Анализ социальных медиа на Python

Марко Бонцанини

[PACKT] PUBLISHING

ДМК ИЗДАТЕЛЬСТВО

Marco Bonzanini

Mastering Social Media Mining with Python

Acquire and analyze data from all corners of the social web with Python

[PACKT]
PUBLISHING
BIRMINGHAM – MUMBAI

Марко Бонцанини

Анализ социальных медиа на Python

Извлекайте и анализируйте данные из всех уголков социальной паутины на Python

ДМК Пресс
издательство
Москва, 2018

УДК 316.77:004.738.5:004.438Python
ББК 60.52с5
Б81

Бонцанини М.
Б81 Анализ социальных медиа на Python / пер. с анг. А. В. Логунова. – М.: ДМК Пресс, 2018. – 288 с.: ил.

ISBN 978-5-97060-574-5

Язык программирования Python является оптимальным выбором для исследователей-аналитиков, поскольку позволяет создавать прототипы, визуализировать и анализировать наборы данных малого и среднего размера. Бесчисленное количество предприятий обращается к Python для решения задач, связанных с выявлением особенностей поведения потребителей и превращением исходных данных в действенную информацию о клиентах. Настоящая книга рассказывает, как с помощью научного инструментария Python получать и анализировать данные из наиболее популярных сетей, таких как Facebook, Twitter, Stack Exchange и др. В русскоязычное издание добавлено приложение об анализе данных из сети «ВКонтакте».

Издание предназначено для специалистов по анализу данных, а также будет полезно всем разработчикам на Python, желающим извлекать коммерческую пользу из социальных сетей.

УДК 316.77:004.738.5:004.438Python
ББК 60.52с5

Copyright © Packt Publishing 2016. First published in the English language under the title «Mastering Social Media Mining with Python – (9781783552016)».

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN 978-1-78646-689-1 (анг.)
ISBN 978-5-97060-574-5 (рус.)

Copyright © 2016 Packt Publishing
© Оформление, издание, перевод, ДМК Пресс, 2018

Содержание

Об авторе ... 8

О рецензенте ... 9

Предисловие ... 10

Глава 1. Социальные медиа, социальные данные и Python ... 21
Начало ... 21
Социальные медиа – проблемы и возможности ... 22
Возможности ... 23
Проблемы ... 25
Технология анализа социальных данных ... 28
Инструменты Python для науки о данных ... 31
Настройка среды разработки Python ... 32
Эффективный анализ данных ... 35
Машинное обучение ... 39
Обработка естественного языка ... 43
Анализ социальных сетей ... 48
Визуализация данных ... 49
Обработка данных в Python ... 51
Создание составных конвейеров данных ... 53
Резюме ... 54

Глава 2. Твиттер – хештеги, темы и временные ряды ... 55
Начало работы ... 55
Twitter API ... 56
Ограничение частоты запросов ... 56
Поиск и потоковая обработка ... 57
Выборка данных из Twitter ... 58
Получение твитов из ленты ... 60
Структура твита ... 62
Применение потокового интерфейса Streaming API ... 66
Анализ твитов – сущности ... 69
Анализ твитов – текст ... 73
Анализ твитов – временные ряды ... 79
Резюме ... 82

Глава 3. Пользователи, читатели и сообщества в Twitter ... 83
Пользователи, друзья и читатели ... 83
Возвращаясь к интерфейсу Twitter API ... 83
Структура профиля пользователя ... 85
Загрузка профилей друзей и читателей ... 87
Анализ связей ... 89
Измерение степени влияния и вовлеченности ... 94
Анализ читателей ... 98
Анализ диалога ... 104

Привязка твитов к географической карте ... 107
От твитов к GeoJSON ... 108
Простота создания карт с Folium ... 110
Резюме ... 116

Глава 4. Сообщения, страницы и взаимодействие пользователей в Facebook ... 117
Интерфейс Facebook Graph API ... 117
Регистрация приложения ... 118
Аутентификация и безопасность ... 119
Доступ к Facebook Graph API из Python ... 121
Анализ сообщений ... 124
Структура сообщения ... 127
Частотно-временной анализ ... 127
Анализ страниц Facebook ... 129
Получение сообщений со страницы ... 131
Измерение степени вовлеченности ... 135
Визуализация сообщений в виде облака слов ... 141
Резюме ... 142

Глава 5. Тематический анализ в Google+ ... 144
Начало работы с Google+ API ... 144
Поиск в Google+ ... 147
Вывод результатов поиска в веб-интерфейсе ... 149
Декораторы в Python ... 150
Маршруты и шаблоны Flask ... 151
Заметки и действия со страницы Google+ ... 154
Анализ текстов и статистическая мера TF-IDF для заметок ... 157
Получение словосочетаний при помощи n-грамм ... 163
Резюме ... 163

Глава 6. Вопросы и ответы в сети Stack Exchange ... 165
Вопросы и ответы ... 165
Начало работы с Stack Exchange API ... 168
Поиск вопросов с тегами ... 170
Поиск пользователя ... 172
Обработка дампов данных из Stack Exchange ... 175
Классификация текстов по тегам в вопросах ... 180
Обучение с учителем и классификация текстов ... 180
Алгоритмы классификации ... 184
Оценка ... 187
Классификация текстов на данных из сети Stack Exchange ... 189
Встраивание классификатора в приложение реального времени ... 193
Резюме ... 198

Глава 7. Блоги, RSS, Википедия и обработка естественного языка ... 199
Блоги и обработка естественного языка ... 199
Получение данных из блогов и веб-сайтов ... 200
WordPress.com API ... 200

Blogger API ... 203
Каналы RSS и Atom ... 206
Получение данных из Википедии ... 207
Несколько слов о выборке данных из веба ... 210
Основы обработки естественного языка ... 210
Предварительная обработка текста ... 211
Извлечение информации ... 220
Резюме ... 225

Глава 8. Анализ других данных ... 226
Большое количество социальных API ... 226
Анализ видео на YouTube ... 226
Анализ открытого программного обеспечения на GitHub ... 231
Анализ сведений о местных предприятиях в Yelp ... 238
Создание собственного клиента на Python ... 243
Простой интерфейс для вызовов по протоколу HTTP ... 243
Резюме ... 245

Глава 9. Связанные данные и Семантическая паутина ... 247
Паутина данных ... 247
Словарь Семантической паутины ... 249
Микроформаты ... 252
Связанные данные и открытые данные ... 254
Среда описания ресурса RDF ... 255
Формат данных JSON-LD ... 256
Инициатива Schema.org ... 257
Анализ связей из DBpedia ... 258
Анализ географических координат ... 260
Извлечение геоданных из Википедии ... 260
Нанесение геоданных на карты Google Maps ... 263
Резюме ... 267

Приложение А. Анализ данных из социальной сети «ВКонтакте» ... 269
Анализ сообщества и определение его типичного участника ... 270
Определение центральных узлов социального графа ... 276
Отображение центральностей на графике ... 277
Прочие операции ... 279

Предметный указатель ... 281

Об авторе

Марко Бонцанини – исследователь-аналитик из Лондона (Соединенное Королевство). Имеет докторскую степень в области информационного поиска Лондонского университета королевы Марии. Специализируется на анализе текстовой информации и поисковых приложениях и многие годы с удовольствием занимался решением разнообразных задач управления информацией и науки о данных.

Ведет персональный блог на http://marcobonzanini.com, где обсуждает различные технические темы, главным образом связанные с языком Python, анализом текстовой информации и наукой о данных.

Когда Марко не занят работой над проектами на Python, он с удовольствием принимает участие в жизни сообщества, посещая конференции и неформальные встречи PyData с разработчиками, а также очень любит варить домашнее пиво.

Эта книга является результатом долгого процесса, который выходит далеко за рамки простой подготовки материалов книги. Многие так или иначе принимали участие в формировании конечного результата. В первую очередь я хотел бы поблагодарить команду издательста Packt Publishing, в особенности Сонали Вернекар (Sonali Vernekar) и Сиддхеш Салви (Siddhesh Salvi) за предоставленную возможность работать над этой книгой и за помощь в течение всего процесса. Доктор Вейай «Уэйн» Сюй (Dr. Wei Ai «Wayne» Xu) выступил в роли научного редактора настоящей книги и внес множество предложений по ее улучшению, за что ему отдельное спасибо. Многие коллеги и друзья в случайных разговорах, глубоких обсуждениях и во время работы над проектами в прошлом немало поспособствовали повышению качества материала, представленного в этой книге. Особо хотел бы поблагодарить доктора Мигеля Мартинеса-Альвареса (Dr. Miguel Martinez-Alvarez), Марко Кампана (Marco Campana) и Стефано Кампана (Stefano Campana). Я также рад быть частью лондонского сообщества PyData, группы умнейших людей, которые регулярно встречаются, чтобы поговорить о Python и науке о данных в располагающей для этого атмосфере. И напоследок, но не в последнюю очередь, хотел бы отметить Даниэлу (Daniela), поощрявшую меня в течение всего процесса, делясь со мной своими мыслями, предлагая улучшения и обеспечивая уютную обстановку по возвращении после работы.

О рецензенте

Вейай Уэйн Сюй – доцент в отделе коммуникаций Массачусетского университета (Амхерст, США) и сотрудничающий с институтом вычислительной социологии этого же университета. Ранее Сюй преподавал теорию сетей в институте сетевых наук Бостонского северо-восточного университета. Результаты его исследований в области онлайн-сообществ, «сарафанного радио» и социального капитала публиковались в различных авторитетных научных журналах. Сюй также оказывал содействие четырем национальным проектам в области стратегической коммуникации и общественного мнения. Кроме основной работы является соучредителем лаборатории данных под названием CuriosityBits Collective (http://www.curiositybits.org/).

Предисловие

За прошедшие несколько лет популярность социальных ресурсов существенно выросла благодаря тому, что все больше и больше пользователей стало обмениваться разнообразными видами информации их посредством. Компании используют социально-медийные платформы для продвижения своих брендов, профессионалы регистрируют общедоступные учетные записи и используют их для налаживания связей, а обычные пользователи занимаются обсуждением любых тем. Увеличение числа пользователей также означает увеличение объемов данных, ожидающих анализа.

Вы, читатель этой книги, вероятнее всего разработчик, инженер, аналитик, исследователь или студент, который хотел бы применить приемы анализа к данным, хранящимся на социальных ресурсах. С этой точки зрения вы, как практик в области анализа данных (или будущий практик), не будете испытывать недостатка в потенциальных возможностях и вызовах.

Книга «Анализ социальных медиа на Python» даст основные инструменты, которые помогут вам использовать это обилие данных в своих интересах. Знакомство с главными инструментами анализа данных на Python начнется с предоставления вводной информации о методах обработки естественного языка, машинного обучения, анализа социальных сетей и визуализации данных. Последующее пошаговое руководство по самым популярным социально-медийным платформам, включая Twitter, Facebook, Google+, Stack Overflow, Blogger, YouTube и др., расскажет, как получить доступ к данным этих сетей и как применять различные виды анализа для извлечения полезной информации из необработанных данных.

В настоящей книге затрагиваются три главных аспекта, а именно:

- **социально-медийные API**: каждая платформа обеспечивает доступ к своим данным по-своему. Понимание, как с ними взаимодействовать, даст возможность ответить на вопросы: *как получить данные?* и *какие данные можно получить?* Это важно потому, что невозможно анализировать данные, не имея к ним доступа. Каждая глава посвящена отдельной социально-медийной платформе и подробно рассказывает, как взаимодействовать с соответствующим API;
- **приемы анализа данных**: простое извлечение данных из API не имеет особой ценности. Следующий шаг – ответ на вопрос: *что можно делать с данными?* Каждая глава описывает понятия, которые помогут вам понять суть того или иного вида анализа данных и в чем заключается их ценность. В теоретическом плане мы лишь слегка затронем внешние аспекты, особо не вдаваясь в подробности, которые оставим академическим учебникам. Главная цель этой книги – привести практические примеры, которые дадут вам возможность легко приступить к работе;
- **инструменты Python для анализа данных**: поняв, что можно делать с данными, нам останется ответить на последний вопрос: *как это делается?* Python зарекомендовал себя как один из главных языков для анализа

данных. Простые синтаксис и семантика языка вместе с его богатой экосистемой поддержки научных вычислений обеспечивают пологую кривую обучения для новичков и всевозможные изощренные инструменты для экспертов. Настоящая книга знакомит с главными библиотеками для Python, используемыми в мире научных вычислений, в том числе NumPy, pandas, NetworkX, scikit-learn, NLTK и многими другими. Практические примеры имеют форму коротких сценариев, которые можно использовать (и возможно расширять) для выполнения различных и интересных видов анализа данных из социальных ресурсов, к которым вы имеете доступ.

Если исследование области, охватывающей эти три главные темы, представляет интерес, тогда эта книга для вас.

О чем рассказывается в книге

Глава 1 «*Социальные медиа, социальные данные и Python*» вводит главные понятия анализа данных с использованием языка Python применительно к социальным медиа. Здесь читатель найдет краткий обзор технологий машинного обучения, обработки естественного языка, анализа социальных сетей и визуализации данных, а также обсуждение главных инструментов Python для анализа данных и справочную информацию о настройке среды программирования на Python.

Глава 2 «*#Твиттер – хештеги, темы и временные ряды*» открывает практическое обсуждение анализа данных из социальной сети Twitter. После подготовки приложения, взаимодействующего с программным интерфейсом Twitter API, эта глава объясняет, как получить данные посредством потокового API и как выполнить частотный анализ хештегов и текста. Здесь также обсуждаются некоторые виды анализа временных рядов для поиска закономерностей распределения твитов во времени.

Глава 3 «*Пользователи, читатели и сообщества в Twitter*» продолжает обсуждение анализа данных из социальной сети Twitter, сосредоточив внимание на пользователях и взаимодействиях между ними. Эта глава показывает, как выявлять связи и диалоги между пользователями. Здесь объясняются такие интересные приемы, как кластеризация (сегментация) пользователей и измерение степени влияния и вовлеченности пользователей.

Глава 4 «*Сообщения, страницы и взаимодействия пользователей в Facebook*» посвящена социальной сети Facebook и ее пограммному интерфейсу Facebook Graph API. В данной главе подробно разбираются способы взаимодействия с Graph API, включая аспекты безопасности и конфиденциальности, и приводятся примеры извлечения сообщений из профиля пользователя и страниц в Facebook. Принципы анализа временных рядов и степени вовлеченности пользователей применяются к таким взаимодействиям пользователей, как комментарии, отметки «Нравится» и реакции.

Глава 5 «*Тематический анализ в Google+*» охватывает социальную сеть Google. Здесь подробно разбираются способы доступа к централизованной платформе Google и обсуждаются примеры поиска содержимого и пользователей в Google+. Эта глава также показывает, как встроить данные, поступающие из Google API, в собственное веб-приложение, сконструированное на основе микрофреймворка Flask.

Глава 6 «*Вопросы и ответы в сети Stack Exchange*» описывает порядок работы с вопросами и ответами и в качестве главного примера использует сеть Stack Exchange. Здесь читатель узнает, как искать пользователей и содержимое на различных сайтах этой сети, и прежде всего на Stack Overflow. Используя дампы данных данной сети для онлайн-обработки, эта глава знакомит с методами машинного обучения с учителем для классификации текстов и показывает, как встроить модель машинного обучения в приложение реального времени.

Глава 7 «*Блоги, RSS, Википедия и обработка естественного языка*» знакомит с приемами анализа текстовой информации. Всемирная паутина полна возможностей с точки зрения анализа текстов, и эта глава покажет, как взаимодействовать с различными источниками данных, такими как WordPress.com API, Blogger API, каналы RSS и Wikipedia API. На основе текстовых данных формализуются и расширяются основные понятия обработки естественного языка, кратко упоминаемые на протяжении всей книги. Затем читателю на специальных примерах демонстрируется процедура извлечения ссылок на сущности из произвольного текста.

Глава 8 «*Анализ других данных*» напоминает о многочисленных возможностях анализа данных, доступных за пределами наиболее распространенных социальных сетей. Демонстрирует примеры извлечения данных из YouTube, GitHub и Yelp, а также обсуждает создание собственного программного клиента, если конкретная платформа его не предоставляет.

Глава 9 «*Связанные данные и Семантическая паутина*» дает обзор Семантической паутины и связанных с ней технологий. Здесь рассматриваются темы связанных данных, микроформатов и RDF и предлагаются примеры анализа семантической информации из DBpedia и Википедии.

Что потребуется для работы с книгой

Прилагаемые к книге примеры программного кода предполагают наличие у читателя последней версии Python в Linux, macOS либо Windows. Програмный код был протестирован в Python 3.4.* и Python 3.5.*. Более старые версии (Python 3.3.* или Python 2.*) в явном виде не поддерживаются.

Глава 1 «*Социальные медиа, социальные данные и Python*» содержит инструкции по настройке локальной среды разработки и знакомит с кратким списком инструментов, используемых на протяжении всей книги. Вам также понадобятся некоторые важнейшие библиотеки для научных вычислений (NumPy, pandas и matplotlib), машинного обучения (scikit-learn), обработки естественного языка (NLTK) и анализа социальных сетей (NetworkX).

Кому адресована книга

Эта книга адресована в меру опытным разработчикам на языке Python, желающим использовать общедоступные API для сбора данных из социально-медийных платформ и выполнять статистический анализ для извлечения полезной информации из данных. Книга предполагает элементарное знакомство со стандартной библиотекой Python и приводит практические примеры, которые станут вам ориентиром для создания своего собственного аналитического проекта на основе социальных данных.

Соглашения

В этой книге используются разные стили оформления текста, которые разделяют различные виды информации. Ниже приводятся примеры этих стилей и поясняется их значение.

Элементы программного кода в тексте, имена таблиц в базах данных, имена папок и файлов, расширения файлов, пути к каталогам в файловой системе, фиктивные адреса URL, ввод пользователя и учетные записи в Twitter оформляются так: «Кроме того, атрибут `genre` представлен списком с переменным числом значений».

Блоки кода оформляются следующим образом:

```
from timeit import timeit
import numpy as np

if __name__ == '__main__':
    setup_sum = 'data = list(range(10000))'
    setup_np = 'import numpy as np;'
    setup_np += 'data_np = np.array(list(range(10000)))'
```

Когда потребуется привлечь ваше внимание к определенному фрагменту в блоке программного кода, он будет выделяться жирным:

```
Наберите ваш вопрос либо наберите "exit", чтобы выйти.
> What's up with Gandalf and Frodo lately? They haven't been in the Shire for a while...
Вопрос: What's up with Gandalf and Frodo lately? They haven't been in the Shire for a while...
Предсказанные метки: plot-explanation, the-lord-of-the-rings
```

Ввод или вывод в командной строке будет оформляться так:

```
$ pip install --upgrade [package name]
```

Новые термины и **важные слова** будут выделены жирным. Текст, отображаемый на экране, например в меню или в диалогах, будет оформляться так: «На странице конфигурации Keys and Access Tokens (Ключи и маркеры доступа) разработчик сможет найти ключ API и пароль, а также маркер доступа и секретный маркер доступа».

Так оформляются предупреждения и важные примечания.

Так оформляются советы и рекомендации.

Так оформляются дополнения от переводчика к тексту оригинальной книги.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф в разделе «Читателям – Файлы к книгам».

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Скачивание цветных иллюстраций

Мы также подготовили файл PDF с цветными иллюстрациями, диаграммами и скриншотами, которые в этой книге имеют черно-белое оформление. Цветные иллюстрации помогут вам лучше понять обсуждаемые темы. Вы можете загрузить файл с иллюстрациями по адресу: https://www.packtpub.com/sites/default/files/downloads/MasteringSocialMediaMiningWithPython_ColorImages.pdf.

Нарушение авторских прав

Пиратство в Интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Packt очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в Интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли принять меры.

Пожалуйста, свяжитесь с нами по адресу электронной почты dmkpress@gmail.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Вопросы

Вы можете присылать любые вопросы, касающиеся данной книги, по адресу dm@dmk-press.ru или questions@packtpub.com. Мы постараемся разрешить возникшие проблемы.

Комментарий переводчика

Весь материал настоящей книги приведен в соответствие с последними действующими версиями программных библиотек (время перевода книги — июль 2017 г.), дополнен свежей информацией и протестирован в среде Windows 10. Тестирование программного кода осуществлялось в Python 3.6.1.

Прилагаемый к книге адаптированный и скорректированный исходный код примеров лучше всего разместить в домашней папке пользователя (`/home/Ваши_проекты_Python` или `C:\Users\[ИМЯ_ПОЛЬЗОВАТЕЛЯ]\Ваши_проекты_Python`). Ниже приведена структура папки с прилагаемыми примерами:

`Chap01-Chap09`	Исходный код примеров на языке Python
`Дополнение`	Сценарий на Python для анализа данных из социальной сети «ВКонтакте»

Далее приведены особенности инсталляции некоторых используемых программных библиотек Python.

Особенности программного обеспечения

Библиотеки для Python обычно можно скачать из каталога библиотек Python PyPi (https://pypi.python.org/). Но имейте в виду, если предполагается использовать библиотеки SciPy и Scikit-learn в Windows, вам придется также установить библиотеку Numpy+MKL. Библиотека Numpy+MKL привязана к библиотеке Intel® Math Kernel Library и включает необходимые динамические библиотеки (DLL) в каталоге numpy.core. Загрузите соответствующие whl-файлы из репозитория (http://www.lfd.uci.edu/~gohlke/pythonlibs/) и установите (например, в 64-разрядной версии Windows и Python 3.6: `pip install numpy-1.13.0+mkl-cp36-cp36m-win_amd64.whl`). Соответствующая процедура установки описана ниже. Далее приводятся сведения о базовых библиотеках:

- **NumPy** – основная библиотека для научных вычислений на Python;
- **Matplotlib** – библиотека для работы с двумерными графиками. Требует наличия numpy и некоторых других пакетов;
- **Pandas** – инструмент для анализа структурных данных и временных рядов. Требует наличия пакета numpy и некоторых других;
- **Scikit-learn** – интегратор классических алгоритмов машинного обучения. Требует наличия `numpy+mkl`;
- **SciPy** – библиотека, используемая в математике, естественных науках и инженерном деле. Требует наличия `numpy+mkl`.

Факультативно:

- **Jupyter** – интерактивная вычислительная среда;
- **PyQt5** – библиотека инструментов для создания графического интерфейса, требуется для работы инструментальной среды программирования Spyder;
- **Spyder** – инструментальная среда программирования на Python.

Протокол установки библиотек

Среди прилагаемых примеров имеется файл `requirements.txt`, выполняющий установку всех используемых в книге библиотек в пакетном режиме одной командой:

```
pip3 install -r requirements.txt
```

Обратите внимание, что, прежде чем выполнить эту команду, необходимо перейти в каталог, где находится этот файл, например командой `cd`. Этот файл также можно найти в репозитории с примерами к книге по адресу: https://github.com/bonzanini/Book-SocialMediaMiningPython.

Если вы захотите установить более свежие версии библиотек и контролировать весь процесс установки, ниже предлагается список команд установки библиотек в том порядке, в каком они встречаются в тексте книги. В некоторых случаях библиотеки должны устанавливаться из whl-файлов, которые можно загрузить из репозитория (http://www.lfd.uci.edu/~gohlke/pythonlibs/).

```
pip3 install --upgrade pip или pip install -U pip
pip3 install virtualenv
pip3 install numpy
```

либо как whl: `pip3 install numpy-1.13.0+mkl-cp36-cp36m-win_amd64.whl`

```
pip3 install pandas
pip3 install scipy
```

либо как whl: `pip3 install scipy-0.19.0-cp36-cp36m-win_amd64.whl`

```
pip3 install Scikit-learn
```

либо как whl: `pip3 install scikit_learn-0.18.1-cp36-cp36m-win_amd64.whl`

```
pip3 install matplotlib
pip3 install nltk
pip3 install Pyro4
pip3 install gensim
pip3 install networkx
pip3 install tweepy
pip3 install folium
pip3 install facebook-sdk
pip3 install wordcloud
```

либо как whl: `pip3 install wordcloud-1.3.1-cp36-cp36m-win_amd64.whl`

```
pip3 install Pillow
pip3 install google-api-python-client
pip3 install flask
pip3 install beautifulsoup4
pip3 install py-stackexchange
pip3 install lxml
pip3 install requests
pip3 install feedparser
pip3 install wikipedia
pip3 install PyGithub
pip3 install yelp
pip3 install rdflib
pip3 install mf2py
pip3 install pykml
```

факультативно:

```
pip3 install jupyter
pip3 install pyqt5
pip3 install spyder
```

В зависимости от типа операционной системы, версии Python и версий библиотек версии whl-файлов для установки могут отличаться от приведенных выше, где показаны последние версии (по состоянию на август 2017) для 64-разрядной версии Windows и Python 3.6.1.

Перечень использованных примеров

Глава 1

```
python demo_gensim.py lord_of_the_rings.txt
```

Глава 2

```
python twitter_get_user_timeline.py marcobonzanini
python twitter_get_user_timeline.py PacktPub
python twitter_streaming.py \#RWC2015 \#RWCFinal rugby
python twitter_hashtag_frequency.py user_timeline_PacktPub.jsonl
python twitter_hashtag_stats.py user_timeline_PacktPub.jsonl
python twitter_mention_frequency.py user_timeline_PacktPub.jsonl
python twitter_term_frequency.py filename.jsonl
python twitter_time_series.py stream__RWC2015__RWCFinal_Rugby.jsonl
```

Глава 3

```
python twitter_get_home_timeline_toscreen.py
python twitter_get_home_timeline.py
python twitter_get_user_timeline.py marcobonzanini
python twitter_get_user_timeline.py PacktPub
python twitter_streaming.py \#RWC2015 \#RWCFinal rugby
python twitter_hashtag_frequency.py user_timeline_PacktPub.jsonl
python twitter_hashtag_stats.py user_timeline_PacktPub.jsonl
python twitter_mention_frequency.py user_timeline_PacktPub.jsonl
python twitter_term_frequency.py user_timeline_PacktPub.jsonl
python twitter_term_frequency_graph.py user_timeline_PacktPub.jsonl
python twitter_time_series.py stream__RWC2015__RWCFinal_Rugby.jsonl
python twitter_make_geojson.py --tweets stream__RWC2015__RWCFinal_Rugby.jsonl --geojson
rwc2015_final.geo.json
```

Глава 4

```
python twitter_get_user.py PacktPub
python twitter_get_user.py marcobonzanini
python twitter_followers_stats.py PacktPub
python twitter_followers_stats_set.py PacktPub
python twitter_followers_stats_numpy.py PacktPub
python twitter_influence.py marcobonzanini PacktPub
python twitter_cluster_users.py --filename users/marcobonzanini/followers.jsonl --k 5
--max-features 200 --max-ngram 3
python twitter_conversation.py stream__RWC2015__RWCFinal_Rugby.jsonl
python twitter_map_example.py --map example_map.htm
python twitter_map_basic.py --geojson rwc2015_final.geo.json --map rwc2015_final_tweets.html
python twitter_map_clustered.py --geojson rwc2015_final.geo.json --map rwc2015_final_tweets_
clustered.html
python facebook_my_profile.py
python facebook_get_friends.py
python facebook_get_my_posts.py
```

```
python facebook_get_my_posts_more_fields.py
python facebook_post_time_stats.py -f my_posts.jsonl
python facebook_get_page_info.py --page PacktPub
python facebook_get_page_posts.py --page PacktPub --n 500
python facebook_top_posts.py --page PacktPub
python facebook_top_posts_plot.py --page PacktPub
python facebook_posts_wordcloud.py --page PacktPub
```

Дополнение:

```
twitter_get_oldtweets_in_bulk.py
```

Глава 5

```
python gplus_search_example.py --query packt
python gplus_search_web_gui.py
python gplus_get_page_activities.py --page +packtpublishing --max-results 1000
python gplus_activities_keywords.py --file activities_+LarryPage.jsonl --keywords 5
```

Глава 6

```
python stack_search_keyword.py --tags "python;nosql" --n 10
python stack_search_user.py --name joel
python stack_search_user.py --name joel --sort creation --order asc
python stack_xml2json.py --xml movies.stackexchange_5.com --json movies.tags.jsonl
python stack_xml2json.py --xml movies.stackexchange_4.com --json movies.posts.jsonl --clean-post
python stack_classification_prepare_dataset.py --tags-file movies.tags.jsonl --posts-file
movies.posts.jsonl --output movies.questions4classification.jsonl  --min-df 10
python stack_classification_predict_tags.py --questions movies.questions4classification.jsonl
python stack_classification_predict_tags.py --questions movies.questions4classification.jsonl
--min-df 5
python stack_classification_save_model.py --questions movies.questions4classification.jsonl
--min-df 5 --output questions-svm-classifier.pickle
python stack_classification_user_input.py --model questions-svm-classifier.pickle
```

Глава 7

```
python blogs_wp_get_posts.py --domain marcobonzanini.com --output posts.marcobonzanini.com.
jsonl --posts 100
python blogs_blogger_get_posts.py --url http://googleresearch.blogspot.co.uk --posts 50
--output posts.googleresearch.jsonl
python blogs_rss_get_posts.py --rss-url http://feeds.bbci.co.uk/news/rss.xml --json rss.bbc.jsonl
python blogs_entities.py --entity London
```

Глава 8

```
python youtube_search_video_pagination.py --query python --n 50 --output videos.jsonl
python github_get_user.py --user bonzanini --get-repos
python github_search_user.py --query [здесь ваш запрос] --sort followers --order desc
python github_search_repos.py --query python --sort stars --order desc
python yelp_search_business.py --location London --search breakfast --limit 5
python yelp_search_business.py --location "San Francisco" --search "craft beer" --limit 5
```

Глава 9

```
python rdf_summarize_entity.py --entity "Python"
python rdf_summarize_entity.py --entity "Python_(programming_language)"
```

```
python micro_geo_wiki.py --url "https://en.wikipedia.org/wiki/London"
python micro_geo_wiki.py --url "https://en.wikipedia.org/wiki/List_of_United_States_cities_by_
population"
python micro_geo2kml.py --url "https://en.wikipedia.org/wiki/List_of_United_States_cities_by_
population" --output us_cities.kml --n 20
```

Установка библиотек Python из whl-файлов

Библиотеки для Python можно разрабатывать не только на чистом Python. Часто библиотеки пишутся на C (динамические библиотеки), и для них пишется обертка Python, или же библиотека пишется на Python, но для оптимизации узких мест часть кода пишется на C. Такие библиотеки получаются очень быстрыми, однако программисту на Python тяжелее установить их ввиду банального отсутствия соответствующих знаний или 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 диспетчера 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-0.19.0-cp36-cp36m-win_amd64.whl
```

Откройте окно командой строки и, выполнив команду `cd`, перейдите в каталог, где находится whl-файл. Затем просто скопируйте в команду выше имя whl-файла. В этом случае полный путь указывать не понадобится. Например:

```
pip install scipy-0.19.0-cp36-cp36m-win_amd64.whl
```

При выборе библиотеки важно, чтобы разрядности устанавливаемой библиотеки и интерпретатора совпадали. Пользователи Windows могут загрузить whl-файлы на веб-странице http://www.lfd.uci.edu/~gohlke/pythonlibs/ Кристофа Голька из Лаборатории динамики флуоресценции Калифорнийского университета в г. Ирвайн. Библиотеки там постоянно обновляются, и в архиве содержится все, что только может понадобиться.

Установка и настройка инструментальной среды Spyder

Spyder — это инструментальная среда для научных вычислений на языке Python (Scientific PYthon Development EnviRonment) для Windows, Mac OS X и Linux. Это простая, легковесная и бесплатная интерактивная среда разработки на Python, которая предлагает функционал, аналогичный среде разработки на MATLAB, включая готовые к использованию виджеты `PyQt5` и `PySide`: редактор исходного кода, редактор массивов данных `NumPy`, редактор словарей, консоли Python и IPython и многое другое.

Чтобы установить среду Spyder в Ubuntu Linux, используя системный диспетчер пакетов, достаточно выполнить всего одну команду:

```
sudo apt-get install spyder
```

Чтобы установить ее с использованием диспетчера библиотек `pip`:

```
sudo apt-get install python-qt5 python-sphinx
sudo pip install spyder
```

А чтобы обновить:

```
sudo pip install -U spyder
```

Установить Spyder в Fedora 25 можно командой:

```
dnf install python-spyder
```

Установка среды Spyder в Windows:

```
pip install spyder
```

Среда Spyder требует обязательной установки библиотеки `PyQt5`.

Глава 1

Социальные медиа, социальные данные и Python

Настоящая книга посвящена технологиям *анализа данных* из *социальных сетей* с использованием языка *Python*. Три выделенных ключевых термина в предыдущем предложении помогают определить целевую аудиторию настоящей книги: любой разработчик, инженер, аналитик, исследователь или студент, интересующийся исследованием области, где встречаются эти три темы.

Эта глава охватывает следующие темы:

- социальные медиа и социальные данные;
- процесс анализа данных из социальных сетей;
- настройка среды Python для разработки приложений;
- инструменты Python для обработки научных данных;
- обработка данных на Python.

Начало

Во втором квартале 2015 г. компания Facebook объявила, что число активных пользователей этой социальной сети превысило 1,5 миллиарда. В 2013 г. компания Twitter заявила, что ежедневно обрабатывает более 500 миллионов твитов. Читателям также будет интересно узнать, что в 2015 г. веб-сайт Stack Overflow объявил о том, что с момента его открытия было задано больше 10 миллионов вопросов.

Данные числа являются лишь верхушкой айсберга, учитывая экспоненциональный рост популярности социальных медиа по мере увеличения числа пользователей, обменивающихся информацией посредством различных платформ. Это обилие данных дает практикам анализа данных уникальные возможности. Цель настоящей книги состоит в том, чтобы помочь читателю овладеть приемами использования API социальных медиа для сбора данных, которые можно проанализировать при помощи инструментов Python, и получить интересные выводы о том, как пользователи взаимодействуют между собой.

Настоящая глава закладывает основы для начального обсуждения проблем и возможностей анализа данных из социальных медиа и знакомит с некоторыми инструментами Python, которые будут использоваться в последующих главах.

Социальные медиа – проблемы и возможности

В традиционных средствах массовой информации пользователи, как правило, являются простыми потребителями. Информация течет в одном направлении: от публикатора к пользователям. Социальные медиа разрушают эту модель, позволяя каждому пользователю одновременно быть потребителем и публикатором. По этой теме написано большое количество академических статей, в которых авторы пытались определить, что на самом деле означает термин «*социальные медиа*», например Андреас М. Каплан (Andreas M. Kaplan) и Майкл Хэенлейн (Michael Haenlein) в своей статье «*Users of the world, unite! The challenges and opportunities of Social Media*» («Пользователи всех стран, объединяйтесь! Проблемы и возможности социальных медиа»). Разные социально-медийные платформы объединяют следующие аспекты:

- интернет-приложения;
- информация, генерируемая пользователями;
- сетевые взаимодействия.

По своей сути социальные медиа – это интернет-приложения. Ясно, что прогресс в области интернет- и мобильных технологий способствовал распространению социальных медиа. Посредством мобильного телефона можно мгновенно соединиться с социально-медийной платформой, опубликовать свой материал или разузнать последние новости.

Социально-медийные платформы управляются информацией, генерируемой пользователями. В противоположность традиционной модели, в них потенциальным публикатором является каждый пользователь. И, что более важно, любой пользователь может взаимодействовать с любым другим пользователем, делясь сведениями, оставляя комментарии или выражая положительную оценку посредством кнопки Like (Нравится) (то есть «лайкая», «плюсуя» или щелкая на пиктограмме с большим пальцем вверх).

Социальные медиа основаны на сетевых взаимодействиях. Как описано выше, суть социальных медиа заключается во взаимодействии пользователей между собой. Быть на связи – вот центральная идея большей части социально-медийных платформ, и информация, которую вы получаете посредством своей *ленты новостей*, обусловлена вашими связями.

С учетом этих ключевых особенностей, характерных для большинства платформ, социальные медиа используются, чтобы:

- оставаться на связи с друзьями и семьей (например, посредством Facebook);
- вести микроблог и узнавать последние новости (например, посредством Twitter);
- оставаться на связи с профессиональной сетью (например, посредством LinkedIn);
- делиться мультимедийным материалом (например, посредством Instagram, YouTube, Vimeo и Flickr);
- находить ответы на вопросы (например, посредством Stack Overflow, Stack Exchange и Quora);
- находить и организовывать интересующую информацию (например, посредством Pinterest).

Цель этой книги – дать ответ на центральный вопрос: как извлекать полезные знания из данных, поступающих из социальных медиа? А теперь отступим на шаг назад и определим, что такое *знание* и что понимать под *полезностью*.

Традиционные определения понятия «знание» пришли из информатики. Обычно «знание» изображается как часть пирамиды, иногда называемой иерархией знаний, в основании которой лежат данные, на среднем уровне – информация и наверху – знание. Эта пирамида изображена на рис. 1.1.

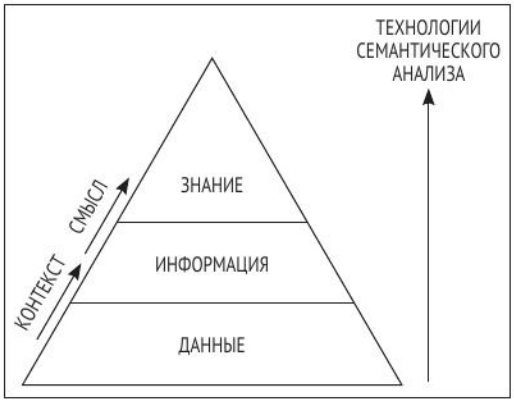

Рис. 1.1 ❖ От исходных данных к семантическому знанию

Восхождение на пирамиду отражает процесс извлечения знания из исходных данных. Путь от исходных данных к чистому знанию лежит через интеграцию контекста и смысла. По мере восхождения вверх по пирамиде применение технологий анализа помогает получать все более глубокое понимание исходных данных и, что еще более важно, понимание пользователей, которые производят эти данные. Другими словами, увеличивается полезность знаний.

В данном контексте полезное знание означает *эффективное* знание, то есть знание, которое позволяет лицу, принимающему решения, реализовать бизнес-стратегию. Прочтя эту книгу, вы поймете ключевые принципы извлечения ценности из социальных данных. Понимание, как пользователи взаимодействуют посредством социально-медийных платформ, является одним из ключевых аспектов в этом процессе.

Следующие ниже разделы формулируют некоторые проблемы и возможности анализа данных из социально-медийных платформ.

Возможности

Ключевой потенциал систем анализа данных кроется в *извлечении полезных выводов* из данных. Цель данного процесса состоит в том, чтобы ответить на интересные (а иногда и трудные) вопросы, используя технологии анализа данных, и достичь более полного понимания конкретной предметной области. Например, розничный онлайн-магазин может применить анализ данных, чтобы понять, как их потребители выбирают товары. По результатам анализа они смогут рекомен-

довать клиентам товары в зависимости от их покупательских привычек (например, пользователи, покупающие изделие А, также покупают изделие В). В общем случае это улучшает качество обслуживания клиентов и их удовлетворенность, что в итоге поможет нарастить продажи.

Многие организации в разных сферах деятельности могут применять технологии анализа данных для совершенствования своего бизнеса. Вот только некоторые примеры:

- банковское дело:
  - выявление лояльных клиентов с целью предложения им эксклюзивных программ;
  - распознавание мошеннических схем с целью уменьшения затрат;
- медицина:
  - изучение поведения пациентов с целью предсказания посещений хирурга;
  - помощь врачам в идентификации успешных способов лечения в зависимости от анамнеза пациента;
- розничная торговля:
  - изучение схем поведения во время покупок с целью улучшения качества обслуживания клиентов;
  - увеличение эффективности рекламных кампаний за счет более точной направленности;
  - анализ оперативных данных транспортных потоков для поиска самого быстрого маршрута доставки продуктов питания.

Как все это транслируется в область социальных медиа? По сути дела, вопрос состоит в том, как пользователи делятся своими данными посредством социально-медийных платформ. Организации больше не ограничиваются анализом данных, собираемых непосредственно, поскольку теперь они обладают доступом к гораздо большему объему данных.

Решение задачи сбора таких данных реализуется посредством удобных и универсальных API. Социально-медийные платформы применяют широко распространенную практику, предлагая разработчикам веб-ориентированные прикладные программные интерфейсы (Web API) для встраивания функций платформы в их приложения.

Прикладной программный интерфейс. Прикладной программный интерфейс (Application Programming Interface, API), или просто программный интерфейс, – это комплекс определений процедур и протоколов, описывающих поведение программного компонента, такого как библиотека или удаленная служба, в терминах разрешенных действий, входных и выходных данных. Используя сторонние API, разработчикам не нужно беспокоиться о внутреннем устройстве компонентов, а только о том, как его использовать.

Термином Web API мы обозначаем веб-службу, которая предоставляет ряд URI-адресов (возможно, после прохождения аутентификации) для доступа к данным. Для разработки таких API широко используется архитектурный подход, который называется **передачей состояния представления** (Representational State Transfer, REST). Программный интерфейс, реализующий архитектуру REST, называется **RESTful API**. Однако мы предпочитаем использовать более общий термин «Web API», потому что многие из существующих API не всегда строго соответствуют

принципам архитектуры REST. Для целей настоящей книги глубокое понимание архитектуры REST не требуется.

Проблемы

Некоторые проблемы анализа социальных медиа обусловлены проблемами анализа данных вообще.

Занимаясь обработкой социальных данных, мы часто имеем дело с **большими данными**. Чтобы понять значение больших данных и проблем, связанных с ними, можно вернуться к традиционному их определению (которое в 2001 г. дал Дуг Лэни (Doug Laney) в своем исследовании «*3D Data Management: Controlling Data Volume, Velocity and Variety, Doug Laney*»), известном также как «*Три V больших данных: объем (volume), разнообразие (variety) и скорость (velocity)*». За прошедшие годы это определение было дополнено новыми V, прежде всего проблемой *ценности* (*value*), поскольку получение добавленной стоимости является для компаний одной из главных целей использования больших данных. Первое из трех исходных V – *объем* (*volume*) – означает, что данные могут храниться на нескольких разных машинах. Это, конечно, требует другой инфраструктуры, чем для обработки малых данных (например, умещающихся в оперативной памяти). Более того, объем также связан со *скоростью* (*velocity*) в том смысле, что объем данных увеличивается настолько быстро, что понятие *большой* превращается в подвижную цель. Наконец, проблема *разнообразия* (*variety*) обусловлена тем, что данные могут иметь разные форматы и структуры, часто несовместимые между собой и с различной семантикой. Данные из социальных медиа соответствуют всем трем V.

Появление больших данных подтолкнуло процесс разработки новых подходов к хранению данных в системах под общим названием NoSQL. Это обобщенный (зонтичный) термин для многочисленных парадигм баз данных, имеющих одну общую черту, которая заключается в уходе от традиционных реляционных схем хранения данных и продвижении взамен нее конструкции с динамической схемой. Данная книга посвящена вовсе не технологиям баз данных, но из этой области для нас могут оказаться ценными приемы работы со смесью структурированных, неструктурированных и полуструктурированных данных. Выражение «*структурированные данные*» относится к информации, имеющей определенную организацию и обычно представленной в табличной форме. В этом случае связь с реляционными базами данных очевидна. Следующая таблица демонстрирует пример структурированных данных, которые описывают книги, проданные книжным магазином:

Заголовок	Жанр	Цена
1984	Политическая фантастика	12
Война и мир	Военный роман	10

Такого рода данные структурированы – каждый элемент имеет ясную организацию, а именно три атрибута: «заголовок», «жанр» и «цена».

Полной противоположностью структурированным данным являются данные неструктурированные, то есть информация, не организованная в соответствии с некоторой предопределенной моделью данных. Неструктурированные данные

обычно представлены в простом текстовом виде, например в виде электронных писем, документов, сообщений в социальных сетях и т. д. Представленные в этой книге методы могут использоваться для извлечения из неструктурированных данных шаблонов, с тем чтобы обеспечить некоторую структуру.

Между структурированными и неструктурированными данными находятся полуструктурированные данные, то есть данные с гибкой или не полностью предопределенной структурой. Подобные структуры иногда называют самоописательными. Типичным примером полуструктурированного формата данных может служить формат JSON (JavaScript Object Notation, то есть формат записи объектов на JavaScript). Как следует из названия, формат JSON заимствует правила определения данных из языка программирования JavaScript. Этот формат приобрел чрезвычайную популярность благодаря широкому применению для обмена данными между клиентом и сервером в веб-приложениях. Ниже приводится пример данных в формате JSON – расширенной версии предыдущих данных о книгах:

```
[
  {
    "заголовок": "1984",
    "цена": 12,
    "автор": "Джордж Оруэлл",
    "жанр": ["Политическая фантастика", "Социальная фантастика"]
  },
  {
    "заголовок": "Война и мир",
    "цена": 10,
    "жанр": ["Исторический роман", "Мелодрама", "Военный роман"]
  }
]
```

Обратите внимание, что первая книга имеет атрибут `автор`, который отсутствует во второй книге. Кроме того, атрибут `жанр` здесь имеет вид списка с переменным числом значений. Оба этих аспекта обычно не допускаются в полностью структурированном (реляционном) формате данных, но прекрасно уживаются в JSON и с полуструктурированными данными в целом.

Обсуждение структурированных и неструктурированных данных сводится к обработке разных форматов и принятию ряда подходов к решению проблемы целостности. Выражение «**целостность данных**» используется, чтобы очертить круг проблем, возникающих из-за присутствия искаженных, непоследовательных или неполных данных.

Непоследовательные и неполные данные часто встречаются при анализе сведений, генерируемых пользователем, и такие данные требуют особенно пристального внимания. Редко можно наблюдать пользователей, которые делятся своими данными систематически, почти в формальной манере. Наоборот, социальные медиа часто имеют неформальную и порой противоречивую организацию. Например, если пользователь захочет оставить жалобу на продукт на странице производителя в Facebook, он сначала должен выставить отметку «*Нравится*» самой странице, что никак не соответствует неудовлетворенности производителем из-за низкокачественного продукта. Понимание, как пользователи взаимодействуют в социально-медийных платформах, играет важнейшую роль при разработке анализа.

Для разработки приложений анализа данных также требуется рассмотреть вопросы, связанные с **доступом к данным**, в частности когда политика компании в итоге приводит к нехватке данных для проведения анализа. Говоря проще, к ситуации, когда данные не всегда открыто доступны. В предыдущем абзаце мы уже обсудили, что проблема нехватки данных при анализе социальных медиа не настолько важна, по сравнению с другими корпоративными средами, поскольку большинство социально-медийных платформ предлагает хорошо спроектированные универсальные API, открывающие доступ к данным, в которых мы нуждаемся. Доступность данных, разумеется, по-прежнему зависит от того, как пользователи обмениваются ими и как они предоставляют доступ. Например, пользователи Facebook могут задавать уровень детализации сведений, отображаемых в их общедоступном профиле, и ограничивать доступность своей информации только кругом своих друзей. Информация о профиле, в частности дне рождения, текущем месте проживания и трудовой биографии (и многом другом), может быть отмечена как частная или как общедоступная. Аналогично, когда мы пытаемся получить доступ к такого рода данным через Facebook API, пользователи, подписавшиеся на наше приложение, имеют возможность открывать нам доступ только к ограниченному подмножеству данных.

Еще одна, и последняя общая, проблема анализа данных кроется в понимании самого процесса анализа и способности объяснить его результаты. Другими словами, не всегда получается правильно сформулировать вопрос перед началом анализа данных. В большинстве случаев **научно-исследовательские и опытно-конструкторские** (R&D) процессы управляются результатами первичного анализа, то есть, чтобы понять, как подступиться к проблеме, прежде нужно начать ее анализировать. Связанное с этим понятие в статистике описывается фразой «*корреляция не подразумевает причинную обусловленность*». Можно применить большое количество статистических исследований и выявить корреляцию между двумя переменными, то есть что два события происходят вместе, но этого недостаточно, чтобы установить причинно-следственную связь в любом направлении. Забавные примеры причудливых корреляций можно найти по всей Всемирной паутине. Популярный случай был опубликован в одном из самых уважаемых медицинских журналов «New England Journal of Medicine», который показал интересную корреляцию между количеством шоколада, потребленного на душу населения в расчете на страну, и числом Нобелевских лауреатов («*Chocolate Consumption, Cognitive Function, and Nobel Laureates*», *Франц Х. Мессерли* (*Franz H. Messerli*), *2012*).

Выполняя первичный анализ, важно помнить, что корреляция (два события, происходящих вместе) является двунаправленной, тогда как причинная обусловленность (событие А вызвало событие В) – однонаправленной. Это шоколад делает вас более умным? Или же умные люди любят шоколад больше, чем средний человек? Случайно ли совпадение этих двух событий? Или же существует третья, пока невидимая переменная, которая играет некую роль в корреляции? Просто наблюдать корреляцию недостаточно, чтобы описать причинную обусловленность, но это часто интересная отправная точка, чтобы задать важные вопросы о данных, которые мы наблюдаем.

Следующий раздел обобщает способы взаимодействий наших приложений с социально-медийным API и выполнения требуемого анализа.

Технология анализа социальных данных

Прежде чем перейти к обсуждению подробностей в последующих главах, в данном разделе мы в общих чертах обсудим процесс создания приложения анализа социальных данных.

В общем случае процесс можно разбить на следующие этапы:

- аутентификация;
- сбор данных;
- очистка и предварительная обработка данных;
- моделирование и анализ;
- представление результатов.

На рис. 1.2 изображена схема данного процесса:

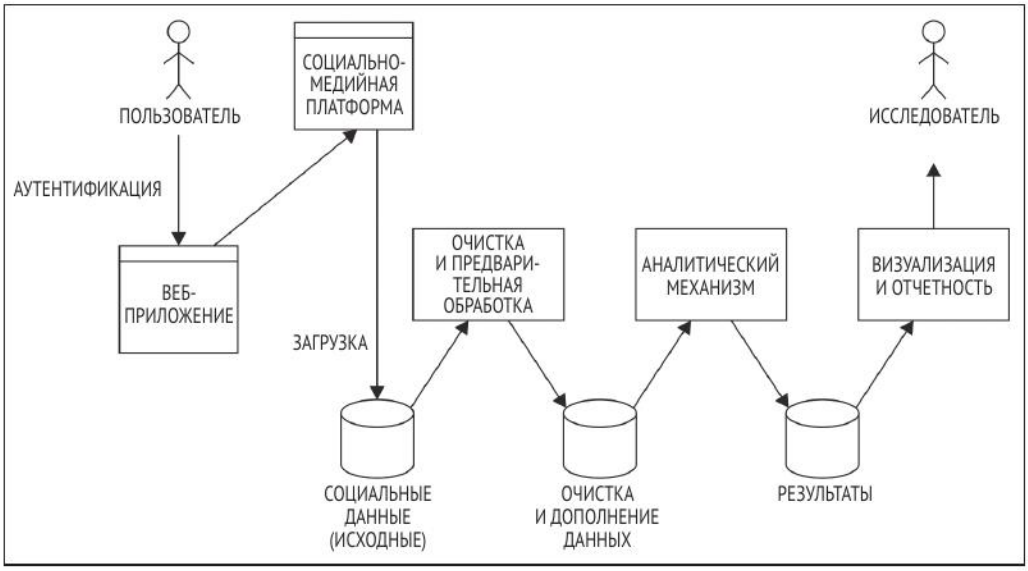

Рис. 1.2 ❖ Процесс анализа социальных данных

Этап аутентификации обычно выполняется с использованием промышленного стандарта под названием **открытый протокол авторизации (OAuth)**. Это трехсторонний процесс, в том смысле, что вовлекает трех агентов: пользователя, потребителя (наше приложение) и провайдера ресурса (социально-медийной платформы). Он состоит из следующих этапов:

1) пользователь соглашается дать потребителю доступ к социально-медийной платформе;
2) поскольку пользователь не передает свой пароль непосредственно потребителю, потребитель предварительно обращается к провайдеру ресурса, чтобы получить маркер и секретный ключ для приложения. Они используются для подписи каждого запроса и предотвращения взлома;
3) затем пользователь перенаправляется с маркером к провайдеру ресурса, который просит подтвердить разрешение на доступ потребителя к данным пользователя;

4) в зависимости от особенностей социально-медийной платформы провайдер может также попросить подтвердить возможность выполнения потребителем каких-либо действий от имени пользователя, например отправку обновлений, публикацию ссылок и т. д.;
5) провайдер ресурса передает потребителю действительный маркер;
6) далее маркер может вернуться к пользователю, который подтверждает доступ.

На рис. 1.3 изображен процесс авторизации со всеми описанными этапами. Важно помнить, что обмен учетными данными (имя пользователя/пароль) происходит только между пользователем и провайдером ресурса на этапах 3 и 4. Весь обмен на других этапах опирается на маркеры:

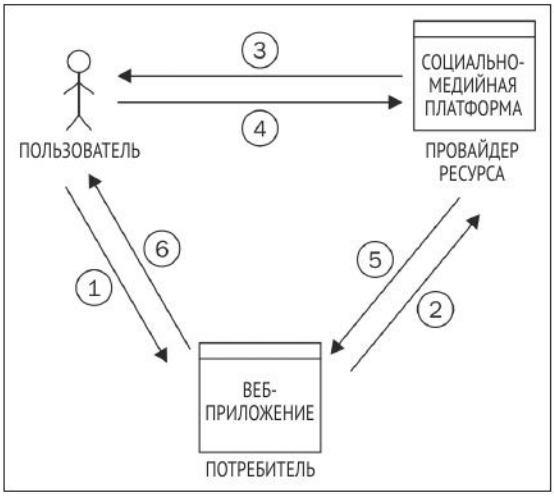

Рис. 1.3 ❖ Процесс авторизации согласно стандарту OAuth

С точки зрения пользователя этот сложный процесс происходит, когда пользователь вызывает наше веб-приложение и нажимает кнопку **«Вход с учетными данными Facebook»** (или Twitter, Google+ и т. д.). Затем пользователь должен подтвердить, что дает нашему приложению полномочия, и все остальное происходит уже без его участия.

С точки зрения разработчика самое замечательное, что экосистема Python уже имеет проверенные библиотеки для большинства социально-медийных платформ, которые включают реализацию процесса аутентификации. Как только вы зарегистрируете свое приложение в целевой службе, соответствующая платформа предоставит вашему приложению все необходимые маркеры авторизации. На рис. 1.4 изображен скриншот Twitter-приложения с названием Intro to Text Mining. На вкладке с настройками Keys and Access Tokens (Ключи и маркеры доступа) разработчик может найти ключ API и секретный ключ, а также маркер доступа и секретный маркер доступа. Более подробно детали авторизации для каждой социально-медийной платформы мы обсудим в соответствующих главах.

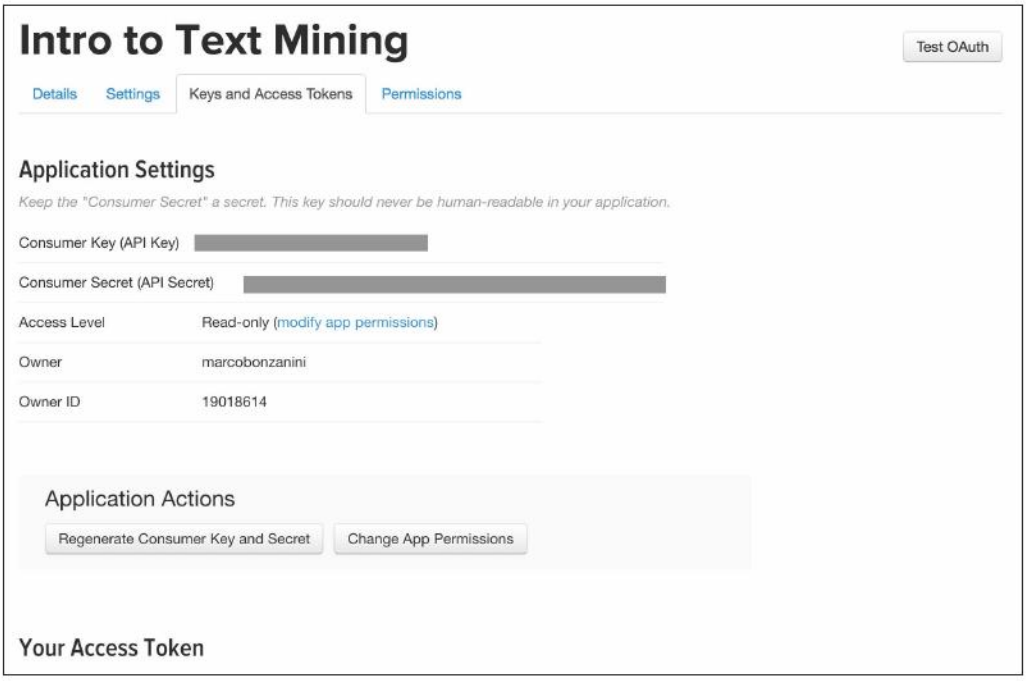

Рис. 1.4 ❖ Настройки для Twitter-приложения с названием Intro to Text Mining. Страница содержит все маркеры авторизации для использования разработчиком в своем приложении

Этапы сбора, очистки и предварительной обработки данных также зависят от особенностей социально-медийной платформы. В частности, этап сбора данных требует предварительной авторизации, поскольку для загрузки доступны только те данные, на получение которых дано разрешение. Очистка и предварительная обработка, с другой стороны, функционально зависят от вида моделирования и анализа данных, которые мы используем для получения желаемых выводов.

Как показано на рис. 1.2, моделирование и анализ выполняются компонентом «**Аналитический механизм**». Типичные задачи обработки данных, с которыми мы будем встречаться на протяжении всей книги, связаны с анализом текстов и графов.

Анализ текстов (или аналитическая обработка текстов) – это процесс получения структурированной информации из неструктурированных текстовых данных. Анализ текстов применим к большинству социально-медийных платформ, где пользователям разрешено публиковать информацию в форме сообщений или комментариев.

Приведем некоторые примеры применения анализа текстов:

- **классификация документов**: задача определения принадлежности документа к одной или нескольким категориям;
- **кластеризация документов**: задача группировки документов в когерентные и отличные друг от друга подмножества (например, по теме или подтеме), называемые кластерами;

- **реферирование документов**: задача создания сокращенной версии документа с целью уменьшения информационной нагрузки на пользователя, при сохранении самых важных аспектов, описанных в первоисточнике;
- **выделение сущностей**: задача поиска и классификации сущностей, упоминаемых в тексте, на категории, такие как люди, организации или географические местоположения;
- **анализ мнений**: задача идентификации и классификации чувств и мнений, выраженных в тексте, с целью понимания отношения к конкретному продукту, теме, службе и т. д.

Не все эти применения рассчитаны на социальные медиа, но растущий объем текстовых данных, доступных благодаря этим платформам, делает социальные медиа естественной площадкой для применения анализа текстов.

Анализ графов также сосредоточен на структуре данных. Графы просты для понимания и одновременно являются мощной структурой данных, которая достаточно универсальна, чтобы применяться к многообразным представлениям данных. В графах есть два главных компонента для рассмотрения: узлы, представляющие сущности, или объекты, и ребра, представляющие связи, или соединения между узлами. В контексте социальных медиа очевидное использование графа – представление социальных связей пользователей. В более общем плане, в общественных науках графовая структура, используемая для представления социальных связей, также именуется социальной сетью.

Использование такой структуры данных внутри социальной сети позволяет естественным образом представить пользователей в виде узлов, а их связи (такие как *друзья* или *читатели*) – как ребра. Тем самым такую информацию, как *друзья друзей, которым нравится Python*, легко получить путем простого обхода графа (то есть обход графа по ребрам от одного узла к другому). Теория и анализ графов предлагают гораздо больше возможностей для получения более глубоких выводов, которые не столь очевидны, как в предыдущем примере.

После высокоуровневого обсуждения анализа социальных данных в следующем разделе мы рассмотрим несколько инструментов Python, которые обычно используются в проектах анализа данных.

Инструменты Python для науки о данных

До сих пор для обозначения задач и технических методов, которые собираемся применять на протяжении всей книги, мы использовали термин «анализ данных». В заголовке этого раздела упоминается термин «*наука о данных*». Использование этого термина имело в последние годы взрывной эффект, в особенности в деловой среде. В то же время многие ученые и журналисты подвергали его использование критике, называя его новомодным словечком. Другие академические институты начали предлагать курсы по науке о данных, и появилось множество книг и статей по данному предмету. Вместо того чтобы выразить твердое мнение о том, где необходимо провести черту между разными дисциплинами, мы ограничимся тем, что посмотрим, как в наше время растет общий интерес к многочисленным областям, включая науку о данных, анализ данных, статистику, машинное обучение, искусственный интеллект, визуализацию данных и др. Обсуждаемые темы по своему характеру междисциплинарные, и все они время от времени что-то

заимствуют друг у друга. Мы живем в удивительное время, когда можно работать в любой из этих областей, испытывая большой интерес со стороны общественности и наблюдая постоянный ажиотаж с появлением новых достижений.

Цель этого раздела – познакомить вас с языком Python как инструментом науки о данных и описать часть экосистемы Python, которую мы собираемся использовать в последующих главах.

Python – один из самых интересных языков для проектов аналитической обработки данных. Ниже перечислены некоторые причины, объясняющие, почему он соответствует данному назначению:

- декларативный и интуитивный синтаксис;
- богатая экосистема для обработки данных;
- эффективность.

Python имеет плоскую кривую обучения благодаря превосходному синтаксису. Будучи динамическим и интерпретируемым языком, он способствует быстрой разработке и интерактивному исследованию. Экосистема для обработки данных частично описывается в последующих разделах, которые познакомят с главными библиотеками, которые мы будем использовать в этой книге.

С точки зрения эффективности, интерпретируемые и высокоуровневые языки не славятся особым быстродействием. Такие инструменты, как NumPy, достигают эффективности, внутренне подключаясь к низкоуровневым динамическим библиотекам и предоставляя дружественный интерфейс Python. Кроме того, большое число проектов использует **Cython** – надмножество Python, обогащающее язык и позволяющее, кроме всего прочего, определять типы переменных и компилировать программный код. Многие другие проекты в мире Python находятся в процессе решения вопросов эффективности с глобальной целью – ускорить выполнение реализаций на чистом Python. В настоящей книге мы не будем углубляться в Cython и другие перспективные проекты, а воспользуемся библиотекой NumPy (в том числе посредством других библиотек, которые используют NumPy) для аналитической обработки данных.

Настройка среды разработки Python

Когда я начал работу над этой книгой, только-только вышла версия Python 3.5, привлекшая внимание некоторыми новейшими возможностями, такими как улучшенная поддержка асинхронного программирования и семантическое определение подсказок о типах в исходном программном коде. С точки зрения использования, версия Python 3.5 пока еще не получила широкого распространения, но она представляет текущее направление развития языка.

Примеры, прилагаемые к книге, совместимы с версиями Python 3.4+ и 3.5+.

Пускаясь в дискуссии о выборе между Python 2 и Python 3, следует помнить один важный аргумент: через несколько лет поддержка Python 2 будет прекращена (во время написания книги разработчики Python предполагали прекратить поддержку Python 2 в 2020). В Python 2 не добавляется новых возможностей, потому что для этой ветви языка поддерживается только исправление ошибок. С другой стороны, многие библиотеки по-прежнему сначала разрабатываются для Python 2, и только потом в них добавляется поддержка Python 3. По этой причине иногда

могут возникать проблемы совместимости с той или иной библиотекой, которые, как правило, быстро разрешаются сообществом разработчиков. В целом, если нет веских причин, предпочтение должно отдаваться Python 3, в особенности для новых «целинных» проектов.

Инструменты pip и virtualenv

Чтобы содержать среду разработки в чистоте и облегчить переход от разработки к эксплуатации, предлагается для управления виртуальной средой и установки зависимостей использовать библиотеку `virtualenv`. Библиотека `virtualenv` позволяет создавать изолированные окружения Python и управлять ими. Такие окружения помогают избежать загрязнения глобального окружения Python библиотеками, которые могуть быть несовместимыми друг с другом. Этот инструмент позволяет поддерживать сразу несколько проектов, требующих разных конфигураций, и легко переключаться между ними. Более того, виртуальное окружение можно установить в локальной папке, доступной пользователям без административных привилегий.

Установить библиотеку `virtualenv` в глобальном окружении Python и сделать ее доступной всем пользователям можно с помощью диспетчера пакетов pip, выполнив команду в терминале (Linux/Unix) или в командной строке (Windows):

```
$ [sudo] pip install virtualenv
```

Команда `sudo` может потребоваться в Linux/Unix или macOS, если текущий пользователь не имеет административных привилегий.

Если библиотека уже установлена, ее можно обновить до последней версии:

```
$ pip install --upgrade [имя-библиотеки]
```

Начиная с Python 3.4 диспетчер `pip` поставляется вместе с Python. В предыдущих версиях его требуется устанавливать отдельно, как описывается на странице проекта (https://github.com/pypa/pip). Данный инструмент можно также использовать для обновления его самого:

```
$ pip install --upgrade pip
```

После установки `virtualenv` в глобальном окружении появляется возможность определить для каждого проекта свое окружение Python со своими зависимостями, не загрязняя глобального окружения. Благодаря этому установка зависимостей для каждого отдельного проекта становится чрезвычайно простым делом.

Чтобы установить виртуальную среду, выполните следующие шаги:

```
$ mkdir my_new_project # создать папку для нового проекта
$ cd my_new_project    # перейти в папку проекта
$ virtualenv my_env    # настроить индивидуальную виртуальную среду
```

В результате будет создана подпапка `my_env`, которая является также именем виртуальной среды, созданной в текущем каталоге. В этой подпапке хранятся все инструменты, необходимые для создания изолированной среды Python, включая двоичные файлы Python и стандартную библиотеку. Активировать среду можно следующей командой:

```
$ source my_env/bin/activate
```

После активации появится следующее приглашение к вводу:

```
(my_env) $
```

Теперь диспетчер библиотек pip для Python будет устанавливать их в эту конкретную среду:

```
(my_env)$ pip install [имя-библиотеки]
```

Все новые библиотеки Python, устанавливаемые в активированном окружении с помощью pip, помещаются в папку my_env/lib/python{ВЕРСИЯ}/site-packages. Отметим, что для выполнения этой команды не требуются административные привилегии, потому что папка является локальной.

Чтобы деактивировать виртуальное окружение, достаточно выполнить команду:

```
$ deactivate
```

Описанный процесс должен работать с официальными дистрибутивами Python, которые поставляются (или доступны для загрузки) с вашей операционной системой.

Conda, Anaconda и Miniconda

Существует еще один вариант управления зависимостями под названием **conda** (http://conda.pydata.org/), набирающий популярность в научном сообществе. Conda – это диспетчер библиотек с открытым исходным кодом и среды, позволяющий устанавливать несколько версий библиотек (и их зависимостей) и упрощающий переключение между ними. Он поддерживает Linux, macOS и Windows и, хотя изначально создавался для Python, может использоваться для упаковки и распространения любого программного обеспечения.

Диспетчер conda распространяется в виде двух дистрибутивов: полная версия Anaconda, включающая около 100 пакетов для научных вычислений, и облегченная версия Miniconda, включающая Python и установщик conda без внешних библиотек.

Если вы только начинаете изучать Python, имеете немного свободного времени, чтобы загрузить большой дистрибутив, и достаточный объем свободного дискового пространства и не хотите устанавливать все библиотеки вручную, начните с дистрибутива Anaconda. Для Windows и macOS дистрибутив Anaconda доступен с графическим установщиком или с установщиком командной строки. Скриншот рис. 1.5 показывает процесс установки в macOS. Для Linux имеется только установщик командной строки. Во всех случаях есть выбор между Python 2 и Python 3. Если вы предпочтете иметь полный контроль над своей системой, тогда Miniconda, скорее всего, будет для вас лучшим выбором.

После установки выбранной версии conda можно создать новое окружение conda, выполнив команду:

```
$ conda create --name my_env python=3.4   # или предпочитаемая версия
```

Среда активируется командой:

```
$ conda activate my_env
```

Подобно тому, как это происходит в virtualenv, в строке приглашения появится имя окружения:

```
(my_env) $
```

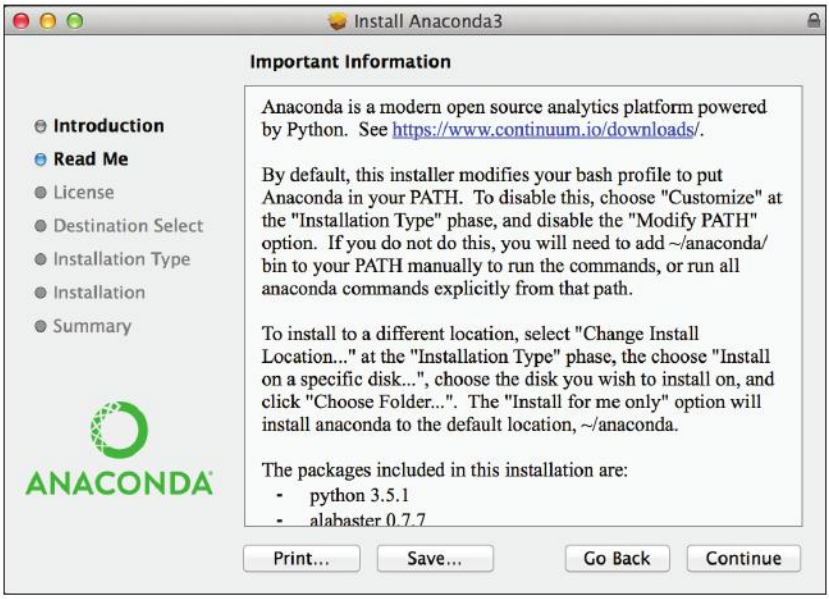

Рис. 1.5 ❖ Процесс установки дистрибутива Anaconda

Новые библиотеки в это окружение можно устанавливать командой:

```
$ conda install [имя-библиотеки]
```

Наконец, деактивировать окружение можно следующей командой:

```
$ conda deactivate
```

Еще одной замечательной чертой conda является возможность устанавливать библиотеки с помощью диспетчера `pip`, то есть если конкретную библиотеку или какую-то ее версию не получается установить командой `conda install`, всегда можно вернуться к традиционному диспетчеру пакетов для Python.

Если не указано обратное, по умолчанию conda ищет библиотеки на веб-сайте https://anaconda.org, тогда как `pip` использует **каталог пакетов Python** (Python Package Index, **PyPI**) по адресу: https://pypi.python.org/pypi. Неофициально последний иногда называют «сырной лавкой» **CheeseShop**. Оба диспетчера можно настроить на установку библиотек из локальной файловой системы или частного хранилища.

В следующем разделе для установки необходимых библиотек будет использоваться диспетчер `pip`, но вы можете переключиться на conda, если предпочитаете использовать данную альтернативу.

Эффективный анализ данных

Этот раздел знакомит с двумя основополагающими библиотеками для научных вычислений на Python: **NumPy** и **pandas**.

Библиотека NumPy (Numerical Python) предлагает инструменты быстрой и эффективной обработки массивоподобных структур данных. Хранение и манипуляция числовыми данными с использованием встроенных в Python структур данных

(к примеру, списков или словарей) оказывается намного медленнее, чем использование массива из NumPy. Кроме того, массивы NumPy часто используются другими библиотеками в качестве контейнеров данных для различных алгоритмов, выполняющих операции с векторами и матрицами.

Чтобы установить NumPy с помощью pip/virtualenv, используйте следующую команду:

```
$ pip install numpy
```

Библиотеки NumPy и pandas уже входят в состав полной версии Anaconda, поэтому тем, кто пользуется ею, предыдущий шаг установки не потребуется.

Основной структурой данных в NumPy является многомерный массив ndarray.

Следующий пример сеанса в интерактивной оболочке Python демонстрирует создание простого массива при помощи NumPy:

```
>>> import numpy as np
>>> data = [1, 2, 3]  # список целых чисел
>>> my_arr = np.array(data)
>>> my_arr
array([1, 2, 3])
>>> my_arr.shape
(3,)
>>> my_arr.dtype
dtype('int64')
>>> my_arr.ndim
1
```

Этот пример наглядно показывает, что данные представлены одномерным массивом (атрибут ndim) с тремя элементами, как и ожидалось. Массив имеет тип данных int64, поскольку на входе все элементы целочисленные.

Быстродействие массива NumPy можно оценить, выполнив профилирование простой операции, например вычисления суммы, с помощью модуля timeit:

```
# Chap01/demo_numpy.py
from timeit import timeit
import numpy as np

if __name__ == '__main__':
    setup_sum = 'data = list(range(10000))'
    setup_np = 'import numpy as np;'
    setup_np += 'data_np = np.array(list(range(10000)))'
    run_sum = 'result = sum(data)'
    run_np = 'result = np.sum(data_np)'

    time_sum = timeit(run_sum, setup=setup_sum, number=10000)
    time_np = timeit(run_np, setup=setup_np, number=10000)

    print("Время выполнения встроенной функции sum(): {}".format(time_sum))
    print("Время выполнения функции np.sum(): {}".format(time_np))
```

Модуль timeit принимает в первом параметре фрагмент кода и многократно выполняет его, после чего выводит время, потребовавшееся для этого. Чтобы сосредоточиться на определенной части анализируемого программного кода, настройка исходных данных и необходимые операторы импорта перемещены

в параметр setup, который будет выполнен всего один раз и не будет включен в профилирование. Последний параметр – number – ограничивает число итераций 10 000 вместо значения по умолчанию, равного 1 000 000. Мы на своем компьютере получили такой результат:

```
Время выполнения встроенной функции sum(): 0.9970562970265746
Время выполнения функции np.sum(): 0.07551316602621228
```

Встроенная функция sum() более чем в 10 раз медленнее функции sum() из NumPy. Более сложные фрагменты кода могут показывать еще большую разницу.

Соглашения об именах. Сообщество разработчиков на Python пришло к соглашениям, касающимся импортирования некоторых популярных библиотек. NumPy и pandas – два самых известных примера. Они обычно импортируются под псевдонимом, например:

```
import numpy as np
```

При таком подходе, как можно заметить в предыдущих примерах, к функциям из NumPy можно обращаться как np.*имя_функции()*. Аналогично библиотека pandas импортируется под псевдонимом pd. В общем случае импортирование всего пространства имен библиотеки оператором from numpy import * считается плохой практикой, потому что в результате загрязняется текущее пространство имен.

Массивы NumPy имеют следующие особенности, о которых следует помнить:

- размер массива NumPy фиксируется при создании, в отличие, к примеру, от списков Python, размеры которых можно изменять динамически, поэтому операции, изменяющие размер массива, в действительности создают новый массив и удаляют исходный;
- все элементы массивов должны быть одного типа (кроме массивов объектов, которые, как следствие, могут иметь разные размеры);
- библиотека NumPy помогает использовать операции с векторами и писать более компактный и читаемый код.

Библиотека pandas – вторая из представленных в этом разделе. Она реализована поверх NumPy, поэтому тоже обеспечивает высокую скорость вычислений и, главное, предлагает удобные структуры данных **Series** и **DataFrame** (то есть числовые ряды и таблицы данных), которые позволяют гибко и сжато манипулировать данными.

Перечислим лишь некоторые из замечательных особенностей библиотеки pandas:

- быстрые и эффективные объекты для манипулирования данными;
- инструменты для чтения и записи данных в разных форматах, таких как CSV, текст, электронные таблицы MS Excel или структуры данных SQL;
- интеллектуальная обработка пропущенных данных и связанное с ней согласование данных;
- создание срезов больших наборов данных на основе меток;
- SQL-подобное агрегирование и преобразование данных;
- поддержка временных рядов;
- интегрированная визуализация данных в виде графиков;
- библиотеку pandas можно установить из «сырной лавки» CheeseShop, как обычно:

```
$ pip install pandas
```

Давайте рассмотрим следующий пример сеанса в интерактивной оболочке Python, демонстрирующий обработку небольшого искусственного набора данных:

```
>>> import pandas as pd
>>> data = {'user_id': [1, 2, 3, 4], 'возраст': [25, 35, 31, 19]}
>>> frame = pd.DataFrame(data, columns=['user_id', 'возраст'])
>>> frame.head()
   user_id  возраст
0        1       25
1        2       35
2        3       31
3        4       19
```

Исходные данные хранятся в словаре, ключами которого являются атрибуты пользователей (числовой идентификатор и возраст в годах), а значениями – списки, в которых каждому пользователю соответствует определенная позиция. После создания объекта `DataFrame` соответствие между этими данными сразу становится понятным. Функция `head()` выводит данные в табличной форме, усекая набор до первых десяти строк, если их больше.

Теперь можно расширить таблицу `DataFrame`, добавив еще один столбец:

```
>>> frame['больше_тридцати'] = frame['возраст'] > 30
>>> frame.head()
   user_id  возраст  больше_тридцати
0        1       25            False
1        2       35             True
2        3       31             True
3        4       19            False
```

Декларативный синтаксис библиотеки pandas избавляет от необходимости выполнять итеративный обход всего столбца, чтобы получить доступ к его данным, и позволяет применить SQL-подобную операцию, как показано в предыдущем примере. Эта операция использует существующие данные, чтобы создать столбец булевых значений. Таблицу данных `DataFrame` можно также расширить, добавив новые данные:

```
>>> frame['любит_python'] = pd.Series([True, False, True, True],
index=frame.index)
>>> frame.head()
   user_id  возраст  больше_тридцати  любит_python
0        1       25            False          True
1        2       35             True         False
2        3       31             True          True
3        4       19            False          True
```

Метод `describe()` позволяет увидеть несколько простых описательных статистик:

```
>>> frame.describe()
        user_id  возраст  больше_тридцати  любит_python
count  4.000000      4.0                4             4
mean   2.500000     27.5              0.5          0.75
std    1.290994      7.0          0.57735           0.5
min    1.000000     19.0            False         False
```

```
25%    1.750000    23.5          0        0.75
50%    2.500000    28.0        0.5           1
75%    3.250000    32.0          1           1
max    4.000000    35.0       True        True
```

Как видите, 50% пользователей старше 30, и 75% из них нравится Python.

Скачивание примеров исходного кода. Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф в разделе «Читателям – Файлы к книгам».

Машинное обучение

Машинное обучение – это дисциплина, которая исследует и разрабатывает алгоритмы, обучающиеся на данных и делающие прогнозы. Она имеет самое прямое отношение к анализу данных, и иногда названия этих двух областей используются взаимозаменяемо. Характерное различие между ними: в центре внимания машинного обучения находятся прогнозы на основе известных свойств данных, в то время как главной целью анализа является получение новых знаний на основе неизвестных свойств данных. Обе области заимствуют алгоритмы и технические методы друг у друга. Одна из целей настоящей книги – практичность, поэтому мы подтверждаем, что в академическом плане эти две области, несмотря на массу точек соприкосновения, часто имеют разные цели и допущения, но мы не будем слишком беспокоиться по этому поводу.

Вот некоторые примеры приложений машинного обучения:

- классификация электронных писем на спам и не спам;
- выбор темы для статьи из списка известных тем, таких как спорт, финансы или политика;
- анализ банковских сделок с целью выявления попыток мошенничества;
- определение по поисковому запросу «*apple*», чем интересуется пользователь – фруктами или компьютерами.

Наиболее популярные методологии можно разделить на обучение с учителем и обучение без учителя, которые будут описаны в следующем разделе. Такое сверхупрощение не описывает всю широту и глубину области машинного обучения, но является хорошей отправной точкой, чтобы оценить некоторые ее технические стороны.

Обучение с учителем можно использовать для решения таких задач, как классификация, когда данные сопровождаются дополнительными атрибутами, которые требуется предсказать, например меткой класса (категории). В этом случае классификатор может связывать каждый входной объект с нужным выходным результатом. На основе выводов из признаков входных объектов классификатор может предсказывать подходящую метку для новых, ранее не встречавшихся входных данных. Широкое применение нашли такие технические методы, как **наивный байесовский классификатор** (Naive Bayes, NB), **метод опорных векторов** (Support Vector Machines, SVM) и модели из семейства **нейронных сетей** (Neural Networks, NN), такие как перцептроны и многослойные перцептроны.

Входные прецеденты, которые используются обучающимся алгоритмом для создания математической модели, называются **тренировочными данными**, в то время как не встречавшиеся ранее входные данные, на которых требуется получить прогноз, называются **тестовыми данными**. Прецеденты, поступающие на

вход алгоритма машинного обучения, обычно имеют форму вектора, каждый элемент которого представляет *признак* входного прецедента. При обучении с учителем требуемый выходной результат, назначаемый каждому ранее не встречавшемуся входному прецеденту, обычно называется **меткой** или **целью**.

Обучение без учителя, напротив, применяется, когда данные не сопровождаются соответствующим выходным значением. Типичным примером такого рода задач является кластеризация. В этом случае алгоритм пытается найти скрытые в данных структуры с целью сгруппировать схожие объекты в кластеры. Еще одно применение – идентификация объектов, которые внешне не принадлежат конкретной группе (к примеру, обнаружение выбросов). Примером широко используемого алгоритма кластеризации является алгоритм K-средних.

Для решения задач машиного обучения на языке Python в основном используется библиотека **scikit-learn**. Это коллекция алгоритмов машинного обучения с открытым исходным кодом, включающая инструменты для доступа к данным, их предварительной обработки, оценки результативности алгоритма и визуализации результатов.

Библиотеку scikit-learn можно установить обычным способом:

```
$ pip install scikit-learn
```

Теперь разберем по шагам применение scikit-learn для решения задачи кластеризации, не вдаваясь в детали самой технологии.

Поскольку мы пока не располагаем социальными данными, можно воспользоваться одним из наборов, который поставляется вместе с библиотекой scikit-learn.

Воспользуемся набором данных Iris, также именуемым Ирисами Фишера. Он был введен в употребление в 1930-х гг. Рональдом Фишером и сегодня является одним из классических наборов данных: благодаря небольшому размеру он часто используется в литературе для демонстрации примеров. Набор содержит 50 прецедентов каждой из трех разновидностей цветка ириса, и каждый прецедент включает четыре признака: длину и ширину лепестков и чашелистиков.

Этот набор данных обычно используется для демонстрации решения задачи классификации, поскольку данные сопровождаются правильными метками для каждого прецедента, тогда как его применение для задачи кластеризации менее распространено, главным образом потому, что имеются всего два хорошо различимых кластера с довольно очевидным разделением. Благодаря небольшому размеру и простой структуре его удобно использовать для введения в анализ данных с применением библиотеки scikit-learn. Если вы захотите выполнить пример, включая часть, связанную с визуализацией, установите также библиотеку **matplotlib**, например командой `pip install matplotlib`. Подробнее о визуализации данных в Python мы поговорим далее в этой главе.

Взгляните на код следующей короткой программы:

