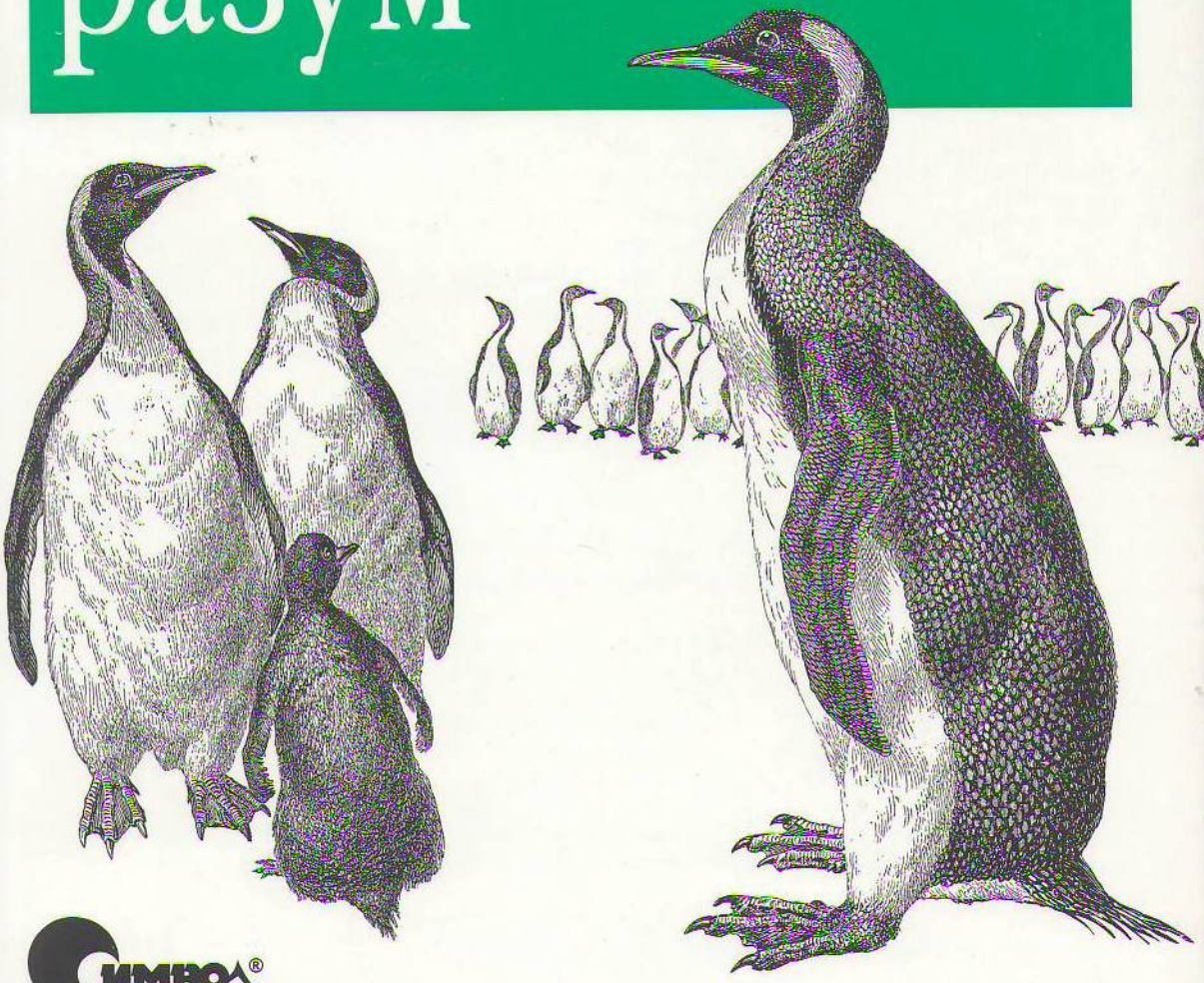
Создание интеллектуальных приложений для Web 2.0

Программируем

Коллективный разум



СИМВОЛ®
O'REILLY®

Тоби Сегаран

Programming Collective Intelligence

Toby Segaran

O'REILLY®

Программируем коллективный разум

Тоби Сегаран

СИМВОЛ®

Тоби Сегаран

Программируем коллективный разум

Перевод А. Слинкина

Главный редактор	*А. Галунов*
Выпускающий редактор	*Л. Пискунова*
Редактор	*Е. Тульсанова*
Научный редактор	*С. Миронов*
Корректор	*Е. Бекназарова*
Верстка	*Д. Белова*

Сегаран. Т.
Программируем коллективный разум. – Пер. с англ. – СПб: Символ-Плюс, 2008. – 368 с., ил.

ISBN13: 978-5-93286-119-6
ISBN10: 5-93286-119-3

Средства эффективной обработки информации в Интернете еще никогда не были настолько важны и востребованы, как сегодня. Эта книга – первое практическое руководство по программированию интеллектуальных приложений для Web 2.0. Здесь вы найдете все необходимое, чтобы научиться создавать самообучаемые программы, которые способны собирать и анализировать огромные массивы данных, имеющиеся в Сети. Вы научитесь пользоваться алгоритмами машинного обучения, адаптируя их под свои собственные нужды. Чтобы овладеть представленным материалом, от вас не потребуется никаких специальных знаний об анализе данных, машинном обучении или математической статистике. Однако предполагается, что вы имеете достаточный опыт программирования и знакомы с основными концепциями. В каждой главе вы найдете практические примеры и задания, которые помогут вам быстро и легко освоить различные аспекты работы алгоритмов.

ISBN13: 978-5-93286-119-6
ISBN10: 5-93286-119-3
ISBN 0-596-52932-5 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 3245353, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.

Налоговая льгота – общероссийский классификатор продукции
ОК 005 93, том 2; 953000 - книги и брошюры.

Подписано в печать 30.06.2008. Формат 70x100 $^{1}/_{16}$. Печать офсетная.
Объем 23 печ. л. Тираж 2000 экз. Заказ N
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Предисловие 9
Введение 10
Благодарности 18

1. Введение в коллективный разум 19

Что такое коллективный разум 20
Что такое машинное обучение 22
Ограничения машинного обучения 23
Примеры из реальной жизни 23
Другие применения обучающих алгоритмов 24

2. Выработка рекомендаций 26

Коллаборативная фильтрация 27
Сбор информации о предпочтениях 27
Отыскание похожих пользователей 29
Рекомендование предметов 35
Подбор предметов 37
Построение рекомендателя ссылок с помощью API сайта del.icio.us ... 39
Фильтрация по схожести образцов 43
Использование набора данных MovieLens 46
Сравнение методов фильтрации по схожести пользователей и по схожести образцов 48
Упражнения 49

3. Обнаружение групп 50

Обучение с учителем и без него 51
Векторы слов 51
Иерархическая кластеризация 54
Рисование дендрограммы 60
Кластеризация столбцов 63

Кластеризация методом K-средних ... 64
Кластеры предпочтений ... 67
Просмотр данных на двумерной плоскости ... 71
Что еще можно подвергнуть кластеризации ... 75
Упражнения ... 76

4. Поиск и ранжирование ... 77

Что такое поисковая машина ... 77
Простой паук ... 79
Построение индекса ... 82
Запросы ... 86
Ранжирование по содержимому ... 88
Использование внешних ссылок на сайт ... 93
Обучение на основе действий пользователя ... 99
Упражнения ... 109

5. Оптимизация ... 111

Групповые путешествия ... 112
Представление решений ... 113
Целевая функция ... 114
Случайный поиск ... 117
Алгоритм спуска с горы ... 118
Алгоритм имитации отжига ... 120
Генетические алгоритмы ... 123
Поиск реальных авиарейсов ... 127
Оптимизация с учетом предпочтений ... 132
Визуализация сети ... 136
Другие возможности ... 142
Упражнения ... 142

6. Фильтрация документов ... 144

Фильтрация спама ... 145
Документы и слова ... 145
Обучение классификатора ... 146
Вычисление вероятностей ... 149
Наивная классификация ... 151
Метод Фишера ... 155
Сохранение обученных классификаторов ... 159
Фильтрация блогов ... 161
Усовершенствование алгоритма обнаружения признаков ... 164
Использование службы Akismet ... 166

Альтернативные методы ... 168
Упражнения ... 168

7. Моделирование с помощью деревьев решений ... 170

Прогнозирование количества регистраций ... 171
Введение в теорию деревьев решений ... 173
Обучение дерева ... 174
Выбор наилучшего разбиения ... 176
Рекурсивное построение дерева ... 178
Отображение дерева ... 180
Классификация новых наблюдений ... 183
Отсечение ветвей дерева ... 184
Восполнение отсутствующих данных ... 186
Числовые результаты ... 188
Моделирование цен на недвижимость ... 188
Моделирование степени привлекательности ... 191
В каких случаях применять деревья решений ... 194
Упражнения ... 196

8. Построение ценовых моделей ... 197

Построение демонстрационного набора данных ... 198
Алгоритм k-ближайших соседей ... 199
Взвешенные соседи ... 203
Перекрестный контроль ... 207
Гетерогенные переменные ... 209
Оптимизация масштаба ... 212
Неравномерные распределения ... 214
Использование реальных данных – API сайта eBay ... 220
В каких случаях применять метод k-ближайших соседей ... 227
Упражнения ... 227

9. Более сложные способы классификации: ядерные методы и машины опорных векторов ... 229

Набор данных для подбора пар ... 230
Затруднения при анализе данных ... 231
Простая линейная классификация ... 234
Категориальные свойства ... 238
Масштабирование данных ... 242
Идея ядерных методов ... 243
Метод опорных векторов ... 247
Библиотека LIBSVM ... 249

Подбор пар на сайте Facebook ... 252
Упражнения ... 258

10. Выделение независимых признаков ... 259

Массив новостей ... 260
Прошлые подходы ... 264
Неотрицательная матричная факторизация ... 267
Вывод результатов ... 273
Использование данных о фондовом рынке ... 277
Упражнения ... 282

11. Эволюционирующий разум ... 284

Что такое генетическое программирование ... 285
Программы как деревья ... 287
Создание начальной популяции ... 292
Проверка решения ... 293
Мутация программ ... 295
Скрещивание ... 298
Построение окружающей среды ... 300
Простая игра ... 303
Направления развития ... 308
Упражнения ... 311

12. Сводка алгоритмов ... 312

Байесовский классификатор ... 312
Классификатор на базе деревьев решений ... 316
Нейронные сети ... 320
Метод опорных векторов ... 324
k-ближайшие соседи ... 329
Кластеризация ... 333
Многомерное шкалирование ... 336
Неотрицательная матричная факторизация ... 338
Оптимизация ... 341

Приложения ... 344

A. Дополнительные библиотеки ... 344

B. Математические формулы ... 351

Алфавитный указатель ... 358

Предисловие

Когда журнал Time выбрал в качестве человека 2006 года «вас»[1], он утвердил идею о том, что сущность Web 2.0 – это «контент, генерируемый пользователями», и что такие сайты, как Википедия, YouTube и MySpace, – это столпы революции, совершившейся с приходом Web 2.0. Но истина гораздо сложнее. Тот контент, который пользователи вводят в Web 2.0 явно, – лишь самая верхушка айсберга. А 80% существенной информации скрыто в темных водах неявно образующихся данных.

Во многих отношениях определяющим моментом революции Web 2.0 стало изобретение компанией Google «ранга страниц» (PageRank). Это стало результатом осознания того факта, что каждая ссылка во Всемирной паутине наполнена скрытым смыслом: ссылка – это голос в пользу важности сайта. Если принять во внимание сами голоса и относительную важность голосующих сайтов, то результаты поиска оказываются лучше, чем при анализе одного лишь содержимого страниц. Именно этот прорыв вывел Google на путь, следуя по которому она стала одной из наиболее авторитетных технологических компаний нового столетия. Ранг страницы – один из сотен неявно учитываемых факторов, которые Google оценивает, решая, как представить результаты поиска.

Никто не назовет компанию Google «генератором пользовательского контента», и тем не менее она, безусловно, находится в самом сердце Web 2.0. Вот почему я предпочитаю считать краеугольным камнем этой революции «обуздание коллективного разума». Ссылка – это контент, генерируемый пользователями, а PageRank – способ извлечения смысла из этого контента. Это же относится и к алгоритму вычисления «интересности» на сайте Flickr, и к функции «те, кто купил этот товар, купили также» на Amazon, и к алгоритмам нахождения «похожих исполнителей» на Last.fm, и к системе репутационного рейтингования на eBay, и к рекламной системе Google AdSense.

Я определяю Web 2.0 следующим образом: «методика проектирования систем, которые путем учета сетевых взаимодействий становятся тем лучше, чем больше людей ими пользуются». Привлечение пользователей – первый шаг. Второй – обучение на основе действий пользователей и адаптация сайта в зависимости от того, какие действия пользователи совершают и на что они обращают внимание.

В книге «Программируем коллективный разум» Тоби Сегаран рассматривает алгоритмы и методы извлечения смысла из данных, в том числе и пользовательских. Это инструментарий программиста, работающего в контексте Web 2.0. Теперь уже недостаточно знать, как создать сайт с хранением данных в базе. Если вы хотите добиться успеха, то должны знать, как из этих данных добывать информацию – явно или путем анализа поведения пользователей на вашем сайте.

С 2004 года, когда мы придумали термин Web 2.0, на эту тему уже написано очень много, но книга Тоби – первое практическое руководство по программированию приложений для Web 2.0.

Тим О'Рейли

---

[1] На обложке журнала было помещено слово «You» (Вы). – *Прим. перев.*

Введение

Количество людей, осознанно или случайно пополняющих Интернет данными, постоянно растет. Они уже создали колоссальный массив данных, анализируя которые можно многое узнать о пользователях – как они работают, что предпочитают, какие товары их интересуют, – да и вообще о человеческом поведении. Эта книга представляет собой введение в активно развивающуюся науку о коллективном разуме. В ней рассказывается о том, как получить интересные наборы данных с многих сайтов, о которых вы, наверное, слышали, о том, как собирать данные от пользователей ваших собственных приложений, и о разнообразных способах анализа этих данных и извлечения из них информации. Цель этой книги – помочь вам перейти от простых приложений, хранящих данные в базе, к написанию более интеллектуальных программ, способных эффективно пользоваться той информацией, которую вы и другие люди накапливают каждодневно.

Что необходимо знать

Все примеры в этой книге написаны на языке Python. Знакомство с ним вам не помешало бы, но я объясняю алгоритмы, чтобы и программистам, пишущим на других языках, все было понятно. У тех, кто владеет такими языками высокого уровня, как Ruby или Perl, код на Python не вызовет затруднений. Эта книга – не учебник по программированию, поэтому предполагается, что у вас уже имеется достаточный опыт кодирования и с основополагающими концепциями вы знакомы. Если вы хорошо понимаете, что такое рекурсия, и имеете представление об основах функционального программирования, вам будет проще.

От вас не требуется никаких специальных знаний об анализе данных, машинном обучении или математической статистике. Я старался излагать математические идеи как можно проще, но наличие минимальных познаний в области тригонометрии и статистики поможет вам лучше понять алгоритмы.

Стиль написания примеров

Примеры в каждом разделе написаны в стиле учебного руководства. Это должно побудить вас строить приложения поэтапно и разбираться в работе алгоритмов. В большинстве случаев после написания новой функции или метода вам будет предложено интерактивно поработать с ней, чтобы понять, как она устроена. Алгоритмы, как правило, выбираются простейшие, допускающие обобщение и расширение во многих направлениях. Проработав и протестировав примеры, вы сами придумаете, как улучшить их для своих приложений.

Почему именно Python

Хотя алгоритмы описываются словами с объяснением применяемых формул, гораздо полезнее (и, пожалуй, проще для восприятия) иметь представление алгоритмов и примеров в виде программного кода. Все примеры в этой книге написаны на Python – великолепном языке высокого уровня. Я выбрал Python по следующим причинам:

Краткость

Код на динамически типизированном языке, каковым является Python, как правило, оказывается короче, чем на других популярных языках. Это значит, что вам придется нажимать меньше клавиш при вводе примеров. Но одновременно это означает, что алгоритм проще уложить в мозгу и понять, что же он делает.

Легкость чтения

Python иногда называют «исполняемым псевдокодом». Это, конечно, преувеличение, но смысл его в том, что опытный программист может прочитать написанный на Python код и понять, что он должен делать. Некоторые не слишком очевидные конструкции объясняются в разделе «Замечания о языке Python» ниже.

Простота расширения

В стандартный дистрибутив Python уже входит много библиотек, в том числе для вычисления математических функций, разбора XML-документов и загрузки веб-страниц. Дополнительные библиотеки, которыми мы пользуемся в этой книге, например анализатор RSS-документов (RSS –Really Simple Syndication, язык описания синдицированного контента) и интерфейс к базе данных SQLite, бесплатны; их можно без труда скачать, установить и использовать.

Интерактивность

Прорабатывая пример, полезно тестировать функции, которые вы пишете, не составляя отдельной тестовой программы. Программы на Python можно запускать прямо из командной строки, а кроме того, у него есть интерактивный режим, в котором разрешается вызывать функции, создавать объекты и тестировать пакеты.

Мультипарадигменность

Python поддерживает объектно-ориентированный, процедурный и функциональный стили программирования. Алгоритмы машинного обучения весьма разнообразны: для реализации одного удобнее одна парадигма, а для реализации другого – иная. Иногда полезно передавать функции как параметры, а иногда – сохранять состояние в объекте. Python позволяет и то и другое.

Многоплатформенность и бесплатность

Существует одна эталонная реализация языка Python для всех основных платформ, и она абсолютно бесплатна. Приведенный в этой книге код будет работать в системах Windows, Linux и Macintosh.

Замечания о языке Python

Для начинающих, которые желают научиться программировать на языке Python, я рекомендую книгу Марка Лутца и Дэвида Эшера (Mark Lutz, David Ascher) «Learning Python», где имеется отличный обзор языка. Программистам, владеющим другими языками, будет несложно разобраться в коде на Python, хотя следует иметь в виду, что я иногда пользуюсь довольно специфическими конструкциями, если они позволяют выразить алгоритм или фундаментальную концепцию более естественно. Вот краткий обзор языка для тех, кто с ним не знаком.

Конструкторы списков и словарей

В языке Python имеется широкий набор примитивных типов и два типа, которые используются в книге особенно часто: *список* и *словарь*. Список – это упорядоченное множество значений любого типа, для его конструирования применяются квадратные скобки:

```
number_list=[1,2,3,4]
string_list=['a', 'b', 'c', 'd']
mixed_list=['a', 3, 'c', 8]
```

Словарь – это неупорядоченное множество пар ключ/значение, аналогичное тому, что в других языках называется хешем или отображением. Он конструируется с помощью фигурных скобок:

```
ages={'John':24,'Sarah':28,'Mike':31}
```

Доступ к элементам списков и словарей осуществляется с помощью квадратных скобок, следующих за именем списка (словаря):

```
string_list[2]  # возвращается 'b'
ages['Sarah']   # возвращается 28
```

Значимые пробелы

В отличие от большинства языков, в Python для определения блока кода используются отступы. Рассмотрим следующий фрагмент:

```
if x==1:
  print 'x is 1'
```

```
    print 'Все еще внутри блока'
print 'Вне блока'
```

Интерпретатор понимает, что первые два предложения `print` выполняются, когда `x` равно 1, поскольку они написаны с отступом. Количество пробелов в отступе произвольно, но будьте последовательны. В этой книге отступ составляют два пробела. Когда будете вводить код примеров, обращайте внимание на отступы.

Трансформации списков

Трансформация списка (list comprehension) – это удобный способ преобразования одного списка в другой путем фильтрации и применения к нему функций. Синтаксически это выглядит следующим образом:

```
[выражение for переменная in список]
```

или

```
[выражение for переменная in список if условие]
```

Например, следующий код:

```
l1=[1,2,3,4,5,6,7,8,9]
print [v*10 for v in l1 if v1>4]
```

напечатает такой список:

```
[50,60,70,80,90]
```

Трансформации списков используются в этой книге часто, поскольку это предельно лаконичный способ применить функцию к целому списку или удалить ненужные элементы. Еще один случай их использования – конструктор словаря:

```
l1=[1,2,3,4,5,6,7,8,9]
timesten=dict([(v,v*10) for v in l1])
```

Этот код создает словарь, в котором элементы исходного списка становятся ключами, а соответствующим ключу значением будет значение элемента, умноженное на 10:

```
{1:10,2:20,3:30,4:40,5:50,6:60,7:70,8:80,9:90}
```

Открытые API

Для работы алгоритмов синтеза коллективного разума необходимы данные от многих пользователей. Помимо алгоритмов машинного обучения, в этой книге обсуждается ряд открытых API (Application Programming Interface – программный интерфейс приложения), доступных через Сеть. С его помощью компании предоставляют свободный доступ к данным со своих сайтов посредством специфицированного протокола. Пользуясь опубликованным API, вы можете писать программы, которые будут загружать и обрабатывать данные. Иногда для доступа к таким API имеются написанные на Python библиотеки; в противном случае не слишком сложно написать собственный интерфейс доступа к данным, пользуясь уже имеющимися в языке средствами загрузки данных и разбора XML.

Вот примеры нескольких встречающихся в этой книге сайтов, для которых имеется открытый API:

del.icio.us

Социальное приложение для хранения закладок. Открытый API позволяет загрузить ссылки по тегу или принадлежащие указанному пользователю.

Kayak

Сайт для путешественников. Его API позволяет искать авиарейсы и гостиницы из своей программы.

eBay

Онлайновый аукцион. API позволяет запрашивать информацию о товарах, которые в настоящий момент выставлены на продажу.

Hot or Not

Сайт знакомств и рейтингования. API позволяет искать людей и получать их рейтинги и демографическую информацию.

Akismet

API для коллаборативной фильтрации спама.

Потенциально можно написать огромное число приложений, которые обрабатывают данные из одного источника, объединяют данные из разных источников и даже комбинируют полученную из внешних источников информацию с данными о ваших пользователях. Умение сводить воедино данные, созданные людьми на различных сайтах и разными способами, – это основной элемент создания коллективного разума. Начать поиск других сайтов, предоставляющих открытый API, можно с сайта ProgrammableWeb (*http://www.programmableweb.com*).

Обзор глав

Каждый алгоритм, рассматриваемый в этой книге, посвящен конкретной задаче, которая, как я надеюсь, будет понятна всем читателям. Я старался избегать задач, требующих обширных знаний в предметной области, и отбирал такие, которые, несмотря на сложность, ассоциируются с чем-то знакомым большинству людей.

Глава 1 «Введение в коллективный разум»

Разъясняются идеи, лежащие в основе машинного обучения, его применение в различных областях и способы, позволяющие с его помощью делать выводы из данных, полученных от многих людей.

Глава 2 «Выработка рекомендаций»

Дается введение в методику *коллаборативной фильтрации*, которая применяется во многих онлайновых магазинах для рекомендования товаров или мультимедийной продукции. Включен также раздел о рекомендовании ссылок на социальном сайте хранения закладок и о построении системы рекомендования кинофильмов на основе анализа набора данных MovieLens.

Глава 3 «Обнаружение групп»

Развивает идеи, изложенные в главе 2, и знакомит с двумя методами *кластеризации*, позволяющими автоматически обнаруживать группы сходных элементов в большом наборе данных. Демонстрируется применение кластеризации для отыскания групп во множестве популярных блогов и во множестве пожеланий, высказанных посетителями сайта социальной сети.

Глава 4 «Поиск и ранжирование»

Описываются различные компоненты поисковой машины, в том числе паук, индексатор, механизм обработки запросов. Рассмотрен алгоритм ранжирования страниц на основе ведущих на них ссылок, *PageRank*, и показано, как создать *нейронную сеть*, которая обучается тому, какие ключевые слова ассоциированы с различными результатами.

Глава 5 «Оптимизация»

Содержит введение в алгоритмы *оптимизации*, предназначенные для отбора наилучшего решения задачи из миллионов возможных. Широта области применения подобных алгоритмов демонстрируется на примерах поиска оптимальных авиарейсов для группы людей, направляющихся в одно и то же место, оптимального способа распределения студентов по комнатам в общежитии и вычерчивания сети с минимальным числом пересекающихся линий.

Глава 6 «Фильтрация документов»

Демонстрируется методика *байесовской фильтрации*, которая используется во многих бесплатных коммерческих системах фильтрации спама для автоматической классификации документов по типам слов и другим обнаруживаемым свойствам. Описанный подход применим ко множеству результатов поиска по RSS-каналам с целью автоматической классификации найденных записей.

Глава 7 «Моделирование с помощью деревьев решений»

Содержит введение в теорию *деревьев решений*, которая позволяет не только делать прогнозы, но и моделировать способ принятия решений. Первое дерево решений строится на основе гипотетических данных, взятых из протоколов сервера, и используется для того, чтобы предсказать, оформит ли пользователь премиальную подписку. В остальных примерах взяты данные с реальных сайтов для моделирования цен на недвижимость и оценки степени привлекательности различных людей.

Глава 8 «Построение ценовых моделей»

Описывается подход к решению задачи предсказания числовых значений, а не классификации. Для этого применяется метод *k-ближайших соседей* и алгоритмы оптимизации из главы 5. Эти методы в сочетании с eBay API используются для построения системы прогнозирования окончательной цены на торгах исходя из некоторого набора свойств.

Глава 9 «Более сложные способы классификации: ядерные методы и машины опорных векторов»

Показано, как можно использовать метод опорных векторов для подбора пар на сайтах знакомств или в системах подбора профессиональных кадров. Машины опорных векторов – довольно сложная техника, она сравнивается с альтернативными методиками.

Глава 10 «Выделение независимых признаков»

Содержит введение в сравнительно новую методику – неотрицательную матричную факторизацию, которая применяется для поиска независимых признаков в наборе данных. Часто элементы набора данных представляют собой конгломерат различных заранее неизвестных признаков. Данная методика позволяет распознать эти признаки. Описанная техника демонстрируется на примере набора новостей, когда одна или несколько тем новости выводятся из ее текста.

Глава 11 «Эволюционирующий разум»

Содержит введение в *генетическое программирование* – весьма продвинутую теорию, которая выходит за рамки оптимизации и позволяет фактически строить алгоритмы решения задачи, основанные на идее эволюции. В качестве примера рассматривается простая игра, в которой компьютер изначально играет плохо, но постепенно модифицирует свой код, и чем больше игр сыграно, тем совершеннее его умение.

Глава 12 «Сводка алгоритмов»

Обзор всех алгоритмов машинного обучения и статистической обработки, рассмотренных в этой книге. Выполняется также их сравнение в плане применимости к решению нескольких искусственных задач. Это поможет вам понять, как они работают, и визуально проследить, как каждый алгоритм производит разбиение данных.

Приложение A «Дополнительные библиотеки»

Приводится информация о дополнительных библиотеках, использованных в этой книге: где их найти и как установить.

Приложение B «Математические формулы»

Представлены формулы, описания и код для многих математических понятий, встречающихся на страницах настоящей книги.

Упражнения в конце каждой главы содержат идеи о том, как обобщить описанные алгоритмы и сделать их более мощными.

Принятые соглашения

В книге применяются следующие шрифтовые выделения:

Шрифт для элементов интерфейса

Названия пунктов меню, кнопок и «горячих» клавиш (например, используемых в сочетании с Alt и Ctrl).

Курсив

Новые термины, элементы, на которые следует обратить внимание, и адреса веб-сайтов.

`Моноширинный шрифт`

Команды, флаги, переключатели, переменные, атрибуты, клавиши, функции, типы, классы, пространства имен, методы, модули, свойства, параметры, значения, объекты, события, обработчики событий, теги XML и HTML, макросы, содержимое файлов и данные, выводимые командами.

`Моноширинный полужирный`

Команды и иной текст, который пользователь должен вводить буквально.

`Моноширинный курсив`

Текст, вместо которого пользователь должен подставить свои значения.

Под такой иконкой представлены советы, предложения и замечания общего характера.

О примерах кода

Эта книга призвана помогать вам в работе. Поэтому вы можете использовать приведенный в ней код в собственных программах и в документации. Спрашивать у нас разрешение необязательно, если только вы не собираетесь воспроизводить значительную часть кода. Например, не возбраняется включить в свою программу несколько фрагментов кода из книги. Однако для продажи или распространения примеров на компакт-диске разрешение требуется. Цитировать книгу и примеры в ответах на вопросы можно без ограничений. Но для включения значительных объемов кода в документацию по собственному продукту нужно получить разрешение.

Мы высоко ценим, хотя и не требуем, ссылки на наши издания. В ссылке обычно указываются название книги, имя автора, издательство и ISBN, например: «Programming Collective Intelligence by Toby Segaran, Copyright 2007 Toby Segaran, 978-0-596-52932-1».

Если вы полагаете, что планируемое использование кода выходит за рамки изложенной выше лицензии, пожалуйста, обратитесь к нам по адресу permissions@oreilly.com.

Как с нами связаться

Вопросы и замечания по поводу этой книги отправляйте по адресу:

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7.

Тел. (812) 324-53-53

Для этой книги имеется веб-страница, на которой выкладываются списки замеченных ошибок, примеры и разного рода дополнительная информация. Адрес страницы:

http://www.oreilly.com/catalog/9780596529321

Замечания, вопросы технического характера, а также свои пожелания вы можете оставить на нашем сайте

http://www.symbol.ru

Там же можно найти информацию о других наших книгах.

Благодарности

Я хотел бы выразить признательность всем сотрудникам издательства O'Reilly, принимавшим участие в подготовке и выпуске этой книги. Прежде всего хочу поблагодарить Нэта Торкингтона (Nat Torkington), который сказал мне, что идея книги заслуживает внимания; Майка Хендриксона (Mike Hendrickson) и Брайана Джепсона (Brian Jepson) за то, что они проявили к книге интерес и побудили меня взяться за ее написание; и особенно Мэри О'Брайен (Mary O'Brien), которая стала редактором вместо Брайана и всегда умела рассеять мои опасения по поводу того, что этот проект мне не по зубам.

Из производственного коллектива я хотел бы поблагодарить Марлоу Шеффера (Marlowe Shaeffer), Роба Романо (Rob Romano), Джессамин Рид (Jessamyn Read), Эми Томсон (Amy Thomson) и Сару Шнайдер (Sarah Schneider) за то, что они превратили мои рисунки и текст в нечто такое, на что можно взглянуть без отвращения.

Спасибо всем, кто принимал участие в рецензировании книги, особенно Полу Тима (Paul Tyma), Мэттью Расселу (Matthew Russell), Джеффу Хаммербахеру (Jeff Hammerbacher), Терри Камерленго (Terry Camerlengo), Андреасу Вейгенду (Andreas Weigend), Дэниелу Расселлу (Daniel Russell) и Тиму Уолтерсу (Tim Wolters).

Спасибо моим родителям.

И наконец я очень благодарен нескольким своим друзьям, которые помогали мне идеями для этой книги и не обижались, когда у меня не хватало на них времени: Андреа Мэттьюсу (Andrea Matthews), Джеффу Бину (Jeff Beene), Лауре Миякава (Laura Miyakawa), Нейлу Строупу (Neil Stroup) и Бруку Блюмештайну (Brooke Blumenstein). Работа над этой книгой была бы куда сложнее, если бы не ваша поддержка, и уж точно в ней не появилось бы нескольких из наиболее увлекательных примеров.

1

Введение в коллективный разум

Компания Netflix занимается онлайновым прокатом DVD. Пользователь выбирает фильм и заказывает его доставку, а компания рекомендует другие фильмы на основе того, что заказывали другие пользователи. В конце 2006 года компания предложила приз в 1 млн долларов тому, кто улучшит точность системы рекомендования на 10%, причем сумму вознаграждения каждый год предполагалось увеличивать на 50 000 долларов. В конкурсе приняли участие тысячи коллективов со всего мира, и в апреле 2007 года победителю удалось добиться улучшения на 7%. Пользуясь данными о том, какие фильмы нравятся пользователям, Netflix удается рекомендовать своим клиентам такие фильмы, о которых они даже не слышали. В результате люди приходят снова и снова. Поэтому любое усовершенствование системы рекомендования приносит Netflix большие деньги.

Поисковая машина Google была запущена в 1998 году, когда на рынке уже имелось несколько крупных поисковиков. Многие считали, что новичку никогда не догнать гигантов. Но основатели Google придумали совершенно новый подход к ранжированию результатов поиска, основанный на использовании ссылок с миллионов сайтов. Именно так они решали, какие страницы наиболее релевантны запросу. Результаты поиска Google оказались настолько лучше, чем у конкурентов, что к 2004 году этот поисковик обслуживал 85% всех поисковых запросов во Всемирной паутине. Основатели компании теперь занимают место в десятке богатейших людей мира.

Что общего между двумя этими компаниями? И та и другая сумели выстроить свой бизнес на применении изощренных алгоритмов объединения данных, полученных от множества людей. Способность собирать

информацию и наличие вычислительных мощностей для ее интерпретации открыли новые возможности сотрудничества и позволили лучше понять потребности пользователей и заказчиков. Такие задачи возникают повсеместно: сайты знакомств помогают людям быстрее находить себе пару; появляются компании, прогнозирующие изменение цен на авиабилеты, и буквально все жаждут лучше понимать своих клиентов, чтобы проводить целенаправленные рекламные кампании.

И это всего лишь несколько примеров из интереснейшей области коллективного разума. А распространение новых служб означает, что новые возможности открываются ежедневно. Я полагаю, что хорошее владение методами машинного обучения и статистической обработки данных будет становиться все более важным и находить применение во все новых сферах. Но особую значимость приобретает умение организовать и интерпретировать колоссальные массивы информации, создаваемой людьми со всего мира.

Что такое коллективный разум

Выражение *«коллективный разум»* в ходу уже несколько десятилетий, но стало важным и популярным с приходом новых коммуникационных технологий. Оно может вызвать ассоциации с групповым сознанием или сверхъестественными явлениями, но технически ориентированные люди обычно понимают под этим извлечение нового знания из объединенных предпочтений, поведения и представлений некоторой группы людей.

Конечно, коллективный разум был возможен и до появления Интернета. Для того чтобы собирать данные от разрозненных групп людей, объединять их и анализировать, Всемирная паутина совершенно не нужна. В число важнейших форм подобных исследований входят социологические опросы и переписи. Получение ответов от большого числа людей позволяет делать о группе такие статистические выводы, которые на основе единичных данных сделать невозможно. Порождение новых знаний исходя из данных, полученных от независимых респондентов, – это и есть суть коллективного разума.

Хорошим примером могут служить финансовые рынки, где цена устанавливается не по желанию индивидуума или путем скоординированных усилий, а в результате поведения на торгах множества независимых людей, которые действуют в собственных интересах. Хотя на первый взгляд это противоречит интуиции, считается, что *фьючерсные рынки*, где многочисленные участники заключают контракты исходя из своих представлений о будущих ценах, способны предсказывать цены более точно, чем независимые эксперты. Объясняется это тем, что такие рынки аккумулируют знания, опыт и интуицию тысяч людей, а эксперт-одиночка может полагаться только на себя.

Хотя методы коллективного разума существовали и до появления Интернета, возможность получать информацию от тысяч и даже миллионов людей во Всемирной паутине открыла широчайший спектр новых возможностей. В любой момент времени кто-то пользуется Интернетом для совершения покупок, в исследовательских целях, в поисках развлечений или ради создания собственного сайта. Эти действия можно отслеживать и извлекать из них информацию, даже не задавая пользователю вопросов. Существует множество способов обработать и интерпретировать эту информацию. Вот парочка примеров, иллюстрирующих принципиально разные подходы.

- *Википедия* – это онлайновая энциклопедия, создаваемая исключительно самими пользователями. Любой человек может создать новую статью или отредактировать уже существующую, а повторяющиеся попытки недозволенного использования пресекают немногочисленные администраторы. В Википедии больше статей, чем в любой другой энциклопедии, и, несмотря на манипуляции некоторых злонамеренных пользователей, считается, что по большинству тем информация точна. Это пример коллективного разума, поскольку каждая статья поддерживается большой группой людей, а в результате получается творение, намного превосходящее все, что могла бы создать одна организованная группа. Программное обеспечение Википедии не подвергает собранную информации какой-то особо интеллектуальной обработке; оно просто отслеживает изменения и отображает последнюю версию.
- Уже упоминавшаяся система *Google* – самая популярная в мире поисковая машина. В ней впервые для ранжирования веб-страниц был применен подход, основанный на количестве ссылок на данную страницу. Для этого необходимо собрать информацию о том, что думают о данной странице тысячи людей, и воспользоваться ею для сортировки результатов поиска. Это совершенно другой пример коллективного разума. Если Википедия приглашает пользователей внести свой вклад в создание сайта, то Google извлекает информацию из того контента, который пользователи уже разместили на своих собственных сайтах, и применяет ее для генерирования оценок от имени пользователей.

Хотя Википедия – замечательный ресурс и впечатляющий пример коллективного разума, своим существованием она обязана скорее количеству пользователей, готовых пополнять сайт информацией, чем изощренным программным алгоритмам. В этой книге нас будет интересовать противоположный конец спектра, на котором разместились такие алгоритмы, как Google PageRank, где собирается информация о пользователях и с помощью вычислений порождается новая информация, способная сделать работу пользователей более удобной и продуктивной. Некоторые данные собираются явно – скажем, пользователя просят что-то оценить; другие – незаметно, например, наблюдая за

тем, что пользователи покупают. В обоих случаях важно не только собрать и отобразить данные, но и обработать их с целью получения новой информации.

В этой книге будут продемонстрированы способы сбора данных с помощью открытых API и описаны различные алгоритмы машинного обучения и статистической обработки. Сочетание того и другого позволит применить методы коллективного разума к данным, собранным вашими приложениями, а также поэкспериментировать с данными из других источников.

Что такое машинное обучение

Машинное обучение – это одна из дисциплин искусственного интеллекта (ИИ), имеющая дело с алгоритмами обучения компьютеров. В большинстве случаев это означает, что алгоритму подается на вход набор данных, а он выводит информацию о свойствах этих данных, причем так, что на основе выведенной информации способен делать предсказания о данных, которые может увидеть в будущем. Это возможно потому, что практически все неслучайные данные содержат какие-то закономерности (паттерны) и, выявив их, машина способна сделать обобщение. Чтобы наделить машину способностью к обобщению, ей надо задать *модель*, которая определяет, какие аспекты данных считать существенными.

Чтобы понять, как устроена модель, рассмотрим простой пример из довольно сложной области фильтрации почтового спама. Предположим, вы получаете кучу спама, содержащего слова online pharmacy. Вы – человек и от природы наделены способностью распознавать закономерности, поэтому сразу же понимаете, что любое сообщение, содержащее фразу online pharmacy, надлежит отправлять прямиком в Корзину. Это обобщение – вы фактически создали мысленную модель того, что такое спам. После того как вы несколько раз показали, что считаете такие сообщения спамом, алгоритм машинного обучения, предназначенный для фильтрации спама, должен быть способен сделать такое же обобщение.

Существует много алгоритмов машинного обучения, у каждого есть свои сильные и слабые стороны, каждый предназначен для решения определенного класса задач. Некоторые, например деревья решений, прозрачны, то есть наблюдатель понимает весь ход рассуждений машины. Другие, например нейронные сети, представляют собой *«черный ящик»*, то есть выдают ответ, но воспроизвести ход рассуждений часто бывает очень сложно.

В основе многих алгоритмов машинного обучения лежит серьезная математика и статистика. Согласно данному мной ранее определению можно даже сказать, что простой корреляционный и регресионный

анализ – тоже формы машинного обучения. В этой книге не предполагается, что читатель хорошо знаком с математической статистикой, поэтому я старался объяснять применяемые методы настолько просто, насколько возможно.

Ограничения машинного обучения

У машинного обучения есть свои слабости. Алгоритмы разнятся по способности делать обобщения на основе больших наборов паттернов, и если некий паттерн никогда прежде не встречался, то возможна его ошибочная интерпретация. Человек обладает общекультурной подготовкой и опытом, на котором основывает свои суждения, а также уникальной способностью распознавать похожие ситуации, когда принимает решения на основе новой информации. Методы же машинного обучения могут делать обобщения лишь для тех данных, которые уже видели раньше, и даже в этом существенно ограничены.

Метод фильтрации спама, с которым вы ознакомитесь в этой книге, основан на анализе вхождений некоторых слов или фраз без учета их семантики и структуры предложения. Хотя теоретически возможно построить алгоритм, который будет принимать в расчет грамматику, на практике так поступают редко, потому что затраты несоразмерны с улучшением, которого можно достичь. Понимание смысла слов или их соотнесенности с жизнью конкретного человека требует куда больше информации, чем могут получить современные фильтры спама.

Кроме того, все алгоритмы машинного обучения в той или иной мере страдают от проблемы чрезмерного обобщения. Как часто бывает в жизни, строгое обобщение на основе немногих примеров редко оказывается точным. Вполне может статься, что вы получите от друга важное письмо, содержащее слова online pharmacy. В таком случае вы сообщите алгоритму, что это сообщение не является спамом, а он может сделать вывод, что от данного отправителя следует принимать любые сообщения. Природа алгоритмов машинного обучения такова, что они продолжают обучаться по мере поступления новой информации.

Примеры из реальной жизни

В Интернете сейчас есть много сайтов, которые собирают информацию от различных людей и обрабатывают ее методами машинного обучения и математической статистики. Поисковая машина Google – один из ярчайших примеров; она не только использует ссылки для ранжирования страниц, но и постоянно собирает информацию о том, по каким рекламным ссылкам переходили разные пользователи. Это позволяет выдавать рекламу более целенаправленно. В главе 4 вы узнаете о поисковых машинах и алгоритме PageRank, который является важной составной частью системы ранжирования страниц в Google.

Примеры иного рода – это сайты с системой рекомендования. Например, Amazon и Netflix используют информацию о том, что люди покупают или берут напрокат, на ее основе решают, какие люди или товары похожи, а затем предлагают рекомендации исходя из истории покупок. А такие сайты, как Pandora и Last.fm, используют оценки, которые вы ставите разным музыкальным группам и песням, для создания персонализированных радиостанций, которые, по их мнению, транслируют приятную вам музыку. В главе 2 мы рассмотрим способы построения систем рекомендования.

Рынки прогнозов – еще один вид коллективного разума. Один из самых известных среди них – Голливудская фондовая биржа (Hollywood Stock Exchange – *http://hsx.com*), где люди торгуют прогнозами для фильмов и звезд кино. Вы можете купить или продать прогноз по текущей цене, зная, что конечная цена будет равна одной миллионной от реальной суммы кассовых сборов. Поскольку цена устанавливается на торгах, то ее величина не зависит ни от конкретного индивидуума, ни от поведения какой-то группы, так что текущая цена является прогнозом всего множества трейдеров относительно суммы кассовых сборов фильма. Прогнозы Голливудской фондовой биржи обычно оказываются точнее оценок отдельных экспертов.

На некоторых сайтах знакомств, например eHarmony, собранная информация об участниках используется для подбора оптимальной пары. Хотя такие компании обычно держат методы подбора пар в секрете, вполне возможно, что при любом подходе, претендующем на успех, необходимо постоянно пересматривать оценки в зависимости от того, оказались ли подобранные пары удачными.

Другие применения обучающих алгоритмов

Описанные в этой книге методы не новы, и, хотя примеры в основном касаются задач коллективного разума, возникающих в Интернете, владение алгоритмами машинного обучения будет полезно и разработчикам программного обеспечения во многих других областях. Особенно это относится к тем отраслям знания, где требуется отыскивать интересные закономерности в больших наборах данных, например:

Биотехнология

В результате развития технологий секвенирования и скрининга были созданы огромные массивы данных разного вида: последовательности ДНК, белковые структуры и РНК-выражения. Техника машинного обучения активно применяется ко всем этим данным в попытке отыскать закономерности, которые помогли бы понять биологические процессы.

Обнаружение финансового мошенничества

Компании, обслуживающие кредитные карты, постоянно изыскивают новые способы обнаружения мошеннических транзакций. На

сегодня для проверки транзакций и распознавания запрещенного использования применяются такие методы, как нейронные сети и индуктивная логика.

Машинное зрение

Интерпретация изображений с видеокамеры для нужд военных и охраны – это область активных исследований. Для автоматического обнаружения вторжения, идентификации транспортных средств и черт лица применяются различные методы машинного обучения. Особый интерес представляют методы обучения без учителя, например *анализ независимых компонентов*, позволяющие находить интересные паттерны в больших наборах данных.

Маркетинг товаров

В течение очень долгого времени оценка демографического состава покупателей и трендов была скорее искусством, нежели наукой. Но недавно появившиеся методы сбора данных о потребителях открыли возможность применения таких методов машинного обучения, как кластеризация, к задаче выявления естественных разделений на рынках и более точного предсказания будущих трендов.

Оптимизация цепочек поставщиков

Крупные организации могут сэкономить миллионы долларов за счет эффективной организации цепочек поставщиков и точного прогноза спроса на продукцию в различных областях. Есть много способов построить цепочку поставщиков и столько же факторов, способных оказать влияние на спрос. Для анализа таких массивов данных часто применяются методы оптимизации и машинного обучения.

Анализ фондовых рынков

С тех пор как появился фондовый рынок, люди стремились использовать математику для зарабатывания денег. По мере того как участники становились все более изощренными, возникла необходимость в анализе больших наборов данных и применении развитых методов выявления закономерностей.

Национальная безопасность

Объем информации, собираемой правительственными агентствами по всему миру, огромен. Для ее анализа необходимы компьютеры, способные распознавать паттерны и ассоциировать их с потенциальными угрозами.

И это лишь небольшая часть ситуаций, в которых интенсивно применяется машинное обучение. Поскольку информации становится все больше, то весьма вероятно, что в недалеком будущем к методам машинного обучения и статистического анализа будут прибегать все в новых областях, поскольку для обработки информации по старинке человеческих способностей уже недостаточно.

Если учесть, сколько новой информации появляется каждодневно, то возможности этих методов поистине неисчерпаемы. Узнав о некоторых алгоритмах машинного обучения, вы увидите, что применять их можно практически везде.

2

Выработка рекомендаций

Начиная знакомство с коллективным разумом, я хочу показать, как можно использовать предпочтения некоторой группы людей для того, чтобы рекомендовать что-то другим людям. У такой техники немало применений, в частности рекомендование товаров на сайте электронной торговли, указание интересных сайтов или помощь в отыскании нужной музыки и фильмов. В этой главе вы узнаете, как построить систему поиска лиц с общими вкусами и автоматически давать рекомендации на основе того, что нравится другим людям.

Возможно, вы уже встречались с системами рекомендования, когда делали онлайновые покупки на таких сайтах, как Amazon. Amazon отслеживает потребительские привычки всех своих посетителей и, когда вы заходите на сайт, пользуется собранной информацией, чтобы предложить товары, которые могут вас заинтересовать. Amazon может даже предложить фильмы, которые вам, возможно, понравятся, хотя раньше вы покупали только книги. Некоторые сайты по продаже билетов на концерты анализируют, что вы посещали раньше, и анонсируют предстоящие концерты, которые могут быть вам интересны. Такие сайты, как *reddit.com*, позволяют голосовать за ссылки на другие сайты, а затем на основе результатов вашего голосования предлагают другие ссылки, которые вас, возможно, заинтересуют.

Из этих примеров видно, что информацию о предпочтениях можно собирать по-разному. Иногда данными являются купленные посетителем товары, а мнения об этих товарах представляются в виде голосования «да/нет» или оценки по пятибалльной шкале. В этой главе мы рассмотрим различные способы представления рейтинга, работающие с одним и тем же набором алгоритмов, и реализуем примеры из области оценки фильмов и социальных закладок.

Коллаборативная фильтрация

С нетехнологичным способом получить рекомендацию о товаре, фильме или развлекательном сайте вы знакомы. Достаточно спросить у друзей. Знаете вы и о том, что у некоторых ваших друзей вкус лучше, чем у других; вы имели возможность убедиться в этом, поскольку не раз оказывалось, что им нравится то же, что и вам. Но по мере увеличения количества предложений становится все менее практично основывать решение на опросе небольшой группы людей, поскольку они могут просто не знать обо всех имеющихся вариантах. Тут-то и приходит на помощь то, что принято называть *коллаборативной фильтрацией*.

Обычно алгоритм коллаборативной фильтрации работает следующим образом: просматривает большую группу людей и отыскивает в ней меньшую группу с такими же вкусами, как у вас. Он смотрит, какие еще вещи им нравятся, объединяет предпочтения и создает ранжированный список предложений. Есть несколько способов решить, какие люди похожи, и объединить их предпочтения в список. В данной главе мы рассмотрим некоторые из этих способов.

Термин «*коллаборативная фильтрация*» впервые употребил Дэвид Голдберг (David Goldberg) из компании Xerox PARC в 1992 году в статье «Using collaborative filtering to weave an information tapestry». Он спроектировал систему *Tapestry*, которая позволяла людям аннотировать документ как интересный или неинтересный и применяла эту информацию для фильтрации документов, предлагаемых другим людям.
Теперь тот же алгоритм коллаборативной фильтрации применяется на тысячах сайтов для рекомендования фильмов, музыки, книг, знакомств, товаров, других сайтов, подкастов, статей и даже анекдотов.

Сбор информации о предпочтениях

Первое, что нам нужно, – это способ представления людей и их предпочтений. В языке Python это делается очень просто с помощью *вложенного словаря*. Если вы собираетесь проработать пример из этого раздела, создайте файл recommendations.py и введите следующий код для создания набора данных:

```
# Словарь кинокритиков и выставленных ими оценок для небольшого набора
# данных о фильмах
critics={'Lisa Rose': {'Lady in the Water': 2.5, 'Snakes on a Plane': 3.5,
'Just My Luck': 3.0, 'Superman Returns': 3.5, 'You, Me and Dupree': 2.5,
'The Night Listener': 3.0},
'Gene Seymour': {'Lady in the Water': 3.0, 'Snakes on a Plane': 3.5,
'Just My Luck': 1.5, 'Superman Returns': 5.0, 'The Night Listener': 3.0,
'You, Me and Dupree': 3.5},
'Michael Phillips': {'Lady in the Water': 2.5, 'Snakes on a Plane': 3.0,
```

```
    'Superman Returns': 3.5, 'The Night Listener': 4.0},
    'Claudia Puig': {'Snakes on a Plane': 3.5, 'Just My Luck': 3.0,
    'The Night Listener': 4.5, 'Superman Returns': 4.0,
    'You, Me and Dupree': 2.5},
    'Mick LaSalle': {'Lady in the Water': 3.0, 'Snakes on a Plane': 4.0,
    'Just My Luck': 2.0, 'Superman Returns': 3.0, 'The Night Listener': 3.0,
    'You, Me and Dupree': 2.0},
    'Jack Matthews': {'Lady in the Water': 3.0, 'Snakes on a Plane': 4.0,
    'The Night Listener': 3.0, 'Superman Returns': 5.0, 'You, Me and Dupree': 3.5},
    'Toby': {'Snakes on a Plane':4.5,'You, Me and Dupree':1.0,'Superman
Returns':4.0}}
```

В этой главе мы будем работать с языком Python интерактивно, поэтому сохраните файл recommendations.py в таком месте, где его сможет найти интерпретатор Python. Это может быть папка python/Lib, но лучше всего запускать интерпретатор, находясь в той папке, где вы сохранили файл.

В этом словаре критик (и я) выставляет фильму оценку от 1 до 5. Как бы ни было выражено предпочтение, необходимо отобразить его в виде числового значения. Если бы вы создавали сайт для онлайновой торговли, то могли бы использовать 1 как признак того, что посетитель делал покупки в прошлом, и 0 – что не делал. На сайте, где люди голосуют за новостные статьи, значения –1, 0, 1 могли бы означать «не понравилось», «не голосовал», «понравилось» (табл. 2.1).

Таблица 2.1. Возможные отображения действий пользователей на числовые оценки

Билеты на концерт		**Онлайновые покупки**		**Рекомендация сайтов**	
Купил	1	Купил	2	Понравился	1
Не купил	0	Смотрел	1	Не голосовал	0
		Не купил	0	Не понравился	–1

Словарь удобен для экспериментов с алгоритмами и для иллюстрации. В нем легко производить поиск и изменения. Запустите интерпретатор Python и введите несколько команд:

```
c:\code\collective\chapter2> python
Python 2.4.1 (#65, Mar 30 2005, 09:13:57) [MSC v.1310 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>> from recommendations import critics
>> critics['Lisa Rose']['Lady in the Water']
2.5
>> critics['Toby']['Snakes on a Plane']=4.5
>> critics['Toby']
{'Snakes on a Plane':4.5,'You, Me and Dupree':1.0}
```

Хотя в словаре, находящемся в памяти, можно сохранить много предпочтений, большие наборы данных, наверное, лучше хранить в базе.

Отыскание похожих пользователей

Собрав данные о том, что людям нравится, нужно как-то определить, насколько их вкусы схожи. Для этого каждый человек сравнивается со всеми другими и вычисляется *коэффициент подобия* (или *оценка подобия*). Для этого есть несколько способов, я расскажу о двух из них: *евклидовом расстоянии* и *коэффициенте корреляции Пирсона*.

Оценка по евклидову расстоянию

Один из самых простых способов вычисления оценки подобия – это евклидово расстояние. В этом случае предметы, которые люди оценивали сообща, представляются в виде координатных осей. Теперь в этой системе координат можно расположить точки, соответствующие людям, и посмотреть, насколько они оказались близки (рис. 2.1).

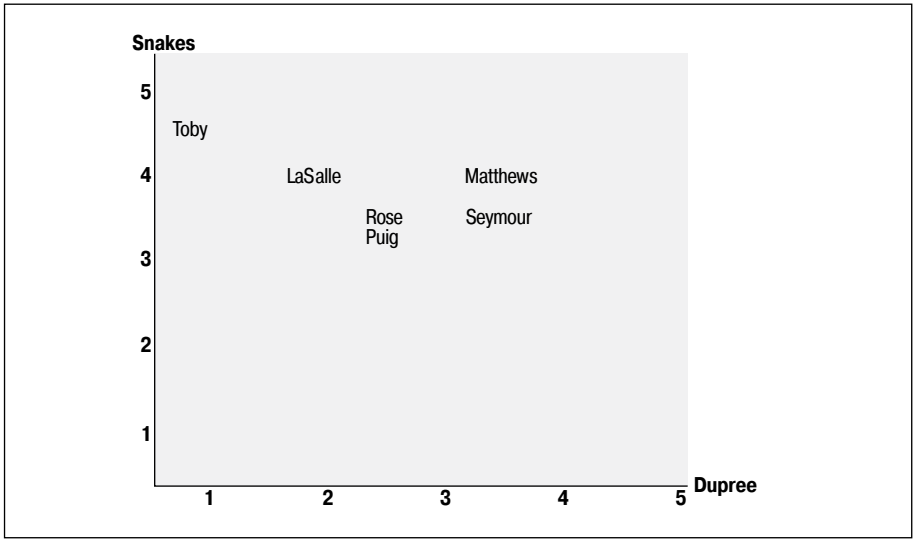


Рис. 2.1. *Люди в пространстве предпочтений*

На этом рисунке показаны люди, представленные точками в пространстве предпочтений. Toby имеет координату 4,5 по оси Snakes и координату 1,0 по оси Dupree. Чем ближе два человека в пространстве предпочтений, тем более схожи их предпочтения. Поскольку эта диаграмма двумерная, то одновременно можно смотреть только на два показателя, но принцип остается тем же самым и для большего числа показателей.

Чтобы вычислить расстояние между Toby и LaSalle на этой диаграмме, возьмем разности координат по каждой оси, возведем их в квадрат,

сложим, а затем извлечем квадратный корень из суммы. В Python для возведения в квадрат можно воспользоваться функцией `pow(n,2)`, а для извлечения квадратного корня служит функция `sqrt`:

```
>> from math import sqrt
>> sqrt(pow(5-4,2)+pow(4-1,2))
3.1622776601683795
```

Расстояние, вычисленное по этой формуле, будет тем меньше, чем больше сходства между людьми. Однако нам нужна функция, значение которой тем больше, чем люди более похожи друг на друга. Этого можно добиться, добавив к вычисленному расстоянию 1 (чтобы никогда не делить на 0) и взяв обратную величину:

```
>> 1/(1+sqrt(pow(5-4,2)+pow(4-1,2)))
0.2402530733520421
```

Новая функция всегда возвращает значение от 0 до 1, причем 1 получается, когда предпочтения двух людей в точности совпадают. Теперь можно собрать все воедино и написать функцию для вычисления оценки подобия. Добавьте в файл recommendations.py такой код:

```
from math import sqrt

# Возвращает оценку подобия person1 и person2 на основе расстояния
def sim_distance(prefs,person1,person2):
  # Получить список предметов, оцененных обоими
  si={}
  for item in prefs[person1]:
    if item in prefs[person2]:
      si[item]=1

  # Если нет ни одной общей оценки, вернуть 0
  if len(si)==0: return 0

  # Сложить квадраты разностей
  sum_of_squares=sum([pow(prefs[person1][item]-prefs[person2][item],2)
                      for item in prefs[person1] if item in prefs[person2]])

  return 1/(1+sum_of_squares)
```

Этой функции при вызове передаются имена двух людей, для которых требуется вычислить оценку подобия. Введите в интерпретаторе Python такую команду:

```
>>> reload(recommendations)
>>> recommendations.sim_distance(recommendations.critics,
...    'Lisa Rose','Gene Seymour')
0.148148148148
```

Это дает оценку подобия между Lisa Rose и Gene Seymour. Попробуйте другие имена, поищите людей, между которыми больше или меньше общего.

Коэффициент корреляции Пирсона

Чуть более сложный способ определить степень схожести интересов людей дает коэффициент корреляции Пирсона. Коэффициент корреляции – это мера того, насколько хорошо два набора данных ложатся на прямую. Формула сложнее, чем для вычисления евклидова расстояния, но она дает лучшие результаты, когда данные плохо нормализованы, например если некоторый критик устойчиво выставляет фильмам более низкие оценки, чем в среднем.

Для визуализации этого метода можете нанести на диаграмму оценки, выставленные двумя критиками, как показано на рис. 2.2. Mick LaSalle оценил фильм «Superman» на 3, а Gene Seymour – на 5, поэтому мы наносим точку (3,5).

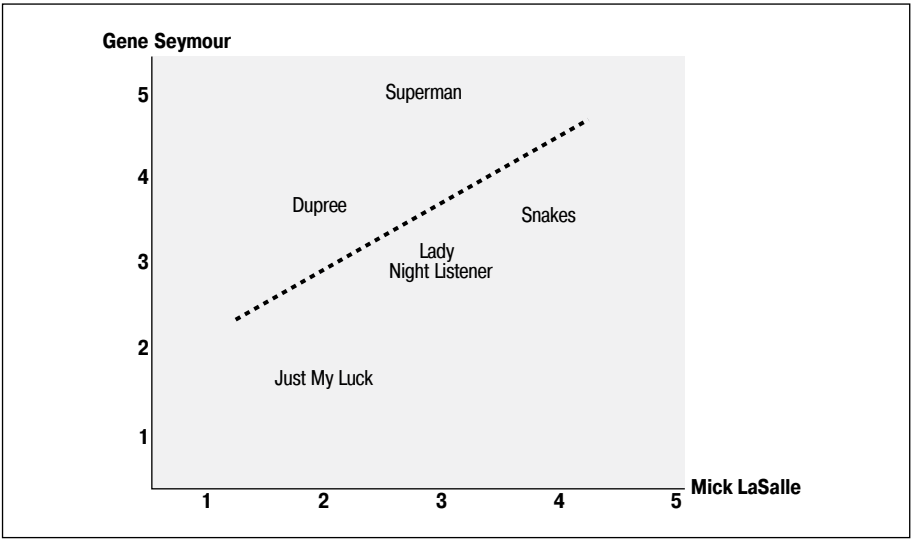


Рис. 2.2. *Сравнение двух кинокритиков на точечной диаграмме*

На диаграмме также изображена прямая линия. Она называется *линией наилучшего приближения*, поскольку проходит настолько близко ко всем точкам на диаграмме, насколько возможно. Если бы оба критика выставили всем фильмам одинаковые оценки, то эта линия оказалась бы диагональной и прошла бы через все точки. В этом случае получилась бы идеальная корреляция с коэффициентом 1. Но в нашем случае критики разошлись в оценках, поэтому коэффициент корреляции равен 0,4. На рис. 2.3 показан пример с гораздо более высоким коэффициентом корреляции 0,75.

У коэффициента корреляции Пирсона есть одно интересное свойство, которое можно наблюдать на рисунке – он корректирует *обесценивание*

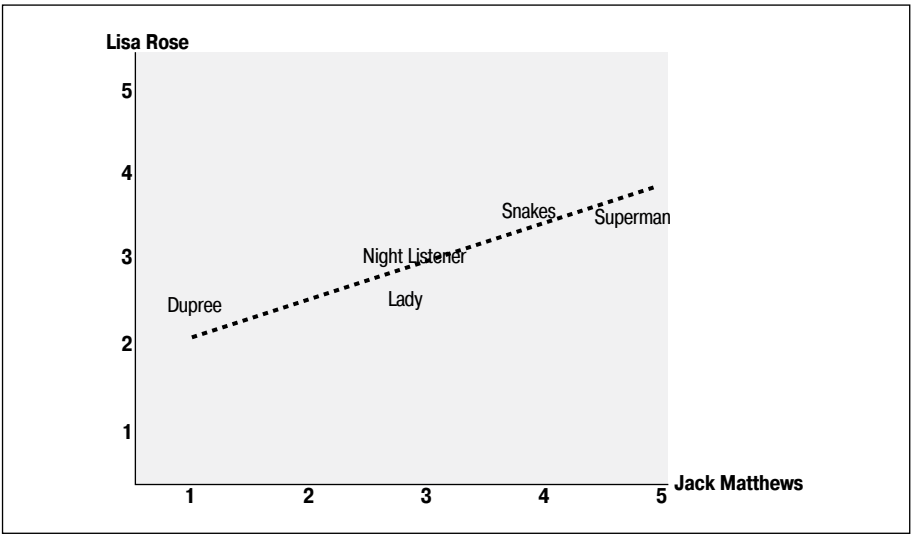


Рис. 2.3. *Два критика с высоким коэффициентом корреляции*

оценок. Видно, что Jack Matthews систематически выставляет более высокие оценки, чем Lisa Rose, но линия все равно проходит близко к точкам, поскольку их предпочтения схожи. Если один критик склонен выставлять более высокие оценки, чем другой, то идеальная корреляция все равно возможна при условии, что разница в оценках постоянна. Метод евклидова расстояния в этом случае выдал бы результат, что критики не похожи, поскольку один всегда оказывается строже другого, несмотря на то что их вкусы, по существу, очень сходны. В зависимости от конкретного приложения такое поведение может вас устраивать или нет.

Программа для вычисления коэффициента корреляции Пирсона сначала находит фильмы, оцененные обоими критиками, и вычисляет сумму и сумму квадратов выставленных ими оценок, а также сумму произведений оценок. На последнем этапе найденные значения используются для вычисления коэффициента корреляции; этот код выделен в листинге ниже полужирным шрифтом. В отличие от евклидовой метрики, эта формула интуитивно не так очевидна.

Чтобы воспользоваться этой формулой, добавьте в файл recommendations.py новую функцию с такой же сигнатурой, как у функции `sim_distance`:

```
# Возвращает коэффициент корреляции Пирсона между p1 и p2
def sim_pearson(prefs,p1,p2):
  # Получить список предметов, оцененных обоими
  si={}
  for item in prefs[p1]:
    if item in prefs[p2]: si[item]=1
```

```
    # Найти число элементов
    n=len(si)

    # Если нет ни одной общей оценки, вернуть 0
    if n==0: return 0

    # Вычислить сумму всех предпочтений
    sum1=sum([prefs[p1][it] for it in si])
    sum2=sum([prefs[p2][it] for it in si])

    # Вычислить сумму квадратов
    sum1Sq=sum([pow(prefs[p1][it],2) for it in si])
    sum2Sq=sum([pow(prefs[p2][it],2) for it in si])

    # Вычислить сумму произведений
    pSum=sum([prefs[p1][it]*prefs[p2][it] for it in si])

    # Вычислить коэффициент Пирсона
    num=pSum-(sum1*sum2/n)
    den=sqrt((sum1Sq-pow(sum1,2)/n)*(sum2Sq-pow(sum2,2)/n))
    if den==0: return 0

    r=num/den

    return r
```

Эта функция возвращает значение от –1 до 1. Значение 1 означает, что два человека выставили каждому предмету в точности одинаковые оценки. В отличие от евклидовой метрики, масштабировать возвращенное значение для приведения к нужному диапазону не требуется. Теперь можете попробовать получить коэффициент корреляции для точек, изображенных на рис. 2.3:

```
>>> reload(recommendations)
>>> print recommendations.sim_pearson(recommendations.critics,
... 'Lisa Rose','Gene Seymour')
0.396059017191
```

Какой оценкой подобия воспользоваться

Я ознакомил вас с двумя разными метриками, но есть и много других способов измерить подобие двух наборов данных. Какой из них оптимален – зависит от конкретного приложения. Имеет смысл попробовать и коэффициент Пирсона, и евклидово расстояние, и другие методы, а потом посмотреть, какой дает наилучшие результаты.

Все последующие функции в этой главе имеют необязательный параметр `similarity`, указывающий на функцию; так проще экспериментировать. Чтобы выбрать ту или иную оценку подобия, задайте для этого параметра значение `sim_pearson` или `sim_vector`. Для вычисления подобия можно брать и различные другие функции, например *коэффициент*

Жаккарда или *манхэттенское расстояние*, при условии что у них такая же сигнатура и возвращаемое значение с плавающей точкой тем больше, чем выше сходство.

Прочитать о других метриках, применяемых для сравнения элементов, можно на странице по адресу *http://en.wikipedia.org/wiki/Metric_%28mathematics%29#Examples*.

Ранжирование критиков

Имея функции для сравнения двух людей, можно написать функцию, которая будет вычислять оценку подобия всех имеющихся людей с данным человеком и искать наилучшее соответствие. В данном случае меня интересуют кинокритики с таким же вкусом, как у меня. Тогда я буду знать, на кого ориентироваться, принимая решение о выборе фильма. Включите в файл recommendations.py следующую функцию, создающую список людей, вкусы которых похожи на вкусы заданного человека:

```
# Возвращает список наилучших соответствий для человека из словаря prefs.
# Количество результатов в списке и функция подобия - необязательные
# параметры.
def topMatches(prefs,person,n=5,similarity=sim_pearson):
  scores=[(similarity(prefs,person,other),other)
                  for other in prefs if other!=person]

  # Отсортировать список по убыванию оценок
  scores.sort( )
  scores.reverse( )
  return scores[0:n]
```

Эта функция сравнивает меня со всеми остальными хранящимися в словаре пользователями с помощью одной из ранее определенных метрик, применяя для этого *трансформацию списка* (list comprehension). И возвращает первые *n* элементов отсортированного списка результатов.

Если вызвать ее, передав мое имя, то она вернет список кинокритиков и оценку подобия со мной для каждого из них:

```
>> reload(recommendations)
>> recommendations.topMatches(recommendations.critics,'Toby',n=3)
[(0.99124070716192991, 'Lisa Rose'), (0.92447345164190486, 'Mick LaSalle'),
 (0.89340514744156474, 'Claudia Puig')]
```

Теперь я знаю, что имеет смысл читать обзоры Lisa Rose, так как ее вкусы больше всего похожи на мои собственные. Если вы смотрели какие-нибудь из указанных фильмов, то можете добавить себя в словарь, прописать свои предпочтения и посмотреть, кто окажется вашим любимым кинокритиком.

Рекомендование предметов

Найти подходящего критика – это, конечно, неплохо, но в действительности-то я хочу, чтобы мне порекомендовали фильм. И прямо сейчас. Можно было бы посмотреть, какие фильмы понравились человеку с похожими на мои вкусами, и выбрать из них те, что я еще не смотрел. Но при таком подходе можно было бы случайно наткнуться на критиков, ничего не писавших о фильмах, которые могли бы мне понравиться. Можно также отобрать критика, которому почему-то понравился фильм, получивший отрицательные отзывы от всех остальных критиков, вошедших в список `topMatches`.

Чтобы разрешить эти проблемы, необходимо ранжировать сами фильмы, вычислив взвешенную сумму оценок критиков. Берем каждого из отобранных критиков и умножаем его оценку подобия со мной на оценку, которую он выставил каждому фильму. В табл. 2.2 показан результат вычислений.

Таблица 2.2. Создание рекомендаций для Тоби

Критик	Подобие	Night	П.х Night	Lady	П.х Lady	Luck	П.х Luck
Rose	0,99	3,0	2,97	2,5	2,48	3,0	2,97
Seymour	0,38	3,0	1,14	3,0	1,14	1,5	0,57
Puig	0,89	4,5	4,02			3,0	2,68
LaSalle	0,92	3,0	2,77	3,0	2,77	2,0	1,85
Matthews	0,66	3,0	1,99	3,0	1,99		
Итого			12,89		8,38		8,07
S подоб.			3,84		2,95		3,18
Итого / S подоб.			3,35		2,83		2,53

В этой таблице приведены коэффициенты корреляции для каждого критика и оценки, поставленные ими трем фильмам («The Night Listener», «Lady in the Water» и «Just My Luck»), которые я сам не оценивал. В столбцах «П.х» находится произведение коэффициента подобия на оценку, выставленную критиком. Смысл в том, чтобы мнение критика с похожими на мои вкусами вносило больший вклад в общую оценку, чем мнение критика, не похожего на меня. В строке «Итого» приведены суммы вычисленных таким образом величин.

Можно было бы использовать для ранжирования сами эти суммы, но тогда фильм, который просмотрело больше людей, получил бы преимущество. Чтобы исправить эту несправедливость, необходимо разделить

полученную величину на сумму коэффициентов подобия для всех критиков, которые рецензировали фильм (строка «S подоб.» в таблице) Поскольку фильм «The Night Listener» рецензировали все, величина «Итого» для него делится на сумму всех коэффициентов подобия. Напротив, фильм «Lady in the Water» критик Puig не рецензировал, следовательно, в этом случае величина «Итого» делится на сумму коэффициентов подобия всех критиков, кроме Puig. В последней строке показано частное от деления.

Код, реализующий этот алгоритм, абсолютно прямолинеен и может работать как с евклидовым расстоянием, так и с коэффициентом корреляции Пирсона. Добавьте его в файл recommendations.py:

```
# Получить рекомендации для заданного человека, пользуясь взвешенным средним
# оценок, данных всеми остальными пользователями
def getRecommendations(prefs,person,similarity=sim_pearson):
  totals={}
  simSums={}
  for other in prefs:
    # сравнивать меня с собой же не нужно
    if other==person: continue
    sim=similarity(prefs,person,other)

  # игнорировать нулевые и отрицательные оценки
  if sim<=0: continue
  for item in prefs[other]:

    # оценивать только фильмы, которые я еще не смотрел
    if item not in prefs[person] or prefs[person][item]==0:
      # Коэффициент подобия * Оценка
      totals.setdefault(item,0)
      totals[item]+=prefs[other][item]*sim
      # Сумма коэффициентов подобия
      simSums.setdefault(item,0)
      simSums[item]+=sim

  # Создать нормализованный список
  rankings=[(total/simSums[item],item) for item,total in totals.items( )]

  # Вернуть отсортированный список
  rankings.sort( )
  rankings.reverse( )
  return rankings
```

Здесь мы в цикле обходим всех людей, присутствующих в словаре `prefs`. Для каждого вычисляется коэффициент подобия с заданным человеком `person`. Далее обходятся все фильмы, которым текущий критик выставил оценку. В строке, выделенной полужирным шрифтом, вычисляется окончательная оценка фильма – оценка, данная каждым критиком, умножается на коэффициент подобия этого критика и произведения суммируются. В самом конце оценки нормализуются путем деления на сумму коэффициентов подобия и возвращается отсортированный список результатов.

Теперь можно узнать, какие фильмы мне следует посмотреть в ближайшем будущем:

```
>>> reload(recommendations)
>>> recommendations.getRecommendations(recommendations.critics,'Toby')
[(3.3477895267131013, 'The Night Listener'), (2.8325499182641614, 'Lady in
the Water'), (2.5309807037655645, 'Just My Luck')]
>>> recommendations.getRecommendations(recommendations.critics,'Toby',
... similarity=recommendations.sim_distance)
[(3.5002478401415877, 'The Night Listener'), (2.7561242939959363, 'Lady in
the Water'), (2.4619884860743739, 'Just My Luck')]
```

Мы получили не только ранжированный список фильмов, но и прогноз оценок, которые я поставлю каждому из них. Имея такую информацию, я могу решить, стоит ли вообще смотреть фильм или лучше заняться каким-нибудь другим делом. В зависимости от приложения вы можете вообще не давать рекомендаций, если не нашлось ничего, отвечающего стандартам данного пользователя. Поэкспериментировав, вы обнаружите, что выбор метрики подобия влияет на результаты очень слабо.

Итак, построена полная система рекомендования, способная работать с товарами любого вида или со ссылками. Необходимо лишь заполнить словарь, поместив в него людей, предметы и оценки, а затем его можно использовать для рекомендования предметов любому пользователю. Ниже в этой главе вы увидите, как с помощью API сайта *del.icio.us* получить реальные данные для рекомендования различных веб-сайтов.

Подбор предметов

Теперь вы знаете, как искать похожих людей и рекомендовать предметы данному человеку. Но что если нужно узнать, какие предметы похожи друг на друга? Вы могли столкнуться с такой ситуацией на сайтах онлайновой торговли, особенно если сайт еще не собрал о вас достаточно информации. На рис. 2.4 показан фрагмент страницы сайта Amazon для книги «Programming Python».

Рис. 2.4. *Amazon показывает книги, похожие на «Programming Python»*

В данном случае вы можете определить степень сходства, выявив людей, которым понравился данный товар, и посмотрев, что еще им понравилось. По существу, это тот же метод, которым мы уже пользовались для определения похожести людей, – нужно лишь вместо людей всюду подставить товары. Стало быть, мы сможем применить уже написанные функции, если преобразуем словарь, заменив

```
{'Lisa Rose': {'Lady in the Water': 2.5, 'Snakes on a Plane': 3.5},
'Gene Seymour': {'Lady in the Water': 3.0, 'Snakes on a Plane': 3.5}}
```

на

```
{'Lady in the Water':{'Lisa Rose':2.5,'Gene Seymour':3.0},
'Snakes on a Plane':{'Lisa Rose':3.5,'Gene Seymour':3.5}} и т. д.
```

Добавьте в файл recommendations.py функцию для выполнения такого преобразования:

```
def transformPrefs(prefs):
  result={}
  for person in prefs:
    for item in prefs[person]:
      result.setdefault(item,{})

      # Обменять местами человека и предмет
      result[item][person]=prefs[person][item]
  return result
```

А теперь вызовем функцию `topMatches`, чтобы найти фильмы, похожие на «Superman Returns»:

```
>> reload(recommendations)
>> movies=recommendations.transformPrefs(recommendations.critics)
>> recommendations.topMatches(movies,'Superman Returns')
[(0.657, 'You, Me and Dupree'), (0.487, 'Lady in the Water'), (0.111, 'Snakes
on a Plane'), (-0.179, 'The Night Listener'), (-0.422, 'Just My Luck')]
```

Обратите внимание, что в этом примере встречаются отрицательные коэффициенты корреляции. Это означает, что тем, кому нравится фильм «Superman Returns», фильм «Just My Luck» обычно не нравится (рис. 2.5).

Можно пойти еще дальше и получить рекомендуемых критиков для данного фильма. Быть может, так вы решите, кого приглашать на премьеру?

```
>> recommendations.getRecommendations(movies,'Just My Luck')
[(4.0, 'Michael Phillips'), (3.0, 'Jack Matthews')]
```

Не всегда очевидно, что перестановка людей и товаров приведет к полезным результатам, но во многих случаях это позволяет провести интересные сравнения. Сайт онлайновой торговли может хранить истории покупок, чтобы рекомендовать товары посетителям. В этом случае описанная выше перестановка людей и товаров позволит найти людей, которые могли бы купить определенный товар. Это может оказаться очень полезным при планировании маркетинговых акций для продвижения товаров. Еще одно потенциальное применение – рекомендование новых ссылок посетителям, которые могли бы ими заинтересоваться.

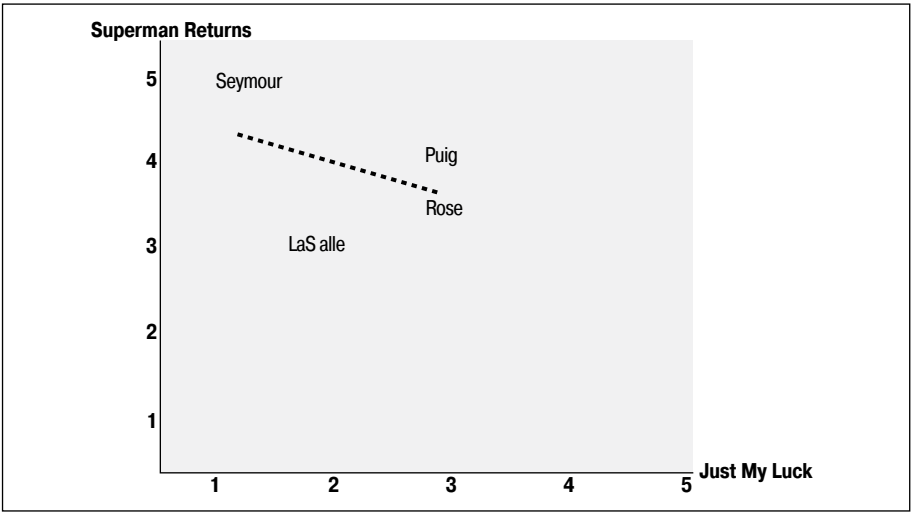


Рис. 2.5. *Между фильмами «Superman Returns» и «Just My Luck» наблюдается отрицательная корреляция*

Построение рекомендателя ссылок с помощью API сайта del.icio.us

В этом разделе я покажу, как запрашивать данные с одного из наиболее популярных сайтов онлайновых закладок и как с их помощью находить похожих пользователей и рекомендовать ссылки, которых они раньше не видели. Сайт *http://del.icio.us* позволяет пользователю создать учетную запись и сохранять интересующие его ссылки. Зайдя на этот сайт, вы можете посмотреть, какие ссылки сохранили другие люди, а также найти «популярные» ссылки, сохраненные многими пользователями. На рис. 2.6 показана одна из страниц сайта *del.icio.us*.

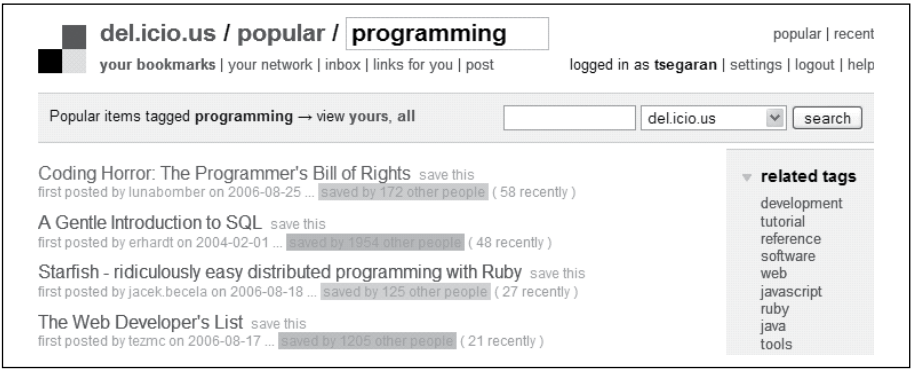


Рис. 2.6. *Страница сайта del.icio.us, содержащая популярные ссылки на тему программирования*

В отличие от некоторых других сайтов обмена ссылками, *del.icio.us* (в настоящее время) не позволяет искать похожих людей и не рекомендует ссылки, которые могли бы вам понравиться. Впрочем, с помощью рассмотренных в этой главе методов вы можете добавить эту функциональность самостоятельно.

API сайта del.icio.us

Сайт *del.icio.us* предоставляет API для доступа к данным, которые возвращаются в формате XML. Чтобы упростить вам жизнь еще больше, имеется обертка этого API на языке Python в виде библиотеки, которую можно скачать со страницы *http://code.google.com/p/pydelicious/source* или *http://oreilly.com/catalog/9780596529321*.

Для работы с примером из этого раздела загрузите последнюю версию этой библиотеки и поместите ее в папку, где интерпретатор Python ищет библиотеки (подробнее об установке этой библиотеки см. приложение А).

В библиотеке есть несколько простых функций для получения ссылок, сохраненных пользователями. Например, чтобы получить последний список популярных ссылок на тему программирования, следует вызвать функцию `get_popular`:

```
>> import pydelicious
>> pydelicious.get_popular(tag='programming')
[{'count': '', 'extended': '', 'hash': '', 'description': u'How To Write
Unmaintainable Code', 'tags': '', 'href': u'http://thc.segfault.net/root/
phun/unmaintain.html', 'user': u'dorsia', 'dt': u'2006-08-19T09:48:56Z'},
{'count': '', 'extended': '', 'hash': '', 'description': u'Threading in C#',
'tags': '', 'href': u'http://www.albahari.com/threading/', 'user':
u'mmihale','dt': u'2006-05-17T18:09: 24Z'},
...и т. д. ...
```

Как видите, возвращается список словарей, каждый из которых содержит URL, описание и имя пользователя, который разместил ссылку. Поскольку мы работаем с живыми данными, полученные вами результаты могут отличаться от приведенных в книге. Есть еще две полезные функции: `get_urlposts` возвращает все хранящиеся ссылки на данный URL, а `get_userposts` – все ссылки, сохраненные данным пользователем. Данные возвращаются в том же формате, что и выше.

Построение набора данных

Загрузить все данные о ссылках с сайта *del.icio.us* невозможно, поэтому придется ограничиться каким-то подмножеством. Вы можете выбрать его как угодно, но, чтобы получить интересные результаты, нам хотелось бы найти людей, которые часто сохраняют ссылки, и тех, чьи наборы ссылок похожи.

Для этого можно, например, запросить список пользователей, которые недавно сохранили популярную ссылку с конкретным признаком. Создайте файл deliciousrec.py и введите в него такой код:

```
from pydelicious import get_popular,get_userposts,get_urlposts

def initializeUserDict(tag,count=5):
  user_dict={}
  # получить подсчет самых популярных ссылок
  for p1 in get_popular(tag=tag)[0:count]:
    # найти всех пользователей, сохранивших эту ссылку
    for p2 in get_urlposts(p1['href']):
      user=p2['user']
      user_dict[user]={}
  return user_dict
```

В результате мы получаем словарь, содержащий нескольких пользователей; каждый из них ссылается на пустой словарь, который предстоит заполнить ссылками. API возвращает только 30 человек, сохранивших ссылку последними, поэтому мы запрашиваем списки пользователей для пяти разных ссылок, чтобы набор данных получился более представительным.

В отличие от примера с кинокритиками, в данном случае есть всего две возможные оценки: 0, если пользователь не сохранял ссылку, и 1 – если сохранял. С помощью API мы теперь готовы написать функцию, которая вычислит оценки для всех пользователей. Добавьте следующий код в файл deliciousrec.py:

```
def fillItems(user_dict):
  all_items={}
  # Найти ссылки, сохраненные всеми пользователями
  for user in user_dict:
    for i in range(3):
      try:
        posts=get_userposts(user)
        break
      except:
        print "Ошибка для пользователя "+user+", пробую еще раз"
        time.sleep(4)
    for post in posts:
      url=post['href']
      user_dict[user][url]=1.0
      all_items[url]=1

  # Вместо отсутствующих элементов записать 0
  for ratings in user_dict.values( ):
    for item in all_items:
      if item not in ratings:
        ratings[item]=0.0
```

Этой функцией можно воспользоваться для построения набора данных, аналогичного словарю `critics`, созданному в начале этой главы:

```
>> from deliciousrec import *
>> delusers=initializeUserDict('programming')
>> delusers ['tsegaran']={} # Добавьте в словарь себя, если пользуетесь сайтом
delicious
>> fillItems(delusers)
```

В третьей строке в список добавляется пользователь `tsegaran`. Можете вместо него указать собственный идентификатор, если вы зарегистрированы на сайте *del.icio.us*.

Обращение к функции `fillItems` может занять некоторое время, поскольку сайту посылается несколько сотен запросов. Иногда API перестает обрабатывать запросы, если они посылаются слишком быстро. В таком случае программа приостанавливается и пытается еще раз получить информацию о том же пользователе (но делает не более трех попыток).

Рекомендование соседей и ссылок

Построив набор данных, мы можем применить к нему те же функции, что и к набору данных о кинокритиках. Чтобы выбрать пользователя наугад и найти других пользователей с похожими вкусами, введите следующие команды:

```
>> import random
>> user=delusers.keys( )[random.randint(0,len(delusers)-1)]
>> user
u'veza'
>> recommendations.topMatches(delusers,user)
[(0.083, u'kuzz99'), (0.083, u'arturoochoa'), (0.083, u'NickSmith'), (0.083,
u'MichaelDahl'), (0.050, u'zinggoat')]
```

Можно также получить рекомендованные ссылки для этого пользователя, вызвав функцию `getRecommendations`. Она возвращает упорядоченный список всех ссылок, поэтому лучше ограничиться первыми десятью:

```
>> recommendations.getRecommendations(delusers,user)[0:10]
[(0.278, u'http://www.devlisting.com/'),
(0.276, u'http://www.howtoforge.com/linux_ldap_authentication'),
(0.191,      u'http://yarivsblog.com/articles/2006/08/09/secret-weapons-for-
startups'),
(0.191, u'http://www.dadgum.com/james/performance.html'),
(0.191, u'http://www.codinghorror.com/blog/archives/000666.html')]
```

Разумеется, как и раньше, список предпочтений можно транспонировать. Это позволит нам формулировать запросы в терминах ссылок, а не людей. Для поиска ссылок, похожих на ту, что показалась вам особенно интересной, попробуйте ввести такие команды:

```
>> url=recommendations.getRecommendations(delusers,user)[0][1]
>> recommendations.topMatches(recommendations.transformPrefs(delusers),url)
[(0.312, u'http://www.fonttester.com/'),
(0.312, u'http://www.cssremix.com/'),
(0.266, u'http://www.logoorange.com/color/color-codes-chart.php'),
(0.254, u'http://yotophoto.com/'),
(0.254, u'http://www.wpdfd.com/editorial/basics/index.html')]
```

Вот и все! Вы только что добавили механизм рекомендования для сайта *del.icio.us*. Но можно еще много чего сделать. Поскольку *del.icio.us* поддерживает поиск по признакам (в его терминологии – тегам (tag)), можно поискать похожие признаки. Можно даже поискать людей, которые пытаются манипулировать страницами «популярных ссылок», сохраняя одни и те же ссылки под разными учетными записями.

Фильтрация по схожести образцов

Мы реализовали механизм рекомендования таким образом, что для создания набора данных необходимы оценки, выставленные каждым пользователем. Для нескольких тысяч людей или предметов это, возможно, и будет работать, но на таком большом сайте, как Amazon, миллионы пользователей и товаров, поэтому сравнение каждого пользователя со всеми другими, а затем сравнение товаров, которым каждый пользователь выставил оценки, займет недопустимо много времени. Кроме того, на сайте, который продает миллионы разных товаров, перекрытие вкусов может быть очень мало́, поэтому нелегко решить, какие пользователи похожи.

Техника, которую мы применяли до сих пор, называется *коллаборативной фильтрацией по схожести пользователей*. Альтернатива известна под названием «*коллаборативная фильтрация по схожести образцов*». Когда набор данных очень велик, коллаборативная фильтрация по схожести образцов может давать лучшие результаты, причем многие вычисления можно выполнить заранее, поэтому пользователь получит рекомендации быстрее.

Процедура фильтрации по схожести образцов во многом основана на уже рассмотренном материале. Основная идея заключается в том, чтобы для каждого образца заранее вычислить большинство похожих на него. Тогда для выработки рекомендаций пользователю достаточно будет найти те образцы, которым он выставил наивысшие оценки, и создать взвешенный список образцов, максимально похожих на эти. Отметим одно существенное отличие: хотя на первом шаге необходимо исследовать все данные, результаты сравнения образцов изменяются не так часто, как результаты сравнения пользователей. Это означает, что не нужно постоянно пересчитывать для каждого образца список похожих на него; это можно делать, когда нагрузка на сайт невелика, или вообще на отдельном компьютере.

Построение набора данных для сравнения образцов

Чтобы сравнивать образцы, нужно первым делом написать функцию, которая построит полный набор данных о похожих образцах. Еще раз повторим, что это необязательно делать при выработке каждой рекомендации; достаточно построить набор один раз, чтобы пользоваться им многократно.

Для генерирования набора данных включите в файл recommendations.ру следующую функцию:

```
def calculateSimilarItems(prefs,n=10):
  # Создать словарь, содержащий для каждого образца те образцы, которые
  # больше всего похожи на него.
  result={}

  # Обратить матрицу предпочтений, чтобы строки соответствовали образцам
  itemPrefs=transformPrefs(prefs)
  c=0
  for item in itemPrefs:
    # Обновление состояния для больших наборов данных
    c+=1
    if c%100==0: print "%d / %d" % (c,len(itemPrefs))
    # Найти образцы, максимально похожие на данный
    scores=topMatches(itemPrefs,item,n=n,similarity=sim_distance)
    result[item]=scores
  return result
```

Эта функция сначала обращает словарь предпочтений, вызывая написанную ранее функцию `transformPrefs`, которой передается список образцов вместе с оценками, выставленными каждым пользователем. Далее в цикле обходятся все образцы и трансформированный словарь передается функции `topMatches`, которая возвращает наиболее похожие образцы и коэффициенты подобия для них. Наконец функция создает и возвращает словарь, в котором каждому образцу сопоставлен список наиболее похожих на него образцов.

В сеансе работы с интерпретатором Python постройте набор данных о схожести образцов и посмотрите, что получится:

```
>>> reload(recommendations)
>>> itemsim=recommendations.calculateSimilarItems(recommendations.critics)
>>> itemsim
{'Lady in the Water': [(0.40000000000000002, 'You, Me and Dupree'),
                       (0.2857142857142857, 'The Night Listener'),...
 'Snakes on a Plane': [(0.22222222222222221, 'Lady in the Water'),
                       (0.18181818181818182, 'The Night Listener'),...
 и т. д.
```

Напомним, что эту функцию следует запускать лишь тогда, когда необходимо обновить данные о схожести образцов. Пока количество пользователей и выставленных ими оценок невелико, это имеет смысл делать чаще, но по мере роста числа пользователей коэффициенты подобия образцов обычно перестают сильно изменяться.

Выдача рекомендаций

Теперь вы готовы выдавать рекомендации, пользуясь словарем данных о схожести образцов без обращения ко всему набору данных. Необходимо получить список всех образцов, которым пользователь выставлял оценки, найти похожие и взвесить их с учетом коэффициентов подобия.

В табл. 2.3 показана процедура выработки рекомендаций на основе фильтрации по схожести образцов. В отличие от табл. 2.2, критики тут вообще не участвуют, а сравниваются фильмы, которые я оценивал, с теми, которые не оценивал.

Таблица 2.3. Рекомендации для Toby, полученные методом фильтрации по схожести образцов

Фильм	Оценка	Night	O.x Night	Lady	O.x Lady	Luck	O.x Luck
«Snakes»	4,5	0,182	0,818	0,222	0,999	0,105	0,474
«Superman»	4,0	0,103	0,412	0,091	0,363	0,065	0,258
«Dupree»	1,0	0,148	0,148	0,4	0,4	0,182	0,182
Итого		0,433	1,378	0,713	1,764	0,352	0,914
После нормализации			3,183		2,598		2,473

В каждой строке указан фильм, который я смотрел, и оценка, которую я ему выставил. Для каждого фильма, который я не смотрел, имеется столбец, где показано, насколько он похож на виденные мной фильмы. Например, коэффициент подобия между фильмами «Superman» и «The Night Listener» равен 0,103. В столбцах с названиями, начинающимися с O.x, показана моя оценка, умноженная на коэффициент подобия; поскольку я поставил фильму «Superman» оценку 4,0, то, умножая число на пересечении строки «Superman» и столбца «Night» на 4,0, получаем: $4{,}0 \times 0{,}103 = 0{,}412$.

В строке «Итого» просуммированы коэффициенты подобия и значения в столбцах «O.x» для каждого фильма. Чтобы предсказать мою оценку фильма, достаточно разделить итог для колонки «O.x» на суммарный коэффициент подобия. Так, для фильма «The Night Listener» прогноз моей оценки равен $1{,}378/0{,}433 = 3{,}183$.

Реализуем эту функциональность, добавив последнюю функцию в файл recommendations.py:

```
def getRecommendedItems(prefs,itemMatch,user):
  userRatings=prefs[user]
  scores={}
  totalSim={}

  # Цикл по образцам, оцененным данным пользователем
  for (item,rating) in userRatings.items( ):

    # Цикл по образцам, похожим на данный
    for (similarity,item2) in itemMatch[item]:

      # Пропускаем, если пользователь уже оценивал данный образец
      if item2 in userRatings: continue
```

```
        # Взвешенная суммы оценок, умноженных на коэффициент подобия
        scores.setdefault(item2,0)
        scores[item2]+=similarity*rating

        # Сумма всех коэффициентов подобия
        totalSim.setdefault(item2,0)
        totalSim[item2]+=similarity

    # Делим каждую итоговую оценку на взвешенную сумму, чтобы вычислить
    # среднее
    rankings=[(score/totalSim[item],item) for item,score in scores.items( )]

    # Возвращает список rankings, отсортированный по убыванию
    rankings.sort( )
    rankings.reverse( )
    return rankings
```

Протестируйте эту функцию на построенном ранее наборе данных о схожести предметов, чтобы получить новые рекомендации для Toby:

```
>> reload(recommendations)
>> recommendations.getRecommendedItems(recommendations.critics,itemsim,'Toby')
[(3.182, 'The Night Listener'),
 (2.598, 'Just My Luck'),
 (2.473, 'Lady in the Water')]
```

Фильм «The Night Listener» по-прежнему лидирует с большим отрывом, а «Just My Luck» и «Lady in the Water» поменялись местами, но остались близки. Важнее тот факт, что функции `getRecommendedItems` не пришлось вычислять коэффициенты подобия для всех остальных критиков, поскольку нужный набор данных был построен заранее.

Использование набора данных MovieLens

В последнем примере мы рассмотрим реальный набор данных с оценками фильмов, который называется MovieLens. Этот набор был подготовлен в ходе работы над проектом GroupLens в университете штата Миннесота. Загрузить его можно со страницы *http://www.grouplens.org/node/12*. Там есть два набора данных. Скачайте набор 100 000 в формате tar.gz или zip в зависимости от платформы, на которой вы работаете.

В архиве упаковано несколько файлов, но для нас представляют интерес только u.item, в котором содержится список идентификаторов и названий фильмов, и u.data, где находятся собственно оценки в следующем формате:

```
196  242  3   881250949
186  302  3   891717742
22   377  1   878887116
244  51   2   880606923
166  346  1   886397596
298  474  4   884182806
```

В каждой строке указан идентификатор пользователя, идентификатор фильма, оценка, выставленная фильму данным пользователем, и временной штамп. Получить список названий фильмов можно, но пользователи анонимны, поэтому нам придется работать только с их идентификаторами. В наборе имеются оценки 1682 фильмов, данные 943 пользователями, каждый из которых оценил не менее 20 фильмов.

Создайте в файле recommendations.py новый метод `loadMovieLens` для загрузки этого набора данных:

```
def loadMovieLens(path='/data/movielens'):

  # Получить названия фильмов
  movies={}
  for line in open(path+'/u.item'):
    (id,title)=line.split('|')[0:2]
    movies[id]=title

  # Загрузить данные
  prefs={}
  for line in open(path+'/u.data'):
    (user,movieid,rating,ts)=line.split('\t')
    prefs.setdefault(user,{})
    prefs[user][movies[movieid]]=float(rating)
  return prefs
```

В сеансе работы с интерпретатором загрузите данные и посмотрите оценки, выставленные каким-нибудь пользователем:

```
>>> reload(recommendations)
>>> prefs=recommendations.loadMovieLens( )
>>> prefs['87']
{'Birdcage, The (1996)': 4.0, 'E.T. the Extra-Terrestrial (1982)': 3.0,
 'Bananas (1971)': 5.0, 'Sting, The (1973)': 5.0, 'Bad Boys (1995)': 4.0,
 'In the Line of Fire (1993)': 5.0, 'Star Trek: The Wrath of Khan
(1982)':5.0,
  'Speechless (1994)': 4.0, и т. д...
```

Теперь можно получить рекомендации путем фильтрации по схожести пользователей:

```
>>> recommendations.getRecommendations(prefs,'87')[0:30]
[(5.0, 'They Made Me a Criminal (1939)'), (5.0, 'Star Kid (1997)'),
 (5.0, 'Santa with Muscles (1996)'), (5.0, 'Saint of Fort Washington
(1993)'),
  и т. д...]
```

Если у вас не очень быстрый компьютер, то при выработке рекомендаций таким способом возникнет небольшая пауза. Связано это с тем, что теперь вы работаете с гораздо более объемным набором данных. Чем больше пользователей, тем больше времени будет занимать процедура выработки рекомендаций. А теперь попробуем фильтрацию по схожести образцов:

```
>>> itemsim=recommendations.calculateSimilarItems(prefs,n=50)
100 / 1664
```

```
200 / 1664
и т. д...
>>> recommendations.getRecommendedItems(prefs,itemsim,'87')[0:30]
[(5.0, "What's Eating Gilbert Grape (1993)"), (5.0, 'Vertigo (1958)'),
 (5.0, 'Usual Suspects, The (1995)'), (5.0, 'Toy Story (1995)'),и т. д...]
```

Хотя на построение словаря данных о коэффициентах подобия уходит много времени, но после того как он построен, выработка рекомендаций производится практически мгновенно. И, что очень важно, время не увеличивается с ростом числа пользователей.

Этот набор данных очень удобен для экспериментов с целью понять, как различные методы оценки подобия влияют на результат, и сравнить производительность фильтрации по схожести пользователей и образцов. На сайте GroupLens есть и другие наборы данных, в том числе о книгах и анекдотах, а также еще один набор данных о кинофильмах.

Сравнение методов фильтрации по схожести пользователей и по схожести образцов

Фильтрация по схожести образцов выполняется гораздо быстрее, чем по схожести пользователей, когда нужно выработать список рекомендаций на большом наборе данных, но она требует дополнительных накладных расходов на хранение таблицы коэффициентов подобия образцов. И точность зависит от того, насколько «разрежен» набор данных. В примере с фильмами каждый критик оценил почти все фильмы, поэтому набор данных плотный (не разреженный). С другой стороны, маловероятно, что на сайте *del.icio.us* найдутся два человека с одинаковым набором закладок, – большинство закладок сохраняется небольшой группой людей, поэтому набор данных оказывается разреженным. На разреженных наборах данных фильтрация по схожести образцов работает быстрее, чем по схожести пользователей, а на плотных наборах их производительность почти одинакова.

Дополнительную информацию о различии в производительности этих двух алгоритмов можно найти в статье Sarwar и др. «Item-based Collaborative Filtering Recommendation Algorithms» на сайте *http://citeseer.ist.psu.edu/sarwar01itembased.html*.

Однако фильтрацию по схожести пользователей проще реализовать, и она не требует дополнительных шагов, поэтому зачастую она более предпочтительна для небольших наборов данных, умещающихся целиком в памяти, которые к тому же очень быстро изменяются. Наконец, в некоторых приложениях отыскание людей, предпочтения которых схожи с предпочтениями данного пользователя, имеет самостоятельную ценность – на сайте онлайновой торговли это, может быть, и ни к чему, а вот на сайтах обмена ссылками или рекомендования музыки было бы очень кстати.

