рассмотрим зашумленную синусоиду (рис. 5.44):

```
In[8]: rng = np.random.RandomState(1)
       x = 10 * rng.rand(50)
       y = np.sin(x) + 0.1 * rng.randn(50)

       poly_model.fit(x[:, np.newaxis], y)
       yfit = poly_model.predict(xfit[:, np.newaxis])

       plt.scatter(x, y)
       plt.plot(xfit, yfit);
```

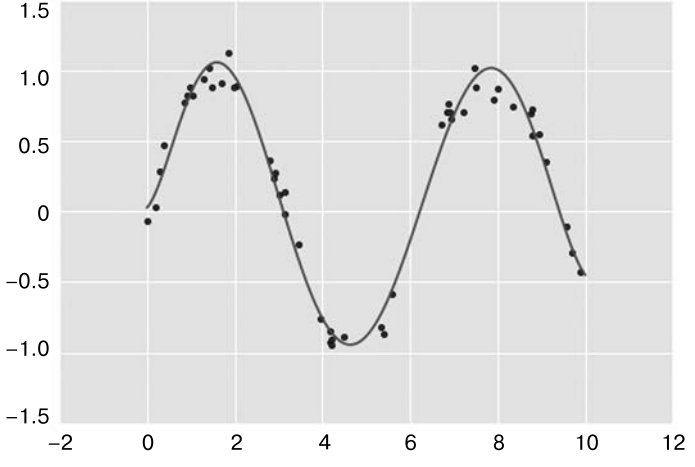

Рис. 5.44. Полиномиальная аппроксимация нелинейной обучающей последовательности

С нашей линейной моделью, используя полиномиальные базисные функции седьмого порядка, мы получили великолепную аппроксимацию этих нелинейных данных!

Гауссовы базисные функции

Можно использовать и другие базисные функции. Например, один из полезных паттернов — обучение модели, представляющей собой сумму не полиномиальных, а Гауссовых базисных функций. Результат будет выглядеть следующим образом (рис. 5.45):

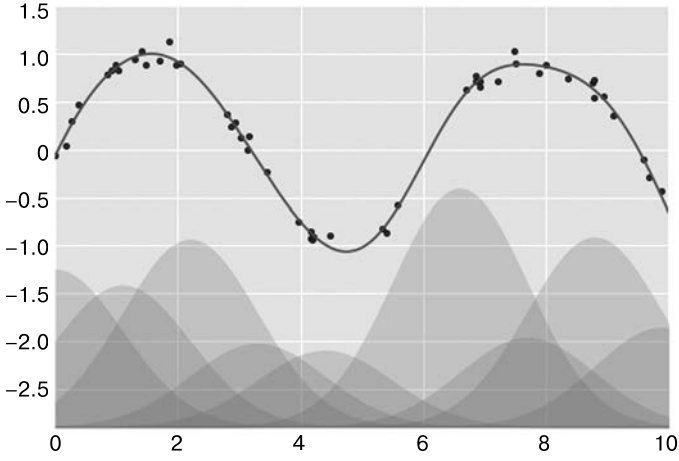

Рис. 5.45. Аппроксимация нелинейных данных с помощью Гауссовых базисных функций

Затененные области на рис. 5.45 — нормированные базисные функции, дающие при сложении аппроксимирующую данные гладкую кривую. Эти Гауссовы базисные функции не встроены в библиотеку Scikit-Learn, но мы можем написать для их создания пользовательский преобразователь, как показано ниже и проиллюстрировано на рис. 5.46 (преобразователи библиотеки Scikit-Learn реализованы как классы языка Python; чтение исходного кода библиотеки Scikit-Learn — отличный способ разобраться с их созданием):

```
In[9]:
from sklearn.base import BaseEstimator, TransformerMixin

class GaussianFeatures(BaseEstimator, TransformerMixin):
    """Равномерно распределенные Гауссовы признаки
       для одномерных входных данных"""

    def __init__(self, N, width_factor=2.0):
        self.N = N
        self.width_factor = width_factor

    @staticmethod
    def _gauss_basis(x, y, width, axis=None):
```

```
        arg = (x - y) / width
        return np.exp(-0.5 * np.sum(arg ** 2, axis))

    def fit(self, X, y=None):
        # Создаем N центров, распределенных по всему диапазону данных
        self.centers_ = np.linspace(X.min(), X.max(), self.N)
        self.width_ = self.width_factor *
        (self.centers_[1] - self.centers_[0])
        return self

    def transform(self, X):
        return self._gauss_basis(X[:, :, np.newaxis], self.centers_,
                                 self.width_, axis=1)

gauss_model = make_pipeline(GaussianFeatures(20),
                            LinearRegression())

gauss_model.fit(x[:, np.newaxis], y)
yfit = gauss_model.predict(xfit[:, np.newaxis])

plt.scatter(x, y)
plt.plot(xfit, yfit)
plt.xlim(0, 10);
```

Мы привели этот пример лишь для того, чтобы подчеркнуть, что в полиномиальных базисных функциях нет никакого колдовства. Если у вас есть какие-то дополнительные сведения о процессе генерации ваших данных, исходя из которых у вас есть основания полагать, что наиболее подходящим будет тот или иной базис, — тоже можете его использовать.

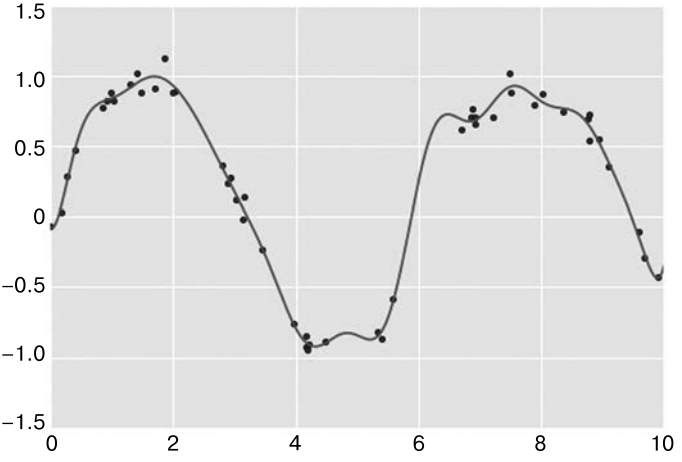

Рис. 5.46. Аппроксимация Гауссовыми базисными функциями, вычисленными с помощью пользовательского преобразователя

Регуляризация

Применение базисных функций в нашей линейной модели делает ее намного гибче, но также и быстро приводит к переобучению (за подробностями обратитесь к разделу «Гиперпараметры и проверка модели» данной главы). Например, если выбрать слишком много Гауссовых базисных функций, мы в итоге получим не слишком хорошие результаты (рис. 5.47):

```
In[10]: model = make_pipeline(GaussianFeatures(30),
                              LinearRegression())
        model.fit(x[:, np.newaxis], y)

        plt.scatter(x, y)
        plt.plot(xfit, model.predict(xfit[:, np.newaxis]))
        plt.xlim(0, 10)
        plt.ylim(-1.5, 1.5);
```

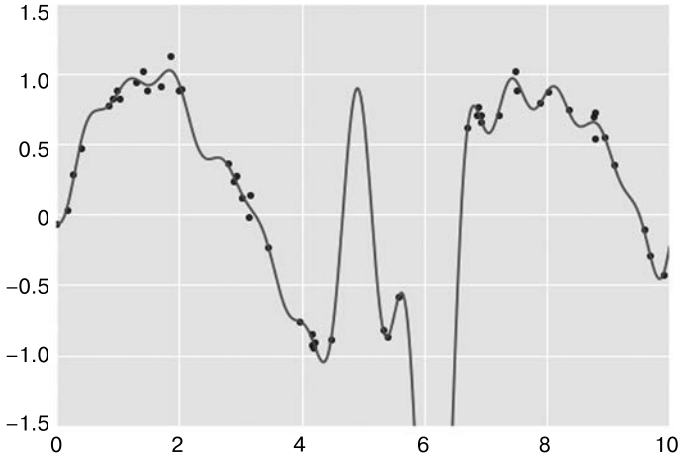

Рис. 5.47. Пример переобучения на данных: слишком сложная модель с базисными функциями

В результате проекции данных на 30-мерный базис модель оказалась слишком уж гибкой и стремится к экстремальным значениям в промежутках между точками, в которых она ограничена данными. Причину этого можно понять, построив график коэффициентов Гауссовых базисных функций в соответствии с координатой x (рис. 5.48):

```
In[11]: def basis_plot(model, title=None):
            fig, ax = plt.subplots(2, sharex=True)
            model.fit(x[:, np.newaxis], y)
            ax[0].scatter(x, y)
            ax[0].plot(xfit, model.predict(xfit[:, np.newaxis]))
            ax[0].set(xlabel='x', ylabel='y', ylim=(-1.5, 1.5))

            if title:
                ax[0].set_title(title)
```

```
    ax[1].plot(model.steps[0][1].centers_,
               model.steps[1][1].coef_)
    ax[1].set(xlabel='basis location', # Базовое местоположение
              ylabel='coefficient',    # Коэффициент
              xlim=(0, 10))

model = make_pipeline(GaussianFeatures(30), LinearRegression())
basis_plot(model)
```

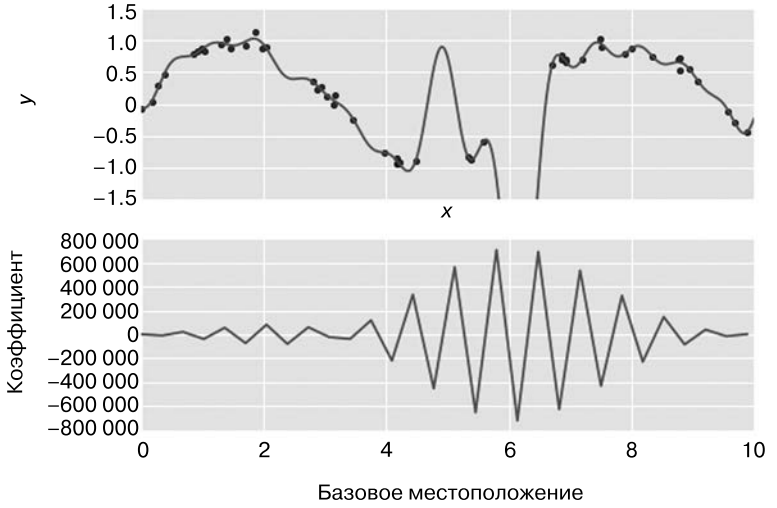

Рис. 5.48. Коэффициенты при Гауссовых базисных функциях в чрезмерно сложной модели

Нижняя часть рис. 5.48 демонстрирует амплитуду базисной функции в каждой из точек. Это типичное поведение для переобучения с перекрытием областей определения базисных функций: коэффициенты соседних базисных функций усиливают и подавляют друг друга. Мы знаем, что подобное поведение приводит к проблемам и было бы неплохо ограничивать подобные пики в модели явным образом, «накладывая штраф» на большие значения параметров модели. Подобное «штрафование» известно под названием регуляризации и существует в нескольких вариантах.

Гребневая регрессия (L_2-регуляризация)

Вероятно, самый часто встречающийся вид регуляризации — *гребневая регрессия* (ridge regression), или *L_2-регуляризация* (L_2-regularization), также иногда называемая *регуляризацией Тихонова* (Tikhonov regularization). Она заключается в наложении штрафа на сумму квадратов (евклидовой нормы) коэффициентов модели. В данном случае штраф для модели будет равен:

$$P = \alpha \sum_{n=1}^{N} \theta_n^2,$$

где α — свободный параметр, служащий для управления уровнем штрафа. Этот тип модели со штрафом встроен в библиотеку Scikit-Learn в виде оценивателя `Ridge` (рис. 5.49):

```
In[12]: from sklearn.linear_model import Ridge
        model = make_pipeline(GaussianFeatures(30), Ridge(alpha=0.1))
        basis_plot(model, title='Ridge Regression') # Гребневая регрессия
```

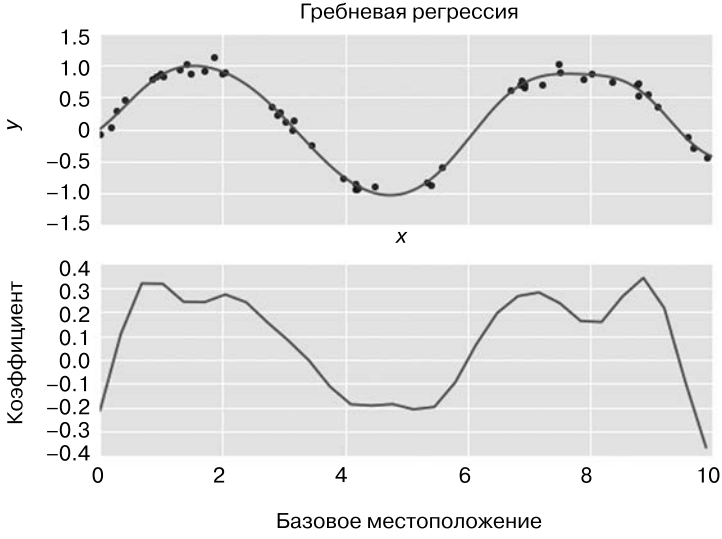

Рис. 5.49. Применение гребневой (L_2) регуляризации к слишком сложной модели (ср. с рис. 5.48)

Параметр α служит для управления сложностью получаемой в итоге модели. В предельном случае $\alpha \to 0$ мы получаем результат, соответствующий стандартной линейной регрессии; в предельном случае $\alpha \to \infty$ будет происходить подавление любого отклика модели. Достоинства гребневой регрессии включают, помимо прочего, возможность ее эффективного расчета — вычислительные затраты практически не превышают затрат на расчет исходной линейной регрессионной модели.

Лассо-регуляризация (L_1)

Еще один распространенный тип регуляризации — так называемая лассо-регуляризация, включающая штрафование на сумму абсолютных значений (L_1-норма) коэффициентов регрессии:

$$P = \alpha \sum_{n=1}^{N} \left| \theta_n \right|.$$

Хотя концептуально эта регрессия очень близка к гребневой, результаты их могут очень сильно различаться. Например, по геометрическим причинам лассо-регрессия любит *разреженные модели*, то есть она по возможности делает коэффициенты модели равными нулю.

Посмотреть на поведение этой регрессии мы можем, воспроизведя показанный на рис. 5.49 график, но с использованием коэффициентов, нормализованных с помощью нормы L_1 (рис. 5.50):

```
In[13]: from sklearn.linear_model import Lasso
        model = make_pipeline(GaussianFeatures(30), Lasso(alpha=0.001))
        basis_plot(model, title='Lasso Regression') # Лассо-регуляризация
```

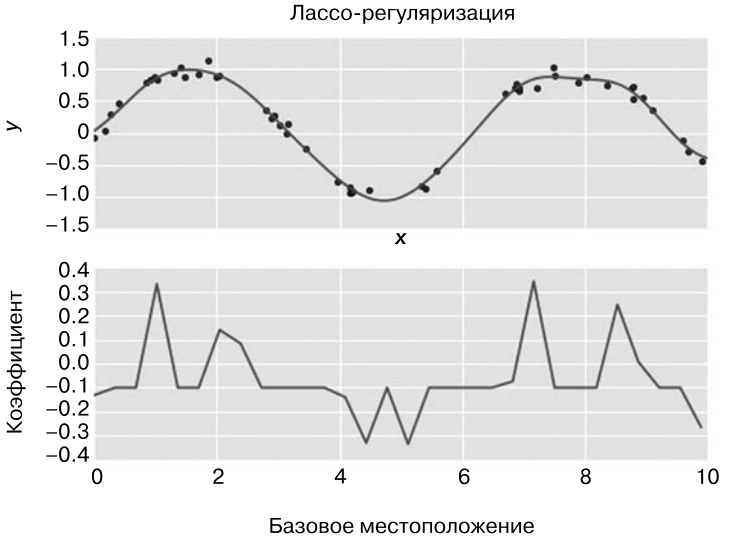

Рис. 5.50. Применение лассо-регуляризации к слишком сложной модели (ср. с рис. 5.48)

При использовании штрафа лассо-регрессии большинство коэффициентов в точности равны нулю, а функциональное поведение моделируется небольшим подмножеством из имеющихся базисных функций. Как и в случае гребневой регуляризации, параметр управляет уровнем штрафа и его следует определять путем перекрестной проверки (см. раздел «Гиперпараметры и проверка модели» данной главы).

Пример: предсказание велосипедного трафика

В качестве примера посмотрим, сможем ли мы предсказать количество пересекающих Фримонтский мост в Сиэтле велосипедов, основываясь на данных о погоде, времени года и других факторах. Мы уже работали с этими данными в разделе «Работа с временными рядами» главы 3.

В этом разделе мы соединим данные по велосипедам с другим набором данных и попробуем установить, насколько погода и сезонные факторы — температура, осадки и световой день — влияют на объем велосипедного трафика по этому коридору. NOAA открыло доступ к ежедневным данным с их метеорологических станций (я воспользовался станцией с ID USW00024233), а библиотека Pandas дает нам возможность с легкостью соединить эти два источника данных. Мы установим отношение между погодой и другой информацией с количеством велосипедов, выполнив простую линейную регрессию, чтобы оценить, как изменения этих параметров повлияют на число велосипедистов в заданный день.

В частности, это пример использования подобных библиотеке Scikit-Learn инструментов в фреймворке статистического моделирования, где предполагается осмысленность параметров модели. Это отнюдь не стандартный подход для машинного обучения, но для некоторых моделей подобная трактовка возможна.

Начнем с загрузки двух наборов данных, индексированных по дате:

```
In[14]:
import pandas as pd
counts = pd.read_csv('fremont_hourly.csv', index_col='Date', parse_dates=True)
weather = pd.read_csv('599021.csv', index_col='DATE', parse_dates=True)
```

Далее вычислим общий ежедневный поток велосипедов и поместим эти данные в отдельный объект `DataFrame`:

```
In[15]: daily = counts.resample('d', how='sum')
        daily['Total'] = daily.sum(axis=1)
        daily = daily[['Total']] # удаляем остальные столбцы
```

Мы видели ранее, что паттерны использования варьируются день ото дня. Учтем это в наших данных, добавив двоичные столбцы-индикаторы дня недели:

```
In[16]: days = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
        for i in range(7):
            daily[days[i]] = (daily.index.dayofweek == i).astype(float)
```

Следует ожидать, что велосипедисты будут вести себя иначе по выходным. Добавим индикаторы и для этого:

```
In[17]: from pandas.tseries.holiday import USFederalHolidayCalendar
        cal = USFederalHolidayCalendar()
        holidays = cal.holidays('2012', '2016')
        daily = daily.join(pd.Series(1, index=holidays, name='holiday'))
        daily['holiday'].fillna(0, inplace=True)
```

Логично предположить, что свое влияние на количество едущих на велосипедах людей окажет и световой день. Воспользуемся стандартными астрономическими расчетами для добавления этой информации (рис. 5.51):

```
In[18]:
        def hours_of_daylight(date, axis=23.44, latitude=47.61):
            """Рассчитываем длительность светового дня для заданной даты"""
            days = (date - pd.datetime(2000, 12, 21)).days
            m = (1. - np.tan(np.radians(latitude))
                 * np.tan(np.radians(axis) *
                 np.cos(days * 2 * np.pi / 365.25)))
            return 24. * np.degrees(np.arccos(1 - np.clip(m, 0, 2))) / 180.

        daily['daylight_hrs'] = list(map(hours_of_daylight, daily.index))
        daily[['daylight_hrs']].plot();
```

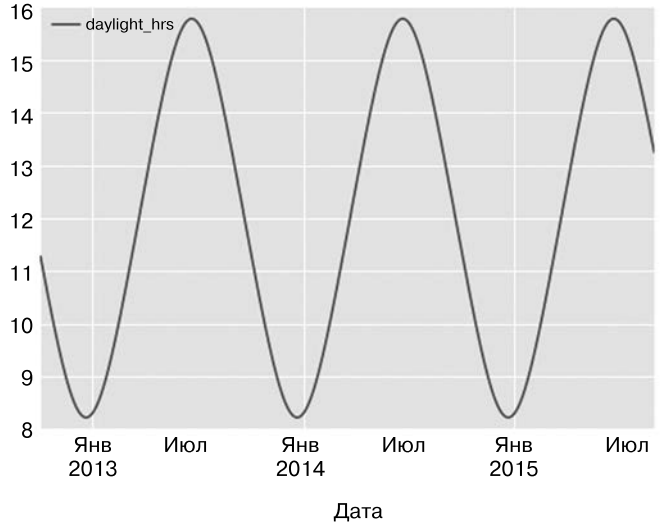

Рис. 5.51. Визуализация длительности светового дня в Сиэтле

Мы также добавим к данным среднюю температуру и общее количество осадков. Помимо количества дюймов осадков, добавим еще и флаг для обозначения засушливых дней (с нулевым количеством осадков):

```
In[19]: # Температуры указаны в десятых долях градуса Цельсия;
        # преобразуем в градусы
        weather['TMIN'] /= 10
        weather['TMAX'] /= 10
        weather['Temp (C)'] = 0.5 * (weather['TMIN'] + weather['TMAX'])

        # Осадки указаны в десятых долях миллиметра; преобразуем в дюймы
        weather['PRCP'] /= 254
        weather['dry day'] = (weather['PRCP'] == 0).astype(int)

        daily = daily.join(weather[['PRCP', 'Temp (C)', 'dry day']])
```

Добавим счетчик, который будет увеличиваться, начиная с первого дня, и отмерять количество прошедших лет. Он позволит нам отслеживать ежегодные увеличения или уменьшения ежедневного количества проезжающих:

```
In[20]: daily['annual'] = (daily.index - daily.index[0]).days / 365.
```

Теперь наши данные приведены в полный порядок, и мы можем посмотреть на них:

```
In[21]: daily.head()
```

```
Out[21]:
            Total  Mon  Tue  Wed  Thu  Fri  Sat  Sun  holiday  daylight_hrs  \\
Date
2012-10-03   3521    0    0    1    0    0    0    0        0     11.277359
2012-10-04   3475    0    0    0    1    0    0    0        0     11.219142
2012-10-05   3148    0    0    0    0    1    0    0        0     11.161038
2012-10-06   2006    0    0    0    0    0    1    0        0     11.103056
2012-10-07   2142    0    0    0    0    0    0    1        0     11.045208

             PRCP  Temp (C)  dry day    annual
Date
2012-10-03      0     13.35        1  0.000000
2012-10-04      0     13.60        1  0.002740
2012-10-05      0     15.30        1  0.005479
2012-10-06      0     15.85        1  0.008219
2012-10-07      0     15.85        1  0.010959
```

После этого можно выбрать нужные столбцы и обучить линейную регрессионную модель на наших данных. Зададим параметр `fit_intercept = False`, поскольку флаги для дней, по сути, выполняют подбор точек пересечения с осями координат по дням:

```
In[22]:
column_names = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun', 'holiday',
                'daylight_hrs', 'PRCP', 'dry day', 'Temp (C)', 'annual']
X = daily[column_names]
y = daily['Total'] # Всего

model = LinearRegression(fit_intercept=False)
model.fit(X, y)
daily['predicted'] = model.predict(X)
```

Сравниваем общий и предсказанный моделью велосипедный трафик визуально (рис. 5.52):

```
In[23]: daily[['Total', 'predicted']].plot(alpha=0.5);
```

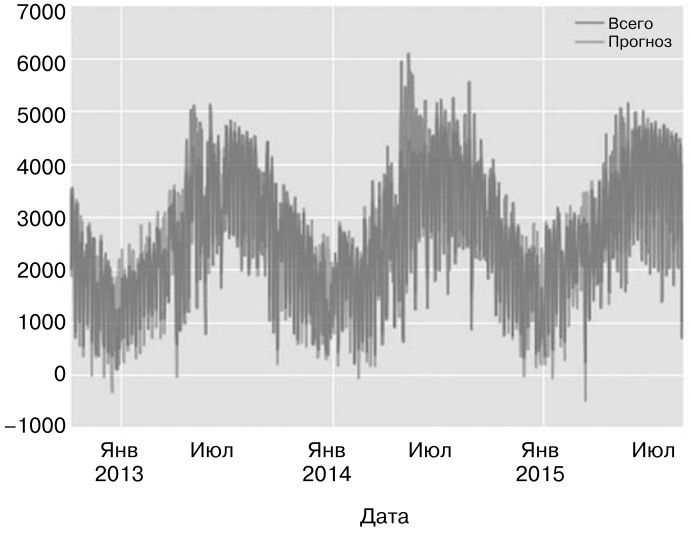

Рис. 5.52. Предсказанные моделью значения велосипедного трафика

Очевидно, что мы упустили какие-то ключевые признаки, особенно летом. Или список наших признаков неполон (то есть люди принимают решение о том, ехать ли на работу на велосипеде, основываясь не только на них), или имеются какие-то нелинейные зависимости, которые нам не удалось учесть (например, возможно, что люди ездят реже как в жару, так и в холод). Тем не менее нашей грубой аппроксимации достаточно, чтобы дать некоторую информацию о данных, и мы можем посмотреть на коэффициенты нашей линейной модели, чтобы оценить, какой вклад вносит в ежедневное количество поездок на велосипедах каждый из признаков:

```
In[24]: params = pd.Series(model.coef_, index=X.columns)
        params

Out[24]: Mon              503.797330
         Tue              612.088879
         Wed              591.611292
         Thu              481.250377
         Fri              176.838999
         Sat            -1104.321406
         Sun            -1134.610322
         holiday        -1187.212688
         daylight_hrs     128.873251
         PRCP            -665.185105
         dry day          546.185613
         Temp (C)          65.194390
         annual            27.865349
         dtype: float64
```

Эти числа нелегко интерпретировать в отсутствие какой-либо меры их неопределенности[1]. Быстро вычислить погрешности можно путем бутстрэппинга — повторных выборок данных:

```
In[25]: from sklearn.utils import resample
        np.random.seed(1)
        err = np.std([model.fit(*resample(X, y)).coef_
                      for i in range(1000)], 0)
```

Оценив эти ошибки, взглянем на результаты еще раз:

```
In[26]: print(pd.DataFrame({'effect': params.round(0),
                            'error': err.round(0)}))
```

```
              effect  error
Mon              504     85
Tue              612     82
Wed              592     82
Thu              481     85
Fri              177     81
Sat            -1104     79
Sun            -1135     82
holiday        -1187    164
daylight_hrs     129      9
PRCP            -665     62
dry day          546     33
Temp (C)          65      4
annual            28     18
```

Прежде всего мы видим, что существует довольно устойчивая тенденция относительно еженедельного минимума: по будним дням велосипедистов намного больше, чем по выходным и праздникам. Мы видим, что с каждым дополнительным часом светлого времени суток велосипедистов становится больше на 129 ± 9; рост температуры на 1 градус Цельсия стимулирует 65 ± 4 человек взяться за велосипед; сухой день означает в среднем на 546 ± 33 больше велосипедистов; каждый дюйм осадков означает, что на 665 ± 62 больше людей оставляют велосипед дома. После учета всех влияний мы получаем умеренный рост ежедневного количества велосипедистов на 28 ± 18 человек в год.

Нашей модели почти наверняка недостает определенной относящейся к делу информации. Например, нелинейные влияния (такие как совместное влияние осадков *и* низкой температуры) и нелинейные тренды в пределах каждой из переменных (такие как нежелание ездить на велосипедах в очень холодную и очень жаркую погоду) не могут быть учтены в этой модели. Кроме того, мы отбросили некоторые нюансы (такие как различие между дождливым утром

[1] То есть погрешности.

и дождливым полуднем), а также проигнорировали корреляции между днями (такие как возможное влияние дождливого вторника на показатели среды или влияние внезапного солнечного дня после полосы дождливых). Это все очень интересные влияния, и у вас, если захотите, теперь есть инструменты для их исследования!

Заглянем глубже: метод опорных векторов

Метод опорных векторов (support vector machines, SVMs) — очень мощный и гибкий класс алгоритмов обучения с учителем как для классификации, так и регрессии. В этом разделе мы научимся интуитивно понимать, как использовать метод опорных векторов в задачах классификации.

Начнем с обычных импортов:

```
In[1]: %matplotlib inline
       import numpy as np
       import matplotlib.pyplot as plt
       from scipy import stats

       # Воспользуемся настройками по умолчанию библиотеки Seaborn
       import seaborn as sns; sns.set()
```

Основания для использования метода опорных векторов

В ходе нашего обсуждения байесовской классификации (см. раздел «Заглянем глубже: наивная байесовская классификация» данной главы) мы изучили простую модель, описывающую распределение всех базовых классов, и воспользовались подобными порождающими моделями для вероятностного определения меток для новых точек. Это был пример *порождающей классификации* (generative classification), здесь же мы рассмотрим *разделяющую классификацию* (discriminative classification).

Вместо моделирования каждого из классов мы найдем прямую или кривую (в двумерном пространстве) или многообразие (в многомерном пространстве), отделяющее классы друг от друга.

В качестве примера рассмотрим простой случай задачи классификации, в котором два класса точек вполне разделены (рис. 5.53):

```
In[2]: from sklearn.datasets.samples_generator import make_blobs
       X, y = make_blobs(n_samples=50, centers=2,
                         random_state=0, cluster_std=0.60)
       plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn');
```

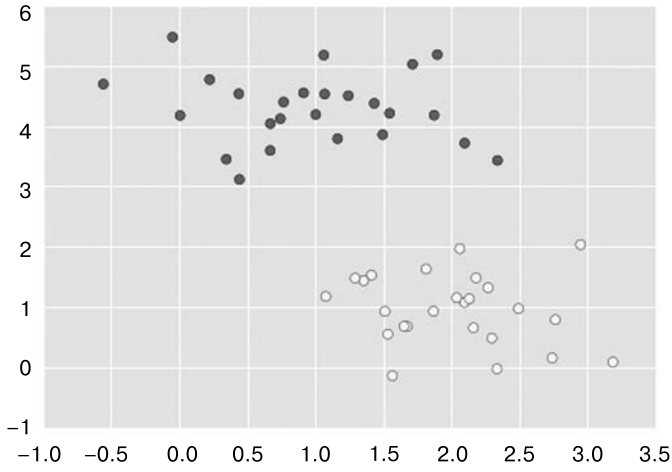

Рис. 5.53. Простые данные для классификации

Линейный разделяющий классификатор попытается провести прямую линию, разделяющую два набора данных, создав таким образом модель для классификации. В случае подобных двумерных данных это задача, которую можно решить вручную.

Однако сразу же возникает проблема: существует более одной идеально разделяющей два класса прямой!

Нарисуем их (рис. 5.54).

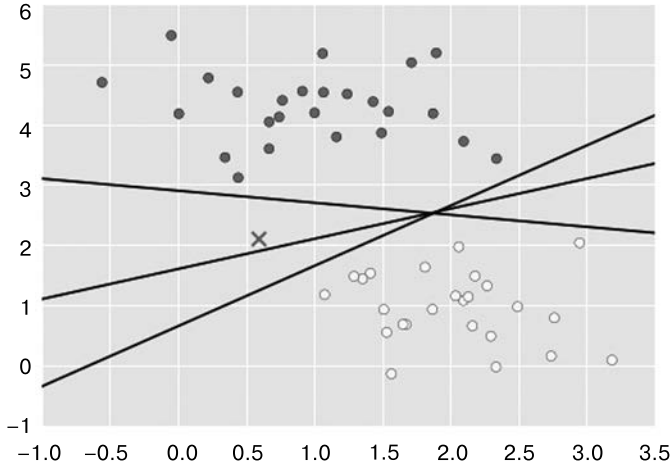

Рис. 5.54. Три идеальных линейных разделяющих классификатора для наших данных

```
In[3]: xfit = np.linspace(-1, 3.5)
       plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
       plt.plot([0.6], [2.1], 'x', color='red', markeredgewidth=2,
                markersize=10)

       for m, b in [(1, 0.65), (0.5, 1.6), (-0.2, 2.9)]:
           plt.plot(xfit, m * xfit + b, '-k')

       plt.xlim(-1, 3.5);
```

На рис. 5.54 показаны три *очень* разных разделителя, тем не менее прекрасно разделяющих наши выборки. В зависимости от того, какой из них вы выберете, новой точке данных (например, отмеченной знаком «X» на рис. 5.54) будут присвоены различные метки! Очевидно, наш интуитивный подход с «проведением прямой между классами» работает недостаточно хорошо и нужно подойти к вопросу с более глубоких позиций.

Метод опорных векторов: максимизируем отступ

Метод опорных векторов предоставляет способ решения этой проблемы. Идея заключается в следующем: вместо того чтобы рисовать между классами прямую нулевой ширины, можно нарисовать около каждой из прямых *отступ* (margin) некоторой ширины, простирающийся до ближайшей точки. Вот пример того, как подобный подход мог бы выглядеть (рис. 5.55).

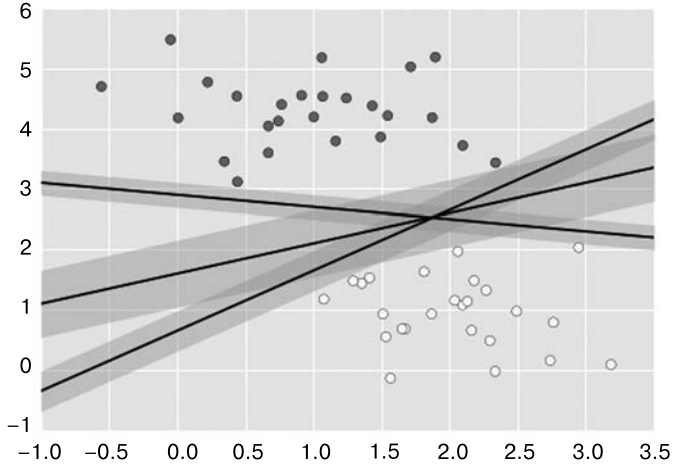

Рис. 5.55. Визуализация «отступов» в разделяющих классификаторах

```
In[4]:
xfit = np.linspace(-1, 3.5)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
```

```
for m, b, d in [(1, 0.65, 0.33), (0.5, 1.6, 0.55), (-0.2, 2.9, 0.2)]:
    yfit = m * xfit + b     plt.plot(xfit, yfit, '-k')
    plt.fill_between(xfit, yfit - d, yfit + d, edgecolor='none',
                     color='#AAAAAA', alpha=0.4)

plt.xlim(-1, 3.5);
```

В методе опорных векторов в качестве оптимальной модели выбирается линия, максимизирующая этот отступ. Метод опорных векторов — пример оценивателя *с максимальным отступом* (maximum margin estimator).

Аппроксимация методом опорных векторов

Взглянем на реальную аппроксимацию этих данных: воспользуемся классификатором на основе метода опорных векторов для обучения SVM-модели на них. Пока мы будем использовать линейное ядро и зададим очень большое значение параметра `C` (что это значит, мы расскажем далее):

```
In[5]: from sklearn.svm import SVC # "Классификатор на основе метода опорных
                                   # векторов"
       model = SVC(kernel='linear', C=1E10)
       model.fit(X, y)

Out[5]: SVC(C=10000000000.0, cache_size=200, class_weight=None, coef0=0.0,
          decision_function_shape=None, degree=3, gamma='auto',
          kernel='linear',
          max_iter=-1, probability=False, random_state=None, shrinking=True,
          tol=0.001, verbose=False)
```

Для лучшей визуализации происходящего создадим простую и удобную функцию для построения графика границ решений метода SVM (рис. 5.56):

```
In[6]:
     def plot_svc_decision_function(model, ax=None, plot_support=True):
          """Строим график решающей функции для двумерной SVC"""
          if ax is None:
              ax = plt.gca()
          xlim = ax.get_xlim()
          ylim = ax.get_ylim()

          # Создаем координатную сетку для оценки модели
          x = np.linspace(xlim[0], xlim[1], 30)
          y = np.linspace(ylim[0], ylim[1], 30)
          Y, X = np.meshgrid(y, x)
          xy = np.vstack([X.ravel(), Y.ravel()]).T
          P = model.decision_function(xy).reshape(X.shape)

          # Рисуем границы принятия решений и отступы
          ax.contour(X, Y, P, colors='k',
```

```
                    levels=[-1, 0, 1], alpha=0.5,
                    linestyles=['--', '-', '--'])

        # Рисуем опорные векторы
        if plot_support:
            ax.scatter(model.support_vectors_[:, 0],
                       model.support_vectors_[:, 1],
                       s=300, linewidth=1, facecolors='none');
        ax.set_xlim(xlim)
        ax.set_ylim(ylim)

In[7]: plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
       plot_svc_decision_function(model);
```

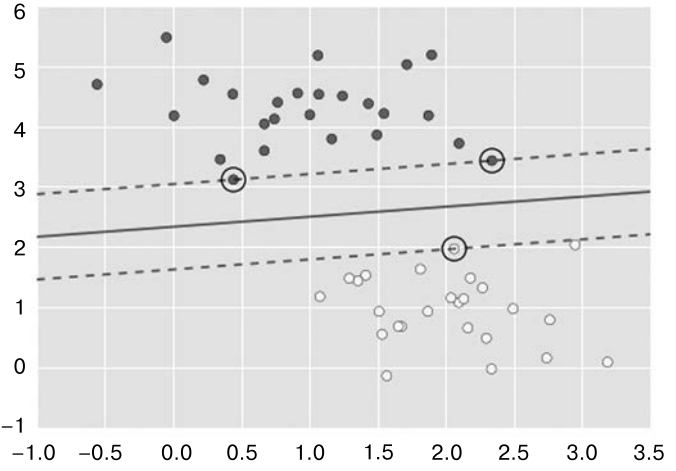

Рис. 5.56. Обучение классификатора на основе метода опорных векторов. На рисунке показаны границы отступов (штриховые линии) и опорные векторы (окружности)

Эта разделяющая линия максимизирует отступ между двумя наборами точек. Обратите внимание, что некоторые из обучающих точек лишь касаются отступа. Они отмечены на рис. 5.56 черными окружностями. Эти точки — ключевые элементы аппроксимации, они известны под названием *опорных векторов* (support vectors), в их честь алгоритм и получил свое название. В библиотеке Scikit-Learn данные об этих точках хранятся в атрибуте `support_vectors_` классификатора:

```
In[8]: model.support_vectors_

Out[8]: array([[ 0.44359863,  3.11530945],
               [ 2.33812285,  3.43116792],
               [ 2.06156753,  1.96918596]])
```

Ключ к успеху классификатора в том, что значение имеет только расположение опорных векторов. Все, находящиеся на правильной стороне, но дальше от

отступа, точки не меняют аппроксимацию! Формально дело в том, что эти точки не вносят вклада в используемую для обучения модели функцию потерь, так что их расположение и количество не имеет значения, если они не пересекают отступов.

