O'REILLY®

Прикладной анализ текстовых данных на Python

МАШИННОЕ ОБУЧЕНИЕ И СОЗДАНИЕ ПРИЛОЖЕНИЙ ОБРАБОТКИ ЕСТЕСТВЕННОГО ЯЗЫКА

Бенджамин Бенгфорт
Ребекка Билбро и Тони Охеда

ПИТЕР®

Applied Text Analysis with Python

Enabling Language-Aware Data Products with Machine Learning

Benjamin Bengfort, Rebecca Bilbro, and Tony Ojeda

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Бенджамин Бенгфорт
Ребекка Билбро и Тони Охеда

Прикладной анализ текстовых данных на Python

МАШИННОЕ ОБУЧЕНИЕ И СОЗДАНИЕ ПРИЛОЖЕНИЙ ОБРАБОТКИ ЕСТЕСТВЕННОГО ЯЗЫКА

ПИТЕР®

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

2019

ББК 32.973.233-018
УДК 004.62
Б46

Бенгфорт Бенджамин, Билбро Ребекка, Охеда Тони

Б46 Прикладной анализ текстовых данных на Python. Машинное обучение и создание приложений обработки естественного языка. — СПб.: Питер, 2019. — 368 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-1153-4

Технологии анализа текстовой информации стремительно меняются под влиянием машинного обучения. Нейронные сети из теоретических научных исследований перешли в реальную жизнь, и анализ текста активно интегрируется в программные решения. Нейронные сети способны решать самые сложные задачи обработки естественного языка, никого не удивляет машинный перевод, «беседа» с роботом в интернет-магазине, перефразирование, ответы на вопросы и поддержание диалога. Почему же Сири, Алекса и Алиса не хотят нас понимать, Google находит не то, что мы ищем, а машинные переводчики веселят нас примерами «трудностей перевода» с китайского на албанский? Ответ кроется в мелочах — в алгоритмах, которые правильно работают в теории, но сложно реализуются на практике. Научитесь применять методы машинного обучения для анализа текста в реальных задачах, используя возможности и библиотеки Python. От поиска модели и предварительной обработки данных вы перейдете к приемам классификации и кластеризации текстов, затем приступите к визуальной интерпретации, анализу графов, а после знакомства с приемами масштабирования научитесь использовать глубокое обучение для анализа текста.

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

ББК 32.973.233-018
УДК 004.62

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1491963043 англ. Authorized Russian translation of the English edition of Mobile Applied Text Analysis with Python, ISBN 9781491963043 © 2018 Benjamin Bengfort, Rebecca Bilbro
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same

ISBN 978-5-4461-1153-4 © Перевод на русский язык ООО Издательство «Питер», 2019
© Издание на русском языке, оформление ООО Издательство «Питер», 2019
© Серия «Бестселлеры O'Reilly», 2019

Краткое содержание

Вступление 11

Глава 1. Естественные языки и вычисления 22

Глава 2. Создание собственного корпуса 42

Глава 3. Предварительная обработка и преобразование корпуса 63

Глава 4. Конвейеры векторизации и преобразования 82

Глава 5. Классификация в текстовом анализе 112

Глава 6. Кластеризация для выявления сходств в тексте 130

Глава 7. Контекстно-зависимый анализ текста 161

Глава 8. Визуализация текста 190

Глава 9. Графовые методы анализа текста 223

Глава 10. Чат-боты 249

Глава 11. Масштабирование анализа текста 288

Глава 12. Глубокое обучение и не только 323

Глоссарий 345

Оглавление

Вступление ... 11
Сложности компьютерной обработки естественного языка ... 12
Лингвистические данные: лексемы и слова ... 12
Внедрение машинного обучения ... 14
Инструменты для анализа текста ... 15
О чем рассказывается в этой книге ... 16
Кому адресована эта книга ... 17
Примеры кода и репозиторий на GitHub ... 18
Типографские соглашения ... 19
Использование программного кода примеров ... 19
От издательства ... 20
Благодарности ... 20

Глава 1. Естественные языки и вычисления ... 22
Парадигма Data Science ... 23
Приложения данных, основанные на анализе естественного языка ... 25
Конвейер приложения данных ... 27
Тройка выбора модели ... 29
Язык как данные ... 31
Компьютерная модель языка ... 31
Лингвистические признаки ... 33
Контекстные признаки ... 36
Структурные признаки ... 38
В заключение ... 41

Глава 2. Создание собственного корпуса ... 42
Что такое корпус? ... 43
Предметные корпусы ... 43
Движок сбора данных Baleen ... 44
Управление корпусом данных ... 46
Структура корпуса на диске ... 48

Объекты чтения корпусов ... 51
Потоковый доступ к данным с помощью NLTK ... 53
Чтение корпуса HTML ... 56
Чтение корпуса из базы данных ... 60
В заключение ... 62

Глава 3. Предварительная обработка и преобразование корпуса ... 63
Разбивка документов ... 64
Выявление и извлечение основного контента ... 65
Разделение документов на абзацы ... 66
Сегментация: выделение предложений ... 68
Лексемизация: выделение лексем ... 70
Маркировка частями речи ... 71
Промежуточный анализ корпуса ... 73
Трансформация корпуса ... 74
Чтение предварительно обработанного корпуса ... 79
В заключение ... 81

Глава 4. Конвейеры векторизации и преобразования ... 82
Слова в пространстве ... 83
Частотные векторы ... 85
Прямое кодирование ... 87
Частота слова — обратная частота документа ... 90
Распределенное представление ... 93
Scikit-Learn API ... 97
Интерфейс BaseEstimator ... 97
Расширение TransformerMixin ... 99
Конвейеры ... 104
Основы конвейеров ... 105
Поиск по сетке для оптимизации гиперпараметров ... 106
Усовершенствование извлечения признаков с помощью объектов FeatureUnion ... 107
В заключение ... 110

Глава 5. Классификация в текстовом анализе ... 112
Классификация текста ... 113
Идентификация задач классификации ... 113

Модели классификации ... 115
Создание приложений классификации текста ... 117
Перекрестная проверка ... 118
Конструирование модели ... 122
Оценка модели ... 124
Эксплуатация модели ... 128
В заключение ... 129

Глава 6. Кластеризация для выявления сходств в тексте ... 130
Обучение на текстовых данных без учителя ... 131
Кластеризация документов по сходству ... 132
Метрики расстояния ... 133
Партитивная кластеризация ... 136
Иерархическая кластеризация ... 142
Моделирование тематики документов ... 146
Латентное размещение Дирихле ... 146
Латентно-семантический анализ ... 155
Неотрицательное матричное разложение ... 157
В заключение ... 159

Глава 7. Контекстно-зависимый анализ текста ... 161
Извлечение признаков на основе грамматики ... 162
Контекстно-свободные грамматики ... 163
Синтаксические парсеры ... 163
Извлечение ключевых фраз ... 165
Извлечение сущностей ... 168
Извлечение признаков на основе *n*-грамм ... 169
Чтение корпуса с поддержкой *n*-грамм ... 171
Выбор размера *n*-грамм ... 173
Значимые словосочетания ... 174
Модели языка *n*-грамм ... 177
Частота и условная частота ... 178
Оценка максимальной вероятности ... 181
Неизвестные слова: возвраты и сглаживание ... 184
Генерация языка ... 186
В заключение ... 188

Глава 8. Визуализация текста ... 190
Визуализация пространства признаков ... 191
Визуальный анализ признаков ... 191
Управление конструированием признаков ... 202
Диагностика моделей ... 210
Визуализация кластеров ... 211
Визуализация классов ... 213
Диагностика ошибок классификации ... 214
Визуальная настройка ... 218
Оценка силуэта и локтевые кривые ... 219
В заключение ... 222

Глава 9. Графовые методы анализа текста ... 223
Вычисление и анализ графов ... 225
Создание тезауруса на основе графа ... 225
Анализ структуры графа ... 227
Визуальный анализ графов ... 228
Извлечение графов из текста ... 229
Создание социального графа ... 230
Исследование социального графа ... 233
Разрешение сущностей ... 241
Разрешение сущностей в графе ... 242
Блокирование по структуре ... 244
Нечеткое блокирование ... 244
В заключение ... 247

Глава 10. Чат-боты ... 249
Основы диалогового взаимодействия ... 250
Диалог: непродолжительный обмен ... 253
Управление диалогом ... 256
Правила вежливой беседы ... 258
Приветствие и прощание ... 259
Обработка ошибок при общении ... 264
Занимательные вопросы ... 267
Анализ зависимостей ... 268
Анализ составляющих ... 269

Выявление вопроса ... 272
От столовых ложек к граммам ... 274
Обучение для рекомендаций ... 279
Соседство ... 281
Предложение рекомендаций ... 284
В заключение ... 286

Глава 11. Масштабирование анализа текста ... 288
Модуль multiprocessing ... 289
Запуск параллельных задач ... 292
Пулы процессов и очереди ... 297
Параллельная обработка корпуса ... 299
Кластерные вычисления с использованием Spark ... 301
Устройство заданий в Spark ... 302
Распределение корпуса ... 304
Операции RDD ... 306
Обработка естественного языка в Spark ... 308
В заключение ... 321

Глава 12. Глубокое обучение и не только ... 323
Прикладные нейронные сети ... 324
Нейронные модели языка ... 324
Искусственные нейронные сети ... 325
Архитектуры глубокого обучения ... 331
Анализ эмоциональной окраски ... 336
Глубокий анализ структуры ... 338
Будущее (почти) наступило ... 343

Глоссарий ... 345

Об авторах ... 362

Выходные данные ... 364

Вступление

Мы живем в мире, все больше наполняющемся цифровыми помощниками, позволяющими взаимодействовать с другими людьми и обширными информационными ресурсами. Привлекательность этих умных устройств отчасти обусловлена тем, что они не просто передают информацию, но также *понимают* ее до некоторой степени, облегчая коммуникацию высокого уровня, комбинируя, фильтруя и обобщая данные в легкоусвояемую форму. Такие приложения, как машинные переводчики, системы «вопрос-ответ», инструменты транскрипции голосовой информации и обобщения текста, а также чат-боты, становятся неотъемлемой частью нашей жизни в компьютерном мире.

Если вы взяли в руки эту книгу, значит, вас, как и нас, интересуют возможности внедрения компонентов анализа естественного языка в широкий спектр прикладного ПО. Компоненты анализа языка основаны на современной инфраструктуре анализа текстовой информации: коллекции приемов и методов, объединяющей инструменты для работы со строками; лексических ресурсах; компьютерной лингвистике; алгоритмах машинного обучения, преобразующих данные на естественном языке в машинную форму и обратно. Однако, прежде чем приступить к обсуждению этих приемов и методов, важно определить проблемы и возможности этой инфраструктуры и ответить на вопрос: почему это происходит сейчас?

Типичный выпускник имеет в своем словаре примерно 60 000 слов и тысячи грамматических конструкций, необходимых для общения в профессиональном контексте. На первый взгляд это довольно много, но представьте, насколько просто было бы написать короткий сценарий на Python, быстро отыскивающий определение, этимологию и примеры использования любого термина в электронном словаре. Фактически в повседневной жизни средний американец использует лишь десятую часть лингвистических конструкций, зафиксированных в Оксфордском словаре, и только 5 % из распознаваемых поисковой системой Google.

И все же мгновенного доступа к правилам и определениям недостаточно для анализа текста. Если бы это было так, Сири (Siri) и Алекса (Alexa)[1] прекрасно

[1] Голосовые виртуальные помощники. Сири разработана в компании Apple, а Алекса — в Amazon. — *Примеч. пер.*

понимали бы нас, Google возвращал бы только значимые результаты поиска, а мы могли бы мгновенно общаться с любыми людьми в мире, говорящими на любом языке. Почему компьютеры так сильно отстают в решении задач, которые люди способны решать без запинки, с самого раннего возраста — задолго до того, как накопят словарный запас, которым обладают взрослые люди? Очевидно, что для общения на естественном языке простого запоминания недостаточно, как недостаточно и детерминированных вычислительных методов.

Сложности компьютерной обработки естественного языка

Естественные языки определяются не правилами, а контекстом *использования*, который требуется реконструировать для компьютерной обработки. Часто мы сами определяем значения используемых слов, хотя и совместно с другими участниками беседы. Словом «краб» мы можем обозначить морское животное или человека с унылым нравом или имеющего характерную привычку двигаться бочком, но при этом оба — говорящий/автор и слушатель/читатель — должны согласиться с общим пониманием в ходе диалога. Поэтому язык обычно ограничивается обществом и регионом — передать смысл часто намного проще людям, имеющим жизненный опыт, похожий на ваш.

В отличие от формальных языков, которые всегда являются предметными, естественные языки намного более универсальны. Мы используем одно и то же слово при заказе морепродуктов на обед, в поэме, описывающей уныние и недовольство, и для обозначения астрономической туманности. Для поддержания множества смыслов язык должен быть избыточным. Избыточность представляет серьезную проблему, потому что мы не можем (и не делаем этого) указать буквальный смысл для каждой ассоциации, каждый символ по умолчанию является неоднозначным. Лексическая и структурная неоднозначность является основным достижением человеческого языка; она не только дает нам возможность генерировать новые идеи, но также позволяет общаться людям с разным опытом и культурой, несмотря на почти гарантированные случайные недоразумения.

Лингвистические данные: лексемы и слова

Чтобы полностью использовать данные, закодированные в языке, мы должны научить наш разум думать о языке не как о понятном и естественном, но как о произвольном и неоднозначном. Единицей анализа текста является *лексема*,

строка кодированных байтов, представляющая собой текст. *Слова*, напротив, — это символы, представляющие смысл и отображающие текстовые или вербальные конструкции как звук или изображение. Лексемы не являются словами (хотя нам трудно смотреть на лексемы и не видеть слов). Рассмотрим лексему `"crab"`, изображенную на рис. B.1. Эта лексема представляет смысловое слово *crab-n1* — первое определение существительного, в котором используется лексема, обозначающее ракообразное, пригодное к употреблению в пищу, живущее в морях и океанах и имеющее клешни, которыми оно может ущипнуть.

Рис. B.1. Слова отображают символы в представления

Все другие понятия некоторым способом связаны с символом, и все же символ совершенно произволен; аналогичная связь будет вызывать, например, у греческого читателя немного иные ассоциации, но с тем же основным смыслом. Это происходит потому, что слова не имеют универсального фиксированного смысла, не зависящего от таких контекстов, как культура и язык. Англоговорящие читатели используют адаптивные формы слов с приставками и суффиксами, изменяющими время, род и т. д. Читатели, говорящие на китайском языке, напротив, распознают множество пиктографических изображений, смысл которых определяет порядок их следования.

Избыточность, неоднозначность и зрительные ассоциации делают естественные языки динамичными, быстро развивающимися, способными отражать текущий опыт. В настоящее время можно смело утверждать, что лингвистическое исследование эмограмм (смайликов) продвинулось в достаточной мере, чтобы перевести «Моби Дика»![1] Даже если бы мы могли систематически развивать

[1] Fred Benenson, *«Emoji Dick»* (2013), http://bit.ly/2GKft1n.

грамматику, определяющую правила использования эмограмм, к тому времени, когда мы закончили бы эту работу, язык ушел бы далеко вперед — даже язык эмограмм! Например, с того момента, как мы начали писать эту книгу, эмограмма с изображением пистолета (🔫) стала восприниматься не как оружие, а как игрушка (по крайней мере, когда она отображается на смартфоне), отражая культурный сдвиг в восприятии назначения этого символа.

Это не просто включение новых символов и структур, адаптирующих язык, но также включение новых определений, контекстов и приемов использования. Лексема «батарея» изменила свой смысл в результате развития электроники и означает «хранилище, преобразующее химическую энергию в электрическую». Однако, согласно службе Google Books Ngram Viewer[1], в XIX и в начале XX вв. слово «батарея» чаще использовалось для обозначения «артиллерийского или минометного подразделения из нескольких орудий или минометов, а также позиции, которую занимает такое подразделение». Контекст понимания языка зависит не только от окружающего текста, но также от периода времени. Для четкого определения смысла слов требуется больше вычислений, чем простой поиск в словаре.

Внедрение машинного обучения

Те же качества, которые делают естественный язык таким богатым инструментом человеческого общения, затрудняют его анализ с применением детерминированных правил. Гибкость интерпретации людьми объясняет, почему, имея всего 60 000 символьных представлений, мы способны превзойти компьютеры в мгновенном понимании языка. Поэтому в программной среде нам нужны столь же нечеткие и гибкие вычислительные методы, и поэтому в настоящее время для анализа текстов применяются статистические методы машинного обучения. Даже при том, что приложения обработки естественного языка существуют уже несколько десятилетий, добавление методов машинного обучения в эту сферу обеспечило определенную гибкость и скорость реагирования, которые при иных условиях были бы невозможны.

Целью машинного обучения является подгонка существующих данных под некоторую модель, создание представления реального мира, помогающего принимать решения или генерировать прогнозы на основе новых данных, путем поиска закономерностей в них. На практике для этого выбирается семейство моделей, определяющих связи между целевыми и входными данными, задается форма, включающая параметры и особенности, а затем с помощью некоторой процеду-

[1] Google, Google Books Ngram Viewer (2013), *http://bit.ly/2GNlKtk*.

ры оптимизации минимизируется ошибка модели на обучающих данных. Затем обученной модели можно передавать новые данные, на основе которых она будет строить прогноз и возвращать метки, вероятности, признаки принадлежности или значения. Задача состоит в том, чтобы найти баланс между способностью с высокой точностью находить закономерности в известных данных и способностью обобщения для анализа данных, которые модель не видела прежде.

Многие приложения для анализа естественного языка включают не одну, а целое множество моделей машинного обучения, взаимодействующих между собой и влияющих друг на друга. Модели могут повторно обучаться на новых данных, нацеливаться на новые пространства решений и даже настраиваться под конкретного пользователя для непрерывного развития по мере поступления новой информации и изменения различных аспектов приложения с течением времени. За кулисами приложения конкурирующие модели могут ранжироваться, стареть и в конечном счете исчезать. Это значит, что приложения машинного обучения реализуют жизненные циклы, обеспечивающие соответствие динамики развития и региональных особенностей языка с рабочим процессом поддержки и мониторинга.

Инструменты для анализа текста

Поскольку методы анализа текста применяются в первую очередь в машинном обучении, необходим язык программирования с богатым набором научных и вычислительных библиотек. На эту роль как нельзя лучше подходит язык Python, включающий в себя набор мощных библиотек, таких как Scikit-Learn, NLTK, Gensim, spaCy, NetworkX и Yellowbrick.

- *Scikit-Learn* — расширение для библиотеки SciPy (Scientific Python), предоставляющее прикладной интерфейс (API) для обобщенного машинного обучения. Основанное на Cython с поддержкой высокопроизводительных библиотек на C, таких как LAPACK, LibSVM, Boost и других, расширение Scikit-Learn сочетает высокую производительность с простотой использования методов анализа наборов данных малого и среднего размера. Это расширение, распространяемое с открытым исходным кодом и допускающее коммерческое использование, предоставляет единый интерфейс для многих моделей регрессии, классификации, кластеризации и уменьшения размерности, а также утилиты для перекрестной проверки и настройки гиперпараметров.
- *NLTK* (Natural Language Tool-Kit — пакет инструментов для обработки естественного языка) — это «батарейки в комплекте», ресурс для обработ-

ки естественного языка, написанный на Python экспертами в академических кругах. Первоначально создававшийся как инструмент для обучения обработке естественных языков, этот пакет содержит корпусы, лексические ресурсы, грамматики, алгоритмы обработки языков и предварительно обученные модели, позволяя программистам на Python быстро приступать к обработке текстовых данных на разных естественных языках.

- *Gensim* — надежная, эффективная и простая в использовании библиотека, главной целью которой является семантическое моделирование текста без учителя. Первоначально создававшаяся для поиска сходств в документах, в настоящее время эта библиотека предоставляет тематическое моделирование для методов латентно-семантического анализа и включает в себя другие библиотеки машинного обучения без учителя, такие как word2vec.
- *spaCy* реализует высококачественную обработку естественного языка, обертывая современные академические алгоритмы в простой и удобный API. В частности, spaCy позволяет выполнить предварительную обработку текста в подготовке к глубокому обучению и может использоваться для создания систем извлечения информации или анализа естественного языка на больших объемах текста.
- *NetworkX* — комплексный пакет для анализа графов, помогающий создавать, упорядочивать, анализировать сложные сетевые структуры и манипулировать ими. Несмотря на то что он не является библиотекой машинного обучения или анализа текстов, применение графовых структур данных позволяет кодировать сложные отношения, которые графовые алгоритмы способны анализировать и находить смысловые особенности, а следовательно, является важным инструментом анализа текста.
- *Yellowbrick* — комплект инструментов визуальной диагностики для анализа и интерпретации результатов машинного обучения. Дополняя Scikit-Learn API, пакет Yellowbrick предоставляет простые и понятные визуальные средства для выбора признаков, моделирования и настройки гиперпараметров, управления процессом выбора моделей, наиболее эффективно описывающих текстовые данные.

О чем рассказывается в этой книге

В этой книге рассказывается о применении методов машинного обучения для анализа текста с использованием только что перечисленных библиотек на Python. Прикладной характер книги предполагает, что мы сосредоточим свое внимание не на академической лингвистике или статистических моделях, а на

эффективном развертывании моделей, обученных на тексте внутри приложения.

Предлагаемая нами модель анализа текста напрямую связана с процессом машинного обучения — поиска модели, состоящей из признаков, алгоритма и гиперпараметров, которая давала бы лучшие результаты на обучающих данных, с целью оценки неизвестных данных. Этот процесс начинается с создания обучающего набора данных, который в сфере анализа текстов называют корпусом. Затем мы исследуем методы извлечения признаков и предварительной обработки для представления текста в виде числовых данных, понятных методам машинного обучения. Далее, познакомившись с некоторыми основами, мы перейдем к исследованию приемов классификации и кластеризации текста, рассказ о которых завершает первые главы книги.

В последующих главах основное внимание уделяется расширению моделей более богатыми наборами признаков и созданию приложений анализа текстов. Сначала мы посмотрим, как можно представить и внедрить контекст в виде признаков, затем перейдем к визуальной интерпретации для управления процессом выбора модели. Потом мы посмотрим, как анализировать сложные отношения, извлекаемые из текста с применением приемов анализа графов. После этого обратим свой взгляд в сторону диалоговых агентов и углубим наше понимание синтаксического и семантического анализа текста. В заключение книги будет представлено практическое обсуждение приемов масштабирования анализа текста в многопроцессорных системах с применением Spark, и, наконец, мы рассмотрим следующий этап анализа текста: глубокое обучение.

Кому адресована эта книга

Эта книга адресована программистам на Python, интересующимся применением методов обработки естественного языка и машинного обучения в своих программных продуктах. Мы не предполагаем наличия у наших читателей специальных академических или математических знаний и вместо этого основное внимание уделяем инструментам и приемам, а не пространным объяснениям. В первую очередь в этой книге обсуждается анализ текстов на английском языке, поэтому читателям пригодится хотя бы базовое знание грамматических сущностей, таких как существительные, глаголы, наречия и прилагательные, и того, как они связаны между собой. Читатели, не имеющие опыта в машинном обучении и лингвистике, но обладающие навыками программирования на Python, не будут чувствовать себя потерянными при изучении понятий, которые мы представим.

Примеры кода и репозиторий на GitHub

Примеры кода, которые встретятся вам в этой книге, описывают порядок реализации на Python решений конкретных задач. Поскольку они ориентированы на читателей книги, мы сократили их, часто опуская важные инструкции, необходимые для фактического выполнения, например инструкции `import`, импортирующие пакеты и модули из стандартной библиотеки. Кроме того, они часто основываются на примерах кода из предыдущих разделов или глав и иногда представляют собой фрагменты кода, которые нужно немного изменить, чтобы обеспечить их работоспособность в новом контексте. Например, мы можем определить класс, как показано ниже:

```
class Thing(object):

    def __init__(self, arg):
        self.property = arg
```

Это определение класса служит целям описания основных свойств и созданию основы для дальнейшего обсуждения деталей реализации. Позднее мы можем добавить в этот класс новые методы, как показано ниже:

```
...

    def method(self, *args, **kwargs):
        return self.property
```

Многоточие в начале фрагмента указывает на то, что это продолжение определения класса из предыдущего фрагмента. Это означает, что простое копирование фрагментов примера не позволит получить работающий код. Важно также отметить, что код предполагает работу с данными, хранящимися на диске, в каталоге, доступном для чтения выполняемой программе на Python. Мы приложили все усилия, чтобы сделать код как можно более обобщенным, но мы не можем гарантировать его работоспособность во всех операционных системах и со всеми источниками данных.

Для поддержки читателей, которые могут пожелать опробовать примеры у себя, мы реализовали полноценные действующие примеры и сохранили их в **репозитории на GitHub** (https://github.com/foxbook/atap). Эти примеры могут немного отличаться от приведенных на страницах книги, но должны с легкостью запускаться под управлением Python 3 в любой системе. Кроме того, имейте в виду, что код в репозитории продолжает обновляться; загляните в файл *README*, где описываются все изменения, имевшие место. Вы можете создать свою копию репозитория и изменить код для выполнения в вашем окружении, и мы настоятельно советуем вам сделать это!

Типографские соглашения

В этой книге приняты следующие типографские соглашения:

Курсив

Используется для обозначения новых терминов.

`Моноширинный шрифт`

Применяется для оформления листингов программ и программных элементов внутри обычного текста, таких как имена переменных и функций, базы данных, типы данных, переменных окружения, инструкции и ключевые слова.

`Моноширинный курсив`

Обозначает текст, который должен замещаться фактическими значениями, вводимыми пользователем или определяемыми из контекста.

Так выделяются советы и предложения.

Так обозначаются советы, предложения и примечания общего характера.

Так обозначаются предупреждения и предостережения.

Использование программного кода примеров

Вспомогательные материалы (примеры кода, упражнения и т. д.) доступны для загрузки по адресу: https://github.com/foxbook/atap.

Эта книга призвана оказать вам помощь в решении ваших задач. В общем случае, все примеры кода из этой книги вы можете использовать в своих программах и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Например, если вы разрабатываете программу и используете в ней несколько

отрывков программного кода из книги, вам не нужно обращаться за разрешением. Однако в случае продажи или распространения компакт-дисков с примерами из этой книги вам необходимо получить разрешение от издательства O'Reilly. Если вы отвечаете на вопросы, цитируя данную книгу или примеры из нее, получение разрешения не требуется. Но при включении существенных объемов программного кода примеров из этой книги в вашу документацию вам необходимо будет получить разрешение издательства.

Мы приветствуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN. Например: «*Applied Text Analysis with Python* by Benjamin Bengfort, Rebecca Bilbro, and Tony Ojeda (O'Reilly). 978-1-491-96304-3».

Ниже приводится определение BibTex для этой книги:

```
@book{
    title = {Applied {{Text Analysis}} with {{Python}}},
    subtitle = {Enabling Language Aware {{Data Products}}},
    shorttitle = {Applied {{Text Analysis}} with {{Python}}},
    publisher = {{O'Reilly Media, Inc.}},
    author = {Bengfort, Benjamin and Bilbro, Rebecca and Ojeda, Tony},
    month = jun,
    year = {2018}
}
```

За получением разрешения на использование значительных объемов программного кода примеров из этой книги обращайтесь по адресу permissions@oreilly.com.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Благодарности

Мы хотели бы поблагодарить наших научных редакторов за время и силы, потраченные ими на наши первые рукописи, и за отзывы, которые помогли существенно улучшить эту книгу. Ден Чуднов (Dan Chudnov) и Даррен Кук

(Darren Cook) оказали великолепную поддержку, которая помогла нам оставаться в курсе последних достижений, а Николь Доннелли (Nicole Donnelly) представила свое видение, позволившее нам адаптировать содержимое книги под нужды читателей. Мы также хотим поблагодарить Льва Константиновского (Lev Konstantinovskiy) и Костаса Ксироганнопулоса (Kostas Xirogiannopoulos), академический взгляд которых помог нам вести обсуждение на современном уровне.

Нет таких слов, которые в полной мере выразили бы нашу признательность вечно улыбающемуся и ободряющему редактору Николь Тач (Nicole Tache). Она вела этот проект с самого начала и верила в нас, даже когда финишная черта, как нам казалось, уходила от нас все дальше и дальше. Ее вера в наши силы, ее неоценимые отзывы и своевременные советы — вот те условия, благодаря которым появилась эта книга.

Мы благодарны нашим друзьям и семьям — мы не смогли бы справиться с этой работой без вашей поддержки. Нашим родителям, Рэнди (Randy) и Лили (Lily), Карле (Carla) и Гриффу (Griff) и Тони (Tony) и Терезе (Teresa); вы воспитали в нас творческое мышление, трудолюбие и любовь к учебе, благодаря которым мы смогли закончить эту книгу. Нашим супругам, Жаклин (Jacquelyn), Джеффу (Jeff) и Никки (Nikki); ваша неизменная поддержка, даже когда все сроки вышли и нам приходилось работать по ночам и выходным, очень много значила для нас. И, наконец, нашим детям, Ирене (Irena), Генри (Henry), Оскару (Oscar) и крошке Охеде (Ojeda). Мы надеемся, что когда-нибудь вы найдете эту книгу и подумаете: «Круто, наши родители писали книги в те времена, когда компьютеры еще не умели говорить как обычные люди... сколько же им лет?»

1 Естественные языки и вычисления

Приложения, использующие приемы обработки естественного языка для анализа текстовых и аудиоданных, становятся неотъемлемой частью нашей жизни. От нашего имени они просматривают огромные объемы информации в Сети и предлагают новые и персонализированные механизмы взаимодействия человека с компьютерами. Эти приложения настолько распространены, что мы привыкли к широкому спектру закулисных инструментов: от спам-фильтров, следящих за нашим почтовым трафиком, до поисковых систем, которые ведут нас прямо туда, куда мы хотим попасть, и виртуальных помощников, всегда готовых выслушать и ответить.

Информационные продукты с поддержкой анализа естественного языка находятся на пересечении экспериментальных исследований и практической разработки ПО. Приложения, анализирующие речь и текст, взаимодействуют непосредственно с пользователем, чьи ответы обеспечивают обратную связь, которая оказывает влияние и на приложение, и на результаты анализа. Этот благотворный цикл часто начинается с самого простого, но с течением времени может перерасти в мощную систему, возвращающую ценные результаты.

Как ни странно, даже при том, что потенциальные возможности внедрения анализа естественного языка в приложения продолжают увеличиваться, непропорционально большое количество внедрений осуществляется крупными игроками. Но почему этим не занимаются другие? Возможно, отчасти потому, что по мере распространения этих возможностей они становятся все менее заметными, маскируя сложность их реализации. А также потому, что развитие науки о данных еще не достигло уровня, необходимого для проникновения в культуру разработки программного обеспечения.

Мы полагаем, что приложения, основанные на использовании естественного языка, только начинают распространяться и в будущем возьмут на себя задачи, которые сейчас решаются с помощью форм и щелчков мышью. Для создания таких приложений индустрия разработки программного обеспечения должна принять на вооружение приемы, используемые в науке о данных и основанные на гипотезах. Чтобы обеспечить высокую надежность продуктов, реализующих методы анализа естественного языка, специалисты по обработке данных должны использовать приемы разработки программного обеспечения, помогающие создавать код высокого качества. Все эти меры объединяются под эгидой новой парадигмы науки о данных, которая способствует созданию *продуктов на основе методов анализа естественного языка* и является главной темой этой книги.

Парадигма Data Science

Благодаря инновациям в области машинного обучения и масштабируемой обработки данных, появившимся в последние десять лет, в наш обиход прочно вошли термины «наука о данных» и «приложение данных». Также появилась новая профессия, *специалист по обработке и анализу данных* (data scientist) — отчасти статистик, отчасти программист и отчасти эксперт в предметной области. Специалисты по обработке и анализу данных являются основными создателями ценностей в информационную эпоху, поэтому эта новая роль стала одной из самых важных и притягательных профессий XXI века, и одной из самых малопонятных.

Специалисты по обработке и анализу данных сочетают типично академическую работу — исследования и эксперименты — с процессом создания коммерческого продукта. Отчасти это объясняется тем, что многие специалисты по обработке данных прежде учились в аспирантуре (где обрели дополнительные знания и творческие навыки, необходимые в науке о данных), но главная причина в том, что процесс создания приложений данных в значительной степени основывается на экспериментах.

Основная сложность, которую подметили видные специалисты в этой области, связана с тем, что процесс исследования данных не всегда совместим с практикой разработки программного обеспечения. Данные могут быть непредсказуемыми, а положительный результат не всегда гарантирован. Как сказала Хилари Мейсон (Hilary Mason) о разработке приложений данных, «наука о данных не всегда отличается гибкостью»[1].

[1] Hillary Mason, *The Next Generation of Data Products* (2017), http://bit.ly/2GOF894

Или, говоря иначе:

> *Существуют фундаментальные различия между разработкой программного обеспечения и ценными идеями, рождающимися в ходе гибкого процесса. Потребность в эффективных идеях создает элемент неопределенности вокруг артефактов науки о данных — они могут быть «полными» в смысле программного обеспечения и при этом не иметь никакой ценности, потому что не несут настоящих, действенных идей... методологии гибкой разработки не справляются с этой неопределенностью.*
>
> *— Рассел Джерни (Russell Jurney), Agile Data Science 2.0*

Как результат, подразделения и сотрудники, занимающиеся проблемами науки о данных, часто работают независимо от коллективов разработчиков, следуя парадигме, изображенной на рис. 1.1. Согласно этой схеме, исследователи данных выдают бизнес-аналитику для старшего руководства, которое информирует об изменениях технических руководителей или руководителей проектов; в конечном итоге эти изменения передаются разработчикам для реализации.

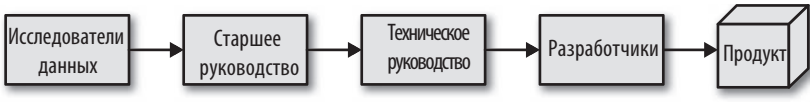


Рис. 1.1. Текущая парадигма науки о данных

Такая структура вполне может удовлетворять потребности некоторых организаций, но в целом она не особенно эффективна. Если бы исследователи данных изначально входили в состав команды разработчиков, как показано на рис. 1.2, продукт совершенствовался бы намного быстрее, а компания имела бы дополнительные конкурентные преимущества. В мире не так много компаний, которые могут позволить себе создавать что-то дважды! Что еще более важно, усилия, прикладываемые исследователями данных, нацелены на пользователей, и поэтому необходимо использовать циклический подход, по аналогии с разработкой пользовательских интерфейсов.

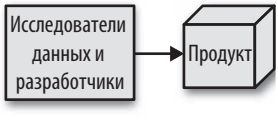


Рис. 1.2. Более удачная парадигма внедрения науки о данных в разработку

Одним из препятствий на пути парадигмы тесного сотрудничества исследователей данных с разработчиками является неразвитость прикладной стороны науки о данных. Большинство публикаций о машинном обучении и обработке естественного языка носят исследовательский характер и не подходят для разработки приложений. Например, несмотря на наличие большого количества превосходных инструментов для машинного обучения на текстовых данных, имеющиеся источники информации — документация, руководства и статьи в блогах — как правило, опираются на искусственно подобранные массивы данных, исследовательские инструменты и экспериментальный код. Лишь в немногих источниках объясняется, например, как создать корпус, достаточно большой для поддержки приложения, как управлять его размером и структурой или как преобразовывать исходные документы в данные, пригодные для использования. Но именно эти аспекты являются важнейшей частью практики создания масштабируемых приложений данных, основанных на анализе естественного языка.

Настоящая книга призвана восполнить этот пробел, продемонстрировав практический подход к анализу текста, ориентированный на разработчиков. Здесь мы покажем, как использовать доступные технологии с открытым исходным кодом для создания модульных, легко тестируемых, гибко настраиваемых и масштабируемых приложений данных. Мы надеемся, что эти инструменты и практические приемы, представленные в книге, помогут исследователям в создании приложений данных нового поколения.

Эта глава послужит основой для следующих глав, более близких к практике программирования. В ней определяются рамки того, что мы называем приложениями данных, основанными на анализе естественного языка, и рассказывается, как проводить их в реальном мире. Затем мы обсудим архитектурные шаблоны проектирования приложений для анализа текста. Наконец, мы рассмотрим особенности языка, которые можно использовать для вычислительного моделирования.

Приложения данных, основанные на анализе естественного языка

Исследователи данных создают приложения данных. Приложения данных извлекают ценные сведения из исходных данных и, в свою очередь, генерируют новые данные[1]. Как нам представляется, целью прикладного анализа текста

[1] Mike Loukides, *What is data science?* (2010), https://oreil.ly/2GJBEoj

является создание «приложений данных, основанных на анализе естественного языка», — пользовательских приложений, не только реагирующих на ввод данных человеком и способных адаптироваться к изменениям, но также предельно точных и имеющих относительно простую организацию. Эти приложения принимают на входе текстовые данные, разбирают их на составные части, выполняют вычисления на основе этих частей и вновь собирают их, возвращая осмысленный и адаптированный результат.

Одним из наших любимых примеров этого является приложение Yelpy Insights для фильтрации отзывов, использующее комбинацию из анализа эмоций, значимых словосочетаний (слов, часто появляющихся вместе) и методов поиска, чтобы определить, насколько тот или иной ресторан соответствует вашим вкусам и диетическим ограничениям. Это приложение использует богатый специализированный корпус и представляет результаты в простом и понятном виде, помогая пользователям выбрать ресторан. Благодаря автоматической идентификации значимых предложений в отзывах и подсветке терминов, приложение позволяет посетителям ресторанов быстро «переварить» большой объем текста и решить, куда пойти ужинать. Несмотря на то что анализ естественного языка не является основной задачей Yelp, влияние этой функции на опыт пользователей неоспоримо. С момента появления Yelpy Insights в 2012 г. Yelp неуклонно внедряет новые возможности, основанные на анализе естественного языка, и за эти годы прибыль компании увеличилась в 6,5 раза[1].

Другим простым примером внедрения анализа естественного языка с потрясающим эффектом может служить поддержка «тегов рекомендаций», реализованная в информационных продуктах таких компаний, как Stack Overflow, Netflix, Amazon, YouTube и других. Теги — это метаинформация о фрагментах контента, важная для поиска и рекомендаций, и они играют важную роль в определении контента, который заинтересует конкретного пользователя. Теги определяют свойства описываемого ими контента и могут использоваться для группировки схожих элементов и предложения описательных названий для таких групп.

Существует много, очень много других примеров. Reverb предлагает персонализированный агрегатор новостей, обученный на словаре Wordnik. Чат-бот Slack поддерживает общение с автоматическим определением контекста. Google Smart Reply может составлять варианты ответов на полученные вами электронные письма. Textra, iMessage и другие инструменты обмена мгновенными сообщениями пытаются предсказать, что вы напечатаете, опираясь на недавно введенный текст, а функция автоматического корректора исправит ваши орфографические ошибки. Имеется также несколько новых голосовых вирту-

[1] Market Watch (2018), https://on.mktw.net/2suTk24

альных помощников — Алекса, Сири, Google Assistant и Кортана, — способных анализировать речь и давать (обычно) более или менее осмысленные ответы.

При чем здесь речевые данные? Действительно, эта книга посвящена анализу текста, а не анализу аудиоданных или речи, однако в речевом анализе аудиоданные обычно сначала преобразуются в текст, к которому затем применяются различные виды анализа, в том числе и описанные в этой книге. Само преобразование речи в текст — это задача машинного обучения, которая также получает все большее распространение!

Упомянутые особенности подсвечивают базовую методологию приложений, основанных на анализе естественного языка: кластеризацию схожего текста в значимые группы или классификацию текста с применением конкретных меток. Иначе говоря — машинное обучение без учителя и с учителем.

В следующем разделе мы познакомимся с некоторыми архитектурными шаблонами проектирования, поддерживающими жизненный цикл модели машинного обучения.

Конвейер приложения данных

Стандартный конвейер типичного приложения данных, изображенный на рис. 1.3, реализует итеративный процесс, состоящий из двух этапов: сборки и развертывания — и является отражением конвейера машинного обучения[1]. На этапе сборки исходные данные преобразуются в форму, пригодную для передачи в модели и экспериментов с ними. На этапе развертывания происходит выбор моделей, которые затем используются для оценок и прогнозов, непосредственно затрагивающих пользователя.

Пользователи отвечают на выходные данные моделей, создавая обратную связь, которая, в свою очередь, подается на вход и используется для адаптации моделей. Четыре стадии — взаимодействие, данные, накопление и вычисления (обработка) — описывают архитектурные компоненты каждого этапа. Например, в процессе взаимодействия на этапе сборки необходима утилита для ввода данных, а пользователю — некоторый прикладной интерфейс. Под стадией данных обычно подразумеваются внутренние компоненты, которые действуют как передаточное звено для перехода к стадии накопления, функции которой обычно выполняет база данных. Стадия вычислений (обработки) может принимать самые разные формы, от простых SQL-запросов до блокнотов Jupyter и даже вычислительных кластеров под управлением Spark.

[1] Benjamin Bengfort, *The Age of the Data Product* (2015), http://bit.ly/2GJBEEP

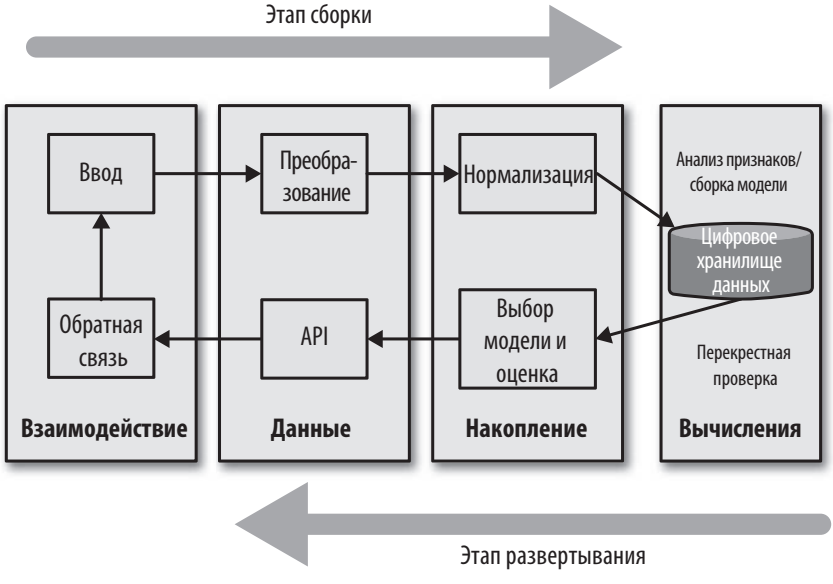


Рис. 1.3. Конвейер приложения данных

Этап развертывания, помимо выбора и использования модели, мало чем отличается от более прямолинейной разработки программного обеспечения. Часто приложения данных реализуют API, который используется другими приложениями, службами или пользовательскими интерфейсами. Однако этап сборки приложений данных требует особого внимания — тем более в случае анализа текста. Реализуя приложения данных на основе анализа естественного языка, мы создаем дополнительные лексические ресурсы и артефакты (такие как словари, переводчики, регулярные выражения и т. д.), от которых зависит работа приложения.

На рис. 1.4 показано более развернутое представление этапа сборки для надежных приложений машинного обучения на основе анализа естественного языка. Процесс перехода от исходных данных к развернутой модели по сути состоит из последовательности инкрементальных преобразований данных. Во-первых, исходные данные преобразуются во входной корпус, накапливаются и сохраняются внутри хранилища данных. Затем входные данные группируются, очищаются, нормализуются и преобразуются в векторы для последующей обработки. В конечном преобразовании модель (или модели) обучается на векторизованном корпусе, и создается обобщенное представление исходных данных, которое потом используется приложением.

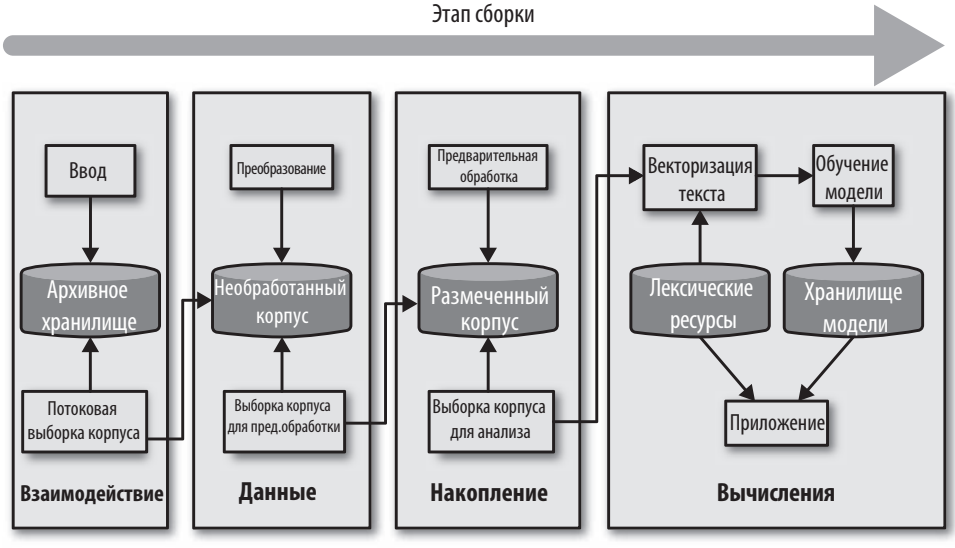


Рис. 1.4. Приложения данных на основе анализа естественного языка

Тройка выбора модели

Архитектура продуктов машинного обучения должна поддерживать и оптимизировать преобразования данных, обеспечивая простоту их тестирования и настройки, что является ее отличительной особенностью. По мере развития приложений данных растет заинтересованность в универсальном определении процесса машинного обучения с ускоренной — или даже автоматической — сборкой моделей. К сожалению, из-за больших размеров пространства поиска автоматических методов оптимизации недостаточно.

Выбор оптимальной модели — сложный и итеративный процесс, заключающийся в повторении цикла, состоящего из конструирования признаков, выбора модели и настройки гиперпараметров. После каждой итерации выполняется оценка результатов с целью получить лучшую комбинацию признаков, модели и параметров, которая решает поставленную задачу. Этот процесс, изображенный на рис. 1.5, мы называем *тройкой выбора модели*[1]. Его цель — представить итерацию как основу науки машинного обучения, которая должна облегчаться, а не ограничиваться.

[1] Arun Kumar, Robert McCann, Jeffrey Naughton, and Jignesh M. Patel, *Model Selection Management Systems: The Next Frontier of Advanced Analytics* (2015), http://bit.ly/2GOFa0G

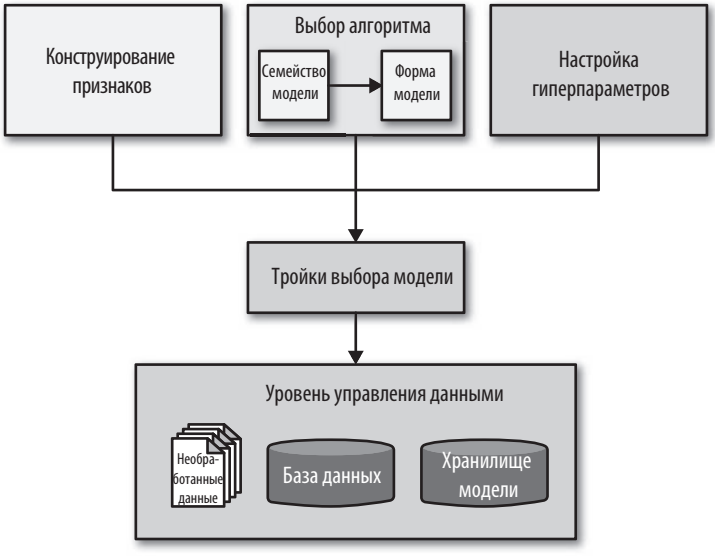


Рис. 1.5. Тройка выбора модели

В своей статье, вышедшей в 2015 году, Викхем (Wickham) с соавторами[1] искусно устраняют неоднозначность перегруженного термина «модель», описав три основных случая его использования в статистическом машинном обучении: семейство модели, форма модели и обученная модель. Широкое понятие семейства моделей определяет отношения между переменными и интересующей целью (например, «линейная модель» или «рекуррентная тензорная нейронная сеть»). Форма модели — конкретный экземпляр тройки выбора модели: набор признаков, алгоритм и конкретные гиперпараметры. Наконец, обученная модель — это форма модели, обученная на конкретном наборе данных и пригодная для получения предсказаний. Приложения данных состоят из множества обученных моделей, сконструированных в процессе их выбора, что создает и оценивает формы моделей.

Мы не привыкли думать о языке как о данных, поэтому основной задачей анализа текста является интерпретация происходящего в каждом из этих преобразований. С каждым последующим преобразованием текст становится все менее значимым для нас, потому что все меньше напоминает естественный язык. Чтобы добиться большей эффективности в конструировании приложений данных, основанных на анализе естественного языка, мы должны изменить свой взгляд на язык.

---

[1] Hadley Wickham, Dianne Cook, and Heike Hofmann, *Visualizing Statistical Models: Removing the Blindfold* (2015), http://bit.ly/2JHq92J

В оставшейся части этой главы мы покажем способы рассуждения о языке как о данных, которые можно подвергнуть компьютерной обработке. Попутно мы составим небольшой словарь, который поможет нам сформулировать виды преобразований текстовых данных, описанных в следующих главах.

Язык как данные

Язык — это *неструктурированные* данные, которые используются людьми для общения между собой. *Структурированные* или *полуструктурированные* данные, в свою очередь, включают поля или разметку, позволяющие компьютеру анализировать их. Но, несмотря на отсутствие машиночитаемой структуры, неструктурированные данные не являются случайными. Напротив, они подчиняются лингвистическим правилам, которые делают эти данные понятными для людей.

Методы машинного обучения, и особенно обучения с учителем, в настоящее время являются наиболее изученными и перспективными инструментами обработки естественного языка. Машинное обучение позволяет обучать (и переобучать) статистические модели по мере изменения языка. Создавая модели на контекстно-зависимых корпусах, приложения могут использовать узкие смысловые окна для увеличения точности без необходимости прибегать к дополнительной интерпретации. Например, для создания приложения, читающего медицинские карты и генерирующего рекомендации по лечению, требуется совершенно иная модель, чем для приложения, отбирающего новости с учетом личных предпочтений пользователя.

Компьютерная модель языка

Специалисты по обработке и анализу данных создают приложения данных, основанные на анализе естественного языка, поэтому наша первейшая задача — создать модель, описывающую язык и способную делать выводы на основе этого описания.

Формально модель языка должна принимать на входе неполную фразу и дополнять ее недостающими словами, наиболее вероятными для завершения высказывания. Этот тип моделей языка сильно влияет на аналитическую обработку текста, потому что демонстрирует основной механизм приложений обработки языка — использование контекста для угадывания смысла. Модели языка также раскрывают базовую гипотезу машинного обучения на текстах: *текст предсказуем*. Фактически, механизм оценки моделей в академическом контексте, *связность*, измеряет предсказуемость текста вычислением *энтропии*

(степень неопределенности или неожиданности) распределения вероятностей модели языка.

Рассмотрим следующие незаконченные фразы: «собака — друг ...» и «ведьма летит на ...». Эти фразы имеют низкую энтропию, и модели языка с высокой степенью вероятности будут угадывать продолжения «человека» и «метле» соответственно (более того, мы удивились бы, если бы эти фразы завершались как-то иначе). С другой стороны, фразы с высокой энтропией, такие как «сегодня я собираюсь поужинать с ...», предполагают множество вариантов продолжения («другом», «родителями» и «коллегами» выглядят одинаково вероятными). Человек, услышавший такую фразу, может использовать свой опыт, воображение и память, а также ситуационный контекст, чтобы восполнить пробел. Компьютерные модели не обязательно имеют одинаковый контекст и в результате оказываются более ограниченными.

Модели языка способны выводить или определять отношения между лексемами, которые модели наблюдают как строки данных в кодировке UTF-8, а люди распознают как слова с определенным смыслом. Формально, модель использует контекст для определения узкого пространства решений, в котором есть небольшое количество вариантов.

Это понимание дает нам возможность генерализировать формальную модель в другие модели языка, работающие, например, в приложениях машинного перевода или анализа настроений. Чтобы воспользоваться предсказуемостью текста, мы должны определить ограниченное числовое пространство решений, на котором может действовать модель. Благодаря этому мы можем использовать методы статистического машинного обучения, с учителем и без учителя, для построения моделей языка, извлекающих смысл из данных.

На первом шаге в машинном обучении выявляются характерные признаки данных, помогающие предсказать цель. Текстовые данные предоставляют массу возможностей для извлечения поверхностных признаков, простым разбиением строк, или более глубоких, позволяющих извлекать из текста морфологические, синтаксические и даже семантические представления.

В следующих разделах мы рассмотрим несколько простых способов извлечения сложных признаков из данных на естественном языке для целей моделирования. Сначала мы посмотрим, как лингвистические свойства языка (например, род в английском языке) могут дать нам возможность выполнить статистическую обработку текста. Затем мы подробно рассмотрим, как контекст меняет интерпретацию и как это обычно используется для создания традиционной модели «мешок слов». Наконец, мы исследуем богатые возможности анализа

с применением морфологической, синтаксической и семантической обработки естественного языка.

Лингвистические признаки

Рассмотрим простую модель, использующую лингвистические признаки для выявления преобладающего рода во фрагменте текста. В 2013 г. Нил Карен (Neal Caren), доцент кафедры социологии в Университете города Чаппел-Хилл (штат Северная Каролина), опубликовал в блоге статью[1], где исследовал роль половой принадлежности в новостях для определения проявлений мужчин и женщин в разных контекстах. Он применил гендерный анализ к текстам статей, опубликованных в *New York Times*, и выяснил, что слова в мужском и женском роде появляются в совершенно разных контекстах, что способно усиливать гендерные предубеждения.

Что особенно интересно, в этом анализе слова в женском и мужском роде использовались для создания частотной оценки мужских и женских признаков. Реализацию подобного анализа на Python можно начать с определения наборов слов, различающих предложения о мужчинах и женщинах. Для простоты допустим, что всякое предложение может классифицироваться как рассказывающее о мужчинах, о женщинах, о мужчинах и женщинах и *неизвестно о ком* (потому, что предложение может рассказывать о чем-то другом, не о мужчинах и не о женщинах, а также потому, что наши множества слов `MALE_WORDS` и `FEMALE_WORDS` не являются исчерпывающими):

```
MALE = 'male'
FEMALE = 'female'
UNKNOWN = 'unknown'
BOTH = 'both'

MALE_WORDS = set([
    'guy','spokesman','chairman',"men's",'men','him',"he's",'his',
    'boy','boyfriend','boyfriends','boys','brother','brothers','dad',
    'dads','dude','father','fathers','fiance','gentleman','gentlemen',
    'god','grandfather','grandpa','grandson','groom','he','himself',
    'husband','husbands','king','male','man','mr','nephew','nephews',
    'priest','prince','son','sons','uncle','uncles','waiter','widower',
    'widowers'
])

FEMALE_WORDS = set([
    'heroine','spokeswoman','chairwoman',"women's",'actress','women',
    "she's",'her','aunt','aunts','bride','daughter','daughters','female',
```

[1] Neal Caren, *Using Python to see how the Times writes about men and women* (2013), http://bit.ly/2GJBGfV

```
    'fiancee','girl','girlfriend','girlfriends','girls','goddess',
    'granddaughter','grandma','grandmother','herself','ladies',
    'lady','mom','moms','mother','mothers','mrs','ms','niece','nieces',
    'priestess','princess','queens','she','sister','sisters','waitress',
    'widow','widows','wife','wives','woman'
])
```

Теперь, когда у нас есть множества слов-признаков, характеризующих половую принадлежность, нам нужен метод определения гендерного класса предложения; создадим функцию `genderize`, которая подсчитывает количество слов в предложении, попадающих в наши списки `MALE_WORDS` и `FEMALE_WORDS`. Если предложение содержит только слова из `MALE_WORDS`, оно классифицируется как *мужское*. Предложение, содержащее только слова из `FEMALE_WORDS`, классифицируется как *женское*. Если предложение содержит мужские и женские слова, отнесем его к категории *двуполых*; а если в нем нет ни мужских, ни женских слов, определим его как имеющее *неизвестный* род:

```
def genderize(words):

    mwlen = len(MALE_WORDS.intersection(words))
    fwlen = len(FEMALE_WORDS.intersection(words))

    if mwlen > 0 and fwlen == 0:
        return MALE
    elif mwlen == 0 and fwlen > 0:
        return FEMALE
    elif mwlen > 0 and fwlen > 0:
        return BOTH
    else:
        return UNKNOWN
```

Нам также нужно подсчитать частоту слов, признаков рода и предложений во всем тексте статьи. Для этой цели можно использовать встроенный в Python класс `collections.Counters`. Функция `count_gender` принимает список предложений и использует функцию `genderize` для определения общего количества слов-признаков и классификации предложений. Для каждого предложения определяется его класс, а все слова в предложении считаются принадлежащими к этой категории:

```
from collections import Counter

def count_gender(sentences):

    sents = Counter()
    words = Counter()

    for sentence in sentences:
```

```
        gender = genderize(sentence)
        sents[gender] + = 1
        words[gender] + = len(sentence)

    return sents, words
```

Наконец, чтобы задействовать функции определения гендерной принадлежности, нам нужен некоторый механизм, преобразующий исходный текст статей в составляющие его предложения и слова. Используем библиотеку NLTK (которую мы рассмотрим далее в этой главе и в следующей), чтобы разбить абзацы на предложения. Выделив отдельные предложения, мы сможем разбить их на лексемы, чтобы выявить отдельные слова и знаки пунктуации, и передать размеченный текст нашим функциям классификации для вывода процентов предложений и слов, относящихся к категории *мужских*, *женских*, *двуполых* и *неизвестной* принадлежности:

```
import nltk

def parse_gender(text):

    sentences = [
        [word.lower() for word in nltk.word_tokenize(sentence)]
        for sentence in nltk.sent_tokenize(text)
    ]

    sents, words = count_gender(sentences)
    total = sum(words.values())

    for gender, count in words.items():
        pcent = (count / total) * 100
        nsents = sents[gender]

        print(
            "{0.3f}% {} ({} sentences)".format(pcent, gender, nsents)
        )
```

Применив нашу функцию `parse_gender` к статье из *New York Times* под заголовком «Rehearse, Ice Feet, Repeat: The Life of a New York City Ballet Corps Dancer»[1], мы получили следующие неудивительные результаты:

```
50.288% female (37 sentences)
42.016% unknown (49 sentences)
4.403% both (2 sentences)
3.292% male (3 sentences)
```

[1] «Репетиция, ледяной компресс на ноги, и все сначала: жизнь танцовщицы Нью-Йоркского городского театра балета». — *Примеч. пер.*

Функция оценки определяет длину предложений по количеству слов в них. Поэтому, даже при том, что чисто женских предложений меньше, чем неизвестной принадлежности, чуть более 50 % содержимого статьи относится к женской категории. В качестве расширения этого метода можно проанализировать слова в женских и мужских предложениях, чтобы посмотреть — имеются ли какие-то дополнительные термины, по умолчанию связанные с мужским и женским полом. Как видите, этот вид анализа относительно просто реализуется на Python, и Карену эти результаты показались поразительными:

> *Если бы знания о роли мужчин и женщин в обществе вы получили только из номеров New York Times за последнюю неделю, то могли бы подумать, что мужчины занимаются спортом или работают в правительстве, а женщины занимаются исключительно женскими и домашними делами. Честно говоря, я был шокирован, насколько стереотипные слова использовались в женских предложениях.*
>
> *— Нил Карен (Neal Caren)*

Что же здесь происходит в действительности? Этот механизм, хотя и детерминированный, очень хорошо демонстрирует, как слова (пусть и стереотипные) способствуют предсказуемости в контексте. Однако этот механизм работает именно потому, что признак половой принадлежности встроен непосредственно в язык. В других языках (например, французском) гендерный признак выражен еще сильнее: идеи, неодушевленные предметы и даже части тела могут иметь пол (даже если это противоречит здравому смыслу). Языковые особенности не всегда имеют определительный смысл, часто они несут другую информацию; например, множественное число и время — еще два языковых признака — теоретически можно использовать для определения прошлого, настоящего и будущего языка. Однако особенности языка составляют лишь часть уравнения, когда речь заходит о предсказании смысла текста.

Контекстные признаки

Анализ настроения, о котором мы подробнее поговорим в главе 12, является очень популярным методом классификации текста, потому что с помощью эмоциональной окраски можно передать массу информации о точке зрения на обсуждаемый предмет; он помогает провести комплексный анализ отзывов, полярности сообщений или реакций. Можно подумать, что анализ настроения легко провести с помощью приема, похожего на использовавшийся в гендерном анализе, представленном в предыдущем разделе: собрать списки слов с положительной («превосходно», «хорошо», «колоссально») и отрицательной окраской

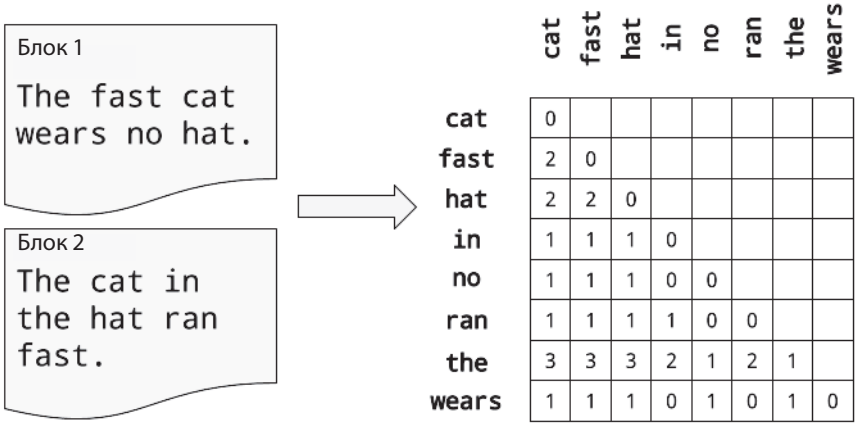
(«ужасно», «безвкусно», «примитивный») и определить относительные частоты встречаемости этих лексем в контексте. К сожалению, это слишком упрощенный подход, и часто он дает очень неточные результаты.

Анализ настроений коренным образом отличается от гендерной классификации, потому что настроение не является особенностью языка, а зависит от *смысла слов*; например, фраза «трюк получился необычным» носит положительный оттенок, тогда как «съев суп, я почувствовал себя необычно плохо» — отрицательный, а «у меня живет необычная игуана» — неоднозначный. Значение слова «необычный» в этих фразах меняется. Кроме того, настроение зависит от контекста, даже когда определение слова остается неизменным; слово «мягкий» может нести отрицательный оттенок в разговоре о жгучем перце, но положительный в описании сиропа от кашля. Наконец, в отличие от пола или времени, настроение можно отрицать: «нехорошо» означает плохо. Отрицание может перевернуть смысл большого фрагмента положительного текста: «У меня были большие надежды и ожидания на фильм, который критики назвали замечательным и волнующим, но, посмотрев его, я разочаровался». Здесь большое количество слов, обычно указывающих на положительный настрой, таких как «большие надежды», «ожидания», «замечательный и волнующий», не только не смогли преуменьшить негативный оттенок единственного слова «разочаровался», но даже усилили его.

Тем не менее все эти примеры предсказуемы; положительное и отрицательное отношение передается ясно, и похоже, что модель машинного обучения сможет определить настроение и, может быть, даже отделит неважные или неоднозначные высказывания. Априори детерминированный или структурный подход утрачивает гибкость контекста и смысла, поэтому большинство моделей языка учитывают также расположение слов в контексте, используя методы машинного обучения для создания прогнозов.

На рис. 1.6 изображен основной метод разработки простых моделей языка, которые часто называют моделями вида «мешок слов». Эти модели оценивают частоту встречаемости слов друг с другом и с другими словами в узком, ограниченном контексте. Подобная оценка помогает определить, какие слова будут следовать друг за другом, и по небольшим фрагментам текста определять их смысл. Далее с помощью методов статистических выводов можно прогнозировать порядок слов.

Расширения модели «мешок слов» рассматривают совместные вхождения не только отдельных слов, но также целых фраз, имеющих высокую значимость для определения смысла. Фраза «снять деньги в банке» придает определенный смысл слову «банк», но то же самое делает фраза «квасить капусту в стеклянной



	cat	fast	hat	in	no	ran	the	wears
cat	0							
fast	2	0						
hat	2	2	0					
in	1	1	1	0				
no	1	1	1	0	0			
ran	1	1	1	1	0	0		
the	3	3	3	2	1	2	1	
wears	1	1	1	0	1	0	1	0

Рис. 1.6. Матрица частот совместного появления слов

банке». Это называется *анализом n-грамм*, где *n* определяет упорядоченную последовательность символов или слов для сканирования (например, 3-грамма `('снять', 'деньги', 'в')` или 4-грамма `('снять, 'деньги', 'в', 'банке')`). *n*-граммы открывают интересные возможности, потому что подавляющее большинство *n*-грамм лишено смысла (например, `('ведро', 'прыгать', 'фейерверк')`), хотя, учитывая развивающуюся природу языка, даже эта 3-грамма может когда-нибудь стать осмысленной! Соответственно, модели языка, использующие преимущества контекста таким способом, требуют наличия возможности изучать отношение текста с некоторой целевой переменной.

Использование языковых и контекстных признаков в анализе способствует общей предсказуемости языка. Но для их выявления требуется умение разбирать и определять единицы языка. В следующем разделе мы обсудим увязку языковых и контекстных признаков в смысловое значение с лингвистической точки зрения.

Структурные признаки

Наконец, модели языка и аналитическая обработка текста получили дополнительные выгоды от развития компьютерной лингвистики. Какие бы модели мы ни строили, с контекстными или лингвистическими признаками (или обоими), необходимо также учитывать высокоуровневые единицы языка, используемые лингвистами, чтобы определить словарь операций, которые мы будем применять к корпусам нашего текста в последующих главах. На разных уровнях

обработке подвергаются разные единицы языка, и понимание лингвистического контекста имеет большое значение для понимания приемов обработки естественного языка, используемых в машинном обучении.

Под термином *семантика* подразумевается смысл. Семантика глубоко вложена в язык и с большим трудом поддается извлечению. В общем случае высказывания (простая фраза, а не целый абзац, например: «Она взяла книгу из библиотеки») следуют простому шаблону: субъект, глагол, объект и определение, относящееся к объекту (подлежащее — сказуемое — дополнение). Используя такие шаблоны, можно конструировать *онтологии*, определяющие конкретные отношения между сущностями, но такая работа требует весомых знаний контекста и предметной области и плохо масштабируется. Тем не менее в последнее время ведутся многообещающие работы по извлечению онтологий из таких источников, как «Википедия» или DBPedia (например, статья с определением слова «библиотека» в DBPedia начинается со слов: «Библиотека (греч. βιβλίον, книга + θήκη, хранилище, вместилище, ящик) — учреждение, собирающее и хранящее произведения печати и письменности для общественного пользования»).

Семантический анализ заключается не только в определении смысла текста, но также в создании структур данных, к которым могут применяться логические рассуждения. Текстовые смысловые представления (или тематические смысловые представления — Thematic Meaning Representations, TMR) можно использовать для кодирования предложений в виде структур предикатов, к которым могут применяться логика первого порядка или *лямбда-исчисление*. Другие структуры, такие как сети, можно использовать для кодирования взаимодействий предикатов интересующих признаков в тексте. Затем можно выполнить обход для анализа центральности терминов или субъектов и причин отношений между элементами. Хотя *анализ графов* не всегда является полным семантическим анализом, иногда он помогает прийти к важным выводам.

Синтаксис — это свод правил формирования предложений; обычно он определяется грамматикой. Предложения — это единицы языка, которые мы используем для создания смысла; они кодируют намного больше информации, чем отдельные слова. По этой причине мы будем считать предложения наименьшей логической единицей языка. Цель синтаксического анализа — показать значимые связи между словами, обычно путем деления предложения на части, или связи между лексемами в древовидной структуре (подобно синтаксическому разбору предложений, который вы могли делать в школе). Синтаксис является обязательной основой для рассуждений о системе понятий или семантике, потому что это жизненно важный инструмент для понимания того, как слова влияют друг на друга при формировании *фраз*. Например, синтаксический

анализ должен выявить предложную фразу «из библиотеки» и именную фразу «книгу из библиотеки» как компоненты глагольной фразы «взяла книгу из библиотеки».

Морфология определяет форму вещей, а в анализе текста — форму отдельных слов или лексем. Структура слов может помочь нам выявить множественное число (*жена* и *жёны*), род (*поэт* и *поэтесса*), время (*бежал* и *бегу*), спряжение по лицам (*я бегу* и *он бежит*), и т. д. Анализ морфологии — сложная задача, потому что в большинстве языков имеется масса исключений из правил и специальных случаев. В английском языке, например, имеются как орфографические правила, просто определяющие окончание слов (*puppy — puppies*)[1], так и морфологические — определяющие полные эквиваленты (*goose — geese*)[2]. Английский — аффиксальный язык, то есть в нем к слову просто добавляются дополнительные символы, в начало или в конец, для изменения его формы. В других языках действуют иные морфологические правила: в иврите используются шаблоны из согласных, в которые вставляются гласные для создания смысла, тогда как в китайском языке используются пиктографические символы, которые не всегда изменяются непосредственно.

Главная задача морфологического анализа — выявить части слов, по которым эти слова можно отнести к тем или иным классам, которые часто называют тегами частей речи. Например, иногда нам важно знать, находится ли существительное в единственном или во множественном числе, или же является именем собственным. Также нам может потребоваться узнать, имеет ли глагол неопределенную форму, прошедшее время, или это деепричастие. Затем полученные части речи используются для создания более крупных структур, таких как фрагменты или фразы, или даже целые древа слов, которые затем можно использовать для построения структур данных семантических рассуждений.

Семантика, синтаксис и морфология позволяют добавлять данные в простые текстовые строки с лингвистическим смыслом. В главе 3 мы посмотрим, как разбить текст на логические и смысловые единицы, используя методы выделения лексем и сегментов, а также как назначить теги частей речи. В главе 4 мы применим прием векторизации к этим структурам для создания числовых пространств признаков — например, нормализацию текста с применением методов стемминга и лемматизации для уменьшения количества признаков. Наконец, в главе 7 мы напрямую будем использовать структуры для кодирования информации в наши протоколы машинного обучения с целью улучшения производительности и специализации для конкретных видов анализа.

---

[1] Щенок — щенки. — *Примеч. пер.*

[2] Гусь — гуси. — *Примеч. пер.*

В заключение

Естественный язык — одна из самых мало используемых форм данных, доступных в настоящее время. Его анализ позволяет увеличить полезность приложений данных и сделать их неотъемлемой частью нашей жизни. Специалисты по обработке и анализу данных обладают уникальными возможностями для создания таких приложений данных, основанных на анализе естественного языка. Объединяя текстовые данные с машинным обучением, они способны создавать мощные приложения в мире, где информация часто приравнивается к ценностям, а обладание ею дает конкурентные преимущества. Современная жизнь основана на источниках данных из естественного языка, и приложения данных, основанные на его анализе, делают эти данные более доступными.

В следующих нескольких главах мы обсудим некоторые обязательные этапы обработки текста, предшествующие машинному обучению, а именно: управление корпусом (в главе 2), предварительную обработку (в главе 3) и векторизацию (в главе 4). Затем мы поэкспериментируем с формулировкой задач машинного обучения для классификации (в главе 5) и кластеризации (в главе 6).

В главе 7 мы реализуем извлечение признаков для максимизации эффективности наших моделей, а в главе 8 посмотрим, какие приемы визуализации нам доступны для отображения результатов и выявления ошибок моделирования. В главе 9 мы рассмотрим другой подход к моделированию языка, использовав граф для представления слов и отношений между ними. После этого, в главе 10, мы исследуем более специализированные методы поиска, извлечения и генерации для чат-ботов. Наконец, в главах 11 и 12 мы рассмотрим методы масштабирования вычислительной мощности с применением Spark и модель масштабирования сложности с применением искусственных нейронных сетей.

Как вы увидите в следующей главе, для масштабируемого анализа и машинного обучения на тексте нам в первую очередь необходимы знания в предметной области и корпус текста, характерный для этой области. Например, если вы работаете в финансовой сфере, ваше приложение должно распознавать биржевые сокращения, финансовые термины и названия компаний. А это значит, что документы в создаваемом вами корпусе должны содержать эти сущности. Иначе говоря, разработка приложения данных на основе анализа естественного языка начинается с получения текстовых данных подходящего типа и создания корпуса, содержащего структурные и контекстные признаки предметной области, в которой вы работаете.

2 Создание собственного корпуса

Главной задачей любого приложения машинного обучения является определение того, что считать полезным сигналом и как выделить его из информационного шума. С этой целью выполняется *анализ признаков*, в ходе которого устанавливается, какие признаки, свойства или размерности в тексте лучше всего передают его смысл и базовую структуру. В предыдущей главе мы увидели, что, несмотря на сложность и гибкость естественного языка, его можно моделировать, извлекая структурные и контекстные признаки.

В последующих главах основное внимание будет уделяться «выделению признаков» и «разработке системы знаний» — где мы займемся идентификацией уникальных словарных слов, наборов синонимов, взаимосвязей между сущностями и семантических контекстов. На протяжении всей книги вы будете видеть, что представление базовой лингвистической структуры во многом определяет успех нашей работы. Определение представления требует от нас определить единицы языка — элементы, которые можно посчитать, измерить, проанализировать или изучить.

Анализ текста в некоторой степени — это разбиение больших фрагментов текста на составляющие их компоненты — уникальные слова, общие фразы, синтаксические шаблоны — с последующим применением к ним статистических механизмов. Изучая эти компоненты, можно создавать модели языка, позволяющие снабжать приложения возможностями прогнозирования. Очень скоро мы увидим, что существует множество уровней, на которых мы можем применить свой анализ, и все они связаны с одним центральным набором текстовых данных: *корпусом*.

Что такое корпус?

Корпус — это коллекция взаимосвязанных документов (текстов) на естественном языке. Корпус может быть большим или маленьким, но обычно состоит из десятков и даже сотен гигабайт данных в тысячах документов. Например, учитывая средний объем почтового ящика в 2 Гбайт (для справки, полная версия корпуса переписки компании Enron, которому сейчас примерно 15 лет, включает 1 миллион электронных писем 118 пользователей и имеет размер[1] 160 Гбайт), компания небольшого размера, насчитывающая 200 сотрудников, способна сгенерировать корпус переписки размером в полтерабайта. Корпусы могут быть *аннотированными*, то есть текст или документы могут быть снабжены специальными метками для алгоритмов обучения с учителем (например, для фильтров спама), или *неаннотированными*, что делает их кандидатами на тематическое моделирование и кластеризацию документов (например, для изучения изменений в темах, скрытых в сообщениях, с течением времени).

Корпус можно разбить на категории документов или на отдельные документы. Документы в корпусе могут различаться размерами, от отдельных сообщений в «Твиттере» до целых книг, но все они содержат текст (а иногда метаданные) и представляют набор связанных тем. Документы, в свою очередь, можно разбить на абзацы — *смысловые единицы* речи (units of discourse), которые обычно выражают одну идею. Абзацы также можно разбить на предложения — *синтаксические* единицы; законченное предложение структурно звучит как конкретное выражение. Предложения состоят из слов и знаков препинания — *лексических* единиц, которые определяют общий смысл, но гораздо более полезных в сочетаниях. Наконец, сами слова состоят из слогов, фонем, аффиксов и символов, то есть единиц, имеющих смысл, только когда они объединены в слова.

Предметные корпусы

Очень часто тестирование модели естественного языка начинают с использованием обобщенного корпуса. Например, есть много примеров и научных статей, в которых используются готовые наборы данных, такие как корпус Brown, корпус из «Википедии» или корпус Cornell диалогов из фильмов. Однако наиболее удачные модели языка часто являются узкоспециализированными для конкретного применения.

Почему модели языка, обученные на ограниченной предметной области, часто действуют лучше, чем такие же модели, но обученные на обобщенном

[1] Federal Energy Regulatory Committee, FERC Enron Dataset. http://bit.ly/2JJTOIv

корпусе? В разных предметных областях используется разный язык (словарь, сокращения, типичные фразы и т. д.), поэтому корпус, специализированный для конкретной области, анализируется и моделируется точнее, чем корпус с документами из разных областей.

Возьмем для примера термин «bank» (банк). В области экономики, финансов или политики этим термином обозначается организация, производящая фискальные и монетарные инструменты, тогда как в авиации и дорожном движении термин «bank» (крен) обозначает форму движения, которая изменяет направление воздушного судна или транспортного средства. Благодаря обучению на более узком контексте, пространство предсказаний модели становится более узким и специализированным, а значит, такая модель лучше справляется с гибкостью языка.

Наличие предметного корпуса имеет большое значение для создания удачного приложения данных на основе анализа естественного языка. Естественно, возникает следующий вопрос: как получить набор данных для создания модели языка? Независимо от способа сбора данных — методом скраппинга, извлечением из RSS или посредством некоторого API — конструирование корпуса с исходным текстом в форме, пригодной для создания приложения данных, является нетривиальной задачей.

Часто специалисты по анализу и обработке данных начинают с создания единого статического набора документов и затем применяют определенные виды анализа. Однако независимо от процедуры анализа и приемов получения данных модель в этом случае получится статичной, не способной реагировать на новую обратную связь и не отражающей динамическую природу языка.

Основное внимание в этой главе уделяется не приобретению данных, а их структурированию и управлению способами, поддерживающими машинное обучение. Тем не менее в следующем разделе мы познакомимся с фреймворком Baleen для построения механизмов сбора данных, который, в частности, хорошо подходит для создания предметно-ориентированных корпусов для прикладного анализа текста.

Движок сбора данных Baleen

Baleen[1] — инструмент с открытым исходным кодом для создания своих корпусов. Извлекает данные на естественном языке из текстов, написанных про-

[1] District Data Labs, *Baleen: автоматическая служба для блогов, помогающая создавать корпусы для исследований в области анализа естественного языка* (2014), http://bit.ly/2GOFaxI

фессиональными писателями и любителями, например блогерами и информационными агентствами, в классифицированном виде.

Используя OPML-файлы с описанием RSS-каналов (распространенный формат экспорта для программ чтения новостей), Baleen загружает все сообщения из этих каналов, сохраняет их в базе данных MongoDB, затем экспортирует корпус текста, который можно использовать для анализа. На первый взгляд кажется, что все это легко реализовать в виде одной функции, но в действительности реализация сбора данных может быть очень сложной задачей, потому что программные интерфейсы (API) и каналы RSS часто изменяются. Требуется многое учесть и продумать, чтобы создать приложение, которое не только обеспечивает надежный и автономный сбор данных, но и гарантирует безопасность управления данными.

Сложность процедуры получения текста через каналы RSS наглядно иллюстрирует рис. 2.1. Выбор каналов, откуда должны извлекаться данные, и как их следует классифицировать, определяет файл OPML, хранящийся на диске.

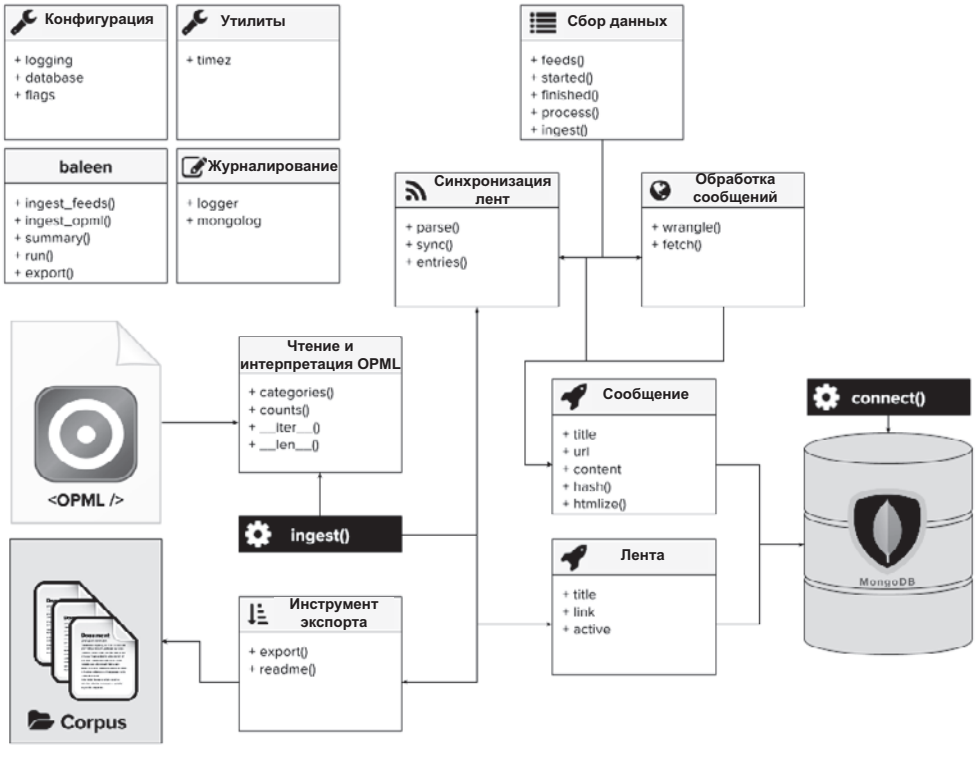


Рис. 2.1. Архитектура движка Baleen сбора данных из каналов RSS

Для подключения к хранилищу MongoDB и записи в него сообщений, лент новостей и другой информации требуется механизм объектно-документного отображения (Object Document Mapping, ODM), также необходимы инструменты для определения одного задания сбора данных, которое синхронизирует все каналы и затем извлекает и обрабатывает отдельные сообщения и статьи.

Используя эти механизмы, Baleen предоставляет утилиты для запуска заданий сбора данных на регулярной основе (например, ежечасно), правда, для подключения к базе данных и настройки частоты запуска требуется определить некоторые конфигурационные параметры. Поскольку сбор данных — длительный процесс, Baleen также предоставляет консоль для настройки расписания, журналирования и проверки ошибок. Наконец, инструмент экспорта в Baleen позволяет вывести корпус в базу данных.

Текущая реализация движка Baleen собирает данные из каналов RSS 12 категорий, включая спорт, игры, политику, кулинарию и новости. Соответственно, Baleen производит не один, а 12 предметных корпусов, образцы которых доступны в репозитории книги на GitHub: https://github.com/foxbook/atap/.

Независимо от того, были ли извлечены документы в процессе сбора данных или они являются частью фиксированной коллекции, необходимо подумать, как управлять данными и подготавливать их для анализа и обучения модели. В следующем разделе мы обсудим, как следить за корпусами по мере продолжающегося сбора данных, изменения данных и увеличения их объема.

Управление корпусом данных

Первое допущение, которое мы должны сделать, — корпусы, которые мы будем использовать, имеют нетривиальные размеры, то есть они будут содержать тысячи и десятки тысяч документов, занимающих гигабайты пространства. Второе допущение — языковые данные, поступающие из источника, необходимо очищать и преобразовывать в структуры данных, пригодные для анализа. Первое допущение требует использования масштабируемых технологий (которые более подробно рассматриваются в главе 11), а второе предполагает выполнение необратимых преобразований данных (как будет показано в главе 3).

Приложения данных часто используют архивные хранилища вида «записал один раз, прочитал много раз» (Write-Once Read-Many, WORM) в качестве промежуточного уровня управления данными между механизмами сбора

и предварительной обработки данных, как показано на рис. 2.2. Архивные хранилища WORM (иногда их называют «озерами данных») поддерживают потоковый доступ для чтения исходных данных повторяемым и масштабируемым способом, стремясь удовлетворить требование высокой производительности. Кроме того, сохраняя данные в хранилище WORM, можно повторно выполнить предварительную обработку без повторного извлечения данных источников, что позволяет легко проверять новые гипотезы в отношении исходных, необработанных данных.

Рис. 2.2. Хранилище WORM поддерживает этап промежуточной обработки

Добавление хранилища WORM в процесс сбора данных означает, что данные должны храниться в двух экземплярах: в виде корпуса исходных, необработанных данных и в виде корпуса предварительно обработанных данных — что влечет резонный вопрос: где должны храниться данные? Размышляя об управлении данными, мы обычно в первую очередь думаем о базах данных. Базы данных, безусловно, ценный инструмент в создании приложений данных, основанных на анализе естественного языка, и многие из них поддерживают полнотекстовый поиск и другие виды индексирования. Но большинство баз данных создаются для извлечения или обновления лишь нескольких записей в рамках одной транзакции. При работе с корпусами текста, напротив, производится чтение документов целиком, при этом не требуется изменять, искать или как-то иначе выбирать отдельные документы. Как результат, широкие возможности баз данных ложатся тяжким бременем на вычисления, не давая никаких выгод.

Системы управления реляционными базами данных прекрасно подходят для одновременного выполнения операций с небольшими коллекциями записей, особенно когда происходит частое изменение записей. Текстовые корпусы в машинном обучении имеют совершенно иной вычислительный профиль: они многократно читаются целиком. Как результат, часто предпочтительнее хранить корпусы на диске (или в документоориентированной базе данных).

Для управления текстовыми данными часто лучше всего использовать документоориентированные базы данных NoSQL, поддерживающие потоковое чтение документов с минимальными накладными расходами, или просто записывать каждый документ на диск. Базы данных NoSQL могут пригодиться в крупных приложениях, но подход на основе обычных файлов имеет свои преимущества: приемы сжатия каталогов отлично подходят для текстовой информации, а применение службы синхронизации файлов обеспечит автоматическую репликацию. Конструирование корпуса в базе данных выходит далеко за рамки этой книги, тем не менее мы рассмотрим создание корпуса в Sqlite далее в этой главе. Учитывая вышесказанное, мы организуем хранение наших данных на диске таким способом, чтобы обеспечить возможность систематического доступа к корпусу.

Структура корпуса на диске

Самый простой и распространенный способ организации текстового корпуса для управления им — хранение документов в файловой системе на диске. Организуя размещение документов в корпусе по подкаталогам, можно обеспечить их классификацию и содержательное разделение по имеющейся метаинформации, такой как даты. Благодаря хранению каждого документа в отдельном файле, инструменты чтения корпуса могут быстро отыскивать разные подмножества документов, обработка которых может осуществляться параллельно, когда каждый процесс получает свое, отличное от других подмножество документов.

Объекты `CorpusReader` из библиотеки NLTK, о которых рассказывается в следующем разделе, могут читать документы из каталога или из zip-архива.

Текст также является наиболее сжимаемым форматом, что делает zip-файлы со структурой каталогов на диске идеальным форматом для передачи и хранения данных. Наконец, корпусы, хранящиеся на диске, обычно статичны и обрабатываются как единое целое, что соответствует требованиям к WORM-хранилищам, перечисленным в предыдущем разделе.

Однако хранение документов в отдельных файлах может приводить к проблемам. Небольшие документы, такие как электронные письма или сообщения из «Твиттера», не имеет смысла хранить в отдельных файлах. Кроме того, электронные письма обычно хранятся в формате MBox — обычном текстовом формате с разделителями, разделяющими разные составные части сообщений, содержащие простой текст, HTML, изображения и вложения. Обычно

такие документы можно организовать по категориям, имеющимся в почтовой службе (входящие, помеченные, архивные и т. д.). Твиты, как правило, — это небольшие структуры данных в формате JSON, включающие в себя не только текст твита, но и другие метаданные, такие как имя и местоположение пользователя. Обычно для хранения сразу нескольких твитов используется формат JSON с символами перевода строки в качестве разделителя, который иногда называют форматом строк JSON. Этот формат позволяет легко читать твиты по одному и выполнять парсинг построчно, а также отыскивать разные твиты в файле. Если все твиты хранить в одном файле, размер такого файла может получиться очень большим, поэтому часто применяется организация твитов в файлы по именам пользователей, местоположению или дате, что позволяет уменьшить размеры отдельных файлов и создать осмысленную структуру каталогов.

Также для сохранения данных с некоторой логической структурой нередко используется прием ограничения размеров файлов. Например, запись данных в файл (с учетом границ между документами) производится до достижения некоторого предела (например, 128 Мбайт), после чего открывается новый файл и запись продолжается в него.

Корпус на диске неизбежно будет состоять из большого количества файлов, представляющих собой один или несколько документов, иногда разбитых на подкаталоги, соответствующие логическим единицам, таким как категории. Метаинформация о корпусе и документах также должна храниться на диске, рядом с документами. Соответственно, структура хранения корпуса на диске чрезвычайно важна для организованного чтения данных программами на Python.

Независимо от способа хранения документов — по нескольку в одном файле или по одному в отдельных файлах — корпус будет составлять множество файлов, которые тоже необходимо организовать. Если сбор данных для корпуса происходит в течение длительного времени, имеет смысл организовать каталоги по году, месяцу и дню. Если документы классифицируются по настроению, позитивному или негативному, документы каждого типа можно сгруппировать в подкаталоги по категориям. Если в системе есть несколько пользователей, генерирующих свои корпусы пользовательских текстов, таких как обзоры или твиты, тогда для каждого пользователя можно создать свой подкаталог. Все подкаталоги должны храниться вместе в одном корневом каталоге корпуса. Важно отметить, что такие метаданные корпуса, как лицензионное соглашение, манифест, файл `README` или список ссылок, также должны храниться рядом с документами, чтобы корпус можно было обрабатывать как единое целое.