Теперь вы знаете, как вычислять коэффициенты подобия и пользоваться ими для сравнения людей и предметов. В этой главе мы рассмотрели два алгоритма выработки рекомендаций – по схожести пользователей и по схожести образцов, а также способы сохранения предпочтений пользователей. Кроме того, вы воспользовались API сайта *del.icio.us* для построения системы рекомендования ссылок. В главе 3 вы узнаете, как на основе идей из этой главы находить группы похожих людей, применяя алгоритмы кластеризации без учителя. В главе 9 мы рассмотрим альтернативные способы подбора пары человеку, когда заранее известно о том, какие люди ему нравятся.

Упражнения

1. *Коэффициент Танимото*. Выясните, что такое коэффициент подобия Танимото. В каких случаях его можно использовать в качестве метрики схожести вместо евклидова расстояния или коэффициента Пирсона? Напишите новую функцию оценки подобия на основе коэффициента Танимото.
2. *Подобие признаков*. Пользуясь API сайта *del.icio.us*, создайте набор данных, содержащий признаки и ссылки. Воспользуйтесь им, чтобы вычислить коэффициенты подобия признаков. Попробуйте отыскать почти идентичные признаки. Найдите несколько ссылок, которые можно было бы пометить признаком programming, но это не сделано.
3. *Эффективность фильтрации по схожести пользователей*. Алгоритм фильтрации по схожести пользователей неэффективен, потому что сравнивает данного пользователя со всеми остальными всякий раз, когда нужно выработать рекомендацию. Напишите функцию, которая будет заранее вычислять коэффициенты подобия пользователей, и измените код выработки рекомендаций так, чтобы в этом процессе участвовали только пять пользователей с наибольшими коэффициентами подобия.
4. *Фильтрация закладок по схожести образцов*. Загрузите набор данных с сайта *del.icio.us* и сохраните его в базе данных. Создайте таблицу образец–образец и воспользуйтесь ею для выработки рекомендаций различным пользователям на основе схожести образцов. Насколько полученные рекомендации будут отличаться от тех, что были выработаны на основе схожести пользователей?
5. *Сайт Audioscrobbler*. Загляните на сайт *http://www.audioscrobbler.net*, где имеется набор данных о музыкальных предпочтениях большого числа пользователей. Воспользуйтесь предоставляемым API в виде веб-служб, чтобы загрузить данные и построить систему рекомендования музыки.

3

Обнаружение групп

В главе 2 мы изучали, как найти тесно связанные между собой объекты, например людей, которым нравятся такие же фильмы, как и вам. В этой главе мы разовьем эти идеи и ознакомимся с *кластеризацией данных* – методом обнаружения и визуализации групп связанных между собой предметов, людей или идей. Вот какие темы мы будем обсуждать: как подготовить данные из различных источников; два разных алгоритма кластеризации; дополнительная информация о метриках подобия; простые способы графической визуализации обнаруженных групп; метод проецирования очень сложных наборов данных на двумерную плоскость.

Кластеризация часто используется в приложениях, обрабатывающих большие объемы данных. Розничные торговцы, которые отслеживают историю покупок, могут воспользоваться этой информацией для обнаружения групп клиентов со схожим потребительским поведением (помимо обычной демографической информации). Люди одного возраста, имеющие примерно одинаковый уровень доходов, могут одеваться совершенно по-разному, но с помощью кластеризации удается выявить «островки моды» и использовать эти данные для выработки стратегии маркетинга или розничных продаж. Кластеризация также активно применяется в вычислительной биологии для обнаружения групп генов, демонстрирующих сходное поведение. Это может служить указанием на то, что они будут одинаково реагировать на лечение или что образовались на одном пути биологического развития.

Поскольку эта книга посвящена коллективному разуму, примеры взяты из тех областей, где многие люди привносят различную информацию. В первом примере мы рассмотрим блоги, обсуждаемые в них темы и паттерны употребления слов. Мы покажем, что блоги можно

сгруппировать в соответствии с текстом, а слова – по способу употребления. Второй пример относится к социальному сайту, где участники перечисляют те вещи, которые у них есть, и те, которые они хотели бы приобрести. Мы увидим, что с помощью этой информации желания людей можно сгруппировать в кластеры.

Обучение с учителем и без него

Если выработке прогноза предшествует стадия обучения, когда машине предъявляются примеры входной и выходной информации, то говорят об *обучении с учителем*. В этой книге мы рассмотрим разнообразные методы обучения с учителем, в том числе нейронные сети, деревья решений, метод опорных векторов и байесовскую фильтрацию. Приложение «обучается», анализируя входные данные и соответствующие им результаты. Желая извлечь новую информацию с помощью любого из таких методов, мы подаем на вход некоторые данные и ожидаем, что приложение выдаст результат, основываясь на том, чему научилось раньше.

Кластеризация – пример *обучения без учителя*. В отличие от нейронных сетей или деревьев решений, алгоритмам обучения без учителя не сообщаются правильные ответы. Их задача – обнаружить структуру в наборе данных, когда ни один элемент данных не является ответом. В примере с модной одеждой кластер ничего не говорит продавцу о том, что может купить конкретный индивидуум, и не дает прогнозов о том, в какой островок моды попадет новый клиент. Назначение алгоритмов кластеризации состоит в том, чтобы обнаружить отдельные группы в предъявленном наборе данных. Примерами обучения без учителя служат алгоритм *неотрицательной матричной факторизации*, который мы рассмотрим в главе 10, и *самоорганизующиеся карты*.

Векторы слов

Обычно при подготовке данных для кластеризации определяют общий набор числовых атрибутов, с помощью которых элементы можно сравнивать. Это очень напоминает то, что мы делали в главе 2, когда сравнивали оценки критиков на общем множестве фильмов или сопоставляли число 1/0 наличию/отсутствию закладки у пользователя сайта *del.icio.us*.

Систематизация блогеров

В этой главе мы рассмотрим два примера набора данных. В первом случае кластеризуемыми элементами будут 120 наиболее посещаемых блогов, а данные, по которым осуществляется кластеризация, – это количество вхождений определенных слов в каждую запись в блоге. Небольшое иллюстративное подмножество приведено в табл. 3.1.

Таблица 3.1. Подмножество частот вхождения слов в блог

	china	kids	music	yahoo
Gothamist	0	3	3	0
GigaOM	6	0	0	2
Quick Online Tips	0	2	2	22

Путем кластеризации блогов по частоте слов, возможно, удастся определить, существуют ли группы блогов, в которых часто пишут на одни и те же темы или в похожем стиле. Такой результат был бы очень полезен для поиска, каталогизации и классификации гигантского числа блогов, расплодившихся в Сети.

Для генерирования этого набора данных необходимо скачать записи из ряда блогов, выделить из них текст и создать таблицу частот слов. Если вам не хочется делать это самостоятельно, то можете загрузить готовый набор со страницы *http://kiwitobes.com/clusters/blogdata.txt*.

Подсчет количества слов в RSS-канале

Почти все блоги можно читать напрямую или с помощью *RSS-каналов*. RSS-канал – это простой XML-документ, содержащий информацию о блоге и всех записях в нем. Первый шаг процедуры получения счетчиков слов – проанализировать эти каналы. На наше счастье, существует библиотека Universal Feed Parser, которая прекрасно справляется с этой задачей. Ее можно скачать с сайта *http://www.feedparser.org*.

Этот модуль позволяет легко получить заголовок, ссылки и записи из любого канала в формате RSS или Atom. Следующий шаг – написать функцию, которая будет извлекать отдельные слова. Создайте новый файл generatefeedvector.py и включите в него такой код:

```
import feedparser
import re

# Возвращает заголовок и словарь слов со счетчиками для RSS-канала
def getwordcounts(url):
  # Проанализировать канал
  d=feedparser.parse(url)
  wc={}

  # Цикл по всем записям
  for e in d.entries:
    if 'summary' in e: summary=e.summary
    else: summary=e.description

    # Сформировать список слов
    words=getwords(e.title+' '+summary)
    for word in words:
      wc.setdefault(word,0)
      wc[word]+=1
  return d.feed.title,wc
```

В каналах формата RSS и Atom всегда имеется заголовок и список записей. В каждой записи обычно есть тег `summary` или `description`, внутри которого находится собственно текст записи. Функция `getwordcounts` передает содержимое этого тега функции `getwords`, которая отфильтровывает HTML-разметку и выделяет слова, считая словом последовательность символов, ограниченную с двух сторон небуквенными символами. На выходе получается список слов. Добавьте функцию `getwords` в файл generatefeedvector.py:

```
def getwords(html):
  # Удалить все HTML-теги
  txt=re.compile(r'<[^>]+>').sub('',html)

  # Выделить слова, ограниченные небуквенными символами
  words=re.compile(r'[^A-Z^a-z]+').split(txt)

  # Преобразовать в нижний регистр
  return [word.lower( ) for word in words if word!='']
```

Теперь нам нужен список каналов, с которыми можно работать. Если хотите, можете составить список URL каналов самостоятельно, а можете взять готовый список, содержащий 100 URL. В этот список были включены каналы самых цитируемых блогов, а затем удалены те, что не содержали полный текст записи или содержали в основном изображения. Скачать список можно со страницы *http://kiwitobes.com/clusters/feedlist.txt*.

Это обычный текст, содержащий по одному URL в каждой строке. Если у вас есть свой блог или какие-нибудь особо любимые и вы хотите узнать, как они соотносятся с наиболее популярными блогами в Сети, можете добавить URL в файл.

Код для обхода каналов и генерирования набора данных – это главный код в файле generatefeedvector.py (то есть он не является функцией). Сначала мы в цикле перебираем все строки файла feedlist.txt и генерируем счетчики слов в каждом блоге, а также количество блогов, в которых встречается каждое слово (`apcount`). Добавьте следующий код в конец файла generatefeedvector.py:

```
apcount={}
wordcounts={}
for feedurl in file('feedlist.txt'):
  title,wc=getwordcounts(feedurl)
  wordcounts[title]=wc
  for word,count in wc.items( ):
    apcount.setdefault(word,0)
    if count>1:
       apcount[word]+=1
```

Далее генерируется список слов, которые учтены в счетчиках для каждого блога. Поскольку слова типа the встречаются практически в каждом блоге, а такие слова, как flim-flam – разве что в одном, то можно уменьшить общее количество слов, оставляя только те, для которых

процент вхождений лежит между нижним и верхним порогом. Например, для начала можно в качестве нижнего порога выбрать 10%, а в качестве верхнего – 50%, а потом варьировать их, если выяснится, что количество общих слов или странных словообразований слишком велико.

```
wordlist=[]
for w,bc in apcount.items( ):
  frac=float(bc)/len(feedlist)
  if frac>0.1 and frac<0.5: wordlist.append(w)
```

Последний шаг состоит в том, чтобы на основе списка слов и списка блогов создать текстовый файл, содержащий большую матрицу счетчиков слов для каждого блога:

```
out=file('blogdata.txt','w')
out.write('Blog')
for word in wordlist: out.write('\t%s' % word)
out.write('\n')
for blog,wc in wordcounts.items( ):
  out.write(blog)
  for word in wordlist:
    if word in wc: out.write('\t%d' % wc[word])
    else: out.write('\t0')
  out.write('\n')
```

Для генерирования файла со счетчиками слов запустите программу generatefeedvector.py из командной строки:

```
c:\code\blogcluster>python generatefeedvector.py
```

На загрузку всех каналов может уйти несколько минут, но в конечном итоге вы получите файл blogdata.txt. Откройте его и убедитесь, что он содержит таблицу, в которой столбцы соответствуют словам, а строки – блогам, причем элементы разделены табуляторами. Такой формат предполагается во всех функциях из этой главы, поэтому в дальнейшем вы можете применить описанные алгоритмы кластеризации к другому набору данных, созданному вручную или сохраненному в нужном формате из электронной таблицы.

Иерархическая кластеризация

Алгоритм иерархической кластеризации строит иерархию групп, объединяя на каждом шаге две самые похожие группы. В начале каждая группа состоит из одного элемента, в данном случае – одного блога. На каждой итерации вычисляются попарные расстояния между группами, и группы, оказавшиеся самыми близкими, объединяются в новую группу. Так повторяется до тех пор, пока не останется всего одна группа. Эта процедура изображена на рис. 3.1.

Здесь схожесть элементов представлена их относительным расположением – чем элементы ближе друг к другу, тем более они схожи. В начале каждый элемент – это отдельная группа. На втором шаге два самых

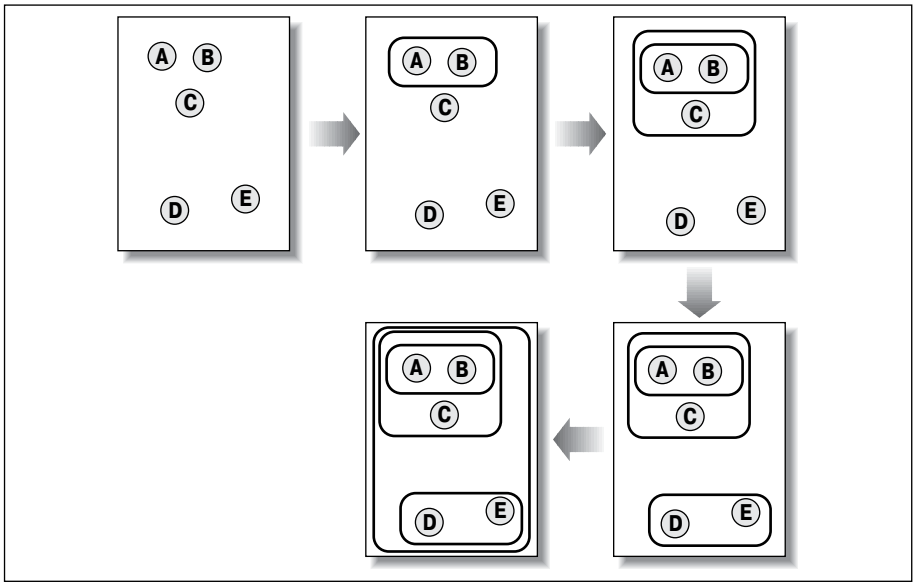


Рис. 3.1. *Иерархическая кластеризация в действии*

близких элемента A и B объединены в новую группу, расположенную посередине между исходными. На третьем шаге эта новая группа объединяется с элементом C. Поскольку теперь два ближайших элемента – это D и E, то из них образуется новая группа. И на последнем шаге две оставшиеся группы объединяются в одну.

Результаты иерархической кластеризации обычно представляются в виде графа, который называется *дендрограммой*. На нем изображено, как из узлов формировалась иерархия. Дендрограмма для рассмотренного выше примера показана на рис. 3.2.

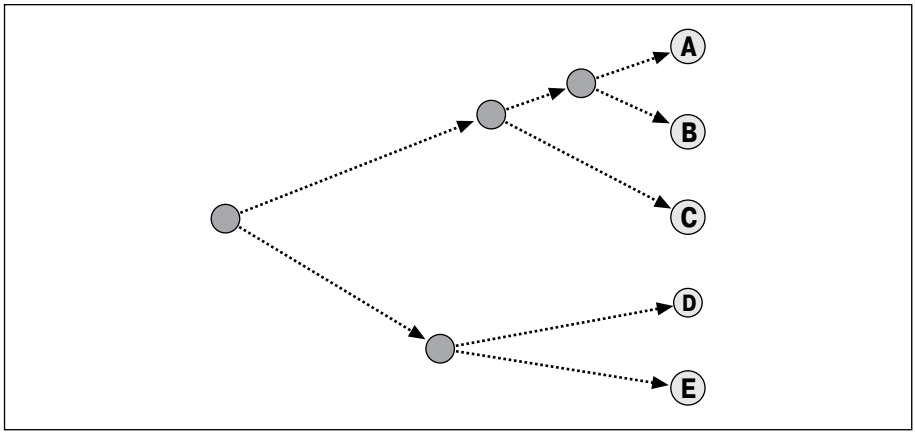


Рис. 3.2. *Дендрограмма как способ визуализации иерархической кластеризации*

На дендрограмме представлены не только ребра графа, показывающие, из каких элементов составлен каждый кластер, но и расстояния, говорящие о том, как далеко эти элементы отстояли друг от друга. Кластер AB гораздо ближе к составляющим его элементам A и B, чем кластер DE к элементам D и E. Изображение графа таким образом помогает понять, насколько схожи элементы, вошедшие в кластер. Эта характеристика называется *теснотой* (tightness) кластера.

В этом разделе мы покажем, как с помощью кластеризации построить иерархию блогов, которая в случае успеха сгруппирует их по тематике. Прежде всего нам понадобится метод для загрузки файла данных. Создайте файл clusters.py и включите в него такую функцию:

```
def readfile(filename):
  lines=[line for line in file(filename)]

  # Первая строка содержит названия столбцов
  colnames=lines[0].strip( ).split('\t')[1:]
  rownames=[]
  data=[]
  for line in lines[1:]:
    p=line.strip( ).split('\t')
    # Первый столбец в каждой строке содержит название строки
    rownames.append(p[0])
    # Остальные ячейки содержат данные этой строки
    data.append([float(x) for x in p[1:]])
  return rownames,colnames,data
```

Эта функция считывает первую строку и формирует из нее список названий столбцов, затем считывает самый левый столбец, формируя названия строк, а после этого помещает данные из остальных ячеек в один большой список, где каждый элемент соответствует одной ячейке текущей строки. Счетчик, находившийся в ячейке, можно найти по индексу строки и столбца в массиве `data`, а индексы получить по именам из списков `rownames` и `colnames`.

Следующий шаг – определить, что такое *близость*. В главе 2 мы рассматривали евклидово расстояние и коэффициент корреляции Пирсона как меры схожести двух кинокритиков. В данном примере некоторые блоги содержат гораздо больше записей, чем другие, или сами записи в них более длинные. Поэтому количество слов в разных блогах может сильно различаться. Коэффициент Пирсона способен скорректировать это расхождение, так как пытается лишь определить, насколько хорошо два набора данных ложатся на прямую. Функция вычисления коэффициента Пирсона для этого модуля принимает на входе два списка чисел и возвращает коэффициент корреляции между ними:

```
from math import sqrt
def pearson(v1,v2):
  # Простые суммы
```

```
    sum1=sum(v1)
    sum2=sum(v2)

    # Суммы квадратов
    sum1Sq=sum([pow(v,2) for v in v1])
    sum2Sq=sum([pow(v,2) for v in v2])

    # Суммы произведений
    pSum=sum([v1[i]*v2[i] for i in range(len(v1))])

    # Вычисляем r (коэффициент Пирсона)
    num=pSum-(sum1*sum2/len(v1))
    den=sqrt((sum1Sq-pow(sum1,2)/len(v1))*(sum2Sq-pow(sum2,2)/len(v1)))
    if den==0: return 0

    return 1.0-num/den
```

Напомним, что коэффициент Пирсона равен 1,0, если два набора данных в точности совпадают, и близок к 0,0, если никакой связи между ними не наблюдается. В последней строке мы возвращаем 1,0 минус коэффициент корреляции, чтобы расстояние между наборами было тем меньше, чем они более схожи.

Каждый кластер в алгоритме иерархической кластеризации – это либо узел в дереве с двумя ветвями, либо конечный (листовый) узел, ассоциированный с конкретной строкой из набора данных (в данном случае с блогом). Каждый кластер содержит также данные о своем положении; это либо строка данных для листовых узлов, либо объединенные данные двух ветвей для остальных узлов. Можно создать класс `bicluster` для представления иерархического дерева, наделив его всеми этими свойствами. Включите класс, описывающий тип кластера, в файл clusters.py:

```
class bicluster:
  def __init__(self,vec,left=None,right=None,distance=0.0,id=None):
    self.left=left
    self.right=right
    self.vec=vec
    self.id=id
    self.distance=distance
```

Алгоритм иерархической кластеризации начинает работу с создания группы кластеров, каждый из которых содержит ровно один исходный элемент. В главном цикле функция ищет два самых похожих элемента, вычисляя коэффициент корреляции между каждой парой кластеров. Наилучшая пара объединяется в новый кластер. Данными для нового кластера служит среднее арифметическое данных двух старых кластеров. Этот процесс повторяется, пока не останется только один кластер. На выполнение всех вычислений может уйти очень много времени, поэтому было бы разумно сохранять коэффициенты корреляции для каждой пары, поскольку они вычисляются снова и снова, пока один из элементов пары не попадет в новый кластер.

Добавьте функцию `hcluster` в файл clusters.py:

```
def hcluster(rows,distance=pearson):
  distances={}
  currentclustid=-1

  # В начале кластеры совпадают со строками
  clust=[bicluster(rows[i],id=i) for i in range(len(rows))]

  while len(clust)>1:
    lowestpair=(0,1)
    closest=distance(clust[0].vec,clust[1].vec)

    # в цикле рассматриваются все пары и ищется пара с минимальным
    # расстоянием
    for i in range(len(clust)):
      for j in range(i+1,len(clust)):
        # вычисленные расстояния запоминаются в кэше
        if (clust[i].id,clust[j].id) not in distances:
          distances[(clust[i].id,clust[j].id)]=
          distance(clust[i].vec,clust[j].vec)

        d=distances[(clust[i].id,clust[j].id)]

       if d<closest:
         closest=d
         lowestpair=(i,j)

    # вычислить среднее для двух кластеров
    mergevec=[
    (clust[lowestpair[0]].vec[i]+clust[lowestpair[1]].vec[i])/2.0
    for i in range(len(clust[0].vec))]

    # создать новый кластер
    newcluster=bicluster(mergevec,left=clust[lowestpair[0]],
                         right=clust[lowestpair[1]],
                         distance=closest,id=currentclustid)

    # идентификаторы кластеров, которых не было в исходном наборе,
    # отрицательны
    currentclustid-=1
    del clust[lowestpair[1]]
    del clust[lowestpair[0]]
    clust.append(newcluster)

  return clust[0]
```

Поскольку в каждом кластере хранятся ссылки на оба кластера, объединением которых он получился, то финальный кластер, возвращенный этой функцией, можно рекурсивно обойти, показав все промежуточные кластеры и листовые узлы.

Чтобы выполнить иерархическую кластеризацию, запустите Python в интерактивном режиме, загрузите файл и вызовите функцию `hcluster`:

```
$ python
>> import clusters
>> blognames,words,data=clusters.readfile('blogdata.txt')
>> clust=clusters.hcluster(data)
```

На это может уйти несколько минут. Кэширование расстояний заметно увеличивает скорость работы, но все равно алгоритм должен вычислить корреляцию между каждой парой блогов. Процедуру можно ускорить, воспользовавшись внешней библиотекой для вычисления расстояний. Для просмотра результатов можно написать простую функцию, которая рекурсивно обходит дерево кластеризации и распечатывает его так же, как дерево файловой системы. Добавьте в файл clusters.py следующую функцию:

```
def printclust(clust,labels=None,n=0):
  # отступ для визуализации иерархии
  for i in range(n): print ' ',
    if clust.id<0:
      # отрицательный id означает, что это внутренний узел
      print '-'
    else:
      # положительный id означает, что это листовый узел
      if labels==None: print clust.id
      else: print labels[clust.id]

    # теперь печатаем правую и левую ветви
    if clust.left!=None: printclust(clust.left,labels=labels,n=n+1)
    if clust.right!=None: printclust(clust.right,labels=labels,n=n+1)
```

Получившийся результат выглядит не слишком красиво, и при таком большом наборе данных, как список блогов, неудобен для восприятия. Тем не менее общее представление о том, как работает алгоритм кластеризации, он дает. В следующем разделе мы поговорим о создании графической версии, которая воспринимается гораздо легче и рисуется так, чтобы была видна протяженность каждого кластера.

В сеансе работы с интерпретатором вызовите эту функцию для только что построенных кластеров:

```
>> reload(clusters)
>> clusters.printclust(clust,labels=blognames)
```

Распечатка содержит все 100 блогов и потому довольно длинная. Вот пример кластера, обнаруженного при обработке этого набора данных:

```
John Battelle's Searchblog
-
  Search Engine Watch Blog
  -
    Read/WriteWeb
    -
```

```
Official Google Blog
-
  Search Engine Roundtable
  -
    Google Operating System
    Google Blogoscoped
```

Показаны элементы исходного набора. Знак минус обозначает кластер, получившийся объединением двух или более элементов. Здесь вы видите прекрасный пример обнаружения группы; интересно также, что в самых популярных каналах так много блогов, имеющих отношение к поиску. Изучая распечатку, вы найдете также кластеры блогов о политике, о технологии и о самих блогах.

Вероятно, вы заметите и некоторые аномалии. Возможно, авторы и не писали на одни и те же темы, но алгоритм кластеризации говорит, что частоты слов коррелированы. Это может быть следствием общего стиля изложения или простым совпадением, характерным только для того дня, когда были загружены данные.

Рисование дендрограммы

Интерпретацию кластеров можно облегчить, если изобразить их в виде дендрограммы. Результаты иерархической кластеризации обычно так и представляются, поскольку дендрограмма позволяет уместить большой объем информации в сравнительно малом пространстве. Поскольку дендрограммы – графические изображения и сохраняются в формате JPG, то нам понадобится библиотека Python Imaging Library (PIL), которую можно скачать с сайта *http://pythonware.com*.

Библиотека поставляется в виде инсталлятора для Windows и набора исходных файлов для других платформ. Дополнительная информация о загрузке и установке PIL имеется в приложении A. Библиотека PIL позволяет легко генерировать изображения, состоящие из текста и линий, а для построения дендрограммы больше ничего и не нужно. Добавьте в начало файла clusters.py предложение `import`:

```
from PIL import Image,ImageDraw
```

Прежде всего мы воспользуемся функцией, которая возвращает полную высоту кластера. Чтобы определить высоту изображения и понять, где расставлять узлы, необходимо знать высоту каждого кластера. Если кластер представлен листовым узлом (то есть от него не отходят ветви), то его высота равна 1. В противном случае высота кластера равна сумме высот его потомков. Высоту легко вычислить с помощью рекурсивной функции, которую мы добавим в файл clusters.py:

```
def getheight(clust):
  # Это листовый узел? Тогда высота равна 1
  if clust.left==None and clust.right==None: return 1

  # Иначе высота равна сумме высот обеих ветвей
  return getheight(clust.left)+getheight(clust.right)
```

Еще нам нужно знать полную ошибку в корневом узле. Поскольку длина линий зависит от величины ошибки в каждом узле, то коэффициент масштабирования будет вычисляться в зависимости от полной ошибки. Ошибка в узле – это просто максимум из значений ошибки обоих его потомков:

```
def getdepth(clust):
  # Расстояние для листового узла равно 0,0
  if clust.left==None and clust.right==None: return 0
  # Расстояние для внутреннего узла равно максимуму из расстояний обеих ветвей
  # плюс его собственное расстояние
  return max(getdepth(clust.left),getdepth(clust.right))+clust.distance
```

Функция `drawdendrogram` создает новое изображение высотой 20 пикселей и фиксированной шириной для каждого кластера. Коэффициент масштабирования определяется путем деления фиксированной ширины на полную глубину. Функция создает объект рисования этого изображения и затем вызывает функцию `drawnode` для корневого узла, сообщая, что его следует расположить на полпути вниз вдоль левого края изображения.

```
def drawdendrogram(clust,labels,jpeg='clusters.jpg'):
  # высота и ширина
  h=getheight(clust)*20
  w=1200
  depth=getdepth(clust)

  # ширина фиксирована, расстояния масштабируются
  scaling=float(w-150)/depth

  # создать новое изображение на белом фоне
  img=Image.new('RGB',(w,h),(255,255,255))
  draw=ImageDraw.Draw(img)

  draw.line((0,h/2,10,h/2),fill=(255,0,0))

  # нарисовать первый узел
  drawnode(draw,clust,10,(h/2),scaling,labels)
  img.save(jpeg,'JPEG')
```

Самой важной является функция `drawnode`, которая принимает кластер и его положение. Она получает высоты дочерних узлов, вычисляет, в каком месте они должны располагаться, и проводит ведущие к ним линии – одну длинную вертикальную и две горизонтальные. Длины горизонтальных линий определяются накопленной в этом кластере ошибкой. Если линия длинная, значит, кластеры, объединением которых был получен данный кластер, были не очень-то похожи. А короткая линия означает, что они были почти идентичны. Добавьте функцию `drawnode` в файл clusters.py:

```
def drawnode(draw,clust,x,y,scaling,labels):
  if clust.id<0:
    h1=getheight(clust.left)*20
    h2=getheight(clust.right)*20
```

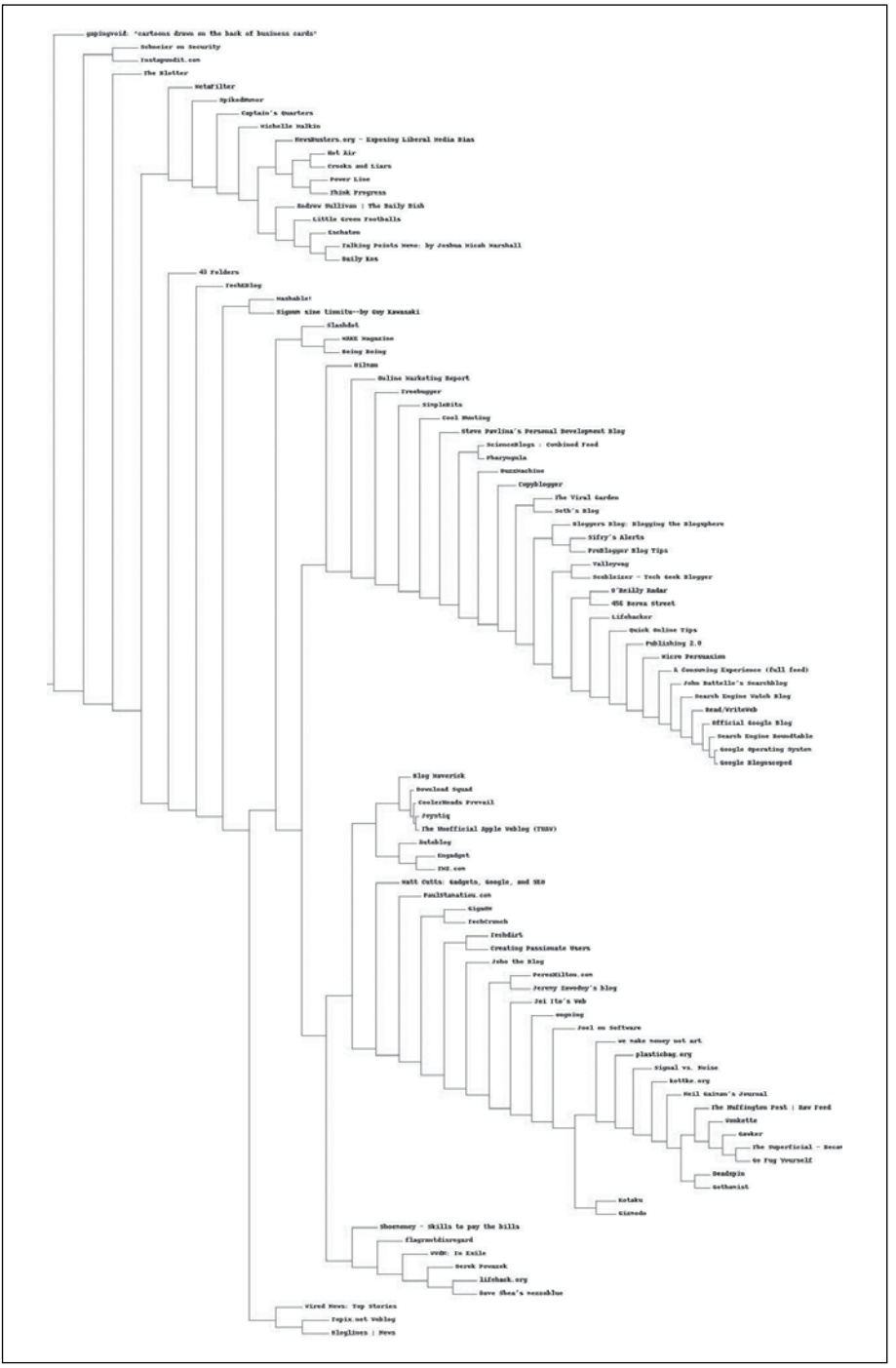


Рис. 3.3. *Дендрограмма, иллюстрирующая кластеры блогов*

```
    top=y-(h1+h2)/2
    bottom=y+(h1+h2)/2
    # Длина линии
    ll=clust.distance*scaling
    # Вертикальная линия от данного кластера к его потомкам
    draw.line((x,top+h1/2,x,bottom-h2/2),fill=(255,0,0))

    # Горизонтальная линия к левому потомку
    draw.line((x,top+h1/2,x+ll,top+h1/2),fill=(255,0,0))

    # Горизонтальная линия к правому потомку
    draw.line((x,bottom-h2/2,x+ll,bottom-h2/2),fill=(255,0,0))

    # Вызываем функцию для изображения левого и правого потомков
    drawnode(draw,clust.left,x+ll,top+h1/2,scaling,labels)
    drawnode(draw,clust.right,x+ll,bottom-h2/2,scaling,labels)
  else:
    # Если это листовый узел, рисуем метку
    draw.text((x+5,y-7),labels[clust.id],(0,0,0))
```

Чтобы сгенерировать изображение, введите в интерактивном сеансе следующие команды:

```
>> reload(clusters)
>> clusters.drawdendrogram(clust,blognames,jpeg='blogclust.jpg')
```

В результате будет создан файл blogclust.jpg, содержащий дендрограмму, которая должна быть похожа на ту, что изображена на рис. 3.3. Если хотите, можете изменить настройки высоты и ширины, чтобы рисунок было проще напечатать или чтобы он был более разреженным.

Кластеризация столбцов

Часто бывает необходимо выполнить кластеризацию одновременно по строкам и столбцам. В маркетинговых исследованиях интересно сгруппировать людей с целью выявления общих демографических признаков или предпочитаемых товаров, а быть может, для того чтобы выяснить, на каких полках размещены товары, которые обычно покупают вместе. В наборе данных о блогах столбцы представляют слова, и можно поинтересоваться, какие слова часто употребляют вместе.

Простейший способ решить эту задачу с помощью уже написанных функций – повернуть весь набор данных, так чтобы столбцы (слова) стали строками. Тогда списки чисел в каждой строке покажут, сколько раз данное слово встречалось в каждом блоге.

Добавьте в файл clusters.py следующую функцию:

```
def rotatematrix(data):
  newdata=[]
  for i in range(len(data[0])):
    newrow=[data[j][i] for j in range(len(data))]
    newdata.append(newrow)
  return newdata
```

Теперь можно повернуть матрицу и выполнить те же операции кластеризации и рисования дендрограммы, что и выше. Поскольку слов гораздо больше, чем блогов, то и времени потребуется больше, чем на кластеризацию блогов. Напомним, что коль скоро матрица повернута, метками теперь будут слова, а не названия блогов.

```
>> reload(clusters)
>> rdata=clusters.rotatematrix(data)
>> wordclust=clusters.hcluster(rdata)
>> clusters.drawdendrogram(wordclust,labels=words,jpeg='wordclust.jpg')
```

Важно понимать, что если элементов гораздо больше, чем переменных, то возрастает вероятность появления бессмысленных кластеров. Поскольку слов значительно больше, чем блогов, то кластеризация блогов позволяет выявить больше осмысленных паттернов, чем кластеризация слов. Однако, как видно из рис. 3.4, некоторые интересные кластеры все же имеются.

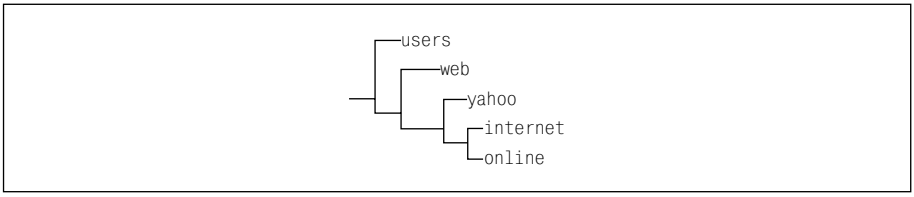


***Рис. 3.4.** Кластеры слов, связанных с онлайновыми службами*

Этот кластер наглядно показывает, какие слова часто употребляются в блогах при обсуждении тем, касающихся онлайновых служб и Интернета. Можно найти и другие кластеры, отражающие характерное употребление слов. Например, кластер из слов fact, us, say, very и think выдает, что автор блога – человек самоуверенный.

Кластеризация методом K-средних

Иерархическая кластеризация дает на выходе симпатичное дерево, но у этого метода есть два недостатка. Древовидное представление само по себе не разбивает данные на группы, для этого нужна дополнительная работа. Кроме того, алгоритм требует очень большого объема вычислений. Поскольку необходимо вычислять попарные соотношения, а затем вычислять их заново после объединения элементов, то на больших наборах данных алгоритм работает медленно.

Альтернативный способ называется *кластеризацией методом K-средних*. Он в корне отличается от иерархической кластеризации, поскольку ему нужно заранее указать, сколько кластеров генерировать. Алгоритм определяет размеры кластеров исходя из структуры данных.

Кластеризация методом *K*-средних начинается с выбора k случайно расположенных центроидов (точек, представляющих центр кластера).

Каждому элементу назначается ближайший центроид. После того как назначение выполнено, каждый центроид перемещается в точку, рассчитываемую как среднее по всем приписанным к нему элементам. Затем назначение выполняется снова. Эта процедура повторяется до тех пор, пока назначения не прекратят изменяться. На рис. 3.5 показано, как развивается процесс для пяти элементов и двух кластеров.

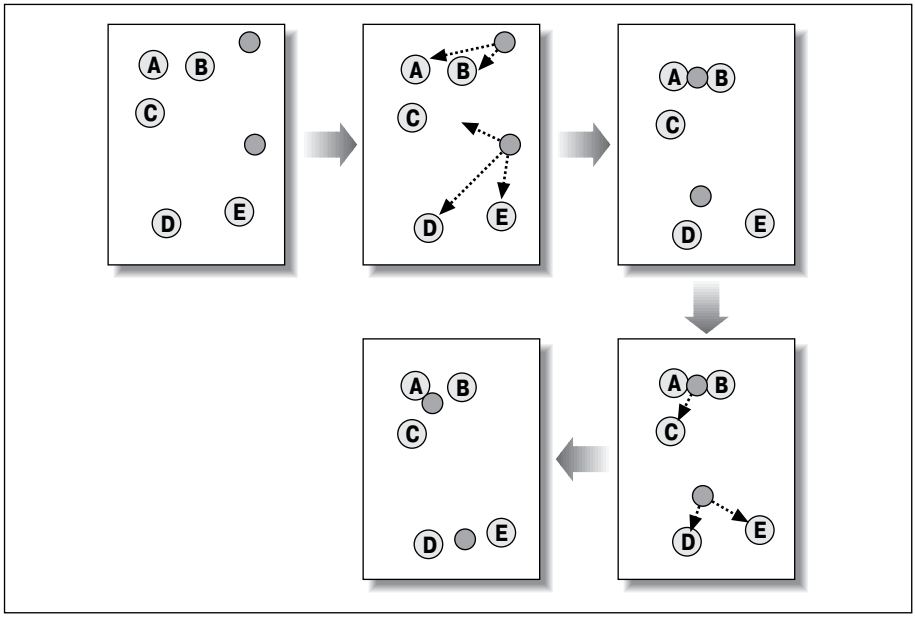


Рис. 3.5. *Кластеризация методом K-средних с двумя кластерами*

На первом шаге случайно размещены два центроида (изображены темными кружками). На втором шаге каждый элемент приписан к ближайшему центроиду. В данном случае A и B приписаны к верхнему центроиду, а C, D и E – к нижнему. На третьем шаге каждый центроид перемещен в точку, рассчитанную как среднее по всем приписанным к нему элементам. После пересчета назначений оказалось, что C теперь ближе к верхнему центроиду, а D и E – по-прежнему ближе к нижнему. Окончательный результат достигается, когда A, B и C помещены в один кластер, а D и E – в другой.

Функция для выполнения кластеризации методом *K*-средних принимает на входе те же строки данных, что и алгоритм иерархической кластеризации, а кроме того, количество кластеров (*k*), которое хотела бы получить вызывающая программа. Добавьте в файл clusters.py такой код:

```
import random

def kcluster(rows,distance=pearson,k=4):
  # Определить минимальное и максимальное значения для каждой точки
```

```
ranges=[(min([row[i] for row in rows]),max([row[i] for row in rows]))
for i in range(len(rows[0]))]

# Создать k случайно расположенных центроидов
clusters=[[random.random( )*(ranges[i][1]-ranges[i][0])+ranges[i][0]
for i in range(len(rows[0]))] for j in range(k)]

lastmatches=None
for t in range(100):
  print 'Итерация %d' % t
  bestmatches=[[] for i in range(k)]

  # Найти для каждой строки ближайший центроид
  for j in range(len(rows)):
    row=rows[j]
    bestmatch=0
    for i in range(k):
      d=distance(clusters[i],row)
      if d<distance(clusters[bestmatch],row): bestmatch=i
    bestmatches[bestmatch].append(j)

  # Если получился такой же результат, как в прошлый раз, заканчиваем
  if bestmatches==lastmatches: break
  lastmatches=bestmatches

  # Перемещаем каждый центроид в центр приписанных к нему элементов
  for i in range(k):
    avgs=[0.0]*len(rows[0])
    if len(bestmatches[i])>0:
      for rowid in bestmatches[i]:
        for m in range(len(rows[rowid])):
          avgs[m]+=rows[rowid][m]
      for j in range(len(avgs)):
         avgs[j]/=len(bestmatches[i])
      clusters[i]=avgs

return bestmatches
```

В этой функции создается случайный набор кластеров в области определения переменных. На каждой итерации каждая строка приписывается одному из центроидов, после чего центроиды перемещаются в центр приписанных к ним элементов. Как только оказывается, что на очередной итерации все строки приписаны тем же центроидам, что и на предыдущей, процесс завершается и функция возвращает k списков, представляющих кластеры. По сравнению с иерархической кластеризацией, количество итераций, необходимых для достижения результата, гораздо меньше.

Поскольку начальные положения центроидов выбираются случайным образом, порядок возвращенных результатов может быть различен. Может оказаться, что и состав кластеров зависит от начальных положений центроидов.

Можете протестировать эту функцию на наборе данных о блогах. Она должна работать заметно быстрее, чем в случае иерархической кластеризации:

```
>> reload(clusters)
>> kclust=clusters.kcluster(data,k=10)
Итерация 0
...
>> [rownames[r] for r in k[0]]
['The Viral Garden', 'Copyblogger', 'Creating Passionate Users', 'Oilman',
 'ProBlogger Blog Tips', "Seth's Blog"]
>> [rownames[r] for r in k[1]]
и т. д...
```

Теперь `kclust` содержит список идентификаторов, вошедших в каждый кластер. Попробуйте выполнить кластеризацию с другим значением *k* и посмотрите, как это отразится на результатах.

Кластеры предпочтений

Одним из факторов растущего интереса к социальным сетям является тот факт, что стали доступны большие наборы данных, добровольно пополняемые различными людьми. Один из таких сайтов – Zebo (*http://www.zebo.com*) – предлагает создать учетную запись и разместить список тех предметов, которые у пользователя есть, и тех, которые он хотел бы приобрести. С точки зрения рекламодателя или социолога, это очень интересная информация, поскольку позволяет выяснить, как естественно группируются выраженные предпочтения.

Получение и подготовка данных

В этом разделе мы рассмотрим процедуру создания набора данных с сайта Zebo. Нам придется скачать с него много страниц и проанализировать их, чтобы извлечь информацию о пожеланиях пользователя. Если вы хотите пропустить этот раздел, то можете загрузить уже готовый набор данных со страницы *http://kiwitobes.com/clusters/zebo.txt*.

Библиотека Beautiful Soup

Beautiful Soup – это великолепная библиотека для разбора и построения структурированного представления веб-страницы. Она позволяет найти элемент страницы по типу, идентификатору или любому свойству и получить строковое представление его содержимого. Beautiful Soup очень терпимо относится к страницам с некорректной HTML-разметкой, что весьма полезно при генерации наборов данных с вебсайтов.

Скачать библиотеку Beautiful Soup можно с сайта *http://crummy.com/software/BeautifulSoup*. Она поставляется в виде одного файла на языке Python, который следует поместить в то место, где интерпретатор

сможет его найти. Или можете поместить ее в свою рабочую папку и запускать интерпретатор, находясь в этой папке.

Установив библиотеку Beautiful Soup, проверьте ее в действии, запустив интерпретатор в интерактивном режиме:

```
>> import urllib2
>> from BeautifulSoup import BeautifulSoup
>> c=urllib2.urlopen('http://kiwitobes.com/wiki/Programming_language.html')
>> soup=BeautifulSoup(c.read( ))
>> links=soup('a')
>> links[10]
<a href="/wiki/Algorithm.html" title="Algorithm">algorithms</a>
>> links[10]['href']
u'/wiki/Algorithm.html'
```

Чтобы «сварить суп» (так в Beautiful Soup называется представление веб-страницы), достаточно передать конструктору содержимое страницы. Затем можно вызвать «суп», передав ему тип тега (в примере выше – `a`) и получить в ответ список объектов этого типа. Каждый объект, в свою очередь, адресуем, то есть можно запросить его свойства и список объектов, находящихся ниже него в иерархии.

Разбор страниц сайта Zebo

Структура страницы результатов поиска на сайте Zebo довольно сложна, но определить, какие части страницы относятся к списку элементов, просто, так как для них задан класс `bgverdanasmall`. Воспользовавшись этим, мы можем извлечь из страницы интересующие нас данные. Создайте файл downloadzebodata.py и включите в него такой код:

```
from BeautifulSoup import BeautifulSoup
import urllib2
import re
chare=re.compile(r'[!-\.&]')
itemowners={}

# Эти слова следует игнорировать
dropwords=['a','new','some','more','my','own','the','many','other','another'
]

currentuser=0
for i in range(1,51):
  # URL страницы результатов поиска
  c=urllib2.urlopen(
  'http://member.zebo.com/Main?event_key=USERSEARCH&wiowiw=wiw&keyword=car&
page=%d'
  % (i))
  soup=BeautifulSoup(c.read( ))
  for td in soup('td'):
    # Найти ячейки таблицы с классом bgverdanasmall
    if ('class' in dict(td.attrs) and td['class']=='bgverdanasmall'):
      items=[re.sub(chare,'',a.contents[0].lower()).strip( ) for a in td('a')]
```

```
    for item in items:
      # Удалить игнорируемые слова
      txt=' '.join([t for t in item.split(' ') if t not in dropwords])
      if len(txt)<2: continue
      itemowners.setdefault(txt,{})
      itemowners[txt][currentuser]=1
    currentuser+=1
```

Эта программа загружает и разбирает первые 50 страниц «пожеланий» с сайта Zebo. Поскольку данные вводятся в виде неструктурированного текста, приходится приложить усилия для их очистки, в частности удалить такие слова, как артикль a и some, избавиться от знаков препинания и перевести все в нижний регистр.

Проделав все это, программа сначала создает список предметов, которые желают иметь более пяти человек, затем строит матрицу, столбцы которой представляют анонимных пользователей, а строки – предметы, и наконец выводит эту матрицу в файл. Добавьте следующий код в файл downloadzebodata.py:

```
out=file('zebo.txt','w')
out.write('Item')
for user in range(0,currentuser): out.write('\tU%d' % user)
out.write('\n')
for item,owners in itemowners.items( ):
  if len(owners)>10:
    out.write(item)
    for user in range(0,currentuser):
      if user in owners: out.write('\t1')
      else: out.write('\t0')
    out.write('\n')
```

Выполните следующую команду, которая создаст файл zebo.txt в том же формате, что и для набора данных о блогах. Единственное отличие состоит в том, что вместо счетчиков мы используем 1, если человек хочет иметь некий предмет, и 0 – если не хочет:

```
c:\code\cluster>python downloadzebodata.py
```

Определение метрики близости

Для набора данных о блогах, где значениями являются счетчики слов, коэффициент корреляции Пирсона работает неплохо. Но в данном случае у нас есть лишь единицы и нули, представляющие соответственно наличие и отсутствие, и было бы полезнее определить некую меру перекрытия между людьми, желающими иметь два предмета. Такая мера существует и называется *коэффициентом Танимото*; это отношение мощности пересечения множеств (элементов, принадлежащих обоим множествам) к мощности их объединения (элементов, принадлежащих хотя бы одному множеству). Коэффициент Танимото для двух векторов вычисляется следующей простой функцией:

```
def tanamoto(v1,v2):
  c1,c2,shr=0,0,0
```

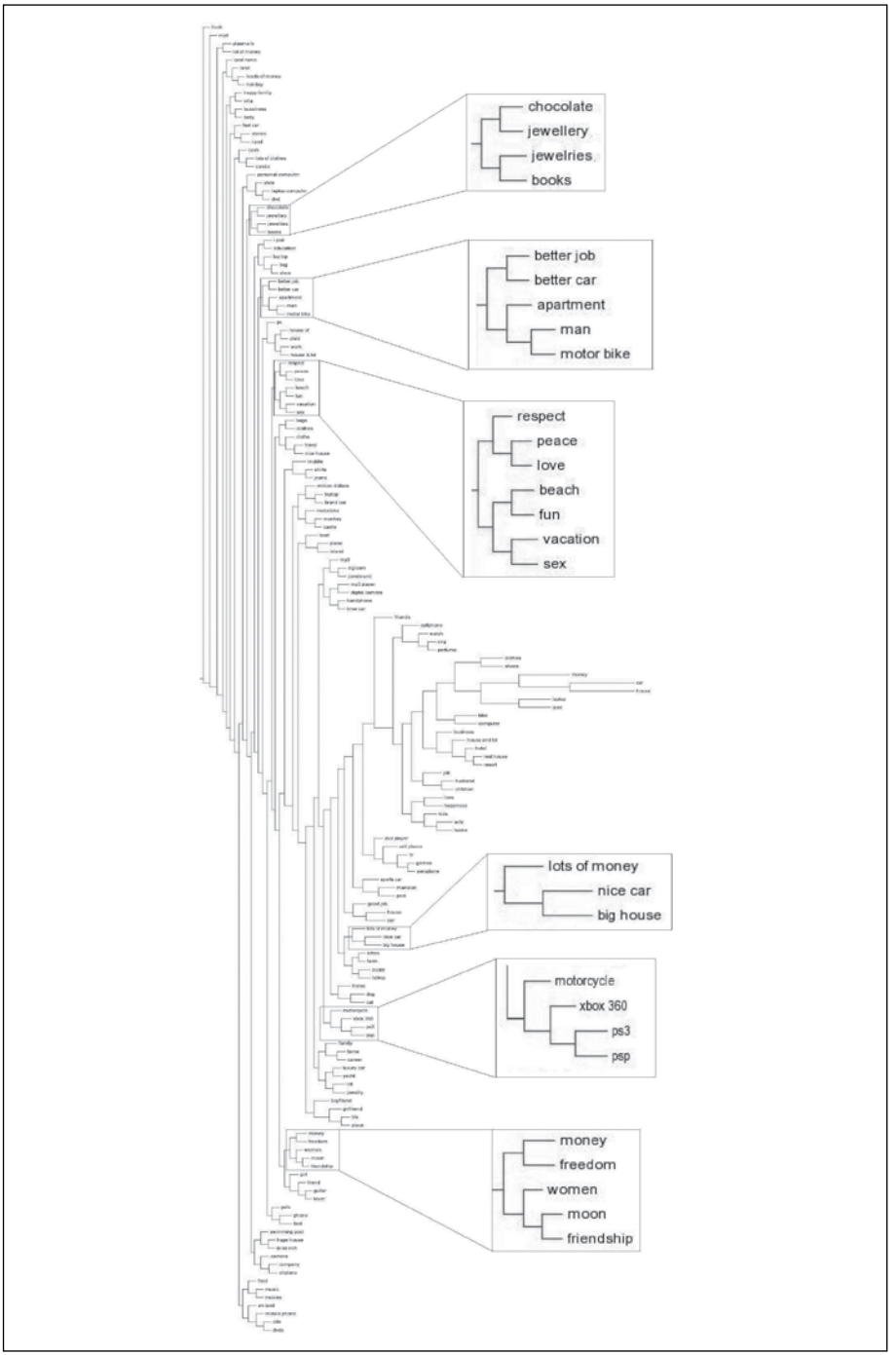


Рис. 3.6. *Кластеры предметов, которые хотели бы иметь люди*

```
  for i in range(len(v1)):
    if v1[i]!=0: c1+=1 # in v1
    if v2[i]!=0: c2+=1 # in v2
    if v1[i]!=0 and v2[i]!=0: shr+=1 # in both

  return 1.0-(float(shr)/(c1+c2-shr))
```

Она возвращает значение от 0,0 до 1,0. Значение 1,0 означает, что ни один человек, желающий приобрести первый предмет, не хочет приобретать второй. А 0,0 говорит о том, что эти два предмета хотят иметь в точности одни и те же люди.

Кластеризация результатов

Поскольку данные представлены в том же формате, что и выше, для генерирования и изображения иерархических кластеров можно воспользоваться уже написанными функциями (надо лишь добавить функцию `tanamoto` в файл clusters.py):

```
>> reload(clusters)
>> wants,people,data=clusters.readfile('zebo.txt')
>> clust=clusters.hcluster(data,distance=clusters.tanamoto)
>> clusters.drawdendrogram(clust,wants)
```

Эти команды создают новый файл clusters.jpg, в котором представлены кластеры желаемых приобретений. На рис. 3.6 изображен результат, полученный в результате обработки загруженного набора данных. С точки зрения маркетинга никаких откровений здесь нет – одни и те же люди хотят иметь Xbox, PlayStation Portable и PlayStation 3, – однако имеются также четко выраженные группы очень амбициозных людей (катер, самолет, остров) и людей, ищущих духовные ценности (друзья, любовь, счастье). Интересно также отметить, что люди желающие иметь «деньги», хотят просто «дом», тогда как алчущие «много денег» предпочли бы «красивый дом».

Изменив начальные условия поиска, количество скачиваемых страниц или запросив не «желаемые», а «располагаемые» предметы, возможно, удастся найти другие интересные группы. Можно также попробовать транспонировать матрицу и сгруппировать пользователей. Особенно любопытно было бы собрать данные о возрасте, чтобы посмотреть, как возраст сказывается на желаниях.

Просмотр данных на двумерной плоскости

Алгоритмы кластеризации, рассмотренные в этой главе, иллюстрировались с помощью визуализации данных на плоскости, а степень различия между предметами обозначалась расстоянием между ними на диаграмме. Но в большинстве реальных задач требуется кластеризовать более двух чисел, поэтому изобразить данные в двух измерениях не получится. Тем не менее, чтобы понять, как соотносятся различные предметы, было бы полезно видеть их на странице и оценивать степень схожести по близости.

В этом разделе мы ознакомимся с методом многомерного шкалирования, позволяющим найти двумерное представление набора данных. Этот алгоритм принимает на входе различие между каждой парой предметов и пытается нарисовать диаграмму так, чтобы расстояния между точками, соответствующими предметам, отражали степень их различия. Для этого сначала вычисляются желаемые расстояния (target distance) между всеми предметами. В случае набора данных о блогах для сравнения предметов применялся коэффициент корреляции Пирсона. В табл. 3.2 приведен пример.

Таблица 3.2. Пример матрицы расстояний

	A	B	C	D
A	0,0	0,2	0,8	0,7
B	0,2	0,0	0,9	0,8
C	0,8	0,9	0,0	0,1
D	0,7	0,8	0,1	0,0

Затем все предметы (в данном случае блоги) случайным образом размещаются на двумерной диаграмме, как показано на рис. 3.7.

Вычисляются попарные евклидовы расстояния (сумма квадратов разностей координат) между всеми текущими положениями предметов, как показано на рис. 3.8.

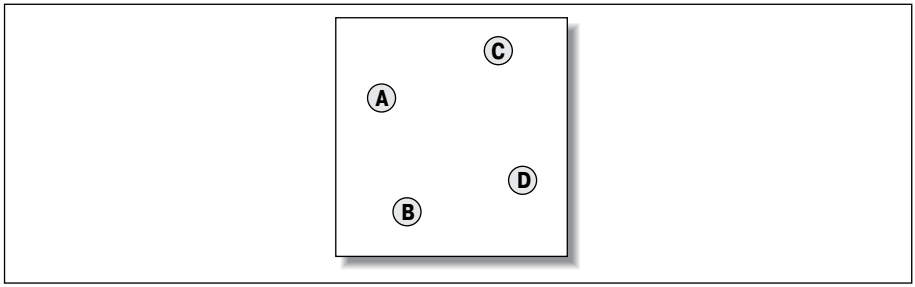


Рис. 3.7. *Начальное размещение на двумерной проекции*

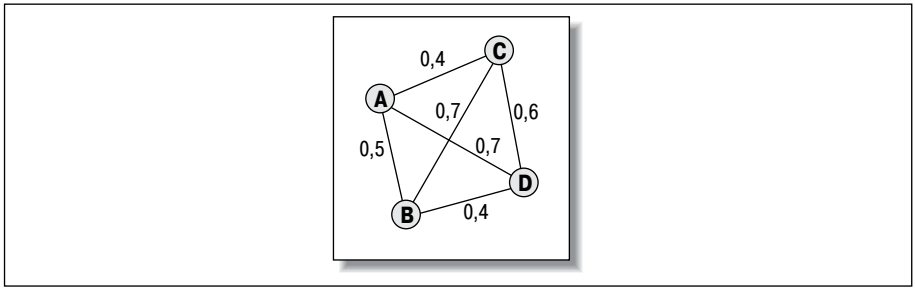


Рис. 3.8. *Попарные расстояния между предметами*

Для каждой пары предметов желаемое расстояние сравнивается с текущим и вычисляется расхождение. Каждый предмет приближается или отодвигается от своего соседа пропорционально расхождению между ними. На рис. 3.9 показаны силы, действующие на предмет A. Расстояние между A и B на этой диаграмме равно 0,5, а желаемое расстояние равно 0,2, поэтому A следует приблизить к B. В то же время A отталкивается от C и D, так как расположен к ним слишком близко.

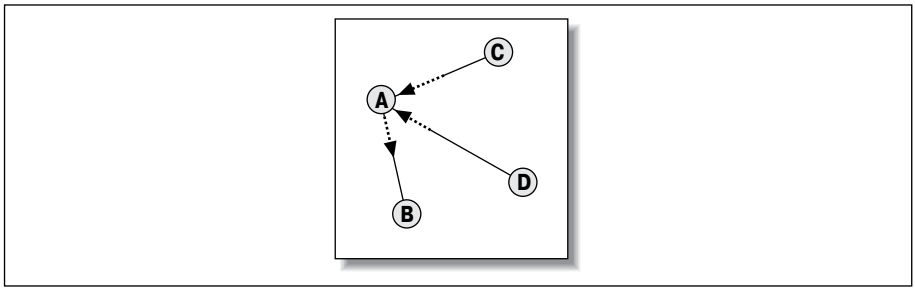


***Рис. 3.9.** Силы, действующие на предмет A*

Каждый узел перемещается под воздействием всех остальных узлов, которые притягивают или отталкивают его. После каждого такого перемещения разница между текущими и желаемыми расстояниями немного уменьшается. Эта процедура повторяется многократно и прекращается, когда общее расхождение не удается уменьшить за счет перемещения предметов.

Реализующая ее функция принимает вектор данных и возвращает массив с двумя столбцами, содержащими координаты X и Y предметов на двумерной диаграмме. Добавьте ее в файл clusters.py:

```
def scaledown(data,distance=pearson,rate=0.01):
  n=len(data)

  # Настоящие расстояния между каждой парой предметов
  realdist=[[distance(data[i],data[j]) for j in range(n)]
            for i in range(0,n)]

  outersum=0.0
  # Случайным образом инициализировать начальные положения предметов на
  # плоскости
  loc=[[random.random(),random.random( )] for i in range(n)]
  fakedist=[[0.0 for j in range(n)] for i in range(n)]

  lasterror=None
  for m in range(0,1000):
    # Вычислить евклидовы расстояния
    for i in range(n):
      for j in range(n):
        fakedist[i][j]=sqrt(sum([pow(loc[i][x]-loc[j][x],2)
                                 for x in range(len(loc[i]))]))
```

```
    # Переместить точки
    grad=[[0.0,0.0] for i in range(n)]

    totalerror=0
    for k in range(n):
      for j in range(n):
        if j==k: continue
        # Расхождение - это относительная разность между расстояниями
        errorterm=(fakedist[j][k]-realdist[j][k])/realdist[j][k]

        # Каждую точку следует отодвинуть или приблизить к другой точке
        # пропорционально вычисленному расхождению между ними
        grad[k][0]+=((loc[k][0]-loc[j][0])/fakedist[j][k])*errorterm
        grad[k][1]+=((loc[k][1]-loc[j][1])/fakedist[j][k])*errorterm

        # Следим за суммарным расхождением
        totalerror+=abs(errorterm)
    print totalerror

    # Если после перемещения ситуация ухудшилась, завершаем операцию
    if lasterror and lasterror<totalerror: break
    lasterror=totalerror

    # Сдвинуть каждую точку на величину, равную скорости обучения,
    # умноженной на градиент
    for k in range(n):
      loc[k][0]-=rate*grad[k][0]
      loc[k][1]-=rate*grad[k][1]

  return loc
```

Чтобы визуализировать результат, можно снова воспользоваться библиотекой PIL для генерирования изображения, на котором будут нанесены метки всех предметов в точках, где они теперь оказались.

```
def draw2d(data,labels,jpeg='mds2d.jpg'):
  img=Image.new('RGB',(2000,2000),(255,255,255))
  draw=ImageDraw.Draw(img)
  for i in range(len(data)):
    x=(data[i][0]+0.5)*1000
    y=(data[i][1]+0.5)*1000
    draw.text((x,y),labels[i],(0,0,0))
  img.save(jpeg,'JPEG')
```

Для прогона этого алгоритма сначала следует вызвать функцию `scaledown` **для получения двумерного набора данных, а затем** `draw2d` **– для его отрисовки.**

```
>> reload(clusters)
>> blognames,words,data=clusters.readfile('blogdata.txt')
>> coords=clusters.scaledown(data)
...
>> clusters.draw2d(coords,blognames,jpeg='blogs2d.jpg')
```

На рис. 3.10 показан результат работы алгоритма многомерного шкалирования. Кластеры видны не так хорошо, как на дендрограмме, но тем не менее налицо группировка по тематике, например блоги, относящиеся к поисковым машинам, разместились у верхнего края. Они оказались далеко от блогов, посвященных политике и знаменитостям. Если бы мы построили трехмерное представление, то кластеры были бы выражены более отчетливо, но по понятным причинам их было бы сложнее изобразить на бумаге.

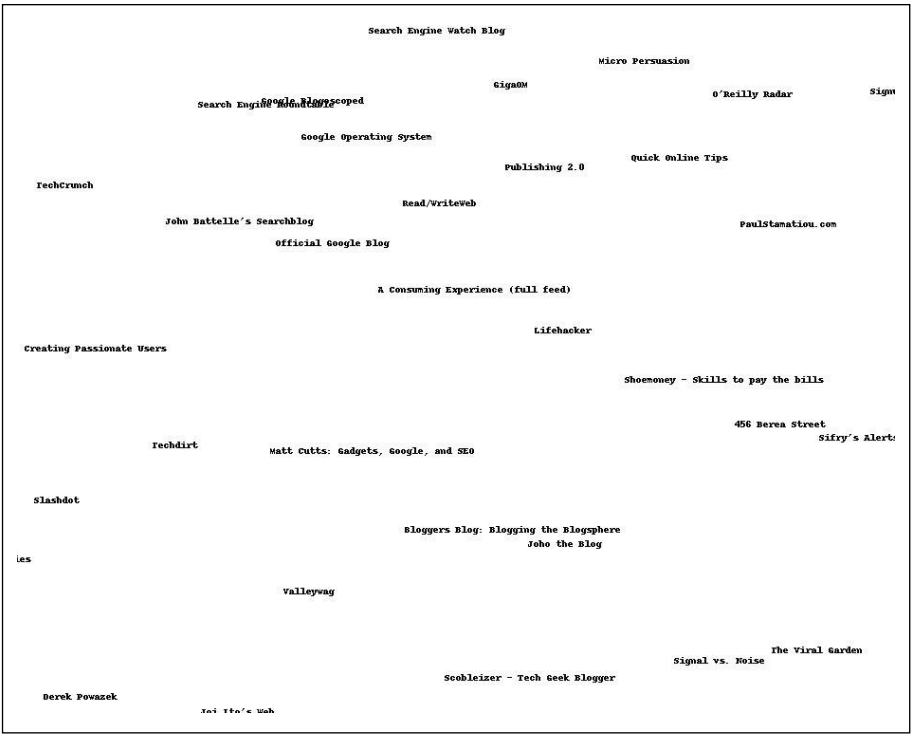


Рис. 3.10. *Часть двумерного представления пространства блогов*

Что еще можно подвергнуть кластеризации

В этой главе мы рассмотрели два набора данных, но можно было бы сделать еще многое. Набор данных с сайта *del.icio.us* из главы 2 тоже можно подвергнуть кластеризации для нахождения групп пользователей или закладок. Как каналы блогов были представлены в виде векторов слов, так и любой набор загруженных страниц можно свести к словам.

Эти идеи можно распространить на другие области и отыскать немало интересных закономерностей – классифицировать форумы по употреблению слов; компании, представленные на сайте Yahoo! Finance, – по

различным статистикам; лучших рецензентов на сайте Amazon – по тому, что им нравится. Было бы также интересно взглянуть на большую социальную сеть типа MySpace и кластеризовать людей по тому, какие у них друзья, или по иной информации, которую они сообщают о себе (любимые музыкальные группы, еда и т. д.).

Тема изображения предметов в пространстве в зависимости от их параметров будет неоднократно затрагиваться на страницах этой книги. Многомерное шкалирование – это эффективный способ представить набор данных в виде, удобном для интерпретации. Важно понимать, что в процессе шкалирования часть информации теряется, но результат поможет лучше понять работу различных алгоритмов.

Упражнения

1. Пользуясь API сайта *del.icio.us* из главы 2, создайте набор данных о закладках, подходящий для кластеризации. Прогоните для него алгоритм иерархической кластеризации и метод K-средних.
2. Модифицируйте код разбора блога так, чтобы кластеризовать отдельные записи, а не блоги целиком. Попадают ли записи из одного и того же блога в один кластер? А что можно сказать о записях, датированных одним и тем же числом?
3. Попробуйте использовать для кластеризации блогов обычное евклидово расстояние. Как изменятся результаты?
4. Выясните, что такое манхэттенское расстояние. Напишите функцию для его вычисления и посмотрите, как изменятся результаты кластеризации набора данных с сайта Zebo.
5. Модифицируйте функцию кластеризации методом K-средних так, чтобы помимо результатов кластеризации она возвращала суммарное расстояние между элементами и соответствующими им центроидами.
6. Выполнив упражнение 5, напишите функцию, которая запускает кластеризацию методом K-средних с различными значениями k. Как суммарное расстояние изменяется по мере увеличения количества кластеров? В какой момент улучшение за счет увеличения количества кластеров становится пренебрежимо малым?
7. Результат двумерного шкалирования легко вывести на печать, но алгоритм позволяет проецировать набор данных на пространство любого числа измерений. Попробуйте изменить код так, чтобы набор данных проецировался на прямую. А потом спроецируйте его на трехмерное пространство.

4

Поиск и ранжирование

В этой главе рассматриваются системы полнотекстового поиска, которые позволяют искать слова в большом наборе документов и сортируют результаты поиска по релевантности найденных документов запросу. Алгоритмы полнотекстового поиска относятся к числу важнейших среди алгоритмов коллективного разума. Новые идеи в этой области помогли сколотить целые состояния. Широко распространено мнение, что своей быстрой эволюцией от академического проекта к самой популярной поисковой машине в мире система Google обязана прежде всего алгоритму ранжирования страниц PageRank. Об одном из его вариантов вы узнаете из этой главы.

Информационный поиск – это очень широкая область с долгой историей. В этой главе мы сможем затронуть лишь немногие ключевые идеи, но тем не менее построим поисковую машину, которая будет индексировать набор документов, и подскажем, в каких направлениях ее можно усовершенствовать. Хотя в центре нашего внимания будут алгоритмы поиска и ранжирования, а не требования к инфраструктуре, необходимой для индексирования больших участков Всемирной паутины, созданная в этой главе система сможет без труда проиндексировать до 100 000 страниц. В этой главе вы ознакомитесь со всеми основными этапами: обходом страниц, индексированием и поиском, – а также научитесь различными способами ранжировать результаты.

Что такое поисковая машина

Первый шаг при создании поисковой машины – разработать методику сбора документов. Иногда для этого применяется *ползание* (начинаем с небольшого набора документов и переходим по имеющимся в них

ссылкам), а иногда отправной точкой служит фиксированный набор документов, быть может, хранящихся в корпоративной сети интранет.

Далее собранные документы необходимо проиндексировать. Обычно для этого строится большая таблица, содержащая список документов и вхождений различных слов. В зависимости от конкретного приложения сами документы могут и не храниться в базе данных; в индексе находится лишь ссылка (путь в файловой системе или URL) на их местонахождение

Ну и последний шаг – это, конечно, возврат ранжированного списка документов в ответ на запрос. Имея индекс, найти документы, содержащие заданные слова, сравнительно несложно; хитрость заключается в том, как отсортировать результаты. Можно придумать огромное количество метрик, и недостатка в способах их настройки для изменения порядка документов тоже нет. Стоит лишь ознакомиться с разными метриками, как возникает желание, чтобы большие поисковики предоставляли средства для более точного контроля («Ну почему я не могу сказать Google, что мои слова должны находиться в документе рядом?»). В этой главе мы сначала рассмотрим несколько метрик, основанных на содержимом страницы, например на частоте вхождения слов, а затем метрики, в основе которых лежит информация, внешняя по отношению к странице, например алгоритм ранжирования страниц PageRank, в котором учитываются ссылки на оцениваемую страницу с других страниц.

Наконец необходимо построить *нейронную сеть* для ранжирования запросов. Такая сеть обучается ассоциировать запросы с результатами в зависимости от того, по каким ссылкам щелкает пользователь, получив список результатов. Эта информация позволяет изменить сортировку результатов с учетом того, как человек переходил по найденным ссылкам в прошлом.

Для проработки примеров из этой главы нам понадобится создать на языке Python модуль `searchengine`, в котором будет два класса: один – для ползания по сети и создания базы данных, а второй – для выполнения полнотекстового поиска в этой базе в ответ на запрос. В примерах мы будем пользоваться базой данных *SQLite*, но алгоритмы можно легко адаптировать для работы с более традиционными клиент-серверными СУБД.

Для начала создайте новый файл searchengine.py и добавьте в него класс `crawler` и заглушки для методов, которые мы напишем на протяжении этой главы:

```
class crawler:
  # Инициализировать паука, передав ему имя базы данных
  def __init__(self,dbname):
    pass

  def __del__(self):
    pass
```

```
def dbcommit(self):
  pass

# Вспомогательная функция для получения идентификатора и
# добавления записи, если такой еще нет
def getentryid(self,table,field,value,createnew=True):
  return None

# Индексирование одной страницы
def addtoindex(self,url,soup):
  print 'Индексируется %s' % url

# Извлечение текста из HTML-страницы (без тегов)
def gettextonly(self,soup):
  return None

# Разбиение текста на слова
def separatewords(self,text):
  return None

# Возвращает true, если данный URL уже проиндексирован
def isindexed(self,url):
  return False

# Добавление ссылки с одной страницы на другую
def addlinkref(self,urlFrom,urlTo,linkText):
  pass

# Начиная с заданного списка страниц, выполняет поиск в ширину
# до заданной глубины, индексируя все встречающиеся по пути
# страницы
def crawl(self,pages,depth=2):
  pass

# Создание таблиц в базе данных
def createindextables(self):
  pass
```

Простой паук

В предположении, что у вас на диске нет большого числа HTML-документов, готовых для индексирования, я покажу, как можно написать простого паука. В качестве затравки мы предложим ему проиндексировать небольшой набор страниц, но он будет переходить по ссылкам, ведущим с этих страниц на другие, а с тех – на третьи и т. д. В этом случае говорят, что паук *ползает* по сети.

Чтобы решить эту задачу, ваша программа должна будет загрузить страницу, передать ее индексатору (его мы напишем в следующем разделе), а затем проанализировать ее текст, ища в нем ссылки на страницы,

куда нужно будет переползти на следующем шаге. К счастью, существует несколько библиотек, помогающих запрограммировать эту процедуру.

Для примеров из этой главы я взял несколько тысяч страниц из Википедии и разместил их статические копии по адресу *http://kiwitobes.com/wiki*.

Вы вольны «натравить» паука на любой набор страниц, но, если хотите сравнить результаты с приведенными в этой главе, пользуйтесь моим набором.

Библиотека urllib2

Библиотека urllib2, входящая в дистрибутив Python, предназначена для скачивания страниц. От вас требуется только указать URL. В этом разделе мы воспользуемся ею, чтобы скачать страницы для последующего индексирования. Чтобы посмотреть, как она работает, запустите интерпретатор Python и введите следующие команды:

```
>> import urllib2
>> c=urllib2.urlopen('http://kiwitobes.com/wiki/Programming_language.html')
>> contents=c.read( )
>> print contents[0:50]
'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Trans'
```

Для того чтобы сохранить HTML-код страницы в строке, вам нужно лишь создать соединение и прочитать содержимое страницы.

Код паука

Программируя паука, мы будем пользоваться библиотекой Beautiful Soup, с которой ознакомились в главе 3; она строит структурированное представление веб-страниц. Эта библиотека очень терпимо относится к некорректной HTML-разметке, что весьма полезно при построении паука, поскольку никогда не знаешь, с чем придется столкнуться. Дополнительную информацию о загрузке и установке библиотеки Beautiful Soup см. в приложении А.

С помощью библиотек urllib2 и Beautiful Soup можно построить паука, который принимает на входе список подлежащих индексированию URL и, переползая по ссылкам, находит другие страницы. Начнем с добавления в файл searchengine.py таких предложений:

```
import urllib2
from BeautifulSoup import *
from urlparse import urljoin

# Создать список игнорируемых слов
ignorewords=set(['the','of','to','and','a','in','is','it'])
```

Теперь можно написать код главной функции паука. Пока она ничего не сохраняет, а просто печатает посещаемые URL, демонстрируя, что какие-то действия выполняются. Добавьте следующий код в конец файла (поместив его в класс `crawler`):

```
def crawl(self,pages,depth=2):
  for i in range(depth):
    newpages=set( )
    for page in pages:
      try:
        c=urllib2.urlopen(page)
      except:
        print "Не могу открыть %s" % page
        continue
      soup=BeautifulSoup(c.read( ))
      self.addtoindex(page,soup)

      links=soup('a')
      for link in links:
        if ('href' in dict(link.attrs)):
          url=urljoin(page,link['href'])
          if url.find("'")!=-1: continue
          url=url.split('#')[0] # удалить часть URL после знака #
          if url[0:4]=='http' and not self.isindexed(url):
            newpages.add(url)
          linkText=self.gettextonly(link)
          self.addlinkref(page,url,linkText)

      self.dbcommit( )

    pages=newpages
```

Эта функция в цикле обходит список страниц, вызывая для каждой функцию `addtoindex` (пока что она всего лишь печатает URL, но в следующем разделе мы это исправим). Далее с помощью библиотеки Beautiful Soup она получает все ссылки на данной странице и добавляет их URL в список `newpages`. В конце цикла `newpages` присваивается `pages`, и процесс повторяется.

Эту функцию можно было бы определить рекурсивно, так что при обработке каждой ссылки она вызывала бы сама себя. Но, реализовав поиск в ширину, мы упростим модификацию кода в будущем, позволив пауку ползать непрерывно или сохранять список неиндексированных страниц для последующего индексирования. Кроме того, таким образом мы избегаем опасности переполнить стек.

Вы можете протестировать функцию, введя следующие команды в интерпретаторе (дожидаться завершения необязательно; когда надоест, можете нажать сочетание клавиш `Ctrl+C`):

```
>> import searchengine
>> pagelist=['http://kiwitobes.com/wiki/Perl.html']
>> crawler=searchengine.crawler('')
>> crawler.crawl(pagelist)
Индексируется http://kiwitobes.com/wiki/Perl.html
Не могу открыть http://kiwitobes.com/wiki/Module_%28programming%29.html
Индексируется http://kiwitobes.com/wiki/Open_Directory_Project.html
Индексируется http://kiwitobes.com/wiki/Common_Gateway_Interface.html
```

Вероятно, вы заметили, что некоторые страницы повторяются. В коде имеется заглушка для функции `isindexed`, которая вызывается перед добавлением страницы в список `newpages` и проверяет, была ли страница недавно проиндексирована. Это позволяет в любой момент «натравливать» паука на любой список URL, не опасаясь, что будет проделана ненужная работа.

Построение индекса

Наш следующий шаг – подготовка базы данных для хранения полнотекстового индекса. Я уже говорил, что такой индекс представляет собой список слов, с каждым из которых ассоциировано множество документов, где это слово встречается, и место вхождения слова в документ. В данном примере мы анализируем только текст на странице, игнорируя все нетекстовые элементы. Кроме того, индексируются только сами слова, а все знаки препинания удаляются. Такой метод выделения слов несовершенен, но для построения простой поисковой машины сойдет.

Поскольку рассмотрение различных баз данных и конфигурирования СУБД выходит за рамки данной книги, я покажу лишь, как хранить индекс в базе данных SQLite. SQLite – это встраиваемая СУБД, которую очень легко настроить, причем вся база данных хранится в одном файле. Поскольку запросы формулируются на языке SQL, будет нетрудно перенести код примеров на другую СУБД. Реализация на Python называется pysqlite, а скачать ее можно со страницы *http://initd.org/tracker/pysqlite*.

Имеется инсталлятор для Windows и инструкции по установке на другие операционные системы. В приложении А приведена дополнительная информация по получению и установке pysqlite.

После установки SQLite добавьте следующую строку в файл searchengine.py:

```
from pysqlite2 import dbapi2 as sqlite
```

Необходимо также изменить методы `__init__`, `__del__` и `dbcommit`, добавив код открытия и закрытия базы данных:

```
def __init__(self,dbname):
  self.con=sqlite.connect(dbname)

def __del__(self):
  self.con.close( )
```

```
def dbcommit(self):
  self.con.commit( )
```

Создание схемы

Подождите запускать программу – нужно сначала подготовить базу данных. Схема простого индекса состоит из пяти таблиц. Первая (`urllist`) – это список проиндексированных URL. Вторая (`wordlist`) – список слов, а третья (`wordlocation`) – список мест вхождения слов в документы. В оставшихся двух таблицах хранятся ссылки между документами. В таблице `link` хранятся идентификаторы двух URL, связанных ссылкой, а в таблице `linkwords` есть два столбца – `wordid` и `linkid` – для хранения слов, составляющих ссылку. Вся схема изображена на рис. 4.1.

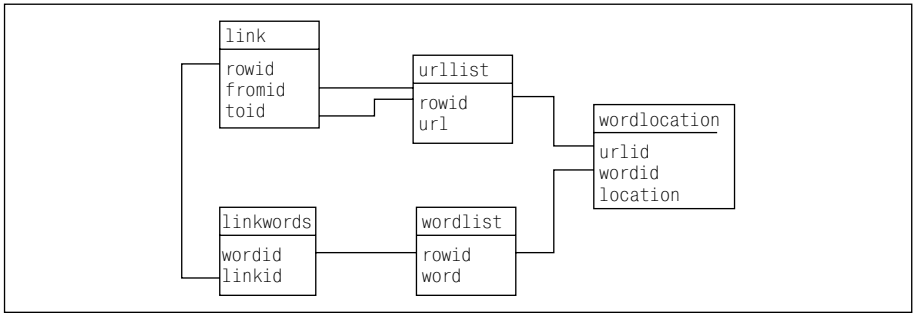


***Рис. 4.1.** Схема базы данных поисковой машины*

В каждой таблице SQLite по умолчанию имеется поле `rowid`, поэтому явно задавать ключевые поля необязательно. Для создания базы данных добавьте в класс `crawler` в файле searchengine.py следующую функцию:

```
def createindextables(self):
  self.con.execute('create table urllist(url)')
  self.con.execute('create table wordlist(word)')
  self.con.execute('create table wordlocation(urlid,wordid,location)')
  self.con.execute('create table link(fromid integer,toid integer)')
  self.con.execute('create table linkwords(wordid,linkid)')
  self.con.execute('create index wordidx on wordlist(word)')
  self.con.execute('create index urlidx on urllist(url)')
  self.con.execute('create index wordurlidx on wordlocation(wordid)')
  self.con.execute('create index urltoidx on link(toid)')
  self.con.execute('create index urlfromidx on link(fromid)')
  self.dbcommit( )
```

Эта функция создает все нужные нам таблицы, а также ряд индексов, ускоряющих поиск. Индексы необходимы, потому что набор данных может быть очень велик. Введите в интерпретаторе следующие команды для создания базы данных searchindex.db:

```
>> reload(searchengine)
>> crawler=searchengine.crawler('searchindex.db')
>> crawler.createindextables( )
```

Позже вы добавим в базу еще одну таблицу для хранения метрики ранжирования на основе указывающих на страницу внешних ссылок.

Выделение слов на странице

Файлы, загруженные из Сети, обычно представлены в формате HTML и потому содержат много тегов, атрибутов и прочей информации, которую индексировать не нужно. Стало быть, для начала нужно выделить те части страницы, которые содержат текст. Для этого можно поискать в «супе» текстовые узлы и взять их содержимое. Добавьте следующий код в функции `gettextonly`:

```
def gettextonly(self,soup):
  v=soup.string
  if v==None:
    c=soup.contents
    resulttext=''
    for t in c:
      subtext=self.gettextonly(t)
      resulttext+=subtext+'\n'
    return resulttext
  else:
    return v.strip( )
```

Эта функция возвращает длинную строку, содержащую весь имеющийся на странице текст. Для этого она рекурсивно обходит объектную модель HTML-документа, отыскивая текстовые узлы. Текст, находящийся в различных разделах, помещается в отдельные абзацы. Для некоторых метрик, которые мы будем в дальнейшем вычислять, порядок разделов имеет значение.

Далее вызывается функция `separatewords`, которая разбивает строку на отдельные слова, чтобы их можно было добавить в индекс. Сделать это правильно не так просто, как может показаться, и существует немало работ о том, как улучшить эту методику. Однако для наших примеров достаточно считать, что все символы, не являющиеся буквами или цифрами, – разделители. Для разбиения на слова можно воспользоваться регулярным выражением. Вставьте в определение `separatewords` следующий код:

```
def separatewords(self,text):
  splitter=re.compile('\\W*')
  return [s.lower( ) for s in splitter.split(text) if s!='']
```

Поскольку эта функция считает разделителями все символы, кроме букв и цифр, то с выделением английских слов особых проблем не будет, но такие термины, как C++, она обрабатывает неправильно (впрочем, со словом python справляется благополучно). Можете попробовать улучшить регулярное выражение, чтобы оно точнее выделяло слова.

Еще одна возможность улучшения – удалять из слов суффиксы, воспользовавшись алгоритмом *выделения основы*. Например, слово indexing при этом превращается в index, поэтому человек, который ищет слово index, получит также документы, содержащие слово indexing. Чтобы реализовать этот подход, необходимо выделять основы при индексировании документов и на этапе начальной обработки запроса. Развернутое обсуждение этой темы выходит за рамки данной главы, но реализация такого алгоритма на Python имеется в библиотеке *Porter Stemmer* по адресу *http://www.tartarus.org/~martin/PorterStemmer/index.html*.

Добавление в индекс

Теперь мы готовы написать код метода `addtoindex`. Он вызывает две функции, написанные в предыдущем разделе, чтобы получить список слов на странице. Затем эта страница и все найденные на ней слова добавляются в индекс и создаются ссылки между словами и их вхождениями в документ. В нашем примере адресом вхождения будет считаться номер слова в списке слов.

Вот код метода `addtoindex`:

```
def addtoindex(self,url,soup):
  if self.isindexed(url): return
  print 'Индексируется '+url

  # Получить список слов
  text=self.gettextonly(soup)
  words=self.separatewords(text)

  # Получить идентификатор URL
  urlid=self.getentryid('urllist','url',url)

  # Связать каждое слово с этим URL
  for i in range(len(words)):
    word=words[i]
    if word in ignorewords: continue
      wordid=self.getentryid('wordlist','word',word)
     self.con.execute("insert into wordlocation(urlid,wordid,location) \
       values (%d,%d,%d)" % (urlid,wordid,i))
```

Необходимо также написать код вспомогательной функции `getentryid`. От нее требуется лишь одно – вернуть идентификатор записи. Если такой записи еще нет, она создается и возвращается ее идентификатор:

```
def getentryid(self,table,field,value,createnew=True):
  cur=self.con.execute(
  "select rowid from %s where %s='%s'" % (table,field,value))
  res=cur.fetchone( )
  if res==None:
```

```
    cur=self.con.execute(
    "insert into %s (%s) values ('%s')" % (table,field,value))
    return cur.lastrowid
  else:
    return res[0]
```

И наконец необходимо написать код функции `isindexed`, которая определяет, есть ли указанная страница в базе данных и, если да, ассоциированы ли с ней какие-нибудь слова:

```
def isindexed(self,url):
  u=self.con.execute \
    ("select rowid from urllist where url='%s'" % url).fetchone( )
  if u!=None:
    # Проверяем, что страница посещалась
    v=self.con.execute(
    'select * from wordlocation where urlid=%d' % u[0]).fetchone( )
    if v!=None: return True
  return False
```

Теперь можно снова запустить паука и дать ему возможность проиндексировать страницы по-настоящему. Это можно сделать в интерактивном сеансе:

```
>> reload(searchengine)
>> crawler=searchengine.crawler('searchindex.db')
>> pages= \
.. ['http://kiwitobes.com/wiki/Categorical_list_of_programming_languages.html']
>> crawler.crawl(pages)
```

Скорее всего, паук будет работать долго. Я предлагаю не дожидаться, пока он закончит, а загрузить готовую базу данных searchindex.db по адресу *http://kiwitobes.com/db/searchindex.db* и сохранить ее в одной папке с программой.

Если вы хотите удостовериться, что паук отработал правильно, проверьте записи, добавленные для какого-нибудь слова, сделав запрос к базе:

```
>> [row for row in crawler.con.execute(
.. 'select rowid from wordlocation where wordid=1')]
[(1,), (46,), (330,), (232,), (406,), (271,), (192,),...
```

В ответ возвращается список идентификаторов всех URL, содержащих слово word, то есть вы успешно выполнили полнотекстовый поиск. Для начала неплохо, но так можно искать только по одному слову, а документы возвращаются в том порядке, в каком загружались. В следующем разделе мы покажем, как обрабатывать запросы, состоящие из нескольких слов.

Запросы

Теперь у нас есть работающий паук и большой набор проиндексированных документов, и мы готовы приступить к реализации той части

поисковой машины, которая выполняет поиск. Сначала создайте в файле searchengine.py новый класс, который будет использоваться для поиска:

```
class searcher:
  def __init__(self,dbname):
    self.con=sqlite.connect(dbname)

  def __del__(self):
    self.con.close( )
```

Таблица `wordlocation` обеспечивает простой способ связать слова с документами, так что мы можем легко найти, какие страницы содержат данное слово. Однако поисковая машина была бы довольно слабой, если бы не позволяла задавать запросы с несколькими словами. Чтобы исправить это упущение, нам понадобится функция, которая принимает строку запроса, разбивает ее на слова и строит SQL-запрос для поиска URL тех документов, в которые входят все указанные слова. Добавьте следующую функцию в класс `searcher`:

```
def getmatchrows(self,q):
  # Строки для построения запроса
  fieldlist='w0.urlid'
  tablelist=''
  clauselist=''
  wordids=[]

  # Разбить поисковый запрос на слова по пробелам
  words=q.split(' ')
  tablenumber=0

  for word in words:
    # Получить идентификатор слова
    wordrow=self.con.execute(
      "select rowid from wordlist where word='%s'" % word).fetchone( )
    if wordrow!=None:
      wordid=wordrow[0]
      wordids.append(wordid)
      if tablenumber>0:
        tablelist+=','
        clauselist+=' and '
        clauselist+='w%d.urlid=w%d.urlid and ' % (tablenumber-1,tablenumber)
      fieldlist+=',w%d.location' % tablenumber
      tablelist+='wordlocation w%d' % tablenumber
      clauselist+='w%d.wordid=%d' % (tablenumber,wordid)
      tablenumber+=1

  # Создание запроса из отдельных частей
  fullquery='select %s from %s where %s' % (fieldlist,tablelist,clauselist)
  cur=self.con.execute(fullquery)
  rows=[row for row in cur]

  return rows,wordids
```

На первый взгляд функция довольно сложна, но, по существу, она просто создает по одной ссылке на таблицу `wordlocation` для каждого слова и выполняет соединение таблиц по идентификаторам URL (рис. 4.2).

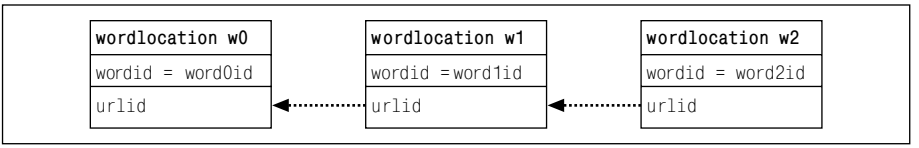


Рис. 4.2. *Соединение таблиц, выполняемое функцией getmatchrows*

Таким образом, для двух слов с идентификаторами 10 и 17 формируется следующий SQL-запрос:

```
select w0.urlid,w0.location,w1.location
from   wordlocation w0,wordlocation w1
where  w0.urlid=w1.urlid
and    w0.wordid=10
and    w1.wordid=17
```

Попробуйте вызвать эту функцию, чтобы выполнить поиск по нескольким словам:

```
>> reload(searchengine)
>> e=searchengine.searcher('searchindex.db')
>> e.getmatchrows('functional programming')
([(1, 327, 23), (1, 327, 162), (1, 327, 243), (1, 327, 261),
  (1, 327, 269), (1, 327, 436), (1, 327, 953),..
```

Обратите внимание, что каждый идентификатор URL возвращается многократно с различными комбинациями адресов вхождений слов. В следующих разделах мы рассмотрим некоторые способы ранжирования результатов. Для *ранжирования по содержимому* есть несколько возможных метрик, которые для определения релевантности запросу учитывают только содержимое страницы. При *ранжировании с учетом внешних ссылок* для определения того, что важно, принимается во внимание структура ссылок на сайт. Мы рассмотрим также вопрос о том, как можно улучшить ранжирование, учитывая, по каким ссылкам пользователи переходят, получив результаты поиска.

Ранжирование по содержимому

Пока что мы научились находить страницы, соответствующие запросам, но возвращаем мы их в том порядке, в котором они посещались пауком. Если набор страниц велик, то вам предстоит просеять кучу не относящихся к делу документов, в которых встречаются упомянутые в запросе слова, чтобы найти то, что вас действительно интересует. Чтобы решить эту проблему, необходимо как-то присвоить страницам *ранг* относительно данного запроса и уметь возвращать их в порядке убывания рангов.

В этом разделе мы рассмотрим несколько способов вычисления ранга на основе самого запроса и содержимого страницы, а именно:

Частота слов

Количество вхождений в документ слова, указанного в запросе, помогает определить степень релевантности документа.

Расположение в документе

Основная тема документа, скорее всего, раскрывается ближе к его началу.

Расстояние между словами

Если в запросе несколько слов, то они должны встречаться в документе рядом.

Самые первые поисковые машины часто работали только с подобными метриками и тем не менее давали пристойные результаты. В следующих разделах мы посмотрим, как можно улучшить результат, принимая во внимание информацию, внешнюю по отношению к самой странице, например количество и качество указывающих на нее ссылок. Но сначала нам понадобится новый метод, который принимает на входе запрос, получает строки, помещает их в словарь и отображает в виде отформатированного списка. Добавьте в класс `searcher` следующие функции:

```
def getscoredlist(self,rows,wordids):
  totalscores=dict([(row[0],0) for row in rows])

  # Сюда мы позже поместим функции ранжирования
  weights=[]

  for (weight,scores) in weights:
    for url in totalscores:
      totalscores[url]+=weight*scores[url]

  return totalscores

def geturlname(self,id):
  return self.con.execute(
  "select url from urllist where rowid=%d" % id).fetchone( )[0]

def query(self,q):
  rows,wordids=self.getmatchrows(q)
  scores=self.getscoredlist(rows,wordids)
  rankedscores=sorted([(score,url) for (url,score) in scores.items( )],\
  reverse=1)
  for (score,urlid) in rankedscores[0:10]:
    print '%f\t%s' % (score,self.geturlname(urlid))
```

Сейчас этот метод не применяет к результатам никакого ранжирования, но отображает URL, оставляя место для будущего ранга:

```
>> reload(searchengine)
>> e=searchengine.searcher('searchindex.db')
```

```
>> e.query('functional programming')
0.000000 http://kiwitobes.com/wiki/XSLT.html
0.000000 http://kiwitobes.com/wiki/XQuery.html
0.000000 http://kiwitobes.com/wiki/Unified_Modeling_Language.html
...
```

Наиболее важна здесь функция `getscoredlist`, код которой мы будем постепенно уточнять на протяжении этого раздела. По мере добавления функций ранжирования мы будем вводить их в список `weights` (строка, выделенная полужирным шрифтом) и начнем получать реальные результаты.

Функция нормализации

Все рассматриваемые ниже функции ранжирования возвращают словарь, в котором ключом является идентификатор URL, а значением – числовой ранг. Иногда лучшим считается больший ранг, иногда – меньший. Чтобы сравнивать результаты, получаемые разными методами, необходимо как-то *нормализовать* их, то есть привести к одному и тому же диапазону и направлению.

Функция нормализации принимает на входе словарь идентификаторов и рангов и возвращает новый словарь, в котором идентификаторы те же самые, а ранг находится в диапазоне от 0 до 1. Ранги масштабируются по близости к наилучшему результату, которому всегда приписывается ранг 1. От вас требуется лишь передать функции список рангов и указать, какой ранг лучше – меньший или больший:

```
def normalizescores(self,scores,smallIsBetter=0):
  vsmall=0.00001 # Предотвратить деление на нуль
  if smallIsBetter:
    minscore=min(scores.values( ))
    return dict([(u,float(minscore)/max(vsmall,l)) for (u,l) \
      in scores.items( )])
  else:
    maxscore=max(scores.values( ))
    if maxscore==0: maxscore=vsmall
    return dict([(u,float(c)/maxscore) for (u,c) in scores.items( )])
```

Каждая функция ранжирования вызывает эту функцию для нормализации полученных результатов.

Частота слов

Метрика, основанная на частоте слов, ранжирует страницу исходя из того, сколько раз в ней встречаются слова, упомянутые в запросе. Если я выполняю поиск по слову python, то в начале списка скорее получу страницу, где это слово встречается много раз, а не страницу о музыканте, который где-то в конце упомянул, что у него дома живет питон.

Ниже приведена соответствующая функция, добавьте эту функцию в класс `searcher`:

```
def frequencyscore(self,rows):
  counts=dict([(row[0],0) for row in rows])
  for row in rows: counts[row[0]]+=1
  return self.normalizescores(counts)
```

Эта функция создает словарь, в котором для каждого идентификатора URL, встречающегося в списке `rows`, указано, сколько раз в нем попадаются слова, указанные в запросе. Затем она нормализует ранги (в данном случае чем больше ранг, тем лучше) и возвращает результат.

Чтобы активировать ранжирование документов по частоте слов, измените строку функции `getscoredlist`, где определяется список `weights`, следующим образом:

```
weights=[(1.0,self.frequencyscore(rows))]
```

Теперь можно снова выполнить поиск и посмотреть, что получится при использовании этой метрики:

```
>> reload(searchengine)
>> e=searchengine.searcher('searchindex.db')
>> e.query('functional programming')
1.000000 http://kiwitobes.com/wiki/Functional_programming.html
0.262476 http://kiwitobes.com/wiki/Categorical_list_of_programming_languages.
html
0.062310 http://kiwitobes.com/wiki/Programming_language.html
0.043976 http://kiwitobes.com/wiki/Lisp_programming_language.html
0.036394 http://kiwitobes.com/wiki/Programming_paradigm.html
...
```

Страница о функциональном программировании (Functional programming) теперь оказалась на первом месте, а за ней следует еще несколько релевантных страниц. Отметим, что ранг Functional programming в четыре раза больше, чем ранг следующей страницы. Большинство поисковых машин не сообщают пользователям о ранге страниц, но для некоторых приложений эти данные весьма ценны. Например, если ранг первого результата выше некоторого порога, то можно сразу отправить пользователя на соответствующую страницу. Или можно отображать результаты шрифтом, размер которого пропорционален релевантности.

Расположение в документе

Еще одна простая метрика для определения релевантности страницы запросу – расположение поисковых слов на странице. Обычно, если страница релевантна поисковому слову, то это слово расположено близко к началу страницы, быть может, даже находится в заголовке. Чтобы воспользоваться этим наблюдением, поисковая машина может

приписывать результату больший ранг, если поисковое слово встречается в начале документа. К счастью, на этапе индексирования страниц мы сохранили адреса вхождения слов и заголовок страницы находится в начале списка.

Добавьте в класс `searcher` следующий метод:

```
def locationscore(self,rows):
  locations=dict([(row[0],1000000) for row in rows])
  for row in rows:
    loc=sum(row[1:])
    if loc<locations[row[0]]: locations[row[0]]=loc

  return self.normalizescores(locations,smallIsBetter=1)
```

Напомним, что первый элемент в каждой строке `row` – это идентификатор URL, а за ним следуют адреса вхождений всех поисковых слов. Каждый идентификатор может встречаться несколько раз, по одному для каждой комбинации вхождений. Для каждой строки этот метод суммирует адреса вхождений всех слов и сравнивает результат с наилучшим, вычисленным для данного URL к текущему моменту. Затем окончательные результаты передаются функции `normalize`. Заметьте, параметр `smallIsBetter` означает, что ранг 1,0 будет присвоен URL с наименьшей суммой адресов вхождений.

Чтобы выяснить, как будет выглядеть результат, если учитывать только расположение в документе, измените определение списка `weights` следующим образом:

```
weights=[(1.0,self.locationscore(rows))]
```

А теперь снова выполните запрос в интерпретаторе:

```
>> reload(searchengine)
>> e=searchengine.searcher('searchindex.db')
>> e.query('functional programming')
```

Вы увидите, что страница Functional programming по-прежнему на первом месте, но за ней следуют примеры языков функционального программирования. Ранее были возвращены документы, в которых поисковые слова встречались несколько раз, но это, как правило, были обсуждения языков программирования вообще. Теперь же присутствие поисковых слов в первых предложениях документа (например, Haskell is a standardized pure functional programming language) придает ему гораздо более высокий ранг.

Важно понимать, что ни одна из вышеупомянутых метрик не лучше другой во всех случаях. Оба списка приемлемы в зависимости от намерений ищущего, и для достижения оптимальных результатов для конкретного набора документов и приложения требуется задавать разные веса. Поэкспериментировать с заданием весов для обеих метрик можно, изменив строку `weights` примерно так:

```
weights=[(1.0,self.frequencyscore(rows)),
         (1.5,self.locationscore(rows))]
```

Поиграйте с разными весами и запросами и посмотрите, как будут меняться результаты. Манипулировать расположением в документе сложнее, чем частотой слов, поскольку в документе есть только одно первое слово и от количества его повторений результат не зависит.

Расстояние между словами

Если запрос содержит несколько слов, то часто бывает полезно ранжировать результаты в зависимости от того, насколько близко друг к другу встречаются поисковые слова. Как правило, вводя запрос из нескольких слов, человек хочет найти документы, в которых эти слова концептуально связаны. Это более слабое условие, чем фраза, заключенная в кавычки, когда слова должны встречаться точно в указанном порядке без промежуточных слов. Рассматриваемая метрика допускает изменение порядка и наличие дополнительных слов между поисковыми.

