O'REILLY®

Машинное обучение с использованием Python. Сборник рецептов

Практические решения от предобработки до глубокого обучения

Крис Элбон

bhv®

Machine Learning with Python Cookbook

Practical Solutions from Preprocessing to Deep Learning

Chris Albon

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY

Крис Элбон

Машинное обучение с использованием Python. Сборник рецептов

Санкт-Петербург
«БХВ-Петербург»
2019

УДК 004.8+004.438Python
ББК 32.973.26-018.1
Э45

Элбон Крис

Э45 Машинное обучение с использованием Python. Сборник рецептов: Пер. с англ. — СПб.: БХВ-Петербург, 2019. — 384 с.: ил.

ISBN 978-5-9775-4056-8

Книга содержит около 200 рецептов решения практических задач машинного обучения, таких как загрузка и обработка текстовых или числовых данных, отбор модели, уменьшение размерности и многие другие. Рассмотрена работа с языком Python и его библиотеками, в том числе pandas и scikit-learn. Решения всех задач сопровождаются подробными объяснениями. Каждый рецепт содержит работающий программный код, который можно вставлять, объединять и адаптировать, создавая собственное приложение.

Приведены рецепты решений с использованием: векторов, матриц и массивов; обработки данных, текста, изображений, дат и времени; уменьшения размерности и методов выделения или отбора признаков; оценивания и отбора моделей; линейной и логистической регрессии, деревьев, лесов и *k* ближайших соседей; опорно-векторных машин (SVM), наивных байесовых классификаторов, кластеризации и нейронных сетей; сохранения и загрузки натренированных моделей.

Для разработчиков систем машинного обучения

УДК 004.8+004.438Python
ББК 32.973.26-018.1

Группа подготовки издания:

Руководитель проекта	*Евгений Рыбаков*
Зав. редакцией	*Екатерина Сависте*
Перевод с английского	*Андрея Логунова*
Компьютерная верстка	*Ольги Сергиенко*
Оформление обложки	*Карины Соловьевой*

Подписано в печать 02.07.19.
Формат 70×100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 30,96.
Тираж 1500 экз. Заказ № 9428.
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Отпечатано с готового оригинал-макета
ООО "Принт-М", 142300, М.О., г. Чехов, ул. Полиграфистов, д. 1

ISBN 978-1-491-98938-8 (англ.)
ISBN 978-5-9775-4056-8 (рус.)

Оглавление

Об авторе

Крис Альбон (Chris Albon) — аналитик данных и политолог с десятилетним опытом применения статистического обучения, искусственного интеллекта и разработки программного обеспечения для политических, социальных и гуманитарных проектов — от мониторинга выборов до оказания помощи в случае стихийных бедствий. В настоящее время Крис является ведущим аналитиком данных в BRCK — кенийском стартапе, создающем прочную сеть для пользователей Интернета на формирующемся рынке.

Предисловие

За последние несколько лет машинное (само)обучение стало частью широкого спектра повседневных, некоммерческих и правительственных операций. По мере роста популярности машинного обучения развивалась мелкосерийная индустрия высококачественной литературы, которая преподносила прикладное машинное обучение практикующим специалистам. Эта литература была очень успешной в подготовке целого поколения аналитиков данных и инженеров машинного обучения. Кроме того, в этой литературе рассматривалась тема машинного обучения с точки зрения предоставления учебного ресурса, демонстрировавшего специалисту, что такое машинное обучение и как оно работает. Вместе с тем, хотя этот подход и был плодотворным, он упустил из виду другую точку зрения на эту тему: как на материальную часть, гайки и болты, повседневного машинного обучения. В этом и состоит мотивация данной книги — предоставить читателям не фолиант по машинному обучению, а гаечный ключ для профессионала, чтобы книга лежала с зачитанными до дыр страницами на рабочих столах, помогла решать оперативные повседневные задачи практикующего специалиста по машинному обучению.

Если говорить точнее, то в книге используется задачно-ориентированный подход к машинному обучению, с почти 200 самодостаточными решениями (программный код можно скопировать и вставить, и он будет работать) наиболее распространенных задач, с которыми столкнется аналитик данных или инженер по машинному обучению, занимающийся созданием моделей.

Конечная цель книги — быть справочником для специалистов, строящих реальные машинно-обучающиеся системы. Например, представьте, что читатель имеет файл JSON, содержащий 1000 категориальных и числовых признаков с пропущенными данными и векторами категориальных целей с несбалансированными классами, и хочет получить интерпретируемую модель. Мотивация для этой книги состоит в предоставлении рецептов, чтобы помочь читателю освоить такие процессы, как:

- загрузка файла JSON (см. рецепт 2.5);
- стандартизация признаков (см. рецепт 4.2);
- кодирование словарей признаков (см. рецепт 5.3);
- импутация пропущенных значений классов (см. рецепт 5.4);
- уменьшение количества признаков с помощью главных компонент (см. рецепт 9.1);
- отбор наилучшей модели с помощью рандомизированного поиска (см. рецепт 12.2);

- тренировка классификатора на основе случайного леса (см. рецепт 14.4);
- отбор случайных признаков в случайных лесах (см. рецепт 14.7).

Читатель имеет возможность:

1. Копировать/вставлять программный код с полной уверенностью, что он действительно работает с включенным игрушечным набором данных.
2. Прочитать обсуждение, чтобы получить представление о теории, лежащей в основе метода, который этот программный код исполняет, и узнать, какие параметры важно учитывать.
3. Вставлять/комбинировать/адаптировать программный код из рецептов для конструирования фактического приложения.

Для кого предназначена книга

Данная книга не является введением в машинное (само)обучение. Если вы не чувствуете себя уверенно в области основных понятий машинного обучения либо никогда не проводили время за изучением машинного обучения, то не покупайте эту книгу. Она предназначена для практикующих специалистов машинного обучения, которые, чувствуя себя комфортно с теорией и понятиями машинного обучения, извлекут пользу из краткого справочника, содержащего программный код для решения задач, с которыми они сталкиваются, работая ежедневно с машинным обучением.

Предполагается также, что читатель уверен в своих знаниях языка программирования Python и управления его пакетами.

Для кого не предназначена книга

Как заявлено ранее, эта книга не является введением в машинное (само)обучение. Данная книга не должна быть вашим первым изданием по этой теме. Если вы не знакомы с такими понятиями, как перекрестная проверка, случайный лес и градиентный спуск, то вы, вероятно, не извлечете из этой книги такой же пользы, которую можно получить от одного из многих высококачественных текстов, специально предназначенных для ознакомления с этой темой. Я рекомендую прочитать одну из таких книг, а затем вернуться к этой книге, чтобы узнать рабочие, практические решения для задач машинного обучения.

Терминология, используемая в книге

Машинное (само)обучение опирается на методы из широкого спектра областей, включая информатику, статистику и математику. По этой причине при обсуждении машинного обучения существуют значительные расхождения в используемой терминологии.

Наблюдение — единое целое в нашем уровне исследования, например человек, сделка купли-продажи или запись.

Обучающийся алгоритм — алгоритм, используемый для того, чтобы обучиться наилучшим параметрам модели, например линейной регрессии, наивного байесовского классификатора или деревьев решений.

Модели — результат тренировки обучающегося алгоритма. Обучающиеся алгоритмы заучивают модели, которые мы затем используем для предсказания.

Параметры — веса или коэффициенты модели, заученные в ходе тренировки.

Гиперпараметры — настройки обучающегося алгоритма, которые необходимо отрегулировать перед тренировкой.

Результативность — метрический показатель, используемый для оценивания качества модели.

Потеря — метрический показатель, который максимизируется или минимизируется посредством тренировки.

Тренировка — применение обучающегося алгоритма к данным с использованием численных подходов, таких как градиентный спуск.

Подгонка — применение обучающегося алгоритма к данным с использованием аналитических подходов.

Данные — коллекция наблюдений.

Признательности

Эта книга была бы невозможна без любезной помощи множества друзей и незнакомых мне людей. Перечислить всех, кто протянул руку помощи в реализации этого проекта, будет невозможно, но я хотел бы хотя бы упомянуть Анжелу Басса, Терезу Борсух, Джастина Бозонье, Андре де Бруина, Нума Дхамани, Дэна Фридмана, Джоэла Груса, Сару Гвидо, Билла Камбороглу, Мэта Келси, Лиззи Кумар, Хилари Паркер, Нити Подьял, Себастьяна Рашку и Шрея Шанкар.

Я должен им всем по кружке пива или дать пять.

Комментарии переводчика

В центре внимания машинного обучения и его подобласти, глубокого обучения, находится обучающаяся система, то есть система, способная с течением времени приобретать новые знания и улучшать свою работу, используя поступающую информацию[1]. В зарубежной специализированной литературе для *передачи* знаний и *приобретения* знаний существуют отдельные термины — train (*натренировать*, обучить) и learn (*выучить*, обучиться).

[1] См. **https://ru.wikipedia.org/wiki/Обучающаяся_система**, а также **https://bigenc.ru/mathematics/text/1810335**. — *Прим. перев.*

Training (тренировка) — это работа, которую выполняет исследователь-проектировщик для получения обучившейся модели, в основе которой лежит обучающийся алгоритм, по сути искатель минимумов (или максимумов) для надлежащим образом сформулированных функций, а learning (самообучение, заучивание) — это работа, которую выполняет алгоритм-ученик по приобретению новых знаний или изменению и закреплению существующих знаний и поведения[2]. Когда же в русской специальной литературе используется термин "обучение", то он несет в себе двусмысленность, потому что под ним может подразумеваться и передача знаний, и получение знаний одновременно, как, например, в случае с термином "машинное обучение", который может означать и тренировку алгоритмических машин, и способность таких машин обучаться, что нередко вносит путаницу и терминологический разброд в переводной литературе при решении дилеммы "training-learning", в то время как появление в зарубежной технической литературе термина learning в любом виде подразумевает исключительно второе — *самообучение*, *заучивание* алгоритмом весов и других параметров. Отсюда вытекает одно важное следствие: английский термин machine learning обозначает *приобретение знаний алгоритмической машиной*, а следовательно, более соответствовать оригиналу будет термин *"машинное самообучение"* или *"автоматическое обучение"*. Весомым аргументом в пользу этого термина является и то, что с начала 60-х и до середины 80-х годов XX столетия у нас в ходу был похожий термин — "обучающиеся машины". Проблематика обучающихся и самопроизводящихся машин изучалась в работах А. Тьюринга "Может ли машина мыслить?" (1960, обучающиеся машины), К. Шеннона "Работы по теории информации и кибернетике" (самовоспроизводящиеся машины), Н. Винера "Кибернетика, или управление и связь в животном и машине" (1961), Н. Нильсона "Обучающиеся машины" (1974) и Я. З. Цыпкина "Основы теории обучающихся систем" (1970).

В настоящем переводе далее за основу принят зарубежный подход, который неизбежно привел к некоторой корректировке терминологии. Соответствующие области исследования переведены как *машинное самообучение* и *глубокое самообучение*. Применяемые в машинном обучении и глубоком обучении алгоритмы, модели и системы переведены как *обучающиеся*, *машинно-обучающиеся* и *глубоко обучающиеся*. То есть, акцент делается не на классификации алгоритма в соответствующей иерархии, а на его характерном свойстве. Далее методы, которые реализуются в обучающихся алгоритмах, переведены как *методы самообучения* (*ср.* методы обучения). Как известно, эти методы делятся на три широкие категории. Следуя принципу бритвы Оккама, они переведены как методы контролируемого самообучения (*ср.* обучение с учителем), методы неконтролируемого самообучения (*ср.* обучение без учителя) и методы самообучения с максимизацией вознаграждения, или подкрепления (*ср.* обучение с подкреплением). Последний термин обусловлен тем, что в его основе лежит алгоритм, который "учится максимизировать некое понятие вознаграждения", получаемого за правильно выполненное действие[3].

---

[2] См. **http://www.basicknowledge101.com/subjects/learningstyles.html#diy**. — *Прим. перев.*

[3] См. **https://en.wikipedia.org/wiki/Reinforcement_learning**. — *Прим. перев.*

Среди многих гиперпараметров, которые позволяют настроить работу обучающегося алгоритма, имеется rate of learing, который переведен как *скорость заучивания* (*ср.* темп обучения).

Исходный код

Перевод книги снабжен пояснениями и ссылками, размещенными в сносках. Вся кодовая база книги протестирована в среде Windows 10. При тестировании исходного кода за основу взят Python версии 3.6.4 (время перевода — май 2018 г.).

В книге используется ряд специализированных библиотек. В обычных условиях библиотеки Python можно скачать и установить из каталога библиотек Python `PyPi` (**https://pypi.python.org/**) при помощи менеджера пакетов `pip`. Однако следует учесть, что в ОС Windows для работы некоторых библиотек, в частности `scipy`, `scikit-learn`, требуется, чтобы в системе была установлена библиотека Numpy+MKL. Библиотека **Numpy+MKL** привязана к библиотеке Intel® Math Kernel Library и включает в свой состав необходимые динамические библиотеки (DLL) в каталоге numpy.core. Библиотеку Numpy+MKL следует скачать с хранилища whl-файлов на веб-странице Кристофа Голька из Лаборатории динамики флуоресценции Калифорнийского университета в г. Ирвайн (**http://www.lfd.uci.edu/~gohlke/pythonlibs/**) и установить при помощи менеджера пакетов `pip` как whl (соответствующая процедура установки пакетов в формате WHL описана ниже). Например, для 64-разрядной операционной системы Windows и среды Python 3.6 команда будет такой:

```
pip install numpy-1.14.2+mkl-cp36-cp36m-win_amd64.whl
```

Стоит также отметить, что эти особенности установки не относятся к ОС Linux и Mac OS X.

Протокол установки библиотек

Далее предлагается список команд локальной установки библиотек, скачанных с хранилища whl-файлов.

```
python -m pip install --upgrade pip
pip install numpy-1.14.2+mkl-cp36-cp36m-win_amd64.whl
pip install scipy-1.1.0-cp36-cp36m-win_amd64.whl
pip install scikit_learn-0.19.1-cp36-cp36m-win_amd64.whl
pip install beautifulsoup4-4.6.0-py3-none-any.whl
pip install opencv_python-3.4.1-cp36-cp36m-win_amd64.whl
```

Следующие ниже библиотеки устанавливаются стандартным образом:

```
pip install matplotlib
pip install pandas
pip install sqlalchemy
pip install nltk
```

```
pip install fancyimpute
pip install seaborn
pip install pydotplus
pip install graphviz
pip install keras
pip install pydot
pip install joblib
```

> **ПРИМЕЧАНИЕ.** В зависимости от базовой ОС, версий языка Python и версий программных библиотек устанавливаемые вами версии whl-файлов могут отличаться от приведенных выше, где показаны последние на май 2018 г. версии для 64-разрядной ОС Windows и Python 3.6.4.

Ниже перечислены адреса библиотек, которые следует скачать из хранилища и установить локально:

- numpy (**https://www.lfd.uci.edu/~gohlke/pythonlibs/#numpy**);
- scipy (**https://www.lfd.uci.edu/~gohlke/pythonlibs/#scipy**);
- scikit-learn (**https://www.lfd.uci.edu/~gohlke/pythonlibs/#scikit-learn**);
- BeautifulSoup (**https://www.lfd.uci.edu/~gohlke/pythonlibs**);
- OpenCV (**https://www.lfd.uci.edu/~gohlke/pythonlibs/#opencv**).

Установка библиотек Python из whl-файлов

Библиотеки для Python можно разрабатывать не только на чистом Python. Довольно часто библиотеки программируются на C (динамические библиотеки), и для них пишется обертка Python. Либо библиотека пишется на Python, но для оптимизации узких мест часть кода пишется на C. Такие библиотеки получаются очень быстрыми, однако библиотеки с вкраплениями кода на C программисту на Python тяжелее установить ввиду банального отсутствия соответствующих знаний либо необходимых компонентов и настроек в рабочей среде (в особенности в Windows). Для решения описанных проблем разработан специальный формат (файлы с расширением whl) для распространения библиотек, который содержит заранее скомпилированную версию библиотеки со всеми ее зависимостями. Формат WHL поддерживается всеми основными платформами (Mac OS X, Linux, Windows).

Установка производится с помощью менеджера библиотек `pip`. В отличие от обычной установки командой `pip install` *<имя_библиотеки>* вместо имени библиотеки указывается путь к whl-файлу: `pip install` *<путь_к_whl_файлу>*. Например,

```
pip install C:\temp\scipy-1.1.0-cp36-cp36m-win_amd64.whl
```

Откройте окно командной строки и при помощи команды `cd` перейдите в каталог, где размещен ваш whl-файл. Просто скопируйте туда имя вашего whl-файла. В этом случае полный путь указывать не понадобится. Например,

```
pip install scipy-1.1.0-cp36-cp36m-win_amd64.whl
```

При выборе библиотеки важно, чтобы разрядность устанавливаемой библиотеки и разрядность интерпретатора совпадали. Пользователи Windows могут брать whl-файлы с веб-сайта Кристофа Голька. Библиотеки там постоянно обновляются, и в архиве содержатся все, какие только могут понадобиться.

Блокноты Jupyter

В корневой папке размещены файлы с расширением ipynb. Это файлы блокнотов интерактивной среды программирования Jupyter (**http://jupyter.org/**). Блокноты Jupyter позволяют иметь в одном месте исходный код, результаты выполнения исходного кода, графики данных и документацию, которая поддерживает синтаксис упрощенной разметки Markdown и мощный синтаксис LaTeX.

Интерактивная среда программирования Jupyter — это зонтичный проект, который наряду с Python предназначен для выполнения в обычном веб-браузере небольших программ и фрагментов программного кода на других языках программирования, в том числе Julia, R и многих других (уже более 40 языков).

Интерактивная среда программирования Jupyter устанавливается стандартным образом при помощи менеджера пакетов `pip`.

```
pip install jupyter
```

Для того чтобы запустить интерактивную среду Jupyter, нужно в командной оболочке или окне терминала набрать и исполнить приведенную ниже команду:

```
jupyter notebook
```

Локальный сервер интерактивной среды Jupyter запустится в браузере, заданном по умолчанию (как правило, по адресу **http://localhost:8888/**).

Векторы, матрицы, массивы

Введение

Библиотека NumPy лежит в основе стека машинного самообучения на Python и позволяет эффективно работать со структурами данных, часто используемыми в машинном самообучении: векторами, матрицами и тензорами. Хотя NumPy не находится в центре внимания книги, эта библиотека будет часто появляться в последующих главах. В данной главе рассматриваются наиболее распространенные операции NumPy, с которыми мы, скорее всего, столкнемся во время работы с потоками операций машинного самообучения.

1.1. Создание вектора

Задача

Требуется создать вектор.

Решение

Использовать библиотеку NumPy для создания одномерного массива:

```
# Загрузить библиотеку
import numpy as np

# Создать вектор как строку
vector_row = np.array([1, 2, 3])

# Создать вектор как столбец
vector_column = np.array([[1],
                          [2],
                          [3]])
```

Обсуждение

Основной структурой данных NumPy является многомерный массив. Для того чтобы создать вектор, мы просто создаем одномерный массив. Как и векторы, эти массивы могут быть представлены горизонтально (т. е. как строки) или вертикально (т. е. как столбцы).

Дополнительные материалы для чтения

- "Векторы", математический ресурс Math's Fun ("Забавная математика") (**http://bit.ly/2FB5q1v**).
- "Евклидов вектор", Википедия (**http://bit.ly/2FtnRoL**).

1.2. Создание матрицы

Задача

Требуется создать матрицу.

Решение

Для создания двумерного массива использовать библиотеку NumPy:

```
# Загрузить библиотеку
import numpy as np

# Создать матрицу
matrix = np.array([[1, 2],
                   [1, 2],
                   [1, 2]])
```

Обсуждение

Для создания матрицы можно использовать двумерный массив NumPy. В нашем решении матрица содержит три строки и два столбца (столбец единиц и столбец двоек).

На самом деле NumPy имеет специальную матричную структуру данных:

```
matrix_object = np.mat([[1, 2],
                        [1, 2],
                        [1, 2]])

matrix([[1, 2],
        [1, 2],
        [1, 2]])
```

Однако матричная структура данных не рекомендуется по двум причинам. Во-первых, массивы являются де-факто стандартной структурой данных NumPy. Во-вторых, подавляющее большинство операций NumPy возвращают не матричные объекты, а массивы.

Дополнительные материалы для чтения

- "Матрица", Википедия (**http://bit.ly/2Ftnevp**).
- "Матрица", математический ресурс Wolfram MathWorld (**http://bit.ly/2Fut7IJ**).

1.3. Создание разреженной матрицы

Задача

Имеются данные с очень малым количеством ненулевых значений, которые требуется эффективно представить.

Решение

Создать разреженную матрицу:

```
# Загрузить библиотеки
import numpy as np
from scipy import sparse

# Создать матрицу
matrix = np.array([[0, 0],
                   [0, 1],
                   [3, 0]])

# Создать сжатую разреженную матрицу-строку (CSR-матрицу)
matrix_sparse = sparse.csr_matrix(matrix)
```

Обсуждение

В машинном самообучении часто возникает ситуация, когда имеется огромное количество данных; однако большинство элементов в данных являются нулями. Например, представьте матрицу, в которой столбцы — все фильмы в Netflix, строки — каждый пользователь Netflix, а значения — сколько раз пользователь смотрел конкретный фильм. Эта матрица будет иметь десятки тысяч столбцов и миллионы строк! Однако, поскольку большинство пользователей не смотрят почти все фильмы, подавляющая часть элементов матрицы будет равняться нулю.

Разреженные матрицы хранят только ненулевые элементы и исходят из того, что все другие значения будут равняться нулю, что приводит к значительной вычислительной экономии. В нашем решении мы создали массив NumPy с двумя ненулевыми значениями, а затем преобразовали его в разреженную матрицу. Если мы посмотрим на разреженную матрицу, то увидим, что в ней хранятся только ненулевые значения:

```
# Взглянуть на разреженную матрицу
print(matrix_sparse)
```

```
  (1, 1) 1
  (2, 0) 3
```

Существует несколько типов разреженных матриц. В *сжатых разреженных матрицах-строках* (compressed sparse row, CSR) элементы (1, 1) и (2, 0) представляют индексы ненулевых значений (с отсчетом от нуля), соответственно 1 и 3. Например,

элемент 1 находится во второй строке и втором столбце. Мы можем увидеть преимущество разреженных матриц, если создадим гораздо более крупную матрицу с еще большим количеством нулевых элементов, а затем сравним эту крупную матрицу с нашей исходной разреженной матрицей:

```
# Создать более крупную матрицу
matrix_large = np.array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                         [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
                         [3, 0, 0, 0, 0, 0, 0, 0, 0, 0]])

# Создать сжатую разреженную матрицу-строку (CSR-матрицу)
matrix_large_sparse = sparse.csr_matrix(matrix_large)

# Взглянуть на исходную разреженную матрицу
print(matrix_sparse)

  (1, 1) 1
  (2, 0) 3

# Взглянуть на более крупную разреженную матрицу
print(matrix_large_sparse)

  (1, 1) 1
  (2, 0) 3
```

Как мы видим, несмотря на то, что мы добавили в более крупную матрицу еще больше нулевых элементов, ее разреженное представление точно такое же, как и наша исходная разреженная матрица. То есть добавление нулевых элементов не изменило размер разреженной матрицы.

Как уже отмечалось, существует множество различных типов разреженных матриц, таких как сжатая разреженная матрица-столбец, список списков и словарь ключей. Хотя объяснение различных типов и их последствий выходит за рамки этой книги, стоит отметить, что "лучшего" типа разреженной матрицы не существует, однако между ними есть содержательные различия, и мы должны понимать, почему мы выбираем один тип и не выбираем другой.

Дополнительные материалы для чтения

- "Разреженные матрицы", документация SciPy (**http://bit.ly/2HReBZR**).
- "101 способ хранения разреженной матрицы", блог-пост (**http://bit.ly/2HS43cI**).

1.4. Выбор элементов

Задача

Требуется выбрать один или несколько элементов в векторе или матрице.

Решение

Массивы NumPy позволяют это легко сделать:

```
# Загрузить библиотеку
import numpy as np

# Создать вектор-строку
vector = np.array([1, 2, 3, 4, 5, 6])

# Создать матрицу
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Выбрать третий элемент вектора
vector[2]
```

```
3
```

```
# Выбрать вторую строку, второй столбец
matrix[1,1]
```

```
5
```

Обсуждение

Как и большинство вещей в Python, массивы NumPy имеют нулевую индексацию, т. е. индекс первого элемента равен 0, а не 1. С учетом этого NumPy предлагает широкий спектр методов для выбора (т. е. индексирования и нарезки) элементов или групп элементов в массивах:

```
# Выбрать все элементы вектора
vector[:]
```

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

```
# Выбрать все вплоть до третьего элемента включительно
vector[:3]
```

```
array([1, 2, 3])
```

```
# Выбрать все после третьего элемента
vector[3:]
```

```
array([4, 5, 6])
```

```
# Выбрать последний элемент
vector[-1]
```

```
6
```

```
# Выбрать первые две строки и все столбцы матрицы
matrix[:2,:]
```

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

```
# Выбрать все строки и второй столбец
matrix[:,1:2]
```

```
array([[2],
       [5],
       [8]])
```

1.5. Описание матрицы

Задача

Требуется описать форму, размер и размерность матрицы.

Решение

Использовать атрибуты `shape`, `size` и `ndim`:

```
# Загрузить библиотеку
import numpy as np

# Создать матрицу
matrix = np.array([[1, 2, 3, 4],
                   [5, 6, 7, 8],
                   [9, 10, 11, 12]])

# Взглянуть на количество строк и столбцов
matrix.shape
```

```
(3, 4)
```

```
# Взглянуть на количество элементов (строки * столбцы)
matrix.size
```

```
12
```

```
# Взглянуть на количество размерностей
matrix.ndim
```

```
2
```

Обсуждение

Эти операции могут показаться тривиальными (и это действительно так). Однако время от времени будет полезно проверить форму и размер массива для дальнейших вычислений и просто в качестве проверки состояния дел после некоторой операции.

1.6. Применение операций к элементам

Задача

Требуется применить некоторую функцию к нескольким элементам массива.

Решение

Использовать класс `vectorize` библиотеки NumPy:

```
# Загрузить библиотеку
import numpy as np

# Создать матрицу
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Создать функцию, которая добавляет к чему-то 100
add_100 = lambda i: i + 100

# Создать векторизованную функцию
vectorized_add_100 = np.vectorize(add_100)

# Применить функцию ко всем элементам в матрице
vectorized_add_100(matrix)

array([[101, 102, 103],
       [104, 105, 106],
       [107, 108, 109]])
```

Обсуждение

Класс NumPy `vectorize` конвертирует обычную функцию в функцию, которая может применяться ко всем элементам массива или части массива. Стоит отметить, что `vectorize` по существу представляет собой цикл `for` над элементами и не увеличивает производительность. Кроме того, массивы NumPy позволяют выполнять операции между массивами, даже если их размерности не совпадают (этот процесс называется *трансляцией*). Например, мы можем создать гораздо более простую версию нашего решения, используя трансляцию:

```
# Добавить 100 ко всем элементам
matrix + 100
```

```
array([[101, 102, 103],
       [104, 105, 106],
       [107, 108, 109]])
```

1.7. Нахождение максимального и минимального значений

Задача

Требуется найти максимальное или минимальное значение в массиве.

Решение

Использовать функции `max` и `min` библиотеки NumPy:

```
# Загрузить библиотеку
import numpy as np

# Создать матрицу
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Вернуть максимальный элемент
np.max(matrix)
```

```
9
```

```
# Вернуть минимальный элемент
np.min(matrix)
```

```
1
```

Обсуждение

Часто требуется узнать максимальное и минимальное значения в массиве или подмножестве массива. Это может быть достигнуто с помощью методов `max` и `min`. Используя параметр `axis`, можно также применить операцию вдоль определенного направления:

```
# Найти максимальный элемент в каждом столбце
np.max(matrix, axis=0)
```

```
array([7, 8, 9])
```

```
# Найти максимальный элемент в каждой строке
np.max(matrix, axis=1)

array([3, 6, 9])
```

1.8. Вычисление среднего значения, дисперсии и стандартного отклонения

Задача

Требуется вычислить некоторые описательные статистические показатели о массиве.

Решение

Использовать функции `mean`, `var` и `std` библиотеки NumPy:

```
# Загрузить библиотеку
import numpy as np

# Создать матрицу
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Вернуть среднее значение
np.mean(matrix)

5.0

# Вернуть дисперсию
np.var(matrix)

6.666666666666667

# Вернуть стандартное отклонение
np.std(matrix)

2.5819888974716112
```

Обсуждение

Так же как с функциями `max` и `min`, мы можем легко получать описательные статистические показатели о всей матрице или делать расчеты вдоль одной оси:

```
# Найти среднее значение в каждом столбце
np.mean(matrix, axis=0)

array([ 4., 5., 6.])
```

1.9. Реформирование массивов

Задача

Требуется изменить форму (количество строк и столбцов) массива без изменения значений элементов.

Решение

Использовать метод `reshape` библиотеки NumPy:

```
# Загрузить библиотеку
import numpy as np

# Создать матрицу 4x3
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9],
                   [10, 11, 12]])

# Реформировать матрицу в матрицу 2x6
matrix.reshape(2, 6)
```

```
array([[ 1, 2, 3,  4,  5,  6],
       [ 7, 8, 9, 10, 11, 12]])
```

Обсуждение

Метод `reshape` позволяет реструктурировать массив так, что мы сохраняем те же самые данные и при этом организуем их как другое количество строк и столбцов. Единственное требование состоит в том, чтобы формы исходной и новой матриц содержали одинаковое количество элементов (т. е. матрицы имели одинаковый размер). Размер матрицы можно увидеть с помощью атрибута `size`:

```
matrix.size
```

```
12
```

Одним из полезных аргументов в методе `reshape` является `-1`, который фактически означает "столько, сколько необходимо", поэтому `reshape(-1, 1)` означает одну строку и столько столбцов, сколько необходимо:

```
matrix.reshape(1, -1)
```

```
array([[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]])
```

Наконец, если мы предоставим одно целое число, то метод `reshape` вернет одномерный массив этой длины:

```
matrix.reshape(12)
```

```
array([ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
```

1.10. Транспонирование вектора в матрицу

Задача

Требуется транспонировать вектор в матрицу.

Решение

Использовать метод `T`:

```
# Загрузить библиотеку
import numpy as np

# Создать матрицу
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Транспонировать матрицу
matrix.T
```

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

Обсуждение

Транспонирование — это универсальная линейно-алгебраическая операция, в которой индексы столбцов и строк каждого элемента меняются местами. Вне предмета линейной алгебры, как правило, игнорируется один нюанс, который заключается в том, что технически вектор не может быть транспонирован, потому что он является лишь коллекцией значений:

```
# Транспонировать вектор
np.array([1, 2, 3, 4, 5, 6]).T
```

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

Вместе с тем общепринято называть транспонирование вектора преобразованием вектора-строки в вектор-столбец (обратите внимание на вторую пару скобок) или наоборот:

```
# Транспонировать вектор-строку
np.array([[1, 2, 3, 4, 5, 6]]).T
```

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

1.11. Сглаживание матрицы

Задача

Требуется преобразовать матрицу в одномерный массив.

Решение

Использовать метод `flatten`:

```
# Загрузить библиотеку
import numpy as np

# Создать матрицу
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Сгладить матрицу
matrix.flatten()
```

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

Обсуждение

Метод `flatten` представляет собой простой метод преобразования матрицы в одномерный массив. В качестве альтернативы, чтобы создать вектор-строку, мы можем применить метод `reshape`:

```
matrix.reshape(1, -1)
```

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

1.12. Нахождение ранга матрицы

Задача

Требуется узнать ранг матрицы.

Решение

Использовать линейно-алгебраический метод `matrix_rank` библиотеки NumPy:

```
# Загрузить библиотеку
import numpy as np

# Создать матрицу
matrix = np.array([[1, 1, 1],
                   [1, 1, 10],
                   [1, 1, 15]])
```

```
# Вернуть ранг матрицы
np.linalg.matrix_rank(matrix)
2
```

Обсуждение

Ранг матрицы — это размерности векторного пространства, которые покрываются ее столбцами или строками. В библиотеке NumPy найти ранг матрицы легко благодаря методу `matrix_rank`.

Дополнительные материалы для чтения

- "Ранг матрицы", учебный ресурс CliffsNotes (**http://bit.ly/2HUzkMs**).

1.13. Вычисление определителя матрицы

Задача

Требуется узнать определитель матрицы.

Решение

Использовать линейно-алгебраический метод `det` библиотеки NumPy:

```
# Загрузить библиотеку
import numpy as np

# Создать матрицу
matrix = np.array([[1, 2, 3],
                   [2, 4, 6],
                   [3, 8, 9]])

# Вернуть определитель матрицы
np.linalg.det(matrix)

0.0
```

Обсуждение

Иногда может быть полезно вычислить определитель матрицы. Бибилиотека NumPy делает это легко с помощью метода `det`.

Дополнительные материалы для чтения

- "Определитель" | "Сущность линейной алгебры", глава 5, Youtube-канал 3Blue1Brown (**http://bit.ly/2FA6ToM**).
- "Определитель", математический ресурс Wolfram MathWorld (**http://bit.ly/2FxSUzC**).

1.14. Получение диагонали матрицы

Задача

Требуется получить элементы главной диагонали матрицы.

Решение

Использовать метод `diagonal`:

```
# Загрузить библиотеку
import numpy as np

# Создать матрицу
matrix = np.array([[1, 2, 3],
                   [2, 4, 6],
                   [3, 8, 9]])

# Вернуть диагональные элементы
matrix.diagonal()
```

Обсуждение

Библиотека NumPy позволяет легко получать элементы главной диагонали матрицы с помощью метода `diagonal`. Кроме того, с помощью параметра `offset` можно получить диагональ в стороне от главной диагонали:

```
# Вернуть диагональ на одну выше главной диагонали
matrix.diagonal(offset=1)

array([2, 6])

# Вернуть диагональ на одну ниже главной диагонали
matrix.diagonal(offset=-1)

array([2, 8])
```

1.15. Вычисление следа матрицы

Задача

Требуется вычислить след матрицы.

Решение

Использовать метод `trace`:

```
# Загрузить библиотеку
import numpy as np
```

```
# Создать матрицу
matrix = np.array([[1, 2, 3],
                   [2, 4, 6],
                   [3, 8, 9]])

# Вырнуть след
matrix.trace()
```

```
14
```

Обсуждение

След матрицы является суммой элементов главной диагонали и часто используется за кадром в методах машинного самообучения. Имея многомерный массив NumPy, мы можем вычислить след с помощью метода `trace`. В качестве альтернативы мы также можем вернуть диагональ матрицы и вычислить сумму ее элементов:

```
# Вернуть диагональ и сумму ее элементов
sum(matrix.diagonal())
```

```
14
```

Дополнительные материалы для чтения

- "След квадратной матрицы", математический ресурс MathOnline (**http://bit.ly/2FunM45**).

1.16. Нахождение собственных значений и собственных векторов

Задача

Требуется найти собственные значения и собственные векторы квадратной матрицы.

Решение

Использовать метод `linalg.eig` библиотеки NumPy:

```
# Загрузить библиотеку
import numpy as np

# Создать матрицу
matrix = np.array([[1, -1, 3],
                   [1,  1, 6],
                   [3,  8, 9]])
```

```
# Вычислить собственные значения и собственные векторы
eigenvalues, eigenvectors = np.linalg.eig(matrix)

# Взглянуть на собственные значения
eigenvalues

array([ 13.55075847, 0.74003145, -3.29078992])

# Взглянуть на собственные векторы
eigenvectors

array([[-0.17622017, -0.96677403, -0.53373322],
       [-0.435951  ,  0.2053623 , -0.64324848],
       [-0.88254925,  0.15223105,  0.54896288]])
```

Обсуждение

Собственные векторы широко используются в библиотеках машинного самообучения. В интуитивном плане, имея линейное преобразование, представленное матрицей **A**, можно сказать, что собственные векторы — это векторы, которые при применении этого преобразования изменяются только в масштабе (а не в направлении). Более формально:

$$\mathbf{Av} = \lambda\mathbf{v},$$

где **A** — это квадратная матрица; λ — собственное значение; **v** — собственный вектор.

В наборе линейно-алгебраических инструментов библиотеки NumPy метод `eig` позволяет вычислять собственные значения и собственные векторы любой квадратной матрицы.

Дополнительные материалы для чтения

- "Собственные векторы и собственные значения с визуальным объяснением", математический проект Explained Visually (**http://bit.ly/2Hb32LV**).
- "Собственные векторы и собственные значения" | "Сущность линейной алгебры", глава 10, Youtube-канал 3Blue1Brown (**http://bit.ly/2HeGppK**).

1.17. Вычисление скалярных произведений

Задача

Требуется вычислить скалярное произведение двух векторов[1].

Решение

Использовать класс `dot` библиотеки NumPy:

```
# Загрузить библиотеку
import numpy as np

# Создать два вектора
vector_a = np.array([1,2,3])
vector_b = np.array([4,5,6])

# Вычислить скалярное произведение
np.dot(vector_a, vector_b)
```

```
32
```

Обсуждение

Скалярное произведение двух векторов **a** и **b** определяется как:

$$\sum_{i=1}^{n} a_i b_i,$$

где a_i, b_i — i-е элементы векторов **a** и **b** соответственно.

Для вычисления скалярного произведения используется класс `dot` библиотеки NumPy. В качестве альтернативы в Python 3.5+ можно применить новый оператор `@`:

```
# Вычислить скалярное произведение
vector_a @ vector_b
```

```
32
```

Дополнительные материалы для чтения

- "Скалярное произведение векторов и длина вектора", общепредметный учебный ресурс Khan Academy (**http://bit.ly/2Fr0AUe**).
- "Скалярное произведение", математический ресурс Paul's Online Math Notes (**http://bit.ly/2HgZHLp**).

[1] Иногда говорят внутреннее произведение или умножение точкой (dot product). — *Прим. перев.*

1.18. Сложение и вычитание матриц

Задача

Требуется сложить или вычесть две матрицы.

Решение

Использовать методы `add` и `subtract` библиотеки NumPy:

```
# Загрузить библиотеку
import numpy as np

# Создать матрицу
matrix_a = np.array([[1, 1, 1],
                     [1, 1, 1],
                     [1, 1, 2]])

# Создать матрицу
matrix_b = np.array([[1, 3, 1],
                     [1, 3, 1],
                     [1, 3, 8]])

# Сложить две матрицы
np.add(matrix_a, matrix_b)

array([[ 2, 4,  2],
       [ 2, 4,  2],
       [ 2, 4, 10]])

# Вычесть из одной матрицы другую
np.subtract(matrix_a, matrix_b)

array([[ 0, -2,  0],
       [ 0, -2,  0],
       [ 0, -2, -6]])
```

Обсуждение

В качестве альтернативы можно просто применить операторы `+` и `-`:

```
# Сложить две матрицы
matrix_a + matrix_b

array([[ 2, 4,  2],
       [ 2, 4,  2],
       [ 2, 4, 10]])
```

1.19. Умножение матриц

Задача

Требуется перемножить две матрицы.

Решение

Использовать класс `dot` библиотеки NumPy:

```
# Загрузить библиотеку
import numpy as np

# Создать матрицу
matrix_a = np.array([[1, 1],
                     [1, 2]])

# Создать матрицу
matrix_b = np.array([[1, 3],
                     [1, 2]])

# Перемножить две матрицы
np.dot(matrix_a, matrix_b)

array([[2, 5],
       [3, 7]])
```

Обсуждение

В качестве альтернативы в Python 3.5+ можно применить оператор `@`:

```
# Перемножить две матрицы
matrix_a @ matrix_b

array([[2, 5],
       [3, 7]])
```

Если требуется выполнить поэлементное умножение, то можно применить оператор `*`:

```
# Перемножить две матрицы поэлементно
matrix_a * matrix_b

array([[1, 3],
       [1, 4]])
```

Дополнительные материалы для чтения

- "Операции над массивами против матричных операций", специализированный ресурс MathWorks для глубокого самообучения (**http://bit.ly/2FtpXVr**).

1.20. Обращение матрицы

Задача

Требуется вычислить обратную квадратную матрицу.

Решение

Использовать линейно-алгебраический метод `inv` библиотеки NumPy:

```
# Загрузить библиотеку
import numpy as np

# Создать матрицу
matrix = np.array([[1, 4],
                   [2, 5]])

# Вычислить обратную матрицу
np.linalg.inv(matrix)

array([[-1.66666667,  1.33333333],
       [ 0.66666667, -0.33333333]])
```

Обсуждение

Матрицей, обратной квадратной матрице **A**, является вторая матрица $\mathbf{A}^{-1}$, такая, что:

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{I},$$

где **I** — единичная матрица. В NumPy для вычисления $\mathbf{A}^{-1}$, если она существует, используется метод `linalg.inv`. Для того чтобы увидеть это в действии, можно умножить матрицу на обратную ей, в результате будет получена единичная матрица:

```
# Умножить матрицу на обратную ей матрицу
matrix @ np.linalg.inv(matrix)

array([[ 1., 0.],
       [ 0., 1.]])
```

Дополнительные материалы для чтения

- "Обратная матрица", математический ресурс mathwords.com (**http://bit.ly/2Fzf0BS**).

1.21. Генерирование случайных значений

Задача

Требуется сгенерировать псевдослучайные значения.

Решение

Использовать метод `random` библиотеки NumPy:

```
# Загрузить библиотеку
import numpy as np

# Задать начальное значение для генератора псевдослучайных чисел
np.random.seed(0)

# Сгенерировать три случайных вещественных числа между 0.0 и 1.0
np.random.random(3)

array([ 0.5488135 , 0.71518937, 0.60276338])
```

Обсуждение

NumPy предлагает широкий спектр средств генерации случайных чисел, количество которых гораздо больше, чем здесь можно охватить. В нашем решении мы сгенерировали вещественные числа; вместе с тем, генерация целых чисел также имеет широкое применение:

```
# Сгенерировать три случайных целых числа между 1 и 10
np.random.randint(0, 11, 3)

array([3, 7, 9])
```

В качестве альтернативы мы можем генерировать числа, извлекая их из распределения:

```
# Извлечь три числа из нормального распределения со средним, равным 0.0,
# и стандартным отклонением, равным 1.0
np.random.normal(0.0, 1.0, 3)

array([-1.42232584, 1.52006949, -0.29139398])

# Извлечь три числа из логистического распределения со средним, равным 0.0,
# и масштабом, равным 1.0
np.random.logistic(0.0, 1.0, 3)

array([-0.98118713, -0.08939902, 1.46416405])
```

```
# Извлечь три числа, которые больше или равны 1.0 и меньше 2.0
np.random.uniform(1.0, 2.0, 3)
```

```
array([ 1.47997717, 1.3927848 , 1.83607876])
```

Наконец, иногда может быть полезно возвращать одни и те же случайные числа несколько раз, чтобы получать предсказуемые, повторяемые результаты. Это можно сделать, задав "начальное значение" (целое число) генератора псевдослучайных чисел. Случайные процессы с одними и теми же начальными значениями всегда будут порождать один и тот же результат. Мы будем использовать начальные значения на протяжении всей этой книги, чтобы программный код, который вы видите в книге, и программный код, который вы выполняете на своем компьютере, давали одинаковые результаты.

Загрузка данных

Введение

Первым шагом в любом начинании в области машинного самообучения является введение исходных данных в нашу систему. Сырые данные могут быть файлом журнала операций, файлом набора данных или базой данных. Кроме того, часто требуется получить данные из нескольких источников. Рецепты в этой главе обращаются к методам загрузки данных из различных источников, включая CSV-файлы и базы данных SQL. Мы также рассмотрим методы генерирования симулированных данных с желаемыми для экспериментов свойствами. Наконец, хотя существует много способов загрузки данных в экосистему Python, мы сосредоточимся на использовании обширного набора методов библиотеки pandas для загрузки внешних данных и использовании scikit-learn — библиотеки машинного самообучения с открытым исходным кодом — для генерирования симулированных данных.

2.1. Загрузка образца набора данных

Задача

Требуется загрузить существующий образец набора данных.

Решение

Библиотека scikit-learn поставляется с рядом популярных наборов данных для использования:

```
# Загрузить наборы данных scikit-learn
from sklearn import datasets

# Загрузить набор изображений рукописных цифр
digits = datasets.load_digits()

# Создать матрицу признаков
features = digits.data

# Создать вектор целей
target = digits.target
```

```
# Взглянуть на первое наблюдение
features[0]
```

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

Обсуждение

Нередко мы хотим избежать работы, связанной с загрузкой, преобразованием и очисткой реального набора данных, до тех пор, пока не изучим какой-либо алгоритм или метод машинного самообучения. К счастью, библиотека scikit-learn поставляется с несколькими универсальными наборами данных, которые можно быстро загрузить. Эти наборы данных часто называют *игрушечными*, потому что они намного меньше и чище, чем набор данных, который мы встречаем в реальном мире. Приведем некоторые популярные образцы наборов данных в scikit-learn[1]:

- `load_boston` содержит 503 образцов цен на жилье в Бостоне; это хороший набор данных для изучения регрессионных алгоритмов;
- `load_iris` содержит 150 образцов измерений цветков ириса; это хороший набор данных для изучения классификационных алгоритмов;
- `load_digits` содержит 1797 образцов изображений рукописных цифр; это хороший набор данных для тренировки модели, классифицирующей изображения.

Дополнительные материалы для чтения

- Игрушечные наборы данных библиотеки scikit-learn (**http://bit.ly/2HS6Dzq**).
- Набор изображений рукописных цифр Digit (**http://bit.ly/2mNSEBZ**).

[1] В машинном самообучении принято работать с матрицами признаков. Каждая строка матрицы называется наблюдением, образцом, прецедентом или записью, а каждый столбец — признаком. Например, известный набор данных цветков ириса Фишера состоит из 150 наблюдений, и каждое наблюдение имеет 4 признака с данными замеров цветков в сантиметрах. Таким образом, размерность наблюдения — это четырехмерный вектор-строка, а размерность признака — 150-мерный вектор-столбец. Функции по работе с игрушечными наборами данных scikit-learn возвращают объект-словарь, в котором по ключу `data` в качестве матрицы признаков хранится массив формы *n*_образцов × *n*_признаков, а по ключу `target` — вектор переменной отклика, или вектор целевых значений, или просто вектор целей (см. **http://scikit-learn.org/stable/datasets/index.html#datasets**). — *Прим. перев.*

2.2. Создание симулированного набора данных

Задача

Требуется сгенерировать набор симулированных данных.

Решение

Библиотека scikit-learn предлагает целый ряд методов для создания симулированных данных. Из них особенно полезными являются три метода.

Если требуется набор данных, предназначенный для использования с линейной регрессией, то метод `make_regression` является хорошим выбором:

```
# Загрузить библиотеку
from sklearn.datasets import make_regression

# Сгенерировать матрицу признаков, вектор целей и истинные коэффициенты
features, target, coefficients = make_regression(n_samples = 100,
                                                 n_features = 3,
                                                 n_informative = 3,
                                                 n_targets = 1,
                                                 noise = 0.0,
                                                 coef = True,
                                                 random_state = 1)

# Взглянуть на матрицу признаков и вектор целей
print('Матрица признаков\n', features[:3])
print('Вектор целей\n', target[:3])

Матрица признаков
 [[ 1.29322588 -0.61736206 -0.11044703]
 [-2.793085 0.36633201 1.93752881]
 [ 0.80186103 -0.18656977 0.0465673 ]]
Вектор целей
 [-10.37865986 25.5124503 19.67705609]
```

Если мы заинтересованы в создании симулированного набора данных для задачи классификации, то можно использовать метод `make_classification`:

```
# Загрузить библиотеку
from sklearn.datasets import make_classification

# Сгенерировать матрицу признаков и вектор целей
features, target = make_classification(n_samples = 100,
                                       n_features = 3,
                                       n_informative = 3,
                                       n_redundant = 0,
                                       n_classes = 2,
                                       weights = [.25, .75],
                                       random_state = 1)
```

```
# Взглянуть на матрицу признаков и вектор целей
print('Матрица признаков\n', features[:3])
print('Вектор целей\n', target[:3])

Матрица признаков
 [[ 1.06354768 -1.42632219 1.02163151]
 [ 0.23156977 1.49535261 0.33251578]
 [ 0.15972951 0.83533515 -0.40869554]]
Вектор целей
 [1 0 0]
```

Наконец, если требуется, чтобы набор данных хорошо работал с кластеризующими методами, то библиотека scikit-learn предлагает метод создания скоплений точек `make_blobs`:

```
# Загрузить библиотеку
from sklearn.datasets import make_blobs

# Сгенерировать матрицу признаков и вектор целей
features, target = make_blobs(n_samples = 100,
                              n_features = 2,
                              centers = 3,
                              cluster_std = 0.5,
                              shuffle = True,
                              random_state = 1)

# Взглянуть на матрицу признаков и вектор целей
print('Матрица признаков\n', features[:3])
print('Вектор целей\n', target[:3])

Матрица признаков
 [[ -1.22685609 3.25572052]
 [ -9.57463218 -4.38310652]
 [-10.71976941 -4.20558148]]
Вектор целей
 [0 1 1]
```

Обсуждение

Как видно из этих решений, функция `make_regression` возвращает матрицу признаков и вектор целей, состоящие из вещественных значений, в то время как функции `make_classification` и `make_blobs` возвращают матрицу признаков, состоящую из вещественных значений, а вектор целей — из целочисленных значений. Значения вектора целей обозначают принадлежность к классу.

Симулированные наборы данных библиотеки scikit-learn предоставляют широкие возможности по управлению типом создаваемых данных. Документация scikit-learn содержит полное описание всех параметров, но некоторые из них стоит отметить.

В функциях `make_regression` и `make_classification` параметр `n_informative` задает количество признаков, используемых для создания вектора целей. Если значение параметра `n_informative` меньше общего количества признаков (`n_features`), то результирующий набор данных будет иметь избыточные признаки, которые можно идентифицировать с помощью методов отбора признаков.

Кроме того, функция `make_classification` содержит параметр весов `weights`, который позволяет симулировать наборы данных с несбалансированными классами. Например, `weights = [.25, .75]` возвращает набор данных с 25% наблюдений, принадлежащих к одному классу, и 75% наблюдений, принадлежащих ко второму классу.

Для функции `make_blobs` параметр `centers` задает количество сгенерированных кластеров. С помощью библиотеки `matplotlib` можно визуализировать кластеры, созданные функцией `make_blobs` (рис. 2.1):

```
# Загрузить библиотеку
import matplotlib.pyplot as plt

# Взглянуть на диаграмму рассеяния
plt.scatter(features[:,0], features[:,1], c=target)
plt.show()
```

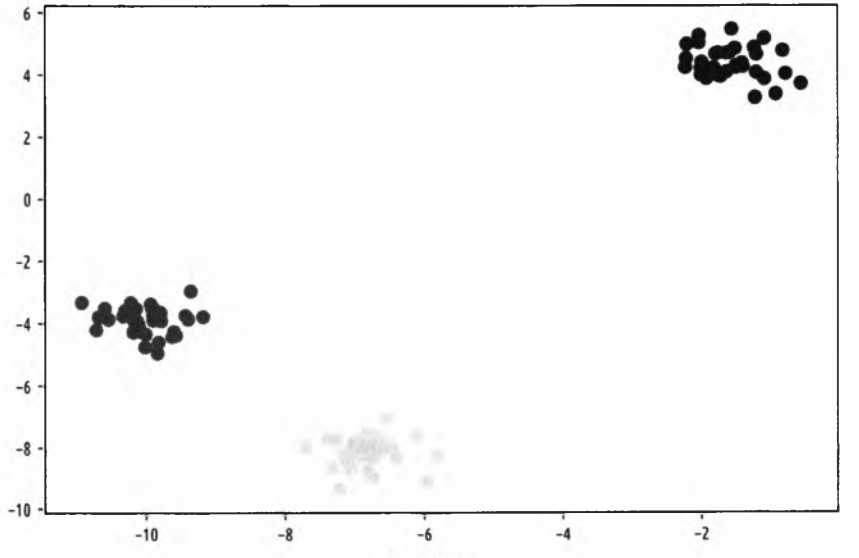


Рис. 2.1

Дополнительные материалы для чтения

- Документация по `make_regression` (**http://bit.ly/2FtIBwo**).
- Документация по `make_classification` (**http://bit.ly/2FtIKzW**).
- Документация по `make_blobs` (**http://bit.ly/2FqKMAZ**).

2.3. Загрузка файла CSV

Задача

Требуется импортировать файл значений с разделителями-запятыми (файл в формате CSV).

Решение

Для того чтобы загрузить локальный или размещенный в сети файл CSV, нужно использовать функцию `read_csv` библиотеки pandas (табл. 2.1):

```
# Загрузить библиотеку
import pandas as pd

# Создать URL-адрес
url = 'https://tinyurl.com/simulated-data'

# Загрузить набор данных
dataframe = pd.read_csv(url)

# Взглянуть на первые две строки
dataframe.head(2)
```

Таблица 2.1

	Integer	datetime	category
0	5	2015-01-01 00:00:00	0
1	5	2015-01-01 00:00:01	0

Обсуждение

В отношении загрузки файлов CSV следует отметить два момента. Во-первых, часто бывает полезно заглянуть внутрь файла перед загрузкой. Это может быть очень полезно для того, чтобы заранее увидеть, как набор данных структурирован и какие параметры нужно установить, чтобы загрузить файл. Во-вторых, функция `read_csv` имеет более 30 параметров, и поэтому документация способна обескуражить. К счастью, эти параметры существуют в основном для того, чтобы дать возможность обрабатывать широкий спектр форматов CSV. Например, в файлах CSV значения разделены запятыми (например, одна строка может быть `2,"2015-01-01 00:00:00",0`); однако в файлах CSV в качестве разделителей часто используются другие символы, например символ табуляции. Параметр `sep` библиотеки pandas позволяет задавать используемый в файле разделитель. Хотя это бывает нечасто, проблема форматирования файлов CSV заключается в том, что первая строка файла используется для определения заголовков столбцов (например, `integer`, `datetime`, `category`, как в нашем решении). Параметр `header` позволяет указывать, существует

ли строка заголовка и где она находится. Если строка заголовка не существует, то мы устанавливаем `header=None`.

2.4. Загрузка файла Excel

Задача

Требуется загрузить электронную таблицу Excel.

Решение

Для того чтобы загрузить электронную таблицу Excel, нужно использовать функцию `read_excel` библиотеки pandas (табл. 2.2):

```
# Загрузить библиотеку
import pandas as pd

# Создать URL-адрес
url = 'https://tinyurl.com/simulated-excel'

# Загрузить данные
dataframe = pd.read_excel(url, sheet_name=0, header=1)

# Взглянуть на первые две строки
dataframe.head(2)
```

Таблица 2.2

	5	2015-01-01 00:00:00	0
0	5	2015-01-01 00:00:01	0
1	9	2015-01-01 00:00:02	0

Обсуждение

Это решение похоже на наше решение для чтения файлов CSV. Основным отличием является дополнительный параметр `sheet_name`, который указывает, какой лист в файле Excel мы хотим загрузить. Параметр `sheet_name` может принимать как строковые значения, содержащие имя листа, так и целые числа, указывающие на позиции листа (с нулевой индексацией). Если нужно загрузить несколько листов, то включите их в список. Например, `sheet_name=[0,1,2, "Ежемесячные продажи"]` вернет словарь фреймов данных pandas, содержащий первый, второй и третий листы и лист с именем `Ежемесячные продажи`.

2.5. Загрузка файла JSON

Задача

Требуется загрузить файл JSON для предобработки данных.

Решение

Библиотека pandas предоставляет функцию `read_json` для преобразования файла JSON в объект pandas (рис. 2.3):

```
# Загрузить библиотеку
import pandas as pd

# Создать URL-адрес
url = 'https://tinyurl.com/simulated-json'

# Загрузить данные
dataframe = pd.read_json(url, orient='columns')

# Взглянуть на первые две строки
dataframe.head(2)
```

Таблица 2.3

	category	datetime	integer
0	0	2015-01-01 00:00:00	5
1	0	2015-01-01 00:00:01	5

Обсуждение

Импорт файлов JSON в pandas похож на предыдущие несколько рецептов, которые мы рассмотрели. Ключевым отличием является параметр `orient`, который указывает pandas, как структурирован файл JSON. Однако, чтобы выяснить, какое значение аргумента (`split`, `records`, `index`, `columns` и `values`) является правильным, может потребоваться немного поэкспериментировать. Библиотека pandas предлагает еще один полезный инструмент — функцию `json_normalize`, которая помогает преобразовывать полуструктурированные данные JSON во фрейм данных pandas.

Дополнительные материалы для чтения

- Документация по `json_normalize` (**http://bit.ly/2HQqwaa**).

2.6. Опрашивание базы данных SQL

Задача

Требуется загрузить данные из базы данных с помощью языка структурированных запросов (SQL).

Решение

Функция `read_sql_query` библиотеки pandas позволяет выполнить запрос на SQL к базе данных и загрузить данные (табл. 2.4):

```
# Загрузить библиотеки
import pandas as pd
from sqlalchemy import create_engine

# Создать подключение к базе данных
database_connection = create_engine('sqlite:///sample.db')

# Загрузить данные
dataframe = pd.read_sql_query('SELECT * FROM data', database_connection)

# Взглянуть на первые две строки
dataframe.head(2)
```

Таблица 2.4

	first_name	last_name	age	preTestScore	postTestScore
0	Jason	Miller	42	4	25
1	Molly	Jacobson	52	24	94

Обсуждение

Из всех рецептов, представленных в этой главе, этот, вероятно, будет наиболее широко используемым в реальном мире. SQL — универсальный язык для извлечения данных из баз данных. В этом рецепте мы сначала применяем функцию `create_engine` для установления подключения к ядру базы данных SQL под названием SQLite. Затем мы используем функцию `read_sql_query` библиотеки pandas, чтобы опросить эту базу данных с помощью SQL и поместить результаты во фрейм данных.

SQL сам по себе является языком, и хотя это выходит за рамки настоящей книги, для любого, кто желает изучить машинное самообучение, он, безусловно, заслуживает того, чтобы познакомиться с ним поближе. Наш SQL-запрос `SELECT * FROM data` просит базу данных предоставить нам все столбцы (`*`) из таблицы `data`.

Дополнительные материалы для чтения

- SQLite, автономное встраиваемое ядро реляционной базы данных (**https://www.sqlite.org**).
- Учебное пособие по SQL от W3Schools (**https://www.w3schools.com/sql**).

Упорядочение данных

Введение

Упорядочение данных[1] — термин, охватывающий широкий диапазон тем и часто неофициально используемый для описания процесса преобразования сырых данных в чистый и организованный формат, готовый к использованию. Для нас упорядочение данных — это лишь один из шагов в предобработке данных, но это важный шаг.

Наиболее распространенной структурой данных, используемой для "упорядочения", является фрейм данных, который может быть и интуитивно понятным, и невероятно универсальным. Фреймы данных являются табличными, т. е. они основаны на строках и столбцах, какие вы встречаете в электронной таблице. Приведем фрейм данных, созданный из данных о пассажирах на "Титанике" (табл. 3.1):

```
# Загрузить библиотеку
import pandas as pd

# Создать URL-адрес
url = 'https://tinyurl.com/titanic-csv'

# Загрузить данные как фрейм данных
dataframe = pd.read_csv(url)

# Показать первые 5 строк
dataframe.head(5)
```

Таблица 3.1

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.00	female	1	1
1	Allison, Miss Helen Loraine	1st	2.00	female	0	1
2	Allison, Mr Hudson Joshua Creighton	1st	30.00	male	0	0
3	Allison, Mrs Hudson JC (Bessie Waldo Daniels)	1st	25.00	female	0	1
4	Allison, Master Hudson Trevor	1st	0.92	male	1	0

[1] Выверка, упорядочение, согласование данных (data wrangling). — *Прим. перев.*

В этом фрейме данных можно заметить три важных момента.

Во-первых, во фрейме данных каждая строка соответствует одному наблюдению (например, пассажиру) и каждый столбец соответствует одному признаку (полу, возрасту и т. д.). Например, глядя на первое наблюдение, мы видим, что мисс Элизабет Уолтон Аллен путешествовала первым классом, ей было 29 лет, она была женщиной и пережила катастрофу.

Во-вторых, каждый столбец содержит имя (например, `Name`, `PClass`, `Age`) и каждая строка содержит индексный номер (например, `0` для счастливой мисс Элизабет Уолтон Аллен). Мы будем использовать их для выбора и управления наблюдениями и признаками.

В-третьих, два столбца, `Sex` и `SexCode`, содержат одну и ту же информацию в разных форматах. В столбце `Sex` женщина обозначается строкой `female`, в то время как в столбце `SexCode` женщина обозначается целым числом `1`. Мы хотим, чтобы все наши признаки были уникальными, и поэтому нам нужно будет удалить один из этих столбцов.

В этой главе мы рассмотрим широкий спектр методов управления фреймами данных с помощью библиотеки pandas с целью создания чистого, хорошо структурированного набора наблюдений для дальнейшей предобработки.

3.1. Создание фрейма данных

Задача

Требуется создать новый фрейм данных.

Решение

Библиотека pandas имеет много методов создания нового объекта `DataFrame`. Одним из простых методов является создание пустого фрейма данных с помощью конструктора `DataFrame`, а затем определение каждого столбца по отдельности (табл. 3.2):

```
# Загрузить библиотеку
import pandas as pd

# Создать фрейм данных DataFrame
dataframe = pd.DataFrame()

# Добавить столбцы
dataframe['Имя'] = ['Джеки Джексон', 'Стивен Стивенсон']
dataframe['Возраст'] = [38, 25]
dataframe['Водитель'] = [True, False]

# Показать DataFrame
dataframe
```

Таблица 3.2

	Имя	Возраст	Водитель
0	Джеки Джексон	38	True
1	Стивен Стивенсон	25	False

В качестве альтернативы, после того как мы создали объект `DataFrame`, мы можем добавить новые строки в конец фрейма данных (табл. 3.3):

```
# Создать строку
new_person = pd.Series(['Молли Муни', 40, True],
                       index=['Имя','Возраст','Водитель'])

# Добавить строку в конец фрейма данных
dataframe.append(new_person, ignore_index=True)
```

Таблица 3.3

	Имя	Возраст	Водитель
0	Джеки Джексон	38	True
1	Стивен Стивенсон	25	False
2	Молли Муни	40	True

Обсуждение

Может показаться, что библиотека pandas предлагает бесконечное количество способов создания фрейма данных. В реальном мире почти никогда не требуется создавать пустой фрейм данных и затем его заполнять. Вместо этого наши фреймы данных будут создаваться из реальных данных, загружаемых из других источников (например, файла CSV или базы данных).

3.2. Описание данных

Задача

Требуется взглянуть на некоторые характеристики фрейма данных.

Решение

Одна из самых простых вещей, которые мы можем сделать после загрузки данных, — это взглянуть на первые несколько строк с помощью метода `head` (табл. 3.4):

```
# Загрузить библиотеку
import pandas as pd
```

```
# Создать URL-адрес
url = 'https://tinyurl.com/titanic-csv'

# Загрузить данные
dataframe = pd.read_csv(url)

# Показать две строки
dataframe.head(2)
```

Таблица 3.4

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1

Мы также можем взглянуть на количество строк и столбцов:

```
# Показать размерности
dataframe.shape
```

```
(1313, 6)
```

Кроме того, используя метод `describe`, мы можем получить описательную статистику для любых числовых столбцов (табл. 3.5):

```
# Показать статистику
dataframe.describe()
```

Таблица 3.5

	Age	Survived	SexCode
count	756.000000	1313.000000	1313.000000
mean	30.397989	0.342727	0.351866
std	14.259049	0.474802	0.477734
min	0.170000	0.000000	0.000000
25%	21.000000	0.000000	0.000000
50%	28.000000	0.000000	0.000000
75%	39.000000	1.000000	1.000000
max	71.000000	1.000000	1.000000

Обсуждение

После того как мы загрузили немного данных, неплохо понять, как они структурированы и какую информацию они содержат. В идеале мы будем просматривать все данные целиком. Но в большинстве реальных случаев данные могут содержать от

тысяч до сотен тысяч или миллионов строк и столбцов. Вместо этого мы должны опираться на извлечение выборок с целью просмотра небольших срезов и вычисления сводной статистики этих данных.

В нашем решении мы используем игрушечный набор данных пассажиров последнего трагического путешествия "Титаника". При помощи метода `head` мы можем взглянуть на первые несколько строк данных (пять по умолчанию). Кроме того, мы можем применить метод `tail` для просмотра последних нескольких строк. С помощью атрибута `shape` мы можем увидеть, сколько строк и столбцов содержится в нашем фрейме. И наконец, с помощью метода `describe` мы можем увидеть некоторые основные описательные статистики для любого числового столбца.

Стоит отметить, что сводная статистика не всегда рассказывает полную историю. Например, библиотека pandas рассматривает столбцы `Survived` и `SexCode` как числовые столбцы, поскольку они содержат единицы и нули. Однако в данном случае числовые значения представляют категории. Например, если `Survived` равняется 1, то это означает, что пассажир пережил катастрофу. По этой причине некоторые из представленных сводных статистических показателей не имеют смысла, например стандартное отклонение столбца `SexCode` (показателя пола пассажира).

3.3. Навигация по фреймам данных

Задача

Требуется выбрать индивидуальные данные или срезы фрейма данных.

Решение

Для выбора одной или нескольких строк либо значений использовать методы `loc` или `iloc`:

```
# Загрузить библиотеку
import pandas as pd

# Создать URL-адрес
url = 'https://tinyurl.com/titanic-csv'

# Загрузить данные
dataframe = pd.read_csv(url)

# Выбрать первую строку
dataframe.iloc[0]
```

```
Name          Allen, Miss Elisabeth Walton
PClass                                 1st
Age                                     29
Sex                                 female
```

```
Survived                    1
SexCode                     1
Name: 0, dtype: object
```

Для определения среза строк, который требуется получить, применяется оператор :, например, для выбора второй, третьей и четвертой строк (табл. 3.6):

```
# Выбрать три строки
dataframe.iloc[1:4]
```

Таблица 3.6

	Name	PClass	Age	Sex	Survived	SexCode
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1
2	Allison, Mr Hudson Joshua Creighton	1st	30.0	male	0	0
3	Allison, Mrs Hudson JC (Bessie Waldo Daniels)	1st	25.0	female	0	1

Его также можно использовать, чтобы получить все строки до определенной точки, например, все строки вплоть до четвертой строки включительно (табл. 3.7):

```
# Выбрать четыре строки
dataframe.iloc[:4]
```

Таблица 3.7

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1
2	Allison, Mr Hudson Joshua Creighton	1st	30.0	male	0	0
3	Allison, Mrs Hudson JC (Bessie Waldo Daniels)	1st	25.0	female	0	1

Фреймы данных не нуждаются в числовой индексации. Мы можем определить индекс фрейма любым значением, где значение является уникальным для каждой строки. Например, мы можем задать индекс на именах пассажиров, а затем выбирать строки, используя имя:

```
# Задать индекс
dataframe = dataframe.set_index(dataframe['Name'])

# Показать строку
dataframe.loc['Allen, Miss Elisabeth Walton']

Name          Allen, Miss Elisabeth Walton
PClass                                 1st
Age                                     29
```

```
Sex                                    female
Survived                                    1
SexCode                                     1
Name: Allen, Miss Elisabeth Walton, dtype: object
```

Обсуждение

Все строки во фрейме данных pandas имеют уникальное индексное значение. По умолчанию этот индекс является целым числом, указывающим на положение строки во фрейме данных; однако это не обязательно. Индексы фреймов могут быть уникальными буквенно-цифровыми строковыми значениями или номерами клиентов. Для выбора отдельных строк и срезов строк библиотека pandas предоставляет два метода:

- метод `loc` полезен, когда индекс фрейма является меткой (например, строковым значением);
- метод `iloc` работает, отыскивая позицию во фрейме данных; например, `iloc[0]` вернет первую строку независимо от того, является ли индекс целым числом или меткой.

Очень важно научиться свободно пользоваться как `loc`, так и `iloc`, т. к. эти методы будут применяться много раз во время очистки данных.

3.4. Выбор строк на основе условных конструкций

Задача

Требуется отобрать строки фрейма данных на основе некоторого условия.

Решение

В библиотеке pandas это делается легко. Например, если требуется выбрать всех женщин на "Титанике" (табл. 3.8):

```
# Загрузить библиотеку
import pandas as pd

# Создать URL-адрес
url = 'https://tinyurl.com/titanic-csv'

# Загрузить данные
dataframe = pd.read_csv(url)

# Показать верхние две строки, где столбец 'sex' равняется 'female'
dataframe[dataframe['Sex'] == 'female'].head(2)
```

Таблица 3.8

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1

Остановитесь на секунду и посмотрите на формат этого решения. Конструкция `dataframe['Sex'] == 'female'` — это наше условное выражение; обернув его в `dataframe[]`, мы поручаем библиотеке pandas "выбрать все строки во фрейме, где значение `dataframe['Sex']` равняется `'female'`".

Составление множественных условий тоже делается легко. Например, здесь мы выбираем все строки, где пассажирами являются женщины 65 лет и старше (табл. 3.9):

```
# Отфильтровать строки
dataframe[(dataframe['Sex'] == 'female') & (dataframe['Age'] >= 65)]
```

Таблица 3.9

	Name	PClass	Age	Sex	Survived	SexCode
73	Crosby, Mrs Edward Gifford (Catherine Elizabet...	1st	69.0	female	1	1

Обсуждение

Отбор по условию и фильтрация данных являются одними из наиболее распространенных задач во время упорядочения данных. Вам редко потребуются абсолютно все сырые данные из источника; вместо этого, как правило, интересует только какой-то их подраздел. Например, вас могут заинтересовать только магазины в конкретных местностях или истории болезней пациентов старше определенного возраста.

3.5. Замена значений

Задача

Требуется заменить значения во фрейме данных.

Решение

Метод `replace` библиотеки pandas — это простой способ найти и заменить значения. Например, любой прецедент "`female`" в столбце `Sex` можно заменить на "`Woman`":

```
# Загрузить библиотеку
import pandas as pd
```

```
# Создать URL-адрес
url = 'https://tinyurl.com/titanic-csv'

# Загрузить данные
dataframe = pd.read_csv(url)

# Заменить значения, показать две строки
dataframe['Sex'].replace("female", "Woman").head(2)
```

```
0    Woman
1    Woman
Name: Sex, dtype: object
```

Мы также можем найти и выполнить замену по всему объекту `DataFrame`, указав весь фрейм данных целиком вместо одного столбца (табл. 3.10):

```
# Заменить значения, показать две строки
dataframe.replace(1, "One").head(2)
```

Таблица 3.10

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	One	One
1	Allison, Miss Helen Loraine	1st	2.0	female	0	One

Метод `replace` также принимает регулярные выражения (табл. 3.11):

```
# Заменить значения, показать две строки
dataframe.replace(r"1st", "First", regex=True).head(2)
```

Таблица 3.11

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	First	29.0	female	1	1
1	Allison, Miss Helen Loraine	First	2.0	female	0	1

Обсуждение

Метод `replace` — это инструмент, используемый для замены значений, который прост и в то же время обладает мощной способностью принимать регулярные выражения.

3.6. Переименование столбцов

Задача

Требуется переименовать столбец во фрейме данных pandas.

Решение

Переименовать столбцы с помощью метода `rename` (табл. 3.12):

```
# Загрузить библиотеку
import pandas as pd

# Создать URL-адрес
url = 'https://tinyurl.com/titanic-csv'

# Загрузить данные
dataframe = pd.read_csv(url)

# Переименовать столбец, показать две строки
dataframe.rename(columns={'PClass': 'Passenger Class'}).head(2)
```

Таблица 3.12

	Name	Passenger Class	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1

Обратите внимание, что метод `rename` в качестве аргумента может принимать словарь, который используется для изменения нескольких имен столбцов одновременно (табл. 3.13):

```
# Переименовать столбцы, показать две строки
dataframe.rename(columns={'PClass': 'Passenger Class', 'Sex': 'Gender'}).head(2)
```

Таблица 3.13

	Name	Passenger Class	Age	Gender	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1

Обсуждение

Использование метода `rename` со словарем в качестве аргумента для параметра `columns` является моим предпочтительным способом переименования столбцов, поскольку он работает с любым количеством столбцов. Если требуется переимено-

вать все столбцы сразу, то ниже приведен полезный фрагмент кода, который создает словарь со старыми именами столбцов в качестве ключей и пустыми строковыми значениями в качестве значений:

```
# Загрузить библиотеку
import collections

# Создать словарь
column_names = collections.defaultdict(str)

# Создать ключи
for name in dataframe.columns:
    column_names[name]

# Показать словарь
column_names

defaultdict(str,
            {'Age': '',
             'Name': '',
             'PClass': '',
             'Sex': '',
             'SexCode': '',
             'Survived': ''})
```

3.7. Нахождение минимума, максимума, суммы, среднего арифметического и количества

Задача

Требуется найти минимум, максимум, сумму, среднее арифметическое и количество значений числового столбца.