Структура каталогов на диске для Baleen

Выбор организации файлов на диске оказывает большое влияние на чтение документов объектами `CorpusReader`, с которыми мы познакомимся в следующем разделе. Вот как движок сбора данных Baleen записывает корпус HTML на диск:

```
corpus
├── citation.bib
├── feeds.json
├── LICENSE.md
├── manifest.json
├── README.md
└── books
|   ├── 56d629e7c1808113ffb87eaf.html
|   ├── 56d629e7c1808113ffb87eb3.html
|   └── 56d629ebc1808113ffb87ed0.html
└── business
|   ├── 56d625d5c1808113ffb87730.html
|   ├── 56d625d6c1808113ffb87736.html
|   └── 56d625ddc1808113ffb87752.html
└── cinema
|   ├── 56d629b5c1808113ffb87d8f.html
|   ├── 56d629b5c1808113ffb87d93.html
|   └── 56d629b6c1808113ffb87d9a.html
└── cooking
    ├── 56d62af2c1808113ffb880ec.html
    ├── 56d62af2c1808113ffb880ee.html
    └── 56d62af2c1808113ffb880fa.html
```

Отметим несколько важных моментов. Во-первых, все документы хранятся как HTML-файлы с их хеш-суммами MD5 в качестве имен (для предотвращения дублирования), и каждый сохраняется в своем подкаталоге, соответствующем некоторой категории. По структуре каталогов и именам файлов легко определить, какие являются документами, а какие хранят метаинформацию. Файл citation.bib с метаинформацией хранит атрибуты корпуса, а файл LICENSE.md определяет условия использования корпуса другими лицами. Эти два файла обычно зарезервированы для общедоступных корпусов, тем не менее их полезно включать всегда, чтобы было очевидно, как можно пользоваться корпусом — по той же причине, следуя которой вы добавляете аналогичную информацию в частный репозиторий с программным обеспечением. Файлы feeds.json и manifest.json служат для хранения информации о категориях и каждом конкретном файле соответственно. Наконец, файл README.md — описание корпуса для человека.

Из этих файлов citation.bib, LICENSE.md и README.md стоят особняком, потому что могут автоматически читаться методами `citation()`, `license()` и `readme()` объекта `CorpusReader` из NLTK.

Структурированный подход к управлению и хранению корпуса обеспечивает следование текстового анализа правилу воспроизводимости, способствуя улучшенной интерпретации и уверенности в результатах. Кроме того, структурирование корпуса, как было показано выше, позволяет использовать объекты `CorpusReader`, о которых рассказывается в следующем разделе.

Эти методы легко поддаются модификации для чтения разметки Markdown или файлов, специфических для корпуса, таких как манифест:

```
import json

    # Нестандартный класс чтения корпуса
    def manifest(self):
        """
        Читает и анализирует файл manifest.json в корпусе, если имеется.
        """
        return json.load(self.open("README.md"))
```

Эти методы специально сделаны доступными для пользовательских программ, чтобы дать возможность хранить корпусы в сжатом состоянии и уменьшить объем занимаемого дискового пространства. Файл README.md играет важную роль, сообщая информацию о структуре корпуса не только для других пользователей или разработчиков, но также для «вас в будущем», чтобы напомнить о каких-то особенностях и моделях, обученных на каждом корпусе, и информации, которой обладают модели.

Объекты чтения корпусов

После структурирования и организации корпуса на диске появляются две возможности: использовать систематический подход к чтению корпуса в программном контексте, а также следить за изменениями в корпусе и управлять ими. Последнюю возможность мы обсудим в конце главы, а пока поговорим о том, как загружать документы для последующего анализа.

Большинство нетривиальных корпусов содержит тысячи документов с общим объемом текстовых данных, порой достигающим нескольких гигабайт. Исходные строки текста, загруженные из документов, необходимо подвергнуть предварительной обработке и преобразовать в представление, пригодное для дальнейшего анализа. Это аддитивный процесс, в ходе которого могут генерироваться и дублироваться данные, что увеличивает объем необходимой для работы памяти. Это важный с вычислительной точки зрения аспект, потому что без некоторого метода выбора документов на диске и потоковой передачи

информации анализ текста будет ограничен быстродействием одного компьютера, препятствуя возможности создания интересующих нас моделей. К счастью, в библиотеке NLTK имеются хорошо продуманные инструменты потокового доступа к корпусу на диске, которые передают корпус в Python через объекты `CorpusReader`.

Для обработки огромных объемов текста, генерируемых роботами в поисковых системах, были созданы фреймворки распределенных вычислений, такие как Hadoop (фреймворк Hadoop появился под влиянием двух статей, опубликованных компанией Google, в рамках проекта разработки поискового механизма Nutch). Применение приемов кластерной обработки для масштабирования с помощью Spark, преемника Hadoop, мы рассмотрим в главе 11.

`CorpusReader` — это программный интерфейс для чтения, поиска, потоковой передачи и фильтрации документов, а также для предоставления возможности преобразования и предварительной обработки данных программному коду, которому требуется доступ к данным в корпусе. При создании экземпляра `CorpusReader` ему передается путь к корневому каталогу с файлами корпуса, сигнатура для распознавания названий документов, а также кодировка файлов (по умолчанию используется UTF-8).

Поскольку кроме документов для анализа в корпусе имеются дополнительные файлы (например, README, citation, license и др.), объект чтения нуждается в некотором механизме, помогающем точно идентифицировать, какие документы являются частью корпуса. Роль такого механизма играет параметр. В нем можно указать точный перечень имен или регулярное выражение для сопоставления со всеми документами в корневом каталоге (например, \w+\.txt, которому соответствует одна или несколько букв или цифр в имени файла, за которыми следует расширение .txt). Например, в следующем каталоге этому регулярному выражению будут соответствовать три файла в каталоге speeches и файл transcript.txt, но не файлы license, README, citation и metadata:

```
corpus
├── LICENSE.md
├── README.md
├── citation.bib
├── transcript.txt
└── speeches
    ├── 04102008.txt
    ├── 10142009.txt
    ├── 09012014.txt
    └── metadata.json
```

Эти три простых параметра дают объекту `CorpusReader` возможность перебрать все документы в корпусе, открыть каждый с использованием правильной кодировки и позволяют программистам получить доступ к метаданным отдельно.

По умолчанию объекты `CorpusReader` из NLTK могут читать корпусы, хранящиеся в сжатых zip-архивах, а с применением дополнительных простых расширений читать также сжатые архивы Gzip и Bzip.

Сама по себе идея `CorpusReader` может показаться не особенно эффектной, но при работе с множеством документов этот интерфейс позволяет программистам прочитать один или несколько документов в память, отыскать определенные места в корпусе, не открывая и не читая ненужные документы, передать данные в процесс анализа, храня в памяти документы по одному, и отфильтровать или выбрать конкретные документы из корпуса. Эти методы позволяют выполнить анализ нетривиальных корпусов в памяти, потому что всякий раз там находятся лишь несколько документов за раз.

Чтобы проанализировать собственный корпус, принадлежащий определенной предметной области и предназначенный для создания конкретных моделей, вам потребуется специализированный объект чтения корпуса. Это настолько важно для прикладного анализа текста, что мы посвятили этой теме бóльшую часть остатка главы! В этом разделе мы рассмотрим инструменты чтения корпусов, которые есть в библиотеке NLTK, и возможности структурирования корпусов, чтобы упростить их чтение с последующим применением. Затем мы покажем, как реализовать свой объект чтения корпуса, выполняющий операции, специфические для приложения, а именно обработку HTML-файлов, полученных в процессе сбора исходных данных.

Потоковый доступ к данным с помощью NLTK

В состав библиотеки NLTK входит большое разнообразие объектов для чтения корпусов (66 на момент написания этих строк), предусматривающих доступ к текстовым корпусам и лексическим ресурсам, которые можно загрузить с ее помощью. Она также включает в себя универсальные вспомогательные объекты `CorpusReader`, которые предъявляют более жесткие требования к структуре корпуса, зато дают возможность быстро создавать корпусы и связывать их с объектами чтения. Кроме того, их имена подсказывают, для каких целей их можно использовать. Вот некоторые из наиболее примечательных объектов `CorpusReader`:

`PlaintextCorpusReader`

Предназначен для чтения корпуса простых текстовых документов, где, как предполагается, абзацы отделены друг от друга пустыми строками.

`TaggedCorpusReader`

Предназначен для чтения корпуса с маркерами частей речи, где каждое предложение находится в отдельной строке, а лексемы разделены соответствующими им тегами.

`BracketParseCorpusReader`

Предназначен для чтения корпуса, состоящего из деревьев синтаксического анализа, заключенных в круглые скобки.

`ChunkedCorpusReader`

Предназначен для чтения фрагментированного корпуса (возможно, размеченного тегами), отформатированного с помощью круглых скобок.

`TwitterCorpusReader`

Предназначен для чтения корпуса, состоящего из твитов, сериализованных в формат строк JSON.

`WordListCorpusReader`

Предназначен для чтения списка слов, по одному в строке. Пустые строки игнорируются.

`XMLCorpusReader`

Предназначен для чтения корпуса, состоящего из XML-документов.

`CategorizedCorpusReader`

Подмешиваемый класс для объектов чтения корпусов, в которых документы организованы по категориям.

Объекты чтения размеченных и фрагментированных корпусов, а также корпусов, состоящих из деревьев синтаксического анализа, предназначены для чтения *аннотированных* корпусов; собираясь вручную осуществлять предметно-ориентированное аннотирование перед машинным обучением, вы обязательно должны изучить распознаваемые ими форматы. Объекты чтения корпусов с твитами, документами XML и простыми текстовыми документами позволяют

указать, как обрабатывать данные, хранящиеся на диске в разных форматах, что дает возможность создавать расширения для чтения корпусов в формате CSV, JSON или даже из базы данных. Если ваш корпус уже в одном из этих форматов, тогда вам почти ничего не придется делать. Например, рассмотрим простой корпус киносценариев сериалов *Star Wars* («Звездные войны») и *Star Trek* («Звездный путь») со следующей организацией:

```
corpus
├── LICENSE
├── README
└── Star Trek
|   ├── Star Trek - Balance of Terror.txt
|   ├── Star Trek - First Contact.txt
|   ├── Star Trek - Generations.txt
|   ├── Star Trek - Nemesis.txt
|   ├── Star Trek - The Motion Picture.txt
|   ├── Star Trek 2 - The Wrath of Khan.txt
|   └── Star Trek.txt
└── Star Wars
|   ├── Star Wars Episode 1.txt
|   ├── Star Wars Episode 2.txt
|   ├── Star Wars Episode 3.txt
|   ├── Star Wars Episode 4.txt
|   ├── Star Wars Episode 5.txt
|   ├── Star Wars Episode 6.txt
|   └── Star Wars Episode 7.txt
└── citation.bib
```

Для доступа к данным в киносценариях отлично подойдет `CategorizedPlaintextCorpusReader`, потому что все документы хранятся в простых текстовых файлах и разбиты на две категории, а именно: Star Wars и Star Trek. Чтобы задействовать `CategorizedPlaintextCorpusReader`, нужно указать регулярные выражения, которые позволят объекту чтения автоматически определять идентификаторы файлов (`fileids`) и категории (`categories`):

```
from nltk.corpus.reader.plaintext import CategorizedPlaintextCorpusReader


DOC_PATTERN = r'(?!\.)[\w_\s]+/[\w\s\d\-]+\.txt'
CAT_PATTERN = r'([\w_\s]+)/.*'

corpus = CategorizedPlaintextCorpusReader(
    '/path/to/corpus/root', DOC_PATTERN, cat_pattern = CAT_PATTERN
)
```

Регулярное выражение с шаблоном документа определяет имена файлов документов как возможно включающие путь, относительно корневого каталога корпуса состоящий из одной или нескольких букв, цифр, пробелов или под-

черкиваний, за которыми следует слеш (/), затем одна или несколько букв, цифр, пробелов или дефисов, и расширение .txt. Это выражение соответствует таким именам файлов с документами, как Star Wars/Star Wars Episode 1.txt, но не соответствует файлу episode.txt. Регулярное выражение с шаблоном категории является усеченной версией первого регулярного выражения и состоит из сохраняющей группы, которая определяет название категории по имени каталога (например, для Star Wars/anything.txt будет определена категория `Star Wars`). Начнем исследование данных на диске с изучения полученных имен:

```
corpus.categories()
# ['Star Trek', 'Star Wars']

corpus.fileids()
# ['Star Trek/Star Trek - Balance of Terror.txt',
#  'Star Trek/Star Trek - First Contact.txt', ...]
```

Регулярные выражения могут быть очень сложными, но они предлагают мощный механизм, помогающий точно определить, что должен загрузить объект чтения корпуса и как. Также можно явно передать список категорий и идентификаторов файлов, но это сделает объект чтения менее гибким. Использование регулярных выражений позволяет добавлять новые категории, просто создавая новые каталоги в корпусе, и новые документы, перемещая файлы в требуемый каталог.

Теперь, научившись создавать объекты `CorpusReader` из библиотеки NLTK, займемся приемами потоковой передачи собранных ранее данных в формате HTML.

Чтение корпуса HTML

Допустим, мы собрали некоторые данные из интернета. Можно смело предположить, что эти данные хранятся в формате HTML. Как вариант, для потокового чтения такого корпуса можно просто удалить все теги из документов HTML, сохранить результат в виде простого текста и использовать `CategorizedPlaintextCorpusReader`. Но, пойдя этим путем, мы потеряем преимущества разметки HTML, а именно *структуру* текста, которой можно воспользоваться на этапе предварительной обработки. Поэтому в данном разделе мы начнем создание своего объекта `HTMLCorpusReader` и дополним в следующей главе:

```
from nltk.corpus.reader.api import CorpusReader
from nltk.corpus.reader.api import CategorizedCorpusReader

CAT_PATTERN = r'([a-z_\s]+)/.*'
```

```
DOC_PATTERN = r'(?!\.)[a-z_\s]+/[a-f0-9]+\.json'
TAGS = ['h1', 'h2', 'h3', 'h4', 'h5', 'h6', 'h7', 'p', 'li']

class HTMLCorpusReader(CategorizedCorpusReader, CorpusReader):
    """
    Объект чтения корпуса с HTML-документами для получения
    возможности дополнительной предварительной обработки.
    """

    def __init__(self, root, fileids = DOC_PATTERN, encoding = 'utf8',
                 tags = TAGS, **kwargs):
        """
        Инициализирует объект чтения корпуса.
        Аргументы, управляющие классификацией
        (``cat_pattern``, ``cat_map`` и ``cat_file``), передаются
        в конструктор ``CategorizedCorpusReader``. остальные аргументы
        передаются в конструктор ``CorpusReader``.
        """
        # Добавить шаблон категорий, если он не был передан в класс явно.
        if not any(key.startswith('cat_') for key in kwargs.keys()):
            kwargs['cat_pattern'] = CAT_PATTERN

        # Инициализировать объекты чтения корпуса из NLTK
        CategorizedCorpusReader.__init__(self, kwargs)
        CorpusReader.__init__(self, root, fileids, encoding)

        # Сохранить теги, подлежащие извлечению.
        self.tags = tags
```

Наш класс `HTMLCorpusReader` наследует два класса, `CategorizedCorpusReader` и `CorpusReader`, подобно тому как `CategorizedPlaintextCorpusReader` использует подмешиваемый класс для категоризации. *Множественное наследование* может порождать дополнительные сложности, поэтому большая часть кода в функции `__init__` явно определяет, какие аргументы должны передаваться каждому унаследованному классу. В частности, классу `CategorizedCorpusReader` передаются именованные аргументы, а класс `CorpusReader` инициализируется корневым каталогом корпуса, так же как и идентификаторами файлов (`fileids`) и кодировкой HTML (`encoding`). Мы также добавили дополнительный параметр, с помощью которого пользователь сможет указать, какие теги HTML должны интерпретироваться как независимые абзацы.

Следующим шагом дополним `HTMLCorpusReader` методом для *фильтрации* файлов на диске по названиям категорий или именам файлов:

```
    def resolve(self, fileids, categories):
        """
        Возвращает список идентификаторов файлов или названий категорий,
        которые передаются каждой внутренней функции объекта чтения корпуса.
```

```
        Реализована по аналогии с ``CategorizedPlaintextCorpusReader`` в NLTK.
        """
        if fileids is not None and categories is not None:
            raise ValueError("Specify fileids or categories, not both")

        if categories is not None:
            return self.fileids(categories)
        return fileids
```

Этот метод возвращает список идентификаторов файлов независимо от категорий, к которым они относятся. Он придает дополнительную гибкость и экспортирует сигнатуру метода, которая будет использоваться практически всеми другими методами объекта чтения. Получив оба параметра, `categories` и `fileids`, метод `resolve` выводит сообщение об ошибке. Если определены только категории, метод использует `CorpusReader` для определения идентификаторов файлов `fileids`, ассоциированных с заданными категориями. Обратите внимание, что в `categories` может передаваться одна категория или список категорий. В противном случае просто возвращается значение аргумента `fileids`: если оно равно `None`, объект `CorpusReader` автоматически прочитает все документы в корпусе без фильтрации.

Обратите внимание, что возможность прочитать только часть корпуса станет важна, когда мы приблизимся к машинному обучению, в частности, для выполнения перекрестной проверки, когда нам понадобится разбить корпус на обучающую и контрольную выборки.

На данный момент наш объект чтения `HTMLCorpusReader` не имеет метода для чтения потока документов по одному за раз. Вместо этого он будет в потоковом режиме поставлять нашим методам весь текст каждого отдельного документа в корпусе. Однако нам нужно выполнить парсинг одного документа HTML за раз — следующий метод дает нам доступ к содержимому отдельных документов:

```
import codecs

    def docs(self, fileids = None, categories = None):
        """
        Возвращает полный текст HTML-документа, закрывая его
        по завершении чтения.
        """
        # Получить список файлов для чтения
        fileids = self.resolve(fileids, categories)

        # Создать генератор, загружающий документы в память по одному.
        for path, encoding in self.abspaths(fileids, include_encoding = True):
            with codecs.open(path, 'r', encoding = encoding) as f:
                yield f.read()
```

Теперь наш нестандартный объект чтения корпуса знает, как обрабатывать документы в корпусе по одному, что дает нам возможность выполнять фильтрацию и поиск разных разделов корпуса. Он может обрабатывать списки файлов и категорий и использовать все инструменты, импортированные из NLTK, для упрощения доступа к диску.

Мониторинг корпуса

Как мы уже установили в этой главе, прикладной анализ текста требует существенного управления данными и предварительной обработки. Описанные методы сбора, управления и предварительной обработки данных трудоемки и требуют времени, но являются важными этапами подготовки к машинному обучению. Учитывая необходимость тратить время, силы и дисковое пространство, рекомендуется снабжать данные некоторой метаинформацией о деталях создания корпуса.

В этом разделе рассказывается, как создать систему мониторинга для нужд сбора и предварительной обработки данных. Для начала рассмотрим конкретные виды информации для мониторинга, такие как даты и источники данных. Учитывая огромные размеры корпусов, с которыми нам придется работать, мы должны также следить за размерами каждого файла на диске.

```
def sizes(self, fileids = None, categories = None):
    """
    Возвращает список кортежей, идентификатор файла и его размер.
    Эта функция используется для выявления необычно больших файлов
    в корпусе.
    """
    # Получить список файлов
    fileids = self.resolve(fileids, categories)

    # Создать генератор, возвращающий имена и размеры файлов
    for path in self.abspaths(fileids):
        yield path, os.path.getsize(path)
```

В своей практике работы с корпусами RSS HTML мы не раз замечали, что кроме текста из источников извлекается большое количество файлов с изображениями, аудио- и видеофайлов. Эти медиафайлы быстро съедают дисковое пространство в процессе сбора данных и мешают предварительной обработке. Метод `sizes`, представленный выше, является нашей реакцией на такой опыт работы с корпусами и помогает нам выявлять отдельные файлы в корпусе, фактические размеры которых превышают ожидаемые (например, изображения и видеофайлы, которые могут быть закодированы как текст). Этот метод

позволяет определить полный размер корпуса и следить за его изменением с течением времени.

Чтение корпуса из базы данных

Нет двух абсолютно одинаковых корпусов, так как каждому новому приложению требуется новый и предметно-ориентированный корпус, а для каждого корпуса требуется свой, специализированный объект чтения. В главе 12 мы рассмотрим приложение анализа настроения, использующее корпус с примерно 18 000 обзорами альбомов, которые можно найти на веб-сайте Pitchfork.com. Набор данных доступен в виде базы данных Sqlite со схемой, изображенной на рис. 2.3.

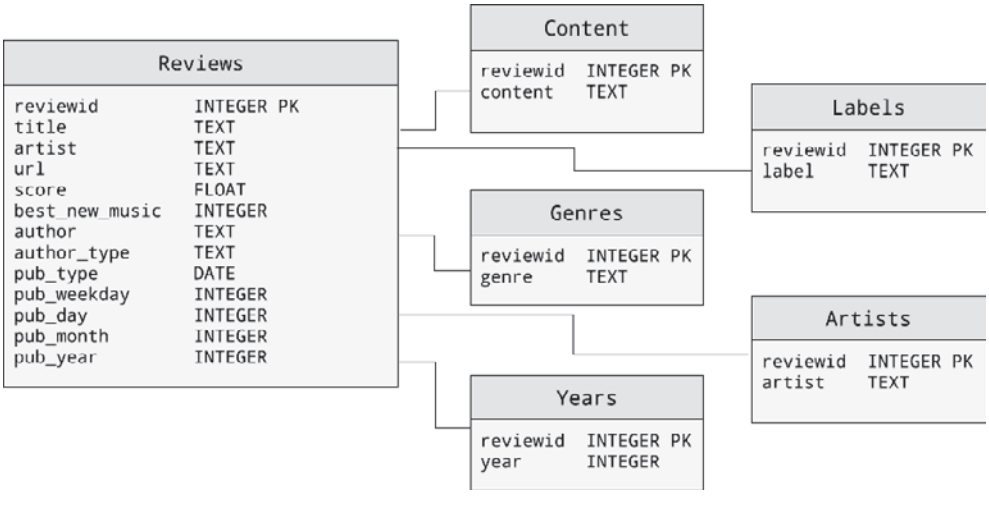


Рис. 2.3. Схема корпуса с обзорами альбомов в базе данных Sqlite

Для работы с этим корпусом создадим свой класс `SqliteCorpusReader`, обращающийся к разным компонентам и имитирующий поведение `CorpusReader` из библиотеки NLTK, но не наследующий его.

Нам нужно, чтобы наш `SqliteCorpusReader` извлекал результаты из базы данных, не потребляя слишком много памяти; так же как в случае с `HTMLCorpusReader` из предыдущего раздела, будем эффективно и рационально извлекать данные по одной записи для последующей обработки, нормализации и преобразования (которые будут обсуждаться в главе 3). По этой причине не рекомендуется использовать SQL-подобную команду `fetchall()`, так как она может вынудить

нас слишком долго ждать результатов, прежде чем мы сможем начать итерации. Вместо этого наши методы `ids()`, `scores()` и `texts()` будут использовать `fetchone()`, неплохую альтернативу в данном случае, хотя при работе с большими базами данных производительнее может оказаться пакетная выборка данных (например, с использованием `fetchmany()`).

```
import sqlite3

class SqliteCorpusReader(object):

    def __init__(self, path):
        self._cur = sqlite3.connect(path).cursor()

    def ids(self):
        """
        Возвращает идентификаторы обзоров, позволяющие извлекать
        другие метаданные обзоров
        """
        self._cur.execute("SELECT reviewid FROM content")
        for idx in iter(self._cur.fetchone, None):
            yield idx

    def scores(self):
        """
        Возвращает оценку обзора с целью использования
        для последующего обучения с учителем
        """
        self._cur.execute("SELECT score FROM reviews")
        for score in iter(self._cur.fetchone, None):
            yield score

    def texts(self):
        """
        Возвращает полный текст всех обзоров с целью предварительной
        обработки и векторизации для последующего обучения с учителем
        """
        self._cur.execute("SELECT content FROM content")
        for text in iter(self._cur.fetchone, None):
            yield text
```

Из примеров реализации `HTMLCorpusReader` и `SqliteCorpusReader` следует, что мы всегда должны быть готовы написать новый объект чтения корпуса для каждого нового набора данных. Однако, как мы надеемся, эти примеры смогли показать не только их полезность, но и сходства. В следующей главе мы расширим `HTMLCorpusReader` так, чтобы его можно было использовать для доступа к более мелким компонентам нашего текста, что пригодится для предварительной обработки и проектирования признаков.

В заключение

В этой главе вы узнали, что для анализа текста необходим большой, надежный, предметно-ориентированный корпус. Поскольку корпусы очень большие и часто непредсказуемого размера, мы обсудили методы их структуризации и управления ими. Мы увидели, как объекты чтения корпусов могут использовать эту структуру, а также снижать требования к объему памяти, осуществляя загрузку данных потоковым способом. Наконец, мы начали создание двух своих объектов чтения: один для корпусов с HTML-документами, хранящимися на диске, а другой для корпусов с документами в базе данных Sqlite.

В следующей главе вы узнаете, как осуществлять предварительную обработку данных, расширять объекты, созданные в этой главе, добавлять методы предварительной обработки разметки HTML потоковым способом, и получать окончательную структуру текстовых данных, пригодную для машинного обучения — список документов со списками абзацев, которые являются списками предложений, а те, в свою очередь, — списками кортежей с лексемами и тегами, определяющими части речи.

3 Предварительная обработка и преобразование корпуса

В предыдущей главе вы узнали, как создать и структурировать свой предметно-ориентированный корпус. К сожалению, любой корпус в его исходном виде совершенно непригоден для анализа — его необходимо предварительно обработать и сжать. Фактически, главной причиной, побудившей написать эту книгу, стала огромная проблема, с которой мы постоянно сталкиваемся в своей практике создания и преобразования корпусов, достаточно больших, что заставляет нас уделять особое внимание эффективности при разработке приложений данных. Учитывая, как много сил и времени требуется для предварительной обработки и преобразования текста, порой удивляет, как мало ресурсов существует в поддержку (или хотя бы для признания важности!) этих этапов.

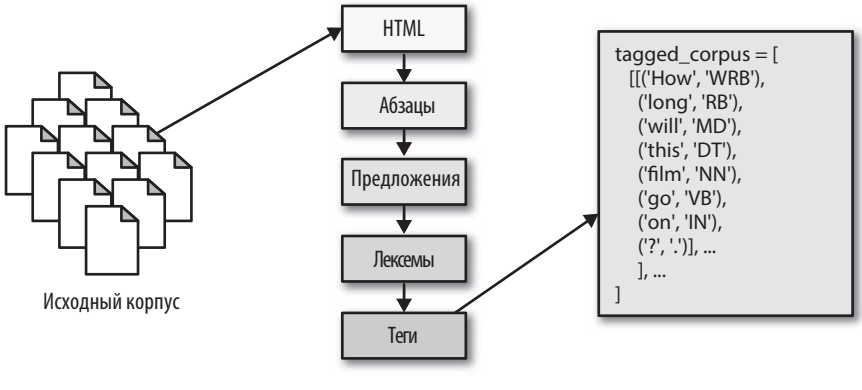


Рис. 3.1. Разбивка документа, сегментация, выделение лексем и маркировка

В этой главе мы предложим вариант многоцелевого фреймворка предварительной обработки, который можно систематически использовать для преобразования собранного исходного текста в форму, пригодную для дальнейшей обработки и моделирования. Наш фреймворк реализует пять основных этапов, изображенных на рис. 3.1: извлечение контента, выделение абзацев, предложений и лексем и маркировка лексем тегами частей речи. Для каждого из этапов мы представим функции в виде методов класса `HTMLCorpusReader`, объявленного в предыдущей главе.

Разбивка документов

В предыдущей главе мы начали конструировать свой класс `HTMLCorpusReader`, снабдив его методами для фильтрации, доступа и подсчета размеров документов (`resolve()`, `docs()` и `sizes()`). Наследуя объект `CorpusReader` из библиотеки NLTK, наш объект чтения корпусов также реализует стандартный API предварительной обработки, в который входят следующие методы:

`raw()`

Предоставляет доступ к исходному тексту, не прошедшему предварительную обработку.

`sents()`

Генератор, выделяющий отдельные предложения из текста.

`words()`

Выделяет из текста отдельные слова.

Основное наше внимание будет сосредоточено на методах обработки естественного языка для объекта чтения корпуса, предназначенного для подготовки HTML-документов, полученных из Сети, однако, в зависимости от формы корпуса, другие методы могут оказаться более удобными. Стоит отметить, что объекты `CorpusReader` в библиотеке NLTK уже имеют множество методов для других случаев использования: например, автоматического добавления тегов или синтаксического анализа предложений, преобразования аннотированного текста в структуры данных, такие как объекты `Tree`, или играют роль вспомогательных утилит для конкретных форматов, таких как выделение отдельных элементов XML.

Для обучения моделей с применением приемов машинного обучения мы должны включить все эти методы в процесс извлечения признаков. На протяжении

оставшейся части этой главы мы обсудим детали предварительной обработки, покажем, как использовать и модифицировать эти методы для доступа к содержимому, и исследуем признаки внутри документов.

Выявление и извлечение основного контента

Интернет является отличным источником текстовой информации для создания новых и полезных корпусов, но точно так же он является областью беззакония, в том смысле, что веб-страницы не обязаны структурироваться в строгом соответствии с каким-то набором стандартов. Как результат, разметка HTML, которая сама по себе структурирована, может производиться и отображаться множеством иногда беспорядочных способов. Эта непредсказуемость затрудняет извлечение данных из HTML-документов методичным и предсказуемым способом.

Отличным инструментом для борьбы с высокой вариативностью документов, собираемых в интернете, является библиотека readability-lxml. Это обертка на Python для эксперимента JavaScript Readability, реализованного Arc90. Подобно браузерам, таким как Safari и Chrome, предлагающим режим чтения текста документов, Readability удаляет из страницы все артефакты, отвлекающие внимание, оставляя только текст.

Обрабатывая документ HTML, Readability использует комплект регулярных выражений для удаления навигационных меню, рекламы, тегов сценариев и CSS, затем конструирует новое дерево объектной модели документа (Document Object Model, DOM), извлекает текст из первоначального дерева и встраивает его во вновь созданное дерево. В следующем примере, расширяющем класс `HTMLCorpusReader`, мы импортируем два модуля из библиотеки Readability, `Unparseable` и `Document`, которые затем используем для извлечения и очистки текста HTML на первом этапе процесса предварительной обработки.

Метод `html` перебирает все файлы и использует метод `summary` из класса `readability.Document` для удаления любого нетекстового содержимого, а также сценариев и тегов стилей. Также он исправляет некоторые из наиболее распространенных ошибок, связанных с использованием тегов (например, `<div>` и `<br>`), и генерирует исключение, только если оказывается не в состоянии разобрать исходную разметку HTML. Типичной причиной исключения является передача функции пустого документа:

```
from readability.readability import Unparseable
from readability.readability import Document as Paper

    def html(self, fileids = None, categories = None):
        """
```

```
        Возвращает содержимое HTML каждого документа, очищая его
        с помощью библиотеки readability-lxml.
        """
        for doc in self.docs(fileids, categories):
            try:
                yield Paper(doc).summary()
            except Unparseable as e:
                print("Could not parse HTML: {}".format(e))
                continue
```

Обратите внимание, что метод выше может выводить предупреждения в журнал `readability`; есть возможность изменить уровень подробности по своему усмотрению, добавив:

```
import logging
log = logging.getLogger("readability.readability")
log.setLevel('WARNING')
```

Результатом метода `html()` является очищенный и хорошо структурированный текст HTML. В следующих нескольких разделах мы добавим дополнительные методы для его последовательного разбиения на составляющие: абзацы, предложения и лексемы.

Разделение документов на абзацы

Теперь, научившись фильтровать исходные документы HTML, которые мы извлекли в предыдущей главе, перейдем к предварительной обработке корпуса и структурируем его, чтобы сделать пригодным для использования в машинном обучении. На рис. 3.2 показано, на какие смысловые элементы делится статья

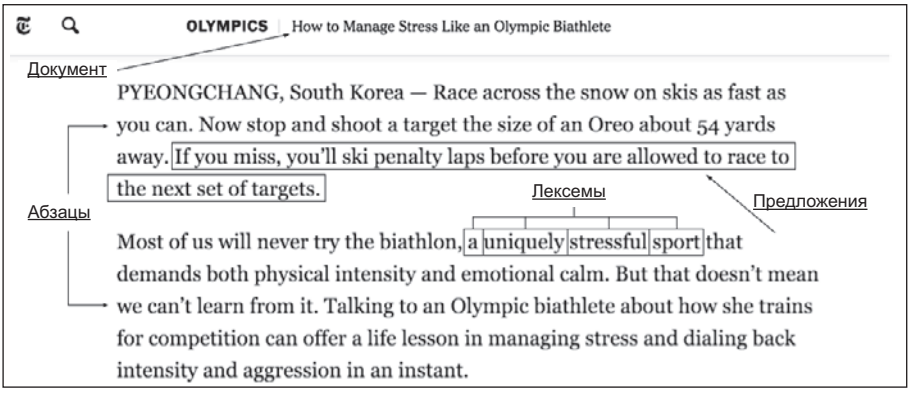


Рис. 3.2. Декомпозиция документа, иллюстрирующая основные смысловые элементы: абзацы, предложения и отдельные лексемы

«Как управлять стрессом подобно биатлонисту» из *New York Times*[1]. Детализация исследования документа может существенно повлиять на то, как будет классифицирована статья: как посвященная «популярному виду спорта» или «личному здоровью» (или и тому, и другому!).

Этот пример иллюстрирует, почему задачи векторизации, извлечения признаков и машинного обучения, которые мы будем выполнять в следующих главах, в значительной мере зависят от нашей способности эффективно разбить документы на составляющие их компоненты, сохранив при этом первоначальную структуру.

Точность и чувствительность наших моделей будет зависеть от того, насколько эффективно мы умеем связывать лексемы с контекстом, в котором они появляются.

Абзацы, заключающие законченные идеи, выполняют роль структурной единицы документа, и в первую очередь мы должны выделить абзацы, имеющиеся в тексте. Некоторые объекты чтения корпусов в библиотеке NLTK, такие как `PlaintextCorpusReader`, реализуют метод `paras()` — генератор абзацев, которые определяются как блоки текста, разделенные двумя символами перевода строки.

Однако сейчас мы имеем дело не с простым текстом, поэтому должны реализовать метод, извлекающий абзацы из HTML. К счастью, наш метод `html()` сохраняет структуру HTML-документов. Это значит, что мы можем выделить содержимое абзацев по тегам `<p>`, которые в языке HTML определяют абзацы. Поскольку содержимое может также оформляться другими способами (например, в виде других структур внутри документа, таких как заголовки и списки), мы выполним более широкий поиск по тексту, используя BeautifulSoup.

Как вы помните, в главе 2 мы определили класс `HTMLCorpusReader` так, что объекты чтения получают искомые HTML-теги в виде атрибута класса. Это множество тегов можно расширить, сократить или изменить как-то иначе в соответствии с конкретными требованиями.

Определим метод `paras()`, выполняющий обход всех файлов и передающий каждый HTML-документ конструктору Beautiful Soup, указав, что анализ разметки HTML должен производиться с использованием HTML-парсера lxml. Получающийся в результате объект `soup` хранит древовидную структуру, по которой можно перемещаться, используя оригинальные теги и элементы HTML.

[1] Tara Parker-Pope, *How to Manage Stress Like a Biathlete* (2018), https://nyti.ms/2GJBGwr

Из каждого такого объекта мы выберем все теги, входящие в предопределенное множество, и вернем текст, содержащийся в них. После этого, закончив работу с файлом, вызовем метод `decompose` класса `BeautifulSoup`, чтобы уничтожить дерево и освободить занимаемую им память.

```
import bs4

# Теги для извлечения абзацев из HTML-документов
tags = [
    'h1', 'h2', 'h3', 'h4', 'h5', 'h6', 'h7', 'p', 'li'
]

    def paras(self, fileids = None, categories = None):
        """
        Использует BeautifulSoup для выделения абзацев из HTML.
        """
        for html in self.html(fileids, categories):
            soup = bs4.BeautifulSoup(html, 'lxml')
            for element in soup.find_all(tags):
                yield element.text
            soup.decompose()
```

Метод `paras()` — это функция-генератор, возвращающая исходный текст абзацев из всех документов, от первого до последнего, и просто перешагивающая через границы документов. Если передать методу `paras()` конкретное значение в параметре `fileids`, он вернет абзацы только из указанного файла.

Стоит отметить, что методы чтения `paras()` многих объектов корпусов в библиотеке NLTK, таких как `PlaintextCorpusReader`, действуют по-разному, часто выделяя предложения и лексемы вдобавок к выделению абзацев. Это объясняется тем, что в большинстве своем методы в NLTK предполагают, что работа ведется с аннотированным корпусом и нет необходимости реконструировать абзацы. Наши методы, напротив, предназначены для обработки исходного, неаннотированного корпуса и поэтому должны производить его реконструирование.

Сегментация: выделение предложений

Если абзацы можно рассматривать как структурные единицы документа, тогда предложения можно считать смысловыми единицами. Подобно абзацу, выражающему единственную идею, предложение содержит законченную мысль, которую мы хотели бы сформулировать и высказать.

В этом разделе мы реализуем *сегментацию* — разделение текста на предложения, которые потом будут обрабатываться методами маркировки слов тегами частей речи, которые будут рассмотрены далее в этой главе и которые в свою очередь

опираются на внутренне согласованную морфологию. Чтобы выделить предложения, напишем новый метод, `sents()`, вызывающий `paras()` и возвращающий генератор (итератор), который перебирает все предложения из всех абзацев.

Синтаксическая сегментация не является необходимым условием для маркировки текста частями речи. В некоторых случаях может потребоваться сначала выполнить маркировку, чтобы правильно разбить текст на предложения, например, при анализе речевой информации или транскрибированного текста, где границы между предложениями не так очевидны. В письменном тексте предварительная сегментация упрощает маркировку лексем частями речи. С речевыми данными часто лучше справляются инструменты SpaCy, тогда как для обработки письменного языка лучше подходит NLTK.

Наш метод `sents()` выполняет обход всех абзацев, выделенных методом `paras`, и использует метод `sent_tokenize` из библиотеки NLTK для выполнения фактической сегментации. Внутренне `sent_tokenize` использует `PunktSentenceTokenizer`, модель, предварительно обученную правилам преобразования (по сути — набор регулярных выражений) для разных видов слов и знаков препинания (например, точки, знаки вопроса, знаки восклицания, капитализация и т. д.), которые служат признаками начала и конца предложения. Модель можно применить к абзацу и получить генератор предложений:

```
from nltk import sent_tokenize

    def sents(self, fileids = None, categories = None):
        """
        Использует встроенный механизм для выделения предложений из
        абзацев. Обратите внимание, что для парсинга разметки HTML
        этот метод использует BeautifulSoup.
        """
        for paragraph in self.paras(fileids, categories):
            for sentence in sent_tokenize(paragraph):
                yield sentence
```

Хотя `PunktSentenceTokenizer` из NLTK обучен распознавать тексты на английском языке, он также хорошо подходит для большинства европейских языков. Он прекрасно справляется со своей задачей, когда получает стандартные абзацы:

```
['Beautiful is better than ugly.', 'Explicit is better than implicit.',
'Simple is better than complex.', 'Complex is better than complicated.',
'Flat is better than nested.', 'Sparse is better than dense.',
'Readability counts.', "Special cases aren't special enough to break the
rules.", 'Although practicality beats purity.', 'Errors should never pass
silently.', 'Unless explicitly silenced.', 'In the face of ambiguity, refuse
the temptation to guess.', 'There should be one -- and preferably only one --
obvious way to do it.', "Although that way may not be obvious at first unless
```

```
you're Dutch.", 'Now is better than never.', 'Although never is often better
than *right* now.', "If the implementation is hard to explain, it's a bad
idea.", 'If the implementation is easy to explain, it may be a good idea.',
"Namespaces are one honking great idea -- let's do more of those!"]
```

Однако знаки препинания не всегда имеют однозначную интерпретацию, например: точки обычно служат признаком конца предложения, но они могут также появляться в вещественных числах, сокращениях и многоточиях. Иными словами, определение границ предложений — не всегда простая задача. Поэтому применение `PunktSentenceTokenizer` к нестандартному тексту не всегда дает приемлемые результаты:

```
['Baa, baa, black sheep,\nHave you any wool?', 'Yes, sir, yes, sir,\nThree
bags full;\nOne for the master,\nAnd one for the dame,\nAnd one for the little
boy\nWho lives down the lane.']
```

NLTK предоставляет альтернативные средства выделения предложений (например, для твитов), с которыми стоит познакомиться. Но, как бы то ни было, если в вашем предметном пространстве имеются свои уникальные особенности разграничения предложений, рекомендуется обучить свою модель, выполняющую такие разграничения, используя предметно-ориентированный контент.

Лексемизация: выделение лексем

Мы определили предложения как смысловые единицы, а абзацы — как структурные единицы документа. В этом разделе мы займемся выделением *лексем*, синтаксических единиц языка, кодирующих семантическую информацию в виде последовательности символов.

Лексемизация — это процесс извлечения лексем, и для его реализации мы используем `WordPunctTokenizer` — класс, основанный на применении регулярных выражений, который выделяет лексемы по пробельным символам и знакам препинания и возвращает список алфавитных и неалфавитных символов:

```
from nltk import wordpunct_tokenize

    def words(self, fileids = None, categories = None):
        """
        Использует встроенный механизм для выделения слов из предложений.
        Обратите внимание, что для парсинга разметки HTML
        этот метод использует BeautifulSoup
        """
        for sentence in self.sents(fileids, categories):
            for token in wordpunct_tokenize(sentence):
                yield token
```

Так же, как разграничение предложений, выделение лексем — не всегда простая задача. Мы должны подумать о многом, например: удалять ли знаки препинания из лексем и, если да, следует ли интерпретировать эти знаки препинания как самостоятельные лексемы? Следует ли сохранять слова с дефисами в виде одной лексемы или разбивать их на составные части? Как интерпретировать сокращенные формы слов: как одну лексему или как две — и где в последнем случае проводить границу?

В зависимости от ответов на эти вопросы выбираются разные инструменты выделения лексем. Из множества таких инструментов, доступных в NLTK (например, `TreebankWordTokenizer`, `WordPunctTokenize`, `PunktWordTokenizer` и др.), чаще всего для лексемизации выбирается функция `word_tokenize`, которая использует `TreebankWordTokenizer` и регулярные выражения из библиотеки Penn Treebank. Она поддерживает разбивку стандартных сокращений (например, «wouldn't» превращается в «would» и «n't») и интерпретирует знаки препинания (такие как точки, одиночные кавычки и запятые, за которыми следуют пробелы) как самостоятельные лексемы. Напротив, `WordPunctTokenizer` основан на классе `RegexpTokenizer`, выделяющем лексемы с помощью регулярного выражения `\w+|[^\w\s]+` , которому соответствуют лексемы или разделители лексем, в результате чего получается последовательность алфавитных и неалфавитных символов. Класс `RegexpTokenizer` также можно использовать для создания своего инструмента выделения лексем.

Маркировка частями речи

Теперь, получив доступ к лексемам внутри предложений в абзацах наших документов, пометим каждую лексему соответствующей ей *частью речи*. Части речи (например, глаголы, существительные, предлоги, прилагательные) указывают на роль слова в контексте предложения. В английском, как и во многих других языках, одно и то же слово может играть разные роли, и нам хотелось бы иметь возможность различать их (например, «building» (строительство, строение) может быть существительным или глаголом). Маркировка частями речи заключается в добавлении к каждой лексеме соответствующего тега, несущего информацию об определении слова и его роли в текущем контексте.

Воспользуемся готовым средством маркировки из NLTK, `pos_tag`, который на момент написания этих слов использовал `PerceptronTagger()` и набор тегов из Penn Treebank. Библиотека Penn Treebank включает 36 тегов, несущих информацию о частях речи, структурной организации, другие грамматические характеристики (`NN` для существительных в единственном числе, `NNS` для суще-

ствительных во множественном числе, `JJ` для прилагательных, `RB` для наречий, `VB` для глаголов, `PRP` для личных местоимений и т. д.).

Метод `tokenize` возвращает генератор, предоставляющий нам список списков абзацев, являющихся списками предложений, которые, в свою очередь, являются списками лексем, маркированных тегами частей речи. Маркированные лексемы представлены кортежами (`tag`, `token`), где `tag` — строка (регистр символов имеет значение), определяющая роль лексемы `token` в текущем контексте:

```
from nltk import pos_tag, sent_tokenize, wordpunct_tokenize

    def tokenize(self, fileids = None, categories = None):
        """
        Сегментирует, лексемизирует и маркирует документ в корпусе.
        """
        for paragraph in self.paras(fileids = fileids):
            yield [
                pos_tag(wordpunct_tokenize(sent))
                for sent in sent_tokenize(paragraph)
            ]
```

Рассмотрим абзац «The old building is scheduled for demolition. The contractors will begin building a new structure next month»[1]. В данном случае метод `pos_tag` определит, что в первом случае слово «building» (строение) используется как существительное в единственном числе, а во втором — как глагол «to build» (строить):

```
[[('The', 'DT'), ('old', 'JJ'), ('building', 'NN'), ('is', 'VBZ'),
('scheduled', 'VBN'), ('for', 'IN'), ('demolition', 'NN'), ('.', '.')],
[('The', 'DT'), ('contractors', 'NNS'), ('will', 'MD'), ('begin', 'VB'),
('building', 'VBG'), ('a', 'DT'), ('new', 'JJ'), ('structure', 'NN'),
('next', 'JJ'), ('month', 'NN'), ('.', '.')]]
```

Вот основное правило расшифровки тегов частей речи: теги, обозначающие существительные (nouns), начинаются с `N`, глаголы (verbs) — с `V`, прилагательные (adjectives) — с `J`, наречия (adverbs) – с `R`. Все остальные теги определяют некоторые структурные элементы. Полный список тегов можно найти по адресу: http://bit.ly/2JfUOrq.

В NLTK есть несколько дополнительных инструментов маркировки частями речи (например, `DefaultTagger`, `RegexpTagger`, `UnigramTagger`, `BrillTagger`). Поддерживается также возможность использовать их комбинации, например,

[1] Старое строение запланировано к сносу. Подрядчики начнут строительство нового здания в следующем месяце. — *Примеч. пер.*

`BrillTagger` позволяет улучшить первоначальную маркировку тегами, применяя правила преобразования Брилла (Brill).

Промежуточный анализ корпуса

Теперь у нашего `HTMLCorpusReader` есть все необходимые методы для декомпозиции документов, которые понадобятся нам в последующих главах. В главе 2 мы снабдили наш инструмент чтения корпуса методом `sizes()`, позволяющим оценить, как изменяется корпус с течением времени. Теперь мы можем добавить новый метод, `describe()`, который даст нам возможность выполнить промежуточный анализ изменения состава категорий, словаря и сложности корпуса.

Первым делом `describe()` запускает секундомер и инициализирует два распределения частот: первое, `counts`, предназначено для подсчета вложенных структур в документах, а второе, `tokens`, предназначено для хранения словаря. Частоту распределений и их применение мы подробно обсудим в главе 7. В процессе анализа мы подсчитаем количество абзацев, предложений и слов, а также сохраним каждую уникальную лексему в словаре. Затем подсчитаем количество файлов и категорий в корпусе и вернем словарь со статистической сводкой о корпусе, содержащий: общее количество файлов и категорий; общее количество абзацев, предложений и слов; количество уникальных лексем; лексическое разнообразие — как отношение числа уникальных лексем к общему их количеству; среднее число абзацев в документе; среднее число предложений в абзаце; общее время обработки:

```
import time

    def describe(self, fileids = None, categories = None):
        """
        Выполняет обход содержимого корпуса и возвращает
        словарь с разнообразными оценками, описывающими
        состояние корпуса.
        """
        started = time.time()

        # Структуры для подсчета.
        counts = nltk.FreqDist()
        tokens = nltk.FreqDist()

        # Выполнить обход абзацев, выделить лексемы и подсчитать их
        for para in self.paras(fileids, categories):
            counts['paras'] + = 1

            for sent in para:
                counts['sents'] + = 1
```

```
        for word, tag in sent:
            counts['words'] + = 1
            tokens[word] + = 1

# Определить число файлов и категорий в корпусе
n_fileids = len(self.resolve(fileids, categories) or self.fileids())
n_topics = len(self.categories(self.resolve(fileids, categories)))

# Вернуть структуру данных с информацией
return {
    'files':  n_fileids,
    'topics': n_topics,
    'paras':  counts['paras'],
    'sents':  counts['sents'],
    'words':  counts['words'],
    'vocab':  len(tokens),
    'lexdiv': float(counts['words']) / float(len(tokens)),
    'ppdoc':  float(counts['paras']) / float(n_fileids),
    'sppar':  float(counts['sents']) / float(counts['paras']),
    'secs':   time.time() - started,
}
```

С ростом корпуса по мере сбора новых данных, предварительной обработки и сжатия метод `describe()` позволит нам вычислять эти характеристики и видеть, как они меняются. Он может стать важным инструментом мониторинга и помочь выявить проблемы в приложении; модели машинного обучения обычно ожидают постоянства определенных характеристик, таких как лексическое разнообразие и количество абзацев на один документ, существенное изменение которых почти наверняка отразится на качестве модели. То есть метод `describe()` можно использовать для определения величины изменений в корпусе, чтобы своевременно запустить последующую векторизацию и перестройку модели.

Трансформация корпуса

Теперь наш объект может в потоковом режиме читать документы из корпуса, выполняя этапы извлечения контента, разделения на абзацы, сегментации на предложения, лексемизации слов, маркировку тегами частей речи, и передавать обработанные документы в модели машинного обучения, как показано на рис. 3.3.

К сожалению, эта обработка обходится недешево. Для небольших корпусов или когда есть возможность выделить для предварительной обработки много виртуальных машин, простого объекта чтения, такого как `HTMLCorpusReader`, мо-

жет быть достаточно. Но для корпуса с 300 000 новостей в формате HTML этап предварительной обработки может занять более 12 часов. Мы определенно не можем позволить себе делать это каждый раз, когда запускаем моделирование или тестируем новый набор гиперпараметров.

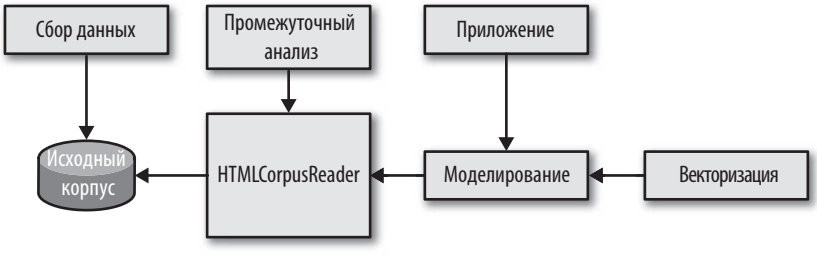


Рис. 3.3. Конвейер предварительной обработки корпуса

На практике эта проблема решается добавлением двух дополнительных классов: класса `Preprocessor`, использующего `HTMLCorpusReader` для подготовки исходного корпуса к сохранению в виде промежуточного артефакта, и класса `PickledCorpusReader`, читающего трансформированные документы с диска и в потоковом режиме передающего их на последующие этапы векторизации и анализа, как показано на рис. 3.4.

Рис. 3.4. Конвейер с промежуточным сохранением предварительно обработанного корпуса

Предварительная обработка и сохранение промежуточного артефакта

В этом разделе мы напишем класс `Preprocessor`, принимающий экземпляр `HTMLCorpusReader`, выполняющий предварительную обработку и записывающий новый текстовый корпус на диск, как показано на рис. 3.5. Именно этот новый корпус мы будем использовать для дальнейшего анализа текста.

Сначала напишем основное определение нового класса, `Preprocessor`, который обертывает объект чтения корпуса и выполняет процедуру лексемизации наших документов и маркировки их частями речи. Объекты этого класса должны инициализироваться исходным корпусом, строкой пути к его корневому каталогу и целевым корпусом, строкой пути к каталогу, куда должен быть сохранен

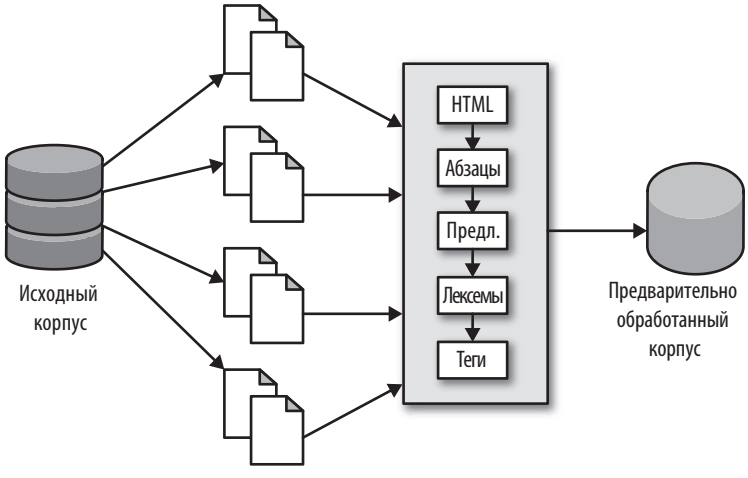


Рис. 3.5. Этап промежуточной обработки и получение трансформированного корпуса

результат предварительной обработки. Метод `fileids()` обеспечивает удобный доступ к свойству `fileids` объекта `HTMLCorpusReader`, а `abspath()` будет возвращать абсолютный путь к каждому файлу целевого корпуса, соответствующему файлу исходного корпуса:

```
import os

class Preprocessor(object):
    """
    Обертывает `HTMLCorpusReader` и выполняет лексемизацию
    с маркировкой частями речи.
    """
    def __init__(self, corpus, target = None, **kwargs):
        self.corpus = corpus
        self.target = target

    def fileids(self, fileids = None, categories = None):
        fileids = self.corpus.resolve(fileids, categories)
        if fileids:
            return fileids
        return self.corpus.fileids()

    def abspath(self, fileid):
        # Найти путь к каталогу относительно корня исходного корпуса.
        parent = os.path.relpath(
            os.path.dirname(self.corpus.abspath(fileid)), self.corpus.root
        )

        # Выделить части пути для реконструирования
```

```
        basename = os.path.basename(fileid)
        name, ext = os.path.splitext(basename)

        # Сконструировать имя файла с расширением .pickle
        basename = name + '.pickle'

        # Вернуть путь к файлу относительно корня целевого корпуса.
        return os.path.normpath(os.path.join(self.target, parent, basename))
```

Далее добавим метод `tokenize()`, который для заданного исходного документа выполняет сегментацию, лексемизацию и маркировку частями речи, используя методы, импортированные из NLTK в предыдущем разделе. Этот метод возвращает для каждого документа генератор абзацев со списками предложений, которые, в свою очередь, хранят списки частей речи маркированных лексем:

```
from nltk import pos_tag, sent_tokenize, wordpunct_tokenize

...

    def tokenize(self, fileid):
        for paragraph in self.corpus.paras(fileids = fileid):
            yield [
                pos_tag(wordpunct_tokenize(sent))
                for sent in sent_tokenize(paragraph)
            ]
```

Постепенно конструируя текстовые данные с необходимой организацией (списки документов, состоящих из списков абзацев, которые сами являются списками предложений, где каждое предложение — это список кортежей с лексемами и тегами), мы добавляем в исходный текст гораздо больше содержимого, чем удаляем. По этой причине мы должны быть готовы применить сжатие для экономии дискового пространства.

Запись в сжатый архив

Есть несколько вариантов преобразования и сохранения предварительно обработанного корпуса, но мы предпочитаем использовать для этого модуль `pickle`. Для этого мы реализуем итератор, который загружает документы в память по одному, преобразует их в целевую структуру данных и записывает строковое представление этой структуры в маленький файл на диск. Это строковое представление не предназначено для чтения человеком, оно легко сжимается, легко загружается, с легкостью сериализуется и десериализуется и благодаря всему этому обеспечивает высокую эффективность.

Для сохранения трансформированных документов добавим метод `process()`. После определения каталогов, где хранятся исходные и файлы и куда должны сохраняться их обработанные и сжатые версии, мы создаем временную переменную `document` и записываем в нее список списков списков[1] с нашей структурой кортежей. Затем сериализуем документ, записываем его на диск с максимальной степенью сжатия и удаляем из памяти перед переходом к следующему файлу, чтобы не занимать память ненужными данными:

```
import pickle
...

    def process(self, fileid):
        """
        Вызывается для одного файла, проверяет местоположение на диске,
        чтобы гарантировать отсутствие ошибок, использует +tokenize()+ для
        предварительной обработки и записывает трансформированный документ
        в виде сжатого архива в заданное место.
        """
        # Определить путь к файлу для записи результата.
        target = self.abspath(fileid)
        parent = os.path.dirname(target)

        # Убедиться в существовании каталога
        if not os.path.exists(parent):
            os.makedirs(parent)

        # Убедиться, что parent — это каталог, а не файл
        if not os.path.isdir(parent):
            raise ValueError(
                "Please supply a directory to write preprocessed data to."
            )

        # Создать структуру данных для записи в архив
        document = list(self.tokenize(fileid))

        # Записать данные в архив на диск
        with open(target, 'wb') as f:
            pickle.dump(document, f, pickle.HIGHEST_PROTOCOL)

        # Удалить документ из памяти
        del document

        # Вернуть путь к целевому файлу
        return target
```

Наш метод `process()` будет многократно вызываться из следующего метода `transform()`:

[1] Это не опечатка!

```
...
    def transform(self, fileids = None, categories = None):
        # Создать целевой каталог, если его еще нет
        if not os.path.exists(self.target):
            os.makedirs(self.target)

        # Получить имена файлов для обработки
        for fileid in self.fileids(fileids, categories):
            yield self.process(fileid)
```

В главе 11 мы исследуем методы распараллеливания этого метода `transform()`, чтобы обеспечить высокую скорость предварительной обработки и сохранения промежуточного корпуса.

Чтение предварительно обработанного корпуса

После предварительной обработки и сжатия корпуса у нас появляется возможность быстро обращаться к данным без повторного применения методов лексемизации или анализа строк — просто загружая структуры данных на языке Python и экономя массу времени и сил.

Для чтения предварительно обработанного корпуса нам потребуется класс `PickledCorpusReader`, использующий `pickle.load()` для быстрого извлечения структур данных на Python по одному документу за раз. Этот объект чтения обладает всеми возможностями `HTMLCorpusReader` (потому что наследует его), но, поскольку не обращается непосредственно к исходному тексту, действует во много раз быстрее. Здесь мы переопределили метод `docs()`, унаследованный от `HTMLCorpusReader`, заменив его версией, которая знает, как загружать документы из архивов:

```
import pickle

PKL_PATTERN = r'(?!\.)[a-z_\s]+/[a-f0-9]+\.pickle'

class PickledCorpusReader(HTMLCorpusReader):

    def __init__(self, root, fileids = PKL_PATTERN, **kwargs):
        if not any(key.startswith('cat_') for key in kwargs.keys()):
            kwargs['cat_pattern'] = CAT_PATTERN
        CategorizedCorpusReader.__init__(self, kwargs)
        CorpusReader.__init__(self, root, fileids)

    def docs(self, fileids = None, categories = None):
        fileids = self.resolve(fileids, categories)
        # Загружать документы в память по одному.
        for path in self.abspaths(fileids):
            with open(path, 'rb') as f:
                yield pickle.load(f)
```

Каждый документ, прошедший предварительную обработку, хранится как список Python-абзацев, поэтому метод `paras()` можно реализовать иначе:

```
...
    def paras(self, fileids = None, categories = None):
        for doc in self.docs(fileids, categories):
            for para in doc:
                yield para
```

Каждый абзац тоже является списком — списком предложений, поэтому метод `sents()` можно реализовать аналогично, чтобы он возвращал все предложения из запрошенного документа. Обратите внимание, что в этом методе, так же как в `docs()` и `paras()`, аргументы `fileids` и `categories` позволяют явно определить, из каких документов извлекать информацию; если оба аргумента имеют значение `None`, тогда метод вернет предложения из всего корпуса. Чтобы прочитать единственный документ, нужно передать в аргументе `fileids` путь к файлу документа относительно корня корпуса.

```
...
    def sents(self, fileids = None, categories = None):
        for para in self.paras(fileids, categories):
            for sent in para:
                yield sent
```

Предложения — это списки кортежей лексем и тегов `(token, tag)`, поэтому нам понадобятся два метода для доступа к упорядоченному множеству слов, составляющих документ или документы. Первый метод, `tagged()`, возвращает лексемы с их тегами, второй, `words()`, — только лексемы.

```
...
    def tagged(self, fileids = None, categories = None):
        for sent in self.sents(fileids, categories):
            for tagged_token in sent:
                yield tagged_token

    def words(self, fileids = None, categories = None):
        for tagged in self.tagged(fileids, categories):
            yield tagged[0]
```

Объект `PickledCorpusReader` значительно упрощает работу с огромными корпусами. Предварительную обработку данных можно распараллелить, используя модуль `multiprocessing` из библиотеки Python (как будет показано в главе 11), но при конструировании моделей сканирование всех документов перед векторизацией должно осуществляться последовательно. Теоретически этот процесс тоже можно распараллелить, но такой подход редко используется на

практике из-за экспериментальной природы исследовательского моделирования. Использование сериализации в архивы значительно увеличивает скорость процесса моделирования и исследования!

В заключение

В этой главе вы узнали, как в подготовке к машинному обучению выполнить предварительную обработку корпуса путем сегментации, лексемизации и маркировки частями речи. В следующей главе мы познакомимся с терминологией машинного обучения и обсудим отличия машинного обучения на текстах от статистического программирования, применявшегося нами в предыдущих приложениях.

Сначала мы посмотрим, как формулируются задачи машинного обучения, когда входными данными является текст, то есть когда мы работаем в многомерном пространстве, где наши экземпляры являются полными документами, а в число признаков могут входить атрибуты уровня слов, такие как словарь и частота лексем, а также метаданные, такие как автор, дата и источник. Следующим нашим шагом станет подготовка результатов предварительной обработки текста к машинному обучению путем их кодирования в векторы. Мы рассмотрим несколько методов кодирования в векторы и обсудим порядок их встраивания в конвейер, чтобы обеспечить возможность систематической загрузки, нормализации и извлечения признаков. Наконец, мы обсудим возможность объединения извлеченных признаков для проведения более сложных видов анализа и моделирования. Эти шаги помогут нам извлечь значимые шаблоны из корпуса и использовать их для прогнозирования на новых данных, которые модель еще не видела.

4 Конвейеры векторизации и преобразования

Алгоритмы машинного обучения действуют в пространстве числовых признаков, ожидая получить на входе двумерный массив, строки которого представляют экземпляры, а столбцы — признаки. Чтобы выполнить машинное обучение на тексте, нужно преобразовать документы в векторные представления, к которым можно применить численное машинное обучение. Этот процесс называется *извлечением признаков*, или просто *векторизацией*, и по сути является первым шагом на пути к анализу естественного языка.

Преобразование документов в численный вид дает возможность осуществлять их анализ и создавать *экземпляры*, с которыми смогут работать алгоритмы машинного обучения. В анализе текста экземплярами являются целые документы или высказывания, которые могут иметь самые разные размеры, от коротких цитат или твитов до целых книг. Но сами векторы всегда имеют одинаковую длину. Каждое свойство в векторном представлении — это *признак*. В случае с текстом признаки представляют атрибуты и свойства документов — включая их содержимое и метаатрибуты, такие как длина документа, имя автора, источник и дата публикации. Вместе все признаки документа описывают многомерное пространство признаков, к которому могут применяться методы машинного обучения.

По этой причине мы теперь должны в корне изменить свой взгляд на естественный язык — перейти от предложений и слов к точкам в многомерном семантическом *пространстве*. Точки в пространстве могут располагаться близко или далеко друг от друга, образовывать тесные группы или распределяться равномерно. Таким образом, семантическое пространство формируется так, что документы, близкие по смыслу, располагаются рядом, а разные — далеко

друг от друга. Кодируя сходство как расстояние, мы можем начать порождать первичные компоненты документов и определять границы решения в нашем семантическом пространстве.

Простейшим представлением семантического пространства является модель *мешок слов*, основная идея которой состоит в том, что смысл и сходство кодируются в виде словаря. Например, статьи в «Википедии» о бейсболе и Джордже Хермане «Бейбе» Руте (Babe Ruth)[1] наверняка будут иметь большое сходство. В обеих будет присутствовать множество одинаковых слов, а кроме того, они будут существенно отличаться словарным составом от статей о приготовлении запеканки или о смягчении денежно-кредитной политики. Несмотря на простоту, эта модель очень эффективна и может служить отправной точкой для более сложных моделей.

В этой главе мы продемонстрируем порядок векторизации для объединения лингвистических приемов из NLTK с приемами машинного обучения из Scikit-Learn и Gensim, и создания собственных *преобразователей* для использования в повторяемых *конвейерах* многократного использования. К концу главы вы будете готовы преобразовать документы из нашего предварительно обработанного корпуса в пространство модели, чтобы получить возможность делать прогнозы.

Слова в пространстве

Для векторизации корпуса в мешок слов (Bag-Of-Words, BOW) представим каждый документ в виде вектора, длина которого равна размеру словаря корпуса. Мы можем упростить вычисления, отсортировав в алфавитном порядке лексемы в векторе, как показано на рис. 4.1. Также можно сформировать словарь, отображающий позиции лексем в векторе. В любом случае мы получим векторное отображение корпуса, позволяющее однозначно представить каждый документ.

Какая информация должна храниться в каждом элементе вектора, представляющего документ? В следующих нескольких разделах мы исследуем разные варианты, каждый из которых расширяет или изменяет базовую модель мешка слов для описания семантического пространства. Мы рассмотрим четыре вида кодирования вектора — частотное, прямое, TF–IDF и распределенное представление — и обсудим их реализацию с применением Scikit-Learn, Gensim и NLTK. В качестве образца будет использоваться маленький корпус из трех предложений.

[1] Профессиональный американский бейсболист. — *Примеч. пер.*

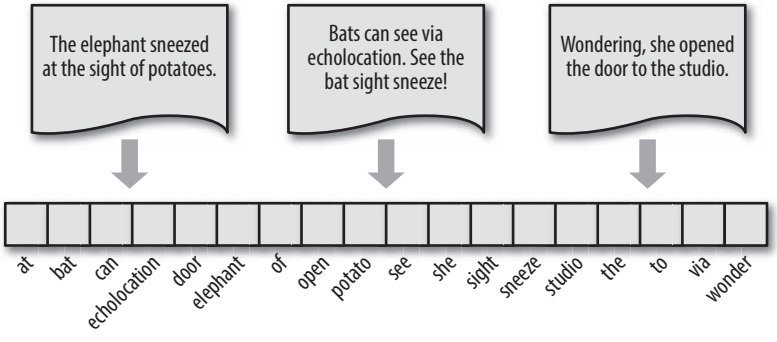


Рис. 4.1. Представление документов в виде векторов

Для начала создадим список документов и выполним их лексемизацию, чтобы потом приступить к векторизации. Следующий метод, `tokenize`, выполняет упрощенную лексемизацию, отбрасывает знаки препинания, используя набор символов `string.punctuation`, и преобразует оставшиеся символы в нижний регистр. Дополнительно эта функция производит свертку свойств с помощью `SnowballStemmer`, удаляя суффиксы, например, обозначающие множественное число («bats» и «bat» интерпретируются как одна и та же лексема). Примеры в следующем разделе будут использовать этот демонстрационный корпус и некоторые из них — этот метод лексемизации.

```
import nltk
import string

def tokenize(text):
    stem = nltk.stem.SnowballStemmer('english')
    text = text.lower()

    for token in nltk.word_tokenize(text):
        if token in string.punctuation: continue
        yield stem.stem(token)

corpus = [
    "The elephant sneezed at the sight of potatoes.",
    "Bats can see via echolocation. See the bat sight sneeze!",
    "Wondering, she opened the door to the studio.",
]
```

Выбор конкретного способа векторизации в значительной степени определяется предметным пространством. Аналогично выбор реализации — NLTK, Scikit-Learn или Gensim — диктуется требованиями приложения. Например, NLTK предлагает множество методов, которые прекрасно подходят для обработки текстовых данных, но влекут за собой слишком много зависимостей.

Библиотека Scikit-Learn создавалась не для обработки текста, но предлагает надежный API и множество разных удобств (которые мы исследуем далее в этой главе), особенно ценных в прикладном контексте. Gensim может сериализовать словари и ссылки в формат Matrix Market, что позволяет использовать данные на разных платформах. Однако, в отличие от Scikit-Learn, Gensim не выполняет никаких операций с документами для их лексемизации или стемминга.

По этим причинам, представляя каждый из четырех методов кодирования, мы покажем несколько вариантов реализации: «С применением NLTK», «С применением Scikit-Learn» и «С применением Gensim».

Частотные векторы

Простейший способ векторизации заключается в заполнении вектора частотами появлений слов в документе. В этой схеме кодирования каждый документ представляется как мультимножество составляющих его лексем, а значением для каждой позиции слова в векторе служит счетчик соответствующего слова. Значения могут быть простыми целочисленными счетчиками, как на рис. 4.2, или взвешиваться общим количеством слов в документе.

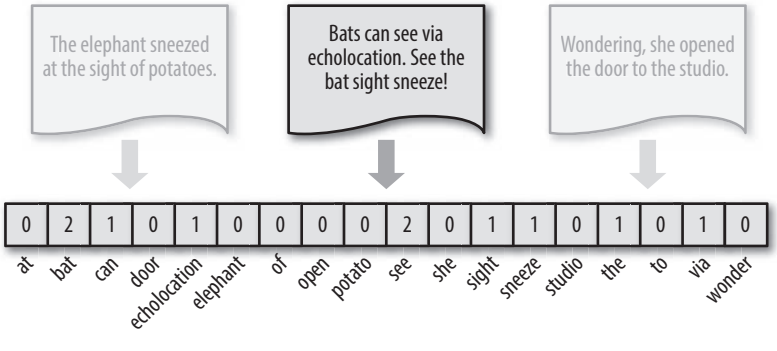


Рис. 4.2. Представление частот лексем в виде вектора

С применением NLTK

NLTK принимает признаки в виде объекта словаря `dict`, ключами которого являются имена признаков, а значениями — логические или числовые величины. Чтобы преобразовать документы в такое представление, напишем функцию `vectorize`, которая создает словарь, роль ключей в котором будут играть лексемы, а роль значений — количества их вхождений в документ.

Используем объект `defaultdict`, позволяющий указать, какое значение должен вернуть словарь при обращении к несуществующему ключу. Вызывая конструктор как `defaultdict(int)`, мы указываем, что по умолчанию должно возвращаться значение `0`, и тем самым создаем простой словарь-счетчик. Мы можем применить эту функцию ко всем документам в корпусе с помощью `map`, как показано в последней строке следующего листинга, и создать векторизованное представление документов.

```
from collections import defaultdict

def vectorize(doc):
    features = defaultdict(int)
    for token in tokenize(doc):
        features[token] + = 1
    return features

vectors = map(vectorize, corpus)
```

С применением Scikit-Learn

В модели `sklearn.feature_extraction` есть преобразователь `CountVectorizer` с собственными методами лексемизации и нормализации. Метод `fit` этого преобразователя принимает итерируемую последовательность или список строк или объектов файлов и создает словарь корпуса. Метод `transform` преобразует каждый отдельный документ в разреженный массив, роль индексов в котором играют кортежи с идентификаторами документов и лексем из словаря, а роль значений — счетчики лексем:

```
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer()
vectors = vectorizer.fit_transform(corpus)
```

Векторы могут получаться очень разреженными, особенно с увеличением размера словаря, что может оказывать существенное влияние на производительность моделей машинного обучения. Поэтому для очень больших корпусов рекомендуется использовать преобразователь `HashingVectorizer` из Scikit-Learn, который поддерживает трюк с хешированием для поиска строк с лексемами и извлечения соответствующих им индексов. Он экономно использует память и хорошо подходит для обработки больших наборов данных, так как не хранит весь словарь целиком и тем самым ускоряет сохранение и обучение. Однако обратное преобразование (из вектора в документ) невозможно, из-за чего вероятны конфликты и не поддерживается обратное взвешивание частоты документов.

С применением Gensim

Частотный кодировщик в Gensim называется `doc2bow`. Чтобы задействовать его, сначала нужно создать объект словаря `Dictionary` из Gensim, отображающий лексемы в индексы в порядке их следования в документе (благодаря чему устраняются накладные расходы на сортировку). Объект словаря можно сохранить на диск или загрузить с диска и реализовать библиотеку `doc2bow`, которая принимает *лексемизированный документ* и возвращает разреженную матрицу кортежей `(id, count)`, где `id` — идентификатор лексемы в словаре. Метод `doc2bow` принимает только один экземпляр документа, поэтому для векторизации всего корпуса мы используем генератор списков, загружающий лексемизированные документы в память, и избавляемся от риска исчерпать генератор:

```
import gensim

corpus = [tokenize(doc) for doc in corpus]
id2word = gensim.corpora.Dictionary(corpus)
vectors = [
    id2word.doc2bow(doc) for doc in corpus
]
```

Прямое кодирование

Игнорируя грамматику и относительные позиции слов в документах, частотные методы кодирования страдают проблемой *вытянутого хвоста*, или распределения Ципфа (Zipfian distribution), характерной для естественных языков. В результате лексемы, встречающиеся очень часто, оказываются на порядки более «значимыми», чем другие, встречающиеся намного реже. Это может оказывать существенное влияние на некоторые модели (например, обобщенные линейные модели), которые предполагают нормальное распределение признаков.

Решением этой проблемы является *прямое кодирование*, метод логической векторизации, который помещает в соответствующий элемент вектора значение `true` (`1`), если лексема присутствует в документе, и `false` (`0`) — если отсутствует. Иными словами, каждый элемент вектора при прямом кодировании говорит о наличии или отсутствии лексемы в описываемом документе, как показано на рис. 4.3.

Прямое кодирование ослабляет проблему дисбаланса распределения лексем, упрощая документ до составляющих его компонентов. Этот метод наиболее эффективен для коротких документов (предложений, твитов), содержащих не так много повторяющихся элементов, и обычно применяется в моделях, имеющих хорошие сглаживающие свойства. Прямое кодирование также часто применяется в искусственных нейронных сетях, функции активации которых требуют подачи на вход значений из дискретного диапазона `[0,1]` или `[-1,1]`.

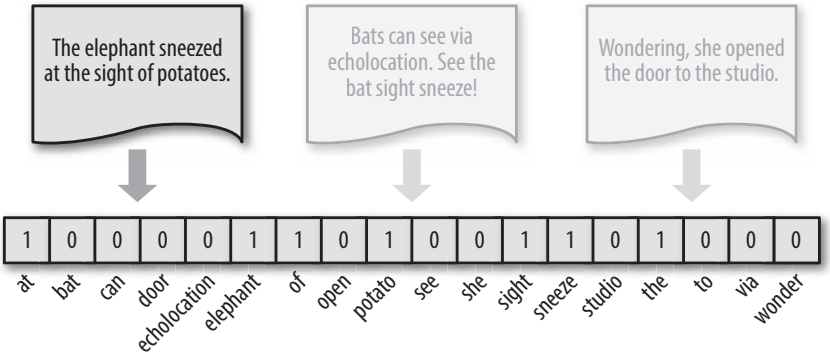


Рис. 4.3. Прямое кодирование

С применением NLTK

Реализация прямого кодирования в NLTK основана на словаре, ключами которого являются лексемы, а значениями — логическая величина `True`:

```
def vectorize(doc):
    return {
        token: True
        for token in doc
    }

vectors = map(vectorize, corpus)
```

Словарь в этом случае действует как простая разреженная матрица, потому что не требует отображать каждое отсутствующее слово в значение `False`. Кроме того, вместо логических допускается использовать целочисленные значения: `1` для обозначения присутствия и `0` — отсутствия.

С применением Scikit-Learn

В Scikit-Learn прямое кодирование реализуется с помощью преобразователя `Binarizer` из модуля `preprocessing`. Он принимает только числовые данные, поэтому перед прямым кодированием текст нужно преобразовать в численное пространство с помощью `CountVectorizer`. Класс `Binarizer` использует пороговое значение (`0` по умолчанию), поэтому все элементы вектора, которые меньше или равны порогу, получат нулевое значение, а элементы больше порога — получат значение `1`. То есть по умолчанию `Binarizer` преобразует все значения частот больше нуля в `1`, сохраняя в неприкосновенности нулевые значения.

```
from sklearn.preprocessing import Binarizer

freq = CountVectorizer()
corpus = freq.fit_transform(corpus)

onehot = Binarizer()
corpus = onehot.fit_transform(corpus.toarray())
```

Дополнительно можно вызвать метод `corpus.toarray()`; он преобразует разреженную матрицу в плотную. Для корпусов с большими словарями предпочтительнее использовать представление в виде разреженной матрицы. Обратите внимание, что в примере выше мы могли бы также использовать `CountVectorizer(binary = True)`, чтобы получить то же самое прямое кодирование без применения `Binarizer`.

Несмотря на свое имя, преобразователь `OneHotEncoder` из модуля `sklearn.preprocessing` не предназначен для решения этой задачи. `OneHotEncoder` интерпретирует каждый компонент вектора (столбец) как независимую категориальную переменную, расширяя размерность вектора для каждого наблюдаемого значения в каждом столбце. В данном случае компоненты `(sight, 0)` и `(sight, 1)` интерпретировались бы как два категориальных измерения, а не как единственный компонент вектора.

С применением Gensim

В библиотеке Gensim нет специализированного метода, реализующего прямое кодирование, но ее метод `doc2bow` возвращает список кортежей, которыми мы можем оперативно управлять. Дополнив код из примера частотной векторизации с применением Gensim в предыдущем разделе, мы можем получить векторы прямого кодирования из словаря `id2word`. Для этого добавим вложенный генератор списков, преобразующий список кортежей, возвращаемый методом `doc2bow`, в список кортежей `(token_id, 1)` и во внешнем генераторе списков применим это преобразование ко всем документам в корпусе:

```
corpus = [tokenize(doc) for doc in corpus]
id2word = gensim.corpora.Dictionary(corpus)
vectors = [
    [(token[0], 1) for token in id2word.doc2bow(doc)]
    for doc in corpus
]
```

Прямое кодирование представляет сходство и различие на уровне *документов*, но, поскольку все слова оказываются равноудаленными, этот метод не в состоянии закодировать сходство на уровне слов. Кроме того, поскольку

все слова равноудалены, *форма слов* приобретает особую важность; лексемы «trying» (попытка) и «try» (попытаться) будут так же далеки друг от друга, как несвязанные слова, такие как «red» (красный) и «bicycle» (велосипед)! Нормализация лексем и их приведение к одному классу посредством *стемминга* или *лемматизации*, о которых рассказывается далее в этой главе, гарантируют, что слова, различающиеся числом, регистром символов, родом, падежом, временем и т. д., будут интерпретироваться как единственный компонент вектора, что обеспечит уменьшение размерности пространства признаков и увеличение производительности моделей.

Частота слова — обратная частота документа

Представление «мешок слов», о котором рассказывалось до сих пор, описывает документы без учета контекста корпуса. Более удачный подход основывается на сопоставлении относительной частоты или редкости лексем в документе с их частотой в других документах. Главная идея этого подхода состоит в том, что основной смысл документа закодирован в более редких словах. Например, в корпусе на спортивную тему такие лексемы, как «судья», «база» и «землянка» (скамейка запасных) появляются чаще в документах на бейсбольную тему, тогда как другие слова, такие как «бег», «очки» и «игра», чаще встречающиеся в остальных документах в корпусе, играют менее важную роль.

Метод кодирования TF–IDF (*Term Frequency–Inverse Document Frequency* — частота слова–обратная частота документа) нормализует частоту лексем в документе с учетом содержимого в остальном корпусе. Этот метод придает больше веса терминам, релевантным для конкретного экземпляра, как показано на рис. 4.4, где лексема `studio` имеет наивысшую релевантность для данного документа, потому что появляется только в нем.

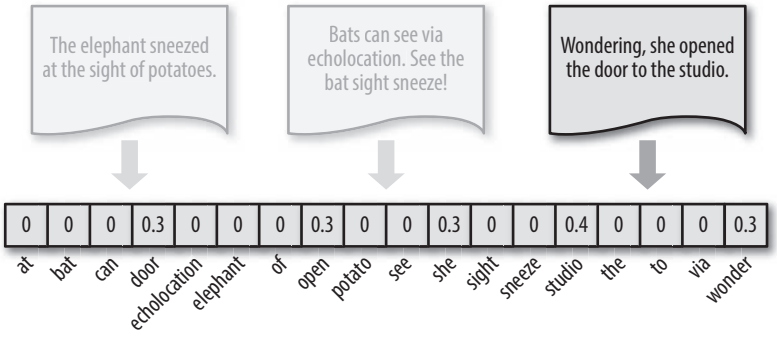


Рис. 4.4. Метод кодирования TF–IDF

Оценка TF–IDF вычисляется на уровне лексем, поэтому релевантность лексемы в документе измеряется масштабированной частотой появления лексемы в документе, нормализованной обратной масштабированной частотой появления лексемы во всем корпусе.

ВЫЧИСЛЕНИЕ TF–IDF

Частота слова для заданной лексемы в документе, $tf(t, d)$ может быть логической частотой (как в прямом кодировании, 1, если t присутствует в d, и 0 в противном случае) или счетчиком. Однако обычно частота слова и обратная частота документа масштабируются логарифмически, чтобы избежать предвзятости в пользу наиболее длинных документов или слов, появляющихся особенно часто: $tf(t, d) = 1 + \log f_{t,d}$.

Аналогично обратную частоту документа для слова в заданном наборе документов можно перевести в логарифмический масштаб: $idf(t, D) = \log 1 + N/n_t$, где N — число документов, а n_t — число вхождений слова t во всех документах. После этого полная оценка TF–IDF вычисляется как $tfidf(t, d, D) = tf(t, d) \times idf(t, D)$.

Поскольку отношение логарифмической функции idf больше или равно 1, величина TF–IDF всегда больше или равна нулю. Соответственно, согласно нашим представлениям, чем ближе к 1 оценка TF–IDF слова, тем более информативным является это слово для данного документа. И наоборот, чем ближе эта оценка к нулю, тем менее информативно слово.

С применением NLTK

Чтобы реализовать векторизацию текста таким способом с применением NLTK, используем класс `TextCollection`, обертывающий список документов или корпус, состоящий из одного или нескольких документов. Этот класс поддерживает подсчет вхождений, вычисление оценки согласованности (конкордантности), определение устойчивых словосочетаний и, что особенно важно, вычисление `tf_idf`.

Так как для вычисления TF–IDF необходим весь корпус, наша новая версия принимает не один, а все документы. После применения функции лексемизации и создания коллекции слов функция выполняет обход всех документов в корпусе и возвращает словарь, ключами которого являются слова, а значениями — их оценки TF–IDF для данного конкретного документа.

```
from nltk.text import TextCollection

def vectorize(corpus):
    corpus = [tokenize(doc) for doc in corpus]
    texts = TextCollection(corpus)

    for doc in corpus:
        yield {
            term: texts.tf_idf(term, doc)
            for term in doc
        }
```

С применением Scikit-Learn

В библиотеке Scikit-Learn есть преобразователь `TfidfVectorizer`, в модуле `feature_extraction.text`, предназначенный для векторизации документов с оценками TF–IDF. Внутренне `TfidfVectorizer` вызывает преобразователь `CountVectorizer`, который мы использовали для превращения мешка слов в количества вхождений лексем, а затем `TfidfTransformer`, нормализующий эти количества обратной частотой документа.

На входе `TfidfVectorizer` принимает последовательность имен файлов, объектов файлов или строк с коллекцией исходных документов, подобно преобразователю `CountVectorizer`. В результате применяются методы по умолчанию лексемизации и предварительной обработки, если не указаны другие функции. Результатом работы преобразователя является разреженная матрица вида `((doc, term), tfidf)`, где каждый ключ является парой «документ/лексема», а значением служит оценка TF–IDF.

```
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf = TfidfVectorizer()
corpus = tfidf.fit_transform(corpus)
```

С применением Gensim

В библиотеке Gensim есть структура данных `TfidfModel`, которая так же, как объект `Dictionary` хранит лексемы с их индексами в векторе в порядке их следования в документе, но, кроме того, хранит частоту лексем в корпусе для возможной последующей векторизации документов. Как было показано выше, Gensim позволяет нам применять свой метод лексемизации, ожидая получить корпус в виде списка списков лексем. В первую очередь мы должны сконструировать словарь лексем и на его основе создать экземпляр `TfidfModel`, который вычислит нормализованную обратную частоту документа. После этого можно извлечь представление TF–IDF для каждого вектора с помощью `getitem`, ис-

пользуя синтаксис обращения к словарям, а затем к каждому документу применить метод `doc2bow` словаря.

```
corpus  = [tokenize(doc) for doc in corpus]
lexicon = gensim.corpora.Dictionary(corpus)
tfidf   = gensim.models.TfidfModel(dictionary = lexicon, normalize = True)
vectors = [tfidf[lexicon.doc2bow(doc)] for doc in corpus]
```

В библиотеке Gensim есть вспомогательные методы для записи словарей и моделей на диск в компактной форме, а это значит, что у нас есть удобная возможность сохранять модели TF–IDF и словари с целью последующего их использования для векторизации новых документов. Тот же результат (пусть и с немного большими усилиями) можно получить с помощью комбинации модуля `pickle` и Scikit-Learn. Вот как можно сохранить модель на диск средствами из библиотеки Gensim:

```
lexicon.save_as_text('lexicon.txt', sort_by_word = True)
tfidf.save('tfidf.pkl')
```

Первая инструкция сохранит словарь в текстовый файл в отсортированном виде, а вторая сохранит модель TF–IDF в виде сжатой разреженной матрицы. Обратите внимание, что объект `Dictionary` можно сохранить в более компактном двоичном формате вызовом его метода `save`, но `save_as_text` позволяет легко проверить и скорректировать сохраненный словарь вручную. Вот как можно загрузить модели с диска:

```
lexicon = gensim.corpora.Dictionary.load_from_text('lexicon.txt')
tfidf = gensim.models.TfidfModel.load('tfidf.pkl')
```

Одно из преимуществ методики TF–IDF — она естественным образом решает проблему *стоп-слов*, которые почти наверняка присутствуют во всех документах в корпусе (например, «a», «the», «of» и др.) и по этой причине получают очень маленький вес в данной схеме кодирования. Благодаря этому наибольшую значимость в модели TF–IDF приобретают относительно редкие слова. Как результат, TF–IDF широко применяется к моделям «мешок слов» и служит прекрасной отправной точкой во многих видах анализа естественного языка.

Распределенное представление

Частотное и прямое кодирование, а также кодирование методом TF–IDF позволяют нам преобразовывать документы в векторное пространство, но часто также бывает полезно закодировать сходства между документами в контексте того же векторного пространства. К сожалению, эти методы векторизации про-

изводят векторы с неотрицательными элементами, что не позволяет сравнивать документы, не имеющие общих лексем (потому что два вектора, косинус угла между которыми равен 1, будут считаться далекими друг от друга даже при очевидном семантическом сходстве).

Если в контексте приложения сходство между документами играет важную роль, текст можно закодировать в виде числовой последовательности методом распределенного представления, как показано на рис. 4.5. При таком подходе вектор документа является не простым отображением позиций лексем в их числовые значения, а набором признаков, определяющих сходство слов. Сложность пространства признаков (и длина вектора) определяется особенностями обучения этого представления и напрямую не связана с самим документом.

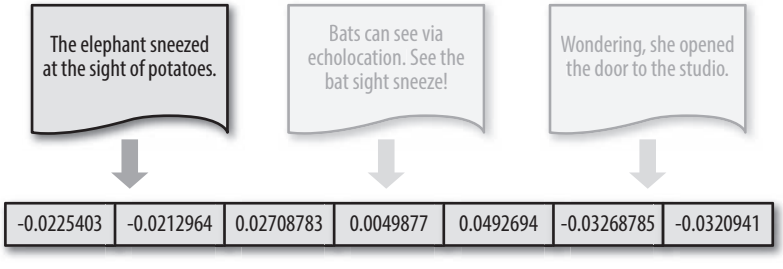


Рис. 4.5. Распределенное представление

Инструмент *word2vec*, созданный группой исследователей в Google, возглавляемой Томашем Миколовым (Tomáš Mikolov), реализует модель векторного представления слов (word embedding model) и позволяет создавать такого рода распределенные представления. Алгоритм *word2vec* обучает представления слов, опираясь на модель непрерывного мешка слов (Continuous Bag-of-Words, CBOW) или skip-gram, в результате чего слова в пространстве признаков оказываются рядом со схожими словами, в зависимости от их контекста. Например, реализация в Gensim использует сеть прямого распространения.

Алгоритм *doc2vec*[1] является расширением алгоритма *word2vec*. Он предлагает *paragraph vector* (вектор абзаца) — алгоритм обучения без учителя, который создает пространство признаков фиксированной длины из документов разной длины. Это представление пытается унаследовать семантические свойства слов, чтобы, например, слова «red» (красный) и «colorful» (разноцветный) считались более похожими друг на друга, чем на «river» (река) или «governance» (власть).

[1] Quoc V. Le and Tomas Mikolov, *Distributed Representations of Sentences and Documents* (2014), http://bit.ly/2GJBHjZ

Кроме того, алгоритм paragraph vector учитывает порядок слов в узком контексте, напоминая модель *n*-грамм. Комбинированный результат получается намного эффективнее, чем модели «мешок слов» или «мешок *n*-грамм», потому что лучше обобщает и имеет меньшую размерность, а благодаря фиксированной длине может использоваться в распространенных алгоритмах машинного обучения.

С применением Gensim

Ни NLTK, ни Scikit-Learn не реализуют получение такого векторного представления слов. Реализация в Gensim дает пользователям возможность обучить модели *word2vec* и *doc2vec* на своих корпусах, а также включает готовую модель, предварительно обученную на корпусе новостей Google.

Чтобы воспользоваться предварительно обученными моделями из библиотеки Gensim, вы должны загрузить двоичный файл модели, размер которого составляет около 1,5 Гбайт. Для приложений, где массивные зависимости нежелательны (например, если они выполняются на экземпляре AWS Lambda), этот подход может оказаться невозможным.

Для обучения своей модели можно использовать следующий алгоритм. Сначала, используя генератор списков, загрузить корпус в память. (Gensim поддерживает потоковую обработку, но такое решение поможет избежать исчерпания генератора.) Затем создать список объектов класса `TaggedDocument`, наследующего класс `LabeledSentence`, который, в свою очередь, реализует распределенное представление *word2vec*. Объекты `TaggedDocument` состоят из *слов* и *тегов*. Создать экземпляр `TaggedDocument` можно из списка лексем с единственным тегом, однозначно идентифицирующим экземпляр. В следующем примере мы пометили каждый документ как `"d{}".format(idx)`, например, `d0`, `d1`, `d2` и т. д.