Функция `distancescore` очень похожа на `locationscore`:

```
def distancescore(self,rows):
  # Если есть только одно слово, любой документ выигрывает!
  if len(rows[0])<=2: return dict([(row[0],1.0) for row in rows])

  # Инициализировать словарь большими значениями
  mindistance=dict([(row[0],1000000) for row in rows])

  for row in rows:
    dist=sum([abs(row[i]-row[i-1]) for i in range(2,len(row))])
    if dist<mindistance[row[0]]: mindistance[row[0]]=dist
  return self.normalizescores(mindistance,smallIsBetter=1)
```

Основное отличие в том, что, когда функция в цикле обходит вхождения (строка, выделенная полужирным шрифтом), она вычисляет разность между адресом текущего и предыдущего вхождений. Поскольку в ответ на запрос возвращаются все комбинации расстояний, то гарантированно будет найдено наименьшее полное расстояние.

Можете протестировать метрику на основе расстояния между словами саму по себе, но вообще-то она лучше работает в сочетании с другими метриками. Попробуйте добавить функцию `distancescore` в список `weights` и поварьируйте веса, чтобы посмотреть, как это скажется на результатах различных запросов.

Использование внешних ссылок на сайт

Все обсуждавшиеся до сих пор метрики ранжирования были основаны на содержимом страницы. Хотя многие поисковые машины и по сей день работают таким образом, часто результаты можно улучшить, приняв во внимание, что говорят об этой странице другие, а точнее те сайты, на которых размещена ссылка на нее. Особенно это полезно при

индексировании страниц сомнительного содержания или таких, которые могли быть созданы спамерами, поскольку маловероятно, что на такие страницы есть ссылки с настоящих сайтов.

Созданный в начале этой главы паук уже собирает всю существенную информацию о ссылках, поэтому изменять его не потребуется. В таблице `links` хранятся идентификаторы URL источника и адресата каждой встретившейся ссылки, а таблица `linkwords` связывает слова со ссылками.

Простой подсчет ссылок

Простейший способ работы с внешними ссылками заключается в том, чтобы подсчитать, сколько их ведет на каждую страницу, и использовать результат в качестве метрики. Так обычно оцениваются научные работы; считается, что их значимость тем выше, чем чаще их цитируют. Представленная ниже функция ранжирования создает словарь счетчиков, делая запрос к таблице ссылок для каждого уникального идентификатора URL в списке `rows`, а затем возвращает нормализованный результат:

```
def inboundlinkscore(self,rows):
  uniqueurls=set([row[0] for row in rows])
  inboundcount=dict([(u,self.con.execute( \
    'select count(*) from link where toid=%d' % u).fetchone( )[0]) \
      for u in uniqueurls])
  return self.normalizescores(inboundcount)
```

Очевидно, что при использовании одной лишь этой метрики будут возвращены все страницы, содержащие поисковые слова, упорядоченные по числу внешних ссылок на них. В нашем наборе данных на страницу Programming language о языках программирования ведет гораздо больше ссылок, чем на страницу Python, но, если вы выполняли поиск по слову Python, то, надо полагать, хотели бы увидеть соответствующую страницу первой в списке. Чтобы объединить релевантность с ранжированием по внешним ссылкам, надо использовать последнее в сочетании с одной из рассмотренных выше метрик.

Кроме того, описанный алгоритм трактует все внешние ссылки одинаково, но такой уравнительный подход открывает возможность для манипулирования, поскольку кто угодно может создать несколько сайтов, указывающих на страницу, ранг которой он хочет поднять. Также возможно, что людям более интересны страницы, которые привлекли внимание каких-то популярных сайтов. Далее мы увидим, как придать ссылкам с популярных сайтов больший вес при вычислении ранга страницы.

Алгоритм PageRank

Алгоритм PageRank был придуман основателями компании Google, и вариации этой идеи теперь применяются во всех крупных поисковых

машинах. Этот алгоритм приписывает каждой странице ранг, оценивающий ее значимость. Значимость страницы вычисляется исходя из значимостей ссылающихся на нее страниц и общего количества ссылок, имеющихся на каждой из них.

Теоретически алгоритм PageRank (названный по фамилии одного из его изобретателей Лари Пейджа (Larry Page)) рассчитывает вероятность того, что человек, случайно переходящий по ссылкам, доберется до некоторой страницы. Чем больше ссылок ведет на данную страницу с других популярных страниц, тем выше вероятность, что экспериментатор чисто случайно наткнется на нее. Разумеется, если пользователь будет щелкать по ссылкам бесконечно долго, то в конце концов он посетит каждую страницу, но большинство людей в какой-то момент останавливаются. Чтобы учесть это, в алгоритм PageRank введен *коэффициент затухания* 0,85, означающий, что пользователь продолжит переходить по ссылкам, имеющимся на текущей странице, с вероятностью 0,85.

На рис. 4.3 приведен пример множества страниц и ссылок.

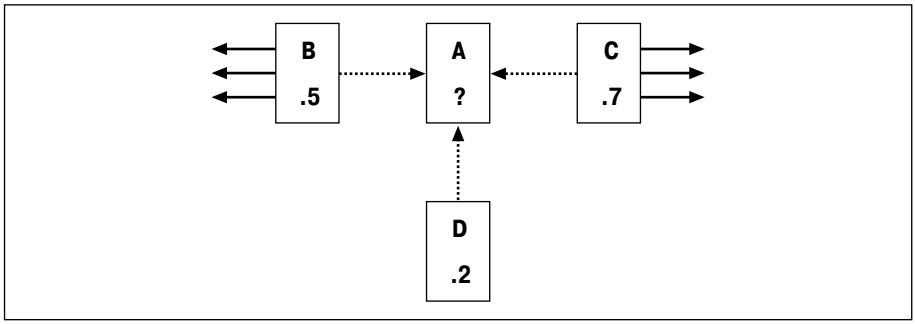


Рис. 4.3. *Вычисление ранга PageRank страницы A*

Каждая из страниц B, C и D ссылается на A, и для них ранг уже вычислен. На странице B есть ссылки еще на три страницы, а на странице C – на четыре. D ссылается только на A. Чтобы вычислить ранг A, берем ранги (PR) каждой ссылающейся на A страницы, делим их на общее число ссылок на этой странице, складываем получившиеся значения, затем умножаем результат на коэффициент затухания 0,85 и добавляем минимальную величину 0,15. Вот как производится вычисление PR(A):

$$PR(A) = 0{,}15 + 0{,}85 \times (PR(B)/ссылки(B) + PR(C)/ссылки(C) + PR(D)/ссылки(D))$$
$$= 0{,}15 + 0{,}85 \times (0{,}5/4 + 0{,}7/5 + 0{,}2/1)$$
$$= 0{,}15 + 0{,}85 \times (0{,}125 + 0{,}14 + 0{,}2)$$
$$= 0{,}15 + 0{,}85 \times 0{,}465$$
$$= 0{,}54525$$

Обратите внимание, что D дает больший вклад в ранг A, чем B или C, хотя ее собственный ранг меньше. Это вызвано тем, что D ссылается только на A и, значит, привносит в результат свой ранг целиком.

Несложно, да? Увы, имеется тут небольшая ловушка – в данном примере для всех страниц, ссылающихся на A, уже вычислен ранг. Но невозможно вычислить ранг страницы, пока неизвестны ранги ссылающихся на нее страниц, а эти ранги можно вычислить, только зная ранги страницы, которые ссылаются на *них*. Так как же вычислить значение PageRank для множества страниц, ранги которых еще неизвестны?

Решение состоит в том, чтобы присвоить всем страницам произвольный начальный ранг (в программе ниже взято значение 1,0, но на самом деле точная величина несущественна) и провести несколько итераций. После каждой итерации ранг каждой страницы будет все ближе к истинному значению PageRank. Количество необходимых итераций зависит от числа страниц, но для того небольшого набора, с которым мы работаем, 20 должно быть достаточно.

Поскольку вычисление значения PageRank занимает много времени, а вычисленный ранг не зависит от запроса, то мы напишем функцию, которая заранее вычисляет ранг каждого URL и сохраняет его в таблице. При каждом вызове эта функция будет пересчитывать ранги всех страниц. Добавьте ее в класс `crawler`:

```
def calculatepagerank(self,iterations=20):
  # стираем текущее содержимое таблицы PageRank
  self.con.execute('drop table if exists pagerank')
  self.con.execute('create table pagerank(urlid primary key,score)')

  # в начальный момент ранг для каждого URL равен 1
  self.con.execute('insert into pagerank select rowid, 1.0 from urllist')
  self.dbcommit( )

  for i in range(iterations):
    print "Итерация %d" % (i)
    for (urlid,) in self.con.execute('select rowid from urllist'):
      pr=0.15

    # В цикле обходим все страницы, ссылающиеся на данную
    for (linker,) in self.con.execute(
    'select distinct fromid from link where toid=%d' % urlid):
      # Находим ранг ссылающейся страницы
      linkingpr=self.con.execute(
      'select score from pagerank where urlid=%d' % linker).fetchone( )[0]

      # Находим общее число ссылок на ссылающейся странице
      linkingcount=self.con.execute(
      'select count(*) from link where fromid=%d' % linker).fetchone( )[0]
      pr+=0.85*(linkingpr/linkingcount)
    self.con.execute(
    'update pagerank set score=%f where urlid=%d' % (pr,urlid))
  self.dbcommit( )
```

Вначале эта функция приписывает каждой странице ранг 1,0. Потом в цикле перебираются все URL и для каждой ссылающейся на него страницы из базы извлекается значение PageRank и общее число ссылок на ней. Строка, выделенная полужирным шрифтом, содержит формулу, применяемую к каждой из внешних ссылок. Эта функция будет работать несколько минут, но запускать ее необходимо только после обновления индекса.

```
>> reload(searchengine)
>> crawler=searchengine.crawler('searchindex.db')
>> crawler.calculatepagerank( )
Итерация 0
Итерация 1
...
```

Если вам интересно, какие страницы из рассматриваемого набора имеют максимальный ранг, можете запросить базу данных напрямую:

```
>> cur=crawler.con.execute('select * from pagerank order by score desc')
>> for i in range(3): print cur.next( )
(438, 2.5285160000000002)
(2, 1.1614640000000001)
(543, 1.064252)
>> e.geturlname(438)
u'http://kiwitobes.com/wiki/Main_Page.html'
```

Наивысший ранг имеет главная страница Main Page, что и неудивительно, поскольку на нее есть ссылки с любой страницы Википедии. Теперь, когда у нас есть таблица рангов, для ее использования достаточно написать функцию, которая будет извлекать ранг и выполнять нормализацию. Добавьте в класс `searcher` следующий метод:

```
def pagerankscore(self,rows):
  pageranks=dict([(row[0],self.con.execute('select score from pagerank where
urlid=%d' % row[0]).fetchone( )[0]) for row in rows])
  maxrank=max(pageranks.values( ))
  normalizedscores=dict([(u,float(l)/maxrank) for (u,l) in \
    pageranks.items( )])
  return normalizedscores
```

Включите этот метод в список `weights`, например, так:

```
weights=[(1.0,self.locationscore(rows)),
         (1.0,self.frequencyscore(rows)),
         (1.0,self.pagerankscore(rows))]
```

Теперь при ранжировании результатов учитывается как содержимое страницы, так и ее ранг. Результаты поиска по фразе Functional programming выглядят даже лучше, чем прежде:

```
2.318146 http://kiwitobes.com/wiki/Functional_programming.html
1.074506 http://kiwitobes.com/wiki/Programming_language.html
0.517633 http://kiwitobes.com/wiki/Categorical_list_of_programming_languages.
html
0.439568 http://kiwitobes.com/wiki/Programming_paradigm.html
0.426817 http://kiwitobes.com/wiki/Lisp_programming_language.html
```

Значение PageRank трудно оценить в полной мере на таком замкнутом, строго контролируемом наборе документов, в котором, скорее всего, почти нет ни вовсе бесполезных страниц, ни страниц, созданных только для привлечения внимания. Но даже в этом случае видно, что PageRank – полезная метрика, позволяющая возвращать более общие страницы, относящиеся к теме запроса.

Использование текста ссылки

Еще один полезный способ ранжирования результатов – использование текста ссылок на страницу при определении степени ее релевантности запросу. Часто удается получить более качественную информацию из того, что сказано в ссылках, ведущих на страницу, чем из самой страницы, поскольку авторы сайтов обычно включают краткое описание того, на что ссылаются.

Метод ранжирования страниц по текстам ссылок требует дополнительного параметра – списка идентификаторов слов, созданного в результате первичной обработки запроса. Добавьте этот метод в класс `searcher`:

```
def linktextscore(self,rows,wordids):
  linkscores=dict([(row[0],0) for row in rows])
  for wordid in wordids:
     cur=self.con.execute('select link.fromid,link.toid from linkwords,link
where wordid=%d and linkwords.linkid=link.rowid' % wordid)
    for (fromid,toid) in cur:
      if toid in linkscores:
     pr=self.con.execute('select score from pagerank where urlid=%d' % fromid).
fetchone( )[0]
        linkscores[toid]+=pr
  maxscore=max(linkscores.values( ))
    normalizedscores=dict([(u,float(l)/maxscore) for (u,l) in linkscores.
items()])
  return normalizedscores
```

Этот код в цикле обходит все слова из списка `wordids` и ищет ссылки, содержащие эти слова. Если страница, на которую ведет ссылка, совпадает с каким-нибудь результатом поиска, то ранг источника ссылки прибавляется к окончательному рангу этой страницы. Страница, на которую ведет много ссылок со значимых страниц, содержащих поисковые слова, получит очень высокий ранг. Для многих найденных страниц вообще не будет ссылок с подходящим текстом, поэтому их ранг окажется равен 0.

Чтобы активировать ранжирование по тексту ссылок, просто добавьте в список `weights` такую строку:

```
(1.0,self.linktextscore(rows,wordids))
```

Не существует стандартных весов для всех рассмотренных метрик, которые были бы одинаково хороши во всех случаях. Даже самые крупные поисковые машины часто изменяют методы ранжирования результатов. Метрики и назначаемые им веса сильно зависят от конкретного приложения.

Обучение на основе действий пользователя

Одно из основных достоинств онлайновых приложений состоит в том, что они все время получают обратную связь в виде поведения пользователей. В случае поисковой машины каждый пользователь тут же сообщает о том, насколько ему понравились результаты поиска, щелкая по одному результату и игнорируя остальные. В этом разделе мы рассмотрим способ регистрации действий пользователя после получения результатов и то, как собранную таком образом информацию можно применить для более качественного ранжирования результатов.

Для этого мы построим *искусственную нейронную сеть*, которую обучим, предлагая ей поисковые слова, предъявленные пользователю результаты обработки запроса и ту ссылку, по которой пользователь перешел. Обученную на многих запросах нейронную сеть можно будет использовать для упорядочения результатов поиска с учетом прошлых действий пользователей.

Проектирование сети отслеживания переходов

Есть много разновидностей нейронных сетей, но все они состоят из множества узлов (нейронов) и связей между ними (синапсов). Сеть, которую мы собираемся построить, называется *многоуровневым перцептроном* (multilayer perceptron – MLP). Такая сеть состоит из нескольких уровней нейронов, первый из которых принимает входные данные – в данном случае поисковые слова, введенные пользователем. Последний уровень возвращает результаты – в нашем примере это список весов найденных URL.

Промежуточных уровней может быть много, но мы ограничимся только одним. Он называется *скрытым уровнем*, так как не взаимодействует напрямую с внешним миром, а реагирует на комбинации входных данных. В данном случае комбинация входных данных – это набор слов, поэтому скрытый уровень можно назвать *уровнем запроса*. На рис. 4.4 изображена структура сети. Все узлы входного уровня связаны с узлами скрытого уровня, а все узлы скрытого уровня – с узлами выходного уровня.

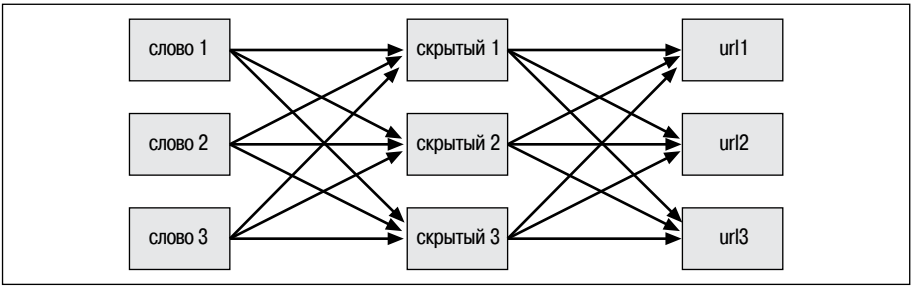


Рис. 4.4. *Проект нейронной сети отслеживания переходов*

Чтобы получить от нейронной сети наилучшие результаты для некоторого запроса, значения входных узлов для указанных в запросе слов устанавливают равными 1. Включаются выходы этих узлов, и они пытаются активировать скрытый слой. В свою очередь, узлы скрытого слоя, получающие достаточно сильный входной сигнал, включают свои выходы и пытаются активировать узлы выходного уровня.

Узлы выходного уровня оказываются активны в разной степени, и по уровню их активности можно судить, как сильно данный URL ассоциирован со словами исходного запроса. На рис. 4.5 показана обработка запроса world bank (Всемирный банк). Сплошными линиями показаны сильные связи, а полужирный текст говорит о том, что узел стал очень активным.

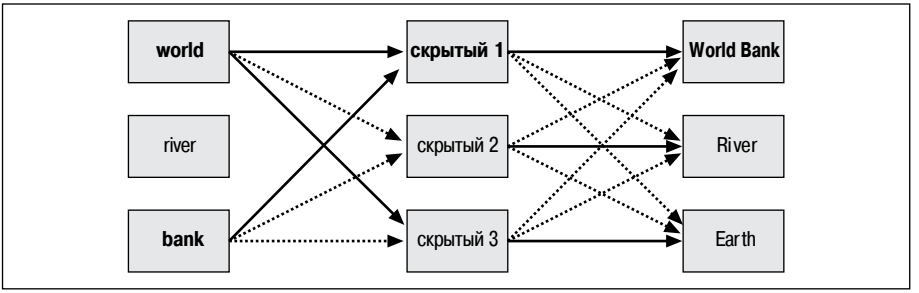


Рис. 4.5. *Ответ нейронной сети на запрос world bank*

Разумеется, для этого необходимо, чтобы сила связей была установлена правильно. Это достигается *обучением* сети всякий раз, как кто-то выполняет поиск и выбирает один из результатов. В сети, изображенной на рис. 4.5, сколько-то человек переходили на сайт Всемирного банка (World Bank) после запроса world bank, и это усилило ассоциацию между данными словами и URL. В этом разделе мы покажем, как сеть обучается с помощью алгоритма обратного распространения `backpropagation`.

Возможно, у вас возник вопрос, зачем нужна такая сложная техника, как нейронные сети? Почему бы просто не запомнить запрос и не подсчитать, сколько раз переходили по найденной ссылке? Но мощь нейронной сети заключается в том, что она может строить осмысленные гипотезы о результатах запросов, которые никогда прежде не видела, исходя из их сходства с другими запросами. Кроме того, нейронные сети полезны в самых разнообразных приложениях, они станут ценным вкладом в наш инструментарий для работы с коллективным разумом.

Подготовка базы данных

Поскольку нейронную сеть необходимо обучать на запросах пользователей, потребуется сохранить представление сети в базе данных. В нашей базе уже есть таблицы слов и URL, поэтому нужна лишь еще одна

таблица для хранения скрытого слоя (которую мы назовем `hiddennode`) и две таблицы для хранения связей (одна – для связей между слоем слов и скрытым слоем, другая – для связей между скрытым и выходным слоем).

Создайте новый файл nn.py и в нем класс `searchnet`:

```
from math import tanh
from pysqlite2 import dbapi2 as sqlite

class searchnet:
  def __init__(self,dbname):
    self.con=sqlite.connect(dbname)

  def __del__(self):
    self.con.close( )

  def maketables(self):
    self.con.execute('create table hiddennode(create_key)')
    self.con.execute('create table wordhidden(fromid,toid,strength)')
    self.con.execute('create table hiddenurl(fromid,toid,strength)')
    self.con.commit( )
```

В данный момент таблицы не имеют никаких индексов, но, если быстродействие окажется недостаточным, их можно будет добавить позже.

Нам потребуется два метода для доступа к базе данных. Первый, `getstrength`, определяет текущую силу связи. Поскольку новые связи создаются только по мере необходимости, этот метод должен возвращать некое специальное значение, если связей еще нет. Для связей между словами и скрытым слоем мы выберем в качестве такого значения –0,2, так что по умолчанию дополнительные слова будут несколько снижать уровень активации скрытого узла. Для связей между скрытым слоем и URL метод по умолчанию будет возвращать значение 0.

```
def getstrength(self,fromid,toid,layer):
  if layer==0: table='wordhidden'
  else: table='hiddenurl'
  res=self.con.execute('select strength from %s where fromid=%d and toid=%d'
% (table,fromid,toid)).fetchone( )
  if res==None:
    if layer==0: return -0.2
    if layer==1: return 0
  return res[0]
```

Еще необходим метод `setstrength`, который выясняет, существует ли уже связь, и создает либо обновляет связь, приписывая ей заданную силу. Он будет использоваться в коде для обучения сети:

```
def setstrength(self,fromid,toid,layer,strength):
  if layer==0: table='wordhidden'
  else: table='hiddenurl'
  res=self.con.execute('select rowid from %s where fromid=%d and toid=%d' %
(table,fromid,toid)).fetchone( )
```

```
    if res==None:
      self.con.execute('insert into %s (fromid,toid,strength) values (%d,%d,%f)'
% (table,fromid,toid,strength))
    else:
      rowid=res[0]
      self.con.execute('update %s set strength=%f where rowid=%d' %
(table,strength,rowid))
```

Обычно при построении нейронной сети все узлы создаются заранее. Можно было бы предварительно создать гигантскую сеть с тысячами узлов в скрытом слое и уже готовыми связями, но в данном случае проще и быстрее создавать скрытые узлы, когда в них возникает надобность.

Следующая функция создает новый скрытый узел всякий раз, как ей передается не встречавшаяся ранее комбинация слов. Затем она создает связи с весами по умолчанию между этими словами и скрытым узлом и между узлом запроса и URL, который этот запрос возвращает.

```
def generatehiddennode(self,wordids,urls):
  if len(wordids)>3: return None
  # Проверить, создавали ли мы уже узел для данного набора слов
  createkey='_'.join(sorted([str(wi) for wi in wordids]))
  res=self.con.execute(
  "select rowid from hiddennode where create_key='%s'" % createkey).
fetchone( )

  # Если нет, создадим сейчас
  if res==None:
    cur=self.con.execute(
    "insert into hiddennode (create_key) values ('%s')" % createkey)
    hiddenid=cur.lastrowid
    # Задать веса по умолчанию
    for wordid in wordids:
      self.setstrength(wordid,hiddenid,0,1.0/len(wordids))
    for urlid in urls:
      self.setstrength(hiddenid,urlid,1,0.1)
    self.con.commit( )
```

В интерпретаторе Python создайте базу данных и сгенерируйте скрытый узел для какого-нибудь слова и идентификатора URL:

```
>> import nn
>> mynet=nn.searchnet('nn.db')
>> mynet.maketables( )
>> wWorld,wRiver,wBank =101,102,103
>> uWorldBank,uRiver,uEarth =201,202,203
>> mynet.generatehiddennode([wWorld,wBank],[uWorldBank,uRiver,uEarth])
>> for c in mynet.con.execute('select * from wordhidden'): print c
(101, 1, 0.5)
(103, 1, 0.5)
>> for c in mynet.con.execute('select * from hiddenurl'): print c
(1, 201, 0.1)
(1, 202, 0.1)
...
```

Только что вы создали новый узел в скрытом слое и связи с ним со значениями по умолчанию. Первоначально функция будет отвечать, когда слова world и bank встречаются вместе, но со временем эти связи могут ослабнуть.

Прямой проход

Теперь все готово для написания функций, которые принимают на входе слова, активируют связи в сети и формируют выходные сигналы для URL.

Для начала выберем функцию, которая описывает, с какой силой каждый узел должен реагировать на входной сигнал. В нашем примере сети мы воспользуемся для этого функцией *гиперболического тангенса* (`tanh`), график которой изображен на рис. 4.6.

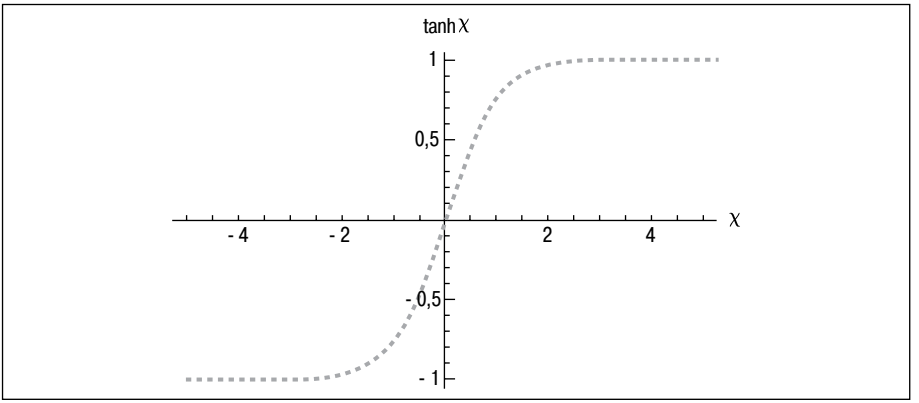


Рис. 4.6. *Функция tanh*

По оси *X* откладывается входной сигнал, подаваемый узлу. В окрестности 0 выходной сигнал быстро нарастает. Когда сила входного сигнала равна 2, выходной сигнал почти равен 1 и дальше уже почти не растет. Функции такого типа называются *сигмоидными*, они имеют форму буквы S. Для вычисления силы выходного сигнала нейронов в нейронных сетях почти всегда используются сигмоидные функции.

Прежде чем вызывать алгоритм `feedforward`, наш класс должен прочитать из базы информацию об узлах и связях и построить в памяти часть сети, релевантную конкретному запросу. Первым делом мы напишем функцию, которая ищет все узлы из скрытого слоя, релевантные запросу; в нашем случае это узлы, связанные с любым из слов запроса или с каким-нибудь URL, принадлежащим множеству результатов поиска. Так как никакие другие узлы не участвуют в определении выхода или в обучении сети, то и включать их необязательно.

```
def getallhiddenids(self,wordids,urlids):
  l1={}
  for wordid in wordids:
```

```
    cur=self.con.execute(
    'select toid from wordhidden where fromid=%d' % wordid)
    for row in cur: l1[row[0]]=1
  for urlid in urlids:
    cur=self.con.execute(
    'select fromid from hiddenurl where toid=%d' % urlid)
    for row in cur: l1[row[0]]=1
  return l1.keys( )
```

Нам также понадобится метод для конструирования релевантной сети с текущими весами из базы данных. Эта функция инициализирует различные переменные экземпляра класса: список слов, относящиеся к запросу узлы и URL, уровень выходного сигнала для каждого узла и веса всех связей между узлами. Веса считываются из базы данных с помощью ранее разработанных функций.

```
def setupnetwork(self,wordids,urlids):
  # списки значений
  self.wordids=wordids
  self.hiddenids=self.getallhiddenids(wordids,urlids)
  self.urlids=urlids

  # выходные сигналы узлов
  self.ai = [1.0]*len(self.wordids)
  self.ah = [1.0]*len(self.hiddenids)
  self.ao = [1.0]*len(self.urlids)

  # создаем матрицу весов
  self.wi = [[self.getstrength(wordid,hiddenid,0)
              for hiddenid in self.hiddenids]
             for wordid in self.wordids]
  self.wo = [[self.getstrength(hiddenid,urlid,1)
              for urlid in self.urlids]
             for hiddenid in self.hiddenids]
```

Вот теперь все готово для реализации алгоритма `feedforward`. Он принимает список входных сигналов, пропускает их через сеть и возвращает выходные сигналы от всех узлов на выходном уровне. Поскольку в данном случае мы сконструировали сеть только для слов, входящих в запрос, то выходной сигнал от всех входных узлов равен 1:

```
def feedforward(self):
  # единственные входные сигналы - слова из запроса
  for i in range(len(self.wordids)):
    self.ai[i] = 1.0

  # возбуждение скрытых узлов
  for j in range(len(self.hiddenids)):
    sum = 0.0
    for i in range(len(self.wordids)):
      sum = sum + self.ai[i] * self.wi[i][j]
    self.ah[j] = tanh(sum)

  # возбуждение выходных узлов
  for k in range(len(self.urlids)):
```

```
        sum = 0.0
        for j in range(len(self.hiddenids)):
          sum = sum + self.ah[j] * self.wo[j][k]
        self.ao[k] = tanh(sum)

    return self.ao[:]
```

Алгоритм `feedforward` в цикле обходит все узлы скрытого слоя и для каждого из них вычисляет сумму величин выходных сигналов от узлов входного слоя, помноженных на вес соответствующей связи. Выходной сигнал каждого скрытого узла – это результат применения функции `tanh` к взвешенной сумме входных сигналов. Этот сигнал передается на выходной уровень. Выходной уровень делает то же самое – умножает полученные от предыдущего уровня сигналы на веса связей и применяет функцию `tanh` для получения окончательного результата. Сеть можно легко обобщить, введя дополнительные уровни, которые будут преобразовывать выходные сигналы от предыдущего уровня во входные сигналы следующему.

Теперь можно написать небольшую функцию, которая подготовит сеть и вызовет метод `feedforward` для получения реакции на заданный набор слов и URL:

```
def getresult(self,wordids,urlids):
  self.setupnetwork(wordids,urlids)
  return self.feedforward( )
```

Протестируем эту сеть:

```
>> reload(nn)
>> mynet=nn.searchnet('nn.db')
>> mynet.getresult([wWorld,wBank],[uWorldBank,uRiver,uEarth])
[0.76,0.76,0.76]
```

Числа в полученном списке обозначают релевантность поданных на вход URL. Неудивительно, что для всех URL нейронная сеть выдает один и тот же ответ, ведь она еще не прошла обучения.

Обучение с обратным распространением

А вот теперь начинаются более интересные вещи. Сеть принимает входные сигналы и генерирует выходные, но, поскольку ее не научили, какой результат считать хорошим, выдаваемые ею ответы практически бесполезны. Сейчас мы обучим сеть, предъявив ей реальные примеры запросов, найденных результатов и действий пользователей.

Чтобы это сделать, нам необходим алгоритм, который будет изменять веса связей между узлами, так чтобы сеть поняла, как выглядит правильный ответ. Веса следует подстраивать постепенно, поскольку нельзя предполагать, что ответ, выбранный одним пользователем, устроит и всех остальных. Описанный ниже алгоритм называется *обратным распространением*, поскольку в процессе подстройки весов он продвигается по сети в обратном направлении.

Во время обучения сети вы в каждый момент знаете желаемый выходной сигнал от каждого узла в выходном слое. В данном случае необходимо получить 1, если пользователь перешел по ссылке, и 0 в противном случае. Изменить выходной сигнал узла можно только одним способом – изменить сигнал, поданный ему на вход.

Чтобы определить, насколько следует изменить суммарный входной сигнал, алгоритм обучения должен знать наклон функции `tanh` для текущего уровня выходного сигнала. В средней точке графика, когда выходной сигнал равен 0,0, функция изменяется очень круто, поэтому небольшое изменение входного сигнала приводит к большому изменению выходного. По мере того как уровень выходного сигнала приближается к –1 или к 1, изменение входного сигнала меньше сказывается на выходном. Наклон нашей функции для любого значения выходного сигнала вычисляется с помощью следующей функции, которую нужно добавить в начало файла nn.py:

```
def dtanh(y):
  return 1.0-y*y
```

Прежде чем запускать метод обратного распространения, следует вызвать функцию `feedforward`, чтобы сохранить в переменных экземпляра текущий уровень выходного сигнала от каждого узла. Затем алгоритм обратного распространения выполняет следующие шаги.

Для каждого узла из выходного слоя необходимо:

1. Вычислить разность между текущим и желательным уровнем выходного сигнала.
2. С помощью функции `dtanh` определить, насколько должен измениться суммарный входной сигнал для этого узла.
3. Изменить вес каждой входящей связи пропорционально ее текущему весу и скорости обучения.

Для каждого узла в скрытом слое необходимо:

1. Изменить выходной сигнал узла на сумму весов каждой выходной связи, умноженных на величину требуемого изменения выходного сигнала конечного узла этой связи.
2. С помощью функции `dtanh` вычислить, насколько должен измениться суммарный входной сигнал для этого узла.
3. Изменить вес каждой входящей связи пропорционально ее текущему весу и скорости обучения.

При реализации этого алгоритма мы вычисляем все поправки заранее, а затем подстраиваем веса, поскольку все расчеты основываются на знании текущих, а не измененных весов. Ниже приведен код:

```
def backPropagate(self, targets, N=0.5):
  # вычислить поправки для выходного слоя
  output_deltas = [0.0] * len(self.urlids)
  for k in range(len(self.urlids)):
    error = targets[k]-self.ao[k]
    output_deltas[k] = dtanh(self.ao[k]) * error
```

```
  # вычислить поправки для скрытого слоя
  hidden_deltas = [0.0] * len(self.hiddenids)
  for j in range(len(self.hiddenids)):
    error = 0.0
    for k in range(len(self.urlids)):
      error = error + output_deltas[k]*self.wo[j][k]
    hidden_deltas[j] = dtanh(self.ah[j]) * error

  # обновить веса связей между узлами скрытого и выходного слоя
  for j in range(len(self.hiddenids)):
    for k in range(len(self.urlids)):
      change = output_deltas[k]*self.ah[j]
      self.wo[j][k] = self.wo[j][k] + N*change

  # обновить веса связей между узлами входного и скрытого слоя
  for i in range(len(self.wordids)):
    for j in range(len(self.hiddenids)):
      change = hidden_deltas[j]*self.ai[i]
      self.wi[i][j] = self.wi[i][j] + N*change
```

Осталось только написать простой метод, который подготовит сеть, вызовет алгоритм `feedforward` и запустит обратное распространение. Этот метод принимает в качестве параметров списки `wordids`, `urlids` и выбранный URL:

```
def trainquery(self,wordids,urlids,selectedurl):
  # сгенерировать скрытый узел, если необходимо
  self.generatehiddennode(wordids,urlids)

  self.setupnetwork(wordids,urlids)
  self.feedforward( )
  targets=[0.0]*len(urlids)
  targets[urlids.index(selectedurl)]=1.0
  error = self.backPropagate(targets)
  self.updatedatabase( )
```

Для сохранения результатов понадобится метод записи в базу данных новых весов, которые хранятся в переменных экземпляра `wi` и `wo`:

```
def updatedatabase(self):
  # записать в базу данных
  for i in range(len(self.wordids)):
    for j in range(len(self.hiddenids)):
      self.setstrength(self.wordids[i],self. hiddenids[j],0,self.wi[i][j])
  for j in range(len(self.hiddenids)):
    for k in range(len(self.urlids)):
      self.setstrength(self.hiddenids[j],self.urlids[k],1,self.wo[j][k])
  self.con.commit( )
```

Теперь можно прогнать простой тест, подав на вход тот же запрос, что и раньше, и посмотреть, как сеть отреагировала на обучение:

```
>> reload(nn)
>> mynet=nn.searchnet('nn.db')
>> mynet.trainquery([wWorld,wBank],[uWorldBank,uRiver,uEarth],uWorldBank)
>> mynet.getresult([wWorld,wBank],[uWorldBank,uRiver,uEarth])
[0.335,0.055,0.055]
```

После того как сеть узнала о выборе пользователя, выходной сигнал для URL Всемирного банка увеличился, а сигналы для остальных URL уменьшились. Чем больше пользователей сделают такой же выбор, тем заметнее станет разница.

Проверка результатов обучения

Выше вы видели, как обучение на одном результате повышает уровень выходного сигнала для этого результата. Это хотя и полезно, но не показывает, на что в действительности способны нейронные сети, – как они могут делать выводы о входных данных, которых никогда прежде не видели. Введите следующие команды в интерактивном сеансе:

```
>> allurls=[uWorldBank,uRiver,uEarth]
>> for i in range(30):
...     mynet.trainquery([wWorld,wBank],allurls,uWorldBank)
...     mynet.trainquery([wRiver,wBank],allurls,uRiver)
...     mynet.trainquery([wWorld],allurls,uEarth)
...
>> mynet.getresult([wWorld,wBank],allurls)
[0.861, 0.011, 0.016]
>> mynet.getresult([wRiver,wBank],allurls)
[-0.030, 0.883, 0.006]
>> mynet.getresult([wBank],allurls)
[0.865, 0.001, -0.85]
```

Хотя сеть никогда раньше не видела запроса bank, она выдает правильную гипотезу. Более того, URL Всемирного банка получает гораздо более высокий ранг, чем URL River, хотя в обучающих примерах слово bank ассоциировалось со словом river так же часто, как с World Bank. Сеть не только поняла, какие URL с какими запросами ассоциируются, но и обучилась тому, какие слова важны в конкретном запросе. Достичь этого путем простого запоминания пары запрос–URL было бы невозможно.

Подключение к поисковой машине

Метод `query` класса `searcher` получает списки идентификаторов URL и идентификаторов слов в процессе создания и распечатки результатов. Можно заставить метод возвращать эти результаты, добавив в конец такую строку (файл searchengine.py):

```
return wordids,[r[1] for r in rankedscores[0:10]]
```

Эти списки можно напрямую передать методу `trainquery` из класса `searchnet`.

Способ получения информации о том, какие результаты больше всего понравились пользователям, зависит от проекта приложения. Можно включить промежуточную страницу, на которую будет отправлен пользователя при щелчке по любой ссылке. Она могла бы вызвать метод `trainquery`, а затем переадресовать пользователя на выбранный им

документ. Альтернативно можно было бы дать пользователям возможность проголосовать за релевантность результатов поиска, объяснив, что это поможет улучшить алгоритм.

Последний шаг при создании искусственной нейронной сети – это создание в классе `searcher` еще одного метода, который будет взвешивать результаты. Соответствующая функция похожа на все остальные весовые функции. Прежде всего следует импортировать класс нейронной сети в файл searchengine.py:

```
import nn
mynet=nn.searchnet('nn.db')
```

Затем добавьте в класс `searcher` такой метод:

```
def nnscore(self,rows,wordids):
  # Получить уникальные идентификаторы URL в виде упорядоченного списка
  urlids=[urlid for urlid in set([row[0] for row in rows])]
  nnres=mynet.getresult(wordids,urlids)
  scores=dict([(urlids[i],nnres[i]) for i in range(len(urlids))])
  return self.normalizescores(scores)
```

Включите его в список `weights` и поэкспериментируйте с заданием разных весов. На практике следует повременить с привлечением этого метода к реальному ранжированию до тех пор, пока сеть не обучится на большом числе примеров.

В этой главе мы рассмотрели целый ряд методик разработки поисковых машин, но это капля в море по сравнению с тем, что еще возможно. В упражнениях приведены кое-какие дополнительные идеи. Мы не уделяли внимание вопросам производительности – для этого потребовалось бы проиндексировать миллионы страниц, – но построенная система достаточно быстро работает на множестве объемом 100 000 страниц, а этого достаточно для новостного сайта или корпоративной интрасети.

Упражнения

1. *Выделение слов*. Сейчас в методе `separatewords` разделителями считаются все символы, кроме букв и цифр, то есть такие слова, как C++, $20, Ph.D или 617-555-1212, будут индексироваться неправильно. Как можно улучшить алгоритм выделения слов? Достаточно ли ограничиться только символами пробела? Напишите функцию, реализующую ваш алгоритм.
2. *Булевские операции*. Многие поисковые машины поддерживают булевские запросы, то есть допускают конструкции типа python OR perl. Чтобы реализовать OR-запрос, можно выполнить оба запроса независимо, а затем объединить результаты. А как насчет такого запроса: python AND (program OR code)? Модифицируйте методы выполнения запроса так, чтобы они поддерживали основные булевские операции.

3. *Точное соответствие*. Многие поисковые машины поддерживают запросы на «точное соответствие», когда страница считается удовлетворяющей запросу, если искомые слова следуют точно в указанном порядке без промежуточных слов. Напишите новую версию функции `getrows`, которая возвращает лишь результаты, являющиеся точным соответствием. (Подсказка: можно воспользоваться операцией вычитания в SQL, чтобы найти разность между адресами вхождений слов.)
4. *Поиск с учетом длины документа*. Иногда релевантность страницы в конкретном приложении или для конкретного пользователя зависит от ее длины. Пользователю может быть интересна объемная статья по сложному предмету или краткая справка по работе с какой-то командной утилитой. Напишите весовую функцию, которая отдает предпочтение длинным или коротким документам в зависимости от указанных параметров.
5. *Учет частоты слов*. Метрика «количество слов» отдает предпочтение длинным документам, поскольку они содержат больше слов и, следовательно, искомых слов тоже встретится много. Напишите новую метрику, которая вычисляет частоту вхождений, то есть долю искомых слов относительно общего их числа.
6. *Поиск во внешних ссылках*. Вы можете ранжировать результаты по тексту внешних ссылок, но, чтобы документ вошел во множество результатов, на странице должны присутствовать поисковые слова. Однако иногда наиболее релевантные страницы вообще не содержат поисковых слов в тексте, зато существует множество ведущих на них ссылок, в тексте которых есть поисковые слова. Так часто бывает со ссылками на изображения. Модифицируйте программу поиска так, чтобы она включала в состав результатов страницы, на которые ведут ссылки, содержащие какие-то поисковые слова.
7. *Различные режимы обучения*. При обучении нейронной сети мы задаем в качестве уровня выходного сигнала 0 для тех URL, на которые пользователь не переходил, и 1 для URL, на который он перешел. Измените обучающую функцию, так чтобы она могла работать в приложении, где пользователю разрешено выставлять результатам поиска оценки от 1 до 5.
8. *Дополнительные уровни*. Сейчас в нейронной сети есть только один скрытый уровень. Измените класс так, чтобы он поддерживал произвольное число уровней, задаваемое на этапе инициализации.

5

Оптимизация

В этой главе мы рассмотрим, как решать задачи со множеством участников, применяя технику *стохастической оптимизации*. Обычно методы оптимизации используются для задач, имеющих множество возможных решений, которые зависят от многих переменных, а результат существенно зависит от сочетания этих переменных. У методов оптимизации весьма широкая область применения: в физике они используются для изучения молекулярной динамики, в биологии – для прогнозирования белковых структур, а в информатике – для определения времени работы алгоритма в худшем случае. В НАСА методы оптимизации применяют даже для проектирования антенн с наилучшими эксплуатационными характеристиками. Выглядят они так, как ни один человек не мог бы вообразить.

По существу, оптимизация сводится к поиску наилучшего решения задачи путем апробирования различных решений и сравнения их между собой для оценки качества. Обычно оптимизация применяется в тех случаях, когда число решений слишком велико и перебрать их все невозможно. Простейший, но и самый неэффективный способ поиска решения – попробовать несколько тысяч случайных гипотез и отобрать из них наилучшую. В этой главе мы рассмотрим более эффективные методы, которые пытаются улучшить имеющееся решение путем его целенаправленной модификации.

Наш первый пример относится к планированию групповых путешествий. Всякий, кто пытался планировать поездку группы людей или даже одного человека, понимает, что нужно учитывать множество исходных данных, например: каким рейсом должен лететь каждый человек, сколько нужно арендовать машин и от какого аэропорта удобнее всего добираться. Также следует принять во внимание ряд выходных

переменных: полная стоимость, время ожидания в аэропортах и непроизводительно потраченное время. Поскольку не существует простой формулы, с помощью которой выходные переменные можно было бы вычислить по исходным данным, то поиск наилучшего решения естественно оказывается задачей на оптимизацию.

Далее в этой главе мы продемонстрируем гибкость оптимизации на примере двух совершенно разных задач: как распределить ограниченные ресурсы с учетом предпочтений людей и как визуализировать социальную сеть с минимальным числом пересечений. К концу главы вы сможете сами распознавать задачи, которые можно решить методами оптимизации.

Групповые путешествия

Планирование путешествия группы людей (в данном примере семейства Глассов), которые, отправляясь из разных мест, должны прибыть в одно и то же место, всегда вызывало сложности и представляет собой интересную оптимизационную задачу. Для начала создайте новый файл optimization.py и вставьте в него такой код:

```
import time
import random
import math

people = [('Seymour','BOS'),
          ('Franny','DAL'),
          ('Zooey','CAK'),
          ('Walt','MIA'),
          ('Buddy','ORD'),
          ('Les','OMA')]

# Место назначения - аэропорт Ла Гардиа в Нью-Йорке
destination='LGA'
```

Члены семьи живут в разных концах страны и хотят встретиться в Нью-Йорке. Все они должны вылететь в один день и в один день улететь и при этом в целях экономии хотели бы уехать из аэропорта и приехать в него на одной арендованной машине. Ежедневно в Нью-Йорк из мест проживания любого члена семьи отправляются десятки рейсов, все в разное время. Цена билета и время в пути для каждого рейса разные.

Пример файла с данными о рейсах можно скачать со страницы *http://kiwitobes.com/optimize/schedule.txt*.

Для каждого рейса в файле через запятую указаны аэропорт отправления, аэропорт назначения, время отправления, время прибытия и цена билета:

```
LGA,MIA,20:27,23:42,169
MIA,LGA,19:53,22:21,173
LGA,BOS,6:39,8:09,86
```

```
BOS,LGA,6:17,8:26,89
LGA,BOS,8:23,10:28,149
```

Загрузим эти данные в словарь, для которого ключом будет аэропорт отправления и аэропорт назначения, а значением – список, содержащий детальную информацию о рейсах. Добавьте следующий код в файл optimization.py:

```
flights={}
#
for line in file('schedule.txt'):
  origin,dest,depart,arrive,price=line.strip( ).split(',')
  flights.setdefault((origin,dest),[])

# Добавить информацию о возможном рейсе в список
flights[(origin,dest)].append((depart,arrive,int(price)))
```

Сейчас будет полезно написать служебную функцию `getminutes`, которая вычисляет смещение данного момента времени от начала суток в минутах. Так нам будет удобнее вычислять время в пути и время ожидания. Добавьте эту функцию в файл optimization.py:

```
def getminutes(t):
  x=time.strptime(t,'%H:%M')
  return x[3]*60+x[4]
```

Задача состоит в том, чтобы решить, каким рейсом должен лететь каждый член семьи. Разумеется, следует минимизировать общую стоимость, но есть и много других факторов, которые следует учитывать при поиске оптимального решения, например общее время ожидания в аэропорту или общее время в пути. Эти факторы мы подробнее обсудим чуть позже.

Представление решений

Для подобных задач необходимо определиться со способом представления потенциальных решений. Функции оптимизации, с которыми вы вскоре ознакомитесь, достаточно общие и применимы к различным задачам, поэтому так важно выбрать простое представление, которое не было бы привязано к конкретной задаче о групповом путешествии. Очень часто для этой цели выбирают список чисел. Каждое число обозначает рейс, которым решил лететь участник группы, причем 0 – это первый рейс в течение суток, 1 – второй и т. д. Так как каждый человек должен лететь туда и обратно, то длина списка в два раза больше количества людей.

Например, в решении, представленном списком

```
[1,4,3,2,7,3,6,3,2,4,5,3]
```

Сеймур (Seymour) летит вторым рейсом из Бостона в Нью-Йорк и пятым рейсом из Нью-Йорка в Бостон, а Фрэнни (Franny) – четвертым рейсом из Далласа в Нью-Йорк и третьим рейсом обратно.

Поскольку интерпретировать решение, имея перед собой лишь список чисел, трудно, нам понадобится функция, которая печатает выбранные рейсы в виде красиво отформатированной таблицы. Добавьте ее в файл optimization.py:

```
def printschedule(r):
  for d in range(len(r)/2):
    name=people[d][0]
    origin=people[d][1]
    out=flights[(origin,destination)][r[d]]
    ret=flights[(destination,origin)][r[d+1]]
    print '%10s%10s %5s-%5s $%3s %5s-%5s $%3s' % (name,origin,
                                                  out[0],out[1],out[2],
                                                  ret[0],ret[1],ret[2])
```

В каждой строке эта функция печатает имя человека, аэропорт назначения, время вылета, время прилета и цены на билеты туда и обратно. Протестируйте ее в интерактивном сеансе:

```
>>> import optimization
>>> s=[1,4,3,2,7,3,6,3,2,4,5,3]
>>> optimization.printschedule(s)
Seymour    Boston 12:34-15:02 $109 12:08-14:05 $142
Franny     Dallas 12:19-15:25 $342 9:49-13:51  $229
Zooey       Akron 9:15-12:14  $247 15:50-18:45 $243
Walt        Miami 15:34-18:11 $326 14:08-16:09 $232
Buddy     Chicago 14:22-16:32 $126 15:04-17:23 $189
Les         Omaha 15:03-16:42 $135 6:19- 8:13  $239
```

Даже если не обращать внимания на цены, в таком расписании есть ряд неувязок. В частности, раз все члены семьи хотят уехать из аэропорта и приехать в него вместе, то все они должны прибыть к 6 часам утра, когда отправляется рейс Леса (Les), хотя некоторые смогут улететь только в 4 часа дня. Чтобы найти оптимальное решение, программа должна взвесить свойства различных расписаний и решить, какое из них наилучшее.

Целевая функция

Ключом к решению любой задачи оптимизации является *целевая функция*, и именно ее обычно труднее всего отыскать. Цель оптимизационного алгоритма состоит в том, чтобы найти такой набор входных переменных – в данном случае рейсов, – который минимизирует целевую функцию. Поэтому целевая функция должна возвращать значение, показывающее, насколько данное решение неудовлетворительно. Не существует никакого определенного масштаба *неудовлетворительности*; требуется лишь, чтобы возвращаемое значение было тем больше, чем хуже решение.

Часто, когда переменных много, бывает трудно сформулировать критерий, хорошее получилось решение или плохое. Рассмотрим несколько параметров, которые можно измерить в примере с групповым путешествием:

Цена

Полная стоимость всех билетов или, возможно, среднее значение, взвешенное с учетом финансовых возможностей.

Время в пути

Суммарное время, проведенное всеми членами семьи в полете.

Время ожидания

Время, проведенное в аэропорту в ожидании прибытия остальных членов группы.

Время вылета

Если самолет вылетает рано утром, это может увеличивать общую стоимость из-за того, что путешественники не выспятся.

Время аренды автомобилей

Если группа арендует машину, то вернуть ее следует до того часа, когда она была арендована, иначе придется платить за лишний день.

Не так уж трудно представить себе дополнительные факторы, которые могут сделать путешествие более или менее приятным. Всякий раз, сталкиваясь с задачей отыскания оптимального решения сложной проблемы, приходится определять, какие факторы важны. Это трудно, но, сделав выбор, вы можете применить изложенные в этой главе алгоритмы оптимизации почти к любой задаче с минимальными изменениями.

Определившись с тем, какие переменные влияют на стоимость, нужно решить, как из них составить одно число. В нашем случае можно, например, выразить в деньгах время в пути или время ожидания в аэропорту. Скажем, каждая минута в воздухе эквивалентна $1 (иначе говоря, можно потратить лишние $90 на прямой рейс, экономящий полтора часа), а каждая минута ожидания в аэропорту эквивалентна $0,50. Можно было бы также приплюсовать стоимость лишнего дня аренды машины, если для всех имеет смысл вернуться в аэропорт к более позднему часу.

Определенная ниже функция `getcost` принимает во внимание полную стоимость поездки и общее время ожидания в аэропорту всеми членами семьи. Кроме того, она добавляет штраф $50, если машина возвращена в более поздний час, чем арендована. Добавьте эту функцию в файл optimization.py и, если хотите, включите в нее дополнительные факторы или измените соотношение между временем и деньгами.

```
def schedulecost(sol):
  totalprice=0
  latestarrival=0
  earliestdep=24*60

  for d in range(len(sol)/2):
    # Получить список прибывающих и убывающих рейсов
    origin=people[d][1]
```

```
    outbound=flights[(origin,destination)][int(sol[d])]
    returnf=flights[(destination,origin)][int(sol[d+1])]

    # Полная цена равна сумме цен на билет туда и обратно
    totalprice+=outbound[2]
    totalprice+=returnf[2]

    # Находим самый поздний прилет и самый ранний вылет
    if latestarrival<getminutes(outbound[1]):
      latestarrival=getminutes(outbound[1])
    if earliestdep>getminutes(returnf[0]): earliestdep=getminutes(returnf[0])

  # Все должны ждать в аэропорту прибытия последнего участника группы.
  # Обратно все прибывают одновременно и должны ждать свои рейсы.
  totalwait=0
  for d in range(len(sol)/2):
    origin=people[d][1]
    outbound=flights[(origin,destination)][int(sol[d])]
    returnf=flights[(destination,origin)][int(sol[d+1])]
    totalwait+=latestarrival-getminutes(outbound[1])
    totalwait+=getminutes(returnf[0])-earliestdep

  # Для этого решения требуется оплачивать дополнительный день аренды?
  # Если да, это обойдется в лишние $50!
  if latestarrival>earliestdep: totalprice+=50

  return totalprice+totalwait
```

Логика этой функции очень проста, но суть вопроса она отражает. Улучшить ее можно несколькими способами. Так, в текущей версии предполагается, что все члены семьи уезжают из аэропорта вместе, когда прибывает самый последний, а возвращаются в аэропорт к моменту вылета самого раннего рейса. Можно поступить по-другому: если человеку приходится ждать два часа или дольше, то он арендует отдельную машину, а цены и время ожидания соответственно корректируются.

Протестируйте эту функцию в интерактивном сеансе:

```
>>> reload(optimization)
>>> optimization.schedulecost(s)
5285
```

После того как целевая функция определена, мы должны минимизировать ее, выбрав подходящие значения входных переменных. Теоретически можно испробовать все возможные комбинации, но в этом примере есть 16 рейсов и для каждого имеется девять вариантов, что в итоге дает 916 (около 300 миллиардов) комбинаций. Проверив их все, мы гарантированно найдем оптимальное решение, но на большинстве компьютеров на этой уйдет слишком много времени.

Случайный поиск

Случайный поиск – не самый лучший метод оптимизации, но он позволит нам ясно понять, чего пытаются достичь все алгоритмы, а также послужит эталоном, с которым можно будет сравнивать другие алгоритмы.

Соответствующая функция принимает два параметра. `Domain` – это список пар, определяющих минимальное и максимальное значения каждой переменной. Длина решения совпадает с длиной этого списка. В нашем примере для каждого человека имеется девять рейсов туда и девять обратно, поэтому `domain` состоит из пары (0,8), повторенной дважды для каждого человека.

Второй параметр, `costf`, – это целевая функция; в нашем примере в этом качестве используется `schedulecost`. Она передается в виде параметра, чтобы алгоритм можно было использовать повторно и для других задач оптимизации. Алгоритм случайным образом генерирует 1000 гипотез и для каждой вызывает функцию `costf`. Возвращается наилучшая гипотеза (с минимальной стоимостью). Добавьте в файл optimization.py следующую функцию:

```
def randomoptimize(domain,costf):
  best=999999999
  bestr=None
  for i in range(1000):
    # Выбрать случайное решение
    r=[random.randint(domain[i][0],domain[i][1])
    for i in range(len(domain))]
      # Вычислить его стоимость
      cost=costf(r)

      # Сравнить со стоимостью наилучшего найденного к этому моменту решения
      if cost<best:
        best=cost
        bestr=r
  return r
```

Разумеется, 1000 гипотез – очень малая доля от общего числа возможностей. Однако в этом примере хороших (если не оптимальных) решений много, поэтому, сделав тысячу попыток, есть шанс наткнуться на что-нибудь приемлемое. Запустите функцию в интерактивном сеансе:

```
>>> reload(optimization)
>>> domain=[(0,8)]*(len(optimization.people)*2)
>>> s=optimization.randomoptimize(domain,optimization.schedulecost)
>>> optimization.schedulecost(s)
3328
>>> optimization.printschedule(s)
Seymour Boston 12:34-15:02 $109 12:08-14:05 $142
Franny Dallas 12:19-15:25 $342 9:49-13:51 $229
```

```
Zooey Akron 9:15-12:14 $247 15:50-18:45 $243
Walt Miami 15:34-18:11 $326 14:08-16:09 $232
Buddy Chicago 14:22-16:32 $126 15:04-17:23 $189
Les Omaha 15:03-16:42 $135 6:19-8:13 $239
```

Благодаря элементу случайности ваши результаты будут отличаться от представленных выше. Найденное решение не слишком хорошее, потому что Зуи (Zooey) придется ждать в аэропорту шесть часов, пока не прилетит Уолт (Walt), но могло быть гораздо хуже. Попробуйте выполнить эту функцию несколько раз и посмотрите, сильно ли будет меняться стоимость. А можете увеличить число итераций до 10 000 и полюбопытствовать, удастся ли улучшить результаты таким способом.

Алгоритм спуска с горы

Случайное апробирование решений очень неэффективно, потому что пренебрегает выгодами, которые можно получить от анализа уже найденных хороших решений. В нашем примере можно предположить, что расписание с низкой полной стоимостью похоже на другие расписания с низкой стоимостью. Но алгоритм случайной оптимизации беспорядочно прыгает с места на место и не пытается автоматически просмотреть похожие расписания, чтобы найти среди них хорошие.

Альтернативный метод случайного поиска называется *алгоритмом спуска с горы* (hill climbing). Он начинает со случайного решения и ищет лучшие решения (с меньшим значением целевой функции) по соседству. Можно провести аналогию со спуском с горы (рис. 5.1), отсюда и название.

Представьте, что человечек на рисунке – это вы и оказались в этом месте по воле случая. Вы хотите добраться до самой низкой точки, чтобы найти воду. Для этого вы, наверное, осмотритесь и направитесь туда, где склон самый крутой. И будете двигаться в направлении наибольшей крутизны, пока не дойдете до точки, где местность становится ровной или начинается подъем.

Применим этот подход к решению задачи об отыскании наилучшего расписания путешествия для семейства Глассов. Начнем со случайно

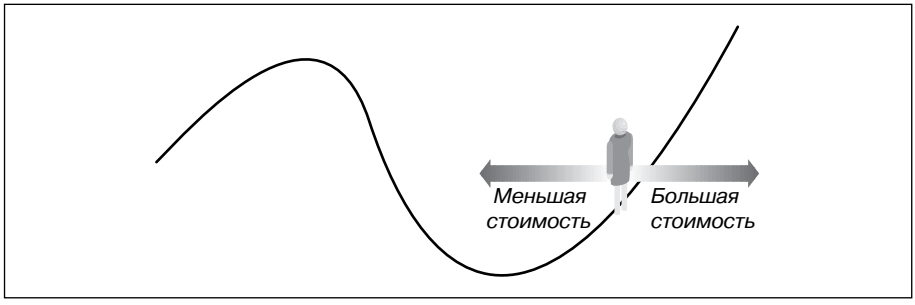


***Рис. 5.1.** Поиск минимума стоимости на горе*

выбранного расписания и просмотрим все расписания в его окрестности. В данном случае это означает просмотр таких расписаний, для которых один человек выбирает рейс, вылетающий чуть раньше или чуть позже. Для каждого из соседних расписаний вычисляется стоимость, и расписание с наименьшей стоимостью становится новым решением. Этот процесс повторяется и завершается, когда ни одно из соседних расписаний не дает улучшения стоимости.

Реализуется алгоритм следующей функцией, которую нужно добавить в файл optimization.py:

```
def hillclimb(domain,costf):
  # Выбрать случайное решение
  sol=[random.randint(domain[i][0],domain[i][1])
    for i in range(len(domain))]

  # Главный цикл
  while 1:

    # Создать список соседних решений
    neighbors=[]
    for j in range(len(domain)):

      # Отходим на один шаг в каждом направлении
      if sol[j]>domain[j][0]:
        neighbors.append(sol[0:j]+[sol[j]+1]+sol[j+1:])
      if sol[j]<domain[j][1]:
        neighbors.append(sol[0:j]+[sol[j]-1]+sol[j+1:])

    # Ищем наилучшее из соседних решений
    current=costf(sol)
    best=current
    for j in range(len(neighbors)):
      cost=costf(neighbors[j])
      if cost<best:
        best=cost
        sol=neighbors[j]

    # Если улучшения нет, мы достигли дна
    if best==current:
      break

  return sol
```

Для выбора начального решения эта функция генерирует случайный список чисел из заданного диапазона. Соседи текущего решения ищутся путем посещения каждого элемента списка и создания двух новых списков, в одном из которых этот элемент увеличен на единицу, а в другом уменьшен на единицу. Лучшее из соседних решений становится новым решением. Выполните эту функцию в интерактивном сеансе и сравните результаты с теми, что были найдены случайным поиском:

```
>>> s=optimization.hillclimb(domain,optimization.schedulecost)
>>> optimization.schedulecost(s)
```

```
3063
>>> optimization.printschedule(s)
Seymour     BOS 12:34-15:02 $109 10:33-12:03 $ 74
Franny      DAL 10:30-14:57 $290 10:51-14:16 $256
Zooey       CAK 10:53-13:36 $189 10:32-13:16 $139
Walt        MIA 11:28-14:40 $248 12:37-15:05 $170
Buddy       ORD 12:44-14:17 $134 10:33-13:11 $132
Les         OMA 11:08-13:07 $175 18:25-20:34 $205
```

Работает эта функция быстро и, как правило, находит лучшее решение, чем при случайном поиске. Однако у алгоритма спуска с горы, есть один существенный недостаток. Взгляните на рис. 5.2.

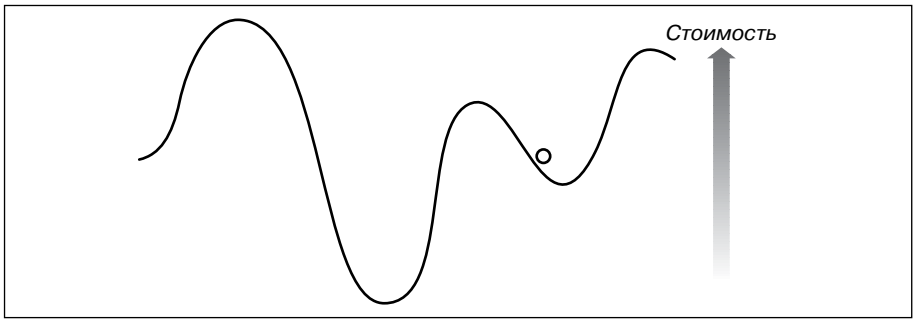


Рис. 5.2. *Застряли в точке локального минимума*

Из рисунка видно, что, спускаясь по склону, мы необязательно отыщем наилучшее возможное решение. Найденное решение будет *локальным минимумом*, то есть лучшим из всех в ближайшей окрестности, но это не означает, что оно вообще лучшее. Решение, лучшее среди всех возможных, называется *глобальным минимумом*, и именно его хотят найти все алгоритмы оптимизации. Один из возможных подходов к решению проблемы называется *спуском с горы со случайным перезапуском*. В этом случае алгоритм спуска выполняется несколько раз со случайными начальными точками в надежде, что какое-нибудь из найденных решений будет близко к глобальному минимуму. В следующих двух разделах мы покажем другие способы избежать застревания в точке локального минимума.

Алгоритм имитации отжига

Алгоритм имитации отжига навеян физическими аналогиями. Отжигом называется процесс нагрева образца с последующим медленным охлаждением. Поскольку сначала атомы заставляют попрыгать, а затем постепенно «отпускают вожжи», то они переходят в новую низкоэнергетичную конфигурацию.

Алгоритмическая версия отжига начинается с выбора случайного решения задачи. В ней используется переменная, интерпретируемая как

температура, начальное значение которой очень велико, но постепенно уменьшается. На каждой итерации случайным образом выбирается один из параметров решения и изменяется в определенном направлении. В нашем случае Сеймур (Seymour) может лететь обратно не вторым, а третьим рейсом. Вычисляются и сравниваются стоимости до и после изменения.

Тут есть важная деталь: если новая стоимость ниже, то новое решение становится текущим, как и в алгоритме спуска с горы. Но, даже если новая стоимость *выше*, новое решение все равно может стать текущим с некоторой вероятностью. Так мы пытаемся избежать ситуации скатывания в локальный минимум, изображенной на рис. 5.2.

Иногда необходимо перейти к худшему решению, чтобы найти лучшее. Алгоритм имитации отжига работает, потому что всегда готов перейти к лучшему решению, но ближе к началу процесса соглашается принять и худшее. По мере того как процесс развивается, алгоритм соглашается на худшее решение все с меньшей охотой, а в конце работы принимает только лучшее. Вероятность того, что будет принято решение с более высокой стоимостью, рассчитывается по формуле

$$p = e^{((-\text{высокая стоимость} - \text{низкая стоимость}) / \text{температура})}$$

Поскольку температура (готовность принять худшее решение) вначале очень высока, то показатель степени близок к 0, поэтому вероятность равна почти 1. По мере уменьшения температуры разность между высокой стоимостью и низкой стоимостью становится более значимой, а чем больше разность, тем ниже вероятность, поэтому алгоритм из всех худших решений будет выбирать те, что лишь немногим хуже текущего.

Добавьте в файл optimization.py новую функцию `annealingoptimize`, реализующую описанный алгоритм:

```
def annealingoptimize(domain,costf,T=10000.0,cool=0.95,step=1):
  # Инициализировать переменные случайным образом
  vec=[float(random.randint(domain[i][0],domain[i][1]))
       for i in range(len(domain))]

  while T>0.1:
    # Выбрать один из индексов
    i=random.randint(0,len(domain)-1)

    # Выбрать направление изменения
    dir=random.randint(-step,step)

    # Создать новый список, в котором одно значение изменено
    vecb=vec[:]
    vecb[i]+=dir
    if vecb[i]<domain[i][0]: vecb[i]=domain[i][0]
    elif vecb[i]>domain[i][1]: vecb[i]=domain[i][1]

    # Вычислить текущую и новую стоимость
```

```
    ea=costf(vec)
    eb=costf(vecb)
    p=pow(math.e,(-eb-ea)/T)

    # Новое решение лучше? Если нет, метнем кости
    if (eb<ea or random.random( )<p):
      vec=vecb

    # Уменьшить температуру
    T=T*cool
  return vec
```

Для выполнения отжига эта функция сначала генерирует случайное решение в виде вектора нужной длины, все элементы которого попадают в диапазон, определенный параметром `domain`. Параметры `temperature` и `cool` (скорость охлаждения) необязательны. На каждой итерации в качестве `i` случайно выбирается одна из переменных решения, а `dir` случайно выбирается между `-step` и `step`. Вычисляется стоимость текущего решения и стоимость решения, получающегося изменением переменной `i` на величину `step`.

В строке, выделенной полужирным шрифтом, вычисляется вероятность, которая уменьшается с уменьшением `T`. Если величина, случайно выбранная между 0 и 1, оказывается меньше вычисленной вероятности или если новое решение лучше текущего, то новое решение становится текущим. Цикл продолжается, пока температура почти не достигнет нуля, причем каждый раз температура умножается на скорость охлаждения.

А теперь попробуйте выполнить оптимизацию методом имитации отжига:

```
>>> reload(optimization)
>>> s=optimization.annealingoptimize(domain,optimization.schedulecost)
>>> optimization.schedulecost(s)
2278
>>> optimization.printschedule(s)
Seymour    Boston 12:34-15:02 $109 10:33-12:03 $ 74
Franny     Dallas 10:30-14:57 $290 10:51-14:16 $256
Zooey       Akron 10:53-13:36 $189 10:32-13:16 $139
Walt        Miami 11:28-14:40 $248 12:37-15:05 $170
Buddy     Chicago 12:44-14:17 $134 10:33-13:11 $132
Les         Omaha 11:08-13:07 $175 15:07-17:21 $129
```

Как видите, нам удалось уменьшить суммарное время ожидания, не увеличивая стоимости. Понятно, что у вас получатся другие результаты, и не исключено, что они будут хуже. Для конкретной задачи всегда имеет смысл поэкспериментировать с различными значениями начальной температуры и скорости охлаждения. Можно также изменить величину шага случайного смещения.

Генетические алгоритмы

Еще один класс методов оптимизации, также навеянный природой, называется *генетическими алгоритмами*. Принцип их работы состоит в том, чтобы создать набор случайных решений, который называется *популяцией*. На каждом шаге оптимизации целевая функция вычисляется для всей популяции, в результате чего получается ранжированный список решений. В табл. 5.1 приведен пример.

Таблица 5.1. Ранжированный список решений и их стоимостей

Решение	Стоимость
[7, 5, 2, 3, 1, 6, 1, 6, 7, 1, 0, 3]	4394
[7, 2, 2, 2, 3, 3, 2, 3, 5, 2, 0, 8]	4661
...	...
[0, 4, 0, 3, 8, 8, 4, 4, 8, 5, 6, 1]	7845
[5, 8, 0, 2, 8, 8, 8, 2, 1, 6, 6, 8]	8088

Проранжировав решения, мы создаем новую популяцию, которая называется следующим *поколением*. Сначала в новую популяцию включаются самые лучшие решения из текущей. Этот процесс называется *элитизмом*. Кроме них, в следующую популяцию входят совершенно новые решения, получающиеся путем модификации наилучших.

Модифицировать решения можно двумя способами. Более простой называется *мутацией*; обычно это небольшое, простое, случайное изменение существующего решения. В нашем случае для мутации достаточно выбрать одну из переменных решения и уменьшить либо увеличить ее. На рис. 5.3 приведено два примера.

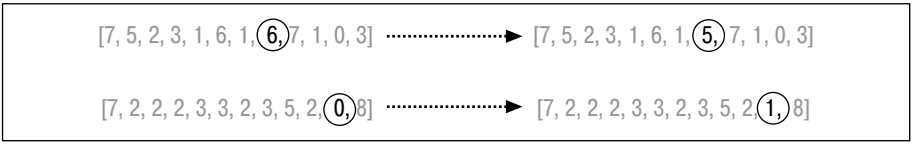


Рис. 5.3. *Примеры мутации решения*

Другой способ называется *скрещиванием* (или *кроссовером*). Состоит он в том, что мы берем какие-нибудь два из лучших решений и как-то комбинируем их. В нашем примере достаточно взять случайное число элементов из одного решения, а остальные – из другого, как показано на рис. 5.4.

Размер новой популяции обычно совпадает с размером предыдущей, а создается она путем случайных мутаций и скрещиваний лучших

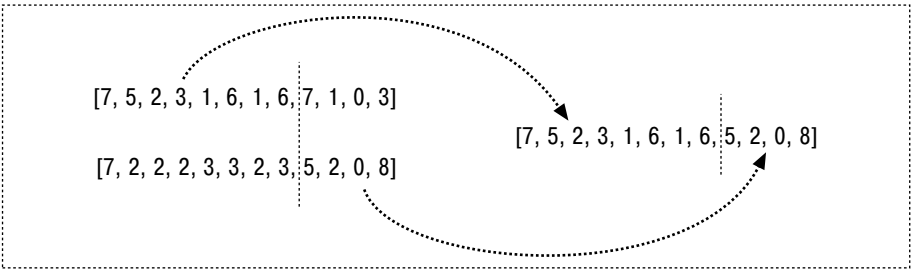


***Рис. 5.4.** Пример скрещивания*

решений. Затем этот процесс повторяется – новая популяция ранжируется и создается очередное поколение. Так продолжается заданное число раз или до тех пор, пока на протяжении нескольких поколений не наблюдается никаких улучшений.

Добавьте функцию `geneticoptimize` в файл optimization.py:

```
def geneticoptimize(domain,costf,popsize=50,step=1,
  mutprod=0.2,elite=0.2,maxiter=100):
  # Операция мутации
  def mutate(vec):
    i=random.randint(0,len(domain)-1)
    if random.random( )<0.5 and vec[i]>domain[i][0]:
      return vec[0:i]+[vec[i]-step]+vec[i+1:]
    elif vec[i]<domain[i][1]:
      return vec[0:i]+[vec[i]+step]+vec[i+1:]

  # Операция скрещивания
  def crossover(r1,r2):
    i=random.randint(1,len(domain)-2)
    return r1[0:i]+r2[i:]

  # Строим начальную популяцию
  pop=[]
  for i in range(popsize):
    vec=[random.randint(domain[i][0],domain[i][1])
         for i in range(len(domain))]
    pop.append(vec)

  # Сколько брать победителей из каждой популяции?
  topelite=int(elite*popsize)

  # Главный цикл
  for i in range(maxiter):
    scores=[(costf(v),v) for v in pop]
    scores.sort( )
    ranked=[v for (s,v) in scores]

    # Сначала включаем только победителей
    pop=ranked[0:topelite]
```

```
    # Добавляем особей, полученных мутацией и скрещиванием победивших
    # родителей
    while len(pop)<popsize:
      if random.random( )<mutprob:
        # Мутация
        c=random.randint(0,topelite)
        pop.append(mutate(ranked[c]))
      else:
        # Скрещивание
        c1=random.randint(0,topelite)
        c2=random.randint(0,topelite)
        pop.append(crossover(ranked[c1],ranked[c2]))

    # Печатаем полученные к текущему моменту наилучшие результаты
    print scores[0][0]

  return scores[0][1]
```

Эта функция принимает несколько необязательных параметров:

`popsize`

Размер популяции.

`mutprob`

Вероятность того, что новая особь будет результатом мутации, а не скрещивания.

`elite`

Доля особей в популяции, считающихся хорошими решениями и переходящих в следующее поколение.

`maxiter`

Количество поколений.

Попробуем оптимизировать план путешествия с помощью генетического алгоритма:

```
>>> s=optimization.geneticoptimize(domain,optimization.schedulecost)
3532
3503
...
2591
2591
2591
>>> optimization.printschedule(s)
   Seymour     BOS 12:34-15:02 $109 10:33-12:03 $ 74
    Franny     DAL 10:30-14:57 $290 10:51-14:16 $256
     Zooey     CAK 10:53-13:36 $189 10:32-13:16 $139
      Walt     MIA 11:28-14:40 $248 12:37-15:05 $170
     Buddy     ORD 12:44-14:17 $134 10:33-13:11 $132
       Les     OMA 11:08-13:07 $175 11:07-13:24 $171
```

В главе 11 мы ознакомимся с обобщением генетических алгоритмов – техникой *генетического программирования*, когда те же идеи применяются для создания совершенно новых программ.

Отцом генетических алгоритмов принято считать ученого-теоретика Джона Холланда (John Holland), который в 1975 году написал книгу «Adaptation in Natural and Artificial Systems» (издательство Мичиганского университета). Но корни этих работ восходят к биологам 1950-х годов, которые пытались моделировать эволюцию на компьютерах. С тех пор генетические алгоритмы и другие методы оптимизации использовались для решения широчайшего круга задач, в том числе:

- Нахождения формы концертного зала с оптимальными акустическими характеристиками.
- Проектирования оптимальной формы крыла сверхзвукового самолета.
- Составления оптимальной библиотеки химических веществ для синтеза потенциальных лекарств.
- Автоматического проектирования микросхем для распознавания речи.

Решения этих задач можно представить в виде списков чисел. Это упрощает применение к ним генетических алгоритмов или метода имитации отжига.

Будет ли работать конкретный метод оптимизации, в значительной мере зависит от задачи. Метод имитации отжига, генетические алгоритмы и прочие методы оптимизации основаны на том факте, что в большинстве задач оптимальное решение близко к другим хорошим решениям. А на рис. 5.5 представлен случай, когда оптимизация может и не сработать.

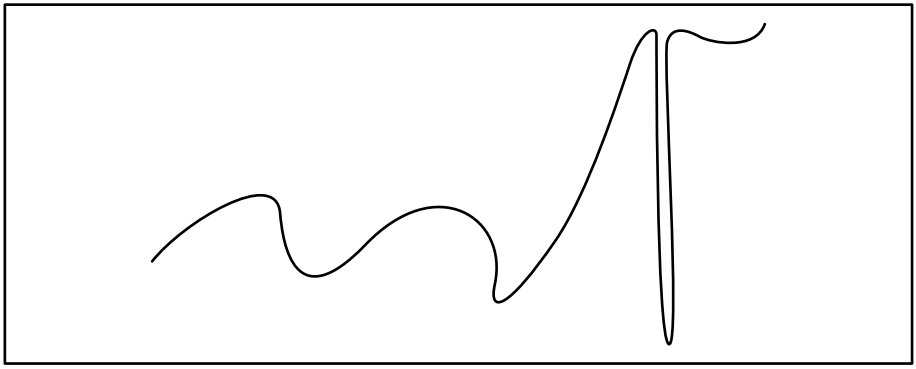


Рис. 5.5. *Задача, плохо поддающаяся оптимизации*

Оптимальная стоимость соответствует низшей точке очень крутого участка кривой в правой части рисунка. Любое достаточно близкое решение, скорее всего, будет отвергнуто из-за его высокой стоимости, поэтому мы никогда не сможем добраться до глобального минимума. Большинство алгоритмов остановятся в каком-то локальном минимуме в левой части рисунка.

Пример с составлением расписания полетов работает, потому что выбор второго рейса вместо третьего, скорее всего, изменит стоимость менее значительно, чем выбор восьмого рейса. Если бы рейсы были упорядочены случайным образом, то методы оптимизации работали бы не лучше случайного поиска; на самом деле не существует ни одного метода оптимизации, который в этом случае работал бы устойчиво лучше, чем случайный поиск.

Поиск реальных авиарейсов

Разобравшись с тестовыми данными, попробуем применить те же самые методы оптимизации к реальным данным об авиарейсах. Мы загрузим данные с сайта Kayak, который предоставляет API для поиска авиарейсов. Основное отличие тестовых данных от реальных заключается в том, что между крупными городами бывает гораздо больше девяти рейсов в день.

API сайта Kayak

Сайт Kayak, изображенный на рис. 5.6, – это популярная *вертикальная поисковая система*, предназначенная для путешественников. В Сети

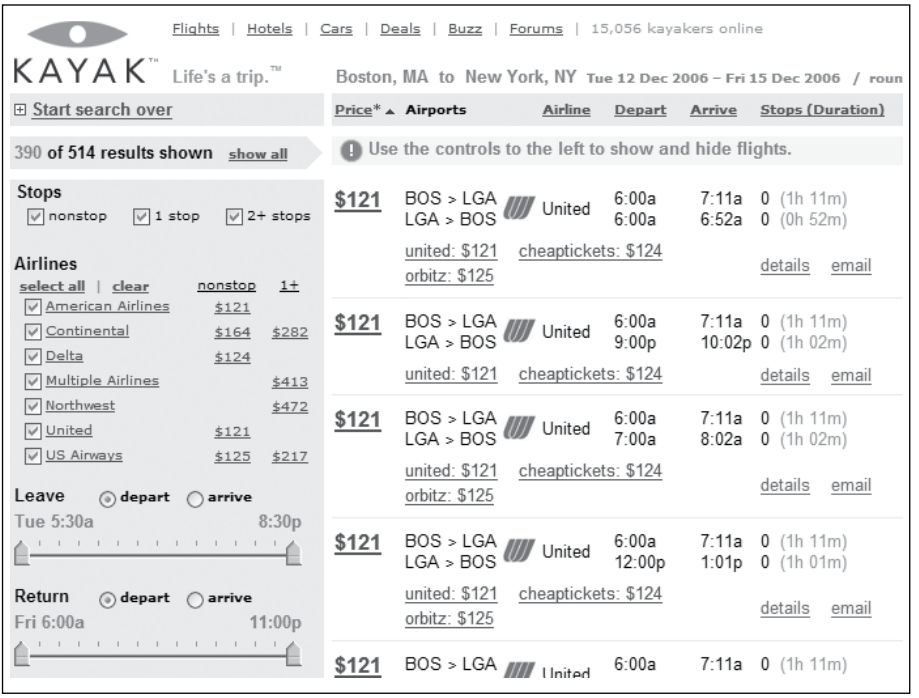


Рис. 5.6. *Интерфейс системы поиска маршрутов Kayak*

много сайтов подобного рода, но Kayak полезен тем, что предоставляет удобный API, написанный на XML, которым можно воспользоваться из программы на языке Python для поиска реальных маршрутов. Чтобы получить доступ к API, вам потребуется зарегистрироваться для получения ключа разработчика, зайдя на страницу *http://www.kayak.com/labs/api/search*.

Ключ разработчика представляет собой длинную строку букв и цифр, которую необходимо указывать при поиске авиарейсов с помощью системы Kayak (он годится и для поиска гостиниц, но эту тему мы обсуждать не будем). В настоящий момент не существует специального API на языке Python для сайта Kayak, похожего на API для сайта *del.icio.us*, но структура XML-документа очень хорошо описана. В этой главе мы покажем, как выполнить поиск с помощью пакетов urllib2 и xml.dom.minidom, включенных в стандартный дистрибутив Python.

Пакет minidom

Пакет minidom – часть стандартного дистрибутива Python. Это простая реализация интерфейса Document Object Model (DOM), который позволяет представить произвольный XML-документ в виде дерева объектов. Пакет принимает на входе строку или открытый файл, содержащий XML-документ, и возвращает объект, позволяющий легко извлекать нужную информацию. Введите, например, следующие команды в интерактивном сеансе:

```
>>> import xml.dom.minidom
>>> dom=xml.dom.minidom.parseString('<data><rec>Hello!</rec></data>')
>>> dom
<xml.dom.minidom.Document instance at 0x00980C38>
>>> r=dom.getElementsByTagName('rec')
>>> r
[<DOM Element: rec at 0xa42350>]
>>> r[0].firstChild
<DOM Text node "Hello!">
>>> r[0].firstChild.data
u'Hello!'
```

Поскольку сейчас многие сайты предлагают XML-интерфейс для доступа к информации, умение обращаться с Python-пакетами для работы с XML окажется весьма полезным для программирования коллективного разума. Вот сводка наиболее важных методов DOM-объектов, которыми мы будем пользоваться в процессе доступа к данным с помощью Kayak API:

`getElementsByTagName(name)`

Возвращает список узлов DOM, соответствующих элементам с тегом `name`.

`firstChild`

Возвращает первый дочерний узел данного объекта. В примере выше первым дочерним узлом узла `r` является узел, представляющий текст Hello.

`data`

Возвращает данные, ассоциированные с объектом. В большинстве случаев это строка в кодировке Unicode, содержащаяся внутри узла.

Поиск авиарейсов

Начнем с создания нового файла kayak.py, в который следует включить такие предложения:

```
import time
import urllib2
import xml.dom.minidom

kayakkey='ВАШ_КЛЮЧ'
```

Прежде всего нужно написать код открытия нового сеанса работы с сайтом Kayak. Он будет посылать запрос странице `apisession`, задавая в качества параметра `token` полученный вами ключ разработчика. В ответ возвращается XML-документ, содержащий тег `sid`, внутри которого находится идентификатор сеанса:

```
<sid>1-hX4lII_wS$8b06aO7kHj</sid>
```

Ваша функция должно просто разобрать этот документ и извлечь содержимое тега `sid`. Добавьте ее в файл kayak.py:

```
def getkayaksession( ):
  # Создаем URL для открытия нового сеанса
  url='http://www.kayak.com/k/ident/apisession?token=%s&version=1' % kayakkey

  # Разбираем полученный XML-документ
  doc=xml.dom.minidom.parseString(urllib2.urlopen(url).read( ))

  # Ищем элемент <sid>xxxxxxxx</sid>
  sid=doc.getElementsByTagName('sid')[0].firstChild.data
  return sid
```

Далее необходима функция, которая отправит запрос на поиск авиарейсов. Соответствующий URL очень длинный, поскольку содержит все параметры поиска. Самыми важными являются параметры `sid` (идентификатор сеанса, возвращенный функцией `getkayaksession`), `destination` (пункт назначения) и `depart_date` (дата вылета).

В XML-документе, возвращенном в ответ на этот запрос, имеется тег `searchid`, который функция извлекает точно так же, как это делает `getkayaksession`. Поскольку поиск может занять много времени, этот запрос не выдает результатов, он лишь начинает поиск и возвращает идентификатор, с помощью которого впоследствии можно периодически справляться о готовности результатов.

Добавьте в файл kayak.py такой код:

```
def flightsearch(sid,origin,destination,depart_date):
  # Создаем URL запроса на поиск
  url='http://www.kayak.com/s/apisearch?basicmode=true&oneway=y&origin=%s' %
    origin
```

```
    url+='&destination=%s&depart_date=%s' % (destination,depart_date)
    url+='&return_date=none&depart_time=a&return_time=a'
    url+='&travelers=1&cabin=e&action=doFlights&apimode=1'
    url+='&_sid_=%s&version=1' % (sid)

    # Получаем в ответ XML
    doc=xml.dom.minidom.parseString(urllib2.urlopen(url).read( ))

    # Извлекаем идентификатор поиска
    searchid=doc.getElementsByTagName('searchid')[0].firstChild.data

    return searchid
```

Наконец нам понадобится функция, которая запрашивает результаты до тех пор, пока у сервера еще что-то для нас осталось. Для получения результатов Kayak предоставляет еще один URL – `flight`. В возвращенном XML-документе имеется тег `morepending`. Если он содержит слово `true`, то имеются еще результаты. Функция должна обращаться к странице до тех пор, пока тег `morepending` содержит слово `true`, а затем вернуть полный набор результатов.

Добавьте ее в файл kayak.py:

```
def flightsearchresults(sid,searchid):

  # Удалить начальный $ и запятые и преобразовать строку в число с плавающей
  # точкой
  def parseprice(p):
    return float(p[1:].replace(',',''))

    # Цикл опроса
    while 1:
      time.sleep(2)

      # Создаем URL для опроса
      url='http://www.kayak.com/s/basic/flight?'
      url+='searchid=%s&c=5&apimode=1&_sid_=%s&version=1' % (searchid,sid)
      doc=xml.dom.minidom.parseString(urllib2.urlopen(url).read( ))

      # Ищем тег morepending и продолжаем, пока он содержит слово true
      morepending=doc.getElementsByTagName('morepending')[0].firstChild
      if morepending==None or morepending.data=='false': break

    # Теперь загружаем весь список
    url='http://www.kayak.com/s/basic/flight?'
    url+='searchid=%s&c=999&apimode=1&_sid_=%s&version=1' % (searchid,sid)
    doc=xml.dom.minidom.parseString(urllib2.urlopen(url).read( ))

    # Представляем различные элементы в виде списков
    prices=doc.getElementsByTagName('price')
    departures=doc.getElementsByTagName('depart')
    arrivals=doc.getElementsByTagName('arrive')
```

```
    # Объединяем их
    return zip([p.firstChild.data.split(' ')[1] for p in departures],
               [p.firstChild.data.split(' ')[1] for p in arrivals],
               [parseprice(p.firstChild.data) for p in prices])
```

Обратите внимание, что на последнем шаге функция формирует три списка, соответствующих элементам с тегами `price`, `depart` и `arrive`. Количество таких элементов одинаково – по одному для каждого рейса, поэтому можно воспользоваться функцией `zip`, которая объединит все три списка в один, элементами которого будут кортежи из трех элементов. Информация о вылете и прилете представлена в виде даты и времени, которые разделены пробелом, поэтому мы вызываем метод `split`, чтобы выделить только время. Кроме того, цена билета преобразуется в число с плавающей точкой с помощью вспомогательной функции `parseprice`.

Выполните поиск реальных авиарейсов в интерактивном сеансе, чтобы убедиться, что все работает (не забудьте изменить дату, указав какой-нибудь день в будущем):

```
>>> import kayak
>>> sid=kayak.getkayaksession( )
>>> searchid=kayak.flightsearch(sid,'BOS','LGA','11/17/2006')
>>> f=kayak.flightsearchresults(sid,searchid)
>>> f[0:3]
[(u'07:00', u'08:25', 60.3),
(u'08:30', u'09:49', 60.3),
(u'06:35', u'07:54', 65.0)]
```

Список рейсов отсортирован по цене, а рейсы с одинаковой стоимостью – по времени. Это хорошо, так как похожие решения оказываются рядом. Чтобы интегрировать этот код с написанным ранее, нам нужно создать полное расписание для всех членов семейства Глассов, сохранив ту же структуру, что и в тестовом файле. Для этого требуется лишь перебрать людей в списке и выполнить для каждого из них поиск рейсов туда и обратно. Добавьте в файл kayak.py функцию `createschedule`:

```
def createschedule(people,dest,dep,ret):
  # Получить идентификатор сеанса для выполнения поиска
  sid=getkayaksession( )
  flights={}

  for p in people:
    name,origin=p
    # Рейс туда
    searchid=flightsearch(sid,origin,dest,dep)
    flights[(origin,dest)]=flightsearchresults(sid,searchid)

    # Рейс обратно
    searchid=flightsearch(sid,dest,origin,ret)
    flights[(dest,origin)]=flightsearchresults(sid,searchid)

  return flights
```

Теперь можно попытаться оптимизировать рейсы для членов семьи, пользуясь реальными данными. Поиск на сайте Kayak занимает довольно много времени, поэтому для начала ограничимся лишь первыми двумя членами. Введите в интерактивном сеансе следующие команды:

```
>>> reload(kayak)
>>> f=kayak.createschedule(optimization.people[0:2],'LGA',
... '11/17/2006','11/19/2006')
>>> optimization.flights=f
>>> domain=[(0,30)]*len(f)
>>> optimization.geneticoptimize(domain,optimization.schedulecost)
770.0
703.0
...
>>> optimization.printschedule(s)
Seymour     BOS 16:00-17:20 $85.0 19:00-20:28 $65.0
Franny      DAL 08:00-17:25 $205.0 18:55-00:15 $133.0
```

Примите поздравления! Вы только что выполнили оптимизацию на реальных данных об авиарейсах. Пространство поиска теперь гораздо больше, так что имеет смысл поэкспериментировать с параметрами – максимальной температурой и скоростью обучения.

Эти идеи можно развить во многих направлениях. Можно добавить поиск на метеорологических сайтах, чтобы оптимизировать сочетание цены и температуры в пункте назначения. Или включить поиск гостиниц, чтобы найти пункт назначения с разумным сочетанием цен на авиабилеты и проживание. В Интернете есть тысячи сайтов, предоставляющих путешественникам данные, которые можно оптимизировать.

API сайта Kayak налагает ограничение на количество поисков в течение суток, но возвращает прямые ссылки на сайты, где можно заказать билеты на самолет или забронировать номер в гостинице, а значит, вы легко сможете включить этот API в состав любого приложения.

Оптимизация с учетом предпочтений

Мы рассмотрели пример одной задачи, которую можно решить методами оптимизации, но есть немало других задач, внешне непохожих, но решаемых с помощью тех же методов. Напомним, что для применения оптимизации должны выполняться два основных требования: наличие целевой функции и тот факт, что близкие параметры дают близкие решения. Не каждую задачу, удовлетворяющую этим требованиям, можно решить методами оптимизации, но есть неплохие шансы, что попытка применить эти методы даст интересные результаты, о которых вы даже не подозревали.

В этом разделе мы займемся другой задачей, для которой оптимизация просто напрашивается. Общая формулировка такова: распределить ограниченные ресурсы между людьми, у которых есть явно выраженные предпочтения, так чтобы все были максимально счастливы (или, в зависимости от склада характера, минимально недовольны).

Оптимизация распределения студентов по комнатам

В этом разделе мы решим задачу о распределении студентов по комнатам в общежитии с учетом их основного и альтернативного пожеланий. Хотя формулировка довольно специфична, ее можно легко обобщить на похожие задачи, – тот же самый код годится для распределения игроков по столам в онлайновой карточной игре, для распределения ошибок между разработчиками в большом программном проекте и даже для распределения работы по дому между прислугой. Как и раньше, наша цель – собрать информацию об отдельных людях и найти такое сочетание, которое приводит к оптимальному результату.

В нашем примере будет пять двухместных комнат и десять претендующих на них студентов. У каждого студента есть основное и альтернативное пожелание. Создайте новый файл dorm.py и добавьте в него список комнат, список людей и два пожелания для каждого человека:

```
import random
import math

# Двухместные комнаты
dorms=['Zeus','Athena','Hercules','Bacchus','Pluto']

# Люди и два пожелания у каждого
prefs=[('Toby', ('Bacchus', 'Hercules')),
       ('Steve', ('Zeus', 'Pluto')),
       ('Andrea', ('Athena', 'Zeus')),
       ('Sarah', ('Zeus', 'Pluto')),
       ('Dave', ('Athena', 'Bacchus')),
       ('Jeff', ('Hercules', 'Pluto')),
       ('Fred', ('Pluto', 'Athena')),
       ('Suzie', ('Bacchus', 'Hercules')),
       ('Laura', ('Bacchus', 'Hercules')),
       ('Neil', ('Hercules', 'Athena'))]
```

Сразу видно, что удовлетворить основное пожелание каждого человека не удастся, так как на два места в комнате Bacchus имеется три претендента. Поместить любого из этих трех в ту комнату, которую он указал в качестве альтернативы, тоже невозможно, так как в комнате Hercules всего два места.

Эта задача намеренно сделана небольшой, чтобы проследить логику было просто. Но в реальности может быть две-три сотни студентов, претендующих на множество комнат в общежитии. Так как в приведенном примере общее число вариантов порядка 100 000, то можно перебрать их все и найти оптимальный. Но если комнаты будут четырехместными, то количество вариантов будет исчислять триллионами.

Чтобы представить решение этой задачи, придется проявить больше смекалки, чем в задаче об авиарейсах. Теоретически можно создать список чисел, по одному для каждого студента, так что число будет представлять ту комнату, куда мы помещаем студента. Проблема в том,

что в таком представлении отсутствует ограничение, запрещающее помещать в одну комнату более двух студентов. Список, состоящий только из нулей, означает, что всех поместили в комнату Zeus, а это не соответствует никакому решению задачи.

Один из способов разрешить эту проблему состоит в том, чтобы выбрать такую целевую функцию, которая будет возвращать очень большое значение для недопустимых решений, но тогда алгоритму оптимизации будет трудно находить лучшие решения, так как он не может сказать, близко ли данное решение к другому хорошему или хотя бы допустимому. Вообще говоря, лучше не тратить процессорное время на поиск среди заведомо недопустимых решений.

Более правильный подход – отыскать такой способ представления решений, при котором любое представимое решение допустимо. Допустимое еще не означает хорошее, просто в каждой комнате должно оказаться ровно два студента. Можно, например, поступить следующим образом. Будем считать, что в каждой комнате есть два отсека, то есть всего их в нашем случае будет десять. Каждому студенту по очереди назначается один из незанятых отсеков; первого можно поместить в любой из десяти отсеков, второго – в любой из оставшихся девяти и т. д.

Область определения для поиска должна быть построена с учетом этого ограничения. Добавьте такие строки в файл dorm.py:

```
# [(0,9),(0,8),(0,7),(0,6),...,(0,0)]
domain=[(0,(len(dorms)*2)-i-1) for i in range(0,len(dorms)*2)]
```

Следующая функция, печатающая решение, иллюстрирует принцип работы отсеков. Сначала она создает список отсеков, по два на каждую комнату. Затем она в цикле пробегает по всем числам, составляющим решение, и для каждого находит номер комнаты в данной позиции списка отсеков. Это та комната, в которую поместили студента. Функция печатает имя студента и название комнаты, а затем удаляет отсек из списка, чтобы в него нельзя было поместить другого студента. После завершающей итерации распределение студентов по комнатам распечатано, а список отсеков пуст.

Добавьте эту функцию в файл dorm.py:

```
def printsolution(vec):
  slots=[]
  # Создаем по два отсека на каждую комнату
  for i in range(len(dorms)): slots+=[i,i]

  # Цикл по распределению студентов по комнатам
  for i in range(len(vec)):
    x=int(vec[i])

    # Выбираем любой отсек из оставшихся
    dorm=dorms[slots[x]]
    # Печатаем имя студента и название комнаты, в которую он попал
    print prefs[i][0],dorm
```

```
    # Удаляем этот отсек
    del slots[x]
```

В интерактивном сеансе импортируйте файл и распечатайте решение:

```
>>> import dorm
>>> dorm.printsolution([0,0,0,0,0,0,0,0,0,0])
Toby Zeus
Steve Zeus
Andrea Athena
Sarah Athena
Dave Hercules
Jeff Hercules
Fred Bacchus
Suzie Bacchus
Laura Pluto
Neil Pluto
```

Если вы захотите изменить числа, чтобы посмотреть на другие решения, то не забывайте, что числа не должны выходить за пределы отведенных для них диапазонов. Первое число в списке должно находиться в диапазоне от 0 до 9, второе – от 0 до 8 и т. д. Если это ограничение будет нарушено, функция возбудит исключение. Поскольку функции оптимизации будут выбирать числа из диапазонов, заданных в параметре `domain`, то мы в реализации алгоритма с этой проблемой не столкнемся.

Целевая функция

Целевая функция работает аналогично функции печати. Создается начальный список отсеков, и уже использованные отсеки из него удаляются. Стоимость вычисляется путем сравнения комнаты, в которую студент помещен, с двумя его пожеланиями. Стоимость не изменяется, если студенту досталась та комната, в которую он больше всего хотел поселиться; увеличивается на 1, если это второе из его пожеланий; и увеличивается на 3, если он вообще не хотел жить в этой комнате.

```
def dormcost(vec):
  cost=0
  # Создаем список отсеков
  slots=[0,0,1,1,2,2,3,3,4,4]

  # Цикл по студентам
  for i in range(len(vec)):
    x=int(vec[i])
    dorm=dorms[slots[x]]
    pref=prefs[i][1]
    # Стоимость основного пожелания равна 0, альтернативного - 1
    if pref[0]==dorm: cost+=0
    elif pref[1]==dorm: cost+=1
    else: cost+=3
    # Если комната не входит в список пожеланий, стоимость увеличивается на 3
```

```
        # Удалить выбранный отсек
        del slots[x]

    return cost
```

При конструировании целевой функции полезно стремиться к тому, чтобы стоимость идеального решения (в данном случае каждый студент заселился в комнату, которую поставил на первое место в своем списке предпочтений) была равна нулю. В рассматриваемом примере мы уже убедились, что идеальное решение недостижимо, но, зная о том, что его стоимость равна нулю, можно оценить, насколько мы к нему приблизились. У этого правила есть еще одно достоинство – алгоритм оптимизации может прекратить поиск, если уже найдено идеальное решение.

Выполнение оптимизации

Наличия представления решений, целевой функций и функции печати результатов достаточно для запуска написанных ранее функций оптимизации. Введите следующие команды:

```
>>> reload(dorm)
>>> s=optimization.randomoptimize(dorm.domain,dorm.dormcost)
>>> dorm.dormcost(s)
18
>>> optimization.geneticoptimize(dorm.domain,dorm.dormcost)
13
10
...
4
>>> dorm.printsolution(s)
Toby Athena
Steve Pluto
Andrea Zeus
Sarah Pluto
Dave Hercules
Jeff Hercules
Fred Bacchus
Suzie Bacchus
Laura Athena
Neil Zeus
```

Можете и тут поиграть с параметрами – вдруг генетический алгоритм сможет найти решение быстрее.

Визуализация сети

В последнем примере мы покажем еще один способ применения оптимизации к, казалось бы, совершенно непохожим задачам. В данном случае речь пойдет о визуализации сетей. Сетью мы будем называть любое множество взаимосвязанных сущностей. Среди онлайновых приложений хорошими примерами могут служить социальные сети на

таких сайтах, как MySpace, Facebook или LinkedIn, где люди взаимосвязаны, потому что являются друзьями или коллегами. Каждый член сайта выбирает, с кем он связан, а все вместе они образуют сеть, составленную из людей. Интересно было бы визуализировать подобные сети, чтобы проявить их структуру, например найти людей-«соединителей» (таких, которые знают многих других или служат связующим звеном между автономными группами).

Задача о размещении

При рисовании сети, визуализирующей большую группу людей и связей между ними, требуется решить, где разместить на картинке каждое имя (или значок). Рассмотрим, например, сеть, изображенную на рис. 5.7.

