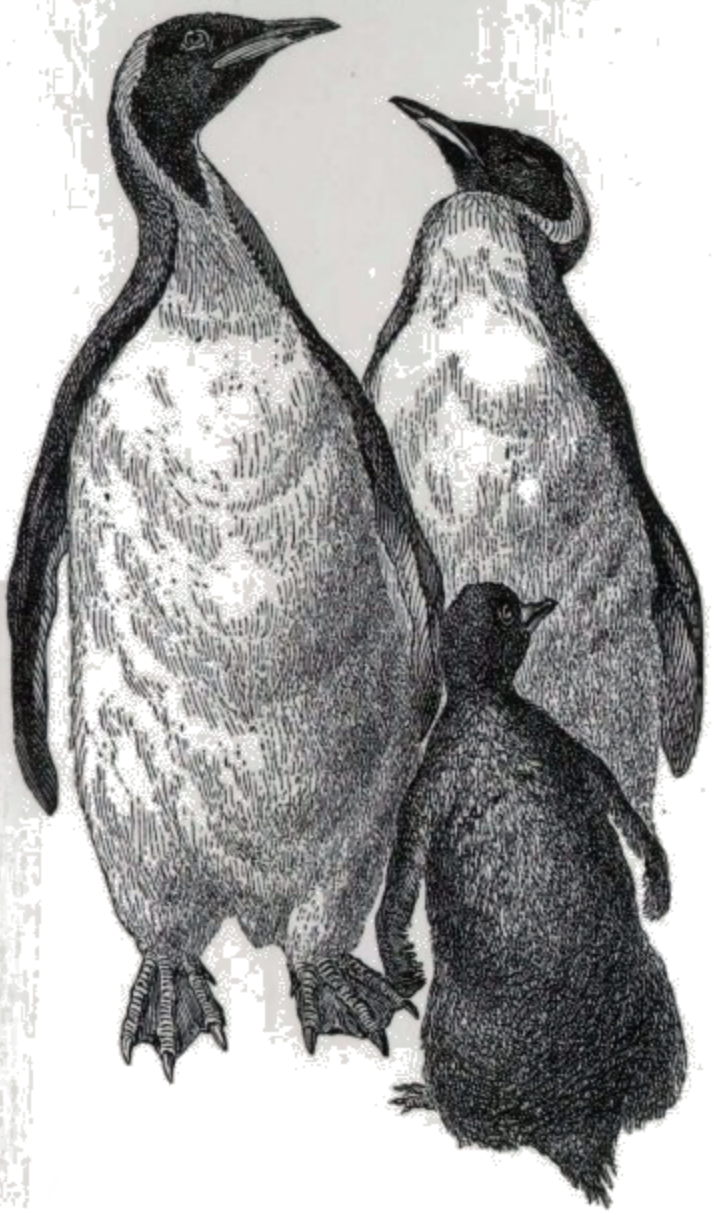
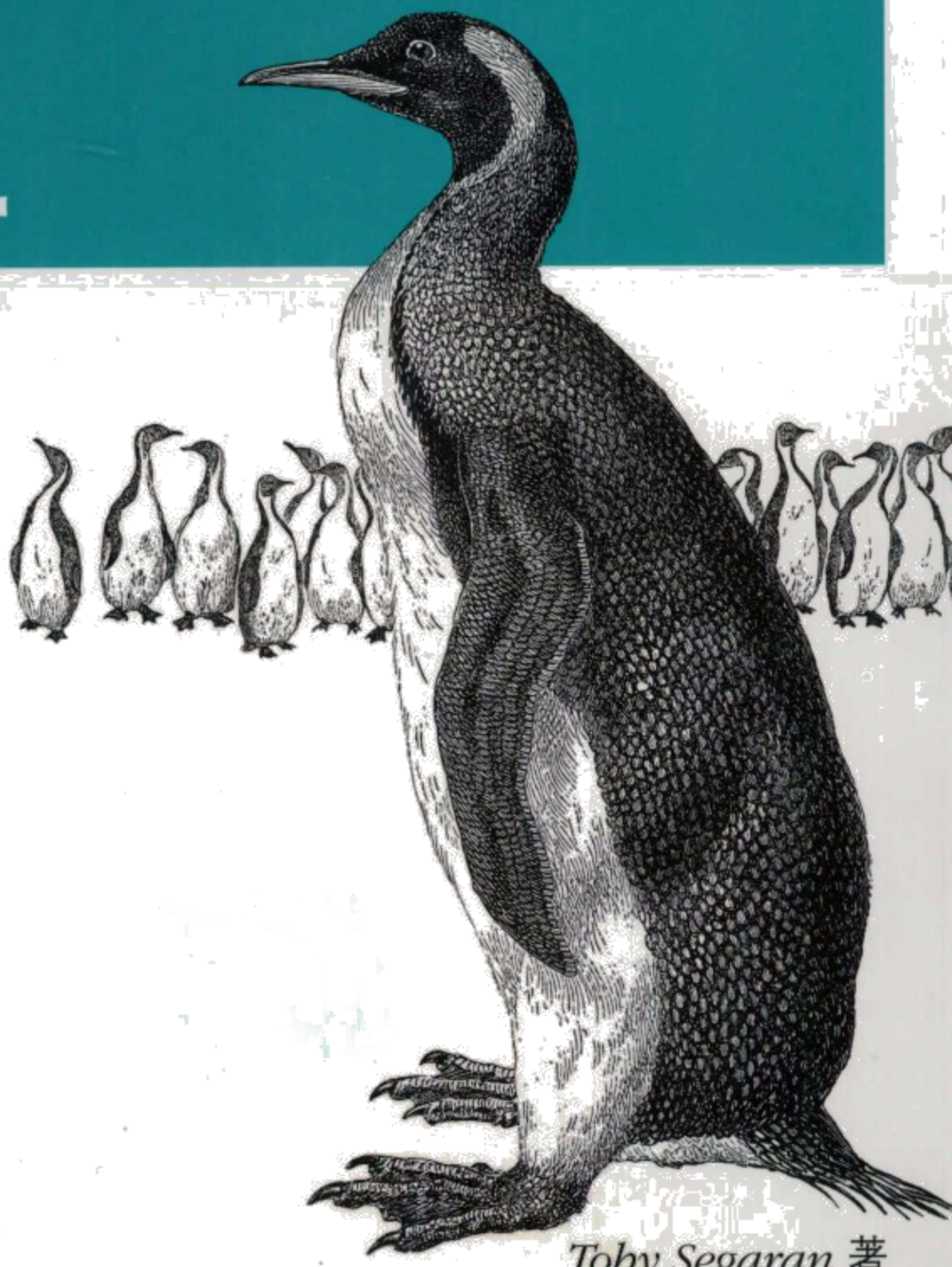
Programming Collective Intelligence

Building Smart Web 2.0 Applications

集体智慧编程





Toby Segaran 著

莫映 王开福 译

O'REILLY®

電子工業出版社

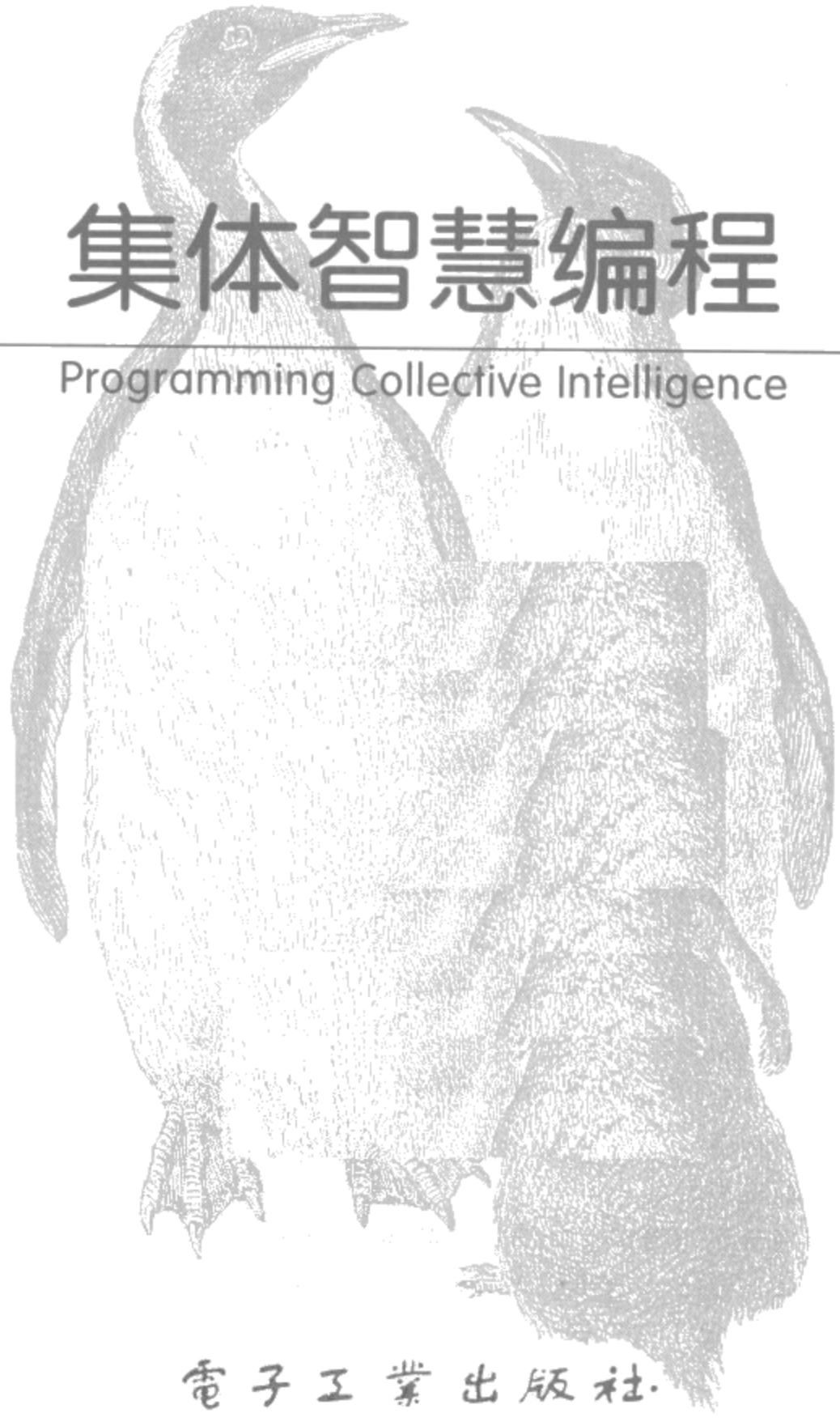
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

http://www.phei.com.cn

O'REILLY®

集体智慧编程

Programming Collective Intelligence



電子工業出版社·

Publishing House of Electronics Industry

北京 • BEIJING

内 容 简 介

本书以机器学习与计算统计为主题背景，专门讲述如何挖掘和分析 Web 上的数据和资源，如何分析用户体验、市场营销、个人品味等诸多信息，并得出有用的结论，通过复杂的算法来从 Web 网站获取、收集并分析用户的数据和反馈信息，以便创造新的用户价值和商业价值。全书内容翔实，包括协作过滤技术（实现关联产品推荐功能）、集群数据分析（在大规模数据集中发掘相似的数据子集）、搜索引擎核心技术（爬虫、索引、查询引擎、PageRank 算法等）、搜索海量信息并进行分析统计得出结论的优化算法、贝叶斯过滤技术（垃圾邮件过滤、文本过滤）、用决策树技术实现预测和决策建模功能、社交网络的信息匹配技术、机器学习和人工智能应用等。

本书是 Web 开发者、架构师、应用工程师等的绝佳选择。

版权贸易合同登记号 图字：01-2007-5378

图书在版编目（CIP）数据

集体智慧编程 /（美）西格兰（Segaran,T.）著；莫映，王开福译. —北京：电子工业出版社，2009.1

书名原文：Programming Collective Intelligence

ISBN 978-7-121-07539-1

Ⅰ.集… Ⅱ.①西…②莫…③王… Ⅲ.主页制作－程序设计 Ⅳ.TP393.092

中国版本图书馆 CIP 数据核字（2008）第 157645 号

责任编辑：王继花

项目管理：梁 晶

封面设计：Karen Montgomery 张 健

印 刷：北京智力达印刷有限公司

装 订：北京中新伟业印刷有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787×980 1/16 印张：22.75 字数：554 千字

印 次：2009 年 1 月第 1 次印刷

定 价：59.80 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。服务热线：（010）88258888。

O'Reilly Media, Inc.介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构 O'Reilly Media, Inc. 授权电子工业出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly Media, Inc. 是世界上在Unix、X、Internet和其他开放系统图书领域具有领导地位的出版公司，同时也是在线出版的先锋。

从最畅销的《The Whole Internet User's Guide & Catalog》(被纽约公共图书馆评为20世纪最重要的50本书之一)到GNN(最早的Internet 门户和商业网站)，再到 WebSite（第一个桌面 PC 的Web服务器软件），O'Reilly Media, Inc. 一直处于Internet发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc.是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc. 具有深厚的计算机专业背景，这使得O'Reilly Media, Inc. 形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc. 所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc.还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly Media, Inc.依靠他们及时地推出图书。因为 O'Reilly Media, Inc.紧密地与计算机业界联系着，所以O'Reilly Media, Inc. 知道市场上真正需要什么图书。

对本书的赞誉

Praise for *Programming Collective Intelligence*

“每年我都要审阅几本图书，自然而然地，在工作当中我阅读了大量的书籍。不得不承认，阅读本书让我获得了以前从未有过的、相当愉悦的阅读体验。太棒了！对于初学这些算法的开发者而言，我想不出有比这本书更好的选择了，而对于像我这样学过 AI 的老朽而言，我也想不出还有什么更好的办法能够让自己重温这些知识的细节。”

——Dan Russell，资深技术经理，Google

“Toby 的这本书非常成功地将机器学习算法这一复杂的议题拆分成了一个个既实用又易懂的例子，我们可以直接利用这些例子来分析当前网络上的社会化交互作用。我要是早两年读过这本书，就会省去许多宝贵的时间，也不至于走那么多的弯路了。”

——Tim Wolters，CTO，Collective Intellect

“本书获得了巨大的成功，它为大量相关数据的处理提供了非常丰富的计算方法。更重要的是，它将这些技术应用到了互联网上，而不是在一个个彼此孤立的数据孤岛中寻求价值。如果你是在为互联网开发应用，那么本书将是你的不二之选。”

——Paul Tyma，高级软件工程师，Google

译者序

还记得上个世纪 90 年代，当大学校园里的学子们还在为能够通过对等网在不同计算机间相互发送消息而兴奋不已的时候，互联网就已经悄然兴起了。很快，人们就从 C/S 时代跨入了 B/S 时代。我们不必再担心每次都要安装复杂的客户端程序，只要有浏览器，就会有绚丽多彩的舞台。然而随着时间的推移，人们又开始有所回归，大家不时地抱怨：为什么不能让浏览器像客户端应用那样具有丰富的表现？为什么每次打开链接都要傻傻地等着空白页面消失？直到有一天，Tim O'Reilly 向世人宣告了一个新的概念——Web 2.0。于是，忽如一夜春风来，大大小小的 Web 2.0 应用如雨后春笋般不断涌现，互联网又迈向了一个新的时代。

Web 2.0 使互联网变得异彩纷呈：来自不同地域的人们可以随时修改别人写过的文字，这就是维基；你有任何想法或观点都可以尽情地表达并欢迎别人评论，这就是博客；甚至连网页上出现的广告也都是与我们当前所关注的内容密切相关，这就是 Google AdSense……所有这一切，都带给我们不同于以往的全新感受。但是，这些应用究竟是怎样实现的？隐藏在它们背后的原理到底是什么？怎样让我们的 Web 2.0 程序变得更加聪明，更加贴心呢？译者相信，本书必定能够为大家逐一解开萦绕在心中的这些谜团。

本书以 Web 2.0 的核心价值观——集体智慧作为出发点，探讨了各种能够让 Web 2.0 程序变得更为智能的算法及其应用。这些算法大多来自机器学习和计算统计领域，其中的一些算法非常普及，而另一些则属于目前相当前沿的课题。它们包括了过滤器、聚类算法、支持向量机、遗传编程、优化技术，以及非常著名的 PageRank 算法，等等。将如此众多的优秀算法有效应用于互联网领域，并构造出具有智能特征的 Web 2.0 应用，应该是本书的一大亮点；同时，这也使本书有别于以往我们所见到过的任何一本纯粹介绍 Web 2.0 技术与概念的书籍。不仅如此，本书还提供了大量可供运行的示例代码，这些代码具有很好的复用性，只要稍加修改就可以用于实际的应用系统之中。书中代码还大量使用了许多时下流行的开放 API，这些 API 来自于 Yahoo!、eBay、FaceBook 等众多热门的 Web 2.0 网站，这使得本书在保有实用价值的同时又不失时效性。

本书的英文版虽只有寥寥 300 多页，比起任何一本大部头的技术书籍都是不足道的，但作为一本为数不多的深入讲解蕴藏于智能 Web 2.0 应用背后的算法原理的书籍，其深度和内涵却远远超出了篇幅的局限。为了尽量将原书的思想内涵以中文形式尽数表达出来，作为译者的我们在本书翻译期间着实不敢懈怠。在将书稿提交给出版社编辑之前，我们对每一章的译文都进行了不少于两遍的仔细校对。作为补充，中文版还随附了翻译期间译者所用的中英文术语对照表，希望本书中文版能够得到诸位读者的认可。

这本译作的完成是团队协作努力的结果，这包括了参与翻译、校审，以及关注和支持本书翻译的所有人。感谢博文视点的周筠老师对我们的信任，感谢本书的前后两位责编王凡毓与王晓菲，尤其是晓菲，她为本书的后期校审与编辑加工付出了辛劳，我们的合作非常愉快。此外，还要感谢李唯一，她为本书的前期翻译提供了诸多帮助。

由于译者水平所限，译文难免有疏误之处，欢迎读者批评指正。

为了便于读者阅读理解，特在此附上本书翻译过程中整理提取的中英文术语对照表。下表所包含的多为专业领域的技术术语。其中部分术语在不同的文献中往往有不同的译法。本书为了统一，选择了比较常见的译法，如 clustering 可译作“聚类”或“聚集”，此处我们选择了“聚类”。类似的还有 k-nearest neighbors、cross-product、dot-product，等等。

另一部分术语，虽有固定译法，但我们结合上下文，采用了更为贴切的翻译。如 computationally intensive 常被译为“计算密集的”，而在此处，我们采用“计算量很大的”。类似的还有 data-intensive、solution、crawl，等等。

此外还有一部分术语，在当下的中文文献中并没有明确的公认译法，因而我们在书中给出了参考翻译，以供大家商榷。如 collective intelligence 被译为“集体智慧”，list comprehension 被译为“列表推导式”，等等。

表 0-1：中英文术语对照表

英　文	中　文	英　文	中　文
clustering	聚类	collective intelligence	集体智慧
computationally intensive	计算量很大的	crawl	（网页）检索
cross-product	叉乘	data-intensive	数据量很大的
dendrogram	树状图	dot-product	点积
groups	群组	inbound link，incoming link	外部回指链接
kernel methods，kernel tricks	核方法，核技法	K-Means	K－均值
k-nearest neighbors	k－最近邻	list comprehension	列表推导式
multidimensional scaling	多维缩放	observations	观测数据，观测值
pattern	模式	similarity	相似度，相似性
solution	（题）解	vertical search engine	垂直搜索引擎

莫　映　　王开福

2008 年 9 月于北京

前言

Preface

无论是有意还是无意，越来越多投身于互联网的人们已经制造出了相当多的数据，这给了我们无数潜在的机会来洞悉用户体验、商业营销、个人偏好和通常所谓的人类行为（human behavior）。本书向大家介绍了一个新兴的领域，称为**集体智慧**（collective intelligence）。这一领域涵盖了诸多方法，借助这些方法我们可以从众多 Web 站点处（这些站点的名字或许你曾经有所耳闻）提取到值得关注的重要数据；借助这些方法我们还可以从使用自己应用程序的用户那里搜集信息，并对我们所掌握的数据进行分析和理解。

本书的目的是要带领你超越以数据库为后端的简单应用系统，并告诉你如何利用自己和他人每天搜集到的信息来编写更为智能的程序。

先决条件

Prerequisites

本书的代码示例是用 Python 语言编写的，因此熟悉 Python 编程将会有助于你对算法的理解，不过笔者给出了所有算法的解释说明，所以其他语言的程序员也能看懂。对于已经了解了像 Ruby 或 Perl 这样高级语言的程序员，Python 代码应该是非常容易理解的。本书的目的不是要作为一本学习编程的指导书，因此尤为重要的一点在于，为了熟悉基本概念，我们最好已编写过足够多的代码。如果懂得递归和一点点函数式编程（functional programming）的基本概念，那么我们就会发觉书中的内容是很容易理解的。

本书并不假设你已经具备了任何有关数据分析、机器学习或统计学方面的知识。笔者在尝试以尽可能浅显易懂的方式来解释数学概念，不过具备一点三角学和统计学的基本知识将会对你理解算法有所助益。

示例风格

Style of Examples

本书每一章节的代码示例都是以一种教程式的风格编写而成的，它鼓励你以循序渐进的方式来构建应用程序，并对算法的工作原理有一个深入的理解和认识。大多数情况下，写完一个新的函数或方法之后，我们会在一个交互的会话环境里使用它，以此来理解算法的工作原理。通常算法是有简单的变体的，我们可以用多种方式对其进行扩展。通过示例讲解并以交互的方式对其进行测试，我们对算法将会有更为深入的理解，从而可以对其进行改造，以适应自己的应用程序。

为何选择 Python

Why Python?

尽管书中的算法是伴随着对相关公式的解释，以文字形式加以描述的，但是假如有针对算法和示例问题的实际代码，那将会是更有助益的（而且有可能更易于理解）。本书中的所有示例代码都是用 Python，一种优秀的高级语言编写而成的。之所以选择 Python 是因为它有如下特性。

简练

使用像 Python 这样的动态类型语言编写的代码往往比用其他主流语言编写而成的代码更加简短。这意味着，在完成示例的过程中会有更少的录入工作，而且这也意味着我们将更容易记住算法并真正领会算法的原理。

易于阅读

Python 不时被人们指为“可执行的伪代码”。虽然很明显这是一种夸大之词，但是它表明，大多数有经验的程序员可以读懂 Python 代码并领会代码所要表达的意图。Python 中一些不是很显见的语言要素将会在后面的“Python 技巧”一节中加以解释。

易于扩展

Python 随附了许多标准库，这些库涉及数学函数、XML（扩展标记语言）解析，以及网页下载。本书中用到的非标准库，如 RSS（Really Simple Syndication）解析器和 SQLite 接口，则是免费的，很容易下载、安装和使用。

交互性

在学习示例的过程中，可以尝试执行我们编好的函数，而无须为此专门编写额外的程序，这一点是非常有价值的。Python 可以直接从命令行运行程序，它还有交互提示，允许我们键入函数调用、创建对象，并以交互的形式来对包进行测试。

多范式

Python 支持面向对象、过程式和函数式编程风格。机器学习算法千差万别，最为清晰

的做法是针对不同算法采用不同的范式。有时将函数作为参数传入很有用处，而有时我们则须要在对象中捕获状态。对于这两种方式，Python 均予以支持。

多平台和免费

Python 有一个针对所有主流平台的单一参考实现，并且它对所有平台都是免费的。本书中所列代码可以运行于 Windows、Linux 和 Macintosh 环境。

Python 技巧

Python Tips

对于有兴趣学习 Python 编程的初学者而言，笔者推荐大家阅读由 Mark Lutz 与 David Ascher 合著的《Learning Python》（O'Reilly），该书有对 Python 的全面论述。为了更为直观地表达算法或基础性概念，笔者在整本书中使用了一些 Python 特有的语法，但是其他语言的程序员应该会发现，Python 的代码相对而言还是较为容易掌握的。下面是为非 Python 程序员提供的一份快速概览。

列表和字典的构造函数

Python 有一组不错的基本类型，其中有两种类型在本书中被大量使用，它们分别是**列表**和**字典**。列表是由一组任意类型的值构成的有序列表，它由方括号构造而成：

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

字典是由一组名值对构成的无序集合，类似于其他语言中的 hash map。它由大括号构造而成：

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

可以通过序列名后跟方括号的形式来访问列表和字典中的元素：

```
string_list[2]   # returns 'c'
ages['Sarah']    # returns 28
```

有意义的空白字符

与大多数语言有所不同，Python 实际上是利用代码的缩进来定义代码块的。请看下列代码片段：

```
if x==1:
  print 'x is 1'
  print 'Still in if block'
print 'outside if block'
```

因为代码是被缩进的，所以语法解释器知道前两个打印语句会在 x 为 1 的时候被执行。缩进可以是任意数量的空格，只要它是常量即可。本书使用的缩进是两个空格。在输入代码的时候，我们须要注意正确拷贝缩进。

列表推导式

列表推导式（list comprehension）是一种方便简洁的语法形式，我们可以利用它将一个列表经过滤后转换成另一个列表，也可以利用它将函数应用于列表中的元素。列表推导式以如下形式书写：

```
[表达式 for 变量 in 列表]
```

或者：

```
[表达式 for 变量 in 列表 if 条件]
```

例如，下列代码：

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

将打印输出如下列表：

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

本书中频繁地使用了列表推导式，因为要将一个函数应用于整个列表，或是删除不需要的列表项时，这种表达方法非常简练。列表推导式的另一种常见用法是与 dict 构造函数结合在一起使用：

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

上述代码将会建立一个字典，以原先的列表作为键，以每个列表项乘以 10 作为值：

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

开放的 API

Open APIs

用于将集体智慧合成起来的算法需要来自许多用户的数据。除机器学习的算法外，本书还论及了许多开放的 Web APIs（应用编程接口）。这些 API 允许我们通过特殊的协议对来自相应 Web 站点的数据进行访问；我们可以编写程序将数据下载下来并加以处理。这些数据通常是由站点的使用者来提供的，我们可以从中挖掘出新的内涵来。有的时候，我们可以用现成的 Python 库来访问这些 API；而有时，如果没有现成的库，那么最为直接的做法莫过于创建自己的接口来访问数据，为此我们须要利用 Python 提供的内建库将数据下载下来，并对 XML 加以解析。

此处列出了一系列提供开放 API 的 Web 站点，我们将在本书中陆续接触到这些站点。

del.icio.us

一个社会型书签应用系统（social bookmarking application），其开放的 API 允许你根据标签（tag）或特定的用户来下载链接。

Kayak

一个提供 API 的旅游网站，你可以利用 API 在自己的程序中集成针对航班和旅馆的搜索。

eBay

一个提供 API 的在线交易站点，允许你查询当前正在出售的货品。

Hot or Not

一个评分与交友的网站，提供 API 对人员进行搜索，并获取其评分及个人资料。

Akismet

一种用于对协作型垃圾信息进行过滤的 API。

通过对来自单一源的数据进行处理，对来自多个源的数据进行组合，甚至通过将外部信息与自有系统的用户输入信息加以组合，我们可以构造出大量的潜在应用。对人们在不同网站以各种不同方式产生的数据加以充分利用的能力，便是构建集体智慧的一个基本要素。如果你想寻找更多的提供开放 API 的 Web 站点，不妨从访问 ProgrammableWeb 开始（*http://www.programmableweb.com*）。

各章概览

Overview of the Chapters

本书的每个算法都来源于某一现实的问题，希望这些问题能够很容易地被广大读者所理解。笔者将尝试尽量避开那些要求大量领域知识的问题，而将焦点集中在那些虽不失复杂性，但对大多数涉足者而言却又是简单易懂的问题上。

第 1 章，集体智慧导言

本章解释了蕴藏于机器学习背后的概念，并解释了如何将其应用于诸多不同的领域，以及如何利用它对搜集自许多不同人群的数据进行分析，并从中得出新的结论。

第 2 章，提供推荐

本章介绍**协作型过滤**（collaborative filtering）技术，这项技术被许多在线零售商用来向顾客推荐商品或媒体。本章中有一节介绍了如何向一个社会型书签服务网站的用户提供推荐链接，还介绍了如何根据 MovieLens 所提供的数据集构筑一个影片推荐系统。

第 3 章，发现群组

本章基于第 2 章中给出的某些观点，介绍了两种不同的**聚类方法**，利用这些方法，我们可以在一个大数据集中自动找出具有相似特征的群组。本章还演示了如何利用聚类算法从一组颇受欢迎的博客之中寻找群组，以及利用聚类算法根据某个社会型网站的用户意愿去寻找群组。

第 4 章，搜索与排名

本章描述了构成一个搜索引擎的各个不同组成部分，它们包括：爬虫程序（crawler）、索引程序（indexer），以及查询引擎（query engine）。本章介绍了以来自外部网站的链接信息为依据给网页打分的 *PageRank* 算法，还向你展示了如何构建**神经网络**，借此获知与不同结果相关联的关键词。

第 5 章，优化

本章介绍了**优化算法**，设计这些算法的目的，是为了对问题的数百万个可能的题解进行搜索，并从中选出最优解来。书中利用示例演示了这些算法的各种不同用法，包括：为一群去往相同地点的旅客寻找最佳航班，寻求为学生安排宿舍的最佳方案，以及给出交叉线数量最小的网络布局。

第 6 章，文档过滤

本章向读者演示了**贝叶斯过滤**，这一方法被广泛应用于许多免费的和商业的垃圾信息过滤系统中，用于根据单词类型及出现在文档中的其他特征对文档进行自动分类。我们将其应用于一组 RSS 搜索结果，以此来说明对内容项的自动分类过程。

第 7 章，决策树建模

本章介绍了**决策树**，我们不仅将它作为一种预测方法，而且还用它来为决策过程进行建模。本章中出现的第一棵决策树是根据假想的服务器日志数据构建而成的，我们利用它来预测用户是否有可能成为付费订户（premium subscriber）。本章的另一个例子则使用了来自真实 Web 站点的数据，用以对住房价格和来自 Hot or Not 网站的“热度（hotness）”评价进行建模。

第 8 章，构建价格模型

本章解决的是数值预测问题而非分类问题，期间用到了 **k-最近邻技术**，并且还用到了第 5 章中的优化算法。我们将这些方法与 eBay API 结合在一起构造出一个系统，能够根据拍卖品的一组属性，预测出最终的拍卖价格。

第 9 章，高阶分类：核方法与 SVM

本章向读者介绍了如何利用**支持向量机**（support-vector machines）对在线约会网站的用户进行匹配，以及如何将其用于针对专业交友网站的好友信息搜索。支持向量机是一项非常高阶的技术，本章将之与其他方法进行了对比。

第 10 章，寻找独立特征

本章介绍了一种相对较新的技术，称为**非负矩阵因式分解**（non-negative matrix factorization），我们利用这项技术在数据集中寻找独立的特征。对许多数据集而言，其中所包含的内容都是可以借助一组独立特征的组合被重新构造出来的，而这些特征是我们事先不知道的；非负矩阵因式分解的思路便是要寻找这些特征。在本章中，我们利用一组新闻报道说明了该项技术的使用；期间，通过新闻故事来寻找其中的主题，一篇给定的新闻故事中会包含一个或多个这样的主题。

第 11 章，智能进化

本章介绍了**遗传编程**（genetic programming）的概念，这是一组非常复杂的技术，它超出了优化的范畴。并且，这项技术实际上借鉴了进化的思想，它是通过自动构造算法的方式来解决特定问题的。我们通过一个简单的游戏来说明这项技术的应用。在游戏中，计算机最初只是一个学艺不精的初级选手，但是随着游戏的不断进行，它会通过逐步改进其所拥有的代码来提升自己的技能。

第 12 章，算法总结

本章回顾了书中所讲述的所有机器学习算法及统计算法，并将它们与一组人为设计的问题做了对比。这将有助于我们理解算法的工作原理，并形象地说明每种算法划分数据的方法。

附录 A，第三方函数库

给出了有关本书所用的第三方库的信息，例如在哪里可以找到这些第三方库，以及如何进行安装。

附录 B，数学公式

包含了一部分数学公式及其说明，以及本书通篇引入的、以代码形式描述的诸多数学概念。

位于每章末尾处的练习，为读者提供了许多相关信息，借此我们可以对算法进行扩展并使其变得更加强大。

排版约定

Conventions

本书使用了下列排版约定。

普通字体

用于指示菜单标题、菜单选项、菜单按钮，以及键盘快捷键（诸如 Alt 和 Ctrl）。

斜体（*Italic*）

用于指示新的术语、URL、E-mail 地址、文件名、文件扩展名、路径名、目录，以及 Unix 工具。

等宽字体（`Courier New`）

用于指示命令（commands）、命令选项（options）、命令开关（switches）、变量（variables）、属性（attributes）、键值（keys）、函数（functions）、类型（types）、类（classes）、名字空间（namespaces）、方法（methods）、模块（modules）、成员属性（properties）、参数（parameters）、值（values）、对象（objects）、事件（events）、事件处理器（event handlers）、XML 标签、HTML 标签、宏、文件内容或命令的输出结果。