После получения списка объектов `TaggedDocument` создается модель `Doc2Vec` и задается размер вектора, а также минимальный порог — лексемы, встречающиеся реже этого порога, игнорируются. Размер вектора (параметр `size`) обычно выбирается не меньше 5; в данном примере мы выбрали маленькое значение исключительно для простоты демонстрации. Минимальный порог (параметр `min_count`) в примере выбран равным нулю, чтобы гарантировать учет всех лексем, но на практике чаще минимальный порог устанавливается равным значению от 3 до 5, в зависимости от того, как много информации должна захватить модель. После создания модели производится обучение нейронной сети для получения векторных представлений, которые затем будут доступны через свойство `docvecs`.

```
from gensim.models.doc2vec import TaggedDocument, Doc2Vec

corpus = [list(tokenize(doc)) for doc in corpus]
corpus = [
    TaggedDocument(words, ['d{}'.format(idx)])
    for idx, words in enumerate(corpus)
]

model = Doc2Vec(corpus, size = 5, min_count = 0)
print(model.docvecs[0])
# [ 0.01797447 -0.01509272 0.0731937 0.06814702 -0.0846546 ]
```

При правильном использовании распределенные представления позволяют получить намного более качественные результаты, чем модели TF–IDF. Саму модель можно сохранить на диск и повторно обучить ее, что делает ее чрезвычайно гибкой для разных случаев использования. Однако для обучения на больших корпусах может потребоваться очень много времени и памяти, а результат может оказаться хуже, чем при использовании модели TF–IDF в совокупности с методом главных компонент (Principal Component Analysis, PCA) или сингулярного разложения (Singular Value Decomposition, SVD) для сокращения пространства признаков. Но, как бы то ни было, этот вид представлений оказался настоящим прорывом и помог добиться значительных улучшений в обработке текста в приложениях данных.

И снова, выбор способа векторизации (а также библиотеки, реализующей его) обычно зависит от конкретного случая применения, как показано в табл. 4.1.

Таблица 4.1. Обзор методов векторизации текста

Метод векторизации	Принцип действия	Хорошо подходит для	Недостатки
Частотный	Подсчет частоты вхождения лексем	Байесовские модели	Самые часто встречающиеся слова не всегда являются самыми информативными
Прямое кодирование	Определение логического признака присутствия лексемы (0, 1)	Нейронные сети	Все слова оказываются равноудаленными, поэтому очень важна нормализация
TF–IDF	Нормализация частоты лексем по документам	Приложения общего назначения	Умеренно часто встречающиеся слова могут быть не репрезентативными для темы документа
Распределенные представления	Кодирование сходства лексем на основе контекста	Моделирование сложных отношений	Большой объем вычислений, сложность масштабирования без применения дополнительных инструментов (например, Tensorflow)

Далее в этой главе мы рассмотрим объект `Pipeline` из библиотеки Scikit-Learn, помогающий упростить векторизацию с последующими этапами моделирования. По этой причине мы часто используем инструменты векторизации, совместимые с библиотекой Scikit-Learn. В следующем разделе мы обсудим организацию API этой библиотеки и покажем, как интегрировать инструменты векторизации в общий конвейер для создания полноценного (и настраиваемого в широких пределах!) ядра приложения машинного обучения на текстах.

Scikit-Learn API

Библиотека Scikit-Learn — это расширение пакета SciPy (scikit), основной целью которого является предоставление алгоритмов машинного обучения, а также инструментов и утилит, необходимых для успешного моделирования. Основой библиотеки является «API для машинного обучения», предоставляющий доступ к реализациям широкого спектра моделей через единый и удобный интерфейс. Благодаря такой организации Scikit-Learn можно использовать для одновременного обучения ошеломляющего разнообразия моделей, их оценки и сравнения и использования обученных моделей для предсказаний на новых данных. Библиотека Scikit-Learn имеет стандартизованный API, поэтому пользоваться ею легко и удобно, а создавать прототипы разных моделей для оценки можно простым изменением пары строк кода.

Интерфейс BaseEstimator

Сам API — объектно-ориентированный и определяется иерархией интерфейсов для разных задач машинного обучения. Корнем иерархии является `Estimator` — практически любой объект, способный обучаться на данных. Объекты `Estimator` реализуют алгоритмы классификации, регрессии и кластеризации. Но также могут включать широкий спектр методов преобразования данных, от сокращения размерности до извлечения признаков из исходных данных. По сути, `Estimator` играет роль интерфейса, и классы, реализующие его, должны реализовать два метода — `fit` и `predict` — как показано ниже:

```
from sklearn.base import BaseEstimator

class Estimator(BaseEstimator):

    def fit(self, X, y = None):
        """
        Принимает входные данные X и необязательные целевые данные y.
        Возвращает self.
```

```
        """
        return self

    def predict(self, X):
        """
        Принимает входные данные X и возвращает вектор предсказаний
        для каждой строки.
        """
        return yhat
```

Метод `Estimator.fit` настраивает состояние экземпляра `Estimator`, опираясь на данные для обучения, `X` и `y`. Обучающие данные `X` должны иметь форму матрицы — например, двумерный массив NumPy с формой `(n_samples, n_features)` или матрицу `DataFrame` Pandas, строки в которой являются экземплярами, а столбцы — признаками. Для обучения с учителем также можно передать одномерный массив NumPy, `y`, хранящий правильные метки. В процессе обучения внутреннее состояние экземпляра `Estimator` изменяется так, чтобы подготовить его к предсказаниям. Это состояние хранится в переменных экземпляра, имена которых обычно заканчиваются символом подчеркивания (например, `Estimator.coefs_`). Так как этот метод изменяет внутреннее состояние, он возвращает ссылку `self` на сам экземпляр, что позволяет составлять цепочки из вызовов методов.

Метод `Estimator.predict` создает предсказания, используя внутреннее состояние модели, обученной на данных `X`. На вход метода должна передаваться матрица с тем же количеством столбцов, что и в обучающей матрице, передававшейся в метод `fit`, которая может иметь любое количество строк. Этот метод возвращает вектор `yhat`, содержащий предсказания для всех строк во входных данных.

Наследование класса `BaseEstimator` из библиотеки Scikit-Learn автоматически открывает доступ к методу `fit_predict`, который объединяет `fit` и `predict` в один вызов.

Объекты `Estimator` имеют параметры (которые также называют гиперпараметрами), управляющие процессом обучения. Эти параметры задаются при создании экземпляра `Estimator` (а в их отсутствие выбираются разумные значения по умолчанию) и могут изменяться с помощью методов `get_param` и `set_param`, которые также доступны в базовом классе `BaseEstimator`.

Используя Scikit-Learn API, мы указываем имя пакета и тип объекта `Estimator`. В примере ниже мы выбрали семейство моделей наивного байесовского классификатора и конкретного члена семейства — полиномиальную модель (которая подходит для классификации текста). Модель определяется при создании

экземпляра класса вместе с гиперпараметрами. Здесь мы передаем параметр `alpha`, который применяется для аддитивного сглаживания, а также предварительные вероятности для каждого из двух классов. Модель обучается на указанных данных (`documents` и `labels`), после чего превращается в обученную модель. Эта базовая последовательность действий одинакова для всех моделей (`Estimator`) в Scikit-Learn, от ансамблей деревьев решений случайного леса до логистической регрессии и др.

```
from sklearn.naive_bayes import MultinomialNB

model = MultinomialNB(alpha = 0.0, class_prior = [0.4, 0.6])
model.fit(documents, labels)
```

Расширение TransformerMixin

Библиотека Scikit-Learn также предоставляет утилиты для машинного обучения в повторяющемся режиме. Мы не сможем продолжить обсуждение Scikit-Learn, не описав интерфейс `Transformer`. `Transformer` — это специальный тип `Estimator`, который создает новый набор данных из старых, опираясь на правила, выявленные в процессе обучения. Вот как выглядит определение этого интерфейса:

```
from sklearn.base import TransformerMixin

class Transfomer(BaseEstimator, TransformerMixin):

    def fit(self, X, y = None):
        """
        Изучает правила преобразования на основе входных данных X.
        """
        return self

    def transform(self, X):
        """
        Преобразует X в новый набор данных Xprime и возвращает его.
        """
        return Xprime
```

Метод `Transformer.transform` принимает старый и возвращает новый набор данных, `X`, создает новые значения, опираясь на правила преобразования. В библиотеке Scikit-Learn есть несколько разновидностей `Transformer`, в том числе для нормализации и масштабирования признаков, обработки отсутствующих значений, сокращения размерности, извлечения признаков или отображения одного пространства признаков в другое.

Хотя обе библиотеки, NLTK и Gensim, и даже более новые библиотеки анализа текста, такие как SpaCy, имеют свои внутренние API и механизмы обучения, объем и полнота моделей и методологий Scikit-Learn для машинного обучения делают ее неотъемлемой частью рабочего процесса моделирования. В результате мы предлагаем использовать аналогичный API для создания своих объектов `Transformer` и `Estimator`, реализующих методы из NLTK и Gensim. Например, можно создать тематические классификаторы `Estimator`, обертывающие модели LDA и LSA из Gensim (в настоящее время отсутствующие в Scikit-Learn), или преобразователи `Transformer`, использующие методы маркировки частями речи из NLTK и методы деления на фрагменты именованных сущностей.

Создание своего преобразователя для векторизации на основе Gensim

Приемы векторизации в Gensim представляют особый интерес, потому что эта библиотека позволяет сохранять и загружать корпусы с диска, отделяя их от конвейера обработки. Однако есть возможность написать свой преобразователь, использующий механизм векторизации из Gensim. Наш преобразователь `GensimVectorizer` будет обертывать объект `Dictionary` из Gensim, сгенерированный методом `fit()`, чей метод `doc2bow` используется методом `transform()`. Объект `Dictionary` (например, `TfidfModel`) можно сохранить на диск и загрузить с диска, поэтому наш преобразователь тоже будет пользоваться такой возможностью, принимая путь при создании экземпляра. Если указанный файл существует, он загружается немедленно. Дополнительно с помощью метода `save()` мы сможем записывать свой `Dictionary` на диск после вызова `fit()`.

Метод `fit()` конструирует объект `Dictionary`, передавая лексемизированные и нормализованные документы конструктору `Dictionary`. Затем экземпляр `Dictionary` немедленно сохраняется на диск, чтобы преобразователь мог загрузить его без повторного обучения. Метод `transform()` вызывает метод `Dictionary.doc2bow`, который возвращает разреженное представление документа в виде списка кортежей `(token_id, frequency)`. Однако такое представление может вызывать проблемы в Scikit-Learn, поэтому мы используем вспомогательную функцию `sparse2full` из Gensim для преобразования разреженного представления в массив NumPy.

```
import os
from gensim.corpora import Dictionary
from gensim.matutils import sparse2full

class GensimVectorizer(BaseEstimator, TransformerMixin):

    def __init__(self, path = None):
```

```
        self.path = path
        self.id2word = None
        self.load()

    def load(self):
        if os.path.exists(self.path):
            self.id2word = Dictionary.load(self.path)

    def save(self):
        self.id2word.save(self.path)

    def fit(self, documents, labels = None):
        self.id2word = Dictionary(documents)
        self.save()
        return self

    def transform(self, documents):
        for document in documents:
            docvec = self.id2word.doc2bow(document)
            yield sparse2full(docvec, len(self.id2word))
```

Нетрудно заметить, как производится обертывание методов векторизации, обсуждавшихся выше в этой главе, преобразователями Scikit-Learn. Это дает нам большую гибкость в выборе подходов и в то же время позволяет использовать утилиты машинного обучения из каждой библиотеки. Дальнейшее расширение этого примера и реализацию преобразователей для получения оценок TF–IDF и распределенных представлений мы оставляем читателям для самостоятельного упражнения.

Создание своего преобразователя для нормализации текста

Многие семейства моделей страдают от «проклятия размерности»: по мере увеличения размерности пространства признаков данные становятся все более разреженными и менее информативными для базового пространства решений. Нормализация текста уменьшает число измерений и тем самым уменьшает разреженность. Кроме простой фильтрации лексем (удаления знаков препинания и стоп-слов) существует два основных метода нормализации: *стемминг* и *лемматизация*.

Стемминг заключается в использовании серии правил (или модели) для разделения строки на меньшие подстроки. Его цель заключается в удалении приставок и суффиксов из слов, меняющих их значение. Например, удаляются суффиксы `'s'` и `'es'`, которые в латинских языках обычно указывают на множественное число. Лемматизация, с другой стороны, выполняет поиск каждой

лексемы в словаре и возвращает каноническую (словарную) форму слова, которая называется леммой. Поскольку этот метод основан на поиске эталона, он помогает справляться с необычными случаями и обрабатывает лексемы, являющиеся разными частями речи. Например, глагол `'gardening'` (садоводство) должен преобразовываться в лемму `'to garden'`, тогда как существительные `'garden'` (сад) и `'gardener'` (садовник) — в разные леммы. Стемминг превратил бы все эти слова в одну лексему `'garden'` (сад).

Стемминг и лемматизация имеют свои достоинства и недостатки. Стемминг требует только разбить слово на подстроки, поэтому выполняется быстрее. Лемматизация, напротив, требует поиска в словаре или в базе данных и использует теги частей речи для выявления корневой леммы слова, что делает ее намного медленнее стемминга, но также намного эффективнее.

Чтобы обеспечить систематическую нормализацию текста, напишем свой преобразователь, который объединяет все эти этапы. Наш класс `TextNormalizer` принимает на входе язык, используемый для загрузки правильного набора стоп-слов из корпуса NLTK. Также можно предусмотреть настройки для `TextNormalizer`, чтобы дать возможность выбирать между стеммингом и лемматизацией, и передачу языка в `SnowballStemmer`. Для фильтрации посторонних лексем определим два метода. Первый, `is_punct()`, сравнивает первую букву в названии категории Юникода каждого символа с `'P'` (Punctuation — знаки препинания); второй, `is_stopword()`, проверяет, присутствует ли данная лексема в нашем множестве стоп-слов.

```
import unicodedata
from sklearn.base import BaseEstimator, TransformerMixin

class TextNormalizer(BaseEstimator, TransformerMixin):

    def __init__(self, language = 'english'):
        self.stopwords = set(nltk.corpus.stopwords.words(language))
        self.lemmatizer = WordNetLemmatizer()

    def is_punct(self, token):
        return all(
            unicodedata.category(char).startswith('P') for char in token
        )

    def is_stopword(self, token):
        return token.lower() in self.stopwords
```

Теперь добавим метод `normalize()`, принимающий единственный документ, то есть список абзацев, содержащих списки предложений, которые представлены

списками кортежей (token, tag) — этот формат данных мы получили в результате предварительной обработки документов HTML в главе 3.

```
def normalize(self, document):
    return [
        self.lemmatize(token, tag).lower()
        for paragraph in document
        for sentence in paragraph
        for (token, tag) in sentence
        if not self.is_punct(token) and not self.is_stopword(token)
    ]
```

Этот метод применяет функции фильтрации для удаления нежелательных лексем и затем выполняет лемматизацию. Метод `lemmatize()` сначала преобразует теги частей речи из набора Penn Treebank, который используется функцией `nltk.pos_tag`, в теги WordNet, выбирая по умолчанию существительное.

```
def lemmatize(self, token, pos_tag):
    tag = {
        'N': wn.NOUN,
        'V': wn.VERB,
        'R': wn.ADV,
        'J': wn.ADJ
    }.get(pos_tag[0], wn.NOUN)

    return self.lemmatizer.lemmatize(token, tag)
```

Наконец, мы должны добавить интерфейс `Transformer`, чтобы наш класс можно было включить в конвейер Scikit-Learn, о котором рассказывается в следующем разделе:

```
def fit(self, X, y = None):
    return self

def transform(self, documents):
    for document in documents:
        yield self.normalize(document)
```

Обратите внимание, что нормализация текста — лишь одна из множества методологий, которая к тому же активно использует NLTK, что может увеличить нежелательные для вашего приложения накладные расходы. В числе других возможных вариантов можно назвать удаление лексем, появляющихся чаще или реже некоторого предела, или удаление стоп-слов с последующим выбором первых 5–10 тысяч наиболее часто встречающихся слов. Еще один вариант — просто вычислить накопленную частоту и отобрать слова, со-

ставляющие 10–50 % накопленной частоты. Эти методы позволили бы нам игнорировать слова, встречающиеся в тексте очень редко или слишком часто, и выявить термины, потенциально наиболее значимые для прогнозирования в данном корпусе.

Операция нормализации текста должна быть необязательной и применяться с осторожностью, потому что имеет разрушительный характер, так как удаляет часть информации. Регистр символов, знаки препинания, стоп-слова и разнообразие конструкций слов — все это очень важно для понимания текста на естественном языке. Некоторые модели могут требовать наличия индикаторов, таких как регистр символов. Примером может служить классификатор именованных сущностей, потому что в английском (и в русском) языке имена собственные принято записывать с заглавной буквы.

Альтернативный способ уменьшения размерности — использовать метод главных компонент (Principal Component Analysis, PCA) или сингулярное разложение (Singular Value Decomposition, SVD) для уменьшения размерности пространства признаков до определенной величины (например, пять или десять тысяч измерений) на основе частоты слов. Эти преобразователи должны применяться после векторизации и могут производить эффект слияния слов, схожих в данном векторном пространстве.

Конвейеры

Процесс машинного обучения часто включает последовательное применение нескольких преобразователей к исходным данным, преобразующих набор данных шаг за шагом перед передачей методу обучения. Но, если не выполнить одинаковую векторизацию документов, мы получим неверные или, по крайней мере, малопонятные результаты. Объект конвейера `Pipeline` из библиотеки Scikit-Learn решает эту проблему.

Объекты `Pipeline` позволяют объединить несколько преобразователей, выполняющих нормализацию, векторизацию и анализ признаков, в общий, четко определенный механизм. Как показано на рис. 4.6, объекты `Pipeline` перемещают данные из загрузчика (объекта, обертывающего `CorpusReader`, созданного нами в главе 2) в механизмы извлечения признаков и, наконец, в объект оценки, реализующий модели прогнозирования. Объекты `Pipeline` — это ориентированные ациклические графы (Directed Acyclic Graph, DAG), которые могут быть простыми линейными цепочками преобразователей или сложными комбинациями ветвящихся и сливающихся путей.

Рис. 4.6. Конвейеры векторизации текста и извлечения признаков

Основы конвейеров

Задача `Pipeline` — связать вместе несколько объектов, представляющих фиксированную последовательность этапов обработки. Все объекты в конвейере, кроме последнего, должны быть преобразователями (то есть реализовывать метод `transform`), а последний может быть объектом оценки `Estimator` любого типа, в том числе с поддержкой прогнозирования. Конвейеры удобны в использовании; они позволяют для одного входного набора данных вызвать методы `fit` и `transform` сразу нескольких объектов. Кроме того, конвейеры реализуют единый интерфейс для поиска по сетке сразу из нескольких объектов. Но самое важное — конвейеры обеспечивают *практическое применение* текстовых моделей, объединяя методологию векторизации с моделью прогнозирования.

Конвейеры строятся путем описания списка пар `(key, value)`, где `key` — это строка с названием этапа, а `value` — объект, реализующий этот этап. Конвейеры можно создавать вызовом вспомогательной функции `make_pipeline`, которая автоматически определяет названия этапов, или явно определяя список. Конструируя конвейер вручную, предпочтительнее использовать второй подход, явно определяя названия этапов для повышения ясности кода, а функцию `make_pipe` применять в ситуациях, когда конвейер создается автоматически.

Объекты `Pipeline` — не только особый механизм, характерный для библиотеки Scikit-Learn, но также важная точка интеграции с библиотеками NLTK и Gensim. Следующий пример демонстрирует объединение объектов `TextNormalizer` и `GensimVectorizer`, созданных в предыдущем разделе, для подготовки данных к передаче в байесовскую модель. Используя интерфейс `Transformer`, как опи-

сывалось выше в этой главе, можно воспользоваться классом `TextNormalizer` и обернуть им объекты `CorpusReader` из NLTK, чтобы выполнить предварительную обработку и извлечь лингвистические признаки. Объект `GensimVectorizer` осуществит векторизацию, а объект `Pipelines` из библиотеки Scikit-Learn объединит их с разнообразными утилитами, такими как утилиты перекрестной проверки, и моделями, от наивного байесовского классификатора до логистической регрессии.

```
from sklearn.pipeline import Pipeline
from sklearn.naive_bayes import MultinomialNB

model = Pipeline([
    ('normalizer', TextNormalizer()),
    ('vectorizer', GensimVectorizer()),
    ('bayes', MultinomialNB()),
])
```

После этого объект `Pipeline` можно использовать как экземпляр единой модели. Вызов `model.fit` в данном случае будет равносилен последовательному вызову методов `fit` объектов в списке, каждый из которых преобразует входные данные и передает их следующему этапу. Другие методы, такие как `fit_transform`, действуют аналогично. Кроме того, объект конвейера будет иметь все методы последнего объекта в конвейере. Если последний объект является преобразователем, тогда и весь конвейер будет действовать как преобразователь. Если последний объект является классификатором, как в примере выше, конвейер автоматически получит методы `predict` и `score`, и его можно будет использовать как классификатор.

Объекты в конвейере хранятся в списке и доступны по индексам. Например, `model.steps[1]` вернет кортеж `('vectorizer', GensimVectorizer (path = None))`. Однако чаще доступ к объектам осуществляется по названиям этапов, с помощью свойства-словаря `named_steps` объекта `Pipeline`. Самый простой способ получить доступ к модели прогнозирования — использовать выражение `model.named_steps["bayes"]` и извлечь объект оценки непосредственно.

Поиск по сетке для оптимизации гиперпараметров

В главе 5 мы подробнее поговорим о настройке и обучении моделей, а пока просто познакомимся с расширением `Pipeline` под названием `GridSearch`, которое удобно использовать для оптимизации гиперпараметров. С помощью поиска по сетке можно реализовать изменение параметров всех объектов в конвейере, как если бы это был единственный объект. Для доступа к атрибутам объектов

можно использовать методы `set_params` и `get_params` конвейера, передавая им имена объектов и параметров, разделенные двойным подчеркиванием, например: `estimator__parameter`.

Предположим, что нам нужно выполнить прямое кодирование только слов, появляющихся в корпусе не менее трех раз. Для этого мы могли бы изменить `Binarizer`, как показано ниже:

```
model.set_params(onehot__threshold = 3.0)
```

Используя этот принцип, мы можем выполнять поиск по сетке, определяя искомые параметры с применением синтаксиса двойных подчеркиваний. Рассмотрим следующий поиск для определения наилучшего прямого кодирования текста для байесовской модели классификации:

```
from sklearn.model_selection import GridSearchCV

search = GridSearchCV(model, param_grid = {
    'count__analyzer': ['word', 'char', 'char_wb'],
    'count__ngram_range': [(1,1), (1,2), (1,3), (1,4), (1,5), (2,3)],
    'onehot__threshold': [0.0, 1.0, 2.0, 3.0],
    'bayes__alpha': [0.0, 1.0],
})
```

Поиск определяет для объекта `CountVectorizer` три возможных значения параметра `analyzer` (создание *n*-грамм по границам слов, по границам символов или только по символам между границами слов) и несколько диапазонов *n*-грамм для лексемизации. Также мы определили разные пороги для прямого кодирования, требующие, чтобы *n*-грамма появилась в тексте определенное количество раз, прежде чем она будет добавлена в модель. Наконец, определены два варианта сглаживания (параметр `bayes__alpha`): без сглаживания (`0.0`) или со сглаживанием Лапласа (`1.0`).

В результате определения поиска по сетке будут созданы конвейеры для каждой комбинации параметров. После этого мы сможем выполнить перекрестную оценку моделей и выбрать наилучшую комбинацию параметров (в данном случае лучшей будет комбинация, максимизирующая оценку F1).

Усовершенствование извлечения признаков с помощью объектов FeatureUnion

Конвейеры необязательно должны быть простыми, линейными последовательностями этапов; на самом деле их можно сделать сколь угодно сложными,

реализовав *объединение признаков*. Объект `FeatureUnion` объединяет несколько объектов преобразователей в единый преобразователь, похожий на объект `Pipeline`. Только преобразователи, включенные `FeatureUnion`, действуют не последовательно, а независимо, и их результаты в конце *объединяются* в составной вектор.

Рассмотрим пример на рис. 4.7. Представьте, что у нас имеется HTML-парсер, который с помощью библиотеки BeautifulSoup или XML производит синтаксический анализ, или парсинг разметки HTML и возвращает тело каждого документа. Затем выполняется этап конструирования признаков, в ходе которого из документов извлекаются сущности и ключевые фразы, и результаты передаются механизму объединения признаков. Для сущностей предпочтительнее использовать частотное кодирование, потому что они относительно невелики, а для ключевых фраз лучше использовать оценку TF–IDF. Затем механизм объединения признаков объединяет два получившихся вектора так, что наше пространство решений перед логистической регрессией отделяет размерности слов в заголовке от размерностей слов в теле.

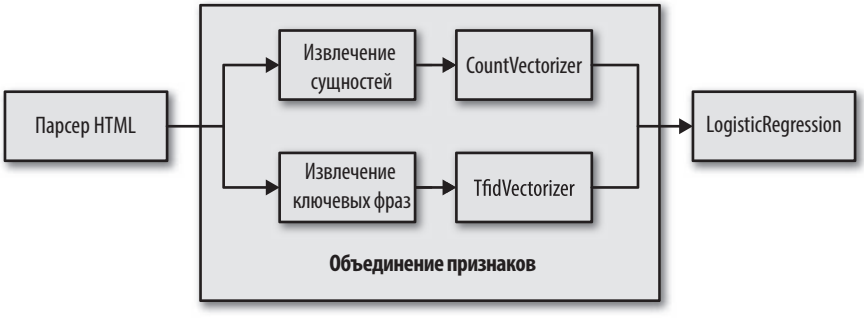


Рис. 4.7. Применение механизма объединения признаков для раздельной векторизации

Объекты `FeatureUnion` определяются так же, как объекты `Pipeline` — в виде списка пар `(key, value)`, где `key` — это название преобразования, а `value` — объект преобразователя. Также существует вспомогательная функция `make_union`, которая может автоматически определять названия преобразований и используется аналогично вспомогательной функции `make_pipeline` — для автоматического конструирования механизмов объединения признаков. Параметры встраиваемых объектов преобразователей доступны с использованием того же синтаксиса с двумя подчеркиваниями, разделяющими название преобразования и имя параметра.

Если допустить, что у нас уже имеются реализации преобразователей для извлечения сущностей и ключевых фраз, `EntityExtractor` и `KeyphraseExtractor`, конвейер с объединением признаков можно определить так:

```
from sklearn.pipeline import FeatureUnion
from sklearn.linear_model import LogisticRegression

model = Pipeline([
    ('parser', HTMLParser()),
    ('text_union', FeatureUnion(
        transformer_list = [
            ('entity_feature', Pipeline([
                ('entity_extractor', EntityExtractor()),
                ('entity_vect', CountVectorizer()),
            ])),
            ('keyphrase_feature', Pipeline([
                ('keyphrase_extractor', KeyphraseExtractor()),
                ('keyphrase_vect', TfidfVectorizer()),
            ])),
        ],
        transformer_weights = {
            'entity_feature': 0.6,
            'keyphrase_feature': 0.2,
        }
    )),
    ('clf', LogisticRegression()),
])
```

Обратите внимание на то, что в настоящий момент объекты `HTMLParser`, `EntityExtractor` и `KeyphraseExtractor` не реализованы, и здесь они упоминаются исключительно для иллюстрации. Объединение признаков выполняется последовательно по отношению к другим компонентам конвейера, но все преобразователи внутри действуют *независимо* — в том смысле, что каждый получает те же данные, что были переданы на вход объединения. Все преобразования применяются параллельно, а возвращаемые ими векторы объединяются в один большой вектор, который при необходимости может взвешиваться, как показано на рис. 4.8.

В этом примере мы придаем результатам преобразования `entity_feature` больший вес, чем результатам преобразования `keyphrase_feature`. Комбинируя преобразования, объединения признаков и конвейеры, можно чрезвычайно улучшить извлечение и преобразование признаков. А собрав реализованные методики в одну последовательность, мы сможем снова и снова применять преобразования, особенно к новым документам, когда понадобится организовать прогнозирование в промышленном окружении.

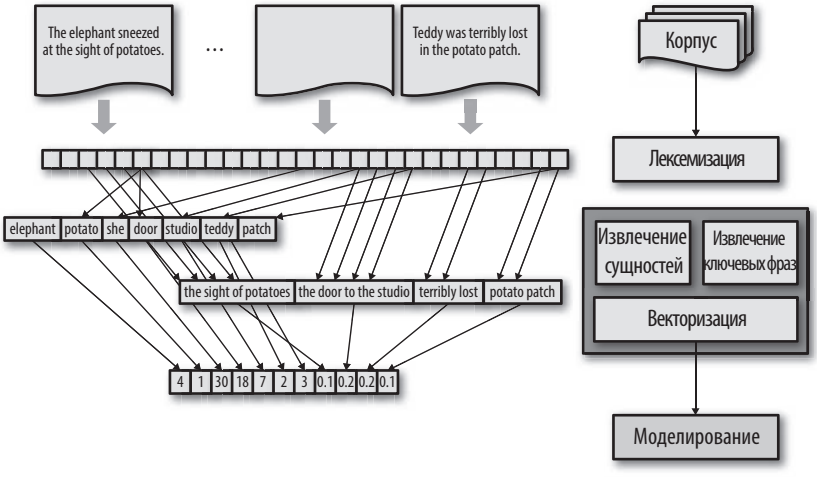


Рис. 4.8. Извлечение и объединение признаков

В заключение

В этой главе мы кратко рассмотрели приемы векторизации и варианты их применения к разным данным и алгоритмам машинного обучения. На практике желательно выбирать схему кодирования, исходя из решаемой задачи; некоторые методы намного лучше других походят для определенных задач.

Например, для моделей рекуррентных нейронных сетей часто лучше подходит прием прямого кодирования, но для разделения текстового пространства можно создать объединенный вектор, включающий векторные представления описания документа, его заголовка, тела и т. д. Результаты частотного кодирования желательно нормализовать, и разные виды частотного кодирования могут принести пользу вероятностным методам, таким как байесовские модели. Кодирование TF–IDF часто с успехом используется в моделировании, но не всегда оказывается безгрешным. Распределенные представления — новый способ, набирающий популярность, но предъявляющий высокие требования к производительности и с трудом поддающийся масштабированию.

Модели типа «мешок слов» имеют очень высокую размерность, что обусловливает большую разреженность и сложность обобщения пространства данных. Они утрачивают порядок слов и другие структурные признаки и осложняют добавление новых знаний (таких, как лексические ресурсы или онтологическое кодирование) в процесс обучения. Локальное кодирование (например, нераспределенные представления) требует большой выборки, что может вызвать

эффект переобучения или недообучения, но распределенные представления, в свою очередь, слишком сложны и добавляют слой «репрезентативного мистицизма».

В конечном счете основная работа приложений на основе анализа естественного языка заключается в анализе предметных признаков, а не только в простой векторизации. В заключительном разделе этой главы мы рассмотрели использование объектов `FeatureUnion` и `Pipeline` для реализации значимых методов извлечения признаков путем объединения преобразователей. И впредь практика конструирования конвейеров из преобразователей и объектов оценки будет оставаться нашим основным механизмом машинного обучения. В главе 5 мы исследуем модели классификации и их применение, затем, в главе 6, рассмотрим модели кластеризации, которые в анализе текста часто называют *тематическим моделированием*. В главе 7 мы познакомимся с более сложными методами анализа и исследования признаков, которые оказывают помощь в тонкой настройке моделей на основе векторных представлений для достижения лучших результатов. Тем не менее простые модели, учитывающие только частоту слов, часто позволяют добиться неплохих результатов. По нашему опыту, модель «мешок слов» дает хорошие результаты примерно в 85 % случаев!

5 Классификация в текстовом анализе

Представьте, что сейчас конец 90-х, вы работаете в крупной компании — провайдере электронной почты и занимаетесь реализацией обработки все увеличивающегося потока писем с серверов, разбросанных по всему свету. Распространенность и дешевизна электронной почты сделали ее основной формой коммуникации, и ваш бизнес процветает. К сожалению, растет и поток нежелательной почты. На более безобидном краю спектра находятся письма, рекламирующие интернет-продукты, но их настолько много, что они оказывают существенную нагрузку на ваши серверы. Кроме того, поскольку почта никем не контролируется, постоянно увеличивается поток вредоносных писем — все больше писем содержат лживую рекламу, завлекая адресатов в финансовые пирамиды и фейковые инвестиционные программы. Что делать?

Сначала можно попробовать заносить адреса почты и IP-адреса в черный список спамеров или организовать поиск по ключевым фразам, идентифицирующим спам. К сожалению, получить новый адрес почты или IP-адрес не представляет труда, поэтому спамеры быстро находят пути обхода ваших даже самых подробных черных списков. Хуже того, черные и белые списки могут отфильтровывать нормальную почту и тем самым вызывать неудовольствие ваших пользователей. Вам нужно другое, более гибкое и стохастическое решение, способное масштабироваться, и такое решение есть: машинное обучение.

Перемотаем время на пару десятилетий вперед. Теперь фильтрация спама получила большое распространение и превратилась в самую успешную, пожалуй, модель классификации текста. Главным новшеством стала классификация писем по их содержимому. Это не просто присутствие слов виагра или нигерийский принц, но сам контекст, частота слов и орфографические ошибки. Формирование

корпусов спама и нормальных писем позволило сконструировать наивную байесовскую модель, которая использует единообразную методику для определения вероятностей присутствия слов в спаме и нормальных письмах по их частоте.

В начале этой главы мы исследуем несколько примеров классификации из реальной жизни и посмотрим, как правильно сформулировать эти проблемы для разработки приложений. Затем мы рассмотрим процесс классификации и расширим методологии векторизации, рассмотренные в главе 4, чтобы создать конвейеры тематической классификации с использованием корпуса Baleen, представленного в главе 2. Наконец, мы приступим к исследованию следующих шагов в нашем процессе, которые строятся на фундаменте уровня данных, который мы создали к настоящему моменту. Все эти следующие шаги мы опишем с позиции тройки выбора модели, о которой рассказывалось в главе 1.

Классификация текста

Классификация является основной формой анализа текста и широко используется в различных областях. В основе классификации лежит простая идея: изучить существующие связи между экземплярами, состоящими из независимых переменных, и целевой категориальной переменной. Поскольку цель известна заранее, классификацию называют машинным обучением *с учителем*, и модель можно обучить для минимизации ошибки между предсказанными и фактическими категориями обучающих данных. После обучения модель классификации сможет присваивать категории новым экземплярам, опираясь на шаблоны, выявленные в процессе обучения.

Эта простая идея имеет широкий спектр применения при условии, что прикладную задачу можно сформулировать в терминах «да/нет» (двоичная классификация) или группы дискретных категорий (многоклассовая классификация). Самая сложная часть прикладного анализа текста — курирование и сбор предметно-ориентированного корпуса для построения моделей. Вторая по сложности часть — выработка аналитического решения конкретной прикладной задачи.

Идентификация задач классификации

Не всегда очевидно, как сформировать решение классификации для прикладных задач, но такая сложность помогает понять, что большинство приложений данных на основе анализа естественного языка в действительности состоит из множества моделей. Например, система рекомендаций, изображенная на рис. 5.1, может включать классификаторы, определяющие целевой возраст

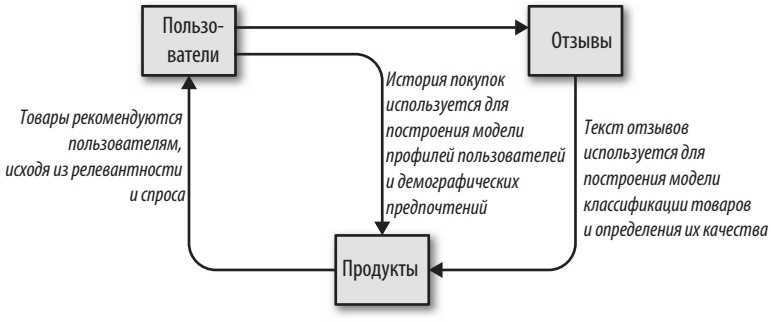


Рис. 5.1. Механизм рекомендаций из нескольких моделей

для товара (например, подростковые и взрослые велосипеды), пол (женская и мужская одежда) или категорию (электроника и видеофильмы), опираясь на описание или другие атрибуты товара. Отзывы о товарах могут затем классифицироваться для определения их качества или схожих продуктов. Далее эти классы могут использоваться как признаки для последующих моделей или создания разделов в ансамблевых моделях.

Объединение нескольких классификаторов показало себя с самой лучшей стороны, особенно в последние годы и в некоторых конкретных применениях классификации текста: от клиентов электронной почты с поддержкой фильтрации спама до приложений, способных предсказывать политическую предвзятость новостных статей. Область применения классификации настолько же широка, насколько мы можем себе представить, а люди обладают хорошим воображением. Новейшие приложения объединяют обучение на текстах и изображениях для совершенствования форм передачи информации — от автоматического создания подписей до распознавания образов, используя приемы классификации.

Пример классификации спама недавно заменило новомодное увлечение: анализ эмоциональной окраски. Модели этого вида анализа пытаются предсказать положительные («Я люблю писать код на Python») или отрицательные («Меня раздражает, когда люди повторяются») эмоции, опираясь на содержимое, и приобрели значительную популярность благодаря выразительности социальных медиа. Поскольку компании участвуют в более общем диалоге, где они не контролируют информационные каналы (например, отзывы об их товарах и услугах), возникло мнение, что анализ эмоциональной окраски может помочь им с целевой поддержкой клиентов и даже повысить эффективность их работы. Но, как мы кратко рассмотрели в главе 1 и более подробно еще обсудим в главе 12, сложности и нюансы, присущие естествен-

ному языку, делают анализ эмоциональной окраски более сложным, чем определение спама.

Если эмоции можно определять по текстовому содержимому, то как быть с такими понятиями, как политическая предвзятость? На основе высказываний, сделанных в ходе избирательной кампании в США, в недавних работах были созданы модели, способные определять приверженность (или ее отсутствие) определенным политическим взглядам. В результате этих усилий был получен интересный результат: как оказалось, пользовательские модели (то есть обученные на данных конкретных пользователей) обеспечивают более эффективный контекст, чем глобальная, обобщенная модель (обученная на объединенных данных большого количества пользователей)[1]. Другим практическим примером может служить автоматическая тематическая классификация текста: на основе блогов, публикующих статьи в одной предметной области (например, в кулинарном блоге обычно не обсуждаются новинки кинематографа), можно создать классификаторы, способные определять тематику неклассифицированных источников, таких как новостные статьи.

Так что же объединяет все эти примеры? Прежде всего, уникальная внешняя цель, определяемая областью применения: что мы собираемся оценивать? Независимо от того, собираемся ли мы фильтровать спам, определять эмоциональную окраску или приверженность политическим взглядам, конкретную тему или язык текста, приложение должно определить классы. Вторая общая черта — возможность судить о классе документа или высказывания по его содержимому. С этими двумя эмпирическими правилами появляется возможность организовать автоматическую классификацию в разных областях: выявление троллей, определение степени удобочитаемости или категории товара, рейтинга развлекательных учреждений, определение имени, идентификация автора и многих других.

Модели классификации

В классической задаче классификации спама используется наивный байесовский метод, который показывает высокую производительность и при построении модели (требуя выполнить только один проход по корпусу), и при прогнозировании (вычислении вероятности путем умножения входного вектора на внутреннюю таблицу истинности). Высокая производительность алгоритма означает возможность использования модели машинного обучения для работы с электронной почтой. Точность такой модели можно повысить,

[1] Benjamin Bengfort, *Data Product Architectures* (2016), *https://bit.ly/2vat7cN*.

добавив нетекстовые признаки, такие как электронный или IP-адрес отправителя, количество вложенных изображений, использование цифр вместо букв `"v14gr4"`[1] и т. д.

Наивный байесовский алгоритм позволяет создавать *оперативные* модели, то есть модели, способные обновляться в режиме реального времени без повторного обучения с самого начала (просто обновляя внутреннюю таблицу истинности и вероятности лексем). Это означало, что поставщики услуг электронной почты могли не отставать от спамеров, просто позволяя пользователям отмечать оскорбительные письма как спам и обновляя общую внутреннюю модель для всех.

Существует большое разнообразие моделей и механизмов классификации, которые математически более разнообразны, чем линейные модели, используемые в основном для регрессии. Приложения анализа текста имеют выбор из широкого круга семейств моделей — от методов на основе экземпляров, использующих дистанционное сходство, схемы разделения и байесовские вероятности, до линейных и нелинейных аппроксимаций и нейронного моделирования. Однако в основе всех семейств моделей классификации лежит один и тот же базовый процесс, и благодаря объектам `Estimator` из библиотеки Scikit-Learn их можно использовать процедурным способом и сравнивать путем приемов перекрестной проверки для выбора наиболее эффективных.

Процесс классификации делится на два этапа: обучения и эксплуатации, как показано на рис. 5.2. На этапе обучения корпус документов преобразуется в векторы признаков. Затем признаки документов вместе с их метками (категориями или классами, распознаванию которых мы хотим обучить модель) передаются в алгоритм классификации, который определяет свое внутреннее состояние и выявленные шаблоны. После обучения можно векторизовать новый документ в то же пространство признаков и передать результат алгоритму прогнозирования, который вернет метку категории документа.

Двоичные классификаторы распознают только два класса, и оба считаются противоположными друг другу (как, например, включено/выключено, да/нет и т. п.). Если говорить в терминах вероятностей: двоичный классификатор с классами `A` и `B` предполагает, что `P(B) = 1 - P(A)`. Однако в реальности такая однозначная связь встречается довольно редко. Возьмем для примера анализ эмоциональной окраски; если документ не положительный, значит ли это, что он отрицательный? Если какие-то документы носят нейтральную окраску (как это часто бывает на самом деле), добавление третьего класса в классификатор может значительно улучшить распознавание положительных и отрицательных

[1] Стилизованное написание слова «viagra». — *Примеч. пер.*

документов. В результате эта задача становится задачей многоклассовой классификации — например, A и ¬A (не A) и B и ¬B (не B).

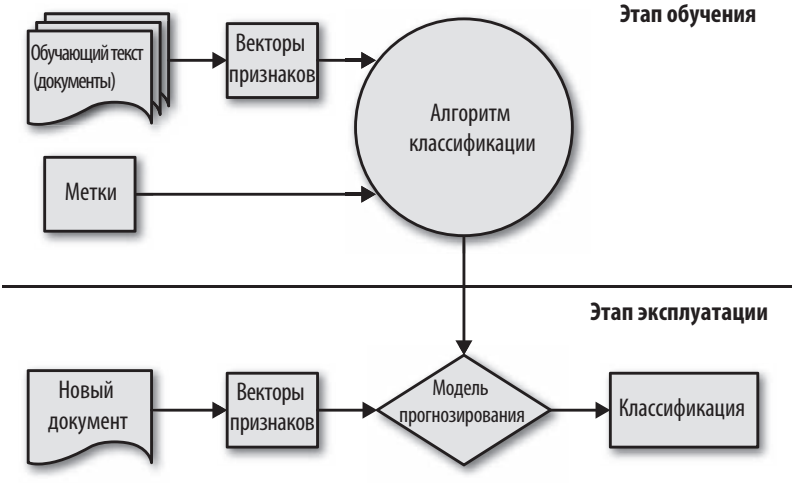


Рис. 5.2. Процесс классификации

Создание приложений классификации текста

В главах 2 и 3 рассказывалось, как собрать, обработать и сохранить документы HTML на диске, чтобы создать корпус. Движок сбора данных Baleen потребовал от нас настроить файл YAML, в котором мы организовали каналы RSS в категории по типам документов, которые в них содержатся. Каналы, имеющие отношение к компьютерным играм, были сгруппированы в свою категорию; каналы, имеющие отношение к технологиям и книгам, — в свои, и т. д. В результате получился корпус, документы в котором получили метки, присвоенные человеком. Эти метки, по сути, соответствуют разным категориям увлечений, а значит, можно попробовать создать классификатор, способный определять новости и статьи, наиболее соответствующие интересам пользователя!

В следующем разделе мы покажем основную методологию классификации документов на примере создания классификатора текста для предсказания метки данного документа («книги», «фильмы», «кулинария», «рукоделие», «игры», «спорт» или «технологии») по его содержимому. Предполагается, что в каждом классе естественный язык используется по-своему, и мы сможем построить надежный классификатор, который будет различать и распознавать категории документов.

В контексте нашей задачи каждый документ — это *экземпляр*, обучающий алгоритм классификации. Конечным результатом шагов, описанных в главах 2 и 3, является коллекция файлов, хранимых на диске определенным образом — по одному документу в файле; файлы хранятся в каталогах, с именами, соответствующими названиям их классов. Каждый документ представлен как объект на языке Python, включающий в себя несколько вложенных друг в друга списков: например, документ — это список абзацев, каждый абзац — это список предложений, а каждое предложение — это список кортежей `(token, tag)`.

Перекрестная проверка

Одной из самых больших проблем в машинном обучении является определение критерия остановки дальнейшего обучения — как узнать, когда наша модель станет готова к развертыванию? Когда следует прекратить настройку? Какая из моделей, имеющихся на выбор, лучше подходит для конкретного случая? Ответить на все эти вопросы нам поможет перекрестная проверка. Перекрестная проверка позволит сравнить результаты работы моделей с использованием *обучающей и контрольной выборки*, и оценить их качество в данном конкретном случае.

Наша главная цель — обучить классификатор, успешно обнаруживающий границы в обучающих данных и обобщающийся на прежде не известные ему данные. Успешное обнаружение границ обеспечивается правильным определением пространства признаков, на основе которого есть возможность построить пространство решений для разграничения классов. Под обобщаемостью понимается способность модели с высокой точностью прогнозировать принадлежность прежде не известных ей данных, не являющихся частью обучающей выборки.

Вся хитрость в том, чтобы пройти по грани, разделяющей *недообучение* и *переобучение*. Недообученная модель имеет низкую *дисперсию* и обычно выдает одни и те же прогнозы, но с большим *смещением*, отклоняясь от правильного ответа на значительную величину. Недообучение является признаком недостаточности точек данных для обучения сложной модели. Переобученная модель, напротив, запоминает обучающие данные и показывает на них очень высокую точность, но дает неудовлетворительные прогнозы на прежде не известных ей данных. Недообученные и переобученные модели не способны к *обобщению* — то есть не способны давать надежные прогнозы на новых, прежде неизвестных данных.

Между смещением и дисперсией существует взаимосвязь, как показано на рис. 5.3. С ростом числа признаков, параметров, глубины, количества эпох обучения и т. д. увеличивается сложность. С увеличением сложности и степени

переобучения модели ошибка на обучающих данных уменьшается, а на контрольных данных увеличивается из-за того, что модель становится все менее обобщенной.

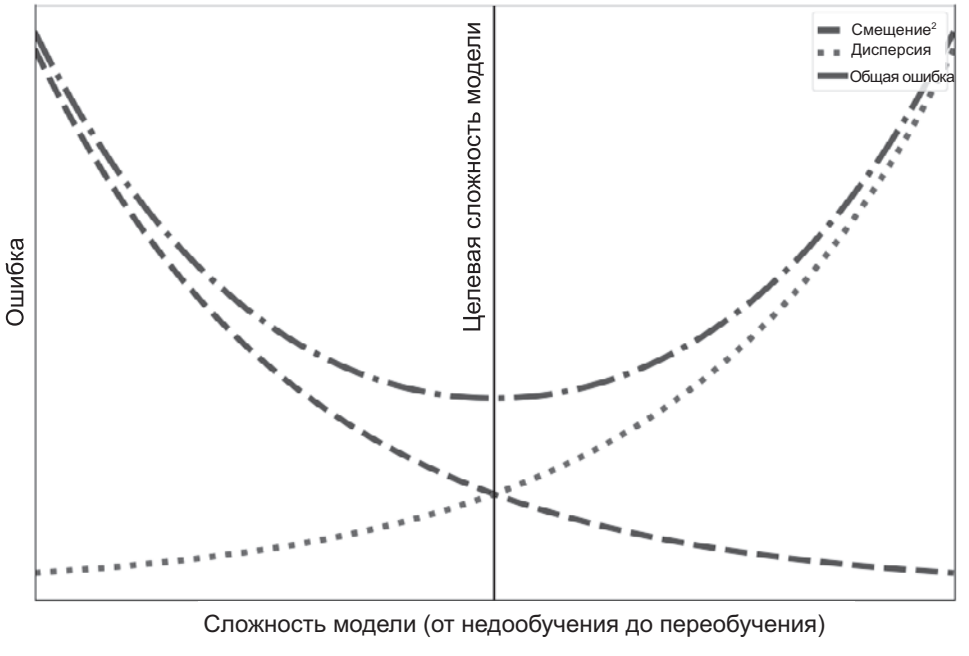


Рис. 5.3. Связь между смещением и дисперсией

Соответственно, наша цель — найти оптимальную точку с достаточной сложностью, чтобы избежать недообучения (и уменьшить смещение) без увеличения ошибки из-за высокой дисперсии. Чтобы найти эту оптимальную точку, нужно оценить модель на данных, не участвовавших в ее обучении. Решение этой задачи заключается в перекрестной проверке: экспериментальном методе, основанном на разделении данных на обучающую и контрольную выборки, последняя из которых используется только для проверки и не участвует в обучении.

Перекрестная проверка начинается с перемешивания данных (для предотвращения ошибок, вызванных упорядоченностью) и их разделения на k блоков, как показано на рис. 5.4. Затем производится обучение k моделей на $(k - 1)/k$ данных (называется обучающей выборкой) с оцениванием на $1/k$ данных (называется контрольной выборкой). Результаты оценки усредняются, и получается окончательная оценка, после этого на всем наборе данных обучается окончательная модель для эксплуатации.

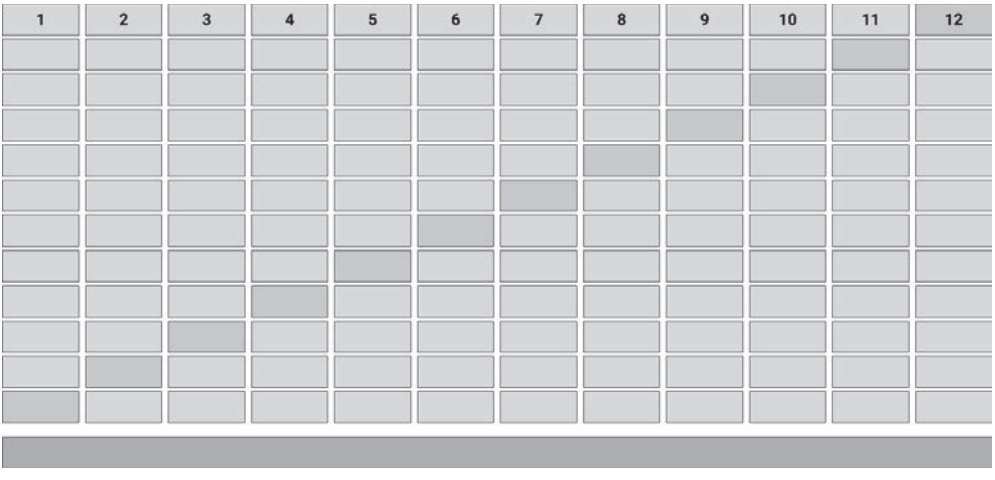


Рис. 5.4. Перекрестная проверка по *k*-блокам

Типичный вопрос — какое число *k* выбрать для перекрестной проверки по *k*-блокам. Мы обычно разбиваем набор данных на 12 блоков, как показано на рис. 5.4, однако на практике также часто используется разбиение на 10 блоков. Чем больше число блоков *k*, тем выше точность оценки на прежде неизвестных данных и тем больше времени требуется для обучения, иногда с уменьшением отдачи.

Потоковый доступ к *k*-блокам

Крайне важно взять в привычку выполнять перекрестную проверку, чтобы гарантировать высокое качество моделей, особенно в процессе выбора из нескольких моделей. Мы считаем это настолько важным в анализе текста, что обычно начинаем с создания объекта `CorpusLoader`, который обертывает объект `CorpusReader` и обеспечивает потоковый доступ к *k*-блокам!

Определим базовый класс `CorpusLoader`, который инициализируется экземпляром `CorpusReader`, количеством блоков и признаком необходимости перемешивания корпуса, который по умолчанию принимает значение `True`. Если число блоков не равно нулю, создадим объект `KFold` из библиотеки Scikit-Learn, который знает, как разбить корпус документов на заданное число блоков.

```
from sklearn.model_selection import KFold

class CorpusLoader(object):
```

```
    def __init__(self, reader, folds = 12, shuffle = True, categories = None):
        self.reader = reader
        self.folds = KFold(n_splits = folds, shuffle = shuffle)
        self.files = np.asarray(self.reader.fileids(categories = categories))
```

Следующим шагом добавим метод для доступа к списку имен файлов по идентификатору блока для обучения или проверки. Получив имена файлов, мы сможем вернуть документы и соответствующие им метки. Метод `documents()` реализует генератор, обращающийся к документам в корпусе и возвращающий для каждого из них список маркированных лексем. Метод `labels()` использует `corpus.categories()`, чтобы получить все метки из корпуса, и возвращает список меток для заданного документа.

```
    def fileids(self, idx = None):
        if idx is None:
            return self.files
        return self.files[idx]

    def documents(self, idx = None):
        for fileid in self.fileids(idx):
            yield list(self.reader.docs(fileids = [fileid]))

    def labels(self, idx = None):
        return [
            self.reader.categories(fileids = [fileid])[0]
            for fileid in self.fileids(idx)
        ]
```

Наконец, добавим свой метод-итератор, вызывающий метод `split()` объекта `KFold1` и возвращающий обучающую и контрольную выборки для каждого блока:

```
    def __iter__(self):
        for train_index, test_index in self.folds.split(self.files):
            X_train = self.documents(train_index)
            y_train = self.labels(train_index)

            X_test = self.documents(test_index)
            y_test = self.labels(test_index)

            yield X_train, X_test, y_train, y_test
```

В разделе «Оценка модели», далее в этой главе, мы будем использовать этот метод для создания решения перекрестной проверки с 12 блоками, которое обучает модель 12 раз и накапливает оценки для последующего усреднения и выявления наиболее качественной модели.

Конструирование модели

Как вы узнали в главе 4, объекты Scikit-Learn `Pipeline` из библиотеки Scikit-Learn помогают скоординировать процесс векторизации с процессом моделирования. Начнем с конвейера, который нормализует наш текст, преобразует его в векторное представление и затем передает классификатору. Такой подход поможет нам сравнить разные модели классификации, такие как наивная байесовская модель, логистическая регрессия и метод опорных векторов. Затем мы сможем применить прием уменьшения размерности пространства признаков, такой как метод *сингулярного разложения* (Singular Value Decomposition), чтобы убедиться в его способности или неспособности улучшить нашу модель.

В конце концов, мы сконструируем шесть моделей классификации: по одной каждого из трех типов в двух комбинациях конвейеров, как показано на рис. 5.5. Использование *гиперпараметров* по умолчанию для каждой из моделей поможет нам начать получать результаты.

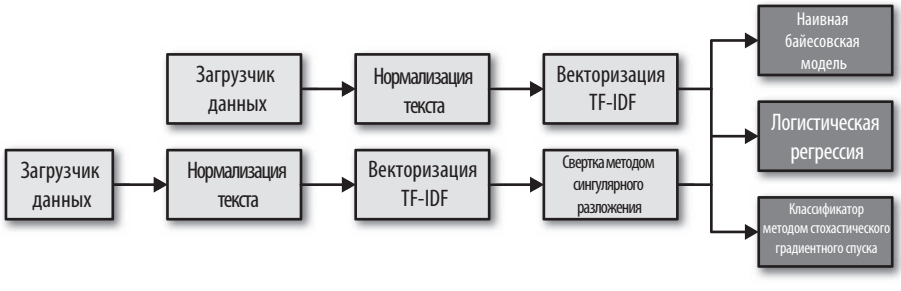


Рис. 5.5. Простой конвейер классификации

Итак, добавим функцию `create_pipeline`, принимающую объект оценки `Estimator` в первом аргументе и логический флаг необходимости применения сингулярного разложения для уменьшения числа признаков. Наш конвейер задействует преимущества `TextNormalizer`, созданного в главе 4, который использует лемматизацию WordNet для уменьшения общего количества классов слов. Так как мы уже обработали и нормализовали текст, в качестве функции `tokenizer` мы должны передать экземпляру `TfidfVectorizer` функцию `identity`, которая просто возвращает свои аргументы. Кроме того, следует предотвратить повторную предварительную обработку и преобразование в нижний регистр, передав объекту векторизации соответствующие аргументы.

```
from sklearn.pipeline import Pipeline
from sklearn.decomposition import TruncatedSVD
from sklearn.feature_extraction.text import TfidfVectorizer
```

```
def identity(words):
    return words

def create_pipeline(estimator, reduction = False):

    steps = [
        ('normalize', TextNormalizer()),
        ('vectorize', TfidfVectorizer(
            tokenizer = identity, preprocessor = None, lowercase = False
        ))
    ]

    if reduction:
        steps.append((
            'reduction', TruncatedSVD(n_components = 10000)
        ))

    # Добавить объект оценки
    steps.append(('classifier', estimator))
    return Pipeline(steps)
```

Теперь мы можем быстро сгенерировать свои модели:

```
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import MultinomialNB
from sklearn.linear_model import SGDClassifier

models = []
for form in (LogisticRegression, MultinomialNB, SGDClassifier):
    models.append(create_pipeline(form(), True))
    models.append(create_pipeline(form(), False))
```

Теперь в списке `models` есть шесть моделей — экземпляров конвейеров, включающих конкретные методы векторизации и извлечения признаков (анализа признаков), конкретный алгоритм и конкретные гиперпараметры (в настоящее время со значениями по умолчанию, как определено в библиотеке Scikit-Learn).

Обучение моделей с использованием заданной обучающей выборки с документами и их метками можно реализовать, как показано ниже:

```
for model in models:
    model.fit(train_docs, train_labels)
```

Вызовом метода `fit()` каждой модели документы и метки из обучающего набора передаются в начало соответствующего конвейера. Конвейеры вызывают методы `fit()` своих преобразователей и передают данные их методам `transform()`. На каждом этапе преобразованные данные поочередно передаются в метод `fit()` следующего преобразователя. Наконец, преобразованные данные передаются методу `fit()` объекта оценки в конце конвейера, в данном случае алгоритму

классификации. Вызов `fit()` преобразует предварительно обработанные документы (списки абзацев со списками предложений, которые состоят из списков кортежей с лексемами и тегами) в двумерный числовой массив, к которому затем применяются алгоритмы оптимизации.

Оценка модели

Но как определить лучшую модель? Так же, как в случае векторизации, выбор модели во многом зависит от особенностей данных и контекста применения. В данном случае нам нужно узнать, какая комбинация даст более точный прогноз принадлежности документа к той или иной категории увлечений, опираясь на его текст. Так как в контрольной выборке уже есть правильные целевые значения, мы можем сравнить ответы модели с этими значениями, вычислить процент правильных ответов и оценить глобальную точность каждой модели.

Давайте сравним наши модели. Используем объект `CorpusLoader`, созданный в разделе «Потоковый доступ к *k*-блокам» выше, чтобы получить обучающие и контрольные выборки для перекрестной проверки. Затем для каждого блока натренируем модель на обучающих данных и сопутствующих метках, затем получим вектор прогнозов для контрольных данных. Затем передадим предсказанные и фактические метки в оценивающую функцию и добавим полученную оценку в конец списка. Наконец, усредним результаты по всем блокам, чтобы получить общую оценку модели.

```
import numpy as np

from sklearn.metrics import accuracy_score

for model in models:
    scores = [] # Список оценок для каждого блока

    for X_train, X_test, y_train, y_test in loader:
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)
        score = accuracy_score(y_test, y_pred)
        scores.append(score)

    print("Accuracy of {} is {:0.3f}".format(model, np.mean(scores)))
```

Вот какие результаты получились у нас:

```
Accuracy of LogisticRegression (TruncatedSVD) is 0.676
Accuracy of LogisticRegression is 0.685
Accuracy of SGDClassifier (TruncatedSVD) is 0.763
```

```
Accuracy of SGDClassifier is 0.811
Accuracy of MultinomialNB is 0.562
Accuracy of GaussianNB (TruncatedSVD) is 0.323
```

Интерпретация точности заключается в изучении глобального поведения модели для всех классов. В данном случае, для классификатора с шестью классами, точность — это сумма правильно определенных классов, поделенная на общее число экземпляров контрольных данных. Однако общая точность не дает глубокого понимания происходящего в модели. Возможно, для нас важнее было бы знать, что определенный классификатор лучше определяет «спортивные», чем «кулинарные» статьи.

Есть ли классы, с которыми модели работают лучше, чем с другими? Может быть, есть какой-то один «плохой» класс, который снижает глобальную точность? Как часто обученный классификатор ошибается в пользу одного класса за счет другого? Чтобы получить представление об этих факторах, нужно взглянуть на оценки для классов: матрицу несоответствий.

В отчете классификации выводится оценка качества модели по классам. Так как отчет составляется путем сопоставления истинных меток с предсказанными, он обычно используется без деления выборки на блоки, а непосредственно на обучающей и контрольной выборках, чтобы точнее идентифицировать проблемную область в модели.

```
from sklearn.metrics import classification_report

model = create_pipeline(SGDClassifier(), False)
model.fit(X_train, y_train)

y_pred = model.predict(X_test)
print(classification_report(y_test, y_pred, labels = labels))
```

Сам отчет организован подобно матрице несоответствий, показывая разбивку по точности (precision), полноте (recall), оценке F1 и поддержке (support) каждого класса, как показано ниже:

```
        precision    recall    f1-score    support

  books      0.85      0.73        0.79         15
 cinema      0.63      0.60        0.62         20
cooking      0.75      1.00        0.86          3
 gaming      0.85      0.79        0.81         28
 sports      0.93      1.00        0.96         26
   tech      0.77      0.82        0.79         33

avg / total  0.81      0.81        0.81        125
```

Точность для класса `A` вычисляется как отношение числа документов, для которых правильно предсказан класс `A` (истинных `A`), к общему числу документов, для которых предсказан класс `A` (истинные `A` плюс ложные `A`). Точность показывает, насколько точно модель предсказывает данный класс, согласно количеству раз, когда она верно определила его.

Полнота для класса `A` вычисляется как отношение числа документов, для которых правильно предсказан класс `A` (истинных `A`), к общему числу документов класса `A` (истинных `A` + ложных `¬A`). Полноту также называют чувствительностью, она показывает, как часто выбираются релевантные классы.

Поддержка для класса показывает количество контрольных экземпляров, участвовавших в вычислении оценок. Как можно судить по отчету выше, класс `cooking` (кулинария) недостаточно представлен в нашей выборке, то есть модель получила недостаточное количество документов данного класса для надежной оценки.

Наконец, *оценка F1* — это среднее гармоническое из точности и полноты; она является более информативной, чем простая точность, потому что учитывает вклад каждого класса в общий результат.

В приложениях часто желательно иметь возможность повторного обучения модели по мере поступления новых данных. Этот процесс повторного обучения протекает за кулисами и в конечном итоге должен обновлять развернутую модель, выбирая наиболее эффективную на данный момент. По этой причине полезно выводить подобные результаты оценки в журнал приложения, чтобы потом можно было посмотреть, как изменяются точность, полнота и оценка F1, и время от времени проводить повторное обучение.

Мы можем свести все оценки модели в таблицу и отсортировать их по оценке F1, чтобы проще было выбрать лучшую модель через некоторое число итераций оценки.

```
import tabulate
import numpy as np

from collections import defaultdict
from sklearn.metrics import accuracy_score, f1_score
from sklearn.metrics import precision_score, recall_score

fields = ['model', 'precision', 'recall', 'accuracy', 'f1']
table = []

for model in models:
    scores = defaultdict(list) # хранит все оценки для текущей модели
```

```
    # перекрестная проверка по k-блокам
    for X_train, X_test, y_train, y_test in loader:
        model.fit(X_train, y_train)
        y_pred = model.predict(X_test)

        # Добавить оценки в scores
        scores['precision'].append(precision_score(y_test, y_pred))
        scores['recall'].append(recall_score(y_test, y_pred))
        scores['accuracy'].append(accuracy_score(y_test, y_pred))
        scores['f1'].append(f1_score(y_test, y_pred))

    # Сгруппировать оценки и добавить в table.
    row = [str(model)]
    for field in fields[1:]
        row.append(np.mean(scores[field]))

    table.append(row)

# Отсортировать модели по оценке F1 в порядке убывания
table.sort(key = lambda row: row[-1], reverse = True)
print(tabulate.tabulate(table, headers = fields))
```

Здесь мы изменили предыдущую оценку по *k*-блокам и использовали `defaultdict` для хранения оценок точности (precision), полноты (recall), качества (accuracy) и F1. После обучения модели на каждом блоке мы вычисляем среднее по всем оценкам и добавляем их в таблицу. Затем мы сортируем таблицу по оценке F1 и выводим ее с помощью модуля `tabulate`, как показано ниже, что позволяет быстро определить лучшую модель:

```
model                                precision   recall   accuracy    f1
-------------------------------- ----------- -------- ---------- -----
SGDClassifier                            0.821    0.811      0.811  0.81
SGDClassifier (TruncatedSVD)             0.81     0.763      0.763  0.766
LogisticRegression                       0.736    0.685      0.685  0.659
LogisticRegression (TruncatedSVD)        0.749    0.676      0.676  0.647
MultinomialNB                            0.696    0.562      0.562  0.512
GaussianNB (TruncatedSVD)                0.314    0.323      0.323  0.232
```

Эта таблица позволила нам быстро выяснить, что лучшей является модель, использующая метод опорных векторов, обученная с использованием алгоритма стохастического градиентного спуска без уменьшения размерности пространства признаков. Обратите внимание, что при выборе некоторых моделей, например `LogisticRegression`, предпочтительнее руководствоваться оценкой F1 вместо качества (accuracy). Подобное сравнение моделей позволяет легко протестировать разные комбинации признаков, гиперпараметров и алгоритмов, и найти лучшую модель для данной ситуации.

Эксплуатация модели

Теперь, выявив лучшую модель, самое время сохранить ее на диск для подготовки к *эксплуатации*. Методы машинного обучения ориентированы на создание моделей, способных выдавать прогнозы по новым данным в режиме реального времени без проверки. Чтобы задействовать модель в приложении, ее сначала нужно сохранить на диск, чтобы потом загрузить и использовать. Для этого лучше всего использовать модуль `pickle`:

```
import pickle
from datetime import datetime

time = datetime.now().strftime("%Y-%m-%d")
path = 'hobby-classifier-{}'.format(time)

with open(path, 'wb') as f:
    pickle.dump(model, f)
```

Модель сохраняется с именем, включающим дату ее создания.

Вместе с моделью важно также сохранить ее метаданные, например, в сопутствующем файле или в базе данных. Обучение модели — рядовой процесс, и, вообще говоря, оно должно производиться через регулярные интервалы времени, соответствующие скорости накопления данных. Построение графиков изменения качества модели с течением времени и принятие решения о сокращении данных и корректировке модели является важнейшей частью сопровождения приложений машинного обучения.

Чтобы задействовать модель в приложении для прогнозирования по новому тексту, просто загрузите модель из архива и вызовите ее метод `predict()`.

```
import nltk

def preprocess(text):
    return [
        [
            list(nltk.pos_tag(nltk.word_tokenize(sent)))
            for sent in nltk.sent_tokenize(para)
        ] for para in text.split("\n\n")
    ]

with open(path, 'rb') as f:
    model = pickle.load(f)

model.predict([preprocess(doc) for doc in newdocs])
```

Поскольку процесс векторизации интегрирован в модель через `Pipeline`, новые входные данные необходимо подготовить, используя в точности ту же

процедуру, что применялась для подготовки обучающих данных. Поэтому мы добавили процедуру предварительной обработки и преобразования строк в тот же формат. Теперь можно открыть файл архива с моделью, загрузить модель и вызвать ее метод `predict()`, чтобы получить метки.

В заключение

Как было показано в этой главе, выбор оптимальной модели — сложный итеративный процесс, намного более сложный, чем, скажем, выбор между методом опорных векторов и классификатором на дереве решений. Дискуссии, посвященные машинному обучению, часто фокусируются на выборе модели. Специалисты, практикующие применение методов машинного обучения, часто почти сразу выражают свои предпочтения в выборе между логистической регрессией, случайными лесами, байесовскими методами или искусственными нейронными сетями. Выбор модели, конечно, очень важен (особенно в контексте классификации текста), однако машинное обучение опирается на значительно более широкий круг факторов, чем простой выбор между «правильным» и «неправильным» алгоритмом.

В отношении прикладного анализа текста поиск модели осуществляется по общему шаблону: создается корпус, выбирается прием векторизации, выполняется обучение модели и ее оценка с применением приема перекрестной проверки. Намыльте, смойте, повторите[1] и сравните результаты. На основе результатов перекрестной проверки выберите лучшую модель и используйте ее для прогнозирования.

Важно отметить, что классификация предлагает такие оценочные параметры, как точность, полнота, качество и оценка F1, которыми можно руководствоваться при выборе алгоритма. Однако не все задачи машинного обучения можно сформулировать как обучение с учителем. В следующей главе мы обсудим еще один важный прием машинного обучения на текстовых данных — *кластеризацию*, который является обучением без учителя. Мы покажем, что несмотря на более высокую сложность, прием кластеризации тоже можно оптимизировать и на его основе создавать впечатляющие приложения, способные обнаруживать удивительные и полезные закономерности в больших объемах данных.

[1] Имеется в виду анекдот про программиста, надолго застрявшего в душе из-за следования инструкции на флаконе с шампунем: «намыльте, смойте, повторите». — *Примеч. пер.*

6 Кластеризация для выявления сходств в тексте

Как бы вы поступили, если бы вам вручили кучу документов — кулинарных рецептов, электронных писем, туристических путеводителей, протоколов заседаний — и предложили рассортировать их? Для этого можно попробовать прочитать каждый документ, выделить наиболее характерные слова и фразы и разложить их по стопкам. Если какая-то стопка получается слишком большой, можно попробовать разбить ее на две стопки меньшего размера. После просмотра и группировки документов можно попробовать исследовать каждую стопку более внимательно. Может быть, вы сможете использовать ключевые фразы или слова из каждой стопки, чтобы охарактеризовать их и дать каждой уникальное название — тему стопки.

Фактически, задачи подобного рода практикуются во многих дисциплинах, от медицины до юриспруденции. В их основе лежит наша способность сравнивать документы и определять их *сходство*. Документы, похожие друг на друга, объединяются в группы, которые описывают темы и закономерности внутри корпуса. Эти закономерности могут быть дискретными (например, когда группы вообще не пересекаются) или нечеткими (например, когда документы настолько схожи, что их трудно различить). В любом случае, получающиеся группы представляют собой модель содержимого всех документов, которая позволяет легко отнести новые документы к той или иной группе.

В настоящее время такая сортировка документов часто выполняется вручную, однако эту задачу вполне можно решить за меньшее время применением *обучения без учителя*, о котором рассказывается в этой главе.

Обучение на текстовых данных без учителя

Приемы обучения без учителя могут оказаться весьма полезными для анализа текста в процессе исследований. Часто корпусы не снабжаются метками, пригодными для классификации. В таких случаях остается единственный выбор (кроме перепоручения кому-то присвоить метки документам) или, по крайней мере, обязательный предварительный этап во многих задачах обработки естественного языка — обучение без учителя.

Цель алгоритмов кластеризации — выявить скрытую структуру немаркированных данных с использованием признаков для организации экземпляров в существенно различающиеся группы. При работе с текстовыми данными под экземпляром понимается единственный документ или высказывание, а под признаками — лексемы, словарь, структура, метаданные и т. д.

В главе 5 мы создали свой конвейер классификации для сравнения и оценки множества разных моделей и выбора наиболее качественной из них для получения прогнозов на новых данных. Методы обучения без учителя в корне отличаются; вместо выявления предопределенных шаблонов модель без учителя пытается найти *заведомо существующие* закономерности.

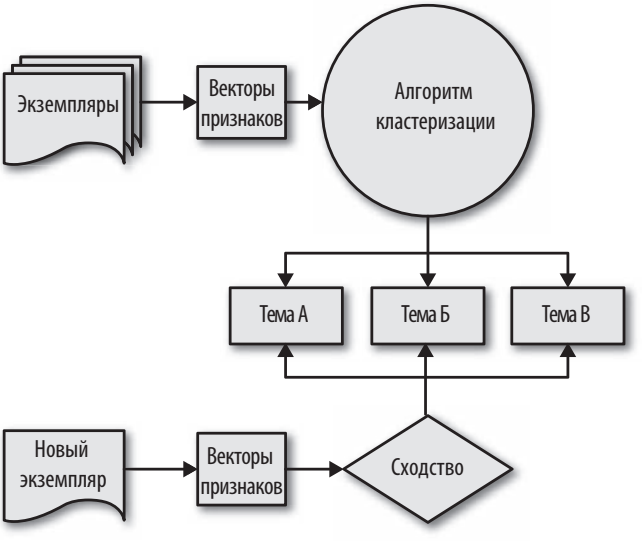


Рис. 6.1. Конвейер кластеризации

Как результат, интеграция этих приемов в архитектуру приложения данных неизбежно отличается. Как можно видеть на рис. 6.1 с изображением конвейера, корпус преобразуется в векторы признаков и передается алгоритму кластеризации для определения кластеров групп или тем с использованием метрик расстояния, согласно которым документы, более близкие в пространстве признаков, считаются более похожими. После этого можно выполнять векторизацию новых документов и помещать их в ближайший кластер. Далее в этой главе мы используем данный конвейер для сквозного кластерного анализа на выборке из корпуса Baleen, о котором рассказывалось в главе 2.

Прежде всего нам нужен метод определения сходства документов, и в следующем разделе мы исследуем спектр метрик расстояния, которые можно использовать для определения сходства двух документов. Далее мы рассмотрим два основных подхода к обучению без учителя, партитивную (partitive) и иерархическую (hierarchical) кластеризацию, и используем методы, реализованные в NLTK и Scikit-Learn. Получившиеся кластеры мы задействуем в экспериментах и используем Gensim для тематического моделирования и описания кластеров. В заключение мы исследуем два альтернативных способа обучения без учителя: матричное разложение и латентное размещение Дирихле (Latent Dirichlet Allocation, LDA).

Кластеризация документов по сходству

Существует множество признаков, которые могут свидетельствовать о сходстве документов, от слов и фраз до грамматики и структуры. Мы могли бы объединить медицинские записи, описывающие симптомы, заявив, что два пациента похожи, потому что у обоих наблюдаются «тошнота и утомление». Для сортировки персональных веб-сайтов и блогов мы, вероятно, будем использовать иной подход, называя похожими блоги, если в них опубликованы рецепты выпечки. Если новый блог содержит рецепты салатов из свежих овощей, он, вероятно, будет более похож на блоги с рецептами выпечки, чем на блоги с рецептами приготовления взрывчатки в домашних условиях.

Для эффективной кластеризации мы должны определить, что значит для любых двух документов из корпуса быть похожими или разными. Существует множество разных метрик, которые можно использовать для определения сходства документов; некоторые из них показаны на рис. 6.2. По сути все они опираются на возможность представлять документы как точки в пространстве, относительная близость которых определяет их сходство.

Сходство строк	Метрическое расстояние	Сходство отношений	Другие метрики
Дистанция редактирования Левенштейна Смита — Ватермана Аффинное **Выравнивание** Джаро — Винклера Soft TF–IDF Монг — Элкан **Фонетика** Созвучие Перевод	Евклидово Манхэттенское (расстояние городских кварталов) Минковского **Анализ текста** Жаккарда TF–IDF Косинусное сходство	**На основе множеств** Кубика Танимото (Жаккара) До общих соседей Взвешенное расстояние Адара **Группировка** Средние значения Максимальные/минимальные значения Медиана Частота (мода)	Числовое расстояние Логическое сходство Нечеткое соответствие Специализированные метрики **Газеты** Лексическое сопоставление Именованные сущности (Named Entities Recognition, NER)

Рис. 6.2. Метрики пространственного сходства

Метрики расстояния

Рассуждая об измерении расстояния между двумя точками, мы обычно имеем в виду соединяющую их прямую линию, или *евклидово расстояние*, изображенную на рис. 6.3 как диагональная линия.

Манхэттенское расстояние, изображенное на рис. 6.3 тремя ступенчатыми линиями, вычисляется как сумма абсолютных разностей декартовых координат. *Расстояние Минковского* — это обобщение евклидова и манхэттенского расстояний, оно определяет расстояние между двумя точками в нормализованном векторном пространстве.

Однако с ростом словаря корпуса также растет его размерность — и она редко бывает равномерно распределена. По этой причине перечисленные меры расстояния не всегда оказываются эффективными, потому что предполагают симметричность данных и равенство расстояний во всех измерениях.

Расстояние Махаланобиса, изображенное на рис. 6.4, напротив, является многомерным обобщением количества стандартных отклонений, отделяющих конкретную точку от распределения точек. В результате происходит сдвиг и масштабирование координат с учетом характеристик распределения. То есть расстояние Махаланобиса дает немного более гибкий способ определения

расстояний между документами; например, позволяет определять сходство высказываний разной длины.

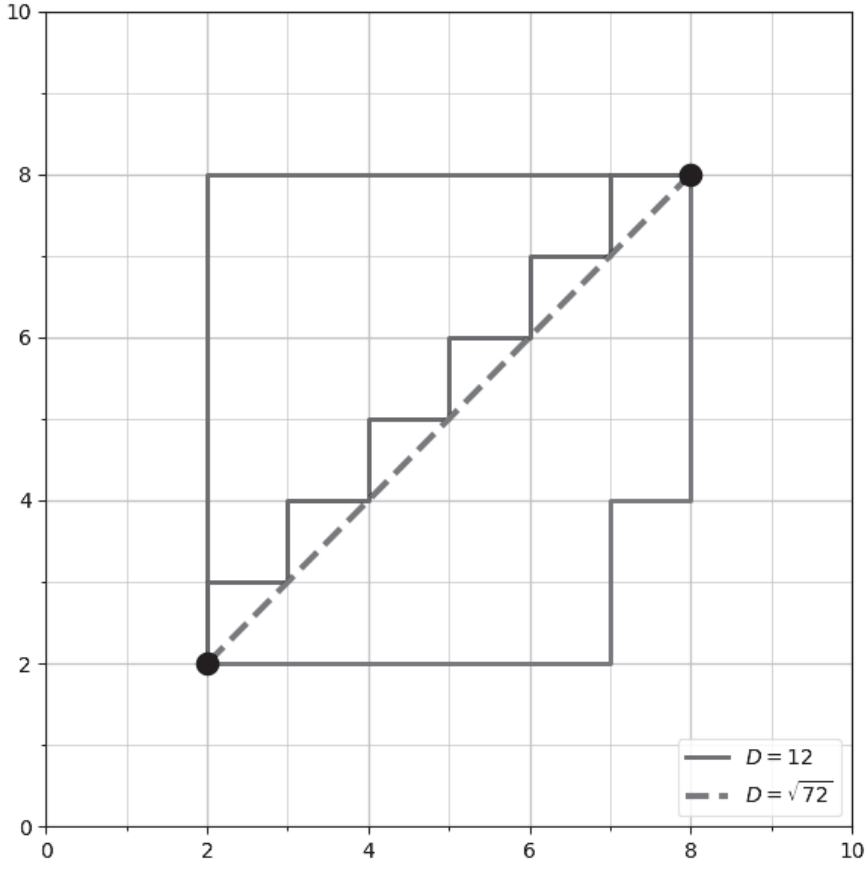


Рис. 6.3. Евклидово и манхэттенское расстояния

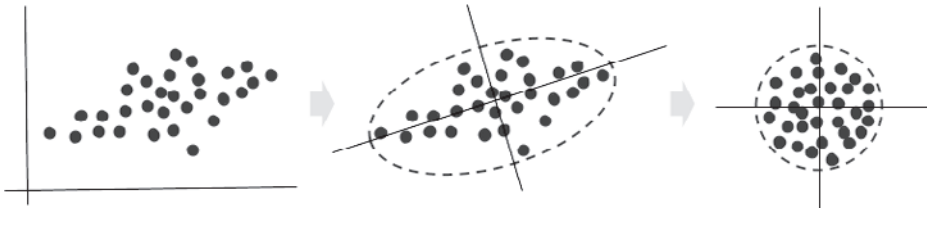


Рис. 6.4. Расстояние Махаланобиса

Расстояние Жаккарда определяет сходство между конечными множествами как частное от их пересечения и объединения, как показано на рис. 6.5. Например, расстояние Жаккарда между двумя документами `A` и `B` можно измерить, разделив количество общих уникальных слов, присутствующих в обоих документах, на количество всех уникальных слов в этих документах. Значение `0` означает полное отсутствие чего-либо общего в двух документах, `1` — что оба экземпляра представляют один и тот же документ, а значения между `0` и `1` определяют степень относительного сходства.

Рис. 6.5. Расстояние Жаккарда

Дистанция редактирования определяет удаленность двух строк друг от друга как количество изменений (правок), которые нужно произвести, чтобы из одной получить другую. Существует несколько реализаций измерения дистанции редактирования, все они основаны на расстоянии Левенштейна и различаются разными размерами штрафа за вставку, удаление и подстановку, а также потенциально увеличивающимися штрафами за пробелы и перемещения. На рис. 6.6 показано, что дистанция редактирования между словами «woodman» и «woodland» включает штрафы за одну вставку и одну подстановку.

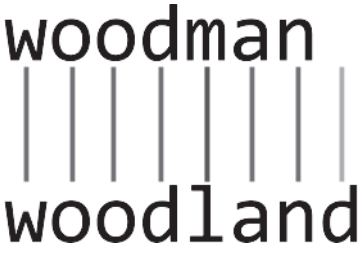


Рис. 6.6. Расстояние редактирования

Также есть возможность измерять расстояние между векторами. Например, сходство векторных представлений двух документов можно определить как *расстояние TF–IDF*; иными словами, количество общих уникальных слов относительно общего количества слов в корпусе. Определить это расстояние можно с помощью оценки TF–IDF, как описывалось в главе 4. Также можно вычислить *косинусное расстояние*, используя косинус угла между векторами, и интерпретировать его как степень сходства их ориентации, как показано на рис. 6.7. То есть, чем меньше угол между двумя векторами, тем более схожими считаются документы (с учетом их величины).

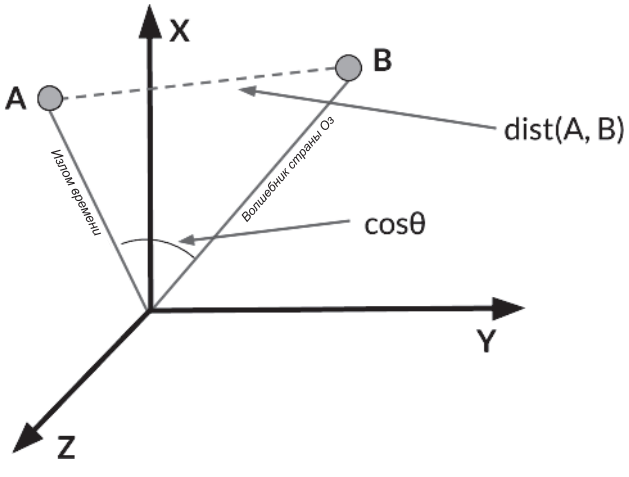


Рис. 6.7. Косинусное сходство

По умолчанию в гиперпараметрах модели кластеризации в качестве метрики сходства часто выбирается евклидово расстояние (как будет показано в следующих разделах), но нередко косинусное расстояние позволяет добиться лучших результатов.

Партитивная кластеризация

Теперь, узнав, как количественно оценить сходство двух документов, можно приступать к исследованию методов обучения без учителя для поиска групп похожих документов. Двумя основными подходами являются *партитивная кластеризация* и *агломеративная кластеризация*. Оба разделяют документы на группы с максимальным сходством внутри в соответствии с выбранной метрикой. В этом разделе мы сосредоточимся на партитивных методах, которые распределяют экземпляры по группам, представляемым центральными векторами

(центроидами) или описываемым плотностью документов в кластере. Центроиды представляют агрегированное значение (например, среднее или медиану) всех документов-членов и дают удобный способ описания документов в кластере.

Как было показано в главе 5, корпус Baleen уже классифицирован некоторым образом — все документы в нем помечены (человеком) категориями источников RSS, откуда они были извлечены. Однако некоторые из этих категорий четко различимы (например, «технологии» и «игры»), а другие более размыты. Например, корпус «новости» содержит новости в широком спектре тем, включая новости политики, новости индустрии развлечений, новости науки и описание других текущих событий. В этом разделе мы используем прием кластеризации для определения подкатегорий в корпусе «новости», которые потом можно будет использовать в качестве целевых значений в задачах классификации.

Кластеризация методом *k*-средних

Метод *k*-средних реализован в знакомых нам библиотеках, таких как NLTK и Scikit-Learn, поэтому начнем с него. Популярный метод для задач обучения без учителя, алгоритм кластеризации *k*-средних начинается с произвольно выбранного количества кластеров, *k*, и разбивает векторизованные экземпляры на кластеры согласно их близости к центроидам, которые минимизируют суммы квадратов внутри кластеров.

Начнем с определения класса `KMeansClusters`, наследующего `BaseEstimator` и `TransformerMixin`, чтобы его можно было включить в `Pipeline` из библиотеки Scikit-Learn. Для нашей модели используем реализацию *k*-средних из NLTK, потому что она позволяет определить свою метрику расстояния. Инициализируем `KMeansClusterer` желаемым количеством кластеров (*k*) и своей метрикой расстояния (`cosine_distance`) и исключим из результата кластеры, не содержащие документов.

```
from nltk.cluster import KMeansClusterer
from sklearn.base import BaseEstimator, TransformerMixin

class KMeansClusters(BaseEstimator, TransformerMixin):

    def __init__(self, k = 7):
        """
        число k определяет количество кластеров
        модель model является реализацией Kmeans
        """
        self.k = k
        self.distance = nltk.cluster.util.cosine_distance
        self.model = KMeansClusterer(self.k, self.distance,
                                     avoid_empty_clusters = True)
```

Теперь добавим пустой метод `fit()` и метод `transform()`, вызывающий метод `cluster()` внутренней модели `KMeansClusterer`, указав, что каждый документ должен помещаться в кластер. Наш метод `transform()` принимает векторные представления документов, полученные прямым кодированием, и, согласно параметру `assign_clusters = True`, будет возвращать список кластеров для каждого документа:

```
    def fit(self, documents, labels = None):
        return self

    def transform(self, documents):
        """
        Обучает модель K-Means векторными представлениями документов,
        полученными прямым кодированием.
        """
        return self.model.cluster(documents, assign_clusters = True)
```

Чтобы подготовить документы для передачи в класс `KMeansClusters`, их нужно векторизовать и нормализовать. Для нормализации используем версию класса `TextNormalizer`, которую мы определили в главе 4, в разделе «Создание своего преобразователя для нормализации текста», с одним маленьким изменением в методе `tranform()`. Вместо возврата документа в виде «мешка слов» эта версия `TextNormalizer` будет удалять стоп-слова, производить лемматизацию и возвращать строку для каждого документа:

```
class TextNormalizer(BaseEstimator, TransformerMixin):

    ...

    def transform(self, documents):
        return [' '.join(self.normalize(doc)) for doc in documents]
```

Для векторизации документов после кластеризации и последующей кластеризации определим класс `OneHotVectorizer`. Векторизацию будем осуществлять с помощью класса `CountVectorizer` с параметром `binary = True` из библиотеки Scikit-Learn, выполняющий частотное и двоичное кодирование. Наш метод `transform()` будет возвращать представление каждого документа в виде массива, полученного методом прямого кодирования:

```
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.feature_extraction.text import CountVectorizer

class OneHotVectorizer(BaseEstimator, TransformerMixin):

    def __init__(self):
        self.vectorizer = CountVectorizer(binary = True)
```

```
    def fit(self, documents, labels = None):
        return self

    def transform(self, documents):
        freqs = self.vectorizer.fit_transform(documents)
        return [freq.toarray()[0] for freq in freqs]
```

Теперь создадим конвейер `Pipeline` в главном выполняемом модуле, который произведет кластеризацию методом *k*-средних. Инициализируем `PickledCorpusReader`, как определено в главе 3, в разделе «Чтение предварительно обработанного корпуса», указав, что нас интересует только категория «news» в корпусе. Затем инициализируем конвейер, включив в него свои классы `TextNormalizer`, `OneHotVectorizer` и `KMeansClusters`. Затем вызовом метода `fit_transform()` конвейера выполним сразу все эти этапы в указанной последовательности:

```
from sklearn.pipeline import Pipeline

corpus = PickledCorpusReader('../corpus')
docs = corpus.docs(categories = ['news'])

model = Pipeline([
    ('norm', TextNormalizer()),
    ('vect', OneHotVectorizer()),
    ('clusters', KMeansClusters(k = 7))
])

clusters = model.fit_transform(docs)
pickles = list(corpus.fileids(categories = ['news']))
for idx, cluster in enumerate(clusters):
    print("Document '{}' assigned to cluster {}.".format(pickles[idx],
           cluster))
```

В результате мы получим список документов из категории «news» и назначенные им кластеры, с помощью которого мы легко можем получить имена файлов архивированных документов:

```
Document 'news/56d62554c1808113ffb87492.pickle' assigned to cluster 0.
Document 'news/56d6255dc1808113ffb874f0.pickle' assigned to cluster 5.
Document 'news/56d62570c1808113ffb87557.pickle' assigned to cluster 4.
Document 'news/56d625abc1808113ffb87625.pickle' assigned to cluster 2.
Document 'news/56d63a76c1808113ffb8841c.pickle' assigned to cluster 0.
Document 'news/56d63ae1c1808113ffb886b5.pickle' assigned to cluster 3.
Document 'news/56d63af0c1808113ffb88745.pickle' assigned to cluster 5.
Document 'news/56d64c7ac1808115036122b4.pickle' assigned to cluster 6.
Document 'news/56d64cf2c1808115036125f5.pickle' assigned to cluster 2.
Document 'news/56d65c2ec1808116aade2f8a.pickle' assigned to cluster 2.
...
```

Итак, мы получили предварительную модель для кластеризации документов по сходству их содержимого; теперь посмотрим, что можно сделать для оптимизации результатов. Но, в отличие от задачи классификации, у нас нет оценки, которая могла бы сообщить нам, насколько правильно выполнено разделение корпуса «news» на подкатегории — в обучении без учителя нет эталона для сравнения. Поэтому часто приходится полагаться на проверку человеком, чтобы оценить значимость выделенных кластеров — действительно ли они различны? Достаточно ли они сконцентрированы? В следующем разделе мы обсудим, что может подразумеваться под «оптимизацией» модели кластеризации.