Решение

Библиотека pandas поставляется с некоторыми встроенными методами для общепринятых описательных статистических показателей:

```
# Загрузить библиотеку
import pandas as pd

# Создать URL-адрес
url = 'https://tinyurl.com/titanic-csv'

# Загрузить данные
dataframe = pd.read_csv(url)
```

```
# Вычислить статистические показатели
print('Максимум:', dataframe['Age'].max())
print('Минимум:', dataframe['Age'].min())
print('Среднее:', dataframe['Age'].mean())
print('Сумма:', dataframe['Age'].sum())
print('Количество:', dataframe['Age'].count())

Максимум: 71.0
Минимум: 0.17
Среднее: 30.397989417989418
Сумма: 22980.88
Количество: 756
```

Обсуждение

Помимо статистических показателей, используемых в данном решении, библиотека pandas предлагает дисперсию (`var`), стандартное отклонение (`std`), коэффициент эксцесса (`kurt`), коэффициент асимметрии (`skew`), стандартную ошибку среднего (`sem`), моду (`mode`), медиану (`meadian`) и ряд других.

Более того, эти методы также можно применять ко всему фрейму данных:

```
# Показать количества значений
dataframe.count()

Name          1313
PClass        1313
Age            756
Sex           1313
Survived      1313
SexCode       1313
dtype: int64
```

3.8. Нахождение уникальных значений

Задача

Требуется выбрать все уникальные значения в столбце.

Решение

Для просмотра массива всех уникальных значений в столбце использовать метод `unique`:

```
# Загрузить библиотеку
import pandas as pd

# Создать URL-адрес
url = 'https://tinyurl.com/titanic-csv'
```

```
# Загрузить данные
dataframe = pd.read_csv(url)

# Выбрать уникальные значения
dataframe['Sex'].unique()

array(['female', 'male'], dtype=object)
```

В качестве альтернативы метод `value_counts` покажет все уникальные значения вместе с количеством появлений каждого значения:

```
# Показать количества появлений
dataframe['Sex'].value_counts()

male 851
female 462
Name: Sex, dtype: int64
```

Обсуждение

Методы `unique` и `value_counts` полезны для манипулирования и разведки категориальных столбцов. Очень часто в категориальных столбцах будут находиться классы, которые должны быть обработаны на этапе упорядочения данных. Например, в наборе данных *Titanic* `PClass` — это столбец, указывающий на класс пассажирского билета. На "Титанике" было три класса; однако если применить метод `value_counts`, то можно обнаружить проблему:

```
# Показать количества появлений
dataframe['PClass'].value_counts()

3rd    711
1st    322
2nd    279
*        1
Name: PClass, dtype: int64
```

Хотя, как и ожидалось, почти все пассажиры принадлежат к одному из трех классов, один пассажир имеет класс `*`. Для решения подобного рода проблем существует ряд стратегий, которые мы рассмотрим в *главе 5*, но пока просто осознаем, что "лишние" классы в категориальных данных являются обыденным делом, и их не следует игнорировать.

Наконец, если требуется просто подсчитать количество уникальных значений, то можно применить метод `nunique`:

```
# Показать количество уникальных значений
dataframe['PClass'].nunique()

4
```

3.9. Отбор пропущенных значений

Задача

Требуется выбрать пропущенные значения во фрейме данных.

Решение

Методы `isnull` и `notnull` возвращают булевы значения, указывающие, отсутствует значение или нет (табл. 3.14):

```
# Загрузить библиотеку
import pandas as pd

# Создать URL-адрес
url = 'https://tinyurl.com/titanic-csv'

# Загрузить данные
dataframe = pd.read_csv(url)

# Выбрать пропущенные значения, показать две строки
dataframe[dataframe['Age'].isnull()].head(2)
```

Таблица 3.14

	Name	PClass	Age	Sex	Survived	SexCode
12	Aubert, Mrs Leontine Pauline	1st	Nan	female	1	1
13	Barkworth, Mr Algernon H	1st	Nan	male	1	0

Обсуждение

Пропущенные значения являются повсеместной проблемой во время упорядочения данных, и все же многие недооценивают сложность работы с пропущенными данными. В библиотеке pandas для обозначения пропущенных значений используется значение `NaN` библиотеки NumPy (от англ. *Not a Number* — не число), но важно отметить, что в библиотеке pandas значение `NaN` реализовано нативно не до конца. Например, если требуется заменить все строки, содержащие столбец `male` с пропущенными значениями, то мы вернем ошибку "имя 'NaN' не определено":

```
# Попытаться заменить значения с NaN
dataframe['Sex'] = dataframe['Sex'].replace('male', NaN)
---------------------------------------------------------------------------
NameError Traceback (most recent call last)
<ipython-input-7-5682d714f87d> in <module>()
1 # Attempt to replace values with NaN
----> 2 dataframe['Sex'] = dataframe['Sex'].replace('male', NaN)
NameError: name 'NaN' is not defined
---------------------------------------------------------------------------
```

Для того чтобы иметь полную функциональность с `NaN`, нам нужно сначала импортировать библиотеку NumPy:

```
# Загрузить библиотеку
import numpy as np

# Заменить значения с NaN
dataframe['Sex'] = dataframe['Sex'].replace('male', np.nan)
```

Нередко для обозначения отсутствующего наблюдения в наборе данных используется некое значение, например NONE, –999 или . (точка). Функция `read_csv` библиотеки pandas имеет параметр, позволяющий указывать значения, используемые для обозначения пропущенных значений:

```
# Загрузить данные, задать пропущенные значения
dataframe = pd.read_csv(url, na_values=[np.nan, 'NONE', -999])
```

3.10. Удаление столбца

Задача

Требуется удалить столбец из фрейма данных.

Решение

Лучший способ удалить столбец — применить метод `drop` с параметром `axis=1` (т. е. с осью столбцов), табл. 3.15:

```
# Загрузить библиотеку
import pandas as pd

# Создать URL-адрес
url = 'https://tinyurl.com/titanic-csv'

# Загрузить данные
dataframe = pd.read_csv(url)

# Удалить столбец
dataframe.drop('Age', axis=1).head(2)
```

Таблица 3.15

	Name	PClass	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	female	One	One
1	Allison, Miss Helen Loraine	1st	female	0	One

В качестве основного аргумента для отбрасывания нескольких столбцов одновременно можно также использовать список имен столбцов (табл. 3.16):

```
# Отбросить столбцы
dataframe.drop(['Age', 'Sex'], axis=1).head(2)
```

Таблица 3.16

	Name	PClass	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	One	One
1	Allison, Miss Helen Loraine	1st	0	One

Если столбец не имеет имени (что иногда бывает), его можно отбросить по индексу столбца с помощью массива `dataframe.columns` (табл. 3.17):

```
# Отбросить столбец
dataframe.drop(dataframe.columns[1], axis=1).head(2)
```

Таблица 3.17

	Name	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	29.0	female	1	1
1	Allison, Miss Helen Loraine	2.0	female	0	1

Обсуждение

Метод `drop` — это идиоматический метод удаления столбца. Альтернативный метод — `del dataframe['Age']`, который работает большую часть времени, но не рекомендуется из-за того, как он вызывается внутри библиотеки pandas (детали описания которого выходят за рамки этой книги).

Рекомендуется взять за привычку никогда не использовать аргумент `inplace=True` библиотеки pandas. Многие методы pandas включают параметр `inplace`, который при равенстве `True` редактирует объект `DataFrame` напрямую. Это может привести к проблемам в более сложных конвейерах обработки данных, потому что в этом случае фреймы данных рассматриваются как мутирующие объекты (которыми они технически являются). Настоятельно рекомендуется рассматривать фреймы данных как немутирующие объекты. Например,

```
# Создать новый фрейм данных
dataframe_name_dropped = dataframe.drop(dataframe.columns[0], axis=1)
```

В этом примере мы не изменяем фрейм данных `dataframe`, а создаем новый, который является измененной версией `dataframe` и называется `dataframe_name_dropped`. Если вы будете рассматривать фреймы данных как немутирующие объекты, то избавите себя от многих головных болей.

3.11. Удаление строки

Задача

Требуется удалить одну или несколько строк из фрейма данных.

Решение

Использовать булево условие для создания нового фрейма данных, исключив строки, которые вы хотите удалить (табл. 3.18):

```
# Загрузить библиотеку
import pandas as pd

# Создать URL-адрес
url = 'https://tinyurl.com/titanic-csv'

# Загрузить данные
dataframe = pd.read_csv(url)

# Удалить строки, показать первые две строки вывода
dataframe[dataframe['Sex'] != 'male'].head(2)
```

Таблица 3.18

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1

Обсуждение

Хотя технически можно использовать метод `drop` (например, чтобы отбросить первые две строки, `df.drop([0, 1], axis=0)`), более практичный метод — просто обернуть булево условие в `df[]`. Причина в том, что мощь условных конструкций можно использовать для удаления либо одной строки, либо (гораздо более вероятно) сразу нескольких строк.

Булевы условия можно использовать, чтобы легко удалять отдельные строки, сопоставляя с уникальным значением (табл. 3.19):

```
# Удалить строку, показать первые две строки вывода
dataframe[dataframe['Name'] != 'Allison, Miss Helen Loraine'].head(2)
```

Таблица 3.19

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
2	Allison, Mr Hudson Joshua Creighton	1st	30.0	male	0	0

Булево условие даже можно использовать для удаления одной строки по ее индексу (табл. 3.20):

```
# Удалить строку, показать первые две строки вывода
dataframe[dataframe.index != 0].head(2)
```

Таблица 3.20

	Name	PClass	Age	Sex	Survived	SexCode
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1
2	Allison, Mr Hudson Joshua Creighton	1st	30.0	male	0	0

3.12. Удаление повторяющихся строк

Задача

Требуется удалить повторяющиеся строки из фрейма данных.

Решение

Использовать метод `drop_duplicates`, но помнить о параметрах (табл. 3.21):

```
# Загрузить библиотеку
import pandas as pd

# Создать URL-адрес
url = 'https://tinyurl.com/titanic-csv'

# Загрузить данные
dataframe = pd.read_csv(url)

# Удалить дубликаты, показать первые две строки вывода
dataframe.drop_duplicates().head(2)
```

Таблица 3.21

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1

Обсуждение

Заинтересованный читатель сразу заметит, что это решение на самом деле строки не отбрасывает:

```
# Показать количество строк
print("Количество строк в исходном фрейме данных:", len(dataframe))
print("Количество строк после дубликации:", len(dataframe.drop_duplicates()))

Количество строк в исходном фрейме данных: 1313
Количество строк после дубликации: 1313
```

Причина в том, что метод `drop_duplicates` по умолчанию отбрасывает только те строки, которые идеально совпадают по всем столбцам. При этом условии каждая строка в нашем фрейме данных `dataframe` на самом деле уникальна. Вместе с тем, для того чтобы выполнить проверку на наличие повторяющихся строк, нередко требуется рассмотреть только подмножество столбцов. Это можно сделать с помощью параметра `subset` (табл. 3.22):

```
# Удалить дубликаты
dataframe.drop_duplicates(subset=['Sex'])
```

Таблица 3.22

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
2	Allison, Mr Hudson Joshua Creighton	1st	30.0	male	0	0

Приглядитесь к приведенному выше результату: мы поручили методу `drop_duplicates` рассматривать как дубликаты любые две строки с одинаковым значением только для столбца `Sex` и их отбрасывать. Теперь у нас остался фрейм данных всего в две строки: один мужчина и одна женщина. Возможно, вы спрашиваете: почему метод `drop_duplicates` решил сохранить именно эти две строки вместо двух других строк? Ответ заключается в том, что метод `drop_duplicates` по умолчанию сохраняет первое появление повторяющейся строки и удаляет остальные. Мы можем управлять этим поведением с помощью параметра `keep` (табл. 3.23):

```
# Удалить дубликаты
dataframe.drop_duplicates(subset=['Sex'], keep='last')
```

Таблица 3.23

	Name	PClass	Age	Sex	Survived	SexCode
1307	Zabour, Miss Tamin	3rd	NaN	female	0	1
1312	Zimmerman, Leo	3rd	29.0	male	0	0

Родственный метод `duplicated` возвращает ряд булевых значений, обозначающих, является ли строка фрейма дубликатом или нет. Это хороший вариант, если вы не хотите просто удалить дубликаты.

3.13. Группирование строк по значениям

Задача

Требуется сгруппировать отдельные строки в соответствии с некоторым общим значением.

Решение

Метод `group` является одним из самых мощных функциональных средств в библиотеке pandas (табл. 3.24):

```
# Загрузить библиотеку
import pandas as pd

# Создать URL-адрес
url = 'https://tinyurl.com/titanic-csv'

# Загрузить данные
dataframe = pd.read_csv(url)

# Сгруппировать строки по значениям столбца 'Sex', вычислить среднее
# каждой группы
dataframe.groupby('Sex').mean()
```

Таблица 3.24

Sex	Age	Survived	SexCode
female	29.396424	0.666667	1.0
male	31.014338	0.166863	0.0

Обсуждение

Метод `groupby` — то средство, после применения которого упорядоченные данные действительно начинают принимать форму. На практике очень часто каждая строка фрейма данных обозначает человека или событие, и требуется их сгруппировать по какому-то критерию, а затем вычислить статистический показатель. Например, можно представить фрейм данных, где каждая строка является отдельной продажей в федеральной сети ресторанов, и нам требуется получить общий объем продаж в расчете на ресторан. Мы можем сделать это, сгруппировав строки по отдельным ресторанам, а затем вычислив сумму каждой группы.

Пользователи, которым метод `groupby` в новинку, нередко пишут следующую ниже строку кода и озадачены тем, что они получают в ответ:

```
# Сгруппировать строки
dataframe.groupby('Sex')

<pandas.core.groupby.DataFrameGroupBy object at 0x10efacf28>
```

Почему метод не вернул что-то более полезное? Причина в том, что метод `groupby` должен идти в паре с некоторой операцией, которую мы хотим применить к каждой группе, в частности с вычислением агрегированного статистического показателя (например, среднего арифметического, медианы, суммы). Когда мы говорим о группировании, мы часто пользуемся сокращением и говорим "группирование по полу", но оно будет не полным. Для того чтобы группирование было полезным, нам нужно собрать нечто в группы, а затем применить функцию к каждой из этих групп:

```
# Сгруппировать строки, подсчитать строки
dataframe.groupby('Survived')['Name'].count()

Survived
0    863
1    450
Name: Name, dtype: int64
```

Обратите внимание на столбец `Name`, указанный после метода `groupby`. Это добавление вызвано тем, что отдельные сводные статистические показатели имеют смысл только для определенных типов данных. Например, в отличие от расчета среднего возраста по полу, который абсолютно оправдан, вычисление суммарного возраста по полу не имеет никакого смысла. В нашем случае мы группируем данные на выживших и погибших пассажиров, а затем подсчитываем количество имен (т. е. пассажиров) в каждой группе.

Мы также можем сгруппировать по первому столбцу, а затем сгруппировать эту группировку по второму столбцу:

```
# Сгруппировать строки, вычислить среднее
dataframe.groupby(['Sex','Survived'])['Age'].mean()

Sex     Survived
female  0           24.901408
        1           30.867143
male    0           32.320780
        1           25.951875
Name: Age, dtype: float64
```

3.14. Группирование строк по времени

Задача

Требуется сгруппировать отдельные строки по периодам времени.

Решение

Для группирования строк фрейма по отрезкам времени использовать метод `resample` (табл. 3.25):

```
# Загрузить библиотеки
import pandas as pd
import numpy as np

# Создать диапазон дат
time_index = pd.date_range('06/06/2017', periods=100000, freq='30S')

# Создать фрейм данных
dataframe = pd.DataFrame(index=time_index)

# Создать столбец случайных значений
dataframe['Sale_Amount'] = np.random.randint(1, 10, 100000)

# Сгруппировать строки по неделе, вычислить сумму за неделю
dataframe.resample('W').sum()
```

Таблица 3.25

	Sale_Amount
2017-06-11	86505
2017-06-18	100489
2017-06-25	101015
2017-07-02	100490
2017-07-09	100288
2017-07-16	10785

Обсуждение

Наш стандартный набор данных *Titanic* не содержит столбца с типом `datetime`, поэтому для этого рецепта мы создали простой фрейм данных, где каждая строка представляет отдельную продажу. Мы знаем дату и время каждой продажи, а также ее сумму в долларах (эти данные нереалистичны, потому что каждая продажа происходит ровно через 30 секунд и является точной суммой в долларах, но ради простоты давайте притворимся).

Сырые данные выглядят следующим образом (табл. 3.26):

```
# Показать три строки
dataframe.head(3)
```

Таблица 3.26

	Sale_Amount
2017-06-06 00:00:00	7
2017-06-06 00:00:30	2
2017-06-06 00:01:00	7

Обратите внимание, что дата и время каждой продажи являются индексом `DataFrame`; это происходит потому, что метод `resample` требует, чтобы индекс имел значения с типом данных `datetime`.

Используя метод `resample`, мы можем сгруппировать строки по широкому массиву периодов времени (смещений), а затем можем вычислить некоторую статистику по каждой группе времени (табл. 3.27, 3.28):

```
# Сгруппировать по двум неделям, вычислить среднее
dataframe.resample('2W').mean()
```

Таблица 3.27

	Sale_Amount
2017-06-11	5.001331
2017-06-25	5.007738
2017-07-09	4.993353
2017-07-23	4.950481

```
# Сгруппировать по месяцу, подсчитать строки
dataframe.resample('M').count()
```

Таблица 3.28

	Sale_Amount
2017-06-30	72000
2017-07-31	28000

Вы, возможно, обратили внимание, что в двух приведенных выше результатах индекс `datetime` является датой, несмотря на то, что мы группируем соответственно по неделям и месяцам. Причина в том, что по умолчанию метод `resample` возвращает метку правого "края" (последнюю метку) временной группы. Мы можем управлять этим поведением с помощью параметра `label` (табл. 3.29):

```
# Сгруппировать по месяцу, подсчитать строки
dataframe.resample('M', label='left').count()
```

Таблица 3.29

	Sale_Amount
2017-05-31	72000
2017-06-30	28000

Дополнительные материалы для чтения

- Перечень псевдонимов для смещений по времени, принятых в библиотеке pandas (**http://bit.ly/2FxfTe4**).

3.15. Обход столбца в цикле

Задача

Требуется выполнить итерацию по каждому элементу в столбце и применить какое-то действие.

Решение

Столбец библиотеки pandas можно рассматривать, как и любую другую последовательность в Python:

```
# Загрузить библиотеку
import pandas as pd

# Создать URL-адрес
url = 'https://tinyurl.com/titanic-csv'

# Загрузить данные
dataframe = pd.read_csv(url)

# Напечатать первые два имени в верхнем регистре
for name in dataframe['Name'][0:2]:
    print(name.upper())
```

```
ALLEN, MISS ELISABETH WALTON
ALLISON, MISS HELEN LORAINE
```

Обсуждение

В дополнение к циклам (часто называемым циклами `for`) мы также можем использовать конструкцию включения в список[2]:

```
# Напечатать первые два имени в верхнем регистре
[name.upper() for name in dataframe['Name'][0:2]]
```

```
['ALLEN, MISS ELISABETH WALTON', 'ALLISON, MISS HELEN LORAINE']
```

Несмотря на соблазн прибегнуть к циклам `for`, более Python-овским решением будет использовать pandas-метод `apply`, который описан в следующем рецепте.

[2] Термин *list comprehension* также переводится не совсем удобным термином *списковое включение*. Дело в том, что в Python помимо включения собственно в список еще существуют конструкции включения в словарь (dictionary comprehension) и включения в множество (set comprehension). — *Прим. перев.*

3.16. Применение функции ко всем элементам в столбце

Задача

Требуется применить некую функцию ко всем элементам в столбце.

Решение

Для применения встроенной или заданной пользователем функции к каждому элементу в столбце использовать метод `apply`:

```
# Загрузить библиотеку
import pandas as pd

# Создать URL-адрес
url = 'https://tinyurl.com/titanic-csv'

# Загрузить данные
dataframe = pd.read_csv(url)

# Создать функцию
def uppercase(x):
    return x.upper()

# Применить функцию, показать две строки
dataframe['Name'].apply(uppercase)[0:2]

0    ALLEN, MISS ELISABETH WALTON
1     ALLISON, MISS HELEN LORAINE
Name: Name, dtype: object
```

Обсуждение

Метод `apply` — это отличный способ выполнить очистку и упорядочение данных. Общепринято писать функцию, которая выполняет некоторые полезные операции (выделение имен и фамилий, преобразования строковых значений в вещественные и т. д.), и затем отображать эту функцию на каждый элемент в столбце.

3.17. Применение функции к группам

Задача

Строки были сгруппировали с помощью метода `groupby`, и теперь требуется применить функцию к каждой группе.

Решение

Объединить методы `groupby` и `apply` (табл. 3.30):

```
# Загрузить библиотеку
import pandas as pd

# Создать URL-адрес
url = 'https://tinyurl.com/titanic-csv'

# Загрузить данные
dataframe = pd.read_csv(url)

# Сгруппировать строки, применить функцию к группам
dataframe.groupby('Sex').apply(lambda x: x.count())
```

Таблица 3.30

Sex	Name	PClass	Age	Sex	Survived	SexCode
female	462	462	288	462	462	462
male	851	851	468	851	851	851

Обсуждение

В рецепте 3.16 был упомянут метод `apply`. Данный метод особенно полезен, когда требуется применить функцию к группам. Объединяя метод `groupby` с методом `apply`, мы можем вычислить собственные статистические показатели или применить любую функцию к каждой группе в отдельности.

3.18. Конкатенация фреймов данных

Задача

Требуется связать последовательно два фрейма данных.

Решение

Для конкатенации фреймов вдоль оси строк использовать метод `concat` с параметром `axis=0` (табл. 3.31):

```
# Загрузить библиотеку
import pandas as pd

# Создать фрейм данных
data_a = {'id': ['1', '2', '3'],
          'first': ['Alex', 'Amy', 'Allen'],
          'last': ['Anderson', 'Ackerman', 'Ali']}
dataframe_a = pd.DataFrame(data_a, columns = ['id', 'first', 'last'])
```

```
# Создать фрейм данных
data_b = {'id': ['4', '5', '6'],
          'first': ['Billy', 'Brian', 'Bran'],
          'last': ['Bonder', 'Black', 'Balwner']}
dataframe_b = pd.DataFrame(data_b, columns = ['id', 'first', 'last'])

# Конкатенировать фреймы данных по строкам
pd.concat([dataframe_a, dataframe_b], axis=0)
```

Таблица 3.31

	Id	first	last
0	1	Alex	Anderson
1	2	Amy	Ackerman
2	3	Allen	Ali
0	4	Billy	Bonder
1	5	Brian	Black
2	6	Bran	Balwner

Для конкатенации вдоль оси столбцов можно использовать `axis=1` (табл. 3.32):

```
# Конкатенировать фреймы данных по столбцам
pd.concat([dataframe_a, dataframe_b], axis=1)
```

Таблица 3.32

	id	first	last	id	first	last
0	1	Alex	Anderson	4	Billy	Bonder
1	2	Amy	Ackerman	5	Brian	Black
2	3	Allen	Ali	6	Bran	Balwner

Обсуждение

Термин "конкатенация" имеет другой смысл вне сферы информатики и программирования, поэтому, если вы не слышали его раньше, не волнуйтесь. Неформальное определение термина *"конкатенировать"* — склеить два объекта вместе. В нашем решении мы склеили два небольших фрейма данных с помощью параметра `axis`, который указывает на то, хотим ли мы сложить два фрейма данных друг на друга либо разместить их рядом.

В качестве альтернативы добавления новой строки во фрейм данных можно применить метод `append` (табл. 3.33):

```
# Создать строку
row = pd.Series([10, 'Chris', 'Chillon'], index=['id', 'first', 'last'])

# Добавить строку в конец
dataframe_a.append(row, ignore_index=True)
```

Таблица 3.33

	id	first	last
0	1	Alex	Anderson
1	2	Amy	Ackerman
2	3	Allen	Ali
3	10	Chris	Chillon

3.19. Слияние фреймов данных

Задача

Требуется выполнить слияние двух фреймов данных.

Решение

Для того чтобы применить внутреннее соединение, использовать метод `merge` с параметром `on`, задающим столбец, по которому происходит слияние (табл. 3.34):

```
# Загрузить библиотеку
import pandas as pd

# Создать фрейм данных
employee_data = {'employee_id': ['1', '2', '3', '4'],
                 'name': ['Amy Jones', 'Allen Keys', 'Alice Bees',
                          'Tim Horton']}
dataframe_employees = pd.DataFrame(employee_data, columns =
                                   ['employee_id', 'name'])

# Создать фрейм данных
sales_data = {'employee_id': ['3', '4', '5', '6'],
              'total_sales': [23456, 2512, 2345, 1455]}
dataframe_sales = pd.DataFrame(sales_data, columns = ['employee_id',
                                                      'total_sales'])

# Выполнить слияние фреймов данных
pd.merge(dataframe_employees, dataframe_sales, on='employee_id')
```

Таблица 3.34

	employee_id	name	total_sales
0	3	Alice Bees	23456
1	4	Tim Horton	2512

Метод `merge` по умолчанию выполняет операцию внутреннего соединения. Если мы хотим сделать внешнее соединение, можем указать это при помощи параметра `how` (табл. 3.35):

```
# Выполнить слияние фреймов данных
pd.merge(dataframe_employees, dataframe_sales,
         on='employee_id', how='outer')
```

Таблица 3.35

	employee_id	name	total_sales
0	1	Amy Jones	NaN
1	2	Allen Keys	NaN
2	3	Alice Bees	23456.0
3	4	Tim Horton	2512.0
4	5	NaN	2345.0
5	6	NaN	1455.0

Этот же параметр можно использовать для указания левого и правого соединений (табл. 3.36):

```
# Выполнить слияние фреймов данных
pd.merge(dataframe_employees, dataframe_sales,
         on='employee_id', how='left')
```

Таблица 3.36

	employee_id	name	total_sales
0	1	Amy Jones	NaN
1	2	Allen Keys	NaN
2	3	Alice Bees	23456.0
3	4	Tim Horton	2512.0

В каждом фрейме данных мы также можем указать имя столбца, по которому выполнять слияние (табл. 3.37):

```
# Выполнить слияние фреймов данных
pd.merge(dataframe_employees,
         dataframe_sales,
         left_on='employee_id',
         right_on='employee_id')
```

Таблица 3.37

	employee_id	name	total_sales
0	3	Alice Bees	23456
1	4	Tim Horton	2512

Если вместо слияния по двум столбцам мы хотим выполнить слияние по индексам каждого фрейма данных, мы можем заменить параметры `left_on` и `right_on` на `right_index=True` и `left_index=True`.

Обсуждение

Зачастую данные, которые нам нужно использовать, имеют сложный вид; они не всегда бывают цельными. Вместо этого в реальном мире мы обычно сталкиваемся с разрозненными наборами данных из многочисленных запросов к базам данных или файлам. Для того чтобы получить все эти данные в одном месте, мы можем загрузить каждый запрос к данным или файл данных в отдельные фреймы данных, а затем слить их в один фрейм.

Этот процесс может быть знаком всем, кто использовал популярный язык SQL для выполнения операций слияния (называемых *соединениями*). В то время как точные параметры, используемые в библиотеке pandas, будут различаться, они следуют тем же общим шаблонам, используемым в других языках программирования и инструментах.

В любой операции слияния необходимо указать три аспекта. Во-первых, мы должны указать два фрейма данных, которые хотим слить воедино. В нашем решении мы назвали их `dataframe_employees` и `dataframe_sales`. Во-вторых, мы должны указать имя (имена) столбцов, по которым выполнять слияние, т. е. столбцы, значения которых являются общими для двух фреймов данных. Например, в нашем решении оба фрейма данных имеют столбец `employee_id`. Для того чтобы объединить два фрейма данных, мы будем сопоставлять друг с другом значения в столбце `employee_id` каждого фрейма данных. Если эти два столбца используют одинаковое имя, мы можем применить параметр `on`. Однако, если у них разные имена, мы можем использовать параметры `left_on` и `right_on`.

Что такое левый и правый фреймы данных? Все просто. Левый фрейм данных — это фрейм, который мы указали в методе `merge` первым, а правый фрейм — вторым. Эти понятия будут использованы в следующих ниже наборах параметров, которые нам понадобятся.

Последний аспект, и наиболее трудный для некоторых людей, — это тип операции слияния, которую требуется выполнить. Он задается параметром `how`. Метод `merge` поддерживает четыре основных типа соединений:

- *внутреннее* — вернуть только те строки, которые совпадают в обоих фреймах (например, вернуть любую строку, в которой значение `employee_id` появляется одновременно в `dataframe_employees` и в `dataframe_sales`);

- *внешнее* — вернуть все строки в обоих фреймах; если строка существует в одном фрейме, но отсутствует в другом, то пропущенные значения заполнить значениями `NaN` (например, вернуть все строки и в `employee_id`, и в `dataframe_sales`);
- *левое* — вернуть все строки из левого фрейма, но только те строки из правого фрейма, которые совпали с левым фреймом; пропущенные значения заполнить значениями `NaN` (например, вернуть все строки из `dataframe_employees`, но только те строки из `dataframe_sales`, которые имеют значение для `employee_id`, появляющееся в `dataframe_employees`);
- *правое* — вернуть все строки из правого фрейма, но только те строки из левого фрейма, которые совпали с правым фреймом; пропущенные значения заполнить значениями `NaN` (например, вернуть все строки из `dataframe_sales`, но только те строки из `dataframe_employees`, которые имеют значение для `employee_id`, появляющееся в `dataframe_sales`).

Если вы не поняли всего этого прямо сейчас, то рекомендуется поэкспериментировать с параметром `how` в своем программном коде и посмотреть, как он влияет на результаты, возвращаемые методом `merge`.

Дополнительные материалы для чтения

- Визуальное объяснение соединений SQL, блог-пост (**http://bit.ly/2Fxgcpe**).
- Документация библиотеки pandas по операции слияния (**http://bit.ly/2Fuo4rH**).

Работа с числовыми данными

Введение

Количественные данные что-то измеряют — будь то размер класса, ежемесячные продажи или оценки учащихся. Естественным способом представления этих величин является численное (например, 29 студентов, $529 392 продаж). В этой главе мы рассмотрим многочисленные стратегии преобразования сырых числовых данных в признаки, целенаправленно формируемые для машинно-обучающихся алгоритмов.

4.1. Шкалирование признака

Задача

Требуется прошкалировать числовой признак в диапазон между двумя значениями.

Решение

Для шкалирования массива признаков использовать класс `MinMaxScaler` библиотеки scikit-learn:

```
# Загрузить библиотеки
import numpy as np
from sklearn import preprocessing

# Создать признак
feature = np.array([[-500.5],
                    [-100.1],
                    [0],
                    [100.1],
                    [900.9]])

# Создать шкалировщик
minmax_scale = preprocessing.MinMaxScaler(feature_range=(0, 1))

# Прошкалировать признак
scaled_feature = minmax_scale.fit_transform(feature)
```

```
# Показать прошкалированный признак
scaled_feature
```

```
array([[0.        ],
       [0.28571429],
       [0.35714286],
       [0.42857143],
       [1.        ]])
```

Обсуждение

Шкалирование — это общепринятая задача предобработки в машинном самообучении. Многие алгоритмы, описываемые далее в этой книге, исходят из того, что все признаки находятся на одинаковой шкале, как правило, от 0 до 1 или от −1 до 1. Существует целый ряд методов шкалирования, но один из самых простых называется *минимаксным шкалированием*. В минимаксном шкалировании минимальное и максимальное значения признака используются для шкалирования значений в пределах диапазона. В частности, минимакс вычисляется следующим образом:

$$x_i' = \frac{x_i - \min(\mathbf{x})}{\max(\mathbf{x}) - \min(\mathbf{x})},$$

где $\mathbf{x}$ — это вектор признака; x_i — отдельный элемент признака x; x_i' — прошкалированный элемент.

В нашем примере из выведенного массива видно, что признак был успешно прошкалирован в диапазон от 0 до 1:

```
array([[0.        ],
       [0.28571429],
       [0.35714286],
       [0.42857143],
       [1.        ]])
```

Класс библиотеки scikit-learn `MinMaxScaler` предлагает два варианта шкалирования признака. Один вариант — использовать метод `fit` для вычисления минимального и максимального значений признака, а затем применить метод `transform` для шкалирования признака. Второй вариант — вызвать метод `fit_transform` для выполнения обеих операций одновременно. Между этими двумя вариантами нет никакой математической разницы, но иногда есть практическая выгода в том, чтобы разделить эти операции, потому что это позволяет применять одно и то же преобразование к разным наборам данных.

Дополнительные материалы для чтения

- "Шкалирование признаков", Википедия (**http://bit.ly/2Fuug2Z**).
- Себастьян Рашка "О шкалировании и нормализации признаков" (**http://bit.ly/2FwwRcM**).

4.2. Стандартизация признака

Задача

Требуется преобразовать признак, чтобы он имел среднее значение 0 и стандартное отклонение 1.

Решение

Класс `StandardScaler` библиотеки scikit-learn выполняет оба преобразования:

```
# Загрузить библиотеки
import numpy as np
from sklearn import preprocessing

# Создать признак
x = np.array([[-1000.1],
              [-200.2],
              [500.5],
              [600.6],
              [9000.9]])

# Создать шкалировщик
scaler = preprocessing.StandardScaler()

# Преобразовать признак
standardized = scaler.fit_transform(x)

# Показать признак
standardized

array([[-0.76058269],
       [-0.54177196],
       [-0.35009716],
       [-0.32271504],
       [ 1.97516685]])
```

Обсуждение

Распространенной альтернативой минимаксному шкалированию, описанному в рецепте 4.1, является шкалирование признаков, при котором они должны быть приближенно стандартно распределены. Для этого мы используем стандартизацию, в ходе которой данные преобразуются таким образом, что они имеют среднее значение $\bar{x} = 0$ и стандартное отклонение $\sigma = 1$. В частности, каждый элемент в признаке преобразуется таким образом, чтобы:

$$x_i' = \frac{x_i - x}{\sigma},$$

где x_i' — наша стандартизированная форма x_i. Преобразованный признак представляет собой количество стандартных отклонений, на которое исходное значение отстоит от среднего значения признака (так называемую *z-оценку* в статистике).

В машинном самообучении стандартизация является распространенным методом шкалирования с целью предобработки и по моему опыту используется больше, чем минимаксное шкалирование. Однако вариант шкалирования зависит от обучающегося алгоритма. Например, анализ главных компонент часто работает лучше с использованием стандартизации, в то время как для нейронных сетей часто рекомендуется минимаксное шкалирование (оба алгоритма обсуждаются далее в этой книге). В качестве общего правила, если у вас нет конкретной причины использовать альтернативу, рекомендуется по умолчанию применять стандартизацию.

Мы можем увидеть эффект стандартизации, обратившись к среднему значению и стандартному отклонению результата нашего решения:

```
# Напечатать среднее значение и стандартное отклонение
print("Среднее:", round(standardized.mean()))
print("Стандартное отклонение:", standardized.std())

Среднее: 0.0
Стандартное отклонение: 1.0
```

Если наши данные имеют значительные выбросы, это может негативно повлиять на стандартизацию, сказываясь на среднем значении и дисперсии признака. В таком случае вместо этого часто бывает полезно прошкалировать признак, используя медиану и квартильный размах. В scikit-learn мы делаем это с помощью класса `RobustScaler`, реализующего метод робастного шкалирования:

```
# Создать шкалировщик
robust_scaler = preprocessing.RobustScaler()

# Преобразовать признак
robust_scaler.fit_transform(x)

array([[-1.87387612],
       [-0.875     ],
       [ 0.        ],
       [ 0.125     ],
       [10.61488511]])
```

4.3. Нормализация наблюдений

Задача

Требуется прошкалировать значения признаков в наблюдениях для получения единичной нормы (общей длиной 1).

Решение

Использовать класс `Normalizer` с аргументом `norm`:

```
# Загрузить библиотеки
import numpy as np
from sklearn.preprocessing import Normalizer

# Создать матрицу признаков
features = np.array([[0.5, 0.5],
                     [1.1, 3.4],
                     [1.5, 20.2],
                     [1.63, 34.4],
                     [10.9, 3.3]])

# Создать нормализатор
normalizer = Normalizer(norm="l2")

# Преобразовать матрицу признаков
normalizer.transform(features)

array([[0.70710678, 0.70710678],
       [0.30782029, 0.95144452],
       [0.07405353, 0.99725427],
       [0.04733062, 0.99887928],
       [0.95709822, 0.28976368]])
```

Обсуждение

Многие методы шкалирования (например, минимаксное шкалирование и стандартизация) работают с признаками; однако мы также можем шкалировать отдельные наблюдения. Класс `Normalizer` шкалирует значения в отдельных наблюдениях, приводя их к единичной норме (сумма их длин равна 1). Этот тип шкалирования часто используют, когда имеется много эквивалентных признаков (например, в классификации текста, когда каждое слово или группа n-слов является признаком).

Класс `Normalizer` предоставляет три варианта нормы, при этом евклидова норма (нередко именуемая L^2-нормой) является аргументом по умолчанию:

$$\|x\|_2 = \sqrt{x_1^2 + x_2^2 + ... + x_n^2},$$

где x — это отдельное наблюдение; x_n — значение этого наблюдения для n-го признака.

```
# Преобразовать матрицу признаков
features_l2_norm = Normalizer(norm="l2").transform(features)

# Показать матрицу признаков
features_l2_norm
```

```
array([[0.70710678, 0.70710678],
       [0.30782029, 0.95144452],
       [0.07405353, 0.99725427],
       [0.04733062, 0.99887928],
       [0.95709822, 0.28976368]])
```

В качестве альтернативы можно указать манхэттенскую норму (L^1):

$$\|x\|_1 = \sum_{i=1}^{n} |x_i|.$$

```
# Преобразовать матрицу признаков
features_l1_norm = Normalizer(norm="l1").transform(features)

# Показать матрицу признаков
features_l1_norm
```

```
array([[0.5       , 0.5       ],
       [0.24444444, 0.75555556],
       [0.06912442, 0.93087558],
       [0.04524008, 0.95475992],
       [0.76760563, 0.23239437]])
```

Интуитивно норму L^2 можно рассматривать как расстояние между двумя точками в Нью-Йорке, пролетаемое птицей (т. е. по прямой), в то время как L^1 можно воспринимать как расстояние пройденного человеком пути по улицам (пройти один квартал на север, один квартал на восток, один квартал на север, один квартал на восток и т. д.), поэтому ее называют "манхэттенской нормой" или " таксомоторной нормой".

На практике обратите внимание, что `norm='l1'` шкалирует значения наблюдений таким образом, что они в сумме дают 1. Иногда такая сумма может быть желательным качеством:

```
# Напечатать сумму
print("Сумма значений первого наблюдения:",
      features_l1_norm[0, 0] + features_l1_norm[0, 1])
```

```
Сумма значений первого наблюдения: 1.0
```

4.4. Генерирование полиномиальных и взаимодействующих признаков

Задача

Требуется создать полиномиальные и взаимодействующие признаки.

Решение

Несмотря на то что некоторые специалисты выбирают создание полиномиальных и взаимодействующих признаков вручную, библиотека scikit-learn предлагает встроенный метод:

```
# Загрузить библиотеки
import numpy as np
from sklearn.preprocessing import PolynomialFeatures

# Создать матрицу признаков
features = np.array([[2, 3],
                     [2, 3],
                     [2, 3]])

# Создать объект PolynomialFeatures
polynomial_interaction = PolynomialFeatures(degree=2, include_bias=False)

# Создать полиномиальные признаки
polynomial_interaction.fit_transform(features)

array([[2., 3., 4., 6., 9.],
       [2., 3., 4., 6., 9.],
       [2., 3., 4., 6., 9.]])
```

Параметр `degree` определяет максимальный порядок полинома. Например, `degree=2` создаст новые признаки, возведенные во вторую степень:

$$x_1, x_2, x_1^2, x_2^2,$$

в то время как `degree=3` создаст новые признаки, возведенные во вторую и третью степени:

$$x_1, x_2, x_1^2, x_2^2, x_1^3, x_2^3.$$

Более того, по умолчанию метод, реализованный в классе `PolynomialFeatures`, включает в себя признаки взаимодействия:

$$x_1, x_2.$$

Мы можем ограничить создаваемые признаки только признаками взаимодействия, установив для `interaction_only` значение `True`:

```
interaction = PolynomialFeatures(degree=2,
                                 interaction_only=True, include_bias=False)
interaction.fit_transform(features)

array([[2., 3., 6.],
       [2., 3., 6.],
       [2., 3., 6.]])
```

Обсуждение

Полиномиальные признаки часто создаются, когда мы предполагаем, что существует нелинейная связь между признаками и целью. Например, мы можем подозревать, что влияние возраста на вероятность наличия серьезных заболеваний не является постоянным с течением времени, а возрастает по мере увеличения возраста. Мы можем закодировать этот неконстантный эффект в признаке x, генерируя формы этого признака более высокого порядка (x^2, x^3 и т. д.).

Кроме того, нередко мы сталкиваемся с ситуациями, когда эффект одного признака зависит от еще одного признака. Простым примером была бы попытка предсказать, является ли наш кофе сладким, и при этом у нас всего два признака: 1) был ли кофе перемешан; 2) добавляли ли мы сахар? Отдельно каждый признак сладость кофе не предсказывает, но сочетание признаков это делает. То есть кофе будет сладким, только если в кофе есть сахар, и он перемешан. Влияние каждого признака на цель (сладость) зависит друг от друга. Мы можем кодировать эту связь, включив взаимодействующий признак, который является произведением отдельных признаков.

4.5. Преобразование признаков

Задача

Требуется выполнить собственное преобразование одного или более признаков.

Решение

Использовать класс `FunctionTransformer` библиотеки scikit-learn для применения функции к набору признаков:

```
# Загрузить библиотеки
import numpy as np
from sklearn.preprocessing import FunctionTransformer

# Создать матрицу признаков
features = np.array([[2, 3],
                     [2, 3],
                     [2, 3]])

# Определить простую функцию
def add_ten(x):
    return x + 10

# Создать преобразователь
ten_transformer = FunctionTransformer(add_ten)

# Преобразовать матрицу признаков
ten_transformer.transform(features)
```

```
array([[12, 13],
       [12, 13],
       [12, 13]])
```

Такое же преобразование можно создать в библиотеке pandas с помощью метода `apply` (табл. 4.1):

```
# Загрузить библиотеку
import pandas as pd

# Создать фрейм данных
df = pd.DataFrame(features, columns=["признак_1", "признак_2"])

# Применить функцию
df.apply(add_ten)
```

Таблица 4.1

	признак_1	признак_2
0	12	13
1	12	13
2	12	13

Обсуждение

Очень часто возникает необходимость выполнить некоторые собственные преобразования для одного или более признаков. Например, мы можем создать признак, который является натуральным логарифмом значений другого признака. Мы можем сделать это, создав функцию, а затем отобразив ее на признаки с помощью либо класса `FunctionTransformer` библиотеки scikit-learn, либо метода `apply` библиотеки pandas. В этом решении мы создали очень простую функцию `add_ten`, которая прибавляла 10 к каждому входу, но нет причин, по которым мы не могли бы определить гораздо более сложную функцию.

4.6. Обнаружение выбросов

Задача

Требуется идентифицировать предельные значения.

Решение

Обнаружение выбросов, к сожалению, это больше искусство, чем наука. Вместе с тем, распространенным методом является принятие допущения о том, что данные нормально распределены. Основываясь на этом допущении, мы можем "рисовать"

эллипс вокруг данных, классифицируя любое наблюдение внутри эллипса как не выброс (помеченный как 1) и любое наблюдение за пределами эллипса как выброс (помеченный как −1):

```
# Загрузить библиотеки
import numpy as np
from sklearn.covariance import EllipticEnvelope
from sklearn.datasets import make_blobs

# Создать симулированные данные
features, _ = make_blobs(n_samples = 10,
                         n_features = 2,
                         centers = 1,
                         random_state = 1)

# Заменить значения первого наблюдения предельными значениями
features[0,0] = 10000
features[0,1] = 10000

# Создать детектор
outlier_detector = EllipticEnvelope(contamination=.1)

# Выполнить подгонку детектора
outlier_detector.fit(features)

# Предсказать выбросы
outlier_detector.predict(features)

array([-1,  1,  1,  1,  1,  1,  1,  1,  1,  1])
```

Основным ограничением этого подхода является необходимость указания параметра загрязнения `contamination`, который представляет собой долю наблюдений, являющихся выбросами, — значение, которое мы не знаем. Подумайте о загрязнении как о нашей оценке чистоты наших данных. Если мы ожидаем, что наши данные будут иметь несколько выбросов, мы можем задать параметр `contamination` с каким-нибудь небольшим значением. Однако, если мы считаем, что данные, скорее всего, будут иметь выбросы, мы можем установить для него более высокое значение.

Вместо того чтобы смотреть на наблюдения в целом, мы можем взглянуть на отдельные признаки и идентифицировать в этих признаках предельные значения, используя межквартильный размах (МКР, IQR):

```
# Создать один признак
feature = features[:,0]

# Создать функцию, которая возвращает индекс выбросов
def indicies_of_outliers(x):
```

```
    q1, q3 = np.percentile(x, [25, 75])
    iqr = q3 - q1
    lower_bound = q1 - (iqr * 1.5)
    upper_bound = q3 + (iqr * 1.5)
    return np.where((x > upper_bound) | (x < lower_bound))

# Выполнить функцию
indicies_of_outliers(feature)

(array([0]),)
```

Межквартильный размах — это разница между первым и третьим квартилями набора данных. МКР можно представить, как разброс основной части данных, где выбросы — это наблюдения, отдаленные от основной части данных. Выбросы обычно определяются как любое значение, которое на 1.5 МКР меньше первого квартиля или на 1.5 МКР больше третьего квартиля.

Обсуждение

Единого наилучшего метода обнаружения выбросов не существует. Вместо этого у нас есть коллекция методов, все со своими преимуществами и недостатками. Наша лучшая стратегия часто состоит в том, чтобы пытаться использовать несколько методов (например, и `EllipticEnvelope`, и обнаружение на основе МКР) и смотреть на результаты в целом.

Если это вообще возможно, мы должны взглянуть на наблюдения, которые мы обнаруживаем как выбросы, и попытаться их понять. Например, возьмем набор данных о домах, в котором одним из признаков является количество комнат. Является ли выброс со 100 комнатами действительно домом или это на самом деле отель, который был неправильно классифицирован?

Дополнительные материалы для чтения

- "Три способа обнаружения выбросов", блог-пост (и исходный код функции МКР, используемой в этом рецепте; **http://bit.ly/2FzMC2k**).

4.7. Обработка выбросов

Задача

Имеются выбросы.

Решение

Для обработки выбросов, как правило, можно использовать три стратегии. Во-первых, мы можем их отбросить (табл. 4.2):

```
# Загрузить библиотеку
import pandas as pd

# Создать фрейм данных
houses = pd.DataFrame()
houses['Цена'] = [534433, 392333, 293222, 4322032]
houses['Ванные'] = [2, 3.5, 2, 116]
houses['Кв_футы'] = [1500, 2500, 1500, 48000]

# Отфильтровать наблюдения
houses[houses['Ванные'] < 20]
```

Таблица 4.2

	Цена	Ванные	Кв_футы
0	534433	2.0	1500
1	392333	3.5	2500
2	293222	2.0	1500

Во-вторых, мы можем пометить их как выбросы и включить их в качестве признака (табл. 4.3):

```
# Загрузить библиотеку
import numpy as np

# Создать признак на основе булева условия
houses["Выброс"] = np.where(houses["Ванные"] < 20, 0, 1)

# Показать данные
houses
```

Таблица 4.3

	Цена	Ванные	Кв_футы	Выброс
0	534433	2.0	1500	0
1	392333	3.5	2500	0
2	293222	2.0	1500	0
3	4322032	116.0	48000	1

Наконец, мы можем преобразовать признак, чтобы ослабить эффект выброса (табл. 4.4):

```
# Взять логарифм признака
houses["Логарифм_кв_футов"] = [np.log(x) for x in houses["Кв_футы"]]

# Показать данные
houses
```

Таблица 4.4

	Цена	Ванные	Кв_футы	Выброс	Логарифм_кв_футов
0	534433	2.0	1500	0	7.313220
1	392333	3.5	2500	0	7.824046
2	293222	2.0	1500	0	7.313220
3	4322032	116.0	48000	1	10.778956

Обсуждение

Подобно обнаружению выбросов, раз и навсегда заведенного правила обработки выбросов не существует. Стратегия должна основываться на двух аспектах. Во-первых, мы должны понять, что делает данные выбросами. Если мы считаем, что это ошибки в данных, например, из-за сломанного датчика или неверного значения, то мы можем исключить это наблюдение или заменить значения выбросов на `NaN`, т. к. этим значениям нельзя верить. Однако если мы считаем, что выбросы являются подлинными предельными значениями (например, дом [особняк] с 200 ванными комнатами), то более уместной будет маркировка их как выбросы или преобразование их значений.

Во-вторых, стратегия обработки выбросов должна основываться на нашей цели машинного самообучения. Например, если мы хотим предсказать цены на жилье на основе признаков дома, то было бы разумно предположить, что цена на особняки с более чем 100 ванными комнатами обусловлена другой динамикой, чем обычные семейные дома. Кроме того, если мы готовим модель для использования в качестве части веб-приложения онлайнового кредитования на жилье, то можно предположить, что среди клиентов не будет миллиардеров, желающих купить особняк.

Итак, что же нам делать, если имеются выбросы? Подумайте о том, почему они являются выбросами, держите в уме конечную цель обучения и, самое главное, помните, что непринятие решения об устранении выбросов само по себе является решением с последствиями.

Один дополнительный момент: если в данных имеются выбросы, то стандартизация может оказаться неуместной, потому что среднее значение и дисперсия сильно зависят от выбросов. В этом случае следует использовать более робастный метод шкалирования против выбросов, наподобие реализованного в классе `RobustScaler`.

Дополнительные материалы для чтения

- Документация по робастному шкалировщику `RobustScaler` **(http://bit.ly/2DcgyNT)**.

4.8. Дискретизация признаков

Задача

Дан числовой признак, и требуется разбить его на дискретные корзины.

Решение

В зависимости от того как мы хотим разбить данные, существует два метода, которые можно применить. Во-первых, можно бинаризовать признак в соответствии с некоторым порогом, т. е. перевести его значения в двоичные:

```
# Загрузить библиотеки
import numpy as np
from sklearn.preprocessing import Binarizer

# Создать признак
age = np.array([[6],
                [12],
                [20],
                [36],
                [65]])

# Создать бинаризатор
binarizer = Binarizer(18)

# Преобразовать признак
binarizer.fit_transform(age)

array([[0],
       [0],
       [1],
       [1],
       [1]])
```

Во-вторых, мы можем разбить числовые признаки в соответствии с несколькими порогами:

```
# Разнести признак по корзинам
np.digitize(age, bins=[20,30,64])

array([[0],
       [0],
       [1],
       [2],
       [3]])
```

Обратите внимание, что аргументы для параметра `bins` обозначают левый край каждой корзины. Например, аргумент 20 не включает элемент со значением 20,

только два значения меньше 20. Это поведение можно переключить, задав для параметра `right` значение `True`:

```
# Разнести признак по корзинам
np.digitize(age, bins=[20,30,64], right=True)
```

```
array([[0],
       [0],
       [0],
       [2],
       [3]])
```

Обсуждение

Дискретизация может быть плодотворной стратегией, когда у нас есть основания полагать, что числовой признак должен вести себя больше как категориальный признак. Например, можно полагать, что существует очень небольшая разница в привычках расходования 19- и 20-летних, но значительная разница между 20- и 21-летними (возраст в Соединенных Штатах, когда молодые люди могут употреблять алкоголь). В этом примере было бы полезно соотнести людей в наших данных с теми категориями, когда можно употреблять алкоголь и когда нельзя этого делать. Аналогичным образом, в других случаях может быть полезно дискретизировать наши данные в три или более корзин.

В нашем решении мы увидели два метода дискретизации — класс `Binarizer` библиотеки scikit-learn для двух корзин и функцию `digitize` библиотеки NumPy для трех корзин или больше. Вместе с тем функция `digitize` также может использоваться для бинаризации признаков, как `Binarizer`, если задать только один порог:

```
# Разнести признак по корзинам
np.digitize(age, bins=[18])
```

```
array([[0],
       [0],
       [1],
       [1],
       [1]])
```

Дополнительные материалы для чтения

- Документация по функции digitize (**http://bit.ly/2HSciFP**).

4.9. Группирование наблюдений с помощью кластеризации

Задача

Требуется сгруппировать наблюдения так, чтобы похожие наблюдения находились в одной группе.

Решение

Если вы знаете, что имеется *k* групп, то можно применить кластеризацию по методу *k* средних для группирования похожих наблюдений и вывода нового признака, содержащего групповую принадлежность каждого наблюдения (табл. 4.5):

```
# Загрузить библиотеки
import pandas as pd
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans

# Создать матрицу симулированных признаков
features, _ = make_blobs(n_samples = 50,
                         n_features = 2,
                         centers = 3,
                         random_state = 1)

# Создать фрейм данных
dataframe = pd.DataFrame(features, columns=["признак_1", "признак_2"])

# Создать кластеризатор по методу k средних
clusterer = KMeans(3, random_state=0)

# Выполнить подгонку кластеризатора
clusterer.fit(features)

# Предсказать значения
dataframe["группа"] = clusterer.predict(features)

# Взглянуть на первые несколько наблюдений
dataframe.head(5)
```

Таблица 4.5

	признак_1	признак_2	группа
0	–9.877554	–3.336145	0
1	–7.287210	–8.353986	2
2	–6.943061	–7.023744	2
3	–7.440167	–8.791959	2
4	–6.641388	–8.075888	2

Обсуждение

Мы немного забегаем вперед и позже в этой книге больше углубимся в кластеризующие алгоритмы. Однако здесь стоить обратить внимание на то, что кластеризация может использоваться в качестве шага предобработки. В частности, мы исполь-

зовали для кластеризации наблюдений в группы неконтролируемые (без учителя) обучающиеся алгоритмы, такие как метод k средних. Конечным результатом служит категориальный признак, в котором аналогичные наблюдения являются членами одной группы.

Не волнуйтесь, если вы не поняли всего этого прямо сейчас: просто возьмите на заметку идею о том, что кластеризация может использоваться в предобработке. И если вы действительно не можете ждать, то прямо сейчас смело перелистайте страницы до *главы 19*.

4.10. Удаление наблюдений с пропущенными значениями

Задача

Требуется удалить наблюдения, содержащие пропущенные значения.

Решение

Удаление наблюдений с пропущенными значениями выполняется легко с помощью умной конструкции в библиотеке NumPy:

```
# Загрузить библиотеку
import numpy as np

# Создать матрицу признаков
features = np.array([[1.1, 11.1],
                     [2.2, 22.2],
                     [3.3, 33.3],
                     [4.4, 44.4],
                     [np.nan, 55]])

# Оставить только те наблюдения, которые не (помечены ~) пропущены
features[~np.isnan(features).any(axis=1)]

array([[ 1.1, 11.1],
       [ 2.2, 22.2],
       [ 3.3, 33.3],
       [ 4.4, 44.4]])
```

В качестве альтернативы можно удалить наблюдения, содержащие пропущенные значения, с помощью библиотеки pandas (табл. 4.6):

```
# Загрузить библиотеку
import pandas as pd

# Загрузить данные
dataframe = pd.DataFrame(features, columns=["признак_1", "признак_2"])
```

```
# Удалить наблюдения с отсутствующими значениями
dataframe.dropna()
```

Таблица 4.6

	признак_1	признак_2
0	1.1	11.1
1	2.2	22.2
2	3.3	33.3
3	4.4	44.4

Обсуждение

Большинство машинно-обучающихся алгоритмов не могут обрабатывать пропущенные значения в массивах целей и признаков. По этой причине мы не можем игнорировать пропущенные значения в наших данных и должны решить эту проблему во время предобработки.

Самым простым решением является удаление каждого наблюдения, содержащего одно или несколько пропущенных значений. Эта задача быстро и легко выполняется с помощью библиотек NumPy или pandas.

Вместе с тем мы должны всячески воздерживаться от удаления наблюдений с отсутствующими значениями. Их удаление является крайним средством, поскольку наш алгоритм теряет доступ к полезной информации, содержащейся в непропущенных значениях наблюдения.

Не менее важно и то, что в зависимости от причины появления пропущенных значений удаление наблюдений может привести к смещению наших данных. Существует три типа пропущенных данных.

- *Пропущены совершенно случайно* (Missing Completely At Random, MCAR). Вероятность того, что значение пропущено, ни от чего не зависит. Например, участник викторины, прежде чем ответить на вопрос, бросает кубик: если выпадает шесть, то он пропускает этот вопрос.
- *Пропущены случайно* (Missing At Random, MAR). Вероятность того, что значение пропущено, не является полностью случайной, но зависит от информации, зафиксированной в других признаках. Например, в опросе общественного мнения ставится вопрос о гендерной идентичности и годовой заработной плате, и женщины чаще пропускают вопрос о заработной плате; однако отсутствие их реакции зависит только от информации, которую мы зафиксировали в нашем признаке гендерной идентичности.
- *Пропущены не случайно* (Missing Not At Random, MNAR). Вероятность того, что значение пропущено, не случайна и зависит от информации, не зафиксированной в наших признаках. Например, в опросе общественного мнения ставится вопрос о гендерной идентичности, и женщины чаще пропускают вопрос о зарплате, и у нас нет признака гендерной идентичности в наших данных.

Иногда допустимо удалять наблюдения, если они пропущены совершенно случайно (MCAR) или просто случайно (MAR). Однако если значение пропущено не случайно (MNAR), то факт, что значение пропущено, сам по себе является информацией. Удаление наблюдений MNAR может привнести в наши данные смещение, потому что мы удаляем наблюдения, порожденные некоторым ненаблюдаемым систематическим эффектом.

Дополнительные материалы для чтения

- "Идентификация трех типов пропущенных значений", блог-пост (**http://bit.ly/2Fto4bx**).
- "Импутация[1] пропущенных значений", глава 25 из книги Андрю Гелмана "Анализ данных: использование регрессионных и многомерных/иерархических моделей" (Gelman A. Data Analysis: Using Regression and Multilevel/Hierarchical Models. — URL: http://bit.ly/2FAkKLI).

4.11. Импутация пропущенных значений

Задача

В ваших данных имеются пропущенные значения, и требуется заполнить или предсказать их значения.

Решение

Если объем данных небольшой, то предсказать пропущенные значения с помощью *k* ближайших соседей (KNN)[2]:

```
# Загрузить библиотеки
import numpy as np
from fancyimpute import KNN
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_blobs

# Создать матрицу симулированных признаков
features, _ = make_blobs(n_samples = 1000,
                         n_features = 2,
                         random_state = 1)
```

[1] Импутация (imputation) — процесс замещения пропущенных, некорректных или несостоятельных значений другими значениями. — *Прим. перев.*

[2] В приведенном фрагменте кода используется библиотека fancyimpute. В отличие от Linux и MacOS установка библиотеки fancyimpute в Windows 10 может вызвать проблемы. В этом случае следует использовать дистрибутив Anaconda Python и установить ее из консоли conda. Так или иначе, в прилагаемых блокнотах Jupyter приведен исходный код ядра библиотеки, который позволяет выполнить приведенный выше рецепт 4.11. — *Прим. перев.*

```
# Стандартизировать признаки
scaler = StandardScaler()
standardized_features = scaler.fit_transform(features)

# Заменить первое значение первого признака на пропущенное значение
true_value = standardized_features[0,0]
standardized_features[0,0] = np.nan

# Предсказать пропущенные значения в матрице признаков
features_knn_imputed = KNN(k=5, verbose=0).complete(standardized_features)

# Сравнить истинные и импутированные значения
print("Истинное значение:", true_value)
print("Импутированное значение:", features_knn_imputed[0,0])
```

```
Истинное значение: 0.8730186114
Импутированное значение: 1.09553327131
```

В качестве альтернативы мы можем использовать модуль `Imputer` библиотеки scikit-learn для заполнения пропущенных значений средними, медианными или наиболее частыми значениями признаков. Вместе с тем мы, как правило, получаем результаты хуже, чем с *k* ближайшими соседями:

```
# Загрузить библиотеку
from sklearn.preprocessing import Imputer

# Создать заполнитель
mean_imputer = Imputer(strategy="mean", axis=0)

# Импутировать значения
features_mean_imputed = mean_imputer.fit_transform(features)

# Сравнить истинные и импутированные значения
print("Истинное значение:", true_value)
print("Импутированное значение:", features_mean_imputed[0,0])
```

```
Истинное значение: 0.8730186113995938
Импутированное значение: -3.058372724614996
```

Обсуждение

Существуют две основные стратегии замены пропущенных данных подстановочными значениями, каждая из которых имеет свои сильные и слабые стороны. Во-первых, для предсказания значений пропущенных данных мы можем использовать машинное самообучение. Для этого признак с пропущенными значениями рассматривается как вектор целей и оставшееся подмножество признаков используется для предсказания пропущенных значений. Хотя для вычисления значений можно

применять широкий спектр машинно-обучающихся алгоритмов, популярным выбором является алгоритм *k* ближайших соседей (KNN). Алгоритм KNN подробно рассматривается в *главе 15*, но краткое его объяснение заключается в том, что в данном алгоритме для предсказания пропущенного значения используется *k* ближайших наблюдений (в соответствии с некоторой метрикой расстояния). В нашем решении мы предсказали пропущенное значение, используя пять ближайших наблюдений.

Недостаток алгоритма KNN: для того чтобы знать, какие наблюдения наиболее близки к пропущенному значению, необходимо вычислить расстояние между пропущенным значением и каждым отдельным наблюдением. Это разумно в небольших наборах данных, но быстро становится проблематичным, если набор данных содержит миллионы наблюдений.

Альтернативной и более масштабируемой стратегией является заполнение всех пропущенных значений некоторым средним значением. Например, в нашем решении мы использовали библиотеку scikit-learn для заполнения пропущенных значений средним значением признака. Импутированное значение часто не так близко к истинному значению, как при использовании KNN, но мы можем легко масштабировать заполнение на основе среднего для данных, содержащих миллионы наблюдений.

Если мы используем импутацию, то хорошей идеей является создание бинарного признака, указывающего на то, содержит ли наблюдение импутированное значение или нет.

Дополнительные материалы для чтения

- Статья "Исследование применения *k* ближайших соседей как метода импутации" ("A Study of K-Nearest Neighbour as an Imputation Method", **http://bit.ly/2HS9sAT**).

Работа с категориальными данными

Введение

Часто бывает полезно разбивать объекты на категории не по количеству, а по качеству. Эта качественная информация нередко представляется как принадлежность наблюдения к отдельной категории, такой как пол, цвета или марка автомобиля. Однако не все категориальные данные одинаковые. Наборы категорий без внутреннего упорядочения называются *номинальными*. Примеры номинальных категорий включают:

- синий, красный, зеленый;
- мужчина, женщина;
- банан, клубника, яблоко.

С другой стороны, когда набор категорий имеет некое естественное упорядочение, мы называем его *порядковым*. Например:

- низкий, средний, высокий;
- молодые, старые;
- согласен, нейтрален, не согласен.

Более того, категориальная информация часто представлена в данных в виде вектора или столбца символьных значений (например, "Мэн", "Техас", "Делавэр"). Проблема в том, что большинство машинно-обучающихся алгоритмов требуют ввода числовых значений.

Алгоритм k ближайших соседей предоставляет простой пример. Одним из шагов в алгоритме является вычисление расстояний между наблюдениями — часто с использованием евклидова расстояния:

$$\sqrt{\sum_{i=1}^{n}(x_i - y_i)^2},$$

где x и y — это два наблюдения; i — номер признака наблюдений.

Однако вычисление расстояния, очевидно, невозможно, если значение x_i является строковым типом (например, "Texas"). Для того чтобы его можно было ввести в уравнение евклидова расстояния, нам нужно преобразовать это строковое значение в числовой формат. Наша цель — сделать преобразование, которое правильно передает информацию в категориях (упорядоченность, относительные интервалы

между категориями и т. д.). В этой главе мы рассмотрим методы этого преобразования, а также выясним, как преодолеть другие проблемы, часто возникающие при обработке категориальных данных.

5.1. Кодирование номинальных категориальных признаков

Задача

Дан признак с номинальными классами, который не имеет внутренней упорядоченности (например, яблоко, груша, банан).

Решение

Преобразовать признак в кодировку с одним активным состоянием[1] с помощью класса `LabelBinarizer` библиотеки scikit-learn:

```
# Импортировать библиотеки
import numpy as np
from sklearn.preprocessing import LabelBinarizer, MultiLabelBinarizer

# Создать признак
feature = np.array([["Texas"],
                    ["California"],
                    ["Texas"],
                    ["Delaware"],
                    ["Texas"]])

# Создать кодировщик одного активного состояния
one_hot = LabelBinarizer()

# Преобразовать признак
# в кодировку с одним активным состоянием
one_hot.fit_transform(feature)

array([[0, 0, 1],
       [1, 0, 0],
       [0, 0, 1],
       [0, 1, 0],
       [0, 0, 1]])
```

Для вывода классов можно воспользоваться атрибутом `classes_`:

```
# Взглянуть на классы признака
one_hot.classes_
```

[1] Словосочетание *кодирование с одним активным состоянием* (англ. one-hot encoding) пришло из терминологии цифровых интегральных микросхем, где оно описывает конфигурацию микросхемы, в которой допускается, чтобы только один бит был положительным (активным). — *Прим. перев.*

```
array(['California', 'Delaware', 'Texas'],
      dtype='<U10')
```

Если требуется обратить кодирование с одним активным состоянием, то можно применить метод `inverse_transform`:

```
# Обратить кодирование с одним активным состоянием
one_hot.inverse_transform(one_hot.transform(feature))
```

```
array(['Texas', 'California', 'Texas', 'Delaware', 'Texas'],
      dtype='<U10')
```

Для преобразования признака в кодировку с одним активным состоянием можно даже использовать библиотеку pandas (табл. 5.1):

```
# Импортировать библиотеку
import pandas as pd

# Создать фиктивные переменные из признака
pd.get_dummies(feature[:,0])
```

Таблица 5.1

	California	Delaware	Texas
0	0	0	1
1	1	0	0
2	0	0	1
3	0	1	0
4	0	0	1

Одной из полезных возможностей библиотеки scikit-learn является обработка ситуации, когда в каждом наблюдении перечисляется несколько классов:

```
# Создать мультиклассовый признак
multiclass_feature = [("Texas", "Florida"),
                      ("California", "Alabama"),
                      ("Texas", "Florida"),
                      ("Delware", "Florida"),
                      ("Texas", "Alabama")]

# Создать мультиклассовый кодировщик, преобразующий признак
# в кодировку с одним активным состоянием
one_hot_multiclass = MultiLabelBinarizer()

# Кодировать мультиклассовый признак
# в кодировку с одним активным состоянием
one_hot_multiclass.fit_transform(multiclass_feature)
```

```
array([[0, 0, 0, 1, 1],
       [1, 1, 0, 0, 0],
       [0, 0, 0, 1, 1],
       [0, 0, 1, 1, 0],
       [1, 0, 0, 0, 1]])
```

Опять-таки, классы можно увидеть с помощью атрибута `classes_`:

```
# Взглянуть на классы
one_hot_multiclass.classes_

array(['Alabama', 'California', 'Delware', 'Florida', 'Texas'],
      dtype=object)
```

Обсуждение

Можно подумать, что правильная стратегия состоит в том, чтобы назначать каждому классу числовое значение (например, Texas = 1, California = 2). Однако, когда классы не имеют внутренней упорядоченности (например, Техас не "меньше" Калифорнии), числовые значения ошибочно создают порядок, которого нет.

Правильной стратегией является создание в исходном признаке бинарного признака для каждого класса. Это нередко называется *кодированием с одним активным состоянием* (в литературе по машинному самообучению) или *фиктивизацией* (в статистической и научно-исследовательской литературе). Признаком в нашем решении был вектор, содержащий три класса (т. е. Техас, Калифорния и Делавэр). В кодировании с одним активным состоянием каждый класс становится одноэлементным признаком с единицами, когда класс появляется, и нулями в противном случае. Поскольку наш признак имел три класса, кодирование с одним активным состоянием вернуло три бинарных признака (по одному для каждого класса). Используя кодирование с одним активным состоянием, можно фиксировать принадлежность наблюдения к классу, сохраняя при этом сведения о том, что в классе отсутствует какая-либо иерархия.

Наконец, стоит отметить, что после кодирования признака в кодировку с одним активным состоянием часто рекомендуется отбрасывать один из закодированных в результирующей матрице признаков, чтобы избежать линейной зависимости.

Дополнительные материалы для чтения

- "Ловушка фиктивной переменной", блог Algosome (**http://bit.ly/2FvVJkC**).
- "Исключение одного из столбцов при использовании кодирования с одним активным состоянием", вопросно-ответный статистический ресурс CrossValidated (**http://bit.ly/2FwrxG0**).

5.2. Кодирование порядковых категориальных признаков

Задача

Дан порядковый категориальный признак (например, высокий, средний, низкий). Выполнить его кодировку.

Решение

Использовать метод `replace` фрейма данных pandas для преобразования строковых меток в числовые эквиваленты:

```
# Загрузить библиотеку
import pandas as pd

# Создать признаки
dataframe = pd.DataFrame({"оценка": ["низкая", "низкая",
                                     "средняя", "средняя", "высокая"]})

# Создать словарь преобразования шкалы
scale_mapper = {"низкая":1,
                "средняя":2,
                "высокая":3}

# Заменить значения признаков значениями словаря
dataframe["оценка"].replace(scale_mapper)
```

```
0    1
1    1
2    2
3    2
4    3
Name: оценка, dtype: int64
```

Обсуждение

Нередко существует признак с классами, которые имеют какую-то естественную упорядоченность. Известный пример — шкала Лайкерта:

- полностью согласен;
- согласен;
- нейтрален;
- не согласен;
- категорически не согласен.

При кодировании признака для использования в машинном самообучении требуется преобразовать порядковые классы в числовые значения, которые поддерживают

идею упорядоченности. Наиболее распространенным подходом является создание словаря, который строковой метке класса ставит в соответствие число, а затем применяет это соотнесение к признаку.

Важно, чтобы выбор числовых значений основывался на имеющейся априорной информации о порядковых классах. В нашем решении метка высокая буквально в три раза больше метки низкая. Это нормально в обычных случаях, но может нарушится, если принятые интервалы между классами не равны:

```
dataframe = pd.DataFrame({"оценка": ["низкая",
                                     "низкая",
                                     "средняя",
                                     "средняя",
                                     "высокая",
                                     "чуть больше средней"]})
scale_mapper = {"низкая":1,
                "средняя":2,
                "чуть больше средней": 3,
                "высокая":4}

dataframe["оценка"].replace(scale_mapper)
0    1
1    1
2    2
3    2
4    4
5    3
Name: оценка, dtype: int64
```

В этом примере расстояние между низкой и средней равно расстоянию между средней и чуть больше средней, что почти наверняка не является точным. Лучший подход состоит в том, чтобы учитывать числовые значения, сопоставленные классам:

```
scale_mapper = {"низкая":1,
                "средняя":2,
                "чуть больше средней": 2.1,
                "высокая":3}

dataframe["оценка"].replace(scale_mapper)

0    1.0
1    1.0
2    2.0
3    2.0
4    3.0
5    2.1
Name: оценка, dtype: float64
```

5.3. Кодирование словарей признаков

Задача

Дан словарь, и требуется его конвертировать в матрицу признаков.