等宽粗体（`Courier New`）

用于显示命令或其他应该由用户手工逐字输入的文本。

等宽斜体（*`Courier New`*）

用于显示可替换文本，这些文本应该被替换成用户提供的内容。

该图标代表了提示、建议，或者一般性注释。

使用示例代码

Using Code Examples

本书旨在帮助你完成手头的工作。一般而言，你可以在自己的程序和文档中随意使用书中代码。除非原样引用大量的代码，否则你无须征得我们的许可。例如，在编写程序时引用了本书的若干代码片段是无须许可的。而销售或发行 O'Reilly 图书的示例光盘则是须要许可的。通过引用书中内容及示例代码的方式来答疑解难是无须许可的。而将书中的大量示例代码加入到你的产品文档中则是须要许可的。

如果你在引用时注明出处，我们将不胜感激，但是这并非必需。引用通常包含了标题、作者、出版商，以及 ISBN 号。例如："Programming Collective Intelligence by Toby Segaran. Copyright 2007 Toby Segaran, 978-0-596-52932-1"。

如果你发现自己对示例代码的使用有失公允或违反了上述条款，敬请通过 *permissions@oreilly.com* 与我们取得联系。

如何联系我们

How to Contact Us

如果你想就本书发表评论或有任何疑问，敬请联系出版社：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

奥莱理软件（北京）有限公司
北京市 西城区 西直门南大街 2 号 成铭大厦 C 座 807 室
邮政编码：100055
网页：*http://www.oreilly.com.cn*
E-mail：*info@mail.oreilly.com.cn*

与本书有关的在线信息如下所示:
http://www.oreilly.com/catalog/9780596529321（原书）
http://www.oreilly.com.cn/book.php?bn=978-7-121-07539-1（中文版）

如果你想就本书发表评论或提问技术问题，请发送 E-mail 至：
bookquestions@oreilly.com

北京博文视点资讯有限公司（武汉分部）
湖北省 武汉市 洪山区 吴家湾 邮科院路特 1 号 湖北信息产业科技大厦 1402 室
邮政编码：430074
电话：（027）87690813　　传真：（027）87690595
网页：*http://bv.csdn.net*
读者服务信箱：
reader@broadview.com.cn（读者信箱）
bvtougao@gmail.com　　（投稿信箱）

致谢

Acknowledgments

我想对每一位曾经参与本书编写与出版的 O'Reilly 工作人员表达我的感谢之情。首先，我要感谢 Nat Torkington，是他告诉我写书这个想法值得一试，我还要感谢 Mike Hendrickson 和 Brian Jepson，是他们听了我的絮叨并令我兴致勃勃地撰写本书，我尤其要感谢 Mary O'Brien，他接手 Brian 的编辑工作，并且总是能够缓解我对工程浩大的担忧。

在出版团队中，我要感谢 Marlowe Shaeffer、Rob Romano、Jessamyn Read、Amy Thomson 和 Sarah Schneider，是他们将我的插图与书稿编辑成您实际想要看到的形式。

感谢每一位参与本书校审工作的人，尤其是 Paul Tyma、Matthew Russell、Jeff Hammerbacher、Terry Camerlengo、Andreas Weigend、Daniel Russell 和 Tim Wolters。

感谢我的父母。

最后，万分感谢我的几位朋友，他们帮助我利用头脑风暴形成了有关本书的一些想法，并且总能对我无暇陪伴他们表示理解：Andrea Matthews、Jeff Beene、Laura Miyakawa、Neil Stroup 和 Brooke Blumenstein。要是没有你们的支持，写作本书会更为艰难，我必会漏掉一些更为有趣的示例。

关于作者

Toby Segaran 是 Genstruct 公司的软件开发主管，这家公司涉足计算生物领域，他本人的职责是设计算法，并利用数据挖掘技术来辅助了解药品机理。Toby Segaran 还为其他几家公司和数个开源项目服务，帮助它们从收集到的数据当中分析并发掘价值。除此以外，Toby Segaran 还建立了几个免费的网站应用，包括流行的 tasktoy 和 Lazybase。他非常喜欢滑雪与品酒，其博客地址是 *blog.kiwitobes.com*，现居于旧金山。

关于封面

本书封面上的动物是王企鹅（Aptenodytes patagonicus）。尽管其命名与巴塔哥尼亚（Patagonia）地区有关，但是王企鹅却并非产于南美洲；它们在那里的最后一片栖息地早在 19 世纪就已经被海豹狩猎者给摧毁了。如今，这些企鹅分布在次南极群岛一带，如爱德华王子（Prince Edward）、克罗泽特（Crozet）、麦格理（Macquarie），以及福克兰群岛（Falkland Islands）等地。它们居住在海滨及靠近大海的地势平坦的冰川地区。王企鹅完全是一种群居性的鸟类；在它们的繁殖地，种群的数量多达 1 万，而且它们时常聚集在一起饲养幼鸟。

王企鹅站立时有 76.2 厘米（30 英寸）高，体重达到 13.6 千克（30 磅），它们是企鹅家族中体型最大的种群之一——仅次于其近邻帝企鹅。除了体型以外，王企鹅还有一个主要的识别特征，那就是位于其头部的鲜橙色斑点，这些斑点一直向下延伸到其胸部的银白色羽毛处。王企鹅身形圆滑，并且不像帝企鹅那样只会在陆地上跳跃，它们还可以奔跑。王企鹅很习惯于海洋生活，它们以鱼类和乌贼为食，并且可以向下潜到 213.36 米（700 英尺）的深度，比其他大多数企鹅潜得还要深。由于雄性和雌性在体型和外观上都非常接近，因此人们一般根据它们的行为迹象（比如交配行为）对其加以区分的。

王企鹅并不筑巢；相反，它们会将唯一的一枚卵塞入肚皮下面，并放在两脚的蹼上。没有任何其他鸟类的繁殖周期会比王企鹅的还要长，这些企鹅每三年繁殖两次，并且每次只孵化一只幼鸟。王企鹅的幼鸟身型肥胖呈褐色，浑身毛茸茸的，以至于早期的探险者们以为这是与王企鹅完全不同的另一类企鹅，并将其称为“woolly penguins（意指毛茸茸的企鹅）”。王企鹅在全世界范围内的数量有 200 万对，它并不属于濒危物种，世界自然保护联盟已将其列入了无危物种。

本书封面的图片取自 J.G. Wood 的《Animate Creation》。

目录

Table of Contents

第 1 章

集体智慧导言

Introduction to Collective Intelligence

Netflix 是一家在线的 DVD 租赁公司，这家公司允许人们在线选购影片，并由公司负责送货上门；另外，Netflix 还会根据顾客以往的租片信息为其提供相应的推荐。2006 年底，该公司宣布将 100 万美元的奖金奖励给第一位能够将其推荐系统的精确度提升至 10%的人，并且只要该项竞赛还在进行，公司每年就会将 5 万美元的进步奖授予当时的头名状元。于是，数千个来自世界各地的团队蜂拥而至，截止 2007 年 4 月为止，领先的团队已经成功取得了 7%的成绩。Netflix 的推荐系统可以利用每位老顾客对影片的喜好数据，为以前从未访问过该网站的其他顾客提供推荐，并保证他们能够再次光顾网站。从 Netflix 的角度而言，无论对推荐系统采取任何形式的改善，都会为它带来大笔的收入。

搜索引擎公司 Google 始创于 1998 年，其时已有数家大型搜索引擎公司存在，而且许多人都认为，一家新兴的小公司是绝不可能撼动业界巨人的。然而，Google 的创立者们采用了一种全新的方法对搜索结果进行排序——它们利用上百万个 Web 站点上的链接来决定哪些页面的相关性最大。Google 的搜索结果远远优于其他同行，以至于到了 2004 年，它已占据了 85%的 Web 搜索市场。当初的创业元老们现在也跻身了世界 10 大富豪之列。

上述这两家公司有何共同之处呢？它们都使用了先进的算法，将来自不同人群的数据加以组合，进而得出新的结论，并创造出新的商机。这种信息采集能力，以及对其加以解释的计算能力已经激发起了很多巨大的协作型商机，并且加深了对用户和顾客更好的理解。这样的例子现在比比皆是——约会网站希望帮助人们更快地找到他们的最佳拍档，预测机票价格变化的公司如雨后春笋般不断涌现，为了创造更有针对性的广告，几乎每个人都想更好地了解他们的顾客。

上面提到的，仅仅是集体智慧（collective intelligence）这一令人振奋的新兴领域中少数几个典型的例子，层出不穷的新服务意味着每天都会有新的商机涌现出来。笔者相信，理解机器学习和统计方法在许多不同领域里都会变得愈加重要，而这一点在针对海量信息的解释和组织方面尤为突出（全世界的人们正在不断创造这些信息）。

什么是集体智慧
What Is Collective Intelligence?

人们使用**集体智慧**这一术语已有十多年之久，随着新型通信技术的出现，这一术语也变得日趋流行和重要。尽管这样的表达也许会让人联想到群体意识或超自然现象，但当技术人员使用这一词汇时，其含义通常是指：为了创造新的想法，而将一群人的行为、偏好或思想组合在一起。

当然，集体智慧的出现可能要早于 Internet。为了从全无关系的一群人中搜集、组合和分析数据，我们不一定要借助于 Web。完成这项工作的一种最为基础的方法，便是使用调查问卷或普查。从一大群人中搜集的答案可以使我们得出关于群组的统计结论：组中的个体成员将会被忽视。从独立的数据提供者那里得出新的结论，是集体智慧所真正关注的。

这里有一个众所周知的例子，是关于金融市场的。在金融市场里，价格并不是由某个个体或某种协作力量所决定的，它是由许多独立个体的交易行为所共同决定的，所有人的行为都建立在这样一种信念基础之上：他们相信当前的交易会为他们带来最大的利益。尽管乍一看这似乎违背直觉，但在**未来的市场**上，大量的参与者都是根据他们对未来价格的信心而进行契约交易的，这样的市场在价格预测的效果方面，往往被认为要比独立进行预测的专家们表现得更好。这是因为，市场将知识、经验和成百上千人的意志组织在一起，形成了一种不依赖个人观点的预测。

尽管寻求集体智慧的方法在 Internet 之前就已经存在，但自从有了 Internet 之后，从数千甚至数百万网民中搜集信息的能力为人们提供了许多新的可能。一直以来，人们都在利用 Internet 来购买所需、搜索信息、寻求娱乐，以及架设自己的 Web 站点。所有这些行为都可以得到监控，并且不必要求用户放下手头的工作来接受询问，而可以借由监控得到的信息提取出有价值的结论。有大量的方法可以用来对这些信息进行加工和解释。这里有两个重要的例子，分别体现了两种彼此对立的做法。

- *Wikipedia* 是一个在线的百科全书，它完全是由用户维护的。任何人都可以新建或编辑网站上的任何一个页面，同时会有为数不多的几名管理员对一再出现的不当内容进行监控。Wikipedia 拥有的词条比其他任何百科全书还要多，尽管存在一些恶意用户的操作，但是人们普遍认为，Wikipedia 的大多数主题都是准确的。这便是集体智慧的一

个例子：因为每一篇文章都有大量人员在维护。而其最终的结果，则形成了一个任何单一协作团队都无法企及的大型百科全书。Wikipedia 软件并没有对用户贡献的内容进行特殊的智能处理，它只跟踪内容的变更情况，并显示最新的版本。

- 前文提及的 *Google*，是世界上最为流行的 Internet 搜索引擎，也是第一个根据其他网页对当前网页的引用数多少来评价网页等级的搜索引擎。这种评价等级的方法，搜集了数以千计的人对某一页面的评价信息，然后利用这些信息对搜索结果进行排序。这是集体智慧的一个非同寻常的例子。Wikipedia 明确邀请网站的用户提供内容，而 Google 则是从 Web 内容的创建者对自己网站的操作中提取重要的信息，并利用这些信息为 Google 的使用者设定各个网站的分值。

虽然 Wikipedia 是一个巨大的资源库，而且也是展现集体智慧的一个令人印象深刻的例子，但它的存在很大程度上要归功于提供内容的用户，而非软件中的那些智能算法。本书的焦点并不在于提供内容的用户，而在于算法，这其中就包括了 Google 的 PageRank 算法，该算法会搜集用户的数据，对数据进行计算分析，并从中创造出可以增强用户体验的新信息。在获得的这些数据当中，有一部分是明确搜集而来的，比如向用户询问与评价网页级别相关的问题。另一部分则是偶然搜集得到的，比如观察用户的购买行为。对于这两种情况，重要的不仅是搜集和显示信息，还包括以一种智能化的方式对这些信息加以处理，并产生出新的信息来。

本书将告诉你如何利用开放的 API 来搜集数据，同时还会讨论到各种机器学习算法和统计方法。将二者结合起来，就可以借助集体智慧的相关方法，对由自己编写的应用程序搜集得到的数据进行分析；同时，也可以从其他地方搜集数据，并对数据进行试验。

什么是机器学习

What Is Machine Learning?

机器学习是人工智能（AI，artificial intelligence）领域中与算法相关的一个子域，它允许计算机不断地进行**学习**。大多数情况下，这相当于将一组数据传递给算法，并由算法推断出与这些数据的属性相关的信息——借助这些信息，算法就能够预测出未来有可能会出现的其他数据。这种预测是完全有可能的，因为几乎所有的非随机数据中，都会包含这样或那样的“模式（patterns）”，这些模式的存在使机器得以据此进行归纳。为了实现归纳，机器会利用它所认定的出现于数据中的重要特征对数据进行“训练”，并借此得到一个**模型**。

为了理解模型得到的过程，我们来看另外一个复杂领域——电子邮件过滤中的一个简单例子。假定我们收到了大量包含“online pharmacy”单词的垃圾邮件。对人而言，我们可以很轻松地识别出其中的模式，并快速确知任何含有“online pharmacy”单词的信息都是垃圾邮

件，应该将其直接移到垃圾箱中。这就是归纳——事实上，我们已经建立起了一个关于垃圾邮件的智力模型。当我们将多条这样的信息报告为垃圾邮件之后，专门设计用以过滤垃圾邮件的机器学习算法应该有能力做出同样的归纳来。

有许多不同的机器学习算法，所有算法都各有所长，适应于不同类型的问题。有些算法，比如决策树，非常的直观，通过眼睛观察就可以完全理解机器执行的推导过程。另有一些算法，比如神经网络，则像一个**黑盒**，它们虽然也给出最终的结果，但通常要复现蕴含在这些结果背后的推导过程则是非常困难的。

许多机器学习算法都很倚仗数学和统计学。根据笔者早些时候给出的定义，我们甚至可以认为，简单的相关性分析和回归都是机器学习的基本形式。本书并没有假定读者具备许多统计学方面的知识，所以笔者会尝试尽可能直观地解释所用到的统计学知识。

机器学习的局限
Limits of Machine Learning

机器学习并非没有缺点。机器学习算法受限于其在大量模式之上的归纳能力，而一个模式如果不同于算法先前所曾见到过的任何其他模式，那么它很有可能会被“误解”。人类拥有大量的文化知识及经验可以借鉴；不仅如此，人们还具备一种非凡的能力，即：当对新的信息进行决策时，人们能够从中识别出相似的信息来，而机器学习方法却只能凭借已经见过的数据进行归纳，而且归纳的方式受到很大的限制。

我们将在本书中见到的垃圾邮件过滤方法，是以单词或单词组合的出现为依据的，至于这些单词的含义及句式结构，则根本未予考虑。尽管在理论上，构造一个考虑语法的算法是可行的，但在现实中却很少这样做，这是因为为此付出的努力与算法的改进相比很不成比例。理解单词的含义及单词与个人生活的相关性所要求掌握的信息，远比垃圾邮件过滤算法中所能访问到的现有信息还要多。

另外，尽管在解决问题的倾向性上各有不同，但是所有机器学习算法都有过度归纳的可能性。就如生活中的大多数事情一样，基于少数示例的强归纳很少是完全精确的。我们的确有可能会收到友人寄来的一封重要邮件，里面包含“online pharmacy”的字样。在这种情况下，我们须要告诉算法这不是垃圾邮件，或许算法可以作出判断，将来自某位好友的邮件判定为可以接收。究其本质，许多机器学习算法在新信息到来之时都是能够持续进行学习的。

真实生活中的例子

Real-Life Examples

当前 Internet 上有大量站点正在不断地从广大用户当中搜集数据，并利用机器学习和统计方法从中获益。Google 是其中的佼佼者——它不仅可以利用 Web 链接对网页进行排名，而且当其广告被不同的用户点击时，它会持续搜集信息，这使得 Google 可以更加有效地进行广告定位。在第 4 章中，我们将了解到搜索引擎和 PageRank 算法，这是 Google 排名系统的重要组成部分。

其他的例子还包括带有推荐系统的 Web 站点。如 Amazon 和 Netflix 这样的站点，它们利用人们的购买或租赁信息来确定人或物品的相似程度，然后再根据买卖历史来给出推荐。另有一些站点，比如 Pandora 和 Last.fm，则可以利用我们对不同乐队和歌曲的评价来建立定制的广播电台，其中包含了网站认为我们会喜欢的音乐。第 2 章将会讨论构建推荐系统的方法。

市场预测也是集体智慧的一种形式。这其中最为有名的一个例子莫过于 Hollywood Stock Exchange（*http://hsx.com*），在那里人们可以进行涉及影片和影星的模拟股票交易。我们可以按照影片的当前价格买卖股票，其对应的价值相当于电影实际首次票房收入的百万分之一。因为价格是通过交易行为来设定的，所以价值不由任何一个个体所决定，而是由群体的行为来确定的，股票的当前价格可以看作是整个群体对电影票房收入数字的预测。通常而言，由 Hollywood Stock Exchange 所给出的预测往往要优于某位专家所给出的预测。

某些交友网站，比如 eHarmony，利用从参与者那里搜集而来的信息确定交友的最佳配对。尽管这些公司对他们所采用的匹配算法守口如瓶，但是任何一种成功的匹配算法很可能都会涉及一个持续不断的求值过程——算法会反复判断选定的匹配成功与否。

学习型算法的其他用途

Other Uses for Learning Algorithms

本书所介绍的并不是什么新方法，尽管这些例子都是在讨论基于 Internet 的集体智慧的相关问题，但是掌握机器学习算法的知识对于许多其他领域的软件开发者而言也是很有助益的。尤其是在须要处理大量数据的领域里，我们可以从中发掘出值得关注的如下各种模式。

生物工艺学

人类在测序技术和筛选技术（sequencing and screening technology）上的进步已经创造出了许多不同种类的海量数据，比如 DNA 序列、蛋白质结构、化合物筛选及 RNA 表

达。为了找到能进一步理解生物进程的模式，机器学习技术被广泛应用于所有这些类型的数据之中。

金融欺诈侦测

信用卡公司一直都在寻找侦测交易是否存在欺诈行为的新方法。最终，他们使用了像神经网络和归纳逻辑这样的技术，对交易行为进行检验，并捕获不正当的使用方法。

机器视觉

出于军事或监控的目的，从摄像机中进行图片解析是一个活跃的研究领域。许多机器学习技术被用来自动侦测入侵者、辨别车辆，或者识别人脸。尤其值得注意的是无人监控技术的使用，比如能从大数据集中发现有趣特征的**独立组元分析技术**。

产品市场化

长期以来，对人口统计资料及其发展趋势的理解被认为是一种艺术而不是科学。最近，人们在消费者数据搜集能力方面的增长，为机器学习技术打开了机会之门，比如聚类方法，就能很好地理解存在于市场中的自然划分，并能更好地预测未来的趋势。

供应链优化

许多企业通过其供应链的有效运行及精确预测不同区域的产品需求，来节省数以百万计的成本投入。构造供应链的方法非常多，影响需求的潜在因素也非常多。优化和学习技术时常被用来分析这些数据集。

股票市场分析

自从有了股票市场，人们就一直在尝试利用数学方法来赚取更多的钱。随着参与股市的股民变得越来越有经验，对大量数据进行分析并采用先进技术来侦测模式已经变得很有必要了。

国家安全

全世界的政府机构都在搜集海量信息，对这些数据的分析过程要求计算机对模式进行检测，并将之与潜在的威胁联系起来。

上述这些仅仅是人们现在大量使用机器学习的典型个案。既然有越来越多的信息被制造出来已是大势所趋，那么有越来越多的领域将依赖于机器学习和统计技术并不是没有可能的，因为信息扩张的规模已经超出了人们利用旧有方法进行处理的能力。

每天可以获得的新信息有多少，显然就会有多少更多的可能性。一旦你掌握了一点机器学习的算法，你就会发现它们的应用随处可见。

第 2 章

提供推荐

Making Recommendations

为了开始集体智慧之旅，本章即将告诉大家，如何根据群体偏好来为人们提供推荐。有许多针对于此的应用，如：在线购物中的商品推荐、热门网站的推荐，以及帮助人们寻找音乐和影片的应用。本章将告诉你如何构筑一个系统，用以寻找具有相同品味的人，并根据他人的喜好自动给出推荐。

也许在使用如 Amazon 这样的在线购物网站之前，你已经接触过某些推荐类引擎了。Amazon 会对所有购物者的购买习惯进行追踪，并在你登录网站时，利用这些信息将你可能会喜欢的商品推荐给你。Amazon 甚至还能够向你推荐你可能会喜欢的影片，即便你此前也许只从该网站购买过书籍。还有一些在线的音乐会售票代理站点，它们会查看你以前观看演出的历史，并提醒你即将到来的演出，说不定这些演出是值得一看的。又比如像 *reddit.com* 这样的站点，它会让你对其他 Web 站点的链接进行投票，然后利用投票结果推荐你也许会感兴趣的其他链接。

从这些例子中，你可以看到，我们能够使用许多不同的方式来搜集兴趣偏好。有时候，这些数据可能来自于人们购买的物品，以及有关这些物品的评价信息，这些评价可能会被表达成“是/否”之类的投票表决，或者是从 1 到 5 的评价值。本章中，我们将对这些形色各异的表达方法进行考查，以便能够利用同一组算法对其进行处理，同时还将建立几个涉及电影评分和社会化书签的可运行的例子。

协作型过滤

Collaborative Filtering

我们知道，要想了解商品、影片或娱乐性网站的推荐信息，最没有技术含量的方法莫过于向朋友们询问。我们也知道，这其中有一部分人的品味会比其他人的高一些，通过观察这些人是否通常也和我们一样喜欢同样的东西，可以逐渐对这些情况有所了解。不过随着选

择越来越多，要想通过询问一小群人来确定我们想要的东西，将会变得越来越不切实际，因为他们可能并不了解所有的选择。这就是为什么人们要发展出一套被称为**协作型过滤**（collaborative filtering）的技术。

一个协作型过滤算法通常的做法是对一大群人进行搜索，并从中找出与我们品味相近的一小群人。算法会对这些人所偏爱的其他内容进行考查，并将它们组合起来构造出一个经过排名的推荐列表。有许多不同的方法可以帮助我们确定哪些人与自己的品味相近，并将他们的选择组合成列表。本章将择其一二详加介绍。

术语：“协作型过滤”是 David Goldberg 1992 年在施乐帕克研究中心（Xerox PARC）的一篇题为《Using collaborative filtering to weave an information tapestry》的论文中首次使用的。他设计了一个名叫 *Tapestry* 的系统，该系统允许人们根据自己对文档感兴趣的程度为其添加标注，并利用这一信息为他人进行文档过滤。

时下，有数以百计的 Web 站点都在采用这样那样的协作型过滤算法，这些算法所要处理的内容涉及电影、音乐、书籍、交友、购物、网站、播客服务（podcast）、文章，甚至还有幽默笑话。

搜集偏好

Collecting Preferences

我们要做的第一件事情，是寻找一种表达不同人及其偏好的方法。在 Python 中，达到这一目的的一种非常简单的方法是使用一个**嵌套的字典**。如果你打算运行本节中的示例，请新建一个名为 *recommendations.py* 的文件，并加入如下代码来构造一个数据集：

```
# 一个涉及影评者及其对几部影片评分情况的字典
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}}
```

本章中，我们将以交互方式使用 Python，因此应该先将 *recommendations.py* 保存起来，以便 Python 的交互解释程序能够读取到它。我们也可以将文件保存在 *python/Lib* 目录下，不过最为简单的做法，是在与我们保存文件的同一目录下启动 Python 解释程序。

上述字典使用从 1 到 5 的评分，以此来体现包括本人在内的每位影评者对某一给定影片的喜爱程度。不管偏好是如何表达的，我们需要一种方法来将它们对应到数字。假如我们正在架设一个购物网站，不妨用数字 1 来代表有人过去曾购买过某件商品，用数字 0 来代表未曾购买过任何商品。而对于一个新闻故事的投票网站，我们可以分别用数字-1、0 和 1 来表达“不喜欢”、“没有投票”、“喜欢”，如表 2-1 所示。

表 2-1：从用户行为到相应评价值的可能对应关系

音乐会门票		在线购物		网站推荐者	
已购买	1	已购买	2	喜欢	1
未购买	0	已浏览	1	未投票	0
		未购买	0	不喜欢	-1

对于算法试验和范例演示而言，使用字典是很方便的。我们可以很容易地对字典进行查询和修改。请启动 Python 的解释程序，并试着输入下列几行命令：

```
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, 'Superman Returns': 4.0, 'You, Me and Dupree':1.0}
```

尽管可以将相当数量的人员偏好信息置于字典内（即内存中），但对于一个规模巨大的数据集而言，也许我们还是会希望将其存入数据库中。

寻找相近的用户

Finding Similar Users

搜集完人们的偏好数据之后，我们须要有一种方法来确定人们在品味方面的相似程度。为此，我们可以将每个人与所有其他人进行对比，并计算他们的**相似度评价值**。有若干种方法可以达到此目的，本节中我们将介绍两套计算相似度评价值的体系：**欧几里德距离**和**皮尔逊相关度**。

欧几里德距离评价

Euclidean Distance Score

计算相似度评价值的一个非常简单的方法是使用欧几里德距离评价方法。它以经过人们一致评价的物品为坐标轴，然后将参与评价的人绘制到图上，并考查他们彼此间的距离远近，如图 2-1 所示：

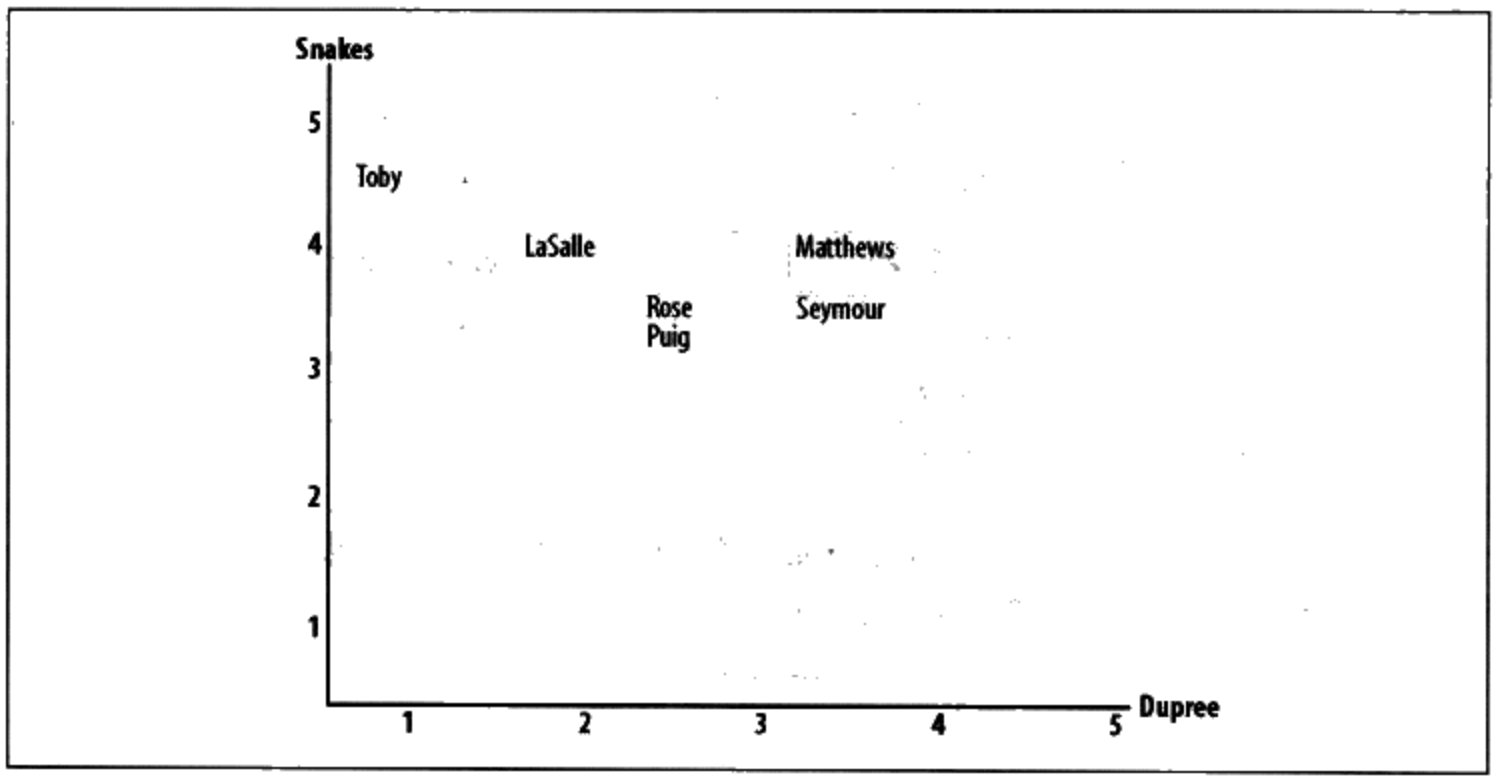


图 2-1：处于“偏好空间”中的人们

该图显示了处于“**偏好空间**”中人们的分布状况。Toby 在 Snakes 轴线和 Dupree 轴线上所标示的数值分别是 4.5 和 1.0。两人在“偏好空间”中的距离越近，他们的兴趣偏好就越相似。因为这张图是二维的，所以在同一时间内你只能看到两项评分，但是这一规则对于更多数量的评分项而言也是同样适用的。

为了计算图上 Toby 和 LaSalle 之间的距离，我们可以计算出每一轴向上的差值，求平方后再相加，最后对总和取平方根。在 Python 中，我们可以利用函数 `pow(n,2)` 对某数求平方，并使用 `sqrt` 函数求平方根：

```
>> from math import sqrt
>> sqrt(pow(4.5-4,2)+pow(1-2,2))
1.1180339887498949
```

上述算式可以计算出距离值，偏好越相似的人，其距离就越短。不过，我们还需要一个函数，来对偏好越相近的情况给出越大的值。为此，我们可以将函数值加 1（这样就可以避免遇到被零整除的错误了），并取其倒数：

```
>> 1/(1+sqrt(pow(4.5-4,2)+pow(1-2,2)))
0.47213595499957939
```

这一新的函数总是返回介于 0 到 1 之间的值，返回 1 则表示两人具有一样的偏好。我们将前述知识结合起来，就可以构造出用来计算相似度的函数了。请将下列代码加入 *recommendations.py*：

```
from math import sqrt

# 返回一个有关person1与person2的基于距离的相似度评价
def sim_distance(prefs,person1,person2):
  # 得到shared_items的列表
  si={}
  for item in prefs[person1]:
    if item in prefs[person2]:
      si[item]=1

  # 如果两者没有共同之处，则返回0
  if len(si)==0: return 0

  # 计算所有差值的平方和
  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+sqrt(sum_of_squares))
```

我们可以调用该函数，分别传入两个人的名字，并计算出相似度的评价值。在你的 Python 解释器里执行如下命令：

```
>>> reload(recommendations)
>>> recommendations.sim_distance(recommendations.critics,
...   'Lisa Rose','Gene Seymour')
0.29429805508554946
```

上述执行过程给出了 Lisa Rose 和 Gene Seymour 之间的相似度评价。请用其他人的名字试一试，看看我们是否能够找到或多或少具有一定共性的人。

皮尔逊相关度评价

Pearson Correlation Score

除了欧几里德距离，还有一种更复杂一些的方法可以用来判断人们兴趣的相似度，那就是皮尔逊相关系数。该相关系数是判断两组数据与某一直线拟合程度的一种度量。对应的公式比欧几里德距离评价的计算公式要复杂，但是它在数据不是很规范（normalized）的时候（比如，影评者对影片的评价总是相对于平均水平偏离很大时），会倾向于给出更好的结果。

为了形象地展现这一方法，我们可以在图上标示出两位评论者的评分情况，如下页图 2-2 所示。Mick LaSalle 为《Superman》评了 3 分，而 Gene Seymour 则评了 5 分，所以该影片被定位在图中的(3,5)处。