```
# Chap01/demo_sklearn.py
from sklearn import datasets
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

if __name__ == '__main__':
    # Загрузить данные
    iris = datasets.load_iris()
```

```
X = iris.data
petal_length = X[:, 2]
petal_width = X[:, 3]
true_labels = iris.target

# Выполнить кластеризацию по методу K-средних
estimator = KMeans(n_clusters=3)
estimator.fit(X)
predicted_labels = estimator.labels_

# Задать цветовую схему: красный, желтый и синий
color_scheme = ['r', 'y', 'b']

# Задать маркеры: круг, "x" и "плюс"
marker_list = ['o', 'x', '+']

# Назначить цвета/маркеры предсказанным меткам
colors_predicted_labels = [color_scheme[lab]
                           for lab in predicted_labels]
markers_predicted = [marker_list[lab]
                     for lab in predicted_labels]

# Назначить цвета/маркеры истинным меткам
colors_true_labels = [color_scheme[lab]
                      for lab in true_labels]
markers_true = [marker_list[lab]
                for lab in true_labels]

# Построить и сохранить две диаграммы рассеяния
for x, y, c, m in zip(petal_width,
                      petal_length,
                      colors_predicted_labels,
                      markers_predicted):
    plt.scatter(x, y, c=c, marker=m)
plt.savefig('iris_clusters.png')

for x, y, c, m in zip(petal_width,
                      petal_length,
                      colors_true_labels,
                      markers_true):
    plt.scatter(x, y, c=c, marker=m)
plt.savefig('iris_true_labels.png')

print(iris.target_names)
```

Прежде всего она загружает набор данных в переменную `iris`, то есть в объект, содержащий сами данные и информацию о данных. В частности, атрибут `iris.data` хранит сами данные в форме массива NumPy, а `iris.target` – числовую метку, представляющую класс прецедента. Вектор каждого прецедента хранит четыре значения: длину чашелистика, ширину чашелистика, длину лепестка и ширину лепестка в сантиметрах. С помощью синтаксиса срезов программа извлекает третий и четвертый элементы из каждого прецедента в переменные `petal_length` и `petal_width`. Они будут использоваться для отображения прецедентов в двумерном представлении, несмотря на то что векторы имеют четыре размерности.

Сам процесс кластеризации выполняется двумя строками программы: одна создает экземпляр алгоритма KMeans, а вторая применяет данные к модели вы-

зовом метода `fit()`. Простота интерфейса является одной из особенностей библиотеки scikit-learn, которая часто позволяет применять обучающиеся алгоритмы всего несколькими строками кода. Для применения алгоритма K-средних мы выбираем число кластеров, равное 3, заданное по условию. Имейте в виду, что число кластеров редко бывает известно заранее. Определение правильного (или самого интересного) числа кластеров – сложная задача, отличная от применения *самого* алгоритма кластеризации. Поскольку цель этого примера – в том, чтобы кратко познакомить с библиотекой scikit-learn и простотой ее интерфейса, мы принимаем этот укороченный способ. Обычно подготовка данных в формате, понятном библиотеке scikit-learn, требует больше усилий.

Вторая половина примера служит для визуализации данных с использованием библиотеки matplotlib. Сначала определяется список `color_scheme` с цветовой схемой «красный, желтый, синий», чтобы обеспечить визуальное отличие кластеров. Затем, учитывая, что реальные метки и кластерные объединения для каждого прецедента заданы в виде целых чисел, начиная с 0, их можно использовать в качестве индексов для определения соответствий цветов.

Обратите внимание, что числа для реальных меток связаны с определенным значением меток, то есть именем класса, а номера кластеров используются лишь с целью показать принадлежность прецедента некоторому кластеру – они не несут никакой информации о значении кластера. В частности, имеются три класса для реальных меток – setosa (щетинистый), versicolor (разноцветный) и virginica (виргинский), – представляющих три разновидности цветка ириса, имеющиеся в наборе данных.

Последние строки примера создают из данных две диаграммы рассеяния, одну для реальных меток и другую для кластерных объединений, используя в качестве размерностей длину и ширину лепестка. Оба графика представлены на рис. 1.6. Положение элементов на графиках, конечно же, одинаковое, но важно отметить, как алгоритм выделил три группы. В частности, кластер слева внизу четко отделяется от двух других, и алгоритму легко удалось его идентифицировать. Другие два кластера, напротив, отличить труднее из-за наложения нескольких элементов. Поэтому алгоритм допустил несколько ошибок.

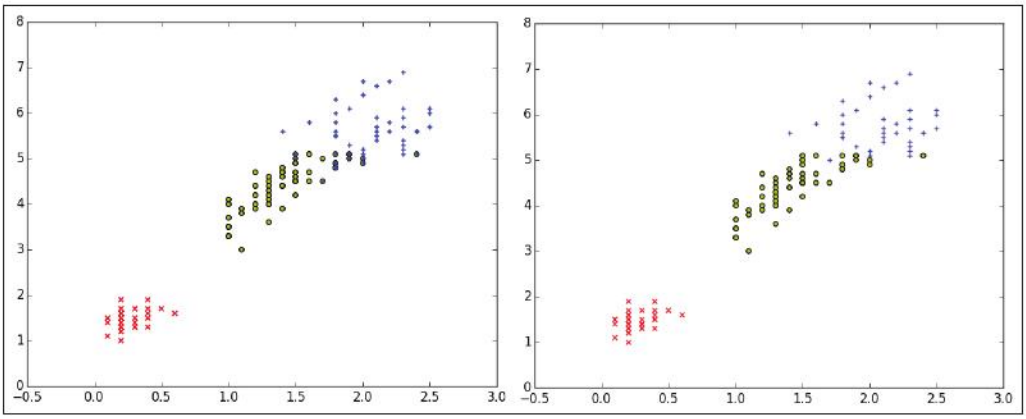

Рис. 1.6 ❖ Двухмерное представление данных из набора Iris, окрашенных согласно реальным меткам (слева) и по результатам кластеризации (справа)

Отметим, что мы смогли заметить ошибки в данном примере благодаря известности реального класса каждого прецедента. Алгоритм просто создал объединения, основываясь на переданных ему признаках.

Обработка естественного языка

Обработка естественного языка (Natural Language Processing, NLP) – это дисциплина, связанная с исследованием методов и приемов автоматического анализа и порождения текстов на естественном языке, то есть на языке, используемом людьми для общения.

Эта область активно исследуется вот уже много десятилетий, так как первые годы часто приписываются Алану Тьюрингу, одному из родоначальников информатики, который в 1950 предложил тест для оценки машинного интеллекта. Его идея довольно проста: человек, судья, взаимодействует с одним компьютером и одним человеком. На основании ответов на вопросы он должен определить, с кем разговаривает: с человеком или компьютерной программой. Задача компьютерной программы — ввести судью в заблуждение, заставив сделать неверный выбор. Если судья не может сказать определенно, кто из собеседников является человеком, считается, что машина прошла тест и показывает признаки *интеллекта*.

Этот тест в наше время известен как «тест Тьюринга». В течение долгого времени о нем знали только в кругах, связанных с информатикой, однако недавно он был представлен популярными медиа более широкой аудитории. Фильм «*Игра в имитацию*» (*The Imitation Game, 2014*), например, основан на биографии Алана Тьюринга, а название фильма ясно указывает на сам тест. Другой фильм, в котором упоминается тест Тьюринга, «*Из машины*» (*Ex Machina, 2015*), делает еще более сильный акцент на разработку искусственного интеллекта, который может лгать и одурачивать людей для своей выгоды. В этом фильме тест Тьюринга выполняется между человеческим судьей и человекоподобным роботом Ава, с ипользованием прямого вербального взаимодействия. Не раскрывая концовку фильма для тех, кто его не видел, отметим лишь, что история развивается вокруг искусственного интеллекта, который в сомнительной и хитрой манере показывает, что он намного умнее человека. Интересно отметить, что футуристический робот был натренирован, используя журналы регистрации событий поисковой системы с целью понять, как люди задают вопросы, и впоследствии им подражать.

Это небольшое путешествие из прошлого в гипотетическое будущее искусственного интеллекта показывает, что овладение человеческим языком является важнейшим направлением в разработке **искусственного интеллекта (ИИ)**. Несмотря на все недавние достижения, эта цель пока еще не совсем достигнута, и обработка естественного языка на данный момент по-прежнему остается горячей темой.

В контексте социальных медиа очевидно, что нас ждет огромный объем информации на естественном языке, ожидающей обработки. Объем текстовых данных, доступных через социальные медиа, непрерывно увеличивается, и для многих вопросов ответы, вероятно, уже написаны; но преобразование простого текста в формализованную информацию – задача не из легких. Беседы и диалоги в социальных сетях происходят постоянно, пользователи задают технические вопросы в форумах и находят ответы, а клиенты в комментариях или отзывах описы-

вают свой опыт использования конкретных продуктов. Знание тематики таких бесед, поиск наиболее опытных пользователей, которые отвечают на вопросы, и понимание мнений клиентов, которые пишут отзывы, являются теми целями, которых можно достичь с достаточно удовлетворительным уровнем точности посредством обработки естественного языка.

В отношении Python наиболее популярной библиотекой для обработки естественного языка является библиотека **NLTK** (Natural Language Toolkit – инструментарий для обработки естественного языка). Данный набор инструментов предоставляет дружественный интерфейс для распространенных задач обработки естественного языка, а также лексические ресурсы и лингвистические данные.

Вот некоторые задачи, которые легко решаются при помощи NLTK:

- выделение слов и предложений, то есть разбиение текстового потока на отдельные лексемы;
- классификация слов по частям речи, то есть определение категорий слов согласно их синтаксическим функциям, таким как существительные, прилагательные, глаголы и т. д.;
- идентификация именованных сущностей, например идентификация и классификация упоминаний людей, географических объектов, организаций и т. д.;
- применение методов машинного обучения (например, классификации) к тексту;
- извлечение любой информации из исходного текста.

Установка NLTK выполняется как обычно:

```
$ pip install nltk
```

Разница между NLTK и многими другими библиотеками в том, что она сопровождается также лингвистическими ресурсами (то есть данными) для конкретных задач. Из-за большого объема эти данные не включены в стандартную установку и должны загружаться отдельно.

Процесс установки полностью описан на странице: http://www.nltk.org/data.html, и я настоятельно рекомендую обращаться за подробностями к этому официальному руководству.

Проще говоря, в интерактивной оболочке Python можно выполнить следующие команды:

```
>>> import nltk
>>> nltk.download()
```

В графическом окружении откроется новое окно с графическим интерфейсом. Если графическое окружение отсутствует, в терминале появится текстовый интерфейс. В нем вы сможете выбрать и загрузить отдельные или даже все пакеты с данными сразу (которые займут порядка 2.2 Гбайта дискового пространства).

Загрузчик сохранит файл в централизованном месте (`C:\nltk_data` в Windows и `/usr/share/nltk_data` в Unix и Mac), если выполнить его с привилегиями администратора, иначе данные будут сохранены в домашней папке (например, `~/nltk_data`). Также можно выбрать специальную папку, но в этом случае библиотеке NLTK потребуется переменная окружения `$NLTK_DATA`, чтобы узнать, где найти свои данные, поэтому ее необходимо определить соответствующим образом.

Если дисковое пространство не является проблемой, установка всех данных, вероятно, является самым удобным вариантом, поскольку это делается всего один раз, и потом об этом можно забыть. С другой стороны, загрузка всех данных не позволяет четко определить, какие ресурсы необходимы. Если вы предпочитаете иметь полный контроль над всем, что устанавливается, необходимые пакеты можно скачивать поочередно. В этом случае время от времени вы будете обнаруживать небольшие неровности в форме сообщения об ошибке `LookupError`, означающей, что ресурс, который вы пытаетесь использовать, отсутствует и его необходимо скачать.

Например, если сразу после установки библиотеки NLTK попытаться разбить текст на лексемы из интерактивной оболочки Python, как показано ниже:

```
>>> from nltk import word_tokenize
>>> word_tokenize('Some sample text')
Traceback (most recent call last):
  # здесь вы увидите длинную трассировку стека
LookupError:
**********************************************************************
  Resource 'tokenizers/punkt/PY3/english.pickle' not found.
  Please use the NLTK Downloader to obtain the resource: >>>
  nltk.download()
  Searched in:
    - '/Users/marcob/nltk_data'
    - '/usr/share/nltk_data'
    - '/usr/local/share/nltk_data'
    - '/usr/lib/nltk_data'
    - '/usr/local/lib/nltk_data'
    - ''
**********************************************************************
```

Эта ошибка сообщает, что ресурс `punkt`, отвечающий за выделение лексем, не найден ни в одной из обычных папок и поэтому необходимо запустить загрузчик NLTK и решить этот вопрос, получив нужный пакет.

Предположим теперь, что у нас установлена полностью рабочая библиотека NLTK, вернемся к предыдущему примеру и обсудим выделение лексем подробнее.

В контексте обработки естественного языка выделение лексем (или сегментирование) – это процесс разбиения фрагмента текста на меньшие элементы, называемые лексемами или сегментами. Лексемы можно интерпретировать поразному, но нас, как правило, интересуют слова и предложения. Вот простой пример применения функции `word_tokenize()`[1]:

```
>>> from nltk import word_tokenize
>>> text = "Съешь еще этих мягких французских булок, да выпей чаю"
>>> words = word_tokenize(text)
>>> print(words)
# ['Съешь', 'еще', 'этих', 'мягких', 'французских', 'булок', ',', 'да', 'выпей', 'чаю']
```

[1] Следующее далее предложение является фразой-пантограммой, содержащей все буквы алфавита. В оригинале «Quick brown fox jumped over a lazy dog». Это предложение применяется для визуализации шрифтов и проверок компьютерных клавиатур. На русский язык английская пантограмма переводится как «Шустрая бурая лиса прыгает через ленивую собаку». – *Прим. перев.*

Функция `word_tokenize()` возвращает список строк, в котором каждая строка представляет слово или знак препинания. Границы слов в этом примере заданы пробелами. Аналогично функция `sent_tokenize()` возвращает список строк, в котором каждая строка представляет предложение, ограниченное знаками препинания. Вот пример использования обеих функций:

```
>>> from nltk import word_tokenize, sent_tokenize
>>> text = "Попробуй еще! Что? Этих мягких французских булок."
>>> sentences = sent_tokenize(text)
>>> print(sentences)
# ['Попробуй еще!', 'Что?', 'Этих мягких французских булок.']
>>> for sentence in sentences:
...     words = word_tokenize(sentence)
...     print(words)
# ['Попробуй', 'еще', '!']
# ['Что', '?']
# ['Этих', 'мягких', 'французских', 'булок', '.']
```

Как видите, знаки пунктуации интерпретируются как самостоятельные лексемы и как таковые включены в результат функции `word_tokenize()`. Возникает вопрос, который до сих пор еще не был задан: как определять лексемы? Функция `word_tokenize()` реализует алгоритм, разработанный для стандартного языка, такого как английский или русский. Поскольку в центре внимания настоящей книги находятся данные из социальных сетей, резонно выяснить, применимы ли правила стандартного естественного языка в нашем контексте. Рассмотрим вымышленный пример для данных Twitter:

```
>>> tweet = '@marcobonzanini: an example! :D http://example.com #NLP'
>>> print(word_tokenize(tweet))
# ['@', 'marcobonzanini', ':', 'an', 'example', '!', ':', 'D', 'http', ':',
'//example.com', '#', 'NLP']
```

Образец твита имеет некоторые особенности, ломающие работу стандартного алгоритма выделения лексем:

- имена пользователей в Twitter предваряются символом `@`, но стандартный алгоритм разделил `@marcobonzanini` на две лексемы, распознав символ `@` как знак пунктуации;
- эмограммы, такие как `:D`, часто встречаются в чатах, текстовых сообщениях и, разумеется, в социальных медиа, но они не являются частью стандартного английского языка, поэтому снова ошибочно были выделены две лексемы вместо одной;
- пользователи часто обмениваются URL-адресами, чтобы поделиться статьями или изображениями, но они также не являются частью стандартного английского языка, поэтому разбиваются на компоненты;
- хештеги, такие как `#NLP`, – это строки, начинающиеся с символа `#`. Они используются для определения темы сообщения, чтобы другие пользователи могли легко отыскать тему или следить за смыслом беседы.

Пример выше показывает, что даже такая простая задача, как выделение лексем, может скрывать много коварных граничных случаев, которые требуют чего-то более изощренного, чем исходные интуитивные соображения. К счастью, библиотека NLTK предлагает следующее стандартное решение:

```
>>> from nltk.tokenize import TweetTokenizer
>>> tokenizer = TweetTokenizer()
>>> tweet = '@marcobonzanini: an example! :D http://example.com #NLP'
>>> print(tokenizer.tokenize(tweet))
# ['@marcobonzanini', ':', 'an', 'example', '!', ':D',
'http://example.com', '#NLP']
```

Предыдущие примеры позволяют почувствовать простоту интерфейса библиотеки NLTK. Мы будем использовать ее по разным поводам на протяжении всей книги.

С ростом интереса к обработке естественного языка эта часть экосистемы Python существенно выросла в последние годы, при этом продолжает появляться большое количество интересных проектов. Примером может служить библиотека **Gensim** с открытым исходным кодом, реализующая *тематическое моделирование для людей*, которая ориентирована на семантический анализ. Библиотека Gensim следует за тенденцией библиотеки NLTK и предлагает простой в использовании интерфейс, отсюда и подзаголовок *для людей*. Еще один аспект, который поднял ее популярность, – эффективность: она максимально оптимизирована для производительности, поддерживает возможность распределенных вычислений и может обрабатывать большие наборы данных без необходимости держать все данные в оперативной памяти.

Установка библиотеки Gensim следует обычной процедуре:

```
$ pip install gensim
```

Главными ее зависимостями являются библиотеки NumPy и SciPy, однако, если вы захотите воспользоваться быстродействием и поддержкой распределенных вычислений в Gensim, вам также придется установить библиотеку **Pyro4** (PYthon Remote Objects – удаленные объекты Python):

```
$ pip install Pyro4
```

Для демонстрации простоты интерфейса библиотеки Gensim рассмотрим модуль реферирования текстов:

```
# Chap01/demo_gensim.py
from gensim.summarization import summarize
import sys

fname = sys.argv[1]

with open(fname, 'r') as f:
    content = f.read()
    summary = summarize(content, split=True)
    for i, sentence in enumerate(summary):
        print("%d) %s" % (i+1, sentence))
```

Сценарий `demo_gensim.py` принимает один аргумент командной строки – имя текстового файла для реферирования. Чтобы протестировать этот сценарий, мы взяли часть текста со страницы Wikipedia о трилогии «Властелин колец», в частности абзац с описанием сюжета первого тома «Братство кольца». Сценарий можно вызвать следующей командой:

```
$ python demo_gensim.py lord_of_the_rings.txt
```

Он произведет следующий результат:

```
1) They nearly encounter the Nazgûl while still in the Shire, but shake off pursuit by cutting
through the Old Forest, where they are aided by the enigmatic Tom Bombadil, who alone is
unaffected by the Ring's corrupting influence.
2) Aragorn leads the hobbits toward the Elven refuge of Rivendell, while Frodo gradually
succumbs to the wound.
3) The Council of Elrond reveals much significant history about Sauron and the Ring, as well
as the news that Sauron has corrupted Gandalf's fellow wizard, Saruman.
4) Frodo volunteers to take on this daunting task, and a "Fellowship of the Ring" is formed to
aid him: Sam, Merry, Pippin, Aragorn, Gandalf, Gimli the Dwarf, Legolas the Elf, and the Man
Boromir, son of the Ruling Steward Denethor of the realm of Gondor.
1) Находясь еще в Графстве, они почти сталкиваются с Назгулом, но уходят от преследования,
прорубаясь через Старый Лес, где им помогает загадочный Том Бомбэдил, единственный не тронутый
тлетворным влиянием Кольца.
2) Арагорн приводит хоббитов к эльфийскому убежищу Ривенделл, а в это время Фродо серьезно
ослабевает от раны.
3) Совет Элронда раскрывает много значительных подробностей о Сауроне и Кольце, а также
сведения, что Саурон обольстил друга Гэндальфа мага Сарумана.
4) Фродо добровольно решается взять на себя эту грандиозную задачу, и "Братство кольца"
сформировано, дабы помочь ему: Сэм, Мэрри, Пипин, Арагорн, Гэндальф, гном Гимли, эльф Леголас
и Боромир, сын Дэнетора, Рулинга Стюарта королевства Гондор.
```

Функция `summarize()` из библиотеки Gensim реализует классический алгоритм TextRank. Алгоритм ранжирует предложения согласно их важности и выбирает самые характерные, чтобы сгенерировать итоговое краткое изложение. Стоит отметить, что этот подход называется методом *квазиреферирования* (extractive summarization), в том смысле, что результат содержит предложения, выбранные из исходного текста как есть, то есть не выполняя преобразования текста, перефразирования и т. д. Размер результата составляет приблизительно 25% от оригинального текста. Им можно управлять при помощи дополнительного аргумента `ratio`, определяющего пропорцию, или `word_count`, задающего фиксированное число слов. В обоих случаях итоговый результат содержит только полные предложения, то есть предложения не разбиваются на части ради соответствия требуемому размеру на выходе.

Анализ социальных сетей

Теория сетей как составная часть теории графов занимается изучением графов как представлений связей между дискретными объектами. Ее применение к социальным медиа принимает форму **анализа социальных сетей** (Social Network Analysis, SNA) – стратегии исследования социальных структур, таких как дружеские связи или знакомства.

NetworkX – одна из главных библиотек Python для создания, изменения и исследования сложных сетевых структур. Она предоставляет структуры данных для графов, а также большое число хорошо известных стандартных графовых алгоритмов.

Установить библиотеку из «сырной лавки» CheeseShop можно как обычно:

```
$ pip install networkx
```

Следующий пример показывает, как создать простой граф с несколькими узлами, представляющими пользователей, и ребрами, представляющими социальные связи между пользователями:

```
# Chap01/demo_networkx.py
import networkx as nx
from datetime import datetime

if __name__ == '__main__':
    g = nx.Graph()
    g.add_node("Джон", {'имя': 'Джон', 'возраст': 25})
    g.add_node("Питер", {'имя': 'Питер', 'возраст': 35})
    g.add_node("Мэри", {'имя': 'Мэри', 'возраст': 31})
    g.add_node("Люси", {'имя': 'Люси', 'возраст': 19})

    g.add_edge("Джон", "Мэри", {'дата рождения': datetime.today()})
    g.add_edge("Джон", "Питер", {'дата рождения': datetime(1990, 7, 30)})
    g.add_edge("Мэри", "Люси", {'дата рождения': datetime(2010, 8, 10)})

    print(g.nodes())
    print(g.edges())
    print(g.has_edge("Люси", "Мэри"))

# ['Джон', 'Питер', 'Мэри', 'Люси']
# [('Джон', 'Мэри'), ('Джон', 'Питер'), ('Мэри', 'Люси')]
# True
```

Узлы и ребра могут нести дополнительные атрибуты в форме словаря Python, которые помогают в описании семантики сети.

Класс `Graph` используется для представления неориентированного графа, в котором направление ребер не учитывается. Это понятно из использования функции `has_edge()`, которая проверяет существование ребра между Люси и Мэри. Ребро, связывающее Мэри и Люси, присутствует, но функция игнорирует его направление. Дополнительные ребра между теми же узлами игнорируются, то есть функция просто проверяет наличие хотя бы одного ребра между парой узлов. Циклы в классе `Graph`, то есть ребра, соединяющие узел с самим собой, разрешены, но в нашем примере они не потребовались.

Библиотекой NetworkX поддерживаются также другие типы графов. Это прежде всего `DiGraph` – направленный граф (направление ребер имеет значение) и их аналоги с многочисленными (параллельными) ребрами между узлами, соответственно `MultiGraph` и `MultiDiGraph`.

Визуализация данных

Визуализация данных – это область информатики, связанная с визуальным представлением данных. Визуальное представление – мощный инструмент, позволяющий исследовать сложные данные, и эффективный способ представления и передачи результатов анализа в целом. Посредством визуализации люди получают возможность увидеть аспекты, которые не заметны с первого взгляда. В конце концов, изображение стоит тысячи слов – удачная визуализация позволяет читателю впитывать сложные понятия при помощи простого изображения. Например, визуализация данных может использоваться исследователями-аналитиками во время первичного анализа данных, чтобы понять их суть. Кроме того,

исследователи-аналитики могут также использовать визуализацию для общения с неспециалистами и объяснения, что есть интересного в данных.

Python предлагает большое количество инструментов для визуализации данных, например библиотеку matplotlib, уже ипользованную нами в разделе «*Машинное обучение*» настоящей главы. Установить эту библиотеку можно командой:

```
$ pip install matplotlib
```

Библиотека matplotlib производит изображения типографского качества в многочисленных форматах. Ее цель – помочь разработчику создавать простые графики несколькими строками программного кода. Графики matplotlib можно сохранять в различных форматах, например **PNG** (Portable Network Graphics – переносимая сетевая графика) или **PDF** (Portable Document Format – переносимый формат документов).

Рассмотрим простой пример построения графика с двумерными данными:

```
# Chap01/demo_matplotlib.py
import matplotlib.pyplot as plt
import numpy as np

if __name__ == '__main__':
    # начертить график функции y = x^2 красными точками
    x = np.array([1, 2, 3, 4, 5])
    y = x * x
    plt.plot(x, y, 'ro')
    plt.axis([0, 6, 0, 30])
    plt.savefig('demo_plot.png')
```

Результат работы сценария показан на рис. 1.7.

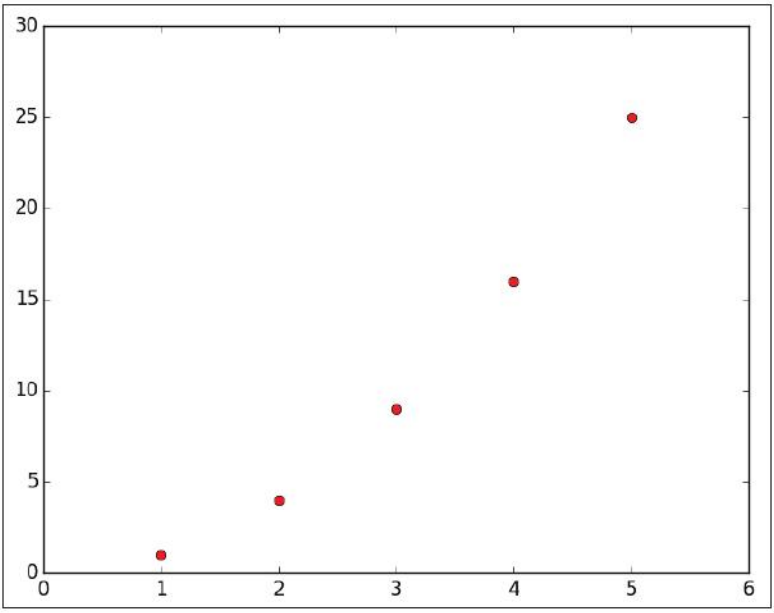

Рис. 1.7 ❖ График, созданный с помощью библиотеки matplotlib

Использование псевдонима `plt` вместо `pyploy` – общепринятое соглашение об именовании, обсуждавшееся выше. Функция `plot()` принимает два аргумента с последовательностями, содержащими координаты `x` и `y` соответственно. В этом примере координаты хранятся в массиве NumPy, но точно так же можно было бы использовать обычные списки. Функция `axis()` определяет видимый интервал для осей. Поскольку мы выводим на график квадраты чисел от 1 до 5, для оси *X* будет выбран интервал 0–6, а для оси Y – 0–30. Наконец, функция `savefig()` создает файл изображения, чей формат можно угадать по расширению файла, а итоговый результат показан на рис. 1.7.

Библиотека matplotlib производит превосходные изображения типографского качества, но иногда требуется некоторая интерактивность, чтобы позволить пользователю исследовать данные, динамически изменяя масштаб представления. Этот вид интерактивности больше относится к сфере других языков программирования, например JavaScript (в особенности посредством популярной библиотеки **D3.js**: https://d3js.org), которые позволяют создавать интерактивные веб-представления данных. Хотя это не является центральной темой настоящей книги, стоит упомянуть, что Python также обладает развитыми возможностями в данной сфере благодаря инструментам, которые транслируют объекты Python в грамматику Vega, то есть декларативный формат на основе JSON, тем самым позволяя создавать интерактивные представления.

Особенно интересна ситуация совместного использования Python и JavaScript в случае с географическими данными. Большинство социально-медийных платформ доступно через мобильные устройства. Это дает возможность отслеживать местоположение пользователя, чтобы включать в анализ географический аспект. Для кодирования и обмена разнообразными географическими структурами данных (такими как точки или многоугольники) используется общепринятый формат данных GeoJSON (http://geojson.org). Как следует из названия, этот формат основан на грамматике JSON.

Для изображения интерактивных географических карт можно использовать популярную JavaScript-библиотеку Leaflet (http://leafletjs.com). Мостом между JavaScript и Python может служить библиотека **folium** для Python, упрощающая визуализацию географических данных, обрабатываемых на Python, например, посредством передачи данных в формате GeoJSON библиотеке Leaflet.js.

Стоит также упомянуть, что такие сторонние веб-службы, как Plotly (https://plot.ly), предлагают возможность автоматического создания визуального представления данных, избавляя от необходимости создавать интерактивные компоненты в своих службах. Точнее, веб-служба Plotly предлагает поддержку по созданию заказных визуализаций данных с использованием ее клиента на Python (https://plot.ly/python). Графики размещаются веб-службой Plotly в сети и закрепляются за учетной записью пользователя (бесплатно для общего доступа, а для ограниченного доступа имеются платные тарифы).

Обработка данных в Python

После знакомства с некоторыми важными библиотеками для анализа данных на Python сделаем небольшую паузу, чтобы описать несколько интересных инструментов для загрузки и управления данными в разных форматах.

Большинство социально-медийных API предоставляет данные в формате JSON или XML. С этой точки зрения Python хорошо укомплектован пакетами поддержки этих форматов, входящими в состав стандартной библиотеки.

Чтобы не распыляться, сосредоточимся на JSON, так как этот формат прост, понятен и легко преобразуется в словари Python. Библиотека JSON имеет простой интерфейс: JSON-данные либо загружаются в словари Python, либо выгружаются из словарей Python.

Рассмотрим для примера следующий фрагмент:

```
# Chap01/demo_json.py
import json

if __name__ == '__main__':
    user_json = '{"user_id": "1", "имя": "Марко"}'
    user_data = json.loads(user_json)
    print(user_data['имя'])
    # Марко

    user_data['нравится'] = ['Python', Анализ данных']
    user_json = json.dumps(user_data, indent=4, ensure_ascii=False)
    print(user_json)
    # {
    #   "user_id": "1",
    #   "имя": "Марко",
    #   "нравится": [
    #       "Python",
    #       "Анализ данных"
    #   ]
    # }
```

Функции `json.loads()` и `json.dumps()` управляют преобразованием строк JSON в словари Python и наоборот. Обратите внимание на аргумент `ensure_ascii=False`, необходимый, если в JSON-коде используются символы Юникода, в частности кириллица. Имеются также две аналогичные функции, `json.load()` и `json.dump()`, которые работают с указателями на файлы и могут использоваться для чтения или записи JSON-данных из файлов или в файлы.

Функция `json.dumps()` также принимает второй аргумент, `indent`, определяющий ширину отступа в символах, который добавляется для структурирования вывода.

Анализируя вручную более сложные файлы JSON, порою удобно использовать внешний инструмент просмотра документов JSON, который выполняет структурную печать внутри браузера, позволяя пользователям сворачивать и разворачивать структуру по своему усмотрению.

Для этого существует множество бесплатных инструментов, часть из которых реализована в виде веб-служб, например JSON Viewer (http://jsonviewer.stack.hu). От пользователя требуется лишь вставить фрагмент документа JSON или передать URL с документом JSON, и служба загрузит и выведет его на экран в удобном для пользователя формате.

Скриншот на рис. 1.8 показывает, как документ JSON из предыдущего примера отображается в JSON Viewer.

Как видно на рис. 1.8, поле `нравится` является списком, который можно свернуть, чтобы скрыть его элементы и облегчить восприятие. В таком небольшом документе трудно оценить все преимущества такой функции, но она чрезвычайно

удобна, когда требуется исследовать сложные документы с несколькими вложенными уровнями.

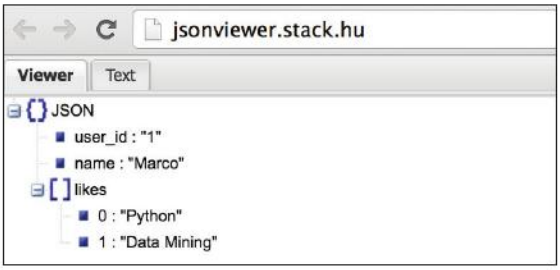

Рис. 1.8 ❖ Пример форматирования документа JSON в JSON Viewer

При использовании веб-службы или расширения для браузера загрузка больших JSON-документов для форматированного отображения может переполнить память браузера и замедлить работу системы в целом.

Создание составных конвейеров данных

Как только создаваемые для работы с данными инструменты превращаются в нечто большее, чем простой сценарий, полезно разделить задачи предварительной обработки на небольшие блоки, чтобы обозначить все этапы и зависимости конвейера данных.

Под термином *конвейер данных* подразумевается последовательность операций с данными, которая очищает, дополняет и управляет исходными данными, преобразуя их в нечто пригодное для передачи аналитическому механизму. Любой нетривиальный проект анализа данных требует наличия конвейера данных, состоящего из многочисленных этапов.

На стадии разработки прототипа эти этапы часто помещают в разные сценарии, которые затем выполняются индивидуально, например:

```
$ python скачать_данные.py
$ python очистить_данные.py
$ python дополнить_данные.py
```

Каждый сценарий в этом примере производит результат для следующего сценария, поэтому между разными этапами имеются зависимости. Из отдельных сценариев можно сформировать один большой сценарий, который делает все, что нужно, и затем выполнить его:

```
$ python выполнить_все.py
```

Содержимое такого рода сценария могло бы выглядеть, как показано в следующем примере:

```
if __name__ == '__main__':
    скачать_данные()
    очистить_данные()
    дополнить_данные()
```

Каждая функция содержит главную логику соответствующего отдельного сценария. Проблема такого решения в том, что в процессе обработки данных могут присходить ошибки, поэтому желательно также включить некоторый типовой код с операторами `try` и `except`, чтобы иметь контроль над исключениями, которые могут возникнуть. Кроме того, параметризация такого программного кода будет выглядеть немного неуклюжей.

В целом, двигаясь от разработки прототипа в сторону чего-то более стабильного, стоит подумать об использовании так называемого диспетчера потока операций. Хорошим примером может служить инструмент с открытым исходным кодом Luigi, разработанный в рамках проекта Spotify. Использование диспетчера, подобного Luigi, дает следующие преимущества:

- шаблоны задач: каждая задача обработки данных определяется как класс с несколькими методами, которые определяют порядок выполнения задачи, ее зависимости и итоговый результат;
- граф зависимостей: визуальные инструменты помогают специалисту по анализу данных представить и понять зависимости между задачами;
- восстановление после промежуточных сбоев: если конвейер потерпит неудачу на полпути, есть возможность его перезапустить, начиная с последнего успешного состояния;
- интеграция с интерфейсом командной строки и системными планировщиками задач;
- настраиваемая отчетность об ошибках.

Мы не будем вдаваться в детали всех возможностей Luigi, так как его подробное обсуждение вышло бы за рамки настоящей книги, но вы можете сами исследовать и опробовать данный инструмент, чтобы получить удобный, воспроизводимый и легко поддерживаемый и расширяемый конвейер данных.

Резюме

В данной главе мы познакомились с различными сторонами многогранной темы анализа данных из социальных медиа с использованием Python. Рассмотрели некоторые проблемы и возможности, которые делают эту тему интересной для изучения и ценной для компаний, которые хотят получать осмысленные выводы из этих данных.

После краткого знакомства с темой мы обсудили полный процесс анализа из социальных медиа, включая такие аспекты, как аутентификация при помощи OAuth. Мы также перечислили инструменты для Python, которые должен иметь под рукой любой практикующий аналитик. В зависимости от анализируемой социально-медийной платформы и типа выводов, в которых мы заинтересованы, Python предлагает надежные и хорошо проработанные программные библиотеки для машинного обучения, обработки естественного языка и анализа социальных сетей.

Мы рекомендуем настроить среду разработки на Python с помощи `virtualenv`, как описано в разделе «*Инструменты pip и virtualenv*» настоящей главы, так как это позволит держать глобальную среду разработки в чистоте.

В следующей главе мы сосредоточимся на социальной сети Twitter, в частности обсудим, как получить доступ к данным посредством программного интерфейса Twitter API и как их анализировать, чтобы получить интересную информацию.

Глава 2

Твиттер – хештеги, темы и временные ряды

Настоящая глава посвящена анализу данных из социальной сети Twitter и охватывает следующие темы:

- взаимодействие с Twitter API с использованием библиотеки Tweepy;
- данные из Twitter – анатомия твита;
- выделение лексем и частотный анализ;
- хештеги и ссылки на пользователей в твитах;
- анализ временных рядов.

Начало работы

Социальная сеть Twitter – это одна из самых известных социальных сетей, которая пользуется большой популярностью. Она предоставляет услугу *микроблогинга*, этакий вариант электронного журнала, в котором информация следует короткими порциями – сообщения, или *твиты*, в Twitter ограничены 140 символами, как СМС-сообщения. В отличие от других социально-медийных платформ, таких как Facebook, сеть Twitter не двунаправлена, то есть связи не обязательно должны быть взаимными: вы можете следовать за пользователями, которые не следуют за вами, и наоборот.

Традиционные средства массовой информации выходят в социальные сети, чтобы расширить свою аудиторию, и многие знаменитости имеют учетные записи в Twitter, чтобы поддерживать контакт со своими поклонниками. Пользователи обсуждают происходящие события в режиме реального времени, включая торжества, телевизионные сериалы, спортивные состязания, национальные выборы и многое другое.

Twitter также дал жизнь популярному термину *хештег*, описывающему тему общения, за которой могут следовать пользователи. Хештег – это одиночное ключевое слово, предваряемое символом `#`, например `#halloween` (используется для показа снимков костюмов для Хэллоуина) или `#WaterOnMars` (который пользовался большой популярностью после того, как агентство NASA объявило, что найдены доказательства присутствия воды на Марсе).

Учитывая разнообразие применений, Twitter представляет собой потенциальное золотое дно для аналитиков данных, так что давайте начнем.

Twitter API

Социальная сеть Twitter предлагает серию прикладных программных интерфейсов (API) для доступа к своим данным, включая чтение твитов, извлечение информации из профилей пользователей и отправку сообщений от имени пользователя.

Чтобы настроить проект для доступа к данным в Twitter, нужно выполнить два предварительных этапа, а именно:

- зарегистрировать приложение;
- выбрать клиента Twitter API.

Этап регистрации займет несколько минут. После входа в сеть со своей учетной записью требуется только перейти на страницу управления приложениями Application Management (Управление приложениями) по адресу: http://apps.twitter.com и создать новое приложение.

После регистрации приложения перейдите на вкладку Keys и Access Tokens (Ключи и маркеры доступа), где находится информация, необходимая для аутентификации приложения. Ключ потребителя Consumer Key и секретный ключ потребителя Consumer Secret (их также называют API Key и API Secret соответственно) – это настройки для вашего приложения. Access Token (Маркер доступа) и Access Token Secret (Секретный маркер доступа) – это, напротив, настройки вашей учетной записи. Вообще, приложение может запрашивать доступ к нескольким пользователям через их маркеры доступа. Уровень доступа Access Level определяет, какие действия может выполнять приложение в Twitter от имени пользователя: вариант read only (только для чтения) является самым ограниченным, он запрещает приложению публиковать сообщения и напрямую обмениваться сообщениями с другими пользователями.

Ограничение частоты запросов

Twitter API ограничивает доступ для приложений. Такие ограничения устанавливаются для каждого пользователя, или, если говорить точнее, маркера доступа. То есть когда приложение использует аутентификацию только для приложения, ограничение частоты запросов действует глобально для всего приложения, тогда как при использовании аутентификации от лица пользователя приложение может повысить общее число запросов к API.

Важно хорошо понимать, как действует ограничение частоты запросов, которое описано в официальной документации (https://dev.twitter.com/rest/public/rate-limiting). Также важно учитывать, что разные API накладывают разные ограничения (https://dev.twitter.com/en/docs/basics/rate-limits).

По достижении установленных пределов вместо запрошенных данных Twitter вернет сообщение об ошибке. Более того, если продолжить и дальше посылать запросы к API, время ожидания, требующееся для возобновления нормального доступа, увеличится, потому что Twitter может рассматривать нас как потенциальных злоумышленников. Если приложению может потребоваться послать много запросов, необходимо предусмотреть способ избежать ситуации такого рода. В стандартной библиотеке Python имеется модуль `time`. Он позволяет при-

останавливать выполнение программного кода на произвольные интервалы вызовом функции `time.sleep()`, как демонстрирует следующий псевдокод:

```
# Допустим, определены функции первый_запрос() и второй_запрос().
# Они выполняют запросы к API.
import time

первый_запрос()
time.sleep(10)
второй_запрос()
```

В данном случае второй запрос выполнится через 10 секунд (как определено в аргументе функции `sleep()`) после первого.

Поиск и потоковая обработка

Twitter поддерживает несколько API. В следующем разделе рассказывается о нескольких способах получения доступа к данным из Twitter. Для простоты разделим их на два класса: обычный поисковый REST API и потоковый Streaming API:

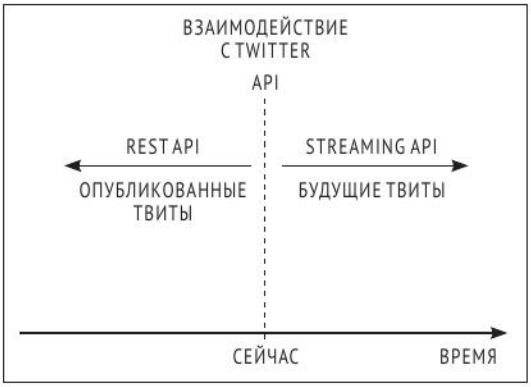

Рис. 2.1 ❖ Области применения поискового и потокового API

На рис. 2.1 демонстрируется разница между этими двумя классами: все поисковые REST API позволяют обращаться только к прошлому. Взаимодействуя с Twitter через REST API, можно искать фактически существующие твиты – опубликованные и доступные для поиска. Нередко эти API ограничивают количество твитов, которые можно получить, ограничивая не только частоту запросов, как обсуждалось в предыдущем разделе, но также интервал времени. Обычно для доступа открыты данные не более чем недельной давности, то есть более старые твиты нельзя извлечь. Второй важный аспект REST API состоит в том, что они не гарантируют возврата абсолютно всех сообщений, которые были опубликованы в Twitter, так как некоторые могут быть недоступны для поиска или просто еще не были проиндексированы.

Потоковый API, напротив, заглядывает в будущее. После установки соединения его можно оставить открытым и двигаться вперед по линии времени. Сохраняя HTTP-соединение открытым, можно получать все твиты, соответствующие заданным критериям фильтрации, по мере их публикации.

В общем случае потоковый интерфейс Streaming API предпочтительнее, когда требуется загрузить большое количество твитов, так как взаимодействие с платформой ограничивается лишь поддержкой соединения открытым. Недостаток такого подхода заключается в его продолжительности, потому что приходится ждать публикации твитов, прежде чем получится их собрать.

Подводя итог, можно сказать, что REST API удобно использовать, когда требуется отыскать твиты, созданные определенным пользователем, или получить доступ к своей собственной ленте, а Streaming API может пригодиться, когда требуется выполнить фильтрацию по конкретному ключевому слову и скачать связанный с ним массивный объем твитов (например, события в прямом эфире).

Выборка данных из Twitter

Для взаимодействий с Twitter API нам нужен клиент на Python, реализующий различные формы обращений к API. Как следует из официальной документации (https://dev.twitter.com/overview/api/twitter-libraries), у нас есть несколько вариантов на выбор. Ни один из них не поддерживается компанией Twitter официально, и все они созданы и развиваются сообществом разработчиков ПО с открытым исходным кодом. Несмотря на кажущееся разнообразие, некоторые варианты почти эквивалентны, поэтому мы остановимся на библиотеке **Tweepy** как обладающей достаточно широкими возможностями и активной поддержкой.

Эту библиотеку можно установить с помощью pip:

```
$ pip install tweepy==3.3.0
```

Совместимость с Python 3. Мы использовали версию Tweepy 3.3 из-за проблем совместимости последней версии Tweepy с Python 3, мешавших выполнению примеров в нашей среде Python 3.4. Во время написания настоящей главы данная проблема по-прежнему оставалась нерешенной, но, по всей видимости, вскоре будет исправлена.

Первый шаг на пути к взаимодействию с Twitter API, как обсуждалось в предыдущем разделе, включает настройку аутентификации. После регистрации приложения у вас должны иметься: ключ потребителя, секретный ключ потребителя, маркер доступа и секретный маркер доступа.

Чтобы отделить прикладную логику от конфигурации, сохраним учетные данные в переменных окружения. Почему просто не использовать переменные Python? Для этого имеется несколько причин, хорошо описанных в Двенадцатифакторном манифесте (https://12factor.net/ru/). Использование окружения для хранения конфигурации позволяет обеспечить независимость от языка и ОС. Изменение конфигурации для разных случаев (например, использование персональных учетных данных для локального тестирования и корпоративной учетной записи для эксплуатации) не требует изменения прикладного кода. Кроме того, переменные окружения невозможно по неосторожности передать в систему управления версиями и сделать доступными для всеобщего обозрения.

В Unix-подобных системах, таких как Linux или macOS, при использовании командной оболочки Bash переменные окружения можно определить так:

```
$ export TWITTER_CONSUMER_KEY="ваш-ключ-потребителя"
```

В Windows:

```
$ set TWITTER_CONSUMER_KEY=ваш-ключ-потребителя
```

Данную команду нужно повторить для всех четырех переменных, и она будет действительна только для активного окна командной строки. Чтобы сделать настройки постоянными, используйте команду `setx TWITTER_CONSUMER_KEY=ваш-ключ-потребителя`.

После настройки окружения создадим две функции, обращающиеся к библиотеке Tweepy: одну – для чтения переменных окружения и аутентификации, другую – для создания объекта, взаимодействующего с Twitter:

```
# Chap02-03/twitter_client.py
import os
import sys
from tweepy import API
from tweepy import OAuthHandler

def get_twitter_auth():
    """Читает настройки и выполняет аутентификацию в Twitter.

    Возвращает: объект tweepy.OAuthHandler
    """
    try:
        consumer_key = os.environ['TWITTER_CONSUMER_KEY']
        consumer_secret = os.environ['TWITTER_CONSUMER_SECRET']
        access_token = os.environ['TWITTER_ACCESS_TOKEN']
        access_secret = os.environ['TWITTER_ACCESS_SECRET']
    except KeyError:
        sys.stderr.write("Переменные окружения TWITTER_* не заданы\n")
        sys.exit(1)
    auth = OAuthHandler(consumer_key, consumer_secret)
    auth.set_access_token(access_token, access_secret)
    return auth

def get_twitter_client():
    """Создает клиента Twitter API.

    Возвращает: объект tweepy.API
    """
    auth = get_twitter_auth()
    client = API(auth)[1]
    return client
```

Функция `get_twitter_auth()` отвечает за аутентификацию. Блок `try/except` показывает, как читать переменные окружения. Модуль `os` содержит словарь `os.environ`, к которому можно обращаться как к обычному словарю, используя в качестве ключей имена переменных окружения. Если одна из переменных окружения `TWITTER_*` отсутствует, попытка обратиться к ключу возбудит исключение `KeyError`, которое мы перехватываем, чтобы вывести сообщение об ошибке и выйти из приложения.

Эта функция вызывается из `get_twitter_client()`, которая создает экземпляр `tweepy.API`, предназначенный для взаимодействия с Twitter. Причина разделения

---

[1] Чтобы автоматически учесть ограничение частоты запросов, выполняя паузы во время обработки, объект клиента можно создать так:

```
API(auth_handler=auth, wait_on_rate_limit=True, wait_on_rate_limit_notify=True)
```

– *Прим. перев.*

логики на две отдельные функции состоит в том, что программный код аутентификации можно повторно использовать и для доступа к потоковому интерфейсу Streaming API, как описывается в следующих разделах.

Получение твитов из ленты

После успешной аутентификации клиента можно приступить к его использованию для загрузки твитов.

Давайте сначала рассмотрим простой сценарий: как получить первые десять твитов из вашей собственной ленты:

```
# Chap02/twitter_get_home_timeline_toscreen.py
from tweepy import Cursor
from twitter_client import get_twitter_client

if __name__ == '__main__':
    client = get_twitter_client()

    for status in Cursor(client.home_timeline).items(10):
        # Обработать одиночный статус
        print(status.text)
```

Ваша домашняя лента – это список, который отображается перед вами после входа в Twitter. Он содержит последовательность твитов из учетных записей, которые вы решили читать, при этом новые и интересные твиты находятся вверху.

Фрагмент выше показывает, как использовать `tweepy.Cursor` для обхода первых десяти элементов вашей домашней ленты. Прежде всего необходимо импортировать объект `Cursor` и функцию `get_twitter_client`, которая была определена ранее. Затем вызвать эту функцию в главном блоке, чтобы создать клиента, который обеспечит доступ к Twitter API. В частности, для доступа к собственной домашней ленте необходим атрибут `home_timeline`, который мы передаем объекту `Cursor`.

`tweepy.Cursor` – это *итерируемый* объект, то есть он имеет простой в использовании интерфейс для выполнения повторяющихся операций и разбиения на страницы по различным объектам. Как показывает пример выше, он реализует минимальную абстракцию, которая позволяет разработчику просматривать содержимое объекта в цикле, не беспокоясь о том, как выполняются запросы к Twitter API.

Итерируемые объекты. Итерируемым считается объект, способный возвращать по одному своему элементу в ответ на каждое обращение. Итерируемыми объектами являются такие встроенные типы данных, как списки или словари, а также объекты любого класса, реализующего метод `__iter__()` или `__getitem__()`.

Переменная цикла, `status`, представляет экземпляр объекта `tweepy.Status`, который в библиотеке Tweepy используется для обертки статусов (то есть сообщений, или твитов). В предыдущем фрагменте мы просто используем его атрибут `text`, но этот объект имеет много других атрибутов. Они все описаны в разделе «*Структура твита*».

Вместо простого вывода на экран хотелось бы сохранить получаемые твиты, чтобы позднее проанализировать их.

Для этого переделаем предыдущий фрагмент и сохраним информацию в файле в формате JSON:

```
# Chap02-03/twitter_get_home_timeline.py
import json
from tweepy import Cursor
from twitter_client import get_twitter_client

if __name__ == '__main__':
    client = get_twitter_client()

    with open('home_timeline.jsonl', 'w') as f:
        for page in Cursor(client.home_timeline, count=200).pages(4):
            for status in page:
                f.write(json.dumps(status._json)+"\n")
```

Через какое-то время этот сценарий создаст файл `home_timeline.jsonl`.

Формат JSON Lines. Предыдущий пример создает файл с расширением `.jsonl` вместо обычного `.json`. Фактически этот файл имеет формат **JSON Lines** (http://jsonlines.org/), то есть каждая строка в нем является допустимым документом JSON. Попытка загрузить все содержимое этого файла, например с помощью функции `json.loads()`, возбудит исключение `ValueError`, потому что все содержимое не является единым допустимым документом JSON. Если в программе используются функции обработки допустимых документов JSON, такие файлы следует обрабатывать построчно. Формат JSON Lines особенно хорошо подходит для обработки больших объемов данных: многие фреймворки обработки больших данных позволяют разработчикам делить входной файл на блоки, которые можно обрабатывать несколькими рабочими процессами параллельно.

В этом примере мы выполняем обход четырех страниц, содержащих по 200 твитов, как объявлено в аргументе `count` объекта-курсора. Причина – ограничение в Twitter: из своей домашней ленты можно извлечь не более 800 последних твитов.

Если твиты извлекаются из ленты конкретного пользователя, то есть с использованием метода `user_timeline` вместо `home_timeline`, этот предел увеличивается до 3200.

Рассмотрим следующий сценарий, извлекающий ленту конкретного пользователя:

```
# Chap02-03/twitter_get_user_timeline.py
import sys
import json
from tweepy import Cursor
from twitter_client import get_twitter_client

if __name__ == '__main__':
    user = sys.argv[1]
    client = get_twitter_client()

    fname = "user_timeline_{}.jsonl".format(user)

    with open(fname, 'w') as f:
        for page in Cursor(client.user_timeline,
                           screen_name=user,
                           count=200).pages(16):
            for status in page:
                f.write(json.dumps(status._json)+"\n")
```

Этот сценарий принимает один аргумент командной строки с именем пользователя. Например, получить всю мою ленту (которая на данный момент значительно меньше предела в 3200 твитов) можно следующей командой:

```
$ python twitter_get_user_timeline.py marcobonzanini
```

Моя лента относительно невелика, поэтому ради эксперимента можно попробовать получить более длинную ленту из учетной записи `@PacktPub` издательства Packt Publishing:

```
$ python twitter_get_user_timeline.py PacktPub
```

Подобно предыдущему сценарию, извлекающему домашнюю ленту, этот сценарий также создаст файл `.jsonl`, каждая строка которого представляет один документ JSON. По достижении предела в 3200 твитов размер файла составит порядка 10 Мбайт.

Концептуально код извлечения ленты пользователя мало отличается от кода, извлекающего домашнюю ленту. Единственное отличие – аргумент командной строки, используемый для передачи дескриптора пользователя.

До этого момента мы использовали единственный атрибут интерфейса библиотеки Tweepy, представляющего твит, – атрибут `_json`, – чтобы сохранить сообщение в формате JSON. В следующем разделе подробно обсуждается структура твита, что позволит нам перейти к другим способам получения данных из Twitter.

Структура твита

Твит – сложный объект. Следующая таблица перечисляет все его атрибуты и дает краткое их описание. В частности, полный ответ Twitter API с отдельным твитом находится в атрибуте `_json`, который загружается в словарь Python:

Название атрибута	Описание
`_json`	Словарь с ответом в формате JSON
`author`	Экземпляр `tweepy.models.User` автора твита
`contributors`	Список участников, если поддерживается
`coordinates`	Словарь с координатами в формате GeoJSON
`created_at`	Экземпляр `datetime.datetime` со временем создания твита
`entities`	Словарь с URL-адресами, хештегами и упоминаниями в твитах
`favorite_count`	Число пользователей, которым понравился твит
`favorited`	Отмечает, понравился ли твит аутентифицированному пользователю
`geo`	Координаты (устаревший, вместо него использовать `coordinates`)
`id`	Уникальный идентификатор твита в виде большого целого числа
`id_str`	Уникальный идентификатор твита в виде строкового значения
`in_reply_to_screen_name`	Экранное имя твита, на который отвечает этот твит
`in_reply_to_status_id`	Идентификатор твита, на который отвечает этот твит (в виде большого целого числа)
`in_reply_to_status_id_str`	Идентификатор твита, на который отвечает этот твит (в виде строкового значения)
`in_reply_to_user_id`	Идентификатор пользователя, оставившего твит, на который отвечает этот твит (в виде большого целого числа)
`in_reply_to_user_id_str`	Идентификатор пользователя, оставившего твит, на который отвечает этот твит (в виде строкового значения)

Название атрибута	Описание
is_quote_status	Признак: является ли твит цитатой (то есть содержит другой твит)
lang	Строк с кодом языка твита
place	Экземпляр tweepy.models.Place места, прикрепленного к твиту
possibly_sensitive	Признак наличия URL-адреса в твите, возможно с конфиденциальным материалом
retweet_count	Сколько раз это сообщение было ретвитнуто
retweeted	Является ли это сообщение ретвитом
source	Строка, описывающая инструмент, использованный для отправки статуса
text	Строка с текстом сообщения
truncated	Признак усечения твита (например, ретвит, превышающий 140 символов)
user	Экземпляр tweepy.models.User автора твита (устаревший, вместо него использовать author)

Атрибуты с идентификаторами (например, пользователя или твита) имеют дубликаты с теми же значениями в виде строки. Такая необходимость обусловлена тем, что некоторые языки программирования (прежде всего JavaScript) не поддерживают чисел, включающих более 53 разрядов, тогда как Twitter использует 64-разрядные числа. Чтобы избежать подобных проблем, Twitter рекомендует использовать атрибуты *_str.

Как видите, не все атрибуты транслируются во встроенные типы Python, такие как строковой или логический тип. На самом деле в ответ API включены некоторые сложные объекты, такие как профиль пользователя, и библиотека Tweepy следит за преобразованием этих объектов в соответствующие модели.

Ниже демонстрируется пример твита издательства Packt Publishing в исходном формате JSON, доступном через атрибут _json (для краткости некоторые поля опущены).

Прежде всего дата создания:

```
{
  "created_at": "Fri Oct 30 05:26:05 +0000 2015",
```

Атрибут entities – это словарь, содержащий различные списки с метками сущностей. Например, элемент hashtags показывает, что в данном твите присутствует хештег #Python. Точно так же имеются фотографии, URL-адреса и упоминания пользователей:

```
  "entities": {
    "hashtags": [
      {
        "indices": [
          72,
          79
        ],
        "text": "Python"
      }
    ],
    "media": [
      {
        "display_url": "pic.twitter.com/muZlMreJNk",
```

```
      "id": 659964467430735872,
      "id_str": "659964467430735872",
      "indices": [
        80,
        103
      ],
      "type": "photo",
      "url": "https://t.co/muZlMreJNk"
    }
  ],
  "symbols": [],
  "urls": [
    {
      "indices": [
        48,
        71
      ],
      "url": "https://t.co/NaBNan3iVt"
    }
  ],
  "user_mentions": [
    {
      "id": 80589255,
      "id_str": "80589255",
      "indices": [
        33,
        47
      ],
      "name": "Open Source Way",
      "screen_name": "opensourceway"
    }
  ]
},
```


Атрибуты, следующие далее, описаны в таблице выше и не требуют дополнительных пояснений. Обратите внимание на отсутствие информации о географическом положении в этом твите. Можно также сравнить предыдущую информацию о сущностях с фактическим содержимым атрибута `text`:


```
"favorite_count": 4,
"favorited": false,
"geo": null,
"id": 659964467539779584,
"id_str": "659964467539779584",
"lang": "en",
"retweet_count": 1,
"retweeted": false,
"text": "Top 3 open source Python IDEs by @opensourceway
  https://t.co/NaBNan3iVt #Python https://t.co/muZlMreJNk",
```


Атрибут `user` – это словарь, представляющий пользователя, который отправил твит, в данном случае `@PacktPub`. Как уже упоминалось выше, это сложный объект с информацией о пользователе, встроенный в твит:

```
  "user": {
    "created_at": "Mon Dec 01 13:16:47 +0000 2008",
    "description": "Providing books, eBooks, video tutorials, and
      articles for IT developers, administrators, and users.",
    "entities": {
      "description": {
        "urls": []
      },
      "url": {
        "urls": [
          {
            "display_url": "PacktPub.com",
            "expanded_url": "http://www.PacktPub.com",
            "indices": [
              0,
              22
            ],
            "url": "http://t.co/vEPCgOu235"
          }
        ]
      }
    },
    "favourites_count": 548,
    "followers_count": 10090,
    "following": true,
    "friends_count": 3578,
    "id": 17778401,
    "id_str": "17778401",
    "lang": "en",
    "location": "Birmingham, UK",
    "name": "Packt Publishing",
    "screen_name": "PacktPub",
    "statuses_count": 10561,
    "time_zone": "London",
    "url": "http://t.co/vEPCgOu235",
    "utc_offset": 0,
    "verified": false
  }
}
```


Этот пример демонстрирует два интересных аспекта, которые следует учитывать при анализе твитов, а именно:

- сущности имеют метки;
- профиль пользователя полностью встроен в твит.

Первый пункт означает упрощение анализа сущностей, так как избавляет от необходимости явно искать такие сущности, как хештеги, упоминания пользователей, встроенные URL-адреса или изображения, потому что все они передаются Twitter API с их смещениями в тексте (в атрибуте `indices`).

Второй пункт означает отсутствие необходимости хранить информацию из профиля пользователя где-то в другом месте и затем выполнять операцию соединения/слияния по внешнему ключу, если потребуется. Профиль пользователя, по сути, избыточно повторяется в каждом твите.

Работа с денормализованными данными. Прием встраивания избыточных данных связан с понятием денормализации. Нормализация считается хорошей практикой в проектировании реляционных баз данных, а денормализация находит применение в крупномасштабной обработке и базах данных, принадлежащих широкому семейству NoSQL. Оправданием для такого подхода является существенный прирост производительности из-за устранения операций по соединению/слиянию, перекрывающий издержки на хранение избыточной информации из профиля пользователя.

Применение потокового интерфейса Streaming API

Потоковый интерфейс Streaming API является одним из предпочтительных способов получения массивных объемов данных без ограничения на частоту запросов. Мы уже знаем, чем этот API отличается от других интерфейсов, в частности поискового Search API, и что нам, возможно, придется заново продумать характер взаимодействия нашего приложения с пользователем.

Прежде чем углубиться в детали, стоит заглянуть в описание Streaming API, чтобы понять его особенности (https://dev.twitter.com/streaming/overview).

В данном разделе мы реализуем прием потока, расширив класс `StreamListener`, предлагаемый библиотекой Tweepy:

```
# Chap02-03/twitter_streaming.py
import sys
import string
import time
from tweepy import Stream
from tweepy.streaming import StreamListener
from twitter_client import get_twitter_auth

class CustomListener(StreamListener):
    """Специальный приемник потока StreamListener
    для получения потоковых данных из Twitter."""

    def __init__(self, fname):
        safe_fname = format_filename(fname)
        self.outfile = "stream_%s.jsonl" % safe_fname

    def on_data(self, data):
        try:
            with open(self.outfile, 'a') as f:
                f.write(data)
                return True
        except BaseException as e:
            sys.stderr.write("Ошибка on_data: {}\n".format(e))
            time.sleep(5)
        return True

    def on_error(self, status):
        if status == 420:
            sys.stderr.write("Превышен предел частоты запросов\n")
            return False
        else:
            sys.stderr.write("Ошибка {}\n".format(status))
            return True
```

```
    def format_filename(fname):
        """Преобразует имя файла fname
        в безопасную строку для имени файла.

        Возвращает: строку с именем
        """
        return ''.join(convert_valid(one_char) for one_char in fname)

    def convert_valid(one_char):
        """Преобразует символ в '_', если он "недопустим".

        Возвращает: строку с символом
        """
        valid_chars = "-_.%s%s" % (string.ascii_letters, string.digits)
        if one_char in valid_chars:
            return one_char
        else:
            return '_'

if __name__ == '__main__':
    query = sys.argv[1:]          # список аргументов командной строки
    query_fname = ' '.join(query) # строковое значение
    auth = get_twitter_auth()
    twitter_stream = Stream(auth, CustomListener(query_fname))
    twitter_stream.filter(track=query, async=True)
```

Основная логика приема потока сосредоточена в классе `CustomListener`, который наследует класс `StreamListener` и переопределяет два метода: `on_data()` и `on_error()`. Эти обработчики вызываются при получении данных или сообщения об ошибке.

Оба метода возвращают логическое значение: `True` – можно продолжить прием потока и `False` – остановить выполнение. По этой причине значение `False` должно возвращаться только в случае фатальных ошибок, чтобы приложение могло продолжить загрузку данных. Благодаря этому можно избежать преждевременной остановки приложения в случае временной ошибки, например в результате отключения сети на нашей стороне или ошибки HTTP 503 в Twitter, то есть когда служба временно недоступна (но, вероятно, вскоре возобновит работу).

Метод `on_error()` обрабатывает сообщения об ошибках, получаемые от Twitter. Найти полный список кодов ошибок в Twitter API можно в документации (https://dev.twitter.com/overview/api/response-codes). Наша реализация `on_error()` останавливает выполнение, только если получена ошибка 420, означающая превышение ограничения на количество запросов. Чем чаще превышается этот предел, тем дольше требуется ждать, прежде чем снова можно будет использовать службу. Поэтому лучше прекратить загрузку и исследовать проблему. Любое сообщение об ошибке посылается в стандартный поток `stderr`. Такой подход в целом лучше, чем простое использование функции `print()`, так как при необходимости позволяет перенаправить вывод ошибок в определенный файл. Также, что еще лучше, можно воспользоваться модулем `logging` и создать надлежащий механизм журналирования событий, но это выходит за рамки настоящей главы.

Метод `on_data()` вызывается при получении данных. Он просто записывает данные в файл `.jsonl` по мере их поступления. Каждая строка в этом файле будет содержать один твит в формате JSON. После записи в файл метод возвращает `True`,

сообщая приложению, что можно продолжить работу. Если что-то пойдет не так, метод перехватит любое исключение, выведет сообщение в `stderr`, приостановит приложение на пять секунд и затем продолжит выполнение, вернув `True`. Кратковременная приостановка в случае исключения помогает предотвратить зависание приложения из-за случайных задержек в сети.

Класс `CustomListener` использует вспомогательную функцию для экранирования служебных символов в запросе и применения его в качестве имени файла. Функция `format_filename()`, по сути, посимвольно просматривает полученную строку и с помощью функции `convert_valid()` преобразует недопустимые символы в подчеркивания. В данном контексте допустимыми являются три символа – тире, подчеркивание и точка (-, _ и .), буквы ASCII и цифры.

Запуская сценарий `twitter_streaming.py`, мы должны передать ему аргументы командной строки. Эти аргументы, разделенные пробелами, будут служить ключевыми словами, используемыми для определения загружаемых твитов.

Например: я попробовал использовать этот сценарий, чтобы получить твиты о финальном матче чемпионата мира по регби 2015 между Новой Зеландией и Австралией (https://en.wikipedia.org/wiki/2015_Rugby_World_Cup). Болельщики, следившие за событием в Twitter, в основном использовали хештеги `#RWC2015` (действовавший в течение всего чемпионата) и `#RWCFinal` (действовавший для обсуждения последнего дня соревнований). Я использовал эти два хештега, а также слово rugby в качестве поисковых ключевых слов:

```
$ python twitter_streaming.py \#RWC2015 \#RWCFinal rugby
```

Символы обратной наклонной черты перед хеш-символами `#` необходимы, чтобы их экранировать, потому что в командной оболочке хеш-символ используется как признак начала комментария. Экранирование гарантирует правильную передачу строкового значения в сценарий.

Начало матча было намечено на 16:00 по **Гринвичу** (GMT) 31 октября 2015. За три часа работы, с 15:00 до 18:00, сценарий сохранил почти 800 Мбайт данных с более чем 200 000 твитов. Ажиотаж в Twitter еще некоторое время продолжался после окончания события, но собранного в течение этого периода времени количества данных было достаточно, чтобы провести интересный анализ.

Поскольку Twitter API возвращает только недавние твиты, максимум недельной давности, если выполнить сценарий, как показано выше, он попытается найти свежие твиты, но никак не 2015 года. Чтобы по достоинству оценить мощь Streaming API и провести анализ данных, необходимо подобрать какое-нибудь предстоящее событие в области спорта, политики, шоу-бизнеса и пр., взять его хештег или серию хештегов, как это сделано в примере, и запустить сценарий `twitter_streaming.py`, то есть подключиться к событию, указав временные рамки. Автор книги писал настоящую главу в предверии финального матча чемпионата мира по регби, что позволило ему собрать данные, которые он далее часто использует в главе. Чтобы смоделировать ситуацию, в загружаемые примеры к данной главе влючены набор данных с идентификаторами твитов, о которых пишет автор (`rwc2015_rwcfinal_rugby.csv`), в формате (идентификатор твита, идентификатор пользователя) и сценарий `twitter_get_oldtweets_in_bulk.py`, который в цикле извлекает полные твиты за 31 октября 2015 по их идентификаторам и создает файл `stream__RWC2015__RWCFinal_Rugby.jsonl`, используемый далее в этой главе. Кроме того, ради экономии вашего времени в загружаемые примеры также включен уже готовый файл `stream__RWC2015__RWCFinal_Rugby.jsonl`.

Анализ твитов – сущности

Данный раздел полностью посвящен анализу сущностей в твитах. Мы проведем частотный анализ на основе данных, собранных в предыдущем разделе. Это исследование поможет получить несколько интересных статистик, способных помочь проникнуть в суть данных и ответить на некоторые вопросы.

Анализ таких сущностей, как хештеги, интересен тем, что такими аннотациями автор явно отмечает тему твита.

Начнем с анализа твитов издательства Packt Publishing. Поскольку это издательство поддерживает и продвигает программное обеспечение с открытым исходным кодом, было бы интересно узнать, какие технологии упоминаются издательством наиболее часто.

Следующий сценарий извлекает хештеги из ленты пользователя и производит список наиболее распространенных из них:

```
# Chap02-03/twitter_hashtag_frequency.py
import sys
from collections import Counter
import json

def get_hashtags(tweet):
    entities = tweet.get('entities', {})
    hashtags = entities.get('hashtags', [])
    return [tag['text'].lower() for tag in hashtags]

if __name__ == '__main__':
    fname = sys.argv[1]
    with open(fname, 'r') as f:
        hashtags = Counter()
        for line in f:
            tweet = json.loads(line)
            hashtags_in_tweet = get_hashtags(tweet)
            hashtags.update(hashtags_in_tweet)
        for tag, count in hashtags.most_common(20):
            print("{}: {}".format(tag, count))
```

Его можно запустить следующей командой:

```
$ python twitter_hashtag_frequency.py user_timeline_PacktPub.jsonl
```

Здесь `user_timeline_PacktPub.jsonl` – это файл JSON Lines, созданный ранее.

Данный сценарий принимает аргумент командной строки с именем файла `.jsonl` и построчно читает его содержимое. Поскольку каждая строка содержит документ JSON, сценарий загружает строку в переменную `tweet` и вызывает вспомогательную функцию `get_hashtags()`, чтобы извлечь список хештегов. Этот тип сущностей сохраняется в переменной `hashtags`, которая объявляется как `collections.Counter`, то есть специальный тип словаря, который используется для подсчета хешируемых объектов – в нашем случае строк. Словарь `hashtags` хранит строковые ключи и соответствующие им частоты – в качестве значений.

Будучи подклассом типа `dict`, объект `Counter` является неупорядоченной коллекцией. Его метод `most_common()` отвечает за упорядочение ключей в зависимости от их значений (в порядке убывания частоты) и возвращает список кортежей `(ключ, значение)`.

Вспомогательная функция `get_hashtags()` извлекает список хештегов из твита. В качестве единственного аргумента ей передается весь твит, загруженный в словарь. Если сущности в твите присутствуют, словарь будет иметь ключ `entities`. Но, так как он может отсутствовать, нельзя напрямую обратиться к твиту, как `tweet['entities']`, потому что эта инструкция возбудит исключение `KeyError` в отсутствие данного ключа, поэтому мы воспользуемся функцией `get()`, передав ей пустой словарь как значение по умолчанию, возвращаемое в отсутствие сущностей. Второй шаг – извлечение хештегов из сущностей. Поскольку `entities` – это тоже словарь, а ключ `hashtags` тоже может отсутствовать, мы снова воспользуемся функцией `get()`, но на этот раз в качестве значения по умолчанию укажем пустой список. Наконец, воспользуемся генератором списков, чтобы выполнить обход хештегов и извлечь из них текст. Хештеги нормализуются функцией `lower()`, которая переводит текст в нижний регистр, поэтому такие упоминания, как `#Python` или `#PYTHON`, все будут сгруппированы в `#python`.

Анализ твитов издательства PacktPub с помощью этого сценария дал следующий результат:

```
packt5dollar: 217
webdev: 147
python: 122
javascript: 121
freelearning: 110
machinelearning: 98
gamedev: 83
appdev: 79
rstats: 61
tech: 58
data: 52
mapt: 52
java: 45
datascience: 45
devops: 43
ai: 38
nodejs: 37
infosec: 36
deeplearning: 35
docker: 28
```

Как видите, здесь присутствуют упоминания событий и рекламных акций издательства PacktPub (таких как `#packt5dollar`), но чаще хештеги упоминают конкретные технологии, среди которых чаще других встречаются Python и JavaScript.

Сценарий выше помог выявить хештеги, наиболее часто используемые издательством PacktPub, но хотелось бы копнуть поглубже. По сути, можно получить более описательные статистики, помогающие получить общую картину использования хештегов в издательстве Packt Publishing:

```
# Chap02-03/twitter_hashtag_stats.py
import sys
from collections import defaultdict
import json
```

```
def get_hashtags(tweet):
    entities = tweet.get('entities', {})
    hashtags = entities.get('hashtags', [])
    return [tag['text'].lower() for tag in hashtags]

def usage():
    print("Применение:")
    print("python {} <имя_файла.jsonl>".format(sys.argv[0]))

if __name__ == '__main__':
    if len(sys.argv) != 2:
        usage()
        sys.exit(1)
    fname = sys.argv[1]
    with open(fname, 'r') as f:
        hashtag_count = defaultdict(int)
        for line in f:
            tweet = json.loads(line)
            hashtags_in_tweet = get_hashtags(tweet)
            n_of_hashtags = len(hashtags_in_tweet)
            hashtag_count[n_of_hashtags] += 1
        tweets_with_hashtags = sum([count for n_of_tags, count in
                                    hashtag_count.items()
                                    if n_of_tags > 0])
        tweets_no_hashtags = hashtag_count[0]
        tweets_total = tweets_no_hashtags + tweets_with_hashtags
        tweets_with_hashtags_percent = "%.2f" % (tweets_with_hashtags
                                                 / tweets_total * 100)
        tweets_no_hashtags_percent = "%.2f" % (tweets_no_hashtags
                                                 / tweets_total * 100)
        print("{} твитов без хештегов
             ({}%)".format(tweets_no_hashtags,
              tweets_no_hashtags_percent))
        print("{} твитов как минимум с одним хештегом
              ({}%)".format(tweets_with_hashtags,
              tweets_with_hashtags_percent))

        for tag_count, tweet_count in hashtag_count.items():
            if tag_count > 0:
                percent_total = "%.2f" % (tweet_count / tweets_total * 100)
                percent_elite = "%.2f" % (tweet_count /
                                          tweets_with_hashtags * 100)
                print("{} твитов с {} хештегами (всего {}% , элитных {}%)"
                      .format(tweet_count, tag_count,
                              percent_total, percent_elite))
```

Запустить этот сценарий можно следующей командой:

```
$ python twitter_hashtag_stats.py user_timeline_PacktPub.jsonl
```

На основе собранных данных эта команда выведет следующее:

```
1319 твитов без хештегов (41.24%)
1879 твитов как минимум с одним хештегом (58.76%)
831 твит с 1 хештегом (всего 25.98%, элитных 44.23%)
```

```
611 твитов с 2 хештегами (всего 19.11%, элитных 32.52%)
291 твит с 3 хештегами (всего 9.10%, элитных 15.49%)
104 твита с 4 хештегами (всего 3.25%, элитных 5.53%)
29 твитов с 5 хештегами (всего 0.91%, элитных 1.54%)
11 твитов с 6 хештегами (всего 0.34%, элитных 0.59%)
1 твит с 7 хештегами (всего 0.03%, элитных 0.05%)
1 твит с 8 хештегами (всего 0.03%, элитных 0.05%)
```

Как видите, большинство твитов издательства PacktPub имеет, по крайней мере, один хештег, что подтверждает важность этого типа сущности в общении людей в Twitter.

С другой стороны, разбивка по количеству хештегов показывает, что обычно в твитах используется небольшое число хештегов. Приблизительно 1% твитов имеет четыре хештега и более. Для этой разбивки мы будем наблюдать за двумя разными процентными показателями: первый вычисляется на общем количестве твитов, а второй – на выборке, по крайней мере, с одним твитом (именуемой элитным набором).

Аналогично можно проанализировать упоминание пользователей:

```
# Chap02-03/twitter_mention_frequency.py
import sys
from collections import Counter
import json

def get_mentions(tweet):
    entities = tweet.get('entities', {})
    hashtags = entities.get('user_mentions', [])
    return [tag['screen_name'] for tag in hashtags]

if __name__ == '__main__':
    fname = sys.argv[1]
    with open(fname, 'r') as f:
        users = Counter()
        for line in f:
            tweet = json.loads(line)
            mentions_in_tweet = get_mentions(tweet)
            users.update(mentions_in_tweet)
        for user, count in users.most_common(20):
            print("{}: {}".format(user, count))
```

Этот сценарий можно выполнить следующей командой:

```
$ python twitter_mention_frequency.py user_timeline_PacktPub.jsonl
```

Он произведет следующий результат:

```
PacktPub: 306
Raspberry_Pi: 36
nodejs: 29
java: 29
legacy99: 29
angular: 26
Android: 26
Dhabolt: 25
antocucciniello: 24
```

```
Docker: 23
HariOfSpades: 23
ThePracticalDev: 16
b0rk: 13
cbronline: 13
Pybonacci: 12
sarah_edo: 12
arifmahmudrana: 12
unity3d: 11
VentureBeat: 11
SaraSoueidan: 11
```

Анализ твитов – текст

В предыдущем разделе было проанализировано поле сущностей твита `entities`. Оно содержит полезные сведения о твите, потому что сущности явно курируются автором твита. В настоящем разделе в центре внимания будут исключительно неструктурированные данные, то есть исходный текст твита. Мы обсудим аспекты анализа текста, такие как предварительная обработка и нормализация, и выполним небольшой статистический анализ твитов. Прежде чем углубиться в детали, познакомимся с терминологией.

Лексемизация – один из важных этапов предварительной обработки. Лексемизация – это процесс разбивки текста на отдельные элементы, лексемы. В самой простой форме лексемы – это слова, но также можем выделять более сложные лексемы, такие как словосочетания, символы и т. д.

На первый взгляд, лексемизация – тривиальная задача, и она вдоль и поперек изучена сообществом, занимающимся исследованиями и разработками в области обработки естественного языка. В главе 1 «*Социальные медиа, социальные данные и Python*» было дано краткое введение в эту область и упоминалось, как Twitter меняет правила лексемизации, позволяя включать в содержимое твита эмограммы, хештеги, ссылки на пользователей, URL-адреса, что очень отличает твиты от обычного естественного языка. Поэтому мы воспользовались библиотекой **NLTK** (Natural Language Toolkit), чтобы продемонстрировать класс `TweetTokenizer` как инструмент анализа содержимого, получаемого из Twitter. В данной главе мы снова воспользуемся этим инструментом.

Еще один этап предварительной обработки, достойный рассмотрения, называется **удалением стоп-слов**. Стоп-слова – это слова, сами по себе не несущие содержательной информации. К этой категории относятся артикли, предлоги, наречия и т. д. Частотный анализ показывает, что такие слова, как правило, являются наиболее распространенными в любом наборе данных. Список стоп-слов можно составить автоматически, анализируя данные (например, включая слова, встречающиеся в некоторой, произвольно выбранной, доле документов, например в 95%), но вообще желательно проявлять некоторый консерватизм и использовать только распространенные, в частности английские, стоп-слова. NLTK предлагает свой список распространенных английских стоп-слов в виде `nltk.corpus.stopwords`.

Удаление стоп-слов можно расширить включением в список символов (таких как знаки пунктуации) или характерных слов. В контексте Twitter распространенными стоп-словами в англоязычном сегменте являются термины **RT** (аббре-

виатура от ReTweet) и **via**, часто используемое для ссылки на автора материала, которым делятся (или который цитируют).

Наконец, еще одним важным этапом предварительной обработки является нормализация. Это обобщающий термин, объединяющий несколько видов обработки. Обычно нормализация используется, когда в том же самом блоке требуется подсчитать количество разных терминов. Особым случаем нормализации считается **нормализация регистра**, когда все термины переводятся в нижний регистр, что обеспечит совпадение строк с первоначально разным регистром (например, `'python' == 'Python'.lower()`). Преимущество нормализации регистра состоит в автоматическом подсчете частоты того или иного термина, и она не будет разбросана по разным вариациям одного и того же термина.

Выполним наш первый анализ частоты терминов при помощи следующего сценария:

```
# Chap02-03/twitter_term_frequency.py
import sys
import string
import json
from collections import Counter
from nltk.tokenize import TweetTokenizer
from nltk.corpus import stopwords

def process(text, tokenizer=TweetTokenizer(), stopwords=[]):
    """Обрабатывает текст твита:
    - Переводит буквы в нижний регистр
    - Выделяет лексемы
    - Удаляет стоп-слова
    - Удаляет цифры