Это можно увидеть, например, если построить график модели, обученной на первых 60 и первых 120 точках набора данных (рис. 5.57):

```
In[9]: def plot_svm(N=10, ax=None):
           X, y = make_blobs(n_samples=200, centers=2,
                             random_state=0, cluster_std=0.60)
           X = X[:N]
           y = y[:N]
           model = SVC(kernel='linear', C=1E10)
           model.fit(X, y)

           ax = ax or plt.gca()
           ax.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
           ax.set_xlim(-1, 4)
           ax.set_ylim(-1, 6)
           plot_svc_decision_function(model, ax)

       fig, ax = plt.subplots(1, 2, figsize=(16, 6))
       fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)
       for axi, N in zip(ax, [60, 120]):
           plot_svm(N, axi)
           axi.set_title('N = {0}'.format(N))
```

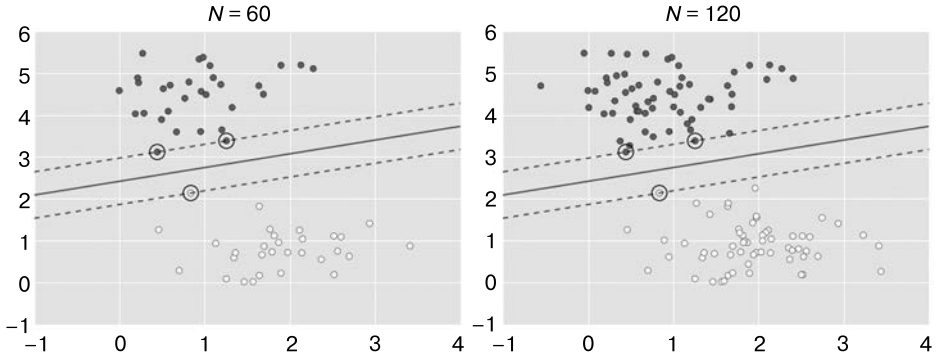

Рис. 5.57. Влияние новых обучающих точек на модель SVM

На левом рисунке мы видим модель и опорные векторы для 60 обучающих точек. На правом рисунке мы удвоили количество обучающих точек, но модель не изменилась: три опорных вектора с левого рисунка остались опорными векторами и справа. Подобная невосприимчивость к тому, как именно ведут себя удаленные точки, — одна из сильных сторон модели SVM.

Если вы работаете с этим блокнотом в интерактивном режиме, можете воспользоваться интерактивными виджетами оболочки IPython для интерактивного просмотра этой возможности модели SVM (рис. 5.58):

```
In[10]: from ipywidgets import interact, fixed
        interact(plot_svm, N=[10, 200], ax=fixed(None));
```

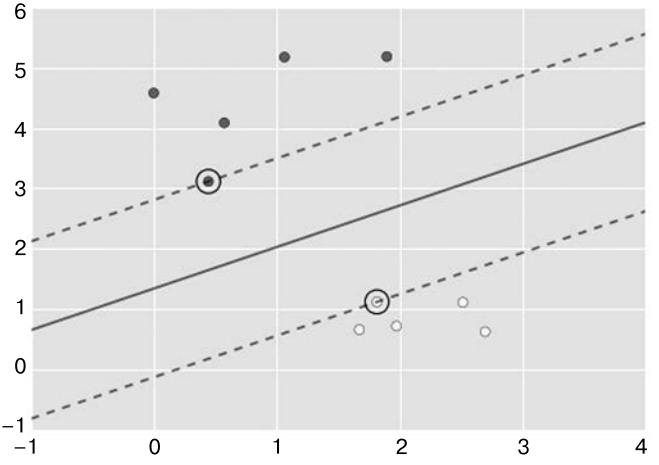

Рис. 5.58. Первый кадр интерактивной визуализации SVM (см. полную версию в онлайн-приложении (https://github.com/jakevdp/PythonDataScienceHandbook))

Выходим за границы линейности: SVM-ядро

Возможности метода SVM особенно расширяются при его комбинации с *ядрами* (kernels). Мы уже сталкивались с ними ранее, в регрессии по комбинации базисных функций из раздела «Заглянем глубже: линейная регрессия» данной главы. Там мы занимались проекцией данных в пространство с бо́льшей размерностью, определяемое полиномиальными и Гауссовыми базисными функциями, и благодаря этому имели возможность аппроксимировать нелинейные зависимости с помощью линейного классификатора.

В SVM-моделях можно использовать один из вариантов той же идеи. Чтобы понять, почему здесь необходимы ядра, рассмотрим следующие данные, которые не допускают линейное разделение (рис. 5.59):

```
In[11]: from sklearn.datasets.samples_generator import make_circles
        X, y = make_circles(100, factor=.1, noise=.1)

        clf = SVC(kernel='linear').fit(X, y)

        plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
        plot_svc_decision_function(clf, plot_support=False);
```

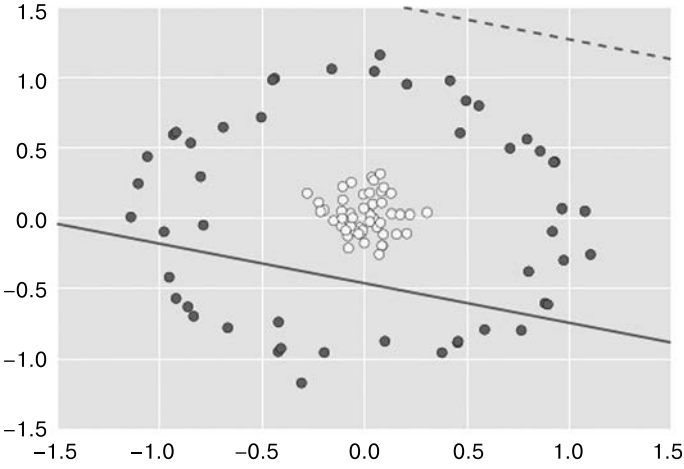

Рис. 5.59. В случае нелинейных границ линейный классификатор неэффективен

Очевидно, что эти данные *никаким образом* линейно не разделимы. Но мы можем извлечь урок из регрессии по комбинации базисных функций, которую рассмотрели в разделе «Заглянем глубже: линейная регрессия» данной главы, и попытаться спроецировать эти данные в пространство более высокой размерности, поэтому линейного разделителя *будет* достаточно. Например, одна из подходящих простых проекций — вычисление *радиальной базисной функции*, центрированной по середине совокупности данных:

```
In[12]: r = np.exp(-(X ** 2).sum(1))
```

Визуализировать это дополнительное измерение данных можно с помощью трехмерного графика. Если вы используете интерактивный блокнот, то сможете вращать график с помощью слайдеров (рис. 5.60):

```
In[13]: from mpl_toolkits import mplot3d

        def plot_3D(elev=30, azim=30, X=X, y=y):
            ax = plt.subplot(projection='3d')
            ax.scatter3D(X[:, 0], X[:, 1], r, c=y, s=50, cmap='autumn')
            ax.view_init(elev=elev, azim=azim)
            ax.set_xlabel('x')
            ax.set_ylabel('y')
            ax.set_zlabel('r')

        interact(plot_3D, elev=[-90, 90], azip=(-180, 180),
                 X=fixed(X), y=fixed(y));
```

Как видим, при наличии третьего измерения данные можно элементарно разделить линейно путем проведения разделяющей плоскости на высоте, скажем, *r = 0,7*.

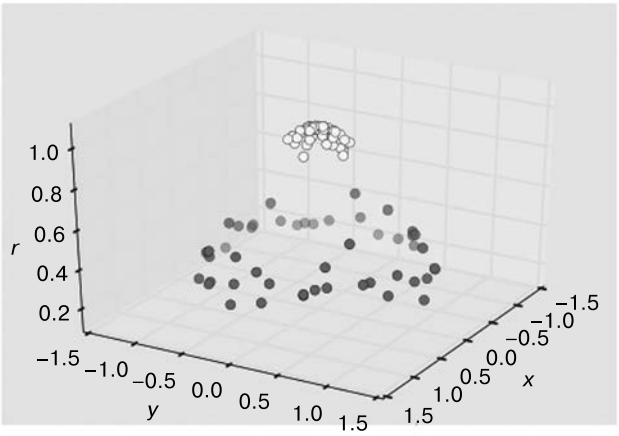

Рис. 5.60. Добавление в данные третьего измерения дает возможность линейного разделения

Нам пришлось тщательно выбрать и внимательно настроить нашу проекцию: если бы мы не центрировали радиальную базисную функцию должным образом, то не получили бы столь «чистых», разделяемых линейно результатов. Необходимость подобного выбора — задача, требующая решения: хотелось бы каким-то образом автоматически находить оптимальные базисные функции.

Одна из применяемых с этой целью стратегий состоит в вычислении базисных функций, центрированных по каждой из точек набора данных, с тем чтобы далее алгоритм SVM проанализировал полученные результаты. Эта разновидность преобразования базисных функций, известная под названием *преобразования ядра* (kernel transformation), основана на отношении подобия (или ядре) между каждой парой точек.

Потенциальная проблема с этой методикой — проекцией N точек на N измерений — состоит в том, что при росте N она может потребовать колоссальных объемов вычислений. Однако благодаря изящной процедуре, известной под названием *kernel trick* (https://en.wikipedia.org/wiki/Kernel_method), обучение на преобразованных с помощью ядра данных можно произвести неявно, то есть даже без построения полного N-мерного представления ядерной проекции! Этот kernel trick является частью SVM и одной из причин мощи этого метода.

В библиотеке Scikit-Learn, чтобы применить алгоритм SVM с использованием ядерного преобразования, достаточно просто заменить линейное ядро на ядро RBF (radial basis function — «радиальная базисная функция») с помощью гиперпараметра модели `kernel` (рис. 5.61):

```
In[14]: clf = SVC(kernel='rbf', C=1E6)
        clf.fit(X, y)
```

```
Out[14]: SVC(C=1000000.0, cache_size=200, class_weight=None, coef0=0.0,
           decision_function_shape=None, degree=3, gamma='auto', kernel='rbf',
           max_iter=-1, probability=False, random_state=None, shrinking=True,
           tol=0.001, verbose=False)
```

```
In[15]: plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
        plot_svc_decision_function(clf)
        plt.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1],
                    s=300, lw=1, facecolors='none');
```

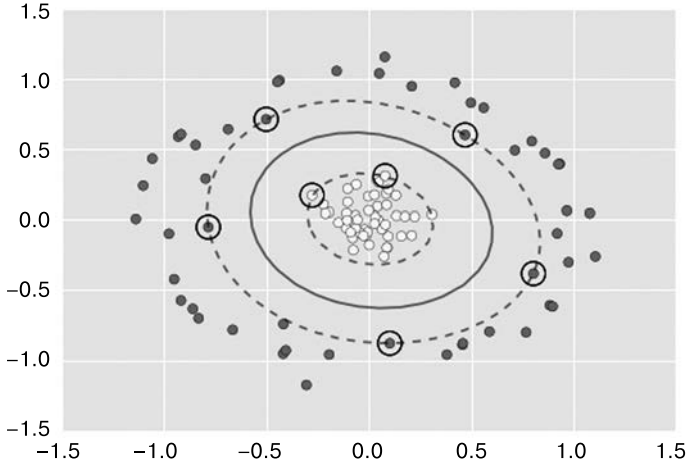

Рис. 5.61. Обучение ядерного SVM на наших данных

С помощью этого ядерного метода опорных векторов мы можем определить подходящую нелинейную границу решений. Такая методика ядерного преобразования часто используется в машинном обучении для превращения быстрых линейных методов в быстрые нелинейные, особенно для моделей, в которых можно воспользоваться kernel trick.

Настройка SVM: размытие отступов

До сих пор наше обсуждение касалось хорошо очищенных наборов данных, в которых существует идеальная граница решений. Но что, если данные в некоторой степени перекрываются?

Например, допустим, мы имеем дело со следующими данными (рис. 5.62):

```
In[16]: X, y = make_blobs(n_samples=100, centers=2,
                          random_state=0, cluster_std=1.2)
        plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn');
```

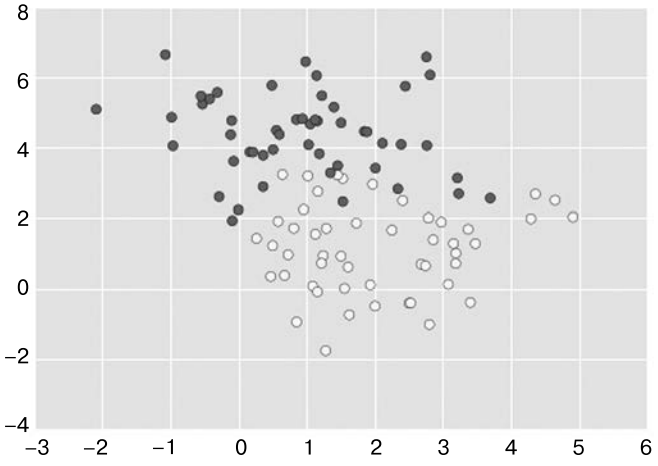

Рис. 5.62. Данные с определенным перекрытием

На этот случай в реализации метода SVM есть небольшой поправочный параметр для «размытия» отступа. Данный параметр разрешает некоторым точкам «заходить» на отступ в тех случаях, когда это приводит к лучшей аппроксимации. Степень размытости отступа управляется настроечным параметром, известным под названием C. При очень большом значении параметра C отступ является «жестким» и точки не могут находиться на нем. При меньшем значении параметра C отступ становится более размытым и может включать в себя некоторые точки.

На рис. 5.63 показана наглядная картина того, какое влияние изменение параметра C оказывает на итоговую аппроксимацию посредством размытия отступа.

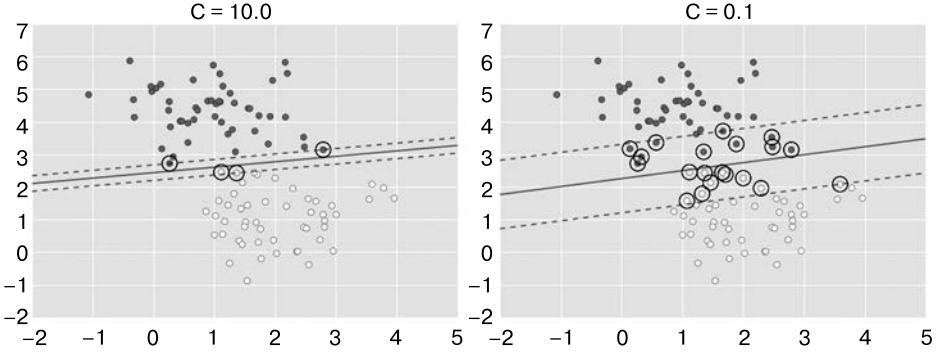

Рис. 5.63. Влияние параметра C на аппроксимацию методом опорных векторов

```
In[17]: X, y = make_blobs(n_samples=100, centers=2,
                          random_state=0, cluster_std=0.8)

        fig, ax = plt.subplots(1, 2, figsize=(16, 6))
```

```
    fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)

    for axi, C in zip(ax, [10.0, 0.1]):
        model = SVC(kernel='linear', C=C).fit(X, y)
        axi.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
        plot_svc_decision_function(model, axi)
        axi.scatter(model.support_vectors_[:, 0],
                    model.support_vectors_[:, 1],
                    s=300, lw=1, facecolors='none');
        axi.set_title('C = {0:.1f}'.format(C), size=14)
```

Оптимальное значение параметра `C` зависит от конкретного набора данных. Его следует настраивать с помощью перекрестной проверки или какой-либо аналогичной процедуры (дальнейшую информацию см. в разделе «Гиперпараметры и проверка модели» данной главы).

Пример: распознавание лиц

В качестве примера работы метода опорных векторов рассмотрим задачу распознавания лиц. Мы воспользуемся набором данных Labeled Faces in the Wild[1] (LFW), состоящим из нескольких тысяч упорядоченных фотографий различных общественных деятелей. В библиотеку Scikit-Learn встроена утилита для загрузки этого набора данных:

```
In[18]: from sklearn.datasets import fetch_lfw_people
        faces = fetch_lfw_people(min_faces_per_person=60)
        print(faces.target_names)
        print(faces.images.shape)

['Ariel Sharon' 'Colin Powell' 'Donald Rumsfeld' 'George W Bush'
 'Gerhard Schroeder' 'Hugo Chavez' 'Junichiro Koizumi' 'Tony Blair']
(1348, 62, 47)
```

Выведем на рисунок несколько из этих лиц, чтобы увидеть, с чем мы будем иметь дело (рис. 5.64):

```
In[19]: fig, ax = plt.subplots(3, 5)
        for i, axi in enumerate(ax.flat):
            axi.imshow(faces.images[i], cmap='bone')
            axi.set(xticks=[], yticks=[],
                    xlabel=faces.target_names[faces.target[i]])
```

Каждое изображение содержит 62×47, то есть примерно 3000 пикселов. Мы можем рассматривать каждый пиксел как признак, но эффективнее использовать какой-либо препроцессор для извлечения более осмысленных признаков. В данном случае мы воспользуемся методом главных компонент (см. раздел «Заглянем глубже: метод главных компонент» данной главы) для извлечения 150 базовых компонент,

[1] См.: http://vis-www.cs.umass.edu/lfw/

которые мы передадим нашему классификатору на основе метода опорных векторов. Упростим эту задачу, объединив препроцессор и классификатор в единый конвейер:

```
In[20]: from sklearn.svm import SVC
        from sklearn.decomposition import RandomizedPCA
        from sklearn.pipeline import make_pipeline

        pca = RandomizedPCA(n_components=150, whiten=True, random_state=42)
        svc = SVC(kernel='rbf', class_weight='balanced')
        model = make_pipeline(pca, svc)
```

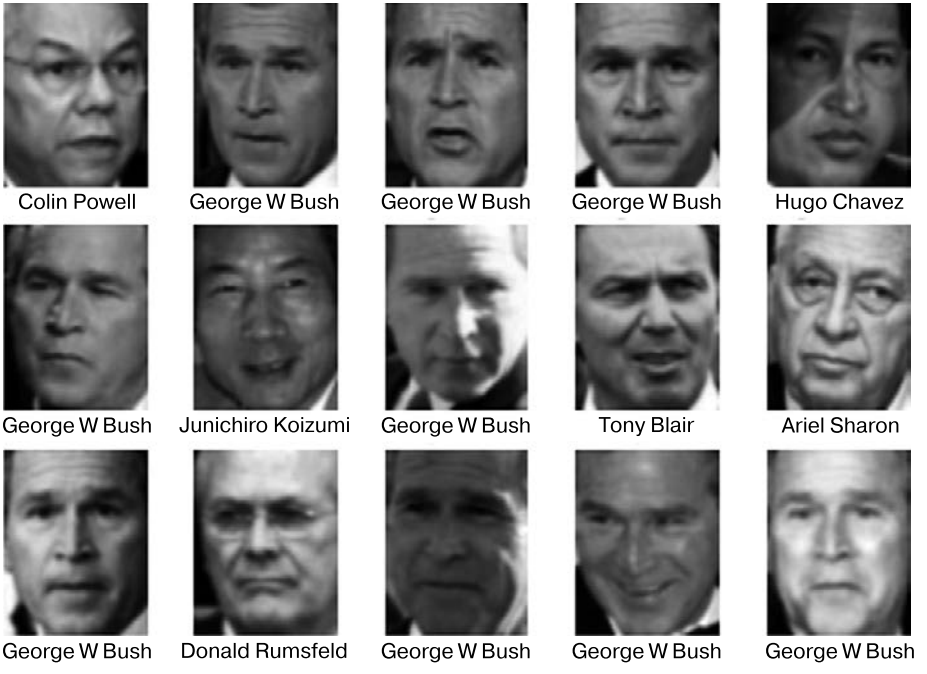

Рис. 5.64. Примеры из набора данных Labeled Faces in the Wild

Для контроля результатов работы нашего классификатора разобьем данные на обучающую и контрольную последовательности:

```
In[21]: from sklearn.cross_validation import train_test_split
        Xtrain, Xtest, ytrain, ytest = train_test_split(faces.data,
                                                        faces.target,
                                                        random_state=42)
```

Наконец, воспользуемся поиском по сетке с перекрестной проверкой для анализа сочетаний параметров. Подберем значения параметров `C` (управляющего размытием отступов) и `gamma` (управляющего размером ядра радиальной базисной функции) и определим оптимальную модель:

```
In[22]: from sklearn.grid_search import GridSearchCV
        param_grid = {'svc__C': [1, 5, 10, 50],
                      'svc__gamma': [0.0001, 0.0005, 0.001, 0.005]}
        grid = GridSearchCV(model, param_grid)
        %time grid.fit(Xtrain, ytrain)
        print(grid.best_params_)

CPU times: user 47.8 s, sys: 4.08 s, total: 51.8 s
Wall time: 26 s
{'svc__gamma': 0.001, 'svc__C': 10}
```

Оптимальные значения приходятся на середину нашей сетки. Если бы они приходились на края сетки, то желательно было бы расширить сетку, чтобы убедиться в нахождении истинного оптимума.

Теперь с помощью этой, подвергнутой перекрестной проверке модели можно предсказать метки для контрольных данных, которые модель еще не видела:

```
In[23]: model = grid.best_estimator_
        yfit = model.predict(Xtest)
```

Рассмотрим некоторые из контрольных изображений и предсказанных для них значений (рис. 5.65):

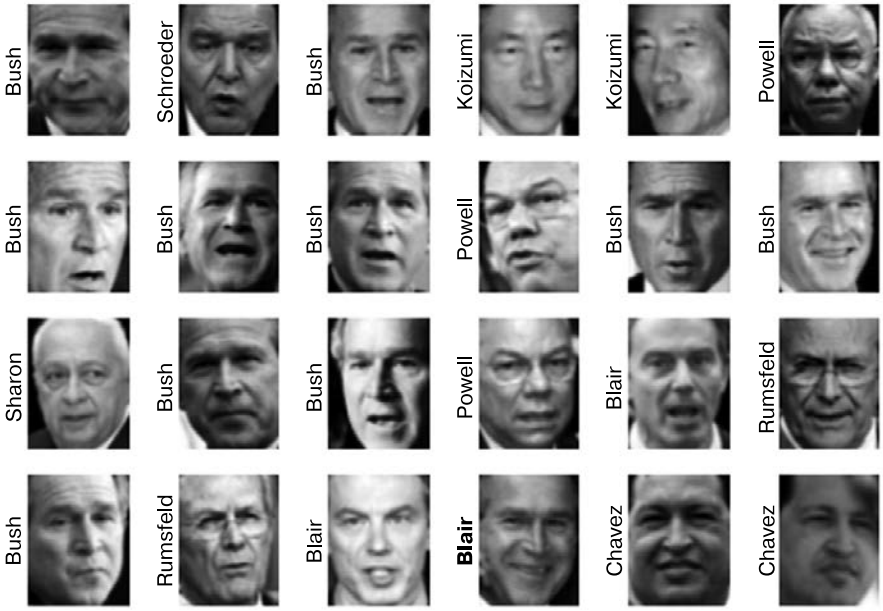

Рис. 5.65. Прогнозируемые имена. Неверные метки выделены полужирным

```
In[24]: fig, ax = plt.subplots(4, 6)
        for i, axi in enumerate(ax.flat):
            axi.imshow(Xtest[i].reshape(62, 47), cmap='bone')
            axi.set(xticks=[], yticks=[])
            axi.set_ylabel(faces.target_names[yfit[i]].split()[-1],
                           color='black' if yfit[i] == ytest[i] else 'red')
        fig.suptitle('Predicted Names; Incorrect Labels in Red', size=14);
```

В этой небольшой выборке наш оптимальный оцениватель ошибся только для одного лица (лицо Дж. Буша в нижнем ряду было ошибочно помечено как лицо Блэра). Чтобы лучше прочувствовать эффективность работы нашего оценивателя, воспользуемся отчетом о классификации, в котором приведена статистика восстановления значений по каждой метке:

```
In[25]: from sklearn.metrics import classification_report
        print(classification_report(ytest, yfit,
                                    target_names=faces.target_names))

                   precision    recall  f1-score   support

     Ariel Sharon       0.65      0.73      0.69        15
     Colin Powell       0.81      0.87      0.84        68
  Donald Rumsfeld       0.75      0.87      0.81        31
    George W Bush       0.93      0.83      0.88       126
Gerhard Schroeder       0.86      0.78      0.82        23
      Hugo Chavez       0.93      0.70      0.80        20
Junichiro Koizumi       0.80      1.00      0.89        12
       Tony Blair       0.83      0.93      0.88        42

      avg / total       0.85      0.85      0.85       337
```

Можем также вывести на экран матрицу различий между этими классами (рис. 5.66):

```
In[26]: from sklearn.metrics import confusion_matrix
        mat = confusion_matrix(ytest, yfit)
        sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False,
                    xticklabels=faces.target_names,
                    yticklabels=faces.target_names)
        plt.xlabel('true label')
        plt.ylabel('predicted label');
```

Эта информация позволяет нам понять, какие метки, вероятно, оцениватель определит неверно.

В реальных задачах распознавания лиц, в которых фотографии не кадрированы предварительно в аккуратные сетки, единственное отличие в схеме классификации лиц будет состоять в выборе признаков. Необходимо будет использовать более сложный алгоритм для поиска лиц и извлекать не зависящие от пикселизации признаки. Для подобных приложений удобно применять библиотеку OpenCV (http://opencv.org/), которая, помимо прочего, включает заранее обученные реализации современных инструментов выделения признаков для изображений вообще и лиц в частности.

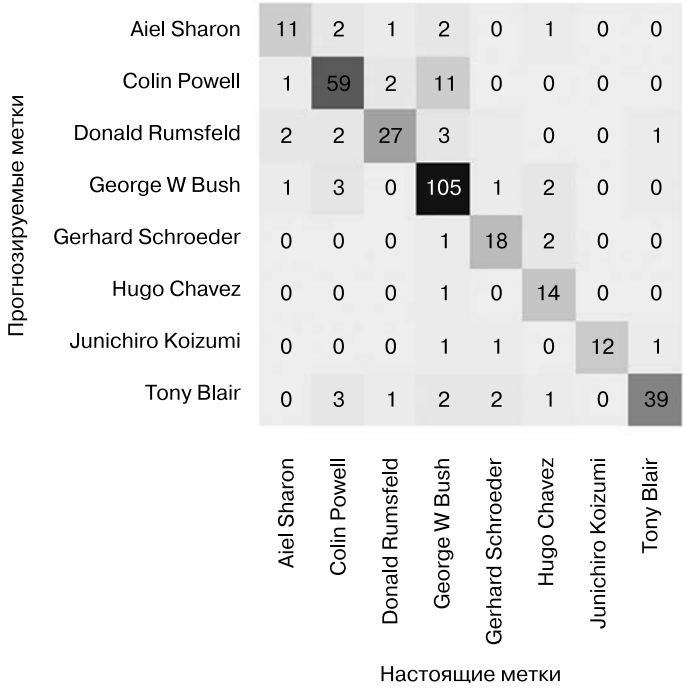

Рис. 5.66. Матрица различий для данных по лицам

Резюме по методу опорных векторов

В этом разделе мы привели краткое и понятное введение в основы методов опорных векторов. Эти методы являются мощными методами классификации по ряду причин.

- Зависимость их от относительно небольшого количества опорных векторов означает компактность модели и небольшое количество используемой оперативной памяти.
- Фаза предсказания после обучения модели занимает очень мало времени.
- Поскольку на работу этих методов влияют только точки, находящиеся возле отступа, они хорошо подходят для многомерных данных — даже данных с количеством измерений бóльшим, чем количество выборок, — непростые условия работы для других алгоритмов.
- Интеграция с ядерными методами делает их универсальными, обеспечивает приспособляемость к множеству типов данных.

Однако у методов опорных векторов есть и несколько недостатков.

- Они масштабируются при количестве выборок N в наихудшем случае как $O[N^3]$ ($O[N^2]$ для более эффективных реализаций). При значительном количестве

обучающих выборок вычислительные затраты могут оказаться непомерно высокими.

- Результаты зависят от удачности выбора параметра размытия `C`. Его необходимо тщательно выбирать с помощью перекрестной проверки, которая тоже может потребовать значительных вычислительных затрат при росте размеров наборов данных.
- У результатов отсутствует непосредственная вероятностная интерпретация. Ее можно получить путем внутренней перекрестной проверки (см. параметр `probability` оценивателя `SVC`), но эта дополнительная оценка обходится недешево в смысле вычислительных затрат.

Учитывая эти особенности, я обращаюсь к SVM только тогда, когда более простые, быстрые и требующие меньшего количества настроек методы не удовлетворяют моим потребностям. Тем не менее, если у вас есть достаточно процессорного времени для выполнения обучения и перекрестной проверки SVM на ваших данных, этот метод может показать превосходные результаты.

Заглянем глубже: деревья решений и случайные леса

Мы подробно рассмотрели простой порождающий классификатор (см. раздел «Заглянем глубже: наивная байесовская классификация» данной главы) и обладающий широкими возможностями разделяющий классификатор (см. раздел «Заглянем глубже: метод опорных векторов» этой главы). В данном разделе мы рассмотрим еще один мощный непараметрический алгоритм — *случайные леса* (random forests). Случайные леса — пример одного из методов *ансамблей* (ensemble), основанных на агрегировании результатов ансамбля более простых оценивателей. Несколько неожиданный результат использования подобных методов ансамблей — то, что целое в данном случае оказывается больше суммы составных частей. Результат «голосования» среди достаточного количества оценивателей может оказаться лучше результата любого из отдельных участников «голосования»! Мы увидим примеры этого в следующих разделах. Начнем с обычных импортов:

```
In[1]: %matplotlib inline
       import numpy as np
       import matplotlib.pyplot as plt
       import seaborn as sns; sns.set()
```

Движущая сила случайных лесов: деревья принятия решений

Случайные леса — пример обучаемого ансамбля на основе деревьев принятия решений. Поэтому мы начнем с обсуждения самих деревьев решений.

Деревья решений — исключительно интуитивно понятные способы классификации или маркирования объектов. По сути, все сводится к классификации путем задания серии уточняющих вопросов. Например, дерево принятия решений для классификации встретившихся во время велосипедной прогулки животных могло бы выглядеть следующим образом (рис. 5.67):

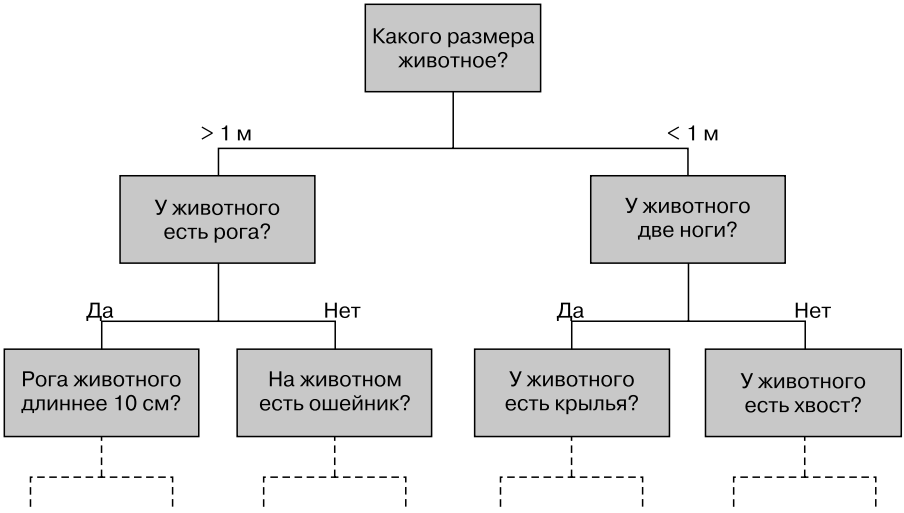

Рис. 5.67. Пример двоичного дерева принятия решений

Бинарное разбиение чрезвычайно эффективно: в хорошо спроектированном дереве каждый вопрос будет уменьшать количество вариантов приблизительно вдвое, очень быстро сужая возможные варианты даже при большом количестве классов. Фокус состоит в том, какие вопросы задавать на каждом шаге. В связанных с машинным обучением реализациях деревьев принятия решений вопросы обычно имеют вид выровненных по осям координат разбиений данных, то есть каждый узел дерева разбивает данные на две группы с помощью порогового значения одного из признаков.

Создание дерева принятия решений

Рассмотрим следующие двумерные данные с четырьмя возможными метками классов (рис. 5.68):

```
In[2]: from sklearn.datasets import make_blobs

       X, y = make_blobs(n_samples=300, centers=4,
                         random_state=0, cluster_std=1.0)
       plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='rainbow');
```

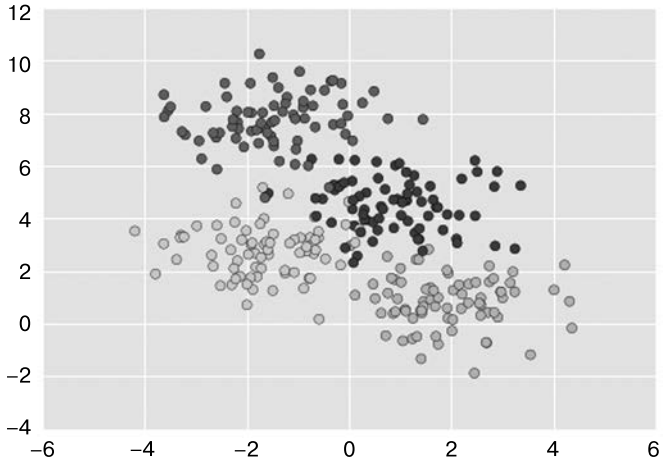

Рис. 5.68. Данные для классификатора на основе дерева принятия решений

Простое дерево принятия решений для этих данных будет многократно разделять данные по одной или нескольким осям, в соответствии с определенным количественным критерием, и на каждом уровне маркировать новую область согласно большинству лежащих в ней точек. На рис. 5.69 приведена визуализация первых четырех уровней классификатора для этих данных, созданного на основе дерева принятия решений.

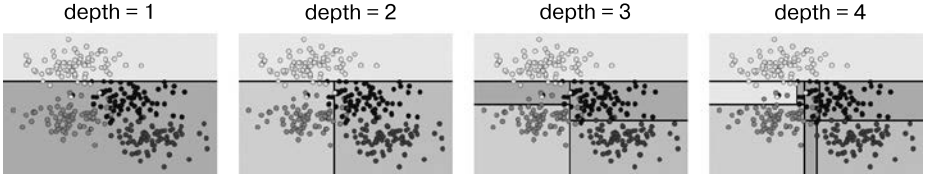

Рис. 5.69. Визуализация разбиения данных деревом принятия решений

Обратите внимание, что после первого разбиения все точки в верхней ветке остаются неизменными, поэтому необходимости в дальнейшем ее разбиении нет. За исключением узлов, в которых присутствует только один цвет, на каждом из уровней *все* области снова разбиваются по одному из двух признаков.