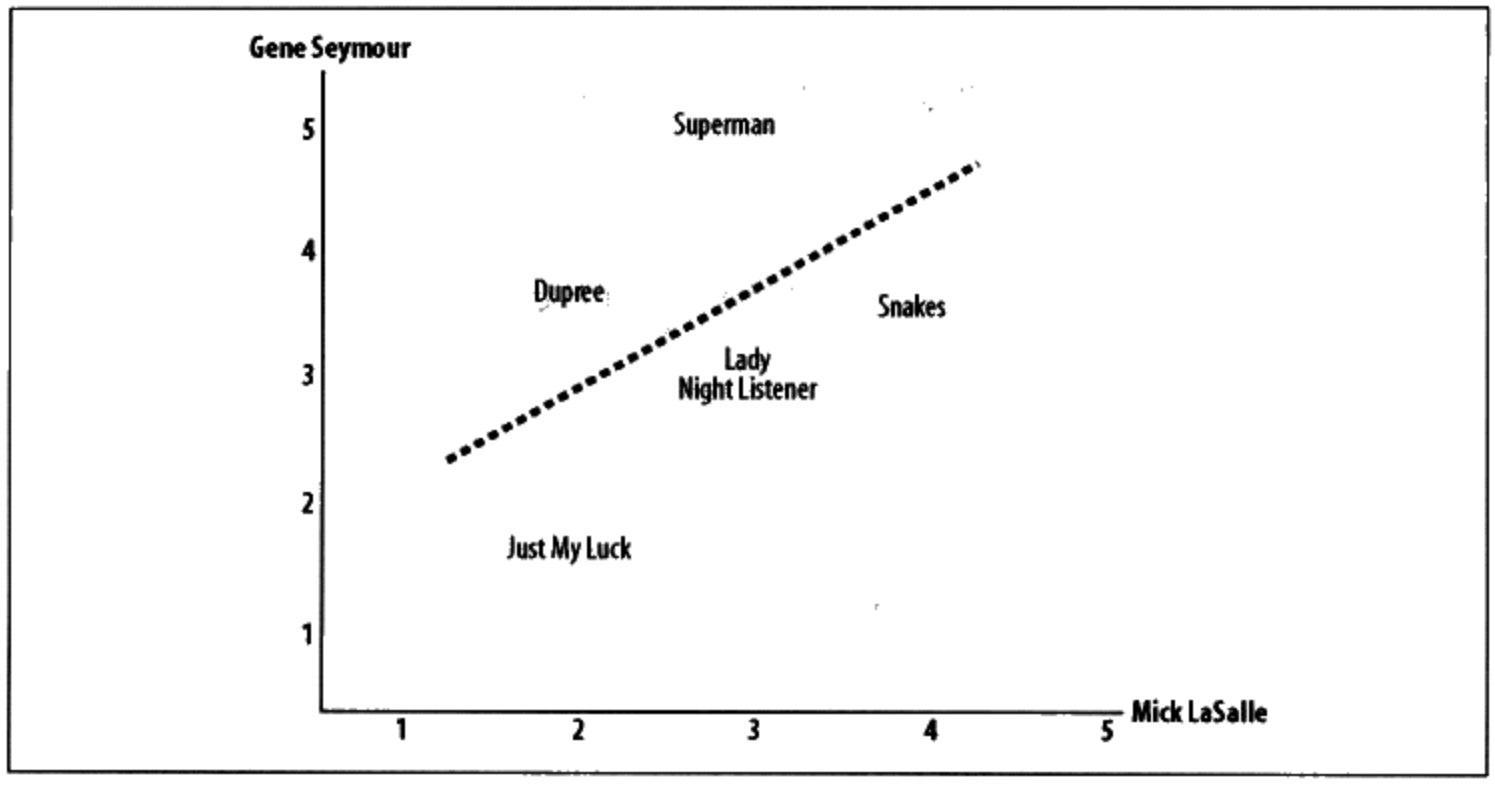


图 2-2：在散点图上比较两位影评者的评分结果

在图上，我们还可以看到一条直线。因其绘制原则是尽可能地靠近图上的所有坐标点，故而被称作**最佳拟合线**（best-fit line）。如果两位评论者对所有影片的评分情况都相同，那么这条直线将成为对角线，并且会与图上所有的坐标点都相交，从而得到一个结果为 1 的理想相关度评价。对于如上图所示的情况，由于评论者对部分影片的评分不尽相同，因而相关系数大约为 0.4 左右。图 2-3 展现了一个有着更高相关系数的例子，约为 0.75。

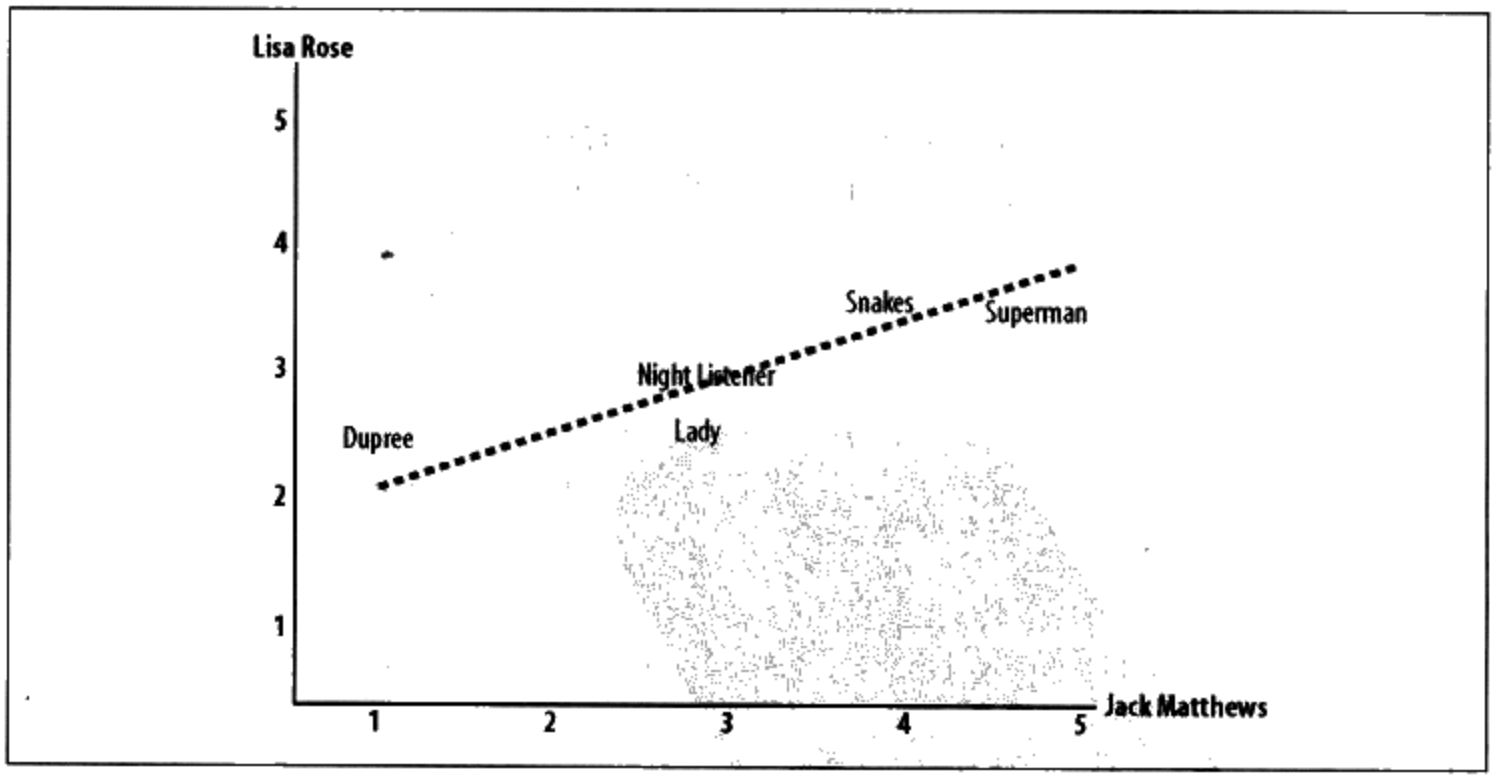


图 2-3：具有较高相关度评价值的两位评论者

在采用皮尔逊方法进行评价时，我们可以从图上发现一个值得注意的地方，那就是它修正了“**夸大分值**（grade inflation）”的情况。在这张图中，虽然 Jack Matthews 总是倾向于给出比 Lisa Rose 更高的分值，但最终的直线仍然是拟合的，这是因为他们两者有着相对近似的偏好。如果某人总是倾向于给出比另一个人更高的分值，而二者的分值之差又始终保持一致，则他们依然可能会存在很好的相关性。此前提到过的欧几里德距离评价方法，会因为一个人的评价始终比另一个人的更为“严格”（从而导致评价始终相对偏低），而得出两者不相近的结论，即使他们的品味很相似也是如此。而这一行为是否就是我们想要的结果，则取决于具体的应用场景。

皮尔逊相关度评价算法首先会找出两位评论者都曾评价过的物品，然后计算两者的评分总和与平方和，并求得评分的乘积之和。最后，算法利用这些计算结果计算出皮尔逊相关系数，如下列代码中粗体部分所示。不同于距离度量法，这一公式不是非常的直观，但是通过除以将所有变量的变化值相乘后得到的结果，它的确能够告诉我们变量的总体变化情况。

为了使用这一公式，请新建一个与 *recommendations.py* 中的 `sim_distance` 函数有同样签名的函数：

```
# 返回 p1 和 p2 的皮尔逊相关系数
def sim_pearson(prefs,p1,p2):
  # 得到双方都曾评价过的物品列表
  si={}
  for item in prefs[p1]:
    if item in prefs[p2]: si[item]=1

  # 得到列表元素的个数
  n=len(si)

  # 如果两者没有共同之处，则返回 1
  if n==0: return 1

  # 对所有偏好求和
  sum1=sum([prefs[p1][it] for it in si])
  sum2=sum([prefs[p2][it] for it in si])

  # 求平方和
  sum1Sq=sum([pow(prefs[p1][it],2) for it in si])
  sum2Sq=sum([pow(prefs[p2][it],2) for it in si])

  # 求乘积之和
  pSum=sum([prefs[p1][it]*prefs[p2][it] for it in si])

  # 计算皮尔逊评价值
  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
```

该函数将返回一个介于-1 与 1 之间的数值。值为 1 则表明两个人对每一样物品均有着完全一致的评价。与距离度量法不同，此处我们无须为达到正确的比率而对这一数值进行变换。现在，我们可以试着求一下图 2-3 中的相关评价值了：

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

应该选用哪一种相似性度量方法

Which Similarity Metric Should You Use?

我们在此处已经介绍了两种不同的度量方法，但实际上，还有许多方法可以衡量两组数据间的相似程度。使用哪一种方法最优，完全取决于具体的应用。如果你想看看哪种方法能够获得更好的实际效果，皮尔逊、欧几里德距离，或者任何其他方法，都是值得一试的。

本章剩余部分出现的函数均有一个可选的**相似性参数**，该参数指向一个实际的算法函数，这可以使针对算法的实验变得更为容易：我们可以指定 `sim_pearson` 或 `sim_distance` 作为相似性参数的取值。我们还可以使用许多其他的函数，如 **Jaccard 系数**或**曼哈顿距离算法**，作为相似度计算函数，只要它们满足如下条件：拥有同样的函数签名，以一个浮点数作为返回值，其数值越大代表相似度越大。

在 *http://en.wikipedia.org/wiki/Metric_%28mathematics%29#Examples* 上，我们还可以了解到其他用于比较的度量算法。

为评论者打分

Ranking the Critics

既然我们已经有了对两个人进行比较的函数，下面我们就可以编写函数，根据指定人员对每个人进行打分，并找出最接近的匹配结果了。在本例中，我们对找寻与自己有相似品味的影评者很感兴趣，因为这样我们就知道在选择影片时应该采纳谁的建议了。请将该函数加入 *recommendations.py* 中，以得到一个人员的有序列表，这些人与某个指定人员具有相近的品味：

```
# 从反映偏好的字典中返回最为匹配者
# 返回结果的个数和相似度函数均为可选参数
def topMatches(prefs,person,n=5,similarity=sim_pearson):
  scores=[(similarity(prefs,person,other),other)
                for other in prefs if other!=person]

# 对列表进行排序，评价值最高者排在最前面
  scores.sort()
  scores.reverse()
  return scores[0:n]
```

该函数利用了 Python 的**列表推导式**，采用先前定义过的某种距离度量算法，将自身和字典中的其他每一位用户都进行了比较。然后，函数返回排序结果中的前 *n* 项。

调用该方法并传入自己的姓名，将得到一个有关影评者及其相似度评价值的列表：

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

根据返回的结果我们了解到，应当阅读 Lisa Rose 所撰写的评论，因为她的品味与我们的很相近。如果你看过这些电影，也不妨将自己的偏好信息加入字典中，然后看看谁是你最喜欢的评论者。

推荐物品
Recommending Items

找到一位趣味相投的影评者并阅读他所撰写的评论固然不错，但现在我们真正想要的不是这些，而是一份影片的推荐。当然，我们也可以查找与自己品味最为相近的人，并从他所喜欢的影片中找出一部自己还未看过的影片，不过这样做太随意了（permissive）。有时，这种方法可能会有问题：评论者还未对某些影片做过评论，而这些影片也许就是我们所喜欢的。还有一种可能是，我们会找到一个热衷某部影片的古怪评论者，而根据 `topMatches` 所返回的结果，所有其他的评论者都不看好这部影片。

为了解决上述问题，我们须要通过一个经过加权的评价值来为影片打分，评论者的评分结果因此而形成了先后的排名。为此，我们须要取得所有其他评论者的评价结果，借此得到相似度后，再乘以他们为每部影片所给的评价值。表 2-2 给出了这一方法的执行过程。

表 2-2：为 Toby 提供推荐

评论者	相似度	Night	S.xNight	Lady	S.xLady	Luck	S.xLuck
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		
总计			12.89		8.38		8.07
Sim. Sum			3.84		2.95		3.18
总计/Sim. Sum			3.35		2.83		2.53

表中列出了每位评论者的相关度评价值，以及他们对三部影片（《The Night Listener》、《Lady in the Water》和《Just My Luck》）的评分情况（我们还不曾参与评分）。以 S.x 打头的列给出了乘以评价值之后的相似度。如此一来，相比于与我们不相近的人，那些与我们相近的人将会对整体评价值拥有更多的贡献。总计一行给出了所有加权评价值的总和。

我们也可以选择利用总计值来计算排名，但是我们还须要考虑到，一部受更多人评论的影片会对结果产生更大的影响。为了修正这一问题，我们须要除以表中名为 Sim.Sum 的那一行，它代表了所有对这部电影有过评论的评论者的相似度之和。由于每个人都对影片《The Night Listener》进行了评论，因此我们用总计值除以全部相似度之和。而对于影片《Lady in the Water》而言，Puig 并未做过评论，因此我们将这部影片的总计值除以所有其他人的相似度之和。表中最后一行给出了相除的结果。

下列代码反映了上述过程，非常的简单易懂，并且它对欧几里德距离评价或皮尔逊相关度评价都是适用的。请将其加入 *recommendations.py* 中：

```
# 利用所有他人评价值的加权平均，为某人提供建议
def getRecommendations(prefs,person,similarity=sim_pearson):
  totals={}
  simSums={}
  for other in prefs:
    # 不要和自己做比较
    if other==person: continue
    sim=similarity(prefs,person,other)

    # 忽略评价值为零或小于零的情况
    if sim<=0: continue
    for item in prefs[other]:

      # 只对自己还未曾看过的影片进行评价
      if item not in prefs[person] or prefs[person][item]==0:
        # 相似度*评价值
        totals.setdefault(item,0)
        totals[item]+=prefs[other][item]*sim
        # 相似度之和
        simSums.setdefault(item,0)
        simSums[item]+=sim

  # 建立一个归一化的列表
  rankings=[(total/simSums[item],item) for item,total in totals.items()]

  # 返回经过排序的列表
  rankings.sort()
  rankings.reverse()
  return rankings
```

上述代码循环遍历所有位于字典 prefs 中的其他人。针对每一次循环，它会计算由 person 参数所指定的人员与这些人的相似度。然后它会循环遍历所有打过分的项。以黑体显示的代码行说明了每一项的最终评价值的计算方法——用每一项的评价值乘以相似度，并将所

得乘积累加起来。最后，我们将每个总计值除以相似度之和，借此对评价值进行归一化处理，然后返回一个经过排序的结果。

这样，我们就可以找到自己接下来应该要看的电影了：

```
>>> 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')]
```

此处，我们不仅得到了一个经过排名的影片列表，而且还推测出了自己对每部影片的评价情况。根据这份结果，我们可以决定自己究竟要不要观看其中的某部影片，还是最好干脆什么也别看。有赖于具体的应用，假如无法满足某一用户给出的标准，我们也可以决定不给予建议。你会发现，选择不同的相似性度量方法，对结果的影响是微乎其微的。

现在，我们已经建立起了一个完整的推荐系统，它适用于任何类型的商品或网络链接。我们所要做的全部事情就是：建立一个涉及人员、物品和评价值的字典，然后就可以借此来为任何人提供建议了。在本章的后续章节中，你将会看到如何利用 del.icio.us API 来获取真实数据，进而向人们推荐 Web 站点。

匹配商品

Matching Products

现在，我们已经知道了如何为指定人员寻找品味相近者，以及如何向其推荐商品的方法，但是假如我们想了解哪些商品是彼此相近的，那又该如何做呢？也许我们曾经在购物网站上遇到过这种情形，尤其是当网站还没有收集到关于用户的足够信息时。图 2-4 显示了 Amazon 网站上有关《Programming Python》一书的局部网页。

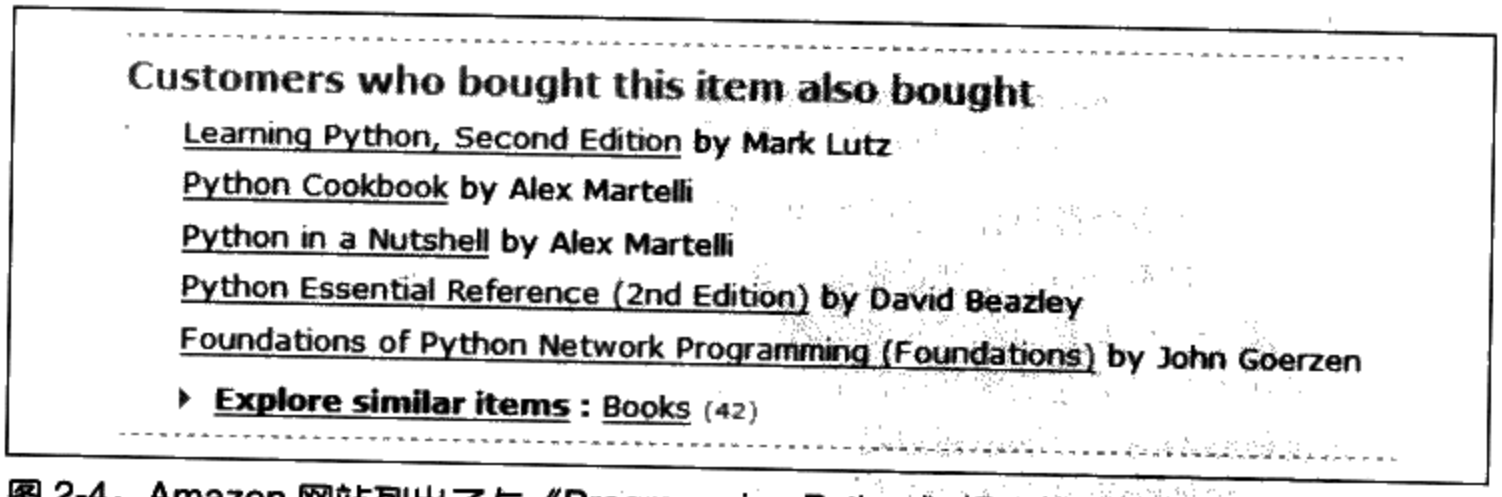


图 2-4：Amazon 网站列出了与《Programming Python》相近的同类商品

在这种情况下，我们可以通过查看哪些人喜欢某一特定物品，以及这些人喜欢哪些其他物品来决定相似度。事实上，这和我们此前用来决定人与人之间相似度的方法是一样的——只须要将人员与物品对换即可。因此，假如我们将

```
{'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}}
```

换成

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

就可以复用以前所写的方法了。将执行这一转换过程的函数加入 *recommendations.py* 中：

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

      # 将物品和人员对调
      result[item][person]=prefs[person][item]
  return result
```

现在，调用以前曾经用过的 `topMatches` 函数，得到一组与《Superman Returns》最为相近的影片：

```
>> 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')]
```

请注意：在本例中，实际存在着一些相关评价值为负的情况，这表明那些喜欢影片《Superman Returns》的人，存在不喜欢《Just My Luck》的倾向，如图 2-5 所示。

上面我们示范了为某部影片提供相关影片的推荐，不仅如此，我们甚至还可以为影片推荐评论者。例如，也许我们正在考虑邀请谁和自己一起参加某部影片的首映式。

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

将人和物对调并不总是会得到有价值的结果，但是大多数情况下，这将有助于我们作出有意义的对比。为了向不同的个体推荐商品，在线零售商可能会收集人们的购买历史。将商品与人进行对调——正如我们此前所做的那样——可以令零售商找到购买某些商品的潜在客户。这对于他们为了清仓处理某些商品而在市场营销投入方面制定的规划，也许是很有助益的。这种做法的另一个潜在用途是，在专门推荐链接的网站上，这样做可以确保新出现的链接，能够被那些最有可能对它产生兴趣的网站用户找到。

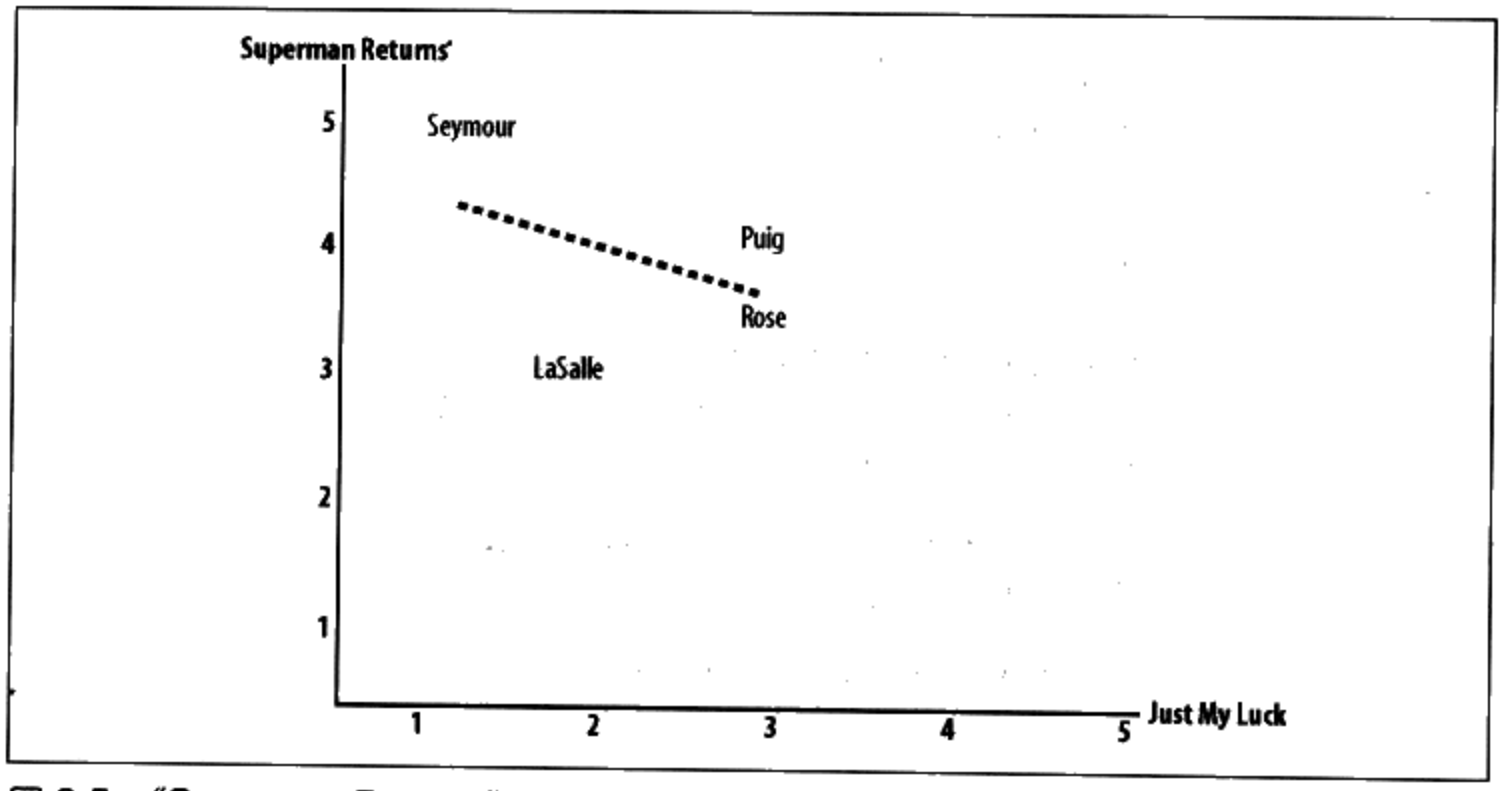


图 2-5：《Superman Returns》和《Just My Luck》存在负值相关性

构建一个基于 del.icio.us 的链接推荐系统

Building a del.icio.us Link Recommender

本节将向大家介绍，如何从最受欢迎的在线书签网站上获取数据，如何利用这些数据查找相近用户，并向他们推荐以前未曾看过的链接。该网站——可以通过 *http://del.icio.us* 访问到——允许人们建立自己的账号，并允许张贴自己感兴趣的链接，以备日后参考之用。我们可以访问网站并查看其他人张贴的链接，也可以浏览许多不同用户所张贴的热门链接。图 2-6 展示了一个取自 del.icio.us 网站的网页样例。

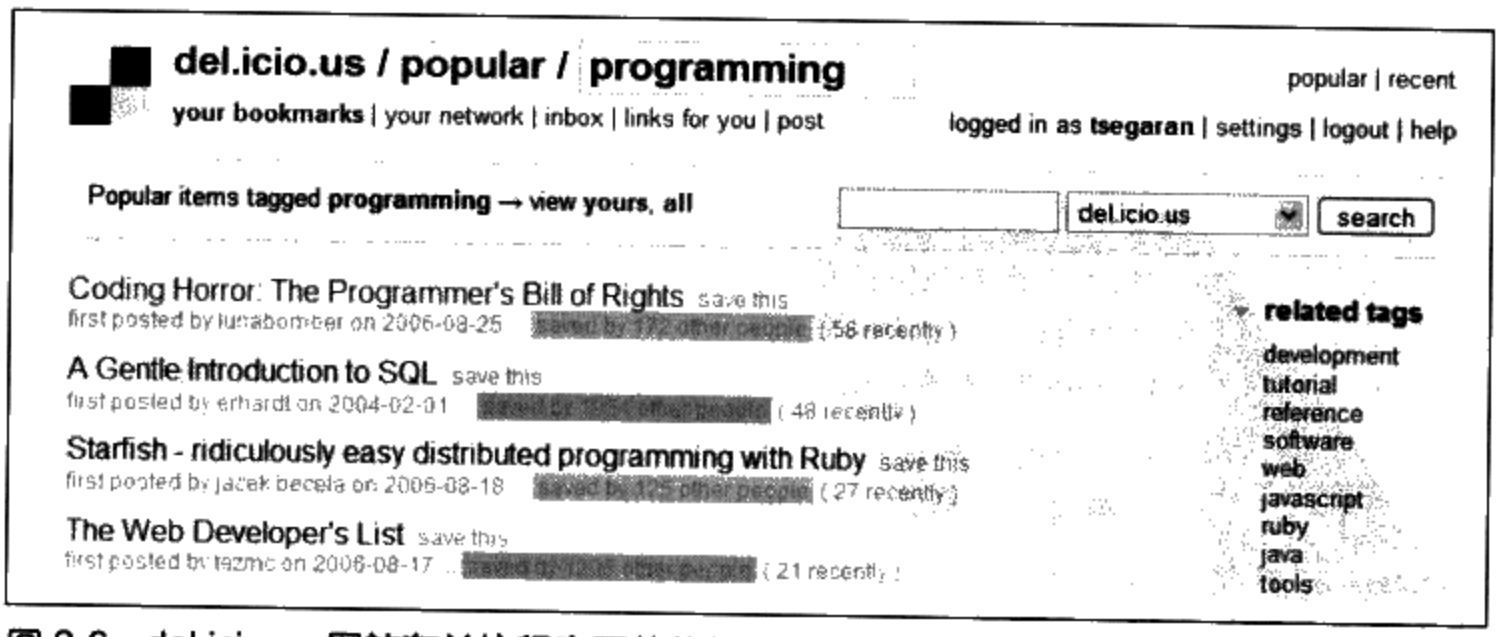


图 2-6：del.icio.us 网站有关编程方面的热门网页

和某些链接共享类的网站（link-sharing sites）不同，在本书撰写期间，del.icio.us 还没有提供寻找相近用户的方法，也不向用户提供链接推荐。所幸的是，我们可以利用本章所讨论的技术来为自己添加这部分功能。

The del.icio.us API

通过 API 访问 del.icio.us 网站所获得的数据是以 XML 格式返回的。为了方便起见，有一套事先编写好的 Python API，我们可以从 *http://code.google.com/p/pydelicious/source* 或 *http://oreilly.com/catalog/9780596529321* 处下载到。

为了使本节中的示例能够正常运行，我们须要下载函数库的最新版本，并将其置于 Python 函数库所在路径下（有关安装该库的更多信息，请见附录 A）。

该函数库提供了一系列简单的函数调用，可以用来获得用户提交的链接信息。例如，为了得到一组近期张贴的有关编程方面的热门链接，我们可以使用 `get_popular` 函数：

```
>> 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'},
...etc...
```

我们可以看到，上述调用返回了一个包含字典的列表，其中的每一项都包含了：URL、描述，以及提交者。因为我们使用的是真实数据，所以实际结果可能会与上例有所不同。还有另外两个函数调用我们也可能会用到：`get_urlposts`，返回给定 URL 的所有张贴记录；`get_userposts`，返回给定用户的所有张贴记录。这些函数调用所返回的数据同样都是 XML 格式的。

构造数据集
Building the Dataset

要想从 del.icio.us 网站将所有用户张贴的信息都下载下来是不可能的，因此我们须要从中选择一个子集。根据自己的喜好，我们可以选择任何形式的子集，不过为了让例子给出的结果显得更有意义，我们不妨找一找那些经常张贴链接的用户和张贴内容类似的用户。

为了达到这一目的，有一种做法是找到一组近期提交过的某一热门链接，且链接附带指定标签（tag）的用户。请新建一个名为 *deliciousrec.py* 的文件，输入如下代码：

```
from pydelicious import get_popular,get_userposts,get_urlposts

def initializeUserDict(tag,count=5):
  user_dict={}
```

```
    # 获取前 count 个最受欢迎的链接张贴记录
    for p1 in get_popular(tag=tag)[0:count]:
      # 查找所有张贴该链接的用户
      for p2 in get_urlposts(p1['href']):
        user=p2['user']
        user_dict[user]={}
    return user_dict
```

执行上述代码将得到一个包含若干用户数据的字典，其中每一项都各自指向一个等待填入具体链接的空字典。由于 API 只会返回最后 30 个张贴链接的用户，因此上述函数只搜集了与前 5 个链接相对应的用户数据，并进而构造出一个数据集。

不同于保存影评者的数据集，在本例中只有两种可能的评价值：0（如果用户没有张贴这一链接）和 1（如果用户张贴了这一链接）。现在，我们可以利用 API 来建立一个填充所有用户评价值的函数了。请将如下代码加入到 *deliciousrec.py* 中：

```
def fillItems(user_dict):
  all_items={}
  # 查找所有用户都提交过的链接
  for user in user_dict:
    for i in range(3):
      try:
        posts=get_userposts(user)
        break
      except:
        print "Failed user "+user+", retrying"
        time.sleep(4)
    for post in posts:
      url=post['href']
      user_dict[user][url]=1.0
      all_items[url]=1

  # 用 0 填充缺失的项
  for ratings in user_dict.values():
    for item in all_items:
      if item not in ratings:
        ratings[item]=0.0
```

我们可以利用上述函数来构造一个数据集，类似于本章开始处手工构造的反映影评者信息的字典：

```
>> from deliciousrec import *
>> delusers=initializeUserDict('programming')
>> delusers ['tsegaran']={} # 如果你也使用 delicious，则将自己也加入字典中
>> fillItems(delusers)
```

上面第三行代码将用户 tsegaran 添加到了列表中。假如你也使用 del.icio.us，则不妨以自己的名字来替换 tsegaran。

对 fillItems 的调用可能会花费几分钟的时间来执行，因为它会向网站发起数百个请求。有时会因为请求反复发起太快而致使 API 阻塞请求响应。在这种情况下，代码会暂停执行并至多重试三次。

推荐近邻与链接

Recommending Neighbors and Links

既然我们已经构造好了数据集，那就可以利用此前在影评数据集中曾经用过的同一组函数进行推荐了。为了随机选择一位用户，并找出与其品味相近的其他用户，请将如下代码输入你的 Python 会话中：