    Возвращает: список строк
    """
    text = text.lower()
    tokens = tokenizer.tokenize(text)
    return [tok for tok in tokens if tok not in stopwords and not
            tok.isdigit()]

if __name__ == '__main__':
    fname = sys.argv[1]
    tweet_tokenizer = TweetTokenizer()
    punct = list(string.punctuation)
    stopword_list = stopwords.words('english') + punct + ['rt',
                                                          'via', '...']
    tf = Counter()
    with open(fname, 'r') as f:
    for line in f:
        tweet = json.loads(line)
        tokens = process(text=tweet['text'],
                         tokenizer=tweet_tokenizer,
                         stopwords=stopword_list)
        tf.update(tokens)
    for tag, count in tf.most_common(20):
        print("{}: {}".format(tag, count))
```

Основная логика предварительной обработки сосредоточена в функции `process()`. Она принимает строку и возвращает список строк. Все упомянутые ранее шаги: нормализация регистра, лексемизация и удаление стоп-слов – реализованы здесь.

Эта функция принимает еще два дополнительных аргумента: лексемизатор, то есть объект, реализующий метод `tokenize()`, и список стоп-слов, благодаря чему есть возможность адаптировать процесс удаления стоп-слов. Вместе со стоп-словами функция удаляет также числовые лексемы (например, `'5'` или `'42'`), применяя функцию `isdigit()`.

Приведенный сценарий принимает аргумент командной строки с именем анализируемого файла `.jsonl`. Он инициализирует объект `TweetTokenizer`, используемый для лексемизации, и затем определяет список стоп-слов. Такой список состоит из общеупотребительных английских стоп-слов из библиотеки NLTK, а также знаков пунктуации, определенных в `string.punctuation`. В список стоп-слов мы также включили лексемы `'rt'`, `'via'` и `'…'` (одиночный символ Юникода горизонтального троеточия).

Его можно запустить следующей командой:

```
$ python twitter_term_frequency.py user_timeline_PacktPub.jsonl
```

Результат работы сценария приведен ниже:

```
…: 1155
free: 637
ebook: 387
@packtpub: 308
get: 268
learning: 194
today: 188
learn: 162
ebooks: 160
we'll: 160
hi: 150
#webdev: 147
book: 147
we're: 140
new: 139
using: 130
#python: 122
#javascript: 121
#freelearning: 110
build: 109
know: 107
guide: 105
today's: 104
web: 104
code: 100
#machinelearning: 98
use: 96
https://t.co/ocxvjqsmhw: 94
tech: 88
need: 86
': 85
#gamedev: 83
```

Как видите, результат содержит смесь слов, хештегов, ссылок на пользователей, URL-адресов и символов Юникода, не входящих в список `string.punctuation`. Чтобы избавиться от этих дополнительных символов в выводе, можно просто добавить их в список стоп-слов. Что касается хештегов и ссылок на пользователей – мы получили именно то, чего ожидали от объекта `TweetTokenizer`, так как все лексемы допустимы. Возможно, вы не ожидали присутствия таких лексем, как `we're` и `we'll`, потому что это сокращенные формы двух отдельных лексем, а не отдельные самостоятельные лексемы. Если сокращенную форму расширить, в обоих случаях (*we are* и *we will*) мы будем иметь последовательность стоп-слов, потому что эти сокращенные формы обычно состоят из местоимений и общеупотребительных глаголов. Полный список английских сокращений приводится, например, в Википедии (https://en.wikipedia.org/wiki/Wikipedia%3aList_of_English_contractions).

Справиться с этим аспектом английского языка можно путем нормализации сокращений в расширенную форму. Например, следующая функция принимает список лексем и возвращает список нормализованных лексем (в действительности это не простая функция, а функция-генератор, о чем сообщает ключевое слово `yield`):

```
def normalize_contractions(tokens):
    token_map = {
      "i'm": "i am",
      "you're": "you are",
      "it's": "it is",
      "we're": "we are",
      "we'll": "we will",
    }
    for tok in tokens:
        if tok in token_map.keys():
            for item in token_map[tok].split():
                yield item
        else:
            yield tok
```

Взгляните на следующий пример сеанса в интерактивной оболочке Python:

```
>>> tokens = ["we're", "trying", "something"]
>>> list(normalize_contractions(tokens))
['we', 'are', 'trying', 'something']
```

Основная проблема данного подхода состоит в необходимости вручную определять все встречающиеся сокращения. Хотя число таких сокращений ограничено, их перенос в словарь может оказаться утомительным занятием. Кроме того, имеется несколько неоднозначных случаев, с которыми нельзя легко справиться, например случай *we'll*, с которым мы столкнулись выше: этому сокращению могут соответствовать *we will* и *we shall,* и таких случаев не один и не два.

Другое решение – рассматривать эти лексемы как стоп-слова, потому что после нормализации все их компоненты фактически являются стоп-словами.

Выбор решения во многом зависит от логики приложения, поэтому важно учитывать, что будет происходить после формализации этапа предварительной обработки в функции `process()`. Пока мы можем оставить предварительную обработку такой, какой она есть, без конкретного решения проблемы сокращений.

Ключевое слово `yield` и функции-генераторы. Функция `normalize_contractions()` использует ключевое слово `yield` вместо `return`. Данное ключевое слово превращает простую функцию в функцию-генератор, то есть итератор, позволяющий выполнить обход только один раз, так как он не хранит свои элементы в памяти, а генерирует их на лету. Одно из преимуществ функций-генераторов – уменьшенная потребность в оперативной памяти, поэтому данный прием рекомендуется использовать для итераций на больших объектах. Это ключевое слово позволяет сгенерировать множество возвращаемых значений в рамках одного вызова функции, тогда как ключевое слово `return` прекратит вычисления, как только будет вызвано (например, цикл `for` в функции `normalize_contractions()` выбрал бы только первую лексему).

В завершение раздела рассмотрим анализ частот терминов и сущностей с другой точки зрения.

Сценарии `twitter_hashtag_frequency.py` и `twitter_term_frequency.py` можно расширить, добавив построение простого графика вместо вывода частот наиболее распространенных терминов:

```
# Chap02-03/twitter_term_frequency_graph.py
y = [count for tag, count in tf.most_common(30)]
x = range(1, len(y)+1)

plt.bar(x, y)
plt.title("Частоты терминов")
plt.ylabel("Частота")
plt.savefig('term_distribution.png')
```

Этот фрагмент показывает, как строится график частот терминов из `twitter_term_frequency.py`; он также легко применяется для частот хештегов. После выполнения сценария на твитах от `@PacktPub` будет создан файл `term_distribution.png` с графиком, изображенным на рис. 2.2:

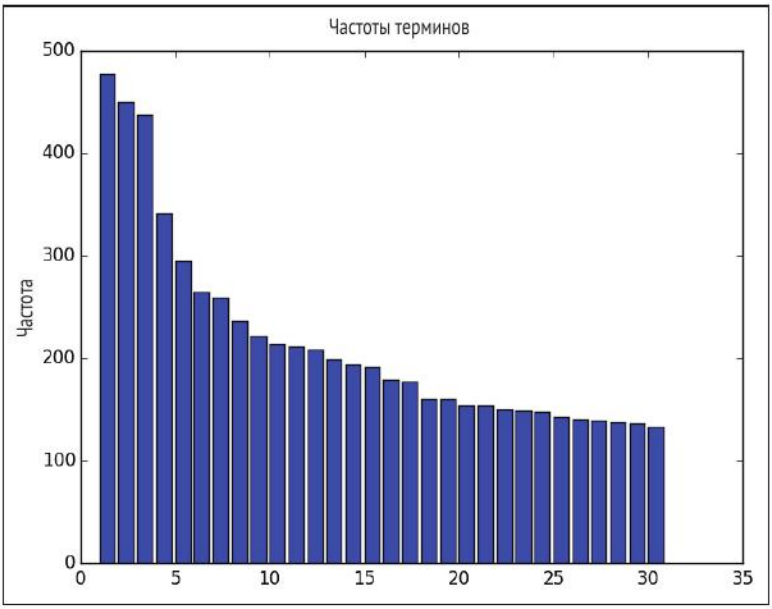

Рис. 2.2 ❖ Частота терминов в недавних твитах от `@PacktPub`

График на рис. 2.2 не показывает самих терминов, так как главной задачей этого последнего анализа является отображение частотного распределения. Слева на рис. 2.2 присутствует несколько терминов с очень высокой частотой, например самый частый термин встречается в несколько раз чаще, чем термины после отметки 10 или около того. По мере движения в правую сторону рисунка кривая становится менее крутой, означая, что термины справа имеют похожие частоты.

Если повторно выполнить этот же сценарий, заменив вызов `most_common(30)` на `most_common(1000)` (то есть 1000 самых частых терминов), это явление можно наблюдать еще четче, как показано на рис. 2.3:

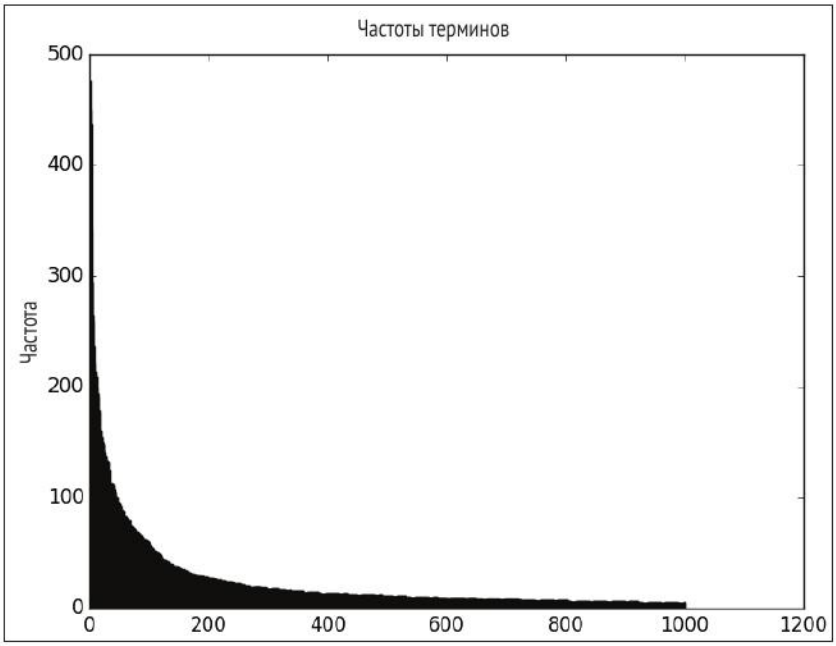

Рис. 2.3 ❖ Частоты 1000 наиболее распространенных терминов в твитах от `@PacktPub`

Кривая на рис. 2.3 напоминает аппроксимацию **степенной зависимости** (https://en.wikipedia.org/wiki/Power_law). В статистике степенная зависимость описывает функциональную связь между двумя количественными показателями; в данном случае частоты термина и его положения в ранжировке терминов по частоте. Этот тип распределения всегда имеет **вытянутый хвост** (https://en.wikipedia.org/wiki/Long_tail), означающий, что небольшая доля частых элементов доминирует в распределении, при условии что имеется большое количество элементов с меньшими частотами. Это явление имеет и другое название – **правило 80-20**, или **принцип Парето** (https://ru.wikipedia.org/wiki/Закон_Парето), согласно которому примерно 80% эффекта вызываются 20% причин (в нашем контексте на 20% уникальных терминов приходятся 80% всех употреблений).

Несколько десятилетий назад американский лингвист Джордж Ципф (George Zipf) популяризировал то, что в наше время стало известно как **закон Ципфа**

(https://ru.wikipedia.org/wiki/Закон_Ципфа). Согласно этому эмпирическому закону, частота любого слова в наборе документов обратно пропорциональна его рангу в частотной таблице. Другими словами, самое частое слово будет встречаться вдвое чаще, чем второе по частоте, в три раза чаще, чем третье по частоте, и т. д. На практике этот закон описывает не точные частоты, а тенденцию. Интересно отметить, что закон Ципфа можно обобщить для разных естественных языков, а также многих не связанных с языком классификаций и исследований в общественных науках[1].

Анализ твитов – временные ряды

Предыдущие разделы были посвящены анализу содержимого твитов. В этом разделе мы обсудим еще один интересный аспект – распределение твитов во времени.

Вообще говоря, временной ряд – это последовательность точек данных, которая состоит из наблюдений за тот или иной интервал времени. Поскольку твиты имеют поле `created_at` с точным временем создания, мы можем перераспределить твиты по временным дискретным интервалам, чтобы выяснить, как пользователи реагируют на те или иные события. В частности, интересно было бы понаблюдать за тем, как пишет твиты не просто отдельный пользователь, а вся популяция. Поэтому для такого анализа лучше всего подходят данные, собираемые через потоковый программный интерфейс Streaming API.

Анализ в настоящем разделе основан на наборе данных, полученных во время финала чемпионата мира по регби 2015. Этот пример наглядно демонстрирует, как пользователи реагируют на такие события, как спортивные соревнования, концерты, национальные выборы и все остальные – от крупных катастроф до ТВ-сериалов. Другим приложением анализа временных рядов данных Twitter является управление репутацией. Компанию может заинтересовать мониторинг мнений пользователей (возможно, клиентов) о ней. И динамическая природа Twitter идеально подходит для выяснения реакции, например, на новости о выпуске новых продуктов.

Воспользуемся возможностями библиотеки pandas по управлению временными рядами и снова применим библиотеку matplotlib для визуализации результатов:

```
# Chap02-03/twitter_time_series.py
import sys
import json
from datetime import datetime
import matplotlib.pyplot as plt
```

[1] К примеру, в соответствии с законом Ципфа городское расселение в большинстве городов РФ находится выше идеальной кривой Ципфа, поэтому ожидаемая тенденция – сокращение численности и людности средних и малых городов за счет миграции в крупные города; 7 городов-миллионников (Санкт-Петербург, Новосибирск, Екатеринбург, Нижний Новгород, Казань, Челябинск, Омск), находящихся ниже идеальной кривой Ципфа, имеют существенный резерв роста населения и ожидают его прироста; существуют риски депопуляции первого города в ранге (Москвы), поскольку второй город и последующие крупные города сильно отстают от идеальной кривой Ципфа в связи со снижением спроса на рабочую силу при одновременном росте стоимости проживания. – *Прим. перев.*

```
import matplotlib.dates as mdates
import pandas as pd
import numpy as np
import pickle

if __name__ == '__main__':
    fname = sys.argv[1]

    with open(fname, 'r') as f:
        all_dates = []
        for line in f:
            tweet = json.loads(line)
            all_dates.append(tweet.get('created_at'))
        idx = pd.DatetimeIndex(all_dates)
        ones = np.ones(len(all_dates))

        # фактический ряд (пока состоящий из единиц)
        my_series = pd.Series(ones, index=idx)

        # Передискретизация в 1-минутные дискретные интервалы
        per_minute = my_series.resample('1Min').sum().fillna(0)

        # Построение графика временного ряда
        fig, ax = plt.subplots()
        ax.grid(True)
        ax.set_title("Частоты твитов")
        hours = mdates.MinuteLocator(interval=20)
        date_formatter = mdates.DateFormatter('%H:%M')

        datemin = datetime(2015, 10, 31, 15, 0)
        datemax = datetime(2015, 10, 31, 18, 0)

        ax.xaxis.set_major_locator(hours)
        ax.xaxis.set_major_formatter(date_formatter)
        ax.set_xlim(datemin, datemax)
        max_freq = per_minute.max()
        ax.set_ylim(0, max_freq)
        ax.plot(per_minute.index, per_minute)

        plt.savefig('tweet_time_series.png')
```

Этот сценарий можно запустить командой:

```
$ python twitter_time_series.py stream__RWC2015__RWCFinal_Rugby.jsonl
```

Он читает файл `.jsonl`, переданный в аргументе командной строки, и загружает твиты построчно, как обсуждалось выше. Поскольку нас интересует только время публикации, мы создаем список `all_dates`, содержащий лишь значения атрибутов `created_at`.

Временные ряды индексируются временем создания, представленным переменной `idx` как `pd.DatetimeIndex` ранее собранных дат. Время определяется с точностью до секунды. Затем вызовом `ones()` создается односекундный массив np.array, который будет использован для вычисления частот.

После создания временного ряда выполняется передискретизация, то есть группировка членов ряда в дискретные интервалы. Эта операция изменяет подход к индексации временного ряда. В данном случае твиты группируются в 1-ми-

нутные интервалы. Поскольку мы указали, что передискретизация выполняется с операцией суммирования `sum`, каждый интервал будет содержать количество твитов, опубликованных в эту конкретную минуту. На случай, если в конкретном интервале твиты отсутствуют, добавлен вызов `fillna(0)`; в этом случае интервал не будет отброшен и получит значение 0. Такое вряд ли произойдет, учитывая размер нашего набора данных, но вполне возможно для данных меньшего объема.

Чтобы понять, что происходит во время передискретизации, рассмотрим следующий ниже пример, полученный в результате вызова `my_series.head()` и `per_minute.head()`:

```
# Перед передискретизацией
# my_series.head()
2015-10-31 15:11:40    1.0
2015-10-31 15:11:42    1.0
2015-10-31 15:11:38    1.0
2015-10-31 15:11:41    1.0
2015-10-31 15:11:44    1.0
dtype: float64
# После передискретизации
# per_minute.head()
2015-10-31 15:11:00    167.0
2015-10-31 15:12:00    380.0
2015-10-31 15:13:00    294.0
2015-10-31 15:14:00    289.0
2015-10-31 15:15:00    390.0
Freq: T, dtype: float64
```

Первый столбец содержит индекс ряда, то есть объект `datetime` (с точностью до секунд), а второй – значение (частоты), связанное с этим индексом.

Поскольку каждую секунду поступало по нескольку твитов, исходный ряд `my_series` показывает один тот же индекс (в примере `15:11:32`) несколько раз подряд. После передискретизации ряда точность индекса ряда снижается до минуты.

Затем сценарий использует библиотеку matplotlib, чтобы создать график временного ряда. Библиотека matplotlib в целом требует писать больше кода, чем другие библиотеки анализа данных, но это совсем несложно. Интересно отметить, как используются `MinuteLocator` и `DateFormatter` для расстановки меток на оси *X*. В данном случае используется интервал в 20 минут.

График сохраняется в файл `tweet_time_series.png`, который можно увидеть на рис. 2.4.

Время на рис. 2.4 соответствует GMT, то есть это британское время. Старт финала был намечен на 16:00. Как видно, по мере приближения времени старта наблюдается рост активности пользователей с пиком сразу после начала матча. Несколько пиков с превышением 2000 твитов в минуту — актуальные моменты матча, когда Новая Зеландия выходит далеко вперед и Австралия сокращает разрыв (это был интересный матч даже для неболельщиков команд, что уж говорить об остальных). Заключительный резкий скачок соответствует концу матча, церемонии награждения и торжествам, которые продолжались еще некоторое время (несмотря на то что поток был остановлен примерно в 18:00).

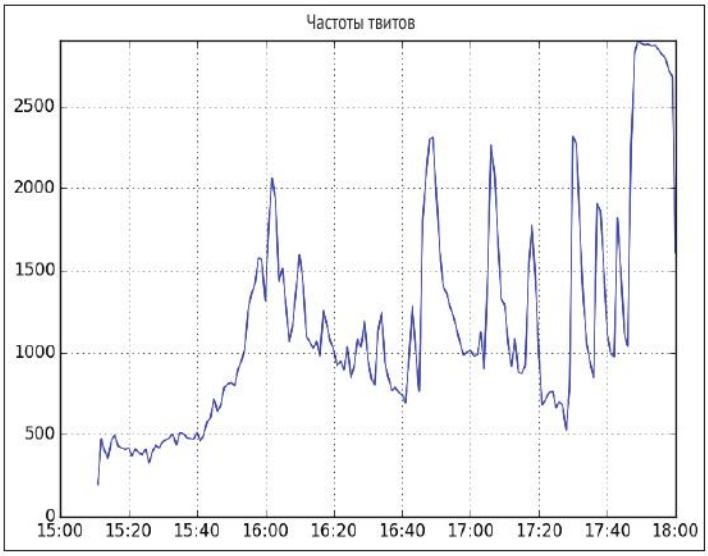

Рис. 2.4 ❖ Частота твитов во время финала чемпионата мира по регби 2015

Резюме

В настоящей главе было представлено несколько приложений анализа данных из Twitter. Мы обсудили вопрос регистрации приложения в Twitter, чтобы получить учетные данные и возможность взаимодействовать с программным интерфейсом Twitter API. Мы рассмотрели различные способы скачивания твитов, в частности используя конечные точки REST для поиска опубликованных твитов и Streaming API для сбора твитов с течением времени.

Рассматривая анатомию твита, мы обнаружили, что твит содержит намного больше 140 символов. По сути, это сложный объект с большим количеством информации.

Отправной точкой стало обсуждение частотного анализа на основе сущностей в твите. Основное внимание мы уделили хештегам – одной из особенностей Twitter, которая применяется пользователями для отслеживания отдельно взятых тем. Мы также обсудили аспекты **обработки естественного языка**, такие как лексемизация и нормализация лексем. Как мы убедились, тексты сообщений в Twitter не следуют правилам стандартного английского языка и имеют такие характерные особенности, как хештеги, ссылки на пользователей, URL-адреса, эмограммы и т. д. С другой стороны, мы обнаружили, что сообщения подчиняются статистическому правилу, именуемому законом Ципфа, как любой другой естестенно-языковой корпус значительного размера.

В завершение мы обратились к временному ряду. Чрезвычайно интересно наблюдать, как пользователи реагируют на события в реальном времени, и анализ временных рядов может оказаться мощным инструментом анализа больших наборов данных.

Следующая глава тоже будет полностью посвящена Twitter, но сосредоточит внимание на пользователях, подключенных к этой социальной сети.

Глава 3

Пользователи, читатели и сообщества в Twitter

Эта глава продолжает обсуждение анализа данных из Twitter. Сделав в предыдущей главе упор на анализе твитов, теперь переключимся на пользователей, их связи и взаимодействия.

Эта глава охватывает следующие темы:

- как загрузить список друзей и читателей для заданного пользователя;
- как анализировать связи между пользователями, общими друзьями и т. д.;
- как измерить степень влияния и вовлеченности в Twitter;
- алгоритмы кластеризации и кластеризация пользователей при помощи библиотеки **scikit-learn**;
- сетевой анализ и его применение для анализа диалогов в Twitter;
- как создавать динамические географические карты для визуализации географии твитов.

Пользователи, друзья и читатели

Одно из главных отличий Twitter от других популярных социальных сетей заключается в способах установления связей между пользователями. Связи в Twitter, по сути, не обязательно двунаправленны. Пользователь может принять решение подписаться на твиты других пользователей, став их читателем (последователем, фолловером), но этот акт может не быть взаимным. Этим Twitter сильно отличается от других социальных сетей, таких как Facebook и LinkedIn, где связи должны подтверждаться обеими сторонами.

Используя терминологию Twitter, два направления связи (люди, которых читаю я, и люди, которые читают меня) имеют разные имена. Люди, которых читаю я, называются *друзьями*, тогда как люди, которые читают меня, называются *читателями*. Когда связь двунаправленная, такой пользователь обычно описывается как *общий друг*.

Возвращаясь к интерфейсу Twitter API

Программный интерфейс Twitter API предоставляет несколько конечных точек для получения информации о читателях, друзьях и профилях пользователей в целом. Выбор конкретной конечной точки диктуется решаемой задачей.

Беглый взгляд на документацию (https://dev.twitter.com/rest/public) позволяет выделить несколько самых интересных из них.

В отношении профиля единственного пользователя наибольший интерес представляет конечная точка `users/show`: по заданному экранному имени или идентификатору пользователя она позволяет получить полный профиль. Эта возможность сильно ограничивается частотой запросов – можно делать всего 180 запросов каждые 15 минут, то есть получить лишь 180 профилей пользователей в течение 15-минутного окна. Учитывая такое ограничение, данная конечная точка должна использоваться, только если нужно получить определенный профиль, она не подходит для массовой загрузки профилей.

Список читателей того или иного пользователя можно получить, используя конечную точку `followers/list` (https://dev.twitter.com/rest/reference/get/followers/list), доступ к которой в библиотеке Tweepy реализует функция `API.followers()`. Аналогично конечная точка `friends/list` позволяет получить список друзей пользователя. Доступ к ней в Tweepy реализует функция `API.friends()`. Основная проблема этих конечных точек заключается в строгом ограничении на частоту запросов: всего 15 запросов в 15-минутном окне, при этом каждый запрос может возвращать до 20 профилей, то есть каждые 15 минут можно получать до 300 профилей.

Этот подход может пригодиться для анализа профилей с очень небольшим количеством читателей и друзей, вместе с тем профили с тысячами читателей вовсе не редкость (если говорить об известных людях, это число может доходить до нескольких миллионов).

Для обхода этого ограничения можно использовать сочетание других конечных точек, более пригодных для извлечения больших пакетов данных. Конечная точка `followers/ids` (https://dev.twitter.com/rest/reference/get/followers/ids) может возвращать группы по 5000 идентификаторов пользователей в ответ на запрос. Несмотря на то что она тоже ограничена 15 запросами в 15-минутном окне, легко посчитать, насколько будет больше итоговое число идентификаторов, которые можно получить (75 000 за четверть часа).

Конечная точка followers/ids возвращает только список идентификаторов пользователей, читающих заданного пользователя, но не позволяет получить полные профили. Однако эти идентификаторы можно использовать для обращения к конечной точке `users/lookup` (https://dev.twitter.com/rest/reference/get/users/lookup), которая принимает до 100 идентификаторов и возвращает список полных профилей. Частота обращений к конечной точке users/lookup ограничена 180 запросами каждые 15 минут, что соответствует 18 000 профилям за четверть часа. Это число эффективно ограничивает наши возможности.

Для загрузки пакета профилей друзей можно использовать похожее обходное решение, основанное на объединении конечных точек `friends/ids` и вышеупомянутой `users/lookup`. С точки зрения частоты запросов, загружая идентификаторы друзей, мы столкнемся с теми же ограничениями, что и для читателей. При этом следует избегать одной небольшой ошибки, когда два процесса загрузки данных рассматриваются как абсолютно независимые; имейте в виду, что узким местом является конечная точка `users/lookup`. Для сценария, который загружает профили друзей и читателей, нужно учитывать, что оба процесса загрузки данных будут обращаться к конечной точке `users/lookup`, поэтому общее количество запросов к `users/lookup` будет представлено суммой друзей и читателей. Для пользователей, которые одновременно являются друзьями и читателями, требуется выполнить поиск только один раз.

Структура профиля пользователя

Прежде чем углубиться в подробности процесса загрузки большого количества профилей читателей и друзей, рассмотрим случай с отдельным пользователем и исследуем структуру профиля. Воспользуемся для этого конечной точкой `users/show`. Так как это разовый пример, мы не превысим ограничения на частоту запросов (массовую загрузку мы рассмотрим в следующем разделе).

Доступ к конечной точке в Tweepy реализует функция `API.get_user()`. Для аутентификации можно повторно использовать код, созданный в главе 2 «*Твиттер – хештеги, темы и временные ряды*» (не забудьте настроить переменные окружения). Выполните в интерактивной оболочке следующие команды:

```
>>> from twitter_client import get_twitter_client
>>> import json
>>> client = get_twitter_client()
>>> profile = client.get_user(screen_name="PacktPub")
>>> print(json.dumps(profile._json, indent=4))
```

Здесь все довольно просто: после аутентификации выполняется одиночный вызов API для загрузки профиля. Функция `get_user()` возвращает экземпляр класса `tweepy.models.User`, уже упоминавшегося в главе 2 «*Твиттер – хештеги, темы и временные ряды*». Это обертка для хранения различных атрибутов пользователя. Оригинал ответа Twitter API в формате JSON хранится в форме словаря Python в атрибуте `_json`, который мы выводим на экран вызовом `json.dumps()` с аргументом `indent` для структурной распечатки.

В результате на экране должен появиться следующий документ JSON (для краткости некоторые атрибуты были опущены):

```
{
  "screen_name": "PacktPub",
  "name": "Packt Publishing",
  "location": "Birmingham, UK",
  "id": 17778401,
  "id_str": "17778401",
  "description": "Providing books, eBooks, video tutorials, and
    articles for IT developers, administrators, and users.",
  "followers_count": 21652,
  "friends_count": 3583,
  "follow_request_sent": false,
  "status": { ... },
  "favourites_count": 1353,
  "protected": false,
  "verified": false,
  "statuses_count": 19440,
  "lang": "en",
  "entities": {
    "description": {
    "urls": []
    },
    "url": {
      "urls": [
        {
```

```
          "indices": [
            0,
            22
          ],
          "display_url": "PacktPub.com",
          "expanded_url": "http://www.PacktPub.com",
          "url": "http://t.co/vEPCgOu235"
        }
      ]
    }
  },
  "following": true,
  "geo_enabled": true,
  "time_zone": "London",
  "utc_offset": 0,
}
```


Следующая таблица перечисляет все атрибуты и дает краткое их описание:

Название атрибута	Описание
_json	Словарь с ответом в формате JSON, содержащим профиль пользователя
created_at	Время и дата в UTC создания учетной записи пользователя
contributors_enabled	Признак включения **режима соавтора** (редко True)
default_profile	Признак, что пользователь не изменял тему профиля
descrition	Строка с описанием профиля пользователя
default_profile_image	Признак отсутствия собственной фотографии для профиля
entities	Список сущностей в URL-адресе или в описании
followers_count	Число читателей
follow_request_sent	Признак отправки запроса на чтение
favourites_count	Число твитов, понравившихся пользователю
following	Признак, показывающий, читает ли аутентифицированный пользователь
friends_count	Число друзей
geo_enabled	Признак разрешения **геопривязки** пользователя
id	Уникальный идентификатор пользователя в виде большого целого числа
id_str	Уникальный идентификатор пользователя в виде строки
is_translator	Признак вхождения пользователя в сообщество переводчиков Twitter
lang	Код языка пользователя
listed_count	Число сообществ, в которых участвует пользователь
location	Заявленное пользователями местоположение в виде строки
name	Имя пользователя
profile_*	Массив информации, связанной с профилем (см. список атрибутов ниже)
protected	Признак, показывающий, что пользователь защитил свои твиты
status	Внедренный объект с последним твитом (описания полей можно найти в главе 2 «*Твиттер – хештеги, темы и временные ряды*»)
screen_name	Экранное имя пользователя, то есть дескриптор Twitter
statuses_count	Число твитов
time_zone	Строка с названием часового пояса, заявленного пользователем
utc_offset	Смещение относительно GMT/UTC в секундах
URL	URL-адрес, предоставленный пользователем, для связи с профилем
verified	Признак, показывающий, проверен ли пользователь

Далее приводится список атрибутов, связанных с профилем:

- profile_background_color: шестнадцатеричный код цвета фона, выбранный пользователем;
- profile_background_tile: признак, что при выводе на экран изображение profile_background_image_url должно размещаться плитками;
- profile_link_color: шестнадцатеричный код цвета для гиперссылок, выбранный пользователем;
- profile_use_background_image: признак, что для оформления фона пользователь желает использовать свое изображение;
- profile_background_image_url: URL-адрес (на основе протокола HTTP) фонового изображения;
- profile_background_image_url_https: то же, что и предыдущий атрибут, но на основе протокола HTTPS;
- profile_text_color: шестнадцатеричный код цвета, выбранный пользователем для вывода текста;
- profile_banner_url: URL-адрес (на основе протокола HTTPS) баннера профиля пользователя;
- profile_sidebar_fill_color: шестнадцатеричный код цвета фона, выбранный пользователем для боковых панелей;
- profile_image_url: URL-адрес (на основе протокола HTTP) фотографии (аватара) профиля пользователя;
- profile_image_url_https: то же самое, но на основе HTTPS;
- profile_sidebar_border_color: шестнадцатеричный код цвета, выбранный пользователем для границ боковых панелей.

Следующий раздел посвящен загрузке профилей друзей и читателей пользователя.

Загрузка профилей друзей и читателей

Опираясь на обсуждение конечных точек в предыдущем разделе, напишем сценарий, который принимает имя пользователя (экранное имя) и загружает его полный профиль, список читателей (с полными профилями) и список друзей (тоже с полными профилями):

```
# Chap02-03/twitter_get_user.py
import os
import sys
import json
import time
import math
from tweepy import Cursor
from twitter_client import get_twitter_client

MAX_FRIENDS = 15000

def usage():
    print("Применение:")
    print("python {} <имя_пользователя>".format(sys.argv[0]))

def paginate(items, n):
    """На основе списка items генерирует фрагменты по n элементов"""
```

```
    for i in range(0, len(items), n):
        yield items[i:i+n]

if __name__ == '__main__':
    if len(sys.argv) != 2:
        usage()
        sys.exit(1)
    screen_name = sys.argv[1]
    client = get_twitter_client()
    dirname = "users/{}".format(screen_name)
    max_pages = math.ceil(MAX_FRIENDS / 5000)
    try:
        os.makedirs(dirname, mode=0o755, exist_ok=True)
    except OSError:
        print("Каталог {} уже существует".format(dirname))
    except Exception as e:
        print("Ошибка при создании каталога {}".format(dirname))
        print(e)
        sys.exit(1)

    # получить читателей заданного пользователя
    fname = "users/{}/followers.jsonl".format(screen_name)
    with open(fname, 'w') as f:
        for followers in Cursor(client.followers_ids,
                                screen_name=screen_name).pages(max_pages):
            for chunk in paginate(followers, 100):
                users = client.lookup_users(user_ids=chunk)
                for user in users:
                    f.write(json.dumps(user._json)+"\n")
            if len(followers) == 5000:
                print("Есть еще результаты. Приостановка на 60 сек., чтобы
                       не превысить ограничения на частоту запросов")
                time.sleep(60)

    # получить друзей заданного пользователя
    fname = "users/{}/friends.jsonl".format(screen_name)
    with open(fname, 'w') as f:
        for friends in Cursor(client.friends_ids,
                              screen_name=screen_name).pages(max_pages):
            for chunk in paginate(friends, 100):
                users = client.lookup_users(user_ids=chunk)
                for user in users:
                    f.write(json.dumps(user._json)+"\n")
            if len(friends) == 5000:
                print("Есть еще результаты. Приостановка на 60 сек., чтобы
                       не превысить ограничения на частоту запросов")
                time.sleep(60)

    # получить профиль пользователя
    fname = "users/{}/user_profile.json".format(screen_name)
    with open(fname, 'w') as f:
        profile = client.get_user(screen_name=screen_name)
        f.write(json.dumps(profile._json, indent=4))
```

Данный сценарий принимает один аргумент командной строки – экранное имя пользователя. Этот сценарий можно запустить следующей командой:

```
$ python twitter_get_user.py PacktPub
```

Поскольку на момент написания настоящей главы издательство PacktPub имело более 10 000 читателей, выполнение сценария займет более 2 минут из-за ограничения, накладываемого программным интерфейсом[1]. Этот сценарий использует `time.sleep()`, чтобы не превысить ограничения на частоту запросов, как описывалось в главе 2 «*Твиттер – хештеги, темы и временные ряды*». Число секунд, передаваемых функции `sleep()`, диктуется требованиями программного интерфейса Twitter.

Анализ связей

После загрузки информации о друзьях и читателях можно приступать к первичному анализу структуры сети, созданной этими связями. На рис. 3.1 изображен фиктивный пример небольшой сети, где связи между пользователями выделены с точки зрения первого лица (то есть с точки зрения пользователя, помеченного как Я, поэтому мы опишем рисунок в первом лице):

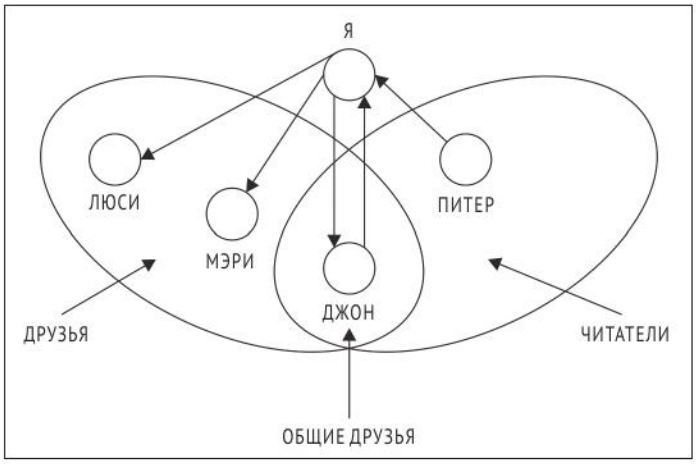

Рис. 3.1 ❖ Пример сети связей: друзья, читатели и общие друзья

В этом примере **Я** связан с четырьмя другими пользователями: **ПИТЕР** и **ДЖОН** меня читают (поэтому они отмечены как **ЧИТАТЕЛИ**). **Я** читаю **ЛЮСИ**, **МЭРИ** и **ДЖОНА** (поэтому они отмечены как **ДРУЗЬЯ**). **ДЖОН** принадлежит обеим группам: пересечение между **ДРУЗЬЯМИ** и **ЧИТАТЕЛЯМИ** описывается как **ОБЩИЕ ДРУЗЬЯ**.

Но из этого представления нельзя понять, связаны ли эти четыре пользователя между собой. В этом заключается суть данных, которые мы загрузили в преды-

[1] На момент перевода книги их было уже более 21 000, соответственно, для выполнения сценария потребуется вдвое больше времени. – *Прим. перев.*

дущем разделе: мы имеем информацию о друзьях и читателях конкретного профиля, но, чтобы выявить связи между ними, необходимо выполнить итеративный обход их профилей и загрузить соответствующие данные для них.

На основе загруженных данных мы получили базовые статистики – число читателей и друзей – и теперь можем ответить на следующие элементарные вопросы:

- кто мои общие друзья?
- кто не читает меня в ответ?
- кого я не читаю в ответ?

Следующий сценарий читает ранее загруженные файлы в формате JSONL и вычисляет статистики для получения ответов на эти вопросы:

```
# Chap02-03/twitter_followers_stats.py
import sys
import json

def usage():
    print("Применение:")
    print("python {} <имя_пользователя>".format(sys.argv[0]))

if __name__ == '__main__':
    if len(sys.argv) != 2:
        usage()
        sys.exit(1)
    screen_name = sys.argv[1]
    followers_file = 'users/{}/followers.jsonl'.format(screen_name)
    friends_file = 'users/{}/friends.jsonl'.format(screen_name)
    with open(followers_file) as f1, open(friends_file) as f2:
        followers = []
        friends = []
        for line in f1:
            profile = json.loads(line)
            followers.append(profile['screen_name'])
            for line in f2:
                profile = json.loads(line)
                friends.append(profile['screen_name'])
            mutual_friends = [user for user in friends
                              if user in followers]
            followers_not_following = [user for user in followers
                                       if user not in friends]
            friends_not_following = [user for user in friends
                                     if user not in followers]

            print("----- Статистика -----")
            print("{} имеет {} читателей".format(screen_name,
                  len(followers)))
            print("{} имеет {} друзей".format(screen_name, len(friends)))
            print("{} имеет {} общих друзей".format(screen_name,
                  len(mutual_friends)))
            print("{} друзей не читают {} в ответ"
                  .format(len(friends_not_following), screen_name))
            print("{} читателей не читаются {} в ответ"
                  .format(len(followers_not_following), screen_name))
```

Данный сценарий принимает аргумент командной строки с именем пользователя, поэтому его можно запустить, например, следующей командой:

```
$ python twitter_followers_stats.py PacktPub
```

Он выведет следующее:

```
----- Статистика -----
PacktPub имеет 14999 читателей
PacktPub имеет 1210 друзей
PacktPub имеет 91 общего друга
1119 друзей не читают PacktPub в ответ
14908 читателей не читаются PacktPub в ответ
```

Для обработки друзей и читателей используется самый распространенный тип данных – обычный список Python, `list()`, который заполняется именами пользователей (точнее, экранными именами, `screen_name`). Программа читает два файла JSONL и добавляет экранные имена в соответствующие списки. Затем для вычисления различных статистик применяются три генератора списков.

Генераторы списков в Python. Python поддерживает идею под названием «генератор списков» (list comprehension) – изящный способ преобразования одного списка (или любой итерируемой последовательности) в другой. В процессе преобразования элементы могут включаться в результат по условию и преобразовываться пользовательской функцией, например:

```
>>> numbers = [1, 2, 3, 4, 5]
>>> squares = [x*x for x in numbers]
>>> squares
[1, 4, 9, 16, 25]
```

Этот генератор списков действует аналогично следующему фрагменту программного кода:

```
>>> squares = []
>>> for x in numbers:
...     squares.append(x*x)
...
>>> squares
[1, 4, 9, 16, 25]
```

Самое интересное, что генераторы списков читаются как обычное предложение на английском языке, благодаря чему программный код становится чрезвычайно понятным. Генераторы не ограничиваются только списками, их также можно использовать для конструирования словарей.

В генераторах списков вместо функций `map` и `filter` используется математическая форма построения последовательности. Такая математическая запись в теории множеств и логике называется «определением интенсионала множества» и описывает условие, которое должно выполняться для всех его элементов. В сущности, более точным термином для данной операции является термин «описание интенсионала» соответственно списка, словаря и множества. Ниже приводятся примеры описания интенсионала, соответственно, списка, словаря, множества и создания объекта собственно генератора, то есть итерируемой последовательности, на Python:

```
[x*x for x in numbers]    # описание интенсионала списка
{x:x*x for x in numbers}  # описание интенсионала словаря
{x*x for x in numbers} == set(x*x for x in numbers) # множества
(x*x for x in numbers)    # итерируемой последовательности
```

Хотелось бы сделать два замечания к данному аспекту реализации. Во-первых, файлы JSONL будут содержать уникальные профили: каждый читатель (или друг) будет представлен только раз, поэтому повторяющиеся записи отсутствуют. Во-вторых, при выполнении статистических расчетов, приведенных выше, порядок следования элементов не важен. По сути, мы всего лишь выполняем операции со множествами (в данном примере это пересечение и разность).

Наш код можно перестроить и реализовать вычисление базовых статистических величин использованием `set()`; изменения в сценарии при этом коснутся загрузки данных и вычисления общих друзей, друзей, которые не читают в ответ, и читателей, которых не читают в ответ:

```
with open(followers_file) as f1, open(friends_file) as f2:
    followers = set()
    friends = set()
    for line in f1:
        profile = json.loads(line)
        followers.add(profile['screen_name'])
    for line in f2:
        profile = json.loads(line)
        friends.add(profile['screen_name'])
    mutual_friends = friends.intersection(followers)
    followers_not_following = followers.difference(friends)
    friends_not_following = friends.difference(followers)
```

Этот сценарий произведет тот же результат, что и предыдущий. Главное преимущество использования множеств вместо списков состоит в вычислительной сложности: такие операции, как включение в список (то есть проверка принадлежности элемента списку `item in list` или множеству `item in set`), выполняются для списков за линейное время и для множеств за постоянное время. Операция проверки принадлежности используется с целью получения значений для переменных `mutual_friends`, `followers_not_following` и `friends_not_following`, поэтому такое простое изменение значительно влияет на общее время выполнения сценария, и при наличии большего числа читателей/друзей эта разница будет еще очевиднее из-за линейной вычислительной сложности списков.

Вычислительная сложность. В предыдущем разделе использовались такие словосочетания, как «линейное время», «постоянное время» и «линейная сложность». Понятие вычислительной сложности в информатике имеет огромную важность, поскольку оценивает объем ресурсов, необходимых алгоритму. В этом примечании мы затронем временную сложность и занимаемое алгоритмом количество времени, которое описывается как функция от объема входных данных.

Когда алгоритм выполняется за линейное время, продолжительность обработки больших входных данных увеличивается линейно с размером данных. Для описания такого класса алгоритмов используется математическое обозначение (так называемое «О» большое) `O(n)`, где `n` – размер входных данных.

Для алгоритмов, выполняющихся за постоянное время, напротив, объем входных данных не влияет на время работы. В этом случае используется обозначение `O(1)`. В целом понимание сложности различных операций и структур данных является ключевым для разработки нетривиальных программ, поскольку может оказать огромное влияние на производительность системы.

В этом месте читателю может быть любопытно узнать об использовании таких библиотек, как NumPy. Как отмечалось в главе 1 «*Социальные медиа, социальные данные и Python*», библиотека NumPy позволяет быстро и эффективно обрабатывать массивоподобные структуры данных и обеспечивает значительное увеличение производительности на простых списках. Несмотря на оптимизацию, использование NumPy в этом конкретном случае не приведет к особенной выгоде с точки зрения производительности просто потому, что вычислительные затраты генератора списков не изменятся от замены одного типа списков другим. Тем не менее ниже приводится результат перестройки сценария с привлечением библиотеки NumPy:

```
with open(followers_file) as f1, open(friends_file) as f2:
    followers = []
    friends = []
    for line in f1:
        profile = json.loads(line)
        followers.append(profile['screen_name'])
    for line in f2:
        profile = json.loads(line)
        friends.append(profile['screen_name'])
    followers = np.array(followers)
    friends = np.array(friends)
    mutual_friends = np.intersect1d(friends,
                                    followers,
                                    assume_unique=True)
    followers_not_following = np.setdiff1d(followers,
                                           friends,
                                           assume_unique=True)
    friends_not_following = np.setdiff1d(friends,
                                         followers,
                                         assume_unique=True)
```

Этот фрагмент предполагает, что пакет NumPy импортирован с псевдонимом `np`, как обсуждалось в главе 1 «*Социальные медиа, социальные данные и Python*». Данная библиотека реализует несколько операций со множествами `intersect1d` и `setdiff1d`, но в их основе лежит все та же структура данных – массивоподобный объект (похожий на список). Третий аргумент в этих функциях, `assume_unique`, используется в предположении, что входные массивы имеют уникальные элементы, как в нашем случае. Это помогает ускорить вычисления, но итог остается прежним: при наличии сотен читателей/друзей операции со множествами будут выполняться быстрее. А разница в производительности для небольшого числа читателей/друзей заметна не будет.

В загружаемых примерах к данной книге – в файлах `twitter_followers_stats.py`, `twitter_followers_stats_set.py` и `twitter_followers_stats_numpy.py` – представлены три разные реализации. В эти файлы включен также хронометраж, что дает возможность оценить производительность на разных наборах данных.

В завершение этого отступления с описанием переделки программного кода отметим, что основная цель – выбрать структуру данных, которая лучше всего подходит для выполняемых операций.

Измерение степени влияния и вовлеченности

Один из наиболее часто упоминаемых персонажей на арене социальных медиа – мифическое *влиятельное лицо*, иногда именуемое просто авторитетом (Influencer). Данная фигура ответственна за сдвиг парадигмы в последних маркетинговых стратегиях (https://en.wikipedia.org/wiki/Influencer_marketing), концентрирующихся на ориентации ключевых индивидов, а не рынка в целом.

Влиятельными обычно считаются пользователи, активные в пределах своего сообщества; в Twitter влиятельное лицо много пишет о темах, которые его беспокоят. Влиятельные лица имеют широкие связи, так как они читают многих других пользователей и их читают многие другие пользователи, которые также участвуют в сообществе. В целом к влиятельному лицу относятся как к эксперту в своей области, и другие пользователи обычно доверяют его мнению.

Этого описания достаточно, чтобы объяснить, почему влиятельные лица составляют важную часть недавних тенденций в маркетинге, – влиятельное лицо может нести полезную информацию или даже стать пропагандистом определенного продукта или бренда среди широкого круга своих сторонников.

Не важно, где сконцентрирован ваш главный интерес: в программировании на Python или в дегустации вин, и насколько огромна (или мала) ваша социальная сеть, вы, вероятно, уже знаете, кто является влиятельным лицом в ваших социальных кругах: друг, знакомый или случайный незнакомец(ка) в вебе, мнению которого вы доверяете и цените в силу его экспертных знаний по данной теме.

Другим, но в некотором роде родственным понятием является *степень вовлеченности*. Степень вовлеченности пользователя, или клиента, – это оценка реакции на конкретное предложение, например продукта или услуги. В контексте социальных медиа элементы информации часто создаются с целью направить веб-трафик к веб-сайту компании или электронному магазину. Измерение степени вовлеченности играет важную роль, потому что помогает в определении и понимании стратегий максимизации взаимодействия с вашей сетью и в конечном счете увеличивает деловую активность. В Twitter пользователи вовлекаются посредством ретвитов конкретного твита или выставления отметки «Мне нравится», что в ответ приводит к большей заметности твита.

В данном разделе мы обсудим некоторые интересные аспекты анализа социальных медиа относительно возможности измерения размеров влияния и степени вовлеченности. В Twitter естественно связывать влияние с числом пользователей в отдельно взятой сети. Большое число читателей, как нетрудно догадаться, говорит о том, что пользователь может добраться до большего количества людей, но оно не говорит о том, как воспринимаются его твиты.

Следующий ниже сценарий сравнивает статистику двух профилей пользователей:

```
# Chap02-03/twitter_influence.py
import sys
import json

def usage():
    print("Применение:")
    print("python {} <имя_пользователя1> <имя_пользователя2>"
          .format(sys.argv[0]))
```

```
if __name__ == '__main__':
    if len(sys.argv) != 3:
        usage()
        sys.exit(1)
    screen_name1 = sys.argv[1]
    screen_name2 = sys.argv[2]
```

После чтения двух экранных имен из аргументов командной строки он составляет список и определяет число читателей для каждого с целью вычислить число достижимых пользователей:

```
    followers_file1 = 'users/{}/followers.jsonl'.format(screen_name1)
    followers_file2 = 'users/{}/followers.jsonl'.format(screen_name2)
    with open(followers_file1) as f1, open(followers_file2) as f2:
        reach1 = []
        reach2 = []
        for line in f1:
            profile = json.loads(line)
            reach1.append((profile['screen_name'],
                           profile['followers_count']))
        for line in f2:
            profile = json.loads(line)
            reach2.append((profile['screen_name'],
                           profile['followers_count']))
```

Затем загружает профили этих двух пользователей и определяет несколько элементарных статистик (количество читателей и количество статусов, или твитов):

```
    profile_file1 = 'users/{}/user_profile.json'.format(screen_name1)
    profile_file2 = 'users/{}/user_profile.json'.format(screen_name2)
    with open(profile_file1) as f1, open(profile_file2) as f2:
        profile1 = json.load(f1)
        profile2 = json.load(f2)
        followers1 = profile1['followers_count']
        followers2 = profile2['followers_count']
        tweets1 = profile1['statuses_count']
        tweets2 = profile2['statuses_count']

    sum_reach1 = sum([x[1] for x in reach1])
    sum_reach2 = sum([x[1] for x in reach2])
    avg_followers1 = round(sum_reach1 / followers1, 2)
    avg_followers2 = round(sum_reach2 / followers2, 2)
```

Потом загружает ленты этих двух пользователей с целью выяснить число ретвитов и отметок «Мне нравится»:

```
    timeline_file1 = 'user_timeline_{}.jsonl'.format(screen_name1)
    timeline_file2 = 'user_timeline_{}.jsonl'.format(screen_name2)
    with open(timeline_file1) as f1, open(timeline_file2) as f2:
        favorite_count1, retweet_count1 = [], []
        favorite_count2, retweet_count2 = [], []
        for line in f1:
            tweet = json.loads(line)
            favorite_count1.append(tweet['favorite_count'])
            retweet_count1.append(tweet['retweet_count'])
```

```
    for line in f2:
        tweet = json.loads(line)
        favorite_count2.append(tweet['favorite_count'])
        retweet_count2.append(tweet['retweet_count'])
```

Далее по полученным значениям вычисляется среднее число отметок «Мне нравится» и ретвитов в абсолютом выражении и в расчете на число читателей:

```
avg_favorite1 = round(sum(favorite_count1) / tweets1, 2)
avg_favorite2 = round(sum(favorite_count2) / tweets2, 2)
avg_retweet1 = round(sum(retweet_count1) / tweets1, 2)
avg_retweet2 = round(sum(retweet_count2) / tweets2, 2)
favorite_per_user1 = round(sum(favorite_count1) / followers1, 2)
favorite_per_user2 = round(sum(favorite_count2) / followers2, 2)
retweet_per_user1 = round(sum(retweet_count1) / followers1, 2)
retweet_per_user2 = round(sum(retweet_count2) / followers2, 2)
print("----- Статистика {} -----".format(screen_name1))
print("{} читателей".format(followers1))
print("{} пользователей, достижимых за 1 шаг"
      .format(sum_reach1))
print("Среднее число читателей среди читателей {}: {}"
      .format(screen_name1, avg_followers1))
print("{} отметок 'Мне нравится' ({} на твит,
       {} на пользователя)"
      .format(sum(favorite_count1), avg_favorite1,
      favorite_per_user1))
print("{} ретвитов ({} в расчете на твит, {} в расчете на
      пользователя)".format(sum(retweet_count1), avg_retweet1,
      retweet_per_user1))
print("----- Статистика {} -----".format(screen_name2))
print("{} читателей".format(followers2))
print("{} пользователей, достижимых за 1 шаг"
      .format(sum_reach2))
print("Среднее число читателей среди читателей {}: {}"
      .format(screen_name2, avg_followers2))
print("{} отметок 'Мне нравится' ({} на твит,
       {} на пользователя)"
      .format(sum(favorite_count2), avg_favorite2,
      favorite_per_user2))
print("{} ретвитов ({} в расчете на твит, {} на
      пользователя)".format(sum(retweet_count2), avg_retweet2,
      retweet_per_user2))
```

Этот сценарий принимает два аргумента командной строки и предполагает, что данные уже были загружены. Для обоих пользователей нам нужны данные о читателях (загруженные при помощи `twitter_get_user.py`) и соответствующие им ленты пользователей (загруженные при помощи `twitter_get_user_timeline.py` из главы 2 «*Твиттер – хештеги, темы и временные ряды*»).

Сценарий получился довольно объемным, потому что выполняет одни и те же операции для двух профилей и выводит все результаты в терминал. Его можно разбить на несколько частей.

Сначала определяются читатели читателей. Это даст кое-какую информацию о части сети, непосредственно связанной с конкретным пользователем. Други-

ми словами, она дает ответ на вопрос: скольких пользователей сможет достичь данный человек, если все его читатели будут ретвитить его твиты? Для этого сценарий читает файл users/<user>/followers.jsonl и сохраняет список кортежей, где каждый кортеж представляет одного читателя и имеет форму (экранное_имя, количество_читателей). Экранное имя сохраняется на случай, если потом появится желание узнать, какие пользователи имеют самое большое количество читателей (в сценарии этот показатель не вычисляется, но его легко получить при помощи функции sorted()).

На втором этапе читается профиль пользователя из файла users/<user>/user_profile.json, чтобы получить информацию об общем количестве читателей и твитов. На основе собранных к настоящему времени данных можно вычислить общее количество пользователей, достижимых на расстоянии одного шага (читатель читателя), и среднее число читателей читателя. Эти величины вычисляют строки:

```
sum_reach1 = sum([x[1] for x in reach1])
avg_followers1 = round(sum_reach1 / followers1, 2)
```

Первая использует генератор списков для обхода кортежей, упомянутых выше, а вторая вычисляет простое среднее арифметическое с округлением до двух знаков после запятой.

Третья часть сценария читает ленту пользователя из файла user_timeline_<user>.jsonl, созданного в главе 2 «*Твиттер – хештеги, темы и временные ряды*», и собирает информацию о числе ретвитов и отметок «Мне нравится» для каждого твита. После этого можно вычислить количество ретвитов, отметок «Мне нравится» и среднее число ретвитов/отметок в расчете на твит и читателя.

Для примера я выполнил небольшой тщеславный анализ, где сравнил свою учетную запись @marcobonzanini с учетной записью издательсва Packt Publishing:

```
$ python twitter_influence.py marcobonzanini PacktPub
```

и получил следующий результат:

```
----- Статистика marcobonzanini -----
1123 читателя
5748125 пользователей, достижимых за 1 шаг
Среднее число читателей среди читателей marcobonzanini: 5118.54
1490 отметок 'Мне нравится' (2.11 на твит, 1.33 на пользователя)
4442 ретвита (6.3 на твит, 3.96 на пользователя)
----- Статистика PacktPub -----
21652 читателя
27756042 пользователя, достижимых за 1 шаг
Среднее число читателей среди читателей PacktPub: 1281.92
14285 отметок 'Мне нравится' (0.73 на твит, 0.66 на пользователя)
550860 ретвитов (28.33 на твит, 25.44 на пользователя)
```

Как видите, число читателей вне конкуренции: у издательства Packt Publishing их почти в 20 раз больше, чем у меня. Интереснее выглядит сравнение среднего числа ретвитов и отметок «Мне нравится»; очевидно, мои читатели намного более вовлечены в мой материал, чем читатели издательства PacktPub. Достаточно ли этого, чтобы объявить авторитетом меня, а не издательство PacktPub? Разумеется, нет. Здесь мы наблюдаем естественное следствие факта, что мои твиты посвящены более конкретным темам (Python и наука о данных) и, как следствие,

мои читатели изначально более заинтересованы в моих публикациях. С другой стороны, материал, производимый издательством Packt Publishing, чрезвычайно разнообразен, так как охватывает самые разные технологии. Это разнообразие находит также отражение в аудитории издательства PacktPub; в ней есть разработчики, проектировщики, ученые, системные администраторы и т. д. По этой причине меньшая доля читателей издательства PacktPub в Twitter рассматривает каждый твит как интересный (то есть заслуживающий ретвита).

Анализ читателей

В реальном мире социальное сообщество – это группа людей, объединенных общими условиями. Такое широкое определение включает, например, сообщества людей, расположенных в одном географическом регионе, людей с одинаковыми политическими или религиозными убеждениями либо людей, разделяющих специфические интересы, такие как чтение книг.

Понятие социальных сообществ также играет центральную роль в социально-медийных платформах. Границы сообщества в виртуальной среде могут быть более размытыми, чем в реальном мире, потому что, например, географический аспект имеет меньшее значение. Так же, как в ситуации с глазу на глаз, сообщество естественным образом проявляется в социально-медийной платформе, когда люди с общими интересами или условием начинают взаимодействовать.

В зависимости от того, осознают ли люди себя членами сообщества, его составной частью, можно сделать первое разграничение между различными типами сообществ. В *явных* сообществах участники и неучастники знают точно, принадлежат они сообществу или нет, и обычно понимают, кто является участником сообщества. Взаимодействия между участниками такого сообщества более частые, чем с неучастниками.

С другой стороны, *неявные* сообщества существуют без явного признания. Участники сообщества этого типа могут разделять общие интересы, не имея четких и сильных связей.

В данном разделе демонстрируется анализ пользовательских данных, делящий набор профилей пользователей на группы для выявления их общих интересов или условий.

Кластеризация, или кластерный анализ, – это методология машинного обучения, используемая для группировки элементов так, чтобы объекты внутри каждой группы (то есть в *кластере*) были больше похожи друг на друга, чем на объекты в других кластерах. Кластеризация относится к категории методов *обучения без учителя*, в том смысле, что анализируемые объекты явно никак не классифицированы. Цель обучения без учителя – обнаружить в данных скрытые структуры.

Типичным примером применениям кластеризации является исследование рынка, например с целью выделить из совокупности группы клиентов с похожим поведением/интересами, чтобы их можно было заинтересовать разными продуктами или маркетинговыми кампаниями. Еще одной важной и интересной областью применения кластеризации является анализ социальных сетей с целью распознавания структуры сообщества внутри более многочисленной группы людей.

Кластеризация – это не определенный алгоритм, а скорее обобщенная задача, для решения которой можно использовать разные алгоритмы. В этом разделе мы остановимся на одном из самых популярных подходов к кластеризации, алгоритме **K-средних**. Алгоритм K-средних – отличный выбор, потому что он прост и понятен, а также имеет высокую вычислительную эффективность, если сравнивать с другими подходами к кластеризации.

В сущности, алгоритм K-средних принимает два аргумента: число входных векторов в n-мерном пространстве, которые представляют объекты, предназначенные для кластеризации, и *K* – число кластеров, на которые требуется разбить входные данные. Применяя алгоритмы обучения без учителя, мы не располагаем информацией об имеющихся в действительности категориях. Точнее, мы даже не знаем правильное (или идеальное) число кластеров. В кластерном анализе эта деталь играет довольно существенную роль, но пока мы не будем слишком беспокоиться на этот счет.

Объекты, подлежащие кластеризации, в предыдущем абзаце были определены как векторы в n-мерном пространстве. Выражаясь проще, это означает, что каждый профиль пользователя должен быть представлен как вектор из *n* числовых элементов, именуемых *признаками*.

Для определения этих признаков мы выберем подход, основанный на описании, предоставленном пользователями, то есть текстовой информации об интересах или занятиях. Процесс преобразования текстового описания в вектор признаков называется векторизацией. Алгоритм K-средних и инструменты векторизации реализованы в библиотеке scikit-learn, которая уже была представлена в главе 1 «*Социальные медиа, социальные данные и Python*» как набор инструментов машинного обучения для среды Python.

Процедура векторизации заключается в разбиении описания на лексемы и присваивании каждой отдельного *веса*. В этом примере принята общеизвестная схема взвешивания на основе статистической меры TF-IDF (https://ru.wikipedia.org/wiki/TF-IDF) – сочетания **частоты термина** (Term Frequency, TF) и **обратной частоты документа** (Inverse Document Frequency, IDF). Частота термина TF – это локальная статистическая величина, она показывает, насколько часто слово появляется в документе. Обратная частота документа IDF – напротив, глобальная статистическая величина, которая показывает, насколько редко слово встречается во всем наборе документов. Эти статистические величины, обычно используемые в поисковых системах и различных приложениях анализа текстов, определяют меру важности слова в данном контексте.

Статистическая мера TF-IDF – это числовой вес, основанный на простом логичном соображении: если слово встречается часто в документе и редко во всем наборе, вероятно, оно является очень представительным для этого конкретного документа и поэтому заслуживает более высокого веса. В этом приложении в качестве документов мы рассматриваем описания пользователей и используем статистическую меру TF-IDF с целью представить эти описания как векторы. Получив n-мерное векторное представление, алгоритм K-средних сможет рассчитать степень сходства между векторами.

В следующем сценарии используются средства, предлагаемые библиотекой scikit-learn, в частности классы `TfidfVectorizer` и `KMeans`:

```
# Chap02-03/twitter_cluster_users.py
import sys
import json
from argparse import ArgumentParser
from collections import defaultdict
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.cluster import KMeans

def get_parser():
    parser = ArgumentParser("Кластеризация читателей")
    parser.add_argument('--filename')
    parser.add_argument('--k', type=int)
    parser.add_argument('--min-df', type=int, default=2)
    parser.add_argument('--max-df', type=float, default=0.8)
    parser.add_argument('--max-features', type=int, default=None)
    parser.add_argument('--no-idf', dest='use_idf', default=True,
                        action='store_false')
    parser.add_argument('--min-ngram', type=int, default=1)
    parser.add_argument('--max-ngram', type=int, default=1)
    return parser

if __name__ == '__main__':
    parser = get_parser()
    args = parser.parse_args()
    if args.min_ngram > args.max_ngram:
        print("Ошибка: неправильное значение для --min-ngram ({}):
              оно не может быть выше --max-value ({})"
              .format(args.min_ngram, args.max_ngram))
        sys.exit(1)

    with open(args.filename) as f:
        # загрузить данные
        users = []
        for line in f:
            profile = json.loads(line)
            users.append(profile['description'])

        # создать векторизатор
        vectorizer = TfidfVectorizer(max_df=args.max_df,
                                     min_df=args.min_df,
                                     max_features=args.max_features,
                                     stop_words='english',
                                     ngram_range=(args.min_ngram,
                                                  args.max_ngram),
                                     use_idf=args.use_idf)
        # выполнить подгонку под данные
        X = vectorizer.fit_transform(users)
        print("Размерности данных: {}".format(X.shape))

        # выполнить кластеризацию
        km = KMeans(n_clusters=args.k)
        km.fit(X)
        clusters = defaultdict(list)
        for i, label in enumerate(km.labels_):
            clusters[label].append(users[i])
```

```
# вывести 10 описаний пользователей для данного кластера
for label, descriptions in clusters.items():
    print('---------- Кластер {}'.format(label+1))
    for desc in descriptions[:10]:
        print(desc)
```

Поскольку класс `TfidfVectorizer` имеет множество атрибутов для настройки способа расчета векторов, мы воспользуемся экземпляром класса `ArgumentParser`, входящего в состав стандартной библиотеки Python, чтобы извлечь соответствующие значения из аргументов командной строки. Функция `get_parser()` определяет эти аргументы так:

- `--filename`: путь к файлу JSONL с данными для анализа;
- `--k`: число кластеров;
- `--min-df`: минимальная частота документа для признака (по умолчанию `2`);
- `--max-df`: максимальная частота документа для признака (по умолчанию `0.8`);
- `--max-features`: максимальное количество признаков (по умолчанию `None`);
- `--no-idf`: логический флаг, отключает вес IDF и использует только TF (по умолчанию вес IDF используется);
- `--min-ngram`: нижняя граница для извлекаемых n-грамм (по умолчанию `1`);
- `--max-ngram`: верхняя граница для извлекаемых n-грамм (по умолчанию `1`).

Например, этот сценарий можно запустить такой командой:

```
$ python twitter_cluster_users.py \
  --filename users/marcobonzanini/followers.jsonl \
  --k 5 \
  --max-features 200 \
  --max-ngram 3
```

Единственными обязательными аргументами являются `--filename`, определяющий имя файла с профилями для анализа, и `--k`, определяющий число кластеров. Все другие аргументы определяют параметры векторизации данных и могут быть опущены.

Число признаков, извлекаемых векторизатором `TfidfVectorizer`, можно явно ограничить с помощью аргумента `--max-features` или аргументов `--min-df` и `--max-df`, указав требуемый диапазон частоты документа. По умолчанию минимальная частота документа равна `2`, то есть векторизатор будет игнорировать признаки, встречающиеся меньше чем в двух документах. Максимальная частота документа по умолчанию равна `0.8`, соответственно, векторизатор будет игнорировать признаки, встречающиеся больше чем в 80% документов. В зависимости от размера набора данных можно подойти к выбору этих значений более или менее консервативно. Цель исключения признаков, исходя из частоты документа, состоит в том, чтобы избежать расчетов для признаков, которые не являются представительными для задачи. Кроме того, ограничение общего числа признаков при помощи `--max-features` также помогает ускорить вычисления, поскольку уменьшает объем входных данных. Если задан этот параметр, выбираются самые частые признаки (в зависимости от их частоты документа).

Аргумент `--no-idf` отличается от других и используется не для определения конкретного значения, а чтобы отключить использование меры IDF (то есть чтобы при вычислении весов признаков рассматривалась только мера TF). Объект `ArgumentParser` позволяет определять, как этот аргумент будет себя вести, предоставляя имя пере-

менной в качестве получателя (`dest=use_idf`) и выполняемое действие, когда аргумент задан: `store_false`. По умолчанию эта переменная принимает значение `True`, если аргумент отсутствует. Мера IDF обычно используется, чтобы уменьшить вес терминов, которые распространены по всему набору документов и, как следствие, не имеют какой-то особой различающей способности. Однако чаще логичнее учитывать меру IDF, потому что во время первичного анализа данных бывает важно проследить за ее влиянием, поэтому наличие параметра для ее отключения является попросту дополнительным инструментом в нашем инструментарии.

Последние два аргумента позволяют выйти за пределы отдельных слов и применить в качестве признаков *n-граммы*. Вообще, n-грамма – это непрерывная последовательность из n элементов. В нашем приложении имеется фрагмент текста, лексемизированный в последовательность слов. Отдельные слова также называются *униграммами* (n-граммы с n = 1). Другими часто используемыми n-граммами являются биграммы (n = 2) и триграммы (n = 3), а в некоторых приложениях могут использоваться и более протяженные n-граммы. Например, для предложения «*съешь еще этих мягких французских булок, да выпей чаю*» биграммы определяются следующим образом: *съешь еще, еще этих, этих мягких, мягких французских* и т. д.

Использование n-грамм вместо униграмм позволяет получать словосочетания. Рассмотрим следующие фиктивные предложения:

- He's a scientist, but he doesn't like data (Он – ученый, но ему не нравятся данные);
- He works as a data scientist (Он работает исследователем-аналитиком).

Используя только униграммы, термины *data* и *scientist* будут получены в обеих строках, но при этом они несут разный смысл, в то время как с биграммами можно получить словосочетание *data scientist*, которое имеет другое значение, нежели эти два слова, взятые по отдельности. По умолчанию нижняя и верхняя границы для n-грамм установлены равными `1`, поэтому используются только униграммы, если не указано иное.

В завершение описания настройки векторизатора `TfidfVectorizer` заметим, что в нем также используется аргумент `stop_words`, определяемый для выделения стоп-слов, взятых из английского словаря. Это список общеупотребительных английских слов, таких как *the* и *and*, которые сами по себе не несут смысловой нагрузки и, очевидно, используются почти в каждом фрагменте текста. В типичном тексте на английском языке такие стоп-слова отфильтровались бы уже параметром `max_df` (так как их частота документа будет близка к 100%), но пользователи Twitter часто не пользуются обычным английским. Вместо этого они используют особые ключевые слова, что позволяет преодолевать ограничение на число символов. Векторизатор `TfidfVectorizer` также позволяет передавать в атрибуте `stop_words` пользовательский список слов, если потребуется его персонализировать.

Описанный выше сценарий производит интересные результаты. Для краткости приведу лишь их часть (рис. 3.2).

Результаты демонстрируют интересное поведение алгоритма K-средних.

В первый кластер читателей были сгруппированы профессора и докторанты, которые главным образом занимаются проблемами анализа текста и извлечения информации. Наличие, например, словосочетаний `PhD candidate` и `Information Retrieval` (аспирант перед защитой диссертации и извлечение информации) в двух описаниях послужило убедительным доказательством сходства пользователей.

```
# ---------- Cluster 0
PhD Student #FutureOfNews, #TextMining, Topic Detection & Tracking,
Summarization, Information Retrieval, mostly with Breaking News
Data
PhD candidate - How and what can search engines learn from their
users?
PhD candidate. Information Retrieval. Data Mining. Open Source.
PhD Candidate | Semantic Search in eDiscovery | Information
Retrieval, Text Mining
# ---------- Cluster 1
Co-fundador de Viz Analytics. Diplomado en Ingeniería Informatica
de Gestión y con un Master Universitario en Inteligencia
Artificial.
Lenguas, comunicación y nuevos medios. No tuitear en caso de
incendio.
Lic. en Administración mención informática, promotor del software
libre y las oportunidades que brinda.
# ---------- Cluster 2
I am part time web-developer and I love python
boiling down to Python and Infosec nowadays. Any advices?
Python Padawan, Django noob.
Sherlock of #Data, Love #machinelearning #python
# ---------- Cluster 3
Data Scientist
Data scientist in the making
Data scientist, interested in text analysis, social networks,
artificial and natural intelligence, data, complexity, illusions...
Data Scientist
Wanna be a Data Scientist: using data for good
```

Рис. 3.2 ❖ Часть результатов

Во второй кластер попали испаноязычные пользователи. Алгоритм K-средних и векторизатор `TfidfVectorizer` не различают естественных языков. Единственный лингвистический аспект заключен в процедуре удаления стоп-слов, которые являются общеупотребительными английскими словами. Действительно ли алгоритм K-средних способен распознать язык и использовать эту информацию как основу для объединения пользователей в группу? Имейте в виду, что векторы, которые мы создаем, основаны на представлении данных в виде мешка слов (или мешка n-грамм, если быть точным), поэтому сходство просто основано на наложении слов и n-грамм. Не будучи испаноговорящими, можно распознать несколько слов (например, `en`, `y` и `de`), которые широко распространены в разных профилях. Эти ключевые слова довольно редко встречаются во всей совокупности читателей (главным образом англоязычных), что отразилось в высоком значении веса IDF. Они к тому же являются испанскими стоп-словами, но поскольку список стоп-слов явно ориентирован на английский язык, они попали в набор признаков. Помимо языка, эти профили имеют мало общего. Если первый и третий упоминают несколько терминов, имеющих отношение к информатике, то второй говорит *Languages, communication and new media. Don't tweet in case of fire* (Языки, общение и новые медиа. Не твитить, только при пожаре). Связь с другими профилями довольно расплывчата, и тем не менее этот результат вызывает интерес.

Третий и четвертый кластеры так же, как и первый, самоочевидны и на удивление логичны: кластер 3 состоит в основном из разработчиков на Python, а кластер 4 – из исследователей-аналитиков.

Поскольку инициализация алгоритма K-средних имеет случайный компонент, повторное выполнение сценария ровно с теми же аргументами не гарантирует, что будут получены те же результаты. Заинтересованным читателям рекоменду-

ется поэкспериментировать с разными значениями аргументов (например, увеличить или уменьшить число признаков, попробовать использовать n-граммы разной длины, сузить или расширить диапазон частот документов и т. д.) и понаблюдать, как они влияют на поведение алгоритма.

В начале обсуждения алгоритма K-средних упомянулось, что помимо самих данных другим важным параметром алгоритма является число кластеров *K*. Поиск оптимальной величины *K* для того или иного набора данных сам по себе является задачей для исследования, и данная процедура отличается от того, как выполняется фактическая кластеризация. Для достижения этой цели разработан целый ряд методов (например: https://en.wikipedia.org/wiki/Determining_the_number_of_clusters_in_a_data_set).

Самый простой подход – эмпирическое правило, согласно которому K устанавливается равным квадратному корню из отношения $n/2$, где n – число объектов в наборе данных. Другие подходы подразумевают использование некой меры внутрикластерной связности, например метод локтя или силуэтный метод.

Анализ диалога

После обсуждения профилей пользователей и того, как они явно связаны отношениями читатель/друг, в этом разделе мы займемся анализом взаимодействия другого типа – диалога. В Twitter пользователи могут опубликовать твит *в ответ на* ту или иную информацию. Когда два или более пользователей сделают это, может развернуться полноценный диалог.

На рис. 3.3 изображен диалог, представленный как сеть. Каждый узел – это твит (однозначно определяемый его идентификатором), а каждое ребро представляет *ответ на* отношение.

Этот тип отношения имеет явно выраженную ориентированность, поскольку всегда направлено в одну сторону (связь родитель/потомок). Например, если твит 2 является ответом на твит 1, мы не увидим твит 1 как ответ на твит 2. Мощность этого отношения всегда равна 1, в том смысле, что тот или иной твит может быть ответом на один и только один твит (но в ответ на данный твит может быть отправлено несколько твитов, что позволяет класифицировать это отношение как «один-ко-многим»). Кроме того, циклы недопустимы (например, невозможна последовательность отношений 1 в 2, 2 в 3, 3 в 1). По этим причинам граф, который мы представляем, относится к категории ориентированных ациклических графов (Directed Acyclic Graph, DAG) (https://ru.wikipedia.org/wiki/Направленный_ациклический_граф). Если быть более точным, тип графа, который мы здесь представляем, часто называется ориентированным деревом:

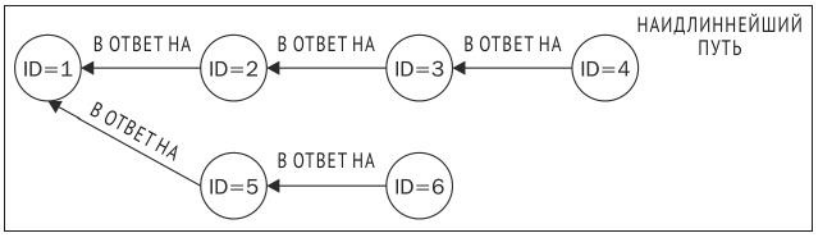

Рис. 3.3 ❖ Пример диалога, представленного как сеть

Хотя явно на рис. 3.3 это не показано, можно заметить, что отношение имеет четко установленные временные рамки: на твит можно ответить, только если он был опубликован.

Описывая твиты и ответы как граф, появляется возможность использовать свойства графа и применять алгоритмы из теории графов для анализа диалога.

Например, *степень* узла есть число узлов в графе, которые являются дочерними для данного узла. С точки зрения диалога степень узла соответствует числу ответов. В примере на рис. 3.3 с узлом 1 непосредственно связаны два узла, поэтому его степень равна двум. С узлами 2, 3 и 5 связан единственный узел, поэтому их степень равна единице. Наконец, с узлами 4 и 6 не связан ни один узел, поэтому их степень равна нулю. Узлы 4 и 6 также называются *листьями* дерева и представляют тупик в диалоге. С другой стороны, начало диалога, узел 1, также называется *корнем* дерева.

Фундаментальным понятием в теории графов является понятие *пути*, который суть последовательность ребер, соединяющих последовательность отдельных вершин. Обход графа, представляющего последовательность твитов, эквивалентен следованию за развитием диалога. Интересной задачей является поиск пути максимальной длины, или длиннейшего пути.

Чтобы применить эти графовые понятия к нашим данным Twitter, можно задействовать библиотеку **NetworkX**, уже представленную в главе 1 «*Социальные медиа, социальные данные и Python*». Она реализует эффективные алгоритмы для работы с графовыми структурами и имеет простой интерфейс.

Следующий сценарий принимает файл JSONL с твитами и создает ориентированный граф, который мы только что обсудили:

```
# Chap02-03/twitter_conversation.py
import sys
import json
from operator import itemgetter
import networkx as nx

def usage():
    print("Применение:")
    print("python {} <имя_файла>".format(sys.argv[0]))

if __name__ == '__main__':
    if len(sys.argv) != 2:
        usage()
        sys.exit(1)
    fname = sys.argv[1]
    with open(fname) as f:
        graph = nx.DiGraph()
        for line in f:
            tweet = json.loads(line)
            if 'id' in tweet:
                graph.add_node(tweet['id'],
                               tweet=tweet['text'],
                               author=tweet['user']['screen_name'],
                               created_at=tweet['created_at'])
                if tweet['in_reply_to_status_id']:
                    reply_to = tweet['in_reply_to_status_id']
```

```
            if tweet['in_reply_to_status_id'] in graph \
            and tweet['user']['screen_name'] != \
                graph.node[reply_to]['author']:
                graph.add_edge(tweet['in_reply_to_status_id'],
                               tweet['id'])

# Вывести несколько элементарных статистик
print(nx.info(graph))

# Твит с самым большим числом ответов
sorted_replied = sorted(graph.degree_iter(),
                        key=itemgetter(1),
                        reverse=True)
most_replied_id, replies = sorted_replied[0]
print("Твит с самым большим числом ответов ({}):".format(replies))
print(graph.node[most_replied_id])

# Поиск длиннейшего диалога
print("Самое длинное обсуждение:")
longest_path = nx.dag_longest_path(graph)
for tweet_id in longest_path:
    node = graph.node[tweet_id]
    print("{} (автор {}, создан {})".format(node['tweet'],
                                            node['author'],
                                            node['created_at']))
```

Его можно запустить командой:

```
$ python twitter_conversation.py <имя_файла>
```

Сценарий импортирует библиотеку NetworkX под псевдонимом `nx`, как обсуждалось в главе 1 «*Социальные медиа, социальные данные и Python*». Инициализирует пустой ориентированный граф, который реализован классом `nx.DiGraph`. Каждая строка входного файла JSONL представляет один твит, поэтому сценарий в цикле обходит весь файл, добавляя один узел для каждого твита. Применяемая с этой целью функция `add_node()` принимает один обязательный аргумент – идентификатор узла, за которым идет ряд необязательных именованных аргументов, через которые можно присвоить узлу дополнительные атрибуты. В данном случае мы включили такие атрибуты, как экранное имя автора, полный текст твита и время создания.

После создания узла проверяется, был ли твит отправлен в ответ на другой твит. В этом случае может потребоваться добавить ребро между узлами, но прежде нужно удостовериться, что твит, на который получен ответ, уже присутствует в графе. Это необходимо, чтобы избежать включения ответов на твиты, отсутствующие в наборе данных (например, на твиты, опубликованные до начала сбора данных с помощью потокового интерфейса Streaming API). Если искомый узел отсутствует в графе, функция `add_edge()` могла бы его добавить, но у нас не будет информации о его атрибутах, кроме идентификатора. Сценарий также проверяет, является ли автор твита и автор ответа разными людьми. Это вызвано тем, что пользовательский интерфейс Twitter автоматически группирует твиты, которые являются частью диалога, однако при этом некоторые пользователи, которые, к примеру, комментируют событие в прямом эфире или просто пишут комментарий длиннее 140 символов, часто делают это, отвечая на свои собственные твиты, чтобы таким

простым способом создать ветку из нескольких твитов. Это, пожалуй, не лучшая особенность, в действительности такая ветка не является диалогом (скорее наоборот), поэтому мы решили игнорировать ветки подобного рода. Если вы интересуетесь поиском монологов, можете изменить этот программный код и добавлять ребра, только когда автор твита и автор его ответов – одно и то же лицо.

После создания графа сценарий выводит несколько элементарных статистик, возвращаемых функцией `nx.info()`. Затем выявляет твит с самым большим количеством ответов (то есть узел с самой высокой степенью) и самым длинным диалогом (длиннейшим путем).

Поскольку функция `degree_iter()` возвращает итератор для последовательности кортежей (узел, степень), в сценарии используется `itemgetter()`, чтобы выполнить сортировку по степени в обратном порядке – в первом элементе отсортированного списка будет храниться твит с самым большим количеством ответов.

Поиск длиннейшего диалога реализует функция `dag_longest_path()`, возвращающая список идентификаторов узлов. Чтобы реконструировать беседу, достаточно просто выполнить обход этих идентификаторов и вывести данные.

Для файла `stream__RWC2015__RWCFinal_Rugby.jsonl`, созданного в главе 2 «*Твиттер – хештеги, темы и временные ряды*», этот сценарий вывел следующий результат (для краткости вывод длиннейшего диалога опущен):

```
Name:
Type: DiGraph
Number of nodes: 198004
Number of edges: 1440
Average in degree: 0.0073
Average out degree: 0.0073
Твит с самым большим числом ответов (15):
{'author': 'AllBlacks', 'tweet': 'Get ready for live scoring here, starting
shortly. You ready #AllBlacks fans? #NZLvAUS #TeamAllBlacks #RWC2015',
'created_at': 'Sat Oct 31 15:49:22 +0000 2015'}
Самое длинное обсуждение:
# ...
```

Как видите, число твитов намного больше числа ребер. Это объясняется тем, что многие твиты не связаны с другими, то есть только небольшая часть твитов отправляется в ответ на другие твиты. Это число, разумеется, может варьировать в зависимости от данных и от решения о включении ответов на свои твиты и на старые твиты (то есть на твиты, отсутствующие в наборе данных).

Привязка твитов к географической карте

Этот раздел посвящен визуальному представлению твитов с использованием географических карт. Визуализация данных позволяет представить данные в легкоусвояемом виде, поскольку изображение помогает получить общее представление о наборе данных.

В небольшой части твитов можно найти сведения о географических координатах устройства пользователя. Хотя многие пользователи отключают эту функциональность на своем мобильном устройстве, все равно имеется интересная возможность проанализировать географическое распределение твитов.

Этот раздел знакомит с GeoJSON, распространенным форматом представления структур географических данных, и процедурой создания интерактивных географических карт твитов.

От твитов к GeoJSON

Формат **GeoJSON** (http://geojson.org) основан на формате JSON и служит для хранения структур географических данных. Объект GeoJSON может представлять геометрию, признак или коллекцию признаков. Геометрии содержат только информацию о геометрической фигуре, такой как точка, ломаная линия, многоугольник или более сложная фигура. Признаки расширяют это понятие и содержат геометрию плюс дополнительные (специальные) свойства. Наконец, коллекция признаков является просто списком признаков.

Структура данных GeoJSON – это всегда объект JSON. В следующем фрагменте приводится пример объекта GeoJSON, представляющего коллекцию с двумя точками, каждая из которых отражает координаты конкретного города:

```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "geometry": {
        "type": "Point",
        "coordinates": [
          -0.12
          51.5
        ]
      },
      "properties": {
        "name": "London"
      }
    },
    {
      "type": "Feature",
      "geometry": {
        "type": "Point",
        "coordinates": [
          -74,
          40.71
        ]
      },
      "properties": {
        "name": "New York City"
      }
    }
  ]
}
```

В этом объекте GeoJSON первым является ключ `type`, описывающий тип представляемого объекта. Это обязательное поле, и оно может иметь одно из следующих значений:

- `Point`: точка, используется для представления одиночной позиции;
- `MultiPoint`: список точек, представляет несколько позиций;
- `LineString`: ломаная линия, определяет последовательность линий, соединяющих две или более позиций;
- `MultiLineString`: список ломаных линий;
- `Polygon`: многоугольник, представляет ломаную линию, образующую замкнутый контур, то есть первая и последняя позиции совпадают;
- `GeometryCollection`: список различных геометрий;
- `Feature`: признак (иногда именуемый геообъектом), один из предыдущих элементов (кроме `GeometryCollection`) с дополнительными специальными свойствами;
- `FeatureCollection`: список признаков.

Поле `type` в предыдущем примере имеет значение `FeatureCollection`, поэтому можно ожидать, что поле `features` будет списком объектов (признаков с типом `Feature`).

Два признака в примере – это простые точки, поэтому в обоих случаях поле `coordinates` содержит массив из двух элементов: долготы и широты. Это поле допускает наличие третьего элемента, представляющего высоту (если он опущен, высота предполагается равной 0).

Разобравшись в структуре, можно попробовать извлечь географическую информацию из набора твитов. Следующий сценарий `twitter_make_geojson.py` читает набор твитов в формате JSON Lines и создает файл GeoJSON с твитами, включающими географическую информацию:

```
# Chap02-03/twitter_make_geojson.py
import json
from argparse import ArgumentParser

def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--tweets')
    parser.add_argument('--geojson')
    return parser

if __name__ == '__main__':
    parser = get_parser()
    args = parser.parse_args()

    # Прочесть коллекцию твитов и построить структуру геоданных
    with open(args.tweets, 'r') as f:
        geo_data = {
            "type": "FeatureCollection",
            "features": []
        }
        for line in f:
            tweet = json.loads(line)
            try:
                if tweet['coordinates']:
                    geo_json_feature = {
                        "type": "Feature",
                        "geometry": {
                          "type": "Point",
```

```
                    "coordinates": tweet['coordinates']['coordinates']
                },
                "properties": {
                    "text": tweet['text'],
                    "created_at": tweet['created_at']
                }
            }
            geo_data['features'].append(geo_json_feature)
        except KeyError:
            # пропустить, если документ json
            # не является твитом (ошибки и пр.)
            continue

    # Сохранить геоданные
    with open(args.geojson, 'w') as fout:
        fout.write(json.dumps(geo_data, indent=4))
```

Для чтения аргументов командной строки сценарий использует экземпляр класса `ArgumentParser`. Его можно запустить командой:

```
$ python twitter_make_geojson.py \
  --tweets stream__RWC2015__RWCFinal_Rugby.jsonl \
  --geojson rwc2015_final.geo.json
```

Сценарий читает файл `stream__RWC2015__RWCFinal_Rugby.jsonl` (аргумент `--tweets`), собранный, как описано в главе 2 «*Твиттер – хештеги, темы и временные ряды*», и сохраняет результат в файле `rwc2015_final.geo.json`, как определено в аргументе `--geojson`.

В качестве структуры геоданных выбрана коллекция признаков `FeatureCollection`, представленная в сценарии словарем `geo_data`. Выполняя обход содержимого в файле, содержащем набор данных с твитами, сценарий загружает каждый твит и добавляет его как одиночный признак в коллекцию признаков. Для каждого признака сохраняются геометрия как точка с координатами, а также дополнительные свойства, такие как текст твита и время его создания. Если какой-либо документ JSON в наборе данных не является твитом (например, ошибка, возвращенная программным интерфейсом Twitter API), отсутствие необходимых атрибутов инициирует ошибку `KeyError`, которая перехватывается и игнорируется блоком `try/except`.

Заключительная часть сценария просто выводит географическую информацию в заданный файл JSON, готовый к применению в пользовательской географической карте.

Простота создания карт с Folium

Этот раздел знакомит с библиотекой Python Folium (https://folium.readthedocs.io/en/latest/), позволяющей с минимальными усилиями создавать интерактивные карты.

Библиотека Folium образует мост между средствами обработки данных на Python и средствами создания пользовательского интерфейса на JavaScript. Точнее, она позволяет разработчикам на Python объединить данные GeoJSON и TopoJSON с библиотекой **Leaflet**, одной из наиболее многофункциональных клиентских библиотек для создания интерактивных географических карт.

Преимущество такой библиотеки, как Folium, состоит в том, что она берет на себя автоматическую трансляцию структур данных из Python в JavaScript, HTML и CSS. Она избавляет разработчика на Python от изучения клиентских технологий и позволяет ему оставаться на территории Python и просто выгружать результат, создаваемый библиотекой, в файл HTML (или выводить его непосредственно в блокнот Jupyter Notebook).

Установить библиотеку можно с помощью `pip`, как обычно:

```
$ pip install folium
```

Примеры в этом разделе основаны на версии библиотеки Folium 0.2, поэтому всегда стоит лишний раз заглянуть в документацию: данная библиотека – еще совсем молодой проект, и в ней возможны существенные изменения интерфейса.

Следующий пример демонстрирует создание простой географической карты с центром в Европе, на которую нанесены два маркера, один в Лондоне и другой в Париже:

```
# Chap02-03/twitter_map_example.py
from argparse import ArgumentParser
import folium

def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--map')
    return parser

def make_map(map_file):
    # Карта
    sample_map = folium.Map(location=[50, 5], zoom_start=17)

    # Маркер для Лондона
    london_marker = folium.Marker([51.5, -0.12], popup='London')
    london_marker.add_to(sample_map)

    # Маркер для Парижа
    paris_marker = folium.Marker([48.85, 2.35], popup='Paris')
    paris_marker.add_to(sample_map)

    # Сохранить в файл HTML
    sample_map.save(map_file)

if __name__ == '__main__':
    parser = get_parser()
    args = parser.parse_args()

    make_map(args.map)
```

Данный сценарий принимает один аргумент с именем выходного файла, используя экземпляр класса `ArgumentParser`, и его можно запустить, например, следующей командой:

```
$ python twitter_map_example.py --map example_map.htm
```

По завершении сценария появится файл `example_map.html` с результатом, который можно открыть в браузере. На рис. 3.4 показан результат работы этого сценария.

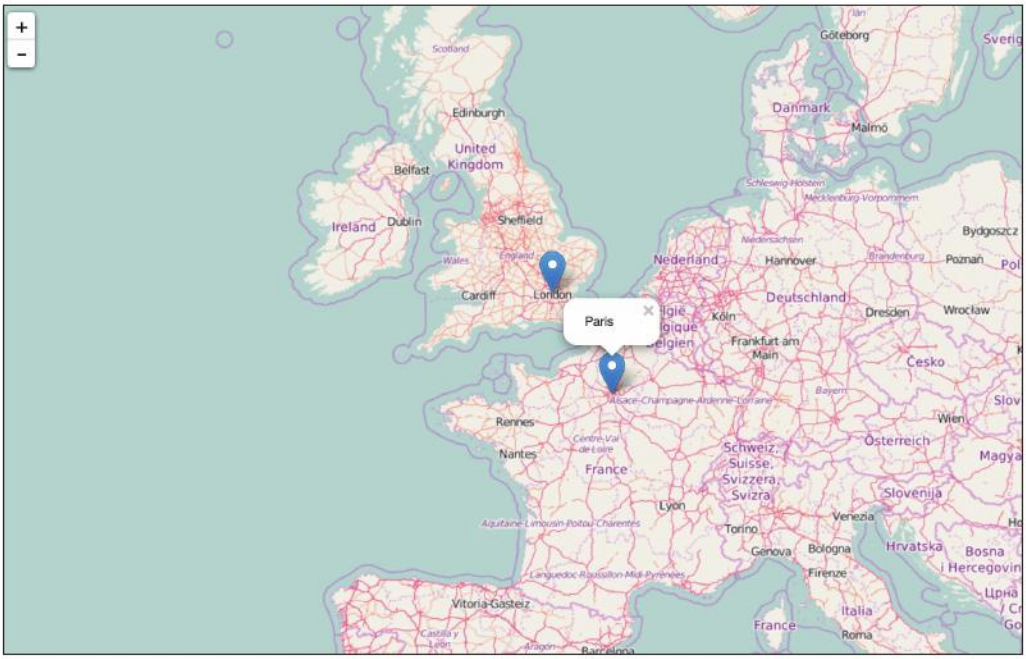

Рис. 3.4 ❖ Пример географической карты, созданной при помощи библиотеки Folium

Основная логика сценария сосредоточена в функции `make_map()`. Она сначала создает объект карты `folium.Map` с центром в определенной географической точке (массив координат `[широта, долгота]`) и начальным масштабом. Параметр `zoom_start` принимает целое число – чем меньше число, тем мельче масштаб; чем больше, тем крупнее.

После создания карты к ней можно прикрепить маркеры. В примере сценария показано, как создавать маркеры с определенными географическими координатами и соответствующими всплывающими подсказками, появляющимися в виде облачка. На рис. 3.4 показано состояние карты после щелчка на маркере, отмечающем Париж, и появления всплывающей подсказки.

Рассмотрев простой пример использования библиотеки Folium, попробуем применить описанные идеи к нашему набору твитов. Следующий пример демонстрирует, как загрузить список маркеров из файла GeoJSON, подобного созданному выше:

```
# Chap02-03/twitter_map_basic.py
from argparse import ArgumentParser
import folium

def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--geojson')
    parser.add_argument('--map')
    return parser
```

```
def make_map(geojson_file, map_file):
    tweet_map = folium.Map(location=[50, 5], zoom_start=5)
    geojson_layer = folium.GeoJson(open(geojson_file), name='geojson')
    geojson_layer.add_to(tweet_map)
    tweet_map.save(map_file)

if __name__ == '__main__':
    parser = get_parser()
    args = parser.parse_args()
    make_map(args.geojson, args.map)
```

Данный сценарий, как обычно, использует экземпляр класса `ArgumentParser`, и его можно запустить следующей командой:

```
$ python twitter_map_basic.py \
  --geojson rwc2015_final.geo.json \
  --map rwc2015_final_tweets.html
```

Аргумент `--geojson` служит для передачи созданного ранее файла с информацией в формате GeoJSON. В аргументе `--map` передается имя выходного файла, в данном случае `rwc2015_final_tweets.html`, который в браузере отображается, как показано на рис. 3.5.

Основное отличие этого сценария от предыдущего заключено в реализации функции `make_map()`. Как и прежде, она начинается с инициализации объекта карты. Но на этот раз вместо поочередного добавления маркеров мы заполняем слой, который добавляется поверх карты, используя файл GeoJSON. Объект `folium.GeoJson` автоматически преобразует данные из формата JSON, поэтому карта заполняется с минимальным усилием:

Рис. 3.5 ❖ Простая географическая карта с твитами

Как и прежде, географическая карта на рис. 3.5 создана с центром в Европе. Можно заметить, что бóльшая часть маркеров сосредоточена в Англии (событие

имело место в Лондоне), но с начальным масштабом трудно понять, как твиты распределены локально.

Проблему можно решить, выбрав более крупный масштаб. Карта это позволяет – она интерактивная. Но библиотеки Folium и Leaflet предлагают дополнительный вариант группировки маркеров в зоне плотного их расположения.

Следующий сценарий использует объект `MarkerCluster`, который специально предназначен для этой цели:

```
# Chap02-03/twitter_map_clustered.py
from argparse import ArgumentParser
import folium

def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--geojson')
    parser.add_argument('--map')
    return parser

def make_map(geojson_file, map_file):
    tweet_map = folium.Map(location=[50, 5], zoom_start=5)
    marker_cluster = folium.MarkerCluster().add_to(tweet_map)
    geojson_layer = folium.GeoJson(open(geojson_file), name='geojson')
    geojson_layer.add_to(marker_cluster)
    tweet_map.save(map_file)

if __name__ == '__main__':
    parser = get_parser()
    args = parser.parse_args()
    make_map(args.geojson, args.map)
```

Данный сценарий, как обычно, использует экземпляр класса `ArgumentParser`, и его можно запустить командой:

```
$ python twitter_map_clustered.py \
  --geojson rwc2015_final.geo.json \
  --map rwc2015_final_tweets_clustered.html
```

Назначение аргументов совпадает с предыдущим сценарием. Но на этот раз результат сохраняется в файле `rwc2015_final_tweets_clustered.html`, который можно открыть в браузере.

На рис. 3.6 изображена часть карты вокруг Лондона после увеличения масштаба. Теперь можно наблюдать эффект объединения маркеров в кластеры, отображаемые кружками с числами, сообщающими количество элементов в них.

Если навести указатель на любой такой кластер, карта выделит область, представленную этим кластером, и пользователь сможет лучше разобраться в локальной плотности, даже не прибегая к увеличению масштаба. На рис. 3.6 выделен кластер со 100 элементами на юго-западе Лондона (стадион Твикенхем, где проходил финальный матч).

Библиотека Folium обладает множеством интересных средств, помогающих дать пользователю более богатый опыт взаимодействия. Например, можно смешать использование документа GeoJSON, кластеры и всплывающие подсказки, чтобы пользователь мог щелкать на маркерах и видеть конкретные твиты.

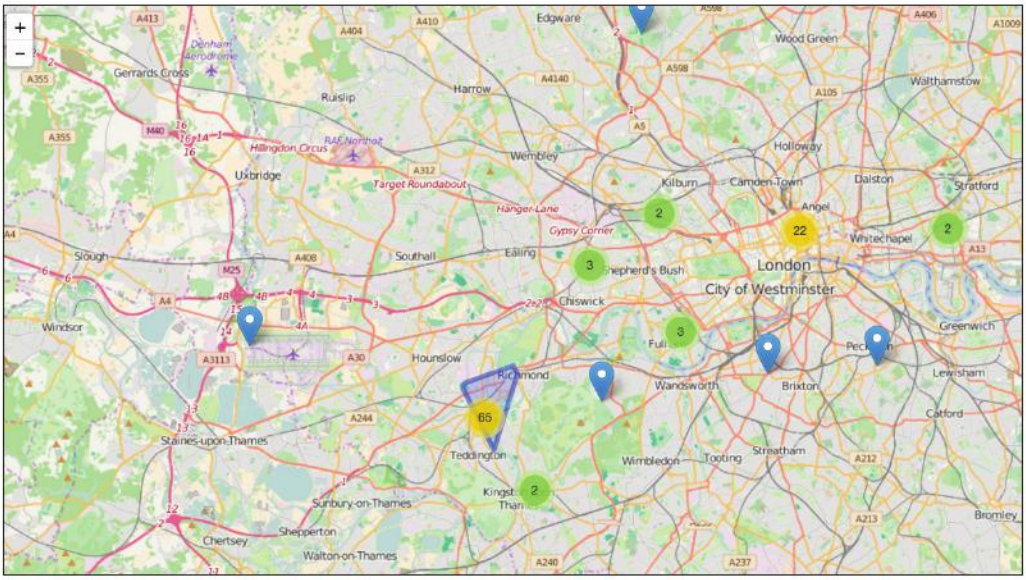

Рис. 3.6 ❖ Пример географической карты с кластеризованными маркерами

Пример создания такой географической карты с кластерами и всплывающими окнами приводится ниже:

```
def make_map(geojson_file, map_file):
    tweet_map = folium.Map(location=[50, 5], zoom_start=5)
    marker_cluster = folium.MarkerCluster().add_to(tweet_map)
    geodata = json.load(open(geojson_file))
    for tweet in geodata['features']:
        tweet['geometry']['coordinates'].reverse()
        marker = folium.Marker(tweet['geometry']['coordinates'],
                               popup=tweet['properties']['text'])
        marker.add_to(marker_cluster)
    tweet_map.save(map_file)
```

Эту версию функции `make_map()` можно подставить взамен одноименной функции в предыдущем сценарии, поскольку ее интерфейс не изменился.

Важно отметить, что объекты `Marker` ожидают координаты в формате `[широта, долгота]`, тогда как документ GeoJSON использует формат `[долгота, широта]`. По этой причине перед созданием объекта `marker` порядок элементов в массиве с координат изменяется на обратный.

На рис. 3.7 показан пример карты после увеличивания масштаба, чтобы выделить стадион, и щелчка на одном из маркеров, поэтому во всплывающей подсказке можно видеть соответствующий твит.

Важно учитывать, что при большом количестве маркеров могут появиться проблемы, связанные с производительностью пользовательского интерфейса. В частности, при наличии нескольких сотен маркеров со всплывающими подсказками навигация по карте может сильно замедлиться, либо браузер может отказаться ее загружать. Если это произошло, рекомендуем поэкспериментировать с мень-

шим набором данных или уменьшить разнообразие функциональных объектов, используемых на одной карте.

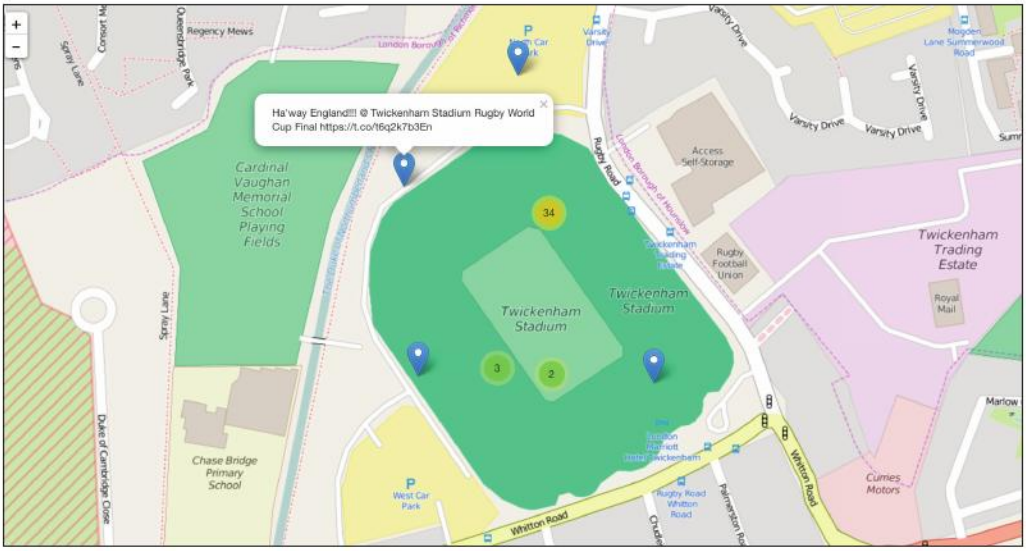

Рис. 3.7 ❖ Предыдущая карта (рис. 3.6) была увеличена, в результате кластеры развернулись в отдельные маркеры

Резюме

Данная глава продолжила обсуждение темы анализа данных из Twitter. После детального рассмотрения текста и частот в главе 2 «*Твиттер – хештеги, темы и временные ряды*» в этой главе все внимание было сосредоточено на анализе связей и взаимодействий пользователей. Мы обсудили способы извлечения информации о явных связях (то есть читателей и друзей) и сравнения степени влияния и вовлеченности между пользователями.

Обсуждение темы сообществ привело к рассмотрению подходов с обучением без учителя для группировки пользователей по описаниям в их профилях с использованием алгоритма кластеризации.

Мы применили методы сетевого анализа к данным, связанным с протекающим событием, чтобы выделить диалоги из потока твитов, и узнали, как выявлять твиты с самым большим количеством ответов и самые длинные диалоги.

Наконец, мы показали, как анализировать географическое распределение твитов путем их нанесения на карту. С помощью библиотеки Folium для Python мы добились красивой визуализации географических данных с минимальными усилиями.

В следующей главе мы переключим наше внимание на другую социальную сеть, вероятно, единственную в своем роде: Facebook.

Глава 4

Сообщения, страницы и взаимодействие пользователей в Facebook

В книге, посвященной анализу социальных медиа, читатель, вероятно, ожидает найти главу, посвященную социальной сети Facebook. Основанная в 2004 г. и первоначально ограничившаяся лишь студентами Гарвардского университета, сегодня Facebook представляет собой компанию стоимостью в несколько миллиардов долларов, с почти 1,5 миллиарда активных пользователей. Высокая популярность делает ее чрезвычайно интересной площадкой для анализа данных.

Эта глава охватывает следующие темы:

- создание приложения для взаимодействия с платформой Facebook;
- взаимодействие с графовым интерфейсом Facebook Graph API;
- анализ сообщений от аутентифицированного пользователя;
- анализ страниц Facebook, визуализация сообщений и измерение степени вовлеченности;
- создание облака слов из набора сообщений.

Интерфейс Facebook Graph API

Графовый программный интерфейс Graph API лежит в основе платформы Facebook и является одним из главных компонентов, который делает возможной интеграцию третьих лиц с данной платформой. Как видно из названия, он предлагает согласованное графоподобное представление данных, отражающее объекты и связи между ними. Различные компоненты платформы позволяют разработчикам получать доступ к данным из социальной сети Facebook и интегрировать ее функции в сторонние приложения.

С точки зрения возможностей анализа данных, с выпуском API версии 2.0 в 2014 г. произошел большой сдвиг. Одним из главных объектов, вызывающих интерес в анализе данных, является социальный граф, то есть список (дружеских) связей между пользователями. Начиная с Graph API версии 2.0 для доступа к этой информации приложения должны явно запрашивать разрешение у друзей пользователя, но при этом API будет возвращать список только тех друзей, которые являются пользователями данного приложения.

Это решение кардинально изменило то, что раньше являлось настоящей золотой жилой для анализа данных. В этом разделе рассматривается создание приложения на Python для анализа данных из Facebook и основы взаимодействия с графовым программным интерфейсом Facebook Graph API.

Регистрация приложения

Для доступа к Facebook API приложение требуется зарегистрировать. В результате такой регистрации разработчики получают учетные данные, необходимые для использования Graph API. Будучи пользователем Facebook, вам необходимо явно зарегистрироваться в качестве разработчика, чтобы создавать приложения, и ваша учетная запись должна быть подтверждена либо по мобильному телефону, либо кредитной картой.

Процедура регистрации на веб-сайте Facebook Developers (https://developers.facebook.com) достаточно проста: после щелчка на пиктограмме в левом верхнем углу экрана откроется всплывающее меню, где нужно выбрать пункт Add a New App (Создать новое приложение), после чего откроется диалог, изображенный на рис. 4.1, где требуется указать отображаемое название Анализ данных социальной сети для нашего демонстрационного приложения (название ограничено 32 символами):

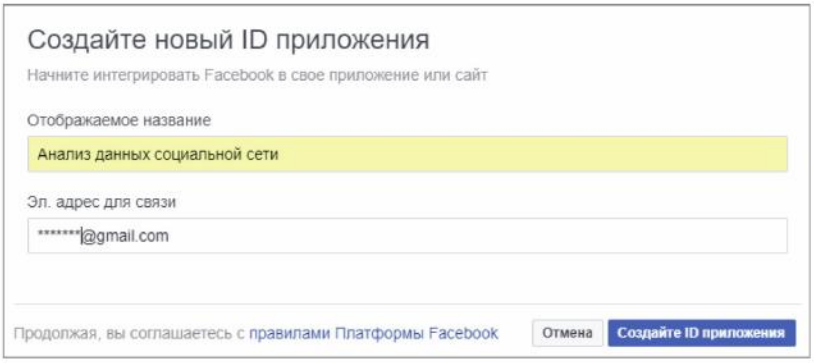

Рис. 4.1 ❖ Диалог создания нового приложения Facebook

После ввода названия приложения можно щелкнуть на кнопке Create App ID (Создайте ID приложения), чтобы подтвердить его создание.

После этого название, выбранное для приложения, появится в меню My Apps (Мои приложения), щелчок на котором откроет инструментальную панель приложения, как показано на рис. 4.2.

Данная панель предоставляет важную информацию, а именно, App ID (Идентификатор приложения) и App Secret (Секрет приложения), необходимые для получения доступа к API. Чтобы увидеть секрет приложения, необходимо ввести пароль Facebook, подтвердив тем самым свою идентичность. Разумеется, этими сведениями не следует ни с кем делиться ввиду очевидных соображений безопасности.

Рядом с полями App ID (Идентификатор приложения) и App Secret (Секрет приложения) имеется также поле API Version (Версия API) с номером версии интер-

фейса (в примере на рис. 4.2 отображается номер v2.9). По умолчанию для нового приложения всегда выбирается последняя версия. На конференции F8 в 2014 г. компания Facebook объявила о своем решении оказывать поддержку определенной версии API в течение, по крайней мере, двух лет. Эту важную информацию следует учитывать, потому что, когда определенная версия API перестает поддерживаться, ваше приложение, если его не обновить, может оказаться неработоспособным.

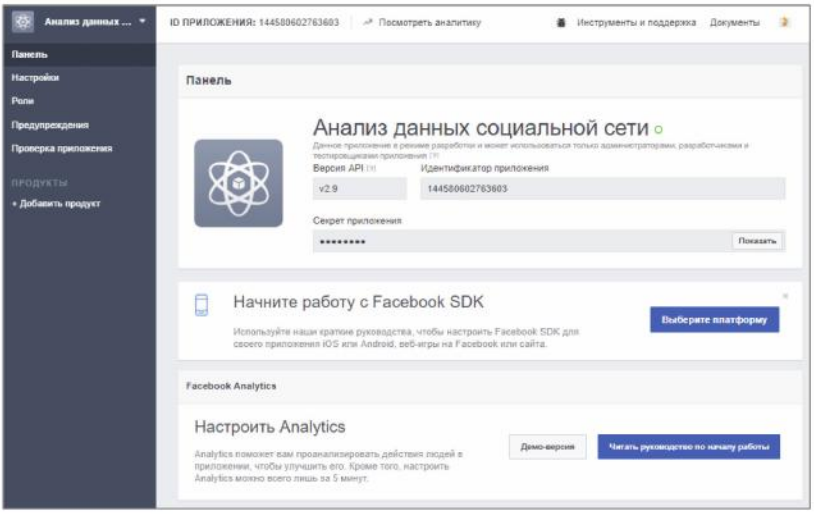

Рис. 4.2 ❖ Вид инструментальной панели приложения

Управление версиями Facebook API. Стратегии управления версиями Facebook API обсуждаются в документации (https://developers.facebook.com/docs/apps/versions); там же обсуждаются даты истечения срока поддержки определенных версий (https://developers.facebook.com/docs/apps/changelog).

Первоначально приложение настроено на работу в режиме разработки. Это делает его доступным только для вас (как автора) и любого пользователя, явно обозначенного как разработчик или тестировщик в меню Roles (Роли). Задержитесь на некоторое время на инструментальной панели и посмотрите, какие функции доступны в базовой конфигурации и каково их назначение.

Аутентификация и безопасность

Чтобы получить доступ к информации о профилях пользователей, а также к информации об их взаимодействиях с другими объектами (к примеру, страницами, местами и т. д.), приложение должно получить маркер доступа с соответствующими разрешениями.

Маркер уникален для сочетания пользователь/приложение и управляет разрешениями, которые пользователь предоставил приложению. Создание маркера доступа требует взаимодействия с пользователем, то есть пользователи должны подтвердить в Facebook (обычно через диалог), что они предоставляют приложению необходимые разрешения.

Для целей тестирования есть еще один способ получить маркер доступа – через проводника Graph API Explorer (https://developers.facebook.com/tools/explorer); этот инструмент был разработан в Facebook для обеспечения разработчиков удобным интерфейсом к Graph API. На рис. 4.3 демонстрируется использование проводника Graph API Explorer. После выбора приложения из списка можно щелкнуть на пункте Get User Access Token (Получить маркер доступа пользователя) в меню Get Token (Получить маркер):

Рис. 4.3 ❖ Создание маркера доступа из проводника Graph API Explorer

В результате откроется диалог, изображенный на рис. 4.4. Теперь можно выбрать разрешения для включения в маркер доступа. После подтверждения разрешений будет сгенерирован алфавитно-цифровой маркер доступа, действительный в течение двух часов с момента его создания. Щелчок на маленькой информационной пиктограмме рядом с маркером покажет эту информацию.

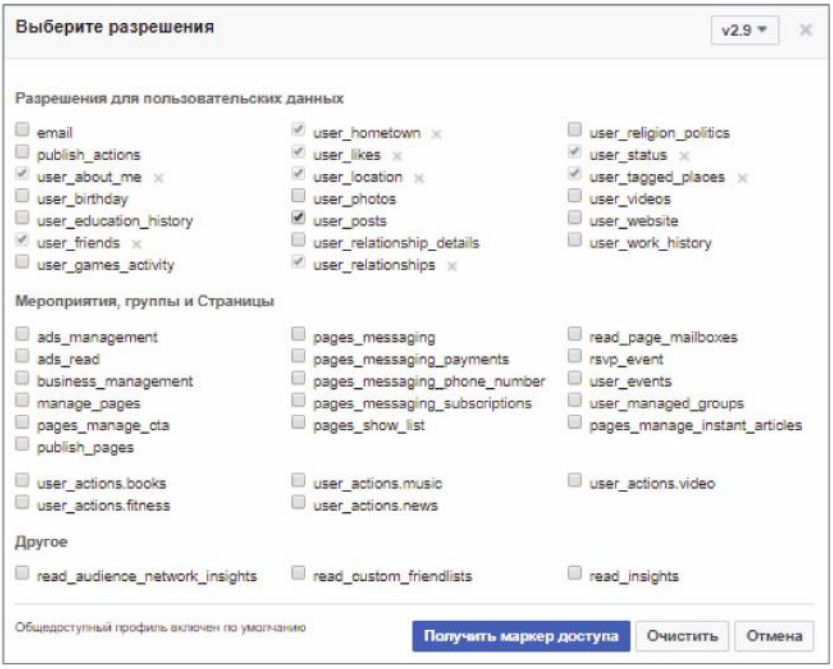

Рис. 4.4 ❖ Выбор разрешений для маркера доступа из проводника Graph API Explorer

Как показано на рис. 4.4, разрешения для приложения Facebook расписаны очень подробно. Поэтому пользователи, устанавливающие ваше приложение, будут иметь полное представление о том, к каким данным они открывают доступ вашему приложению.

Для примеров, следующих ниже, мы будем использовать несколько полей, включая полное имя пользователя, местоположение и сообщения. Чтобы сценарии могли получать эту информацию, необходимо отметить соответствующие разрешения (например, user_location, user_posts и т. д.).

Когда пользователь получает доступ к приложению впервые, платформа Facebook выводит диалог с итоговым списком разрешений. В дальнейшем пользователь будет иметь возможность пересмотреть список полномочий, выдаваемых приложению.

Доступ к Facebook Graph API из Python

После определения деталей приложения появляется возможность программного доступа к Facebook Graph API из сценариев на Python.

Facebook не предлагает официальных клиентских инструментов для Python. Реализация собственного клиентского модуля с использованием библиотеки **requests** могла бы стать интересным упражнением и помогла бы разобраться в особенностях API, но, к счастью, уже имеются другие варианты, которые упрощают эту процедуру.

В наших примерах мы будем использовать библиотеку **facebook-sdk**, тоже основанную на библиотеке requests, которая обеспечивает простой интерфейс обмена данными с веб-службой. Во время написания данной главы последней доступной в PyPI версией библиотеки была 1.0.0 (на момент перевода – июнь 2017 – последняя версия 2.0.0), которая полностью поддерживает Python 3. Предыдущие версии имели некоторые шероховатости совместимости с Python 3. Данную библиотеку можно устанавить при помощи pip следующей командой:

```
$ pip install facebook-sdk
```

Получив временный маркер и выполнив шаги, описанные в предыдущем разделе, можно сразу же протестировать установленную библиотеку.

Прежде всего, по аналогии с настройкой доступа к Twitter в главе 2 «*Твиттер – хештеги, темы и временные ряды*» и главе 3 «*Пользователи, читатели и сообщества в Twitter*», сохраним маркер в переменной окружения, которая потом будет читаться сценарием. В Unix-подобных ОС наберите в командной строке следующую команду:

```
$ export FACEBOOK_TEMP_TOKEN="ваш-временный-маркер"
```

в ОС Windows:

```
$ set FACEBOOK_TEMP_TOKEN=ваш-временный-маркер
```

Следующий сценарий `facebook_my_profile.py` подключается к Graph API и запрашивает профиль аутентифицированного пользователя:

```
# Chap04/facebook_my_profile.py
import os
import json
import facebook
```

```
if __name__ == '__main__':
    token = os.environ.get('FACEBOOK_TEMP_TOKEN')
    graph = facebook.GraphAPI(token)
    profile = graph.get_object('me', fields='name,location')
    print(json.dumps(profile, indent=4, ensure_ascii=False))
```

Он не требует никаких аргументов, поэтому его можно запустить простой командой:

```
$ python facebook_my_profile.py
```

В результате он выведет дамп объекта JSON, возвращенного интерфейсом Graph API:

```
{
  "name": "Marco Bonzanini",
  "location": {
    "name": "London, United Kingdom",
    "id": "106078429431815"
  },
  "id": "10207505820417553"
}
```

Функция `get_object()` принимает в первом аргументе идентификатор или имя объекта в графе социальной сети Facebook и возвращает информацию о нем. В нашем примере идентификатор `me` – это псевдоним аутентифицированного пользователя. Если опустить второй аргумент и поля, API просто вернет идентификатор и имя объекта. В данном случае мы явно требуем включить в результат имя (`name`) и местоположение (`location`). Как видите, `location` – это не простая последовательность символов, а сложный объект со своими собственными полями (так как ничего другого не было определено, местоположение описывается только полями `id` и `name`).

Класс `GraphAPI` имеет довольно простой интерфейс для чтения данных. Кроме того, он предоставляет средства для публикации и обновления данных в социальной сети Facebook, позволяя приложению взаимодействовать с ней (например, отправлять сообщения на стену аутентифицированного пользователя).

Наибольший интерес для нас представляют следующие основные методы:

- `get_object(id, **args)`: получает объект по его `id`, а также принимает дополнительные именованные аргументы;
- `get_objects(ids, **args)`: получает список объектов по списку `ids`, а также принимает дополнительные именованные аргументы;
- `get_connections(id, connection_name, **args)`: получает список объектов, связанных отношением `connection_name` с объектом, идентифицируемым по `id`, а также принимает дополнительные именованные аргументы;
- `request(path, args=None, post_args=None, files=None, method=None)`: обобщенный метод для реализации специфических запросов к API, использующих путь `path`, как определено в документации API, необязательные аргументы определяют способ выполнения вызова API.

Для загрузки профиля текущего пользователя в сценарии `facebook_my_profile.py` используется метод `get_object()`. В данном случае ему передается необязательный именованный аргумент `fields` с атрибутами, которые мы хотим получить из API.

Полный список атрибутов профилей пользователей определен в документации (https://developers.facebook.com/docs/graph-api/reference/v2.9/user).

Заглянув в описание API, можно увидеть, как персонализировать строковое значение поля, чтобы получить больше информации из того или иного профиля. Также имеется возможность выполнять вложенные запросы и включать подробные сведения об объектах, которые связаны с данным профилем. В нашем примере мы получили местоположение – объект типа **Page** (https://developers.facebook.com/docs/graph-api/reference/page/), то есть Страницу. Поскольку каждый объект `Page` имеет несколько закрепленных за ним атрибутов, их тоже можно включить в запрос, например:

```
profile = graph.get_object("me", fields='name,location{location}')
```

Синтаксическая конструкция `первый_уровень{второй_уровень}` позволяет запрашивать вложенные объекты. В данном конкретном примере такое обозначение выглядит несколько запутанным, поскольку имя `location` используется для получения атрибутов первого и второго уровней. Чтобы избежать путаницы, важно знать и понимать структуру типов данных. `location` на первом уровне представляет атрибут профиля пользователя, а его тип данных – Страница Facebook (со своим собственным идентификатором, именем и другими атрибутами). Напротив, `location` на втором уровне представляет атрибут вышеупомянутой Страницы Facebook – это дескриптор фактического местоположения, включающего, кроме прочих, такие атрибуты, как `latitude` (широта) и `longitude` (долгота). Ниже приводится результат выполнения предыдущего сценария с дополнительным именем `location` на втором уровне:

```
{
  "name": "Marco Bonzanini",
  "location": {
    "name": "London, United Kingdom",
    "id": "106078429431815"
  },
  "id": "10207505820417553",
  "location": {
    "id": "106078429431815",
    "location": {
      "city": "London",
      "latitude": 51.516434161634,
      "longitude": -0.12961888255995,
      "country": "United Kingdom"
    }
  }
}
```

Атрибуты `city`, `country`, `latitude` и `longitude` извлекаются для объекта `location` по умолчанию (https://developers.facebook.com/docs/graph-api/reference/v2.9/location).

Как упоминалось в начале раздела, в новых версиях Facebook Graph API некоторые возможности анализа данных были ограничены. В частности, анализ графа социальной сети (то есть дружеских связей) возможен, только если все вовлекаемые профили являются пользователями нашего приложения. Следующий сценарий пытается получить список друзей пользователя:

```
# Chap04/facebook_get_friends.py
import os
import facebook
import json

if __name__ == '__main__':
    token = os.environ.get('FACEBOOK_TEMP_TOKEN')

    graph = facebook.GraphAPI(token)
    user = graph.get_object("me")
    friends = graph.get_connections(user["id"], "friends")
    print(json.dumps(friends, indent=4))
```

Несмотря на то что этот вызов API требует разрешения `user_friends` от друзей пользователя, которое должно быть предоставлено нашему приложению, этот сценарий все равно получит о друзьях немного данных, поскольку аутентифицированный пользователь (то есть `me`) в настоящее время является единственным пользователем приложения:

```
{
  "data": [],
  "summary": {
    "total_count": 266
  }
}
```

Как видите, нам удалось получить только общее количество друзей конкретного пользователя, а конкретные данные о друзьях представлены пустым списком. Если некоторые из друзей решат использовать наше приложение, мы сможем получить их профили.

Мы завершим этот раздел кратким упоминанием об ограничении на частоту запросов, налагаемых интерфейсом Graph API. Как говорится в документации (https://developers.facebook.com/docs/graph-api/advanced/rate-limiting), с ограничениями на частоту запросов сталкиваются редко. Ограничения рассчитываются на уровне приложения и на уровне пользователя, то есть если приложение достигнет суточного предела, будут ограничены абсолютно все вызовы, выполняемые им, а не только те, которые касаются данного пользователя. Суточные пределы вычисляются на число пользователей в предыдущий день плюс новые входы за сегодняшний день – эта сумма дает базовое число пользователей. Приложению предоставляется возможность сделать 200 вызовов API на пользователя в 60-минутом окне. Для наших примеров этого более чем предостаточно, но вы обязательно загляните в документацию, чтобы понять возможное влияние ограничений частоты вызовов на ваше приложение.

В следующем разделе мы загрузим все сообщения пользователя и проанализируем их.

Анализ сообщений

Познакомившись с библиотекой facebook-sdk, начнем разбираться в возможностях анализа данных. Для начала загрузим наши собственные сообщения (то есть сообщения, опубликованные аутентифицированным пользователем).

Сценарий facebook_get_my_posts.py подключается к Graph API и получает список сообщений, опубликованных пользователем me. Сообщения сохраняются в файле my_posts.jsonl в формате JSON Lines, который мы уже использовали в главе 2 «*Твиттер – хештеги, темы и временные ряды*» и главе 3 «*Пользователи, читатели и сообщества в Twitter*» (каждая строка файла представляет документ JSON):

```
# Chap04/facebook_get_my_posts.py
import os
import json
import facebook
import requests

if __name__ == '__main__':
    token = os.environ.get('FACEBOOK_TEMP_TOKEN')

    graph = facebook.GraphAPI(token)
    posts = graph.get_connections('me', 'posts')

    while True:  # загрузить другие страницы
        try:
            with open('my_posts.jsonl', 'a') as f:
                for post in posts['data']:
                    f.write(json.dumps(post)+"\n")
                # получить следующую страницу
                posts = requests.get(posts['paging']['next']).json()
        except KeyError:
            # страниц больше нет, прервать цикл
            break
```

Данный сценарий не имеет аргументов командной строки, поэтому его легко запустить командой:

```
$ python facebook_get_my_posts.py
```

Сценарий демонстрирует интересный пример постраничного просмотра. Поскольку список сообщений слишком длинный, чтобы его можно было получить одним вызовом API, платформа Facebook поддерживает возможность постраничной загрузки информации.

Начальный вызов API, выполняемый методом get_connections(), возвращает первую страницу сообщений (сохраняется в posts['data']), а также информацию, необходимую для итеративного обхода других страниц, доступную в posts['paging']. Поскольку постраничная загрузка не реализована в библиотеке facebook-sdk, нам придется отступить назад и воспользоваться библиотекой requests напрямую. К счастью, ответ, возвращаемый Graph API, содержит точный URL-адрес очередной страницы сообщений. Если заглянуть в элемент posts['paging']['next'], можно увидеть строку с точным URL-адресом запроса, включающим маркер доступа, номер версии API и все необходимые детали.

Постраничный просмотр выполняется внутри цикла while True, который прерывается исключением KeyError по достижении последней страницы. Поскольку последняя страница не содержит ссылки на posts['paging']['next'], попытка получить доступ к этому ключу словаря возбудит исключение, и работа цикла будет прервана.

После выполнения сценария можно приступать к исследованию содержимого файла my_posts.jsonl. Каждая строка в файле представляет документ JSON, содержащий уникальный идентификатор, текст сообщения и время создания в формате ISO 8601. Вот пример документа JSON, представляющего одно из загруженных сообщений:

```
{
  "created_time": "2015-11-04T08:01:21+0000",
  "id": "10207505820417553_10207338487234328",
  "message": "The slides of my lighting talk at the PyData London
    meetup last night\n"
}
```

Функция get_connections(), так же как get_object(), принимает аргумент fields с дополнительными атрибутами нужного объекта. Ниже представлена версия предыдущего сценария, переделанная, чтобы получить более интересные атрибуты наших сообщений:

```
# Chap04/facebook_get_my_posts_more_fields.py
import os
import json
import facebook
import requests

if __name__ == '__main__':
    token = os.environ.get('FACEBOOK_TEMP_TOKEN')

    graph = facebook.GraphAPI(token)
    all_fields = [
        'message',
        'created_time',
        'description',
        'caption',
        'link',
        'place',
        'status_type'
    ]
    all_fields = ','.join(all_fields)
    posts = graph.get_connections('me', 'posts', fields=all_fields)

    while True:  # загрузить другие страницы
        try:
            with open('my_posts.jsonl', 'a') as f:
                for post in posts['data']:
                    f.write(json.dumps(post)+"\n")
                # получить следующую страницу
                posts = requests.get(posts['paging']['next']).json()
        except KeyError:
            # страниц больше нет, прервать цикл
            break
```

Все необходимые поля объявляются в списке all_fields и затем объединяются в строку через запятую, как того требует Graph API. Затем полученная строка передается методу get_connections() в именованном аргументе fields.

В следующем разделе мы подробнее остановимся на структуре сообщения.

Структура сообщения

Сообщение (`Post`) – это сложный объект, поскольку оно может быть любой частью информации, которую пользователь решит опубликовать. Следующая таблица перечисляет наиболее интересные атрибуты сообщения с их кратким описанием:

Название атрибута	Описание
`id`	Строка, представляющая уникальный идентификатор
`application`	Объект `App` с информацией о приложении, использованном для публикации сообщения
`status_type`	Строка, представляющая тип сообщения (например, `added_photos`, то есть добавленные фото, или `shared_story`, то есть цитируемая информация)
`message`	Строка с текстом сообщения
`created_time`	Строка с датой публикации сообщения в формате ISO 8601
`updated_time`	Строка с датой последнего изменения в формате ISO 8601
`message_tags`	Список профилей, отмеченных в сообщении
`from`	Профиль, отправивший сообщение
`to`	Список профилей, упомянутых в сообщении, или кому адресовано сообщение
`place`	Информация о местоположении, приложенная к сообщению
`privacy`	Объект с настройками конфиденциальности сообщения
`story_tags`	То же самое, что `message_tags`
`with_tags`	Список профилей, отмеченных как *вместе с* автором сообщения
`properties`	Список свойств любого приложенного видео (например, продолжительность видео)

Объект `Post` имеет также другие атрибуты, кроме представленных в таблице. Полный список приводится в официальной документации (https://developers.facebook.com/docs/graph-api/reference/v2.9/post), где многосложность этого объекта еще более очевидна.

Частотно-временной анализ

После загрузки сообщений выполним первый анализ, основанный на времени создания разных сообщений. Цель анализа – выяснить варианты поведения пользователя, как, например, в какое время дня пользователь отправляет бóльшую часть сообщений.

Для чтения аргумента командной строки с именем файла `.jsonl` с сообщениями `facebook_post_time_stats.py` использует экземпляр класса `ArgumentParser`:

```
# Chap04/facebook_post_time_stats.py
import json
from argparse import ArgumentParser
import dateutil.parser
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from datetime import datetime

def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--file',
                        '-f',
                        required=True,
```

```
                        help='Файл .jsonl со всеми сообщениями')
    return parser

if __name__ == '__main__':
    parser = get_parser()
    args = parser.parse_args()
    with open(args.file) as f:
        posts = []
        for line in f:
            post = json.loads(line)
            created_time = dateutil.parser.parse(post['created_time'])
            posts.append(created_time.strftime('%H:%M:%S'))
        ones = np.ones(len(posts))
        idx = pd.DatetimeIndex(posts)

        # фактический ряд (пока состоящий из единиц)
        my_series = pd.Series(ones, index=idx)

        # Передискретизация в почасовые дискретные интервалы
        per_hour = my_series.resample('1H').sum().fillna(0)

        # Построение графика
        fig, ax = plt.subplots()
        ax.grid(True)
        ax.set_title("Частоты сообщений")
        width = 0.8
        ind = np.arange(len(per_hour))
        plt.bar(ind, per_hour)
        tick_pos = ind + width / 2
        labels = []
        for i in range(24):
            d = datetime.now().replace(hour=i, minute=0)
            labels.append(d.strftime('%H:%M'))
        plt.xticks(tick_pos, labels, rotation=90)
        plt.savefig('posts_per_hour.png')
```

Этот сценарий можно запустить из командной строки так:

```
$ python facebook_post_time_stats.py -f my_posts.jsonl
```

Сначала он создает список с временами создания сообщений. Функция `dateutil.parser.parse()` помогает преобразовать строку с датой в формате ISO 8601 в объект `datetime`, который затем вызовом функции `strftime()` преобразуется в строку в формате HH:MM:SS.

Затем полученный список используется для индексации объекта `Series` из библиотеки pandas, который первоначально является простой последовательностью из единиц. Затем числовой ряд пересчитывается, и в каждый его элемент записывается количество сообщений, опубликованных в течение данного часа. В результате у нас получается числовой ряд с 24 элементами, по одному для каждого часа суток, с числом сообщений, опубликованных в течение каждого из них. Заключительная часть сценария создает график числового ряда в виде простой столбчатой диаграммы с распределением сообщений в течение дня (рис. 4.5).

Нетрудно заметить пик публикаций между концом дня и вечером и примерно равномерное распределение в течение дня (с наименьшим количеством сообще-

ний в ночные и утренние часы). Справедливости ради следует отметить, что здесь не учитывается местоположение – время создания нормализовано по **всемирному координированному времени** (UTC), то есть часовой пояс пользователя не учитывается. Например, если опубликовать сообщение на восточном побережье США в 16:00, оно будет зарегистрировано с временем создания 21:00 UTC. Это вызвано тем, что **стандартное восточное время** (EST) эквивалентно UTC – 05:00, то есть отстает от UTC на пять часов по стандартному времени (осень/зима, когда действует зимнее время).

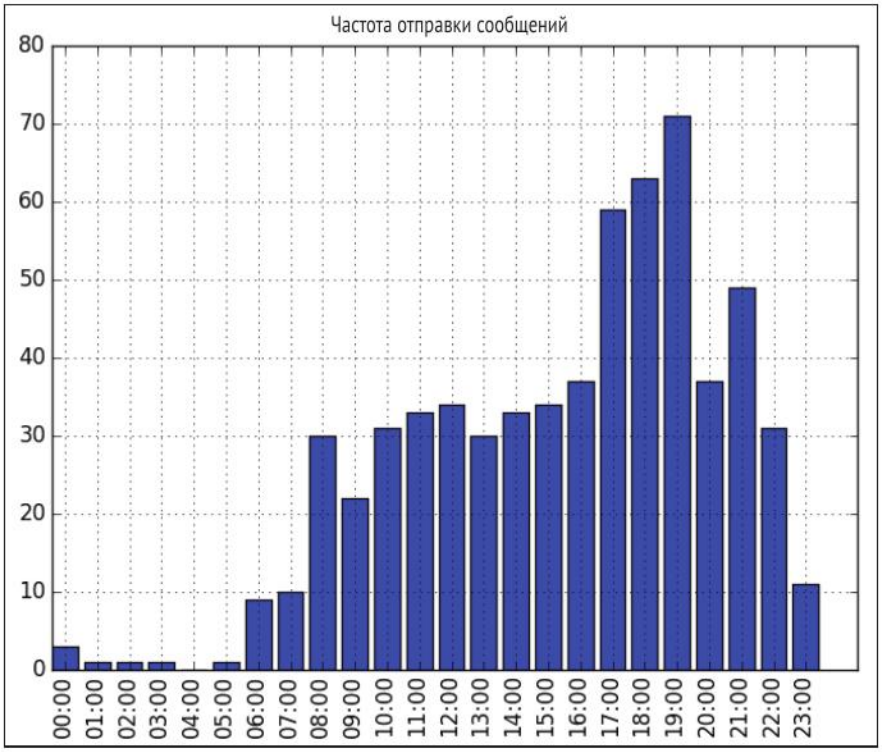

Рис. 4.5 ❖ Сообщения в частотно-временном измерении

Анализ страниц Facebook

Социальная сеть Facebook используется не только людьми, желающими поддерживать связь с друзьями и родственниками, но также компаниями и организациями, которые хотят с людьми сотрудничать. Страница Facebook создается и управляется на основе личной учетной записи в Facebook и может использоваться, чтобы:

- делиться информацией о бизнесе (например, о ресторане или онлайн-магазине);
- представлять известную личность или группу людей (например, футболиста или рок-группу);
- связываться с аудиторией (например, как продолжение сетевого сообщества).

В отличие от личных учетных записей, на страницах могут публиковаться общедоступные сообщения[1]. Обычные пользователи могут подписаться на получение обновлений страницы непосредственно в свои ленты новостей.

Например, страница издательства Packt Publishing в Facebook находится по адресу https://www.facebook.com/PacktPub и содержит общую информацию об издательстве PacktPub, а также сообщения, связанные с обычными и электронными книгами и учебными видеоруководствами, опубликованными издательством.

В документации с описанием API (https://developers.facebook.com/docs/graph-api/reference/page) можно увидеть, что для любой страницы доступна самая разнообразная информация. К наиболее интересным атрибутам страницы можно отнести следующие:

- `id`: числовой идентификатор для этой страницы;
- `name`: отображаемое имя страницы;
- `about`: текстовое описание страницы;
- `link`: ссылка на эту страницу;
- `website`: веб-сайт организации, если есть;
- `general_info`: текстовое поле с общей информацией о странице;
- `likes`: число пользователей, которым эта страница нравится.

Объект `Page` может быть связан с другими объектами. В документации описаны все имеющиеся возможности в терминах ребер графа (то есть связей с другими объектами). Например:

- `posts`: список сообщений, опубликованных страницей;
- `photos`: фотографии этой страницы;
- `albums`: список фотоальбомов, опубликованных этой страницей;
- `picture`: фотография профиля (аватар) этой страницы.

Определенные типы страниц могут также отображать дополнительную информацию в зависимости от типа деятельности (например, ресторан может опубликовать график работы, музыкальная группа – список участников и т. д.).

Следующий сценарий обращается к Facebook Graph API, чтобы получить основную информацию о конкретной странице Facebook:

```
# Chap04/facebook_get_page_info.py
import os
import json
import facebook
from argparse import ArgumentParser

def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--page')
    return parser

if __name__ == '__main__':
    parser = get_parser()
    args = parser.parse_args()
```

[1] В качестве уточнения: страницы предназначены для групп, компаний, организаций и общественных деятелей, которые желают создать свое присутствие в Facebook, а профили представляют отдельных людей. – *Прим. перев.*

```
    token = os.environ.get('FACEBOOK_TEMP_TOKEN')
    fields = [
        'id',
        'name',
        'about',
        'likes',
        'website',
        'link'
    ]
    fields = ','.join(fields)

    graph = facebook.GraphAPI(token)
    page = graph.get_object(args.page, fields=fields)

    print(json.dumps(page, indent=4))
```

Для чтения аргумента командной строки с именем (или идентификатором) страницы в данном сценарии используется экземпляр класса `ArgumentParser`:

```
$ python facebook_get_page_info.py --page PacktPub
```

Результат выполнения этой команды:

```
{
    "id": "204603129458",
    "name": "Packt",
    "about": "Packt provides software learning resources, from eBooks
              to video courses, to everyone from web developers to
              data scientists.",
    "likes": 24672,
    "website": "https://www.packtpub.com",
    "link": "https://www.facebook.com/PacktPub/"
}
```

Чтобы получить краткий обзор всех имеющихся полей, можно также воспользоваться проводником Facebook Graph API Explorer.

Получение сообщений со страницы

Обсудив способы получения базовой информации о странице, рассмотрим теперь процедуру загрузки сообщений, опубликованных на странице, и их сохранения в обычном формате JSON Lines для дальнейшего анализа.

Данная процедура подобна той, что использовалась для загрузки сообщений, опубликованных пользователем, но в данном случае доступна дополнительная информация для вычисления степени вовлеченности пользователей:

```
# Chap04/facebook_get_page_posts.py
import os
import json
from argparse import ArgumentParser
import facebook
import requests

def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--page')
```

```
    parser.add_argument('--n', default=100, type=int)
    return parser

if __name__ == '__main__':
    parser = get_parser()
    args = parser.parse_args()

    token = os.environ.get('FACEBOOK_TEMP_TOKEN')

    graph = facebook.GraphAPI(token)
    all_fields = [
        'id',
        'message',
        'created_time',
        'shares',
        'likes.summary(true)',
        'comments.summary(true)'
    ]
    all_fields = ','.join(all_fields)
    posts = graph.get_connections(args.page, 'posts', fields=all_fields)

    downloaded = 0
    while True:  # продолжить постраничную загрузку
        if downloaded >= args.n:
            break
        try:
            fname = "posts_{}.jsonl".format(args.page)
            with open(fname, 'a') as f:
                for post in posts['data']:
                    downloaded += 1
                    f.write(json.dumps(post)+"\n")
                # получить следующую страницу
                posts = requests.get(posts['paging']['next']).json()
        except KeyError:
            # страниц больше нет, прервать цикл
            break
```

Для чтения аргументов командной строки с именем (или идентификатором) страницы, а также числом загружаемых сообщений в данном сценарии используется экземпляр класса `ArgumentParser`. Число сообщений можно опустить (по умолчанию будет загружено 100 сообщений). По аналогии с загрузкой сообщений пользователя сценарий определяет список полей для включения в результат. В частности, поскольку данные предполагается использовать для анализа степени вовлеченности пользователей, нам нужна информация о количестве цитирований, отметок «Нравится» и комментариев. С этой целью добавляются поля `shares`, `likes.summary(true)` и `comments.summary(true)`. Для получения количества отметок «Нравится» и комментариев необходим дополнительный атрибут `summary(true)`, чтобы включить сводные статистики, то есть суммы.

Загрузка выполняется в цикле `while True`, по аналогии с загрузкой сообщений пользователя. Разница лишь в дополнительном счетчике `downloaded`, который увеличивается с каждым загруженным сообщением. Он используется для установки предела числа загружаемых сообщений. Это делается главным образом потому, что на страницах часто публикуется огромный объем информации.

Данный сценарий можно запустить так:

```
$ python facebook_get_page_posts.py --page PacktPub --n 500
```

Эта команда выполнит запрос к Facebook Graph API и создаст файл `posts_PacktPub.jsonl` с 500 сообщениями, по одному в строке. Далее приводится пример одного сообщения (то есть одной строки из файла `.jsonl`):

```
{
  "id": "ид-сообщения",
  "created_time": "дата в формате ISO 8601",
  "message": "Текст сообщения",
  "comments": {
    "data": [ /* список комментариев */ ],
    "paging": {
      "cursors": {
        "after": "идентификатор-курсора",
        "before": "идентификатор-курсора"
      }
    },
    "summary": {
      "can_comment": true,
      "order": "ranked",
      "total_count": 4
    }
  },
  "likes": {  /* отметки "Нравится" */
    "data": [ /* список пользователей */ ],
    "paging": {
      "cursors": {
        "after": "идентификатор-курсора",
        "before": "идентификатор-курсора"
      }
    },
    "summary": {
      "can_like": true,
      "has_liked": false,
      "total_count": 10
    }
  },
  "shares": { /* цитирования */
    "count": 9
  }
}
```

Как видите, сообщение – сложный объект, включающий разнообразную информацию. Степень вовлеченности пользователей измеряют поля `shares`, `likes` и `comments`. Поле `shares` сообщает общее количество пользователей, которые поделились информацией. Другие подробности не включены из-за настроек конфиденциальности. Когда пользователь делится информацией, он фактически создает собственное сообщение, поэтому новое сообщение не должно быть видно другим пользователям вне их сети. Поля `comments` и `likes`, с другой стороны, – это

объекты, связанные с самим сообщением, поэтому для них доступна более подробная информация.

Ключ `data` поля `comments` содержит список объектов с информацией, связанной с комментарием. Например, вот как выглядит каждый комментарий:

```
{
  "created_time": "date in ISO 8601 format",
  "from": {
    "id": "идентификатор-пользователя",
    "name": "имя-пользователя"
  },
  "id": "идентификатор-комментария",
  "message": "текст сообщения"
}
```

Объект `comments` также включает поле `paging` со ссылками на курсоры, если число комментариев превышает одну страницу. При наличии в запросе прямой ссылки на атрибут `summary(true)` также включается краткая сводная статистика. В частности, нас интересует общее количество `total_count`.

Объект `likes` имеет некоторое сходство с объектом `comments`, хотя в этом случае он не такой сложный. Если быть точным, он содержит список идентификаторов пользователей, которым понравилось сообщение. Также имеется сводная статистика с общим количеством отметок «Нравится», так как в запросе был указан атрибут `summary(true)`.

После загрузки данных можно приступать к их анализу.

Реакции Facebook и Graph API 2.6

Вскоре после того, как был написан черновик настоящей главы, компания Facebook реализовала новую функцию под названием **Reactions** (Реакции). Появившись как расширение кнопки Like (Нравится), функция «Реакции» позволяет пользователям выразить свои чувства к конкретному сообщению, выходя за пределы простой отметки «Нравится». Пользователи теперь могут выражать такие новые чувства, как Love (Супер), Haha (Ха-ха), Wow (Ух ты!), Sad (Сочувствую), Angry (Возмутительно) и, разумеется, Like (Нравится). На рис. 4.6 показано, в каком виде отображаются новые кнопки:

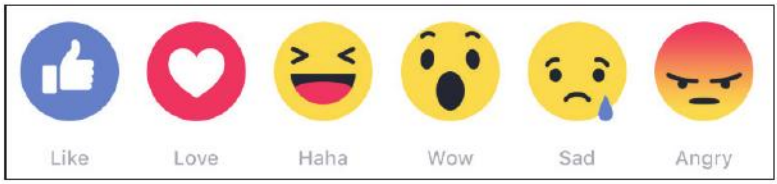

Рис. 4.6 ❖ Реакции в Facebook

Поддержка Facebook Reactions впервые была включена в версию Graph API 2.6. Примеры в этой главе главным образом основаны на версии 2.5, поэтому если вы планируете включить анализ реакций, необходимо указать правильную версию. Информацию о том, как получить доступ к данным, связанным с реакциями,

можно найти в официальной документации (https://developers.facebook.com/docs/graph-api/reference/post/reactions).

С точки зрения разработчика реакции очень похожи на отметку «Нравится». Если запросить информацию о реакциях, структура документа, возвращаемого для каждого сообщения, будет выглядеть так:

```
{
  "message": "Информационный материал сообщения",
  "created_time": "дата создания в формате ISO 8601",
  "id": "идентификатор-сообщения",
  "reactions": {
    "data": [
      {
        "id": "идентификатор-пользователя",
        "name": "John Doe",
        "type": "WOW"
      },
      {
        "id": "еще-один-идентификатор-пользователя",
        "name": "Jane Doe",
        "type": "LIKE"
      },
      /* еще реакции... */
    ]
  },
  "likes": {
    "data": /* данные об отметках "Нравится", как и ранее */
  }
}
```

Примеры анализа степени вовлеченности пользователей в данной главе основаны на отметках «Нравится» и комментариях, но их можно расширить, задействовав эту новую возможность, которая, как предполагается, поможет публикаторам лучше понять, как пользователи и друзья относятся к их публикациям.

Измерение степени вовлеченности

Приступим к анализу сообщений, опубликованных страницей Facebook, начав с небольшого обсуждения, связанного с вовлеченностью пользователей. Данные, предоставляемые интерфейсом Graph API, в частности, содержат следующую информацию:

- число цитирований, то есть сколько раз пользователи поделились сообщением;
- число пользователей, которым сообщение понравилось;
- число пользователей, которые отреагировали на сообщение;
- число комментариев к сообщению.

Цитирование, выставление отметки «Нравится» и комментирование – все это фактически действия, которые пользователь может выполнять в отношении сведений, опубликованных кем-то еще. Трудно понять истинные причины, лежащие в основе определенного действия (например, как отличить сарказм от неподдельной реакции?), а глубокий анализ психологии, лежащий в основе таких действий,

выходит за рамки настоящей книги, поэтому в интересах примеров анализа данных мы предположим, что чем больше ответных реакций связано с сообщением, тем оно «*успешнее*». Термин «*успешный*» взят в кавычки, потому что мы не определили точного значения слова *успех*, поэтому в данном разделе мы просто будем подсчитывать количество ответных реакций пользователей на страницу.

Следующий сценарий выводит информацию о сообщении, опубликованном на странице Facebook, которая получила самое большое количество ответных реакций:

```
# Chap04/facebook_top_posts.py
import json
from argparse import ArgumentParser

def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--page')
    return parser

if __name__ == '__main__':
    parser = get_parser()
    args = parser.parse_args()

    fname = "posts_{}.jsonl".format(args.page)

    all_posts = []
    with open(fname) as f:
        for line in f:
            post = json.loads(line)
            n_likes = post['likes']['summary']['total_count']
            n_comments = post['comments']['summary']['total_count']
            try:
                n_shares = post['shares']['count']
            except KeyError:
                n_shares = 0
            post['all_interactions'] = n_likes + n_shares + n_comments
            all_posts.append(post)

    most_liked_all = sorted(all_posts,
                            key=lambda x: x['all_interactions'],
                            reverse=True)
    most_liked = most_liked_all[0]
    message = most_liked.get('message', '-empty-')
    created_at = most_liked['created_time']
    n_likes = most_liked['likes']['summary']['total_count']
    n_comments = most_liked['comments']['summary']['total_count']
    print("Сообщение с максимальным числом взаимодействий:")
    print("Сообщение: {}".format(message))
    print("Время создания: {}".format(created_at))
    print("Понравилось: {}".format(n_likes))
    print("Комментировали: {}".format(n_comments))
    try:
        n_shares = most_liked['shares']['count']
        print("Поделились: {}".format(n_shares))
    except KeyError:
        pass
    print("Общее количество: {}".format(most_liked['all_interactions']))
```

Для чтения аргументов командной строки в этот сценарий использует экземпляр класса `ArgumentParser` и запускается так:

```
$ python facebook_top_posts.py --page PacktPub
```

В процессе обхода сообщений в каждое добавляется ключ `all_interactions` (все взаимодействия) с суммой отметок «Нравится», цитирований и комментариев. Если ни один пользователь пока еще не поделился сообщением, то есть не цитировал его, ключа `shares` в словаре не окажется, поэтому обращение к `post['shares']['count']` заключено в блок `try/except`, который в отсутствие ключа устанавливает число цитирований в значение по умолчанию 0.

Новый ключ `all_interactions` используется в качестве параметра сортировки в функции `sorted()`, которая возвращает список сообщений, отсортированный по числу ответных реакций в порядке убывания.

Заключительная часть сценария просто выводит сводную информацию:

```
Сообщение с максимальным числом взаимодействий:
Сообщение: It's back! Our $5 sale returns!
Время создания: 2016-12-17T11:51:00+0000
Понравилось: 10
Комментировали: 4
Поделились: 9
Общее количество: 23
```

Поиск сообщения с самым большим количеством ответных реакций – довольно интересное упражнение, но оно почти ничего не говорит об общей картине.

Наш следующий шаг: выявление наиболее успешного времени суток, то есть времени, когда публикация сообщения набирает наибольшее число ответных реакций.

Следующий сценарий использует объект `pandas.DataFrame` для определения статистик ответных реакций и строит график результатов, используя почасовые дискретные интервалы, как мы делали это ранее для пользователя:

```
# Chap04/facebook_top_posts_plot.py
import json
from argparse import ArgumentParser
import numpy as np
import pandas as pd
import dateutil.parser
import matplotlib.pyplot as plt
from datetime import datetime

def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--page')
    return parser

if __name__ == '__main__':
    parser = get_parser()
    args = parser.parse_args()

    fname = "posts_{}.jsonl".format(args.page)

    all_posts = []
```

```
    n_likes = []
    n_shares = []
    n_comments = []
    n_all = []
    with open(fname) as f:
        for line in f:
            post = json.loads(line)
            created_time = dateutil.parser.parse(post['created_time'])
            n_likes.append(post['likes']['summary']['total_count'])
            n_comments.append(post['comments']['summary']['total_count'])
            try:
                n_shares.append(post['shares']['count'])
            except KeyError:
                n_shares.append(0)
            n_all.append(n_likes[-1] + n_shares[-1] + n_comments[-1])
            all_posts.append(created_time.strftime('%H:%M:%S'))

        idx = pd.DatetimeIndex(all_posts)
        data = {
            'likes': n_likes,
            'comments': n_comments,
            'shares': n_shares,
            'all': n_all
        }
        my_series = pd.DataFrame(data=data, index=idx)

        # Передискретизация в почасовые дискретные интервалы
        per_hour = my_series.resample('1h').sum().fillna(0)

        # Построение графика
        fig, ax = plt.subplots()
        ax.grid(True)
        ax.set_title("Частоты взаимодействий")
        width = 0.8
        ind = np.arange(len(per_hour['all']))
        plt.bar(ind, per_hour['all'])
        tick_pos = ind + width / 2
        labels = []
        for i in range(24):
            d = datetime.now().replace(hour=i, minute=0)
            labels.append(d.strftime('%H:%M'))
        plt.xticks(tick_pos, labels, rotation=90)
        plt.savefig('interactions_per_hour.png')
```

Данный сценарий можно запустить следующей командой:

```
$ python facebook_top_posts_plot.py --page PacktPub
```

В результате будет получена диаграмма в файле `interactions_per_hour.png`, которую можно увидеть на рис. 4.7.

Сценарий читает файл `.jsonl` в цикле, собирая для каждого сообщения списки с разными статистиками: число отметок «Нравится», число цитирований, число комментариев и их сумму. Каждый из этих списков соответствует столбцу в таблице данных, которая индексируется по времени создания (только по времени, не

по дате). Используя прием передискретизации, уже применявшийся ранее, сценарий агрегирует все сообщения, опубликованные внутри часовых дискретных интервалов, и суммирует их частоты. Для целей данного упражнения рассматривается только общая сумма ответных реакций, но вообще можно построить отдельные графики для индивидуальных статистических показателей.

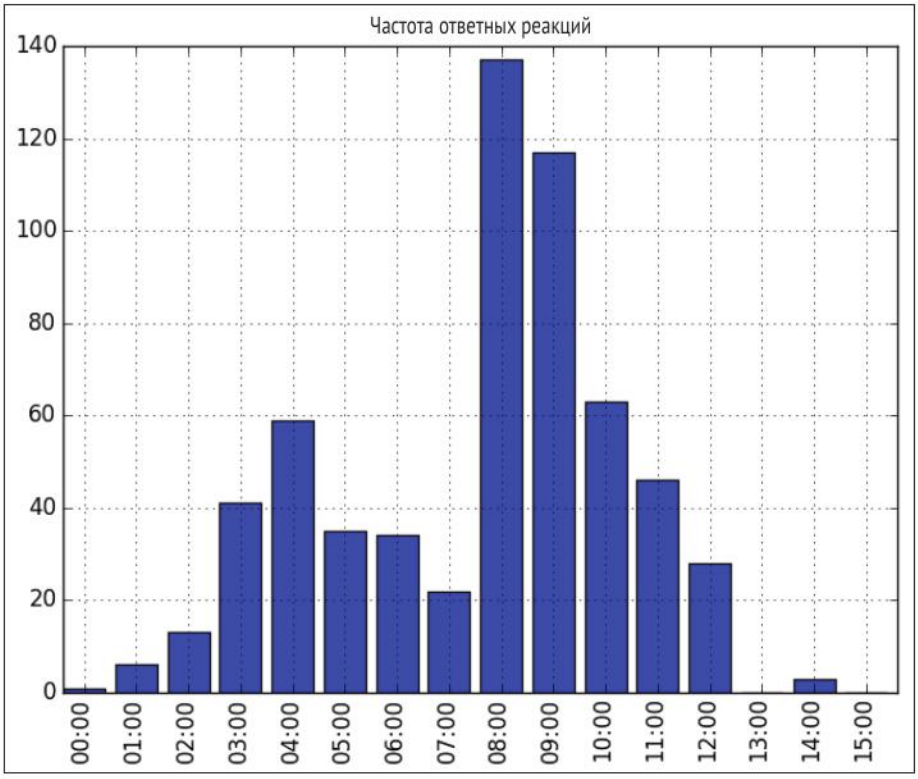

Рис. 4.7 ❖ Частоты ответных реакций (сумма по часам)

Данный график показывает, что наибольшее количество ответных реакций соответствует интервалам с метками **08:00** и **09:00**, то есть сообщения, опубликованные в период с 8:00 до 10:00, получили самое большое количество ответных реакций.

Исследовав набор данных, мы заметили, что временной интервал с 8:00 до 10:00 также характерен тем, что в это время публикуется большинство сообщений. Если бы мы решили построить график частоты сообщений, в этом случае мы наблюдали бы распределение, аналогичное показанному на рис. 4.7 (эту задачу мы оставляем в качестве упражнения для заинтересованных читателей).

В данном случае главная проблема заключается в режиме агрегирования, то есть в методе передискретизации `resample`. Мы используем метод `sum()`, и, по всей видимости, дискретные интервалы с 8:00 до 10:00 получили наибольшее количество ответных реакций, просто потому что на этот период приходится наибольшее количество публикуемых сообщений, поэтому полная сумма в действитель-

ности не говорит, являются ли эти сообщения наиболее успешными. Эту проблему легко решить: вместо метода `sum()` можно применить метод `mean()`, возвращающий среднее значение, и запустить сценарий еще раз. Полученный результат показан на рис. 4.8.

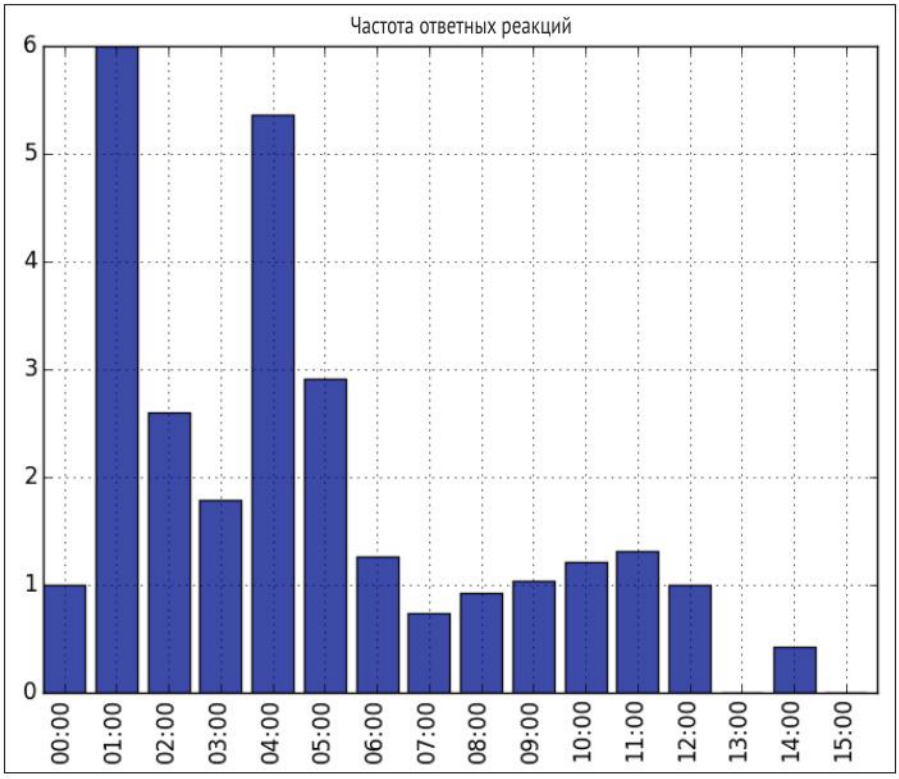

Рис. 4.8 ❖ Частота ответных реакций (усреднение по часам)

На рис. 4.8 показано распределение среднего числа ответных реакций на каждый час. На данном графике временной интервал с 8:00 по 10:00 выглядит не столь успешным, как прежде. Напротив, наибольшее количество ответных реакций приходится на интервалы 1:00 и 5:00.

Это простое упражнение показало, что агрегирование данных разными способами может дать разные результаты в ответ на один и тот же вопрос. Углубляясь в обсуждение данной темы, нужно отметить, что в данном случае мы не располагаем двумя важными элементами информации. Во-первых, у нас нет исторической информации о количестве подписчиков на страницу PacktPub, то есть сколько людей в действительности видит сообщения в своей ленте новостей. Если большое количество людей оформило подписку недавно, они вряд ли реагировали на более старые сообщения, поэтому статистика может быть смещена в пользу более новых сообщений. Во-вторых, у нас нет информации о демографии и в особенности о географическом положении пользователей: временной интервал с 1:00 по 5:00 выглядит немного странным, но, как отмечалось ранее, времена создания

нормализованы по часовому поясу UTC. Другими словами, интервалы 1:00 и 5:00 UTC соответствуют, например, утру в Индии или предвечернему времени/вечеру на Тихоокеанском побережье.

При наличии таких демографических данных появляется также возможность выяснить лучшее время для публикации нового сообщения. Например, если большинство наших пользователей расположено на Тихоокеанском побережье и предпочитает реагировать на новости по вечерам (согласно их часовому поясу), стоит запланировать публикацию сообщения на 1:00 UTC.

Визуализация сообщений в виде облака слов

После анализа ответных реакций снова переключим наше внимание на содержание сообщений.

Облака слов, или так называемые облака дескрипторов, или тегов (https://ru.wikipedia.org/wiki/Облако_тегов), – это визуальные представления текстовых данных. Важность каждого слова в облаке обычно представляется его размером.

В этом разделе мы воспользуемся библиотекой **wordcloud**, которая позволяет легко и просто создавать облака слов. Прежде всего необходимо установить данную библиотеку и библиотеку обработки изображений, от которой она зависит, выполнив следующие команды:

```
$ pip install wordcloud
$ pip install Pillow
```

Библиотека **Pillow** – это ответвление старого проекта графической библиотеки **Python Imaging Library (PIL)**, поскольку проект PIL прекратил развитие. Кроме всего прочего, библиотека Pillow поддерживает Python 3, поэтому после короткой установки она готова к работе.

Следующий сценарий читает файл `.jsonl`, подобный файлу с сообщениями от издательства PacktPub, и создает файл `.png` с облаком слов:

```
# Chap04/facebook_posts_wordcloud.py
import json
from argparse import ArgumentParser
import matplotlib.pyplot as plt
from nltk.corpus import stopwords
from wordcloud import WordCloud

def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--page')
    return parser

if __name__ == '__main__':
    parser = get_parser()
    args = parser.parse_args()

    fname = "posts_{}.jsonl".format(args.page)

    all_posts = []
    with open(fname) as f:
        for line in f:
            post = json.loads(line)
```