Решение

Использовать класс-векторизатор словаря `DictVectorizer`:

```
# Импортировать библиотеку
from sklearn.feature_extraction import DictVectorizer

# Создать словарь
data_dict = [{"красный": 2, "синий": 4},
             {"красный": 4, "синий": 3},
             {"красный": 1, "желтый": 2},
             {"красный": 2, "желтый": 2}]

# Создать векторизатор словаря
dictvectorizer = DictVectorizer(sparse=False)

# Конвертировать словарь в матрицу признаков
features = dictvectorizer.fit_transform(data_dict)

# Взглянуть на матрицу признаков
features

array([[4., 2., 0.],
       [3., 4., 0.],
       [0., 1., 2.],
       [0., 2., 2.]])
```

По умолчанию `DictVectorizer` выводит разреженную матрицу, в которой хранятся только элементы со значением, отличным от 0. Это может быть очень полезно, когда имеются массивные матрицы (часто встречающиеся в обработке естественного языка) и требуется минимизировать потребности в оперативной памяти. Мы можем заставить `DictVectorizer` вывести плотную матрицу, используя `sparse=False`.

Имена каждого созданного признака можно получить с помощью метода `get_feature_names`:

```
# Получить имена признаков
feature_names = dictvectorizer.get_feature_names()

# Взглянуть на имена признаков
feature_names

['желтый', 'красный', 'синий']
```

Хотя это и не обязательно, для иллюстрации мы можем создать фрейм данных pandas, чтобы результат лучше выглядел (табл. 5.2):

```
# Импортировать библиотеку
import pandas as pd

# Создать фрейм данных из признаков
pd.DataFrame(features, columns=feature_names)
```

Таблица 5.2

	желтый	красный	синий
0	0.0	2.0	4.0
1	0.0	4.0	3.0
2	2.0	1.0	0.0
3	2.0	2.0	0.0

Обсуждение

Словарь является популярной структурой данных, используемой многими языками программирования; однако машинно-обучающиеся алгоритмы ожидают, что данные будут в виде матрицы. Конвертировать словарь в матрицу можно, используя объект `dictvectorizer` библиотеки scikit-learn.

Такая ситуация является обычной во время обработки естественного языка. Например, дана коллекция документов, и для каждого документа имеется словарь, содержащий количество вхождений каждого слова в документ. Используя объект `dictvectorizer`, можно легко создать матрицу признаков, где каждый признак — это количество вхождений слова в каждый документ:

```
# Создать словари частотностей слов для четырех документов
doc_1_word_count = {"красный": 2, "синий": 4}
doc_2_word_count = {"красный": 4, "синий": 3}
doc_3_word_count = {"красный": 1, "желтый": 2}
doc_4_word_count = {"красный": 2, "желтый": 2}

# Создать список
doc_word_counts = [doc_1_word_count,
                   doc_2_word_count,
                   doc_3_word_count,
                   doc_4_word_count]

# Конвертировать список словарей частотностей слов в матрицу признаков
dictvectorizer.fit_transform(doc_word_counts)
```

```
array([[0., 2., 4.],
       [0., 4., 3.],
       [2., 1., 0.],
       [2., 2., 0.]])
```

В нашем игрушечном примере имеется всего три уникальных слова (красный, желтый, синий), поэтому в нашей матрице всего три признака; однако вы можете себе представить, что если бы каждый документ был на самом деле книгой в университетской библиотеке, то наша матрица признаков была бы очень большой (и тогда потребовалось бы установить параметр разреженности `sparse` в `True`).

Дополнительные материалы для чтения

- "Как применять словари в Python", ресурс для новичков PythonForBeginners (**http://bit.ly/2HReoWz**).
- "Разреженные матрицы SciPy" (**http://bit.ly/2HReBZR**).

5.4. Импутация пропущенных значений классов

Задача

Дан категориальный признак, содержащий пропущенные значения, которые требуется заменить предсказанными значениями.

Решение

Идеальное решение — натренировать машинно-обучающийся классификационный алгоритм для предсказания пропущенных значений, обычно классификатор *k* ближайших соседей (KNN):

```
# Загрузить библиотеки
import numpy as np
from sklearn.neighbors import KNeighborsClassifier

# Создать матрицу признаков с категориальным признаком
X = np.array([[0, 2.10, 1.45],
              [1, 1.18, 1.33],
              [0, 1.22, 1.27],
              [1, -0.21, -1.19]])

# Создать матрицу признаков
# с отсутствующими значениями в категориальном признаке
X_with_nan = np.array([[np.nan, 0.87, 1.31],
                       [np.nan, -0.67, -0.22]])

# Натренировать ученика KNN
clf = KNeighborsClassifier(3, weights='distance')
trained_model = clf.fit(X[:,1:], X[:,0])
```

```
# Предсказать класс пропущенных значений
imputed_values = trained_model.predict(X_with_nan[:,1:])

# Соединить столбец предсказанного класса с другими признаками
X_with_imputed = np.hstack((imputed_values.reshape(-1,1), X_with_nan[:,1:]))

# Соединить две матрицы признаков
np.vstack((X_with_imputed, X))

array([[ 0.  ,  0.87,  1.31],
       [ 1.  , -0.67, -0.22],
       [ 0.  ,  2.1 ,  1.45],
       [ 1.  ,  1.18,  1.33],
       [ 0.  ,  1.22,  1.27],
       [ 1.  , -0.21, -1.19]])
```

Альтернативным решением является заполнение пропущенных значений наиболее частыми значениями признаков:

```
from sklearn.preprocessing import Imputer

# Соединить две матрицы признаков
X_complete = np.vstack((X_with_nan, X))
imputer = Imputer(strategy='most_frequent', axis=0)
imputer.fit_transform(X_complete)

array([[ 0.  ,  0.87,  1.31],
       [ 0.  , -0.67, -0.22],
       [ 0.  ,  2.1 ,  1.45],
       [ 1.  ,  1.18,  1.33],
       [ 0.  ,  1.22,  1.27],
       [ 1.  , -0.21, -1.19]])
```

Обсуждение

Когда в категориальном признаке имеются пропущенные значения, лучшее решение — взять инструментальный комплект машинно-обучающихся алгоритмов для предсказания значений пропущенных наблюдений. Мы можем сделать это, рассматривая признак с пропущенными значениями как вектор целей, а другие признаки — как матрицу признаков. Общепринято использовать алгоритм (подробно рассмотренный далее в этой книге), который присваивает пропущенному значению медианный класс *k* ближайших наблюдений.

Кроме того, пропущенные значения можно заполнить наиболее часто используемым классом признака. Хотя он менее сложен, чем KNN, он гораздо более масштабируем для более крупных данных. В любом случае рекомендуется включить бинарный признак, указывающий на то, какие наблюдения содержат импутированные значения.

Дополнительные материалы для чтения

- "Преодоление пропущенных значений в классификаторе на основе случайного леса", веб-сайт для аналитиков данных Airbnb Engineering & Data Science (**http://bit.ly/2HSsNBF**).
- "Исследование применения *k* ближайших соседей как метода импутации" ("A Study of K-Nearest Neighbour as an Imputation Method"), статья (**http://bit.ly/2HS9sAT**).

5.5. Работа с несбалансированными классами

Задача

Дан вектор целей с очень несбалансированными классами.

Решение

Собрать больше данных. Если это невозможно, то изменить метрические показатели, используемые для оценивания модели. Если это не работает, рассмотреть возможность использования встроенных в модель параметров веса классов (если таковые имеются), делая понижающий или повышающий отбор. Мы рассмотрим метрические показатели оценивания чуть позже в одной из следующих глав, а пока сосредоточимся на параметрах веса классов, понижающем и повышающем отборе.

Для того чтобы продемонстрировать наши решения, нам нужно создать немного данных с несбалансированными классами. Набор данных ирисов Фишера содержит три сбалансированных класса по 50 наблюдений, каждый из которых указывает на вид цветка — *ирис щетинистый* (*Iris setosa*), *ирис виргинский* (*Iris virginica*) и *ирис разноцветный* (*Iris versicolor*). Для того чтобы разбалансировать набор данных, мы удаляем 40 из 50 наблюдений *ириса щетинистого*, а затем объединяем классы *ирис виргинский* и *ирис разноцветный*. Конечным результатом является бинарный вектор целей, указывающий на то, является ли наблюдение цветком *ириса щетинистого* или нет. Результатом станут 10 наблюдений *ирис щетинистый* (класс 0) и 100 наблюдений не *ирис щетинистый* (класс 1):

```
# Загрузить библиотеки
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_iris

# Загрузить данные цветков ириса Фишера
iris = load_iris()

# Создать матрицу признаков
features = iris.data
```

```
# Создать вектор целей
target = iris.target

# Удалить первые 40 наблюдений
features = features[40:,:]
target = target[40:]

# Создать бинарный вектор целей, указывающий, является ли класс 0
target = np.where((target == 0), 0, 1)

# Взглянуть на несбалансированный вектор целей
target
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 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, 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])
```

Многие алгоритмы в scikit-learn предлагают параметр для взвешивания классов во время тренировки, чтобы противодействовать эффекту их разбалансировки. Хотя мы еще его не рассматривали, классификатор на основе случайного леса `RandomForestClassifier` **является популярным классификационным алгоритмом и включает параметр** `class_weight`**. Можете передать аргумент, явно задающий желаемые веса классов:**

```
# Создать веса
weights = {0: .9, 1: 0.1}

# Создать классификатор на основе случайного леса с весами
RandomForestClassifier(class_weight=weights)
```

```
RandomForestClassifier(bootstrap=True, class_weight={0: 0.9, 1: 0.1},
                       criterion='gini', max_depth=None,
                       max_features='auto',
                       max_leaf_nodes=None, min_impurity_decrease=0.0,
                       min_impurity_split=None, min_samples_leaf=1,
                       min_samples_split=2, min_weight_fraction_leaf=0.0,
                       n_estimators=10, n_jobs=1, oob_score=False,
                       random_state=None, verbose=0, warm_start=False)
```

Либо передать аргумент `balanced`**, который автоматически создает веса, обратно пропорциональные частотам классов:**

```
# Натренировать случайный лес с помощью сбалансированных весов классов
RandomForestClassifier(class_weight="balanced")
```

```
RandomForestClassifier(bootstrap=True, class_weight='balanced',
                       criterion='gini', max_depth=None,
                       max_features='auto',
```

```
                max_leaf_nodes=None, min_impurity_decrease=0.0,
                min_impurity_split=None, min_samples_leaf=1,
                min_samples_split=2, min_weight_fraction_leaf=0.0,
                n_estimators=10, n_jobs=1, oob_score=False,
                random_state=None, verbose=0, warm_start=False)
```

В качестве альтернативы можно понизить мажоритарный класс или повысить миноритарный класс. В понижающем отборе мы произвольно делаем выборку без возврата из мажоритарного класса (т. е. класса с бóльшим количеством наблюдений), чтобы создать новое подмножество наблюдений, равное по размеру миноритарному классу. Например, если миноритарный класс имеет 10 наблюдений, мы случайным образом отберем 10 наблюдений из мажоритарного класса и используем эти 20 наблюдений в качестве наших данных. Сделаем это, используя наши несбалансированные данные цветков ириса:

```
# Индексы наблюдений каждого класса
i_class0 = np.where(target == 0)[0]
i_class1 = np.where(target == 1)[0]

# Количество наблюдений в каждом классе
n_class0 = len(i_class0)
n_class1 = len(i_class1)

# Для каждого наблюдения класса 0 сделать случайную выборку
# из класса 1 без возврата
i_class1_downsampled = np.random.choice(i_class1, size=n_class0, replace=False)

# Соединить вектор целей класса 0 с
# вектором целей пониженного класса 1
np.hstack((target[i_class0], target[i_class1_downsampled]))

array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])

# Соединить матрицу признаков класса 0 с
# матрицей признаков пониженного класса 1
np.vstack((features[i_class0,:], features[i_class1_downsampled,:]))[0:5]

array([[5. , 3.5, 1.3, 0.3],
       [4.5, 2.3, 1.3, 0.3],
       [4.4, 3.2, 1.3, 0.2],
       [5. , 3.5, 1.6, 0.6],
       [5.1, 3.8, 1.9, 0.4]])
```

Другой вариант — повысить миноритарный класс. В повышающем отборе для каждого наблюдения в мажоритарном классе мы случайно отбираем наблюдение из миноритарного класса с возвратом. Конечным результатом является одинаковое количество наблюдений из миноритарных и мажоритарных классов. Повышающий отбор реализуется очень похоже на понижающий отбор, только наоборот:

```
# Для каждого наблюдения в классе 1 сделать случайную выборку
# из класса 0 с возвратом
i_class0_upsampled = np.random.choice(i_class0, size=n_class1, replace=True)

# Соединить повышенный вектор целей класса 0
# с вектором целей класса 1
np.concatenate((target[i_class0_upsampled], target[i_class1]))
```

```
array([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, 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, 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,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 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, 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])
```

```
# Соединить повышенную матрицу признаков класса 0
# с матрицей признаков класса 1
np.vstack((features[i_class0_upsampled,:], features[i_class1,:]))[0:5]
```

```
array([[5.1, 3.8, 1.9, 0.4],
       [5. , 3.5, 1.3, 0.3],
       [5. , 3.5, 1.6, 0.6],
       [5. , 3.5, 1.3, 0.3],
       [5.1, 3.8, 1.9, 0.4]])
```

Обсуждение

В реальном мире несбалансированные классы встречаются повсюду — большинство посетителей не нажимают кнопку **Купить**, и многие виды рака, к счастью, — редки. По этой причине обработка несбалансированных классов является в машинном самообучении распространенной задачей.

Наша лучшая стратегия — просто собрать больше наблюдений, в особенности наблюдений из миноритарного класса. Однако зачастую это просто невозможно, поэтому приходится прибегать к другим вариантам.

Вторая стратегия заключается в использовании модельногоо оценочного показателя, более подходящего для несбалансированных классов. В качестве метрического показателя оценки результативности модели часто используется точность, но при наличии несбалансированных классов показатель точности может не подходить. Например, если только 0.5% наблюдений указывают на редкий вид рака, то даже наивная модель, предсказывающая, что ни у кого нет рака, будет точной на 99.5%. Очевидно, что это не идеально. Некоторые более эффективные метрические пока-

затели, которые мы обсудим в последующих главах, — это матрицы ошибок, прецизионность, полнота, оценки F_1 и кривые ROC.

Третья стратегия заключается в использовании параметров взвешивания класса, включенных в реализации некоторых моделей. Это позволяет настроить алгоритм для несбалансированных классов. К счастью, многие классификаторы библиотеки scikit-learn имеют параметр `class_weight`, что делает эту стратегию хорошим вариантом.

Четвертая и пятая стратегии взаимосвязаны: понижающий и повышающий отбор. При понижающем отборе мы создаем случайное подмножество мажоритарного класса одинакового размера с миноритарным классом. При повышающем отборе мы неоднократно отбираем образцы из миноритарного класса с заменой, чтобы сделать его равного размера с мажоритарным классом. Решение между использованием понижающего и повышающего отбора зависит от контекста, и в целом мы должны попытаться увидеть, который из них дает лучшие результаты.

Работа с текстом

Введение

Неструктурированные текстовые данные, такие как содержимое книги или твита, являются одним из самых интересных источников признаков и одними из самых сложных для обработки. В этой главе мы рассмотрим стратегии преобразования текста в информационно богатые признаки. Это не означает, что рецепты, описанные здесь, являются всеобъемлющими. Существуют целые академические дисциплины, ориентированные на обработку этого и подобных типов данных, и содержимое всех их методов заполнит небольшую библиотеку. Несмотря на это, есть некоторые часто используемые методы, и овладение ими добавит ценные инструменты в наш набор инструментов предобработки.

6.1. Очистка текста

Задача

Даны некоторые неструктурированные текстовые данные, и требуется выполнить их элементарную очистку.

Решение

Большинство элементарных операций очистки текста соответствуют элементарным операциям языка Python над строковыми значениями, в частности `strip`, `replace` и `split`:

```
# Создать текст
text_data = ["   Interrobang. By Aishwarya Henriette     ",
             "Parking And Going. By Karl Gautier",
             "    Today Is The night. By Jarek Prakash   "]

# Удалить пробелы
strip_whitespace = [string.strip() for string in text_data]

# Показать текст
strip_whitespace
```

```
['Interrobang. By Aishwarya Henriette',
 'Parking And Going. By Karl Gautier',
 'Today Is The night. By Jarek Prakash']

# Удалить точки
remove_periods = [string.replace(".", "") for string in strip_whitespace]

# Показать текст
remove_periods

['Interrobang By Aishwarya Henriette',
 'Parking And Going By Karl Gautier',
 'Today Is The night By Jarek Prakash']
```

Кроме того, мы обычно создаем и применяем собственную функцию преобразования:

```
# Создать функцию
def capitalizer(string: str) -> str:
    return string.upper()

# Применить функцию
[capitalizer(string) for string in remove_periods]

['INTERROBANG BY AISHWARYA HENRIETTE',
 'PARKING AND GOING BY KARL GAUTIER',
 'TODAY IS THE NIGHT BY JAREK PRAKASH']
```

Наконец, для выполнения мощных строковых операций можно воспользоваться регулярными выражениями:

```
# Импортировать библиотеку
import re

# Создать функцию
def replace_letters_with_X(string: str) -> str:
    return re.sub(r"[a-zA-Z]", "X", string)

# Применить функцию
[replace_letters_with_X(string) for string in remove_periods]

['XXXXXXXXXXX XX XXXXXXXXX XXXXXXXXX',
 'XXXXXXX XXX XXXXX XX XXXX XXXXXXX',
 'XXXXX XX XXX XXXXX XX XXXXX XXXXXXX']
```

Обсуждение

Большинство текстовых данных требуется очистить перед тем, как их использовать для построения признаков. Подавляющую часть элементарной очистки текста

можно выполнять с помощью стандартных строковых операций Python. В реальном мире мы, скорее всего, определим собственную функцию очистки (например, `capitalizer`), объединяющую несколько задач очистки, и применим ее к текстовым данным.

Дополнительные материалы для чтения

- "Практическое руководство по регулярным выражениям в Python для начинающих", ресурс для аналитиков данных Analytics Vidhya (**http://bit.ly/2HTGZuu**).

6.2. Разбор и очистка разметки HTML

Задача

Даны текстовые данные с элементами HTML, и требуется извлечь только текст.

Решение

Использовать обширный набор функциональных средств библиотеки Beautiful Soup для анализа и извлечения текста из HTML:

```
# Загрузить библиотеку
from bs4 import BeautifulSoup

# Создать немного кода с разметкой HTML
html = """
       <div class='full_name'><span style='font-weight:bold'>
       Masego</span> Azra</div>"
       """

# Выполнить разбор html
soup = BeautifulSoup(html, "lxml")

# Найти элемент div с классом "full_name", показать текст
soup.find("div", { "class" : "full_name" }).text
'Masego Azra'
```

Обсуждение

Несмотря на странное название, Beautiful Soup — это мощная библиотека Python, предназначенная для "выскабливания" HTML. Как правило, Beautiful Soup используется для вычищения "живых" веб-сайтов, но ее можно также легко применять для извлечения текстовых данных, встроенных в HTML. Полный спектр операций библиотеки Beautiful Soup выходит за рамки этой книги, но даже те несколько методов, примененных в нашем решении, показывают, как легко можно проанализировать разметку HTML для извлечения данных, которые нам необходимы.

Дополнительные материалы для чтения

- Веб-сайт библиотеки Beautiful Soup (**http://bit.ly/2pwZcYs**).

6.3. Удаление знаков препинания

Задача

Дан признак в виде текстовых данных, и требуется удалить знаки препинания.

Решение

Определить функцию, которая использует метод `translate` со словарем знаков препинания:

```
# Загрузить библиотеки
import unicodedata
import sys

# Создать текст
text_data = ['Hi!!!! I. Love. This. Song....',
             '10000% Agree!!!! #LoveIT',
             'Right?!?!']

# Создать словарь знаков препинания
punctuation = dict.fromkeys(i for i in range(sys.maxunicode)
                            if unicodedata.category(chr(i)).startswith('P'))

# Удалить любые знаки препинания во всех строковых значениях
[string.translate(punctuation) for string in text_data]
['Hi I Love This Song', '10000 Agree LoveIT', 'Right']
```

Обсуждение

Метод `translate` языка Python популярен благодаря своей невероятной скорости. В нашем решении сначала мы создали словарь `punctuation`, в котором в качестве ключей размещены все знаки препинания, присутствующие в Юникоде, и в качестве значений — `None`. Затем мы преобразовали все символы строкового значения, которые находятся среди знаков препинания, в `None`, фактически удалив их. Существуют более читаемые способы удаления знаков препинания, но это несколько хакерское решение имеет преимущество в том, что оно гораздо быстрее, чем другие альтернативы.

Важно осознавать тот факт, что знаки препинания содержат информацию (например, сравните "правильно?" и "правильно!"). Удаление знаков препинания часто является необходимым злом для создания признаков; однако, если знаки препинания важны, мы должны обязательно принимать их во внимание.

6.4. Лексемизация текста

Задача

Дан текст, и требуется разбить его на отдельные слова.

Решение

Комплект естественно-языковых инструментов NLTK (Natural Language Toolkit for Python) имеет мощный набор операций над текстом, включая лексемизацию на слова:

```
# Загрузить библиотеку
from nltk.tokenize import word_tokenize

# Создать текст
string = "Сегодняшняя наука — это технология завтрашнего дня"

# Лексемизировать на слова
word_tokenize(string)
```

```
['Сегодняшняя', 'наука', '-', 'это', 'технология', 'завтрашнего', 'дня']
```

Мы также можем лексемизировать текст на предложения:

```
# Загрузить библиотеку
from nltk.tokenize import sent_tokenize

# Создать текст
string = """Сегодняшняя наука — это технология завтрашнего дня.
            Затра начинается сегодня."""

# Лексемизировать на предложения
sent_tokenize(string)
```

```
['Сегодняшняя наука — это технология завтрашнего дня.',
 'Затра начинается сегодня.']
```

Обсуждение

Лексемизация, в особенности лексемизация на слова, является распространенной задачей после очистки текстовых данных, потому что это первый шаг в процессе превращения текста в данные, которые мы будем использовать для создания полезных признаков.

6.5. Удаление стоп-слов

Задача

Имеются лексемизированные текстовые данные, из которых требуется удалить чрезвычайно общеупотребительные слова (например, *a*, *is*, *of*, *on*), несущие в себе мало информации.

Решение

Использовать функцию `stopwords` библиотеки NLTK:

```
# Загрузить библиотеку
from nltk.corpus import stopwords

# Перед этим вам следует скачать набор стоп-слов
# import nltk
# nltk.download('stopwords')

# Создать лексемы слов
tokenized_words = ['i',
                   'am',
                   'going',
                   'to',
                   'go',
                   'to',
                   'the',
                   'store',
                   'and',
                   'park']

# Загрузить стоп-слова
stop_words = stopwords.words('english')

# Удалить стоп-слова
[word for word in tokenized_words if word not in stop_words]
```

```
['going', 'go', 'store', 'park']
```

Для удаления русских стоп-слов следует просто поменять аргумент функции:

```
stop_list = stopwords.words('russian')

# Создать словарные лексемы
tokenized_words = ['я',
                   'бы',
                   'пошел',
                   'в',
```

```
                    'пищерию',
                    'покушать',
                    'пиццы',
                    'и',
                    'потом',
                    'в',
                    'парк']

# Загрузить стоп-слова
stop_words = stopwords.words('russian')

# Удалить стоп-слова
[word for word in tokenized_words if word not in stop_words]

['пошел', 'пищерию', 'покушать', 'пиццы', 'парк']
```

Обсуждение

Хотя стоп-слова могут относиться к любому набору слов, которые мы хотим удалить перед обработкой, часто этот термин относится к чрезвычайно распространенным словам, которые сами по себе содержат мало информации. NLTK имеет список общеупотребительных стоп-слов, который можно использовать для их отыскания и удаления в наших лексемизированных словах:

```
# Удалить стоп-слова
stop_words[:5]

['i', 'me', 'my', 'myself', 'we']
```

Обратите внимание, что функция `stopwords` библиотеки NLTK исходит из того, что все лексемизированные слова находятся в нижнем регистре.

6.6. Выделение основ слов

Задача

Даны лексемизированные слова, которые требуется преобразовать в их корневые формы.

Решение

Использовать класс `PorterStemmer` библиотеки NLTK:

```
# Загрузить библиотеку
from nltk.stem.porter import PorterStemmer

# Создать лексемы слов
tokenized_words = ['i', 'am', 'humbled', 'by', 'this', 'traditional', 'meeting']
```

```
# Создать стеммер
porter = PorterStemmer()

# Применить стеммер
[porter.stem(word) for word in tokenized_words]
```

```
['i', 'am', 'humbl', 'by', 'thi', 'tradit', 'meet']
```

Для слов русского языка используется стеммер `Snowball`:

```
# Загрузить библиотеку
from nltk.stem.snowball import SnowballStemmer

# Создать лексемы слов
tokenized_words = ['рыбаки', 'рыбаков', 'рыбаками']

# Создать стеммер
snowball = SnowballStemmer("russian")

# Применить стеммер
[snowball.stem(word) for word in tokenized_words]
```

```
['рыбак', 'рыбак', 'рыбак']
```

Обсуждение

Выделение основ слов сводит слово к его основе путем выявления и отсечения аффиксов (например, герундийных форм в английском языке), сохраняя при этом корневой смысл слова. Например, слова "tradition" и "traditional" имеют "tradit" в качестве основы, указывая на то, что, хотя это и разные слова, они представляют одно и то же общее понятие. Используя наши текстовые данные, мы преобразуем их во что-то менее читаемое, но более близкое к базовому значению и, следовательно, более подходящее для сопоставления наблюдений. Класс NLTK `PorterStemmer` реализует широко используемый алгоритм выделения основ Портера, который отсекает или заменяет общеупотребительные суффиксы для получения основы слова.

Дополнительные материалы для чтения

- Алгоритм выделения основ слов (стеммер) Портера (**http://bit.ly/2FB5ZZb**).
- Алгоритм выделения основ слов русского языка (стеммер) Snowball (**https://bit.ly/2rDzrp9**).

6.7. Лемматизация слов

Задача

Даны лексемизированные слова, которые требуется собрать в синонимические ряды.

Решение

Использовать класс NLTK `WordNetLemmatizer`:

```
# Загрузить библиотеку
from nltk.stem import WordNetLemmatizer

# Создать лексемы слов
tokenized_words = ['go', 'went', 'gone', 'am', 'are', 'is', 'was', 'were']

# Создать лемматизатор
lemmatizer = WordNetLemmatizer()

# Применить лемматизатор
[lemmatizer.lemmatize(word, pos='v') for word in tokenized_words]
```

```
['go', 'go', 'go', 'be', 'be', 'be', 'be', 'be']
```

Обсуждение

Аналогично выделению основы слова, лемматизация также собирает разные флективные формы слова в группу, чтобы они могли анализироваться как одинаковые.

В отличие от выделения основы слова, данная процедура более запутанная и требует некоторых дополнительных знаний, как например правильная метка части речи, связанная с каждым словом, которое подлежит лемматизации. Результат лемматизации называется леммой и в сущности является словом в буквальном смысле. Простой подход с отсечением суффиксов для лемматизации работать не будет, потому что, например, некоторые формы неправильного глагола в английском языке имеют совершенно другую морфологию, чем их лемма. Например, go, goes, going и went все должны быть поставлены в соответствие глаголу go.

6.8. Разметка слов на части речи

Задача

Даны текстовые данные, и требуется пометить каждое слово или символ своей частью речи.

Решение

Использовать предварительно натренированный разметчик частей речи библиотеки NLTK:

```
# Загрузить библиотеки
from nltk import pos_tag
from nltk import word_tokenize
```

```
# Создать текст
text_data = "Chris loved outdoor running"

# Использовать предварительно натренированный
# разметчик частей речи
text_tagged = pos_tag(word_tokenize(text_data))

# Показать части речи
text_tagged
```

```
[('Chris', 'NNP'), ('loved', 'VBD'), ('outdoor', 'RP'), ('running', 'VBG')]
```

В результате получен список кортежей со словом и тегом части речи. NLTK использует метки частей речи текстового корпуса Penn Treebank. Приведем несколько примеров меток текстового корпуса Penn Treebank:

Метка	Часть речи
NNP	Имя собственное, единственное число
NN	Существительное, единственное число или неисчисляемое
RB	Наречие
VBD	Глагол, прошедшее время
VBG	Глагол, герундий или причастие настоящего времени
JJ	Прилагательное
PRP	Личное местоимение

После того как текст был помечен, метки можно использовать для поиска определенных частей речи. Вот, например, все существительные:

```
# Отфильтровать слова
[word for word, tag in text_tagged if tag in ['NN','NNS','NNP','NNPS'] ]
```

```
['Chris']
```

Более реалистичной является ситуация, когда есть данные, где каждое наблюдение содержит твит, и мы хотим преобразовать эти предложения в признаки отдельных частей речи (например, признак с 1, если присутствует собственное существительное, и 0 в противном случае):

```
# Загрузить библиотеки
import nltk
from sklearn.preprocessing import MultiLabelBinarizer

# Создать текст
tweets = ["I am eating a burrito for breakfast",
          "Political science is an amazing field",
          "San Francisco is an awesome city"]

# Создать список
tagged_tweets = []
```

```
# Пометить каждое слово и каждый твит
for tweet in tweets:
    tweet_tag = nltk.pos_tag(word_tokenize(tweet))
    tagged_tweets.append([tag for word, tag in tweet_tag])

# Применить кодирование с одним активным состоянием, чтобы
# конвертировать метки в признаки
one_hot_multi = MultiLabelBinarizer()
one_hot_multi.fit_transform(tagged_tweets)

array([[1, 1, 0, 1, 0, 1, 1, 1, 0],
       [1, 0, 1, 1, 0, 0, 0, 0, 1],
       [1, 0, 1, 1, 1, 0, 0, 0, 1]])
```

Применив атрибут `classes_`, мы увидим, что каждый признак является меткой части речи:

```
# Показать имена признаков
one_hot_multi.classes_

array(['DT', 'IN', 'JJ', 'NN', 'NNP', 'PRP', 'VBG', 'VBP', 'VBZ'],
      dtype=object)
```

Обсуждение

Если наш текст написан на английском языке и не относится к специализированной теме (например, медицине), то самым простым решением является использование предварительно натренированного разметчика на части речи NLTK. Однако если функция `pos_tag` не очень точна, то NLTK также дает нам возможность натренировать наш собственный разметчик. Основным недостатком тренировки разметчика является то, что нам нужен большой текстовый корпус, где метка каждого слова известна. Создание такого помеченного корпуса, очевидно, является трудоемкой задачей и, вероятно, будет последним средством, к которому мы обратимся.

С учетом сказанного, если бы у нас был помеченный корпус и мы хотели бы натренировать разметчик, то мы могли бы воспользоваться примером, приведенным далее. Текстовый корпус, который мы используем, — это стандартный корпус американского английского языка университета Брауна (Brown Corpus), один из самых популярных источников помеченного текста. Здесь мы используем *n*-граммный разметчик с откатом, где *n* — это количество предыдущих слов, которые мы учитываем во время предсказания метки части речи слова. Сначала мы принимаем во внимание два предыдущих слова, используя триграммный разметчик `TrigramTagger`; если два слова отсутствуют, то мы "откатываем назад" и принимаем во внимание метку предыдущего слова, используя биграммный разметчик `BigramTagger`, и, наконец, если и это не удается, то мы смотрим только на само слово, используя униграммный разметчик `UnigramTagger`. Для того чтобы проверить точность нашего разметчика, мы разбиваем наши текстовые данные на две части, тренируем наш разметчик на одной части и тестируем, насколько хорошо он предсказывает метки, на второй части:

```
# Загрузить библиотеку
from nltk.corpus import brown
from nltk.tag import UnigramTagger
from nltk.tag import BigramTagger
from nltk.tag import TrigramTagger

# Получить немного текста из стандартного текстового корпуса
# Brown Corpus, разбитого на предложения
sentences = brown.tagged_sents(categories='news')

# Разбить на 4000 предложений для тренировки и 4000 для тестирования
train = sentences[:4000]
test = sentences[4000:]

# Создать разметчик с откатом
unigram = UnigramTagger(train)
bigram = BigramTagger(train, backoff=unigram)
trigram = TrigramTagger(train, backoff=bigram)

# Показать точность
trigram.evaluate(test)
```

```
0.8179229731754832
```

Дополнительные материалы для чтения

- Алфавитный список меток частей речи текстового корпуса Penn Treebank (**http://bit.ly/2HROPo5**).
- Стандартный корпус современного американского варианта английского языка университета Брауна (Brown Corpus), Википедия (**http://bit.ly/2FzgMTx**).
- "Определение частей речи слов в русском тексте", статья на информационно-технологическом ресурсе Хабр (**https://bit.ly/2jWXTh3**).

6.9. Кодирование текста в качестве мешка слов

Задача

Даны текстовые данные, и требуется создать набор признаков, указывающих на количество вхождений определенного слова в текст наблюдения.

Решение

Использовать класс `CountVectorizer` библиотеки scikit-learn:

```
# Загрузить библиотеки
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer
```

```
# Создать текст
text_data = np.array(['Бразилия — моя любовь. Бразилия!',
                      'Швеция — лучше',
                      'Германия бьет обоих'])

# Создать матрицу признаков на основе мешка слов
count = CountVectorizer()
bag_of_words = count.fit_transform(text_data)

# Показать матрицу признаков
bag_of_words
```

```
<3x8 sparse matrix of type '<class 'numpy.int64'>'
       with 8 stored elements in Compressed Sparse Row format>
```

Этот результат является разреженным массивом, который часто необходим, когда объем текста большой. Однако в нашем игрушечном примере для просмотра матрицы частотности слов по каждому наблюдению мы можем применить метод `toarray`:

```
bag_of_words.toarray()
```

```
array([[2, 0, 0, 0, 1, 1, 0, 0],
       [0, 0, 0, 1, 0, 0, 0, 1],
       [0, 1, 1, 0, 0, 0, 1, 0]], dtype=int64)
```

Для просмотра слова, связанного с каждым признаком, можно применить метод `get_feature_names`:

```
# Показать имена признаков
count.get_feature_names()
```

```
['бразилия', 'бьет', 'германия', 'лучше', 'любовь', 'моя', 'обоих', 'швеция']
```

Этот результат может сбить с толку, поэтому для ясности в табл. 6.1 показано, как выглядит матрица признаков со словами в качестве имен столбцов (каждая строка является одним наблюдением).

Таблица 6.1

бразилия	бьет	германия	лучше	любовь	моя	обоих	швеция
2	0	0	0	1	1	0	0
0	0	0	1	0	0	0	1
0	1	1	0	0	0	1	0

Обсуждение

Одним из наиболее распространенных методов преобразования текста в признаки является использование модели *мешка слов*. Эти модели выводят признак для каждого уникального слова в текстовых данных, при этом каждый признак содержит

количество вхождений в наблюдениях. Например, в нашем решении предложение `'Бразилия - моя любовь. Бразилия!'` имеет значение 2 в признаке "бразилия", потому что слово *Бразилия* появляется два раза.

Текстовые данные в нашем решении были намеренно малы. В реальном мире одноединственное наблюдение в виде текстовых данных может представлять собой содержимое целой книги! Поскольку наша модель мешка слов создает признак для каждого уникального слова в данных, результирующая матрица может содержать тысячи признаков. Это означает, что размер матрицы иногда может стать очень большим для оперативной памяти. Однако, к счастью, мы можем использовать общую особенность матриц признаков на основе мешка слов, чтобы уменьшить объем данных, которые нам нужно хранить.

Большинство слов, скорее всего, не встречаются в большинстве наблюдений, и поэтому матрицы признаков на основе мешка слов будут в качестве значений содержать в основном нули. Мы называем матрицы этого типа *разреженными*. Вместо того чтобы хранить все значения матрицы, мы можем хранить только ненулевые значения и исходить из того, что все остальные значения равняются 0. При наличии крупных матриц признаков это сэкономит нам оперативную память. Одной из приятных особенностей векторизатора частотностей `CountVectorizer` является то, что результатом по умолчанию будет разреженная матрица.

Класс `CountVectorizer` сопровождается рядом полезных параметров, которые упрощают создание матриц признака на основе мешка слов. Во-первых, хотя по умолчанию каждый признак является словом, это не обязательно. Вместо этого мы можем установить, чтобы каждый признак был комбинацией двух слов (так называемыми 2-граммами) или даже трех слов (3-граммами). Параметр `ngram_range` устанавливает минимальный и максимальный размеры наших *n*-грамм. Например, `(2,3)` вернет все 2-граммы и 3-граммы. Во-вторых, мы можем легко удалить слова с низкой информацией, используя стоп-слова в параметре `stop_words` либо с помощью встроенного списка, либо с помощью собственного списка. Наконец, мы можем ограничить слова или фразы, которые хотим рассмотреть, заданным списком слов, используя параметр `vocabulary`. Например, можно создать матрицу признаков на основе мешка слов только для вхождений названий стран:

```
# Создать матрицу признаков с аргументами
count_2gram = CountVectorizer(ngram_range=(1,2),
                              stop_words="english", # использовать свой список
                              vocabulary=['бразилия'])

bag = count_2gram.fit_transform(text_data)

# Взглянуть на матрицу признаков
bag.toarray()

array([[2],
       [0],
       [0]], dtype=int64)
```

```
# Взглянуть на 1-граммы и 2-граммы
count_2gram.vocabulary_
```

```
{'бразилия': 0}
```

Дополнительные материалы для чтения

- "*N*-грамма", Википедия (**https://bit.ly/2IAjLwt**).
- "Мешок слов встречается со стаканом попкорна", статья на ресурсе для конкурсов по предсказательному моделированию и аналитике Kaggle (**http://bit.ly/2HRba5v**).

6.10. Взвешивание важности слов

Задача

Требуется мешок слов, но со словами, взвешенными по их важности для наблюдения.

Решение

Сравнить частоту слова в документе (твит, обзор видео, стенограмма выступления и т. д.) с частотой слова во всех других документах, используя статистическую меру словарной частоты — обратной документной частоты (tf-idf). Библиотека scikit-learn позволяет легко это делать с помощью класса `TfidfVectorizer`:

```
# Загрузить библиотеки
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer

# Создать текст
text_data = np.array(['Бразилия - моя любовь. Бразилия!',
                      'Швеция - лучше',
                      'Германия бьет обоих'])

# Создать матрицу признаков на основе меры tf-idf
tfidf = TfidfVectorizer()
feature_matrix = tfidf.fit_transform(text_data)

# Показать матрицу признаков на основе меры tf-idf
feature_matrix
```

```
<3x8 sparse matrix of type '<class 'numpy.float64'>'
        with 8 stored elements in Compressed Sparse Row format>
```

Как и в рецепте 6.9, на выходе получается разреженная матрица. Однако, если требуется взглянуть на результат, как на плотную матрицу, можем использовать метод `toarray`:

```
# Показать матрицу признаков на основе меры tf-idf как плотную
feature_matrix.toarray()
```

```
array([[0.81649658, 0.        , 0.        , 0.        , 0.40824829,
        0.40824829, 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.70710678, 0.        ,
        0.        , 0.        , 0.70710678],
       [0.        , 0.57735027, 0.57735027, 0.        , 0.        ,
        0.        , 0.57735027, 0.        ]])
```

Атрибут `vocabulary_` показывает нам слово каждого признака:

```
# Показать имена признаков
tfidf.vocabulary_
```

```
{'бразилия': 0,
 'бьет': 1,
 'германия': 2,
 'лучше': 3,
 'любовь': 4,
 'моя': 5,
 'обоих': 6,
 'швеция': 7}
```

Обсуждение

Чем чаще слово появляется в документе, тем более вероятно, что оно важно для этого документа. Например, если слово "экономика" встречается часто, это свидетельствует о том, что документ может касаться экономики. Мы называем эту меру частотой встречаемости слова в документе, или *словарной частотой* (*tf*).

Напротив, если слово появляется во многих документах, оно, скорее всего, менее важно для любого отдельного документа. Например, если каждый документ в некоторых текстовых данных содержит слово *после*, то это слово, скорее всего, не представляет важность. Мы называем эту меру частотой встречаемости слова во всех документах, или *документной частотой* (*df*).

Объединив эти два статистических показателя, мы можем назначить оценку каждому слову, тем самым показывая, насколько важно это слово в документе. В частности, мы умножаем *tf* на обратную частоту документа (*idf*):

$$tf\text{-}idf(t,d) = tf(t,d) \cdot idf(t,d),$$

где t — слово; d — документ.

Существует ряд отличий в том, как рассчитываются *tf* и *idf*. В библиотеке scikit-learn *tf* — это просто количество раз, когда слово появляется в документе, и *idf* рассчитывается следующим образом:

$$idf(t) = \log\frac{1+n_d}{1+df(d,t)} + 1,$$

где n_d — это количество документов; $df(d, t)$ — документная частота слова t (т. е. количество документов, в которых появляется это слово).

По умолчанию scikit-learn затем нормализует векторы *tf-idf*, используя евклидову норму (норму L^2). Чем выше результирующее значение, тем важнее слово для документа.

Дополнительные материалы для чтения

- Документация библиотеки scikit-learn: взвешивание на основе весового коэффициента tf-idf (**http://bit.ly/2HT2wmW**).

Работа с датами и временем

Введение

Даты и время (тип `datetime`) часто встречаются во время предобработки данных для машинного самообучения, будь то время конкретной продажи или год сбора каких-либо статистических данных общественного здравоохранения. В этой главе мы соберем инструментальный набор стратегий для обработки данных временных рядов, включая решение проблемы, связанной с часовыми поясами, и создание признаков с запаздыванием во времени. В частности, мы сконцентрируемся на инструментах временных рядов в библиотеке pandas, которая сосредотачивает в себе функциональность многих других библиотек.

7.1. Конвертирование строковых значений в даты

Задача

Дан вектор строк, представляющий даты и время, и требуется преобразовать их в данные временных рядов.

Решение

Использовать параметр `to_datetime` библиотеки pandas с форматом даты и/или времени, указанным в параметре `format`:

```
# Загрузить библиотеки
import numpy as np
import pandas as pd

# Создать строки
date_strings = np.array(['03-04-2005 11:35 PM',
                         '23-05-2010 12:01 AM',
                         '04-09-2009 09:09 PM'])

# Конвертировать в метки datetime
[pd.to_datetime(date, format='%d-%m-%Y %I:%M %p') for date in date_strings]
```

```
[Timestamp('2005-04-03 23:35:00'),
 Timestamp('2010-05-23 00:01:00'),
 Timestamp('2009-09-04 21:09:00')]
```

Возможно, также возникнет потребность добавить аргумент в параметр `errors` для устранения проблем:

```
# Конвертировать в метки datetime
[pd.to_datetime(date, format="%d-%m-%Y %I:%M %p", errors="coerce")
for date in date_strings]

[Timestamp('2005-04-03 23:35:00'),
 Timestamp('2010-05-23 00:01:00'),
 Timestamp('2009-09-04 21:09:00')]
```

Если задан параметр `errors="coerce"`, то любая возникающая проблема не вызовет ошибку (поведение по умолчанию), а вместо этого установит значение, вызывающее ошибку, равным `NaT` (от англ. *Not a Time* — пропущенное значение).

Обсуждение

Когда даты и время поступают в виде строковых значений, требуется их преобразовать в тип данных, который Python сможет понять. Хотя существует ряд инструментов Python для преобразования строк в тип `datetime`, следуя нашему использованию библиотеки pandas в других рецептах для преобразования данных, мы можем применить ее функцию `to_datetime`. Одним из препятствий для строк, представляющих даты и времена, является то, что формат строк может существенно различаться от источника к источнику данных. Например, один вектор дат может представлять 23 марта 2018 года как "03-23-18", в то время как другой может использовать "3/23/2018". Для указания точного формата строкового значения можно использовать параметр `format`. Приведем некоторые общепринятые коды форматирования даты и времени (табл. 7.1).

Таблица 7.1

Код	Описание	Пример
%Y	Полный год	2001
%m	Месяц с дополнением нулем	04
%d	День месяца с дополнением нулем	09
%I	Час (12-часовое измерение) с дополнением нулем	02
%p	AM (до полудня) или PM (после полудня)	AM
%M	Минута с дополнением нулем	05
%S	Секунда с дополнением нулем	09

Дополнительные материалы для чтения

- Полный список строковых временных кодов Python (**http://strftime.org**).

7.2. Обработка часовых поясов

Задача

Дан временной ряд, и требуется добавить или изменить информацию о часовом поясе.

Решение

Если не указано иное, объекты pandas часового пояса не имеют. Вместе с тем мы можем добавить часовой пояс с помощью параметра `tz` во время создания:

```
# Загрузить библиотеку
import pandas as pd

# Создать метку datetime
pd.Timestamp('2017-05-01 06:00:00', tz='Europe/London')

Timestamp('2017-05-01 06:00:00+0100', tz='Europe/London')
```

Часовой пояс можно добавить к ранее созданной метке вермени `datetime` с помощью метода `tz_localize`:

```
# Создать метку datetime
date = pd.Timestamp('2017-05-01 06:00:00')

# Задать часовой пояс
date_in_london = date.tz_localize('Europe/London')

# Показать метку datetime
date_in_london

Timestamp('2017-05-01 06:00:00+0100', tz='Europe/London')
```

Мы также можем выполнить преобразование в другой часовой пояс:

```
# Изменить часовой пояс
date_in_london.tz_convert('Africa/Abidjan')

Timestamp('2017-05-01 05:00:00+0000', tz='Africa/Abidjan')
```

Наконец, одномерные объекты Series библиотеки pandas могут применять методы `tz_localize` и `tz_convert` к каждому элементу:

```
# Создать три даты
dates = pd.Series(pd.date_range('2/2/2002', periods=3, freq='M'))

# Задать часовой пояс
dates.dt.tz_localize('Africa/Abidjan')
```

```
0   2002-02-28 00:00:00+00:00
1   2002-03-31 00:00:00+00:00
2   2002-04-30 00:00:00+00:00
dtype: datetime64[ns, Africa/Abidjan]
```

Обсуждение

Библиотека pandas поддерживает два набора строковых значений, представляющих часовые пояса; однако я предлагаю использовать строки библиотеки pytz. Импортировав массив `all_timezones`, можно увидеть все строки, используемые для представления часовых поясов:

```
# Загрузить библиотеку
from pytz import all_timezones

# Показать два часовых пояса
all_timezones[0:2]
```

```
['Africa/Abidjan', 'Africa/Accra']
```

7.3. Выбор дат и времени

Задача

Дан вектор дат, и требуется выбрать одну дату или несколько.

Решение

Использовать два булевых условия в качестве начальной и конечной дат (табл. 7.2):

```
# Загрузить библиотеку
import pandas as pd

# Создать фрейм данных
dataframe = pd.DataFrame()

# Создать метки datetime
dataframe['дата'] = pd.date_range('1/1/2001', periods=100000, freq='H')

# Выбрать наблюдения между двумя метками datetime
dataframe[(dataframe['дата'] > '2002-1-1 01:00:00') &
(dataframe['дата'] <= '2002-1-1 04:00:00')]
```

Таблица 7.2

	дата
8762	2002-01-01 02:00:00
8763	2002-01-01 03:00:00
8764	2002-01-01 04:00:00

В качестве альтернативы мы можем установить столбец даты как индекс фрейма данных, а затем сделать срез с помощью метода `loc` (табл. 7.3):

```
# Задать индекс
dataframe = dataframe.set_index(dataframe['дата'])

# Выбрать наблюдения между двумя метками datetime
dataframe.loc['2002-1-1 01:00:00':'2002-1-1 04:00:00']
```

Таблица 7.3

дата	дата
2002-01-01 01:00:00	2002-01-01 01:00:00
2002-01-01 02:00:00	2002-01-01 02:00:00
2002-01-01 03:00:00	2002-01-01 03:00:00
2002-01-01 04:00:00	2002-01-01 04:00:00

Обсуждение

Какую стратегию использовать — булевы условия либо нарезку по индексу, определяет ситуация. Если требуется выполнить некие сложные манипуляции с временными рядами, установка столбца даты в качестве индекса фрейма данных может вылиться в накладные расходы, но если нам нужно сделать простую проверку данных, то булевы условия могут оказаться проще.

7.4. Разбиение данных даты на несколько признаков

Задача

Дан столбец дат и времен, и требуется создать признаки для года, месяца, дня, часа и минуты.

Решение

Использовать свойства времени объекта-получателя `Series.dt` библиотеки pandas (табл. 7.4):

```
# Загрузить библиотеку
import pandas as pd

# Создать фрейм данных
dataframe = pd.DataFrame()
```

```
# Создать пять дат
dataframe['дата'] = pd.date_range('1/1/2001', periods=150, freq='W')

# Создать признаки для года, месяца, дня, часа и минуты
dataframe['год']    = dataframe['дата'].dt.year
dataframe['месяц']  = dataframe['дата'].dt.month
dataframe['день']   = dataframe['дата'].dt.day
dataframe['час']    = dataframe['дата'].dt.hour
dataframe['минута'] = dataframe['дата'].dt.minute

# Показать три строки фрейма
dataframe.head(3)
```

Таблица 7.4

	дата	год	месяц	день	час	минута
0	2001-01-07	2001	1	7	0	0
1	2001-01-14	2001	1	14	0	0
2	2001-01-21	2001	1	21	0	0

Обсуждение

Иногда бывает полезно разбить столбец дат на компоненты. Например, может понадобиться признак, который включает только год наблюдения, либо мы можем рассматривать только месяц некоторого наблюдения, чтобы их сравнивать независимо от года.

7.5. Вычисление разницы между датами

Задача

Даны два признака `datetime`, и требуется для каждого наблюдения рассчитать время между ними.

Решение

Вычесть один признак `datetime` из другого с помощью библиотеки pandas:

```
# Загрузить библиотеку
import pandas as pd

# Создать фрейм данных
dataframe = pd.DataFrame()

# Создать два признака datetime
dataframe['Прибыло'] = [pd.Timestamp('01-01-2017'), pd.Timestamp('01-04-2017')]
dataframe['Осталось'] = [pd.Timestamp('01-01-2017'), pd.Timestamp('01-06-2017')]
```

```
# Вычислить продолжительность между признаками
dataframe['Осталось'] - dataframe['Прибыло']
```

```
0   0 days
1   2 days
dtype: timedelta64[ns]
```

Часто требуется удалить из результата `days` и оставить только числовое значение:

```
# Вычислить продолжительность между признаками
pd.Series(delta.days
          for delta in (dataframe['Осталось'] - dataframe['Прибыло']))
```

```
0    0
1    2
dtype: int64
```

Обсуждение

Бывают случаи, когда требуемый признак является разностью (дельтой) между двумя точками во времени. Например, мы можем иметь даты заезда и выезда клиента из гостиницы, но признак, который нам требуется, — это продолжительность его пребывания. Библиотека pandas упрощает этот расчет, используя тип данных `TimeDelta`.

Дополнительные материалы для чтения

- **Документация pandas: дельта времени (http://bit.ly/2HOS3sJ).**

7.6. Кодирование дней недели

Задача

Дан вектор дат, и требуется узнать день недели для каждой даты.

Решение

Использовать атрибут `weekday_name` объекта-получателя `Series.dt` библиотеки pandas:

```
# Загрузить библиотеку
import pandas as pd

# Создать даты
dates = pd.Series(pd.date_range("2/2/2002", periods=3, freq="M"))

# Показать дни недели
dates.dt.weekday_name
```

```
0    Thursday
1      Sunday
2     Tuesday
dtype: object
```

Если мы хотим, чтобы результат был числовым значением и, следовательно, более пригодным для использования в качестве признака для машинного самообучения, мы можем использовать атрибут `weekday`, где дни недели представлены в виде целого числа (понедельник равен 0):

```
# Показать дни недели
dates.dt.weekday
```

```
0    3
1    6
2    1
dtype: int64
```

Обсуждение

Знание дня недели может быть полезным, если, например, требуется сравнить общий объем продаж по воскресеньям за последние три года. Библиотека pandas упрощает создание вектора признаков, содержащего информацию о будних днях.

Дополнительные материалы для чтения

- Перечень `datetime`-подобных свойств объекта Series библиотеки pandas (**http://bit.ly/2HShhq1**).

7.7. Создание запаздывающего признака

Задача

Требуется создать признак, который запаздывает на *n* периодов времени.

Решение

Использовать метод `shift` библиотеки pandas (табл. 7.5):

```
# Загрузить библиотеку
import pandas as pd

# Создать фрейм данных
dataframe = pd.DataFrame()

# Создать дату
dataframe["даты"] = pd.date_range("1/1/2001", periods=5, freq="D")
dataframe["цена_акций"] = [1.1,2.2,3.3,4.4,5.5]
```

```
# Значения с запаздыванием на одну строку
dataframe["цена_акций_в_предыдущий_день"] = dataframe["цена_акций"].shift(1)

# Показать фрейм данных
dataframe
```

Таблица 7.5

	даты	цена_акций	цена_акций_в_предыдущий_день
0	2001-01-01	1.1	NaN
1	2001-01-02	2.2	1.1
2	2001-01-03	3.3	2.2
3	2001-01-04	4.4	3.3
4	2001-01-05	5.5	4.4

Обсуждение

Очень часто данные основаны на регулярных интервалах времени (например, каждый день, каждый час, каждые три часа), и чтобы делать предсказания, мы заинтересованы в использовании значений в прошлом (при этом нередко используется выражение *"задерживание признака во времени"*). Например, может возникнуть необходимость предсказать цену акций на основе цены, которая была накануне. Можно применить метод `shift` из библиотеки pandas для задерживания значений во времени на одну строку, создавая новый признак, содержащий прошлые значения.

В нашем решении первая строка для поля `цена_акций_в_предыдущий_день` является пропущенным значением, поскольку предыдущее значение цены акций `stock_price` отсутствует.

7.8. Использование скользящих временных окон

Задача

Дан временной ряд, и требуется рассчитать некоторый статистический показатель для скользящего времени.

Решение

```
# Загрузить библиотеку
import pandas as pd

# Создать метки datetime
time_index = pd.date_range("01/01/2010", periods=5, freq="M")
```

```
# Создать фрейм данных, задать индекс
dataframe = pd.DataFrame(index=time_index)

# Создать признак
dataframe["цена_акций"] = [1,2,3,4,5]

# Вычислить скользящее среднее
dataframe.rolling(window=2).mean()
```

Таблица 7.6

	цена_акций
2010-01-31	NaN
2010-02-28	1.5
2010-03-31	2.5
2010-04-30	3.5
2010-05-31	4.5

Обсуждение

Скользящие (так называемые перемещающиеся) временные окна концептуально просты, но поначалу могут быть трудны для понимания. Представьте, что имеются ежемесячные наблюдения за ценой акций. Часто бывает полезно иметь временно́е окно продолжительностью в заданное количество месяцев, а затем перемещаться по наблюдениям, вычисляя статистический показатель для всех наблюдений во временном окне.

Например, если есть временное окно продолжительностью три месяца, и требуется получить скользящее среднее, то мы вычисляем:

Среднее(январь, февраль, март).

Среднее(февраль, март, апрель).

Среднее(март, апрель, май).

И т. д.

Другими словами, наше трехмесячное временное окно "скользит" по наблюдениям, вычисляя среднее значение окна на каждом шаге.

Метод `rolling` библиотеки pandas позволяет задавать размер окна с помощью параметра `window`, а затем быстро рассчитывать некоторые распространенные статистические показатели, в том числе максимальное значение (`max()`), среднее значение (`mean()`), количество значений (`count()`) и скользящую корреляцию (`corr()`).

Скользящие средние часто используются для сглаживания данных временных рядов, поскольку использование среднего значения всего временного окна ослабляет эффект краткосрочных колебаний.

Дополнительные материалы для чтения

- Документация pandas: скользящие окна (**http://bit.ly/2HTIicA**).
- "Что такое скользящее среднее и методы сглаживания", специализированный ресурс "Справочник по инженерной статистике" (Engineering Statistics Handbook, **http://bit.ly/2HRRHBn**).

7.9. Обработка пропущенных дат во временном ряду

Задача

В данных временных рядов пропущены значения. Обработать эту ситуацию.

Решение

В дополнение к ранее обсуждавшимся стратегиям обработки пропущенных данных в случае данных временных рядов можно использовать интерполяцию, которая позволяет заполнять промежутки, вызванные пропущенными значениями (табл. 7.7):

```
# Загрузить библиотеки
import pandas as pd
import numpy as np

# Создать дату
time_index = pd.date_range("01/01/2010", periods=5, freq="M")

# Создать фрейм данных, задать индекс
dataframe = pd.DataFrame(index=time_index)

# Создать признак с промежутком пропущенных значений
dataframe["продажи"] = [1.0,2.0,np.nan,np.nan,5.0]

# Интерполировать пропущенные значения
dataframe.interpolate()
```

Таблица 7.7

	продажи
2010-01-31	1.0
2010-02-28	2.0
2010-03-31	3.0
2010-04-30	4.0
2010-05-31	5.0

В качестве альтернативы можем заменить пропущенные значения последним перед промежутком известным значением (т. е. выполнить прямое заполнение), табл. 7.8:

```
# Прямое заполнение
dataframe.ffill()
```

Таблица 7.8

	продажи
2010-01-31	1.0
2010-02-28	2.0
2010-03-31	2.0
2010-04-30	2.0
2010-05-31	5.0

Мы также можем заменить пропущенные значения последним после промежутка известным значением (т. е. выполнить обратное заполнение), табл. 7.9:

```
# Обратное заполнение
dataframe.bfill()
```

Таблица 7.9

	продажи
2010-01-31	1.0
2010-02-28	2.0
2010-03-31	5.0
2010-04-30	5.0
2010-05-31	5.0

Обсуждение

Интерполяция — это метод заполнения промежутков, вызванных пропущенными значениями, путем, по сути, построения прямой линии или кривой между известными значениями, граничащими с промежутком, и использования этой прямой или кривой для предсказания разумных значений. Интерполяция может быть в особенности полезна, когда промежутки времени между точками постоянны, данные не подвержены шумовым колебаниям, а промежутки, вызванные пропущенными значениями, невелики. Например, в нашем решении промежуток из двух пропущенных значений граничит с `2.0` и `5.0`. Путем подгонки прямой, начинающейся с `2.0` и заканчивающейся в `5.0`, можно сделать разумные предположения для двух пропущенных значений между `3.0` и `4.0`.

Если мы считаем, что прямая между двумя известными точками нелинейна, то можно применить метод `interpolate` для задания метода интерполяции (табл. 7.10):

```
# Интерполировать пропущенные значения
dataframe.interpolate(method="quadratic")
```

Таблица 7.10

	продажи
2010-01-31	1.000000
2010-02-28	2.000000
2010-03-31	3.059808
2010-04-30	4.038069
2010-05-31	5.000000

Наконец, могут быть случаи, когда имеются большие промежутки пропущенных значений, и мы не хотим интерполировать значения по всему промежутку. Тогда можно использовать параметр `limit` для ограничения количества интерполируемых значений и параметр `limit_direction` для задания, следует ли интерполировать значения вперед от последнего перед промежутком известного значения, или наоборот (табл. 7.11):

```
# Интерполировать пропущенные значения
dataframe.interpolate(limit=1, limit_direction="forward")
```

Таблица 7.11

	продажи
2010-01-31	1.0
2010-02-28	2.0
2010-03-31	3.0
2010-04-30	NaN
2010-05-31	5.0

Обратное заполнение и прямое заполнение можно рассматривать как формы наивной интерполяции, где мы рисуем плоскую прямую из известного значения и используем ее для заполнения пропущенных значений. Одно (незначительное) преимущество обратного и прямого заполнения перед интерполяцией заключается в отсутствии необходимости в известных значениях с *обеих* сторон пропущенных значений.

Работа с изображениями

Введение

Классификация изображений — одна из самых интересных областей машинного самообучения. Способность компьютеров распознавать шаблоны и объекты на изображениях является невероятно мощным инструментом в нашем арсенале. Однако, прежде чем применить машинное самообучение к изображениям, нередко сначала нужно преобразовать сырые изображения в признаки, используемые нашими обучающимися алгоритмами.

Для работы с изображениями мы будем применять библиотеку компьютерного зрения с открытым исходным кодом (OpenCV). Хотя существует ряд других хороших библиотек, библиотека OpenCV — самая популярная и документированная библиотека для обработки изображений. Одним из самых больших препятствий для использования OpenCV является ее инсталляция. Однако, к счастью, если мы работаем с Python 3 (на момент публикации OpenCV не работает с Python 3.6+), можно использовать инструмент диспетчера пакетов Anaconda — conda — и инсталлировать библиотеку OpenCV в одной строке кода в своем терминале[1]:

```
conda install --channel https://conda.anaconda.org/menpo opencv3
```

После этого можно проверить инсталляцию, открыв консоль Python или блокнот Jupyter, импортировав OpenCV и проверив номер версии (3.1.0):

```
import cv2

cv2.__version__
```

Если инсталляция OpenCV с помощью conda не работает, в Интернете есть много руководств по установке данной библиотеки.

Наконец, в качестве примеров в этой главе мы будем использовать набор изображений, которые доступны для загрузки на GitHub (**https://github.com/chrisalbon/simulated_datasets**).

[1] Для инсталляции библиотеки OpenCV в Windows 10 надо скачать whl-файл из **https://www.lfd.uci.edu/~gohlke/pythonlibs/#opencv** и установить при помощи обычного менеджера пакетов `pip`. Например, установка версии библиотеки для Python 3.6+ и 64-разрядной ОС будет следующей: `pip install opencv_python-3.4.1-cp36-cp36m-win_amd64.whl`. — *Прим. перев.*

8.1. Загрузка изображений

Задача

Требуется загрузить изображение для предобработки.

Решение

Использовать функцию `imread` библиотеки OpenCV:

```
# Загрузить библиотеки
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Загрузить изображение в оттенках серого
image = cv2.imread("images/plane.jpg", cv2.IMREAD_GRAYSCALE)
```

Если требуется просмотреть изображение, для этого можно использовать библиотеку Python matplotlib (рис. 8.1):

```
# Показать изображение
plt.imshow(image, cmap="gray")
plt.axis("off")
plt.show()
```

Рис. 8.1

Обсуждение

В свой основе изображения — это данные, и когда мы используем функцию `imread`, мы преобразуем эти данные в тип данных, с которым мы уже знакомы, — массив NumPy:

```
# Показать тип данных
type(image)
```

```
numpy.ndarray
```

Мы превратили изображение в матрицу, элементы которой соответствуют отдельным пикселам. Мы даже можем взглянуть на фактические значения матрицы:

```
# Показать данные изображения
image
```

```
array([[140, 136, 146, ..., 132, 139, 134],
[144, 136, 149, ..., 142, 124, 126],
[152, 139, 144, ..., 121, 127, 134],
...,
[156, 146, 144, ..., 157, 154, 151],
[146, 150, 147, ..., 156, 158, 157],
[143, 138, 147, ..., 156, 157, 157]], dtype=uint8)
```

Разрешающая способность нашего изображения составляет 3600×2270, точные размеры нашей матрицы:

```
# Показать размерности
image.shape
```

```
(2270, 3600)
```

Что на самом деле представляет собой каждый элемент матрицы? В полутоновых изображениях значением отдельного элемента является интенсивность пикселов. Значения интенсивности варьируются от черного (0) до белого (255). Например, интенсивность правого верхнего пиксела изображения имеет значение 140:

```
# Показать первый пиксел
image[0,0]
```

```
140
```

В матрице цветного изображения каждый элемент содержит три значения, соответствующие синему, зеленому и красному цветам (BGR):

```
# Загрузить цветное изображение
image_bgr = cv2.imread("images/plane.jpg", cv2.IMREAD_COLOR)

# Показать пиксел
image_bgr[0,0]
```

```
array([195, 144, 111], dtype=uint8)
```

Один небольшой нюанс: по умолчанию в библиотеке OpenCV используется цветовая схема BGR, но многие графические приложения — включая matplotlib — работают с цветовой схемой красный-зеленый-синий (RGB), т. е. красные и синие зна-

чения меняются местами. Для того чтобы правильно использовать цветные изображения OpenCV в matplotlib, нам нужно сначала преобразовать цвет в схему RGB (приносим извинения читателям печатного издания книги), рис. 8.2:

```
# Конвертировать в RGB
image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB)

# Показать изображение
plt.imshow(image_rgb)
plt.axis("off")
plt.show()
```

Рис. 8.2

Дополнительные материалы для чтения

- "Разница между цветовыми схемами RGB и BGR", блог-пост (**http://bit.ly/2Fws76E**).
- "Цветовая модель RGB", Википедия (**http://bit.ly/2FxZjKZ**).

8.2. Сохранение изображений

Задача

Требуется сохранить изображение для предобработки.

Решение

Использовать функцию `imwrite` библиотеки OpenCV:

```
# Загрузить библиотеки
import cv2
```

```
import numpy as np
from matplotlib import pyplot as plt

# Загрузить изображение в оттенках серого
image = cv2.imread("images/plane.jpg", cv2.IMREAD_GRAYSCALE)

# Сохранить изображение
cv2.imwrite("images/plane_new.jpg", image)

True
```

Обсуждение

Функция `imwrite` библиотеки OpenCV сохраняет изображения в файле по указанному пути. Формат изображения определяется расширением файла (jpg, png и т. д.). Однако следует быть осторожным: функция `imwrite` пишет поверх существующих файлов без вывода ошибки или запроса подтверждения.

8.3. Изменение размера изображений

Задача

Требуется изменить размер изображения для дальнейшей предобработки.

Решение

Использовать функцию `resize` для изменения размера изображения (рис. 8.3):

```
# Загрузить библиотеки
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Загрузить изображение в оттенках серого
image = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# Изменить размер изображения до 50 пикселов на 50 пикселов
image_50x50 = cv2.resize(image, (50, 50))

# Взглянуть на изображение
plt.imshow(image_50x50, cmap="gray")
plt.axis("off")
plt.show()
```

Рис. 8.3

Обсуждение

Изменение размера изображений является распространенной задачей в предобработке изображений по двум причинам. Во-первых, изображения бывают всех форм и размеров, и для использования в качестве признаков изображения должны иметь одинаковые размеры. Однако эта стандартизация размера сопряжена с затратами; изображения являются матрицами информации, и когда мы уменьшаем размер, мы уменьшаем размер этой матрицы и сокращаем информацию, которую она содержит. Во-вторых, машинное самообучение может потребовать тысячи или сотни тысяч изображений. Когда эти изображения очень большие, они могут занимать много оперативной памяти, и, изменив их размер, мы можем значительно освободить память. Некоторые распространенные размеры изображения для машинного самообучения составляют 32×32, 64×64, 96×96 и 256×256.

8.4. Обрезка изображений

Задача

Требуется удалить внешнюю часть изображения, чтобы изменить его размеры.

Решение

Изображение кодируется как двумерный массив NumPy, поэтому мы можем легко отсечь кромки изображения путем нарезки массива (рис. 8.4):

```
# Загрузить библиотеки
import cv2
import numpy as np
from matplotlib import pyplot as plt
```

```
# Загрузить изображение в оттенках серого
image = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# Выбрать первую половину столбцов и все строки
image_cropped = image[:,:128]

# Показать изображение
plt.imshow(image_cropped, cmap="gray")
plt.axis("off")
plt.show()
```

Рис. 8.4

Обсуждение

Поскольку библиотека OpenCV представляет изображения в виде матрицы элементов, выбирая строки и столбцы, которые требуется сохранить, изображение можно легко обрезать. Обрезка может быть особенно полезной, если мы знаем, что требуется сохранить только определенную часть каждого изображения.

Например, если изображения поступают со стационарной камеры видеонаблюдения, мы можем обрезать все изображения так, чтобы они содержали только интересующую область.

Дополнительные материалы для чтения

- Нарезка массивов NumPy (**http://bit.ly/2FrVNBV**).

8.5. Размытие изображений

Задача

Требуется сгладить изображение.

Решение

Для того чтобы размыть изображение, каждый пиксел нужно преобразовать в среднее значение его соседей. Этот сосед и выполняемая операция математически представлены в виде ядра (не переживайте, если не знаете, что такое ядро). Размер ядра определяет степень размытости, при этом более крупные ядра создают более гладкие изображения. Здесь мы размываем изображение, усредняя значения ядра 5×5 вокруг каждого пиксела (рис. 8.5):

```
# Загрузить библиотеки
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Загрузить изображение в оттенках серого
image = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# Размыть изображение
image_blurry = cv2.blur(image, (5,5))

# Показать изображение
plt.imshow(image_blurry, cmap="gray")
plt.axis("off")
plt.show()
```

Рис. 8.5

Для того чтобы подчеркнуть влияние размера ядра на степень размытости, ниже приведено то же самое изображение с ядром размытия 100×100 (рис. 8.6):

```
# Размыть изображение
image_very_blurry = cv2.blur(image, (100,100))

# Показать изображение
plt.imshow(image_very_blurry, cmap="gray")
plt.xticks([])
plt.yticks([])
plt.show()
```

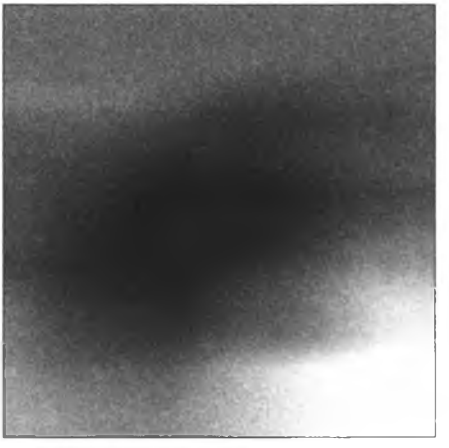


Рис. 8.6

Обсуждение

Ядра широко используются в обработке изображений, позволяя делать все, от повышения резкости до обнаружения краев, и будут неоднократно упоминаться в этой главе. Использованное нами размывающее ядро выглядит следующим образом:

```
# Создать ядро
kernel = np.ones((5,5)) / 25.0

# Показать ядро
kernel

array([[0.04, 0.04, 0.04, 0.04, 0.04],
       [0.04, 0.04, 0.04, 0.04, 0.04],
       [0.04, 0.04, 0.04, 0.04, 0.04],
       [0.04, 0.04, 0.04, 0.04, 0.04],
       [0.04, 0.04, 0.04, 0.04, 0.04]])
```

Центральным элементом в ядре является исследуемый пиксел, а остальные — его соседи. Поскольку все элементы имеют одинаковое значение (нормализованные, давая в сумме 1), каждый из них имеет равный голос в результирующем значении интересующего пиксела. Ядро можно применить к изображению вручную с помощью фильтра `filter2D` и создать аналогичный эффект размытия (рис. 8.7):

```
# Применить ядро
image_kernel = cv2.filter2D(image, -1, kernel)

# Показать изображение
plt.imshow(image_kernel, cmap="gray")
plt.xticks([])
plt.yticks([])
plt.show()
```

Рис. 8.7

Дополнительные материалы для чтения

- "Ядра изображений с визуальным объяснением", математический проект Explained Visually (**https://bit.ly/2b8xgB4**).
- "Распространенные ядра изображений", Википедия (**http://bit.ly/2FxZCFD**).

8.6. Увеличение резкости изображений

Задача

Требуется увеличить резкость изображения.

Решение

Создать ядро, выделяющее целевой пиксел. Затем применить его к изображению с помощью фильтра `filter2D` (рис. 8.8):

```
# Загрузить библиотеки
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Загрузить изображение в оттенках серого
image = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# Создать ядро
kernel = np.array([[0, -1, 0],
                   [-1, 5,-1],
                   [0, -1, 0]])

# Увеличить резкость изображения
image_sharp = cv2.filter2D(image, -1, kernel)

# Показать изображение
plt.imshow(image_sharp, cmap="gray")
plt.axis("off")
plt.show()
```

Рис. 8.8

Обсуждение

Резкость работает аналогично размытию, за исключением того, что вместо использования ядра для усреднения соседних значений мы построили ядро, чтобы выделить сам пиксел. Полученный эффект делает контрасты в краях более заметными на изображении.

8.7. Усиление контрастности

Задача

Требуется усилить контрастность между пикселами изображения.