Из рисунка видно, что Август (Augustus) дружит с Вилли (Willy), Вайолетом (Violet) и Мирандой (Miranda). Но сеть нарисована беспорядочно, а при добавлении новых людей в ней вообще будет невозможно разобраться. На рис. 5.8 изображено более понятное размещение.

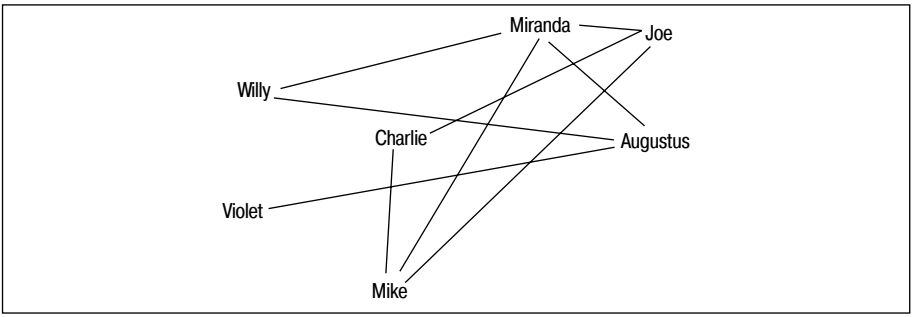


Рис. 5.7. *Неудачное размещение сети*

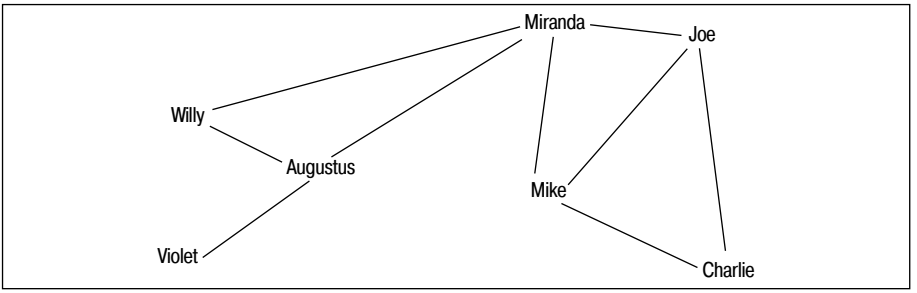


Рис. 5.8. *Улучшенное размещение сети*

В этом разделе мы посмотрим, как применить методы оптимизации для создания упорядоченных визуальных образов. Заведите новый файл socialnetwork.py и включите в него некоторые факты о малом подмножестве социальной сети:

```
import math

people=['Charlie','Augustus','Veruca','Violet','Mike','Joe','Willy',
'Miranda']

links=[('Augustus', 'Willy'),
       ('Mike', 'Joe'),
       ('Miranda', 'Mike'),
       ('Violet', 'Augustus'),
       ('Miranda', 'Willy'),
       ('Charlie', 'Mike'),
       ('Veruca', 'Joe'),
       ('Miranda', 'Augustus'),
       ('Willy', 'Augustus'),
       ('Joe', 'Charlie'),
       ('Veruca', 'Augustus'),
       ('Miranda', 'Joe')]
```

Наша задача – написать программу, которая получает на входе информацию о том, кто с кем дружит, и генерирует удобную для восприятия диаграмму сети. Обычно для этого применяется *алгоритм масс и пружин*. В его основе лежит физическая модель, поскольку различные узлы давят друг на друга, пытаясь «оттолкнуться», тогда как связи сближают соединенные между собой узлы. Следовательно, сеть медленно приближается к такому состоянию, когда несвязанные узлы находятся далеко друг от друга, а связанные – близко, но не слишком.

К сожалению, алгоритм масс и пружин не препятствует пересечению линий. В сети с большим количеством звеньев из-за этого трудно понять, какие узлы связаны между собой, поскольку визуально проследить линию проблематично. Но если применить к задаче размещения оптимизацию, то нужно лишь определить целевую функцию и попытаться ее минимизировать. В данном случае интересная целевая функция – это количество пересекающих друг друга линий.

Подсчет пересекающихся линий

Чтобы воспользоваться уже разработанными функциями оптимизации, нам необходимо представить решение в виде списка чисел. К счастью, для этой задачи найти такое представление совсем просто. У каждого узла есть координаты x и y, поэтому достаточно поместить координаты всех узлов в один длинный список:

```
sol=[120,200,250,125 ...
```

Здесь Чарли (Charlie) помещен в точку с координатами (120, 200), Август (Augustus) – в точку (250, 125) и т. д.

Наша целевая функция должна подсчитать количество пересекающихся отрезков. Вывод формулы, определяющей точку пересечения двух прямолинейных отрезков, выходит за рамки данной главы, но

основная идея в том, чтобы вычислить, какую часть один отрезок перекрывает на другом. Если эта величина заключена между 0 (один конец отрезка) и 1 (другой конец), то отрезки пересекаются, в противном случае не пересекаются.

Следующая функция перебирает все пары соединительных отрезков и, зная координаты их концов, определяет, пересекаются ли они. Если да, к общей стоимости добавляется 1. Добавьте в файл socialnetwork.py функцию `crosscount`:

```
def crosscount(v):
  # Преобразовать список чисел в словарь человека: координаты (x,y)
  loc=dict([(people[i],(v[i*2],v[i*2+1])) for i in range(0,len(people))])
  total=0

  # Перебрать все пары отрезков
  for i in range(len(links)):
    for j in range(i+1,len(links)):

      # Получить координаты концов
      (x1,y1),(x2,y2)=loc[links[i][0]],loc[links[i][1]]
      (x3,y3),(x4,y4)=loc[links[j][0]],loc[links[j][1]]

      den=(y4-y3)*(x2-x1)-(x4-x3)*(y2-y1)

      # den==0, если прямые параллельны
      if den==0: continue

      # В противном случае ua и ub - доли, содаваемые на каждом отрезке
      # точкой их пересечения
      ua=((x4-x3)*(y1-y3)-(y4-y3)*(x1-x3))/den
      ub=((x2-x1)*(y1-y3)-(y2-y1)*(x1-x3))/den

      # Если для обоих отрезков доля оказалась между 0 и 1, то они
      # пересекаются
      if ua>0 and ua<1 and ub>0 and ub<1:
        total+=1
  return total
```

Областью определения в данном случае является диапазон изменения каждой координаты. Если сеть нужно уместить на картинке размером 400×400 пикселей, то область определения должна быть чуть меньше, чтобы осталось место для полей. Добавьте в файл socialnetwork.py такую строку:

```
domain=[(10,370)]*(len(people)*2)
```

Теперь попробуйте прогнать какие-нибудь алгоритмы оптимизации и найти решения с небольшим числом пересекающихся линий:

```
>>> import socialnetwork
>>> import optimization
>>> sol=optimization.randomoptimize(socialnetwork.domain,
```

```
    socialnetwork.crosscount)
>>> socialnetwork.crosscount(sol)
12
>>> sol=optimization.annealingoptimize(socialnetwork.domain,
    socialnetwork.crosscount,step=50,cool=0.99)
>>> socialnetwork.crosscount(sol)
1
>>> sol
[324, 190, 241, 329, 298, 237, 117, 181, 88, 106, 56, 10, 296, 370, 11, 312]
```

Алгоритм имитации отжига, скорее всего, сможет найти решение с очень малым количеством пересечений, но интерпретировать список координат тяжело. В следующем разделе мы покажем, как автоматически нарисовать сеть.

Рисование сети

Нам понадобится библиотека Python Imaging Library, которой мы пользовались в главе 3. Если вы ее еще не установили, обратитесь к приложению А, где приведены инструкции по скачиванию и установке последней версии.

Код рисования сети прямолинеен. Надо создать изображение, нарисовать отрезки между людьми, а потом – узлы, представляющие каждого человека. Имена людей наносятся во вторую очередь, чтобы линии их не перекрывали. Добавьте следующую функцию в файл socialnetwork.py:

```
def drawnetwork(sol):
  # Создать изображение
  img=Image.new('RGB',(400,400),(255,255,255))
  draw=ImageDraw.Draw(img)

  # Создать словарь позиций
  pos=dict([(people[i],(sol[i*2],sol[i*2+1])) for i in range(0,len(people))])

  # Нарисовать соединительные отрезки
  for (a,b) in links:
    draw.line((pos[a],pos[b]),fill=(255,0,0))

  # Нанести имена людей
  for n,p in pos.items( ):
    draw.text(p,n,(0,0,0))

  img.show( )
```

Для запуска этой функции в интерактивном сеансе перегрузите модуль и вызовите функцию:

```
>>> reload(socialnetwork)
>>> drawnetwork(sol)
```

На рис. 5.9 показан возможный результат оптимизации.

Разумеется, ваше решение будет отличаться от приведенного. Иногда решение выглядит довольно странно. Так как мы ставили себе целью

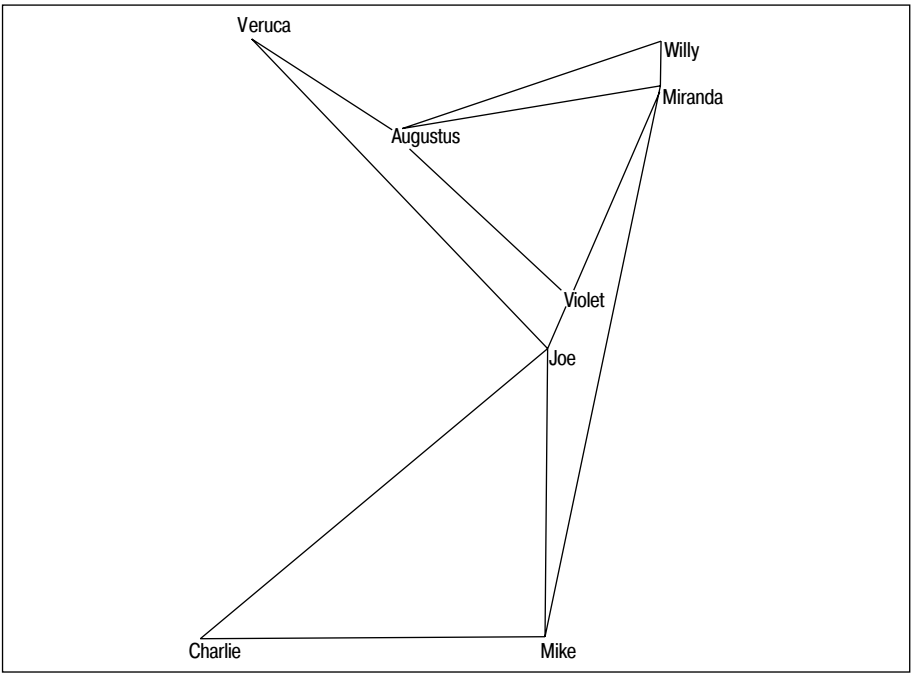


Рис. 5.9. *Размещение, получившееся в результате минимизации количества пересечений*

лишь минимизировать количество пересечений, то целевая функция не штрафует за слишком острые углы или за то, что два узла оказались чересчур близко друг к другу. В данном случае алгоритм оптимизации ведет себя как джинн, исполняющий ваши желания буквально, поэтому так важно четко формулировать, чего вы хотите. Зачастую решение удовлетворяет критерию оптимальности, но выглядит совершенно не так, как вы ожидали.

Чтобы штрафовать за слишком близкое расположение узлов, можно вычислить расстояние между узлами и разделить его на желательное минимальное расстояние. Модифицируйте функцию `crosscount`, добавив в конец (перед `return`) следующий код:

```
for i in range(len(people)):
  for j in range(i+1,len(people)):
    # Получить позиции обоих узлов
    (x1,y1),(x2,y2)=loc[people[i]],loc[people[j]]

    # Вычислить расстояние между ними
    dist=math.sqrt(math.pow(x1-x2,2)+math.pow(y1-y2,2))
    # Штраф, если расстояние меньше 50 пикселей
    if dist<50:
      total+=(1.0-(dist/50.0))
```

Тем самым мы увеличиваем стоимость для каждой пары узлов, расположенных на расстоянии менее 50 пикселей друг от друга, пропорционально расстоянию. Если они оказались в одной и той же точке, то штраф составит 1. Снова выполните оптимизацию и посмотрите, удалось ли растянуть размещение.

Другие возможности

В этой главе мы рассмотрели три совершенно разных применения алгоритмов оптимизации, но это лишь малая толика возможного. Как мы неоднократно отмечали, существенными этапами оптимизации являются выбор представления решения и целевой функции. Если это удается сделать, то есть все шансы, что оптимизация поможет решить стоящую перед вами задачу.

Интересно было бы подумать над такой задачей: взять группу людей и разделить их на команды с равномерным распределением умений. Например, для какого-нибудь игрового конкурса (типа игры «Что? Где? Когда?») хорошо бы сформировать команды игроков, обладающих познаниями в спорте, истории, литературе и телевидении. Другой вариант – распределить задачи по группам разработчиков, принимая во внимание квалификацию их членов. С помощью оптимизации можно найти такой способ распределения, при котором все задачи будут решены в кратчайшие сроки.

Если дан длинный список сайтов, снабженных ключевыми словами, то было бы интересно найти среди них оптимальную группу, соответствующую ключевым словам, заданным пользователем. Такая группа должна состоять из сайтов, у которых мало общих ключевых слов, но в совокупности они должны отвечать максимально большому числу слов, указанных пользователем.

Упражнения

1. *Целевая функция группового путешествия.* Добавьте общее время полета из расчета $0,50 за минуту. Еще добавьте штраф $20, если кому-то приходится приезжать в аэропорт раньше 8 часов утра.
2. *Начальные точки отжига.* Результат имитации отжига сильно зависит от начальной точки. Постройте новую функцию оптимизации, которая начинала бы отжиг в разных точках и возвращала лучшее из найденных решений.
3. *Критерий остановки генетического алгоритма.* Описанная в этой главе функция выполняла фиксированное число итераций генетического алгоритма. Измените ее так, чтобы она прекращала работу, когда на протяжении 10 последовательных итераций не удается улучшить ни одно из найденных к этому моменту наилучших решений.

4. *Цены на билеты туда и обратно.* Функция, получающая данные с сайта Kayak, запрашивает рейсы только в одну сторону. Возможно, удастся сэкономить, купив билеты на рейс туда и обратно. Измените код так, чтобы запрашивать цены при покупке билета в обе стороны, и модифицируйте целевую функцию, чтобы она искала цену сразу на пару рейсов, а не суммировала цены на билеты в одну сторону.
5. *Сочетание студентов.* Предположим, что студенты выражают пожелания не по конкретной комнате, а по подбору соседа. Как бы вы представили решение в этом случае? Как могла бы выглядеть целевая функция?
6. *Штраф за величину угла.* Включите в целевую функцию для алгоритма размещения сети дополнительную плату, если угол между двумя линиями, исходящими из одного узла, слишком мал. (Подсказка: воспользуйтесь векторным произведением.)

6

Фильтрация документов

В этой главе мы покажем, как можно классифицировать документы по их содержимому. Это очень полезное применение искусственного интеллекта получает все большее распространение, и в первую очередь для фильтрации спама. Из-за легкодоступности и исключительной дешевизны отправки сообщений по электронной почте возникла серьезная проблема – любой человек, чей адрес попал в нечистоплотные руки, начинает получать непрошенную почту, а это затрудняет выделение действительных важных писем.

Разумеется, проблеме спама подвержена не только электронная почта. Веб-сайты, ставшие со временем более интерактивными, предлагают посетителям оставлять свои комментарии и даже создавать оригинальный контент. И это тоже оставляет возможность для спама. Публичные форумы, например Yahoo! Groups и Usenet, уже давно стали мишенями для размещения не относящихся к теме форума сообщений или рекламы сомнительной продукции. Теперь от той же напасти страдают сайты Википедии и блоги. При создании приложений, разрешающих что-то публиковать любому желающему, необходимо заранее подумать о том, как бороться со спамом.

Описанные в этой главе алгоритмы применимы не только для фильтрации спама. Они решают общую задачу обучения распознаванию того, принадлежит ли документ к той или иной категории, а стало быть, могут применяться и для не столь дурно пахнущих целей. Например, можно автоматически рассортировать поступающую в ваш ящик почту на письма, относящиеся к работе и к личной жизни. Или распознавать письма с запросом информации и автоматически направлять их лицу, способному дать наиболее компетентный ответ. В конце главы мы приведем пример автоматической классификации сообщений из RSS-канала.

Фильтрация спама

Первые попытки фильтрации спама полагались на классификаторы на основе правил, когда человек сам формулировал правила, в соответствии с которыми сообщение признавалось полезным или спамом. Обычно правила касались слишком большого числа заглавных букв, упоминания фармацевтической продукции или чрезмерно пестрой раскраски. Но довольно быстро выявился очевидный недостаток подобных классификаторов – спамеры изучили все правила и перестали явно выдавать себя, научившись обходить фильтры. И наоборот, люди, чьи престарелые родители так и не научились нажимать клавишу Caps Lock, обнаружили, что вполне нормальные сообщения от них классифицируются как спам.

У фильтров на основе правил есть и еще один минус. Признаки спама меняются в зависимости от того, откуда отправлено сообщение и кому оно адресовано. Ключевые слова, которые для одного пользователя, форума или раздела Википедии служат несомненным признаком спама, в другой ситуации считаются совершенно нормальными. Чтобы разрешить эту проблему, мы в данной главе рассмотрим программы, которые *обучаются* по мере того, как вы сообщаете им, что считать спамом, а что нет, причем это происходит как на начальной стадии, так и в процессе получения новых сообщений. Таким образом можно подготовить различные экземпляры и наборы данных для разных пользователей, групп и сайтов, каждый из которых будет по-своему уточнять, что такое спам.

Документы и слова

Классификатору, который мы построим, будут необходимы *признаки* для классификации различных образцов. Признаком можно считать любое свойство, относительно которого можно сказать, присутствует оно в образце или нет. Если классифицируются документы, то образцом считается документ, а признаками – встречающиеся в нем слова. Когда слова рассматриваются как признаки, мы предполагаем, что некоторые слова вероятнее встретить в спаме, чем в нормальных сообщениях. Именно это допущение лежит в основе большинство антиспамных фильтров. Но никто не заставляет ограничиваться только словами. В качестве признаков можно рассматривать пары слов, целые фразы и вообще все, о чем можно сказать, присутствует оно в документе или отсутствует.

Создайте новый файл docclass.py и включите в него функцию `getwords`, которая будет извлекать признаки из текста:

```
import re
import math
```

```
def getwords(doc):
  splitter=re.compile('\\W*')
  # Разбить на слова по небуквенным символам
  words=[s.lower( ) for s in splitter.split(doc)
         if len(s)>2 and len(s)<20]

  # Вернуть набор уникальных слов
  return dict([(w,1) for w in words])
```

Эта функция разбивает текст на слова по любому символу, отличному от буквы. Она оставляет только сами слова, преобразованные в нижний регистр.

Решить, что будет считаться признаками, – это самая сложная и самая важная задача. Признаки должны быть достаточно распространенными, чтобы встречаться часто, но не настолько распространенными, чтобы попадаться в каждом документе. Теоретически признаком может быть весь текст документа, но это почти всегда бессмысленно, разве что вы получаете одно и то же сообщение снова и снова. Другая крайность – объявить признаками отдельные символы. Но, поскольку они входят в каждое сообщение, то не позволят отделить зерна от плевел. Даже при выборе слов в качестве признаков возникает ряд вопросов: как выделять слова, какие знаки препинания в них включать и следует ли учитывать информацию в заголовке.

Еще один фактор, который нужно принимать во внимание, когда выбирается набор признаков, – насколько хорошо они смогут распределить документы по категориям. Например, приведенная выше функция `getwords` сокращает число признаков, приводя все слова к нижнему регистру. Это означает, что написанное заглавными буквами слово, встречающееся в начале предложения, ничем не отличается от такого же слова, написанного строчными буквами, в середине предложения. Но тем самым мы полностью игнорируем КРИКЛИВЫЙ стиль, характерный для многих спамных сообщений, а это мог бы быть важный классифицирующий признак спама. Альтернативной могло быть стать объявление признаком такой характеристики: более половины слов написаны заглавными буквами.

Как видите, при выборе признаков приходится идти на различные компромиссы и постоянно уточнять ранее принятые решения. Пока мы будем пользоваться простой функцией `getwords`, а в конце главы предложим ряд идей по совершенствованию механизма выделения признаков.

Обучение классификатора

Рассматриваемые в этой главе классификаторы способны обучаться тому, как надо классифицировать документы. Многие алгоритмы, описанные в этой книге, в частности нейронные сети из главы 4, обучаются, когда им предъявляют примеры правильных ответов. Чем больше документов и правильных способов классификации видит алгоритм,

тем лучше он классифицирует последующие документы. Кроме того, в начальный момент классификатор намеренно делают очень «робким», так чтобы его «уверенность в себе» повышалась по мере того, как он узнает, какие признаки важны для проведения различий.

Прежде всего нам понадобится класс для представления классификатора. Он инкапсулирует те факты, которым классификатор успел обучиться. У такой структуры есть важное достоинство – можно создавать разные экземпляры классификаторов, ориентированные на различных пользователей, группы или запросы, и обучать их в соответствии с потребностями конкретной группы. Создайте в файле docclass.py следующий класс `classifier`:

```
class classifier:
  def __init__(self,getfeatures,filename=None):
    # Счетчики комбинаций признак/категория
    self.fc={}
    # Счетчики документов в каждой категории
    self.cc={}
    self.getfeatures=getfeatures
```

В этом классе есть три переменных экземпляра: `fc`, `cc` и `getfeatures`. В переменной `fc` хранятся счетчики признаков при различных классификациях: например:

```
{'python': {'bad': 0, 'good': 6}, 'the': {'bad': 3, 'good': 3}}
```

Это означает, что слово the трижды появлялось в документах, классифицированных как плохие, и трижды – в документах, классифицированных как хорошие. Слово Python появлялось только в хороших документах.

Переменная `cc` – это словарь, в котором отражено, сколько раз применялась каждая классификация. Это необходимо для вычисления вероятности, о чем чуть позже. И последняя переменная экземпляра, `getfeatures`, – это функция, с помощью которой из классифицируемых образцов выделяются признаки. В нашем примере это функция `getwords`.

Методы класса не будут пользоваться этими словарями напрямую, поскольку это ограничило бы возможности для сохранения данных обучения – в файле, в базе или еще где-то. Создайте следующие вспомогательные функции для увеличения и получения счетчиков:

```
# Увеличить счетчик пар признак/категория
def incf(self,f,cat):
  self.fc.setdefault(f,{})
  self.fc[f].setdefault(cat,0)
  self.fc[f][cat]+=1

# Увеличить счетчик применений категории
def incc(self,cat):
  self.cc.setdefault(cat,0)
  self.cc[cat]+=1
```

```
# Сколько раз признак появлялся в данной категории
def fcount(self,f,cat):
  if f in self.fc and cat in self.fc[f]:
    return float(self.fc[f][cat])
  return 0.0

# Сколько образцов отнесено к данной категории
def catcount(self,cat):
  if cat in self.cc:
    return float(self.cc[cat])
  return 0

# Общее число образцов
def totalcount(self):
  return sum(self.cc.values( ))

# Список всех категорий
def categories(self):
  return self.cc.keys( )
```

Метод `train` принимает образец (в данном случае документ) и классификацию. С помощью функции `getfeatures` он выделяет из образца признаки. Затем он вызывает метод `incf`, чтобы увеличить счетчики для каждого признака в данной классификации. И наконец он увеличивает счетчик применений этой классификации:

```
def train(self,item,cat):
  features=self.getfeatures(item)
  # Увеличить счетчики для каждого признака в данной классификации
  for f in features:
    self.incf(f,cat)

  # Увеличить счетчик применений этой классификации
  self.incc(cat)
```

Убедиться в том, что класс работает правильно, можно, запустив интерпретатор Python и импортировав этот модуль:

```
$ python
>>> import docclass
>>> cl=docclass.classifier(docclass.getwords)
>>> cl.train('the quick brown fox jumps over the lazy dog','good')
>>> cl.train('make quick money in the online casino','bad')
>>> cl.fcount('quick','good')
1.0
>>> cl.fcount('quick','bad')
1.0
```

Сейчас нам был бы полезен метод, который сохраняет в классификаторе некоторые данные, полученные в ходе обучения, чтобы не приходилось каждый раз проводить обучение заново. Добавьте такую функцию в файл docclass.py:

```
def sampletrain(cl):
  cl.train('Nobody owns the water.','good')
  cl.train('the quick rabbit jumps fences','good')
```

```
cl.train('buy pharmaceuticals now','bad')
cl.train('make quick money at the online casino','bad')
cl.train('the quick brown fox jumps','good')
```

Вычисление вероятностей

Теперь у нас есть счетчики, показывающие, как сообщения были разнесены по категориям. Следующий шаг – превратить эти числа в вероятности. Вероятностью называется число от 0 до 1, показывающее частоту возникновения некоторого события. В данном случае мы вычисляем вероятность того, что слово принадлежит конкретной категории. Для этого нужно разделить количество вхождений данного слова в документ, отнесенный к этой категории, на общее число документов в той же категории.

Добавьте в класс `classifier` метод `fprob`:

```
def fprob(self,f,cat):
  if self.catcount(cat)==0: return 0
  # Общее число раз, когда данный признак появлялся в этой категории,
  # делим на количество образцов в той же категории
  return self.fcount(f,cat)/self.catcount(cat)
```

Эта величина называется *условной вероятностью*, обычно записывается в виде $Pr(A \mid B)$ и произносится так: «Вероятность A при условии B». Мы только что вычислили значение $Pr(\text{Слово} \mid \text{Классификация})$; то есть вероятность того, что некоторое слово появится в данной классификации.

Протестируем эту функцию в интерактивном сеансе:

```
>>> reload(docclass)
<module 'docclass' from 'docclass.py'>
>>> cl=docclass.classifier(docclass.getwords)
>>> docclass.sampletrain(cl)
>>> cl.fprob('quick','good')
0.66666666666666663
```

Как видите, слово quick входит всего в три документа, из которых два классифицированы как хорошие. Следовательно, вероятность того, что хороший документ будет содержать это слово, равна $Pr(\text{Quick} \mid \text{Хороший}) = 0{,}666\ (2/3)$.

Начинаем с разумной гипотезы

Метод `fprob` дает точный результат для тех признаков и классификаций, которые он уже видел, однако имеется небольшая проблема – использование лишь той информации, с которой метод уже сталкивался, делает его слишком чувствительным на ранних этапах обучения и по отношению к редко встречающимся словам. В нашем примере слово money (деньги) встретилось только в одном документе, который классифицирован как плохой, потому что содержит рекламу казино. Раз

это слово встретилось в одном плохом документе и не встречалось в хороших, то вероятность того, что оно может появиться в хороших документах, согласно функции `fprob` равна 0. Это, пожалуй, перебор, так как слово money вполне нейтральное, просто ему не повезло – угораздило первый раз встретиться в плохом документе. Было бы разумнее, если бы вероятность постепенно приближалась к нулю по мере того, как обрабатывается все больше и больше документов в той же категории.

Чтобы справиться с этой проблемой, мы выберем некую *предполагаемую вероятность*, которой будем пользоваться, когда информации о рассматриваемом признаке слишком мало. Хорошей отправной точкой может послужить вероятность 0,5. Еще нужно решить, какой вес приписать предполагаемой вероятности, – 1 означает, что вес предполагаемой вероятности равен одному слову. Взвешенная вероятность – это средневзвешенное значение, возвращаемое функцией `getprobability`, и предполагаемая вероятность.

В рассмотренном выше примере взвешенная вероятность слова money первоначально равна 0,5 во всех категориях. После того как классификатору был предъявлен один плохой документ и он узнал, что слово money отнесено к категории плохих, вероятность, что оно окажется плохим, возросла до 0,75. Это значение дает следующее вычисление:

```
(weight*assumedprob + count*fprob)/(count+weight)
= (1*1.0+1*0.5)/(1.0 + 1.0)
= 0.75
```

Добавьте метод `weightedprob` в класс `classifier`:

```
def weightedprob(self,f,cat,prf,weight=1.0,ap=0.5):
  # Вычислить текущую вероятность
  basicprob=prf(f,cat)

  # Сколько раз этот признак встречался во всех категориях
  totals=sum([self.fcount(f,c) for c in self.categories( )])

  # Вычислить средневзвешенное значение
  bp=((weight*ap)+(totals*basicprob))/(weight+totals)
  return bp
```

Протестируйте эту функцию в интерактивном сеансе. Перезагрузите модуль и повторно выполните метод `sampletrain`, поскольку при создании нового экземпляра класса вся накопленная за время обучения информация стирается:

```
>>> reload(docclass)
<module 'docclass' from 'docclass.pyc'>
>>> cl=docclass.classifier(docclass.getwords)
>>> docclass.sampletrain(cl)
>>> cl.weightedprob('money','good',cl.fprob)
0.25
>>> docclass.sampletrain(cl)
>>> cl.weightedprob('money','good',cl.fprob)
0.16666666666666666
```

Как видите, после повторного запуска `sampletrain` классификатор еще сильнее утвердился во мнении относительно вероятностей различных слов – из-за вклада предполагаемой вероятности.

Мы выбрали для предполагаемой вероятности значение 0,5 просто потому, что оно лежит посередине между 0 и 1. Однако не исключено, что у вас имеется более точная априорная информация даже для необученного классификатора. Например, человек, приступающий к обучению антиспамного фильтра, может позаимствовать вероятности у фильтров, обученных другими людьми. Он по-прежнему сможет настроить фильтр под себя, но при этом классификатор будет лучше обрабатывать редко встречающиеся слова.

Наивная классификация

Имея вероятности для слов, входящих в документ, вы должны выбрать какой-то способ комбинирования вероятностей отдельных слов для вычисления вероятности того, что документ в целом принадлежит данной категории. В этой главе мы рассмотрим два разных метода классификации. Оба работают в большинстве случаев, но несколько отличаются по качеству при решении конкретных задач. Предметом этого раздела будет *наивный байесовский классификатор*.

Слово «*наивный*» в данном контексте означает следующее: классификатор предполагает, что комбинируемые вероятности *независимы* друг от друга. Иными словами, вероятность того, что одно слово в документе относится к некоторой категории, никак не связана с вероятностями отнесения к той же категории других слов. Это предположение неверно, так как шансы, что документы, содержащие слово casino, содержат также и слово money, гораздо выше, чем шансы встретить слово money в документах о программировании на языке Python.

Следовательно, из-за ложного допущения о независимости нельзя использовать вероятность, вычисленную наивным байесовским классификатором, как истинную вероятность того, что документ принадлежит к некоторой категории. Однако можно *сравнить* результаты, полученные для разных категорий, и посмотреть, для какой из них вероятность оказалась самой большой. На практике, несмотря на ложность исходного предположения, этот метод классификации документов оказывается на удивление эффективным.

Вероятность для всего документа

Чтобы воспользоваться наивным байесовским классификатором, нужно сначала вычислить вероятность для документа в целом в рамках данной классификации. Поскольку мы предположили, что вероятности независимы, то для этого достаточно их перемножить.

Предположим, например, что слово Python встречается в 20% плохих документов, то есть

$$Pr(\text{Python} \mid \text{Плохой}) = 0{,}2,$$

а слово casino – в 80% плохих документов:

$$Pr(\text{Casino} \mid \text{Плохой}) = 0{,}8.$$

Тогда в предположении независимости вероятность появления обоих слов в плохом документе

$$Pr(\text{Python \& Casino} \mid \text{Плохой}) = 0{,}8 \times 0{,}2 = 0{,}16.$$

Стало быть, вычисление вероятности для документа в целом сводится к перемножению вероятностей отдельных встречающихся в нем слов.

В файле docclass.py создайте подкласс `naivebayes` класса `classifier` и включите в него метод `docprob`, который выделяет признаки (слова) и перемножает их вероятности:

```
class naivebayes(classifier):
  def docprob(self,item,cat):
    features=self.getfeatures(item)

    # Перемножить вероятности всех признаков
    p=1
    for f in features: p*=self.weightedprob(f,cat,self.fprob)
    return p
```

Теперь вы знаете, как вычислить вероятность $Pr(\text{Документ} \mid \text{Категория})$, но само по себе это не слишком полезно. Для классификации документов нужно знать, чему равно значение $Pr(\text{Категория} \mid \text{Документ})$. Иными словами, при условии наличия *конкретного* документа какова вероятность, что он попадает в данную категорию? К счастью, английский математик Томас Байес еще 250 лет назад понял, как это сделать.

Краткое введение в теорему Байеса

Теорема Байеса описывает соотношение между условными вероятностями. Обычно она записывается в виде

$$Pr(A \mid B) = Pr(B \mid A) \times Pr(A) \,/\, Pr(B)$$

В применении к нашему примеру эта формула принимает следующий вид:

$$Pr(\text{Категория} \mid \text{Документ}) = Pr(\text{Документ} \mid \text{Категория}) \times$$
$$\times Pr(\text{Категория}) \,/\, Pr(\text{Документ})$$

В предыдущем разделе мы показали, как вычислить $Pr(\text{Документ} \mid \text{Категория})$, но как быть с остальными двумя членами? Ничего сложного. $Pr(\text{Категория})$ – это вероятность попадания случайно выбранного документа в данную категорию, поэтому она просто равна числу документов из этой категории, поделенному на общее число документов.

Что касается вероятности $Pr(\text{Документ})$, то ее тоже можно было бы вычислить, но это лишний труд. Напомним, что конечный результат не

будет использоваться как истинная вероятность. Мы лишь по отдельности вычислим вероятности попадания в каждую категорию, а затем сравним их. Поскольку величина *Pr*(Документ) одна и та же для всех категорий, то ее можно попросту игнорировать.

Метод `prob` вычисляет вероятность попадания в категорию и возвращает произведение *Pr*(Документ | Категория) и *Pr*(Категория). Добавьте его в класс `naivebayes`:

```
def prob(self,item,cat):
  catprob=self.catcount(cat)/self.totalcount( )
  docprob=self.docprob(item,cat)
  return docprob*catprob
```

Протестируйте эту функцию и посмотрите, как изменяются числа для разных строк и категорий:

```
>>> reload(docclass)
<module 'docclass' from 'docclass.pyc'>
>>> cl=docclass.naivebayes(docclass.getwords)
>>> docclass.sampletrain(cl)
>>> cl.prob('quick rabbit','good')
0.15624999999999997
>>> cl.prob('quick rabbit','bad')
0.050000000000000003
```

По результатам обучения оказывается следующее: вероятность того, что фраза quick rabbit является частью хорошего документа, гораздо выше, чем плохого.

Выбор категории

Последним шагом построения наивного байесовского классификатора является выбор категории, к которой следует отнести новый образец. Простейший подход состоит в том, чтобы вычислить вероятности попадания образца в каждую категорию и выбрать ту, для которой вероятность максимальна. Если вам просто нужно определить самое подходящее место, куда положить образец, то такая стратегия годится, но во многих приложениях категории нельзя считать равноценными, а иногда лучше, чтобы классификатор признал, что не знает точного ответа, и не пытался выдать за ответ категорию, для которой вероятность оказалась лишь чуть выше, чем для прочих.

В случае фильтрации спама гораздо важнее избежать классификации хороших сообщений как спама, чем отловить все спамные сообщения без исключения. С редкими появлениями спамных сообщений в своем почтовом ящике вы готовы будете смириться, а вот помещение важного письма в ящик «Сомнительные» легко может остаться незамеченным. Если же вы внимательно просматриваете этот ящик в поисках важных сообщений, то к чему тогда вообще антиспамный фильтр?

Для решения этой проблемы можно задать для каждой категории минимальное пороговое значение. Чтобы новый образец был отнесен

к некоторой категории, его вероятность должна быть по крайней мере на эту величину больше вероятности попадания в любую другую категорию. Эта величина называется *порогом*. Для фильтрации спама порог отнесения к категории плохих документов можно было бы установить равным 3. Это означает, что вероятность оказаться хорошим документом должна быть хотя бы в 3 раза выше, чем плохим. Для категории «Хороший» задается порог 1, то есть документ будет признаваться хорошим, если вероятность этого хоть чуть-чуть выше, чем для категории «Плохой». Все сообщения, для которых вероятность оказаться «плохим» выше, чем «хорошим», но менее чем в 3 раза, классифицируются как «Неизвестный».

Для задания порогов добавьте в класс `classifier` новую переменную экземпляра, изменив метод инициализации:

```
def __init__(self,getfeatures):
  classifier.__init__(self,getfeatures)
  self.thresholds={}
```

Добавьте два простых метода для установки и получения значений, причем по умолчанию пусть возвращается значение 1,0:

```
def setthreshold(self,cat,t):
  self.thresholds[cat]=t

def getthreshold(self,cat):
  if cat not in self.thresholds: return 1.0
  return self.thresholds[cat]
```

Теперь все готово для написания метода `classify`. Он вычисляет вероятность для каждой категории, ищет среди них максимальную и проверяет, отличается ли она от следующей по порядку более чем в пороговое число раз. Если такую категорию найти не удается, то метод просто возвращает значение по умолчанию. Добавьте в класс `classifier` такой код:

```
def classify(self,item,default=None):
  probs={}
  # Найти категорию с максимальной вероятностью
  max=0.0
  for cat in self.categories( ):
    probs[cat]=self.prob(item,cat)
    if probs[cat]>max:
      max=probs[cat]
      best=cat

  # Убедиться, что найденная вероятность больше чем threshold*следующая по
  # величине
  for cat in probs:
    if cat==best: continue
    if probs[cat]*self.getthreshold(best)>probs[best]: return default
  return best
```

Все сделано! Вы построили законченную систему классификации документов. Ее можно обобщить на классификацию других образцов, написав методы для выделения признаков. Протестируйте свой классификатор в интерактивном сеансе:

```
>>> reload(docclass)
<module 'docclass' from 'docclass.pyc'>
>>> cl=docclass.naivebayes(docclass.getwords)
>>> docclass.sampletrain(cl)
>>> cl.classify('quick rabbit',default='unknown')
'good'
>>> cl.classify('quick money',default='unknown')
'bad'
>>> cl.setthreshold('bad',3.0)
>>> cl.classify('quick money',default='unknown')
'unknown'
>>> for i in range(10): docclass.sampletrain(cl)
...
>>> cl.classify('quick money',default='unknown')
'bad'
```

Можете изменить пороги и посмотреть, как это отразится на результатах. Некоторые подключаемые модули фильтрации спама позволяют пользователям подстраивать пороги, если в ящике «Входящие» остается слишком много спама или, напротив, хорошие сообщения часто попадают в спам. Для разных приложений фильтрации документов пороги задаются по-разному; иногда все категории считаются равноценными, а порой отнесение к категории «неизвестные» недопустимо.

Метод Фишера

Метод Фишера, названный по имени Р. А. Фишера (R. A. Fisher), – это альтернативный метод классификации, который дает очень точные результаты, особенно применительно к фильтрации спама. Он используется в подключаемом к программе Outlook фильтре *SpamBayes*, который написан на языке Python. В отличие от наивной байесовской фильтрации, когда для вычисления вероятности всего документа перемножаются вероятности отдельных признаков, по методу Фишера вычисляется вероятность отнесения к той или иной категории для каждого признака документа, после чего эти вероятности комбинируются и проверяется, насколько получившееся множество похоже на случайное. Хотя этот метод более сложен, с ним стоит ознакомиться, так как он обеспечивает бóльшую гибкость при настройке параметров классификации.

Вероятности отнесения признаков к категориям

В наивном байесовском фильтре, который обсуждался выше, мы комбинировали вероятности *Pr*(Признак| Категория) для получения полной вероятности документа, а затем сравнивали их. В этом разделе мы начнем с вычисления вероятности попадания документа в некоторую категорию при условии наличия у этого документа конкретного признака, то есть с вычисления величины *Pr*(Категория | Признак). Если слово casino встречается в 500 документах, из которых 499 плохие, то оценка casino относительно категории «Плохой» будет очень близка к 1.

Обычно значение *Pr*(Категория | Признак) вычисляется так:

(Количество документов с данным признаком в этой категории) / (Общее количество документов с данным признаком)

Эта формула не учитывает, что для одной категории могло быть предъявлено гораздо больше документов, чем для другой. Если у вас есть много хороших документов и совсем мало плохих, то у слова, встречающегося во всех плохих документах, будет высокая вероятность оказаться плохим, пусть даже сообщение с равным успехом могло бы быть и хорошим. Методы работают лучше в предположении, что в будущем будет получено равное количество документов из каждой категории, поскольку это позволяет в полной мере воспользоваться признаками, по которым различаются категории.

Чтобы выполнить такую нормализацию, в методе Фишера вычисляются три величины:

- `clf` = *Pr*(Признак | Категория) для этой категории
- `freqsum` = сумма *Pr*(Признак | Категория) для всех категорий
- `cprob` = `clf` / (`clf` + `nclf`)

Создайте в файле docclass.py новый подкласс `fisherclassifier` класса `classifier` и включите в него такой метод:

```
class fisherclassifier(classifier):
  def cprob(self,f,cat):
    # Частота появления данного признака в данной категории
    clf=self.fprob(f,cat)
    if clf==0: return 0

    # Частота появления данного признака во всех категориях
    freqsum=sum([self.fprob(f,c) for c in self.categories( )])

   # Вероятность равна частоте появления в данной категории, поделенной на
   # частоту появления во всех категориях
   p=clf/(freqsum)

   return p
```

Эта функция возвращает вероятность того, что образец с указанным признаком принадлежит указанной категории, в предположении, что в каждой категории будет одинаковое число образцов. Можете посмотреть в интерактивном сеансе, какие получаются цифры:

```
>>> reload(docclass)
>>> cl=docclass.fisherclassifier(docclass.getwords)
>>> docclass.sampletrain(cl)
>>> cl.cprob('quick','good')
0.57142857142857151
>>> cl.cprob('money','bad')
1.0
```

Как видите, документы, содержащие слово casino с вероятностью 0,9, классифицируются как спам. Это соответствует данным обучения, но

остается все та же проблема – недостоверность для редко встречающихся слов, когда вероятность может оказаться сильно завышенной. Поэтому, как и раньше, лучше пользоваться взвешенной вероятностью, начав со значения 0,5 и постепенно улучшая точность по мере обучения класса:

```
>>> cl.weightedprob('money','bad',cl.cprob)
0.75
```

Комбинирование вероятностей

Теперь надо из вероятностей отдельных признаков получить полную вероятность. Теоретически их можно просто перемножить и сравнить результат для данной категории с результатами для других категорий. Конечно, из-за того что признаки не являются независимыми, истинной вероятности мы так не получим, но все же этот алгоритм работает гораздо лучше байесовского классификатора, разработанного в предыдущем разделе. Метод Фишера дает существенно более точную оценку вероятности, что может оказаться крайне полезным в отчетах или при решении вопроса об уровне отсечки.

По методу Фишера все вероятности перемножаются, затем берется натуральный логарифм (`math.log` в Python) и результат умножается на –2. Добавьте следующий метод в класс `fisherclassifier`:

```
def fisherprob(self,item,cat):
  # Перемножить все вероятности
  p=1
  features=self.getfeatures(item)
  for f in features:
    p*=(self.weightedprob(f,cat,self.cprob))

  # Взять натуральный логарифм и умножить на -2
  fscore=-2*math.log(p)

  # Для получения вероятности пользуемся обратной функцией хи-квадрат
  return self.invchi2(fscore,len(features)*2)
```

Фишер доказал, что если бы вероятности были независимы и случайны, то результат этого вычисления подчинялся бы *распределению хи-квадрат*. Следует ожидать, что образец, не принадлежащий некоторой категории, будет содержать слова с различными вероятностями признаков для данной категории (их распределение было бы случайным), а в образце, принадлежащем категории, будет много признаков с высокими вероятностями. Подав результат вычисления по формуле Фишера на вход *обратной функции хи-квадрат*, мы получаем вероятность того, что случайный набор вероятностей вернет такое большое число.

Добавьте обратную функцию хи-квадрат в класс `fisherclassifier`:

```
def invchi2(self,chi,df):
  m = chi / 2.0
  sum = term = math.exp(-m)
```

```
    for i in range(1, df//2):
      term *= m / i
      sum += term
    return min(sum, 1.0)
```

Протестируйте метод Фишера и посмотрите, как он оценивает различные строки:

```
>>> reload(docclass)
>>> cl=docclass.fisherclassifier(docclass.getwords)
>>> docclass.sampletrain(cl)
>>> cl.cprob('quick','good')
0.57142857142857151
>>> cl.fisherprob('quick rabbit','good')
0.78013986588957995
>>> cl.fisherprob('quick rabbit','bad')
0.35633596283335256
```

Как видите, результаты всегда оказываются в диапазоне от 0 до 1. Сами по себе они являются хорошей оценкой того, насколько точно документ соответствует категории. Поэтому классификатор можно сделать более изощренным.

Классификация образцов

Значения, возвращенные функцией `fisherprob`, можно использовать для классификации. Вместо того чтобы задавать мультипликативные пороги, как в байесовском фильтре, можно определить нижние границы для каждой классификации. Тогда классификатор вернет максимальное значение, укладывающееся в эти границы. Для антиспамного фильтра минимальную границу для категории «Плохой» можно задать достаточно высокой, например 0,6. А для категории «Хороший» можно задать гораздо меньшую границу, скажем 0,2. Это уменьшает шансы попадания в эту категорию хороших сообщений, но допускает «просачивание» небольшого числа спамных сообщений в ящик для входящей почты. Любое сообщение, для которого оценка для категории «Хороший» меньше 0,2, а для категории «Плохой» меньше 0,6, классифицируется как «Неизвестный».

Создайте в классе `fisherclassifier` метод `init`, инициализирующий еще одну переменную экземпляра для хранения отсечек:

```
def __init__(self,getfeatures):
  classifier.__init__(self,getfeatures)
  self.minimums={}
```

Добавьте методы получения и установки значений, считая, что по умолчанию принимается значение 0:

```
def setminimum(self,cat,min):
  self.minimums[cat]=min
```

```
def getminimum(self,cat):
  if cat not in self.minimums: return 0
  return self.minimums[cat]
```

И наконец добавьте метод вычисления вероятностей для каждой категории и определения наилучшего результата, превышающего заданный минимум:

```
def classify(self,item,default=None):
  # Цикл для поиска наилучшего результата
  best=default
  max=0.0
  for c in self.categories( ):
    p=self.fisherprob(item,c)
    # Проверяем, что значение больше минимума
    if p>self.getminimum(c) and p>max:
      best=c
      max=p
  return best
```

Испытаем классификатор по методу Фишера на тестовых данных. Введите в интерактивном сеансе следующие команды:

```
>>> reload(docclass)
<module 'docclass' from 'docclass.py'>
>>> docclass.sampletrain(cl)
>>> cl.classify('quick rabbit')
'good'
>>> cl.classify('quick money')
'bad'
>>> cl.setminimum('bad',0.8)
>>> cl.classify('quick money')
'good'
>>> cl.setminimum('good',0.4)
>>> cl.classify('quick money')
>>>
```

Результаты аналогичны полученным от наивного байесовского классификатора. Но считается, что на практике классификатор Фишера фильтрует спам лучше; правда, убедиться в этом на небольшом обучающем наборе вряд ли получится. Какой классификатор использовать – зависит от приложения. Заранее не скажешь, что будет работать лучше и какие выбирать отсечки. Впрочем, приведенный код позволяет поэкспериментировать с обоими алгоритмами, задавая разные параметры.

Сохранение обученных классификаторов

В реальном приложении маловероятно, что обучение и классификацию удастся полностью провести в рамках одного сеанса. Если классификатор – часть веб-приложения, то, наверное, вы захотите сохранить результаты обучения, проведенного пользователем, и восстановить их, когда пользователь придет в следующий раз.

Использование SQLite

В этом разделе мы покажем, как сохранить результаты обучения классификатора в базе данных, в данном случае SQLite. Если у вашего приложения много пользователей, одновременно обучающих и опрашивающих классификатор, то, вероятно, имеет смысл хранить счетчики в базе. SQLite – это та самая СУБД, с которой мы работали с главе 4. Если вы этого еще не сделали, загрузите и установите пакет pysqlite; о том, как это сделать, см. приложение А. Доступ к SQLite из Python организуется так же, как к другим СУБД, так что серьезных затруднений у вас не возникнет.

Чтобы импортировать pysqlite, включите в начало файла docclass.py следующее предложение:

```
from pysqlite2 import dbapi2 as sqlite
```

Код, приведенный в этом разделе, предназначен для замены словарей, которые сейчас имеются в классе `classifier`, постоянным хранилищем данных. Добавьте в класс `classifier` метод для открытия базы данных и создания необходимых таблиц. Структура таблиц повторяет структуру заменяемых ими словарей:

```
def setdb(self,dbfile):
  self.con=sqlite.connect(dbfile)
  self.con.execute('create table if not exists fc(feature,category,count)')
  self.con.execute('create table if not exists cc(category,count)')
```

Если вы собираетесь адаптировать классификатор для другой базы данных, то, возможно, потребуется изменить предложения `create table` в соответствии с диалектом SQL в используемой СУБД.

Необходимо заменить вспомогательные методы получения и увеличения счетчиков:

```
def incf(self,f,cat):
  count=self.fcount(f,cat)
  if count==0:
    self.con.execute("insert into fc values ('%s','%s',1)"
                     % (f,cat))
  else:
    self.con.execute(
      "update fc set count=%d where feature='%s' and category='%s'"
      % (count+1,f,cat))

def fcount(self,f,cat):
  res=self.con.execute(
    'select count from fc where feature="%s" and category="%s"'
   %(f,cat)).fetchone( )
  if res==None: return 0
  else: return float(res[0])

def incc(self,cat):
  count=self.catcount(cat)
  if count==0:
```

```
      self.con.execute("insert into cc values ('%s',1)" % (cat))
    else:
      self.con.execute("update cc set count=%d where category='%s'"
                        % (count+1,cat))

  def catcount(self,cat):
    res=self.con.execute('select count from cc where category="%s"'
                         %(cat)).fetchone( )
    if res==None: return 0
    else: return float(res[0])
```

Необходимо также заменить методы для получения списка всех категорий и общего числа документов:

```
  def categories(self):
    cur=self.con.execute('select category from cc');
    return [d[0] for d in cur]

  def totalcount(self):
    res=self.con.execute('select sum(count) from cc').fetchone( );
    if res==None: return 0
    return res[0]
```

Наконец по завершении обучения нужно добавить вызов `commit`, чтобы сохранить обновленные счетчики. Добавьте следующую строку в конец метода `train`:

```
  self.con.commit( )
```

Вот и все! После инициализации классификатора следует вызвать метод `setdb`, указав имя файла базы данных. Данные, полученные в ходе обучения, автоматически сохраняются и могут быть использованы кем угодно. Можно даже воспользоваться данными из одного классификатора, чтобы выполнить классификацию другого типа:

```
>>> reload(docclass)
<module 'docclass' from 'docclass.py'>
>>> cl=docclass.fisherclassifier(docclass.getwords)
>>> cl.setdb('test1.db')
>>> docclass.sampletrain(cl)
>>> cl2=docclass.naivebayes(docclass.getwords)
>>> cl2.setdb('test1.db')
>>> cl2.classify('quick money')
u'bad'
```

Фильтрация блогов

Для тестирования классификатора на реальных данных и демонстрации различных способов использования можно применить его к записям из блогов или других RSS-каналов. Для этого вам потребуется библиотека Universal Feed Parser, с которой мы работали в главе 3. Если вы еще не скачали ее, зайдите на сайт *http://feedparser.org*. Дополнительную информацию об установке Feed Parser см. в приложении А.

Хотя в записях блога спама может и не быть, но многие блоги содержат как интересующие вас заметки, так и не представляющие никакого интереса. Например, иногда вы хотите читать только материалы из какой-то одной категории или написанные определенным автором, хотя часто ситуация бывает и сложнее. Как и раньше, можно задать правила, описывающие, что именно представляет для вас интерес, а что нет, – например, при чтении блога, посвященного электронным устройствам, вы хотите пропускать все записи, содержащие фразу cell phone (сотовый телефон). Но гораздо проще воспользоваться классификатором, который сам выработает такие правила.

Классификация записей в RSS-каналах удобна тем, что можно воспользоваться каким-нибудь инструментом поиска по блогам, например Google Blog Search, и подготовить результаты поиска для подачи на вход программе чтения канала. Многие делают так для отслеживания товаров, других интересных для них вещей и даже собственных имен. Однако при этом вы будете находить заспамленные или бесполезные блоги, создатели которых пытаются сделать деньги на трафике.

В примере, описываемом ниже, вы можете взять любой канал, хотя надо иметь в виду, что во многих каналах слишком мало записей для эффективного обучения. Мы остановились на результатах поиска по слову Python, возвращенных Google Blog Search и представленных в формате RSS. Загрузить их можно со страницы *http://kiwitobes.com/feeds/python_search.xml*.

Создайте файл feedfilter.py и включите в него такой код:

```
import feedparser
import re

# Принимает URL канала блога и классифицирует записи
def read(feed,classifier):
  # Получить и в цикле перебрать записи
  f=feedparser.parse(feed)
  for entry in f['entries']:
    print
    print '-----'
    # Распечатать содержимое записи
    print 'Заголовок: '+entry['title'].encode('utf-8')
    print 'Автор:     '+entry['publisher'].encode('utf-8')
    print
    print entry['summary'].encode('utf-8')

    # Объединить весь текст с целью создания одного образца для
    # классификатора
    fulltext='%s\n%s\n%s' % (entry['title'],entry['publisher'],
    entry['summary'])

    # Напечатать наилучшую гипотезу о текущей категории
    print 'Гипотеза: '+str(classifier.classify(fulltext))
```

```
    # Попросить пользователя ввести правильную категорию и обучиться
    # на его ответе
    cl=raw_input('Введите категорию: ')
    classifier.train(fulltext,cl)
```

Эта функция перебирает все записи и с помощью классификатора вырабатывает наилучшую гипотезу о принадлежности к той или иной категории. Она сообщает об этой гипотезе пользователю и просит ввести правильную категорию. Поначалу классификатор вырабатывает гипотезы случайным образом, но со временем его решения должны улучшаться.

Построенный классификатор по природе своей весьма общий. Мы воспользовались в качестве примера фильтрацией спама, чтобы объяснить все детали кода, но, в принципе, категории могут быть любыми. Так, в файле python_search.xml можно выделить четыре категории: о языке программирования, о комедийном шоу «Monty Python», о питонах и обо всем остальном. Попробуйте запустить интерактивный фильтр в сеансе интерпретатора, обучить классификатор и передать его программе `feedfilter`:

```
>>> import feedfilter
>>> cl=docclass.fisherclassifier(docclass.getwords)
>>> cl.setdb('python_feed.db') # Только если реализовали интерфейс с SQLite
>>> feedfilter.read('python_search.xml',cl)
-----
Заголовок: My new baby boy!
Автор:     Shetan Noir, the zombie belly dancer! - MySpace Blog
This is my new baby, Anthem. He is a 3 and half month old ball <b>python</b>,
orange shaded normal pattern. I have held him about 5 times since I brought
him home tonight at 8:00pm...
Гипотеза: None
Введите категорию: snake
-----
Заголовок: If you need a laugh...
Автор:     Kate&#39;s space
Even does 'funny walks' from Monty <b>Python</b>. He talks about all the ol'
Гипотеза: snake
Введите категорию: monty
-----
Заголовок: And another one checked off the list..New pix comment ppl
Автор:     And Python Guru - MySpace Blog
Now the one of a kind NERD bred Carplot male is in our possesion. His name is
Broken (not because he is sterile) lol But check out the pic and leave one
Гипотеза: snake
Введите категорию: snake
```

Вы увидите, что со временем гипотезы улучшаются. Примеров о змеях довольно мало, поэтому классификатор часто допускает для них ошибки, тем более что в этой категории есть еще две подкатегории: домашние животные и мода. Закончив курс обучения, вы можете получить

вероятности для указанного признака – как вероятность слова при условии категории, так и вероятность категории при условии слова:

```
>>> cl.cprob('python','prog')
0.33333333333333331
>>> cl.cprob('python','snake')
0.33333333333333331
>>> cl.cprob('python','monty')
0.33333333333333331
>>> cl.cprob('eric','monty')
1.0
>>> cl.fprob('eric','monty')
0.25
```

Вероятности для слова python распределены равномерно, поскольку оно встречается в каждой записи. Слово Eric встречается в 25% записей, относящихся к шоу «Monty Python», и ни разу не встречается в других записях. Следовательно, вероятность этого слова при условии категории равно 0,25, а вероятность категории при условии слова равна 1,0.

Усовершенствование алгоритма обнаружения признаков

Во всех рассмотренных до сих пор примерах функция создания списка признаков просто разбивает текст на слова по символам, отличным от букв и цифр. Кроме того, она преобразует слова в нижний регистр, поэтому пропадает возможность обнаружить чрезмерное количество слов, написанных заглавными буквами. Ситуацию можно улучшить несколькими способами:

- Не считая слова, записанные заглавными и строчными буквами, совершенно различными, признать наличие большого количества «кричащих» слов признаком.
- Использовать помимо отдельных слов еще и словосочетания.
- Собирать дополнительную метаинформацию, например, о том, кто отправил письмо или в какую категорию была помещена запись блога, и помечать, что это именно метаданные.
- Сохранять URL и числа неизменными.

Не забывайте, что недостаточно просто выбрать более специфичные признаки. Чтобы классификатор мог воспользоваться признаком, последний должен встречаться во многих документах.

Класс `classifier` может принять в качестве `getfeatures` любую функцию и применять ее к предъявляемым образцам. Он ожидает лишь, что функция вернет список или словарь выделенных из образца признаков. В силу общности природы классификатора вы можете написать функцию, которая работает с более сложными типами, чем строки. Например, для классификации записей в блоге можно придумать функцию,

которая принимает всю запись целиком, а не только извлеченный из нее текст, и вставляет аннотации о том, откуда взялось каждое слово. Можно также выбирать пары слов из тела текста и отдельные слова из темы. По всей видимости, разбивать на части поле `creator` бессмысленно, так как сообщения от некоего «Джона Смита» ничего не скажут о сообщениях какого-нибудь другого «Джона».

Добавьте в файл feedfilter.py следующую функцию выделения признаков. Обратите внимание, что она ожидает на входе всю полученную из канала запись, а не просто строку:

```
def entryfeatures(entry):
  splitter=re.compile('\\W*')
  f={}

  # Извлечь и аннотировать слова из заголовка
  titlewords=[s.lower( ) for s in splitter.split(entry['title'])
  if len(s)>2 and len(s)<20]
  for w in titlewords: f['Title:'+w]=1

  # Извлечь слова из резюме
  summarywords=[s.lower( ) for s in splitter.split(entry['summary'])
  if len(s)>2 and len(s)<20]

  # Подсчитать количество слов, написанных заглавными буквами
  uc=0
  for i in range(len(summarywords)):
    w=summarywords[i]
    f[w]=1
    if w.isupper( ): uc+=1

    # Выделить в качестве признаков пары слов из резюме
    if i<len(summarywords)-1:
      twowords=' '.join(summarywords[i:i+1])
      f[twowords]=1

  # Оставить информацию об авторе без изменения
  f['Publisher:'+entry['publisher']]=1

  # UPPERCASE - специальный признак, описывающий степень "крикливости"
  if float(uc)/len(summarywords)>0.3: f['UPPERCASE']=1

  return f
```

Эта функция извлекает слова из заголовка и резюме так же, как написанная ранее функция `getwords`. Все слова из заголовка она рассматривает как отдельные признаки. Слова из резюме тоже помещаются в список слов, но дополнительно признаками считаются пары соседних слов. Информация об авторе трактуется как неделимый признак. Напоследок функция подсчитывает количество слов, написанных заглавными буквами. Если их доля больше 30% от общего числа слов, то функция добавляет в список специальный признак `UPPERCASE`. В отличие

от правила, которое говорит, что написанные заглавными буквами слова означают нечто особенное, мы просто включаем еще один признак, который можно использовать для обучения классификатора. В некоторых случаях классификатор может решить, что этот признак бесполезен для распознавания категории документа.

Если вы хотите применить новую версию совместно с функцией `filterfeed`, то последнюю придется изменить, так чтобы она передавала записи непосредственно классификатору, а не функции `fulltext`. Достаточно внести следующие модификации в конце ее кода:

```
# Напечатать наилучшую гипотезу о текущей категории
print 'Гипотеза: '+str(classifier.classify(entry))

# Попросить пользователя ввести правильную категорию и обучиться
# на его ответе
cl=raw_input('Enter category: ')
classifier.train(entry,cl)
```

Теперь можно инициализировать классификатор, так чтобы для выделения признаков он пользовался функцией `entryfeatures`:

```
>>> reload(feedfilter)
<module 'feedfilter' from 'feedfilter.py'>
>>> cl=docclass.fisherclassifier(feedfilter.entryfeatures)
>>> cl.setdb('python_feed.db') # Только если используется версия для СУБД
>>> feedfilter.read('python_search.xml',cl)
```

С признаками можно еще много чего сделать. Построенный базовый каркас позволяет определить функцию выделения признаков и настроить классификатор для работы с ней. Он будет классифицировать любой предъявленный образец при условии, что написанная функция способна выделить из него признаки.

Использование службы Akismet

Разговор о *службе Akismet* уведет нас немного в сторону от изучения алгоритмов классификации текстов, но для некоторого класса приложений эта служба позволяет решить задачу фильтрации спама с минимальными усилиями и избавить вас от необходимости строить собственный классификатор.

Первоначально Akismet была подключаемым модулем к системе ведения блогов WordPress, который позволял пользователям извещать о спаме в своих блогах и отфильтровывал новые сообщения, если они были похожи на те, что другие люди определили как спам. Теперь API открыт, и вы можете послать службе Akismet произвольную строку и спросить, считает ли она ее спамом.

Первым делом вам понадобится ключ для работы с API Akismet, который можно получить на сайте *http://akismet.com*. Для вызова API Akismet применяются обычные HTTP-запросы, и существуют интерфейсные библиотеки на разных языках программирования. В этом

разделе мы будем пользоваться библиотекой, доступной по адресу *http://kemayo.wordpress.com/2005/12/02/akismet-py*. Скачайте файл akismet.py и поместите его в папку, где находятся все библиотеки Python.

Работать с этим API очень просто. Создайте новый файл akismettest.py и включите в него такую функцию:

```
import akismet

defaultkey = "ВАШ_КЛЮЧ"
pageurl="http://yoururlhere.com"
defaultagent="Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.0.7) "
defaultagent+="Gecko/20060909 Firefox/1.5.0.7"

def isspam(comment,author,ipaddress,
           agent=defaultagent,
           apikey=defaultkey):
  try:
    valid = akismet.verify_key(apikey,pageurl)
    if valid:
      return akismet.comment_check(apikey,pageurl,
        ipaddress,agent,comment_content=comment,
        comment_author_email=author,comment_type="comment")
    else:
      print 'Недействительный ключ'
    return False
  except akismet.AkismetError, e:
    print e.response, e.statuscode
    return False
```

Теперь у вас есть метод, который можно вызвать, передав любую строку, и посмотреть, похожа ли она на встречавшиеся в комментариях из блогов. Выполните в интерактивном сеансе такие команды:

```
>>> import akismettest
>>> msg='Make money fast! Online Casino!'
>>> akismettest.isspam(msg,'spammer@spam.com','127.0.0.1')
True
```

Поэкспериментируйте с разными именами пользователей, агентами и IP-адресами и посмотрите, как будут меняться результаты.

Поскольку служба Akismet предназначена в первую очередь для фильтрации спамных комментариев в блогах, то с другими типами документов, например с почтовыми сообщениями, она может справляться не так хорошо. Кроме того, в отличие от классификатора, она не позволяет подстраивать параметры и не сообщает о ходе вычислений, приведших к ответу. Однако спамные комментарии она обрабатывает очень точно, поэтому имеет смысл применить ее в приложениях, где встречаются похожие разновидности спама. Ведь Akismet располагает для сравнения куда бóльшим набором документов, чем сможете собрать вы.

Альтернативные методы

Оба рассмотренных в данной главе классификатора – это примеры *обучения с учителем*, когда программе предъявляются правильные результаты и она постепенно улучшает качество вырабатываемых гипотез. Рассмотренная в главе 4 *искусственная нейронная сеть*, которая взвешивала результаты поиска на предмет их ранжирования, – еще один пример обучения с учителем. Нейронную сеть можно было бы адаптировать для решения задач из этой главы, если интерпретировать признаки как входные сигналы, а выходные сигналы сопоставить всем возможным классификациям. Аналогично, к задачам из этой главы применим *метод опорных векторов*, который описывается в главе 9.

Для классификации документов используется именно байесовская фильтрация просто потому, что она требует гораздо меньше вычислительных ресурсов, чем прочие методы. В почтовом сообщении могут быть сотни и даже тысячи слов, а для простого обновления счетчиков требуется куда меньше памяти и процессорного времени, чем для обучения нейронной сети такого размера. Как было показано, все это можно сделать в рамках базы данных. В зависимости от необходимой скорости обучения и получения ответа, а также от окружения, нейронная сеть иногда может оказаться приемлемой альтернативой. Но сложность нейронной сети препятствует какому-либо взаимодействию с ней. В этой главе вы видели, что вычисленные вероятности слов легкодоступны и известно, какой вклад они вносят в окончательную оценку. А для весов связей между нейронами сети нет столь же простой интерпретации.

С другой стороны, у нейронных сетей и машин опорных векторов есть одно весомое преимущество над описанными выше классификаторами: они могут улавливать более сложные взаимосвязи между подаваемыми на вход признаками. В байесовском классификаторе у каждого признака имеется вероятность принадлежности к каждой категории, и для получения полной вероятности они комбинируются. В нейронной сети вероятность признака может изменяться в зависимости от наличия или отсутствия других признаков. Например, вы хотели бы блокировать спам от онлайновых казино, но интересуетесь ставками на бегах; тогда слово casino следует считать плохим, если еще где-то в сообщении не встречается слово horse (лошадь). Наивный байесовский классификатор не способен уловить такую перекрестную зависимость, а нейронная сеть может.

Упражнения

1. *Переменные предполагаемые вероятности.* Измените класс `classifier` так, чтобы он поддерживал разные предполагаемые вероятности для различных признаков. Измените метод `init` так, чтобы

он принимал другой классификатор и начинал работу со значениями предполагаемых вероятностей, лучшими нежели 0,5.

2. *Вычисление Pr(Документ)*. В наивном байесовском классификаторе мы опустили вычисление *Pr*(Документ), поскольку этот постоянный множитель не имел значения при сравнении вероятностей. В тех случаях, когда признаки независимы, им можно воспользоваться для вычисления полной вероятности. Как бы вы подошли к вычислению *Pr*(Документ)?
3. *Почтовый фильтр для протокола POP-3*. В состав дистрибутива Python входит библиотека poplib для скачивания почтовых сообщений с сервера. Напишите сценарий, который скачивает сообщения и пытается их классифицировать. Какие признаки можно выделить в почтовом сообщении и как построить для них функцию извлечения?
4. *Фразы произвольной длины*. В этой главе мы показали, как извлекать одиночные слова и пары слов. Сделайте механизм выделения признаков конфигурируемым, чтобы он мог выделять признаки, состоящие из соседних слов, причем максимальное количество слов является параметром.
5. *Сохранение IP-адресов*. IP-адреса, номера телефонов и другая числовая информация может оказаться полезной при идентификации спама. Измените функцию выделения признаков так, чтобы она возвращала такие элементы в качестве признаков. (Хотя компоненты IP-адреса разделяются точками, от точек в конце предложений необходимо избавляться.)
6. *Другие виртуальные признаки*. Для классификации документов могут оказаться полезны и другие виртуальные признаки, помимо `UPPERCASE`. Например, интересно выделять очень длинные документы или документы с большим количеством длинных слов. Можете вы придумать еще какие-нибудь подобные характеристики?
7. *Классификатор на базе нейронной сети*. Модифицируйте нейронную сеть из главы 4 так, чтобы ее можно было применить к классификации документов. Сравните результаты. Напишите программу, которая классифицирует и обучается несколько тысяч раз. Замерьте время работы каждого алгоритма. Что у вас получилось?

7

Моделирование с помощью деревьев решений

Выше вы ознакомились с различными автоматическими классификаторами, а в этой главе мы продолжим эту тему и поговорим об очень полезном методе, который называется *обучением деревьев решений*. В отличие от других классификаторов, модели, порождаемые деревьями решений, легко поддаются интерпретации. Список чисел, которые выдает байесовский классификатор, говорит об относительной важности каждого слова, но для получения окончательного результата необходимо произвести вычисления. Интерпретировать результаты, вырабатываемые нейронной сетью, еще сложнее, поскольку вес связи между двумя нейронами сам по себе мало что значит. Для того же чтобы понять, как «рассуждало» дерево решения, достаточно просто взглянуть на него, а при желании можно даже представить весь процесс в виде последовательности предложений if–then (если–то).

В этой главе мы рассмотрим три примера использования деревьев решений. В первом из них мы покажем, как спрогнозировать количество пользователей сайта, которые готовы заплатить за премиальный доступ. Многие онлайновые сайты, взимающие плату за подписку или за каждое использование, предлагают пользователям сначала немного поработать с приложением, а только потом раскошеливаться. Если речь идет о подписке, то сайт обычно предлагает бесплатную пробную версию с ограниченным временем или же с урезанными возможностями.

Сайты, взимающие плату за каждое использование, могут предложить один бесплатный сеанс или еще что-то в этом роде.

В других примерах мы применим деревья решений к моделированию цен на недвижимость и к оценке степени привлекательности.

Прогнозирование количества регистраций

Когда сайт с большим числом посетителей развертывает новое приложение, предлагая бесплатный доступ и подписку, оно может привлечь тысячи новых пользователей. Многие из них просто движимы любопытством и на самом деле в этом приложении не заинтересованы, поэтому вероятность того, что они станут платными клиентами, крайне мала. Из-за этого трудно выделить потенциальных клиентов, на которых стоит акцентировать маркетинговые усилия, поэтому многие сайты прибегают к массовой рассылке писем всем открывшим учетную запись, вместо того чтобы действовать более целенаправленно.

Для решения этой проблемы было бы полезно уметь прогнозировать вероятность того, что некий пользователь станет платным клиентом. Вы уже знаете, что для этой цели можно воспользоваться байесовским классификатором или нейронной сетью. Однако в данном случае очень важна четкость – если вы знаете, какие факторы указывают на то, что пользователь может стать клиентом, то можете использовать эту информацию при выработке рекламной стратегии, для того чтобы сделать некоторые разделы сайты более легкодоступными или для придумывания других способов увеличения количества платных клиентов.

Мы будем рассматривать гипотетическое онлайновое приложение, которое предлагается бесплатно на пробный период. Пользователь регистрируется для пробной работы, на какое-то число дней получает доступ к сайту, а потом должен решить, хочет ли он перейти на базовое или премиальное обслуживание. После того как пользователь зарегистрировался, о нем собирается информация, а в конце пробного периода владельцы сайта узнают, какие пользователи решили стать платными клиентами.

Чтобы не раздражать пользователей и завершить регистрацию как можно скорее, сайт не просит заполнять длинную анкету, а собирает информацию из протоколов сервера, например: с какого сайта пользователь попал сюда, его географическое положение, сколько страниц он просмотрел, прежде чем зарегистрировался, и т. д. Если собрать все эти данные и свести их в таблицу, то получится что-то похожее на табл. 7.1.

Таблица 7.1. Поведение пользователя на сайте и окончательное решение об оплате

Откуда пришел	Местонахождение	Читал FAQ	Сколько просмотрел страниц	Выбранное обслуживание
Slashdot	США	Да	18	Нет
Google	Франция	Да	23	Премиальное
Digg	США	Да	24	Базовое
Kiwitobes	Франция	Да	23	Базовое
Google	Великобритания	Нет	21	Премиальное
(напрямую)	Новая Зеландия	Нет	12	Нет
(напрямую)	Великобритания	Нет	21	Базовое
Google	США	Нет	24	Премиальное
Slashdot	Франция	Да	19	Нет
Digg	США	Нет	18	Нет
Google	Великобритания	Нет	18	Нет
Kiwitobes	Великобритания	Нет	19	Нет
Digg	Новая Зеландия	Да	12	Базовое
Google	Великобритания	Да	18	Базовое
Kiwitobes	Франция	Да	19	Базовое

Организуйте данные в виде списка строк, каждая из которых представляет собой список столбцов. В последнем столбце указывается, оформил данный пользователь платный контракт или нет. Именно это значение мы и хотели бы уметь прогнозировать. Создайте новый файл treepredict.py, с которым вы будете работать в этой главе. Если вы хотите вводить данные вручную, включите в начало файла такие строки:

```
my_data=[['slashdot','USA','yes',18,'None'],
         ['google','France','yes',23,'Premium'],
         ['digg','USA','yes',24,'Basic'],
         ['kiwitobes','France','yes',23,'Basic'],
         ['google','UK','no',21,'Premium'],
         ['(direct)','New Zealand','no',12,'None'],
         ['(direct)','UK','no',21,'Basic'],
         ['google','USA','no',24,'Premium'],
         ['slashdot','France','yes',19,'None'],
         ['digg','USA','no',18,'None'],
         ['google','UK','no',18,'None'],
         ['kiwitobes','UK','no',19,'None'],
         ['digg','New Zealand','yes',12,'Basic'],
         ['slashdot','UK','no',21,'None'],
```

```
['google', 'UK', 'yes', 18, 'Basic'],
['kiwitobes', 'France', 'yes', 19, 'Basic']]
```

Можете вместо этого загрузить набор данных со страницы *http://kiwitobes.com/tree/decision_tree_example.txt*.

Для загрузки нужно включить в файл такую строку:

```
my_data=[line.split('\t') for line in file('decision_tree_example.txt')]
```

Теперь у нас есть информация о том, где пользователь находится, как он попал на сайт и сколько времени провел на сайте перед тем, как зарегистрироваться. Нужно лишь заполнить последний столбец, поместив в него обоснованную гипотезу.

Введение в теорию деревьев решений

Деревья решений – один из простейших методов машинного обучения. Это совершенно прозрачный способ классификации наблюдений, и после обучения они представляются в виде последовательности предложений if–then (если–то), организованных в виде дерева. На рис. 7.1 приведен пример дерева решений для классификации фруктов.

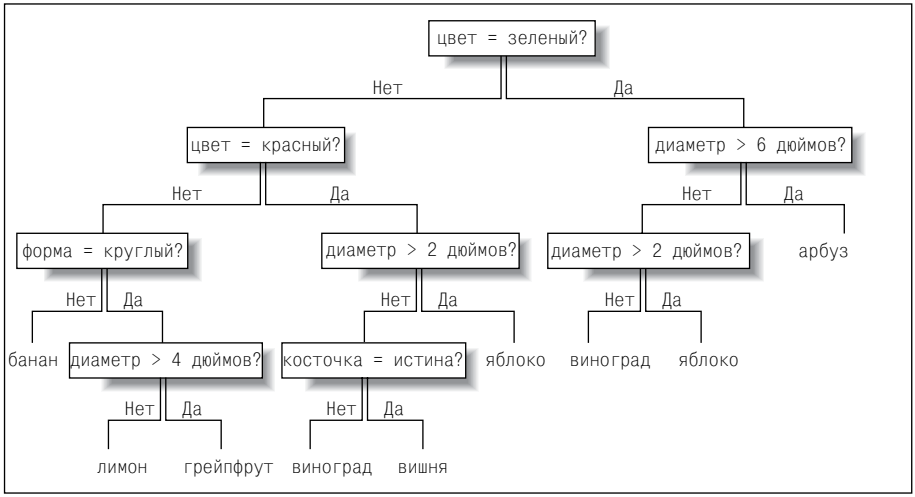


***Рис. 7.1.** Пример дерева решений*

Имея дерево решений, нетрудно понять, как оно принимает решения. Достаточно проследовать вниз по дереву, правильно отвечая на вопросы, – и в конечном итоге вы доберетесь до ответа. Обратная трассировка от узла, в котором вы остановились, до корня дает обоснование выработанной классификации.

В этой главе мы рассмотрим способ представления дерева решений, код построения дерева из реальных данных и код для классификации

новых наблюдений. Первым делом нужно создать представление дерева. Создайте новый класс `decisionnode`, с помощью которого мы будем представлять один узел дерева:

```
class decisionnode:
  def __init__(self,col=-1,value=None,results=None,tb=None,fb=None):
    self.col=col
    self.value=value
    self.results=results
    self.tb=tb
    self.fb=fb
```

В каждом узле имеется пять переменных экземпляра, и все они могут быть заданы в инициализаторе:

- `col` – индекс столбца проверяемого условия.
- `value` – значение, которому должно соответствовать значение в столбце, чтобы результат был равен `true`.
- `tb` и `fb` – экземпляры класса `decisionnodes`, в которые происходит переход в случае, если результаты – `true` или `false` соответственно.
- `results` – словарь результатов для этой ветви. Значение равно `None` для всех узлов, кроме листовых.

Функции, создающие дерево, возвращают корневой узел, от которого можно начать обход, следуя по ветвям `true` или `false`, пока не дойдем до узла, содержащего результаты.

Обучение дерева

В этой главе используется алгоритм *CART* (Classification and Regression Trees – деревья классификации и регрессии). Для построения дерева решений алгоритм сначала создает корневой узел. Рассмотрев все наблюдения в таблице, он выбирает наилучшую переменную, по которой можно разбивать данные на две части. Для этого нужно решить, какое условие (например, «Читал ли пользователь FAQ (Часто задаваемые вопросы)?») разобьет множество выходных данных так, чтобы было проще догадаться, что пользователь собирается сделать (на какое обслуживание он подпишется).

Функция `divideset` разбивает множество строк на два подмножества исходя из данных в одном столбце. На входе она принимает список строк, номер столбца и значение, по которому делить столбец. В случае прочтения FAQ возможные значения – «Да» или «Нет», а для столбца «Откуда пришел» есть несколько возможностей. Функция возвращает два списка строк: первый содержит те строки, для которых данные в указанной колонке соответствуют переданному значению, второй – остальные строки.

```
# Разбиение множества по указанному столбцу. Может обрабатывать как числовые,
# так и дискретные значения.
def divideset(rows,column,value):
```

```
  # Создать функцию, которая сообщит, относится ли строка к первой группе
  # (true) или ко второй (false)
  split_function=None
  if isinstance(value,int) or isinstance(value,float):
    split_function=lambda row:row[column]>=value
  else:
    split_function=lambda row:row[column]==value

  # Разбить множество строк на две части и вернуть их
  set1=[row for row in rows if split_function(row)]
  set2=[row for row in rows if not split_function(row)]
  return (set1,set2)
```

В этом коде создается функция `split_function`, которая разбивает данные на две части. Это делается по-разному в зависимости от того, является ли множество значений непрерывным (float) или дискретным (int). В первом случае возврат `true` означает, что значение в столбце больше `value`. Во втором `split_function` просто проверяет, совпадает ли значение в столбце с `value`. Затем созданная функция применяется для разбиения множества данных на две части: первая состоит из строк, для которые `split_function` вернула `true`, вторая – из тех, для которых она вернула `false`.

Запустите интерпретатор Python и попробуйте разбить множество результатов по столбцу «Читал FAQ»:

```
$ python
>>> import treepredict
>>> treepredict.divideset(treepredict.my_data,2,'yes')
([['slashdot', 'USA', 'yes', 18, 'None'], ['google', 'France', 'yes', 23,
'Premium'],...]]
[['google', 'UK', 'no', 21, 'Premium'], ['(direct)', 'New Zealand', 'no',
12,
'None'],...])
```

В табл. 7.2 показано получившееся разбиение.

Таблица 7.2. Разбиение множества результатов по значениям в столбце «Читал FAQ»

True	False
Нет	Премиальное
Премиальное	Нет
Базовое	Базовое
Базовое	Премиальное
Нет	Нет
Базовое	Нет
Базовое	Нет

Не похоже, что на данном этапе это хорошая переменная для разбиения результатов, поскольку множества в обоих столбцах неоднородны – они содержат все возможные значения. Нужно отыскать переменную получше.

Выбор наилучшего разбиения

Сделанное нами неформальное наблюдение о том, что переменная выбрана не очень хорошо, может быть и верным, но для реализации программы нужен способ измерения неоднородности множества. Требуется найти такую переменную, чтобы множества как можно меньше пересекались. Первое, что нам понадобится, – это функция для вычисления того, сколько раз каждый результат представлен в множестве строк. Добавьте ее в файл treepredict.py:

```
# Вычислить счетчики вхождения каждого результата в множество строк
# (результат - это последний столбец в каждой строке)
def uniquecounts(rows):
  results={}
  for row in rows:
    # Результат находится в последнем столбце
    r=row[len(row)-1]
    if r not in results: results[r]=0
    results[r]+=1
  return results
```

Функция `uniquecounts` возвращает словарь, в котором для каждого результата указано, сколько раз он встретился. Она используется в других функциях для вычисления неоднородности множества. Существует несколько метрик для этой цели, мы рассмотрим две из них: коэффициент Джини и энтропию.

Коэффициент Джини

Предположим, что есть множество образцов, принадлежащих нескольким категориям. Коэффициентом Джини называется вероятность того, что при случайном выборе образца и категории окажется, что образец не принадлежит к указанной категории. Если все образцы в множестве принадлежат к одной и той же категории, то гипотеза всегда будет верна, поэтому вероятность ошибки равна 0. Если в группе в равной пропорции представлены четыре возможных результата, то в 75% случаев гипотеза окажется неверной, поэтому коэффициент ошибки равен 0,75.

Ниже показана функция для вычисления коэффициента Джини:

```
# Вероятность того, что случайный образец принадлежит не к той категории
def giniimpurity(rows):
  total=len(rows)
  counts=uniquecounts(rows)
  imp=0
```

```
  for k1 in counts:
    p1=float(counts[k1])/total
    for k2 in counts:
      if k1==k2: continue
      p2=float(counts[k2])/total
      imp+=p1*p2
  return imp
```

Эта функция вычисляет вероятность каждого из возможных результатов путем деления счетчика появлений этого результата на общее число строк в множестве. Затем произведения этих вероятностей складываются. Это дает полную вероятность того, что для случайно выбранной строки будет спрогнозирован не тот результат, который в действительности имеет место. Чем выше эта вероятность, тем хуже разбиение. Вероятность 0 – это идеал, поскольку в этом случае все строки уже распределены правильно.

Энтропия

В теории информации *энтропией* называют меру беспорядочности множества – по сути говоря, меру его неоднородности. Добавьте в файл treepredict.py следующую функцию:

```
# Энтропия вычисляется как сумма p(x)log(p(x)) по всем различным
# результатам
def entropy(rows):
  from math import log
  log2=lambda x:log(x)/log(2)
  results=uniquecounts(rows)
  # Теперь вычислим энтропию
  ent=0.0
  for r in results.keys( ):
    p=float(results[r])/len(rows)
    ent=ent-p*log2(p)
  return ent
```

Функция `entropy` вычисляет частоту вхождения каждого образца (количество его вхождений, поделенное на общее число образцов – строк) и применяет следующие формулы:

$p(i)$ = частота вхождения = количество вхождений / количество строк

Энтропия = сумма $p(i) \times \log(p(i))$ по всем результатам

Эта величина является мерой того, насколько результаты отличаются друг от друга. Если все они одинаковы (например, вам повезло и все пользователи оформили премиальную подписку), то энтропия равна 0. Чем менее однородны группы, тем выше их энтропия. Наша цель состоит в том, чтобы разбить данные на две новые группы, так чтобы энтропия уменьшилась. Протестируйте метрики, основанные на коэффициенте Джини и на энтропии, в интерактивном сеансе:

```
>>> reload(treepredict)
<module 'treepredict' from 'treepredict.py'>
```

```
>>> treepredict.giniimpurity(treepredict.my_data)
0.6328125
>>> treepredict.entropy(treepredict.my_data)
1.5052408149441479
>>> set1,set2=treepredict.divideset(treepredict.my_data,2,'yes')
>>> treepredict.entropy(set1)
1.2987949406953985
>>> treepredict.giniimpurity(set1)
0.53125
```

Основное отличие энтропии от коэффициента Джини в том, что энтропия выходит на максимум более медленно. Поэтому она штрафует неоднородные множества чуть сильнее. В оставшейся части главы мы будем пользоваться энтропией, поскольку эта метрика чаще употребляется, но при желании ее легко заменить коэффициентом Джини.

Рекурсивное построение дерева

Чтобы оценить, насколько хорош выбранный атрибут, алгоритм сначала вычисляет энтропию всей группы. Затем он пытается разбить группу по возможным значениям каждого атрибута и вычисляет энтропию двух новых групп. Для определения того, какой атрибут дает наилучшее разбиение, вычисляется *информационный выигрыш*, то есть разность между текущей энтропией и средневзвешенной энтропией двух новых групп. Он вычисляется для каждого атрибута, после чего выбирается тот, для которого информационный выигрыш максимален.

Определив условие для корневого узла, алгоритм создает две ветви: по одной надо будет идти, когда условие истинно, по другой – когда ложно (рис. 7.2).

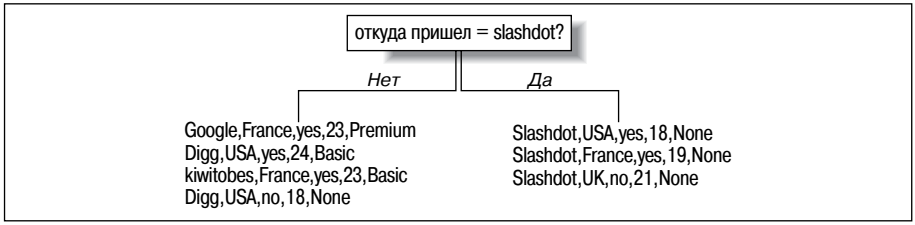


Рис. 7.2. *Дерево решений после одного расщепления*

Наблюдения разбиваются на удовлетворяющие и не удовлетворяющие условию. Для каждой ветви алгоритм определяет, нужно ли разбивать ее далее или мы уже пришли к однозначному заключению. Если какую-то из новых ветвей можно разбить, то с помощью описанного выше метода отыскивается подходящий атрибут. Результат после второго расщепления показан на рис. 7.3.

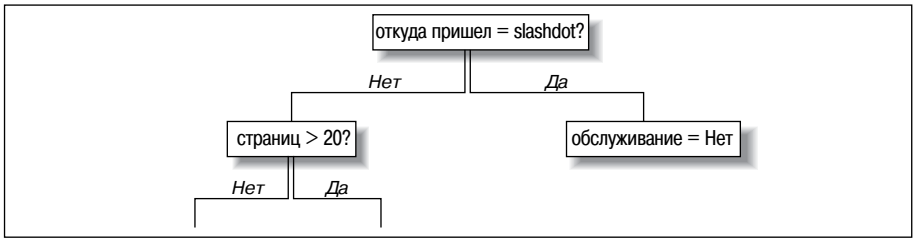


Рис. 7.3. *Дерево решений после двух расщеплений*

Таким образом, вычисляя для каждого узла наилучший атрибут и расщепляя ветви, алгоритм создает дерево. Рост ветви прекращается, когда информационный выигрыш, полученный от расщепления в данном узле, оказывается меньше или равен нулю.

Добавьте в файл treepredict.py функцию `buildtree`. Она рекурсивно строит дерево, выбирая на каждом шаге наилучший критерий расщепления:

```
def buildtree(rows,scoref=entropy):
  if len(rows)==0: return decisionnode( )
  current_score=scoref(rows)

  # Инициализировать переменные для выбора наилучшего критерия
  best_gain=0.0
  best_criteria=None
  best_sets=None

  column_count=len(rows[0])-1
  for col in range(0,column_count):
    # Создать список различных значений в этом столбце
    column_values={}
    for row in rows:
      column_values[row[col]]=1
    # Пробуем разбить множество строк по каждому значению
    # из этого столбца
    for value in column_values.keys( ):
      (set1,set2)=divideset(rows,col,value)

      # Информационный выигрыш
      p=float(len(set1))/len(rows)
      gain=current_score-p*scoref(set1)-(1-p)*scoref(set2)
      if gain>best_gain and len(set1)>0 and len(set2)>0:
        best_gain=gain
        best_criteria=(col,value)
        best_sets=(set1,set2)
  # Создаем подветви
  if best_gain>0:
```

```
    trueBranch=buildtree(best_sets[0])
    falseBranch=buildtree(best_sets[1])
    return decisionnode(col=best_criteria[0],value=best_criteria[1],
                        tb=trueBranch,fb=falseBranch)
  else:
    return decisionnode(results=uniquecounts(rows))
```

При первом вызове этой функции передается весь список строк. Она в цикле перебирает все столбцы (кроме последнего, в котором хранится результат) и по каждому значению, присутствующему в текущем столбце, разбивает множество строк на два подмножества. Для каждой пары подмножеств вычисляется средневзвешенная энтропия, для чего энтропия подмножества умножается на число попавших в него строк. Запоминается, у какой пары эта энтропия оказалась самой низкой.

Если средневзвешенная энтропия наилучшей пары подмножеств не меньше, чем у текущего множества, то рост этой ветви прекращается и сохраняются счетчики возможных результатов. В противном случае для каждого подмножества снова вызывается `buildtree` и результаты вызова присоединяются к ветвям `true` и `false`, исходящим из текущего узла. В итоге будет построено все дерево.

Описанный алгоритм можно применить к исходному набору данных. Приведенный выше код пригоден как для числовых, так и для дискретных данных. Кроме того, предполагается, что в последнем столбце каждой строки находится результат, поэтому для построения дерева достаточно передать только набор строк:

```
>>> reload(treepredict)
<module 'treepredict' from 'treepredict.py'>
>>> tree=treepredict.buildtree(treepredict.my_data)
```

Сейчас в переменной `tree` находится обученное дерево решений. В следующем разделе вы увидите, как построить его визуальное представление, а затем – как с его помощью делать прогнозы.

Отображение дерева

Итак, дерево у нас есть, но что с ним делать? Прежде всего, хотелось бы на него взглянуть. Функция `printtree` распечатывает дерево в текстовом виде. Представление получается не очень красивым, но это простой способ визуализировать небольшие деревья:

```
def printtree(tree,indent=''):
  # Это листовый узел?
  if tree.results!=None:
    print str(tree.results)
  else:
    # Печатаем критерий
    print str(tree.col)+':'+str(tree.value)+'? '

    # Печатаем ветви
    print indent+'T->',
```

```
    printtree(tree.tb,indent+' ')
    print indent+'F->',
    printtree(tree.fb,indent+' ')
```

Это еще одна рекурсивная функция. Она принимает на входе дерево, которое вернула функция `buildtree`, и обходит его сверху вниз. Обход прекращается, когда встретился узел, содержащий результаты (`results`). В противном случае печатается критерий выбора ветви `true` или `false`, а потом для каждой ветви снова вызывается `printtree` с предварительным увеличением ширины отступа.

Если вызвать эту функцию для построенного выше дерева, получится такая картина:

```
>>> reload(treepredict)
>>> treepredict.printtree(tree)
0:google?
T-> 3:21?
  T-> {'Premium': 3}
  F-> 2:yes?
    T-> {'Basic': 1}
    F-> {'None': 1}
F-> 0:slashdot?
  T-> {'None': 3}
  F-> 2:yes?
    T-> {'Basic': 4}
    F-> 3:21?
      T-> {'Basic': 1}
      F-> {'None': 3}
```

Это визуальное представление процедуры, выполняемой деревом решений при попытке классифицировать новый образец. В корневом узле проверяется условие «в столбце 0 находится Google?». Если это условие выполнено, то мы идем по ветви `T->` и обнаруживаем, что каждый пользователь, пришедший с Google, становится платным подписчиком, если просмотрел 21 страницу или более. Если условие не выполнено, мы идем по ветви `F->` и проверяем условие «в столбце 0 находится Slashdot?». Так продолжается до тех пор, пока мы не достигнем узла с результатами. Как уже упоминалось выше, возможность увидеть логику рассуждений – одно из существенных достоинств деревьев решений.

Графическое представление

Текстовое представление хорошо подходит для небольших деревьев, но по мере роста следить за тем, как выполнялся обход дерева, становится все труднее. В этом разделе мы покажем, как построить графическое представление дерева; это будет полезно для просмотра деревьев, которые мы будем создавать далее.

Код рисования дерева похож на код рисования дендрограмм в главе 3. В обоих случаях требуется изобразить двоичное дерево произвольной глубины, поэтому нам понадобятся функции для определения того,

сколько места отвести под каждый узел. Для этого нужно знать полную ширину всех его потомков и глубину узла, чтобы оценить, сколько места по вертикали потребуется для его ветвей. Полная ширина узла равна сумме ширин его дочерних узлов или 1, если дочерних узлов нет:

```
def getwidth(tree):
  if tree.tb==None and tree.fb==None: return 1
  return getwidth(tree.tb)+getwidth(tree.fb)
```

Глубина узла равна 1 плюс глубина самого глубокого из дочерних узлов:

```
def getdepth(tree):
  if tree.tb==None and tree.fb==None: return 0
  return max(getdepth(tree.tb),getdepth(tree.fb))+1
```

Для рисования дерева нам потребуется библиотека Python Imaging Library. Ее можно скачать с сайта *http://pythonware.com*, а в приложении А приведена дополнительная информация по установке. Добавьте в начало файла treepredict.py следующее предложение:

```
from PIL import Image,ImageDraw
```

Функция `drawtree` вычисляет требуемый размер и подготавливает холст. Затем она передает холст и корневой узел дерева функции `drawnode`. Добавьте ее в файл treepredict.py:

```
def drawtree(tree,jpeg='tree.jpg'):
  w=getwidth(tree)*100
  h=getdepth(tree)*100+120

  img=Image.new('RGB',(w,h),(255,255,255))
  draw=ImageDraw.Draw(img)

  drawnode(draw,tree,w/2,20)
  img.save(jpeg,'JPEG')
```

Функция `drawnode` отвечает за изображение узлов дерева решений. Она сначала рисует текущий узел и вычисляет позиции его дочерних узлов, а затем рекурсивно вызывает себя для каждого из дочерних узлов. Добавьте ее в файл treepredict.py:

```
def drawnode(draw,tree,x,y):
  if tree.results==None:
    # Вычислить ширину каждой ветви
    w1=getwidth(tree.fb)*100
    w2=getwidth(tree.tb)*100

    # Вычислить, сколько всего места нужно данному узлу
    left=x-(w1+w2)/2
    right=x+(w1+w2)/2

    # Вывести строку, содержащую условие
    draw.text((x-20,y-10),str(tree.col)+':'+str(tree.value),(0,0,0))

    # Нарисовать линии, ведущие к дочерним узлам
    draw.line((x,y,left+w1/2,y+100),fill=(255,0,0))
    draw.line((x,y,right-w2/2,y+100),fill=(255,0,0))
```

```
    # Нарисовать дочерние узлы
    drawnode(draw,tree.fb,left+w1/2,y+100)
    drawnode(draw,tree.tb,right-w2/2,y+100)
  else:
    txt=' \n'.join(['%s:%d'%v for v in tree.results.items( )])
    draw.text((x-20,y),txt,(0,0,0))
```

Попробуйте нарисовать текущее дерево в интерактивном сеансе:

```
>>> reload(treepredict)
<module 'treepredict' from 'treepredict.pyc'>
>>> treepredict.drawtree(tree,jpeg='treeview.jpg')
```

В результате будет создан файл treeview.jpg, изображенный на рис. 7.4. Метки True и False для ветвей не печатаются, так как они лишь загромождают диаграмму; просто следует иметь в виду, что ветвь True всегда правая. При таком соглашении становится легко следить за процессом рассуждения.

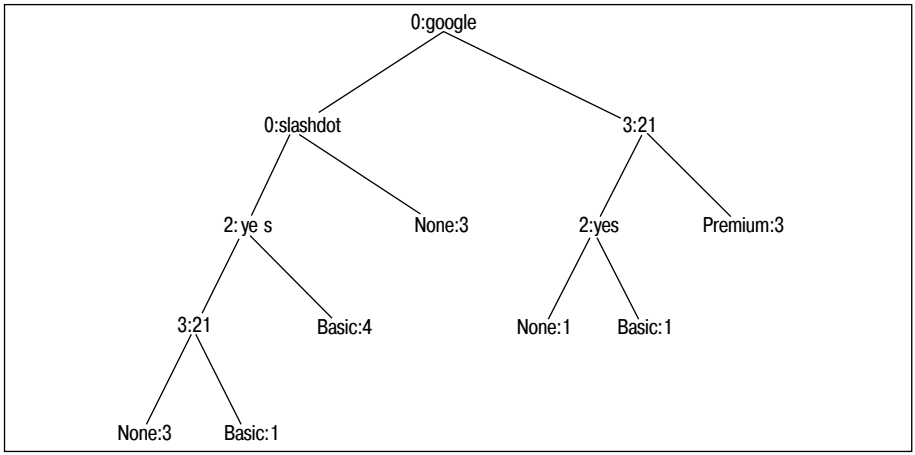


Рис. 7.4. *Дерево решений для прогноза платных подписчиков*

Классификация новых наблюдений

Теперь нам необходима функция, которая классифицирует новое наблюдение в соответствии с деревом решений. Добавьте ее в файл treepredict.py:

```
def classify(observation,tree):
  if tree.results!=None:
    return tree.results
  else:
    v=observation[tree.col]
    branch=None
    if isinstance(v,int) or isinstance(v,float):
      if v>=tree.value: branch=tree.tb
      else: branch=tree.fb
```

```
    else:
      if v==tree.value: branch=tree.tb
      else: branch=tree.fb
    return classify(observation,branch)
```

Эта функция обходит дерево примерно так же, как `printtree`. После каждого вызова проверяется, достигнут ли конец дерева, то есть имеется ли в узле список результатов `results`. Если нет, то для данного наблюдения проверяется условие в текущем узле. Если оно выполнено, то `classify` рекурсивно вызывается для ветви `true`, иначе – для ветви `false`.

Давайте воспользуемся функцией `classify` для получения прогноза относительного нового наблюдения:

```
>>> reload(treepredict)
<module 'treepredict' from 'treepredict.pyc'>
>>> treepredict.classify(['(direct)','USA','yes',5],tree)
{'Basic': 4}
```

Итак, мы имеем функции для создания дерева решений по набору данных, для отображения и интерпретации дерева и для классификации новых наблюдений. Эти функции можно применить к любому набору данных, состоящему из нескольких строк, каждая из которых содержит наблюдения и результат.

Отсечение ветвей дерева

У описанных выше методов обучения дерева есть один недостаток: оно может оказаться *переученным* (overfitted), то есть излишне ориентированным на данные, предъявленные в процессе обучения. Вероятность ответа, возвращенного переученным деревом, может оказаться выше, чем на самом деле, из-за того что были созданы ветви, лишь немного уменьшающие энтропию предъявленного множества наблюдений, хотя выбранное условие расщепления в действительности ничего не характеризует.

Деревья решений в реальном мире

Благодаря простоте интерпретации деревья решений являются одним из самых широко используемых методов добычи данных в анализе бизнеса, принятии решений в медицине и в выработке стратегии. Часто дерево решений создается автоматически, эксперт на его основе выделяет ключевые факторы, а затем уточняет дерево. Эта процедура позволяет эксперту воспользоваться помощью машины, а возможность наблюдать за процессом рассуждений помогает оценить качество прогноза.

Деревья решений применяются в таких приложениях, как профилирование клиентов, анализ финансовых рисков, помощь в диагностике и транспортное прогнозирование.

Описанный выше алгоритм продолжает расщеплять ветви, пока энтропия не перестанет уменьшаться. Остановить ветвление можно, например, задав минимальный порог уменьшения энтропии. Такая стратегия применяется часто, но страдает от одного мелкого недостатка – возможны такие наборы данных, для которых при одном расщеплении энтропия уменьшается чуть-чуть, а при последующих – очень сильно. Альтернативный подход – построить дерево целиком, как описано выше, а затем попытаться удалить лишние узлы. Эта процедура называется *сокращением* (pruning) дерева.