Оптимизация метода *k*-средних

Как можно «улучшить» модель кластеризации? В нашем случае под этим вопросом понимается: как сделать результаты более полезными и простыми в интерпретации. Во-первых, упростить интерпретацию можно, экспериментируя с разными значениями *k*. В кластеризации методом *k*-средних выбор *k* — часто итеративный процесс; несмотря на существование некоторых эмпирических правил, первоначальный выбор часто делается произвольно до определенной степени.

В главе 8, в разделе «Оценка силуэта и локтевые кривые», мы обсудим два визуальных приема, которые можно использовать в экспериментах с выбором числа *k*: оценка силуэта и локтевые кривые.

Также можно настроить другие части конвейера; например, вместо прямого кодирования использовать векторизацию методом TF–IDF. Вместо `TextNormalizer` можно задействовать селектор признаков, выбирающий ограниченное подмножество из всего набора признаков (например, 5000 наиболее часто используемых слов, за исключением стоп-слов).

Несмотря на то что большие данные не являются главной темой этой книги, следует отметить, что метод *k*-средних эффективно масштабируется до уровня больших данных с применением навесной (canopy) кластеризации. Другие алгоритмы кластеризации текста, такие как LDA, намного труднее поддаются распараллеливанию без применения дополнительных инструментов (например, Tensorflow).

Имейте в виду, что алгоритм *k*-средних не является легковесным и может работать довольно медленно на данных с большим количеством измерений, таких как текст. Если конвейер кластеризации получился слишком медлен-

ным, скорость можно повысить, переключившись с модуля `nltk.cluster` на реализацию `MiniBatchKMeans` из `sklearn.cluster`. `MiniBatchKMeans` — это вариант алгоритма *k*-средних, который использует случайно выбранные подмножества (или «мини-пакеты») из всего обучающего набора для оптимизации той же целевой функции, но за меньшее время.

```
from sklearn.cluster import MiniBatchKMeans
from sklearn.base import BaseEstimator, TransformerMixin

class KMeansClusters(BaseEstimator, TransformerMixin):

    def __init__(self, k = 7):
        self.k = k
        self.model = MiniBatchKMeans(self.k)

    def fit(self, documents, labels = None):
        return self

    def transform(self, documents):
        return self.model.fit_predict(documents)
```

Мы выбрали более тяжеловесное решение, потому что более быстрая реализация `MiniBatchKMeans` использует евклидово расстояние — менее эффективную меру сходства для текста. На момент написания этих строк реализации `KMeans` и `MiniBatchKMeans` в Scikit-Learn не поддерживали других мер сходства, кроме евклидова расстояния.

Обработка неравномерной геометрии

Алгоритм *k*-средних делает несколько упрощенных предположений о данных: они равномерно распределены, являются принципиально сферическими по своей природе, и кластеры имеют сопоставимые дисперсии. Как результат, во многих случаях кластеризация методом *k*-средних не дает хороших результатов; например, когда резко выделяющиеся точки отрицательно сказываются на согласованности кластеров и когда кластеры имеют заметно иную, несферическую плотность. Исправить ситуацию в таких случаях может помочь применение альтернативных мер сходства, таких как косинусное расстояние или расстояние Махаланобиса.

В библиотеке Scikit-Learn есть реализации других партитивных методов, таких как метод распространения близости, спектральная кластеризация и гауссова смесь распределений, которые могут оказаться более эффективными в случаях, когда метод *k*-средних оказывается бессильным.

В целом, преимущества метода *k*-средних делают его важным инструментом в сфере анализа естественного языка; он обладает концептуальной простотой, производит плотные сферические кластеры, удобные центроиды, которые обеспечивают простоту интерпретации модели, и гарантирует схождение. В оставшейся части этой главы мы рассмотрим некоторые другие, более сложные методы, которые, как нам кажется, могут пригодиться для анализа текста, но нет панацеи от всех бед, и простой метод *k*-средних часто оказывается самой удобной отправной точкой.

Иерархическая кластеризация

В предыдущем разделе мы исследовали партитивные методы, которые распределяют точки по кластерам. Иерархическая кластеризация, напротив, предполагает создание кластеров с предопределенным порядком сверху вниз. Иерархические модели могут быть либо *агломеративными* (объединяющие), когда кластеры начинают формироваться с отдельных экземпляров и затем итеративно объединяются по подобию, пока все не окажутся в одной группе, либо *дивизимными* (делящие), когда сначала создается кластер, объединяющий всю выборку, и затем последовательно делится на более мелкие кластеры до достижения уровня отдельных экземпляров.

Эти методы создают древовидное представление структуры кластеров, как показано внизу слева на рис. 6.8.

Агломеративная кластеризация

Метод агломеративной кластеризации заключается в последовательном объединении ближайших экземпляров, пока все они не окажутся в одной группе. В случае с текстовыми данными результатом является иерархия разноразмерных групп, описывающая сходство документов на разных уровнях и с разной степенью подробности.

Мы легко можем переключить нашу реализацию кластеризации на использование агломеративного подхода. Сначала определим класс `HierarchicalClusters`, инициализирующий модель `AgglomerativeClustering` из библиотеки Scikit-Learn.

```
from sklearn.cluster import AgglomerativeClustering

class HierarchicalClusters(object):

    def __init__(self):
        self.model = AgglomerativeClustering()
```

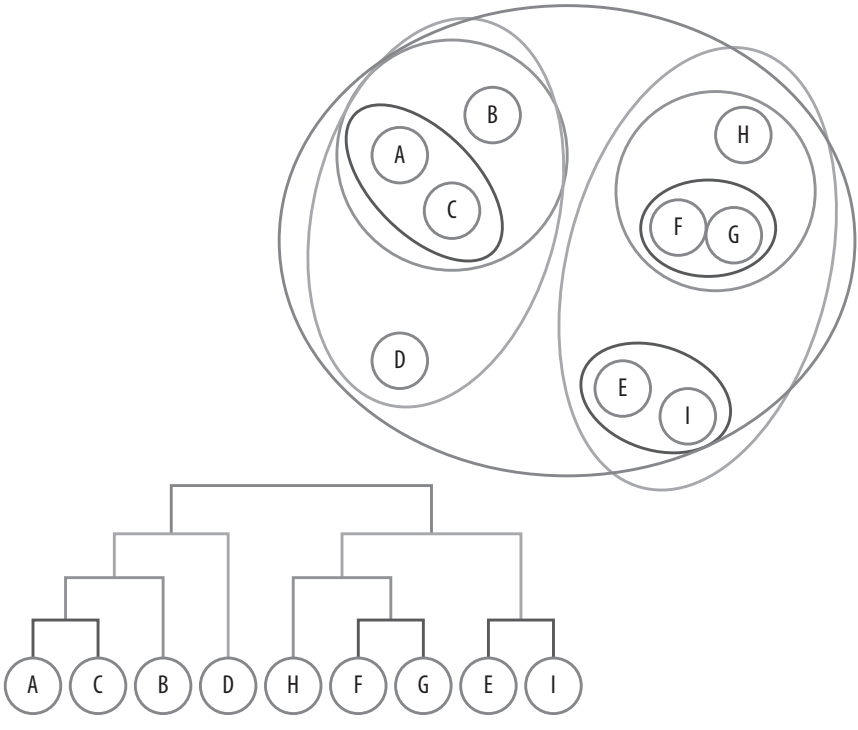


Рис. 6.8. Иерархическая кластеризация

Добавим пустой метод `fit()` и метод `tranform()`, который вызывает метод `fit_predict` внутренней модели, сохраняет получившиеся атрибуты `children` и `labels` для дальнейшего использования и возвращает кластеры.

```
...
    def fit(self, documents, labels = None):
        return self

    def transform(self, documents):
        """
        Обучает агломеративную модель на указанных данных.
        """
        clusters = self.model.fit_predict(documents)
        self.labels = self.model.labels_
        self.children = self.model.children_

        return clusters
```

Затем объединим все части в конвейер `Pipeline` и проверим метки кластера и членство каждого потомка в каждом неконцевом узле.

```
model = Pipeline([
    ('norm', TextNormalizer()),
    ('vect', OneHotVectorizer()),
    ('clusters', HierarchicalClusters())
])

model.fit_transform(docs)
labels = model.named_steps['clusters'].labels
pickles = list(corpus.fileids(categories = ['news']))

for idx, fileid in enumerate(pickles):
    print("Document '{}' assigned to cluster {}.".format(fileid,labels[idx]))
```

В результате получается следующее:

```
Document 'news/56d62554c1808113ffb87492.pickle' assigned to cluster 1.
Document 'news/56d6255dc1808113ffb874f0.pickle' assigned to cluster 0.
Document 'news/56d62570c1808113ffb87557.pickle' assigned to cluster 1.
Document 'news/56d625abc1808113ffb87625.pickle' assigned to cluster 1.
Document 'news/56d63a76c1808113ffb8841c.pickle' assigned to cluster 1.
Document 'news/56d63ae1c1808113ffb886b5.pickle' assigned to cluster 0.
Document 'news/56d63af0c1808113ffb88745.pickle' assigned to cluster 1.
Document 'news/56d64c7ac1808115036122b4.pickle' assigned to cluster 1.
Document 'news/56d64cf2c1808115036125f5.pickle' assigned to cluster 0.
Document 'news/56d65c2ec1808116aade2f8a.pickle' assigned to cluster 0.
...
```

Одна из проблем агломеративной кластеризации — отсутствие возможности использовать центроиды для маркировки кластеров документов, как мы делали это в примере реализации метода k-средних. Поэтому, чтобы получить возможность визуального исследования кластеров, полученных моделью `AgglomerativeClustering`, определим метод `plot_dendrogram`, создающий визуальное представление.

Наш метод `plot_dendrogram` будет использовать метод `dendrogram` из SciPy, а также модуль `pyplot` из библиотеки Matplotlib. Используем NumPy для вычисления расстояния между дочерними листовыми узлами и определим равные диапазоны значений для представления позиции каждого потомка. Затем создадим матрицу связей для хранения позиций потомков и расстояний между ними. Наконец, вызовем метод `dendrogram` из SciPy, передав ему матрицу связей и любые именованные аргументы, которые позднее можно будет передать для корректировки изображения.

```
import numpy as np
from matplotlib import pyplot as plt
from scipy.cluster.hierarchy import dendrogram

def plot_dendrogram(children, **kwargs):
```

```
    # Расстояния между парами потомков
    distance = position = np.arange(children.shape[0])

    # Создать матрицу связей и сформировать дендрограмму
    linkage_matrix = np.column_stack([
        children, distance, position]
    ).astype(float)

    # Нарисовать дендрограмму
    fig, ax = plt.subplots(figsize = (10, 5)) # set size
    ax = dendrogram(linkage_matrix, **kwargs)
    plt.tick_params(axis = 'x', bottom = 'off', top = 'off',
labelbottom = 'off')
    plt.tight_layout()
    plt.show()

children = model.named_steps['clusters'].children
plot_dendrogram(children)
```

Получившуюся диаграмму можно видеть на рис. 6.9. В кластеры с кратчайшими ветвями объединяются документы с наименьшими различиями. Кластеры с самыми длинными ветвями объединяют наиболее различающиеся группы и создаются на последних этапах процесса кластеризации.

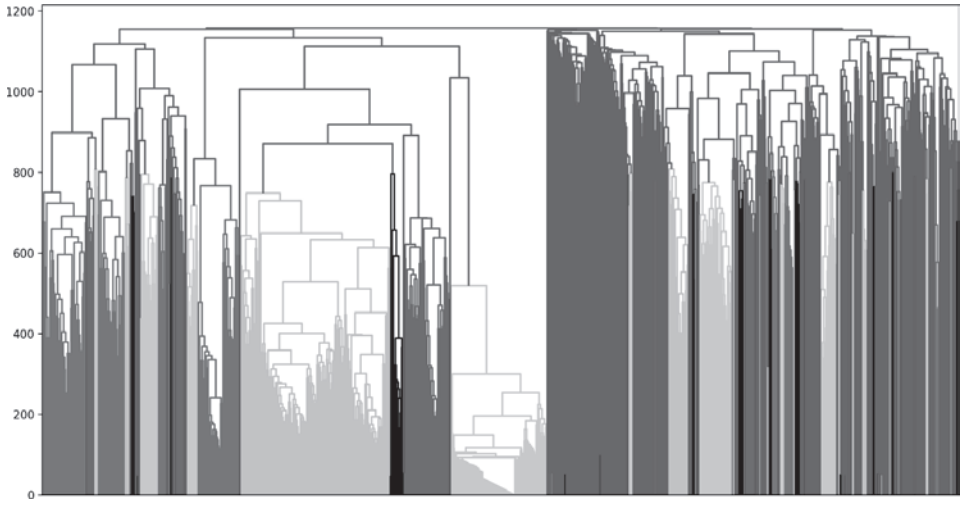


Рис. 6.9. Дендрограмма

Существует большое количество способов количественной оценки различий между любыми двумя документами, и точно так же существует большое количество критериев определения связей между ними. Для агломератив-

ной кластеризации требуется определить не только функцию расстояния, но также критерий связывания. По умолчанию в Scikit-Learn используется критерий Варда (Ward), который минимизирует дисперсию внутри кластера. На каждом шаге объединения алгоритм отыскивает пару кластеров, которые вносят наименьший вклад в общую дисперсию внутри кластера после объединения.

В числе других критериев, доступных в библиотеке Scikit-Learn, можно назвать `"average"`, использующий средние расстояния между точками в кластере, и `"complete"`, использующий максимальные расстояния между всеми точками в кластере.

Как можно заметить по результатам `KMeansClusters` и `HierarchicalClusters`, партитивные и агломеративные методы кластеризации не дают полного понимания, *почему* тот или иной документ оказался в конкретном кластере. В следующем разделе мы исследуем другой набор методов с поддержкой стратегий, которые не только позволяют быстро сгруппировать документы, но также описывают их содержимое.

Моделирование тематики документов

Теперь, рассортировав документы по стопкам, посмотрим, как выполнить маркировку и описать их содержимое. В этом разделе мы займемся *тематическим моделированием*, приемом машинного обучения без учителя для определения тем коллекций документов. Если цель кластеризации — разделить корпус документов на группы, то цель тематического моделирования — выделить основные темы из набора высказываний; кластеризация *дедуктивна*, а тематическое моделирование — *индуктивно*.

Методы тематического моделирования и удобные реализации с открытым исходным кодом достигли существенного прогресса за последнее десятилетие. В следующем разделе мы сравним три таких метода: латентное размещение Дирихле (Latent Dirichlet Allocation, LDA), латентно-семантический анализ (Latent Semantic Analysis, LSA) и неотрицательное матричное разложение (Non-Negative Matrix Factorization, NNMF).

Латентное размещение Дирихле

Впервые предложенный Дэвидом Блеем (David Blei), Эндрю Ыном (Andrew Ng) и Майклом Джорданом (Michael Jordan) в 2003 г., метод *латентного*

размещения Дирихле (Latent Dirichlet Allocation, LDA) является методом определения темы. Он принадлежит семейству порождающих вероятностных моделей, в которых темы представлены вероятностями появления каждого слова из заданного набора. Документы, в свою очередь, могут быть представлены как сочетания этих тем. Уникальная особенность моделей LDA состоит в том, что темы не обязательно должны быть различными и слова могут встречаться в нескольких темах; это придает некоторую нечеткость определяемым темам, что может пригодиться для совладания с гибкостью языка (рис. 6.10).

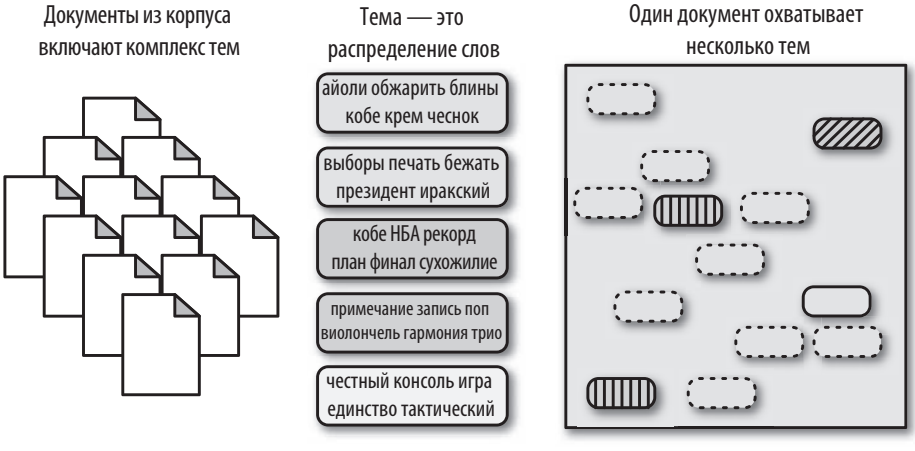


Рис. 6.10. Латентное размещение Дирихле

Блей (Blei) с коллегами (2003) установили, что распределение Дирихле, семейство непрерывных распределений (способ измерения группировки по распределениям), — это удобный способ выявления тем, присутствующих в корпусе, а также проявляющихся в разных сочетаниях в каждом документе в корпусе[1]. Фактически, метод *латентного размещения Дирихле* (LDA, Latent Dirichlet Allocation) дает нам наблюдаемое слово или лексему, по которому можно попытаться определить вероятную тему, распределение слов в каждой теме и сочетание тем в документе.

Чтобы задействовать методы тематического моделирования в приложении, нужно создать настраиваемый конвейер, который будет экстраполировать темы из неструктурированных текстовых данных, и способ сохранения лучшей

[1] David M. Blei, Andrew Y. Ng, and Michael I. Jordan, *Latent Dirichlet Allocation* (2003), https://stanford.io/2GJBHR1

модели, чтобы потом ее можно было использовать для анализа новых данных. Сделаем это сначала с применением Scikit-Learn, а затем — Gensim.

С применением Scikit-Learn

Начнем с создания класса `SklearnTopicModels`. Функция `__init__` создает конвейер с нашими объектами `TextNormalizer`, `CountVectorizer` и реализацией `LatentDirichletAllocation` из Scikit-Learn. Мы должны определить количество тем (здесь выбрано число 50), в точности как при кластеризации методом *k*-средних.

```
from sklearn.pipeline import Pipeline
from sklearn.decomposition import LatentDirichletAllocation
from sklearn.feature_extraction.text import CountVectorizer

class SklearnTopicModels(object):

    def __init__(self, n_topics = 50):
        """
        n_topics -- желаемое число тем
        """
        self.n_topics = n_topics
        self.model = Pipeline([
            ('norm', TextNormalizer()),
            ('vect', CountVectorizer(tokenizer = identity,
                                     preprocessor = None, lowercase = False)),
            ('model', LatentDirichletAllocation(n_topics = self.n_topics)),
        ])
```

В остальных примерах в этой главе мы будем использовать оригинальную версию `TextNormalizer` из раздела «Создание своего преобразователя для нормализации текста» в главе 4, которая возвращает представление документов в виде «мешка слов».

Затем определим метод `fit_transform`, вызывающий внутренние методы `fit` и `transform` на каждом этапе конвейера:

```
    def fit_transform(self, documents):
        self.model.fit_transform(documents)

        return self.model
```

Теперь, когда у нас есть все необходимое для создания конвейера обучения, нам нужен какой-нибудь механизм проверки тем. Темы никак не отмечаются,

и у нас нет центроидов, на основе которых можно было бы генерировать метки, как это делалось в примере кластеризации методом *k*-средних. Поэтому будем проверять каждую тему с точки зрения наиболее вероятных слов.

Определим метод `get_topics`, который получает из объекта конвейера векторное представление и извлекает лексемы из его атрибута `get_feature_names()`. Метод выполняет обход содержимого атрибута `components_` модели LDA и для каждой темы выполняет обратную сортировку лексем по весу, чтобы переместить в начало списка 25 лексем с самым высоким весом. Затем соответствующие лексемы извлекаются и сохраняются в словаре `topics`, ключами в котором служат индексы из числа 50 тем, а значениями — наиболее часто встречающиеся слова, связанные с данной темой.

```
    def get_topics(self, n = 25):
        """
        n -- число лексем с наибольшим весом в каждой теме
        """
        vectorizer = self.model.named_steps['vect']
        model = self.model.steps[-1][1]
        names = vectorizer.get_feature_names()
        topics = dict()

        for idx, topic in enumerate(model.components_):
            features = topic.argsort()[:-(n - 1): -1]
            tokens = [names[i] for i in features]
            topics[idx] = tokens

        return topics
```

Теперь можно создать экземпляр `SklearnTopicModels` и передать корпус документов в конвейер. Присвоим результат метода `get_topics()` (словарь) переменной `topics` и выведем его содержимое на экран, чтобы увидеть темы и наиболее информативные лексемы, соответствующие им:

```
if __name__ == '__main__':
    corpus = PickledCorpusReader('corpus/')

    lda       = SklearnTopicModels()
    documents = corpus.docs()

    lda.fit_transform(documents)
    topics = lda.get_topics()
    for topic, terms in topics.items():
        print("Topic #{}:".format(topic+1))
        print(terms)
```

Вот как выглядит результат работы этой программы:

```
Topic #1:
['science', 'scientist', 'data', 'daviau', 'human', 'earth', 'bayesian',
'method', 'scientific', 'jableh', 'probability', 'inference', 'crater',
'transhumanism', 'sequence', 'python', 'engineer', 'conscience',
'attitude', 'layer', 'pee', 'probabilistic', 'radio']
Topic #2:
['franchise', 'rhoden', 'rosemary', 'allergy', 'dewine', 'microwave',
'charleston', 'q', 'pike', 'relmicro', '($', 'wicket', 'infant',
't20', 'piketon', 'points', 'mug', 'snakeskin', 'skinnytaste',
'frankie', 'uninitiated', 'spirit', 'kosher']
Topic #3:
['cosby', 'vehicle', 'moon', 'tesla', 'module', 'mission', 'hastert',
'air', 'mars', 'spacex', 'kazakhstan', 'accuser', 'earth', 'makemake',
'dragon', 'model', input', 'musk', 'recall', 'buffon', 'stage',
'journey', 'capsule']
...
```

С применением Gensim

В библиотеке Gensim тоже есть реализация латентного размещения Дирихле, обладающая рядом интересных атрибутов, отличающих ее от реализации в Scikit-Learn. Начиная с версии 2.2.0 библиотека Gensim включает в себя удобную обертку для своей реализации `LDAModel` с именем `ldamodel.LdaTransformer`, которая упрощает интеграцию с конвейерами Scikit-Learn.

Чтобы воспользоваться классом `LdaTransformer` из Gensim, нужно создать свою обертку для объектов `TfidfVectorizer` из Gensim, чтобы их можно было использовать внутри конвейера `Pipeline` из Scikit-Learn. Класс `GensimTfidfVectorizer` выполняет векторизацию документов перед LDA, а также сохраняет и загружает специально подобранный лексикон для последующего использования.

```
class GensimTfidfVectorizer(BaseEstimator, TransformerMixin):

    def __init__(self, dirpath = ".", tofull = False):
        """
        Принимает путь к каталогу с лексиконом в файле corpus.dict
        и моделью TF-IDF в файле tfidf.model.

        Передайте tofull = True, если следующим этапом является объект
        Estimator из Scikit-Learn, иначе, если следующим этапом является
        модель Gensim, оставьте значение False.
        """
        self._lexicon_path = os.path.join(dirpath, "corpus.dict")
        self._tfidf_path = os.path.join(dirpath, "tfidf.model")

        self.lexicon = None
```

```
        self.tfidf = None
        self.tofull = tofull

        self.load()

    def load(self):
        if os.path.exists(self._lexicon_path):
            self.lexicon = Dictionary.load(self._lexicon_path)

        if os.path.exists(self._tfidf_path):
            self.tfidf = TfidfModel().load(self._tfidf_path)

    def save(self):
        self.lexicon.save(self._lexicon_path)
        self.tfidf.save(self._tfidf_path)
```

Если модель уже обучена, `GensimTfidfVectorizer` можно инициализировать лексиконом и векторным представлением, загрузив их с диска вызовом метода `load`. Мы также предусмотрели метод `save()`, чтобы получить возможность сохранять векторное представление.

Далее реализуем метод `fit()`, в котором создадим объект `Dictionary` из библиотеки Gensim, передав конструктору список нормализованных документов. Затем создадим объект `TfidfModel` на основе списка документов, каждый из которых преобразуется в мешок слов с помощью `lexicon.doc2bow`. Вызовем метод `save`, чтобы сохранить лексикон и векторное представление на диск, и, наконец, вернем ссылку `self`, как того требует Scikit-Learn API.

```
    def fit(self, documents, labels = None):
        self.lexicon = Dictionary(documents)
        self.tfidf = TfidfModel([
            self.lexicon.doc2bow(doc)
            for doc in documents],
            id2word = self.lexicon)
        self.save()
        return self
```

Теперь добавим метод `transform()`. Он создает генератор, который в цикле обходит все нормализованные документы и векторизует их, используя обученную модель и представление «мешок слов». Если следующим этапом в конвейере является модель Gensim, векторизатор инициализируется параметром `tofull = False` и в результате возвращает разреженные векторные представления документов (в виде последовательности 2-элементных кортежей). Но если следующим этапом является объект `Estimator` из библиотеки Scikit-Learn, объект `GensimTfidfVectorizer` должен инициализироваться параметром `tofull = True`, и тогда метод `transform` преобразует разреженное представление в плотное для сохранения в массиве `np`.

```
    def transform(self, documents):
        def generator():
            for document in documents:
                vec = self.tfidf[self.lexicon.doc2bow(document)]
                if self.tofull:
                    yield sparse2full(vec)
                else:
                    yield vec
        return list(generator())
```

Теперь, когда у нас есть обертка для векторизатора из библиотеки Gensim, мы можем объединить все компоненты в класс `GensimTopicModels`:

```
from sklearn.pipeline import Pipeline
from gensim.sklearn_api import ldamodel

class GensimTopicModels(object):

    def __init__(self, n_topics = 50):
        """
        n_topics -- желаемое число тем
        """
        self.n_topics = n_topics
        self.model = Pipeline([
            ('norm', TextNormalizer()),
            ('vect', GensimTfidfVectorizer()),
            ('model', ldamodel.LdaTransformer(num_topics = self.n_topics))
        ])

    def fit(self, documents):
        self.model.fit(documents)

        return self.model
```

И передать в конвейер наш `corpus.docs`:

```
if __name__ == '__main__':
    corpus = PickledCorpusReader('../corpus')

    gensim_lda = GensimTopicModels()

    docs = [
        list(corpus.docs(fileids = fileid))[0]
        for fileid in corpus.fileids()
    ]

    gensim_lda.fit(docs)
```

Получить темы для проверки можно из этапа LDA, который представлен атрибутом `gensim_model` последнего этапа конвейера. Вывести темы и десять самых

влиятельных лексем с их весами можно с помощью метода `show_topics` класса `LDAModel` из библиотеки Gensim:

```
lda = gensim_lda.model.named_steps['model'].gensim_model
print(lda.show_topics())
```

Также можно определить свою функцию `get_topics`, принимающую обученную модель `LDAModel` и векторизованный корпус, и возвращающую список тем с наибольшими весами для каждого документа в корпусе:

```
def get_topics(vectorized_corpus, model):
    from operator import itemgetter

    topics = [
        max(model[doc], key = itemgetter(1))[0]
        for doc in vectorized_corpus
    ]

    return topics

lda = gensim_lda.model.named_steps['model'].gensim_model

corpus = [
    gensim_lda.model.named_steps['vect'].lexicon.doc2bow(doc)
    for doc in gensim_lda.model.named_steps['norm'].transform(docs)
]

topics = get_topics(corpus,lda)

for topic, doc in zip(topics, docs):
    print("Topic:{}".format(topic))
    print(doc)
```

Визуализация тем

При использовании методов обучения без учителя часто полезно иметь возможность визуального представления результатов моделирования, потому что традиционные способы оценки качества моделей хороши только в задачах обучения с учителем. Более подробно методы визуализации, используемые при анализе текста, будут обсуждаться в главе 8, а здесь мы лишь кратко рассмотрим применение библиотеки pyLDAvis, предназначенной для создания визуального интерфейса с целью помочь в интерпретации тем, полученных в результате тематического моделирования.

PyLDAvis извлекает информацию из обученной тематической модели LDA и отображает ее в интерактивном веб-интерфейсе, который легко можно открыть внутри Jupyter Notebook или сохранить в файл HTML. Для визуализации

тем документа с помощью pyLDAvis можно выполнить конвейер внутри Jupyter Notebook, как показано ниже:

```
import pyLDAvis
import pyLDAvis.gensim

lda = gensim_lda.model.named_steps['model'].gensim_model

corpus = [
    gensim_lda.model.named_steps['vect'].lexicon.doc2bow(doc)
    for doc in gensim_lda.model.named_steps['norm'].transform(docs)
]
lexicon = gensim_lda.model.named_steps['vect'].lexicon

data = pyLDAvis.gensim.prepare(model,corpus,lexicon)
pyLDAvis.display(data)
```

Сначала нужно вызвать основной метод `pyLDAvis.gensim.prepare`, который принимает модель LDA, векторизованный корпус и порожденный лексикон, а затем вызвать метод `display`, чтобы создать визуальное представление, подобное изображенному на рис. 6.11.

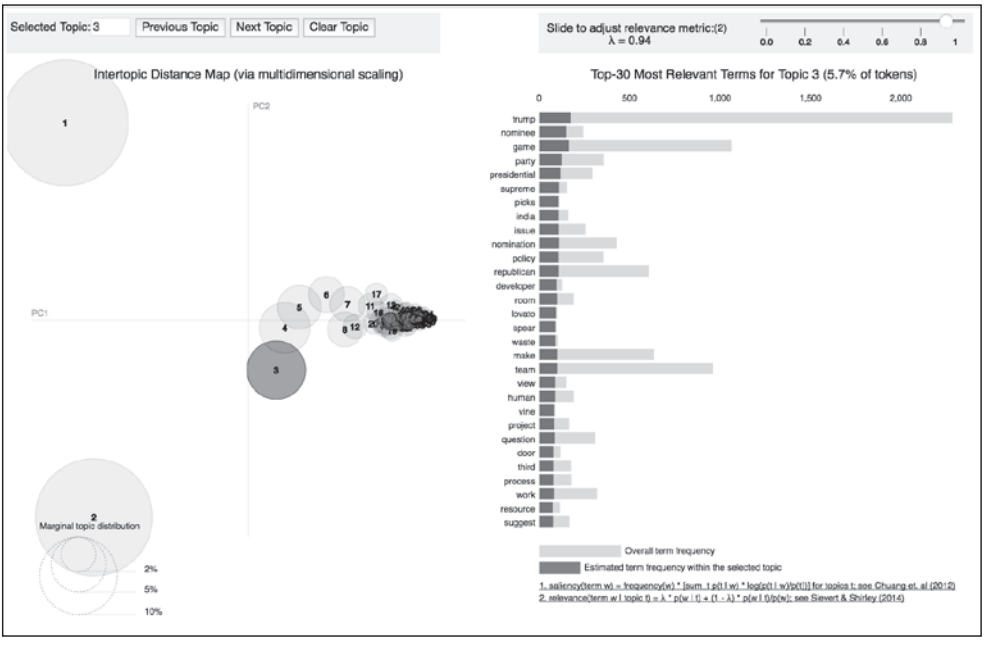


Рис. 6.11. Интерактивное визуальное представление тематической модели, полученное с помощью pyLDAvis

Латентно-семантический анализ

Латентно-семантический анализ (Latent Semantic Analysis, LSA) — это решение на основе векторов, впервые предложенное для тематического моделирования Дирвестером (Deerwester) с коллегами в 1990 г.[1]

В отличие от латентного размещения Дирихле, извлекающего темы из документов, которые затем можно использовать для классификации документов по доле тематических слов в них, латентно-семантический анализ просто отыскивает группы документов с одинаковыми словами. Метод LSA для тематического моделирования (также известный как *латентно-семантическое индексирование* — Latent Semantic Indexing) определяет темы внутри корпуса, создавая разреженную матрицу «лексемы/документы», где строками являются лексемы, а столбцами — документы. Каждое значение в матрице соответствует частоте встречаемости данной лексемы в данном документе и может нормализоваться с помощью TF–IDF. Затем к матрице можно применить метод *сингулярного разложения* (Singular Value Decomposition, SVD) для ее разложения на матрицы, представляющие собой комбинации «лексемы/темы», важность тем и «темы/документы».

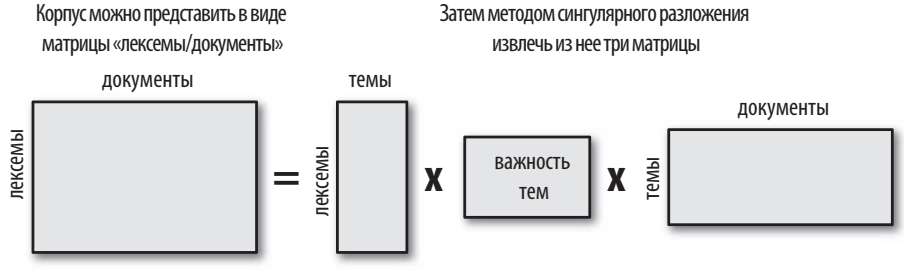


Рис. 6.12. Латентно-семантический анализ

По производной диагональной матрице важностей тем можно определить, какие темы являются наиболее важными в корпусе, и удалить строки, соответствующие наименее важным тематическим лексемам. По оставшимся строкам (лексемы) и столбцам (документы) можно определить темы, имеющие наибольший вес важности.

[1] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman, *Indexing by Latent Semantic Analysis* (1990), http://bit.ly/2JHWx57

С применением Scikit-Learn

Чтобы выполнить латентно-семантический анализ с помощью Scikit-Learn, создадим конвейер, который нормализует текст, создает матрицу «лексемы/документы» с помощью `CountVectorizer` и передает ее классу `TruncatedSVD`, реализующему сингулярное разложение. Реализация в Scikit-Learn вычисляет только k наибольших сингулярных значений, где k — это гиперпараметр, который мы должны определить через атрибут `n_components`. К счастью, нам почти ничего не придется делать, мы лишь изменим немного метод `__init__` нашего класса `SklearnTopicModels`!

```
class SklearnTopicModels(object):

    def __init__(self, n_topics = 50, estimator = 'LDA'):
        """
        n_topics — желаемое количество тем
        Чтобы выполнить латентно-семантический анализ, нужно присвоить
        параметру estimator строку 'LSA', потому что иначе по умолчанию
        будет выполнено латентное размещение Дирихле ('LDA').
        """
        self.n_topics = n_topics

        if estimator == 'LSA':
            self.estimator = TruncatedSVD(n_components = self.n_topics)
        else:
            self.estimator = LatentDirichletAllocation(n_topics =
                                self.n_topics)

        self.model = Pipeline([
            ('norm', TextNormalizer()),
            ('tfidf', CountVectorizer(tokenizer = identity,
                                      preprocessor = None, lowercase = False)),
            ('model', self.estimator)
        ])
```

Оригинальные методы `fit_transform` и `get_topics`, которые были определены в реализации латентного размещения Дирихле с применением Scikit-Learn, не требуют изменений для нормальной работы в обновленном классе `SklearnTopicModels`. Благодаря этому легко можно переключаться между двумя алгоритмами, чтобы определить, какой из них дает лучшие результаты в контексте нашего приложения и корпуса документов.

С применением Gensim

Добавление поддержки латентно-семантического анализа в класс `GensimTopicModels` осуществляется почти так же. Мы снова задействуем `TextNormalizer`

и `GensimTfidfVectorizer`, которые использовались в конвейере латентного размещения Дирихле, и в конец добавим обертку `LsiModel` из Gensim, доступную в модуле `gensim.sklearn_api` как `lsimodel.LsiTransformer`:

```
from gensim.sklearn_api import lsimodel, ldamodel

class GensimTopicModels(object):

    def __init__(self, n_topics = 50, estimator = 'LDA'):
        """
        n_topics — желаемое количество тем

        Чтобы выполнить латентно-семантический анализ, нужно присвоить
        параметру estimator строку 'LSA', потому что иначе по умолчанию
        будет выполнено латентное размещение Дирихле ('LDA').
        """
        self.n_topics = n_topics

        if estimator == 'LSA':
            self.estimator = lsimodel.LsiTransformer(num_topics =
                                self.n_topics)
        else:
            self.estimator = ldamodel.LdaTransformer(num_topics =
                                self.n_topics)

        self.model = Pipeline([
            ('norm', TextNormalizer()),
            ('vect', GensimTfidfVectorizer()),
            ('model', self.estimator)
        ])
```

Теперь легко можно переключаться между двумя алгоритмами, меняя значение именованного аргумента `estimator`.

Неотрицательное матричное разложение

Еще один прием обучения без учителя, который можно применять для тематического моделирования, — *неотрицательное матричное разложение* (Non-Negative Matrix Factorization, NNMF). Впервые предложенный Пентти Паатеро (Pentti Paatero) и Унто Таппером (Unto Tapper) в 1994 г.[1] и популярно описанный в статье *Nature* Дэниелом Ли (Daniel Lee) и Себастьяном Сеунгом (H. Sebastian Seung) в 1999 г.[2], метод NNMF имеет множество практических

[1] Pentti Paatero and Unto Tapper, *Positive matrix factorization: A non-negative factor model with optimal utilization of error estimates of data values* (1994), http://bit.ly/2GOFdJU

[2] Daniel D. Lee and H. Sebastian Seung, *Learning the parts of objects by non-negative matrix factorization* (1999), http://bit.ly/2GJBIV5

применений, включая спектральный анализ данных, совместную фильтрацию для рекомендательных систем и извлечение тем (рис. 6.13).

Применение NNMF для тематического моделирования мы начнем с представления корпуса в виде нормализованной матрицы «лексемы/документы», как это делалось в латентно-семантическом анализе. Затем разложим эту матрицу на две, произведение которых аппроксимирует оригинал, так чтобы каждое значение в обеих матрицах было неотрицательным. Полученные матрицы иллюстрируют темы, имеющие положительные связи с лексемами и документами в корпусе.

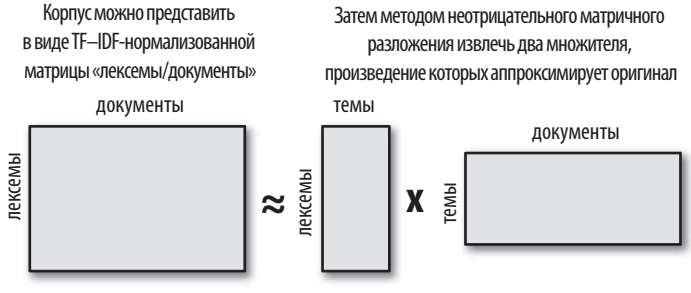


Рис. 6.13. Неотрицательное матричное разложение

С применением Scikit-Learn

Неотрицательное матричное разложение в библиотеке Scikit-Learn реализует класс `sklearn.decomposition.NMF`. Немного изменив метод `__init__` нашего класса `SklearnTopicModels`, мы легко можем реализовать NNMF в нашем конвейере без каких-либо других изменений в коде:

```
from sklearn.decomposition import NMF

class SklearnTopicModels(object):

    def __init__(self, n_topics = 50, estimator = 'LDA'):
        """
        n_topics — желаемое количество тем
        Чтобы выполнить латентно-семантический анализ, нужно присвоить
        параметру estimator строку 'LSA'.
        Чтобы использовать неотрицательное матричное разложение, нужно
        присвоить параметру estimator строку 'NMF', иначе по умолчанию
        будет выполнено латентное размещение Дирихле ('LDA').
        """
        self.n_topics = n_topics
```

```
        if estimator == 'LSA':
            self.estimator = TruncatedSVD(n_components = self.n_topics)
        elif estimator == 'NMF':
            self.estimator = NMF(n_components = self.n_topics)
        else:
            self.estimator = LatentDirichletAllocation(n_topics =
                                self.n_topics)

        self.model = Pipeline([
            ('norm', TextNormalizer()),
            ('tfidf', CountVectorizer(tokenizer = identity,
                                      preprocessor = None, lowercase = False)),
            ('model', self.estimator)
        ])
```

Итак, какой алгоритм тематического моделирования лучше? Как показывает опыт[1], метод LSA иногда лучше подходит для выявления описательных тем, что полезно в случаях, когда есть более длинные документы и более рассеянные корпусы. С другой стороны, латентное размещение Дирихле и неотрицательное матричное разложение могут давать более качественные результаты в выявлении компактных тем, что полезно для создания коротких меток.

В конечном счете качество модели во многом зависит от корпуса и целей приложения. Теперь у вас есть все необходимые инструменты для экспериментов с несколькими моделями, и вы сами сможете определить, какая из них дает лучшие результаты в вашем случае!

В заключение

В этой главе мы увидели, что обучение без учителя может оказаться непростым делом из-за отсутствия надежного способа оценить качество модели. Тем не менее существуют методы, позволяющие количественно оценить сходство документов, которые могут быстро и эффективно обрабатывать большие корпусы и представлять интересную и актуальную информацию.

Как уже говорилось, метод *k*-средних — это эффективный и универсальный метод кластеризации, который хорошо масштабируется для обработки больших корпусов (в частности, если используется реализация NLTK с косинусным расстоянием или `MiniBatchKMeans` из Scikit-Learn), особенно если кластеров не очень много и геометрия не слишком сложна. В случаях с большим количеством

[1] Keith Stevens, Philip Kegelmeyer, David Andrzejewski, and David Buttler, *Exploring Topic Coherence over many models and many topics* (2012), http://bit.ly/2GNHg11

кластеров и менее равномерно распределенными данными хорошей альтернативой может оказаться агломеративная кластеризация.

Мы также узнали, что для эффективного обобщения корпуса документов без меток часто требуется не только предварительная классификация, но и метод описания категорий. Это делает тематическое моделирование — методом латентного размещения Дирихле, латентно-сематического анализа или неотрицательного матричного разложения — еще одним важным инструментом в арсенале прикладного анализа текстов на естественном языке.

Кластеры могут стать отличной отправной точкой для аннотирования набора данных с целью дальнейшего применения методов обучения с учителем. Создание коллекций похожих документов может привести к созданию сложных структур, таких как отношения в графе (которые мы увидим в главе 9), способствующих увеличению эффективности последующего анализа. В главе 7 мы подробнее рассмотрим некоторые более совершенные стратегии конструирования контекстных признаков, которые позволят нам выявить некоторые из этих сложных структур и сделать моделирование более эффективным.

7 Контекстно-зависимый анализ текста

Модели, которые вы видели выше в этой книге, используют метод анализа текста как *мешка слов*, который позволяет исследовать отношения между документами, содержащими одинаковые наборы из отдельных слов. Это очень полезно, и мы действительно видели, что оценка частоты встречаемости лексем может быть очень эффективна, когда словарь терминов конкретной области или темы достаточно богат, чтобы можно было определить принадлежность или непринадлежность к ней другого текста.

Однако до сих пор мы не учитывали *контекст* появления слов, который, как мы знаем по опыту, играет важную роль в передаче смысла. Рассмотрим следующие фразы: «она любит запах роз» и «она пахнет розами». Если использовать приемы нормализации, представленные в предыдущих главах, такие как удаление стоп-слов и лемматизация, эти два высказывания превратятся в идентичные векторы мешков слов, хотя в действительности имеют совершенно разный смысл.

Это не значит, что модели, работающие с представлениями типа мешок слов, можно полностью сбросить со счетов — на самом деле такие модели обычно очень полезны на начальном этапе анализа. Кроме того, качество неэффективных моделей нередко можно значительно улучшить, добавив извлечение контекстных признаков. Один простой, но эффективный подход заключается в дополнении моделей *грамматиками* для создания шаблонов, которые помогут акцентироваться на определенных типах фраз, передающих больше нюансов, чем отдельные слова.

Эту главу мы начнем с использования грамматики для извлечения из документов *ключевых фраз*. Затем мы исследуем применение *n-грамм* для выявления

важных словосочетаний, которые можно использовать для расширения модели мешка слов. Наконец, мы посмотрим, как можно расширить модель на основе *n*-грамм с использованием оценки *условного распределения частот*, *сглаживания* и *возвратов*, чтобы получить модель, способную генерировать язык, — важный элемент многих приложений, включая машинный перевод, чат-боты, интеллектуальное автодополнение и многие другие.

Извлечение признаков на основе грамматики

Грамматические признаки, такие как части речи, позволяют закодировать дополнительную информацию о языке. Одним из наиболее эффективных способов улучшения качества модели является внедрение *грамматик* и *парсеров* для создания облегченных синтаксических структур, непосредственно затрагивающих динамические коллекции текста, которые могут иметь большое значение.

Чтобы получить информацию о языке, на котором написано предложение, нам понадобится набор грамматических правил, определяющих компоненты верно оформленных предложений на этом языке — вот что дает *грамматика*. Грамматика — это набор правил, описывающих, как синтаксические единицы (предложения, фразы и т. д.) в языке должны делиться на составляющие их элементы. Вот несколько примеров таких синтаксических категорий:

Символ	Синтаксическая категория
S	Предложение (Sentence)
NP	Именное словосочетание (Noun Phrase)
VP	Глагольное словосочетание (Verb Phrase)
PP	Предложное словосочетание (Prepositional Phrase)
DT	Определяющее слово (Determiner)
N	Существительное (Noun)
V	Глагол (Verb)
ADJ	Прилагательное (Adjective)
P	Предлог (Preposition)
TV	Переходный глагол (Transitive Verb)
IV	Непереходный глагол (Intransitive Verb)

Контекстно-свободные грамматики

С помощью грамматик можно определить разнообразные правила для сборки фраз или фрагментов из частей речи. *Контекстно-свободная грамматика* — это набор правил объединения синтаксических компонентов в осмысленные строки. Например, именное словосочетание «the castle» (замок) включает определяющее слово (обозначаемое тегом `DT` из набора Penn Treebank) и существительное (`N`). Предложное словосочетание (`PP`) «in the castle» (в замке) включает предлог (`P`) и именное словосочетание (`NP`). Глагольное словосочетание (`VP`) «looks in the castle» (заглянул в замок) включает глагол (`V`) и предложное словосочетание (`PP`). Предложение (`S`) «Gwen looks in the castle» (Гвен заглянула в замок) включает имя собственное (`NNP`) и глагольное словосочетание (`VP`). Используя эти теги, можно определять контекстно-свободную грамматику:

```
GRAMMAR = """
    S -> NNP VP
    VP -> V PP
    PP -> P NP
    NP -> DT N
    NNP -> 'Gwen' | 'George'
    V -> 'looks' | 'burns'
    P -> 'in' | 'for'
    DT -> 'the'
    N -> 'castle' | 'ocean'
    """
```

В библиотеке NLTK есть объект `nltk.grammar.CFG`, определяющий контекстно-свободную грамматику и взаимосвязи между разными синтаксическими элементами. Мы можем использовать этот объект для парсинга нашей грамматики:

```
from nltk import CFG
cfg = nltk.CFG.fromstring(GRAMMAR)

print(cfg)
print(cfg.start())
print(cfg.productions())
```

Синтаксические парсеры

Теперь, когда у нас имеется грамматика, нам нужен механизм, выполняющий систематический поиск осмысленных синтаксических структур в корпусе; механизм, играющий эту роль, называют *парсером*. Если грамматика определяет критерий поиска «осмысленности» в контексте нашего языка, то парсер выполняет поиск. *Синтаксический парсер* — это программный компонент, ко-

торый преобразует предложения в дерево синтаксического анализа, состоящее из иерархических элементов, или синтаксических категорий.

Когда парсер встречает предложение, он проверяет соответствие его структуры известной грамматике и, если такое есть, выполняет парсинг предложения согласно правилам этой грамматики, производя дерево синтаксического анализа. Парсеры часто используются для выявления важных структур, таких как субъект и объект действия в предложении или последовательности слов, которые должны быть сгруппированы в каждой синтаксической категории.

Сначала определим грамматику `GRAMMAR` для выявления в тексте предложений, соответствующих шаблону следования частей речи, а затем создадим экземпляр `RegexpParser` из NLTK, использующий нашу грамматику для деления текста на фрагменты:

```
from nltk.chunk.regexp import RegexpParser

GRAMMAR = r'KT: {(<JJ>* <NN.*>+ <IN>)? <JJ>* <NN.*>+}'
chunker = RegexpParser(GRAMMAR)
```

В данном случае `GRAMMAR` — это регулярное выражение, которое будет использоваться парсером `RegexpParser` для создания деревьев с меткой `KT` (key term — важный термин). Наш экземпляр `chunker` будет искать фразы, начинающиеся с необязательного компонента, включающего ноль или более прилагательных с одним или несколькими существительными разного вида и предлогами. Вслед за необязательным компонентом могут следовать ноль или больше прилагательных и одно или более существительных любого вида. Этой грамматике соответствуют такие фразы, как «red baseball bat» (красная бейсбольная бита) или «United States of America» (Соединенные Штаты Америки).

Рассмотрим пример предложения из новостей о бейсболе: «Dusty Baker proposed a simple solution to the Washington National's early-season bullpen troubles Monday afternoon and it had nothing to do with his maligned group of relievers» (Дасти Бейкер предложил в понедельник днем простое решение проблем с подачами, преследовавших команду «Вашингтон Нэшнлс» в начале сезона, и оно не связано с его группой раскритикованных питчеров).

```
(S
  (KT Dusty/NNP Baker/NNP)
  proposed/VBD
  a/DT
  (KT simple/JJ solution/NN)
  to/TO
```

```
the/DT
(KT Washington/NNP Nationals/NNP)
(KT
  early-season/JJ
  bullpen/NN
  troubles/NNS
  Monday/NNP
  afternoon/NN)
and/CC
it/PRP
had/VBD
(KT nothing/NN)
to/TO
do/VB
with/IN
his/PRP$
maligned/VBN
(KT group/NN of/IN relievers/NNS)
./.)
```

Это предложение было разделено на шесть фраз, включая: «Dusty Baker», «early-season bullpen troubles Monday afternoon» и «group of relievers».

Извлечение ключевых фраз

На рис. 4.8 изображен конвейер, включающий этап объединения признаков с использованием классов извлечения ключевых фраз и сущностей — `KeyphraseExtractor` и `EntityExtractor`. В этом разделе мы реализуем класс `KeyphraseExtractor`, преобразующий документы в представление вида «мешок фраз».

Ключевые термины и *фразы*, содержащиеся внутри корпуса, часто позволяют получать представление о теме анализируемых документов или о сущностях, содержащихся в них. Извлечение ключевых фраз заключается в идентификации и выделении фраз динамического размера, чтобы охватить как можно больше нюансов в темах документов.

Наш класс `KeyphraseExtractor` создан под влиянием превосходной статьи Бартона Де Вильде (Burton DeWilde)[1].

Первый шаг в извлечении ключевых фраз — определение кандидатов на фразы (например, какие слова или фразы лучше передают тему или взаимосвязи в до-

[1] Burton DeWilde, *Intro to Automatic Keyphrase Extraction* (2014), http://bit.ly/2GJBKwb

кументах). Мы определим `KeyphraseExtractor` с использованием грамматики и парсера, выявляющего *именные словосочетания* в тексте, маркированном тегами частей речи.

```
GRAMMAR = r'KT: {(<JJ>* <NN.*>+ <IN>)? <JJ>* <NN.*>+}'
GOODTAGS = frozenset(['JJ','JJR','JJS','NN','NNP','NNS','NNPS'])

class KeyphraseExtractor(BaseEstimator, TransformerMixin):
    """
    Обертывает PickledCorpusReader, содержащий маркированные документы.
    """
    def __init__(self, grammar = GRAMMAR):
        self.grammar = GRAMMAR
        self.chunker = RegexpParser(self.grammar)
```

Так как предполагается, что `KeyphraseExtractor` будет выполняться первым в конвейере после лексемизации, добавим метод `normalize()`, выполняющий некоторую простую нормализацию текста, удаляя знаки препинания и преобразуя все буквы в нижний регистр:

```
from unicodedata import category as unicat

    def normalize(self, sent):
        """
        Удаляет знаки препинания из лексемизированного/маркированного
        предложения и преобразует буквы в нижний регистр.
        """
        is_punct = lambda word: all(unicat(c).startswith('P') for c in word)
        sent = filter(lambda t: not is_punct(t[0]), sent)
        sent = map(lambda t: (t[0].lower(), t[1]), sent)
        return list(sent)
```

Теперь напишем метод `extract_keyphrases()`. Этот метод принимает документ и сначала нормализует его текст, а затем использует наш парсер для синтаксического анализа. Результатом работы парсера является дерево с несколькими наиболее интересными ветвями (ключевыми фразами!). Чтобы извлечь эти фразы, используем функцию `tree2conlltags` для преобразования дерева в формат разметки CoNLL IOB — список кортежей `(word, tag, IOB-tag)`.

Тег IOB сообщает положение термина в контексте фразы; термин может *начинать* ключевую фразу (`B-KT`), находиться *в середине* (`I-KT`) или *за пределами* (`O`) ключевой фразы. Поскольку нас интересуют только термины, являющиеся частью ключевой фразы, используем функцию `groupby()` из пакета `itertools` стандартной библиотеки, чтобы написать лямбда-функцию, продолжающую группировать термины, пока не встретится тег `O`:

```
from itertools import groupby
from nltk.chunk import tree2conlltags

    def extract_keyphrases(self, document):
        """
        Выполняет парсинг предложений из документа, используя наш парсер
        с грамматикой, преобразует дерево синтаксического анализа в
        маркированную последовательность.
        Возвращает извлеченные фразы.
        """
        for sents in document:
            for sent in sents:
                sent = self.normalize(sent)
                if not sent: continue
                chunks = tree2conlltags(self.chunker.parse(sent))
                phrases = [
                    " ".join(word for word, pos, chunk in group).lower()
                    for key, group in groupby(
                        chunks, lambda term: term[-1] ! = 'O'
                    ) if key
                ]
                for phrase in phrases:
                    yield phrase
```

Поскольку наш класс фактически является преобразователем, завершим его определение пустым методом `fit` и методом `transform`, который вызывает `extract_keyphrases()` для каждого документа в корпусе:

```
    def fit(self, documents, y = None):
        return self

    def transform(self, documents):
        for document in documents:
            yield self.extract_keyphrases(document)
```

Вот пример результата преобразования одного из документов:

```
['lonely city', 'heart piercing wisdom', 'loneliness', 'laing',
'everyone', 'feast later', 'point', 'own hermetic existence in new york',
'danger', 'thankfully', 'lonely city', 'cry for connection',
'overcrowded overstimulated world', 'blueprint of urban loneliness',
'emotion', 'calls', 'city', 'npr jason heller', 'olivia laing',
'lonely city', 'exploration of loneliness',
'others experiences in new york city', 'rumpus', 'review', 'lonely city',
'related posts']
```

В главе 12 мы вновь вернемся к этому классу, но уже с другой грамматикой, и создадим свой преобразователь вида «мешок ключевых фраз» для классификатора эмоциональной окраски на основе нейронной сети.

Извлечение сущностей

По аналогии с извлечением ключевых фраз можно также реализовать свой механизм извлечения сущностей, преобразующий документы в «мешки сущностей». Используем для этого утилиту `ne_chunk` распознавания именованных сущностей из NLTK, которая создает вложенную древовидную структуру с синтаксическими категориями и тегами частей речи, содержащимися в каждом предложении.

Начнем создание класса `EntityExtractor` с инициализации его множеством меток сущностей. Затем добавим метод `get_entities`, использующий утилиту `ne_chunk` для получения дерева синтаксического анализа данного документа. Затем этот метод выполняет обход поддеревьев, извлекает сущности с метками, присутствующими в нашем множестве (имена людей, названия организаций, объектов, геополитические сущности и геосоциальные политические сущности). Найденные сущности добавляются в конец списка `entities`, который возвращается методом по завершении обхода всех поддеревьев документа:

```
from nltk import ne_chunk

GOODLABELS = frozenset(['PERSON', 'ORGANIZATION', 'FACILITY', 'GPE', 'GSP'])

class EntityExtractor(BaseEstimator, TransformerMixin):
    def __init__(self, labels = GOODLABELS, **kwargs):
        self.labels = labels

    def get_entities(self, document):
        entities = []
        for paragraph in document:
            for sentence in paragraph:
                trees = ne_chunk(sentence)
                for tree in trees:
                    if hasattr(tree, 'label'):
                        if tree.label() in self.labels:
                            entities.append(
                                ' '.join([child[0].lower() for child in tree])
                            )
        return entities

    def fit(self, documents, labels = None):
        return self

    def transform(self, documents):
        for document in documents:
            yield self.get_entities(document)
```

Вот пример документа из корпуса:

```
['lonely city', 'loneliness', 'laing', 'new york', 'lonely city',
'npr', 'jason heller', 'olivia laing', 'lonely city', 'new york city',
'rumpus', 'lonely city', 'related']
```

Мы еще вернемся к извлечению признаков на основе грамматик в главе 9, где используем наш класс `EntityExtractor` совместно с графами для моделирования относительной важности разных сущностей в документах.

Извлечение признаков на основе *n*-грамм

К сожалению, подходы на основе грамматик, даже самые эффективные, не всегда дают хороший результат. Во-первых, они в значительной степени зависят от успешной маркировки слов тегами частей речи, то есть мы должны быть уверены, что наш механизм маркировки правильно определяет существительные, глаголы, прилагательные и другие части речи. Как будет показано в главе 8, механизмы разметки частей речи с настройками по умолчанию легко ошибаются при анализе нестандартного или безграмотного текста.

Извлечение признаков на основе грамматики страдает некоторой негибкостью, требуя предварительно определить грамматику. Однако часто очень трудно заранее понять, какой грамматический шаблон позволит наиболее эффективно выявить в тексте значимые термины и фразы.

Эти проблемы можно решать итеративно, экспериментируя с множеством разных грамматик или обучив свою модель для расстановки частей речи. Но в этом разделе мы исследуем другой подход, основанный на применении *n-грамм*, дающий более обобщенный способ идентификации последовательностей лексем.

Рассмотрим предложение «The reporters listened closely as the President of the United States addressed the room» (Журналисты внимательно слушали обращение президента Соединенных Штатов к залу). Сканируя всю последовательность окном фиксированной ширины n, можно собрать все возможные непрерывные последовательности лексем. До сих пор мы работали с униграммами — n-граммами с $n = 1$ (то есть отдельными лексемами). С шириной окна $n = 2$ мы получаем биграммы, кортежи лексем, такие как (`"The"`, `"reporters"`) и (`"reporters"`, `"listened"`). С $n = 3$ — триграммы, то есть трехэлементные кортежи: (`"The"`, `"reporters"`, `"listened"`) и так далее, для любого значения n. На рис. 7.1 показана последовательность окон для выделения триграмм.

Рис. 7.1. Выделение *n*-грамм скользящим окном

Чтобы выделить из текста все *n*-граммы, достаточно просканировать список слов скользящим окном фиксированной ширины. Сделать это на Python можно следующим образом:

```
def ngrams(words, n = 2):
    for idx in range(len(words)-n+1):
        yield tuple(words[idx:idx+n])
```

Эта функция перебирает значения индекса от `0` до позиции, находящейся от конца списка слов точно на расстоянии ширины окна. Затем последовательно выбирает и возвращает *n*-граммы в виде неизменяемых кортежей. Если применить эту функцию к примеру предложения выше, мы получим:

```
words = [
    "The", "reporters", "listened", "closely", "as", "the", "President",
    "of", "the", "United", "States", "addressed", "the", "room", ".",
]

for ngram in ngrams(words, n = 3):
    print(ngram)

('The', 'reporters', 'listened')
('reporters', 'listened', 'closely')
('listened', 'closely', 'as')
('closely', 'as', 'the')
('as', 'the', 'President')
('the', 'President', 'of')
('President', 'of', 'the')
('of', 'the', 'United')
('the', 'United', 'States')
('United', 'States', 'addressed')
('States', 'addressed', 'the')
('addressed', 'the', 'room')
('the', 'room', '.')
```

Неплохо! Однако эти результаты вызывают несколько вопросов. Первый: как обрабатывать начало и конец предложений? И второй: как определить оптимальный размер *n*-грамм? Ответы на оба вопроса мы рассмотрим в следующем разделе.

Чтение корпуса с поддержкой *n*-грамм

Извлечение *n*-грамм — это часть процедуры обработки текста, выполняемой перед моделированием. Поэтому было бы удобно включить метод `ngrams()` в состав классов `CorpusReader` и `PickledCorpusReader`. Это позволит легко выделить *n*-граммы из всего корпуса и извлекать их позднее. Например:

```
class HTMLCorpusReader(CategorizedCorpusReader, CorpusReader):

    ...

    def ngrams(self, n = 2, fileids = None, categories = None):
        for sent in self.sents(fileids = fileids, categories = categories):
            for ngram in nltk.ngrams(sent, n):
                yield ngram
    ...
```

Поскольку нас интересует в первую очередь *контекст*, и каждое предложение представляет законченную мысль, есть смысл принимать во внимание только *n*-граммы, не пересекающие границ между предложениями.

Самый простой путь обеспечить соблюдение сложных правил отбора *n*-грамм — воспользоваться методом `ngrams()` из библиотеки NLTK, который можно сочетать с методами фрагментирования и выделения лексем из этой же библиотеки. Метод `ngrams()` позволяет добавлять специальные маркеры до и после предложений, чтобы *n*-граммы также включали границы между предложениями. Это даст возможность определить, какие *n*-граммы начинают и заканчивают предложения.

Здесь для обозначения начала и конца предложений мы используем символы XML, потому что они легко идентифицируются как разметка и почти наверняка не являются уникальными лексемами в тексте. Однако наш выбор во многом произволен, и с не меньшим успехом можно использовать другие символы. Например, мы часто используем ★ (`"\u2605"`) и ☆ (`"\u2606"`) для парсинга текстов, не содержащих этих символов.

Сперва определим константы для обозначения начала и конца предложения как `<s>` и `</s>` (так как в английском языке слова записываются слева направо, назовем константы `left_pad_symbol` и `right_pad_symbol` соответственно). В языках с письмом справа налево эти константы могли бы иметь противоположный смысл.

Во второй части кода определяется функция `nltk_ngrams`, использующая функцию `partial`, чтобы создать свою обертку для функции `nltk.ngrams`, которая под-

ставляет значения в именованные аргументы. Это позволит нам гарантировать требуемое поведение `nltk_ngrams` при каждом вызове, не повторяя полностью сигнатуру вызова везде в нашем коде, где эти вызовы производятся. Наконец, наша новая переопределенная функция `ngrams` принимает строку с текстом и размер *n*-грамм. Затем она применяет к тексту функции `sent_tokenize` и `word_tokenize` и передает результат в вызов `nltk_ngrams` для получения *n*-грамм:

```
import nltk
from functools import partial

LPAD_SYMBOL = "<s>"
RPAD_SYMBOL = "</s>"

nltk_ngrams = partial(
    nltk.ngrams,
    pad_right = True, right_pad_symbol = RPAD_SYMBOL,
    left_pad = True, left_pad_symbol = LPAD_SYMBOL
)

    def ngrams(self, n = 2, fileids = None, categories = None):
        for sent in self.sents(fileids = fileids, categories = categories):
            for ngram in nltk.ngrams(sent, n):
                yield ngram
```

Например, для значения $n = 4$ и текста «After, there were several follow-up questions. The *New York Times* asked when the bill would be signed» (Затем последовало несколько вопросов. Репортер из *New York Times* спросил, когда будет пописан закон) будут получены следующие тетраграммы:

```
('<s>', '<s>', '<s>', 'After')
('<s>', '<s>', 'After', ',')
('<s>', 'After', ',', 'there')
('After', ',', 'there', 'were')
(',', 'there', 'were', 'several')
('there', 'were', 'several', 'follow')
('were', 'several', 'follow', 'up')
('several', 'follow', 'up', 'questions')
('follow', 'up', 'questions', '.')
('up', 'questions', '.', '</s>')
('questions', '.', '</s>', '</s>')
('.', '</s>', '</s>', '</s>')
('<s>', '<s>', '<s>', 'The')
('<s>', '<s>', 'The', 'New')
('<s>', 'The', 'New', 'York')
('The', 'New', 'York', 'Times')
('New', 'York', 'Times', 'asked')
('York', 'Times', 'asked', 'when')
('Times', 'asked', 'when', '</s>')
('asked', 'when', '</s>', '</s>')
('when', '</s>', '</s>', '</s>')
```

Обратите внимание, что функция добавляет маркеры во все возможные последовательности *n*-грамм. Эта ее особенность пригодится нам позже, когда мы будем обсуждать *возвраты*, но, если вам достаточно знать, где находятся начало и конец последовательности, можете просто отфильтровать *n*-граммы, содержащие более одного маркера.

Выбор размера *n*-грамм

А теперь займемся выбором размера *n*-грамм. Рассмотрим приложение, которое использует *n*-граммы для выявления кандидатов на именованные сущности. Если выбрать размер $n = 2$, мы получим в результате «The reporters», «the President», «the United» и «the room». Хотя эта модель далеко не идеальна, она успешно идентифицировала три соответствующих сущностей без особых вычислительных затрат.

С другой стороны, модель с размером окна 2 для выделения *n*-грамм не способна уловить некоторые нюансы исходного текста. Например, если предложение в тексте упоминает нескольких глав государств, тогда биграмма «the President» может оказаться неоднозначной. Чтобы охватить все словосочетание «the President of the United States», мы должны выбрать $n = 6$:

```
('The', 'reporters', 'listened', 'closely', 'as', 'the'),
('reporters', 'listened', 'closely', 'as', 'the', 'President'),
('listened', 'closely', 'as', 'the', 'President', 'of'),
('closely', 'as', 'the', 'President', 'of', 'the'),
('as', 'the', 'President', 'of', 'the', 'United'),
n-Gram Feature Extraction | 135
('the', 'President', 'of', 'the', 'United', 'States'),
('President', 'of', 'the', 'United', 'States', 'addressed'),
('of', 'the', 'United', 'States', 'addressed', 'the'),
('the', 'United', 'States', 'addressed', 'the', 'room'),
('United', 'States', 'addressed', 'the', 'room', '.')
```

К сожалению, как можно судить по результатам выше, очень маловероятно, что результаты модели на основе *n*-грамм слишком большого размера будут включать повторяющиеся сущности. Это затруднит определение вероятностей, позволяющих выявить цели анализа. Кроме того, с увеличением *n* увеличивается число возможных правильных *n*-грамм, из-за чего снижается вероятность определения всех правильных *n*-грамм в корпусе. Слишком большое значение *n* может увеличить шум, перекрывающий независимые контексты. Если окно больше размеров предложений, оно вообще может не вернуть ни одной *n*-граммы.

Выбор *n* можно также рассматривать как компромисс между *систематическим отклонением* и *дисперсией*. При малом значении *n* получается более простая (сла-

бая) модель, вызывающая больше ошибок из-за систематического отклонения. При большом значении *n* получается более сложная модель (модель высокого порядка), вызывающая больше ошибок из-за дисперсии. Так же, как в случае с задачами машинного обучения с учителем, мы должны найти правильный баланс между чувствительностью и специфичностью модели. Чем больше зависимых слов находится на большом расстоянии от главного слова, тем большая сложность необходима для создания прогнозирующей модели на основе *n*-грамм.

Значимые словосочетания

Теперь, когда наш объект чтения корпуса поддерживает *n*-граммы, мы можем внедрить эти признаки в последующие модели путем векторизации текста с использованием *n*-грамм вместо простого словаря слов. Однако использование исходных *n*-грамм породит много, слишком много кандидатов, большинство из которых не являются релевантными. Например, предложение «I got lost in the corn maze during the fall picnic» (я заблудился в лабиринте кукурузы на осеннем пикнике) содержит триграмму `('in', 'the', 'corn')` (в кукурузе), не являющуюся типичной предложной целью, тогда как триграмма `('I', 'got', 'lost')` (я потерялся) имеет смысл сама по себе.

На практике обработка лишних *n*-грамм может обходиться слишком дорого для простых приложений с точки зрения вычислительных ресурсов. Эту проблему можно решить вычислением *условной вероятности*. Например, какова вероятность появления в тексте лексем `('the', 'fall')` вместе с лексемой `'during'`? Можно найти эмпирические вероятности, вычисляя частоту $(n - 1)$-граммы, которой предшествует первая лексема из *n*-граммы. Таким способом можно увеличить вес *n*-грамм из слов, чаще используемых вместе, таких как `('corn', 'maze')`, относительно редко используемых комбинаций, которые менее значимы.

Идея о более высокой ценности некоторых *n*-грамм ведет к еще одному инструменту в сфере анализа текста: *значимым словосочетаниям* (significant collocations). Слово collocation — это обобщенный синоним для *n*-граммы (без уточнения размера окна) и просто обозначает последовательность лексем, вероятность совместного появления которых обусловлена не только случайным совпадением. С помощью условных вероятностей можно проверить гипотезу о значимости выбранного словосочетания.

В библиотеке NLTK есть два инструмента для выявления значимых словосочетаний: `CollocatioFinder`, отыскивающий и оценивающий *n*-граммы словосочетаний, и `NgramAssocMeasures`, содержащий коллекцию метрик для оценки значимости словосочетания. Обе утилиты зависят от величины *n*, а кроме них

в модулях имеются утилиты оценки для работы с биграммами, триграммами и тетраграммами. К сожалению, утилиты для обработки *n*-грамм из 5 и большего количества лексем вам придется реализовать самостоятельно, расширяя соответствующий родительский класс и используя как шаблон один из инструментов обработки словосочетаний.

А теперь посмотрим, как производится выявление существенных тетраграмм. Так как поиск и оценка *n*-грамм в большом корпусе требует очень много времени, результаты рекомендуется сохранять в файл на диске. Напишем функцию `rank_quadgrams`, которая принимает корпус, откуда она будет читать слова, а также метрику из `QuadgramAssocMeasures`, отыскивает и оценивает тетраграммы, затем записывает результаты в файл на диск:

```
from nltk.collocations import QuadgramCollocationFinder
from nltk.metrics.association import QuadgramAssocMeasures

def rank_quadgrams(corpus, metric, path = None):
    """
    Находит и оценивает тетраграммы в указанном корпусе с применением
    заданной метрики. Записывает тетраграммы в файл, если указан,
    иначе возвращает список в памяти.
    """
    # Создать объект оценки словосочетаний из слов в корпусе.
    ngrams = QuadgramCollocationFinder.from_words(corpus.words())

    # Оценить словосочетания в соответствии с заданной метрикой
    scored = ngrams.score_ngrams(metric)

    if path:
        # Записать результаты в файл на диске
        with open(path, 'w') as f:
            f.write("Collocation\tScore ({})".format(metric.__name__))
            for ngram, score in scored:
                f.write("{}\t{}\n".format(repr(ngram), score))
    else:
        return scored
```

Например, вот как можно использовать метрику отношения подобия:

```
rank_quadgrams(
    corpus, QuadgramAssocMeasures.likelihood_ratio, 'quadgrams.txt'
)
```

Она вернет тетраграммы из корпуса с оценками подобия, например:

```
Collocation Score (likelihood_ratio)
('New', 'York', "'", 's') 156602.26742890902
('pictures', 'of', 'the', 'Earth')      28262.697780596758
```

```
('the', 'majority', 'of', 'users')      28262.36608379526
('numbed', 'by', 'the', 'mindlessness') 3091.139615301832
('There', 'was', 'a', 'time')   3090.2332736791095
```

Класс `QuadgramAssocMeasures` включает в себя несколько методов оценки значимости путем проверки гипотез. Эти методы предполагают, что между словами нет связи (основная гипотеза), а затем вычисляют вероятность ассоциации, если основная гипотеза верна. Если основная гипотеза имеет низкий уровень значимости и ее можно отвергнуть, мы можем принять альтернативную гипотезу.

Класс `QuadgramAssocMeasures` из библиотеки NLTK включает ряд инструментов проверки значимости с использованием таких критериев, как t-критерий Стъюдента, хи-квадрат Пирсона, поточечная взаимная информация, функция правдоподобия Пуассона—Стирлинга и даже коэффициент Жаккара. Для проверки ассоциаций биграмм предоставляется еще больше методов, таких как фи-квадрат (квадрат коэффициента корреляции Пирсона), точный критерий Фишера или коэффициент сходства Дайса.

Теперь можно представить преобразователь `SignificantCollocations`, предназначенный для использования в конвейере, таком как на рис. 7.2.

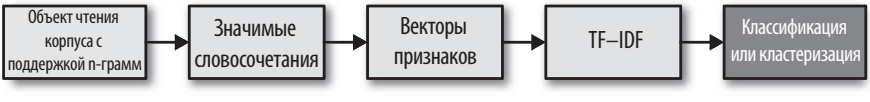


Рис. 7.2. Конвейер извлечения признаков в виде *n*-грамм

В методе `fit()` можно найти и оценить значимые словосочетания, а в методе `transform()` сгенерировать вектор, кодирующий оценки всех значимых словосочетаний, найденных в документе. Эти признаки затем можно объединить с другими векторами с помощью `FeatureUnion`.

```
from sklearn.base import BaseEstimator, TransformerMixin

class SignificantCollocations(BaseEstimator, TransformerMixin):

    def __init__(self,
                 ngram_class = QuadgramCollocationFinder,
                 metric = QuadgramAssocMeasures.pmi):
        self.ngram_class = ngram_class
        self.metric = metric

    def fit(self, docs, target):
```

```
        ngrams = self.ngram_class.from_documents(docs)
        self.scored_ = dict(ngrams.score_ngrams(self.metric))

    def transform(self, docs):
        for doc in docs:
            ngrams = self.ngram_class.from_words(docs)
            yield {
                ngram: self.scored_.get(ngram, 0.0)
                for ngram in ngrams.nbest(QuadgramAssocMeasures.raw_freq, 50)
            }
```

После этого можно сконструировать модель, как показано ниже:

```
from sklearn.linear_model import SGDClassifier
from sklearn.pipeline import Pipeline, FeatureUnion
from sklearn.feature_extraction import DictVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer

model = Pipeline([
    ('union', FeatureUnion(
        transformer_list = [
            ('ngrams', Pipeline([
                ('sigcol', SignificantCollocations()),
                ('dsigcol', DictVectorizer()),
            ])),

            ('tfidf', TfidfVectorizer()),
        ]
    ))

    ('clf', SGDClassifier()),
])
```

Обратите внимание, что это только заготовка кода, но ее можно использовать как шаблон для включения анализа контекста в стандартную модель типа «мешок слов».

Модели языка *n*-грамм

Рассмотрим приложение, которое принимает от пользователя несколько первых слов из фразы, а затем предлагает завершение этой фразы из наиболее вероятных слов (как это делает поисковая система Google). Для принятия решений модели на основе *n*-грамм используют статистическую частоту *n*-грамм. Чтобы получить модель языка *n*-грамм, предсказывающую слово, следующее за серией слов, можно сначала подсчитать количество всех *n*-грамм в тексте и затем использовать их частоты для предсказания вероятного продолжения. Теперь у нас

есть причина использовать значимые словосочетания не только как средство извлечения признаков, но также как модель языка!

Чтобы построить модель языка, способную генерировать текст, создадим класс, объединяющий все компоненты, которые мы рассмотрели в предыдущих разделах, и реализующий дополнительный прием: вычисление *условной частоты*.

Когда-то в библиотеке NLTK был модуль, позволявший генерировать текст на естественном языке, но он удален после появления проблем в методе создания моделей *n*-грамм. Идея классов `NgramModel` и `NgramCounter`, которые мы реализуем в этом разделе, появилась под влиянием ветви NLTK, решающей многие из этих недостатков, но на момент написания этих строк та ветвь все еще находилась на стадии разработки и не была объединена с главной ветвью.

Частота и условная частота

Впервые понятие частоты лексем мы рассмотрели на рис. 4.2, где использовали частотные представления в модели типа «мешок слов», предполагая, что частота слов в документах может служить значимой характеристикой содержимого документов для выявления различий между ними. Частота также является полезным признаком для моделей на основе *n*-грамм — можно предположить, что *n*-граммы с большой частотой появления в обучающем корпусе будут появляться и в новых документах.

Представьте, что мы читаем книгу, слово за словом, и нам нужно определить вероятность появления следующего слова. Самый простой и наивный способ — выбрать слово, чаще других появляющееся в тексте, как показано на рис. 7.3.

Однако мы знаем, что такого простейшего подхода на основе частот недостаточно; некоторые слова наиболее вероятны в начале предложений, а другие чаще появляются после определенных предшествующих им слов. Например, давайте определим, насколько вероятность следования слова «chair» (стул) за словом «lawn» (газон) отличается от вероятности следования слова «chair» (стул) за словом «lava» (лава) или «lamp» (лампа). Эти вероятности сообщаются *условными вероятностями* и обозначаются как `P(chair|lawn)` (читается как «вероятность появления слова chair за словом lawn»). Чтобы смоделировать эти вероятности, нужно иметь возможность вычисления *условных частот* для всех возможных *n*-грамм.

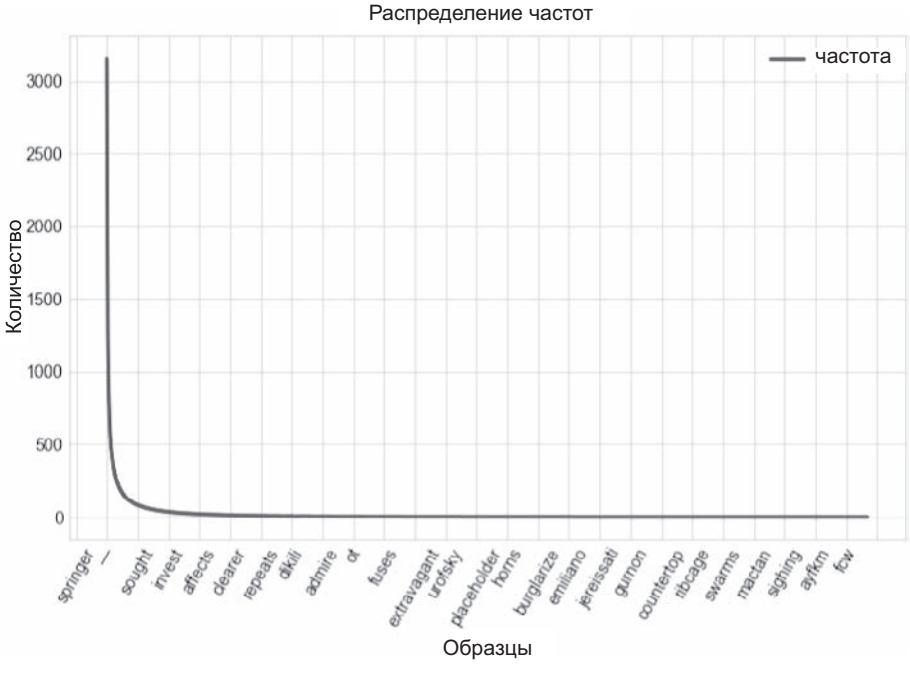


Рис. 7.3. График распределения частот в корпусе текста

Начнем с определения класса `NgramCounter`, который сможет определить условные частоты всех субграмм, от униграмм до *n*-грамм, с использованием `FreqDist` и `ConditionalFreqDist`. Наш класс также реализует дополнение предложений, как было показано выше в главе, и определяет слова, отсутствующие в словаре оригинального корпуса.

```
from nltk.util import ngrams
from nltk.probability import FreqDist, ConditionalFreqDist

from collections import defaultdict

# Символы дополнения
UNKNOWN = "<UNK>"
LPAD = "<s>"
RPAD = "</s>"

class NgramCounter(object):
    """
    Класс NgramCounter подсчитывает n-граммы для заданного словаря и размера
    окна.
    """
```

```
    def __init__(self, n, vocabulary, unknown = UNKNOWN):
        """
        n -- размер n-грамм
        """
        if n < 1:
            raise ValueError("ngram size must be greater than or equal to 1")

        self.n = n
        self.unknown = unknown
        self.padding = {
            "pad_left": True,
            "pad_right": True,
            "left_pad_symbol": LPAD,
            "right_pad_symbol": RPAD,
        }

        self.vocabulary = vocabulary
        self.allgrams = defaultdict(ConditionalFreqDist)
        self.ngrams = FreqDist()
        self.unigrams = FreqDist()
```

Далее добавим в класс `NgramCounter` метод, который позволит систематически вычислять распределение частот и условное распределение частот для заданного размера *n*-грамм.

```
    def train_counts(self, training_text):
        for sent in training_text:
            checked_sent = (self.check_against_vocab(word) for word in sent)
            sent_start = True
            for ngram in self.to_ngrams(checked_sent):
                self.ngrams[ngram] + = 1
                context, word = tuple(ngram[:-1]), ngram[-1]
                if sent_start:
                    for context_word in context:
                        self.unigrams[context_word] + = 1
                    sent_start = False

                for window, ngram_order in enumerate(range(self.n, 1, -1)):
                    context = context[window:]
                    self.allgrams[ngram_order][context][word] + = 1
                self.unigrams[word] + = 1

    def check_against_vocab(self, word):
        if word in self.vocabulary:
            return word
        return self.unknown

    def to_ngrams(self, sequence):
        """
        Обертка для метода ngrams из библиотеки NLTK
        """
        return ngrams(sequence, self.n, **self.padding)
```

Теперь определим функцию (за пределами класса `NgramCounter`), которая создает экземпляр класса и вычисляет релевантные частоты. Функция `count_ngrams` принимает параметр с желаемым размером *n*-грамм, словарь и список предложений в виде строк, разделенных запятыми.

```
def count_ngrams(n, vocabulary, texts):
    counter = NgramCounter(n, vocabulary)
    counter.train_counts(texts)
    return counter

if __name__ == '__main__':
    corpus = PickledCorpusReader('../corpus')
    tokens = [''.join(word[0]) for word in corpus.words()]
    vocab = Counter(tokens)
    sents = list([word[0] for word in sent] for sent in corpus.sents())
    trigram_counts = count_ngrams(3, vocab, sents)
```

Распределение частоты для униграмм можно получить из атрибута `unigrams`.

```
print(trigram_counts.unigrams)
```

Для *n*-грамм более высокого порядка условное распределение частот можно получить из атрибута `ngrams`.

```
print(trigram_counts.ngrams[3])
```

Ключи условного распределения частот показывают возможные контексты, предшествующие каждому слову.

```
print(sorted(trigram_counts.ngrams[3].conditions()))
```

Наша модель также способна возвращать список возможных следующих слов:

```
print(list(trigram_counts.ngrams[3][('the', 'President')]))
```

Оценка максимальной вероятности

Класс `NgramCounter` дает возможность преобразовать корпус в условное распределение частот *n*-грамм. В нашем воображаемом приложении, предсказывающем следующее слово, нам понадобится механизм оценки возможных кандидатов на роль слова, следующего за *n*-граммой, чтобы мы могли выбрать наиболее вероятного из них. Иными словами, нам нужна модель, определяющая вероятность лексемы `t` по предшествующей последовательности `s`.

Один из самых простых способов оценки вероятности *n*-граммы `(s,t)` основан на вычислении относительной частоты. Это количество появлений `t` вслед за `s`,

деленное на общее количество появлений `s` в корпусе. Это отношение дает оценку максимальной вероятности для *n*-граммы `(s,t)`.

Начнем с создания класса `BaseNgramModel`, конструктор которого принимает объект `NgramCounter` и создает модель языка. Инициализируем модель `BaseNgramModel` атрибутами для хранения *n*-грамм более высокого порядка из обученного объекта `NgramCounter`, самих *n*-грамм, их условных распределений частот и словаря.

```
class BaseNgramModel(object):
    """
    BaseNgramModel создает модель языка n-грамм.
    """

    def __init__(self, ngram_counter):
        """
        BaseNgramModel инициализируется объектом NgramCounter.
        """
        self.n = ngram_counter.n
        self.ngram_counter = ngram_counter
        self.ngrams = ngram_counter.ngrams[ngram_counter.n]
        self._check_against_vocab = self.ngram_counter.check_against_vocab
```

Затем добавим в класс `BaseNgramModel` метод `score` вычисления относительной частоты слова в данном контексте, который сначала проверит, что контекст короче *n*-грамм более высокого порядка из `NgramCounter`. Так как атрибут `ngrams` в `BaseNgramModel` является объектом `ConditionalFreqDist` из библиотеки NLTK, мы можем получить `FreqDist` для любого данного контекста и его относительную частоту из атрибута `freq`:

```
    def score(self, word, context):
        """
        Для данного строкового представления слова и строки с контекстом
        возвращает оценку максимальной вероятности, что данное слово
        продолжит этот контекст.

        fdist[context].freq(word) == fdist[(context, word)] / fdist[context]
        """
        context = self.check_context(context)

        return self.ngrams[context].freq(word)

    def check_context(self, context):
        """
        Проверяет длину контекста, которая должна быть меньше длины
        n-граммы высшего порядка модели.

        Возвращает контекст как кортеж.
```

```
        """
        if len(context) > = self.n:
            raise ValueError("Context too long for this n-gram")

        return tuple(context)
```

На практике вероятности *n*-грамм получаются небольшими, поэтому их часто представляют как логарифмы вероятностей. Поэтому добавим метод `logscore`, преобразующий результат метода `score` в логарифм, если оценка не равна нулю и не меньше его, иначе возвращающий отрицательную бесконечность:

```
    def logscore(self, word, context):
        """
        Для данного строкового представления слова и строки с контекстом
        вычисляет логарифм вероятности появления слова в данном контексте.
        """
        score = self.score(word, context)
        if score < = 0.0:
            return float("-inf")

        return log(score, 2)
```

Теперь, когда у нас есть методы оценки экземпляров *n*-грамм, реализуем метод оценки модели языка в целом, который будет вычислять *энтропию*. Добавим метод `entropy` в `BaseNgramModel`, определяющий средний логарифм вероятностей всех *n*-грамм из `NgramCounter`.

```
    def entropy(self, text):
        """
        Вычисляет приближенную перекрестную энтропию модели n-грамм
        для заданного текста в форме списка строк, разделенных запятыми.
        Это средний логарифм вероятности всех слов в тексте.
        """
        normed_text = (self._check_against_vocab(word) for word in text)
        entropy = 0.0
        processed_ngrams = 0
        for ngram in self.ngram_counter.to_ngrams(normed_text):
            context, word = tuple(ngram[:-1]), ngram[-1]
            entropy + = self.logscore(word, context)
            processed_ngrams + = 1
        return - (entropy / processed_ngrams)
```

В главе 1 мы столкнулись с понятием *неопределенности* и пришли к выводу, что для данного высказывания достаточно знать несколько первых слов, чтобы предсказать несколько последующих слов. Суть состоит в том, что смысл очень ограничен и является вариацией марковских допущений (Markov assumption). В случае с моделью *n*-грамм нам нужно минимизировать неопределенность,

выбрав наиболее вероятную (n + 1)-грамму для данной входной n-граммы. Для оценки прогнозирующей мощности модели обычно принято вычислять ее неопределенность, которую можно вычислить в терминах `entropy`, как 2 в степени `entropy`:

```
    def perplexity(self, text):
        """
        Для заданного списка строк, разделенных запятыми, вычисляет
        неопределенность текста.
        """
        return pow(2.0, self.entropy(text))
```

Неопределенность — это нормализованная оценка вероятности; чем выше условная вероятность последовательности лексем, тем ниже неопределенность. Как правило, модели более высокого порядка демонстрируют меньшую неопределенность:

```
trigram_model = BaseNgramModel(count_ngrams(3, vocab, sents))
fivegram_model = BaseNgramModel(count_ngrams(5, vocab, sents))

print(trigram_model.perplexity(sents[0]))
print(fivegram_model.perplexity(sents[0]))
```

Неизвестные слова: возвраты и сглаживание

Естественные языки очень гибки, поэтому было бы наивно ожидать, что даже весьма большие корпусы будут содержать все возможные n-граммы. А значит, наши модели тоже должны быть достаточно гибкими, чтобы обрабатывать n-граммы, которые прежде никогда не видели (например, «the President of California» (президент Калифорнии), «the United States of Canada» (Соединенные Штаты Канады)). В символических моделях эту проблему можно решить с использованием *возвратов* — если вероятность данной n-граммы неизвестна, модель пытается определить вероятность (n – 1)-граммы («the President of», «the United States of») и так далее, пока не дойдет до одиночной лексемы, или униграммы. Как правило, возвраты к меньшим n-граммам выполняются, пока не будет получено достаточно информации для оценки вероятности.

Поскольку `BaseNgramModel` использует оценку максимальной вероятности, некоторые (возможно, многие) n-граммы будут иметь нулевую вероятность появления, в результате `score()` будет возвращать ноль, и оценка неопределенности будет получаться равной `+` или `-` бесконечности. Для решения проблемы нулевых вероятностей n-грамм можно организовать *сглаживание*. Сглаживание заключается в передаче некоторой доли вероятности часто встречающихся n-грамм прежде не встречавшимся n-граммам. Простейшим видом сглаживания

является сглаживание Лапласа («add-one smoothing»), когда новому термину присваивается частота 1 и производится пересчет вероятностей, но существует много других видов сглаживания, таких как обобщенное сглаживание Лапласа, или «add-k smoothing».