Решение

Выравнивание гистограммы — это инструмент обработки изображений, который может выделять объекты и фигуры. При наличии полутонового изображения можно применить функцию `equalizeHist` библиотеки OpenCV непосредственно на изображении (рис. 8.9):

```
# Загрузить библиотеки
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Загрузить изображение в оттенках серого
image = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# Улучшить изображение
image_enhanced = cv2.equalizeHist(image)

# Показать изображение
plt.imshow(image_enhanced, cmap="gray")
plt.axis("off")
plt.show()
```

Рис. 8.9

Однако когда изображение — цветное, сначала нужно преобразовать изображение в цветовой формат YUV: Y — это яркость, а U и V обозначают цвет. После преобразования можно применить к изображению функцию выравнивания гистограммы `equalizeHist`, а затем преобразовать его обратно в BGR или RGB (рис. 8.10):

```
# Загрузить изображение
image_bgr = cv2.imread("images/plane.jpg")

# Конвертировать в YUV
image_yuv = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2YUV)

# Применить выравнивание гистограммы
image_yuv[:, :, 0] = cv2.equalizeHist(image_yuv[:, :, 0])

# Конвертировать в RGB
image_rgb = cv2.cvtColor(image_yuv, cv2.COLOR_YUV2RGB)

# Показать изображение
plt.imshow(image_rgb)
plt.axis("off")
plt.show()
```

Рис. 8.10

Обсуждение

Хотя подробное описание того, как работает выравнивание гистограммы, выходит за рамки этой книги, краткое объяснение заключается в том, что выравнивание преобразует изображение так, чтобы можно было использовать более широкий диапазон интенсивностей пикселов.

Хотя результирующее изображение часто не выглядит "реалистичным", нам нужно помнить, что изображение — это просто визуальное представление лежащих в основе данных. Если выравнивание гистограммы может сделать интересующие объекты более отличимыми от других объектов или фонов (что не всегда имеет место), то это может быть ценным дополнением к нашему конвейеру предобработки изображений.

8.8. Выделение цвета

Задача

Требуется выделить в изображении цвет.

Решение

Определить диапазон цветов, а затем применить маску к изображению (рис. 8.11, 8.12):

```
# Загрузить библиотеки
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Загрузить изображение
image_bgr = cv2.imread('images/plane_256x256.jpg')

# Конвертировать BGR в HSV
image_hsv = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2HSV)

# Определить диапазон синих значений в HSV
lower_blue = np.array([50,100,50])
upper_blue = np.array([130,255,255])

# Создать маску
mask = cv2.inRange(image_hsv, lower_blue, upper_blue)

# Наложить маску на изображение
image_bgr_masked = cv2.bitwise_and(image_bgr, image_bgr, mask=mask)

# Конвертировать BGR в RGB
image_rgb = cv2.cvtColor(image_bgr_masked, cv2.COLOR_BGR2RGB)

# Показать изображение
plt.imshow(image_rgb)
plt.axis("off")
plt.show()
```

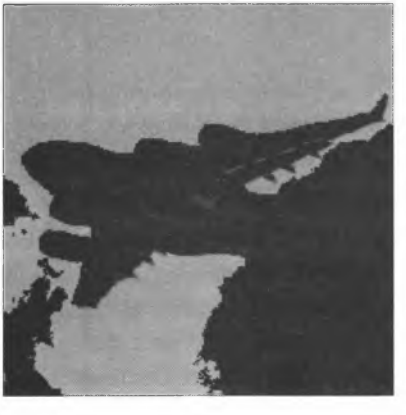


Рис. 8.11

Рис. 8.12

Обсуждение

Выделение цвета в OpenCV выполняется просто. Во-первых, мы преобразуем изображение в HSV (оттенок, насыщенность и значение цвета). Во-вторых, определяем диапазон значений, которые требуется выделить, что, вероятно, является наиболее сложной и трудоемкой частью. В-третьих, создаем маску для изображения (мы оставим только белые области):

```
# Показать изображение
plt.imshow(mask, cmap='gray')
plt.axis("off")
plt.show()
```

Наконец, применяем маску к изображению с помощью функции `bitwise_and` и конвертируем в нужный нам выходной формат.

8.9. Бинаризация изображений

Задача

Дано изображение, и требуется вывести его упрощенную версию.

Решение

Пороговая обработка (порогование) — это процесс установки пикселов с большей интенсивностью, чем некоторое значение, в белый цвет и меньшей — в черный цвет. Более продвинутым методом является *адаптивная пороговая обработка*, где пороговое значение пиксела определяется интенсивностью его соседей. Это может быть полезно при изменении условий освещенности в разных участках изображения (рис. 8.13):

```
# Загрузить библиотеки
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Загрузить изображение в оттенках серого
image_grey = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# Применить адаптивную пороговую обработку
max_output_value = 255
neighborhood_size = 99
subtract_from_mean = 10
image_binarized = cv2.adaptiveThreshold(image_grey,
                                        max_output_value,
                                        cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                                        cv2.THRESH_BINARY,
                                        neighborhood_size,
                                        subtract_from_mean)
```

```
# Показать изображение
plt.imshow(image_binarized, cmap="gray")
plt.axis("off")
plt.show()
```

Рис. 8.13

Обсуждение

Наше решение имеет четыре важных аргумента в функции `adaptiveThreshold`. Аргумент `max_output_value` просто определяет максимальную интенсивность среди интенсивностей выходных пикселов. Аргумент `cv2.ADAPTIVE_THRESH_GAUSSIAN_C` устанавливает порог пиксела как взвешенную сумму интенсивностей соседних пикселов. Веса определяются гауссовым окном. В качестве альтернативы с помощью аргумента `cv2.ADAPTIVE_THRESH_MEAN_C` можно установить порог как просто среднее значение соседних пикселов (рис. 8.14):

```
# Применить cv2.ADAPTIVE_THRESH_MEAN_C
image_mean_threshold = cv2.adaptiveThreshold(image_grey,
                                             max_output_value,
                                             cv2.ADAPTIVE_THRESH_MEAN_C,
                                             cv2.THRESH_BINARY,
                                             neighborhood_size,
                                             subtract_from_mean)

# Показать изображение
plt.imshow(image_mean_threshold, cmap="gray")
plt.axis("off")
plt.show()
```

Последние два параметра — это размер блока (размер окрестности, используемой для определения порога пиксела) и константа, вычитаемая из вычисленного порога (используется для ручной точной настройки порога).

Рис. 8.14

Одним из основных преимуществ пороговой обработки является *шумоподавление* изображения — оставление только наиболее важных элементов. Например, пороговая обработка часто применяется к фотографиям печатного текста, чтобы выделить буквы на странице.

8.10. Удаление фонов

Задача

Требуется выделить на изображении передний план.

Решение

Отметить прямоугольник вокруг нужного переднего плана, затем выполнить алгоритм захвата и обрезки GrabCut (рис. 8.15):

```
# Загрузить библиотеки
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Загрузить изображение и конвертировать в RGB
image_bgr = cv2.imread('images/plane_256x256.jpg')
image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB)

# Значения прямоугольника: начальная X, начальная Y, ширина, высота
rectangle = (0, 56, 256, 150)

# Создать первоначальную маску
mask = np.zeros(image_rgb.shape[:2], np.uint8)
```

```
# Создать временные массивы, используемые в grabCut
bgdModel = np.zeros((1, 65), np.float64)
fgdModel = np.zeros((1, 65), np.float64)

# Выполнить алгоритм grabCut
cv2.grabCut(image_rgb, # Наше изображение
            mask,      # Маска
            rectangle, # Наш прямоугольник
            bgdModel,  # Временный массив для фона
            fgdModel,  # Временный массив для переднего плана
            5,         # Количество итераций
            cv2.GC_INIT_WITH_RECT) # Инициализировать, используя прямоугольник

# Создать маску, где фоны уверенно или потенциально равны 0, иначе 1
mask_2 = np.where((mask==2) | (mask==0), 0, 1).astype('uint8')

# Умножить изображение на новую маску, чтобы вычесть фон
image_rgb_nobg = image_rgb * mask_2[:, :, np.newaxis]

# Показать изображение
plt.imshow(image_rgb_nobg)
plt.axis("off")
plt.show()
```

Рис. 8.15

Обсуждение

Первое, что обращает на себя внимание, — алгоритм GrabCut неплохо справился с работой. Вместе с тем есть еще участки фона, к примеру, слева на изображении. Мы могли бы вернуться назад и вручную отметить эти области как фон, но в реаль-

ном мире у нас будут тысячи изображений, и вручную фиксировать их по отдельности не представляется возможным. Следовательно, нужно просто принять, что данные изображения по-прежнему будут содержать некоторый фоновый шум.

В нашем решении мы начинаем с выделения прямоугольника вокруг области, содержащей передний план. Алгоритм GrabCut исходит из того, что все, что находится за пределами этого прямоугольника, является фоном, и использует эту информацию, чтобы выяснить, что является вероятным фоном внутри прямоугольника (чтобы узнать, как алгоритм делает это, обратитесь к внешним ресурсам в конце этого рецепта). Затем создается маска, обозначающая различные определенные/вероятные области фона/переднего плана (рис. 8.16):

```
# Показать маску
plt.imshow(mask, cmap='gray')
plt.axis("off")
plt.show()
```

Рис. 8.16

Рис. 8.17

Черная область — это область за пределами нашего прямоугольника, который определенно принимается за фон. Серая область — это то, что алгоритм GrabCut посчитал как вероятный фон, а белая область — вероятный передний план.

Эта маска затем используется для создания второй маски, которая объединяет черные и серые области (рис. 8.17).

Вторая маска затем применяется к изображению таким образом, что остается лишь передний план.

8.11. Обнаружение краев изображений

Задача

Требуется найти края изображения.

Решение

Использовать методику обнаружения, такую как детектор границ Джона Ф. Кэнни (рис. 8.18):

```
# Загрузить библиотеки
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Загрузить изображение в оттенках серого
image_gray = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# Вычислить медиану интенсивности
median_intensity = np.median(image_gray)

# Установить пороговые значения на одно стандартное отклонение
# выше и ниже медианы интенсивности
lower_threshold = int(max(0, (1.0 - 0.33) * median_intensity))
upper_threshold = int(min(255, (1.0 + 0.33) * median_intensity))

# Применить детектор границ Кэнни
image_canny = cv2.Canny(image_gray, lower_threshold, upper_threshold)

# Показать изображение
plt.imshow(image_canny, cmap="gray")
plt.axis("off")
plt.show()
```

Рис. 8.18

Обсуждение

Обнаружение краев на изображении является главной темой исследований в компьютерном зрении. Края важны тем, что они являются областями высокой информативности. Например, на нашем изображении один участок неба очень похож на другой и вряд ли будет содержать уникальную или интересную информацию. Вместе с тем участки, где фоновое небо граничит с самолетом, содержат много информации (например, форму объекта). Обнаружение краев позволяет нам извлечь низкоинформативные области и выделить области изображений, содержащие полезную информацию.

Существует целый ряд методов обнаружения краев (фильтры Собела, лапласов детектор границ и т. д.). Тем не менее в нашем решении используется широко применяемый детектор границ Кэнни. Детальное описание работы детектора Кэнни выходит за рамки этой книги, но есть один момент, который нам нужно уточнить. Детектор Кэнни требует два параметра, обозначающие низкие и высокие градиентные пороговые значения. Пикселы потенциального края между низким и высоким порогами считаются пикселами слабого края, в то время как пикселы выше высокого порога считаются пикселами сильного края. Метод `Canny` библиотеки OpenCV включает низкие и высокие пороговые значения в качестве обязательных параметров. В нашем решении мы устанавливаем нижний и верхний пороги на одно стандартное отклонение ниже и выше медианы интенсивности пикселов изображения. Тем не менее часто бывают случаи, когда можно получить лучшие результаты, если перед тем как выполнить метод `Canny` на всей коллекции наших изображений, мы находим хорошую пару низких и высоких пороговых значений путем ручного метода проб и ошибок, используя для этого несколько изображений.

Дополнительные материалы для чтения

- "Детектор границ Кэнни", Википедия (**http://bit.ly/2FzDXNt**).
- "Автоматическая пороговая обработка на основе детекции границ Кэнни", блог-пост (**http://bit.ly/2nmQERq**).

8.12. Обнаружение углов

Задача

Требуется обнаружить углы изображения.

Решение

Использовать реализацию детектора углов Харриса `cornerHarris` из библиотеки OpenCV (рис. 8.19):

```
# Загрузить библиотеки
import cv2
import numpy as np
from matplotlib import pyplot as plt
```

```
# Загрузить изображение в оттенках серого
image_bgr = cv2.imread("images/plane_256x256.jpg")
image_gray = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2GRAY)
image_gray = np.float32(image_gray)

# Задать параметры детектора углов
block_size = 2
aperture = 29
free_parameter = 0.04

# Обнаружить углы
detector_responses = cv2.cornerHarris(image_gray,
                                      block_size,
                                      aperture,
                                      free_parameter)

# Крупные угловые маркеры
detector_responses = cv2.dilate(detector_responses, None)

# Оставить только те отклики детектора, которые больше порога,
# пометить белым цветом
threshold = 0.02
image_bgr[detector_responses >
          threshold *
          detector_responses.max()] = [255,255,255]

# Конвертировать в оттенки серого
image_gray = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2GRAY)

# Показать изображение
plt.imshow(image_gray, cmap="gray")
plt.axis("off")
plt.show()
```

Рис. 8.19

Обсуждение

Метод детекции углов Харриса широко используется для обнаружения пересечения двух краев. Наш интерес к обнаружению углов мотивирован той же причиной, что и к обнаружению краев: углы являются точками высокой информативности. Полное объяснение принципа работы детектора углов Харриса можно найти на внешних ресурсах в конце этого рецепта, но упрощенное объяснение заключается в том, что детектор ищет окна (так называемые окрестности или участки), где небольшие движения окна (представьте встряхивание окна) создают большие изменения в содержимом пикселов внутри окна. Метод `cornerHarris` содержит три важных параметра, которые можно использовать для управления обнаруженными краями. Во-первых, `block_size` — это размер окрестности вокруг каждого пиксела, используемого для определения углов. Во-вторых, `aperture` — это размер используемого ядра Собела (не переживайте, если не знаете, что это такое), и, наконец, свободный параметр `free_parameter`, в котором более крупные значения соответствуют идентификации более "слабых" углов.

На выходе получается изображение в оттенках серого с изображением потенциальных углов (рис.8.20):

```
# Показать потенциальные углы
plt.imshow(detector_responses, cmap='gray')
plt.axis("off")
plt.show()
```

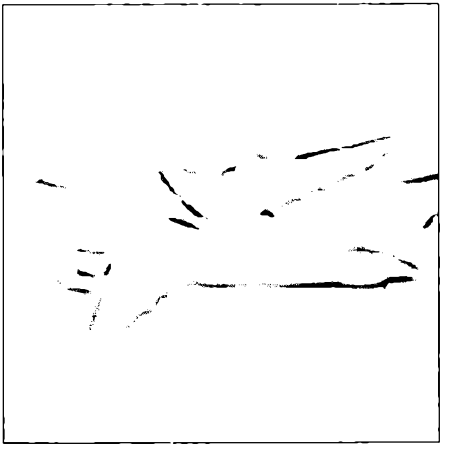


Рис. 8.20

Затем мы применяем пороговую обработку, чтобы сохранить только наиболее вероятные углы. В качестве альтернативы можно использовать аналогичный детектор — детектор углов Ши — Томази, который работает похожим с детектором Харриса образом (`goodFeaturesToTrack`) с целью идентификации фиксированного количества "сильных" углов. Метод `goodFeaturesToTrack` принимает три основных

аргумента — количество углов для обнаружения, минимальное качество угла (от 0 до 1) и минимальное евклидово расстояние между углами (рис. 8.21):

```
# Загрузить изображения
image_bgr = cv2.imread('images/plane_256x256.jpg')
image_gray = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2GRAY)

# Количество углов для обнаружения
corners_to_detect = 10
minimum_quality_score = 0.05
minimum_distance = 25

# Обнаружить углы
corners = cv2.goodFeaturesToTrack(image_gray,
                                  corners_to_detect,
                                  minimum_quality_score,
                                  minimum_distance)
corners = np.float32(corners)

# Нарисовать белый круг в каждом углу
for corner in corners:
    x, y = corner[0]
    cv2.circle(image_bgr, (x,y), 10, (255,255,255), -1)

# Конвертировать в оттенки серого
image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2GRAY)

# Показать изображение
plt.imshow(image_rgb, cmap='gray')
plt.axis("off")
plt.show()
```

Рис. 8.21

Дополнительные материалы для чтения

- Документация библиотеки OpenCV по методу `cornerHarris` **(http://bit.ly/2HQXwz6)**.
- Документация библиотеки OpenCV по методу `goodFeaturesToTrack` **(http://bit.ly/2HRSVwF)**.

8.13. Создание признаков для машинного самообучения

Задача

Требуется преобразовать изображение в наблюдение для машинного самообучения.

Решение

Использовать метод `flatten` библиотеки NumPy, чтобы преобразовать многомерный массив с данными изображения в вектор, содержащий значения наблюдения:

```
# Загрузить библиотеки
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Загрузить изображение в оттенках серого
image = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# Изменить размер изображения до 10 пикселов на 10 пикселов
image_10x10 = cv2.resize(image, (10, 10))

# Конвертировать данные изображения в одномерный вектор
image_10x10.flatten()
```

```
array([133, 130, 130, 129, 130, 129, 129, 128, 128, 127, 135, 131, 131,
       131, 130, 130, 129, 128, 128, 128, 134, 132, 131, 131, 130, 129,
       129, 128, 130, 133, 132, 158, 130, 133, 130,  46,  97,  26, 132,
       143, 141,  36,  54,  91,   9,   9,  49, 144, 179,  41, 142,  95,
        32,  36,  29,  43, 113, 141, 179, 187, 141, 124,  26,  25, 132,
       135, 151, 175, 174, 184, 143, 151,  38, 133, 134, 139, 174, 177,
       169, 174, 155, 141, 135, 137, 137, 152, 169, 168, 168, 179, 152,
       139, 136, 135, 137, 143, 159, 166, 171, 175], dtype=uint8)
```

Обсуждение

Изображения представлены в виде пиксельной решетки. Если изображение полутоновое, то каждый пиксел представлен одним значением (т. е. интенсивностью пик-

селов: 1 для белого, 0 для черного). Например, представим, что у нас имеется изображение 10×10 пикселов (рис. 8.22):

```
plt.imshow(image_10x10, cmap="gray")
plt.axis("off")
plt.show()
```

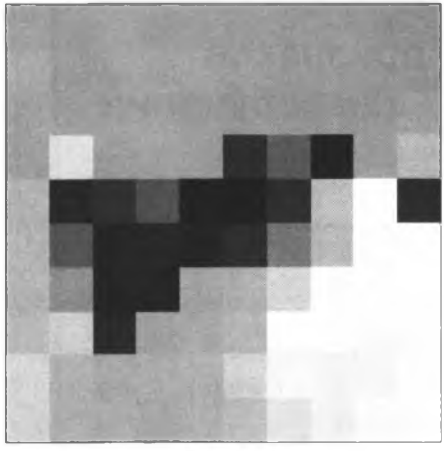


Рис. 8.22

В этом случае размерности данных изображений будут 10×10:

```
image_10x10.shape
```

```
(10, 10)
```

И если мы сгладим массив, то получим вектор длины 100 (10 умноженное на 10):

```
image_10x10.flatten().shape
```

```
(100,)
```

Этот вектор содержит признаки нашего изображения. Он может быть соединен с векторами из других изображений для создания данных, которые мы будем подавать в наши машинно-обучающиеся алгоритмы.

Если изображение имеет цвет, то вместо одного значения в расчете на каждый пиксел он (цвет) задается несколькими значениями (чаще всего тремя), представляющими каналы (красный, зеленый, синий и т. д.), которые смешиваются, чтобы получить окончательный цвет пиксела. По этой причине, если наше изображение 10×10 имеет цвет, то у нас будет 300 значений признаков для каждого наблюдения:

```
# Загрузить изображение в цвете
image_color = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_COLOR)
```

```
# Изменить размер изображения до 10 пикселов на 10 пикселов
image_color_10x10 = cv2.resize(image_color, (10, 10))

# Конвертировать данные изображения в одномерный вектор,
# показать размерности
image_color_10x10.flatten().shape
```

```
(300,)
```

Одна из основных сложностей обработки изображений и компьютерного зрения в целом заключается в том, что поскольку позиция каждого пиксела в коллекции изображений является признаком, по мере увеличения изображения наблюдается взрывной рост количества признаков:

```
# Загрузить изображение в оттенках серого
image_256x256_gray = cv2.imread("images/plane_256x256.jpg",
                                cv2.IMREAD_GRAYSCALE)

# Конвертировать данные изображения в одномерный вектор,
# показать размерности
image_256x256_gray.flatten().shape
```

```
(65536,)
```

И количество признаков только увеличивается, когда изображение цветное:

```
# Загрузить изображение в цвете
image_256x256_color = cv2.imread("images/plane_256x256.jpg",
                                 cv2.IMREAD_COLOR)

# Конвертировать данные изображения в одномерный вектор,
# показать размерности
image_256x256_color.flatten().shape
```

```
(196608,)
```

Как видно из результатов, даже небольшое цветное изображение имеет почти 200 тыс. признаков, что может вызвать проблемы во время тренировки наших моделей, поскольку количество признаков может значительно превышать количество наблюдений.

Эта проблема будет мотивировать стратегии уменьшения размерности, которые пытаются сократить количество признаков, не теряя при этом чрезмерный объем информации, содержащейся в данных. Мы рассмотрим этот вопрос в одной из следующих глав.

8.14. Кодирование среднего цвета в качестве признака

Задача

Требуется признак, основанный на цветах изображения.

Решение

Каждый пиксел изображения представлен комбинацией нескольких цветовых каналов (часто три: красный, зеленый и синий). Вычислить средние значения красного, зеленого и синего каналов для изображения, чтобы создать три цветовых признака, представляющих средние цвета в этом изображении:

```
# Загрузить библиотеки
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Загрузить изображение как BGR
image_bgr = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_COLOR)

# Вычислить среднее значение каждого канала
channels = cv2.mean(image_bgr)

# Поменять местами синее и красное значения (переведя в RGB, а не BGR)
observation = np.array([(channels[2], channels[1], channels[0])])

# Показать значения среднего канала
observation

array([[ 90.53204346, 133.11735535, 169.03074646]])
```

Средние значения канала можно просмотреть непосредственно (приносим извинения читателям печатного издания книги), рис. 8.23:

```
# Показать изображение
plt.imshow(observation)
plt.axis("off")
plt.show()
```

Рис. 8.23

Обсуждение

В результате мы получим три значения признаков для наблюдения, по одному для каждого цветового канала изображения. Эти признаки могут использоваться, как и любые другие признаки в обучающихся алгоритмах, для классификации изображений согласно их цветам.

8.15. Кодирование гистограмм цветовых каналов в качестве признаков

Задача

Требуется создать набор признаков, представляющих цвета на изображении.

Решение

Вычислить гистограммы для каждого цветового канала:

```
# Загрузить библиотеки
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Загрузить изображение
image_bgr = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_COLOR)

# Конвертировать в RGB
image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB)

# Создать список для значений признаков
features = []

# Вычислить гистограмму для каждого цветового канала
colors = ("r","g","b")

# Для каждого цветового канала:
# вычислить гистограмму и добавить в список значений признаков
for i, channel in enumerate(colors):
    histogram = cv2.calcHist([image_rgb], # Изображение
                             [i],         # Индекс канала
                             None,        # Маска отсутствует
                             [256],       # Размер гистограммы
                             [0,256])     # Диапазон
    features.extend(histogram)
```

```
# Создать вектор для значений признаков наблюдения
observation = np.array(features).flatten()

# Показать значение наблюдения для первых пяти признаков
observation[0:5]
```

```
array([1008.,  217.,  184.,  165.,  116.], dtype=float32)
```

Обсуждение

В цветовой модели RGB каждый цвет представляет собой комбинацию трех цветовых каналов (красный, зеленый, синий). В свою очередь, каждый канал может принимать одно из 256 значений (представленных целым числом от 0 до 255). Например, самый верхний левый пиксел изображения имеет следующие значения канала:

```
# Показать значения канала RGB
image_rgb[0,0]
```

```
array([107, 163, 212], dtype=uint8)
```

Гистограмма — это представление распределения значений в данных. Приведем простой пример (рис. 8.24):

```
# Импортировать pandas
import pandas as pd

# Создать немного данных
data = pd.Series([1, 1, 2, 2, 3, 3, 3, 4, 5])

# Показать гистограмму
data.hist(grid=False)
plt.show()
```

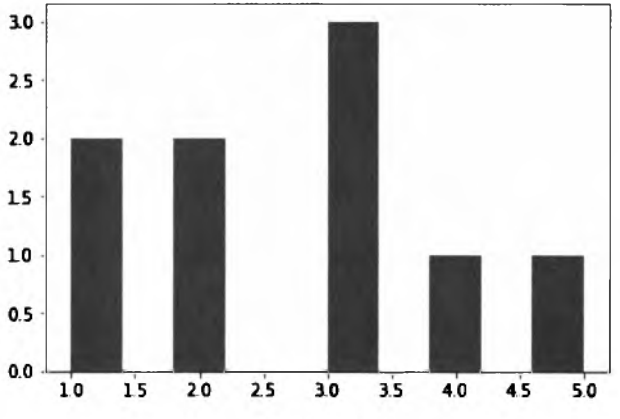


Рис. 8.24

В этом примере имеются некоторые данные с двумя единицами, двумя двойками, тремя тройками, одной четверкой и одной пятеркой. В гистограмме каждый столбец представляет, сколько раз каждое значение (1, 2 и т. д.) появляется в наших данных.

Этот же метод можно применить к каждому из цветовых каналов, но вместо пяти возможных значений будет 256 (диапазон возможных значений для значения канала). Ось x представляет 256 возможных значений канала, а ось y — количество появлений определенного значения канала во всех пикселах изображения (рис. 8.25):

```
# Вычислить гистограмму для каждого цветового канала
colors = ("r","g","b")

# Для каждого канала: вычислить гистограмму, построить график
for i, channel in enumerate(colors):
    histogram = cv2.calcHist([image_rgb], # Изображение
                             [i],         # Индекс канала
                             None,        # Маска отсутствует
                             [256],       # Размер гистограммы
                             [0,256])     # Диапазон
    plt.plot(histogram, color = channel)
    plt.xlim([0,256])

# Показать график
plt.show()
```

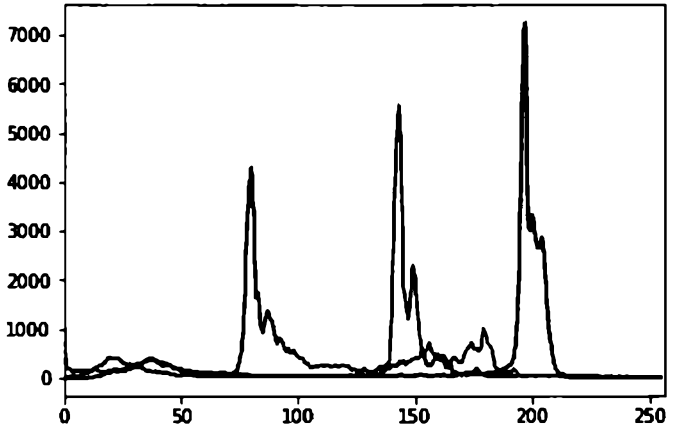


Рис. 8.25

Как мы видим на гистограмме, едва ли какой-либо из пикселов содержит значения синего канала между 0 и ~180, в то время как многие пикселы содержат значения синего канала от ~190 до ~210. Такое распределение значений канала показано для всех трех каналов. Однако гистограмма это не просто визуализация; это 256 при-

знаков для каждого цветового канала, что составляет в целом 768 признаков, представляющих распределение цветов в изображении.

Дополнительные материалы для чтения

- "Гистограмма", Википедия (**https://bit.ly/1N8vqcZ**).
- Документация библиотеки pandas по методу `hist` (**http://bit.ly/2HT4Fz0**).
- Руководство библиотеки OpenCV по применению гистограмм (**http://bit.ly/2HSyoYH**).

Снижение размерности с помощью выделения признаков

Введение

Обычно имеется доступ к тысячам и даже сотням тысяч признаков. Например, в *главе 8* мы преобразовали цветное изображение размера 256×256 пикселов в 196 608 признаков. Кроме того, поскольку каждый из этих пикселов может принимать одно из 256 возможных значений, в конечном итоге наши наблюдения могут принимать $256^{196\,608}$ различных конфигураций. Это порождает проблемы, потому что мы практически никогда не сможем собрать достаточно наблюдений, чтобы охватить даже небольшую часть этих конфигураций, и наши обучающиеся алгоритмы не будут иметь достаточно данных для правильной работы.

К счастью, не все признаки создаются равными, и выделение признаков для снижения размерности имеет совершенно конкретную цель — преобразование нашего набора признаков $p_{\text{исх}}$ таким образом, чтобы в конечном итоге прийти к новому набору $p_{\text{нов}}$, где $p_{\text{исх}} > p_{\text{нов}}$, сохраняя при этом подавляющую часть исходной информации. Другими словами, мы уменьшаем количество признаков с небольшой потерей способности наших данных генерировать высококачественные предсказания. В этой главе мы рассмотрим ряд методов выделения признаков, которые делают именно это.

Один недостаток методов выделения признаков, которые мы обсудим, состоит в том, что новые признаки, которые мы генерируем, не поддаются интерпретированию людьми. Они будут содержать столько же или почти столько же возможностей для тренировки наших моделей, но для человеческого глаза будут выглядеть как набор случайных чисел. Если мы хотим сохранить нашу способность интерпретировать наши модели, то лучше применить другой метод снижения размерности — уменьшение размерности посредством отбора признаков.

9.1. Снижение признаков с помощью главных компонент

Задача

Дан набор признаков, и требуется сократить количество признаков, сохраняя при этом дисперсию данных.

Решение

Использовать анализ главных компонент с помощью класса PCA библиотеки scikit-learn:

```
# Загрузить библиотеки
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn import datasets

# Загрузить данные
digits = datasets.load_digits()

# Стандартизировать матрицу признаков
features = StandardScaler().fit_transform(digits.data)

# Создать объект PCA, который сохранит 99% дисперсии
pca = PCA(n_components=0.99, whiten=True)

# Выполнить анализ PCA
features_pca = pca.fit_transform(features)

# Показать результаты
print("Исходное количество признаков:", features.shape[1])
print("Сокращенное количество признаков:", features_pca.shape[1])
```

```
Исходное количество признаков: 64
Сокращенное количество признаков: 54
```

Обсуждение

Анализ главных компонент (principal component analysis, PCA) — это популярный метод уменьшения линейной размерности. Метод PCA проецирует наблюдения на главные компоненты матрицы признаков (надо надеяться, на меньшее их количество), которые сохраняют наибольшую дисперсию. PCA является неконтролируемым методом (без учителя), т. е. он не использует информацию из вектора целей и вместо этого рассматривает только матрицу признаков.

По поводу математического описания того, как работает PCA, обратитесь к внешним ресурсам, перечисленным в конце этого рецепта. Тем не менее мы можем интуитивно понять принцип работы PCA, используя простой пример. На рис. 9.1 наши данные содержат два признака: x_1 и x_2. Глядя на визуализацию, должно быть ясно, что наблюдения разбросаны сигарообразно, с большой длиной и очень маленькой высотой. Если быть более конкретным, можно сказать, что дисперсия "длины" значительно больше дисперсии "высоты". Вместо длины и высоты мы именуем "направления" с наибольшей дисперсией первой главной компонентой (1-й ГК) и "направление" со второй наибольшей дисперсией — второй главной компонентой (2-й ГК, и т. д.).

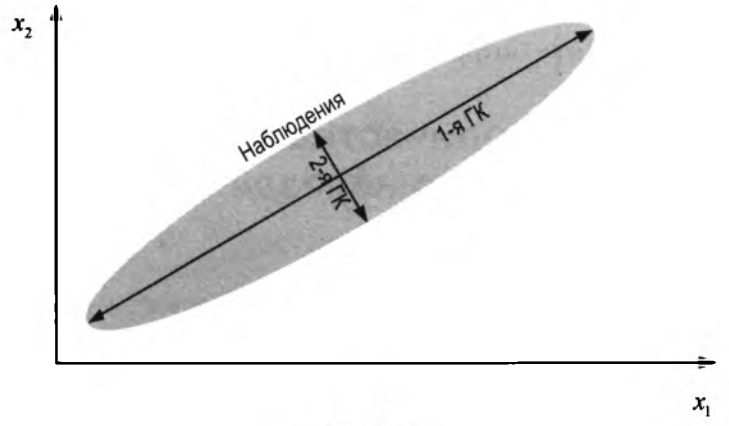


Рис. 9.1

Если требуется сократить наши признаки, одной из стратегий будет проецирование всех наблюдений в двумерном пространстве на одномерную главную компоненту. Мы потеряем информацию, собранную во второй главной компоненте, но в некоторых ситуациях такой подход будет приемлемым компромиссом. Это и есть анализ главных компонент (PCA).

Анализ главных компонент реализуется в библиотеке scikit-learn с использованием класса `PCA`. У аргумента `n_components` есть две операции в зависимости от заданного значения. Если значение этого аргумента больше 1, то `n_components` вернет указанное количество признаков. Это приводит к вопросу выбора оптимального количества признаков. К счастью для нас, если значение аргумента `n_components` находится между 0 и 1, то объект, созданный на основе класса `PCA`, возвращает минимальное количество признаков, которые сохраняют указанную дисперсию. Обычно используются значения 0.95 и 0.99, т. е. 95 и 99% дисперсии исходных признаков будут сохранены. Аргумент `whiten=True` преобразует значения каждой главной компоненты таким образом, чтобы они имели нулевое среднее и единичную дисперсию. Еще одним аргументом является `svd_solver="randomized"`, который реализует стохастический алгоритм нахождения первых главных компонент, который, как правило, занимает значительно меньше времени.

Результат нашего решения показывает, что метод PCA позволяет уменьшить размерность на 10 признаков, сохраняя при этом 99% информации (дисперсии) в матрице признаков.

Дополнительные материалы для чтения

- Документация библиотеки scikit-learn по анализу главных компонент и классу PCA (**http://bit.ly/2FrSvyx**).
- "Выбор количества главных компонент", видеоурок на Coursera (**http://bit.ly/2FrSGtH**).

- "Анализ главных компонент с линейной алгеброй", статья на математическом факультете Union College (**http://bit.ly/2FuzdIW**).

9.2. Уменьшение количества признаков, когда данные линейно неразделимы

Задача

Вы подозреваете, что ваши данные линейно неразделимы, и требуется сократить размерности.

Решение

Применить расширение анализа главных компонент, в котором используются ядра для нелинейного уменьшения размерности:

```
# Загрузить библиотеки
from sklearn.decomposition import PCA, KernelPCA
from sklearn.datasets import make_circles

# Создать линейно неразделимые данные
features, _ = make_circles(n_samples=1000, random_state=1,
                           noise=0.1, factor=0.1)

# Применить ядерный PCA
# с радиально-базисным функциональным ядром (RBF-ядром)
kpca = KernelPCA(kernel="rbf", gamma=15, n_components=1)
features_kpca = kpca.fit_transform(features)

print("Исходное количество признаков:", features.shape[1])
print("Сокращенное количество признаков:", features_kpca.shape[1])

Исходное количество признаков: 2
Сокращенное количество признаков: 1
```

Обсуждение

Метод анализа главных компонет (PCA) способен уменьшить размерность нашей матрицы признаков (например, количество признаков). В стандартном методе PCA для уменьшения количества признаков используется линейная проекция. Если данные линейно разделимы (т. е. вы можете прочертить прямую или гиперплоскость между разными классами), то PCA работает хорошо. Однако, если ваши данные не являются линейно разделимыми (например, вы можете разделить классы только с помощью изогнутой границы решения), линейное преобразование работать не будет. В нашем решении для генерирования симулированного набора данных с вектором целей из двух классов и двумя признаками мы использовали функцию

`make_circles` библиотеки scikit-learn. Функция `make_circles` создает линейно неразделимые данные; в частности один класс окружен со всех сторон другим классом (рис. 9.2).

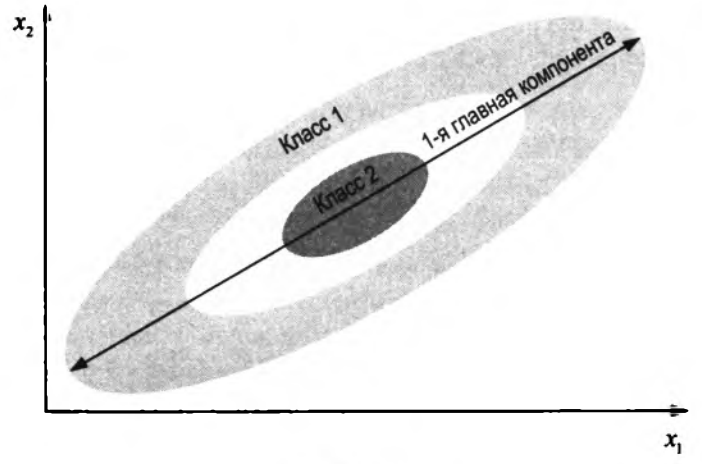


Рис. 9.2

Если бы для уменьшения размерностей наших данных мы использовали линейный метод PCA, то два класса были бы линейно спроецированы на первую главную компоненту таким образом, что они стали бы переплетенными (рис. 9.3).

Рис. 9.3

В идеале мы хотели бы иметь преобразование, которое сокращало бы размерности, а также делало бы данные линейно разделимыми (рис. 9.4). Ядерный PCA способен делать обе вещи.

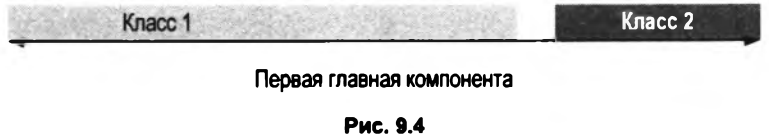


Рис. 9.4

Ядра позволяют проецировать линейно неразделимые данные в более высокую размерность, где они линейно разделимы; этот подход называется *ядерным трюком*. Не переживайте, если не понимаете детали ядерного трюка; просто представьте ядра как разные способы проецирования данных. В объекте, созданном из класса `KernelPCA` библиотеки scikit-learn можно использовать несколько ядер, задаваемых с помощью параметра `kernel`. Широко используемым ядром является гауссово

радиально-базисное функциональное ядро `rbf`, но есть и другие варианты — полиномиальное ядро (`poly`) и сигмоидное ядро (`sigmoid`). Мы даже можем указать линейную проекцию (`linear`), которая даст те же результаты, что и стандартный метод PCA.

Одним из недостатков ядерного метода PCA является то, что мы должны указать ряд параметров. Например, в рецепте 9.1 мы задаем `n_components` равным 0.99, чтобы PCA отбирал количество компонент с сохранением 99% дисперсии. Такая возможность в ядерном методе PCA отсутствует. Вместо этого мы должны задать ряд параметров (например, `n_components=1`). Более того, ядра сопровождаются своими гиперпараметрами, которые нам придется устанавливать; например, радиально-базисная функция требует значения `gamma`.

Тогда как мы узнаем, какие значения использовать? Методом проб и ошибок. В частности, мы можем натренировать нашу машинно-обучающуюся модель несколько раз, каждый раз с другим ядром или другим значением параметра. Как только мы находим комбинацию значений, которая производит предсказанные значения высшего качества, работа сделана. Мы подробно рассмотрим эту стратегию в *главе 12*.

Дополнительные материалы для чтения

- Документация библиотеки scikit-learn по ядерному PCA (**http://bit.ly/2HRkxC3**).
- "Ядерные трюки и уменьшение нелинейной размерности посредством ядерного PCA на основе радиально-базисной функции RBF", веб-сайт Себастьяна Рашки (**http://bit.ly/2HReP3f**).

9.3. Уменьшение количества признаков путем максимизации разделимости классов

Задача

Требуется сократить признаки, используемые классификатором.

Решение

Попробовать линейный дискриминантный анализ (linear discriminant analysis, LDA), чтобы спроецировать объекты на оси компонент, которые максимизируют разделение классов:

```
# Загрузить библиотеки
from sklearn import datasets
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

# Загрузить набор данных цветков ириса:
iris = datasets.load_iris()
features = iris.data
target = iris.target
```

```
# Создать объект и выполнить LDA, затем использовать
# его для преобразования признаков
lda = LinearDiscriminantAnalysis(n_components=1)
features_lda = lda.fit(features, target).transform(features)

# Напечатать количество признаков
print("Исходное количество признаков:", features.shape[1])
print("Сокращенное количество признаков:", features_lda.shape[1])

Исходное количество признаков: 4
Сокращенное количество признаков: 1
```

Для просмотра объема дисперсии, объясненной каждой компонентой можно использовать атрибут `explained_variance_ratio_`. В нашем решении одна компонента объяснила более 99% дисперсии:

```
lda.explained_variance_ratio_

array([0.99147248])
```

Обсуждение

Линейный дискриминантный анализ (LDA) — это классификационный метод, который так же популярен, как метод уменьшения размерности. Метод LDA работает аналогично анализу главных компонент (PCA): он проецирует пространство признаков на пространство более низкой размерности. Однако в PCA нас интересовали только те оси компонент, которые максимизируют дисперсию данных, в то время как в LDA есть дополнительная цель максимизировать различия между классами. В приведенном на рис. 9.5 наглядном примере имеются данные, содержащие два целевых класса и два признака. Если мы спроецируем данные на ось y, то два класса не будут легко разделимыми (т. е. они перекрываются), тогда как, если мы спроецируем данные на ось x, то мы останемся с вектором признаков (т. е. мы

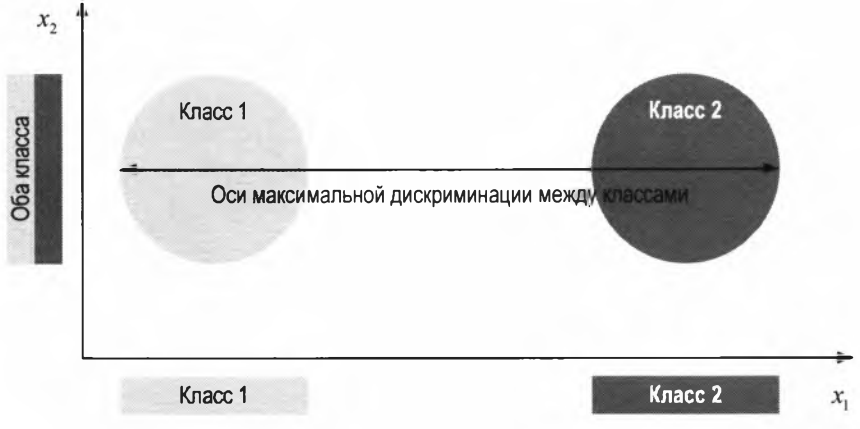


Рис. 9.5

сократили нашу размерность на единицу), который по-прежнему сохраняет разделимость классов. В реальном мире, безусловно, связь между классами будет более сложной и размерность будет выше, но концепция остается той же.

В библиотеке scikit-learn метод LDA реализован с использованием класса `LinearDiscriminantAnalysis`, который включает параметр `n_components`, указывающий на количество признаков, которые требуется вернуть. Для того чтобы выяснить, какое значение аргумента использовать в `n_components` (например, сколько параметров сохранить), можно воспользоваться тем фактом, что коэффициент `explained_variance_ratio_` сообщает нам дисперсию, объясняемую каждым выводимым признаком, и является отсортированным массивом. Например,

```
lda.explained_variance_ratio_
```

```
array([ 0.99147248])
```

В частности, можно выполнить линейный дискриминантный анализ на основе класса `LinearDiscriminantAnalysis` с компонентами `n_components`, заданными как `None`, чтобы вернуть коэффициент дисперсии, объясненный каждой компонентой, а затем вычислить, сколько требуется компонент, чтобы превысить некоторый порог дисперсии (часто 0.95 или 0.99):

```
# Создать объект и выполнить LDA
lda = LinearDiscriminantAnalysis(n_components=None)
features_lda = lda.fit(features, target)

# Создать массив коэффициентов объясненной дисперсии
lda_var_ratios = lda.explained_variance_ratio_

# Создать функцию
def select_n_components(var_ratio, goal_var: float) -> int:
    # Задать исходную объясненную на данный момент дисперсию
    total_variance = 0.0

    # Задать исходное количество признаков
    n_components = 0

    # Для объясненной дисперсии каждого признака:
    for explained_variance in var_ratio:

        # Добавить объясненную дисперсию к итогу
        total_variance += explained_variance

        # Добавить единицу к количеству компонент
        n_components += 1

        # Если достигнут целевой уровень объясненной дисперсии
        if total_variance >= goal_var:
            # Завершить цикл
            break
```

```
    # Вернуть количество компонент
    return n_components

# Выполнить функцию
select_n_components(lda_var_ratios, 0.95)
```

```
1
```

Дополнительные материалы для чтения

- "Сравнение двумерной проекции набора данных цветков ириса методами линейного дискриминантного анализа (LDA) и анализа главных компонент (PCA)", документация библиотеки scikit-learn (**http://bit.ly/2Fs4cWe**).
- "Линейный дискриминантный анализ", веб-сайт Себастьяна Рашки (**http://bit.ly/2FtiKEL**).

9.4. Уменьшение количества признаков с использованием разложения матрицы

Задача

Дана матрица признаков с неотрицательными значениями, и требуется уменьшить ее размерность.

Решение

Использовать разложение неотрицательной матрицы (non-negative matrix factorization, NMF) с целью уменьшения размерности матрицы признаков:

```
# Загрузить библиотеки
from sklearn.decomposition import NMF
from sklearn import datasets

# Загрузить данные
digits = datasets.load_digits()

# Загрузить матрицу признаков
features = digits.data

# Создать NMF и выполнить его подгонку
nmf = NMF(n_components=10, random_state=1)
features_nmf = nmf.fit_transform(features)

# Показать результаты
print("Исходное количество признаков:", features.shape[1])
print("Сокращенное количество признаков:", features_nmf.shape[1])
```

```
Исходное количество признаков: 64
Сокращенное количество признаков: 10
```

Обсуждение

Разложение неотрицательной матрицы (NMF) является неконтролируемым (без учителя) методом уменьшения линейной размерности, который факторизует (т. е. разбивает на несколько матриц, произведение которых соответствует исходной матрице) матрицу признаков в матрицы, представляющие скрытую связь между наблюдениями и их признаками. Интуитивно понятно, что метод NMF может сократить размерность, поскольку в матричном умножении два сомножителя (умножаемые матрицы) могут иметь значительно меньшие размерности, чем матрица произведения. Формально, если дано желаемое число возвращаемых признаков r, то метод NMF разлагает матрицу признаков так, что:

$$\mathbf{V} \approx \mathbf{WH},$$

где $\mathbf{V}$ — наша матрица признаков $d \times n$ (т. е. d признаков, n наблюдений); $\mathbf{W}$ — матрица $d \times r$; $\mathbf{H}$ — матрица $r \times n$. Скорректировав величину r, можно задать объем желаемого уменьшения размерности.

Одно из основных требований метода разложения неотрицательной матрицы (NMA) состоит в том, что, как следует из названия, матрица признаков не может содержать отрицательные значения. Кроме того, в отличие от методв PCA и других методов, которые мы рассмотрели, метод NMA не предоставляет нам объясненную дисперсию результирующих признаков. Таким образом, лучший для нас способ найти оптимальное значение компонент `n_components` — это попытаться найти в диапазоне значений то, которое дает наилучший результат в нашей конечной модели *(см. главу 12)*.

Дополнительные материалы для чтения

- "Разложение неотрицательной матрицы (NMF)", блог-пост (**http://bit.ly/2FvtWRj**).

9.5. Уменьшение количества признаков на разреженных данных

Задача

Дана разреженная матрица признаков, и требуется уменьшить ее размерность.

Решение

Использовать усеченное сингулярное разложение (truncated singular value decomposition, TSVD):

```
# Загрузить библиотеки
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import TruncatedSVD
from scipy.sparse import csr_matrix
from sklearn import datasets
import numpy as np

# Загрузить данные
digits = datasets.load_digits()

# Стандартизировать матрицу признаков
features = StandardScaler().fit_transform(digits.data)

# Сделать разреженную матрицу
features_sparse = csr_matrix(features)

# Создать объект TSVD
tsvd = TruncatedSVD(n_components=10)

# Выполнить TSVD на разреженной матрице
features_sparse_tsvd = tsvd.fit(features_sparse).transform(features_sparse)

# Показать результаты
print("Исходное количество признаков:", features_sparse.shape[1])
print("Сокращенное количество признаков:", features_sparse_tsvd.shape[1])
```

```
Исходное количество признаков: 64
Сокращенное количество признаков: 10
```

Обсуждение

Усеченное сингулярное разложение (TSVD) похоже на анализ главных компонент (PCA), и действительно в PCA на одном из своих шагов часто используется неусеченное сингулярное разложение (SVD). В обычном сингулярном разложении при наличии d признаков создаются матрицы-сомножители размера $d \times d$, тогда как усеченное сингулярное разложение вернет сомножители, которые имеют размер $n \times n$, где n предварительно заданы параметром. Практическое преимущество TSVD заключается в том, что в отличие от PCA, этот метод работает на разреженных матрицах признаков.

Одна из проблем с методом TSVD связана с тем, что в зависимости от того, как этот метод использует генератор случайных чисел, знаки результата могут меняться от подгонки к подгонке. Простым решением будет использовать метод `fit` в конвейере предобработки всего один раз, а затем несколько раз использовать метод `transform`.

Как и в случае линейного дискриминантного анализа, мы должны указать количество признаков (компонент), которые мы хотим вывести. Это делается с помощью

параметра `n_components`. Тогда возникает естественный вопрос: каково оптимальное количество компонент? Одна стратегия состоит в том, чтобы включить `n_components` как гиперпараметр для оптимизации при отборе модели (т. е. выбрать значение для `n_components`, которое производит наилучшую натренированную модель). В качестве альтернативы, поскольку метод TSVD предоставляет нам коэффициент объясненной дисперсии каждой компоненты исходной матрицы признаков, мы можем выделить ряд компонент, которые объясняют желаемый объем дисперсии (общепринято использовать значения 95 или 99%). Например, в нашем решении первые три выводимые компоненты объясняют примерно 30% дисперсии исходных данных:

```
# Суммировать коэффициенты объясненной дисперсии первых трех компонент
tsvd.explained_variance_ratio_[0:3].sum()
```

```
0.30039385386597783
```

Этот процесс можно автоматизировать, создав функцию, которая выполняет усеченное сингулярное разложение с компонентами `n_components`, установленными на единицу меньше, чем количество исходных признаков, а затем вычислить количество компонент, которые объясняют желаемый объем дисперсии исходных данных:

```
# Создать и выполнить TSVD с числом признаков меньше на единицу
tsvd = TruncatedSVD(n_components=features_sparse.shape[1]-1)
features_tsvd = tsvd.fit(features)

# Поместить в список объясненные дисперсии
tsvd_var_ratios = tsvd.explained_variance_ratio_

# Создать функцию
def select_n_components(var_ratio, goal_var):
    # Задать исходную объясненную на данный момент дисперсию
    total_variance = 0.0

    # Задать исходное количество признаков
    n_components = 0

    # Для объясненной дисперсии каждого признака:
    for explained_variance in var_ratio:

        # Добавить объясненную дисперсию к итогу
        total_variance += explained_variance

        # Добавить единицу к количеству компонент
        n_components += 1

        # Если достигнут целевой уровень объясненной дисперсии
        if total_variance >= goal_var:
            # Завершить цикл
            break
```

```
    # Вернуть количество компонент
    return n_components

# Выполнить функцию
select_n_components(tsvd_var_ratios, 0.95)
```

```
40
```

Дополнительные материалы для чтения

- Документация библиотеки scikit-learn по усеченному сингулярному разложению и классу TruncatedSVD (**http://bit.ly/2GTfxvh**).

Снижение размерности с помощью отбора признаков

Введение

В главе 9 мы обсудили, как уменьшить размерность нашей матрицы признаков путем создания новых объектов (в идеале) с аналогичной способностью тренировать качественные модели, но со значительно меньшим числом размерностей. Этот подход называется *выделением признаков*. В данной главе мы рассмотрим альтернативный подход: отбор высококачественных, информативных признаков и удаление менее полезных признаков. Этот подход называется *отбором признаков*.

Существует три типа методов отбора признаков: фильтрующие, циклические и вложенные. Фильтрующие методы отбирают наилучшие признаки, изучая их статистические свойства. Циклические (wrapper) методы для поиска подмножества признаков, которые создают модели с предсказаниями самого высокого качества, используют метод проб и ошибок. Наконец, вложенные (embedded) методы отбирают наилучшее подмножество признаков как часть или как продолжение процесса тренировки обучающегося алгоритма.

В этой главе идеально было бы описать все три метода. Однако, поскольку вложенные методы тесно переплетены с конкретными обучающимися алгоритмами, их трудно объяснить без более глубокого погружения в сами алгоритмы. Поэтому мы рассмотрим только фильтрующие и циклические методы отбора, отложив обсуждение конкретных вложенных методов до глав, где эти обучающиеся алгоритмы будут рассмотрены подробно.

10.1. Пороговая обработка дисперсии числовых признаков

Задача

Дан набор числовых признаков, и требуется удалить те из них, которые имеют низкую дисперсию (т. е., скорее всего, содержат мало информации).

Решение

Отобрать подмножество признаков с дисперсиями выше заданного порога:

```
# Загрузить библиотеки
from sklearn import datasets
from sklearn.feature_selection import VarianceThreshold
```

```
# Импортировать немного данных для экспериментирования
iris = datasets.load_iris()

# Создать признаки и цель
features = iris.data
target = iris.target

# Создать обработчик порога
thresholder = VarianceThreshold(threshold=.5)

# Создать матрицу высокодисперсионных признаков
features_high_variance = thresholder.fit_transform(features)

# Взглянуть на матрицу высокодисперсионных признаков
features_high_variance[0:3]

array([[5.1, 1.4, 0.2],
       [4.9, 1.4, 0.2],
       [4.7, 1.3, 0.2]])
```

Обсуждение

Пороговая обработка дисперсии (variance thresholding, VT) является одним из основных подходов к отбору признаков. Она мотивирована идеей, что низкодисперсные признаки скорее всего менее интересны (и полезны), чем высокодисперсные признаки. Метод пороговой обработки дисперсии сначала вычисляет дисперсию каждого признака:

$$Var(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2,$$

где $\mathbf{x}$ — вектор признаков; x_i — значение отдельного признака; μ — среднее значение этого признака. Затем он исключает все признаки, дисперсия которых не удовлетворяет этому порогу.

Задействуя метод пороговой обработки дисперсии, следует иметь в виду два момента. Во-первых, дисперсия не центрирована, она измеряется в квадратах единиц измерения самого признака. Поэтому этот метод не будет работать, если наборы признаков содержат разные единицы измерения (например, один признак измеряется в годах, а другой — в долларах). Во-вторых, порог дисперсии отбирается вручную, поэтому для выбора хорошего значения мы должны использовать наше собственное суждение (или применять метод отбора модели, описанный в *главе 12*). Дисперсию для каждого признака можно увидеть с помощью атрибута `variances_`:

```
# Взглянуть на дисперсии
thresholder.fit(features).variances_
```

```
array([ 0.68112222, 0.18675067, 3.09242489, 0.57853156])
```

Наконец, если признаки были стандартизированы (в нулевое среднее и единичную дисперсию), то по очевидным причинам пороговая обработка дисперсии будет работать неправильно:

```
# Загрузить библиотеку
from sklearn.preprocessing import StandardScaler

# Стандартизировать матрицу признаков
scaler = StandardScaler()
features_std = scaler.fit_transform(features)

# Вычислить дисперсию каждого признака
selector = VarianceThreshold()
selector.fit(features_std).variances_
```

```
array([1., 1., 1., 1.])
```

10.2. Пороговая обработка дисперсии бинарных признаков

Задача

Дан набор бинарных категориальных признаков, и требуется удалить те из них, которые имеют низкую дисперсию (т. е., скорее всего, содержат мало информации).

Решение

Отобрать подмножество признаков с дисперсией бернуллиевых случайных величин выше заданного порога:

```
# Загрузить библиотеку
from sklearn.feature_selection import VarianceThreshold

# Создать матрицу признаков, где:
# признак 0: 80% класс 0
# признак 1: 80% класс 1
# признак 2: 60% класс 0, 40% класс 1
features = [[0, 1, 0],
            [0, 1, 1],
            [0, 1, 0],
            [0, 1, 1],
            [1, 0, 0]]
```

```
# Выполнить пороговую обработку по дисперсии
thresholder = VarianceThreshold(threshold=(.75 * (1 - .75)))
thresholder.fit_transform(features)

array([[0],
       [1],
       [0],
       [1],
       [0]])
```

Обсуждение

Как и в случае с числовыми признаками, одной из стратегий отбора высокоинформативных категориальных признаков является исследование их дисперсий. В бинарных признаках (т. е. бернуллиевых случайных величинах) дисперсия рассчитывается как:

$$Var(\mathbf{x}) = p(1-p),$$

где p — доля наблюдений класса 1. Поэтому, установив p, мы можем удалить признаки, где подавляющее большинство наблюдений является одним классом.

10.3. Обработка высококоррелированных признаков

Задача

Дана матрица признаков, и есть подозрения, что некоторые признаки сильно коррелированы.

Решение

Использовать корреляционную матрицу для выполнения проверки на сильно коррелированные признаки. Если существуют сильно коррелированные признаки, то рассмотреть возможность исключения одного из коррелированных признаков (табл. 10.1):

```
# Загрузить библиотеки
import pandas as pd
import numpy as np

# Создать матрицу признаков с высококоррелированными признаками
features = np.array([[1, 1, 1],
                     [2, 2, 0],
                     [3, 3, 1],
                     [4, 4, 0],
                     [5, 5, 1],
                     [6, 6, 0],
```

```
                    [7, 7, 1],
                    [8, 7, 0],
                    [9, 7, 1]])

# Конвертировать матрицу признаков во фрейм данных
dataframe = pd.DataFrame(features)

# Создать корреляционную матрицу
corr_matrix = dataframe.corr().abs()

# Выбрать верхний треугольник корреляционной матрицы
upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape),
k=1).astype(np.bool))

# Найти индекс столбцов признаков с корреляцией больше 0.95
to_drop = [column for column in upper.columns if any(upper[column] > 0.95)]

# Исключить признаки
dataframe.drop(dataframe.columns[to_drop], axis=1).head(3)
```

Таблица 10.1

	0	2
0	1	1
1	2	0
2	3	1

Обсуждение

Одна из проблем, с которой мы часто сталкиваемся в машинном самообучении, — это сильно коррелированные признаки. Если два признака сильно коррелированы, то информация, которую они содержат, очень похожа, и, скорее всего, включать оба признака будет излишним. Решение проблемы сильно коррелированных признаков простое: удалить один из них из набора признаков.

Во-первых, мы создаем корреляционную матрицу всех признаков (табл. 10.2):

```
# Корреляционная матрица
dataframe.corr()
```

Таблица 10.2

	0	1	2
0	1.000000	0.976103	0.000000
1	0.976103	1.000000	–0.034503
2	0.000000	–0.034503	1.000000

Во-вторых, мы смотрим на верхний треугольник корреляционной матрицы, чтобы определить пары сильно коррелированных признаков (табл. 10.3):

```
# Верхний треугольник корреляционной матрицы
upper
```

Таблица 10.3

	0	1	2
0	NaN	0.976103	0.000000
1	NaN	NaN	0.034503
2	NaN	NaN	NaN

В-третьих, из каждой пары из набора признаков мы удаляем по одному признаку.

10.4. Удаление нерелевантных признаков для классификации

Задача

Дан категориальный вектор целей, и требуется удалить неинформативные признаки.

Решение

Если признаки являются категориальными, то вычислить статистический показатель хи-квадрат (χ^2) между каждым признаком и вектором целей:

```
# Загрузить библиотеки
from sklearn.datasets import load_iris
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2, f_classif

# Загрузить данные
iris = load_iris()
features = iris.data
target = iris.target

# Конвертировать в категориальные данные путем
# преобразования данных в целые числа
features = features.astype(int)

# Отобрать два признака с наивысшими значениями
# статистического показателя хи-квадрат
chi2_selector = SelectKBest(chi2, k=2)
features_kbest = chi2_selector.fit_transform(features, target)
```

```
# Показать результаты
print("Исходное количество признаков:", features.shape[1])
print("Сокращенное количество признаков:", features_kbest.shape[1])
```

```
Исходное количество признаков: 4
Сокращенное количество признаков: 2
```

Если признаки являются количественными, то вычислить статистический показатель *F* дисперсионного анализа (ANalysis Of VAriance, ANOVA)[1] между каждым признаком и вектором целей:

```
# Отобрать два признака с наивысшими значениями
# статистического показателя F
fvalue_selector = SelectKBest(f_classif, k=2)
features_kbest = fvalue_selector.fit_transform(features, target)

# Показать результаты
print("Исходное количество признаков:", features.shape[1])
print("Сокращенное количество признаков:", features_kbest.shape[1])
```

```
Исходное количество признаков: 4
Сокращенное количество признаков: 2
```

Вместо отбора конкретного количества признаков также можно использовать объект класса `SelectPercentile` для отбора верхнего *n* процента признаков:

```
# Загрузить библиотеку
from sklearn.feature_selection import SelectPercentile

# Отобрать верхние 75% признаков с наивысшими значениями
# статистического показателя F
fvalue_selector = SelectPercentile(f_classif, percentile=75)
features_kbest = fvalue_selector.fit_transform(features, target)

# Показать результаты
print("Исходное количество признаков:", features.shape[1])
print("Сокращенное количество признаков:", features_kbest.shape[1])
```

```
Исходное количество признаков: 4
Сокращенное количество признаков: 3
```

Обсуждение

Статистический показатель хи-квадрат проверяет независимость двух категориальных векторов. То есть этот статистический показатель является разницей

[1] *F*-статистика — стандартизированная статистическая величина, измеряющая степень, с которой разницы в групповых средних превышают то, что можно ожидать в случайной модели. — *Прим. перев.*

между наблюдаемым числом наблюдений в каждом классе категориального признака и тем, что мы ожидали бы, если бы этот объект был независимым от вектора целей (т. е. не связанным с ним):

$$\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i},$$

где O_i — количество наблюдений в классе i; E_i — количество наблюдений в классе i, которое мы ожидаем, если нет связи между объектом и вектором целей.

Статистический показатель хи-квадрат представляет собой единственное число, которое говорит о том, насколько велика разница между вашими наблюдаемыми количествами и количествами, которые вы ожидали бы, если бы в популяции не было никакой связи вообще. Вычисляя статистический показатель хи-квадрат между признаком и вектором целей, мы получаем меру независимости между ними. Если вектор целей не зависит от признаковой переменной, то она не годится для нас, поскольку не содержит информации, которую мы можем использовать для классификации. С другой стороны, если две величины сильно зависят, то они, скорее всего, очень информативны для тренировки нашей модели.

Для того чтобы использовать хи-квадрат в отборе признаков, мы вычисляем статистический показатель хи-квадрат между каждым признаком и вектором целей, а затем отбираем признаки с наилучшими статистическими показателями хи-квадрат. В библиотеке scikit-learn можно использовать объект `SelectKBest` для отбора признаков с наилучшими статистическими показателями. Параметр k определяет количество признаков, которые мы хотим оставить.

Важно отметить, что статистический показатель хи-квадрат может быть вычислен только между двумя категориальными векторами. По этой причине хи-квадрат для выбора признаков требует, чтобы и вектор целей, и признаки были категориальными. Однако если у нас числовой признак, то можно применить метод хи-квадрат, сначала преобразовав количественный признак в категориальный. Наконец, чтобы использовать наш подход на основе хи-квадрат, все значения должны быть неотрицательными.

В качестве альтернативы, если имеется числовой признак, можно использовать метод `f_classif` для вычисления статистического показателя F дисперсионного анализа (ANOVA) с каждым признаком и вектором целей. Оценочные значения F проверяют, различаются ли значимо средние значения для каждой группы, когда мы группируем этот числовой признак по вектору целей. Например, если бы у нас был бинарный вектор целей — пол и количественный признак — экзаменационные баллы, то оценочный показатель F говорил бы нам о том, отличается ли средний экзаменационный балл для мужчин от среднего экзаменационного балла для женщин. Если нет, то экзаменационный балл не помогает нам предсказывать пол, и поэтому этот признак не имеет значения.

10.5. Рекурсивное устранение признаков

Задача

Требуется возможность автоматически отбирать наилучшие признаки для использования в дальнейшем.

Решение

Для выполнения рекурсивного устранения признаков (recursive feature elimination, RFE) использовать класс `RFECV` библиотеки scikit-learn совместно с перекрестной проверкой (cross-validation, CV). То есть многократно тренировать модель, каждый раз удаляя признак до тех пор, пока результативность модели (например, точность) не станет хуже. Оставшиеся признаки являются наилучшими:

```
# Загрузить библиотеки
import warnings
from sklearn.datasets import make_regression
from sklearn.feature_selection import RFECV
from sklearn import datasets, linear_model

# Убрать раздражающее, но безвредное предупреждение
warnings.filterwarnings(action="ignore", module="scipy",
                        message="^internal gelsd")

# Сгенерировать матрицу признаков, вектор целей и истинные коэффициенты
features, target = make_regression(n_samples = 10000,
                                   n_features = 100,
                                   n_informative = 2,
                                   random_state = 1)

# Создать объект линейной регрессии
ols = linear_model.LinearRegression()

# Рекурсивно устранить признаки
rfecv = RFECV(estimator=ols, step=1, scoring="neg_mean_squared_error")
rfecv.fit(features, target)
rfecv.transform(features)

array([[-0.76165969,  0.00850799, -1.72570086, ..., -1.15861018,
        -0.29639545, -0.88199355],
       [-0.46550514, -1.07500204, -1.65067126, ...,  0.42556217,
         0.53734944, -0.57765351],
       [ 1.36836958,  1.37940721,  0.24628915, ...,  0.70017292,
        -0.32395153, -0.17692642],
       ...,
       [ 0.39023196, -0.80331656,  0.71806912, ...,  0.44188555,
         0.13996708,  0.98775903],
```

```
       [ 0.44825266,  0.39508844,  0.01838076, ..., -0.74637847,
         0.11077605,  1.00252214],
       [ 1.15616404, -0.55383035, -0.20621076, ..., -0.1457644 ,
         0.19528325, -0.04260289]])
```

После того как мы выполнили рекурсивное устранение признаков, мы можем увидеть количество признаков, которые мы должны оставить:

```
# Количество наилучших признаков
rfecv.n_features_
```

```
9
```

Мы также можем увидеть, какие именно из этих признаков мы должны оставить:

```
# Какие категории самые лучшие
rfecv.support_
```

```
array([False,  True, False, False, False,  True, False, False, False,
       False,  True, False, False, False, False, False, False, False,
       False, False, False,  True, False, False, False, False, False,
       False, False,  True, False, False, False, False, False, False,
       False, False, False,  True, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False,  True, False, False,
       False, False, False, False, False, False, False, False, False,
        True, False, False, False, False, False, False, False, False,
       False,  True, False, False, False, False, False, False, False,
       False])
```

Мы даже можем взглянуть на ранги признаков:

```
# Ранжировать признаки от самого лучшего (1) до самого плохого
rfecv.ranking_
```

```
array([48,  1, 51, 57, 13,  1, 14, 85, 56, 65,  1,  6, 81, 41, 82,  2, 55,
       78, 20,  8, 92,  1, 90, 23, 59, 80, 24, 88, 39,  1, 42, 11, 34, 47,
       71, 69, 27, 72, 76,  1, 31, 91, 54, 61, 74, 75, 64, 89,  5, 19, 43,
       25,  3, 66, 52, 22, 26, 60, 12, 30, 87,  7, 37, 35, 15, 38, 18, 10,
       58,  1,  9, 28, 40, 33, 79, 68, 46, 16,  4, 84, 36,  1, 44, 21, 50,
       83, 63, 53, 86, 70, 62,  1, 77, 32, 67, 45, 29, 73, 49, 17])
```

Обсуждение

Это, вероятно, самый продвинутый рецепт в этой книге из уже рассмотренных нами, объединяющий ряд тем, которые нам еще предстоит подробно изучить. Однако его интуитивная идея достаточно проста, что позволяет нам применить этот метод здесь, а не откладывать до более поздней главы. Идея рекурсивного устранения признаков (RFE) заключается в неоднократной тренировке модели, которая содер-

жит некоторые параметры (так называемые веса или коэффициенты), такой как линейная регрессия или опорно-векторные машины. В первый раз, когда мы тренируем модель, мы включаем все признаки. Затем мы находим признак с наименьшим параметром (обратите внимание, что мы исходим из того, что признаки прошкалированы или стандартизированы), т. е. наименее важный, и удаляем его из набора признаков.

Тогда возникает очевидный вопрос: сколько признаков мы должны оставить? Мы можем (гипотетически) повторять этот цикл до тех пор, пока у нас не останется всего один признак. Более верный подход требует, чтобы мы включили новую концепцию под названием *перекрестная проверка* (cross-validation, CV). Мы будем обсуждать перекрестную проверку подробно в следующей главе, но ее общая идея состоит в следующем.

При наличии данных, содержащих цель, которую мы хотим предсказать, и матрицу признаков, во-первых, мы разделяем данные на две группы: тренировочный набор и тестовый набор. Во-вторых, мы тренируем нашу модель, используя тренировочный набор. В-третьих, мы делаем вид, что не знаем цели тестового набора, и применяем нашу модель к признакам тестового набора, чтобы предсказать значения тестового набора. Наконец, мы сравниваем наши предсказанные целевые значения с истинными целевыми значениями, чтобы оценить нашу модель.

Перекрестная проверка может использоваться для поиска оптимального количества признаков, которые следует оставить во время рекурсивного устранения признаков. В частности, в рекурсивном устранении признаков с перекрестной проверкой после каждой итерации мы используем перекрестную проверку для оценивания нашей модели. Если перекрестная проверка показывает, что после того как мы исключили признак, наша модель улучшилась, мы продолжаем следующий цикл. Однако, если она показывает, что после того как мы исключили признак, наша модель ухудшилась, мы помещаем этот признак обратно в набор признаков и отбираем эти признаки как наилучшие.

В библиотеке scikit-learn рекурсивное устранение признаков с перекрестной проверкой реализуется с помощью класса `RFECV`, который содержит ряд важных параметров. Параметр `estimator` определяет тип модели, которую мы хотим натренировать (например, линейную регрессию). Параметр `step` задает количество или долю признаков, которые необходимо удалять во время каждого цикла. Параметр `scoring` задает метрический показатель качества, который мы используем для оценивания нашей модели во время перекрестной проверки.

Дополнительные материалы для чтения

- "Рекурсивное устранение признаков с перекрестной проверкой", документация библиотеки scikit-learn (**http://bit.ly/2Ftuffz**).

Оценивание моделей

Введение

В этой главе мы рассмотрим стратегии оценивания качества моделей, созданных с помощью наших обучающихся алгоритмов. Может показаться странным заниматься оцениванием моделей до того, как мы обсудили их создание, но нашему безумию есть объяснение. Модели настолько полезны, насколько высоко качество их предсказаний, и, следовательно, наша основная цель не создание моделей как таковых (что очень легко), а создание высококачественных моделей (что довольно трудно). Именно поэтому, прежде чем мы приступим к исследованию несметного числа обучающихся алгоритмов, мы сначала сформулируем то, как мы будем оценивать модели, которые они производят.

11.1. Перекрестная проверка моделей

Задача

Требуется оценить, насколько хорошо модель будет работать в реальном мире.

Решение

Создать конвейер, который предварительно обрабатывает данные, тренирует модель, а затем оценивает ее с помощью перекрестной проверки:

```
# Загрузить библиотеки
from sklearn import datasets
from sklearn import metrics
from sklearn.model_selection import KFold, cross_val_score
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler

# Загрузить набор данных рукописных цифр
digits = datasets.load_digits()

# Создать матрицу признаков
features = digits.data
```

```
# Создать вектор целей
target = digits.target

# Создать стандартизатор
standardizer = StandardScaler()

# Создать объект логистической регрессии
logit = LogisticRegression()

# Создать конвейер, который стандартизирует,
# затем выполняет логистическую регрессию
pipeline = make_pipeline(standardizer, logit)

# Создать k-блочную перекрестную проверку
kf = KFold(n_splits=10, shuffle=True, random_state=1)

# Выполнить k-блочную перекрестную проверку
cv_results = cross_val_score(pipeline,  # Конвейер
                             features,  # Матрица признаков
                             target,    # Вектор целей
                             cv=kf,     # Метод перекрестной проверки
                             scoring="accuracy",     # Функция потери
                             n_jobs=-1) # Использовать все ядра CPU

# Вычислить среднее значение
cv_results.mean()

0.964931719428926
```

Обсуждение

При первом рассмотрении оценивание контролируемо обучающихся моделей может выглядеть прямолинейной задачей: натренировать модель, а затем вычислить, насколько хорошо она работает, используя некоторые показатели результативности (точность, квадратические ошибки и т. д.). Однако этот подход в корне ошибочен. Если мы тренируем модель, используя наши данные, а затем оцениваем, насколько хорошо она работает с этими же данными, то мы не достигаем желаемой цели. Наша цель состоит не в том, чтобы оценить, насколько хорошо модель работает с нашими тренировочными данными, а в том, насколько хорошо она работает с данными, которые она никогда не видела раньше (например, новый клиент, новое преступление, новое изображение). По этой причине наш метод оценивания должен помочь нам понять, насколько хорошо модели способны делать предсказания на основе данных, которые они никогда не видели раньше.