Для сокращения необходимо проверить пары узлов, имеющих общего родителя, и посмотреть, насколько увеличится энтропия при их объединении. Если увеличение окажется меньше заданного порога, то оба листа сливаются в один, для которого множество результатов получается объединением результатов в исходных листьях. Таким образом, мы избегаем феномена переучивания и не даем дереву сделать прогноз с большей долей уверенности, чем позволяют данные.

Добавьте в файл treepredict.py функцию сокращения дерева:

```
def prune(tree,mingain):
  # Если ветви не листовые, вызвать рекурсивно
  if tree.tb.results==None:
    prune(tree.tb,mingain)
  if tree.fb.results==None:
    prune(tree.fb,mingain)

  # Если обе подветви заканчиваются листьями, смотрим, нужно ли их
  # объединить
  if tree.tb.results!=None and tree.fb.results!=None:
    # Строим объединенный набор данных
    tb,fb=[],[]
    for v,c in tree.tb.results.items( ):
      tb+=[[v]]*c
    for v,c in tree.fb.results.items( ):
      fb+=[[v]]*c

  # Вычисляем, насколько уменьшилась энтропия
  delta=entropy(tb+fb)-(entropy(tb)+entropy(fb)/2)
  if delta<mingain:
    # Объединить ветви
    tree.tb,tree.fb=None,None
    tree.results=uniquecounts(tb+fb)
```

При вызове для корневого узла эта функция обходит все дерево, спускаясь до узлов, единственными дочерними узлами которых являются листья. Она объединяет списки результатов, хранящиеся в обоих листьях, и проверяет, насколько изменилась энтропия. Если изменение меньше параметра `mingain`, то оба листовых узла удаляются, а хранившиеся в них результаты перемещаются в узел-родитель. Теперь объединенный узел сам становится кандидатом на удаление и объединение с другим узлом.

Протестируем эту функцию на текущем наборе данных и посмотрим, приведет ли это к объединению каких-нибудь узлов:

```
>>> reload(treepredict)
<module 'treepredict' from 'treepredict.pyc'>
>>> treepredict.prune(tree,0.1)
>>> treepredict.printtree(tree)
0:google?
T-> 3:21?
  T-> {'Premium': 3}
  F-> 2:yes?
    T-> {'Basic': 1}
    F-> {'None': 1}
F-> 0:slashdot?
  T-> {'None': 3}
  F-> 2:yes?
    T-> {'Basic': 4}
    F-> 3:21?
      T-> {'Basic': 1}
      F-> {'None': 3}
>>> treepredict.prune(tree,1.0)
>>> treepredict.printtree(tree)
0:google?
T-> 3:21?
  T-> {'Premium': 3}
  F-> 2:yes?
    T-> {'Basic': 1}
    F-> {'None': 1}
F-> {'None': 6, 'Basic': 5}
```

В этом примере данные разбиваются довольно просто, поэтому сокращение с разумным минимумом ничего не дает. Только сделав минимум очень высоким, удается отсечь ветвь, исходящую из одного узла. Позже вы увидите, что реальные наборы данных разбиваются совсем не так удачно, как этот, поэтому отсечение ветвей оказывается куда более эффективным.

Восполнение отсутствующих данных

Еще одно достоинство деревьев решений заключается в их способности восполнять отсутствующие данные. В имеющемся у вас наборе данных какая-то информация может отсутствовать. Так, в рассматриваемом примере не всегда удается определить географическое местонахождение по IP-адресу, поэтому соответствующее поле может быть пусто. Чтобы приспособить дерево решений к такой ситуации, нужно будет по-другому реализовать функцию прогнозирования.

Если отсутствуют данные, необходимые для принятия решения о том, по какой ветви идти, то следует идти по *обеим* ветвям. Но при этом результаты по обе стороны не считаются равноценными, а взвешиваются. В обычном дереве решений каждому узлу неявно приписывается

вес 1, то есть считается, что при вычислении вероятности попадания образца в некоторую категорию имеющихся наблюдений достаточно. Если же мы идем одновременно по нескольким ветвям, то каждой ветви следует приписать вес, равный доле строк, оказавшихся по соответствующую сторону от узла.

Реализующая эту идею функция `mdclassify` получается небольшой модификацией `classify`. Добавьте ее в файл treepredict.py:

```
def mdclassify(observation,tree):
  if tree.results!=None:
    return tree.results
  else:
    v=observation[tree.col]
    if v==None:
      tr,fr=mdclassify(observation,tree.tb),mdclassify(observation,tree.fb)
      tcount=sum(tr.values( ))
      fcount=sum(fr.values( ))
      tw=float(tcount)/(tcount+fcount)
      fw=float(fcount)/(tcount+fcount)
      result={}
      for k,v in tr.items( ): result[k]=v*tw
      for k,v in fr.items( ): result[k]=v*fw
      return result
    else:
      if isinstance(v,int) or isinstance(v,float):
        if v>=tree.value: branch=tree.tb
        else: branch=tree.fb
      else:
        if v==tree.value: branch=tree.tb
        else: branch=tree.fb
      return mdclassify(observation,branch)
```

Единственное отличие имеется в конце: если существенная информация отсутствует, то мы вычисляем результаты для каждой ветви, а затем складываем их, предварительно умножив на соответствующие веса.

Протестируем функцию `mdclassify` на строке, в которой отсутствует важная информация:

```
>>> reload(treepredict)
<module 'treepredict' from 'treepredict.py'>
>>> treepredict.mdclassify(['google',None,'yes',None],tree)
{'Premium': 1.5, 'Basic': 1.5}
>>> treepredict2.mdclassify(['google','France',None,None],tree)
{'None': 0.125, 'Premium': 2.25, 'Basic': 0.125}
```

Как и следовало ожидать, отсутствие переменной «Сколько просмотрел страниц» дает сильные шансы на премиальное обслуживание и слабые шансы на базовое. Отсутствие переменной «Читал FAQ» дает другое распределение, где шансы на обоих концах взвешены с учетом количества образцов по обе стороны.

Числовые результаты

И моделирование поведения пользователей, и распознавание фруктов – это задачи классификации, поскольку результатом являются категории, а не числа. В следующих примерах, относящихся к ценам на недвижимость и к оценке степени привлекательности, мы рассмотрим задачи с числовыми результатами.

Хотя функцию `buildtree` можно применить к набору данных, где результатами являются числа, ничего хорошего из этого, скорее всего, не получится. Если все числа трактовать как различные категории, то алгоритм не будет принимать во внимание то, насколько сильно числа отличаются; все они будут рассматриваться как абсолютно независимые величины. Чтобы справиться с этой проблемой, необходимо в качестве критерия расщепления использовать дисперсию вместо энтропии или коэффициента Джини. Добавьте в файл treepredict.py функцию `variance`:

```
def variance(rows):
  if len(rows)==0: return 0
  data=[float(row[len(row)-1]) for row in rows]
  mean=sum(data)/len(data)
  variance=sum([(d-mean)**2 for d in data])/len(data)
  return variance
```

Эта функция, передаваемая `buildtree` в качестве параметра, вычисляет статистический разброс набора строк. Низкая дисперсия означает, что числа близки друг к другу, а высокая – что они сильно отличаются. При построении дерева с такой критериальной функцией условие расщепления в узле выбирается так, чтобы большие числа остались по одну сторону от узла, а малые – по другую. В результате должна уменьшиться суммарная дисперсия ветвей.

Моделирование цен на недвижимость

У деревьев решений много потенциальных применений, но наиболее полезны они в тех случаях, когда есть несколько возможных переменных и вас интересует процесс рассуждения. Иногда результаты уже известны, а смысл моделирования заключается в том, чтобы понять, почему они получились именно такими. Одна из областей, где подобная постановка вопроса особенно интересна, – это анализ цен на товары, особенно на такие, для которых измеряемые параметры существенно различаются. В этом разделе мы рассмотрим построение деревьев решений для моделирования цен на недвижимость, поскольку цены на дома различаются очень сильно и при этом существует много числовых и дискретных переменных, легко поддающихся измерению.

API сайта Zillow

Zillow – это бесплатная веб-служба, которая отслеживает цены на недвижимость и с помощью собранной информации дает оценку стоимости других домов. Она анализирует группы похожих домов и на этой основе прогнозирует новое значение. Процедура напоминает ту, которой пользуются оценщики. На рис. 7.5 приведена часть страницы сайта Zillow, на которой показана информация о доме и оценка его стоимости.

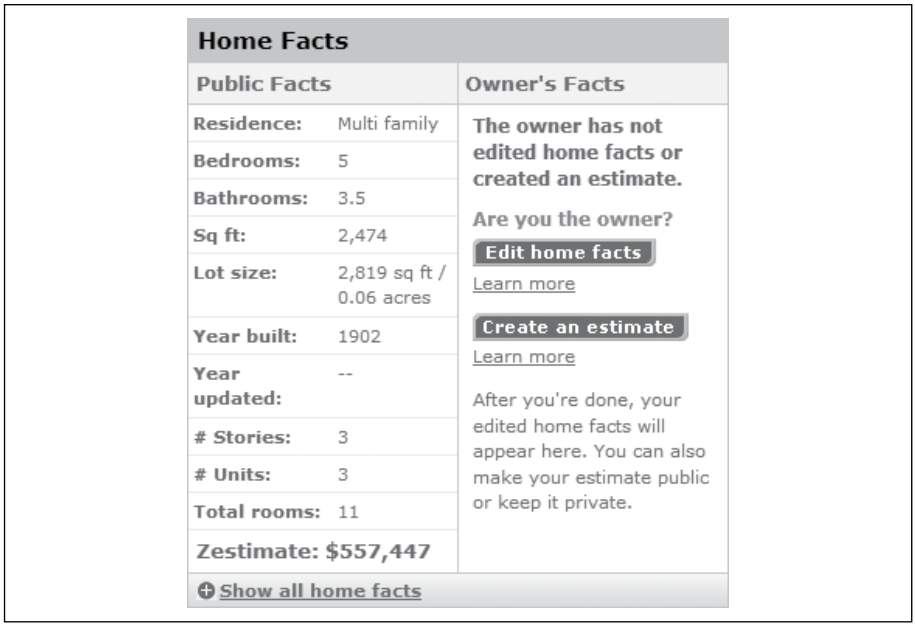


***Рис. 7.5**. Страница сайта zillow.com*

На наше счастье, сайт Zillow предоставляет API, позволяющий получить подробную информацию о доме и его оценочную стоимость. Этот API описан на странице *http://www.zillow.com/howto/api/APIOverview.htm*.

Для доступа к API необходим ключ разработчика, который можно бесплатно получить на сайте. Сам API несложен – достаточно включить в URL все параметры поиска, обратиться к сайту с запросом и разобрать полученный в ответ XML-документ, выделив из него такую информацию, как число спален и оценочная стоимость. Создайте новый файл zillow.py и включите в него такой код:

```
import xml.dom.minidom
import urllib2

zwskey="X1-ZWz1chwxis15aj_9skq6"
```

Как и в главе 5, для разбора XML-документа мы воспользуемся библиотекой minidom. Функция `getaddressdata` получает на входе адрес и город и создает URL с запросом к Zillow. Затем она разбирает полученный ответ, выделяет из него существенную информацию и возвращает ее в виде кортежа. Добавьте эту функцию в файл zillow.py:

```
def getaddressdata(address,city):
  escad=address.replace(' ','+')

  # Создаем URL
  url='http://www.zillow.com/webservice/GetDeepSearchResults.htm?'
  url+='zws-id=%s&address=%s&citystatezip=%s' % (zwskey,escad,city)

  # Разбираем возвращенный XML-документ
  doc=xml.dom.minidom.parseString(urllib2.urlopen(url).read( ))
  code=doc.getElementsByTagName('code')[0].firstChild.data

  # Код 0 означает успех; иначе произошла ошибка
  if code!='0': return None

  # Извлекаем информацию о данной недвижимости
  try:
    zipcode=doc.getElementsByTagName('zipcode')[0].firstChild.data
    use=doc.getElementsByTagName('useCode')[0].firstChild.data
    year=doc.getElementsByTagName('yearBuilt')[0].firstChild.data
    bath=doc.getElementsByTagName('bathrooms')[0].firstChild.data
    bed=doc.getElementsByTagName('bedrooms')[0].firstChild.data
    rooms=doc.getElementsByTagName('totalRooms')[0].firstChild.data
    price=doc.getElementsByTagName('amount')[0].firstChild.data
  except:
    return None

  return (zipcode,use,int(year),float(bath),int(bed),int(rooms),price)
```

Возвращенный этой функцией кортеж можно поместить в список в качестве наблюдения, поскольку «результат» – группа цен – находится в конце. Чтобы воспользоваться этой функцией для генерирования всего набора данных, нам необходим список адресов. Можете составить его сами или скачать список случайно сгенерированных адресов для города Кембридж, штат Массачусетс, со страницы *http://kiwitobes.com/addresslist.txt*.

Добавьте функцию `getpricelist`, которая читает этот файл и генерирует список данных:

```
def getpricelist( ):
  l1=[]
  for line in file('addresslist.txt'):
    data=getaddressdata(line.strip( ),'Cambridge,MA')
    l1.append(data)
  return l1
```

С помощью этих функций можно создать набор данных и построить дерево решений. Сделайте это в интерактивном сеансе:

```
>>> import zillow
>>> housedata=zillow.getpricelist( )
>>> reload(treepredict)
>>> housetree=treepredict.buildtree(housedata,scoref=treepredict.variance)
>>> treepredict.drawtree(housetree,'housetree.jpg')
```

На рис. 7.6 изображен созданный в результат файл housetree.jpg.

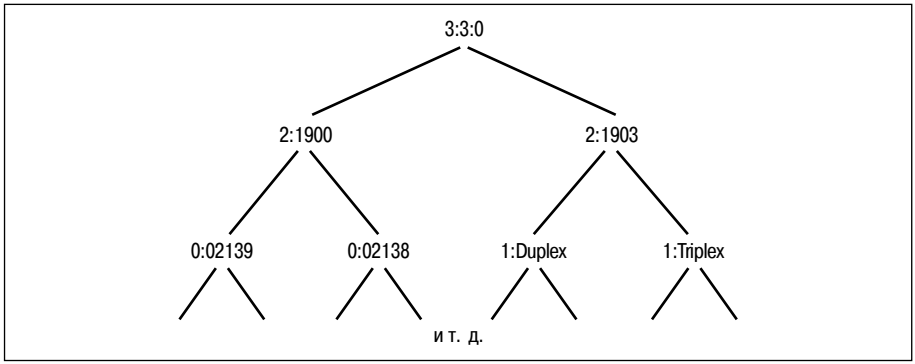


Рис. 7.6. *Дерево решений для цен на дома*

Разумеется, если вам нужна только оценка стоимости конкретного дома, то можно просто воспользоваться API сайта Zillow. Но заметьте следующее: вы только что построили модель факторов, принимаемых в расчет при определении цен на недвижимость. Обратите внимание, что в корне дерева оказался параметр `bathrooms` (число ванных комнат). Это означает, что дисперсия уменьшается больше всего, если разбить набор данных по числу ванных комнат. Решающим фактором при определении цен на дом в Кембридже является то, есть ли в нем три или более ванных комнат (обычно это означает, что речь идет о большом доме на несколько семей).

Очевидный недостаток использования деревьев решений в данном случае – это необходимость создавать группы цен, так как все они различны и должны быть как-то объединены, чтобы создать полезный листовый узел. Возможно, что для реальных данных о ценах больше подошел бы какой-нибудь другой метод прогнозирования. Один такой метод обсуждается в главе 8.

Моделирование степени привлекательности

Hot or Not – это сайт, на который пользователи могут загружать собственные фотографии. Первоначально идея состояла в том, чтобы одни пользователи могли оценивать внешность других. Собранные результаты обрабатывались, и каждому человеку выставлялась оценка от 1 до 10. С тех пор Hot or Not превратился в сайт знакомств и теперь предоставляет API, позволяющий получать демографическую информацию

о пользователях вместе с рейтингом их «привлекательности». Это интересный тестовый пример для модели деревьев решений, поскольку мы имеем набор входных переменных, единственную выходную переменную и потенциально любопытный процесс рассуждения. Да и сам по себе сайт представляет хороший пример коллективного разума.

Как обычно, для доступа к API нужен ключ разработчика. Вы можете зарегистрироваться и получить ключ на странице *http://dev.hotornot.com/signup*.

API сайта Hot or Not работает практически так же, как и другие рассмотренные выше API. В URL передаются параметры, и разбирается возвращенный XML-документ. Для начала создайте файл hotornot.py и включите в него предложения импорта и определение вашего ключа:

```
import urllib2
import xml.dom.minidom

api_key="479NUNJHETN"
```

Далее нужно получить случайный список людей, составляющих набор данных. К счастью, в API сайта Hot or Not предусмотрен вызов, возвращающий список людей, отвечающих заданным критериям. В нашем примере единственным критерием будет наличие профиля meet me (встретиться со мной), так как только из него можно получить такую информацию, как местонахождение и интересы. Добавьте в файл hotornot.py следующую функцию:

```
def getrandomratings(c):
  # Создаем URL для вызова функции getRandomProfile
  url="http://services.hotornot.com/rest/?app_key=%s" % api_key
  url+="&method=Rate.getRandomProfile&retrieve_num=%d" % c
  url+="&get_rate_info=true&meet_users_only=true"

  f1=urllib2.urlopen(url).read( )

  doc=xml.dom.minidom.parseString(f1)

  emids=doc.getElementsByTagName('emid')
  ratings=doc.getElementsByTagName('rating')

  # Объединяем идентификаторы и рейтинги в один список
  result=[]
  for e,r in zip(emids,ratings):
    if r.firstChild!=None:
      result.append((e.firstChild.data,r.firstChild.data))
  return result
```

После того как список идентификаторов и рейтингов пользователей сгенерирован, нам потребуется функция для загрузки информации о людях: пола, возраста, местонахождения и ключевых слов. Если в качестве местонахождения брать все 50 штатов, то получится слишком много ветвлений. Поэтому объединим некоторые штаты в регионы. Добавьте следующий код для задания регионов:

```
stateregions={'New England':['ct','mn','ma','nh','ri','vt'],
              'Mid Atlantic':['de','md','nj','ny','pa'],
              'South':['al','ak','fl','ga','ky','la','ms','mo',
                       'nc','sc','tn','va','wv'],
              'Midwest':['il','in','ia','ks','mi','ne','nd','oh','sd','wi'],
              'West':['ak','ca','co','hi','id','mt','nv','or','ut','wa','wy']
```

API предоставляет метод загрузки демографических данных об одном человеке, поэтому функция `getpeopledata` просто обходит в цикле результаты первого поиска и запрашивает у API детали. Добавьте ее в файл hotornot.py:

```
def getpeopledata(ratings):
  result=[]
  for emid,rating in ratings:
    # URL метода MeetMe.getProfile
    url="http://services.hotornot.com/rest/?app_key=%s" % api_key
    url+="&method=MeetMe.getProfile&emid=%s&get_keywords=true" % emid

    # Получить всю информацию об этом человеке
    try:
      rating=int(float(rating)+0.5)
      doc2=xml.dom.minidom.parseString(urllib2.urlopen(url).read( ))
      gender=doc2.getElementsByTagName('gender')[0].firstChild.data
      age=doc2.getElementsByTagName('age')[0].firstChild.data
      loc=doc2.getElementsByTagName('location')[0].firstChild.data[0:2]

      # Преобразуем штат в регион
      for r,s in stateregions.items( ):
        if loc in s: region=r

      if region!=None:
        result.append((gender,int(age),region,rating))
    except:
      pass
  return result
```

Теперь можно импортировать этот модуль в интерактивном сеансе и сгенерировать набор данных:

```
>>> import hotornot
>>> l1=hotornot.getrandomratings(500)
>>> len(l1)
442
>>> pdata=hotornot.getpeopledata(l1)
>>> pdata[0]
(u'female', 28, 'West', 9)
```

Список содержит информацию о каждом пользователе, причем в последнем поле представлен рейтинг. Эту структуру данных можно передать функции `buildtree` для построения дерева:

```
>>> hottree=treepredict.buildtree(pdata,scoref=treepredict.variance)
>>> treepredict.prune(hottree,0.5)
>>> treepredict.drawtree(hottree,'hottree.jpg')
```

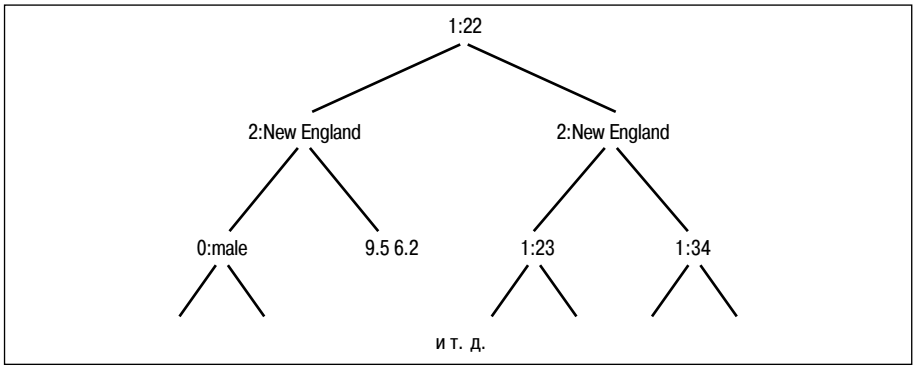


Рис. 7.7. *Моделирование привлекательности с помощью деревьев решений*

Построенное дерево изображено на рис. 7.7.

Корневой узел, по которому проводится начальное разбиение, соответствует полу. Остаток дерева довольно сложен, и читать его трудно. Но его, безусловно, можно использовать для прогнозов о людях, которые дереву не предъявлялись. Кроме того, поскольку алгоритм поддерживает восполнение отсутствующих данных, можно группировать людей по крупным переменным. Так, можно сравнить привлекательность живущих в южных и в среднеатлантических штатах:

```
>>> south=treepredict2.mdclassify((None,None,'South'),hottree)
>>> midat=treepredict2.mdclassify((None,None,'Mid Atlantic'),hottree)
>>> south[10]/sum(south.values( ))
0.055820815183261735
>>> midat[10]/sum(midat.values( ))
0.048972797320600864
```

Как видим, на основе этого набора данных можно сделать вывод о том, что выходцы с юга чуть более привлекательны. Можете попробовать рассмотреть другие факторы, например возраст, или проверить, у кого рейтинг выше – у мужчин или у женщин.

В каких случаях применять деревья решений

Пожалуй, основное достоинство деревьев решений – это простота интерпретации обученной модели. Применив алгоритм к рассмотренной задаче, мы получили не только дерево, способное делать прогнозы о поведении новых пользователей, но и список вопросов, на которые нужно ответить для выработки решения. Например, видно, что люди, приходящие на данный сайт с сайта Slashdot, никогда не становятся платными подписчиками, тогда как нашедшие его с помощью Google и просмотревшие по меньшей мер 20 страниц, вероятно, оформят подписку на премиальное обслуживание. Это, в свою очередь, наводит на мысль изменить рекламную стратегию, сфокусировавшись на сайтах, дающих наиболее высококачественный трафик. Мы выяснили также, что

некоторые переменные, например место проживания, не влияют на результат. Если какие-то данные трудно собрать, а в итоге оказывается, что они несущественны, то можно прекратить их сбор.

В отличие от других алгоритмов машинного обучения, деревья решения могут работать как с числовыми, так и с дискретными данными. В первом примере мы классифицировали страницы по нескольким дискретным показателям. Далее некоторые алгоритмы требуют предварительной подготовки или нормализации данных, а программы из этой главы принимают любой список данных, содержащих числовые или дискретные параметры, и строят соответствующее им дерево решений.

Деревья решений допускают также вероятностные прогнозы. В некоторых задачах для проведения четкого разграничения иногда не хватает данных – в дереве решений может встретиться узел, для которого есть несколько возможностей, а дальнейшее расщепление невозможно. Программа, представленная в этой главе, возвращает словарь счетчиков для различных результатов, и с помощью этой информации мы можем решить, в какой мере результат заслуживает доверия. Не все алгоритмы способны оценить вероятность результата в условиях неопределенности.

Однако у деревьев решений есть и очевидные недостатки. Они хорошо подходят для задач с небольшим числом возможных результатов, но неприменимы к наборам данных, где число возможных исходов велико. В нашем первом примере было всего три результата: «Нет», «Базовое» и «Премиальное». Если бы количество результатов исчислялось сотнями, то построенное дерево оказалось бы слишком сложным и, скорее всего, давало бы плохие прогнозы.

Еще один крупный недостаток рассмотренных выше деревьев решений заключается в том, что хотя они и способны работать с простыми числовыми данными, но условие может формулироваться только в терминах «больше/меньше». Это затрудняет применение деревьев решений к задачам, где класс определяется более сложным сочетанием переменных. Например, если бы результат определялся на основе величины разности между двумя переменными, то дерево выросло бы до невообразимых размеров и очень быстро утратило бы точность прогнозирования.

Подводя итог, можно сказать, что деревья решения – не самый удачный выбор для задач с большим количеством числовых входов и выходов или со сложными взаимосвязями между числовыми входами, какие встречаются, например, при интерпретации финансовых данных или анализе изображений. Напротив, деревья решения – отличный инструмент анализа наборов с большим числом дискретных и числовых данных с четкими точками расщепления. Они оптимальны, когда важно понимать процесс принятия решения; как вы могли убедиться, наблюдение за рассуждением иногда не менее важно, чем конечный прогноз.

Упражнения

1. *Вероятности результатов.* Текущие версии функции `classify` и `mdclassify` возвращают результат в виде набора счетчиков. Модифицируйте их так, чтобы они возвращали вероятности совпадения результатов с той или иной категорией.
2. *Диапазоны отсутствующих данных.* Функция `mdclassify` позволяет указывать в качестве отсутствующего значения строку `None`. Для числовых значений может случиться так, что точное значение неизвестно, но известно, что оно находится в некотором диапазоне. Модифицируйте `mdclassify` так, чтобы она могла принимать кортеж вида (20,25) в качестве значения и обходила обе ветви, если это необходимо.
3. *Ранний останов.* Вместо того чтобы сокращать построенное дерево, функция `buildtree` может просто прекратить расщепление, когда энтропия перестает уменьшаться достаточно ощутимо. В некоторых случаях такое решение не идеально, зато позволяет исключить один шаг. Модифицируйте `buildtree` так, чтобы она принимала в качестве параметра минимальный выигрыш и прекращала расщеплять ветвь, когда энтропия уменьшается на меньшую величину.
4. *Построение дерева в случае отсутствия данных.* Мы написали функцию, которая способна классифицировать строку с отсутствующими данными, но что если данные отсутствуют в обучающем наборе? Модифицируйте `buildtree` так, чтобы она проверяла отсутствие данных и в случае, когда невозможно отправить результат в одну ветвь, отправляла бы его в обе.
5. *Многопутевое расщепление* (трудная задача). Все рассмотренные в этой главе деревья решений были двоичными. Но для некоторых наборов данных структуру дерева можно было бы упростить, если бы было разрешено создавать более двух ветвей из одного узла. Как бы вы представили такое дерево? А как бы вы стали его обучать?

8

Построение ценовых моделей

Мы рассмотрели несколько классификаторов, бóльшая часть которых хорошо приспособлена для прогнозирования того, к какой категории принадлежит новый образец. Однако байесовские классификаторы, деревья решений и машины опорных векторов (с которыми мы ознакомимся в следующей главе) не оптимальны для выработки прогнозов о числовых данных на основе многих различных атрибутов, например цен. В этой главе мы рассмотрим алгоритмы, которые можно не только обучить делать числовые прогнозы, исходя из предъявленных им ранее образцов, но даже показывать распределение вероятностей прогноза, чтобы пользователю было проще интерпретировать цепочку рассуждений.

Применение этих алгоритмов мы рассмотрим на примере построения моделей прогнозирования цен. Экономисты считают, что цены, особенно аукционные, – хороший способ использования коллективного разума для определения реальной стоимости вещи; на большом рынке, где много продавцов и покупателей, цена обычно достигает оптимального для обоих участников сделки значения. Прогнозирование цен – это, ко всему прочему, неплохой тест для такого рода алгоритмов, поскольку на цену обычно влияет множество разнообразных факторов. Например, участвуя в торгах по ноутбуку, вы принимаете в расчет тактовую частоту процессора, объем памяти, емкость дисков, разрешение экрана и т. д.

Важной особенностью числового прогнозирования является определение того, какие переменные существенны и в каких сочетаниях. Так, в случае ноутбука есть несколько переменных, которые если и влияют на цену, то в минимальной степени, например бесплатные аксессуары или предустановленное программное обеспечение. Кроме того, разрешение экрана не так сильно сказывается на цене, как емкость жесткого

диска. Для автоматического определения наилучших весов переменных мы воспользуемся методами оптимизации, разработанными в главе 5.

Построение демонстрационного набора данных

Интересный набор данных для тестирования алгоритмов числового прогнозирования должен обладать несколькими свойствами, которые усложняли бы выработку прогноза. Если посмотреть телевизор, то можно придти к выводу, что чем больше, тем лучше; подобные задачи проще решать традиционными статистическими методами. Поэтому было бы более любопытно изучить такой набор данных, где цена не просто возрастает пропорционально размеру или количеству характеристик.

В этом разделе мы создадим набор данных о ценах на вина, основанный на простой искусственной модели. Цена зависит от сочетания рейтинга и возраста вина. В модели предполагается, что у вина имеется оптимальный возраст, который больше для хороших вин и очень близок к году производства – для плохих. У вина с высоким рейтингом высокая начальная цена, которая возрастает до достижения оптимального возраста, а низкорейтинговое вино изначально стоит дешево и со временем только дешевеет.

Для моделирования этой задачи создайте файл numpredict.py и включите в него функцию `wineprice`:

```
from random import random,randint
import math

def wineprice(rating,age):
  peak_age=rating-50

  # Вычислить цену в зависимости от рейтинга
  price=rating/2
  if age>peak_age:
    # Оптимальный возраст пройден, через 5 лет испортится
    price=price*(5-(age-peak_age))
  else:
    # Увеличивать до пятикратной начальной цены по мере
    # приближения к оптимальному возрасту
    price=price*(5*((age+1)/peak_age))
  if price<0: price=0
  return price
```

Понадобится также функция построения набора данных о ценах на вина. Следующая функция генерирует набор из 200 бутылок вина и вычисляет их цены на основе построенной модели. Затем она случайным образом прибавляет или вычитает 20%, чтобы учесть такие факторы, как налогообложение и местные вариации цен, а также еще немного затруднить прогнозирование числовых значений. Добавьте функцию `wineset1` в файл numpredict.py:

```
def wineset1( ):
  rows=[]
  for i in range(300):
    # Случайным образом выбрать рейтинг и возраст
    rating=random( )*50+50
    age=random( )*50

    # Вычислить эталонную цену
    price=wineprice(rating,age)

    # Добавить шум
    price*=(random( )*0.4+0.8)

    # Включить в набор данных
    rows.append({'input':(rating,age),
                 'result':price})
  return rows
```

Запустите интерпретатор Python, проверьте цены на некоторые вины и сгенерируйте новый набор данных:

```
$ python
>>> import numpredict
>>> numpredict.wineprice(95.0,3.0)
21.111111111111114
>>> numpredict.wineprice(95.0,8.0)
47.5
>>> numpredict.wineprice(99.0,1.0)
10.102040816326529
>>> data=numpredict.wineset1( )
>>> data[0]
{'input': (63.602840187200407, 21.574120872184949), 'result':
34.565257353086487}
>>> data[1]
{'input': (74.994980945756794, 48.052051269308649), 'result': 0.0}
```

В этом наборе вторая бутылка слишком старая и вино в ней уже выдохлось, а в первой как раз созрело. Наличие зависимостей между переменными делает этот набор данных вполне подходящим для тестирования алгоритмов.

Алгоритм k-ближайших соседей

Простейший подход к решению задачи о ценах на вина не отличается от того, которым вы пользуетесь, рассчитывая цены вручную, – найти несколько похожих образцов и предположить, что цены будут примерно одинаковыми. Найдя множество образцов, похожих на тот, что вас интересует, алгоритм может усреднить их цены и предположить, какой будет цена на ваш образец. В этом и состоит суть алгоритма *k-ближайших соседей* (*k*-nearest neighbors – kNN).

Количество соседей

Буква k в аббревиатуре kNN означает количество образцов, по которым нужно проводить усреднение, чтобы получить конечный результат. Если бы данные были идеальными, то достаточно было бы взять $k = 1$, то есть выбрать ближайшего соседа и вернуть его цену в качестве ответа. Но в реальном мире всегда существуют отклонения от идеала. Чтобы их смоделировать, мы специально добавили шум (случайным образом увеличили или уменьшили цену на 20%). Кому-то повезет совершить очень выгодную сделку, а неинформированный покупатель может здорово переплатить, ориентируясь на ближайшего соседа. Поэтому, чтобы подавить шум, лучше взять нескольких соседей и усреднить для них цену.

Чтобы наглядно представить задачу выбора нескольких соседей, вообразите, что существует только одна дескриптивная переменная, например возраст. На рис. 8.1 изображена диаграмма изменения цены (по оси y) в зависимости от возраста (по оси x). Кроме того, на рисунке присутствует кривая, показывающая, что вы получите, если будете ориентироваться только на одного ближайшего соседа.

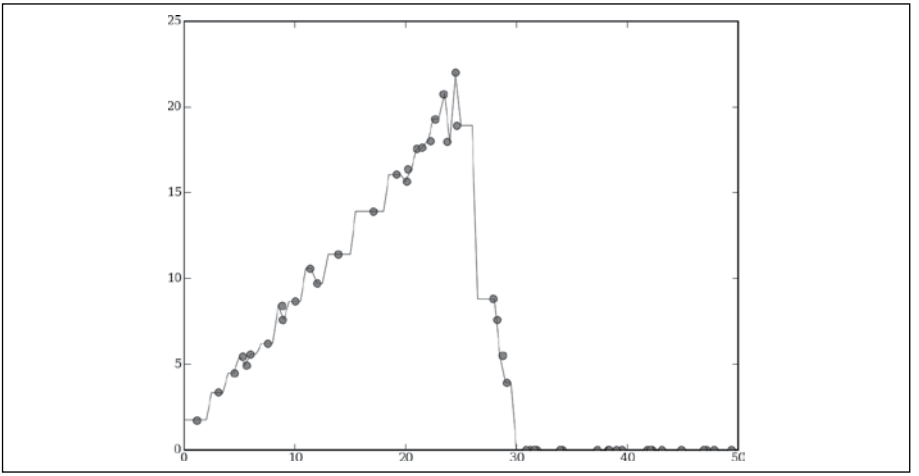


Рис. 8.1. *Алгоритм kNN в случае, когда соседей слишком мало*

Обратите внимание, что прогноз цены слишком сильно зависит от случайных вариаций. Если бы вы пользовались этой далекой от плавности кривой для выработки прогноза, то решили бы, что между 15-летним и 16-летним вином существует большой ценовой скачок, хотя на самом деле это всего лишь результат различия в цене двух конкретных бутылок.

С другой стороны, выбор слишком большого числа соседей снижает точность, поскольку алгоритм будет усреднять данные по образцам,

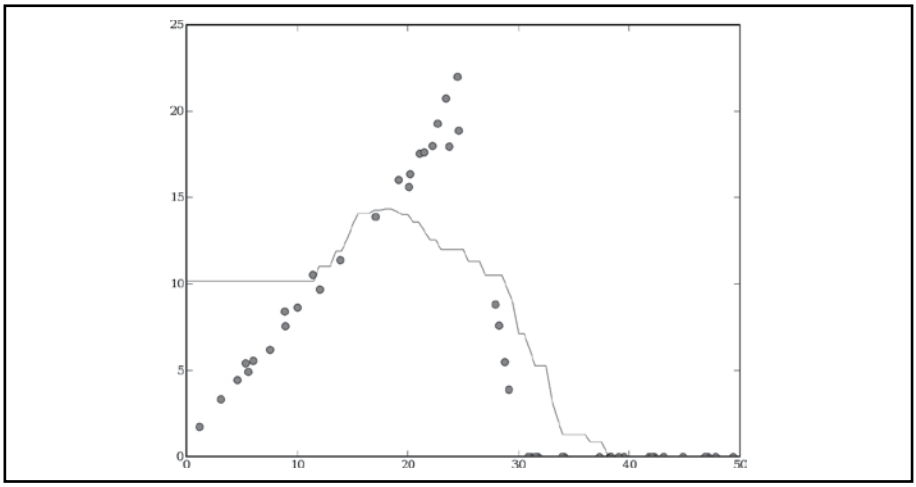


***Рис. 8.2.** Алгоритм kNN в случае, когда соседей слишком много*

которые совсем не похожи на изучаемый. На рис. 8.2 представлен тот же самый набор данных, но кривая цен усреднена по 20 ближайшим соседям.

Понятно, что усреднение слишком многих цен приводит к недооценке вин, достигших 25-летней отметки. Выбор правильного количества соседей можно делать вручную или воспользоваться оптимизацией.

Определение подобия

Первое, что необходимо сделать для применения алгоритма kNN, – найти способ измерения схожести образцов. На страницах этой книги вы уже встречались с различными метриками. В данном случае мы воспользуемся евклидовым расстоянием. Добавьте функцию `euclidian` в файл numpredict.py:

```
def euclidean(v1,v2):
  d=0.0
  for i in range(len(v1)):
    d+=(v1[i]-v2[i])**2
  return math.sqrt(d)
```

В интерактивном сеансе вызовите эту функцию для какой-нибудь пары точек из набора данных:

```
>>> reload(numpredict)
<module 'numpredict' from 'numpredict.py'>
>>> data[0]['input']
(82.720398223643514, 49.21295829683897)
>>> data[1]['input']
(98.942698715228076, 25.702723509372749)
>>> numpredict.euclidean(data[0]['input'],data[1]['input'])
28.56386131112269
```

Как легко заметить, при вычислении расстояния эта функция трактует возраст и рейтинг одинаково, хотя почти в любой реальной задаче одни переменные дают больший вклад в окончательную цену, чем другие. Эта слабость алгоритма kNN хорошо известна, и ниже мы покажем, как ее устранить.

Реализация алгоритма k-ближайших соседей

Алгоритм kNN реализовать довольно просто. Он потребляет много вычислительных ресурсов, но обладает одним достоинством – не требует повторного обучения при каждом добавлении новых данных. Добавьте в файл numpredict.py функцию `getdistances`, которая вычисляет расстояния от заданного образца до всех остальных образцов в исходном наборе данных:

```
def getdistances(data,vec1):
  distancelist=[]
  for i in range(len(data)):
    vec2=data[i]['input']
    distancelist.append((euclidean(vec1,vec2),i))
  distancelist.sort( )
  return distancelist
```

Эта функция в цикле вызывает функцию `distance`, передавая ей заданный вектор и каждый из остальных векторов в наборе данных. Вычисленные расстояния помещаются в большой список. Затем этот список сортируется, так что ближайший образец оказывается в начале.

Алгоритм kNN выполняет усреднение по первым k образцам в получившемся списке. Добавьте в файл numpredict.py функцию `knnestimate`:

```
def knnestimate(data,vec1,k=3):
  # Получить отсортированный список расстояний
  dlist=getdistances(data,vec1)
  avg=0.0

  # Усреднить по первым k образцам
  for i in range(k):
    idx=dlist[i][1]
    avg+=data[idx][‘result’]
  avg=avg/k
  return avg
```

Теперь можно получить оценку цены нового образца:

```
>>> reload(numpredict)
>>> numpredict.knnestimate(data,(95.0,3.0))
29.176138546872018
>>> numpredict.knnestimate(data,(99.0,3.0))
22.356856188108672
>>> numpredict.knnestimate(data,(99.0,5.0))
```

```
37.610888778473793
>>> numpredict.wineprice(99.0,5.0) # Получить фактическую цену
30.306122448979593
>>> numpredict.knnestimate(data,(99.0,5.0),k=1) # Уменьшить число соседей
38.078819347238685
```

Поэкспериментируйте с различными параметрами и значениями k и посмотрите, как это отразится на результатах.

Взвешенные соседи

Один из способов компенсировать тот факт, что алгоритм может отбирать слишком далеких соседей, заключается в том, чтобы взвешивать их с учетом расстояния. Это напоминает метод, который мы применяли в главе 2, когда предпочтения людей взвешивались с учетом того, насколько они схожи с предпочтениями человека, нуждающегося в рекомендации.

Чем более схожи образцы, тем меньше расстояние между ними, поэтому нам необходим способ преобразовать расстояния в веса. Таких способов несколько, и у каждого есть плюсы и минусы. В этом разделе мы рассмотрим три возможных функции.

Инвертирующая функция

В главе 4 для преобразования расстояний в веса мы пользовались инвертирующей функцией. На рис. 8.3 показано, что получится, если отложить по одной оси вес, а по другой – цену.

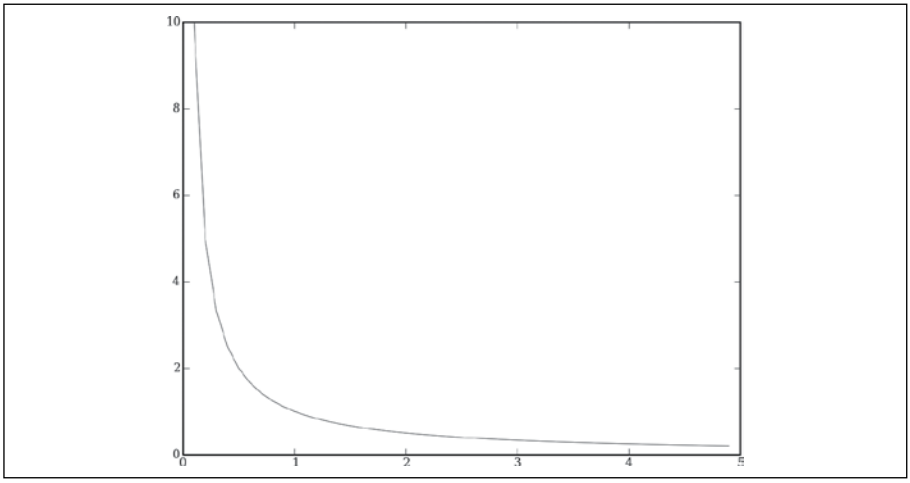


***Рис. 8.3.** Функция инвертирования весов*

В простейшем случае функция возвращает просто 1, поделенную на расстояние. Но иногда образцы оказываются в точности одинаковы или расположены очень близко друг к другу, и тогда значение получается слишком большим или бесконечным. Поэтому перед инвертированием необходимо добавить к расстоянию небольшую величину.

Добавьте в файл numpredict.py функцию `inverseweight`:

```
def inverseweight(dist,num=1.0,const=0.1):
  return num/(dist+const)
```

Эта функция работает очень быстро, и реализовать ее ничего не стоит. Можете поварьировать значение `num`, чтобы понять, когда получаются хорошие результаты. Основной недостаток этой функции в том, что она назначает очень большие веса близко расположенным образцам, а с увеличением расстояния вес быстро убывает. Иногда такое поведение желательно, а порой повышает чувствительность алгоритма к шуму.

Функция вычитания

Второй вариант – это *функция вычитания*, график которой приведен на рис. 8.4.

Она просто вычитает расстояние из некоторой константы. Если получившийся в результате вес оказывается меньше нуля, то он приравнивается к нулю. Добавьте в файл numpredict.py функцию `subtractweight`:

```
def subtractweight(dist,const=1.0):
  if dist>const:
    return 0
  else:
    return const-dist
```

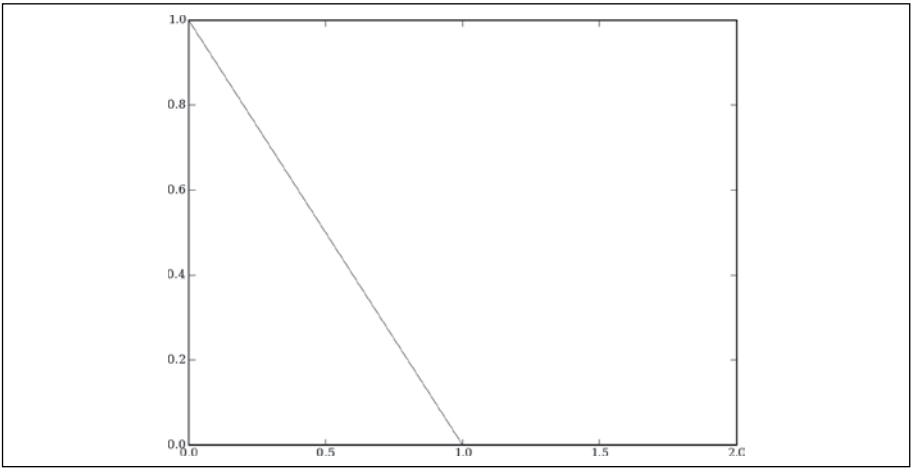


***Рис. 8.4.** Функция вычитания весов*

Эта функция не назначает чрезмерно больших весов близким образцам, но у нее есть свой недостаток. Поскольку рано или поздно вес становится равен 0, может случиться, что не окажется ни одного соседа, которого можно было бы считать близким, а это означает, что алгоритм не даст вообще никакого прогноза.

Гауссова функция

И последней мы рассмотрим *гауссову функцию*, которую иногда называют *колоколообразной кривой*. Она несколько сложнее прочих функций, но, как вы увидите, не страдает от некоторых присущих им ограничений. График гауссовой функций изображен на рис. 8.5.

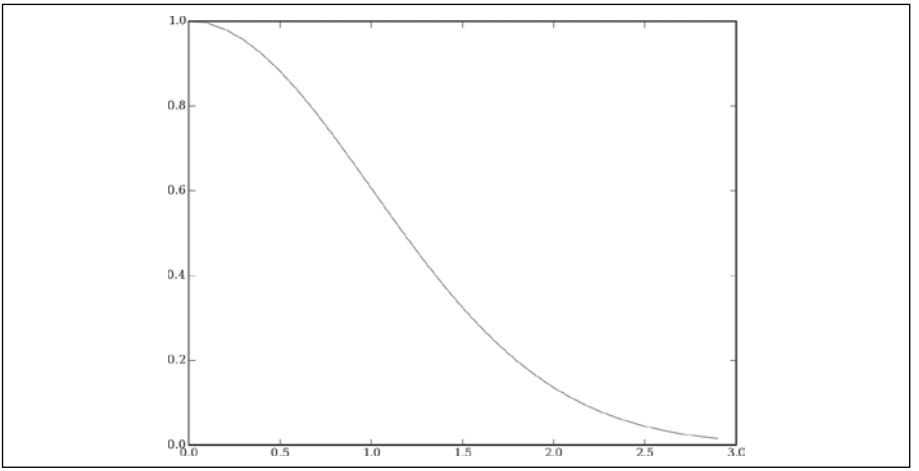


Рис. 8.5. *Гауссова функция*

Эта функция назначает вес 1, когда расстояние равно 0, а по мере увеличения расстояния вес уменьшается. Но, в отличие от функции вычитания, вес не становится равен 0, поэтому прогноз всегда возможен. Реализация гауссовой функции немного сложнее, и она работает не так быстро, как две другие функции.

Добавьте в файл numpredict.py функцию `gaussian`:

```
def gaussian(dist,sigma=10.0):
  return math.e**(-dist**2/(2*sigma**2))
```

Попробуйте применить к одним и тем же образцам разные функции с различными параметрами и посмотрите, как будут изменяться результаты:

```
>>> reload(numpredict)
<module 'numpredict' from 'numpredict.py'>
>>> numpredict.subtractweight(0.1)
```

```
0.9
>>> numpredict.inverseweight(0.1)
5.0
>>> numpredict.gaussian(0.1)
0.99501247919268232
>>> numpredict.gaussian(1.0)
0.60653065971263342
>>> numpredict.subtractweight(1)
0.0
>>> numpredict.inverseweight(1)
0.90909090909090906
>>> numpredict.gaussian(3.0)
0.01110899653824231
```

Вы увидите, что все функции принимают максимальное значение в точке 0,0, а затем убывают, хотя и по-разному.

Взвешенный алгоритм kNN

Код взвешенного алгоритма kNN устроен так же, как и код обычного, то есть сначала мы получаем отсортированный массив расстояний и выбираем из него *k* ближайших элементов. Но дальше мы вычисляем не простое, а *взвешенное среднее*. Для этого каждый элемент умножается на вес, произведения суммируются, и результат делится на сумму всех весов.

Добавьте в файл numpredict.py функцию `weightedknn`:

```
def weightedknn(data,vec1,k=5,weightf=gaussian):
  # Получить расстояния
  dlist=getdistances(data,vec1)
  avg=0.0
  totalweight=0.0

  # Вычислить взвешенное среднее
  for i in range(k):
    dist=dlist[i][0]
    idx=dlist[i][1]
    weight=weightf(dist)
    avg+=weight*data[idx]['result']
    totalweight+=weight
  avg=avg/totalweight
  return avg
```

Эта функция в цикле обходит *k* ближайших соседей и передает расстояние до каждого из них одной из рассмотренных выше функций вычисления весов. Переменная `age` вычисляется путем суммирования произведений веса и значения каждого соседа. В переменной `totalweight` накапливается сумма весов. В самом конце `avg` делится на `totalweight`.

Протестируйте эту функцию в интерактивном сеансе и сравните результаты с обычным алгоритмом kNN:

```
>>> reload(numpredict)
<module 'numpredict' from 'numpredict.py'>
>>> numpredict.weightedknn(data,(99.0,5.0))
32.640981119354301
```

В этом примере результат, возвращенный `weightedknn`, оказался ближе к правильному ответу. Но это всего лишь одно испытание. Для строгого тестирования надо было бы взять много разных образцов из набора данных и по итогам выбрать наилучший алгоритм и оптимальные параметры. Ниже мы посмотрим, как можно провести такое тестирование.

Перекрестный контроль

Перекрестным контролем называется методика, смысл которой заключается в разделении данных на *обучающие* и *тестовые наборы*. Обучающий набор передается алгоритму вместе с правильными ответами (в данном случае ценами) и затем используется для прогнозирования. После этого у алгоритма запрашиваются прогнозы для каждого образца из тестового набора. Полученные ответы сравниваются с правильными, и вычисляется суммарная оценка качества алгоритма.

Обычно эта процедура выполняется несколько раз, причем при каждом прогоне данные разбиваются на два набора по-разному. Как правило, в тестовый набор включается небольшая доля всех данных, скажем 5%, а оставшиеся 95% составляют обучающий набор. Начнем с создания функции `dividedata`, которая разбивает набор данных на две части в соответствии с заданным коэффициентом:

```
def dividedata(data,test=0.05):
  trainset=[]
  testset=[]
  for row in data:
    if random( )<test:
      testset.append(row)
    else:
      trainset.append(row)
  return trainset,testset
```

Следующий шаг – тестирование алгоритма. Для этого мы передаем ему обучающий набор, а затем вызываем его для каждого образца из тестового набора. Функция тестирования вычисляет разности между ответами и, каким-то образом комбинируя их, получает конечную оценку отклонения от идеала. Обычно в качестве оценки берется сумма квадратов разностей.

Добавьте функцию `testalgorithm` в файл numpredict.py:

```
def testalgorithm(algf,trainset,testset):
  error=0.0
  for row in testset:
    guess=algf(trainset,row['input'])
    error+=(row['result']-guess)**2
  return error/len(testset)
```

Этой функции передается алгоритм `algf`, который принимает набор данных и запрос. Она перебирает в цикле все строки из тестового набора и получает от алгоритма прогноз для каждой из них. Полученное значение вычитается из известного результата.

Суммирование квадратов разностей – это общепринятая техника, поскольку чем больше разность, тем сильнее она влияет на сумму, причем ее вклад зависит от величины разности нелинейно. Это означает, что алгоритм, который в большинстве случаев дает очень близкие результаты, но иногда продуцирует большие отклонения, будет считаться хуже алгоритма, который во всех случаях дает умеренно близкие результаты. Зачастую такое поведение желательно, но бывают ситуации, когда можно смириться с крупной ошибкой, если она возникает редко, а в большинстве случаев точность очень высока. Если вы столкнетесь с таким случаем, то можно модифицировать функцию, заменив квадраты разностей на их абсолютные значения.

Последний шаг – это создание функции, которая выполняет несколько разбиений набора данных, прогоняет `testalgorithm` для каждого разбиения и суммирует результаты для получения окончательной оценки. Добавьте ее в файл numpredict.py:

```
def crossvalidate(algf,data,trials=100,test=0.05):
  error=0.0
  for i in range(trials):
    trainset,testset=dividedata(data,test)
    error+=testalgorithm(algf,trainset,testset)
  return error/trials
```

Написанный код можно варьировать различными способами. Например, можно протестировать функцию `knnestimate` с различными значениями k.

```
>>> reload(numpredict)
<module 'numpredict' from 'numpredict.py'>
>>> numpredict.crossvalidate(numpredict.knnestimate,data)
254.06864176819553
>>> def knn3(d,v): return numpredict.knnestimate(d,v,k=3)
...
>>> numpredict.crossvalidate(knn3,data)
166.97339783733005
>>> def knn1(d,v): return numpredict.knnestimate(d,v,k=1)
...
>>> numpredict.crossvalidate(knn1,data)
209.54500183486215
```

Как и следовало ожидать, при слишком малом и слишком большом количестве соседей результаты получаются неудовлетворительными. В данном случае значение 3 оказывается лучше, чем 1 или 5. Можно также попробовать различные весовые функции, которые мы написали для взвешенного алгоритма kNN, и посмотреть, какая даст наилучшие результаты:

```
>>> numpredict.crossvalidate(numpredict.weightedknn,data)
200.34187674254176
>>> def knninverse(d,v):
... return numpredict.weightedknn(d,v,\\
    weightf=numpredict.inverseweight)
>>> numpredict.crossvalidate(knninverse,data)
148.85947702660616
```

При правильно подобранных параметрах взвешенный алгоритм kNN, похоже, дает лучшие результаты для этого набора данных. На подбор параметров может уйти много времени, но делать это необходимо только один раз для конкретного обучающего набора, быть может, повторяя процедуру по мере его роста. В разделе «Оптимизация масштаба» ниже мы рассмотрим некоторые способы автоматического подбора параметров.

Гетерогенные переменные

Набор данных, созданный в начале этой главы, намеренно был сделан простым. Точнее, все переменные, используемые для прогнозирования цены, более-менее сравнимы между собой и все существенны для получения конечного результата.

Так как все переменные попадают в один и тот же диапазон, то имеет смысл вычислять сразу все расстояния между ними. Представьте, однако, что имеется еще одна переменная, влияющая на цену, скажем объем бутылки в миллилитрах. В отличие от остальных переменных, которые изменяются в диапазоне от 0 до 100, эта может принимать значения до 1500. На рис. 8.6 показано, как это отразится на выборе ближайшего соседа и на вычислении весов, соответствующих расстояниям.

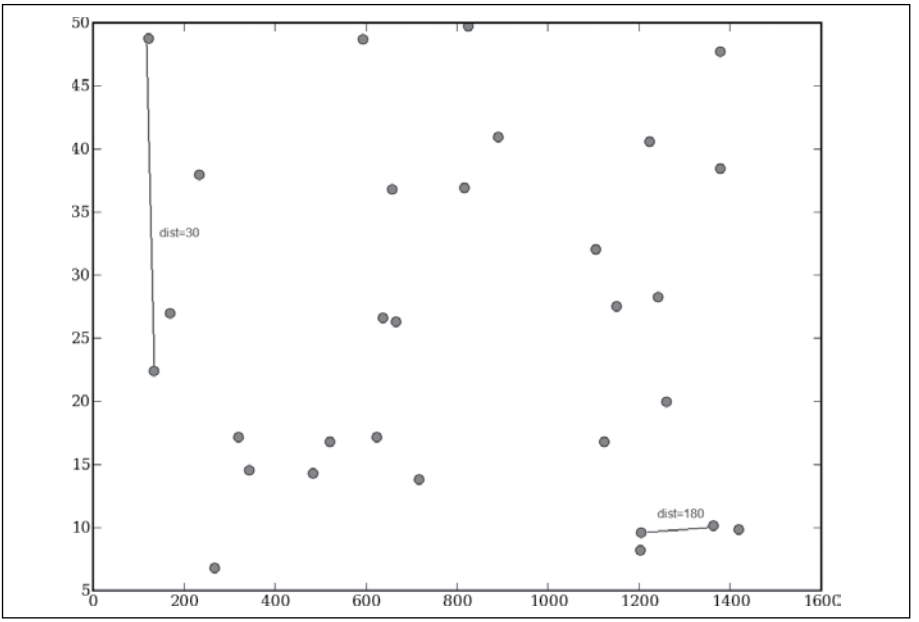


Рис. 8.6. *Наличие гетерогенных переменных приводит к проблемам с расстоянием*

Очевидно, что новая переменная оказывает куда большее влияние на вычисленные расстояния, чем исходные; именно она вносит в расстояние подавляющий вклад, то есть остальные переменные фактически не принимаются в расчет вовсе.

Другая проблема состоит в том, что переменные никак не связаны друг с другом. Если бы мы включили в набор данных еще и номер ряда в магазине, где выставлено вино, то и эта переменная учитывалась бы при вычислении расстоянии. Два образца, идентичные во всех отношениях, кроме ряда, считались бы далеко отстоящими, и это сильно ухудшило бы способность алгоритма делать точные прогнозы.

Расширение набора данных

Чтобы смоделировать эти эффекты, давайте включим в набор данных еще несколько переменных. Можете скопировать код функции `wineset1`, назвать новую функцию `wineset2` и модифицировать ее, добавив строки, выделенные полужирным шрифтом:

```
def wineset2( ):
  rows=[]
  for i in range(300):
    rating=random( )*50+50
    age=random( )*50
    aisle=float(randint(1,20))
    bottlesize=[375.0,750.0,1500.0,3000.0][randint(0,3)]
    price=wineprice(rating,age)
    price*=(bottlesize/750)
    price*=(random( )*0.9+0.2)
    rows.append({'input':(rating,age,aisle,bottlesize),
                 'result':price})
  return rows
```

Теперь можно создать новые наборы данных, содержащие номер ряда и объем бутылки:

```
>>> reload(numpredict)
<module 'numpredict' from 'numpredict.py'>
>>> data=numpredict.wineset2( )
```

Чтобы понять, как это повлияло на прогностические способности различных вариантов алгоритма kNN, прогоните их на новых наборах, задав оптимальные параметры, которые удалось подобрать в предыдущих тестах:

```
>>> numpredict.crossvalidate(knn3,data)
1427.3377833596137
>>> numpredict.crossvalidate(numpredict.weightedknn,data)
1195.0421231227463
```

Отметим, что, хотя новый набор данных содержит даже больше информации и меньше шума, чем предыдущий (теоретически это должно повышать точность прогнозирования), значения, которые вернула функция

`crossvalidate`, ухудшились. Это объясняется следующим: алгоритмы пока не знают о том, что разные переменные следует обрабатывать поразному.

Масштабирование измерений

Нам необходимо не измерять расстояния по фактическим значениям, а найти способ нормализовать переменные, приведя их к единому масштабу. Было бы также хорошо избавиться от лишних переменных или, по крайней мере, уменьшить их влияние на результаты вычислений. Один из способов решить обе задачи состоит в том, чтобы предварительно изменить масштаб по осям, соответствующим различным переменным.

Проще всего это сделать, умножив все значения одной переменной на некую константу. Пример такого масштабирования приведен на рис. 8.7.

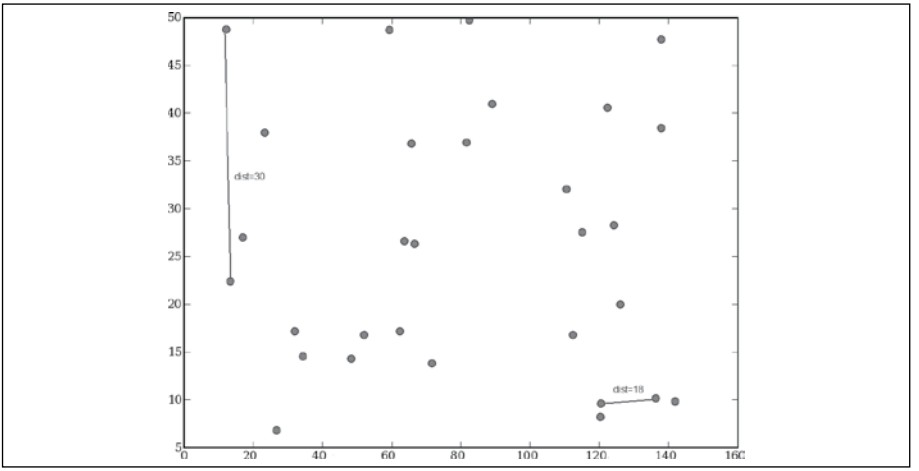


Рис. 8.7. *Масштабирование осей решает проблему расстояний*

Как видите, масштаб по оси, соответствующей объему бутылки, уменьшен в 10 раз, следовательно, ближайшие соседи для некоторых образцов изменились. Это решает проблему, возникающую, когда диапазон значений одних переменных намного шире, чем других. Но как быть с несущественными переменными? Посмотрим, что произойдет, если все значения по одной из осей умножить на 0 (рис. 8.8).

Теперь все точки имеют одну и ту же координату по оси номера ряда, поэтому расстояния между ними зависят только от положения по оси возраста. Таким образом, номер ряда перестал принимать участие в вычислении ближайших соседей и полностью исключен из рассмотрения. Если для всех несущественных переменных установить значение 0, то алгоритмы станут работать гораздо точнее.

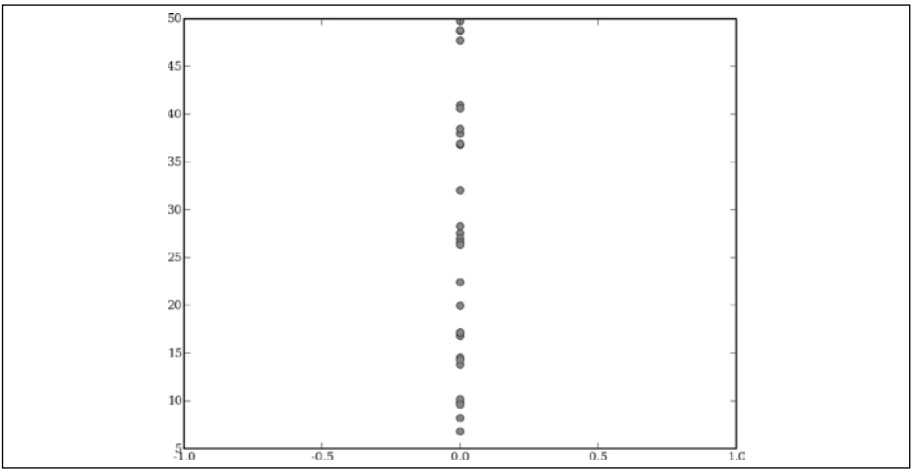


***Рис. 8.8.** Несущественные оси свернуты*

Функция `rescale` принимает список образцов и параметр `scale`, представляющий собой список вещественных чисел. Она возвращает новый набор данных, в котором все значения умножены на числа из набора `scale`. Добавьте эту функцию в файл numpredict.py:

```
def rescale(data,scale):
  scaleddata=[]
  for row in data:
    scaled=[scale[i]*row['input'][i] for i in range(len(scale))]
    scaleddata.append({'input':scaled,'result':row['result']})
  return scaleddata
```

Протестируем эту функцию, применив к набору данных масштабирование с разумно выбранными коэффициентами, и посмотрим, позволит ли это улучшить прогноз:

```
>>> reload(numpredict)
<module 'numpredict' from 'numpredict.py'>
>>> sdata=numpredict.rescale(data,[10,10,0,0.5])
>>> numpredict.crossvalidate(knn3,sdata)
660.9964024835578
>>> numpredict.crossvalidate(numpredict.weightedknn,sdata)
852.32254222973802
```

В этих примерах результаты оказались неплохими; заведомо лучше, чем раньше. Попробуйте поиграть с масштабными коэффициентами и посмотрите, не удастся ли еще улучшить результаты.

Оптимизация масштаба

В данном случае выбрать подходящие параметры масштабирования не так уж сложно, потому что вы заранее знаете, какие переменные

существенны. Но обычно приходится работать с наборами данных, построенными кем-то другим, поэтому априорной информации о том, какие переменные существенны, а какие – нет, может и не быть.

Теоретически можно попробовать много разных сочетаний коэффициентов, пока не найдется дающее приемлемые результаты, но количество переменных может исчисляться сотнями, поэтому такой подход потребует слишком много времени. Но вы ведь уже прочитали главу 5 и знаете, что автоматизировать поиск хорошего решения, когда приходится учитывать много переменных, можно с помощью *оптимизации*.

Напомним, что для оптимизации нужно задать область определения переменных, диапазон и целевую функцию. Функция `crossvalidate` возвращает тем большее значение, чем хуже решение, а следовательно, может служить целевой функцией. Единственное, что осталось сделать, – это написать для нее функцию-обертку, которая принимает список значений, масштабирует данные и вычисляет отклонение при перекрестном контроле. Добавьте функцию `createcostfunction` в файл numpredict.py:

```
def createcostfunction(algf,data):
  def costf(scale):
    sdata=rescale(data,scale)
    return crossvalidate(algf,sdata,trials=10)
  return costf
```

Областью определения является диапазон весов по каждому измерению. В данном случае минимально возможное значение равно 0, поскольку отрицательные числа приводят просто к зеркальному отражению данных, а для вычисления расстояний это неважно. Теоретически веса могут быть произвольно большими, но из практических соображений ограничимся пока значениями не выше 20. Поэтому добавьте в файл numpredict.py такую строку:

```
weightdomain=[(0,20)]*4
```

Теперь у нас есть все необходимое для автоматической оптимизации весов. Убедитесь, что созданный в главе 5 файл optimization.py находится в текущей папке, и попробуйте применить оптимизацию методом имитации отжига:

```
>>> import optimization
>>> reload(numpredict)
<module 'numpredict' from 'numpredict.pyc'>
>>> costf=numpredict.createcostfunction(numpredict.knnestimate,data)
>>> optimization.annealingoptimize(numpredict.weightdomain,costf,step=2)
[11,18,0,6]
```

Блестяще! Алгоритм не только определил, что номер ряда – бесполезная переменная, и умножил масштаб по этой оси на 0, но и выяснил, что диапазон изменения объема бутылки непропорционально велик по сравнению с ее вкладом, и соответственно увеличил масштаб для двух других переменных.

Попробуйте более медленный, но, как правило, более точный алгоритм `geneticoptimize` и посмотрите, получатся ли аналогичные результаты:

```
>>> optimization.geneticoptimize(numpredict.weightdomain,costf,popsize=5,\\
    lrate=1,maxv=4,iters=20)
[20,18,0,12]
```

Преимущество подобной оптимизации масштабов по осям, соответствующим переменным, заключается в том, что вы сразу же видите, какие переменные существенны и насколько они существенны. Иногда бывает, что определенные данные собирать трудно или дорого, и если оказывается, что ценность их на самом деле невелика, то можно избежать лишних затрат. В других случаях одно лишь знание того, какие переменные существенны – особенно в плане влияния на цену, – может определить направление маркетинговых действий или подсказать, как следует изменить дизайн продуктов, чтобы их можно было продать по наивысшей цене.

Неравномерные распределения

До сих пор мы предполагали, что если взять среднее или взвешенное среднее данных, то мы получим приемлемую оценку конечной цены. Часто так оно и есть, но бывают ситуации, когда на результат влияет какая-то неизмеренная переменная. Представьте себе, что покупатели вина делятся на две группы: покупающие в специализированных винных магазинах и в магазинах эконом-класса, где дают скидку 50%. К сожалению, эта информация никак не отражена в наборе данных.

Функция `createhiddendataset` создает набор данных для моделирования описанных свойств. Переменные, введенные только с целью усложнения примера, опущены, мы возвращаемся к исходным. Добавьте эту функцию в файл numpredict.py:

```
def wineset3( ):
  rows=wineset1( )
  for row in rows:
    if random( )<0.5:
      # Вино куплено в магазине эконом-класса
      row['result']*=0.6
  return rows
```

Посмотрим, что произойдет, если запросить оценку цены нового образца с помощью алгоритма kNN или его взвешенной версии. Поскольку набор данных не содержит информации о том, где покупатель приобретает вино, то алгоритм не может принять ее во внимание, поэтому ищет ближайших соседей вне зависимости от места покупки. В результате усреднение производится по обеим группам и, скорее всего, мы получим цену с 25-процентной скидкой. Это можно проверить в интерактивном сеансе:

```
>>> reload(numpredict)
<module 'numpredict' from 'numpredict.py'>
```

```
>>> data=numpredict.wineset3( )
>>> numpredict.wineprice(99.0,20.0)
106.07142857142857
>>> numpredict.weightedknn(data,[99.0,20.0])
83.475441632209339
>>> numpredict.crossvalidate(numpredict.weightedknn,data)
599.51654107008562
```

Если вам нужно получить единственное число, то это, быть может, и неплохой способ оценивания, но он не отражает ту цену, которую заплатит каждый конкретный покупатель. Чтобы избежать некорректного усреднения, необходимо внимательнее взглянуть на данные.

Оценка плотности распределения вероятности

Вместо того чтобы вычислять средневзвешенное соседей и получать оценку в виде одного числа, в данном случае интересно знать вероятность того, что образец попадает в некоторую ценовую категорию. Для 20-летнего вина с рейтингом 99% искомая функция должна была бы сообщить, что с вероятностью 50% цена окажется в диапазоне от $40 до $80 и с вероятностью 50% – в диапазоне от $80 до $100.

Для этого нужна функция, которая возвращает значение от 0 до 1, представляющее вероятность. Сначала она вычисляет веса соседей в заданном диапазоне, а затем – веса всех соседей. Вероятность равна сумме весов соседей в заданном диапазоне, поделенной на сумму всех весов. Назовите новую функцию `probguess` и добавьте ее в файл numpredict.py:

```
def probguess(data,vec1,low,high,k=5,weightf=gaussian):
  dlist=getdistances(data,vec1)
  nweight=0.0
  tweight=0.0

  for i in range(k):
    dist=dlist[i][0]
    idx=dlist[i][1]
    weight=weightf(dist)
    v=data[idx]['result']

    # Данная точка попадает в диапазон?
    if v>=low and v<=high:
      nweight+=weight
      tweight+=weight
  if tweight==0: return 0

  # Вероятность равна сумме весов в заданном диапазоне,
  # поделенной на сумму всех весов
  return nweight/tweight
```

Как и в алгоритме kNN, эта функция сортирует данные по убыванию расстояния от `vec1` и вычисляет веса ближайших соседей. Затем она вычисляет сумму весов всех соседей – `tweight`. Кроме того, она проверяется,

попадает ли цена соседа в заданный диапазон (от `low` до `high`); если да, то вес прибавляется к переменной `nweight`. Вероятность того, что цена `vec1` окажется между `low` и `high`, равна частному от деления `nweight` на `tweight`.

Протестируем эту функцию на нашем наборе данных:

```
>>> reload(numpredict)
<module 'numpredict' from 'numpredict.py'>
>>> numpredict.probguess(data,[99,20],40,80)
0.62305988451497296
>>> numpredict.probguess(data,[99,20],80,120)
0.37694011548502687
>>> numpredict.probguess(data,[99,20],120,1000)
0.0
>>> numpredict.probguess(data,[99,20],30,120)
1.0
```

Функция дает хорошие результаты. Для диапазонов, далеко выходящих за пределы реальных цен, вероятность равна 0, а для диапазонов, полностью перекрывающих возможные цены, она близка к 1. Разбивая диапазон цен на небольшие участки, можно определить, какие из них наиболее вероятны. Однако, чтобы получить ясное представление о структуре данных, вы должны подобрать и ввести набор диапазонов. В следующем разделе мы увидим, как получить полную картину распределения вероятностей.

Графическое представление вероятностей

Чтобы не гадать, какие диапазоны апробировать, можно создать графическое представление плотности распределения вероятности. Для построения графиков существует отличная библиотека *matplotlib*, которую можно скачать с сайта *http://matplotlib.sourceforge.net*.

Инструкции по установке имеются на сайте, а дополнительную информацию о библиотеке matplotlib вы найдете в приложении А. У этой библиотеки масса возможностей, из которых нам в этой главе понадобится совсем немного. После установки попробуйте создать простой график в интерактивном сеансе:

```
>>> from pylab import *
>>> a=array([1,2,3,4])
>>> b=array([4,2,3,1])
>>> plot(a,b)
>>> show( )
>>> t1=arange(0.0,10.0,0.1)
>>> plot(t1,sin(t1))
[<matplotlib.lines.Line2D instance at 0x00ED9300>]
>>> show( )
```

В результате должен получиться график, изображенный на рис. 8.9. Функция `arange` создает список чисел в виде массива примерно так же, как это делает функция `range`. В данном случае мы рисуем синусоиду на отрезке от 0 до 10.

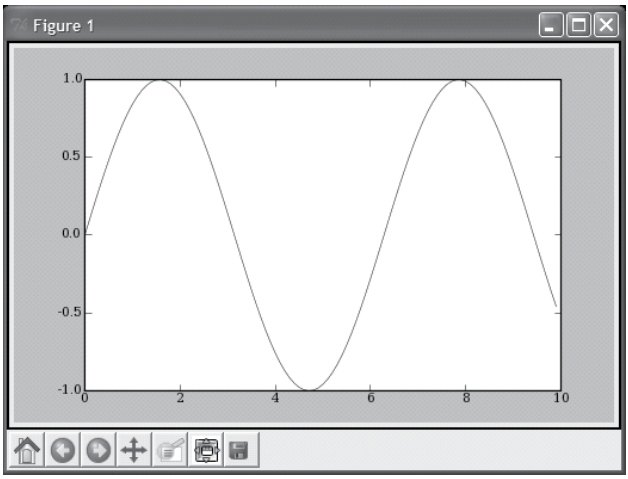


Рис. 8.9. *Простое применение библиотеки matplotlib*

В этом разделе будет описано два разных взгляда на распределение вероятностей. Первый называется *кумулятивным распределением*. На графике кумулятивного распределения показана вероятность того, что результат меньше данной величины. В случае цен кривая начинается в начале координат, поскольку вероятность того, что цена меньше нуля, равна 0, и возрастает, пока не встретится группа образцов с одинаковой ценой. В точке, соответствующей максимальной цене, кривая достигает значения 1, поскольку истинная цена со 100-процентной вероятностью не превышает максимальной.

Чтобы создать набор данных для построения графика кумулятивной вероятности, нужно лишь пробежаться по диапазону цен, вызывая функцию `probguess` со значением 0 в качестве нижней границы и заданной ценой – в качестве верхней. Результаты следует передать функции `plot`, которая нарисует график. Добавьте функцию `cumulativegraph` в файл numpredict.py:

```
def cumulativegraph(data,vec1,high,k=5,weightf=gaussian):
  t1=arange(0.0,high,0.1)
  cprob=array([probguess(data,vec1,0,v,k,weightf) for v in t1])
  plot(t1,cprob)
  show( )
```

Теперь вызовите эту функцию в интерактивном сеансе для построения графика:

```
>>> reload(numpredict)
<module 'numpredict' from 'numpredict.py'>
>>> numpredict.cumulativegraph(data,(1,1),6)
```

График будет выглядеть примерно так, как показано на рис. 8.10. Как и следовало ожидать, в начальной точке кумулятивная вероятность

равна 0, а в конечной – 1. Но интересно, как именно возрастает график. Вероятность остается равной 0 примерно до цены $50, потом, очень быстро возрастая, выходит на плато на уровне 0,6 при цене $110, где снова совершает скачок. Таким образом, глядя на график, мы понимаем, что вероятности группируются в районе цен $60 и $110, поскольку именно в этих точках наблюдаются скачки кумулятивной вероятности. Заранее располагая этой информацией, вы можете рассчитывать вероятности без гадания на кофейной гуще.

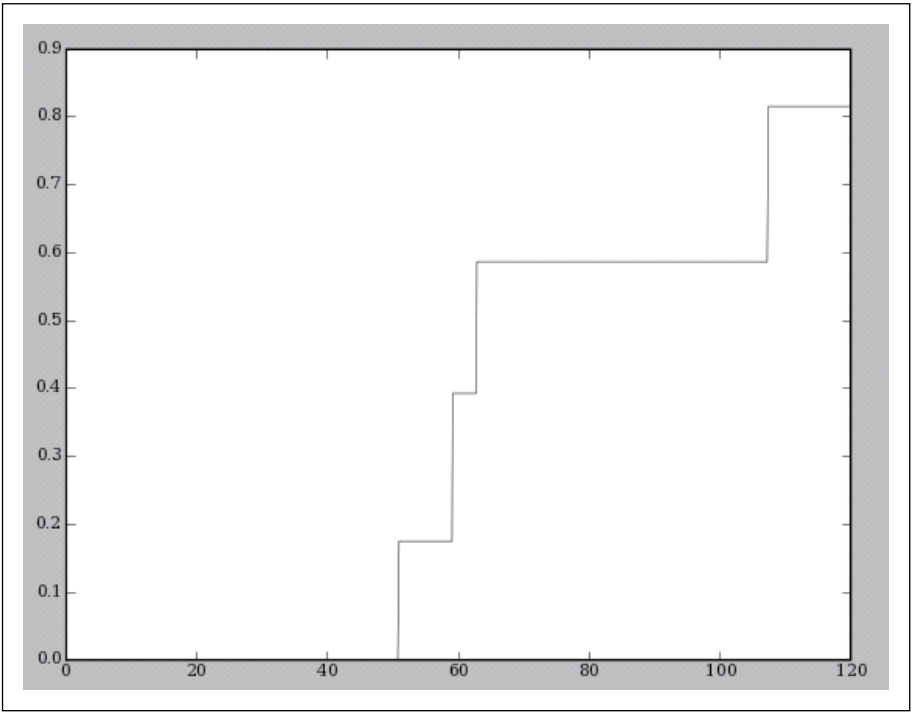


Рис. 8.10. *График кумулятивной вероятности*

Другой вариант – нанести на график фактические вероятности в различных точках по оси цен. Это сложнее, так как вероятность, что цена случайно выбранного образца точно совпадет с заданной, очень мала. Кривая на графике будет равна 0 почти всюду с резкими пиками в спрогнозированных ценах. Вместо этого необходим способ комбинирования вероятностей по некоторым диапазонам.

Решить эту задачу можно, например, предположив, что вероятность в каждой точке есть взвешенное среднее вероятностей в окружающих точках, как во взвешенном алгоритме kNN.

Чтобы продемонстрировать эту идею на практике, добавьте функцию `probabilitygraph` в файл numpredict.py:

```
def probabilitygraph(data,vec1,high,k=5,weightf=gaussian,ss=5.0):
  # Подготовить диапазон цен
  t1=arange(0.0,high,0.1)

  # Вычислить вероятности для всего диапазона
  probs=[probguess(data,vec1,v,v+0.1,k,weightf) for v in t1]

  # Сгладить их, применив гауссову функцию к соседним вероятностям
  smoothed=[]
  for i in range(len(probs)):
    sv=0.0
    for j in range(0,len(probs)):
      dist=abs(i-j)*0.1
      weight=gaussian(dist,sigma=ss)
      sv+=weight*probs[j]
    smoothed.append(sv)
  smoothed=array(smoothed)

  plot(t1,smoothed)
  show( )
```

Эта функция создает диапазон от 0 до `high`, а затем вычисляет вероятности в каждой точке. Поскольку результирующая кривая обычно оказывается зубчатой, то функция пробегает по всему массиву и создает

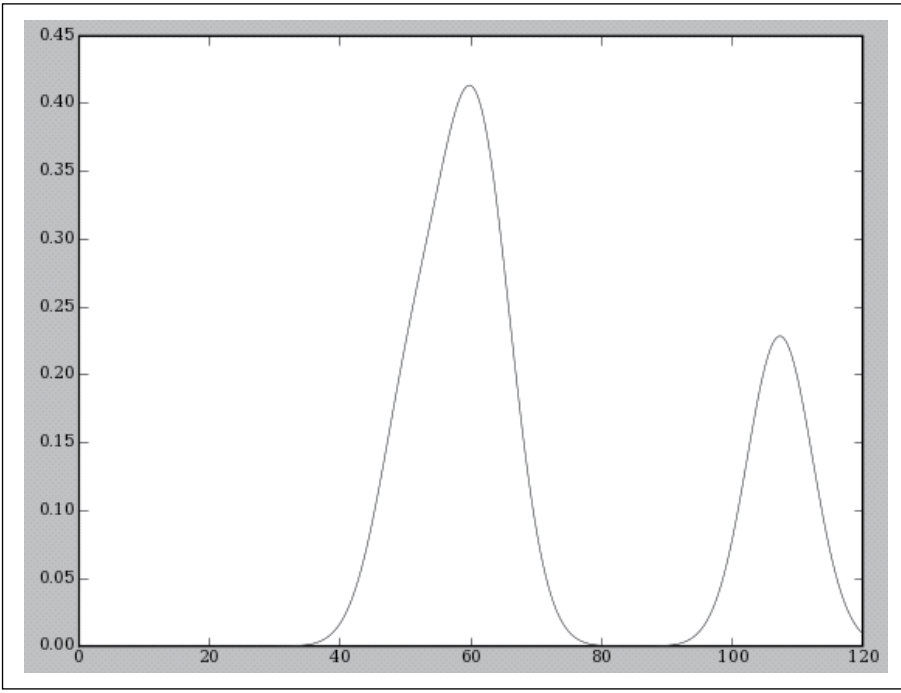


Рис. 8.11. *График плотности распределения вероятности*

сглаженный массив, складывая близкие вероятности. Каждая точка на сглаженной кривой – это взвешенная с помощью гауссовой функции сумма соседей. Параметр ss определяет степень сглаживания.

Вызовите эту функцию в интерактивном сеансе:

```
>>> reload(numpredict)
<module 'numpredict' from 'numpredict.py'>
>>> numpredict.probabilitygraph(data,(1,1),6)
```

Должен получиться примерно такой график, как на рис. 8.11.

На этом графике еще проще увидеть, в каких точках группируются результаты. Попробуйте различные значения параметра ss и посмотрите, как изменяются результаты. Из этого распределения вероятностей ясно видно, что, прогнозируя стоимость бутылки вина, вы упускаете из виду ключевую информацию о том, что одни люди делают более удачные покупки, чем другие. В некоторых случаях удается определить, что это за информация, а иногда вы просто понимаете, что нужно походить по магазинам и поискать более низкую цену.

Использование реальных данных – API сайта eBay

eBay – это онлайновый аукцион, один из наиболее популярных сайтов в Интернете. Там выставлены миллионы товаров, и миллионы пользователей торгуются и совместно устанавливают цены. Поэтому этот сайт – отличный пример коллективного разума. eBay, ко всему прочему, предоставляет бесплатный API на основе XML, с помощью которого можно производить поиск, получать подробную информацию о товаре и даже выставлять товары на торги. В этом разделе вы увидите, как с помощью API сайта eBay получить информацию о ценах и преобразовать ее в формат, допускающий применение описанных в данной главе алгоритмов прогнозирования.

Получение ключа разработчика

Процедура доступа к API сайта eBay состоит из нескольких шагов, но она относительно проста и автоматизирована. Хорошее описание можно найти в кратком руководстве для начинающих по адресу *http://developer.ebay.com/quickstartguide*.

Из этого руководства вы узнаете, как создать учетную запись разработчика, получить ключи и создать маркер. По завершении процедуры у вас будет четыре строки, которые понадобятся для работы с примером из этого раздела:

- Ключ разработчика.
- Ключ приложения.
- Ключ сертификата.
- Маркер аутентификации (очень длинный).

Создайте новый файл ebaypredict.py и включите в него следующий код, в котором импортированы необходимые модули и прописаны полученные вами ключи:

```
import httplib
from xml.dom.minidom import parse, parseString, Node

devKey = 'ключ разработчика'
appKey = 'ключ приложения'
certKey = 'ключ сертификата'
userToken = 'маркер'
serverUrl = 'api.ebay.com'
```

Официального интерфейса к API сайта eBay на языке Python не существует, но есть стандартный API на языке XML, реализованный в библиотеках *httplib* и *minidom*. В этом разделе нам потребуется только два вызова API сайта eBay: `GetSearchResults` и `GetItem`, но бóльшую часть приведенного кода можно использовать и для других вызовов. Дополнительную информацию о структуре API можно найти в документации по адресу *http://developer.ebay.com/DevZone/XML/docs/WebHelp/index.htm*.

Подготовка соединения

Получив ключи, необходимо подготовить соединение для работы с API сайта eBay. От вас требуется передать несколько HTTP-заголовков, в которых заданы ключи и указано, какую функцию API вы собираетесь вызывать. Для этой цели включите в файл ebaypredict.py функцию `getHeaders` – она получает имя функции API и возвращает словарь заголовков, которые предстоит передать библиотеке httplib:

```
def getHeaders(apicall,siteID="0",compatabilityLevel = "433"):
  headers = {"X-EBAY-API-COMPATIBILITY-LEVEL": compatabilityLevel,
             "X-EBAY-API-DEV-NAME": devKey,
             "X-EBAY-API-APP-NAME": appKey,
             "X-EBAY-API-CERT-NAME": certKey,
             "X-EBAY-API-CALL-NAME": apicall,
             "X-EBAY-API-SITEID": siteID,
             "Content-Type": "text/xml"}
  return headers
```

Помимо заголовков, API сайта eBay требует XML-документ с параметрами запроса, а в ответ возвращает XML-документ, который можно разобрать с помощью функции `parseString` из библиотеки minidom.

Функция отправки запроса `sendrequest` открывает соединение с сервером, посылает XML-документ с параметрами и разбирает результат. Добавьте ее в файл ebaypredict.py:

```
def sendRequest(apicall,xmlparameters):
  connection = httplib.HTTPSConnection(serverUrl)
  connection.request("POST", '/ws/api.dll', xmlparameters,
                     getHeaders(apicall))
  response = connection.getresponse( )
  if response.status != 200:
```

```
    print "Ошибка при отправке запроса:" + response.reason
  else:
    data = response.read( )
    connection.close( )
  return data
```

Эти функции годятся для выполнения любого вызова API сайта eBay. Но для конкретных вызовов необходимо по-разному генерировать XML-запрос и интерпретировать разобранный результат.

Поскольку разбор DOM – утомительное занятие, нам потребуется простой вспомогательный метод `getSingleValue`, который ищет указанный узел и возвращает его содержимое:

```
def getSingleValue(node,tag):
  nl=node.getElementsByTagName(tag)
  if len(nl)>0:
    tagNode=nl[0]
    if tagNode.hasChildNodes( ):
      return tagNode.firstChild.nodeValue
  return '-1'
```

Выполнение поиска

Для выполнения поиска нужно лишь задать параметры в формате XML для вызова функции API `GetSearchResults` и передать их определенной выше функции `sendrequest`. Параметры записываются в таком виде:

```
<GetSearchResultsRequest xmlns="urn:ebay:apis:eBLBaseComponents">
<RequesterCredentials><eBayAuthToken>token</eBayAuthToken>
</RequesterCredentials>
<parameter1>value</parameter1>
<parameter2>value</parameter2>
</GetSearchResultsRequest>
```

Этой функции API можно передать десятки параметров, но в данном примере мы ограничимся только двумя:

`Query`

Строка, содержащая поисковые слова. Параметр используется точно так же, как если бы вы набрали запрос на начальной странице сайта eBay.

`CategoryID`

Числовое значение, определяющее категорию, в которой вы ищете. На сайте eBay определена огромная иерархия категорий, которую можно запросить, вызвав функцию API `GetCategories`. Это можно сделать автономно или в сочетании с параметром `Query`.

Функция `doSearch` принимает два параметра и выполняет поиск. Она возвращает список идентификаторов товаров (они нам понадобятся для вызова функции `GetItem`) вместе с их описаниями и текущими ценами. Добавьте эту функцию в файл ebaypredict.py:

```
def doSearch(query,categoryID=None,page=1):
  xml = "<?xml version='1.0' encoding='utf-8'?>"+\
```

```
    "<GetSearchResultsRequest xmlns=\"urn:ebay:apis:eBLBaseComponents\">"+\
      "<RequesterCredentials><eBayAuthToken>" +\
      userToken +\
      "</eBayAuthToken></RequesterCredentials>" + \
      "<Pagination>"+\
        "<EntriesPerPage>200</EntriesPerPage>"+\
        "<PageNumber>"+str(page)+"</PageNumber>"+\
      "</Pagination>"+\
      "<Query>" + query + "</Query>"
  if categoryID!=None:
     xml+="<CategoryID>"+str(categoryID)+"</CategoryID>"
  xml+="</GetSearchResultsRequest>"

  data=sendRequest('GetSearchResults',xml)
  response = parseString(data)
  itemNodes = response.getElementsByTagName('Item');
  results = []
  for item in itemNodes:
    itemId=getSingleValue(item,'ItemID')
    itemTitle=getSingleValue(item,'Title')
    itemPrice=getSingleValue(item,'CurrentPrice')
    itemEnds=getSingleValue(item,'EndTime')
    results.append((itemId,itemTitle,itemPrice,itemEnds))
  return results
```

Чтобы воспользоваться параметром `category`, понадобится также функция для получения иерархии категорий. Вызов API в данном случае очень прост, но XML-документ со всеми данными категории огромен, загружается долго и разбирать его трудно. Поэтому ограничьтесь какой-нибудь категорией верхнего уровня.

Функция `getCategory` получает строку и идентификатор родительской категории и возвращает все ее подкатегории, в названии которых присутствует заданная строка. Если идентификатор родителя опущен, то возвращается просто список всех категорий верхнего уровня. Добавьте эту функцию в файл ebaypredict.py:

```
def getCategory(query='',parentID=None,siteID='0'):
  lquery=query.lower( )
  xml = "<?xml version='1.0' encoding='utf-8'?>"+\
        "<GetCategoriesRequest xmlns=\"urn:ebay:apis:eBLBaseComponents\">"+\
        "<RequesterCredentials><eBayAuthToken>" +\
        userToken +\
        "</eBayAuthToken></RequesterCredentials>"+\
        "<DetailLevel>ReturnAll</DetailLevel>"+\
        "<ViewAllNodes>true</ViewAllNodes>"+\
        "<CategorySiteID>"+siteID+"</CategorySiteID>"
  if parentID==None:
     xml+="<LevelLimit>1</LevelLimit>"
  else:
     xml+="<CategoryParent>"+str(parentID)+"</CategoryParent>"
  xml += "</GetCategoriesRequest>"
  data=sendRequest('GetCategories',xml)
  categoryList=parseString(data)
```

```
  catNodes=categoryList.getElementsByTagName('Category')
  for node in catNodes:
    catid=getSingleValue(node,'CategoryID')
    name=getSingleValue(node,'CategoryName')
    if name.lower( ).find(lquery)!=-1:
      print catid,name
```

Протестируйте ее в интерактивном сеансе:

```
>>> import ebaypredict
>>> laptops=ebaypredict.doSearch('laptop')
>>> laptops[0:10]
[(u'110075464522', u'Apple iBook G3 12" 500MHZ Laptop , 30 GB HD ', u'299.99',
u'2007-01-11T03:16:14.000Z'),
 (u'150078866214', u'512MB PC2700 DDR Memory 333MHz 200-Pin Laptop SODIMM',
  u'49.99',u'2007-01-11T03:16:27.000Z'),
(u'120067807006', u'LAPTOP USB / PS2 OPTICAL MOUSE 800 DPI SHIP FROM USA',
u '4.99', u'2007-01-11T03:17:00.000Z'),
...
```

Похоже, что поиск по слову laptop возвращает и все аксессуары, так или иначе относящиеся к лэптопам (портативным компьютерам). К счастью, можно поискать в категории «Laptops, Notebooks» (Лэптопы, Ноутбуки), чтобы ограничиться лишь собственно лэптопами. Сначала следует запросить список категорий верхнего уровня, найти в категории «Computers and Networking» (Компьютеры и Построение сети) подкатегорию «Laptops, Notebooks» (Лэптопы, Ноутбуки), получить ее идентификатор и затем искать по слову laptop в нужной категории.

```
>>> ebaypredict.getCategory('computers')
58058 Computers & Networking
>>> ebaypredict.getCategory('laptops',parentID=58058)
25447 Apple Laptops, Notebooks
...
31533 Drives for Laptops
51148 Laptops, Notebooks...
>>> laptops=ebaypredict.doSearch('laptop',categoryID=51148)
>>> laptops[0:10]
[(u'150078867562', u'PANASONIC TOUGHBOOK Back-Lit KeyBoard 4 CF-27 CF-28',
u'49.95', u'2007-01-11T03:19:49.000Z'),
(u'270075898309', u'mini small PANASONIC CFM33 CF M33 THOUGHBOOK ! libretto',
u'171.0', u'2007-01-11T03:19:59.000Z'),
(u'170067141814', u'Sony VAIO "PCG-GT1" Picturebook Tablet Laptop MINT ',
u'760.0', u'2007-01-11T03:20:06.000Z'),...
```

Во время работы над этой книгой у категории «Laptops, Notebooks» (Лэптопы, Ноутбуки) был идентификатор 51148. Как видите, ограничив поиск этой категорией, мы смогли устранить многие не относящиеся к делу результаты, которые возвращает поиск по одному лишь слову laptop. В результате получился набор данных, гораздо более пригодный для построения ценовой модели.

Получение подробной информации о товаре

В состав результатов поиска входят название и цена, и из текста названия можно получить дополнительные детали, например емкость диска и цвет корпуса. Для лэптопов указываются такие атрибуты, как тип процессора и объем памяти, а для плееров iPod – емкость диска. Кроме того, можно узнать еще рейтинг продавца, количество заявок на покупку и начальную цену.

Чтобы получить все эти детали, необходимо обратиться к функции API `GetItem`, передав ей идентификатор товара, возвращенный в результате поиска. Для этого создайте и включите в файл ebaypredict.py функцию `getItem`:

```
def getItem(itemID):
  xml = "<?xml version='1.0' encoding='utf-8'?>"+\
        "<GetItemRequest xmlns=\"urn:ebay:apis:eBLBaseComponents\">"+\
        "<RequesterCredentials><eBayAuthToken>" +\
        userToken +\
        "</eBayAuthToken></RequesterCredentials>" + \
        "<ItemID>" + str(itemID) + "</ItemID>"+\
        "<DetailLevel>ItemReturnAttributes</DetailLevel>"+\
        "</GetItemRequest>"
  data=sendRequest('GetItem',xml)
  result={}
  response=parseString(data)
  result['title']=getSingleValue(response,'Title')
  sellingStatusNode = response.getElementsByTagName('SellingStatus')[0];
  result['price']=getSingleValue(sellingStatusNode,'CurrentPrice')
  result['bids']=getSingleValue(sellingStatusNode,'BidCount')
  seller = response.getElementsByTagName('Seller')
  result['feedback'] = getSingleValue(seller[0],'FeedbackScore')
  attributeSet=response.getElementsByTagName('Attribute');
  attributes={}
  for att in attributeSet:
    attID=att.attributes.getNamedItem('attributeID').nodeValue
    attValue=getSingleValue(att,'ValueLiteral')
    attributes[attID]=attValue
  result['attributes']=attributes
  return result
```

Эта функция получает с помощью `sendrequest` XML-документ, содержащий описание товара, а затем извлекает из него интересующие нас данные. Поскольку набор атрибутов для каждого товара свой, то все они возвращаются в виде словаря. Протестируйте функцию на результатах предыдущего поиска:

```
>>> reload(ebaypredict)
>>> ebaypredict.getItem(laptops[7][0])
{'attributes': {u'13': u'Windows XP', u'12': u'512', u'14': u'Compaq',
                u'3805': u'Exchange', u'3804': u'14 Days',
```

```
                u'41': u'-', u'26445': u'DVD+/-RW', u'25710': u'80.0',
                u'26443': u'AMD Turion 64', u'26444': u'1800', u'26446': u'15',
                u'10244': u'-'},
 'price': u'515.0', 'bids': u'28', 'feedback': u'2797',
 'title': u'COMPAQ V5210US 15.4" AMD Turion 64 80GB Laptop Notebook'}
```

По-видимому, атрибут 26444 представляет тактовую частоту процессора, 26446 – размер экрана, 12 – объем оперативной памяти, а 25710 – емкость жесткого диска. Если добавить сюда еще рейтинг продавца, количество заявок и начальную цену, то мы получим интересный набор данных для прогнозирования цен.

Построение предсказателя цен

Чтобы воспользоваться разработанным в этой главе механизмом прогнозирования цен, нам потребуется получить описания товаров с сайта eBay и преобразовать их в списки чисел, которые можно передать как наборы данных функции перекрестного контроля. Для этого функция `makeLaptopDataset` сначала вызывает `doSearch`, чтобы получить список лэптопов, а затем для каждого выполняет индивидуальный запрос. Извлекая описанные в предыдущем разделе атрибуты, эта функция создает список чисел, пригодный для прогнозирования, и помещает данные в структуру, необходимую для работы алгоритма kNN.

Добавьте функцию `makeLaptopDataset` в файл ebaypredict.py:

```
def makeLaptopDataset( ):
  searchResults=doSearch('laptop',categoryID=51148)
  result=[]
  for r in searchResults:
    item=getItem(r[0])
    att=item['attributes']
    try:
      data=(float(att['12']),float(att['26444']),
            float(att['26446']),float(att['25710']),
            float(item['feedback'])
           )
      entry={'input':data,'result':float(item['price'])}
      result.append(entry)
    except:
      print item['title']+' failed'
  return result
```

Эта функция игнорирует товары, у которых нет необходимых атрибутов. Для получения и обработки результатов потребуется некоторое время, но в итоге у вас окажется интересный набор данных с реальными ценами и атрибутами. Вызовите функцию в интерактивном сеансе:

```
>>> reload(ebaypredict)
<module 'ebaypredict' from 'ebaypredict.py'>
>>> set1=ebaypredict.makeLaptopDataset( )
...
```

Теперь можно попробовать получить с помощью алгоритма kNN оценки для разных конфигураций:

```
>>> numpredict.knnestimate(set1,(512,1000,14,40,1000))
667.89999999999998
>>> numpredict.knnestimate(set1,(1024,1000,14,40,1000))
858.42599999999982
>>> numpredict.knnestimate(set1,(1024,1000,14,60,0))
482.02600000000001
>>> numpredict.knnestimate(set1,(1024,2000,14,60,1000))
1066.8
```

Вот вы и увидели, как влияют на цену объем памяти, быстродействие процессора и количество заявок. Попробуйте поиграть с параметрами, выполните масштабирование данных и постройте графики распределения вероятностей.

В каких случаях применять метод k-ближайших соседей

У метода *k*-ближайших соседей есть несколько недостатков. Прогнозирование потребляет очень много вычислительных ресурсов, так как приходится вычислять расстояния до каждой точки. Кроме того, если в наборе данных много переменных, то трудно подобрать подходящие веса и определить, какие переменные несущественны. Оптимизация может помочь, но отыскание хорошего решения для большого набора данных отнимает много времени.

И все же, как вы могли убедиться, метод kNN обладает рядом преимуществ по сравнению с другими методами. Оборотной стороной вычислительной сложности прогнозирования является тот факт, что новые наблюдения можно добавлять без каких бы то ни было временных затрат. К тому же результаты легко поддаются интерпретации, так как вы знаете, что прогнозирование основано на вычислении средневзвешенных значений других наблюдений.

Хотя определение весов может оказаться сложным делом, но если они уже вычислены, то позволяют лучше понять характеристики набора данных. Наконец, если вы подозреваете, что в наборе не учтены какие-то существенные переменные, то можно построить график распределения вероятностей.

Упражнения

1. *Оптимизация количества соседей.* Разработайте целевую функцию для определения оптимального количества соседей в простом наборе данных.