```
        all_posts.append(post.get('message', ''))
text = ' '.join(all_posts)
stop_list = ['save', 'free', 'today',
             'get', 'title', 'titles', 'bit', 'ly']
stop_list.extend(stopwords.words('english'))
wordcloud = WordCloud(stopwords=stop_list).generate(text)
plt.imshow(wordcloud)
plt.axis("off")
plt.savefig('wordcloud_{}.png'.format(args.page))
```

Для чтения аргумента командной строки с именем или идентификатором страницы этот сценарий, как обычно, использует экземпляр класса `ArgumentParser`.

Он создает список `all_posts`, включающий текст всех сообщений. Вместо прямого обращения к словарю он использует метод `post.get('message', '')`, поскольку ключ `message` может отсутствовать в сообщениях (к примеру, в случае изображения без комментариев), хотя такое случается довольно редко.

Затем список сообщений собирается в одну строку `text`, которая послужит основой для создания облака слов. Объект `WordCloud` принимает несколько необязательных аргументов, определяющих некоторые аспекты облака слов. В частности, в примере используется аргумент `stopwords` со списком слов, которые не должны попасть в облако. В этот список включены стандартные английские стоп-слова, как определено в библиотеке **NLTK** (Natural Language Toolkit, инструменты для обработки естественного языка), а также несколько специальных ключевых слов, которые часто используются в учетных записях PacktPub, но на самом деле не несут особого смысла (например, ссылки на `bit.ly` и упоминания о предложениях по поводу конкретных заголовков книг).

На рис. 4.9 изображен пример получившегося результата.

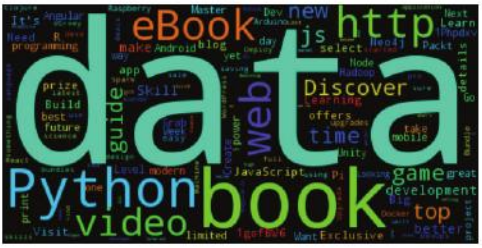

Рис. 4.9 ❖ Пример облака слов

Здесь можно заметить некоторые продукты, предлагаемые издательством Packt Publishing (печатные и электронные книги и видео), а также основные технологии, освещаемые в их публикациях (такие как Python, JavaScript, R, Arduino, разработка компьютерных игр и т. д.), где главным ключевым словом является слово **данные**.

Резюме

В настоящей главе были представлены некоторые приемы анализа данных из Facebook, одного из крупнейших игроков, если не главного, на арене социальных медиа.

После обсуждения некоторых аспектов графового программного интерфейса Facebook Graph API и его эволюции, а также следствий для анализа данных мы создали приложение Facebook, которым воспользовались для взаимодействия с платформой Facebook.

Представленные в настоящей главе приложения для анализа данных связаны с профилем аутентифицированного пользователя и страницами Facebook. Мы узнали, как вычислять статистики, связанные с привычками пользователя размещать публикации и со взаимодействиями пользователей, которые следят за обновлениями заданной страницы Facebook. С привлечением простых методов агрегирования и визуализации этот вид анализа реализуется относительно небольшим объемом программного кода. Наконец, мы применили технику визуализации важных ключевых слов из набора опубликованных сообщений, которые, как мы надеемся, представляют важные темы, именуемую облаком слов.

В следующей главе мы обратим внимание на одну из новейших социальных сетей, разработанную компанией Google, – социальную сеть Google+.

Глава 5

Тематический анализ в Google+

Настоящая глава целиком посвящена одному из крупнейших игроков, появившихся на арене социальных сетей совсем недавно, – социальной сети Google+ (или просто G+). Основанная в 2011 г., она характеризуется как «*социальный слой с охватом всех служб Google*» (https://ru.wikipedia.org/wiki/Google%2B). Данная социальная сеть испытала чрезвычайно быстрый рост, достигнув 10 миллионов пользователей за первые две недели работы. После многих модернизаций к концу 2015 г. компания Google обнародовала большую заинтересованность в сообществах и коллекциях, переместив обслуживание в сторону организации групп по интересам.

Эта глава охватывает следующие темы:

- как взаимодействовать с Google+ API из Python;
- как искать людей или страницы в Google+;
- как использовать веб-платформу **Flask** для визуализации результатов поиска;
- как обрабатывать сообщения пользователя с целью извлечения интересующих ключевых слов.

Начало работы с Google+ API

Google+ API – это программный интерфейс социальной сети Google+, который может использоваться для интеграции вашего приложения или веб-сайта с социальной сетью Google+, по аналогии с обсуждавшимися выше социальными сетями Twitter и Facebook. В данном разделе рассматриваются процесс регистрации приложения и начало работы с Google+ API.

Перед началом нужно создать учетную запись Google (https://www.google.com/accounts), новым, незарегистрированным пользователям Google предлагает несколько служб (к примеру, Gmail, Blogger и другие), но вообще учетные записи ведутся централизованно. То есть если вы являетесь пользователем одной из этих служб, ваша учетная запись будет быстро настроена для Google+.

После регистрации и входа откройте страницу Диспетчера API (Google Developers Console, https://console.developers.google.com/start). На этой странице можно создать наш первый проект. На рис. 5.1 изображен диалог создания проекта, от нас требуется только указать название:

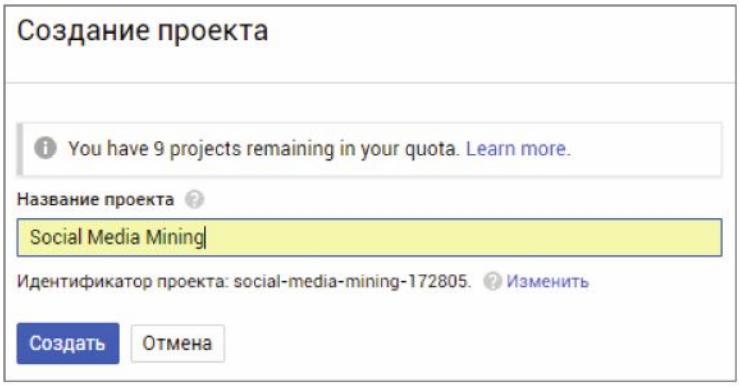

Рис. 5.1 ❖ Создание проекта в Диспетчере API

После создания проекта необходимо включить Google+ API. В инструментальной панели проекта имеется компонент Use Google APIs (Google+ API Включить), который позволяет управлять доступом к API, создавать новые учетные данные и прочее. Подобно тому, как одиночная учетная запись Google может использоваться для доступа к нескольким службам Google, одиночный проект может использоваться несколькими API, если они включены. Отыщите ссылку Google+ API в группе Social API и включите его одним щелчком. На рис. 5.2 изображено содержимое раздела Overview (Обзор) на этой странице.

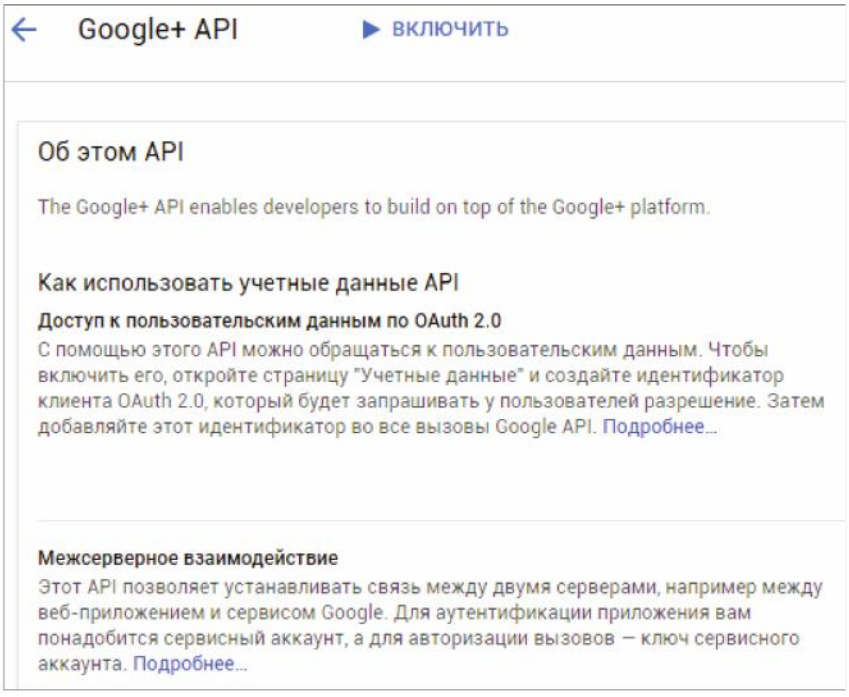

Рис. 5.2 ❖ Включение Google+ API для проекта

Сразу после включения API система напомнит, что для использования программного интерфейса требуются учетные данные, как показано на рис. 5.3.

Рис. 5.3 ❖ После включения API необходимо настроить учетные данные

Слева в боковом меню имеется пункт Credentials (Учетные данные). Поддерживаются три вида учетных данных: API key (ключ API), OAuth client ID (идентификатор клиента OAuth) и Service account key (ключ сервисного аккаунта, последний необходим только для организации взаимодействий сервер–сервер посредством облачного интерфейса Google Cloud API).

Ключ API требуется для простого доступа к API, то есть вызов API не получает доступа ни к каким частным данным пользователя. Этот ключ активирует аутентификацию прикладного уровня, который главным образом используется для измерения показателей использования проекта в целях отчетности (более подробно об ограничениях на частоту запросов можно узнать в последующих разделах).

Идентификатор клиента OAuth требуется для авторизованного доступа к API, то есть вызова API, который действительно получает доступ к частным данным пользователя. Перед вызовом пользователь, имеющий доступ к частным данным, должен явно предоставить доступ вашему приложению. Разные программные интерфейсы Google объявляют различные области, то есть набор разрешенных операций, которые должны быть одобрены пользователем.

Для тех, кто не уверен в выборе типа учетных данных, инструментальная панель также предлагает вариант Help me choose (Мастер создания учетных записей, как показано на рис. 5.4). Этот мастер проведет вас через процедуру создания, задав несколько простых вопросов и разъяснив уровень полномочий, необходимый вашему приложению.

После настройки проекта и получения ключей доступа можно заняться вопросом получения доступа к Google+ API программными средствами.

Компания Google предлагает своим пользователям официальную клиентскую библиотеку для Python (https://developers.google.com/api-client-library/python/apis/plus/v1), которую можно установить, используя обычную процедуру:

```
$ pip install google-api-python-client
```

Данный клиент будет доступен как пакет googleapiclient, который обычно импортируется с псевдонимом apiclient. В следующем разделе приводится первый рабочий пример, который тестирует использование API.

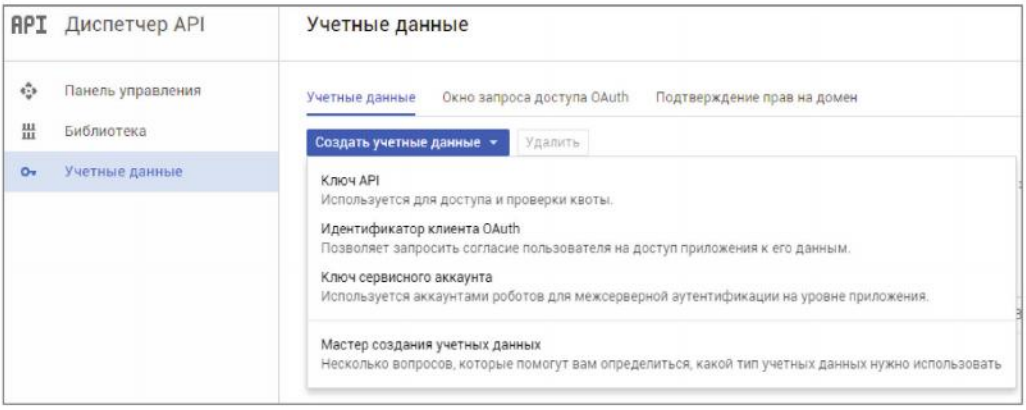

Рис. 5.4 ❖ Раскрывающее меню выбора типа учетных данных для доступа к API

Поиск в Google+

Первый пример, представленный в сценарии `gplus_search_example.py`, обращается к Google+ API в поисках людей или страниц.

Этот пример предполагает, что вы уже настроили ключ API для простого доступа в разделе Credentials (Учетные данные) инструментальной панели проекта – к персональным данным мы пока не обращаемся. Так же, как в примерах доступа к Twitter и Facebook, мы сохраним учетные данные в переменных окружения. Например, в командной оболочке Bash можно выполнить следующую команду:

```
$ export GOOGLE_API_KEY="вставьте-ваш-ключ-api-сюда"
```

При необходимости команду `export` можно также включить в сценарий инициализации оболочки:

```
# Chap05/gplus_search_example.py
import os
import json
from argparse import ArgumentParser
from apiclient.discovery import build

def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--query', nargs='*')
    return parser

if __name__ == '__main__':
    api_key = os.environ.get('GOOGLE_API_KEY')
    parser = get_parser()
    args = parser.parse_args()

    service = build('plus',
                    'v1',
                    developerKey=api_key)

    people_feed = service.people()
    search_query = people_feed.search(query=args.query)
    search_results = search_query.execute()

    print(json.dumps(search_results, indent=4))
```

Для чтения строки запроса из аргумента командной строки сценарий использует экземпляр класса `ArgumentParser`. Например, чтобы выполнить поиск по слову «packt», запустите сценарий следующей командой:

```
$ python gplus_search_example.py --query packt
```

Аргумент `--query` в парсере определен с атрибутом `nargs='*'`, благодаря чему в нем можно передать несколько слов для поиска.

Перед использованием программных интерфейсов Google требуется создать объект `service` вызовом функции `build()`. Из клиента Google+ API мы импортируем компоновщик службы, который в качестве обязательных аргументов принимает название службы, с которой предполагается взаимодействовать (`plus`), версию ее API (`v1`) и ключ разработчика `developerKey`, то есть ключ API, который мы получили ранее.

Список поддерживаемых API с их последними версиями приводится в документации: https://developers.google.com/api-client-library/python/apis/.

В общем случае компоновщик службы используется следующим образом:

```
from apiclient.discovery import build
service = build('имя_api', 'версия_api', [дополнительные аргументы])
```

После создания объекта `service` его можно использовать для доступа к различным ресурсам (сгруппированным в коллекции). В данном примере мы запрашиваем коллекцию `people`, то есть людей.

Сконструировав и выполнив поисковый запрос, данный сценарий просто выводит результат на экран в виде документа JSON, чтобы мы имели представление о его формате:

```
{
  "nextPageToken": "строковое-значение-маркера",
  "selfLink": "ссылка-на-этот-запрос",
  "title": "Результаты поиска людей в Google+",
  "etag": "строковое-значение-идентификатора-etag",
  "kind": "plus#peopleFeed",
  "items": [
    {
      "kind": "plus#person",
      "objectType": "страница",
      "image": {
        "url": "url-адрес-изображения"
      },
      "id": "112328881995125817822",
      "etag": "строковое-значение-идентификатора-etag",
      "displayName": "Издательство Packt Publishing",
      "url": "https://plus.google.com/+packtpublishing"
    },
    {
      /* еще элементы ... */
    }
  ]
}
```

Приведенный выше результат упрощен для краткости, но общая структура должна быть понятна. Атрибуты первого уровня (например, `title`, `selfLink` и прочие) определяют некоторые особенности набора результатов. Собственно результаты поиска возвращаются в списке `items`. Каждый элемент списка однозначно определяется атрибутом `id` и содержит также адрес `URL`. Показанные в данном примере результаты могут содержать смесь страниц и людей, как определено в атрибуте `objectType`. Атрибут `displayName` выбирается пользователем (или администратором страниц) и в нормальных условиях отображается на соответствующей странице G+.

Вывод результатов поиска в веб-интерфейсе

В данном разделе мы дополним первый пример визуализацией результатов поиска. Для этого реализуем автоматическое создание веб-страницы с изображениями профилей (аватаров) и именами людей или названиями страниц, используя знакомый интерфейс. Это поможет устранить неоднозначность между разными результатами, чьи отображаемые имена совпадают.

Чтобы достичь этой цели, используем микрофреймворк для веб-разработки с названием **Flask** (http://flask.pocoo.org/), позволяющий создавать веб-интерфейсы в короткие сроки.

Вообще говоря, язык Python и веб-разработка идут рука об руку, и уже много лет для Python существует несколько веб-библиотек, достигнувших высокого уровня зрелости. Микрофреймворк Flask, по сравнению с другими веб-фреймворками, относительно молод, но достиг высокого уровня зрелости и был принят сообществом. Поскольку веб-разработка не является ключевой темой настоящей книги, мы применим микрофреймворк Flask из-за его *микро*характера – он не задает лишних вопросов по поводу структуры приложения и позволяет быстрее запустить проект с относительно небольшим количеством программного кода.

Перечислим некоторые возможности, включенные в микрофреймворк Flask:

- сервер разработки и отладчик;
- интегрированная поддержка модульного тестирования;
- поддержка обработки шаблонов с использованием библиотеки Jinja2;
- поддержка Юникода;
- большое количество расширений для различных вариантов использования.

Процесс установки выглядит как обычно:

```
$ pip install flask
```

Эта команда установит микрофреймворк и все его зависимости.

Для читателей, желающих расширить свои познания, издательство Packt Publishing выпустило множество книг, посвященных этому программному продукту, например *Learning Flask Framework* (Matt Copperwaite, Charles Leifer) или *Mastering Flask* (Jack Stouffer). В настоящей главе обсуждение микрофреймворка Flask будет сведено до уровня непосредственных примеров:

```
# Chap05/gplus_search_web_gui.py
import os
from flask import Flask
from flask import request
```

```
from flask import render_template
from apiclient.discovery import build

app = Flask(__name__)
api_key = os.environ.get('GOOGLE_API_KEY')

@app.route('/')
def index():
    return render_template('search_form.html')

@app.route('/search', methods=['POST'])
def search():
    query = request.form.get('query')
    if not query:
        # если нет запроса, показать сообщение об ошибке
        message = 'Введите поисковый запрос!'
        return render_template('search_form.html', message=message)
    else:
        # search
        service = build('plus',
                        'v1',
                        developerKey=api_key)

        people_feed = service.people()
        search_query = people_feed.search(query=query)
        search_results = search_query.execute()
        return render_template('search_results.html',
                               query=query,
                               results=search_results['items'])

if __name__ == '__main__':
    app.run(debug=True)
```

Сценарий `gplus_search_web_gui.py` реализует элементарное веб-приложение на основе микрофреймворка Flask, которое отображает простую форму для отправки запросов в Google+ API и выводит результаты на экран.

Данное приложение реализовано как экземпляр класса `Flask`, и его основным методом является метод `run()`, который запускает веб-сервер (в данном примере он выполняется в отладочном режиме, чтобы в случае необходимости упростить процесс отладки).

Поведение веб-приложения определяется его маршрутами, то есть, получив запрос с определенным URL, микрофреймворк передает его тому или иному фрагменту кода, который создаст ответ. В простейшей форме маршруты в микрофреймворке Flask представлены декорированными функциями.

Декораторы в Python

Пользоваться декораторами в Python чрезвычайно просто, но разобраться в них довольно сложно, если вы плохо знакомы с темой.

Декоратор – это просто функция, обертывает другую, целевую функцию, и служит для изменения поведения этой целевой функции. Изменение поведения происходит динамически и не требует ни изменения исходного кода целевой функции, ни создания подкласса. То есть любую функцию можно дополнить и усовершенствовать без усложнения существующей кодовой базы.

Вообще говоря, декораторы – это мощный и изящный инструмент, который может оказаться чрезвычайно полезным в самых разных ситуациях.

В примере выше функции `index()` и `search()` являются декорированными, и в обоих случаях декоратором является `app.route()`. Имя декоратора помещается непосредственно перед целевой функцией и предваряется символом `@`.

Маршруты и шаблоны Flask

Декоратор `app.route()` принимает первый аргумент с относительным URL-адресом ресурса, к которому мы хотим открыть доступ. Во втором аргументе принимается список поддерживаемых методов HTTP для конкретного URL-адреса. Если второй аргумент не задан, по умолчанию предполагается метод GET.

Функция `index()` выводит на экран титульную страницу с формой поиска, доступную по относительному URL-адресу `/`. Данная функция просто возвращает веб-страницу из шаблона `search_form.html`, используя для этого функцию `render_template()`.

Микрофреймворк Flask широко использует библиотеку шаблонов Jinja2 (http://jinja.pocoo.org/), с помощью которой функция `render_template()` читает разметку, хранящуюся в файле HTML, применяет директивы шаблонной обработки и возвращает итоговую страницу HTML.

Не вдаваясь в детали, которые лучше оставить заинтересованному читателю, чтобы он сверился с официальной документацией, отметим лишь, что цель использования библиотеки шаблонной обработки – дать возможность внедрять в веб-страницы специальные синтаксические конструкции, динамически генерирующие разметку HTML.

Ниже приводится содержимое файла `templates/search_form.html` с разметкой. Обратите внимание, что подкаталог `templates` находится в том же каталоге, что и файл на Python с приложением Flask, в нашем случае `gplus_search_web_gui.py`:

```
<html>
  <body>

  {% if message %}
  <p>{{ message }}</p>
  {% endif %}

  <form action="/search" method="post">
    Искать:
    <input type="text" name="query" />
    <input type="submit" name="submit" value="Выполнить!" />
    </form>
  </body>
</html>
```

Это простая страница (для краткости сильно упрощенная) с одной формой. Единственная директива шаблонной обработки на этой странице – блок `if` – проверяет наличие переменной `message` и отображает ее значение в тег абзаца, если оно определено.

Атрибут `action` формы настроен на относительный URL-адрес `/search`, а атрибут `method` имеет значение `POST`. Они соответствуют конфигурации функции `search()`, второй декорированной функции в `gplus_search_web_gui.py`.

Функция search() реализует взаимодействие с Google+ API. Прежде всего она ожидает, что с формой будет передан параметр query, доступный посредством словаря request.form.

Глобальный объект request используется в микрофреймворке Flask в основном для доступа к данным входящего запроса. Словарь request.form открывает доступ к данным, переданным с формой методом POST.

Если запрос отсутствует, функция search() вновь отобразит поисковую форму, включив в нее сообщение об ошибке. Функция render_template() принимает именованные аргументы и передает их в шаблон, в данном случае единственным аргументом является сообщение message.

Напротив, если запрос присутствует, функция search() отправит запрос в Google+ API и включит результаты в шаблон search_results.html. Содержимое файла templates/search_results.html приводится ниже:

```
<html>
  <link rel="stylesheet"
    href="{{ url_for('static', filename='style.css') }}" />
  <body>

  Выполняется поиск <strong>{{ query }}</strong>:

  {% for item in results %}
  <div class="{{ loop.cycle('row_odd', 'row_even') }}">
    <a href="{{ item.url }}">{{ item.displayName }}</a>
    ({{ item.objectType}})<br />
    <a href="{{ item.url }}">
      <img src="{{ item.image.url }}" />
    </a>
  </div>
  {% endfor %}

  <p><a href="/">Новый поиск</a></p>

  </body>
</html>
```

Основу тега body составляет цикл for, который выполняет обход списка результатов. Для каждого элемента в списке создается HTML-элемент с информацией из текущего элемента списка. Синтаксическая конструкция для вывода значения переменной имеет вид двойных фигурных скобок, например конструкция {{item.displayName}} добавит в шаблон отображаемое из элемента списка. Сама инструкция for управления потоком выполнения заключается в символы {% и %}.

В данном примере также использовано несколько интересных средств, которые позволяют легко интегрировать **каскадные таблицы стилей** (CSS), необходимые для оформления внешнего вида страницы. Функция loop.cycle() используется для циклического переключения между строковыми значениями из списка или переменных, и здесь она применяется, чтобы присвоить различные классы CSS блокам с результатами. Таким способом разные строки можно выделить разными (чередующимися) цветами. Еще одна функция – url_for() – используется для вставки URL-адреса конкретного ресурса. Папка static, аналогичная папке templates, должна находиться в одном каталоге с файлом gplus_search_web_gui.py,

который выполняет приложение Flask. Она используется для хранения статических файлов, таких как изображения или таблицы CSS. Файл `static/style.css` содержит определения CSS, используемые в странице с результатом поиска, и выглядит следующим образом:

```
.row_odd {
  background-color: #eaeaea;
}
.row_even {
  background-color: #fff;
}
```

Хотя тема веб-разработки с анализом данных напрямую не связана, она поможет легко создать прототип простого пользовательского интерфейса. Данная тема довольно обширная, и эта глава не предназначена для всестороннего ее обсуждения. Заинтересованному читателю рекомендуется заняться углубленным изучением подробностей веб-разработки и микрофреймворка Flask самостоятельно.

Данный пример можно запустить командой:

```
$ python gplus_search_web_gui.py
```

Она запустит приложение Flask, которое будет ожидать запроса HTTP, чтобы вернуть отклик в том виде, как определено в программном коде на Python. После запуска сценария в терминале должна появиться следующая строка:

```
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

`127.0.0.1` – это IP-адрес локального хоста, а `5000` – это номер порта, используемый по умолчанию микрофреймворком.

Доступность сервера Flask из сети. Если вы выполняете приложение Flask на виртуальной машине или машине в сети, вместо адреса `localhost` необходимо использовать соответствующий сетевой адрес данной машины.
Кроме того, необходимо удостовериться, что сервер видим извне, чтобы он был достижим через сеть. Поскольку это может иметь последствия для безопасности, сначала желательно подумать, доверяете ли вы пользователям своей сети.
Чтобы сделать приложение Flask видимым извне, можно отключить отладочный режим или привязать сервер к соответствующему адресу, например:

```
app.run(host='0.0.0.0')
```

Здесь `0.0.0.0` означает «все адреса на локальной машине».

Теперь можно запустить браузер и открыть страницу по адресу `http://127.0.0.1:5000`, как показано на рис. 5.5.

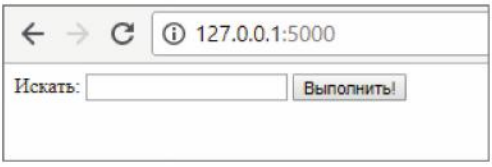

Рис. 5.5 ❖ Титульная страница приложения Flask

Если теперь щелкнуть на кнопке Искать!, оставив поле ввода пустым, приложение снова покажет форму с сообщением об ошибке, как показано на рис. 5.6.

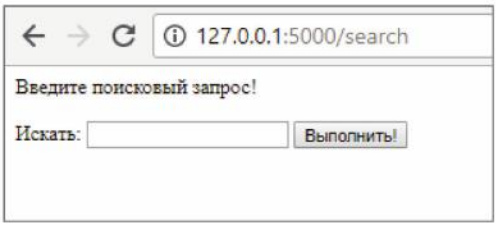

Рис. 5.6 ❖ Сообщение об ошибке в отсутствие текста запроса

Напротив, если поле ввода заполнено, будет показана страница с результатами, как показано на рис. 5.7 (на рисунке показаны результаты поиска по слову «packt»).

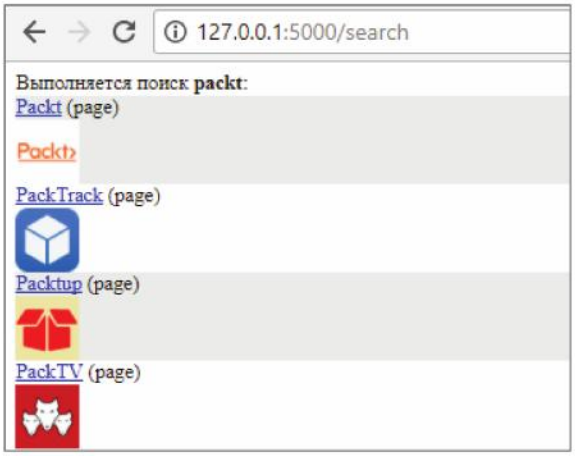

Рис. 5.7 ❖ Результаты поиска в Google+ API

Как видите, каждый элемент списка заключен в собственные блоки с чередующимся цветом фона и информацией о самом элементе. Названия элементов выводятся в якорных тегах `<a>`, что позволяет щелкать на них и переходить на соответствующие страницы Google+.

Заметки и действия со страницы Google+

Реализовав поиск страницы Google+ и отображение результатов в веб-интерфейсе, можно переходить к загрузке списка действий для той или иной страницы. Действия – это эквивалент Google+ для сообщений Facebook. По умолчанию действие рассматривается как заметка, то есть текст, которым делятся в Google+.

Следующий сценарий `gplus_get_page_activities.py` извлекает список действий для страницы Google+:

```
# Chap05/gplus_get_page_activities.py
import os
import json
from argparse import ArgumentParser
from apiclient.discovery import build

def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--page')
    parser.add_argument('--max-results', type=int, default=100)
    return parser

if __name__ == '__main__':
    api_key = os.environ.get('GOOGLE_API_KEY')
    parser = get_parser()
    args = parser.parse_args()

    service = build('plus',
                    'v1',
                    developerKey=api_key)

    activity_feed = service.activities()
    activity_query = activity_feed.list(
        collection='public',
        userId=args.page,
        maxResults='100'
    )

    fname = 'activities_{}.jsonl'.format(args.page)
    with open(fname, 'w') as f:
        retrieved_results = 0
        while activity_query and retrieved_results < args.max_results:
            activity_results = activity_query.execute()
            retrieved_results += len(activity_results['items'])
            for item in activity_results['items']:
                f.write(json.dumps(item)+"\n")

            activity_query = service.activities().list_next(activity_query,
                                                            activity_results)
```

Для чтения аргументов командной строки данный сценарий использует экземпляр класса `ArgumentParser`. Аргумент `--page` определяет идентификатор искомой страницы (или пользователя) и является обязательным. Он может содержать числовой идентификатор страницы или ее дескриптор Google+ (например, `+packtpublishing` для страницы издательства Packt Publishing в G+). Другой аргумент определяет максимальное число результатов (то есть действий). Этот аргумент можно опустить, и тогда по умолчанию будет возвращено до 100 результатов.

Сценарий можно запустить, например, командой:

```
$ python gplus_get_page_activities.py --page +packtpublishing
  --max-results 1000
```

Через несколько секунд появится файл `activities_+packtpublishing.jsonl` со списком действий в формате JSON. Данный файл так же, как в случае с Twitter и Facebook, отформатирован согласно спецификации JSON Lines, то есть каждая строка представляет отдельный допустимый документ JSON.

Полное описание одиночного действия имеется в документации на https://developers.google.com/+/web/api/rest/latest/activities#resource. Ниже перечислены наиболее значимые поля:

- `kind`: тип элемента (то есть `plus#activity`);
- `etag`: строка с дескриптором сущности (https://ru.wikipedia.org/wiki/HTTP_ETag);
- `title`: краткое название действия;
- `verb`: сообщение или цитата, то есть когда сообщение цитируется кем-то другим;
- `actor`: объект, представляющий пользователя, который делится действием или его публикует;
- `published`: дата публикации в формате ISO 8601;
- `updated`: дата последнего обновления в формате ISO 8601;
- `id`: уникальный идентификатор действия;
- `url`: URL-адрес действия;
- `object`: составной объект с полной информацией о действии, такой как подробное описание, исходный агент (если действие является цитированием, это значение будет отличаться от агента действия), сведения об ответах, цитированиях и действиях «плюс один», список вложений и их описаний (например, изображения) и информация о географических координатах, если имеется.

В описании объекта введено понятие «*плюс один*», которое чем-то похоже на кнопку «*Нравится*» в Facebook. Когда пользователям нравится что-то в Google+, они могут поставить оценку *+1*. Кнопка «*+1*» позволяет пользователям делиться понравившейся информацией в Google+, и она также рекомендует этот материал службе Google Search (Google Поиск). Подробности о реакциях +1 для конкретного объекта доступны в документе JSON по ключу `plusoners`. Элемент `plusoners.totalItems` содержит число реакций +1, тогда как элемент `plusoners.selfLink` – API-ссылку на список отметок +1. Аналогично поля `replies` и `resharers` содержат ту же информацию о прямых комментариях и цитированиях.

Рассматривая код загрузки списка действий, можно заметить, что отправной точкой по-прежнему является функция `apiclient.discovery.build()`, которая компонует объект службы (`service`). Далее коллекция действий сохраняется в объекте `activity_feed`. Чтобы опросить объект `activity_feed`, вместо функции `search()` мы воспользовались его функцией `list()`, поскольку нам нужен полный список действий (по меньшей мере, вплоть до максимального числа имеющихся результатов). Данная функция требует передачи двух обязательных аргументов: `collection` и `userId`. Аргумент `collection` может принимать лишь значение `public`, а в аргументе `userId` передается идентификатор пользователя/страницы, действия которых мы получаем. Его значением может быть уникальный числовой идентификатор или управляющая строка Google+. Мы также передаем необязательный аргумент `maxResults` со значением `100` в этом примере. Это число элементов, которое можно получить одиночным вызовом API (по умолчанию он принимает значение 20, максимально возможное значение равно 100).

Сценарий также демонстрирует, как в этом API используется постраничное чтение. Запрос выполняется внутри цикла `while`, который увеличивает счетчик на число полученных результатов (цикл выполняется, пока счетчик не достигнет значения, определенного в `--max-results`) и проверяет, существует ли следующая страница с результатами. Функция `list_next()`, которая принимает текущий запрос

и текущий список результатов, занимается обработкой постраничного чтения, обновляя объект запроса. Если следующая страница с результатами отсутствует (то есть сообщения пользователя/страницы исчерпаны), функция вернет `None`.

Анализ текстов и статистическая мера TF-IDF для заметок

Обсудив, как загружать список заметок и действий для той или иной страницы или пользователя, переключим наше внимание на анализ текстовой информации.

Попробуем из каждого сообщения, опубликованного тем или иным пользователем, извлечь наиболее интересные ключевые слова, чтобы потом по ним получить обобщенное представление о самом сообщении.

Это упражнение выглядит простым и понятным, но в нем имеется ряд тонкостей, которые стоит обсудить. Легко заметить, что сообщения не всегда содержат только чистый текст; в него могут включаться теги HTML. Поэтому, прежде чем приступить к вычислениям, нужно вычленить чистый текст. В то время как объект JSON, возвращаемый программным интерфейсом Google+ API, имеет четкую структуру, само содержимое не обязательно оформлено в виде правильно построенного структурированного документа. К счастью, для Python существует хорошая библиотека, которая придет на выручку. Библиотека **BeautifulSoup** на самом деле способна анализировать документы HTML и XML, включая некорректно оформленную разметку. Она совместима с Python 3, и ее можно установить из «сырной лавки» CheeseShop, как обычно:

```
$ pip install beautifulsoup4
```

Эта команда установит версию 4.* библиотеки. Переход с версии 3.* к версии 4.* засвидетельствовал ряд важных преобразований, поэтому примеры, следующие далее, несовместимы с более старыми версиями данной библиотеки.

Следующий важный вопрос: как определить важность ключевого слова?

Данную проблему можно рассматривать с разных сторон, и определение понятия важности может меняться в зависимости от приложения. В нашем случае мы используем подход, основанный на статистике, определяя важность ключевого слова по его присутствию внутри документа и в коллекции документов.

Предложенный подход называется статистической мерой **TF-IDF** (https://ru.wikipedia.org/wiki/TF-IDF), представляющей сочетание двух отметок (**TF** и **IDF**), основанных на частотах терминов. В главе 3 «*Пользователи, читатели и сообщества в Twitter*» мы кратко познакомились со статистической мерой TF-IDF, используя стандартную реализацию, предлагаемую библиотекой scikit-learn. В целом использование существующей реализации представляется неплохой идеей, в особенности если она исходит из высококачественной библиотеки, такой как scikit-learn. Однако в данном разделе мы предложим собственную реализацию с целью показать детали расчета статистики TF-IDF, чтобы читатель смог лучше понять принцип действия этого весового фактора.

В основе статистической меры TF-IDF лежит довольно простая идея: слово является хорошим кандидатом, чтобы представлять документ, если часто встречается внутри документа и редко – во всей коллекции. Эти два свойства отражены в двух оценках: **частоте термина** (TF), то есть частоте термина внутри документа, и **обратной частоте документа** (IDF), вычисленной по всей коллекции документов.

Оценку IDF предложила Карен Спарк Джонс (Karen Sparck Jones) в 1972 г. в контексте исследований в области информационного поиска (https://ru.wikipedia.org/wiki/Джонс,_Карен_Спарк), которая является одним из первопроходцев в данной области. Введенная как эвристическое правило, она показывает связь между специфичностью термина и законом Ципфа (https://ru.wikipedia.org/wiki/Закон_Ципфа).

Закон и распределение Ципфа. Закон Ципфа – это эмпирический закон, который отмечает тот факт, что многие виды данных в различных областях науки можно аппроксимировать распределением Ципфа (то есть распределением с вытянутым хвостом). В главе 2 «*Твиттер – хештеги, темы и временные ряды*» мы уже отмечали, что слова, используемые издательством Packt Publishing в своих твитах, подчиняются этому распределению. За дополнительной информацией о законе Ципфа обращайтесь к главе 2 «*Твиттер – хештеги, темы и временные ряды*».

С математической точки зрения за прошедшие годы был предложен целый ряд схем расчета оценок TF и IDF.

Наибольшее распространение получили два варианта расчета оценки TF: использовать необработанную частоту слова в документе или нормализовать ее по числу слов в документе (то есть частотная интерпретация вероятности наблюдения слова в документе).

Оценка IDF традиционно вычисляется по формуле `log(N/n)`, где `N` – общее число документов, а `n` – число документов, содержащих данный термин. Для терминов, присутствующих в каждом документе, она дает значение оценки IDF, равное нулю (то есть `log(1)`). По этой причине одной из возможных нормализаций для IDF является нормализация «+1»: `1+log(N/n)`.

На рис. 5.8 изображены визуальное представление идеи, лежащей в основе меры TF-IDF, и ее связи с законом Ципфа – слишком частые или слишком редкие слова не являются репрезентативными.

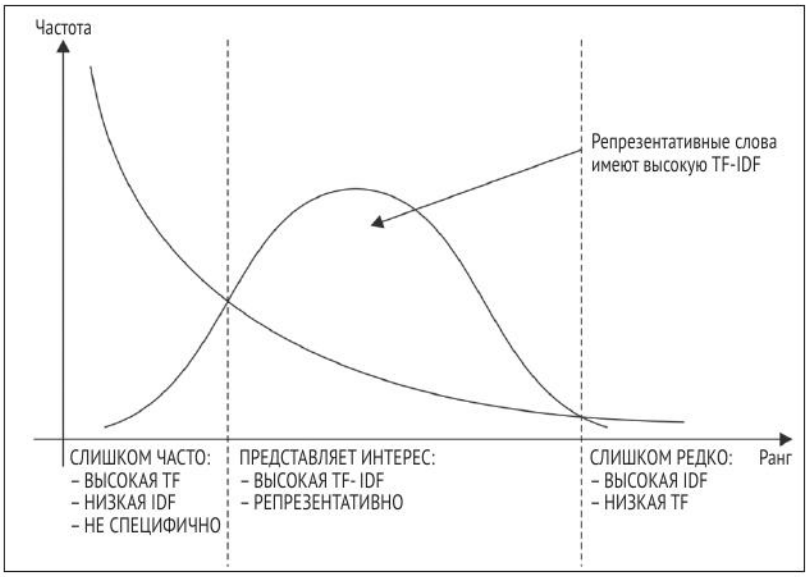

Рис. 5.8 ❖ Распределение слов и их важность

К числу часто встречающихся слов относятся стоп-слова, такие как артикли, союзы и предлоги. Сами по себе они не несут какого-то смысла, поэтому их можно проигнорировать (то есть удалить). С другой стороны, к очень редко встречающимся словам можно отнести опечатки и узкоспециализированные технические термины. В зависимости от приложения иногда желательно провести первичный анализ набора данных, чтобы понять, насколько оправданно удаление стоп-слов.

Сценарий `gplus_activities_keywords.py` читает список действий из заданного файла JSON Lines, вычисляет меру TF-IDF для каждого слова из текстового содержимого каждого действия и затем выводит главные ключевые слова рядом с каждым сообщением.

Для выполнения некоторых операций по обработке текста, таких как удаление стоп-слов и лексемизация, сценарий использует библиотеки NLTK. В главе 1 «*Социальные медиа, социальные данные и Python*» я рассказывал, как загружать дополнительные пакеты NLTK, необходимые для некоторых из этих операций, например пакет `punkt` для лексемизации.

Вот как выглядит начало сценария `gplus_activities_keywords.py`:

```
# Chap05/gplus_activities_keywords.py
import os
import json
from argparse import ArgumentParser
from collections import defaultdict
from collections import Counter
from operator import itemgetter
from math import log
from string import punctuation
from bs4 import BeautifulSoup
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk import bigrams
```

Далее определяется список стоп-слов, включающий общеупотребительные английские стоп-слова и символы пунктуации:

```
punct = list(punctuation)
all_stopwords = stopwords.words('english') + punct

def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--file',
                        '-f',
                        required=True,
                        help='Файл .jsonl со всеми действиями')
    parser.add_argument('--keywords',
                        type=int,
                        default=3,
                        help='Число ключевых слов, извлекаемых из каждого сообщения')
    return parser
```

Функция `preprocess()` выполняет этапы предварительной обработки, включая нормализацию, очистку данных (с помощью `clean_html()`) и лексемизацию:

```
def clean_html(html):
    soup = BeautifulSoup(html, "html.parser")
    text = soup.get_text(" ", strip=True)
    text = text.replace('\xa0', ' ')
    text = text.replace('\ufeff', ' ')
    text = ' '.join(text.split())
    return text

def preprocess(text, stop=all_stopwords, normalize=True):
    if normalize:
        text = text.lower()
    text = clean_html(text)
    tokens = word_tokenize(text)
    return list(bigrams(tokens))
    # return [tok for tok in tokens if tok not in stop]
```

Следующие далее функции работают с полученными статистиками. Функция `make_idf_matrix()` вычисляет оценки IDF в масштабе всей коллекции, которые используются функцией `get_keywords()` для вычисления меры TF-IDF каждого документа, с целью извлечения характеризующих ключевых слов:

```
def make_idf_matrix(corpus):
    df = defaultdict(int)
    for doc in corpus:
        terms = set(doc)
        for term in terms:
            df[term] += 1
    idf = {}
    for term, term_df in df.items():
        idf[term] = 1 + log(len(corpus) / term_df)
    return idf

def get_keywords(doc, idf, normalize=False):
    tf = Counter(doc)
    if normalize:
        tf = {term: tf_value/len(doc) for term, tf_value in tf.items()}
    tfidf = {term: tf_value*idf[term] for term, tf_value in tf.items()}
    return sorted(tfidf.items(), key=itemgetter(1), reverse=True)

if __name__ == '__main__':
    parser = get_parser()
    args = parser.parse_args()
    with open(args.file) as f:

        posts = []
        for line in f:
            activity = json.loads(line)
            posts.append(preprocess(activity['object']['content']))

    idf = make_idf_matrix(posts)

    for i, post in enumerate(posts):
        keywords = get_keywords(post, idf)
        print("----------")
```

```
    print("Сообщение: {}".format(post))
    print("Ключевые слова: {}".format(keywords[:args.keywords]))
```

Как обычно, для чтения аргументов командной строки мы пользуемся экземпляром класса `ArgumentParser`. Аргумент `--file` используется для передачи имени файла, а в необязательном аргументе `--keywords` можно передать число ключевых слов для каждого документа (по умолчанию равно `3`).

Данный сценарий можно запустить так:

```
$ python gplus_activities_keywords.py --file
  activities_+LarryPage.jsonl --keywords 5
```

Команда прочитает файл действий `.jsonl` для пользователя +LarryPage (предполагается, что он уже загружен) и выведет пять наиболее значимых ключевых слов для каждого действия.

Например, вот одно из сообщений Ларри Пейджа:

```
----------
Сообщение: ['fun', 'day', 'kiteboarding', 'alaska', 'pretty', 'cold',
'gusty', 'ago']
Ключевые слова: [('alaska', 5.983606621708336), ('gusty', 5.983606621708336),
('kiteboarding', 5.983606621708336), ('cold', 4.884994333040227),
('pretty', 3.9041650800285006)]
```

Как видите, сообщение выводится в виде списка терминов, а ключевые слова – в виде списка кортежей, где каждый кортеж содержит термин и оценку TF-IDF.

Переход от полного текста к списку лексем выполняет функция `preprocess()`, которая использует вспомогательную функцию `clean_html()`.

Процесс предварительной обработки включает следующие этапы:

- нормализация текста (то есть перевод букв в нижний регистр);
- очистка разметки HTML и удаление тегов;
- лексемизация;
- удаление стоп-слов.

Функция `preprocess()` принимает один обязательный аргумент, `text`, с текстом сообщения и два необязательных именованных аргумента: `stop` с настроенным списком удаляемых стоп-слов (в этом аргументе можно передать пустой список, чтобы исключить удаление каких-либо слов) и `normalize` – признак необходимости перевода букв в нижний регистр.

Нормализация текста используется для группировки слов, отличающихся лишь регистром букв, например в процессе нормализации слова «*The*» и «*the*» будут преобразованы в слово «*the*». Перевод в нижний регистр имеет смысл для большого количества применений, но есть ситуации, где он не помогает, например в случае слова «*us*» (местоимение «нас») против «*U.S.*» (аббревиатура для «*United States*»). Важно понимать, что перевод в нижний регистр необратим, поэтому, если позднее в приложении потребуется сослаться на оригинальный текст, его нужно сохранить.

Очистка разметки HTML выполняется несколькими строками программного кода с применением библиотеки BeautifulSoup. Когда объект `BeautifulSoup` создается с передачей `html.parser` во втором аргументе, он обретает способность обрабатывать некорректно оформленную разметку HTML. Функция `get_text()` делает

именно то, что ожидается: она возвращает только текст, отбрасывая теги HTML. Пробел в первом аргументе этой функции играет роль замены для удаляемых тегов HTML. Это может пригодиться в следующих ситуациях:

```
... некое предложение<br/>Начало следующего
```

Если в этом случае просто удалить тег переноса строки, в результате будет создана лексема `предложениеНачало`, которая, очевидно, не является существующим словом. Замена тега пробелом проблему эту решает.

Кроме того, функция `clean_html()` предусматривает замену двух пробельных символов Юникода. Наконец, если в тексте присутствуют многочисленные последовательности пробелов, они замещаются единственным пробелом путем разбивки и последующего объединения текста (например, `' '.join(text.split())`).

Лексемизация, процесс разбиения строки на отдельные лексемы, реализуется в библиотеке NLTK функцией `word_tokenize()`.

В заключение выполняется удаление стоп-слов с использованием простого генератора списков, который сверяет каждую лексему со списком `stop`. По умолчанию список стоп-слов включает список английских стоп-слов из NLTK и список символов пунктуации из `string.punctuation`.

Реализация расчета меры TF-IDF основана на двух функциях: `make_idf_matrix()` и `get_keywords()`. Поскольку оценка IDF относится ко всему набору, сначала вычисляется она. Аргумент `corpus` – это список списков, то есть список лексемизированных документов, обработанных функцией `preprocess()`.

Сначала вычисляется частота документа с использованием словаря `df`. Это просто число документов, в которых встречается термин, то есть повторения не рассматриваются. Каждый документ в корпусе приводится к типу `set`. Благодаря этому каждое слово в документе учитывается только раз. Затем производится обход элементов словаря `df` и на их основе вычисляется оценка IDF, которая затем возвращается функцией. В этой версии IDF используется нормализация «+1», как обсуждалось выше.

После этого можно приступать к вычислению меры TF-IDF для каждого термина в каждом документе. Выполняя обход всех сообщений, мы вызываем функцию `get_keywords()`, которая принимает два аргумента: документ (список лексем) и словарь IDF, созданный ранее. Третий необязательный аргумент – это признак необходимости нормализации по длине документа, которая эффективно превращает TF в условную вероятность наблюдения термина в документе, то есть `P(t|d)`.

Функция `get_keywords()` сначала создает ненормализованную частоту термина TF посредством специальной структуры данных `collections.Counter`, которая в действительности является словарем. Если требуется нормализация в объеме документа, с помощью генератора словарей производится пересборка словаря `tf` делением значения каждой исходной частоты термина (TF) на общее число терминов в документе (то есть на длину документа).

Мера TF-IDF вычисляется путем умножения TF на IDF для каждого термина. Для этого снова используется генератор словарей.

В заключение функция возвращает список терминов с их оценками TF-IDF, при этом список отсортирован по значениям оценок. Функция `sorted()` вызывается с `key=itemgetter(1)`, сортировка осуществлялась по величине меры TF-IDF

(второй элемент в кортеже) и с `reverse=True`, для сортировки в порядке убывания значений.

Получение словосочетаний при помощи n-грамм

Мера TF-IDF дает простое статистическое объяснение важности ключевого слова в том или ином документе. Эта модель рассматривает каждое слово в отдельности и без учета порядка следования. В действительности же слова используются в определенной последовательности, и часто последовательность слов выступает как единый блок внутри структуры предложения (который в лингвистике также называется «составляющей»).

n-граммами в лексимизированном документе (то есть последовательности лексем) называются последовательности из `n` смежных слов. При `n=1` n-граммы вырождаются в одиночные слова (или униграммы). Со значением `n` больше единицы рассматриваются последовательности соответствующей длины. Типичные примеры включают биграммы (`n=2`) и триграммы (`n=3`), но теоретически возможна любая длина последовательности.

Библиотека NLTK предоставляет средства для быстрого получения списка n-грамм из имеющегося списка лексем. К ним, например, относится функция `nltk.util.ngrams`:

```
>>> from nltk.util import ngrams
>>> s = 'Съешь же еще этих мягких французских булок, да выпей чаю'.split()
>>> s
['Съешь', 'же', 'еще', 'этих', 'мягких', 'французских', 'булок,', 'да', 'выпей', 'чаю']
>>> list(ngrams(s, 4))
[('Съешь', 'же', 'еще', 'этих'), ('же', 'еще', 'этих', 'мягких'), ('еще', 'этих', 'мягких',
'французских'), ('этих', 'мягких', 'французских', 'булок,'), ('мягких', 'французских',
'булок,', 'да'), ('французских', 'булок,', 'да', 'выпей'), ('булок,', 'да', 'выпей', 'чаю')]
```

Как видите, функция `ngrams()` принимает два аргумента: последовательность лексем и число – и возвращает объект-генератор, который нужно преобразовать в список, чтобы вывести его содержимое в командной оболочке, но его можно использовать непосредственно, например в генераторе списков.

Библиотека NLTK также предоставляет две вспомогательные функции: `nltk.bigrams()` и `nltk.trigrams()` – которые просто вызывают `ngrams()` с аргументами `n=2` и `n=3` соответственно.

Мы завершим данный раздел следующими вопросами, ответы на которые мы предлагаем читателям найти самостоятельно:

- можно ли изменить предложенный в данном разделе программный код, чтобы получать биграммы или триграммы, а не только отдельные ключевые слова?
- каков будет эффект от удаления стоп-слов во время получения n-грамм?

Резюме

В настоящей главе мы рассмотрели программный интерфейс Google+ API. Мы также обсудили регистрирацию проекта в Диспетчере API и включение API, необходимого проекту.

Данная глава началась с примера, который показал, как выполнять поиск с использованием Google+ API. Дальнейшее обсуждение основывалось на данном примере, с тем чтобы встроить применение API в веб-приложение. Задействовав микрофреймворк Flask для веб-разработки, позволяющий реализовать проект в короткие сроки всего несколькими строками программного кода, мы создали веб-интерфейс для отображения результатов поиска.

Затем мы проанализировали текстовые заметки. После загрузки пользовательских действий и сохранения их в формате JSON Lines мы обсудили технические детали вычисления статистической меры TF-IDF и использовали ее для извлечения характеризующих ключевых слов из фрагмента текста.

В следующей главе мы переключим наше внимание на область вопросно-ответных систем и исследуем программный интерфейс Stack Overflow API.

Глава 6

Вопросы и ответы в сети Stack Exchange

Настоящая глава посвящена вопросно-ответной сети Stack Exchange и более широкой теме ответов на вопросы.

Эта глава охватывает следующие темы:

- создание приложения, взаимодействующего с Stack Exchange API;
- поиск пользователей и вопросов в Stack Exchange;
- обработка дампов данных в офлайн;
- методы машинного обучения с учителем для классификации текстов;
- предсказание тегов вопроса с использованием машинного обучения;
- встраивание модели машинного обучения в приложение, работающее в режиме реального времени.

Вопросы и ответы

Поиск ответов на вопросы является одним из главных применений Всемирной паутины. Со временем технология эволюционировала, и интернет-пользователи стали менять свои модели поведения.

Поведение людей, ищущих информацию во Всемирной паутине, сильно изменилось за последние 15–20 лет. В прошлом поиск ответов главным образом означал использование поисковой системы. Исследование журналов запросов популярной поисковой системы в конце 90-х («*Searching the Web: The Public and Their Queries*», *Amanda Spink and others, 2001*) показало, что в те дни длина типичного поискового запроса составляла в среднем 2,4 термина.

В последние годы мы начали замечать переход от коротких поисковых запросов по ключевым словам к более длинным диалоговым запросам (можно даже сказать «вопросам»). Другими словами, поисковые системы двигались от сопоставления по ключевым словам к **обработке естественного языка**. Например, на рис. 6.1 показано, как Google пытается дополнить запросы/вопросы пользователя. Система использует статистику популярных запросов, пытаясь предсказать, как пользователь намеревается завершить вопрос.

Пользовательский интерфейс на естественном языке помогает дать более естественный опыт конечному пользователю. Более сложный пример – интеллектуальные личные помощники, встроенные в приложения для смартфонов, такие как Google Now, Apple Siri или Microsoft Cortana, позволяющие пользователям вы-

полнять поиск во Всемирной паутине, получать рекомендации о ресторане или планировать деловую встречу.

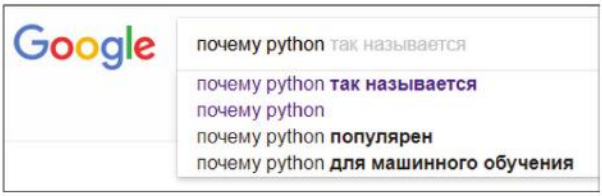

Рис. 6.1 ❖ Пример автодополнения запроса от Google

Задать вопрос – это лишь полдела, поскольку самая важная часть заключается в поиске ответа на заданный вопрос. На рис. 6.2 изображено начало страницы с результатами, полученными в Google в ответ на вопрос: `происхождение названия языка Python`. Когда Google интерпретирует запрос как диалоговый, то есть вопрос в буквальном смысле, а не обобщенный информационный запрос, активируется так называемое *поле ответа*. Данное поле содержит отрывок веб-страницы, который по мнению Google является ответом на заданный вопрос. Остальная часть страницы, которая на рис. 6.2 не показана, содержит традиционную **страницу с результатами поиска** (search engine result page, SERP):

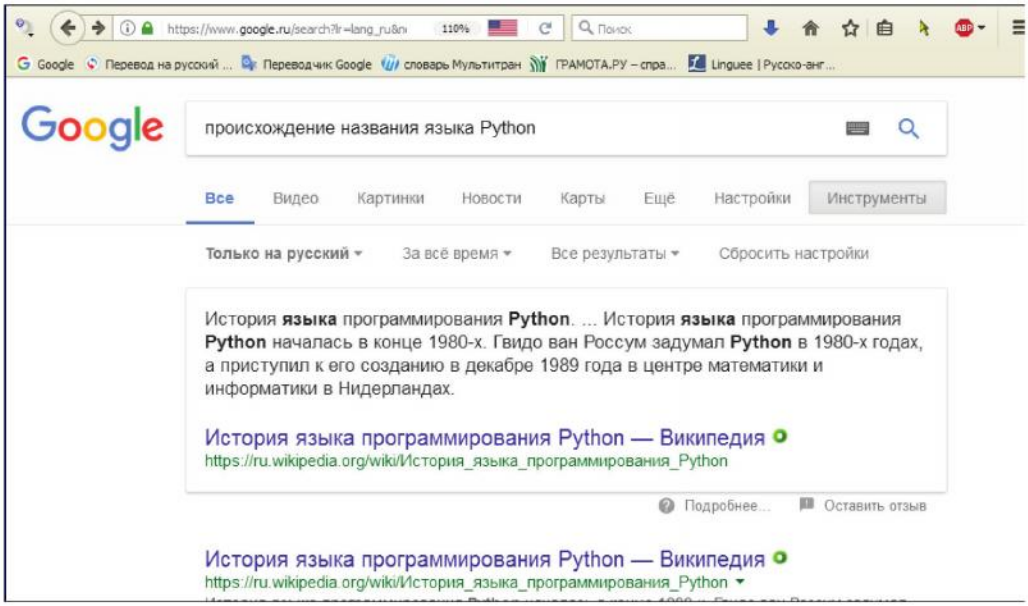

Рис. 6.2 ❖ Пример поля ответа в результате диалогового запроса

Исследование в области автоматического формирования **ответов на вопросы** (question answering, QA) не ограничивается областью поисковых систем. Известным достижением в области искусственного интеллекта является суперкомпью-

тер IBM Watson с вопросно-ответной системой, разработанной в компании IBM и названной в честь первого руководителя IBM Томаса Уотсона (Thomas Watson) (https://ru.wikipedia.org/wiki/IBM_Watson). Эта машина получила известность, когда соревновалась с людьми-знатоками в игре *Jeopardy!* и в итоге победила. Это популярное американское игровое телешоу – конкурс знатоков, – где соперники должны найти правильный вопрос после нескольких подсказок, связанных с ответом. Хотя данное состязание полностью меняет процесс местами, то есть требуется найти вопрос для того или иного ответа, технология решения задачи может работать в обоих направлениях.

В то время как поисковые системы и интеллектуальные личные помощники совершенствовали способы поиска информации онлайн, насущная потребность находить ответы также способствовала распространению вопросно-ответных веб-сайтов (https://en.wikipedia.org/wiki/Comparison_of_Q%26A_sites). Одной из самых популярных вопросно-ответных сетей является Stack Exchange (http://stackexchange.com), получившая известность благодаря веб-сайту Stack Overflow (http://stackoverflow.com), чья особенность заключается в вопросах и ответах по широкому кругу тем, связанных с программированием. В наши дни данная сеть состоит из десятков тематических веб-сайтов, от технологии до науки и от бизнеса до развлечений. Другие примеры веб-сайтов Stack Exchange включают «Кинофильмы & ТВ» (для любителей кино и ТВ-сериалов), «Солидный совет» (для профессионалов и поваров-любителей), «Английский язык & его применение» (для лингвистов, этимологов и энтузиастов английского языка) и «Академия» (для академиков и специалистов в области высшего образования). Список отдельных веб-сайтов очень длинный, и темы разнообразны (http://stackexchange.com/sites).

Одна из причин успеха сети Stack Exchange заключена в высококачественном информационном наполнении, курируемом сообществом. Пользователи сайтов фактически голосуют за вопросы и ответы, в результате чего лучшие ответы поднимаются на вершину списка. Механизм репутационных очков, зарабатываемых посредством использования веб-сайта, позволяет выявить наиболее активных членов сообщества и областей их знаний. Пользователи могут зарабатывать (или терять) очки, когда их ответы получают голос «за» или «против» и когда они предлагают свои исправления с целью улучшить качество веб-сайта. Игрофикация системы также включает серию *бейджей*, которые пользователи могут заработать за свой вклад.

Чтобы поддерживать качество материала на высоком уровне, повторяющиеся и низкокачественные вопросы обычно рассматриваются модераторами и редактируются или закрываются.

Некоторые из основных особенностей, присущих службам социальных сетей, в сети Stack Exchange отсутствуют. Например, отсутствует возможность прямого контакта с другими пользователями (как связь «друг» или «читатель» в Facebook и Twitter) и нельзя вести частные беседы с другими пользователями. И все же эффект сети Stack Exchange как социально-медийной платформы вполне понятен – люди обмениваются знанием и сотрудничают.

Следующий раздел знакомит с программным интерфейсом Stack Exchange API.

Начало работы с Stack Exchange API

Программный интерфейс Stack Exchange API (https://api.stackexchange.com/docs) открывает программируемый доступ ко всем веб-сайтам сети Stack Exchange. Сторонние приложения, намеревающиеся использовать его, должны регистрироваться (https://stackapps.com/apps/oauth/register), чтобы получить ключ запроса, позволяющий выполнять большое число запросов в день. Кроме того, зарегистрированные приложения могут взаимодействовать с сетью Stack Exchange от имени пользователя. Как показано на рис. 6.3, процесс регистрации довольно прост – требуется указать только **название приложения** и его **описание**:

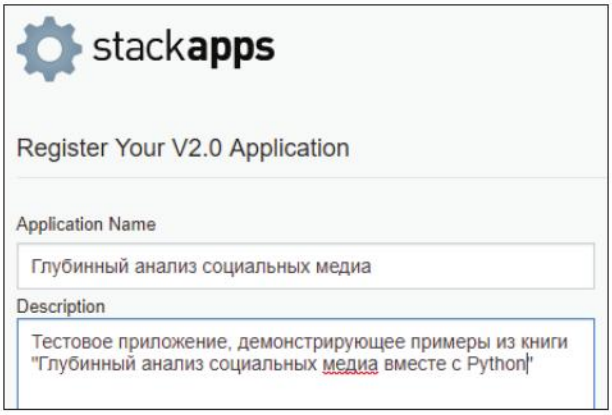

Рис. 6.3 ❖ Регистрация приложения на stackapps.com

Непосредственный эффект регистрации приложения – увеличение числа запросов в день, разрешаемых программным интерфейсом. Важно отметить, что ограничения на частоту запросов устанавливаются не только на ежедневной основе, но и предотвращают лавинообразный рост числа обращений в сети (например, если тот же самый IP отправляет более 30 запросов в секунду, его запросы будут отбрасываться). Подробнее об ограничениях на частоту запросов можно узнать в документации (https://api.stackexchange.com/docs/throttle).

В регистрационной форме сообщается, что «Требуется URL-адрес веб-сайта». Если у вас нет своего веб-сайта, чтобы зарегистрировать приложение, введите в поле «Application Website» URL-адрес stackexchange.com и этот же адрес введите в поле «OAuth Domain». Это позволит использовать https://stackexchange.com/oauth/login_success в качестве вашего URL-адреса перенаправления (redirect_uri), а также авторизоваться, если потребуется.

После регистрации приложения запишите на лист бумаги ключи, которые были предоставлены сетью Stack Exchange. *client ID* (идентификатор клиента) и *client secret* (секрет клиента) потребуется использовать в процедуре аутентификации посредством OAuth. После аутентификации для приложения увеличивается квота запросов. Важно отметить, что согласно документации идентификатор и ключ клиента не являются секретными. В то же время секрет клиента, как следует из названия, относится к конфиденциальной информации, которая не подлежит распространению.

Мы можем настроить использование ключа приложения, следуя той же процедуре, что и в предыдущих главах, применив переменные окружения:

```
$ export STACK_KEY="your-application-key"
```

Следующий инструмент, который нам потребуется для взаимодействий с API, – это клиент API. На данный момент имеется один (неофициальный) клиент Stack Exchange API для Python под названием **Py-StackExchange** (https://github.com/lucjon/Py-StackExchange). Данный клиент просто привязывает конечные точки API к методам Python при помощи простого интерфейса.

Библиотеку Py-StackExchange можно установить из «сырной лавки» CheeseShop, следуя обычной процедуре:

```
$ pip install py-stackexchange
```

Чтобы протестировать клиента и продемонстрировать его базовые классы, воспользуемся интерактивной оболочкой Python и попробуем получить некоторую обобщенную статистику о веб-сайте Stack Overflow:

```
>>> import os
>>> import json
>>> from stackexchange import Site
>>> from stackexchange import StackOverflow
>>> api_key = os.environ.get('STACK_KEY')
>>> so = Site(StackOverflow, api_key)
>>> info = so.info()
>>> print(info.total_questions)
14177191
>>> print(info.questions_per_minute)
3.02
>>> print(info.total_comments)
70215276
>>> print(json.dumps(info.json, indent=4))
{
    "new_active_users": 22,
    "total_users": 7372637,
    "badges_per_minute": 4.96,
    "total_badges": 23319245,
    "total_votes": 103403588,
    "total_comments": 70215276,
    "answers_per_minute": 4.74,
    "questions_per_minute": 3.02,
    "total_answers": 22257829,
    "total_accepted": 7634665,
    "total_unanswered": 4042626,
    "total_questions": 14177191,
    "api_revision": "2017.6.29.26176",
    "_params_": {
        "body": "false",
        "comments": "false",
        "site": "stackoverflow.com"
    }
}
```

Основной класс, взаимодействующий с API, называется `stackexchange.Site`. Его конструктор вызывается с двумя параметрами: первый – определение класса для конкретного веб-сайта сети Stack Exchange (в данном случае `stackexchange.StackOverflow`) и второй (необязательный) – ключ приложения. После определения объекта `Site` можно использовать его методы для получения доступа к API.

В этом примере мы использовали метод `info()`, чтобы получить доступ к конечной точке `/info` (https://api.stackexchange.com/docs/info). Возвращаемый им объект имеет ряд атрибутов. В данном примере мы вывели значения атрибутов `total_questions`, `questions_per_minute` и `total_comments` (общее число вопросов, число вопросов в минуту и общее число комментариев). Полный список атрибутов можно увидеть в словаре `info.__dict__`, но вообще объект `info` имеет также атрибут `info.json`, содержащий оригинальный ответ API, где все вышеупомянутые атрибуты показаны в формате JSON.

Поиск вопросов с тегами

После начального знакомства с основами Stack Exchange API попробуем выполнить поиск вопросов с помощью клиента для Python. В частности, используем наши ключевые слова в роли тегов в вопросах.

Такой поиск реализует следующий сценарий:

```
# Chap06/stack_search_keyword.py
import os
import json
from argparse import ArgumentParser
from stackexchange import Site
from stackexchange import StackOverflow

def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--tags')
    parser.add_argument('--n', type=int, default=20)
    return parser

if __name__ == '__main__':
    parser = get_parser()
    args = parser.parse_args()
    my_key = os.environ.get('STACK_KEY')

    so = Site(StackOverflow, my_key)
    questions = so.questions(tagged=args.tags, pagesize=20)[:args.n]

    for i, item in enumerate(questions):
        print("{}) {}, автор: {}".format(i,
                                         item.title,
                                         item.owner.display_name))
```

Для чтения аргументов командной строки в этом примере используется экземпляр класса `ArgumentParser`. В аргументе `--tags` сценарию передается список искомых тегов, разделенных точкой с запятой. Число возвращаемых результатов можно задать с помощью флага `--n` (по умолчанию имеет значение `20`).

Данный сценарий можно выполнить, например, следующей командой:

```
$ python stack_search_keyword.py --tags "python;nosql" --n 10
```

Список тегов необходимо заключить в двойные кавычки, поскольку в таких оболочках, как Bash, точка с запятой является разделителем команд и без кавычек такая команда интерпретировалась бы как две отдельные команды (вторая начиналась бы со слова `nosql`). И напротив, используя двойные кавычки, подстрока `python;nosql` интерпретируется как одно строковое значение, а вся командная строка – как одна команда, что нам и требуется.

Список тегов передается API в аргументе `tagged` метода `questions()`. Теги, указанные в этом аргументе, объединяются по «И», в том смысле, что возвращаемые вопросы отмечены обоими тегами, `python` *И* `nosql`.

Логические запросы. Метод `questions()` реализует конечную точку `/questions` (https://api.stackexchange.com/docs/questions), которая объединяет искомые теги по «И». В результате данный метод возвращает список вопросов, содержащих все указанные теги.
Метод `search()`, напротив, реализует конечную точку `search/` (https://api.stackexchange.com/docs/search), которая реализует противоположное поведение, объединяя искомые теги по «ИЛИ». В результате данный метод возвращает список вопросов, содержащих любой из указанных тегов.

Следует отметить одну важную деталь: клиент Py-StackExchange часто использует ленивые списки с целью минимизировать количество вызовов API, которые выполняются за кулисами. Это означает, что такие методы, как `questions()`, возвращают не список, а обертку вокруг набора результатов. Во время итераций через такой набор результатов будет сделана попытка обратиться ко всем элементам, соответствующим запросу (если поступить неосмотрительно, это лишит смысла использование ленивого списка), а не только списку с размером, соответствующим странице.

По этой причине мы используем операцию извлечения среза, чтобы явно ограничить число результатов (которое задается в командной строке аргументом `--n` и, следовательно, доступно как `args.n`).

Получение срезов и диапазонов из списков Python. Операция извлечения среза – мощный инструмент, но ее синтаксис может показаться малопонятным для неопытных программистов на Python. Следующий пример демонстрирует основные формы применения:

```
#  выбрать элементы, начиная с start и до end-1
array[start:end]
#  выбрать элементы, начиная с start и до конца массива
array[start:]
#  выбрать элементы с начала и до end-1
array[:end]
#  получить копию всего массива
array[:]
#  выбрать элементы, начиная с start и до end с шагом step
array[start:end:step]
#  получить последний элемент
array[-1]
#  получить последние n элементов
array[-n:]
#  получить все, кроме n элементов
array[:-n]
```

Например:

```
>>> data = ['один', 'два', 'три', 'четыре', 'пять']
>>> data[2:4]
['три', 'четыре']
>>> data[2:]
['три', 'четыре', 'пять']
>>> data[:4]
['один', 'два', 'три', 'четыре']
>>> newdata = data[:]
>>> newdata
['один', 'два', 'три', 'четыре', 'пять']
>>> data[1:5:2]
['два', 'четыре']
>>> data[-1]
'пять'
>>> data[-1:]
['пять']
>>> data[:-2]
['один', 'два', 'три']
```

Помните, что индексация начинается с нуля, то есть `data[0]` – это первый элемент, `data[1]` – второй и т. д.

Результатом работы сценария `stack_search_keyword.py`, запущенного командой выше, является последовательность из 10 заголовков вопросов, связанных с Python и NoSQL, в следующем формате:

```
n) Заголовок вопроса, автор: пользователь
```

Отдельные вопросы (доступные в цикле `for` через переменную `item`) обертываются в класс модели `stackexchange.model.Question` с такими атрибутами, как `title` или `tags`. Каждый вопрос также связан через атрибут `owner` с моделью пользователя, который его задал. В приведенном примере мы получаем доступ к заголовку `title` вопроса и отображаемому имени `display_name` пользователя.

Набор результатов отсортирован по дате последней активности, то есть вопросы с наиболее поздней активностью (например, с новым ответом) возвращаются первыми.

Такое поведение можно изменить с помощью атрибута `sort` метода `questions()`, который может принимать любое из следующих значений:

- `activity` (значение по умолчанию): первыми возвращаются вопросы с недавними действиями;
- `creation`: первыми возвращаются недавние вопросы;
- `votes`: первыми возвращаются вопросы с высшей оценкой;
- `hot`: используется формула для вкладки **горячих вопросов**;
- `week`: используется формула для вкладки **вопросов недели**;
- `month`: используется формула для вкладки **вопросов месяца**.

Поиск пользователя

Научившись выполнять поиск вопросов, в этом разделе мы обсудим, как отыскать конкретного пользователя. Процедура поиска пользователя очень похожа на поиск вопросов. Следующий пример строится на предыдущем и расширяет исполь-

зование экземпляра класса ArgumentParser для чтения дополнительных параметров поиска:

```
# Chap06/stack_search_user.py
import os
import json
from argparse import ArgumentParser
from argparse import ArgumentTypeError
from stackexchange import Site
from stackexchange import StackOverflow
```

Следующие далее функции проверяют допустимость аргументов, переданных через ArgumentParser. Если значение какого-то аргумента недопустимо, они возбуждают исключение, останавливая выполнение сценария:

```
def check_sort_value(value):
    valid_sort_values = [
        'reputation',
        'creation',
        'name',
        'modified'
    ]
    if value not in valid_sort_values:
         raise ArgumentTypeError(
                 "{} является недопустимым значением для \"sort\""
                 .format(value))
    return value

def check_order_value(value):
    valid_order_values = ['asc', 'desc']
    if value not in valid_order_values:
        raise ArgumentTypeError(
                 "{} является недопустимым значением для \"order\""
                 .format(value))
    return value
```

Затем функции check_sort_value() и check_order_value() применяются в определениях типов соответствующих параметров:

```
def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--name')
    parser.add_argument('--sort',
                        default='reputation',
                        type=check_sort_value)
    parser.add_argument('--order',
                        default='desc',
                        type=check_order_value)
    parser.add_argument('--n', type=int, default=20)
    return parser
```

Главная логика сценария, реализующая поиск, довольно проста:

```
if __name__ == '__main__':
    parser = get_parser()
```

```
    args = parser.parse_args()
    my_key = os.environ.get('STACK_KEY')

    so = Site(StackOverflow, my_key)

    users = so.users(inname=args.name,
                     sort=args.sort,
                     order=args.order)
    users = users[:args.n]

    for i, user in enumerate(users):
        print("{}) {}, репутация {}, присоединился {}".format(i,
              user.display_name,
              user.reputation,
              user.creation_date))
```

Для чтения аргументов командной строки сценарий `stack_search_user.py`, как обычно, использует объект `ArgumentParser`. Разница с предыдущим использованием состоит в уточнении требований к типам данных для некоторых аргументов, а именно `--sort` и `--order`. Прежде чем вдаваться в подробности типов данных, извлекаемых классом `ArgumentParser`, рассмотрим пример их использования.

Допустим, нам нужно найти основателей Stack Exchange Джеффа Этвуда (Jeff Atwood) и Джоэла Спольски (Joel Spolsky). Выполнить поиск только по одному имени можно так:

```
$ python stack_search_user.py --name joel
```

Эта команда выведет следующее:

```
0) Joel Coehoorn, репутация 267423, присоединился 2008-08-26 23:24:14
1) Joel Etherton, репутация 30987, присоединился 2010-01-15 01:39:24
2) Joel Martinez, репутация 29519, присоединился 2008-09-09 23:41:43
3) Joel Spolsky, репутация 27276, присоединился 2008-08-01 00:22:31
# (обрезано)
```

По умолчанию метод `Site.users()`, реализующий конечную точку `/users` (https://api.stackexchange.com/docs/users), возвращает список пользователей в порядке убывания репутационных очков.

Интересно отметить, что один из основателей Stack Overflow в этом наборе результатов имеет не самую высокую репутацию. Давайте посмотрим, является ли дата его регистрации самой старой. Для этого добавим в поисковый запрос два аргумента:

```
$ python stack_search_user.py --name joel --sort creation --order asc
```

Теперь, как и ожидалось, пользователь `Joel Spolsky` соответствует именно тому Джоэлу и имеет самую старую дату регистрации:

```
0) Joel Spolsky, репутация 27276, присоединился 2008-08-01 00:22:31
1) Joel Lucsy, репутация 6814, присоединился 2008-08-07 22:58:41
2) Joel Meador, репутация 2023, присоединился 2008-08-20 02:34:45
3) JoelB, репутация 99, присоединился 2008-08-24 09:05:44
4) Joel Coehoorn, репутация 267433, присоединился 2008-08-26 23:24:14
# (обрезано)
```

Применение экземпляра класса `ArgumentParser` в этом примере позволяет контролировать типы данных для отдельных аргументов. Как правило, в аргументе

type функции add_argument() передаются встроенные типы данных (такие как int или bool), но в нашем случае мы можем пойти дальше и добавить дополнительную элементарную проверку.

Проблема в том, что параметры sort и order, которые передаются конечной точке /users, принимают ограниченный набор строковых значений. Функции check_sort_value() и check_order_value() просто подтверждают, что то или иное значение находится в списке приемлемых. Если это не так, функции возбудят исключение ArgumentTypeError, которое будет перехвачено анализатором ArgumentParser и использовано для вывода сообщения. Например, если выполнить следующую команду:

```
$ python stack_search_user.py --name joel --sort FOOBAR --order asc
```

то недопустимое значение аргумента --sort приведет к выводу сообщения:

```
usage: stack_search_user.py [-h] [--name NAME] [--sort SORT] [--order
ORDER]
  [--n N]
stack_search_user.py: error: FOOBAR является недопустимым значением для --sort
```

Без такой проверки вызов сценария с неправильными аргументами заставит API вернуть ошибку, которая будет перехвачена клиентом Py-StackExchange и показана как StackExchangeError со связанной с ней обратной трассировкой:

```
Traceback (most recent call last):
  File "stack_search_user.py", line 48, in <module>
# (длинная трассировка)
stackexchange.core.StackExchangeError: 400 [bad_parameter]: sort
```

Другими словами, такая проверка позволяет обрабатывать ошибки и показывать пользователю более понятные сообщения.

Обработка дампов данных из Stack Exchange

Сеть Stack Exchange также предоставляет для загрузки полные дампы своих данных (https://archive.org/details/stackexchange). Данные доступны в сжатом формате 7Z с высоким коэффициентом сжатия (http://www.7-zip.org). Для работы с этим форматом необходимо установить утилиту 7-zip для Windows или одну из ее версий для Linux/Unix или macOS.

Во время написания данной главы дампы данных Stack Overflow предлагались в виде отдельных сжатых файлов: каждый файл представлял сущность или таблицу со своим набором данных. Например, файл stackoverflow.com-Posts.7z содержит дамп из таблицы сообщений Posts (то есть вопросов и ответов). Размер первой версии этого файла, опубликованной в 2016, составлял приблизительно 7,9 Гбайта, который в несжатом виде имеет объем 39 Гбайт (примерно в пять раз больше, чем в сжатой версии). Все другие веб-сайты Stack Exchange имеют гораздо меньшие дампы данных, в которые включены все их таблицы.

Например, данные с веб-сайта «Кинофильмы & ТВ» (http://movies.stackexchange.com), где любители кино и телевидения задают и отвечают на вопросы по данной тематике, доступны в виде одного файла movies.stackexchange.com.7z. В этом архиве находятся восемь файлов (один для каждой сущности), перечисленных в следующей таблице:

Имя файла	Представленный объект
movies.stackexchange.com	Бейджи
movies.stackexchange_1.com	Комментарии
movies.stackexchange_2.com	История сообщений
movies.stackexchange_3.com	Ссылки в сообщениях
movies.stackexchange_4.com	Сообщения (вопросы и ответы)
movies.stackexchange_5.com	Теги
movies.stackexchange_6.com	Пользователи
movies.stackexchange_7.com	Голоса

К файлам применяются соглашения об именовании, общие для всех дампов разных сайтов, например в дампах любых сайтов сети файл `website_4.com` должен содержать список сообщений.

Файлы в дампе содержат информацию в формате XML. Структура файла подчинена следующему шаблону:

```
<?xml version="1.0" encoding="utf-8"?>
  <объект>
    <row [атрибуты] />
    <!-- еще строки -->
  </объект>
```

Первая строка – стандартное объявление XML. Корневой элемент XML-документа представляет тип сущности (например, `posts`, `users` и т. д.). Каждая запись для конкретной сущности представлена элементом `<row>` с рядом атрибутов. Например, вот небольшой фрагмент с описанием сущности `posts`:

```
<?xml version="1.0" encoding="utf-8"?>
  <posts>
    <row Id="1" PostTypeId="1" ... />
    <row Id="2" PostTypeId="2" ... />
    <row Id="3" PostTypeId="1" ... />
    <!-- еще строки -->
  </posts>
```

Поскольку элемент `posts` используется для представления вопросов и ответов одновременно, атрибуты от строки к строке могут отличаться.

В следующей таблице перечисляются самые важные атрибуты элемента `<row>`, представляющего вопрос:

Название атрибута	Описание
`Id`	Уникальный идентификатор сообщения
`PostTypeId`	1 для вопросов
`AcceptedAnswerId`	Идентификатор сообщения, отмеченного как принятый ответ
`CreationDate`	Дата отправки вопроса в формате ISO 8601
`Score`	Число голосов «за» минус голоса «против»
`ViewCount`	Число просмотров вопроса
`Title`	Заголовок вопроса
`Body`	Полный текст вопроса
`OwnerUserId`	Идентификатор пользователя, отправившего вопрос

Название атрибута	Описание
Tags	Список тегов для вопроса
AnswerCount	Число ответов на вопрос
CommentCount	Число комментариев к вопросу
FavoriteCount	Число пользователей, отметивших вопрос как лучший

Аналогичо следующая далее таблица перечисляет важные атрибуты элемента <row>, представляющего ответ:

Название атрибута	Описание
Id	Уникальный идентификатор сообщения
PostTypeId	2 для ответов
ParentId	Идентификатор вопроса, на который отвечает это сообщение
OwnerUserId	Идентификатор пользователя, отправившего ответ
Score	Число голосов «за» минус голоса «против»
CommentCount	Число комментариев к ответу
Body	Полный текст ответа

На протяжении всей книги мы работали в основном с данными в формате JSON Lines, поскольку он обеспечивает удобное представление.

Предложенный ниже сценарий stack_xml2json.py демонстрирует преобразование дампа данных из XML в более знакомый формат JSON Lines. При этом он выполняет дополнительную очистку данных.

Python сопровождается достойной поддержкой формата XML, встроенной в стандартную библиотеку, тем не менее стоит обратить внимание на интересную библиотеку под названием **lxml** (http://lxml.de). Она реализует интерфейс между Python и низкоуровневыми библиотеками на C: **libxml2** и **libxslt**, совмещающий чрезвычайную производительность и богатство возможностей с простотой использования типичных хорошо продуманных библиотек на Python.

Единственный потенциальный недостаток заключается в процессе установки, если с библиотеками libxml2 и libxslt, которые являются необходимыми зависимостями, возникнут какие-то проблемы. Процедура установки обычная и сводится к использованию диспетчера пакетов pip, как это делалось для всех ранее установленных библиотек:

```
$ pip install lxml
```

Документация к библиотеке (http://lxml.de/installation.html) включает детальное обсуждение предпочтительных версий библиотек на C, а также приемы оптимизации. В зависимости от платформы и параметров системы имеется много деталей, которые могут вызвать проблемы в процессе установки. Диагностика всех возможных неисправностей выходит за рамки настоящей книги, однако в вебе можно найти много информации по данной теме (и сам веб-сайт Stack Overflow является видным источником).

К счастью, библиотека lxml разрабатывалась с прицелом на максимальную совместимость с пакетом **ElementTree** (частью стандартной библиотеки Python), поэтому установка lxml не является критически важной для чтения данного раздела. Предлагаем попробовать ее установить, и если процесс усложнится, вы смо-

жете откатиться назад, не слишком переживая по этому поводу. Фактически, если библиотека lxml отсутствует, сценарий автоматически переходит на использование пакета ElementTree:

```
# Chap06/stack_xml2json.py
import json
from argparse import ArgumentParser
from bs4 import BeautifulSoup
try:
    from lxml import etree
except ImportError:
    import xml.etree.ElementTree as etree

def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--xml')
    parser.add_argument('--json')
    parser.add_argument('--clean-post', default=False, action='store_true')
    return parser

def clean_post(doc):
    try:
        doc['Tags'] = doc['Tags'].replace('><', ' ')
        doc['Tags'] = doc['Tags'].replace('<', '')
        doc['Tags'] = doc['Tags'].replace('>', '')
    except KeyError:
        pass
    soup = BeautifulSoup(doc['Body'], 'html.parser')
    doc['Body'] = soup.get_text(" ", strip=True)
    return doc

if __name__ == '__main__':
    parser = get_parser()
    args = parser.parse_args()

    xmldoc = etree.parse(args.xml)
    entity = xmldoc.getroot()
    with open(args.json, 'w') as fout:
        for row in entity:
            doc = dict(row.attrib)
            if args.clean_post:
                doc = clean_post(doc)
            fout.write("{}\n".format(json.dumps(doc)))
```

Импортирование библиотеки lxml заключено в блок `try/except`, перехватывающий исключение `ImportError`. Исключение этого типа возникает при попытке импортировать модуль/пакет, отсутствующий в пути поиска Python.

Для чтения аргументов командной строки сценарий `stack_xml2json.py` вновь использует экземпляр класса `ArgumentParser`. Данному сценарию можно передать необязательный флаг `--clean-post`, чтобы включить режим особой интерпретации для сущностей posts, отличной от всех других сущностей, как описывается далее в этом разделе.

Например, преобразовать в формат JSON файл тегов для набора данных «Кинофильмы & ТВ» можно следующей командой:

```
$ python stack_xml2json.py --xml movies.stackexchange_5.com --json
  movies.tags.jsonl
```

В результате будет создан файл movies.tags.jsonl, где каждый тег представлен однострочным документом JSON. С помощью командной оболочки Bash можно быстро обследовать файл:

```
$ wc -l movies.tags.jsonl
```

Эта команда подсчитает число строк в файле и, соответственно, число тегов:

```
2218 movies.tags.jsonl
```

Чтобы исследовать первый документ JSON, то есть первую строку в файле, выполните следующую команду:

```
$ head -1 movies.tags.jsonl
```

Она выведет следующий результат:

```
{"Count": "108", "WikiPostId": "279", "Id": "1", "TagName": "comedy",
"ExcerptPostId": "280"}
```

Парсинг XML-документа в переменную xmldoc с древовидным представлением в форме экземпляра ElementTree, указанный сценарий использует функцию lxml.etree. Чтобы получить доступ к конкретному объекту в дереве, в качестве точки входа для обхода строк используется метод getroot(). Данный метод возвращает объект Element, хранящийся в переменной posts. Как описывалось выше, все файлы в дампе имеют схожую структуру с корневым элементом, содержащим имя сущности (например, <tags>, <posts> и т. д.).

Для доступа к отдельным записям (строкам) той или иной сущности можно выполнить обход, начиная с корневого элемента, при помощи обычного цикла for. Строки тоже являются экземплярами класса Element. Преобразование из XML в JSON требует выгрузки атрибутов элемента в словарь; это достигается приведением объекта row.attrib к типу dict. Затем данный словарь можно выгрузить в выходной файл вызовом json.dumps().

Как отмечалось ранее, сущность posts требует особой интерпретации. Когда сценарий stack_xml2json.py применяется для преобразования сообщений, ему нужно передать дополнительный аргумент --clean-post:

```
$ python stack_xml2json.py --xml movies.stackexchange_4.com --json
  movies.posts.jsonl --clean-post
```

Эта команда создаст файл movies.posts.jsonl. Дополнительный флаг --clean-post заставляет сценарий вызвать функцию clean_post(), чтобы выполнить небольшую очистку данных в словаре, содержащем одиночное сообщение. Если точнее, очистке подвергаются атрибуты Tags (для вопросов) и Body (для вопросов и для ответов).