```
>> 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')]
```

我们也可以通过调用 getRecommendations 函数为该用户获取推荐链接。因为方法调用将会依序返回全部物品，所以最好将其限制在前 10 条：

```
>> 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')]
```

当然，与之前所演示的一样，偏好列表中的各项是可以被调换的，这样我们就可以依据链接而非人员来进行搜索了。为了寻找与我们所关注的某一链接相类似的一组链接，我们可以尝试输入下列命令：

```
>> 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')]
```

就这样！我们成功地为 del.icio.us 网站增加了一个推荐引擎。我们还可以做更多的事情。因为 del.icio.us 网站支持根据标签进行搜索，所以我们还可以找出那些彼此相似的标签。甚至，我们还可以搜索出那些使用不同账号张贴同一链接的、企图以此来炒作网页的人。

基于物品的过滤

Item-Based Filtering

迄今为止已经完成的推荐引擎，要求我们使用来自每一位用户的全部评分来构造数据集。这种方法对于数量以千计的用户或物品规模而言或许是没问题的，但对于像 Amazon 这样有着上百万客户和商品的大型网站而言，将一个用户和所有其他用户进行比较，然后再对每位用户评过分的商品进行比较，其速度可能是无法忍受的。同样，一个商品销售量为数百万的网站，也许用户在偏好方面彼此间很少会有重叠，这可能会令用户的相似性判断变得十分困难。

目前为止我们所采用的技术被称为**基于用户的协作型过滤**（user-based collaborative filtering）。除此以外，还有另一种可供选择的方法被称为**基于物品的协作型过滤**（item-based collaborative filtering ）。在拥有大量数据集的情况下，基于物品的协作型过滤能够得出更好的结论，而且它允许我们将大量计算任务预先执行，从而使须要给予推荐的用户能够更快地得到他们所要的结果。

基于物品的过滤过程沿用了我们之前已经讨论过的许多内容。其总体思路就是为每件物品预先计算好最为相近的其他物品。然后，当我们想为某位用户提供推荐时，就可以查看他曾经评过分的物品，并从中选出排位靠前者，再构造出一个加权列表，其中包含了与这些选中物品最为相近的其他物品。此处最为显著的区别在于，尽管第一步要求我们检查所有的数据，但是**物品间的比较不会像用户间的比较那么频繁变化**。这意味着，无须不停地计算与每样物品最为相近的其他物品，我们可以将这样的运算任务安排在网络流量不是很大的时候进行，或者在独立于主应用之外的另一台计算机上单独进行。

构造物品比较数据集

Building the Item Comparison Dataset

为了对物品进行比较，我们要做的第一件事情就是编写一个函数，构造一个包含相近物品的完整数据集。再重申一次，这项工作无须在每次提供推荐时都做一遍——相反，构建完一次数据集之后，我们就可以在需要的时候重复使用它。

为了生成数据集，请将下列函数加入 *recommendations.py* 中：

```
def calculateSimilarItems(prefs,n=10):
  # 建立字典，以给出与这些物品最为相近的所有其他物品
  result={}

  # 以物品为中心对偏好矩阵实施倒置处理
  itemPrefs=transformPrefs(prefs)
  c=0
  for item in itemPrefs:
    # 针对大数据集更新状态变量
    c+=1
    if c%100==0: print "%d / %d" % (c,len(itemPrefs))
    # 寻找最为相近的物品
    scores=topMatches(itemPrefs,item,n=n,similarity=sim_distance)
    result[item]=scores
  return result
```

该函数首先利用了此前定义过的 transformPrefs 函数，对反映评价值的字典进行倒置处理，从而得到一个有关物品及其用户评价情况的列表。然后，程序又循环遍历每项物品，并将转换了的字典传入 topMatches 函数中，求得最为相近的物品及其相似度评价值。最后，它建立并返回了一个包含物品及其最相近物品列表的字典。

请在你的 Python 会话中构造一个物品相似度的数据集，然后看一看运行的结果：

```
>>> reload(recommendations)
>>> itemsim=recommendations.calculateSimilarItems(recommendations.critics)
>>> itemsim
{'Lady in the Water': [(0.40000000000000002, 'You, Me and Dupree'),
                       (0.28571428571428571, 'The Night Listener'),...
 'Snakes on a Plane': [(0.22222222222222221, 'Lady in the Water'),
                       (0.18181818181818182, 'The Night Listener'),...
etc.
```

请记住，只有频繁执行该函数，才能令物品的相似度不至过期。通常我们须要在用户基数和评分数量还不是很大的时候执行这一函数，但是随着用户数量的不断增长，物品间的相似度评价值通常会变得越来越稳定。

获得推荐

Getting Recommendations

现在，我们已经可以在不遍历整个数据集的情况下，利用反映物品相似度的字典来给出推荐了。我们可以取到用户评价过的所有物品，找出其相近物品，并根据相似度对其进行加权。我们可以很容易地根据物品字典来得到相似度。

表 2-3 给出了利用基于物品的方法寻找推荐的过程。不同于表 2-2，此处并没有涉及所有评论者，而是给出了一个表格，对我们打过分和未打过分的影片进行了对照。

表 2-3：为 Toby 提供基于物品的推荐

影片	评分	Night	R.xNight	Lady	R.xLady	Luck	R.xLuck
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
总计		0.433	1.378	0.713	1.762	0.352	0.914
归一化结果			3.183		2.473		2.598

此处的每一行都列出了一部我们曾经观看过的影片，以及对该片的个人评价。对于每一部我们还未曾看过的影片，相应有一列会指出它与已观看影片的相近程度——例如：影片《Superman》和《The Night Listener》之间的相似度评价值为 0.103。以 R.x 打头的列给出了我们对影片的评价值乘以相似度之后的结果——由于我们对《Superman》的评分是 4.0，所以“Night in the Superman”的后一列对应取值为：4.0 * 0.103 = 0.412.

总计一行给出了每部影片相似度评价值的总计值及其 R.x 列的总计值。为了预测我们对每部影片的评分情况，只要将 R.x 列的总计值除以相似度一列的总计值即可。我们对影片《The Night Listener》的评分情况为：1.378/0.433 = 3.183。

请将下列函数加入 *recommendations.py* 中，我们就可以利用上述功能了：

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

  # 循环遍历由当前用户评分的物品
  for (item,rating) in userRatings.items():

    # 循环遍历与当前物品相近的物品
    for (similarity,item2) in itemMatch[item]:

      # 如果该用户已经对当前物品做过评价，则将其忽略
      if item2 in userRatings: continue

      # 评价值与相似度的加权之和
      scores.setdefault(item2,0)
      scores[item2]+=similarity*rating

      # 全部相似度之和
      totalSim.setdefault(item2,0)
      totalSim[item2]+=similarity

  # 将每个合计值除以加权和，求出平均值
  rankings=[(score/totalSim[item],item) for item,score in scores.items()]

  # 按最高值到最低值的顺序，返回评分结果
  rankings.sort()
  rankings.reverse()
  return rankings
```

我们可以试着运行一下该函数，并传入以前构造好的相似度数据集，为 Toby 提供一个新的推荐结果：

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

《The Night Listener》依然以较大的评价值排在最前面，而《Just My Luck》和《Lady in the Water》尽管依然靠得很近，但它们的排列位置有了变化。更为重要的是，在调用 `getRecommendedItems` 时，我们不必再为所有其他评论者计算相似度评价值，因为物品相似度数据集已经事先构造好了。

使用 MovieLens 数据集

Using the MovieLens Dataset

作为本章的最后一个示例，让我们来看一个涉及电影评价的真实数据集，叫做 *MovieLens*。MovieLens 是由明尼苏达州立大学的 GroupLens 项目组开发的。我们可以从 *http://www.grouplens.org/node/12* 处下载到数据集（译注 1）。有两个数据集可供下载。请选择下载十万数据集。根据所使用的平台，我们可以选择以 *tar.gz* 格式或 *zip* 格式进行下载。

译注 1：此处提供的下载链接有误，实际为 *http://www.grouplens.org/node/73*。

归档文件中包含了不少文件，不过我们关心的是 *u.item* 和 *u.data*，前者包含了一组有关影片 ID 和片名的列表，后者则包含了以如下形式给出的实际评价情况：

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

此处的每一行数据都包含了一个用户 ID、影片 ID、用户对该片所给的评分，以及评价的时间。我们可以通过影片的 ID 获取到片名，但对于用户数据而言，由于是匿名的，因此在本节中我们只能对用户 ID 进行处理。该数据集中包含了 943 名用户对 1 682 部影片所作的评价，每位用户至少曾为 20 部影片做过评价。

请在 *recommendations.py* 文件中新建一个方法，取名 `loadMovieLens`，用以加载上述数据：

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

  # 获取影片标题
  movies={}
  for line in open(path+'/u.item'):
    (id,title)=line.split('|')[0:2]
    movies[id]=title

  # 加载数据
  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
```

请在你的 Python 会话中将数据加载进来，并随机查看任意一位用户的评分情况：

```
>>> 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, etc...
```

我们可以获取基于用户的推荐：

```
>>> 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)'),
 etc...]
```

当利用上述方式获取推荐时，我们可能会注意到有停顿的现象，停顿时间的长短取决于我们所用机器的速度。那是因为此刻我们正在处理一个较之先前更大规模的数据集。拥有的用户越多，获取基于用户的推荐所花费的时间就越长。现在，请改换为基于物品的推荐试一下：

```
>>> itemsim=recommendations.calculateSimilarItems(prefs,n=50)
100 / 1664
200 / 1664
等等……
>>> 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)'),etc...]
```

尽管构造物品的相似度字典花费了较长的时间，但是推荐过程几乎是在数据构造完毕后瞬间完成的。而且，获取推荐所花费的时间不会随着用户数量的增加而增加。

这是一个很好的测试用数据集，利用它我们可以了解不同评价方法对结果的影响程度，以及基于物品和基于用户的过滤方法是如何以不同方式执行的。GroupLens 网站还有另外一些数据集可供使用，内容涉及图书、笑话，以及更多的影片。

基于用户进行过滤还是基于物品进行过滤

User-Based or Item-Based Filtering?

在针对大数据集生成推荐列表时，基于物品进行过滤的方式明显要比基于用户的过滤更快，不过它的确有维护物品相似度表的额外开销。同时，这种方法根据数据集“稀疏”程度上的不同也存在精准度上的差异。在涉及电影的例子中，由于每个评论者几乎对每部影片都做过评价，所以数据集是密集的（而非稀疏的）。另一方面，它又不同于查找两位有相近 del.icio.us 书签的用户——大多数书签都是为小众群体所收藏的，这就形成了一个稀疏数据集。对于稀疏数据集，基于物品的过滤方法通常要优于基于用户的过滤方法，而对于密集数据集而言，两者的效果则几乎是一样的。

提示：要了解更多有关这些算法在执行效率上的差异情况，请从 *http://citeseer.ist.psu.edu/sarwar01itembased.html* 处下载由 Sarwar 等人所撰写的一篇论文，名为《基于物品的协作型过滤推荐算法（Item-based Collaborative Filtering Recommendation Algorithms）》。

尽管如此，基于用户的过滤方法更加易于实现，而且无需额外步骤，因此它通常更适用于规模较小的变化非常频繁的内存数据集。最后，在一些应用中，告诉用户还有哪些人与自己有着相近偏好是有一定价值的——也许对于一个购物网站而言，我们并不想这么做，但是对于一个链接共享类或音乐推荐类的网站，这种潜在需求却是存在的。

现在，我们已经学会了怎样计算相似度评价值，以及怎样利用它们对用户和物品进行比较。本章介绍了两种不同的推荐算法，基于用户的推荐算法和基于物品的推荐算法，还介绍了记录用户偏好信息的方法，以及使用 del.icio.us API 构建链接推荐系统的方法。在第 3 章

中，我们将会看到如何以本章中的某些观点为基础，运用无监督聚类算法（unsupervised clustering algorithms）来查找相近的用户组。第 9 章中，我们将会考查其他可行的方法，以实现在得知人们偏好类型的前提下对用户进行匹配。

练习

Exercises

1. **Tanimoto 分值**　求出 Tanimoto 相似度评价值。在何种情况下，我们可以将该方法作为相似性的度量方法，用以替代欧几里德距离法或皮尔逊系数法？请利用 Tanimoto 分值建立一个新的相似度函数。

2. **标签相似度**　请使用 del.icio.us API 构造一个涉及标签和链接的数据集。利用它来计算不同标签间的相似度，看一看是否能找到相似度几乎一样的情况。请找出某些本该被标记为“programming”，但却没被标记的链接。

3. **基于用户算法的执行效率**　由于基于用户的过滤算法，在每次须要推荐时，都会将某位用户与其他所有用户进行比较，故而效率低下。请编写一个预先计算用户相似度的函数，并修改涉及推荐的相关代码，只取出当前用户外的其他前 5 名用户来给出推荐。

4. **基于物品的书签过滤**　请从 del.icio.us 网站下载一组数据，并将其加入数据集中。建立一张“物－物”表，并利用它为不同用户提供基于物品的推荐。将之与基于用户的推荐做一个对比。

5. **Audioscrobbler**　请访问 *http://www.audioscrobbler.net*，该网站拥有一个由大群用户的音乐偏好所构成的数据集。利用网站提供的 Web Services API 获取一组数据，并以此来构造一个音乐推荐系统。

第 3 章

发现群组

Discovering Groups

在第 2 章中我们讨论了如何寻找紧密相关的事物，借此我们可以找到在电影欣赏方面与自己有着相同品味的人。本章对上一章中的思想加以拓展，并引入“**数据聚类**”(data clustering)的概念，这是一种用以寻找紧密相关的事、人或观点，并将其可视化的方法。本章中，我们将学习到如下内容：从各种不同的来源中构造算法所需的数据；两种不同的聚类算法；更多有关距离度量（distance metrics）的知识；简单的图形可视化代码，用以观察所生成的群组；最后，我们还会学习如何将异常复杂的数据集投影到二维空间中。

聚类时常被用于数据量很大（data-intensive）的应用中。跟踪消费者购买行为的零售商们，除了利用常规的消费者统计信息外，还可以利用这些信息自动检测出具有相似购买模式的消费者群体。年龄和收入都相仿的人也许会有迥然不同的着装风格，但是通过使用聚类算法，我们就可以找到“时装岛屿（fashion islands)”，并据此开发出相应的零售或市场策略。聚类在计量生物学领域里也有大量的运用，我们用它来寻找具有相似行为的基因组，相应的研究结果可以表明，这些基因组中的基因会以同样的方式响应外界的活动，或者表明它们是相同生化通路（biological pathway）中的一部分。

因为本书讨论的是集体智慧，所以本章中的例子所涉及的，都是由多个人贡献不同信息的问题。第一个例子将对博客用户所讨论的话题，以及他们所使用的特殊词汇进行考查，并以此来说明，我们可以根据用户撰写的文字对博客进行分组，还可以根据他们对词汇的用法将词汇分组。第二个例子将对社区网站进行考查，人们可以在这些网站上列举他们已经拥有的和希望拥有的物品，我们将利用这些信息来示范，如何对人们的意愿进行分组。

监督学习和无监督学习

Supervised Versus Unsupervised Learning

利用样本输入和期望输出来学习如何预测的技术被称为监督学习法（supervised learning methods)。在本书中，我们将学习到许多监督学习法，其中包括：神经网络、决策树、向量支持机，以及贝叶斯过滤。采用这些方法的应用程序，会通过检查一组输入和期望的输出

来进行“学习”。当我们想要利用这些方法中的任何一种来提取信息时，我们可以传入一组输入，然后期望应用程序能够根据其此前学到的知识来产生一个输出。

聚类是**无监督学习**（unsupervised learning）的一个例子。与神经网络或决策树不同，无监督学习算法不是利用带有正确答案的样本数据进行“训练”。它们的目地是要在一组数据中找寻某种结构，而这些数据本身并不是我们要找的答案。在前面提到的时装的例子中，聚类的结果不会告诉零售商每一位顾客可能会买什么，也不会预测新来的顾客适合哪种时尚。聚类算法的目标是采集数据，然后从中找出不同的群组。其他无监督学习的例子还包括**非负矩阵因式分解**（non-negative matrix factorization，将在第 10 章讨论）和**自组织映射**（self-organizing maps）。

单词向量
Word Vectors

为聚类算法准备数据的常见做法是定义一组公共的数值型属性，我们可以利用这些属性对数据项进行比较。这很类似于我们在第 2 章中所采取的做法，在那里，我们比较了评论者对同样一组影片所给出的评分情况，我们还分别用 1 和 0 来代表 del.icio.us 网站用户所用书签的存在和缺失情况。

对博客用户进行分类
Pigeonholing the Bloggers

本章我们将对两个示例数据集进行处理。在第一个数据集中，被用来聚类的是排名在前 120 位的一系列博客，为了对这些博客进行聚类，我们需要的是一组指定的词汇在每个博客订阅源中出现的次数。有关这一数据的一个小范围的子集，如表 3-1 所示。

表 3-1：单词在博客中出现次数的一个子集

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

根据单词出现的频度对博客进行聚类，或许可以帮助我们分析出是否存在这样一类博客用户，这些人经常撰写相似的主题，或者在写作风格上十分类似。这样的分析结果对于搜索、分类和挖掘当前大量的在线博客而言，可能是非常有价值的。

为了构造这样一个数据集，可能须要下载一系列博客订阅源，从中提取出文本，并据此建立起一个单词频度表。如果你想跳过数据集的构造环节，那么也可以去 *http://kiwitobes.com/clusters/blogdata.txt* 下载现成的数据集。

对订阅源中的单词进行计数

Counting the Words in a Feed

几乎所有的博客都可以在线阅读，或者通过 RSS 订阅源进行阅读。RSS 订阅源是一个包含博客及其所有文章条目信息的简单的 XML 文档。为了给每个博客中的单词计数，首先第一步就是要解析这些订阅源。所幸的是，有一个非常不错的程序能够完成这项工作， 它就是 Universal Feed Parser，我们可以从 *http://www.feedparser.org* 上下载到它。

有了 Universal Feed Parser，我们就可以很轻松地从任何 RSS 或 Atom 订阅源中得到标题、链接和文章的条目了。下一步，我们来编写一个从订阅源中提取所有单词的函数。请新建一个名为 *generatefeedvector.py* 的文件，并加入下列代码：

```
import feedparser
import re

# 返回一个 RSS 订阅源的标题和包含单词计数情况的字典
def getwordcounts(url):
  # 解析订阅源
  d=feedparser.parse(url)
  wc={}

  # 循环遍历所有的文章条目
  for e in d.entries:
    if 'summary' in e: summary=e.summary
    else: summary=e.description

    # 提取一个单词列表
    words=getwords(e.title+' '+summary)
    for word in words:
      wc.setdefault(word,0)
      wc[word]+=1
  return d.feed.title,wc
```

每个 RSS 和 Atom 订阅源都会包含一个标题和一组文章条目。通常，每个文章条目都有一段摘要，或者是包含了条目中实际文本的描述性标签。函数 getwordcounts 将摘要传给函数 getwords，后者会将其中所有的 HTML 标记剥离掉，并以非字母字符作为分隔符拆分出单词，再将结果以列表的形式加以返回。请将 getwords 函数加入 *generatefeedvector.py* 中：

```
def getwords(html):
  # 去除所有 HTML 标记
  txt=re.compile(r'<[^>]+>').sub('',html)

  # 利用所有非字母字符拆分出单词
  words=re.compile(r'[^A-Z^a-z]+').split(txt)

  # 转化成小写形式
  return [word.lower() for word in words if word!='']
```

为了开始下一步工作，我们现在需要一个订阅源的列表。如果愿意的话，我们也可以自己构造包含博客订阅源的 URL 列表，否则我们可以使用一个预先构造好的列表，该列表包含有 100 个 RSS 的 URL。为了构造这一列表，我们选取了目前被引用最多的所有博客的订阅

源，并从中去除了某些未包含文章正文，或者正文内容几乎都是图片信息的订阅源。我们可以从 *http://kiwitobes.com/clusters/feedlist.txt* 下载到此列表。

这是一个普通的文本文件，每一行对应一个 URL。如果我们拥有自己的博客，或者有一些博客是我们特别喜欢的，同时很想看看它们和某些热门博客的对比情况如何，那么我们也可以将这些博客的 URL 加入到文件中。

generatefeedvector.py 文件中的主体代码（这些代码不单独构成一个函数）循环遍历订阅源并生成数据集。代码的第一部分遍历 *feedlist.txt* 文件中的每一行，然后生成针对每个博客的单词统计，以及出现这些单词的博客数目（apcount）。请将下列代码加入到 *generatefeedvector.py* 文件的末尾：

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

下一步，我们来建立一个单词列表，将其实际用于针对每个博客的单词计数。因为像“the”这样的单词几乎到处都是，而像“flim-flam”这样的单词则有可能只出现在个别博客中，所以通过只选择介于某个百分比范围内的单词，我们可以减少须要考查的单词总量。在本例中，我们可以将 10%定为下界，将 50%定为上界，不过假如你发现有过多常见或鲜见的单词出现，不妨尝试一下不同的边界值。

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

最后，我们利用上述单词列表和博客列表来建立一个文本文件，其中包含一个大的矩阵，记录着针对每个博客的所有单词的统计情况：

```
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')
```

为了生成单词计数文件，请在命令行运行 *generatefeedvector.py* 文件：

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

下载所有订阅源可能须要花几分钟的时间，这一过程最终将会生成一个名为 *blogdata.txt* 的输出文件。请打开文件验证一下，是否包含一个以制表符分隔的表格，其中的每一列对应一个单词，每一行对应一个博客。本章中出现的函数都将统一采用这一文件格式，日后我们还可以据此来构造新的数据集，我们甚至还可以将一个电子表格另存为如此格式的文本文件，并沿用本章中的算法对其实施聚类。

分级聚类

Hierarchical Clustering

分级聚类通过连续不断地将最为相似的群组两两合并，来构造出一个群组的层级结构。其中的每个群组都是从单一元素开始的，在本章的例子中，这个单一元素就是博客。在每次迭代的过程中，分级聚类算法会计算每两个群组间的距离，并将距离最近的两个群组合并成一个新的群组。这一过程会一直重复下去，直到只剩一个群组为止。如图 3-1 所示。

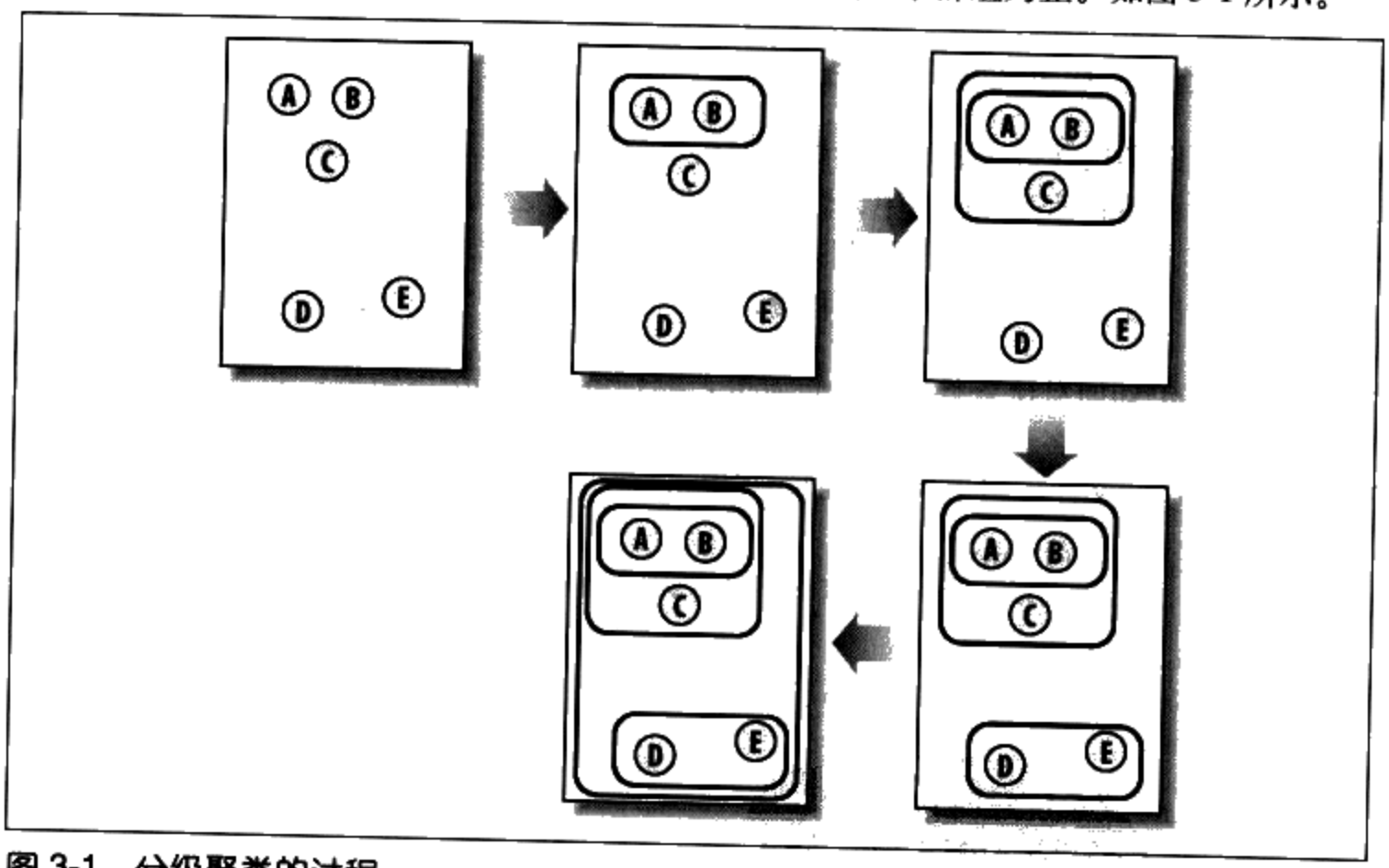


图 3-1：分级聚类的过程

在上图中，元素的相似程度是通过它们的相对位置来体现的——两个元素距离越近，它们就越相似。开始时，群组还只有一个元素。在第二步中，我们可以看到 A 和 B，这两个紧靠在一起的元素，已经合并成了一个新的群组，新群组所在的位置位于这两个元素的中间。在第三步中，新群组又与 C 进行了合并。因为 D 和 E 现在是距离最近的两个元素，所以它们共同构成了一个新的群组。最后一步将剩下的两个群组合并到了一起。

通常，待分级聚类完成之后，我们可以采用一种图形化的方式来展现所得的结果，这种图被称为**树状图**（dendrogram），图中显示了按层级排列的节点。上述例子中的树状图如图 3-2 所示。

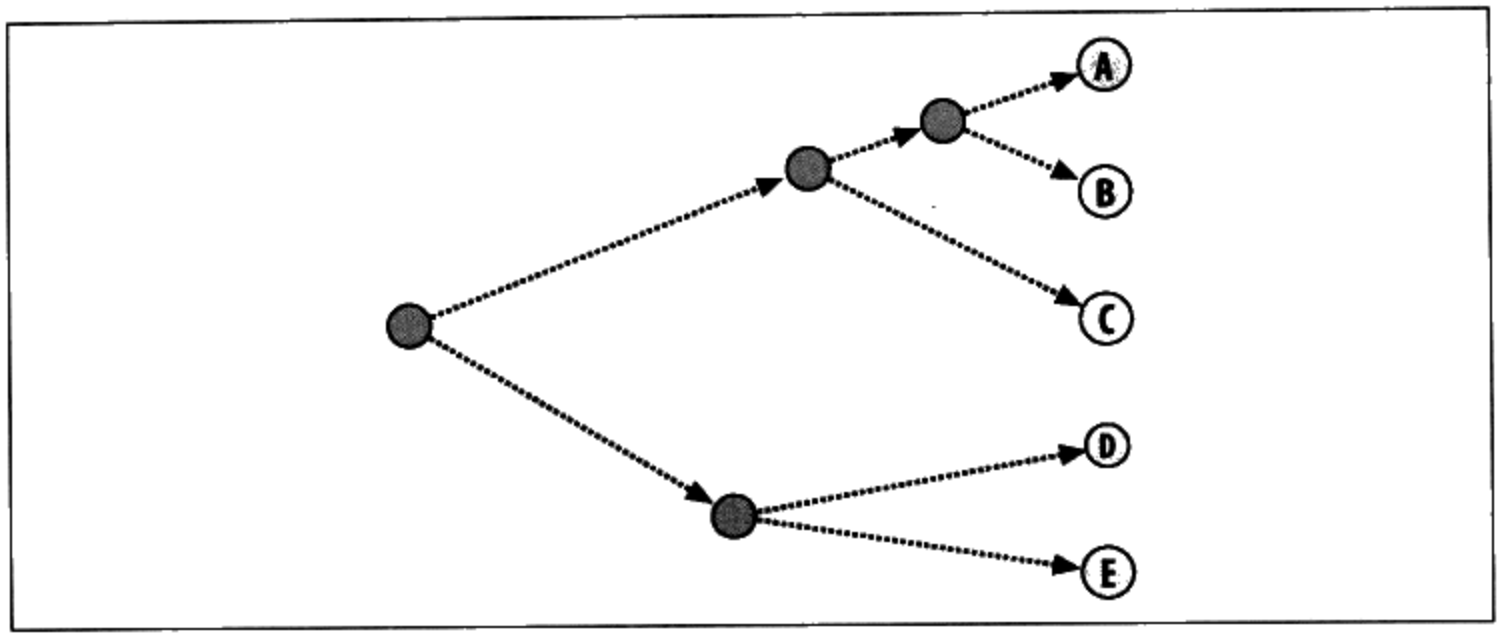


图 3-2：树状图是分级聚类的一种可视化形式

树状图不仅可以利用连线来表达每个聚类的构成情况，而且还可以利用距离来体现构成聚类的各元素间相隔的远近。在图中，聚类 AB 与 A 和 B 之间的距离要比聚类 DE 与 D 和 E 之间的距离更加接近。这种图形绘制方式能够帮助我们有效地确定一个聚类中各元素间的相似程度，并以此来指示聚类的**紧密程度**。

本节我们将示范如何对博客数据集进行聚类，以构造博客的层级结构；如果构造成功，我们将实现按主题对博客进行分组。首先，我们需要一个方法来加载数据文件。请新建一个名为 *clusters.py* 的文件，将下列函数加入其中：

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

  # 第一行是列标题
  colnames=lines[0].strip().split('\t')[1:]
  rownames=[]
  data=[]
  for line in lines[1:]:
    p=line.strip().split('\t')
    # 每行的第一列是行名
    rownames.append(p[0])
    # 剩余部分就是该行对应的数据
    data.append([float(x) for x in p[1:]])
  return rownames,colnames,data
```

上述函数将数据集中的头一行数据读入了一个代表列名的列表，并将最左边一列读入了一个代表行名的列表，最后它又将剩下的所有数据都放入了一个大列表，其中的每一项对应于数据集中的一行数据。数据集中任一单元格内的计数值，都可以由一个行号和列号来唯一定位，此行号和列号同时还对应于列表 `rownames` 和 `colnames` 中的索引。

下一步我们来定义**紧密度**（closeness）。我们曾在第 2 章讨论过这一问题，在那一章中我们以欧几里德距离和皮尔逊相关度为例对两位影评者的相似程度进行了评价。在本章的例子中，一些博客比其他博客包含更多的文章条目，或者文章条目的长度比其他博客的更长，这样会导致这些博客在总体上比其他博客包含更多的词汇。皮尔逊相关度可以纠正这一问题，因为它判断的其实是两组数据与某条直线的拟合程度。此处，皮尔逊相关度的计算代码将接受两个数字列表作为参数，并返回这两个列表的相关度分值：

```
from math import sqrt
def pearson(v1,v2):
  # 简单求和
  sum1=sum(v1)
  sum2=sum(v2)

  # 求平方和
  sum1Sq=sum([pow(v,2) for v in v1])
  sum2Sq=sum([pow(v,2) for v in v2])

  # 求乘积之和
  pSum=sum([v1[i]*v2[i] for i in range(len(v1))])

  # 计算 r (Pearson score)
  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
```

请记住皮尔逊相关度的计算结果在两者完全匹配的情况下为 1.0，而在两者毫无关系的情况下则为 0.0。上述代码的最后一行，返回的是以 1.0 减去皮尔逊相关度之后的结果，这样做的目的是为了让相似度越大的两个元素之间的距离变得更小。

分级聚类算法中的每一个聚类，可以是树中的枝节点，也可以是与数据集中实际数据行相对应的叶节点（在本例中，即为一个博客）。每一个聚类还包含了指示其位置的信息，这一信息可以是来自叶节点的行数据，也可以是来自枝节点的经合并后的数据。我们可以新建一个 `bicluster` 类，将所有这些属性存放其中，并以此来描述这棵层级树。请在 *clusters.py* 中新建一个类，以代表“聚类”这一类型：

```
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
```