Процесс обучения дерева принятия решений на наших данных можно выполнить в Scikit-Learn с помощью оценивателя `DecisionTreeClassifier`:

```
In[3]: from sklearn.tree import DecisionTreeClassifier
       tree = DecisionTreeClassifier().fit(X, y)
```

Напишем небольшую вспомогательную функцию, чтобы облегчить визуализацию вывода классификатора:

```
In[4]:
       def visualize_classifier(model, X, y, ax=None, cmap='rainbow'):
           ax = ax or plt.gca()

           # Рисуем обучающие точки
           ax.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap=cmap,
                      clim=(y.min(), y.max()), zorder=3)
           ax.axis('tight')
           ax.axis('off')
           xlim = ax.get_xlim()
           ylim = ax.get_ylim()

           # Обучаем оцениватель
           model.fit(X, y)
           xx, yy = np.meshgrid(np.linspace(*xlim, num=200),
                                np.linspace(*ylim, num=200))
           Z = model.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

           # Создаем цветной график с результатами
           n_classes = len(np.unique(y))
           contours = ax.contourf(xx, yy, Z, alpha=0.3,
                                  levels=np.arange(n_classes + 1) - 0.5,
                                  cmap=cmap, clim=(y.min(), y.max()),
                                  zorder=1)

           ax.set(xlim=xlim, ylim=ylim)
```

Теперь можно посмотреть, какая у нас получилась классификация на основе дерева принятия решений (рис. 5.70):

```
In[5]: visualize_classifier(DecisionTreeClassifier(), X, y)
```

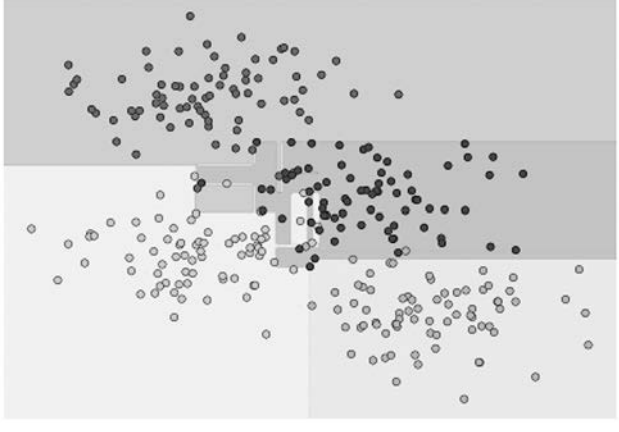

Рис. 5.70. Визуализация классификации на основе дерева принятия решений

Если вы применяете интерактивный блокнот, то можете воспользоваться вспомогательным сценарием, включенным в онлайн-приложение (https://github.com/jakevdp/PythonDataScienceHandbook) для вызова интерактивной визуализации процесса построения дерева принятия решений (рис. 5.71):

```
In[6]: # Модуль helpers_05_08 можно найти в онлайн-приложении к книге
       # (https://github.com/jakevdp/PythonDataScienceHandbook)
       import helpers_05_08
       helpers_05_08.plot_tree_interactive(X, y);
```

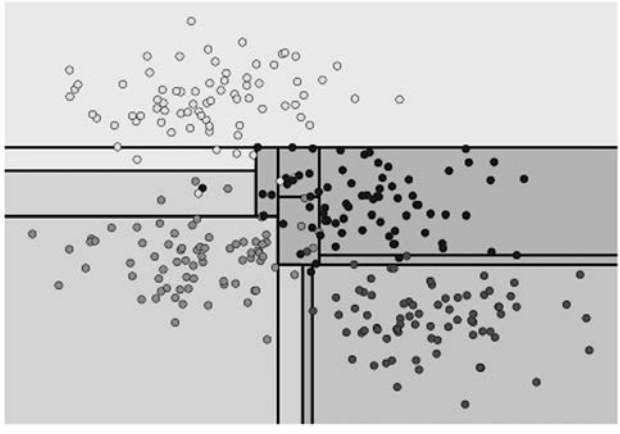

Рис. 5.71. Первый кадр интерактивного виджета дерева принятия решений. Полную версию см. в онлайн-приложении (https://github.com/jakevdp/PythonDataScienceHandbook)

Обратите внимание, что по мере возрастания глубины мы получаем области классификации очень странной формы. Например, на глубине, равной 5, между желтой и синей областями появляется узкая и вытянутая в высоту фиолетовая область. Очевидно, что такая конфигурация является скорее не результатом собственно распределения данных, а конкретной их дискретизации или свойств шума в них. То есть это дерево принятия решений на глубине всего лишь пяти уровней уже очевидным образом переобучено.

Деревья принятия решений и переобучение

Подобное переобучение присуще всем деревьям принятия решений: нет ничего проще, чем дойти до слишком глубокого уровня дерева, аппроксимируя таким образом нюансы конкретных данных вместо общих характеристик распределений, из которых они получены. Другой способ увидеть это переобучение — обратиться к моделям, обученным на различных подмножествах набора данных, например, на рис. 5.72 показано обучение двух различных деревьев, каждое на половине исходного набора данных.

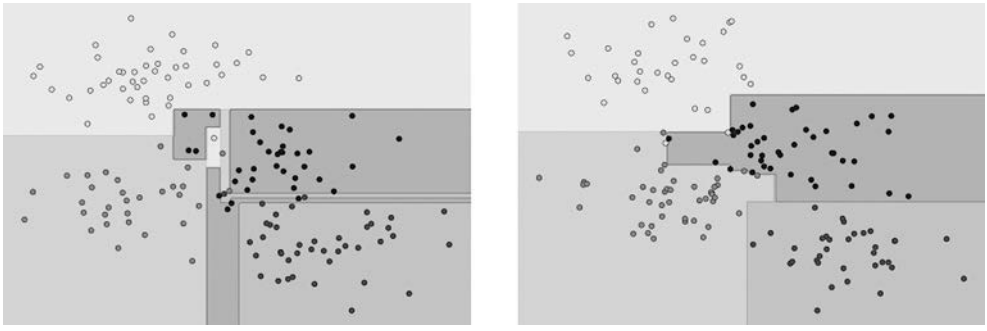

Рис. 5.72. Пример двух случайных деревьев принятия решений

Очевидно, что в некоторых местах результаты этих двух деревьев не противоречат друг другу (например, в четырех углах), а в других местах их классификации очень сильно различаются (например, в областях между любыми двумя кластерами). Важнейший вывод из этого рисунка: расхождения имеют тенденцию появляться в тех местах, где степень достоверности классификации ниже, а значит, мы можем добиться лучшего результата, используя информацию из *обоих* деревьев!

При применении интерактивного блокнота можно воспользоваться следующей функцией для интерактивного отображения деревьев, обученных на случайных подмножествах набора данных (рис. 5.73):

```
In[7]: # Модуль helpers_05_08 можно найти в онлайн-приложении к книге
       # (https://github.com/jakevdp/PythonDataScienceHandbook)
       import helpers_05_08
       helpers_05_08.randomized_tree_interactive(X, y)
```

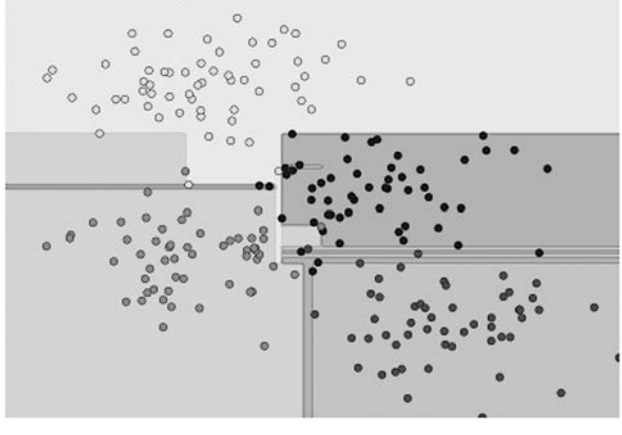

Рис. 5.73. Первый кадр интерактивного виджета случайного дерева принятия решений. Полную версию см. в онлайн-приложении (https://github.com/jakevdp/PythonDataScienceHandbook)

Аналогично тому, как использование информации из двух деревьев позволяет достичь лучшего результата, применение информации из многих обеспечит еще лучший результат.

Ансамбли оценивателей: случайные леса

Идея комбинации нескольких переобученных оценивателей для снижения эффекта этого переобучения лежит в основе метода ансамблей под названием *«баггинг»* (bagging). Баггинг использует ансамбль (например, своеобразную «шляпу фокусника») параллельно работающих переобучаемых оценивателей и усредняет результаты для получения оптимальной классификации. Ансамбль случайных деревьев принятия решений называется *случайным лесом* (random forest).

Выполнить подобную баггинг-классификацию можно вручную с помощью метаоценивателя `BaggingClassifier` из библиотеки Scikit-Learn, как показано на рис. 5.74:

```
In[8]: from sklearn.tree import DecisionTreeClassifier
       from sklearn.ensemble import BaggingClassifier

       tree = DecisionTreeClassifier()
       bag = BaggingClassifier(tree, n_estimators=100, max_samples=0.8,
                               random_state=1)
       bag.fit(X, y)
       visualize_classifier(bag, X, y)
```

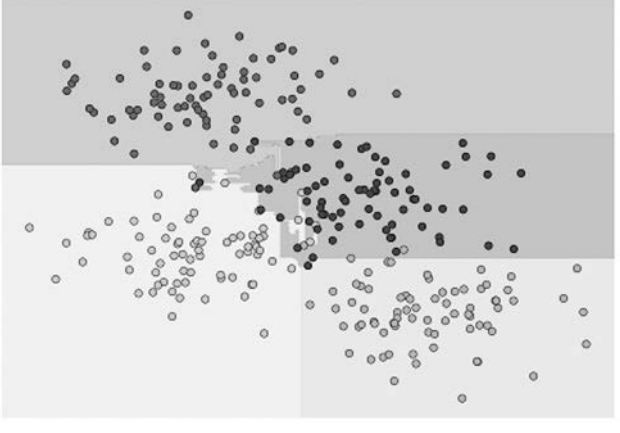

Рис. 5.74. Границы принятия решений для ансамбля случайных деревьев решений

В этом примере мы рандомизировали данные путем обучения всех оценивателей на случайном подмножестве, состоящем из 80 % обучающих точек. На практике

для более эффективной рандомизации деревьев принятия решений обеспечивается определенная стохастичность процесса выбора разбиений. При этом всякий раз в обучении участвуют все данные, но результаты обучения все равно сохраняют требуемую случайность. Например, при выборе, по какому признаку выполнять разбиение, случайное дерево может выбирать из нескольких верхних признаков. Узнать больше технических подробностей о стратегиях рандомизации можно в документации библиотеки Scikit-Learn (http://scikit-learn.org/stable/modules/ensemble.html#forest) и упомянутых в ней справочных руководствах.

В библиотеке Scikit-Learn подобный оптимизированный ансамбль случайных деревьев принятия решений, автоматически выполняющий всю рандомизацию, реализован в оценивателе `RandomForestClassifier`. Все, что остается сделать, — выбрать количество оценивателей и он очень быстро (при необходимости параллельно) обучит ансамбль деревьев (рис. 5.75):

```
In[9]: from sklearn.ensemble import RandomForestClassifier

       model = RandomForestClassifier(n_estimators=100, random_state=0)
       visualize_classifier(model, X, y);
```

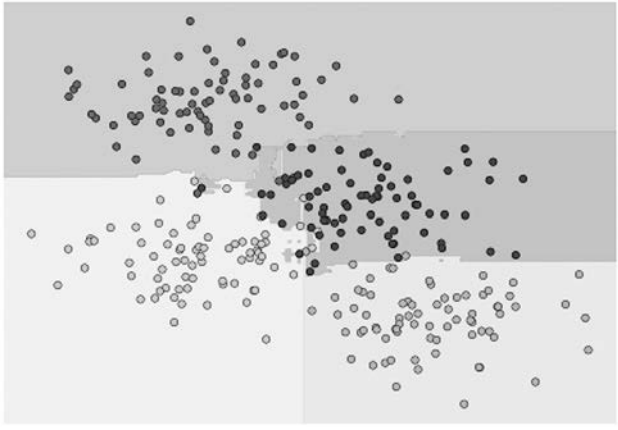

Рис. 5.75. Ансамбль случайных деревьев принятия решений

Как видим, путем усреднения более чем 100 случайно возмущенных моделей мы получаем общую модель, намного более близкую к нашим интуитивным представлениям о правильном разбиении параметрического пространства.

Регрессия с помощью случайных лесов

В предыдущем разделе мы рассмотрели случайные леса в контексте классификации. Случайные леса могут также оказаться полезными для регрессии (то есть непрерывных, а не категориальных величин). В этом случае используется оцениватель `RandomForestRegressor`, синтаксис которого напоминает показанный выше.

Рассмотрим данные, полученные из сочетания быстрых и медленных колебаний (рис. 5.76):

```
In[10]: rng = np.random.RandomState(42)
        x = 10 * rng.rand(200)

        def model(x, sigma=0.3):
            fast_oscillation = np.sin(5 * x)
            slow_oscillation = np.sin(0.5 * x)
            noise = sigma * rng.randn(len(x))

            return slow_oscillation + fast_oscillation + noise

        y = model(x)
        plt.errorbar(x, y, 0.3, fmt='o');
```

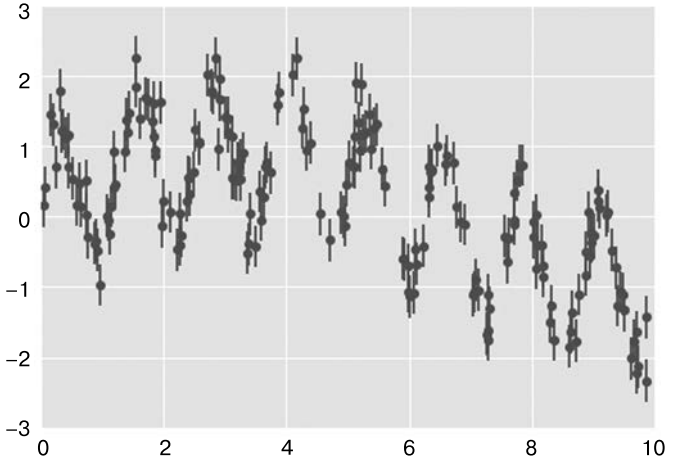

Рис. 5.76. Данные для регрессии с помощью случайного леса

Найти оптимальную аппроксимирующую кривую с помощью регрессии на основе случайного леса можно следующим образом (рис. 5.77):

```
In[11]: from sklearn.ensemble import RandomForestRegressor
        forest = RandomForestRegressor(200)
        forest.fit(x[:, None], y)

        xfit = np.linspace(0, 10, 1000)
        yfit = forest.predict(xfit[:, None])
        ytrue = model(xfit, sigma=0)

        plt.errorbar(x, y, 0.3, fmt='o', alpha=0.5)
        plt.plot(xfit, yfit, '-r');
        plt.plot(xfit, ytrue, '-k', alpha=0.5);
```

На этом рисунке настоящая модель показана гладкой, а модель на основе случайного леса — «рваной» кривой. Как вы можете видеть, непараметрическая модель на основе случайного леса достаточно гибка для аппроксимации мультипериодических данных, без необходимости использования мультипериодической модели!

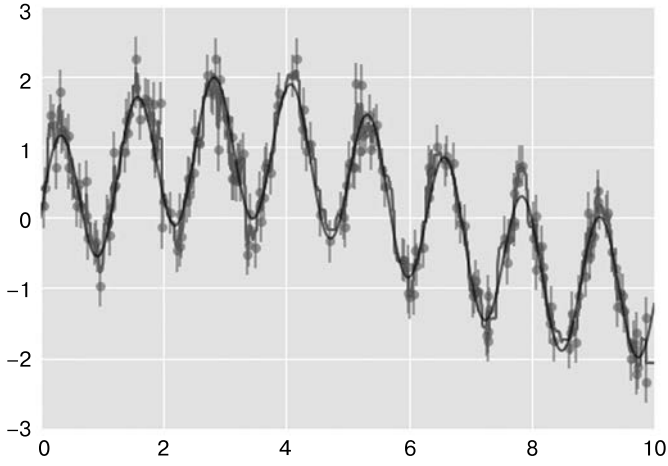

Рис. 5.77. Аппроксимация данных моделью на основе случайного леса

Пример: использование случайного леса для классификации цифр

Ранее мы уже видели данные по рукописным цифрам (см. раздел «Знакомство с библиотекой Scikit-Learn» этой главы). Воспользуемся ими снова, чтобы посмотреть на применение классификатора на основе случайных лесов в данном контексте.

```
In[12]: from sklearn.datasets import load_digits
        digits = load_digits()
        digits.keys()

Out[12]: dict_keys(['target', 'data', 'target_names', 'DESCR', 'images'])
```

В качестве напоминания, с чем мы имеем дело, визуализируем несколько первых точек данных (рис. 5.78):

```
In[13]:
# Настройки рисунка
fig = plt.figure(figsize=(6, 6))  # размер рисунка в дюймах
fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.05,
wspace=0.05)

# Рисуем цифры: размер каждого изображения 8 x 8 пикселов
for i in range(64):
    ax = fig.add_subplot(8, 8, i + 1, xticks=[], yticks=[])
    ax.imshow(digits.images[i], cmap=plt.cm.binary, interpolation='nearest')
```

```
    # Маркируем изображение целевыми значениями
    ax.text(0, 7, str(digits.target[i]))
```

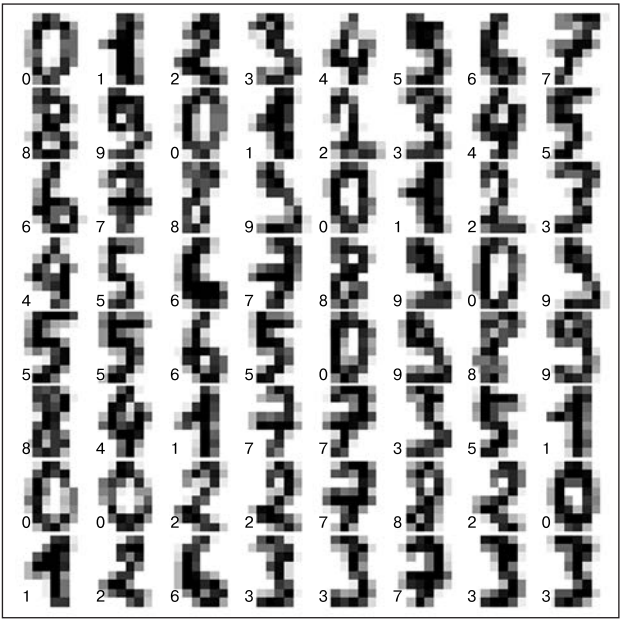

Рис. 5.78. Визуальное представление данных по рукописным цифрам

Быстро классифицировать цифры с помощью случайного леса можно следующим образом (рис. 5.79):

```
In[14]:
from sklearn.cross_validation import train_test_split

Xtrain, Xtest, ytrain, ytest = train_test_split(digits.data, digits.target,
                                                random_state=0)
model = RandomForestClassifier(n_estimators=1000)
model.fit(Xtrain, ytrain)
ypred = model.predict(Xtest)
```

Взглянем на отчет о классификации для данного классификатора:

```
In[15]: from sklearn import metrics
        print(metrics.classification_report(ypred, ytest))

             precision    recall  f1-score   support
          0       1.00      0.97      0.99        38
          1       1.00      0.98      0.99        44
          2       0.95      1.00      0.98        42
```

```
          3       0.98      0.96      0.97        46
          4       0.97      1.00      0.99        37
          5       0.98      0.96      0.97        49
          6       1.00      1.00      1.00        52
          7       1.00      0.96      0.98        50
          8       0.94      0.98      0.96        46
          9       0.96      0.98      0.97        46

avg / total       0.98      0.98      0.98       450
```

В дополнение нарисуем матрицу различий (см. рис. 5.79):

```
In[16]: from sklearn.metrics import confusion_matrix
        mat = confusion_matrix(ytest, ypred)
        sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False)
        plt.xlabel('true label')
        plt.ylabel('predicted label');
```

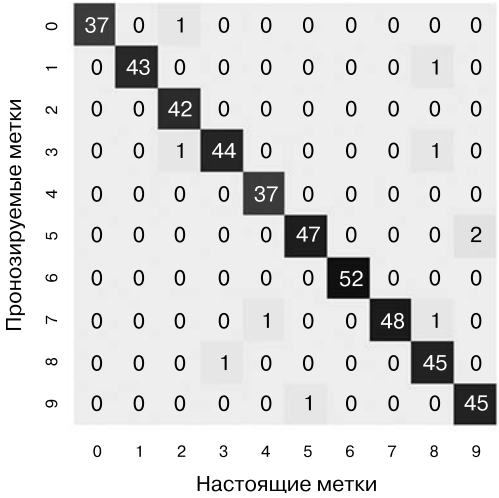

Рис. 5.79. Матрица различий для классификации цифр с помощью случайных лесов

Оказалось, что простой, не настроенный каким-то специальным образом случайный лес дает очень точную классификацию данных по рукописным цифрам.

Резюме по случайным лесам

В этом разделе мы познакомили вас с понятием *ансамблей оценивателей* (ensemble estimators) и, в частности, модели случайного леса — ансамбля случайных деревьев

принятия решений. Случайные леса — мощный метод, обладающий несколькими достоинствами.

- Как обучение, так и предсказание выполняются очень быстро в силу простоты лежащих в основе модели деревьев принятия решений. Кроме того, обе задачи допускают эффективную параллелизацию, так как отдельные деревья представляют собой совершенно независимые сущности.
- Вариант с несколькими деревьями дает возможность использования вероятностной классификации: решение путем «голосования» оценивателей дает оценку вероятности (в библиотеке Scikit-Learn ее можно получить с помощью метода `predict_proba()`).
- Непараметрическая модель исключительно гибка и может эффективно работать с задачами, на которых другие оцениватели оказываются недообученными.

Основной недостаток случайных лесов состоит в том, что результаты сложно интерпретировать.Чтобы сделать какие-либо выводы относительно *смысла* модели классификации, случайные леса — не лучший вариант.

Заглянем глубже: метод главных компонент

До сих пор мы подробно изучали оцениватели для машинного обучения с учителем, предсказывающие метки на основе маркированных обучающих данных. Здесь же мы изучим несколько оценивателей без учителя, позволяющих подчеркнуть интересные аспекты данных безотносительно каких-либо известных меток.

Мы обсудим, возможно, один из наиболее широко используемых алгоритмов машинного обучения без учителя — метод главных компонент (principal component analysis, PCA). PCA представляет собой алгоритм понижения размерности, но он может быть также удобен в качестве инструмента визуализации, фильтрации шума, выделения и проектирования признаков, а также многого другого. После краткого концептуального обзора алгоритма PCA мы рассмотрим несколько примеров прикладных задач. Начнем с обычных импортов:

```
In[1]: %matplotlib inline
       import numpy as np
       import matplotlib.pyplot as plt
       import seaborn as sns; sns.set()
```

Знакомство с методом главных компонент

Метод главных компонент — быстрый и гибкий метод машинного обучения без учителя, предназначенный для понижения размерности данных. Мы познакомились с ним в разделе «Знакомство с библиотекой Scikit-Learn» этой главы. Легче

всего визуализировать его поведение на примере двумерного набора данных. Рассмотрим следующие 200 точек (рис. 5.80):

```
In[2]: rng = np.random.RandomState(1)
       X = np.dot(rng.rand(2, 2), rng.randn(2, 200)).T
       plt.scatter(X[:, 0], X[:, 1])
       plt.axis('equal');
```

Визуально очевидно, что зависимость между величинами x и y практически линейна. Это напоминает данные линейной регрессии, которые мы изучали в разделе «Заглянем глубже: линейная регрессия» этой главы, но постановка задачи здесь несколько иная: задача машинного обучения без учителя состоит в выяснении *зависимости* между величинами x и y, а не в *предсказании* значений величины y по значениям величины x.

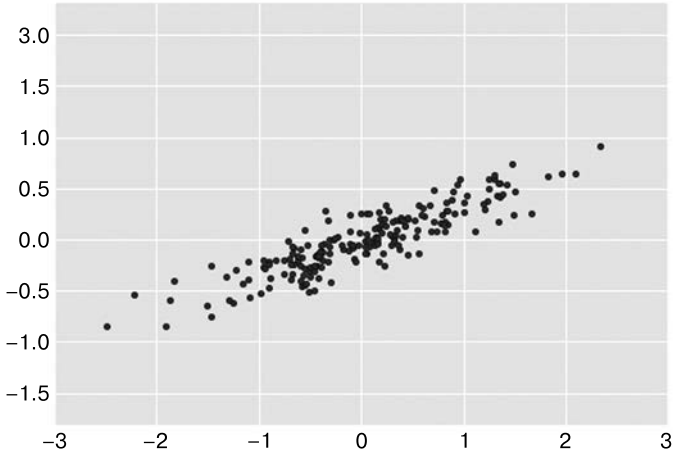

Рис. 5.80. Данные для демонстрации алгоритма PCA

В методе главных компонент выполняется количественная оценка этой зависимости путем нахождения списка *главных осей координат* (principal axes) данных и их использования для описания набора данных. Выполнить это с помощью оценивателя `PCA` из библиотеки Scikit-Learn можно следующим образом:

```
In[3]: from sklearn.decomposition import PCA
       pca = PCA(n_components=2)
       pca.fit(X)

Out[3]: PCA(copy=True, n_components=2, whiten=False)
```

При обучении алгоритм определяет некоторые относящиеся к данным величины, самые важные из них — компоненты и объяснимая дисперсия (explained variance):

```
In[4]: print(pca.components_)
```

```
[[ 0.94446029  0.32862557]
 [ 0.32862557 -0.94446029]]

In[5]: print(pca.explained_variance_)

[ 0.75871884  0.01838551]
```

Чтобы понять смысл этих чисел, визуализируем их в виде векторов над входными данными, используя компоненты для задания направления векторов, а объяснимую дисперсию — в качестве квадратов их длин (рис. 5.81):

```
In[6]: def draw_vector(v0, v1, ax=None):
           ax = ax or plt.gca()
           arrowprops=dict(arrowstyle='->',
                           linewidth=2,
                           shrinkA=0, shrinkB=0)
           ax.annotate('', v1, v0, arrowprops=arrowprops)

       # Рисуем данные
       plt.scatter(X[:, 0], X[:, 1], alpha=0.2)
       for length, vector in zip(pca.explained_variance_, pca.components_):
           v = vector * 3 * np.sqrt(length)
           draw_vector(pca.mean_, pca.mean_ + v)
       plt.axis('equal');
```

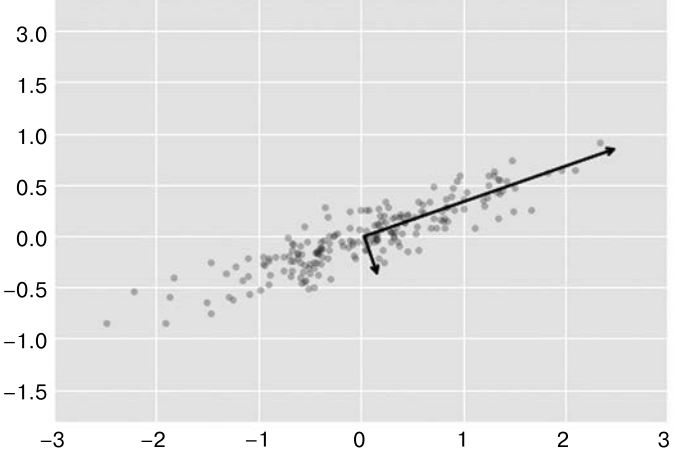

Рис. 5.81. Визуализация главных осей данных

Эти векторы отражают *главные оси координат* данных, а показанная на рис. 5.81 длина соответствует «важности» роли данной оси при описании распределения данных, точнее говоря, это мера дисперсии данных при проекции на эту ось. Проекции точек данных на главные оси и есть главные компоненты данных.

Нарисовав эти главные компоненты рядом с исходными данными, получаем графики, показанные на рис. 5.82.

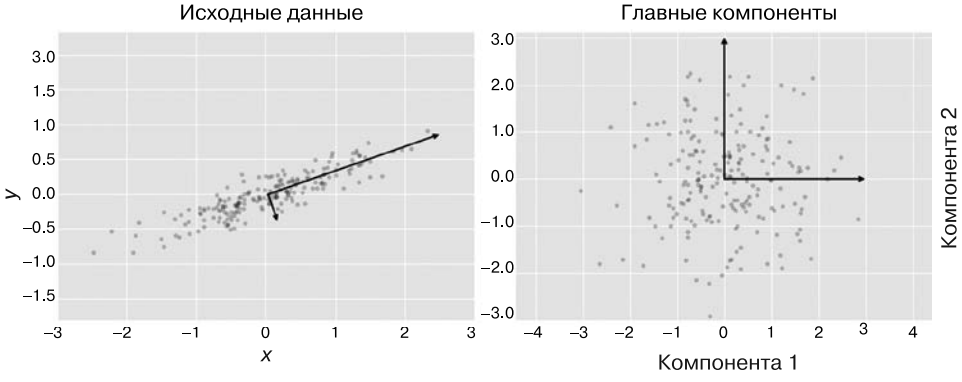

Рис. 5.82. Преобразованные главные оси данных

Это преобразование от осей координат данных к главным осям представляет собой *аффинное преобразование* (affine transformation). По существу, это значит, что оно состоит из сдвига (translation), вращения (rotation) и пропорционального масштабирования (uniform scaling).

Хотя этот алгоритм поиска главных компонент может показаться всего лишь математической диковиной, оказывается, что у него есть весьма перспективные приложения в сфере машинного обучения и исследования данных.

PCA как метод понижения размерности

Использование метода PCA для понижения размерности включает обнуление одной или нескольких из наименьших главных компонент, в результате чего данные проецируются на пространство меньшей размерности с сохранением максимальной дисперсии данных.

Вот пример использования PCA в качестве понижающего размерность преобразования:

```
In[7]: pca = PCA(n_components=1)
       pca.fit(X)
       X_pca = pca.transform(X)
       print("original shape:   ", X.shape)
       print("transformed shape:", X_pca.shape)

original shape:    (200, 2)
transformed shape: (200, 1)
```

Преобразованные данные стали одномерными. Для лучшего понимания эффекта этого понижения размерности можно выполнить обратное преобразование этих данных и нарисовать их рядом с исходными (рис. 5.83):

```
In[8]: X_new = pca.inverse_transform(X_pca)
       plt.scatter(X[:, 0], X[:, 1], alpha=0.2)
       plt.scatter(X_new[:, 0], X_new[:, 1], alpha=0.8)
       plt.axis('equal');
```

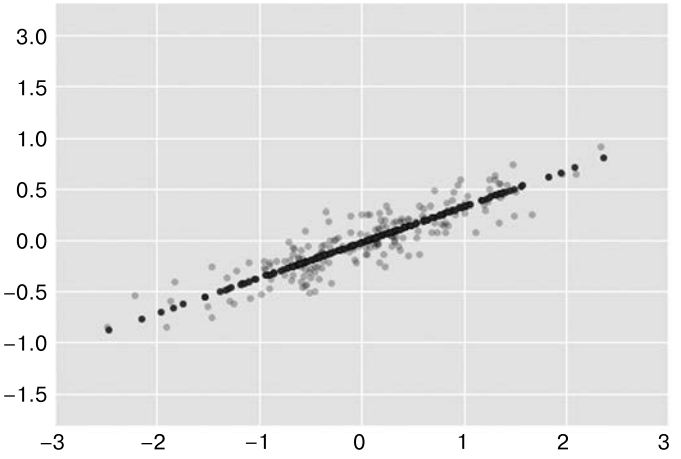

Рис. 5.83. Визуализация PCA как метода понижения размерности

Более светлые точки — исходные данные, а более темные — спроецированная версия. Из рисунка становится понятно, что означает понижение размерности с помощью PCA: информация по наименее важной главной оси/осям координат уничтожается и остается только компонента (-ы) данных с максимальной дисперсией. Отсекаемая часть дисперсии (пропорциональная разбросу точек рядом с линией, показанному на рис. 5.83) является приближенной мерой того, сколько «информации» отбрасывается при этом понижении размерности.

Набор данных пониженной размерности в каком-то смысле «достаточно хорошо» подходит для кодирования важнейших зависимостей между точками: несмотря на понижение размерности данных на 50 %, общая зависимость между точками данных по большей части была сохранена.

Использование метода PCA для визуализации: рукописные цифры

Полезность метода понижения размерности, возможно, не вполне ясна в случае двух измерений, но становится более очевидной при работе с многомерными данными. Рассмотрим приложение метода PCA к данным по рукописным цифрам, с которыми мы уже работали в разделе «Заглянем глубже: деревья принятия решений и случайные леса» этой главы.

Начнем с загрузки данных:

```
In[9]: from sklearn.datasets import load_digits
       digits = load_digits()
       digits.data.shape

Out[9]:
(1797, 64)
```

Напоминаем, что данные состоят из изображений 8 × 8 пикселов, то есть 64-мерны. Чтобы понять зависимости между этими точками, воспользуемся методом PCA для проекции их в пространство более подходящей размерности, допустим, 2:

```
In[10]: pca = PCA(2)  # Проекция из 64-мерного в двумерное пространство
        projected = pca.fit_transform(digits.data)
        print(digits.data.shape)
        print(projected.shape)

(1797, 64)
(1797, 2)
```

Теперь можно построить график двух главных компонент каждой точки, чтобы получить больше информации о наших данных (рис. 5.84):

```
In[11]: plt.scatter(projected[:, 0], projected[:, 1],
                    c=digits.target, edgecolor='none', alpha=0.5,
                    cmap=plt.cm.get_cmap('spectral', 10))
        plt.xlabel('component 1') # Компонента 1
        plt.ylabel('component 2') # Компонента 2
        plt.colorbar();
```

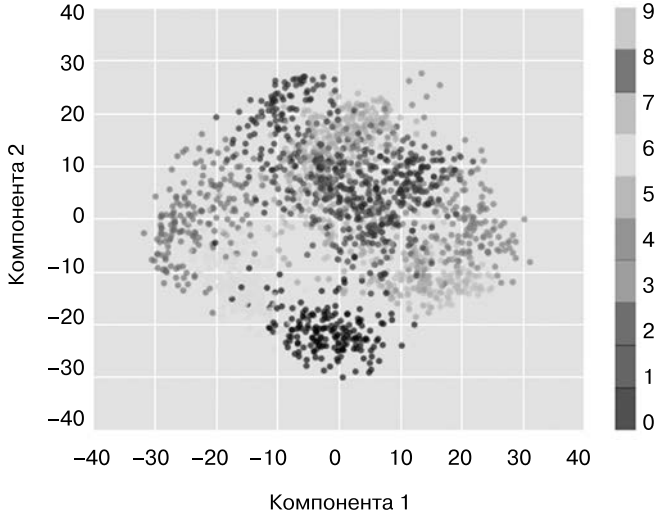

Рис. 5.84. Применение метода PCA к данным по рукописным цифрам

Вспомним, что означают эти компоненты: полный набор данных представляет собой 64-мерное облако, а эти точки — проекции каждой из точек данных вдоль направлений максимальной дисперсии. По существу, мы нашли оптимальные растяжение и вращение в 64-мерном пространстве, позволяющие увидеть, как цифры выглядят в двух измерениях, причем сделать это с помощью метода без учителя, то есть безотносительно меток.

В чем смысл компонент?

Заглянем еще глубже и спросим себя, что *означает* понижение размерности. Ответ на этот вопрос удобнее всего выразить в терминах сочетаний базисных векторов. Например, каждое изображение из обучающей последовательности описывается набором 64 значений пикселов, которые мы назовем вектором x:

$$x = [x_1,\ x_2,\ x_3 \ldots\ x_{64}].$$

Мы можем рассматривать это в терминах пиксельного базиса, то есть для формирования изображения необходимо умножить каждый элемент вышеприведенного вектора на значение описываемого им пиксела, после чего сложить результаты:

$$\text{image}(x) = x_1 \times (\text{пиксел 1}) + x_2 \times (\text{пиксел 2}) + x_3 \times (\text{пиксел 3}) + \ldots + x_{64} \times (\text{пиксел 64}).$$

Один из возможных способов понижения размерности этих данных — обнуление большей части базисных векторов. Например, если мы будем использовать только первые восемь пикселов, то получим восьмимерную проекцию данных (рис. 5.85), но она будет плохо отражать изображение в целом: мы отбрасываем почти 90 % пикселов!

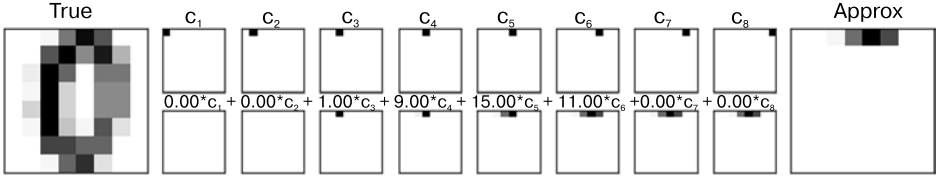

Рис. 5.85. Наивный метод понижения размерности путем отбрасывания пикселов

Верхний ряд на рисунке демонстрирует отдельные пикселы, а нижний — общий вклад этих пикселов в структуру изображения. С помощью только восьми из компонент пиксельного базиса можно сконструировать лишь небольшую часть 64-пиксельного изображения. Продолжив эту последовательность действий и использовав все 64 пиксела, мы бы получили исходное изображение.