Атрибут Tags – это строка в формате <тег><тег2>...<тегN>, где угловые скобки используются в качестве разделителей тегов. Удаление скобок и добавление пробела между именами тегов упрощают процесс их извлечения в последующем. Например, вопрос о кинофильме может сопровождаться тегами comedy (комедия) и romance (роман). В данном случае до очистки значение атрибута Tags будет иметь вид <comedy><romance>, а после очистки – comedy romance. Эта операция замены заключена в блок try/except, чтобы перехватить исключение KeyError, которое воз-

буждается, когда ключ `Tags` отсутствует в документе, то есть когда обрабатывается ответ (тегами обладают только вопросы).

На следующем шаге в игру включается библиотека BeautifulSoup. С ее помощью сценарий извлекает текст из тела сообщения. Иногда тело сообщения может содержать разметку HTML для форматирования абзаца, которая не нужна для анализа текста вопроса. Метод `get_text()` из библиотеки BeautifulSoup удаляет разметку HTML и возвращает только текст, что нам и требуется.

Классификация текстов по тегам в вопросах

Данный раздел посвящен обучению с учителем. Задача назначения тегов вопросу определяется как задача классификации текстов, и мы применим ее к набору данных с вопросами из сети Stack Exchange.

Прежде чем знакомиться с деталями классификации текстов, рассмотрим следующий вопрос, опубликованный на веб-сайте Stack Exchange «Кинофильмы & ТВ» (заголовок и тело вопроса были объединены):

> *«В какой серии доктор Хаус нанял женщину, чтобы она притворилась мертвой с целью одурачить его команду? Я помню, как женщина (притворившаяся мертвой), просыпается и дает "пять" Хаусу. Из какой это серии?»*

В приведенном вопросе запрашиваются подробности о конкретной серии из популярного сериала «*Доктор Хаус*». Как уже говорилось, вопросы в сети Stack Exchange отмечаются тегами для быстрой идентификации темы вопроса. Этому вопросу пользователь назначил теги `house` и `identify-this-episode`, то есть «идентифицировать эту серию», где первый упоминает название сериала, а второй описывает суть вопроса. Можно было бы возразить и сказать, что для описания темы этого вопроса вполне достаточно одного тега `house`. Но обратите внимание, что эти два тега не исключают друг друга, то есть многочисленные теги помогают лучше представить вопрос.

Процесс назначения меток документам, как описывалось ранее, с целью лучшего понимания темы документа выглядит незамысловатым и является особым случаем давней задачи классификации элементов (реальных объектов, людей, стран, понятий и т. д.). Классификация как область исследования охватывает многочисленные дисциплины, от философии до информатики и управления бизнесом. В настоящей книге классификация рассматривается как задача машинного обучения.

Обучение с учителем и классификация текстов

Обучение с учителем – это область машинного обучения, которая выводит функцию из классифицированных тренировочных данных. Классификация как особый случай обучения с учителем служит цели присваивания правильного класса (метки) новому объекту на основе тренировочного набора объектов, классы которых известны.

Вот примеры практического применения классификации:

- **фильтрация спама**: классификация электронных писем: спам/не спам;
- **определение языка**: автоматическое определение языка текста;
- **идентификация жанра**: автоматическое определение жанра/темы текста.

Приведенные далее примеры описывают различные варианты классификации:

- задача фильтрации спама является случаем бинарной классификации, поскольку даны всего два возможных класса, спам или не спам, из которых должен быть выбран один. Это также случай однозначной классификации, потому что тот или иной документ может быть описан одной и только одной меткой;
- задача определения языка имеет несколько решений. Если предположить, что документ написан на одном языке, задача идентификации языка становится однозначной. С точки зрения числа классов к ней можно подойти как к задаче бинарной классификации (например, этот документ написан на английском языке или нет?), однако ее, вероятно, лучше описать как мультиклассовую задачу, когда число классов больше двух и документ можно отнести к любому из них;
- наконец, задача идентификации жанра (или тематическая классификация) обычно решается как многозначная и мультиклассовая задача: число потенциальных классов больше двух, и тот же самый документ можно отнести к нескольким категориям (как в примере с вопросом о сериале «*Доктор Хаус*»).

Классификация как особый случай обучения с учителем требует тренировочных данных, то есть прецедентов данных с назначенной меткой. Классификатор предусматривает два этапа работы, как показано на рис. 6.4, этап обучения (или тренировки) и этап предсказания.

Рис. 6.4 ❖ Обобщенная схема обучения с учителем

На этапе обучения тренировочные данные передаются на вход алгоритма машинного обучения, МО. Соответственно, на вход алгоритма подается серия пар (объект, метки), где для каждого объекта определена правильная метка (или метки). Этот процесс создает модель классификатора, которая затем помогает предсказывать метки для новых, ранее не встречавшихся данных. На этапе предсказания на вход модели классификации подается новый объект, а на выходе получается метка (либо метки), назначенная(ые) объекту.

В процессе создания модели классификации алгоритм машинного обучения *обращает внимание* на некоторые элементы, чтобы научиться классифицировать документы.

В контексте машинного обучения для представления документов (и объектов в более широком смысле) часто используются n-мерные векторы, каждый элемент которых называют *признаком*. Процедура создания таких векторов из исходных данных обычно называется *выделением признаков*, или *конструированием признаков* (последнее словосочетание чаще используется, когда в данной процедуре используются экспертные знания).

С прагматической точки зрения отбор признаков и выбор способа их представления может оказать огромное влияние на результативность алгоритма машинного обучения. Нередко можно получить довольно достойную результативность путем использования набора очевидных признаков. Зачастую на ранней стадии разработки выполняется итеративный эмпирический (путем проб и ошибок) процесс тестирования разных подходов на основе некоторого интуитивного понимания. В контексте анализа текста в качестве отправной точки обычно принимается подход на основе частот слов.

Чтобы пояснить процедуру преобразования документа в векторы, рассмотрим следующий пример, где имеется корпус из двух документов в исходной и векторизованной форме:

Исходные документы	
Doc1	Джон учится в университете
Doc2	Джон любит программировать. Он также любит Python
Векторизованные документы	
Doc1	[1, 1, 1, 1, 0, 0, 0, 0, 0]
Doc2	[1, 0, 0, 0, 2, 1, 1, 1, 1]
Признаки	[Джон, учится, университете, любит, программировать, он, также, Python]
Тело	Полный текст ответа

Векторизованная версия каждого документа в примере содержит девять элементов. Это число размерностей векторов, то есть число разных слов во всем корпусе. Такое представление не учитывает грамматику и порядок слов в исходном документе, однако порядок элементов в векторах имеет значение. Например, первый элемент в каждом векторе представляет слово «**Джон**», второй элемент – слово «**учится**», пятый элемент – слово «**любит**» и т. д.

Каждое слово представлено частотой, то есть количеством вхождений. Другие широко используемые подходы включают двоичное представление (все ненулевые количества устанавливаются в 1 и просто обозначают простое присутствие слов в документе) либо некую более сложную статистическую величину, такую как вариант статистической меры TF-IDF.

На первый взгляд, задача создания векторов из документов может показаться простой, но в действительности она требует некоторых нетривиальных операций, в том числе лексемизации и нормализации текста.

К счастью, на помощь приходит библиотека **scikit-learn**, включающая несколько инструментов, позволяющих с минимальными усилиями создавать векторы из документов. Основной интерес для нас представляют классы `CountVectorizer`

и TfidfVectorizer. Оба находятся в модуле feature_extraction.text, и их можно импортировать, как показано ниже:

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer
```

Класс CountVectorizer подсчитывает простые частоты и поддерживает возможность создания двоичного представления (по умолчанию атрибут binary имеет значение False), а класс TfidfVectorizer преобразует частоты в оценки TF, TF-IDF или нормализованную оценку TF-IDF. Поведением обоих классов можно управлять множеством аргументов их конструкторов.

Вот пример использования класса TfidfVectorizer:

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> corpus = [
... "Джон учится в университете.",
... "Джон любит программировать. Он также любит Python."
... ]
>>> vectorizer = TfidfVectorizer()
>>> vectors = vectorizer.fit_transform(corpus)
>>> vectors
<2x8 sparse matrix of type '<class 'numpy.float64'>'
        with 9 stored elements in Compressed Sparse Row format>
>>> vectors[0]
<1x8 sparse matrix of type '<class 'numpy.float64'>'
        with 3 stored elements in Compressed Sparse Row format>
>>> vectors[1]
<1x8 sparse matrix of type '<class 'numpy.float64'>'
        with 6 stored elements in Compressed Sparse Row format>
>>> print(vectors)
  (0, 1)        0.449436416524
  (0, 7)        0.631667201738
  (0, 6)        0.631667201738
  (1, 1)        0.243955725
  (1, 2)        0.685742518822
  (1, 4)        0.342871259411
  (1, 3)        0.342871259411
  (1, 5)        0.342871259411
  (1, 0)        0.342871259411
>>> vectorizer.get_feature_names()
['python', 'джон', 'любит', 'он', 'программировать', 'также', 'университете', 'учится']
```

Можно заметить, что в процессе векторизации по умолчанию выполняется лексемизация, как и следовало ожидать, однако слово «**в**» в списке признаков отсутствует (по сути дела, векторы 8-мерные, а не 9-мерные, как изначально объявлено).

Это вызвано тем, что лексемизация по умолчанию выполняется регулярным выражением, которое захватывает все лексемы из двух или более алфавитно-цифровых символов, рассматривая знаки пунктуации и пробелы как разделители лексем. По умолчанию лексемы нормализуются в нижний регистр.

Ниже перечислено несколько интересных атрибутов, оказывающих влияние на TfidfVectorizer:

- `tokenizer`: функция, реализующая этап лексемизации;
- `ngram_range`: кортеж `(min_n, max_n)`, определяющий допустимые размеры n-грамм, когда в качестве лексем требуется рассматривать не только отдельные слова, но и словосочетания;
- `stop_words`: список стоп-слов для удаления (по умолчанию удаление стоп-слов не происходит);
- `lowercase`: логическое значение, по умолчанию равно `True`;
- `min_df`: минимальный порог частоты документа; может быть целым числом (например, для порога 5 лексема будет включена, только если она присутствует в 5 документах и более) или вещественным, в интервале [0.0, 1.0], определяющим долю документов. По умолчанию имеет значение `1`, то есть порог отсутствует;
- `max_df`: максимальный порог частоты документа; как и `min_df`, может быть целым или вещественным числом. По умолчанию имеет значение `1.0` (100% документов), то есть порог отсутствует;
- `binary`: логическое значение, по умолчанию равно `False`. Если этот атрибут имеет значение `True`, все ненулевые количества термина устанавливаются в 1, то есть компонент TF меры TF-IDF носит двоичный характер, но окончательное значение по-прежнему определяется оценкой IDF;
- `use_idf`: логическое значение, по умолчанию равно `True`. Если этот атрибут имеет значение `False`, взвешивание с участием меры IDF не выполняется;
- `smooth_idf`: логическое значение, по умолчанию равно `True`. Включает сглаживание частот документов по схеме «*плюс один*», чтобы избежать деления на ноль;
- `sublinear_tf`: логическое значение, по умолчанию равно `False`. Если этот атрибут имеет значение `True`, применяется сублинейное масштабирование оценки TF, то есть взамен TF используется результат выражения *1+log(TF)*.

Следующий раздел описывает три общих подхода к классификации, реализованных в библиотеке scikit-learn в готовом виде.

Алгоритмы классификации

В этом разделе мы в общих чертах обрисуем три подхода к машинному обучению, которые можно использовать для классификации текстов: **наивный байесовский** (Naïve Bayes, NB), **k ближайших соседей** (k-Nearest Neighbor, k-NN) и **метод опорных векторов** (Support Vector Machine, SVM). Наша цель – не окунуться в детали алгоритмов, а просто познакомиться с инструментами, уже имеющимися в библиотеке scikit-learn, которые можно использовать прямо сейчас. При этом стоит заглянуть в документацию к библиотеке scikit-learn, так как она предлагает обширный набор алгоритмов (в том числе и алгоритмов обучения с учителем: http://scikit-learn.org/stable/supervised_learning.html).

Важно помнить, что разные алгоритмы могут показывать разную производительность с разными данными, поэтому во время разработки системы классификации желательно протестировать разные реализации. Предлагаемый библиотекой scikit-learn интерфейс существенно облегчает нашу жизнь, позволяя менять алгоритмы путем простой замены одной-двух строк программного кода.

Наивный байесовский алгоритм

Начнем с так называемого «наивного байесовского алгоритма» – семейства классификаторов на основе теоремы Байеса (https://ru.wikipedia.org/wiki/Теорема_Байеса) с (наивным) допущением независимости признаков.

Теорема Байеса (также известная как закон, или правило, Байеса) описывает условную вероятность события:

$$P(A / B) = \frac{P(B \mid A)P(A)}{P(B)}.$$

Член слева – это вероятность наблюдения события A при условии, что наблюдалось событие B.

Чтобы поместить это уравнение в контекст, давайте рассмотрим пример с фильтрацией спама:

$$P(\text{Спам}/\text{"деньги"}) = \frac{P(\text{"деньги"}/\text{Спам})P(\text{Спам})}{P(\text{"деньги"})}.$$

Данное уравнение показывает, как вычислить вероятность назначения метки *Спам* (новому, ранее не встречавшемуся) документу, содержащему слово «*деньги*». С точки зрения условной вероятности вопрос часто формулируется так: какова вероятность назначения метки *Спам* этому документу при условии присутствия слова «*деньги*»?

Используя тренировочные данные, три вероятности справа можно вычислить во время этапа обучения.

Поскольку классификатор не использует отдельного слова/признака для назначения конкретного класса документу, рассмотрим следующий общий случай:

$$P(c \mid d) = \frac{P(d \mid c)P(c)}{P(d)} \propto P(c)\prod_{t \in d} P(t \mid c).$$

Это уравнение описывает вероятность присваивания класса c наблюдаемому документу d. Согласно допущению о независимости признаков, документ описывается как последовательность независимых терминов t. Поскольку признаки (и, следовательно, их вероятности) независимы, вероятность $P(d|c)$ можно записать как произведение вероятностей $P(t|c)$ всех терминов t в документе d.

Внимательный читатель, вероятно, обратил внимание на отсутствие знаменателя в последней части последнего уравнения, где документ выражается как последовательность независимых терминов. Дело в том, что это просто оптимизация: вероятность $P(d)$ будет одинаковой по всем классам, следовательно, ее не нужно вычислять, чтобы вычислить и сравнить отдельные оценки. По этой причине в последней части уравнения используется символ *прямой пропорциональной зависимости*, а не символ *эквивалентности*.

Мы не будем дальше углубляться в математические основы, поскольку не должны терять из виду практических аспектов книги. Заинтересованный читатель может самостоятельно продолжить изучение этой темы (например, в книге «*Python Machine Learning*» Себастьяна Рашки (Sebastian Raschka), выпущенной издательством Packt Publishing[1]).

---

[1] *Рашка С.* Python и машинное обучение. М.: ДМК Пресс, 2017. ISBN: 978-5-97060-409-0. – *Прим. ред.*

Модуль `naive_bayes` в библиотеке scikit-learn предлагает поддержку наивного байесовского метода в виде полиномиального наивного байесовского классификатора, обычно используемого в классификации текстов, реализованного классом `MultinomialNB`:

```
from sklearn.naive_bayes import MultinomialNB
```

k ближайших соседей

Алгоритм k-NN (https://ru.wikipedia.org/wiki/Метод_k-ближайших_соседей) – это метод, который используется для классификации и регрессионного анализа. На вход алгоритма подаются `k` самых близких тренировочных примеров в пространстве признаков, то есть `k` наиболее похожих документов. На выходе алгоритм возвращает класс, назначенный конкретному документу, полученный большинством голосов его соседей.

Во время тренировочного этапа многомерные векторы, представляющие документы в тренировочном наборе данных, просто хранятся со своими метками. Во время этапа предсказания ранее не встречавшийся документ используется для опроса `k` наиболее похожих на него документов, где `k` – заданная пользователем константа. Ранее не встречавшемуся документу присваивается класс, наиболее типичный по всем этим `k` документам (ближайшим соседям).

Алгоритм k-NN принадлежит категории непараметрических методов, потому что решение алгоритма диктуется непосредственно данными (то есть векторами документов), а не параметром, оцененным на данных (как в наивном байесовском алгоритме, где назначением классов управляют оценочные вероятности).

В основе данного алгоритма лежит простое логическое соображение, но он оставляет нерешенными два вопроса:

- как вычислить расстояние между векторами?
- как выбрать лучшее значение `k`?

Первый вопрос имеет разные допустимые ответы. Чаще всего выбирают эвклидово расстояние (https://ru.wikipedia.org/wiki/Евклидова_метрика). Реализация данного алгоритма в библиотеке scikit-learn по умолчанию использует расстояние Минковского, которое эквивалентно эвклидову, когда используются заранее заданные параметры.

Ответ на второй вопрос немного сложнее и может зависеть от данных. Выбор малого `k` означает большее влияние шума. Большое значение `k` делает алгоритм более дорогостоящим в вычислительном плане. Часто в качестве `k` выбирается квадратный корень из числа признаков `n`, то есть `k = sqrt(n)`; однако в обязательном порядке следует поэкспериментировать.

Реализация алгоритма классификации k-NN доступна в модуле `neighbors` библиотеки scikit-learn:

```
from sklearn.neighbors import KneighborsClassifier
```

Конструктор класса `KneighborsClassifier` принимает ряд аргументов, включая `k` и `distance`. Что касается вариантов для разных метрик расстояния, все они описаны в документации к классу `sklearn.neighbors.DistanceMetric`.

Метод опорных векторов

Метод опорных векторов (SVM) (https://ru.wikipedia.org/wiki/Метод_опорных_векторов) – это набор моделей обучения с учителем и связанных с ними алгоритмов классификации и регрессионного анализа.

Алгоритм SVM применяет подход на основе бинарной классификации, то есть ранее не встречавшиеся документы относятся к одному из двух возможных классов. Данную методику можно распространить на случай многоклассовой классификации, используя один из двух подходов: «один против всех», когда тот или иной класс сравнивается с собирательным классом, или «один против одного», когда рассматриваются все возможные пары или классы. В методе «один против одного» создаются `n` классификаторов, где `n` – общее количество классов. Документу назначается класс с самой высокой выходной функцией. В методе «один против одного» каждый классификатор увеличивает счетчик очков выбранного класса. В конце документу назначается класс, получивший наибольшее количество голосов.

Семейство классификаторов SVM известно своей эффективностью в задачах классификации текстов, где данные обычно представлены в высокоразмерном пространстве. В случаях, когда число размерностей больше числа прецедентов, классификатор по-прежнему может оставаться эффективным. Если число признаков намного больше числа прецедентов (то есть в тренировочном наборе мало документов), данный метод может потерять свою эффективность.

Его реализация также доступна в библиотеке scikit-learn (в модуле `svm`):

```
from sklearn.svm import LinearSVC
```

Класс `LinearSVC` реализует линейную основу, один из вариантов базовой функции. Грубо говоря, базовую функцию можно рассматривать как функцию подобия, используемую для вычисления подобия между парой прецедентов. Библиотека scikit-learn предлагает несколько базовых функций. Имеется также возможность задействовать пользовательскую реализацию. Экземпляр класса `SVC` можно создать с параметром `kernel='linear'`; хотя этот классификатор тоже основан на линейной базовой функции, его реализация отличается – класс SVC основан на динамической библиотеке `libSVM` общего назначения, в то время как класс `LinearSVC` основан на более эффективной реализации, оптимизированной для линейной базовой функции и динамической библиотеки `liblinear`.

Оценка

После обсуждения разных алгоритмов классификации в этом разделе мы исследуем интересный вопрос: как выбрать наилучший алгоритм?

В контексте классификации существуют четыре разные категории, характеризующие исход:

- документ принадлежит классу *C*, и система относит его к классу *C* (**истинно положительный** исход, или **ИП**);
- документ не принадлежит классу *C*, но система относит его к классу *C* (**ложноположительный** исход, или **ЛП**);
- документ принадлежит классу *C*, но система не относит его к классу *C* (**ложноотрицательный** исход, или **ЛО**);
- документ не принадлежит классу *C*, и система не относит его к классу *C* (**истинно отрицательный** исход, или **ИО**).

Эти исходы обычно представляются в виде таблицы, которая называется матрицей ошибок. Вот как выглядит матрица ошибок для случая бинарной классификации:

	Предсказанная метка: С	**Предсказанная метка: не С**
Правильная метка: С	ИП	ЛО
Правильная метка: не С	ЛП	ИО

Матрицу ошибок можно расширить для случая классификации с несколькими классами. Имея исходы для всех прецедентов в тестовом наборе, можно вычислить различные метрики результативности. Вот некоторые из них:

- **точность**: как часто классификатор дает правильный результат?

  (*ИП* + *ИО*) / *общее количество*;

- **коэффициент ошибочной классификации**: как часто классификатор дает неправильный результат?

  (*ЛП* + *ЛО*) / *общее количество*;

- **прецизионность**: как часто классификатор дает правильный результат при назначении класса С?

  ИП / число прогнозов для класса С;

- **полнота**: какая доля документов в *С* идентифицирована правильно?

  ИП / число меток в С.

Все эти метрики имеют значение в диапазоне между 0 и 1, которое можно интерпретировать как долю. Прецизионность и полнота часто объединяются в одно значение, которое называется средним гармоническим (а также *отметкой F* или *F1*).

В контексте классификации с несколькими классами отметки F по всем классам могут усредняться разными способами. Обычно используются подходы *микро-F1*, когда каждому решению дается в среднем равный вес, и *макро-F1*, когда каждому классу дается в среднем равный вес (*Machine Learning in Automated Text Categorization, Fabrizio Sebastiani, 2002*).

До сих пор, с точки зрения оценки результативности, предполагалось, что имеется четкое разграничение между тренировочными и тестовыми данными (набор данных, на котором выполняется оценка результативности). Некоторые хорошо известные наборы данных сопровождаются четкой аннотацией о разделении по схеме «тренировка/тестирование», чтобы исследователи могли воспроизвести результаты. Иногда это разделение неочевидно, поэтому нам нужно зарезервировать долю аннотированных данных для тестирования. Небольшой тестовый набор означает, что оценка может оказаться неточной, в то время как большой тестовый набор означает меньше данных для тренировочного этапа.

Одно из возможных решений данной проблемы называется *перекрестной проверкой*, при этом наиболее распространенный подход называется *k-кратной проверкой*. Идея такой проверки состоит в том, чтобы повторять оценивание определенное число раз, используя всякий раз разное разделение «тренировка/тестирование», и затем вернуть агрегированную оценку (например, среднее арифметическое).

Например, при 10-кратной проверке можно разделить набор данных на десять уникальных подмножеств, или блоков. В этом случае последовательно выбирает-

ся один из блоков и используется в качестве тестового набора, а все оставшиеся объединяются в тренировочный набор. После оценки на первом блоке откладываем результаты в сторону и переходим к следующей итерации. Теперь тестовым набором становится второй блок, а другие девять блоков используются в качестве тренировочных данных. После 10-й итерации все оценки можно усреднить и получить единственный результат. Часто в методике k-кратной проверки используются 10-кратная (разделение «тренировка/тестирование» – 90/10) и 5-кратная (разделение «тренировка/тестирование» – 80/20) проверки. Главные преимущества использования перекрестной проверки состоят в том, что наша уверенность в точности оценки результативности повышается даже в условиях небольших наборов данных.

Классификация текстов на данных из сети Stack Exchange

После длинного введения в классификацию, связанные с ней алгоритмы и оценку результативности в данном разделе мы применим новые знания для классификации набора вопросов, опубликованных на веб-сайте «Кинофильмы и ТВ».

Поскольку для представления классов мы используем имена тегов, выполним подготовительную обработку набора данных, чтобы исключить наиболее редкие классы, которые очень трудно проверить. Установим порог равным 10, то есть теги, появляющиеся менее десяти раз, будут игнорироваться. Этот порог выбран произвольно, и вы можете поэкспериментировать с различными числами. Кроме того, файл сообщений содержит не только вопросы, но и ответы, однако здесь мы рассматриваем только вопросы (распознаются по атрибуту `PostTypeId="1"`), поэтому все ответы будем отбрасывать.

Следующий сценарий читает файлы JSON Lines с тегами и сообщениями и создает файл JSON Lines только с тегами, которые встречаются достаточно часто, согласно нашему порогу:

```
# Chap06/stack_classification_prepare_dataset.py
import os
import json
from argparse import ArgumentParser

def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--posts-file')
    parser.add_argument('--tags-file')
    parser.add_argument('--output')
    parser.add_argument('--min-df', type=int, default=10)
    return parser

if __name__ == '__main__':
    parser = get_parser()
    args = parser.parse_args()

    valid_tags = []
    with open(args.tags_file, 'r') as f:
        for line in f:
            tag = json.loads(line)
            if int(tag['Count']) >= args.min_df:
                valid_tags.append(tag['TagName'])
```

```
    with open(args.posts_file, 'r') as fin, open(args.output, 'w') as fout:
        for line in fin:
            doc = json.loads(line)
            if doc['PostTypeId'] == '1':
                doc_tags = doc['Tags'].split(' ')
                tags_to_store = [tag for tag in doc_tags
                                 if tag in valid_tags]
                if tags_to_store:
                    doc['Tags'] = ' '.join(tags_to_store)
                    fout.write("{}\n".format(json.dumps(doc)))
```

Данный сценарий предполагает, что мы уже преобразовали файлы с тегами и сообщениями из XML в JSON Lines, как было описано ранее.

Этот сценарий можно запустить командой:

```
$ python stack_classification_prepare_dataset.py \
  --tags-file movies.tags.jsonl \
  --posts-file movies.posts.jsonl \
  --output movies.questions4classification.jsonl \
  --min-df 10
```

Получившийся в результате файл `movies.questions4classification.jsonl` содержит вопросы только с часто встречающимися тегами. Сразу же бросается в глаза разница в размерах с исходным файлом сообщений:

```
$ wc -l movies.posts.jsonl
  28146 movies.posts.jsonl
$ wc -l movies.questions4classification.jsonl
  8413 movies.questions4classification.jsonl
```

Теперь мы готовы выполнить эксперимент на наборе данных с сайта «Кинофильмы и ТВ». По горячим следам обсуждения процедуры оценки результативности выполним десятикратную перекрестную проверку, разбив корпус вопросов на десять равных блоков и выполнив классификацию десять раз, используя 90% данных для тренировки и оставшиеся 10% для тестирования, причем для тестирования каждый раз будет использоваться другой блок:

```
# Chap06/stack_classification_predict_tags.py
import json
from argparse import ArgumentParser
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import LinearSVC
from sklearn.multiclass import OneVsRestClassifier
from sklearn.metrics import f1_score
from sklearn.preprocessing import MultiLabelBinarizer
from sklearn.cross_validation import cross_val_score
import numpy as np

def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--questions')
    parser.add_argument('--max-df', default=1.0, type=float)
    parser.add_argument('--min-df', default=1, type=int)
    return parser
```

```
if __name__ == '__main__':
    parser = get_parser()
    args = parser.parse_args()

    stop_list = stopwords.words('english')

    all_questions = []
    all_labels = []
    with open(args.questions, 'r') as f:
        for line in f:
            doc = json.loads(line)
            question = "{} {}".format(doc['Title'], doc['Body'])
            all_questions.append(question)
            all_labels.append(doc['Tags'].split(' '))

    vectorizer = TfidfVectorizer(min_df=args.min_df,
                                 stop_words=stop_list,
                                 max_df=args.max_df)
    X = vectorizer.fit_transform(all_questions)
    mlb = MultiLabelBinarizer()
    y = mlb.fit_transform(all_labels)
    classifier = OneVsRestClassifier(LinearSVC())

    scores = cross_val_score(classifier, X, y=y, cv=10, scoring='f1_micro')
    print("Средняя оценка F1: {}".format(np.mean(scores)))
```

Сценарий `stack_classification_predict_tags.py` принимает три аргумента командной строки, используя класс `ArgumentParser`. Первый и единственный обязательный аргумент `--questions` определяет имя файла `.jsonl` с вопросами. Другие два аргумента, `--min-df` и `--max-df`, позволяют влиять на поведение классификатора с точки зрения создания векторов признаков. Значения по умолчанию этих двух аргументов не влияют на процесс выделения признаков, так как пределы установлены в `min_df=1` и `max_df=1.0`, то есть включаются все признаки (слова).

Данный сценарий можно запустить командой:

```
$ python stack_classification_predict_tags.py \
  --questions movies.questions4classification.jsonl
```

Она выведет:

```
Средняя оценка F1: 0.6271980062798452
```

Рассмотрим данный сценарий подробнее. После анализа аргументов командной строки он загружает список стоп-слов в переменную `stop_list`, используя библиотеку NLTK. Это список общеупотребительных английских слов (приблизительно 130 слов, включая артикли, союзы, местоимения и т. д.), используемый по умолчанию. На следующем этапе выполняется чтение входного файла с загрузкой каждого документа JSON в оперативную память и созданием структур данных, которые будут поданы в классификатор. В частности, в переменной `all_questions` создается список всех документов путем объединения заголовков и тел всех вопросов. Затем в переменной `all_labels` создается словарь для отслеживания исходных тегов (меток/классов для классификатора).

Далее следует этап выделения признаков, на котором создается экземпляр класса `TfidfVectorizer` с передачей конструктору списка стоп-слов, а также мини-

мальной и максимальной частот документа. Затем создаются векторы признаков для классификатора: каждое слово в исходном тексте преобразуется в численную оценку TF-IDF. Стоп-слова в векторы не включаются. Точно так же не включаются слова с частотой за пределами заданного диапазона.

Преобразование исходного текста в векторы выполняется методом `fit_transform()`.

Поскольку наша задача является многозначной, необходимо выполнить еще один этап, чтобы правильно отобразить все метки в двоичные векторы. Это необходимо, потому что классификатор на входе ожидает список двоичных векторов, а не список имен классов. Чтобы понять, как `MultiLabelBinarizer` выполняет это преобразование, исследуем следующий фрагмент:

```
>>> from sklearn.preprocessing import MultiLabelBinarizer
>>> mlb = MultiLabelBinarizer()
>>> labels = [['house', 'drama'], ['star-wars'], ['drama', 'uk']]
>>> y = mlb.fit_transform(labels)
>>> y
array([[1, 1, 0, 0],
       [0, 0, 1, 0],
       [1, 0, 0, 1]])
>>> mlb.classes_
array(['drama', 'house', 'star-wars', 'uk'], dtype=object)
```

В этой точке корпус документов и метки находятся в формате, понятном классификатору. Примечание по поводу наименований: использование заглавной `X` для корпуса и строчной `y` для меток является общепринятым в машинном обучении написанием, которое можно часто найти в учебниках по машинному обучению. Если вам не нравятся однобуквенные переменные, можете их переименовать.

На роль классификатора мы выбрали классификатор, реализующий метод опорных векторов `LinearSVC` с линейной базовой функцией. Вы можете поэкспериментировать со множеством других классификаторов. Поскольку данная задача многозначная, выберем подход «один против всех», реализованный классом `OneVsRestClassifier`, конструктор которого в первом аргументе принимает экземпляр фактического классификатора и использует его для решения задачи классификации, как описано в предыдущем разделе.

На заключительном этапе выполняется сама классификация. Поскольку решено выполнить перекрестную проверку, как отмечалось ранее, мы не используем объект классификатора непосредственно, а передаем его в функцию `cross_val_score()` вместе с векторизованными данными, метками и еще несколькими аргументами. Цель этой функции состоит в том, чтобы выполнить обход данных согласно k-кратной стратегии, объявленной в аргументе `cv=10` (иными словами, мы используем десять блоков данных), выполнить классификацию и наконец показать оценки для выбранной оценочной метрики (в приведенном примере мы используем микроусредненную оценку F1).

Функция перекрестной проверки возвращает массив NumPy из оценок, по одной для каждого блока. При помощи функции `numpy.mean()` мы можем усреднить их и вывести их среднее арифметическое.

Стоит отметить, что итоговая оценка, приблизительно `0.62`, *сама по себе* ни хорошая, ни плохая. Она говорит, что система классификации далеко не идеальна,

но на данной стадии у нас нет четкой картины о том, есть ли место для ее повышения. Оценка не говорит нам, плох ли классификатор или задача является слишком трудной, чтобы достигнуть улучшения в работе. Усредненная оценка просто помогает сравнить поведение разных классификаторов или разных прогонов одного и того же классификатора с разными параметрами.

В качестве дополнительного примера можно просто запустить этот сценарий еще раз, указав дополнительный аргумент для фильтрации редко встречающихся признаков:

```
$ python stack_classification_predict_tags.py \
  --questions movies.questions4classification.jsonl \
  --min-df 5
```

Установив минимальную частоту документа равной 5, которая просто «подрезает» длинный хвост в частотном распределении, мы получим следующий результат:

```
Средняя отметка F1: 0.635184845312832
```

Данный пример показывает, что простая доводка конфигурации может привести к другим (в данном случае улучшенным) результатам.

Встраивание классификатора в приложение реального времени

В предыдущем разделе мы рассмотрели чисто академический подход к задаче классификации: начав с набора предварительно классифицированных документов, мы выполнили эксперимент, произведя перекрестную проверку для получения оценочной метрики. Это, конечно, очень интересно, только если мы хотим всего лишь поэкспериментировать с разными классификаторами, тонко настроить процесс выделения признаков и понять, какой классификатор и какая конфигурация работают на данном наборе данных лучше всего.

Данный раздел продвигает нас немного дальше и обсуждает некоторые простые шаги для встраивания классификатора (или, в более широком смысле, модели машинного обучения) в приложение, взаимодействующее с пользователем в режиме реального времени. Популярные примеры таких приложений включают поисковые системы, рекомендательные системы, спам-фильтры и многие другие интеллектуальные приложения, которые мы, возможно, используем ежедневно, не представляя, как они приводятся в действие методами машинного обучения.

В этом разделе мы создадим приложение, выполняющее следующие шаги:

- тренировка классификатора (этап обучения на локальном наборе данных);
- сохранение классификатора на диске, чтобы его можно было загрузить в приложение реального времени;
- встраивание классификатора в приложение, взаимодействующее с пользователем (этап предсказания в режиме реального времени).

Первые два этапа выполняет следующий сценарий:

```
# Chap06/stack_classification_save_model.py
import json
import pickle
from datetime import datetime
from argparse import ArgumentParser
```

```
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import LinearSVC
from sklearn.multiclass import OneVsRestClassifier
from sklearn.preprocessing import MultiLabelBinarizer

def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--questions')
    parser.add_argument('--output')
    parser.add_argument('--max-df', default=1.0, type=float)
    parser.add_argument('--min-df', default=1, type=int)
    return parser

if __name__ == '__main__':
    parser = get_parser()
    args = parser.parse_args()

    stop_list = stopwords.words('english')

    all_questions = []
    all_labels = []
    with open(args.questions, 'r') as f:
        for line in f:
            doc = json.loads(line)
            question = "{} {}".format(doc['Title'], doc['Body'])
            all_questions.append(question)
            all_labels.append(doc['Tags'].split(' '))

    vectorizer = TfidfVectorizer(min_df=args.min_df,
                                 stop_words=stop_list,
                                 max_df=args.max_df)
    X = vectorizer.fit_transform(all_questions)
    mlb = MultiLabelBinarizer()
    y = mlb.fit_transform(all_labels)

    classifier = OneVsRestClassifier(LinearSVC())

    classifier.fit(X, y)
    model_to_save = {
        'classifier': classifier,
        'vectorizer': vectorizer,
        'mlb': mlb,
        'created_at': datetime.today().isoformat()
    }
    with open(args.output, 'wb') as f:
        pickle.dump(model_to_save, f)
```

Он очень похож на сценарий классификации из предыдущего раздела. Основное различие состоит в том, что после тренировки классификатора мы не используем его для предсказания чего бы то ни было, а просто сохраняем в файле.

Выходной файл открывается в режиме `wb`, то есть для записи двоичных данных. Пакет pickle управляет сериализацией сложного объекта Python в байт-код, который затем может быть выгружен в файл.

Интерфейс данного модуля очень похож на интерфейс пакета `json` и располагает простыми функциями `load()` и `dump()`, которые читают/записывают объекты

Python в файлы. Нам понадобится не только классификатор, но также векторизатор и экземпляр класса `MultiLabelBinarizer`. Поэтому мы используем переменную `model_to_save` – словарь для включения всех необходимых данных. Ключ `created_at` в данном словаре хранит текущую системную дату в формате ISO 8601 на случай, если потребуется отслеживать различные версии классификатора либо имеются новые данные и мы решаем переобучить модель.

Данный сценарий можно запустить командой:

```
$ python stack_classification_save_model.py \
  --questions movies.questions4classification.jsonl \
  --min-df 5 \
  --output questions-svm-classifier.pickle
```

Через несколько секунд будет создан файл `questions-svm-classifier.pickle`, готовый к использованию другими приложениями.

Ради простоты мы реализуем элементарный интерфейс командной строки. Тот же подход можно легко интегрировать в веб-приложение (например, используя микрофреймворк **Flask**, как мы сделали в предыдущей главе), чтобы предложить пользователям возможность прогнозирования через веб-интерфейс.

Сценарий `stack_classification_user_input.py` реализует приложение с интерфейсом пользователя:

```
# Chap06/stack_classification_ui.py
import sys
import pickle
from argparse import ArgumentParser

def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--model')
    return parser

def exit():
    print("До свидания.")
    sys.exit()

if __name__ == '__main__':
    parser = get_parser()
    args = parser.parse_args()

    with open(args.model, 'rb') as f:
        model = pickle.load(f)
    classifier = model['classifier']
    vectorizer = model['vectorizer']
    mlb = model['mlb']

    while True:
        print("Введите вопрос или \"выход\", чтобы завершить.")
        user_input = input('> ')
        if user_input == 'выход':
            exit()
        else:
            X = vectorizer.transform([user_input])
            print("Вопрос: {}".format(user_input))
```

```
prediction = classifier.predict(X)
labels = mlb.inverse_transform(prediction)[0]
labels = ', '.join(labels)
if labels:
    print("Предсказанные метки: {}".format(labels))
else:
    print("Для данного вопроса метки отсутствуют")
```

Для чтения аргумента `--model` командной строки он использует экземпляр класса `ArgumentParser`. В этом аргументе передается сериализованный файл `.pickle`.

Сценарий загружает файл, используя функции из пакета pickle. Файл хранит двоичные данные, поэтому он открывается в режиме `rb` (`r` – режим чтения (read) и `b` – двоичные данные (binary)). Это позволяет получить доступ к ранее созданным классификатору, векторизатору и многозначному бинаризатору.

Затем приложение входит в бесконечный цикл `while True`, запрашивающий у пользователя ввод некоторых входных данных. Если выполнить команду:

```
$ python stack_classification_user_input.py \
  --model questions-svm-classifier.pickle
```

то на экране появится следующее приглашение:

```
Введите вопрос или "выход", чтобы завершить.
>
```

Когда пользователь введет какой-то текст, функция `input()` прочитает его в переменную `user_input`. Затем ее содержимое сравнивается со словом `выход`. Если пользователь запросил выход, приложение завершается. Иначе выполняется классификация текста, введенного пользователем.

Этап предсказания требует преобразования входных данных в векторы признаков. Данная работа выполняется векторизатором. Важно отметить, что функция `vectorizer.transform()` принимает список (или любую итерируемую последовательность), а не строковое значение `user_input` непосредственно. Результат векторизации входных данных передается в функцию `predict()` классификатора, и начинается процесс классификации.

Затем результат классификации в читаемом формате возвращается пользователю. Предсказание на данной стадии имеет форму массива NumPy с двоичной маской для списка допустимых меток. Экземпляр `MultiLabelBinarizer`, отвечающий за создание этой маски на этапе тренировки, может отобразить ее на исходные метки, используя функцию `inverse_transform()`.

В результате этого вызова получается список кортежей. Точнее, поскольку мы передавали для классификации всего один документ, данный список включает всего один кортеж (поэтому мы использовали индекс `[0]` во время обратного преобразования).

Прежде чем показать результаты пользователю, элементы кортежа с метками объединяются в строку, если он не пустой (иначе выводится специальное сообщение).

Теперь можно посмотреть, как действует классификатор:

```
$ python stack_classification_user_input.py \
  --model questions-svm-classifier.pickle
Введите вопрос или "выход", чтобы завершить.
```

```
> What's up with Gandalf and Frodo lately? They haven't been in the Shire for a while...
Вопрос: What's up with Gandalf and Frodo lately? They haven't been in the Shire for a while...
Предсказанные метки: plot-explanation, the-lord-of-the-rings
Введите вопрос или "выход", чтобы завершить.
> What's the title of the latest Star Wars movie?
Вопрос: What's the title of the latest Star Wars movie?
Предсказанные метки: identify-this-movie, star-wars, title
Введите вопрос или "выход", чтобы завершить.
> How old is Tom Cruise?
Вопрос: How old is Tom Cruise?
Для данного вопроса метки отсутствуют
Введите вопрос или "выход", чтобы завершить.
> How old is Brad Pitt?
Вопрос: How old is Brad Pitt?
Предсказанные метки: character, plot-explanation
Введите вопрос или "выход", чтобы завершить.
> выход
До свидания.
```

После запуска сценарий предлагает ввести вопрос. Выделенные строки – это примеры ввода данных пользователем, перемежающиеся откликом приложения.

Примечательно, что классификатор распознает некоторые имена и названия из фильма «*Властелин колец*» (Фродо (Frodo), Гэндальф (Gandalf) и Графство (Shire)) и правильно присваивает метки вопросу, в котором отсутствует упоминание названия фильма. По некоторым причинам вопрос тоже интерпретируется как запрос на объяснение сюжета. Точнее, признаки вопросов вынуждают классификатор отметить этот документ как принадлежащий классу `plot-explanation`, то есть запрос на объяснение сюжета.

Второй ввод содержит словосочетание «Star Wars» («Звездные войны»), поэтому идентифицировать эту метку, как и ожидалось, не составило труда. Вопрос также содержит просьбу сообщить название, поэтому ему присваиваются два дополнительных класса, `title` и `identify-this-film` (идентифицировать этот фильм). Пока классификация выглядит довольно точной.

Последние два вопроса показывают, что классификатор не всегда идеален. Вопросу о Томе Крузе (Tom Cruise) классификатор не присваивает ни одной метки. Это, вероятно, вызвано отсутствием значимых признаков в тренировочных данных, а именно слов `tom` и `cruis`. Ситуация с Брэдом Питтом (Brad Pitt) выглядит немного лучше – вопросу присвоены метки `character`, то есть «главный герой» (немного натянуто) и `plot-explanation` (просто неправильно).

Пример завершается вводом слова `выход`, которое заставляет приложение закончить работу.

Эксперименты в предыдущих разделах показали, что точность классификатора далека от идеальной. С этим приложением реального времени мы перво-наперво протестировали способности классификатора. Его легко можно одурачить расплывчатыми вопросами или просто не встречавшимися ранее словами, но главной целью данного раздела было не это, а показать, что при проведении проверки наших моделей мы не ограничены академическими экспериментами. Интеграция модели машинного обучения в настоящее приложение является удивительно простой задачей, в особенности с изящным и простым интерфейсом Python.

Примеры применения классификации в режиме реального времени можно перечислять до бесконечности:

- анализ тональности текста для автоматической идентификации отношения пользователя и его отображение;
- фильтрация для блокировки спама или фильтрации сообщений, похожих на спам;
- обнаружение ненормативной лексики для идентификации пользователей, посылающих оскорбительные сообщения, и автоматического удаления таких сообщений.

Заинтересованного читателя приглашаем снова воспользоваться тем, что узнали о микрофреймворке Flask в главе 5 «*Тематический анализ в Google+*», и расширить на его основе простое веб-приложение, загружающее модель машинного обучения и использующее ее для классификации вопросов пользователя в режиме реального времени.

Резюме

Настоящая глава познакомила с одним из самых популярных применений Всемирной паутины – вопросно-ответными приложениями. Популярность сети Stack Exchange и веб-сайта Stack Overflow для программистов обусловлена высококачественным материалом, курируемым сообществом. В данной главе мы обсудили приемы взаимодействий с программным интерфейсом Stack Exchange API и узнали, как получить доступ ко всему набору данных в сети Stack Exchange, используя дампы.

Вторая часть главы познакомила с задачей классификации и связанными с ней подходами к машинному обучению с учителем. Наличие классифицированных данных в сети Stack Exchange позволяет создавать модели прогнозирования. Одна из таких моделей была продемонстрирована в данной главе. Ее главная задача – предсказание меток, которые будут присвоены вопросу, но представленные здесь методы можно применить для решения множества других практических задач. Заключительная часть главы расширила обсуждение, показав, насколько просто интегрировать модель машинного обучения в приложение реального времени.

В центре внимания следующей главы будут блоги. В ней, в частности, мы применим методы обработки естественного языка.

Глава 7

Блоги, RSS, Википедия и обработка естественного языка

Основная тема настоящей главы: **обработка естественного языка**, область исследования, связанная с естественным языком. Прежде чем углубиться в подробности, мы проанализируем некоторые возможности по загрузке текстовых данных из веба.

Эта глава охватывает следующие темы:

- взаимодействие с WordPress.com API и Blogger API;
- форматы веб-лент новостей (RSS и Atom) и их использование;
- сохранение данных из блогов в формате JSON;
- взаимодействие с Wikipedia API и поиск информации о сущностях;
- основные понятия обработки естественного языка и особенно предварительной обработки текста;
- обработка текстовых данных с целью идентификации сущностей, упоминаемых в тексте.

Блоги и обработка естественного языка

Блоги (сокращенно weblog, то есть «интернет-журнал событий») являются в наше время важной частью Всемирной паутины и невероятно привлекательной социально-медийной платформой. Блоги используются компаниями, профессионалами и любителями для общения с аудиторией, рекламирования продуктов и услуг или просто обсуждения интересных тем. Благодаря изобилию инструментов и служб, помогающих пользователям, не обладающим специальными знаниями, публиковать свою информацию, настройка персонального блога является делом нескольких минут.

С точки зрения аналитика блоги – отличная платформа для оттачивания практических навыков в анализе текстов. Настоящая глава охватывает две главные темы: получение текстовых данных из блогов и применение методов обработки естественного языка к таким данным.

Обработка естественного языка – многосложная область, освещение которой может потребовать больше одной книги, поэтому мы предпримем прагматический подход и представим основы теории на практических примерах.

Получение данных из блогов и веб-сайтов

Учитывая изобилие веб-сайтов с интересными статьями, поиск текстовых данных для анализа не представляет серьезной проблемы. Сохранение статей вручную плохо масштабируется, поэтому в данном разделе мы обсудим некоторые возможности автоматизации процесса извлечения данных из веб-сайтов.

Но прежде обсудим две популярные бесплатные службы блогинга – WordPress.com и Blogger, – предлагающие удобный программный интерфейс. Затем представим веб-стандарты RSS и Atom, используемые многими блогами и издателями новостей для распространения своего материала в формате, который легко читается компьютером. Наконец, мы кратко обсудим другие возможные варианты, такие как подключение к Википедии и методы извлечения данных из веба, так называемый веб-скрапинг (web scraping).

WordPress.com API

WordPress.com – поставщик услуг по размещению и хранению информации в вебе, включая блоги (то есть услуг блог-хостинга и веб-хостинга), который работает на открытом программном обеспечении WordPress. Данная служба предоставляет зарегистрированным пользователям бесплатные услуги по размещению и поддержанию блогов, а также оплачиваемые обновления и дополнительные оплачиваемые услуги. Регистрация не требуется для пользователей, только читающих и комментирующих сообщения в блоге, если иное не указано владельцем блога (владелец может пометить свой блог как частный).

WordPress.com и WordPress.org. Новых пользователей WordPress часто смущает эта разница. Открытое программное обеспечение WordPress разрабатывается и распространяется через WordPress.org; вы можете загрузить копию программного обеспечения и установить его на собственном сервере. WordPress.com, напротив, – это поставщик услуг хостинга, который предлагает стандартное решение для пользователей, не желающих устанавливать программное обеспечение и выполнять настройку сервера самостоятельно. Домен для блога в WordPress.com обычно предоставляется в форме `имя-блога.wordpress.com`, если, конечно, владелец блога не оплатит приобретение другого доменного имени.

Обсуждаемый в данном разделе API предоставляется WordPress.com, поэтому данные, к которым мы можем получить доступ через этот API, предоставляются этим поставщиком.

Поддержка клиента на Python для WordPress.com API еще не особенно устоялась. Это вынуждает нас взаимодействовать с API напрямую (https://developer.wordpress.com/docs/api), или, другими словами, писать свой клиентский модуль.

К счастью, это не требует значительных усилий, к тому же в нашем распоряжении имеется библиотека **requests**, реализующая простой и удобный интерфейс для работы с HTTP. Мы воспользуемся этой библиотекой и реализуем несколько запросов к WordPress.com API с целью загрузки публикаций из конкретного блога.

Сначала установим библиотеку в нашей виртуальной среде:

```
$ pip install requests
```

Затем заглянем в документацию, обратив особое внимание на конечную точку `/sites/$site/posts` (https://developer.wordpress.eom/docs/api/1.1/get/sites/%24site/posts/).

Следующий сценарий определяет функцию `get_posts()`, посылающую запрос WordPress.com API и загружающую данные постранично:

```
# Chap07/blogs_wp_get_posts.py
import json
from argparse import ArgumentParser
import requests

API_BASE_URL = 'https://public-api.wordpress.com/rest/v1.1'

def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--domain')
    parser.add_argument('--posts', type=int, default=20)
    parser.add_argument('--output')
    return parser

def get_posts(domain, n_posts=20):
    url = "{}/sites/{}/posts".format(API_BASE_URL, domain)
    next_page = None
    posts = []
    while len(posts) <= n_posts:
        payload = {'page_handle': next_page}
        response = requests.get(url, params=payload)
        response_data = response.json()
        for post in response_data['posts']:
            posts.append(post)
        next_page = response_data['meta'].get('next_page', None)
        if not next_page:
            break
    return posts[:n_posts]

if __name__ == '__main__':
    parser = get_parser()
    args = parser.parse_args()

    posts = get_posts(args.domain, args.posts)

    with open(args.output, 'w') as f:
        for i, post in enumerate(posts):
            f.write(json.dumps(post)+"\n")
```

Для чтения аргументов командной строки данный сценарий использует экземпляр класса `ArgumentParser`. Он принимает два обязательных аргумента: `--domain`, определяющий доменное имя блога для загрузки данных, и `--output`, определяющий имя файла JSON Lines. Дополнительно можно передать аргумент `--posts`, определяющий количество публикаций для загрузки (по умолчанию равен 20).

Доменное имя блога – это полный URL-адрес блога, например `название-блога.wordpress.com`.

Этот сценарий можно запустить, например, следующей командой:

```
$ python blogs_wp_get_posts.py \
  --domain marcobonzanini.com \
  --output posts.marcobonzanini.com.jsonl \
  --posts 100
```

По прошествии нескольких секунд он создаст файл JSON Lines по одной публикации в строке, в формате JSON.

Перед описанием формата результата давайте подробно проанализируем функцию `get_posts()`, которая является ядром нашего сценария. Данная функция принимает два аргумента: доменное имя блога (`domain`) и число публикаций (`n_posts`) для загрузки, по умолчанию равное `20`.

Сначала функция определяет URL-адрес соответствующей конечной точки API и инициализирует переменную `next_page`, которая используется для обхода многочисленных страниц результатов. По умолчанию API возвращает страницы по 20 результатов (20 публикаций в блоге), поэтому, если блог содержит больше публикаций, нам придется выполнить обход нескольких страниц.

Основой функции является цикл `while`, который поочередно получает результаты по одной странице. Запрос к конечной точке API посылается методом GET и принимает несколько аргументов, как определено в документации. Для получения страницы результатов мы используем аргумент `page_handle`. В первой итерации эта переменная имеет значение `None`, чтобы извлечение результатов начиналось с самого начала. Ответ на запрос, возвращаемый методом `response.json()` в формате JSON, содержит список публикаций `response_data['posts']`, а также некоторые метаданные о запросе, хранящиеся в `response_data['meta']`. Если следующая страница существует, словарь с метаданными будет содержать ключ `next_page`. Если этот ключ отсутствует, мы присваиваем ему значение `None`.

Цикл останавливается, когда следующая страница `next_page` отсутствует, то есть после загрузки всех имеющихся публикаций или по достижении заданного количества публикаций, как объявлено в `n_posts`. В заключение возвращается срез списка публикаций. Это необходимо в случае, если требуемое число сообщений `n_posts` не кратно размеру страницы. Например, если мы затребуем 30 публикаций, сценарий загрузит две страницы, с 20 публикациями в каждой, и операция извлечения среза отсечет последние 10 элементов, которые не были затребованы.

Каждый объект-публикация определяется атрибутами, перечисленными в следующей таблице:

Атрибут	Описание
`ID`	Идентификатор публикации внутри блога
`URL`	Полный URL-адрес публикации
`attachment_count`	Число вложений (например, медиафайлов и т. д.)
`attachments`	Список объектов вложений
`author`	Объект профиля автора, в том числе полное имя, имя для входа в систему, аватар, URL-адрес профиля граватара (глобально узнаваемого аватара) и т. д.
`categories`	Список категорий, присвоенных данной публикации
`content`	Полное содержание публикации, включая разметку HTML
`date`	Дата публикации в формате ISO 8601
`discussion`	Информация о комментариях, пингах и т. д.
`excerpt`	Краткое изложение публикации, обычно первые несколько предложений
`feature_image`	URL-адрес иллюстрации к публикации, если есть
`global_ID`	Глобальный идентификатор данной публикации
`guid`	Домен блога и идентификатор публикации, объединенные в допустимый URL-адрес
`i_like`	Сообщает, понравилась ли данная публикация авторизованному пользователю

Атрибут	Описание
is_following	Сообщает, следит ли авторизованный пользователь за блогом
is_reblogged	Сообщает, цитировал ли авторизованный пользователь данную публикацию
like_count	Число пользователей, которым понравилась данная публикация
meta	Метаданные из API, например ссылки на гипермедиа для автоматического обнаружения API и т. д.
modified	Последняя дата изменения в формате ISO 8601
short_url	Короткий URL-адрес (http://wp.me) для цитирования в социальных и мобильных сетях
site_ID	Идентификатор блога
slug	Описательная часть URL-адреса данной публикации
status	Текущий статус данной публикации, например опубликована, черновик, в работе и т. д.
sticky	Сообщает, закреплена ли публикация (отображается вверху, независимо от даты публикации)
tags	Список тегов, связанных с публикацией
title	Заголовок публикации

Как видите, структура сообщения в блоге намного сложнее, чем его содержимое, но в целях настоящей главы мы главным образом сосредоточимся на текстовом содержимом, включая заголовки и выдержки.

Blogger API

Blogger – еще одна популярная веб-служба публикации в блогах, которая была приобретена компанией Google еще в 2003 г. Блоги, размещенные в Blogger, обычно доступны в субдомене blogspot.com, например `название-блога.blogspot.com`. Также для нескольких стран имеются особые домены (например, blogspot.co.uk для Соединенного Королевства), имея в виду, что пользователи из определенных стран автоматически перенаправляются к назначенному домену.

Будучи составной частью служб Google, ведение учетных записей централизованно, то есть после регистрации учетной записи в Google пользователи автоматически получают доступ к Blogger. Это касается и Blogger API; разработчики могут получить к нему доступ через Диспетчер API (https://console.developers.google.com/start), как это делалось для Google+ в главе 5 «*Тематический анализ в Google+*», где был описан процесс создания проекта, требующего доступа к (любому) Google API. Мы можем использовать тот же проект повторно и активировать Blogger API из Диспетчера API или создать новый проект с нуля. За подробным описанием этих шагов обращайтесь к главе 5 «*Тематический анализ в Google+*». В любом случае важно помнить, что нужно *включить Blogger API для нашего проекта*, иначе вы не получите к нему доступа.

Рассматривая Google+ API, мы также обсудили создание ключей доступа, и в частности серверного ключа, который также потребуется для доступа к Blogger API. Если вы следовали за примерами в главе 5 «*Тематический анализ в Google+*», у вас уже должен иметься серверный ключ, который можно сохранить в переменной окружения:

```
$ export GOOGLE_API_KEY="ваш-ключ-api"
```

В роли клиентской библиотеки мы используем ту же библиотеку, которой пользовались в главе 5 «*Тематический анализ в Google+*», потому что она реализует еди-

ный интерфейс для всех Google API. Если вы еще не установили ее, сделайте это прямо сейчас с помощью диспетчера пакетов `pip`:

```
$ pip install google-api-python-client
```

После настройки окружения можно приступить к работе над сценарием, взаимодействующим с Blogger API и загружающим публикации из заданного блога:

```
# Chap07/blogs_blogger_get_posts.py
import os
import json
from argparse import ArgumentParser
from apiclient.discovery import build

def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--url')
    parser.add_argument('--posts', type=int, default=20)
    parser.add_argument('--output')
    return parser


class BloggerClient(object):

    def __init__(self, api_key):
        self.service = build('blogger',
                             'v3',
                             developerKey=api_key)

    def get_posts(self, blog_url, n_posts):
        blog_service = self.service.blogs()
        blog = blog_service.getByUrl(url=blog_url).execute()
        posts = self.service.posts()
        request = posts.list(blogId=blog['id'])
        all_posts = []
        while request and len(all_posts) <= n_posts:
            posts_doc = request.execute()
            try:
                for post in posts_doc['items']:
                    all_posts.append(post)
            except KeyError:
                break
            request = posts.list_next(request, posts_doc)
        return all_posts[:n_posts]

if __name__ == '__main__':
    api_key = os.environ.get('GOOGLE_API_KEY')
    parser = get_parser()
    args = parser.parse_args()

    blogger = BloggerClient(api_key)

    posts = blogger.get_posts(args.url, args.posts)

    with open(args.output, 'w') as f:
        for post in posts:
            f.write(json.dumps(post)+"\n")
```

Клиент Google API, дирижирующий взаимодействиями со всеми Google API, реализует понятие компоновщика службы – своеобразный фабричный метод, используемый для создания объекта службы, который мы будем использовать для отправки запросов. Чтобы получить простой в использовании интерфейс, мы объявляем класс `BloggerClient`, хранящий ключ API, настраивающий объект службы и предоставляющий функцию `get_posts()`, очень похожую на ту, что мы определили для WordPress.com API.

Для чтения аргументов командной строки данный сценарий, как обычно, использует экземпляр класса `ArgumentParser`. Он принимает три аргумента:

```
$ python blogs_blogger_get_posts.py \
  --url http://googleresearch.blogspot.co.uk \
  --posts 50 \
  --output posts.googleresearch.jsonl
```

По аналогии с примером для WordPress.com мы используем аргументы `--posts` и `--output`, определяющие, соответственно, число публикаций и имя выходного файла. В отличие от предыдущего примера, аргумент `--url` требует, чтобы передаваемая строка была полным URL-адресом блога, а не только доменным именем (то есть она должна начинаться с имени протокола `http://`). Запустив приведенную выше команду, через несколько секунд мы получим последние 50 публикаций из блога Google Research.

Теперь внимательнее рассмотрим класс `BloggerClient`. Его конструктор принимает всего один аргумент – ключ API разработчика, полученный при регистрации приложения в Диспетчере API. Он используется для настройки объекта `service` фабричной функцией `build()`. Стоит отметить, что мы используем третью версию API, `v3`, поэтому, когда будете читать документацию, обращайте внимание на эту деталь.

Метод `get_posts()` выполняет основную работу. Он очень похож на функцию `get_posts()`, которую мы определили для WordPress.com API. Данный метод принимает два аргумента: URL-адрес блога и число публикаций.

В отличие от WordPress.com API, Blogger API требует передать объект блога, поэтому нам нужно преобразовать URL-адрес в числовой идентификатор. Это выполняется методом `getByUrl()` объекта, созданного вызовом `self.service.blogs()`.

Еще одно интересное отличие: вызовы большинства вспомогательных функций не инициируют немедленной отправки запросов к API, поэтому мы должны явно вызвать метод `execute()`, чтобы послать запрос и получить ответ.

Вызов `request.execute()` создает объект `posts_doc`, который может содержать или не содержать ключ `items` (список публикаций). Обертывание его в блок `try/except` гарантирует, что если в ответ на какой-то запрос не будет возвращено ни одной публикации, их обход прервется без возбуждения исключения.

По достижении заданного количества публикаций или их исчерпании возвращается срез списка публикаций.

Документы JSON, возвращаемые Blogger API, проще, чем в WordPress.com, но все ключевые атрибуты присутствуют. В следующей таблице перечислены основные атрибуты, которыми обладает каждая публикация:

Атрибут	Описание
author	Объект, представляющий автора публикации с отображаемым именем, идентификатором пользователя, аватаром и URL-адресом
blog	Объект, представляющий сам блог (например, при помощи идентификатора)
content	Полное содержимое публикации, включая форматирование HTML
id	Идентификатор публикации
labels	Список меток, связанных с публикацией
published	Дата публикации в формате ISO 8601
replies	Объект, представляющий комментарии/ответы к публикации
selfLink	API-ссылка на публикацию
title	Заголовок публикации
updated	Дата последнего обновления в формате ISO 8601
url	URL-адрес публикации

В данном случае тоже есть информация, более интересная, чем просто содержимое публикации, но основное внимание в настоящей главе уделяется только текстовым данным.

Каналы RSS и Atom

Многие блоги, и вообще многие веб-сайты, предлагают рассылку своей информации в стандартных форматах, предназначенных не для отображения на экране конечного пользователя, а для использования сторонними публикаторами. Это касается двух XML-форматов синдикации – **RSS** (Rich Site Summary – обогащенная сводка сайта) (https://ru.wikipedia.org/wiki/RSS) и **Atom** (https://ru.wikipedia.org/wiki/Atom), которые можно использовать для быстрого доступа к информации из веб-сайтов, в которых эта возможность реализована.

RSS и Atom принадлежат семейству форматов, которые обычно называют веб-каналами и используют для рассылки обновляемой информации. Поставщики каналов объединяют отдельные каналы в консорциум и дают пользователям возможность подписаться на конкретный канал при помощи приложения и получать свежую информацию по мере ее поступления. Некоторые приложения, в том числе и почтовые клиенты, реализуют возможность интеграции веб-каналов в свой технологический процесс.

В контексте анализа блогов и статей каналы интересны тем, что обеспечивают единую точку входа для того или иного веб-сайта, который предлагает свое информационное наполнение в машиночитаемом формате.

Например, **Би-би-си** (British Broadcasting Corporation, BBC) предлагает множество новостных каналов, сгруппированных под общими темами, такими как сообщения из-за рубежа, технологии, спорт и т. д. К примеру, канал для главных новостей доступен по адресу http://feeds.bbci.co.uk/news/rss.xml (его можно прочитать в браузере).

Для поддержки чтения веб-каналов в Python достаточно только установить библиотеку feedparser, которая реализует загрузку канала и преобразование необработанного документа XML в объект Python.

Прежде всего нужно установить данную библиотеку в виртуальную среду:

```
$ pip install feedparser
```

Следующий далее сценарий загружает канал и сохраняет записи с новостями в обычном формате JSON Lines:

```
# Chap07/blogs_rss_get_posts.py
import json
from argparse import ArgumentParser
import feedparser

def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--rss-url')
    parser.add_argument('--json')
    return parser

if __name__ == '__main__':
    parser = get_parser()
    args = parser.parse_args()

    feed = feedparser.parse(args.rss_url)
    if feed.entries:
        with open(args.json, 'w') as f:
            for item in feed.entries:
                f.write(json.dumps(item)+"\n")
```

Для чтения двух аргументов командной строки данный сценарий использует экземпляр класса `ArgumentParser`: `--rss-url` определяет URL-адрес канала, а `--json` – имя файла для сохранения канала в формате JSON Lines.

Например, чтобы загрузить RSS-канал с главными новостями BBC, можно выполнить следующую ниже команду:

```
$ python blogs_rss_get_posts.py \
  --rss-url http://feeds.bbci.co.uk/news/rss.xml \
  --json rss.bbc.jsonl
```

Сценарий не требует особых пояснений. Вся основная работа выполняется функцией `parse()` из библиотеки feedparser. Она распознает формат и преобразует документ XML в объект с атрибутом `entries` (списком элементов новостей). Написав всего несколько строк кода, можно выполнить обход списка и выгрузить отдельные элементы в наш формат JSON Lines.

Каждый элемент новостей содержит следующие атрибуты:

- `id`: URL-адрес новостного элемента;
- `published`: дата публикации;
- `title`: заголовок новости;
- `summary`: краткое изложение новости.

Получение данных из Википедии

Википедия, будучи одним из самых популярных веб-сайтов и всеобъемлющим справочником, вероятно, не нуждается в широком представлении. К концу 2015 г. в английском сегменте Википедии хранилось порядка 5 миллионов статей, а суммарное количество по всем языкам составляло более 38 миллионов.

Википедия предлагает свой API для доступа к материалам. Кроме того, регулярно формируются дампы данных.

Существует несколько проектов, предлагающих обертку вокруг Wikipedia API для Python. Особенно часто используется библиотека **wikipedia**. Ее можно установить как обычно:

```
$ pip install wikipedia
```

Библиотека имеет простой интерфейс, позволяющий, как говорится в ее документации, сосредоточиться не на получении данных из Википедии, а на их использовании.

Помимо доступа к полному содержимому страницы, Wikipedia API также предлагает такие возможности, как поиск и реферирование. Рассмотрим следующие примеры:

```
>>> import wikipedia
# Получить доступ к странице в Википедии
>>> packt = wikipedia.page('Packt')
>>> packt.title
'Packt'
>>> packt.url
'https://en.wikipedia.org/wiki/Packt'
# Получить краткое содержание страницы
>>> wikipedia.summary('Packt')
"Packt, pronounced Packed, is a print on demand publishing company based in Birmingham and
Mumbai." # на самом деле описание чуть длиннее
```

В примере выше мы получаем прямой доступ к странице об издательстве Packt Publishing, зная, что имя `Packt` – уникальное.

Если нет уверенности в точном названии страницы, можно воспользоваться функциями поиска и разрешения неоднозначностей.

Например, термин `London` может быть связан, по крайней мере, с двумя городами (один в Великобритании и другой в Канаде) и несколькими сущностями или событиями (например, Tower of London (лондонский Тауэр), Great Fire of London (Великий лондонский пожар) и т. д.):

```
>>> wikipedia.search('London')
['London', 'List of bus routes in London', 'Tower of London', 'London,
Ontario', 'SE postcode area', 'List of public art in London', 'Bateaux
London', 'London Assembly', 'Great Fire of London', 'N postcode area']
```

В случае орфографической ошибки можно получить неожиданные результаты:

```
>>> wikipedia.search('Londn')
['Ralph Venning', 'Gladys Reynell', 'GestiFute', 'Delphi in the Ottoman period']
```

Эту проблему можно обойти, запросив в Википедии варианты:

```
>>> wikipedia.suggest('Londn')
'london'
```

Представив пример о Лондоне, отметим также, что краткое изложение для этой сущности будет очень длинным. Если требуется изложение покороче, то можно указать число предложений, как показано на рис. 7.1.

Наконец, рассмотрим пример омонимии, которая инициирует задачу разрешения неоднозначности. В ответ на запрос страницы в Википедии с каким-то назва-

нием, которое относится к нескольким разным сущностям, когда явный кандидат среди них отсутствует, на экран выводится страница разрешения неоднозначности.

```
>>> wikipedia.summary('London', sentences=2)
'London /ˈlʌndən/ is the capital and most populous city of England
and the United Kingdom. Standing on the River Thames in the south
east of Great Britain, London has been a major settlement for two
millennia.'
```

Рис. 7.1 ❖ Краткое изложение

С точки зрения API такой запрос вернет исключение `DisambiguationError`, как показано ниже:

```
>>> wikipedia.summary('Mercury')
Traceback (most recent call last):
  # длинное описание ошибки
wikipedia.exceptions.DisambiguationError: "Mercury" may refer to:
Mercury (element)
Mercury (planet)
Mercury (mythology)
# ... длинный список вариантов для "Mercury"
```

Чтобы избежать проблем со снятием неоднозначности, можно заключить запрос в блок `try/except`:

```
>>> try:
...   wikipedia.summary('Mercury')
... except wikipedia.exceptions.DisambiguationError as e:
...   print("Среди имеющихся вариантов следующие: {}".format(e.options))
...
Среди имеющихся вариантов следующие: ['Mercury (element)', 'Mercury (planet)', 'Mercury
(mythology)', ...] # длинный список
```

Данная библиотека располагает удобным методом `set_lang()` для переключения на другие языки. Например, чтобы получить страницы на русском языке из русского сегмента Википедии, нужно сделать, как показано ниже:

```
>>> wikipedia.set_lang("ru")
>>> wikipedia.search('петербург')
['Санкт-Петербург', 'Московский вокзал (Санкт-Петербург)', 'Санкт-Петербург
(телерадиокомпания)', 'Санкт-Петербург-Сортировочный-Московский', 'Автомагистраль Москва
— Санкт-Петербург', 'СТС-Петербург', 'Крестовский остров (Санкт-Петербург)', 'Саперный
(Санкт-Петербург)', 'Радио Петербург', 'Санкт-Петербург — Витебский регион Октябрьской
железной дороги']
>>> wikipedia.summary('Санкт-Петербург', sentences=3)
'Санкт-Петербу́рг (с 18 [31] августа 1914 года до 26 января 1924 года — Петрогра́д,
с 26 января 1924 года до 6 сентября 1991 года — Ленингра́д) — второй по численности
населения город России. Город федерального значения. Административный центр Северо-
Западного федерального округа и Ленинградской области.'
```

Доступ к API Википедии можно использовать для загрузки полного текста статей, но это не единственное полезное применение. Мы также увидим, как извле-

кать существительные и именованные сущности из фрагмента текста. В данном контексте Википедией можно воспользоваться путем дополнения результатов, возвращаемых процедурой извлечения именованных сущностей, кратким описанием сущностей.

Несколько слов о выборке данных из веба

Выборка, или веб-скрапинг (web scraping), данных из веба – это процесс автоматической загрузки и извлечения информации с веб-сайтов. Программное обеспечение, реализующее выборку данных из веба, в сущности, имитирует действия человека, просматривающего (и сохраняющего) информацию о веб-сайте с целью получения большого объема данных или решения некой задачи, которая требует взаимодействия автоматическим способом.

Создание веб-скрапера (поискового робота, или просто бота) иногда является ответом на ситуации, когда веб-служба не поддерживает иного API или канала для загрузки данных.

Веб-скрапер является совершенно законным приложением, но важно иметь в виду, что автоматический сбор данных может противоречить условиям использования определенного веб-сайта, который вы намереваетесь «выскрести». Следует также учитывать этические и юридические нормы (которые могут разниться от страны к стране):

- имеем ли мы право получать доступ и загружать данные?
- не перегружаем ли мы серверов веб-сайта?

Ответ на первый вопрос обычно дается в условиях использования веб-сайта. На второй вопрос ответить сложнее. В принципе, посылая огромное число запросов веб-сайту (например, обращаясь ко множеству страниц в очень короткое время), мы определенно выходим за рамки имитации сеанса пользователя и можем вызвать ухудшения в работе веб-сайта. Мы можем даже вызвать избыточный трафик, ведущий к «отказу в обслуживании» (то есть подобная попытка будет рассматриваться как DoS-атака), если веб-сайт не в состоянии обработать такое количество запросов.

Учитывая изобилие данных и число служб, которые предоставляют такие данные, к созданию веб-скрапера следует прибегать только в особых случаях, когда отсутствуют другие механизмы извлечения данных.

Данная тема довольно сложная. Специализированные решения можно создавать при помощи таких библиотек, как **requests** и **BeautifulSoup**, хотя существует инструмент, который специально скроен для такого рода задач, – Scrapy (http://scrapy.org). Поскольку данная тема выходит за рамки настоящей книги, читатель, интересующийся этой тематикой, может найти множество публикаций, например *Web Scraping with Python*, Richard Lawson, 2-е издание, 2017, издательство Packt Publishing, или *Getting Started with Python Web Scraping*, Charles Clayton, 2017, Packt Publishing.

Основы обработки естественного языка

В данном разделе будут затронуты лишь внешние аспекты многосложной области обработки естественного языка. В предыдущих главах упоминались некоторые основы, необходимые для работы с текстовыми данными (например, лексемизация), но без особых подробностей. Здесь мы попытаемся продвинуться вперед

в понимании основ данной дисциплины. Из-за ее сложности и множества аспектов мы предпримем прагматический подход и лишь коснемся теоретических основ, сосредоточив основное внимание на практических примерах.

Предварительная обработка текста

Существенной частью любой системы обработки естественного языка является конвейер предварительной обработки. Прежде чем выполнить любую интересную задачу на фрагменте текста, его сначала нужно преобразовать в некоторое представление.

В предыдущих главах мы уже выполняли анализ текстовых данных, не вдаваясь в детали предварительной обработки и используя распространенные инструменты. В данном разделе мы выделим несколько широко используемых этапов предварительной обработки и обсудим их роль в общей картине построения системы обработки естественного языка.

Определение границ предложений

Задача разбиения фрагмента текста (то есть строки) на список предложений называется **определением границ предложений** (а также *обнаружением конца предложений*, или *лексемизацией предложений*).

Предложения – это лингвистические единицы, которые содержательно и грамматически объединяют слова с целью выражения законченной мысли. С точки зрения лингвистики (или языкознания, то есть науки, изучающей языки), всестороннее обсуждение всех возможных аспектов предложений заняло бы несколько страниц. Но с практической точки зрения, для создания системы обработки естественного языка такого обобщенного понятия лингвистической единицы более чем предостаточно.

Стоит отметить, что предложения могут быть осмысленными в отрыве друг от друга, но иногда их следует рассматривать в более широком контексте (например, абзаца или раздела). Это вызвано тем, что некоторые слова в предложениях могут ссылаться на понятия, выраженные в других предложениях. Рассмотрим пример: *Елизавета II – королева Соединенного Королевства. Она родилась в Лондоне в 1926 г., и ее правление началось в 1952 г.*

Этот простой фрагмент текста содержит два предложения. Во втором предложении используются местоимения *она* и *ее*. Если второе предложение рассматривать отдельно, эти два слова ничего не скажут об упоминаемом субъекте. Находясь в контексте, читатель легко свяжет эти два слова с подлежащим первого предложения – *Елизаветой II*. Задача такого рода называется *разрешением анафор* и признается очень сложной проблемой обработки естественного языка. В целом предложения со ссылками на другие предложения обычно используют локальный контекст, то есть ссылки в тексте указывают на предыдущее/следующее предложение, но не дальше. Распознавание границ более широкого контекста (например, абзаца и раздела текста) может пригодиться в некоторых приложениях, однако обычно отправной точкой являются именно предложения.

С практической стороны расщепление на предложения не представляет серьезной проблемы, поскольку для определения границ предложений можно использовать пунктуацию (например, точки, восклицательные знаки и т. д.). Такое упрощение приводит к некоторым ошибкам, как показано в следующем примере:

```
>>> text = "В 1989 г., работая в CERN над внутренней сетью организации, Тим Бернерс-Ли
предложил глобальный гипертекстовый проект, теперь известный как 'Всемирная паутина'"
>>> sentences = text.split('.')

>>> sentences
['В 1989 г', ", работая в CERN над внутренней сетью организации, Тим Бернерс-Ли предложил
глобальный гипертекстовый проект, теперь известный как 'Всемирная паутина'"]
```

Как видите, интерпретация каждой *точки* как признака конца предложения делит данное предложение на две части, оставляя нас с бессмысленным предложением *В 1989 г*. Можно привести много других подобных примеров, главным образом связанных с сокращениями (например, в английском это Dr., Ph.D, U.S.A. и т. д.).

К счастью, библиотека **NLTK**, реализующая набор инструментов для обработки естественного языка, предоставляет простой обходной путь:

```
>>> from nltk.tokenize import sent_tokenize
>>> sentences = sent_tokenize(text)
>>> sentences
["В 1989 г., работая в CERN над внутренней сетью организации, Тим Бернерс-Ли предложил
глобальный гипертекстовый проект, теперь известный как 'Всемирная паутина'"]
```

Теперь предложения определяются правильно. Приведенный пример чрезвычайно прост, и иногда библиотека NLTK тоже не способна выполнить правильную идентификацию предложений. В целом же данная библиотека прекрасно справляется со своей задачей, и вы можете смело пользоваться ею в большом числе приложений, не слишком беспокоясь.

Лексемизация слов

Задача разбиения предложения (то есть строки) на список слов называется **лексемизацией**, иногда эту процедуру также называют *токенизацией*. В данном контексте слова часто называются *лексемами*.

Вообще говоря, для большинства европейских языков, включая английский и русский, лексемизация представляется интуитивно понятной задачей, поскольку отдельные слова разделяются пробелами. Однако данное упрощение быстро показывает свои недостатки:

```
>>> s = "Данное предложение короткое, хорошее и по теме."
>>> wrong_tokens = s.split()
>>> wrong_tokens
['Данное', 'предложение', 'короткое,', 'хорошее', 'и', 'по', 'теме.']
```

Обратите внимание на лексемы «короткое,» и «теме.» (особенно на концевую пунктуацию): лексемизация потерпела неудачу, потому что разбиение по пробелам не учитывает пунктуацию. Создание лексемизатора, учитывающего все нюансы языка, может быть далеко непростым делом. Все становится гораздо сложнее для некоторых европейских языков, таких как немецкий или финский, в которых используется огромное разнообразие составных слов, когда, например, два или более слов объединяются для образования более длинного слова со значением, которое можетъ совпадать или отличаться от исходного значения составляющих его компонентов. Еще больше проблем возникает в связи с некоторыми азиатскими языками, такими как китайский, который в стандартном написании между словами не имеет пробелов вовсе. Для работы с конкретным языком могут

потребоваться специализированные библиотеки, например библиотека Python **jieba** (https://github.com/fxsjy/jieba), предназначенная для сегментации текста на слова в китайском языке.

Правда, читателю не стоит переживать по поводу подобных аспектов обработки языка, поскольку в центре нашего внимания находится английский и частично русский языки. Кроме того, библиотека NLTK уже достаточно хорошо оснащена для работы со стандартным английским языком:

```
>>> from nltk.tokenize import word_tokenize
>>> s = "Данное предложение короткое, хорошее и по теме."
>>> correct_tokens = word_tokenize(s)
>>> correct_tokens
['Данное', 'предложение', 'короткое', ',', 'хорошее', 'и', 'по', 'теме', '.']
```

Как видите, в данном примере отдельные слова были определены правильно, и знаки препинания тоже включены в результат как отдельные лексемы, потому что они могут пригодиться позже для обработки. Функция `word_tokenize()` из библиотеки NLTK требует установки пакета punkt. Если у вас эта функция возбудила исключение, обратитесь к главе 1 «*Социальные медиа, социальные данные и Python*», где описывается, как решить эту проблему.

Частеречная разметка

Получив последовательность слов, для каждого из них можно определить грамматическую категорию. Эта процедура называется **частеречной разметкой** (part-of-speech tagging, POS-tagging) и имеет широкое применение в различных задачах анализа текстов.

Широко используются такие грамматические категории, как существительные, глаголы или прилагательные, каждая из которых определяется разным символом тега. Разные инструменты могут опираться на разные наборы тегов, то есть использовать разные теги. Кроме того, результаты работы разных инструментов тоже могут сильно отличаться. По умолчанию библиотека NLTK использует набор тегов из банка синтаксических деревьев Penn Treebank Project (https://www.cis.upenn.edu/~treebank).

Полный список имеющихся тегов и их значений можно получить в справке:

```
>>> nltk.help.upenn_tagset()
```

Данная команда выведет длинный список, включающий все теги, их описания и некоторые примеры. Вот некоторые наиболее распространенные теги: `nn` (имена нарицательные), `nnp` (имена собственные), `JJ` (прилагательные) и `V*` (несколько тегов, начинающихся с `V`, описывают глаголы в разных временах).

Чтобы связать список лексем с соответствующими тегами, можно напрямую использовать функцию `nltk.pos_tag()`. NLTK предоставляет несколько реализаций частеречной разметки. В последних версиях по умолчанию используется усредненный персептрон. Как уже обсуждалось в главе 1 «*Социальные медиа, социальные данные и Python*», некоторые модели в библиотеке NLTK требуют загрузки дополнительных данных через интерфейс `nltk.download()`; это же относится к модели усредненного персептрона (см. рис. 7.2)[1].

---

[1] Имеется также версия для русского языка. – *Прим. перев.*

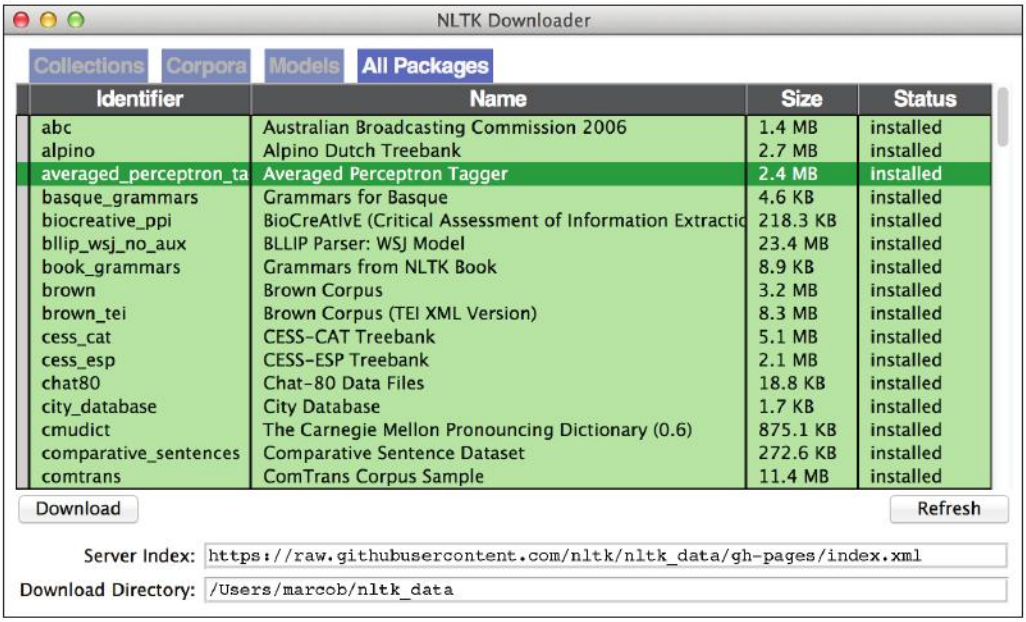

Рис. 7.2 ❖ Интерфейс NLTK для загрузки пакетов с выделенной моделью усредненного персептрона для частеречной разметки

После установки соответствующей модели можно протестировать функцию `pos_tag()` на примере предложения:

```
>>> from nltk import pos_tag
>>> tokens = word_tokenize("Данное предложение короткое, хорошее и по теме.")
>>> pos_tag(tokens, lang='rus')
[('Данное', 'A-PRO=n'), ('предложение', 'S'), ('короткое', 'A=n'), (',', 'NONLEX'),
('хорошее', 'A=n'), ('и', 'CONJ'), ('по', 'PR'), ('теме', 'S'), ('.', 'NONLEX')]
```

Функция `pos_tag()` возвращает список кортежей, в котором каждый кортеж имеет формат (лексема, тег). Например, первая лексема `Данное` является местоименным прилагательным (`A-PRO=n`), вторая лексема `предложение` – существительным (`S`) и т. д.