Оба вида сглаживания легко реализовать, определив класс `AddKNgramModel`, наследующий наш класс `BaseNgramModel` и переопределяющий метод `score`, который будет добавлять сглаживающее значение k к счетчику n-граммы и делить на значение счетчика $(n-1)$-граммы, нормализованное счетчиком униграммы, умноженным на k:

```
class AddKNgramModel(BaseNgramModel):
    """
    Реализует обобщенное сглаживание Лапласа (add-k).
    """
    def __init__(self, k, *args):
        """
        Принимает значение k, на которое должны увеличиваться
        счетчики слов в процессе оценки.
        """
        super(AddKNgramModel, self).__init__(*args)

        self.k = k
        self.k_norm = len(self.ngram_counter.vocabulary) * k

    def score(self, word, context):
        """
        В обобщенном сглаживании Лапласа оценка нормализуется
        значением k.
        """
        context = self.check_context(context)
        context_freqdist = self.ngrams[context]
        word_count = context_freqdist[word]
        context_count = context_freqdist.N()
        return (word_count + self.k) / \
            (context_count + self.k_norm)
```

Теперь можно определить класс `LaplaceNgramModel`, который инициализирует `AddKNgramModel` значением `k = 1`:

```
class LaplaceNgramModel(AddKNgramModel):
    """
    Реализует сглаживание Лапласа .
    Сглаживание Лапласа является базовым случаем обобщенного
    сглаживания add-k со значением k = 1.
    """
    def __init__(self, *args):
        super(LaplaceNgramModel, self).__init__(1, *args)
```

Модуль `probability` библиотеки NLTK поддерживает несколько способов вычисления вероятности, включая разновидности метода максимального подобия и обобщенного сглаживания Лапласа (add-k), а также:

- `UniformProbDist` присваивает равные вероятности всем образцам, имеющимся в заданном наборе, и нулевую вероятность всем остальным.
- `LidstoneProbDist` сглаживает вероятности образцов с использованием вещественного числа `gamma` в диапазоне от 0 до 1.
- `KneserNeyProbDist` реализует разновидность возвратов, подсчитывая вероятность n-граммы по числу $(n - 1)$-грамм, встреченных при обучении.

Сглаживание Кнесера — Нея (Kneser—Ney) использует частоту униграммы не саму по себе, а в отношении к n-граммам, которые она завершает. Одни слова появляются в самых разных контекстах, но другие появляются часто и только в определенных контекстах; мы должны интерпретировать их по-разному.

Мы можем создать обертку для реализации сглаживания Кнесера — Нея из NLTK, определив класс `KneserNeyModel`, наследующий `BaseNgramModel` и переопределяющий метод `score`, в котором используется `nltk.KneserNeyProbDist`. Обратите внимание на то, что NLTK-реализация, `nltk.KneserNeyProbDist`, работает только с триграммами:

```
class KneserNeyModel(BaseNgramModel):
    """
    Реализует сглаживание Кнесера — Нея
    """
    def __init__(self, *args):
        super(KneserNeyModel, self).__init__(*args)
        self.model = nltk.KneserNeyProbDist(self.ngrams)

    def score(self, word, context):
        """
        Использует KneserNeyProbDist из NLTK для получения оценки
        """
        trigram = tuple((context[0], context[1], word))
        return self.model.prob(trigram)
```

Генерация языка

Вместе с возможностью присваивать вероятности n-граммам мы получили механизм примерной генерации языка. Чтобы применить наш класс `KneserNeyModel` для построения генератора следующего слова, добавим два дополнительных метода, `samples` и `prob`, чтобы получить доступ к списку всех триграмм с ненулевыми вероятностями и к вероятности каждого образца.

```
    def samples(self):
        return self.model.samples()

    def prob(self, sample):
        return self.model.prob(sample)
```

Теперь можно написать простую функцию, которая принимает текст, извлекает вероятности всех возможных триграмм продолжения для двух последних слов и добавляет наиболее вероятное следующее слово. Если передано меньше двух слов, запросим дополнительный ввод. Если `KneserNeyModel` присвоит нулевую вероятность, попробуем изменить тему:

```
corpus = PickledCorpusReader('../corpus')
tokens = [''.join(word) for word in corpus.words()]
vocab = Counter(tokens)
sents = list([word[0] for word in sent] for sent in corpus.sents())

counter = count_ngrams(3, vocab, sents)
knm = KneserNeyModel(counter)

def complete(input_text):
    tokenized = nltk.word_tokenize(input_text)
    if len(tokenized) < 2:
        response = "Say more."
    else:
        completions = {}
        for sample in knm.samples():
            if (sample[0], sample[1]) == (tokenized[-2], tokenized[-1]):
                completions[sample[2]] = knm.prob(sample)
        if len(completions) == 0:
            response = "Can we talk about something else?"
        else:
            best = max(
                completions.keys(), key = (lambda key: completions[key])
            )
            tokenized + = [best]
            response = " ".join(tokenized)

    return response

print(complete("The President of the United"))
print(complete("This election year will"))

The President of the United States
This election year will suddenly
```

Довольно легко написать приложение, решающее простые задачи вероятностной генерации языка, но, как мы видели, чтобы создать что-то более сложное (например, приложение, генерирующее целые предложения), в модели не-

обходимо закодировать больше информации о языке. Этого можно добиться с применением *n*-грамм высокого порядка и больших предметно-ориентированных корпусов.

Но как определить, насколько хороша наша модель? Модели на основе *n*-грамм можно оценить двумя способами. Во-первых, качество модели на контрольных данных можно оценить с использованием вероятностной меры, такой как неопределенность или энтропия. В этом случае лучшей можно считать модель, максимизирующую энтропию или минимизирующую неопределенность для контрольного набора. Качество символических моделей принято описывать их максимальным контекстом (размером *n*-грамм) и применяемым механизмом сглаживания. На момент написания этих строк лучшими считались символические модели с 5-граммами и сглаживанием Кнесера — Нея[1].

С другой стороны, более надежную оценку модели можно получить, встроив ее в приложение и проанализировав отзывы пользователей!

В заключение

В этой главе мы исследовали несколько новых методов конструирования контекстных признаков для улучшения простых моделей типа «мешок слов». Структура текста имеет большое значение для его понимания. Использование контекста через извлечение ключевых фраз или значимых словосочетаний на основе грамматик позволяет существенно улучшить качество модели.

В этой главе мы использовали *символический* подход к анализу текста, то есть смоделировали язык как набор дискретных фрагментов с вероятностями их появления. Дополнив эту модель механизмами *возврата* и *сглаживания* для обработки неизвестных слов, мы получили модель языка на основе *n*-грамм, способную генерировать текст. Такой подход к моделированию языка может показаться излишне академическим, однако возможность статистической оценки отношений в тексте нашла широкое применение в самых разных коммерческих приложениях, включая современные системы поиска в сети, чат-боты и машинный перевод.

Второй подход, не обсуждавшийся в этой главе, но заслуживающий упоминания: нейронная, или *ассоциативная* модель языка, которая использует нейронные сети в качестве ассоциативных единиц с непредсказуемым поведением. В настоящее время нейронные сети стали широкодоступными благо-

[1] Frankie James, *Modified Kneser–Ney smoothing of n-gram models* (2000), http://bit.ly/2JIc5pN

даря появлению таких инструментов, как word2vec, SpaCy и TensorFlow, но их обучение может требовать значительных вычислительных ресурсов, а отладка и интерпретация результатов сопряжены с большими сложностями. По этой причине многие приложения используют более понятные символические модели, которые часто легко модифицировать с использованием простых эвристик, как будет показано в главе 10. В главе 12 мы используем ассоциативный подход для создания модели классификации языка и обсудим случаи, когда такие модели могут оказаться более предпочтительными.

Но прежде чем перейти к этим более продвинутым моделям, рассмотрим в главе 8 приемы визуализации текста и диагностики визуальных моделей, использующие статистические вычисления для визуализации происходящего в моделях.

8 Визуализация текста

Машинное обучение часто ассоциируется с автоматизацией принятия решений, но на практике процесс создания прогностической модели обычно требует участия человека. Компьютеры прекрасно справляются с точными математическими вычислениями, а люди способны быстро выявлять закономерности. Объединить эти два важных аспекта поможет механизм *визуализации*, точно и аккуратно отображающий данные на экране компьютера в некотором визуальном представлении, которое позволит человеку быстро понять их суть.

В главах 5 и 6 мы рассмотрели несколько практических примеров прикладных моделей машинного обучения. В ходе реализации этих примеров мы заметили, что интеграция машинного обучения часто оказывается намного сложнее простого обучения модели. Во-первых, первая модель редко бывает оптимальной, что означает необходимость организации итеративного процесса обучения, оценки и настройки моделей.

Кроме того, оценка, интерпретация и представление результатов прикладного анализа текста осуществляются с большими сложностями, чем в случае с числовыми данными. Какой способ поиска наиболее информативных признаков лучше — когда признаками являются слова, фрагменты слов или фразы? Как определить, какая модель классификации лучше подходит для нашего корпуса? Как узнать, какое значение k дает наибольший эффект в кластеризации методом k-средних?

Именно эти вопросы вместе с необходимостью итеративного движения к достижению наиболее оптимального решения привели нас к *тройке выбора модели*, как описано в разделе «Тройка выбора модели», в главе 1. В этой главе мы увидим, как *визуальная диагностика* дополняет этот процесс механизмами визуализации для выявления проблем или для более простой оценки качества

моделей относительно друг друга. Мы исследуем набор инструментов визуализации, которые могут пригодиться для *управления* моделями машинного обучения, помогая эффективнее вмешиваться в процесс моделирования благодаря нашим врожденным способностям выявлять закономерности в изображениях.

Сначала мы реализуем различные методы анализа и конструирования признаков для текстовых данных, от графиков временных рядов *n*-грамм до стохастического алгоритма вложения соседей (Stochastic Neighbor Embedding, SNE). Затем перейдем к визуальному анализу текстовых моделей и инструментам выявления ошибок в моделях, таким как матрицы несоответствий и графики ошибок прогнозирования классов. Наконец, мы рассмотрим некоторые визуальные методы, которые можно использовать для оптимизации гиперпараметров с целью улучшения качества моделей.

Визуализация пространства признаков

Конструирование признаков, оценка и настройка моделей в традиционном процессе числового прогнозирования выполняются относительно просто. Выявить наиболее информативные признаки в пространстве с небольшим числом измерений можно путем обучения модели и вычисления доли наблюдаемой дисперсии, объясняемой каждым признаком. Полученные результаты можно визуализировать с использованием столбчатых диаграмм или двумерных тепловых карт попарных корреляций.

В случае с текстовыми данными визуализировать пространство признаков намного сложнее. Отчасти это связано с тем, что визуализация многомерных данных по сути сложнее, а также потому, что визуализация текстовых данных в Python требует дополнительных вычислений в сравнении с визуализацией чисто числовых данных. В этом разделе мы рассмотрим приемы визуализации с применением Matplotlib, которые могут вам пригодиться при *анализе* и *конструировании признаков*.

Визуальный анализ признаков

По сути, анализ признаков сводится к знакомству с особенностями данных. Анализируя числовые данные с небольшим числом измерений, можно использовать блочные и «скрипичные» (violin) диаграммы, гистограммы, диаграммы рассеяния, радиальные диаграммы и диаграммы параллельных координат. К сожалению, высокая размерность текстовых данных делает эти приемы не только неудобными, но и не всегда актуальными.

В контексте текстовых данных анализ признаков сводится к изучению содержимого корпуса. Например, насколько велики документы и словарь? Какие закономерности или комбинации *n*-грамм позволяют получать наибольший объем информации о документах? И вообще, насколько грамматически правилен наш текст? Является ли он сугубо техническим, состоящим из множества узкоспециальных составных именных словосочетаний? Является ли он переводом с другого языка? Насколько правильно используется пунктуация?

Все эти вопросы помогают приступить к формулированию обоснованных гипотез, позволяющих эффективно экспериментировать и создавать прототипы. В этом разделе мы познакомимся с несколькими специализированными приемами анализа признаков, которые особенно хорошо подходят для текстовых данных: временные ряды *n*-грамм, сетевой анализ и проекционные диаграммы.

Временные ряды *n*-грамм

В главе 7 мы исследовали приемы извлечения признаков на основе грамматик с целью выявления значимых закономерностей следования лексем в большом количестве документов. Этим этапом рабочего процесса проще управлять, имея возможность визуально исследовать частоту комбинаций лексем как функцию от времени. В этом разделе рассказывается, как реализовать визуализацию *n*-грамм для поддержки анализа признаков такого вида.

В объектах чтения корпуса, сконструированных нами в главах 3 и 4, отсутствует метод `dates`, однако этот прием визуализации требует некоторого механизма, который позволял бы получить время создания документа по его `fileid`.

Допустим, что корпус данных сформирован как словарь, ключами в котором являются лексемы, а значениями — кортежи `(счетчик лексемы, время создания документа)`. Допустим также, что у нас имеется список `terms` строк, разделенных запятыми, которые соответствуют *n*-граммам для построения временных рядов.

Построение графика зависимости частоты *n*-грамм от времени начнем с инициализации фигуры Matplotlib и осей, указав ширину и высоту в дюймах. Для каждого термина в списке построим график, в котором ось *Y* соответствует частоте *n*-граммы, а ось *X* — времени создания документа, в котором встречается термин. Добавим также заголовок графика, легенду цветов и метки на осях *Y* и *X*. Также можно задать диапазон дат для поддержки масштабирования и фильтрации графика:

```
fig, ax = plt.subplots(figsize = (9,6))

for term in terms:
    data[term].plot(ax = ax)

ax.set_title("Token Frequency over Time")
ax.set_ylabel("word count")
ax.set_xlabel("publication date")
ax.set_xlim(("2016-02-29","2016-05-25"))
ax.legend()
plt.show()
```

Получившийся график, пример которого показан на рис. 8.1, показывает частоту упоминаний политических кандидатов (здесь они представлены униграммами) в новостях перед выборами.

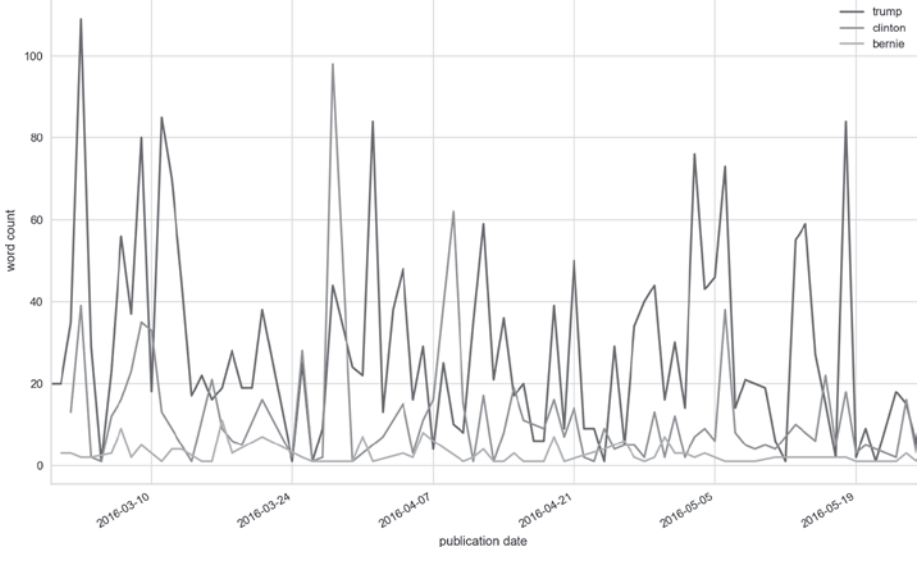


Рис. 8.1. График зависимости частоты *n*-грамм от времени

Как видите, график зависимости частоты от времени может оказаться полезным инструментом изучения и сравнения числа вхождений *n*-грамм в корпус с течением времени.

Сетевые диаграммы

Сетевые диаграммы пользуются большой популярностью в наши дни, о чем свидетельствует их преобладание в недавнем обзоре методов визуализации

текстовых данных[1]. Это объясняется тем, что сетевые диаграммы позволяют отобразить сложные взаимоотношения, которые иначе можно выразить только на естественном языке. Они особенно широко используются для анализа социальных сетей. Такие диаграммы можно использовать для иллюстрации связей между сущностями, документами и даже идеями в корпусе, которые мы подробно рассмотрим в главе 9. В этом разделе мы создадим диаграмму, представляющую книгу *The Wizard of Oz* («Волшебник страны Оз») Фрэнка Баума в виде социальной сети. Персонажи из книги будут играть роль узлов, а их отношения будут изображаться ребрами, длина которых тем меньше, чем теснее связь.

В этом разделе мы построим граф взаимодействия сил (force-directed graph), смоделированный в виде сети совхождения персонажей *Les Misérables* («Отверженные») Майком Бостоком (Mike Bostock)[2]. Такие графы проще отображать с помощью D3 и других фреймворков на JavaScript, но мы покажем, как это делается на Python с применением Matplotlib и NetworkX.

Для построения графа используем обработанную версию из библиотеки проекта Гутенберга, хранящуюся в файле JSON, который содержит словарь с двумя элементами: первый — список имен персонажей, отсортированный в порядке убывания их частот в тексте, и второй — словарь персонажей с парами ключ/значение `{название главы: текст главы}`. Для простоты из глав удалены пустые строки (то есть каждая строка представляет отдельный абзац) и все двойные кавычки в тексте (например, в диалогах) заменены одиночными кавычками.

Так как персонажи из книги играют роль узлов в графе, нам требуется установить связи между узлами. Для этого напишем функцию `cooccurrence`, которая сканирует главы и для каждой пары персонажей проверяет, как часто они появляются вместе. Инициализируем словарь, ключами в котором являются все возможные пары. Затем для каждой главы используем метод `sent_tokenize` из NLTK, чтобы выделить предложения в тексте, и для каждого предложения, содержащего имена обоих персонажей, будем увеличивать соответствующее значение в словаре на единицу:

```
import itertools
from nltk import sent_tokenize

def cooccurrence(text, cast):
    """
```

[1] The ISOVIS Group. *Text Visualization Browser: A Visual Survey of Text Visualization Techniques* (2015), *http://textvis.lnu.se/*

[2] Mike Bostock, *Force-Directed Graph* (2018), *http://bit.ly/2GNRKNU*

```
    Принимает на входе словарь text с главами {название: текст}
    и cast — список имен персонажей, разделенных запятыми.
    Возвращает словарь счетчиков совхождений для всех возможных
    пар имен.
    """
    possible_pairs = list(itertools.combinations(cast, 2))
    cooccurring = dict.fromkeys(possible_pairs, 0)
    for title, chapter in text['chapters'].items():
        for sent in sent_tokenize(chapter):
            for pair in possible_pairs:
                if pair[0] in sent and pair[1] in sent:
                    cooccurring[pair] + = 1
    return cooccurring
```

Далее откроем файл JSON, загрузим текст, извлечем список персонажей и инициализируем граф NetworkX. Для каждой пары, сгенерированной функцией `cooccurrence`, имеющей ненулевое значение, добавим ребро, хранящее счетчик совхождений как свойство.

Затем выполним извлечение `ego_graph`, чтобы поместить имя Dorothy в центр. Используем `spring_layout`, чтобы добавить узлы с силой отталкивания, обратно пропорциональной весам их ребер, указав желаемое расстояние между узлами через параметр `k` (чтобы избежать образования «комков шерсти») и количество итераций уравновешивания сил отталкивания через параметр `iterations`. Наконец, используем метод `draw` из библиотеки NetworkX, чтобы сгенерировать фигуру Matplotlib с желаемыми цветами и размерами узлов и ребер, и размером шрифта для подписей (с именами персонажей), обеспечивающим достаточную читаемость:

```
import json
import codecs
import networkx as nx
import matplotlib.pyplot as plt

with codecs.open('oz.json', 'r', 'utf-8-sig') as data:
    text = json.load(data)
    cast = text['cast']

    G = nx.Graph()
    G.name = "The Social Network of Oz"

    pairs = cooccurrence(text, cast)
    for pair, wgt in pairs.items():
        if wgt>0:
            G.add_edge(pair[0], pair[1], weight = wgt)

    # Поместить Dorothy в центр
    D = nx.ego_graph(G, "Dorothy")
    edges, weights = zip(*nx.get_edge_attributes(D, "weight").items())
```

```
    # Добавить узлы, поместить в узлы, менее тесно связанные с Dorothy
    pos = nx.spring_layout(D, k = 5, iterations = 40)
    nx.draw(D, pos, node_color = "gold", node_size = 50, edgelist = edges,
            width = .5, edge_color = "orange", with_labels = True, font_
size = 12)
    plt.show()
```

Получившийся граф, изображенный на рис. 8.2, наглядно иллюстрирует связи Dorothy (Дороти) в книге; ближе всего к ней расположены узлы, представляющие ближайших друзей — ее собаку Toto (Тотошку), Scarecrow (Страшилу) и Tin Woodman (Железного Дровосека), а дальше — персонажи, с которыми Dorothy связана менее тесно. Социальный граф также показывает близость Dorothy к Oz (волшебнику Оз) и Wicked Witch (Злой ведьме), с которыми у нее тоже сложились значимые, хотя и более сложные взаимоотношения.

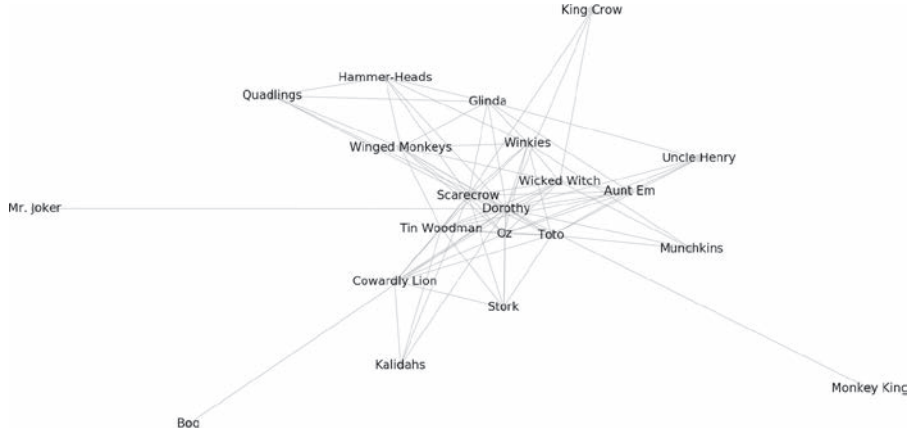


Рис. 8.2. Граф взаимодействия сил для книги «Wizard of Oz»

Создание графов и их свойства, а также методы `add_edge`, `ego_graph` и `draw` из библиотеки NetworkX подробно будут обсуждаться в главе 9.

Диаграммы совхождений

Построение диаграммы совхождений — еще один прием, помогающий быстро разобраться в связях между сущностями или другими *n*-граммами в терминах частоты их совместного появления. В этом разделе мы используем библиотеку Matplotlib для создания диаграммы совхождения персонажей в книге *The Wizard of Oz*.

Сначала напишем функцию `matrix`, которая принимает текст книги и список персонажей. Инициализируем многомерный массив, который будет играть роль списка, содержащего для каждого персонажа список счетчиков совхождений со всеми другими персонажами:

```
from nltk import sent_tokenize

def matrix(text, cast):
    mtx = []
    for first in cast:
        row = []
        for second in cast:
            count = 0
            for title, chapter in text['chapters'].items():
                for sent in sent_tokenize(chapter):
                    if first in sent and second in sent:
                        count + = 1
            row.append(count)
        mtx.append(row)
    return mtx
```

Теперь можно построить диаграмму на основе нашей матрицы. Для наглядности отобразим две диаграммы с помощью D3; в одной персонажи будут отсортированы в алфавитном порядке по именам, а в другой — по общей частоте появления в тексте. Инициализируем фигуру и оси, добавим название, увеличим отступы между диаграммами, чтобы освободить достаточно места для вывода имен персонажей, и увеличим размеры делений на осях *X* и *Y*, соответствующих персонажам.

Затем определим диаграмму, которую будем изменять, сославшись на ее индекс (`121`) — номер строки (`1`), номер столбца (`2`) и номер диаграммы (`1`). Определим параметры делений на осях *X* и *Y* и подписи для делений с именами персонажей, уменьшим шрифт подписей и угол поворота подписей, равный 90°, чтобы их было проще читать. Укажем, что деления для оси *X* должны отображаться сверху, и добавим подписи к осям первой диаграммы. Наконец, вызовем метод `imshow`, чтобы получить тепловую карту, передав параметр `interpolation`, желто-оранжево-коричневую цветовую палитру и указав необходимость логарифмической нормализации частот совхождений, чтобы редкие совхождения не отображались слишком яркими цветами:

```
...

    # Сначала создадим матрицы с сортировкой
    # По частоте
    mtx = matrix(text,cast)

    # Теперь создадим диаграммы
    fig, ax = plt.subplots()
```

```
    fig.suptitle('Character Co-occurrence in the Wizard of Oz', fontsize = 12)
    fig.subplots_adjust(wspace = .75)

    n = len(cast)
    x_tick_marks = np.arange(n)
    y_tick_marks = np.arange(n)

    ax1 = plt.subplot(121)
    ax1.set_xticks(x_tick_marks)
    ax1.set_yticks(y_tick_marks)
    ax1.set_xticklabels(cast, fontsize = 8, rotation = 90)
    ax1.set_yticklabels(cast, fontsize = 8)
    ax1.xaxis.tick_top()
    ax1.set_xlabel("By Frequency")
    plt.imshow(mtx,
               norm = matplotlib.colors.LogNorm(),
               interpolation = 'nearest',
               cmap = 'YlOrBr')
```

Чтобы создать диаграмму совхождений персонажей с сортировкой по именам, сначала создадим сортированный список персонажей и определим вторую диаграмму, с которой будем работать, (`122`). Добавим элементы осей, как в первом случае:

```
...
    # И по алфавиту
    alpha_cast = sorted(cast)
    alpha_mtx = matrix(text,alpha_cast)

    ax2 = plt.subplot(122)
    ax2.set_xticks(x_tick_marks)
    ax2.set_yticks(y_tick_marks)
    ax2.set_xticklabels(alpha_cast, fontsize = 8, rotation = 90)
    ax2.set_yticklabels(alpha_cast, fontsize = 8)
    ax2.xaxis.tick_top()
    ax2.set_xlabel("Alphabetically")
    plt.imshow(alpha_mtx,
               norm = matplotlib.colors.LogNorm(),
               interpolation = 'nearest',
               cmap = 'YlOrBr')
    plt.show()
```

Как и в случае с сетевым графом, данное представление является лишь аппроксимацией, потому что мы рассматриваем персонажей как последовательности символов, тогда как в действительности персонажи могут обозначаться множеством разных способов («Дороти» и «девочка из Канзаса», «Тотошка» и «ее маленькая собачка»). Тем не менее диаграмма, изображенная на рис. 8.3, довольно точно отражает тесноту связей между персонажами в книге.

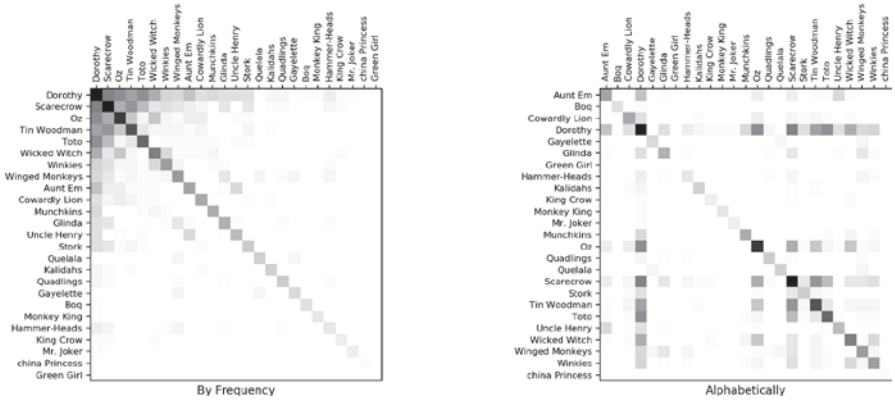


Рис. 8.3. Совместное появление персонажей в книге «The Wizard of Oz»

Рентген текста и диаграммы рассеяния

Сетевые диаграммы и диаграммы совхождений действительно помогают выявить связи между сущностями (или персонажами) в тексте, а также наиболее важные сущности, но они не отражают различий между их ролями в повествовании. Для этого нам нужно что-то вроде диаграмм рассеяния Джеффа Кларка (Jeff Clark)[1] и Тревора Стивена (Trevor Stephen)[2].

Диаграмма рассеяния помогает получить своеобразный «рентгеновский снимок» текста, располагая персонажей вдоль оси *Y* и упоминания о них в повествовании вдоль оси *X* так, что рядом с каждым персонажем на диаграмме отображается горизонтальная линия точек, соответствующих местам упоминания персонажа в тексте.

Создадим диаграмму рассеяния с помощью Matplotlib, использовав текст *The Wizard of Oz*. Сначала построим список `oz_words` всех слов в тексте в порядке их появления. Также будем запоминать длину и название каждой главы, чтобы потом использовать их для отображения начала и конца глав на оси *X*:

```
...

from nltk import word_tokenize, sent_tokenize

    # График упоминаний персонажей в главах
    oz_words = []
    headings = []
```

[1] Jeff Clark, *Novel Views: Les Miserables* (2013), *http://bit.ly/2GLzYKV*
[2] Trevor Stephens, *Catch-22: Visualized* (2014), *http://bit.ly/2GQKX6c*

```
    chap_lens = []
    for heading, chapter in text['chapters'].items():
        # Добавить название главы в список
        headings.append(heading)
        for sent in sent_tokenize(chapter):
            for word in word_tokenize(sent):
                # Добавить каждое слово
                oz_words.append(word)
        # Запомнить длину главы в словах
        chap_lens.append(len(oz_words))

    # Отметить начала глав
    chap_starts = [0] + chap_lens[:-1]
    # Объединить с названиями глав
    chap_marks = list(zip(chap_starts,headings))
```

Теперь отыщем в списке `oz_words` все места, где упоминаются персонажи, и добавим их в список точек для отображения на диаграмме. В нашем случае у некоторых персонажей имена из одного слова (например, «Dorothy» (Дороти), «Scarecrow» (Страшила), «Glinda» (Глинда)), а у других из двух («Cowardly Lion» (Трусливый Лев), «Monkey King» (Король обезьян)). Учитывая это, имена из одного слова будем просто сравнивать с очередным словом из текста, а имена из двух слов — со словом из текста, объединенным с предшествующим ему словом:

```
...
    cast.reverse()
    points = []
    # Добавлять точку, как только встретится имя персонажа
    for y in range(len(cast)):
        for x in range(len(oz_words)):
            # У некоторых персонажей имена из одного слова
            if len(cast[y].split()) == 1:
                if cast[y] == oz_words[x]:
                    points.append((x,y))
            # У некоторых из двух слов
            else:
                if cast[y] == ' '.join((oz_words[x-1], oz_words[x])):
                    points.append((x,y))
    if points:
        x, y = list(zip(*points))
    else:
        x = y = ()
```

Создадим фигуру и оси, сделаем ось X шире, чем по умолчанию, чтобы диаграмму было легче читать. Добавим также вертикальные линии, отмечающие начало каждой главы, и используем имена глав в роли меток, поместив их непосредственно под осью X с поворотом на 90° и уменьшив немного шрифт. Затем отобразим точки `x` и `y` и изменим `tick_params`, чтобы отключить отображение де-

лений и подписей по умолчанию внизу. После этого добавим метки вдоль оси *Y* для всех персонажей и подписи с их именами и, наконец, добавим заголовок:

...

```
# Создать диаграмму
fig, ax = plt.subplots(figsize = (12,6))
# Добавить вертикальные линии, отмечающие начала глав, с их названиями
# в подписях
for chap in chap_marks:
    plt.axvline(x = chap[0], linestyle = '-',
        color = 'gainsboro')
    plt.text(chap[0], -2, chap[1], size = 6, rotation = 90)
# Вывести точки упоминаний персонажей
plt.plot(x, y, "|", color = "darkorange", scalex = .1)
plt.tick_params(
    axis = 'x', which = 'both', bottom = 'off', labelbottom = 'off'
)
plt.yticks(list(range(len(cast))), cast, size = 8)
plt.ylim(-1, len(cast))
plt.title("Character Mentions in the Wizard of Oz")
plt.show()
```

Получившаяся диаграмма, изображенная на рис. 8.4, дает общее представление о структуре текста. Она не только позволяет увидеть, где упоминаются персонажи (или, в более общем случае, *n*-граммы), но также ясно показывает, какие персонажи играют главную роль, и даже подсвечивает некоторые интересные области в тексте (например, где упоминается сразу несколько персонажей или где частота упоминания того или иного персонажа резко изменяется).

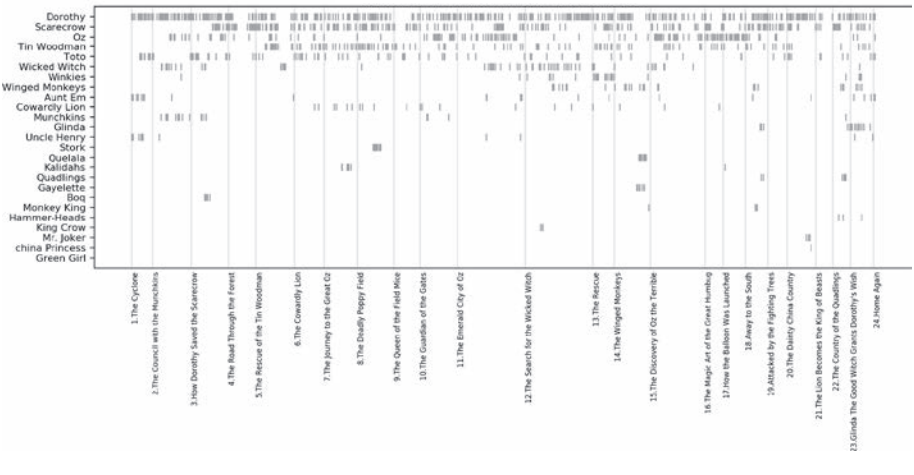


Рис. 8.4. Диаграмма рассеяния упоминаний персонажей в книге «The Wizard of Oz»

Управление конструированием признаков

Получив более полное представление об исходном содержимом корпуса, нужно спроектировать минимальный набор признаков, обладающих наибольшими прогностическими свойствами для использования в моделировании. Этот набор должен быть как можно меньше, потому что каждое новое измерение будет вносить дополнительный шум и затруднять моделирование пространства решений.

При работе с текстовыми данными особенно важно использовать творческий подход, чтобы максимально снизить размерность без значительной потери качества сигнала. В числе эффективных способов сжатия размерности исходных данных можно назвать метод главных компонент (Principal Component Analysis) и линейный дискриминантный анализ (и Doc2Vec в случае с текстовыми данными). Однако эти методы могут вызвать проблемы позже, если пользователям потребуется восстановить оригинальные признаки (например, конкретные термины и фразы, на основе которых было определено сходство двух документов).

В этом разделе мы рассмотрим несколько визуальных приемов управления конструированием признаков, которые, по нашему мнению, особенно хорошо подходят для текстовых данных: визуальная разметка частями речи и плотность распределения.

Разметка частями речи

Как мы узнали в главе 3, части речи (глаголы, существительные, предлоги, прилагательные и др.) указывают на роль слова в контексте предложения. Во многих языках одно и то же слово может играть разные роли, и нам нужна возможность различать их (например, слова «печь» и «жила» в разных контекстах могут быть глаголами или существительными). Разметка (или маркировка) частями речи позволяет закодировать информацию не только об определении слова, но также о его роли в данном контексте.

В главе 7 мы использовали теги частей речи вместе с грамматикой для извлечения ключевых фраз. Одна из проблем такого подхода к конструированию признаков заключается в сложности определения грамматики, которая лучше подойдет для поиска ключевых фраз. В общем случае стратегия состоит в использовании эвристик и проведении экспериментов, пока не будет получено регулярное выражение, хорошо справляющееся с извлечением высокоинформативных ключевых фраз. На самом деле, эта стратегия хорошо работает с грамматически правильными текстами. Но что, если анализируемый текст написан безграмотно или изобилует орфографическими и пунктуационными

ошибками? В таких случаях разметка частями речи может принести больше вреда, чем пользы.

Представьте, что текст не содержит значимых ключевых фраз, соответствующих шаблону «прилагательное/существительное». Например, во многих случаях существенная информация может передаваться не через прилагательные, а через глагольные или наречные словосочетания или имена собственные. В этом случае, даже если механизм разметки частями речи работает безукоризненно и наш парсер выглядит как-то так...

```
grammar = r'KT: {(<JJ>* <NN.*>+ <IN>)? <JJ>* <NN.*>+}'
chunker = nltk.chunk.regexp.RegexpParser(grammar)
```

...мы действительно можем пропустить важный сигнал в нашем корпусе!

Было бы полезно сначала визуально оценить распределение частей речи в тексте, и только потом переходить к нормализации, векторизации и моделированию (возможно, в качестве диагностического инструмента при выяснении причин неудовлетворительных результатов моделирования). Например, обнаружив, что значительная доля текста оказалась неклассифицированной (или классифицированной неправильно) имеющимся механизмом разметки частями речи, мы сможем реализовать свой механизм на основе регулярных выражений для обработки нашего конкретного корпуса. Также это может повлиять на выбор способа нормализации текста (например, если в тексте присутствует множество значимых вариаций слов с одним корнем, мы могли бы предпочесть заменить стемминг лемматизацией, несмотря на то что это увеличит время обработки).

Библиотека Yellowbrick предлагает возможность вывода текста разными цветами, обозначающими части речи. Объект `PosTagVisualizer` раскрашивает текст, позволяя пользователю визуально оценить пропорциональное соотношение существительных, глаголов и т. д. и использовать эту информацию при принятии решений, касающихся разметки частями речи, нормализации (например, при выборе между стеммингом и лемматизацией) и векторизации.

Метод `transform` преобразует исходный текст, подготавливая его к визуализации частей речи. Обратите внимание: он требует, чтобы документы были представлены последовательностями кортежей `(tag, token)`:

```
from nltk import pos_tag, word_tokenize
from yellowbrick.text.postag import PosTagVisualizer

pie = """
    In a small saucepan, combine sugar and eggs
    until well blended. Cook over low heat, stirring
```

```
    constantly, until mixture reaches 160° and coats
    the back of a metal spoon. Remove from the heat.
    Stir in chocolate and vanilla until smooth. Cool
    to lukewarm (90°), stirring occasionally. In a small
    bowl, cream butter until light and fluffy. Add cooled
    chocolate mixture; beat on high speed for 5 minutes
    or until light and fluffy. In another large bowl,
    beat cream until it begins to thicken. Add
    confectioners' sugar; beat until stiff peaks form.
    Fold into chocolate mixture. Pour into crust. Chill
    for at least 6 hours before serving. Garnish with
    whipped cream and chocolate curls if desired.
    """

tokens = word_tokenize(pie)
tagged = pos_tag(tokens)

visualizer = PosTagVisualizer()
visualizer.transform(tagged)

print(' '.join((visualizer.colorize(token, color)
                for color, token in visualizer.tagged)))
print('\n')
```

Если выполнить этот код в командной строке или в Jupyter Notebook, он произведет вывод, изображенный на рис. 8.5.

```
In a small saucepan , combine sugar and eggs until well blended . Cook over low heat , stirri
ng constantly , until mixture reaches 160° and coats the back of a metal spoon . Remove from
the heat . Stir in chocolate and vanilla until smooth . Cool to lukewarm ( 90° ) , stirring o
ccasionally . In a small bowl , cream butter until light and fluffy . Add cooled chocolate mi
xture ; beat on high speed for 5 minutes or until light and fluffy . In another large bowl ,
beat cream until it begins to thicken . Add confectioners ' sugar ; beat until stiff peaks fo
rm . Fold into chocolate mixture . Pour into crust . Chill for at least 6 hours before servin
g . Garnish with whipped cream and chocolate curls if desired .
```

Рис. 8.5. Текст рецепта после разметки частями речи

Как показано на рис. 8.5, текст из поваренной книги оказался достаточно хорошо размечен частями речи, и лишь в нескольких местах механизм разметки ошибся или не смог определить, какой тег поставить. Однако, как показывает следующий пример (рис. 8.6), базовый механизм разметки частями речи из библиотеки NLTK допускает слишком много ошибок в некоторых областях, например в детских стишках.

```
Baa , baa , black sheep , Have you any wool ? Yes , sir , yes , sir , Three bags full ; One f
or the master , And one for the dame , And one for the little boy Who lives down the lane .
```

Рис. 8.6. Детский стишок после разметки частями речи

Как видите, визуальную маркировку частей речи можно использовать как инструмент для оценки эффективности инструментов предварительной обработки (как описано в главе 3), а также для конструирования признаков и диагностики моделей.

Наиболее информативные признаки

Выявление наиболее информативных (то есть прогнозирующих) признаков в наборе данных — важная часть в тройке выбора модели. Тем не менее методы, хорошо знакомые нам по численному моделированию (например, L1- и L2-регуляризация, утилиты из Scikit-Learn, такие как `select_from_model`, и т. д.), часто оказываются малополезными, когда роль данных играет текст, а признаками являются лексемы или другие лингвистические сущности. После векторизации данных, как рассказывалось в главе 4, кодирование затрудняет извлечение информации с одновременным сохранением естественного повествования нетронутым.

Один из методов визуального исследования текста — определение плотности распределения. В контексте текстового корпуса плотность распределения помогает оценить преобладание некоторого элемента словаря или лексемы.

В следующих нескольких примерах мы с помощью библиотеки Yellowbrick проведем визуальное исследование подкорпуса «hobbies» (увлечения) из корпуса Baleen, который можно загрузить вместе с другими наборами данных Yellowbrick.

ЗАГРУЗКА НАБОРОВ ДАННЫХ YELLOWBRICK

Как загрузить наборы данных Yellowbrick:

Библиотека Yellowbrick включает несколько наборов данных, полученных из репозитория UCI Machine Learning Repository. Чтобы загрузить наборы данных, извлеките библиотеку Yellowbrick и запустите загрузку, как показано ниже:

```
$ git clone https://github.com/DistrictDataLabs/yellowbrick.git
$ cd yellowbrick/examples
$ python download.py
```

Обратите внимание на то, что в результате будет создан каталог *data* с подкаталогами, содержащими данные.

После загрузки используйте объект `sklearn.datasets.base.Bunch`, чтобы загрузить корпус в атрибуты `features` и `target` соответственно, подобно тому, как структурированы наборы данных в Scikit-Learn:

```
import os
import yellowbrick as yb
from sklearn.datasets.base import Bunch

## Путь к тестовым наборам данных
FIXTURES = os.path.join(os.getcwd(), "data")

## Механизмы загрузки корпуса
corpora = {
    "hobbies": os.path.join(FIXTURES, "hobbies")
}

def load_corpus(name):
    """
    Загружает и извлекает указанный корпус по имени name.
    """

    # Получить путь из наборов данных
    path = corpora[name]

    # Прочитать каталоги в каталоге как категории.
    categories = [
        cat for cat in os.listdir(path)
        if os.path.isdir(os.path.join(path, cat))
    ]

    files  = [] # список имен файлов относительно корневого каталога
    data   = [] # текст, прочитанный из файла
    target = [] # строка с названием категории

    # Загрузить данные из файлов в корпус
    for cat in categories:
        for name in os.listdir(os.path.join(path, cat)):
            files.append(os.path.join(path, cat, name))
            target.append(cat)

            with open(os.path.join(path, cat, name), 'r') as f:
                data.append(f.read())

    # Вернуть пакет данных для использования, как в примере с группами
    # новостей
    return Bunch(
        categories = categories,
        files = files,
        data = data,
        target = target,
    )

corpus = load_corpus('hobbies')
```

После загрузки корпуса «hobbies» можно с помощью Yellowbrick провести исследование словаря путем анализа плотности распределения. Библиотека NLTK также предлагает возможность построить диаграмму плотности распределения 50 наиболее часто встречающихся лексем, но здесь мы используем Yellowbrick, потому что будем ее использовать в следующих нескольких примерах. Обратите внимание на то, что ни метод `FreqDist` из NLTK, ни метод `FreqDistVisualizer` из Yellowbrick не выполняют нормализацию или векторизацию; оба предполагают, что текст уже векторизован.

Для начала создадим объект визуализации `FreqDistVisualizer` и затем вызовем его метод `fit()` с векторизованными документами и признаками (то есть слова из корпуса), который рассчитает плотность распределения. Он построит столбчатую диаграмму для наиболее часто встречающихся терминов (50 по умолчанию, но это число можно изменить с помощью параметра `N`), в которой термины располагаются вдоль оси *Y*, а их частоты — вдоль оси *X*. После этого можно сгенерировать окончательное визуальное представление вызовом метода `poof()`:

```
from yellowbrick.text.freqdist import FreqDistVisualizer
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer()
docs = vectorizer.fit_transform(corpus.data)
features = vectorizer.get_feature_names()

visualizer = FreqDistVisualizer(features = features)
visualizer.fit(docs)
visualizer.poof()
```

На рис. 8.7 показаны 50 самых часто встречающихся терминов в корпусе «hobbies». Однако, если взглянуть на список слов, расположенных вдоль оси *Y*, можно заметить, что большая часть из них не представляет никакого интереса (например, «the», «and», «to», «that», «of», «it»). То есть, несмотря на большую частоту, эти термины определенно не являются информативными признаками.

В главе 4 мы использовали удаление стоп-слов как способ уменьшения размерности и получения признаков, несущих значимую информацию. Давайте посмотрим, как повлияет удаление из корпуса распространенных английских слов на результаты визуализации, передав параметр `stop_words` конструктору объекта `CountVectorizer`:

```
vectorizer = CountVectorizer(stop_words = 'english')
docs       = vectorizer.fit_transform(corpus.data)
features   = vectorizer.get_feature_names()

visualizer = FreqDistVisualizer(features = features)
visualizer.fit(docs)
visualizer.poof()
```

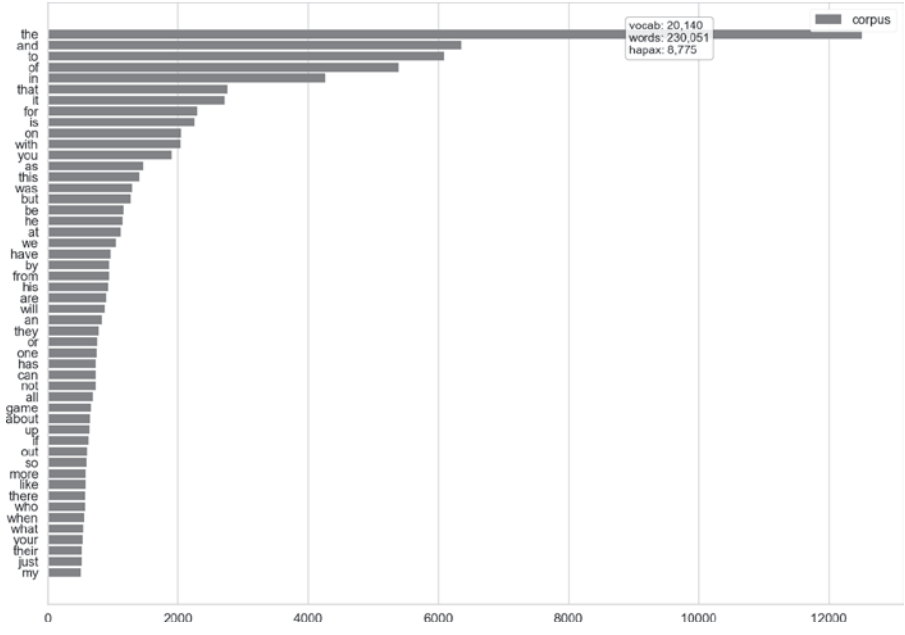


Рис. 8.7. Плотность распределения терминов в корпусе Baleen

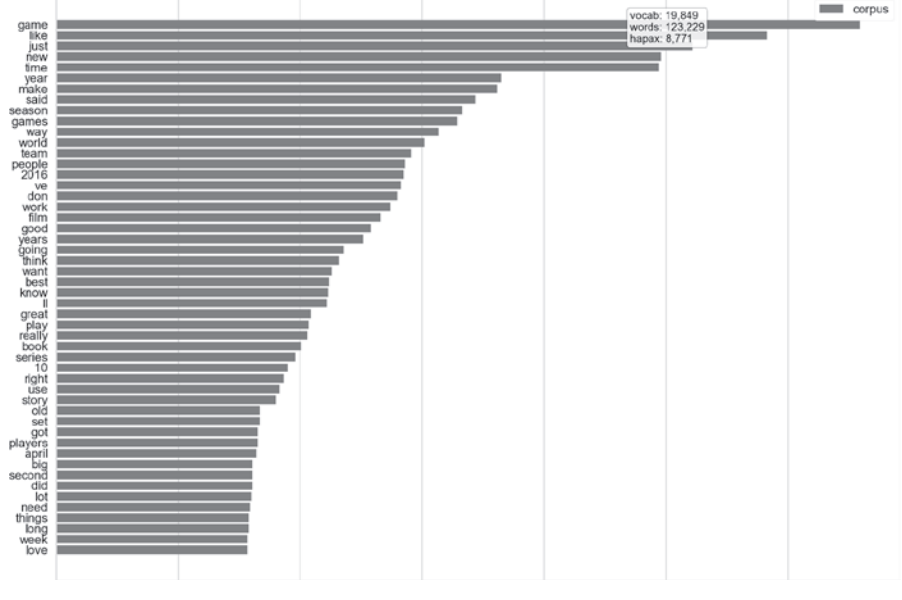


Рис. 8.8. Плотность распределения терминов в корпусе Baleen после удаления стоп-слов

Теперь, как можно заметить на рис. 8.8, после удаления стоп-слов мы получили более интересный результат (например, «game», «season», «team», «world», «film», «book», «week»). Но также стала очевидной размытость данных. В главе 1 мы узнали, что при создании приложений для анализа естественного языка следует опираться на специализированные, а не на обобщенные корпусы. Поэтому теперь проверим, является ли корпус «hobbies» достаточно специализированным для моделирования. Мы можем продолжить использовать диаграммы плотности распределения для поиска в корпусе более узких подтем и других закономерностей.

Корпус «hobbies», распространяемый вместе с библиотекой Yellowbrick, уже классифицирован (попробуйте прочитать `corpus['categories']`). На рис. 8.9 и 8.10 показаны диаграммы плотности распределения для двух категорий, «cooking» (кулинария) и «gaming» (игры) соответственно, после удаления стоп-слов.

Сравнив эти диаграммы, можно сразу заметить, насколько они различаются; в корпусе «cooking» чаще всего используются слова «pasta» (паста), «pan» (противень), «broccoli» (брокколи) и «pepper» (перец), а в корпусе «gaming» — слова «players» (игроки), «developers» (разработчики), «character» (персонаж) и «support» (поддержка).

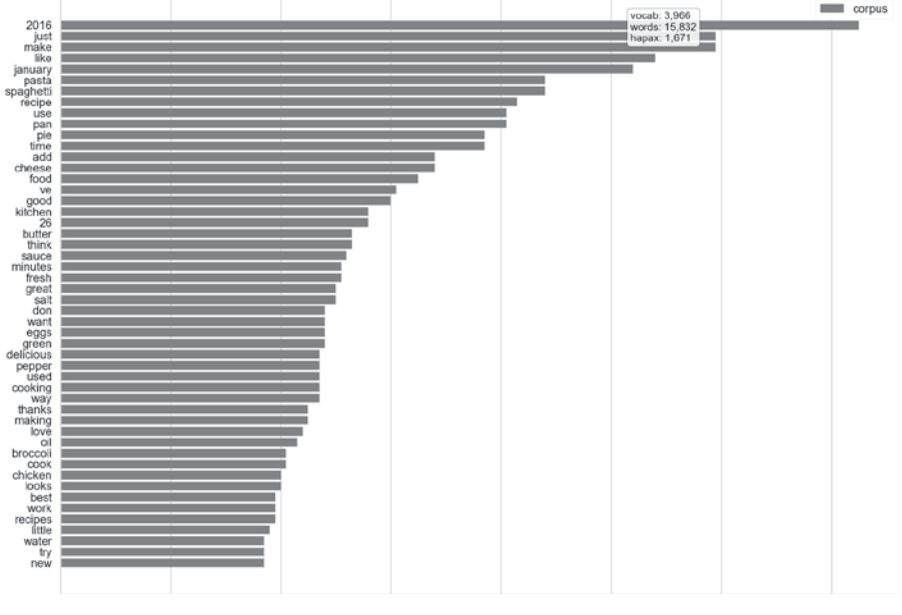


Рис. 8.9. Плотность распределения терминов в подкорпусе «cooking»

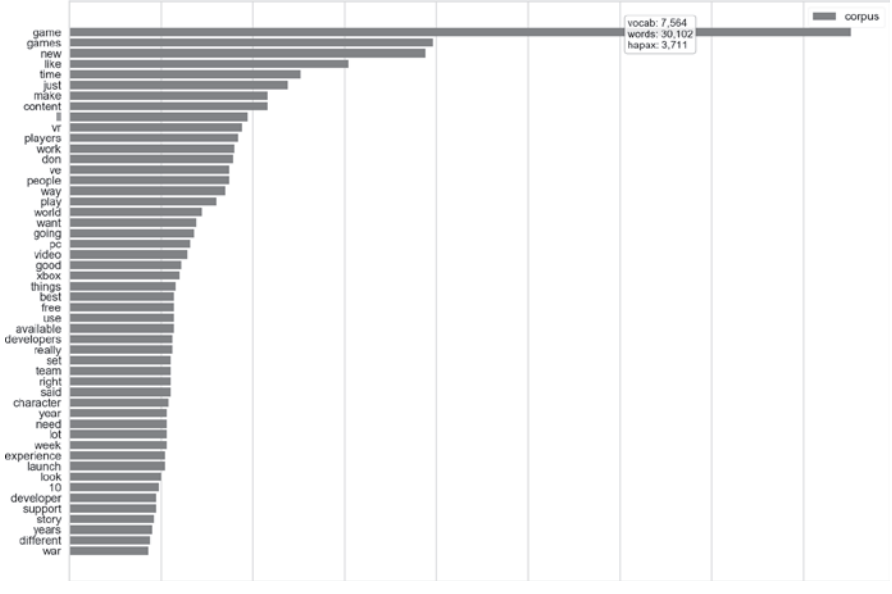


Рис. 8.10. Плотность распределения терминов в подкорпусе «gaming»

Диагностика моделей

После конструирования и анализа признаков наступает следующий этап в процессе тройки выбора модели — выбор модели. На практике мы выбираем и сравниваем множество моделей, потому что сложно предсказать наперед, какая модель окажется наиболее эффективной для нового корпуса. То есть наша следующая задача — определить, когда модели работают хорошо или плохо.

В традиционном контексте машинного обучения можно положиться на такие оценки эффективности модели, как среднеквадратичная ошибка и коэффициент смешанной корреляции в случае регрессии или точность, качество и F1 в случае классификации, и с их помощью определить, какие модели являются самыми оптимальными. Эти методы также можно дополнить средствами визуального анализа. Регрессия реже применяется для анализа текста, впрочем, в главе 12 мы рассмотрим пример предсказания оценки альбомов в виде вещественного числа исключительно на основе текстов отзывов о них. Как рассказывалось в главах 5 и 6, в машинном обучении на текстовых данных чаще применяются классификация и кластеризация, и в этом разделе мы познакомимся с несколькими приемами оценки моделей в этих контекстах.

Визуализация кластеров

Когда дело доходит до алгоритмов кластеризации, оценка модели оказывается не такой простой, как в случае машинного обучения с учителем, когда у нас есть преимущество знания правильных и неправильных ответов. На самом деле, в кластеризации нет количественной оценки — относительный успех модели обычно зависит от того, насколько благополучно она обнаруживает закономерности, различимые и значимые для человека. По этой причине применение методов визуализации для оценки приобретает особую важность.

Подобно тому, как с помощью диаграмм плотности распределения мы искали мелкие признаки разграничения и размытости, мы можем исследовать степень сходства документов по всем признакам. Для этой цели часто используется очень популярный метод *t*-распределенного стохастического вложения соседей (*t*-distributed Stochastic Neighbor Embedding, *t*-SNE).

Библиотека Scikit-Learn реализует метод разложения *t*-SNE в виде преобразователя `sklearn.manifold.TSNE`. Метод *t*-SNE способен сгруппировать схожие документы путем разложения векторных представлений документов с большим числом измерений в два измерения (с использованием распределений вероятностей из оригинальной и разложенной размерностей). Разложением на два или на три измерения можно построить диаграммы рассеяния.

К сожалению, метод *t*-SNE требует слишком много вычислительных ресурсов, поэтому на первых этапах чаще применяются более простые методы разложения, такие как SVD (сингулярное разложение) или PCA (метод главных компонент). В библиотеке Yellowbrick есть класс `TSNEVisualizer`, который создает внутренний конвейер преобразования, применяющий сначала разложение SVD с 50 компонентами по умолчанию, а затем выполняющий вложение *t*-SNE. Объект `TSNEVisualizer` принимает документы в векторном представлении, поэтому будем использовать `TfidfVectorizer` из Scikit-Learn перед передачей документов в метод `fit` класса `TSNEVisualizer`:

```
from yellowbrick.text import TSNEVisualizer
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf = TfidfVectorizer()
docs = tfidf.fit_transform(corpus.data)

tsne = TSNEVisualizer()
tsne.fit(docs)
tsne.poof()
```

В таких диаграммах нас интересует пространственное сходство между точками (документами) и любые другие заметные закономерности. На рис. 8.11 показана двумерная проекция векторизованного подкорпуса «hobbies» из корпуса Baleen, полученная методом *t*-SNE. Результатом является диаграмма рассеяния векторизованного корпуса, где каждая точка представляет документ или высказывание. Расстояние между двумя точками в визуальном пространстве вычисляется с использованием распределения вероятностей попарных сходств в пространстве более высокой размерности; то есть `TSNEVisualizer` показывает кластеры схожих документов в корпусе «hobbies» и отношения между группами документов.

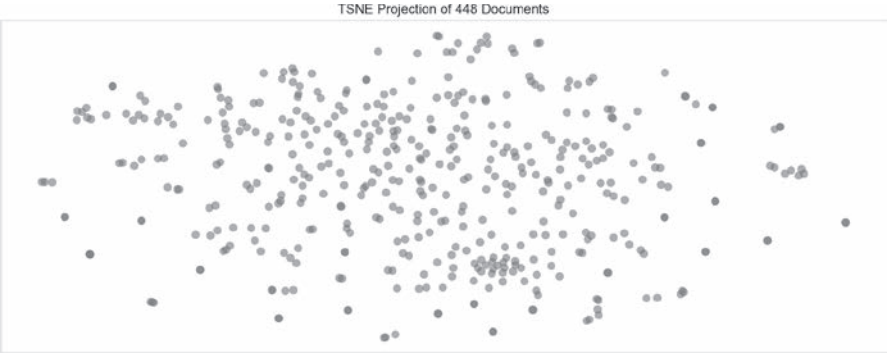


Рис. 8.11. Диаграмма *t*-распределенного стохастического вложения соседей для подкорпуса «hobbies» из корпуса Baleen

Как упоминалось выше, `TSNEVisualizer` принимает векторное представление текста, и в данном случае мы использовали преобразование TF–IDF, хотя мы могли бы использовать другие способы векторизации, описанные в главе 4, а затем сгенерировать диаграмму *t*-SNE и сравнить результаты. Для ускорения отображения перед стохастическим вложением соседей `TSNEVisualizer` выполняет разложение, используя по умолчанию `TruncatedSVD`; мы могли бы также поэкспериментировать с методом сжатия, таким как PCA, для чего достаточно передать параметр `decompose = "pca"` в конструктор `TSNEVisualizer()` на этапе инициализации.

Объект `TSNEVisualizer` можно также использовать для визуализации кластеров при использовании его совместно с алгоритмом кластеризации. С помощью этого приема можно оценить эффективность разных методов кластеризации. В следующем примере мы используем `sklearn.cluster.KMeans` с числом кластеров 5, а затем передадим полученный атрибут `cluster.labels_` в параметре `y` методу `fit()` объекта `TSNEVisualizer`:

```
# Использовать кластеры вместо имен классов.
from sklearn.cluster import KMeans

clusters = KMeans(n_clusters = 5)
clusters.fit(docs)

tsne = TSNEVisualizer()
tsne.fit(docs, ["c{}".format(c) for c in clusters.labels_])
tsne.poof()
```

Теперь все точки, принадлежащие одному кластеру, будут не только сгруппированы, но и раскрашены в соответствии с коэффициентом сходства (рис. 8.12). В данном случае можно поэкспериментировать с разными методами кластеризации или значениями *k*, но более подробно мы поговорим чуть позже, в разделе, посвященном настройке гиперпараметров.

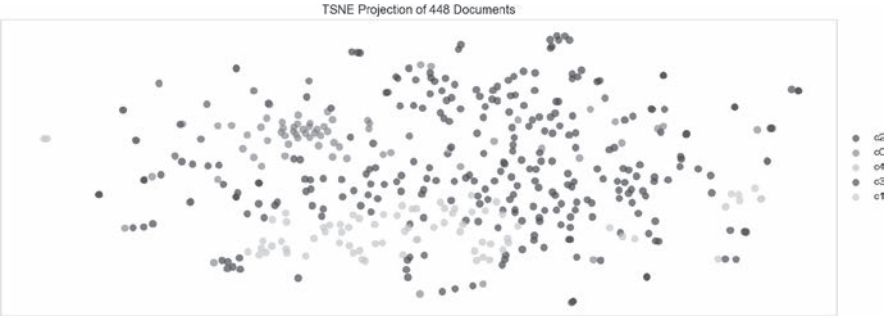


Рис. 8.12. Диаграмма *t*-SNE для подкорпуса «hobbies» из корпуса Baleen после кластеризации методом *k*-средних

Визуализация классов

При решении задач классификации можно просто подставлять целевое значение (хранящееся в `corpus.target`) в `TSNEVisualizer`, чтобы получить диаграмму, на которой цвет точек представляет метки категорий, которым соответствуют документы. Определив метки классов как аргумент в вызове `fit()` визуализатора *t*-SNE, можно раскрасить точки согласно их категориям:

```
from yellowbrick.text import TSNEVisualizer
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf = TfidfVectorizer()
docs = tfidf.fit_transform(corpus.data)
labels = corpus.target
```

```
tsne = TSNEVisualizer()
tsne.fit(docs, labels)
tsne.poof()
```

Как можно заметить на диаграмме рассеяния, изображенной на рис. 8.13, этот прием позволяет дополнить внедрение соседей дополнительной информацией о сходстве и упростить интерпретацию классов. Если бы нас интересовали лишь некоторые из категорий в нашем корпусе, мы могли бы просто передать их объекту `TSNEVisualizer` в параметре `classes` как список строк с названиями категорий (например, `TSNEVisualizer(classes = ['sports', 'cinema', 'gaming'])`).

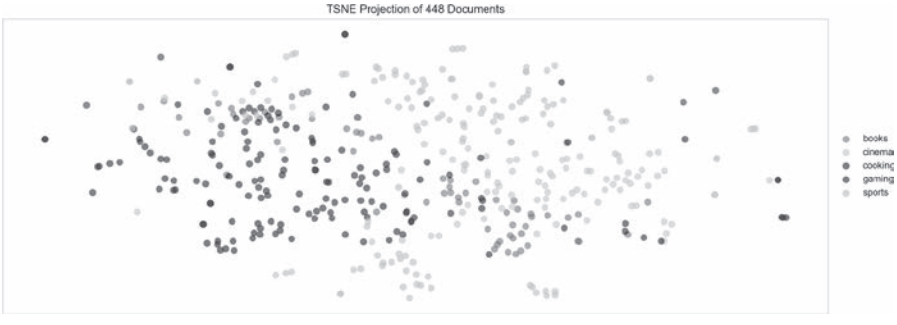


Рис. 8.13. Диаграмма *t*-распределенного стохастического вложения соседей для подкорпуса «hobbies» из корпуса Baleen с метками категорий

Поочередный обзор таких диаграмм поможет выявить наиболее плотные категории документов или, наоборот, размытые, которые могут усложнить процесс моделирования.

Диагностика ошибок классификации

В традиционном подходе к классификации обученные модели можно оптимизировать и затем определить их точность, полноту и величину оценки F1. Эти значения можно визуализировать, используя матрицы несоответствий, тепловые карты и кривые ROC-AUC.

В главе 5 мы рассмотрели прием перекрестной проверки для тестирования качества модели с использованием отдельных обучающих и контрольных выборок из корпуса. Там же мы реализовали метод тестирования множества разных моделей, чтобы получить возможность сравнить их с помощью отчетов классификации и матриц несоответствий. Мы использовали определенные метрики, чтобы выявить более удачные модели, но одни только метрики не

особенно помогают понять, почему конкретная модель (или разбиение на обучающие и контрольные выборки) действует именно так.

В этом разделе мы рассмотрим два наших излюбленных приема визуального анализа и сравнения качества классификаторов текста: тепловые карты и матрицы несоответствий.

Отчеты классификации с тепловыми картами

Отчет классификации — это текстовый перечень основных метрик, по которым можно судить об успешности классификатора: *точность* (precision) — способность не относить к данной категории экземпляр, фактически принадлежащий другой категории; *полнота* (recall) — способность обнаруживать все экземпляры, принадлежащие данной категории; и *оценка f1* — взвешенное среднее гармоническое точности и полноты. Получить отчет классификации можно с помощью метода `classification_report` из модуля `metrics` в библиотеке Scikit-Learn, однако, как нам кажется, версия в Yellowbrick, сочетающая вывод числовых оценок с цветной тепловой картой, упрощает интерпретацию и выявление проблем.

Чтобы воспользоваться механизмом создания тепловой карты из Yellowbrick, загрузим корпус, как описано во врезке «Загрузка наборов данных Yellowbrick» выше, выполним векторизацию документов методом TF–IDF и создадим обучающие и контрольные выборки. Затем создадим экземпляр `ClassificationReport`, передадим ему желаемый классификатор и имена классов, вызовем методы `fit` и `score`, которые, в свою очередь, вызовут внутренние механизмы обучения и оценки модели, находящиеся в библиотеке Scikit-Learn. Наконец, вызовем метод `poof` визуализатора, который добавит в диаграмму метки и цвет и затем вызовет метод `draw` из Matplotlib:

```
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import train_test_split
from yellowbrick.classifier import ClassificationReport

corpus = load_corpus('hobbies')
docs = TfidfVectorizer().fit_transform(corpus.data)
labels = corpus.target

X_train, X_test, y_train, y_test = train_test_split(
    docs.toarray(), labels, test_size = 0.2
)

visualizer = ClassificationReport(GaussianNB(), classes = corpus.categories)

visualizer.fit(X_train, y_train)
visualizer.score(X_test, y_test)
visualizer.poof()
```

Получившаяся тепловая карта показана на рис. 8.14. Она отражает точность, полноту и оценку F1 обученной модели, где более темные зоны соответствуют областям модели с наибольшим качеством. В этом примере мы видим, что гауссова модель успешно классифицирует большинство категорий, но имеет высокую ошибку второго рода (нераспознавание) для категории «books» (книги).

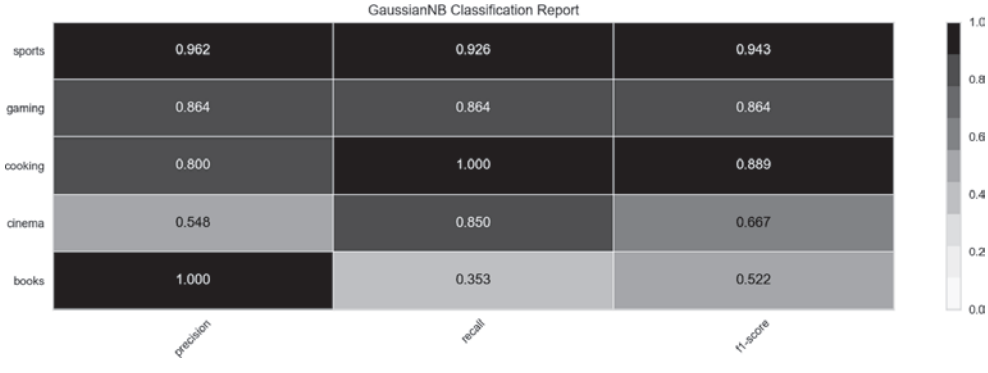


Рис. 8.14. Тепловая карта наивного байесовского гауссова классификатора на корпусе «hobbies»

Мы можем сравнить гауссову модель с другой моделью, просто импортировав другой классификатор из Scikit-Learn и передав его конструктору `ClassificationReport`. Для сравнения на рис. 8.15 изображена тепловая карта классификатора `SGDClassifier`, которая говорит, что он хуже справляется с классификацией корпуса «hobbies» и имеет высокую ошибку первого рода (ложное распознавание) для категории «gaming» и ошибку второго рода (нераспознавание) для категории «books».

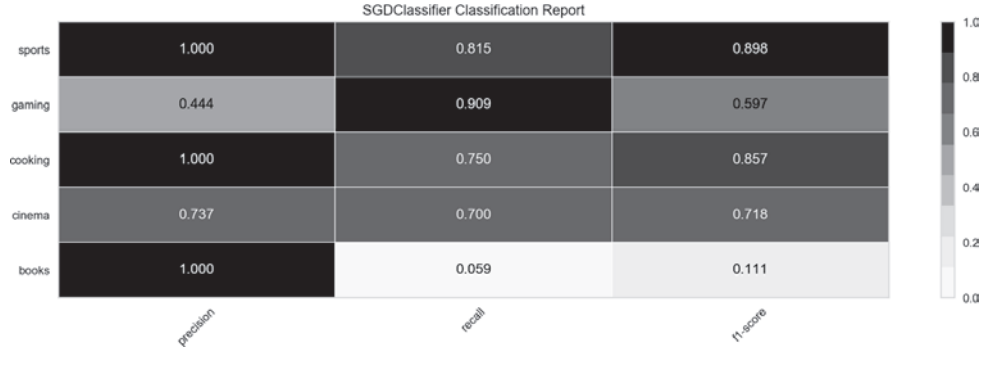


Рис. 8.15. Тепловая карта классификатора методом стохастического градиентного спуска на корпусе «hobbies»

Матрицы несоответствий

Матрица несоответствий включает похожую информацию, как в отчете классификации, но вместо высокоуровневых оценок предлагает более подробные сведения.

Чтобы построить такую визуализацию с использованием Yellowbrick, создадим экземпляр `ConfusionMatrix` и передадим ему желаемый классификатор с именами классов, как мы это делали, создавая экземпляр `ClassificationReport`, и последовательно вызовем методы `fit`, `score` и `poof`:

```
from yellowbrick.classifier import ConfusionMatrix
from sklearn.linear_model import LogisticRegression

...

visualizer = ConfusionMatrix(LogisticRegression(), classes = corpus.
categories)

visualizer.fit(X_train, y_train)
visualizer.score(X_test, y_test)
visualizer.poof()
```

Получившаяся матрица несоответствий, изображенная на рис. 8.16, помогает увидеть, какие классы хуже всего распознаются моделью. В этом примере модель `LogisticRegression` прекрасно справилась с распознаванием категорий «sports» (спорт) и «gaming» (игры), но испытывает большие сложности при распознавании других категорий.

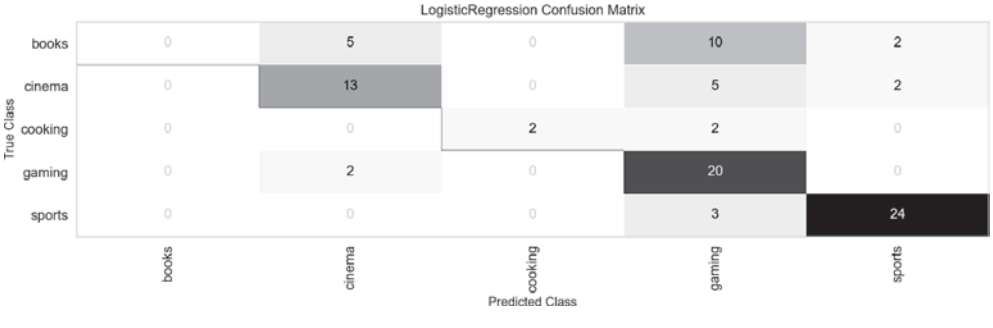


Рис. 8.16. Матрица несоответствий применения классификатора, использующего метод логистической регрессии, к корпусу «hobbies»

Мы можем сравнить качество моделей, подставив их в вызов конструктора `ConfusionMatrix`, как это делалось в случае с `ClassificationReport`. Для сравнения на рис. 8.17 показана матрица несоответствий классификатора `MultinomialNB`,

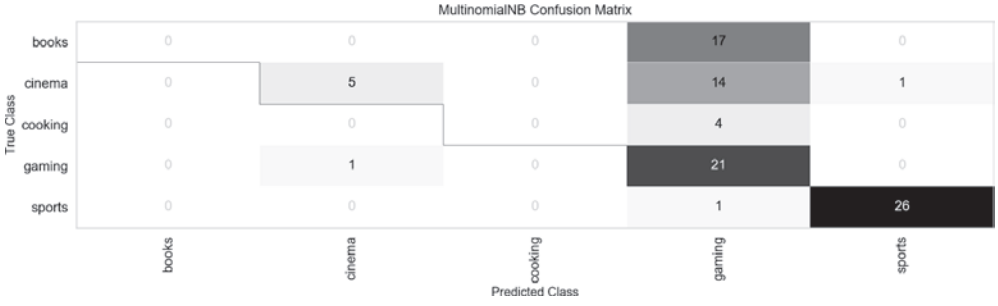


Рис. 8.17. Матрица несоответствий применения полиномиального наивного байесовского классификатора к корпусу «hobbies»

которая показывает, что он тоже недостаточно хорошо справляется с классификацией большинства категорий в корпусе «hobbies» и, похоже, часто путает категории «books» и «gaming».

В общем случае качество модели сильно зависит от контекста, поэтому, определяя адекватность модели, важно выбирать специализированные ориентиры и не полагаться на высокоуровневые метрики, такие как оценка F1. Например, если от нашего гипотетического приложения требуется классифицировать контент, имеющий отношение к спорту, этих моделей вполне достаточно. Но, если целью является поиск обзоров книг, неудовлетворительное качество этих классификаторов предполагает, что нам, возможно, придется пересмотреть оригинальный набор данных.

Визуальная настройка

Когда вызывается метод `fit` объекта `Estimator` из библиотеки Scikit-Learn, модель подбирает параметры алгоритма, лучше соответствующие имеющимся данным. Однако некоторые параметры не связаны напрямую с обучением в объекте `Estimator`. Мы называем их *гиперпараметрами* и определяем в момент создания экземпляра модели.

Гиперпараметры зависят от конкретной модели и включают такие характеристики, как величина штрафа, используемая для регулировки, функция ядра для метода опорных векторов, количество листьев или глубина дерева решений, количество соседей в классификаторе методом ближайших соседей или количество кластеров в кластеризации методом k-средних.

Модели Scikit-Learn часто на удивление успешно справляются со своими задачами почти без изменения гиперпараметров по умолчанию. Это не случайная

удача, а показатель того, что создателями библиотеки накоплен значительный опыт и объем знаний в данной области. Тем не менее после выбора нескольких моделей, которые лучше всего справляются с поставленной задачей, мы должны сделать следующий шаг и поэкспериментировать с гиперпараметрами, чтобы найти наиболее оптимальные настройки для каждой модели.

В этом разделе мы рассмотрим приемы визуального исследования гиперпараметров и, в частности, разберем выбор значения *k* для алгоритма кластеризации методом *k*-средних.

Оценка силуэта и локтевые кривые

Как мы видели в главе 6, метод *k*-средних — это простой алгоритм машинного обучения без учителя, группирующий данные в указанное количество *k* кластеров. Поскольку выбор числа *k* осуществляется пользователем заранее, алгоритму остается только распределить все экземпляры по *k* кластерам, независимо от того, является ли правильным выбор числа *k* для набора данных. Библиотека Yellowbrick реализует два механизма выбора оптимального числа *k* для центроидной кластеризации, *оценка силуэта* (silhouette scores) и *локтевые кривые* (elbow curves), которые мы рассмотрим в этом разделе.

Оценка силуэта

В отсутствие достоверной информации о природе набора данных вычисляется коэффициент силуэта, отражающий плотности кластеров, производимых моделью. Далее, для каждого образца определяется оценка силуэта путем усреднения коэффициента силуэта, вычисляемого как разность среднего расстояния внутри кластера и среднего расстояния до ближайшего кластера от этого образца, нормализованная максимальным значением.

В результате получается значение в диапазоне от 1 до –1, где `1` говорит о том, что кластеры имеют очень высокую плотность, `-1` — что кластеризация выполнена абсолютно неверно, а значения, близкие к нулю, — что присутствуют перекрывающиеся кластеры. Чем выше оценка, тем лучше, потому что высокая оценка присваивается плотным и изолированным друг от друга кластерам. Отрицательные значения указывают, что образцы были отнесены к неверным кластерам, а положительные — что в корпусе действительно имеются дискретные кластеры. Затем полученные оценки можно отобразить в виде диаграммы, чтобы показать меру близости каждой точки в одном кластере к точкам в соседних кластерах.

Для визуализации оценок силуэтов каждого кластера можно использовать объект `SilhouetteVisualizer` из библиотеки Yellowbrick. Поскольку модели

кластеризации с трудом поддаются оценке, визуализаторы из Yellowbrick оберТывают модели кластеризации из Scikit-Learn своим методом `fit()`. После обучения модели можно вызвать метод `poof()` визуализатора и отобразить метрики оценки результатов кластеризации. Чтобы создать диаграмму, сначала обучим модель кластеризации, создадим экземпляр визуализатора, обучим его на данных из корпуса и вызовем метод `poof()`:

```
from sklearn.cluster import KMeans
from yellowbrick.cluster import SilhouetteVisualizer

# Создать модель кластеризации и визуализатор
visualizer = SilhouetteVisualizer(KMeans(n_clusters = 6))
visualizer.fit(docs)
visualizer.poof()
```

Объект `SilhouetteVisualizer` отображает коэффициент силуэта для каждого образца в каждом кластере, показывая, какие кластеры плотные, а какие нет. Вертикальная высота кластера на диаграмме соответствует его размеру, а красная пунктирная линия обозначает глобальное среднее. Эту информацию можно использовать для определения степени сбалансированности кластеров или выбора значения *k* путем сравнения нескольких диаграмм. Например, на рис. 8.18 видно, что некоторые кластеры имеют значительную высоту, но низкую оценку. Это говорит о том, что следует попробовать выбрать большее значение *k*.

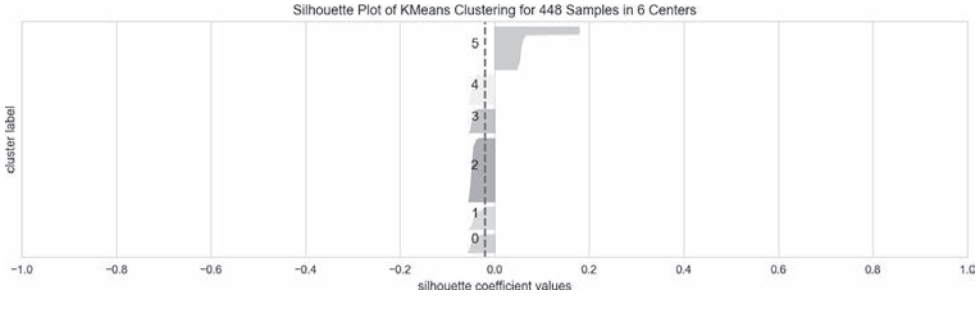


Рис. 8.18. Диаграмма с оценками силуэта результата кластеризации методом *k*-средних

Локтевые кривые

Другой метод визуализации, который можно использовать для подбора значения *k*, — метод локтя (elbow method). Этот метод основан на отображении графика изменения качества моделей с изменением значения *k*. Выбор определяется наличием точки «локтевого сгиба» на кривой (то есть кривая выглядит как рука, согнутая в локте, с явным изменением ее направления).

Объект `KElbowVisualizer` в библиотеке Yellowbrick реализует метод локтя для выбора оптимального числа кластеров при использовании способа кластеризации методом *k*-средних. Для этого нужно создать экземпляр визуализатора, передав ему необученную модель `KMeans()` и диапазон значений для *k* (например, от 4 до 10). Затем вызвать метод `fit()` с документами из корпуса (в следующем примере предполагается, что документы уже векторизованы методом TF–IDF), в результате визуализатор выполнит кластеризацию набора данных методом *k*-средних для каждого значения *k* и рассчитает оценку силуэта, средний коэффициент силуэта по всем образцам. Последующий вызов `poof()` построит график изменения оценки силуэта для разных *k*:

```
from sklearn.cluster import KMeans
from yellowbrick.cluster import KElbowVisualizer

# Создать модель кластеризации и визуализатор
visualizer = KElbowVisualizer(KMeans(), metric = 'silhouette', k = [4,10])
visualizer.fit(docs)
visualizer.poof()
```

Если линия графика имеет вид согнутого «локтя», точка перегиба является лучшим значением *k*; для нас желательно иметь минимально возможное значение *k*, чтобы кластеры не перекрывались. Если данные не имеют четко выраженных кластеров, кривая может получиться слишком гладкой или иметь несколько точек перегиба, метод локтя оказывается бесполезным. В таких случаях можно посоветовать использовать визуализатор `SilhouetteScore` или применить партитивную кластеризацию. Глядя на график, изображенный на рис. 8.19, даже притом, что он имеет несколько точек перегиба, можно предположить, что, выбрав семь кластеров, мы сможем улучшить плотность и изолированность кластеров документов.

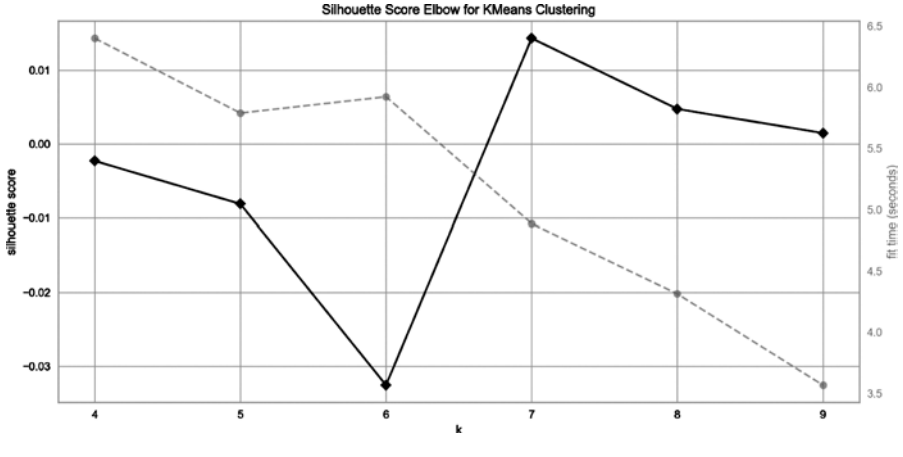


Рис. 8.19. График изменения оценки силуэта для кластеризации методом *k*-средних

В заключение

С какими бы данными мы ни работали — числовыми или текстовыми (или даже с пикселами или нотами) — единственной оценки или даже единственного графика часто недостаточно для процесса тройки выбора модели. Методы визуализации пригодятся вам для исследовательского анализа, конструирования признаков, выбора модели и ее оценки. В комбинации с числовыми оценками они могут помочь получить более полное представление об эффективности. Однако текстовые данные могут вызывать некоторые специфические проблемы для визуализации, особенно в отношении размерности и интерпретируемости.

По нашему опыту, управление выбором помогает получать качественные модели (например, с высокой оценкой F1, изолированными кластерами и т. д.) быстрее и достигать более полного общего представления. Зрительная кора нашего мозга часто намного лучше справляется с выявлением закономерностей, представленных не в числовой, а в визуальной форме. То есть, используя методы визуализации, мы можем более эффективно управлять процессом моделирования.

Несмотря на отсутствие большого многообразия библиотек визуальной диагностики моделирования на текстовых данных, приемы, предлагающие интерактивный интерфейс к машинному обучению и продемонстрированные в этой главе, могут оказаться хорошими инструментами, помогающими снизить барьер между человеком и машиной. В числе библиотек визуализации есть два очень удобных инструмента — Matplotlib и Yellowbrick, — которые вместе позволяют выполнить визуальную фильтрацию, агрегирование, индексирование и форматирование, чтобы помочь отобразить большой корпус с многомерным пространством признаков в более простом для понимания и интерактивном виде.

Одним из наиболее эффективных способов визуализации, представленных в этой главе, являются графы, позволяющие представить огромный объем информации в простом и понятном виде. В главе 9 мы детальнее рассмотрим графовые модели — их способность к эффективной визуальной группировке и моделированию информации, для чего в ином случае потребовались бы более серьезные усилия по конструированию признаков.

9 Графовые методы анализа текста

До этого момента мы использовали традиционные алгоритмы классификации и кластеризации для анализа текста. Эти алгоритмы определяют расстояние между терминами, присваивают веса словосочетаниям и вычисляют вероятности высказываний, помогая нам рассуждать о связях между документами. Однако такие задачи, как машинный перевод, поиск ответов на вопросы и выполнение инструкций, часто требуют более сложных семантических рассуждений.

Например, основываясь на большом количестве новостных статей, как бы вы смоделировали содержащиеся в них описания: действий ключевых игроков и ответных действий других, последовательностей событий, причин и следствий? Используя приемы из главы 7, вы можете извлекать сущности или ключевые фразы или определять темы с применением методов тематического моделирования из главы 6. Но, чтобы смоделировать информацию о связях между этими сущностями, фразами и темами, необходим иной тип структуры данных.

Рассмотрим, как можно выразить такие связи между названиями статей[1]:

```
headlines = ['FDA approves gene therapy',
             'Gene therapy reduces tumor growth',
             'FDA recalls pacemakers']
```

[1] Перевод названий статей:
«Управление одобрило генную терапию»
«Генная терапия замедляет рост опухоли»
«Управление отзывает кардиостимуляторы»

Традиционно подобные фразы кодируются с использованием *представлений смысла текста* (Text Meaning Representation, TMR). Представления смысла имеют вид троек `('субъект', 'утверждение', 'объект')`, например `('FDA', 'recalls', 'pacemakers')`, к которым можно применять логику первого порядка и лямбда-исчисление для семантического анализа.

К сожалению, конструирование представлений TMR часто требует существенного объема предварительных знаний. Например, мы должны знать, что аббревиатура «FDA»[1] определяет действующую сторону, а также что слово «recalling» (отзывает) описывает действие, которое предпринимается одной сущностью в отношении другой. Для большинства приложений анализа естественного языка создать достаточное количество представлений TMR для поддержки осмысленного семантического анализа практически невозможно.

Однако ту же структуру «субъект-утверждение-объект» можно представить в виде *графа*, где утверждения являются *ребрами* между *узлами* — субъектами и объектами, как показано на рис. 9.1. Извлекая пары сущностей и ключевые фразы из заголовков, можно сконструировать графовое представление связей между «кто», «что» и даже «где», «как» и «когда» событий. Благодаря этому мы получим возможность конструировать ответы на аналитические вопросы, такие как «Кто более всего причастен к этому события?» или «Как меняются отношения с течением времени?», выполнив обход графа. Конечно, такой подход не является полноценным семантическим анализом, но он помогает прийти к полезным выводам.

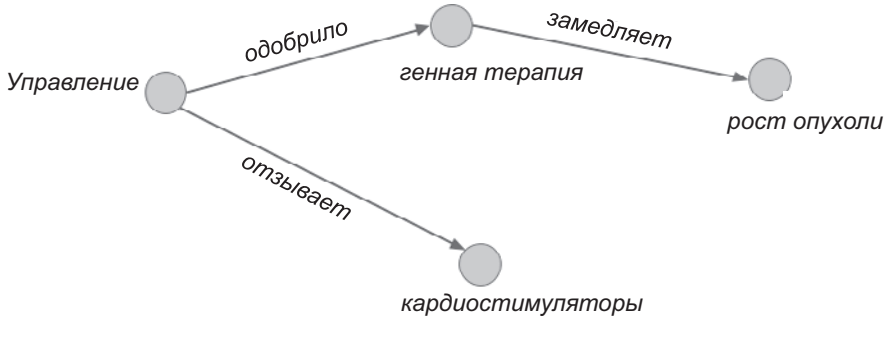


Рис. 9.1. Семантические рассуждения в тексте, представленные в виде графов

[1] FDA (Food and Drug Administration) — Управление по контролю за продуктами и лекарствами (США). — *Примеч. пер.*

В этой главе мы будем анализировать текстовые данные с использованием графовых алгоритмов. Сначала мы построим тезаурус (словарь синонимов), основанный на графах, и определим некоторые наиболее полезные характеристики графа. Затем извлечем социальный граф из нашего корпуса Baleen, связав субъекты, появляющиеся вместе в одних и тех же документах, и используем некоторые простые приемы извлечения и анализа *подграфов*. В заключение познакомимся с подходом к *разрешению сущностей* на основе графов, который называется *нечетким блокированием* (fuzzy blocking).

NetworkX и Graph-tool — это две основные библиотеки для Python, реализующие графовые алгоритмы и модель графа свойств (которую мы исследуем далее в этой главе). Graph-tool масштабируется существенно лучше, чем NetworkX, но реализована на C++ и должна компилироваться из исходного кода. Для визуализации графов мы часто используем написанные на других языках инструменты, такие как Gephi, D3.js и Cytoscape.js. Однако ради простоты в этой главе мы будем придерживаться библиотеки NetworkX.

Вычисление и анализ графов

Главной задачей в анализе графов является определение узлов и ребер. Обычно роль узлов играют сущности реального мира, которые мы хотели бы проанализировать, а роль ребер — отношения между узлами разного вида (и величины).

После определения схемы конструирование графа не представляет большой проблемы. Рассмотрим простой пример моделирования тезауруса в виде графа. Традиционный тезаурус отображает одни слова во множества других слов, имеющих похожий смысл, значение и порядок использования. Тезаурус на основе графа, который представляет слова как узлы, а синонимы как ребра, может принести существенную выгоду, моделируя семантическое сходство как функцию длины и веса пути между любыми двумя связанными терминами.

Создание тезауруса на основе графа

Для реализации только что описанного тезауруса на основе графа используем WordNet[1], большую лексическую базу данных английских слов, сгруппированных во взаимосвязанные *синсеты* (synset), коллекции когнитивных синонимов, выражающих разные понятия. В нашем тезаурусе роль узлов будут играть слова из синсетов WordNet (доступных через интерфейс WordNet библиотеки NLTK), а ребра будут представлены отношениями и взаимосвязями синсетов.

[1] George A. Miller and Christiane Fellbaum, *WordNet: A Lexical Database for English* (1995), http://bit.ly/2GQKXmI

Определим функцию `graph_synsets()`, конструирующую граф и добавляющую в него все узлы и ребра. Она принимает список терминов, а также максимальную глубину, создает неориентированный граф с помощью NetworkX и присваивает ему свойство `name` для быстрой идентификации в последующем. Затем внутренняя функция `add_term_links()` добавляет синонимы, получая их с помощью функции `wn.synsets()` из NLTK, которая возвращает все возможные определения данного слова.