Однако попиксельное представление не единственный вариант базиса. Возможно и использование других базисных функций, когда каждый пиксел вносит в каждую из них некий заранее определенный вклад:

$$\text{image}(x) = \text{среднее\_значение} + x_1 \times (\text{базис 1}) + x_2 \times (\text{базис 2}) + x_3 \times (\text{базис 3}) + \ldots$$

Метод PCA можно рассматривать как процесс выбора оптимальных базисных функций, таких, чтобы комбинации лишь нескольких из них было достаточно для удовлетворительного воссоздания основной части элементов набора данных. Главные компоненты, служащие низкоразмерным представлением наших данных, будут в этом случае просто коэффициентами, умножаемыми на каждый из элементов ряда. На рис. 5.86 показано аналогичное рис. 5.85 восстановление цифры с помощью среднего значения и первых восьми базисных функций PCA.

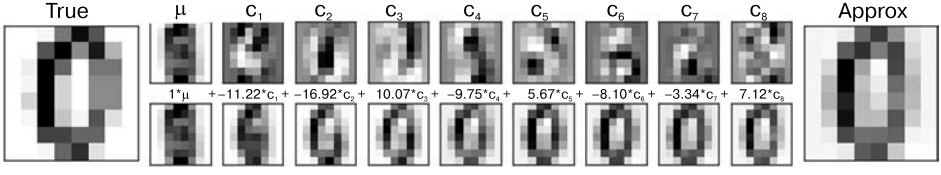

Рис. 5.86. Более продвинутый метод понижения размерности путем отбрасывания наименее важных главных компонент (ср. с рис. 5.85)

В отличие от пиксельного базиса базис PCA позволяет восстановить наиболее заметные признаки входного изображения с помощью среднего значения и восьми компонент! Вклад каждого пиксела в каждый компонент в нашем двумерном примере зависит от направленности вектора. Именно в этом смысле PCA обеспечивает низкоразмерное представление данных: он находит более эффективный набор базисных функций, чем естественный пиксельный базис исходных данных.

Выбор количества компонент

Важнейшая составная часть использования метода PCA на практике — оценка количества компонент, необходимого для описания данных. Определить это количество можно с помощью представления интегральной *доли объяснимой дисперсии* (explained variance ratio) в виде функции от количества компонент (рис. 5.87):

```
In[12]: pca = PCA().fit(digits.data)
        plt.plot(np.cumsum(pca.explained_variance_ratio_))
        plt.xlabel('number of components')
        plt.ylabel('cumulative explained variance');
```

Эта кривая представляет количественную оценку содержания общей, 64-мерной, дисперсии в первых *N*-компонентах. Например, из рисунка видно, что в случае

цифр первые десять компонент содержат примерно 75 % дисперсии, а для описания близкой к 100 % доли дисперсии необходимо около 50 компонент.

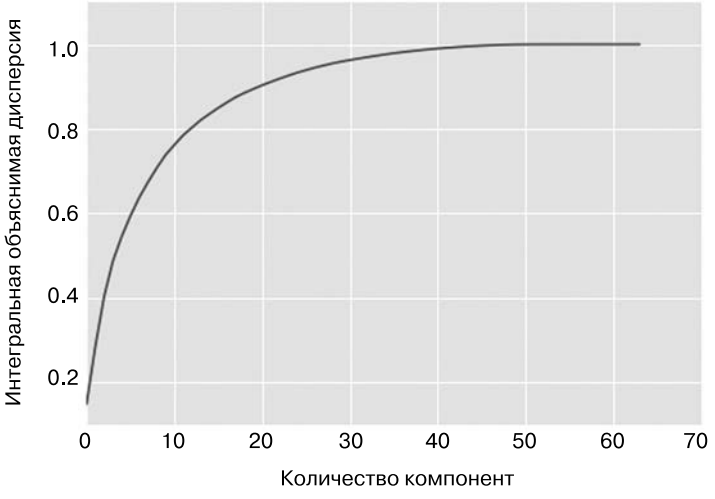

Рис. 5.87. Интегральная объяснимая дисперсия — мера сохранения методом PCA информационного наполнения данных

В данном случае мы видим, что при нашей двумерной проекции теряется масса информации (по оценке на основе объяснимой дисперсии) и что для сохранения 90 % дисперсии необходимо около 20 компонент. Подобный график для высокоразмерного набора данных помогает оценить присутствующий в множественных наблюдениях уровень избыточности.

Использование метода PCA для фильтрации шума

Метод PCA можно применять для фильтрации зашумленных данных. Основная идея состоит в следующем: шум должен довольно мало влиять на компоненты с дисперсией, значительно превышающей его уровень. Восстановление данных с помощью лишь самого крупного подмножества главных компонент должно приводить к относительному сохранению сигнала и отбрасыванию шума.

Посмотрим, как это будет выглядеть в случае данных по цифрам. Сначала нарисуем часть незашумленных входных данных (рис. 5.88):

```
In[13]:
       def plot_digits(data):
            fig, axes = plt.subplots(4, 10, figsize=(10, 4),
                                     subplot_kw={'xticks':[], 'yticks':[]},
                                     gridspec_kw=dict(hspace=0.1, wspace=0.1))
            for i, ax in enumerate(axes.flat):
```

```
        ax.imshow(data[i].reshape(8, 8),
                  cmap='binary', interpolation='nearest',
                  clim=(0, 16))
plot_digits(digits.data)
```

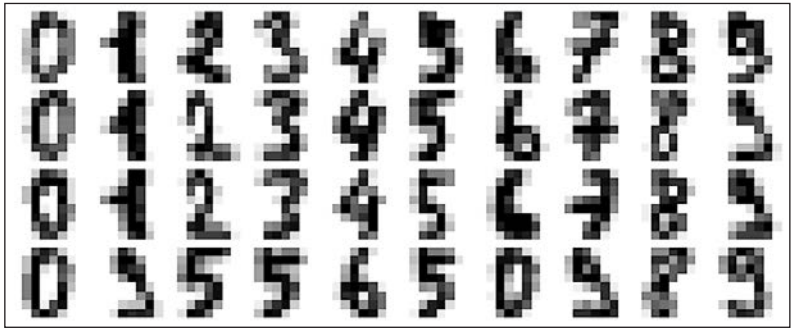

Рис. 5.88. Цифры без шума

Добавим случайный шум для создания зашумленного набора данных и нарисуем уже его (рис. 5.89):

```
In[14]: np.random.seed(42)
        noisy = np.random.normal(digits.data, 4)
        plot_digits(noisy)
```

Рис. 5.89. Цифры с добавленным Гауссовым случайным шумом

Визуально очевидно, что изображения зашумлены и содержат фиктивные пикселы. Обучим алгоритм PCA на этих зашумленных данных, указав, что проекция должна сохранять 50 % дисперсии:

```
In[15]: pca = PCA(0.50).fit(noisy)
        pca.n_components_

Out[15]: 12
```

В данном случае 50 % дисперсии соответствует 12 главным компонентам. Вычислим эти компоненты, после чего воспользуемся обратным преобразованием для восстановления отфильтрованных цифр (рис. 5.90):

```
In[16]: components = pca.transform(noisy)
        filtered = pca.inverse_transform(components)
        plot_digits(filtered)
```

Рис. 5.90. Цифры с устраненным с помощью метода PCA шумом

Эти возможности по сохранению сигнала/фильтрации шума делают метод PCA очень удобной процедурой выбора признаков, например, вместо обучения классификатора на чрезвычайно многомерных данных можно обучить его на низкоразмерном представлении, что автоматически приведет к фильтрации случайного шума во входных данных.

Пример: метод Eigenfaces

Ранее мы рассмотрели пример использования проекции PCA в качестве процедуры выбора признаков для распознавания лиц с помощью метода опорных векторов (см. раздел «Заглянем глубже: метод опорных векторов» данной главы). Теперь мы вернемся к этому примеру и рассмотрим его подробнее. Напоминаю, что мы используем набор данных Labeled Faces in the Wild (LFW), доступный через библиотеку Scikit-Learn:

```
In[17]: from sklearn.datasets import fetch_lfw_people
        faces = fetch_lfw_people(min_faces_per_person=60)
        print(faces.target_names)
        print(faces.images.shape)
```

```
['Ariel Sharon' 'Colin Powell' 'Donald Rumsfeld' 'George W Bush'
 'Gerhard Schroeder' 'Hugo Chavez' 'Junichiro Koizumi' 'Tony Blair']
(1348, 62, 47)
```

Выясним, какие главные оси координат охватывают этот набор данных. Поскольку набор данных велик, воспользуемся классом `RandomizedPCA` — содержащийся в нем

рандомизированный метод позволяет аппроксимировать первые N компонент намного быстрее, чем обычный оцениватель PCA, что очень удобно для многомерных данных (в данном случае размерность равна почти 3000). Рассмотрим первые 150 компонент:

```
In[18]: from sklearn.decomposition import RandomizedPCA
        pca = RandomizedPCA(150)
        pca.fit(faces.data)

Out[18]: RandomizedPCA(copy=True, iterated_power=3, n_components=150,
                 random_state=None, whiten=False)
```

В нашем случае будет интересно визуализировать изображения, соответствующие первым нескольким главным компонентам (эти компоненты формально носят название собственных векторов (eigenvectors), так что подобные изображения часто называют «собственными лицами» (eigenfaces)). Как вы видите из рис. 5.91, они такие же жуткие, как и их название:

```
In[19]: fig, axes = plt.subplots(3, 8, figsize=(9, 4),
                                 subplot_kw={'xticks':[], 'yticks':[]},
                                 gridspec_kw=dict(hspace=0.1, wspace=0.1))
        for i, ax in enumerate(axes.flat):
            ax.imshow(pca.components_[i].reshape(62, 47), cmap='bone')
```

Рис. 5.91. Визуализация собственных лиц, полученных из набора данных LFW

Результат весьма интересен и позволяет ощутить разнообразие изображений: например, вид первых нескольких собственных лиц (считая с левого верхнего угла связан с углом падения света на лицо, а в дальнейшем главные векторы выделяют некоторые черты, например глаза, носы и губы. Посмотрим на интегральную дисперсию этих компонент, чтобы выяснить, какая доля информации сохраняется (рис. 5.92):

```
In[20]: plt.plot(np.cumsum(pca.explained_variance_ratio_))
        plt.xlabel('number of components')
        plt.ylabel('cumulative explained variance');
```

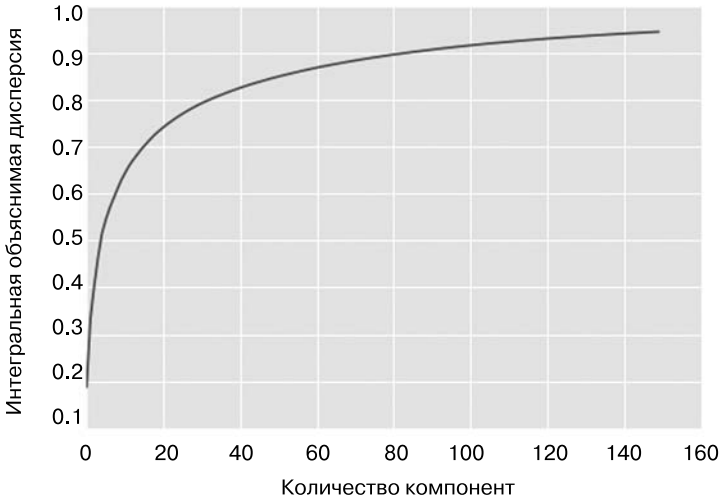

Рис. 5.92. Интегральная объяснимая дисперсия для набора данных LFW

Мы видим, что эти 150 компонент отвечают за более чем 90 % дисперсии. Это дает нам уверенность в том, что при использовании 150 компонент мы сможем восстановить бо́льшую часть существенных характеристик данных. Ради уточнения сравним входные изображения с восстановленными из этих 150 компонент (рис. 5.93):

```
In[21]: # Вычисляем компоненты и проекции лиц
        pca = RandomizedPCA(150).fit(faces.data)
        components = pca.transform(faces.data)
        projected = pca.inverse_transform(components)

In[22]: # Рисуем результаты
        fig, ax = plt.subplots(2, 10, figsize=(10, 2.5),
                               subplot_kw={'xticks':[], 'yticks':[]},
                               gridspec_kw=dict(hspace=0.1, wspace=0.1))
        for i in range(10):
            ax[0, i].imshow(faces.data[i].reshape(62, 47), cmap='binary_r')
            ax[1, i].imshow(projected[i].reshape(62, 47), cmap='binary_r')
        ax[0, 0].set_ylabel('full-dim\ninput')
        # Полноразмерные входные данные
        ax[1, 0].set_ylabel('150-dim\nreconstruction');
        # 150-мерная реконструкция
```

Рис. 5.93. 150-мерная реконструкция данных из LFW с помощью метода PCA

В верхнем ряду на этом рисунке показаны входные изображения, а в нижнем — восстановленные на основе лишь 150 из почти 3000 изначальных признаков. Из этой визуализации становится понятно, почему применявшийся в разделе «Заглянем глубже: метод опорных векторов» данной главы выбор признаков с помощью PCA работал настолько успешно: хотя он понизил размерность данных почти в 20 раз, спроецированные изображения содержат достаточно информации для визуального распознавания изображенных на фотографиях персон. А значит, наш алгоритм классификации достаточно обучить на 150-мерных, а не 3000-мерных данных, что в зависимости от конкретного алгоритма может оказаться намного более эффективным.

Резюме метода главных компонент

В этом разделе мы обсудили использование метода главных компонент для понижения размерности, визуализации многомерных данных, фильтрации шума и выбора признаков в многомерных данных. Метод PCA благодаря своей универсальности и легкой интерпретируемости результатов оказался эффективным в множестве контекстов и дисциплин. Я стараюсь при работе с любым многомерным набором данных начинать с использования метода PCA для визуализации зависимостей между точками (аналогично тому, как мы сделали с рукописными цифрами), выяснения дисперсии данных (аналогично тому, как мы поступили с собственными лицами) и выяснения внутренней размерности данных (путем построения графика доли объяснимой дисперсии). PCA не подходит для всех многомерных наборов данных, но с его помощью можно просто и эффективно почерпнуть о наборе многомерных данных полезную информацию.

Основной недостаток метода PCA состоит в том, что на него оказывают сильное влияние аномальные значения в данных. Поэтому было разработано немало ошибкоустойчивых вариантов PCA, многие из них стремятся итеративно отбрасывать те точки данных, которые описываются исходными компонентами недостаточно хорошо. Библиотека Scikit-Learn содержит несколько интересных вариантов метода PCA, включая классы `RandomizedPCA` и `SparsePCA`, находящиеся в модуле `sklearn.decomposition`. `RandomizedPCA`, который мы уже встречали ранее, использует недетерминированный метод для быстрой аппроксимации нескольких первых из главных компонент данных с очень высокой размерностью, а `SparsePCA` вводит понятие регуляризации (см. раздел «Заглянем глубже: линейная регрессия» данной главы), служащее для обеспечения разреженности компонент.

В следующих разделах мы рассмотрим другие методы машинного обучения без учителя, основывающиеся на некоторых идеях метода PCA.

Заглянем глубже: обучение на базе многообразий

Мы уже ознакомились с возможностями метода главных компонент для решения задачи понижения размерности — снижения количества признаков набора данных с сохранением существенных зависимостей между точками. Хотя метод PCA гибок,

быстр и обеспечивает легкость интерпретации результатов, в случае *нелинейных* зависимостей в данных он работает не столь хорошо.

Чтобы справиться с этой проблемой, можно обратиться к классу методов, известных под названием *обучения на базе многообразий* (manifold learning). Это класс оценивателей без учителя, нацеленных на описание наборов данных как низкоразмерных многообразий, вложенных в пространство большей размерности. Чтобы понять, что такое многообразие, представьте себе лист бумаги: это двумерный объект в нашем привычном трехмерном мире, который можно изогнуть или свернуть в трех измерениях. В терминах обучения на базе многообразий можно считать этот лист двумерным многообразием, вложенным в трехмерное пространство.

Вращение, смена ориентации или растяжение листа бумаги в трехмерном пространстве не меняет его плоской геометрии: подобные операции эквивалентны линейному вложению. Если согнуть, скрутить или скомкать бумагу, она все равно остается двумерным многообразием, но вложение ее в трехмерное пространство уже не будет линейным. Алгоритмы обучения на базе многообразий нацелены на изучение базовой двумерной природы листа бумаги, даже если он был смят ради размещения в трехмерном пространстве.

В этом разделе мы продемонстрируем настройки методов обучения на базе многообразий, причем наиболее подробно рассмотрим *многомерное масштабирование* (multidimensional scaling, MDS), *локально линейное вложение* (locally linear embedding, LLE) и *изометрическое отображение* (isometric mapping, Isomap). Начнем с обычных импортов:

```
In[1]: %matplotlib inline
       import matplotlib.pyplot as plt
       import seaborn as sns; sns.set()
       import numpy as np
```

Обучение на базе многообразий: HELLO

Для облегчения понимания вышеупомянутых понятий начнем с генерации двумерных данных, подходящих для описания многообразия. Вот функция, создающая данные в форме слова HELLO:

```
In[2]:
def make_hello(N=1000, rseed=42):
    # Создаем рисунок с текстом "HELLO"; сохраняем его в формате PNG
    fig, ax = plt.subplots(figsize=(4, 1))
    fig.subplots_adjust(left=0, right=1, bottom=0, top=1)
    ax.axis('off')
    ax.text(0.5, 0.4, 'HELLO', va='center', ha='center', weight='bold',
    size=85)
    fig.savefig('hello.png')
    plt.close(fig)
```

```
    #  Открываем этот PNG-файл и берем из него случайные точки
    from matplotlib.image import imread
    data = imread('hello.png')[::-1, :, 0].T
    rng = np.random.RandomState(rseed)
    X = rng.rand(4 * N, 2)
    i, j = (X * data.shape).astype(int).T
    mask = (data[i, j] < 1)
    X = X[mask]
    X[:, 0] *= (data.shape[0] / data.shape[1])
    X = X[:N]
    return X[np.argsort(X[:, 0])]
```

Вызываем эту функцию и визуализируем полученные данные (рис. 5.94):

```
In[3]: X = make_hello(1000)
       colorize = dict(c=X[:, 0], cmap=plt.cm.get_cmap('rainbow', 5))
       plt.scatter(X[:, 0], X[:, 1], **colorize)
       plt.axis('equal');
```

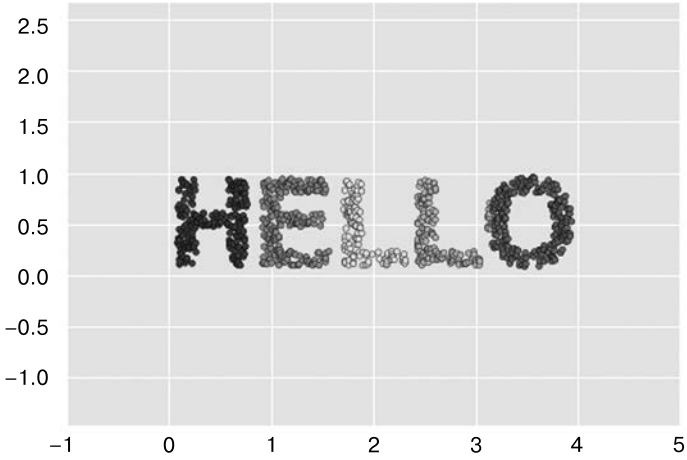

Рис. 5.94. Данные для обучения на базе многообразий

Выходные данные двумерны и состоят из точек, формирующих слово HELLO. Эта внешняя форма данных поможет нам отслеживать визуально работу алгоритмов.

Многомерное масштабирование (MDS)

При взгляде на подобные данные становится ясно, что конкретные значения x и y — не самая существенная характеристика этого набора данных: мы можем пропорционально увеличить/сжать или повернуть данные, а надпись HELLO все равно останется четко различимой. Например, при использовании матрицы вращения для вращения данных значения x и y изменятся, но данные, по существу, останутся теми же (рис. 5.95):

```
In[4]:
       def rotate(X, angle):
           theta = np.deg2rad(angle)
           R = [[np.cos(theta), np.sin(theta)],
                [-np.sin(theta), np.cos(theta)]]
           return np.dot(X, R)

       X2 = rotate(X, 20) + 5
       plt.scatter(X2[:, 0], X2[:, 1], **colorize)
       plt.axis('equal');
```

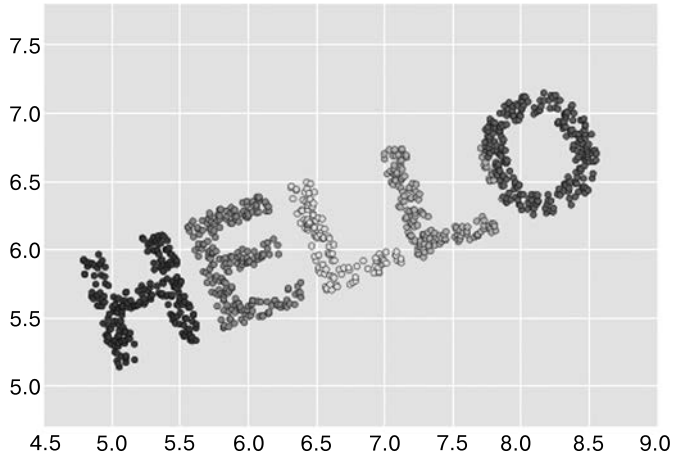

Рис. 5.95. Набор данных после вращения

Это говорит о том, что значения *x* и *y* не обязательно важны для внутренних зависимостей данных. Существенно в таком случае *расстояние* между каждой из точек и всеми остальными точками набора данных. Для представления его часто используют так называемую матрицу расстояний: при *N* точках создается такой массив размера $N \times N$, что его элемент *(i, j)* содержит расстояние между точками *i* и *j*. Воспользуемся эффективной функцией `pairwise_distances` из библиотеки Scikit-Learn, чтобы выполнить этот расчет для наших исходных данных:

```
In[5]: from sklearn.metrics import pairwise_distances
       D = pairwise_distances(X)
       D.shape
```

```
Out[5]: (1000, 1000)
```

Как я и обещал, при наших $N = 1000$ точках мы получили матрицу размером 1000×1000, которую можно визуализировать так, как показано на рис. 5.96:

```
In[6]: plt.imshow(D, zorder=2, cmap='Blues', interpolation='nearest')
       plt.colorbar();
```

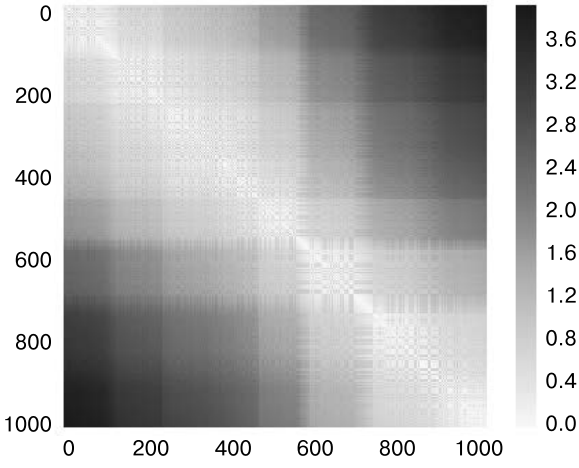

Рис. 5.96. Визуализация попарных расстояний между точками

Сформировав аналогичным образом матрицу расстояний между подвергшимися вращению и сдвигу точками, увидим, что она не поменялась:

```
In[7]: D2 = pairwise_distances(X2)
       np.allclose(D, D2)

Out[7]: True
```

Благодаря подобной матрице расстояний мы получаем инвариантное к вращениям и сдвигам представление данных, но визуализация матрицы интуитивно не слишком ясна. В представлении на рис. 5.96 потеряны всякие следы интересной структуры данных: виденного нами ранее слова HELLO.

Хотя вычисление матрицы расстояний на основе координат *(x, y)* не представляет труда, обратное преобразование расстояний в координаты x и y — непростая задача. Именно для этого и служит алгоритм многомерного масштабирования: по заданной матрице расстояний между точками он восстанавливает D-мерное координатное представление данных. Посмотрим, как это будет выглядеть для нашей матрицы расстояний. Воспользуемся, чтобы указать, что мы передаем матрицу расстояний, опцией `precomputed` параметра `dissimilarity` (рис. 5.97):

```
In[8]: from sklearn.manifold import MDS
       model = MDS(n_components=2, dissimilarity='precomputed',
       random_state=1)
       out = model.fit_transform(D)
       plt.scatter(out[:, 0], out[:, 1], **colorize)
       plt.axis('equal');
```

Алгоритм MDS восстанавливает одно из возможных двумерных координатных представлений данных на основе *одной лишь* матрицы расстояний размера $N \times N$, описывающей зависимости между точками данных.

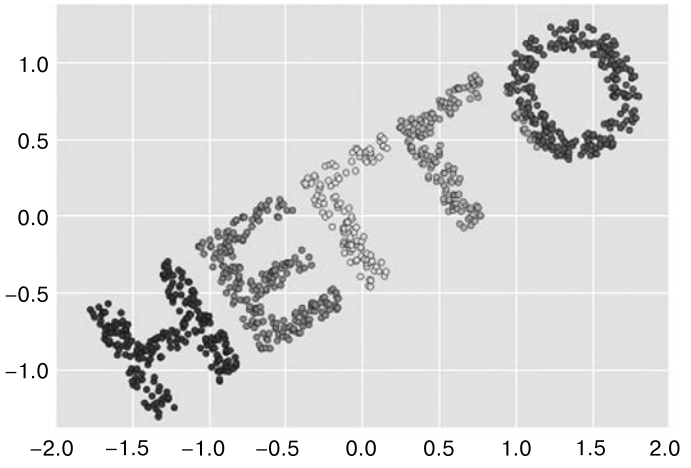

Рис. 5.97. MDS-вложение, вычисленное на основе попарных расстояний между точками

MDS как обучение на базе многообразий

Полезность этого метода становится еще очевиднее, если учесть, что матрицы расстояний можно вычислить для данных *любой* размерности. Так, например, вместо вращения данных в двумерном пространстве можно спроецировать их в трехмерное пространство с помощью следующей функции (по существу, это трехмерное обобщение матрицы вращения, с которой мы имели дело ранее):

```
In[9]:
       def random_projection(X, dimension=3, rseed=42):
           assert dimension >= X.shape[1]
           rng = np.random.RandomState(rseed)
           C = rng.randn(dimension, dimension)
           e, V = np.linalg.eigh(np.dot(C, C.T))
           return np.dot(X, V[:X.shape[1]])

       X3 = random_projection(X, 3)
       X3.shape

Out[9]: (1000, 3)
```

Визуализируем эти точки, чтобы понять, с чем имеем дело (рис. 5.98):

```
In[10]: from mpl_toolkits import mplot3d
        ax = plt.axes(projection='3d')
        ax.scatter3D(X3[:, 0], X3[:, 1], X3[:, 2],
                     **colorize)
        ax.view_init(azim=70, elev=50)
```

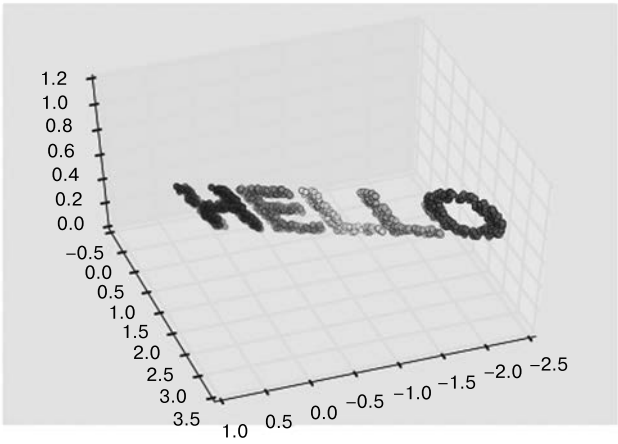

Рис. 5.98. Данные, линейно вложенные в трехмерное пространство

Можно теперь передать эти данные оценивателю `MDS` для вычисления матрицы расстояний и последующего определения оптимального двумерного вложения для нее. В результате мы получаем восстановленное представление исходных данных (рис. 5.99):

```
In[11]: model = MDS(n_components=2, random_state=1)
        out3 = model.fit_transform(X3)
        plt.scatter(out3[:, 0], out3[:, 1], **colorize)
        plt.axis('equal');
```

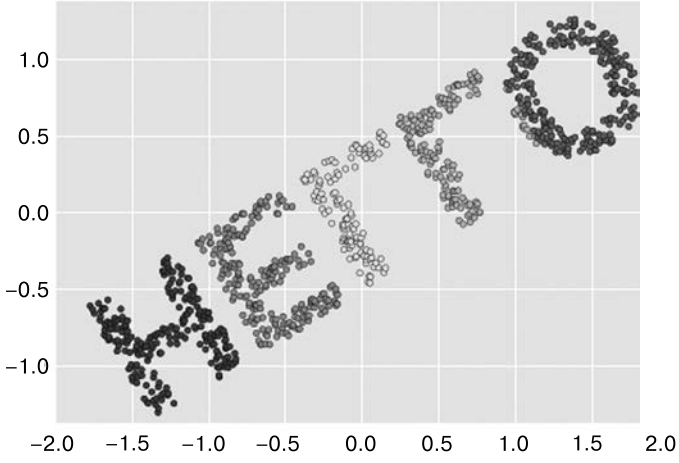

Рис. 5.99. MDS-вложение трехмерных данных позволяет восстановить исходные данные с точностью до вращения и отражения

В этом и состоит задача оценивателя обучения на базе многообразий: при заданных многомерных вложенных данных он находит низкоразмерное их представление, сохраняющее определенные зависимости внутри данных. В случае метода MDS сохраняется расстояние между всеми парами точек.

Нелинейные вложения: там, где MDS не работает

До сих пор мы говорили о линейных вложениях, состоящих из вращений, сдвигов и масштабирований данных в пространствах более высокой размерности. Однако в случае нелинейного вложения, то есть при выходе за пределы этого простого набора операций, метод MDS терпит неудачу. Рассмотрим следующее вложение, при котором входные данные деформируются в форму трехмерной буквы S:

```
In[12]: def make_hello_s_curve(X):
            t = (X[:, 0] - 2) * 0.75 * np.pi
            x = np.sin(t)
            y = X[:, 1]
            z = np.sign(t) * (np.cos(t) - 1)
            return np.vstack((x, y, z)).T


        XS = make_hello_s_curve(X)
```

Речь опять идет о трехмерных данных, но мы видим, что вложение — намного более сложное (рис. 5.100):

```
In[13]: from mpl_toolkits import mplot3d
        ax = plt.axes(projection='3d')
        ax.scatter3D(XS[:, 0], XS[:, 1], XS[:, 2],
                     **colorize);
```

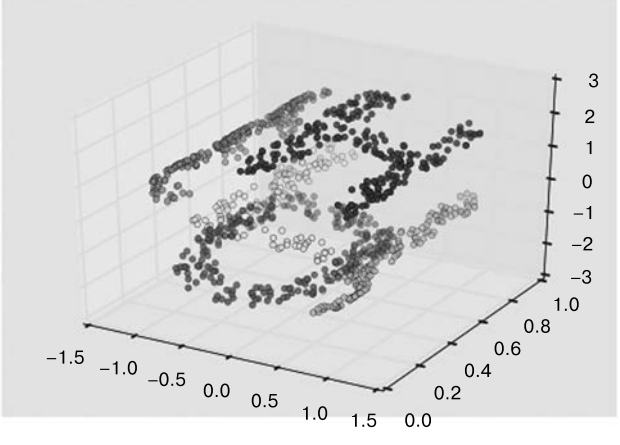

Рис. 5.100. Нелинейно вложенные в трехмерное пространство данные

Базовые зависимости между точками данных сохранены, но на этот раз данные были преобразованы нелинейным образом: они были свернуты в форму буквы S.

Если попытаться использовать для этих данных простой алгоритм MDS, он не сумеет «развернуть» это нелинейное вложение и мы потеряем из виду существенные зависимости во вложенном многообразии (рис. 5.101):

```
In[14]: from sklearn.manifold import MDS
        model = MDS(n_components=2, random_state=2)
        outS = model.fit_transform(XS)
        plt.scatter(outS[:, 0], outS[:, 1], **colorize)
        plt.axis('equal');
```

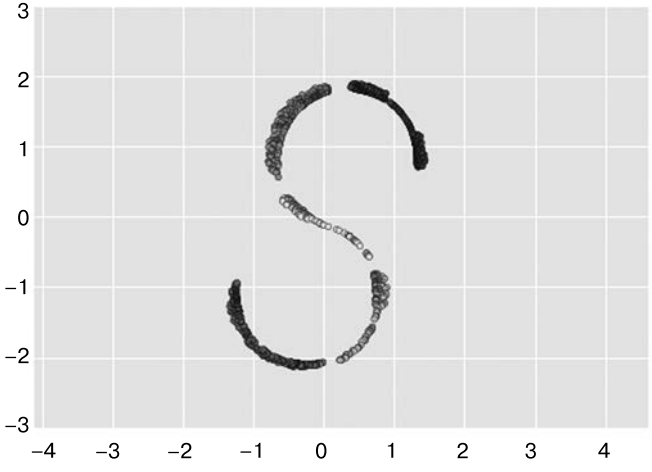

Рис. 5.101. Использование алгоритма MDS для нелинейных данных: попытка восстановления исходной структуры оказывается неудачной

Даже самое лучшее двумерное *линейное* вложение не сможет развернуть обратно нашу S-образную кривую, а отбросит вместо этого исходную ось координат *Y*.

Нелинейные многообразия: локально линейное вложение

Корень проблемы в том, что MDS пытается сохранять расстояния между удаленными точками при формировании вложения. Но что, если изменить алгоритм так, чтобы он сохранял расстояния только между близлежащими точками? Полученное в результате вложение будет лучше решать нашу задачу.

Наглядно можно представить этот метод так, как показано на рис. 5.102.

Тонкие линии отражают расстояния, которые необходимо сохранить при вложении. Слева представлена модель, используемая в методе MDS: сохраняются расстоо-

яния между всеми парами точек в наборе данных. Справа — модель, используемая алгоритмом обучения на базе многообразий, который называется локально линейным вложением (LLE). Вместо сохранения всех расстояний сохраняются только расстояния между *соседними точками*: в данном случае ближайшими 100 соседями каждой точки.

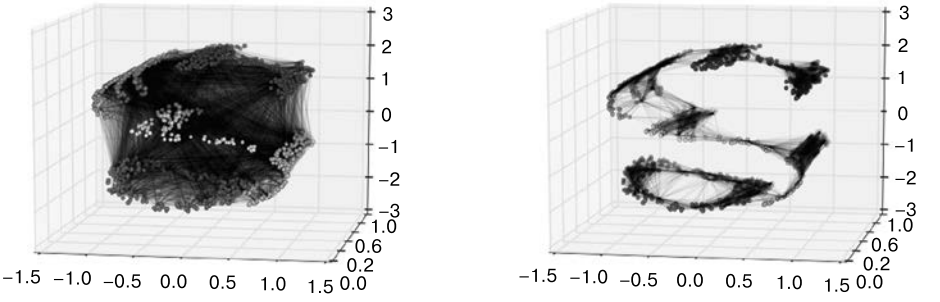

Рис. 5.102. Представление связей между точками в MDS и LLE

При изучении рисунка слева становится понятно, почему метод MDS не работает: способа «уплощения» этих данных с сохранением в достаточной степени длины каждого отрезка между двумя точками просто не существует. На рисунке справа, с другой стороны, ситуация выглядит несколько более оптимистично. Вполне можно представить себе разворачивание данных так, чтобы хотя бы приблизительно сохранить длины отрезков. Именно это и делает метод LLE путем нахождения глобального экстремума отражающей эту логику функции стоимости.