Одной из стратегий может быть откладывание среза данных для тестирования. Он называется *контрольным* (или *отложенным*) набором. В обычной проверке наши наблюдения (признаки и цели) разбиваются на два набора, традиционно называе-

мые *тренировочным набором* и *тестовым набором*. Мы берем тестовый набор и откладываем его в сторону, делая вид, что никогда его раньше не видели. Затем мы тренируем нашу модель, применяя наш тренировочный набор с использованием признаков и вектора целей, чтобы научить модель тому, как делать наилучшее предсказание. Наконец, мы симулируем, что никогда раньше не видели внешних данных, и оцениваем то, как наша модель, натренированная на нашем тренировочном наборе, работает на нашем тестовом наборе. Однако этот проверочный подход имеет два основных недостатка. Во-первых, результативность модели может сильно зависеть от того, какое количество наблюдений было выбрано для тестового набора. Во-вторых, модель не тренируется с использованием всех имеющихся данных и не оценивается по всем имеющимся данным.

Более оптимальная стратегия, которая преодолевает эти недостатки, называется *k-блочной перекрестной проверкой* (k-fold cross-validation, KFCV). В *k*-блочной перекрестной проверке мы разделяем данные на *k* частей, называемых "блоками". Модель обучается с помощью $k - 1$ блоков, объединенных в один тренировочный набор, и затем последний блок используется в качестве тестового набора. Мы повторяем это *k* раз, на очередном шаге в качестве тестового набора используя другой блок. Затем результативность модели для каждой из *k* итераций усредняется для получения общей меры.

В нашем решении мы провели *k*-блочную перекрестную проверку с использованием 10 блоков и вывели оценки в `cv_results`:

```
# Взглянуть на оценки для всех 10 блоков
cv_results
```

```
array([0.97222222, 0.97777778, 0.95555556, 0.95      , 0.95555556,
       0.98333333, 0.97777778, 0.96648045, 0.96089385, 0.94972067])
```

Когда мы используем *k*-блочную перекрестную проверку, следует учесть несколько важных моментов. Во-первых, *k*-блочная перекрестная проверка исходит из того, что каждое наблюдение было создано независимо от другого (т. е. данные являются независимыми одинаково распределенными [IID]). Если данные являются независимыми одинаково распределенными, то при назначении блоков рекомендуется наблюдения перетасовывать. В библиотеке scikit-learn можно установить `shuffle=True` для выполнения перетасовки.

Во-вторых, когда мы используем *k*-блочную перекрестную проверку для оценивания классификатора, часто полезно иметь блоки, содержащие примерно одинаковый процент наблюдений из каждого отдельного целевого класса. Такая проверка называется *стратифицированной k-блочной*. Например, если бы наш вектор целей содержал пол, и 80% наблюдений были бы мужского пола, то каждый блок содержал бы 80% наблюдений с мужским полом и 20% наблюдений с женским полом. В библиотеке scikit-learn можно проводить стратифицированную *k*-блочную перекрестную проверку, поменяв класс `KFold` на класс `StratifiedKFold`.

Наконец, при использовании перекрестно-проверочных наборов или перекрестной проверки важно предварительно обработать данные на основе тренировочного на-

бора, а затем применить эти преобразования к обоим наборам: тренировочному и тестовому. Например, когда мы выполняем подгонку (с помощью метода `fit`) нашего объекта стандартизации, `standardizer`, мы вычисляем среднее и дисперсию только тренировочного набора. Затем мы применяем это преобразование (с помощью метода `transform`) и к тренировочному, и к тестовому наборам:

```
# Импортировать библиотеку
from sklearn.model_selection import train_test_split

# Создать тренировочный и тестовый наборы
features_train, features_test, target_train, target_test = train_test_split(
    features, target, test_size=0.1, random_state=1)

# Выполнить подгонку стандартизатора к тренировочному набору
standardizer.fit(features_train)

# Применить к обоим наборам: тренировочному и тестовому
features_train_std = standardizer.transform(features_train)
features_test_std = standardizer.transform(features_test)
```

Здесь мы делаем вид, будто тестовый набор содержит неизвестные данные. Если мы выполним подгонку обоих наших препроцессоров, используя наблюдения из тренировочного и тестового наборов, то часть информации из тестового набора просочится в наш тренировочный набор. Это правило применимо для любого шага предобработки, например для отбора признаков.

Пакет pipeline библиотеки scikit-learn упрощает задачу при использовании методов перекрестной проверки. Сначала мы создаем конвейер, который предварительно обрабатывает данные (например, `standardizer`), а затем тренирует модель (логистическая регрессия, `logit`):

```
# Создать конвейер
pipeline = make_pipeline(standardizer, logit)
```

Затем мы запускаем *k*-блочную перекрестную проверку, используя этот конвейер, и библиотека scikit-learn делает всю работу за нас:

```
# Выполнить k-блочную перекрестную проверку
cv_results = cross_val_score(pipeline,  # Конвейер
                             features,  # Матрица признаков
                             target,    # Вектор целей
                             cv=kf,     # Метод перекрестной проверки
                             scoring="accuracy", # Функция потери
                             n_jobs=-1) # Использовать все ядра CPU
```

Метод `cross_val_score` сопровождается тремя параметрами, которые мы не обсуждали, и их стоит отметить. Параметр `cv` определяет нашу методику перекрестной проверки. *K*-блочный метод является наиболее распространенным на сегодняшний день, но есть и другие, такие как перекрестная проверка с исключением по одному (leave-one-out), где количество блоков *k* равно количеству наблюдений. Параметр

`scoring` определяет наш метрический показатель успеха, некоторые из метрических показателей успеха обсуждаются в других рецептах в этой главе. Наконец, `n_jobs=-1` предписывает библиотеке scikit-learn использовать все доступные ядра CPU. Например, если ваш компьютер имеет четыре ядра (распространенное количество для ноутбуков), то для ускорения работы библиотека scikit-learn будет использовать все четыре ядра сразу.

Дополнительные материалы для чтения

- "Почему каждый статистик обязан разбираться в перекрестной проверке", блог-пост (**http://bit.ly/2Fzhz6X**).
- "Перекрестная проверка пошла не так", пост в Github (**http://bit.ly/2FzfIiw**).

11.2. Создание базовой регрессионной модели

Задача

В качестве ориентира требуется простая базовая регрессионная модель для сравнения с вашей моделью.

Решение

Использовать объект класса `DummyRegressor` библиотеки scikit-learn, которая создает простую модель фиктивной регрессии для использования в качестве ориентира:

```
# Загрузить библиотеки
from sklearn.datasets import load_boston
from sklearn.dummy import DummyRegressor
from sklearn.model_selection import train_test_split

# Загрузить данные
boston = load_boston()

# Создать матрицу признаков и вектор целей
features, target = boston.data, boston.target

# Разбить на тренировочный и тестовый наборы
features_train, features_test, target_train, target_test =
    train_test_split(features, target, random_state=0)

# Создать фиктивный регрессор
dummy = DummyRegressor(strategy='mean')

# "Натренировать" фиктивный регрессор
dummy.fit(features_train, target_train)
```

```
# Получить оценку коэффициента детерминации (R-squared)
dummy.score(features_test, target_test)
```

```
-0.0011193592039553391
```

Для сравнения, мы тренируем нашу модель и вычисляем оценку результативности:

```
# Загрузить библиотеку
from sklearn.linear_model import LinearRegression

# Натренировать простую линейно-регрессионную модель
ols = LinearRegression()
ols.fit(features_train, target_train)

# Получить оценку коэффициента детерминации (R-squared)
ols.score(features_test, target_test)
```

```
0.63536207866746675
```

Обсуждение

Фиктивный регрессор `DummyRegressor` позволяет нам создавать очень простую модель, которую можно использовать в качестве ориентира для сравнения с нашей реальной моделью. Это часто может быть полезно с целью симулировать "наивный" существующий предсказательный процесс в программном продукте или системе. Например, программный продукт может быть изначально жестко запрограммирован допускать, что все новые пользователи будут тратить $100 в первый месяц, независимо от их признаков. Если мы закодируем это допущение в базовую модель, то сможем вещественно заявить о преимуществах применения подхода на основе машинного самообучения.

В классе `DummyRegressor` используется параметр `strategy` для задания метода предсказания, включая среднее или медианное значение в тренировочном наборе. Более того, если присвоить параметру `strategy` значение `constant` и учесть параметр `constant`, мы сможем заставить фиктивный регрессор предсказывать некоторое постоянное значение для каждого наблюдения:

```
# Создать фиктивный регрессор, который
# предсказывает 20 для всех наблюдений
clf = DummyRegressor(strategy='constant', constant=20)
clf.fit(features_train, target_train)

# Вычислить оценку
clf.score(features_test, target_test)
```

```
-0.065105020293257265
```

Одно небольшое замечание о методе `score`. По умолчанию данный метод возвращает оценку коэффициента детерминации (R-squared, R^2)[1]:

$$R^2 = 1 - \frac{\sum_i (y_i + \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2},$$

где y_i — истинное целевое значение наблюдения; $\hat{y}_i$ — предсказанное значение; $\bar{y}$ — среднее значение вектора целей.

Чем ближе R^2 к 1, тем больше дисперсия в векторе целей, которая объясняется признаками.

Дополнительные материалы для чтения

- Документация scikit-learn по классу фиктивного регрессора DummyRegressor (**https://bit.ly/2wIIXfq**).

11.3. Создание базовой классификационной модели

Задача

В качестве ориентира требуется простой базовый классификатор для сравнения с вашей моделью.

Решение

Использовать класс `DummyClassifier` библиотеки scikit-learn:

```
# Загрузить библиотеки
from sklearn.datasets import load_iris
from sklearn.dummy import DummyClassifier
from sklearn.model_selection import train_test_split

# Загрузить данные
iris = load_iris()
```

[1] На практике широко используются два определения R^2. Распространенная ошибка состоит в том, что часто путают два определения: квадратичный коэффициент корреляции Пирсона и коэффициент детерминации. Первый является мерой корреляции между двумя переменными, которая принимает значения от +1 до 0: $R^2 = \frac{\text{cov}(X, Y)^2}{\sigma_X^2 \sigma_Y^2}$, где cov(*X*, *Y*) — это ковариация *X* и *Y*, мера того, каким образом два набора данных варьируются совместно, в то время как σ_X и σ_Y — это стандартные отклонения, меры того, насколько каждый набор варьируется индивидуально. Последний показатель R^2 из двух не имеет нижней границы 0, как у квадратичной корреляции Пирсона. — *Прим. перев.*

```
# Создать матрицу признаков и вектор целей
features, target = iris.data, iris.target

# Разбить на тренировочный и тестовый наборы
features_train, features_test, target_train, target_test =
    train_test_split(features, target, random_state=0)

# Создать фиктивный классификатор
dummy = DummyClassifier(strategy='uniform', random_state=1)

# "Натренировать" модель
dummy.fit(features_train, target_train)

# Получить оценку точности
dummy.score(features_test, target_test)
```

```
0.42105263157894735
```

Путем сопоставления базового классификатора с нашим натренированным классификатором можно увидеть улучшение:

```
# Загрузить библиотеку
from sklearn.ensemble import RandomForestClassifier

# Создать классификатор случайного леса
classifier = RandomForestClassifier()

# Натренировать модель
classifier.fit(features_train, target_train)

# Получить оценку точности
classifier.score(features_test, target_test)
```

```
0.94736842105263153
```

Обсуждение

Общепринятая мера результативности классификатора — показатель, насколько он лучше, чем случайное угадывание. Фиктивный классификатор `DummyClassifier` библиотеки scikit-learn облегчает выполнение такого сравнения. Параметр `strategy` предоставляет ряд вариантов для генерации значений. Есть две особенно полезных стратегии. Во-первых, стратегия `stratified` делает предсказания, которые пропорциональны долям классов вектора целей в тренировочном наборе (т. е. если 20% наблюдений в тренировочных данных составляют женщины, то фиктивный классификатор `DummyClassifier` будет предсказывать женщин в 20% случаях). Во-вторых, стратегия `uniform` будет генерировать предсказания случайным образом между разными классами. Например, если 20% наблюдений — женщины и 80% — муж-

чины, то стратегия `uniform` будет давать предсказания, которые составят 50% женщин и 50% мужчин.

Дополнительные материалы для чтения

- Документация scikit-learn по классу фиктивного классификатора DummyClassifier (**http://bit.ly/2Fr178G**).

11.4. Оценивание предсказаний бинарного классификатора

Задача

Дана натренированная классификационная модель, и требуется оценить ее качество.

Решение

Применить метод `cross_val_score` библиотеки scikit-learn для проведения перекрестной проверки, используя при этом параметр `scoring` для определения одного из нескольких метрических показателей результативности, включая точность, прецизионность, полноту и оценку F_1.

Точность является общепринятым метрическим показателем результативности. Это просто доля правильно предсказанных наблюдений:

$$\text{точность} = \frac{TP + TN}{TP + TN + FP + FN},$$

где:

- TP — количество истинноположительных исходов; наблюдения, которые являются частью положительного класса (имеет заболевание, приобретен товар и т. д.) и которые мы предсказали правильно;
- TN — количество истинноотрицательных исходов; наблюдения, которые являются частью отрицательного класса (не имеет этого заболевания, не преобретен товар и т. д.) и которые мы предсказали правильно;
- FP — количество ложноположительных исходов, также называется ошибкой 1-го рода; наблюдения, предсказанные как часть положительного класса, которые на самом деле являются частью отрицательного класса;
- FN — количество ложноотрицательных исходов, также называется ошибкой 2-го рода; наблюдения, предсказанные как часть отрицательного класса, которые на самом деле являются частью положительного класса.

Мы можем измерить точность в трехблочной (количество блоков, принятое по умолчанию) перекрестной проверке, установив параметр `scoring="accuracy"`:

```
# Загрузить библиотеки
from sklearn.model_selection import cross_val_score
```

```
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import make_classification

# Сгенерировать матрицу признаков и вектор целей
X, y = make_classification(n_samples = 10000,
                           n_features = 3,
                           n_informative = 3,
                           n_redundant = 0,
                           n_classes = 2,
                           random_state = 1)

# Создать объект логистической регрессии
logit = LogisticRegression()

# Перекрестно проверить модель, используя показатель точности
cross_val_score(logit, X, y, scoring="accuracy")
```

```
array([ 0.95170966, 0.9580084 , 0.95558223])
```

Привлекательность точности заключается в том, что она имеет интуитивное и простое объяснение на русском языке: доля наблюдений, предсказанных правильно. Однако в реальном мире часто наши данные имеют несбалансированные классы (например, 99.9% наблюдений относятся к классу 1 и только 0.1% — к классу 2). При наличии несбалансированных классов точность страдает от парадокса, когда модель высокоточна, но не обладает предсказательной силой. Например, представьте, что мы пытаемся предсказать наличие очень редкого рака, который встречается у 0.1% населения. После тренировки нашей модели мы обнаруживаем, что точность находится на уровне 95%. Вместе с тем 99.9% людей не болеют раком: если мы просто создадим модель, которая "предсказывает", что никто не имеет такой формы рака, то наша наивная модель будет на 4.9% точнее, но совершенно очевидно, не будет способна ничего предсказывать. По этой причине мы часто мотивированы использовать другие метрические показатели, такие как прецизионность, полнота и оценка F_1.

Прецизионность — это доля каждого наблюдения, предсказанного положительно, которое на самом деле положительно. Эту меру можно представить как измерительный шум в наших предсказаниях, т. е. когда мы предсказываем что-то положительное, насколько вероятно, что мы будем правы. Модели с высокой прецизионностью пессимистичны, т. е. они предсказывают наблюдение как принадлежащее положительному классу, только когда они очень уверены в этом. Формально прецизионность рассчитывается по формуле:

$$\text{прецизионность} = \frac{TP}{TP + FP}.$$

```
# Перекрестно проверить модель, используя показатель прецизионности
cross_val_score(logit, X, y, scoring="precision")
```

```
array([ 0.95252404, 0.96583282, 0.95558223])
```

Полнота — это доля каждого положительного наблюдения, которое по-настоящему положительно. Полнота измеряет способность модели идентифицировать наблюдение положительного класса. Модели с высокой полнотой оптимистичны, т. е. они имеют низкую планку для предсказания, что наблюдение находится в положительном классе:

$$\text{полнота} = \frac{TP}{TP + FN}.$$

```
# Перекрестно проверить модель, используя показатель полноты
cross_val_score(logit, X, y, scoring="recall")
```

```
array([ 0.95080984, 0.94961008, 0.95558223])
```

Если вы впервые встречаете показатели прецизионности и полноты, вы должны затратить немного времени, чтобы полностью в них разобраться. Это один из их недостатков данных характеристик по сравнению с точностью; прецизионность и полнота менее интуитивны. Почти всегда нужен какой-то баланс между прецизионностью и полнотой, и эта роль отводится оценке F_1. *Оценка* F_1 является средним гармоническим (видом среднего значения, используемого для соотношений):

$$F_1 = 2 \times \frac{\text{прецизионность} \times \text{полнота}}{\text{прецизионность} + \text{полнота}}.$$

Она является мерой правильности, достигаемой в положительном предсказании, т. е. сколько наблюдений, помеченных как положительные, на самом деле положительные:

```
# Перекрестно проверить модель, используя показатель f1
cross_val_score(logit, X, y, scoring="f1")
```

```
array([ 0.95166617, 0.95765275, 0.95558223])
```

Обсуждение

В качестве метрического оценочного показателя точность имеет некоторые ценные свойства, в особенности свою простую интуитивную понятность. Вместе с тем более верные метрические показатели часто основаны на некотором балансе прецизионности и полноты, т. е. компромиссном соотношении между оптимизмом и пессимизмом нашей модели. Оценка F_1 представляет собой баланс между полнотой и прецизионностью, где относительные вклады обеих характеристик равны.

В качестве альтернативы метода `cross_val_score`, если у нас уже есть истинные значения y и предсказанные значения $\hat{y}$, можно вычислить метрические показатели, такие как точность и полнота, напрямую:

```
# Загрузить библиотеку
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

```
# Разбить на тренировочный и тестовый наборы
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size=0.1,
                                                    random_state=1)

# Предсказать значения для тренировочного вектора целей
y_hat = logit.fit(X_train, y_train).predict(X_test)

# Вычислить точность
accuracy_score(y_test, y_hat)

0.94699999999999995
```

Дополнительные материалы для чтения

- "Парадокс показателя точности", Википедия (**http://bit.ly/2FxTpK0**).

11.5. Оценивание порогов бинарного классификатора

Задача

Требуется оценить двоичный классификатор и различные вероятностные пороги.

Решение

Кривая рабочей характеристики приемника (receiver operating characteristic, ROC-кривая) часто используется для оценки качества бинарного классификатора. Кривая ROC сравнивает наличие истинноположительных и ложноположительных исходов на каждом вероятностном пороге (т. е. вероятности, при которой наблюдение предсказывается как класс). Построив график кривой ROC, можно увидеть, насколько хорошо работает модель (рис. 11.1). Классификатор, который правильно предсказывает каждое наблюдение, будет выглядеть как сплошная светло-серая кривая на приведенном ниже графике, которая резко поднимается наверх. Классификатор, который предсказывает наугад, имеет вид диагональной прямой. Чем лучше модель, тем ближе она к сплошной линии. В библиотеке scikit-learn для вычисления истинно- и ложноположительных исходов на каждом пороге можно использовать метод `roc_curve`, а затем вывести их на график:

```
# Загрузить библиотеки
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_curve, roc_auc_score
from sklearn.model_selection import train_test_split
```

```
# Создать матрицу признаков и вектор целей
features, target = make_classification(n_samples=10000,
                                       n_features=10,
                                       n_classes=2,
                                       n_informative=3,
                                       random_state=3)

# Разбить на тренировочный и тестовый наборы
features_train, features_test, target_train, target_test =
   train_test_split(features, target, test_size=0.1, random_state=1)

# Создать логистический регрессионный классификатор
logit = LogisticRegression()

# Натренировать модель
logit.fit(features_train, target_train)

# Получить предсказанные вероятности
target_probabilities = logit.predict_proba(features_test)[:,1]

# Создать доли истинно- и ложноположительных исходов
false_positive_rate, true_positive_rate, threshold =
    roc_curve(target_test, target_probabilities)

# Построить график кривой ROC
plt.title("Кривая ROC")
plt.plot(false_positive_rate, true_positive_rate)
plt.plot([0, 1], ls="--")
plt.plot([0, 0], [1, 0] , c=".7")
plt.plot([1, 1] , c=".7")
plt.ylabel("Доля истинноположительных исходов")
plt.xlabel("Доля ложноположительных исходов")
plt.show()
```

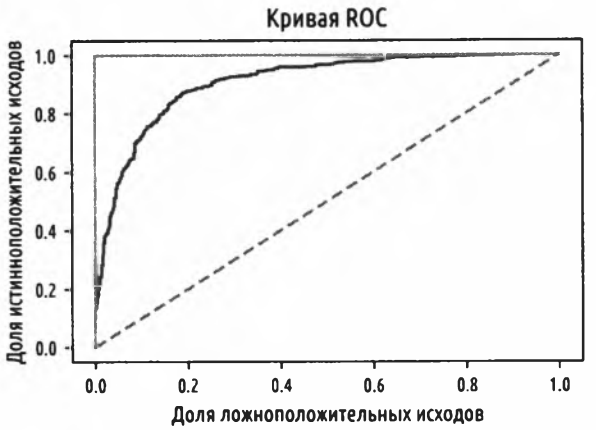


Рис. 11.1

Обсуждение

До сих пор мы рассматривали только модели, основанные на предсказываемых ими значениях. Однако во многих обучающихся алгоритмах эти предсказанные значения основаны на вероятностных оценках, т. е. каждому наблюдению дается явная вероятность принадлежности в каждом классе. В нашем решении, чтобы увидеть предсказанные вероятности для первого наблюдения, мы можем применить метод `predict_probe`:

```
# Получить предсказанные вероятности
logit.predict_proba(features_test)[0:1]
```

```
array([[ 0.8688938, 0.1311062]])
```

Классы можно увидеть с помощью атрибута `classes_`:

```
logit.classes_
```

```
array([0, 1])
```

В этом примере первое наблюдение имеет ~87% шанс быть в отрицательном классе (0) и 13% шанс быть в положительном классе (1). По умолчанию библиотека scikit-learn предсказывает, что наблюдение является частью положительного класса, если вероятность больше 0.5 (эта величина называется *вероятностным порогом*). Однако вместо принятия средней позиции нередко требуется явным образом сместить нашу модель, чтобы использовать другой порог по неким существенным причинам. Например, если ложноположительный исход очень дорого обходится нашей компании, мы можем предпочесть модель с высоким вероятностным порогом. Нам не удастся предсказать некоторые положительные исходы, но когда наблюдение предсказано как положительное, мы можем быть очень уверены в том, что предсказание является правильным. Этот компромисс представлен в доле истинноположительных (true positive rate, TPR) и доле ложноположительных исходов (false positive rate, FPR). Доля истинноположительных исходов — это количество наблюдений, предсказанных правильно, деленное на все истинноположительные наблюдения:

$$\text{TPR} = \frac{\text{Истинноположительные}}{\text{Истинноположительные} + \text{Ложноотрицательные}}.$$

Доля ложноположительных исходов — это количество неправильно предсказанных положительных исходов, деленное на все истинноотрицательные исходы:

$$\text{FPR} = \frac{\text{Ложноположительные}}{\text{Ложноположительные} + \text{Истинноотрицательные}}.$$

Кривая ROC представляет соответствующие доли TPR и FPR для каждого вероятностного порога. Например, в нашем решении примерный порог 0.50 имеет долю истинноположительных (TPR) 0.81 и долю ложноположительных (FPR) 0.15:

```
print("Порог:", threshold[116])
print("Доля истинноположительных:", true_positive_rate[116])
print("Доля ложноположительных:", false_positive_rate[116])
```

```
Порог: 0.5282247778873397
Доля истинноположительных: 0.810204081632653
Доля ложноположительных: 0.15490196078431373
```

Однако если мы увеличим порог до ~80% (т. е. повысим уверенность модели перед тем, как она будет предсказывать наблюдение в качестве положительного), то доля истинноположительных (TPR) заметно упадет, и то же самое касается доли ложноположительных (FPR):

```
print("Порог:", threshold[45])
print("Доля истинноположительных:", true_positive_rate[45])
print("Доля ложноположительных:", false_positive_rate[45])
```

```
Порог: 0.8080195665631111
Доля истинноположительных: 0.563265306122449
Доля ложноположительных: 0.047058823529411764
```

Это связано с тем, что наше более высокое требование для предсказания в положительном классе заставило модель не идентифицировать ряд положительных наблюдений (более низкая доля истинноположительных TPR), но также уменьшить шум от отрицательных наблюдений, предсказываемых как положительные (более низкая доля ложноположительных FPR).

В дополнение к возможности визуализировать компромиссное соотношение между долями TPR и FPR, кривая ROC также может использоваться в качестве общего метрического показателя для всей модели. Чем лучше модель, тем выше кривая и тем больше площадь под кривой. По этой причине обычно вычисляется площадь под кривой ROC (area under the curve, AUCROC) для оценки общего качества модели при всех возможных пороговых значениях. Чем ближе AUCROC к 1, тем лучше модель. В библиотеке scikit-learn можно рассчитать AUCROC с помощью метода `roc_auc_score`:

```
# Вычислить площадь под кривой
roc_auc_score(target_test, target_probabilities)
```

```
0.90733893557422962
```

Дополнительные материалы для чтения

- "Кривые ROC в Python и R", платформы Alteryx для науки о данных и машинного самообучения (**http://bit.ly/2FuqoyV**).
- "Площадь под кривой ROC", интерпретация диагностических тестов (медицинский центр университета Небраски, **http://bit.ly/2FxTrl6**).

11.6. Оценивание предсказаний мультиклассового классификатора

Задача

Дана модель, которая предсказывает три класса или более, и требуется оценить ее результативность.

Решение

Использовать перекрестную проверку с оценочным метрическим показателем, способным справляться с более чем двумя классами:

```
# Загрузить библиотеки
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import make_classification

# Сгенерировать матрицу признаков и вектор целей
features, target = make_classification(n_samples = 10000,
                                       n_features = 3,
                                       n_informative = 3,
                                       n_redundant = 0,
                                       n_classes = 3,
                                       random_state = 1)

# Создать объект логистической регрессии
logit = LogisticRegression()

# Перекрестно проверить модель, используя показатель точности
cross_val_score(logit, features, target, scoring='accuracy')

array([ 0.83653269, 0.8259826 , 0.81308131])
```

Обсуждение

Когда имеются сбалансированные классы (например, содержащие ориентировочно равное количество наблюдений в каждом классе вектора целей), точность — так же, как и в условиях двоичного класса — является простым и интерпретируемым параметром для оценочного метрического показателя. Точность — это количество правильных предсказаний, деленное на количество наблюдений, и она работает хорошо и в мультиклассовой, и в бинарной конфигурации. Однако когда имеются несбалансированные классы (широко распространенный сценарий), мы должны использовать другие оценочные метрические показатели.

Многие встроенные в библиотеку scikit-learn метрические показатели предназначены для оценивания бинарных классификаторов. Однако многие из этих метрических показателей могут быть расширены для использования в условиях более двух

классов. Прецизионность, полнота и оценки F_1 являются полезными метрическими показателями, которые мы уже подробно рассмотрели в предыдущих рецептах. Хотя все они изначально были разработаны для бинарных классификаторов, их можно применить и к мультиклассовой конфигурации, рассматривая наши данные как набор бинарных классов. Это позволяет применять метрические показатели к каждому классу, как если бы он был единственным классом в данных, а затем агрегировать оценочные показатели для всех классов путем их усреднения:

```
# Перекрестно проверить модель,
# используя макроусредненную оценку F1
cross_val_score(logit, features, target, scoring='f1_macro')
```

```
array([ 0.83613125, 0.82562258, 0.81293539])
```

В этом фрагменте кода постфикс `_macro` относится к методу, используемому для усреднения оценок из классов:

- `macro` — рассчитать среднее метрических оценок для каждого класса, взвешивая каждый класс одинаково;
- `weighted` — рассчитать среднее метрических оценок для каждого класса, взвешивая каждый класс пропорционально его размеру в данных;
- `micro` — рассчитать среднее метрических оценок для каждой комбинации "класс — наблюдение".

11.7. Визуализация результативности классификатора

Задача

Даны предсказанные классы и истинные классы тестовых данных, и требуется визуально сопоставить качество модели.

Решение

Использовать матрицу ошибок (несоответствий), которая сравнивает предсказанные и истинные классы (рис. 11.2):

```
# Загрузить библиотеки
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import datasets
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
import pandas as pd

# Загрузить данные
iris = datasets.load_iris()
```

```
# Создать матрицу признаков
features = iris.data

# Создать вектор целей
target = iris.target

# Создать список имен целевых классов
class_names = iris.target_names

# Создать тренировочный и тестовый наборы
features_train, features_test, target_train, target_test =
   train_test_split(features, target, random_state=1)

# Создать объект логистической регрессии
classifier = LogisticRegression()

# Натренировать модель и сделать предсказания
target_predicted = classifier.fit(features_train,
    target_train).predict(features_test)

# Создать матрицу ошибок
matrix = confusion_matrix(target_test, target_predicted)

# Создать фрейм данных pandas
dataframe = pd.DataFrame(matrix, index=class_names, columns=class_names)

# Создать тепловую карту
sns.heatmap(dataframe, annot=True, cbar=None, cmap="Blues")
plt.title("Матрица ошибок")
plt.tight_layout()
plt.ylabel("Истинный класс")
plt.xlabel("Предсказанный класс")
plt.show()
```

Рис. 11.2

Обсуждение

Матрицы ошибок (несоответствий) — это простая и эффективная визуализация качества работы классификатора. Одним из основных преимуществ матриц ошибок является их интерпретируемость. Каждый столбец матрицы (часто визуализируемый как тепловая карта) представляет предсказанные классы, в то время как каждая строка показывает истинные классы. Конечный результат заключается в том, что каждая ячейка является одной возможной комбинацией предсказанных и истинных классов. Это, вероятно, лучше всего объяснить на примере. В нашем решении левая верхняя ячейка — это количество наблюдений, предсказанных как цветок ириса щетинистого *Iris setosa* (обозначенные столбцом), которые действительно являются цветками ириса щетинистого (обозначенные строкой). Это означает, что модель точно предсказала все цветки ириса щетинистого. Однако модель не так хорошо справилась с предсказанием цветка ириса вергинского *Iris virginica*. Правая нижняя ячейка указывает на то, что модель успешно предсказала девять наблюдений цветка ириса вергинского, но (глядя на одну ячейку вверх) при этом предсказала шесть цветков ириса вергинского, которые в действительности были цветками ириса разноцветного *Iris versicolor*.

В отношении матриц ошибок стоит отметить три момента. Во-первых, идеальная модель будет иметь значения вдоль диагонали и нули в остальных ячейках. Плохая модель будет выглядеть так, будто количество наблюдений равномерно распределено по ячейкам. Во-вторых, матрица ошибок позволяет увидеть не только, где модель была неправильной, но и как она была неправильной, т. е. мы можем взглянуть на шаблоны неправильной классификации. Например, наша модель легко смогла различить *Iris virginica* и *Iris setosa*, но она гораздо хуже справилась с классификацией *Iris virginica* и *Iris versicolor*. Наконец, матрицы ошибок работают с любым количеством классов (хотя, если бы в нашем векторе целей был миллион классов, визуализацию матрицы ошибок было бы трудно прочитать).

Дополнительные материалы для чтения

- "Матрица ошибок", Википедия (**http://bit.ly/2FuGKaP**).
- Документация библиотеки scikit-learn по матрице ошибок (**http://bit.ly/2DmnICk**).

11.8. Оценивание регрессионных моделей

Задача

Требуется оценить результативность регрессионной модели.

Решение

Использовать среднеквадратическую ошибку (mean squared error, MSE):

```
# Загрузить библиотеки
from sklearn.datasets import make_regression
```

```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LinearRegression

# Сгенерировать матрицу признаков, вектор целей
features, target = make_regression(n_samples = 100,
                                   n_features = 3,
                                   n_informative = 3,
                                   n_targets = 1,
                                   noise = 50,
                                   coef = False,
                                   random_state = 1)

# Создать объект линейной регрессии
ols = LinearRegression()

# Перекрестно проверить линейную регрессию,
# используя (отрицательный) показатель MSE
cross_val_score(ols, features, target, scoring='neg_mean_squared_error')

array([-1718.22817783, -3103.4124284 , -1377.17858823])
```

Еще одним распространенным метрическим показателем регрессии является коэффициент детерминации R^2:

```
# Перекрестно проверить линейную регрессию,
# используя показатель R-квадрат
cross_val_score(ols, features, target, scoring='r2')

array([ 0.87804558, 0.76395862, 0.89154377])
```

Обсуждение

MSE является одним из наиболее распространенных оценочных показателей для регрессионных моделей. Формально MSE имеет вид:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2,$$

где n — число наблюдений; y_i — истинное целевое значение, которое мы пытаемся предсказать для наблюдения i; $\hat{y}_i$ — предсказанное моделью значение для y_i.

MSE — это мера квадратичной суммы всех расстояний между предсказанными и истинными значениями. Чем выше значение MSE, тем больше общая квадратичная ошибка, и тем хуже модель. Существует ряд математических преимуществ от возведения ошибок в квадрат (одно из них заключается в том, что модель заставляет все значения ошибок быть положительными), но одно часто несознаваемое последствие состоит в том, что возведение в квадрат штрафует несколько крупных ошибок больше, чем множество мелких ошибок, даже если абсолютные значения

ошибки одинаковые. Например, представьте две модели, *A* и *B*, каждая с двумя наблюдениями:

- модель *A* имеет ошибки 0 и 10, и, следовательно, ее MSE $0^2 + 10^2 = 100$;
- модель *B* имеет две ошибки по 5 каждая, и, следовательно, ее MSE $5^2 + 5^2 = 50$.

Обе модели имеют одинаковую общую ошибку 10; однако MSE будет считать модель *A* ($\text{MSE} = 100$) хуже, чем модель *B* ($\text{MSE} = 50$). На практике это последствие редко является проблемой (и в действительности может быть теоретически полезным), и MSE прекрасно работает как оценочная метрика.

Одно важное замечание: по умолчанию в библиотеке scikit-learn аргументы параметрической переменной `scoring` исходят из того, что более высокие значения лучше, чем более низкие. Однако это не относится к MSE, где более высокие значения означают худшую модель. По этой причине библиотека scikit-learn обращается к отрицательному показателю MSE с помощью аргумента `neg_mean_squared_error`.

Общепринятым альтернативным метрическим оценочным показателем регрессии является R^2, который измеряет величину дисперсии в векторе целей, которая объясняется моделью:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y}_i)^2},$$

где y_i — истинное целевое значение для *i*-го наблюдения; $\hat{y}_i$ — предсказанное значение для *i*-го наблюдения; $\bar{y}$ — среднее значение вектора целей. Чем ближе R^2 к 1.0, тем лучше модель.

Дополнительные материалы для чтения

- "Среднеквадратическая ошибка", Википедия (**http://bit.ly/2HgALnc**).
- "Коэффициент детерминации", Википедия (**http://bit.ly/2HjW7Qn**).

11.9. Оценивание кластеризующих моделей

Задача

Применен неконтролируемо обучающийся алгоритм с целью кластеризации данных. Теперь требуется узнать, насколько хорошо он справился со своей работой.

Решение

Короткий ответ заключается в том, что вы, вероятно, не сможете это сделать, по крайней мере не тем способом, каким хотелось бы.

Тем не менее одним из вариантов является оценка кластеризации с использованием силуэтных коэффициентов, которые оценивают качество кластеров:

```
# Загрузить библиотеки
import numpy as np
from sklearn.metrics import silhouette_score
from sklearn import datasets
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

# Сгенерировать матрицу признаков
features, _ = make_blobs(n_samples = 1000,
                         n_features = 10,
                         centers = 2,
                         cluster_std = 0.5,
                         shuffle = True,
                         random_state = 1)

# Кластеризовать данные, используя алгоритм k средних,
# чтобы предсказать классы
model = KMeans(n_clusters=2, random_state=1).fit(features)

# Получить предсказанные классы
target_predicted = model.labels_

# Оценить модель
silhouette_score(features, target_predicted)
```

```
0.89162655640721422
```

Обсуждение

Оценивание контролируемой модели сводится к сопоставлению предсказаний (например, значений классов или количественных значений) с соответствующими истинными значениями в векторе целей. Вместе с тем наиболее распространенной мотивацией для использования кластеризующих методов является то, что данные не имеют вектора целей. Существует ряд метрических оценочных показателей кластеризации, которые требуют вектора целей, но опять же использование неконтролируемых подходов к самообучению, таких как кластеризация, когда имеется доступный вам вектор целей, вероятно, мешает вам без необходимости.

В то время как при отсутствии вектора целей мы не можем оценить предсказания относительно истинных значений, мы способны оценить природу кластеров как таковых. Интуитивно можно представить, что "хорошие" кластеры имеют очень малые расстояния между наблюдениями в одном кластере (т. е. плотные кластеры) и большие расстояния между разными кластерами (т. е. хорошо разделенные кластеры). Силуэтные коэффициенты обеспечивают одно-единственное значение, из-

меряющее обе черты. Формально силуэтный коэффициент i-го наблюдения равняется:

$$s_i = \frac{b_i - a_i}{\max(a_i, b_i)},$$

где s_i — силуэтный коэффициент для наблюдения i; a_i — среднее расстояние между i и всеми наблюдениями одного класса; b_i — среднее расстояние между i и всеми наблюдениями из ближайшего кластера другого класса. Значение, возвращаемое параметром `silhouette_score`, является средним силуэтным коэффициентом для всех наблюдений. Силуэтные коэффициенты варьируются от -1 до 1, причем 1 указывает на плотные, хорошо разделенные кластеры.

Дополнительные материалы для чтения

- Документация библиотеки scikit-learn по оценочному показателю `silhouette_score` (**http://bit.ly/2BEVQV5**).

11.10. Создание собственного оценочного метрического показателя

Задача

Требуется оценить модель с помощью созданного метрического показателя.

Решение

Создать метрический показатель как функцию и конвертировать его в оценочную функцию, используя фабричную функцию `make_scorer` библиотеки scikit learn:

```
# Загрузить библиотеки
from sklearn.metrics import make_scorer, r2_score
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Ridge
from sklearn.datasets import make_regression

# Сгенерировать матрицу признаков и вектор целей
features, target = make_regression(n_samples = 100,
                                   n_features = 3,
                                   random_state = 1)

# Создать тренировочный и тестовый наборы
features_train, features_test, target_train, target_test =
    train_test_split(features, target, test_size=0.10, random_state=1)

# Создать собственный метрический показатель
def custom_metric(target_test, target_predicted):
```

```
    # Вычислить оценочный показатель r-квадрат
    r2 = r2_score(target_test, target_predicted)
    # Вернуть оценочный показатель r-квадрат
    return r2

# Создать оценочную функцию и установить,
# что чем выше оценки, тем они лучше
score = make_scorer(custom_metric, greater_is_better=True)

# Создать объект гребневой регрессии
classifier = Ridge()

# Натренировать гребневую регрессионную модель
model = classifier.fit(features_train, target_train)

# Применить собственную оценочную функцию
score(model, features_test, target_test)
```

```
0.99979061028820582
```

Обсуждение

Хотя библиотека scikit-learn имеет ряд встроенных метрических показателей для оценивания результативности модели, часто бывает полезно определить собственные метрические показатели. Библиотека scikit-learn упрощает эту задачу с помощью фабричной функции `make_scorer`. Во-первых, мы определяем функцию, которая принимает два аргумента — целевой вектор полевых наблюдений и наши предсказанные значения — и выдает некоторую оценку. Во-вторых, мы используем функцию `make_scorer` для создания объекта `scorer`, обязательно указав, являются ли более высокие или более низкие оценки желательными (с помощью параметра `greater_is_better`).

Собственный метрический показатель в данном решении (`custom_metric`) является игрушечным примером, поскольку он просто обертывает встроенный метрический показатель, вычисляющий оценку R^2. В реальной ситуации мы заменили бы функцию `custom_metric` на любой собственный метрический показатель, который требуется. Вместе с тем мы видим, что собственный метрический показатель, который вычисляет R^2, действительно работает, сравнивая результаты со встроенным в библиотеку scikit-learn методом `r2_score`:

```
# Предсказать значения
target_predicted = model.predict(features_test)

# Вычислить оценочный показатель r-квадрат
r2_score(target_test, target_predicted)
```

```
0.99979061028820582
```

Дополнительные материалы для чтения

- Документация библиотеки scikit-learn по методу `make_scorer` **(http://bit.ly/2FwMm4m)**.

11.11. Визуализация эффекта размера тренировочного набора

Задача

Требуется оценить эффект влияния количества наблюдений в тренировочном наборе на некоторый метрический показатель (точность, F_1 и т. д.).

Решение

Построить график кривой заучивания[2] (рис. 11.3):

```
# Загрузить библиотеки
import numpy as np
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_digits
from sklearn.model_selection import learning_curve

# Загрузить данные
digits = load_digits()

# Создать матрицу признаков и вектор целей
features, target = digits.data, digits.target

# Создать перекрестно-проверочные тренировочные и тестовые
# оценки для разных размеров тренировочного набора
train_sizes, train_scores, test_scores = learning_curve(
                            RandomForestClassifier(), # Классификатор
                            features,   # Матрица признаков
                            target,     # Вектор целей
                            cv=10,      # Количество блоков
                            scoring='accuracy',  # Показатель
                                                 # результативности
                            n_jobs=-1,  # Использовать все ядра CPU
                                 # Размеры 50 тренировочных наборов
                            train_sizes=np.linspace(0.01, 1.0, 50))
```

[2] Этот термин взят из психологии (его варианты — кривая приобретения навыка, кривая освоения), поскольку он более точно соответствует характеру процесса — самообучению модели на тренировочных данных — чем термин "кривая обучения". — *Прим. перев.*

```
# Создать средние и стандартные отклонения оценок
# тренировочного набора
train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)

# Создать средние и стандартные отклонения оценок
# тестового набора
test_mean = np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)

# Нанести линии
plt.plot(train_sizes, train_mean, '--', color="#111111",
         label="Тренировочная оценка")
plt.plot(train_sizes, test_mean, color="#111111",
         label="Перекрестно-проверочная оценка")

# Нанести полосы
plt.fill_between(train_sizes, train_mean - train_std,
                 train_mean + train_std, color="#DDDDDD")
plt.fill_between(train_sizes, test_mean - test_std,
                 test_mean + test_std, color="#DDDDDD")

# Построить график
plt.title("Кривая заучивания")
plt.xlabel("Размер тренировочного набора")
plt.ylabel("Оценка точности")
plt.legend(loc="best")
plt.tight_layout()
plt.show()
```

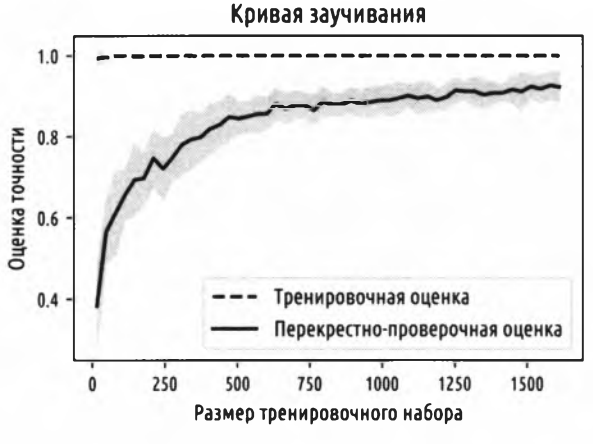


Рис. 11.3

Обсуждение

Кривые заучивания визуализируют результативность (например, точность, полноту) модели на тренировочном наборе и во время перекрестной проверки по мере увеличения количества наблюдений в тренировочном наборе. Они широко используются для определения того, выиграют ли наши обучающиеся алгоритмы от сбора дополнительных тренировочных данных.

В нашем решении мы строим график точности классификатора случайного леса при 50 разных размерах тренировочного набора от 1% наблюдений до 100%. Возрастающая оценка точности перекрестно-проверяемых моделей говорит о том, что мы, скорее всего, выиграем от дополнительных наблюдений (хотя на практике это может быть неосуществимо).

Дополнительные материалы для чтения

- Документация библиотеки scikit-learn по кривой заучивания (**http://bit.ly/2FwjBVe**).

11.12. Создание текстового отчета об оценочных метрических показателях

Задача

Требуется краткое описание результативности классификатора.

Решение

Использовать функцию `classification_report` библиотеки scikit-learn:

```
# Загрузить библиотеки
from sklearn import datasets
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report

# Загрузить данные
iris = datasets.load_iris()

# Создать матрицу признаков
features = iris.data

# Создать вектор целей
target = iris.target

# Создать список имен целевых классов
class_names = iris.target_names
```

```
# Создать тренировочный и тестовый наборы
features_train, features_test, target_train, target_test =
    train_test_split(features, target, random_state=1)

# Создать объект логистической регрессии
classifier = LogisticRegression()

# Натренировать модель и сделать предсказания
model = classifier.fit(features_train, target_train)
target_predicted = model.predict(features_test)

# Создать классификационный отчет
print(classification_report(target_test,
                            target_predicted,
                            target_names=class_names))
```

```
   precision    recall  f1-score   support

     setosa       1.00      1.00      1.00        13
 versicolor       1.00      0.62      0.77        16
  virginica       0.60      1.00      0.75         9

avg / total       0.91      0.84      0.84        38
```

Обсуждение

Функция `classification_report` обеспечивает оперативное средство для того, чтобы увидеть некоторые распространенные оценочные метрические показатели, включая точность, полноту и оценку F_1 (описанные ранее в этой главе). Поддержка относится к количеству наблюдений в каждом классе.

Дополнительные материалы для чтения

- "Прецизионность и полнота", Википедия (**https://bit.ly/2piTCZv**).

11.13. Визуализация эффекта значений гиперпараметра

Задача

Требуется разобраться в том, как результативность модели изменяется по мере изменения значений некоторого гиперпараметра.

Решение

Построить график валидационной кривой (рис. 11.4):

```
# Загрузить библиотеки
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import load_digits
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import validation_curve

# Загрузить данные
digits = load_digits()

# Создать матрицу признаков и вектор целей
features, target = digits.data, digits.target

# Создать диапазон значений для параметра
param_range = np.arange(1, 250, 2)

# Вычислить точность на тренировочном и тестовом наборах,
# используя диапазон значений параметра
train_scores, test_scores = validation_curve(
                RandomForestClassifier(),  # Классификатор
                features,                  # Матрица признаков
                target,                    # Вектор целей
                param_name="n_estimators", # Исследуемый гиперпараметр
                param_range=param_range,   # Диапазон значений
                                           # гиперпараметра
                cv=3,                      # Количество блоков
                scoring="accuracy",        # Показатель результативности
                n_jobs=-1)                 # Использовать все ядра CPU

# Вычислить среднее и стандартное отклонение для оценок
# тренировочного набора
train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)

# Вычислить среднее и стандартное отклонение для оценок
# тестового набора
test_mean = np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)

# Построить график средних оценок точности
# для тренировочного и тестового наборов
plt.plot(param_range, train_mean, color="black",
         label="Тренировочная оценка")
```

```
plt.plot(param_range, test_mean, color="dimgrey",
         label="Перекрестно-проверочная оценка")

# Нанести полосы точности
# для тренировочного и тестового наборов
plt.fill_between(param_range, train_mean - train_std,
                 train_mean + train_std, color="gray")
plt.fill_between(param_range, test_mean - test_std,
                 test_mean + test_std, color="gainsboro")

# Создать график
plt.title("Валидационная кривая со случайным лесом")
plt.xlabel("Количество деревьев")
plt.ylabel("Оценка точности")
plt.tight_layout()
plt.legend(loc="best")
plt.show()
```

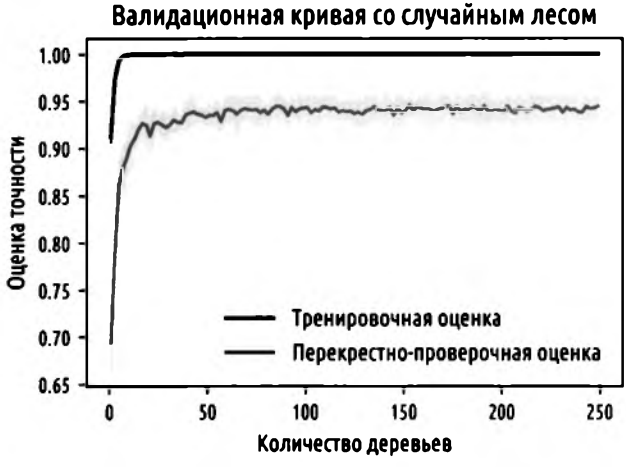


Рис. 11.4

Обсуждение

Большинство тренируемых алгоритмов (включая многие из тех, которые описаны в этой книге) содержат гиперпараметры, которые должны быть выбраны до начала тренировочного процесса. Например, классификатор случайного леса создает "лес" деревьев принятия решений, каждый из которых "голосует" за предсказанный класс наблюдения. Один гиперпараметр в классификаторах случайного леса — это количество деревьев в лесу. Чаще всего значения гиперпараметров выбираются при отборе модели *(см. главу 12)*. Тем не менее иногда полезно представить, как результативность модели изменяется по мере изменения значения гиперпараметра. В нашем решении мы строим график изменения точности классификатора случай-

ного леса для тренировочного набора и во время перекрестной проверки по мере увеличения количества деревьев. Когда имеется небольшое количество деревьев, тренировочная и перекрестно-проверочная оценки — низкие, что свидетельствует о том, что модель недостаточно подогнана. Когда количество деревьев увеличивается до 250, точность обоих выравнивается, свидетельствуя о том, что, вероятно, не будет какой-то особой ценности в вычислительных затратах на тренировку массивного леса.

В библиотеке scikit-learn можно рассчитать валидационную кривую с помощью функции `validation_curve`, которая содержит три важных параметра:

- `param_name` — имя варьируемого гиперпараметра;
- `param_range` — используемое значение гиперпараметра;
- `scoring` — оценочный метрический показатель, используемый для оценивания модели.

Дополнительные материалы для чтения

- Документация библиотеки scikit-learn по валидационной кривой **(http://bit.ly/2FuwYFG)**.

Отбор модели

Введение

В машинном самообучении обучающиеся алгоритмы используются для заучивания параметров модели путем минимизации некоторой функции потери. Вдобавок к этому многие обучающиеся алгоритмы (например, опорно-векторный классификатор и случайные леса) также имеют гиперпараметры, которые должны быть определены вне процесса самообучения. Например, случайные леса являются коллекциями деревьев принятия решений (отсюда и слово *"лес"*). Однако число деревьев принятия решений в лесу не заучивается алгоритмом и должно быть задано априорно до подгонки. Это часто обозначается как *тонкая настройка гиперпараметров*, *гиперпараметрическая оптимизация* либо *отбор моделей*. Кроме того, часто нам может потребоваться попробовать несколько обучающихся алгоритмов (например, попробовать и опорно-векторный классификатор, и случайные леса, чтобы увидеть, какой метод самообучения производит наилучшую модель).

В то время как существует широкое разнообразие в использовании терминологии в этой области, в данной книге мы обозначаем отбор наилучшего обучающегося алгоритма и его наилучших гиперпараметров, как отбор модели. Причина проста: представьте, что имеются данные, и требуется натренировать опорно-векторный классификатор с 10 вариантами значений гиперпараметра и классификатор случайного леса с 10 вариантами значений гиперпараметра. В результате мы пытаемся отобрать наилучшую модель из набора, состоящего из 20 вариантов моделей. В этой главе мы рассмотрим методы эффективного отбора наилучшей модели из набора потенциальных вариантов.

Мы будем ссылаться на конкретные гиперпараметры, такие как C (обратная величина силы регуляризации). Не переживайте, если не знаете, что такое гиперпараметры. Мы рассмотрим их в последующих главах. Вместо этого просто обращайтесь с гиперпараметрами, как настройками обучающегося алгоритма, которые мы должны выбрать перед началом его тренировки.

12.1. Отбор наилучших моделей с помощью исчерпывающего поиска

Задача

Требуется отобрать наилучшую модель, выполнив поиск по диапазону гиперпараметров.

Решение

Использовать класс `GridSearchCV` библиотеки scikit-learn:

```
# Загрузить библиотеки
import numpy as np
from sklearn import linear_model, datasets
from sklearn.model_selection import GridSearchCV

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать объект логистической регрессии
logistic = linear_model.LogisticRegression()

# Создать диапазон вариантов значений штрафного гиперпараметра
penalty = ['l1', 'l2']

# Создать диапазон вариантов значений регуляризационного гиперпараметра
C = np.logspace(0, 4, 10)

# Создать словарь вариантов гиперпараметров
hyperparameters = dict(C=C, penalty=penalty)

# Создать объект решеточного поиска
gridsearch = GridSearchCV(logistic, hyperparameters, cv=5, verbose=0)

# Выполнить подгонку объекта решеточного поиска
best_model = gridsearch.fit(features, target)
```

Обсуждение

Класс `GridSearchCV` реализует отбор модели по методу грубой силы с использованием перекрестной проверки. В частности, пользователь определяет наборы возможных значений для одного или нескольких гиперпараметров, а затем объект класса `GridSearchCV` тренирует модель, используя каждое значение и/или комбинацию значений. Модель с лучшей оценкой результативности отбирается как наилучшая. Например, в нашем решении в качестве обучающегося алгоритма мы использовали логистическую регрессию, содержащую два гиперпараметра: C и регуляризационный штраф. Не переживайте, если не знаете, что такое C и регуляризация; мы рассмотрим их в следующих главах. Просто для начала уясните, что гиперпараметры C и регуляризационный штраф могут принимать диапазон значений, которые должны быть указаны до начала тренировки. Для гиперпараметра C мы определяем 10 возможных значений:

```
np.logspace(0, 4, 10)
```

```
array([1.00000000e+00, 2.78255940e+00, 7.74263683e+00, 2.15443469e+01,
       5.99484250e+01, 1.66810054e+02, 4.64158883e+02, 1.29154967e+03,
       3.59381366e+03, 1.00000000e+04])
```

И точно так же мы определяем два возможных значения регуляризационного штрафа: `['l1', 'l2']`. Для каждой комбинации значений гиперпараметров C и регуляризационного штрафа мы тренируем модель и оцениваем ее с помощью k-блочной перекрестной проверки. В нашем решении мы имели 10 возможных значений C, 2 возможных значения регуляризационного штрафа и 5 блоков. Они создали $10 \times 2 \times 5 = 100$ вариантов моделей, из которых была выбрана лучшая.

После завершения работы объекта `GridSearchCV` можем увидеть гиперпараметры лучшей модели:

```
# Взглянуть на наилучшие гиперпараметры
print('Лучший штраф:', best_model.best_estimator_.get_params()['penalty'])
print('Лучший C:', best_model.best_estimator_.get_params()['C'])
```

```
Лучший штраф: l1
Лучший C: 7.742636826811269
```

По умолчанию после определения наилучших гиперпараметров объект `GridSearchCV` может переобучить модель, используя наилучшие гиперпараметры, на всем наборе данных (вместо откладывания блока для перекрестной проверки). Эту модель можно использовать для предсказания значений, как и любую другую модель библиотеки scikit-learn:

```
# Предсказать вектор целей
best_model.predict(features)
```

```
array([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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

Стоит отметить один параметр `verbose` объекта `GridSearchCV`. В нем нет особой необходимости, однако он может служить обнадеживающей мерой во время длительных процессов поиска, чтобы получать уведомления о том, что поиск прогрессирует. Параметр `verbose` задает объем сообщений, выводимых во время поиска, при этом `0` обозначает отсутствие сообщений, а значения от `1` до `3` — вывод сообщений с большей детализацией.

Дополнительные материалы для чтения

- Документация библиотеки scikit-learn по решеточному поиску и классу `GridSearchCV` (**http://bit.ly/2Fuctc4**).

12.2. Отбор наилучших моделей с помощью рандомизированного поиска

Задача

Для отбора наилучшей модели требуется вычислительно более дешевый метод, чем исчерпывающий поиск.

Решение

Использовать класс `RandomizedSearchCV` библиотеки scikit-learn:

```
# Загрузить библиотеки
from scipy.stats import uniform
from sklearn import linear_model, datasets
from sklearn.model_selection import RandomizedSearchCV

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать объект логистической регрессии
logistic = linear_model.LogisticRegression()

# Создать диапазон вариантов значений штрафного гиперпараметра
penalty = ['l1', 'l2']

# Создать диапазон вариантов значений регуляризационного гиперпараметра
C = uniform(loc=0, scale=4)

# Создать словарь вариантов гиперпараметров
hyperparameters = dict(C=C, penalty=penalty)

# Создать объект рандомизированного поиска
randomizedsearch = RandomizedSearchCV(
    logistic, hyperparameters, random_state=1, n_iter=100,
    cv=5, verbose=0, n_jobs=-1)

# Выполнить подгонку объекта рандомизированного поиска
best_model = randomizedsearch.fit(features, target)
```

Обсуждение

В рецепте 12.1 мы использовали объект класса `GridSearchCV` на заданном пользователем наборе значений гиперпараметров, чтобы отыскать наилучшую модель в соответствии с оценочной функцией. Более эффективный метод, чем решеточный

поиск GridSearchCV методом грубой силы с перебором всех значений, — это поиск на определенном количестве случайных сочетаний гиперпараметрических значений из предоставленных пользователем распределений (например, нормального, равномерного). В библиотеке scikit-learn этот метод рандомизированного поиска реализован с помощью класса RandomizedSearchCV.

В случае с объектом класса RandomizedSearchCV, если мы зададим распределение, то библиотека scikit-learn будет случайным образом отбирать без возврата образцы значений гиперпараметров из этого распределения. В качестве примера общего принципа работы здесь мы случайным образом отберем 10 значений из равномерного распределения в диапазоне от 0 до 4:

```
# Определить равномерное распределение между 0 и 4,
# отобрать 10 значений
uniform(loc=0, scale=4).rvs(10)
```

```
array([0.93050861, 1.80860785, 1.31748985, 3.52128795, 3.639449  ,
       2.42981975, 3.1098117 , 2.56694891, 0.36913549, 3.73275217])
```

В качестве альтернативы, если мы зададим список значений, таких как два значения гиперпараметра регуляризационного штрафа, ['l1', 'l2'], объект класса RandomizedSearchCV будет выполнять случайный отбор с заменой из списка.

Так же как с объектом класса GridSearchCV, мы можем увидеть значения гиперпараметров наилучшей модели:

```
# Взглянуть на лучшие гиперпараметры
print('Лучший штраф:', best_model.best_estimator_.get_params()['penalty'])
print('Лучший C:', best_model.best_estimator_.get_params()['C'])
```

```
Лучший штраф: l1
Лучший C: 1.668088018810296
```

И так же как с GridSearchCV, после завершения поиска RandomizedSearchCV выполняет подгонку новой модели, используя наилучшие гиперпараметры, на всем наборе данных. Эту модель можно использовать, как и любую другую в scikit-learn; например, делать предсказания:

```
# Предсказать вектор целей
best_model.predict(features)
```

```
array([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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2,
       2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

Количество отобранных комбинаций гиперпараметров (т. е. количество вариантов натренированных моделей) задается установкой параметра n_iter (количества итераций).

Дополнительные материалы для чтения

- Документация библиотеки scikit-learn по рандомизированному поиску и классу `RandomizedSearchCV` (**http://bit.ly/2B7p1zT**).
- "Случайный поиск для гиперпараметрической оптимизации", статья журнала Machine Learning Research (**http://bit.ly/2FrUinf**).

12.3. Отбор наилучших моделей из нескольких обучающихся алгоритмов

Задача

Требуется отобрать наилучшую модель путем поиска по диапазону обучающихся алгоритмов и их соответствующих гиперпараметров.

Решение

Создать словарь вариантов обучающихся алгоритмов и их гиперпараметров:

```
# Загрузить библиотеки
import numpy as np
from sklearn import datasets
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline

# Задать начальное число для генератора псевдослучайных чисел
np.random.seed(0)

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать конвейер
pipe = Pipeline([("classifier", RandomForestClassifier())])

# Создать словарь вариантов обучающихся алгоритмов и их гиперпараметров
search_space = [{"classifier": [LogisticRegression()],
                 "classifier__penalty": ['l1', 'l2'],
                 "classifier__C": np.logspace(0, 4, 10)},
                {"classifier": [RandomForestClassifier()],
                 "classifier__n_estimators": [10, 100, 1000],
                 "classifier__max_features": [1, 2, 3]}]
```

```
# Создать объект решеточного поиска
gridsearch = GridSearchCV(pipe, search_space, cv=5, verbose=0)

# Выполнить подгонку объекта решеточного поиска
best_model = gridsearch.fit(features, target)
```

Обсуждение

В двух предыдущих рецептах мы нашли наилучшую модель путем поиска возможных гиперпараметрических значений обучающегося алгоритма. Однако что делать, если мы не уверены, какой обучающийся алгоритм использовать? Последние версии библиотеки scikit-learn позволяют включить в поисковое пространство обучающиеся алгоритмы. В нашем решении мы определяем поисковое пространство, которое включает два обучающихся алгоритма: логистический регрессионный классификатор и классификатор случайного леса. Каждый обучающийся алгоритм имеет свои гиперпараметры, и мы определяем их варианты значений, используя формат `classifier__[имя гиперпараметра]`. Например, для нашей логистической регрессии для определения множества возможных значений регуляризационного гиперпараметрического пространства `C` и потенциальных типов регуляризационных штрафов `penalty` мы создаем словарь:

```
{'classifier': [LogisticRegression()],
 'classifier__penalty': ['l1', 'l2'],
 'classifier__C': np.logspace(0, 4, 10)}
```

Мы также можем создать аналогичный словарь для гиперпараметров случайного леса:

```
{'classifier': [RandomForestClassifier()],
 'classifier__n_estimators': [10, 100, 1000],
 'classifier__max_features': [1, 2, 3]}
```

После завершения поиска можно использовать атрибут `best_estimator_` для просмотра обучающегося алгоритма и гиперпараметров наилучшей модели:

```
# Взглянуть на лучшую модель
best_model.best_estimator_.get_params()["classifier"]

LogisticRegression(C=7.742636826811269, class_weight=None, dual=False,
          fit_intercept=True, intercept_scaling=1, max_iter=100,
          multi_class='ovr', n_jobs=1, penalty='l1', random_state=None,
          solver='liblinear', tol=0.0001, verbose=0, warm_start=False)
```

Так же как и с последними двумя рецептами, после подгонки объекта поиска для отбора модели эту наилучшую модель можно использовать, как и любую другую модель scikit-learn:

```
# Предсказать вектор целей
best_model.predict(features)
```

```
array([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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

12.4. Отбор наилучших моделей во время предобработки

Задача

Требуется включить шаг предобработки во время отбора модели.

Решение

Создать конвейер, включающий шаг предобработки и любой из его параметров:

```
# Загрузить библиотеки
import numpy as np
from sklearn import datasets
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline, FeatureUnion
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# Задать начальное число для генератора псевдослучайных чисел
np.random.seed(0)

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать объект предобработки, который включает
# признаки стандартного шкалировщика StandardScaler и объект PCA
preprocess = FeatureUnion([("std", StandardScaler()), ("pca", PCA())])

# Создать конвейер
pipe = Pipeline([("preprocess", preprocess),
("classifier", LogisticRegression())])

# Создать пространство вариантов значений
search_space = [{"preprocess__pca__n_components": [1, 2, 3],
                 "classifier__penalty": ["l1", "l2"],
                 "classifier__C": np.logspace(0, 4, 10)}]
```

```
# Создать объект решеточного поиска
clf = GridSearchCV(pipe, search_space, cv=5, verbose=0, n_jobs=-1)

# Выполнить подгонку объекта решеточного поиска
best_model = clf.fit(features, target)
```

Обсуждение

Очень часто сначала требуется предварительно обработать данные и только потом использовать их для тренировки модели. При этом выполнять предобработку во время отбора модели следует осмотрительно. Во-первых, класс `GridSearchCV` использует перекрестную проверку для определения модели с наибольшей результативностью. Однако в перекрестной проверке мы фактически претворяемся, что блок отложен в сторону, поскольку тестовый набор не виден и, следовательно, не является частью подгонки любых шагов предобработки (например, масштабирования или стандартизации). По этой причине мы не можем предварительно обработать данные и выполнить решеточный поиск `GridSearchCV`. Вместо этого шаги предобработки должны быть частью набора действий, выполняемых объектом `GridSearchCV`. Хотя это может показаться сложным, но в реальности библиотека scikit-learn нашу задачу упрощает. Класс `FeatureUnion` позволяет правильно объединить несколько действий предобработки. В нашем решении мы используем объект класса `FeatureUnion` для объединения двух этапов предварительной обработки: стандартизации значений признаков (`StandardScaler`) и анализ главных компонент (`PCA`). Этот объект называется `preprocess` и содержит оба шага предобработки. Затем мы включаем `preprocess` в конвейер с нашим обучающимся алгоритмом. В конечном счете это позволяет нам передать на аутсорсинг в библиотеку scikit-learn надлежащую (и запутанную) работу по подгонке, преобразованию и тренировке моделей с комбинациями гиперпараметров.

Во-вторых, некоторые методы предобработки имеют свои параметры, которые часто должны быть предоставлены пользователем. Например, уменьшение размерности с помощью метода PCA требует от пользователя определения количества главных компонент, используемых для создания преобразованного набора признаков. В идеале мы бы выбрали количество компонент, которое производит модель с наибольшей результативностью для некоторого оценочного тестового показателя. К счастью, библиотека scikit-learn позволяет это легко делать. Когда мы включаем значения вариантов компонент в поисковое пространство, при поиске они обрабатываются как любой другой гиперпараметр. В нашем решении мы определили `features__pca__n_components': [1, 2, 3]` в поисковом пространстве, тем самым говоря, что мы хотели бы обнаружить, производят наилучшую модель один, два или три главных компоненты.

После завершения отбора модели можно просмотреть значения предобработки, которые произвели наилучшую модель. Например, мы можем увидеть наилучшее количество главных компонент:

```
# Взглянуть на самую лучшую модель
best_model.best_estimator_.get_params()['preprocess__pca__n_components']
```

```
1
```

12.5. Ускорение отбора модели с помощью распараллеливания

Задача

Требуется ускорить отбор модели.

Решение

Использовать на компьютере все ядра центрального процессора, установив `n_jobs=-1`:

```
# Загрузить библиотеки
import numpy as np
from sklearn import linear_model, datasets
from sklearn.model_selection import GridSearchCV

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать объект логистической регрессии
logistic = linear_model.LogisticRegression()

# Создать диапазон вариантов значений штрафного гиперпараметра
penalty = ["l1", "l2"]

# Создать диапазон вариантов значений регуляризационного гиперпараметра
C = np.logspace(0, 4, 1000)

# Создать словарь вариантов гиперпараметров
hyperparameters = dict(C=C, penalty=penalty)

# Создать объект решеточного поиска
gridsearch = GridSearchCV(logistic, hyperparameters, cv=5,
                          n_jobs=-1, verbose=1)

# Выполнить подгонку объекта решеточного поиска
best_model = gridsearch.fit(features, target)
```

```
[Parallel(n_jobs=-1)]: Done 255 tasks          | elapsed:  3.7s
[Parallel(n_jobs=-1)]: Done 3855 tasks         | elapsed:  22.2s
[Parallel(n_jobs=-1)]: Done 9855 tasks         | elapsed:  1.1min
[Parallel(n_jobs=-1)]: Done 10000 out of 10000 | elapsed:  1.1min finished
```

Обсуждение

В рецептах этой главы мы держали количество вариантов моделей небольшим, чтобы выполнение программного кода завершалось быстро. В реальном мире у нас часто будет много тысяч или десятков тысяч тренируемых моделей. В конечном счете на то, чтобы найти наилучшую модель, может уйти много часов. Для ускорения этого процесса библиотека scikit-learn позволяет тренировать несколько моделей одновременно. Не вдаваясь в слишком много технических деталей, можно сказать, что библиотека scikit-learn способна одновременно тренировать модели, используя ядра процессора. Большинство современных ноутбуков имеют четыре ядра, поэтому (если вы в настоящее время работаете на ноутбуке) мы потенциально можем тренировать четыре модели одновременно. Это значительно увеличит скорость процесса отбора модели. Параметр `n_jobs` задает количество моделей для параллельной тренировки.

В нашем решении мы передали `n_jobs` значение `-1`, что сообщает библиотеке scikit-learn использовать все ядра. Однако параметр `n_jobs` по умолчанию имеет значение `1`, т. е. использует только одно ядро. Для того чтобы продемонстрировать это, если мы выполним тот же решеточный поиск `GridSearch`, что и в решении, но с параметром `n_jobs=1`, то увидим, что для поиска наилучшей модели требуется значительно больше времени (обратите внимание, что точное время будет зависеть от вашего компьютера):

```
# Создать объект решеточного поиска с использованием одного ядра
clf = GridSearchCV(logistic, hyperparameters, cv=5, n_jobs=1, verbose=1)

# Выполнить подгонку объекта решеточного поиска
best_model = clf.fit(features, target)

Fitting 5 folds for each of 2000 candidates, totalling 10000 fits
[Parallel(n_jobs=1)]: Done 10000 out of 10000 | elapsed:  2.6min finished
```

12.6. Ускорение отбора модели с помощью алгоритмически специализированных методов

Задача

Требуется ускорить отбор модели.

Решение

Если используется отборное число обучающихся алгоритмов, применить модельно специфическую перекрестно-проверочную настройку гиперпараметров, имеющуюся в библиотеке scikit-learn, например реализованную в классе `LogisticRegressionCV`:

```
# Загрузить библиотеку
from sklearn import linear_model, datasets

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать объект перекрестно-проверяемой логистической регрессии
logit = linear_model.LogisticRegressionCV(Cs=100)

# Натренировать модель
logit.fit(features, target)
```

```
LogisticRegressionCV(Cs=100, class_weight=None, cv=None, dual=False,
    fit_intercept=True, intercept_scaling=1.0, max_iter=100,
    multi_class='ovr', n_jobs=1, penalty='l2', random_state=None,
    refit=True, scoring=None, solver='lbfgs', tol=0.0001, verbose=0)
```

Обсуждение

Иногда характеристики обучающегося алгоритма позволяют нам отыскать наилучшие гиперпараметры значительно быстрее, чем поиск методом грубой силы либо рандомизированный поиск модели. В библиотеке scikit-learn многие обучающиеся алгоритмы (например, гребневая регрессия, лассо-регрессия и комбинированная регрессия elastic net с линейной комбинацией L^1- и L^2-штрафов) имеют алгоритмически специфичный метод перекрестной проверки, которым следует воспользоваться. Например, класс `LogisticRegression` используется для выполнения стандартного логистического регрессионного классификатора, в то время как класс `LogisticRegressionCV` реализует эффективный перекрестно-проверяемый логистический регрессионный классификатор, который способен идентифицировать оптимальную величину гиперпараметра C.

Класс `LogisticRegressionCV` библиотеки scikit-learn реализует метод с включением параметра `Cs`. Если передается список, то параметр `Cs` содержит варианты гиперпараметрических значений на выбор. Если задано целое число, то параметр `Cs` создает список из этого количества возможных значений. Варианты значений выводятся логарифмически из диапазона от 0.0001 до 1.0000 (диапазона разумных значений для C).

Однако основной недостаток класса `LogisticRegressionCV` заключается в том, что он может выполнять поиск только диапазона значений для C. В рецепте 12.1 наше возможное гиперпараметрическое пространство включало как C, так и еще один гиперпараметр (регуляризационную штрафную норму). Это ограничение характерно для многих модельно специфических перекрестно-проверяемых подходов, принятых в библиотеке scikit-learn.