Для каждого определения переберем все синонимы, возвращаемые внутренним методом `lemma_names()`, и добавим узлы и ребра за один шаг вызовом метода `G.add_edge()` из NetworkX. Если заданный порог глубины еще не достигнут, рекурсивно добавим ссылки на термины в синсете. После этого функция `graph_synsets` выполнит обход всех терминов и с помощью нашей рекурсивной функции `add_term_links()` извлечет синонимы, сконструирует ребра и, наконец, вернет получившийся граф:

```
import networkx as nx
from nltk.corpus import wordnet as wn

def graph_synsets(terms, pos = wn.NOUN, depth = 2):
    """
    Создает из терминов terms граф networkx с глубиной depth.
    """
    G = nx.Graph(
        name = "WordNet Synsets Graph for {}".format(", ".join(terms)),
depth = depth,
    )

    def add_term_links(G, term, current_depth):
        for syn in wn.synsets(term):
            for name in syn.lemma_names():
                G.add_edge(term, name)
                if current_depth < depth:
                    add_term_links(G, name, current_depth+1)

    for term in terms:
        add_term_links(G, term, 0)

    return G
```

Чтобы получить описательную статистическую информацию о графе NetworkX, можно воспользоваться функцией `info()`, которая возвращает количество узлов, количество ребер и среднюю степень графа. Теперь проверим нашу функцию: получим с ее помощью графа для слова «trinket» (брелок) и извлечем основные статистики графа:

```
G = graph_synsets(["trinket"])
print(nx.info(G))
```

Вот какие результаты мы получили:

```
Name: WordNet Synsets Graph for trinket
Type: Graph
Number of nodes: 25
Number of edges: 49
Average degree:   3.9200
```

Теперь у нас есть полноценный тезаурус! Вы можете поэкспериментировать, создавая графы для других слов, списков слов или изменяя глубину сбора синонимов.

Анализ структуры графа

Экспериментируя с функцией `graph_synsets` и передавая ей разные термины, можно заметить, что получающиеся графы могут быть очень большими или очень маленькими и иметь более или менее сложную структуру в зависимости от того, как связаны термины. При анализе графы описываются их структурой. В этом разделе мы рассмотрим набор стандартных метрик для описания структуры графов.

В предыдущем разделе вызов `nx.info` вывел количество узлов (или *порядок*) в графе, количество ребер (или *размер*) и среднюю *степень*. *Соседями* узла называют набор узлов, находящихся от него на расстоянии одного перехода, а количество соседей определяет *степень* узла. Средняя степень графа отражает средний размер всех соседей внутри графа.

Диаметр графа — это количество узлов, которые нужно миновать, чтобы пройти *кратчайшим путем* между двумя наиболее удаленными узлами. Получить эту статистику можно с помощью функции `diameter()`:

```
>>> nx.diameter(G)
4
```

В контексте графа для слова «trinket» кратчайший путь `4` может говорить об узкой специализации термина (табл. 9.1), в отличие от других терминов с более широкой интерпретацией (например, «building») или более широким кругом использования («bank»).

Таблица 9.1. Кратчайшие пути для некоторых распространенных терминов

Термин	trinket	bank	hat	building	boat	whale	road	tickle
Диаметр	4	6	2	6	1	5	3	6

Вот некоторые ключевые вопросы, ответы на которые нужно получить в процессе анализа графа:

- Каковы глубина и диаметр графа?
- Является ли граф полносвязанным (то есть имеются ли в графе пути между любыми парами узлов)?
- Если есть несвязанные компоненты, каковы их размеры и другие свойства?
- Можно ли извлечь подграф (или *эго-граф*, о котором мы расскажем ниже) для конкретного узла?
- Можно ли создать подграф, фильтрующий определенный объем или тип информации? Например, можно ли из 25 возможных результатов вернуть только 5 самых важных?
- Можно ли вставлять узлы или ребра разных типов для создания разнородных структур? Например, можно ли организовать представление синонимов и антонимов в одном графе?

Визуальный анализ графов

Графы также можно подвергнуть визуальному анализу, однако с настройками по умолчанию могут получиться «комки», которые трудно распутать (ниже мы подробнее поговорим об этом). Одним из популярных механизмов анализа структуры графов является пружинная модель блоков. Пружинная модель блоков отображает каждый узел как массу (или блок), а ребра — как пружины с силой притягивания и отталкивания, пропорциональной их весу. Это предотвращает наложение друг на друга узлов, которые они соединяют, и часто помогает получить более управляемые изображения графов.

С помощью встроенной функции `nx.spring_layout` из библиотеки NetworkX можно нарисовать граф синсета нашего слова «trinket». Для этого сначала получим позиции узлов с пружинной компоновкой. Затем нарисуем узлы в виде больших белых окружностей с очень тонкими линиями, чтобы текст был читаемым. Далее нарисуем текстовые метки и ребра (определив достаточно большой размер шрифта и светло-серый цвет для ребер, чтобы текст легко читался). Наконец, удалим деления и подписи из диаграммы, так как они в контексте графа тезауруса не несут полезной информации, и отобразим получившуюся диаграмму (рис. 9.2):

```
import matplotlib.pyplot as plt

def draw_text_graph(G):
    pos = nx.spring_layout(G, scale = 18)
    nx.draw_networkx_nodes(
        G, pos, node_color = "white", linewidths = 0, node_size = 500
```

```
)
nx.draw_networkx_labels(G, pos, font_size = 10)
nx.draw_networkx_edges(G, pos, edge_color = 'lightgrey')

plt.tick_params(
    axis = 'both',       # изменяются обе оси, X и Y
    which = 'both',      # затрагиваются большие и малые деления
    bottom = 'off',      # скрыть деления вдоль нижнего края
    left = 'off',        # скрыть деления вдоль левого края
    labelbottom = 'off', # скрыть подписи вдоль нижнего края
    labelleft = 'off')   # скрыть подписи вдоль левого края

plt.show()
```

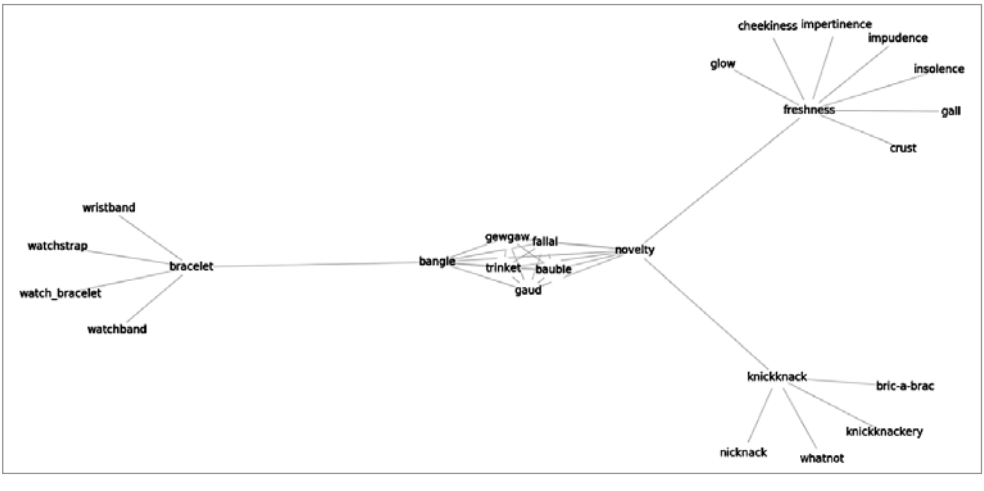


Рис. 9.2. Пружинная модель помогает получить более управляемые изображения графов

В следующем разделе мы исследуем методы извлечения и анализа графов, которые используются при работе конкретно с текстом.

Извлечение графов из текста

Извлечение графа из текста — сложная задача. Ее решение обычно зависит от предметной области, и, вообще говоря, поиск структурированных элементов в неструктурированных или полуструктурированных данных определяется контекстно-зависимыми аналитическими вопросами.

Мы предлагаем разбить эту задачу на более мелкие шаги, организовав простой процесс анализа графов, как показано на рис. 9.3.

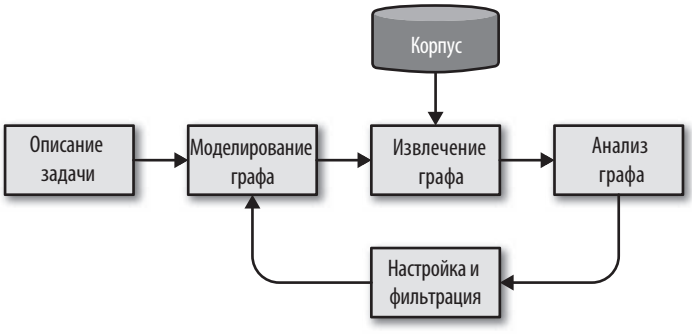


Рис. 9.3. Процесс анализа графа, извлеченного из текста

В этом процессе мы сначала определяем сущности и связи между ними, исходя из описания задачи. Далее, на основе этой схемы определяем методику выделения графа из корпуса, используя метаданные, документы в корпусе и словосочетания или лексемы в документах для извлечения данных и связей между ними. Методика выделения графа — это циклический процесс, который можно применить к корпусу, сгенерировать граф и сохранить этот граф на диск или в память для дальнейшей аналитической обработки.

На этапе анализа производятся вычисления на извлеченном графе, например, кластеризация, структурный анализ, фильтрация или оценка, и создается новый граф, который используется в приложениях. По результатам этапа анализа мы можем вернуться к началу цикла, уточнить методику и схему, извлечь или свернуть группы узлов или ребер, чтобы попробовать добиться более точных результатов.

Создание социального графа

Рассмотрим наш корпус новостных статей и задачу моделирования связей между разными сущностями в тексте. Если рассматривать вопрос различий в охвате между разными информационными агентствами, можно построить граф из элементов, представляющих названия публикаций, имена авторов и источники информации. А если целью является объединение упоминаний одной сущности во множестве статей, в дополнение к демографическим деталям наши сети могут зафиксировать форму обращения (уважительную и другие). Интересующие нас сущности могут находиться в структуре самих документов или содержаться непосредственно в тексте.

Допустим, наша цель — выяснить людей, места и все что угодно, связанные друг с другом в наших документах. Иными словами, нам нужно построить

социальную сеть, выполнив серию преобразований, как показано на рис. 9.4. Начнем конструирование графа с применения класса `EntityExtractor`, созданного в главе 7. Затем добавим преобразователи, один из которых отыскивает пары связанных сущностей, а второй преобразует эти пары в граф.

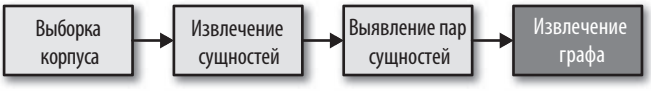


Рис. 9.4. Конвейер извлечения графа сущностей

Поиск пар сущностей

Наш следующий шаг — создание класса `EntityPairs`, который получает документы в виде списков сущностей (созданных классом `EntityExtractor` из главы 7). Этот класс должен действовать как преобразователь в конвейере `Pipeline` из Scikit-Learn, а значит, наследовать классы `BaseEstimator` и `TransformerMixin`, как рассказывалось в главе 4. Предполагается, что сущности в одном документе безусловно связаны друг с другом, поэтому добавим метод `pairs`, использующий функцию `itertools.permutations` для создания всех возможных пар сущностей в одном документе. Наш метод `transform` будет вызывать `pairs` для каждого документа в корпусе:

```
import itertools
from sklearn.base import BaseEstimator, TransformerMixin

class EntityPairs(BaseEstimator, TransformerMixin):
    def __init__(self):
        super(EntityPairs, self).__init__()

    def pairs(self, document):
        return list(itertools.permutations(set(document), 2))

    def fit(self, documents, labels = None):
        return self

    def transform(self, documents):
        return [self.pairs(document) for document in documents]
```

Теперь можно последовательно извлечь сущности из документов и составить пары. Но мы пока не можем отличить пары сущностей, встречающихся часто, от пар, встречающихся только один раз. Мы должны как-то закодировать вес связи между сущностями в каждой паре, чем мы и займемся в следующем разделе.

Графы свойств

Математическая модель графа определяет только наборы узлов и ребер и может быть представлена в виде матрицы смежности (adjacency matrix), которой можно пользоваться в самых разных вычислениях. Но она не поддерживает механизм моделирования силы или типов связей. Появляются ли две сущности только в одном документе или во многих? Встречаются ли они вместе в статьях определенного жанра? Для поддержки подобных рассуждений нам нужен некий способ, позволяющий сохранить значимые свойства в узлах и ребрах графа.

Модель графа свойств позволяет встроить в граф больше информации, тем самым расширяя наши возможности. В графе свойств узлами являются объекты с входящими и исходящими ребрами и, как правило, содержащие поле `type`, напоминая таблицу в реляционной базе данных. Ребра — это объекты, определяющие начальную и конечную точки; эти объекты обычно содержат поле `label`, идентифицирующее тип связи, и поле `weight`, определяющее силу связи. Применяя графы для анализа текста, в роли узлов мы часто используем существительные, а в роли ребер — глаголы. После перехода к этапу моделирования это позволит нам описать *типы* узлов, *метки* связей и предполагаемую структуру графа.

Реализация извлечения графа

Теперь можно определить класс `GraphExtractor`, который преобразует сущности в узлы и присвоит вес ребрам, исходя из частоты совместного появления сущностей в корпусе. Наш класс инициализирует граф NetworkX, а его метод `transform` перебирает документы в корпусе (списки пар сущностей) и проверяет наличие в графе ребра между ними. Если ребро существует, его свойство `weight` увеличивается на `1`. Если ребро отсутствует, вызовом метода `add_edge` создается новое ребро с весом `1`. По аналогии с конструированием графа тезауруса, метод `add_edge` также добавляет новый узел, если какой-то элемент пары отсутствует в графе:

```
import networkx as nx

class GraphExtractor(BaseEstimator,TransformerMixin):
    def __init__(self):
        self.G = nx.Graph()

    def fit(self, documents, labels = None):
        return self

    def transform(self, documents):
```

```
        for document in documents:
            for first, second in document:
                if (first, second) in self.G.edges():
                    self.G.edges[(first, second)]['weight'] + = 1
                else:
                    self.G.add_edge(first, second, weight = 1)
        return self.G
```

Теперь можно определить конвейер в виде объекта `Pipeline`, извлекающий сущности, отыскивающий их пары и конструирующий граф:

```
if __name__ == '__main__':
    from reader import PickledCorpusReader
    from sklearn.pipeline import Pipeline

    corpus = PickledCorpusReader('../corpus')
    docs = corpus.docs()

    graph = Pipeline([
        ('entities', EntityExtractor()),
        ('pairs', EntityPairs()),
        ('graph', GraphExtractor())
    ])

    G = graph.fit_transform(docs)
    print(nx.info(G))
```

Граф, построенный на основе корпуса Baleen, имеет следующие характеристики:

```
Name: Entity Graph
Type: Graph
Number of nodes: 29176
Number of edges: 1232644
Average degree: 84.4971
```

Исследование социального графа

Теперь, когда у нас появилась графовая модель отношений между разными сущностями в корпусе, можно начинать задавать интересные вопросы об этих отношениях. Например, является ли наш граф социальной сетью? Как предполагается, в графе социальной сети можно увидеть некоторые очень специфические структуры, такие как области *концентрации* точек или некоторые узлы, имеющие больше ребер, чем в среднем по графу.

В этом разделе мы посмотрим, как для поддержки анализа можно использовать такие метрики из теории графов, как мера центральности, распределение узлов по числу связей и коэффициент кластеризации.

Центральность

В контексте обсуждения сетей наиболее важные узлы занимают *центральное место* в графе, потому что они прямо или косвенно связаны с большинством узлов. Мера центральности помогает понять связь конкретного узла с ближайшими соседями в сети и выявить сущности с наибольшей престижностью и влиянием. В этом разделе мы сравним несколько способов вычисления меры центральности, включая *центральность по степени* (degree centrality), *центральность по посредничеству* (betweenness centrality), *центральность по близости* (closeness centrality), *центральность по собственному вектору* (eigenvector centrality) и *меру авторитетности*, или важности (pagerank).

Сначала напишем функцию, которая принимает произвольную меру центральности в виде именованного аргумента и использует ее для ранжирования наиболее значимых `n` узлов и присваивания каждому свойства с оценкой. Библиотека NetworkX реализует алгоритмы вычисления центральности в виде функций верхнего уровня, принимающих граф `G` в первом аргументе и возвращающих словари оценок для всех узлов в графе. Для присваивания вычисленных значений оценок наша функция будет использовать `nx.set_node_attributes()`:

```
import heapq
from operator import itemgetter

def nbest_centrality(G, metrics, n = 10):
    # Вычисляет оценку центральности для каждой вершины
    nbest = {}
    for name, metric in metrics.items():
        scores = metric(G)

        # Записать оценку в свойство узла
        nx.set_node_attributes(G, name = name, values = scores)

        # Найти n узлов с наивысшей оценкой и вернуть их вместе с индексами
        topn = heapq.nlargest(n, scores.items(), key = itemgetter(1))
        nbest[name] = topn

    return nbest
```

Этот интерфейс можно использовать для автоматического присваивания оценок узлам, чтобы потом сохранить их на диск или использовать для визуализации.

Простейшая оценка центральности, *центральность по степени* (degree centrality), является мерой популярности, которая определяется как количество соседей данного узла, нормализованное общим количеством узлов в графе. Центральность по степени отражает степень связанности узла и может интерпретироваться как уровень влияния или значимости. В отличие от централь-

ности по степени, отражающей степень связанности узла, *центральность по посредничеству* (betweenness centrality) показывает, насколько важную роль играет данный узел в связанности графа. Центральность по посредничеству вычисляется как отношение кратчайших путей, пролегающих через данный узел, к общему числу всех *кратчайших путей*.

Вот как можно использовать конвейер извлечения сущностей и функцию `nbest_centrality` для сравнения 10 наиболее центральных узлов, определяемых по величине степени центральности и центральности по посредничеству:

```
from tabulate import tabulate

corpus = PickledCorpusReader('../corpus')
docs = corpus.docs()

graph = Pipeline([
    ('entities', EntityExtractor()),
    ('pairs', EntityPairs()),
     ('graph', GraphExtractor())
])

G = graph.fit_transform(docs)

centralities = {"Degree Centrality" : nx.degree_centrality,
                "Betweenness Centrality" : nx.betweenness_centrality}

centrality = nbest_centrality(G, centralities, 10)

for measure, scores in centrality.items():
    print("Rankings for {}:".format(measure))
    print((tabulate(scores, headers = ["Top Terms", "Score"])))
    print("")
```

Судя по результатам, наборы узлов, полученные методами оценки степени центральности и центральности по посредничеству, включают одни и те же наиболее центральные узлы (например, «american», «new york», «trump», «twitter»). Эти влиятельные и тесно связанные узлы соответствуют сущностям, появляющимся во многих документах, и располагаются на главных «перекрестках» в графе:

```
Rankings for Degree Centrality:
Top Terms      Score
----------    ---------
american       0.054093
new york       0.0500643
washington     0.16096
america        0.156744
united states  0.153076
```

```
los angeles    0.139537
republican     0.130077
california     0.120617
trump          0.116778
twitter        0.114447

Rankings for Betweenness Centrality:
Top Terms      Score
----------    ---------
american       0.224302
new york       0.214499
america        0.0258287
united states  0.0245601
washington     0.0244075
los angeles    0.0228752
twitter        0.0191998
follow         0.0181923
california     0.0181462
new            0.0180939
```

Хотя центральность по степени и центральность по посредничеству можно использовать как меру известности, мы часто обнаруживаем, что наиболее связанный узел имеет обширное окружение, но отделен от подавляющего большинства узлов в графе.

Рассмотрим контекст конкретного *эго-графа*, то есть подграфа, из нашего полного графа, образующего сеть с точки зрения одного конкретного узла. Извлечь такой эго-граф можно с помощью метода `nx.ego_graph` из библиотеки NetworkX:

```
H = nx.ego_graph(G, "hollywood")
```

Теперь у нас есть граф, включающий только связи с одной конкретной сущностью («Hollywood»). Такое преобразование не только помогает значительно уменьшить размер графа (и тем самым увеличить эффективность последующего поиска), но также позволяет строить рассуждения об этой сущности.

Допустим, нам нужно выявить сущности, которые в среднем находятся ближе к другим сущностям в графе «Hollywood». *Центральность по близости* (closeness centrality) — в библиотеке NetworkX реализуется методом `nx.closeness_centrality` — основана на статистической мере исходящих путей и определяется как средняя длина пути ко всем другим узлам, нормализованная размером графа. В общем случае центральность по близости определяет, как быстро распространится по сети информация, исходящая из определенного узла.

Центральность по собственному вектору (eigenvector centrality), напротив, говорит о том, что важность узла тем выше, чем больше количество важных

узлов, с которыми он связан, то есть выражает «известность по связям». Это означает, что узлы с небольшим количеством очень важных соседей по своей влиятельности могут превосходить узлы с большим количеством связей, соединяющих их с маловажными узлами. Центральность по собственному вектору лежит в основе нескольких других мер центральности, включая *центральность Каца* (Katz centrality) и известный алгоритм авторитетности *PageRank*.

Мы можем воспользоваться нашей функцией `nbest_centrality`, чтобы посмотреть, какие узлы каждая из этих мер считает наиболее важными в эго-графе «Hollywood»:

```
hollywood_centralities = {"closeness" : nx.closeness_centrality,
                          "eigenvector" : nx.eigenvector_centrality_numpy,
                          "katz" : nx.katz_centrality_numpy,
                          "pagerank" : nx.pagerank_numpy,}

hollywood_centrality = nbest_centrality(H, hollywood_centralities, 10)
for measure, scores in hollywood_centrality.items():
    print("Rankings for {}:".format(measure))
    print((tabulate(scores, headers = ["Top Terms", "Score"])))
    print("")
```

Как показывают полученные результаты, сущности с наивысшей центральностью по близости (например, «video», «british», «Brooklyn») не особенно показательны и, скорее всего, определяют скрытые силы, связывающие более заметные структуры с наибольшей известностью. Центральность по собственным векторам, PageRank и центральность Каца уменьшают влияние количества связей (например, наиболее часто появляющихся сущностей) и демонстрируют влияние «скрытых сил», выделяя тесно связанные сущности («republican», «Obama») и их сторонников:

```
Rankings for Closeness Centrality:
Top Terms      Score
----------    ---------
hollywood       1
new york        0.687801
los angeles     0.651051
british         0.6356
america         0.629222
american        0.625243
video           0.621315
london          0.612872
china           0.612434
brooklyn        0.607227

Rankings for Eigenvector Centrality:
Top Terms       Score
```

```
----------     ---------
hollywood       0.0510389
new york        0.0493439
los angeles     0.0485406
british         0.0480387
video           0.0480122
china           0.0478956
london          0.0477556
twitter         0.0477143
new york city   0.0476534
new             0.0475649

Rankings for PageRank Centrality:
Top Terms       Score
----------     ---------
hollywood       0.0070501
american        0.00581407
new york        0.00561847
trump           0.00521602
republican      0.00513387
america         0.00476237
donald trump    0.00453808
washington      0.00417929
united states   0.00398346
obama           0.00380977

Rankings for Katz Centrality:
Top Terms           Score
----------         ---------
video               0.107601
washington          0.104191
chinese             0.1035
hillary             0.0893112
cleveland           0.087653
state               0.0876141
muslims             0.0840494
editor              0.0818608
paramount pictures  0.0801545
republican party    0.0787887
```

Для определения центральности по посредничеству и по близости требуется найти все кратчайшие пути в графе, на что может потребоваться очень много времени, особенно для больших графов. Поэтому для большей производительности предпочтительнее использовать низкоуровневые реализации, способные распараллеливать вычисления. В библиотеке Graph-tool есть такие механизмы для вычисления центральности по посредничеству и по близости, поэтому она является хорошим выбором для обработки больших графов.

Структурный анализ

Как было показано выше в этой главе, визуальный анализ графов позволяет выявлять интересные закономерности в их структуре. Мы можем нарисовать наш эго-граф «Hollywood», чтобы исследовать его структуру, по аналогии с графом тезауруса, использовав метод `nx.spring_layout` и передав ему значение *k*, определяющее минимальное расстояние между узлами, и именованный аргумент `iterations`, чтобы гарантировать достаточно заметное разделение узлов (рис. 9.5).

```
H = nx.ego_graph(G, "hollywood")
edges, weights = zip(*nx.get_edge_attributes(H, "weight").items())
pos = nx.spring_layout(H, k = 0.3, iterations = 40)

nx.draw(
    H, pos, node_color = "skyblue", node_size = 20, edgelist = edges,
    edge_color = weights, width = 0.25, edge_cmap = plt.cm.Pastel2,
    with_labels = True, font_size = 6, alpha = 0.8)
plt.show()
```

Многие графы большого размера страдают проблемой «комка шерсти», когда узлы и ребра оказываются сконцентрированными настолько плотно, что в них сложно заметить значимые структуры. В таких случаях мы часто пытаемся выявить структуры, прибегая к исследованию распределения узлов по числу связей.

Рис. 9.5. Связи узла «Hollywood» в корпусе Baleen

Это распределение можно исследовать с помощью метода `distplot` из библиотеки Seaborn, передав в параметре `norm_hist` значение `True`, чтобы высота графика отражала степень плотности, а не простое количество:

```
import seaborn as sns

sns.distplot([G.degree(v) for v in G.nodes()], norm_hist = True)
plt.show()
```

Во многих графах большинство узлов имеют относительно небольшое количество связей, поэтому обычно мы ожидаем увидеть значительное смещение вправо в распределении количества связей, как на графике Baleen Entity Graph, на рис. 9.6 вверху слева. Небольшое число узлов с наибольшим количеством связей являются точками концентрации из-за их частого появления и большого количества связей в корпусе.

Некоторые социальные сети, которые называют *безмасштабными сетями* (scale-free networks), подчиняются закону степенного распределения и имеют особенно высокую *надежность*. Такие структуры указывают на устойчивую сеть, связанность которой не зависит от какого-то одного конкретного узла (в нашем случае это может быть человек или организация).

Интересно отметить, что распределение количества связей в некоторых эго-графах имеет совершенно разное поведение, как, например, в эго-графе «Hollywood», имеющем почти симметричное распределение вероятностей (на рис. 9.6, вверху справа).

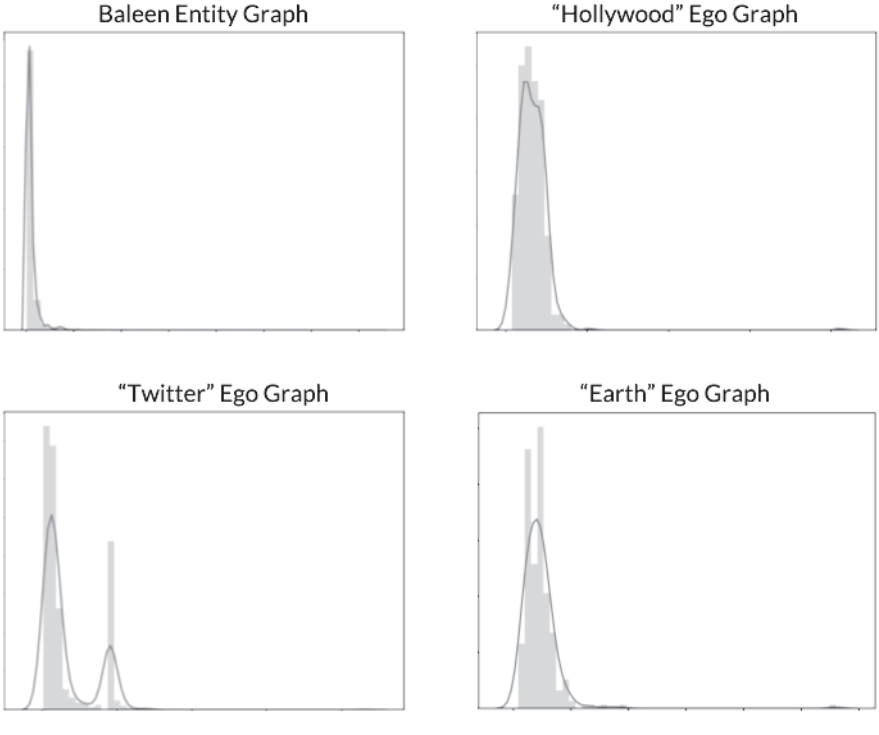


Рис. 9.6. Гистограммы распределения количества связей

В числе других метрик сетей, заслуживающих внимания, можно назвать *коэффициент кластеризации* (`nx.average_clustering`) и *транзитивность* (`nx.transitivity`). Обе можно использовать для оценки организации социальной сети. Коэффициент кластеризации — это вещественное число, которое принимает значение `0`, когда вообще отсутствуют кластеры, и `1`, когда граф целиком состоит из не связанных между собой групп (`nx.graph_number_of_cliques`). Транзитивность определяет вероятность наличия общих соседей у двух узлов:

```
print("Baleen Entity Graph")
print("Average clustering coefficient: {}".format(nx.average_clustering(G)))
print("Transitivity: {}".format(nx.transitivity(G)))
print("Number of cliques: {}".format(nx.graph_number_of_cliques(G)))

print("Hollywood Ego Graph")
print("Average clustering coefficient: {}".format(nx.average_clustering(H)))
print("Transitivity: {}".format(nx.transitivity(H)))
print("Number of cliques: {}".format(nx.graph_number_of_cliques(H)))
```

```
Baleen Entity Graph
Average clustering coefficient: 0.7771459590548481
Transitivity: 0.29504798606584176
Number of cliques: 51376

Hollywood Ego Graph
Average clustering coefficient: 0.9214236425410913
Transitivity: 0.6502420989124886
Number of cliques: 348
```

В контексте нашего графа сущностей высокие коэффициенты кластеризации и транзитивности говорят, что данная социальная сеть является очень организованной. Например, транзитивность эго-графа «Hollywood» говорит, что 65 % пар сущностей, имеющих общие связи, сами являются соседями!

Феномен тесного мира можно наблюдать в графах, где большинство узлов, даже если они не являются соседями, достижимо из любого конкретного узла за небольшое количество переходов. Сеть тесного мира можно идентифицировать по структурным признакам: они часто содержат множество групп, имеют высокий коэффициент кластеризации и высокое значение транзитивности. В контексте нашего графа сущностей это означает, что большинство незнакомцев в такой сети связано друг с другом очень короткими цепочками общих знакомых.

Разрешение сущностей

Одной из проблем, которые нам предстоит обсудить, является многообразие способов появления сущностей. Поскольку мы почти не подвергали нормализации извлеченные сущности, можно ожидать увидеть множество вариантов

написания, форм обращения и соглашений именования, таких как псевдонимы и сокращения. В результате может появиться большое количество узлов, соответствующих одной сущности (например, узлов «America», «US» и «the United States»).

На рис. 9.7 можно видеть множество часто неоднозначных ссылок на «Hilton», которые обозначают не только всю семью в целом, но также отдельных ее членов, корпорацию и конкретные гостиницы в разных городах. В идеале было бы желательно идентифицировать такие многозначные узлы и преобразовать их в уникальные узлы сущностей.

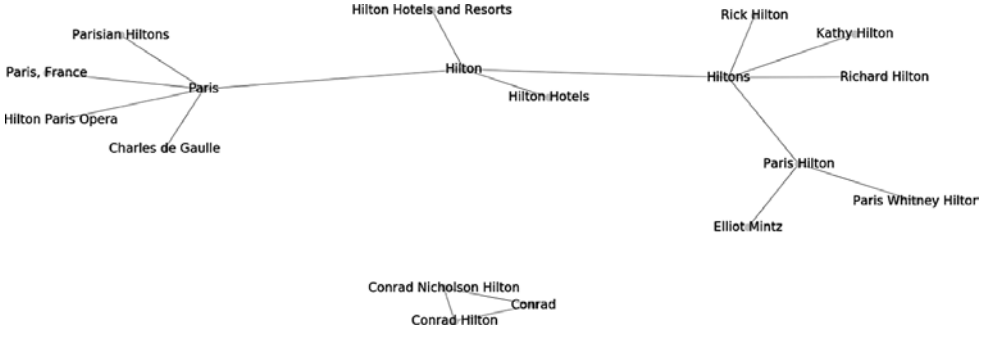


Рис. 9.7. Многозначные ссылки на сущности в графе

Под *разрешением сущностей* понимаются вычислительные приемы, идентифицирующие, группирующие или связывающие цифровые *упоминания* (записи) некоторого объекта в реальном мире (сущности). Разрешение сущностей является частью процесса извлечения данных и гарантирует объединение всех упоминаний одной уникальной сущности под одной ссылкой.

Такие задачи, как дедупликация (удаление повторяющихся элементов), связывание записей (объединение двух записей) и канонизация (создание единственной представительной записи для сущности), опираются на вычисление *степени сходства* (*расстояния*) двух записей и определение их соответствия.

Разрешение сущностей в графе

Как объясняют Бхаттахари (Bhattacharya) и Гетур (Getoor) (2005 г.)[1], граф, где отдельным сущностям может соответствовать несколько узлов, в действитель-

[1] Indrajit Bhattacharya and Lise Getoor, *Entity Resolution in Graph Data* (2005), http://bit.ly/2GQKXDe

ности является не графом сущностей, а графом ссылок. Графы ссылок представляют большую проблему для задач семантического анализа, потому что могут искажать отношения между сущностями.

Разрешение сущностей приближает нас к анализу и извлечению полезной информации об истинной структуре реальной сети, устраняя неясность, вызванную многообразием возможностей обозначения одной и той же сущности. В этом разделе мы исследуем методологию, включающую дополнительный шаг разрешения сущностей в конвейер извлечения графа, как показано на рис. 9.8.

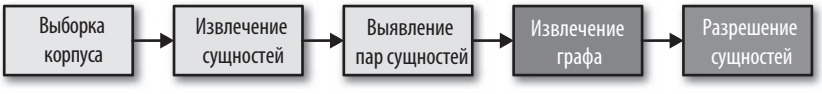


Рис. 9.8. Конвейер извлечения графа сущностей

Прежде чем приступить к разрешению сущностей в нашем графе «Hilton», определим функцию `pairwise_comparisons`, которая принимает граф NetworkX и использует метод `itertools.combinations` для создания генератора всех возможных пар узлов:

```
import networkx as nx
from itertools import combinations

def pairwise_comparisons(G):
    """
    Создает генератор пар узлов.
    """
    return combinations(G.nodes(), 2)
```

К сожалению, даже такой маленький набор данных, как наш граф «Hilton», состоящий всего из 18 узлов, породит 153 попарных сравнения. Учитывая, что сравнение подобия обычно является дорогостоящей операцией, вовлекающей динамическое программирование и другие ресурсоемкие методы вычислений, совершенно очевидно, что оно плохо масштабируется. Мы можем выполнить несколько простых операций, таких как удаление пар, которые никогда не совпадут, и тем самым уменьшить число сравнений. Чтобы такие решения можно было принимать автоматически или с помощью пользователя, нужен механизм, выявляющий потенциально похожие и отсеивающий очевидно разные ссылки. Одним из таких механизмов является блокирование.

Блокирование по структуре

Блокирование — это стратегия уменьшения количества необходимых попарных сравнений с использованием структуры естественного графа. Блокирование обеспечивает разрешение сущностей и предлагает два основных преимущества: увеличение производительности за счет уменьшения объема вычислений и сужение пространства поиска, чтобы предложить вероятные дубликаты пользователю.

Одно из разумных предположений, которое мы могли бы сделать: если два узла имеют ребро, связывающее их с одной и той же сущностью (например, «Hilton Hotels» и «Hilton Hotels and Resorts» имеют ребра, связывающие их с «Hilton»), скорее всего, они являются ссылками на одну и ту же сущность. Если у нас останутся только наиболее вероятные совпадения, это значительно уменьшит объем необходимых вычислений. Для сравнения ребер любых двух узлов и определения подобия соседей можно использовать метод `neighbors` из библиотеки NetworkX:

```
def edge_blocked_comparisons(G):
    """
    Генератор попарных сравнений, который выявляет вероятно подобные
    узлы, связанные ребрами с одной и той же сущностью.
    """
    for n1, n2 in pairwise_comparisons(G):
        hood1 = frozenset(G.neighbors(n1))
        hood2 = frozenset(G.neighbors(n2))
        if hood1 & hood2:
            yield n1,n2
```

Нечеткое блокирование

Даже после выполнения процедуры блокирования все еще могут оставаться ненужные попарные сравнения. Например, даже притом, что оба узла, «Kathy Hilton» и «Richard Hilton», связаны с одним и тем же узлом «Hiltons», они явно обозначают разных людей. Хотелось бы иметь возможность еще больше упростить граф, определив наиболее вероятные совпадения с помощью некоторой меры сходства, определяющей расстояние между «Kathy Hilton» и «Richard Hilton».

Существует несколько методов вычисления расстояния между строками, большинство из которых предназначено для конкретных случаев. Здесь мы продемонстрируем реализацию на основе метода `partial_ratio` из библиотеки `fuzzywuzzy`, который определяет расстояние Левенштейна, то есть количество

операций удаления, вставки и подстановки, необходимых для преобразования первой строки во вторую.

Определим функцию `similarity`, принимающую два узла NetworkX, определяющую расстояния между строковыми названиями узлов и их типами (хранящимися в атрибутах узлов) и возвращающую их среднее:

```
from fuzzywuzzy import fuzz

def similarity(n1, n2):
    """
    Возвращает среднюю оценку подобия, сравнивая поля двух сущностей.
    Обратите внимание, что, если сущности не имеют полей для сопоставления,
    функция вернет ноль.
    """
    scores = [
        fuzz.partial_ratio(n1, n2),
        fuzz.partial_ratio(G.node[n1]['type'], G.node[n2]['type'])
    ]

    return float(sum(s for s in scores)) / float(len(scores))
```

Если две сущности имеют очень близкие имена (например, «Richard Hilton» и «Rick Hilton») и один и тот же тип (например, PERSON), они получат более высокую оценку, чем сущности с одинаковыми именами, но разными типами (например, PERSON и ORG).

Теперь, реализовав возможность идентифицировать высоковероятные совпадения, можно внедрить ее как фильтр в новую функцию `fuzzy_blocked_comparison`. Эта функция будет перебирать все возможные пары узлов и определять величину структурного подобия между ними. Для пар с высокой оценкой она вычислит их сходство и вернет пары с похожими соседями, для которых высока вероятность подобия (выше некоторого порога, который реализован как именованный аргумент со значением по умолчанию 65 %):

```
def fuzzy_blocked_comparisons(G, threshold = 65):
    """
    Генератор попарных сравнений, выявляющий подобные узлы,
    связанные с одной и той же сущностью, но отсекающий пары,
    степень подобия которых ниже заданного порога threshold.
    """
    for n1, n2 in pairwise_comparisons(G):
        hood1 = frozenset(G.neighbors(n1))
        hood2 = frozenset(G.neighbors(n2))
        if hood1 & hood2:
            if similarity(n1, n2) > threshold:
                yield n1,n2
```

Это очень эффективный способ уменьшения числа попарных сравнений, потому что сначала он выполняет менее ресурсоемкое сравнение по соседям, и только потом производит более дорогостоящее вычисление оценки подобия строк. Уменьшение общего числа сравнений получается весьма существенным:

```
def info(G):
    """
    Обертка для nx.info с несколькими вспомогательными функциями.
    """
    pairwise = len(list(pairwise_comparisons(G)))
    edge_blocked = len(list(edge_blocked_comparisons(G)))
    fuzz_blocked = len(list(fuzzy_blocked_comparisons(G)))

    output = [""]
    output.append("Number of Pairwise Comparisons: {}".format(pairwise))
    output.append("Number of Edge Blocked Comparisons: {}".format
                  (edge_blocked))
    output.append("Number of Fuzzy Blocked Comparisons: {}".format(fuzz_
blocked))

    return nx.info(G) + "\n".join(output)

Name: Hilton Family
Type: Graph
Number of nodes: 18
Number of edges: 17
Average degree: 1.8889
Number of Pairwise Comparisons: 153
Number of Edge Blocked Comparisons: 32
Number of Fuzzy Blocked Comparisons: 20
```

Теперь мы определили 20 пар узлов, которые, вероятнее всего, ссылаются на одну и ту же сущность, более чем на 85 % уменьшив количество попарных сравнений.

Теперь можно реализовать свой преобразователь, чтобы встроить его в конвейер и с его помощью преобразовать граф ссылок в список сравнений методом нечеткого блокирования. Например, реализацию `FuzzyBlocker` можно начать со следующих методов и расширять ее по мере появления конкретных требований к разрешению сущностей:

```
from sklearn.base import BaseEstimator, TransformerMixin

class FuzzyBlocker(BaseEstimator, TransformerMixin):

    def __init__(self, threshold = 65):
        self.threshold = threshold
```

```
    def fit(self, G, y = None):
        return self

    def transform(self, G):
        return fuzzy_blocked_comparisons(G, self.threshold)
```

В зависимости от контекста и порога нечеткого сопоставления объединяемые узлы можно свернуть в один узел (при необходимости обновив свойства ребра) или передать эксперту в предметной области для проверки вручную. В любом случае, разрешение сущностей часто оказывается важным первым шагом в построении качественных приложений данных, и, как мы видели в этом разделе, графы способны сделать этот процесс более эффективным и действенным.

В заключение

Графы часто используются для представления и моделирования сложных систем реального мира, таких как коммуникационные сети и биологические экосистемы. Однако их также можно использовать для структурирования более общих задач. Проявив творческий подход, можно представить в виде графа почти любую задачу.

Первое время проблема извлечения графа кажется сложной, но это всего лишь еще один итеративный процесс, требующий разумного подхода к обработке языка и творческой смекалки при моделировании целевых данных. Подобно другим семействам моделей, представленным в этой книге, итеративное уточнение и анализ являются частью *тройки выбора модели*, и для увеличения их производительности с успехом можно использовать такие приемы, как нормализация, фильтрация и агрегирование.

Однако, в отличие от других семейств, представленных в предыдущих главах, семейство графовых моделей содержит алгоритмы, работа которых основана на обходе узлов графа. Обрабатывая локальный узел, можно использовать информацию из соседних узлов, выполняя переходы по ребрам, связывающим любые два узла. Такие алгоритмы помогают определить связанные узлы, то, как образованы связи между ними, и выявить наиболее важные узлы сети. Обход узлов графа позволяет извлекать значимую информацию из документов, не требуя большого объема предварительных знаний и представлений об объектной модели.

Графы способны хранить сложные семантические представления в компактной форме. Поэтому моделирование данных в форме сетей связанных

сущностей является мощным механизмом как для визуального анализа, так и для машинного обучения. Отчасти эта мощь обусловлена высокой производительностью алгоритмов обработки графовых структур данных, а отчасти — врожденной человеческой способностью интуитивно взаимодействовать с небольшими сетями.

Анализ графов занимает важное место; приложения анализа текстовых данных делятся по степени простоты интерпретации их результатов — если пользователь не понимает их, глубина и ценность результатов теряют свое значение. Далее, в главе 10, мы продолжим следовать за потоком взаимодействия «человек — машина», начатым в главе 8, и посмотрим, как приложения чат-ботов, основанные на рядовых задачах анализа естественного языка, способны улучшить качество взаимодействия с пользователем.

10 Чат-боты

В этой главе мы познакомимся с одним из самых быстро развивающихся классов приложений обработки естественного языка — диалоговыми агентами. Диалоговые агенты, от Slackbot до Алексы и Dragon Drive компании BMW, быстро становятся неотъемлемой частью нашей повседневной жизни, внедряясь в самые разные ее аспекты. Они улучшают нашу жизнь, расширяя память (например, выполняя поиск в интернете), ускоряя вычисления (например, осуществляя преобразования или прокладывая маршруты) и обеспечивая более гибкую связь и управление (например, посылая сообщения и управляя системами умного дома).

Главным отличием таких агентов является не информация и не помощь, предоставляемая ими (как в давно существующих веб- и мобильных приложениях с интерфейсом «укажи и щелкни»), а интерфейс, делающий их такими привлекательными. Взаимодействие на естественном языке обеспечивает более гладкий, естественный и простой способ доступа к вычислительным ресурсам. По этой причине чат-боты являются важным шагом в развитии пользовательского интерфейса, позволяя встраивать в текстовые приложения команды на естественном языке и тем самым уменьшая неудобство интерфейсов на основе меню. Важно отметить, что они также открывают возможность взаимодействия человека с машиной в новых вычислительных контекстах, например, с экранными устройствами, плохо приспособленными для ввода, такими как автомобильные навигаторы.

Но почему бурное развитие протекает именно сейчас, с учетом долгой истории диалоговых агентов в реальности (начиная с ранних версий Eliza и PARRY) и в фантастических произведениях («Компьютер» из произведения *Star Trek* («Звездный путь») или «Hal» из *2001: A Space Odyssey* («2001 г.: Космическая одиссея»))? Отчасти потому, что для работы «сногсшибательных приложений»

с таким интерфейсом требуется повсеместное проникновение вычислительных устройств, которое стало возможно с современным интернетом. Но самое главное, потому, что современные диалоговые агенты опираются на обширные массивы пользовательских данных, расширяющие их *возможности*, что, в свою очередь, увеличивает их ценность для нас. Мобильные устройства используют данные GPS, чтобы узнать наше местоположение и предложить уместные рекомендации; игровые консоли адаптируют течение игры, опираясь на количество игроков, которых они могут видеть и слышать. Чтобы эффективно справляться со стоящими перед ними задачами, такие приложения должны не только обрабатывать естественный язык, но и поддерживать внутреннее *состояние*, запоминать информацию, получаемую от пользователя, и ситуационный контекст.

В этой главе мы предложим вашему вниманию фреймворк для создания чатботов, обеспечивающий хранение состояния и использующий это состояние для поддержания осмысленного диалога в определенном контексте. Мы продемонстрируем этот фреймворк на примере создания кулинарного помощника, который поприветствует нового пользователя, выполняет преобразование между единицами измерения и советует хороший рецепт. Работая над этим прототипом, мы реализуем три функции: систему правил, анализирующую команды с помощью регулярных выражений; систему «вопрос-ответ», использующую предварительно обученные синтаксические парсеры для фильтрации входящих вопросов и выбора правильных ответов; и рекомендательную систему, использующую метод машинного обучения без учителя для определения подходящих рекомендаций.

Основы диалогового взаимодействия

В 1940-х годах Клод Шеннон (Claude Shannon) и Уоррен Уивер (Warren Weaver), пионеры в области теории информации и машинного перевода, разработали модель взаимодействия настолько мощную, что она и поныне используется для анализа диалогов[1]. Согласно их модели, связь сводится к последовательности операций кодирования и преобразования по мере прохождения сообщений через каналы связи с разными уровнями шумов и энтропии.

Современное понятие *диалога*, как показано на рис. 10.1, расширяет модель Шеннона — Уивера, где две (или больше) стороны отвечают на сообщения друг друга. Диалог протекает в течение некоторого времени и обычно имеет

[1] Claude Shannon, *A Mathematical Theory of Communication* (1948), http://bit.ly/2JJnVjd (Перевод на русский язык можно найти в сборнике статей «Работы по теории информации и кибернетике», 1963, Издательство иностранной литературы, с. 243–332. — *Примеч. пер.*).

фиксированную продолжительность. Во время диалога участник может либо слушать, либо говорить. Для эффективного диалога необходимо, чтобы в каждый конкретный момент времени говорил только один участник, а остальные слушали. Наконец, упорядоченный во времени диалог должен быть согласованным, чтобы каждое утверждение имело смысл с учетом предыдущих утверждений.

Для вас, как для человека, такое описание диалога, вероятно, покажется очевидным и естественным, но оно имеет важные следствия для вычислительных устройств (представьте, насколько искаженным и запутанным получится диалог, если игнорировать любое из требований). Один из простых способов удовлетворить требования к диалогам — переключать режим вещания и прослушивания каждого участника по очереди. На каждом этапе *инициатива* вещания передается следующему участнику, который решает, куда повернуть диалог, опираясь на сказанное в предыдущем этапе. Поочередная передача инициативы помогает поддержать диалог, пока кто-то из участников не решит завершить его. В результате беседа получится согласованной и будет соответствовать всем требованиям, описанным выше.

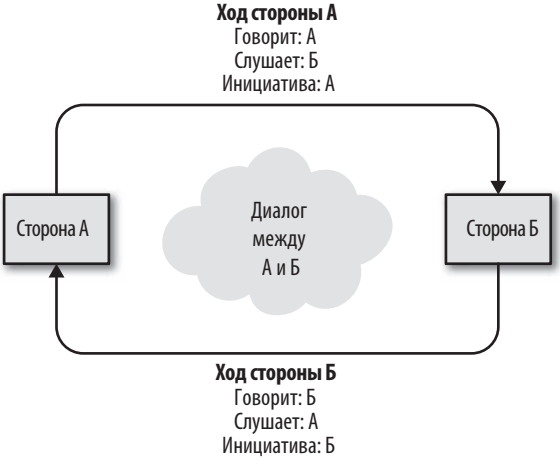


Рис. 10.1. Структура диалога

Чат-бот — это программа, участвующая в диалоге с поочередной передачей инициативы, целью которой является интерпретация входного текста или речи и вывод соответствующего полезного ответа. В отличие от людей, с которыми они взаимодействуют, для достижения этих результатов чат-боты должны полагаться на эвристики и методы машинного обучения. По этой причине им необходимы средства борьбы с неоднозначностью естественного языка и опре-

деления ситуационного контекста, чтобы эффективно анализировать входящие сообщения и выдавать осмысленные ответы.

Архитектура чат-бота, изображенная на рис. 10.2, состоит из двух основных компонентов. Первый — интерфейс с пользователем, осуществляющий прием ввода пользователя (например, через микрофоны для ввода речи или через веб-интерфейс для ввода текста) и передачу интерпретируемого вывода (через динамики для вывода синтезированной речи или через экран мобильного устройства для вывода текста). Этот внешний компонент обертывает второй — внутреннюю *диалоговую систему*, которая интерпретирует входной текст, управляет внутренним состоянием и генерирует ответы.

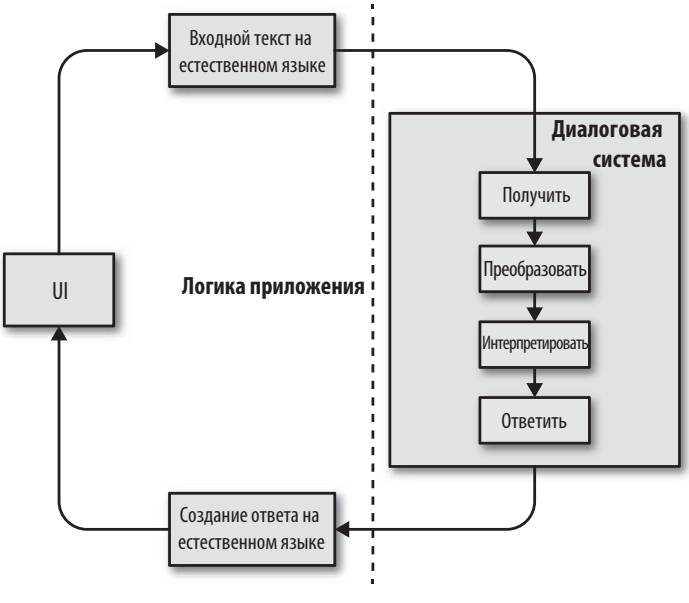


Рис. 10.2. Архитектура чат-бота

Внешний компонент пользовательского интерфейса, очевидно, может существенно отличаться в зависимости от особенностей и требований приложения. В этой главе мы сосредоточимся на внутреннем диалоговом компоненте и покажем, как легко обобщить его для любых приложений и сконструировать из множества поддиалогов. С этой целью мы сначала определим базовый абстрактный класс, который формально определяет поведение, или интерфейс диалога. Затем исследуем три реализации этого класса для управления состоянием, поиска ответов на вопросы и выбора рекомендаций и покажем, как объединить их в один диалоговый агент.

Диалог: непродолжительный обмен

Чтобы создать обобщенную и настраиваемую диалоговую систему, сначала нужно определить минимальную единицу работы, выполняемой во время взаимодействия чат-бота с пользователем. Исходя из архитектуры, описанной в предыдущем разделе, мы знаем, что минимальная единица работы включает прием входного и создание выходного текста на естественном языке. В процессе диалога должно участвовать множество разных механизмов анализа и создания ответа, поэтому будем рассматривать диалоговый агент как состоящий из множества внутренних диалогов, у каждого из которых своя область ответственности.

Чтобы гарантировать согласованную работу внутренних диалогов, мы должны описать единый интерфейс, определяющий порядок действий. В языке Python отсутствует формальный тип интерфейса, зато можно определить *абстрактный базовый класс*, использовав модуль `abc` из стандартной библиотеки для перечисления методов и их сигнатур, которые должны быть реализованы во всех подклассах (если подкласс не реализует какой-то из абстрактных методов, обращение к нему во время выполнения приведет к исключению). Благодаря этому можно гарантировать, что все подклассы нашего интерфейса `Dialog` будут действовать в точности так, как ожидается.

В общем случае `Dialog` отвечает за прием высказываний, парсинг текста, интерпретацию результатов парсинга, изменение внутреннего состояния и формулирование осмысленного ответа. Поскольку предполагается, что иногда система будет неправильно интерпретировать входной текст, объекты `Dialog` должны также определять оценку релевантности в месте ответа, чтобы можно было количественно оценить, насколько правильно интерпретировано входящее высказывание. Разделим определение поведения интерфейса на несколько методов, которые должны быть реализованы в подклассах. Но для начала создадим неабстрактный метод `listen`, основную точку входа в объекты `Dialog`, реализующий обобщенное поведение диалога с использованием абстрактных методов (подлежащих реализации в подклассах):

```
import abc

class Dialog(abc.ABC):
    """
    Объект диалога принимает высказывания, выполняет их парсинг
    и интерпретацию, а затем изменяет внутреннее состояние. После этого
    он может сформулировать ответ.
    """

    def listen(self, text, response = True, **kwargs):
        """
```

```
        Принимает текст text высказывания и выполняет его парсинг. Передает
        результат методу interpret для определения ответа. Если ответ
        необходим, вызывается метод respond, генерирующий текст ответа
        на основе последнего поступившего текста и текущего состояния
        объекта Dialog.
        """
        # Парсинг входного текста
        sents = self.parse(text)

        # Интерпретация
        sents, confidence, kwargs = self.interpret(sents, **kwargs)

        # Определение ответа
        if response:
            reply = self.respond(sents, confidence, **kwargs)
        else:
            reply = None

        # Передача инициативы
        return reply, confidence
```

Метод `listen` определяет глобальную реализацию, унифицирующую абстрактную функциональность. Согласно сигнатуре, метод `listen` принимает строку с текстом, а также логический флаг `response`, указывающий, была ли передана инициатива этому объекту и требуется ли вернуть ответ (если передать в аргументе `response` значение `False`, объект `Dialog` просто воспримет высказывание и обновит свое внутреннее состояние). Кроме того, метод `listen` принимает произвольное количество именованных аргументов (`kwargs`), в которых может передаваться дополнительная контекстная информация, такая как имя пользователя, идентификатор сеанса или оценка транскрипции.

Результатом этого метода является ответ, если он необходим (`None`, если нет), а также оценка *достоверности*, вещественное число в диапазоне между 0,0 и 1,0. Поскольку нет полной уверенности в успехе парсинга и интерпретации входного текста или в правильной формулировке ответа, эта оценка выражает достоверность интерпретации, выполненной объектом `Dialog`, где `1` означает абсолютную достоверность, а `0` — полную неуверенность в достоверности. Достоверность можно вычислить или изменить в любой момент в методе `Dialog.listen`, который, как мы определили, выполняет три абстрактных шага: `parse`, `interpret` и `respond`, хотя вообще достоверность определяется на этапе `interpret`:

```
    @abc.abstractmethod
    def parse(self, text):
        """
        Каждый диалог может реализовать свою стратегию парсинга, некоторые
        с использованием внешних механизмов, другие - с применением
        регулярных выражений или стандартных парсеров.
```

```
        """
        return []
```

Метод `parse` позволяет подклассам `Dialog` реализовать свои алгоритмы обработки строк исходных данных. Например, некоторые подклассы `Dialog` могут использовать внешние механизмы, а другие — применять регулярные выражения или стандартные парсеры. Абстрактный метод определяет сигнатуру конкретной реализации: подкласс должен реализовать метод `parse`, принимающий строку и возвращающий список структур данных, соответствующих конкретному поведению подкласса `Dialog`. В идеале мы включили бы также оптимизацию, чтобы дорогостоящий парсинг выполнялся только один раз и одна и та же работа не выполнялась дважды:

```
    @abc.abstractmethod
    def interpret(self, sents, **kwargs):
        """
        Интерпретирует высказывание, передаваемое как список предложений,
        прошедших парсинг, обновляет внутреннее состояние диалога, вычисляет
        оценку достоверности интерпретации. Также может возвращать аргументы,
        необходимые конкретному механизму создания ответа.
        """
        return sents, 0.0, kwargs
```

Метод `interpret` отвечает за интерпретацию полученного списка предложений, обновление внутреннего состояния объекта `Dialog` и вычисление оценки достоверности интерпретации. Возвращает интерпретированные предложения, которые фильтруются в зависимости от необходимости ответа, а также оценку достоверности в диапазоне между 0 и 1. Далее в этой главе мы рассмотрим несколько алгоритмов определения достоверности. Метод `interpret` может также принимать произвольное количество именованных аргументов и возвращать обновленные именованные аргументы для влияния на поведение метода `respond`:

```
    @abc.abstractmethod
    def respond(self, sents, confidence, **kwargs):
        """
        Создает ответ, опираясь на входящие высказывания и текущее состояние
        диалога, а также на любые именованные аргументы, переданные методами
        listen и interpret.
        """
        return None
```

Наконец, метод `respond` принимает интерпретированные предложения, оценку достоверности и произвольное количество именованных аргументов и генерирует текст ответа, опираясь на текущее состояние объекта `Dialog`. Оценка достоверности может влиять на результат, возвращаемый методом `respond`;

например, если оценка достоверности равна `0.0`, метод может вернуть `None` или запрос на пояснения. Если достоверность оценивается не очень высоко, ответ может включать предположения или приблизительные формулировки, а при высокой достоверности — содержать строгие и точные формулировки.

Теперь в форме абстрактного базового класса `Dialog` у нас есть основа, позволяющая управлять состоянием диалога в коротких взаимодействиях с пользователем. Объект `Dialog` будет служить основным строительным блоком для всех остальных компонентов, которые мы реализуем в оставшейся части главы.

Управление диалогом

Класс `Dialog` определяет порядок обработки простых и непродолжительных взаимодействий и является основным строительным блоком для диалоговых агентов. Но как управлять состоянием в ходе длительных взаимодействий, когда инициатива может неоднократно передаваться от пользователя системе и обратно, и требуется генерировать разные типы ответов?

Ответом на этот вопрос является класс `Conversation`, специализированный диалог, содержащий множество внутренних диалогов. В контексте чат-бота экземпляр `Conversation`, по сути, является оберткой вокруг внутренних диалоговых компонентов, описываемых нашей архитектурой. Объект `Conversation` содержит один или несколько отдельных подклассов `Dialog`, каждый из которых имеет свое внутреннее состояние и поддерживает разные виды интерпретаций и ответов. Когда `Conversation` находится в состоянии приема, он передает входной текст внутренним диалогам и возвращает ответ с наибольшей оценкой достоверности.

В этом разделе мы реализуем класс `SimpleConversation`, наследующий предыдущий класс `Dialog`. Основная задача класса `SimpleConversation` — управлять состоянием последовательности диалогов, хранящихся во внутреннем атрибуте. Этот класс также наследует класс `collections.abc.Sequence` из стандартной библиотеки, который позволит классу `SimpleConversation` действовать подобно индексируемому списку диалогов (благодаря абстрактному методу `__getitem__`) и получать количество диалогов в коллекции (методом `__len__`):

```
from collections.abc import Sequence

class SimpleConversation(Dialog, Sequence):
    """
    Простейшая версия реализации беседы.
    """

    def __init__(self, dialogs):
        self._dialogs = dialogs
```

```
    def __getitem__(self, idx):
        return self._dialogs[idx]

    def __len__(self):
        return len(self._dialogs)
```

В методе `Conversation.listen` мы сделаем еще один шаг вперед и передадим входной текст методам `listen` всех внутренних экземпляров `Dialog`, которые, в свою очередь, вызовут внутренние методы `parse`, `interpret` и `respond`. В результате получится список кортежей `(reply, confidence)`, и `SimpleConversation` просто вернет ответ с наибольшей оценкой достоверности, выбрав кортеж с наибольшим значением во втором элементе с помощью оператора `itemgetter`. Более сложные реализации бесед могут включать дополнительные правила, помогающие сделать выбор, если два внутренних диалога вернут ответы с одинаковой оценкой достоверности, но вообще оптимальной реализацией `Conversation` считается та, в которой такие сходства случаются редко:

```
from operator import itemgetter

...

    def listen(self, text, response = True, **kwargs):
        """
        Просто возвращает ответ с лучшей оценкой достоверности
        """
        responses = [
            dialog.listen(text, response, **kwargs)
            for dialog in self._dialogs
        ]

        # responses -- это список кортежей (reply, confidence)
        return max(responses, key = itemgetter(1))
```

Так как класс `SimpleConversation` наследует `Dialog`, он должен реализовать методы `parse`, `interpret` и `respond`. В каждом из них мы вызовем соответствующие внутренние методы и вернем результаты. Также добавим поддержку оценки достоверности, чтобы можно было вести беседу согласно уверенности в достоверности интерпретации входного текста:

```
...

    def parse(self, text):
        """
        Возвращает все результаты парсинга, полученные внутренними
        диалогами, для отладки
        """
        return [dialog.parse(text) for dialog in self._dialogs]

    def interpret(self, sents, **kwargs):
        """
```

```
        Возвращает все результаты интерпретации, полученные внутренними
        диалогами, для отладки
        """
        return [dialog.interpret(sents, **kwargs) for dialog in self._dialogs]

    def respond(self, sents, confidence, **kwargs):
        """
        Возвращает все ответы, созданные внутренними диалогами, для отладки
        """
        return [
            dialog.respond(sents, confidence, **kwargs)
            for dialog in self._dialogs
        ]
```

Цель фреймворка `Dialog` — обеспечить модульность, чтобы можно было использовать диалоговые компоненты в группе (как в нашем классе `SimpleConversation`) или по отдельности. Наша реализация интерпретирует диалоги как полностью независимые, но есть множество других моделей, например:

Параллельные/асинхронные беседы

Возвращается первый ответ с положительной оценкой достоверности.

Концептуально ориентированные беседы

Диалоги помечаются как «открытые» и «закрытые».

Динамические беседы

Диалоги могут динамически добавляться и удаляться.

Древовидные беседы

Диалоги имеют родителей и потомков.

В следующем разделе мы посмотрим, как использовать класс `Dialog` с некоторыми простыми эвристиками, чтобы создать диалоговую систему, управляющую взаимодействиями с пользователями.

Правила вежливой беседы

В 1950 г. известный информатик и математик Алан Тьюринг (Alan Turing) впервые предложил то, что впоследствии стало известно как тест Тьюринга[1], —

[1] Alan Turing, *Computing Machinery and Intelligence* (1950), http://bit.ly/2GLop6D (Перевод на русский язык: «Может ли машина мыслить?», Государственное издательство физико-математической литературы, Москва, 1960: http://www.etheroneph.com/files/can_the_machine_think.pdf. — *Примеч. пер.*).

способность машины заставить собеседника думать, что он беседует с человеком. Тест Тьюринга дал толчок к развитию в течение следующих нескольких десятилетий множества диалоговых систем, основанных на определенных правилах, многие из которых не только прошли тест, но также сформировали первое поколение диалоговых агентов и стали прародителями современных чат-ботов.

Самым известным, пожалуй, примером является программа ELIZA, созданная Джозефом Вейценбаумом (Joseph Weizenbaum) в 1966 году в Массачусетском технологическом институте (MIT). Она использовала логику выделения значимых слов и фраз из текста, введенного человеком, и возвращала запрограммированные ответы для поддержания беседы. Программа PARRY, созданная Кеннетом Колби (Kenneth Colby) несколько лет спустя в Стэнфордском университете, конструировала ответы, используя ментальную модель в комбинации с алгоритмом сопоставления с образцом. Эта ментальная модель заставляла PARRY становиться все более взволнованной и неустойчивой, чтобы сымитировать пациента, страдающего параноидальной шизофренией, и успешно обманывала врачей, заставляя их думать, что они беседуют с настоящим пациентом.

В этом разделе мы реализуем функцию приветствия, основанную на наборе правил и в духе упомянутых ранних моделей, которая использует регулярные выражения для сопоставления высказываний. Наша версия будет использовать внутреннее состояние в основном для распознавания собеседников, вступающих в диалог и покидающих его, и отвечать им соответствующими приветствиями и вопросами. Эта реализация `Dialog` призвана подчеркнуть важность внутреннего состояния для отслеживания течения диалога, а также показать эффективность чат-ботов на основе регулярных выражений. В заключение мы покажем, как с помощью фреймворка тестирования можно проверить работу компонента `Dialog` в разных ситуациях, чтобы сделать его более устойчивым к разнообразию пользовательского ввода.

Приветствие и прощание

Диалог `Greeting` служит основой реализации нашего диалогового фреймворка и наследует базовый класс `Dialog`. Он запоминает участников, вступающих в беседу и покидающих ее, а также возвращает соответствующие приветствия и прощальные слова, когда кто-то вступает в беседу или покидает ее. Это достигается сопоставлением активных пользователей и их имен.

В основе диалога `Greeting` лежит словарь `PATTERNS`, хранимый как переменная класса. Этот словарь отображает виды взаимодействий (описываются ключами) в регулярные выражения, определяющие ожидаемый ввод для этих взаимо-

действий. В частности, наш простой диалог `Greeting` поддерживает начальное приветствие, знакомство, прощание и перекличку. Эти регулярные выражения мы используем позднее в методе `parse`:

```
class Greeting(Dialog):
    """
    Запоминает участников, вступающих в беседу и покидающих ее, и отвечает
    соответствующими приветствием и прощанием. Это пример системы,
    основанной на правилах, которая управляет своим внутренним состоянием
    и использует регулярные выражения и логику для поддержания диалога.
    """

    PATTERNS = {
        'greeting': r'hello|hi|hey|good morning|good evening',
        'introduction': r'my name is ([a-z\-\s]+)',
        'goodbye': r'goodbye|bye|ttyl',
        'rollcall': r'roll call|who\'s here?',
    }

    def __init__(self, participants = None):
        # Participants - это словарь, отображающий имя пользователя
        # в его настоящее имя
        self.participants = {}

        if participants is not None:
            for participant in participants:
                self.participants[participant] = None

        # Скомпилировать регулярные выражения
        self._patterns = {
            key: re.compile(pattern, re.I)
            for key, pattern in self.PATTERNS.items()
        }
```

При создании экземпляр `Greeting` можно инициализировать предварительно подготовленным списком участников. Внутреннее состояние диалога — это словарь, отображающий имена пользователей в их настоящие имена. Его заполнение мы рассмотрим ниже, когда приступим к интерпретации знакомства. Чтобы ускорить парсинг текста в процессе общения, мы добавили в инициализацию компиляцию регулярных выражений и их сохранение во внутреннем словаре экземпляра. В результате компиляции регулярных выражений получаются объекты регулярных выражений, помогающие сэкономить время на этом шаге при последующем многократном использовании, которое имеет место в данном диалоге.

Эту весьма упрощенную реализацию класса `Greeting` можно расширить дополнительными правилами для поддержки других речевых оборотов и других языков.

Далее реализуем метод `parse`, цель которого — сопоставить текст, введенный пользователем, с каждым скомпилированным регулярным выражением и определить шаблон для приветствия, знакомства, прощания или переклички:

```
def parse(self, text):
    """
    Применяет все регулярные выражения к тексту в поисках совпадения.
    """
    matches = {}
    for key, pattern in self._patterns.items():
        match = pattern.match(text)
        if match is not None:
            matches[key] = match
    return matches
```

Если совпадение найдено, `parse` возвращает его. Этот результат затем можно использовать для передачи методу `interpret`, который по найденному совпадению определяет, какое действие выполнить. Если `interpret` получит результат парсинга, не соответствующий ни одному шаблону, он сразу возвращает оценку достоверности `0.0`. Если совпадение найдено, `interpret` просто возвращает оценку `1.0`, потому что регулярные выражения не поддерживают нечетких совпадений.

Метод `interpret` отвечает за обновление внутреннего состояния диалога `Greeting`. Например, если входной аргумент соответствует знакомству (то есть, если пользователь ввел фразу «my name is *something*» (меня зовут *так-то*)), `interpret` извлечет имя и добавит нового пользователя и его настоящее имя (новое или существующее) в `self.participants`. Если взаимодействие определяется как приветствие, `interpret` проверит присутствие пользователя в словаре `self.participants`. Если такого пользователя нет, добавит значение именованного аргумента в окончательный результат, показав, что в методе `respond` необходимо запросить знакомство. Иначе, если взаимодействие определяется как прощание, он удалит пользователя (если тот известен) из словаря `self.participants` и из именованных аргументов:

```
def interpret(self, sents, **kwargs):
    """
    Принимает найденные совпадения и определяет действие для выполнения:
    приветствие, прощание или изменение имени.
    """
    # В отсутствие совпадения ничего не делать
    if len(sents) == 0:
        return sents, 0.0, kwargs

    # Получить имя пользователя из participants
    user = kwargs.get('user', None)
```

```
        # Если выполняется знакомство
        if 'introduction' in sents:
            # Извлечь имя из высказывания
            name = sents['introduction'].groups()[0]
            user = user or name.lower()

            # Определить необходимость изменения имени
            if user not in self.participants or self.participants[user]
               ! = name:
                kwargs['name_changed'] = True

            # Изменить словарь participants
            self.participants[user] = name
            kwargs['user'] = user

        # Если выполняется приветствие
        if 'greeting' in sents:
            # Если имя пользователя неизвестно
            if not self.participants.get(user, None):
                kwargs['request_introduction'] = True

        # Если выполняется прощание
        if 'goodbye' in sents and user is not None:
            # Удалить участника из словаря
            self.participants.pop(user)
            kwargs.pop('user', None)

        # Если найдено то, что искалось, вернуть максимальную оценку
        # достоверности
        return sents, 1.0, kwargs
```

Наконец, метод `respond` будет определять, что должен ответить наш чат-бот пользователю. Если оценка достоверности равна `0.0`, в ответ ничего не выводится. Если пользователь послал приветствие или представился, `respond` вернет текст с просьбой представиться или с приветствием известного ему пользователя. В случае прощания `respond` вернет обезличенное пожелание встретиться позже. Если пользователь спросил, кто еще участвует в беседе, `respond` получит список участников и вернет их имена (если другие участники имеются) или только имя данного пользователя (если кроме него никого нет). Если в данный момент в `self.participants` не зафиксировано ни одного участника, чат-бот выразит нетерпение:

```
    def respond(self, sents, confidence, **kwargs):
        """
        Выводит приветствие или прощание, в зависимости от ситуации.
        """
        if confidence == 0:
            return None

        name = self.participants.get(kwargs.get('user', None), None)
```

```
        name_changed = kwargs.get('name_changed', False)
        request_introduction = kwargs.get('request_introduction', False)

        if 'greeting' in sents or 'introduction' in sents:
            if request_introduction:
                return "Hello, what is your name?"
            else:
                return "Hello, {}!".format(name)

        if 'goodbye' in sents:
            return "Talk to you later!"

        if 'rollcall' in sents:
            people = list(self.participants.values())

            if len(people) > 1:
                roster = ", ".join(people[:-1])
                roster + = " and {}.".format(people[-1])
                return "Currently in the conversation are " + roster

            elif len(people) == 1:
                return "It's just you and me right now, {}.".format(name)
            else:
                return "So lonely in here by myself ... wait who is that?"

        raise Exception(
            "expected response to be returned, but could not find rule"
        )
```

Обратите внимание, что в обоих методах, `interpret` и `respond`, мы используем простую логику ветвления для обработки разных типов ввода. По мере увеличения класса может быть полезно разбить эти методы на более мелкие составляющие, такие как `interpret_goodbye` и `respond_goodbye`, чтобы провести логические границы и предотвратить появление ошибок. Давайте поэкспериментируем с классом `Greeting`, передав ему разный входной текст:

```
if __name__ == '__main__':
    dialog = Greeting()
    # `listen` возвращает кортежи  (reply, confidence) и мы просто выводим
ответы
    print(dialog.listen("Hello!", user = "jakevp321")[0])
    print(dialog.listen("my name is Jake", user = "jakevp321")[0])
    print(dialog.listen("Roll call!", user = "jakevp321")[0])
    print(dialog.listen("Have to go, goodbye!", user = "jakevp321")[0])
```

Вот какие результаты у нас получились:

```
Hello, what is your name?
Hello, Jake!
It's just you and me right now, Jake.
```

Важно отметить, что наша система получилась чересчур жесткой и быстро ломается. Для примера посмотрим, что получится, если опустить аргумент `user` в одном из вызовов метода `Greeting.listen`:

```
if __name__ == '__main__':
    dialog = Greeting()
    print(dialog.listen("hey", user = "jillmonger")[0])
    print(dialog.listen("my name is Jill.", user = "jillmonger")[0])
    print(dialog.listen("who's here?")[0])
```

На этот раз чат-бот распознал приветствие пользователя Jill, попросил представиться и поприветствовал нового участника. Но в третьем вызове `listen` чат-бот не получил аргумента `user` с именем пользователя и не смог назвать его по имени в ответ на перекличку:

```
Hello, what is your name?
Hello, Jill!
It's just you and me right now, None.
```

Системы на основе правил действительно склонны легко ломаться. Прием разработки через тестирование, о котором рассказывается в следующем разделе, может помочь нам предвидеть и упредить проблемы, которые могут возникнуть на практике.

Обработка ошибок при общении

Строгое тестирование помогает предусмотреть способы устранения ошибок, которые могут возникнуть на этапах парсинга и создания ответа. В этом разделе мы используем библиотеку PyTest и с ее помощью протестируем ограничения класса `Greeting`, поэкспериментировав с краевыми случаями, и посмотрим, какие ситуации приводят к нарушениям в общении.

Реализацию чат-бота мы могли бы начать с определения комплекта тестов для базового класса `Dialog`. Ниже показана общая структура класса `TestBaseClasses`, тестирующего подклассы, которые наследуют класс `Dialog` и его метод `listen`.

Наш первый тест `test_dialog_abc` использует декоратор `pytest.mark.parametrize`, позволяющий послать в тест много разных примеров с минимальными усилиями:

```
import pytest

class TestBaseClasses(object):
    """
```

```
    Тестирует класс Dialog
    """
    @pytest.mark.parametrize("text", [
        "Gobbledeguk", "Gibberish", "Wingdings"
    ])
    def test_dialog_abc(self, text):
        """
        Тестирует абстрактный базовый класс Dialog и метод listen
        """
        class SampleDialog(Dialog):
            def parse(self, text):
                return []

            def interpret(self, sents):
                return sents, 0.0, {}

            def respond(self, sents, confidence):
                return None

        sample = SampleDialog()
        reply, confidence = sample.listen(text)
        assert confidence == 0.
        assert reply is None
```

Теперь можно реализовать несколько тестов для нашего класса `Greeting`. Первый тест, `test_greeting_intro`, использует декоратор `parametrize` для проверки разных комбинаций входных строк и имен пользователей, чтобы убедиться, что класс благополучно возвращает `1` в качестве оценки достоверности интерпретации, что `respond` генерирует ожидаемый ответ и чат-бот просит пользователя представиться:

```
class TestGreetingDialog(object):
    """
    Проверяет правильность интерпретации входного текста и создания
    ожидаемого ответа классом Greeting
    """

    @pytest.mark.parametrize("text", ["Hello!", "hello", 'hey', 'hi'])
    @pytest.mark.parametrize("user", [None, "jay"], ids = ["w/ user",
                             "w/o user"])
    def test_greeting_intro(self, user, text):
        """
        Проверяет вывод с просьбой представиться в ответ на начальное
        приветствие
        """
        g = Greeting()
        reply, confidence = g.listen(text, user = user)
        assert confidence == 1.0
        assert reply is not None
        assert reply == "Hello, what is your name?"
```

Если любая из проверок потерпит неудачу, это послужит сигналом, что мы должны переделать класс `Greeting`, чтобы он мог обрабатывать более широкий диапазон возможных вариантов ввода.

Также мы должны определить класс `test_initial_intro`, чтобы проверить, что случится, если пользователь представится до того, как введет приветствие. Поскольку мы заранее знаем, что это приведет к ошибке, используем декоратор `pytest.mark.xfail` для проверки случаев, которые вероятнее всего приведут к ошибке; это поможет нам запомнить краевые случаи для их обработки в будущих версиях:

```
...
    @pytest.mark.xfail(reason = "a case that must be handled")
    @pytest.mark.parametrize("text", ["My name is Jake", "Hello, I'm Jake."])
    @pytest.mark.parametrize("user", [None, "jkm"], ids = ["w/ user",
                             "w/o user"])
    def test_initial_intro(self, user, text):
        """
        Проверяет обработку представления без предварительного приветствия
        """
        g = Greeting()
        reply, confidence = g.listen(text, user = user)

        assert confidence == 1.0
        assert reply is not None
        assert reply == "Hello, Jake!"

        if user is None:
            user = 'jake'

        assert user in g.participants
        assert g.participants[user] == 'Jake'
```

Системы, основанные на правилах, все еще остаются весьма эффективным методом управления состоянием в рамках диалога, особенно когда создаются с применением методики разработки через тестирование и снабжаются тестами, проверяющими надежность обработки краевых случаев. Применение простой комбинации логики и регулярных выражений для управления диалогом (как в ELIZA, PARRY и в нашем классе `Greeting`) может оказаться на удивление эффективным приемом. Однако современные диалоговые агенты редко полагаются только на эвристики. Причины этого вы поймете в следующей части этой главы, где мы начнем интегрировать лингвистические признаки.

Занимательные вопросы

Чат-боты часто применяются для быстрого получения ответов на практические вопросы, такие как: «Какова протяженность реки Нил?» В интернете существует множество баз знаний, таких как DBPedia, Yago2 и Google Knowledge Graph; также есть большое количество мелких баз знаний для конкретных приложений, таких как ответы на часто задаваемые вопросы. Все эти инструменты позволяют получить ответ на поставленный вопрос, и наша задача состоит в том, чтобы преобразовать вопрос на естественном языке в запрос к базе данных. Самый простой способ реализации состоит в статистическом сопоставлении вопросов с ответами, однако в более надежных решениях обычно используется и статистическая, и семантическая информация; например, с применением фреймового подхода к созданию шаблонов для передачи в запросах SPARQL или методов классификации для идентификации типа требуемого ответа (например, местоположение, объем и т. д.).

Однако в диалоговых системах возникает еще одна проблема — идентификация вопросов и определение их типов. Первым делом следует понять, как выглядит вопрос; мы могли бы использовать регулярное выражение для поиска предложений, начинающихся с вопросительных слов («Кто», «Что», «Где», «Почему», «Когда», «Как») и завершающихся вопросительным знаком.

Однако этот подход наверняка приведет к ошибкам, так как не распознает вопросы, начинающиеся с других слов (например, «Вы умеете готовить?», «У вас есть чеснок?»), и предложения, «обертывающие» вопросы (например, «Присоединитесь к нам?»). Кроме того, этот подход может приводить к ложной идентификации предложений, начинающихся с вопросительных слов (например, «Когда в Риме...»), или риторических вопросов, не требующих ответа (например, «А судьи кто?»).

Несмотря на то что вопросы часто задаются необычным, непредсказуемым способом, все же можно выделить некоторые их закономерности. Предложения кодируют глубоко вложенные структуры и связи, которые выходят далеко за рамки простого сопоставления. Для выявления этих закономерностей нам нужен синтаксический анализ некоторого вида или, иными словами, механизм, использующий контекстно-свободные грамматики для определения синтаксической структуры текста.

В разделе «Извлечение ключевых фраз» главы 7 мы видели, что в библиотеке NLTK есть большое количество парсеров, использующих грамматики в своей

работе, но все они требуют, чтобы мы предоставили грамматику, определяющую правила конструирования фраз или словосочетаний из частей речи, что излишне ограничит гибкость нашего чат-бота. В следующих разделах мы подойдем с другой стороны и рассмотрим более гибкие альтернативы: *анализ зависимостей* и *составляющих*.

Анализ зависимостей

Парсеры зависимостей — это легковесный механизм извлечения синтаксической структуры предложения путем связывания фраз конкретными отношениями. С этой целью они сначала определяют главное слово в фразе, а затем выявляют связи с зависимыми словами. Результатом является структура перекрывающихся дуг, определяющих смысловые подструктуры в предложении.

В отличие от библиотеки NLTK, использующей набор тегов Penn Treebank, с которым мы впервые познакомились в разделе «Маркировка частями речи» главы 3, SpaCy использует набор тегов Universal PoS (например, «ADJ» — для обозначения прилагательных (adjective), «ADV» — для наречий (adverb), «PROPN» — для имен собственных (proper noun), «PRON» — для местоимений (pronoun), и т. д.). Теги Universal PoS определены лингвистами (не программистами, как в случае с набором Penn Treebank), а это значит, что данный набор богаче, хотя в нем не допускается, например, использование таких конструкций, как `tag.startswith("N")` для определения существительных (noun).

Рассмотрим предложение «How many teaspoons are in a tablespoon?» (Сколько чайных ложек содержится в столовой ложке?). На рис. 10.3 изображен результат анализа зависимостей с использованием модуля DisplaCy из библиотеки SpaCy (появившегося в версии 2.0). Перед этим парсером стоит цель показать, как слова в предложении связаны между собой и как они влияют друг на друга. Например, ясно видно, что основой этой фразы является глагол (`VERB`) «are» (содержится), который объединяет наречную группу (`ADV,ADJ,NOUN`) «How many teaspoons» (Сколько чайных ложек) с предложной (`ADP,DET,NOUN`) «in a tablespoon» (в столовой ложке) через зависимость головного существительного «teaspoons» (чайных ложек) и зависимость предлога «in» (`ADP` обозначает предлоги).

Рис. 10.3. Дерево зависимостей, построенное с помощью библиотеки SpaCy

Чтобы воссоздать результаты, изображенные на рис. 10.3, сначала загрузим предварительно обученную модель парсинга для английского языка. Затем определим функцию `plot_displacy_tree` для парсинга входящих предложений с применением обученной модели и выведем результаты анализа зависимостей с помощью метода `displacy.serve`. Если выполнить этот код в Jupyter Notebook, диаграмма появится на странице блокнота; если выполнить его в командной строке, диаграмму можно будет увидеть с помощью браузера, открыв страницу http://localhost:5000/:

```
import spacy
from spacy import displacy

# Предварительно необходимо выполнить команду: python -m spacy download en

spacy_nlp = spacy.load("en")

def plot_displacy_tree(sent):
  doc = spacy_nlp(sent)
  displacy.serve(doc, style = 'dep')
```

Анализ зависимостей часто используется для реализации быстрого и правильного грамматического разбора, а в сочетании с маркировкой частями речи делает большую часть из необходимого для анализа на уровне фраз. Отношения между словами, определяемые в ходе анализа зависимостей, также могут пригодиться в синтаксическом анализе. Однако анализ зависимостей не дает достаточно полного представления о структуре предложений, и поэтому не всегда оказывается достаточным. Далее мы покажем, как получить более полное древовидное представление с использованием анализа составляющих.

Анализ составляющих

Анализ составляющих — это одна из форм синтаксического анализа, цель которого состоит в том, чтобы разбить предложение на фразовые структуры, подобные тем, что мы рисовали, когда учились в школе. Результатом этого анализа является древовидная структура, отражающая сложные взаимосвязи между подфразами. Анализ составляющих дает возможность применить алгоритмы обхода деревьев и обсчитать текст, но из-за многообразия естественного языка дерево предложения обычно можно построить несколькими способами.

В результате анализа нашего примера вопроса «How many teaspoons in a tablespoon?» получается структура, изображенная на рис. 10.4. Здесь мы видим гораздо более сложную структуру подфраз и более непосредственные, немаркированные связи между узлами в дереве.

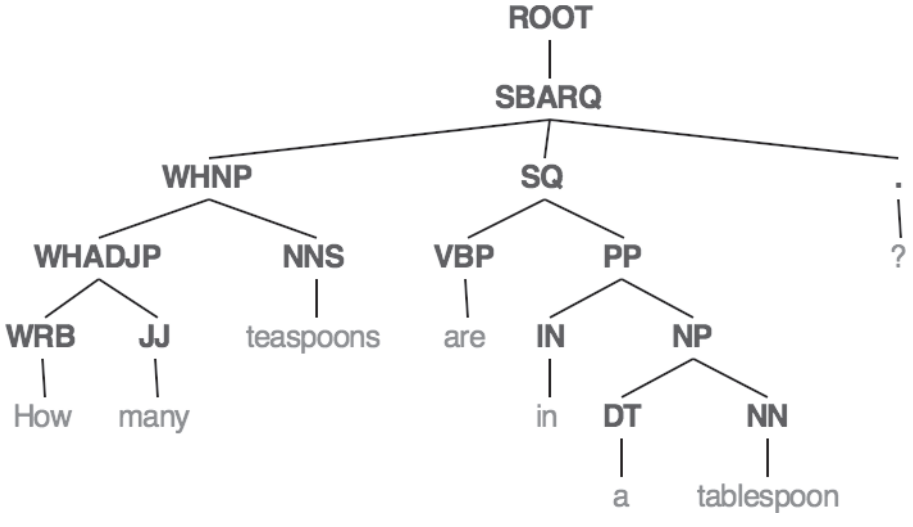


Рис. 10.4. Дерево составляющих, полученное с помощью пакета Stanford CoreNLP

Деревья анализа составляющих образованы из конечных листовых узлов, тегов частей речи и самих слов. Нетерминальные узлы представляют фразы, объединяющие теги частей речи в связанные группы. В нашем вопросе корневой является фраза `SBARQ`, то есть предложение идентифицируется как прямой вопрос из-за вводного вопросительного слова. Это предложение состоит из `WHNP` (именного словосочетания с вопросительным словом) и `SQ`, основного предложения `SBARQ`, которое само является глагольным словосочетанием. Как видите, в этом разборе присутствует множество мелких деталей, которые описывают синтаксические составляющие вопроса!

Теги `WRB`, `WP` и `WDT` способны помочь системе «вопрос-ответ» идентифицировать слова, представляющие особый интерес в нашем конкретном контексте, и выделить вопросы относительно перевода между единицами измерения; `WRB` — это вопросительное наречие (например, вопросительное слово, используемое как модификатор слова, как в вопросе «When are you leaving?» (Когда вы уезжаете?)); `WP` — это вопросительное местоимение (как в вопросе «Who won the bet?» (Кто выиграл?)); и `WDT` — вопросительный определитель (как в вопросе «Which Springfield?» (Какой Спрингфилд?)).

Пакет Stanford CoreNLP — это полный комплект инструментов обработки естественного языка, написанный на Java. Он включает средства для маркировки частями речи, синтаксического анализа, распознавания именованных сущностей, анализа эмоциональной окраски и многого другого.

УСТАНОВКА ЗАВИСИМОСТЕЙ ДЛЯ ПАКЕТА STANFORD CORENLP

Для работы текущая версия пакета требует наличия на компьютере Java 8 (JDK1.8) или более поздней версии.

Загрузите и извлеките инструменты Stanford NLP:

```
wget http://nlp.stanford.edu/software/stanford-ner-2015-04-20.zip
wget http://nlp.stanford.edu/software/stanford-parser-full-2015-04-20.zip
wget http://nlp.stanford.edu/software/stanford-postagger-full-2015-04-20.zip

unzip stanford-ner-2015-04-20.zip
unzip stanford-parser-full-2015-04-20.zip
unzip stanford-postagger-full-2015-04-20.zip
```

Затем добавьте пути к распакованным файлам .jar в переменные окружения. Для этого откройте в редакторе файл `.bash_profile` и добавьте в него следующие строки:

```
export STANFORDTOOLSDIR = $HOME
export CLASSPATH = $STANFORDTOOLSDIR/stanford-postagger-full-2015-04-20/
  stanford-postagger.jar:$STANFORDTOOLSDIR/stanford-ner-2015-04-20/
  stanford-ner.jar:$STANFORDTOOLSDIR/stanford-parser-full-2015-04-20/
  stanford-parser.jar:$STANFORDTOOLSDIR/stanford-parser-full-2015-04-20/
  stanford-parser-3.5.2-models.jar
export STANFORD_MODELS = $STANFORDTOOLSDIR/stanford-postagger-full-2015-04-20/
  models:$STANFORDTOOLSDIR/stanford-ner-2015-04-20/classifiers
```

В недавнем обновлении библиотеки NLTK появился новый модуль `nltk.parse.stanford`, который позволяет использовать парсеры из пакета Stanford внутри NLTK (при условии, что вы установили все необходимые файлы .jar и определили пути к ним в переменных окружения), например:

```
from nltk.parse.stanford import StanfordParser

stanford_parser = StanfordParser(
    model_path = "edu/stanford/nlp/models/lexparser/englishPCFG.ser.gz"
)
def print_stanford_tree(sent):
    """
    Использует предварительно обученную модель из пакета Stanford для
    извлечения дерева зависимостей с целью дальнейшего его использования
    другими методами.
    Возвращает список деревьев
    """
    parse = stanford_parser.raw_parse(sent)
    return list(parse)
```

Нарисовать дерево составляющих, полученное с применением Stanford и изображенное на рис. 10.4, можно с помощью `nltk.tree`. Это позволит нам визуально исследовать структуру вопроса:

```
def plot_stanford_tree(sent):
    """
    Представляет дерево зависимостей Stanford в виде изображения
    """
    parse = stanford_parser.raw_parse(sent)
    tree = list(parse)
    tree[0].draw()
```

Исследуя визуальное отображение результатов синтаксического анализа, полученных с помощью StanfordNLP, можно заметить, что более длинные фразы имеют более сложную структуру. Анализ составляющих дает много информации, которая может оказаться избыточной для некоторых приложений. Оба вида анализа — зависимостей и составляющих — страдают от структурной неоднозначности, в том смысле, что они также могут определять вероятность правильности фразы, которую можно использовать для вычисления оценки достоверности. Однако уровень детализации синтаксиса делает анализ составляющих отличным кандидатом для простой идентификации вопросов и применения фреймов для извлечения запрашиваемой информации, как будет показано в следующем разделе.