Одна из типичных проблем частеречной разметки заключается в отсутствии единого набора тегов, даже притом, что многие исследователи пришли к определенному соглашению по поводу именования основных категорий. Поэтому использование различных инструментов частеречной разметки нередко приводит к разным результатам. С учетом неоднозначности и других нюансов естественного языка то же самое слово может быть отнесено к разным категориям, в зависимости от контекста. Например, такие слова, как fish, walk и view, могут быть глаголами или существительными[1].

Частеречная разметка обычно применяется для определенной цели. Например, информация о частях речи может пригодиться, по крайней мере, в двух случаях:

[1] Для русского языка можно привести примеры таких слов, как «пила», «жила», «пристань», «село». – *Прим. ред.*

при поиске исходной леммы слова (см. раздел «*Нормализация слов*» далее в этой главе) и при попытке извлечь структурированную информацию из неструктурированного текста (см. раздел «*Извлечение информации*»).

Нормализация слов

Лексемизация и частеречная разметка являются основными этапами предварительной обработки, характерными для многих приложений. В зависимости от целей может также потребоваться выполнить дополнительные шаги, чтобы повысить точность приложения в целом.

В качестве обобщающего понятия, охватывающего все дополнительные шаги, которые мы хотели бы выполнить на отдельных терминах, здесь будет использоваться словосочетание «нормализация слов».

Нормализация регистра букв Чтобы подчеркнуть необходимость нормализации текста, начнем со следующего простого примера:

```
>>> "президент" == "Президент"
False
```

Приведенный пример очевиден для большинства программистов, поскольку `президент` и `Президент` по существу являются двумя разными строками, но с лингвистической точки зрения эти два слова часто рассматриваются как одинаковые. Например, при подсчете частот было бы нежелательно разделять слова, которые являются одинаковыми.

Как один из способов исправить ситуацию можно преобразовать все буквы в тексте в нижний или в верхний регистр:

```
>>> "президент".lower() == "Президент".lower()
True
```

Слова в этом примере преобразуются в нижний регистр, благодаря чему исходные термины определяются как одинаковые.

Важно помнить, что такое преобразование необратимо. Если заменить текст его версией в нижнем регистре, мы не сможем воспроизвести оригинальный текст. Во многих приложениях требуется показывать пользователю оригинальный текст, а обрабатывать нормализованный. В этом случае важно сохранить копию оригинала и рассматривать ее как неизменяемые данные.

Также стоит отметить, что нормализация текста не всегда приводит к желаемому результату. Классическим примером является акроним для *United States, US*, который в результате нормализации превращается в личное местоимение в объектном падеже *us*. Чтобы отловить некоторые другие граничные случаи, может потребоваться провести ряд испытаний методом проб и ошибок.

Выделение основ слов В английском, а также в других языках имеются слова, слегка отличающиеся, но имеющие то же значение, например *fish*, *fishes* и *fishing* (рыба, рыбы, рыбалка). Определение этих слов как принадлежащих общему *концептуальному* классу может пригодиться в некоторых ситуациях, когда важна близость слов, а не их точное написание.

Данная процедура называется *выделением основы*, или *стеммингом*, а неизменяемая (базовая) форма слова называется *основой*. Обычно выделение основы

заключается в *отсечении суффиксов*, когда из слова удаляются конечные буквы, пока не будет достигнута его базовая форма. Классическим примером алгоритма отсечения суффиксов, по-прежнему широко используемым, является стеммер Портера (алгоритм отсечения суффиксов, М. Ф. Портера, 1980).

Библиотека NLTK реализует стеммер Портера и множество других стеммеров:

```
>>> from nltk.stem import PorterStemmer
>>> stemmer = PorterStemmer()
>>> stemmer.stem('fish')
'fish'
>>> stemmer.stem('fishes')
'fish'
>>> stemmer.stem('fishing')
'fish'
```

Для русских слов можно использовать стеммер Snowball (http://snowball.tartarus.org/algorithms/russian/stemmer.html), как показано ниже:

```
>>> from nltk import SnowballStemmer
>>> stemmer = SnowballStemmer("russian")
>>> stemmer.stem("рыбаки")
'рыбак'
>>> stemmer.stem("рыбаков")
'рыбак'
>>> stemmer.stem("рыбаками")
'рыбак'
```

Выделение основы слова, как и нормализация регистра, необратимо. Применяя стемминг, рекомендуется сохранять копию оригинала на случай, если понадобится его воспроизвести.

В отличие от нормализации регистра, выделение основы слова является также языковозависимым. Альтернативой стеммеру Портера может служить стеммер Snowball, поддерживающий несколько языков, благодаря чему его можно использовать для обработки многоязычных данных:

```
>>> from nltk.stem import SnowballStemmer
>>> SnowballStemmer.languages
('danish', 'dutch', 'english', 'finnish', 'french', 'german', 'hungarian',
'italian', 'norwegian', 'porter', 'portuguese', 'romanian', 'russian',
'spanish', 'swedish')
>>> stemmer = SnowballStemmer('italian')
>>> stemmer.stem('pesce') # рыба
'pesc
>>> stemmer.stem('pesci') # рыбы
'pesc'
>>> stemmer.stem('pescare') # ловить рыбу
'pesc'
```

И последнее, о чем следует помнить: основа слова не всегда является словом в буквальном смысле; это лишь базовая его часть, иногда совпадающая с корнем слова.

Лемматизация Аналогично выделению основы слова, лемматизация объединяет разные флективные формы слова в группу, чтобы их можно было анализировать как одинаковые.

В отличие от выделения основы слова, данная процедура более запутанная и требует некоторых дополнительных сведений, как, например, наличия частеречных тегов для всех слов, подлежащих лемматизации. Результат лемматизации называется *леммой* и является словом в буквальном смысле. Простой подход с отсечением суффиксов непригоден для лемматизации, потому что, например, некоторые формы неправильного глагола имеют совершенно другую морфологию, чем их лемма. Например, go, goes, going и went все должны быть поставлены в соответствие глаголу go, однако лемму для слова went стеммеру распознать не удастся:

```
>>> from nltk.stem import PorterStemmer
>>> stemmer = PorterStemmer()
>>> stemmer.stem('go')
'go'
>>> stemmer.stem('went')
'went'
```

Лемматизация в NLTK реализована с использованием лексического ресурса WordNet (https://wordnet.princeton.edu), который группирует английские слова в наборы синонимов (так называемые *synsets*) и предоставляет определения и другую информацию, связанную со словами:

```
>>> from nltk.stem import WordNetLemmatizer
>>> lemmatizer = WordNetLemmatizer()
>>> lemmatizer.lemmatize('go', pos='v')
'go'
>>> lemmatizer.lemmatize('went')
'went'
>>> lemmatizer.lemmatize('went', pos='v')
'go'
```

Функция `lemmatize()` принимает второй необязательный аргумент с тегом части речи. Без информации о частях речи лемматизатор, скорее всего, потерпит неудачу, как в следующем ниже примере:

```
>>> lemmatizer.lemmatize('am')
'am'
>>> lemmatizer.lemmatize('am', pos='v')
'be'
>>> lemmatizer.lemmatize('is')
'is'
>>> lemmatizer.lemmatize('is', pos='v')
'be'
>>> lemmatizer.lemmatize('are')
'are'
>>> lemmatizer.lemmatize('are', pos='v')
'be'
```

Теги частей речи для лемматизатора на основе WordNet сгруппированы в макрокатегории: прилагательные (`a`), существительные (`n`), глаголы (`v`) и наречия (`r`).

Удаление стоп-слов Стоп-слова – это термины, которые не несут особого содержания, по крайней мере, когда берутся по отдельности, такие как артикли и союзы, но часто употребляются в большинстве естественных языков. Ранние исследования в области информационного поиска (например, Лун (Luhn), 1958) показали, что наиболее значимые слова в наборе не являются ни самыми частыми, ни самыми редкими). В ходе дальнейших исследований выяснилось, что эти слова можно удалить без потери значимого содержания. В те далекие годы, когда дисковое пространство и оперативная память были намного дороже, удаление стоп-слов позволяло сократить объем данных и, следовательно, сэкономить некоторое пространство при построении индекса для набора документов.

В наше время терабайты стали намного дешевле, а мотивация экономии дискового пространства – намного слабее, чем прежде, но вопрос о выгоде удаления *неинтересных* слов до сих пор остается актуальным для многих приложений.

Однако легко можно найти контрпримеры, когда удаление стоп-слов вредно. Представьте, что поисковая система не индексирует стоп-слова. Как тогда мы смогли бы отыскать веб-страницы об английской рок-группе «*The Who*» или известном стихотворении Шекспира «*To be, or not to be: that is the question*» (Быть или не быть: вот в чем вопрос?). Все термины в названии группы и в строке из стихотворения (кроме «question») являются общеупотребительными английскими стоп-словами. Поскольку примеров, когда удаление стоп-слов нежелательно, бесчисленное множество, современные поисковые системы, такие как Google или Bing, на самом деле хранят стоп-слова в своих индексах.

Другой подход к удалению неинтересных терминов основан на частотном анализе. Наблюдая за глобальной частотой слова в документах (доля документов, в которых оно появляется), можно определить произвольный порог с целью отсечения терминов, которые появляются слишком часто или слишком редко. Это решение реализовано в таких библиотеках, как scikit-learn, где конструктор другого векторизатора (например, класса `TfidfVectorizer`) позволяет передавать аргументы `max_df` и `min_df`.

В предметно-ориентированных наборах некоторые термины встречаются чаще, чем в обычном английском языке. Например, в наборе с обзорами кинофильмов термины «`movie`» (фильм) или «`actor`» (актер) присутствуют почти в каждом документе. Несмотря на то что эти термины сами по себе стоп-словами не являются (они действительно имеют некоторое значение, даже поодиночке), можно поддаться соблазну считать их таковыми. Автоматическое удаление терминов в силу их частот может породить дополнительные проблемы: как охватить такие аспекты языка, как «`bad movie`» (плохой фильм) или «`terrific actor`» (потрясающий актер)?

После обсуждения примеров подведем итог этому длинному обсуждению такого простого вопроса, как удаление стоп-слов: ответ на вопрос, выгодно удалять стоп-слова или нет, в значительной степени зависит от приложения. Важно протестировать разные варианты и оценить результативность наших алгоритмов с этим этапом предварительной обработки и без него.

В Интернете можно найти несколько списков общеупотребительных английских слов, поэтому мы легко можем использовать любой из них. Кроме того, библиотека NLTK включает свой список стоп-слов, как показано в примере ниже:

```
>>> from nltk.corpus import stopwords
>>> stop_list = stopwords.words('english')
>>> len(stop_list)
127
>>> stop_list[:10] # first 10 words
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you',
'your']
```

Сопоставление синонимов Последний шаг нормализации, который мы обсудим в данном разделе, – сопоставление единственного элемента группе синонимов. Цели, преследуемые этим шагом, схожи с целями нормализации регистра. Единственное отличие: синонимы не являются полностью идентичными словами, поэтому сопоставление терминов, несущих отличающиеся значения в зависимости от контекста, может привести к неоднозначным результатам.

Для этого проще всего использовать контролируемый список слов, то есть лексический ресурс, обеспечивающий такое сопоставление. В Python он имеет форму словаря:

```
synonyms = {'big': 'large', 'приобрести': 'купить'}
```

В приведенном примере словарь `synonyms` можно использовать для приведения слова `big` к слову `large`, а слово `приобрести` – к слову `купить`. Такая замена выполняется очень легко, простым обращением к нужному ключу в словаре, например `synonyms['big']` вернет `large`. Словарь также имеет метод `get(ключ, default=None)`, который пытается получить доступ к конкретному ключу. Данный метод принимает второй аргумент, который возвращается, если ключ не найден:

```
>>> text = "Я хотел бы приобрести книгу по теме Big Data".lower().split()
>>> text
['я', 'хотел', 'бы', 'приобрести', 'книгу', 'по', 'теме', 'big', 'data']
>>> new_text = [synonyms.get(word, word) for word in text]
>>> new_text
['я', 'хотел', 'бы', 'купить', 'книгу', 'по', 'теме', 'large', 'data']
```

Для обхода элементов списка `text` в этом примере используется генератор списков. Для каждого слова в списке проверяется наличие синонима в словаре `synonyms`. Если синоним отсутствует, использование метода `get()` со вторым аргументом предотвратит появление исключения `KeyError` и на выходе оставит первоначальное слово, как есть, без изменений.

Как обычно, во время работы с естественным языком случаи неоднозначности могут возникать в самых простых ситуациях. С одной стороны, приведение слова «*приобрести*» к «*купить*» выглядит разумным, но при наличии предыдущего примера возникает вопрос, а что именно может означать *large data*. Проблема в том, что мы не рассматриваем контекст, в котором используется слово *big*, тогда как в действительности его следует рассматривать не в изоляции, а как составную часть словосочетания *big data*. Помимо этого особого случая, в естественном языке можно найти бесчисленные примеры слов с многочисленными значениями (https://ru.wikipedia.org/wiki/Полисемия).

Такие ресурсы, как ранее упомянутый проект WordNet, предоставляют обширную информацию обо всех потенциальных семантических группах слова. Только

чтобы подчеркнуть, насколько сложно данную задачу решить с помощью простого словаря, рассмотрим следующий пример:

```
>>> from nltk.corpus import wordnet
>>> syns = wordnet.synsets('big', pos='a')
>>> for syn in syns:
...   print(syn.lemma_names())
...
['large', 'big']
['big']
['bad', 'big']
['big']
['big', 'large', 'prominent']
['big', 'heavy']
['boastful', 'braggart', 'bragging', 'braggy', 'big', 'cock-a-hoop',
'crowing', 'self-aggrandizing', 'self-aggrandising']
['big', 'swelled', 'vainglorious']
['adult', 'big', 'full-grown', 'fully_grown', 'grown', 'grownup']
['big']
['big', 'large', 'magnanimous']
['big', 'bighearted', 'bounteous', 'bountiful', 'freehanded', 'handsome',
'giving', 'liberal', 'openhanded']
['big', 'enceinte', 'expectant', 'gravid', 'great', 'large', 'heavy', 'with_child']
```

Как видите, слово big используется приблизительно в дюжине разных наборов синонимов. Несмотря на то что все они так или иначе вращаются вокруг понятия *big*, нюансы языка требуют, чтобы это разнообразие было должным образом зафиксировано.

Другими словами, это нелегкая задача. Разрешение лексической многозначности является областью активного исследования, которая оказывает влияние на приложения анализа естественного языка. Некоторые приложения на четко очерченной области данных выиграют от использования небольших контролируемых словарей, но в общем случае с синонимами следует обращаться с осторожностью.

Извлечение информации

Одной из самых сложных и интересных в обработке естественного языка является задача извлечения структурированной информации из неструктурированного текста. Эта процедура обычно так и называется – **извлечением информации**, и решает целый ряд подзадач, среди которых самой популярной, пожалуй, является **распознавание именованных сущностей** (Named Entity Recognition, NER).

Цель выделения именованных сущностей – выявить во фрагменте текста упоминания сущностей, таких как имена людей или названия компаний, и назначить им правильные метки. К наиболее распространенным видам сущностей относятся: имена людей, названия организаций, географические названия, числительные, время и дата, названия валют. Например, рассмотрим следующий текст:

> «*Ford Motor Company (или просто “Форд”) – это американская многонациональная автомобилестроительная компания, расположенная в городе Дирборн в пригороде Детройта, штат Мичиган. Была основана Генри Фордом 16 июня 1903 г.*».

Предыдущий фрагмент (взятый из Википедии) содержит несколько именованных сущностей, перечисленных в следующей таблице:

Ссылка на сущность	Тип сущности
Ford Motor Company	Организация
Дирборн, Мичиган	Местоположение
Детройт	Местоположение
Генри Форд	Человек
16 июня 1903 г.	Время

С точки зрения извлечения структурированной информации тот же фрагмент текста описывает связи между объектами, как показано ниже:

Субъект	Связь	Объект
Ford Motor Company	Расположенный в	Детройт
Ford Motor Company	Основанный	Генри Форд

Часто в тексте можно выявить дополнительные связи. Например, если Форд – это американская компания, расположенная в Детройте, можно заключить, что Детройт расположен в США.

Представление знаний и логические выводы. Пример с компанией Форд намеренно упрощен для большей ясности. Важно помнить, что представление знаний является многосложной темой и самостоятельной областью исследования, которая затрагивает несколько дисциплин, включая искусственный интеллект, информационный поиск, обработку естественного языка, базы данных и Семантическую паутину.
Конечно, существуют определенные стандарты представления знаний (например, формат RDF), однако, разрабатывая структуру хранилища знаний, важно разбираться в предметной области приложения и понимать, как будут использоваться эти знания.

В целом архитектура системы извлечения информации подчинена структуре на рис. 7.3.

Первый структурный элемент подписан как **предварительная обработка**, но он может включать несколько этапов, как описано в разделе «*Предварительная обработка текста*». Предполагается, что на вход системе подается исходный текст, то есть строка. Это предположение означает, что мы уже извлекли текст из некоторого формата данных (например, из файла JSON, из записи в базе данных и т. д.). Конечный результат этапа предварительной обработки зависит от выбранных шагов. Для извлечения сущностей не требуется удалять стоп-слова, извлекать основу слов и производить нормализацию, поскольку изменение исходной последовательности слов помешало бы правильно идентифицировать составные сущности. Например, если словосочетание «*Bank of*

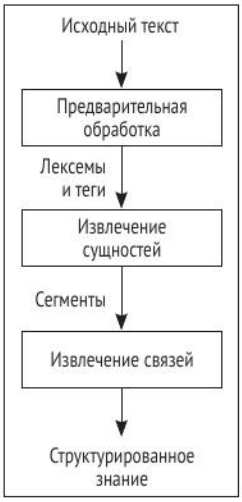

Рис. 7.3 ❖ Общая схема системы извлечения информации

America» преобразовать в *bank america*, определить правильную сущность будет невозможно. По этой причине на данном этапе часто производятся только лексемизация и частеречная разметка, поэтому в результате получается список кортежей `(термин, тег_части_речи)`.

Второй этап в данном конвейере – **извлечение сущностей**. Из списка лексем выделяются сегменты с сущностями и при необходимости отмечаются согласно их типам. В зависимости от приложения мы можем интересоваться только именами собственными или дополнительно выявить неопределенные существительные и именные группы. Результатом этого этапа является список сегментов с короткими и непересекающимися словосочетаниями, которые можно организовать в деревья. Сегментирование также называется поверхностным синтаксическим анализом, или частичным синтаксическим разбором, в отличие от полного синтаксического разбора, цель которого состоит в том, чтобы произвести полное дерево синтаксического анализа.

Заключительный этап – **извлечение связей** между сущностями. Связи интересны тем, что позволяют логически выводить сущности, упомянутые в тексте. Этот этап часто самый трудный из-за сложного характера работы с языком. Он также зависит от качества результатов предыдущих этапов, например если модуль извлечения сущностей пропустит сущность, все соответствующие связи тоже будут пропущены. Окончательный результат можно представить как список кортежей `(субъект, связь, объект)`, как описано в примере с компанией *Ford Motor*, или в ином виде, лучше соответствующем потребностям приложения.

Библиотека NLTK включает готовую к применению реализацию, позволяющую легко выполнить процедуру выделения именованных сущностей. Функция `nltk.chunk.ne_chunk()` принимает список кортежей в формате `(термин, тег_части_речи)` и возвращает дерево сегментов. Каждой именованной сущности, если она распознана правильно, присваивается соответствующий тип (человек, географическое название и т. д.). Данная функция также принимает второй необязательный аргумент `binary` со значением по умолчанию `False`. Если передать в нем `True`, распознавание типа сущности выполняться не будет и все они получат метку `NE` (Named Entity – именованная сущность).

Следующий пример объединяет все вместе и использует Wikipedia API, чтобы получить краткие описания идентифицированных сущностей.

Как обычно, сначала следует блок инструкций `import` и затем определяется парсер аргументов командной строки `ArgumentParser`:

```
# Chap07/blogs_entities.py
from argparse import ArgumentParser
from nltk.tokenize import word_tokenize
from nltk import pos_tag
from nltk.chunk import ne_chunk
import wikipedia

def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--entity')
    return parser
```

Далее следует функция, которая выполняет обход списка лексем, обработанных процедурой частеречной разметки, и возвращает список именных групп. Имен-

ные группы – это последовательности лексем, отмеченных одинаковым тегом и связанных с существительным (например, NN или NNP). Список существительных может сравниваться с фактическими именованными сущностями в том виде, как они были определены библиотекой NLTK:

```
def get_noun_phrases(pos_tagged_tokens):
    all_nouns = []
    previous_pos = None
    current_chunk = []
    for (token, pos) in pos_tagged_tokens:
        if pos.startswith('NN'):
            if pos == previous_pos:
                current_chunk.append(token)
            else:
                if current_chunk:
                    all_nouns.append((' '.join(current_chunk),
                                      previous_pos))
                current_chunk = [token]
        else:
            if current_chunk:
                all_nouns.append((' '.join(current_chunk),
                                  previous_pos))
            current_chunk = []
        previous_pos = pos
    if current_chunk:
        all_nouns.append((' '.join(current_chunk), pos))
    return all_nouns
```

Затем определяется функция, которая принимает дерево сегментов и тип сущности и возвращает список сущностей из дерева, соответствующих указанному типу:

```
def get_entities(tree, entity_type):
    for ne in tree.subtrees():
        if ne.label() == entity_type:
            tokens = [t[0] for t in ne.leaves()]
            yield ' '.join(tokens)
```

И наконец, основная логика сценария: мы посылаем Википедии запрос для конкретной сущности и получаем ее краткое описание. Затем это описание лексемизируется и обрабатывается процедурой частеречной разметки. В кратком описании выявляются именованные группы и именованные сущности. Затем мы выводим каждую именованную сущность с ее кратким описанием (снова из Википедии):

```
if __name__ == '__main__':
    parser = get_parser()
    args = parser.parse_args()

    entity = wikipedia.summary(args.entity, sentences=2)

    tokens = word_tokenize(entity)
    tagged_tokens = pos_tag(tokens)
    chunks = ne_chunk(tagged_tokens, binary=True)

    print("-----")
    print("Описание понятия {}".format(args.entity))
    print(entity)
```

```
print("-----")
print("Группы существительных в описании:")
for noun in get_noun_phrases(tagged_tokens):
    print(noun[0])  # кортеж (noun, pos_tag)
print("-----")
print("Именованные сущности в описании:")
for ne in get_entities(chunks, entity_type='NE'):
    summary = wikipedia.summary(ne, sentences=1)
    print("{}: {}".format(ne, summary))
```

Этот сценарий можно запустить, например, следующей командой:

```
$ python blogs_entities.py --entity London
```

Результат работы сценария частично показан на рис. 7.4.

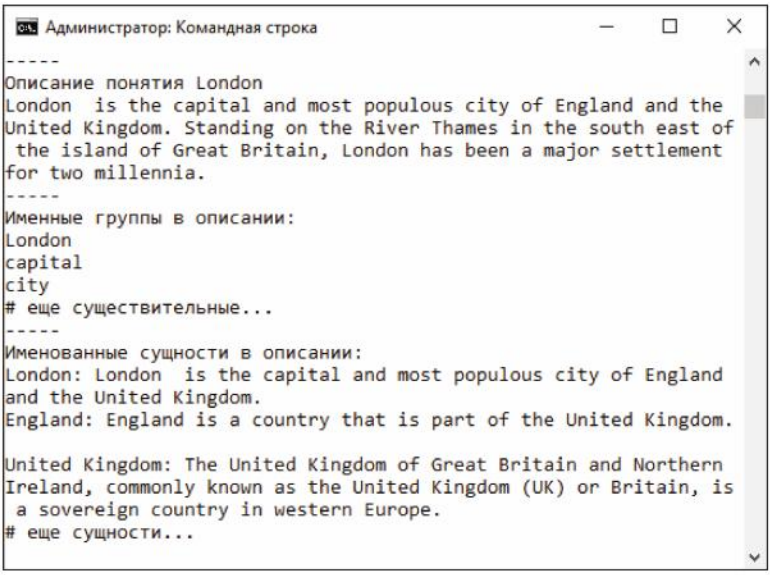

Рис. 7.4 ❖ Результат работы сценария

Полный список идентифицированных именных групп показан на рис. 7.5:

Рис. 7.5 ❖ Выявленные именные группы

Конечно, не все термины представляют интерес (например, `millennia`), но все существительные на месте. С другой стороны, список выявленных именованных сущностей выглядит довольно полным: `London`, `England`, `United Kingdom`, `River Thames`, `Great Britain` и `London`.

Резюме

В этой главе была представлена сложная область исследований со многими вызовами и возможностями – область обработки естественного языка.

В первой части главы были перечислены способы получения текстовых данных из веба. Блоги являются наиболее подходящим кандидатом для анализа текстов, учитывая изобилие текстовых данных в них. После знакомства с двумя самыми популярными платформами блогинга – WordPress.com и Blogger – мы обобщили проблему, представив стандарт XML для веб-каналов, таких как RSS и Atom. Учитывая повсеместное присутствие Википедии в вебе и, вероятно, в повседневной жизни многих веб-пользователей, она тоже заслужила право быть упомянутой в обсуждаемом контексте. Мы увидели, насколько просто взаимодействовать со всеми этими службами из Python, используя имеющиеся библиотеки или путем быстрой реализации собственных функций.

Вторая часть главы была посвящена обработке естественного языка. Ранее в данной книге мы уже познакомились с некоторыми понятиями из этой области, но здесь впервые было дано более формальное введение. Мы описали конвейер обработки естественного языка, от исходного текста до идентификации именованных сущностей, и прошли через все этапы предварительной обработки, необходимые для реализации продвинутых приложений.

В следующей главе основное внимание будет уделено другим социально-медийным API, где мы раздвинем наши горизонты с точки зрения возможностей анализа данных.

Глава 8

Анализ других данных

Данная глава расскажет о некоторых других социально-медийных API. В частности, мы обсудим следующие темы:

- анализ видео на YouTube;
- анализ проектов с открытым исходным кодом на GitHub;
- анализ местных предприятий в Yelp;
- использование библиотеки requests для работы с любыми веб-API;
- создание собственного клиента на основе библиотеки requests.

Большое количество социальных API

Все предыдущие главы были посвящены отдельным социально-медийным платформам, завоевавшим популярность в последние годы. К счастью, это еще не конец рассказа. Существует большое число других платформ, поддерживающих аналогичные возможности и неплохие API для анализа данных. С другой стороны, всестороннее описание всех возможных API с исчерпывающими примерами и интересными вариантами использования вышло бы далеко за рамки настоящей книги.

Чтобы дать пищу для размышлений, данная глава будет посвящена двум аспектам анализа социальных медиа. Во-первых, мы по шагам рассмотрим поиск и анализ сложных сущностей с использованием нескольких интересных API, таких как видео, проекты с открытым исходным кодом и местные предприятия. Во-вторых, обсудим, что делать в отсутствие хорошего клиентского модуля для Python, реализующего поддержку конкретного API.

Анализ видео на YouTube

В наши дни YouTube (http://youtube.com) не нуждается в особом представлении, будучи одним из наиболее посещаемых веб-сайтов в мире (признанный в марте 2016 вторым в рейтинге аналитической компании Alexa). Данная служба видеоцитирования (и распространения видеоматериалов) хранит широкий спектр информационных материалов, которые создаются и используются широким кругом разных авторов, от видеоблогеров-любителей до крупных корпораций. Зарегистрированные пользователи YouTube могут публиковать, оценивать и комментировать видеоролики. Просмотр и цитирование видеоматериалов не требуют от пользователей регистрации профиля.

Видеохостинг YouTube был приобретен компанией Google в 2006 и сегодня является составной частью более крупной платформы Google. В главе 5 «*Тематический*

анализ в Google+» и главе 7 «*Блоги, RSS, Википедия и обработка естественного языка*» мы уже представили другие службы Google: Google+ и Blogger. Веб-служба YouTube предоставляет три разных API для интеграции приложений с платформой YouTube: YouTube Data, YouTube Analytics и YouTube Reporting. Для извлечения и анализа данных мы используем первый из них, так как другие два скроены для авторов.

Первые шаги на пути к получению доступа к YouTube Data API очень похожи на виденные ранее, поэтому мы советуем читателю вернуться к главе 5 «*Тематический анализ в Google+*», где описана процедура доступа к Диспетчеру API (https://console.developers.google.com). Если вы уже зарегистрировались в Диспетчере API, просто включите YouTube Data API, как показано на рис. 8.1.

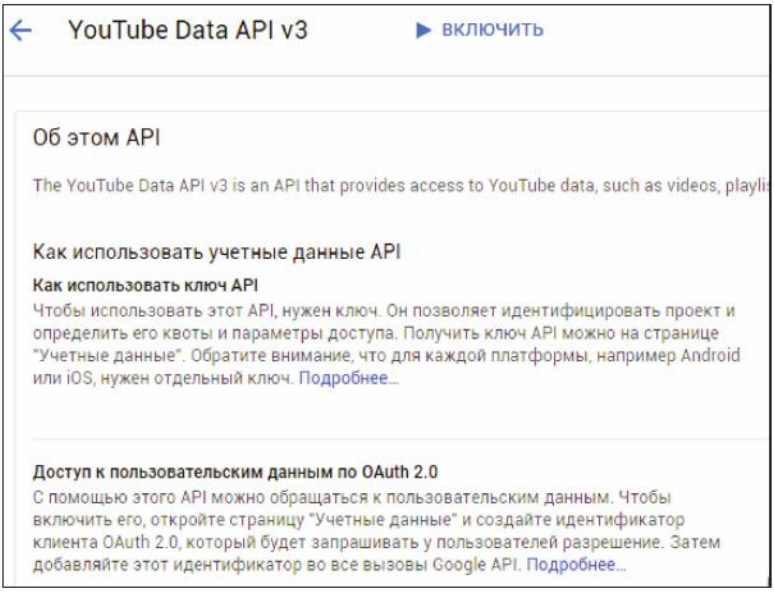

Рис. 8.1 ❖ Включение YouTube Data API в Диспетчере API

На вкладке Credentials (Учетные данные) в Диспетчере API можно использовать тот же ключ API, который мы получили для проекта Google+, обсуждавшегося в главе 5 «*Тематический анализ в Google+*». Разобравшись с учетными данными, можно сохранить ключ Google API в переменной окружения:

```
$ export GOOGLE_API_KEY="ваш-ключ-api"
```

Также снова используем ту же клиентскую библиотеку для Python, что и для других служб Google, установив ее из «сырной лавки» CheeseShop:

```
$ pip install google-api-python-client
```

Напомним, что только что установленная библиотека доступна в программном коде как `googleapiclient` и часто импортируется под псевдонимом `apiclient` (используемым в примерах программ).

Данный API ограничивает частоту запросов на основе системы квот. Каждое приложение получает ежедневную квоту в 1 000 000 единиц. Разные вызовы раз-

ных конечных точек имеют разную квотную стоимость (измеряемую в единицах запросов), однако ограничения в один миллион нам более чем достаточно, чтобы приступить к работе и начать экспериментировать с API. Число выполненных запросов и квоту для данного API можно посмотреть в Диспетчере API.

Первый пример взаимодействия с YouTube Data API – обобщенное поисковое приложение. Сценарий youtube_search_video.py реализует вызов конечной точки API search.list (https://developers.google.com/youtube/v3/docs/search/list). Один вызов данной конечной точки имеет квотную стоимость 100 единиц:

```
# Chap08/youtube_search_video.py
import os
import json
from datetime import datetime
from argparse import ArgumentParser
from apiclient.discovery import build

def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--query')
    parser.add_argument('--n')
    return parser

if __name__ == '__main__':
    parser = get_parser()
    args = parser.parse_args()

    api_key = os.environ.get('GOOGLE_API_KEY')
    service = build('youtube',
                    'v3',
                    developerKey=api_key)

    search_feed = service.search()
    search_query = search_feed.list(q=args.query,
                                    part="id,snippet",
                                    maxResults=args.n,
                                    type='channel')
    search_response = search_query.execute()

    print(json.dumps(search_response, indent=4))
```

Для чтения аргументов командной строки данный сценарий, как и прежде, использует экземпляр класса ArgumentParser. Аргумент --query служит для передачи строки запроса, а аргумент --n (необязательный, имеет значение по умолчанию 10) определяет число результатов, которые мы хотели бы получить.

Основное взаимодействие с YouTube API выполняет объект service, создаваемый вызовом функции apiclient.discovery.build(), подобно тому, как мы это делали, когда исследовали Google+ и Blogger. Функция принимает два позиционных аргумента: название службы (YouTube) и версию API (v3). Третий именованный аргумент служит для передачи ключа API, который мы сохранили в переменной окружения. После создания объекта service можно вызвать его функцию search(), чтобы создать объект search_feed, который имеет функцию list(), выполняющую фактический вызов API.

Функция list() принимает только именованные аргументы. Аргумент q определяет запрос для передачи в API. Аргумент part задает список свойств, разделен-

ных запятыми, которые API должен включить в ответ. В результате вызова возвращается объект (словарь Python), который мы для простоты выведем с помощью функции `json.dumps()`.

Словарь `search_response` имеет несколько атрибутов первого уровня:

- `pageInfo`, включающий `resultsPerPage` и `totalResults`;
- `items`: список результатов поиска;
- `kind`: тип объекта результата (в данном случае `youtube#searchListResponse`);
- `etag`: дескриптор сущности;
- `regionCode`: двухбуквенный код страны, например `GB`;
- `nextPageToken`: маркер для создания запроса на получение следующей страницы с результатами.

Каждый элемент в списке `items` представляет результат поиска, а именно идентификатор элемента (видеоролик, канал или плей-лист) и фрагмент с дополнительными атрибутами.

Ниже перечислены некоторые интересные атрибуты, сопровождающие видеоматериалы:

- `channelId`: идентификатор создателя;
- `channelTitle`: имя создателя;
- `title`: заголовок;
- `description`: текстовое описание;
- `publishedAt`: строка с датой публикации в формате ISO 8601.

Базовый запрос можно дополнить параметрами, уточняющими поисковый запрос. Например, запрос из сценария `youtube_search_video.py` получает не только видео, но также плей-листы и каналы. Чтобы ограничить результаты конкретным типом объектов, в вызов функции `list()` можно передать аргумент `type` с одним из возможных значений: `video`, `playlist` или `channel`. Кроме того, с помощью атрибута `order` можно повлиять на сортировку результатов, передав в нем одно из возможных значений:

- `date`: сортировка результатов в обратном хронологическом порядке (новые следуют первыми);
- `rating`: сортировка результатов по рейтингу, от высокого до самого низкого;
- `relevance`: (действует по умолчанию) сортировка результатов по релевантности в отношении к запросу;
- `title`: сортировка результатов в алфавитном порядке заголовков;
- `videoCount`: сортировка каналов по количеству видеороликов в порядке убывания;
- `viewCount`: сортировка в порядке убывания по количеству просмотров.

Наконец, используя аргументы `publishedBefore` и `publishedAfter`, можно также ограничить поиск конкретным диапазоном дат публикации.

Рассмотрим на примере, как на основе этих сведений создать специализированный запрос к поисковой конечной точке API. Допустим, нас интересуют видеоматериалы, опубликованные в январе 2017, отсортированные в порядке релевантности. Для этого вызов функции `list()` можно переписать, как показано ниже:

```
search_query = search_feed.list(
                                q=args.query,
                                part="id,snippet",
                                maxResults=args.n,
```

```
                                type='video',
                                publishedAfter='2017-01-01T00:00:00Z',
                                publishedBefore='2017-02-01T00:00:00Z')
```

Аргументы publishedAfter и publishedBefore принимают дату и время в формате, согласно RFC 3339.

Чтобы получить возможность обрабатывать произвольное число результатов поиска, реализуем механизм постраничного чтения. Самый простой способ достижения этой цели – завернуть вызов к поисковой конечной точке в пользовательскую функцию, выполняющую итеративный обход страниц до достижения нужного числа результатов.

Следующий класс реализует клиента YouTube с методом, специально написанным для поиска видео:

```
class YoutubeClient(object):

    def __init__(self, api_key):
        self.service = build('youtube',
                             'v3',
                             developerKey=api_key)

    def search_video(self, query, n_results):
        search = self.service.search()
        request = search.list(q=query,
                              part="id,snippet",
                              maxResults=n_results,
                              type='video')
        all_results = []
        while request and len(all_results) <= n_results:
            response = request.execute()
            try:
                for video in response['items']:
                    all_results.append(video)
            except KeyError:
                break
            request = search.list_next(request, response)
        return all_results[:int(n_results)]
```

При инициализации классу YoutubeClient передается ключ API api_key, который используется в вызове метода apiclient.discovery.build() и для настройки службы.

Основная логика поиска реализована в методе search_video(), который принимает два аргумента: запрос и число результатов. Сначала настроим объект запроса, как и прежде, используя метод list(). Цикл while сравнивает объект request со значением None и проверяет достижение заданного числа результатов. Запрос возвращает результаты в виде объекта request, являющегося, как и прежде, словарем. Данные о видеоматериалах, хранящиеся в response['items'], добавляются в список all_results. В конце цикла вызывается метод list_next(), который переопределяет объект request, готовясь к получению следующей страницы результатов.

Ниже показано, как можно использовать класс YoutubeClient:

```
# Chap08/youtube_search_video_pagination.py
import os
import json
```

```
from argparse import ArgumentParser
from apiclient.discovery import build

def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--query')
    parser.add_argument('--n', default=50)
    parser.add_argument('--output')
    return parser

if __name__ == '__main__':
    parser = get_parser()
    args = parser.parse_args()

    api_key = os.environ.get('GOOGLE_API_KEY')

    youtube = YoutubeClient(api_key)
    videos = youtube.search_video(args.query, args.n)

    with open(args.output, 'w') as f:
        for video in videos:
            f.write(json.dumps(video)+"\n")
```

Данный сценарий можно запустить следующей командой:

```
$ python youtube_search_video_pagination.py \
  --query python \
  --n 50 \
  --output videos.jsonl
```

В результате будет создан файл `videos.jsonl` с данными о 50 видеоматериалах, соответствующих строке запроса `python`. Файл имеет формат JSON Lines, как во всех предыдущих главах.

Данный раздел показал некоторые примеры взаимодействий с YouTube Data API. Использованные шаблоны очень похожи на те, что уже использовались с Google+ API и Blogger API, поэтому, освоив методику работы с одной из служб Google, ее легко можно распространить на другие.

Анализ открытого программного обеспечения на GitHub

GitHub (https://github.com) – это веб-служба, реализующая инфраструктуру для репозиториев Git. Хотя частные репозитории являются одним из ключевых элементов для платных услуг, данная служба хорошо известна тем, что предоставляет много бесплатных услуг. В дополнение к средствам распределенного управления исходным кодом Git веб-служба GitHub предлагает набор функций, которые упрощают управление проектом с открытым исходным кодом (например, отслеживание ошибок, вики-справочники, запросы на расширение и т. д.).

Программное обеспечение для управления исходным кодом. Система управления исходным кодом (управления версиями) является одним из самых важных инструментов в управлении программным обеспечением, помогая отслеживать развитие разрабатываемого программного обеспечения. Этим аспектом часто пренебрегают новички и разработчики-одиночки, но он крайне важен, когда сложный проект разрабатывается командой или независимыми разработчиками. Из преимуществ можно назвать возможность отката не-

желательного изменения (например, хранение и восстановление различных версий программного обеспечения), а также эффективное сотрудничество членов команды. Система Git, первоначально разработанная Линусом Торвольдсом (Linus Torvalds, автором ОС Linux), является одним из самых популярных инструментов управления версиями. Знание основ систем управления версиями будет одинаково выгодно как для разработчиков-новичков, так и для аналитиков и исследователей.

Веб-служба GitHub предлагает доступ к своим данным через несколько API (REST API – https://developer.github.com/v3, GraphQL API – https://developer.github.com/v4), использующих общий механизм регистрации приложения. Аутентификация приложения требуется только при получении доступа к частной информации, которая предлагается лишь подтвержденным пользователям.

REST API имеет строгие ограничения на частоту запросов. Неаутентифицированные приложения могут послать до 60 вызовов в час, что очень немного. Реализовав аутентификацию, можно увеличить предел до 5000 вызовов в час. Поисковый API также имеет специальные ограничения (10 запросов в минуту для неаутентифицированного приложения и 30 – для аутентифицированного). Аутентификацию можно выполнить тремя разными способами:

- стандартная аутентификация по имени пользователя/паролю;
- через маркер Oauth2;
- через идентификатор и секрет клиента по протоколу авторизации Oauth2.

Стандартная аутентификация требует отправки фактических имени пользователя и пароля, как определено стандартами HTTP. Аутентификация через маркер OAuth2 требует получения маркера программными средствами (https://developer.github.com/v3/oauth_authorizations/#create-a-new-authorization) и его отправки в конечные точки API в заголовке запроса или в параметре URL. Последний вариант заключается в передаче идентификатора и секрета клиента в API. Все необходимое можно получить в ходе регистрации приложения в GitHub (https://github.com/settings/applications/new). На рис. 8.2 изображена форма регистрации нового приложения.

Учетные данные, полученные в ходе регистрации приложения, можно сохранить в переменных окружения:

```
$ export GITHUB_CLIENT_ID="ваш-ид-клиента"
$ export GITHUB_CLIENT_SECRET="ваш-секрет-клиента"
```

Теперь, когда все настроено, можно приступать к взаимодействиям с GitHub API. Существует несколько библиотек для Python, которые можно использовать в качестве клиентов, все они являются сторонними разработками, то есть официально веб-службой GitHub не поддерживаются (https://developer.github.com/libraries). Для примеров в данном разделе мы выбрали библиотеку PyGithub (https://github.com/PyGithub/PyGithub), однако при желании вы можете протестировать другие библиотеки.

Установить библиотеку можно командой:

```
$ pip install PyGithub
```

Следующий сценарий позволяет выполнить поиск по конкретному имени пользователя и получает базовую информацию о его профиле и репозиториях на GitHub:

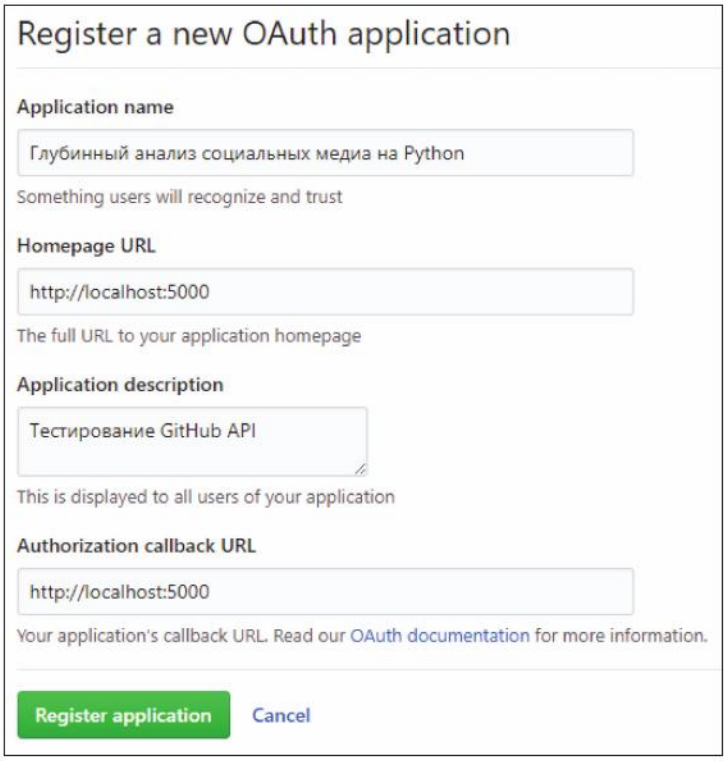

Рис. 8.2 ❖ Регистрация нового приложения в GitHub

```
# Chap08/github_get_user.py
import os
from argparse import ArgumentParser
from github import Github
from github.GithubException import UnknownObjectException

def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--user')
    parser.add_argument('--get-repos', action='store_true', default=False)
    return parser

if __name__ == '__main__':
    parser = get_parser()
    args = parser.parse_args()
    client_id = os.environ['GITHUB_CLIENT_ID']
    client_secret = os.environ['GITHUB_CLIENT_SECRET']

    g = Github(client_id=client_id, client_secret=client_secret)

    try:
        user = g.get_user(args.user)
        print("Имя пользователя: {}".format(args.user))
        print("Полное имя: {}".format(user.name))
        print("Местоположение: {}".format(user.location))
```

```
        print("Число репозиториев: {}".format(user.public_repos))
        if args.get_repos:
            repos = user.get_repos()
            for repo in repos:
                print("Репозиторий: {} ({} stars)".format(repo.name,
                      repo.stargazers_count))
    except UnknownObjectException:
        print("Пользователь не найден")
```

Для чтения аргументов командной строки данный сценарий использует экземпляр класса `ArgumentParser`: аргумент `--user` служит для передачи искомого имени пользователя, а необязательный аргумент `--get-repos` используется как логический флаг, подтверждающий необходимость включения в результаты списка репозиториев пользователя (для простоты постраничное чтение сюда не включено).

Этот сценарий можно запустить следующей командой:

```
python github_get_user.py --user bonzanini --get-repos
```

Эта команда выведет следующий результат:

```
Имя пользователя: bonzanini
Полное имя: Marco Bonzanini
Местоположение: London, UK
Число репозиториев: 15
Репозиторий: 2016.pyconuk.org (0 stars)
Репозиторий: beer2vec-analysis (0 stars)
Репозиторий: bonzanini.github.io (1 stars)
Репозиторий: Book-SocialMediaMiningPython (160 stars)
# еще репозитории ...
```

Функция `Github.get_user()` предполагает, что мы уже знаем точное имя пользователя.

Некоторые наиболее интересные атрибуты объекта пользователя перечислены в следующей таблице:

Название атрибута	Описание
avatar_url	URL-адрес изображения аватара
bio	Краткая биография пользователя
blog	Блог пользователя
company	Компания пользователя
created_at	Дата создания профиля
email	Электронная почта пользователя
followers	Число читателей пользователя
followers_url	URL-адрес для получения списка читателей пользователя
following	Число профилей, читаемых пользователем
following_url	URL-адрес для получения списка профилей, читаемых пользователем
location	Географическое местоположение пользователя
login	Имя для входа в систему
name	Полное имя
public_repos	Число общедоступных хранилищ
public_gists	Число общедоступных фрагментов кода
repos_url	URL-адрес для получения хранилищ

Научившись извлекать профиль по точному имени пользователя, обратимся к задаче поиска пользователя. Программный интерфейс GitHub API имеет конечную точку, к которой обращается функция `Github.search_users()`. Эта функция принимает запрос и дополнительные аргументы сортировки и упорядочения. Следующий сценарий реализует поиск:

```
# Chap08/github_search_user.py
import os
from argparse import ArgumentParser
from argparse import ArgumentTypeError
from github import Github

def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--query')
    parser.add_argument('--sort',
                        default='followers',
                        type=check_sort_value)
    parser.add_argument('--order',
                        default='desc',
                        type=check_order_value)
    parser.add_argument('--n', default=5, type=int)
    return parser

def check_sort_value(value):
    valid_sort_values = ['followers', 'joined', 'repositories']
    if value not in valid_sort_values:
        raise ArgumentTypeError(
                '"{}" является недопустимым значением для "sort"'
                .format(value))
    return value

def check_order_value(value):
    valid_order_values = ['asc', 'desc']
    if value not in valid_order_values:
        raise ArgumentTypeError(
                '"{}" является недопустимым значением для "order"'
                .format(value))
    return value

if __name__ == '__main__':
    parser = get_parser()
    args = parser.parse_args()
    client_id = os.environ['GITHUB_CLIENT_ID']
    client_secret = os.environ['GITHUB_CLIENT_SECRET']

    g = Github(client_id=client_id, client_secret=client_secret)

    users = g.search_users(args.query,
                           sort=args.sort,
                           order=args.order)
    for i, u in enumerate(users[:args.n]):
        print("{}) {} ({}) с {} репозиториями "
              .format(i+1, u.login, u.name, u.public_repos))
```

Для чтения аргументов командной строки он использует экземпляр класса `ArgumentParser`. Поисковый запрос передается в аргументе `--query`. API принимает также два аргумента, определяющих порядок сортировки: `--sort` и `--order`. В них допускается передавать только определенные значения, поэтому мы должны указать конкретный тип для каждого из них. Если быть точнее, вспомогательные функции `check_sort_value()` и `check_order_value()` реализуют следующую логику: если заданное значение допустимое, оно возвращается для передачи в API, иначе возбуждается исключение `ArgumentTypeError`. Наконец, экземпляр `ArgumentParser` принимает также аргумент `--n`, определяющий число результатов (по умолчанию 5).

Сценарий можно запустить командой:

```
$ python github_search_user.py \
  --query [здесь ваш запрос] \
  --sort followers \
  --order desc
```

В результате будет получен список из пяти (по умолчанию) самых популярных пользователей, то есть пользователей с самым большим числом читателей.

Аналогично можно реализовать сценарий для поиска популярных репозиториев, как показано ниже:

```
# Chap08/github_search_repos.py
import os
from argparse import ArgumentParser
from argparse import ArgumentTypeError
from github import Github

def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--query')
    parser.add_argument('--sort',
                        default='stars',
                        type=check_sort_value)
    parser.add_argument('--order',
                        default='desc',
                        type=check_order_value)
    parser.add_argument('--n', default=5, type=int)
    return parser

def check_sort_value(value):
    valid_sort_values = ['stars', 'forks', 'updated']
    if value not in valid_sort_values:
         raise ArgumentTypeError(
                '"{}" является недопустимым значением для "sort"'
                .format(value))
    return value

def check_order_value(value):
    valid_order_values = ['asc', 'desc']
    if value not in valid_order_values:
         raise ArgumentTypeError(
                '"{}" является недопустимым значением для "order"'
                .format(value))
    return value
```

```
if __name__ == '__main__':
    parser = get_parser()
    args = parser.parse_args()
    client_id = os.environ['GITHUB_CLIENT_ID']
    client_secret = os.environ['GITHUB_CLIENT_SECRET']

    g = Github(client_id=client_id, client_secret=client_secret)

    repos = g.search_repositories(args.query,
                                  sort=args.sort,
                                  order=args.order)
    for i, r in enumerate(repos[:args.n]):
        print("{}) {} - {} ({} звезд) Полное имя: {}"
              .format(i+1, r.name, r.owner.name,
                      r.stargazers_count, r.full_name))
```

Сценарий `github_search_repos.py` очень похож на предыдущий. В нем тоже используются экземпляр класса `ArgumentParser` и проверка типов аргументов `--sort` и `--order`. Для поиска репозиториев в нем используется метод `Github.search_repositories()`, который принимает запрос и несколько именованных аргументов.

Сценарий можно запустить следующей командой:

```
$ python github_search_repos.py \
  --query python \
  --sort stars \
  --order desc
```

Запрос с ключевым словом `python` вернет несколько интересных результатов:

```
1) oh-my-zsh - Robby Russell (36163 stars)
2) jQuery-File-Upload - Sebastian Tschan (23381 stars)
3) awesome-python - Vinta (20093 stars)
4) requests - Kenneth Reitz (18616 stars)
5) scrapy - Scrapy project (13652 stars)
```

В списке присутствует несколько популярных проектов на Python (таких как библиотеки requests и Scrapy), но первые два результата, похоже, не связаны с Python. Это обусловлено тем, что по умолчанию посылается запрос на поиск по полям заголовок/описание. В результате этот конкретный вызов выбрал несколько репозиториев, в описаниях которых упоминается ключевое слово `python`, но проекты в них реализованы не на Python.

Можно переделать поиск, чтобы извлекать только репозитории с проектами на определенном языке, используя этот язык в качестве главного. Для этого нужно изменить всего одну строку в коде – вызов функции `Github.search_repositories()`, добавив в него квалификатор `language`:

```
repos = g.search_repositories("language:{}".format(args.query),
                              sort=args.sort,
                              order=args.order)
```

После этого маленького изменения результаты будут выглядеть совсем иначе:

```
1) httpie by Jakub RoztoÃil (22130 stars)
2) awesome-python by Vinta (20093 stars)
3) thef*** by Vladimir Iakovlev (19868 stars)
```

```
4) flask by The Pallets Projects (19824 stars)
5) django by Django (19044 stars)
```

Сейчас в число результатов включены только проекты, реализованные главным образом на Python.

В следующей таблице перечисляются основные атрибуты объектов-репозиториев:

Название атрибута	Описание
name	Имя репозитория в том виде, каким оно появляется в URL-адресе
full_name	Полное имя репозитория (то есть user_name/repo_name)
owner	Объект, представляющий пользователя, владеющего проектом
id	Числовой идентификатор репозитория
stargazers_count	Количество звезд
description	Текстовое описание репозитория
created_at	Объект datetime со временем создания
updated_at	Объект datetime со временем последнего обновления
open_issues_count	Число нерешенных вопросов
language	Главный язык проекта
languages_url	URL-адрес для получения информации о других языках
homepage	URL-адрес домашней страницы проекта

Как видите, для каждого конкретного хранилища возвращается много атрибутов и деталей. Исчерпывающее их описание можно найти в документации к API (https://developer.github.com/v3/repos).

В данном разделе было показано несколько примеров извлечения данных из GitHub API. По этим данным можно узнать, например, какие языки программирования используются наиболее часто, выявить наиболее активные проекты по числу вопросов, открытых в тот или иной период времени, и т. д.

Анализ сведений о местных предприятиях в Yelp

Yelp – это оперативная информационная служба для размещения отзывов потребителей о местных предприятиях, таких как бары и рестораны (https://www.yelp.com). Информационное наполнение этой службы, которая ежемесячно насчитывает 135 миллионов посещений, создается главным образом сообществом.

Веб-служба Yelp предлагает три API для поиска данных о местных предприятиях. Эти три API сгруппированы в зависимости от цели. Поисковый API предназначен для поиска по ключевым словам. Деловой API позволяет получить информацию об определенном предприятии. Телефонный поисковый API используется для поиска предприятия по номеру телефона.

Yelp API ограничивает частоту запросов 25 000 обращений в день.

В этом разделе описываются основные шаги настройки приложения, выполняющие поиск в базе данных Yelp конкретных ключевых слов.

Доступ к API защищен маркерами. Поэтому, прежде чем начать использовать любой API, нужно зарегистрировать учетную запись в Yelp и получить маркеры на веб-сайте разработчиков (https://www.yelp.com/developers). На рис. 8.3 показана

форма, которую следует заполнить, чтобы получить доступ к API. В данном примере в поле Your website URL (URL-адрес вашего веб-сайта) указан локальный хост `localhost`:

Complete the following form to gain access to the Yelp API.

Your website URL:

http://127.0.0.1:5000

How you will use the API:

Testing Yelp API capabilities

Industry:

Other

Publishing

Рис. 8.3 ❖ Доступ к Yelp API

Выполнив первый шаг, мы получим четыре разных ключа доступа (ключ потребителя, секрет потребителя, маркер и секрет маркера). Эти строковые значения следует хранить в защищенном месте и никогда никому их не передавать. Они используются для аутентификации в OAuth, которую нужно выполнить до обращений к API.

Следуя соглашениям, принятым в настоящей книге, сохраним ключи в переменных окружения:

```
$ export YELP_CONSUMER_KEY="ваш-ключ-потребителя"
$ export YELP_CONSUMER_SECRET="ваш-секрет-потребителя"
$ export YELP_TOKEN="ваш-маркер"
$ export YELP_TOKEN_SECRET="ваш-секрет-маркера"
```

Чтобы получить доступ к API программными средствами, необходимо установить официальную клиентскую библиотеку для Python, например так:

```
$ pip install yelp
```

Следующий сценарий `yelp_client.py` определяет функцию, которую мы будем использовать для подготовки к использованию API:

```
# Chap08/yelp_client.py
import os
from yelp.client import Client
from yelp.oauth1_authenticator import Oauth1Authenticator

def get_yelp_client():
    auth = Oauth1Authenticator(
        consumer_key=os.environ['YELP_CONSUMER_KEY'],
        consumer_secret=os.environ['YELP_CONSUMER_SECRET'],
        token=os.environ['YELP_TOKEN'],
```

```
        token_secret=os.environ['YELP_TOKEN_SECRET']
    )

    client = Client(auth)
    return client
```

Теперь все готово и можно выполнить первый поиск местного предприятия. Сценарий yelp_search_business.py вызывает поисковый API, передавая ему местоположение и ключевые слова:

```
# Chap08/yelp_search_business.py
from argparse import ArgumentParser
from yelp_client import get_yelp_client

def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--location')
    parser.add_argument('--search')
    parser.add_argument('--language', default='en')
    parser.add_argument('--limit', default=20)
    parser.add_argument('--sort', default=2)
    return parser

if __name__ == '__main__':
    client = get_yelp_client()
    parser = get_parser()
    args = parser.parse_args()

    params = {
        'term': args.search,
        'lang': args.language,
        'limit': args.limit,
        'sort': args.sort
    }

    response = client.search(args.location, **params)
    for business in response.businesses:
        address = ', '.join(business.location.address)
        categories = ', '.join([cat[0] for cat in business.categories])
        print("{} id={} ({}, {}); рейтинг {}; категории
              {}".format(business.name, business.id,
              address,
              business.location.postal_code,
              business.rating,
              categories))
```

Для чтения аргументов командной строки используется объект ArgumentParser. Сценарий принимает два обязательных аргумента: --location, определяющий город или местность, и аргумент --search с ключевыми словами для поиска. Также можно передать третий, необязательный аргумент --language, чтобы указать язык для вывода результатов (в особенности для отзывов). В этом аргументе должен передаваться код языка (например, en – для английского, fr – для французского, de – для немецкого и т. д.). Есть еще два необязательных аргумента. Аргумент --limit, ограничивающий число результатов (по умолчанию равно 20), и аргумент --sort, определяющий вид сортировки результатов (0 – по оптимальному соответствию, 1 – по расстоянию и 2 – по рейтингу).

В главной части сценария выполняется настройка клиента Yelp и парсера. Затем определяется словарь параметров для передачи клиенту Yelp. Эти параметры получают значения из аргументов командной строки.

Сценарий можно запустить следующей командой:

```
$ python yelp_search_business.py \
  --location London \
  --search breakfast \
  --limit 5
```

В результате он выведет список из (пяти в данном случае) местных предприятий:

```
Friends of Ours (61 Pitfield Street, N1 6BU); рейтинг 4.5; категории Cafes,
Breakfast & Brunch, Coffee & Tea
Maison D'être (154 Canonbury Road, N1 2UP); рейтинг 4.5; категории Coffee &
Tea, Breakfast & Brunch
Dishoom (5 Stable Street, N1C 4AB); рейтинг 4.5; категории Indian, Breakfast & Brunch
E Pellicci (332 Bethnal Green Road, E2 0AG); рейтинг 4.5; категории Italian, Cafes, Breakfast
& Brunch
Alchemy (8 Ludgate Broadway, EC4V 6DU); рейтинг 4.5; категории Coffee & Tea, Cafes
```

Запрос на поиск по нескольким ключевым словам должен заключаться в двойные кавычки, чтобы исключить его неверную интерпретацию командной оболочкой, например:

```
$ python yelp_search_business.py \
  --location "San Francisco" \
  --search "craft beer" \
  --limit 5
```

Объекты `business`, возвращаемые API и представляющие предприятия, имеют несколько свойств, наиболее интересные из которых перечислены в следующей таблице:

Название атрибута	Описание
`id`	Идентификатор предприятия
`name`	Название предприятия
`image_url`	URL-адрес фотографии предприятия
`is_claimed`	Логический признак изъятия страницы владельцем предприятия
`is_closed`	Логический признак ликвидации предприятия
`url`	URL-адрес предприятия в Yelp
`mobile_url`	Мобильный URL-адрес предприятия в Yelp
`phone`	Строка с телефонным номером в международном формате
`display_phone`	Строка с телефонным номером для отображения на экране
`categories`	Список категорий, связанных с бизнесом. Каждая категория – это кортеж `(display_name, filter_name)`, первый элемент – название для отображения на экране, а второй – для фильтрации запроса
`review_count`	Число отзывов для этого предприятия
`rating`	Округленный рейтинг предприятия (например, 1, 1.5... 4.5,5)
`snippet_text`	Краткое описание предприятия
`snippet_image_url`	URL-адрес фрагмента изображения, связанного с предприятием
`location`	Объект `location`, связанный с предприятием

Объект `location` имеет следующие атрибуты:

- `address` (список) – содержит только поля адреса;
- `display_address` (список) – адрес для отображения на экране, включает пересечение улиц, город, код области/штата и т. д.;
- `city` (строка);
- `state_code` (строка) – код ISO 3166-2 штата предприятия;
- `postal_code` (строка);
- `country_code` (строка) – код ISO 3166-1 страны предприятия;
- `cross_streets` (строка) – пересечение улиц, где находится предприятие (США);
- `neighborhoods` (список) – информация об окрестностях предприятия;
- `coordinates.latitude` (число) – географическая широта;
- `coordinates.longitude` (число) – географическая долгота.

Деловой API веб-службы Yelp создан для получения информации о конкретном предприятии. Данный API предполагает, что вам уже известен конкретный идентификатор предприятия.

Следующий сценарий демонстрирует пример использования клиента для Python для доступа к деловому API:

```
# Chap08/yelp_get_business.py
from argparse import ArgumentParser
from yelp_client import get_yelp_client

def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--id')
    parser.add_argument('--language', default='en')
    return parser

if __name__ == '__main__':
    client = get_yelp_client()
    parser = get_parser()
    args = parser.parse_args()

    params = {
        'lang': args.language
    }

    response = client.get_business(args.id, **params)
    business = response.business
    print("Количество отзывов: {}".format(business.review_count))
    for review in business.reviews:
        print("{} (автор {})".format(review.excerpt, review.user.name))
```

Одно из главных ограничений делового API – возврат неполного списка отзывов о предприятии. Фактически возвращается только один отзыв. Этот отзыв является единственным отличием делового API от поискового, поэтому он не дает особых выгод с точки зрения анализа данных.

Интересно отметить, что веб-служба Yelp предоставляет набор данных для академических исследований, который время от времени обновляется и расширяется (https://www.yelp.com/dataset_challenge). Этот набор данных включает подробное описание предприятий, пользователей, отзывов, изображений

и связей между пользователями. Эти данные могут использоваться для приложений в области обработки естественного языка, анализа графов и машинного обучения в целом. Указанный набор данных также связан с конкурсными мероприятиями, где студентов награждают за техническое качество поданных на конкурс работ.

Создание собственного клиента на Python

На протяжении всей книги для доступа к различным социально-медийным платформам мы использовали библиотеки для Python, официально поддерживаемые самими платформами или разработанные третьими лицами.

Данный раздел отвечает на вопрос: что делать, если для какого-то API не удалось найти клиентскую библиотеку на Python?

Простой интерфейс для вызовов по протоколу HTTP

Для организации простых взаимодействий с API по протоколу HTTP можно использовать библиотеку **requests**. Установить ее можно как обычно:

```
$ pip install requests
```

Библиотека requests предлагает простой интерфейс для выполнения вызовов по протоколу HTTP. Для примера реализуем простого клиента веб-службы `httpbin.org`, выполняющего элементарные взаимодействия по схеме «запрос/ответ».

На странице веб-службы (http://httpbin.org/) можно найти описание различных конечных точек. Для демонстрации практического применения библиотеки requests реализуем класс `HttpBinClient` с методами, вызывающими некоторые основные конечные точки:

```
import json
import requests

class HttpBinClient(object):

    base_url = 'http://httpbin.org'

    def get_ip(self):
        response = requests.get("{}/ip".format(self.base_url))
        my_ip = response.json()['origin']
        return my_ip

    def get_user_agent(self):
        response = requests.get("{}/user-agent".format(self.base_url))
        user_agent = response.json()['user-agent']
        return user_agent

    def get_headers(self):
        response = requests.get("{}/headers".format(self.base_url))
        headers = HeadersModel(response.json())
        return headers

    def get_image(self, image_format):
        image_map = {
            'png': 'image/png',
```

```
        'jpg': 'image/jpg'
    }
    try:
        headers = {'Accept': image_map[image_format]}
    except KeyError:
        raise Exception("Формат {} недопустим".format(image_format))
    r = requests.get("{}/image", headers=headers)
```

Данный клиент имеет переменную класса `base_url` для хранения базового URL-адреса. Методы `get_ip()`, `get_user_agent()` и `get_headers()` вызывают конечные точки `/ip`, `/user-agent` и `/headers`, соответственно конструируя полный URL-адрес на основе `base_url`.

Все три метода практически одинаково используют библиотеку requests. Они посылают конечной точке HTTP GET-запрос, вызывая метод `requests.get()`, и получают объект с ответом. Этот объект имеет несколько атрибутов, таких как `status_code` (состояние отклика) и `text` (данные отклика), а также метод `json()`, который мы использовали для загрузки отклика в формате JSON в словарь.

IP-адрес и информация о клиентском приложении (user-agent) – это простые строковые значения, но объект `headers` немного сложнее. Он содержит несколько атрибутов. Поэтому мы завернули отклик в специальный объект `HeadersModel`:

```
class HeadersModel(object):

    def __init__(self, data):
        self.host = data['headers']['Host']
        self.user_agent = data['headers']['User-Agent']
        self.accept = data['headers']['Accept']
```

Сохранив предыдущие объявления классов в файле `httpbin_client.py`, мы сможем применить свой клиентский модуль из демонстрационного приложения, как показано ниже:

```
>>> from httpbin_client import HttpBinClient
>>> client = HttpBinClient()
>>> print(client.get_user_agent())
python-requests/2.8.1
>>> h = client.get_headers()
>>> print(h.host)
httpbin.org
```

Простота веб-API и предложенного примера не должна подводить читателя к ошибочному выводу, что мы просто добавили лишний уровень сложности в простую задачу и фактически, пользуясь библиотекой requests, могли бы взаимодействовать с веб-API непосредственно из приложения. С одной стороны, это, конечно, возможно, но с другой – социально-медийные API по своей природе чрезвычайно динамичны. То и дело добавляются новые конечные точки, изменяются форматы ответов и постоянно разрабатываются новые возможности. Например, когда уже был готов черновик главы 4 «*Сообщения, страницы и взаимодействие пользователей в Facebook*», компания Facebook внедрила новую функцию «Реакции», из-за чего произошли изменения в структуре ответа.

Другими словами, собственный клиент Python добавляет уровень абстракции, позволяющий сосредоточиться на интерфейсе клиента, а не на веб-API. Разраба-

тывая приложение, мы можем сосредоточиться на вызовах методов клиента, таких как `HttpBinClient.get_headers()`, и оградить себя от подробностей, как конечная точка API `/headers` возвращает ответ внутри такого метода. Если формат ответа веб-API изменится в будущем, от нас потребуется только обновить клиента, чтобы отразить новые изменения, и все наши приложения, взаимодействующие с веб-API, продолжат работать как обычно, благодаря этому дополнительному уровню абстракции.

Служба `httpbin.org` в действительности несложная, поэтому реализации клиентов для других API, возможно, получатся не такими прямолинейными, но в целом суть в следующем: скрыть детали реализации при помощи уровня абстракции. Именно это делают все клиенты для Python, которых мы до сих пор использовали.

Коды состояния HTTP. Для лучшего понимания взаимодействий по протоколу HTTP следует подчеркнуть важность кода состояния ответа. На низком уровне каждый ответ сервера HTTP начинается с числового кода состояния, сопровождаемого строкой с описанием этого кода состояния. Например, когда мы открываем в браузере конкретный URL-адрес, веб-сервер может ответить кодом состояния `200` со строкой описания `OK`. В этом случае ответ содержит фактическую веб-страницу для отображения в браузере. В некоторых случаях веб-сервер может вернуть код состояния `404` с описанием `Not Found` (Страница не найдена). Браузер, обнаружив этот код, показывает соответствующее сообщение об ошибке.

Многие веб-API действуют аналогично. В зависимости от запроса сервер может вернуть ответ с кодом состояния `2xx`, сообщающим об успехе, или с признаком ошибки `4xx` на стороне клиента или `5xx` – на стороне сервера. Реализация клиента должна уметь правильно интерпретировать различные коды состояний в соответствии с веб-API и действовать соответственно. Также стоит упомянуть, что некоторые службы могут возвращать нестандартные коды, как, например Twitter, возвращающий ошибку `420 Enhance your calm` (что примерно можно перевести как «Возьмите себя в руки»), когда клиент посылает слишком много запросов и должен быть ограничен по частоте запросов.

Полный список кодов состояний и их значений с кратким описанием можно найти на странице Википедии https://ru.wikipedia.org/wiki/Список_кодов_состояния_HTTP, а официальное описание – в соответствующих документах RFC.

Учитывая доступность библиотек с открытым исходным кодом, в наше время редко возникает потребность писать своего клиента с нуля. Но если она все же возникнет, вы теперь в достаточной мере готовы, чтобы встретить ее.

Резюме

В настоящей главе обсуждались некоторые дополнительные возможности для анализа данных социальных медиа. Кроме обычных популярных социальных сетей, таких как Twitter, Facebook и Google+, существует много других платформ, предлагающих API для доступа к своим данным.

Если клиент на Python для конкретной платформы отсутствует, можно написать свой клиентский модуль, задействовав простую библиотеку requests. На первый взгляд реализация клиентского модуля может показаться ненужной сложностью, однако он обеспечит дополнительный уровень абстракции, что в конечном счете окупится в долгосрочной перспективе.

Помимо инструкций по реализации собственного клиента, мы также исследовали некоторые другие популярные службы с целью анализа сложных объектов. Мы получили данные о видеоматериалах из YouTube, проектах с открытым исходным кодом из GitHub и местных предприятиях из Yelp, таких как рестораны.

Существует еще много других социально-медийных платформ, которые остались за рамками настоящей книги, потому что всестороннее обсуждение этой темы просто нереалистично, но мы надеемся, что читатели достаточно хорошо разбираются во многих существующих в сети возможностях.

Следующая глава – последняя в этой книге. Она посвящена Семантической паутине и поможет читателям понять важность семантической разметки данных.

Глава 9

Связанные данные и Семантическая паутина

Настоящая глава дает обзор Семантической паутины и связанных с ней технологий. В ней мы обсудим следующие темы:

- основы Семантической паутины как паутины данных;
- введение в микроформаты, связанные данные и RDF;
- анализ семантических связей из проекта DBpedia;
- анализ географической информации из Википедии;
- отображение географической информации на картах Google Maps.

Паутина данных

Международная организация World Wide Web Consortium (W3C), которая занимается разработкой веб-стандартов, предлагает простое определение термина «Семантическая паутина» (https://www.w3.org/standards/semanticweb/data, взято в апреле 2016 г.):

> «*Семантическая паутина – это паутина данных*».

Термин «*семантическая паутина*», как и данное определение, был предложен изобретателем Всемирной паутины, самим сэром Тимом Бернерсом-Ли (Tim Berners-Lee, https://ru.wikipedia.org/wiki/Бернерс-Ли_Тим), который также является директором W3C. Когда он говорит о своем величайшем изобретении, он часто подчеркивает его социальные последствия и что веб (Всемирная паутина) – больше социальное творение, чем техническое (*Weaving the Web*, Тим Бернерс-Ли, 1999).

Роль Всемирной паутины как социальной платформы становится очевиднее, если кратко проанализировать ее развитие. Одним из ключевых слов (или модных словечек), получившим популярность в первом десятилетии нового тысячелетия, было слово *Веб 2.0*, выдуманное в конце 1990-х и затем популяризированное Тимом О'Райли (Tim O'Reilly). Данный термин предлагает новую версию веба, но не относится ни к какому конкретному техническому обновлению или изменению в технических требованиях, а напротив – касается постепенного эволюционного перехода от старой статической паутины Веб 1.0 к ориентированному на пользователя пониманию веба и опыту взаимодействия с широкими функциональными возможностями.

Эволюционный переход от Веб 1.0 к Веб 2.0 можно охарактеризовать одним словом – сотрудничество. Тим Бернерс-Ли не согласен с таким разграничением, потому что технологии и стандарты, используемые в Веб 2.0, по сути, являются теми же, что и в старом Веб 1.0. Более того, сотрудничество и объединение людей были исходными целями с момента появления Всемирной паутины. Несмотря на это выдающееся мнение, термин «Веб 2.0» – жаргонизм это или нет – стал составной частью нашего словаря, часто в сочетании с термином «социальная паутина» (который, что еще больше осложняет ситуацию, иногда упоминается как «Веб 2.X»).

Итак, куда же нам поместить Семантическую паутину? По словам создателя веба, это будет следующим шагом в эволюционной шкале, то есть Веб 3.0.

Естественное развитие Всемирной паутины заключается в ее постепенном становлении как паутины данных. Первостепенную роль в этом движении играют модели и представления данных, которые позволяют делиться знаниями понятным для машин способом. Используя данные, доступные в непротиворечивом и семантическом формате, машина может пользоваться этой информацией, поддерживать сложные информационные потребности пользователей и способствовать принятию решений.

Доминирующим форматом для документов во Всемирной паутине является HTML. Эта согласованная разметка используется для описания структуры документа, в котором сочетаются текст, мультимедийные объекты, такие как изображения или видео, и связи между документами.

HTML позволяет кураторам информационного материала определять некоторые метаданные в заголовке документа. Такая информация не обязательно предназначена для отображения, поскольку браузер, как правило, показывает только тело документа. Теги метаданных включают имя автора, информацию об авторском праве, краткое описание документов и ключевые слова, описывающие документ. Все эти подробности могут интерпретироваться компьютером с целью классификации документа. Однако возможностей HTML недостаточно для определения более сложной информации.

Например, используя метаданные HTML, можно указать, что документ посвящен Вуди Аллену (Woody Allen), но язык разметки не поддерживает возможности разрешения неоднозначности более замысловатых понятий. Например, о чем говорит упоминание о Вуди Аллене? Что документ о герое фильма, чью роль сыграл Вуди Аллен? Что кинофильм снят Вуди Алленом? Или что это документальный фильм о Вуди Аллене, снятый кем-то еще? Поскольку цель HTML – описывать структуру документа, этот язык оказывается не лучшим инструментом для представления знаний на таком уровне детальности.

С другой стороны, технологии Семантической паутины позволяют сделать шаг вперед и описывать понятия, характеризующие наш мир с точки зрения его сущностей, их атрибутов и связей между ними. Например, фрагмент фильмографии Аллена можно представить в виде следующей таблицы:

Художник	Роль	Кинофильм
Вуди Аллен	Режиссер	Манхэттен
Вуди Аллен	Актер	Манхэттен
Вуди Аллен	Режиссер	Матч-пойнт

Приведенная таблица явно показывает связь между типами сущностей «*Художник*» и «*Кинофильм*». Представление этого вида структурированного знания является территорией Семантической паутины.

В оставшейся части данного раздела будет предоставлен широкий обзор главных концепций и терминологии, связанных с Семантической паутиной и представлением знаний. В ней вы познакомитесь с технологиями, которые призваны отвечать на сложные запросы, как в примере с Вуди Алленом.

Словарь Семантической паутины

Данный раздел кратко перечислит и опишет часть базового словаря, который широко используется в более широком контексте семантических технологий.

Язык разметки Это система аннотирования документов. Язык разметки позволяет кураторам информационного материала отмечать блоки текста (данные) согласно определенному значению языка разметки. Хорошо известным языком разметки, с которым мы уже столкивались, является HTML. Языки разметки могут содержать аннотации как для представления, так и для описания. Первые определяют, как должны отображаться документы, а последние описывают аннотируемые данные. Например, в HTML имеются оба вида тегов: представления и описания. Теги `<b>` или `<i>` определяют стиль визуального оформления (представления), и дизайнер может пользоваться ими, чтобы указывать, что конкретная часть текста должна отображаться жирным или наклонным шрифтом. С другой стороны, теги `<strong>` и `<em>` несут определенную семантическую нагрузку, так как явно указывают на важность фрагмента текста. В обычных браузерах содержимое обоих тегов – `<b>` и `<strong>` – отображается жирным шрифтом, так же как содержимое обоих тегов – `<i>` и `<em>` – отображается наклонным шрифтом. Слабовидящие пользователи не могут видеть выделения жирным шрифтом, но разметка фразы как `<strong>` позволяет программе чтения с экрана понять, как сама фраза должна быть прочитана.

Семантическая разметка Как уже говорилось, между представлением и смысловой нагрузкой элементов существует разница. Семантическая разметка всецело касается описания значения информации, а не ее внешнего вида. В HTML и XHTML теги разметки, отвечающие за визуальное представление, не объявлены устаревшими явно, хотя и не рекомендуются для использования. HTML5 на один шаг приблизился к семантической разметке, введя несколько семантических тегов, таких как `<article>` и `<section>`. В то же время теги визуального представления, такие как `<b>` и `<i>`, были сохранены с точным значением (то есть чтобы стилистически отличать выделенный текст от обычного без передачи какой-то особой важности).

Онтология Одним из основополагающих элементов Семантической паутины являются онтологии. Это формальные схемы именования и определения типов, атрибутов и связей сущностей в конкретной предметной области. В числе примеров онтологий можно назвать лексический ресурс WordNet (https://wordnet.princeton.edu/), где термины классифицированы в понятия согласно их значению, и медицинский словарь SNOMED CT (http://www.ihtsdo.org/snomed-ct).

Википедия демонстрирует небольшой фрагмент из WordNet (https://en.wikipedia.org/wiki/WordNet):

```
собака, домашняя собака, Canis familiaris
  => представитель семейства псовых, псовые
    => плотоядное животное, хищное
      => плацентарное, плацентарное млекопитающее, высшие звери
        => млекопитающее
          => позвоночное, черепное
            => хордовые
              => животное, живое существо, зверь, тварь, существо, фауна
                => ...
```

В данном примере представлено одно из значений слова «*собака*» и можно оценить уровень детализации, используемый для представления этого знания: собака является не просто животным, но также классифицируется согласно принадлежности конкретному биологическому семейству (псовые) или отряду (хищник).

Онтологии можно описывать при помощи семантической разметки, например **язык описания онтологий OWL** (Web Ontology Language) **для Семантической паутины** (https://en.wikipedia.org/wiki/Web_Ontology_Language) позволяет представлять знания. С построением онтологии связаны такие проблемы, как необъятность представляемых предметных областей, а также свойственная неоднозначность и неопределенность, которая окружает человеческие знания и естественный язык как механизм передачи знания. Создание онтологии вручную требует глубокого понимания моделируемой предметной области. Описание и представление знаний были одними из основных проблем человечества с первых лет философии, поэтому онтологии Семантической паутины можно рассматривать как практическое применение философской онтологии.

Таксономия Более узкие, чем онтологии, таксономии связаны с иерархическим представлением знаний. Другими словами, они используются для выделения четко определенных классов сущностей, определяя связи **is-a** («является», или «наследует») и **has-a** («владеет», или «включает») между классами. Основное различие между онтологиями и таксономиями в том, что онтологии моделируют более широкое разнообразие связей.

Фолксономия Также известная как социальная таксономия. Практика построения фолксономии связана с социальной разметкой. Пользователи могут отмечать фрагменты информационного наполнения, используя конкретный тег, соответствующий конкретной категории. Результатом такого процесса является своеобразная демократизация таксономии, которая приводит к отображению информации в том виде, в каком пользователи ее воспринимают без наложения жесткой структуры. С другой стороны, фолксономии в естественной среде могут быть полностью дезорганизованы, и их тяжелее использовать. Кроме того, они могут представлять отдельные тенденции, а не глубокое понимание описываемой предметной области.

В контексте социальных медиа классическим примером фолксономии является социальная сеть Twitter. Используя хештеги, пользователи могут отмечать свои твиты как принадлежащие конкретной теме. Это дает возможность другим поль-

зователям находить определенные темы или следить за событиями. Особенности фолксономий довольно хорошо проявляются в использовании хештегов в Twitter – они выбираются пользователем, публикующим твит, не имеют иерархии и формально не организованы, но подхватываются толпой, и, как следствие, становятся модными (демократизация) только те, которые рассматриваются как значащие.

Логический вывод и логическое обоснование В области формальной логики логический вывод – это процесс получения выводов из известных фактов, о которых известно (или предполагается), что они истинны. Еще один термин, который часто появляется в этом контексте, – *логическое обоснование*. На эту дихотомию можно взглянуть с другой стороны, когда логическое обоснование видится как цель, а логический вывод – как ее реализация.

Одним из классических примеров логического вывода является силлогизм Сократа. Рассмотрим следующие факты (утверждения, которые считаются истинными):

- всякий человек смертен;
- Сократ – человек.

Исходя из этих предпосылок, рассуждающая система должна суметь ответить на вопрос: «*Смертен ли Сократ?*» Пробирающим до пяток ответом будет «*Да*».

Обобщенный случай силлогизма:

- Всякий *A* является *B*;
- *C* является *A*;
- Следовательно, *C* является *B*.