2. *Перекрестный контроль с исключением по одному образцу.* Это альтернативный способ вычисления ошибки прогнозирования, в котором каждая строка набора данных по отдельности рассматривается как тестовый набор, а все остальные строки – как обучающий набор. Реализуйте соответствующую функцию. Сравните результаты, полученные с помощью этой функции и с помощью метода, описанного в настоящей главе.
3. *Исключение переменных.* Вместо того чтобы оптимизировать масштабы для большого числа переменных, которые в итоге могут оказаться бесполезными, можно предварительно попытаться исключить переменные, резко ухудшающие качество прогноза. Сможете ли вы придумать, как это сделать?
4. *Изменение параметра* `ss` *для построения графика распределения вероятностей.* Параметр `ss` в функции `probabilityguess` определяет степень сглаживания кривой распределения. Что будет, если выбрать слишком большое значение? А если слишком маленькое? Можете ли вы определить, каким должно быть хорошее значение, не глядя на график?
5. *Набор данных о лэптопах.* Попытайтесь выполнить оптимизацию для набора данных о лэптопах, полученного с сайта eBay. Какие переменные существенны? Теперь попробуйте построить график плотности распределения вероятности. Есть ли на нем заметные пики?
6. *Другие виды товаров.* У каких еще товаров на сайте eBay есть подходящие числовые атрибуты? В описаниях плееров iPod, сотовых телефонов и автомобилей имеется масса интересной информации. Попробуйте создать еще один набор для числового прогнозирования.
7. *Поиск по атрибутам.* В API сайта eBay есть немало функций, которые мы в этой главе не рассматривали. У функции `GetSearchResults` имеются разнообразные параметры, позволяющие ограничить поиск некоторыми атрибутами. Модифицируйте написанную нами функцию так, чтобы она поддерживала эту возможность, и попробуйте найти только лэптопы с процессором Core Duo.

9

Более сложные способы классификации: ядерные методы и машины опорных векторов

В предыдущих главах мы рассмотрели несколько классификаторов: деревья решений, байесовские классификаторы и нейронные сети. Сейчас мы ознакомимся с линейными классификаторами и ядерными методами, и это послужит прелюдией к одному из самых продвинутых методов классификации, который все еще является предметов активных исследований, – *машинам опорных векторов* (Support Vector Machines – SVM).

На протяжении этой главы мы будем работать с набором данных для задачи о подборе пар на сайте знакомств. Если имеется информация о двух людях, можно ли предсказать, составят ли они хорошую пару? Задача интересна, поскольку в ней много переменных, числовых и дискретных, и немало нелинейных взаимосвязей. На этом наборе данных мы продемонстрируем некоторые слабости описанных ранее классификаторов и покажем, как можно слегка модифицировать набор, чтобы он лучше подходил для этих алгоритмов. Важный урок, который вы должны вынести из этой главы, заключается в том, что ситуация, когда вы подаете на вход алгоритма сложный набор данных и ожидаете, что он научится его точно классифицировать, на практике встречается редко. Для получения хороших результатов зачастую необходимо правильно выбрать алгоритм и осуществить предварительную обработку данных. Надеюсь, что после знакомства с описанной ниже процедурой модификации набора данных у вас появятся собственные идеи о том, как делать это в будущем.

В конце главы вы узнаете, как построить набор данных о реальных людях с помощью сайта популярной социальной сети Facebook, и воспользуетесь представленными алгоритмами, чтобы спрогнозировать, смогут ли люди с определенными характеристиками подружиться.

Набор данных для подбора пар

Используемый в этой главе набор данных относится к гипотетическому сайту знакомств. На большинстве таких сайтов собирается интересная информация об участниках, в том числе демографического характера, об интересах и о поведении. Предположим, что собираются следующие данные:

- Возраст.
- Курит или нет?
- Хочет ли иметь детей?
- Перечень интересов.
- Местонахождение.

Кроме того, на сайте собирается информация о том, составили ли два человека хорошую пару, контактировали ли они между собой и решили ли встретиться «в реале». На основе этих данных создается набор для алгоритма подбора пар. Я подготовил два файла для загрузки:

http://kiwitobes.com/matchmaker/agesonly.csv

http://kiwitobes.com/matchmaker/matchmaker.csv

Файл matchmaker.csv выглядит следующим образом:

```
39,yes,no,skiing:knitting:dancing,220 W 42nd St New York
 NY,43,no,yes,soccer:reading:scrabble,824 3rd Ave New York NY,0
23,no,no,football:fashion,102 1st Ave New York
 NY,30,no,no,snowboarding:knitting:computers:shopping:tv:travel,
 151 W 34th St New York NY,1
50,no,no,fashion:opera:tv:travel,686 Avenue of the Americas
 New York NY,49,yes,yes,soccer:fashion:photography:computers:
 camping:movies:tv,824 3rd Ave New York NY,0
```

В каждой строке содержится информация об одном мужчине и одной женщине, а 1 или 0 в последнем столбце обозначает, считается ли эта пара хорошей. (Автор осознает, что здесь принято много упрощающих предположений; жизнь всегда оказывается сложнее компьютерных моделей.) Для сайта с большим количеством профилей эту информацию можно использовать для построения прогностического алгоритма, который поможет человеку подыскать себе подходящую пару. Кроме того, можно будет найти, какого сорта люди на сайте не представлены; это полезно для выработки стратегии целенаправленного привлечения новой аудитории. В файле agesonly.csv содержится информация о подборе пар только на основе возраста. Мы воспользуемся им для иллюстрации работы классификаторов, поскольку две переменные визуализировать гораздо проще.

Прежде всего нам понадобится функция для загрузки этого набора данных. Достаточно было бы поместить все поля в список, но для проведения последующих экспериментов мы заведем необязательный параметр, который позволит загружать только некоторые поля. Создайте новый файл advancedclassify.py и включите в него класс `matchrow` и функцию `loadmatch`:

```
class matchrow:
  def __init__(self,row,allnum=False):
    if allnum:
      self.data=[float(row[i]) for i in range(len(row)-1)]
    else:
      self.data=row[0:len(row)-1]
    self.match=int(row[len(row)-1])

def loadmatch(f,allnum=False):
  rows=[]
  for line in file(f):
    rows.append(matchrow(line.split(','),allnum))
  return rows
```

Функция `loadmatch` создает список экземпляров класса `matchrow`, каждый из которых содержит необработанные данные и признак того, составляют ли они пару. Воспользуйтесь этой функцией для загрузки обоих наборов:

```
>>> import advancedclassify
>>> agesonly=advancedclassify.loadmatch('agesonly.csv',allnum=True)
>>> matchmaker=advancedclassify.loadmatch('matchmaker.csv')
```

Затруднения при анализе данных

У этого набора есть две интересные особенности: нелинейность и взаимозависимость переменных. Если по ходу чтения главы 8 вы установили библиотеку *matplotlib* (*http://matplotlib.sourceforge.net*), то получите возможность визуализировать некоторые переменные с помощью класса `advancedclassify` и сгенерировать пару списков. (Этот шаг необязателен для понимания остального материала данной главы.) Введите в интерактивном сеансе такой код:

```
from pylab import *
def plotagematches(rows):
  xdm,ydm=[r.data[0] for r in rows if r.match==1],\
          [r.data[1] for r in rows if r.match==1]
  xdn,ydn=[r.data[0] for r in rows if r.match==0],\
          [r.data[1] for r in rows if r.match==0]

  plot(xdm,ydm,'go')
  plot(xdn,ydn,'ro')

  show( )
```

а затем выполните следующие команды:

```
>>> reload(advancedclassify)
<module 'advancedclassify' from 'advancedclassify.py'>
>>> advancedclassify.plotagematches(agesonly)
```

В результате будет создана *точечная диаграмма*, по одной оси которой отложен возраст мужчины, а по другой – возраст женщины. Точки, соответствующие парам, на ней обозначены кружком, точки, не соответствующие парам, – крестиком. Получившаяся картина изображена на рис. 9.1.

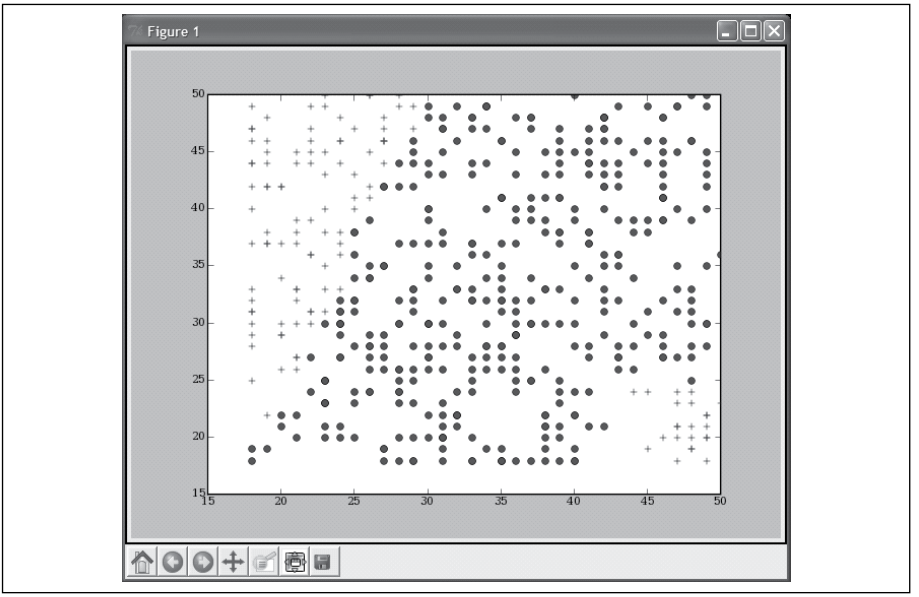


Рис. 9.1. *Точечная диаграмма возраст–возраст*

Составится ли из двух людей хорошая пара, конечно, зависит и от многих других факторов. Этот рисунок получен из набора, содержащего только данные о возрасте, но на нем прослеживается отчетливая граница, показывающая, что люди ищут себе пару примерно своего возраста. Граница эта выглядит криволинейной и по мере увеличения возраста постепенно размывается, то есть чем человек старше, тем он терпимее к возрастным различиям.

Классификатор на основе дерева решений

В главе 7 мы рассматривали способ автоматической классификации данных путем построения деревьев решений. Описанный там алгоритм разбивает данные по некоторой числовой границе. Но при этом возникает проблема, когда критерий разбиения более точно выражается в виде функции от двух переменных. В данном случае для прогнозирования

было бы гораздо удобнее взять разность двух возрастов. Обучение дерева решений непосредственно на исходных данных дало бы результат, изображенный на рис. 9.2.

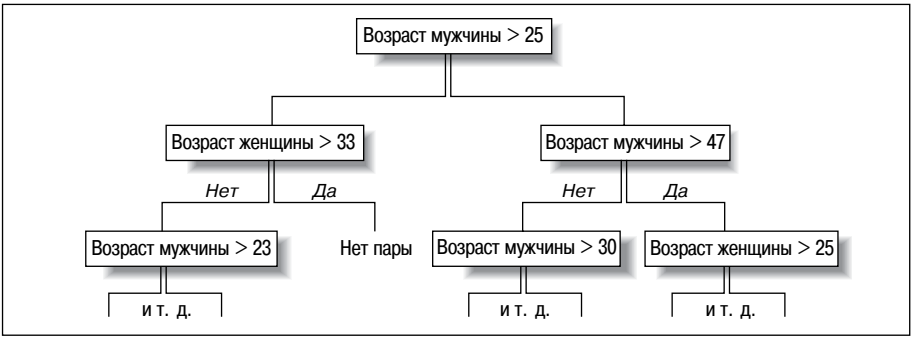


Рис. 9.2. *Дерево решений, соответствующее криволинейной границе*

Очевидно, что для интерпретации это не годится. Для автоматической классификации, возможно, и подошло бы, но она получилась бы слишком беспорядочной и жесткой. А если бы мы ввели в рассмотрение и другие переменные, помимо возраста, то результат оказался бы еще хуже. Чтобы понять, что же делает это дерево, обратимся к точечной диаграмме и *границе решения*, созданной этим деревом (рис. 9.3).

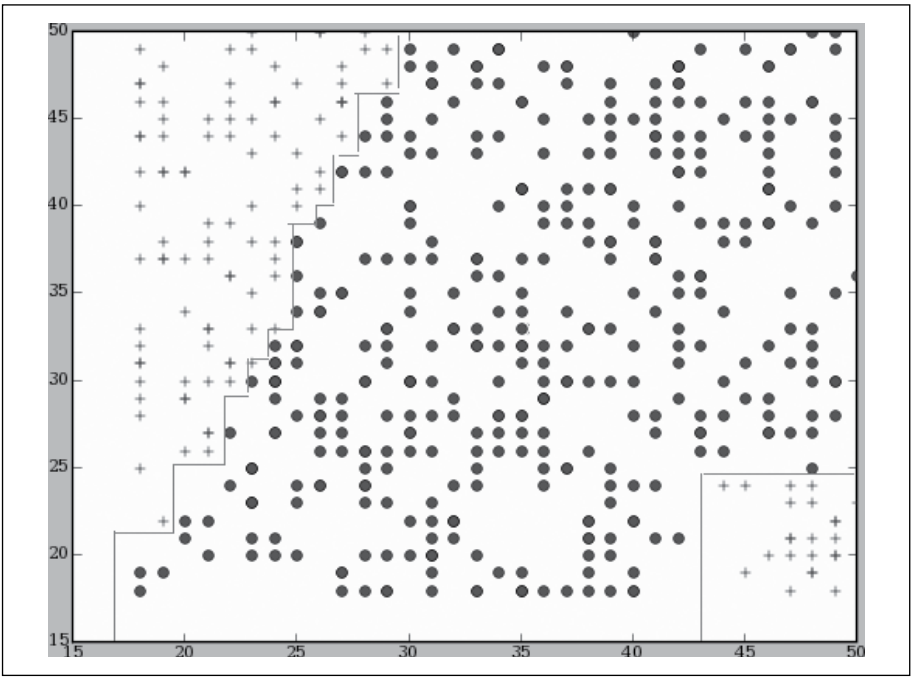


Рис. 9.3. *Граница, созданная деревом решений*

Границей решения называется линия, по одну сторону которой находятся точки, принадлежащие к одной категории, а по другую – к другой. Из рисунка видно, что вследствие ограничений дерева решений граница состоит из горизонтальных и вертикальных отрезков.

Таким образом, следует констатировать два момента. Во-первых, не всегда стоит наивно использовать исходные данные, не понимая, что они означают и как их можно трансформировать для более удобной интерпретации. Генерирование точечной диаграммы может подсказать, как на самом деле разделены данные. Во-вторых, несмотря на свои сильные стороны, описанные в главе 7, деревья решений зачастую неприменимы к задачам, в которых есть несколько числовых переменных, не связанных простыми отношениями.

Простая линейная классификация

Этот классификатор будет совсем простым, но послужит неплохой основой для дальнейшей работы. Он ищет среднее по всем данным в каждом классе и строит точку, представляющую центр этого класса. Новые точки классифицируются по близости к имеющимся центрам.

Нам потребуется функция, которая вычисляет *среднюю точку* класса. В данном случае есть всего два класса, соответствующие 0 и 1. Добавьте функцию `lineartrain` в файл advancedclassify.py:

```
def lineartrain(rows):
  averages={}
  counts={}

  for row in rows:
    # Получить класс данной точки
    cl=row.match

    averages.setdefault(cl,[0.0]*(len(row.data)))
    counts.setdefault(cl,0)

    # Добавить точку к средним
    for i in range(len(row.data)):
      averages[cl][i]+=float(row.data[i])

    # Подсчитываем количество точек в каждом классе
    counts[cl]+=1

  # Делим суммы на счетчики и получаем средние
  for cl,avg in averages.items( ):
    for i in range(len(avg)):
      avg[i]/=counts[cl]

  return averages
```

Выполните эту функцию в интерактивном сеансе и получите средние:

```
>>> reload(advancedclassify)
<module 'advancedclassify' from 'advancedclassify.pyc'>
>>> avgs=advancedclassify.lineartrain(agesonly)
```

Чтобы понять, чем это полезно, обратимся снова к диаграмме возрастных данных, показанной на рис. 9.4.

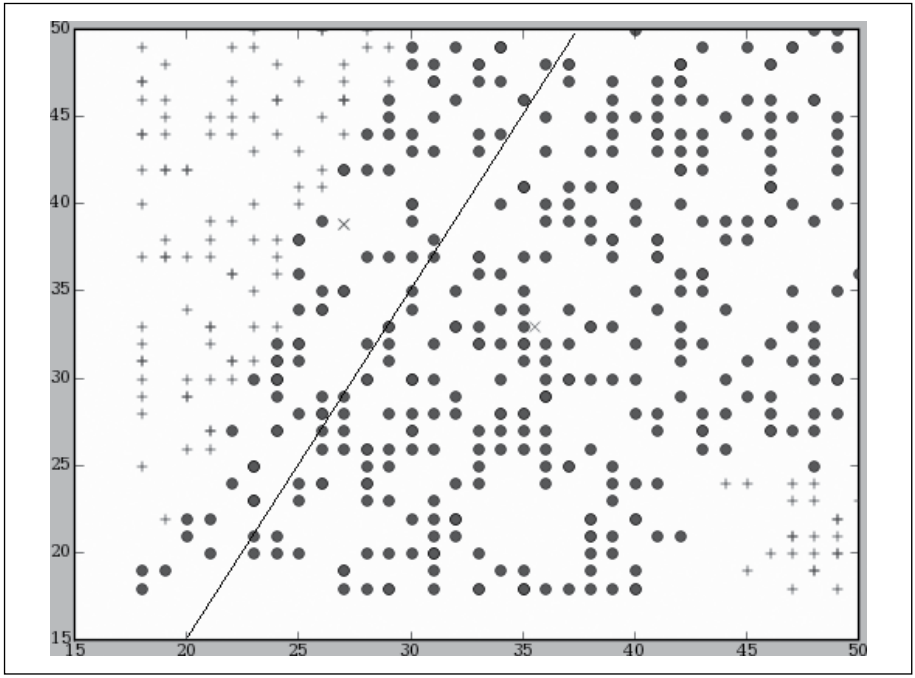


Рис. 9.4. *Линейный классификатор по средним точкам*

Крестиками представлены средние точки, вычисленные функцией `lineartrain`. Линия, разделяющая данные, проходит посредине между двумя крестиками. Это означает, что все точки слева от линии ближе к средней точке класса «не пара», в точки справа от линии – к средней точке класса «пара». Если вы получаете новую пару возрастов и хотите понять, будут ли эти люди составлять пару, то можете нанести соответствующую точку на диаграмму и посмотреть, к какому среднему она окажется ближе.

Определять близость новой точки можно двумя способами. В предыдущих главах вы уже сталкивались с евклидовым расстоянием; мы можем вычислить расстояния между новой точкой и средними точками всех классов и выбрать из них наименьшее. Хотя для данного классификатора этот подход годится, его обобщение потребует использования *векторов* и *скалярных произведений*.

У вектора есть модуль (длина) и направление, часто он изображается в виде стрелки на плоскости или записывается как пара чисел. На рис. 9.5 изображен пример вектора. Здесь же показано, что вычитание одной точки из другой порождает вектор, соединяющий их.

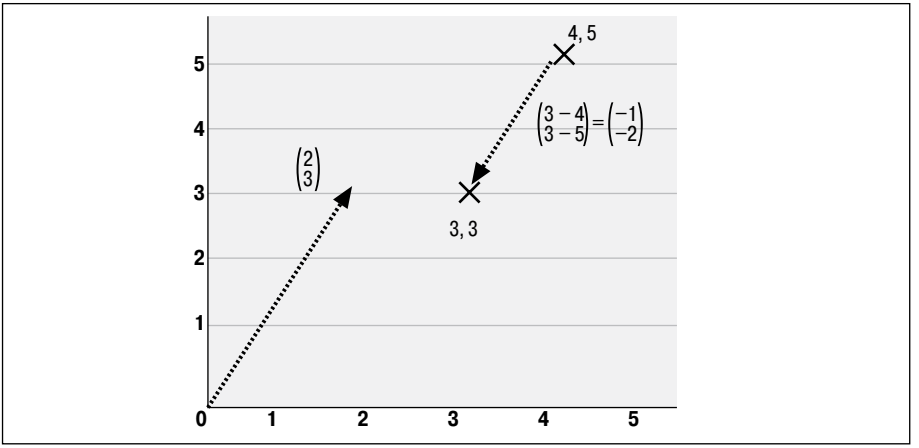


Рис. 9.5. *Примеры векторов*

Скалярное произведение двух векторов – это число, получающееся в результате суммирования попарных произведений координат векторов. Создайте в файле advancedclassify.py новую функцию `dotproduct`:

```
def dotproduct(v1,v2):
return sum([v1[i]*v2[i] for i in range(len(v1))])
```

По-другому скалярное произведение можно вычислить путем умножения длин двух векторов на косинус угла между ними. Самое важное здесь то, что косинус отрицателен, если угол больше 90°, а следовательно, и скалярное произведение в этом случае отрицательно. Чтобы понять, как этим можно воспользоваться, взгляните на рис. 9.6.

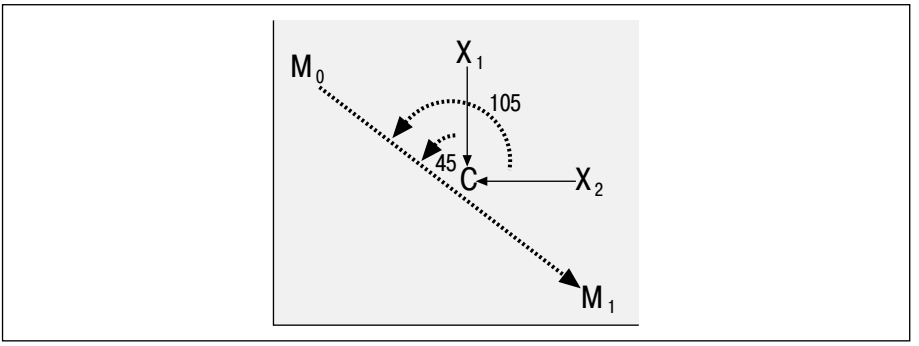


Рис. 9.6. *Использование скалярных произведений для вычисления расстояния*

На этой диаграмме вы видите две средние точки для классов «пара» (M_0) и «не пара» (M_1), а также точку C посредине между ними. Есть и еще две точки, X_1 и X_2, которые надлежит классифицировать. Показан вектор, соединяющий M_0 с M_1, а также векторы, соединяющие X_1 и X_2 с C.

На этом рисунке X_1 расположена ближе к M_0, поэтому классифицируется как «пара». Обратите внимание, что угол между векторами $X_1 \rightarrow C$ и $M_0 \rightarrow M_1$ составляет 45°, то есть меньше 90°, а следовательно, скалярное произведение $X_1 \rightarrow C$ и $M_0 \rightarrow M_1$ положительно.

С другой стороны, угол между $X_2 \rightarrow C$ и $M_0 \rightarrow M_1$ больше 90°, так как векторы направлены в разные стороны. Поэтому их скалярное произведение отрицательно.

Итак, скалярное произведение отрицательно для тупых углов и положительно для острых, поэтому для определения того, к какому классу принадлежит новая точка, достаточно определить знак скалярного произведения.

Точка C лежит посредине между M_0 и M_1, то есть $C = (M_0 + M_1) / 2$, поэтому формула для определения класса выглядит так:

$$\text{Класс} = \text{sign}((X - (M_0 + M_1) / 2) \,.\, (M_0 - M_1))$$

Раскрыв скобки, получаем:

$$\text{Класс} = \text{sign}(X.M_0 - X.M_1 + (M_0.M_0 - M_1.M_1) / 2)$$

По этой формуле мы и будем определять класс. Добавьте в файл advancedclassify.py функцию `dpclassify`:

```
def dpclassify(point,avgs):
  b=(dotproduct(avgs[1],avgs[1])-dotproduct(avgs[0],avgs[0]))/2
  y=dotproduct(point,avgs[0])-dotproduct(point,avgs[1])+b
  if y>0: return 0
  else: return 1
```

Теперь воспользуемся построенным классификатором в интерактивном сеансе и посмотрим, что получится:

```
>>> reload(advancedclassify)
<module 'advancedclassify' from 'advancedclassify.py'>
>>> advancedclassify.dpclassify([30,30],avgs)
1
>>> advancedclassify.dpclassify([30,25],avgs)
1
>>> advancedclassify.dpclassify([25,40],avgs)
0
>>> advancedclassify.dpclassify([48,20],avgs)
1
```

Напомним, что это линейный классификатор, то есть построенная им разделяющая линия – прямая. Следовательно, если прямой, разделяющей данные, не существует или имеется несколько разделов, как в случае попарного сравнения возрастов, то классификатор иногда будет давать неверные ответы. В рассматриваемом примере сравнение

возрастов 48 и 20 не должно было бы дать пару, но поскольку линия только одна и точка находится справа от нее, то функция решает, что это пара. В разделе «Идея ядерных методов» ниже вы увидите, как можно усовершенствовать этот метод, чтобы он умел выполнять и нелинейную классификацию.

Категориальные свойства

Набор данных для подбора пар содержит как числовые, так и дискретные данные. Некоторые классификаторы, например деревья решений, справляются с данными обоих видов без предварительной обработки, но классификаторы, рассматриваемые далее в этой главе, умеют работать только с числовыми данными. Поэтому нужно каким-то образом преобразовать дискретные данные в числовые.

Ответы «да/нет»

Это самый простой случай: ответ «да» преобразуется в 1, ответ «нет» – в –1. При этом еще остается возможность для сопоставления отсутствующему или неоднозначному ответу (например, «не знаю») значения 0. Добавьте в файл advancedclassify.py функцию `yesno`, которая будет выполнять такое преобразование:

```
def yesno(v):
  if v=='yes': return 1
  elif v=='no': return -1
  else: return 0
```

Списки интересов

Представить список интересов человека в наборе данных можно несколькими способами. Самый простой – считать каждый элемент списка отдельной числовой переменной, которая принимает значение 0, если человек выразил интерес, и 1 – в противном случае. Если речь идет об отдельных людях, то это самый лучший подход. Но когда мы имеем дело с парами, то более интуитивным подходом будет взять за переменную количество общих интересов.

Добавьте в файл advancedclassify.py функцию `matchcount`, которая возвращает количество совпавших элементов в двух списках в виде числа с плавающей точкой:

```
def matchcount(interest1,interest2):
  l1=interest1.split(':')
  l2=interest2.split(':')
  x=0
  for v in l1:
    if v in l2: x+=1
  return x
```

Количество общих интересов – любопытная переменная, но очевидно, что она отбрасывает некоторую потенциально полезную информацию. Быть может, некоторые сочетания интересов способствуют образованию хорошей пары, например катание на лыжах и сноубординг или любовь к выпивке и к танцам. Классификатор, который не был обучен на реальных данных, никогда не сможет узнать о таких сочетаниях.

Альтернатива созданию отдельной переменной для каждого интереса (это увеличивает общее число переменных и, следовательно, сложность классификатора) – иерархическая организация интересов. Например, можно сказать, что катание на лыжах и сноубординг – примеры зимних видов спорта, которые, в свою очередь, являются подкатегорией спорта вообще. Если оба члена пары интересуются зимними видами спорта, но не одними и теми же, то `matchcount` добавляет к их индексу сочетаемости не 1, а 0,8. Чем выше приходится подниматься по иерархии, чтобы найти соответствие, тем меньше вклад в индекс. Хотя в наборе данных для подбора пар такой иерархии нет, этот подход заслуживает внимания при решении аналогичных задач.

Вычисление расстояний с помощью сайта Yahoo! Maps

Самая сложная часть обработки рассматриваемого набора данных – местонахождение. Конечно, можно предположить, что чем ближе люди живут, тем больше у них шанс составить пару, но местонахождения в файле данных задаются в виде адресов и почтовых индексов. Проще всего было бы определить переменную «проживают в регионах с одним и тем же почтовым индексом», но такое условие было бы чересчур ограничительным – даже дома, расположенные в соседних кварталах, могут иметь разные почтовые индексы. В идеале хотелось бы, чтобы переменная отражала расстояние.

Конечно, точно вычислить расстояние между двумя адресами без дополнительной информации невозможно. Но, к счастью, сайт Yahoo! Maps предлагает службу *Geocoding*, которая по адресу в США возвращает широту и долготу. Получив эти данные для двух адресов, можно приблизительно вычислить расстояние между ними.

Если по какой-то причине вы не можете воспользоваться API службы Yahoo!, просто добавьте в файл advancedclassify.py такую заглушку:

```
def milesdistance(a1,a2):
  return 0
```

Получение ключа разработчика для Yahoo!

Для доступа к API службы Yahoo! необходимо сначала получить ключ разработчика, который будет включаться в запросы для идентификации вашего приложения. Чтобы его получить, нужно зайти на страницу *http://api.search.yahoo.com/webservices/register_application* и ответить на несколько вопросов. Если у вас еще нет учетной записи в Yahoo!, создайте ее. Ключ вы получите сразу же, ждать ответа по электронной почте не придется.

Работа с Geocoding API

Для обращения к API службы Geocoding необходимо задать URL вида *http://api.local.yahoo.com/MapsService/V1/geocode?appid=appid&location=location*.

Параметр location – это местонахождение в свободном формате: адрес, почтовый индекс или даже только город и штат. В ответ возвращается XML-документ:

```
<ResultSet>
<Result precision="address">
<Latitude>37.417312</Latitude>
<Longitude>-122.026419</Longitude>
<Address>755 FIRST AVE</Address>
<City>SUNNYVALE</City>
<State>CA</State>
<Zip>94089-1019</Zip>
<Country>US</Country>
</Result>
</ResultSet>
```

Вас интересуют в нем поля `Longitude` (Долгота) и `Latitude` (Широта). Для разбора документа воспользуемся библиотекой minidom, с которой мы уже работали в предыдущих главах. Добавьте в файл advancedclassify.py функцию `getlocation`:

```
yahookey="Ваш ключ"
from xml.dom.minidom import parseString
from urllib import urlopen,quote_plus

loc_cache={}
def getlocation(address):
  if address in loc_cache: return loc_cache[address]
  data=urlopen('http://api.local.yahoo.com/MapsService/V1/'+\
               'geocode?appid=%s&location=%s' %
               (yahookey,quote_plus(address))).read( )
  doc=parseString(data)
  lat=doc.getElementsByTagName('Latitude')[0].firstChild.nodeValue
  long=doc.getElementsByTagName('Longitude')[0].firstChild.nodeValue
  loc_cache[address]=(float(lat),float(long))
  return loc_cache[address]
```

Эта функция конструирует URL, содержащий ваш ключ и местонахождение, а затем извлекает из полученного документа широту и долготу. Хотя для вычисления расстояния больше ничего не нужно, API службы Yahoo! Geocoding можно применять и в других целях, например чтобы найти почтовый индекс по адресу или узнать, какому региону соответствует заданный почтовый индекс.

Вычисление расстояния

Получить точное расстояние между двумя точками, заданными широтой и долготой, довольно сложно. Но в нашем случае расстояния

относительно невелики и нужны только для сравнения, поэтому достаточно приближенных значений. Для аппроксимации применяется евклидово расстояние, только разность широт предварительно умножается на 69,1, а разность долгот – на 53.

Добавьте функцию `milesdistance` в файл advancedclassify.py:

```
def milesdistance(a1,a2):
  lat1,long1=getlocation(a1)
  lat2,long2=getlocation(a2)
  latdif=69.1*(lat2-lat1)
  longdif=53.0*(long2-long1)
  return (latdif**2+longdif**2)**.5
```

Здесь сначала вызывается написанная ранее функция `getlocation` для получения координат обоих адресов, а потом вычисляется расстояние между ними. Если хотите, можете протестировать функцию в интерактивном сеансе:

```
>>> reload(advancedclassify)
<module 'advancedclassify' from 'advancedclassify.py'>
>>> advancedclassify.getlocation('1 alewife center, cambridge, ma')
(42.398662999999999, -71.140512999999999)
>>> advancedclassify.milesdistance('cambridge, ma','new york,ny')
191.77952424273104
```

Обычно погрешность вычисления расстояний с помощью этой функции не превышает 10%, что для нашего приложения вполне приемлемо.

Создание нового набора данных

Теперь все готово для создания набора данных, пригодного для обучения классификатора. Необходима лишь функция, которая сведет все составные части воедино. Она загрузит данные с помощью `loadmatch` и преобразует их в набор столбцов. Добавьте функцию `loadnumerical` в файл advancedclassify.py:

```
def loadnumerical( ):
  oldrows=loadmatch('matchmaker.csv')
  newrows=[]
  for row in oldrows:
    d=row.data
    data=[float(d[0]),yesno(d[1]),yesno(d[2]),
          float(d[5]),yesno(d[6]),yesno(d[7]),
          matchcount(d[3],d[8]),
          milesdistance(d[4],d[9]),
          row.match]
    newrows.append(matchrow(data))
  return newrows
```

Эта функция генерирует новую строку данных из каждой строки исходного набора. Для преобразования данных в числа она пользуется ранее написанными функциями, в том числе для вычисления расстояний и счетчика перекрытия интересов.

Вызовите ее в интерактивном сеансе, чтобы создать новый набор данных:

```
>>> reload(advancedclassify)
>>> numericalset=advancedclassify.loadnumerical( )
>>> numericalset[0].data
[39.0, 1, -1, 43.0, -1, 1, 0, 0.90110601059793416]
```

При желании легко создать поднаборы, указав интересующие вас столбцы. Это полезно для визуализации данных и чтобы понять, как классификатор работает для различных переменных.

Масштабирование данных

Когда для сравнения использовался только возраст, нам было достаточно исходных данных, по которым вычислялись средние и расстояния; ведь сравнивать переменные, обозначающие одно и то же, вполне допустимо. Однако теперь мы ввели несколько новых переменных, которые с возрастом несравнимы, так как принимают гораздо меньшие значения. Разное отношение к детям (возможные значения 1 и –1, максимальное расстояние 2) может быть куда весомее шестилетней разницы в возрасте, но если брать данные как есть, то разница в возрасте будет весить в три раза больше.

Для решения этой проблемы необходимо привести все данные к единому масштабу, чтобы все переменные можно было осмысленно сравнивать. Для этого требуется определить минимальное и максимальное значения каждой переменной и масштабировать данные так, чтобы минимум соответствовал нулю, а максимум – единице.

Добавьте функцию `scaledata` в файл advancedclassifier.py:

```
def scaledata(rows):
  low=[999999999.0]*len(rows[0].data)
  high=[-999999999.0]*len(rows[0].data)
  # Ищем наименьшее и наибольшее значения
  for row in rows:
    d=row.data
    for i in range(len(d)):
      if d[i]<low[i]: low[i]=d[i]
      if d[i]>high[i]: high[i]=d[i]

  # Создаем функцию масштабирования
  def scaleinput(d):
    return [(d.data[i]-low[i])/(high[i]-low[i])
            for i in range(len(low))]

  # Масштабируем данные
  newrows=[matchrow(scaleinput(row.data)+[row.match])
           for row in rows]

  # Возвращаем новые данные и функцию
  return newrows,scaleinput
```

Внутри этой функции определяется функция `scaleinput`, которая ищет наименьшее значение и вычитает его из всех остальных, так что минимум оказывается равным 0. Затем все значения делятся на разность между максимальным и минимальным, чтобы привести их к диапазону от 0 до 1. Функция `scaledata` применяет `scaleinput` к каждой строке набора данных и возвращает новый набор вместе с самóй функцией `scaleinput`, которую вы сможете в дальнейшем использовать для масштабирования запросов.

Теперь попробуем применить линейный классификатор к более широкому набору переменных:

```
>>> reload(advancedclassify)
<module 'advancedclassify' from 'advancedclassify.py'>
>>> scaledset,scalef=advancedclassify.scaledata(numericalset)
>>> avgs=advancedclassify.lineartrain(scaledset)
>>> numericalset[0].data
[39.0, 1, -1, 43.0, -1, 1, 0, 0.90110601059793416]
>>> numericalset[0].match
0
>>> advancedclassify.dpclassify(scalef(numericalset[0].data),avgs)
1
>>> numericalset[11].match
1
>>> advancedclassify.dpclassify(scalef(numericalset[11].data),avgs)
1
```

Отметим, что числовые данные предварительно масштабируются. Хотя для некоторых примеров этот классификатор работает, недостатки проведения разделяющей прямой становятся более очевидны. Чтобы улучшить результат, нам необходимо выйти за рамки линейной классификации.

Идея ядерных методов

Посмотрите, что получится, если применить линейный классификатор к набору данных, подобному изображенному на рис. 9.7.

Где окажутся средние точки каждого класса? В точности в одном и том же месте! Хотя и вам, и мне ясно, что все точки внутри круга – крестики, а вне него – кружки, линейный классификатор не в состоянии различить эти классы.

Но что если сначала возвести координаты x и y каждой точки в квадрат? Точка, которая имела координаты (–1, 2), теперь превратится в (1, 4), точка (0,5, 1) – в (0,25, 1) и т. д. Новая диаграмма изображена на рис. 9.8.

Теперь все крестики переместились в угол, а все нолики оказались вне угла. Разделить их прямой линией стало совсем просто, а для классификации новой точки достаточно возвести ее координаты в квадрат и посмотреть, по какую сторону от прямой она окажется.

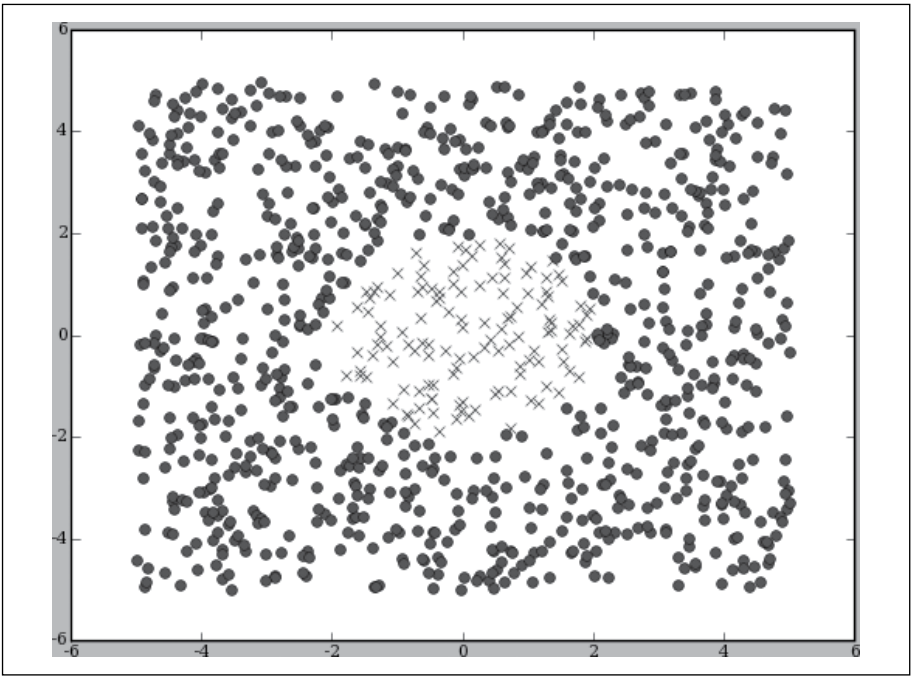


Рис. 9.7. *Один класс окружен другим*

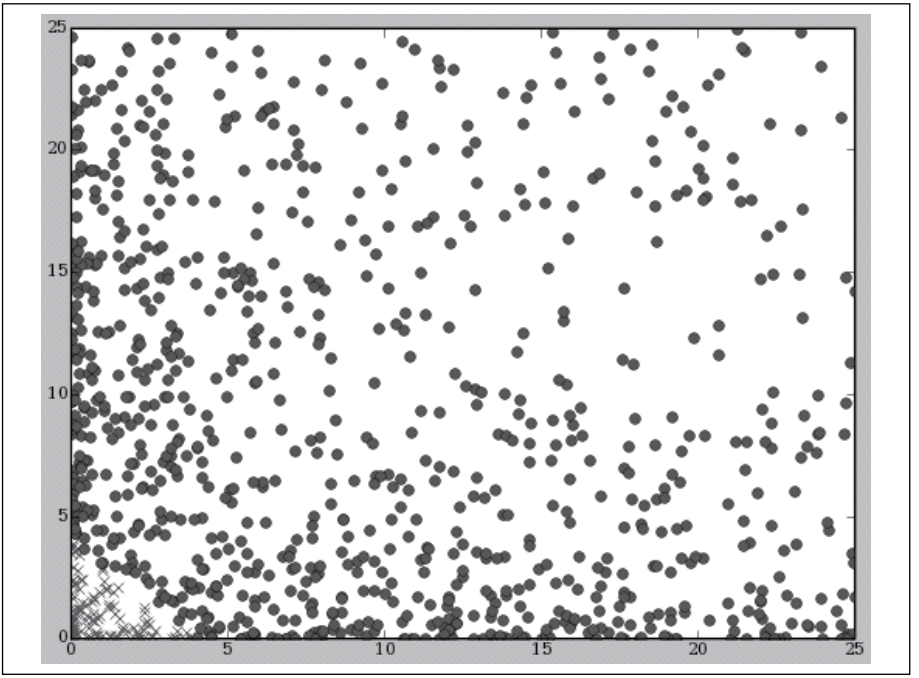


Рис. 9.8. *Перемещение точек в другое пространство*

Этот пример показывает, что путем предварительной трансформации точек можно создать новый набор данных, допускающий разделение прямой линией. Однако это специально подобранный пример, для которого трансформация описывается просто; в реальных задачах она оказывается куда сложнее, к тому же производится в пространстве большей размерности. Например, можно взять двумерный набор данных с координатами x и y и трансформировать его в трехмерный по формулам $a = x \hat{} 2$, $b = x * y$, $c = y \hat{} 2$. Увеличив размерность задачи, иногда становится проще найти разделяющую два класса поверхность.

Переход к ядру

Хотя для перевода данных в новое пространство можно написать код, подобный приведенному выше, на практике так поступают редко, потому что при работе с реальными наборами данных для поиска разделителя иногда требуется переходить в пространство с сотнями и тысячами измерений, что вряд ли реализуемо. Однако для любого алгоритма, в котором используются скалярные произведения, – в том числе и для линейного классификатора – можно применить метод *перехода к ядру* (kernel trick).

Переход к ядру подразумевает замену скалярного произведения другой функцией, которая возвращает число, равное величине скалярного произведения, в случае если бы данные сначала были каким-то образом отображены на пространство с большим числом измерений. На отображение не накладывается никаких ограничений, но на практике применяется лишь небольшое число трансформаций. Чаще всего рекомендуют (и мы этой рекомендацией воспользуемся) так называемую *функцию радиального базиса*.

Функция радиального базиса аналогична скалярному произведению в том смысле, что принимает два вектора и возвращает число. Но, в отличие от скалярного произведения, она нелинейна и, следовательно, может использоваться для отображения более сложных пространств. Добавьте функцию `rbf` в файл advancedclassify.py:

```
def rbf(v1,v2,gamma=20):
  dv=[v1[i]-v2[i] for i in range(len(v1))]
  l=veclength(dv)
  return math.e**(-gamma*l)
```

Эта функция принимает единственный параметр `gamma`, который можно подстраивать для поиска оптимального линейного разделителя.

Далее нам понадобится новая функция, которая вычисляет расстояния между средними точками в трансформированном пространстве. К сожалению, средние вычислялись в исходном пространстве, поэтому здесь мы ими воспользоваться не можем. Более того, средние вообще невозможно вычислить, потому что мы не вычисляем координаты точек в новом пространстве. Но, на наше счастье, усреднение по множеству

векторов и вычисление скалярного произведения результата усреднения с вектором A дает тот же результат, что усреднение скалярных произведений вектора A со всеми векторами из того же множества.

Поэтому, вместо того чтобы вычислять скалярное произведение классифицируемой точки со средней точкой класса, можно вычислить скалярные произведения или функции радиального базиса для данной точки и всех остальных точек класса, а затем выполнить усреднение. Добавьте функцию `nlclassify` в файл advancedclassify.py:

```
def nlclassify(point,rows,offset,gamma=10):
  sum0=0.0
  sum1=0.0
  count0=0
  count1=0

  for row in rows:
    if row.match==0:
      sum0+=rbf(point,row.data,gamma)
      count0+=1
    else:
      sum1+=rbf(point,row.data,gamma)
      count1+=1
  y=(1.0/count0)*sum0-(1.0/count1)*sum1+offset

  if y<0: return 0
  else: return 1

def getoffset(rows,gamma=10):
  l0=[]
  l1=[]
  for row in rows:
    if row.match==0: l0.append(row.data)
    else: l1.append(row.data)
  sum0=sum(sum([rbf(v1,v2,gamma) for v1 in l0]) for v2 in l0)
  sum1=sum(sum([rbf(v1,v2,gamma) for v1 in l1]) for v2 in l1)

  return (1.0/(len(l1)**2))*sum1-(1.0/(len(l0)**2))*sum0
```

Величина смещения в трансформированном пространстве тоже отличается, и на ее вычисление может уйти заметное время. Поэтому ее следует вычислять для набора данных один раз и передавать при каждом вызове `nlclassify`:

Попробуйте воспользоваться новым классификатором для набора данных о возрасте и посмотрите, решает ли он обнаруженную ранее проблему:

```
>>> advancedclassify.nlclassify([30,30],agesonly,offset)
1
>>> advancedclassify.nlclassify([30,25],agesonly,offset)
1
>>> advancedclassify.nlclassify([25,40],agesonly,offset)
0
```

```
>>> advancedclassify.nlclassify([48,20],agesonly,offset)
0
```

Блестяще! После трансформации классификатор сумел понять, что существует полоса пар с близкими возрастами и что по обе стороны от этой полосы составление пары крайне маловероятно. Теперь он распознает, что 48 и 20 не подходят для составления пары. Попробуем еще раз, включив и другие данные:

```
>>> ssoffset=advancedclassify.getoffset(scaledset)
>>> numericalset[0].match
0
>>> advancedclassify.nlclassify(scalef(numericalset[0].data),scaledset,
ssoffset)
0
>>> numericalset[1].match
1
>>> advancedclassify.nlclassify(scalef(numericalset[1].data),scaledset,
ssoffset)
1
>>> numericalset[2].match
0
>>> advancedclassify.nlclassify(scalef(numericalset[2].data),scaledset,
ssoffset)
0
>>> newrow=[28.0,-1,-1,26.0,-1,1,2,0.8] # Мужчина не хочет иметь детей,
а женщина хочет
>>> advancedclassify.nlclassify(scalef(newrow),scaledset,ssoffset)
0
>>> newrow=[28.0,-1,1,26.0,-1,1,2,0.8] # Оба хотят иметь детей
>>> advancedclassify.nlclassify(scalef(newrow),scaledset,ssoffset)
1
```

Точность классификации заметно улучшилась. Видно, что если мужчина не хочет иметь детей, а женщина хочет, то пары не получится, пусть даже они близки по возрасту и имеют общие интересы. Попробуйте поиграть с другими переменными и посмотрите, как это отразится на результате.

Метод опорных векторов

Рассмотрим снова задачу отыскания прямой, разделяющей два класса. На рис. 9.9 приведен пример. Показаны средние точки для каждого класса и соответствующий им разделитель.

Обратите внимание, что разделяющая линия, вычисленная на основе средних, неправильно классифицирует две точки, потому что они оказались к ней гораздо ближе, чем бóльшая часть данных. Проблема в том, что, поскольку почти все данные находятся далеко от разделителя, эти точки не учитывались при его построении.

Для создания классификаторов, решающих эту проблему, широко применяется метод опорных векторов. Смысл его в том, чтобы найти

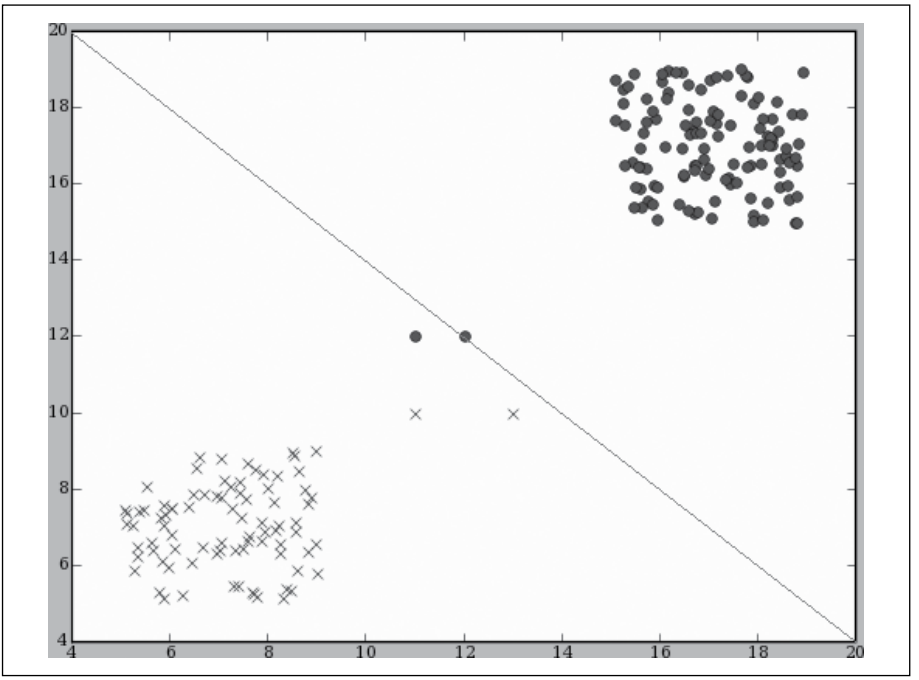


Рис. 9.9. *Линейный классификатор по средним неправильно классифицирует точки*

линию, которая отстояла бы максимально далеко от каждого класса. Эта линия, называемая *гиперплоскостью с максимальным отступом*, изображена на рис. 9.10.

Разделитель выбран так, чтобы параллельные линии, задевающие точки из каждого класса, отстояли от него максимально далеко. Как и раньше, для классификации новой точки достаточно посмотреть, по какую сторону от разделителя она оказалась. Обратите внимание, что для вычисления положения разделителя нужны только точки, отстоящие от него на величину отступа; все остальные данные можно вообще отбросить, а положение разделителя от этого не изменится. Точки, лежащие близко к разделителю, называются *опорными векторами*. Алгоритм, который находит опорные векторы и по ним строит разделитель, называется *машиной опорных векторов*.

Вы уже видели, как линейный классификатор можно превратить в нелинейный с помощью перехода к ядру при условии, что сравнение выполняется с помощью скалярного произведения. Машины опорных векторов тоже пользуются скалярным произведением, поэтому могут применяться совместно с ядерными алгоритмами для нелинейной классификации.

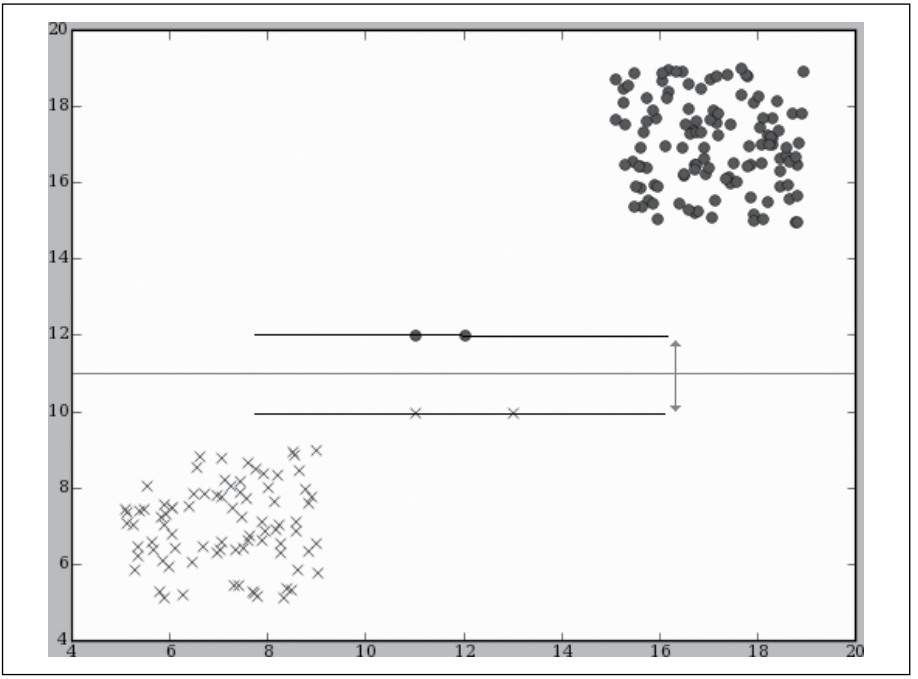


Рис. 9.10. *Нахождение наилучшего разделителя*

Применения машин опорных векторов

Поскольку машины опорных векторов хорошо работают с многомерными наборами данных, то чаще всего они применяются при решении научных и других задач, требующих трудоемкой обработки очень сложных данных, например:

- Классификация выражений лица.
- Обнаружений вторжений на военные объекты.
- Прогнозирование структуры белков по их последовательности.
- Распознавание рукописных данных.
- Определение потенциального ущерба от землетрясения.

Библиотека LIBSVM

Из предыдущего раздела вам, наверное, ясно, как и почему работают машины опорных векторов, но в их обучении задействованы математические методы, требующие огромного объема вычислений и выходящие далеко за рамки этой главы. Поэтому мы воспользуемся библиотекой LIBSVM с открытыми исходными текстами, которая позволяет

обучить SVM-модель, делать прогнозы и проверять их на том же наборе данных. В нее даже встроена поддержка функции радиального базиса и других ядерных методов.

Получение LIBSVM

Скачать библиотеку LIBSVM можно со страницы *http://www.csie.ntu.edu.tw/~cjlin/libsvm*.

Она написана на языке C++, но имеется также версия на Java. В дистрибутив входит обертка svm.py для вызова из программ на Python. Чтобы ею воспользоваться, необходима версия LIBSVM, откомпилированная под вашу платформу. Для Windows в дистрибутиве имеется готовая DLL – svmc.dll. (Для версии Python 2.5 этот файл нужно переименовать в svmc.pyd, так как она не умеет импортировать библиотеки с расширением DLL.) В документации, поставляемой вместе с LIBSVM, описывается, как откомпилировать ее на других платформах.

Пример сеанса

Поместите откомпилированную версию LIBSVM и пакет svm.py в свою рабочую папку или в папку, где интерпретатор Python ищет библиотеки. Теперь можно импортировать ее и попытаться решить простую задачку:

```
>>> from svm import *
```

Первым делом создадим простой набор данных. LIBSVM читает данные из кортежа, содержащего два списка. Первый список содержит классы, второй – исходные данные. Попробуем создать набор данных всего с двумя классами:

```
>>> prob = svm_problem([1,-1],[[1,0,1],[-1,0,-1]])
```

Еще нужно с помощью функции `svm_parameter` задать ядро, которым вы собираетесь пользоваться:

```
>>> param = svm_parameter(kernel_type = LINEAR, C = 10)
```

Далее следует обучить модель:

```
>>> m = svm_model(prob, param)
*
optimization finished, #iter = 1
nu = 0.025000
obj = -0.250000, rho = 0.000000
nSV = 2, nBSV = 0
Total nSV = 2
```

И наконец воспользуемся ею для прогнозирования принадлежности классам:

```
>>> m.predict([1, 1, 1])
1.0
```

Это вся функциональность LIBSVM, которая вам нужна для создания модели по обучающим данным и последующего прогнозирования.

У библиотеки LIBSVM есть еще одна полезная особенность – возможность сохранять и загружать обученные модели:

```
>>> m.save(test.model)
>>> m=svm_model(test.model)
```

Применение метода опорных векторов к набору данных для подбора пар

Чтобы применить библиотеку LIBSVM к набору данных для подбора пар, следует преобразовать его в кортеж списков, необходимый функции `svm_model`. Это простое преобразование можно записать в одной строке:

```
>>> answers,inputs=[r.match for r in scaledset],[r.data for r in scaledset]
```

Мы снова применили масштабирование, чтобы избежать приписывания переменным чрезмерно большого веса, поскольку это улучшает результаты работы алгоритма. Воспользуемся описанными выше функциями для генерирования нового набора данных и построения модели, взяв в качестве ядра функцию радиального базиса:

```
>>> param = svm_parameter(kernel_type = RBF)
>>> prob = svm_problem(answers,inputs)
>>> m=svm_model(prob,param)
*
optimization finished, #iter = 319
nu = 0.777538
obj = -289.477708, rho = -0.853058
nSV = 396, nBSV = 380
Total nSV = 396
```

Теперь можно делать прогнозы относительно того, составят ли пару два человека с заданными атрибутами. Необходимо вызывать функцию `scale` для масштабирования оцениваемых данных, чтобы переменные были выражены в том же масштабе, что и построенная модель:

```
>>> newrow=[28.0,-1,-1,26.0,-1,1,2,0.8] # Мужчина не хочет иметь детей,
а женщина хочет
>>> m.predict(scalef(newrow))
0.0
>>> newrow=[28.0,-1,1,26.0,-1,1,2,0.8] # Оба хотят иметь детей
>>> m.predict(scalef(newrow))
1.0
```

Хотя прогноз выглядит неплохо, хотелось бы знать, насколько он в действительности хорош, и подобрать оптимальные параметры для функции базиса. В библиотеку LIBSVM встроена возможность перекрестного контроля моделей. Как этот механизм работает, вы видели в главе 8, – набор данных автоматически разбивается на обучающий и тестовый. Обучающий набор используется для построения модели, а тестовый – для проверки того, насколько хорошо модель выполняет прогнозирование.

Проверим качество модели с помощью перекрестного контроля. Функция принимает параметр `n` и разбивает набор данных на *n* частей. Затем

каждая часть последовательно используется в качестве тестового набора, в то время как остальные служат для обучения. Функция возвращает список ответов, который вы можете сравнить с исходным списком.

```
>>> guesses = cross_validation(prob, param, 4)
...
>>> guesses
[0.0, 0.0, 0.0, 0.0, 1.0, 0.0,...
0.0, 0.0, 0.0, 0.0, 0.0, 0.0,...
1.0, 1.0, 0.0, 0.0, 0.0, 0.0,...
...]
>>> sum([abs(answers[i]-guesses[i]) for i in range(len(guesses))])
116.0
```

Количество расхождений между ответами и прогнозами равно 116. Поскольку в исходном наборе всего 500 строк, то 384 прогноза оказались правильными. Если хотите, можете прочитать в документации по LIBSVM про другие ядра и параметры и посмотреть, удастся ли добиться улучшения путем изменения `param`.

Подбор пар на сайте Facebook

Сайт Facebook – это популярная социальная сеть, которая первоначально была ориентирована на студентов колледжей, но затем открылась для более широкой аудитории. Как и другие социальные сети, она позволяет пользователям создавать профили, вводить о себе демографическую информацию и общаться с друзьями на сайте. Кроме того, Facebook предоставляет API, позволяющий запрашивать информацию о людях и узнавать, являются ли два указанных человека друзьями. С помощью этого API можно построить реальный набор данных для подбора пар.

Во время работы над этой книгой сайт Facebook очень трепетно относился к конфиденциальности, поэтому смотреть можно лишь профили собственных друзей. API следует таким же правилам, то есть требует наличия учетной записи и ограничивает возможности запросов. Поэтому, чтобы проработать пример из этого раздела, вы должны создать учетную запись на сайте Facebook и обзавестись по меньшей мере 20 друзьями.

Получение ключа разработчика

Если у вас есть учетная на сайте Facebook, то вы можете запросить ключ разработчика на странице *http://developers.facebook.com*.

Вы получите две строки – ключ для доступа к API и секретный ключ. Первый необходим для идентификации, второй – для созданий свертки запросов, о чем мы поговорим ниже. Создайте новый файл facebook.ру, импортируйте в него необходимые модули и задайте некоторые константы:

```
import urllib,md5,webbrowser,time
from xml.dom.minidom import parseString
```

```
apikey="Ваш ключ для доступа к API"
secret="Ваш секретный ключ"
FacebookSecureURL = "https://api.facebook.com/restserver.php"
```

Добавим еще два вспомогательных метода: `getsinglevalue` получает следующее значение из поименованного узла, а `callid` возвращает число, сгенерированное на основе системного таймера.

```
def getsinglevalue(node,tag):
  nl=node.getElementsByTagName(tag)
  if len(nl)>0:
    tagNode=nl[0]
    if tagNode.hasChildNodes( ):
      return tagNode.firstChild.nodeValue
  return ''

def callid( ):
  return str(int(time.time( )*10))
```

Для некоторых вызовов API сайта Facebook требуется посылать порядковый номер, в качестве которого можно взять любое число, лишь бы оно было больше предыдущего номера. Системный таймер позволяет легко удовлетворить этому условию.

Создание сеанса

Процедура создания сеанса на сайте Facebook спроектирована так, что позволяет создавать приложение, которым смогут воспользоваться другие люди, хотя их имя и пароль вам неизвестны. Достигается это путем выполнения нескольких шагов:

1. С помощью API сайта Facebook запросить маркер.
2. Послать на страницу регистрации запрос, в URL которого указан маркер.
3. Подождать, пока пользователь зарегистрируется.
4. Запросить у API сайта Facebook сеанс, указав маркер.

Поскольку в этих вызовах используется несколько общих переменных, лучше обернуть их в класс. Создайте в файле facebook.py новый класс `fbsession` и включите в него метод `__init__`, реализующий перечисленные выше шаги:

```
class fbsession:
  def __init__(self):
    self.session_secret=None
    self.session_key=None
    self.token=self.createtoken( )
    webbrowser.open(self.getlogin( ))
    print "После регистрации нажмите клавишу Enter:",
    raw_input( )
    self.getsession( )
```

Метод `__init__` вызывает несколько других методов, которые нам предстоит добавить в класс. Прежде всего необходим способ отправки запросов

API сайта Facebook. Метод `sendrequest` открывает соединение с сайтом Facebook и посылает запрос, содержащий указанные аргументы. Возвращаемый документ в формате XML разбирается с помощью библиотеки minidom. Добавьте этот метод в класс:

```
def sendrequest(self, args):
  args['api_key'] = apikey
  args['sig'] = self.makehash(args)
  post_data = urllib.urlencode(args)
  url = FacebookURL + "?" + post_data
  data=urllib.urlopen(url).read( )
  return parseString(data)
```

В строке, выделенной полужирным шрифтом, генерируется сигнатура запроса. Для этого вызывается метод `makehash`, который конкатенирует все параметры в одну строку и вычисляет ее свертку с помощью секретного ключа. Скоро вы увидите, что после получения сеанса секретный ключ изменяется, поэтому метод проверяет, есть ли у вас уже секретный ключ сеанса. Добавьте метод `makehash` в класс:

```
def makehash(self,args):
    hasher = md5.new(''.join([x + '=' + args[x] for x in sorted(args.
keys( ))]))
  if self.session_secret: hasher.update(self.session_secret)
  else: hasher.update(secret)
  return hasher.hexdigest( )
```

Теперь можно обращаться к функциям API сайта Facebook. Начнем с функции `createtoken`, которая создает и сохраняет маркер, необходимый для открытия страницы регистрации:

```
def createtoken(self):
  res = self.sendrequest({'method':"facebook.auth.createToken"})
  self.token = getsinglevalue(res,'token')
```

Добавьте также метод `getlogin`, который просто возвращает URL страницы регистрации:

```
def getlogin(self):
  return "http://api.facebook.com/login.php?api_key="+apikey+\
         "&auth_token=" + self.token
```

После того как пользователь зарегистрировался, следует вызвать метод `getsession` для получения ключа и секретного ключа сеанса, который понадобится для шифрования последующих запросов. Добавьте этот метод в класс:

```
def getsession(self):
  doc=self.sendrequest({'method':'facebook.auth.getSession',
                        'auth_token':self.token})
  self.session_key=getsinglevalue(doc,'session_key')
  self.session_secret=getsinglevalue(doc,'secret')
```

Для подготовки сеанса надо проделать много работы, но дальше все пойдет легче. В этой главе нас будет интересовать только получение информации о людях, но, заглянув в документацию, вы обнаружите, что можно столь же просто загружать фотографии и информацию о событиях.

Загрузка данных о друзьях

Теперь можно перейти к написанию полезных методов. Метод `getfriends` загружает идентификаторы друзей зарегистрировавшегося пользователя и возвращает их в виде списка. Добавьте его в класс `fbsession`:

```
def getfriends(self):
  doc=self.sendrequest({'method':'facebook.friends.get',
                        'session_key':self.session_key,'call_id':callid( )})
  results=[]
  for n in doc.getElementsByTagName('result_elt'):
    results.append(n.firstChild.nodeValue)
  return results
```

Поскольку `getfriends` возвращает только идентификаторы, нужен еще один метод, который загрузит собственно информацию о конкретном человеке. Метод `getinfo` вызывает функцию API `getInfo`, передавая ей список идентификаторов. Он запрашивает всего несколько полей, но при желании вы можете добавить дополнительные поля в строку `fields` и модифицировать код разбора XML так, чтобы он извлекал соответствующую информацию. Полный список полей приведен в документации для разработчика на сайте Facebook:

```
def getinfo(self,users):
  ulist=','.join(users)

  fields='gender,current_location,relationship_status,'+\
         'affiliations,hometown_location'

  doc=self.sendrequest({'method':'facebook.users.getInfo',
  'session_key':self.session_key,'call_id':callid( ),
  'users':ulist,'fields':fields})

  results={}
  for n,id in zip(doc.getElementsByTagName('result_elt'),users):
    # Получить местонахождение
    locnode=n.getElementsByTagName('hometown_location')[0]
   loc=getsinglevalue(locnode,'city')+',  '+getsinglevalue(locnode,'state')

   # Получить название учебного заведения
   college=''
   gradyear='0'
   affiliations=n.getElementsByTagName('affiliations_elt')
   for aff in affiliations:
     # Тип 1 - это колледж
     if getsinglevalue(aff,'type')=='1':
       college=getsinglevalue(aff,'name')
       gradyear=getsinglevalue(aff,'year')

  results[id]={'gender':getsinglevalue(n,'gender'),
               'status':getsinglevalue(n,'relationship_status'),
               'location':loc,'college':college,'year':gradyear}
  return results
```

Результаты представлены в виде словаря, который отображает идентификатор пользователя на набор информации о нем. Этим словарем можно воспользоваться, чтобы создать набор данных для подбора пар. Если хотите, можете протестировать новый класс в интерактивном сеансе:

```
>>> import facebook
>>> s=facebook.fbsession( )
После регистрации нажмите клавишу Enter:
>>> friends=s.getfriends( )
>>> friends[1]
u'iY5TTbS-Ofvs.'
>>> s.getinfo(friends[0:2])
{u'iA81OMUfhfsw.': {'gender': u'Female', 'location': u'Atlanta, '},
u'iY5TTbS-Ofvs.': {'gender': u'Male', 'location': u'Boston, '}}
```

Построение набора данных для подбора пар

Последний вызов API сайта Facebook, который нам потребуется, определяет, являются ли два человека друзьями. Эта информация станет «ответом» в нашем наборе данных. Функции нужно передать два списка идентификаторов одинаковой длины, а вернет она список, содержащий по одному числу для каждой пары, – 1, если это друзья, и 0 в противном случае. Добавьте в класс следующий метод:

```
def arefriends(self,idlist1,idlist2):
  id1=','.join(idlist1)
  id2=','.join(idlist2)
  doc=self.sendrequest({'method':'facebook.friends.areFriends',
                        'session_key':self.session_key,'call_id':callid( ),
                        'id1':id1,'id2':id2})
  results=[]
  for n in doc.getElementsByTagName('result_elt'):
    results.append(n.firstChild.nodeValue)
  return results
```

И соберем все вместе, чтобы создать набор данных для работы с библиотекой LIBSVM. В результате мы получим список друзей зарегистрировавшегося пользователя, загрузим информацию о них и создадим строку для каждой пары. Затем для каждой пары проверим, являются ли они друзьями. Добавьте в класс метод `makedataset`:

```
def makedataset(self):
  from advancedclassify import milesdistance
  # Получить всю информацию о моих друзьях
  friends=self.getfriends( )
  info=self.getinfo(friends)
  ids1,ids2=[],[]
  rows=[]

  # Вложенный цикл для проверки каждой пары
  for i in range(len(friends)):
    f1=friends[i]
    data1=info[f1]
```

```
      # Начинаем с i + 1, чтобы не повторять уже сделанное
      for j in range(i+1,len(friends)):
        f2=friends[j]
        data2=info[f2]
        ids1.append(f1)
        ids2.append(f2)

        # Генерируем на основе данных некоторые числа
        if data1['college']==data2['college']: sameschool=1
        else: sameschool=0
        male1=(data1['gender']=='Male') and 1 or 0
        male2=(data2['gender']=='Male') and 1 or 0

        row=[male1,int(data1['year']),male2,int(data2['year']),sameschool]
        rows.append(row)
    # Вызываем arefriends для каждой пары блоками
    arefriends=[]
    for i in range(0,len(ids1),30):
      j=min(i+20,len(ids1))
      pa=self.arefriends(ids1[i:j],ids2[i:j])
      arefriends+=pa
    return arefriends,rows
```

Этот метод заменяет пол и состояние числами, чтобы набор можно было использовать совместно с LIBSVM. В последнем цикле запрашивается состояние «дружественности» для каждой пары людей. Это делается блоками, так как Facebook ограничивает длину одного запроса.

Создание SVM-модели

Чтобы построить SVM-модель на основе полученных данных, создайте новый сеанс и сгенерируйте набор данных:

```
>>> reload(facebook)
<module 'facebook' from 'facebook.pyc'>
>>> s=facebook.fbsession( )
После регистрации нажмите клавишу Enter:
>>> answers,data=s.makedataset( )
```

К этому набору библиотечные функции применимы непосредственно:

```
>>> param = svm_parameter(kernel_type = RBF)
>>> prob = svm_problem(answers,data)
>>> m=svm_model(prob,param)
>>> m.predict([1,2003,1,2003,1]) # Два человека, окончившие один колледж
в один год
1.0
>>> m.predict([1,2003,1,1996,0]) # Разные колледжи, разные годы окончания
0.0
```

Разумеется, вы получите другие результаты, но, как правило, модель считает, что два человека, окончившие один и тот же колледж или родившиеся в одном городе, могут быть друзьями.

Упражнения

1. *Байесовский классификатор.* Можете ли придумать, как применить построенный в главе 6 байесовский классификатор к набору данных для подбора пар? Что имеет смысл взять в качестве свойств?
2. *Оптимизация разделителя.* Как вы думаете, можно ли выбрать разделитель с помощью рассмотренных в главе 5 методов оптимизации, а не просто по средним? Какую бы вы выбрали целевую функцию?
3. *Выбор наилучших параметров ядра.* Напишите функцию, которая в цикле перебирает различные значения `gamma` и определяет наиболее подходящее для имеющегося набора данных.
4. *Иерархия интересов.* Спроектируйте простую иерархию интересов и структуру данных для ее представления. Измените функцию `matchcount` так, чтобы она использовала эту иерархию для вычисления индекса сочетаемости.
5. *Другие ядра в библиотеке LIBSVM.* Посмотрите в документации по LIBSVM, какие еще ядра поддерживаются. Попробуйте полиномиальное ядро. Улучшилось ли качество прогнозирования?
6. *Другие прогнозы для сайта Facebook.* Ознакомьтесь со всеми полями, которые поддерживает API сайта Facebook. Какие наборы данных можно построить для отдельных людей? Можно ли воспользоваться SVM-моделью для прогнозирования того, назвал ли человек, окончивший определенное учебное заведение, некий фильм своим любимым? Что еще вы могли бы спрогнозировать?

10

Выделение независимых признаков

До сих пор мы в основном занимались обучением классификаторов *с учителем* и только в главе 3 затронули технику *кластеризации без учителя*. В этой главе мы узнаем, как выделять признаки из набора данных, в котором результаты заранее не проставлены. Как и в случае кластеризации, задача состоит не столько в том, чтобы делать прогнозы, сколько в попытке охарактеризовать данные и сообщить о них интересную информацию.

Напомним, что методы кластеризации, рассмотренные в главе 3, относили каждую строку набора данных к некоторой группе или точке в иерархии, – каждый образец попадал ровно в одну группу, которая была представлена средним своих членов. *Выделение признаков* – обобщение этой идеи; мы пытаемся найти новые строки, сочетание которых позволило бы реконструировать строки исходного набора. Мы не относим строки к кластеру, а создаем каждую строку из комбинации признаков.

Классическая задача, иллюстрирующая необходимость нахождения независимых признаков, – это *задача о вечеринке*, суть которой в том, чтобы понять смысл разговора, когда беседуют сразу много людей. У человеческого слуха есть замечательная особенность – способность выделять один голос среди многих, не обращая внимания на остальные. Наш мозг прекрасно умеет вычленять независимые звуки из шума. С помощью алгоритмов, подобных описанному в этой главе, и при наличии в комнате нескольких микрофонов компьютер может сделать то же самое – выделить отдельные звуки из какофонии, не зная заранее, на что они похожи.

Еще одно интересное применение методов выделения признаков – идентификация повторяющихся паттернов употребления слов в массиве документов. Это помогает выделять темы, которые могут независимо

присутствовать в различных сочетаниях. В настоящей главе мы построим систему, которая загружает новости из различных источников и идентифицирует упоминаемые в них основные темы. Может оказаться, что в некоторых новостях упоминается сразу несколько тем, и уж наверняка обнаружится, что одна и та же тема присутствует в нескольких новостях.

Второй пример касается фондового рынка. Предположительно, наблюдаемые на нем результаты обусловлены целым рядом причин. С помощью того же алгоритма мы сумеем отыскать эти причины и вызываемые ими независимые последствия.

Массив новостей

Прежде всего нам потребуется набор новостей, с которыми можно работать. Они должны быть взяты из различных источников, чтобы было проще различить темы, обсуждаемые в разных местах. К счастью, большинство крупных новостных агентств и сайтов поддерживают каналы в формате RSS или Atom либо для всех новостей, либо для отдельных категорий. В предыдущих главах мы уже пользовались библиотекой *Universal Feed Parser* для разбора каналов блогов, она годится и для загрузки новостей. Если вы еще не установили эту библиотеку, скачайте ее с сайта *http://feedparser.org*.

Выбор источников

Существуют тысячи источников новостей – от крупных телеграфных агентств и газет до политических блогов. Вот несколько примеров:

- Агентство «Рейтерс».
- Агентство Associated Press.
- Газета The New York Times.
- Google News.
- Salon.com.
- Fox News.
- Журнал «Форбс».
- Агентство CNN International.

И это лишь малая толика. Выбор источников с различных концов политического спектра и придерживающихся разных литературных стилей позволит лучше проверить алгоритм, так как он сможет выделить существенные признаки и отбросить не относящиеся к делу. При наличии правильно подобранных данных алгоритм сможет также идентифицировать признак, присутствующий только в текстах определенной политической направленности, и приписать его новости в дополнение к тем признакам, которые описывают обсуждаемую тему.

Создайте файл newsfeatures.py, включите в него импорт необходимых библиотек и задайте список источников:

```
import feedparser
import re

feedlist=['http://today.reuters.com/rss/topNews',
          'http://today.reuters.com/rss/domesticNews',
          'http://today.reuters.com/rss/worldNews',
          'http://hosted.ap.org/lineups/TOPHEADS-rss_2.0.xml',
          'http://hosted.ap.org/lineups/USHEADS-rss_2.0.xml',
          'http://hosted.ap.org/lineups/WORLDHEADS-rss_2.0.xml',
          'http://hosted.ap.org/lineups/POLITICSHEADS-rss_2.0.xml',
          'http://www.nytimes.com/services/xml/rss/nyt/HomePage.xml',
          'http://www.nytimes.com/services/xml/rss/nyt/International.xml',
          'http://news.google.com/?output=rss',
          'http://feeds.salon.com/salon/news',
          'http://www.foxnews.com/xmlfeed/rss/0,4313,0,00.rss',
          'http://www.foxnews.com/xmlfeed/rss/0,4313,80,00.rss',
          'http://www.foxnews.com/xmlfeed/rss/0,4313,81,00.rss',
          'http://rss.cnn.com/rss/edition.rss',
          'http://rss.cnn.com/rss/edition_world.rss',
          'http://rss.cnn.com/rss/edition_us.rss']
```

В список включены различные источники, в первую очередь – разделы «Главные новости» (`topNews`), «Мировые новости» (`worldNews`) и «Новости США» (`domesticNews`). Вы можете изменить этот список, как пожелаете, но необходимо, чтобы тематика перекрывалась. Если разные новости не будут иметь между собой ничего общего, алгоритму будет очень трудно извлечь существенные признаки и в результате получится набор ничего не значащих признаков.

Загрузка исходных данных

Для работы алгоритма выделения признаков, как и для алгоритма кластеризации, требуется большая числовая матрица, в которой каждая строка представляет один образец, а каждый столбец – некоторый признак. В нашем случае строками будут новости, а столбцами – слова. Число на пересечении строки и столбца показывает, сколько раз данное слово встречается в данной новости. Так, из показанной ниже матрицы видно, что в новости A слово hurricane (ураган) встречается три раза, в новости B слово democrats (демократы) встречается дважды и т. д.

```
articles = ['A','B','C',...
words = ['hurricane','democrats','world',...
matrix = [[3,0,1,...]
          [1,2,0,...]
          [0,0,2,...]
                 ...]
```

Чтобы перейти от новостного канала к такой матрице, нам понадобятся два метода, аналогичные тем, что уже встречались в предыдущих

главах. Первый метод исключает из новостей все изображения и разметку. Добавьте метод `stripHTML` в файл newsfeatures.py:

```
def stripHTML(h):
  p=''
  s=0
  for c in h:
    if c=='<': s=1
  elif c=='>':
    s=0
    p+=' '
  elif s==0: p+=c
  return p
```

Кроме того, нужно уметь выделять из текста слова, как мы это уже делали ранее. Если вы придумали более изощренный способ выделения слов, чем с помощью простого регулярного выражения, распознающего буквы и цифры, можете повторно воспользоваться им здесь. В противном случае добавьте в файл newsfeatures.py такую функцию:

```
def separatewords(text):
  splitter=re.compile('\\W*')
  return [s.lower( ) for s in splitter.split(text) if len(s)>3]
```

Следующая функция перебирает в цикле все каналы, разбирает их с помощью класса `feedparser`, убирает HTML-теги и выделяет отдельные слова. Она отслеживает как общее число вхождений каждого слова, так и число вхождений в каждую новость.

Добавьте в файл newsfeatures.py следующую функцию:

```
def getarticlewords( ):
  allwords={}
  articlewords=[]
  articletitles=[]
  ec=0
  # Цикл по каналам
  for feed in feedlist:
    f=feedparser.parse(feed)

    # Цикл по новостям
    for e in f.entries:
      # Повторяющиеся новости игнорируются
      if e.title in articletitles: continue

      # Выделяем слова
      txt=e.title.encode('utf8')+stripHTML(e.description.encode('utf8'))
      words=separatewords(txt)
      articlewords.append({})
      articletitles.append(e.title)

      # Увеличиваем счетчики вхождений слова в allwords и в articlewords
      for word in words:
        allwords.setdefault(word,0)
        allwords[word]+=1
```

```
            articlewords[ec].setdefault(word,0)
            articlewords[ec][word]+=1
        ec+=1
    return allwords,articlewords,articletitles
```

В этой функции есть три переменных:

- В словаре `allwords` хранятся счетчики вхождений слов во все новости. Он понадобится нам для того, чтобы определить, какие слова следует считать признаками.
- В списке `articlewords` хранятся счетчики слов для каждой новости.
- В списке `articletitles` хранятся заголовки новостей.

Построение матрицы

Теперь у нас есть словари счетчиков вхождений слов во все статьи, а также в каждую отдельную статью, и предстоит преобразовать их в матрицу, которая была описана выше. На первом шаге мы создаем список слов, которые станут столбцами матрицы. Чтобы уменьшить размер матрицы, можно исключить слова, встречающиеся лишь в одной-двух новостях (скорее всего, для выделения признаков они бесполезны), а также присутствующие практически во всех новостях.

Для начала попробуйте оставить лишь слова, встречающиеся как минимум в четырех новостях, но не более чем в 60% всех новостей. Затем можно воспользоваться трансформацией вложенных списков для создания матрицы, которая будет представлена просто как список списков. Каждый вложенный список создается путем прохода по списку `wordvec` с поиском слова в словаре. Если слово отсутствует, прибавляется 0, иначе создается счетчик вхождений данного слова в текущую новость.

Добавьте функцию `makematrix` в файл newsfeatures.py:

```
def makematrix(allw,articlew):
  wordvec=[]

  # Берем только слова, которые встречаются часто, но не слишком часто
  for w,c in allw.items( ):
    if c>3 and c<len(articlew)*0.6:
      wordvec.append(w)

  # Создаем матрицу слов
  l1=[[(word in f and f[word] or 0) for word in wordvec] for f in articlew]
  return l1,wordvec
```

Запустите интерпретатор Python и импортируйте модуль `newsfeatures`. Можно приступать к разбору каналов и созданию матрицы:

```
$ python
>>> import newsfeatures
>>> allw,artw,artt= newsfeatures.getarticlewords( )
>>> wordmatrix,wordvec= newsfeatures.makematrix(allw,artw)
>>> wordvec[0:10]
['increase', 'under', 'regan', 'rise', 'announced', 'force',
```

```
'street', 'new', 'men', 'reported']
>>> artt[1]
u'Fatah, Hamas men abducted freed: sources'
>>> wordmatrix[1][0:10]
[0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
```

Мы вывели первые 10 элементов вектора слов. Показан заголовок второй новости, а за ним – первые 10 значений из соответствующей ей строки матрицы слов. Как видите, в этой статье один раз встречается слово men (мужчины), а больше никаких слов из первых десяти не встретилось.

Прошлые подходы

В предыдущих главах вы видели различные способы обработки счетчиков слов в текстовых документах. Для сравнения полезно сначала попробовать их и посмотреть, какие получатся результаты, а затем сравнить эти результаты с теми, что даст алгоритм выделения признаков. Если у вас сохранились модули, написанные в предыдущих главах, можете импортировать их и применить к загруженным из каналов данным. Если нет, тоже не страшно – в этом разделе мы продемонстрируем результаты работы вышеупомянутых методов на тестовых данных.

Байесовская классификация

Напомним, что *байесовская классификация* – это метод обучения с учителем. Если вы собираетесь воспользоваться классификатором, написанным в главе 6, то сначала должны сами классифицировать несколько новостей, чтобы его обучить. Затем классификатор сможет распределить остальные новости по заранее заданным категориям. Помимо очевидного недостатка – необходимости начального обучения – у этого подхода есть и еще одно ограничение – разработчик должен сам составить набор категорий. Все встречавшиеся нам ранее классификаторы – деревья решений и машины опорных векторов – при использовании с подобным набором данных страдают тем же недостатком.

Если вы хотите опробовать байесовский классификатор на этом наборе, сначала поместите модуль, написанный в главе 6, в свою рабочую папку. Затем словарь `articlewords` можно использовать в качестве набора признаков для каждой статьи.

Выполните в интерактивном сеансе следующие команды:

```
>>> def wordmatrixfeatures(x):
...     return [wordvec[w] for w in range(len(x)) if x[w]>0]
...
>>> wordmatrixfeatures(wordmatrix[0])
['forces', 'said', 'security', 'attacks', 'iraq', 'its', 'pentagon',...]
>>> import docclass
>>> classifier=docclass.naivebayes(wordmatrixfeatures)
>>> classifier.setdb('newstest.db')
```

```
>>> artt[0]
u'Attacks in Iraq at record high: Pentagon'
>>> # Сообщаем классификатору что это новость на тему 'iraq'
>>> classifier.train(wordmatrix[0],'iraq')
>>> artt[1]
u'Bush signs U.S.-India nuclear deal'
>>> # Сообщаем классификатору что это новость на тему 'india'
>>> classifier.train(wordmatrix[1],'india')
>>> artt[2]
u'Fatah, Hamas men abducted freed: sources'
>>> # А как будет классифицирована эта новость?
>>> classifier.classify(wordmatrix[1])
u'iraq'
```

В используемых примерах много разных тем, но к каждой относится лишь небольшое число новостей. В конечном итоге байесовский классификатор изучит все темы, но поскольку для обучения необходимо хотя бы по несколько примеров на каждую тему, то лучше применять его в случаях, когда категорий меньше, а примеров в каждой категории больше.

Кластеризация

В главе 3 мы видели еще один метод обучения без учителя – кластеризацию.

Там данные были организованы в виде такой же матрицы, что и здесь. Если написанный тогда модуль еще сохранился, импортируйте его в сеансе работы с интерпретатором и примените к только что заданной матрице алгоритм кластеризации:

```
>>> import clusters
>>> clust=clusters.hcluster(wordmatrix)
>>> clusters.drawdendrogram(clust,artt,jpeg='news.jpg')
```

На рис. 10.1 показан возможный результат кластеризации, который сохранен в файле news.jpg.

Как и следовало ожидать, похожие новости сгруппированы вместе. Результат получился даже лучше, чем в примере с блогами из главы 3, поскольку в различных публикациях обсуждаются в точности одни и те же события примерно одинаковым языком. Но пара примеров на рис. 10.1 показывает, что распределение новостей по разным «корзинам» не всегда дает точную картину. Так, статья на тему здоровья «The Nose Knows Better» (Нос знает лучше) оказалась в одной группе с «Suffolk Strangler» (Душитель из Суффолка). Иногда новости, как и люди, не раскладываются по полочкам, и каждую следует считать уникальной.

Если хотите, поверните матрицу на 90° и посмотрите, как кластеризуются слова, встречающиеся в новостях. В нашем примере слова station (станция), solar (солнечная) и astronauts (астронавты) оказались в одной группе.

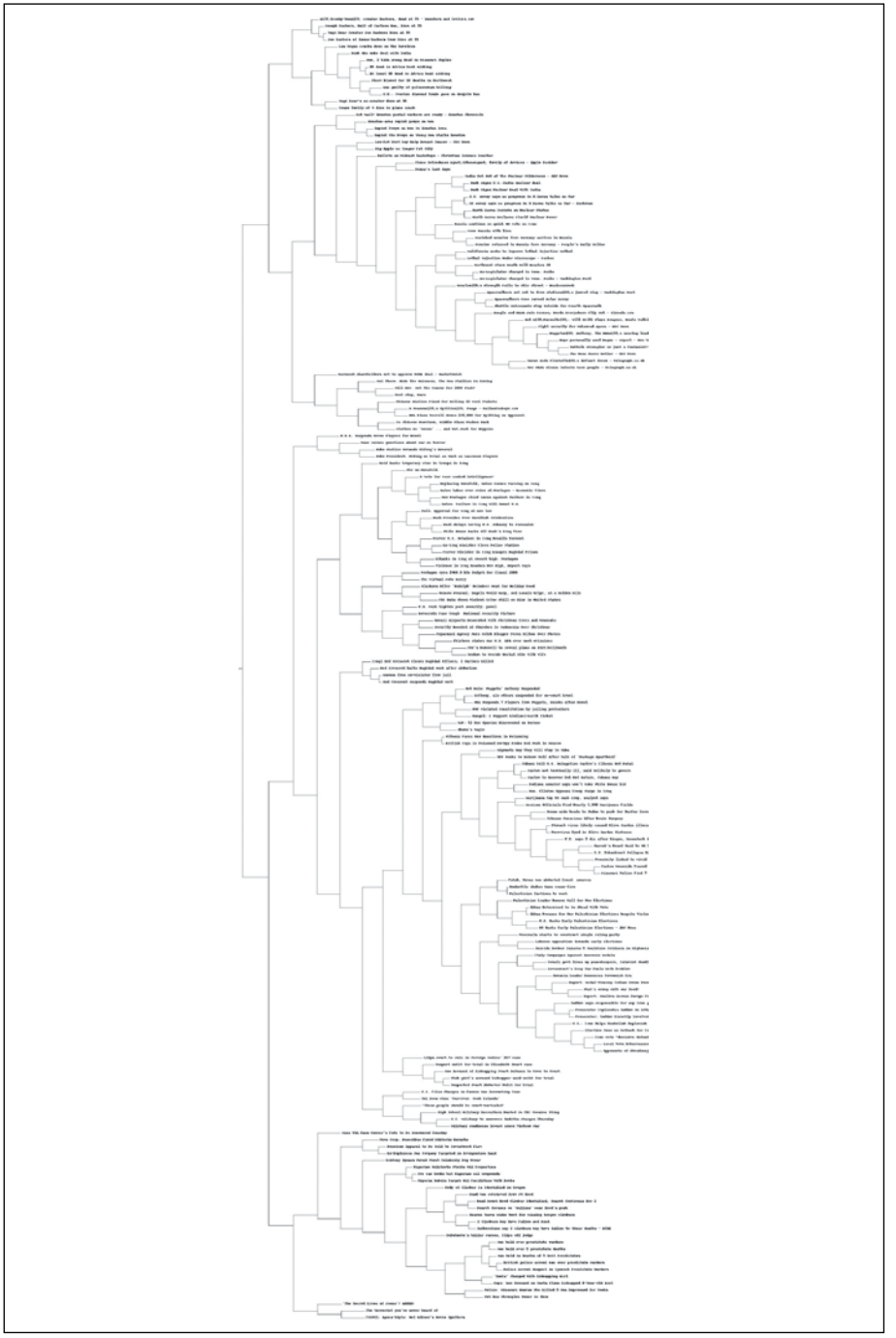


Рис. 10.1. *На этой дендрограмме изображен результат кластеризации новостей*

Неотрицательная матричная факторизация

Техника выделения существенных признаков из данных называется *неотрицательной матричной факторизацией* (Non-negative Matrix Factorization – NMF). Это один из наиболее сложных методов во всей книге, поэтому потребуется чуть больше объяснений и краткое введение в линейную алгебру. Но в этом разделе мы рассмотрим все, что нужно знать.

Краткое введение в математику матриц

Чтобы понять, как работает алгоритм NMF, необходимо кое-что знать об умножении матриц. Если вы знакомы с линейной алгеброй, можете смело пропустить этот раздел.

На рис. 10.2 приведен пример умножения матриц.

$$\underset{\mathbf{A}}{\begin{bmatrix} 1 & 4 \\ 0 & 3 \end{bmatrix}} \times \underset{\mathbf{B}}{\begin{bmatrix} 0 & 3 & 0 \\ 2 & 1 & 4 \end{bmatrix}} = \begin{bmatrix} 1*0+4*2 & 1*3+4*1 & 1*0+4*4 \\ 0*0+3*2 & 0*3+3*1 & 0*0+3*4 \end{bmatrix} = \begin{bmatrix} 8 & 7 & 16 \\ 6 & 3 & 12 \end{bmatrix}$$

Рис. 10.2. *Пример умножения матриц*

Первая из двух перемножаемых матриц (матрица A на рисунке) должна иметь столько столбцов, сколько есть строк во второй матрице (B). В данном случае в матрице A – два столбца, а в матрице B – две строки. Результирующая матрица (C) имеет столько строк, сколько матрица A, и столько столбцов, сколько матрица B.

Значение в каждой клетке (i, j) матрицы C вычисляется путем суммирования произведений чисел в i-й строке матрицы A на соответственные числа в j-м столбце матрицы B. Так, число в левом верхнем углу матрицы C равно сумме произведений чисел в первой строке матрицы A на числа в первом столбце матрицы B. Значения в остальных клетках матрицы C вычисляются аналогично.

Еще одна операция над матрицами называется *транспонированием*. Она меняет строки и столбцы местами. Обычно эта операция обозначается буквой T, как показано на рис. 10.3.

Операции транспонирования и умножения необходимы для реализации алгоритма NMF.

$$\begin{pmatrix} a & d \\ b & e \\ c & f \end{pmatrix}^{T} = \begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix}$$

Рис. 10.3. *Транспонирование матрицы*

Какое отношение все это имеет к матрице новостей

Пока что мы имеем матрицу новостей, в которой хранятся счетчики слов. Наша цель – *факторизовать* ее, то есть найти две меньшие матрицы, при перемножении которых получается исходная. Вот что это за матрицы:

Матрица признаков

В ней каждая строка соответствует признаку, а столбец – слову. Значения в матрице определяют, насколько слово релевантно признаку. Каждый признак должен представлять тему, выявленную при анализе новостей, поэтому можно ожидать, что статья о новом телевизоре будет иметь высокий вес относительно слова television.

Матрица весов

Эта матрица отображает признаки на матрицу новостей. Каждая строка в ней соответствует новости, а каждый столбец – признаку. Значения показывают, в какой степени признак соотносится с новостью.

В матрице признаков, как и в матрице новостей, имеется по одному столбцу для каждого слова. Каждая строка матрицы представляет один признак, поэтому признак – это список весов слов. На рис. 10.4 приведена часть матрицы признаков.

	hurricane	democrats	florida	elections
признак 1	2	0	3	0
признак 2	0	2	0	1
признак 3	0	0	1	1

Рис. 10.4. *Часть матрицы признаков*

Так как каждая строка – это признак, составленный из комбинации слов, то должно быть ясно, что реконструкция матрицы новостей сводится к комбинированию строк из матрицы признаков в разных количествах. Матрица весов, пример которой приведен на рис. 10.5, отображает признаки на новости. В ней имеется по одному столбцу на каждый признак и по одной строке на каждую новость.

На рис. 10.6 показано, как реконструируется матрица новостей путем перемножения матрицы признаков и матрицы весов.

	признак 1	признак 2	признак 3
hurricane in Florida	10	0	0
Democrats sweep elections	0	8	1
Democrats dispute Florida ballots	0	5	6

Рис. 10.5. *Часть матрицы весов*

***Рис. 10.6.** Умножение матрицы весов на матрицу признаков*

Если количество признаков совпадает с количеством новостей, то наилучшим ответом будет сопоставление каждой новости ровно одного признака. Но цель факторизации – свести большой массив наблюдений (в данном случае новостей) к меньшему массиву, который будет содержать общие признаки. В идеале этот меньший набор признаков можно было бы скомбинировать с различными весами и точно воспроизвести исходный набор данных, но на практике это крайне маловероятно, поэтому задача алгоритма – добиться настолько точной реконструкции, насколько возможно.

В названии алгоритма фигурирует слово *«неотрицательная»*, потому что возвращаемые признаки и слова представлены неотрицательными числами. На практике это означает, что значения всех признаков должны быть положительны, и в нашем примере так оно и есть, поскольку ни для какой новости не может быть отрицательного счетчика вхождений слов. Кроме того, признаки не должны «красть» части других признаков – алгоритм NMF устроен так, что никакие слова не будут явно исключены. Хотя это ограничение может воспрепятствовать нахождению оптимальной факторизации, зато результаты легче интерпретировать.

Библиотека NumPy

В стандартном дистрибутиве Python нет функций для операций над матрицами. Хотя их несложно написать самостоятельно, но лучше установить пакет NumPy, который не только предоставляет объект `matrix` и поддерживает все необходимые операции, но и сравним по производительности с коммерческими программами. Загрузить этот пакет можно с сайта *http://numpy.scipy.org*.

Дополнительную информацию об установке пакета NumPy см. в приложении А.

NumPy предоставляет объект `matrix`, конструктору которого передается список списков. Он очень похож на ту матрицу, которую мы создали для представления новостей. Чтобы увидеть пакет NumPy в действии, импортируйте его в интерактивном сеансе и создайте матрицу:

```
>>> from numpy import *
>>> l1=[[1,2,3],[4,5,6]]
>>> l1
[[1, 2, 3], [4, 5, 6]]
>>> m1=matrix(l1)
```

```
>>> m1
matrix([[1, 2, 3],
[4, 5, 6]])
```

Объекты-матрицы поддерживают такие математические операции, как сложение и умножение с помощью стандартных операторов. Для транспонирования матрицы применяется функция `transpose`:

```
>>> m2=matrix([[1,2],[3,4],[5,6]])
>>> m2
matrix([[1, 2],
[3, 4],
[5, 6]])
>>> m1*m2
matrix([[22, 28],
[49, 64]])
```

Функция `shape` возвращает количество строк и столбцов матрицы, что полезно для обхода всех ее элементов в цикле:

```
>>> shape(m1)
(2, 3)
>>> shape(m2)
(3, 2)
```

Наконец, пакет NumPy предоставляет также высокопроизводительный объект-массив `array`, который, как и матрица, может быть многомерным. Матрицу можно легко преобразовать в массив и наоборот. При выполнении умножения массив ведет себя иначе, чем матрица; массивы можно перемножать, только если они имеют в точности одинаковую форму, причем каждый элемент произведения вычисляется перемножением соответственных элементов сомножителей. Например:

```
>>> a1=m1.A
>>> a1
array([[1, 2, 3],
       [4, 5, 6]])
>>> a2=array([[1,2,3],[1,2,3]])
>>> a1*a2
array([[ 1, 4, 9],
       [ 4, 10, 18]])
```

Высокая производительность NumPy критически важна для работы алгоритма, который, как вы скоро увидите, выполняет много операций над матрицами.

Алгоритм

Описываемый алгоритм факторизации матриц впервые был опубликован в конце 1990-х годов и является одним из самых современных алгоритмов, рассматриваемых в настоящей книге. Было показано, что он очень хорошо работает для определенного класса задач, в частности для автоматического распознавания черт лица по фотографиям.

Алгоритм пытается максимально близко реконструировать матрицу новостей путем вычисления оптимальных матриц признаков и весов. В данном случае было бы полезно иметь способ измерения степени близости результата. Функция `difcost` перебирает все значения в двух матрицах одинакового размера и вычисляет квадраты разностей между ними.

Создайте файл nmf.py и включите в него функцию `difcost`:

```
from numpy import *

def difcost(a,b):
  dif=0
  # Цикл по строкам и столбцам матрицы
  for i in range(shape(a)[0]):
    for j in range(shape(a)[1]):
      # Суммируем квадраты разностей
      dif+=pow(a[i,j]-b[i,j],2)
  return dif
```

Теперь нужно придумать, как постепенно изменять матрицы, чтобы эта целевая функция уменьшалась. Если вы прочли главу 5, то, несомненно, заметили, что это действительно целевая функция, поэтому для поиска хорошего решения можно применить алгоритм имитации отжига или генетический алгоритм. Однако в данном случае более эффективно использование *мультипликативных правил обновления*.

Вывод этих правил выходит за рамки книги, но, если вам интересно, ознакомьтесь с оригинальной статьей по адресу *http://hebb.mit.edu/people/seung/papers/nmfconverge.pdf*.

Согласно этим правилам генерируются четыре новых матрицы. В описании исходная матрица новостей называется *матрицей данных*:

`hn`

Произведение транспонированной матрицы весов и матрицы данных.

`hd`

Произведение транспонированной матрицы весов, самой матрицы весов и матрицы признаков.

`wn`

Произведение матрицы данных и транспонированной матрицы признаков.

`wd`

Произведение матрицы весов, матрицы признаков и транспонированной матрицы признаков.

Для обновления матриц признаков и весов все эти матрицы преобразуются в массивы. Каждый элемент матрицы признаков умножается на соответственный элемент `hn` и делится на соответственный элемент `hd`. Аналогично, каждый элемент матрицы весов умножается на соответственный элемент `wn` и делится на элемент `wd`.

Функция `factorize` выполняет все эти вычисления. Добавьте ее в файл nmf.py:

```
def factorize(v,pc=10,iter=50):
  ic=shape(v)[0]
  fc=shape(v)[1]

  # Матрицы весов и признаков инициализируются случайными значениями
  w=matrix([[random.random( ) for j in range(pc)] for i in range(ic)])
  h=matrix([[random.random( ) for i in range(fc)] for i in range(pc)])

  # Выполняем операцию не более iter раз
  for i in range(iter):
    wh=w*h

    # Вычисляем текущую разность
    cost=difcost(v,wh)

    if i%10==0: print cost

    # Выходим из цикла, если матрица уже факторизована
    if cost==0: break

    # Обновляем матрицу признаков
    hn=(transpose(w)*v)
    hd=(transpose(w)*w*h)

    h=matrix(array(h)*array(hn)/array(hd))

    # Обновляем матрицу весов
    wn=(v*transpose(h))
    wd=(w*h*transpose(h))

    w=matrix(array(w)*array(wn)/array(wd))

  return w,h
```

Функции, которая факторизует матрицу, необходимо указать, сколько признаков вы хотите обнаружить. Иногда количество признаков заранее известно (два голоса в записи или пять главных новостей за день). В других случаях о количестве признаков ничего нельзя сказать. Не существует общего способа автоматически определить правильное число признаков, но путем экспериментов можно подобрать подходящий диапазон.