Выявление вопроса

Предварительно обученные модели в SpaCy и CoreNLP являются мощными средствами автоматического анализа и аннотирования входящих предложений. Полученные аннотации затем можно использовать для обхода проанализированных предложений и поиска тегов частей речи, соответствующих вопросам.

Сначала проверим тег, присвоенный самому верхнему узлу `ROOT` (нулевой элемент в дереве разбора, куда сходятся все его ветви). Затем проверим теги, присвоенные ветвям и листьям, что можно сделать с помощью метода `tree.pos` из модуля `nltk.tree.Tree`:

```
tree = print_stanford_tree("How many teaspoons are in a tablespoon?")
root = tree[0] # Корень -- первый элемент в дереве разбора предложения
print(root)
print(root.pos())
```

После разбора можно посмотреть, чем различаются разные вопросы, используя теги Penn Treebank, с которым мы впервые познакомились в разделе «Маркировка частями речи» главы 3. В нашем случае мы видим, что корнем в нашем предложении является `SBARQ` (прямой вопрос с вводным вопросительным словом, в данном случае вопросительным наречием `WRB`). Предложение начинается с `WHNP` (вопросительного именного словосочетания), содержащего `WHADJP` (вопросительное прилагательное словосочетание):

```
(ROOT
  (SBARQ
    (WHNP (WHADJP (WRB How) (JJ many)) (NNS teaspoons))
    (SQ (VBP are) (PP (IN in) (NP (DT a) (NN tablespoon))))
    (. ?)))
[('How', 'WRB'), ('many', 'JJ'), ('teaspoons', 'NNS'), ('are', 'VBP'),
('in', 'IN'), ('a', 'DT'), ('tablespoon', 'NN'), ('?', '.')]
```

Основным преимуществом такого способа идентификации вопросов является его гибкость. Например, если изменить наш вопрос так: «Sorry to trouble you, but how many teaspoons are in a tablespoon?» (Простите за беспокойство, но сколько чайных ложек содержится в столовой ложке?), результат получится иным, но признаки вопроса, `WHADJP` и `WRB`, останутся на месте:

```
(ROOT
  (FRAG
    (FRAG
      (ADJP (JJ Sorry))
      (S (VP (TO to) (VP (VB trouble) (NP (PRP you))))))
    (, ,)
    (CC but)
    (SBAR
      (WHADJP (WRB how) (JJ many))
      (S
        (NP (NNS teaspoons))
        (VP (VBP are) (PP (IN in) (NP (DT a) (NN tablespoon))))))
    (. ?)))
[('Sorry', 'JJ'), ('to', 'TO'), ('trouble', 'VB'), ('you', 'PRP'), (',', ','),
('but', 'CC'), ('how', 'WRB'), ('many', 'JJ'), ('teaspoons', 'NNS'),
('are', 'VBP'), ('in', 'IN'), ('a', 'DT'), ('tablespoon', 'NN'), ('?', '.')]
```

В табл. 10.1 перечислены некоторые теги, которые мы считаем наиболее полезными для идентификации вопросов. Полный список можно найти в книге Энн Бис (Ann Bies) и других. «Bracketing Guidelines for Treebank II Style»[1].

Таблица 10.1. Теги из Penn Treebank II для идентификации вопросов

Тег	Значение	Пример
SBARQ	Прямой вопрос с вводным вопросительным словом или словосочетанием	«How hot is the oven?» (Насколько горяча печь?)
SBAR	Предложение с подчинительным союзом (например, косвенный вопрос)	«If you're in town, try the beignets». (Если вы в городе, попробуйте бенье.)

[1] Ann Bies, Mark Ferguson, Karen Katz, and Robert MacIntyre, *Bracketing Guidelines for Treebank II Style: Penn Treebank Project* (1995), http://bit.ly/2GQKZLm

Таблица 10.1 (*окончание*)

Тег	Значение	Пример
SINV	Перевернутое повествовательное (декларативное) предложение	«Rarely have I eaten better». (Редко где я обедал лучше.)
SQ	Перевернутый вопрос да/нет или главная вопросительная фраза	«Is the gumbo spicy?» (Этот гамбо острый?)
S	Простое повествовательное (декларативное) предложение	«I like jalapenos». (Мне нравится халапеньо.)
WHADJP	Вопросительное прилагательное словосочетание	Словосочетание «How hot» (Насколько горяча) в вопросе «How hot is the oven?» (Насколько горяча печь?)
WHADVP	Вопросительное наречное словосочетание	Словосочетание «Where do» (Где) в вопросе «Where do you keep the chicory?» (Где вы храните цикорий?)
WHNP	Вопросительное именное словосочетание	Словосочетание «Which bakery» (Какая пекарня) в вопросе «Which bakery is best?» (Какая пекарня лучшая?)
WHPP	Вопросительное предложное словосочетание	Словосочетание «on which» (на которой) в предложении «The roux, on which this recipe depends, should not be skipped». (Заправку, на которой основан этот рецепт, нельзя игнорировать.)
WRB	Вопросительное наречие	Слово «How» (Насколько) в вопросе «How hot is the oven?» (Насколько горяча печь?)
WDT	Вопросительный определитель	Слово «What» (Какая) в вопросе «What temperature is it?» (Какая сейчас температура?)
WP$	Притяжательное вопросительное местоимение	Слово «Whose» (Чья) в вопросе «Whose bowl is this?» (Чья это чашка?)
WP	Вопросительное местоимение	Слово «Who» (Кто) в вопросе «Who's hungry?» (Кто голоден?)

В следующем разделе мы посмотрим, как с помощью этих тегов можно идентифицировать вопросы, наиболее подходящие для виртуального кулинара.

От столовых ложек к граммам

Далее мы добавим в наш чат-бот систему «вопрос-ответ», использующую предварительно обученные модели, с которыми мы познакомились в предыдущем

разделе, для поддержания диалогов на кулинарную тему, касающихся перевода единиц измерения. В повседневной жизни люди часто формулируют вопросы для перевода между единицами измерения, начиная их со слова «How» (Сколько) — например «How many teaspoons are in a tablespoon?» (Сколько чайных ложек содержится в столовой ложке?) или «How many cups make a liter?» (Сколько стаканов в литре?). Для нашей задачи идентификации типа вопроса мы будем стремиться интерпретировать вопросы вида «How many *X* are in a *Y*?» (Сколько *X* в *Y*?).

Начнем с определения класса `Converter`, наследующего наш класс `Dialog`. Предполагается, что экземпляры `Converter` будут инициализироваться базой знаний о преобразованиях единиц измерений. Это простой файл JSON, хранящийся в пути `CONVERSION_PATH` и описывающий преобразования между всеми единицами измерения. В момент инициализации эти преобразования будут загружаться с помощью `json.load`. Кроме того, мы инициализируем парсер (в данном случае будет использоваться CoreNLP), а также `stemmer` из NLTK и `inflect.engine` из библиотеки `inflect`, с помощью которых будем обрабатывать множественное число в методах `parse` и `respond` соответственно. Метод `parse` будет использовать метод `raw_parse` из CoreNLP, чтобы получить дерево анализа составляющих, как было показано в предыдущем разделе:

```
import os
import json
import inflect

from nltk.stem.snowball import SnowballStemmer
from nltk.parse.stanford import StanfordParser

class Converter(Dialog):
    """
    Система вопрос-ответ для преобразования единиц измерения
    """

    def __init__(self, conversion_path = CONVERSION_PATH):
        with open(conversion_path, 'r') as f:
            self.metrics = json.load(f)

        self.inflect = inflect.engine()
        self.stemmer = SnowballStemmer('english')
        self.parser = StanfordParser(model_path = STANFORD_PATH)

    def parse(self, text):
        parse = self.parser.raw_parse(text)
        return list(parse)
```

Далее, в методе `interpret` инициализируем список единиц измерения, для которых будут поддерживаться преобразования, начальную оценку достоверности

числом `0` и словарь для сбора результатов интерпретации. Извлечем корень дерева анализа предложения и используем метод `nltk.tree.Tree.pos`, чтобы отыскать теги частей речи, соответствующие вопросительному наречию (`WRB`). Если такие теги присутствуют, увеличим оценку достоверности и начнем обход дерева с использованием метода поиска `nltk.util.breadth_first`, выбрав максимальную глубину 8 (чтобы ограничить глубину рекурсии). В любых поддеревьях, соответствующих синтаксическому шаблону вопросов типа «сколько», определим и сохраним существительные единственного или множественного числа, представляющие начальную и конечную единицы измерения. Если в поддереве обнаружатся какие-либо числа, сохраним их в словаре `results` с ключом `quantity` для целевой единицы измерения.

Для простоты используем простой, но эффективный механизм вычисления достоверности; возможно, с учетом вашего контекста целесообразнее будет использовать более изощренное решение. Если удалось успешно идентифицировать исходную и конечную единицу измерения, снова увеличим оценку достоверности `confidence` и добавим эти единицы в словарь `results`. Если единицы измерения присутствуют в базе знаний (то есть в файле JSON), снова увеличим оценку достоверности `confidence`. Наконец, вернем кортеж (`results`, `confidence`, `kwargs`), который метод `respond` будет использовать для формирования ответа пользователю:

```
from nltk.tree import Tree
from nltk.util import breadth_first

...

    def interpret(self, sents, **kwargs):
        measures = []
        confidence = 0
        results = dict()

        # Корень -- первый элемент в дереве разбора предложения
        root = sents[0]

        # Проверить наличие вопросительных наречий
        if "WRB" in [tag for word, tag in root.pos()]:
            # Если есть, увеличить confidence и выполнить обход дерева
            confidence + = .2

            # Ограничить рекурсию передачей аргумента maxdepth
            for clause in breadth_first(root, maxdepth = 8):
                # Найти простые повествовательные предложения (+S+)
                if isinstance(clause, Tree):
                if clause.label() in ["S", "SQ", "WHNP"]:
                    for token,tag in clause.pos():
                        # Сохранить существительные как целевые единицы
                        # измерения
```

```
                    if tag in ["NN", "NNS"]:
                        measures.append(token)
                    # Сохранить числа как целевые величины
                    elif tag in ["CD"]:
                        results["quantity"] = token

        # Обработать повторения для всех вложенных деревьев
        measures = list(set([self.stemmer.stem(mnt) for mnt in measures]))

        # Если обе единицы измерения заданы...
        if len(measures) == 2:
            confidence + = .4
            results["src"] = measures[0]
            results["dst"] = measures[1]

            # Проверить их присутствие в нашей таблице поиска
            if results["src"] in self.metrics.keys():
                confidence + = .2
                if results["dst"] in self.metrics[results["src"]]):
                    confidence + = .2

    return results, confidence, kwargs
```

Но, прежде чем перейти к реализации метода `respond`, добавим несколько вспомогательных утилит. Первая из них, `convert`, выполняет преобразование между исходной и целевой единицами измерения. Метод `convert` принимает строки с названиями исходной (`src`) и целевой (`dst`) единиц измерений, количество исходных единиц измерения, которое может быть вещественным или целым числом, и возвращает кортеж с единицами измерения (`converted`, `source`, `target`):

```
def convert(self, src, dst, quantity = 1.0):
    """
    Преобразует заданное количество исходных единиц измерения в целевые.
    """
    # Выполнить стемминг исходной единицы и привести к единственному числу
    src, dst = tuple(map(self.stemmer.stem, (src,dst)))

    # Проверить возможность преобразования
    if dst not in self.metrics:
        raise KeyError("cannot convert to '{}' units".format(src))
    if src not in self.metrics[dst]:
        raise KeyError("cannot convert from {} to '{}'".format(src, dst))

    return self.metrics[dst][src] * float(quantity), src, dst
```

Добавим также методы `round`, `pluralize` и `numericalize`, использующие утилиты из библиотеки `humanize` для преобразования чисел в форму, более понятную человеку:

```
import humanize

...

    def round(self, num):
        num = round(float(num), 4)
        if num.is_integer():
            return int(num)
        return num

    def pluralize(self, noun, num):
        return self.inflect.plural_noun(noun, num)

    def numericalize(self, amt):
        if amt > 100.0 and amt < 1e6:
            return humanize.intcomma(int(amt))
        if amt > = 1e6:
            return humanize.intword(int(amt))
        elif isinstance(amt, int) or amt.is_integer():
            return humanize.apnumber(int(amt))
        else:
            return humanize.fractional(amt)
```

Наконец, в методе `respond` проверим уровень достоверности и, если он достаточно высок, выполним фактическое преобразование между единицами измерения с помощью `convert`, а затем вызовем `round`, `pluralize` и `numericalize`, чтобы привести ответ в удобочитаемый вид:

```
    def respond(self, sents, confidence, **kwargs):
        """
        Использует библиотеки humanize и inflect, чтобы придать ответу
        более удобочитаемый вид.
        """
        if confidence < .5:
            return "I'm sorry, I don't know that one."

        try:
            quantity = sents.get('quantity', 1)
            amount, source, target = self.convert(**sents)

            # Округлить число
            amount = self.round(amount)
            quantity = self.round(quantity)

            # Оформить множественное число
            source = self.pluralize(source, quantity)
            target = self.pluralize(target, amount)
            verb = self.inflect.plural_verb("is", amount)

            # Отформатировать числа
```

```
            quantity = self.numericalize(quantity)
            amount = self.numericalize(amount)

            return "There {} {} {} in {} {}.".format(
                verb, amount, target, quantity, source
            )

        except KeyError as e:
            return "I'm sorry I {}".format(str(e))
```

Теперь можно поэкспериментировать с методом `listen` и попробовать передать ему несколько возможных вопросов, чтобы посмотреть, насколько хорошо наш класс `Converter` справится с разными комбинациями единиц измерения и количественными величинами:

```
if __name__ == "__main__":
    dialog = Converter()
    print(dialog.listen("How many cups are in a gallon?"))
    print(dialog.listen("How many gallons are in 2 cups?"))
    print(dialog.listen("How many tablespoons are in a cup?"))
    print(dialog.listen("How many tablespoons are in 10 cups?"))
    print(dialog.listen("How many tablespoons are in a teaspoon?"))
```

Результаты, имеющие форму кортежей `(reply, confidence)`, показывают, что `Converter` способен успешно выполнять преобразования с высокой степенью достоверности:

```
('There are 16 cups in one gallon.', 1.0)
('There are 32 cups in two gallons.', 1.0)
('There are 16 tablespoons in one cup.', 1.0)
('There are 160 tablespoons in 10 cups.', 1.0)
('There are 1/3 tablespoons in one teaspoon.', 1.0)
```

Обучение для рекомендаций

Поддержка преобразований между единицами измерения, безусловно, удобная функция, но давайте реализуем нечто более уникальное — систему рекомендации рецептов. В этом разделе мы рассмотрим процесс (изображенный на рис. 10.5), получающий корпус рецептов, выполняющий нормализацию текста, векторизацию и свертку размерности пространства признаков, и затем использующий алгоритм ближайших соседей для выбора рекомендуемых рецептов. В данном случае хранимым состоянием нашего варианта `Dialog` будет активная модель машинного обучения, способная выдавать рекомендации и добавлять новые отзывы пользователей, появляющиеся с течением времени.

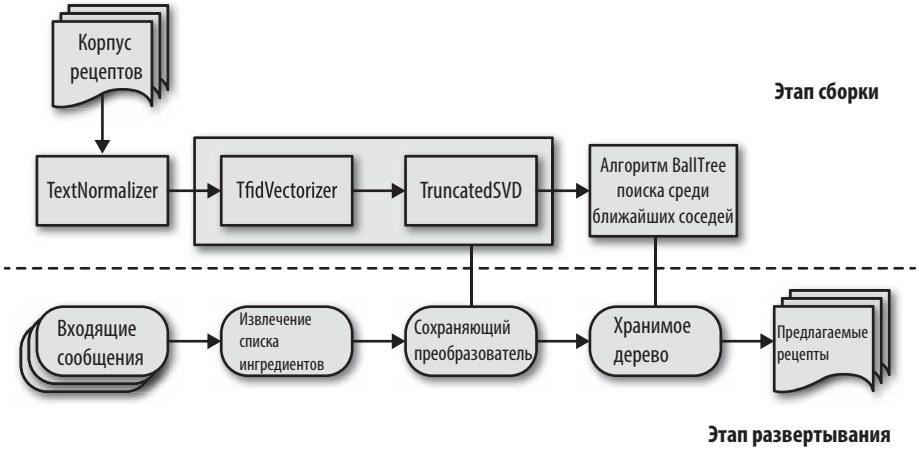


Рис. 10.5. Структурная схема рекомендательной системы

ПРЕДМЕТНО-ОРИЕНТИРОВАННЫЙ КОРПУС

Для обучения рекомендательной системы мы будем использовать корпус с текстами на кулинарную тему, состоящий из сообщений и статей в блогах, содержащих рецепты конкретных блюд, а также рассказы и описания этих блюд. Такой корпус можно создать, извлекая адреса URL из списка кулинарных блогов, перебирая веб-сайты и индексируя их страницы. Наш корпус содержит 60 000 документов HTML общим объемом примерно 8 Гбайт, хранящихся в дереве каталогов с плоской структурой:

```
food_corpus
├── 2010-12-sweet-potato-chili.html
├── 2013-09-oysters-rockefeller.html
├── 2013-07-nc-style-barbeque.html
├── 2013-11-cornbread-stuffing.html
└── 2015-04-next-level-grilled-cheese.html
```

Мы написали класс `HTMLCorpusReader`, напоминающий одноименный класс из раздела «Объекты чтения корпусов» в главе 2, добавили метод `titles()`, который извлекает заголовки HTML-страниц, чтобы иметь возможность дать удобочитаемую ссылку на каждый рецепт:

```
    def titles(self, fileids = None, categories = None):
        """
        Анализирует HTML и выбирает заголовки из тега head.
        """
        for doc in self.docs(fileids, categories):
            soup = bs4.BeautifulSoup(doc, 'lxml')
            try:
```

```
            yield soup.title.text
            soup.decompose()
        except AttributeError as e:
            continue
```

Учитывая размер корпуса, мы реализовали версию класса `Preprocessor`, представленного в главе 3, которая использует Python-библиотеку `multiprocessing` для параллельной обработки корпуса в методе `transform`. В главе 11 мы подробнее обсудим применение этой библиотеки и других способов параллельной обработки.

Соседство

Главный недостаток алгоритма ближайших соседей — сложность поиска с увеличением размерности пространства признаков; чтобы найти *k* ближайших соседей некоторого векторизованного документа `d`, необходимо вычислить расстояние от `d` до каждого другого документа в корпусе. Так как в нашем корпусе содержится порядка 60 000 документов, это означает, что нам придется вычислить 60 000 расстояний и учесть каждое измерение в векторе документа. То есть для пространства признаков с 100 000 измерений, что не так уж необычно для текста, это означает необходимость выполнения 6 миллиардов операций для каждого рецепта!

Очевидно, нам нужно найти способ ускорить поиск, чтобы чат-бот мог быстро возвращать рекомендации. С этой целью мы, во-первых, должны уменьшить размерность пространства признаков. Мы уже делали подобную свертку размерности в классе `TextNormalizer`, который использовали в главах 4 и 5, выполняя лемматизацию и другие «упрощенные» приемы очистки, эффективно снижающие размерность данных. Мы могли бы еще уменьшить размерность с помощью класса `FreqDist` из главы 7 в роли преобразователя, чтобы отобрать только лексемы, составляющие 10–50 % распределения.

Также можно связать наш `TfidfVectorizer` с `TruncatedSVD` из библиотеки Scikit-Learn, который уменьшит размерность наших векторов до нескольких компонентов (для текстовых документов желательно передавать в аргументе `n_components` число не меньше 200). Имейте в виду, что `TruncatedSVD` не центрирует данные перед вычислением сингулярного разложения (Singular Value Decomposition, SVD), из-за чего результаты могут получаться неточными, в зависимости от особенности распределения данных.

Также можно использовать менее дорогостоящую альтернативу традиционному поиску ближайших соседей, например, алгоритмы шаровых деревьев (Ball-

Tree), *k*-мерных деревьев (K-D trees) и хеширования с учетом близости (Local Sensitivity Hashing, LSH). Алгоритм K-D tree (`sklearn.neighbors.KDTree`) — это аппроксимация ближайших соседей, вычисляющая расстояния от заданного экземпляра `d` до ограниченного подмножества экземпляров в наборе. Но метод *K*-мерных деревьев плохо справляется с разреженными данными с большим количеством измерений, потому что использует значения случайных признаков для разделения данных (как вы наверняка помните, в векторном представлении документа большинство признаков будут иметь нулевые значения). Метод хеширования с учетом близости эффективнее справляется с многомерными данными, однако текущая его реализация в Scikit-Learn, `sklearn.neighbors.NearestNeighbors`, признана неэффективной и запланирована для удаления из библиотеки.

Существуют также реализации аппроксимирующих алгоритмов поиска ближайших соседей в других библиотеках, например Annoy (библиотека на C++, для которой есть модуль на Python), которая в настоящее время используется в Spotify для динамического выбора рекомендаций относительно музыкальных произведений.

В нашем чат-боте мы будем использовать алгоритм Ball-Tree. Подобно алгоритму *k*-мерных деревьев, Ball-Tree разбивает экземпляры данных так, что поиск ближайших соседей можно выполнить лишь в ограниченном подмножестве данных (в данном случае во вложенных гиперсферах). Алгоритм Ball-Tree продолжает эффективно работать с увеличением размерности пространства поиска ближайших соседей[1]. Реализация алгоритма в Scikit-Learn, `sklearn.neighbors.BallTree`, поддерживает несколько разных метрик расстояния, которые можно использовать и сравнивать для оптимизации качества результатов.

Наконец, ускорить поиск рекомендуемых рецептов можно за счет сериализации обученного преобразователя (чтобы к входным данным, получаемым от пользователей, можно было применять те же самые преобразования) и дерева, чтобы чат-бот мог выполнять запросы, не перестраивая дерево каждый раз.

Сначала определим класс `BallTreeRecommender`, инициализируемый числом *k* желаемого количества рекомендаций (по умолчанию равно 3), путями к хранимому обученному преобразователю `svd.pkl` и обученному дереву `tree.pkl`. Если модель уже обучена, эти компоненты будут существовать, и метод `load` загрузит их с диска. Иначе они будут обучены и сохранены сериализатором `joblib` из Scikit-Learn в нашем методе `fit_transform`:

[1] Ting Liu, Andrew W. Moore, and Alexander Gray, *New Algorithms for Efficient High-Dimensional Nonparametric Classification* (2006), http://bit.ly/2GQL0io

```
import pickle

from sklearn.externals import joblib
from sklearn.pipeline import Pipeline
from sklearn.neighbors import BallTree
from sklearn.decomposition import TruncatedSVD
from sklearn.feature_extraction.text import TfidfVectorizer

class BallTreeRecommender(BaseEstimator, TransformerMixin):
    """
    Для заданного набора входных терминов возвращает k рекомендованных
    рецептов
    """
    def __init__(self, k = 3, **kwargs):
        self.k = k
        self.trans_path = "svd.pkl"
        self.tree_path = "tree.pkl"
        self.transformer = False
        self.tree = None
        self.load()

    def load(self):
        """
        Загружает хранимый преобразователь и дерево с диска,
        если они существуют.
        """
        if os.path.exists(self.trans_path):
            self.transformer = joblib.load(open(self.trans_path, 'rb'))
            self.tree = joblib.load(open(self.tree_path, 'rb'))
        else:
            self.transformer = False
            self.tree = None

    def save(self):
        """
        Требует много времени на обучение, поэтому должен вызываться только
        один раз!
        """
        joblib.dump(self.transformer, open(self.trans_path, 'wb'))
        joblib.dump(self.tree, open(self.tree_path, 'wb'))

    def fit_transform(self, documents):
        if self.transformer == False:
            self.transformer = Pipeline([
                ('norm', TextNormalizer(minimum = 50, maximum = 200)),
                ('transform', Pipeline([
                    ('tfidf', TfidfVectorizer()),
                    ('svd', TruncatedSVD(n_components = 200))
                ])
            ])
            self.lexicon = self.transformer.fit_transform(documents)
            self.tree = BallTree(self.lexicon)
            self.save()
```

После обучения модели `sklearn.neighbors.BallTree` можно вызвать метод `tree.query`, чтобы получить расстояния до *k* ближайших документов и их индексы. Добавим в наш класс `BallTreeRecommender` метод-обертку `query`, использующий обученный преобразователь для векторизации и преобразования входного текста и возвращающий только индексы ближайших рецептов:

```
    def query(self, terms):
        """
        Для входного списка ингредиентов возвращает k ближайших рецептов.
        """
        vect_doc = self.transformer.named_steps['transform'].fit_
                   transform(terms)
        dists, inds = self.tree.query(vect_doc, k = self.k)
        return inds[0]
```

Предложение рекомендаций

Допустим, что мы обучили `BallTreeRecommender` на нашем корпусе рецептов и сохранили артефакты модели. Теперь мы можем реализовать выборку рекомендаций в контексте абстрактного базового класса `Dialog`.

Экземпляр нашего нового класса `RecipeRecommender` будет инициализироваться объектом чтения корпуса и моделью с методом `query`, такой как наш класс `BallTreeRecommender`. Для получения ссылок на хранимые рецепты по заголовкам сообщений будем использовать метод `corpus.titles()`, упоминавшийся выше во врезке «Предметно-ориентированный корпус». Если экземпляр `RecipeRecommender` еще не обучен, его метод `__init__` выполнит необходимые обучение и преобразование:

```
class RecipeRecommender(Dialog):
    """
    Диалог, возвращающий рекомендации
    """

    def __init__(self, recipes, recommender = BallTreeRecommender(k = 3)):
        self.recipes = list(corpus.titles())
        self.recommender = recommender

        # Обучить модель выбора рекомендаций на корпусе
        self.recommender.fit_transform(list(corpus.docs()))
```

Следующий метод, `parse`, разбивает входную строку на список слов и выполняет маркировку частями речи:

```
    def parse(self, text):
        """
        Извлекает ингредиенты из текста
        """
        return pos_tag(wordpunct_tokenize(text))
```

Наш метод `interpret` принимает результат парсинга текста и определяет, является ли он списком ингредиентов. Если это действительно список ингредиентов, высказывание трансформируется в коллекцию существительных и определяет оценку достоверности как процент числа существительных от общего числа слов в исходном сообщении. И снова мы используем здесь упрощенный способ оценки достоверности в первую очередь ради простоты примера. На практике было бы полезно проверить механизм оценки достоверности; например, предложив экспертам оценить качество аннотированного тестового набора, чтобы убедиться, что низкие оценки достоверности действительно соотносятся с ответами с более низким качеством:

```
def interpret(self, sents, **kwargs):
    # Если это обратная связь, обновить модель
    if 'feedback' in kwargs:
        self.recommender.update(kwargs['feedback'])

    n_nouns = sum(1 for pos, tag in sents if pos.startswith("N"))
    confidence = n_nouns/len(sents)
    terms = [tag for pos, tag in sents if pos.startswith("N")]
    return terms, confidence, kwargs
```

Наконец, метод `respond` принимает список существительных, извлеченных методом `interpret`, и оценку достоверности. Если `interpret` извлек относительно большое количество существительных, оценка достоверности получится достаточно высокой для выбора рекомендаций. Рекомендации извлекаются вызовом внутреннего метода `recommender.query`, которому передается список существительных:

```
def respond(self, terms, confidence, **kwargs):
    """
    Возвращает рекомендации, если оценка достоверности
    confidence > 0.15, иначе возвращается None.
    """
    if confidence < 0.15:
        return None

    output = [
        "Here are some recipes related to {}".format(", ".join(terms))
    ]
    output + = [
        "- {}".format(self.recipes[idx])
        for idx in self.recommender.query(terms)
    ]

    return "\n".join(output)
```

Теперь можно проверить, как работает наш новый класс `RecipeRecommender`:

```
if __name__ == '__main__':
    corpus = HTMLPickledCorpusReader('../food_corpus_proc')
    recommender = RecipeRecommender(corpus)
    question = "What can I make with brie, tomatoes, capers, and pancetta?"
    print(recommender.listen(question))
```

Вот какие результаты получились у нас:

```
('Here are some recipes related to brie, tomatoes, capers, pancetta
- My Trip to Jamaica - Cookies and Cups
- Turkey Bolognese | Well Plated by Erin
- Cranberry Brie Bites', 0.2857142857142857)
```

Как рассказывалось в главе 1, модели машинного обучения выигрывают от наличия обратной связи, которую можно использовать для повышения качества результатов. В контексте кулинарного чат-бота, возвращающего рекомендуемые рецепты, у нас есть прекрасная возможность добавить поддержку обратной связи на естественном языке. Например: мы знаем, что пользователь явно заинтересован, если отвечает сообщением «OK, show me the Cranberry Brie Bites recipe» (Хорошо, покажи мне рецепт бри с клюквой), и, опираясь на эту информацию, можно увеличить ранг сырного деликатеса в результатах, изменив новый компонент вектора, связанный с предпочтениями пользователя.

Как вариант, можно инициировать диалог с пользователем и явно спросить его, что он думает о предложенных рецептах! Чат-боты дают уникальную возможность получить более подробные отзывы, которая отсутствует в современных механизмах clickstream.

В заключение

Благодаря гибким языковым моделям, обученным на предметных корпусах, в сочетании с узкой специализацией чат-боты позволяют людям находить информацию и получать ответы не только быстрее, чем при использовании других средств, но и намного проще. По мере совершенствования механизмов анализа и создания текстов на естественном языке общение с ботом может в конечном итоге оказаться более продуктивным, чем общение с человеком!

В этой главе мы представили основу для создания диалоговых агентов, в центре которой находится абстрактный класс `Dialog`. Объекты `Dialog` получают ввод пользователя, анализируют и интерпретируют его, обновляют внутреннее состояние и затем возвращают ответ, если необходимо, опираясь на обновленное состояние. Класс `Conversation` — это коллекция объектов `Dialog`, позволяющая создавать комбинированные чат-боты, в которых каждый объект `Dialog` отвечает

за интерпретацию конкретного ввода. В этой главе мы представили простую версию класса `Conversation`, который передает входные данные всем внутренним диалогам и возвращает ответ с наивысшей оценкой достоверности.

Фреймворк `Dialog` дает возможность легко создавать диалоговые компоненты и добавлять их в приложения, а также расширять эти компоненты без особых усилий и тестировать их по отдельности. Мы не затронули эту тему в данной главе, но в нашем репозитории GitHub есть примеры реализации общения внутри приложения командной строки или веб-приложения на основе Flask: https://github.com/foxbook/atap/.

Наш кулинарный помощник теперь способен анализировать входной текст, определять типы вопросов и производить преобразования между единицами измерений или рекомендовать рецепты в зависимости от вопроса, заданного пользователем. На следующих этапах развития чат-бота можно было бы добавить возможность ведения беседы, обучив на корпусе с беседами на кулинарную тему языковую модель на основе *n*-грамм, как рассказывалось в главе 7, или ассоциативную модель, с которой мы познакомимся в главе 12. В следующей главе мы исследуем приемы масштабирования, с помощью которых можно создавать более производительные приложения на основе анализа естественного языка, и посмотрим, как их применять для увеличения скорости от сбора корпуса и его предварительной обработки до преобразования и моделирования.

11 Масштабирование анализа текста

Текстовые корпусы, используемые в приложениях данных с поддержкой анализа естественного языка — это нестатические наборы данных, они постоянно растут и изменяются. Возьмем, к примеру, систему «вопрос-ответ»; с нашей точки зрения, это приложение не только возвращает ответы, но также собирает вопросы. То есть даже относительно скромный корпус вопросов может быстро превратиться в обширный ресурс для обучения приложения поиску более точных ответов.

К сожалению, методы обработки текста очень дороги, с точки зрения как пространства (в памяти или на диске), так и времени (потребления вычислительной мощности). Поэтому с увеличением размеров корпуса для его анализа требуется все больше вычислительных ресурсов. Возможно, вы уже заметили, как много времени тратится на обработку корпусов в экспериментах в этой книге! Основное решение проблемы дороговизны обработки больших наборов данных заключается в увеличении вычислительных ресурсов (процессоров, дисков, памяти) для распределения рабочей нагрузки. Когда несколько ресурсов одновременно обрабатывают разные фрагменты набора данных, мы говорим, что они действуют *параллельно*.

Параллелизм (параллельные или распределенные вычисления) имеет две основные формы. Под *параллелизмом на уровне задач* подразумевается одновременное выполнение разных и независимых операций с одними и теми же данными. *Параллелизм на уровне данных* предполагает одновременное применение одной и той же операции сразу к нескольким фрагментам данных. Оба этих вида параллелизма часто используют для ускорения последовательных вычислений (когда операции выполняются друг за другом).

Важно помнить, что для достижения высокой скорости в параллельном окружении часто приходится идти на компромиссы. Большее количество менее емких дисков позволяет читать данные быстрее, чем меньшее количество более емких, но при этом каждый диск будет хранить меньше данных и каждый из них придется читать отдельно. Параллельные вычисления позволяют выполнять задания быстрее, но только если им не приходится ждать завершения друг от друга. На подготовку ресурсов к параллельным вычислениям требуется время, и если это время превышает прирост производительности, тогда параллелизм теряет свою ценность (см. закон Амдала (Amdahl)[1]). Еще одно следствие борьбы за высокую скорость — необходимость использования аппроксимации вместо полноценных вычислений, потому что каждый отдельный ресурс не имеет полного представления входных данных.

В этой главе мы рассмотрим два подхода к организации параллельных вычислений, а также их достоинства и недостатки. Первый, на основе использования модуля `multiprocessing`, позволяет программам использовать многоядерные процессоры и потоки выполнения операционной системы, но ограничивается возможностями компьютера, на котором эти программы выполняются. Второй, на основе фреймворка Spark, дает возможность использовать кластеры, способные масштабироваться до любого размера, но требует организации новых рабочих процессов и поддержки. Цель данной главы — познакомить вас с этими механизмами, чтобы вы могли быстро вовлечь их в процесс анализа текста, а также дать вам достаточный объем информации для принятия обоснованных решений в выборе технологии в каждом конкретном случае.

Модуль multiprocessing

Современные операционные системы способны одновременно выполнять сотни процессов на многоядерных процессорах. Процесс планируется для выполнения на центральном процессоре (CPU) и располагается в своей, изолированной области памяти. Когда запускается программа на Python, операционная система выполняет ее код в отдельном *процессе*. Процессы действуют независимо, и для обмена информацией им необходим некоторый внешний механизм (например, файлы на диске, таблицы в базе данных или сетевые соединения).

Кроме того, единственный процесс может породить несколько *потоков выполнения*. Поток выполнения — это минимальная единица работы для планировщика, представляет собой последовательность инструкций, которые должны

[1] Gene M. Amdahl, *Validity of the single processor approach to achieving large scale computing capabilities* (1967), http://bit.ly/2GQKWza

быть выполнены процессором. Процессами управляет операционная система, но управление потоками выполнения осуществляет сама программа, и обмен данными между потоками в рамках одного процесса может производиться без привлечения любых внешних механизмов.

Современные процессоры имеют несколько ядер, поддерживают конвейерную обработку и другие способы оптимизации выполнения потоков. Языки программирования, такие как C, Java и Go, могут использовать системные потоки выполнения для поддержки конкурентного, а при наличии нескольких ядер в процессоре — и параллельного выполнения в рамках одной программы. К сожалению, Python не может использовать преимущество наличия нескольких ядер из-за глобальной блокировки интерпретатора (Global Interpreter Lock, GIL), которая обеспечивает безопасные интерпретацию и выполнение байт-кода Python. То есть, если программа на Go может добиться 200-процентного использования CPU на компьютере с двухъядерным процессором, то Python не может использовать больше 100 % единственного ядра.

В Python есть дополнительные механизмы поддержки *конкурентного выполнения*, такие как библиотека `asyncio`. Конкурентное и параллельное выполнение имеют много общего, но все же различаются: под параллельным понимается одновременное выполнение вычислений, а под конкурентным — комбинация независимых вычислений, выполнение которых планируется так, чтобы максимально эффективно использовать ресурсы. Обсуждение асинхронного программирования и сопрограмм выходит далеко за рамки этой книги, но они являются мощными механизмами масштабирования анализа текста, особенно для задач, интенсивно использующих операции ввода-вывода, таких как сбор данных или операции с базами данных.

Основная поддержка параллелизма в Python реализована в двух модулях: `threading` и `multiprocessing`. Оба имеют похожий программный интерфейс (в том смысле, что при необходимости вы легко сможете переключаться между ними), но в корне различаются внутренней архитектурой.

Модуль `threading` создает потоки Python. Для таких задач, как сбор данных, которые используют другие системные ресурсы помимо CPU (такие как диск и сетевые соединения), модуль `threading` может оказаться хорошим выбором для поддержки конкурентного выполнения. Но при его использовании в каждый конкретный момент времени будет выполняться только один поток, параллельные вычисления в этом случае просто невозможны.

Для достижения истинного параллелизма в Python необходимо использовать модуль `multiprocessing`. Этот модуль запускает дополнительные дочерние процессы, выполняющие тот же код, что и родительский процесс, либо путем

ветвления (forking) в системах Unix (операционная система создает мгновенный снимок памяти действующего процесса и копирует его в новый процесс), либо путем *порождения* (spawning) в Windows (запускается новый экземпляр интерпретатора Python, которому передается код той же программы). Каждый процесс выполняет свой экземпляр интерпретатора Python со своей глобальной блокировкой GIL, и каждый может задействовать процессор на 100 %. То есть, если запустить четыре процесса в системе с четырехъядерным процессором, можно добиться 400-процентной загрузки CPU.

На рис. 11.1 показана типичная организация параллельного выполнения программы на Python. Здесь есть родительский (или главный) процесс и несколько дочерних процессов (обычно по одному на ядро, хотя их может быть и больше). Родительский процесс подготавливает задания (исходные данные, или ввод) для дочерних процессов и собирает полученные ими результаты (или вывод). Передача данных между потомками и родителем осуществляется с использованием модуля `pickle`. Когда родительский процесс завершается, вместе с ним обычно завершаются и дочерние процессы, впрочем, они могут остаться «осиротевшими» и продолжить работу независимо.

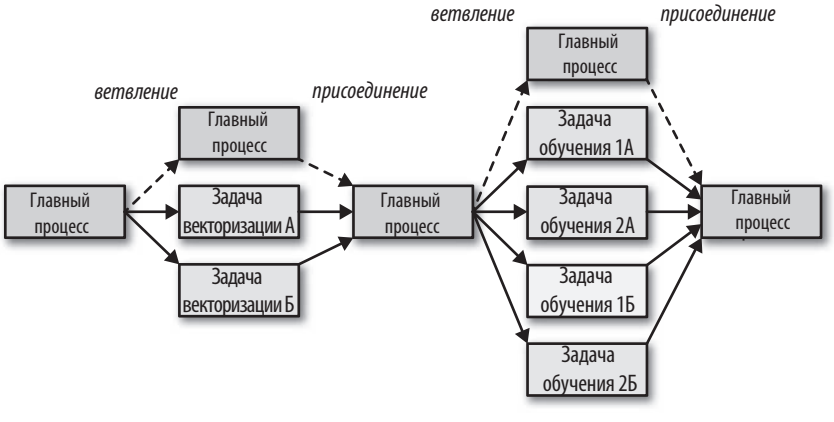


Рис. 11.1. Организация параллельного выполнения задач

На рис. 11.1 изображены две задачи векторизации, выполняющиеся параллельно, и главный процесс, ожидающий их завершения до перехода управления к задачам обучения (например, двух моделей для каждого из двух способов векторизации), которые также выполняются параллельно. Операция *ветвления* порождает несколько дочерних процессов, а операция *присоединения* (joining) заставляет дочерние процессы завершиться и вернуть управление главному процессу. Например, на первом этапе векторизации у нас есть три процесса:

главный и два дочерних, А и Б. Завершив векторизацию, дочерние процессы присоединяются обратно к главному процессу. Параллельная программа на рис. 11.1 запускает шесть параллельных задач, полностью независимых, за исключением того, что задачи обучения должны запускаться только после завершения векторизации.

В следующем разделе мы посмотрим, как организовать такое параллельное выполнение с применением модуля `multiprocessing`.

Запуск параллельных задач

Чтобы увидеть, как модуль `multiprocessing` может помочь ускорить процедуру машинного обучения, рассмотрим пример, обучающий несколько моделей, выполняющий их перекрестную проверку и сохраняющий модели на диск. Для начала определим три функции, реализующие наивную байесовскую модель, логистическую регрессию и многослойный перцептрон. Каждая функция будет создавать свою модель, определяемую объектом `Pipeline`, извлекая текст из корпуса в заданном файле. Каждая задача также будет получать путь для сохранения модели и выводить результаты с помощью модуля `logging` (подробнее об этом чуть ниже):

```
from transformers import TextNormalizer, identity

from sklearn.pipeline import Pipeline
from sklearn.naive_bayes import MultinomialNB
from sklearn.linear_model import LogisticRegression
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.neural_network import MLPClassifier

def fit_naive_bayes(path, saveto = None, cv = 12):

    model = Pipeline([
        ('norm', TextNormalizer()),
        ('tfidf', TfidfVectorizer(tokenizer = identity, lowercase = False)),
        ('clf', MultinomialNB())
    ])

    if saveto is None:
        saveto = "naive_bayes_{}.pkl".format(time.time())

    scores, delta = train_model(path, model, saveto, cv)
    logger.info((
        "naive bayes training took {:0.2f} seconds "
        "with an average score of {:0.3f}"
    ).format(delta, scores.mean()))

def fit_logistic_regression(path, saveto = None, cv = 12):
    model = Pipeline([
```

```
        ('norm', TextNormalizer()),
        ('tfidf', TfidfVectorizer(tokenizer = identity, lowercase = False)),
        ('clf', LogisticRegression())
    ])

    if saveto is None:
        saveto = "logistic_regression_{}.pkl".format(time.time())

    scores, delta = train_model(path, model, saveto, cv)
    logger.info((
        "logistic regression training took {:0.2f} seconds "
        "with an average score of {:0.3f}"
    ).format(delta, scores.mean()))

def fit_multilayer_perceptron(path, saveto = None, cv = 12):
    model = Pipeline([
        ('norm', TextNormalizer()),
        ('tfidf', TfidfVectorizer(tokenizer = identity, lowercase = False)),
        ('clf', MLPClassifier(hidden_layer_sizes = (10,10),
                early_stopping = True))
    ])

    if saveto is None:
        saveto = "multilayer_perceptron_{}.pkl".format(time.time())

    scores, delta = train_model(path, model, saveto, cv)
    logger.info((
        "multilayer perceptron training took {:0.2f} seconds "
        "with an average score of {:0.3f}"
    ).format(delta, scores.mean()))
```

Для простоты конвейеры в функциях `fit_naive_bayes`, `fit_logistic_regression` и `fit_multilayer_perceptron` выполняют два одинаковых первых шага, используя объекты нормализации и векторизации, обсуждавшиеся в главе 4; однако можно предположить, что для разных моделей лучше подошли бы разные методы извлечения признаков.

Даже при том, что каждую функцию можно изменять и настраивать отдельно, все они используют один и тот же общий код — функцию `train_model()`, которая создает `PickledCorpusReader` для заданного пути. Функция `train_model()` использует этот объект для создания экземпляров и меток, вычисляет оценки с помощью утилиты `cross_val_score` из библиотеки Scikit-Learn, обучает модель, записывает ее на диск с помощью `joblib` (специализированный модуль, используемый библиотекой Scikit-Learn) и возвращает оценки:

```
from reader import PickledCorpusReader

from sklearn.externals import joblib
from sklearn.model_selection import cross_val_score
```

```
@timeit
def train_model(path, model, saveto = None, cv = 12):
    # Загрузить данные из корпуса и метки для классификации
    corpus = PickledCorpusReader(path)
    X = documents(corpus)
    y = labels(corpus)

    # Вычислить оценки методом перекрестной проверки
    scores = cross_val_score(model, X, y, cv = cv)

    # Обучить модель на всем массиве данных
    model.fit(X, y)

    # Записать на диск, если требуется
    if saveto:
        joblib.dump(model, saveto)

    # Вернуть оценки, а также время обучения через декоратор
    return scores
```

Обратите внимание на то, что функция `train_model` сама создает объект чтения корпуса (а не получает его извне). При использовании модуля `multiprocessing` все аргументы функций, а также возвращаемые объекты должны допускать возможность сериализации модулем `pickle`. Если объект чтения корпуса будет создаваться в дочерних процессах, его не придется сериализовать и передавать между процессами. Сложные объекты не всегда поддаются сериализации, и даже при том, что `CorpusReader` можно сериализовать и передавать в функции, иногда проще и быстрее передавать только простые данные, такие как строки.

`documents()` и `labels()` — это вспомогательные функции, которые извлекают данные из объекта чтения корпуса в список, как показано ниже:

```
def documents(corpus):
    return [
        list(corpus.docs(fileids = fileid))
        for fileid in corpus.fileids()
    ]

def labels(corpus):
    return [
        corpus.categories(fileids = fileid)[0]
        for fileid in corpus.fileids()
    ]
```

Время выполнения фиксируется с помощью обертки `@timeit`, простого отладочного декоратора, который мы используем для сравнения производительности разных версий:

```
import time
from functools import wraps

def timeit(func):
  @wraps(func)
  def wrapper(*args, **kwargs):
    start = time.time()
    result = func(*args, **kwargs)
    return result, time.time() - start
  return wrapper
```

Модуль `logging` в Python часто используется для координации операций журналирования в нескольких потоках выполнения и модулях. Настройки журналирования определяются на верхнем уровне модуля, за пределами любых функций, поэтому они вступают в силу на этапе импортирования модуля. В настройках можно указать директиву `%(processName)s`, чтобы определить, какой процесс будет выводить сообщения. Также в настройках задается имя модуля, чтобы потом можно было различить сообщения, выводимые разными модулями:

```
import logging

# Настройки журналирования configuration
logging.basicConfig(
    level = logging.INFO,
    format = "%(processName)-10s %(asctime)s %(message)s",
    datefmt = "%Y-%m-%d %H:%M:%S"
)
logger = logging.getLogger(__name__)
logger.setLevel(logging.INFO)
```

Модуль `logging` не поддерживает безопасное журналирование в один файл из нескольких процессов (но гарантирует безопасность при записи из нескольких потоков выполнения). Вообще говоря, запись в стандартный вывод или стандартный вывод ошибок обычно выполняется безупречно, однако для управления журналированием из нескольких процессов в контексте приложения существуют более надежные решения. Поэтому рекомендуется сразу начинать с использования `logging` (вместо инструкций `print`), чтобы заранее приготовиться к развертыванию приложения в эксплуатационной среде.

Теперь мы готовы выполнить наш код параллельно, с помощью функции `run_parallel`. Эта функция получает путь к корпусу и передает его всем задачам. В начале функции определяется список задач, затем для каждой задачи из списка создается объект `mp.Process` с именем, соответствующим имени задачи, ссылкой на вызываемый объект в аргументе `target` и позиционными и именованными аргументами в виде кортежа и словаря соответственно. Чтобы не потерять процессы, мы добавляем их в список `procs` перед запуском.

На данном этапе, если ничего не предпринять, главный процесс завершится сразу после выхода из функции `run_parallel`, что приведет к преждевременному завершению дочерних процессов или их осиротению (то есть они никогда не завершатся). Чтобы предотвратить это, мы перебираем дочерние процессы в цикле и присоединяем их к главному процессу. Благодаря этому функция заблокируется (приостановится), пока не выполнятся все вызовы метода `join` процессов. Выполняя обход всех процессов в цикле, мы гарантируем, что функция не продолжит выполнение, пока не завершатся все дочерние процессы, после чего мы сможем вывести время, затраченное на обработку:

```
def run_parallel(path):
    tasks = [
        fit_naive_bayes, fit_logistic_regression, fit_multilayer_perceptron,
    ]

    logger.info("beginning parallel tasks")
    start = time.time()

    procs = []
    for task in tasks:
        proc = mp.Process(name = task.__name__, target = task, args = (path,))
        procs.append(proc)
        proc.start()

    for proc in procs:
        proc.join()

    delta = time.time() - start
    logger.info("total parallel fit time: {:0.2f} seconds".format(delta))

if __name__ == '__main__':
    run_parallel("corpus/")
```

Запуск трех параллельных задач требует некоторых затрат и чуть больше усилий, чтобы обеспечить правильную работу. А что мы получим взамен? В табл. 11.1 приводятся для сравнения общее и среднее время выполнения задач по результатам десяти прогонов.

Таблица 11.1. Время последовательного и параллельного обучения (по результатам 10 прогонов)

Задача	Последовательное выполнение	Параллельное выполнение
Обучение наивной байесовской модели	86,93 секунды	94,18 секунды
Обучение модели логистической регрессии	91,84 секунды	100,56 секунды
Обучение модели многослойного перцептрона	95,16 секунды	103,40 секунды
Общее время обучения	273,94 секунды	103,47 секунды

Как видите, в параллельном режиме каждая отдельная задача выполнялась чуть дольше, чем в последовательном. Можно сказать, что это дополнительное время представляет накладные расходы на подготовку и запуск нескольких процессов и управление ими. Небольшое увеличение времени на выполнение отдельных задач с лихвой компенсируется уменьшением общего времени выполнения примерно до продолжительности самой долгой задачи, почти в 2,6 раза в сравнении с последовательной версией. При выполнении большого количества задач моделирования обработка в нескольких процессах позволяет получить серьезный выигрыш!

Пример использования `multiprocessing.Process` демонстрирует ряд важных понятий, однако на практике намного чаще используются пулы процессов, о которых мы поговорим в следующем разделе.

Важно иметь в виду, какой объем данных загружается в память, особенно в контексте использования нескольких процессов. Все процессы, выполняющиеся на одном компьютере, используют общую физическую память. Например, если на компьютере с объемом памяти 16 Гбайт запустить четыре задачи-процесса, загружающие корпус размером 4 Гбайт, они полностью исчерпают всю доступную память, из-за чего увеличится общее время выполнения. Чтобы избежать таких проблем, запуск параллельных задач обычно распределяется по времени: первыми запускаются самые продолжительные, а затем более быстрые задачи. Реализовать это можно с помощью задержек или пулов процессов, о которых рассказывается в следующем разделе.

Пулы процессов и очереди

В предыдущем разделе мы увидели, как с помощью объекта `multiprocessing.Process` выполнить несколько задач параллельно. Объект `Process` позволяет определять функции, которые должны выполняться независимо, и закрывать их по завершении. Но есть более совершенный способ, основанный на применении подклассов `Process`, каждый из которых использует метод `run()`, реализующий выполняемую задачу.

В больших архитектурах такой подход упрощает управление отдельными процессами и их аргументами (например, определять независимые имена, управлять подключениями к базам данных или другими атрибутами процессов) и часто применяется для организации параллельного выполнения задач. В *параллелизме на уровне задач* каждая задача имеет независимые входные и выходные данные (или выходные данные одной задачи могут служить входными данными другой). Напротив, *параллелизм на уровне данных* требует, чтобы

одна и та же задача *отображалась* на несколько блоков входных данных. Так как входные данные независимы, все задачи могут действовать параллельно и независимо. Для реализации *параллелизма на уровне данных* библиотека `multiprocessing` предоставляет более простые абстракции в форме классов `Pool` и `Queue`, о которых рассказывается в этом разделе.

Типичный шаблон, комбинирующий оба вида параллелизма на уровне задач и данных, основан на использовании двух задач распараллеливания данных; первая *отображает* операцию на несколько фрагментов входных данных, а вторая *сводит* результаты операций в единый массив. Этот стиль параллельных вычислений пользуется большой популярностью в фреймворках Hadoop и Spark, о которых мы поговорим в следующем разделе.

Сложные процедуры обработки можно описать в виде ориентированного ациклического графа (Directed Acyclic Graph, DAG), где выполнение последовательностей параллельных шагов синхронизируется в определенных точках. Точки синхронизации гарантируют завершение параллельной обработки всех фрагментов данных до перехода к следующему этапу. Обмен данными также обычно осуществляется в точках синхронизации, где главная задача передает данные параллельным задачам или получает их результаты.

Параллелизм на уровне данных также можно реализовать с помощью модуля `multiprocessing`, но при этом нужно учитывать некоторые дополнительные обстоятельства. Первое: сколько использовать процессов и как. В случае простых операций неэффективно запускать отдельный процесс для каждого фрагмента входных данных, а затем останавливать его (это может создать слишком большую нагрузку на операционную систему). Намного предпочтительнее определить фиксированное число процессов в `multiprocessing.Pool`, каждый из которых будет получать фрагмент входных данных, применять операцию и возвращать результаты, и так до тех пор, пока не будут обработаны все входные данные.

Такой подход влечет за собой второе обстоятельство: как организовать безопасные передачу и прием данных от процессов, не допуская дублирования или повреждения? Для этого можно воспользоваться структурой данных `multiprocessing.Queue`, обеспечивающей безопасность в многопоточной и многопроцессной среде за счет синхронизации операций с помощью блокировок и позволяющей обращаться к очереди только одному потоку или процессу в каждый конкретный момент времени. Процессы могут безопасно добавлять и извлекать элементы из очереди, действующей по принципу «первый пришел, первый ушел» (First-In, First-Out, FIFO).

На рис. 11.2 показана типичная архитектура такого способа обработки Пул запускает n процессов, каждый из которых начинает выполнять задание, читая данные из входной и сохраняя результаты в выходной очереди. Главный процесс в цикле добавляет исходные данные во входную очередь. Обычно после записи всех входных данных главный процесс добавляет в очередь n семафоров — флагов, сигнализирующих процессам из пула, что данных больше нет и они могут завершиться. Затем главный процесс присоединяет пул и приостанавливается в ожидании завершения всех процессов или сразу же приступает к извлечению результатов из выходной очереди для заключительной обработки.

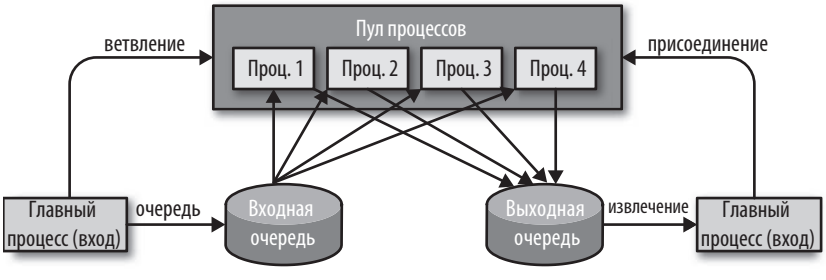


Рис. 11.2. Пул процессов и очереди

Если вам кажется, что для подготовки всех структур данных и управления кодом, который обрабатывает данные и работает с очередями, придется приложить немалые усилия, спешим вас успокоить — для всего этого объект `Pool` из библиотеки `multiprocessing` предоставляет простые методы. Методы `apply`, `map`, `imap` и `starmap` принимают функцию с ее аргументами и передают ее в пул для выполнения, блокируясь до получения результата. Эти методы имеют также асинхронные версии с окончанием `_async` в именах. Например, `apply_async` не блокирует работу главного процесса, а возвращает объект `AsynchronousResult`, который заполнится по завершении задачи, он может также вызвать указанные вами функции в случае успешного завершения или ошибки. Использование метода `apply_async` мы покажем в следующем разделе.

Параллельная обработка корпуса

Адаптация объекта чтения корпуса к параллельной обработке выполняется просто, если учесть, что в большинстве случаев документы можно обрабатывать независимо, особенно когда требуется выполнить частотный анализ, векторизацию и оценку. В таких ситуациях рабочим процессам из пула достаточно передать функцию, путь к корпусу на диске и идентификаторы файлов.

Однако самой, пожалуй, типичной и длительной операцией является предварительная обработка корпуса и его преобразование в форму, пригодную для вычислений. Процедура предварительной обработки, как обсуждалось в главе 3, принимает документ и преобразует его в стандартную структуру данных: список абзацев со списками предложений, которые в свою очередь являются списками кортежей `(token, part_of_speech)`. Результат предварительной обработки обычно сохраняется в формате модуля `pickle`, который обычно компактнее исходного документа и проще для загрузки в Python с целью дальнейшей обработки.

В главе 3 мы определили класс `Preprocessor`, обертывающий объект `CorpusReader`, метод `process` которого применяется к каждому документу в корпусе. Главной точкой входа для запуска предварительной обработки нам служил метод `transform`, осуществляющий преобразование документов из корпуса и сохраняющий их в указанный каталог.

Здесь мы расширим этот класс, чтобы получить возможность использовать `apply_async` с функцией обратного вызова для сохранения состояния. В данном примере мы создаем список `self.results` для хранения результатов, возвращаемых методом `process()`, но при необходимости вы легко сможете адаптировать `on_result()` и добавить в него дополнительную обработку или журналирование.

Также изменим метод `transform`. В нем мы определим количество доступных ядер процессора с помощью `mp.cpu_count()`. Затем создадим пул процессов, добавим задачи в очередь, выполнив обход всех документов и передав их в пул (в этой точке в игру вступает функция обратного вызова, изменяющая состояние). В заключение закроем пул (вызовом `pool.close()`), после чего в пул нельзя будет передать никакие другие задачи, и дочерние процессы присоединятся по завершении обработки, теперь подождем их завершения (вызовом `pool.join()`):

```
class ParallelPreprocessor(Preprocessor):

    def on_result(self, result):
        self.results.append(result)

    def transform(self, tasks = None):
        [...]

        # Очистить список results
        self.results = []

        # Создать пул процессов
        tasks = tasks or mp.cpu_count()
        pool = mp.Pool(processes = tasks)
```

```
# Добавить задачи в пул
for fileid in self.fileids():
    pool.apply_async(
        self.process, (fileid,), callback = self.on_result
    )

# Закрыть и присоединить пул
pool.close()
pool.join()

return self.results
```

Результат применения параллельной обработки поражает. На подмножестве корпуса Baleen, содержащем около 1,5 миллиона документов, последовательная обработка заняла примерно 30 часов — примерно 13 документов в секунду. За счет использования параллелизма на уровне задач и данных с 16 рабочими процессами обработка была выполнена менее чем за 2 часа.

Наше знакомство с потоками и процессами получилось довольно кратким и поверхностным, но мы надеемся, что вы смогли получить представление о проблемах и возможностях параллельной обработки. Многие задачи анализа текста можно ускорить и масштабировать просто за счет использования модуля `multiprocessing` и применения уже имеющегося кода. Выполняя параллельный код на современном ноутбуке или большом облачном экземпляре, относительно легко получить преимущества многопроцессорной обработки без необходимости настраивать кластер. В сравнении с кластерными вычислениями разрабатывать, отлаживать и сопровождать параллельные программы намного проще.

Кластерные вычисления с использованием Spark

Модуль `multiprocessing` дает простой и эффективный способ использования преимуществ современных многоядерных процессоров; однако с ростом сложности решаемых задач мы неизбежно столкнемся с физическим и экономическим пределом, ограничивающим количество ядер, доступных на одном компьютере. В какой-то момент покупка компьютера с вдвое большим количеством ядер окажется дороже покупки двух компьютеров с тем же количеством процессоров. Эта простая причина породила новую волну развития упрощенных методов кластерных вычислений.

Основой кластерных вычислений является координация нескольких компьютеров, связанных сетью, для последовательной и надежной работы — например,

если один компьютер выйдет из строя (что вполне вероятно, когда компьютеров много), кластер, как единое целое, продолжит работу. В кластерах, в отличие от многопроцессного контекста, нет единой операционной системы, организующей доступ к ресурсам и данным. Поэтому для управления *распределенным хранилищем*, хранением и копированием данных между узлами, а также координации *распределенных вычислений* между компьютерами в сети необходимо использовать некоторую инфраструктуру.

Хотя тема распределенного хранения данных выходит далеко за рамки этой книги, вы должны с ней ознакомиться, прежде чем приступать к кластерным вычислениям, если хотите обеспечить высокую надежность этих вычислений. Как правило, для управления данными на дисках в кластерах используются специализированные кластерные файловые системы, такие как HDFS и S3, а также базы данных, такие как Cassandra и HBase.

В оставшейся части главы мы рассмотрим приемы использования Apache Spark — фреймворка распределенных вычислений — для задач анализа текста. Мы дадим вам лишь краткое введение, не погружаясь глубоко в эту обширную тему. Установку Spark и PySpark мы оставляем вам как самостоятельное упражнение, а подробные инструкции вы найдете в документации к Spark[1]. Желающим получить более исчерпывающее введение мы рекомендуем книгу *Data Analytics with Hadoop* (O'Reilly)[2].

Устройство заданий в Spark

Spark — это механизм выполнения распределенных программ, основным преимуществом которого является поддержка вычислений в оперативной памяти. Поскольку приложения на основе Spark можно быстро писать на Java, Scala, Python и R, его название превратилось в синоним термина «большие данные» (big data). На основе Spark написано несколько библиотек, таких как Spark SQL и DataFrames, MLlib и GraphX, которые позволяют исследователям данных, использующим локальные вычисления на ноутбуках, быстро освоить кластерные вычисления с их помощью. Spark дает возможность разрабатывать приложения для обработки данных, ранее недоступных для машинного обучения из-за их объема; к этой категории относятся также многие текстовые корпусы. Фактически, первоначально фреймворки кластерных вычислений разрабатывались для обработки текстовых данных, извлекаемых из интернета.

---

[1] The Apache Software Foundation, *Apache Spark: Lightning-fast cluster computing* (2018), http://bit.ly/2GKR6k1

[2] Benjamin Bengfort and Jenny Kim, *Data Analytics with Hadoop: An Introduction for Data Scientists* (2016), https://oreil.ly/2JHfi8V

Spark может работать в двух режимах: в режиме клиента и в режиме кластера. В режиме кластера задание передается в кластер и обрабатывается независимо. В режиме клиента локальный клиент устанавливает интерактивное соединение с кластером, посылает задания в кластер и ожидает их завершения и получения результатов. Это позволяет взаимодействовать с кластером с помощью PySpark, интерактивного интерпретатора, аналогичного командной оболочке Python или Jupyter Notebook. Клиентский режим отлично подходит для динамического анализа и быстрого получения ответов на вопросы на основе небольших наборов данных и корпусов. Для более рутинных или продолжительных заданий лучше подходит кластерный режим.

В этом разделе мы кратко рассмотрим создание программ на Python с использованием Spark 2.x API. Представленный далее код можно запускать локально в PySpark или с помощью команды `spark-submit`.

В PySpark:

```
$ pyspark
Python 3.6.3 (v3.6.3:2c5fed86e0, Oct 3 2017, 00:32:08)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.3.0
      /_/

Using Python version 3.6.3 (v3.6.3:2c5fed86e0, Oct 3 2017 00:32:08)
SparkSession available as 'spark'.
>>> import hello
```

С использованием `spark-submit`:

```
$ spark-submit hello.py
```

На самом деле команде `spark-submit` нужно передать ряд флагов и аргументов, чтобы сообщить фреймворку Spark, например, URL кластера (аргумент `--master`), точку входа в приложение (аргумент `--class`), местоположение управляющей программы для выполнения в среде развертывания (аргумент `--deploy-mode`) и т. д.

Для подключения к кластеру необходим объект `SparkContext`. Обычно он хранится в глобальной переменной `sc`, и, если PySpark запускается в Jupyter Notebook или в терминале, эта переменная становится доступной автоматически. Если Spark запускается локально, объект `SparkContext`, по сути, дает вам

доступ к среде выполнения Spark, будь то кластер или единственный компьютер. Чтобы получить автономное задание на Python, сначала нужно создать экземпляр `SparkContext`, как показано ниже, где демонстрируется типичный шаблон заданий для Spark:

```
from pyspark import SparkConf, SparkContext

APP_NAME = "My Spark Application"

def main(sc):
    # Определите наборы данных RDD и примените к ним требуемые операции.

if __name__ == "__main__":
    # Настройка Spark
    conf = SparkConf().setAppName(APP_NAME)
    sc   = SparkContext(conf = conf)

    # Выполнение главной функции
    main(sc)
```

Теперь, определив основной шаблон для запуска заданий в Spark, нужно загрузить данные с диска так, чтобы Spark смог использовать их, а как это сделать, мы покажем в следующем разделе.

Распределение корпуса

Задания для Spark часто описываются как *ориентированные ациклические графы* (Directed Acyclic Graph, DAG) или ациклические потоки данных. Под этими словами понимается стиль программирования, предусматривающий загрузку данных в один или несколько разделов и последующее преобразование, слияние или разделение до достижения некоторого конечного состояния. То есть формирование задания для Spark начинается с создания *устойчивых распределенных наборов данных* (Resilient Distributed Dataset, RDD), коллекций данных, распределенных по нескольким машинам, что позволяет безопасно применять операции распределенным способом.

Самый простой способ создать наборы RDD — организовать данные по аналогии с организацией корпуса на диске: каталоги с метками документов и каждый документ в отдельном файле. Например, чтобы загрузить корпус «hobbies», представленный во врезке «Загрузка наборов данных Yellowbrick» в главе 8, мы использовали `sc.wholeTextFiles`, возвращающий RDD-коллекцию пар `(file name, content)`. В качестве аргумента этому методу передается путь, содержащий шаблонные символы и ссылающийся на каталог с файлами, одиночный файл

или сжатые файлы. В данном случае путь `"hobbies/*/*.txt"` соответствует любым файлам с расширением .txt в любых подкаталогах в каталоге hobbies:

```
corpus = sc.wholeTextFiles("hobbies/*/*.txt")
print(corpus.take(1))
```

Операция `take(1)` выводит первый элемент в корпусе RDD — это позволяет наглядно убедиться, что корпус является коллекцией кортежей строк. Первый элемент в корпусе «hobbies» имеет следующий вид:

```
[('file:/hobbies/books/56d62a53c1808113ffb87f1f.txt',
"\r\n\r\nFrom \n\n to \n\n, Oscar voters can't get enough of book
adaptations. Nowhere is this trend more obvious than in the Best
Actor and Best Actress categories.\n\n\r\n\r\nYes, movies have
been based on books and true stories since the silent film era,
but this year represents a notable spike...")]
```

Хранить на диске и анализировать можно данные в разных форматах, включая другие текстовые форматы, такие как JSON, CSV и XML, или двоичные форматы, такие как Avro, Parquet, Pickle, Protocol Buffers и т. д., которые могут оказаться более компактными. Как бы то ни было, после предварительной обработки (с помощью Spark или как описывалось в разделе «Параллельная обработка корпуса» выше) данные можно сохранить в виде объектов Python с использованием метода `RDD.saveAsPickleFile` (подобно тому, как мы сохраняли файлы в формате `pickle` для нашего предварительно обработанного корпуса) и затем загрузить вызовом `sc.pickleFile`.

ЭФФЕКТИВНОЕ ХРАНЕНИЕ В ФОРМАТЕ JSON

Сохранение в кластере множества маленьких файлов может оказаться сложной задачей из-за проблем с передачей данных и необходимости управления пространствами имен, не говоря уже об экономии пространства, которое может дать сжатие больших файлов. Поэтому данные лучше хранить в нескольких больших файлах, чем в большом количестве маленьких файлов.

Для объединения данных в большие файлы часто используется прием их преобразования в формат JSON Lines (или JSONL), где сериализованные объекты JSON представлены строками в файле, а не отдельными файлами. Загрузить и преобразовать такие данные JSON Lines в набор RDD можно так:

```
import json

corpus = sc.wholeTextFiles("corpus/*.jsonl")
corpus = corpus.flatMap(
    lambda d: [
        json.loads(line)
        for line in d[1].split("\n")
        if line
    ]
)
```

В данном случае мы используем функцию, которая возвращает массив объектов JSON, извлеченных из каждой строки в документе. Метод `flatMap` превращает список списков в один плоский список; таким образом, набор RDD — это коллекция словарей Python, полученных из строк во всех файлах в наборе данных.

После загрузки корпуса «hobbies» в наборы RDD к ним можно применить преобразования и другие действия, которые мы подробно обсудим в следующем разделе.

Операции RDD

В программах на основе Spark доступны операции двух основных типов: *преобразования* и *действия*. Преобразования — это операции с данными, создающие новый набор RDD из существующего. Преобразования не выполняются в кластере немедленно, они просто описывают последовательность шагов, применяемых к одному или нескольким наборам RDD. Действия, напротив, выполняются немедленно, возвращая результат управляющей программе (в режиме клиента), записывая данные на диск или производя любые другие вычисления (в режиме кластера).

Преобразования выполняются в отложенном режиме, то есть они применяются, только когда действию потребуется результат преобразования, что позволяет фреймворку Spark оптимизировать создание и хранение в памяти наборов RDD. Это может вызывать путаницу у пользователей; при появлении исключения может быть неочевидно, какая операция вызвала его; кроме того, иногда действие может запустить весьма продолжительные вычисления в кластере. Поэтому мы рекомендуем вести разработку в клиентском режиме и использовать небольшую выборку из общего набора данных, а затем создавать приложения, действующие в режиме кластера.

В практике наиболее распространены три преобразования: `map`, `filter` и `flatMap`. Каждое принимает функцию в первом аргументе. Эта функция применяется к каждому элементу в наборе RDD, а ее возвращаемые значения помещаются в новый набор RDD.

Например, воспользуемся модулем Python `operator` для получения подкатегории каждого документа. Для этого определим функцию `parse_label`, извлекающую имя категории из пути к файлу документа. Так же как в примере выше, создадим набор RDD пар «ключ/значение» `(filename, content)`, загрузив все текстовые файлы из указанного корпуса. Затем создадим набор RDD `labels`, применив к каждому элементу операцию `itemgetter(0)`, которая выбирает из набора RDD `data` только имена файлов, и затем функцию `parse_label`:

```
import os
from operator import itemgetter

def parse_label(path):
    # Возвращает имя каталога, содержащего файл
    return os.path.basename(os.path.dirname(path))

data = sc.wholeTextFiles("hobbies/*/*.txt")
labels = data.map(itemgetter(0)).map(parse_label)
```

Обратите внимание, что в этой точке кластер пока ничего не выполнил, потому что мы определили преобразование, но не действие. Допустим, нам понадобилось подсчитать документы в каждой подкатегории.

В настоящий момент наш набор RDD `labels` является коллекцией строк, однако многие операции в Spark (например, `groupByKey` и `reduceByKey`) работают только с парами «ключ/значение». Чтобы преодолеть это ограничение, можно создать новый набор RDD с именем `label_counts`, который получается отображением каждой метки в пару «ключ/значение», где ключ — это имя метки, а значение — число 1. После этого можно вызвать `reduceByKey` с оператором `add`, который вычислит сумму всех единиц и вернет общее количество документов в категории. После этого используем действие `collect`, которое выполнит преобразования в кластере, загрузив данные и создав наборы RDD `labels` и `label_count`, и вернет список кортежей `(label, count)`, а мы выведем его в клиентской программе:

```
from operator import add

label_count = labels.map(lambda l: (l, 1)).reduceByKey(add)
for label, count in label_count.collect():
    print("{}: {}".format(label, count))
```

Вот какие результаты мы получили:

```
books: 72
cinema: 100
gaming: 128
sports: 118
cooking: 30
```

К действиям также относятся: `reduce` (объединяет элементы коллекции), `count` (возвращает количество элементов в коллекции), а также `take` и `first` (возвращают первый или *n* первых элементов из коллекции соответственно). Эти действия могут пригодиться в интерактивном режиме и для отладки, но будьте осторожны, работая с большими наборами RDD — по ошибке легко можно попытаться загрузить огромный набор данных, не умещающийся в памяти компьютера! Чтобы избежать подобной проблемы, при работе с большими наборами данных часто используют действие `takeSample`, создающее случайную выборку с заменой или без замены исходной коллекции или просто сохраняющее получившийся набор данных на диск для последующего использования.

Как рассказывалось в предыдущих разделах, приложения на основе Spark определяют потоки операций с данными. Сначала мы загружаем один или несколько наборов RDD, применяем к ним преобразования, объединяем их, затем применяем действия и сохраняем полученные результаты на диск или объединяем данные и возвращаем их обратно управляющей программе. Эта мощная абстракция позволяет рассуждать в терминах коллекции данных, а не в терминах распределенных вычислений, и облегчает освоение кластерных вычислений. Более сложный вариант использования Spark включает создание коллекций DataFrame и Graph, с которыми мы встретимся в следующем разделе.

Обработка естественного языка в Spark

Обработка естественного языка представляет особый интерес для сообщества пользователей распределенных систем. И не только потому, что текстовые наборы являются самыми большими (фактически, фреймворк Hadoop проектировался специально для парсинга документов HTML в поисковых механизмах), но также потому, что для большей эффективности современным приложениям анализа естественного языка требуются особенно большие корпусы. В результате библиотека машинного обучения MLLib[1] в Spark может похвастаться большим количеством инструментов для интеллектуального извлечения при-

[1] The Apache Software Foundation, *MLlib: Apache Spark's scalable machine learning library* (2018), http://bit.ly/2GJQP0Y

знаков и векторизации текста, аналогичных обсуждавшимся в главе 4, включая утилиты частотного, прямого, TF–IDF и word2vec кодирования.

Машинное обучение в Spark начинается с получения коллекции данных, похожей на набор RDD, — SparkSQL DataFrame. Коллекции Spark DataFrame концептуально похожи на таблицы в реляционных базах данных или их аналоги Pandas и организуют данные в таблицу, состоящую из строк и столбцов. При этом они добавили обширный спектр оптимизаций, основанных на использовании механизма выполнения SparkSQL, и быстро превратились в стандарт распределенной обработки данных.

Если бы мы пожелали задействовать Spark MLLib для обработки корпуса «hobbies», мы сначала преобразовали бы корпус RDD с помощью `SparkSession` (доступна в PySpark как глобальная переменная `spark`), чтобы создать `DataFrame` из коллекции кортежей, определив имя каждого столбца:

```
# Загрузить данные с диска
corpus = sc.wholeTextFiles("hobbies/*/*.txt")

# Получить метки из путей к файлам
corpus = corpus.map(lambda d: (parse_label(d[0]), d[1]))

# Создать DataFrame с двумя столбцами
df = spark.createDataFrame(corpus, ["category", "text"])
```

`SparkSession` — это точка входа в SparkSQL и Spark 2.x API. Чтобы воспользоваться этим механизмом в приложении для `spark-submit`, нужно сначала создать его, подобно тому как мы создавали `SparkContext` в предыдущем разделе. Измените шаблон программы для Spark, добавив следующие строки:

```
from pyspark.sql import SparkSession
from pyspark import SparkConf, SparkContext

APP_NAME = "My Spark Text Analysis"

def main(sc, spark):
    # Определите коллекции DataFrame и примените к ним
    # алгоритмы преобразования и оценки

if __name__ == "__main__":
    # Настройка Spark
    conf = SparkConf().setAppName(APP_NAME)
    sc   = SparkContext(conf = conf)

    # Создание сеанса SparkSQL
    spark = SparkSession(sc)

    # Выполнение главной функции
    main(sc, spark)
```

После преобразования корпуса в коллекцию Spark `DataFrame` можно приступать к обучению моделей и преобразованию наборов данных. К счастью, программный интерфейс Spark очень похож на Scikit-Learn, поэтому переход от Scikit-Learn или совместное использование Scikit-Learn и Spark не представляют никаких сложностей.

От Scikit-Learn к MLLib

Библиотека Spark MLLib быстро развивается и включает реализации алгоритмов классификации, регрессии, кластеризации, совместной фильтрации и выявления закономерностей. Многие из них создавались по образу и подобию аналогичных алгоритмов в Scikit-Learn, поэтому пользователи Scikit-Learn быстро освоят библиотеку MLLib и доступные в ней модели.

Важно помнить, что основное предназначение Spark — обработка очень больших наборов данных в кластере. По этой причине в распределенных вычислениях используется множество оптимизаций. Это означает, что алгоритмы, легко поддающиеся распараллеливанию, почти наверняка будут доступны в MLLib, но другие, для которых сложно реализовать параллельные варианты, могут отсутствовать. Например, алгоритм Random Forest (случайный лес), суть которого заключается в случайном разбиении набора данных и обучении комплекса деревьев решений на подмножествах данных, легко можно распределить между несколькими компьютерами. Алгоритм стохастического градиентного спуска, напротив, очень трудно распараллелить, потому что его внутреннее состояние обновляется после каждой итерации. Spark выбирает стратегии, оптимальные для вычислений в кластере, а не базовые модели. Для пользователей Scikit-Learn это может проявляться в виде менее точных моделей (из-за аппроксимаций в Spark) или меньшего количества гиперпараметров. Другие модели (например, *k*-ближайших соседей) могут быть вообще недоступны из-за невозможности организовать эффективное их распределение.

Spark ML API, так же как Scikit-Learn, основан на идее конвейера (`Pipeline`). Конвейеры позволяют определять последовательности алгоритмов, представляющие единую модель, которую можно обучить на данных и использовать для прогнозирования.

В Spark, в отличие от Scikit-Learn, алгоритмы обучения и оценки, а также конвейеры вместо изменения внутреннего состояния возвращают совершенно новый объект модели. Это объясняется неизменяемостью наборов RDD.

Объект `Pipeline` в Spark состоит из *этапов*, каждый из которых должен быть экземпляром `Transformer` или `Estimator`. Экземпляры `Transformer` преобразуют одну коллекцию `DataFrame` в другую, читая столбцы из исходной коллекции `DataFrame`, преобразуя их в методе `transform()` и добавляя новые столбцы. По этой причине все преобразователи обычно требуют указать имена столбцов во входной и выходной коллекциях, которые должны быть уникальными.

Объекты `Estimator` реализуют метод `fit()`, обучающий и возвращающий новую модель. Модели сами являются преобразователями, поэтому объекты `Estimator` также определяют входные и выходные столбцы коллекций `DataFrame`. Когда вызывается метод `transform()` *прогностической модели*, предсказания сохраняются в выходном столбце. Такое решение несколько отличается от реализованного в Scikit-Learn, где имеется метод `predict()`, но в распределенном контексте метод `transform()` более уместен, потому что применяется к потенциально очень большому набору данных.

Этапы конвейера должны быть уникальными, чтобы гарантировать успешное преодоление проверки на этапе компиляции и преобразование в ациклический граф — граф преобразований и действий в среде выполнения Spark. Это означает, что параметры `inputCol` и `outputCol` должны быть уникальными на каждом этапе, и один и тот же экземпляр нельзя использовать на разных этапах.

Поскольку сохранение модели и ее повторное значение чрезвычайно важны для машинного обучения, Spark предусматривает возможность экспортирования и импортирования моделей по требованию. Многие объекты `Transformer` и `Pipeline` имеют метод `save()` для сохранения модели на диск и соответствующий метод `load()` для загрузки сохраненной модели.

Наконец, библиотека Spark MLLib реализует еще одно дополнительное понятие: `Parameter`. Объекты `Parameter` — это распределенные структуры данных с собственным описанием. Эта структура данных совершенно необходима в машинном обучении, потому что обеспечивает безопасную *рассылку* переменных всем исполнителям в кластере. Рассылаемые переменные — это данные, доступные только для чтения как глобальные переменные всем исполнителям в кластере.

Некоторые параметры могут даже изменяться в процессе обучения. Такие параметры (их называют *аккумуляторами*) представляют структуры данных, поддерживающие ассоциативные и коммутативные параллельные операции. Поэтому многие параметры должны извлекаться или изменяться с помощью

специальных методов класса `Transformer`, из которых наиболее часто используются `getParam(param)`, `getOrDefault(param)` и `setParams()`. С помощью метода `explainParam()` можно просмотреть параметр и связанные с ним значения, и он часто используется в PySpark и Jupyter Notebook.

Теперь, получив общее представление о Spark MLLib API, перейдем к следующему разделу, где рассмотрим несколько примеров.

Извлечение признаков

Первый шаг в обработке естественного языка с использованием Spark — извлечение признаков из текста, лексемизация и векторизация высказываний и документов. В Spark имеется богатый набор инструментов извлечения признаков из текста, включая индексирование, удаление стоп-слов и извлечение *n*-грамм. Также в Spark есть утилиты векторизации для частотного, прямого, TF–IDF и word2vec кодирования. Все эти утилиты принимают список лексем, поэтому на первом шаге обычно выполняется лексемизация.

В следующем фрагменте мы инициализируем преобразователь `RegexTokenizer` несколькими параметрами. Параметры `inputCol` и `outputCol` определяют, как преобразователи будут работать с данной коллекцией DataFrame; регулярное выражение `"\\w+"` определяет, как должна производиться разбивка текста; а `gaps = False` гарантирует, что этому выражению будут соответствовать слова, а не пробелы между ними. После преобразования корпуса DataFrame (здесь предполагается, что корпус уже загружен) в нем появится новый столбец «tokens» с типом данных «массив строк», который необходим для большинства инструментов извлечения признаков:

```
from pyspark.ml.feature import RegexTokenizer

# Создать RegexTokenizer
tokens = RegexTokenizer(
    inputCol = "text", outputCol = "tokens",
    pattern = "\\w+", gaps = False, toLowercase = True)

# Преобразовать корпус
corpus = tokens.transform(corpus)
```

Этот лексемизатор удалит все знаки препинания и разобьет составные слова, которые записываются через дефис. Более сложное выражение, такое как `"\\w+|\$[\\d\.]+|\\S+"`, выполнит разбивку по знакам препинания, но не удалит их и даже сохранит денежные суммы, такие как `"$8.31"`.

Некоторые модели в Spark определяют значения по умолчанию для `inputCol` и `outputCol`, такие как `"features"` или `"predictions"`, которые могут приводить к ошибкам или неправильной работе; как правило, лучше явно указывать значения этих параметров для каждого преобразователя и модели.

Поскольку преобразование документов в векторы признаков выполняется в несколько этапов, которые должны быть согласованы, векторизация реализована как конвейер `Pipeline`. Создание локальных переменных для каждого преобразователя и их передача в конвейер — распространенный прием, но это может сделать сценарии для Spark слишком подробными и подверженными ошибкам. Одно из решений этой проблемы — определить функцию, возвращающую стандартный конвейер `Pipeline` векторизации корпуса, которую можно импортировать в сценарий.

Функция `make_vectorizer` создает экземпляр `Pipeline` с лексемизатором `Tokenizer` и векторизатором `HashingTF`, отображающим лексемы в их частоты методом *хеширования признаков*, суть которого заключается в вычислении хеш-суммы Murmur3 лексемы и ее использования в роли числового признака. Эта функция может также добавлять в конвейер удаление стоп-слов и преобразователи TF–IDF. Чтобы гарантировать правильное преобразование коллекции DataFrame с уникальными столбцами, каждый преобразователь получает уникальное имя выходного столбца и использует `stages[-1].getOutputCol()`, чтобы определить имя входного столбца по предыдущему этапу:

```
def make_vectorizer(stopwords = True, tfidf = True, n_features = 5000):
    # Создает конвейер векторизации, который начинается с лексемизации
    stages = [
        Tokenizer(inputCol = "text", outputCol = "tokens"),
    ]

    # Добавить в конвейер удаление стоп-слов, если затребовано
    if stopwords:
        stages.append(
            StopWordsRemover(
                caseSensitive = False, outputCol = "filtered_tokens",
                inputCol = stages[-1].getOutputCol(),
            ),
        )

    # Добавить частотный векторизатор HashingTF
    stages.append(
        HashingTF(
            numFeatures = n_features,
            inputCol = stages[-1].getOutputCol(),
```

```
            outputCol = "frequency"
        )
    )

    # Добавить векторизатор IDF, если затребовано
    if tfidf:
        stages.append(
            IDF(inputCol = stages[-1].getOutputCol(), outputCol = "tfidf")
        )

    # Вернуть получившийся конвейер
    return Pipeline(stages = stages)
```

Поскольку `HashingTF` и `IDF` являются моделями, этот векторизатор нужно обучить на входных данных; вызов `make_vectorizer().fit(corpus)` гарантирует такое обучение модели и ее способность преобразовывать данные:

```
vectors = make_vectorizer().fit(corpus)
corpus = vectors.transform(corpus)
corpus[['label', 'tokens', 'tfidf']].show(5)
```

Вот как выглядят первые пять строк результата:

```
+-----+--------------------+--------------------+
|label|              tokens|               tfidf|
+-----+--------------------+--------------------+
|books|[, name, :, ian, ...|(5000,[15,24,40,4...|
|books|[, written, by, k...|(5000,[8,177,282,...|
|books|[, last, night,as...|(5000,[3,9,13,27,...|
|books|[, a, sophisticat...|(5000,[26,119,154...|
|books|[, pools, are, so...|(5000,[384,569,60...|
+------+--------------------+--------------------+
only showing top 5 rows
```

По окончании преобразования корпус будет содержать шесть столбцов: два оригинальных столбца и четыре столбца, представляющие отдельные этапы процесса преобразования. С помощью DataFrame API мы выбрали три из них для исследования и отладки.

После извлечения признаков можно начинать создавать модели, применив прием тройки выбора модели. Сначала создадим модель кластеризации, а затем классификации.

Кластеризация текста с помощью MLLib

На момент написания этих строк в Spark было реализовано четыре алгоритма кластеризации, хорошо подходящих для тематического моделирования:

k-средних и дивизимный *k*-средних (bisecting *k*-means), латентное размещение Дирихле (Latent Dirichlet Allocation, LDA) и гауссова смесь распределений (Gaussian Mixture Models, GMM). В этом разделе мы рассмотрим конвейер кластеризации, сначала использующий алгоритм word2vec для преобразования «мешка слов» в вектор фиксированной длины, а затем `BisectingKMeans` для выявления кластеров похожих документов.

Дивизимный алгоритм *k*-средних — это нисходящий подход к иерархической кластеризации, использующий метод *k*-средних для рекурсивного деления кластеров (например, алгоритм *k*-средних применяется к каждому кластеру в иерархии с `k = 2`). После разделения кластера сохраняется та его часть, которая имеет наивысшую оценку сходства, а остальные данные продолжают делиться, пока не будет достигнуто желаемое число кластеров. Этот алгоритм имеет быструю сходимость, и, поскольку выполняет несколько итераций с `k = 2`, обычно он работает быстрее, чем алгоритм *k*-средних с бóльшим значением *k*, но в результате получает совершенно иное деление на кластеры.

В следующем фрагменте кода мы создаем начальный конвейер `Pipeline` и определяем входные и выходные столбцы, а также параметры для наших преобразователей, такие как фиксированный размер векторов слов и величину *k* для алгоритма *k*-средних. Вызов метода `fit` конвейера с нашими данными вернет модель, которую затем можно использовать для преобразования корпуса:

```
from tabulate import tabulate
from pyspark.ml import Pipeline
from pyspark.ml.clustering import BisectingKMeans
from pyspark.ml.feature import Word2Vec, Tokenizer

# Создать конвейер векторизации/кластеризации
pipeline = Pipeline(stages = [
    Tokenizer(inputCol = "text", outputCol = "tokens"),
    Word2Vec(vectorSize = 7, minCount = 0, inputCol = "tokens",
outputCol = "vecs"),
    BisectingKMeans(k = 10, featuresCol = "vecs", maxIter = 10),
])

# Обучить модель
model = pipeline.fit(corpus)
corpus = model.transform(corpus)
```

Чтобы оценить качество кластеризации, извлечем объекты `BisectingKMeans` и `word2vec` и сохраним их в локальных переменных: `bkm` — обратившись к последнему этапу *модели* (не конвейера `Pipeline`), и `wvec` — к предпоследнему этапу. Для вычисления суммы квадратов расстояний документов от центра используем метод `computeCost` (и снова здесь предполагается, что корпус до-

кументов уже загружен). Чем меньше эта сумма, тем более плотными и четко выраженными получились кластеры. Также можно вычислить их размеры по количеству документов:

```
# Получить этапы
bkm = model.stages[-1]
wvec = model.stages[-2]

# Оценить результат кластеризации
cost = bkm.computeCost(corpus)
sizes = bkm.summary.clusterSizes
```

Чтобы получить текстовое представление каждого центра, сначала выполним обход всех индексов (`ci`) и центроидов (`c`) кластеров, перебирая центры кластеров. Для каждого центра найдем семь ближайших синонимов, векторы слов, ближайших к центру, затем построим таблицу, отображающую индексы центров, размеры кластеров и соответствующие им синонимы. Затем выведем эту таблицу в отформатированном виде с помощью библиотеки `tabulate`:

```
# Получить текстовое представление каждого кластера
table = [["Cluster", "Size", "Terms"]]
for ci, c in enumerate(bkm.clusterCenters()):
    ct = wvec.findSynonyms(c, 7)
    size = sizes[ci]
    terms = " ".join([row.word for row in ct.take(7)])
    table.append([ci, size, terms])

# Вывести результаты
print(tabulate(table))
print("Sum of square distance to center: {:0.3f}".format(cost))
```

Вот какие результаты получились у нас:

```
Cluster Size Terms
------- ---- ---------------------------------------------------------------
    0     81 the"soros caption,"bye markus henkes clarity. presentation,elon
    1      3 novak hiatt veered monopolists,then,would clarity. cilantro.
    2      2 publics. shipmatrix. shiri flickr groupon,meanwhile,has sleek!
    3      2 barrymore 8,2016 tips? muck glorifies tags between,earning
    4    265 getting sander countervailing officers,ohio,then voter. dykstra
    5    550 back peyton's condescending embryos racist,any voter. nebraska
    6    248 maxx,and davan think'i smile,i 2014,psychologists thriving.
    7    431 ethnography akhtar problem,and studies,taken monica,california.
    8    453 instilled wife! pnas,the ideology,with prowess,pride
    9    503 products,whereas attacking grouper sets,facebook flushing,
Sum of square distance to center: 39.750
```

Как видите, некоторые кластеры получились большими и размытыми, поэтому следующим нашим шагом могла бы стать итеративная процедура выбора

и оценки значения *k*, как описывалось в главах 6 и 8, до достижения подходящего качества модели.