分级聚类算法以一组对应于原始数据项的聚类开始。函数的主循环部分会尝试每一组可能的配对并计算它们的相关度，以此来找出最佳配对。最佳配对的两个聚类会被合并成一个

新的聚类。新生成的聚类中所包含的数据，等于将两个旧聚类的数据求均值之后得到的结果。这一过程会一直重复下去，直到只剩下一个聚类为止。由于整个计算过程可能会非常耗时，所以不妨将每个配对的相关度计算结果保存起来，因为这样的计算会反复发生，直到配对中的某一项被合并到另一个聚类中为止。

请将 hcluster 算法加入 *clusters.py* 文件中：

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

  # 最开始的聚类就是数据集中的行
  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)

    # 遍历每一个配对，寻找最小距离
    for i in range(len(clust)):
      for j in range(i+1,len(clust)):
        # 用 distances 来缓存距离的计算值
        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)

    # 计算两个聚类的平均值
    mergevec=[
    (clust[lowestpair[0]].vec[i]+clust[lowestpair[1]].vec[i])/2.0
    for i in range(len(clust[0].vec))]

    # 建立新的聚类
    newcluster=bicluster(mergevec,left=clust[lowestpair[0]],
                         right=clust[lowestpair[1]],
                         distance=closest,id=currentclustid)

    # 不在原始集合中的聚类，其 id 为负数
    currentclustid-=1
    del clust[lowestpair[1]]
    del clust[lowestpair[0]]
    clust.append(newcluster)

  return clust[0]
```

因为每个聚类都指向构造该聚类时被合并的另两个聚类，所以我们可以递归搜索由该函数最终返回的聚类，以重建所有的聚类及叶节点。

为了运行分级聚类算法，须要启动一个 Python 会话，将该文件加载进来，然后针对试验数据，调用 hcluster 方法：

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

执行过程也许会花费一定的时间。将距离值保存起来可以极大地加快执行速度，但是对于算法而言，计算每一对博客的相关度仍然是必要的。为了加快这一过程，我们可以借助外部库来计算距离值。为了检视执行的结果，我们可以编写一个简单的函数，递归遍历聚类树，并将其以类似文件系统层级结构的形式打印出来。请将 printclust 函数添加到 *clusters.py* 中：

```
def printclust(clust,labels=None,n=0):
  # 利用缩进来建立层级布局
  for i in range(n): print ' ',
  if clust.id<0:
    # 负数标记代表这是一个分支
    print '-'
  else:
    # 正数标记代表这是一个叶节点
    if labels==None: print clust.id
    else: print labels[clust.id]

  # 现在开始打印右侧分支和左侧分支
  if clust.left!=None: printclust(clust.left,labels=labels,n=n+1)
  if clust.right!=None: printclust(clust.right,labels=labels,n=n+1)
```

这样的输出结果看起来不是非常的美观，并且对于读取博客列表这样的大数据集而言，这种做法会比较困难，不过它确实为我们提供了一个有关聚类算法是否工作良好的大体感觉。在下一节，我们将会看到如何建立一个图形版本的聚类树，这棵树更容易阅读，而且能够按比例缩放，从而可以显示出每个聚类的整体布局。

在 Python 会话中，请针对此前构造好的聚类调用上述函数：

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

由于输出列表包含了所有的 100 个博客，因此它非常冗长。下面是我们在运行上述数据集时，从中找到的一个聚类示例：

```
John Battelle's Searchblog
-
  Search Engine Watch Blog
  -
    Read/WriteWeb
    -
      Official Google Blog
      -
        Search Engine Roundtable
        -
          Google Operating System
          Google Blogoscoped
```

此处列出的是集合中的原始数据项。破折号代表的，是由两个或更多项合并而成的聚类。这是一个极好的寻找群组的例子；而且从中我们还可以发现，有如此众多与搜索相关的博客（search-related blogs）会出现在最为热门的博客订阅源中。通过仔细观察，我们还应该能够从中找到政治类博客的聚类、技术类博客的聚类，以及与撰写博客相关的聚类。

另外，从上述结果中我们可能也会注意到一些例外情况。一些博客的作者也许并没有撰写过相同主题的文章，但是聚类算法却会判断他们的单词频度具有相关性。这有可能是博客作者们写作风格的一种反应，当然也有可能只是基于数据下载当天的一个巧合而得出的结论。

绘制树状图

Drawing the Dendrogram

借助树状图，我们可以更加清晰地理解聚类。人们通常以树状图的形式来观察分级聚类的结果，这是因为树状图可以在一个相对狭小的空间里展示大量的信息。由于我们的树状图是图形的，并且要被保存成 JPG 格式，所以首先须要下载 Python Imaging Library（PIL），我们可从 *http://pythonware.com* 获取到该函数库。

PIL 有一个针对 Windows 平台的安装程序和一个针对其他平台的源代码发布包。关于下载和安装 PIL 的更多信息，请见附录 A。借助 PIL，我们可以非常轻松地生成带有文本和线条的图形，这是构造树状图所必需的。请将相关的 `import` 语句加入 *clusters.py* 文件的开始处：

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

首先，须要利用一个函数来返回给定聚类的总体高度。在确定图形的整体高度和放置不同节点的位置时，知道聚类的总体高度是很有必要的。如果聚类是一个叶节点（即它没有分支），则其高度为 1；否则，高度为所有分支高度之和。我们可以简单地将其定义成一个递归函数，并加入到 *clusters.py* 中：

```
def getheight(clust):
  # 这是一个叶节点吗？若是，则高度为 1
  if clust.left==None and clust.right==None: return 1

  # 否则，高度为每个分支的高度之和
  return getheight(clust.left)+getheight(clust.right)
```

除此以外，我们还须要知道根节点的总体误差。因为线条的长度会根据每个节点的误差进行相应的调整，所以我们须要根据总的误差值生成一个缩放因子（scaling factor）。一个节点的误差深度等于其下所属的每个分支的最大可能误差。

```
def getdepth(clust):
  # 一个叶节点的距离是 0.0
  if clust.left==None and clust.right==None: return 0
```

```
    # 一个枝节点的距离等于左右两侧分支中距离较大者，
    # 加上该枝节点自身的距离
    return max(getdepth(clust.left),getdepth(clust.right))+clust.distance
```

函数 drawdendrogram 为每一个最终生成的聚类创建一个高度为 20 像素、宽度固定的图片。其中的缩放因子是由固定宽度除以总的深度值得到的。该函数为图片建立相应的 draw 对象，然后在根节点的位置调用 drawnode 函数，并令其处于整幅图片左侧正中间的位置。

```
def drawdendrogram(clust,labels,jpeg='clusters.jpg'):
  # 高度和宽度
  h=getheight(clust)*20
  w=1200
  depth=getdepth(clust)

  # 由于宽度是固定的，因此我们须要对距离值做相应的调整
  scaling=float(w-150)/depth

  # 新建一个白色背景的图片
  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))

  # 画第一个节点
  drawnode(draw,clust,10,(h/2),scaling,labels)
  img.save(jpeg,'JPEG')
```

此处最为重要的函数是 drawnode，它接受一个聚类及其位置作为输入参数。函数取到子节点的高度，并计算出这些节点所在的位置，然后用线条将它们连接起来——包括一条长长的垂直线和两条水平线。水平线的长度是由聚类中的误差情况决定的。线条越长就越表明，合并在一起的两个聚类差别很大，而线条越短则越表明，两个聚类的相似度很高。请将 drawnode 函数加入到 *clusters.py* 中：

```
def drawnode(draw,clust,x,y,scaling,labels):
  if clust.id<0:
    h1=getheight(clust.left)*20
    h2=getheight(clust.right)*20
    top=y-(h1+h2)/2
    bottom=y+(h1+h2)/2
    # 线的长度
    ll=clust.distance*scaling
    # 聚类到其子节点的垂直线
    draw.line((x,top+h1/2,x,bottom-h2/2),fill=(255,0,0))

    # 连接左侧节点的水平线
    draw.line((x,top+h1/2,x+ll,top+h1/2),fill=(255,0,0))

    # 连接右侧节点的水平线
    draw.line((x,bottom-h2/2,x+ll,bottom-h2/2),fill=(255,0,0))
```

```
    # 调用函数绘制左右节点
    drawnode(draw,clust.left,x+l1,top+h1/2,scaling,labels)
    drawnode(draw,clust.right,x+l1,bottom-h2/2,scaling,labels)
  else:
    # 如果这是一个叶节点，则绘制节点的标签
    draw.text((x+5,y-7),labels[clust.id],(0,0,0))
```

为了生成图片，请在你的 Python 会话中输入：

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

执行上述代码将生成一个包含树状图的 *blogclust.jpg* 文件。树状图的样子应该如图 3-3 所示。为了更便于打印，或者不至于太过凌乱，如果愿意的话，我们也可以修改高度和宽度的设置。

列聚类

Column Clustering

同时在行和列上对数据进行聚类常常是很有必要的。当我们进行市场研究的时候，对消费群体进行分组可能是很有意义的，这将有助于我们摸清消费者的统计信息和产品的状况，还可能有助于我们确定哪些上架商品可以进行捆绑销售。在博客数据集中，列代表的是单词，知道哪些单词时常会结合在一起使用，可能是非常有意义的。

要利用此前编好的函数实现针对列的聚类，最容易的一种方式就是将整个数据集转置（rotate），使列（也就是单词）变成行，其中的每一行都对应一组数字，这组数字指明了某个单词在每篇博客中出现的次数。请将下列函数加入到 *clusters.py* 中：

```
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
```

现在，我们可以将矩阵进行转置，然后执行同样的聚类操作，并将最终的结果以树状图的形式绘制出来。因为单词的数量要比博客的多，所以整个执行过程花费的时间相对会更长一些。请记住，因为矩阵已经被转置，所以现在的标签已不再是博客，而变成了单词。

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

关于聚类有一点很重要：当数据项的数量比变量多的时候，出现无意义聚类的可能性就会增加。由于单词的数量比博客多很多，因此我们会发现，在博客聚类中出现的模式（pattern）要比单词聚类中出现的更为合理。不过即便如此，我们还是会找到一些很有意思的聚类，如图 3-4 所示。

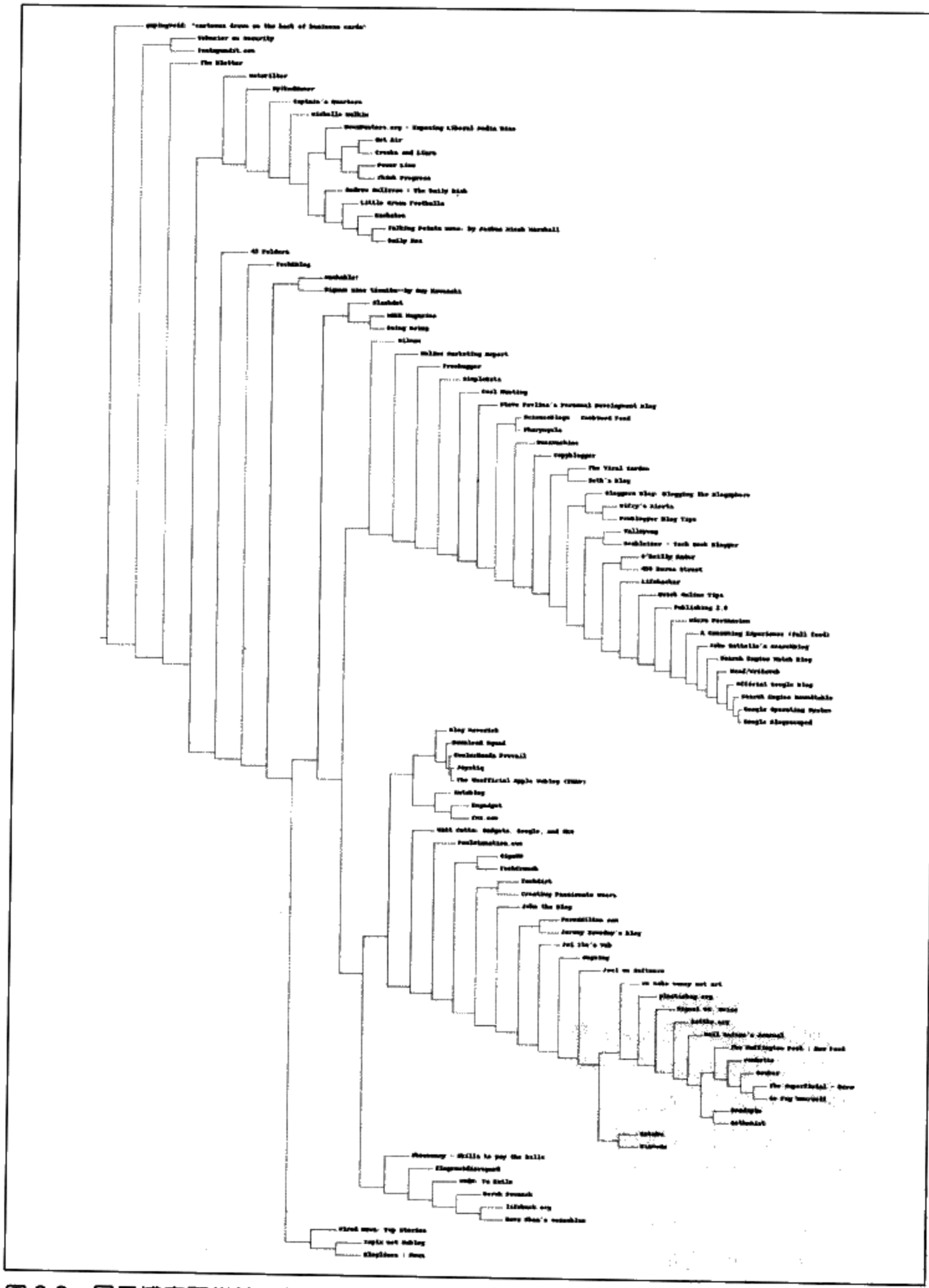


图 3-3：展示博客聚类情况的树状图

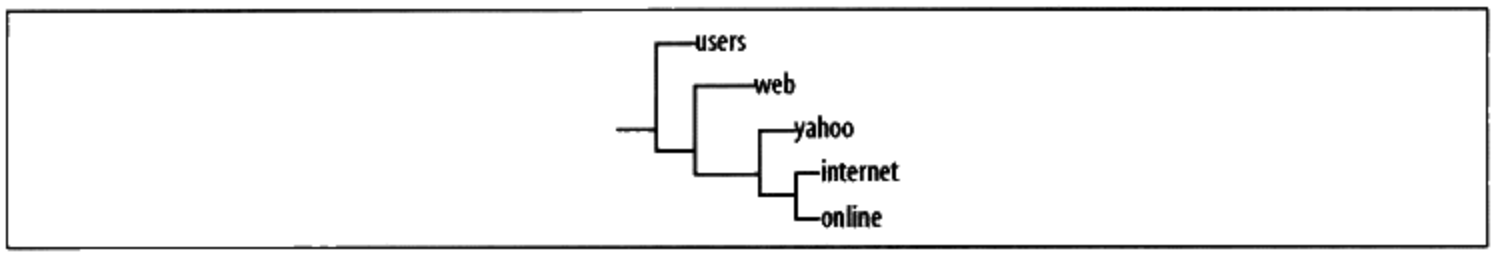


图 3-4：单词聚类给出了与在线服务相关的词汇

上述聚类清楚地显示了，当人们在博客中探讨在线服务或与 Internet 相关的话题时，经常会用到的一组词汇。另外，我们也可能会找到一些反映使用模式（usage patterns）的聚类，例如“fact”、“us”、“say”、“very”，以及“think”，这些单词的出现说明博客的写作风格是偏主观的（opinionated）。

K-均值聚类
K-Means Clustering

分级聚类的结果为我们返回了一棵形象直观的树，但是这种方法有两个缺点。在没有额外投入的情况下，树形视图是不会真正将数据拆分成不同组的，而且该算法的计算量非常惊人。因为我们必须计算每两个配对项之间的关系，并且在合并项之后，这些关系还得重新再计算，所以在处理很大规模的数据集时，该算法的运行速度会非常缓慢。

除了分级聚类外，另一种可供选择的聚类方法被称为 **K-均值聚类**。这种算法完全不同于分级聚类，因为我们会预先告诉算法希望生成的聚类数量，然后算法会根据数据的结构状况来确定聚类的大小。

K-均值聚类算法首先会随机确定 k 个**中心**位置（位于空间中代表聚类中心的点），然后将各个数据项分配给最临近的中心点。待分配完成之后，聚类中心就会移到分配给该聚类的所有节点的平均位置处，然后整个分配过程重新开始。这一过程会一直重复下去，直到分配过程不再产生变化为止。图 3-5 显示了这一过程，其中涉及了 5 个数据项和 2 个聚类。

在第 1 幅图中，两个中心点（以黑圆圈显示）的位置是随机选择的。第 2 幅图显示了算法将每个数据项都分配给了距离最近的中心点——在本例中，A 和 B 被分配给了上方的中心点，C、D 和 E 则被分配给了下方的中心点。在第 3 幅图中，中心位置已经移到了分配给原中心点的所有项的平均位置处。当再次进行分配时，我们可以看到现在的 C 距离上方的中心点较之以前更加接近了，而 D 和 E 则依然是距离下方中心点最近的两项。如此，最终所得的结果是 A、B、C 在一个聚类中，而 D、E 则在另一个聚类中。

实现 K-均值聚类算法的函数与分级聚类算法的一样，接受相同的数据行作为输入，此外它还接受一个调用者期望返回的聚类数（k）作为参数。请将下列代码添加到 *clusters.py* 中：

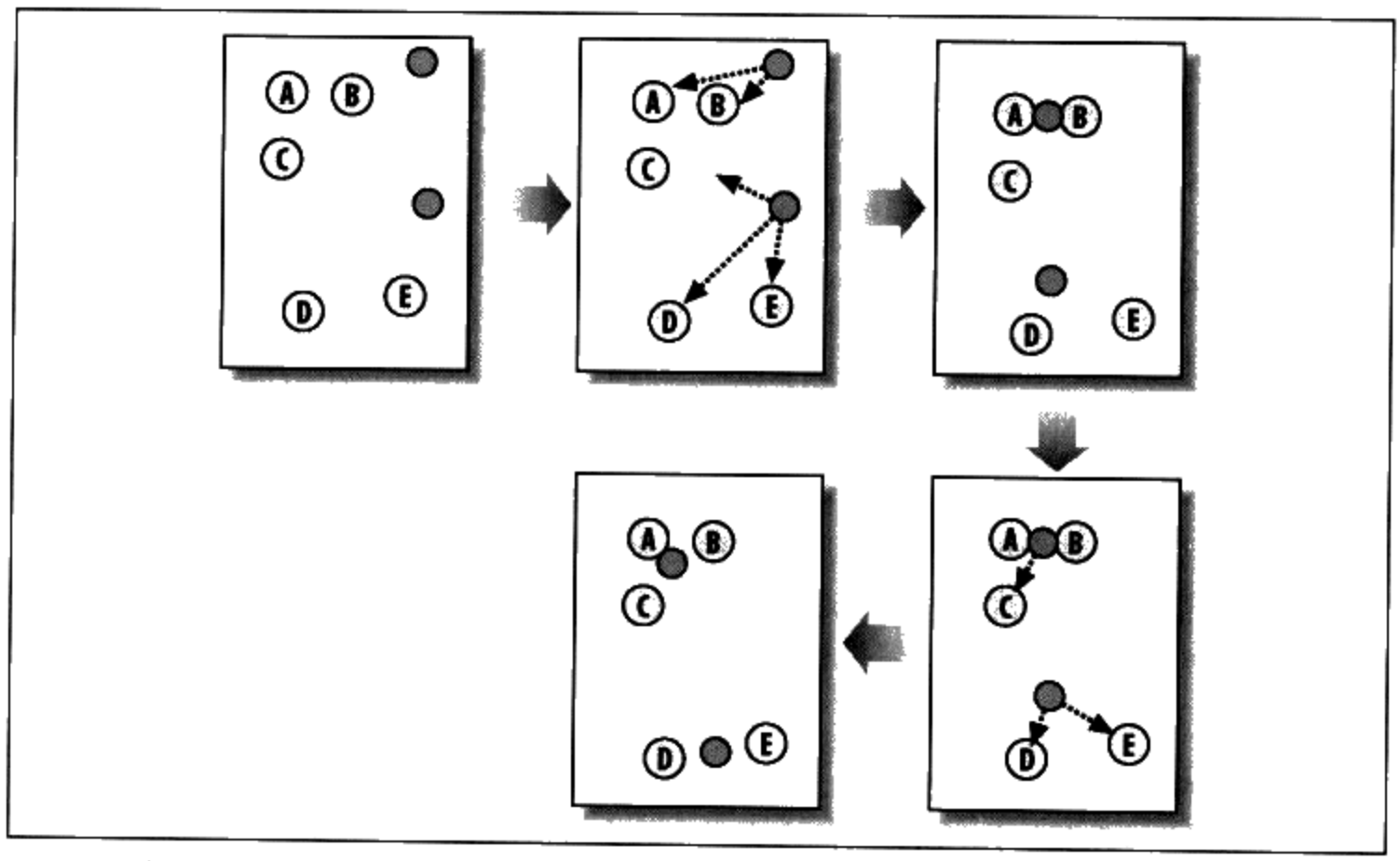


图 3-5：包含两个聚类的 K-均值聚类过程

```
import random

def kcluster(rows,distance=pearson,k=4):
  # 确定每个点的最小值和最大值
ranges=[(min([row[i] for row in rows]),max([row[i] for row in rows]))
for i in range(len(rows[0]))]

# 随机创建 k 个中心点
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 'Iteration %d' % t
  bestmatches=[[] for i in range(k)]

  # 在每一行中寻找距离最近的中心点
  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)

  # 如果结果与上一次相同，则整个过程结束
  if bestmatches==lastmatches: break
  lastmatches=bestmatches
```

```
    # 把中心点移到其所有成员的平均位置处
    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
```

上述代码在每个变量的值域范围内随机构造了一组聚类。当每次迭代进行的时候，算法会将每一行数据分配给某个中心点，然后再将中心点的数据更新为分配给它的所有项的平均位置。当分配情况与前一次相同时，迭代过程就结束了，同时算法会返回 k 组序列，其中每个序列代表一个聚类。与分级聚类相比，该算法为产生最终结果所须迭代的次数是非常少的。

由于函数选用随机的中心点作为开始，所以返回结果的顺序几乎总是不同的。根据中心点初始位置的不同，最终聚类中所包含的内容也可能会有所不同。

我们可以针对博客数据集试验一下该函数。算法的执行速度应该会比分级聚类更快一些：

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

现在，kclust 中应该包含了一组代表聚类的 ID 序列。请尝试用不同的 k 值进行聚类，看看对结果会有怎样的影响。

针对偏好的聚类

Clusters of Preferences

如今人们对社会化网络站点的兴趣日渐增长，这其中最为人称道的就是，我们可以从网站上获取到越来越多的数据，而所有这些数据都是网站用户无偿贡献的。在这些站点中，有一个叫做 Zebo（*http://www.zebo.com*）的，它鼓励人们在网站上建立账号，并将他们已经拥有的和希望拥有的物品列举出来。从广告商或社会评论家的角度而言，这些信息都是非常有价值的，因为他们可以借此找到方法，将偏好相近者很自然地分在一组。

获取数据和准备数据

Getting and Preparing the Data

本节我们将讨论如何利用 Zebo 站点来构造数据集，如何从站点将大量网页下载到本地加以解析，并从中提取出每位用户希望拥有的物品。如果你想跳过本节的内容，也可以从 *http://kiwitobes.com/clusters/zebo.txt* 处下载到一个预先构造好的数据集。

Beautiful Soup

Beautiful Soup 是一个解析网页和构造结构化数据表达形式的优秀函数库。它允许我们利用类型（type）、ID，或者任何其他的属性来访问网页内的任何元素，并获取到代表其内容的字符串。Beautiful Soup 还可以很好地处理包含不规范 HTML 标记的 Web 页面，当我们根据站点的内容来构造数据集时，这一点是非常有用的。

我们可以从 *http://crummy.com/software/BeautifulSoup* 下载到 Beautiful Soup。这是一个单独的 Python 文件，我们可以将它放到 Python 库所在路径下，或者放到工作路径——即启动 Python 解释器的路径下。

待 Beautiful Soup 安装完毕之后，我们就可以在 Python 解释器中使用它了：

```
>> 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'
```

要构造一个 *soup* 对象——这是 Beautiful Soup 描述 Web 页面的方式——只须利用页面内容对其进行初始化即可。我们可以将标签类型作为参数来调用 soup，比如上例中的“a”，调用的结果将返回一个属于该标签类型的对象列表。其中的每一个对象也都是可访问的（addressable），我们可以逐层深入地访问到对象的属性，以及其下所属的其他对象。

收集来自 Zebo 的结果

Scraping the Zebo Results

在 Zebo 上搜索到的网页，其结构是相当复杂的，但是我们很容易就可以判断出页面的哪些部分对应于物品的列表，因为它们都带有名为 `bgverdanasmall` 的 CSS 类。我们可以利用这一点来提取网页中的重要数据。请新建一个名为 *downloadzebodata.py* 的文件，并将下面的代码加入其中：

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

```
# 要去除的单词
dropwords=['a','new','some','more','my','own','the','many','other','another']

currentuser=0
for i in range(1,51):
  # 搜索“用户希望拥有的物品”所对应的URL
  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'):
    # 寻找带有bgverdanasmall类的表格单元格
    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:
        # 去除多余的单词
        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
```

上述代码将下载和解析从 Zebo 上搜索到的包含“用户希望拥有的物品”的前 50 个页面。因为所有物品的文字都是随意输入的，所以须要进行大量的清理工作，其中包括去除像“a”和“some”这样的单词，去除标点符号，以及将所有文本转换成小写。

待完成上述工作之后，代码首先会构造一个列表，其中包含的是超过 5 个人都希望拥有的物品，然后再构造一个以匿名用户为列、以物品为行的矩阵，最后再将该矩阵写入一个文件。请将下列代码加入到 *downloadzebodata.py* 文件的末尾处：