Дополнительные материалы для чтения

- Документация библиотеки scikit-learn по логистической регрессии и классу `LogisticRegressionCV` (**http://bit.ly/2GPJvjY**).
- Документация библиотеки scikit-learn по модельно специфической перекрестной проверке (**http://bit.ly/2F0TQsL**).

12.7. Оценивание результативности после отбора модели

Задача

Требуется оценить результативность модели, найденной с помощью отбора модели.

Решение

Использовать вложенную перекрестную проверку, чтобы избежать смещенной оценки:

```
# Загрузить библиотеки
import numpy as np
from sklearn import linear_model, datasets
from sklearn.model_selection import GridSearchCV, cross_val_score

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать объект логистической регрессии
logistic = linear_model.LogisticRegression()

# Создать диапазон из 20 вариантов значений для C
C = np.logspace(0, 4, 20)

# Создать словарь вариантов гиперпараметров
hyperparameters = dict(C=C)

# Создать объект решеточного поиска
gridsearch = GridSearchCV(logistic, hyperparameters, cv=5,
                          n_jobs=-1, verbose=0)

# Выполнить вложенную перекрестную проверку и выдать среднюю оценку
cross_val_score(gridsearch, features, target).mean()

0.95343137254901966
```

Обсуждение

Идея вложенной перекрестной проверки во время отбора модели для многих людей на первых порах воспринимается с трудом. Напомним, что в *k*-блочной перекрестной проверке мы тренируем нашу модель на $k-1$ блоках данных, используем эту модель для предсказания на оставшемся блоке, а затем оцениваем нашу модель по тому, насколько хорошо предсказания нашей модели соответствуют истинным значениям. Затем мы повторяем этот процесс *k* раз.

В тех видах поиска для отбора модели, описанных в этой главе (например, реализованных в классах `GridSearchCV` и `RandomizedSearchCV`), мы применяли перекрестную проверку для оценки того, какие значения гиперпараметра порождают наилучшие модели. Вместе с тем возникает тонкая и в целом недооцененная проблема: поскольку мы использовали данные для отбора наилучших значений гиперпараметра, мы не можем использовать те же данные для оценки результативности модели. Каково же решение? Обернуть перекрестную проверку, используемую для поиска модели, еще в одну перекрестную проверку! Во вложенной перекрестной проверке "внутренняя" перекрестная проверка отбирает наилучшую модель, в то время как "внешняя" перекрестная проверка предоставляет нам объективную оценку результативности модели. В нашем решении внутренняя перекрестная проверка — это объект `GridSearchCV`, который мы затем переносим во внешнюю перекрестную проверку с помощью метода `cross_val_score`.

Если вы запутались, попробуйте провести простой эксперимент. Во-первых, установите `verbose=1`, чтобы можно было видеть, что происходит:

```
gridsearch = GridSearchCV(logistic, hyperparameters, cv=5, verbose=1)
```

Затем выполните подгонку объекта решеточного поиска `gridsearch.fit(features, target)`, который является нашей внутренней перекрестной проверкой, используемой для поиска наилучшей модели:

```
best_model = gridsearch.fit(features, target)

Fitting 5 folds for each of 20 candidates, totalling 100 fits

[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:    0.3s finished
```

Из результатов вы можете увидеть, что внутренняя перекрестная проверка натренировала 20 вариантов моделей пять раз, в общей сложности 100 моделей. Затем вложите `clf` внутрь новой перекрестной проверки, которая по умолчанию имеет три блока:

```
scores = cross_val_score(gridsearch, features, target)

Fitting 5 folds for each of 20 candidates, totalling 100 fits

[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:    0.3s finished

Fitting 5 folds for each of 20 candidates, totalling 100 fits
```

```
[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:    0.2s finished

Fitting 5 folds for each of 20 candidates, totalling 100 fits

[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:    0.3s finished
```

Результат показывает, что внутренняя перекрестная проверка натренировала 20 моделей пять раз и нашла наилучшую модель, а эта модель была оценена с использованием внешней трехблочной перекрестной проверки, создав в общей сложности 300 натренированных моделей.

Линейная регрессия

Введение

Линейная регрессия является одним из самых простых обучающихся алгоритмов в нашем инструментарии. Если вы когда-либо проходили вводный курс статистики в колледже, вероятно, последней темой, которую вы рассматривали, была линейная регрессия. На самом деле, она настолько проста, что иногда и вовсе не считается за машинное самообучение! Что бы вы ни думали, факт остается фактом: линейная регрессия — и ее расширения — продолжает оставаться распространенным и полезным методом предсказания, когда вектор целей является количественным значением (например, цена дома, возраст). В этой главе мы рассмотрим различные методы линейной регрессии (и некоторые расширения) для создания хорошо работающих предсказательных моделей.

13.1. Подгонка прямой

Задача

Требуется натренировать модель, представляющую линейную связь между признаком и вектором целей.

Решение

Использовать линейную регрессию (в библиотеке scikit-learn это класс `LinearRegression`):

```
# Загрузить библиотеки
from sklearn.linear_model import LinearRegression
from sklearn.datasets import load_boston

# Загрузить данные только с двумя признаками
boston = load_boston()
features = boston.data[:,0:2]
target = boston.target

# Создать объект линейной регрессии
regression = LinearRegression()
```

```
# Выполнить подгонку линейной регрессии
model = regression.fit(features, target)
```

Обсуждение

Линейная регрессия исходит из того, что связь между признаками и вектором целей является приблизительно линейной, т. е. *эффект* (также называемый *коэффициентом*, *весом* или *параметром*) признаков на вектор целей является постоянным. В нашем решении для целей объяснения мы натренировали нашу модель, используя всего два признака. Это значит, что наша линейная модель будет:

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 + \varepsilon,$$

где $\hat{y}$ — наша цель; x_i — данные для одного признака; $\hat{\beta}_1$ и $\hat{\beta}_2$ — коэффициенты, идентифицированные путем подгонки модели; ε — ошибка.

После выполнения подгонки нашей модели мы можем взглянуть на значение каждого параметра. Например, $\hat{\beta}_0$, так называемое смещение или пересечение, можно просмотреть с помощью атрибута `intercept_`:

```
# Взглянуть на точку пересечения
model.intercept_
```

```
22.46681692105723
```

И $\hat{\beta}_1$, и $\hat{\beta}_2$ можно показать с помощью `coef_`:

```
# Взглянуть на коэффициенты признаков
model.coef_
```

```
array([-0.34977589, 0.11642402])
```

В нашем наборе данных целевым значением является медианное значение жилья в Бостоне (в 1970-х годах) в тысячах долларов. Поэтому цена первого дома в наборе данных:

```
# Первое значение в векторе целей, умноженное на 1000
target[0]*1000
```

```
24000.0
```

Используя метод `predict`, мы можем предсказать значение для этого дома:

```
# Предсказать целевое значение первого наблюдения, умноженное на 1000
model.predict(features)[0]*1000
```

```
24560.23872370844
```

Не плохо! Наша модель сместилась всего на $560.24!

Основным преимуществом линейной регрессии является ее интерпретируемость в значительной степени потому, что модельные коэффициенты представляют собой

эффект единичного изменения на вектор целей. Например, первый признак нашего решения — это количество преступлений на одного жителя. Коэффициент этого признака нашей модели составил -0.35. Это значит, что если мы умножим этот коэффициент на 1000 (т. к. вектором целей является цена дома в тысячах долларов), то у нас будет изменение в цене дома для каждого дополнительного преступления на душу населения:

```
# Первый коэффициент, умноженный на 1000
model.coef_[0]*1000
```

```
-349.77588707748947
```

Это говорит о том, что каждое преступление на душу населения снизит цену дома примерно на $350!

13.2. Обработка интерактивных эффектов

Задача

Имеется признак, влияние которого на целевую переменную зависит от другого признака.

Решение

Создать член взаимодействия для захвата этой зависимости с помощью объекта класса `PolynomialFeatures` библиотеки scikit-learn:

```
# Загрузить библиотеки
from sklearn.linear_model import LinearRegression
from sklearn.datasets import load_boston
from sklearn.preprocessing import PolynomialFeatures

# Загрузить данные только с двумя признаками
boston = load_boston()
features = boston.data[:,0:2]
target = boston.target

# Создать член, характеризующий взаимодействие между признаками
interaction = PolynomialFeatures(
    degree=3, include_bias=False, interaction_only=True)
features_interaction = interaction.fit_transform(features)

# Создать объект линейной регрессии
regression = LinearRegression()

# Выполнить подгонку линейной регрессии
model = regression.fit(features_interaction, target)
```

Обсуждение

Иногда влияние признака на целевую переменную по крайней мере частично зависит от другого признака. Например, представьте себе простой пример с кофе, где у нас два бинарных признака — наличие сахара (`sugar`) и было ли выполнено перемешивание (`stirred`) — и требуется предсказать, сладкий ли кофе на вкус. Просто положив сахар в кофе (`sugar=1, stirred=0`), мы не сделаем вкус кофе сладким (весь сахар на дне!), а одно перемешивание кофе без добавления сахара (`sugar=0, stirred=1`) не сделает его сладким. Как раз наоборот, именно взаимодействие брошенного в кофе сахара и перемешивания кофе (`sugar=1, stirred=1`) сделает вкус кофе сладким. Влияния сахара и перемешивания на сладость зависят друг от друга. В этом случае мы говорим, что существует эффект взаимодействия между признаками `sugar` и `stirred`.

Мы можем учесть эффекты взаимодействия, включив новый признак, состоящий из произведения соответствующих значений из взаимодействующих признаков:

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 + \hat{\beta}_3 x_1 x_2 + \varepsilon,$$

где x_1 и x_2 — значения соответственно `sugar` и `stirred`; $x_1 x_2$ — взаимодействие между ними.

В нашем решении мы использовали набор данных, содержащий только два признака. Вот значения первого наблюдения для каждого из этих признаков:

```
# Взглянуть на признаки для первого наблюдения
features[0]
```

```
array([ 6.32000000e-03, 1.80000000e+01])
```

Для того чтобы создать член, характеризующий взаимодействие между признаками, мы просто умножаем эти два значения между собой для каждого наблюдения:

```
# Импортировать библиотеку
import numpy as np

# Для каждого наблюдения перемножить значения
# первого и второго признаков
interaction_term = np.multiply(features[:, 0], features[:, 1])
```

Затем мы можем взглянуть на член взаимодействия для первого наблюдения:

```
# Взглянуть на член взаимодействия для первого наблюдения
interaction_term[0]
```

```
0.11376
```

Хотя часто будет возникать веская причина полагать, что между двумя признаками существует взаимодействие, всё же иногда ее у нас не будет. В этих случаях может быть полезно использовать класс `PolynomialFeatures` библиотеки scikit-learn для создания членов взаимодействия для всех сочетаний признаков. Затем мы можем

применить стратегии отбора модели, чтобы определить сочетание признаков и членов взаимодействия, которые производят наилучшую модель.

Для создания членов взаимодействия с использованием объекта `PolynomialFeatures` необходимо задать три важных параметра. Самый главный `interaction_only=True` сообщает объекту `PolynomialFeatures` возвращать только члены взаимодействия (а не полиномиальные признаки, которые мы обсудим в рецепте 13.3). По умолчанию класс `PolynomialFeatures` добавляет признак, содержащий те, которые называются смещением. Мы можем предотвратить это с помощью `include_bias=False`. Наконец, параметр `degree` определяет максимальное количество признаков для создания членов взаимодействия (в случае, если мы хотим создать член взаимодействия, который является сочетанием трех признаков). Результат работы объекта класса `PolynomialFeatures` можно увидеть из нашего решения, проверив, соответствуют ли значения признаков первого наблюдения значению члена взаимодействия нашей вручную рассчитанной версии:

```
# Взглянуть на значения для первого наблюдения
features_interaction[0]
```

```
array([ 6.32000000e-03, 1.80000000e+01, 1.13760000e-01])
```

13.3. Подгонка нелинейной связи

Задача

Требуется смоделировать нелинейную связь.

Решение

Создать полиномиальную регрессию путем включения полиномиальных признаков в линейную регрессионную модель:

```
# Загрузить библиотеки
from sklearn.linear_model import LinearRegression
from sklearn.datasets import load_boston
from sklearn.preprocessing import PolynomialFeatures

# Загрузить данные с одним признаком
boston = load_boston()
features = boston.data[:,0:1]
target = boston.target

# Создать полиномиальные признаки x^2 и x^3
polynomial = PolynomialFeatures(degree=3, include_bias=False)
features_polynomial = polynomial.fit_transform(features)

# Создать объект линейной регрессии
regression = LinearRegression()
```

```
# Выполнить подгонку линейной регрессии
model = regression.fit(features_polynomial, target)
```

Обсуждение

До сих пор мы обсуждали моделирование только линейных связей. Примером линейной связи может служить количество этажей здания и его высота. В линейной регрессии мы исходим из того, что эффект количества этажей и высоты здания примерно постоянен, а это означает, что 20-этажное здание будет примерно в два раза выше, чем 10-этажное, которое будет примерно в два раза выше, чем 5-этажное здание. Однако многие представляющие интерес связи не являются строго линейными.

Нередко требуется смоделировать нелинейную связь — например, связь между количеством часов, которые учащийся тратит на учебу, и оценкой, которую он получает за контрольную работу. Интуитивно мы можем себе представить, что в оценках между учащимися, которые учились в течение одного часа, и учащимися, которые вообще не учились, есть большая разница. Вместе с тем существует гораздо меньшая разница в оценках между учащимся, который учился в течение 99 часов, и учащимся, который учился в течение 100 часов. Влияние одного часа учебы на итоговую оценку учащегося уменьшается по мере увеличения количества часов.

Полиномиальная регрессия — это расширение линейной регрессии, позволяющее моделировать нелинейные связи. Для того чтобы создать полиномиальную регрессию, следует конвертировать линейную функцию, которую мы использовали в рецепте 13.1:

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \varepsilon$$

в полиномиальную функцию путем добавления полиномиальных признаков:

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_1^2 + ... + \hat{\beta}_d x_i^d + \varepsilon,$$

где d — степень полинома. Как использовать линейную регрессию для нелинейного признака? Ответ заключается в том, что мы ничего не изменяем в том, как линейная регрессия подбирает модель, а только добавляем полиномиальные признаки. То есть, линейная регрессия не "знает", что x^2 является квадратичным преобразованием x. Она просто рассматривает его, как еще одну переменную.

Более практичное описание может лежать в порядке. Для того чтобы смоделировать нелинейные связи, мы можем создать новые признаки, которые возводят существующий признак, x, в некоторую степень: x^2, x^3 и т. д. Чем больше этих новых признаков мы добавим, тем более гибкой будет "линия", созданная нашей моделью. Для того чтобы более четко отразить суть, представьте, что мы хотим создать полином третьей степени. Для простоты мы сосредоточимся только на одном наблюдении (первом наблюдении в наборе данных) x_0:

```
# Взглянуть на первое наблюдение
features[0]
```

```
array([ 0.00632])
```

Для того чтобы создать полиномиальный признак, мы возведем первое значение наблюдения во вторую степень — x_1^2:

```
# Взглянуть на первое наблюдение, возведенное во вторую степень, x^2
features[0]**2
```

```
array([ 3.99424000e-05])
```

Это будет нашим новым признаком. Затем мы также возведем значение первого наблюдения в третью степень — x_1^3:

```
# Взглянуть на первое наблюдение, возведенное в третью степень, x^3
features[0]**3
```

```
array([ 2.52435968e-07])
```

Включив все три признака (x, x^2 и x^3) в матрицу признаков, а затем выполнив линейную регрессию, мы провели полиномиальную регрессию:

```
# Взглянуть на значения первого наблюдения для x, x^2 и x^3
features_polynomial[0]
```

```
array([ 6.32000000e-03, 3.99424000e-05, 2.52435968e-07])
```

Полиномиальные признаки имеют два важных параметра. Во-первых, `degree` определяет максимальное число степеней для полиномиальных признаков. Например, `degree=3` будет генерировать x^2 и x^3. Наконец, по умолчанию объект `PolynomialFeatures` включает в себя признаки, содержащие одни единицы (называемые смещением). Мы можем удалить их, установив `include_bias=False`.

13.4. Снижение дисперсии с помощью регуляризации

Задача

Требуется уменьшить дисперсию линейной регрессионной модели.

Решение

Использовать обучающийся алгоритм, который включает сжимающий штраф (так называемую регуляризацию), такой как гребневая регрессия и лассо-регрессия:

```
# Загрузить библиотеки
from sklearn.linear_model import Ridge
from sklearn.datasets import load_boston
from sklearn.preprocessing import StandardScaler
```

```
# Загрузить данные
boston = load_boston()
features = boston.data
target = boston.target

# Стандартизировать признаки
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# Создать объект гребневой регрессии со значением альфа
regression = Ridge(alpha=0.5)

# Выполнить подгонку линейной регрессии
model = regression.fit(features_standardized, target)
```

Обсуждение

В стандартной линейной регрессии модель тренируется минимизировать сумму квадратической ошибки (погрешности) между истинными y_i и предсказываемыми $\hat{y}_i$ целевыми значениями, или остаточную сумму квадратов (residual sum of squares, RSS):

$$\text{RSS} = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2.$$

Регуляризованные регрессионные ученики похожи, за исключением того, что они пытаются минимизировать RSS *и* некий штраф за общий размер значений коэффициентов, называемый сжимающим штрафом, потому что он пытается "сжать" модель. Существуют два распространенных типа регуляризованных учеников для линейной регрессии: гребневая регрессия и лассо. Единственным формальным различием между ними является тип применяемого сжимающего штрафа. В гребневой регрессии сжимающий штраф — это настроечный гиперпараметр, умноженный на квадрат суммы всех коэффициентов:

$$\text{RSS} + \alpha\sum_{j=1}^{p}\beta_j^2,$$

где $\hat{\beta}_j$ — коэффициент j-го из p признаков; α — гиперпараметр (обсуждается далее). Лассо-регрессия очень похожа на гребневую регрессию, за исключением того, что сжимающий штраф — это настроечный параметр, умножаемый на сумму абсолютных значений всех коэффициентов:

$$\frac{1}{2n}\text{RSS} + \alpha\sum_{j=1}^{p}\left|\hat{\beta}_j\right|,$$

где n — количество наблюдений. Так какой же из них следует использовать? В качестве очень общего эмпирического правила нужно учитывать, что гребневая рег-

рессия часто дает несколько лучшие предсказания, чем лассо, но лассо (по причинам, которые мы обсудим в рецепте 13.5) производит более интерпретируемые модели. Если мы хотим сбалансировать штрафные функции между гребнем и лассо, мы можем использовать эластичную сеть, представляющую собой регрессионную модель, в которую включены оба штрафа. Независимо от того, какой из них мы используем, гребневая регрессия и лассо-регрессия могут штрафовать большие или сложные модели, включая значения коэффициентов в функцию потери, которую мы пытаемся минимизировать.

Гиперпараметр α позволяет нам контролировать то, насколько мы штрафуем коэффициенты, где более высокие значения α создают более простые модели. Идеальное значение α должно быть настроено, как и любой другой гиперпараметр. В библиотеке scikit-learn α устанавливается с помощью параметра `alpha`.

Библиотека scikit-learn включает класс `RidgeCV`, реализующий метод, который позволяет отбирать идеальное значение для α:

```
# Загрузить библиотеку
from sklearn.linear_model import RidgeCV

# Создать объект гребневой регрессии с тремя значениями alpha
regr_cv = RidgeCV(alphas=[0.1, 1.0, 10.0])

# Выполнить подгонку линейной регрессии
model_cv = regr_cv.fit(features_standardized, target)

# Взглянуть на коэффициенты
model_cv.coef_

array([-0.91215884,  1.0658758 ,  0.11942614,  0.68558782, -2.03231631,
        2.67922108,  0.01477326, -3.0777265 ,  2.58814315, -2.00973173,
       -2.05390717,  0.85614763, -3.73565106])
```

Затем мы можем легко просмотреть значение α наилучшей модели:

```
# Взглянуть на alpha
model_cv.alpha_

1.0
```

И последнее замечание: поскольку в линейной регрессии значение коэффициентов частично определяется шкалой признака, а в регуляризованных моделях все коэффициенты суммируются вместе, перед тренировкой мы должны убедиться, что стандартизировали признак.

13.5. Уменьшение количества признаков с помощью лассо-регрессии

Задача

Требуется упростить линейную регрессионную модель, уменьшив количество признаков.

Решение

Использовать лассо-регрессию:

```
# Загрузить библиотеки
from sklearn.linear_model import Lasso
from sklearn.datasets import load_boston
from sklearn.preprocessing import StandardScaler

# Загрузить данные
boston = load_boston()
features = boston.data
target = boston.target

# Стандартизировать признаки
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# Создать объект лассо-регрессии со значением alpha
regression = Lasso(alpha=0.5)

# Выполнить подгонку линейной регрессии
model = regression.fit(features_standardized, target)
```

Обсуждение

Одна из интересных характеристик штрафа лассо-регрессии состоит в том, что он может сжимать коэффициенты модели до нуля, эффективно уменьшая количество признаков в модели. Например, в нашем решении мы установили `alpha` на уровне `0.5` и видим, что многие коэффициенты равны 0. Иными словами, их соответствующие признаки в модели не используются:

```
# Взглянуть на коэффициенты
model.coef_

array([-0.10697735,  0.        , -0.        ,  0.39739898, -0.        ,
        2.97332316, -0.        , -0.16937793, -0.        , -0.        ,
       -1.59957374,  0.54571511, -3.66888402])
```

Однако, если мы увеличим α до гораздо более высокого значения, мы увидим, что буквально ни один из признаков не используется:

```
# Создать лассо-регрессию с высоким alpha
regression_a10 = Lasso(alpha=10)
model_a10 = regression_a10.fit(features_standardized, target)
model_a10.coef_

array([-0.,  0., -0.,  0., -0.,  0., -0.,  0., -0., -0., -0.,  0., -0.])
```

Практическая выгода от этого эффекта в следующем: он означает, что мы можем включить в нашу матрицу признаков 100 признаков, а затем путем настройки гиперпараметра α лассо-регрессии создать модель, которая использует только (например) 10 наиболее важных признаков. Это позволяет нам уменьшить дисперсию, улучшая интерпретируемость нашей модели (поскольку меньшее количество признаков легче объяснить).

Деревья и леса

Введение

Древесные обучающиеся алгоритмы являются широким и популярным семейством родственных непараметрических, контролируемых (с учителем) методов как для классификации, так и для регрессии. Основой для древесных учеников является дерево принятия решений, в котором серия правил принятия решений (например, "если пол мужской, то...") связаны в цепочку. Результат отдаленно напоминает перевернутое дерево, в котором первое правило принятия решения находится наверху, а последующие правила принятия решений расположены ниже. В дереве принятия решений каждое правило принятия решения выполняется в решающем узле, при этом данное правило создает ветви, ведущие к новым узлам. Ветвь без правила принятия решения в конце называется *листом*.

Одной из причин популярности древесных моделей является их интерпретируемость. Для создания интуитивно понятной модели деревья принятия решений можно буквально нарисовать в их полной форме (см. рецепт 14.3). От этой базовой древесной системы отходит целый ряд расширений, начиная со случайных лесов и заканчивая укладкой в ярусы. В этой главе мы рассмотрим способы тренировки, обработки, корректировки, визуализации и оценивания ряда древесных моделей.

14.1. Тренировка классификационного дерева принятия решений

Задача

Требуется натренировать классификатор, используя дерево принятия решений.

Решение

Использовать класс `DecisionTreeClassifier` библиотеки scikit-learn:

```
# Загрузить библиотеки
from sklearn.tree import DecisionTreeClassifier
from sklearn import datasets

# Загрузить данные
iris = datasets.load_iris()
```

```
features = iris.data
target = iris.target

# Создать объект-классификатор дерева принятия решений
decisiontree = DecisionTreeClassifier(random_state=0)

# Натренировать модель
model = decisiontree.fit(features, target)
```

Обсуждение

"Древесные" ученики на основе дерева принятия решений пытаются найти правило принятия решения, которое приводит к наибольшему снижению разнородности в узле. Хотя существует ряд мер разнородности, по умолчанию в классе `DecisionTreeClassifier` используется *коэффициент разнородности Джини*:

$$G(t) = 1 - \sum_{i=1}^{c} p_i^2,$$

где $G(t)$ — коэффициент разнородности Джини в узле t; p_i — доля наблюдений класса c в узле t. Данный процесс нахождения правил принятия решений, которые создают расщепления для уменьшения разнородности, повторяется рекурсивно до тех пор, пока все конечные узлы не станут однородными (т. е. пока не будут содержать только один класс) или пока не будет достигнута некоторая произвольная точка отсечения.

В библиотеке scikit-learn объект класса `DecisionTreeClassifier` работает, как и другие методы самообучения; после того как модель натренирована с использованием метода `fit`, полученную модель можно применять для предсказания класса наблюдения:

```
# Сконструировать новое наблюдение
observation = [[ 5, 4, 3, 2]]

# Предсказать класс наблюдения
model.predict(observation)
```

```
array([1])
```

Мы также можем увидеть предсказанные вероятности классов наблюдения:

```
# Взглянуть на предсказанные вероятности трех классов
model.predict_proba(observation)
```

```
array([[ 0., 1., 0.]])
```

Наконец, если требуется использовать другую меру разнородности, мы можем применить параметр `criterion`:

```
# Создать объект-классификатор дерева принятия решений,
# используя энтропию
decisiontree_entropy = DecisionTreeClassifier(
    criterion='entropy', random_state=0)

# Натренировать модель
model_entropy = decisiontree_entropy.fit(features, target)
```

Дополнительные материалы для чтения

- "Самообучение на основе дерева принятия решений", глава 3 учебника по курсу "Взаимодействие с данными" принстонского университета (**http://bit.ly/2FqJxlj**).

14.2. Тренировка регрессионного дерева принятия решений

Задача

Требуется натренировать регрессионную модель с помощью дерева принятия решений.

Решение

Использовать класс `DecisionTreeRegressor` библиотеки scikit-learn:

```
# Загрузить библиотеки
from sklearn.tree import DecisionTreeRegressor
from sklearn import datasets

# Загрузить данные всего с двумя признаками
boston = datasets.load_boston()
features = boston.data[:,0:2]
target = boston.target

# Создать объект-классификатор дерева принятия решений
decisiontree = DecisionTreeRegressor(random_state=0)

# Натренировать модель
model = decisiontree.fit(features, target)
```

Обсуждение

Регрессионное дерево принятия решений работает аналогично классификационному дереву принятия решений; однако вместо уменьшения коэффициента разнородности Джини или энтропии потенциальные расщепления по умолчанию измеряются по тому, насколько они уменьшают среднюю квадратическую ошибку (MSE):

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2,$$

где y_i — истинное значение цели; $\hat{y}_i$ — предсказанное значение. В библиотеке scikit-learn регрессия на основе дерева принятия решений может проводиться с помощью класса `DecisionTreeRegressor`. После того как мы натренировали дерево принятия решений, его можно использовать для предсказания целевого значения для некого наблюдения:

```
# Сконструировать новое наблюдение
observation = [[0.02, 16]]

# Предсказать значение наблюдения
model.predict(observation)

array([ 33.])
```

Так же как с классом `DecisionTreeClassifier`, для выбора желаемой меры качества расщепления можно использовать параметр `criterion`. Например, можно построить дерево, расщепления которого уменьшают среднюю абсолютную ошибку (mean absolute error, MAE):

```
# Создать объект-классификатор дерева принятия решений,
# используя энтропию
decisiontree_mae = DecisionTreeRegressor(criterion="mae", random_state=0)

# Натренировать модель
model_mae = decisiontree_mae.fit(features, target)
```

Дополнительные материалы для чтения

- Документация библиотеки scikit-learn по регрессии дерева принятия решений и классу `DecisionTreeRegressor` (**http://bit.ly/2GR63AZ**).

14.3. Визуализация модели дерева принятия решений

Задача

Требуется визуализировать модель, созданную обучающимся алгоритмом дерева принятия решений.

Решение

Экспортировать модель дерева принятия решений в формат DOT, затем визуализировать (рис. 14.1):

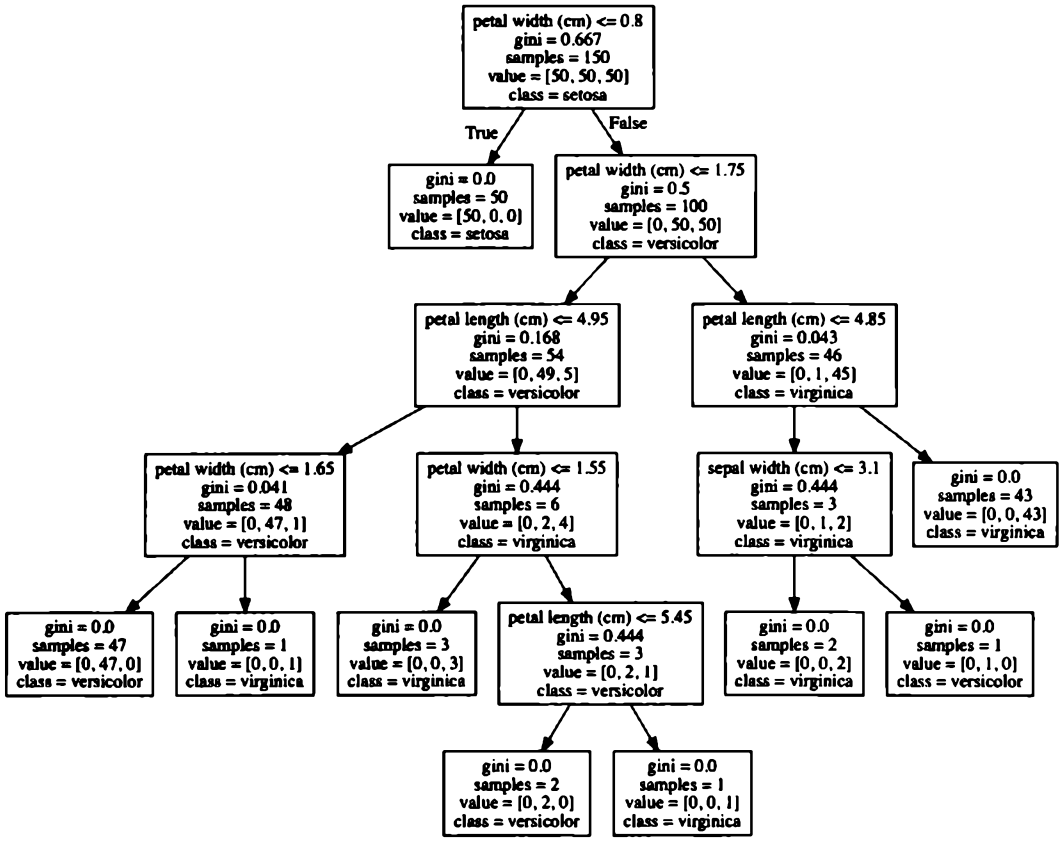


Рис. 14.1

```
# Загрузить библиотеки
import pydotplus
from sklearn.tree import DecisionTreeClassifier
from sklearn import datasets
from IPython.display import Image
from sklearn import tree

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать объект-классификатор дерева принятия решений
decisiontree = DecisionTreeClassifier(random_state=0)

# Натренировать модель
model = decisiontree.fit(features, target)
```

```
# Создать данные в формате DOT
dot_data = tree.export_graphviz(decisiontree,
                                out_file=None,
                                feature_names=iris.feature_names,
                                class_names=iris.target_names)

# Начертить граф
graph = pydotplus.graph_from_dot_data(dot_data)

# Показать граф
Image(graph.create_png())
```

Обсуждение

Одним из преимуществ классификационного дерева принятия решений является то, что мы можем визуализировать всю натренированную модель целиком, и это делает деревья принятия решений одними из самых интерпретируемых моделей в машинном самообучении. В нашем решении мы экспортировали нашу натренированную модель в формат DOT (на языке описания графов), а затем использовали его для построения графа.

Если мы посмотрим на корневой узел, то увидим, что правило принятия решения заключается в том, что если ширина лепестка меньше или равна 0.8, то следует перейти к левой ветви, иначе перейти к правой ветви. Мы также можем увидеть коэффициент неоднородности Джини (`0.667`), количество наблюдений (`150`), количество наблюдений в каждом классе (`[50,50,50]`) и класс, в котором наблюдения будут предсказаны, если мы остановимся на этом узле (`setosa` — щетинистый). Мы также видим, что в этом узле ученик обнаружил, что одно правило принятия решения (ширина лепестка `petal width (cm) <= 0.8`) смогло идеально идентифицировать все наблюдения класса `setosa`. Кроме того, при наличии еще одного правила принятия решения с тем же признаком (`petal width (cm) <= 1.75`) дерево принятия решений способно правильно классифицировать 144 из 150 наблюдений. Это делает ширину лепестка очень важным признаком!

Если мы хотим использовать дерево принятия решений в других приложениях или отчетах, мы можем легко экспортировать визуализацию в файл PDF или изображение в формате PNG:

```
# Создать PDF
graph.write_pdf("iris.pdf")
True

# Создать PNG
graph.write_png("iris.png")
True
```

В то время как это решение визуализировало классификационное дерево принятия решений, его можно также легко использовать для визуализации регрессионного дерева принятия решений.

ПРИМЕЧАНИЕ. Если вы используете MacOS, возможно, придется установить исполняемый файл GraphViz, который потребуется для выполнения приведенного выше программного кода. Это можно сделать с помощью менеджера пакетов Homebrew: `brew install graphviz`. Инструкции по установке менеджера пакетов Homebrew см. на веб-сайте Homebrew.

Дополнительные материалы для чтения

- Менеджер пакетов Homebrew (**https://brew.sh**).

14.4. Тренировка классификационного случайного леса

Задача

Требуется натренировать классификационную модель, используя "лес" рандомизированных деревьев принятия решений.

Решение

Натренировать классификационную модель случайного леса, используя класс `RandomForestClassifier` библиотеки scikit-learn:

```
# Загрузить библиотеки
from sklearn.ensemble import RandomForestClassifier
from sklearn import datasets

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать объект-классификатор случайного леса
randomforest = RandomForestClassifier(random_state=0, n_jobs=-1)

# Натренировать модель
model = randomforest.fit(features, target)
```

Обсуждение

Распространенная проблема деревьев принятия решений заключается в том, что они, как правило, слишком плотно прилегают к тренировочным данным (т. е. подвержены переподгонке). Этот факт мотивировал широкое использование ансамблевого метода самообучения, называемого *случайным лесом*. В случайном лесу тренируется множество деревьев принятия решений, но каждое дерево получает только бутстраповскую выборку наблюдений (т. е. случайную выборку наблюдений

с возвратом, которая соответствует исходному количеству наблюдений), и каждый узел во время определения наилучшего расщепления рассматривает только подмножество признаков. Этот лес рандомизированных деревьев принятия решений (отсюда и название) принимает участие в голосовании с целью определить предсказанный класс.

Как мы можем видеть, сравнивая это решение с рецептом 14.1, класс `RandomForestClassifier` библиотеки scikit-learn работает аналогично классу `DecisionTreeClassifier`:

```
# Сконструировать новое наблюдение
observation = [[ 5, 4, 3, 2]]

# Предсказать класс наблюдения
model.predict(observation)
```

```
array([1])
```

Класс `RandomForestClassifier` использует многие из тех же параметров, что и класс `DecisiontreeClassifier`. Например, можно изменить используемую меру качества расщепления:

```
# Создать объект-классификатор случайного леса,
# используя энтропию
randomforest_entropy = RandomForestClassifier(
    criterion="entropy", random_state=0)

# Натренировать модель
model_entropy = randomforest_entropy.fit(features, target)
```

Однако, будучи лесом, а не отдельным деревом принятия решений, класс `RandomForestClassifier` имеет определенные параметры, которые либо уникальны для случайных лесов, либо особенно важны. Во-первых, параметр `max_features` определяет максимальное количество признаков, которые будут рассматриваться на каждом узле, и принимает ряд аргументов, включая целые числа (количество признаков), вещественные (процент признаков) и `sqrt` (квадратный корень из числа признаков). По умолчанию параметру `max_features` присваивается значение `auto`, которое действует так же, как `sqrt`. Во-вторых, параметр `bootstrap` позволяет задать, будет ли подмножество наблюдений, рассматриваемых для дерева, создаваться с использованием выборки с возвратом (настройка по умолчанию) либо без возврата. В-третьих, `n_estimators` задает количество деревьев решений для включения в лес. В рецепте 10.4 мы рассматривали `n_estimators` как гиперпараметр и визуализировали эффект увеличения числа деревьев на оценочном метрическом показателе. Наконец, хотя это не относится к классификационным случайным лесам, поскольку мы практически тренируем множество моделей деревьев принятия решений, часто полезно использовать все доступные ядра, задав `n_jobs=-1`.

Дополнительные материалы для чтения

- "Случайные леса", исследовательская работа от создателей алгоритма случайных лесов Лео Бреймана и Адель Катлер, факультет статистики Университета в Беркли (**http://bit.ly/2Fxm0Ps**).

14.5. Тренировка регрессионного случайного леса

Задача

Требуется натренировать регрессионную модель, используя "лес" рандомизированных деревьев принятия решений.

Решение

Натренировать регрессионную модель случайного леса, используя класс `RandomForestRegressor` библиотеки scikit-learn:

```
# Загрузить библиотеки
from sklearn.ensemble import RandomForestRegressor
from sklearn import datasets

# Загрузить данные только с двумя признаками
boston = datasets.load_boston()
features = boston.data[:,0:2]
target = boston.target

# Создать объект-классификатор случайного леса
randomforest = RandomForestRegressor(random_state=0, n_jobs=-1)

# Натренировать модель
model = randomforest.fit(features, target)
```

Обсуждение

Точно так же, как мы создаем классификационные леса деревьев принятия решений, мы создаем и регрессионные леса деревьев принятия решений, где каждое дерево использует бутстраповское подмножество наблюдений, и на каждом узле правило принятия решения рассматривает только подмножество признаков. Как и в случае с классификатором на основе случайного леса, имеется несколько важных параметров:

- `max_features` задает максимальное количество признаков для рассмотрения на каждом узле; по умолчанию используется $\sqrt{p}$ признаков, где p — это общее количество признаков;

- `bootstrap` устанавливает, следует ли выполнять выборку с возвратом или нет; по умолчанию `True`;
- `n_estimators` устанавливает количество конструируемых деревьев принятия решений; по умолчанию равно `10`.

Дополнительные материалы для чтения

- Документация библиотеки scikit-learn по регрессии на основе случайного леса и классу `RandomForestRegressor` (**http://bit.ly/2GQZ3nx**).

14.6. Идентификация важных признаков в случайных лесах

Задача

Требуется узнать, какие признаки наиболее важны в модели случайного леса.

Решение

Вычислить и визуализировать важность каждого признака (рис. 14.2):

```
# Загрузить библиотеки
import numpy as np
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestClassifier
from sklearn import datasets

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать объект-классификатор случайного леса
randomforest = RandomForestClassifier(random_state=0, n_jobs=-1)

# Натренировать модель
model = randomforest.fit(features, target)

# Вычислить важности признаков
importances = model.feature_importances_

# Отсортировать важности признаков в нисходящем порядке
indices = np.argsort(importances)[::-1]

# Перераспределить имена признаков, чтобы они совпадали
# с отсортированными важностями признаков
names = [iris.feature_names[i] for i in indices]
```

```
# Создать график
plt.figure()

# Создать заголовок графика
plt.title("Важности признаков")

# Добавить столбики
plt.bar(range(features.shape[1]), importances[indices])

# Добавить имена признаков как метки оси x
plt.xticks(range(features.shape[1]), names, rotation=90)

# Показать график
plt.show()
```

Рис. 14.2

Обсуждение

Одним из основных преимуществ деревьев принятия решений является интерпретируемость. В частности, мы можем визуализировать всю модель целиком (*см. рецепт 13.3*). Однако модель случайного леса состоит из десятков, сотен и даже тысяч деревьев принятия решений. Это делает простую, интуитивную визуализацию модели случайного леса непрактичной. Вместе с тем есть еще один вариант: мы можем сравнить (и визуализировать) относительную важность каждого признака.

В рецепте 13.3 мы визуализировали модель с классификационным деревом принятия решений и увидели, что правила принятия решений, основанные лишь на ширине лепестка, смогли правильно расклассифицировать многие наблюдения. В интуитивном смысле можно сказать: это означает, что в нашем классификаторе ширина

лепестка является важным признаком. Более формально, признаки с расщеплениями, которые имеют более высокое среднее уменьшение разнородности (например, коэффициент разнородности Джини или энтропию в классификаторах и дисперсию в регрессорах) считаются более важными.

Вместе с тем относительно важности признака следует иметь в виду два момента. Во-первых, библиотека scikit-learn требует, чтобы мы разбивали номинальные категориальные признаки на многочисленные бинарные признаки. Это приводит к эффекту разброса важности этого признака по всем бинарным признакам и часто может сделать каждый признак внешне неважным, даже если исходный номинальный категориальный признак очень важен. Во-вторых, если два признака сильно коррелированы, один признак затребует подавляющую часть важности, сделав другой признак внешне гораздо менее важным — что, если этого не учесть, может сказаться на интерпретации.

В библиотеке scikit-learn классификационные и регрессионные деревья принятия решений и случайные леса могут сообщать об относительной важности каждого признака с помощью атрибута `feature_importances_`:

```
# Взглянуть на важности признаков
model.feature_importances_

array([0.11896532, 0.0231668 , 0.36804744, 0.48982043])
```

Чем выше число, тем важнее признак (все оценки важности в сумме составляют 1). Строя график этих значений, мы можем добавить в наши модели случайного леса интерпретируемость.

14.7. Отбор важных признаков в случайных лесах

Задача

Требуется произвести отбор признаков в случайном лесе.

Решение

Идентифицировать важные признаки и перетренировать модель с использованием только наиболее важных признаков:

```
# Загрузить библиотеки
from sklearn.ensemble import RandomForestClassifier
from sklearn import datasets
from sklearn.feature_selection import SelectFromModel

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target
```

```
# Создать объект-классификатор случайного леса
randomforest = RandomForestClassifier(random_state=0, n_jobs=-1)

# Создать объект, который отбирает признаки с важностью,
# большей или равной порогу
selector = SelectFromModel(randomforest, threshold=0.3)

# Выполнить подгонку новой матрицы признаков, используя селектор
features_important = selector.fit_transform(features, target)

# Натренировать случайный лес, используя наиболее важные признаки
model = randomforest.fit(features_important, target)
```

Обсуждение

Существуют ситуации, когда может потребоваться уменьшить количество признаков в нашей модели. Например, может возникнуть потребность в уменьшении дисперсии модели, или нам может понадобиться улучшить интерпретируемость, включив только самые важные признаки.

В библиотеке scikit-learn для создания модели с уменьшенным количеством признаков можно использовать простой двухэтапный рабочий процесс. Во-первых, мы тренируем модель случайного леса, учитывая все признаки. Далее мы применяем эту модель для определения наиболее важных признаков. Затем мы создаем новую матрицу признаков, которая включает только эти признаки. В нашем решении мы использовали класс `SelectFromModel`, реализующий метод для создания матрицы признаков, содержащей только те признаки, значение которых больше или равно некоторому пороговому значению. Наконец, мы создали новую модель, используя только эти признаки.

Следует отметить, что существует два предостережения в отношении такого подхода. Во-первых, номинальные категориальные признаки, которые были кодированы с помощью кодировки с одним активным состоянием, засвидетельствуют разбавление важности признаков по бинарным признакам. Во-вторых, важность сильно коррелированных признаков будет практически присвоена одному признаку и неравномерно распределена между обоими признаками.

Дополнительные материалы для чтения

- "Отбор переменных с использованием случайных лесов" (Robin Genuer, Jean-Michel Poggi, Christine Tuleau-Malot. Variable selection using Random Forests), исследовательская статья на открытых архивах HAL (**http://bit.ly/2FvG70D**).

14.8. Обработка несбалансированных классов

Задача

Дан вектор целей с очень несбалансированными классами, и требуется натренировать модель случайного леса.

Решение

Натренировать модель дерева принятия решений или случайного леса с параметром `class_weight="balanced"`:

```
# Загрузить библиотеки
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn import datasets

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Сделать класс сильно несбалансированным,
# удалив первые 40 наблюдений
features = features[40:,:]
target = target[40:]

# Создать вектор целей, назначив классам значение 0 либо 1
target = np.where((target == 0), 0, 1)

# Создать объект-классификатор случайного леса
randomforest = RandomForestClassifier(
    random_state=0, n_jobs=-1, class_weight="balanced")

# Натренировать модель
model = randomforest.fit(features, target)
```

Обсуждение

Когда мы выполняем машинное самообучение в реальном мире, несбалансированные классы становятся широко распространенной проблемой. Если эту проблему оставить без внимания, то наличие несбалансированных классов может снизить результативность нашей модели. В рецепте 17.5 рассмотрено несколько стратегий решения проблемы несбалансированности классов во время предобработки. Однако многие обучающиеся алгоритмы в scikit-learn сопровождаются встроенными методами, позволяющими корректировать несбалансированность классов. С помощью аргумента `class_weight` объекту `RandomForestClassifier` можно поручить корректировать несбалансированные классы. Если ему передать словарь в виде имен классов и соответствующих желаемых весов (например, `{"мужской": 0.2, "женский": 0.8}`), то объект `RandomForestClassifier` выполнит соответствующую перевесовку классов. Тем не менее нередко более полезным аргументом является `balanced`, при котором классы автоматически взвешиваются обратно пропорционально тому, как часто они появляются в данных:

$$w_j = \frac{n}{kn_j},$$

где w_j — вес класса j; n — количество наблюдений; n_j — количество наблюдений класса j; k — общее количество классов. Например, в нашем решении имеется 2 класса (k), 110 наблюдений (n) и соответственно 10 и 100 наблюдений в каждом классе (n_j). Если мы взвешиваем классы, используя `class_weight= "balanced"`, то меньший класс взвешивается больше:

```
# Вычислить вес для малого класса
110/(2*10)
```

```
5.5
```

В то время как более крупный класс взвешивается меньше:

```
# Вычислить вес для крупного класса
110/(2*100)
```

```
0.55
```

14.9. Управление размером дерева

Задача

Требуется вручную определить структуру и размер дерева принятия решений.

Решение

Использовать древесные структурные параметры в древесных обучающихся алгоритмах библиотеки scikit-learn:

```
# Загрузить библиотеки
from sklearn.tree import DecisionTreeClassifier
from sklearn import datasets

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать объект-классификатор дерева принятия решений
decisiontree = DecisionTreeClassifier(random_state=0,
                                      max_depth=None,
                                      min_samples_split=2,
                                      min_samples_leaf=1,
                                      min_weight_fraction_leaf=0,
                                      max_leaf_nodes=None,
                                      min_impurity_decrease=0)

# Натренировать модель
model = decisiontree.fit(features, target)
```

Обсуждение

Древесные обучающиеся алгоритмы библиотеки scikit-learn имеют множество приемов управления размером дерева принятия решений. Доступ к ним осуществляется через следующие параметры.

- `max_depth` — максимальная глубина дерева. Если `None`, то дерево выращивается до тех пор, пока все листья не станут однородными. Если целое число, то дерево практически "обрезается" до этой глубины.
- `min_samples_split` — минимальное количество наблюдений в узле прежде, чем этот узел будет расщеплен. Если в качестве аргумента задано целое число, то оно определяет голый минимум, а если вещественное, то минимум равен проценту от общего количества наблюдений.
- `min_samples_leaf` — минимальное количество наблюдений, которое должно находиться в листе. Использует те же аргументы, что и `min_samples_split`.
- `max_leaf_nodes` — максимальное количество листьев.
- `min_impurity_split` — минимальное требуемое уменьшение разнородности прежде, чем расщепление будет выполнено.

Хотя полезно знать, что эти параметры существуют, скорее всего, мы будем использовать только `max_depth` и `min_impurity_split`, потому что более мелкие деревья (иногда называемые *пнями*) являются более простыми моделями и, следовательно, имеют более низкую дисперсию.

14.10. Улучшение результативности с помощью бустинга

Задача

Требуется модель с лучшей результативностью, чем деревья принятия решений или случайные леса.

Решение

Натренировать бустированную модель, используя объекты классов `AdaBoostClassifier` или `AdaBoostRegressor`:

```
# Загрузить библиотеки
from sklearn.ensemble import AdaBoostClassifier
from sklearn import datasets

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target
```

```
# Создать объект-классификатор дерева принятия решений
# на основе алгоритма adaboost
adaboost = AdaBoostClassifier(random_state=0)

# Натренировать модель
model = adaboost.fit(features, target)
```

Обсуждение

В случайном лесе ансамбль (группа) рандомизированных деревьев принятия решений предсказывает вектор целей. Альтернативный и часто более мощный подход называется *бустингом*. В одной из форм бустинга, которая называется AdaBoost[1], мы итеративно тренируем ряд слабых моделей (чаще всего это неглубокое дерево принятия решений, иногда называемое пнем), при этом каждая итерация дает более высокий приоритет наблюдениям, которые предыдущая модель предсказывала неправильно. Более конкретно, алгоритм AdaBoost работает следующим образом:

1. Присвоить каждому наблюдению x начальное значение веса $w_i = 1/n$, где n — это общее количество наблюдений в данных.
2. Натренировать "слабую" модель на данных.
3. Для каждого наблюдения:
   - если слабая модель правильно предсказывает x_i, то вес w_i увеличивается;
   - если слабая модель неправильно предсказывает x_i, то вес w_i уменьшается.
4. Натренировать новую слабую модель, в которой наблюдения с большим весом w_i имеют больший приоритет.
5. Повторять шаги 3 и 4 до тех пор, пока данные не будут точно предсказаны либо не будет натренировано текущее количество слабых моделей.

В результате будет получена агрегированная модель, в которой отдельные слабые модели фокусируются на более сложных (с точки зрения предсказания) наблюдениях. В библиотеке scikit-learn алгоритм AdaBoost можно реализовать с помощью классов `AdaBoostClassifier` или `AdaBoostRegressor`. Наиболее важными параметрами являются `base_estimator`, `n_estimators` и `learning_rate`.

- `base_estimator` — это обучающийся алгоритм, используемый для тренировки слабых моделей. Этот параметр почти всегда будет оставаться без изменений, потому что, безусловно, наиболее распространенным учеником для использования с AdaBoost является дерево принятия решений, и передается в этот параметр по умолчанию.

[1] AdaBoost — ранняя версия бустинга, опирающаяся на перевесовку данных на основе остатков, т. е. разницы между наблюдаемыми и подогнанными значениями. Бустинг — общая методика подгонки последовательности моделей путем предоставления большего веса признакам с более крупными остатками для каждой последующей итерации цикла. — *Прим. перев.*

- `n_estimators` — количество моделей, подлежащих итеративной тренировке.
- `learning_rate` — это вклад каждой модели в веса и по умолчанию равняется 1. Уменьшение скорости заучивания будет означать, что вес будет увеличиваться или уменьшаться в небольшой степени, заставляя модель тренироваться медленнее (но иногда приводя к лучшим показателям результативности).
- `loss` используется исключительно с объектом `AdaBoostRegressor` и задает функцию потери для использования при обновлении весов. По умолчанию используется линейная функция потери, но ее можно заменить на квадратичную или экспоненциальную.

Дополнительные материалы для чтения

- Объяснение алгоритма AdaBoost от создателя алгоритма Роберта Шапире, персональная веб-страница (**http://bit.ly/2FCS30E**).

14.11. Оценивание случайных лесов с помощью ошибок внепакетных наблюдений

Задача

Требуется оценить модель случайного леса без перекрестной проверки.

Решение

Вычислить внепакетную (out-of-bag, OOB) оценку модели:

```
# Загрузить библиотеки
from sklearn.ensemble import RandomForestClassifier
from sklearn import datasets

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать объект-классификатор случайного леса
randomforest = RandomForestClassifier(
    random_state=0, n_estimators=1000, oob_score=True, n_jobs=-1)

# Натренировать модель
model = randomforest.fit(features, target)

# Взглянуть на внепакетную ошибку
randomforest.oob_score_

0.95333333333333337
```

Обсуждение

В случайных лесах каждое дерево принятия решений тренируется с использованием бутстраповского подмножества наблюдений. Это означает, что для каждого дерева существует отдельное подмножество наблюдений, которые не используются для тренировки этого дерева. Они называются *внепакетными наблюдениями* (out-of-bag, OOB). Их можно использовать в качестве тестового набора для оценки результативности нашего случайного леса.

Для каждого наблюдения обучающийся алгоритм сравнивает истинное значение наблюдений с предсказанием из подмножества деревьев, не натренированных с помощью этого наблюдения. Вычисляемая общая оценка предоставляет единую меру результативности случайного леса. Внепакетная оценка является альтернативой перекрестной проверке[2].

В библиотеке scikit-learn можно выполнять внепакетное оценивание случайного леса, задав в объекте случайного леса (т. е. объекте класса `RandomForestClassifier`) параметр `oob_score=True`. Значение оценки может быть получено с помощью атрибута `oob_score_`.

[2] Внепакетная оценка, или ошибка (out-of-bag, OOB) — это средняя ошибка предсказания на каждом тренировочном примере x_i с использованием только тех деревьев, которые не имели x_i в своей бутстраповской выборке (см. **https://en.wikipedia.org/wiki/Out-of-bag_error**). — *Прим. перев.*

K ближайших соседей

Введение

Классификатор *k* ближайших соседей (KNN) является одним из самых простых и наиболее часто используемых классификаторов в контролируемом машинном самообучении. KNN часто считается ленивым учеником; он технически не тренирует модель делать предсказания. Вместо этого предсказывается, что наблюдение будет классом наибольшей доли *k* ближайших наблюдений. Например, если наблюдение с неизвестным классом окружено наблюдением класса 1, то это наблюдение классифицируется как класс 1. В этой главе мы рассмотрим, как с помощью библиотеки scikit-learn создавать и использовать классификатор KNN.

15.1. Отыскание ближайших соседей наблюдения

Задача

Требуется найти *k* ближайших наблюдений (соседей) некоторого наблюдения.

Решение

Использовать класс `NearestNeighbors` библиотеки scikit-learn:

```
# Загрузить библиотеки
from sklearn import datasets
from sklearn.neighbors import NearestNeighbors
from sklearn.preprocessing import StandardScaler

# Загрузить данные
iris = datasets.load_iris()
features = iris.data

# Создать стандартизатор
standardizer = StandardScaler()

# Стандартизировать признаки
features_standardized = standardizer.fit_transform(features)
```

```
# Два ближайших соседа
nearest_neighbors =
    NearestNeighbors(n_neighbors=2).fit(features_standardized)

# Создать наблюдение
new_observation = [ 1, 1, 1, 1]

# Найти расстояния и индексы ближайших соседей наблюдения
distances, indices = nearest_neighbors.kneighbors([new_observation])

# Взглянуть на ближайших соседей
features_standardized[indices]

array([[[1.03800476, 0.56925129, 1.10395287, 1.1850097 ],
        [0.79566902, 0.33784833, 0.76275864, 1.05353673]]])
```

Обсуждение

В нашем решении мы использовали набор данных цветков ириса. Мы создали наблюдение, `new_observation`, с некоторыми значениями, а затем нашли два наблюдения, которые ближе всего к нашему наблюдению. Переменная `indices` содержит наиболее близкие местоположения наблюдений в нашем наборе данных, и `X[indices]` показывает значения этих наблюдений. Интуитивно расстояние можно рассматривать, как меру сходства, поэтому два ближайших наблюдения — это два цветка, наиболее похожих на цветок, который мы создали.

Как измерять расстояние? Библиотека scikit-learn предлагает широкий выбор метрических показателей расстояния d, включая евклидово расстояние:

$$d_{\text{евклидово}} = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

и манхэттенское расстояние:

$$d_{\text{манхэттенское}} = \sum_{i=1}^{n}|x_i - y_i|.$$

По умолчанию класс `NearestNeighbors` использует расстояние Минковского:

$$d_{\text{Минковского}} = \left(\sum_{i=1}^{n}|x_i - y_i|^p\right)^{1/p},$$

где x_i и y_i — два наблюдения, между которыми мы вычисляем расстояние. Расстояние Минковского включает гиперпараметр p, где $p=1$ — это манхэттенское расстояние, $p=2$ — евклидово расстояние и т. д. В библиотеке scikit-learn по умолчанию $p=2$.

Метрический показатель расстояния можно установить с помощью параметра `metric`:

```
# Найти двух ближайших соседей на основе евклидова расстояния
nearestneighbors_euclidean = NearestNeighbors(
    n_neighbors=2, metric='euclidean').fit(features_standardized)
```

Созданная нами переменная `distance` содержит фактическое измерение расстояния до каждого из двух ближайших соседей:

```
# Взглянуть на расстояния
distances

array([[0.48168828, 0.73440155]])
```

Кроме того, для создания матрицы, показывающей ближайших соседей каждого наблюдения, можно использовать метод `kneighbors_graph`:

```
# Найти трех ближайших соседей каждого наблюдения
# на основе евклидова расстояния (включая себя)
nearestneighbors_euclidean = NearestNeighbors(
    n_neighbors=3, metric="euclidean").fit(features_standardized)

# Список списков, показывающий трех ближайших соседей
# каждого наблюдения (включая себя)
nearest_neighbors_with_self = nearestneighbors_euclidean.kneighbors_graph(
    features_standardized).toarray()

# Удалить единицы, отметив наблюдение, как ближайший сосед к себе
for i, x in enumerate(nearest_neighbors_with_self):
    x[i] = 0

# Взглянуть на ближайших соседей первого наблюдения
nearest_neighbors_with_self[0]

array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 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., 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., 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., 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., 0., 0., 0., 0., 0., 0.])
```

Когда мы находим ближайших соседей или используем любой обучающийся алгоритм, основанный на расстоянии, важно преобразовать признаки так, чтобы они были на одной шкале. Причина в том, что метрики расстояния относятся ко всем признакам, как если бы они были на одной шкале, но если один признак находится в миллионах долларов, а второй — в процентах, то вычисленное расстояние будет смещено в сторону первого. В нашем решении мы решили эту потенциальную проблему путем стандартизации признаков с помощью класса `StandardScaler`.

15.2. Создание классификационной модели *k* ближайших соседей

Задача

Дано наблюдение неизвестного класса, и требуется предсказать его класс на основе класса его соседей.

Решение

Если набор данных не очень большой, то использовать класс `KNeighborsClassifier`:

```
# Загрузить библиотеки
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
from sklearn import datasets

# Загрузить данные
iris = datasets.load_iris()
X = iris.data
y = iris.target

# Создать стандартизатор
standardizer = StandardScaler()

# Стандартизировать признаки
X_std = standardizer.fit_transform(X)

# Натренировать классификатор KNN с 5 соседями
knn = KNeighborsClassifier(n_neighbors=5, n_jobs=-1).fit(X_std, y)

# Создать два наблюдения
new_observations = [[ 0.75, 0.75, 0.75, 0.75],
                    [ 1, 1, 1, 1]]

# Предсказать класс двух наблюдений
knn.predict(new_observations)

array([1, 2])
```

Обсуждение

В KNN, если дано наблюдение x_u с неизвестным целевым классом, то алгоритм сначала идентифицирует k ближайших наблюдений (иногда называемых окрестностью x_u) на основе некоторого метрического показателя расстояния (например, евклидова расстояния), затем эти k наблюдений "голосуют" на основе их класса,

и класс, который побеждает в голосовании, является предсказанным классом для x_u. Более формально вероятность x_u некоторого класса j имеет вид:

$$\frac{1}{k}\sum_{i \in v} I(y_i = j),$$

где v — k-е наблюдение в окрестности x_u; y_i — класс i-го наблюдения; I — индикаторная функция (т. е. 1 — истинно, 0 — в противном случае). В библиотеке scikit-learn эти вероятности можно увидеть с помощью метода `predict_proba`:

```
# Взглянуть на вероятность, что каждое наблюдение
# является одним из трех классов
knn.predict_proba(new_observations)
```

```
array([[0. , 0.6, 0.4],
       [0. , 0. , 1. ]])
```

Класс с наибольшей вероятностью становится предсказанным классом. Например, в приведенных выше результатах первое наблюдение должно быть классом 1 ($Pr = 0.6$), в то время как второе наблюдение должно быть классом 2 ($Pr = 1$), и это именно то, что мы видим:

```
knn.predict(new_observations)
```

```
array([1, 2])
```

В библиотеке scikit-learn класс `KNeighborsClassifier` содержит ряд важных параметров, которые следует рассмотреть. Во-первых, параметр `metric` задает используемый метрический показатель расстояния (подробнее см. рецепт 14.1). Во-вторых, параметр `n_jobs` определяет, сколько ядер компьютера использовать. Поскольку предсказание требует вычисления расстояния от некой точки до каждой отдельной точки данных, настоятельно рекомендуется использовать несколько ядер. В-третьих, параметр `algorithm` задает метод расчета ближайших соседей. Хотя существуют реальные различия в алгоритмах, по умолчанию класс `KNeighborsClassifier` пытается автоматически выбрать наилучший алгоритм, поэтому вам беспокоиться об этом параметре особо не нужно. В-четвертых, по умолчанию классификатор `KNeighborsClassifier` работает так, как мы описывали ранее, при этом каждое наблюдение в окрестности получает один голос; однако если мы зададим параметр веса `weights` равным значению `distance`, то голоса более близких наблюдений будут весить больше, чем наблюдения, расположенные дальше. В интуитивном плане это имеет смысл, поскольку более похожие соседи могут рассказать нам больше о классе наблюдения, чем другие.

Наконец, как описано в рецепте 14.1, поскольку при вычислении расстояний все объекты обрабатываются так, как если бы они находились на одной шкале, важно перед использованием классификатора KNN стандартизировать признаки.

15.3. Идентификация наилучшего размера окрестности

Задача

Требуется выбрать наилучшее значение для k в классификационной модели k ближайших соседей.

Решение

Использовать методы отбора модели, такие как поиск в решетке параметров, реализованный в классе `GridSearchCV`:

```
# Загрузить библиотеки
from sklearn.neighbors import KNeighborsClassifier
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline, FeatureUnion
from sklearn.model_selection import GridSearchCV

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать стандартизатор
standardizer = StandardScaler()

# Стандартизировать признаки
features_standardized = standardizer.fit_transform(features)

# Создать классификатор KNN
knn = KNeighborsClassifier(n_neighbors=5, n_jobs=-1)

# Создать конвейер
pipe = Pipeline([("standardizer", standardizer), ("knn", knn)])

# Создать пространство вариантов значений
search_space = [{"knn__n_neighbors": [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]}]

# Создать объект решеточного поиска
classifier = GridSearchCV(
    pipe, search_space, cv=5, verbose=0).fit(features_standardized, target)
```

Обсуждение

В классификаторах KNN размер k имеет реальные последствия. В машинном самообучении мы пытаемся найти баланс между смещением и дисперсией, и не так уж много мест, где это проявляется так же явно, как в значении k. Если $k = n$, где n —

число наблюдений, то мы имеем высокое смещение, но низкую дисперсию. Если $k=1$, то мы будем иметь низкое смещение, но высокую дисперсию. Лучшая модель будет получена из нахождения значения k, которое уравновешивает это компромиссное соотношение смещения-дисперсии. В нашем решении мы использовали объект класса `GridSearchCV` для проведения пятиблочной перекрестной проверки на классификаторах KNN с различными значениями k. Когда она будет завершена, мы увидим k, которое производит наилучшую модель:

```
# Наилучший размер окрестности (k)
classifier.best_estimator_.get_params()["knn__n_neighbors"]
```

```
6
```

15.4. Создание радиусного классификатора ближайших соседей

Задача

Дано наблюдение неизвестного класса, и требуется предсказать его класс на основе класса всех наблюдений в пределах определенного расстояния.

Решение

Использовать класс `RadiusNeighborsClassifier`:

```
# Загрузить библиотеки
from sklearn.neighbors import RadiusNeighborsClassifier
from sklearn.preprocessing import StandardScaler
from sklearn import datasets

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать стандартизатор
standardizer = StandardScaler()

# Стандартизировать признаки
features_standardized = standardizer.fit_transform(features)

# Натренировать радиусный классификатор соседей
rnn = RadiusNeighborsClassifier(
    radius=.5, n_jobs=-1).fit(features_standardized, target)

# Создать наблюдение
new_observations = [[ 1, 1, 1, 1]]
```

```
# Предсказать класс наблюдения
rnn.predict(new_observations)
```

```
array([2])
```

Обсуждение

В классификации KNN класс наблюдений предсказывается из классов его k соседей. Менее распространенным методом является классификация в радиусном классификаторе (RNN), где класс наблюдения предсказывается из классов всех наблюдений в пределах заданного радиуса r. В библиотеке scikit-learn радиусный классификатор `RadiusNeighborsClassifier` очень похож на классификатор k ближайших соседей `KNeighborsClassifier`, за исключением двух параметров. Во-первых, в радиусном классификаторе `RadiusNeighborsClassifier` нужно с помощью параметра `radius` указать радиус фиксированной области, используемой для определения, является ли наблюдение соседом. Если нет какой-либо существенной причины для установки параметра `radius` в некоторое значение, лучше рассматривать его как любой другой гиперпараметр и настраивать его во время отбора модели. Второй полезный параметр — `outlier_label`, указывающий на то, какую метку дать наблюдению, которое не имеет наблюдений в заданном радиусе, что само по себе часто может быть полезным инструментом для идентификации выбросов.

Логистическая регрессия

Введение

Несмотря на то, что логистическая регрессия называется регрессией, она фактически является широко используемым методом контролируемой классификации. Логистическая регрессия и ее расширения, такие как полиномиальная логистическая регрессия, позволяют предсказывать вероятность того, что наблюдение относится к определенному классу, используя простой и понятный подход. В этой главе мы рассмотрим тренировку различных классификаторов с помощью библиотеки scikit-learn.

16.1. Тренировка бинарного классификатора

Задача

Требуется натренировать простую классификационную модель.

Решение

Натренировать логистическую регрессию в библиотеке scikit-learn с помощью класса `LogisticRegression`:

```
# Загрузить библиотеки
from sklearn.linear_model import LogisticRegression
from sklearn import datasets
from sklearn.preprocessing import StandardScaler

# Загрузить данные только с двумя классами
iris = datasets.load_iris()
features = iris.data[:100,:]
target = iris.target[:100]

# Стандартизировать признаки
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# Создать объект логистической регрессии
logistic_regression = LogisticRegression(random_state=0)
```

```
# Натренировать модель
model = logistic_regression.fit(features_standardized, target)
```

Обсуждение

Несмотря на наличие слова "регрессия" в названии, логистическая регрессия на самом деле является широко используемым бинарным классификатором (т. е. вектор целей может принимать только два значения). В логистической регрессии линейная модель (например, $\beta_0 + \beta_1 x$) включается в логистическую (так называемую сигмоидальную) функцию $\frac{1}{1+e^{-z}}$ таким образом, что:

$$P(y_i = 1 | X) = \frac{1}{1+e^{-(\beta_0+\beta_1 x)}},$$

где $P(y_i = 1 | X)$ — вероятность, что целевое значение i-го наблюдения y_i является классом 1; X — тренировочные данные; β_0 и β_1 — параметры, которые необходимо заучить; e — эйлерово число. Эффект логистической функции заключается в ограничении значения результата функции диапазоном между 0 и 1, чтобы его можно было интерпретировать как вероятность. Если $P(y_i = 1 | X)$ больше 0.5, то предсказывается класс 1, в противном случае — класс 0.

В библиотеке scikit-learn можно заучить логистическую регрессионную модель с помощью класса `LogisticRegression`. После тренировки модели ее можно использовать для предсказания класса новых наблюдений:

```
# Создать новое наблюдение
new_observation = [[.5, .5, .5, .5]]

# Предсказать класс
model.predict(new_observation)
```

```
array([1])
```

В этом примере наше наблюдение было предсказано как класс 1. Кроме того, мы можем увидеть вероятность принадлежности наблюдения к каждому классу:

```
# Взглянуть на предсказанные вероятности
model.predict_proba(new_observation)
```

```
array([[0.18823041, 0.81176959]])
```

Наше наблюдение имело 18.8% шанс быть классом 0 и 81.1% шанс быть классом 1.

16.2. Тренировка мультиклассового классификатора

Задача

Дано более двух классов, и требуется натренировать классификационную модель.

Решение

Натренировать логистическую регрессию в библиотеке scikit-learn с помощью класса `LogisticRegression`, используя методы "один против остальных" либо полиномиальные методы:

```
# Загрузить библиотеки
from sklearn.linear_model import LogisticRegression
from sklearn import datasets
from sklearn.preprocessing import StandardScaler

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Стандартизировать признаки
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# Создать объект логистической регрессии
# по методу "один против остальных"
logistic_regression = LogisticRegression(random_state=0, multi_class="ovr")

# Натренировать модель
model = logistic_regression.fit(features_standardized, target)
```

Обсуждение

Сами по себе логистические регрессии являются исключительно бинарными классификаторами, т. е. они не могут обрабатывать векторы целей с более чем двумя классами. Однако два умных расширения логистической регрессии делают именно это. Во-первых, в логистической регрессии по методу "один против остальных" (one-vs-rest, OVR) для каждого класса тренируется отдельная модель, предсказывающая, является ли наблюдение этим классом или нет (что делает его задачей бинарной классификации). Такой классификатор исходит из того, что каждая классификационная задача (например, класс 0 или нет) является независимой.

В качестве альтернативы в полиномиальной логистической регрессии (multinomial logistic regression, MLR) логистическая функция, которую мы видели в рецепте 15.1, заменяется функцией softmax (многопеременной логистической функцией):

$$P(y_i = k \mid X) = \frac{e^{\beta_k x_i}}{\sum_{j=1}^{K} e^{\beta_j x_i}},$$

где $P(y_i = k \mid X)$ — вероятность, что целевое значение i-го наблюдения y_i является классом k; K — общее количество классов. Одним из практических преимуществ MLR является то, что его предсказанные вероятности с использованием метода `predict_proba` более надежны (т. е. лучше откалиброваны).

При использовании класса `LogisticRegression` можно выбрать один из двух методов по своему усмотрению, при этом метод OVR, `ovr`, указывается в аргументе `multi_class` по умолчанию. Мы можем переключиться на метод MNL, присвоив этому аргументу значение `multinomial`.

16.3. Снижение дисперсии с помощью регуляризации

Задача

Требуется уменьшить дисперсию логистической регрессионной модели.

Решение

Настроить гиперпараметр силы регуляризации `C`:

```
# Загрузить библиотеки
from sklearn.linear_model import LogisticRegressionCV
from sklearn import datasets
from sklearn.preprocessing import StandardScaler

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Стандартизировать признаки
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# Создать объект-классификатор на основе дерева принятия решений
logistic_regression = LogisticRegressionCV(
    penalty='l2', Cs=10, random_state=0, n_jobs=-1)

# Натренировать модель
model = logistic_regression.fit(features_standardized, target)
```

Обсуждение

Регуляризация — это метод штрафования сложных моделей с целью уменьшения их дисперсии. В частности, штрафной член добавляется в функцию потери, которую мы пытаемся минимизировать, как правило, это штрафы L^1 и L^2. В штрафе L^1:

$$\alpha\sum_{j=1}^{p}\left|\hat{\beta}_j\right|,$$

где $\hat{\beta}_j$ — параметры j-го из p заучиваемых признаков; α — гиперпараметр, обозначающий силу регуляризации. В случае со штрафом L^2:

$$\alpha\sum_{j=1}^{p}\hat{\beta}_j^2.$$

Более высокие значения α увеличивают штраф за более крупные значения параметров (т. е. более сложные модели). Библиотека scikit-learn следует общему методу использования обозначения C вместо α, где C — это обратная величина силы регуляризации: $C = 1/\alpha$. Для того чтобы уменьшить дисперсию при использовании логистической регрессии, мы можем рассматривать C как гиперпараметр, который нужно настроить, чтобы найти значение C, создающее наилучшую модель. В библиотеке scikit-learn можно использовать класс `LogisticRegressionCV` для эффективной настройки C. Параметр `Cs` класса `LogisticRegressionCV` может либо принимать диапазон значений C для поиска (если в качестве аргумента указан список вещественных чисел), либо, если задано целое число, сгенерирует список из заданного количества потенциальных значений, полученных из логарифмической шкалы между $-10\,000$ и $10\,000$.

К сожалению, класс `LogisticRegressionCV` не позволяет производить поиск по различным штрафным членам. Для этого мы должны использовать менее эффективные методы отбора модели, описанные в *главе 12*.

16.4. Тренировка классификатора на очень крупных данных

Задача

Требуется натренировать простую классификационную модель на очень крупном наборе данных.