Классификация текста с помощью MLLib

Из числа алгоритмов классификации в настоящее время в Spark реализованы модели логистической регрессии, деревья решений, случайные леса, градиентный бустинг, многослойные перцептроны и метод опорных векторов (Support Vector Machine, SVM). Эти модели хорошо подходят для анализа текста и обычно используются для решения задач классификации. Классификация выполняется подобно кластеризации, но включает дополнительные шаги индексирования меток для каждого документа и оценки точности модели. Так как оценка должна выполняться на контрольных данных, не использовавшихся при обучении модели, мы должны разбить корпус на обучающую и контрольную выборки.

После этапа векторизации закодируем метки документа с помощью `StringIndexer`, который преобразует столбец строк в нашей коллекции DataFrame в столбец индексов `[0, len(column)]`, упорядоченный по частоте (например, наиболее часто встречающаяся метка получит индекс 0). Затем разобьем коллекцию DataFrame на случайные выборки — обучающую, включающую 80 % данных из исходного набора, и контрольную, включающую 20 %:

```
from pyspark.ml.feature import StringIndexer

# Создать объект для векторизации корпуса
vector = make_vectorizer().fit(corpus)

# Индексировать классифицирующие метки
labelIndex = StringIndexer(inputCol = "label", outputCol = "indexedLabel")
labelIndex = labelIndex.fit(corpus)

# Разбить данные на обучающую и контрольную выборки
training, test = corpus.randomSplit([0.8, 0.2])
```

В предыдущем примере перед разбиением на две выборки оба преобразователя, `vector` и `labelIndex`, обучаются на полном объеме данных, чтобы гарантировать кодирование всех индексов и слов. Это может быть правильной или неправильной стратегией, в зависимости от ваших ожиданий относительно реальных данных.

Теперь создадим конвейер `Pipeline` и определим процесс кодирования меток и документов перед построением модели `LogisticRegression`:

```
from pyspark.ml.classification import LogisticRegression

model = Pipeline(stages = [
    vector, labelIndex, clf
]).fit(training)

# Получить предсказания
predictions = model.transform(test)
predictions.select("prediction", "indexedLabel", "tfidf").show(5)
```

Для оценки модели можно воспользоваться утилитами оценки классификации в Spark. Например:

```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

evaluator = MulticlassClassificationEvaluator(
    labelCol = "indexedLabel",
    predictionCol = "prediction",
    metricName = "f1"
)

score = evaluator.evaluate(predictions)
print("F1 Score: {:0.3f}".format(score))
```

В Spark есть и другие утилиты для перекрестной проверки и выбора моделей, учитывающие тот факт, что модели обучаются на больших объемах данных. Вообще говоря, разделение больших корпусов и многократное обучение моделей для получения их оценок требует много времени, поэтому кеширование и активное обучение являются важными элементами процесса моделирования, минимизирующими случаи многократного повторения одних и тех же операций. Понимание компромиссов, неизбежных при использовании распределенного подхода к машинному обучению, подводит нас к следующей теме — использованию локальных вычислений с глобальным набором данных.

Локальное обучение, глобальное применение

Запуская фрагменты кода Spark локально с помощью PySpark или `spark-submit`, вы могли заметить, что фреймворк Spark действует не так быстро, как рекламируется; на самом деле он работает даже медленнее, чем эквивалентные приемы моделирования с использованием Scikit-Learn на локальном компьютере. Spark имеет очень большие накладные расходы, создавая процессы для мониторинга заданий, и организует многочисленные взаимодействия между процессами для их синхронизации и поддержки отказоустойчивости; выигрыш в скорости начинает проявляться, только когда наборы данных становятся намного больше, чем может обработать один компьютер.

Одно из решений этой проблемы — параллельные предварительная обработка данных и векторизация в кластере, получение выборки данных, локальное обучение модели и затем глобальное ее применение ко всему набору данных. Несмотря на то что при этом исчезает преимущество обучения моделей на больших наборах данных, зачастую это единственный способ создания узкоспециализированных моделей в кластере. Этот прием можно использовать для создания разных моделей для разных фрагментов набора данных или для ускорения процесса тестирования.

Чтобы реализовать такое решение, сначала векторизуем корпус, а затем получим выборку. Векторизация в кластере гарантирует, что используемая модель из Scikit-Learn не будет зависеть от терминов или других состояний, которые могут быть исключены в процессе создания выборки. Выборка производится без замены (значение `False` в первом аргументе) и отбирает 10 % данных (аргумент `0.1`). Будьте осторожны с размером выборки; вы легко можете исчерпать память компьютера, даже запросив всего 10 % от общего объема корпуса! Действие `collect()` выполняет код извлечения выборки и помещает набор данных в память локального компьютера в виде списка. После этого из полученных данных можно сконструировать `X` и `y` и обучить модель:

```
# Векторизовать корпус в кластере
vector = make_vectorizer().fit(corpus)
corpus = vector.transform(corpus)

# Получить выборку из набора данных
sample = corpus.sample(False, 0.1).collect()
X = [row['tfidf'] for row in sample]
y = [row['label'] for row in sample]

# Обучить модель Scikit-Learn
clf = AdaBoostClassifier()
clf.fit(X, y)
```

Для оценки точности модели отправим ее в кластер вызовом `broadcast` и используем аккумуляторы для получения числа верных и неверных предсказаний:

```
# Отправить модель Scikit-Learn в кластер
clf = sc.broadcast(clf)

# Создать аккумуляторы для получения числа правильных и неправильных
# предсказаний
correct = sc.accumulator(0)
incorrect = sc.accumulator(1)
```

Чтобы задействовать эти переменные в параллельных вычислениях, мы должны как-то сослаться на них в операциях DataFrame. В Spark это можно сделать

путем отправки *замыкания* (closure). Обычно с этой целью определяется функция, возвращающая замыкание. Мы можем определить замыкание для оценки точности, которое применяется к методу `predict` классификатора, и затем увеличивать аккумулятор `correct` или `incorrect`, сравнивая предсказание с фактической меткой. Чтобы создать такое замыкание, определим функцию `make_accuracy_closure`, как показано ниже:

```
def make_accuracy_closure(model, correct, incorrect):
    # model должна быть широковещательной переменной
    # correct и incorrect должны быть аккумуляторами
    def inner(rows):
        X = []
        y = []

        for row in rows:
            X.append(row['tfidf'])
            y.append(row['label'])

        yp = model.value.predict(X)
        for yi, ypi in zip(y, yp):
            if yi == ypi:
                correct.add(1)
            else:
                incorrect.add(1)
    return inner
```

Теперь можно использовать действие `foreachPartition` коллекции DataFrame, чтобы каждый исполнитель послал свою порцию из DataFrame в замыкание, которое обновит содержимое аккумуляторов. По завершении мы сможем вычислить точность модели на глобальном наборе данных:

```
# Создать замыкание точности
accuracy = make_accuracy_closure(clf, incorrect, correct)

# вычислить количество верных и неверных предсказаний
corpus.foreachPartition(accuracy)

accuracy = float(correct.value) / float(correct.value + incorrect.value)
print("Global accuracy of model was {}".format(accuracy))
```

Стратегия локального обучения и глобальной оценки позволяет разработчикам легко создавать и оценивать модели параллельно и может повысить скорость обработки данных в Spark, особенно на этапах исследований и экспериментов. К многоуровневым моделям намного проще применить перекрестную проверку на наборах данных среднего размера, а модели с частично распределенной реализацией (или вообще без реализации) можно исследовать более полно.

Это обобщение стратегии «последней мили», характерной для систем обработки больших данных. Фреймворк Spark позволяет манипулировать большими объемами данных, но, чтобы использовать эти данные в кластере, выполняются операции с данными (использующие вычислительные ресурсы облака, которые могут составлять десятки или даже сотни гигабайт), такие как фильтрация, агрегирование или обобщение, приводящие их в форму, умещающуюся в памяти локального компьютера. Это одна из причин, почему интерактивная модель выполнения в Spark пользуется большой популярностью у специалистов по обработке данных; она позволяет сочетать локальные и кластерные вычисления, как было показано в этом разделе.

В заключение

Одна из замечательных сторон проведения анализа текста в информационную эпоху — простота создания приложений, осуществляющих анализ текста и затем генерирующих дополнительные текстовые данные из ответов человека. Это означает быстрый рост корпусов для машинного обучения и выход за рамки вычислительных возможностей единственного процесса или даже единственного компьютера. С увеличением времени анализа снижается производительность итеративных или экспериментальных процедур и встает необходимость использования некоторого метода масштабирования для придания дополнительного импульса.

Модуль `multiprocessing` — первый шаг к масштабированию анализа. В этом случае готовый код, осуществляющий анализ, нужно лишь адаптировать или включить в контекст многозадачной обработки. Библиотека `multiprocessing` пользуется преимуществами многоядерных процессоров и наличия большого объема оперативной памяти в современных компьютерах для параллельного выполнения кода. Данные все еще хранятся локально, но с достаточно мощной машиной время выполнения работы огромного объема можно сократить до разумных пределов.

Когда объем данных увеличивается до такой степени, что они не умещаются в памяти одного компьютера, следует использовать технологии кластерных вычислений. Spark предоставляет среду выполнения и позволяет интерактивно сочетать вычисления на локальном компьютере с использованием PySpark в Jupyter Notebook и кластерные вычисления, производимые множеством исполнителей с ограниченными фрагментами данных. Такая интерактивность объединяет лучшее из миров последовательных и параллельных алгоритмов. Фреймворк Spark требует изменений в коде и подходов к программированию,

а иногда даже выбора других библиотек машинного обучения, но позволяет обрабатывать огромные наборы данных, иначе недоступные для методов анализа текста, и может стать богатой основой для новых приложений.

Ключом к использованию распределенных вычислений является хорошее понимание компромиссов. Знание, к какой категории принадлежит операция — вычислительных или ввода-вывода, — может избавить от множества проблем, связанных с поиском причин, почему задания выполняются дольше, чем могли бы, или используют слишком много ресурсов (и поможет сбалансировать параллелизм на уровне задач и данных). Кроме того, параллельные вычисления несут с собой дополнительные накладные расходы, которые должны быть оправданы; понимание этого поможет принимать решения о расширении экспериментальной разработки. Наконец, понимание того, как происходит аппроксимация алгоритмов в распределенном контексте, совершенно необходимо для построения значимых моделей и интерпретации результатов.

Spark — первый фреймворк на волне современных технологий, который дал нам возможность обрабатывать все больше и больше данных для создания значимых моделей. В следующей главе мы рассмотрим распределенные методы обучения так называемых глубоких моделей — нейронных сетей, имеющих несколько скрытых уровней.

12 Глубокое обучение и не только

В этой книге мы старались заострить внимание на методах и инструментах, достаточно надежных для *практического* применения. Это заставляло нас обходить стороной многообещающие, но менее зрелые библиотеки, а также библиотеки, предназначенные в первую очередь для индивидуальных исследований. Вместо этого мы отдали предпочтение инструментам, которые легко масштабируются и могут применяться для специализированного анализа как на одном компьютере, так и на больших кластерах, взаимодействующих с сотнями и тысячами пользователей. В предыдущей главе мы рассмотрели несколько таких инструментов, от модуля `multiprocessing` до мощного фреймворка Spark, которые дают возможность запускать множество моделей одновременно и делают это достаточно быстро, чтобы использовать их в крупномасштабных промышленных приложениях. В этой главе мы рассмотрим не менее значительное достижение: нейронные сети, которые быстро проникают в мир обработки естественного языка.

Как ни странно, нейронные сети являются одной из самых «старых» технологий из описываемых в этой книге, уходящей корнями в прошлое почти на 70 лет. В течение почти всего этого времени нейронные сети не могли считаться практическим методом машинного обучения. Однако ситуация резко изменилась около двух десятков лет назад благодаря трем основным достижениям: первое — значительное увеличение вычислительной мощности из-за появления графических процессоров и методов распределенных вычислений в начале 2000-х годов; второе — оптимизация скорости обучения за последнее десятилетие, о которой мы поговорим далее в этой главе; и третье — появление несколько лет тому назад библиотек с открытым исходным кодом для Python, таких как PyTorch, TensorFlow и Keras.

Полное обсуждение этих достижений выходит далеко за рамки этой книги, но мы решили дать краткий обзор нейронных сетей, потому что они имеют отношение к семейству моделей машинного обучения, рассмотренных в главах с 5-й по 9-ю. Здесь мы рассмотрим задачу классификации эмоциональной окраски, которая хорошо подходит для моделей нейронных сетей, и в заключение обсудим возможные пути развития этой области в будущем.

Прикладные нейронные сети

Как прикладные программисты, мы склонны с осторожным оптимизмом относиться к новейшим технологиям, которые хорошо смотрятся на бумаге, но могут привести к множеству проблем, когда дело дойдет до практического применения. По этой причине мы решили начать эту главу с обоснования нашего решения закончить эту книгу главой о нейронных сетях.

С практической точки зрения у нейронных сетей два основных отличия от традиционных моделей: высокая сложность и низкая скорость. Обучение нейронных сетей обычно требует больше времени, поэтому они могут замедлять итеративный рабочий процесс, который был рассмотрен в главе 5. Также нейронные сети часто более сложны, чем традиционные модели, а это означает, что их гиперпараметры труднее настраивать и для выявления ошибок моделирования приходится прикладывать больше усилий.

Тем не менее нейронные сети не только становятся все более практичными, но обещают значительный прирост качества результатов по сравнению с традиционными моделями. Это объясняется тем, что, в отличие от традиционных моделей, имеющих качественный предел, непреодолимый даже с увеличением объемов данных, нейронные сети продолжают совершенствоваться.

Нейронные модели языка

В главе 7 мы говорили о том, что модель языка можно обучить на достаточно большом и специализированном корпусе, используя вероятностный подход. Такие модели называют *символическими* моделями языка. В этой главе мы рассмотрим другой подход: нейронную, или *ассоциативную* модель языка.

Ассоциативный подход основывается на убеждении, что языковые единицы, взаимодействующие друг с другом значимыми способами, необязательно образуют последовательный контекст. Например, контекстно связанные слова могут следовать друг за другом, а могут быть разделены другими фразами,

как показано на рис. 12.1. В первом примере хорошая символическая модель выбрала бы слова «heard», «listened to» и «purchased» как наиболее вероятные продолжения фразы[1]. Но во втором примере она затруднилась бы выбрать заключительное слово, которое зависит от знания, что «Yankee Hotel Foxtrot» в первом предложении — это название альбома[2].

```
"That was one of the worst albums I've ever _____."

"Yankee Hotel Foxtrot is not cool enough for hipsters and
non mainstream enough for soccer moms. Do not buy this
_____."
```

Рис. 12.1. Непоследовательный контекст

Поскольку многие связи не поддаются прямой интерпретации, для их описания необходимо использовать какое-то промежуточное представление. Поэтому для извлечения внутренних связей ассоциативные модели часто конструируются на основе искусственных нейронных или байесовских сетей.

Традиционные *символические* модели требуют значительных усилий для управления возвратами и сглаживанием и могут предъявлять жесткие требования к объему оперативной памяти, необходимой для хранения большого количества *n*-грамм. Ассоциативный подход, напротив, позволяет масштабировать сложность модели. Фактически, основное преимущество нейронных моделей состоит в том, что они помогают избавиться от продолжительного этапа конструирования признаков за счет создания бесконечно сглаженных функций из входных данных произвольно большого объема.

В следующих нескольких разделах мы обсудим некоторые типы нейронных сетей, рассмотрим их компоненты и покажем, как можно реализовать ассоциативную модель в прикладном контексте — в данном случае для анализа эмоциональной окраски.

Искусственные нейронные сети

Нейронные сети включают в себя очень широкое и разнообразное семейство моделей, но все в той или иной степени эволюционировали из перцептрона,

[1] Фраза: «Это один из худших альбомов, которые я когда-либо...», и вероятные слова, продолжающие ее: «видел», «слушал» и «покупал». — *Примеч. пер.*

[2] Перевод второго предложения: «Yankee Hotel Foxtrot недостаточно крут для хипстеров и недостаточно популярен для мамочек. Не покупайте этот ...» — *Примеч. пер.*

механизма линейной классификации, разработанного в конце 50-х годов Фрэнком Розенблаттом (Frank Rosenblatt) в Корнелле и моделирующего обучение человеческого мозга.

В основе нейронной модели лежит семейство из нескольких компонентов, как показано на рис. 12.2 — *входной слой*, первое векторизованное представление данных, *скрытый слой*, состоящий из нейронов и синапсов, и *выходной слой* с предсказанными значениями. Синапсы внутри скрытого слоя отвечают за передачу сигналов между нейронами, которые используют нелинейную функцию активации для буферизации входящих сигналов. Синапсы применяют весовые коэффициенты к входным значениям, а функция активации определяет, достаточно ли высок уровень взвешенного входного сигнала для активации нейрона и передачи значения в следующий слой сети.

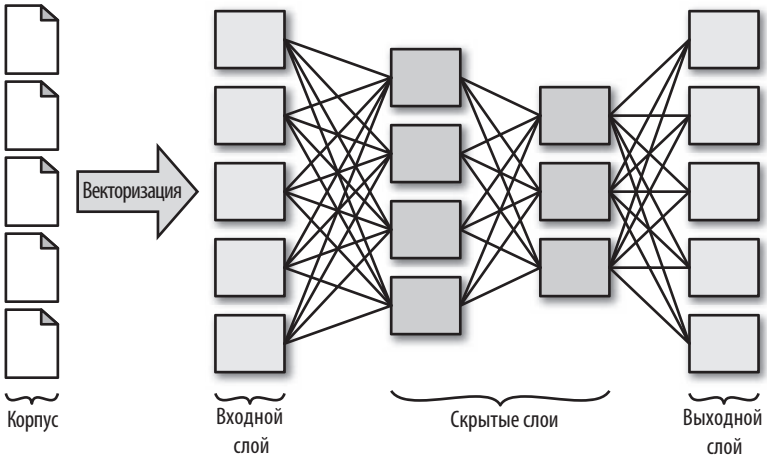


Рис. 12.2. Компоненты нейронной модели

Существует много самых разных функций активации, но вообще предпочтительнее использовать нелинейные функции, позволяющие нейронным сетям моделировать более сложные пространства решений. На практике чаще используются сигмоидальные функции, даже при том, что они могут замедлять градиентный спуск, когда угол наклона почти равен нулю. По этой причине все большей популярностью начинают пользоваться усеченные линейные преобразования (Rectified Linear Units, ReLU), которые выводят сумму взвешенных входных значений (или ноль, если сумма получилась отрицательной).

В *сети прямого распространения* сигналы передаются в одном направлении, от входного слоя к выходному. В более сложных архитектурах, таких как *рекур-*

рентные и *рекурсивные* сети, буферизованные сигналы могут комбинироваться и дублироваться между узлами слоя.

Обратное распространение (backpropagation) — это процесс передачи ошибки, вычисленной в конечном слое сети, предыдущим слоям для корректировки весов синапса и увеличения точности в следующей итерации обучения. После каждой итерации модель вычисляет градиент функции потерь и определяет направление корректировки весов.

Обучение многослойного перцептрона

Многослойные перцептроны — это одна из простейших форм искусственных нейронных сетей. В этом разделе мы обучим многослойный перцептрон с использованием библиотеки Scikit-Learn.

В качестве входных данных будет использоваться набор с 18 000 отзывов об альбомах с веб-сайта Pitchfork.com[1]; каждый отзыв включает текст отзыва, в котором рецензент обсуждает достоинства альбома, а также оценку в виде вещественного числа в диапазоне от 0 до 10. Выдержку[2] из одного такого отзыва можно видеть на рис. 12.3.

Наша цель — определить относительную эмоциональную окраску отзыва по его тексту. С помощью модуля `sklearn.neural_network` из библиотеки Scikit-Learn мы можем обучить многослойный перцептрон для классификации или регрессии, используя для этого уже знакомые нам методы `fit` и `predict`. Попробуем реализовать оба подхода, регрессию для предсказания фактической числовой оценки альбома и классификацию для предсказания эмоциональной оценки: «terrible» (плохо), «okay» (нормально), «good» (хорошо) или «amazing» (отлично).

```
“Ghost Stories is unmistakably Coldplay's "breakup album,"
a subdued work that finds Chris Martin and his band
crisply moping through mid-tempo soundscapes and fuzzy
electronic touches that have the visceral impact of a down
comforter tumbling down a flight of stairs… Rating: 4.4”

                                    -  Larry Fitzmaurice
                                          May 20, 2014
```

Рис. 12.3. Пример отзыва с сайта Pitchfork

[1] Cond Nast, Pitchfork: *The Most Trusted Voice In Music* (2018), http://bit.ly/2GNLO7F
[2] Larry Fitzmaurice, *Review of Coldplay's Ghost Stories* (2014), http://bit.ly/2GQL1ms

Сначала определим функцию `documents`, извлекающую документы, маркированные частями речи, из объекта чтения корпуса, функцию `continuous` для получения исходного числового рейтинга альбома и функцию `categorical`, использующую метод `digitize` из NumPy для преобразования рейтинга в одну из четырех эмоциональных категорий:

```
import numpy as np

def documents(corpus):
    return list(corpus.reviews())

def continuous(corpus):
    return list(corpus.scores())

def make_categorical(corpus):
    """
    terrible : 0.0 < y <= 3.0
    okay     : 3.0 < y <= 5.0
    great    : 5.0 < y <= 7.0
    amazing  : 7.0 < y <= 10.1
    """
    return np.digitize(continuous(corpus), [0.0, 3.0, 5.0, 7.0, 10.1])
```

Затем добавим определение функции `train_model`, принимающей путь к корпусу, объект `Estimator` из Scikit-Learn и именованные аргументы с признаком непрерывности меток, необязательным каталогом для сохранения обученной модели и количеством блоков для перекрестной проверки.

Наша функция должна создать объект чтения корпуса, вызвать функцию `documents`, а также `continuous` или `make_categorical`, чтобы получить входные значения `X` и целевые значения `y`. Затем вычислить оценки перекрестной проверки, обучить и сохранить модель с помощью утилиты `joblib` из библиотеки Scikit-Learn и вернуть оценки:

```
from sklearn.externals import joblib
from sklearn.model_selection import cross_val_score

def train_model(path, model, continuous = True, saveto = None, cv = 12):
    """
    Обучает модель на корпусе, находящемся по указанному пути path;
    Вычисляет оценки перекрестной проверки, используя параметр cv,
    Затем обучает модель на всем объеме данных.
    Возвращает оценки.
    """
    # Загрузить данные из корпуса и метки для классификации
    corpus = PickledReviewsReader(path)
    X = documents(corpus)
    if continuous:
        y = continuous(corpus)
```

```
        scoring = 'r2_score'
    else:
        y = make_categorical(corpus)
        scoring = 'f1_score'

    # Вычислить оценки перекрестной проверки
    scores = cross_val_score(model, X, y, cv = cv, scoring = scoring)

    # Записать на диск, если необходимо
    if saveto:
        joblib.dump(model, saveto)

    # Обучить модель на полном объеме данных
    model.fit(X, y)

    # Вернуть оценки
    return scores
```

Подобно другим объектам `Estimator` из Scikit-Learn, `MLPRegressor` и `MLPClassifier` принимают массивы NumPy вещественных чисел. Несмотря на то что массивы могут быть плотными или разреженными, входные векторы лучше всего масштабировать с использованием прямого или стандартного частотного кодирования.

Определим обучаемые модели, создав два конвейера, выполняющие нормализацию текста, векторизацию и моделирование:

```
if __name__ == '__main__':
    from transformer import TextNormalizer
    from reader import PickledReviewsReader

    from sklearn.pipeline import Pipeline
    from sklearn.neural_network import MLPRegressor, MLPClassifier
    from sklearn.feature_extraction.text import TfidfVectorizer

    # Путь к обработанному корпусу отзывов, маркированному частями речи
    cpath = '../review_corpus_proc'

    regressor = Pipeline([
        ('norm', TextNormalizer()),
        ('tfidf', TfidfVectorizer()),
        ('ann', MLPRegressor(hidden_layer_sizes = [500,150], verbose = True))
    ])
    regression_scores = train_model(cpath, regressor, continuous = True)

    classifier = Pipeline([
        ('norm', TextNormalizer()),
        ('tfidf', TfidfVectorizer()),
        ('ann', MLPClassifier(hidden_layer_sizes = [500,150], verbose = True))
    ])
    classifer_scores = train_model(cpath, classifier, continuous = False)
```

Подобно выбору значения *k* для метода кластеризации *k*-средних, выбор оптимального количества и размеров скрытых слоев для начального прототипа нейронной сети — скорее искусство, чем наука. Чем больше слоев и узлов в них, тем сложнее получится модель, а чем сложнее модель, тем больше данных требуется для ее обучения. Хорошее эмпирическое правило: начинать с простой модели (количество узлов в начальном слое не должно превышать количества экземпляров, и в сети не должно быть больше двух слоев) и затем итеративно увеличивать сложность, пока перекрестная проверка по *k*-блокам не обнаружит факт переобучения.

Библиотека Scikit-Learn предлагает большое количество параметров для настройки нейронных сетей. Например, по умолчанию оба класса, `MLPRegressor` и `MLPClassifier`, используют функцию активации ReLU, которую можно изменить с помощью параметра `activation`, и алгоритм стохастического градиентного спуска для минимизации функции потерь, который можно изменить с помощью параметра `solver`:

```
Mean score for MLPRegressor: 0.27290534221341
Mean score for MLPClassifier: 0.7115215174722
```

Как показывает оценка R2, модель `MLPRegressor` получилась очень слабой и продемонстрировала очень низкое качество обучения. Регрессия выигрывает от сокращения числа размерностей, особенно в отношении количества экземпляров. Мы можем судить об этом через призму «проклятия размерности»; отзывы на сайте Pitchfork в среднем содержат около 1000 слов, и каждое слово добавляет еще одно измерение в наше пространство решений. Поскольку наш корпус включает примерно 18 000 отзывов, нашей модели `MLPRegressor` просто не хватило экземпляров для предсказания числовой оценки с достаточно высокой степенью точности.

Однако, как можно видеть, модель `MLPClassifier` показала намного лучшие результаты и, возможно, стоит попробовать повозиться с ее настройками. Чтобы улучшить качество `MLPClassifier`, можно поэкспериментировать с увеличением и уменьшением сложности модели. Увеличить сложность можно добавлением слоев и нейронов в `hidden_layer_sizes`. Также можно увеличить параметр `max_iter`, определяющий количество эпох обучения, и дать модели больше времени на обучение с применением процесса обратного распространения.

Уменьшить сложность можно удалением слоев, уменьшением числа нейронов или добавлением критерия регуляризации в функцию потерь с помощью параметра `alpha`, который, подобно `sklearn.linear_model.RidgeRegression`, искусственно сократит число параметров, чтобы помочь избежать переобучения.

Прикладной интерфейс библиотеки Scikit-Learn очень удобен для создания простых нейронных моделей. Но, как будет показано в следующем разделе,

такие библиотеки, как TensorFlow, предлагают намного больше возможностей для гибкой настройки архитектуры модели, а также более высокую скорость вычислений за счет использования GPU.

Архитектуры глубокого обучения

Под термином *глубокое обучение* часто подразумеваются такие модели, как рекуррентные нейронные сети (Recurrent Neural Networks, RNN), сети с долгой краткосрочной памятью (Long Short-Term Memory, LSTM), рекурсивные тензорные нейронные сети (Recursive Neural Tensor Networks, RNTN), сверточные нейронные сети (Convolutional Neural Networks, CNN или ConvNets) и генеративно-состязательные сети (Generative Adversarial Networks, GAN), получившие большую популярность в последние годы.

Хотя многие под глубокими нейронными сетями подразумевают нейронные сети с несколькими внутренними слоями, термин «глубокое обучение» в действительности мало чем отличается от искусственных нейронных сетей. Однако разные архитектуры реализуют уникальные возможности в слоях, что дает им возможность моделировать очень сложные данные.

Сверточные нейронные сети (Convolutional Neural Networks, CNN), например, объединяют многослойные перцептроны со сверточным слоем, который итеративно строит карту для извлечения важных признаков, а также имеют этап свертки, уменьшающий размерность признаков, но сохраняющий наиболее информативные их компоненты. Сверточные сети очень эффективны для моделирования изображений и решения задач классификации и обобщения.

Для моделирования данных на естественном языке лучше всего подходят разновидности рекуррентных нейронных сетей (Recurrent Neural Networks, RNN), такие как сети с долгой краткосрочной памятью (Long Short-Term Memory, LSTM). Архитектура RNN позволяет модели поддерживать порядок слов в предложении и выявлять долгосрочные зависимости. Сети LSTM, например, реализуют управляемые ячейки, которые дают возможность «запоминания» и «забывания». Разновидности этой модели часто используются в машинном переводе и в задачах генерирования текстов на естественном языке.

TensorFlow: фреймворк для глубокого обучения

TensorFlow — распределенный вычислительный движок, реализующий основу для глубокого обучения. Разработан в компании Google с целью организации параллельного обучения моделей, не только на графических процессорах (GPU), но и на множестве компьютеров в сети. Исходный код движка был

открыт в ноябре 2015 года и с того времени стал одной из самых популярных среди общедоступных библиотек глубокого обучения.

Для работы с TensorFlow пользователь должен хорошо знать архитектуры нейронных сетей и ориентироваться на построение графов потоков данных со значительной степенью настройки. Использование TensorFlow начинается с определения каждого слоя и всех гиперпараметров, после чего производится компиляция всех этапов в статический граф и начинается сеанс обучения. Это упрощает управление моделями и их оптимизацию с ростом сложности, но также и усложняет быстрое прототипирование.

По сути, модели глубокого обучения — всего лишь цепочки функций, а это означает, что многие библиотеки глубокого обучения поддерживают функциональный и избыточно многословный декларативный стиль программирования. Как следствие, быстро эволюционировала экосистема других библиотек, включая Keras, TF-slim, TFLearn и SkFlow, с более абстрактным, объектно-ориентированным интерфейсом. В следующем разделе мы покажем, как пользоваться фреймворком TensorFlow посредством Keras API.

Keras: прикладной интерфейс для глубокого обучения

Несмотря на частое объединение с фреймворками глубокого обучения, такими как TensorFlow, Caffe, Theano и PyTorch, библиотека Keras предлагает более обобщенный программный интерфейс для глубокого обучения. Первоначально библиотека Keras разрабатывалась для работы с фреймворком Theano, но после открытия исходных текстов и взрывного роста популярности TensorFlow библиотека Keras быстро стала одним из основных инструментов для многих пользователей TensorFlow и была включена в ядро TensorFlow в начале 2017 года.

В Keras все сущее является объектом, что делает ее особенно удобным инструментом для прототипирования. Чтобы получить аналог многослойного классифицирующего перцептрона, продемонстрированного в предыдущем разделе, определим функцию `build_network`, которая создает экземпляр модели `Sequential` из Keras, добавляет два скрытых слоя `Dense` (*полносвязанных*), из которых первый содержит 500 узлов и второй — 150, и оба используют усеченные линейные преобразования (ReLU) в качестве функции активации. Обратите внимание на то, что в первый скрытый слой передается кортеж `input_shape`, описывающий форму входного слоя.

Для выходного слоя нужно определить функцию уплотнения измерений из предыдущего слоя в пространство категорий для классификации. В данном случае используется функция *softmax*, популярный выбор в задачах обработки

естественного языка, потому что представляет распределение по категориям, соответствующим лексемам в нашем корпусе. Затем `build_network` вызывает метод `compile`, передав ему функции потерь (`loss`) и оптимизации (`optimizer`) для алгоритма градиентного спуска, и возвращает скомпилированную сеть:

```
from keras.layers import Dense
from keras.models import Sequential

N_FEATURES = 5000
N_CLASSES = 4

def build_network():
    """
    Возвращает скомпилированную нейронную сеть
    """
    nn = Sequential()
    nn.add(Dense(500, activation = 'relu', input_shape = (N_FEATURES,)))
    nn.add(Dense(150, activation = 'relu'))
    nn.add(Dense(N_CLASSES, activation = 'softmax'))
    nn.compile(
        loss = 'categorical_crossentropy',
        optimizer = 'adam',
        metrics = ['accuracy']
    )
    return nn
```

Модуль `keras.wrappers.scikit_learn` экспортирует классы `KerasClassifier` и `KerasRegressor` — две обертки, реализующие интерфейс Scikit-Learn. Благодаря этому модели `Sequential` из Keras можно внедрять в конвейеры Scikit-Learn — `Pipeline` или `Gridsearch`.

Чтобы задействовать `KerasClassifier`, сконструируем конвейер, аналогичный представленному в предыдущем разделе, с нормализатором `TextNormalizer` (описанным в разделе «Создание своего преобразователя для нормализации текста» главы 4) и векторизатором `TfidfVectorizer`. Используем параметр `max_features` векторизатора для передачи глобальной переменной `N_FEATURES`, гарантировав тем самым соответствие размерности векторов и форме `input_shape` скомпилированной нейронной сети.

Для тех, кто привык пользоваться объектами `Estimator` из библиотеки Scikit-Learn, которые поддерживают разумные значения по умолчанию для гиперпараметров, построение моделей глубокого обучения первое время может вызывать сложности. Keras и TensorFlow не делают почти никаких предположений о форме и размере входных данных и не будут пытаться определить гиперпараметры пространства решений. Тем не менее знакомство с особенностями конструирования моделей TensorFlow через Keras API даст нам возможность не только создавать свои модели, но и обучать их намного быстрее, чем модели `sklearn.neural_network`.

Наконец, добавим в конвейер `KerasClassifier`, передав функции создания сети, желаемое число эпох и необязательный размер пакета. Обратите внимание на то, что мы должны передать ссылку на функцию `build_network`, а не экземпляр скомпилированной сети, возвращаемый ею:

```
if __name__ == '__main__':
    from sklearn.pipeline import Pipeline
    from transformer import TextNormalizer
    from keras.wrappers.scikit_learn import KerasClassifier
    from sklearn.feature_extraction.text import TfidfVectorizer

    pipeline = Pipeline([
        ('norm', TextNormalizer()),
        ('vect', TfidfVectorizer(max_features = N_FEATURES)),
        ('nn', KerasClassifier(build_fn = build_network,
                               epochs = 200,
                               batch_size = 128))
    ])
```

Теперь можно запустить полученный конвейер, использовав немного измененную версию функции `train_model`. Так же, как в предыдущем разделе, эта функция создает объект чтения корпуса, вызывает `documents` и `make_categorical`, чтобы получить входные и целевые значения `X` и `y`, вычисляет оценки перекрестной проверки, обучает и сохраняет модель и возвращает оценки.

К сожалению, на момент написания этих строк сохранение конвейера Scikit-Learn, обертывающего модель Keras, было несколько затруднено, потому что компонент нейронной сети должен сохраняться вызовом специализированного метода `save` из библиотеки Keras в формате иерархических данных (*.h5*). Поэтому в качестве обходного решения мы сначала сохраняем модель, обратившись к ней по индексу в `Pipeline`, а затем записываем на диск остальной обученный конвейер с помощью `joblib`:

```
def train_model(path, model, saveto = None, cv = 12):
    """
    Обучает модель на всем корпусе, находящемся по указанному пути path.
    Если задан словарь saveto, сохраняет на диск модель Keras и компоненты
    конвейера Sklearn по отдельности. Возвращает оценки.
    """
    corpus = PickledReviewsReader(path)
    X = documents(corpus)
    y = make_categorical(corpus)

    scores = cross_val_score(model, X, y, cv = cv, scoring = 'accuracy',
             n_jobs = -1)
    model.fit(X, y)
```

```
    if saveto:
        model.steps[-1][1].model.save(saveto['keras_model'])
        model.steps.pop(-1)
        joblib.dump(model, saveto['sklearn_pipe'])

    return scores
```

Теперь вернемся к инструкции `if __name__ == '__main__'` и передадим путь к корпусу и словарь с путями для сохранения модели:

```
cpath = '../review_corpus_proc'
mpath = {
    'keras_model'  : 'keras_nn.h5',
    'sklearn_pipe' : 'pipeline.pkl'
}
scores = train_model(cpath, pipeline, saveto = mpath, cv = 12)
```

С 5000 входных признаков наш классификатор Keras на основе нейронной сети справился весьма неплохо; в целом модель сумела определить эмоциональную окраску отзыва, выбирая между «terrible» (плохо), «okay» (нормально), «good» (хорошо) или «amazing» (отлично):

```
Mean score for KerasClassifier: 0.70533893018807
```

Даже при том, что средняя оценка классификатора Scikit-Learn оказалась несколько выше, обучение модели Keras заняло у нас всего два часа на MacBook Pro, или примерно одну шестую часть времени обучения `MLPClassifier` из Scikit-Learn. Благодаря этому мы сможем выполнить дополнительную настройку модели Keras (например, увеличить число скрытых слоев и узлов, опробовать другие функции активации и потерь, произвести случайное «прореживание», то есть выборочно обнулить некоторые входные данные, чтобы избежать переобучения, и т. д.) и намного быстрее усовершенствовать ее.

Однако главной проблемой нашей модели является малый объем набора данных; нейронные сети обычно превосходят по качеству все другие семейства моделей машинного обучения, но только когда объем обучающих данных превысит некоторый порог (подробнее эта проблема обсуждается в видеолекции Эндрю Ына «Why Is Deep Learning Taking Off?»[1]). В следующем разделе мы исследуем намного больший объем данных, а также поэкспериментируем с некоторыми синтаксическими признаками, которые видели в главах 7 и 10, чтобы улучшить отношение сигнал/шум.

[1] Andrew Ng, *Why is Deep Learning Taking Off?* (2017), http://bit.ly/2JJ93kU

Анализ эмоциональной окраски

До сих пор мы рассматривали отзывы как обычные «мешки слов», что не редкость для нейронных сетей. Функции активации обычно требуют, чтобы входные данные были представлены числами в диапазоне `[0,1]` или `[-1,1]`, что делает прямое кодирование удобным методом векторизации.

Однако модель «мешка слов» плохо подходит для более сложных задач анализа текста, потому что она выделяет более широкие и важные элементы текста вместо микросигналов, описывающих значимые коррективы или изменения. Приемы генерирования текста на естественном языке, которые мы кратко рассмотрели в главах 7 и 10, относятся к категории приложений, в которых моделей типа «мешок слов» часто недостаточно для выявления тонкостей речевых шаблонов. Другим подобным примером может служить анализ эмоциональной окраски, где относительная положительность или отрицательность высказывания является функцией сложных взаимосвязей между положительными или отрицательными модификаторами и нелексическими факторами, такими как сарказм, гипербола и символизм.

В главе 1 мы кратко представили анализ эмоциональной окраски, чтобы подчеркнуть важность контекстных признаков. Если такие языковые признаки, как пол, часто напрямую закодированы в структуре языка, то эмоциональная окраска слишком сложна для кодирования на уровне лексем. Возьмем для примера один из отзывов, оставленный на сайте Amazon покупателем садового инвентаря:

> *I used to use a really primitive manual edger that I inherited from my father. Blistered hands and time wasted, I decided there had to be a better way and this is surely it. Edging is one of those necessary evils if you want a great looking house. I don't edge every time I mow. Usually I do it every other time. The first time out after a long winter, edging usually takes a little longer. After that, edging is a snap because you are basically in maintenance mode. I also use this around my landscaping and flower beds with equally great results. The blade on the Edge Hog is easily replaceable and the tell tale sign to replace it is when the edge starts to look a little rough and the machine seems slower.*[1]
>
> *— Отзыв с сайта Amazon*

[1] Перевод: «Раньше я пользовался простенькой ручной машиной для окантовки газонов, доставшейся мне в наследство от отца. Набив мозоли и потратив впустую массу времени, я пришел к мысли, что должен быть какой-то другой способ, менее трудоемкий, и я нашел его. Окантовка газонов — одно из неизбежных зол, с которым вам придется мириться, если хотите, чтобы ваш дом выглядел красиво. Окантовку я выполняю не всякий раз, когда кошу. Обычно я делаю это отдельно. Первый раз — после зимы — окантовка занимает

Если попробовать оценить этот отзыв, подсчитывая «положительные» и «отрицательные» слова, как в случае со словами, определяющими половую принадлежность в главе 1, какую оценку вы ожидали бы получить? Несмотря на большое количество слов с кажущимся негативным оттенком (например, «primitive» (примитивной), «blistered» (мозоли), «wasted» (впустую), «rough» (неровной), «slower» (медленнее)) этот текст соответствует отзыву с оценкой в 5 звезд — наивысший возможный рейтинг!

Даже если взять единственное предложение из отзыва (например, «Edging is one of those necessary evils if you want a great looking house»[1]), легко заметить, как положительные и отрицательные конструкции изменяют друг друга, создавая эффект обращения эмоциональной окраски на противоположную или усиливая ее. На рис. 12.4 показано, как синтаксические блоки совместно влияют на общую эмоциональную окраску предложения.

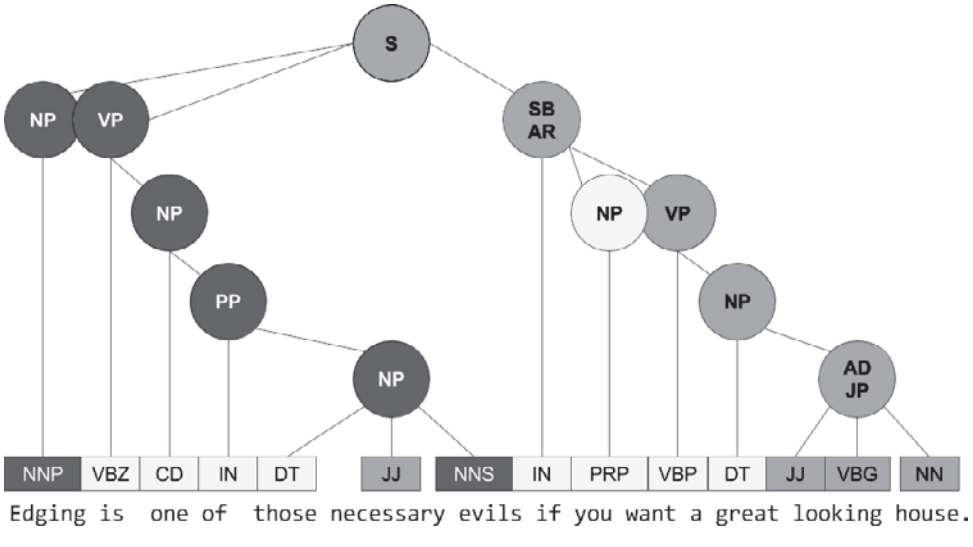


Рис. 12.4. Синтаксический анализ

обычно чуть больше времени. После этого она дается проще, потому что остается лишь поддерживать аккуратный кант. Я также делаю окантовку лужайки и клумб с цветами с таким же превосходным результатом. Ножи в Edge Hog легко заменяются, а время замены легко определяется по тому, как кромка начинает выглядеть немного неровной, а машина начинает работать медленнее». — *Примеч. пер.*

[1] Перевод: «Окантовка газонов — одно из неизбежных зол, с которым вам придется мириться, если хотите, чтобы ваш дом выглядел красиво». — *Примеч. пер.*

Глубокий анализ структуры

В 2013 году Ричард Сочер (Richard Socher) с коллегами[1] предложил подход синтаксической сегментации к анализу эмоциональной окраски. Они предложили метод классификации эмоций, основанный на использовании банка синтаксических деревьев фраз (то есть корпуса с синтаксическими аннотациями), позволяющий более точно определять общую эмоциональную окраску высказывания. Они показали, как классификация предложений фраза за фразой вместо всего высказывания целиком в конечном итоге приводит к значительному увеличению точности, в частности, потому, что дает возможность комплексного моделирования эффекта отрицания на разных уровнях дерева фраз, как было показано на рис. 12.4.

Сочер представил также новый вид нейронных сетей для моделирования, *рекурсивные тензорные нейронные сети* (Recursive Neural Tensor Networks, RNTN). В отличие от обычных рекуррентных моделей и моделей с прямым распространением, рекурсивные сети предполагают наличие у данных иерархической структуры, а процесс обучения сводится к обходу дерева. Эти модели используют векторные отображения, подобные тем, что были показаны в главе 4, для представления слов, или «листьев» дерева, а также функцию композиционности, которая определяет порядок рекурсивного объединения векторизованных листьев для представления фраз.

Как и объект `KerasClassifier`, сконструированный нами выше в этой главе, рекурсивные нейронные сети используют в скрытых слоях функции активации для моделирования нелинейности, а также функцию сжатия для уменьшения размерности конечного слоя до заданного количества классов (обычно двух в случае с анализом эмоциональной окраски), такую как *softmax*.

Важно отметить, что эффективность модели Сочера тоже во многом зависит от объема обучающих данных. Члены команды Сочера извлекли все синтаксически возможные фразы из тысяч отзывов к фильмам на *rottentomatoes.com*, которые затем вручную оценили с использованием интерфейса маркировки на Amazon Mechanical Turk, и таким способом сконструировали обучающий набор данных. Получившийся результат в виде оригинального кода для Matlab и множества визуальных представлений деревьев анализа эмоциональной окраски доступен как Stanford Sentiment Treebank[2].

---

[1] Richard Socher, Alex Perelygin, Jean Y. Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng, and Christopher Potts, *Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank* (2013), http://bit.ly/2GQL2Xy

[2] Jean Wu, Richard Socher, Rukmani Ravisundaram, and Tayyab Tariq, *Stanford Sentiment Treebank* (2013), https://stanford.io/2GQL3uA

В следующем разделе мы реализуем классификатор эмоциональной окраски, заимствующий некоторые идеи из работы Сочера и использующий структуру языка для усиления сигнала.

Определение эмоциональной окраски по мешку фраз

В предыдущем разделе мы использовали библиотеку Keras для обучения простого многослойного перцептрона, довольно успешно предсказывающего оценки отзывов к музыкальным альбомам на Pitchfork на основе слов, содержащихся в отзыве. В этом разделе мы попробуем создать более сложную модель, использовав корпус большего размера и подход на основе «мешка фраз».

В качестве корпуса используем подмножество отзывов к товарам на сайте Amazon из корпуса, созданного Джулианом Маколи (Julian McAuley) из Калифорнийского университета в Сан-Диего[1]. Полный набор данных содержит более миллиона отзывов к фильмам и телепередачам, каждый из которых включает текст отзыва и оценку. Оценки определяют уровень рейтинга, от низшего (1) до высшего (5).

Наша модель предполагает, что большая часть семантической информации в предложениях содержится в виде небольших синтаксических субструктур. Вместо подхода «мешок слов» мы реализуем облегченный метод извлечения синтаксической структуры отзывов, добавив новый класс `KeyphraseExtractor`, являющийся модификацией приема извлечения ключевых фраз из главы 7, с помощью которого преобразуем корпус отзывов в векторное представление ключевых фраз.

В частности, для определения грамматики мы применим регулярное выражение, использующее теги частей речи для идентификации наречных фраз («without care» (беззаботно)) и фраз с прилагательными («terribly helpful» (ужасно полезный)). С его помощью мы сможем разбить текст на ключевые фразы с помощью `RegexpParser` из NLTK. Мы будем использовать нейронную сеть, которая требует заранее определить общее количество признаков (то есть лексикон) и длину каждого документа, с этой целью мы добавим соответствующие параметры в функцию `__init__`.

Настройка гиперпараметров для максимального сокращения словаря и длины документов зависит от данных и требует применения итеративного подхода к проектированию признаков. Для начала можно определить общее количество уникальных ключевых слов и присвоить параметру `nfeatures` значение меньше этого числа. Чтобы выбрать длину документа, можно подсчитать количество ключевых слов в каждом из них и взять среднее значение.

[1] Julian McAuley, *Amazon product data* (2016), http://bit.ly/2GQL2H2

```
class KeyphraseExtractor(BaseEstimator, TransformerMixin):
    """
    Извлекает наречные и прилагательные словосочетания и преобразует
    документы в списки словосочетаний, при этом общее количество
    словосочетаний ограничивается параметром nfeatures, а длина документа
    ограничивается параметром doclen
    """
    def __init__(self, nfeatures = 100000, doclen = 60):
        self.grammar = r'KT: {(<RB.> <JJ.*>|<VB.*>|<RB.*>)|(<JJ> <NN.*>)}'
        self.chunker = RegexpParser(self.grammar)
        self.nfeatures = nfeatures
        self.doclen = doclen
```

Чтобы еще больше уменьшить сложность, добавим метод `normalize`, удаляющий знаки препинания из каждого лексемизированного и размеченного предложения и преобразующий буквы в нижний регистр, а также метод `extract_candidate_phrases`, извлекающий словосочетания из каждого предложения с использованием нашей грамматики:

```
...
    def normalize(self, sent):
        is_punct = lambda word: all(unicat(c).startswith('P') for c in word)
        sent = filter(lambda t: not is_punct(t[0]), sent)
        sent = map(lambda t: (t[0].lower(), t[1]), sent)
        return list(sent)

    def extract_candidate_phrases(self, sents):
        """
        Анализирует предложения в документе с использованием нашего парсера,
        образованного нашей грамматикой, преобразует дерево синтаксического
        анализа в маркированную последовательность. Извлекает фразы, добавляя
        пробелы, и возвращает документ в виде списка фраз.
        """
        for sent in sents:
            sent = self.normalize(sent)
            if not sent: continue
            chunks = tree2conlltags(self.chunker.parse(sent))
            phrases = [
                " ".join(word for word, pos, chunk in group).lower()
                for key, group in groupby(
                    chunks, lambda term: term[-1] ! = 'O'
                ) if key
            ]
            for phrase in phrases:
                yield phrase
```

Для передачи размера входного слоя в нейронную сеть определим метод `get_lexicon`, извлекающий ключевые фразы из всех отзывов и собирающий лексикон с желаемым числом признаков. Наконец, добавим метод `clip`, оставляющий в документах только ключевые фразы из лексикона:

```
...
    def get_lexicon(self, keydocs):
        """
        Собирает лексикон с размером nfeatures
        """
        keyphrases = [keyphrase for doc in keydocs for keyphrase in doc]
        fdist = FreqDist(keyphrases)
        counts = fdist.most_common(self.nfeatures)
        lexicon = [phrase for phrase, count in counts]
        return {phrase: idx+1 for idx, phrase in enumerate(lexicon)}

    def clip(self, keydoc, lexicon):
        """
        Удаляет из документа фразы, отсутствующие в лексиконе
        """
        return [lexicon[keyphrase] for keyphrase in keydoc
                if keyphrase in lexicon.keys()]
```

В отличие от метода `fit`, не делающего ничего и просто возвращающего ссылку `self`, метод `transform` проделывает всю основную работу по извлечению фраз, созданию лексикона, усечению документов и дополнению каждого до желаемого размера с помощью функции `sequence.pad_sequences` из библиотеки Keras:

```
from keras.preprocessing import sequence

...
    def fit(self, documents, y = None):
        return self

    def transform(self, documents):
        docs = [list(self.extract_candidate_phrases(doc)) for doc in
                documents]
        lexicon = self.get_lexicon(docs)
        clipped = [list(self.clip(doc, lexicon)) for doc in docs]
        return sequence.pad_sequences(clipped, maxlen = self.doclen)
```

Теперь напишем функцию для создания нейронной сети; в данном случае мы сконструируем сеть с долгой краткосрочной памятью (Long Short-Term Memory, LSTM).

Наша сеть LSTM начинается со слоя `Embedding`, который создает векторные представления из документов, состоящих из ключевых фраз, и принимает три параметра: общее число признаков (то есть общий размер лексикона), желаемая размерность векторных представлений и входная длина `input_length` каждого документа с ключевыми фразами. Слой `LSTM` с 200 узлами располагается между двумя слоями `Dropout`, которые в каждом цикле обучения случайным образом

присваивают части входных данных значение `0`, чтобы помочь предотвратить переобучение. Конечный слой определяет число ожидаемых категорий для классификации:

```
N_FEATURES = 100000
N_CLASSES = 2
DOC_LEN = 60

def build_lstm():
    lstm = Sequential()
    lstm.add(Embedding(N_FEATURES, 128, input_length = DOC_LEN))
    lstm.add(Dropout(0.4))
    lstm.add(LSTM(units = 200, recurrent_dropout = 0.2, dropout = 0.2))
    lstm.add(Dropout(0.2))
    lstm.add(Dense(N_CLASSES, activation = 'sigmoid'))
    lstm.compile(
        loss = 'categorical_crossentropy',
        optimizer = 'adam',
        metrics = ['accuracy']
    )
    return lstm
```

Мы будем обучать нашу модель как бинарный классификатор (как это сделал Сочер в своей реализации), поэтому добавим функцию `binarize` для разделения меток на две категории для использования в функции `train_model`. Эти две категории примерно соответствуют понятиям «liked it» (нравится) и «hated it» (не нравится):

```
def binarize(corpus):
    """
    hated it : 0.0 < y < = 3.0
    liked it : 3.0 < y < = 5.1
    """
    return np.digitize(continuous(corpus), [0.0, 3.0, 5.1])

def train_model(path, model, cv = 12, **kwargs):
    corpus = PickledAmazonReviewsReader(path)
    X = documents(corpus)
    y = binarize(corpus)
    scores = cross_val_score(model, X, y, cv = cv, scoring = 'accuracy')
    model.fit(X, y)

    ...

    return scores
```

Наконец, определим входные данные и компоненты конвейера `Pipeline` и вызовем функцию `train_model`, чтобы получить оценки перекрестной проверки:

```
if __name__ == '__main__':
    am_path = '../am_reviews_proc'
    pipeline = Pipeline([
        ('keyphrases', KeyphraseExtractor()),
        ('lstm', KerasClassifier(build_fn = build_nn,
                                 epochs = 20,
                                 batch_size = 128))
    ])

    scores = train_model(am_path, pipeline, cv = 12)
```

```
Mean score: 0.8252357452734355
```

Как показывают предварительные результаты, мы получили на удивление эффективную модель, предположив, что извлечение ключевых фраз является действенным способом снижения размерности текстовых данных без значительной потери семантической информации, закодированной в синтаксических структурах.

Будущее (почти) наступило

Наступило очень интересное время в развитии технологий анализа текста, не только потому, что появилось множество новых применений машинному обучению на текстовых данных, но также благодаря появлению технологий, обеспечивающих практическую поддержку этим применениям. Успехи в области аппаратного обеспечения, достигнутые за последние несколько десятилетий, и появление удачных программных реализаций с открытым исходным кодом в последние годы перенесли нейронные сети из области академических исследований в область практического применения. Поэтому прикладной анализ текста должен быть готов интегрировать поддержку новой аппаратуры и результаты академических исследований в существующие программные решения, чтобы оставаться современным и актуальным. Язык меняется, и вместе с ним должны меняться подходы к его обработке.

Одни из самых сложных задач в области обработки естественного языка — машинный перевод, обобщение, перефразирование, поиск ответов на вопросы и диалог — в настоящее время решаются с привлечением нейронных сетей. Все чаще исследователи, занимающиеся моделями глубокого обучения, направляют свои усилия не на обработку языка, а на его понимание.

Несмотря на широкую известность современных коммерческих приложений, их все еще слишком мало. Такие технологии, как механизм Alexa для распознавания речи, приложение машинного перевода Google Translate и инстру-

мент добавления титров к изображениям в Facebook для слабовидящих, все чаще объединяют текст, звуки и изображения инновационными способами. В ближайшие годы ожидается появление более совершенных алгоритмов и гибридных моделей, которые, например, смогут объединить классификацию изображений с созданием текстов на естественном языке или распознавание речи с машинным переводом.

Вместе с тем можно ожидать внедрения меньшего по масштабам применения анализа текста в обычные приложения для увеличения удобства пользователей — средств автоматического дополнения текста, диалоговых агентов, улучшенных механизмов поиска рекомендаций и т. д. Такие приложения будут опираться не только (и не столько) на массивные наборы данных, но также на специализированные корпусы, ориентированные на узкий круг применений.

В настоящее время не так много компаний, обладающих достаточными объемами данных, квалифицированным персоналом и высокопроизводительной вычислительной инфраструктурой, способных сделать применение нейронных сетей практичным и экономически выгодным, но и это положение дел постепенно меняется. Однако для большинства исследователей данных и разработчиков приложений будущее прикладного анализа текста будет заключаться не столько в поиске новых алгоритмов, сколько в выявлении интересных задач в повседневной жизни и применении надежных и масштабируемых инструментов и приемов для создания небольших, но ценных функций, отличающих современные приложения от предыдущих поколений.

Глоссарий

агломеративный (agglomerative)

Агломеративная кластеризация — разновидность приема иерархической кластеризации, в котором создание кластеров начинается с отдельных экземпляров и объединение производится итеративно, пока все они не будут принадлежать одной группе.

анализ графов (graph analytics)

Анализ графов — один из подходов к анализу текста, использующий структуру графов и численные методы из теории графов для исследования взаимоотношений между сущностями или элементами текста.

анализ эмоциональной окраски (sentiment analysis)

Анализ эмоциональной окраски — это процесс идентификации и классификации эмоциональной полярности, выраженной в высказывании — например, для определения относительной положительности или отрицательности чувств пишущего или говорящего.

ассоциативная модель языка (connectionist language model)

Ассоциативная модель языка утверждает, что значимые взаимодействия между единицами языка не всегда кодируются последовательным контекстом, но могут извлекаться нейронными сетями.

вектор абзаца (paragraph vector)

Вектор абзаца — алгоритм обучения без учителя, который создает пространство признаков фиксированной длины из документов разной длины и расширяет алгоритм *word2vec* до уровня экземпляров документов.

векторизация (vectorize/vectorization)

Векторизация — процесс преобразования нечисловых данных (например, текста, изображений и т. д.) в векторное представление, к которому можно применить методы машинного обучения.

визуализатор (visualizer)

Визуализатор — инструмент визуальной диагностики, позволяющий человеку управлять процессами анализа признаков, выбора модели и настройки гиперпараметров (например, тройка выбора модели).

высказывание (utterance)

Высказывания — это короткие, законченные цепочки устной или письменной речи. В анализе речи высказывания обычно разграничиваются отчетливо заметными паузами. В анализе текста высказывания выделяются знаками препинания, предназначенными для передачи пауз.

вытянутый хвост (long tail)

Вытянутый хвост, или распределение Ципфа (Zipfian distribution), отражает большое количество вхождений далеко от центра частотного распределения.

гапаксы (hapaxes/hapax legomena)

Гапакс — это термин, встречающийся в корпусе только один раз.

гиперпараметр (hyperparameter)

В машинном обучении гиперпараметрами называют параметры, определяющие особенности работы модели; они определяются на этапе создания экземпляра модели и не связаны напрямую с обучением. Примерами могут служить параметр `alpha` (штраф) для регуляризации, функция ядра в методе опорных векторов, количество листьев или глубина дерева решений, количество соседей в классификаторе методом ближайших соседей или количество кластеров в кластеризации методом *k*-средних.

глубокое обучение (deep learning)

Под глубоким обучением в широком смысле подразумевается большое семейство нейронных сетей, содержащих несколько взаимодействующих скрытых слоев.

грамматика (grammar)

Грамматика — это набор правил, определяющих компоненты правильно структурированных предложений.

границы предложения (sentence boundaries)

Границы предложений, такие как заглавные буквы и знаки пунктуации, отмечают начало и конец предложений. Большинство инструментов парсинга и маркировки частями речи опираются на наличие границ предложений.

граф (graph)

Сетевой граф — это структура данных, состоящая из узлов, связанных ребрами, которую можно использовать для моделирования сложных взаимоотношений, включая взаимоотношения в тексте.

граф свойств (property graph)

В контексте графов модель графа свойств позволяет встроить в граф больше информации в виде меток и весов, хранящихся в узлах и ребрах графа.

дедупликация (deduplication)

Дедупликация — одна из трех основных задач, имеющих отношение к разрешению сущностей, и заключающаяся в устранении копий (точных и смысловых) повторяющихся данных.

диалоговая система (dialog system)

В контексте чат-бота диалоговая система — это внутренний компонент, интерпретирующий ввод, поддерживающий внутреннее состояние и производящий ответы.

диаметр (diameter)

Диаметр графа `G` — это количество узлов, которые нужно миновать, чтобы пройти кратчайшим путем между двумя наиболее удаленными его узлами.

дивизимный (divisive)

Дивизимная (делящая) кластеризация — это разновидность иерархической кластеризации, согласно которой сначала создается кластер, объединяющий всю выборку, и затем последовательно делится на более мелкие кластеры до достижения уровня отдельных экземпляров.

дискурс (discourse)

Дискурс — формальная письменная или устная коммуникация, обычно более структурированная, чем неформальная письменная или устная коммуникация.

дисперсия (variance)

Дисперсия — один из двух источников ошибок в обучении с учителем. Определяется как среднее квадратов расстояний от каждой точки до среднего. Низкая дисперсия служит признаком недообученности модели, которая часто возвращает один и тот же прогноз, независимо от значений входных признаков. Высокая дисперсия служит признаком переобученности, когда

объект оценки запомнил обучающие данные и плохо обобщает ранее не виденные данные.

документ (document)

В контексте анализа текста документ — это один экземпляр дискурса. Корпусы состоят из множества документов.

иерархическая кластеризация (hierarchical clustering)

Иерархическая кластеризация — это разновидность обучения без учителя, которая производит кластеры в виде древовидной структуры, где на каждом уровне есть переменное число кластеров. Иерархические модели могут быть или агломеративными (agglomerative, когда кластеризация осуществляется снизу вверх), или дивизимными (divisive, сверху вниз).

извлечение признаков (feature extraction)

В контексте конвейеров анализа текста под извлечением признаков понимается процесс преобразования документов в векторные представления, к которым можно применить методы машинного обучения.

канонизация (canonicalization)

Канонизация — одна из трех основных задач, имеющих отношение к разрешению сущностей, заключается в преобразовании данных с несколькими возможными представлениями в одну стандартную форму.

классификация (classification)

Классификация — это разновидность машинного обучения с учителем, целью которой является выявление общих черт в экземплярах, состоящих из независимых признаков, и их отношения к той или иной категории. Классификатор можно научить минимизировать ошибку между прогнозируемыми и фактическими категориями в обучающих данных, а после обучения использовать для присваивания категорий новым экземплярам на основе закономерностей, выявленных на этапе обучения.

кластеризация (clustering)

Обучение без учителя, или кластеризация, — это способ выявления скрытой структуры неклассифицированных данных. Цель алгоритмов кластеризации — выявить скрытые закономерности в неклассифицированных данных путем организации экземпляров в существенно различающиеся группы на основе их признаков.

конвейер (pipeline)

В контексте анализа текста конвейер — это метод объединения последовательности преобразований, таких как (например) нормализация, векторизация и анализ признаков, в единый механизм.

корпус (corpus/corpora)

Корпус — это коллекция родственных документов или высказываний на естественном языке.

кратчайший путь (shortest path)

Для заданного графа `G` с узлами `N` и `V` кратчайшим между `N` и `V` является такой путь, который содержит меньше ребер.

латентное размещение Дирихле (latent Dirichlet allocation, LDA)

Латентное размещение Дирихле — это метод определения темы, в котором темы представлены вероятностями появления каждого слова из заданного набора. Документы, в свою очередь, могут быть представлены как сочетания этих тем.

латентно-семантический анализ (latent semantic analysis, LSA)

Латентно-семантический анализ — это решение на основе векторов для тематического моделирования, которое отыскивает группы документов с одинаковыми словами и производит разреженную матрицу «лексемы/документы».

лексема (token)

Лексемы — атомарные единицы данных в анализе текста. Это строки, закодированные байтами и представляющие семантическую информацию, но не содержащие никакой другой информации (например, значения слова).

лексемизация (tokenization)

Лексемизация — процесс разбиения предложений на изолированные лексемы.

лексикон (lexicon)

В контексте анализа текста лексикон — это набор всех уникальных словарных слов в корпусе. Лексические ресурсы часто включают отображения этого набора в другие наборы, такие как описания смысла слов, синонимы или фонетические представления.

локтевая кривая (elbow curve)

Локтевые кривые — прием визуализации качества нескольких моделей кластеризации методом *k*-средних с разными значениями *k*. Выбор модели основывается на наличии «локтевого сгиба» на кривой. Если кривая выглядит как рука, согнутая в локте, с явно выраженным углом между двумя ветвями кривой, точка перегиба соответствует оптимальному значению *k*.

матрица несоответствий (confusion matrix)

Матрица несоответствий — один из методов оценки точности классификатора. Построение матрицы несоответствий после обучения классификатора позволяет получить представление, как предсказанные классы для отдельных контрольных значений соотносятся с фактическими классами.

машинное обучение (machine learning)

Машинное обучение описывает широкий набор методов извлечения значимых закономерностей из существующих данных и применения этих закономерностей для принятия решений или прогнозирования на основе новых данных.

метод главных компонент (principal component analysis, PCA)

Метод главных компонент — это метод преобразования признаков в новую систему координат, описывающую наибольшую долю дисперсии в данных. Метод главных компонент часто используется для свертки размерности плотных данных.

мешок слов/непрерывный мешок слов (bag-of-words (BOW)/continuous bag-of-words (CBOW))

Мешок слов — это метод кодирования текста, когда каждый документ из корпуса преобразуется в вектор с длиной, равной размеру словаря корпуса. Основная идея метода мешка слов заключается в предположении, что смысл и сходство закодированы в словаре.

многопроцессорная обработка (multiprocessing)

Под многопроцессорной обработкой понимается одновременное использование нескольких процессоров (CPU) и возможность системы распределять решение задач между несколькими процессорами для одновременного их выполнения.

модель языка (language model)

Модель языка пытается на основании неполной фразы вывести недостающие слова, которые вероятнее всего приведут высказывание к законченному виду.

морфология (morphology)

Морфология — это форма сущностей, таких как отдельные слова или лексемы. Морфологический анализ — это процесс получения представлений о том, как конструируются слова и как формы слов влияют на их отношение к той или иной части речи.

недообучение (underfitting)

Под недообученностью модели обычно подразумевается ситуация, когда обученная модель каждый раз возвращает один и тот же прогноз (то есть имеет низкую дисперсию), при этом прогноз существенно отклоняется от истины (то есть имеет высокое смещение). Недообучение является признаком недостаточности объема обучающих данных или сложности обучаемой модели.

нейронная сеть (neural network)

Нейронные сети относятся к семейству моделей, которые определяются входным слоем (векторным представлением входных данных), скрытым слоем, состоящим из нейронов и синапсов, и выходным слоем с предсказанными значениями. Синапсы внутри скрытого слоя отвечают за передачу сигналов между нейронами, которые и используют нелинейную функцию активации для буферизации входящих сигналов. Синапсы применяют весовые коэффициенты к входным значениям, а функция активации определяет, достаточно ли высок уровень взвешенного входного сигнала для активации нейрона и передачи значения в следующий слой сети.

неопределенность (perplexity)

Неопределенность — это оценка предсказуемости текста, основанная на энтропии (уровне непредсказуемости или неожиданности) распределения вероятностей в модели языка.

обучающая и контрольная выборки (training and test splits)

В машинном обучении с учителем данные разделяются на обучающую и контрольную выборки, на которых можно проводить обучение моделей с целью сравнения (перекрестной проверки) и выявления моделей, показывающих наиболее качественные результаты на ранее не виденных ими данных. Деление данных на обучающую и контрольную выборки обычно

используется, чтобы избежать переобучения модели и добиться лучшей обобщающей способности на данных, не участвовавших в обучении.

обучение без учителя (unsupervised learning)

Обучение без учителя, или кластеризация, — это способ выявления скрытой структуры в неклассифицированных данных. Целью алгоритмов кластеризации является определение скрытых закономерностей в неклассифицированных данных с использованием признаков для организации экземпляров в изолированные значимые группы.

обработка естественного языка (natural language processing)

Под обработкой естественного языка понимается комплекс вычислительных приемов для отображения между формальным и естественным языками.

обход (traversal)

В контексте графов под обходом понимается процесс перехода от одного узла к другому по ребрам.

объединение признаков (feature union)

Объединение признаков позволяет применить к данным несколько независимых преобразований и затем объединить результаты в составной вектор.

объект оценки (estimator)

В контексте библиотеки Scikit-Learn объект оценки (`Estimator`) — это любой объект, способный обучаться на оцениваемых данных. Например, объект оценки может быть моделью машинного обучения, векторизатором или преобразователем.

объект чтения корпуса (corpus reader)

Объект чтения корпуса — это программный интерфейс для чтения, поиска, фильтрации и потоковой передачи документов, предоставляющий также методы преобразования данных, такие как кодирование и предварительная обработка, программному коду, которому требуется доступ к данным в корпусе.

онтология (ontology)

Онтология — это структура данных, кодирующая смысл за счет определения свойств и отношений понятий и категорий в конкретном предметном дискурсе.

отчет классификации/тепловая карта (classification report/classification heatmap)

Отчет классификации отображает основные характеристики классификации (точность, полноту, и оценку F1) для каждого класса.

оценка силуэта (silhouette score)

Оценка силуэта — это метод определения плотности и обособленности кластеров, производимых моделью кластеризации на основе центроидов путем усреднения коэффициента силуэта. Вычисляется как разность среднего расстояния внутри кластера и среднего расстояния до ближайшего кластера от этого образца, нормализованная максимальным значением.

оценка F1 (F1 score)

Оценка F1 — это взвешенное гармоническое среднее точности. Лучшей оценкой F1 считается значение 1,0, а худшей — значение 0,0. Обычно оценки F1 ниже оценки качества, потому что вычисляются на основе оценок точности и полноты. Как правило, для сравнения моделей классификации следует использовать среднюю взвешенную оценку F1 вместо глобальной точности.

параллелизм (parallelism)

Под параллелизмом понимаются многопроцессорные вычисления; подразумевается параллелизм на уровне выполнения задач (когда над одними и теми же данными параллельно выполняются разные операции) и на уровне данных (когда одна и та же операция одновременно применяется к нескольким разным фрагментам данных).

парсинг (parsing)

В контексте анализа данных парсинг — это процесс деления высказываний на составные части (например, документов на абзацы, абзацев на предложения, предложений на лексемы) и составления из них синтаксических или семантических структур, которые могут участвовать в вычислениях.

партитивная кластеризация (partitive clustering)

В контексте анализа текста под партитивной кластеризацией подразумеваются методы кластеризации, делящие документы на группы, представляемые центральными векторами (центроидами) или описываемые плотностью документов в кластере. Центроиды представляют агрегированное значение (например, среднее или медиану) всех документов-членов и дают удобный способ описания документов в кластере.

перекрестная проверка/перекрестная проверка по *k*-блокам (cross-validation/ k-fold cross-validation)

Перекрестная проверка или перекрестная проверка по *k*-блокам — это процесс независимого обучения модели с учителем на *k* выборках (обучающих и контрольных) из набора данных, позволяющий сравнить модели и выяснить наперед, какие будут давать более качественные прогнозы на прежде не встречавшихся им данных. Перекрестная проверка помогает найти компромисс между смещением и дисперсией.

переобучение (overfitting)

В контексте обучения с учителем под переобучением модели понимается модель, запомнившая обучающие данные и показывающая абсолютную точность на данных, виденных ранее, но часто ошибающаяся на прежде не встречавшихся данных.

полнота (recall)

Полнота — это способность классификатора правильно классифицировать все экземпляры. Для каждого класса определяется как отношение числа экземпляров, для которых данный класс предсказан правильно, к общему числу документов этого класса. Проще говоря: «Процент охвата экземпляров данного класса».

понимание естественного языка (natural language understanding)

Понимание естественного языка — это одна из тем в обработке естественного языка, обозначающая вычислительные приемы приблизительной интерпретации естественного языка.

порядок (order)

В контексте графа `G` порядком `G` является число узлов в `G`.

преобразователь (transformer)

Преобразователь — особая разновидность объекта оценки, который создает новый набор данных из исходного методом сингулярного разложения (singular value decomposition, SVD), основываясь на правилах, выявленных в процессе обучения.

признак (feature)

В машинном обучении данные представлены пространством числовых признаков, где каждое свойство в векторном представлении является признаком.

прикладной программный интерфейс (Application Programming Interface, API)

Прикладной программный интерфейс формально определяет порядок взаимодействия компонентов. Программный интерфейс данных может давать пользователям возможность систематического извлечения данных из интернета. Scikit-Learn API предоставляет обобщенный доступ к алгоритмам машинного обучения, реализованным в виде иерархий классов.

приложение данных (data product)

Приложения данных — это компьютерные программы, извлекающие ценную информацию из данных и в свою очередь генерирующие новые данные.

прореживание (dropout)

В контексте нейронных сетей слой прореживания предназначен для отдаления момента наступления переобучения за счет случайного обнуления некоторой доли входных единиц в каждом цикле обучения.

прямое кодирование (one-hot encoding)

Прямое кодирование — это метод создания вектора логических значений, в котором конкретный элемент получает значение `True`, если заданная лексема присутствует в документе, и `False`, если нет.

размер (графов) (size (graphs))

Размер графа `G` определяется как число содержащихся в нем ребер.

разрешение сущностей (entity resolution (ER))

Разрешение сущностей — это задача устранения неоднозначностей записей, соответствующих сущностям в наборах данных.

распределение частот (frequency distribution)

Распределение частот отображает относительную частоту результатов (например, лексем, словосочетаний, сущностей) в данной выборке.

распределенное представление (distributed representation)

Распределенное представление — это метод кодирования текста в непрерывном масштабе. То есть получающееся векторное представление документа не просто отражает позиции лексем в их оценки, но образует пространство признаков для определения сходства.

ребро/связь (edge/link)

Ребро `E` между узлами `N` и `V` в графе `G` представляет связь между `N` и `V`.

регрессия (regression)

Регрессия — это разновидность машинного обучения с учителем, целью которой является выявление общих черт в экземплярах, состоящих из независимых признаков, и их принадлежности к тому или иному целевому значению. Регрессор можно научить минимизировать ошибку между прогнозируемыми и фактическими категориями в обучающих данных, а после обучения использовать для присваивания целевых значений новым экземплярам на основе закономерностей, выявленных на этапе обучения.

сбор данных (ingestion)

В контексте исследований данных под сбором данных понимается процесс извлечения данных из источников и сохранение их в корпусе.

связывание записей (record linkage)

Связывание записей — одна из трех основных задач, имеющих отношение к разрешению сущностей, заключающаяся в идентификации записей в разных источниках, ссылающихся на одну и ту же сущность.

сегментация (segmentation)

В контексте анализа текста под сегментацией подразумевается процесс разбиения абзацев на предложения для получения более мелких единиц дискурса.

семантика (semantics)

Семантика — это значение языка (например, значение документа или предложения).

сеть (network)

Сеть — это структура данных с узлами, связанными ребрами, которая может использоваться для моделирования сложных взаимоотношений, включая взаимоотношения в тексте. *См. также «граф (graph)».*

символическая модель языка (symbolic language model)

Символические модели языка интерпретируют текст как дискретную последовательность лексем с вероятностями их появления.

сингулярное разложение (singular value decomposition, SVD)

Сингулярное разложение — это метод матричного разложения, преобразующий исходное пространство признаков в три матрицы, включая диагональную матрицу сингулярных значений, описывающую подпространство. Метод сингулярного разложения широко используется для снижения размерности разреженных данных и применяется в латентно-семантическом анализе (Latent Semantic Analysis, LSA).

синсет (synset)

Синсет для слова `W` — это коллекция когнитивных синонимов, выражающих разные понятия для `W`.

синтаксис (syntax)

Синтаксис описывает правила формирования предложений, определяемые грамматикой.

скраппинг (scraping)

Под скраппингом понимается процесс (автоматизированный, полуавтоматизированный или ручной) сбора информации из интернета и ее копирования в хранилище.

скрытый слой (hidden layer)

В нейронных сетях скрытый слой состоит из нейронов и синапсов и связывает входной слой с выходным. Синапсы передают сигналы между нейронами, функции активации в которых буферизуют входящие сигналы, тем самым создавая эффект обучения модели.

смещение (bias)

Смещение является одним из двух источников ошибок в задачах обучения с учителем и вычисляется как разность между предсказанным и истинным значением. Большое смещенение указывает, что прогнозы, возвращаемые моделью, существенно отклоняются от верных ответов.

смысл слова (word sense)

Под смыслом слова понимается значение конкретного слова с учетом контекста и исходя из предположения, что многие слова имеют несколько коннотаций, интерпретаций и случаев использования.

соседи (neighborhood)

В контексте сетевого графа `G` и заданного узла `N` соседом узла `N` является подграф `F`, содержащий все смежные узлы, связанные ребрами с узлом `N`.

способность к обобщению (generalizable)

Модель, способная к обобщению, имеет лучший баланс смещения и дисперсии, что позволяет ей давать надежные прогнозы на новых, прежде не встречавшихся данных.

степень (degree)

Степень узла `N` графа `G` — это число ребер в `G`, затрагивающих `N`.

стоп-слова (stopwords)

Стоп-слова — это слова, которые вручную исключаются из текста, нередко потому, что они встречаются слишком часто во всех документах в корпусе.

сущность (entity)

Сущность — это нечто уникальное (например, человек, организация, продукт) со множеством атрибутов, описывающих его (например, имя, адрес, форма, название, цена и т. д.). В источниках данных может содержаться несколько ссылок на сущность, таких как два разных адреса электронной почты, принадлежащих одному человеку, два номера телефона одной компании или два веб-сайта, где упоминается один и тот же продукт.

тематическое моделирование (topic modeling)

Тематическое моделирование — это метод машинного обучения без учителя для определения тем из коллекций документов. *См. также «кластеризация (clustering)».*

точность (precision)

Точность — это способность классификатора не отметить положительным экземпляр, который фактически является отрицательным. Для каждого класса определяется как отношение числа экземпляров, для которых данный класс предсказан правильно, к общему числу документов, для которых предсказан этот же класс (правильно или по ошибке). Проще говоря: «Процент правильно классифицированных экземпляров».

транзитивность (transitivity)

В сетевых графах транзитивность служит мерой вероятности наличия общих соседей у двух узлов.

тройка выбора модели (model selection triple)

Тройка выбора модели описывает обобщенный процесс машинного обучения, включающий многократные итерации через этапы конструирования

признаков, выбора модели и настройки гиперпараметров для достижения наиболее точной модели, способной к обобщению.

узел/вершина (node/vertex)

В контексте графа узел является фундаментальной единицей данных. Узлы связаны ребрами и все вместе образуют сеть.

управление (steering)

Управление — это процесс управления машинным обучением, например, путем визуального представления разных отчетов классификации и построения тепловых карт для определения, какая из обученных моделей дает более качественные результаты, или исследованием компромиссов между смещением и дисперсией по разным значениям определенного гиперпараметра.

хранилище WORM (WORM storage)

Хранилище, доступное для записи одному и для чтения многим (write-once read-many, WORM), подразумевает хранение версии исходных данных, не изменяющейся на этапах извлечения, преобразования или моделирования.

центральность (centrality)

В сетевом графе центральность служит мерой относительной важности узла. Важные узлы прямо или косвенно связаны с большинством других узлов и поэтому имеют высокую центральность.

центральность по близости (closeness centrality)

Центральность по близости определяется как средняя длина пути от узла `N` в графе `G` ко всем другим узлам, нормализованная размером графа; она описывает, как быстро распространится по сети `G` информация, исходящая из узла `N`.

центральность по посредничеству (betweenness centrality)

Для заданного узла `N` в графе `G` центральность по посредничеству отражает важность этого узла для связанности графа `G`. Центральность по посредничеству вычисляется как отношение кратчайших путей в `G`, пролегающих через `N`, к общему числу всех кратчайших путей в `G`.

центральность по степени (degree centrality)

Центральность по степени определяется как количество соседей данного узла, нормализованное общим количеством узлов в графе `G`.

центральность по собственному вектору (eigenvector centrality)

Центральность по собственному вектору определяет важность узла `N` в графе `G` по степени узлов, с которыми связан `N`. Даже если у `N` небольшое количество соседей, но они имеют высокие степени, `N` может опережать некоторых из них величиной центральности по собственному вектору. Центральность по собственному вектору лежит в основе нескольких других мер центральности, включая центральность Каца (Katz centrality) и известный алгоритм авторитетности PageRank.

частота термина — обратная частота документа (term frequency–inverse document frequency, TF–IDF)

Частота термина — обратная частота документа — это метод кодирования, нормализующий частоту появления лексем в документе с учетом содержимого в остальном корпусе. Оценка TF–IDF измеряет релевантность лексемы в документе масштабированной частотой появления лексемы в документе, нормализованной обратной масштабированной частотой появления лексемы во всем корпусе.

часть речи (part-of-speech)

Части речи — это классы, назначенные анализируемому тексту, определяющие, как действуют лексемы в контексте предложения. К частям речи относятся, например, существительные, глаголы, прилагательные и наречия.

чат-бот (chatbot)

Чат-бот — это программа, участвующая в разговоре с передачей права голоса, целью которой является интерпретация входного текста или речи и вывод соответствующих полезных ответов.

экземпляр (instance)

В машинном обучении экземпляры — это точки, которыми оперирует алгоритм. В контексте анализа текста экземпляр — это документ целиком или законченное высказывание.

энтропия (entropy)

Энтропия определяет непредсказуемость или неожиданность распределения вероятностей в модели языка.

baleen

Baleen — это механизм с открытым кодом, автоматизирующий сбор текстовых данных и создание корпусов для исследований в области анализа естественного языка.

doc2vec

Doc2vec (расширение для word2vec) — это алгоритм обучения без учителя, извлекающий представления признаков фиксированной длины из документов переменной длины.

***n*-грамма (*n*-gram)**

n-грамма — упорядоченная последовательность символов или слов с длиной `N`.

rss

RSS — это разновидность веб-каналов, в которых публикуются новости в стандартном формате, удобном для компьютеров.

***t*-распределенное стохастическое вложение соседей (t-distributed stochastic neighbor embedding, t-SNE)**

t-распределенное стохастическое вложение соседей — это метод нелинейного уменьшения размерности. *t*-SNE можно использовать для объединения похожих документов путем разложения векторных представлений документов с большим числом измерений в два измерения с использованием распределений вероятностей из оригинальной и разложенной размерностей.

word2vec

word2vec — алгоритм, реализующий такую модель векторного представления слов, что слова оказываются в пространстве признаков рядом со схожими словами.

Об авторах

Бенджамин Бенгфорт (Benjamin Bengfort) — специалист в области data science, живущий в Вашингтоне, внутри кольцевой автострады, но полностью игнорирующий политику (обычное дело для округа Колумбия) и предпочитающий заниматься технологиями. В настоящее время работает над докторской диссертацией в Университете штата Мериленд, где изучает машинное обучение и распределенные вычисления. В его лаборатории есть роботы (хотя это не является его любимой областью), и к его большому огорчению, помощники постоянно вооружают этих роботов ножами и инструментами, вероятно, с целью победить в кулинарном конкурсе. Наблюдая, как робот пытается нарезать помидор, Бенджамин предпочитает сам хозяйничать на кухне, где готовит французские и гавайские блюда, а также шашлыки и барбекю всех видов. Профессиональный программист по образованию, исследователь данных по призванию, Бенджамин часто пишет статьи, освещающие широкий круг вопросов — от обработки естественного языка до исследования данных на Python и применения Hadoop и Spark в аналитике.

Д-р Ребекка Билбро (Dr. Rebecca Bilbro) — специалист в области data science, программист на Python, учитель, лектор и автор статей; живет в Вашингтоне (округ Колумбия). Специализируется на визуальной оценке результатов машинного обучения: от анализа признаков до выбора моделей и настройки гиперпараметров. Проводит исследования в области обработки естественного языка, построения семантических сетей, разрешения сущностей и обработки информации с большим количеством измерений. Как активный участник сообщества пользователей и разработчиков открытого программного обеспечения, Ребекка с удовольствием сотрудничает с другими разработчиками над такими проектами, как Yellowbrick (пакет на языке Python, целью которого является прогнозное моделирование на манер черного ящика). В свободное время часто катается на велосипедах с семьей или практикуется в игре на укулеле. Получила докторскую степень в Университете штата Иллинойс, в Урбана-Шампейн, где занималась исследованием практических приемов коммуникации и визуализации в технике.

Тони Охеда (Tony Ojeda) — специалист в области data science, писатель и предприниматель с опытом оптимизации бизнес-процессов и более чем десятилетним опытом создания и внедрения инновационных приложений данных и решений. Основал District Data Labs, консалтинговую фирму, специализирующуюся на корпоративных тренингах в области исследования данных и совместных исследованиях, где люди разных специальностей работают над интересными проектами, расширяют свои возможности и помогают друг другу стать более успешными исследователями данных. Также является сооснователем Data Community DC, профессиональной организации, поддерживающей и продвигающей исследователей данных и их работу. В свободное время любит плавать, бегать, заниматься боевыми искусствами, открывать для себя новые рестораны и смотреть увлекательные телепередачи. Имеет степень магистра финансов, полученную в Международном университете штата Флорида, и степень MBA. Обучался стратегии предпринимательства в Университете Де Поля в Чикаго.

Выходные данные

На обложке книги «Прикладной анализ текстовых данных на Python» изображена американская лисица (Vulpes macrotis) — небольшое млекопитающее, обитающее на юго-западе США и севере Мексики. Это самый маленький вид лисиц в Северной Америке, весом от 1,5 до 3 килограммов и ростом от 45 до 54 сантиметров. Интересно отметить, что у американских лисиц непропорционально большие уши, которые обеспечивают им превосходный слух и помогают регулировать температуру тела. Они имеют серый окрас, часто с красноватым или оранжевым оттенком.

Американские лисицы обитают в безводных, пустынных районах и ведут ночной образ жизни. Охотятся на мелких млекопитающих, таких как мыши, кролики и полевки, а также насекомых, ящериц и птиц. Американские лисицы моногамны и спариваются ежегодно, с декабря по февраль. Детеныши рождаются в марте или апреле, и в помете обычно насчитывается от 1 до 7 щенков. В воспитании детенышей участвуют оба родителя. В возрасте 5–6 месяцев детеныши покидают семейное логово.

Американские лисицы пока не занесены в Красную книгу, но в настоящее время нет достоверных данных о численности их популяции. Есть признаки, свидетельствующие об их постепенном исчезновении в районах, где всё новые земли занимаются под сельскохозяйственные угодья. Они также сталкиваются с конкуренцией за добычу с другими хищниками, такими как койоты, рыси и беркуты.

Многие животные, изображенные на обложках книг издательства O'Reilly, находятся под угрозой вымирания; все они очень важны для биосферы. Чтобы узнать, чем вы можете помочь, посетите сайт animals.oreilly.com.

Изображение для обложки взято из энциклопедии Лидеккера (Lydekker) *Royal Natural History*.

Бенджамин Бенгфорт, Ребекка Билбро, Тони Охеда

Прикладной анализ текстовых данных на Python. Машинное обучение и создание приложений обработки естественного языка

Перевел с английского А. Киселев

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Научный редактор	*Д. Дрошнев*
Художественный редактор	*В. Мостипан*
Корректор	*И. Тимофеева*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 01.2019. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 24.12.18. Формат 70×100/16. Бумага офсетная. Усл. п. л. 29,670. Тираж 1200. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,
pitvolga@samara-ttk.ru

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;
e-mail: og@minsk.piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanovaa@piter.com
Подробная информация здесь: http://www.piter.com/page/avtoru

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок: *тел./факс: (812) 703-73-73; e-mail: sales@piter.com*

Заказ книг для вузов и библиотек:
тел./факс: (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

---

Заказ книг по почте: *на сайте www.piter.com; тел.: (812) 703-73-74, доб. 6216; e-mail: books@piter.com*

Вопросы по продаже электронных книг: *тел.: (812) 703-73-74, доб. 6217; e-mail: kuznetsov@piter.com*

КНИГА-ПОЧТОЙ

ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: **www.piter.com**
- по электронной почте: **books@piter.com**
- по телефону: **(812) 703-73-74**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

- Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.
- С помощью банковской карты. Во время заказа вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
- Электронными деньгами. Мы принимаем к оплате Яндекс.Деньги, Webmoney и Qiwi-кошелек.
- В любом банке, распечатав квитанцию, которая формируется автоматически после совершения вами заказа.

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ДОСТАВКИ:

- Посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку ваших покупок максимально быстро. Дату отправления вашей покупки и дату доставки вам сообщат по e-mail.
- Вы можете оформить курьерскую доставку своего заказа (более подробную информацию можно получить на нашем сайте www.piter.com).
- Можно оформить доставку заказа через почтоматы,(адреса почтоматов можно узнать на нашем сайте www.piter.com).

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

БЕСПЛАТНАЯ ДОСТАВКА:

- курьером по Москве и Санкт-Петербургу при заказе на сумму **от 2000 руб.**
- почтой России при предварительной оплате заказа на сумму **от 2000 руб.**

ИЗДАТЕЛЬСКИЙ ДОМ
ПИТЕР®
WWW.PITER.COM

ВАША УНИКАЛЬНАЯ КНИГА

Хотите издать свою книгу? Она станет идеальным подарком для партнеров и друзей, отличным инструментом для продвижения вашего бренда, презентом для памятных событий! Мы сможем осуществить ваши любые, даже самые смелые и сложные, идеи и проекты.

МЫ ПРЕДЛАГАЕМ:

- издать вашу книгу
- издание книги для использования в маркетинговых активностях
- книги как корпоративные подарки
- рекламу в книгах
- издание корпоративной библиотеки

Почему надо выбрать именно нас:

Издательству «Питер» более 20 лет. Наш опыт – гарантия высокого качества.

Мы предлагаем:

- услуги по обработке и доработке вашего текста
- современный дизайн от профессионалов
- высокий уровень полиграфического исполнения
- продажу вашей книги во всех книжных магазинах страны

Обеспечим продвижение вашей книги:

- рекламой в профильных СМИ и местах продаж
- рецензиями в ведущих книжных изданиях
- интернет-поддержкой рекламной кампании

Мы имеем собственную сеть дистрибуции по всей России, а также на Украине и в Беларуси. Сотрудничаем с крупнейшими книжными магазинами. Издательство «Питер» является постоянным участником многих конференций и семинаров, которые предоставляют широкую возможность реализации книг.

Мы обязательно проследим, чтобы ваша книга постоянно имелась в наличии в магазинах и была выложена на самых видных местах.

Обеспечим индивидуальный подход к каждому клиенту, эксклюзивный дизайн, любой тираж.

Кроме того, предлагаем вам выпустить электронную книгу. Мы разместим ее в крупнейших интернет-магазинах. Книга будет сверстана в формате ePub или PDF – самых популярных и надежных форматах на сегодняшний день.

Свяжитесь с нами прямо сейчас:

Санкт-Петербург – *Анна Титова, (812) 703-73-73, titova@piter.com*
Москва – *Сергей Клебанов, (495) 234-38-15, klebanov@piter.com*