Попробуйте выполнить этот алгоритм для матрицы размерностью `m1*m2` в текущем сеансе и посмотрите, найдет ли алгоритм решение, похожее на исходную матрицу:

```
>>> import nmf
>>> w,h= nmf.factorize(m1*m2,pc=3,iter=100)
7632.94395925
0.0364091326734
...
```

```
1.12810164789e-017
6.8747907867e-020
>>> w*h
matrix([[ 22., 28.],
[ 49., 64.]])
>>> m1*m2
matrix([[22, 28],
[49, 64]])
```

Алгоритм сумел подобрать матрицы весов и признаков так, что при их перемножении получается в точности исходная матрица. Применим его к матрице новостей и посмотрим, как он справится с задачей выделения существенных признаков (на это может уйти заметное время):

```
>>> v=matrix(wordmatrix)
>>> weights,feat=nmf.factorize(v,pc=20,iter=50)
1712024.47944
2478.13274637
2265.75996871
2229.07352131
2211.42204622
```

Теперь в переменной `feat` хранятся признаки, а в переменной `weights` — значения, характеризующие релевантность признаков новостям. Простой взгляд на матрицу ничего не дает, а хотелось бы как-то визуализировать и интерпретировать результаты.

Вывод результатов

Довольно сложно решить, как именно визуализировать результаты. Каждому свойству в матрице признаков соответствует набор весов, показывающих, как сильно связано с этим признаком каждое слово. Поэтому можно попытаться вывести пять-десять слов для каждого признака, чтобы узнать, какие слова для него наиболее характерны. Аналогично, столбец в матрице весов говорит о том, в какой мере этот признак соотносится с каждой новостью, поэтому было бы также интересно показать верхние три новости для данного признака вместе с индексом релевантности.

Добавьте в файл newsfeatures.py функцию `showfeatures`:

```
from numpy import *
def showfeatures(w,h,titles,wordvec,out='features.txt'):
  outfile=file(out,'w')
  pc,wc=shape(h)
  toppatterns=[[] for i in range(len(titles))]
  patternnames=[]

  # Цикл по всем признакам
  for i in range(pc):
    slist=[]
    # Создаем список слов и их весов
    for j in range(wc):
```

```
      slist.append((h[i,j],wordvec[j]))
    # Сортируем список слов в порядке убывания
    slist.sort( )
    slist.reverse( )

    # Печатаем первые шесть элементов
    n=[s[1] for s in slist[0:6]]
    outfile.write(str(n)+'\n')
    patternnames.append(n)

    # Создаем список слов для этого признака
    flist=[]
    for j in range(len(titles)):
      # Добавляем новость вместе с ее весом
      flist.append((w[j,i],titles[j]))
      toppatterns[j].append((w[j,i],i,titles[j]))

    # Сортируем список в порядке убывания
    flist.sort( )
    flist.reverse( )

    # Выводим первые три новости
    for f in flist[0:3]:
      outfile.write(str(f)+'\n')
    outfile.write('\n')

  outfile.close( )
  # Возвращаем списки слов и новостей для дальнейшего использования
  return toppatterns,patternnames
```

Эта функция в цикле перебирает все признаки и создает список всех слов и их весов из вектора слов. Затем этот список сортируется так, чтобы слова с наибольшим весом оказались в начале, и первые шесть слов печатаются. Это должно дать представление о том, какая тема соответствует данному признаку. Функция возвращает список наиболее четких паттернов и их названий, чтобы не вычислять их каждый раз заново при использовании в функции `showarticles`, показанной ниже.

После вывода признака функция перебирает заголовки всех новостей и сортирует их в соответствии со значениями в матрице весов для данной новости и признака. Далее печатаются три новости, наиболее релевантные текущему признаку, вместе со значениями из матрицы весов. Вы увидите, что иногда признак релевантен нескольким новостям, а иногда – только одной.

Вызовите эту функцию, чтобы посмотреть, какие признаки найдены:

```
>>> reload(newsfeatures)
<module 'newsfeatures' from 'newsfeatures.py'>
>>> topp,pn= newsfeatures.showfeatures(weights,feat,artt,wordvec)
```

Поскольку список результатов длинный, программа сохраняет его в текстовом файле. При вызове функции мы попросили выделить

20 признаков. Очевидно, что в сотнях новостей тем будет гораздо больше, но хочется надеяться, что самые важные мы все-таки выявим. Вот пример:

```
[u'palestinian', u'elections', u'abbas', u'fatah', u'monday', u'new']
(14.189453058041485, u'US Backs Early Palestinian Elections - ABC News')
(12.748863898714507, u'Abbas Presses for New Palestinian Elections Despite
Violence')
(11.286669969240645, u'Abbas Determined to Go Ahead With Vote')
```

Видно, что этому признаку релевантны слова, относящиеся к выборам в Палестине, и имеется целая подборка статей на эту тему. Поскольку результат определяется как заголовком, так и текстом новости, то вторая и третья новости оказались ассоциированы с этим свойством, хотя в их заголовках нет ни одного слова из списка. Кроме того, поскольку наиболее важны слова, встречающихся во многих новостях, то слова palestinian (палестинский) и elections (выборы) оказались на первых местах.

Для некоторых признаков такой четкой подборки новостей нет, но результаты все равно интересны. Взгляните:

```
[u'cancer', u'fat', u'low', u'breast', u'news', u'diet']
(29.808285029040864, u'Low-Fat Diet May Help Breast Cancer')
(2.3737882572527238, u'Big Apple no longer Fat City')
(2.3430261571622881, u'The Nose Knows Better')
```

Очевидно, этот признак наиболее тесно связан с новостью о раке груди (breast cancer). Однако и новости с более слабой ассоциацией тоже касаются здоровья. По-видимому, в них встречаются какие-то слова, общие с первой новостью.

Вывод новости

Другой способ визуализировать данные – показать каждую новость и три наиболее релевантных ей признака. Это даст возможность понять, обсуждаются ли в новости несколько тем одинаковой важности или она посвящена главным образом одной теме.

Добавьте в файл newsfeatures.py функцию `showarticles`:

```
def showarticles(titles,toppatterns,patternnames,out='articles.txt'):
  outfile=file(out,'w')

  # Цикл по всем новостям
  for j in range(len(titles)):
    outfile.write(titles[j].encode('utf8')+'\n')

    # Получить наиболее релевантные этой новости признаки
    # и отсортировать их в порядке убывания
    toppatterns[j].sort( )
    toppatterns[j].reverse( )
```

```
    # Напечатать три верхних признака
    for i in range(3):
       outfile.write(str(toppatterns[j][i][0])+' '+
                     str(patternnames[toppatterns[j][i][1]])+'\n')
    outfile.write('\n')

  outfile.close( )
```

Так как релевантные каждой статье признаки были вычислены функцией `showfeatures`, то здесь нам остается только обойти все новости, напечатать их названия и для каждой вывести три основных признака.

Для тестирования перезагрузите модуль newsfeatures.py и вызовите функцию `showfeatures`:

```
>>> reload(newsfeatures)
<module 'newsfeatures' from 'newsfeatures.py'>
>>> newsfeatures.showarticles(artt,topp,pn)
```

В результате будет создан файл articles.txt, который содержит заголовки новостей и наиболее характерные для них паттерны. Вот пример новости, в которой обсуждаются две одинаково важные темы:

```
Attacks in Iraq at record high: Pentagon
5.4890098003 [u'monday', u'said', u'oil', u'iraq', u'attacks', u'two']
5.33447632219 [u'gates', u'iraq', u'pentagon', u'washington', u'over',
u'report']
0.618495842404 [u'its', u'iraqi', u'baghdad', u'red', u'crescent', u'monday']
```

Очевидно, что оба признака связаны с Ираком, но относятся они не только к этой новости, поскольку в ее тексте не встречаются слова oil (нефть) и gates (Гейтс). Поскольку алгоритм выделяет признаки, которые могут сочетаться друг с другом, но необязательно относятся к единственной новости, то общее число признаков оказывается меньше числа новостей.

А вот пример новости с высокорелевантным признаком, который вряд ли ассоциируется еще с чем-нибудь:

```
Yogi Bear Creator Joe Barbera Dies at 95
11.8474089735 [u'barbera', u'team', u'creator', u'hanna', u'dies', u'bear']
2.21373704749 [u'monday', u'said', u'oil', u'iraq', u'attacks', u'two']
0.421760994361 [u'man', u'was', u'year', u'his', u'old', u'kidnapping']
```

Так как количество признаков невелико, вероятно появление *новостей-сирот*, которые больше ни на что не похожи и потому не удостоились собственного признака. Вот пример:

```
U.S. Files Charges in Fannie Mae Accounting Case
0.856087848533 [u'man', u'was', u'year', u'his', u'old', u'kidnapping']
0.784659717694 [u'climbers', u'hood', u'have', u'their', u'may', u'deaths']
0.562439763693 [u'will', u'smith', u'news', u'office', u'box', u'all']
```

Как видите, наиболее релевантные этой новости признаки не очень-то связаны с ней и кажутся случайными. Но их веса очень малы, поэтому вы сразу понимаете, что к данной новости они имеют слабое отношение.

Использование данных о фондовом рынке

Алгоритм NMF может работать не только с дискретными данными, например счетчиками слов, но и с числовыми. В этом разделе мы покажем, как применить его к *объему торгов* на рынке акций США, воспользовавшись данными с сайта Yahoo! Finance. Возможно, что в результате анализа нам удастся обнаружить паттерны, определяющие важность торговых сессий, или понять, от каких факторов зависит объем торгов различными акциями.

Финансовые рынки считаются идеальным примером коллективного разума, поскольку для них характерно большое количество участников, которые ведут себя независимо, пользуются различной информацией, проявляют различные предпочтения и вырабатывают небольшое число результатов, например цену и объем. Как выяснилось, отдельному человеку очень трудно сравняться в точности предсказания будущих цен с коллективным разумом. Существует немало научных работ на тему о том, почему группам людей дается лучше устанавливать цены на финансовом рынке, чем любому индивидууму.

Что такое объем торгов

Объем торгов по конкретной акции – это количество сделок купли/продажи по ней за некоторый период времени, обычно за день. На рис. 10.7 показан график торгов акциями компании Yahoo!, за которой закреплен символ *YHOO*. Кривая в верхней части – это *цена закрытия*, то есть цена последней сделки за день. На столбчатой диаграмме внизу показан объем торгов.

Бросается в глаза, что объем больше в те дни, когда цена акций сильнее всего изменялась. Такое часто бывает, когда компания делает какое-то заявление или публикует результаты финансовой деятельности. Пики могут возникать также из-за появления каких-то новостей о компании или индустрии в целом. В отсутствие внешних событий объем торгов обычно, хотя и не всегда, остается постоянным.

В этом разделе мы рассмотрим временные ряды объема торгов акциями нескольких компаний. Это даст нам возможность поискать паттерны, затрагивающие сразу несколько акций, или события, оказавшиеся настолько важными, что стали отдельными признаками. Мы используем объем торгов, а не цену закрытия, поскольку алгоритм NMF пытается искать положительные признаки, которые можно складывать; цены же часто реагируют на внешние события снижением, а NMF не умеет выделять отрицательные признаки. Напротив, моделировать объем торгов проще, так как имеется базовый уровень, который может лишь увеличиваться в ответ на внешние сигналы, поэтому пригоден для построения положительных матриц.

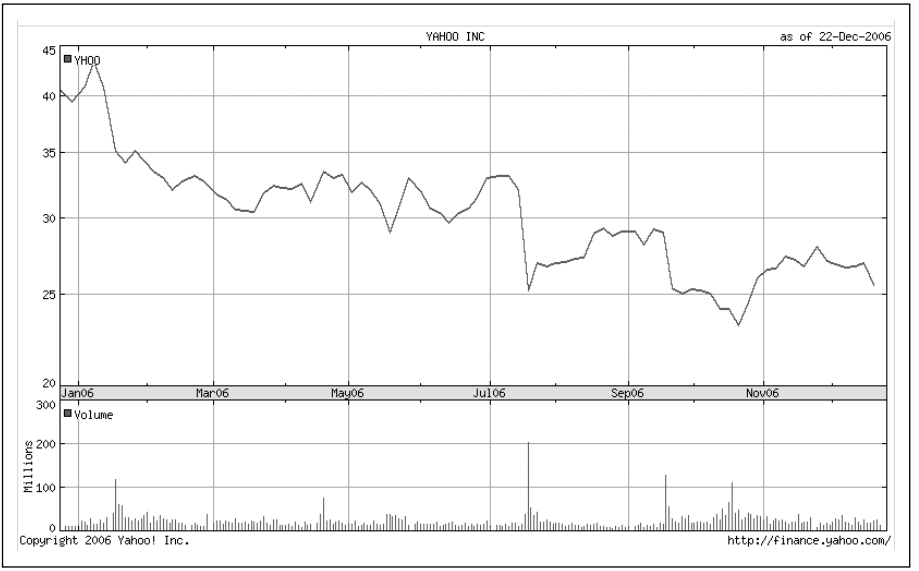


Рис. 10.7. *График изменения цены и объема торгов акциями одной компании*

Загрузка данных с сайта Yahoo! Finance

Yahoo! Finance – отличный ресурс для получения разного рода финансовых данных, в том числе цен на акции, опционов, курсов валют и процентных ставок по облигациям. Кроме того, он позволяет загружать исторические данные об объемах торгов и ценах в удобном для обработки формате CSV. Обратившись к URL вида *http://ichart.finance.yahoo.com/table.csv?s=YHOO&d=11&e=26&f=2006&g=d&a=3&b=12&c=1996&ignore=.csv*, вы получите для указанной акции список данных в формате CSV, разбитый по дням. Первые строки списка выглядят примерно так:

```
Date,Open,High,Low,Close,Volume,Adj. Close*
22-Dec-06,25.67,25.88,25.45,25.55,14666100,25.55
21-Dec-06,25.71,25.75,25.13,25.48,27050600,25.48
20-Dec-06,26.24,26.31,25.54,25.59,24905600,25.59
19-Dec-06,26.05,26.50,25.91,26.41,18973800,26.41
18-Dec-06,26.89,26.97,26.07,26.30,19431200,26.30
```

В каждой строке представлены дата, цена открытия и закрытия, наибольшая и наименьшая цена, объем торгов и скорректированная цена закрытия. В скорректированной цене учтено, что акции могут дробиться или по ним могут выплачиваться дивиденды. Она позволяет вычислить, сколько у вас было бы денег, если бы вы владели данной акцией в промежутке между двумя датами.

Для рассматриваемого примера мы загрузим данные об объеме торгов по нескольким акциям. Создайте новый файл stockvolume.py и включите в него приведенный ниже код, который загружает файлы в формате CSV для заданного списка символов и сохраняет их в словаре. Кроме того, запоминается, по какому символу количество дней в истории минимально. Это число мы затем используем в качестве вертикальной размерности матрицы наблюдений:

```
import nmf
import urllib2
from numpy import *

tickers=['YHOO','AVP','BIIB','BP','CL','CVX',
         'DNA','EXPE','GOOG','PG','XOM','AMGN']

shortest=300
prices={}
dates=None

for t in tickers:
  # Открыть URL
  rows=urllib2.urlopen('http://ichart.finance.yahoo.com/table.csv?'+\
                       's=%s&d=11&e=26&f=2006&g=d&a=3&b=12&c=1996'%t +\
                       '&ignore=.csv').readlines( )

  # Выделить из каждой строки поле объема торгов
  prices[t]=[float(r.split(',')[5]) for r in rows[1:] if r.strip( )!='']
  if len(prices[t])<shortest: shortest=len(prices[t])

  if not dates:
    dates=[r.split(',')[0] for r in rows[1:] if r.strip( )!='']
```

Здесь для каждого символа акции создается URL и из него загружаются данные. Затем каждая строка разбивается по запятым и из получившегося списка берется пятый элемент – объем торгов в виде числа с плавающей точкой.

Подготовка матрицы

На следующем шаге мы должны преобразовать эти данные в матрицу наблюдений, которую можно будет передать алгоритму NMF. Для этого достаточно создать список, каждый элемент которого является списком, содержащим данные об объеме торгов всеми акциями за один день. Рассмотрим пример:

```
[[4453500.0, 842400.0, 1396100.0, 1883100.0, 1281900.0,...]
 [5000100.0, 1486900.0, 3317500.0, 2966700.0, 1941200.0,...
 [5222500.0, 1383100.0, 3421500.0, 2885300.0, 1994900.0,...
 [6028700.0, 1477000.0, 8178200.0, 2919600.0, 2061000.0,...]
 ...]
```

Согласно этому списку в последний день торговалось 4453 500 акций AMGN, 842 000 акций AVP и т. д. В предыдущий день объем составил соответственно 5000 100 и 1486 900 акций. Сравните с примером, относящимся к новостям; подставьте вместо новостей дни, вместо слов – символы акций, а вместо счетчиков слов – объемы торгов.

Матрицу легко построить с помощью трансформации списков. Внутренний цикл перебирает список символов, а внешний – список наблюдений (дней). Добавьте следующий код в файл stockvolume.py:

```
l1=[[prices[tickers[i]][j]
     for i in range(len(tickers))]
   for j in range(shortest)]
```

Прогон алгоритма NMF

Осталось лишь вызвать функцию `factorize` из модуля NMF. Придется задать количество искомых признаков; для небольшого набора акций будет достаточно четырех-пяти.

Добавьте в файл stockvolume.py такой код:

```
w,h= nmf.factorize(matrix(l1),pc=5)

print h
print w
```

Теперь запустите из командной строки следующую команду:

```
$ python stockvolume.py
```

Матрицы, которые она выведет, представляют веса и признаки. Каждая строка в матрице признаков соответствует одному признаку, который представляет собой совокупность объемов торгов по акциям. В сочетании с другими признаками его можно использовать для реконструкции данных об объеме торгов за текущий день. Каждая строка в матрице весов представляет один день, а значения показывают, в какой мере каждый из выделенных признаков применим к этому дню.

Вывод результатов

Понятно, что интерпретировать сами матрицы трудно, поэтому нужно найти более удобный способ вывода признаков. Хотелось бы видеть, какой вклад вносит объем торгов по каждой акции в каждый признак, а также даты, наиболее тесно связанные с признаками.

Добавьте в файл stockvolume.py следующий код:

```
# Цикл по всем признакам
for i in range(shape(h)[0]):
  print "Признак %d" %i

  # Получить верхние 10 акций для этого признака
  ol=[(h[i,j],tickers[j]) for j in range(shape(h)[1])]
  ol.sort( )
  ol.reverse( )
```

```
  for j in range(12):
    print ol[j]
  print

  # Показать верхние 10 дат для этого признака
  porder=[(w[d,i],d) for d in range(300)]
  porder.sort( )
  porder.reverse( )
  print [(p[0],dates[p[1]]) for p in porder[0:3]]
  print
```

Поскольку текста будет много, имеет смысл перенаправить вывод в файл. Введите такую команду:

```
$ python stockvolume.py > stockfeatures.txt
```

Теперь в файле stockfeatures.txt находится список признаков с указанием того, каким акциям они наиболее релевантны и в какие дни проявлялись наиболее отчетливо. Вот пример, выбранный из файла потому, что он имеет очень высокий вес для данной акции и данной даты:

```
Признак 4
(74524113.213559602, 'YHOO')
(6165711.6749675209, 'GOOG')
(5539688.0538382991, 'XOM')
(2537144.3952459987, 'CVX')
(1283794.0604679288, 'PG')
(1160743.3352889531, 'BP')
(1040776.8531969623, 'AVP')
(811575.28223116993, 'BIIB')
(679243.76923785623, 'DNA')
(377356.4897763988,  'CL')
(353682.37800343882, 'EXPE')
(0.31345784102699459,'AMGN')

[(7.950090052903934,  '19-Jul-06'),
 (4.7278341805021329, '19-Sep-06'),
 (4.6049947721971245, '18-Jan-06')]
```

Как видно, этот признак относится почти исключительно к YHOO и наиболее отчетливо проявился 19 июля 2006 года. Именно в тот день наблюдался большой всплеск объема торгов акциями компании Yahoo!, которая опубликовала отчет о прибылях.

Вот еще один признак, распределенный более равномерно между двумя компаниями:

```
Признак 2
(46151801.813632453, 'GOOG')
(24298994.720555616, 'YHOO')
(10606419.91092159,  'PG')
(7711296.6887903402, 'CVX')
(4711899.0067871698, 'BIIB')
(4423180.7694432881, 'XOM')
(3430492.5096612777, 'DNA')
(2882726.8877627672, 'EXPE')
```

```
(2232928.7181202639, 'CL')
(2043732.4392455407, 'AVP')
(1934010.2697886101, 'BP')
(1801256.8664912341, 'AMGN')

[(2.9757765047938824, '20-Jan-06'),
 (2.8627791325829448, '28-Feb-06'),
 (2.356157903021133,  '31-Mar-06'),
```

Он показывает сильные всплески в объеме торгов акциями Google, которые в первых трех случаях были обусловлены публикацией новостей. В самый насыщенный день, 20 января, Google объявила, что она не собирается раскрывать правительству информацию о своей поисковой машине. Но интересно, что признак, обусловленный событием, которое повлияло на объем торгов акциями Google, сказался и на объеме торгов акциями Yahoo!, хотя к этой компании событие не имеет никакого отношения. Второй всплеск пришелся на день, когда финансовый директор Google объявил о замедлении темпов роста; на графике видно, что объем торгов по Yahoo! в тот день также возрос, возможно, потому, что люди прикинули, как эта информация может сказаться на положении Yahoo!.

Важно понимать разницу между представленной здесь картиной и простым отысканием корреляции между объемами торгов. Два выделенных признака показывают, что в некоторые дни торги акциями компаний Google и Yahoo! протекают схоже, а в другие ведут себя совершенно по-разному. При поиске корреляции эти наблюдения были бы усреднены и тот факт, что в определенные дни на объем торгов повлияли заявления Yahoo!, остался бы незамеченным.

Этот пример с небольшой выборкой акций иллюстрирует простой факт, но если взять более представительную выборку и задать большее количество искомых признаков, то удается выявить более сложные взаимодействия.

Упражнения

1. *Разнородные источники новостей.* В примере из этой главы мы брали чисто новостные источники. Попробуйте добавить еще известные политические блоки (найти их поможет сайт *http://technorati.com*). Как это отразится на результатах? Существуют ли признаки, характерные главным образом для политических комментариев? Удается ли сгруппировать новости с соответствующими им комментариями?
2. *Кластеризация методом K-средних.* Для матрицы новостей мы применили иерархическую кластеризацию. А что если воспользоваться кластеризацией методом *K*-средних? Сравните число признаков, необходимых в обоих случаях для выделения всех тем.

3. *Оптимизация и факторизация.* Можно ли воспользоваться программой оптимизации, написанной в главе 5, для факторизации матрицы? Будет ли работа проходить быстрее или медленнее? Сравните результаты.
4. *Критерий останова*. Алгоритм NMF, описанный в этой главе, останавливается, когда целевая функция принимает значение 0 или после выполнения заданного числа итераций. Иногда уже найденное хорошее, пусть даже не идеальное, решение далее не удается существенно улучшить. Модифицируйте код так, чтобы алгоритм прекращал работу, когда целевая функция уменьшается на каждой итерации не более чем на один процент.
5. *Альтернативные способы вывода.* Описанные в этой главе функции вывода результатов достаточно просты и показывают существенные признаки, но при этом теряется значительная часть контекста. Можете ли вы предложить другие варианты вывода результатов? Попробуйте написать функцию, которая выводит тексты новостей, выделяя ключевые слова, вошедшие в состав каждого признака. Или нарисуйте диаграмму объема торгов, на которой отчетливо видны существенные даты.

11

Эволюционирующий разум

В этой книге было представлено много разных задач, и в каждом случае применялся алгоритм, наиболее подходящий для решения конкретной задачи. В некоторых примерах приходилось настраивать параметры или пользоваться оптимизацией для отыскания наилучшего набора параметров. В этой главе мы рассмотрим иной подход. Вместо того чтобы выбирать алгоритм решения задачи, мы напишем программу, которая попытается автоматически построить наилучшую программу для решения данной задачи. Иными словами, мы будем создавать алгоритм создания алгоритмов.

Для этого мы применим *генетическое программирование*. Для этой главы я выбрал тему, по которой до сих пор ведутся активные исследования. Эта глава отличается от предыдущих тем, что в ней не будет ни открытых API, ни публично доступных наборов данных, а программы, модифицирующие себя на основе взаимодействия со многими людьми, – это интересный и совершенно иной вид коллективного разума. Генетическое программирование – весьма обширная тема, на которую написано много книг. Я смогу дать лишь краткое введение, но надеюсь возбудить в вас достаточный интерес; возможно, вы захотите заняться самостоятельными исследованиями и экспериментами в этой области.

В этой главе рассматриваются две задачи: реконструкция математической функции по имеющемуся набору данных и автоматическое обучение компьютера простой игре (искусственный интеллект). Это лишь

мизерная доля возможностей генетического программирования – единственным ограничением на типы решаемых задач служит наличие вычислительных ресурсов.

Что такое генетическое программирование

Генетическое программирование – это методика машинного обучения, прототипом которой является биологическая эволюция. В общем случае мы начинаем с большого набора программ (именуемых *популяцией*), сгенерированных случайным образом или написанных вручную, о которых известно, что это достаточно хорошие решения. Затем эти программы конкурируют между собой в попытке решить некоторую поставленную пользователем задачу. Это может быть игра, в которой программы соревнуются между собой напрямую, или специальный тест, призванный определить, какая программа лучше. По завершении состязания составляется ранжированный список программ – от наилучшей к наихудшей.

Затем – и вот тут-то вступает в дело эволюция – лучшие программы копируются и модифицируются одним из двух способов. Самый простой называется *мутацией*; в этом случае некоторые части программы случайным образом и очень незначительно изменяются в надежде, что от этого решение станет лучше. Другой способ называется *скрещиванием* (или *кроссовером*) – часть одной из отобранных программ перемещается в другую. В результате процедуры копирования и модификации создается много новых программ, которые основаны на наилучших особях предыдущей популяции, но не совпадают с ними.

На каждом этапе качество программ вычисляется с помощью *функции выживаемости* (fitness function). Так как размер популяции не изменяется, программы, оказавшиеся плохими, удаляются из популяции, освобождая место для новых. Создается новая популяция, которая называется «следующим поколением», и весь процесс повторяется. Поскольку сохраняются и изменяются самые лучшие программы, то есть надежда, что с каждым поколением они будут совершенствоваться, как дети, которые могут превзойти своих родителей.

Новые поколения создаются до тех пор, пока не будет выполнено условие завершения, которое в зависимости от задачи может формулироваться одним из следующих способов:

- Найдено идеальное решение.
- Найдено достаточно хорошее решение.
- Решение не удается улучшить на протяжении нескольких поколений.
- Количество поколений достигло заданного предела.

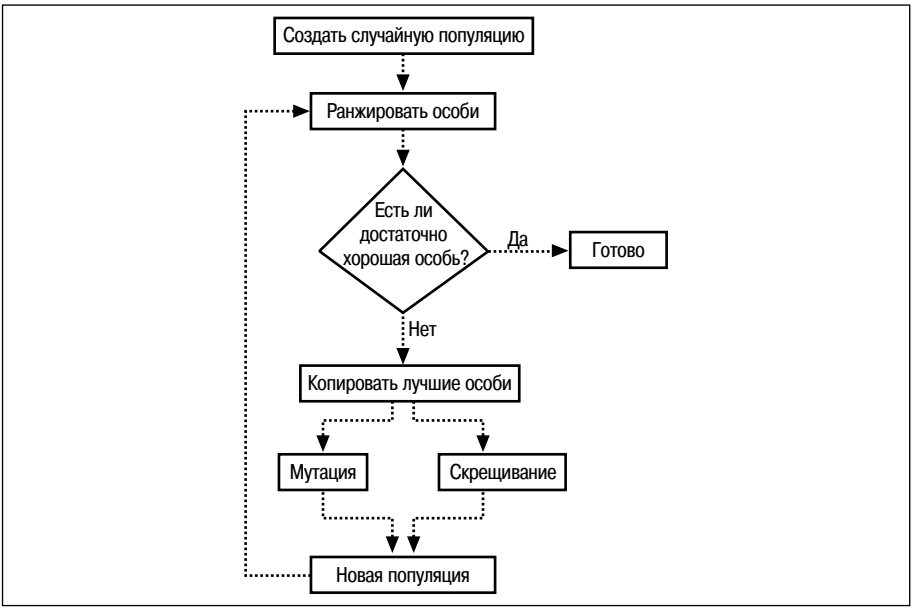


Рис. 11.1. *Блок-схема генетического программирования*

Для некоторых задач, например определения математической функции, отображающей один набор значений на другой, можно найти идеальное решение. Для других, например когда речь идет о настольной игре, найденное решение может быть не идеальным, поскольку его качество зависит от стратегии противника.

Блок-схема, изображенная на рис. 11.1, дает представление о процессе генетического программирования.

Генетическое программирование и генетические алгоритмы

В главе 5 вы ознакомились с родственным понятием *генетического алгоритма*. Генетический алгоритм – это метод оптимизации, в основе которого лежит идея естественного отбора как средства достижения наилучшего результата. При любом способе оптимизации изначально имеется некоторая метрика или алгоритм и мы просто пытаемся подобрать для него наилучшие параметры.

Как и в случае оптимизации, для генетического программирования необходим способ измерения качества решения. Но, в отличие от оптимизации, решением является не просто набор параметров заданного алгоритма. Путем естественного отбора автоматически проектируется сам алгоритм со всеми своими параметрами.

Успехи генетического программирования

Генетическое программирование существует еще с 1980-х годов, но для него нужны гигантские вычислительные ресурсы, а с теми, что были в наличии в то время, можно было решать лишь самые простые задачи. Но по мере того как компьютеры становились все быстрее, генетическое программирование начали применять и к более сложным проблемам. Используя основанные на нем методы, удалось повторить или усовершенствовать многие ранее запатентованные изобретения. И некоторые последние изобретения, достойные патентования, были сделаны с помощью компьютеров.

Методы генетического программирования применялись в проектировании антенн для НАСА, при создании фотонных кристаллов, в оптике, квантовых компьютерах и в других научных областях. Использовались они и при разработке игровых программ, в том числе для игры в шахматы и в нарды. В 1998 году исследователи из университета Карнеги Меллон заявили команду роботов, созданную на основе только генетического программирования, на соревнования по футболу среди роботов и заняли одно из средних мест.

Программы как деревья

Для создания программ, которые можно тестировать, подвергать мутации и скрещиванию, необходимо как-то представлять их в виде кода на языке Python и запускать. Представление должно допускать простую модификацию и, что более существенно, быть настоящей исполняемой программой. Поэтому просто генерировать случайные строки и пытаться трактовать их как Python-программу не получится. Было предложено несколько способов представления программ для генетического программирования, но самым распространенным является древовидное представление.

Программы на большинстве языков – неважно, компилируемых или интерпретируемых – сначала преобразуются в *дерево разбора*, очень напоминающее то, с чем мы будем работать далее. (Язык программирования Lisp и его диалекты – это, по существу, способ непосредственного представления программы в виде дерева разбора.) На рис. 11.2 представлен пример дерева разбора.

Каждый узел представляет либо операцию над дочерними узлами либо является листовым, например параметром или константой. Так, кружочком представлена операция суммирования двух ветвей, в данном случае значений переменной *Y* и константы 5. Вычисленная сумма передается вышестоящему узлу, который применяет собственную операцию

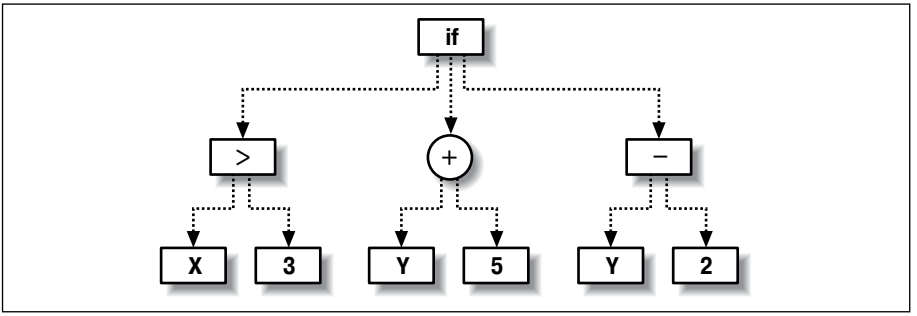


***Рис. 11.2.** Программа, представленная в виде дерева*

к своим потомкам. Обратите внимание, что один из узлов соответствует операции `if`; это означает, что если значение левой ветви равно `true`, то он возвращает свою центральную ветвь, а если `false` – то правую ветвь.

Обход всего дерева эквивалентен следующей функции на языке Python:

```
def func(x,y)
  if x>3:
    return y + 5
  else:
    return y - 2
```

Может показаться, что подобные деревья годятся только для построения совсем простых функций. Но следует отметить две вещи. Во-первых, узлы, входящие в дерево, могут представлять очень сложные компоненты, например вычисление расстояния или гауссовой функции. Во-вторых, дерево может быть рекурсивным, если допускаются ссылки на узлы, расположенные выше. С помощью таких деревьев можно организовывать циклы и более сложные управляющие структуры.

Представление деревьев на языке Python

Теперь мы готовы к конструированию древовидных программ на языке Python. Дерево состоит из узлов, которые в зависимости от ассоциированных с ними функций могут иметь дочерние узлы. Некоторые узлы возвращают переданные программе параметры, другие – константы, но наиболее интересны узлы, возвращающие результат какой-то операции над своими дочерними узлами.

Создайте новый файл gp.py и в нем четыре класса – `fwrapper`, `node`, `paramnode` и `constnode`:

```
from random import random,randint,choice
from copy import deepcopy
from math import log

class fwrapper:
  def __init__(self,function,childcount,name):
```

```
        self.function=function
        self.childcount=childcount
        self.name=name

    class node:
      def __init__(self,fw,children):
        self.function=fw.function
        self.name=fw.name
        self.children=children

      def evaluate(self,inp):
        results=[n.evaluate(inp) for n in self.children]
        return self.function(results)

    class paramnode:
      def __init__(self,idx):
        self.idx=idx

      def evaluate(self,inp):
        return inp[self.idx]

    class constnode:
      def __init__(self,v):
        self.v=v

      def evaluate(self,inp):
        return self.v
```

Эти классы предназначены для следующих целей:

`fwrapper`

Обертка для функций, которые будут находиться в узлах, представляющих функции. Его члены – имя функции, сама функция и количество принимаемых параметров.

`node`

Класс функциональных узлов (имеющих потомков). Инициализируется экземпляром класса `fwrapper`. Метод `evaluate` вычисляет значения дочерних узлов и передает их представленной данным узлом функции в качестве параметров.

`paramnode`

Класс узлов, которые просто возвращают один из переданных программе параметров. Его метод `evaluate` возвращает параметр, соответствующий значению `idx`.

`constnode`

Узлы, возвращающие константы. Метод `evaluate` просто возвращает то значение, которым экземпляр был инициализирован.

Потребуются также функции, которые будут вызываться при посещении узлов. Такие функции вы должны будете написать и передать вместе с именем и счетчиком параметров конструктору класса `fwrapper`.

Добавьте в файл gp.py следующий список функций:

```
addw=fwrapper(lambda l:l[0]+l[1],2,'add')
subw=fwrapper(lambda l:l[0]-l[1],2,'subtract')
mulw=fwrapper(lambda l:l[0]*l[1],2,'multiply')

def iffunc(l):
  if l[0]>0: return l[1]
  else: return l[2]
ifw=fwrapper(iffunc,3,'if')

def isgreater(l):
  if l[0]>l[1]: return 1
  else: return 0
gtw=fwrapper(isgreater,2,'isgreater')

flist=[addw,mulw,ifw,gtw,subw]
```

Простые функции типа `add` и `subtract` можно встроить с помощью лямбда-выражений. Для остальных функцию придется написать в отдельном блоке. В любом случае функция обертывается в экземпляр класса `fwrapper` вместе со своим именем и числом параметров. В последней строке создается список всех функций, чтобы впоследствии из него можно было выбирать элементы случайным образом.

Построение и вычисление деревьев

Теперь с помощью класса `node` можно построить дерево программы, изображенное на рис. 11.2. Добавьте в файл gp.py функцию `exampletree`:

```
def exampletree( ):
  return node(ifw,[
                   node(gtw,[paramnode(0),constnode(3)]),
                   node(addw,[paramnode(1),constnode(5)]),
                   node(subw,[paramnode(1),constnode(2)]),
                  ]
              )
```

Запустите интерпретатор Python и протестируйте программу:

```
>>> import gp
>>> exampletree=gp.exampletree( )
>>> exampletree.evaluate([2,3])
1
>>> exampletree.evaluate([5,3])
8
```

Она успешно выполняет те же действия, что и эквивалентная ей функция. Таким образом, мы построили миниязык для представления древовидных программ и написали для него интерпретатор на Python. Этот язык можно легко расширить за счет дополнительных типов узлов. Он послужит нам в качестве основы для знакомства с генетическим программированием. Попробуйте написать еще несколько древовидных программ и убедитесь, что вы понимаете, как они работают.

Визуализация программы

Древовидные программы будут создаваться автоматически, об их структуре вы заранее ничего знать не будете. Поэтому необходим способ визуализации программы, так чтобы ее было легко интерпретировать. К счастью, при проектировании класса `node` мы предусмотрели, что в каждом узле будет храниться имя представляемой им функции в виде строки, поэтому функция вывода должна просто вернуть эту строку и отображаемые строки от своих дочерних узлов. Чтобы программу было проще читать, строки дочерних узлов нужно печатать с отступом, тогда будут сразу видны отношения родитель–потомок, существующие в дереве.

Создайте в классе `node` метод `display`, который выводит представление дерева в виде строки:

```
def display(self,indent=0):
  print (' '*indent)+self.name
  for c in self.children:
    c.display(indent+1)
```

Необходимо также добавить метод `display` в класс `paramnode`, где он будет просто печатать индекс возвращаемого параметра:

```
def display(self,indent=0):
  print '%sp%d' % (' '*indent,self.idx)
```

и в класс `constnode`:

```
def display(self,indent=0):
  print '%s%d' % (' '*indent,self.v)
```

Воспользуйтесь этими методами для распечатки дерева:

```
>>> reload(gp)
<module 'gp' from 'gp.py'>
>>> exampletree=gp.exampletree( )
>>> exampletree.display( )
if
 isgreater
  p0
  3
 add
  p1
  5
 subtract
  p1
  2
```

Если вы прочли главу 7, то, конечно, заметили сходство с представлением деревьев решений. В главе 7 было также показано, как отобразить деревья в графическом виде, более удобном для восприятия. Если хотите, можете воспользоваться той же идеей для графического представления древовидных программ.

Создание начальной популяции

Хотя программы для генетического программирования можно создавать и вручную, но обычно начальная популяция состоит из случайно сгенерированных программ. Это упрощает запуск процесса, поскольку отпадает необходимость проектировать несколько программ, которые почти решают задачу. Кроме того, таким образом в начальную популяцию вносится *разнообразие*, тогда как разные программы для решения одной задачи, написанные одним программистом, скорее всего, были бы похожи и, хотя давали бы почти правильный ответ, идеальное решение могло бы выглядеть совершенно иначе. О важности разнообразия мы еще поговорим чуть позже.

Для построения случайной программы нужно создать узел, поместив него случайно выбранную функцию, а затем – необходимое количество дочерних узлов, каждый из которых может иметь свои дочерние узлы и т. д. Как обычно при работе с деревьями, проще всего сделать это рекурсивно. Добавьте в файл gp.py функцию `makerandomtree`:

```
def makerandomtree(pc,maxdepth=4,fpr=0.5,ppr=0.6):
  if random( )<fpr and maxdepth>0:
    f=choice(flist)
    children=[makerandomtree(pc,maxdepth-1,fpr,ppr)
              for i in range(f.childcount)]
    return node(f,children)
  elif random( )<ppr:
    return paramnode(randint(0,pc-1))
  else:
    return constnode(randint(0,10))
```

Эта функция создает узел, содержащий случайно выбранную функцию, и проверяет, сколько у этой функции должно быть параметров. Для каждого дочернего узла функция вызывает себя рекурсивно, чтобы создать новый узел. Так конструируется все дерево, причем процесс построения ветвей завершается в тот момент, когда у очередного узла нет дочерних (то есть он представляет либо константу, либо переменную-параметр). Параметр `pc`, который будет встречаться на протяжении всей этой главы, равен числу параметров, принимаемых деревом на входе. Параметр `fpr` задает вероятность того, что вновь создаваемый узел будет соответствовать функции, а `ppr` – вероятность того, что узел, не являющийся функцией, будет иметь тип `paramnode`.

В сеансе интерпретатора постройте с помощью этой функции несколько программ и посмотрите, какие результаты будут получаться при различных параметрах:

```
>>> random1=gp.makerandomtree(2)
>>> random1.evaluate([7,1])
7
>>> random1.evaluate([2,4])
2
```

```
>>> random2=gp.makerandomtree(2)
>>> random2.evaluate([5,3])
1
>>> random2.evaluate([5,20])
0
```

Если все листовые узлы оказываются константами, то программа вообще не принимает параметров, поэтому результат ее работы не зависит от того, что вы подадите на вход. С помощью функции, написанной в предыдущем разделе, можно распечатать случайно сгенерированные деревья:

```
>>> random1.display( )
p0
>>> random2.display( )
subtract
 7
 multiply
  isgreater
   p0
   p1
  if
   multiply
    p1
    p1
   p0
   2
```

Выяснится, что некоторые деревья оказываются довольно глубокими, поскольку каждая ветвь продолжает расти, пока не встретится узел без потомков. Поэтому так важно ограничивать максимальную глубину, иначе растущее дерево может переполнить стек.

Проверка решения

Можно было бы считать, что теперь у вас есть все необходимое для автоматического построения программ, если бы вы могли позволить себе генерировать случайные программы, пока одна из них не окажется правильной. Очевидно, что это неприемлемо, поскольку возможных программ бесконечно много и шансы случайно наткнуться на правильную за разумное время бесконечно малы. Поэтому сейчас самое время ознакомиться со способами, позволяющими проверить, является ли решение правильным, и если нет, то насколько оно близко к правильному.

Простой математический тест

Один из самых простых тестов, применяемых в генетическом программировании, – реконструкция простой математической функции. Предположим, что задана таблица входных и выходных данных (табл. 11.1).

Таблица 11.1. Входные и выходные данные неизвестной функции

X	*Y*	Результат
26	35	829
8	24	141
20	1	467
33	11	1215
37	16	1517

Существует некая функция, отображающая заданные пары (X,Y) на результаты, но что это за функция, вам не сказали. Статистик мог бы попытаться применить к этой задаче регрессионный анализ, но для этого нужно знать структуру формулы. Другой вариант – построить прогностическую модель с помощью метода *k*-ближайших соседей, описанного в главе 8, но для этого требуется хранить все данные. В некоторых случаях нужна лишь формула, быть может, для встраивания в другую, гораздо более простую программу или чтобы объяснить человеку, что происходит.

Уверен, вы горите желанием узнать, какая функция скрыта за этими данными. Я вам скажу. Добавьте ее в файл gp.py:

```
def hiddenfunction(x,y):
return x**2+2*y+3*x+5
```

Мы воспользуемся этой функцией для построения набора данных, на котором будем проверять сгенерированные программы. Добавьте функцию `buildhiddenset` для создания набора данных:

```
def buildhiddenset( ):
  rows=[]
  for i in range(200):
    x=randint(0,40)
    y=randint(0,40)
    rows.append([x,y,hiddenfunction(x,y)])
  return rows
```

И вызовите ее в интерактивном сеансе:

```
>>> reload(gp)
<module 'gp' from 'gp.py'>
>>> hiddenset=gp.buildhiddenset( )
```

Вам-то известно, какой функцией генерировать набор данных, но для настоящей проверки требуется, чтобы программа реконструировала функцию без подсказки.

Измерение успеха

Как и в случае оптимизации, необходимо научиться измерять качество найденного решения. В данном случае мы сравниваем результаты

работы программы с известными числами, поэтому для проверки нужно посмотреть, насколько мало оказалось расхождение. Добавьте в файл gp.py следующую функцию:

```
def scorefunction(tree,s):
  dif=0
  for data in s:
    v=tree.evaluate([data[0],data[1]])
    dif+=abs(v-data[2])
  return dif
```

Эта функция перебирает все строки набора данных, вычисляет функцию от указанных в ней аргументов и сравнивает с результатом. Абсолютные значения разностей суммируются. Чем меньше сумма, тем лучше программа, а значение 0 говорит о том, что все результаты в точности совпали. Проверим, насколько хороши оказались сгенерированные ранее программы:

```
>>> reload(gp)
<module 'gp' from 'gp.py'>
>>> gp.scorefunction(random2,hiddenset)
137646
>>> gp.scorefunction(random1,hiddenset)
125489
```

Поскольку мы сгенерировали лишь две программы, причем совершенно случайные, шансы на отыскание правильной функции ничтожно малы. (Если одна из ваших программ дала правильную функцию, предлагаю отложить книгу и купить себе лотерейный билет.) Однако теперь у вас есть способ проверить, насколько хорошо программа дает прогноз математической функции. Это пригодится, когда мы будем решать, какие программы отобрать в следующее поколение.

Мутация программ

Отобранные наилучшие программы копируются и модифицируются для включения в следующее поколение. Выше уже отмечалось, что мутация заключается в небольшом изменении одной программы. Изменить древовидную программу можно разными способами, например изменив функцию в каком-то узле или одну из ветвей. Если для новой функции требуется другое количество дочерних узлов, то либо какие-то ветви удаляются, либо добавляются новые (рис. 11.3).

Другой способ мутации – замена какого-то поддерева целиком, как показано на рис. 11.4.

Мутации должны быть незначительными. Не следует, к примеру, заменять бóльшую часть узлов дерева. Вместо этого нужно задать относительно малую вероятность того, что какой-либо узел будет изменен. Далее вы начинаете обход дерева от корня, и если случайно выбранное

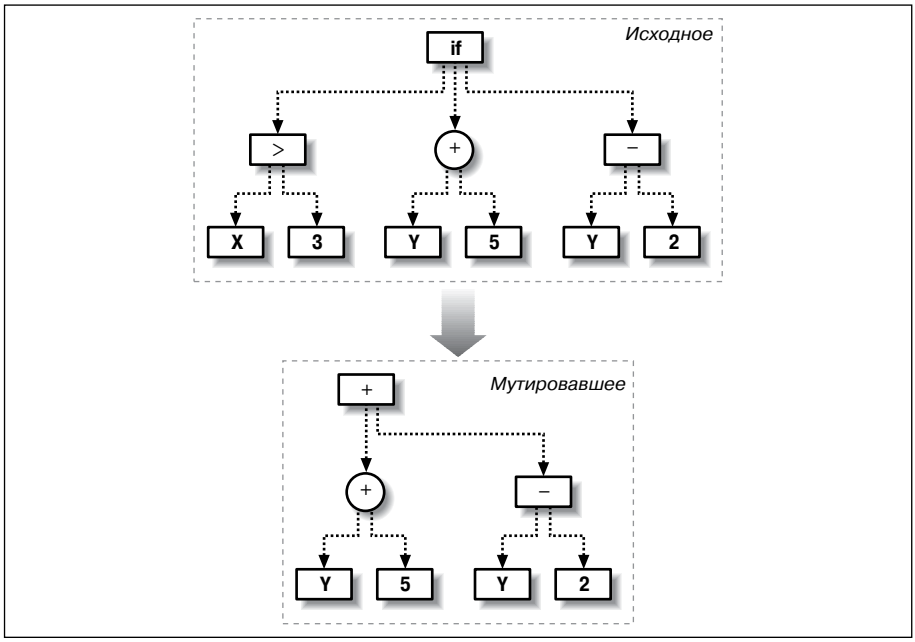


Рис. 11.3. *Мутация путем изменения функции в одном из узлов*

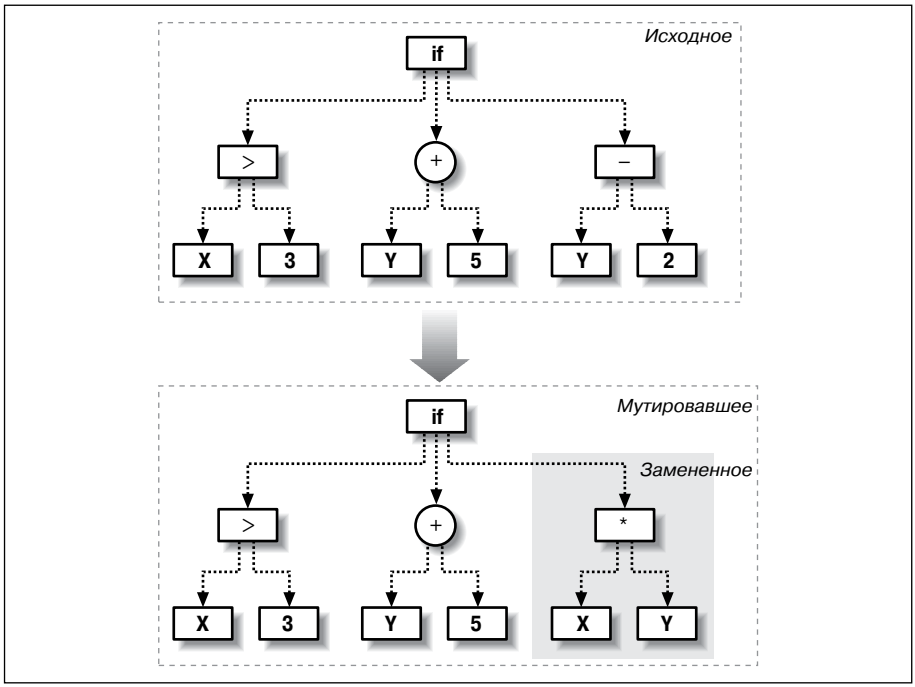


Рис. 11.4. *Мутация путем замены поддерева*

число оказалось меньше пороговой вероятности, то узел мутирует одним из описанных выше способов. В противном случае проверяются дочерние узлы.

Простоты ради мы реализуем только второй вид мутации. Создайте для выполнения этой операции функцию `mutate`:

```
def mutate(t,pc,probchange=0.1):
  if random( )<probchange:
    return makerandomtree(pc)
  else:
    result=deepcopy(t)
    if isinstance(t,node):
      result.children=[mutate(c,pc,probchange) for c in t.children]
    return result
```

Эта функция начинает с корня дерева и решает, следует ли изменить узел. Если нет, она рекурсивно вызывает `mutate` для дочерних узлов. Может случиться, что мутации подвергнутся все узлы, а иногда дерево вообще не изменится.

Несколько раз запустим функцию `mutate` для случайно сгенерированных программ и посмотрим, как она модифицирует деревья:

```
>>> random2.display( )
subtract
 7
 multiply
  isgreater
   p0
   p1
  if
   multiply
    p1
    p1
   p0
   2
>>> muttree=gp.mutate(random2,2)
>>> muttree.display( )
subtract
 7
 multiply
  isgreater
   p0
   p1
  if
   multiply
    p1
    p1
  p0
  p1
```

Посмотрим, существенно ли изменились результаты, возвращаемые функцией `scorefunction` после мутации, стали они лучше или хуже:

```
>>> gp.scorefunction(random2,hiddenset)
125489
>>> gp.scorefunction(muttree,hiddenset)
125479
```

Напомним, что мутации случайны, они необязательно должны вести к улучшению решения. Мы лишь надеемся, что в каких-то случаях результат станет лучше. Изменение будет продолжаться, и после смены нескольких поколений мы все-таки найдем наилучшее решение.

Скрещивание

Другой вид модификации программ – это скрещивание. Для этого две успешные программы комбинируются с целью получения новой. Обычно это делается путем замены какой-то ветви одной программы ветвью другой. На рис. 11.5 приведен соответствующий пример.

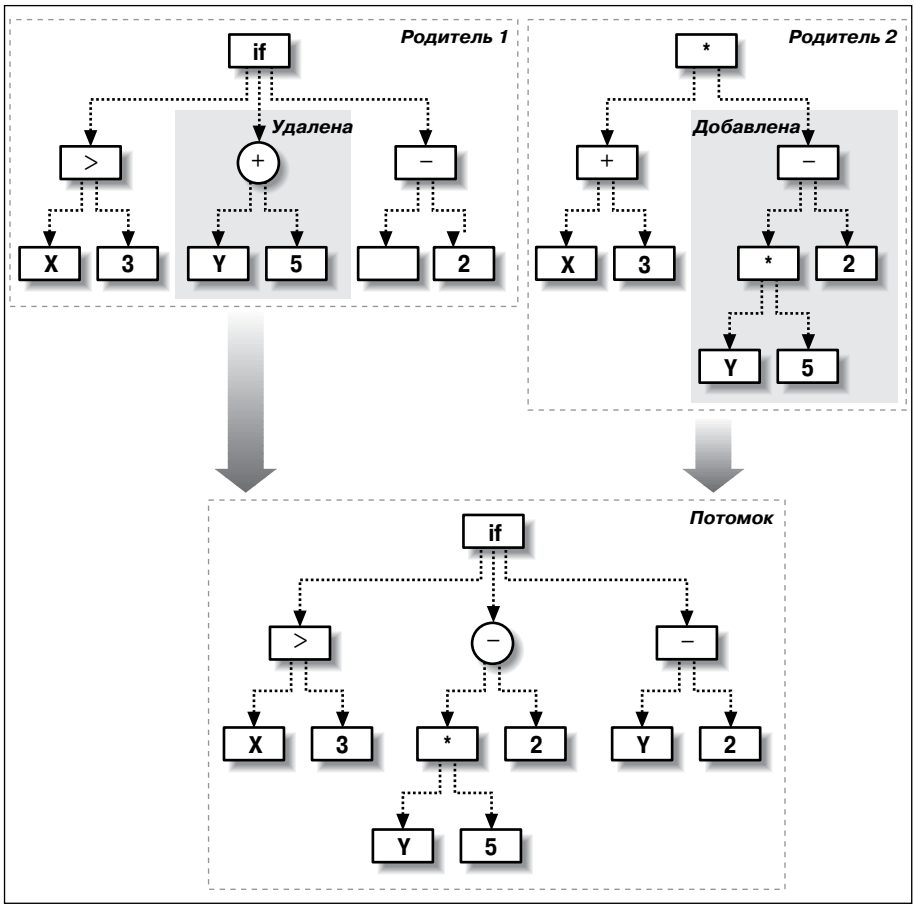


Рис. 11.5. Операция скрещивания

Функции, выполняющей скрещивание, передаются два дерева, и она обходит оба. Если случайно выбранное число не превышает пороговой вероятности, то функция возвращает копию первого дерева, в которой одна из ветвей заменена какой-то ветвью, взятой из второго дерева. Поскольку обход выполняется параллельно, то скрещивание произойдет примерно на одном уровне каждого дерева. Добавьте в файл gp.py функцию `crossover`:

```
def crossover(t1,t2,probswap=0.7,top=1):
  if random( )<probswap and not top:
    return deepcopy(t2)
  else:
    result=deepcopy(t1)
    if isinstance(t1,node) and isinstance(t2,node):
      result.children=[crossover(c,choice(t2.children),probswap,0)
                       for c in t1.children]
    return result
```

Попробуйте применить функцию `crossover` к нескольким случайно сгенерированным программам. Посмотрите, что получается после скрещивания. Получается ли иногда более удачная программа?

```
>>> random1=gp.makerandomtree(2)
>>> random1.display( )
multiply
 subtract
  p0
  8
 isgreater
  p0
  isgreater
   p1
   5
>>> random2=gp.makerandomtree(2)
>>> random2.display( )
if
 8
 p1
 2
>>> cross=gp.crossover(random1,random2)
>>> cross.display( )
multiply
 subtract
  p0
  8
 2
```

Вероятно, вы заметили, что перестановка ветвей может радикально изменить поведение программы. Кроме того, результат работы каждой программы может оказаться близок к правильному по совершенно различным причинам, поэтому скрещенная программа может давать результаты, совершенно не похожие ни на одного из родителей. Как и раньше, мы надеемся, что в результате некоторых скрещиваний решение удастся улучшить и оно перейдет в следующее поколение.

Построение окружающей среды

Вооружившись средством для измерения успеха и двумя методами модификации наилучших программ, мы можем перейти к созданию конкурентной среды, в которой программы будут эволюционировать. Необходимые шаги представлены блок-схемой на рис. 11.1. Смысл в том, чтобы создать набор случайных программ, отобрать из них наилучшие для копирования и модификации и повторять процесс, пока не будет выполнено некое условие останова.

Создайте новую функцию `evolve`, реализующую эту процедуру:

```
def evolve(pc,popsize,rankfunction,maxgen=500,
           mutationrate=0.1,breedingrate=0.4,pexp=0.7,pnew=0.05):
  # Возвращает случайное число, отдавая предпочтение более маленьким числам
  # Чем меньше значение pexp, тем больше будет доля маленьких чисел
  def selectindex( ):
    return int(log(random( ))/log(pexp))

  # Создаем случайную исходную популяцию
  population=[makerandomtree(pc) for i in range(popsize)]
  for i in range(maxgen):
    scores=rankfunction(population)
    print scores[0][0]
    if scores[0][0]==0: break

    # Две наилучшие особи отбираются всегда
    newpop=[scores[0][1],scores[1][1]]

    # Строим следующее поколение
    while len(newpop)<popsize:
      if random( )>pnew:
        newpop.append(mutate(
                      crossover(scores[selectindex( )][1],
                      scores[selectindex( )][1],
                      probswap=breedingrate),
                      pc,probchange=mutationrate))
      else:
        # Добавляем случайный узел для внесения неопределенности
        newpop.append(makerandomtree(pc))

    population=newpop
  scores[0][1].display( )
  return scores[0][1]
```

Эта функция создает случайную исходную популяцию. Затем она выполняет не более `maxgen` итераций цикла, вызывая каждый раз функцию `rankfunction` для ранжирования программ от наилучшей до наихудшей. Наилучшая программа автоматически попадает в следующее поколение без изменения. Иногда эту стратегию называют *элитизмом*. Все остальные особи в следующем поколении конструируются путем случайного выбора программ, расположенных близко к началу списка,

и применения к ним операций мутации и скрещивания. Процесс повторяется, пока не будет получено идеальное совпадение (расхождение с известным результатом равно 0) или не исчерпаются все итерации.

У функции есть несколько параметров, управляющих различными аспектами окружающей среды:

`rankfunction`

Функция, применяемая для ранжирования списка программ от наилучшей к наихудшей.

`mutationrate`

Вероятность мутации, передаваемая функции `mutate`.

`breedingrate`

Вероятность скрещивания, передаваемая функции `crossover`.

`popsize`

Размер исходной популяции.

`probexp`

Скорость убывания вероятности выбора программ с низким рангом. Чем выше значение, тем более суров процесс естественного отбора, то есть производить потомство разрешается только программам с наивысшим рангом.

`probnew`

Вероятность включения в новую популяцию совершенно новой случайно сгенерированной программы. Смысл параметров `probexp` и `probnew` мы рассмотрим в разделе «Важность разнообразия».

Последнее, что осталось сделать перед тем, как начать эволюцию программ, – реализовать способ их ранжирования по результатам, возвращенным `scorefunction`. Добавьте в файл gp.py функцию `getrankfunction`, которая возвращает функцию ранжирования для имеющегося набора данных:

```
def getrankfunction(dataset):
  def rankfunction(population):
    scores=[(scorefunction(t,dataset),t) for t in population]
    scores.sort( )
    return scores
  return rankfunction
```

Все готово для автоматического создания программы, которая будет искать математическую формулу, соответствующую набору данных. Сделаем это в интерактивном сеансе:

```
>>> reload(gp)
>>> rf=gp.getrankfunction(gp.buildhiddenset( ))
>>> gp.evolve(2,500,rf,mutationrate=0.2,breedingrate=0.1,pexp=0.7,pnew=0.1)
16749
10674
5429
3090
```

```
491
151
151
0
add
 multiply
  p0
  add
   2
   p0
  add
   add
    p0
    4
   add
    p1
    add
     p1
     isgreater
      10
      5
```

Числа изменяются медленно, но должны убывать, пока не достигнут 0. Интересно, что найденная функция правильна, но выглядит несколько сложнее той, с помощью которой набор был сгенерирован. (Вполне вероятно, что получившееся у вас решение тоже покажется сложнее, чем есть на самом деле.) Но с помощью простых алгебраических преобразований легко убедиться, что функции в действительности одинаковы – напомним, что `p0` – это `X`, а `p1` – `Y`. Получившееся дерево представляет следующую функцию:

```
(X*(2+X))+X+4+Y+Y+(10>5)
= 2*X+X*X+X+4+Y+Y+1
= X**2 + 3*X + 2*Y + 5
```

Мы продемонстрировали важную особенность генетического программирования: найденные решения могут быть правильными или очень хорошими, но из-за способа построения часто оказываются гораздо сложнее, чем то, что мог бы придумать человек. Нередко в программе обнаруживаются крупные участки, которые вообще не выполняют ничего полезного или вычисляют по сложной формуле одно и то же значение. Обратите внимание на узел `(10>5)` в построенной программе – это просто странный способ записи значения 1.

Можно заставить алгоритм строить простые программы, но во многих случаях это лишь затруднит поиск хорошего решения. Гораздо лучше позволить программам свободно эволюционировать, а потом упростить найденное хорошее решение, исключив лишние участки дерева. Иногда это можно сделать вручную, а иногда – автоматически с помощью алгоритма отсечения ветвей.

Важность разнообразия

Функция `evolve` среди прочего ранжирует программы от лучших к худшим, поэтому возникает искушение просто взять две-три программы, оказавшиеся в начале списка, и модифицировать их для включения в новую популяцию. В конце концов, зачем отказываться от лучшего?

Но проблема в том, что, выбирая всякий раз два лучших решения, мы быстро приходим к чрезмерно однородной популяции (если хотите, назовите это вырождением). Все решения в ней достаточно хороши, но мало изменяются, так как операции скрещивания порождают практически то же, что было раньше. Таким образом, мы выходим на локальный максимум – хорошее, но не идеальное состояние, в котором малые изменения не приводят к улучшению результата.

Как выясняется, комбинирование самых лучших решений с большим количеством умеренно хороших дает более качественные результаты. Поэтому у функции `evolve` есть два дополнительных параметра, которые позволяют внести разнообразие в процесс селекции. Уменьшая значение `probexp`, вы позволяете более слабым решениям принять участие в формировании результата, так что процесс описывается уже не фразой «выживают сильнейшие», а фразой «выживают сильнейшие и самые удачливые». Увеличивая значение `probnew`, вы позволяете иногда включать совершенно новые программы. Оба параметра повышают степень разнообразия эволюционного процесса, но не вмешиваются в него чрезмерно, так как худшие программы в конечном итоге все равно «вымирают».

Простая игра

Более интересной задачей для генетического программирования является создание искусственного интеллекта для игры. Программы можно заставить эволюционировать, принудив их состязаться между собой или с людьми, причем победителям предоставляются более высокие шансы перейти в следующее поколение. В этом разделе мы создадим симулятор для очень простой игры «Погоня» (рис. 11.6).

Два игрока по очереди делают ходы на небольшой расчерченной на клеточки доске. Можно перейти на любую из четырех соседних клеток, но размеры доски ограничены, поэтому игрок, пытающийся выйти за ее пределы, пропускает ход. Цель игры – взять соперника в плен, перейдя в свой ход на ту же клетку, где он сейчас стоит. Налагается лишь одно ограничение – нельзя два раза подряд ходить в одном и том же направлении, иначе будет засчитано поражение. Игра очень простая, но, поскольку в ней сталкиваются интересы двух участников, на ее примере можно изучить конкурентные аспекты эволюции.

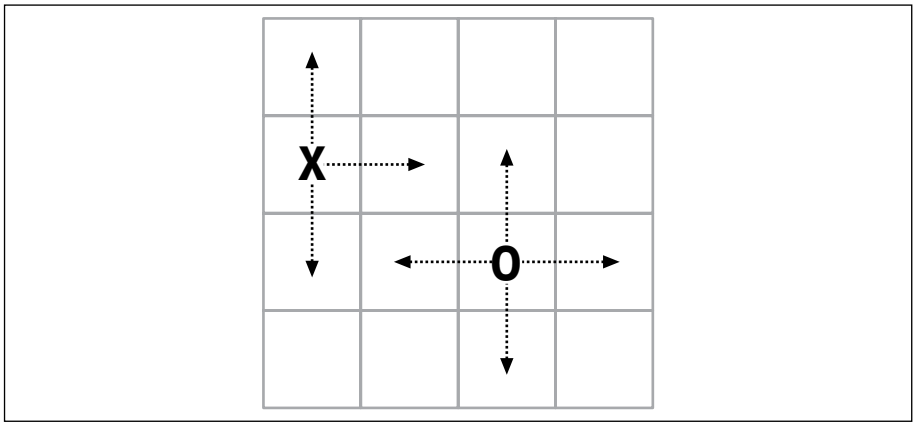


***Рис. 11.6.** Игра «Погоня»*

Прежде всего напишем функцию, которая имитирует игру двух участников. Функция попеременно передает каждой программе текущее положение ходящего игрока и его противника, а также последний сделанный ходящим игроком ход и возвращает в качестве результата новый ход.

Ход представляется числом от 0 до 3, обозначающим одно из четырех возможных направлений. Но так как случайные программы могут вернуть любое целое число, функция должна как-то привести результат к допустимому диапазону. Для этого она возвращает остаток от деления полученного числа на 4. Случайная программа может также создать игрока, который будет ходить по кругу, поэтому число ходов ограничивается – после 50 ходов объявляется ничья.

Добавьте в файл gp.py функцию `gridgame`:

```
def gridgame(p):
  # Размер доски
  max=(3,3)

  # Запоминаем последний ход каждого игрока
  lastmove=[-1,-1]

  # Запоминаем положения игроков
  location=[[randint(0,max[0]),randint(0,max[1])]]

  # Располагаем второго игрока на достаточном удалении от первого
  location.append([(location[0][0]+2)%4,(location[0][1]+2)%4])

  # Не более 50 ходов до объявления ничьей
  for o in range(50):

    # Для каждого игрока
    for i in range(2):
```

```
    locs=location[i][:]+location[1-i][:]
    locs.append(lastmove[i])
    move=p[i].evaluate(locs)%4

    # Если игрок два раза подряд ходит в одном направлении, ему
    # засчитывается проигрыш
    if lastmove[i]==move: return 1-i
    lastmove[i]=move
    if move==0:
      location[i][0]-=1
      # Доска ограничена
      if location[i][0]<0: location[i][0]=0
    if move==1:
      location[i][0]+=1
      if location[i][0]>max[0]: location[i][0]=max[0]
    if move==2:
      location[i][1]-=1
      if location[i][1]<0: location[i][1]=0
    if move==3:
      location[i][1]+=1
      if location[i][1]>max[1]: location[i][1]=max[1]

  # Если противник захвачен в плен, вы выиграли
  if location[i]==location[1-i]: return i
return -1
```

Программа возвращает 0, если выиграл первый игрок, 1 – если второй, и –1 – в случае ничьей. Попробуем создать две случайные программы и заставим их сыграть между собой:

```
>>> reload(gp)
<module 'gp' from 'gp.py'>
>>> p1=gp.makerandomtree(5)
>>> p2=gp.makerandomtree(5)
>>> gp.gridgame([p1,p2])
1
```

Программы еще не подвергались эволюции, поэтому, скорее всего, они будут проигрывать, делая два хода подряд в одном направлении. В идеале эволюционировавшая программа должна понять, что так делать нельзя.

Круговой турнир

Следуя идеологии коллективного разума, надо было бы проверять пригодность программ в игре против людей и соответственно проводить эволюцию. Было бы замечательно учесть при разработке «умной» программы поведение тысяч людей. Но при большой популяции и многих поколениях пришлось бы сыграть десятки тысяч игр, в большинстве своем с очень слабыми противниками. На практике это нереализуемо, поэтому сначала мы разовьем программы, заставив их состязаться друг с другом на турнире.

Функция `tournament` принимает на входе список игроков и организует игру каждого из них со всеми другими, отслеживая, сколько раз каждая программа проиграла. За проигрыш программе начисляется два очка, за ничью – одно. Добавьте эту функцию в файл gp.py:

```
def tournament(pl):
  # Массив для подсчета проигрышей
  losses=[0 for p in pl]

  # Каждый игрок встречается со всеми другими
  for i in range(len(pl)):
    for j in range(len(pl)):
      if i==j: continue

    # Кто выиграл?
    winner=gridgame([pl[i],pl[j]])

    # Два очка за поражение, одно за ничью
    if winner==0:
      losses[j]+=2
    elif winner==1:
      losses[i]+=2
    elif winner==-1:
      losses[i]+=1
      losses[i]+=1
      pass

  # Отсортировать и вернуть результаты
  z=zip(losses,pl)
  z.sort( )
  return z
```

В конце функция сортирует результаты и возвращает их вместе с программами, которые потерпели меньше всего поражений. Именно эта информация необходима функции `evolve`, чтобы организовать эволюцию, поэтому функция `tournament` может использоваться в качестве аргумента `evolve`, следовательно, у нас все готово для выбора программы-победителя. Выполните в интерактивном сеансе следующие команды (на это может уйти заметное время):

```
>>> reload(gp)
<module 'gp' from 'gp.py'>
>>> winner=gp.evolve(5,100,gp.tournament,maxgen=50)
```

Обратите внимание, что в ходе эволюции числа не убывают монотонно, как было при поиске математической функции. Подумайте, чем это можно объяснить, – ведь мы же отбираем в следующее поколение лучшего игрока, не так ли? Но поскольку следующее поколение целиком состоит из новых эволюционировавших программ, то лучший игрок прошлого поколения может проявить себя гораздо хуже в следующем.

Игра с человеком

Вы получили в ходе эволюции программу, которая победила всех своих кибернетических соперников. Теперь самое время сыграть с ней самому. Для этого создайте еще один класс, где также будет метод `evaluate`, который рисует доску и просит пользователя сделать ход. Добавьте в файл gp.py класс `humanplayer`:

```
class humanplayer:
  def evaluate(self,board):

    # Получить мою позицию и позиции других игроков
    me=tuple(board[0:2])
    others=[tuple(board[x:x+2]) for x in range(2,len(board)-1,2)]

    # Нарисовать доску
    for i in range(4):
      for j in range(4):
        if (i,j)==me:
          print 'O',
        elif (i,j) in others:
          print 'X',
        else:
          print '.',
      print

    # Показать ходы, для справки
    print 'Ваш последний ход %d' % board[len(board)-1]
    print ' 0'
    print '2 3'
    print ' 1'
    print 'Введите ход: ',

    # Вернуть введенное пользователем число
    move=int(raw_input( ))
    return move
```

Начинайте игру в интерактивном сеансе:

```
>>> reload(gp)
<module 'gp' from 'gp.py'>
>>> gp.gridgame([winner,gp.humanplayer( )])
. O . .
. . . .
. . . .
. . . X
Ваш последний ход -1
 0
2 3
 1
Введите ход:
```

Если программа хорошо эволюционировала, то победить ее будет довольно трудно. Ваша программа почти наверняка научилась не ходить два раза подряд в одном направлении, поскольку это влечет немедленную гибель, но то, в какой мере она освоила другие стратегии, зависит от конкретного прогона `evolve`.

Направления развития

Эта глава была лишь кратким введением в генетическое программирование – обширную и быстро развивающуюся дисциплину. Вы ознакомились с самыми простыми задачами, когда на построение программы уходят считанные минуты, а не дни, но те же принципы можно распространить и на более сложные проблемы. Наши популяции состояли из очень небольшого количества программ по сравнению с тем, что встречается в реальных задачах. Обычно популяция насчитывает тысячи или десятки тысяч особей. Рекомендую попробовать свои силы на более трудных задачах и увеличить размер популяции, но будьте готовы ждать несколько часов или дней, пока программа будет работать.

В следующем разделе мы наметим несколько направлений развития простой модели генетического программирования для различных приложений.

Другие числовые функции

До сих пор для построения программ использовался очень небольшой набор функций. Это ограничивает возможности программы. Чтобы решать более сложные задачи, необходимо намного увеличить количество функций, включаемых в дерево. Вот несколько предложений:

- Тригонометрические функции – синус, косинус и тангенс.
- Другие математические функции – возведение в степень, извлечение квадратного корня и взятие абсолютного значения.
- Статистические распределения, например гауссово.
- Метрики, например евклидово расстояние и коэффициент Танимото.
- Функция с тремя параметрами, которая возвращает 1, если первый параметр расположен между вторым и третьим.
- Функция с тремя параметрами, которая возвращает 1, если расстояние между первыми двумя параметрами меньше третьего.

Сложность функций ничем не ограничена, а подбираются они исходя из особенностей конкретной задачи. Скажем, тригонометрические функции необходимы для обработки сигналов, но вряд ли пригодятся в рассмотренной выше игре.

Память

Рассмотренные в этой главе программы, по существу, являются реагирующими: возвращаемый результат зависит исключительно от входных данных. Для поиска математических функций этот подход правилен, но не позволяет программам вырабатывать долговременную стратегию. Игра в преследование передает программе последний сделанный ею ход – главным образом для того, чтобы программа научилась не ходить в одном направлении дважды, – но это еще один входной параметр, а не что-то, устанавливаемое программой самостоятельно.

Чтобы программа могла выработать долговременную стратегию, ей необходимо как-то сохранять информацию, которой она сможет воспользоваться в следующем цикле. Есть простой способ – ввести дополнительные виды узлов, которые способны сохранить данные в предопределенных ячейках и извлекать их оттуда. Узел *сохранения* имеет единственный дочерний узел и индекс ячейки памяти; он получает результат от своего потомка, сохраняет его в ячейке памяти и передает родителю. У узла *восстановления* нет потомков, он просто возвращает значение, хранящееся в соответствующей ячейке. Если узел сохранения находится в корне дерева, то конечный результат доступен в любой точке дерева, где имеется соответствующий узел восстановления.

Помимо индивидуальной памяти можно организовать еще и общую память, доступную для чтения и записи всем программам. Она устроена аналогично индивидуальной памяти, однако работает с ячейками, доступными всем программам. Это повышает уровень кооперации и конкуренции.

Различные типы данных

Описанный каркас ориентирован на программы, принимающие и возвращающие целые числа. Его легко модифицировать для работы с числами с плавающей точкой, так как операции при этом остаются теми же самыми. Для этого измените функцию `makerandomtree` так, чтобы она инициализировала константные узлы значениями с плавающей точкой, а не целыми.

Для построения программ, способных обрабатывать данные других типов, потребуется более масштабная модификация, сводящаяся главным образом к изменению функций в узлах. Основной каркас можно приспособить для работы со следующими типами данных:

Строки

Потребуются такие операции, как конкатенация, разбиение, поиск по индексу и выделение подстрок.

Списки

Потребуются те же операции, что для строк.

Словари

Потребуются операции замены и добавления.

Объекты

На вход дереву можно подать произвольный объект, тогда функции в узлах будут интерпретироваться как вызовы методов этого объекта.

Важно отметить один существенный момент – во многих случаях требуется, чтобы узлы дерева могли обрабатывать значения разных типов. Так, в операции выделения строки на вход подаются строка и два целых числа, то есть один из дочерних узлов должен возвращать строку, а два остальных – целые числа.

Наивный подход к этой проблеме состоит в том, чтобы случайно генерировать деревья, подвергать их мутации и скрещиванию, а потом отбрасывать те, где обнаружилось несоответствие типов данных. Но это приводит к бесполезному расходованию вычислительных ресурсов, а кроме того, вы уже видели, как можно наложить ограничения при построении дерева, – каждая функция в целочисленных деревьях знает о количестве своих параметров. Эту идею легко обобщить на типы дочерних узлов и возвращаемых ими значений. Например, можно следующим образом переписать класс `fwrapper`, задавая в `params` список строк, описывающих тип данных каждого параметра:

```
class fwrapper:
  def __init__(self,function,params,name):
    self.function=function
    self.childcount=param
    self.name=name
```

Вероятно, вы захотите подготовить словарь `flist`, содержащий типы возвращаемых функциями значений, например:

```
flist={'str':[substringw,concatw],'int':[indexw,addw,subw]}
```

Теперь начало функции `makerandomtree` можно переписать так:

```
def makerandomtree(pc,datatype,maxdepth=4,fpr=0.5,ppr=0.5):
  if random( )<fpr and maxdepth>0:
    f=choice(flist[datatype])
    # Вызвать makerandomtree, указав типы всех параметров f
    children=[makerandomtree(pc,type,maxdepth-1,fpr,ppr)
              for type in f.params]
    return node(f,children)
и т. д.
```

Функцию `crossover` также придется изменить, чтобы переставляемые узлы гарантированно возвращали значение одного и того же типа.

Хотелось бы надеяться, что из этого раздела вы почерпнули некоторые идеи о том, как можно развить описанную здесь простую модель генетического программирования, и загорелись желанием улучшить ее и применить к автоматическому генерированию программ для решения

более сложных задач. Хотя для генерации может потребоваться много времени, но, отыскав хорошую программу, вы сможете применять ее снова и снова.

Упражнения

1. *Дополнительные функции.* Мы начали с очень небольшого списка функций. Какие еще функции вы могли бы предложить? Реализуйте функцию вычисления евклидова расстояния с четырьмя параметрами.
2. *Мутация заменой узла.* Реализуйте процедуру мутации, которая выбирает произвольный узел в дереве и изменяет его. Позаботьтесь о правильной обработке узлов, представляющих функции, константы и параметры. Как отразится на результате эволюции использование этой функции вместо замены ветви?
3. *Случайное скрещивание.* Сейчас функция скрещивания выбирает ветви двух деревьев, расположенные на одном уровне. Напишите другую функцию скрещивания, которая будет переставлять две произвольные ветви. Как это отразится на результате эволюции?
4. *Прекращение эволюции.* Добавьте в функцию `evolve` условие, которое прекращает процесс и возвращает полученный результат, если ранг наилучшей программы не уменьшается на протяжении X поколений.
5. *Скрытые функции.* Попробуйте предложить программе угадать другие математические функции. Какие функции найти легко, а какие – трудно?
6. *Игра «Погоня».* Попробуйте вручную составить древовидную программу, которая хорошо играет в игру «Погоня». Если с этой задачей вы справились легко, попробуйте написать совершенно иную программу для той же цели. Вместо того чтобы создавать полностью случайную исходную популяцию, сделайте ее почти случайной, добавив написанные вручную программы. Сравните свои программы со случайными. Можно ли их улучшить в процессе эволюции?
7. *Крестики-нолики.* Постройте симулятор игры в крестики-нолики. Организуйте турнир, аналогичный турниру для игры «Погоня». Насколько хорошо играют получающиеся программы? Можно ли обучить их беспроигрышной стратегии?
8. *Узлы с типами данных.* В этой главе было представлено несколько мыслей по поводу того, как реализовать узлы со смешанными типами данных. Сделайте это и попытайтесь путем эволюции получить программу, которая возвращает второй, третий, шестой и седьмой символы строки (то есть из genetic получает enic).

12

Сводка алгоритмов

В этой книге вы ознакомились с разнообразными алгоритмами, и если вы проработали приведенные примеры, то теперь для их реализации у вас есть работоспособные программы на языке Python. В предыдущих главах давались примеры некоторой задачи и описания различных алгоритмов ее решения. Сейчас мы приведем сводку всех рассмотренных алгоритмов, так что, если вам понадобится применить какой-нибудь метод получения информации или машинного обучения к новому набору данных, вы можете, используя эту главу как справочник, найти подходящий алгоритм и воспользоваться ранее написанным кодом.

Чтобы не заставлять вас возвращаться назад в поисках деталей, я приведу описание каждого алгоритма и высокоуровневый обзор принципов его работы, напомню, к каким наборам данных он применим и как следует пользоваться написанной ранее реализацией. Я отмечу также сильные и слабые стороны каждого алгоритма (если хотите, можете считать этот материал руководством по продаже идеи своему начальнику). В некоторых случаях особенности алгоритмов будут проиллюстрированы на примерах. Примеры нарочито упрощены – по большей части они настолько просты, что для решения достаточно одного взгляда на данные, – но полезны в качестве иллюстрации.

Сначала обратимся к методам обучения с учителем, они позволяют провести классификацию или спрогнозировать значение исходя из ранее предъявленных примеров.

Байесовский классификатор

Байесовские классификаторы рассматривались в главе 6. Мы показали, как построить систему классификации документов, например, для фильтрации спама или разбиения множества документов по категориям при наличии неоднозначных результатов поиска по ключевым словам.

Хотя все примеры были связаны с документами, описанный байесовский классификатор применим к любому набору данных, который можно представить в виде списков *признаков*. Признаком считается любое свойство, которое может присутствовать или отсутствовать в образце. Для документов признаками служат слова, но это могут быть характеристики неидентифицированного объекта, симптомы заболевания и вообще все, о чем можно сказать, имеется оно или нет.

Обучение

Как и все алгоритмы обучения с учителем, байесовский классификатор обучается на примерах. Пример – это список признаков образца и его классификация. Предположим, что вы хотите научить классификатор распознавать, относится ли документ, содержащий слово python к языку программирования или к змеям. В табл. 12.1 приведен пример обучающего набора для этой цели.

Таблица 12.1. Признаки и классификация набора документов

Признаки	Классификация
Питоны – это констрикторы, которые питаются птицами и млекопитающими	Змея
Python с самого начала разрабатывался как язык сценариев	Язык
В Индонезии нашли питона длиной 49 футов	Змея
В Python реализована динамическая система типов	Язык
Питон с яркими чешуйками	Змея
Проект с открытыми исходными текстами	Язык

Классификатор запоминает все встретившиеся признаки, а также вероятности того, что признак ассоциирован с конкретной классификацией. Примеры предъявляются классификатору по одному. После каждого примера классификатор обновляет свои данные, вычисляя вероятность того, что документ из указанной категории содержит то или иное слово. Например, после обучения на показанном выше примере может получиться набор вероятностей, представленный в табл. 12.2.

Таблица 12.2. Вероятности принадлежности слов категориям

Признак	Язык	Змея
Динамический	0,6	0,1
Констриктор	0,0	0,6
Длина	0,1	0,2
Исходный	0,3	0,1
И	0,95	0,95

Из этой таблицы видно, что после обучения ассоциации признаков с различными категориями становятся сильнее или слабее. Слово «конструктор» имеет бóльшую вероятность для змей, а слово «динамический» – для языка программирования. Неоднозначные признаки, например союз «и», имеют близкие вероятности для обеих категорий (слово «и» встречается почти в любом документе вне зависимости от его тематики). Обученный классификатор – это всего лишь список признаков вместе с ассоциированными вероятностями. В отличие от некоторых других методов классификации, здесь не нужно хранить исходные данные, на которых проводилось обучение.

Классификация

После обучения байесовский классификатор можно применять для автоматической классификации новых образцов. Предположим, что имеется документ, содержащий слова «длина», «динамический» и «исходный». В табл. 12.2 имеются вероятности для каждого слова, но они вычислены лишь для отдельных слов. Если бы все слова с высокой вероятностью попадали в одну и ту же категорию, то ответ был бы ясен. Однако для слова «динамический» выше вероятность попадания в категорию «Язык», а для слова «длина» – в категорию «Змея». Чтобы классифицировать новый документ, необходимо по вероятностям признаков вычислить вероятность всего образца.

В главе 6 описан один из методов решения этой задачи – наивный байесовский классификатор. Он вычисляет совокупную вероятность по следующей формуле:

$$Pr(\text{Категория} \mid \text{Документ}) = Pr(\text{Документ} \mid \text{Категория}) \times Pr(\text{Категория})$$

где

$$Pr(\text{Документ} \mid \text{Категория}) = Pr(\text{Слово1} \mid \text{Категория}) \times \\ \times Pr(\text{Слово2} \mid \text{Категория}) \times \ldots$$

Величины $Pr(\text{Слово} \mid \text{Категория})$ – значения, взятые из таблицы, например $Pr(\text{Динамический} \mid \text{Язык}) = 0{,}6$. Величина $Pr(\text{Категория})$ равна полной частоте встречаемости данной категории. Поскольку категория «Язык» встречалась в половине случаев, то $Pr(\text{Язык}) = 0{,}5$. Результатом считается та категория, для которой $Pr(\text{Категория} \mid \text{Документ})$ принимает максимальное значение.

Использование ранее написанного кода

Чтобы применить построенный в главе 6 байесовский классификатор к произвольному набору данных, необходимо написать функцию выделения признаков, которая преобразует данные, используемые на этапе обучения или классификации, в список признаков. В упомянутой главе мы работали с документами, поэтому функция разбивала строки на слова, но, в принципе, годится любая функция, принимающая некоторый объект и возвращающая список:

```
>>> docclass.getwords('python is a dynamic language')
{'python': 1, 'dynamic': 1, 'language': 1}
```

Этой функцией можно воспользоваться для создания нового классификатора, обучения его на примерах строк:

```
>>> cl=docclass.naivebayes(docclass.getwords)
>>> cl.setdb('test.db')
>>> cl.train('pythons are constrictors','snake')
>>> cl.train('python has dynamic types','language')
>>> cl.train('python was developed as a scripting language','language')
```

и последующей классификации:

```
>>> cl.classify('dynamic programming')
u'language'
>>> cl.classify('boa constrictors')
u'snake'
```

Количество категорий не ограничено, но, если вы хотите, чтобы классификатор работал хорошо, ему нужно предъявить достаточно много примеров из каждой категории.

Сильные и слабые стороны

Пожалуй, самое существенное преимущество наивных байесовских классификаторов по сравнению с другими методами заключается в том, что их можно обучать и затем опрашивать на больших наборах данных. Даже если обучающий набор очень велик, обычно для каждого образца есть лишь небольшое количество признаков, а обучение и классификация сводятся к простым математическим операциям над вероятностями признаков.

Это особенно важно, когда обучение проводится инкрементно, – каждый новый предъявленный образец можно использовать для обновления вероятностей без использования старых обучающих данных. (Обратите внимание, что код для обучения байесовского классификатора запрашивает по одному образцу за раз, тогда как для других методов, скажем деревьев решений или машин опорных векторов, необходимо предъявлять сразу весь набор.) Поддержка инкрементного обучения очень важна для таких приложений, как антиспамный фильтр, который постоянно обучается на вновь поступающих сообщениях, должен обновляться быстро и, возможно, даже не имеет доступа к старым сообщениям.

Еще одно достоинство наивных байесовских классификаторов – относительная простота интерпретации того, чему классификатор обучился. Поскольку вероятности всех признаков сохраняются, можно в любой момент заглянуть в базу данных и посмотреть, какие признаки оптимальны для отделения спама от нормальных сообщений или языков программирования от змей. Эта информация интересна сама по себе и может быть использована в других приложениях или как отправная точка для обучения других классификаторов.

Основной недостаток наивных байесовских классификаторов – их неспособность учитывать зависимость результата от сочетания признаков. Рассмотрим следующий сценарий отделения спама от нормальных сообщений. Вы занимаетесь созданием веб-приложений, поэтому в рабочей почте часто встречается слово online. Ваш лучший друг работает в фармацевтической промышленности и любит посылать смешные истории о том, что происходит у него на работе. Но, как и все люди, неосмотрительно раскрывшие кому-то свой адрес, вы периодически получаете спам, содержащий фразу online pharmacy (онлайновая аптека).

Вы, конечно, уже поняли, в чем проблема – классификатору постоянно говорят, что слова online и pharmacy встречаются в нормальных сообщениях, поэтому их вероятность для этой категории высока. Если же вы говорите классификатору, что некое сообщение со словами online pharmacy – спам, то их вероятности слегка корректируются в сторону спама. И эта битва протекает постоянно. Поскольку вероятности всем признакам назначаются независимо, то классификатор не способен опознать их комбинации. При классификации документов это, может быть, и не страшно, поскольку сообщение со словосочетанием online pharmacy (онлайновая аптека), вероятно, содержит и другие индикаторы спама, но в иных задачах умение распознавать комбинации признаков может оказаться гораздо более важным.

Классификатор на базе деревьев решений

Деревья решений рассматривались в главе 7 на примере построения модели поведения пользователя исходя из записей в протоколах сервера. Отличительной особенностью деревьев решений является исключительная простота интерпретации. На рис. 12.1 показан пример дерева:

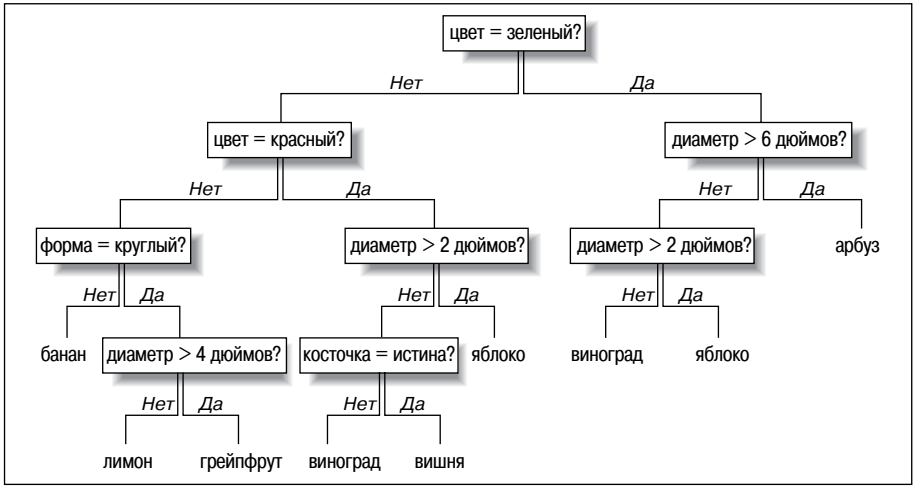


Рис. 12.1. *Пример дерева решений*

Из рисунка понятно, что делает дерево решений, сталкиваясь с задачей классификации нового образца. Начиная с корня, для образца проверяется условие, хранящееся в узле. Если образец удовлетворяет условию, мы переходим по ветви «Да», иначе – по ветви «Нет». Процесс повторяется, пока не будет достигнут листовый узел, который и соответствует спрогнозированной категории.

Обучение

Классификация с помощью дерева решения проста, обучение несколько сложнее. Описанный в главе 7 алгоритм строит дерево, начиная с корня. На каждом шаге выбирается атрибут, который оптимально разбивает данные. Для иллюстрации рассмотрим набор данных о фруктах в табл. 12.3. Будем называть его исходным набором.

Таблица 12.3. Данные о фруктах

Диаметр	Цвет	Фрукт
4	Красный	Яблоко
4	Зеленый	Яблоко
1	Красный	Вишня
1	Зеленый	Виноград
5	Красный	Яблоко

Разбить набор данных можно по любой из двух переменных: диаметр или цвет – и в обоих случаях будет создан корень дерева. На первом шаге мы должны попробовать обе переменные и решить, какая дает лучшее разбиение. При разбиении по цвету получаются результаты, показанные в табл. 12.4.

Таблица 12.4. Разбиение данных о фруктах по цвету

Красный	Зеленый
Яблоко	Яблоко
Вишня	Виноград
Яблоко	

Данные все еще перемешаны. Если же провести разбиение набора по диаметру (меньше 4 дюймов и больше или равен 4 дюймам), то результат (табл. 12.5) окажется гораздо более понятным; данные в левом столбце будем далее называть поднабором 1, а в правом – поднабором 2.

Таблица 12.5. Разбиение данных о фруктах по диаметру

Диаметр < 4 дюймов	Диаметр ≥ 4 дюймов
Вишня	Яблоко
Виноград	Яблоко

Это намного лучше, потому что в поднабор 1 попали все яблоки (apple) из исходного набора. В данном примере оптимальная переменная видна сразу, но в больших наборах данных столь очевидное разбиение бывает далеко не всегда. В главе 7 было введено понятие энтропии (меры беспорядочности множества) для измерения качества разбиения:

- $p(i)$ = частота вхождения = количество вхождений / количество строк
- Энтропия = сумма $p(i) \times \log(p(i))$ по всем результатам

Если у множества низкая энтропия, то оно близко к однородному, а значение 0 означает, что все элементы множества одинаковы. Энтропия поднабора 2 (диаметр ≥ 4) в табл. 12.5 равна 0. Энтропии обоих наборов используются для вычисления *информационного выигрыша*:

- вес 1 = размер поднабора 1 / размер исходного набора
- вес 2 = размер поднабора 2 / размер исходного набора
- выигрыш = энтропия(начальная) − вес 1 × энтропия(поднабор 1) − − вес 2 × энтропия(поднабор 2)

Информационный выигрыш вычисляется для каждого возможного разбиения, и в зависимости от него выбирается разделяющая переменная. Как только выбрана разделяющая переменная, можно создавать первый узел (рис. 12.2).

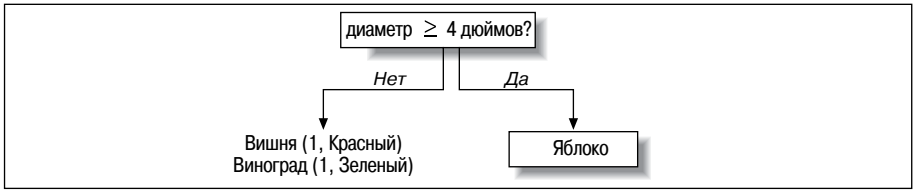


Рис. 12.2. *Корневой узел дерева решений для набора данных о фруктах*

В само́м узле показано условие. Если данные удовлетворяют условию, то выбирается ветвь «Да», иначе − ветвь «Нет». Поскольку в ветви «Да» всего один узел, он становится листовым. В ветви «Нет» данные еще разнородны, поэтому ее можно разбить далее, применяя тот же самый метод. В данном случае наилучшей переменной для разбиения будет цвет. Процесс продолжается до тех пор, пока информационный выигрыш при делении данных не прекращает уменьшаться.

Использование классификатора на основе дерева решений

В главе 7 данные для обучения деревьев решений были представлены в виде списка списков. В каждом внутреннем списке хранится набор значений, причем последним идет категория. Набор данных о фруктах будет выглядеть следующим образом:

```
>>> fruit=[[4,'red','apple'],
... [4,'green','apple'],
```

```
... [1,'red','cherry'],
... [1,'green','grape'],
... [5,'red','apple']]
```

Теперь можно обучить дерево решений и использовать его для классификации новых образцов:

```
>>> import treepredict
>>> tree=treepredict.buildtree(fruit)
>>> treepredict.classify([2,'red'],tree)
{'cherry': 1}
>>> treepredict.classify([5,'red'],tree)
{'apple': 3}
>>> treepredict.classify([1,'green'],tree)
{'grape': 1}
>>> treepredict.classify([120,'red'],tree)
{'apple': 3}
```

Очевидно, что предмет шириной 10 футов красного цвета яблоком (apple) не является, но дерево решений знает только о том, что ему предъявили. Чтобы понять, как дерево принимает решение, можно его распечатать или представить в графическом виде:

```
>>> treepredict.printtree(tree)
0:4?
T-> {'apple': 3}
F-> 1:green?
  T-> {'grape': 1}
  F-> {'cherry': 1}
```

Сильные и слабые стороны

Бросающееся в глаза достоинство деревьев решений – простота интерпретации обученной модели и то, как хорошо алгоритм помещает наиболее важные факторы ближе к корню дерева. Это означает, что дерево решений полезно не только для классификации, но и для интерпретации результатов. Как и в случае байесовского классификатора, можно «заглянуть под капот» и понять, почему получился именно такой, а не иной результат. Это может облегчить принятие решений вне процесса классификации. Например, в главе 7 модель предсказывала, какие пользователи станут платными клиентами; наличие дерева решений, показывающего, какие переменные наиболее существенны для разбиения данных, помогает при планировании рекламной кампании или принятии решения о том, какие данные собирать.

Деревья решения могут работать не только с дискретными, но и с числовыми данными, поскольку ищут разделяющую линию, которая максимизирует информационный выигрыш. Умение смешивать дискретные и числовые данные полезно для многих классов задач, а традиционные статистические методы, например регрессионный анализ, испытывают в этой части затруднения. С другой стороны, деревья решения не так хороши для прогнозирования числовых результатов.

Данные можно разделить по критерию минимальной дисперсии, но если они сложны, то дерево окажется очень большим и неспособным давать точные прогнозы.

Основное преимущество деревьев решений по сравнению с байесовским классификатором – способность легко справляться с взаимозависимыми переменными. Антиспамный фильтр, построенный на базе дерева решений, легко определит, что слова online и pharmacy (аптека) по отдельности нормальные, но при употреблении вместе являются индикатором спама.

К сожалению, применение алгоритма из главы 7 к фильтрации спама на практике невозможно, так как он не поддерживает инкрементное обучение. (Альтернативные алгоритмы деревьев решений, которые такое обучение поддерживают, – предмет активных исследований.) Можно взять большой набор документов, построить по нему дерево решений, но учесть новые сообщения оно уже не сможет, его придется каждый раз заново переучивать. Поскольку многие люди хранят десятки тысяч сообщений, переучивание становится практически нереализуемым. Кроме того, поскольку количество возможных узлов очень велико (каждый признак либо присутствует, либо отсутствует), то дерево может оказаться чрезмерно большим и сложным, что замедлит процесс классификации.

Нейронные сети

В главе 4 было показано, как построить простую нейронную сеть для изменения ранга результатов в зависимости от того, по каким ссылкам пользователи переходили в прошлом. Эта нейронная сеть смогла обучиться тому, какие слова и в каких сочетаниях важны, а какие вовсе несущественны для конкретного запроса. Нейронные сети можно применять к задачам классификации и прогнозирования числовых результатов.

Нейронная сеть из главы 4 применялась в качестве классификатора – она возвращала числовое значение для каждой ссылки и прогнозировала, что пользователь скорее всего перейдет по ссылке с максимальным значением. Поскольку значение возвращалось для каждой ссылки, то их совокупность можно было использовать для ранжирования результатов поиска.

Существует много видов нейронных сетей. Та, что рассмотрена в настоящей книге, называется многоуровневым перцептроном. Своим названием она обязана тому, что имеется входной уровень нейронов, который передают сигналы на один или несколько скрытых уровней. Принципиальная структура сети изображена на рис. 12.3.

В этой сети есть два уровня нейронов. Нейроны из разных уровней связаны между собой *синапсами*, каждому из которых приписан вес. Выходные сигналы от нейронов на одном уровне передаются нейронам

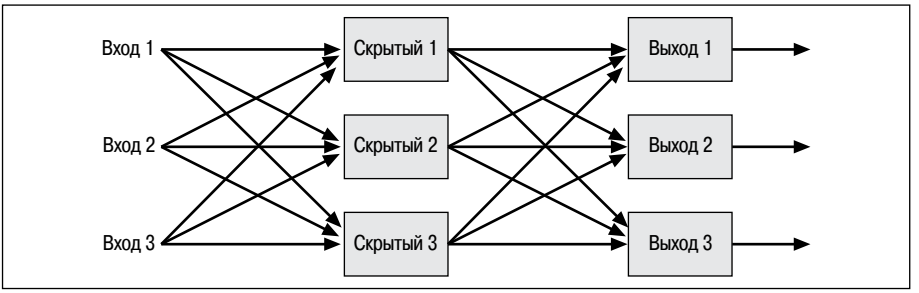


Рис. 12.3. *Принципиальная структура нейронной сети*

следующего уровня по синапсам. Чем выше вес синапса, исходящего из некоторого нейрона, тем большее влияние он оказывает на выходной сигнал от этого нейрона.