```
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')
```

请在命令行运行下列命令，以生成一个与博客数据集相同格式的名为 zebo.txt 的文件。和博客数据集相比，此处唯一的区别在于没有了计数，如果一个人希望拥有某件物品，那么我们将其标记为 1，否则就标记为 0：

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

定义距离度量标准

Defining a Distance Metric

皮尔逊相关度很适合于博客数据集，该数据集中所包含的是单词的实际统计值。而在此处，数据集却只有 1 和 0 两种取值，分别代表着有或无。并且，假如我们对同时希望拥有两件物品的人在物品方面互有重叠的情况进行度量，那或许是一件更有意义的事情。为此，我们采用一种被称为 **Tanimoto 系数**（Tanimoto coefficient）的度量方法，它代表的是交集（只包含那些在两个集合中都出现的项）与并集（包含所有出现于任一集合中的项）的比率。利用如下两个向量，我们可以很容易的定义出这样的度量：

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

  for i in range(len(v1)):
    if v1[i]!=0: c1+=1 # 出现在 v1 中
    if v2[i]!=0: c2+=1 # 出现在 v2 中
    if v1[i]!=0 and v2[i]!=0: shr+=1 # 在两个向量中都出现

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

上述代码将返回一个介于 1.0 和 0.0 之间的值。其中 1.0 代表不存在同时喜欢两件物品的人，而 0.0 则代表所有人都同时喜欢两个向量中的物品。

对结果进行聚类

Clustering Results

因为数据的格式与先前所用的相同，所以我们可以利用同样的函数来生成和绘制分级聚类。利用上面的函数并相应传入两个向量，我们很容易就可以实现聚类的功能；请将函数 `tanimoto` 加入到 *clusters.py* 中：

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

上述代码的执行会生成一个新的文件，*clusters.jpg*，其中包含的聚类反映了人们希望拥有的物品。利用下载得到的数据集生成的结果如图 3-6 所示。就市场营销的角度而言，此处并没有什么惊人的发现——希望得到 Xbox、PlayStation Portable 和 PlayStation 3 的人都是同属于一类的——不过我们的确也可以从中发掘出一些明显的群组，比如：有些人雄心勃勃（船、飞机、岛屿），有些人则热衷于寻找心灵的慰藉（朋友、爱情、幸福）。我们还可以注意到一些有趣的现象：希望拥有“钱”的人只想要一栋“房子”，而希望拥有“许多钱”的人则更倾向于要一栋“好房子”（nice house）。

通过改变初始的搜索条件，改变获取到的网页数量，或是将搜索“我想拥有的物品（I want）”改为搜索“我所拥有的物品（I own）”，并从搜索得到的结果中获取数据，我们也许还有可能会从中发现其他值得关注的物品群组。我们还可以尝试将矩阵转置，并对用户进行分组，通过收集年龄信息来了解人们在年龄上的划分情况，或许这样会得出更为有趣的结论也未可知。

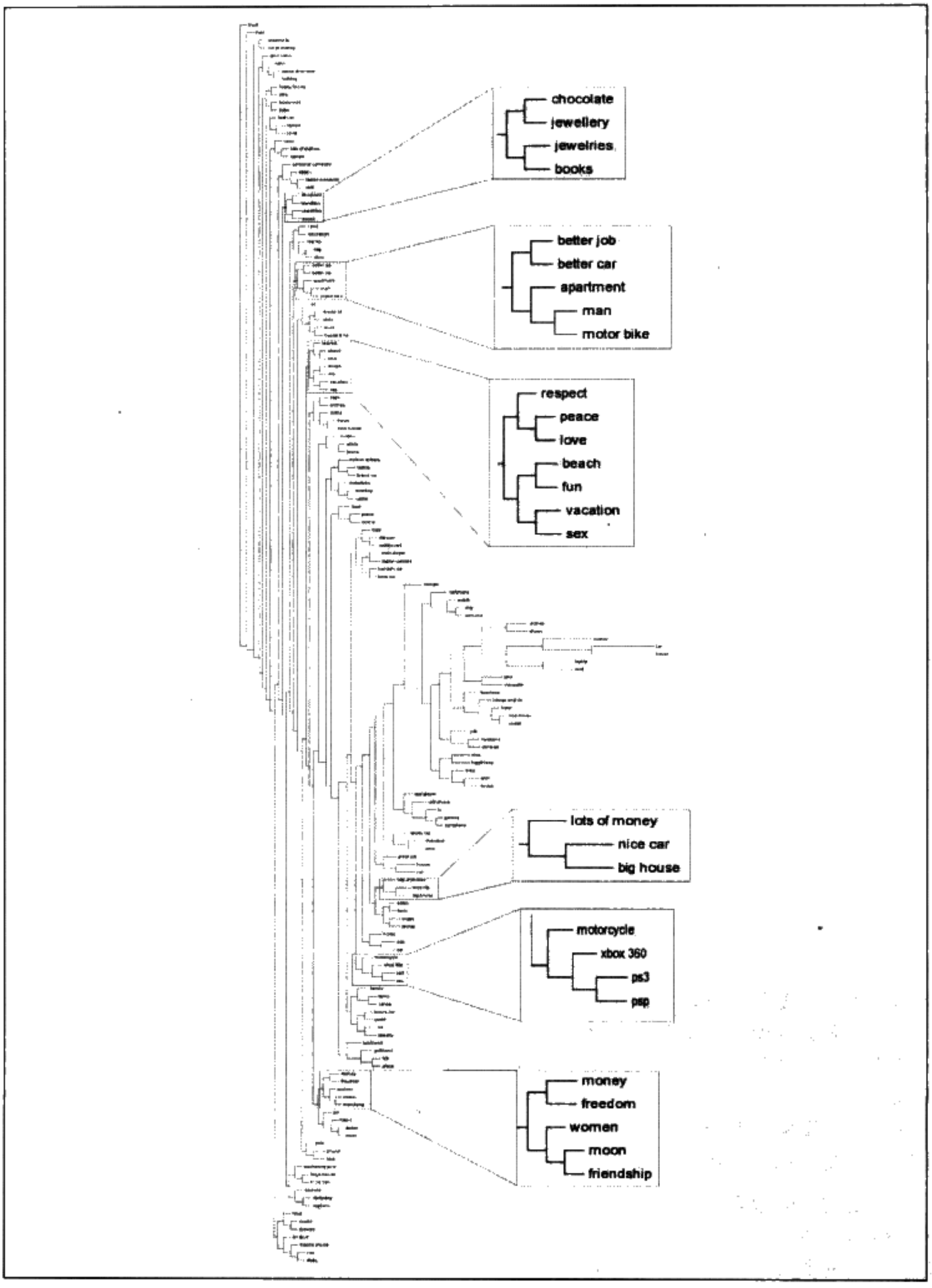


图 3-6：聚类所反映的是人们想要拥有的物品

以二维形式展现数据

Viewing Data in Two Dimensions

在本章中，我们已经采用了一种程式化的数据可视化表达方法，在二维空间中为大家演示了聚类算法的运用，利用了不同物品间的图上距离来指示它们彼此间的差异。由于在大多数真实生活的例子中，我们所要聚类的内容都不只包含两个数值，所以我们不可能按照前面的方法来采集数据并以二维的形式将其绘制出来。但是，为了要弄明白不同物品之间的关系，将它们绘制在一页纸上，并且用距离的远近来表达相似程度，又是一种非常有效的做法。

本节我们将介绍一种称为**多维缩放**（multidimensional scaling）的技术，利用这项技术，我们可以为数据集找到一种二维表达形式。算法根据每对数据项之间的差距情况，尝试绘制出一幅图来，图中各数据项之间的距离远近，对应于它们彼此间的差异程度。为了做到这一点，算法首先须要计算出所有项之间的目标距离。在博客数据集中，我们采用了皮尔逊相关度技术来对各数据项进行比较。此处有一个示例，如表 3-2 所示。

表 3-2：包含距离信息的示例矩阵

	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

下一步，我们将所有数据项（在本例中为博客）随机放置在二维图上，如图 3-7 所示。

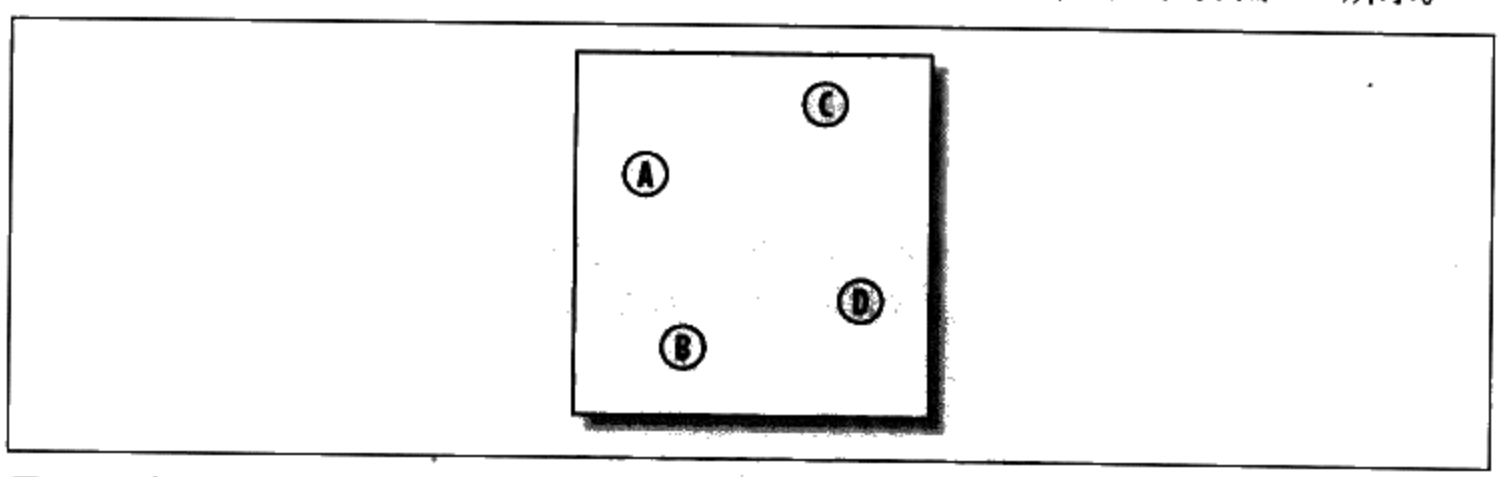


图 3-7：投影到二维坐标上的各数据项的初始位置

所有数据项两两间的当前距离值都是根据实际距离（即差平方之和）计算求得的，如图 3-8 所示。

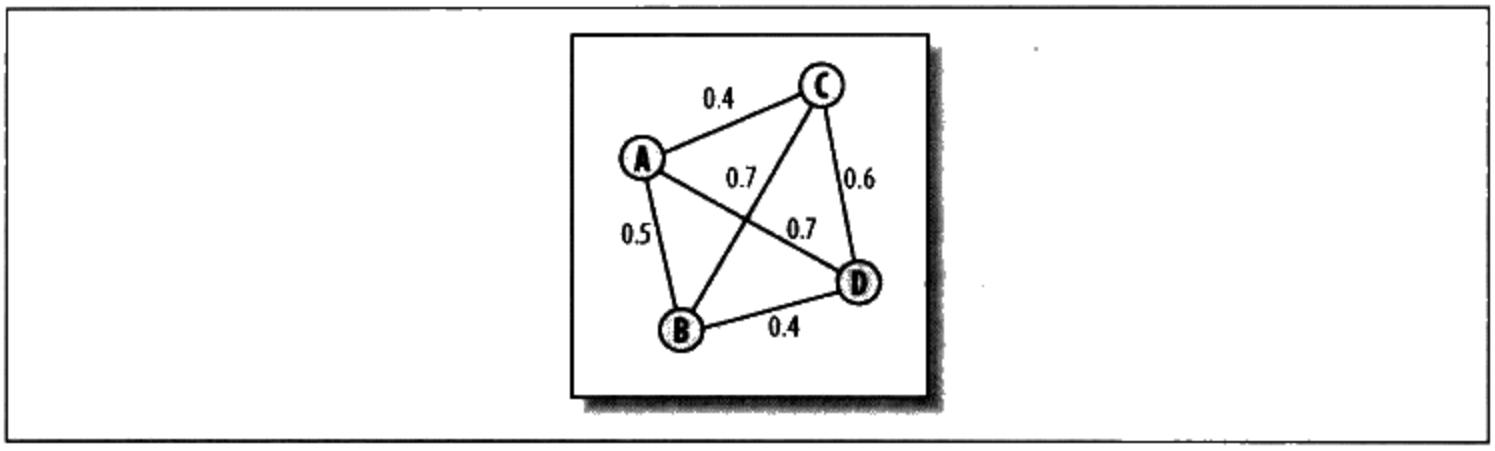


图 3-8：各项之间的距离

针对每两两构成的一对数据项，我们将它们的目标距离与当前距离进行比较，并求出一个误差值。根据误差的情况，我们会按照比例将每个数据项的所在位置移近或移远少许量。图 3-9 显示了我们对数据项 A 的施力情况。图中 A 与 B 之间的距离为 0.5，而两者的目标距离仅为 0.2，因此我们必须将 A 朝 B 的方向移近一点才行。与此同时，我们还将 A 推离了 C 和 D，因为它距离 C 和 D 都太近了。

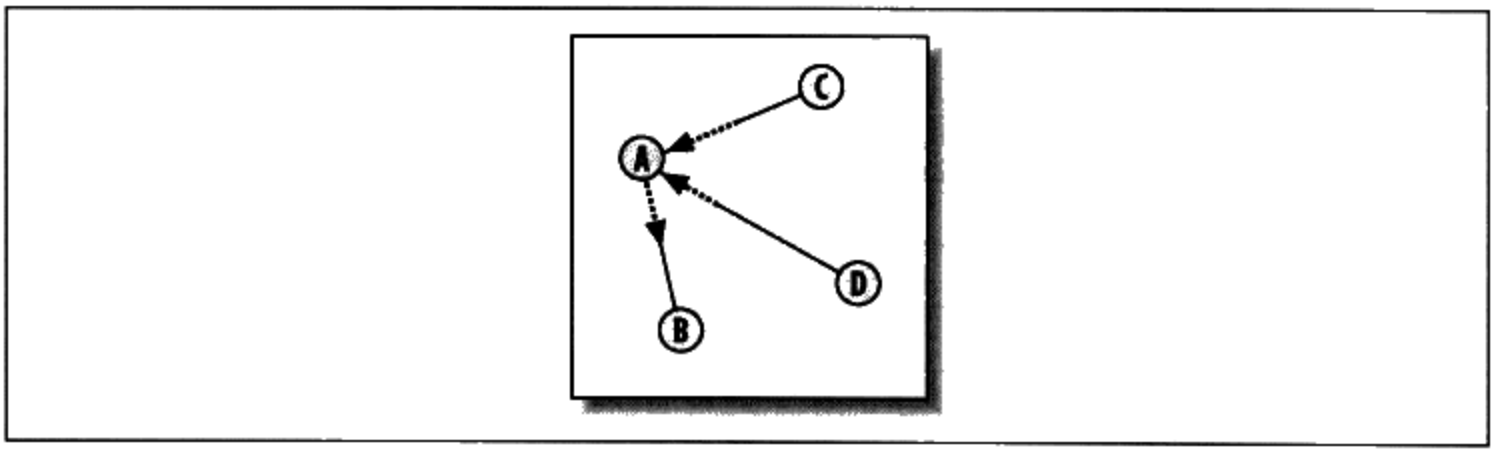


图 3-9：对数据项 A 的施力情况

每一个节点的移动，都是所有其他节点施加在该节点上的推或拉的综合效应。节点每移动一次，其当前距离和目标距离间的差距就会减少一些。这一过程会不断地重复多次，直到我们无法再通过移动节点来减少总体误差为止。

实现这一功能的函数接受一个数据向量作为参数，并返回一个只包含两列的向量，即数据项在二维图上的 X 坐标和 Y 坐标。请将该函数添加到 *clusters.py* 中：

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

  # 每一对数据项之间的真实距离
  realdist=[[distance(data[i],data[j]) for j in range(n)]
            for i in range(0,n)]

  outersum=0.0
```

```
    # 随机初始化节点在二维空间中的起始位置
    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):
      # 寻找投影后的距离
      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]))]))

      # 移动节点
      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
          # 误差值等于目标距离与当前距离之间差值的百分比
    errorterm=(fakedist[j][k]-realdist[j][k])/realdist[j][k]

          # 每一个节点都须要根据误差的多少，按比例移离或移向其他节点
          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

          # 记录总的误差值
          totalerror+=abs(errorterm)
      print totalerror

      # 如果节点移动之后的情况变得更糟，则程序结束
      if lasterror and lasterror<totalerror: break
      lasterror=totalerror

      # 根据 rate 参数与 grad 值相乘的结果，移动每一个节点
      for k in range(n):
        loc[k][0]-=rate*grad[k][0]
        loc[k][1]-=rate*grad[k][1]

    return loc
```

为了看到函数执行的效果，我们可以利用 PIL 再生成一幅图，根据新的坐标值，在图上标出所有数据项的位置及其对应的标签。

```
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')
```

要运行上述算法，请调用 scaledown 获得二维形式的数据集，然后再调用 draw2d 将其绘制出来：

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

图 3-10 显示了多维缩放算法的执行结果。虽然聚类的分布情况没有像树状图那样直观，但是我们仍然可以清晰地找出一些主题分组（topical grouping），比如靠近顶部的是与搜索引擎相关的集合。构成这一集合的所有博客与政治类博客及名人类博客的距离都非常的远。假如我们以三维的形式加以表现，也许聚类的效果会更好，但很显然，我们很难在纸上将这一效果展现出来。

图 3-10：部分博客空间的二维表示形式

有关聚类的其他事宜

Other Things to Cluster

本章研究了两个数据集，但是还有许多其他的工作我们可以去做。我们可以对第 2 章中来自 del.icio.us 网站的数据集进行聚类，从中寻找有关用户或书签的分组。利用与将博客订阅源转换成单词向量同样的方式，我们可以将任何一组下载到本地的网页解析成一个个的单词。

我们可以将本章提供的这些思路扩展到许多不同的领域，从中寻找到富有价值的结论——基于单词的信息公告栏，基于各种数字统计的来自 Yahoo! Finance 的公司信息，以及 Amazon 上的书评者排行。同样，研究像 MySpace 这样的大型社区网络，根据用户的好友关系，或者是利用用户可能提供的其他信息（喜欢的乐队、食物，等等）对人群进行聚类，也是一件很有意思的事情。

根据数据项所包含的参数信息将其绘制在一个空间范围之内，这样的概念将会成为本书中反复出现的一个主题。多维缩放是一种非常有效的方法，利用它我们可以将获取到的数据集以一种易于解释的方式形象化地展现出来。在缩放的过程中一些信息可能会丢失掉，明白这一点非常的重要，但是缩放后的结果应该会更加有助于我们理解算法的原理。

练习

Exercises

1. **请利用第 2 章中的 del.icio.us API，构造一个适合聚类的标签数据集** 针对该数据集分别运行分级聚类算法和 K-均值聚类算法。

2. **请修改解析博客的代码，以实现针对每个文章条目（entries）而非整个博客的聚类** 来自同一博客的不同条目能否聚类在一起？拥有相同日期信息的条目又如何？

3. **请尝试使用实际距离（即毕达哥拉斯距离）对博客进行聚类** 这样会对结果产生什么样的影响呢？

4. **找到曼哈顿（Manhattan）距离的定义** 为其编写一个函数，看看它是如何影响 Zebo 数据集的结果的。

5. **请修改 K-均值聚类函数** 令其在返回聚类结果的同时，一并返回所有数据项彼此之间的距离总和，以及它们各自的中心点位置。

6. **待完成第 5 个练习之后 请编写一个函数，令其选择不同的 k 值来运行 K-均值聚类算法** 看一看总的距离值是如何随着聚类数的增加而改变的？当处于哪个位置的时候，聚类数的多少对最终结果的影响才会变得微乎其微？

7. **在两个维度上的多维缩放易于打印** 不过这项技术也可以用于任意数量的维度。请尝试修改代码，以实现在一个维度上的缩放（即所有点都在一条直线上）。再尝试令其支持三个维度。

第 4 章

搜索与排名

Searching and Ranking

本章介绍的全文搜索引擎，允许人们在大量文档中搜索一系列单词，并根据文档与这些单词的相关程度对结果进行排名。全文搜索算法是最重要的集体智慧算法之一，人们在这一领域里所产生的新想法已经创造出了大量的财富。大家普遍相信，Google 从一个研究型项目迅速崛起为世界范围内最受欢迎的搜索引擎，这在很大程度上要归功于它的 PageRank 算法。作为标准算法的一个变体，我们将在本章中学习到一些有关 PageRank 的知识。

信息检索（Information retrieval）是一个有着悠久历史的巨大领域。本章仅就其中的一部分关键概念加以解释，不过我们会完整地介绍一个搜索引擎的构造过程，利用它来对一组文档建立索引（index），并向你提供某些进一步的改进建议。尽管我们在此处关注的是搜索和排名的算法，而在为大量网络资源建立索引时所需的基础设施方面并未考虑太多，不过我们构筑的搜索引擎对于数量接近 100 000 规模的网页搜集而言，应该是不成问题的。本章中，我们将会学到如何检索网页（crawl）、建立索引、对网页进行搜索，以及以多种不同方式对搜索的结果进行排名。

搜索引擎的组成

What's in a Search Engine?

建立搜索引擎的首要步骤是找到一种搜集文档的方法。有时，这可能会涉及针对网页的抓取——（在互联网上）先从一小组网页开始，然后再根据网页内的链接逐步追踪其他的网页。而有时，则可能需要我们在一组固定数量的文档范围内进行搜集，这些文档可能来自于某个公司的内部网络。

待搜集完文档之后，我们须要为这些文档建立索引。通常我们须要建立一张大的表，表中包含了文档及所有不同单词的位置信息。取决于具体的应用，文档本身不一定非要保存于数据库中；索引信息只须简单地保存一个指向文档所在位置的引用即可（例如文件系统的路径或 URL）。

待上述工作全部完成之后，剩下最后一步自然是通过查询返回一个经过排序的文档列表了。一旦建立了索引，根据给定单词或词组来获取文档就变得非常简单了，而此处真正的奥妙之处则在于结果的排列方式。我们可以选择许多不同的度量方法，每种方法都各有千秋，对它们适当加以调整，就可以改变网页的排名次序。通过对各种度量方法的学习，也许会使我们产生这样的想法：为什么我们不能对大型搜索引擎进行更多的有效控制呢（为什么我们不能告诉 Google，查询的单词在搜索时必须紧挨在一起呢）？本章我们将考查几种基于网页内容的度量方法，比如单词频度，然后再介绍几种基于网页外部信息的度量方法，如 PageRank 算法。PageRank 算法会考查其他网页对当前网页的引用情况。

在本章的最后，我们还将建立一个**神经网络**，用以对查询结果进行排名。通过了解人们在得到搜索结果以后都点击了哪些链接，神经网络会将搜索过程与搜索结果关联起来。它会利用这一信息来改变搜索结果的排列顺序，以更好地反映人们过去的点击情况。

为了运行本章中的示例，我们须要建立一个 Python 的模块，并取名 *searchengine*，其中包含两个类：一个用于检索网页和创建数据库；另一个则通过查询数据库进行全文搜索。在这些示例中，我们会用到 *SQLite* 数据库，不过你可以轻易地将其移植到传统的 C-S 数据库上。

首先，请新建一个名为 *searchengine.py* 的文件，并加入如下所示的 `crawler` 类和相应的方法签名，稍后我们将进一步完善该类：

```
class crawler:
  # 初始化crawler类并传入数据库名称
  def __init__(self,dbname):
    pass

  def __del__(self):
    pass
  def dbcommit(self):
    pass

  # 辅助函数，用于获取条目的id，并且如果条目不存在，就将其加入数据库中
  def getentryid(self,table,field,value,createnew=True):
    return None

  # 为每个网页建立索引
  def addtoindex(self,url,soup):
    print 'Indexing %s' % url

  # 从一个HTML网页中提取文字（不带标签的）
  def gettextonly(self,soup):
    return None

  # 根据任何非空白字符进行分词处理
  def separatewords(self,text):
    return None
```

```
# 如果 url 已经建过索引，则返回 true
def isindexed(self,url):
  return False

# 添加一个关联两个网页的链接
def addlinkref(self,urlFrom,urlTo,linkText):
  pass

# 从一小组网页开始进行广度优先搜索，直至某一给定深度，
# 期间为网页建立索引
def crawl(self,pages,depth=2):
  pass

# 创建数据库表
def createindextables(self):
  pass
```

一个简单的爬虫程序

A Simple Crawler

假设现在我们的硬盘中并没有成堆的 HTML 文档在等待建立索引，我将告诉大家如何构建一个简单的爬虫程序。该程序将接受一小组等待建立索引的网页，然后再根据这些网页内部的链接进而找到其他的网页，依此类推。这一过程被称为检索或**蛛行**（spidering）。

为了做到这一点，代码会通过网络将网页下载下来，并将网页递送给索引程序（indexer，将在下一节中介绍），然后再对网页进行解析，找出所有指向后续待检索网页的链接。所幸的是，有几个现成的函数库可以有助于我们完成这项工作。

针对本章中给出的示例，笔者已经事先建好了一份包含有数千个文件的拷贝，并将其置于 *http://kiwitobes.com/wiki*，供读者实验之用，这些文件都来自于 Wikipedia 网站。

当然，你可以根据自己的喜好对任何一组网页运行爬虫程序，不过假如你想将自己的运行结果与本章中给出的结果进行对照，那么不妨就使用上述站点中所提供的数据文件。

使用 urllib2

Using urllib2

urllib2 是一个绑定于 Python 的库，其作用是方便网页的下载——我们要做的全部工作就是提供一个 URL。本节中，我们将利用该函数库下载等待建立索引的网页。如果你想尝试一下 urllib2，就请启动你的 Python 解释器，并试着输入下列命令：

```
>> 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'
```

为了将一个网页的 HTML 源码存入字符串中，我们要作的全部工作就是建立起一个指向目标地址的网络连接，然后再读取网页的内容。

爬虫程序的代码

Crawler Code

爬虫程序将用到曾在第 3 章中介绍过的 Beautiful Soup API，这是一个为网页建立结构化表现形式的优秀函数库。对于 HTML 语法不是很规范的 Web 页面，Beautiful Soup 具有很好的容错性，这对于构建爬虫程序而言是非常有用的，因为我们不知道即将遇到的网页会是什么样的。有关下载和安装 Beautiful Soup 的更多信息，请见附录 A。

利用 urllib2 和 Beautiful Soup，我们可以建立起一个爬虫程序，接受一个等待建立索引的 URL 列表，检索网页链接，并找出其他须要建立索引的网页。首先，请在 *searchengine.py* 文件的首部加入下列 `import` 语句：

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

# 构造一个单词列表，这些单词将被忽略
ignorewords=set(['the','of','to','and','a','in','is','it'])
```

现在，我们可以为 crawler 中的函数填入代码了。实际上，crawler 目前还不能保存任何抓取到的内容，不过它会在执行期间将 URL 打印输出，这样我们就可以观察到它的执行过程了。我们须要将下列代码置于文件末尾处（使其成为 `crawler` 类的一部分）：

```
  def crawl(self,pages,depth=2):
    for i in range(depth):
      newpages=set()
      for page in pages:
        try:
          c=urllib2.urlopen(page)
        except:
          print "Could not open %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]  # 去掉位置部分
            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
```

该函数循环遍历网页列表，并针对每个网页调用 `addtoindex` 函数（目前该函数除了将 URL

打印输出外并没有做其他工作，不过下一节我们就将对其进行充实）。随后，该函数利用 Beautiful Soup 取到网页中的所有链接，并将这些链接加入到一个名为 newpages 的集合中。在循环结束之前，我们将 newpages 赋给 pages，而后这一过程将再次循环。

上述函数也可以采用递归的形式进行定义，如果是这样，每个链接就都会触发函数的再次调用。但是，广度优先的搜索方式可以使代码的后续修改更为容易，我们可以一直持续进行检索，也可以将未经索引的网页列表保存起来，以备后续再行检索之用。同时，这样也避免了栈溢出的风险。

可以在 Python 的解释器中测试一下该函数（由于没有必要让检索过程结束，因此如果你想结束运行，就请按 Ctrl-C）

```
>> import searchengine
>> pagelist=['http://kiwitobes.com/wiki/Perl.html']
>> crawler=searchengine.crawler('')
>> crawler.crawl(pagelist)
Indexing http://kiwitobes.com/wiki/Perl.html
Could not open http://kiwitobes.com/wiki/Module_%28programming%29.html
Indexing http://kiwitobes.com/wiki/Open_Directory_Project.html
Indexing http://kiwitobes.com/wiki/Common_Gateway_Interface.html
```

也许你已经注意到某些网页有重复。上述代码中有一处调用了另一个函数——isindexed，该函数将决定一个网页在被加入 newpages 之前是否已经在近期做过了索引。这样，我们就可以在任何时候针对任意的 URL 列表调用检索函数，而不必担心无谓的重复劳动了。

建立索引

Building the Index

接下来，我们须要为全文索引建立数据库。正如笔者此前曾经提到过的，索引对应于一个列表，其中包含了所有不同的单词、这些单词所在的文档，以及单词在文档中出现的位置。在本例中，我们将对出现在网页中的真正的文本内容进行考查，并忽略非文本元素。我们还会为每个单词建立索引，并去除所有标点符号。用于分词的函数不算高效，不过对于构建一个初级的搜索引擎而言，这已经足够了。

对不同数据库软件的介绍或数据库服务器的安装已经超出了本书的讨论范围，本章将向你展示利用 SQLite 保存索引的方法。SQLite 是一个嵌入式数据库，其安装过程非常简单，它将整个数据库存入了一个文件之中。由于 SQLite 是利用 SQL 语言进行查询的，因此将示例代码修改成使用其他数据库应该不会很难。SQLite 的 Python 实现被称为 pysqlite，我们可以从 *http://initd.org/tracker/pysqlite* 处下载到它。

pysqlite 有一个 Windows 安装程序，还有一些针对其他操作系统的安装说明。附录 A 包含了有关获取与安装 pysqlite 的更多信息。

待安装完 SQLite 之后，请将下列代码行加入 *searchengine.py* 的起始处：

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

我们还须要修改__init__、__del__，以及 dbcommit 方法，以便打开和关闭数据库：

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

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

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

建立数据库 Schema

请不要急着运行代码——我们还须要准备数据库表。为了实现基本的索引功能，我们须要建立 5 张表。第一张表（urllist）保存的是已经过索引的 URL 列表。第二张表（wordlist）保存的是单词列表，第三张表（wordlocation）保存的是单词在文档中所处位置的列表。还剩下两张表则保存了介于文档之间的链接信息。link 表保存了两个 URL ID，指明从一张表到另一张表的链接关系，而 linkwords 表则利用字段 wordid 和 linkid 记录了哪些单词与链接实际相关。完整的数据库 schema 如图 4-1 所示。

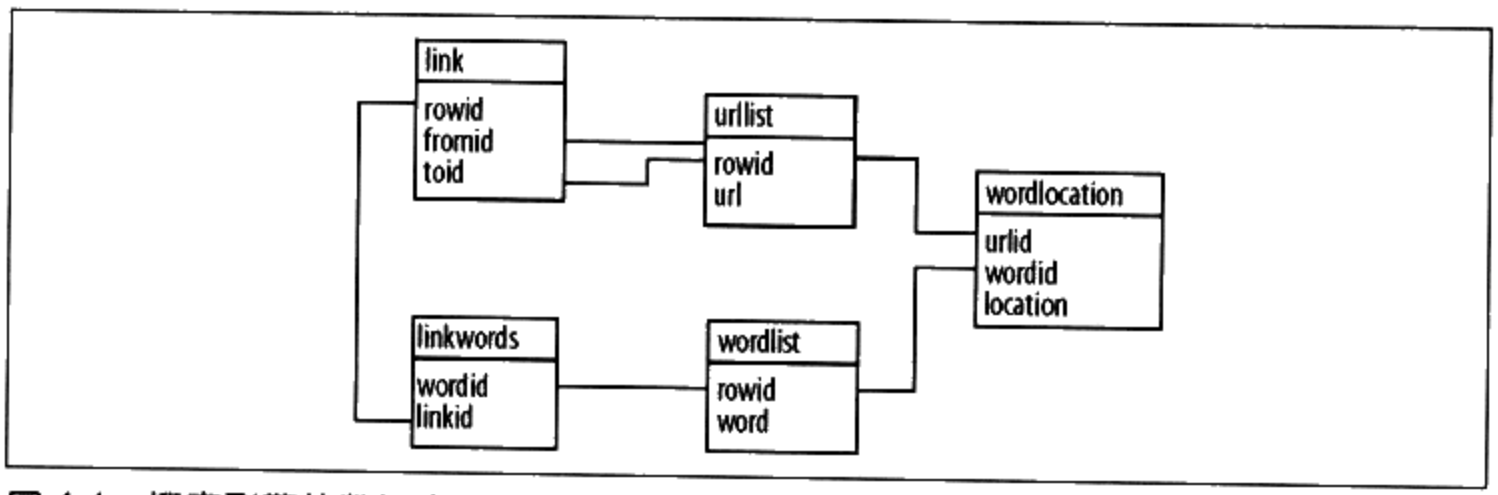


图 4-1：搜索引擎的数据库 schema

所有 SQLite 中的表默认都有一个名为 rowid 的字段，因此没必要显式地为这些表指定 ID 字段。为了建立一个函数将数据填入所有数据表中，请将下列代码加入 *searchengine.py* 文件的末尾处，使之成为 crawler 类的一部分：

```
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()
```

该函数的用途是为即将用到的所有表建立 schema，并建立一些旨在加快搜索速度的索引。这些索引非常重要，因为数据集可能会变得非常巨大。在你的 Python 会话中输入下列命令，建立一个名为 *searchindex.db* 的数据库：

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

稍后，我们还会在 schema 中再加入一张表，以根据外部回指链接（inbound link）的计数情况来评价度量值。

在网页中查找单词

Finding the Words on a Page

从网上下载的文件都是 HTML 格式的，其中包含有大量的标签、属性，以及其他不在索引范围内的信息。我们首先要从网页中提取出所有的文字部分。为此，我们可以对文本节点进行搜索，搜集所有的文字内容。请将下列代码加入 `gettextonly` 函数：

```
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()
```

该函数返回一个长字符串，其中包含了网页中的所有文字。它以递归向下的方式对 HTML 文档对象模型进行遍历，并找出其中的文本节点。在网页中，散布在各章节内的文字内容被分散于不同的段落。为了利于稍后某些度量的计算，在这一阶段保留各章节的前后顺序是很重要的。

接下来是 `separatewords` 函数，该函数将字符串拆分成一组独立的单词，以便我们将其加入到索引之中。要完美地做到这一点，可能未必如你所想的那样简单，已经有大量相关的研究旨在改进这项技术。不过，对于本章中的示例而言，将任何非字母或非数字的字符作为分隔符就已经足够了。我们还可以利用正则表达式来进行分词。请将 `separatewords` 的定义替换为下列代码：

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

由于该函数将任何非字母非数字字符都看作是分隔符，因此对于英文单词的提取是不会有问题的，但是这种方式无法正确处理类似“C++”这样的词汇（尽管搜索“python”是没有问题的）。为了令算法更好地适应于各种不同类型的搜索，我们可以尝试一下正则表达式。

提示：还有一项可能要做的工作，是利用某种**词干提取算法**（stemming algorithm）去除掉单词的后缀。词干提取算法试图将单词转换成对应的词干。例如：将单词“indexing”变成“index”。这样，人们在搜索单词“index”时同样也会得到包含单词“indexing”的文档。要做到这一点，我们可以在检索文档期间提取单词的词干，也可以在搜索查询期间提取单词的词干。有关词干提取的完整讨论超出了本章的讨论范围，不过我们可以在 *http://www.tartarus.org/~martin/PorterStemmer/index.html* 找到一个很有名的词干提取的 Python 实现——*Porter Stemmer*。

加入索引

Adding to the Index

接下来，我们将实现 addtoindex 方法。该方法会调用曾在前面章节中定义过的两个函数，得到一个出现于网页中的单词的列表。然后，它会将网页及所有单词加入索引，在网页和单词之间建立关联，并保存单词在文档中出现的位置。对于本章的示例而言，单词的位置就是其在列表中的索引号。

以下是 addtoindex 的代码：

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

  # 获取每个单词
  text=self.gettextonly(soup)
  words=self.separatewords(text)

  # 得到 URL 的 id
  urlid=self.getentryid('urllist','url',url)

  # 将每个单词与该 url 关联
  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))
```

同样，我们还须要更新辅助函数 getentryid。该函数的作用是返回某一条目的 ID。如果条目不存在，则程序会在数据库中新建一条记录，并将 ID 返回：

```
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]
```

最后，我们还须要实现 isindexed 函数。该函数的作用是判断网页是否已经存入数据库，如果存在，则判断是否有任何单词与之关联：

```
def isindexed(self,url):
  u=self.con.execute \
    ("select rowid from urllist where url='%s'" % url).fetchone()
  if u!=None:
    # 检查它是否已经被检索过了
    v=self.con.execute(
    'select * from wordlocation where urlid=%d' % u[0]).fetchone()
    if v!=None: return True
  return False
```

现在，我们可以再次执行 crawler，并在程序运行期间真正为网页建立索引了。可以在你的交互会话中输入下列命令：

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

crawler 的运行可能会花费较长的时间。与其等待它结束，笔者建议大家不妨从 *http://kiwitobes.com/db/searchindex.db* 处下载一份事先加载好的 *searchindex.db* 拷贝，并将其保存到你的 Python 代码所在的同一目录下。

如果你想确认检索过程是否正常执行，可以通过数据库查询操作，试着检查一下某个单词对应的条目：

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

此处返回的是所有由包含单词“word”的 URL ID 所构成的列表，这意味着我们已经成功执行了一次全文搜索。这是一个很好的开始，不过目前的代码一次只能处理一个单词，而且只能以文档当初被加载的顺序返回文档。下一节将会告诉你如何对这一功能进行扩展，实现在一次查询中利用多个单词进行搜索。

查询

Querying

现在我们已经有了一个可用的 crawler 程序和经过索引的大堆文档，接下来可以开始准备搜索引擎的搜索部分了。首先，请在 *searchengine.py* 中新建一个用于搜索的类：

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

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

wordlocation 表为连接单词与数据表提供了一种简单的方法，这样一来，考查哪些网页包含某个单词就会变得非常容易。不过，除非搜索引擎允许多词搜索，否则其功能就非常有限。为此，我们须要一个查询函数，接受一个查询字符串作为参数，并将其拆分为多个单词，然后构造一个 SQL 查询，只查找那些包含所有不同单词的 URL。请将该函数加到 searcher 类的定义中：

```
  def getmatchrows(self,q):
    # 构造查询的字符串
    fieldlist='w0.urlid'
    tablelist=''
    clauselist=''
    wordids=[]

    # 根据空格拆分单词
    words=q.split(' ')
    tablenumber=0

    for word in words:
      # 获取单词的 ID
      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

    # 根据各个组分，建立查询
    fullquery='select %s from %s where %s' % (fieldlist,tablelist,clauselist)
    cur=self.con.execute(fullquery)
    rows=[row for row in cur]

    return rows,wordids
```

该函数看起来有一些复杂，不过它所做的只是为列表中的每个单词建立一个指向wordlocation表的引用，并根据对应的URL ID将它们连结起来进行联合查询（如图4-2）。

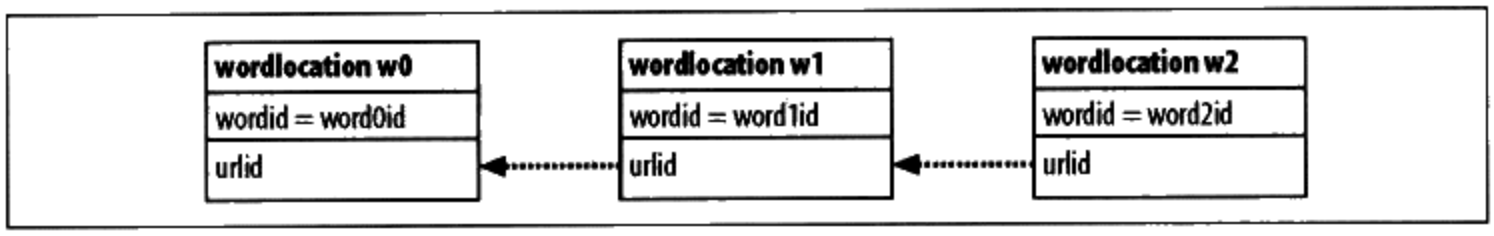


图4-2：针对getmatchrows的数据库表连接关系

因此，一个涉及两个单词（对应ID为10和17）的查询如下所示：

```
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
```