Решение

Натренировать логистическую регрессию в библиотеке scikit-learn с помощью класса `LogisticRegression`, используя стохастический среднеградиентный (stochastic average gradient, SAG) решатель:

```
# Загрузить библиотеки
from sklearn.linear_model import LogisticRegression
from sklearn import datasets
from sklearn.preprocessing import StandardScaler

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Стандартизировать признаки
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# Создать объект логистической регрессии
logistic_regression = LogisticRegression(random_state=0, solver="sag")

# Натренировать модель
model = logistic_regression.fit(features_standardized, target)
```

Обсуждение

Класс `LogisticRegression` библиотеки scikit-learn предлагает ряд методов для тренировки логистической регрессии, называемых решателями. Подавляющую часть времени scikit-learn будет выбирать для нас наилучший решатель автоматически или предупреждать, что мы не можем что-то сделать с этим решателем. Однако один особый случай мы должны знать.

Хотя его точное объяснение лежит за рамками этой книги (см. слайды Марка Шмидта в *разд. "Дополнительные материалы для чтения"* далее), стохастический среднеградиентный спуск позволяет тренировать модель намного быстрее, чем другие решатели, в тех случаях, когда данные очень крупные. Вместе с тем он также очень чувствителен к шкалированию признаков, поэтому особенно важна стандартизация наших признаков. Обучающийся алгоритм можно настроить на использование этого решателя, задав параметр `solver='sag'`.

Дополнительные материалы для чтения

- Марк Шмидт "Минимизация конечных сумм с помощью алгоритма стохастического среднего градиента" (Международная исследовательская станция Banff, Канада; **http://bit.ly/2GRrVw0**).

16.5. Обработка несбалансированных классов

Задача

Требуется натренировать простую классификационную модель.

Решение

Натренировать логистическую регрессию в библиотеке scikit-learn с помощью класса `LogisticRegression`:

```
# Загрузить библиотеки
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn import datasets
from sklearn.preprocessing import StandardScaler

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Сделать класс сильно несбалансированным, удалив первые 40 наблюдений
features = features[40:,:]
target = target[40:]

# Создать вектор целей, указав либо класс 0, либо 1
target = np.where((target == 0), 0, 1)

# Стандартизировать признаки
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# Создать объект-классификатор дерева принятия решений
logistic_regression = LogisticRegression(random_state=0, class_weight="balanced")

# Натренировать модель
model = logistic_regression.fit(features_standardized, target)
```

Обсуждение

Как и многие другие обучающиеся алгоритмы в библиотеке scikit-learn, логистическая регрессия сопровождается встроенным методом обработки несбалансированных классов. Если мы имеем сильно несбалансированные классы и не решили эту проблему во время предобработки, у нас есть возможность использовать параметр

`class_weight` для взвешивания классов, чтобы определенно получить сбалансированное сочетание каждого класса. В частности, аргумент `balanced` будет взвешивать классы автоматически обратно пропорциональны их частоте:

$$w_j = \frac{n}{kn_j},$$

где w_j — вес класса j; n — количество наблюдений; n_j — количество наблюдений класса j; k — общее количество классов.

Опорно-векторные машины

Введение

Для того чтобы разобраться в принципе работы опорно-векторных машин (или машин опорных векторов), мы должны понимать, что такое гиперплоскости. Формально гиперплоскость является $n-1$ подпространством в n-мерном пространстве. Хотя это звучит сложно, на самом деле это не так. Например, если мы хотим разделить двухмерное пространство, мы используем одномерную гиперплоскость (т. е. прямую). Если мы хотим разделить трехмерное пространство, мы используем двухмерную гиперплоскость (т. е. плоский лист бумаги или простыню). Гиперплоскость — всего лишь обобщение этого понятия до n размерностей.

Опорно-векторные машины классифицируют данные путем нахождения гиперплоскости, которая максимизирует допустимый промежуток между классами в тренировочных данных. В двухмерном примере с двумя классами мы можем представить гиперплоскость как самую широкую прямую "полосу" (т. е. линию с полями), которая разделяет два класса.

В этой главе мы обсудим тренировку опорно-векторных машин в различных ситуациях и внимательно рассмотрим их внутреннее устройство, чтобы понять, как этот подход можно расширить для решения распространенных задач.

17.1. Тренировка линейного классификатора

Задача

Требуется натренировать модель, чтобы классифицировать наблюдения.

Решение

Использовать опорно-векторный классификатор (support vector classifier, SVC), чтобы найти гиперплоскость, которая максимизирует промежутки (зазоры) между классами:

```
# Загрузить библиотеки
from sklearn.svm import LinearSVC
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
import numpy as np
```

```
# Загрузить данные всего с двумя классами и двумя признаками
iris = datasets.load_iris()
features = iris.data[:100,:2]
target = iris.target[:100]

# Стнадартизировать признаки
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# Создать опорно-векторный классификатор
svc = LinearSVC(C=1.0)

# Натренировать модель
model = svc.fit(features_standardized, target)
```

Обсуждение

Класс `LinearSVC` библиотеки scikit-learn реализует простой опорно-векторный классификатор (SVC). Для того чтобы на интуитивном уровне понять, что делает опорно-векторный классификатор, давайте выведем данные и гиперплоскость на график. Хотя опорно-векторные классификаторы хорошо работают в больших размерностях, в нашем решении мы загрузили только два признака и взяли подмножество наблюдений, чтобы данные содержали только два класса. Это позволит визуализировать модель. Напомним, что опорно-векторный классификатор пытается найти гиперплоскость — прямую, когда имеется всего две размерности — с максимальным промежутком между классами. В приведенном далее фрагменте кода мы построим два класса на двухмерном пространстве, а затем начертим гиперплоскость (рис. 17.1):

```
# Загрузить библиотеку
from matplotlib import pyplot as plt

# Вывести точки данных на график и расцветить, используя их класс
color = ["black" if c == 0 else "lightgrey" for c in target]
plt.scatter(features_standardized[:,0],
            features_standardized[:,1], c=color)

# Создать гиперплоскость
w = svc.coef_[0]
a = -w[0] / w[1]
xx = np.linspace(-2.5, 2.5)
yy = a * xx - (svc.intercept_[0]) / w[1]

# Начертить гиперплоскость
plt.plot(xx, yy)
plt.axis("off")
plt.show()
```

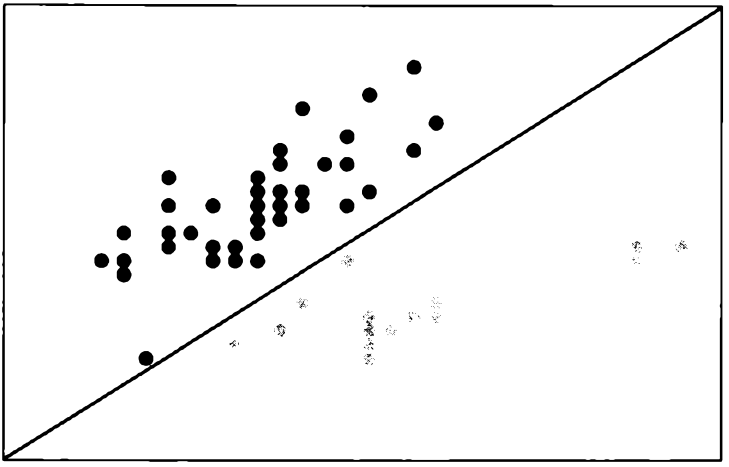


Рис. 17.1

В этой визуализации все наблюдения класса 0 являются черными, а наблюдения класса 1 — светло-серыми. Гиперплоскость — это граница решения, определяющая то, как классифицируются новые наблюдения. В частности, любое наблюдение над прямой будет отнесено к классу 0, а любое наблюдение под прямой — к классу 1. Мы можем доказать это, создав новое наблюдение в левом верхнем углу нашей визуализации, подразумевая, что оно должно быть предсказано как класс 0:

```
# Создать новое наблюдение
new_observation = [[ -2, 3]]

# Предсказать класс нового наблюдения
svc.predict(new_observation)
```

```
array([0])
```

В отношении опорно-векторных классификаторов необходимо отметить несколько моментов. Во-первых, ради визуализации мы ограничили нашу задачу бинарным примером (а именно, только двумя классами); однако опорно-векторные классификаторы могут хорошо работать с многочисленными классами. Во-вторых, как показывает наша визуализация, гиперплоскость по определению линейна (т. е. не изогнута). Для данного примера это приемлемо, потому что данные были линейно разделяемыми, т. е. имелась гиперплоскость, которая могла идеально разделить эти два класса. К сожалению, в реальном мире такое случается редко.

Более типичной является ситуация, когда мы не можем разделить классы идеально. В этих случаях существует баланс между максимизацией промежутка гиперплоскости и минимизацией ошибочной классификации. В опорно-векторном классификаторе последнее контролируется с помощью гиперпараметра C — штрафа, налагаемого на ошибки. C — это параметр ученика SVC и штраф за ошибочное классифицирование точки данных. Когда значение C мало, классификатор получает одобрение на ошибочно классифицированные точки данных (высокое смещение, но

низкая дисперсия). Когда значение C большое, классификатор сильно штрафуется за ошибочно классифицированные данные и поэтому отклоняется назад, чтобы избежать любых ошибочно классифицированных точек данных (низкое смещение, но высокая дисперсия).

В библиотеке scikit-learn значение C определяется параметром `C` и по умолчанию `C=1.0`. Мы должны относиться к C, как к гиперпараметру обучающегося алгоритма, который мы настраиваем, используя методы отбора модели из *главы 12*.

17.2. Обработка линейно неразделимых классов с помощью ядер

Задача

Требуется натренировать опорно-векторный классификатор, но ваши классы линейно неразделимы.

Решение

Натренировать расширение опорно-векторной машины с использованием ядерных функций для создания нелинейных границ решений:

```
# Загрузить библиотеки
from sklearn.svm import SVC
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
import numpy as np

# Задать начальное значение рандомизации
np.random.seed(0)

# Сгенерировать два признака
features = np.random.randn(200, 2)

# Применить вентиль XOR (вам не обязательно знать, что это такое),
# чтобы сгенерировать линейно разделимые классы
target_xor = np.logical_xor(features[:, 0] > 0, features[:, 1] > 0)
target = np.where(target_xor, 0, 1)

# Создать опорно-векторную машину
# с радиально-базисным функциональным ядром (RBF-ядром)
svc = SVC(kernel="rbf", random_state=0, gamma=1, C=1)

# Натренировать классификатор
model = svc.fit(features, target)
```

Обсуждение

Полное объяснение опорных векторов выходит за рамки этой книги. Однако краткое объяснение, вероятно, будет полезным для понимания опорно-векторых машин и ядер. По причинам, которые лучше всего выяснить из других источников, опорно-векторный классификатор может быть представлен как:

$$f(x) = \beta_0 + \sum_{i \in S} \alpha_i K(x_i, x_{i'}),$$

где β_0 — смещение; S — набор всех опорно-векторных наблюдений; α_i — модельные параметры, которые необходимо заучить; $(x_i, x_{i'})$ — пары двух опорно-векторных наблюдений x_i и $x_{i'}$. Самое главное, K — это ядерная функция, которая сравнивает сходство между x_i и $x_{i'}$. Не переживайте, если не разбираетесь в ядерных функциях. Для наших целей просто следует понять, что K, во-первых, определяет тип гиперплоскости, используемой для разделения наших классов, и во-вторых, мы создаем разные гиперплоскости, используя разные ядра. Например, если нам нужна элементарная линейная гиперплоскость, подобная той, что мы создали в рецепте 17.1, то можем использовать линейное ядро:

$$K(x_i, x_{i'}) = \sum_{j=1}^{p} x_{ij} x_{i'j},$$

где p — количество признаков. Однако если нам нужна нелинейная граница решения, то вместо линейного ядра мы подставляем полиномиальное ядро:

$$K(x_i, x_{i'}) = \left(1 + \sum_{j=1}^{p} x_{ij} x_{i'j}\right)^2,$$

где 2 — это степень полиномиальной ядерной функции. В качестве альтернативы мы можем использовать одно из наиболее распространенных ядер в опорно-векторных машинах — ядро радиально-базисной функции (радиально-базисное функциональное ядро):

$$K(x_i, x_{i'}) = \exp\left(-\gamma \sum_{j=1}^{p} (x_{ij} x_{i'j})^2\right),$$

где γ — это гиперпараметр, который должен быть больше нуля. Основной смысл приведенного выше объяснения заключается в том, что если имеются линейно неразделимые данные, то вместо линейного ядра можно подставить альтернативное ядро и создать нелинейную гиперплоскостную границу решения.

Лежащую в основе ядер идею можем понять, выполнив визуализацию простого примера. Приведенная далее функция, основанная на функции Себастьяна Рашки, выводит на график наблюдения и гиперплоскость границы решения двумерного пространства. Вам не обязательно понимать, как работает эта функция; она включена сюда, чтобы вы могли поэкспериментировать самостоятельно:

```
# Вывести на график наблюдения и гиперплоскость границы решения
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt

def plot_decision_regions(X, y, classifier):
    cmap = ListedColormap(("red", "blue"))
    xx1, xx2 = np.meshgrid(np.arange(-3, 3, 0.02), np.arange(-3, 3, 0.02))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.1, cmap=cmap)

    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                    alpha=0.8, c=cmap(idx),
                    marker="+", label=cl)
```

В нашем решении мы имеем данные, содержащие два признака (т. е. две размерности) и вектор целей с классом каждого наблюдения. Важно отметить, что классы назначены так, что они линейно неразделимы. То есть нет прямой линии, которую мы могли бы провести, чтобы разделить два класса. Сначала создадим классификатор опорно-векторной машины с линейным ядром:

```
# Создать опорно-векторный классификатор с линейным ядром
svc_linear = SVC(kernel="linear", random_state=0, C=1)

# Натренировать модель
svc_linear.fit(features, target)
```

```
SVC(C=1, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto', kernel='linear',
  max_iter=-1, probability=False, random_state=0, shrinking=True,
  tol=0.001, verbose=False)
```

Далее, поскольку у нас только два признака, мы работаем в двумерном пространстве и можем визуализировать наблюдения, их классы и линейную гиперплоскость нашей модели (рис. 17.2):

```
# Вывести на график наблюдения и гиперплоскость
plot_decision_regions(features, target, classifier=svc_linear)
plt.axis("off")
plt.show()
```

Как мы видим, линейная гиперплоскость очень плохо справилась с разделением двух классов! Теперь давайте вместо линейного ядра подставим радиально-базисное функциональное ядро и используем его для того, чтобы натренировать новую модель:

```
# Создать опорно-векторную машину
# с радиально-базисным функциональным ядром (RBF-ядром)
svc = SVC(kernel="rbf", random_state=0, gamma=1, C=1)

# Натренировать классификатор
model = svc.fit(features, target)
```

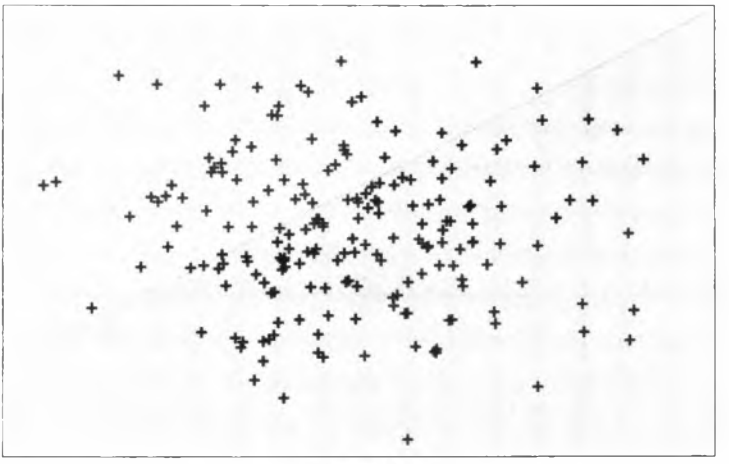


Рис. 17.2

А затем визуализируем наблюдения и гиперплоскость (рис. 17.3):

```
# Вывести на график наблюдения и гиперплоскость
plot_decision_regions(features, target, classifier=svc)
plt.axis("off")
plt.show()
```

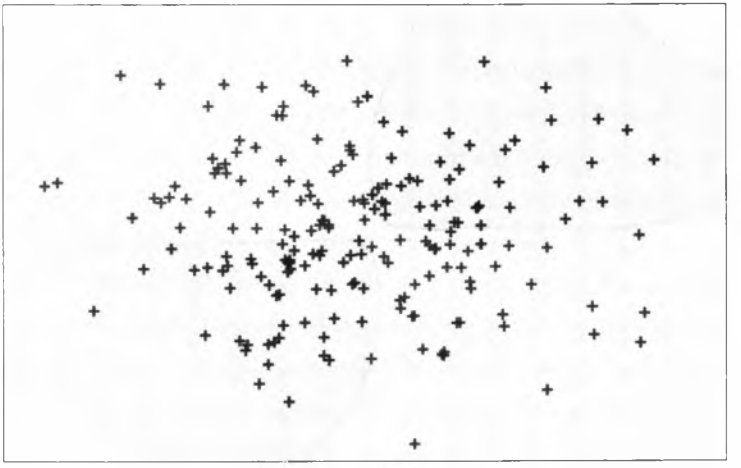


Рис. 17.3

Применив радиально-базисное функциональное ядро, мы смогли создать границу решения, способную справиться с разделением двух классов гораздо лучше, чем линейное ядро. Этим обусловлено использование ядер в опорно-векторных машинах.

В библиотеке scikit-learn можно выбрать требуемое ядро с помощью параметра `kernel`. После того как мы выберем ядро, нам нужно указать соответствующие параметры ядра, такие как значение *d* (используя параметр степени `degree`) в полино-

миальных ядрах и γ (используя параметр `gamma`) в радиально-базисных функциональных ядрах. Нам также нужно задать штрафной параметр `C`. Во время тренировки модели в большинстве случаев все они должны рассматриваться как гиперпараметры, при этом, чтобы определить комбинацию их значений, которая производит модель с наилучшей результативностью, мы должны использовать методы отбора модели.

17.3. Создание предсказанных вероятностей

Задача

Требуется узнать предсказанные вероятности класса для наблюдения.

Решение

При использовании класса `SVC` библиотеки scikit-learn установить `probability=True`, натренировать модель, а затем для просмотра откалиброванных вероятностей применить метод `predict_probe`:

```
# Загрузить библиотеки
from sklearn.svm import SVC
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
import numpy as np

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Стандартизировать признаки
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# Создать объект опорно-векторного классификатора
svc = SVC(kernel="linear", probability=True, random_state=0)

# Натренировать классификатор
model = svc.fit(features_standardized, target)

# Создать наблюдение
new_observation = [[.4, .4, .4, .4]]

# Взглянуть на предсказанные вероятности
model.predict_proba(new_observation)

array([[0.00579881, 0.96697354, 0.02722765]])
```

Обсуждение

Многие алгоритмы контролируемого самообучения, которые мы рассмотрели, используют для предсказания классов оценки вероятности. Например, в k ближайших соседях k классов соседей наблюдения рассматривались как голоса с целью создать вероятность того, что наблюдение принадлежит конкретному классу. Затем предсказывается класс с наибольшей вероятностью. Использование гиперплоскости опорно-векторного классификатора (SVC) для создания областей принятия решений, естественно, не выводит оценку вероятности того, что наблюдение является членом определенного класса. Однако калиброванные вероятности классов можно фактически вывести с несколькими оговорками. В SVC с двумя классами может использоваться шкалирование Платта[1], в котором сначала тренируется SVC, а затем тренируется отдельная перекрестно-проверочная логистическая регрессия для отображения выходных данных SVC в вероятности:

$$P(y=1|x)=\frac{1}{1+e^{\mathbf{a}\cdot f(x)+\mathbf{b}}},$$

где $\mathbf{a}$ и $\mathbf{b}$ — параметрические векторы; f — расстояние со знаком i-го наблюдения от гиперплоскости. Если имеется более двух классов, то применяется расширение шкалирования Платта.

С практической точки зрения создание предсказанных вероятностей имеет две основные проблемы. Во-первых, поскольку мы тренируем вторую модель с перекрестной проверкой, генерирование предсказанных вероятностей может значительно увеличить время, необходимое для того, чтобы натренировать нашу модель. Во-вторых, поскольку предсказанные вероятности создаются с помощью перекрестной проверки, они не всегда могут совпадать с предсказанными классами. То есть, наблюдение может быть предсказано как класс 1, но с предсказанной вероятностью меньше 0.5 и принадлежностью к классу 1.

В библиотеке scikit-learn предсказанные вероятности должны генерироваться, когда модель тренируется. Это можно сделать, присвоив параметру `probability` объекта `SVC` значение `True`. После того как модель натренирована, мы можем вывести оценочные вероятности для каждого класса, используя метод `predict_proba`.

17.4. Идентификация опорных векторов

Задача

Требуется определить, какие наблюдения являются опорными векторами гиперплоскости решения.

---

[1] Шкалирование, или калибровка, Платта — это способ преобразования выходов из классификационной модели в распределение вероятностей на классах (см. **https://en.wikipedia.org/wiki/Platt_scaling**). — *Прим. перев.*

Решение

Натренировать модель, а затем использовать атрибут `support_vectors_`:

```
# Загрузить библиотеки
from sklearn.svm import SVC
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
import numpy as np

# Загрузить данные только с двумя классами
iris = datasets.load_iris()
features = iris.data[:100,:]
target = iris.target[:100]

# Стандартизировать признаки
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# Создать объект опорно-векторного классификатора
svc = SVC(kernel="linear", random_state=0)

# Натренировать классификатор
model = svc.fit(features_standardized, target)

# Взглянуть на опорные векторы
model.support_vectors_
```

```
array([[-0.5810659 ,  0.43490123, -0.80621461, -0.50581312],
       [-1.52079513, -1.67626978, -1.08374115, -0.8607697 ],
       [-0.89430898, -1.46515268,  0.30389157,  0.38157832],
       [-0.5810659 , -1.25403558,  0.09574666,  0.55905661]])
```

Обсуждение

Опорно-векторные машины получили свое название благодаря тому, что гиперплоскость определяется относительно небольшим количеством наблюдений, называемых *опорными векторами*. В интуитивном плане можно представить, что эти опорные векторы "несут в себе" гиперплоскость. Поэтому эти опорные векторы очень важны для нашей модели. Например, если мы удалим из данных наблюдение, которое не является опорным вектором, модель не изменится; однако, если мы удалим опорный вектор, гиперплоскость не будет иметь максимального промежутка.

После того как мы натренировали опорно-векторный классификатор, библиотека scikit-learn предлагает несколько вариантов идентификации опорного вектора. В нашем решении для вывода признаков фактических наблюдений четырех опорных векторов в нашей модели мы использовали атрибут `support_vectors_`. В качест-

ве альтернативы можно взглянуть на индексы опорных векторов, используя атрибут `support_`:

```
model.support_
```

```
array([23, 41, 57, 98])
```

Наконец, чтобы найти количество опорных векторов, принадлежащих каждому классу, можно использовать атрибут `n_support_`:

```
model.n_support_
```

```
array([2, 2])
```

17.5. Обработка несбалансированных классов

Задача

Требуется натренировать опорно-векторный классификатор в присутствии несбалансированных классов.

Решение

Увеличить штраф за ошибочное классифицирование меньшего по количеству класса с помощью параметра `class_weight`:

```
# Загрузить библиотеки
from sklearn.svm import SVC
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
import numpy as np

# Загрузить данные только с двумя классами
iris = datasets.load_iris()
features = iris.data[:100,:]
target = iris.target[:100]

# Сделать класс сильно несбалансированным, удалив первые 40 наблюдений
features = features[40:,:]
target = target[40:]

# Создать вектор целей, указав класс 0 либо 1
target = np.where((target == 0), 0, 1)

# Стандартизировать признаки
scaler = StandardScaler()
features_standardized = scaler.fit_transform(features)

# Создать опорно-векторный классификатор
svc = SVC(kernel="linear", class_weight="balanced", C=1.0, random_state=0)
```

```
# Натренировать классификатор
model = svc.fit(features_standardized, target)
```

Обсуждение

В опорно-векторных машинах C — это гиперпараметр, определяющий штраф за ошибочное классифицирование наблюдения. Одним из методов обработки несбалансированных классов в опорно-векторных машинах является взвешивание гиперпараметра C в зависимости от классов таким образом, что:

$$C_j = C \cdot w_j,$$

где C — штраф за ошибочную классификацию; w_j — вес, обратно пропорциональный частоте класса j; C_j — значение C для класса j. Общая идея взвешивания состоит в том, чтобы увеличить штраф за ошибочное классифицирование миноритарных классов, чтобы они не были "подавлены" мажоритарным классом.

В библиотеке scikit-learn при использовании объекта `SVC` можно устанавливать значения для C_j автоматически, задав параметр `class_weight='balanced'`. Аргумент `balanced` автоматически взвешивает подобные классы таким образом, что:

$$w_j = \frac{n}{kn_j},$$

где w_j — вес класса j; n — количество наблюдений; n_j — количество наблюдений класса j; k — общее количество классов.

Наивный Байес

Введение

Теорема Байеса является главным методом для понимания вероятности некоторого события $P(A|B)$ при наличии некой новой информации, $P(B|A)$ и априорной субъективной оценки вероятности события $P(A)$:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}.$$

Популярность байесова метода резко возросла в последнее десятилетие, все больше и больше конкурируя с традиционными фриквентистскими приложениями в академических, правительственных и деловых кругах. В машинном самообучении одно из применений теоремы Байеса к классификации проявляется в виде наивного байесова классификатора. Наивные байесовы классификаторы объединяют в общий классификатор ряд желательных в практическом машинном самообучении качеств. К ним относятся:

- интуитивно понятный подход;
- возможность работы с малыми данными;
- низкие затраты на тренировку и предсказание;
- часто надежные результаты в разнообразных условиях.

В частности, наивный байесов классификатор основан на следующей формуле:

$$P\left(y|x_1, ..., x_j\right) = \frac{P\left(x_1, ..., x_j|y\right)P(y)}{P\left(x_1, ..., x_j\right)},$$

где:

- $P\left(y|x_1, ..., x_j\right)$ называется *апостериорным значением* и является вероятностью того, что наблюдение принадлежит классу y при условии, что это наблюдение имеет значения j признаков $x_1, ..., x_j$;
- $P\left(x_1, ..., x_j|y\right)$ называется *правдоподобием* и является *правдоподобной оценкой вероятности*, что наблюдение имеет значения признаков $x_1, ..., x_j$ при условии, что дан их класс y;

- $P(y)$ называется *априорной вероятностью* и является нашей субъективной оценкой вероятности класса y перед рассмотрением данных;
- $P(x_1, ..., x_j)$ называется *предельной вероятностью*[1].

В наивном Байесе мы сравниваем апостериорные значения наблюдения для каждого возможного класса. В частности, поскольку во всех сравнениях предельная вероятность постоянна, мы сравниваем числители апостериорного значения для каждого класса. Для каждого наблюдения класс с наибольшим апостериорным числителем становится предсказанным классом $\hat{y}$.

В отношении наивных байесовых классификаторов следует отметить два важных момента. Во-первых, для каждого признака в данных мы должны принять статистическое распределение правдоподобно оцененных вероятностей $P(x_j \mid y)$. Общепринятыми распределениями являются нормальное (гауссово), полиномиальное и бернуллиево распределения. Выбор распределения нередко определяется природой признаков (непрерывных, бинарных и т. д.). Во-вторых, название "наивный Байес" обусловлено тем, что мы допускаем независимость каждого признака и результирующей правдоподобной оценки его вероятности. Это "наивное" допущение часто неверно, но на практике мало чем мешает созданию высококачественных классификаторов.

В этой главе мы рассмотрим использование библиотеки scikit-learn для тренировки трех типов наивных байесовых классификаторов с использованием трех различных распределений правдоподобно оцениваемых вероятностей.

18.1. Тренировка классификатора для непрерывных признаков

Задача

Даны только непрерывные признаки, и требуется натренировать наивный байесов классификатор.

Решение

Использовать гауссов наивный байесов классификатор в библиотеке scikit-learn:

```
# Загрузить библиотеки
from sklearn import datasets
from sklearn.naive_bayes import GaussianNB

# Загрузить данные
iris = datasets.load_iris()
```

[1] В оригинале marginal probability — маржинальная вероятность. В данной формуле она также называется полной вероятностью. — *Прим. перев.*

```
features = iris.data
target = iris.target

# Создать объект гауссова наивного Байеса
classifer = GaussianNB()

# Натренировать модель
model = classifer.fit(features, target)
```

Обсуждение

Наиболее распространенным типом наивного байесова классификатора является гауссов наивный Байес. В гауссовом наивном Байесе мы принимаем допущение, что правдоподобие признаковых значений x при условии, что наблюдение принадлежит классу y, подчиняется нормальному распределению:

$$P(x_j \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)}{2\sigma_y^2}\right),$$

где σ_y^2 и μ_y — дисперсия и средние значения признака x_j для класса y. Из-за принятого допущения о нормальном распределении гауссов наивный Байес лучше всего использовать в случаях, когда все наши признаки непрерывны.

В библиотеке scikit-learn мы тренируем гауссов наивный Байес, как и любую другую модель, используя метод `fit`, и, в свою очередь, можем делать предсказания о классе наблюдения:

```
# Создать новое наблюдение
new_observation = [[ 4, 4, 4, 0.4]]

# Предсказать класс
model.predict(new_observation)

array([1])
```

Один из интересных аспектов наивных байесовых классификаторов заключается в том, что они позволяют назначать априорную субъективную оценку в отношении вероятностей целевых классов. Это можно сделать с помощью параметра `priors` класса `GaussianNB`; этот параметр принимает список вероятностей, присваиваемых каждому классу вектора целей:

```
# Создать объект гауссова наивного Байеса
# с априорными вероятностями для каждого класса
clf = GaussianNB(priors=[0.25, 0.25, 0.5])

# Натренировать модель
model = classifer.fit(features, target)
```

Если мы не добавляем никаких аргументов в параметр `priors`, то априорные оценки настраиваются на основе данных.

Наконец, обратите внимание, что сырые предсказанные вероятности из гауссова наивного Байеса (выведенные с помощью метода `predict_proba`) не калиброваны, т. е. им не следует верить. Если требуется создать полезные предсказанные вероятности, нам нужно их откалибровать с помощью изотонической регрессии или родственного метода.

Дополнительные материалы для чтения

- "Как работает наивный байесов классификатор в машинном самообучении", портал науки о данных для начинающих Dataaspirant (**http://bit.ly/2F6trtt**).

18.2. Тренировка классификатора для дискретных и счетных признаков

Задача

Даны дискретные или счетные данные, и требуется натренировать наивный байесов классификатор.

Решение

Использовать полиномиальный наивный байесов классификатор:

```
# Загрузить библиотеки
import numpy as np
from sklearn.naive_bayes import MultinomialNB
from sklearn.feature_extraction.text import CountVectorizer

# Создать текст
text_data = np.array(['Бразилия - моя любовь. Бразилия!',
                      'Бразилия - лучше',
                      'Германия бьет обоих'])

# Создать мешок слов
count = CountVectorizer()
bag_of_words = count.fit_transform(text_data)

# Создать матрицу признаков
features = bag_of_words.toarray()

# Создать вектор целей
target = np.array([0,0,1])
```

```
# Создать объект полиномиального наивного байесова классификатора
# с априорными вероятностями каждого класса
classifer = MultinomialNB(class_prior=[0.25, 0.5])

# Натренировать модель
model = classifer.fit(features, target)
```

Обсуждение

Полиномиальный наивный Байес работает аналогично гауссовому наивному Байесу, только здесь признаки подчиняются полиномиальному распределению. На практике это означает, что данный классификатор обычно используется, когда имеются дискретные данные (например, рейтинги фильмов в диапазоне от 1 до 5). Одним из наиболее распространенных применений полиномиальных наивных байесовых классификаторов является классификация текста с использованием подходов на основе мешков слов или статистических мер tf-idf (см. рецепты 6.9 и 6.10).

В нашем решении мы создали игрушечный набор текстовых данных из трех наблюдений и преобразовали текстовые строки в матрицу признаков и сопутствующий вектор целей по методу мешка слов. Затем мы использовали класс `MultinomialNB` для того, чтобы натренировать модель, при этом определив априорные вероятности для двух классов (пробразильский и прогерманский).

Класс `MultinonialNB` работает аналогично классу `GaussianNB`; модели тренируются с использованием метода `fit`, а наблюдения можно предсказать с помощью метода `predict`:

```
# Создать новое наблюдение
new_observation = [[0, 0, 0, 1, 0, 1, 0]]

# Предсказать класс нового наблюдения
model.predict(new_observation)
```

```
array([0])
```

Если параметр `class_prior` не задан, то априорные вероятности заучиваются на основе данных. Однако если мы хотим, чтобы в качестве априорного распределения использовалось равномерное, то можно задать `fit_prior=False`.

Наконец, класс `MultinomialNB` содержит гиперпараметр аддитивного сглаживания `alpha`, который должен быть настроен. Принятое значение по умолчанию равняется `1.0`, при этом значение `0.0` означает отсутствие сглаживания.

18.3. Тренировка наивного байесова классификатора для бинарных признаков

Задача

Имеются двоичные признаковые данные, и требуется натренировать наивный байесов классификатор.

Решение

Использовать бернуллиев наивный байесов классификатор:

```
# Загрузить библиотеки
import numpy as np
from sklearn.naive_bayes import BernoulliNB

# Создать три бинарных признака
features = np.random.randint(2, size=(100, 3))

# Создать вектор бинарных целей
target = np.random.randint(2, size=(100, 1)).ravel()

# Создать объект бернуллиева наивного Байеса
# с априорными вероятностями каждого класса
classifer = BernoulliNB(class_prior=[0.25, 0.5])

# Натренировать модель
model = classifer.fit(features, target)
```

Обсуждение

Бернуллиев наивный байесов классификатор принимает допущение, что все наши признаки являются бинарными, т. е. принимают только два значения (например, номинальный категориальный признак, который был представлен в кодировке с одним активным состоянием). Как и его полиномиальный двоюродный брат, бернуллиев наивный Байес часто используется в классификации текста, когда наша матрица признаков — это просто присутствие или отсутствие слова в документе. Кроме того, как и класс `MultinomialNB`, класс `BernoulliNB` имеет гиперпараметр аддитивного сглаживания `alpha`, который требуется настроить с помощью методов отбора модели. Наконец, если необходимо использовать априорные вероятности, то можно применить параметр `class_prior` со списком, содержащим априорные вероятности для каждого класса. Если требуется указать равномерную априорную вероятность, то можно указать `fit_prior=False`:

```
model_uniform_prior = BernoulliNB(class_prior=None, fit_prior=True)
```

18.4. Калибровка предсказанных вероятностей

Задача

Требуется откалибровать предсказанные вероятности из наивных байесовых классификаторов, чтобы они были интерпретируемыми.

Решение

Использовать класс `CalibratedClassifierCV`:

```
# Загрузить библиотеки
from sklearn import datasets
from sklearn.naive_bayes import GaussianNB
from sklearn.calibration import CalibratedClassifierCV

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать объект гауссова наивного Байеса
classifer = GaussianNB()

# Создать откалиброванную перекрестную проверку
# с сигмоидальной калибровкой
classifer_sigmoid = CalibratedClassifierCV(classifer, cv=2,
                                           method='sigmoid')

# Откалибровать вероятности
classifer_sigmoid.fit(features, target)

# Создать новое наблюдение
new_observation = [[ 2.6, 2.6, 2.6, 0.4]]

# Взглянуть на откалиброванные вероятности
classifer_sigmoid.predict_proba(new_observation)

array([[0.31859969, 0.63663466, 0.04476565]])
```

Обсуждение

Вероятности классов являются общепринятой и полезной частью машинно-обучающихся моделей. В библиотеке scikit-learn большинство обучающихся алгоритмов позволяют нам видеть предсказанные вероятности принадлежности классу с помощью метода `predict_proba`. Это может быть очень полезно, когда, например, требуется предсказать только конкретный класс, если модель предсказывает вероятность того, что этот класс превышает 90%. Однако некоторые модели, включая наивные байесовы классификаторы, выводят вероятности, не основанные на реальном мире. То есть, метод `predict_proba` может предсказать, что наблюдение имеет шанс быть конкретным классом, составляющий 0.70, когда в реальности он равняется 0.10 или 0.99. В частности, в то время как в наивном Байесе ранжирование предсказанных вероятностей для разных целевых классов соответствует действительности, сырые предсказанные вероятности имеют тенденцию принимать предельные значения, близкие к 0 и 1.

Для того чтобы получить содержательные предсказанные вероятности, нам нужно провести так называемую калибровку. Для создания хорошо откалиброванных предсказанных вероятностей в библиотеке scikit-learn можно применять класс `CalibratedClassifierCV` с использованием *k*-блочной перекрестной проверки. В классе `CalibratedClassifierCV` тренировочные наборы применяются для тренировки модели, а тестовый набор используется для калибровки предсказанных вероятностей. Возвращаемые предсказанные вероятности представляют собой среднее из *k* блоков.

С помощью нашего решения мы можем увидеть разницу между сырыми и хорошо откалиброванными предсказанными вероятностями. В нашем решении мы создали гауссов наивный байесов классификатор. Если натренировать этот классификатор, а затем предсказать вероятности классов для нового наблюдения, то мы увидим абсолютно предельные оценки вероятностей:

```
# Натренировать гауссов наивный Байес и
# затем предсказать вероятности классов
classifer.fit(features, target).predict_proba(new_observation)
```

```
array([[2.58229098e-04, 9.99741447e-01, 3.23523643e-07]])
```

Однако после калибровки предсказанных вероятностей (что мы и сделали в нашем решении) мы получим совсем другие результаты:

```
# Взглянуть на откалиброванные вероятности
classifer_sigmoid.predict_proba(new_observation)
```

```
array([[0.31859969, 0.63663466, 0.04476565]])
```

Класс `CalibratedClassifierCV` предлагает два метода калибровки — сигмоидную модель Платта и изотоническую регрессию, определяемые параметром `method`. Не вдаваясь в специфику, все же отметим, что поскольку изотоническая регрессия является непараметрической, она имеет тенденцию к переподгонке, когда размеры выборки очень малы (например, 100 наблюдений). В нашем решении мы использовали набор данных цветков ириса со 150 наблюдениями и поэтому выбрали сигмоидную модель Платта.

Кластеризация

Введение

В большей части этой книги мы рассматривали контролируемое машинное самообучение — задачи, в которых имеется доступ как к признакам, так и к цели. Это, к сожалению, не всегда так. Часто мы сталкиваемся с ситуациями, когда нам известны только признаки. Например, представьте, что даны записи о продажах из продуктового магазина, и требуется разбить продажи по признаку, является ли покупатель членом дисконтного клуба или нет. Это было бы невозможно при использовании контролируемого самообучения, потому что у нас нет цели для проведения тренировки и оценивания наших моделей. Однако есть и другой вариант: неконтролируемое самообучение. Если поведение членов и нечленов дисконтного клуба в продуктовом магазине фактически разрозненное, то разница в поведении между двумя членами будет меньше, чем разница в поведении между покупателями-членами и нечленами. Иными словами, будут две группы наблюдений.

Цель кластеризующих алгоритмов состоит в том, чтобы идентифицировать эти скрытые группы наблюдений, которые при приемлемом исполнении позволят предсказывать класс наблюдений даже без вектора целей. Существует множество кластеризующих алгоритмов, и они имеют широкий спектр подходов к идентификации кластеров в данных. В этой главе мы рассмотрим подборку кластеризующих алгоритмов с использованием библиотеки scikit-learn и то, как их применять на практике.

19.1. Кластеризация с помощью *k* средних

Задача

Требуется сгруппировать наблюдения в *k* групп.

Решение

Использовать кластеризацию методом *k* средних:

```
# Загрузить библиотеки
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
```

```
# Загрузить данные
iris = datasets.load_iris()
features = iris.data

# Стандартизировать признаки
scaler = StandardScaler()
features_std = scaler.fit_transform(features)

# Создать объект k средних
cluster = KMeans(n_clusters=3, random_state=0, n_jobs=-1)

# Натренировать модель
model = cluster.fit(features_std)
```

Обсуждение

Кластеризация методом *k* средних является одним из наиболее распространенных методов кластеризации. В этом методе алгоритм пытается сгруппировать наблюдения в *k* групп, причем каждая группа имеет примерно равную дисперсию. Количество групп *k* задается пользователем в качестве гиперпараметра. В частности, в алгоритме *k* средних:

1. В случайных позициях создается *k* "центральных" точек, по одной на кластер.
2. Для каждого наблюдения:
   - вычисляется расстояние между каждым наблюдением и *k* центральными точками;
   - наблюдение назначается кластеру ближайшей центральной точки.
3. Центральные точки передвигаются в средние значения (т. е. в центры) своих соответствующих кластеров.
4. Шаги 2 и 3 повторяются до тех пор, пока ни одно из наблюдений не изменит свою принадлежность кластеру.

На этом этапе алгоритм считается достигшим схождения и останавливается.

Относительно алгоритма *k* средних важно отметить три момента. Во-первых, кластеризация методом *k* средних исходит из того, что кластеры имеют выпуклую форму (например, форму круга или сферы). Во-вторых, все признаки шкалированы одинаково. В нашем решении для того чтобы соблюсти это допущение, мы стандартизировали признаки. В-третьих, группы сбалансированы (т. е. имеют примерно одинаковое количество наблюдений). Если мы подозреваем, что не сможем выполнить эти допущения, мы можем попробовать другие подходы к кластеризации.

В библиотеке scikit-learn кластеризация методом *k* средних реализована в классе `KMeans`. Наиболее важным параметром является `n_clusters`, который задает количество кластеров *k*. В некоторых ситуациях характер данных будет определять значение *k* (например, данные об учениках школы будут иметь один кластер на класс), но зачастую количество кластеров мы не знаем. В этих случаях требуется выбрать *k*

на основе некоторых критериев. Например, силуэтные коэффициенты (см. рецепт 11.9) измеряют подобие внутри кластеров по сравнению с подобием между кластерами. Кроме того, поскольку кластеризация методом k средних является вычислительно дорогостоящей, можно воспользоваться всеми ядрами на нашем компьютере. Мы можем сделать это, установив параметр `n_jobs=-1`.

В нашем решении мы пошли на небольшой обман и воспользовались данными цветков ириса, о которых мы знаем, что они имеют три класса. Поэтому мы устанавливаем $k = 3$. Для просмотра предсказанных классов каждого наблюдения можно использовать атрибут `labels_`:

```
# Взглянуть на предсказанный класс
model.labels_
```

```
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, 0, 0, 0, 2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 2, 0,
       2, 2, 2, 2, 0, 2, 2, 2, 2, 0, 0, 0, 2, 2, 2, 2, 2, 2, 0, 0, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 2, 0, 0, 0, 0, 2, 0, 0, 0,
       0, 0, 0, 2, 2, 0, 0, 0, 0, 2, 0, 2, 0, 2, 0, 0, 2, 0, 0, 0, 0, 0,
       0, 2, 2, 0, 0, 0, 2, 0, 0, 0, 2, 0, 0, 0, 2, 0, 0, 2])
```

Если мы сравним этот результат с истинным классом наблюдений, то увидим, что, несмотря на разницу в метках классов (т. е. `1`, `2` и `3`), алгоритм k средних справился с работой достаточно хорошо:

```
# Взглянуть на истинный класс
iris.target
```

```
array([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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 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, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

Однако, как вы можете себе представить, производительность алгоритма k средних значительно падает, и даже критически, если мы выберем неправильное количество кластеров.

Наконец, как и в случае с другими результатами библиотеки scikit-learn, мы можем использовать натренированный кластер для предсказания значения новых наблюдений:

```
# Создать новое наблюдение
new_observation = [[0.8, 0.8, 0.8, 0.8]]

# Предсказать кластер наблюдения
model.predict(new_observation)
```

```
array([0])
```

Согласно предсказаниям, наблюдение принадлежит тому кластеру, центральная точка которого находится ближе всего. Для того чтобы увидеть эти центральные точки, мы даже можем воспользоваться атрибутом `cluster_centers_`:

```
# Взглянуть на центры кластеров
model.cluster_centers_

array([[ 1.13597027,  0.09659843,  0.996271  ,  1.01717187],
       [-1.01457897,  0.84230679, -1.30487835, -1.25512862],
       [-0.05021989, -0.88029181,  0.34753171,  0.28206327]])
```

Дополнительные материалы для чтения

- "Введение в кластеризацию методом *k* средних", DataScience.com (**http://bit.ly/2Hjik1f**).

19.2. Ускорение кластеризации методом *k* средних

Задача

Требуется сгруппировать наблюдения в *k* групп, но алгоритм *k* средних занимает слишком много времени.

Решение

Использовать мини-пакетный алгоритм *k* средних:

```
# Загрузить библиотеки
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import MiniBatchKMeans

# Загрузить данные
iris = datasets.load_iris()
features = iris.data

# Стандартизировать признаки
scaler = StandardScaler()
features_std = scaler.fit_transform(features)

# Создать объект k средних
cluster = MiniBatchKMeans(n_clusters=3, random_state=0, batch_size=100)

# Натренировать модель
model = cluster.fit(features_std)
```

Обсуждение

Мини-пакетный алгоритм *k* средних работает аналогично алгоритму *k* средних, описанному в рецепте 19.1. Не вдаваясь в подробности, разница состоит в том, что в мини-пакетном алгоритме *k* средних наиболее вычислительно затратный шаг выполняется только на случайной выборке наблюдений в отличие от всех наблюдений, используемых в обычном алгоритме. Такой подход позволяет значительно сократить время, необходимое алгоритму для нахождения сходимости (т. е. подгонки данных) при небольшой стоимости в качестве.

Класс `MiniBatchKMeans` работает аналогично классу `KMeans`, но с одним существенным отличием: это параметр размера пакета `batch_size`. Параметр `batch_size` управляет количеством случайно отбираемых наблюдений в каждом пакете. Чем больше размер пакета, тем более дорогостоящим является тренировочный процесс.

19.3. Кластеризация методом сдвига к среднему

Задача

Требуется сгруппировать наблюдения без учета количества кластеров или их формы.

Решение

Использовать кластеризацию методом сдвига к среднему `MeanShift`:

```
# Загрузить библиотеки
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import MeanShift

# Загрузить данные
iris = datasets.load_iris()
features = iris.data

# Стандартизировать признаки
scaler = StandardScaler()
features_std = scaler.fit_transform(features)

# Создать объект кластеризации методом сдвига к среднему
cluster = MeanShift(n_jobs=-1)

# Натренировать модель
model = cluster.fit(features_std)
```

Обсуждение

Один из недостатков кластеризации методом *k* средних, который мы обсуждали выше, заключается в том, что нам нужно устанавливать количество кластеров *k* априорно до начала тренировки, и что этот метод делает допущения о форме кластеров. Одним из кластеризующих алгоритмов, в котором эти ограничения отсутствуют, называется *сдвигом к среднему*.

В алгоритме сдвига к среднему заключена простая идея, которую, правда, несколько трудно объяснить. Поэтому наилучшим подходом может быть аналогия. Представьте себе очень туманное футбольное поле (т. е. двухмерное пространство), на котором находятся 100 человек (т. е. наши наблюдения). Из-за тумана человек может видеть только на небольшое расстояние. Каждую минуту каждый человек оглядывается и делает шаг в сторону большинства людей, которых он видит. С течением времени люди начинают собираться, поскольку они неоднократно предпринимают шаги в сторону все более крупных скоплений людей. В конечном счете получаются кластеры людей в пределах поля. Люди назначаются кластерам, в которых они оказываются[1].

Фактическая реализация алгоритма сдвига к среднему в scikit-learn (класс `MeanShift`) сложнее, но следует той же основной логике. Класс `MeanShift` имеет два важных параметра, о которых мы должны знать. Во-первых, ширина полосы `bandwidth` задает радиус области (т. е. ядро), который используется наблюдением для определения направления сдвига. По нашей аналогии ширина полосы будет тем, как далеко человек может видеть сквозь туман. Мы можем задать этот параметр вручную, но по умолчанию разумная ширина полосы вычисляется автоматически (при значительном увеличении вычислительных затрат). Во-вторых, иногда в алгоритме сдвига к среднему внутри ядра наблюдения нет других наблюдений. То есть человек на нашем футбольном поле не может видеть ни одного другого человека. По умолчанию класс `MeanShift` присваивает все эти "сиротские" наблюдения ядру ближайшего наблюдения. Однако если требуется исключить этих сирот, то можно установить параметр `cluster_all=False`, при котором сиротским наблюдениям присваивается метка `-1`.

Дополнительные материалы для чтения

- "Кластеризующий алгоритм на основе сдвига к среднему", ресурс о машинном самообучении и науке о данных EFAVDB (**https://bit.ly/2ILVQdI**).

[1] Еще одно объяснение — гравитационное. Области высокой плотности соответствуют локальным максимумам. Для того чтобы отыскать эти локальные максимумы, алгоритм работает, позволяя точкам притягивать друг друга, через то, что можно рассматривать, как "гравитационную" силу малой дальности. Позволяя точкам тяготеть к областям более высокой плотности, алгоритм в конечном счете получает локальные максимумы — кластеры точек (см. **http://efavdb.com/mean-shift/**). — *Прим. перев.*

19.4. Кластеризация методом DBSCAN

Задача

Требуется сгруппировать наблюдения в кластеры высокой плотности.

Решение

Использовать кластеризацию методом DBSCAN (Density-Based Spatial Clustering of Applications with Noise, плотностной алгоритм пространственной кластеризации с присутствием шума):

```
# Загрузить библиотеки
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import DBSCAN

# Загрузить данные
iris = datasets.load_iris()
features = iris.data

# Стандартизировать признаки
scaler = StandardScaler()
features_std = scaler.fit_transform(features)

# Создать объект плотностной кластеризации dbscan
cluster = DBSCAN(n_jobs=-1)

# Натренировать модель
model = cluster.fit(features_std)
```

Обсуждение

Алгоритм DBSCAN руководствуется идеей о том, что кластеры будут областями, где много наблюдений плотно упакованы вместе, и не делает никаких допущений о форме кластера. В частности, в алгоритме DBSCAN:

1. Выбирается случайное наблюдение x_i.
2. Если x_i имеет минимальное число близких соседей, то мы рассматриваем его как часть кластера.
3. Шаг 2 повторяется рекурсивно для всех соседей x_i, затем для соседа соседа и т. д. Это основные наблюдения кластера.
4. После завершения шага 3 выбирается новая случайная точка (т. е. перезапуск шага 1).

Когда все это будет завершено, мы получим набор ключевых наблюдений для ряда кластеров. Наконец, любое наблюдение, близкое к кластеру, но не являющееся ключевым образцом, считается частью кластера, в то время как любое наблюдение, не близкое к кластеру, помечается как выброс.

DBSCAN имеет три основных устанавливаемых параметра:

- `eps` — максимальное расстояние от наблюдения, чтобы считать другое наблюдение его соседом;
- `min_samples` — минимальное число наблюдений, находящихся на расстоянии менее `eps` от наблюдения, для того чтобы его можно было считать ключевым наблюдением;
- `metric` — метрический показатель расстояния, используемый параметром `eps`, например `minkowski` или `euclidean` (обратите внимание, что если используется расстояние Минковского, то может быть использован параметр `p` для установки мощности метрического показателя Минковского).

Если мы посмотрим на кластеры в наших тренировочных данных, то увидим, что были идентифицированы два кластера, `0` и `1`, в то время как наблюдения-выбросы помечены `-1`:

```
# Показать принадлежность к кластерам
model.labels_
```

```
array([ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, -1, -1,  0,
        0,  0,  0,  0,  0, -1,  0,  0,  0,  0,  0,  0,  0,  0,  0, -1, -1,
        0,  0,  0,  0,  0,  0,  0, -1,  0,  0,  0,  0,  0,  0,  0,  0,  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,  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],
      dtype=int64)
```

Дополнительные материалы для чтения

- "Плотностная кластеризация DBSCAN", Википедия (**https://bit.ly/2L8Hxy7**).

19.5. Кластеризация методом иерархического слияния

Задача

Требуется сгруппировать наблюдения, используя иерархию кластеров.

Решение

Использовать агломеративную кластеризацию:

```
# Загрузить библиотеки
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import AgglomerativeClustering

# Загрузить данные
iris = datasets.load_iris()
features = iris.data

# Стандартизировать признаки
scaler = StandardScaler()
features_std = scaler.fit_transform(features)

# Создать объект агломеративной кластеризации
cluster = AgglomerativeClustering(n_clusters=3)

# Натренировать модель
model = cluster.fit(features_std)
```

Обсуждение

Агломеративная кластеризация — это мощный и гибкий алгоритм, который выполняет кластеризацию иерархически. В агломеративной кластеризации все наблюдения начинаются как одноэлементные кластеры. Далее кластеры, удовлетворяющие некоторым критериям, объединяются. Этот процесс повторяется, выращивая кластеры до тех пор, пока не будет достигнута некоторая конечная точка. В библиотеке scikit-learn в классе `AgglomerativeClustering` используется параметр связи `linkage` для определения стратегии слияния, которая сводит к минимуму следующее:

- дисперсию объединенных кластеров по методу Уорда (`ward`);
- среднее расстояние между наблюдениями от пар кластеров по методу средней связи (`average`);
- максимальное расстояние между наблюдениями от пар кластеров по методу полной связи (`complete`).

Два других параметра полезно знать тоже. Во-первых, параметр близости `affinity` определяет метрический показатель расстояния, используемый для параметра `linkage` (`minkowski`, `euclidean` и т. д.). Во-вторых, `n_clusters` задает количество кластеров, которые кластеризующий алгоритм будет пытаться найти. То есть кластеры последовательно объединяются до тех пор, пока количество оставшихся кластеров не будет равняться `n_clusters`.

Как и с другими кластеризующими алгоритмами, которые мы рассмотрели выше, мы можем использовать атрибут `labels_`, чтобы увидеть кластер, которому назначено каждое наблюдение:

```
# Показать принадлежность к кластерам
model.labels_
```

```
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, 2, 1, 1,
       1, 1, 1, 1, 1, 1, 0, 0, 0, 2, 0, 2, 0, 2, 0, 2, 2, 0, 2, 0, 2, 0,
       2, 2, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2, 0, 2, 0, 0, 2,
       2, 2, 2, 0, 2, 2, 2, 2, 2, 0, 2, 2, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 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, 0], dtype=int64)
```

Нейронные сети

Введение

В самом центре нейронных сетей лежит *структурно-функциональная единица*, так называемый *узел*, или *нейрон*. Структурно-функциональная единица принимает один или несколько входов, умножает каждый вход на параметр, так называемый *вес*, суммирует взвешенные входные значения между собой с некоторым значением смещения (обычно 1), а затем передает значение в активационную функцию. Этот вывод затем направляется вперед к другим нейронам, расположенным глубже в нейронной сети (если они существуют).

Нейронные сети *прямого распространения* — так называемые *многослойные персептроны* — являются простейшей искусственной нейронной сетью, используемой в любых реальных условиях. Нейронные сети могут быть визуализированы как ряд связанных слоев, которые образуют сеть, соединяющую значения признаков наблюдения на одном конце и целевое значение (например, класс наблюдения) на другом конце. Термин "прямое распространение" происходит из того факта, что значения признаков наблюдения передаются по сети "вперед", при этом каждый слой последовательно преобразовывает значения признаков с целью, чтобы выходной результат в конце был таким же, как и целевое значение.

В частности, нейронные сети прямого распространения содержат три типа слоев структурно-функциональных единиц. В начале нейронной сети находится входной слой, где каждая единица содержит значение наблюдения для одного признака. Например, если наблюдение имеет 100 признаков, то входной слой имеет 100 узлов. В конце нейронной сети находится выходной слой, который преобразует выходной результат скрытых слоев в значения, полезные для выполнения поставленной задачи. Например, если бы нашей целью являлась бинарная классификация, то мы могли бы использовать выходной слой с единственной структурно-функциональной единицей, которая применяла бы сигмоидальную функцию для шкалирования собственного выходного результата в диапазон от 0 до 1, представляя предсказанную вероятность класса. Между входным и выходным слоями находятся так называемые "скрытые" слои (которые в общем-то не скрыты). Эти скрытые слои последовательно преобразуют значения признаков из входного слоя во что-то, что после обработки выходным слоем напоминает целевой класс. Нейронные сети со многими скрытыми слоями (например, 10, 100, 1000) считаются "глубокими" сетями, и их применение называется *глубоким самообучением*.

Нейронные сети, как правило, создаются с параметрами, которые инициализируются как малые случайные значения из гауссового, или нормального, равномерного

распределения. Когда наблюдение (или чаще заданное количество наблюдений, называемое *пакетом*) подается через сеть, выводимое значение сравнивается с истинным значением наблюдения с помощью функции потери. Этот процесс называется *прямым распространением*. Затем алгоритм проходит через сеть "в обратную сторону", идентифицируя, насколько большим был вклад каждого параметра в ошибку между предсказанными и истинными значениями. Этот процесс называется *обратным распространением*. В каждом параметре оптимизационный алгоритм определяет, насколько каждый вес должен быть скорректирован для улучшения выходного результата.

Нейронные сети учатся, несколько раз повторяя этот процесс прямого распространения значений признаков и обратного распространения ошибки для каждого из наблюдений в тренировочных данных (каждый случай передачи всех наблюдений через сеть называется *эпохой*, и процесс тренировки сети обычно состоит из нескольких эпох), многократно обновляя значения параметров.

В этой главе для построения, тренировки и оценивания различных нейронных сетей мы будем использовать популярную библиотеку Python под названием Keras. Keras — это библиотека высокого уровня, в которой в качестве своего "движка" используются другие библиотеки, такие как TensorFlow и Theano. Для нас преимущество Keras состоит в том, что мы можем сосредоточиться на процессе проектирования и тренировки сети, оставив специфику тензорных операций другим библиотекам.

Нейронные сети, созданные с помощью программного кода Keras, могут быть натренированы как на центральных процессарах (т. е. на ноутбуке), так и на графических процессорах (т. е. на специализированном глубоко обучающемся компьютере). В реальном мире с реальными данными *настоятельно* рекомендуется тренировать нейронные сети с помощью графических процессоров (GPU), однако ради учебных целей все нейронные сети в этой книге малы и достаточно просты и могут быть натренированы на вашем ноутбуке (на CPU) всего за несколько минут. Просто имейте в виду, что при наличии больших сетей и больших тренировочных данных тренировка с использованием CPU проходит значительно медленнее, чем тренировка с помощью GPU.

20.1. Предобработка данных для нейронных сетей

Задача

Требуется выполнить предобработку данных для использования в нейронной сети.

Решение

Стандартизировать каждый признак, используя класс `StandardScaler` библиотеки scikit-learn:

```
# Загрузить библиотеки
from sklearn import preprocessing
import numpy as np

# Создать признак
features = np.array([[-100.1, 3240.1],
                     [-200.2, -234.1],
                     [5000.5, 150.1],
                     [6000.6, -125.1],
                     [9000.9, -673.1]])

# Создать шкалировщик
scaler = preprocessing.StandardScaler()

# Преобразовать признак
features_standardized = scaler.fit_transform(features)

# Показать признак
features_standardized

array([[-1.12541308,  1.96429418],
       [-1.15329466, -0.50068741],
       [ 0.29529406, -0.22809346],
       [ 0.57385917, -0.42335076],
       [ 1.40955451, -0.81216255]])
```

Обсуждение

Хотя этот рецепт очень похож на рецепт 4.3, его стоит повторить по причине своей крайней важности для нейронных сетей. Обычно параметры нейронной сети инициализируются (т. е. создаются) небольшими случайными числами. Нейронные сети часто ведут себя плохо, когда значения признаков намного больше значений параметров. Более того, поскольку при прохождении через отдельные структурно-функциональные единицы значения признаков наблюдений объединяются, важно, чтобы все признаки имели одинаковую шкалу измерения.

По этим причинам рекомендуется (хотя это не всегда необходимо; например, когда все признаки — бинарные) стандартизировать каждый признак таким образом, что значения признака имели среднее значение, равное 0, и стандартное отклонение, равное 1. Это можно легко сделать с помощью объекта класса `StandardScaler` библиотеки scikit-learn.

Эффект стандартизации можно увидеть, проверив среднее значение и стандартное отклонение наших первых признаков:

```
# Напечатать среднее значение и стандартное отклонение
print("Среднее значение:", round(features_standardized[:,0].mean()))
print("Стандартное отклонение:", features_standardized[:,0].std())
```

```
Среднее значение: 0.0
Стандартное отклонение: 1.0
```

20.2. Проектирование нейронной сети

Задача

Требуется спроектировать нейронную сеть.

Решение

Использовать модель `Sequential` библиотеки Keras:

```
# Загрузить библиотеки
from keras import models
from keras import layers

# Инициализировать нейронную сеть
network = models.Sequential()

# Добавить полносвязный слой с активационной функцией ReLU
network.add(layers.Dense(units=16, activation="relu", input_shape=(10,)))

# Добавить полносвязный слой с активационной функцией ReLU
network.add(layers.Dense(units=16, activation="relu"))

# Добавить полносвязный слой с сигмоидальной активационной функцией
network.add(layers.Dense(units=1, activation="sigmoid"))

# Скомпилировать нейронную сеть
network.compile(
    loss="binary_crossentropy", # Перекрестная энтропия
    optimizer="rmsprop",  # Распространение среднеквадратической ошибки
    metrics=["accuracy"]) # Точностный показатель результативности
```

```
Using TensorFlow backend.
```

Обсуждение

Нейронные сети состоят из слоев структурно-функциональных единиц, далее блоков. Вместе с тем существует невероятное разнообразие типов слоев и способов их объединения для формирования сетевой архитектуры. Хотя в настоящее время существуют часто используемые архитектурные шаблоны (которые мы рассмотрим в этой главе), истина заключается в том, что выбор правильной архитектуры в основном является искусством и темой многих исследований.

Для построения нейронной сети прямого распространения в Keras нам необходимо сделать ряд решений в отношении вариантов как сетевой архитектуры, так и тренировочного процесса. Помните, что каждый блок в скрытых слоях:

1. Получает серию входов.
2. Взвешивает каждый вход на значение параметра.
3. Суммирует все взвешенные входы между собой с некоторым смещением (обычно 1).
4. Чаще всего затем применяется некая функция (называемая активационной функцией).
5. Направляет выходы дальше в блоки в следующем слое.

Во-первых, для каждого слоя в скрытом и выходном слоях мы должны определить число блоков в слое и активационную функцию. В целом, чем больше блоков в слое, тем больше наша сеть способна заучивать сложные шаблоны. Однако чрезмерное увеличение количества блоков может привести к тому, что наша сеть будет чересчур подогнана (слишком плотно прилегать) под тренировочные данные, что пагубно скажется на ее результативности на тестовых данных.

Для скрытых слоев популярной активационной функцией является выпрямленный линейный блок (rectified linear unit, ReLU):

$$f(z) = \max(0, z),$$

где z — сумма взвешенных входов и смещения. Как мы видим, если $z > 0$, то активационная функция возвращает z; в противном случае она возвращает 0. Эта простая активационная функция имеет ряд желательных свойств (обсуждение которых выходит за рамки данной книги), что сделало ее популярной в нейронных сетях. Однако мы должны понимать, что помимо нее существует еще множество десятков других активационных функций.

Во-вторых, нам нужно определить количество скрытых слоев для использования в сети. Больше слоев позволяет сети заучивать более сложные связи, но с бóльшими вычислительными затратами.

В-третьих, мы должны определить структуру активационной функции (если таковая имеется) выходного слоя. Характер выходной функции часто определяется целью сети. Приведем некоторые распространенные шаблоны выходного слоя:

- *бинарная классификация* — один блок с сигмоидальной активационной функцией;
- *мультиклассовая классификация* — k блоков (где k — это количество целевых классов) и активационная функция softmax;
- *регрессия* — один блок без активационной функции.

В-четвертых, нам нужно определить функцию потери (функцию, которая измеряет, насколько хорошо предсказанное значение соответствует истинному значению); это опять-таки часто определяется типом задачи:

- *бинарная классификация* — бинарная перекрестная энтропия;
- *мультиклассовая классификация* — категориальная перекрестная энтропия;
- *регрессия* — среднеквадратическая ошибка.

В-пятых, мы должны определить оптимизатор, который интуитивно можно рассматривать как нашу стратегию "хождения вокруг" функции потери, чтобы найти значения параметров, которые производят наименьшую ошибку. Наиболее распространенными вариантами оптимизаторов являются стохастический градиентный спуск, стохастический градиентный спуск с импульсом, распространение среднеквадратической ошибки и адаптивное оценивание момента (подробнее об этих оптимизаторах можно узнать, если перейти по ссылкам в *разд. "Дополнительные материалы для чтения"*).

В-шестых, мы можем выбрать один или несколько метрических показателей для оценивания результативности, таких как точность.

Библиотека Keras предлагает два способа создания нейронных сетей. Последовательная модель Keras создает нейронные сети путем ярусной укладки слоев. Альтернативный метод создания нейронных сетей называется функциональным API, но он больше предназначен для исследователей, чем для практиков.

В нашем решении мы создали двухслойную нейронную сеть (при подсчете слоев мы не включаем входной слой, потому что он не имеет никаких заучиваемых параметров), используя последовательную модель Keras. Каждый слой является "плотным" (или полносвязанным), т. е. все блоки в предыдущем слое связаны со всеми нейронами в следующем слое. В первом скрытом слое мы устанавливаем количество блоков `units=16`, имея в виду, что слой содержит 16 блоков с активационными функциями ReLU: `activation='relu'`. В библиотеке Keras первый скрытый слой любой сети должен включать параметр `input_shape`, который описывает форму признаковых данных. Например, `(10,)` сообщает первому слою, что каждое наблюдение будет иметь 10 значений признаков. Наш второй слой такой же, как и первый, но без необходимости в параметре `input_shape`. Эта сеть предназначена для бинарной классификации, поэтому выходной слой содержит только один блок с сигмоидной активационной функцией, которая ограничивает выход диапазоном от 0 до 1 (представляя вероятность, что наблюдение принадлежит классу 1).

Наконец, прежде чем мы сможем натренировать нашу модель, мы должны сообщить Keras, как мы хотим, чтобы наша сеть училась. Мы делаем это с помощью метода `compile` с оптимизационным алгоритмом (`RMSProp`), функцией потери (`binary_crossentropy`) и одной или несколькими метрическими показателями результативности.

Дополнительные материалы для чтения

- Документация библиотеки Keras по функциям потерь (**https://keras.io/losses/**).
- "Функции потерь для задачи классификации", Википедия (**http://bit.ly/2FwpCkM**).
- "О функциях потерь для глубоких нейронных сетей в задаче классификации", статья в библиотеке Корнуэллского университета (Katarzyna Janocha, Wojciech Marian Czarnecki. "On Loss Functions for Deep Neural Networks in Classification"; **https://arxiv.org/abs/1702.05659**).

20.3. Тренировка бинарного классификатора

Задача

Требуется натренировать бинарно-классификационную нейронную сеть.

Решение

Использовать библиотеку Keras для построения нейронной сети прямого распространения и ее тренировки с помощью метода `fit`:

```
# Загрузить библиотеки
import numpy as np
from keras.datasets import imdb
from keras.preprocessing.text import Tokenizer
from keras import models
from keras import layers

# Задать начальное значение для генератора псевдослучайных чисел
np.random.seed(0)

# Задать желаемое количество признаков
number_of_features = 1000

# Загрузить данные и вектор целей из данных с отзывами о кинофильмах
(data_train, target_train), (data_test, target_test) = imdb.load_data(
    num_words=number_of_features)

# Конвертировать данные с отзывами о кинофильмах в
# матрицу признаков в кодировке с одним активным состоянием
tokenizer = Tokenizer(num_words=number_of_features)
features_train = tokenizer.sequences_to_matrix(data_train, mode="binary")
features_test = tokenizer.sequences_to_matrix(data_test, mode="binary")

# Инициализировать нейронную сеть
network = models.Sequential()

# Добавить полносвязный слой с активационной функцией ReLU
network.add(layers.Dense(units=16, activation="relu",
                         input_shape=(number_of_features,)))

# Добавить полносвязный слой с активационной функцией ReLU
network.add(layers.Dense(units=16, activation="relu"))

# Добавить полносвязный слой с сигмоидальной активационной функцией
network.add(layers.Dense(units=1, activation="sigmoid"))
```

```
# Скомпилировать нейронную сеть
network.compile(
    loss="binary_crossentropy", # Перекрестная энтропия
    optimizer="rmsprop",  # Распространение среднеквадратической ошибки
    metrics=["accuracy"]) # Точностный показатель результативности

# Натренировать нейронную сеть
history = network.fit(
    features_train, # Признаки
    target_train,   # Вектор целей
    epochs=3,       # Количество эпох
    verbose=1,      # Печатать описание после каждой эпохи
    batch_size=100, # Количество наблюдений на пакет
    validation_data=(features_test, target_test)) # Тестовые данные

Using TensorFlow backend.