Существует несколько вариантов метода LLE, мы здесь будем использовать для восстановления вложенного двумерного многообразия *модифицированный алгоритм LLE*. В целом модифицированный алгоритм LLE работает лучше других его вариантов при восстановлении хорошо структурированных многообразий с очень небольшой дисторсией (рис. 5.103):

```
In[15]:
from sklearn.manifold import LocallyLinearEmbedding
model = LocallyLinearEmbedding(n_neighbors=100, n_components=2,
method='modified',
                               eigen_solver='dense')
out = model.fit_transform(XS)

fig, ax = plt.subplots()
ax.scatter(out[:, 0], out[:, 1], **colorize)
ax.set_ylim(0.15, -0.15);
```

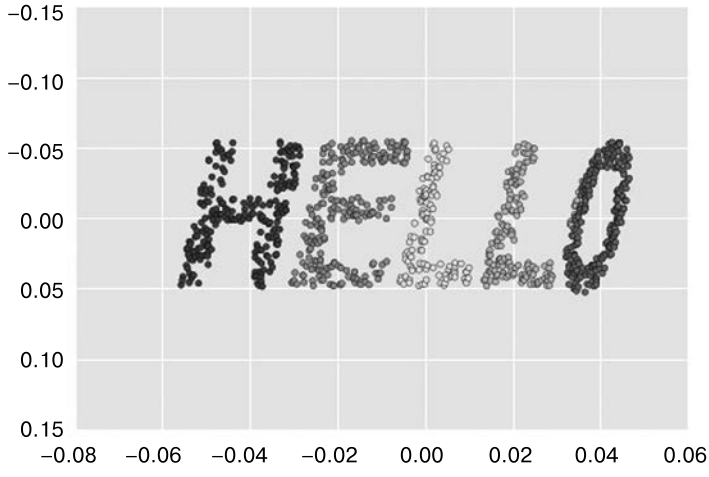

Рис. 5.103. Локально линейное вложение может восстанавливать изначальные данные из нелинейно вложенных входных данных

Результат остается несколько искаженным по сравнению с исходным многообразием, но существенные зависимости внутри данных метод уловил!

Некоторые соображения относительно методов обучения на базе многообразий

Хотя это и была захватывающая история, на практике методы обучения на базе многообразий оказываются настолько привередливыми, что они редко используются для чего-то большего, чем простая качественная визуализация многомерных данных.

Вот несколько конкретных вопросов, в которых обучение на базе многообразий выглядит плохо по сравнению с методом PCA.

- При обучении на базе многообразий не существует удачного фреймворка для обработки отсутствующих данных. В отличие от него в методе PCA существуют простые итеративные подходы для работы с отсутствующими данными.
- При обучении на базе многообразий наличие шума в данных может «закоротить» многообразие и коренным образом изменить вложение. В отличие от него метод PCA естественным образом отделяет шум от наиболее важных компонент.
- Результат вложения многообразия обычно сильно зависит от количества выбранных соседей, и не существует надежного, формулируемого количественно способа выбора оптимального числа соседей. В отличие от него метод PCA не требует подобного выбора.
- При обучении на базе многообразий непросто определить оптимальное число измерений на выходе алгоритма. В отличие от него метод PCA позволяет определить выходную размерность, основываясь на объяснимой дисперсии.

- При обучении на базе многообразий смысл вложенных измерений не всегда понятен. В методе PCA смысл главных компонент совершенно ясен.
- При обучении на базе многообразий вычислительная сложность методов составляет $O[N^2]$ или даже $O[N^3]$. Некоторые рандомизированные варианты метода PCA работают гораздо быстрее (однако в пакете `megaman` (https://github.com/mmp2/megaman) реализованы методы обучения на базе многообразий, масштабирующиеся гораздо лучше).

С учетом всего этого единственное безусловное преимущество методов обучения на базе многообразий перед PCA состоит в их способности сохранять нелинейные зависимости в данных. Именно поэтому я стараюсь сначала изучать данные с помощью PCA, а затем использовать методы обучения на базе многообразий.

В библиотеке Scikit-Learn реализовано несколько распространенных вариантов обучения на базе многообразий и локально линейного вложения: в документации Scikit-Learn имеется их обсуждение и сравнение (http://scikit-learn.org/stable/modules/manifold.html). Исходя из моего собственного опыта, могу дать вам следующие рекомендации.

- В модельных задачах, подобных S-образной кривой, локально линейное вложение (LLE) и его варианты (особенно *модифицированный метод LLE*) демонстрируют отличные результаты. Они реализованы в классе `sklearn.manifold.LocallyLinearEmbedding`.
- В случае многомерных данных, полученных из реальных источников, метод LLE часто работает плохо, и изометрическое отображение (Isomap), похоже, выдает более осмысленные вложения. Оно реализовано в классе `sklearn.manifold.Isomap`.
- Для сильно кластеризованных данных отличные результаты демонстрирует метод стохастического вложения соседей на основе распределения Стьюдента (t-distributed stochastic neighbor embedding), хотя и работает иногда очень медленно по сравнению с другими методами. Он реализован в классе `sklearn.manifold.TSNE`.

Если вы хотите посмотреть, как они работают, запустите каждый из них на данных из этого раздела.

Пример: использование Isomap для распознавания лиц

Обучение на базе многообразий часто применяется при исследовании зависимостей между многомерными точками данных. Один из распространенных случаев многомерных данных — изображения. Например, набор изображений, состоящих каждое из 1000 пикселов, можно рассматривать как набор точек в 1000-мерном пространстве — яркость каждого пиксела в каждом изображении соответствует координате в соответствующем измерении.

Применим алгоритм Isomap к данным, содержащим какие-либо лица. Воспользуемся набором данных Labeled Faces in the Wild (LFW), с которым уже сталкивались в разделах «Заглянем глубже: метод опорных векторов» и «Заглянем глубже: метод

главных компонент» этой главы. Следующая команда позволяет скачать данные и кэшировать их в вашем домашнем каталоге для дальнейшего использования:

```
In[16]: from sklearn.datasets import fetch_lfw_people
        faces = fetch_lfw_people(min_faces_per_person=30)
        faces.data.shape

Out[16]: (2370, 2914)
```

Итак, у нас имеется 2370 изображений, каждое размером 2914 пикселов. Другими словами, изображения можно считать точками данных в 2914-мерном пространстве!

Быстро визуализируем несколько изображений, чтобы посмотреть, с чем мы имеем дело (рис. 5.104):

```
In[17]: fig, ax = plt.subplots(4, 8, subplot_kw=dict(xticks=[], yticks=[]))
        for i, axi in enumerate(ax.flat):
            axi.imshow(faces.images[i], cmap='gray')
```

Рис. 5.104. Примеры исходных лиц

Не помешает нарисовать низкоразмерное вложение 2914-мерных данных, чтобы ознакомиться с основными зависимостями между изображениями. Удобно также начать с вычисления PCA и изучения полученной доли объяснимой дисперсии. Это даст нам представление о том, сколько линейных признаков необходимо для описания этих данных (рис. 5.105):

```
In[18]: from sklearn.decomposition import RandomizedPCA
        model = RandomizedPCA(100).fit(faces.data)
        plt.plot(np.cumsum(model.explained_variance_ratio_))
        plt.xlabel('n components')         # Количество компонент
        plt.ylabel('cumulative variance'); # Интегральная дисперсия
```

Как видим, для сохранения 90 % дисперсии необходимо почти 100 компонент. Это значит, что данные, по своей сути, имеют чрезвычайно высокую размерность и их невозможно описать линейно с помощью всего нескольких компонент.

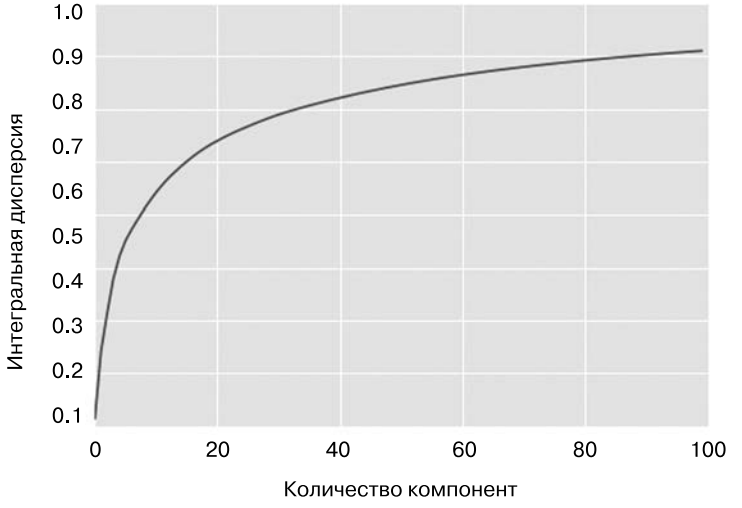

Рис. 5.105. Интегральная дисперсия, полученная из проекции методом PCA

В подобном случае могут оказаться полезны нелинейные вложения на базе многообразий, такие как LLE и Isomap. Рассчитать вложение Isomap для этих лиц можно аналогичным вышеприведенному образом:

```
In[19]: from sklearn.manifold import Isomap
        model = Isomap(n_components=2)
        proj = model.fit_transform(faces.data)
        proj.shape

Out[19]: (2370, 2)
```

Результат представляет собой двумерную проекцию всех исходных изображений. Чтобы лучше представить, что говорит нам эта проекция, опишем функцию, выводящую миниатюры изображений в местах проекций:

```
In[20]: from matplotlib import offsetbox

        def plot_components(data, model, images=None, ax=None,
                            thumb_frac=0.05, cmap='gray'):
            ax = ax or plt.gca()

            proj = model.fit_transform(data)
            ax.plot(proj[:, 0], proj[:, 1], '.k')

            if images is not None:
```

```
        min_dist_2 = (thumb_frac * max(proj.max(0) -
            proj.min(0))) ** 2
        shown_images = np.array([2 * proj.max(0)])
        for i in range(data.shape[0]):
            dist = np.sum((proj[i] - shown_images) ** 2, 1)
            if np.min(dist) < min_dist_2:
                # Не отображаем слишком близко расположенные точки
                Continue
            shown_images = np.vstack([shown_images, proj[i]])
            imagebox = offsetbox.AnnotationBbox(
                offsetbox.OffsetImage(images[i], cmap=cmap),
                                      proj[i])
            ax.add_artist(imagebox)
```

Вызываем эту функцию и видим результат (рис. 5.106):

```
In[21]: fig, ax = plt.subplots(figsize=(10, 10))
        plot_components(faces.data,
                        model=Isomap(n_components=2),
                        images=faces.images[:, ::2, ::2])
```

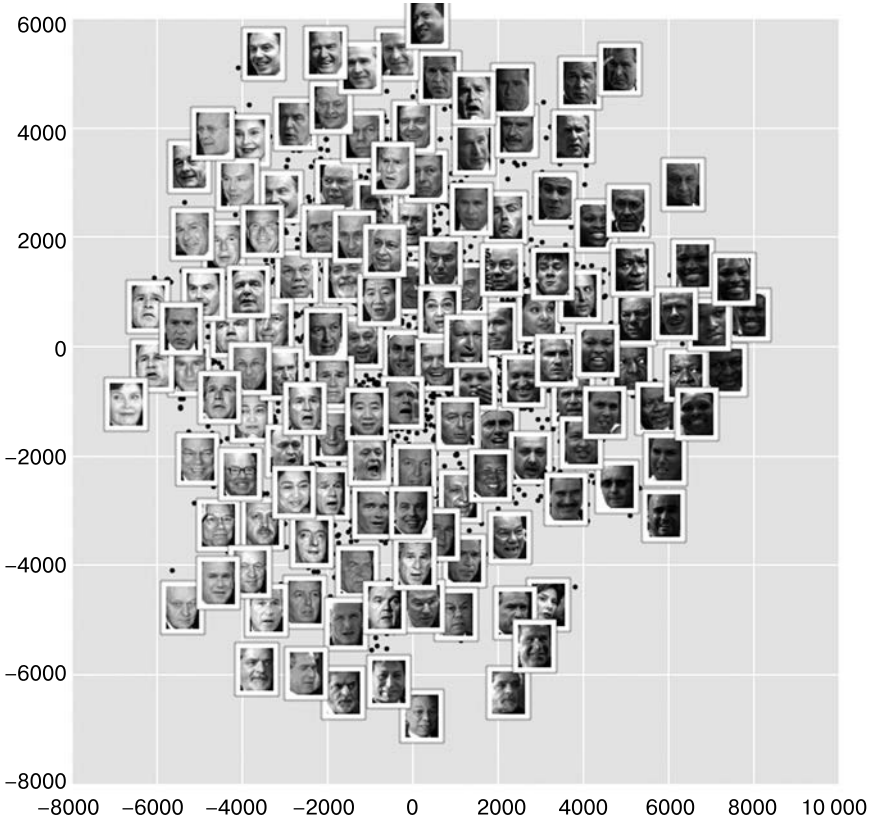

Рис. 5.106. Вложение с помощью Isomap данных о лицах

Результат интересен: первые два измерения Isomap, вероятно, описывают общие признаки изображения: низкую или высокую яркость изображения слева направо и общее расположение лица снизу вверх. Это дает нам общее представление о некоторых базовых признаках данных.

Далее можно перейти к классификации этих данных, возможно, с помощью признаков на базе многообразий в качестве входных данных для алгоритма классификации, аналогично тому, как мы поступили в разделе «Заглянем глубже: метод опорных векторов» этой главы.

Пример: визуализация структуры цифр

В качестве еще одного примера использования обучения на базе многообразий для визуализации рассмотрим набор MNIST рукописных цифр. Эти данные аналогичны цифрам, с которыми мы сталкивались в разделе «Заглянем глубже: деревья принятия решений и случайные леса» этой главы, но с намного большей детализацией изображений. Скачать их можно с сайта http://mldata.org/ с помощью утилиты библиотеки Scikit-Learn:

```
In[22]: from sklearn.datasets import fetch_mldata
        mnist = fetch_mldata('MNIST original')
        mnist.data.shape

Out[22]: (70000, 784)
```

Этот набор состоит из 70 000 изображений, каждое размером 784 пиксела, то есть 28×28 пикселов. Как и ранее, рассмотрим несколько первых изображений (рис. 5.107):

```
In[23]: fig, ax = plt.subplots(6, 8, subplot_kw=dict(xticks=[], yticks=[]))
        for i, axi in enumerate(ax.flat):
            axi.imshow(mnist.data[1250 * i].reshape(28, 28), cmap='gray_r')
```

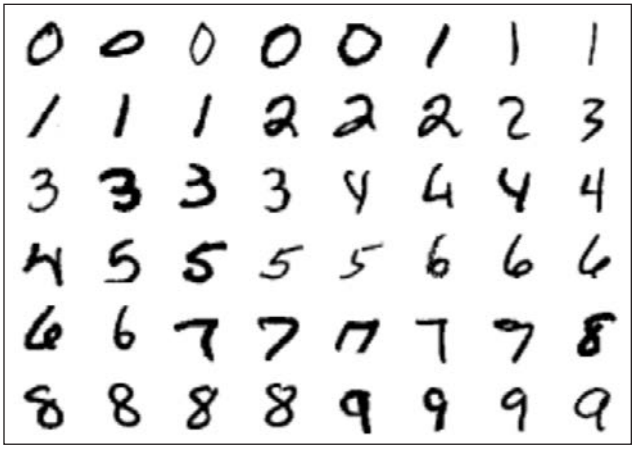

Рис. 5.107. Примеры цифр из набора MNIST

Этот рисунок дает нам представление о разнообразии рукописных цифр в наборе данных.

Вычислим с помощью обучения на базе многообразий проекцию для этих данных, показанную на рис. 5.108. Для ускорения будем использовать только 1/30 часть данных, то есть примерно 2000 точек данных (из-за относительно плохой масштабируемости методов обучения на базе многообразий я пришел к выводу, что несколько тысяч выборок — хорошее количество для начала, чтобы относительно быстро исследовать набор, прежде чем перейти к полномасштабным вычислениям):

```
In[24]:
# используем только 1/30 часть данных:
# вычисления для полного набора данных занимают длительное время!
data = mnist.data[::30]
target = mnist.target[::30]

model = Isomap(n_components=2)
proj = model.fit_transform(data)
plt.scatter(proj[:, 0], proj[:, 1], c=target, cmap=plt.cm.get_cmap('jet', 10))
plt.colorbar(ticks=range(10))
plt.clim(-0.5, 9.5);
```

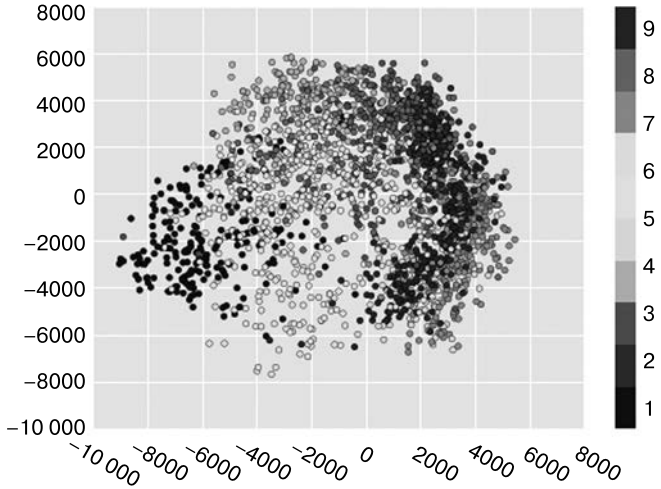

Рис. 5.108. Isomap-вложение для набора данных цифр MNIST

Полученная диаграмма рассеяния демонстрирует некоторые зависимости между точками данных, но точки на ней расположены слишком тесно. Можно получить больше информации, изучая за раз данные лишь об одной цифре (рис. 5.109):

```
In[25]: from sklearn.manifold import Isomap

        # Выбираем для проекции 1/4 цифр "1"
        data = mnist.data[mnist.target == 1][::4]
```

```
fig, ax = plt.subplots(figsize=(10, 10))
model = Isomap(n_neighbors=5, n_components=2, eigen_solver='dense')
plot_components(data, model, images=data.reshape((-1, 28, 28)),
                ax=ax, thumb_frac=0.05, cmap='gray_r')
```

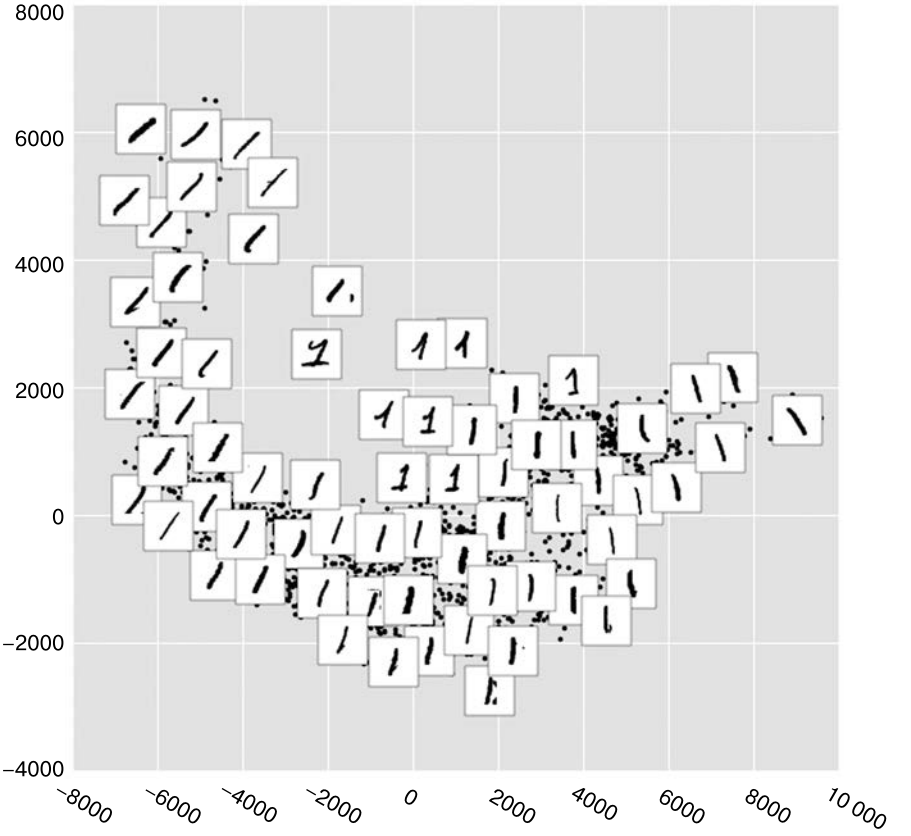

Рис. 5.109. Isomap-вложение только для единиц из набора данных о цифрах

Результат дает нам представление о разнообразии форм, которые может принимать цифра 1 в этом наборе данных. Данные располагаются вдоль широкой кривой в пространстве проекции, отражающей ориентацию цифры. При перемещении вверх по графику мы видим единицы со «шляпками» и/или «подошвами», хотя они в этом наборе данных редки. Проекция дает нам возможность обнаружить аномальные значения с проблемами в данных (например, части соседних цифр, попавших в извлеченные изображения).

Хотя само по себе для задачи классификации цифр это и не особо полезно, но может помочь нам получить представление о данных и подсказать, что делать дальше, например какой предварительной обработке необходимо подвергнуть данные до создания конвейера классификации.

Заглянем глубже: кластеризация методом k-средних

В нескольких предыдущих разделах мы рассматривали только одну разновидность машинного обучения без учителя — понижение размерности. В этом разделе мы перейдем к другому классу моделей машинного обучения без учителя — алгоритмам кластеризации. Алгоритмы кластеризации нацелены на то, чтобы найти, исходя из свойств данных, оптимальное разбиение или дискретную маркировку групп точек.

В библиотеке Scikit-Learn и других местах имеется множество алгоритмов кластеризации, но, вероятно, наиболее простой для понимания — алгоритм *кластеризации методом k-средних* (k-means clustering), реализованный в классе `sklearn.cluster.KMeans`. Начнем с обычных импортов:

```
In[1]: %matplotlib inline
       import matplotlib.pyplot as plt
       import seaborn as sns; sns.set()  # для стилизации графиков
       import numpy as np
```

Знакомство с методом k-средних

Алгоритм *k*-средних выполняет поиск заранее заданного количества кластеров в немаркированном многомерном наборе данных. Достигается это с помощью простого представления о том, что такое оптимальная кластеризация.

- «Центр кластера» — арифметическое среднее всех точек, относящихся к этому кластеру.
- Каждая точка ближе к центру своего кластера, чем к центрам других кластеров.

Эти два допущения составляют основу модели метода *k*-средних. Далее мы рассмотрим детальнее, *каким именно образом* алгоритм находит это решение, а пока возьмем простой набор данных и посмотрим на результаты работы метода *k*-средних для него.

Во-первых, сгенерируем двумерный набор данных, содержащий четыре отдельных «пятна». Чтобы подчеркнуть отсутствие учителя в этом алгоритме, мы не будем включать метки в визуализацию (рис. 5.110):

```
In[2]: from sklearn.datasets.samples_generator import make_blobs
       X, y_true = make_blobs(n_samples=300, centers=4,
                              cluster_std=0.60, random_state=0)
       plt.scatter(X[:, 0], X[:, 1], s=50);
```

Визуально выделить здесь четыре кластера не представляет труда. Алгоритм *k*-средних делает это автоматически, используя в библиотеке Scikit-Learn API статистических оценок:

```
In[3]: from sklearn.cluster import KMeans
       kmeans = KMeans(n_clusters=4)
       kmeans.fit(X)

       y_kmeans = kmeans.predict(X)
```

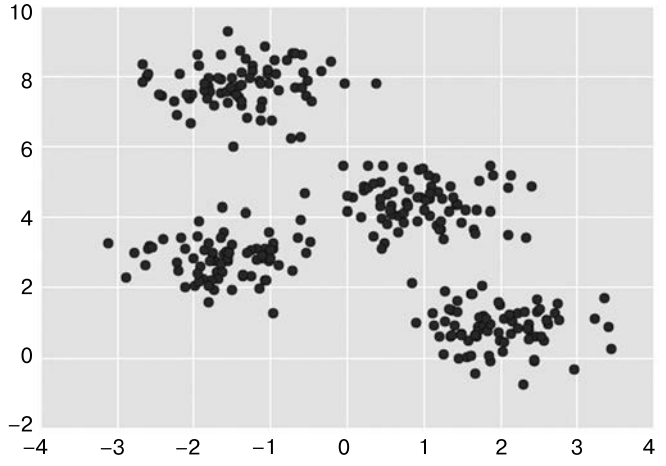

Рис. 5.110. Данные для демонстрации кластеризации

Визуализируем результаты, выведя на график окрашенные в соответствии с метками данные. Нарисуем найденные оценивателем метода *k*-средних центры кластеров (рис. 5.111):

```
In[4]: plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, s=50, cmap='viridis')

       centers = kmeans.cluster_centers_
       plt.scatter(centers[:, 0], centers[:, 1], c='black', s=200, alpha=0.5);
```

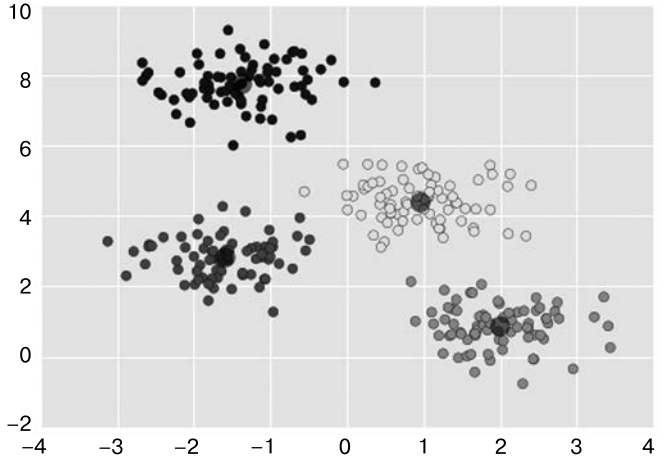

Рис. 5.111. Центры кластеров метода k-средних с окрашенными в разные цвета кластерами

Хорошая новость состоит в том, что алгоритм *k*-средних (по крайней мере в этом простом случае) задает соответствие точек кластерам очень схоже с тем, как мы бы могли сделать это визуально. Но у вас может возникнуть вопрос: как этому

алгоритму удалось найти кластеры так быстро? В конце концов, количество возможных сочетаний точек и кластеров зависит экспоненциально от числа точек данных, так что поиск методом полного перебора оказался бы чрезвычайно ресурсозатратным. Поиск методом полного перебора здесь не требуется. Вместо него в типичном варианте метода *k*-средних применяется интуитивно понятный подход, известный под названием *«максимизация математического ожидания»* (expectation-maximization, EM).

Алгоритм k-средних: максимизация математического ожидания

Максимизация математического ожидания (EM) — мощный алгоритм, встречающийся во множестве контекстов науки о данных. Метод *k*-средних — особенно простое и понятное приложение этого алгоритма, и мы рассмотрим его здесь вкратце. Подход максимизации математического ожидания состоит из следующей процедуры.

1. Выдвигаем гипотезу о центрах кластеров.
2. Повторяем до достижения сходимости:
   - *E-шаг:* приписываем точки к ближайшим центрам кластеров;
   - *M-шаг:* задаем новые центры кластеров в соответствии со средними значениями.

   E-шаг, или шаг ожидания (expectation), назван так потому, что включает актуализацию математического ожидания того, к каким кластерам относятся точки. M-шаг, или шаг максимизации (maximization), назван так потому, что включает максимизацию некоторой целевой функции, описывающей расположения центров кластеров. В таком случае максимизация достигается путем простого усреднения данных в кластере.

   Этому алгоритму посвящена обширная литература, но если коротко, то можно подвести его итоги следующим образом: при обычных обстоятельствах каждая итерация шагов E и M всегда будет приводить к улучшению оценки показателей кластера. Визуализировать этот алгоритм можно так, как показано на рис. 5.112.

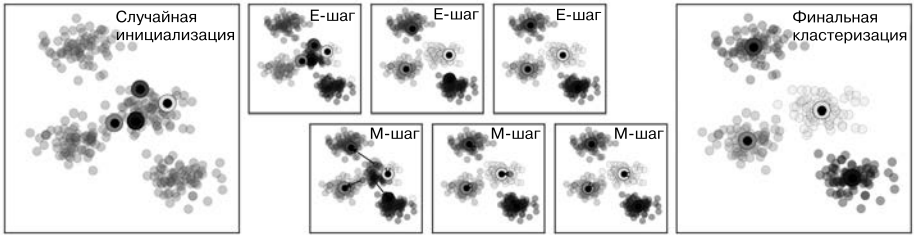

Рис. 5.112. Визуализация EM-алгоритма для метода k-средних

Для показанных на рис. 5.112 конкретных начальных значений кластеры сходятся всего за три итерации. Интерактивную версию рисунка можно увидеть в онлайн-приложении (https://github.com/jakevdp/PythonDataScienceHandbook).

Алгоритм *k*-средних достаточно прост для того, чтобы уместиться в нескольких строках кода. Вот простейшая его реализация (рис. 5.113):

```
In[5]: from sklearn.metrics import pairwise_distances_argmin

       def find_clusters(X, n_clusters, rseed=2):
           # 1. Выбираем кластеры случайным образом
           rng = np.random.RandomState(rseed)
           i = rng.permutation(X.shape[0])[:n_clusters]
           centers = X[i]

           while True:
               # 2a. Присваиваем метки в соответствии с ближайшим центром
               labels = pairwise_distances_argmin(X, centers)

               # 2b. Находим новые центры, исходя из средних значений точек
               new_centers = np.array([X[labels == i].mean(0)
                                       for i in range(n_clusters)])

               # 2c. Проверяем сходимость
               if np.all(centers == new_centers):
                   break
               centers = new_centers

           return centers, labels

       centers, labels = find_clusters(X, 4)
       plt.scatter(X[:, 0], X[:, 1], c=labels,
                   s=50, cmap='viridis');
```

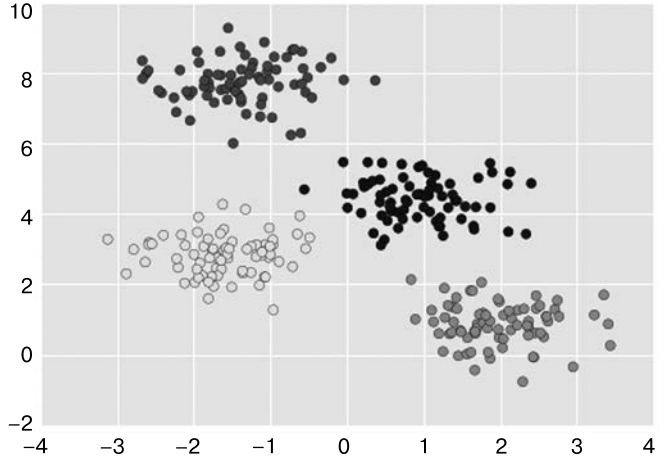

Рис. 5 113. Данные, маркированные с помощью метода k-средних

Самые проверенные реализации выполняют «под капотом» немного больше действий, но основное представление о методе максимизации математического ожидания предыдущая функция дает.

Предупреждения относительно метода максимизации математического ожидания. При использовании алгоритма максимизации математического ожидания следует иметь в виду несколько нюансов.

- *Глобально оптимальный результат может оказаться недостижимым в принципе.* Во-первых, хотя процедура EM гарантированно улучшает результат на каждом шаге, уверенности в том, что она ведет к *глобально* наилучшему решению, нет. Например, если мы воспользуемся в нашей простой процедуре другим начальным значением для генератора случайных чисел, полученные начальные гипотезы приведут к неудачным результатам (рис. 5.114):

```
In[6]: centers, labels = find_clusters(X, 4, rseed=0)
       plt.scatter(X[:, 0], X[:, 1], c=labels,
                   s=50, cmap='viridis');
```

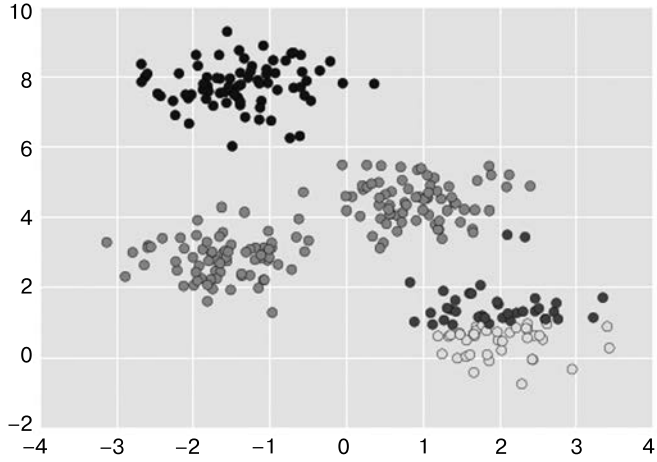

Рис. 5.114. Пример плохой сходимости в методе k-средних

В этом случае EM-метод сошелся к глобально неоптимальной конфигурации, поэтому его часто выполняют для нескольких начальных гипотез, что и делает по умолчанию библиотека Scikit-Learn (это задается с помощью параметра `n_init`, по умолчанию имеющего значение `10`).

- *Количество кластеров следует выбирать заранее.* Еще одна часто встречающаяся проблема с методом *k*-средних заключается в том, что ему необходимо сообщить, какое количество кластеров вы ожидаете: он не умеет вычислять количество кластеров на основе данных. Например, если предложить алгоритму выделить шесть кластеров, он с радостью это сделает и найдет шесть оптимальных кластеров (рис. 5.115):

```
In[7]: labels = KMeans(6, random_state=0).fit_predict(X)
       plt.scatter(X[:, 0], X[:, 1], c=labels,
                   s=50, cmap='viridis');
```

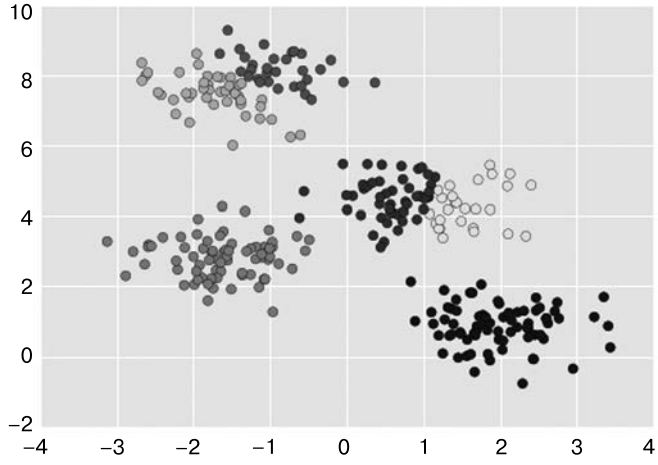

Рис. 5.115. Пример неудачного выбора количества кластеров

Осмысленный ли результат получен — сказать трудно. Один из полезных в этом случае и интуитивно довольно понятных подходов, который мы не станем обсуждать подробно, — *силуэтный анализ* (http://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_silhouette_analysis.html).

В качестве альтернативы можно воспользоваться более сложным алгоритмом кластеризации с лучшим количественным показателем зависимости качества аппроксимации от количества кластеров (например, смесь Гауссовых распределений, см. раздел «Заглянем глубже: смеси Гауссовых распределений» данной главы) или *с возможностью* выбора приемлемого количества кластеров (например, методы DBSCAN, сдвиг среднего или распространения аффинности (affinity propagation), находящиеся в подмодуле `sklearn.cluster`).

- *Применение метода k-средних ограничивается случаем линейных границ кластеров*. Базовое допущение модели *k*-средних (точки должны быть ближе к центру их собственного кластера, чем других) означает, что этот алгоритм зачастую будет неэффективен в случае сложной геометрии кластеров.

  В частности, границы между кластерами в методе *k*-средних всегда будут линейными, а значит, он будет плохо работать в случае более сложных границ.

  Рассмотрим следующие данные и найденные для них при обычном подходе метода *k*-средних метки кластеров (рис. 5.116):

```
In[8]: from sklearn.datasets import make_moons
       X, y = make_moons(200, noise=.05, random_state=0)

In[9]: labels = KMeans(2, random_state=0).fit_predict(X)
       plt.scatter(X[:, 0], X[:, 1], c=labels,
                s=50, cmap='viridis');
```

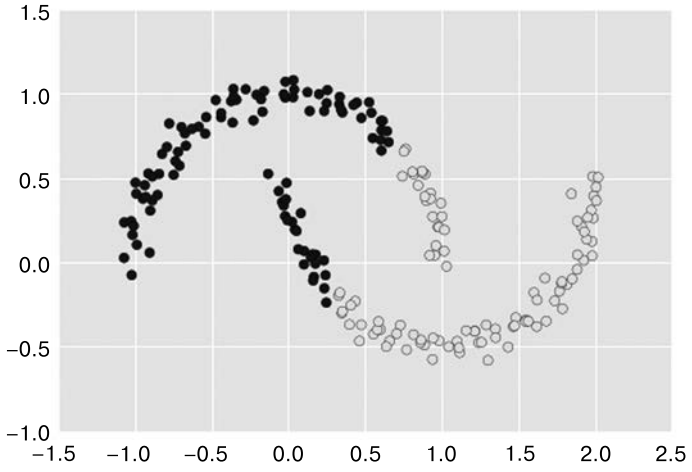

Рис. 5.116. Неудача метода k-средних в случае нелинейных границ

Ситуация напоминает обсуждавшееся в разделе «Заглянем глубже: метод опорных векторов» этой главы, где мы использовали ядерное преобразование для проецирования данных в пространство более высокой размерности, в котором возможно линейное разделение. Можно попробовать воспользоваться той же уловкой, чтобы метод *k*-средних стал распознавать нелинейные границы.