Здесь *A*, *B* и *C* могут быть категориями или людьми. Силлогизмы являются классическим примером дедуктивного рассуждения. В целом глаголы «*заключая из*», «*делая вывод из*» или «*рассуждая над*» в нашем контексте являются в значительной степени синонимами, поскольку они все требуют вывода нового знания из некоторых известных (истинных) фактов.

С учетом приведенной выше базы знаний о Сократе и других людях, чтобы отвечать на такие вопросы, как «*Смертен ли Платон?*», мы должны рассмотреть следующий абзац.

Допущения о закрытости и открытости мира Рассуждающая система может содержать допущения о закрытости или открытости мира относительно Вселенной. Эта разница определяет, как делаются выводы из (ограниченного) набора известных фактов. В системах, основанных на допущении о закрытом мире, каждый неизвестный факт рассматривается как ложный. Системы, основанные на допущении об открытом мире, с другой стороны, допускают, что неизвестные факты могут быть или не быть истинными, и поэтому они не заставляют интерпретировать неизвестное как ложное. Тем самым они могут иметь дело с неполным знанием, другими словами, с базами знаний, которые лишь частично определены либо уточняются со временем.

Например, классический язык логического программирования **Prolog** (название языка произошло от Programming in Logic) основан на допущении о закрытости мира. База знаний о Сократе синтаксически выражается на Prolog так:

```
mortal(X) :- man(X).
man(socrates).
```

На вопрос, смертен ли Платон, будет получен ответ «*Нет*». Это вызвано тем, что система не имеет знаний о Платоне и поэтому не может вывести дальнейшие факты.

Дуализм между допущениями о закрытости и открытости мира также сказывается на подходах к объединию разных баз знаний. Если две базы знаний предоставляют противоречивые факты, система, основанная на допущении о закрытости мира, инициирует ошибку, поскольку не может справиться с этим типом несоответствия. Система, основанная на допущении об открытости мира, напротив, попытается преодолеть несоответствие, произведя дополнительные знания, и попытается найти способ разобраться в несоответствии. Решение состоит в назначении фактам вероятностей: для простоты два противоречащих друг другу факта могут быть на 50% истинными.

В качестве примера допущения о закрытости мира рассмотрим базу данных бронирования билетов авиакомпании. Система имеет полную информацию о конкретной предметной области. В результате она четко отвечает на некоторые вопросы: если места распределяются при регистрации и пассажир забронировал билет на рейс, но не получил места, можно заключить, что он еще не зарегистрировался и ему следует отправить электронное письмо, чтобы напомнить о возможности пройти регистрацию онлайн.

С другой стороны, допущение об открытости мира применяется, когда система не обладает полной информацией о той или иной предметной области. Например, веб-платформа для онлайн-рекламы свободных вакансий знает определенное местоположение той или иной вакансии и где находится заявитель. Если компания ищет сотрудника в свой офис в Нью-Йорке и получает заявление от профессионала, находящегося в Европе, система не может сделать вывод о том, имеет ли профессионал право работать в США, используя только то знание, которое описано в этом разделе. Поэтому подобной системе, возможно, потребуется явно задать заявителю этот вопрос, прежде чем принять такое заявление к рассмотрению.

Формальная логика и логическое программирование являются обширными областями исследований, поэтому в коротком разделе невозможно просеять все знание, которое является важным для оценки всех аспектов данной тематики. Поэтому мы закончим данный раздел упоминанием, что, вообще говоря, в вебе будет безопаснее, если допустить, что база знаний неполная, потому что веб является системой с неполной информацией. По сути, такие платформы, как **RDF** (Resource Description Framework), основаны на допущении об открытости мира.

Микроформаты

Микроформаты (http://microformats.org) расширяют HTML и позволяют автору определять машиночитаемые семантические аннотации о людях, организациях, событиях и практически любом другом типе объектов. По сути, это согласованные правила, которые предлагают создателям информационного наполнения возможность встраивать в веб-страницы недвусмысленно структурированные данные. Новейшие разработки в этой сфере воплощены в стандарте микроформата **microformat2** (http://microformats.org/wiki/microformats2).

Вот примеры интересных микроформатов:

- `h-card`: представляет людей, организации и связанную с ними контактную информацию;

- h-event: представляет информацию о событии, такую как местоположение и время начала;
- h-geo: представляет географические координаты, которые используются вместе с h-card и h-event;
- **XHTML Friends Network** (XFN, микроформат для разметки социальных взаимоотношений): используется для представления связей между людьми посредством гиперссылок;
- h-resume: публикует резюме и биографические справки в вебе, включая образование, опыт и навыки;
- h-product: определяет такие данные о продуктах, как бренд, цена или описание;
- h-recipe: представляет рецепты, включая ингредиенты, количества и инструкции по применению;
- h-review: публикует сводную информацию о любом предмете (например, ссылаясь на h-card, h-event, h-geo или h-product), включая информацию о рейтинге и описание.

Следующий фрагмент содержит пример разметки XFN, используемой для описания человеческих связей посредством гиперссылок:

```
<div>
  <a href="http://marcobonzanini.com" rel="me">Marco<a>
  <a href="http://example.org/Peter" rel="friend met">Peter<a>
  <a href="http://example.org/Mary" rel="friend met">Mary<a>
  <a href="http://example.org/John" rel="friend">John<a>
</div>
```

XFN добавляет в обычный HTML атрибут rel. Некоторые ссылки HTML были отмечены согласно связям между людьми. Первая ссылка отмечена как me, то есть она связана с автором документа. Другие отмечены классами friend и met, из которых последний указывает, что кто-то встречался лично (в противоположность онлайн-связи). В числе других классов можно назвать: spouse (супруг(а)), child (ребенок), parent (родитель), acquaintance (знакомый) или co-worker (коллега).

Следующий фрагмент содержит пример разметки h-geo для представления географических координат Лондона:

```
<p class="h-geo">
  <span class="p-latitude">51.50722</span>,
  <span class="p-longitude">-0.12750</span>
</p>
```

h-geo является последней версией данного микроформата, и в наши дни еще широко используется старая версия (просто geo). Предыдущий пример можно переписать в старом формате:

```
<div class="geo">
  <span class="latitude">51.50722</span>,
  <span class="longitude">-0.12750</span>
</div>
```

Если эти примеры микроформатов кажутся довольно простыми для понимания, то это потому, что микроформаты предназначены быть простыми. Часть «микро» в названии термина «микроформат» означает особое свойство концент-

рации на очень конкретной области. Микроформаты решают четко очерченную задачу в ограниченной области (такой как описание человеческих связей или предоставление географических координат) и ничего больше. Этот аспект микроформатов также делает их легко компонуемыми — в составном документе можно использовать несколько микроформатов с целью обогатить предоставленную информацию. Более того, некоторые микроформаты могут включать другие для передачи более полных данных — микроформат `h-card` может включать `h-geo`, а микроформат `h-event` может содержать микроформаты `h-card` и `h-geo` и т. д.

Связанные данные и открытые данные

Термин «*связанные данные*» относится к набору принципов, применяемых для публикации структурированных данных в вебе, используя стандарты W3C способом, который благоприятствует специализированным алгоритмам, задействующим связи между данными.

Поскольку Семантическая паутина касается не только размещения данных в вебе, но и позволяет людям (и машинам) пользоваться и исследовать эти данные, Тим Бернерс-Ли предложил набор правил, чтобы стимулировать распространение связанных данных и переход к тому, что он называет *связанными открытыми данными*, то есть связанными данными, доступными благодаря открытой лицензии (https://www.w3.org/DesignIssues/LinkedData.html).

Аналогично Всемирной паутине как сети документов, связанных гиперссылками, паутина данных тоже основана на документах, опубликованных в вебе и связанных ссылками. С другой стороны, паутина данных касается данных в универсальной форме, а не только документов. И предпочтительным форматом описания произвольных *вещей* является RDF, а не (X)HTML.

Вот четыре правила, предложенные Бернерсом-Ли как руководство для реализации связанных данных:

- использовать URI-адреса как названия вещей;
- использовать HTTP URI-адреса, чтобы люди могли просматривать эти названия;
- когда кто-то обращается к URI-адресу, предоставлять полезную информацию, используя стандарты (RDF*, SPARQL);
- включать ссылки на другие URI-адреса, чтобы они могли обнаруживать еще какие-то вещи.

Эти базовые принципы обеспечивают ряд ожиданий, которые в целях создания связанных данных должны удовлетворяться публикаторами информации.

Чтобы поощрить использование связанных данных, была разработана пятизвездочная система оценок, чтобы люди могли сами оценивать свои связанные данные и понимать важность связанных открытых данных. В особенности это касается публикации данных, имеющих отношение к правительствам, что способствует движению к прозрачности. Пятизвездочная система оценок описывается следующим образом:

- **одна звезда**: имеется в вебе (в любом формате) с открытой лицензией;
- **две звезды**: имеется в виде машиночитаемых структурированных данных (например, в формате Excel вместо скана изображения таблицы);
- **три звезды**: имеется в двухзвездочном виде плюс открытый формат (например, CSV вместо Excel);

- **четыре звезды**: имеется в виде всех предыдущих оценок, а также использует открытые стандарты от W3C (RDF и SPARQL) для идентификации вещей, с тем чтобы люди могли ссылаться на материал;
- **пять звезд**: имеется в виде всех предыдущих оценок, но также связывает ваши данные с данными других людей, обеспечивая контекст.

После достижения пяти звезд следующим шагом для набора данных должно стать предоставление дополнительных метаданных (данных о данных), а в случае правительственных данных – занесение в главный каталог данных, такой как https://www.data.gov/, https://data.gov.uk/ или http://data.europa.eu/euodp/en/data, соответственно для США, Великобритании и Европейского союза.

Совершенно очевидно, что связанные данные являются фундаментальным понятием Семантической паутины. Одним из известных примеров набора данных с реализацией связанных данных является краудсорсинговый проект DBpedia (http://dbpedia.org), направленный на извлечение структурированных знаний из информации, опубликованной в Википедии.

Среда описания ресурса RDF

Среда описания ресурса (Resource Description Framework, RDF), принадлежащая семейству спецификаций W3C, первоначально разрабатывалась как модель метаданных и использовалась для моделирования структурированных знаний (https://ru.wikipedia.org/wiki/Resource_Description_Framework).

Подобно классическим платформам концептуального моделирования, таким как модель сущностей и связей, которая популярна в проектировании баз данных, среда RDF опирается на идею объявления суждений о ресурсах. Такие суждения выражаются в виде *троек* (или *триплетов*), включающих три важные составляющие: субъект, предикат и объект.

Вот пример суждений, выраженных в виде троек «субъект–предикат–объект»:

```
(Рим, столица, Италия)
(Мадрид, столица, Испания)
(Питер, друг, Мэри)
(Мэри, друг, Питер)
(Мэри, живет_в, Рим)
(Питер, живет_в, Мадрид)
```

Синтаксис приведенных выше суждений произволен и не подчиняется какой-то конкретной спецификации RDF, но служит примером простоты этой модели данных. Подход к хранению суждений, основанный на тройках, позволяет представить любой вид знаний. Коллекцию суждений RDF можно естественным образом рассматривать как граф (https://ru.wikipedia.org/wiki/Теория_графов), и в особенности как маркированный ориентированный мультиграф; маркированный, потому что узлы (ресурсы) и ребра (предикаты) несут информацию, такую как названия ребер; ориентированный, потому что связи «субъект–предикат–объект» имеют четкое направление; а мультиграф, потому что узлы могут соединяться многочисленными параллельными ребрами, то есть два идентичных ресурса могут иметь между собой нескольких разных связей, причем каждая с разной семантикой.

Рисунок 9.1 позволяет наглядно представить, как можно выразить знание в виде графа.

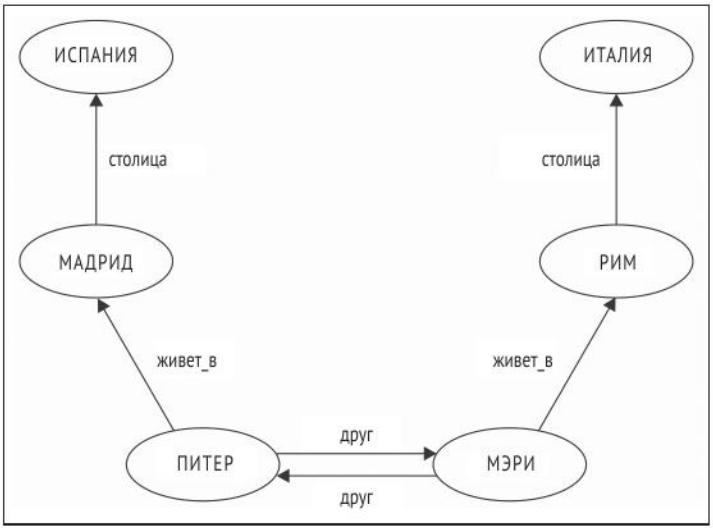

Рис. 9.1 ❖ Представление в виде графа знания на основе триплетов

Следующий уровень представления знаний проходит там, где в игру включается RDF, так как эта среда позволяет определять структуру дополнительно к триплетам. Предикат `rdf:type` позволяет объявлять типы определенных объектов. Вдобавок к этому схема RDF (RDF Schema, RDFS) (https://ru.wikipedia.org/wiki/RDFS) позволяет определять классы и связи между классами. Родственная технология OWL тоже дает возможность выражать связи между классами, но информация, которую она предоставляет о вашей модели данных, более богата с точки зрения ограничений и аннотаций.

RDF можно сериализовать (экспортировать) во многие другие форматы. Самым известным, пожалуй, является формат XML, но существуют и другие варианты, такие как N3 (https://ru.wikipedia.org/wiki/Нотация_3). Данная информация приводится с целью разъяснить, что RDF является прежде всего способом представления знаний при помощи триплетов, а не только форматом файлов.

Формат данных JSON-LD

Мы уже встречались с форматом JSON почти во всех предыдущих главах книги и оценили его гибкость. Связь между JSON и связанными данными материализована в легковесном формате данных JSON-LD (http://json-ld.org), опирающемся на формат JSON и, следовательно, легком для чтения и записи как для людей, так и для машин.

Во время связывания данных из разных источников данных можно столкнуться с проблемой неоднозначности. Рассмотрим следующие два документа JSON, представляющие человека:

```
/* Документ 1 */
{
  "имя": "Marco",
  "главная_страница": "http://marcobonzanini.com"
}
```

```
/* Документ 2 */
{
  "имя": "mb123",
  "главная_страница": "http://marcobonzanini.com"
}
```

Обратите внимание, что и там, и там схема одинаковая: оба документа имеют атрибуты `имя` и `главная_страница`. Неоднозначность проистекает из того, что эти два источника данных могут использовать одинаковые названия атрибутов с разными значениями. По сути, в первом документе `имя` используется для представления имени человека, а во втором – `имя` используется для представления имени пользователя системы.

Возникает ощущение, что эти два документа относятся к одной и той же сущности, потому что значение атрибута `главная_страница` одинаковое, однако без дополнительной информации мы не в состоянии разрешить эту неоднозначность.

Формат JSON-LD вводит простое понятие *контекста*. Используя контекст, мы в состоянии разрешиь неоднозначные ситуации как эту, задействуя метод, с которым уже знакомы, — URL-адрес.

Давайте рассмотрим следующий фрагмент, взятый из веб-сайта JSON-LD:

```
{
  "@context": "http://json-ld.org/contexts/person.jsonld",
  "@id": "http://dbpedia.org/resource/John_Lennon",
  "name": "John Lennon",
  "born": "1940-10-09",
  "spouse": "http://dbpedia.org/resource/Cynthia_Lennon"
}
```

Данный документ использует атрибут `@context` как ссылку на собственный список допустимых атрибутов и их значение. Тем самым исключается неоднозначность атрибутов в документе.

Приведенный пример также вводит еще один используемый в JSON-LD специальный атрибут – `@id`. Этот атрибут используется в качестве глобального идентификатора. Разные приложения могут использовать глобальный идентификатор, когда обращаются к одной и той же сущности. Так можно снять неоднозначность с данных из многочисленных источников и связать их между собой. Например, приложение может взять стандартную схему определения человека (http://json-ld.org/contexts/person.jsonld) и расширить ее дополнительными атрибутами.

Контекст (список и значение атрибутов) и способ глобальной идентификации сущностей лежат в основе связанных данных, и формат JSON-LD решает эту задачу простым и изящным способом.

Инициатива Schema.org

Данная инициатива возникла в 2011 в результате совместных усилий некоторых ведущих игроков в области поисковых систем (Google, Bing и Yahoo!, к которым позже присоединился Yandex) и представляет результат попытки создать коллективно используемый набор схем для данных со структурированной разметкой, которые предполагается использовать в веб-страницах.

Данная инициатива предлагает использовать словарь `schema.org` (http://schema.org/) наряду с микроформатами, форматами на основе RDF или JSON-LD для рас-

ширения информационного наполнения веб-сайтов метаданными о самом сайте. Используя предложенные схемы, кураторы могут отмечать свои веб-сайты дополнительным знанием, которое помогает поисковым системам и другим программам лучше понимать содержание.

Схемы об организациях и людях используются, чтобы оказывать влияние на такие системы, как Google Knowledge Graph, являющийся базой знаний, расширяющей поисковый механизм Google семантической информацией каждый раз, когда предметом поиска является конкретная сущность.

Анализ связей из DBpedia

Проект DBpedia – один из самых известных источников связанных данных. Опираясь на Википедию, он расширяет содержимое популярной энциклопедии, основанной на технологии совместной правки информационного материала wiki (от гавайск. *wiki* – быстрый), семантическими связями между сущностями. Доступ к структурированной информации из DBpedia можно получить через веб, используя SQL-подобный семантический язык запросов к RDF под названием SPARQL.

Несмотря на то что в Python имеется возможность выполнять запросы к базе данных при помощи SPARQL, предпочтительнее все же пользоваться библиотекой RDFLib.

Ее можно установить при помощи диспетчера пакетов `pip`:

```
$ pip install rdflib
```

Учитывая сложность темы Семантической паутины, рассмотрим пример, чтобы ощутить возможности DBpedia и одновременно получить общее представление о том, как использовать библиотеку RDFLib.

Сценарий `rdf_summarize_entity.py` выполняет поиск заданной сущности и пытается вывести ее краткое содержание:

```
# Chap09/rdf_summarize_entity.py
from argparse import ArgumentParser
import rdflib

def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--entity')
    return parser

if __name__ == '__main__':
    parser = get_parser()
    args = parser.parse_args()
    entity_url = 'http://dbpedia.org/resource/{}'.format(args.entity)

    g = rdflib.Graph()
    g.parse(entity_url)
    disambiguate_url = 'http://dbpedia.org/ontology/wikiPageDisambiguates'
    query = (rdflib.URIRef(entity_url),
             rdflib.URIRef(disambiguate_url),
             None)
    disambiguate = list(g.triples(query))
    if len(disambiguate) > 1:
```

```
        print("Ресурс {}:".format(entity_url))
        for subj, pred, obj in disambiguate:
            print('... может относиться к: {}'.format(obj))
    else:
        query = (rdflib.URIRef(entity_url),
                 rdflib.URIRef('http://dbpedia.org/ontology/abstract'),
                 None)
        abstract = list(g.triples(query))
        for subj, pred, obj in abstract:
            if obj.language == 'en': # для русского языка использовать 'ru'
                print(obj)
```

Как обычно, для чтения аргументов командной строки в данном сценарии используется объект `ArgumentParser`. Аргумент `--entity` служит для передачи имени сущности. Сценарий конструирует URL-адрес сущности в DBpedia согласно шаблону `dbpedia.org/resource/<имя-сущности>`. Этот URL-адрес первоначально используется для построения графа RDF с помощью класса `rdflib.Graph`.

Следующий шаг – разрешение любых неоднозначностей, или, другими словами, проверка принадлежности заданного имени нескольким сущностям. Это делается путем опроса триплетов в графе с использованием URL-адреса сущности в качестве субьекта, связи `wikiPageDisambiguates` в качестве предиката и пустого объекта. Чтобы собрать субъект, предикат и объект, необходимо задействоать класс `rdflib.URIRef`, который принимает URL-адреса сущности/связи в качестве единственного аргумента. Объект `None` как один из элементов триплета используется методом `triples()` в качестве заменителя любого возможного значения (по сути, это подстановочный символ).

При обнаружении неоднозначностей они выводятся на экран. В противном случае отыскивается сущность и, в данном случае, извлекается ее краткое содержание. Для той или иной сущности могут быть получены многочисленные краткие описания, потому что информационный материал может быть на разных языках. По мере обхода результатов сценарий выводит только английскую версию (с признаком `en`).

Например, с помощью данного сценария можно получить краткое описание языка Python:

```
$ python rdf_summarize_entity.py --entity "Python"
```

Использование всего одного слова `Python` для поиска в Википедии непосредственно не приводит к желаемому результату, потому что сам термин неоднозначен. Фактически может быть обнаружено несколько сущностей, соответствующих искомой строке, поэтому сценарий выводит список всех совпадений. Вот короткий фрагмент полученных результатов:

```
Ресурс http://dbpedia.org/resource/Python:
... может относиться к: http://dbpedia.org/resource/HMS_Python
... может относиться к: http://dbpedia.org/resource/Python_(programming_language)
... может относиться к: http://dbpedia.org/resource/Python_(mythology)
... может относиться к: http://dbpedia.org/resource/Python_(missile)
... может относиться к: http://dbpedia.org/resource/Boa_vs._Python
# (еще неоднозначности)
```

После идентификации требуемой сущности, например `Python_(programming_language)`, этот сценарий можно выполнить повторно с правильным именем сущности:

```
$ python rdf_summarize_entity.py --entity "Python_(programming_language)"
```

На этот раз результат будет в точности соответствовать нашим ожиданиям. Вот отрывок результата:

```
Python is a widely used general-purpose, high-level programming language.
Its design philosophy emphasizes code readability, (обрезано)
```

Этот пример показывает, что совмещение высокоуровневой концептуальной модели RDF, основанной на триплетах, с интерфейсом на Python является относительно простой задачей.

Анализ географических координат

Как упоминалось выше, микроформаты `geo` и `h-geo` служат для публикации географической информации. В книге о социальных медиа возникает резонный вопрос: относятся ли географические метаданные к описанию социальных данных. Думая о геоданных, следует учитывать центральную идею, которая состоит в том, что географическая информация может использоваться в целом ряде приложений. Например, пользователи могут пожелать искать предприятия с привязкой к конкретной местности (как в Yelp) или фотографии, сделанные в конкретном географическом месте. Вообще говоря, все мы физически находимся где-то или ищем нечто, физически находящееся в каком-то месте. Географические метаданные позволяют приложениям удовлетворить потребность в привязке к конкретной местности. В эру социальных и мобильных данных такого рода специализация открывает разработчикам приложений множество новых горизонтов.

А теперь вернемся к нашему исследованию данных с семантической разметкой и посмотрим, как извлекать географические метаданные из веб-страниц, используя для наших примеров Википедию. Вспомним, как выглядит разметка классического микроформата `geo`:

```
<p class="geo">
  <span class="latitude">51.50722</span>,
  <span class="longitude">-0.12750</span>
</p>
```

Более новый формат `h-geo` очень похож на него, кроме некоторых изменений в именах классов. В шаблонах Википедии используется немного отличающийся формат, который выглядит так:

```
<span class="geo">51.50722; -0.12750</span>
```

Чтобы извлечь эту информацию из страницы Википедии, требуется выполнить два шага – извлечь саму страницу и исследовать разметку HTML в поисках нужных имен классов. Следующий раздел подробно описывает эту процедуру.

Извлечение геоданных из Википедии

Процедура извлечения геоданных из веб-страницы легко решается, например, с привлечением библиотек, которые мы уже обсуждали, таких как **requests**

и **BeautifulSoup**. Несмотря на всю простоту, специально для решения таких задач был создан отдельный пакет **mf2py**, предлагающий простой интерфейс для быстрого анализа веб-ресурса по заданному URL-адресу. Данная библиотека также реализует несколько вспомогательных функций для преобразования полученных данных из словаря Python в строку в формате JSON.

Библиотеку `mf2py` можно установить, как обычно, используя pip:

```
$ pip install mf2py
```

Данная библиотека предлагает три подхода к разбору данных с семантической разметкой. Мы можем передать в функцию `parse()` фрагмент страницы в виде строки, указатель на файл или URL-адрес. Следующий пример показывает, как выполнить анализ строки с семантической разметкой:

```
>>> import mf2py
>>> content = '<span class="geo">51.50722; -0.12750</span>'
>>> obj = mf2py.parse(doc=content)
```

Переменная `obj` сейчас содержит словарь с результатом анализа. Его содержимое можно обследовать, просто преобразовав в формат JSON:

```
>>> import json
>>> print(json.dumps(obj, indent=2))
{
  "items": [
  {
    "type": [
      "h-geo"
    ],
    "properties": {
      "name": [
        "51.50722; -0.12750"
      ]
    }
  }
  ],
  "rels": {},
  "rel-urls": {}
}
```

Анализ содержимого файла выглядит очень похоже:

```
>>> with open('some_content.xml') as fp:
... obj = mf2py.parse(doc=fp)
```

Но если предполагается выполнить анализ содержимого по URL, этот адрес следует передавать в аргументе `url` вместо аргумента `doc`:

```
>>> obj = mf2py.parse(url='http://example.com/your-url')
```

Механизм парсинга в библиотеке mf2py опирается на библиотеку BeautifulSoup. Вызывая BeautifulSoup, не задав явно синтаксического анализатора, библиотека попытается угадать наилучший вариант из имеющихся. Например, если установить библиотеку **lxml** (в главе 6 «*Вопросы и ответы в сети Stack Exchange*»), то она будет выбрана по умолчанию. Однако вызов функции `parse()` все равно инициирует предупреждение:

```
UserWarning: No parser was explicitly specified, so I'm using the best
available HTML parser for this system ("lxml"). This usually isn't a
problem, but if you run this code on another system, or in a different
virtual environment, it may use a different parser and behave differently.
To get rid of this warning, change this:

  BeautifulSoup([your markup])

to this:

  BeautifulSoup([your markup], "lxml")
```

Это сообщение породила библиотека BeautifulSoup, но мы не используем ее непосредственно, поэтому предложение о том, как решить эту проблему, может ввести в заблуждение. Чтобы предотвратить вывод данного предупреждения, нужно явно передать требуемый парсер в вызов функции `parse()`:

```
>>> obj = mf2py.parse(doc=content, html_parser='lxml')
```

После знакомства с библиотекой mf2py объединим вновь полученные знания и попробуем извлечь географическую информацию из страниц Википедии.

Сценарий `micro_geo_wiki.py` принимает URL-адрес страницы в Википедии и выводит список географических мест их координатами, если в тексте найдена любая георазметка:

```
# Chap09/micro_geo_wiki.py
from argparse import ArgumentParser
import mf2py

def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--url')
    return parser

def get_geo(doc):
    coords = []
    for d in doc['items']:
        try:
            data = {
                'name': d['properties']['name'][0],
                'geo': d['properties']['geo'][0]['value']
            }
            coords.append(data)
        except (IndexError, KeyError):
            pass
    return coords

if __name__ == '__main__':
    parser = get_parser()
    args = parser.parse_args()

    doc = mf2py.parse(url=args.url, html_parser='lxml')
    coords = get_geo(doc)
    for item in coords:
        print(item)
```

Для чтения аргументов командной строки данный сценарий использует объект `ArgumentParser`. Аргумент `--url` служит для передачи URL-адреса страницы, анализ которой требуется выполнить.

Затем сценарий вызывает функцию `mf2py.parse()`, передает ей этот URL-адрес в аргументе и получает словарь с микроформатной информацией.

Основная логика извлечения географических координат находится в функции `get_geo()`, которая принимает словарь с полученными данными и возвращает список словарей. Каждый словарь имеет два ключа: `name` (географическое название) и `geo` (координаты).

Посмотрим, как действует сценарий, выполнив следующую команду:

```
$ python micro_geo_wiki.py \
  --url "https://en.wikipedia.org/wiki/London"
```

Эта команда вернет следующий результат:

```
{'name': 'London', 'geo': '51.50722; -0.12750'}
```

Неудивительно, что на странице Википедии о Лондоне мы найдем единственные координаты Лондонского сити.

Википедия также предлагает много страниц со списками географических мест, как показано ниже:

```
$ python micro_geo_wiki.py --url \
"https://en.wikipedia.org/wiki/List_of_United_States_cities_by_population"
```

Эта команда выведет длинный список (более 300 пунктов) с географической информацией о нескольких американских городах. Вот короткий фрагмент этого списка:

```
{'geo': '40.6643; -73.9385', 'name': '1 New York City'}
{'geo': '34.0194; -118.4108', 'name': '2 Los Angeles'}
{'geo': '41.8376; -87.6818', 'name': '3 Chicago'}
# (обрезано)
```

Приведенный пример показывает, что извлечение геоданных из веб-страниц не представляет особой трудности. Простота библиотеки mf2py позволяет решить эту задачу всего несколькими строками кода.

Следующий раздел делает еще шаг вперед и отвечает на вопрос, что можно сделать с полученной географической информацией. Ответ напрашивается сам собой: создать географическую карту. Поэтому для визуализации географических данных мы воспользуемся веб-службой Google Maps.

Нанесение геоданных на карты Google Maps

Веб-служба Google Maps (https://www.google.com/maps) настолько популярна, что, вероятно, не нуждается в особом представлении. Среди всего прочего она предлагает возможность создания собственной карты со списком интересных мест.

Один из способов автоматизировать создание географических карт и гарантировать функциональную совместимость между разными приложениями – использовать распространенный формат представления координат. В этом разделе мы обсудим использование языка разметки **KML** (Keyhole Markup Language) (https://ru.wikipedia.org/wiki/KML) как способа представления данных в формате, понятном

службе Google Maps. Формат KML, основанный на формате XML, служит для выражения географических данных на двух- и трехмерных картах. Первоначально разработанный компанией Keyhole для еще одного продукта Google – Google Earth, теперь он используется для самых разнообразных приложений.

Следующий фрагмент содержит образец документа KML, представляющего географическое положение Лондона:

```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2">
  <Document>
    <Placemark>
      <name>London</name>
      <description>London</description>
      <Point>
        <coordinates>-0.12750,51.50722</coordinates>
      </Point>
    </Placemark>
  </Document>
</kml>
```

Обратите внимание на значение тега `coordinates`. Геоданные, извлекаемые из Википедии, имеют формат `latitude;longitude` (широта; долгота – через точку с запятой), в разметке KML используется формат `longitude,latitude,altitude` (долгота, широта, высота – через запятую), причем высота может отсутствовать и принимается равной нулю, если она опущена, как показано в примере выше.

Чтобы отобразить список географических координат на карте Google Maps, мы должны сгенерировать этот список в формате KML. В Python имеется библиотека под названием **PyKML**, которая упрощает данный процесс.

Ее можно установить как обычно, при помощи `pip`:

```
$ pip install pykml
```

Следующий сценарий `micro_geo2kml.py` опирается на геоданные, прежде извлеченные из Википедии, и создает файл в формате KML.

Сценарий пытается использовать библиотеку lxml для преобразования XML в строку. Если данная библиотека отсутствует, будет использован модуль `ElementTree` из стандартной библиотеки Python. За дополнительной информацией о библиотеке lxml обращайтесь к главе 6 «*Вопросы и ответы в сети Stack Exchange*».

```
# Chap09/micro_geo2kml.py
from argparse import ArgumentParser
import mf2py
from pykml.factory import KML_ElementMaker as KML
try:
    from lxml import etree
except ImportError:
    import xml.etree.ElementTree as etree

def get_parser():
    parser = ArgumentParser()
    parser.add_argument('--url')
```

```
    parser.add_argument('--output')
    parser.add_argument('--n', default=20)
    return parser

def get_geo(doc):
    coords = []
    for d in doc['items']:
        try:
            data = {
                'name': d['properties']['name'][0],
                'geo': d['properties']['geo'][0]['value']
            }
            coords.append(data)
        except (IndexError, KeyError):
            pass
    return coords

if __name__ == '__main__':
    parser = get_parser()
    args = parser.parse_args()

    doc = mf2py.parse(url=args.url, html_parser='lxml')
    coords = get_geo(doc)
    folder = KML.Folder()
    for item in coords[:int(args.n)]:
        lat, lon = item['geo'].split('; ')
        place_coords = ','.join([lon, lat])
        place = KML.Placemark(
            KML.name(item['name']),
            KML.Point(KML.coordinates(place_coords))
        )
        folder.append(place)

    with open(args.output, 'w') as fout:
        xml = etree.tostring(folder, pretty_print=True).decode('utf8')
        fout.write(xml)
```

Для чтения аргументов командной строки данный сценарий использует объект `ArgumentParser`. В аргументе `--url` передается URL страницы для анализа, в аргументе `--output` – имя файла KML для сохранения географической информации, и в необязательном аргументе `--n` можно передать максимальное количество координат, извлекаемых из страницы (по умолчанию `20`).

В главном блоке, как и раньше, вызовом функции `get_geo()` выполняются парсинг веб-страницы и извлечение географических координат. Эта функция возвращает список словарей, каждый из которых имеет ключи `name` и `geo`. В терминологии KML одиночное географическое место называется меткой *placemark*, а список мест называется папкой *folder*. Другими словами, сценарий должен преобразовать список словарей в папку из меток.

Для этого сценарий создает переменную `folder` и инициализирует ее пустым объектом `KML.Folder`. По мере обхода списка словарей с географической информацией для каждого элемента создается объект `KML.Placemark`. Метка состоит из имени `KML.name` и точки `KML.Point` – объекта с географическими координатами.

Чтобы преобразовать имеющиеся координаты в формат KML, необходимо разбить исходную строку координат в формате `latitude;longitude`, затем поменять эти два значения местами и подставить запятую вместо точки с запятой, чтобы в итоге получить строку в формате `longitude,latitude`.

После создания списка меток его нужно выгрузить в файл в формате XML. Это делается вызовом метода `tostring()`, который возвращает объект `bytes` и, следовательно, должен декодироваться перед записью в файл.

Данный сценарий можно выполнить со следующим примером:

```
$ python micro_geo2kml.py \
  --url "https://en.wikipedia.org/wiki/
    List_of_United_States_cities_by_population" \
  --output us_cities.kml \
  --n 20
```

Теперь у нас есть файл `us_cities.kml` с географическими данными, который можно передать службе Google Maps, чтобы сгенерировать карту.

Сценарий `micro_geo2kml.py` может возбудить исключение с сообщением:

```
Traceback (most recent call last):
  File "micro_geo2kml.py", line 4, in <module>
    from pykml.factory import KML_ElementMaker as KML
  File "C:\Python36\lib\site-packages\pykml\factory.py", line 216
    print write_python_script_for_kml_document(doc)
                                               ^
SyntaxError: invalid syntax
```

Данная ошибка связана с тем, что в файле `factory.py` в библиотеке `pykml` присутствует оператор вывода `print` без круглых скобок, то есть в формате Python 2. Чтобы сценарий, где используется эта библиотека, заработал в Python 3, нужно в файле `factory.py`, расположенном в папке `\Lib\site-packages\pykml`, исправить строку

```
print write_python_script_for_kml_document(doc)
```

на

```
print(write_python_script_for_kml_document(doc))
```

и сохранить файл. После это можно запускать сценарий.

Кроме того, для правильного отображения кириллицы на карте следует добавить пару строк в созданный файл `us_cities.kml`. Этот файл должен начинаться с XML-объявления кодировки UTF8: `<?xml version="1.0" encoding="UTF-8"?>`, а после элемента `Folder`, перед первым элементом `Placemark`, можно вставить элемент `name`:

```
<name>Города США по численности населения</name>
```

В меню Google Maps выберите элемент Your places (Мои места) со списком ваших географических мест и затем MAPS (КАРТЫ) со списком пользовательских карт, как показано на рис. 9.2.

Щелкнув на ссылке CREATE MAP (СОЗДАТЬ НОВУЮ КАРТУ), вы получите возможность импортировать данные из файла, который может быть электронной таблицей, файлом CSV или KML. Если выбрать файл `us_cities.kml`, созданный нашим сценарием, на экране появится результат, изображенный на рис. 9.3.

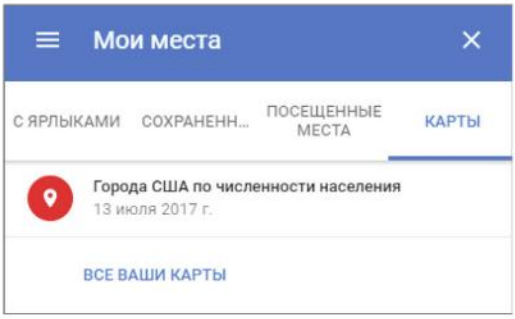

Рис. 9.2 ❖ Список пользовательских карт в веб-службе Google Maps

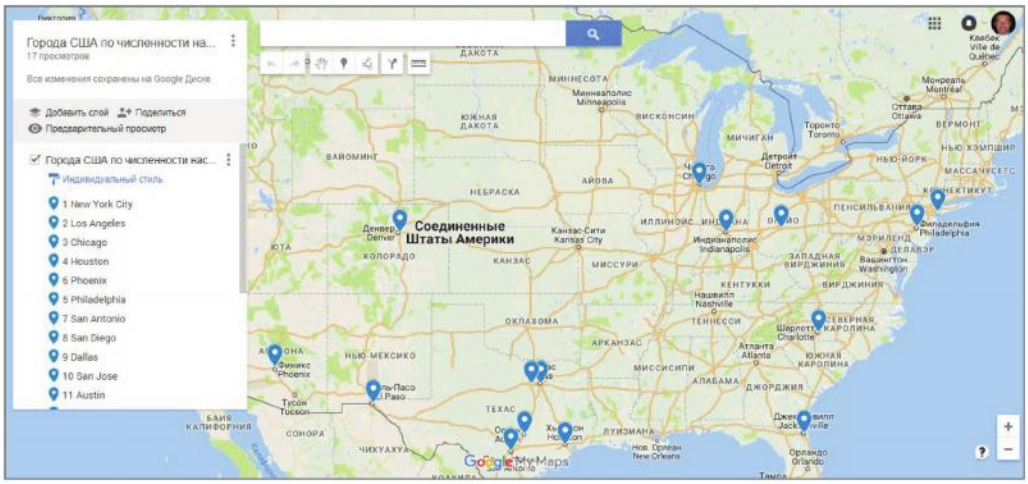

Рис. 9.3 ❖ Список крупных городов США в Google Maps

Каждой метке соответствует значок на карте. Щелчок на любом из значков приводит к выводу соответствующего описания, в данном случае названия и координат.

Резюме

В начале настоящей главы мы сослались на точку зрения создателя Всемирной паутины Тима Бернерса-Ли. Взгляд на Всемирную паутину как социальное, а не техническое, творение размывает границу между социальными и семантическими данными.

Мы также описали более широкий контекст Семантической паутины и то, как разные технологии, предлагавшиеся за прошедшие годы, согласуются между собой и дают возможность конкретизировать это видение. Несмотря на то что вымысел и надежда в отношении Семантической паутины за прошедшие 15–20 лет оставались довольно высокими, видение Тима Бернерса-Ли еще не стало повсеместным.

Сообщества, направляемые консорциумом W3C, правительствами и частными компаниями, такими как `schema.org`, пытаются совершить мощный рывок в этом направлении. В то время как скептики имеют предостаточно материала для обсуждения, почему обещания Семантической паутины еще не полностью сдержаны, мы предпочитаем сосредоточиться на возможностях, которые имеем на данный момент.

Современные тенденции социально-медийной экосистемы показывают, что социальные и семантические данные соединены глубокими связями. Настоящая книга обратилась к теме анализа данных нескольких социально-медийных платформ, используя возможности, предлагаемые их прикладными программными интерфейсами (API), и применяя некоторые самые популярные инструменты на Python для предварительного и основного анализа данных. Хотя одна книга может лишь прикоснуться к излагаемой в ней теме, мы надеемся, что сейчас, прочтя эту книгу, вы по достоинству оценили все возможности, которые имеются в контексте социальных медиа для практиков и исследователей в области анализа данных.

Приложение А

Анализ данных из социальной сети «ВКонтакте»

Данные социальных сетей являются настоящей золотой жилой для проведения анализа в исследовательских и деловых целях. На примере API «ВКонтакте» и языка Python разберем практический пример, который в дополнение к основному материалу книги поможет узнать, как работать с API «ВКонтакте» на языке Python, используя стандартные библиотеки, такие как requests, networkx, numpy и pandas, и как получить список друзей и членов групп.

Стоит отметить, что здесь намеренно не использованы две широко известные библиотеки для Python, которые упрощают работу с API «ВКонтакте». Для полноты картины приведу примеры их использования:

```
>>> import vk
>>> session = vk.Session()
>>> api = vk.API(session)
>>> api.users.get(user_ids=1)
[{'first_name': 'Pavel', 'last_name': 'Durov', 'id': 1}]
```

В настоящее время эта библиотека достаточно активно развивается. Также стоит упомянуть другую библиотеку для Python. Вот как выглядит выгрузка фотографии на сервер с ее помощью:

```
import vk_api

def main():
    """ Пример: выгрузка фото """
    login, password = 'python@vk.com', 'mypassword'
    vk_session = vk_api.VkApi(login, password)

    try:
        vk_session.authorization()
    except vk_api.AuthorizationError as error_msg:
        print(error_msg)
        return

    """
        В VkUpload реализованы методы выгрузки файлов в ВК.
```

```
    """
    upload = vk_api.VkUpload(vk_session)

    photo = upload.photo(  # Подставьте свои данные
        'D:/downloads/tube.jpg',
        album_id=200851098,
        group_id=74030368
    )

    vk_photo_url = 'https://vk.com/photo{}_{}'.format(
        photo[0]['owner_id'], photo[0]['id']
    )

    print(photo, '\nLink: ', vk_photo_url)

if __name__ == '__main__':
main()
```

Вместо этих библиотек мы будем обращаться к конечным точкам API «ВКонтакте» напрямую.

Анализ сообщества и определение его типичного участника

Анализ открытых данных участников сообщества позволяет узнать об аудитории сообщества, определить типичного участника группы и косвенно узнать об инициаторе общения, следуя известной поговорке «*Скажи мне, кто твой друг, и я скажу, кто ты*». Чтобы решить эту задачу, используем методы API «ВКонтакте» напрямую при помощи библиотеки requests.

Начнем с построения графа друзей конкретного пользователя.

```
import time
import datetime
import dateutil.parser as parser
from collections import Counter

import requests
import networkx as nx
import numpy as np
import pandas as pd

def query_vkontakte(items, url, islist=True):
    '''Обобщенная функция
    Выбрать данные из соцсети «ВКонтакте»

    Также можно добавить в запрос access_token, получив его через OAuth 2.0
    '''
    json_response = requests.get(url.format(items)).json()
    if islist==False:  # вернуть в формате множества
        if json_response.get('error'): return set()
        return set(json_response['response'])
    else:              # по умолчанию вернуть список
        if json_response.get('error'): return list()
        return json_response['response']

def get_friends(user_id):
    '''Собрать идентификаторы друзей
```

```
    для указанного пользователя
    '''
    url = 'https://api.vk.com/method/friends.get?user_id={}'
    return query_vkontakte(user_id, url, islist=False)

def make_adjacency(user_id):
    '''Создать таблицу инцидентности
    в виде словаря в формате (узел, список узлов)

    user_id: id пользователя, для которого требуется
             построить граф друзей

    Возвращает: словарь и список идентификаторов друзей
    '''
    dic = {}

    friend_ids = get_friends(user_id)
    for friend in friend_ids:
        ids = get_friends(friend)  # идентификаторы друзей друга
        dic[friend] = {i for i in ids if i in friend_ids}

    return dic, friend_ids

def transform_to_graph(dic, f_ids):
    '''Преобразовать словарь инцидентности
    в ненаправленный граф библиотеки networkx

    Возвращает: граф библиотеки networkx
    '''
    g = nx.Graph(directed=False)
    for i in dic:
        g.add_node(i)
        for j in dic[i]:
            if i != j and i in f_ids and j in f_ids:
                g.add_edge(i, j)
    return g

user_id = 168597426   # анализируемый пользователь
dic, friend_ids = make_adjacency(user_id)
G = transform_to_graph(dic, friend_ids)

pos = nx.nx_pydot.graphviz_layout(G, prog="neato")
nx.draw(G, pos, node_size=30, with_labels=False, width=0.2)

#plt.savefig('graph.png', dpi=600)
plt.show()
```

В приведенном выше примере используется метод API «ВКонтакте» `friends.get` с параметром `user_id`, который получает список идентификаторов друзей по идентификатору пользователя. Результатом работы сценария будет граф, показанный на рис. А.1.

В данном случае можно выделить две большие компоненты связанности: участников сообщества из двух разных городов проживания, которые связаны между собой одним узлом инициатора общения. Это так называемый эгоцентричный граф сообщества, который отображает связи инициатора с другими участниками.

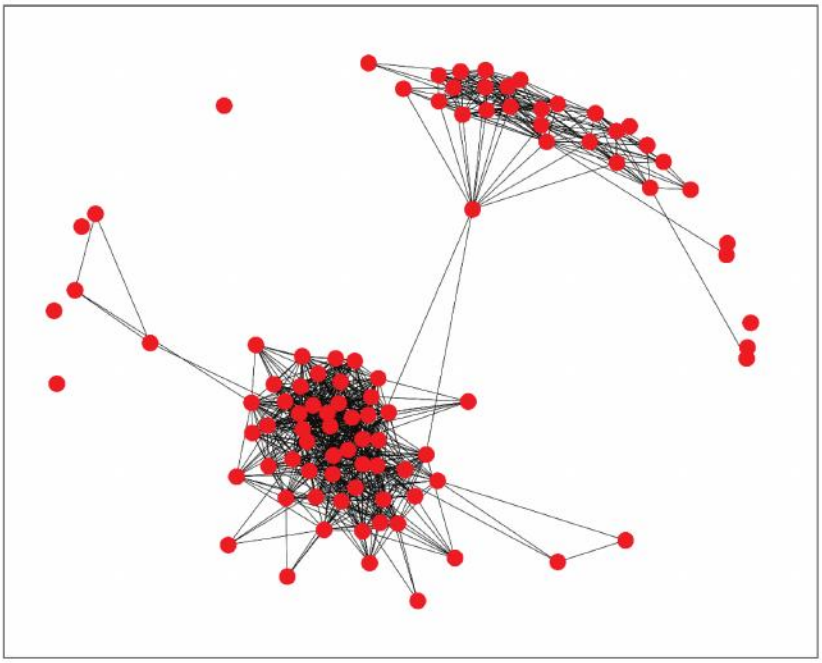

Рис. А.1 ❖ Эгоцентричный граф друзей

Посмотрим на свойства полученного социального графа, воспользовавшись небольшой вспомогательной функцией, которая на лету переводит английские термины:

```
trans = {'Name':'Имя', 'Type':'Тип', 'Number of nodes':'Число узлов',
         'Number of edges':'Число дуг', 'Average degree':'Средняя степень'}
info = nx.info(G)
for k,v in trans.items():
    if k in info: info = info.replace(k,v)
print(info)
# Имя:
# Тип: Graph
# Число узлов: 95
# Число дуг: 832
# Средняя степень:  17.5158
```

Средняя степень, то есть среднее количесто инцидентных узлу ребер, в размере 17.5158 говорит о стабильности социального графа. На данный момент узлы графа содержат только идентификаторы. Теперь заполним узлы информацией из социальной сети, взяв оттуда четыре базовых признака: пол, дату рождения, семейное положение и место проживания. Данная информация содержится соответственно в полях `sex`, `bdate`, `relation` и `city` (подробно об используемых полях можно узнать на https://vk.com/dev/objects/user).

```
def get_user_info(user_id):
    '''Выбрать для пользователя его данные
    из полей sex, bdate, city, relation
```

```
    Возвращает: словарь
    '''
    url = 'https://api.vk.com/method/users.get?user_id={}&fields=sex,bdate,city,relation'
    return query_vkontakte(user_id, url)

# списки, куда будут записываться результаты расчетов
res = {'sex':[], 'rel':[], 'dat':[], 'loc':[]}
data = dict()

curr_year = datetime.datetime.now().year
y = str(curr_year % 200)  # 2017 -> 17

# обработать данные каждого участника,
# собрав их в 4 списка
for userid in friend_ids:
    data[userid] = get_user_info(userid)
    q = data[userid][0]

    res['sex'].append(int(q.get('sex', 0)))
    res['rel'].append(int(q.get('relation', 0)))
    res['dat'].append(curr_year - parser.parse(q.get('bdate', '1.'+y)).year)
    res['loc'].append(int(q.get('city', 0)))
```

Сейчас мы использовали метод `users.get` с параметром `user_id` и списком интересующих полей и в результате получили четыре списка со значениями признаков по каждому участнику. Например, для признака «пол» список будет иметь следующий вид:

```
'sex': [2, 2, 2, 1, 1, 1, 1, ...]
```

Теперь агрегируем промежуточные результаты в специальный словарь `Counter`. Например, преобразованный список с информацией о половой принадлежности теперь будет иметь следующие значения:

```
Counter({2: 56, 1: 39})
```

В API «ВКонтакте» поле `sex` может принимать три значения: 0 – не указано, 1 – женский и 2 – мужской, то есть в рассматриваемой группе имеются 56 участников мужского пола.

И в заключение покажем итоговые результаты на графиках (рис. А.2). Отметим, что в двух случаях мы отфильтровали полученные данные, чтобы убрать выбросы, то есть ошибочные или нерепрезентативные данные.

```
def changeKeys(d, a):
    '''Поменять целочисленные ключи на строковые

    Возвращает: словарь
    '''
    for i, v in enumerate(a):
        if d[i]: d[v] = d.pop(i)
    return d

def get_cities(d):
    '''Выбрать для списка идентификаторов городов
    их полные названия

    Возвращает: словарь 'строка=количество'
```

```
    '''
    # собрать коды городов в строку для запроса
    s = ','.join(map(str, list(d.keys())))
    # выполнить запрос
    url = 'https://api.vk.com/method/database.getCitiesById?country_id=1&city_ids={}'
    locations = query_vkontakte(s, url)  # список {cid=целое, name=строка}

    # преобразовать список {cid=целое, name=строка} в словарь 'целое=строка'
    locs = dict()
    for loc in locations:
        locs.update({loc['cid']:loc['name']})

    # преобразовать словарь 'целое=строка' в 'строка=количество'
    for k,v in locs.items():
        if d[k]: d[v] = d.pop(k)
    return d

d1 = changeKeys(d1, ['не указано', 'жен', 'муж'])

d2 = changeKeys(d2, ['не указано',
                     'не женат/не замужем',
                     'есть друг/есть подруга',
                     'помолвлен/помолвлена',
                     'женат/замужем',
                     'все сложно',
                     'в активном поиске',
                     'влюблен/влюблена',
                     'в гражданском браке'])

d3 = changeKeys(dict((k,v)   # фильтрация выбросов
                      for (k, v) in d3.items()
                      if k < 100),
                ['не указано'])

d4 = changeKeys(dict((k,v)   # фильтрация выбросов
                      for (k, v) in get_cities(d4).items()
                      if v > 1),
                ['не указано'])

df = pd.DataFrame.from_dict({'Пол':d1,
                             'Семейное положение':d2,
                             'Возраст':d3,
                             'Место проживания':d4})

df.plot.pie(subplots=True, legend=True, layout=(2, 2),
            figsize=(8, 8), startangle=-45)
plt.tight_layout()

#plt.savefig('four_plots.png', dpi=600)
plt.show()
```

В результате анализа по четырем признакам (пол, семейное положение, возраст и место проживания) можно сделать следующий предварительный вывод: мы имеем дело с молодежным сообществом, инициатором общения в котором является молодой человек в возрасте примерно 20 лет, проживающий в Санкт-Петербурге. Наличие двух компонент связности в данном возрасте говорит о рас-

ширении им своего круга общения в связи с окончанием школы и поступлением в учебное заведение, но не на работу, что подтверждается географическим положением двух главных городов и семейным положением, то есть эти две компоненты связности можно условно назвать школьной и студенческой, и, скорее всего, молодой человек учится в Санкт-Петербурге и прибыл из Хабаровска.

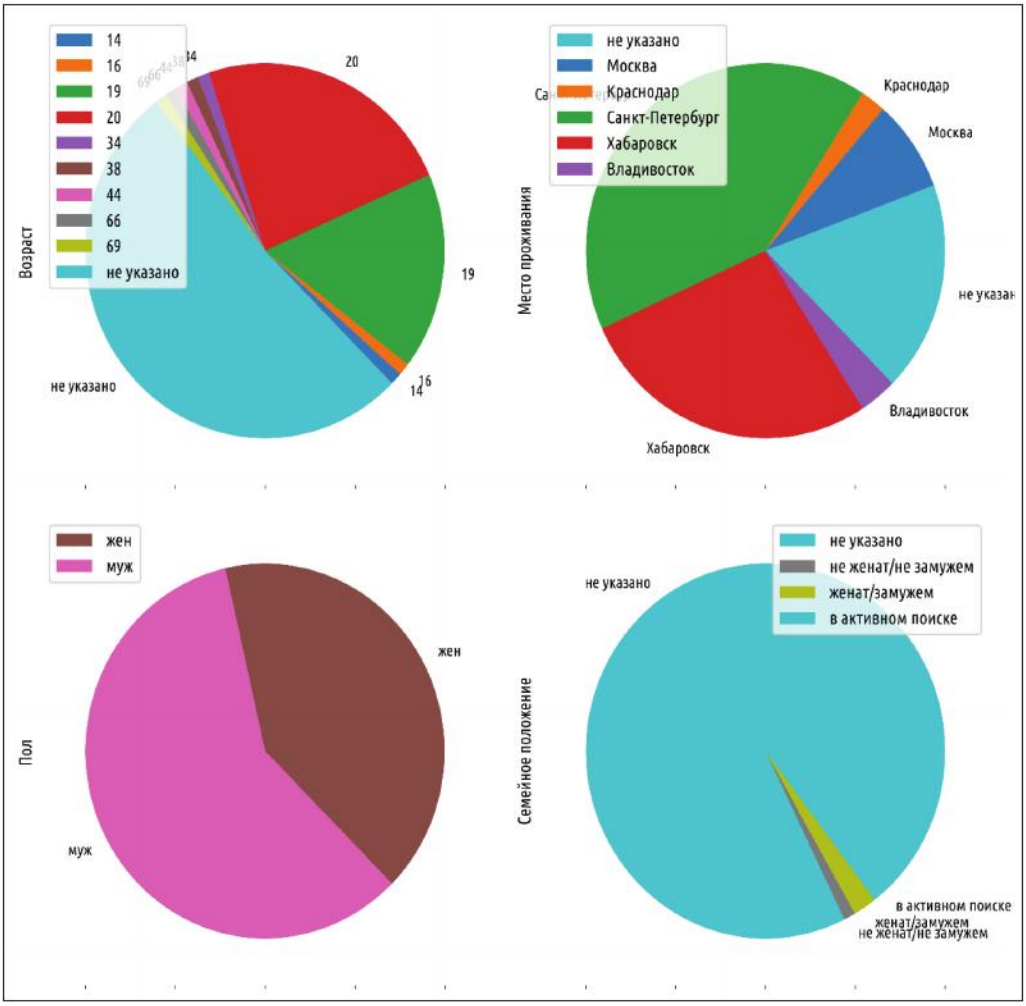

Рис. А.2 ❖ Характеристика сообщества

Разумеется, тут приводится лишь базовый анализ, который можно расширить за счет таких полей, как `personal`, `interests` и других, и привлечения методов обработки естественного языка, которые хорошо описаны в настоящей книге, а также добавить перечень характерных сообществ, в которых пользователь участвует.

Задача определения типичного представителя сообщества часто применяется коммерческими предприятиями с целью получения рекомендаций по привлечению новых участников и корректировки маркетинговых и сбытовых кампаний.

Далее определим центральные узлы полученного социального графа и покажем их на графиках с использованием библиотеки networkx, вычислим коэффициент кластеризации и применим алгоритм вычисления рейтинга авторитетности узлов.

Определение центральных узлов социального графа

В социодинамике при расчете индексов отдельных узлов графа, как правило, используются 7 следующих показателей:

- входная степень узла — количество ребер графа, входящих в узел;
- выходная степень узла — количество ребер графа, выходящих из узла;
- расстояние от данного узла до каждого из остальных;
- среднее расстояние от данного узла до остальных;
- эксцентричность — наибольшее из географических расстояний от данного узла до остальных;
- промежуточность — число путей, включающих данный узел;
- центральность (центральным агентом сети является тот, у кого больше всего связей).

В этом разделе мы рассмотрим показатель центральности. Центральность — это степень «важности» или «влияния» определенного пользователя (кластера пользователей). Самыми распространенными мерами центральности являются центральность по степени узлов, по посредничеству (промежуточности), по близости и по собственному вектору.

Чтобы рассчитать центральные узлы социального графа, сначала извлечем самую большую компоненту связности в графе и затем вычислим меры центральности узлов.

```
# Взять компоненту связности с максимальным размером
graph = max(nx.connected_component_subgraphs(G), key=len)
```

Теперь вычислим простейшую меру – *центральность по степени узлов*. Она означает, что тот, кто обладает большим количеством связей с другими, занимает центральное положение в локальной общности. Иными словами, центральным можно считать того, кто связан с наибольшим числом членов группы.

```
# Центральность по степени
deg_cen = nx.degree_centrality(graph)
```

Затем рассчитаем меру *центральности по посредничеству*, то есть количество кратчайших путей, проходящих через узел, или, выражаясь иначе, число раз, которое требуется любому узлу пройти через заданный узел, чтобы достичь любого другого узела по кратчайшему пути:

```
# Центральность по посредничеству
bet_cen = nx.betweenness_centrality(graph)
```

Далее получим меру *центральности по близости*, которая определяет, насколько данный узел близок ко всем остальным узлам в сети. Иными словами, эта мера говорит о скорости распространения информации в различные узлы.

```
# Центральность по близости
clo_cen = nx.closeness_centrality(graph)
```

И в заключение рассчитаем меру *центральности по собственному вектору*, или меру влияния узла на всю сеть, т. е. узел с незначительной информацией, но связанный большим числом связей с такими же узлами, будет считаться в сети влиятельным.

```
# Центральность по собственному вектору
eig_cen = nx.eigenvector_centrality(graph)
```

И в конце этого краткого обзора представим вспомогательную функцию, которая при заданном словаре и пороге *K* возвращает верхние *K* ключей в соответствии со значениями элементов. Затем применим данную функцию к различным метрикам центральности. Следующий пример извлекает 5 центральных узлов для каждого случая.

```
def get_top_keys(dic, top):
    items = sorted(dic.items(), key=lambda x: x[1], reverse=True)
    return map(lambda x: x[0], items[:top])

top_deg_cen = get_top_keys(deg_cen, 5)
top_bet_cen = get_top_keys(bet_cen, 5)
top_clo_cen = get_top_keys(clo_cen, 5)
top_eig_cen = get_top_keys(eig_cen, 5)

print(list(top_deg_cen), list(top_bet_cen),
      list(top_clo_cen), list(top_eig_cen))
```

Отображение центральностей на графике

Теперь применим меры центральности к нашему графу сообщества. Сначала представим еще одну вспомогательную функцию, которая при заданном графе, функции центральности и множителе отсечения узлов возвращает самые важные узлы в сообществе.

```
def most_important(g, fn, mult=1):
    """Вычислить рейтинг узлов

    Возвращает: копию g с самыми важными узлами
                согласно рейтингу узла
    """
    ranking = fn(g).items()
    r = [x[1] for x in ranking]
    m = sum(r)/len(r) # средняя центральность
    t = m*mult # порог, оставить только те узлы,
               # чья центральность больше средней в mult раз
    gcopy = g.copy()
    for k, v in ranking:
        if v < t:  # подрежем
            gcopy.remove_node(k)
    return gcopy
```

Далее рассчитаем самые важные узлы графа сообщества и построим графики (рис. A.3), отметив на базовом графе самые важные узлы:

```
Gb = most_important(G, nx.betweenness_centrality)
Gd = most_important(G, nx.degree_centrality)
Gc = most_important(G, nx.closeness_centrality)
```

```
Ge = most_important(G, nx.eigenvector_centrality)

def draw_graph(g, g2, title):
    # создать оформление
    pos = nx.spring_layout(g)
    #pos = nx.nx_pydot.graphviz_layout(G, prog="neato")

    # вывести все узлы и дуги
    nx.draw_networkx_nodes(g, pos, node_color='r', alpha=0.7, node_size=12)
    nx.draw_networkx_edges(g, pos, alpha=0.1)

    # вывести самые важные узлы в другом стиле
    nx.draw_networkx_nodes(g2, pos, node_color='b', alpha=0.3, node_size=254)

    # то же самое и с метками
    #nx.draw_networkx_labels(Gt, pos) #, font_size=12, font_color='b')
    plt.axis('off')

    #plt.savefig(title + '.png', dpi=600)
    plt.show()

draw_graph(G, Gb, 'Центральность по посредничеству')
draw_graph(G, Gd, 'Центральность по степени узлов')
draw_graph(G, Gc, 'Центральность по близости узлов')
draw_graph(G, Ge, 'Центральность по собственному вектору')
```

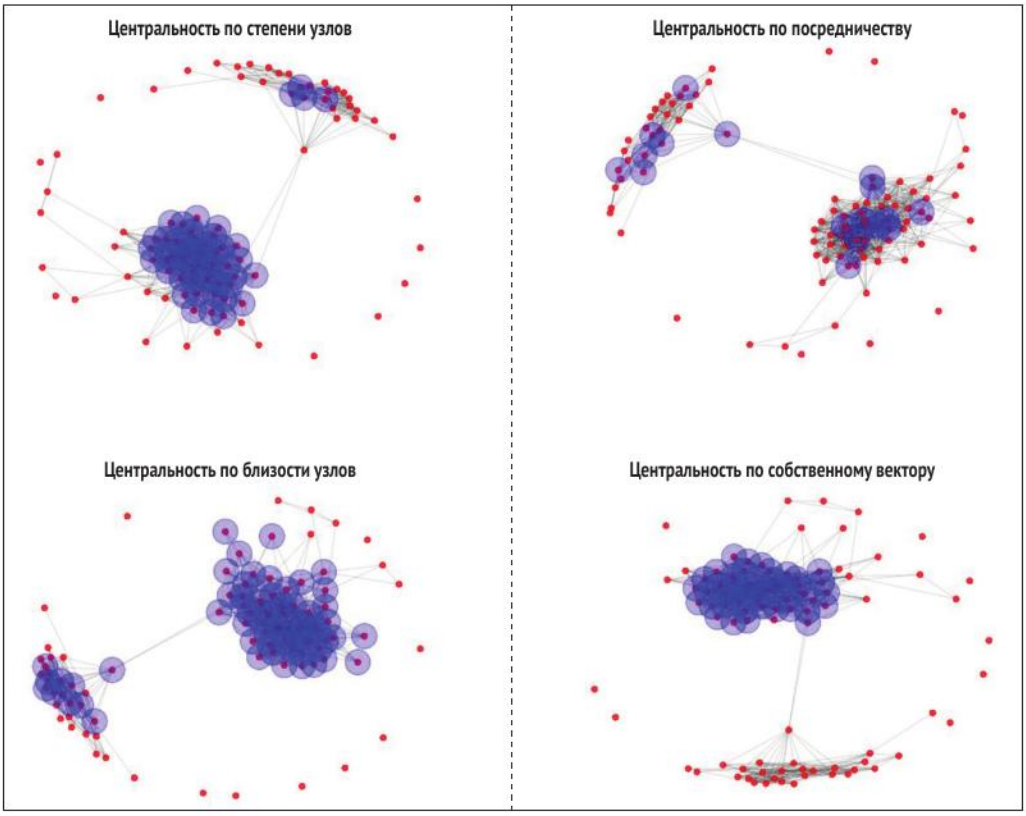

Рис. А.3 ❖ Визуализация мер центральности

Как показывает центральность по степени, исследуемое сообщество оказалась достаточно равномерным, однако центр общения находится в нижней компоненте связности (*A*), где сосредоточено большинство лидеров. Практически все члены сообщества вовлечены в общение, за исключением нескольких «изолянтов». Мера центральности по близости, определяющая, насколько данный узел близок ко всем остальным узлам в сети, показывает, что в сообществе есть члены, напрямую связанные со всеми остальными, и опять же их большинство находится в компоненте *A*. Посредничество в сети выражено относительно слабо, поскольку все члены сообщества тесно между собой связаны. Мера центральности по собственному вектору показывает центральных акторов, то есть с наименьшей удаленностью от других, и указывает на компоненту *A* как место, где находится инициатор общения.

Прочие операции

Коэффициент кластеризации — это степень вероятности, что два разных пользователя, связанных с конкретным пользователем, тоже связаны.

```
# Коэффициент кластеризации конкретного узла
print('Коэффициент кластеризации узла 1:', nx.clustering(G, list(G)[1]))
# Коэффициент кластеризации всех узлов (в словаре)
clust_coefficients = nx.clustering(G)

# Средний коэффициент кластеризации
avg_clust = sum(clust_coefficients.values()) / len(clust_coefficients)
print('Коэффициент кластеризации (вручную):', avg_clust)
# Либо непосредственно с использованием встроенного метода
print('Коэффициент кластеризации:', nx.average_clustering(G))

# Коэффициент кластеризации узла 1: 0.7663817663817664
# Коэффициент кластеризации (вручную): 0.6212776594189214
# Коэффициент кластеризации: 0.6212776594189214
```

Авторитетность, или алгоритм PageRank. Данный алгоритм, как правило, используется для веб-страниц или иных документов, связанных ссылками. Чем больше ссылок на страницу, тем она важнее. Он также применяется и к графам. Авторитетность в социальном графе можно анализировать по-разному. Самый простой — отсортировать участников по количеству входящих ребер. Чем больше ребер, тем выше авторитет. Такой способ годится для небольших графов. При выполнении поиска в вебе в качестве одного из критериев для составления рейтинга авторитетности страниц поисковый механизм Google использует алгоритм PageRank. Он вычисляется на основе случайного блуждания по графу, где узлами являются страницы, а ребрами — ссылки между страницами, если таковые имеются. Случайный блуждатель двигается по графу и время от времени перемещается на случайный узел и начинает блуждание заново. PageRank равен доли пребывания на каком-то узле за все время блуждания. Чем он больше, тем узел более авторитетен. Ниже показаны 5 самых авторитетных узлов графа.

```
import operator
print('Рейтинг авторитетности узлов:', sorted(nx.pagerank(G).items(),
                        key=operator.itemgetter(1), reverse=True)[:5])
```

Представленный в данном приложении материал призван лишь продемонстрировать простоту и доступность работы с социальными сетями на основе библиотеки networkx. В реальных условиях стоят куда более сложные и вычислительно емкие задачи. Например, социальные сети можно использовать в качестве источника для создания резюме. В связи с тем, что информация в социальных сетях имеет обрывочный характер, одной из главных задач могут быть поиск одного и того же пользователя в разных социальных сетях и на основе данных, полученных из социальных сетей, создание полного резюме.

Предметный указатель

A

Anaconda, научный дистрибутив, 34
Atom
  URL-адрес, 206
  обработка канала, 206
  описание, 206

B

BeautifulSoup, библиотека, 157
Blogger API, использование, 203

C

CheeseShop, каталог библиотек, 35
conda
  URL-адрес, 34
  описание, 34
Cython, компилируемый Python, 32

D

D3.js, библиотека, URL-адрес, 51
DBpedia, краудсорсинговый проект
  URL-адрес, 255
  анализ связей, 258
  описание, 258

E

ElementTree, модуль, 264
ElementTree, пакет, 177
etag, дескриптор сущности, 156

F

facebook-sdk, библиотека, 121
Facebook для разработчиков, URL-адрес, 118, 130
feedparser, библиотека, 206
Flask, микрофреймворк
  URL-адрес, 149
  маршруты, 151
  описание, 149
  шаблоны, 151
Folium, библиотека
  URL-адрес, 110
  описание, 51, 110

G

Gensim, библиотека, 47
GeoJSON, формат
  URL-адрес, 51, 108
  описание, 108
GitHub, веб-служба
  URL-адрес, 231, 232
  URL-адрес для доступа к данным, 232
  анализ открытого программного обеспечения, 231
Google+ API
  URL-адрес для действий, 156
  действия, 154
  заметки, 154
  описание, 144
  поиск, 147
Google Maps
  URL-адрес, 263
  нанесение геоданных, 263
Graph API платформы Facebook
  аутентификация, 119
  безопасность, 119
  доступ при помощи Python, 121
  методы, 122
  описание, 117
  регистрация приложения, 118

H

HTTP, интерфейс для клиентского модуля на Python, 243
httpbin.org, URL-адрес, 243

J

jieba, библиотека, URL-адрес, 213
Jinja2, библиотека
  URL-адрес, 151
  описание, 149
JSON-LD, формат
  URL-адрес, 256
  URL-адрес для примера, 257
  описание, 256

K
KML, язык разметки Keyhole, URL-адрес, 263
k ближайших соседей, алгоритм (k-NN)
  URL-адрес, 186
  описание, 184, 186
k-кратная перекрестная проверка, 188
K-средних, алгоритм, 99

L
Leaflet, библиотека
  URL-адрес, 51
  описание, 110
libxml2, низкоуровневая библиотека, 177
lxml, библиотека
  URL-адрес, 177
  описание, 261

M
matplotlib, библиотека, 40
mf2py, пакет, 261
microformat2, стандарт, URL-адрес, 252
Miniconda, 34

N
N3, формат, URL-адрес, 256
NetworkX, библиотека, 48, 105
NLTK, библиотека, 73, 142, 159
  URL-адрес, 44
  описание, 44, 212
NumPy, библиотека
  описание, 35
  характеристики, 37
n-граммы, получение словосочетаний, 163

O
OWL (Web Ontology Language – язык описания онтологий), URL-адрес, 250

P
pandas, библиотека, функционал, 37
PDF (Portable Document Format, переносимый формат документа), 50
pickle, библиотека, 194
Pillow, библиотека, 141
PIL (Python Imaging Library), графическая библиотека, 141
pip, диспетчер библиотек, описание, 33
Plotly, веб-служба, URL-адрес, 51
PNG (Portable Network Graphics, переносимая сетевая графика), 50
Prolog, язык логического программирования, 251
PyGithub, библиотека, URL-адрес, 232
PyKML, библиотека, 264
Pyro4 (PYthon Remote Objects), библиотека, 47
Py-StackExchange, библиотека
  URL-адрес, 169
  описание, 169
Python, язык программирования
  декораторы, 150
  доступ к Graph API платформы Facebook, 121
  обработка данных, 51

R
RDFLib, библиотека, 258
RDF (Resource Description Framework, среда описания ресурса)
  URL-адрес, 255
  описание, 255
requests, библиотека, 210, 243, 260
RESTful API, технология, 24
REST (Representational State Transfer, передача состояния представления), 24
RSS (Rich Site Summary, обогащенная сводка сайта)
  URL-адрес, 206
  URL-адрес для получения каналов, 207
  обработка канала, 206
  описание, 206

S
schema.org, инициатива
  URL-адрес, 257
  описание, 257
scikit-learn, библиотека, 40, 182
Scrapy, библиотека
  URL-адрес, 210
  описание, 237
Series, числовой ряд, 37
SNOMED CT, медицинский словарь, URL-адрес, 249

Stack Exchange
дампы данных, получение и обработка, 175
классификация текстов, 189
Stack Exchange API
URL-адрес, 167, 168
описание, 168, 169
поиск вопросов с тегами, 170
поиск пользователей, 172
Stack Overflow, URL-адрес, 167
Streaming API
URL-адрес, 66
применение, 66

T

TF-IDF, статистическая мера
URL-адрес, 99, 158
на заметках, 157
получение словосочетаний при помощи n-грамм, 163
throttle, описание ограничений, URL-адрес, 168
Tweepy, библиотека, 58
Twitter API
URL-адрес, 58
ограничения на частоту запросов, 56
описание, 56, 83
поиск против потока, 57
Twitter, социальная сеть
URL-адрес, 56, 83
анализ, 83
выборка данных из соцсети, 58
друзья, 83
общий обзор, 55
пользователи, 83
читатели, 83

V

virtualenv, библиотека, 33

W

wordcloud, библиотека, 141
WordNet, URL-адрес, 249
WordPress.com API, применение, 200

X

XFN (XHTML Friends Network, микроформат для разметки социальных взаимоотношений), 253

Y

Yelp
URL-адрес, 238
анализ местных предприятий, 238
справочная информация, 242
YouTube
URL-адрес, 226
анализ видеоматериалов, 226

А

Алгоритмы классификации
k ближайших соседей (k-NN), 186
метод опросных векторов (SVM), 186
наивный байесовский (NB), 185
Анализ графов, 31
Анализ социальных сетей (Social Network Analysis, SNA), 48
Анализ текста на заметках, 157
Анализ текстов, 30
Аналитический механизм, 30
Атрибут order, значения, 229
Атрибуты, URL-адрес, 123

Б

Банк синтаксических деревьев Penn Treebank, проект, URL-адрес, 213
Беседа, анализ, 104
Библиотеки Python, URL-адрес, 232
Блоги
Blogger API, 203
WordPress.com API, 200
выборка данных из Интернета, 210
описание, 199
получение данных, 200
разбор канала Atom, 206
разбор канала RSS, 206
Британская вещательная корпорация (Би-би-си, BBC), 206

В

Введение в анализ данных, 29
Веб-API, 24
Веб-интерфейс
вывод результатов поиска, 149
декораторы в Python, 150
маршруты Flask, 151
шаблоны Flask, 151
Веб-сайт «Кинофильмы & ТВ», 175

Веб-сайт вопросов и ответов
URL-адрес, 167
описание, 165
Веб-сайты, получение данных, 200
Версия API, 118
Визуализация данных, 49
Википедия, веб-энциклопедия
извлечение географических данных, 260
описание, 207
получение данных, 207
Влияние, измерение степени, 94
Влиятельное лицо
URL-адрес, 94
описание, 94
Вовлеченность, измерение степени, 94
Вопросно-ответные системы (QA), 166
Вопросы с тегами, поиск, 170
Вопрос, элемент row, атрибут Id, 176
Восточное стандартное время (EST), 129
Всемирное координированное время (UTC), 129
Выборка данных из веба, 210

Г

Географические данные
извлечение из Википедии, 260
нанесение на карты Google Maps, 263
Географические координаты
анализ, 260
извлечение геоданных из Википедии, 260
нанесение геоданных на карты Google Maps, 263

Д

Данные
выборка данных из Twitter, 58
получение твитов из ленты, 60
применение Streaming API, 66
структура твита, 62
Двенадцатифакторное веб-приложение, URL-адрес, 58
Действия Google+, поля, 156
Декораторы в Python, 150
Дирижер данных
граф зависимостей, 54
диспетчер потока операций, 54
преимущества, 54
шаблоны задач, 54
Диспетчер Google API, URL-адрес, 144, 227
Длинный хвост, URL-адрес, 78

З

Закон Ципфа, URL-адрес, 78, 158

И

Идентификатор клиента OAuth, 146
Идентификатор приложения, 118
Иерархия знаний, 23
Извлечение информации, 220
Инструментарий Python
анализ социальных сетей (SNA), 48
визуализация данных, 49
для науки о данных, 31
машинное обучение, 39
настройка среды разработки на Python, 32
обработка естественного языка (NLP), 43
эффективный анализ данных, 35
Интернет-архивы, URL-адрес, 175

К

Каскадные таблицы стилей (CSS), 152
Каталог библиотек Python (PyPI)
URL-адрес, 35
описание, 35
Классификатор, встраивание в приложение реального времени, 193
Классификация текстов
алгоритмы классификации, 184
в сети Stack Exchange, 189
встраивание классификатора в приложение реального времени, 193
идентификация жанра, 180
на тегах вопросов, 180
обучение с учителем, 180
определение языка, 180
оценка, 187
фильтрация спама, 180
Кластеризация
URL-адрес, 104
описание, 98

Клиентский модуль на Python
интерфейс HTTP, 243
создание, 243
Ключ API, 146
Ключ потребителя, 56
Коды состояний, Twitter API, URL-адрес, 67
Конвейеры данных, создание, 53
Конечная точка, URL-адреса
API search.list, 228
followers, 84
followers/ids, 84
followers/list, 84
/info, 170
/sites/$site/posts, 200
/users, 174
users/lookup, 84
Консорциум Всемирной паутины (W3C), 247

Л

Лексемизация слов, 212
Лексемизация (токенизация), 73

М

Маркер Oauth2, URL-адрес, 232
Маркированный ориентированный мультиграф, 255
Машинное обучение, 39
Машины опорных векторов (SVM), описание, 39
Местное предприятие Yelp, 238
Метка, 40
Метод questions(), 171
значения, 172
Метод опорных векторов (SVM)
URL-адрес, 186
описание, 186
Микроформаты, URL-адрес, 252

Н

Наивный Байес (NB), описание, 39
Наивный байесовский (NB)
URL-адрес, 185
описание, 185
Направленные ациклические графы (DAG), URL-адрес, 104
Наука о данных, инструменты Python, 31
Научно-исследовательские и опытно-конструкторские работы (R&D), 27
Нейронные сети (NN), 39
Нормализация регистра букв, 74, 215
Нормализация слов
выделение основ слов, 215
лемматизация, 217
нормализация регистра букв, 215
описание, 215
сопоставление синонимов, 219
удаление стоп-слов, 218

О

Облака дескрипторов, URL-адрес, 141
Облако слов, 141
Обработка данных на Python, 51
Обработка естественного языка (NLP)
извлечение информации, 220
описание, 43, 199, 210
предобработка текста, 211
Обучение без учителя, 40
Обучение с учителем
URL-адрес, 184
классификация текстов, 180
Объект пользователя, атрибуты, 234
Объекты-местоположения, атрибуты, 241
Объекты-репозитории, атрибуты, 238
Ограничения на частоту запросов
URL-адрес, 56, 124
описание, 56
ОЕЯ. *См.* Обработка естественного языка
Онтология, 249
Определение границ предложений, 211
Открытая авторизация (OAuth), 28
Открытое программное обеспечение, анализ на GitHub, 231
Открытые данные, 254

П

Паутина данных
RDF (среда описания ресурса), 255
schema.org, инициатива, 257
микроформаты, 252
описание, 247

открытые данные, 254
связанные данные, 254
словарь Семантической паутины, 249
формат JSON-LD, 256
Поиск по сравнению с потоком, 57
Поле ответа, 166
Полисемия, URL-адрес, 219
Пользователи Twitter
Twitter API, 83
измерение степени влияния, 94
измерение степени вовлеченности, 94
профили друзей, скачивание, 87
профили читателей, скачивание, 87
профиль пользователя, структура, 85
сетевой анализ, 89
Пользователь, поиск, 172, 175
Правило 80-20, 78
Предобработка, 221
Приложение, регистрация, 118
Приложение реального времени, встраивание классификатора, 193
Приложения анализа текстов
анализ мнений, 31
выделение сущностей, 31
классификация документов, 30
кластеризация документов, 30
реферирование документов, 31
Принцип Парето, URL-адрес, 78
Проводник Graph API, URL-адрес, 120
Профили друзей, скачивание, 87
Профили читателей, скачивание, 87
Профиль пользователя
атрибуты, 87
структура, 85
Пятизвездочная система оценок, 254

Р

Распознавание именованных сущностей (NER), 220
Распределение частоты слов естественного языка по Ципфу, описание, 158
Реакции Facebook, URL-адрес, 134
Ресурсы REST API, URL-адрес, 200
Ретвит (RT), 73

С

Связанные данные
URL-адрес, 254
описание, 254
Сегментирование, 222
Секрет потребителя, 56
Секрет приложения, 118
Семантическая паутина
URL-адрес, 247
описание, 247
словарь, 249
Сеть, проведение анализа, 89
Словарь Семантической паутины
допущения об открытости мира, 251
допущения о закрытости мира, 251
онтология, 249
семантическая разметка, 249
таксономия, 250
фолксономия, 250
язык разметки, 249
Сокращения в английском языке, URL-адрес, 76
Сообщения, анализ
описание, 124
структура, 127
частотно-временной анализ, 129
Сообщения, получение со страницы, 131
Graph API 2.6, 134
реакции Facebook, 134
Составной конвейер данных, создание, 53
Социальные API, 226
Социальные медиа
возможности, 22, 23
вызовы, 22, 25
функциональные особенности, 22
Среда разработки на Python
Anaconda, 34
conda, 34
Miniconda, 34
pip, 33
virtualenv, 33
Среднее время по гринвичскому меридиану (GMT), 68
Степенная зависимость, URL-адрес, 78
Страница с результатами поиска (SERP), 166

Страницы Facebook, анализ
визуализация сообщений в виде облака слов, 141
выполнение, 129
измерение степени вовлеченности, 135
получение сообщений со страницы, 131
Схема RDF, URL-адрес, 256

Т

Твиты
анализ временных рядов, 79
анализ сущностей, 69
анализ текстов, 73
атрибуты, 62
нанесение на географическую карту, 107
получение из ленты, 60
структура, 62
Твиты, нанесение на географическую карту
Folium, библиотека, 110
GeoJSON, формат, 108
Теги вопросов, классификация текста, 180
Текст, предобработка
лексемизация слов, 212
нормализация слов, 215
определение границ предложений, 211
частеречная (POS) разметка, 213
Теорема Байеса, URL-адрес, 185
Теория графов, URL-адрес, 255
Тестировщики, 119
Тестовые данные, 39
Тренировочные данные, 39
Триплеты, 255

У

Удаление стоп-слов, 73, 218
Учетная запись Google, URL-адрес, 144

Ц

Целостность данных, 26
Цель, 40

Ч

Частеречная (POS) разметка, 213
Частота термина (TF), 99, 157
Читатели, анализ, 98

Э

Эвклидово расстояние, URL-адрес, 186
Эффективный анализ данных, 35

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: books@alians-kniga.ru.

Марко Бонцанини

Анализ социальных медиа на Python

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Логунов А. В.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать офсетная.
Усл. печ. л. 27. Тираж 200 экз.

Веб-сайт издательства: **www.дмк.рф**