Downloading data from https://s3.amazonaws.com/text-datasets/imdb.npz
17465344/17464789 [==============================] - 12s 1us/step
Train on 25000 samples, validate on 25000 samples
Epoch 1/3
25000/25000 [==============================] - 4s 162us/step - loss: 0.4159 - acc:
0.8135 - val_loss: 0.3349 - val_acc: 0.8582
Epoch 2/3
25000/25000 [==============================] - 2s 64us/step - loss: 0.3249 - acc:
0.8633 - val_loss: 0.3300 - val_acc: 0.8604
Epoch 3/3
25000/25000 [==============================] - 2s 65us/step - loss: 0.3155 - acc:
0.8660 - val_loss: 0.3305 - val_acc: 0.8596
```

Обсуждение

В рецепте 20.2 мы обсуждали, каким образом строить нейронную сеть, используя последовательную модель библиотеки Keras. В данном рецепте мы тренируем эту нейронную сеть, используя реальные данные. В частности, мы используем 50 000 отзывов о кинофильмах (25 000 в качестве тренировочных данных, 25 000 для тестирования), классифицируемых как положительные или отрицательные. Мы преобразуем текст отзывов в 5000 бинарных признаков, указывающих на наличие одного из 1000 наиболее часто встречающихся слов. Проще говоря, наши нейронные сети будут использовать 25 000 наблюдений, каждое с 1000 признаками, чтобы предсказать, является ли отзыв о фильме положительным или отрицательным.

Используемая нами нейронная сеть такая же, как и в рецепте 20.2 (см. подробное объяснение). Единственное дополнение состоит в том, что в том рецепте мы лишь создали нейронную сеть и не тренировали ее.

В библиотеке Keras мы тренируем нашу нейронную сеть с помощью метода `fit`. Необходимо определить шесть важных параметров. Первые два параметра — это

признаки и вектор целей в тренировочных данных. Мы можем осмотреть форму матрицы признаков, используя атрибут `shape`:

```
# Взглянуть на форму матрицы признаков
features_train.shape
```

```
(25000, 1000)
```

Параметр `epochs` определяет, сколько эпох следует использовать во время тренировки на данных. Параметр `verbose` определяет, сколько информации выводится во время тренировочного процесса, при этом `0` означает, что информация не выводится, `1` — выводится индикатор выполнения и `2` — выводится одна строка регистрации операций на эпоху. Параметр `batch_size` задает количество наблюдений для распространения по сети перед обновлением параметров.

Наконец, мы отложили в сторону тестовый набор данных для оценивания модели. Эти тестовые признаки и тестовый вектор целей могут быть значениями параметра `validation_data`, который будет их использовать для оценивания. В качестве альтернативы можно применить метод `validation_split`, чтобы определить, какую часть тренировочных данных следует отложить для оценивания.

В библиотеке scikit-learn метод `fit` возвращал натренированную модель, но в библиотеке Keras метод `fit` возвращает объект `History`, содержащий значения потери и метрические показатели результативности в каждой эпохе.

20.4. Тренировка мультиклассового классификатора

Задача

Требуется натренировать мультиклассовую классификационную нейронную сеть.

Решение

Использовать библиотеку Keras для построения нейронной сети прямого распространения с выходным слоем с активационными функциями softmax:

```
# Загрузить библиотеки
import numpy as np
from keras.datasets import reuters
from keras.utils.np_utils import to_categorical
from keras.preprocessing.text import Tokenizer
from keras import models
from keras import layers

# Задать начальное значение для ГПСЧ
np.random.seed(0)
```

```
# Задать желаемое количество признаков
number_of_features = 5000

# Загрузить признаковые и целевые данные
data = reuters.load_data(num_words=number_of_features)
(data_train, target_vector_train), (data_test, target_vector_test) = data

# Конвертировать признаковые данные
# в матрицу признаков в кодировке с одним активным состоянием
tokenizer = Tokenizer(num_words=number_of_features)
features_train = tokenizer.sequences_to_matrix(data_train, mode="binary")
features_test = tokenizer.sequences_to_matrix(data_test, mode="binary")

# Преобразовать вектор целей в кодировке с одним активным состоянием,
# чтобы создать матрицу целей
target_train = to_categorical(target_vector_train)
target_test = to_categorical(target_vector_test)

# Инициализировать нейронную сеть
network = models.Sequential()

# Добавить полносвязный слой с активационной функцией ReLU
network.add(layers.Dense(
    units=100,
    activation="relu",
    input_shape=(number_of_features,)))

# Добавить полносвязный слой с активационной функцией ReLU
network.add(layers.Dense(units=100, activation="relu"))

# Добавить полносвязный слой с активационной функцией softmax
network.add(layers.Dense(units=46, activation="softmax"))

# Скомпилировать нейронную сеть
network.compile(
    loss="categorical_crossentropy", # Перекрестная энтропия
    optimizer="rmsprop",  # Распространение СКО
    metrics=["accuracy"]) # Точностный показатель результативности

# Натренировать нейронную сеть
history = network.fit(
    features_train, # Признаки
    target_train,   # Цель
    epochs=3,       # Три эпохи
    verbose=0,      # Вывода нет
    batch_size=100, # Количество наблюдений на пакет
    validation_data=(features_test, target_test)) # Тестовые данные
```

```
Using TensorFlow backend.
```

Обсуждение

В этом решении мы создали нейронную сеть, похожую на бинарный классификатор из предыдущего рецепта, но с некоторыми заметными изменениями. Во-первых, наши данные — это 11 228 новостных выпусков информагентства Reuters. Каждый выпуск разделен на 46 тем. Мы подготовили наши признаковые данные, преобразовав новостные выпуски в 5000 бинарных признаков (обозначая присутствие определенного слова в новостном выпуске). Мы подготовили целевые данные, преобразовав их в кодировку с одним активным состоянием, чтобы получить матрицу целей, обозначающую, к какому из 46 классов относится наблюдение:

```
# Осмотреть матрицу целей
target_train

array([[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., 0.],
       [0., 0., 0., ..., 0., 0., 0.]], dtype=float32)
```

Во-вторых, мы увеличили количество блоков в каждом из скрытых слоев, чтобы помочь нейронной сети представлять более сложные связи между 46 классами.

В-третьих, поскольку эта задача является мультиклассовой классификационной задачей, мы использовали выходной слой с 46 блоками (по одному на класс), содержащими активационную функцию softmax. Активационная функция softmax вернет массив из 46 значений, дающих в сумме 1. Эти 46 значений представляют вероятность того, что наблюдение принадлежит каждому из 46 классов.

В-четвертых, мы использовали функцию потери, подходящую для мультиклассовой классификации, категориальную перекрестно-энтропийную функцию потери, `categorical_crossentropy`.

20.5. Тренировка регрессора

Задача

Требуется натренировать регрессионную нейронную сеть.

Решение

Использовать библиотеку Keras для построения нейронной сети прямого распространения с одним выходным блоком и без активационной функции:

```
# Загрузить библиотеки
import numpy as np
from keras.preprocessing.text import Tokenizer
```

```
from keras import models
from keras import layers
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn import preprocessing

# Задать начальное значение для ГПСЧ
np.random.seed(0)

# Сгенерировать матрицу признаков и вектор целей
features, target = make_regression(n_samples = 10000,
                                   n_features = 3,
                                   n_informative = 3,
                                   n_targets = 1,
                                   noise = 0.0,
                                   random_state = 0)

# Разделить данные на тренировочный и тестовый наборы
features_train, features_test, target_train, target_test = train_test_split(
    features, target, test_size=0.33, random_state=0)

# Инициализировать нейронную сеть
network = models.Sequential()

# Добавить полносвязный слой с активационной функцией ReLU
network.add(layers.Dense(
    units=32,
    activation="relu",
    input_shape=(features_train.shape[1],)))

# Добавить полносвязный слой с активационной функцией ReLU
network.add(layers.Dense(units=32, activation="relu"))

# Добавить полносвязный слой без активационной функции
network.add(layers.Dense(units=1))

# Скомпилировать нейронную сеть
network.compile(
    loss="mse",          # Среднеквадратическая ошибка
    optimizer="RMSprop", # Оптимизационный алгоритм
    metrics=["mse"])     # Среднеквадратическая ошибка

# Натренировать нейронную сеть
history = network.fit(
    features_train, # Признаки
    target_train,   # Вектор целей
    epochs=10,      # Количество эпох
```

```
    verbose=0,       # Вывода нет
    batch_size=100, # Количество наблюдений на пакет
    validation_data=(features_test, target_test)) # Тестовые данные
```

```
Using TensorFlow backend.
```

Обсуждение

Абсолютно возможно создать нейронную сеть, которая вместо вероятностей классов будет предсказывать непрерывные значения. В случае нашего бинарного классификатора (см. рецепт 20.3) мы использовали выходной слой с одним блоком и сигмоидальную активационную функцию для получения вероятности того, что наблюдение принадлежит классу 1. Важно отметить, что сигмоидальная активационная функция ограничивала выводимое значение диапазоном между 0 и 1. Если это ограничение удалить, вообще убрав активационную функцию, то тем самым мы позволим на выходе получать непрерывное значение.

Кроме того, поскольку мы тренируем регрессию, мы должны использовать соответствующую функцию потери и оценочный метрический показатель, в нашем случае среднюю квадратическую ошибку:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i),$$

где n — количество наблюдений; y_i — истинное значение цели, которую мы пытаемся предсказать для наблюдения i; $\hat{y}_i$ — предсказанное моделью значение для y_i.

Наконец, поскольку мы используем данные, симулированные с помощью функции `make_regression` библиотеки scikit-learn, нам не пришлось стандартизировать признаки. Однако следует отметить, что стандартизация потребуется практически во всех реальных случаях.

20.6. Выполнение предсказаний

Задача

Требуется применить нейронную сеть для выполнения предсказаний.

Решение

Использовать библиотеку Keras для построения нейронной сети прямого распространения, а затем делать предсказания с помощью метода `predict`:

```
# Загрузить библиотеки
import numpy as np
from keras.datasets import imdb
from keras.preprocessing.text import Tokenizer
from keras import models
from keras import layers
```

```
# Задать начальное значение для ГПСЧ
np.random.seed(0)

# Задать желаемое количество признаков
number_of_features = 10000

# Загрузить данные и вектор целей из набора данных IMDB о кинофильмах
(data_train, target_train), (data_test, target_test) = imdb.load_data(
    num_words=number_of_features)

# Конвертировать данные IMDB о кинофильмах
# в матрицу признаков в кодировке с одним активным состоянием
tokenizer = Tokenizer(num_words=number_of_features)
features_train = tokenizer.sequences_to_matrix(data_train, mode="binary")
features_test = tokenizer.sequences_to_matrix(data_test, mode="binary")

# Инициализировать нейронную сеть
network = models.Sequential()

# Добавить полносвязный слой с активационной функцией ReLU
network.add(layers.Dense(
    units=16,
    activation="relu",
    input_shape=(number_of_features,)))

# Добавить полносвязный слой с активационной функцией ReLU
network.add(layers.Dense(units=16, activation="relu"))

# Добавить полносвязный слой с сигмоидальной активационной функцией
network.add(layers.Dense(units=1, activation="sigmoid"))

# Скомпилировать нейронную сеть
network.compile(
    loss="binary_crossentropy", # Перекрестная энтропия
    optimizer="rmsprop",  # Распространение СКО
    metrics=["accuracy"]) # Точностный показатель результативности

# Натренировать нейронную сеть
history = network.fit(
    features_train, # Признаки
    target_train,   # Вектор целей
    epochs=3,       # Количество эпох
    verbose=0,      # Вывода нет
    batch_size=100, # Количество наблюдений на пакет
    validation_data=(features_test, target_test)) # Тестовые данные
```

```
# Предсказать классы тестового набора
predicted_target = network.predict(features_test)
```

```
Using TensorFlow backend.
```

Обсуждение

В библиотеке Keras предсказания делаются легко. После того как мы натренировали нашу нейронную сеть, мы можем использовать метод `predict`, который в качестве аргумента принимает набор признаков и возвращает предсказанный результат для каждого наблюдения. В нашем решении нейронная сеть настроена для бинарной классификации, поэтому предсказываемым выходом является вероятность принадлежности классу 1. Наблюдения с предсказанными значениями очень близкими к 1 с большой вероятностью будут принадлежать классу 1, а наблюдения с предсказанными значениями очень близкими к 0, весьма вероятно, будут принадлежать классу 0. Например, ниже приведена предсказанная вероятность, что первое наблюдение в нашей тестовой матрице признаков принадлежит классу 1:

```
# Взглянуть на вероятность, что первое наблюдение принадлежит классу 1
predicted_target[0]
```

```
array([0.05590831], dtype=float32)
```

20.7. Визуализация истории процесса тренировки

Задача

Требуется найти "золотую середину" в потере и/или оценке точности нейронной сети.

Решение

Использовать библиотеку matplotlib, чтобы визуализировать потерю на тестовом и тренировочном наборах данных по каждой эпохе (рис. 20.1):

```
# Загрузить библиотеки
import numpy as np
from keras.datasets import imdb
from keras.preprocessing.text import Tokenizer
from keras import models
from keras import layers
import matplotlib.pyplot as plt

# Задать начальное значение для ГПСЧ
np.random.seed(0)
```

```
# Задать желаемое количество признаков
number_of_features = 10000

# Загрузить данные и вектор целей из данных с отзывами о кинофильмах
(data_train, target_train), (data_test, target_test) = imdb.load_data(
    num_words=number_of_features)

# Конвертировать данные отзывов о кинофильмах
# в матрицу признаков в кодировке с одним активным состоянием
tokenizer = Tokenizer(num_words=number_of_features)
features_train = tokenizer.sequences_to_matrix(data_train, mode="binary")
features_test = tokenizer.sequences_to_matrix(data_test, mode="binary")

# Инициализировать нейронную сеть
network = models.Sequential()

# Добавить полносвязный слой с активационной функцией ReLU
network.add(layers.Dense(
    units=16,
    activation="relu",
    input_shape=(number_of_features,)))

# Добавить полносвязный слой с активационной функцией ReLU
network.add(layers.Dense(units=16, activation="relu"))

# Добавить полносвязный слой с сигмоидальной активационной функцией
network.add(layers.Dense(units=1, activation="sigmoid"))

# Скомпилировать нейронную сеть
network.compile(
    loss="binary_crossentropy", # Перекрестная энтропия
    optimizer="rmsprop",  # Распространение СКО
    metrics=["accuracy"]) # Точностный показатель результативности

# Натренировать нейронную сеть
history = network.fit(
    features_train,  # Признаки
    target_train,    # Цели
    epochs=15,       # Количество эпох
    verbose=0,       # Вывода нет
    batch_size=1000, # Количество наблюдений на пакет
    validation_data=(features_test, target_test)) # Тестовые данные

# Получить истории потерь на тренировочных и тестовых данных
training_loss = history.history["loss"]
test_loss = history.history["val_loss"]
```

```
# Создать счетчик количества эпох
epoch_count = range(1, len(training_loss) + 1)

# Визуализировать историю потери
plt.plot(epoch_count, training_loss, "r--")
plt.plot(epoch_count, test_loss, "b-")
plt.legend(["Потеря на тренировке",
            "Потеря на тестировании"])
plt.xlabel("Эпоха")
plt.ylabel("Потеря")
plt.show();

Using TensorFlow backend.
```

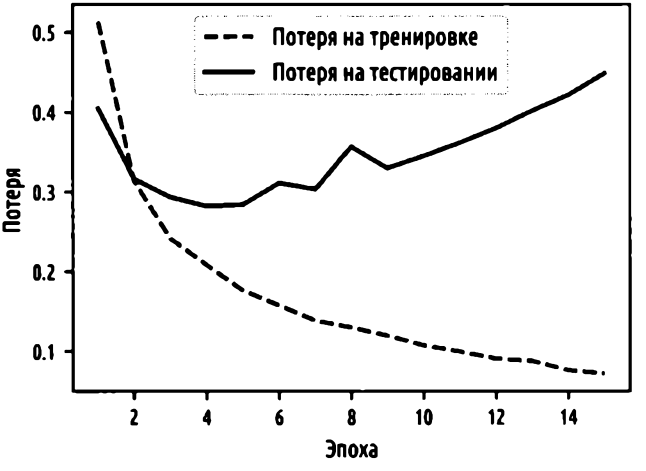


Рис. 20.1

В качестве альтернативы можно использовать тот же самый подход для визуализации точности на тренировочных и на тестовых данных по каждой эпохе (рис. 20.2):

```
# Получить истории потерь на тренировочных и тестовых данных
training_accuracy = history.history["acc"]
test_accuracy = history.history["val_acc"]
plt.plot(epoch_count, training_accuracy, "r--")
plt.plot(epoch_count, test_accuracy, "b-")

# Визуализировать историю потери
plt.legend(["Потеря на тренировке",
            "Потеря на тестировании"])
plt.xlabel("Эпоха")
plt.ylabel("Оценка точности")
plt.show();
```

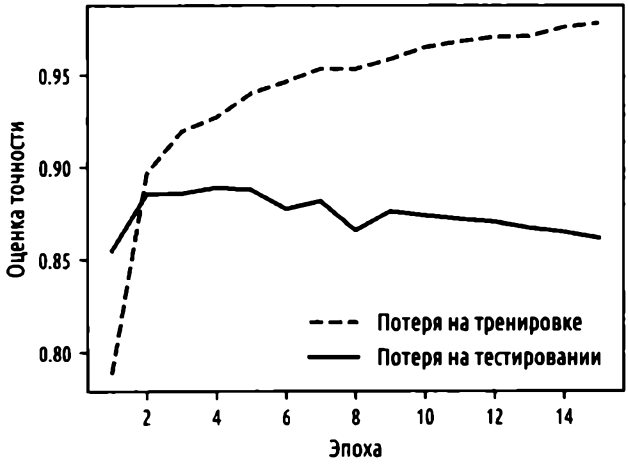


Рис. 20.2

Обсуждение

В самом начале наша нейронная сеть будет иметь низкую результативность. По мере того как нейронная сеть учится на тренировочных данных, модельная ошибка как на тренировочном, так и на тестовом наборе будет стремиться к увеличению. Однако в определенный момент нейронная сеть начинает "запоминать" тренировочные данные, и наступает ее переподгонка, т. е. модель начинает слишком плотно прилегать к данным. Когда это начнет происходить, ошибка на тренировочных данных будет уменьшаться, а ошибка на тестовых данных начнет увеличиваться. Поэтому во многих случаях существует "золотая середина", где ошибка на тестовых данных (т. е. ошибка, которая нас интересует обычно в первую очередь) находится в самой низкой точке. Этот эффект можно ясно увидеть в решении, где мы визуализируем потерю на тренировочных данных и потерю на тестовых данных в каждой эпохе. Обратите внимание, что на первом графике ошибка на тестовых данных является самой низкой около пятой эпохи, после чего потеря на тренировочных данных продолжает уменьшаться, в то время как потеря на тестовых данных начинает увеличиваться. В этой точке наступает переподгонка модели.

20.8. Снижение переподгонки с помощью регуляризации весов

Задача

Требуется снизить переподгонку.

Решение

Попробовать применить штраф к параметрам сети, так называемую *регуляризацию весов*:

```
# Загрузить библиотеки
import numpy as np
from keras.datasets import imdb
from keras.preprocessing.text import Tokenizer
from keras import models
from keras import layers
from keras import regularizers

# Задать начальное значение для ГПСЧ
np.random.seed(0)

# Задать желаемое количество признаков
number_of_features = 1000

# Загрузить данные и вектор целей из данных с отзывами о кинофильмах
(data_train, target_train), (data_test, target_test) = imdb.load_data(
    num_words=number_of_features)

# Конвертировать данные отзывов о кинофильмах
# в матрицу признаков в кодировке с одним активным состоянием
tokenizer = Tokenizer(num_words=number_of_features)
features_train = tokenizer.sequences_to_matrix(data_train, mode="binary")
features_test = tokenizer.sequences_to_matrix(data_test, mode="binary")

# Инициализировать нейронную сеть
network = models.Sequential()

# Добавить полносвязный слой с активационной функцией ReLU
network.add(layers.Dense(
    units=16,
    activation="relu",
    kernel_regularizer=regularizers.l2(0.01),
    input_shape=(number_of_features,)))

# Добавить полносвязный слой с активационной функцией ReLU
network.add(layers.Dense(
    units=16,
    kernel_regularizer=regularizers.l2(0.01),
    activation="relu"))

# Добавить полносвязный слой с сигмоидальной активационной функцией
network.add(layers.Dense(units=1, activation="sigmoid"))

# Скомпилировать нейронную сеть
network.compile(
    loss="binary_crossentropy", # Перекрестная энтропия
    optimizer="rmsprop",  # Распространение СКО
    metrics=["accuracy"]) # Точностный показатель результативности
```

```
# Натренировать нейронную сеть
history = network.fit(
    features_train, # Признаки
    target_train,   # Вектор целей
    epochs=3,       # Количество эпох
    verbose=0,      # Вывода нет
    batch_size=100, # Количество наблюдений на пакет
    validation_data=(features_test, target_test)) # Тестовые данные
```

```
Using TensorFlow backend.
```

Обсуждение

Одна из стратегий борьбы с переподгонкой нейронных сетей заключается в наложении штрафа на параметры (т. е. веса) нейронной сети таким образом, чтобы они сводились к малым значениям — создавая более простую модель, менее склонную к переподгонке. Этот метод называется регуляризацией весов или снижением весов. Если быть конкретнее, в регуляризации весов штраф, такой как норма L^2, добавляется к функции потери.

В библиотеке Keras можно добавить регуляризацию весов, включив в параметры слоя использование `kernel_regularizer=regularizers.l2(0.01)`. В этом примере значение `0.01` определяет, насколько мы штрафуем более высокие значения параметров.

20.9. Снижение переподгонки с помощью ранней остановки

Задача

Требуется снизить переподгонку.

Решение

Попробовать остановить тренировку, когда потеря на тестовых данных перестанет уменьшаться, такая стратегия называется *ранней остановкой*:

```
# Загрузить библиотеки
import numpy as np
from keras.datasets import imdb
from keras.preprocessing.text import Tokenizer
from keras import models
from keras import layers
from keras.callbacks import EarlyStopping, ModelCheckpoint

# Задать начальное значение ГПСЧ
np.random.seed(0)
```

```
# Задать желаемое количество признаков
number_of_features = 1000

# Загрузить данные и вектор целей из данных с отзывами о кинофильмах
(data_train, target_train), (data_test, target_test) = imdb.load_data(
    num_words=number_of_features)

# Конвертировать данные отзывов о кинофильмах
# в матрицу признаков в кодировке с одним активным состоянием
tokenizer = Tokenizer(num_words=number_of_features)
features_train = tokenizer.sequences_to_matrix(data_train, mode="binary")
features_test = tokenizer.sequences_to_matrix(data_test, mode="binary")

# Инициализировать нейронную сеть
network = models.Sequential()

# Добавить полносвязный слой с активационной функцией ReLU
network.add(layers.Dense(
    units=16,
    activation="relu",
    input_shape=(number_of_features,)))

# Добавить полносвязный слой с активационной функцией ReLU
network.add(layers.Dense(units=16, activation="relu"))

# Добавить полносвязный слой с сигмоидальной активационной функцией
network.add(layers.Dense(units=1, activation="sigmoid"))

# Скомпилировать нейронную сеть
network.compile(
    loss="binary_crossentropy", # Перекрестная энтропия
    optimizer="rmsprop",  # Распространение СКО
    metrics=["accuracy"]) # Точностный показатель результативности

# Задать функции обратного вызова для ранней остановки тренировки
# и сохранения наилучшей достигнутой модели
callbacks = [EarlyStopping(monitor="val_loss", patience=2),
             ModelCheckpoint(filepath="best_model.h5",
             monitor="val_loss",
             save_best_only=True)]

# Натренировать нейронную сеть
history = network.fit(
    features_train, # Признаки
    target_train,   # Вектор целей
    epochs=20,      # Количество эпох
    callbacks=callbacks, # Ранняя остановка
```

```
    verbose=0,      # Печатать описание после каждой эпохи
    batch_size=100, # Количество наблюдений на пакет
    validation_data=(features_test, target_test)) # Тестовые данные
```

```
Using TensorFlow backend.
```

Обсуждение

Как мы отмечали в рецепте 20.7, обычно в первые тренировочные эпохи ошибки на тренировочных данных и ошибки на тестовых данных будут уменьшаться, но в какой-то момент сеть начнет "запоминать" тренировочные данные, в результате чего ошибка на тренировочных данных будет продолжать уменьшаться даже тогда, когда ошибка на тестовых данных начнет увеличиваться. Это явление называется *переподгонкой*, и одним из наиболее распространенных и очень эффективных методов противодействия переподгонке является отслеживание тренировочного процесса и его остановка, когда ошибка на тестовых данных начинает увеличиваться. Эта стратегия называется *ранней остановкой*.

В библиотеке Keras можно реализовать раннюю остановку в качестве функции обратного вызова. *Обратные вызовы* — это функции, которые могут быть применены на определенных этапах тренировочного процесса, например, в конце каждой эпохи. В частности, в нашем решении мы включили `EarlyStopping(monitor= 'val_loss', patience=2)`, чтобы указать, что мы хотим отслеживать потерю на тестовых данных в каждую эпоху, и после того, как потеря на тестовых данных не улучшается на протяжении двух эпох, тренировка прерывается. Вместе с тем, поскольку мы установили `patience=2`, мы не получим наилучшую модель; это будет модель на две эпохи после наилучшей модели. Поэтому, дополнительно, мы можем включить вторую операцию `ModelCheckpoint`, которая сохраняет модель в файл после каждой контрольной точки (что может быть полезно в случае, если по какой-либо причине многодневная тренировка прерывается). Будет полезно установить параметр `save_best_only=True`, потому что тогда класс `ModelCheckpoint` сохранит только лучшую модель.

20.10. Снижение переподгонки с помощью отсева

Задача

Требуется уменьшить переподгонку.

Решение

Внести шумовую компоненту в сетевую архитектуру с помощью отсева:

```
# Загрузить библиотеки
import numpy as np
```

```
from keras.datasets import imdb
from keras.preprocessing.text import Tokenizer
from keras import models
from keras import layers

# Задать начальное значение для ГПСЧ
np.random.seed(0)

# Задать желаемое количество признаков
number_of_features = 1000

# Загрузить данные и вектор целей из данных с отзывами о кинофильмах
(data_train, target_train), (data_test, target_test) = imdb.load_data(
    num_words=number_of_features)

# Конвертировать данные отзывов о кинофильмах
# в матрицу признаков в кодировке с одним активным состоянием
tokenizer = Tokenizer(num_words=number_of_features)
features_train = tokenizer.sequences_to_matrix(data_train, mode="binary")
features_test = tokenizer.sequences_to_matrix(data_test, mode="binary")

# Инициализировать нейронную сеть
network = models.Sequential()

# Добавить отсеивающий слой для входного слоя
network.add(layers.Dropout(0.2, input_shape=(number_of_features,)))

# Добавить полносвязный слой с активационной функцией ReLU
network.add(layers.Dense(units=16, activation="relu"))

# Добавить отсеивающий слой для предыдущего скрытого слоя
network.add(layers.Dropout(0.5))

# Добавить полносвязный слой с активационной функцией ReLU
network.add(layers.Dense(units=16, activation="relu"))

# Добавить отсеивающий слой для предыдущего скрытого слоя
network.add(layers.Dropout(0.5))

# Добавить полносвязный слой с сигмоидальной активационной функцией
network.add(layers.Dense(units=1, activation="sigmoid"))

# Скомпилировать нейронную сеть
network.compile(
    loss="binary_crossentropy", # Перекрестная энтропия
    optimizer="rmsprop",  # Распространение СКО
    metrics=["accuracy"]) # Точностный показатель результативности
```

```
# Натренировать нейронную сеть
history = network.fit(
    features_train, # Признаки
    target_train,   # Вектор целей
    epochs=3,       # Количество эпох
    verbose=0,      # Вывода нет
    batch_size=100, # Количество наблюдений на пакет
    validation_data=(features_test, target_test)) # Тестовые данные
```

```
Using TensorFlow backend.
```

Обсуждение

Отсев, или прореживание, является популярным и мощным методом регуляризации нейронных сетей. В случае отсева каждый раз, когда создается пакет наблюдений для тренировки, доля блоков в одном или нескольких слоях умножается на ноль (т. е. сбрасывается). В этих условиях каждый пакет тренируется на той же сети (например, на одних и тех же параметрах), но каждый пакет сталкивается с немного другой версией *архитектуры* этой сети.

Отсев эффективен тем, что за счет постоянного и случайного обнуления блоков в каждом пакете он принуждает блоки заучивать значения параметров, способные хорошо работать в условиях большого разнообразия сетевых архитектур. То есть, они учатся быть устойчивыми к повреждениям (т. е. шуму) в других скрытых блоках, и это не дает сети просто механически запоминать тренировочные данные.

Отсев можно добавлять и в скрытые, и во входные слои. Когда сбрасывается входной слой, его значение признака не вводится в сеть для конкретного пакета. Общепринято выбирать долю отсеиваемых блоков в размере 0.2 для входных блоков и 0.5 для скрытых блоков.

В библиотеке Keras можно реализовать отсев, добавив в нашу сетевую архитектуру отсеивающие слои `Dropout`. Каждый слой `Dropout` будет отсеивать заданный пользователем гиперпараметр блоков в предыдущем слое каждого пакета. Напомним, что в библиотеке Keras входной слой считается первым слоем и не добавляется с помощью метода `add`. Поэтому, если мы хотим добавить отсев во входной слой, то первым слоем, который мы добавим в нашу сетевую архитектуру, будет отсеивающий слой. Этот слой содержит долю блоков входного слоя, подлежащих отсеву `0.2`, и параметр `input_shape`, определяющий форму данных наблюдений. Затем после каждого скрытого слоя мы добавляем отсеивающий слой с долей `0.5`.

20.11. Сохранение процесса тренировки модели

Задача

Дана нейронная сеть, на тренировку которой уходит много времени, и требуется возможность сохранить ход выполнения в случае, если процесс тренировки будет прерван.

Решение

Использовать функцию обратного вызова, реализованную классом `ModelCheckpoint` библиотеки Keras, чтобы записывать модель на диск после каждой эпохи:

```
# Загрузить библиотеки
import numpy as np
from keras.datasets import imdb
from keras.preprocessing.text import Tokenizer
from keras import models
from keras import layers
from keras.callbacks import ModelCheckpoint

# Задать начальное значение для ГПСЧ
np.random.seed(0)

# Задать желаемое количество признаков
number_of_features = 1000

# Загрузить данные и вектор целей из данных с отзывами о кинофильмах
(data_train, target_train), (data_test, target_test) = imdb.load_data(
    num_words=number_of_features)

# Конвертировать данные отзывов о кинофильмах
# в матрицу признаков в кодировке с одним активным состоянием
tokenizer = Tokenizer(num_words=number_of_features)
features_train = tokenizer.sequences_to_matrix(data_train, mode="binary")
features_test = tokenizer.sequences_to_matrix(data_test, mode="binary")

# Инициализировать нейронную сеть
network = models.Sequential()

# Добавить полносвязный слой с активационной функцией ReLU
network.add(layers.Dense(
    units=16,
    activation="relu",
    input_shape=(number_of_features,)))

# Добавить полносвязный слой с активационной функцией ReLU
network.add(layers.Dense(units=16, activation="relu"))

# Добавить полносвязный слой с сигмоидальной активационной функцией
network.add(layers.Dense(units=1, activation="sigmoid"))

# Скомпилировать нейронную сеть
network.compile(
    loss="binary_crossentropy", # Перекрестная энтропия
    optimizer="rmsprop",  # Распространение СКО
    metrics=["accuracy"]) # Точностный показатель результативности
```

```
# Задать функции обратного вызова для ранней остановки тренировки
# и сохранения наилучшей достигнутой модели
checkpoint = [ModelCheckpoint(filepath="models.hdf5")]

# Натренировать нейронную сеть
history = network.fit(
    features_train, # Признаки
    target_train,   # Вектор целей
    epochs=3,       # Количество эпох
    callbacks=checkpoint, # Контрольная точка
    verbose=0,      # Вывода нет
    batch_size=100, # Количество наблюдений на пакет
    validation_data=(features_test, target_test)) # Тестовые данные
```

```
Using TensorFlow backend.
```

Обсуждение

В рецепте 20.8 мы использовали функцию обратного вызова, реализованную в классе `ModelCheckpoint`, в сочетании с ранней остановкой `EarlyStopping` для завершения отслеживания и тренировки, когда ошибка на тестовых данных перестала улучшаться. Вместе с тем есть еще одна, более мирская причина для использования класса `ModelCheckpoint`. В реальном мире нейронные сети обычно тренируются часами или даже днями. За это время многое может пойти не так: компьютеры могут потерять питание, серверы могут рухнуть, или невнимательные аспиранты могут закрыть ваш ноутбук.

Класс `ModelCheckpoint` устраняет эту проблему, сохраняя модель после каждой эпохи. В частности, после каждой эпохи класс `ModelCheckpoint` сохраняет модель в расположение, указанное параметром `filepath`. Если мы включаем только имя файла (например, models.hdf5), этот файл будет перезаписан последней моделью каждой эпохи. Если требуется сохранить только лучшую модель в соответствии с результативностью некоторой функции потерь, то можно установить `save_best_only=True` и `monitor='val_loss'`, чтобы не перезаписывать файл, если модель имеет худшую потерю на тестовых данных, чем предыдущая модель. В качестве альтернативы можно сохранить модель каждой эпохи в отдельный файл, включив номер эпохи и оценку потери на тестовых данных в само имя файла. Например, если для параметра `filepath` задать значение `model_{epoch:02d}_{val_loss:.2f}.hdf5`, то имя файла, содержащего модель, сохраненную после 11-й эпохи со значением потери на тестовых данных 0.33, будет `model_10_0.35.hdf5` (обратите внимание, что номер эпохи индексируется с отсчетом от нуля).

20.12. *k*-блочная перекрестная проверка нейронных сетей

Задача

Требуется оценить нейронную сеть, используя *k*-блочную перекрестную проверку.

Решение

Часто *k*-блочная перекрестная проверка нейронных сетей не является ни необходимой, ни целесообразной. Однако, если это необходимо, следует использовать оболочку библиотеки Keras для scikit-learn, чтобы позволить последовательным моделям Keras применять API библиотеки scikit-learn:

```
# Загрузить библиотеки
import numpy as np
from keras import models
from keras import layers
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import cross_val_score
from sklearn.datasets import make_classification

# Задать начальное значение для ГПСЧ
np.random.seed(0)

# Количество признаков
number_of_features = 100

# Сгенерировать матрицу признаков и вектор целей
features, target = make_classification(n_samples = 10000,
                                       n_features = number_of_features,
                                       n_informative = 3,
                                       n_redundant = 0,
                                       n_classes = 2,
                                       weights = [.5, .5],
                                       random_state = 0)

# Создать функцию, возвращающую скомпилированную сеть
def create_network():

    # Инициализировать нейронную сеть
    network = models.Sequential()

    # Добавить полносвязный слой с активационной функцией ReLU
    network.add(layers.Dense(
        units=16,
        activation="relu",
        input_shape=(number_of_features,)))
```

```
    # Добавить полносвязный слой с активационной функцией ReLU
    network.add(layers.Dense(units=16, activation="relu"))

    # Добавить полносвязный слой с сигмоидальной активационной функцией
    network.add(layers.Dense(units=1, activation="sigmoid"))

    # Скомпилировать нейронную сеть
    network.compile(
        loss="binary_crossentropy", # Перекрестная энтропия
        optimizer="rmsprop",  # Распространение СКО
        metrics=["accuracy"]) # Точностный показатель результативности

    # Вернуть скомпилированную сеть
    return network

# Обернуть модель Keras, чтобы она могла
# использоваться библиотекой scikit-learn
neural_network = KerasClassifier(build_fn=create_network,
                                 epochs=10,
                                 batch_size=100,
                                 verbose=0)

# Оценить нейронную сеть с помощью трехблочной перекрестной проверки
cross_val_score(neural_network, features, target, cv=3)
```

```
Using TensorFlow backend.
```

```
array([0.90461907, 0.77647764, 0.87008701])
```

Обсуждение

Теоретически нет причин, по которым мы не можем применять перекрестную проверку для оценивания нейронных сетей. Вместе с тем нейронные сети часто используются на очень крупных данных, и на их тренировку может уходить несколько часов или даже дней. По этой причине, если время тренировки длинное, добавление вычислительных расходов за счет *k*-блочной перекрестной проверки нецелесообразно. Например, оценивание модели, на тренировку которой обычно требуется один день, с использованием 10-блочной перекрестной проверки уйдет 10 дней. Если имеются крупные данные, то часто целесообразно оценить нейронную сеть просто на некотором тестовом наборе.

Если имеются данные меньшего объема, то *k*-блочная перекрестная проверка может быть полезна с целью максимизировать нашу способность оценивать результативность нейронной сети. В библиотеке Keras это возможно, потому что мы можем "обернуть" любую нейронную сеть таким образом, чтобы она могла использовать оценивающие функции, доступные в библиотеке scikit-learn, включая *k*-блочную

перекрестную проверку. Для этого сначала нужно создать функцию, которая возвращает скомпилированную нейронную сеть. Затем для обертки модели мы применяем класс `KerasClassifier` (если у нас классификатор; если же у нас регрессор, то можно выбрать класс `KerasRegressor`), чтобы она могла использоваться библиотекой scikit-learn. После этого мы можем использовать нашу нейронную сеть, как и любой другой обучающийся алгоритм библиотеки scikit-learn (например, случайные леса, логистическая регрессия). В нашем решении, чтобы выполнить трехблочную перекрестную проверку на нашей нейронной сети, мы вызвали метод `cross_val_score`.

20.13. Тонкая настройка нейронных сетей

Задача

Требуется автоматически отобрать наилучшие гиперпараметры для вашей нейронной сети.

Решение

Объединить нейронную сеть библиотеки Keras с инструментами отбора модели библиотеки scikit-learn, такими как класс `GridSearchCV`, реализующий поиск по решетке параметров с перекрестной проверкой:

```
# Загрузить библиотеки
import numpy as np
from keras import models
from keras import layers
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.datasets import make_classification

# Задать начальное значение для ГПСЧ
np.random.seed(0)

# Количество признаков
number_of_features = 100

# Сгенерировать матрицу признаков и вектор целей
features, target = make_classification(n_samples = 10000,
                                       n_features = number_of_features,
                                       n_informative = 3,
                                       n_redundant = 0,
                                       n_classes = 2,
                                       weights = [.5, .5],
                                       random_state = 0)
```

```
# Создать функцию, возвращающую скомпилированную сеть
def create_network(optimizer="rmsprop"):

    # Инициализировать нейронную сеть
    network = models.Sequential()

    # Добавить полносвязный слой с активационной функцией ReLU
    network.add(layers.Dense(
        units=16,
        activation="relu",
        input_shape=(number_of_features,)))

    # Добавить полносвязный слой с активационной функцией ReLU
    network.add(layers.Dense(units=16, activation="relu"))

    # Добавить полносвязный слой с сигмоидальной активационной функцией
    network.add(layers.Dense(units=1, activation="sigmoid"))

    # Скомпилировать нейронную сеть
    network.compile(
        loss="binary_crossentropy", # Перекрестная энтропия
        optimizer=optimizer,  # Оптимизатор
        metrics=["accuracy"]) # Точностный показатель результативности

    # Вернуть скомпилированную сеть
    return network

# Обернуть модель Keras, чтобы она могла
# использоваться библиотекой scikit-learn
neural_network = KerasClassifier(build_fn=create_network, verbose=0)

# Создать гиперпараметрическое пространство
epochs = [5, 10]
batches = [5, 10, 100]
optimizers = ["rmsprop", "adam"]

# Создать словарь вариантов гиперпараметров
hyperparameters = dict(optimizer=optimizers, epochs=epochs,
                       batch_size=batches)

# Создать объект решеточного поиска
grid = GridSearchCV(estimator=neural_network, param_grid=hyperparameters)

# Выполнить подгонку объекта решеточного поиска
grid_result = grid.fit(features, target)

Using TensorFlow backend.
```

Обсуждение

В рецептах 12.1 и 12.2 мы рассмотрели использование методов отбора модели, которые идентифицируют наилучшие гиперпараметры модели scikit-learn. В рецепте 20.12 мы узнали, что можем обернуть нейронную сеть, чтобы она могла использовать API библиотеки scikit-learn. В этом рецепте мы совмещаем эти два метода, чтобы идентифицировать наилучшие гиперпараметры нейронной сети.

Гиперпараметры модели очень важны и должны тщательно отбираться. Однако прежде чем мы вобьем себе в голову, что стратегии отбора модели, такие как решеточный поиск, являются неплохой идеей, мы должны понять, что если на тренировку нашей модели обычно уходит 12 часов или день, то этот процесс решеточного поиска может занять неделю или больше. Поэтому автоматическая гиперпараметрическая настройка нейронных сетей не является серебряной пулей, но при определенных обстоятельствах это полезный инструмент.

В нашем решении мы провели перекрестно-проверочный решеточный поиск по ряду вариантов оптимизационного алгоритма, количеству эпох и размеру пакета. Даже этот игрушечный пример занял несколько минут, но после того, как он будет завершен, для просмотра гиперпараметров нейронной сети с наилучшими результатами можно воспользоваться атрибутом `best_params_`:

```
# Взглянуть на гиперпараметры наилучшей нейронной сети
grid_result.best_params_
```

```
{'batch_size': 5, 'epochs': 5, 'optimizer': 'adam'}
```

20.14. Визуализация нейронных сетей

Задача

Требуется быстро визуализировать архитектуру нейронной сети.

Решение

Использовать функции `model_to_dot` либо `plot_model` библиотеки Keras (рис. 20.3):

```
# Загрузить библиотеки
from keras import models
from keras import layers
from IPython.display import SVG
from keras.utils.vis_utils import model_to_dot
from keras.utils import plot_model

# Инициализировать нейронную сеть
network = models.Sequential()

# Добавить полносвязный слой с активационной функцией ReLU
network.add(layers.Dense(units=16, activation="relu", input_shape=(10,)))
```

```
# Добавить полносвязный слой с активационной функцией ReLU
network.add(layers.Dense(units=16, activation="relu"))

# Добавить полносвязный слой с сигмоидальной активационной функцией
network.add(layers.Dense(units=1, activation="sigmoid"))

# Визуализировать сетевую архитектуру
SVG(model_to_dot(network, show_shapes=True).create(prog="dot",
                                                   format="svg"))
```

```
Using TensorFlow backend.
```

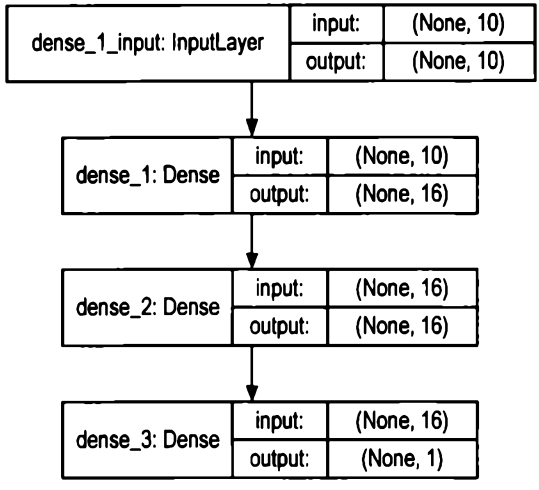


Рис. 20.3

В качестве альтернативы, если мы хотим сохранить визуализацию в виде файла, можно применить функцию `plot_model`:

```
# Сохранить визуализацию в виде файла
plot_model(network, show_shapes=True, to_file="network.png")
```

Обсуждение

Библиотека Keras предоставляет полезные функции для быстрой визуализации нейронных сетей (рис. 20.4). Если требуется отобразить нейронную сеть в блокноте Jupyter, то можно использовать функцию `model_to_dot`. **Параметр** `show_shapes` **показывает форму входных и выходных данных и может помочь при отладке. Для более простой модели можно задать** `show_shapes=True`:

```
# Визуализировать сетевую архитектуру
SVG(model_to_dot(network, show_shapes=False).create(prog="dot",
                                                    format="svg"))
```

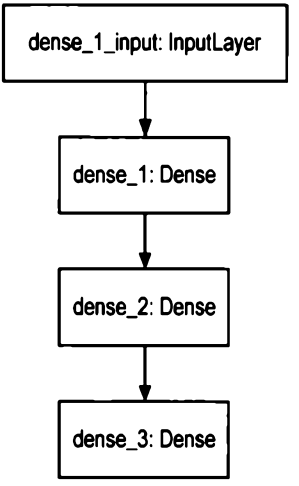


Рис. 20.4

20.15. Классификация изображений

Задача

Требуется расклассифицировать изображения с помощью сверточной нейронной сети.

Решение

Использовать библиотеку Keras для создания нейронной сети с хотя бы одним сверточным слоем:

```
# Загрузить библиотеки
import numpy as np
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers.convolutional import Conv2D, MaxPooling2D
from keras.utils import np_utils
from keras import backend as K

# Сделать значение цветового канала первым
K.set_image_data_format("channels_first")

# Задать начальное значение для ГПСЧ
np.random.seed(0)

# Задать информацию об изображении
channels = 1
```

```
height = 28
width = 28

# Загрузить данные и цель из набора данных MNIST рукописных цифр
(data_train, target_train), (data_test, target_test) = mnist.load_data()

# Реформировать тренировочные данные об изображениях в признаки
data_train = data_train.reshape(data_train.shape[0], channels, height, width)

# Реформировать тестовые данные об изображениях в признаки
data_test = data_test.reshape(data_test.shape[0], channels, height, width)

# Прошкалировать пиксельную интенсивность в диапазон между 0 и 1
features_train = data_train / 255
features_test = data_test / 255

# Преобразовать цель в кодировку с одним активным состоянием
target_train = np_utils.to_categorical(target_train)
target_test = np_utils.to_categorical(target_test)
number_of_classes = target_test.shape[1]

# Инициализировать нейронную сеть
network = Sequential()

# Добавить сверточный слой с 64 фильтрами, окном 5x5
# и активационной функций ReLU
network.add(Conv2D(filters=64,
                   kernel_size=(5, 5),
                   input_shape=(channels, width, height),
                   activation='relu'))

# Добавить максимально редуцирующий слой с окном 2x2
network.add(MaxPooling2D(pool_size=(2, 2)))

# Добавить отсеивающий слой
network.add(Dropout(0.5))

# Добавить слой для сглаживания входа
network.add(Flatten())

# Добавить полносвязный слой из 128 блоков
# с активационной функцией ReLU
network.add(Dense(128, activation="relu"))

# Добавить отсеивающий слой
network.add(Dropout(0.5))
```

```
# Добавить полносвязный слой
# с активационной функцией softmax
network.add(Dense(number_of_classes, activation="softmax"))

# Скомпилировать нейронную сеть
network.compile(
    loss="categorical_crossentropy", # Перекрестная энтропия
    optimizer="rmsprop",  # Распространение СКО
    metrics=["accuracy"]) # Точностный показатель результативности

# Натренировать нейронную сеть
network.fit(
    features_train,  # Признаки
    target_train,    # Цель
    epochs=2,        # Количество эпох
    verbose=0,       # Не печатать описание после каждой эпохи
    batch_size=1000, # Количество наблюдений на пакет
    validation_data=(features_test, target_test)) # Данные для оценивания
```

```
Using TensorFlow backend.
```

```
<keras.callbacks.History at 0x133f37e80>
```

Обсуждение

Сверточные нейронные сети (так называемые ConvNet-сети) — это популярный тип сети, который доказал свою очень высокую эффективность в компьютерном зрении (например, в распознавании кошек, собак, самолетов и даже хот-догов). Нейронные сети прямого распространения вполне можно использовать на изображениях, где каждый пиксел является признаком. Однако при этом мы сталкиваемся с двумя основными проблемами. Во-первых, нейронные сети прямого распространения не учитывают пространственную структуру пикселов. Например, изображение размером 10×10 пикселов можно преобразовать в вектор из 100 пикселов, и в этом случае сеть прямого распространения будет считать, что первый признак (например, значение пиксела) имеет ту же связь с 10-м признаком, что и с 11-м признаком. Однако на самом деле 10-й признак представляет пиксел на дальней стороне изображения как первый признак, в то время как 11-й признак представляет пиксел непосредственно под первым пикселом. Во-вторых, и в связи с этим, нейронные сети прямого распространения вместо локальных шаблонов заучивают глобальные связи в признаках. С практической точки зрения это означает, что нейронные сети прямого распространения не способны обнаруживать объект независимо от того, где он появляется на изображении. Например, представьте, что мы тренируем нейронную сеть распознавать лица, и эти лица могут появляться на изображении в любом месте — от правого верхнего участка до левого среднего до нижнего.

Сила сверточных нейронных сетей заключается в их способности справляться с обеими этими проблемами (и другими). Подробное описание сверточных нейрон-

ных сетей выходит далеко за рамки этой книги, но краткое объяснение будет полезно. Данные отдельного изображения содержат две или три размерности: высоту, ширину и глубину. Первые две должны быть очевидными, но последняя заслуживает объяснения. Глубина — это цвет пиксела. В полутоновых изображениях глубина всего одна (интенсивность пиксела) и поэтому изображение представлено матрицей. Однако в цветных изображениях цвет пиксела представлен несколькими значениями. Например, в изображении RGB цвет пиксела представлен тремя значениями, обозначающими красный, зеленый и синий цвета. Поэтому данные для изображения можно представить, как трехмерный тензор: *ширина* × *высота* × *глубина* (называемые *картами признаков*). В сверточных нейронных сетях свертку (не переживайте, если вы не знаете, что это значит) можно представить, как скользящее окно над пикселами изображения, охватывающее и отдельный пиксел, и его соседей. Затем она преобразовывает сырые данные изображения в новый трехмерный тензор, где первые две размерности имеют приблизительно ширину и высоту, в то время как третья размерность (которая содержала значения цвета) теперь представляет шаблоны, называемые *фильтрами* — например, острый угол или широкий градиент), к которому "принадлежит" этот пиксел.

Вторая важная идея для наших целей — это редуцирующие слои. Редуцирующие слои перемещают окно над нашими данными (хотя обычно они смотрят с интервалами только на каждый *n*-й пиксел; это называется перемещением с *шагом*) и уменьшают размер наших данных, тем или иным образом подытоживая окно. Наиболее распространенным методом является максимальное редуцирование, при котором на следующий уровень отправляется максимальное значение в каждом окне. Одна из причин для максимального редуцирования носит чисто практический характер; сверточный процесс создает чрезвычайно много параметров для заучивания, которые очень быстро могут стать громоздкими. Вторая причина интуитивно понятнее: максимальное редуцирование можно рассматривать как "уменьшение масштаба" изображения.

Здесь может быть полезен пример. Представьте, что у нас есть изображение, содержащее морду собаки. Первый сверточный слой может найти шаблоны, такие как края фигуры. Затем мы используем максимально редуцирующий слой для "уменьшения масштаба" и второй сверточный слой для поиска шаблонов, таких как собачьи уши. Наконец, мы используем еще один максимально редуцирующий слой, чтобы снова уменьшить масштаб, и заключительный сверточный слой, чтобы найти шаблоны, такие как морды собак.

Наконец, полносвязные слои часто используются в конце сети для классификации.

Хотя наше решение может выглядеть как целая куча строк кода, оно на самом деле очень похоже на наш бинарный классификатор из более ранней главы. В этом решении мы использовали знаменитый набор данных MNIST, который в машинном самообучении фактически является эталонным набором данных. Набор данных MNIST содержит 70 000 небольших изображений (28×28) рукописных цифр от 0 до 9. Этот набор данных помечен, чтобы мы знали фактическую цифру (т. е. класс) каждого небольшого изображения. Стандартное разбиение тренировка-тести-

рование состоит в использовании 60 000 изображений для тренировки и 10 000 для тестирования.

Мы реорганизовали данные в формат, ожидаемый сверточной сетью. В частности, мы использовали метод `reshape`, чтобы преобразовать данные наблюдений в ту форму, которую ожидает библиотека Keras. Затем мы прошкалировали значения, приведя их к диапазону между 0 и 1, т. к. результативность тренировки может пострадать, если значения наблюдения будут намного больше, чем параметры сети (которые инициализируются как малые числа). Наконец, мы преобразовали целевые данные в кодировку с одним активным состоянием, чтобы каждая цель наблюдения имела 10 классов, представляющих цифры 0–9.

Благодаря указанной предобработке данных изображения мы можем построить нашу сверточную сеть. Сначала мы добавляем сверточный слой и задаем количество фильтров и другие характеристики. Размер окна является гиперпараметром; однако размер 3×3 является стандартной практикой для большинства изображений, тогда как большие окна часто используются в больших изображениях. Во-вторых, мы добавляем максимально редуцирующий слой, подытоживая окрестные пикселы. В-третьих, мы добавляем отсеивающий слой, чтобы уменьшить вероятность переподгонки модели. В-четвертых, мы добавляем сглаживающий слой для преобразования входов в формат, который может быть использован полносвязным слоем. Наконец, мы добавляем полносвязные слои и выходной слой, который выполняет фактическую классификацию. Обратите внимание, что, поскольку это задача мультиклассовой классификации, мы в выходном слое используем активационную функцию softmax.

Следует отметить, что это довольно простая сверточная нейронная сеть. Часто можно увидеть гораздо более глубокую сеть с гораздо бóльшим количеством сверточных и максимально редуцирующих слоев, уложенных в ярусы.

20.16. Улучшение результативности с помощью расширения изображения

Задача

Требуется улучшить производительность сверточной нейронной сети.

Решение

Для получения более оптимальных результатов предварительно обработать изображения и расширить (аргументировать, "раздуть") данные с помощью класса-генератора `ImageDataGenerator`:

```
# Загрузить библиотеку
from keras.preprocessing.image import ImageDataGenerator
```

```
# Создать объект расширения изображения
augmentation = ImageDataGenerator(
    featurewise_center=True, # Применить отбеливание ZCA
    zoom_range=0.3,          # Случайно приблизить изображения
    width_shift_range=0.2,   # Случайно сместить изображения
    horizontal_flip=True,    # Случайно перевернуть изображения
    rotation_range=90)       # Случайно повернуть изображения

# Обработать все изображения из каталога 'raw/images'
augment_images = augmentation.flow_from_directory(
                                    "raw/images",  # Папка с изображениями
                                    batch_size=32, # Размер пакета
                                    class_mode="binary", # Классы
                                    save_to_dir="processed/images")
```

```
Using TensorFlow backend.
```

```
Found 12665 images belonging to 2 classes.
```

Обсуждение

Для начала приносим извинение — программный код этого решения у вас сразу не заработает, потому что у вас нет необходимых папок изображений. Вместе с тем наиболее распространенной ситуацией является наличие каталога изображений, и поэтому данный пример был сюда включен. Этот прием должен легко транслироваться на ваши собственные данные изображений.

Один из способов повышения результативности сверточной нейронной сети состоит в предобработке изображений. Мы обсудили ряд таких методов в *главе 8*; однако стоит отметить, что класс `ImageDataGenerator` библиотеки Keras содержит ряд базовых методов предобработки. Например, в нашем решении мы использовали `featurewise_center=True` для стандартизации пикселов по всем графическим данным.

Второй метод повышения результативности — добавление шума. Интересной особенностью нейронных сетей является то, что их результативность часто улучшается при добавлении в данные шумовой компоненты. Причина в том, что дополнительный шум может сделать нейронные сети более устойчивыми перед лицом реального шума и предотвратить их чрезмерную подгонку к данным.

Во время тренировки сверточных нейронных сетей для коллекции изображений в наблюдения можно добавлять шум путем случайного преобразования изображений различными способами, такими как переворот или увеличение изображений. Результативность модели могут заметно улучшить даже небольшие изменения. Для проведения этих преобразований можно использовать тот же самый класс `ImageDataGenerator`. В документации Keras (ссылка на которую приведена в *разд. "Дополнительные материалы для чтения"* далее) указан полный список доступных

преобразований. Однако в нашем примере приведен пример таких преобразований, как случайное увеличение, сдвиг, переворот и поворот изображения.

Важно отметить, что результатом метода `flow_from_directory` является объект-генератор Python. Это вызвано тем, что в большинстве случаев возникает необходимость обрабатывать изображения по требованию, когда они отправляются в нейронную сеть для тренировки. Если бы требовалось обработать все изображения сразу до начала тренировки, то мы могли бы просто обойти генератор в цикле.

Наконец, поскольку параметр `augment_images` является генератором, вместо метода `fit` во время тренировки нашей нейронной сети придется использовать метод `fit_generator`. Например,

```
# Натренировать нейронную сеть
network.fit_generator(
    augment_images,
    steps_per_epoch=2000, # Количество вызовов генератора для каждой эпохи
    epochs=5,             # Количество эпох
    validation_data=augment_images_test, # Генератор тестовых данных
    validation_steps=800) # Количество вызовов генератора
                          # для каждой тестовой эпохи
```

Обратите внимание, что все исходные изображения, используемые в этом рецепте, доступны в хранилище на Github (**http://bit.ly/2FvrVVq**)[1].

Дополнительные материалы для чтения

- Документация библиотеки Keras по предобработке изображений (**https://bit.ly/2KxVKne**).

20.17. Классификация текста

Задача

Требуется классифицировать текст.

Решение

Использовать рекуррентную нейронную сеть с долгой краткосрочной памятью:

```
# Загрузить библиотеки
import numpy as np
from keras.datasets import imdb
from keras.preprocessing import sequence
from keras import models
from keras import layers
```

[1] В прилагаемом к книге исходном коде примеров программ эти данные расположены в папке raw. — *Прим. перев.*

```
# Задать начальное значение для ГПСЧ
np.random.seed(0)

# Задать желаемое количество признаков
number_of_features = 1000

# Загрузить данные и вектор целей из данных с отзывами о кинофильмах
(data_train, target_train), (data_test, target_test) = imdb.load_data(
    num_words=number_of_features)

# Использовать дополнение и усечение,
# чтобы каждое наблюдение имело 400 признаков
features_train = sequence.pad_sequences(data_train, maxlen=400)
features_test = sequence.pad_sequences(data_test, maxlen=400)

# Инициализировать нейронную сеть
network = models.Sequential()

# Добавить встраивающий слой
network.add(layers.Embedding(input_dim=number_of_features, output_dim=128))

# Добавить слой длинной краткосрочной памяти с 128 блоками
network.add(layers.LSTM(units=128))

# Добавить полносвязный слой с сигмоидальной активационной функцией
network.add(layers.Dense(units=1, activation="sigmoid"))

# Скомпилировать нейронную сеть
network.compile(
    loss="binary_crossentropy", # Перекрестная энтропия
    optimizer="Adam",           # Оптимизация Adam
    metrics=["accuracy"]) # Точностный показатель результативности

# Натренировать нейронную сеть
history = network.fit(
    features_train,  # Признаки
    target_train,    # Цель
    epochs=3,        # Количество эпох
    verbose=0,       # Не печатать описание после каждой эпохи
    batch_size=1000, # Количество наблюдений на пакет
    validation_data=(features_test, target_test)) # Тестовые данные

Using TensorFlow backend.
```

Обсуждение

Часто возникает необходимость классифицировать текстовые данные. Хотя для этого можно применить вид сверточной сети, мы сосредоточимся на более популярном варианте: рекуррентной нейронной сети. Ключевой особенностью рекуррентных нейронных сетей является то, что информация в ней зациклена. Эта особенность позволяет рекуррентным нейронным сетям иметь тип памяти, который они могут использовать для лучшего понимания последовательных данных. Популярным типом рекуррентной нейронной сети является сеть с долгой краткосрочной памятью (long short-term memory), или LSTM-сеть, которая позволяет осуществлять обратное циркулирование информации в сети. За дополнительными сведениями обращайтесь к внешним ресурсам.

В этом решении используются данные с отзывами о кинофильмах из рецепта 20.3, и требуется натренировать LSTM-сеть предсказывать, являются ли эти отзывы положительными или отрицательными. Прежде чем мы сможем натренировать нашу сеть, требуется небольшая обработка данных. Наши текстовые данные поступают в виде списка целых чисел:

```
# Взглянуть на первое наблюдение
print(data_train[0])
```

```
[1, 14, 22, 16, 43, 530, 973, 2, 2, 65, 458, 2, 66, 2, 4, 173, 36, 256, 5, 25, 100,
43, 838, 112, 50, 670, 2, 9, 35, 480, 284, 5, 150, 4, 172, 112, 167, 2, 336, 385,
39, 4, 172, 2, 2, 17, 546, 38, 13, 447, 4, 192, 50, 16, 6, 147, 2, 19, 14, 22, 4,
2, 2, 469, 4, 22, 71, 87, 12, 16, 43, 530, 38, 76, 15, 13, 2, 4, 22, 17, 515, 17,
12, 16, 626, 18, 2, 5, 62, 386, 12, 8, 316, 8, 106, 5, 4, 2, 2, 16, 480, 66, 2, 33,
4, 130, 12, 16, 38, 619, 5, 25, 124, 51, 36, 135, 48, 25, 2, 33, 6, 22, 12, 215,
28, 77, 52, 5, 14, 407, 16, 82, 2, 8, 4, 107, 117, 2, 15, 256, 4, 2, 7, 2, 5, 723,
36, 71, 43, 530, 476, 26, 400, 317, 46, 7, 4, 2, 2, 13, 104, 88, 4, 381, 15, 297,
98, 32, 2, 56, 26, 141, 6, 194, 2, 18, 4, 226, 22, 21, 134, 476, 26, 480, 5, 144,
30, 2, 18, 51, 36, 28, 224, 92, 25, 104, 4, 226, 65, 16, 38, 2, 88, 12, 16, 283, 5,
16, 2, 113, 103, 32, 15, 16, 2, 19, 178, 32]
```

Каждое целое число в этом списке соответствует некоторому слову. Однако, поскольку каждый отзыв о кинофильме содержит неодинаковое количество слов, каждое наблюдение имеет разную длину. Поэтому, прежде чем мы сможем ввести эти данные в нашу нейронную сеть, нам нужно сделать все наблюдения одинаковой длины. Мы можем сделать это с помощью метода `pad_sequences`, который дополняет все данные наблюдений таким образом, чтобы они были одинакового размера. Мы можем увидеть это, если посмотрим на первое наблюдение после его обработки методом `pad_sequences`:

```
# Взглянуть на первое наблюдение
print(features_test[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   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   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   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   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
  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   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   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   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   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   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   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   1 591 202  14  31   6 717  10  10   2
  2   5   4 360   7   4 177   2 394 354   4 123   9   2   2   2  10  10
 13  92 124  89 488   2 100  28   2  14  31  23  27   2  29 220 468   8
124  14 286 170   8 157  46   5  27 239  16 179   2  38  32  25   2 451
202  14   6 717]
```

Далее мы используем один из самых перспективных методов в обработке естественного языка: встраивание слов (word embedding). В нашем двоичном классификаторе из рецепта 20.3 мы преобразовали наблюдения в кодировку с одним активным состоянием и использовали их в качестве входов в нейронную сеть. Однако на этот раз мы представим каждое слово как вектор в многомерном пространстве и дадим расстоянию между двумя векторами представлять сходство между словами. В библиотеке Keras это можно сделать, добавив встраивающий слой `Embedding`. Для каждого значения, отправленного в слой `Embedding`, он выведет вектор, представляющий это слово. Следующий слой — это наш слой LSTM с 128 блоками, который позволяет использовать информацию из более ранних входов в будущих, непосредственно обращаясь к последовательному характеру данных. Наконец, поскольку перед нами задача бинарной классификации (каждый отзыв либо положительный, либо отрицательный), мы добавляем полносвязный выходной слой с одним блоком и сигмоидальной активационной функцией.

Стоит отметить, что LSTM-сети представляют собой очень широкую тему и находятся в центре внимания многих исследований. Этот рецепт, хотя и будет полезен, как мы надеемся, далек от последнего слова в реализации LSTM-сетей.

Дополнительные материалы для чтения

- "Как работают рекуррентные нейронные сети и долгая краткосрочная память", учебное пособие (**https://bit.ly/2KCTxH8**).
- "Волшебство LSTM-нейросетей", блог-пост (**https://bit.ly/2GuR4fv**).
- "Минимальный пример LSTM-нейросети на чистом Python", репозиторий на Gitgub (**https://bit.ly/2IuviKq**).

Сохранение и загрузка натренированных моделей

Введение

В предыдущих 20 главах и примерно в 200 рецептах мы рассмотрели, как использовать сырые данные и машинное самообучение для создания эффективных предсказательных моделей. Однако, чтобы вся наша работа была стоящей, нам в конечном итоге нужно что-то сделать с нашей моделью, например, интегрировать ее с существующим программным приложением. Для достижения этой цели мы должны уметь сохранять наши модели после тренировки и загружать их, когда они необходимы в приложении. В этом и заключается суть нашей последней главы.

21.1. Сохранение и загрузка модели scikit-learn

Задача

Дана натренированная модель scikit-learn, и требуется ее сохранить и загрузить в другом месте.

Решение

Сохранить модель в качестве файла консервации:

```
# Загрузить библиотеки
from sklearn.ensemble import RandomForestClassifier
from sklearn import datasets
from sklearn.externals import joblib

# Загрузить данные
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Создать объект-классификатор случайных лесов
classifer = RandomForestClassifier()

# Натренировать модель
model = classifer.fit(features, target)
```

```
# Сохранить модель в качестве файла консервации
joblib.dump(model, "model.pkl")
```

```
['model.pkl']
```

После того как модель будет сохранена, мы можем использовать библиотеку scikit-learn в нашем конечном приложении (например, в веб-приложении), чтобы загрузить модель:

```
# Загрузить модель из файла
classifer = joblib.load("model.pkl")
```

И использовать ее для выполнения предсказаний:

```
# Создать новое наблюдение
new_observation = [[ 5.2, 3.2, 1.1, 0.1]]

# Предсказать класс наблюдения
classifer.predict(new_observation)
```

```
array([0])
```

Обсуждение

Первым шагом при использовании модели в производственной среде является сохранение этой модели в виде файла, который может быть загружен другим приложением или рабочим процессом. Мы можем сделать это, сохранив модель в виде файла pickle, специфического для Python формата встроенной одноименной библиотеки для консервации данных. В частности, для сохранения модели мы используем библиотеку joblib, которая расширяет библиотеку pickle для случаев, когда имеются большие массивы NumPy — распространенное явление для моделей, натренированных в библиотеке scikit-learn.

При сохранении моделей scikit-learn имейте в виду, что сохраненные модели могут быть несовместимы между версиями scikit-learn; поэтому может быть полезно включить используемую в модели версию библиотеки scikit-learn в имя файла:

```
# Импортировать библиотеку
import sklearn

# Получить версию библиотеки scikit-learn
scikit_version = joblib.__version__

# Сохранить модель в качестве файла консервации
joblib.dump(model, "model_{version}.pkl".format(version=scikit_version))
```

```
['model_0.11.pkl']
```

21.2. Сохранение и загрузка модели Keras

Задача

Дана натренированная модель Keras, и требуется ее сохранить и загрузить в другом месте.

Решение

Сохранить модель в файле формата HDF5:

```
# Загрузить библиотеки
import numpy as np
from keras.datasets import imdb
from keras.preprocessing.text import Tokenizer
from keras import models
from keras import layers
from keras.models import load_model

# Задать начальное значение для ГПСЧ
np.random.seed(0)

# Задать желаемое количество признаков
number_of_features = 1000

# Загрузить данные и вектор целей из данных с отзывами о кинофильмах
(train_data, train_target), (test_data, test_target) = imdb.load_data(
    num_words=number_of_features)

# Конвертировать данные отзывов о кинофильмах
# в матрицу признаков в кодировке с одним активным состоянием
tokenizer = Tokenizer(num_words=number_of_features)
train_features = tokenizer.sequences_to_matrix(train_data, mode="binary")
test_features = tokenizer.sequences_to_matrix(test_data, mode="binary")

# Инициализировать нейронную сеть
network = models.Sequential()

# Добавить полносвязный слой с активационной функцией ReLU
network.add(layers.Dense(
    units=16,
    activation="relu",
    input_shape=(number_of_features,)))

# Добавить полносвязный слой с сигмоидальной активационной функцией
network.add(layers.Dense(units=1, activation="sigmoid"))
```

```
# Скомпилировать нейронную сеть
network.compile(
    loss="binary_crossentropy", # Перекрестная энтропия
    optimizer="rmsprop",  # Распространение СКО
    metrics=["accuracy"]) # Точностный показатель результативности

# Train neural network
history = network.fit(
    train_features, # Признаки
    train_target,   # Вектор целей
    epochs=3,       # Количество эпох
    verbose=0,      # Вывода нет
    batch_size=100, # Количество наблюдений на пакет
    validation_data=(test_features, test_target)) # Тестовые данные

# Сохранить нейронную сеть
network.save("model.h5")
```

```
Using TensorFlow backend.
```

Затем мы можем загрузить модель в другом приложении либо для дополнительной тренировки:

```
# Загрузить нейронную сеть
network = load_model("model.h5")
```

Обсуждение

В отличие от библиотеки scikit-learn, в библиотеке Keras не рекомендуется сохранять модели с помощью встроенной библиотеки pickle. Вместо этого модели сохраняются как файл HDF5. Файл HDF5 содержит все необходимое для того, чтобы не только загрузить модель для выполнения предсказаний (т. е. архитектуру и натренированные параметры), но и перезапустить процесс тренировки (т. е. настройки потери и оптимизатора и текущее состояние).

Предметный указатель

Л

М

Н

О

П

Р

С

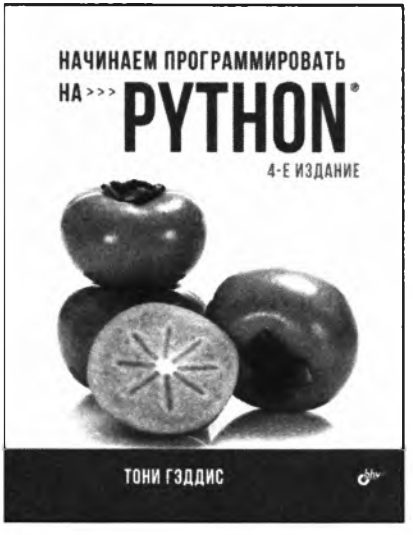
www.bhv.ru

Гэддис Т.

Начинаем программировать на Python, 4-е изд.

Отдел оптовых поставок

E-mail: opt@bhv.spb.su



- Краткое введение в компьютеры и программирование
- Ввод, обработка и вывод данных
- Управляющие структуры и булева логика
- Структуры с повторением и функции
- Файлы и исключения
- Списки и кортежи
- Строковые данные, словари и множества
- Классы и объектно-ориентированное программирование
- Наследование и рекурсия
- Функциональное программирование

В книге изложены принципы программирования, с помощью которых вы приобретете навыки алгоритмического решения задач на языке Python, даже если у вас нет опыта программирования. Для облегчения понимания сути алгоритмов широко использованы блок-схемы, псевдокод и другие инструменты. Приведено большое количество сжатых и практичных примеров программ. В каждой главе предложены тематические задачи с пошаговым анализом их решения.

Отличительной особенностью издания является его ясное, дружественное и легкое для понимания изложение материала.

Книга идеально подходит для вводного курса по программированию и разработке программного обеспечения на языке Python.

Тони Гэддис — ведущий автор всемирно известной серии книг «Начинаем программировать...» (Starting Out With) с двадцатилетним опытом преподавания курсов информатики в колледже округа Хейвуд, шт. Северная Каролина, удостоен звания «Преподаватель года», лауреат премии «Педагогическое мастерство».

www.bhv.ru

Прохоренок Н., Дронов В.

Python 3. Самое необходимое, 2-е изд.

Отдел оптовых поставок:
e-mail: opt@bhv.spb.su

Быстро и легко осваиваем Python — самый стильный язык программирования

- Основы языка Python 3
- Утилита pip
- Работа с файлами и каталогами
- Доступ к данным SQLite и MySQL
- Pillow и Wand: работа с графикой
- Получение данных из Интернета
- Библиотека Tkinter
- Разработка оконных приложений
- Параллельное программирование
- Потоки
- Примеры и советы из практики

В книге описан базовый синтаксис языка Python 3: типы данных, операторы, условия, циклы, регулярные выражения, встроенные функции, объектно-ориентированнное программирование, обработка исключений, часто используемые модули стандартной библиотеки и установка дополнительных модулей с помощью утилиты pip. Даны основы SQLite, описан интерфейс доступа к базам данных SQLite и MySQL, рассказано об использовании ODBC для доступа к данным. Рассмотрена работа с изображениями с помощью библиотек Pillow и Wand, получение данных из Интернета, разработка оконных приложений с помощью библиотеки Tkinter, параллельное программирование и работа с архивными файлами различных форматов. Книга содержит более двухсот практических примеров, помогающих начать программировать на языке Python самостоятельно. Весь материал тщательно подобран, хорошо структурирован и компактно изложен, что позволяет использовать книгу как удобный справочник.

Прохоренок Николай Анатольевич, профессиональный программист, имеющий большой практический опыт создания и продвижения динамических сайтов с использованием HTML, JavaScript, PHP, Perl и MySQL. Автор книг «HTML, JavaScript, PHP и MySQL. Джентльменский набор Web-мастера», «Python 3 и PyQt 5. Разработка приложений», «Основы Java», «OpenCV и Java. Обработка изображений и компьютерное зрение» и др.

Дронов Владимир Александрович, профессиональный программист, писатель и журналист, работает с компьютерами с 1987 года. Автор более 20 популярных компьютерных книг, в том числе «HTML, JavaScript, PHP и MySQL. Джентльменский набор Web-мастера», «Python 3 и PyQt 5. Разработка приложений», «Laravel. Быстрая разработка современных динамических Web-сайтов на PHP, MySQL, HTML и CSS», «Angular 4. Быстрая разработка сверхдинамических Web-сайтов на TypeScript и PHP» и книг по продуктам Adobe Flash и Adobe Dreamweaver различных версий. Его статьи публикуются в журналах «Мир ПК» и «ИнтерФейс» (Израиль) и интернет-порталах «IZ City» и «TheVista.ru».

O'REILLY®

Машинное обучение с использованием Python. Сборник рецептов

Книга содержит около 200 рецептов, которые помогут решить задачи машинного обучения, возникающие в повседневной работе практикующего специалиста, такие как загрузка и обработка текстовых или числовых данных, отбор модели, уменьшение размерности и многие другие. Рассмотрена работа с языком Python и его библиотеками, в том числе pandas и scikit-learn. Решения всех задач сопровождаются подробными объяснениями. Каждый рецепт содержит программный код, который можно скопировать и опробовать на игрушечном наборе данных (toy dataset). Затем этот код можно вставлять, объединять и адаптировать, создавая собственные приложения.

В книге вы найдете рецепты для:

- обработки числовых и категориальных данных, текста, изображений, дат и времени;
- уменьшения размерности с использованием методов выделения или отбора признаков;
- оценивания и отбора моделей;
- сохранения и загрузки натренированных моделей.

Научитесь решать задачи с использованием:

- векторов, матриц и массивов;
- линейной и логистической регрессии, деревьев, лесов и *k* ближайших соседей;
- опорно-векторных машин (SVM), наивных байесовых классификаторов, кластеризации и нейронных сетей.

Крис успешно воспользовался форматом книги рецептов. В ней есть простые и понятные учебные задачи для начинающих специалистов, ценная информация для самоподготовки перед собеседованием на должность аналитика данных, полезные для опытных профессионалов ссылки и многое другое. Книгу также удобно использовать как краткий, но всесторонний справочник.

Джастин Бозонье,
ведущий аналитик данных крупнейшего онлайн-сервиса по доставке еды из ресторанов Grubhub

Крис Элбон (Chris Albon) — аналитик данных и политолог с десятилетним опытом применения статистического обучения, искусственного интеллекта и разработки программного обеспечения для политических, социальных и гуманитарных проектов — от мониторинга выборов до оказания помощи в случае стихийных бедствий. В настоящее время является ведущим аналитиком данных в компании BRCK, продвигающей интернет-технологии на африканский рынок.

bhv®

191036, Санкт-Петербург.
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru
wɜt:

ISBN 978-5-9775-4056-8