Одна из версий этого ядерного метода *k*-средних реализована в библиотеке Scikit-Learn в оценивателе `SpectralClustering`. Она использует граф ближайших соседей для вычисления представления данных более высокой размерности, после чего задает соответствие меток с помощью алгоритма *k*-средних (рис. 5.117):

```
In[10]: from sklearn.cluster import SpectralClustering
        model = SpectralClustering(n_clusters=2,
                                   affinity='nearest_neighbors',
                                   assign_labels='kmeans')
        labels = model.fit_predict(X)
        plt.scatter(X[:, 0], X[:, 1], c=labels,
                    s=50, cmap='viridis');
```

Как видим, метод *k*-средних с помощью этого ядерного преобразования способен обнаруживать более сложные нелинейные границы между кластерами.

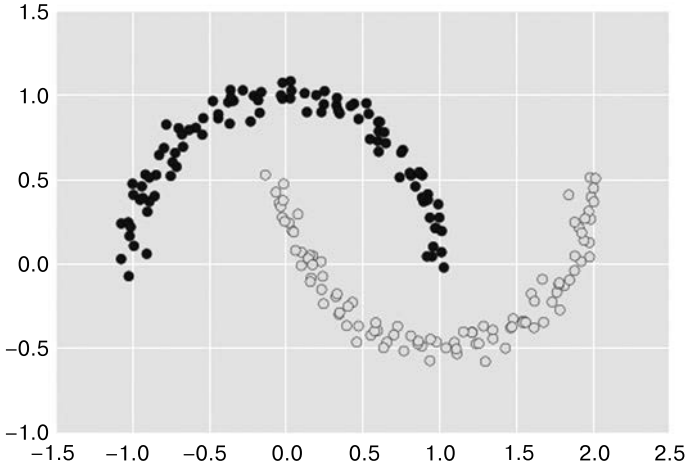

Рис. 5.117. Нелинейные границы, обнаруженные с помощью SpectralClustering

- *Метод k-средних работает довольно медленно в случае большого количества выборок*. Алгоритм может работать довольно медленно при росте числа выборок, ведь при каждой итерации методу *k*-средних необходимо обращаться к каждой точке в наборе данных. Интересно, можно ли смягчить это требование относительно использования всех данных при каждой итерации? Например, можно применить лишь подмножество данных для корректировки центров кластеров на каждом шаге. Эта идея лежит в основе пакетных алгоритмов *k*-средних, один из которых реализован в классе `sklearn.cluster.MiniBatchKMeans`. Их интерфейс не отличается от обычного `KMeans`. В дальнейшем мы рассмотрим пример их использования.

Примеры

При соблюдении некоторой осторожности в плане вышеупомянутых ограничений можно успешно использовать метод *k*-средних во множестве ситуаций. Рассмотрим несколько примеров.

Пример 1: применение метода k-средних для рукописных цифр

Для начала рассмотрим применение метода *k*-средних к тем же простым данным по цифрам, которые мы уже видели в разделах «Заглянем глубже: деревья принятия решений и случайные леса» и «Заглянем глубже: метод главных компонент» данной главы. Мы попробуем воспользоваться методом *k*-средних для распознания схожих цифр *без использования информации об исходных метках*. Это напоминает

первый шаг извлечения смысла нового набора данных, для которого отсутствует какая-либо *априорная* информации о метках.

Начнем с загрузки цифр, а затем перейдем к поиску кластеров с помощью алгоритма `KMeans`. Напомним, что набор данных по цифрам состоит из 1797 выборок с 64 признаками, где каждый из признаков представляет собой яркость одного пиксела в изображении размером 8×8:

```
In[11]: from sklearn.datasets import load_digits
        digits = load_digits()
        digits.data.shape

Out[11]: (1797, 64)
```

Кластеризация выполняется так же, как и ранее:

```
In[12]: kmeans = KMeans(n_clusters=10, random_state=0)
        clusters = kmeans.fit_predict(digits.data)
        kmeans.cluster_centers_.shape

Out[12]: (10, 64)
```

В результате мы получили десять кластеров в 64-мерном пространстве. Обратите внимание, что и центры кластеров представляют собой 64-мерные точки, а значит, их можно интерпретировать как «типичные» цифры в кластере. Посмотрим, что представляют собой эти центры кластеров (рис. 5.118):

```
In[13]: fig, ax = plt.subplots(2, 5, figsize=(8, 3))
        centers = kmeans.cluster_centers_.reshape(10, 8, 8)
        for axi, center in zip(ax.flat, centers):
            axi.set(xticks=[], yticks=[])
            axi.imshow(center, interpolation='nearest', cmap=plt.cm.binary)
```

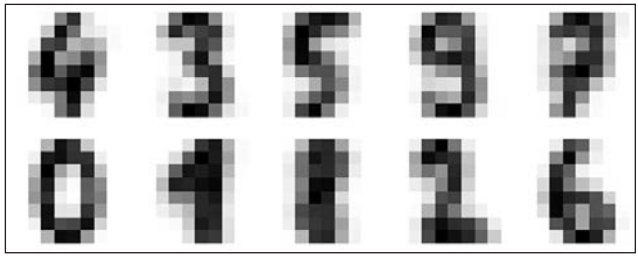

Рис. 5.118. Центры кластеров

Как видим, алгоритм `KMeans` *даже без меток* способен определить кластеры, чьи центры представляют собой легко узнаваемые цифры, возможно, за исключением 1 и 8.

В силу того что алгоритм *k*-средних ничего не знает о сущности кластеров, метки 0–9 могут оказаться перепутаны местами. Исправить это можно, задав соответствие всех полученных меток кластеров имеющимся в них фактическим меткам:

```
In[14]: from scipy.stats import mode

        labels = np.zeros_like(clusters)
        for i in range(10):
            mask = (clusters == i)
            labels[mask] = mode(digits.target[mask])[0]
```

Теперь можно проверить, насколько точно кластеризация без учителя определила подобие цифр в наших данных:

```
In[15]: from sklearn.metrics import accuracy_score
        accuracy_score(digits.target, labels)

Out[15]: 0.79354479688369506
```

С помощью простого алгоритма *k*-средних мы определили правильную группировку для почти 80 % исходных цифр! Посмотрим на матрицу различий (рис. 5.119):

```
In[16]: from sklearn.metrics import confusion_matrix
        mat = confusion_matrix(digits.target, labels)
        sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False,
                    xticklabels=digits.target_names,
                    yticklabels=digits.target_names)
        plt.xlabel('true label')
        plt.ylabel('predicted label');
```

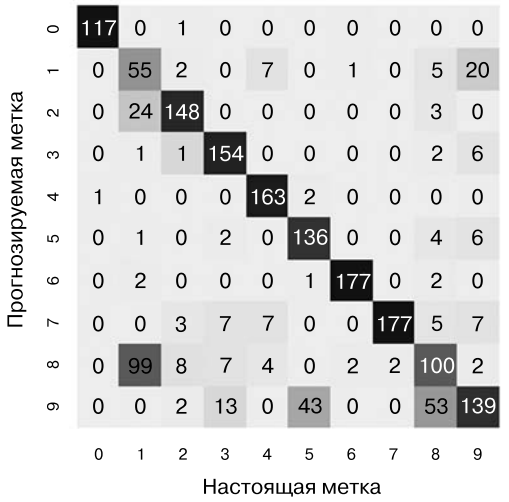

Рис. 5.119. Матрица различий для классификатора методом k-средних

Как и можно было ожидать, судя по визуализированным ранее центрам кластеров, больше всего путаницы между единицами и восьмерками. Но все равно, с помощью метода *k*-средних мы фактически создали классификатор цифр *без каких-либо известных меток*!

Попробуем пойти еще дальше. Воспользуемся для предварительной обработки данных до выполнения *k*-средних алгоритмом стохастического вложения соседей на основе распределения Стьюдента (t-SNE), упоминавшимся в разделе «Заглянем глубже: обучение на базе многообразий» этой главы. t-SNE — нелинейный алгоритм вложения, особенно хорошо умеющий сохранять точки внутри кластеров. Посмотрим, как он работает:

```
In[17]: from sklearn.manifold import TSNE

        # Проекция данных: выполнение этого шага займет несколько секунд
        tsne = TSNE(n_components=2, init='pca', random_state=0)
        digits_proj = tsne.fit_transform(digits.data)

        # Расчет кластеров
        kmeans = KMeans(n_clusters=10, random_state=0)
        clusters = kmeans.fit_predict(digits_proj)

        # Перестановка меток местами
        labels = np.zeros_like(clusters)
        for i in range(10):
            mask = (clusters == i)
            labels[mask] = mode(digits.target[mask])[0]

        # Оценка точности
        accuracy_score(digits.target, labels)

Out[17]: 0.93356149137451305
```

Точность, даже *без использования меток*, составляет почти 94 %. При разумном использовании алгоритмы машинного обучения без учителя демонстрируют отличные результаты: они могут извлекать информацию из набора данных даже тогда, когда сделать это вручную или визуально очень непросто.

Пример 2: использование метода k-средних для сжатия цветов

Одно из интересных приложений кластеризации — сжатие цветов в изображениях. Например, допустим, что у нас есть изображение с миллионами цветов. Почти во всех изображениях большая часть цветов не используется и цвета многих пикселов изображения будут похожи или даже совпадать.

Например, рассмотрим изображение, показанное на рис. 5.120, взятом из модуля `datasets` библиотеки Scikit-Learn (для работы следующего кода у вас должен быть установлен пакет `pillow` языка Python):

```
In[18]: # Обратите внимание: для работы этого кода
        # должен быть установлен пакет pillow
        from sklearn.datasets import load_sample_image
        china = load_sample_image("china.jpg")
        ax = plt.axes(xticks=[], yticks=[])
        ax.imshow(china);
```

Рис. 5.120. Исходное изображение

Само изображение хранится в трехмерном массиве размера (высота, ширина, RGB), содержащем вклад в цвет по красному/синему/зеленому каналам в виде целочисленных значений от 0 до 255:

```
In[19]: china.shape

Out[19]: (427, 640, 3)
```

Этот набор пикселов можно рассматривать как облако точек в трехмерном цветовом пространстве. Изменим форму данных на [n_samples × n_features] и масштабируем шкалу цветов так, чтобы они располагались между 0 и 1:

```
In[20]: data = china / 255.0 # используем шкалу 0...1
        data = data.reshape(427 * 640, 3)
        data.shape

Out[20]: (273280, 3)
```

Визуализируем эти пикселы в данном цветовом пространстве, используя подмножество из 10 000 пикселов для быстроты работы (рис. 5.121):

```
In[21]:
       def plot_pixels(data, title, colors=None, N=10000):
            if colors is None:
                colors = data

            # Выбираем случайное подмножество
            rng = np.random.RandomState(0)
```

```
        i = rng.permutation(data.shape[0])[:N]
        colors = colors[i]
        R, G, B = data[i].T

        fig, ax = plt.subplots(1, 2, figsize=(16, 6))
        ax[0].scatter(R, G, color=colors, marker='.')
        ax[0].set(xlabel='Red', ylabel='Green', xlim=(0, 1), ylim=(0, 1))

        ax[1].scatter(R, B, color=colors, marker='.')
        ax[1].set(xlabel='Red', ylabel='Blue', xlim=(0, 1), ylim=(0, 1))

        fig.suptitle(title, size=20);
```

```
In[22]: plot_pixels(data, title='Input color space: 16 million possible
                    colors') # Исходное цветовое пространство: 16 миллионов
                             # возможных цветов
```

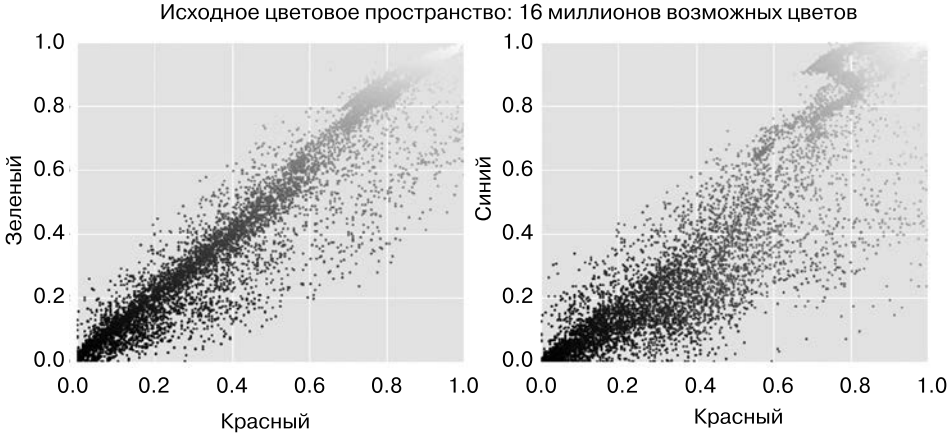

Рис. 5.121. Распределение пикселов в цветовом пространстве RGB

Теперь уменьшим количество цветов с 16 миллионов до 16 путем кластеризации методом *k*-средних на пространстве пикселов. Так как наш набор данных очень велик, воспользуемся мини-пакетным методом *k*-средних, который вычисляет результат гораздо быстрее, чем стандартный метод *k*-средних путем работы с подмножествами набора данных (рис. 5.122):

```
In[23]: from sklearn.cluster import MiniBatchKMeans
        kmeans = MiniBatchKMeans(16)
        kmeans.fit(data)
        new_colors = kmeans.cluster_centers_[kmeans.predict(data)]

        plot_pixels(data, colors=new_colors,
                    title="Reduced color space: 16 colors")
                    # Редуцированное цветовое пространство: 16 цветов
```

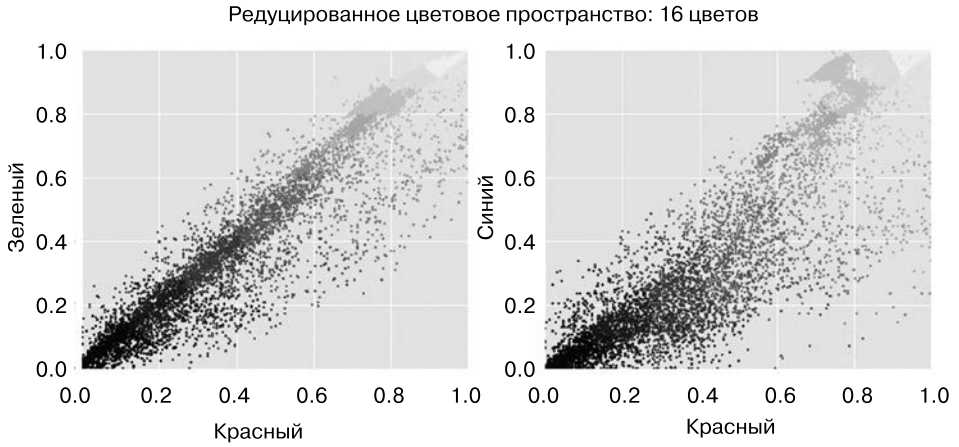

Рис. 5.122. Шестнадцать кластеров в цветовом пространстве RGB

В результате исходные пикселы перекрашиваются в другие цвета: каждый пиксел получает цвет ближайшего центра кластера. Рисуя эти новые цвета в пространстве изображения вместо пространства пикселов, видим эффект от перекрашивания (рис. 5.123).

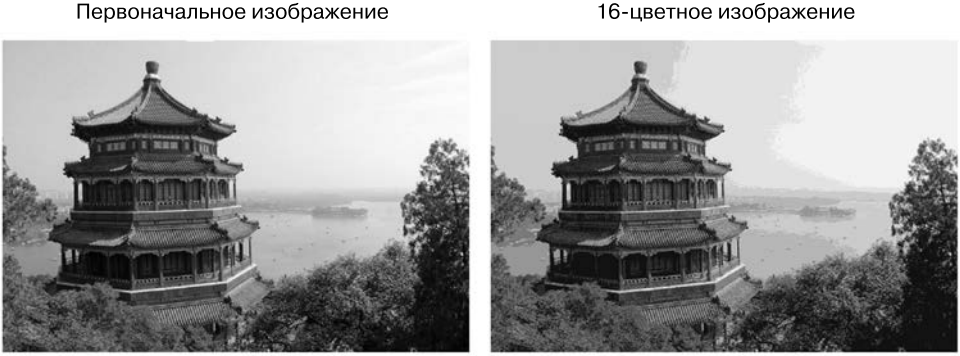

Рис. 5.123. Полноцветное изображение *(слева)* по сравнению с 16-цветным *(справа)*

```
In[24]:
china_recolored = new_colors.reshape(china.shape)

fig, ax = plt.subplots(1, 2, figsize=(16, 6),
          subplot_kw=dict(xticks=[], yticks=[]))
fig.subplots_adjust(wspace=0.05)
ax[0].imshow(china)
ax[0].set_title('Original Image', size=16) # Первоначальное изображение
ax[1].imshow(china_recolored)
ax[1].set_title('16-color Image', size=16); # 16-цветное изображение
```

Некоторые детали в изображении справа утрачены, но изображение в целом остается вполне узнаваемым. Коэффициент сжатия этого изображения — почти 1 миллион! Существуют лучшие способы сжатия информации в изображениях, но этот пример демонстрирует возможности творческого подхода к методам машинного обучения без учителя, таким как метод k-средних.

Заглянем глубже: смеси Гауссовых распределений

Модель кластеризации методом *k*-средних, которую мы обсуждали в предыдущем разделе, проста и относительно легка для понимания, но ее простота приводит к сложностям в применении. В частности, не вероятностная природа метода *k*-средних и использование им простого расстояния от центра кластера для определения принадлежности к кластеру приводит к низкой эффективности во многих встречающихся на практике ситуациях. В этом разделе мы рассмотрим смеси Гауссовых распределений, которые можно рассматривать в качестве развития идей метода *k*-средних, но которые могут также стать мощным инструментом для статистических оценок, выходящих за пределы простой кластеризации. Начнем с обычных импортов:

```
In[1]: %matplotlib inline
       import matplotlib.pyplot as plt
       import seaborn as sns; sns.set()
       import numpy as np
```

Причины появления GMM: недостатки метода k-средних

Рассмотрим некоторые недостатки метода *k*-средних и задумаемся о том, как можно усовершенствовать кластерную модель. Как мы уже видели в предыдущем разделе, в случае простых, хорошо разделяемых данных метод *k*-средних обеспечивает удовлетворительные результаты кластеризации.

Например, для случая простых «пятен» данных алгоритм *k*-средних позволяет быстро маркировать кластеры достаточно близко к тому, как мы бы маркировали их на глаз (рис. 5.124):

```
In[2]: # Генерируем данные
       from sklearn.datasets.samples_generator import make_blobs
       X, y_true = make_blobs(n_samples=400, centers=4,
                              cluster_std=0.60, random_state=0)
       X = X[:, ::-1] # Транспонируем для удобства оси координат
```

```
In[3]: # Выводим данные на график с полученными методом k-средних метками
       from sklearn.cluster import KMeans
       kmeans = KMeans(4, random_state=0)
       labels = kmeans.fit(X).predict(X)
       plt.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis');
```

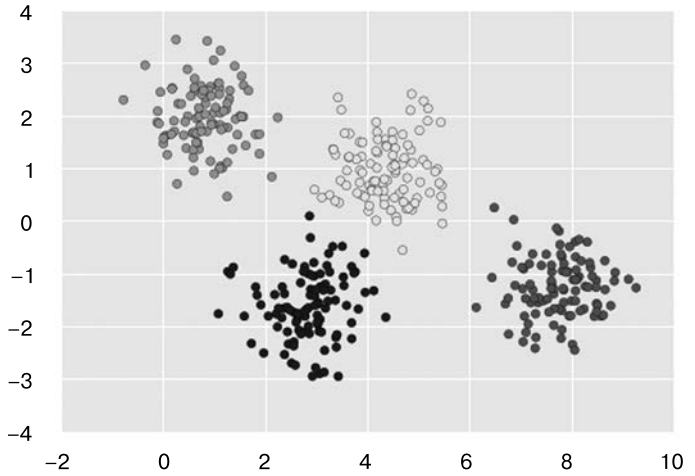

Рис. 5.124. Полученные методом k-средних метки для простых данных

Интуитивно мы можем ожидать, что при кластеризации одним точкам метки присваиваются с большей долей достоверности, чем другим. Например, два средних кластера чуть-чуть пересекаются, поэтому нельзя быть уверенными в том, к какому из них отнести точки, находящиеся посередине между ними. К сожалению, в модели *k*-средних отсутствует внутренняя мера вероятности или достоверности отнесения точек к кластерам (хотя можно использовать бутстрэппинг для оценки этой вероятности). Для этого необходимо рассмотреть возможность обобщения модели.

Модель *k*-средних можно рассматривать, в частности, как помещающую окружности (или в пространствах большей размерности гиперсферы) с центрами в центрах каждого из кластеров и радиусом, соответствующим расстоянию до наиболее удаленной точки кластера. Этот радиус задает жесткую границу соответствия точки кластеру в обучающей последовательности: все точки, находящиеся снаружи этой окружности, не считаются членами кластера. Визуализируем эту модель кластера с помощью следующей функции (рис. 5.125):

```
In[4]:
from sklearn.cluster import KMeans
from scipy.spatial.distance import cdist

def plot_kmeans(kmeans, X, n_clusters=4, rseed=0, ax=None):
    labels = kmeans.fit_predict(X)

    # Выводим на рисунок входные данные
    ax = ax or plt.gca()
    ax.axis('equal')
    ax.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis', zorder=2)

    # Выводим на рисунок представление модели k-средних
    centers = kmeans.cluster_centers_
    radii = [cdist(X[labels == i], [center]).max()
```

```
            for i, center in enumerate(centers)]
    for c, r in zip(centers, radii):
        ax.add_patch(plt.Circle(c, r, fc='#CCCCCC', lw=3, alpha=0.5,
                                zorder=1))

In[5]: kmeans = KMeans(n_clusters=4, random_state=0)
       plot_kmeans(kmeans, X)
```

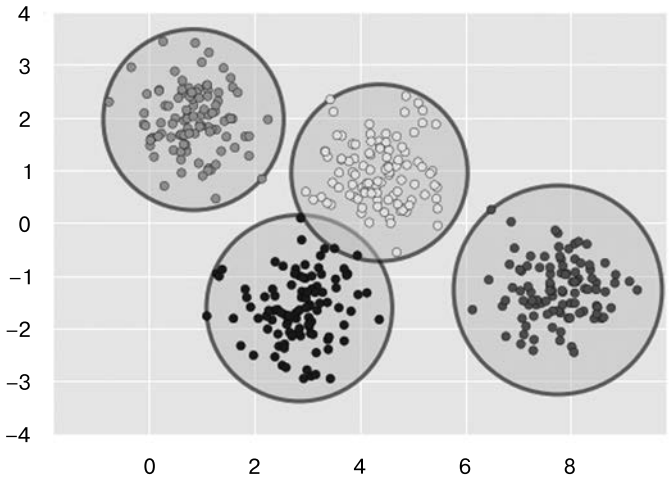

Рис. 5.125. Круглые кластеры, подразумеваемые моделью k-средних

Немаловажный нюанс, касающийся метода *k*-средних, состоит в том, что эти модели кластеров *обязательно должны иметь форму окружностей*: метод *k*-средних не умеет работать с овальными или эллипсовидными кластерами. Так, например, если несколько преобразовать те же данные, присвоенные метки окажутся перепутаны (рис. 5.126).

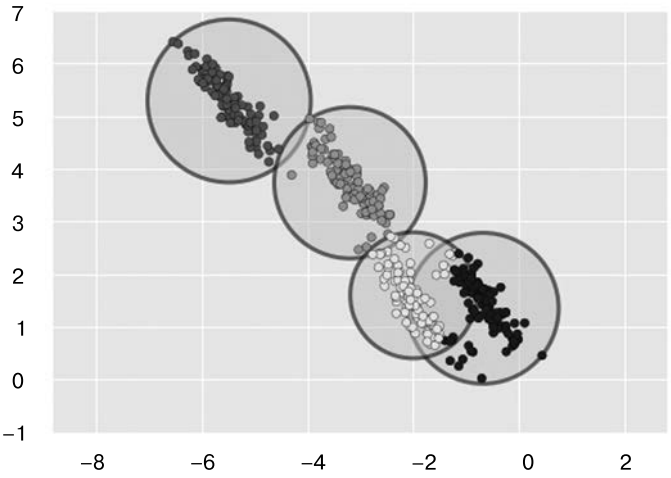

Рис. 5.126. Неудовлетворительная работа метода k-средних в случае кластеров некруглой формы

```
In[6]: rng = np.random.RandomState(13)
       X_stretched = np.dot(X, rng.randn(2, 2))

       kmeans = KMeans(n_clusters=4, random_state=0)
       plot_kmeans(kmeans, X_stretched)
```

Визуально заметно, что форма этих преобразованных кластеров некруглая, а значит, круглые кластеры плохо подойдут для их описания. Тем не менее метод *k*-средних недостаточно гибок для учета этого нюанса и пытается втиснуть данные в четыре круглых кластера. Это приводит к перепутанным меткам кластеров в местах перекрытия получившихся окружностей, см. нижнюю правую часть графика.

Можно было бы попытаться решить эту проблему путем предварительной обработки данных с помощью PCA (см. раздел «Заглянем глубже: метод главных компонент» этой главы), но на практике нет никаких гарантий, что подобная глобальная операция позволит разместить по окружностям отдельные точки данных.

Отсутствие гибкости в вопросе формы кластеров и отсутствие вероятностного присвоения меток кластеров — два недостатка метода *k*-средних, означающих, что для многих наборов данных (особенно низкоразмерных) он будет работать не столь хорошо, как хотелось бы.

Можно попытаться избавиться от этих недостатков путем обобщения модели *k*-средних. Например, можно оценивать степень достоверности присвоения меток кластеров, сравнивая расстояния от каждой точки до *всех* центров кластеров, а не сосредотачивая внимание лишь на ближайшем. Можно также разрешить эллипсовидную форму границ кластеров, а не только круглую, чтобы учесть кластеры некруглой формы. Оказывается, что это базовые составляющие другой разновидности модели кластеризации — смеси Гауссовых распределений.

Обобщение EM-модели: смеси Гауссовых распределений

Смесь Гауссовых распределений (gaussian mixture model, GMM) нацелена на поиск многомерных Гауссовых распределений вероятностей, моделирующих наилучшим возможным образом любой исходный набор данных.

В простейшем случае смеси Гауссовых распределений можно использовать для поиска кластеров аналогично методу *k*-средних (рис. 5.127):

```
In[7]: from sklearn.mixture import GMM
       gmm = GMM(n_components=4).fit(X)
       labels = gmm.predict(X)
       plt.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis');
```

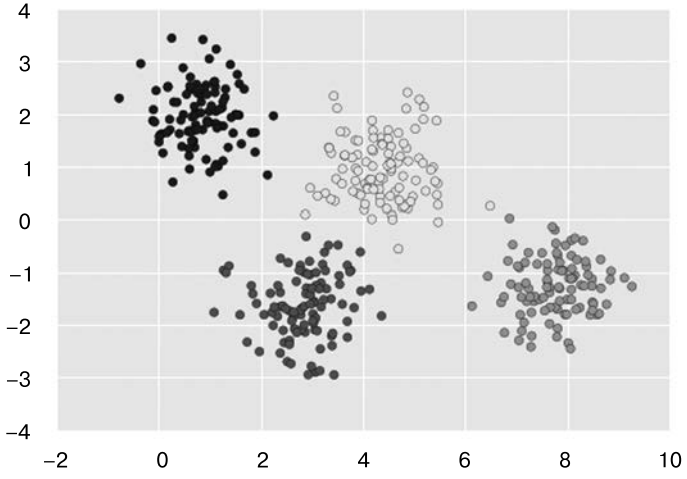

Рис. 5.127. Метки смеси Гауссовых распределений для наших данных

Но в силу того, что GMM содержит «под капотом» вероятностную модель, с ее помощью можно также присваивать метки кластеров на вероятностной основе — в библиотеке Scikit-Learn это можно сделать методом `predict_proba`. Он возвращает матрицу размера `[n_samples, n_clusters]`, содержащую оценки вероятностей принадлежности точки к конкретному кластеру:

```
In[8]: probs = gmm.predict_proba(X)
       print(probs[:5].round(3))

[[ 0.     0.     0.475  0.525]
 [ 0.     1.     0.     0.   ]
 [ 0.     1.     0.     0.   ]
 [ 0.     0.     0.     1.   ]
 [ 0.     1.     0.     0.   ]]
```

Для визуализации этой вероятности можно, например, сделать размеры точек пропорциональными степени достоверности их предсказания. Глядя на рис. 5.128, можно увидеть, что как раз точки на границах между кластерами отражают эту неопределенность отнесения точек к кластерам:

```
In[9]: size = 50 * probs.max(1) ** 2  # Возведение в квадрат усиливает
                                      # влияние различий
       plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', s=size);
```

«Под капотом» смесь Гауссовых распределений очень напоминает метод k-средних: она использует подход с максимизацией математического ожидания, который с качественной точки зрения делает следующее.

1. Выбирает первоначальные гипотезы для расположения и формы кластеров.
2. Повторяет до достижения сходимости:

- *E-шаг* — для каждой точки находит веса, кодирующие вероятность ее принадлежности к каждому кластеру;
- *M-шаг* — для каждого кластера корректирует его расположение, нормализацию и форму на основе информации обо *всех* точках данных с учетом весов.

В результате каждый кластер оказывается связан не со сферой с четкой границей, а с гладкой Гауссовой моделью. Аналогично подходу максимизации математического ожидания из метода k-средних этот алгоритм иногда может промахиваться мимо наилучшего из возможных решений, поэтому на практике применяют несколько случайных наборов начальных значений.

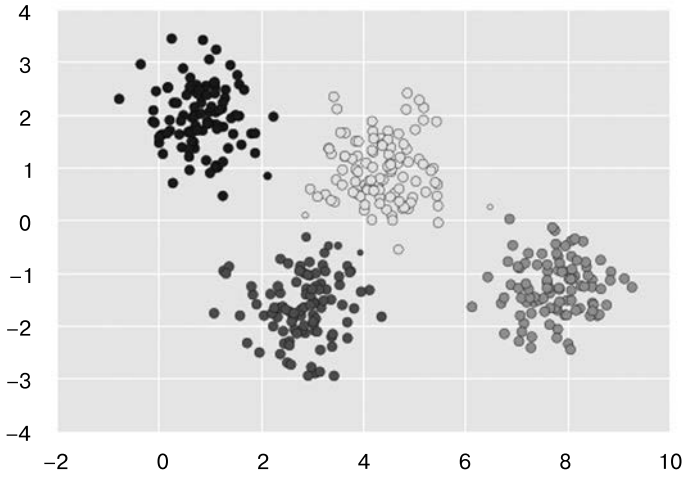

Рис. 5.128. Вероятностные метки GMM

Создадим функцию для упрощения визуализации расположений и форм кластеров метода GMM путем рисования эллипсов на основе получаемой на выходе `gmm` информации:

```
In[10]:
from matplotlib.patches import Ellipse

def draw_ellipse(position, covariance, ax=None, **kwargs):
    """Рисует эллипс с заданными расположением и ковариацией"""
    ax = ax or plt.gca()

    # Преобразуем ковариацию к главным осям координат
    if covariance.shape == (2, 2):
        U, s, Vt = np.linalg.svd(covariance)
        angle = np.degrees(np.arctan2(U[1, 0], U[0, 0]))
        width, height = 2 * np.sqrt(s)
    else:
        angle = 0
        width, height = 2 * np.sqrt(covariance)
```

```
    # Рисуем эллипс
    for nsig in range(1, 4):
        ax.add_patch(Ellipse(position, nsig * width, nsig * height,
                             angle, **kwargs))

def plot_gmm(gmm, X, label=True, ax=None):
    ax = ax or plt.gca()
    labels = gmm.fit(X).predict(X)
    if label:
        ax.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis', zorder=2)
    else:
        ax.scatter(X[:, 0], X[:, 1], s=40, zorder=2)
    ax.axis('equal')
    w_factor = 0.2 / gmm.weights_.max()
    for pos, covar, w in zip(gmm.means_, gmm.covars_, gmm.weights_):
        draw_ellipse(pos, covar, alpha=w * w_factor)
```

После этого можем посмотреть, какие результаты выдает четырехкомпонентный метод GMM на наших данных (рис. 5.129):

```
In[11]: gmm = GMM(n_components=4, random_state=42)
        plot_gmm(gmm, X)
```

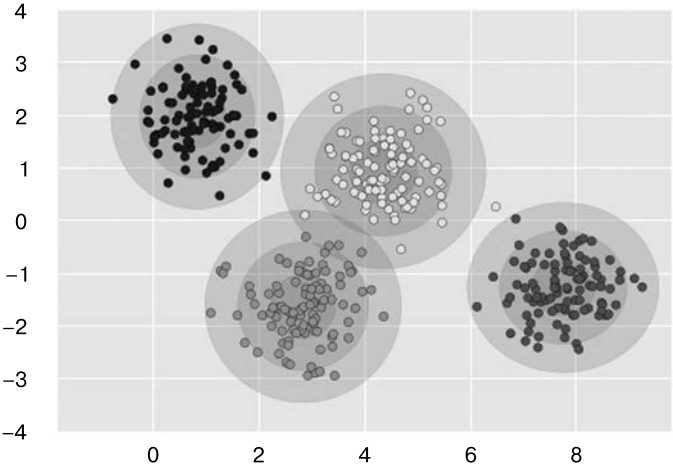

Рис. 5.129. Четырехкомпонентный метод GMM в случае круглых кластеров

Аналогично можно воспользоваться подходом GMM для «растянутого» набора данных. С учетом полной ковариации модель будет подходить даже для очень продолговатых, вытянутых в длину кластеров (рис. 5.130):

```
In[12]: gmm = GMM(n_components=4, covariance_type='full', random_state=42)
        plot_gmm(gmm, X_stretched)
```

Из этого ясно, что метод GMM решает две известные нам основные практические проблемы метода *k*-средних.

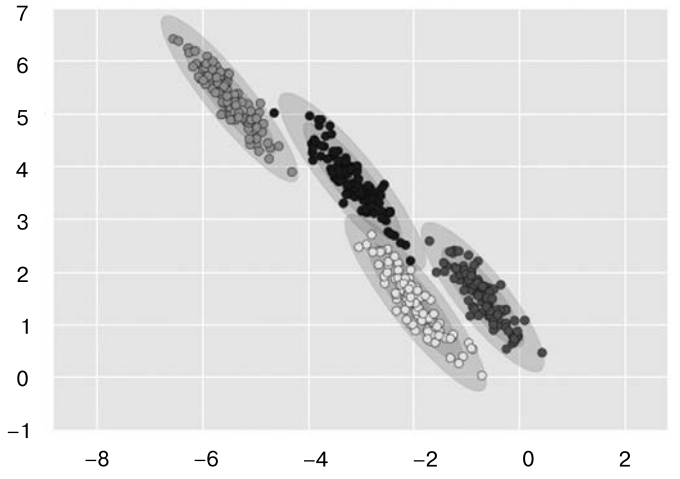

Рис. 5.130. Четырехкомпонентный метод GMM в случае некруглых кластеров

Выбор типа ковариации

Если вы внимательно посмотрите на предыдущие фрагменты кода, то увидите, что в каждом из них были заданы различные значения параметра `covariance_type`. Этот гиперпараметр управляет степенями свободы форм кластеров. Очень важно для любой задачи задавать его значения аккуратно. Значение его по умолчанию — `covariance_type="diag"`, означающее возможность независимого задания размеров кластера по всем измерениям с выравниванием полученного эллипса по осям координат.

Несколько более простая и быстро работающая модель — `covariance_type="spherical"`, ограничивающая форму кластера таким образом, что все измерения равнозначны между собой. Получающаяся в этом случае кластеризация будет аналогична методу *k*-средних, хотя и не полностью идентична.

Вариант с `covariance_type="full"` представляет собой более сложную и требующую бóльших вычислительных затрат модель (особенно при росте числа измерений), в которой любой из кластеров может быть эллипсом с произвольной ориентацией.

Графическое представление этих трех вариантов для одного кластера приведено на рис. 5.131.