В качестве простого примера снова рассмотрим задачу о фильтрации спама, описанную в разделе «Байесовский классификатор». В нашем упрощенном мире почтовое сообщение может содержать слова online и pharmacy (аптека) по отдельности или вместе. Чтобы определить, какие сообщения являются спамом, необходима нейронная сеть, показанная на рис. 12.4.

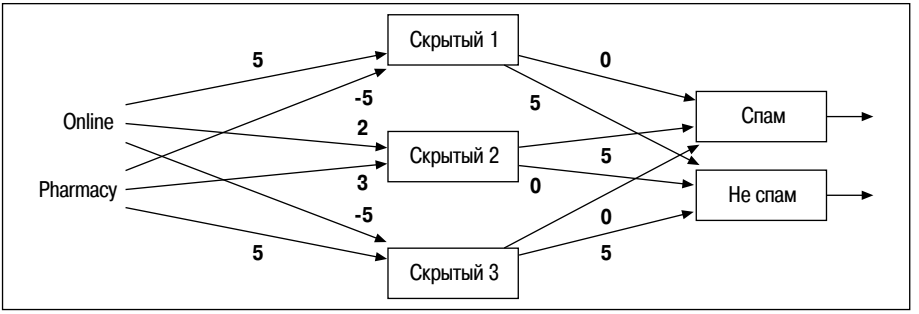


Рис. 12.4. *Нейронная сеть для классификации спама*

Здесь веса синапсов уже установлены для решения данной задачи. (Как это делается, вы увидите в следующем разделе.) Нейроны первого уровня реагируют на слова, выступающие в роли входных сигналов, – если слово присутствует в сообщении, то активируются нейроны, сильно связанные с этим словом. Второй уровень получает сигналы от первого, то есть реагирует на словосочетания.

И наконец нейроны скрытого уровня передают результаты на выходной уровень, а конкретные словосочетания могут быть сильно или слабо связаны с возможными результатами. Окончательное решение принимается исходя из того, какой нейрон выходного уровня получил самый сильный сигнал. На рис. 12.5 показано, как сеть реагирует на слово online, когда оно не сопровождается словом pharmacy (аптека).

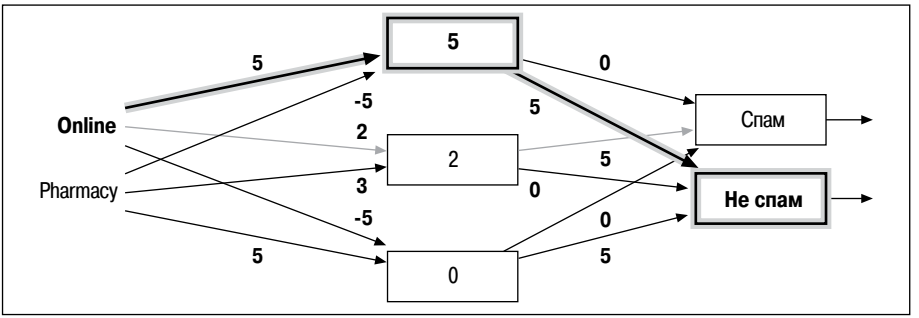


Рис. 12.5. *Реакция сети на слово online*

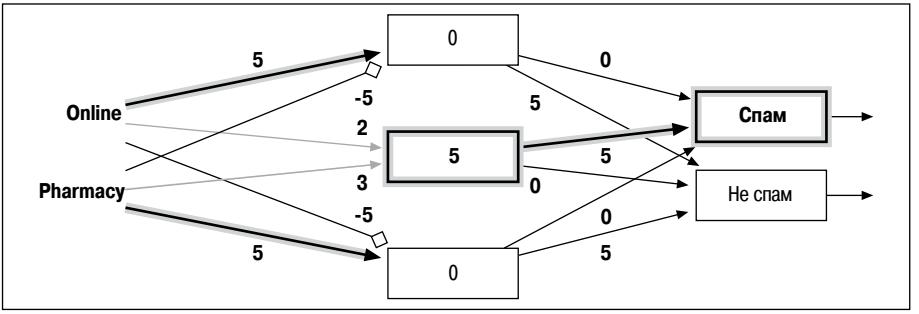


Рис. 12.6. *Реакция нейронной сети на словосочетание online pharmacy (онлайновая аптека)*

Один из нейронов на первом уровне реагирует на слово online и посылает сигнал на второй уровень, где определенный нейрон обучился распознавать сообщения, содержащие лишь слово online. Синапс этого нейрона, ведущий на нейрон «Не спам» имеет гораздо больший вес, чем синапс, идущий к нейрону «Спам», поэтому сообщение классифицируется как «Не спам». На рис. 12.6 показано, что происходит, когда слова online и pharmacy (аптека) подаются на вход сети вместе.

Так как нейроны первого уровня реагируют на отдельные слова, то активируются оба. На втором уровне картина интереснее. Наличие слова pharmacy (аптека) отрицательно влияет на нейрон только online, но оба нейрона первого уровня активируют средний нейрон, который был обучен реагировать на словосочетание online pharmacy (онлайновая аптека). Этот нейрон посылает очень сильный сигнал в категорию «Спам», поэтому документ классифицируется как спам. На этом примере продемонстрировано, как многоуровневая нейронная сеть легко справляется с признаками, которые в разных сочетаниях должны интерпретироваться по-разному.

Обучение нейронной сети

В рассмотренной выше нейронной сети веса синапсов уже были проставлены. Но истинная мощь нейронных сетей состоит в том, что они могут начать работу со случайного набора весов, а затем обучаться на предъявляемых примерах. Наиболее распространенным методом обучения многоуровневых перцептронов является метод обратного распространения (он был описан в главе 4).

Чтобы обучить сеть методом обратного распространения, вы начинаете с примера, содержащего только слово online и правильный ответ – в данном случае «Не спам». Этот пример передается нейронной сети, чтобы посмотреть, как она на него прореагирует.

В начальный момент сеть может решить, что ответ «Спам» несколько вероятнее, чем «Не спам»; это неправильно. Чтобы исправить ситуацию, сети сообщают, что результат для категории «Спам» должен быть ближе к 0, а для «Не спам» – ближе к 1. Веса синапсов, ведущих к нейрону «Спам», слегка уменьшаются пропорционально вкладу каждого, а ведущих к нейрону «Не спам» – увеличиваются. Веса синапсов между входным и скрытым уровнем также корректируются в соответствии с тем, какой вклад они вносят в важные узлы выходного уровня.

Формулы корректировки приведены в главе 4. Чтобы предотвратить перекомпенсацию в случае обучения на зашумленных или недостоверных данных, сеть обучают медленно; чем чаще она видела конкретный образец, тем лучше будет его классифицировать в будущем.

Использование ранее написанного кода

К этой задаче легко применить код, написанный в главе 4. Единственная хитрость заключается в том, что в этом коде вместо слов используются числовые идентификаторы, поэтому предварительно всем возможным входным сигналам надо будет сопоставить числа. Обучающие данные хранятся в базе, поэтому просто запускайте программу и начинайте обучение:

```
>>> import nn
>>> online,pharmacy=1,2
>>> spam,notspam=1,2
>>> possible=[spam,notspam]
>>> neuralnet=nn.searchnet('nntest.db')
>>> neuralnet.maketables( )
>>> neuralnet.trainquery([online],possible,notspam)
>>> neuralnet.trainquery([online,pharmacy],possible,spam)
>>> neuralnet.trainquery([pharmacy],possible,notspam)
>>> neuralnet.getresult([online,pharmacy],possible)
[0.7763, 0.2890]
>>> neuralnet.getresult([online],possible)
```

```
[0.4351, 0.1826]
>>> neuralnet.trainquery([online],possible,notspam)
>>> neuralnet.getresult([online],possible)
[0.3219, 0.5329]
>>> neuralnet.trainquery([online],possible,notspam)
>>> neuralnet.getresult([online],possible)
[0.2206, 0.6453]
```

Как видите, чем дольше сеть обучается, тем увереннее она выдает ответы. Она даже может пропустить случайный неправильный пример, сохранив хорошую способность к прогнозированию.

Сильные и слабые стороны

Главный плюс нейронных сетей заключается в том, что они способны справляться со сложными нелинейными функциями и вскрывать зависимости между различными входными данными. Хотя в примере мы подавали на вход только сигналы 1 или 0 (присутствует либо отсутствует), в принципе, годится любое число, и сеть может выдавать числовые оценки на выходе.

Нейронные сети также допускают инкрементное обучение и обычно не требуют много места для хранения обученных моделей, так как модель является просто списком чисел, представляющих веса синапсов. Хранить исходные данные для последующего обучения не нужно, а следовательно, нейронные сети пригодны для приложений с непрерывным потоком обучающих данных.

Основной недостаток нейронных сетей в том, что они являются «черным ящиком». Для рассмотренного примера была взята искусственно упрощенная ситуация, когда следить за процессом принятия решения было очень легко, но в реальных сетях имеются сотни нейронов и тысячи синапсов, поэтому понять, как сеть выработала ответ, невозможно. Невозможность ознакомиться с процессом рассуждения для некоторых приложений неприемлема.

Еще один недостаток – отсутствие твердых правил по выбору скорости обучения и размера сети для решения конкретной задачи. Тут необходимо экспериментировать. Если скорость обучения слишком высока, то сеть станет делать излишне общие выводы на основе зашумленных данных, а если слишком мала, то сеть может вообще никогда не обучиться на предъявляемых данных.

Метод опорных векторов

В главе 9 были рассмотрены машины опорных векторов (SVM). Пожалуй, это самый сложный метод классификации из всех описанных в книге. SVM принимает набор данных, состоящий из чисел, и пытается спрогнозировать, в какую категорию он попадает. Можно, например, определить роль игрока в баскетбольной команде по его росту

и скорости бега. Для простоты рассмотрим всего две возможности: позиция в нападении, для которой требуется высокий рост, и в защите, где игрок должен быстро перемещаться.

SVM строит прогностическую модель, отыскивая линию, разделяющую две категории. Если отложить по одной оси рост, а по другой – скорость и нанести наилучшие позиции для каждого игрока, то получится диаграмма, изображенная на рис. 12.7. Нападающие представлены крестиками, а защитники – кружками. Также на диаграмме показано несколько линий, разделяющих данные на две категории.

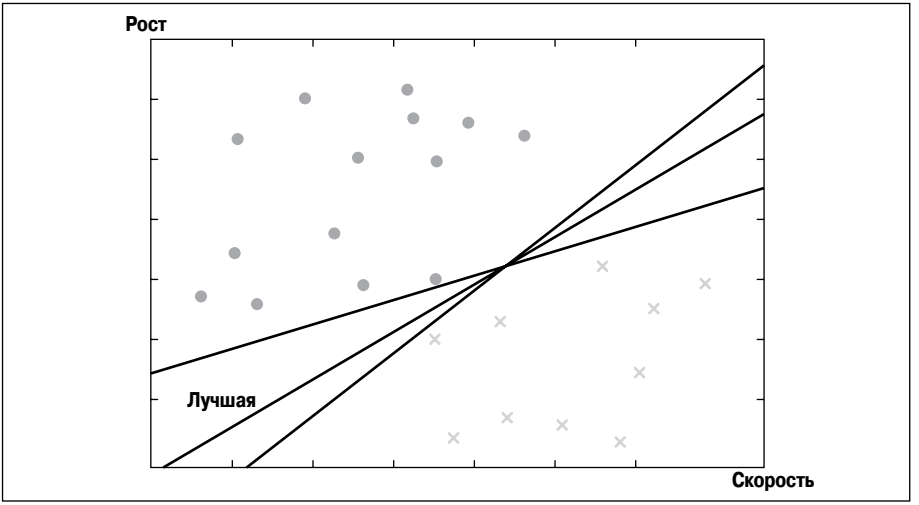


Рис. 12.7. *Диаграмма бескетболистов и разделяющие линии*

Машина опорных векторов находит линию, наилучшим образом разделяющую данные. Это означает, что она проходит на максимальном расстоянии от расположенных вблизи нее точек. На рис. 12.7 разделяющих линий несколько, но наилучшая из них помечена надписью «Лучшая». Для определения того, где должна пройти линия, необходимы лишь ближайшие к ней точки, они и называются *опорными векторами*.

После того как разделяющая линия найдена, для классификации новых образцов требуется лишь нанести их на диаграмму и посмотреть, по какую сторону от разделителя они окажутся. Просматривать обучающие данные при классификации новых образцов ни к чему, поэтому классификация производится очень быстро.

Переход к ядру

Машины опорных векторов, равно как и другие линейные классификаторы, основанные на использовании *скалярного произведения*, часто можно улучшить за счет применения техники *перехода к ядру* (kernel

trick). Чтобы понять ее суть, рассмотрим, как изменилась бы задача, если бы нужно было прогнозировать не позицию, а пригодность игрока для любительской команды, в которой позиции часто меняются. Эта задача более интересна, так как разделение нелинейное. Вам не нужны ни слишком высокие, ни слишком быстрые игроки, поскольку другим было бы трудно играть с ними, но совсем уж низенькие или медлительные тоже ни к чему. На рис. 12.8 показано, как могла бы выглядеть ситуация; здесь кружок означает, что игрок подходит, а крестик – что не подходит.

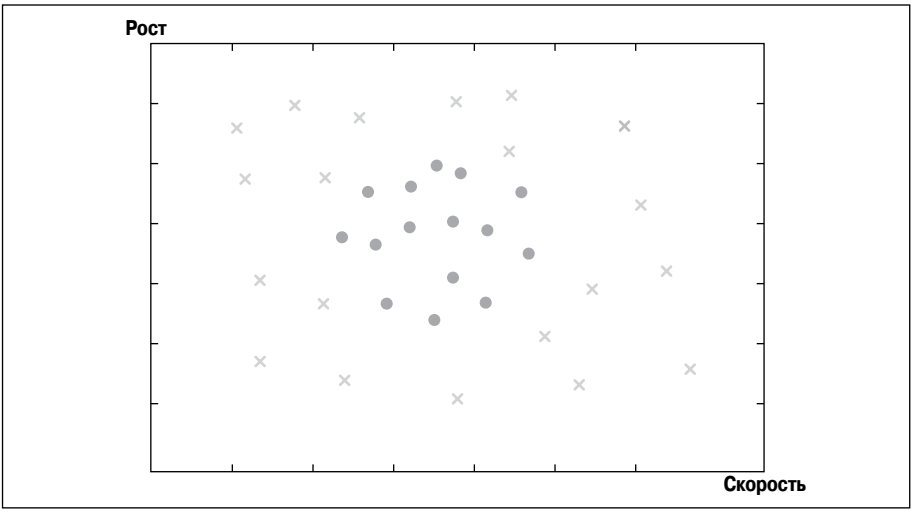


Рис. 12.8. *Диаграмма баскетболистов для любительской команды*

Прямой разделяющей линии здесь не существует, поэтому применить линейный классификатор не удастся, если предварительно не выполнить то или иное преобразование данных. Часто прибегают к переходу в другое пространство, возможно, с большим числом измерений, применяя различные функции к переменным. В данном случае новое пространство можно создать путем вычитания из роста и скорости средних значений и последующего возведения в квадрат. Результат показан на рис. 12.9.

Этот прием называется *полиномиальной трансформацией*, в результате данные представляются в новых осях. Как легко видеть, теперь подходящих и неподходящих игроков нетрудно разделить прямой линией, которую можно найти с помощью линейного классификатора. Для классификации новой точки ее следует трансформировать и посмотреть, по какую сторону от разделителя она окажется.

В данном примере такая трансформация работает, но во многих случаях для нахождения разделяющей линии необходимо переходить к гораздо более сложным пространствам. Некоторые из них имеют тысячи, а то и бесконечное число измерений, поэтому выполнить трансформацию

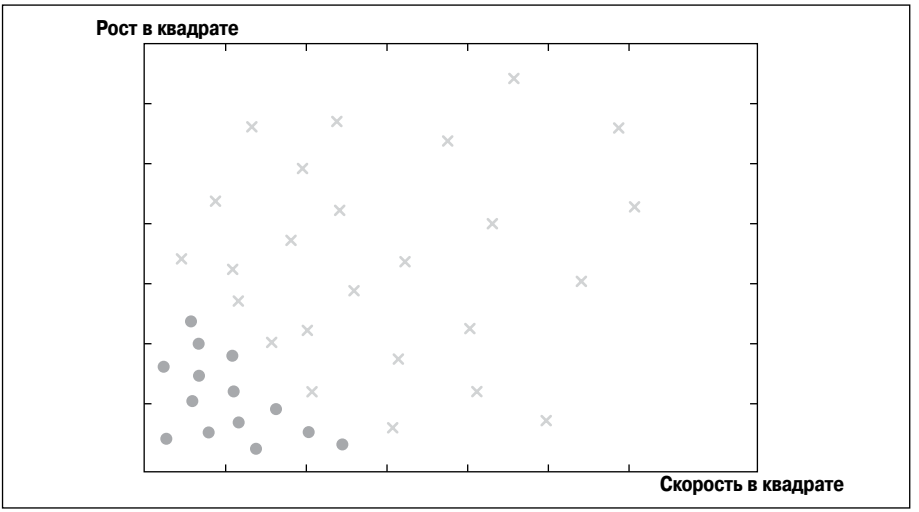


Рис. 12.9. *Баскетболисты в полиномиальном пространстве*

не всегда возможно. Тут-то и приходит на помощь переход к ядру – вместо того чтобы трансформировать пространство, мы заменяем настоящее скалярное произведение функцией, возвращающей значение, которому скалярное произведение *было бы* равно после трансформации. Например, вместо выполнения описанной выше полиномиальной трансформации можно было заменить

```
dotproduct(A,B)
```

на

```
dotproduct(A,B)**2
```

В главе 9 мы построили простой линейный классификатор, в котором использовались групповые средние. Вы видели, как можно его изменить, подставив вместо скалярного произведения другие функции над векторами. В результате удалось применить его к нелинейным задачам.

Использование библиотеки LIBSVM

В главе 9 мы ознакомились с библиотекой LIBSVM. С ее помощью можно обучить классификатор на некотором наборе данных (чтобы он находил разделяющую линию в трансформированном пространстве), а затем классифицировать новые наблюдения:

```
>>> from random import randint
>>> # Создать 200 случайных точек
>>> d1=[[randint(-20,20),randint(-20,20)] for i in range(200)]
>>> # Классифицировать точку как 1, если она лежит внутри круга, и как 0 -
>>> # если вне
>>> result=[(x**2+y**2)<144 and 1 or 0 for (x,y) in d1]
>>> from svm import *
```

```
>>> prob=svm_problem(result,d1)
>>> param=svm_parameter(kernel_type=RBF)
>>> m=svm_model(prob,param)
>>> m.predict([2,2])
1.0
>>> m.predict([14,13])
0.0
>>> m.predict([-18,0])
0.0
```

Библиотека LIBSVM поддерживает много ядерных функций, и с ее помощью легко опробовать их все с различными параметрами, чтобы выяснить, какая лучше всего подходит для имеющегося набора данных. Чтобы протестировать модель, можно воспользоваться функцией `cross_validation`, которая принимает параметр `n` и разбивает множество на *n* подмножеств. Затем каждое подмножество поочередно считается тестовым, а обучение производится на остальных подмножествах. Функция возвращает список ответов, который можно сравнить с исходным списком:

```
>>> guesses=cross_validation(prob,param,4)
>>> sum([abs(guesses[i]-result[i]) for i in range(len(guesses))])
28.0
```

Сталкиваясь с новой задачей, вы можете опробовать различные ядерные функции с разными параметрами и посмотреть, какая комбинация окажется наилучшей. Результат зависит от набора данных, но, определившись с параметрами, вы сможете создать модель, пригодную для классификации новых наблюдений. На практике бывает полезно написать несколько вложенных циклов, в которых проверяются новые значения и запоминается, при каких параметрах получен самый лучший результат.

Сильные и слабые стороны

Машина опорных векторов – очень мощный классификатор; после подбора правильных параметров она работает не хуже, а часто даже лучше любого другого из рассмотренных в книге классификаторов. Кроме того, после обучения классификация новых результатов производится очень быстро, поскольку требуется лишь определить, по какую сторону от разделителя оказалась точка. Преобразовав дискретные величины в числа, вы сможете заставить SVM работать с дискретными и числовыми данными одновременно.

Недостаток заключается в том, что оптимальная ядерная функция и ее параметры зависят от конкретного набора данных, так что всякий раз приходится подбирать их заново. Перебор возможных значений в цикле отчасти решает проблему, но для этого требуется иметь достаточно большой набор данных, чтобы результатам перекрестной проверки можно было доверять. В общем случае метод опорных векторов лучше

приспособлен для таких задач, где доступен большой объем данных, тогда как другие методы, скажем деревья решений, дают интересную информацию уже на весьма скромных наборах данных.

Как и нейронные сети, SVM представляет собой «черный ящик»; понять ход рассуждений здесь даже сложнее из-за трансформации в многомерное пространство. SVM может дать правильный ответ, но вы никогда не узнаете, как он получен.

k-ближайшие соседи

В главе 8 рассматривалась тема числового прогнозирования с помощью алгоритма *k*-ближайших соседей (kNN). С его помощью были построены модели прогнозирования цен. Алгоритм рекомендования в главе 2, который прогнозировал, понравится ли данному человеку некий фильм или ссылка, тоже был основан на упрощенном варианте kNN.

Алгоритм kNN, получая новый образец, для которого вы хотите иметь прогноз числового значения, сравнивает его с образцами, значения которых уже известны. Он ищет образцы, максимально похожие на вновь предъявленный, и усредняет их значения. В табл. 12.6 приведен перечень цифровых камер, для каждой из которых указаны разрешение в мегапикселях, максимальное увеличение фокусного расстояния (зум) и цена.

Таблица 12.6. Цены на цифровые камеры

Камера	Мегапиксели	Зум	Цена
C1	7,1	3,8×	$399
C2	5,0	2,4×	$299
C3	6,0	4,0×	$349
C4	6,0	12,0×	$399
C5	10,0	3×	$449

Предположим, что вы хотите узнать цену новой камеры с шестью мегапикселями и объективом, имеющим зум 6×. Первым делом нужно найти способ измерить степень подобия двух образцов. В главе 8 мы пользовались евклидовым расстоянием, но в книге были описаны и другие метрики, в частности коэффициент корреляции Пирсона и коэффициент Танимото. В терминах евклидова расстояния ближайшей камерой будет C3. Для визуализации результата нанесите образцы на диаграмму, где по оси x отложены мегапиксели, а по оси y – зум. Сами образцы представлены на диаграмме своими ценами (рис. 12.10).

Можно было бы в качестве ответа просто взять цену $349 (это ведь ближайшее соответствие, правда?), но вдруг эта цена – аномалия? Поэтому лучше взять несколько самых похожих камер и усреднить их цены.

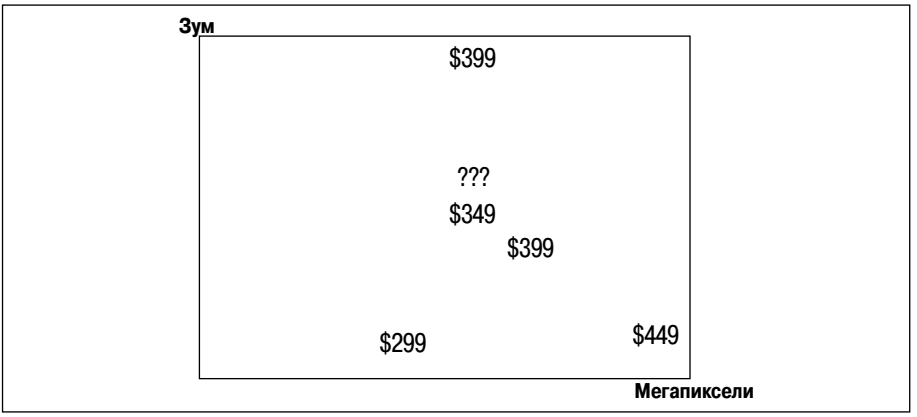


Рис. 12.10. *Цены камер в пространстве зум–мегапиксели*

Параметр k в методе k-ближайших соседей – это количество усредняемых образцов. Например, проводя усреднение по трем самым похожим камерам, вы применяете алгоритм kNN с $k = 3$.

Вместо прямолинейного усреднения можно вычислить средневзвешенное с учетом того, насколько далеко отстоят образцы. Чем больше расстояние, тем меньше вес. В главе 8 были рассмотрены различные функции вычисления весов. В данном примере цене $349 можно приписать наибольший вес, а двум ценам $399 – веса поменьше. Например:

$$\text{Цена} = 0{,}5 \times 349 + 0{,}25 \times 399 + 0{,}25 \times 399 = 374$$

Масштабирование и лишние переменные

У описанного выше варианта алгоритма kNN есть существенный недостаток – он вычисляет расстояние по всем переменным. Но если переменные измеряют несопоставимые характеристики и одна из них оказывается заметно больше прочих, то она будет несоразмерно сильно влиять на понятие «близости». Представьте, что в рассмотренном наборе разрешение измеряется в пикселях, а не в мегапикселях. Понятно, что разница в зуме на 10 единиц гораздо более важна, чем разница в разрешении на 10 пикселей, но алгоритм трактует их одинаково. Бывают также ситуации, когда некоторые переменные абсолютно бесполезны для прогнозирования, но они тоже вносят вклад в прогноз.

Эту проблему можно решить масштабированием данных до вычисления расстояний. В главе 8 был написан метод для масштабирования, подразумевающий увеличение амплитуды одних переменных и уменьшение – других. Бесполезные переменные можно умножить на 0, после чего они перестают влиять на результат. Ценные переменные, принимающие значения из существенно различающихся диапазонов, можно привести к общему масштабу, например, решив, что разница в 2000 пикселей эквивалентна разнице в 1 единицу зума.

Поскольку выбор масштабных коэффициентов зависит от приложения, можно подобрать подходящий набор с помощью перекрестного контроля алгоритма прогнозирования. Смысл этой методики в том, что мы изымаем из набора данных часть образцов, а затем смотрим, насколько хорошо алгоритм сможет спрогнозировать их значения, пользуясь только оставшимися образцами. Принцип работы перекрестного контроля иллюстрируется на рис. 12.11.

Подвергая контролю разные коэффициенты масштабирования, мы вычисляем для каждого образца расхождение и затем решаем, какие коэффициенты лучше использовать для прогнозирования значений новых образцов.

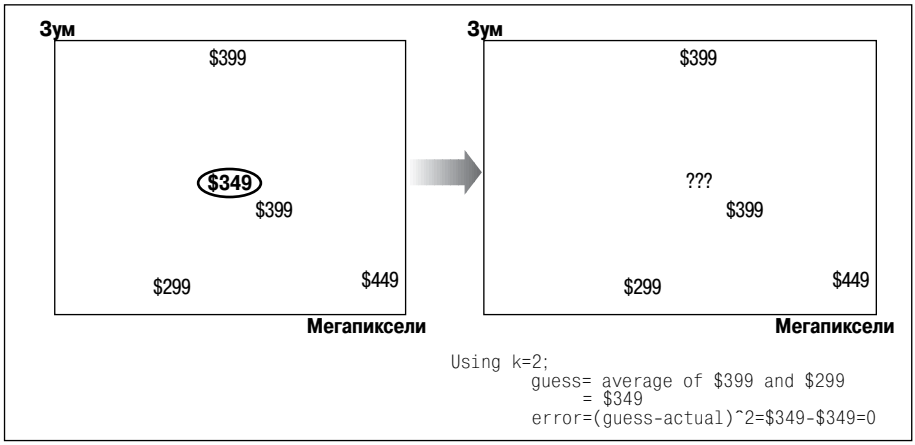


Рис. 12.11. *Перекрестный контроль с одним изъятым образцом*

Использование ранее написанного кода

В главе 8 были написаны функции, реализующие простой и взвешенный алгоритм kNN. Их легко применить к набору данных, показанному в табл. 12.6.

```
>>> cameras=[{'input':(7.1,3.8),'result':399},
... {'input':(5.0,2.4),'result':299},
... {'input':(6.0,4.0),'result':349},
... {'input':(6.0,12.0),'result':399},
... {'input':(10.0,3.0),'result':449}]
>>> import numpredict
>>> numpredict(cameras,(6.0,6.0),k=2)
374.0
>>> numpredict.weightedknn(cameras,(6.0,6.0),k=3)
351.52666892719458
```

Возможно, результат удастся улучшить путем масштабирования данных. Для этого служит функция `rescale`:

```
>>> scc=numpredict.rescale(cameras,(1,2))
>>> scc
```

```
[{'input': [7.1, 7.6], 'result': 399}, {'input': [5.0, 4.8], 'result': 299},
{'input': [6.0, 8.0], 'result': 349}, {'input': [6.0, 24.0], 'result': 399},
{'input': [10.0, 6.0], 'result': 449}]
```

А с помощью функции `crossvalidate` вы сможете найти наилучший коэффициент масштабирования:

```
>>> numpredict.crossvalidate(knn1,cameras,test=0.3,trials=2)
3750.0
>>> numpredict.crossvalidate(knn1,scc,test=0.3,trials=2)
2500.0
```

По мере увеличения количества переменных в наборе данных становится все труднее угадывать возможные коэффициенты масштабирования, поэтому можно либо организовать цикл по всем различным значениям, либо воспользоваться одним из алгоритмов оптимизации, как показано в главе 8.

Сильные и слабые стороны

Алгоритм k-ближайших соседей – это один из немногих алгоритмов, способных прогнозировать числовые значения сложных функций, не утрачивая при этом простоты интерпретации. Процесс рассуждения легко понять, а слегка изменив код, вы сможете увидеть, какие соседи принимали участие в вычислениях. Нейронные сети тоже способны давать прогнозы числовых значений сложных функций, но, безусловно, не могут показать вам похожие образцы, чтобы вы поняли, как был получен результат.

Далее, процедура определения подходящих коэффициентов масштабирования позволяет не только улучшить качество прогноза, но и подсказать, какие переменные существенны для прогнозирования. Переменные, для которых коэффициент оказался равен 0, можно не принимать во внимание. В некоторых случаях данные собирать трудно, поэтому информация о том, что они бесполезны, поможет сэкономить время и деньги в будущем.

Алгоритм kNN относится к числу *оперативных методов*, то есть данные можно добавлять в любой момент, в отличие, скажем, от машины опорных векторов, которую требуется переучивать при каждом изменении данных. Более того, при добавлении новых данных не нужны вообще никакие вычисления; достаточно просто включить данные в набор.

Основной недостаток kNN заключается в том, что для прогнозирования ему требуются все данные, на которых производилось обучение. Если в наборе миллионы образцов, то на это расходуется не только память, но и время – для выработки каждого прогноза приходится сравнивать новый образец с каждым из имеющихся, чтобы найти ближайшие. Для некоторых приложений это недопустимо медленно.

Еще один недостаток – утомительность поиска подходящих коэффициентов масштабирования. Хотя существуют способы автоматизации

этой процедуры, но на большом наборе данных для оценки тысяч возможных коэффициентов и перекрестного контроля могут понадобиться немалые вычислительные ресурсы. Если приходится опробовать много разных переменных, то для нахождения подходящего сочетания масштабных коэффициентов, возможно, потребуется пересмотреть миллионы комбинаций.

Кластеризация

Иерархическая кластеризация и кластеризация методом K-средних – это методы обучения без учителя, то есть для выработки прогнозов им не требуется предварительно предъявлять примеры. В главе 3 мы видели, как, имея список наиболее популярных блогов, автоматически кластеризовать их и выявить группы блогеров, пишущих на схожие темы или похожими словами.

Иерархическая кластеризация

Кластеризация применима к любому набору образцов, имеющих одно или несколько числовых свойств. В примере из главы 3 свойствами блогов были счетчики слов, но, в принципе, для кластеризации подойдет любой набор чисел. Для демонстрации работы алгоритма кластеризации рассмотрим простую таблицу образцов (какие-то буквы алфавита) и некоторые числовые свойства (табл. 12.7).

Таблица 12.7. Простая таблица для кластеризации

Образец	***P*1**	***P*2**
A	1	8
B	3	8
C	2	6
D	1,5	1
E	4	2

На рис. 12.12 представлен процесс кластеризации этих образцов. Образцы представлены на двумерной диаграмме, где по оси x отложено свойство $P1$, а по оси y – свойство $P2$. Сначала находятся два самых близких образца и объединяются в кластер (первый рисунок). На втором рисунке мы видим, что образцы A и B попали в одну группу. Координатами этого кластера считаются усредненные координаты включенных в него образцов. На следующем рисунке ближайшими оказываются образец C и кластер A–B. Процесс продолжается, пока не останется один большой кластер.

В результате создается иерархия, которую можно визуализировать в виде *дендрограммы* – древовидной структуры, из которой видно, какие

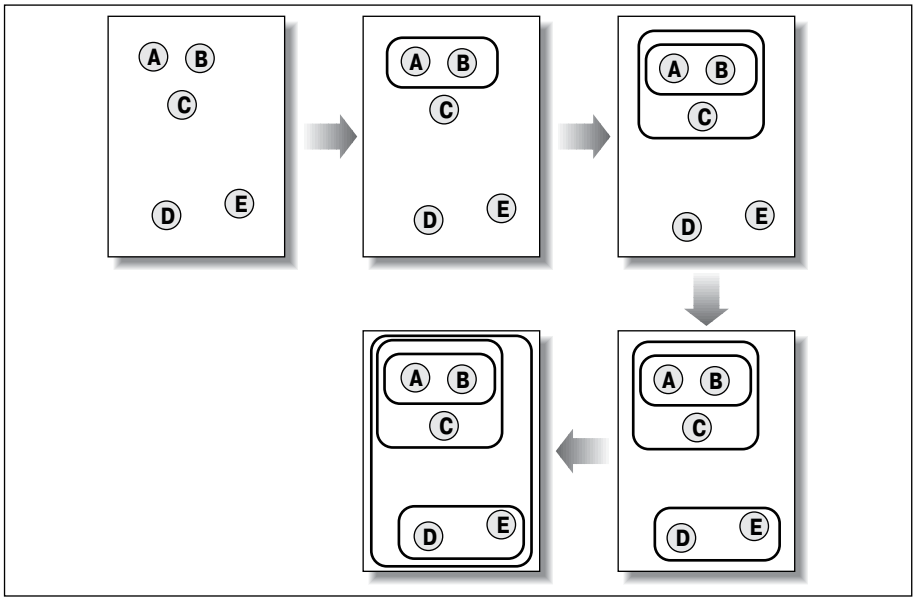


Рис. 12.12. *Процесс иерархической кластеризации*

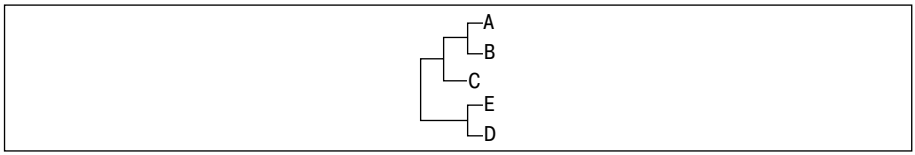


Рис. 12.13. *Дендрограмма кластеризованных букв*

образцы и группы были сочтены ближайшими. Дендрограмма рассматриваемого набора данных изображена на рис. 12.13.

Два ближайших элемента A и B соединены между собой. Затем к комбинации A и B присоединен образец C. Вы можете выбрать любую точку ветвления на дендрограмме и решить, представляет ли данная группа интерес. В главе 3 мы видели ветви, состоящие почти исключительно из политических блогов, ветви, объединяющие технологические блоги, и т. д.

Кластеризация методом K-средних

Еще один метод кластеризации – это метод *K*-средних. Если при иерархической кластеризации создается дерево образцов, то метод *K*-средних разбивает данные на отдельные группы. Но перед началом работы алгоритма вы должны задать количество желаемых групп. На рис. 12.14 иллюстрируется кластеризация методом *K*-средних в действии. Здесь мы пытаемся найти два кластера в несколько ином наборе данных.

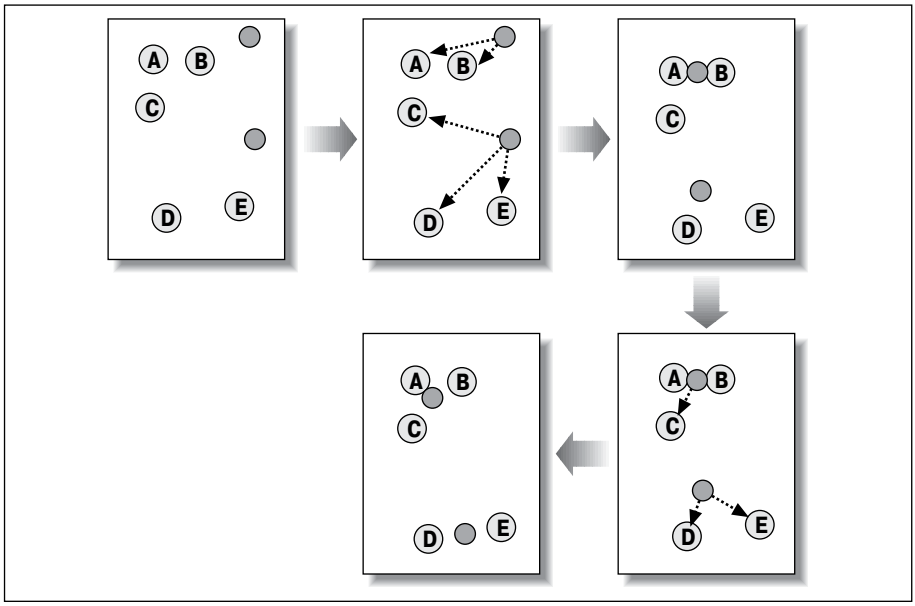


Рис. 12.14. *Процесс кластеризации методом K-средних*

На первом рисунке случайным образом расположены два центроида (темные кружочки). На втором рисунке каждому образцу приписывается ближайший к нему центроид – в данном случае A и B приписаны верхнему центроиду, а C, D и E – нижнему. На третьем рисунке центроиды перемещены в среднюю точку приписанных к ним образцов. Далее распределение образцов по центроидам производится снова и обнаруживается, что C теперь оказался ближе к верхнему центроиду, а D и E остались ближе к нижнему. В результате A, B и C образуют один кластер, а D и E – другой.

Использование ранее написанного кода

Для выполнения кластеризации необходим набор данных и метрика. Набор данных представляет собой список чисел, в котором каждое число соответствует некоторой переменной. В главе 3 в качестве метрики мы применяли коэффициент корреляции Пирсона и коэффициент Танимото, но годятся и другие метрики, например евклидово расстояние:

```
>>> data=[[1.0,8.0],[3.0,8.0],[2.0,7.0],[1.5,1.0],[4.0,2.0]]
>>> labels=['A','B','C','D','E']
>>> def euclidean(v1,v2): return sum([(v1[i]-v2[i])**2 for i in range(len(v1))])
>>> import clusters
>>> hcl=clusters.hcluster(data,distance=euclidean)
>>> kcl=clusters.kcluster(data,distance=euclidean,k=2)
Итерация 0
Итерация 1
```

В случае кластеризации методом *K*-средних можно напечатать, какие образцы в какой кластер попали:

```
>>> kcl
[[0, 1, 2], [3, 4]]
>>> for c in kcl: print [labels[l] for l in c]
...
['A', 'B', 'C']
['D', 'E']
```

Для иерархической кластеризации очевидного способа распечатки не существует, но в главе 3 был приведен код функции для рисования дендрограммы иерархических кластеров:

```
>>> clusters.drawdendrogram(hcl,labels,jpeg='hcl.jpg')
```

Выбор алгоритма зависит от того, что вы хотите сделать. Часто бывает полезно разбить данные на отдельные группы, поскольку так их проще увидеть и охарактеризовать. В этом случае подойдет кластеризация методом *K*-средних. С другой стороны, имея совершенно новый набор данных, вы зачастую не знаете, сколько в нем групп, и хотите посмотреть, какие получаются группы похожих образцов. В таком случае лучше остановиться на алгоритме иерархической кластеризации.

Можно взять и лучшее из обоих методов: сначала методом *K*-средних создать набор групп, а затем выполнить для них иерархическую кластеризацию, взяв в качестве метрики расстояние между центроидами. Тогда вы получите на одном уровне отдельные группы, которые организованы в дерево, чтобы было видно, как группы связаны между собой.

Многомерное шкалирование

В главе 3 к анализу блогов был применен также метод многомерного шкалирования. Как и кластеризация, это метод обучения без учителя. Он предназначен не для прогнозирования, а чтобы понять, как различные образцы связаны между собой. Алгоритм создает представление набора данных в пространстве меньшей размерности, стараясь по возможности сохранить исходные расстояния между элементами. Если речь идет о представлении на экране или на бумаге, то многомерный набор представляется в двумерном пространстве.

Возьмем, к примеру, четырехмерный набор данных, показанный в табл. 12.8 (каждый образец характеризуется четырьмя переменными).

Таблица 12.8. Простой четырехмерный набор для шкалирования

A	0,5	0,0	0,3	0,1
B	0,4	0,15	0,2	0,1
C	0,2	0,4	0,7	0,8
D	1,0	0,3	0,6	0,0

Вычисляем евклидово расстояние между каждой парой образцов. Например, расстояние между A и B равно sqrt(0,1 ^ 2 + 0,15 ^ 2 + 0,1 ^ 2 + + 0,0 ^ 2) = 0,2. Полная матрица попарных расстояний приведена в табл. 12.9.

Таблица 12.9. Матрица расстояний

	A	B	C	D
A	0,0	0,2	0,9	0,8
B	0,2	0,0	0,9	0,7
C	0,9	0,9	0,0	1,1
D	0,8	0,7	1,1	0,0

Наша цель – нарисовать все образцы на двумерной диаграмме так, чтобы расстояния между их «проекциями» были как можно ближе к исходным расстояниям в четырехмерном пространстве. Сначала все образцы наносятся на диаграмму в случайном порядке и вычисляются текущие попарные расстояния (рис. 12.15).

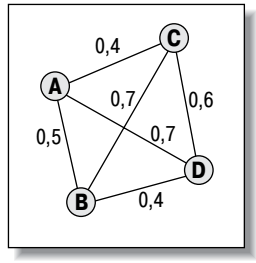


Рис. 12.15. *Расстояния между образцами*

Для каждой пары образцов целевое расстояние сравнивается с текущим и вычисляется расхождение. Каждый образец немного приближается к своему соседу или отодвигается от него на расстояние, пропорциональное расхождению между ними. На рис. 12.16 показаны

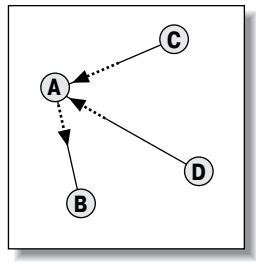


Рис. 12.16. *Силы, действующие на образец A*

силы, действующие на образец A. Расстояние между A и B на диаграмме равно 0,5, а целевое расстояние между ними составляет 0,2, поэтому A перемещается ближе к B. В то же время A отталкивается образцами C и D, потому что он оказался слишком близко.

Каждый узел перемещается в соответствии с комбинацией всех других узлов, притягивающих и отталкивающих его. Каждый раз, когда это происходит, разница между текущим и целевым расстоянием слегка уменьшается. Эта процедура повторяется многократно до тех пор, пока общее количество ошибок больше нельзя будет уменьшить путем дальнейшего перемещения образцов.

Использование ранее написанного кода

В главе 3 были написаны две функции для многомерного шкалирования – одна реализует сам алгоритм, а другая отображает результаты. Первая функция, `scaledown`, принимает список точек в многомерном пространстве и возвращает соответствующий список точек на двумерной плоскости в том же порядке:

```
>>> labels=['A','B','C','D']
>>> scaleset=[[0.5,0.0,0.3,0.1],
... [0.4,0.15,0.2,0.1],
... [0.2,0.4,0.7,0.8],
... [1.0,0.3,0.6,0.0]]
>>> twod=clusters.scaledown(scaleset,distance=euclidean)
>>> twod
[[0.45, 0.54],
[0.40, 0.54],
[-0.30, 1.02],
[0.92, 0.59]]
```

Вторая функция, `draw2d`, принимает шкалированный список и создает изображение:

```
>>> clusters.draw2d(twod,labels,jpeg='abcd.jpg')
```

В результате создается файл abcd.jpg. Визуализировать список можно и другими способами, например загрузив данные, возвращенные `scaledown`, в какую-нибудь подходящую программу, скажем в электронную таблицу.

Неотрицательная матричная факторизация

В главе 10 рассматривался алгоритм неотрицательной матричной факторизации (NMF), который разбивает набор числовых наблюдений на компоненты. Этот метод был применен к задаче тематической классификации новостей и к задаче обнаружения событий, повлиявших на объемы торгов отдельными акциями или группами акций. Алгоритм не нуждается в учителе, поскольку применяется для того, чтобы охарактеризовать данные, а не для прогнозирования значений или принадлежности к категории.

Чтобы понять, как работает алгоритм NMF, рассмотрим набор значений в табл. 12.10.

Таблица 12.10. Простые данные для алгоритма NMF

Номер наблюдения	A	B
1	29	29
2	43	33
3	15	25
4	40	28
5	24	11
6	29	29
7	37	23
8	21	6

Предположим, что наблюдения A и B – это некоторые комбинации двух пар чисел (признаков), но вы не знаете, что это за пары и каков вклад каждой пары (вес) в каждом наблюдении. Алгоритм NMF способен отыскать возможные значения признаков и весов. В примере с новостями из главы 10 наблюдения представляли собой новости, а столбцы – слова, встречающиеся в текстах новостей. В примере с объемом торгов акциями в качестве наблюдений брались даты, а в качестве столбцов – символы различных акций. В обоих случаях алгоритм пытался найти как можно меньшее количество частей, скомбинировав которые с различными весами, можно получить исходные наблюдения.

Одно из возможных решений для приведенной выше таблицы – пары (3,5) и (7,2). Из этих частей наблюдения можно реконструировать следующим образом:

$$5 \times (3, 5) + 2 \times (7, 2) = (29, 29)$$

$$5 \times (3, 5) + 4 \times (7, 2) = (43, 33)$$

Эти формулы можно рассматривать как умножение матриц, показанное на рис. 12.17.

$$\begin{pmatrix} 5 & 2 \\ 5 & 4 \\ 5 & 0 \\ 4 & 4 \\ 1 & 3 \\ 5 & 2 \\ 3 & 4 \\ 0 & 3 \end{pmatrix} \times \begin{pmatrix} 3 & 5 \\ 7 & 2 \end{pmatrix} = \begin{pmatrix} 29 & 29 \\ 43 & 33 \\ 15 & 25 \\ 40 & 28 \\ 24 & 11 \\ 29 & 29 \\ 37 & 23 \\ 21 & 6 \end{pmatrix}$$

Веса *Признаки* *Набор данных*

Рис. 12.17. *Факторизация набора данных на признаки и веса*

Цель алгоритма NMF заключается в том, чтобы автоматически найти матрицы признаков и весов. Для этого он начинает с матриц, выбранных случайным образом, и модифицирует их по *правилам обновления*. Согласно этим правилам генерируются четыре новые матрицы. В описании исходная матрица называется матрицей данных:

`hn`

Произведение транспонированной матрицы весов и матрицы данных.

`hd`

Произведение транспонированной матрицы весов, самой матрицы весов и матрицы признаков.

`wn`

Произведение матрицы данных и транспонированной матрицы признаков.

`wd`

Произведение матрицы весов, матрицы признаков и транспонированной матрицы признаков.

Для обновления матриц признаков и весов все эти матрицы преобразуются в массивы. Каждый элемент матрицы признаков умножается на соответственный элемент `hn` и делится на соответственный элемент `hd`. Аналогично, каждый элемент матрицы весов умножается на соответственный элемент `wn` и делится на элемент `wd`.

Процесс повторяется, пока произведение матриц весов и признаков не окажется достаточно близко к матрице данных. Зная матрицу признаков, можно сказать, какие факторы – например, темы новостей или события на фондовом рынке – в различных сочетаниях привели к образованию именно такого набора данных.

Использование ранее написанного кода

Для применения алгоритма NMF достаточно вызвать функцию `factorize`, передав ей список наблюдений и количество искомых признаков:

```
>>> from numpy import *
>>> import nmf
>>> data=matrix([[ 29., 29.],
... [ 43., 33.],
... [ 15., 25.],
... [ 40., 28.],
... [ 24., 11.],
... [ 29., 29.],
... [ 37., 23.],
... [ 21., 6.]])
>>> weights,features=nmf.factorize(data,pc=2)
>>> weights
matrix([[ 0.64897525, 0.75470755],
[ 0.98192453, 0.80792914],
```

```
[ 0.31602596, 0.70148596],
[ 0.91871934, 0.66763194],
[ 0.56262912, 0.22012957],
[ 0.64897525, 0.75470755],
[ 0.85551414, 0.52733475],
[ 0.49942392, 0.07983238]])
>>> features
matrix([[ 41.62815416, 6.80725866],
[ 2.62930778, 32.57189835]])
```

Возвращаются веса и признаки. При разных прогонах могут получаться различные результаты, потому что для небольшого набора наблюдений возможно несколько разных наборов признаков. Чем больше набор наблюдений, тем вероятнее, что результаты будут примерно одинаковы, хотя признаки могут возвращаться в разном порядке.

Оптимизация

Рассмотренные в главе 5 методы оптимизации несколько отличаются от всех прочих; они не столько работают с набором данных, сколько пытаются найти значения, минимизирующие целевую функцию. В главе 5 приведены примеры нескольких задач оптимизации: планирование группового путешествия (целевой функцией была комбинация цены билета и времени ожидания в аэропорту), распределение студентов по комнатам в общежитии и рисование графа с наименьшим числом пересечений. Было описано два подхода к решению: имитация отжига и генетические алгоритмы.

Целевая функция

Целевой функцией называется любая функция, принимающая гипотетическое решение и возвращающая число, которое больше для плохих решений и меньше для хороших. Алгоритмы оптимизации вызывают эту функцию для проверки решений и отыскания наилучшего. Целевая функция может учитывать много переменных, и не всегда ясно, какую из них надо изменить для улучшения результата. Рассмотрим, к примеру, такую функцию от одной переменной:

$$y = 1/x \times \sin(x)$$

Ее график изображен на рис. 12.18.

Поскольку это функция всего от одной переменной, то на графике сразу видно, где находится ее минимум. Мы привели ее лишь в качестве иллюстрации принципов работы алгоритмов оптимизации; на практике нарисовать график сложной функции многих переменных, чтобы найти минимум, не удастся.

Эта функция интересна тем, что у нее несколько локальных минимумов. Это точки, в которых значение функции меньше, чем в ближайшей окрестности, но необязательно является абсолютно наименьшим.

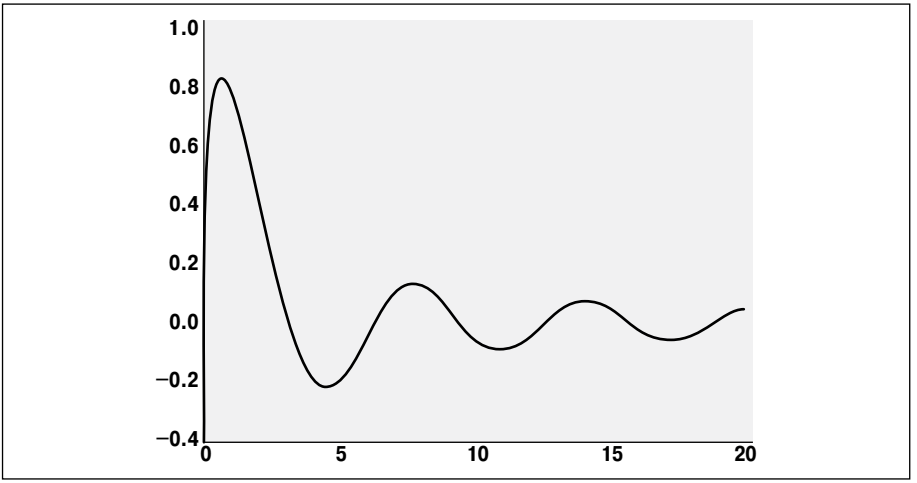


Рис. 12.18. *График функции* $1/x \times sin(x)$

Следовательно, задачу не всегда можно решить, просто взяв произвольное решение и спускаясь по склону, потому что при таком подходе мы можем «застрять» в локальном минимуме и никогда не найти глобального.

Имитация отжига

Алгоритм имитации отжига, навеянный физическим процессом охлаждения сплавов, начинает работу с выбора случайного решения. Далее он пытается улучшить решение, вычисляя стоимость похожего решения, отстоящего от текущего на небольшое расстояние в случайном направлении. Если стоимость нового решения выше, оно становится текущим с некоторой вероятностью, зависящей от текущей *температуры*. Начальное значение температуры выбирается большим и медленно уменьшается, так что на ранних этапах работы алгоритм с большей вероятностью будет соглашаться на ухудшение решения, чтобы избежать скатывания в локальный минимум.

Когда температура становится равна 0, алгоритм возвращает найденное к этому моменту решение.

Генетические алгоритмы

Генетические алгоритмы навеяны теорией эволюции. В начале работы выбирается несколько случайных решений, образующих популяцию. Затем отбираются сильнейшие члены популяции – имеющие наименьшую стоимость – и модифицируются путем внесения небольших изменений (мутация) или комбинирования характеристик (скрещивание). В результате создается новая популяция, которая называется следующим поколением. После смены нескольких поколений решения улучшаются.

Процесс прекращается по достижении некоторого порога: либо когда на протяжении нескольких поколений не наблюдалось заметного улучшения популяции, либо после смены максимального числа поколений. Алгоритм возвращает наилучшее среди всех поколений решение.

Использование ранее написанного кода

Для обоих алгоритмов необходимо знать целевую функцию и область определения решения. Областью определения называется возможный диапазон изменения каждой переменной. В нашем простом примере можно взять область определения `[(0,20)]`, то есть единственная переменная будет изменяться от 0 до 20. Теперь можно вызвать любой метод оптимизации, передав ему в качестве параметров целевую функцию `cost` и область определения `domain`:

```
>>> import math
>>> def costf(x): return (1.0/(x[0]+0.1))*math.sin(x[0])
>>> domain=[(0,20)]
>>> optimization.annealingoptimize(domain,costf)
[5]
```

Для любой задачи оптимизацию, возможно, придется провести несколько раз, чтобы подобрать параметры и добиться приемлемого компромисса между временем работы и качеством решения. При построении оптимизатора для решения схожих задач – например, планирования путешествий, когда цель остается неизменной, а меняются лишь детали (время в полете и цены), – можно один раз подобрать хорошие параметры, а затем пользоваться ими для всех похожих задач.

Существует масса возможностей сочетать машинное обучение, открытые API и открытое сотрудничество. А в будущем по мере совершенствования алгоритмов, появления новых API и увеличения числа активных участников сообщества возможности будут только расширяться. Надеюсь, что, прочитав эту книгу, вы получили в свое распоряжение необходимые инструменты и загорелись желанием искать новые пути их применения!

Приложение A

Дополнительные библиотеки

В этой книге упоминались некоторые дополнительные библиотеки, которыми мы пользовались для сбора, сохранения и анализа данных. В настоящем приложении приводятся инструкции по их скачиванию и установке, а заодно некоторые примеры использования.

Universal Feed Parser

Библиотека Universal Feed Parser написана на языке Python Марком Пилгримом (Mark Pilgrim) для разбора каналов в формате RSS и Atom. Она неоднократно использовалась в книге для упрощения загрузки сообщений из блогов и статей из сетевых источников новостей. Домашняя страница библиотеки – *http://feedparser.org*.

Установка для всех платформ

Скачать библиотеку можно со страницы *http://code.google.com/p/feedparser/downloads/list*. Скачивайте последнюю версию, файл называется feedparser-X.Y.zip.

Распакуйте архив в пустую папку и введите следующую команду:

```
c:\download\feedparser>python setup.py install
```

Эта команда найдет папку, в которую был установлен Python, и установит библиотеку в нужное место. После установки достаточно ввести в интерпретаторе команду `import feedparser` и начать пользоваться библиотекой.

Примеры использования приведены на сайте *http://feedparser.org/*.

Python Imaging Library

Python Imaging Library (PIL) – библиотека с открытыми исходными текстами, которая наделяет Python средствами для создания и обработки изображений. Домашняя страница – *http://www.pythonware.com/products/pil*.

Установка на платформе Windows

Для библиотеки PIL на платформе Windows имеется Мастер установки. Зайдя на домашнюю страницу, прокрутите окно вниз, найдите раздел Downloads и скачайте последнюю версию исполняемого файла для вашей версии Python. Запустите файл и следуйте инструкциям на экране.

Установка на другие платформы

Для платформ, отличных от Windows, библиотеку придется собирать из исходных файлов. Исходные файлы можно скачать с домашней страницы, они будут работать с любой из последних версий Python.

Скачайте исходные файлы последней версии и введите следующие команды, подставив вместо 1.1.6 номер той версии, которую вы скачали:

```
$ gunzip Imaging-1.1.6.tar.gz
$ tar xvf Imaging-1.1.6.tar
$ cd Imaging-1.1.6
$ python setup.py install
```

В результате будут откомпилированы расширения, и библиотека будет установлена в папку Python.

Простой пример использования

В следующем примере создается небольшое изображение, рисуется несколько отрезков и выводится сообщение. Затем изображение сохраняется в виде JPEG-файла.

```
>>> from PIL import Image,ImageDraw
>>> img=Image.new('RGB',(200,200),(255,255,255)) # 200x200 белый фон
>>> draw=ImageDraw.Draw(img)
>>> draw.line((20,50,150,80),fill=(255,0,0))     # Красная линия
>>> draw.line((150,150,20,200),fill=(0,255,0))   # Зеленая линия
>>> draw.text((40,80),'Hello!',(0,0,0))          # Черный текст
>>> img.save('test.jpg','JPEG')                  # Сохранить файл test.jpg
```

Более полный набор примеров имеется на веб-странице *http://www.pythonware.com/library/pil/handbook/introduction.htm*.

Beautiful Soup

Библиотека Beautiful Soup – это написанный на Python анализатор документов в форматах HTML и XML. Он спроектирован так, что способен

работать с плохо написанными веб-страницами. В книге мы неоднократно применяли эту библиотеку для создания наборов данных с сайтов, не имеющих API, а также для выделения из страниц текста для индексирования. Домашняя страница библиотеки – *http://www.crummy.com/software/BeautifulSoup*.

Установка для всех платформ

Библиотека Beautiful Soup поставляется в виде одного исходного файла. Ближе к концу домашней страницы имеется ссылка для скачивания файла BeautifulSoup.py. Скачайте и поместите библиотеку либо в рабочую папку, либо в папку Python/Lib.

Простой пример использования

В следующем примере разбирается HTML-разметка домашней страницы Google и показывается, как извлекать элементы из DOM и искать ссылки:

```
>>> from BeautifulSoup import BeautifulSoup
>>> from urllib import urlopen
>>> soup=BeautifulSoup(urlopen('http://google.com'))
>>> soup.head.title
<title>Google</title>
>>> links=soup('a')
>>> len(links)
21
>>> links[0]
<a href="http://www.google.com/ig?hl=en">iGoogle</a>
>>> links[0].contents[0]
u'iGoogle'
```

Более полный набор примеров имеется на странице *http://www.crummy.com/software/BeautifulSoup/documentation.html*.

pysqlite

pysqlite – это интерфейс из языка Python к встраиваемой базе данных SQLite. В отличие от традиционных СУБД, встраиваемая база работает не в отдельном процессе, поэтому установка и настройка не требует больших усилий. Вся база данных SQLite хранится в одном файле. В этой книге библиотека pysqlite использовалась для сохранения некоторых собранных данных.

Домашняя страница pysqlite – *http://www.initd.org/tracker/pysqlite/wiki/pysqlite*.

Установка на платформе Windows

На домашней странице есть ссылка для скачивания двоичного инсталлятора для Windows. Скачайте и запустите этот файл. Мастер установки спросит, куда установлен Python на вашей машине, и произведет установку в указанную папку.

Установка на другие платформы

Для платформ, отличных от Windows, pysqlite придется собирать из исходных файлов. Исходные файлы можно скачать с домашней страницы в виде TGZ-архива. Скачайте последнюю версию и введите следующие команды, подставив вместо 2.3.3 номер скачанной версии:

```
$ gunzip pysqlite-2.3.3.tar.gz
$ tar xvf pysqlite-2.3.3.tar.gz
$ cd pysqlite-2.3.3
$ python setup.py build
$ python setup.py install
```

Простой пример использования

В следующем примере мы создаем новую таблицу, вставляем в нее строку и фиксируем транзакцию. Затем запрашиваем только что вставленную строку:

```
>>> from pysqlite2 import dbapi2 as sqlite
>>> con=sqlite.connect('test1.db')
>>> con.execute('create table people (name,phone,city)')
<pysqlite2.dbapi2.Cursor object at 0x00ABE770>
>>> con.execute('insert into people values ("toby","555-1212","Boston")')
<pysqlite2.dbapi2.Cursor object at 0x00AC8A10>
>>> con.commit( )
>>> cur=con.execute('select * from people')
>>> cur.next( )
(u'toby', u'555-1212', u'Boston')
```

Отметим, что в SQLite типы полей указывать необязательно. Чтобы этот пример работал с более традиционной СУБД, необходимо при создании таблиц указывать типы полей.

NumPy

NumPy – это математическая библиотека для языка Python, предоставляющая объект-массив, функции линейной алгебры и преобразование Фурье. Она очень популярна при выполнении научных расчетов

на Python и иногда даже используется вместо специализированных инструментов типа пакета MATLAB. В главе 10 мы пользовались библиотекой NumPy для реализации алгоритма NMF. Домашняя страница – *http://numpy.scipy.org*.

Установка на платформе Windows

На странице *http://sourceforge.net/project/showfiles.php?group_id=1369&package_id=175103* имеется двоичный инсталлятор NumPy для Windows.

Скачайте EXE-файл, соответствующий вашей версии Python, и запустите его. Мастер установки спросит, куда установлен Python на вашей машине, и произведет установку в указанную папку.

Установка на другие платформы

На других платформах NumPy собирается из исходных файлов, которые можно скачать со страницы *http://sourceforge.net/project/showfiles.php?group_id=1369&package_id=175103*. Скачайте TAR.GZ-файл, соответствующий вашей версии Python, и выполните следующие команды, подставив вместо 1.0.2 номер скачанной версии:

```
$ gunzip numpy-1.0.2.tar.gz
$ tar xvf numpy-1.0.2.tar.gz
$ cd numpy-1.0.2
$ python setup.py install
```

Простой пример использования

В этом примере показано, как создать матрицы, перемножить их, транспонировать и линеаризовать:

```
>>> from numpy import *
>>> a=matrix([[1,2,3],[4,5,6]])
>>> b=matrix([[1,2],[3,4],[5,6]])
>>> a*b
matrix([[22, 28],
        [49, 64]])
>>> a.transpose( )
matrix([[1, 4],
        [2, 5],
        [3, 6]])
>>> a.flatten( )
matrix([[1, 2, 3, 4, 5, 6]])
```

matplotlib

matplotlib – это библиотека двумерной графики для Python, которая гораздо лучше приспособлена для создания графиков математических

функций, чем Python Imaging Library. Качество создаваемых ею рисунков вполне приемлемо для включения в печатные публикации.

Установка

Перед установкой matplotlib необходимо сначала установить библиотеку NumPy, как описано в предыдущем разделе. Имеются двоичные дистрибутивы matplotlib для большинства платформ, в том числе Windows и Mac OS X, и пакеты для Linux в форматах RPM и Debian. Подробные инструкции по установке matplotlib на любую платформу имеются на странице *http://matplotlib.sourceforge.net/installing.html*.

Простой пример использования

В следующем примере мы наносим на график четыре точки (1,1), (2,4), (3,9) и (4,16) в виде оранжевых кружочков. Затем результат сохраняется в файле и выводится в окне на экране:

```
>>> from pylab import *
>>> plot([1,2,3,4], [1,4,9,16], 'ro')
[<matplotlib.lines.Line2D instance at 0x01878990>]
>>> savefig('test1.png')
>>> show( )
```

Большая подборка примеров имеется на странице *http://matplotlib.sourceforge.net/tutorial.html*.

pydelicious

pydelicious – библиотека для получения данных с сайта социальных закладок *del.icio.us*. Для этого сайта имеется официальный API, который мы использовали в некоторых вызовах, но pydelicious добавляет кое-какие дополнительные возможности, задействованные нами в главе 2 при построении механизма рекомендования. В настоящее время библиотека pydelicious размещена на сайте Google Code; вы найдете ее по адресу *http://code.google.com/p/pydelicious/source*.

Установка для всех платформ

Получить последнюю версию pydelicious проще всего, если на вашей машине установлена система управления версиями Subversion. В этом случае достаточно ввести такую команду:

```
svn checkout http://pydelicious.googlecode.com/svn/trunk/pydelicious.py
```

Если Subversion у вас нет, то файлы можно скачать со страницы *http://pydelicious.googlecode.com/svn/trunk*.

Имея необходимые файлы, выполните команду `python setup.py install`, находясь в той же папке, где и сами файлы. В результате pydelicious будет установлена в папку Python.

Простой пример использования

В библиотеке pydelicious имеется целый ряд функций для получения популярных закладок или закладок, оставленных конкретным пользователем. Она также позволяет добавлять в свою учетную запись новые закладки:

```
>> import pydelicious
>> pydelicious.get_popular(tag='programming')
[{'count': '', 'extended': '', 'hash': '',
'description': u'How To Write Unmaintainable Code',
'tags': '', 'href': u'http://thc.segfault.net/root/phun/unmaintain.html',
'user': u'dorsia', 'dt': u'2006-08-19T09:48:56Z'},
{'count': '', 'extended': '', 'hash': '',
'description': u'Threading in C#', 'tags': '',
'href':u'http://www.albahari.com/threading/', etc...
>> pydelicious.get_userposts('dorsia')
[{'count': '', 'extended': '', 'hash': '',
'description': u'How To Write Unmaintainable Code',
'tags': '', 'href': u'http://thc.segfault.net/root/phun/unmaintain.html',
'user': u'dorsia', 'dt': u'2006-08-19T09:48:56Z'}, etc...
>>> a = pydelicious.apiNew(user, passwd)
>>> a.posts_add(url="http://my.com/", desciption="my.com",
                extended="the url is my.moc", tags="my com")
True
```

Приложение B

Математические формулы

На страницах этой книги встречался ряд математических понятий. В этом приложении они собраны и описаны, приведены соответствующие формулы и код.

Евклидово расстояние

Евклидово расстояние определяет расстояние между двумя точками в многомерном пространстве. Это то расстояние, которое вы измеряете с помощью обычной линейки. Расстояние между точками с координатами $(p_1, p_2, p_3, p_4, \ldots)$ и $(q_1, q_2, q_3, q_4, \ldots)$ выражается по формуле B.1.

$$\sqrt{(p_1-q_1)^2+(p_2-q_2)^2+\ldots+(p_n-q_n)^2} = \sqrt{\sum_{i=1}^{n}(p_i-q_i)^2}$$

Формула B.1. *Евклидово расстояние*

Вот очевидная реализация этой формулы:

```
def euclidean(p,q):
  sumSq=0.0
  # Суммируем квадраты разностей
  for i in range(len(p)):
    sumSq+=(p[i]-q[i])**2
  # Извлекаем квадратный корень
return (sumSq**0.5)
```

Евклидово расстояние используется в разных местах книги для определения степени схожести двух образцов.

Коэффициент корреляции Пирсона

Коэффициент корреляции Пирсона – это мера скоррелированности двух переменных. Он принимает значения от 1 до –1, где 1 означает, что корреляция между переменными идеальна, 0 – что корреляции нет, а –1 – что имеется идеальная обратная корреляция.

Корреляция Пирсона рассчитывается по формуле B.2.

$$r = \frac{\sum XY - \frac{\sum X \sum Y}{N}}{\sqrt{\left(\sum X^2 - \frac{(\sum X)^2}{N}\right)\left(\sum Y^2 - \frac{(\sum Y)^2}{N}\right)}}$$

Формула B.2. *Коэффициент корреляции Пирсона*

Эта формула реализуется такой функцией:

```
def pearson(x,y):
  n=len(x)
  vals=range(n)

  # Простые суммы
  sumx=sum([float(x[i]) for i in vals])
  sumy=sum([float(y[i]) for i in vals])

  # Суммы квадратов
  sumxSq=sum([x[i]**2.0 for i in vals])
  sumySq=sum([y[i]**2.0 for i in vals])

  # Сумма произведений
  pSum=sum([x[i]*y[i] for i in vals])

  # Вычисляем коэффициент корреляции Пирсона
  num=pSum-(sumx*sumy/n)
  den=((sumxSq-pow(sumx,2)/n)*(sumySq-pow(sumy,2)/n))**.5
  if den==0: return 0

  r=num/den

  return r
```

Мы пользовались корреляцией Пирсона в главе 2 для вычисления степени подобия между предпочтениями двух людей.

Взвешенное среднее

Взвешенное среднее – это разновидность среднего значения, при котором каждому усредняемому наблюдению приписывается определенный

вес. Оно используется для прогнозирования числовых значений на основе оценок подобия. Взвешенное среднее вычисляется по формуле B.3, в которой x_1, ..., x_n – наблюдения, а w_1, ..., w_n – веса.

$$\bar{x} = \frac{w_1x_1 + w_2x_2 + \ldots + w_nx_n}{w_1 + w_2 + \ldots + w_n}$$

Формула B.3. *Взвешенное среднее*

Ниже приведена простая реализация этой формулы в виде функции, принимающей списки значений и весов:

```
def weightedmean(x,w):
  num=sum([x[i]*w[i] for i in range(len(w))])
  den=sum([w[i] for i in range(len(w))])

  return num/den
```

В главе 2 взвешенные средние применялись для прогнозирования того, в какой мере вам понравится фильм. Для этого вычисляется средняя оценка из тех, что были поставлены фильму другими людьми, взвешенная с учетом схожести их предпочтений с вашими. С главе 8 взвешенные средние применялись для прогнозирования цен.

Коэффициент Танимото

Коэффициент Танимото измеряет степень схожести двух множеств. В этой книге он использовался для того, чтобы оценить подобие образцов, представленных списками свойств. Пусть есть два множества:

A = [рубашка, туфли, брюки, носки]

B = [рубашка, юбка, туфли]

Их пересечение C равно [рубашка, туфли]. Коэффициент Танимото вычисляется по формуле B.4, в которой N_a – количество элементов в A, N_b – количество элементов в B, N_c – количество элементов в пересечении C.

В данном случае коэффициент Танимото равен $2/(4 + 3 - 2) = 2/5 = 0{,}4$.

$$T = \frac{N_c}{(N_a + N_b - N_c)}$$

Формула B.4. *Коэффициент Танимото*

Ниже приведена простая функция, которая принимает два списка и вычисляет для них коэффициент Танимото:

```
def tanimoto(a,b):
  c=[v for v in a if v in b]
  return float(len(c))/(len(a)+len(b)-len(c))
```

Коэффициент Танимото использовался в главе 3 для вычисления подобия между людьми в процессе кластеризации.

Условная вероятность

Вероятность – это характеристика частоты возникновения некоторого события. Обычно ее записывают следующим образом: $Pr(A) = x$, где A – событие. Например, можно сказать, что сегодня вероятность дождя составляет 20%, и записать это в виде $Pr(\text{Дождь}) = 0{,}2$.

Если бы, выглянув в окно, вы увидели, что небо в тучах, то могли бы заключить, что шансы на дождь выше. В этом случае говорят об *условной вероятности* события A при условии, что известно B. Записывается это в виде $Pr(A \mid B)$, а в данном примере $Pr(\text{Дождь} \mid \text{Пасмурно})$.

Условная вероятность равна вероятности того, что произойдут оба события, поделенной на вероятность условия (формула B.5).

$$\Pr(A|B) = \frac{\Pr(A \cap B)}{\Pr(B)}$$

Формула B.5. *Условная вероятность*

Таким образом, если в 10% случаев утром была пасмурно, а позже шел дождь, а в 25% случаев утром пасмурно, то $Pr(\text{Дождь} \mid \text{Пасмурно}) = 0{,}1/0{,}25 = 0{,}4$.

Поскольку это элементарное деление, то никакой функции не приводится. Условная вероятность применялась в главе 6 для фильтрации документов.

Коэффициент Джини

Коэффициент Джини измеряет неоднородность набора. Если имеется набор $[A, A, B, B, B, C]$, то коэффициент Джини – это вероятность того, чтобы вы ошибетесь, выбрав какой-то элемент и попытавшись случайно угадать его метку. Если бы набор состоял только из элементов A, то вы всегда говорили бы A и никогда не ошибались, следовательно, набор полностью однороден.

Коэффициент Джини рассчитывается по формуле B.6.

$$I_G(i) = 1 - \sum_{j=1}^{m} f(i,j)^2 = \sum_{j \neq k} f(i,j)\, f(i,k)$$

Формула B.6. *Коэффициент Джини*

Следующая функция принимает список элементов и вычисляет для него коэффициент Джини:

```
def giniimpurity(l):
  total=len(l)
  counts={}
  for item in l:
```

```
    counts.setdefault(item,0)
    counts[item]+=1

  imp=0
  for j in l:
    f1=float(counts[j])/total
    for k in l:
      if j==k: continue
      f2=float(counts[k])/total
      imp+=f1*f2
  return imp
```

В главе 7 коэффициент Джини использовался при моделировании с помощью деревьев решений, чтобы определить, станет ли набор более однородным после разбиения.

Энтропия

Энтропия – это еще один способ измерения неоднородности набора. Это понятие заимствовано из теории информации и измеряет степень беспорядочности набора. Неформально говоря, энтропия характеризует то, насколько вы удивитесь случайно выбранному из набора элементу. Если бы набор состоял только из элементов A, то, выбрав A, вы нисколько не удивились бы, поэтому энтропия такого набора равна 0. Энтропия вычисляется по формуле B.7.

$$H(X) = \sum_{i=1}^{n} p(x_i) \log_2 \frac{1}{p(x_i)} = -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i)$$

***Формула B.7.** Энтропия*

Следующая функция принимает список элементов и вычисляет его энтропию:

```
def entropy(l):
  from math import log
  log2=lambda x:log(x)/log(2)

  total=len(l)
  counts={}
  for item in l:
    counts.setdefault(item,0)
    counts[item]+=1

  ent=0
  for i in counts:
    p=float(counts[i])/total
    ent-=p*log2(p)
  return ent
```

В главе 7 энтропия использовалась при моделировании с помощью деревьев решений, чтобы определить, уменьшает ли разбиение беспорядочность набора.

Дисперсия

Дисперсия измеряет степень разброса списка чисел относительно их среднего значения. Она часто используется в статистике для определения того, насколько велики различия между членами некоторого набора. Вычисляется она путем усреднения квадратов разностей между каждым числом и средним, как показано в формуле B.8.

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2$$

Формула B.8. *Дисперсия*

Ниже приведена простая функция, реализующая эту формулу:

```
def variance(vals):
  mean=float(sum(vals))/len(vals)
  s=sum([(v-mean)**2 for v in vals])
  return s/len(vals)
```

В главе 7 дисперсия использовалась при моделировании с помощью деревьев решений, чтобы определить, как лучше разбить множество, чтобы подмножества оказались более «кучными».

Гауссова функция

Гауссова функция описывает плотность распределения вероятности нормальной кривой. Она использовалась в книге как весовая функция во взвешенном методе *k*-ближайших соседей, поскольку ее начальное значение велико и быстро убывает, никогда не становясь равным 0.

Формула гауссова распределения с дисперсией σ показана ниже:

$$\frac{1}{\sigma\sqrt{2\pi}}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

Формула B.9. *Гауссова функция*

Реализовать ее можно функцией из двух строк:

```
import math
def gaussian(dist,sigma=10.0):
  exp=math.e**(-dist**2/(2*sigma**2))
  return (1/(sigma*(2*math.pi)**.5))*exp
```

В главе 8 гауссова функция использовалась как одна из весовых функций для построения механизма прогнозирования числовых значений.

Скалярное произведение

Скалярное произведение – это способ перемножения двух векторов. Скалярное произведение векторов $a = (a_1, a_2, a_3, ...)$ и $b = (b_1, b_2, b_3, ...)$ вычисляется по формуле B.10.

$$a \bullet b = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + ... + a_n b_n$$

Формула B.10. *Скалярное произведение, выраженное через координаты*

Для реализации скалярного произведения применяется следующая функция:

```
def dotproduct(a,b):
return sum([a[i]*b[i] for i in range(len(a))])
```

Если θ – угол между двумя векторами, то скалярное произведение можно вычислить также по формуле B.11.

$$a \bullet b = |a||b| \cos\theta$$

Формула B.11. *Скалярное произведение, выраженное через угол*

Это означает, что скалярное произведение можно использовать для вычисления угла между векторами:

```
from math import acos

# Вычисляем длину вектора
def veclength(a):
  return sum([a[i] for i in range(len(a))])**.5

# Вычисляем угол между векторами
def angle(a,b):
  dp=dotproduct(a,b)
  la=veclength(a)
  lb=veclength(b)
  costheta=dp/(la*lb)
  return acos(costheta)
```

Скалярные произведения применялись в главе 9 для вычисления углов между векторами, соответствующими классифицируемым образцам.

Алфавитный указатель

А

авиарейсы, поиск 127
алгоритмы 22
  выделение признаков 261
  выделения основы 85
  иерархическая кластеризация 57
  коллаборативная фильтрация 27
  масс и пружин 138
  матричная математика 270
  обучающие, другие применения 24
  рекомендование на основе метода коллаборативной фильтрации по схожести образцов 48
  сводка 312
    байесовский классификатор 312
  спуска с горы 118
    со случайным перезапуском 120
анализ
  независимых компонентов 25
  фондовых рынков 25

Б

Байеса теорема 152
байесовская классификация 264
байесовский классификатор 168, 312
  классификация 314
  наивный 314
  обучение 313
  сильные и слабые стороны 315
  сочетание признаков 316
биотехнология 24
блоги
  подсчет слов 52
  фильтрация 161
  кластеризация по частоте слов 51
булевские операции 109

В

векторы слов 51
  кластеризация блогов по частоте слов 52
  подсчет количества слов в RSS-канале 52
вероятность 354
  график 217
  для всего документа 151
  комбинирование 157
  предполагаемая 150
  теорема Байеса 152
  условная 149
взвешенное среднее 206, 330, 352
взвешенные соседи 203
  гауссова функция 205
  инвертирующая функция 203
  колоколообразная кривая 205
  функция вычитания 204
визуализация сети
  задача о размещении 137
  подсчет пересекающихся линий 138
  рисование сети 140
Википедия 21, 80
виртуальные признаки 169
вложенный словарь 27

внешние ссылки 93
алгоритм PageRank 94
коэффициент затухания 95
использование текста ссылки 98
простой подсчет 94
выделение основы, алгоритм 85
выделение признаков 259
из новостей 260
выделение слов 109
выработка рекомендаций 26
взвешенная сумма оценок 35
использование набора данных MovieLens 46
коллаборативная фильтрация 27
отыскание похожих пользователей 29
какой оценкой подобия пользоваться 33
коэффициент корреляции Пирсона 31
оценка по евклидову расстоянию 29
ранжирование критиков 34
подбор предметов 37
построение рекомендателя ссылок для сайта del.icio.us 39
del.icio.us API 40
рекомендование соседей и ссылок 42
создание набора данных 40
рекомендование предметов 35
сбор информации о предпочтениях 27
упражнения 49
фильтрация по схожести образцов 43
выдача рекомендаций 44
и фильтрация по схожести пользователей 48
набор данных для сравнения образцов 43

Г

гауссова функция 205, 356
взвешенная сумма 220
генетические алгоритмы 123, 342
и генетическое программирование 286
критерий остановки 142
мутация 123
поколение 123
популяция 123
скрещивание 123
генетическое программирование 125, 285
дерево разбора 287
и генетические алгоритмы 286
и язык Python 288
игра с человеком 307
измерение успеха 294
круговой турнир 305
мутация заменой узла 311
мутация программ 295
направления развития 308
обзор 285
память 309
построение окружающей среды 300
прекращение эволюции 311
проверка решения 293
программы как деревья 287
простые игры 303
игра с человеком 307
круговой турнир 305
Погоня 303
разнообразие 303
упражнения 311
симулятор игры в крестики-нолики 311
скрещивание 285
скрытые функции 311
случайное скрещивание 311
создание начальной популяции 292
типы данных 309
объекты 310
словари 310
списки 310
строки 309
типы функций 311
узлы с типами данных 311
успехи 287
функция выживаемости 285
числовые функции 308
элитизм 300
гетерогенные переменные 209
масштабирование измерений 211
гиперболический тангенс 103
гиперплоскость с максимальным отступом 248
глобальный минимум 120
Голдберг, Дэвид 27
Голливудская фондовая биржа 24
граница решения 233

группы, обнаружение 50
кластеризация блогов 76
манхэттенское расстояние 76
обучение с учителем и без него 51
упражнения 76

Д

данные, просмотр на двумерной плоскости 71
данные о фондовом рынке 277
Yahoo! Finance 278
вывод результатов 280
объем торгов акциями Google 282
подготовка матрицы 279
прогон алгоритма NMF 280
цена закрытия 277
дерево разбора 287
дендрограмма 55
рисование 60
drawnode, функция 61
деревья решений 170
API сайта Zillow 189
алгоритм CART 174
введение 173
вероятности результатов 196
в реальном мире 184
диапазоны отсутствующих данных 196
классификация новых наблюдений 183
когда применять 194
коэффициент Джини 176
многопутевое расщепление 196
моделирование
степени привлекательности 191
цен на недвижимость 188
наилучшее разбиение 176
недостатки 195
обучение 174
отображение 180
в графическом виде 181
отсутствующие данные 186
прогнозирование количества регистраций 171
ранний останов 196
рекурсивное построение 178
сокращение 185
упражнения 196
числовые результаты 188
энтропия 177
дисперсия 356
дополнительные библиотеки 344
Beautiful Soup 345
matplotlib 348
NumPy 347
pydelicious 349
pysqlite 346
Python Imaging Library 345
Universal Feed Parser 344

Е

евклидово расстояние 29, 235, 351
k-ближайшие соседи 329
код 351
оценка подобия 29

З

задача о вечеринке 259
запросы 86
знакомства, сайты 24

И

иерархическая кластеризация 54, 333
алгоритм 57
близость 56
выполнение 59
дендрограмма 55
коэффициент корреляции Пирсона 56
распечатка 59
имитация отжига
алгоритм 120
определение 120
инвертирующая функция 203
индексирование 77
выделение слов на странице 84
добавление в индекс 85
построение индекса 82
создание схемы 83
таблицы 83
искусственный интеллект (ИИ) 22
использование данных о фондовом рынке 277
источники новостей 260

К

категориальные свойства
вычисление расстояний с помощью Yahoo! Maps 239
ответы да/нет 238
списки интересов 238

классификаторы
на базе нейронной сети 169
на основе дерева решений 232
взаимозависимые переменные 320
обучение 317
сильные и слабые стороны 319
обучение 146
с учителем 259
простой линейный 234
сохранение обученных 159
SQLite 160
классификация
байесовский классификатор 314
документов 145
обучение классификатора 146
кластеризация 50, 259, 265
методом K-средних 64, 282, 334
столбцов 63
типичные применения 50
кластеры предпочтений 67
Beautiful Soup, библиотека 67
кластеризация результатов 71
определение метрики близости 69
получение и подготовка данных 67
разбор страниц сайта Zebo 68
коллаборативная фильтрация 27
алгоритм 27
первое употребление термина 27
по схожести пользователей 43
эффективность 49
коллективный разум
введение 19
определение 20
колоколообразная кривая 205
коэффициент Джини 176
формула 354
коэффициент Жаккарда 34
коэффициент корреляции Пирсона 29, 31, 352
иерархическая кластеризация 56
код 352
многомерное шкалирование 72
коэффициент Танимото 49, 69, 353
код 353
круговой турнир 305
кумулятивная вероятность 217

Л

линейная классификация 234
векторы 235
скалярные произведения 236
линия наилучшего приближения 31
локальный минимум 120, 341

М

манхэттенское расстояние 34, 76
маркетинг 25
масс и пружин, алгоритм 138
масштабирование
данных 242
измерений 211
математические формулы 351
взвешенное среднее 353
гауссова функция 356
дисперсия 356
евклидово расстояние 351
коэффициент Джини 354
коэффициент корреляции Пирсона 352
коэффициент Танимото 353
скалярное произведение 357
условная вероятность 354
энтропия 355
матрица
весов 268
данных 271
признаков 268
матричная математика 267
factorize, функция 272
NumPy, библиотека 269
алгоритм факторизации 270
вывод результатов 273, 280
матрица данных 271
мультипликативные правила обновления 271
подготовка матрицы 279
транспонирование 267
умножение 267
факторизация 268
машинное зрение 25
машинное обучение 22
ограничения 23
машины опорных векторов (SVM) 229, 324
байесовский классификатор 258
библиотека LIBSVM 327
другие ядра, поддерживаемые LIBSVM 258
иерархия интересов 258
оптимизация разделителя 258
переход к ядру 325
полиномиальная трансформация 326

сильные и слабые стороны 328
скалярные произведения 325
создание модели 257
упражнения 258
методы обучения с учителем 51, 313
метрика близости 69
метрики ранжирования 93
алгоритм PageRank 94
использование текста ссылки 98
простой подсчет ссылок 94
мутация 285, 295
многомерное шкалирование 72, 76, 336
код 338
корреляция Пирсона 72
функция 73
многоуровневый перцептрон 99, 320
модели 22
мультипликативные правила обновления 271
мутация 123

Н

набор данных для подбора пар 230
затруднения при анализе данных 231
категориальные свойства 238
классификатор на основе дерева решений 232
масштабирование данных 242
применение библиотеки LIBSVM 251
создание 241
наивный байесовский классификатор 151, 314
выбор категории 153
и метод Фишера 155
сильные и слабые стороны 315
национальная безопасность 25
независимые признаки 259
альтернативные способы вывода 283
источники новостей 282
кластеризация методом K-средних 282
критерий останова 283
оптимизация и факторизация 283
упражнения 282
нейронная сеть 78, 320
и словосочетания 320
искусственная 99
обратное распространение 105
подготовка базы данных 100
подключение к поисковой машине 108
проверка результатов обучения 108
проектирование сети отслеживания переходов 99
прямой проход 103
использование кода 323
как черный ящик 324
метод обратного распространения 323
многоуровневый перцептрон 320
обучение 323
сильные и слабые стороны 324
синапсы 320
неотрицательная матричная факторизация (NMF) 51, 267, 338
использование кода 340
правила обновления 340
цель 340
неравномерные распределения 214
графическое представление вероятностей 216
оценка плотности распределения вероятности 215
нормализация 90

О

обесценивание оценок 32
обратная функция хи-квадрат 157
обратное распространение 105, 323
обучающие наборы 207
обучение
байесовского классификатора 313
классификатора на основе деревьев решений 317
нейронной сети 323
без учителя 51, 259
объем торгов 277
оперативные методы 332
опорные векторы 248
оптимизация 111, 213, 227, 341
алгоритм спуска с горы 118
имитация отжига 120
когда может не работать 126
критерий остановки генетического алгоритма 142
начальные точки отжига 142
поиск авиарейсов 127
представление решений 113
случайный поиск 117
сочетание студентов 143
с учетом предпочтений 132

выполнение 136
распределение студентов по комнатам 133
целевая функция 135
упражнения 142
целевая функция 114, 341
группового путешествия 142
цены на билеты туда и обратно 143
штраф за величину угла 143
оптимизация цепочек поставщиков 25

П

паттерны употребления слов 259
паук 79
Beautiful Soup, библиотека 80
urllib2, библиотека 80
код 81
перекрестный контроль 207, 331
обучающие наборы 207
с исключением по одному образцу 228
суммирование квадратов разностей 208
тестовые наборы 207
функция cross_validation 251
переход к ядру 245, 325
функция радиального базиса 245
Пилгрим Марк 344
планирование группового путешествия 112
время
аренды автомобилей 115
в пути 115
вылета 115
ожидания 115
цена 115
Погоня, игра 303, 311
подобие признаков 49
поиск во внешних ссылках 110
поисковые машины
булевские операции 109
вертикальные 127
выделение слов 109
запросы 86
обзор 77
поиск во внешних ссылках 110
поиск с учетом длины документа 110
точное соответствие 110
упражнения 109
учет частоты слов 110
поколение 123
ползание 77
полиномиальная трансформация 326
популяция 123, 285, 342
и разнообразие 292
правила обновления 340
признаки 313
прогнозирование числовых результатов 197
прямой проход 103

Р

ранжирование по содержимому 88
нормализация 90
расположение в документе 89
расстояние между словами 89, 93
частота слов 89, 90
распределения, неравномерные 214
расстояние между словами 89, 93
рекомендование на основе истории покупок 24
рекурсивное построение дерева 178
разнообразие 303
рынки прогнозов 24

С

самоорганизующиеся карты 51
сеть отслеживания переходов 99
сигмоидные функции 103
синапсы 320
скалярное произведение 236, 325, 357
скрещивание 123
случайный поиск 117
списки интересов 238
скрещивание 285, 298
стохастическая оптимизация 111

Т

температура 342
тестовые наборы 207
точное соответствие 110
транспонирование 267

У

угол между векторами, вычисление 357
уровень запроса 99
условная вероятность 149, 354
теорема Байеса 152

Ф

фильтрация документов 144
Akismet 166
виртуальные признаки 169
вычисление Pr(Документ) 169
вычисление вероятностей 149
предполагаемая вероятность 150
условная вероятность 149
каналы блогов 161
классификация документов 145
обучение классификаторов 146
метод Фишера 155
классификация образцов 158
комбинирование вероятностей 157
сравнение с наивным байесовским фильтром 155
на основе правил 145
наивный байесовский классификатор 151
выбор категории 153
переменные предполагаемые вероятности 168
сохранение обученных классификаторов 159
SQLite 160
спам 144, 145
порог 154
упражнения 168
усовершенствование алгоритма обнаружения признаков 164
фильтр на базе нейронной сети 169
фразы произвольной длины 169
фильтрация по схожести образцов 43
выдача рекомендаций 44
закладок 49
и фильтрация по схожести пользователей 48
набор данных для сравнения образцов 43
финансовое мошенничество 24
финансовые рынки 20
Фишера метод 155
классификация образцов 158
комбинирование вероятностей 157
сравнение с наивным байесовским фильтром 155
форумы 144
функция
вычитания 204
радиального базиса 245
выживаемости 285
фьючерсные рынки 20

Х

хи-квадрат, распределение 157
Холланд Джон 126

Ц

целевая функция 114, 135, 341
глобальный минимум 342
группового путешествия 142
локальный минимум 341
цена закрытия 277
ценовые модели 197
k-ближайшие соседи 199
изменение параметра ss для построения графика распределения вероятностей 228
исключение переменных 228
набор данных о лэптопах 228
оптимизация количества соседей 227
перекрестный контроль с исключением по одному образцу 228
поиск по атрибутам 228
построение демонстрационного набора данных 198
упражнения 227
центроиды 335
цены на недвижимость, моделирование 188
API сайта Zillow 189

Ч

частота слов 89, 90, 110
черный ящик 324

Ш

штраф за величину угла 143

Э

элитизм 300
энтропия 177, 355
код 355

Я

ядерные методы 229
идея 243
ядро
в библиотеке LIBSVM 258
выбор наилучших параметров 258

A

advancedclassify.py
dotproduct, функция 236
dpclassify, функция 237
getlocation, функция 240
getoffset, функция 246
lineartrain, функция 234
loadnumerical, функция 241
matchcount, функция 238
matchrow, класс
loadmatch, функция 231
milesdistance, функция 239
nlclassify, функция 246
rbf, функция 245
scaledate, функция 242
yesno, функция 238
agesonly.csv, файл 230
Akismet 14, 166
akismettest.py 167
Amazon 24, 76
API
сайта eBay 220, 228
выполнение поиска 222
ключ разработчика 220
краткое руководство для начинающих 220
подготовка соединения 221
получение подробной информации о товаре 225
построение предсказателя цен 226
сайта Zillow 189
articlewords, словарь 264
Atom, каналы
подсчет слов 52
разбор 344
Audioscrobbler 49

B

Beautiful Soup, библиотека 67, 345
паук 80
пример использования 346
установка 346

C

CART (Classification and Regression Trees) 174
clusters.py
bicluster, класс 57
draw2d, функция 74
drawdendrogram, функция 61
drawnode, функция 61
getheight, функция 60
hcluster, функция 58
printclust, функция 59
readfile, функция 56
rotatematrix, функция 63
scaledown, функция 73

D

del.icio.us 14, 349
построение рекомендателя ссылок 39
API сайта del.icio.us 40
построение набора данных 40
рекомендование соседей и ссылок 42
deliciousrec.py
fillitems, функция 41
initializeUserDict, функция 41
docclass.py
classifier, класс
catcount, метод 161
categories, метод 161
classify, метод 154
fcount, метод 160
fisherclassifier, класс 156
fprob, метод 149
incc, метод 160
incf, метод 160
setdb, метод 160
totalcount, метод 161
train, метод 148
weightedprob, метод 150
fisherclassifier, класс
classify, метод 159
fisherprob, метод 157
setminimum, метод 158
getwords, функция 145
naivebayes, класс 152
prob, метод 153
sampletrain, функция 148
dorm.py 133
dormcost, функция 135
printsolution, функция 134

E

eBay 9
ebaypredict.py
doSearch, функция 222
getCategory, функция 223

getHeaders, функция 221
getItem, функция 225
getSingleValue, функция 222
makeLaptopDataset, функция 226
sendRequest, функция 221

F

Facebook 137
другие прогнозы 258
загрузка данных о друзьях 255
ключ разработчика 252
подбор пар 252
построение набора данных для подбора пар 256
создание сеанса 253
facebook.py
arefriends, функция 256
createtoken, функция 254
fbsession, класс 253
getfriends, метод 255
getinfo, метод 255
getlogin, метод 254
getsession, метод 254
makedataset, метод 256
makehash, метод 254
sendrequest, метод 254
factorize, функция 272
feedfilter.py 162
entryfeatures, метод 165
feedparser, класс 262

G

generatefeedvector.py 52, 53
getwords, функция 53
Geocoding, служба 239
API 240
Google 19, 21, 23
PageRank, алгоритм 23, 94
Google Blog Search 162
gp.py 288
buildhiddenset, функция 294
constnode, класс 288, 289
crossover, функция 299
evolve, функция 300, 303
fwrapper, класс 288, 289
getrankfunction, функция 301
gridgame, функция 304
hiddenfunction, функция 294
humanplayer, класс 307
mutate, функция 297
node, класс 288, 289
display, метод 291
exampletree, функция 290
makerandomtree, функция 292
paramnode, класс 288, 289
rankfunction, функция 300
scorefunction, функция 295
tournament, функция 306
GroupLens, проект 46
сайт 48

H

Hot or Not 14, 191
hotornot.py
getpeopledata, функция 193
getrandomratings, функция 192
HTML-документы, анализатор 345

I

IP-адреса 169
Item-based Collaborative Filtering Recommendation Algorithms, статья 48

K

k-ближайшие соседи (kNN) 199, 329
взвешенное среднее 330
в каких случаях применять 227
евклидово расстояние 329
количество соседей 200
масштабирование и лишние переменные 330
определение подобия 201
перекрестный контроль 331
сильные и слабые стороны 332
Kayak 14, 143
API 127, 132
data 129
firstChild 128
getElementsByTagName 128
kayak.py 129
createschedule, функция 131
flightsearch, функция 129
flightsearchresult, функция 130
getkayaksession, функция 129

L

Last.fm, сайт 24
LIBSVM, библиотека 327
другие ядра 258
и набор данных для подбора пар 251

приложения 249
пример сеанса 250
LinkedIn 137

M

matchmaker.csv, файл 230
matplotlib, библиотека 216
пример использования 349
установка 349
minidom, пакет 128, 190
MovieLens, набор данных 46

N

Netflix, компания 19, 24
newsfeatures.py 261
getarticlewords, функция 262
makematrix, функция 263
separatewords, функция 262
shape, функция 270
showarticles, функция 274, 275
showfeatures, функция 273, 276
stripHTML, функция 262
transpose, функция 270
nmf.py
difcost, функция 271
nn.py
searchnet, класс 101
generatehiddennode, функция 102
getstrength, метод 101
setstrength, метод 101
numpredict.py
createcostfunction, функция 213
createhiddendataset, функция 214
crossvalidate, функция 208, 213
cumulativegraph, функция 217
dividedata, функция 207
euclidian, функция 201
gaussian, функция 205
getdistances, функция 202
inverseweight, функция 204
knnestimate, функция 202
probabilitygraph, функция 218
probguess, функция 215, 217
rescale, функция 212
subtractweight, функция 204
testalgorithm, функция 207
weightedknn, функция 206
wineprice, функция 198
wineset1, функция 198
wineset2 функция 210
NumPy, библиотека 269, 347
пример использования 348
установка 348

O

optimization.py 112, 213
annealingoptimize, функция 121
geneticoptimize, функция 124
getminutes, функция 113
hillclimb, функция 119
printschedule, функция 114
randomoptimize, функция 117
schedulecost, функция 115

P

PageRank, алгоритм 23, 94
Pandora, сайт 24
poplib, библиотека 169
Porter Stemmer, библиотека 85
Pr(Документ) 169
pydelicious, библиотека 349
пример использования 350
установка 349
pysqlite, библиотека 82, 346
импорт 160
пример использования 347
установка 347
Python, язык программирования
достоинства 11
замечания 12
и генетическое
программирование 287
визуализация программы 291
обход дерева 288
построение и вычисление деревьев 290
представление деревьев 288
Python Imaging Library, библиотека 60, 345
пример использования 345
установка 345

R

recommendations.py 27
calculateSimilarItems, функция 44
getRecommendations, функция 36
getRecommendedItems, функция 45
loadMovieLens, функция 47
sim_distance, функция 30

sim_pearson, функция 32
topMatches, функция 34
transformPrefs, функция 38
RSS-каналы
подсчет количества слов 52
разбор 344
фильтрация 161

S

searchengine.py
addtoindex, функция 85
crawler, класс 78, 81, 83
createindextables, функция 83
distancescore, функция 93
frequencyscore, функция 91
getentryid, функция 85
getmatchrows, функция 87
gettextonly, функция 84
inboundlinkscore, функция 94
isindexed, функция 82, 86
linktextscore, функция 98
nnscore, функция 109
normalizescores, функция 90
query, метод 108
searcher, класс 89
searchnet, класс
backPropagate, метод 106
separatewords, функция 84
trainquery, метод 107
updatedatabase, метод 107
подключение нейронной сети 108
предложения import 80
searchindex.db 83, 86
socialnetwork.py 137
crosscount, функция 139
drawnetwork, функция 140
SpamBayes, подключаемый модуль 155
SQLite 82
интерфейс к встраиваемой базе данных 346
сохранение обученных классификаторов 159
таблицы 83
stockfeatures.txt 281
stockvolume.py 279, 280
factorize, метод 280

T

treepredict.py 172
buildtree, функция 179
classify, функция 183
decisionnode, класс 174
divideset, функция 174
drawnode, функция 182
drawtree, функция 182
entropy, функция 177
mdclassify, функция 187
printtree, функция 180
prune, функция 185
split_function, функция 175
uniquecounts, функция 176
variance, функция 188

U

Universal Feed Parser, библиотека 52, 161, 344
urllib2, библиотека 80, 128
Usenet 144

W

wordlocation, таблица 88

X

XML-документы, анализатор 345
xml.dom 128

Y

Yahoo!, ключ разработчика 239
Yahoo! Finance 75
Yahoo! Groups 144
Yahoo! Maps 239

Z

Zebo, сайт 67
zillow.py
getaddressdata, функция 190
getpricelist, функция 190