请尝试调用一下该函数，开始我们的首次多词搜索：

```
>> 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),..
```

我们注意到，根据单词位置的不同组合，此处每个URL ID会返回多次。后续几节中我们将会介绍几种对搜索结果进行排名的方法。其中，**基于内容的排名法**（Content-based ranking）是根据网页的内容，利用某些可行的度量方式来对查询结果进行判断的。而**外部回指链接排名法**（Inbound-link ranking）则是利用站点的链接结构来决定查询结果中各项内容的重要程度的。除此以外，我们还将介绍一种方法，能够通过考查人们在搜索时对搜索结果的实际点击情况，逐步改善搜索排名。

基于内容的排名
Content-Based Ranking

到目前为止，我们已经成功获得了与查询条件相匹配的网页。不过，其返回结果的排列顺序却很简单，即其被检索时的顺序。而面对大量的网页，为了能够从中找出与查询真正匹配的页面，我们就必须要在大量毫不相干的内容中逐一进行浏览，以搜寻任何对查询条件中某一部分内容有所提及的结果。为了解决这一问题，我们须要找到一种能够针对给定查询条件为网页进行评价的方法，并且能在返回结果中将评价最高者排在最前面。

本节我们将对几种只依据查询条件和网页内容进行评价计算的方法进行考查。这些评价度量包括以下三种。

单词频度

位于查询条件中的单词在文档中出现的次数能有助于我们判断文档的相关程度。

文档位置

文档的主题有可能会出现在靠近文档的开始处。

单词距离

如果查询条件中有多个单词，则它们在文档中出现的位置应该靠得很近。

早期的搜索引擎通常只利用了上述几种度量方式，而且据此便能得到很有价值的结果。后续几节中，我们将会介绍如何利用网页以外的信息进一步改善搜索结果，例如：考查外部回指链接（incoming link）的数量和质量。

首先，我们须要一个新的方法，该方法将接受查询请求，将获取到的行集置于字典中，并以格式化列表的形式显示输出。请将下列函数加入 searcher 类中：

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

  # 此处是稍后放置评价函数的地方
  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))
```

query 方法现在还没有对结果进行任何评价，不过它的确输出了 URL 和代表评价值的占位符：

```
>> 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
...
```

此处最为重要的函数是 getscoredlist，我们将在本节中实现该函数。待我们加入评价函数以后，便可添加对 weights 列表的调用（以黑体显示的代码行），并开始获得一些真实的评价。

归一化函数

Normalization Function

此处介绍的所有评价方法返回的都是包含 URL ID 与数字评价值的字典。令事情变得复杂化的是，有的评价方法分值越大越好，而有的则分值越小越好。为了对不同方法的返回结果进行比较，我们需要一种对结果进行归一化处理的方法，即，令它们具有相同的值域及变化方向。

归一化函数将接受一个包含 ID 与评价值的字典，并返回一个带有相同 ID，而评价值则介于 0 和 1 之间的新字典。函数根据每个评价值与最佳结果的接近程度(最佳结果的对应值为 1)，对其做了相应的缩放处理。我们所要做的全部工作，就是将评价值列表传入该函数，并指明数值越小越好，还是越大越好：

```
def normalizescores(self,scores,smallIsBetter=0):
  vsmall=0.00001 # 避免被零整除
  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()])
```

每个评价函数都会调用该函数，将结果进行归一化处理，并返回一个介于 0 和 1 之间的值。

单词频度

Word Frequency

这种方法以单词频度作为度量手段，根据查询条件中的单词在网页中出现的次数对网页进行评价。假如要搜索“python”，我们更希望得到的是一个内容中多次提到该单词的有关 Python（或 pythons）语言的网页；而不是一个关于某位音乐家的网页，可能这位音乐家在文章的末尾处偶尔提到了他有一条宠物蟒蛇（译注 1）。

单词频度函数如下所示。我们可以将其加入 searcher 类中：

```
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)
```

该函数建立了一个字典，其中包含了为行集中每个唯一的 URL ID 所建的条目，函数还对每个单词的出现次数进行了计数，随后又对评价值做了归一化处理（在本例中，分值越大越好），并返回结果。

为了在返回结果中使用频度评价算法，请修改 getscoredlist 中的 weights 一行：

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

现在，我们可以再试着搜索一次，看看这种评价度量的效果如何。

译注 1：蟒蛇的英文即为 python。

```
>> 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
...
```

上述返回结果将涉及“Functional programming”的网页放在了最前面，其后所跟的是另外几个相关的网页。请注意，对“Functional programming”的评价结果是紧随其后者的 4 倍。大多数搜索引擎都不会将评价结果告诉最终用户，但是对某些应用而言，这些评价值可能会非常有用。例如，也许我们希望在结果超出某个域值的时候，直接向用户返回排名最靠前的内容，或者希望根据返回结果的相关程度，按一定比例的字体大小加以显示。

文档位置

另一个判断网页与查询条件相关程度的简单度量方法，是搜索单词在网页中的位置。通常，如果一个网页与待搜索的单词相关，则该单词就更有可能在靠近网页开始处的位置出现，或者甚至是出现在标题中。利用这一点，搜索引擎可以对待查单词在文档中出现越早的情况给予越高的评价。所幸的是，网页已建立了索引，单词在文档中所处的位置也已记录了下来，而网页的标题则位于列表中的第一项。

请将该方法加入 searcher 类中：

```
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)
```

请记住，行集中每一行的第一项是 URL ID，后面紧跟的是所有各待查单词的位置信息。每个 ID 可以出现多次，每次对应的是不同的位置组合。针对每一行，该方法将会计算所有单词的位置之和，并将这一结果与迄今为止的最佳结果进行对比判断。然后将最终结果传入归一化处理函数之中。注意，smallIsBetter 意味着，位置之和最小的 URL 获得的评价值为 1.0。

如果想看一下只使用位置评价法的效果，就请将 weights 一行修改如下：

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

现在，请在解释器中试着再查询一次：

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

我们会发现，“Functional programming”依然是胜出者，不过其他位于前列的查询结果现在已经变成了函数式编程语言的例子了。在上一次搜索返回的结果中，虽然待查询的单词也出现了好多次，但是这通常是涉及编程语言方面的讨论。而在本次搜索中，程序对单词在首句中出现的情况给予了更高的评价值（例如，“Haskell is a standardized pure functional programming language”）。

截止目前所介绍的所有度量方法中，没有任何一种方法对于每一种情况而言都是最优的，认识到这一点很重要。取决于搜索者的意图，上述各种搜索结果都是有效的；而且对于一组特定的文档与应用而言，为了给出最佳结果，不同的加权组合也是必要的。我们可以对`weights`一行按类似如下的方式进行修改，试一下为两种度量分别配以不同的权重：

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

请尝试不同的权重和查询，看看对查询结果的影响如何。

相比于单词频度，位置度量更难于作假。网页的作者可以将一个单词置于文档的开始处，然后不断地加以重复，但是这对结果不会有任何影响。

单词距离
Word Distance

当查询中包含多个单词时，寻找单词彼此间距很近的网页往往是很有意义的。大多数时候，在进行多词查询时，人们时常会关注于那些在概念意义上与这些单词有关联的网页。这要比大多数搜索引擎所支持的以引号相括的短语搜索更为宽松。在那种情形下，单词必须以正确的顺序出现，且中间不夹杂任何额外的单词——而在此处，这种度量方法将允许单词顺序不一，也允许单词间夹带其他的单词。

`distancescore`函数看起来非常类似于`locationscore`：

```
def distancescore(self,rows):
  # 如果仅有一个单词，则得分都一样
  if len(rows[0])<=2: return dict([(row[0],1.0) for row in rows])

  # 初始化字典，并填入一个很大的数
  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)
```

此处的主要区别在于，当函数循环遍历单词的位置时（以黑体显示的代码行），它会计算出每个位置与上一个位置间的差距。由于查询会返回每一种距离组合，因此函数将确保找出总距离的最小值。

如果你愿意的话，可以单独测试一下单词距离这一度量。不过假如它与其他度量组合使用，则会有更好的表现。请尝试将 distancescore 加入 weights 列表并修改权重值，看一下它对不同的查询结果有何影响。

利用外部回指链接

Using Inbound Links

到目前为止，我们对评价度量的讨论都是基于网页内容的。尽管许多搜索引擎依然采用这种方式，不过我们往往可以通过考查外界就该网页所提供的信息——尤其是谁链向了该网页，以及他们对该网页的评价，来进一步改善搜索结果。当对存在可疑内容的网页或垃圾内容制造者生成的网页建立索引时，这一方法特别管用，因为与包含真实内容的网页相比，这些网页被他人引用的可能性非常的小。

由于本章开始时所建立的网路爬虫程序早已获取到与链接有关的所有重要信息，因此我们无须对它做任何修改。对于每一个遇到的链接，links 表中记录了与其源和目的相对应的 URL ID，而且 linkwords 表还记录了单词与链接的关联。

简单计数

Simple Count

处理外部回指链接最为简单的做法，是在每个网页上统计链接的数目，并将链接总数作为针对网页的度量。科研论文的评价就经常采用这样的方式，人们将论文的重要程度与其他论文对该论文的引用次数联系了起来。下面的评价函数，通过对查询 link 表所得行集中的每个唯一的 URL ID 进行计数，建立起了一个字典。随后，函数返回一个经过归一化处理的评价结果：

```
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)
```

很显然，单独使用这一度量只会简单地返回匹配搜索条件的所有网页，这些网页仅根据其拥有的外部回指链接数进行排序。在数据集中，“Programming language”与“Python”相比有更多的外部回指链接，但是假如我们想搜索“Python”，那么也许我们更希望在结果中最先看到的是与“Python”相关的内容。为了将相关性与排名结合起来，我们须要结合使用外部回指链接与此前介绍过的任何一种度量方法。

上述算法对每个外部回指链接都给予了同样的权重，这样做即公平又合理，这种方法在操作上具有开放性，因为如果人们希望增加某个网页的评价值，他们只须建立起若干站点，并将链接都指向该网页即可。除此以外，有时人们也可能会对那些受热门站点关注的搜索结果更加感兴趣。接下来，我们将会看到在计算排名的过程中，如何令来自热门网页的链接拥有更高的权重值。

PageRank 算法

PageRank 算法是由 Google 的创始人发明的，现在基于这一思路的各种变体已被所有大型搜索引擎采用。该算法为每个网页都赋予了一个指示网页重要程度的评价值。网页的重要性是依据指向该网页的所有其他网页的重要性，以及这些网页中所包含的链接数求得的。

提示：理论上，PageRank（以其发明者之一 Larry Page 命名）计算的是某个人在任意次链接点击之后到达某一网页的可能性。如果某个网页拥有来自其他热门网页的外部回指链接越多，人们无意间到达该网页的可能性也就越大。当然，如果用户始终不停地点击，那么他们终将到达每一个网页，但是大多数人在浏览一段时间之后都会停止点击。为了反映这一情况，PageRank 还使用了一个值为 0.85 的**阻尼因子**，用以指示用户持续点击每个网页中链接的概率为 85%。

图 4-3 给出了一组网页与链接的例子。

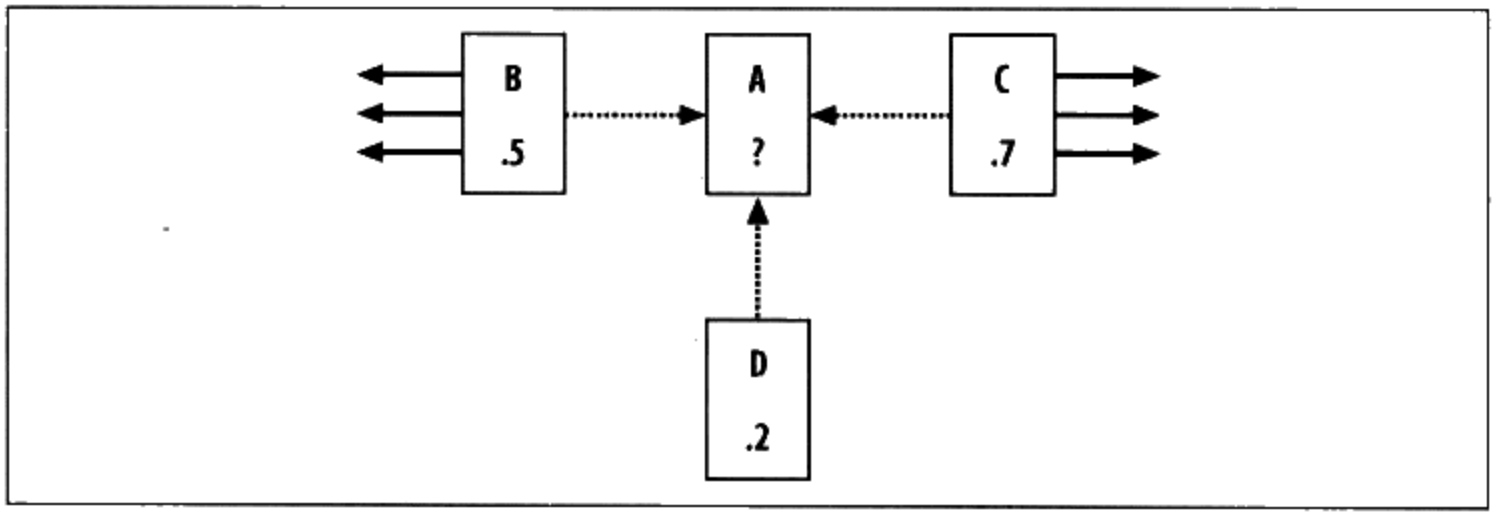


图 4-3：计算 A 的 PageRank 值

网页 B、C 和 D 均指向 A，它们的 PageRank 值已经计算得出。B 还指向另外三个网页，而 C 则指向其他四个网页，D 只指向 A。为了得到 A 的 PageRank 值，我们将指向 A 的每个网页的 PageRank（PR）值除以这些网页中的链接总数，然后乘以阻尼因子 0.85，再加上一个 0.15 的最小值。PR(A)的计算公式如下所示：

```
PR(A) = 0.15 + 0.85 * ( PR(B)/links(B) + PR(C)/links(C) + PR(D)/links(D) )
      = 0.15 + 0.85 * ( 0.5/4 + 0.7/5 + 0.2/1 )
      = 0.15 + 0.85 * ( 0.125 + 0.14 + 0.2)
      = 0.15 + 0.85 * 0.465
      = 0.54525
```

我们会发现，由于 D 只指向 A，而且能够贡献出它的全部分值，所以相比于 B 或 C，尽管 D 本身的 PageRank 值较低，但是实际上它对 A 的 PageRank 值贡献度更大。

是不是非常简单？不过，此处还有一点须要注意——在本例中，所有指向 A 的网页均已有了 PageRank 值。只有在知道了指向同一网页的所有其他网页的评价值后，我们才能计算出该网页的评价值。同样，对于指向这些网页的所有其他网页，如果不先计算它们的评价值，那么这些网页的 PageRank 值也是无法计算的。如何对一组还没有 PageRank 值的网页进行 PageRank 计算呢？

解决这一问题的方法，是为所有的 PageRank 都设置一个任意的初始值（示例代码中将使用 1.0，这对最终的实际值不会有任何影响），然后反复计算，迭代若干次。在每次迭代期间，每个网页的 PageRank 值将会越来越接近其真实值。迭代所需的次数要视网页的数量而定，不过对于目前正在处理的这一小组网页而言，20 次应该就足够了。

因为 PageRank 的计算是一项耗时的工作，而且其计算结果又不随查询的变化而变化，因此我们可以建立一个函数，预先为每个 URL 计算好 PageRank 值，并将计算结果存入数据表中。该函数会在每次执行期间重新计算所有的 PageRank 值。请将下列函数加入 crawler 类中：

```
def calculatepagerank(self,iterations=20):
  # 清除当前的 PageRank 表
  self.con.execute('drop table if exists pagerank')
  self.con.execute('create table pagerank(urlid primary key,score)')

  # 初始化每个 url，令其 PageRank 值为 1
self.con.execute('insert into pagerank select rowid, 1.0 from urllist')
self.dbcommit()

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

   # 循环遍历指向当前网页的所有其他网页
   for (linker,) in self.con.execute(
   'select distinct fromid from link where toid=%d' % urlid):
     # 得到链接源对应网页的 PageRank 值
     linkingpr=self.con.execute(
     'select score from pagerank where urlid=%d' % linker).fetchone()[0]

     # 根据链接源，求得总的链接数
     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()
```

该函数最初将每个网页的 PageRank 值都设置为 1.0。然后遍历每个 URL，并针对每个外部回指链接，得到其 PageRank 值与链接的总数。以粗体显示的代码行给出了应用于每个外部回指链接的计算公式。

运行该函数将会花费几分钟的时间，不过我们只有在更新索引时才须要做这项工作。

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

如果我们想知道示例数据集中哪个网页的 PageRank 值最高，可以直接查询数据库：

```
>> 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'
```

在本例中，“Main Page”的 PageRank 值是最高的，这一点并不奇怪，因为 Wikipedia 中的其他每一个网页都指向该页面。既然现在我们已经拥有了一张包含 PageRank 评价值的数据表，那么要利用这些数据，只须创建一个函数，将评价值从数据库中取出，然后对其做归一化处理。请将下列函数加入 searcher 类中：

```
  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
```

我们须要再次修改 weights 列表，将 PageRank 算法纳入其中。例如，如下所示：

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

最终的搜索结果会综合考虑网页内容与排名的影响。现在，针对“Functional programming”的搜索结果看起来就更加合理了：

```
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
```

与通过 Web 进行搜索相比，对于这样一组封闭而又严格受控的文档而言，其中所包含的无用网页和蓄意引起大家注意的网页可能很少，因此要想从中看出 PageRank 评价方法的价值是有些困难的。不过即便如此，我们依然可以从中看出，对于返回更高层次和更大众化的网页而言， PageRank 无疑是一种有效的度量方法。

利用链接文本

另一种对搜索结果进行排名的非常有效的方法，是根据指向某一网页的链接文本来决定网页的相关程度。大多数时候，相比于被链接的网页自身所提供的信息而言，我们从指向该网页的链接中所得到信息会更有价值。因为针对其所指向的网页，网站的开发者们会倾向于提供一些解释其内容的简短描述。

根据链接的文本对网页进行评价的方法接受一个额外的参数，该参数是一个单词 ID 的列表，这是在我们执行查询时构造的。我们可以将该方法加入 searcher 中：

```
  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
```

上述代码循环遍历 wordids 中的所有单词，并查找包含这些单词的链接。如果链接的目标地址与搜索结果中的某一条数据相匹配，则链接源对应的 PageRank 值将被加入到目标网页的最终评价值中。一个网页，如果拥有大量来自其他重要网页的链接指向，且这些网页又满足查询条件，则该网页将会得到一个很高的评价值。在查询结果中，有许多网页不具备包含有效文本的链接，其所得分值将为 0。

为了能够利用链接文本进行排名，只须将下列代码片段加入 weights 列表中的任何位置：

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

对于前述这些度量，并不存在一组标准的权重分配，能够适应所有的情况。即使是主流的搜索网站，也在时常改变他们对搜索结果的排名方法。我们所用到的度量方法，以及赋予它们的权重系数，很大程度上取决于我们正在试图构建的应用。

从点击行为中学习
Learning from Clicks

在线应用的一个最大优势就在于，它们会持续收到以用户行为为表现形式的反馈信息。对于搜索引擎而言，每一位用户可以通过只点击某条搜索结果，而不选择点击其他内容，向引擎及时提供有关于他对搜索结果喜好程度的信息。本节我们将介绍一种方法，它将记录用户点击查询结果的情况，并利用这一信息来改进搜索结果的排名。

为此，我们须要构造一个**人工神经网络**，向其提供：查询条件中的单词，返回给用户的搜索结果，以及用户的点击决策，然后再对其加以训练。一旦网络经过了许多不同查询的训练之后，我们就可以利用它来改进搜索结果的排序，以更好地反映用户在过去一段时间里的实际点击情况。

一个点击跟踪网络的设计
Design of a Click-Tracking Network

有许多种不同类型的神经网络，它们都以一组节点（**神经元**）构成，并且彼此相连。我们即将学习到的这种网络，被称为**多层感知机**（multilayer perceptron，MLP）网络。此类网络由多层神经元构造而成，其中第一层神经元接受输入——在本例中，即用户输入的单词。最后一层神经元则给予输出，在本例中，即一个涉及被返回的不同 URL 的权重列表。

神经网络可以有多个中间层，不过在本例中，我们只使用了一层。因为外界无法直接与其交互，所以该中间层被称为**隐藏层**，其职责是对输入进行组合。在本例中，对输入的组合结果就是一组单词，因此我们也可以将这一层看作是“**查询层**”。图 4-4 展示了网络的结构。所有位于输入层中的节点都与所有位于隐藏层中的节点相连接，而所有位于隐藏层中的节点也都与所有位于输出层中的节点相连接。

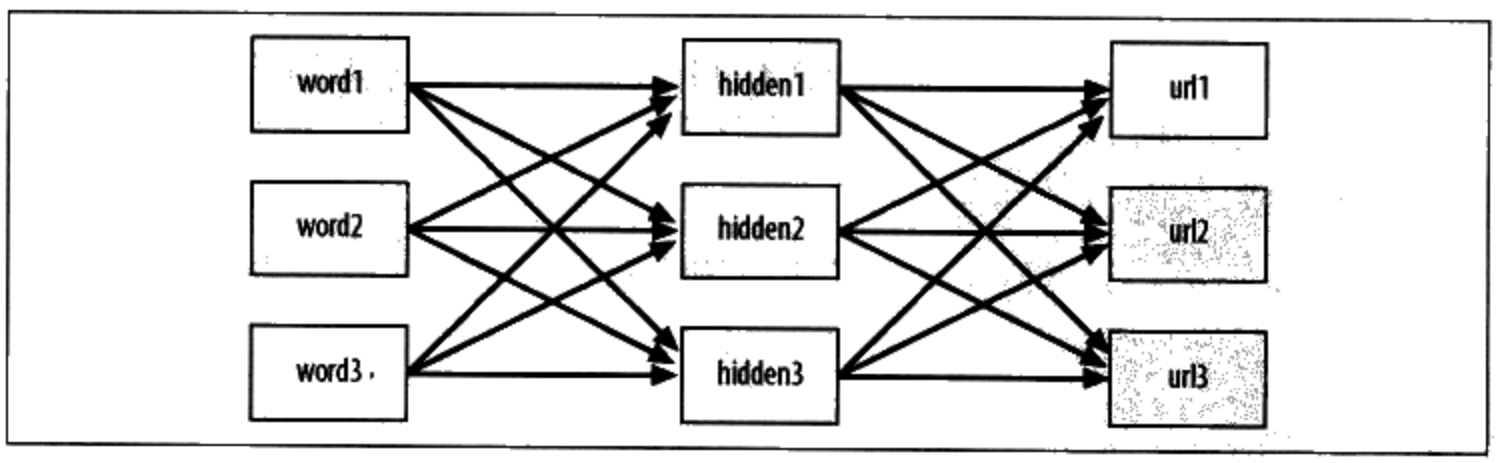


图 4-4：点击跟踪神经网络的设计

为了让神经网络得到最佳查询结果，我们将与查询条件中出现的单词相对应的输入节点设值为 1。这些节点的输出端开启后，会试图激活隐藏层。相应地，位于隐藏层中的节点如果得到一个足够强力的输入，就会触发其输出端，并试图激活位于输出层中的节点。

随后，位于输出层中的节点将处于不同程度的活跃状态，我们可以利用其活跃程度来判断一个 URL 与原查询中出现的单词在相关性上的紧密程度。图 4-5 给出了一个针对“world bank”的查询。图中的实线代表强连接，粗体文字则表示节点已经变得非常活跃。

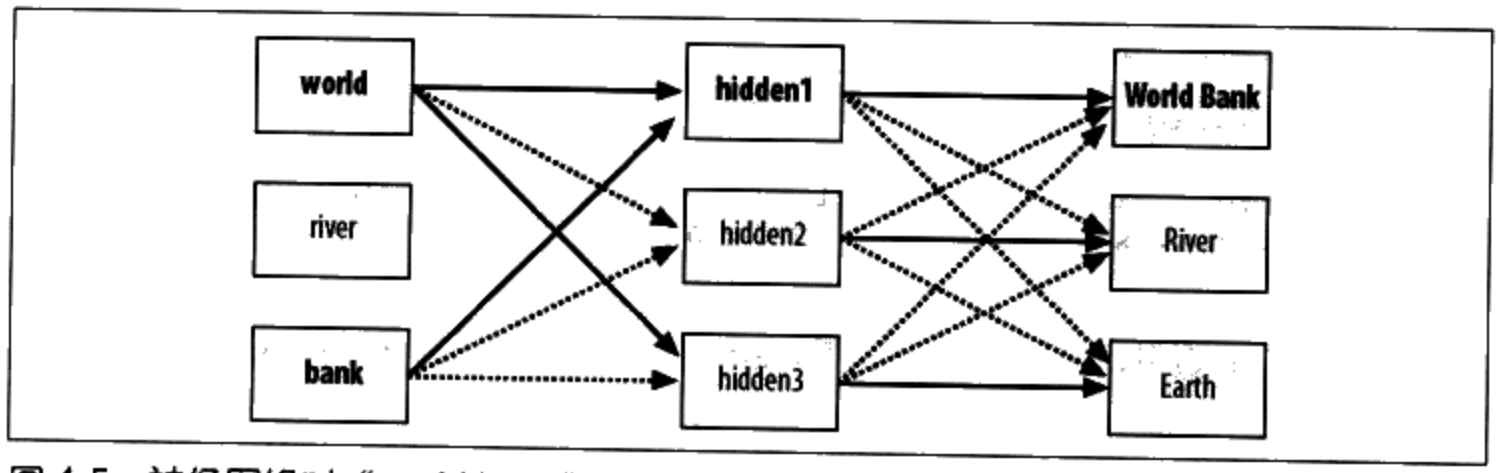


图 4-5：神经网络对“world bank”所作出的反应

当然，最终的结果还要取决于被逐渐纠正的连接强度。为此，只要有人执行搜索，并从结果中选择链接，我们就对该网络进行**训练**。在如图 4-5 所示的网络中，许多人已在搜索“world bank”之后，点击过有关 World Bank（世界银行）的相关结果，而这一点加强了单词与 URL 的关联。本节将向大家展示，如何利用一种被称为反向传播（backpropagation）的算法对网络进行训练。

也许你会好奇，为什么我们需要一种像神经网络这样的复杂技术，而不是简单地记录下查询条件以及每个搜索结果被点击的次数呢。接下来即将要构造的神经网络，其威力在于，它能根据与其他查询的相似度情况，对以前从未见过的查询结果做出合理的猜测。不仅如此，神经网络还广泛适用于许多其他的应用；对于我们的集体智慧“工具箱”（toolbox）而言，神经网络也是一个极大的补充。

设置数据库

由于神经网络须要在用户查询时经过不断的训练，因此我们须要将反映网络现状的信息存入数据库中。数据库中已经有一张涉及单词与 URL 的数据表，另外我们还需要一张代表隐藏层的数据表（我们称之为 hiddennode），以及两张反映网络节点连接状况的表（一张从单词层到隐藏层，另一张则连接隐藏层与输出层）。

请新建一个名为 *nn.py* 的文件，并在其中新建一个类，取名 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()
```

这些表目前还没有索引，如果效率问题非常突出，稍后我们可以为其添加索引。

为了访问数据库，我们须要创建两个方法。第一个方法名为 getstrength，用以判断当前连接的强度。由于新连接只在必要时才会被创建，因此该方法在连接不存在时将会返回一个默认值。对于从单词层到隐藏层的连接，其默认值将为-0.2。所以在默认情况下，附加的单词将会对处于隐藏层的节点在活跃程度上产生轻微的负面影响。对于从隐藏层到 URL 的连接而言，方法返回的默认值为 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]
```

此外，我们还需要一个 setstrength 方法，用以判断连接是否已存在，并利用新的强度值更新连接或创建连接。该函数将会为训练神经网络的代码所用：

```
    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))
```

大多数时候，当我们在构建神经、网络时，网络中的所有节点都是预先建好的。我们可以预先建立一个隐藏层中有上千节点，全部连接均已就绪的巨大网络，不过在本例中，只在需要时建立新的隐藏节点会更高效，也更简单。

每传入一组以前从未见过的单词组合，该函数就会在隐藏层中建立一个新的节点。随后，函数会为单词与隐藏节点之间，以及查询节点与由查询所返回的URL结果之间，建立起具有默认权重的连接。

```
def generatehiddennode(self,wordids,urls):
  if len(wordids)>3: return None
  # 检查我们是否已经为这组单词建好了一个节点
  createkey='_'.join(sorted([str(wi) for wi in wordids]))
  res=self.con.execute(
  "select rowid from hiddennode where create_key='%s'" % createkey).fetchone()

  # 如果没有，则建立之
  if res==None:
    cur=self.con.execute(
    "insert into hiddennode (create_key) values ('%s')" % createkey)
    hiddenid=cur.lastrowid
    # 设置默认权重
    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()
```

请在Python解释器中试着新建一个数据库，并生成一个带有样例单词和URL ID的隐藏节点：

```
>> 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)
...
```

上述执行过程在隐藏层中建立了一个新的节点，还建立了一个指向该新建节点的带默认值的连接。开始阶段，无论“world”与“bank”何时被一起输入，函数都会作出相应的响应，不过随着时间的推移，这些连接将会逐渐变弱。

前馈法

Feeding Forward

接下来，我们将编写相关的函数，接受一组单词作为输入，激活网络中的连接，并针对 URL 给出一组输出结果。

首先，我们来选择一个函数，用以指示每个节点对输入的响应程度。此处介绍的神经网络将使用**反双曲正切变换函数**（hyperbolic tangent，`tanh`），如图 4-6 所示。

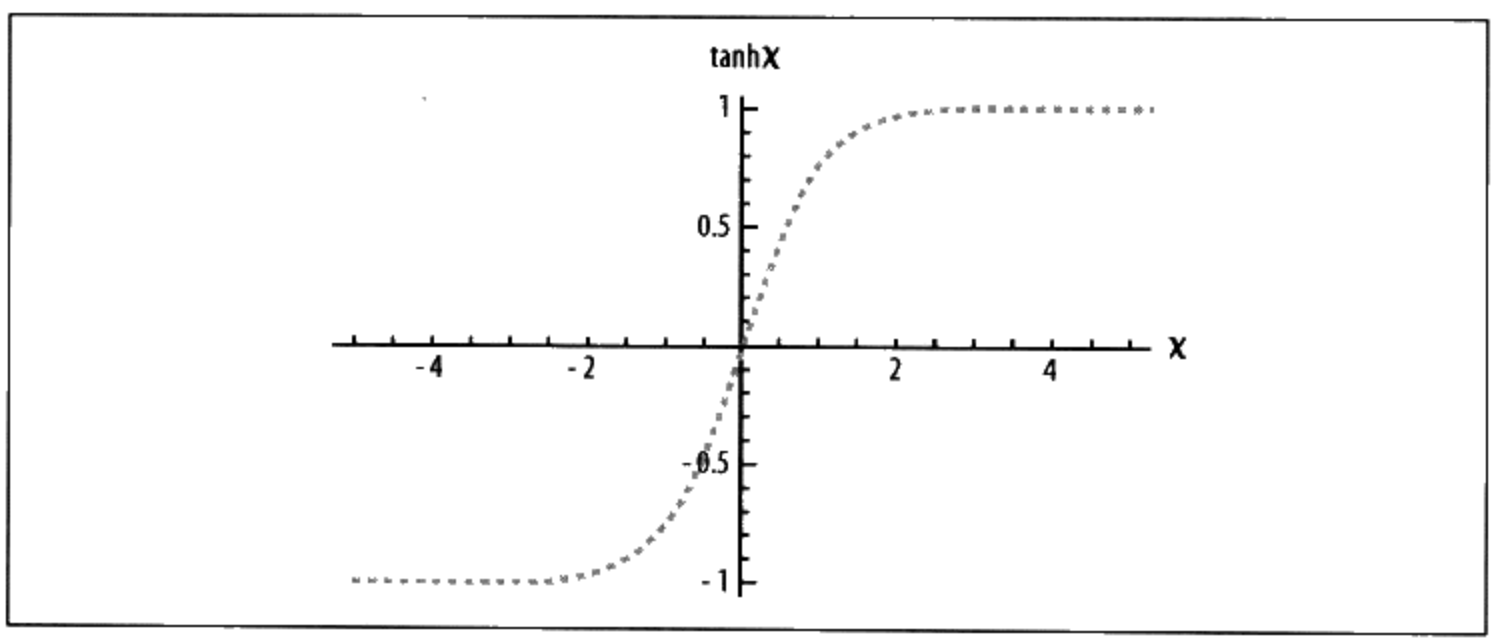


图 4-6：tanh 函数

其中，X 轴代表了针对节点的总输入。当输入接近 0 时，输出便开始快速爬高。当输入为 2 时，则输出几乎停留在 1 的位置不再变化。这是一类 **S 型函数**（sigmoid function），所有该类型的函数都会呈现这样的 S 形状。神经网络几乎总是利用 S 型函数来计算神经元的输出。

在开始执行前馈算法之前，`searchnet` 类的代码必须先从数据库中查询出节点与连接的信息，然后在内存中建立起与某项查询相关的那一部分网络。为此，首先第一步是编写一个函数，从隐藏层中找出与某项查询相关的所有节点——在本例中，这些节点必须关联于查询条件中的某个单词，或者关联于查询结果中的某个 URL。由于其他节点不会被用来判断结果或训练网络，所以我们没必要将它们包含在内：