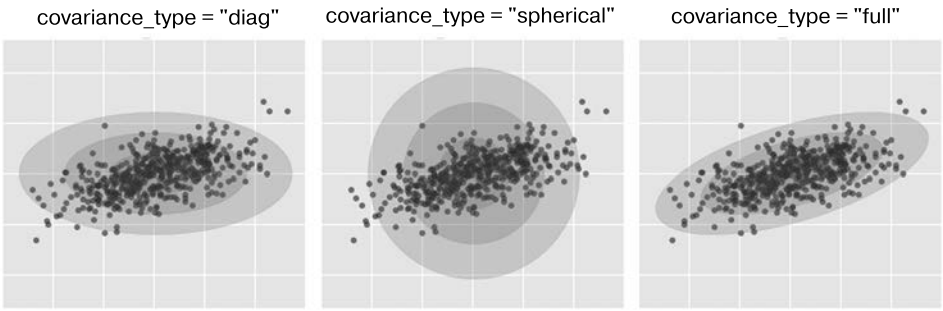

Рис. 5.131. Визуализация типов ковариации метода GMM

GMM как метод оценки плотности распределения

Хотя GMM часто относят к алгоритмам кластеризации, по существу, это алгоритм, предназначенный для *оценки плотности распределения*. Таким образом, аппроксимация каких-либо данных методом GMM формально является не моделью кластеризации, а порождающей вероятностной моделью, описывающей распределение данных.

В качестве примера изучим данные, сгенерированные с помощью функции `make_moons` библиотеки Scikit-Learn (показанные на рис. 5.132), которые мы уже рассматривали в разделе «Заглянем глубже: кластеризация методом k-средних» этой главы:

```
In[13]: from sklearn.datasets import make_moons
        Xmoon, ymoon = make_moons(200, noise=.05, random_state=0)
        plt.scatter(Xmoon[:, 0], Xmoon[:, 1]);
```

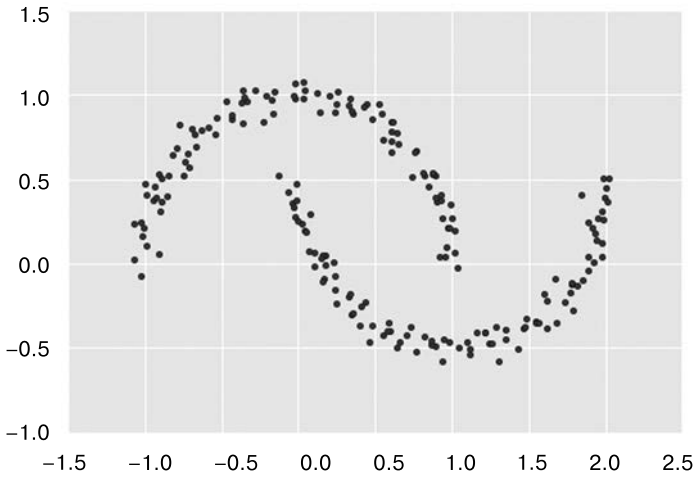

Рис. 5.132. Использование метода GMM для кластеров с нелинейными границами

Если попытаться использовать для этих данных двухкомпонентный GMM, рассматриваемый как модель кластеризации, то практическая пригодность результатов окажется сомнительной (рис. 5.133):

```
In[14]: gmm2 = GMM(n_components=2, covariance_type='full', random_state=0)
        plot_gmm(gmm2, Xmoon)
```

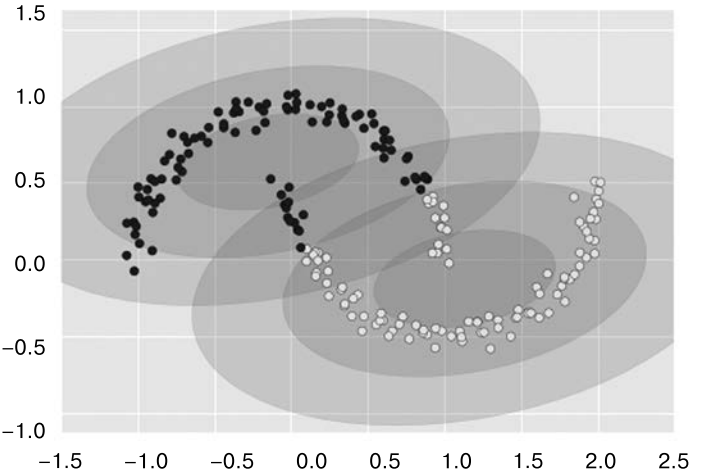

Рис. 5.133. Двухкомпонентная GMM-аппроксимация в случае нелинейных кластеров

Но если взять намного больше компонент и проигнорировать метки кластеров, мы получим намного более подходящую для исходных данных аппроксимацию (рис. 5.134):

```
In[15]: gmm16 = GMM(n_components=16, covariance_type='full', random_state=0)
        plot_gmm(gmm16, Xmoon, label=False)
```

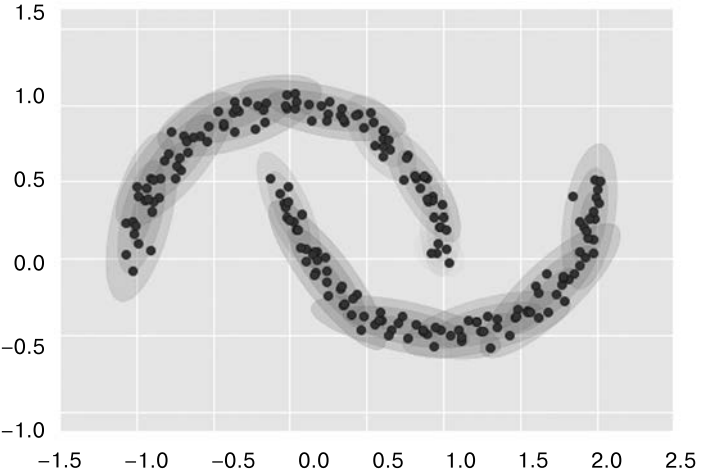

Рис. 5.134. Использование большого количества кластеров GMM для моделирования распределения точек

В данном случае смесь 16 нормальных распределений служит не для поиска отдельных кластеров данных, а для моделирования общего *распределения* входных данных. Это порождающая модель распределения, то есть GMM предоставляет нам способ генерации новых случайных данных, распределенных аналогично исходным. Например, вот 400 новых точек, полученных из этой аппроксимации наших исходных данных 16-компонентным алгоритмом GMM (рис. 5.135):

```
In[16]: Xnew = gmm16.sample(400, random_state=42)
        plt.scatter(Xnew[:, 0], Xnew[:, 1]);
```

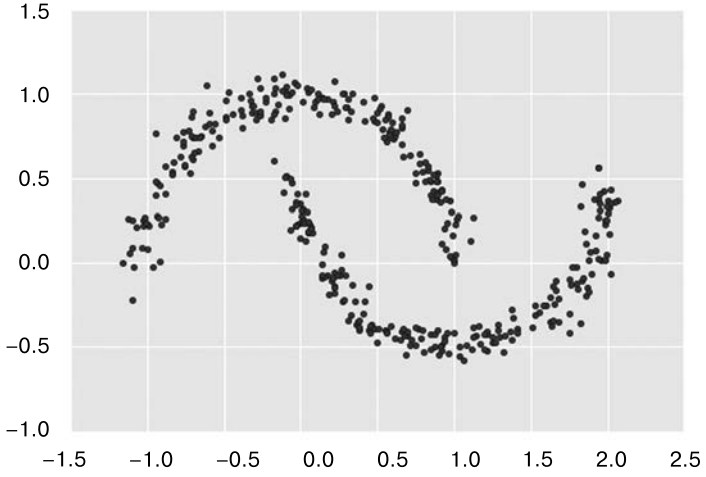

Рис. 5.135. Новые данные, полученные из 16-компонентного GMM

Метод GMM — удобное гибкое средство моделирования произвольного многомерного распределения данных.

Сколько компонент необходимо?

Благодаря тому что GMM — порождающая модель, у нас появляется естественная возможность определения оптимального количества компонент для заданного набора данных. Порождающая модель, по существу, представляет собой распределение вероятности для набора данных, поэтому можно легко вычислить *функцию правдоподобия* (likelihood function) для лежащих в ее основе данных, используя перекрестную проверку во избежание переобучения. Другой способ введения поправки на переобучение — подстройка функции правдоподобия модели с помощью некоторого аналитического критерия, например информационного критерия Акаике (Akaike information criterion, AIC, см.: https://ru.wikipedia.org/wiki/ Информационный_критерий_Акаике) или байесовского информационного критерия (bayesian information

criterion, BIC, см.: https://ru.wikipedia.org/wiki/Информационный_критерий). Оцениватель GMM библиотеки Scikit-Learn включает встроенные методы для вычисления этих критериев, что сильно упрощает указанный подход.

Посмотрим на критерии AIC и BIC как функции от количества компонент GMM для нашего набора данных `moon` (рис. 5.136):

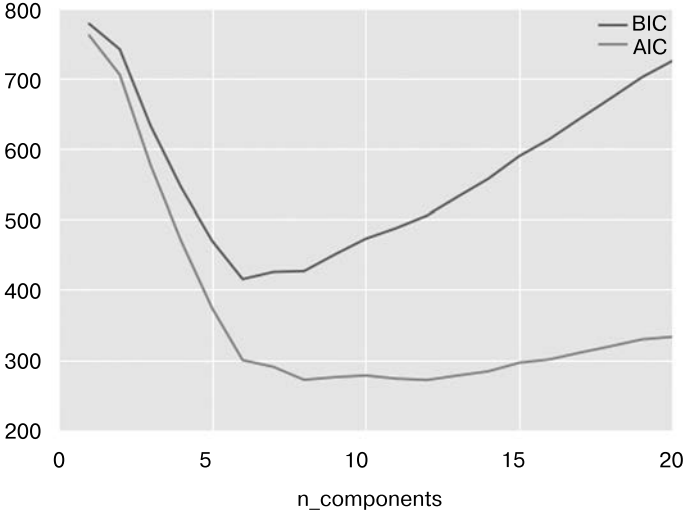

Рис. 5.136. Визуализация AIC и BIC с целью выбора количества компонент GMM

```
In[17]: n_components = np.arange(1, 21)
        models = [GMM(n, covariance_type='full', random_state=0).fit(Xmoon)
                  for n in n_components]

        plt.plot(n_components, [m.bic(Xmoon) for m in models], label='BIC')
        plt.plot(n_components, [m.aic(Xmoon) for m in models], label='AIC')
        plt.legend(loc='best')
        plt.xlabel('n_components');
```

Оптимальное количество кластеров — то, которое минимизирует AIC или BIC, в зависимости от требуемой аппроксимации. Согласно AIC, наших 16 компонент, вероятно, слишком много, лучше взять 8–12. Как это обычно бывает в подобных задачах, критерий BIC говорит в пользу более простой модели.

Обратите внимание на важный момент: подобный метод выбора числа компонент представляет собой меру успешности работы GMM как *оценивателя плотности* распределения, а не как *алгоритма кластеризации*. Я советовал бы вам рассматривать GMM в основном как оцениватель плотности и использовать его для кластеризации только заведомо простых наборов данных.

Пример: использование метода GMM для генерации новых данных

Мы увидели простой пример применения метода GMM в качестве порождающей модели данных с целью создания новых выборок на основе соответствующего исходным данным распределения. В этом разделе мы продолжим воплощение этой идеи и сгенерируем *новые рукописные цифры* на основе корпуса стандартных цифр, который мы использовали ранее.

Для начала загрузим набор данных по цифрам с помощью инструментов библиотеки Scikit-Learn:

```
In[18]: from sklearn.datasets import load_digits
        digits = load_digits()
        digits.data.shape

Out[18]: (1797, 64)
```

Далее выведем на рисунок первые 100 из них, чтобы вспомнить, с чем мы имеем дело (рис. 5.137):

```
In[19]: def plot_digits(data):
            fig, ax = plt.subplots(10, 10, figsize=(8, 8),
                                   subplot_kw=dict(xticks=[], yticks=[]))
            fig.subplots_adjust(hspace=0.05, wspace=0.05)
            for i, axi in enumerate(ax.flat):
                im = axi.imshow(data[i].reshape(8, 8), cmap='binary')
                im.set_clim(0, 16)
        plot_digits(digits.data)
```

Наш набор данных состоит почти из 1800 цифр в 64 измерениях. Построим на их основе GMM, чтобы сгенерировать еще. У смесей Гауссовых распределений могут быть проблемы со сходимостью в пространстве столь высокой размерности, поэтому начнем с применения обратимого алгоритма для понижения размерности данных. Воспользуемся для этой цели простым алгоритмом PCA с сохранением 99 % дисперсии в проекции данных:

```
In[20]: from sklearn.decomposition import PCA
        pca = PCA(0.99, whiten=True)
        data = pca.fit_transform(digits.data)
        data.shape

Out[20]: (1797, 41)
```

Результат оказался 41-мерным, то есть размерность была снижена почти на 1/3 практически без потерь информации. Воспользуемся для этих спроецированных данных критерием AIC для определения необходимого количества компонент GMM (рис. 5.138):

```
In[21]: n_components = np.arange(50, 210, 10)
        models = [GMM(n, covariance_type='full', random_state=0)
```

```
          for n in n_components]
aics = [model.fit(data).aic(data) for model in models]
plt.plot(n_components, aics);
```

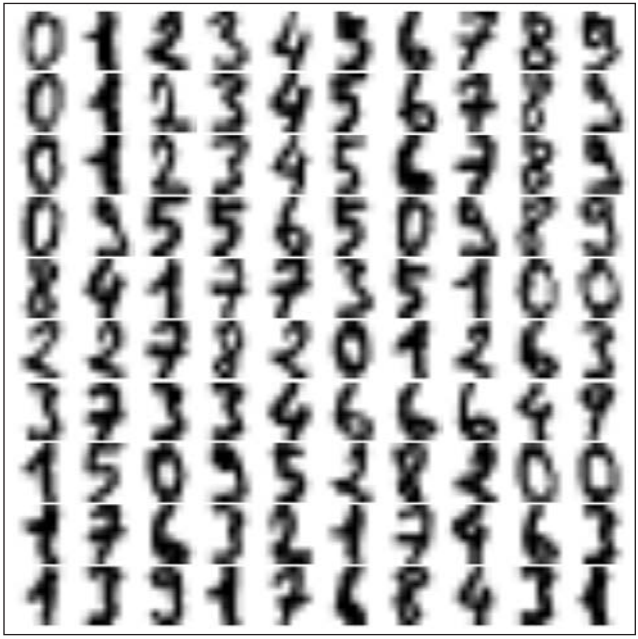

Рис. 5.137. Исходные рукописные цифры

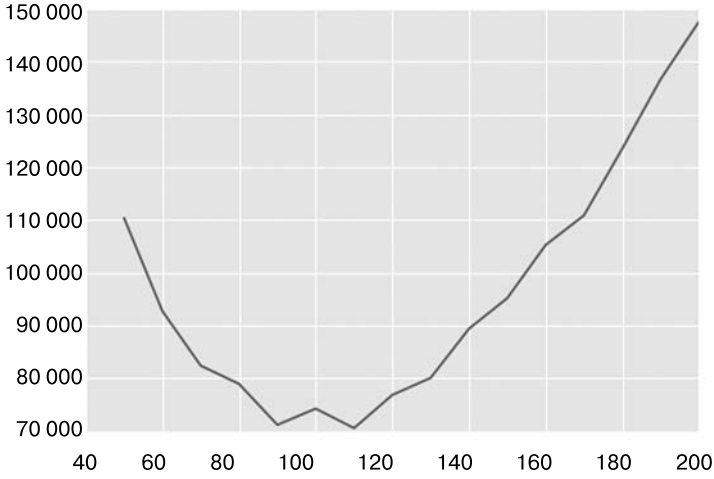

Рис. 5.138. Кривая AIC для выбора подходящего количества компонент GMM

Похоже, что AIC минимизируют примерно 110 компонент; этой моделью мы и воспользуемся. Обучим этот алгоритм на наших данных и убедимся, что он сошелся:

```
In[22]: gmm = GMM(110, covariance_type='full', random_state=0)
        gmm.fit(data)
        print(gmm.converged_)

True
```

Теперь можно сгенерировать 100 новых точек в этом 41-мерном пространстве, используя GMM как порождающую модель:

```
In[23]: data_new = gmm.sample(100, random_state=0)
        data_new.shape

Out[23]: (100, 41)
```

Наконец, можно воспользоваться обратным преобразованием объекта PCA для формирования новых цифр (рис. 5.139):

```
In[24]: digits_new = pca.inverse_transform(data_new)
        plot_digits(digits_new)
```

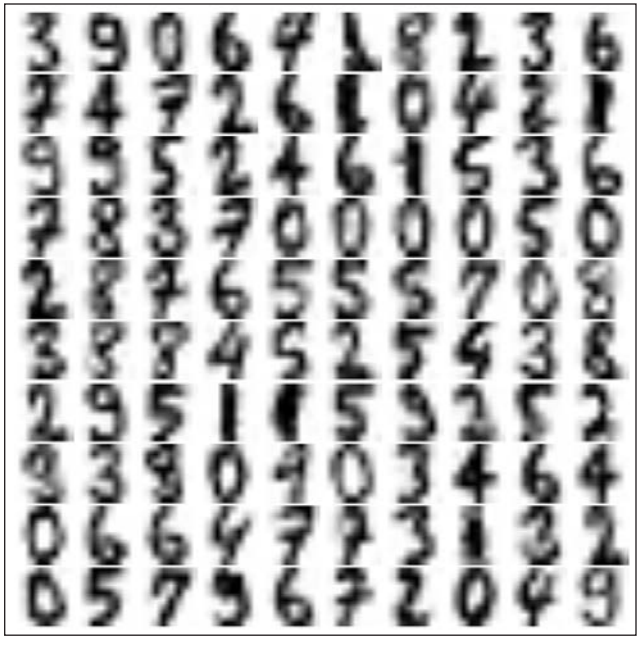

Рис. 5.139. «Новые» цифры, полученные случайным образом из модели оценивателя GMM

Результаты по большей части выглядят как вполне правдоподобные цифры из набора данных!

Резюмируем сделанное: мы смоделировали распределение для заданной выборки рукописных цифр таким образом, что смогли сгенерировать совершенно новые выборки цифр на основе этих данных: это рукописные цифры, не встречающиеся в исходном наборе данных, но отражающие общие признаки входных данных, смоделированные моделью смеси распределений. Подобная порождающая модель цифр может оказаться очень удобной в качестве компонента байесовского порождающего классификатора.

Заглянем глубже: ядерная оценка плотности распределения

В предыдущем разделе мы рассмотрели смеси Гауссовых распределений (GMM), представляющие собой своеобразный гибрид оценивателя для кластеризации и оценивателя плотности. Напомню, что оцениватель плотности — алгоритм, выдающий для D-мерного набора данных оценку D-мерного распределения вероятности, из которого взята эта выборка данных. Для этого алгоритм GMM представляет плотность распределения в виде взвешенной суммы Гауссовых распределений. Ядерная оценка плотности распределения (KDE) — в некотором смысле алгоритм, доводящий идею смеси Гауссовых функций до логического предела: в нем используется смесь, состоящая из одной Гауссовой компоненты *для каждой точки*, что приводит к непараметрическому оценивателю плотности. В этом разделе мы рассмотрим обоснования и сферы использования метода KDE. Начнем с обычных импортов:

```
In[1]: %matplotlib inline
       import matplotlib.pyplot as plt
       import seaborn as sns; sns.set()
       import numpy as np
```

Обоснование метода KDE: гистограммы

Оцениватель плотности — алгоритм, предназначенный для моделирования распределения вероятностей, на основе которого был сгенерирован набор данных. Вероятно, вы уже хорошо знакомы с одним простым оценивателем плотности для одномерных данных — гистограммой. Гистограмма делит данные на дискретные интервалы значений, подсчитывает число точек, попадающих в каждый из интервалов, после чего визуализирует результат интуитивно понятным образом.

Для примера сгенерируем данные на основе двух нормальных распределений:

```
In[2]:
        def make_data(N, f=0.3, rseed=1):
           rand = np.random.RandomState(rseed)
           x = rand.randn(N)
           x[int(f * N):] += 5
```

```
        return x
    x = make_data(1000)
```

Такую обычную гистограмму на основе числа точек можно создать с помощью функции `plt.hist()`. Задав параметр `normed` гистограммы, мы получим нормализованную гистограмму, в которой высота интервалов отражает не число точек, а плотность вероятности (рис. 5.140):

```
In[3]: hist = plt.hist(x, bins=30, normed=True)
```

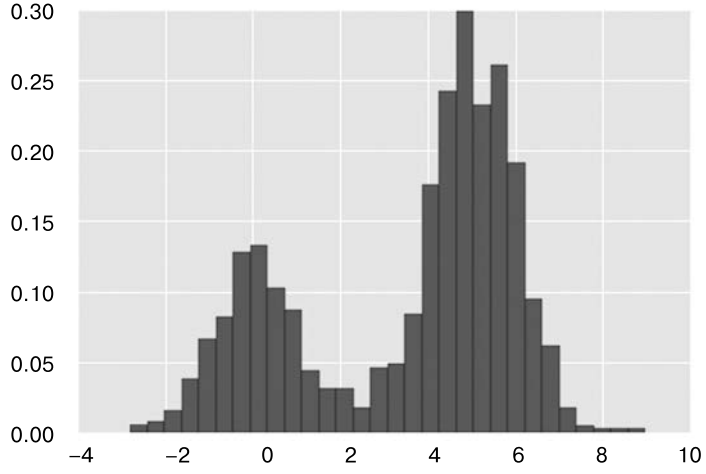

Рис. 5.140. Данные, полученные из сочетания нормальных распределений

Обратите внимание, что при равных интервалах такая нормализация просто меняет масштаб по оси *Y*, при этом относительная высота интервалов остается такой же, как и в гистограмме для числа точек. Нормализация выбирается таким образом, чтобы общая площадь под гистограммой была равна 1, в чем можно убедиться, глядя на вывод функции построения гистограммы:

```
In[4]: density, bins, patches = hist
       widths = bins[1:] - bins[:-1]
       (density * widths).sum()

Out[4]: 1.0
```

Одна из проблем использования гистограмм заключается в том, что конкретный выбор размера и расположения интервалов может привести к представлениям с качественно различными признаками. Например, если посмотреть на версию этих данных из 20 точек, конкретный выбор интервалов может привести к совершенно другой интерпретации данных! Рассмотрим следующий пример (рис. 5.141):

```
In[5]: x = make_data(20)
       bins = np.linspace(-5, 10, 10)
```

```
In[6]: fig, ax = plt.subplots(1, 2, figsize=(12, 4),
                              sharex=True, sharey=True,
                              subplot_kw={'xlim':(-4, 9),
                                          'ylim':(-0.02, 0.3)})
       fig.subplots_adjust(wspace=0.05)
       for i, offset in enumerate([0.0, 0.6]):
           ax[i].hist(x, bins=bins + offset, normed=True)
           ax[i].plot(x, np.full_like(x, -0.01), '|k',
                      markeredgewidth=1)
```

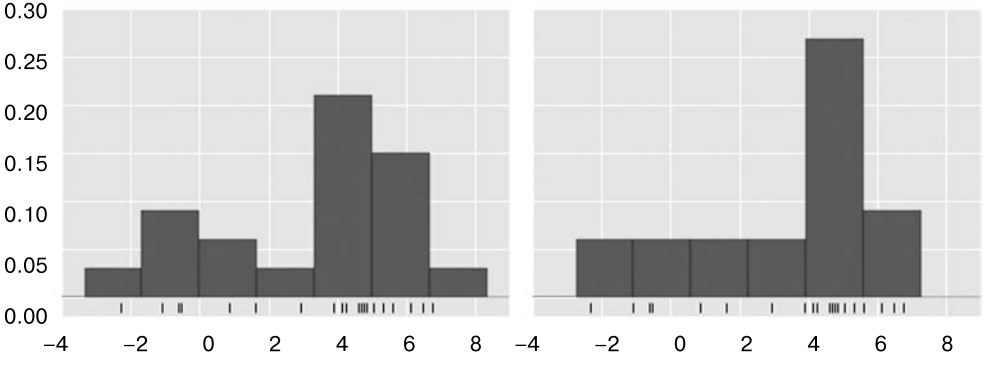

Рис. 5.141. Проблема гистограмм: различная интерпретация в зависимости от расположения интервалов

Из гистограммы слева очевидно, что мы имеем дело с бимодальным распределением. Справа же мы видим унимодальное распределение с длинным «хвостом». Без вышеприведенного кода вы бы вряд ли предположили, что эти две гистограммы были построены на одних данных. С учетом этого возникает вопрос: как можно доверять интуиции, полученной на основе подобных гистограмм? Что с этим можно сделать?

Небольшое отступление: гистограммы можно рассматривать как «стопки» блоков, в котором для каждой точки набора данных в соответствующий интервал помещается один блок. Посмотрим на это непосредственно (рис. 5.142):

```
In[7]: fig, ax = plt.subplots()
       bins = np.arange(-3, 8)
       ax.plot(x, np.full_like(x, -0.1), '|k',
               markeredgewidth=1)
       for count, edge in zip(*np.histogram(x, bins)):
           for i in range(count):
               ax.add_patch(plt.Rectangle((edge, i), 1, 1,
                                          alpha=0.5))
       ax.set_xlim(-4, 8)
       ax.set_ylim(-0.2, 8)

Out[7]: (-0.2, 8)
```

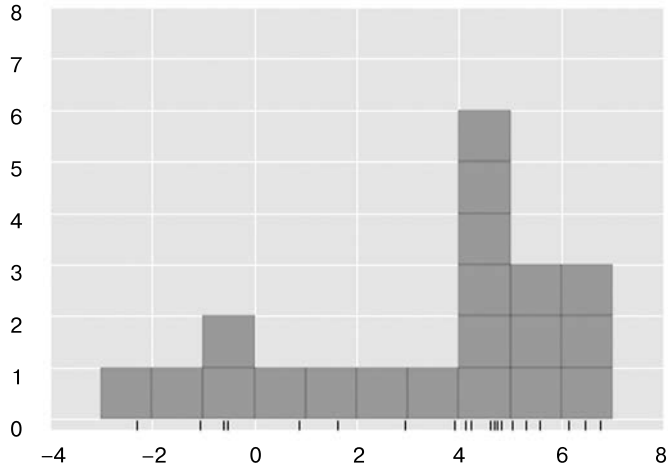

Рис. 5.142. Гистограмма как «стопки» блоков

В основе проблемы с нашими двумя разбиениями по интервалам лежит тот факт, что высота «стопки» блоков часто отражает не фактическую плотность близлежащих точек, а случайные стечения обстоятельств, выражающиеся в выравнивании интервалов по точкам данных. Это рассогласование точек и их блоков может приводить к наблюдаемым здесь неудовлетворительным результатам гистограмм. Но что, если вместо складывания в «стопки» блоков, выровненных *по интервалам*, мы складывали бы блоки, выровненные *по точкам, которым они соответствуют*? В этом случае блоки не были бы выровненными, но получить требуемый результат можно было бы, сложив их вклады в значение в каждом месте на оси *X*. Давайте это проделаем (рис. 5.143):

```
In[8]: x_d = np.linspace(-4, 8, 2000)
       density = sum((abs(xi - x_d) < 0.5) for xi in x)

       plt.fill_between(x_d, density, alpha=0.5)
       plt.plot(x, np.full_like(x, -0.1), '|k', markeredgewidth=1)

       plt.axis([-4, 8, -0.2, 8]);
```

Результат выглядит немного неряшливо, но отражает подлинные характеристики данных гораздо надежнее, чем стандартная гистограмма. Тем не менее неровные края не слишком приятны для глаз, да и не отражают никаких фактических свойств данных. Для их сглаживания можно попытаться заменить блоки в каждой точке гладкой функцией, например Гауссовой. Мы будем использовать вместо блока в каждой точке стандартную нормальную кривую (рис. 5.144):

```
In[9]: from scipy.stats import norm
       x_d = np.linspace(-4, 8, 1000)
       density = sum(norm(xi).pdf(x_d) for xi in x)
```

```
plt.fill_between(x_d, density, alpha=0.5)
plt.plot(x, np.full_like(x, -0.1), '|k', markeredgewidth=1)

plt.axis([-4, 8, -0.2, 5]);
```

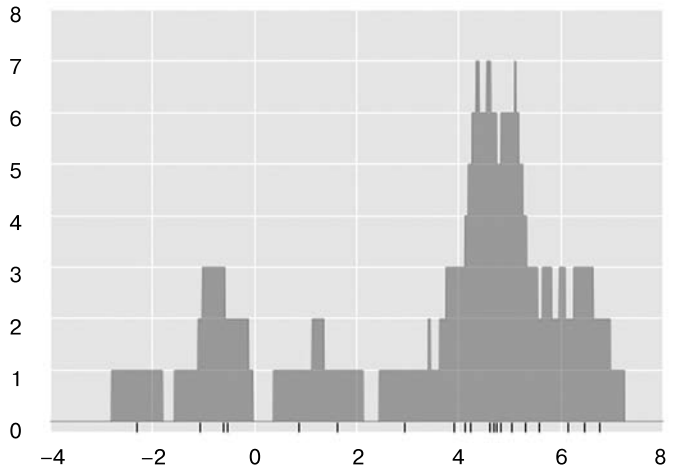

Рис. 5.143. Гистограмма с центрированием блоков по отдельным точкам — пример ядерной оценки плотности распределения

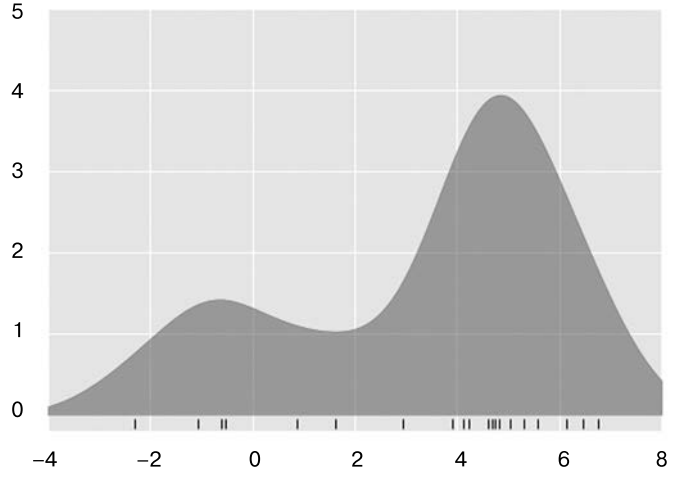

Рис. 5.144. Ядерная оценка плотности распределения с Гауссовым ядром

Этот сглаженный график со вкладом Гауссового распределения в соответствующих всем исходным точкам местам обеспечивает намного более точное представление о форме распределения данных, причем с намного меньшей дисперсией, то есть отличия выборок приводят к намного меньшим его изменениям.

Два последних графика представляют собой примеры одномерной ядерной оценки плотности распределения: в первом используется так называемое ядро типа «цилиндр», а во втором — Гауссово ядро. Рассмотрим ядерную оценку плотности распределения более подробно.

Ядерная оценка плотности распределения на практике

Свободными параметрами ядерной оценки плотности распределения являются *ядро* (kernel), определяющее форму распределения в каждой точке, и *ширина ядра* (kernel bandwidth), определяющая размер ядра в каждой точке. На практике для ядерной оценки плотности распределения существует множество различных ядер: в частности, реализация KDE библиотеки Scikit-Learn поддерживает использование одного из шести ядер, о которых вы можете прочитать в посвященной оцениванию плотности документации библиотеки Scikit-Learn (http://scikit-learn.org/stable/modules/density.html).

Хотя в языке Python реализовано несколько вариантов ядерной оценки плотности (особенно в пакетах SciPy и StatsModels), я предпочитаю использовать вариант из Scikit-Learn по причине гибкости и эффективности. Он реализован в оценивателе `sklearn.neighbors.KernelDensity`, умеющем работать с KDE в многомерном пространстве с одним из шести ядер и одной из нескольких дюжин метрик. В силу того что метод KDE может потребовать значительных вычислительных затрат, этот оцениватель использует «под капотом» алгоритм на основе деревьев и умеет достигать компромисса между временем вычислений и точностью с помощью параметров `atol` (absolute tolerance, допустимая абсолютная погрешность) и `rtol` (relative tolerance, допустимая относительная погрешность). Определить ширину ядра — свободный параметр — можно стандартными инструментами перекрестной проверки библиотеки Scikit-Learn.

Рассмотрим простой пример воспроизведения предыдущего графика с помощью оценивателя `KernelDensity` библиотеки Scikit-Learn (рис. 5.145):

```
In[10]: from sklearn.neighbors import KernelDensity

        # Создание экземпляра модели KDE и ее обучение
        kde = KernelDensity(bandwidth=1.0, kernel='gaussian')
        kde.fit(x[:, None])

        # score_samples возвращает логарифм плотности
        # распределения вероятности
        logprob = kde.score_samples(x_d[:, None])

        plt.fill_between(x_d, np.exp(logprob), alpha=0.5)
        plt.plot(x, np.full_like(x, -0.01), '|k', markeredgewidth=1)
        plt.ylim(-0.02, 0.22)

Out[10]: (-0.02, 0.22)
```

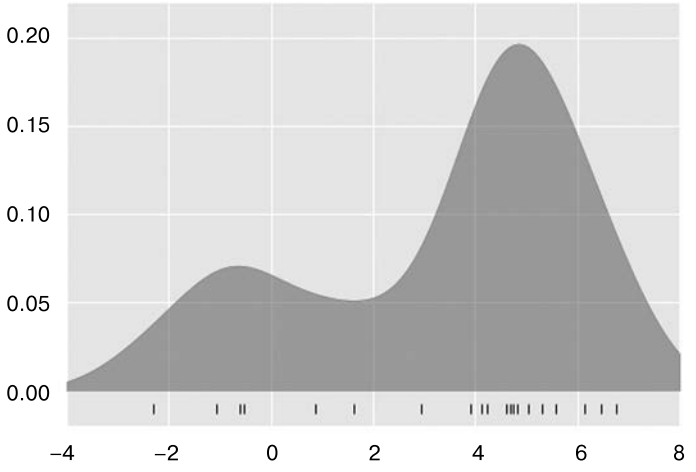

Рис. 5.145. Ядерная оценка плотности, вычисленная с помощью библиотеки Scikit-Learn

Результат нормализован так, что площадь под кривой равна 1.

Выбор ширины ядра путем перекрестной проверки. Выбор ширины ядра в методе KDE исключительно важен для получения удовлетворительной оценки плотности. Это и есть тот параметр, который при оценке плотности служит для выбора компромисса между систематической ошибкой и дисперсией. Слишком маленькая ширина ядра приводит к оценке с высокой дисперсией, то есть переобучению, при которой наличие или отсутствие одной-единственной точки может серьезно повлиять на модель. Слишком же широкое ядро ведет к оценке со значительной систематической ошибкой, то есть недообучению, при которой структура данных размывается этим широким ядром.

В статистике существует долгая предыстория методов быстрой оценки оптимальной ширины ядра на основе довольно строгих допущений относительно данных: если заглянуть в реализации метода KDE в пакетах SciPy и StatModels, например, можно увидеть основанные на некоторых из этих правил реализации.

В контексте машинного обучения мы уже видели, что выбор подобных гиперпараметров зачастую производится эмпирически посредством перекрестной проверки. Учитывая это, оцениватель `KernelDensity` из библиотеки Scikit-Learn спроектирован в расчете на непосредственное использование его в стандартных инструментах Scikit-Learn для поиска по сетке. В данном случае мы воспользуемся классом `GridSearchCV`, чтобы выбрать оптимальную ширину ядра для предыдущего набора данных. Поскольку наш набор данных очень невелик, мы будем использовать перекрестную проверку по отдельным объектам, при

которой размер обучающей последовательности максимален для каждого испытания перекрестной проверки:

```
In[11]: from sklearn.grid_search import GridSearchCV
        from sklearn.cross_validation import LeaveOneOut

        bandwidths = 10 ** np.linspace(-1, 1, 100)
        grid = GridSearchCV(KernelDensity(kernel='gaussian'),
                            {'bandwidth': bandwidths},
                            cv=LeaveOneOut(len(x)))
        grid.fit(x[:, None]);
```

Теперь можно узнать вариант ширины ядра, максимизирующий оценку эффективности модели (которая в этом случае по умолчанию представляет собой логарифмическую функцию правдоподобия):

```
In[12]: grid.best_params_

Out[12]: {'bandwidth': 1.1233240329780276}
```

Оптимальная ширина ядра оказалась очень близка к той, которую мы использовали выше в примере, где ширина была равно 1.0 (это ширина ядра по умолчанию объекта `scipy.stats.norm`).

Пример: KDE на сфере

Вероятно, чаще всего KDE используется для визуального представления распределений точек. Например, KDE встроен в библиотеку визуализации Seaborn (которую мы обсуждали в разделе «Визуализация с помощью пакета Seaborn» главы 4) и применяется там автоматически для визуализации точек в одномерном и двумерном пространствах.

В этом разделе мы рассмотрим несколько более сложный сценарий использования KDE для визуализации распределений. Воспользуемся следующими географическими данными, которые можно загрузить с помощью библиотеки Scikit-Learn: географическое распределение зафиксированных наблюдений особей двух южноамериканских млекопитающих — *Bradypus variegatus* (бурогорлый ленивец) и *Microryzomys minutus* (малая лесная рисовая крыса).

Извлечем данные с помощью библиотеки Scikit-Learn следующим образом[1]:

```
In[13]: from sklearn.datasets import fetch_species_distributions
```

[1] В апреле 2017 года этот набор данных был перемещен на другой сайт, что привело к неработоспособности данного кода в текущей версии библиотеки Scikit-Learn, но соответствующие изменения планируются к внесению в самое ближайшее время и должны быть доступны на момент выхода данной книги.

```
data = fetch_species_distributions()

# Получаем матрицы/массивы идентификаторов и местоположений животных
latlon = np.vstack([data.train['dd lat'],
                    data.train['dd long']]).T
species = np.array([d.decode('ascii').startswith('micro')
                    for d in data.train['species']], dtype='int')
```

После загрузки данных можно воспользоваться набором инструментов Basemap (упоминавшимся ранее в разделе «Отображение географических данных с помощью Basemap» главы 4) для отображения мест, где наблюдались особи этих двух видов на карте Южной Америки (рис. 5.146):

```
In[14]: from mpl_toolkits.basemap import Basemap
        from sklearn.datasets.species_distributions import construct_grids

        xgrid, ygrid = construct_grids(data)

        # Рисуем береговые линии с помощью Basemap
        m = Basemap(projection='cyl', resolution='c',
                    llcrnrlat=ygrid.min(), urcrnrlat=ygrid.max(),
                    llcrnrlon=xgrid.min(), urcrnrlon=xgrid.max())
        m.drawmapboundary(fill_color='#DDEEFF')
        m.fillcontinents(color='#FFEEDD')
        m.drawcoastlines(color='gray', zorder=2)
        m.drawcountries(color='gray', zorder=2)

        # Отображаем места, где наблюдались особи
        m.scatter(latlon[:, 1], latlon[:, 0], zorder=3,
                  c=species, cmap='rainbow', latlon=True);
```

Рис. 5.146. Места, где наблюдались особи, в обучающей последовательности

К сожалению, этот рисунок не дает хорошего представления о концентрации особей, поскольку точки могут перекрываться. Вряд ли вы догадаетесь по нему, что здесь показано более 1600 точек!

Воспользуемся ядерной оценкой плотности распределения, чтобы отобразить это распределение в более удобном для интерпретации виде — сглаженной индикации плотности на карте. Поскольку координатная система наложена на сферическую поверхность, а не на плоскость, воспользуемся метрикой `haversine`, подходящей для адекватного отображения расстояний на криволинейной поверхности.

Нам придется использовать немного шаблонного кода (один из недостатков набора инструментов Basemap), но смысл каждого блока кода должен быть вам вполне понятен (рис. 5.147):

```
In[15]:
# Настраиваем сетку данных для контурного графика
X, Y = np.meshgrid(xgrid[::5], ygrid[::5][::-1])
land_reference = data.coverages[6][::5, ::5]
land_mask = (land_reference > -9999).ravel()
xy = np.vstack([Y.ravel(), X.ravel()]).T xy = np.radians(xy[land_mask])

# Создаем два графика друг возле друга
fig, ax = plt.subplots(1, 2)
fig.subplots_adjust(left=0.05, right=0.95, wspace=0.05)
species_names = ['Bradypus Variegatus', 'Microryzomys Minutus']
cmaps = ['Purples', 'Reds']

for i, axi in enumerate(ax):
    axi.set_title(species_names[i])

    # Рисуем береговые линии с помощью Basemap
    m = Basemap(projection='cyl', llcrnrlat=Y.min(),
                urcrnrlat=Y.max(), llcrnrlon=X.min(),
                urcrnrlon=X.max(), resolution='c', ax=axi)
    m.drawmapboundary(fill_color='#DDEEFF')
    m.drawcoastlines()
    m.drawcountries()

    # Формируем сферическую ядерную оценку плотности распределения
    kde = KernelDensity(bandwidth=0.03, metric='haversine')
    kde.fit(np.radians(latlon[species == i]))

    # Выполняем расчеты только на поверхности Земли:
    # -9999 соответствует океану
    Z = np.full(land_mask.shape[0], -9999.0)
    Z[land_mask] = np.exp(kde.score_samples(xy))
    Z = Z.reshape(X.shape)

    # Рисуем изолинии плотности
    levels = np.linspace(0, Z.max(), 25)
    axi.contourf(X, Y, Z, levels=levels, cmap=cmaps[i])
```

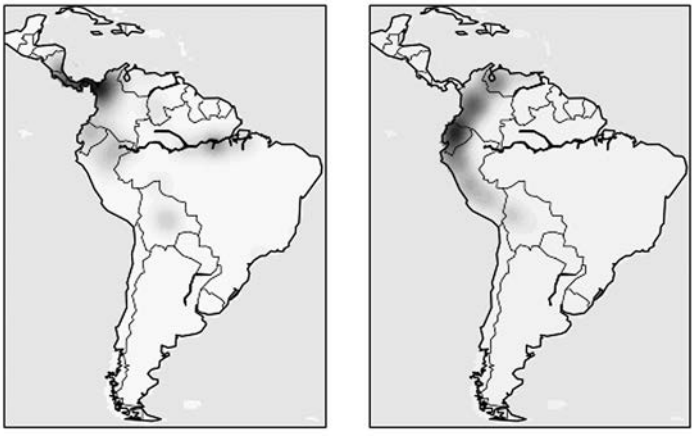

Рис. 5.147. Визуальное представление ядерной оценки плотности распределения особей

По сравнению с первоначальным простым контурным графиком, эта визуализация обеспечивает намного более понятную картину географического распределения наблюдений особей двух данных видов.

Пример: не столь наивный байес

В этом примере мы изучим выполнение байесовской порождающей классификации с помощью KDE и рассмотрим создание пользовательского оценивателя на основе архитектуры библиотеки Scikit-Learn.

В разделе «Заглянем глубже: наивная байесовская классификация» этой главы мы рассмотрели наивную байесовскую классификацию, в которой создали простые порождающие модели для всех классов и построили на их основе быстрый классификатор. В случае наивного байесовского классификатора порождающая модель — это просто выровненная по осям координат Гауссова функция. Алгоритм оценки плотности, например KDE, позволяет убрать «наивную» составляющую и произвести ту же самую классификацию с более сложными порождающими моделями для каждого из классов. Эта классификация остается байесовской, но уже не будет «наивной».

Общая методика порождающей классификации такова.

1. Разбиение обучающих данных по меткам.
2. Для каждого набора, находится порождающая модель путем обучения KDE. Это дает возможность вычисления функции правдоподобия $P(x \mid y)$ для каждого наблюдения x и метки y.

3. Вычисляем априорную вероятность принадлежности к классу (class prior), $P(y)$, на основе количества экземпляров каждого класса в обучающей последовательности.
4. Для неизвестной точки x апостериорная вероятность принадлежности к классу равна $P(y \mid x) \propto P(x \mid y)\, P(y)$. Метка каждой точки — класс, при котором достигается максимум этой апостериорной вероятности.

Данный алгоритм достаточно прост и интуитивно понятен. Несколько сложнее будет реализовать его с помощью фреймворка Scikit-Learn так, чтобы воспользоваться поиском по сетке и перекрестной проверкой.

Вот код, реализующий этот алгоритм на базе фреймворка Scikit-Learn, мы последовательно рассмотрим его блок за блоком:

```
In[16]: from sklearn.base import BaseEstimator, ClassifierMixin


        class KDEClassifier(BaseEstimator, ClassifierMixin):
            """Байесовская порождающая классификация на основе метода KDE

            Параметры
            ----------
            bandwidth : float
                Ширина ядра в каждом классе
            kernel : str
                Название ядра, передаваемое функции KernelDensity
            """
            def __init__(self, bandwidth=1.0, kernel='gaussian'):
                self.bandwidth = bandwidth
                self.kernel = kernel

            def fit(self, X, y):
                self.classes_ = np.sort(np.unique(y))
                training_sets = [X[y == yi] for yi in self.classes_]
                self.models_ = [KernelDensity(bandwidth=self.bandwidth,
                                              kernel=self.kernel).fit(Xi)
                                for Xi in training_sets]
                self.logpriors_ = [np.log(Xi.shape[0] / X.shape[0])
                                   for Xi in training_sets]
                return self

            def predict_proba(self, X):
                logprobs = np.array([model.score_samples(X)
                                     for model in self.models_]).T
                result = np.exp(logprobs + self.logpriors_)
                return result / result.sum(1, keepdims=True)

            def predict(self, X):
                return self.classes_[np.argmax(self.predict_proba(X), 1)]
```

Внутреннее устройство пользовательского оценивателя

Рассмотрим этот код и обсудим основные его особенности:

```
from sklearn.base import BaseEstimator, ClassifierMixin

class KDEClassifier(BaseEstimator, ClassifierMixin):
    """Байесовская порождающая классификация на основе метода KDE

    Параметры
    ----------
    bandwidth : float
        Ширина ядра в каждом классе
    kernel : str
        Название ядра, передаваемое функции KernelDensity
    """
```

Каждый оцениватель в библиотеке Scikit-Learn представляет собой класс, наследующий класс `BaseEstimator`, а также соответствующую примесь (mixin), которые обеспечивают стандартную функциональность. Например, помимо прочего, класс `BaseEstimator` включает логику, необходимую для клонирования/копирования оценивателя, чтобы использовать его в процедуре перекрестной проверки, а `ClassifierMixin` определяет используемый по умолчанию метод `score()`. Мы также задали docstring, который будет собран справочной системой языка Python (см. раздел «Справка и документация в оболочке Python» главы 1).

Вот метод инициализации нашего класса:

```
    def __init__(self, bandwidth=1.0, kernel='gaussian'):
        self.bandwidth = bandwidth
        self.kernel = kernel
```

Это тот код, который фактически выполняется при создании объекта посредством конструктора `KDEClassifier()`. В библиотеке Scikit-Learn важно, чтобы в методе инициализации *не содержалось никаких команд, кроме* присваивания объекту `self` переданных значений по имени. Причина в том, что содержащаяся в классе `BaseEstimator` логика необходима для клонирования и модификации оценивателей для перекрестной проверки, поиска по сетке и других целей. Аналогично все аргументы метода `__init__` должны быть объявлены явным образом, то есть следует избегать аргументов `*args` или `**kwargs`, так как они не могут быть корректно обработаны внутри процедур перекрестной проверки.

Дальше идет метод `fit()`, в котором мы обрабатываем обучающие данные:

```
    def fit(self, X, y):
        self.classes_ = np.sort(np.unique(y))
        training_sets = [X[y == yi] for yi in self.classes_]
        self.models_ = [KernelDensity(bandwidth=self.bandwidth,
                                      kernel=self.kernel).fit(Xi)
                        for Xi in training_sets]
```

```
        self.logpriors_ = [np.log(Xi.shape[0] / X.shape[0])
                           for Xi in training_sets]
        return self
```

В нем мы находим в обучающих данных уникальные классы, обучаем модель `KernelDensity` для всех классов и вычисляем априорные вероятности на основе количеств исходных выборок. Наконец, метод `fit()` должен всегда возвращать объект `self`, чтобы можно было связывать команды в цепочку. Например:

```
label = model.fit(X, y).predict(X)
```

Обратите внимание, что все сохраняемые результаты обучения сохраняются с подчеркиванием в конце названия (например, `self.logpriors_`). Такие условные обозначения используются в библиотеке Scikit-Learn, чтобы можно было быстро просмотреть список членов оценивателя (с помощью TAB-автодополнения оболочки IPython) и выяснить, какие именно члены были обучены на обучающих данных.

Наконец, у нас имеется логика для предсказания меток новых данных:

```
    def predict_proba(self, X):
        logprobs = np.vstack([model.score_samples(X)
                              for model in self.models_]).T
        result = np.exp(logprobs + self.logpriors_)
        return result / result.sum(1, keepdims=True)

    def predict(self, X):
        return self.classes_[np.argmax(self.predict_proba(X), 1)]
```

Поскольку мы имеем дело с вероятностным классификатором, мы сначала реализовали метод `predict_proba()`, возвращающий массив формы `[n_samples, n_classes]` вероятностей классов. Элемент `[i, j]` этого массива представляет собой апостериорную вероятность того, что выборка `i` — член класса `j`, вычисленная путем умножения функции правдоподобия на априорную вероятность и нормализации.

Наконец, эти вероятности используются в методе `predict()`, который возвращает класс с максимальной вероятностью.

Использование пользовательского оценивателя

Воспользуемся этим пользовательским оценивателем для решения задачи классификации рукописных цифр. Мы загрузим цифры и вычислим оценку эффективности модели для диапазона вариантов ширины ядра с помощью метаоценивателя `GridSearchCV` (см. более подробную информацию по этому вопросу в разделе «Гиперпараметры и проверка модели» данной главы):

```
In[17]: from sklearn.datasets import load_digits
        from sklearn.grid_search import GridSearchCV

        digits = load_digits()
```

```
        bandwidths = 10 ** np.linspace(0, 2, 100)
        grid = GridSearchCV(KDEClassifier(), {'bandwidth': bandwidths})
        grid.fit(digits.data, digits.target)
        scores = [val.mean_validation_score for val in grid.grid_scores_]
```

Далее можно построить график полученной при перекрестной проверке оценки эффективности модели как функции от ширины ядра (рис. 5.148):

```
In[18]: plt.semilogx(bandwidths, scores)
        plt.xlabel('bandwidth')            # Ширина ядра
        plt.ylabel('accuracy')             # Точность
        plt.title('KDE Model Performance') # Эффективность модели KDE

        print(grid.best_params_)
        print('accuracy =', grid.best_score_)
```

```
{'bandwidth': 7.0548023107186433}
accuracy = 0.966611018364
```

Как видим, этот «не столь наивный» байесовский классификатор достигает точности перекрестной проверки в более чем 96%. И это по сравнению с примерно 80% у «наивного» байесовского классификатора:

```
In[19]: from sklearn.naive_bayes import GaussianNB
        from sklearn.cross_validation import cross_val_score
        cross_val_score(GaussianNB(), digits.data, digits.target).mean()
```

```
Out[19]: 0.81860038035501381
```

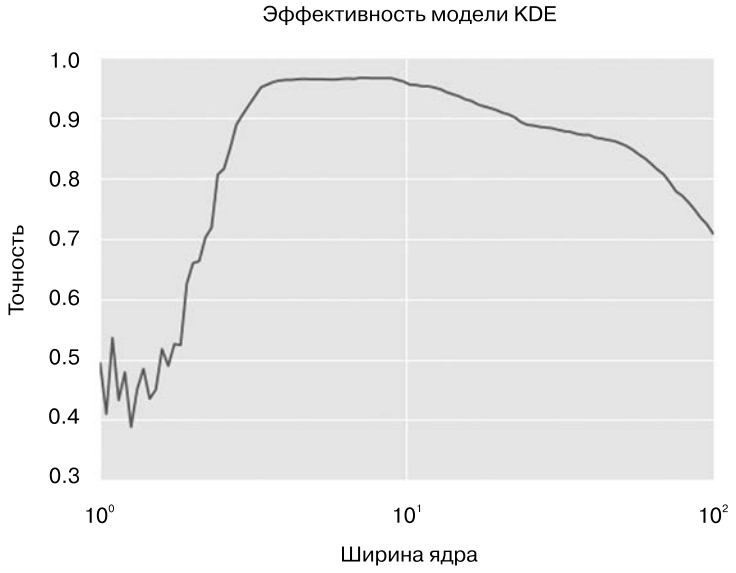

Рис. 5.148. Кривая проверки для основанного на KDE байесовского классификатора

Одно из преимуществ подобного порождающего классификатора — удобство интерпретации результатов: мы получаем для каждой неизвестной выборки не только вероятностную классификацию, но и *полную модель* распределения точек, с которыми мы ее сравниваем! При необходимости это позволяет пролить свет на причины того, почему конкретная классификация именно такова, причины, которые такие алгоритмы, как SVM и случайные леса, скрывают.

Чтобы достичь еще большего, можно внести в нашу модель классификатора KDE некоторые усовершенствования:

- допустить независимое изменение ширины ядра для каждого класса;
- оптимизировать ширину ядер не на основе оценки точности предсказания, а на основе функции правдоподобия для обучающих данных при порождающей модели для каждого класса, то есть использовать оценки эффективности непосредственно из функции `KernelDensity`, а не общую оценку точности предсказания.

И наконец, если вы хотите приобрести опыт создания своих собственных оценивателей, можете попробовать создать аналогичный байесовский классификатор с использованием смесей Гауссовых распределений вместо KDE.

Прикладная задача: конвейер распознавания лиц

В этой главе мы рассмотрели несколько основных идей и алгоритмов машинного обучения. Но перейти от теоретических идей к настоящим прикладным задачам может оказаться непростым делом. Реальные наборы данных часто бывают зашумлены и неоднородны, в них могут отсутствовать признаки, они могут содержать данные в таком виде, который сложно преобразовать в аккуратную матрицу `[n_samples, n_features]`. Вам придется, прежде чем воспользоваться любым из изложенных здесь методов, сначала извлечь эти признаки из данных. Не существует готового единого шаблона, подходящего для всех предметных областей. В этом вопросе вам как исследователю данных придется использовать ваши собственные интуицию и накопленный опыт.

Одно из очень интересных приложений машинного обучения — анализ изображений, и мы уже видели несколько примеров его с использованием пиксельных признаков для классификации. На практике данные редко оказываются настолько однородными, и простых пикселов будет недостаточно. Это привело к появлению обширной литературы, посвященной методам *выделения признаков* (feature extraction) для изображений (см. раздел «Проектирование признаков» данной главы).

В этом разделе мы рассмотрим одну из подобных методик выделения признаков, гистограмму направленных градиентов (histogram of oriented gradients, HOG, см. https://ru.wikipedia.org/wiki/Гистограмма_направленных_градиентов), которая преобразует пикселы изображения в векторное представление, чувствительное к несущим

информацию признакам изображения, без учета таких факторов, как освещенность. Мы воспользуемся этими признаками для разработки простого конвейера распознавания лиц, используя алгоритмы и идеи машинного обучения, которые мы обсуждали ранее в этой главе. Начнем с обычных импортов:

```
In[1]: %matplotlib inline
       import matplotlib.pyplot as plt
       import seaborn as sns; sns.set()
       import numpy as np
```

Признаки в методе HOG

Гистограмма направленных градиентов — простая процедура выделения признаков, разработанная для идентификации пешеходов на изображениях. Метод HOG включает следующие этапы.

1. Необязательная предварительная нормализация изображений. В результате получаются признаки, слабо зависящие от изменений освещенности.
2. Операция свертывания изображения с помощью двух фильтров, чувствительных к горизонтальным и вертикальным градиентам яркости. Это позволяет уловить информацию о границах, контурах и текстурах изображения.
3. Разбивка изображения на ячейки заранее определенного размера и вычисление гистограммы направлений градиентов в каждой из ячеек.
4. Нормализация гистограмм в каждой из ячеек путем сравнения с несколькими близлежащими ячейками. Это еще больше подавляет влияние освещенности на изображение.
5. Формирование одномерного вектора признаков из информации по каждой ячейке.

В проект Scikit-Image[1] встроена процедура выделения признаков на основе HOG, которую мы сможем достаточно быстро применить на практике и визуализировать направленные градиенты во всех ячейках (рис. 5.149):

```
In[2]: from skimage import data, color, feature
       import skimage.data

       image = color.rgb2gray(data.chelsea())
       hog_vec, hog_vis = feature.hog(image, visualise=True)

       fig, ax = plt.subplots(1, 2, figsize=(12, 6),
                              subplot_kw=dict(xticks=[], yticks=[]))
       ax[0].imshow(image, cmap='gray')
       ax[0].set_title('input image')
```

[1] Для установки его в вашей системе выполните следующую команду:
conda install scikit-image

```
ax[1].imshow(hog_vis)
ax[1].set_title('visualization of HOG features');
```

Рис. 5.149. Визуализация HOG-признаков, вычисленных для изображения

Метод HOG в действии: простой детектор лиц

На основе этих признаков HOG можно создать простой алгоритм обнаружения лиц с помощью любого из оценивателей библиотеки Scikit-Learn. Мы воспользуемся линейным методом опорных векторов (см. раздел «Заглянем глубже: метод опорных векторов» данной главы). Алгоритм включает следующие шаги.

1. Получение миниатюр изображений, на которых представлены лица, для формирования набора «положительных» обучающих выборок.
2. Получение миниатюр изображений, на которых не представлены лица для формирования набора «отрицательных» обучающих выборок.
3. Выделение HOG-признаков из этих обучающих выборок.
4. Обучение линейного SVM-классификатора на этих выборках.
5. В случае «незнакомого» изображения перемещаем по изображению скользящее окно, применяя нашу модель для определения того, содержится ли в этом окне лицо или нет.
6. Если обнаруженные лица частично пересекаются, объединяем их в одно окно.

Пройдемся по этим шагам подробнее.

1. Получаем набор положительных обучающих выборок.

   Найдем положительные обучающие выборки с разнообразными лицами. У нас есть уже подходящий набор данных Labeled Faces in the Wild (LFW), который можно скачать с помощью библиотеки Scikit-Learn:

```
In[3]: from sklearn.datasets import fetch_lfw_people
       faces = fetch_lfw_people()
       positive_patches = faces.images
```

```
positive_patches.shape
```

```
Out[3]: (13233, 62, 47)
```

Мы получили пригодную для обучения выборку из 13 000 изображений лиц.

2. Получаем набор отрицательных обучающих выборок.

Далее нам необходимо найти набор миниатюр такого же размера, на которых *не* изображены лица. Чтобы сделать это, можно, например, взять любой корпус исходных изображений и извлечь из них миниатюры в различных масштабах. Воспользуемся некоторыми из поставляемых вместе с пакетом Scikit-Image изображений, а также классом `PatchExtractor` библиотеки Scikit-Learn:

```
In[4]: from skimage import data, transform

       imgs_to_use = ['camera', 'text', 'coins', 'moon',
                      'page', 'clock', 'immunohistochemistry',
                      'chelsea', 'coffee', 'hubble_deep_field']
       images = [color.rgb2gray(getattr(data, name)())
                 for name in imgs_to_use]
```

```
In[5]:
from sklearn.feature_extraction.image import PatchExtractor

def extract_patches(img, N, scale=1.0,
                    patch_size=positive_patches[0].shape):
    extracted_patch_size = \
    tuple((scale * np.array(patch_size)).astype(int))
    extractor = PatchExtractor(patch_size=extracted_patch_size,
                               max_patches=N, random_state=0)
    patches = extractor.transform(img[np.newaxis])
    if scale != 1:
        patches = np.array([transform.resize(patch, patch_size)
                            for patch in patches])
    return patches

negative_patches = np.vstack([extract_patches(im, 1000, scale)
                      for im in images for scale in [0.5, 1.0, 2.0]])
negative_patches.shape
```

```
Out[5]: (30000, 62, 47)
```

У нас теперь есть 30 000 подходящих фрагментов изображений, не содержащих лиц. Рассмотрим некоторые из них, чтобы лучше представить, как они выглядят (рис. 5.150):

```
In[6]: fig, ax = plt.subplots(6, 10)
       for i, axi in enumerate(ax.flat):
           axi.imshow(negative_patches[500 * i], cmap='gray')
           axi.axis('off')
```

Рис. 5.150. Отрицательные фрагменты изображений, не содержащие лиц

Надеемся, что они достаточно хорошо охватывают пространство «не лиц», которые могут встретиться нашему алгоритму.

3. Объединяем наборы и выделяем HOG-признаки.

   При наличии положительных и отрицательных выборок мы можем их объединить и вычислить HOG-признаки. Этот шаг займет некоторое время, поскольку признаки HOG требуют непростых вычислений для каждого изображения.

```
In[7]: from itertools import chain
       X_train = np.array([feature.hog(im)
                           for im in chain(positive_patches,
                                           negative_patches)])
       y_train = np.zeros(X_train.shape[0])
       y_train[:positive_patches.shape[0]] = 1

In[8]: X_train.shape

Out[8]: (43233, 1215)
```

   Итак, мы получили 43 000 обучающих выборок в 1215-мерном пространстве и наши данные находятся в подходящем для библиотеки Scikit-Learn виде!

4. Обучаем метод опорных векторов.

   Воспользуемся изученными ранее в данной главе инструментами для создания классификатора фрагментов миниатюр. Линейный метод опорных векторов — хороший выбор для задачи бинарной классификации в случае столь высокой размерности. Воспользуемся классификатором `LinearSVC`, поскольку он обычно лучше масштабируется при росте числа выборок по сравнению с `SVC`.

Но сначала воспользуемся простым Гауссовым наивным байесовским классификатором, чтобы было с чем сравнивать:

```
In[9]: from sklearn.naive_bayes import GaussianNB
       from sklearn.cross_validation import cross_val_score
       cross_val_score(GaussianNB(), X_train, y_train)

Out[9]: array([ 0.9408785 ,  0.8752342 ,  0.93976823])
```

Как видим, на наших данных даже наивный байесовский алгоритм достигает более чем 90%-ной точности. Попробуем теперь метод опорных векторов с поиском по сетке из нескольких вариантов параметра `C`:

```
In[10]: from sklearn.svm import LinearSVC
        from sklearn.grid_search import GridSearchCV
        grid = GridSearchCV(LinearSVC(), {'C': [1.0, 2.0, 4.0, 8.0]})
        grid.fit(X_train, y_train)
        grid.best_score_

Out[10]: 0.98667684407744083

In[11]: grid.best_params_

Out[11]: {'C': 4.0}
```

Обучим этот оптимальный оцениватель на полном наборе данных:

```
In[12]: model = grid.best_estimator_
        model.fit(X_train, y_train)

Out[12]: LinearSVC(C=4.0, class_weight=None, dual=True,
              fit_intercept=True, intercept_scaling=1,
              loss='squared_hinge', max_iter=1000,
              multi_class='ovr', penalty='l2',
              random_state=None, tol=0.0001, verbose=0)
```

5. Выполняем поиск лиц в новом изображении.

   Теперь, когда у нас есть модель, возьмем новое изображение и посмотрим, насколько хорошо она в нем себя покажет. Воспользуемся для простоты одним из изображений астронавтов (см. обсуждение этого вопроса в разделе «Предостережения и дальнейшие усовершенствования» этой главы), перемещая по нему скользящее окно и оценивая каждый фрагмент (рис. 5.151):

```
In[13]: test_image = skimage.data.astronaut()
        test_image = skimage.color.rgb2gray(test_image)
        test_image = skimage.transform.rescale(test_image, 0.5)
        test_image = test_image[:160, 40:180]

        plt.imshow(test_image, cmap='gray')
        plt.axis('off');
```

Рис. 5.151. Изображение, в котором мы попытаемся найти лицо

Далее создадим окно, которое будет перемещаться по фрагментам этого изображения с вычислением HOG-признаков для каждого фрагмента:

```
In[14]: def sliding_window(img, patch_size=positive_patches[0].shape,
                           istep=2, jstep=2, scale=1.0):
            Ni, Nj = (int(scale * s) for s in patch_size)
            for i in range(0, img.shape[0] - Ni, istep):
                for j in range(0, img.shape[1] - Ni, jstep):
                    patch = img[i:i + Ni, j:j + Nj]
                    if scale != 1:
                        patch = transform.resize(patch, patch_size)
                    yield (i, j), patch

        indices, patches = zip(*sliding_window(test_image))
        patches_hog = np.array([feature.hog(patch) for patch in patches])
        patches_hog.shape

Out[14]: (1911, 1215)
```

Наконец, возьмем эти фрагменты, для которых вычислены признаки HOG, и воспользуемся нашей моделью, чтобы определить, содержат ли какие-то из них лица:

```
In[15]: labels = model.predict(patches_hog)
        labels.sum()
Out[15]: 33.0
```

Таким образом, среди 2000 фрагментов найдено 33 лица. Воспользуемся имеющейся о фрагментах информацией, чтобы определить, где в нашем контрольном изображении они располагаются, нарисовав их границы в виде прямоугольников (рис. 5.152):

```
In[16]: fig, ax = plt.subplots()
        ax.imshow(test_image, cmap='gray')
```

```
ax.axis('off')

Ni, Nj = positive_patches[0].shape
indices = np.array(indices)

for i, j in indices[labels == 1]:
    ax.add_patch(plt.Rectangle((j, i), Nj, Ni, edgecolor='red',
                               alpha=0.3, lw=2,
                               facecolor='none'))
```

Рис. 5.152. Окна, в которых были обнаружены лица

Все обнаруженные фрагменты перекрываются и содержат имеющееся на изображении лицо! Отличный результат для всего нескольких строк кода на языке Python.

Предостережения и дальнейшие усовершенствования

Если посмотреть на предшествующий код и примеры немного внимательнее, можно обнаружить, что нужно сделать еще немало, прежде чем можно будет назвать наше приложение распознания лиц готовым к промышленной эксплуатации. В нашем коде имеется несколько проблемных мест. Кроме того, в него не помешает внести несколько усовершенствований.

- *Наша обучающая последовательность, особенно в части отрицательных признаков, неполна*. Основная проблема заключается в том, что существует множество напоминающих лица текстур, не включенных в нашу обучающую последовательность, поэтому нынешняя модель будет склонна выдавать ложноположительные результаты. Это будет заметно, если попытаться выполнить предыдущий алгоритм для *полного* изображения астронавта: текущая модель приведет к множеству ложных обнаружений лиц в других областях изображения.

Можно было бы попытаться решить эту проблему путем добавления в отрицательную обучающую последовательность множества разнообразных изображений, и это, вероятно, действительно привело бы к некоторому улучшению ситуации. Другой способ — использование узконаправленного подхода, например, *hard negative mining*. При подходе hard negative mining, берется новый, еще не виденный классификатором набор изображений и все фрагменты в нем, соответствующие ложноположительным результатам, явным образом добавляются в качестве отрицательных примеров в обучающую последовательность до повторного обучения классификатора.

- *Текущий конвейер выполняет поиск только при одном значении масштаба.* В текущем виде наш алгоритм будет распознавать только те лица, чей размер примерно равен 62 × 47 пикселов. Эту проблему можно решить довольно просто путем применения скользящих окон различных размеров и изменения размера каждого из фрагментов с помощью функции `skimage.transform.resize` до подачи его на вход модели. На самом деле используемая здесь вспомогательная функция `sliding_window()` уже учитывает этот нюанс.
- *Желательно комбинировать перекрывающиеся фрагменты, на которых обнаружены лица.* В случае готового к промышленной эксплуатации конвейера получение 30 обнаружений одного и того же лица представляется нежелательным. Хотелось бы сократить перекрывающиеся группы обнаруженных лиц до одного. Это можно сделать с помощью одного из методов кластеризации без учителя (хороший кандидат на эту роль — кластеризация путем сдвига среднего значения (meanshift clustering)) или посредством процедурного подхода, например алгоритма *подавления немаксимумов* (nonmaximum suppression), часто используемого в сфере машинного зрения.
- *Конвейер должен быть более продвинутым.* После решение вышеописанных проблем неплохо было бы создать более продвинутый конвейер, который бы получал на входе обучающие изображения и выдавал предсказания на основе скользящих окон. Именно в этом вопросе язык Python как инструмент науки о данных демонстрирует все свои возможности: приложив немного труда, мы сможем скомпоновать наш предварительный код с качественно спроектированным объектно-ориентированным API, обеспечивающим для пользователя легкость в использовании. Оставлю это в качестве упражнения читателю.
- *Желательно обдумать возможность применения более современных средств предварительной обработки, таких как глубокое обучение.* Наконец, мне хотелось бы добавить, что HOG и другие процедурные методы выделения признаков для изображений более не считаются современными. Вместо них многие современные конвейеры обнаружения объектов используют различные варианты глубоких нейронных сетей. Нейронные сети можно рассматривать как оцениватель, определяющий оптимальную стратегию выделения признаков на основе самих данных, а не полагающийся на интуицию пользователя. Знакомство с методами глубоких нейронных сетей выходит за рамки этого раздела концептуально (и вычислительно!), хотя некоторые инструменты с открытым

исходным кодом, такие как TensorFlow (https://www.tensorflow.org/), выпущенный корпорацией Google, сделали в последнее время подход глубокого обучения значительно более доступным. На момент написания книги глубокое обучение в языке Python остается еще довольно «незрелой» концепцией, поэтому я не могу рекомендовать вам какие-либо авторитетные источники информации по этому вопросу. Тем не менее список литературы в следующем разделе покажет вам, с чего можно начать.

Дополнительные источники информации по машинному обучению

В этой главе мы кратко рассмотрели машинное обучение в языке Python, в основном используя инструменты из библиотеки Scikit-Learn. Как бы объемна ни была эта глава, в ней все равно невозможно было охватить многие интересные и важные алгоритмы, подходы и вопросы. Я хотел бы предложить тем, кто желает узнать больше о машинном обучении, некоторые дополнительные источники информации.

Машинное обучение в языке Python

Если вы хотите узнать больше о машинном обучении, обратите внимание на следующие источники информации.

- *Сайт библиотеки Scikit-Learn*. На сайте библиотеки Scikit-Learn содержатся поразительные объемы документации и примеров, охватывающие не только некоторые из рассмотренных в книге моделей, но и многое другое. Если вам необходим краткий обзор наиболее важных и часто используемых алгоритмов машинного обучения, этот сайт будет для вас отличной отправной точкой.
- *Обучающие видео с таких конференций, как SciPy, PyCon и PyData*. Библиотека Scikit-Learn и другие вопросы машинного обучения — неизменные фавориты учебных пособий ежегодных конференций, посвященных языку Python, в частности PyCon, SciPy и PyData. Найти наиболее свежие материалы можно путем поиска в Интернете.
- *Книга Introduction to Machine Learning with Python* (*«Введение в машинное обучение с помощью Python»*, http://shop.oreilly.com/product/0636920030515.do[1]). Написанная Андреасом Мюллером и Сарой Гвидо книга полностью освещает изложенные в данной главе вопросы. Если вы хотели бы детально разобраться в важнейших вопросах машинного обучения и узнать, как использовать набор инструментов библиотеки Scikit-Learn на все 100 %, эта книга — отличный источник информации, написанный одним из разработчиков команды Scikit-Learn.

[1] http://www.williamspublishing.com/Books/978-5-9908910-8-1.html.

- *Книга Python Machine Learning* (*«Python и машинное обучение»,* https://www.packtpub.com/big-data-and-business-intelligence/python-machine-learning[1]). Книга Себастьяна Рашки в меньшей степени акцентирует внимание на самой библиотеке Scikit-Learn, и в большей — на диапазоне имеющихся в языке Python инструментов машинного обучения. В ней приведено очень полезное обсуждение масштабирования основанных на языке Python подходов машинного обучения на большие и сложные наборы данных.

Машинное обучение в целом

Машинное обучение не ограничивается только миром языка Python. Существует множество отличных источников информации, с помощью которых вы сможете расширить свои познания в этом вопросе. Я отмечу здесь несколько, на мой взгляд, наиболее полезных.

- *Машинное обучение* (https://www.coursera.org/learn/machine-learning). Этот бесплатный онлайн-курс, преподаваемый Эндрю Энгом из проекта Coursera, представляет собой исключительно ясно изложенный материал по основам машинного обучения с алгоритмической точки зрения. Он предполагает знания математики и программирования на уровне старших курсов университета и последовательно и подробно обсуждает некоторые из наиболее важных алгоритмов машинного обучения. Алгоритмически ранжированные домашние задания позволят вам реализовать некоторые из этих моделей самостоятельно.
- *Книга Pattern Recognition and Machine Learning* (*«Распознавание образов и машинное обучение»,* https://www.springer.com/us/book/9780387310732). Написанная Кристофером Бишопом, эта классическая книга предназначена для специалистов. Она охватывает во всех подробностях рассмотренные в данной главе понятия машинного обучения. Если вы хотите продвинуться в вопросе дальше, вам не обойтись без этой книги на полке.
- *Machine Learning: A Probabilistic Perspective* («Машинное обучение: вероятностная точка зрения», https://mitpress.mit.edu/books/machine-learning-0). В пособии уровня выпускников университета, написанном Кевином Мерфи, исследуются практически все важные алгоритмы машинного обучения с единой вероятностной точки зрения.

Подход в этих источниках информации более формализован, чем представленный в данной книге материал, но подлинное понимание основ представленных методов требует некоторого углубления в математический аппарат. Если вы готовы попробовать свои силы и поднять свои знания науки о данных на новый уровень, ныряйте в них немедля!

[1] http://dmkpress.com/catalog/computer/data/978-5-97060-409-0/.

Дж. Вандер Плас

Python для сложных задач: наука о данных и машинное обучение

Перевели с английского *И. Пальти*

Изготовлено в России. Изготовитель: ООО «Питер Пресс».
Место нахождения и фактический адрес: 192102, Россия, город Санкт-Петербург,
улица Андреевская, дом 3, литер А, помещение 7Н. Тел.: +78127037373.

Дата изготовления: 09.2017. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Подписано в печать 23.08.17. Формат 70×100/16. Бумага офсетная. Усл. п. л. 46,440. Тираж 1200. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87