```
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()
```

此外，我们还需要一个方法，利用数据库中保存的信息，建立起包括所有当前权重值在内的相应网络。该函数为 searchnet 类定义了多个实例变量，包括：单词列表、查询节点及 URL，每个节点的输出级别，以及每个节点间连接的权重值。此处的权重值是利用先前定义的函数，从数据库中取得的。

```
def setupnetwork(self,wordids,urlids):
    # 值列表
    self.wordids=wordids
    self.hiddenids=self.getallhiddenids(wordids,urlids)
    self.urlids=urlids

    # 节点输出
    self.ai = [1.0]*len(self.wordids)
    self.ah = [1.0]*len(self.hiddenids)
    self.ao = [1.0]*len(self.urlids)

    # 建立权重矩阵
    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]
```

最后我们来构造前馈算法。算法接受一列输入，将其推入网络，然后返回所有输出层节点的输出结果。在本例中，由于我们已经构造了一个只与查询条件中的单词相关的网络，因此所有来自输入层节点的输出结果都将总是为 1：

```
def feedforward(self):
    # 查询单词是仅有的输入
    for i in range(len(self.wordids)):
        self.ai[i] = 1.0

    # 隐藏层节点的活跃程度
    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)

    # 输出层节点的活跃程度
    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[:]
```

前馈算法的执行过程是循环遍历所有位于隐藏层中的节点，并将所有来自输入层的输出结果乘以连接强度之后累加起来。每个节点的输出等于所有输入之和经过 tanh 函数计算之后的结果，这一结果将被传给输出层。输出层的处理过程类似，也是将上一层的输出结果乘以强度值，然后应用 tanh 函数给出最终的输出结果。只要持续不断地将上一层的输出作为下一层的输入，我们可以很容易地对网络加以扩展，令其包含更多的层。

现在，我们可以编写一个简短的函数，建立神经网络，并调用 feedforward 函数针对一组单词与 URL 给出输出：

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

我们可以利用 Python 的执行环境，试验一下神经网络：

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

返回列表中的数字对应于输入 URL 的相关性。不必惊讶，因为尚未经过任何训练，所以此处的神经网络对于每个 URL 给出的结果都是一样的。

利用反向传播法进行训练

Training with Backpropagation

有趣之处就在于此。神经网络会接受输入并给出输出，但是由于我们还没有告诉它一个好的结果是什么样的，因而其返回的结果是毫无价值的。现在，我们将通过为神经网络提供某些人实际搜索的例子、相应的返回结果，以及用户决定点击的情况，对网络展开训练。

为此，我们需要一个算法，来修改介于两节点间连接的权重值，以便更好地反映人们告知网络的正确答案。由于无法假设每个用户都会点击一个适合所有人的答案，因此权重值须要逐步加以调整。我们将要使用的算法被称为**反向传播法**，因为该算法在调整权重值时是沿着网络反向行进的。

因为在对网络进行训练时，我们始终都知道每个输出层节点的期望输出，所以在这种情况下，如果用户点击了预期的结果，则它应该朝着 1 的方向推进，否则就朝 0 的方向推进。修改某一节点输出结果的唯一方法，是修改针对该节点的总输入。

为了确定我们应该如何改变总的输入，训练算法须要知道 tanh 函数在其当前输出级别上的斜率（slope）。在函数的中段，当输出为 0.0 时，斜率就会非常的“陡”，因此只改变一点点

输入便会获得很大的变化。如果输出结果越接近于-1 或 1，则改变输入对输出构成的影响就会变得越来越小。函数针对任何输出值的斜率均由下列函数指定，我们可以将其加入 *nn.py* 的开始处：

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

在执行反向传播算法之前，有必要运行一下 `feedforward` 函数。这样一来，每个节点的当前输出结果都将被存入实例变量中。而后，反向传播算法将执行如下步骤。

对于输出层中的每个节点：

1. 计算节点当前输出结果与期望结果之间的差距；
2. 利用 `dtanh` 函数确定节点的总输入须要如何改变；
3. 改变每个外部回指链接的强度值，其值与链接的当前强度及学习速率（learning rate）成一定比例。

对于每个隐藏层中的节点：

1. 将每个输出链接（output link）的强度值乘以其目标节点所需的改变量，再累加求和，从而改变节点的输出结果；
2. 使用 `dtanh` 函数确定节点的总输入所需的改变量；
3. 改变每个输入链接（input link）的强度值，其值与链接的当前强度及学习速率成一定比例。

由于全部计算都依赖于对当前权重的了解，而非对更新后权重的了解，该算法的实现逻辑实际上是预先对所有误差进行计算，然后再对权重加以调整。下面是算法的实现代码，可以将其加入 `searchnet` 类中：

```
def backPropagate(self, targets, N=0.5):
    # 计算输出层的误差
    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

    # 计算隐藏层的误差
    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
```

```
        # 更新输出权重
        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

        # 更新输入权重
        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
```

现在，我们所要做的全部工作就是编写一个简单的方法，建立神经网络，运行前馈算法和反向传播算法。该方法接受 wordids 列表、urlids 列表，以及一个 URL 作为参数：

```
def trainquery(self,wordids,urlids,selectedurl):
  # 如有必要，生成一个隐藏节点
  self.generatehiddennode(wordids,urlids)

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

为了将结果保存起来，我们还需要一个方法来更新数据库中的权重值，权重信息位于实例变量 wi 和 wo 中：

```
def updatedatabase(self):
  # 将值存入数据库中
  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()
```

现在，我们可以做一个简单的实验，选择一个以前曾经试过的查询，看看网络是如何响应训练过程的：

```
>> 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]
```

待神经网络了解到某位特定用户所做的选择以后，有关 World Bank 的 URL 输出结果就会有所增加，而其他 URL 的输出结果则会有所减少。用户做这样的选择越多，两类输出结果间的差距就会越大。

训练实验

至此我们已经看到了，利用某个样例结果对网络加以训练，可以增加针对该结果的输出。尽管这是有价值的，但是它并没有真正展示出神经网络的能力——即，对以前未曾见过的输入情况进行推理。请在你的 Python 会话中尝试输入如下代码：

```
>> 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]
```

尽管网络本身从未见过有关“bank”的查询，但是它给出了一个合理的猜测。不仅如此，即使在训练用的查询样例中，“bank”与“river”间的联系和它与 World Bank 间的联系相差无几，网络对于 World Bank URL 的评价依然还是会比 River URL 更高。神经网络不仅掌握了 URL 与查询的联系，还了解到一次特定查询中，哪些单词是重要的——这些信息是单纯从查询与 URL 的关联关系中无法获取到的。

与搜索引擎结合

Connecting to the Search Engine

searcher 类的 query 方法在生成与打印结果输出期间得到了一个有关 URL ID 与单词 ID 的列表。我们可以将下列代码加入 searchengine.py 中 query 方法的结尾处，令其将这一结果返回：

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

上述结果可以被直接传入 searchnet 的 trainquery 方法中。

搜集用户有关结果偏好信息的具体方法取决于我们对应用程序的设计。我们可以在网页中包含一个中间页面，以截获用户的点击行为，并在重定向到实际搜索页面之前调用 trainquery 方法，或者，甚至还可以让用户对搜索结果的相关性进行投票，以此来改进算法。

构建人工神经网络的最后一步，是在 searcher 类中新建一个方法，对搜索结果进行加权处理。该函数看起来与其他权重函数很类似。我们要做的第一件事情是在 *searchengine.py* 中引入神经网络的对应类：

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

然后将下列方法加入 searcher 类中：

```
def nnscore(self,rows,wordids):
  # 获得一个由唯一的URL ID构成的有序列表
  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)
```

同样，我们依然可以将其纳入到包含有多种不同权重的 weights 列表中进行试验。事实上，更值得推荐的做法是，等到网络经过大量不同样例的训练之后，再将其作为评价值的一部分纳入进来。

本章介绍了开发一个搜索引擎所需的许多知识，不过相比于实际情况这依然是非常有限的。本章的练习部分将涉及某些进阶议题。在这里，我们并未着力于性能问题——这可能要求我们对数以百万计的网页建立索引——不过前文构建的搜索引擎，对于 100 000 规模的网页而言，其性能应该是绰绰有余的，这对于新闻站点或公司内部网而言，已经足够了。

练习

Exercises

1. **分词** separatewords 方法目前将任何非字母和非数字字符都当作了分隔符，这意味着它无法为诸如“C++”、“$20”、“Ph.D.”或“617-555-1212”这样的词条建立正确的索引。更好的分词方法是什么呢？使用空白符作为分隔符是否可以？请编写一个更好的分词函数。

2. **布尔操作符** 许多搜索引擎都支持布尔查询，它允许用户构造诸如“python OR perl”这样的搜索条件。一个 OR 查询可以通过分别执行两次查询后再对结果进行组合的方式来实现，但是对于“python AND (program OR code)”又该如何处理呢？请修改查询方法，以支持某些基本的布尔操作。

3. **精确匹配** 搜索引擎通常都支持“精确匹配”查询：网页中与查询匹配的单词，其出现顺序必须与查询条件中的单词顺序相同，而且中间不允许夹杂任何其他的单词。请编写一个新的 getrows 函数，只返回精确匹配的结果。（提示：你可以使用 SQL 中的减法运算，得到两个单词间的位置之差。）

4. **长文/短文搜索** 有时，我们会利用网页的长度来判断其是否与某一类特定的搜索应用或用户相关。用户有可能会对查找有关疑难问题的长篇文章感兴趣，也有可能会对命令行工具的快速参考感兴趣。请编写一个权重函数，该函数将根据传入的参数，倾向于给出较长或较短的文档。

5. **单词频度偏好** “单词计数”的度量方法更偏向于较长的文档，因为篇幅较长的文档拥有更多的单词，而且因此也更有可能包含要搜索的单词。请编写一个新的度量方法，以文档中单词数量的百分比作为频度进行计算。

6. **外部回指链接搜索** 目前的代码可以根据外部回指链接的文本对返回结果进行排序，但是为此我们必须先利用基于内容的算法将这些网页找到。而有时，相关度很大的网页并不包含任何查询文本；相反，它们带有许多指向该网页的链接，这些链接对应的文本则满足搜索条件——指向图片的链接通常就属于此种情况。请修改搜索代码，令搜索结果中也包含那些外部回指链接中含有待搜索单词的情况。

7. **不同的训练选项** 对神经网络进行训练时，我们使用了一组 0 值代表所有用户没有点击的 URL，而用 1 来代表用户点击过的 URL。请修改训练函数，令其能够允许用户对结果给予从 1 到 5 的评价。

8. **附加层** 目前的神经网络只有一个隐藏层。请修改 `searchnet` 类，令其支持任意数量的隐藏层。关于层数，我们可以在初始化时加以指定。

第 5 章

优化

Optimization

本章我们将向大家介绍，如何使用一系列被称为**随机优化**（stochastic optimization）的技术来解决协作类问题。优化技术特别擅长于处理：受多种变量的影响，存在许多可能解的问题，以及结果因这些变量的组合而产生很大变化的问题。这些优化技术有着大量的应用：在物理学中，我们用它们来研究分子的运动；在生物学中，我们用它们来预测蛋白质的结构；在计算机科学中，我们用它们来测定算法的最坏可能运行时间。NASA（美国国家航空和宇宙航行局）甚至使用优化技术来设计具有正确操作特性的天线，而这些天线看起来似乎不像是人类设计者创造出来的。

优化算法是通过尝试许多不同题解并给这些题解打分以确定其质量的方式来找到一个问题的最优解的。优化算法的典型应用场景是，存在大量可能的题解以至于我们无法对它们进行一一尝试的情况。最简单但也是最低效的求解方法，莫过于去尝试随机猜测的上千个题解，并从中找出最佳解来。而更有效率的方法（将在本章中讨论），则是以一种对题解可能有改进的方式来对其进行智能化地修正。

本章的第一个例子是关于制定组团旅游计划的。曾经为团体哪怕是个人安排过旅游计划的人都知道，计划的制定要求有许多不同的输入，比如：每个人的航班时间表应该是什么，须要租用多少辆汽车，哪个飞机场是最通畅的。许多输出结果也必须考虑，比如，总的成本、候机的时间、起飞的时间。因为我们无法将这些输入用一个简单的公式映射到输出，所以要想找到最优解，就必须借助于优化算法。

本章的其他两个例子通过考查两个截然不同的问题，显示出了优化算法的灵活性。这两个例子分别是：如何基于人们的偏好来分配有限的资源？如何用最少的交叉线来可视化社会网络。在本章的末尾，我们还可以找到能够利用优化技术来解决的其他类型的问题。

组团旅游

为来自不同地方去往同一地点的人们（本例中是 Glass 一家）安排一次旅游是一件极富挑战性的事情，它引出了一个非常有趣的优化问题。首先，我们创建一个文件，命名为 *optimization.py*，然后加入下面的代码：

```
import time
import random
import math

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

# New York 的 LaGuardia 机场
destination='LGA'
```

家庭成员们来自全国各地，并且他们希望在纽约会面。他们将在同一天到达，并在同一天离开，而且他们想搭乘相同的交通工具往返飞机场。每天有许多航班从任何一位家庭成员的所在地飞往纽约，飞机的起飞时间都是不同的。这些航班在价格和续航时间上也都不尽相同。

我们可以从这个网址 *http://kiwitobes.com/optimize/schedule.txt* 下载有关航班数据的样本文件。

该文件包含一组以逗号分隔的、具有如下数据格式的航班数据：起点、终点、起飞时间、到达时间、价格。

```
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
```

我们将这些数据载入到一个字典中，并以起止点为键，以可能的航班详情明细为值。请将如下这段加载数据的代码添加到 *optimization.py* 文件中：

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

  # 将航班详情添加到航班列表中
  flights[(origin,dest)].append((depart,arrive,int(price)))
```

我们还在此处定义了一个非常有用的工具函数 getminutes，该函数用于计算某个给定时间在一天中的分钟数。这使飞行时间和候机时间的计算变得非常容易。现在我们将这个函数添加到 *optimization.py* 文件中：

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

接下来的问题是，家庭中的每个成员应该乘坐哪个航班呢？当然，使总的票价降下来是一个目标，但是最优解将要考虑和尝试最小化的还有许多其他可能的因素，比如：总的候机时间或总的飞行时间。稍候我们将详细讨论这些因素。

描述题解

当处理类似这样的问题时，我们有必要明确潜在的题解将如何表达。后面我们将看到的优化函数是非常通用的，它能应用于许多不同类型的问题上，因此，选择一个题解的简单表示方法而不局限于组团旅游问题是非常重要的。有一种非常通用的表达方式，就是数字序列。在本例中，一个数字可以代表某人选择乘坐的航班——0 是这天中的第一次航班，1 是第二次，依次类推。因为每个人都须要往返两个航班，所以列表的长度是人数的两倍。

例如：

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

上述列表代表了一种题解：Seymour 搭乘当天的第 2 次航班从 Boston 飞往 New York，并搭乘当天的第 5 次航班返回到 Boston。Franny 搭乘第 4 次航班从 Dallas 飞往 New York，并搭乘第 3 次航班返回。

因为要从一列数字中解释清楚题解是很困难的，所以我们需要一个程序，能将人们决定搭乘的所有航班打印成表格。请将下列函数加入 *optimization.py* 中：

```
def printschedule(r):
  for d in range(len(r)/2):
    name=people[d][0]
    origin=people[d][1]
    out=flights[(origin,destination)][r[2*d]]
    ret=flights[(destination,origin)][r[2*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])
```

上述函数将打印出一行，包括：人名和起点、出发时间、到达时间，以及往返航班的票价。请在你的 Python 会话中尝试执行该函数。

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

即使不考虑价格，上述安排仍然是有问题的。尤其是，因为家庭成员都一起往返于机场，因此以 Les 的返程航班为例，即使有些人直到下午 4 点才会飞离机场，每个人也都必须在早晨 6 点到达机场。为了确定最佳组合，程序需要一种方法来为不同日程安排的各种属性进行评估，从而决定哪一个方案是最好的。

成本函数

The Cost Function

成本函数是用优化算法解决问题的关键，它通常是最难确定的。任何优化算法的目标，就是要寻找一组能够使成本函数的返回结果达到最小化的输入（在本例中，输入即为航班信息），因此成本函数须要返回一个值用以表示方案的好坏。对于**好坏的程度**并没有特定的衡量尺度，唯一的要求就是函数返回的值越大，表示该方案越差。

通常，根据众多变量来鉴别方案的好坏是比较困难的。我们来考查一些在组团旅游的例子中能够被度量的变量。

价格

所有航班的总票价，或者有可能是考虑财务因素之后的加权平均。

旅行时间

每个人在飞机上花费的总时间。

等待时间

在机场等待其他成员到达的时间。

出发时间

早晨太早起飞的航班也许会产生额外的成本，因为这要求旅行者减少睡眠的时间。

汽车租用时间

如果集体租用一辆汽车，那么他们必须在一天内早于起租时刻之前将车辆归还，否则将多付一天的租金。

为了让旅途体验能够更加愉快一些，对某一特定的时间安排从更多方面进行考查并非难事。每当我们为一个复杂问题寻求最佳方案时，都须要明确什么是最重要的因素。尽管这可能是有难度的，但是这样做的最大好处在于，一旦找到这些最重要的因素，只须做少量的修改，我们就可以运用本章中介绍的优化算法解决几乎任何一个问题。

选择好对成本产生影响的变量之后，我们就须要找到办法将它们组合在一起形成一个值。例如在本例中，我们就有必要明确，在飞机上的时间或在机场等待时所消耗的时间价值是多少。或许我们可以假定，在飞行旅行中节省的每一分钟价值 1 美元（这相当于，再加 90 美元选择乘坐直达航班，就可以节省一个半小时的时间），而在机场等待中所节省的每一分钟则价值 0.50 美元。如果每个人回到机场的时刻都晚于最初租用汽车的时刻，那么我们还可以将多付一天的租车费用也算在内。

此处定义的 `schedulecost` 函数有很多种可能的结果。该函数考查了总的旅行成本以及不同家庭成员在机场总的等待时间。如果汽车是在租用时间点之后归还的，则还会追加 50 美元的罚款。请将该函数添加到 *optimization.py* 文件中，并且你还可以随意追加额外的成本，或者调整金额和时间的相关重要性（译注 1）：

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

  for d in range(len(sol)/2):
    # 得到往程航班和返程航班
    origin=people[d][1]
    outbound=flights[(origin,destination)][int(sol[2*d])]
    returnf=flights[(destination,origin)][int(sol[2*d+1])]
    # 总价格等于所有往程航班和返程航班价格之和
    totalprice+=outbound[2]
    totalprice+=returnf[2]

    # 记录最晚到达时间和最早离开时间
    if latestarrival<getminutes(outbound[1]): latestarrival=getminutes(outbound[1])
    if earliestdep>getminutes(returnf[0]): earliestdep=getminutes(returnf[0])

  # 每个人必须在机场等待直到最后一个人到达为止
  # 他们也必须在相同时间到达，并等候他们的返程航班
  totalwait=0
  for d in range(len(sol)/2):
    origin=people[d][1]
    outbound=flights[(origin,destination)][int(sol[2*d])]
    returnf=flights[(destination,origin)][int(sol[2*d+1])]
    totalwait+=latestarrival-getminutes(outbound[1])
    totalwait+=getminutes(returnf[0])-earliestdep
```

译注 1：schedulecost 中对 latestarrival>earliestdep 的判断似乎有误，应改为 latestarrival<earliestdep。根据上下文，租用汽车的时间不应小于 latestarrival，而归还汽车的时间则不应大于 earliestdep。在 latestarrival>earliestdep 的情况下，一定会有租车时间>还车时间，那么此时不应罚款才对。只有当 latestarrival<earliestdep，且假定租车时间为 latestarrival，而还车时间为 earliestdep 时，我们才可以判定须要罚钱。

```
    # 这个题解要求多付一天的汽车租用费用吗？如果是，则费用为 50 美元
    if latestarrival>earliestdep: totalprice+=50

    return totalprice+totalwait
```

上述函数中的逻辑虽然非常简单，但是它却阐明了关键的因素。我们还可以采用若干方法对其功能进行增强——目前，总的等待时间的计算是假定：当家庭成员中的最后一人到达时所有人才会一起离开机场，并且所有人会一起赶往机场搭乘最早一班飞机离开。我们可以对此进行修改，允许任何要面临两小时或更长时间等待的人可以独自租车离开，我们还可以对其中的价格和等待时间做出相应的调整。

我们可以在 Python 会话中尝试运行一下该函数：

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

成本函数既已建立，那么应该很清楚，我们的目标就是要通过选择正确的数字序列来最小化该成本。理论上，我们可以尝试每种可能的组合，但在这个例子中，一共有 12 个航班，每种航班又有 10 种可能，因此会得到 10^{12}（大约 1000 亿）种组合。测试每种组合能确保我们得到最优的答案，但是在大多数计算机上，这会花费非常长的时间。

随机搜索

Random Searching

随机搜索不是一种非常好的优化算法，但是它却使我们很容易领会所有算法的真正意图，并且它也是我们评估其他算法优劣的基线（baseline）。

这个函数接受两个参数。Domain 是一个由二元组（2-tuples）构成的列表，它指定了每个变量的最小最大值。题解的长度与此列表的长度相同。在当前的例子中，每个人都有 10 个往程航班和 10 个返程航班，因此列表中的 domain 是(0,9)，每个人重复两次。

第二个参数，`costf`，是成本函数，本例中即是 `schedulecost`。将其作为参数传入是为了让这个函数能够为其他优化问题所重用。此函数会随机产生 1 000 次猜测，并对每一次猜测调用 `costf`。它会跟踪最佳猜测（即具有最低成本的题解）并将结果返回。请将该函数加入 *optimization.py*：

```
def randomoptimize(domain,costf):
  best=999999999
  bestr=None
  for i in range(1000):
    # 创建一个随机解
    r=[random.randint(domain[i][0],domain[i][1])
       for i in range(len(domain))]
```

```
    # 得到成本
    cost=costf(r)

    # 与到目前为止的最优解进行比较
    if cost<best:
      best=cost
      bestr=r
  return r
```

当然，1 000 次猜测在全部可能性中仅占非常小的一部分。然而在本例中，我们可能会从中找到不少表现尚可的题解（即便不是最优的），因此在尝试一千次之后，该函数有可能会找到一个看似不算很差的解。请在你的 Python 会话中尝试执行一下该函数：

```
>>> reload(optimization)
>>> domain=[(0,9)]*(len(optimization.people)*2)
>>> s=optimization.randomoptimize(domain,optimization.schedulecost)
>>> optimization.schedulecost(s)
3328
>>> optimization.printschedule(s)
   Seymour      BOS   12:34-15:02 $109  12:08-14:05 $142
    Franny      DAL   12:19-15:25 $342   9:49-13:51 $229
     Zooey      CAK    9:15-12:14 $247  15:50-18:45 $243
      Walt      MIA   15:34-18:11 $326  14:08-16:09 $232
     Buddy      ORD   14:22-16:32 $126  15:04-17:23 $189
       Les      OMA   15:03-16:42 $135   6:19- 8:13 $239
```

由于是随机的原因，你所得到的结果将会与此处给出的结果有所不同。上述结果不算很好，因为 Zooey 须要在机场等候 6 个小时直至 Walt 到达，但是这样的结果显然不算是最差的。多执行几次该函数，看看成本值是否变化很大，或者把循环次数增加到 10 000，看看是否能按此方向找到更优的结果。

爬山法

Hill Climbing

随机尝试各种题解是非常低效的，因为这种方法没有充分利用已经发现的优解。在我们的例子中，拥有较低总成本的时间安排很可能接近于其他低成本安排。因为随机优化是到处跳跃的（jumps around），所以它不会自动去寻找与已经被发现的优解相接近的题解。

随机搜索的一个替代方法叫做**爬山法**。爬山法以一个随机解开始，然后在其临近的解集中寻找更好的题解（具有更低的成本）。这类似于从斜坡上向下走，如图 5-1 所示。

想象一下你就是图中所示的那个人，不经意间陷入了这块区域中，并且想走到最低点去寻找水源。为此我们可以选择任何一个方向，然后朝着最为险峻的斜坡向下走去。你可以朝

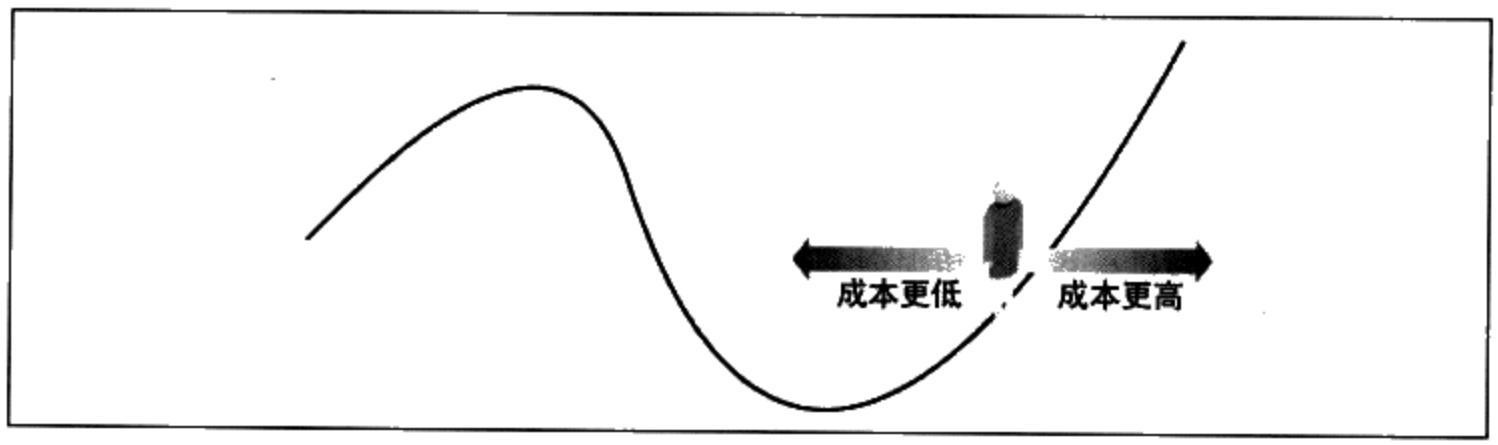


图 5-1：用爬山法寻找最低成本

着最为险峻的斜坡方向一直走下去，直至到达地势平坦或坡度开始向上倾斜的区域。

我们可以应用这种爬山法为 Glass 一家找到最好的旅行安排方案。先从一个随机的时间安排开始，然后再找到所有与之相邻的安排。在本例中，亦即找到所有相对于最初的随机安排，能够让某个人乘坐的航班稍早或者稍晚一些的安排。我们对每一个相邻的时间安排都进行成本计算，具有最低成本的安排将成为新的题解。重复这一过程直到没有相邻安排能够改善成本为止。

为了实现这一功能，请将 `hillclimb` 添加到 *optimization.py* 文件中：

```
def hillclimb(domain,costf):
  # 创建一个随机解
  sol=[random.randint(domain[i][0],domain[i][1])
     for i in range(len(domain))]
# 主循环
while 1:

  # 创建相邻解的列表
  neighbors=[]
  for j in range(len(domain)):

    # 在每个方向上相对于原值偏离一点
    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:])

  # 在相邻解中寻找最优解
  current=costf(sol)
  best=current
  for j in range(len(neighbors)):
    cost=costf(neighbors[j])
    if cost<best:
      best=cost
      sol=neighbors[j]
```

```
    # 如果没有更好的解，则退出循环
    if best==current:
      break

  return sol
```

该函数在给定域内随机生成一个数字列表，用以构造初始的题解。它通过循环遍历列表中的每一个元素，找到当前解的所有相邻题解，然后创建出两个新的列表：一个列表中的元素加 1，另一个列表中的元素减 1。相邻解中最优的一个将成为新的当前题解。

请在你的 Python 会话中尝试执行该函数，看看与随机搜索方法相比，其效果如何：

```
>>> 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
```

该函数的运行速度很快，并且找到的解通常要比随机搜索方法找到的更好。然而，爬山法有一个较大的缺陷。如图 5-2 所示：

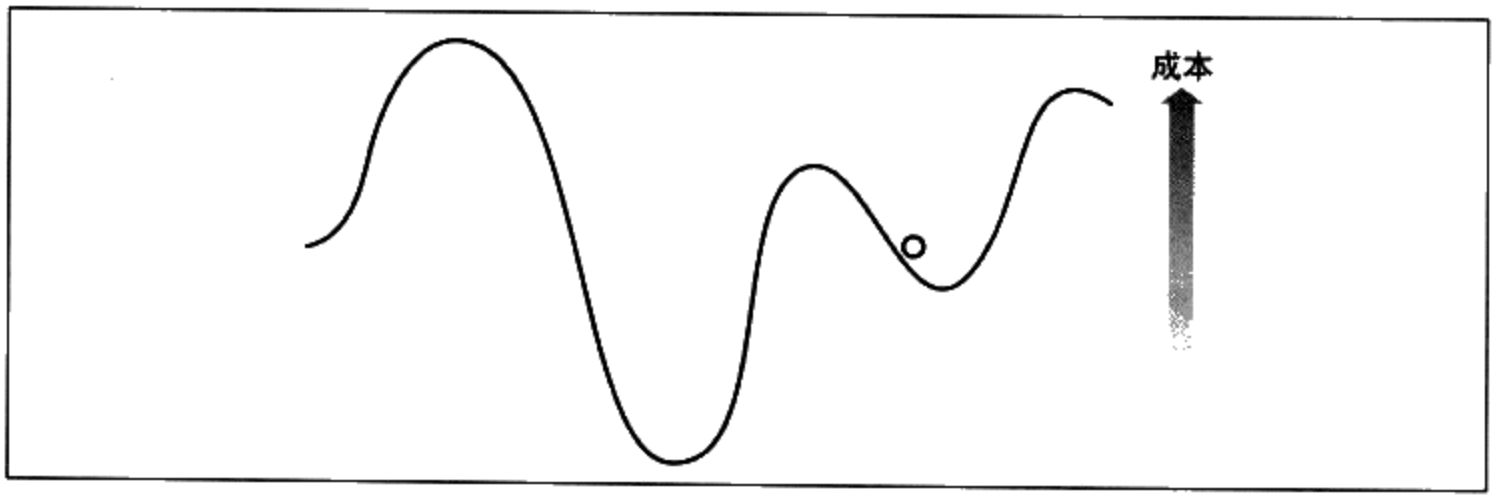


图 5-2：陷入局部范围内的最小值

从上图中我们可以很明显地看出，简单地从斜坡滑下不一定会产生全局最优解。最后的解会是一个**局部范围内的最小值**，它比邻近解的表现都好，但却不是全局最优的。全局最优解就是**全局最小值**，它是优化算法最终应该找到的那个解。解决这一难题的一种方法被称为**随机重复爬山法**（random-restart hill climbing），即让爬山法以多个随机生成的初始解为起点运行若干次，借此希望其中有一个解能够逼近全局的最小值。在后面两节中，我们将向大家展示避免陷入局部最小值的其他方法，它们分别是：模拟退火算法和遗传算法。

模拟退火算法

Simulated Annealing

模拟退火算法是受物理学领域启发而来的一种优化算法。退火是指将合金加热后再慢慢冷却的过程。大量的原子因为受到激发而向周围跳跃，然后又逐渐稳定到一个低能阶的状态，所以这些原子能够找到一个低能阶的配置（configuration）。

退火算法以一个问题的随机解开始。它用一个变量来表示温度，这一温度开始时非常高，尔后逐渐变低。每一次迭代期间，算法会随机选中题解中的某个数字，然后朝某个方向变化。在我们的例子中，Seymour 的返程航班也许会从当天的第二趟移到第三趟。其成本会在这一变化前后分别计算出来，并进行比较。

算法最为关键的部分在于：如果新的成本值更低，则新的题解就会成为当前题解，这和爬山法非常相似。不过，如果成本值更高的话，则新的题解仍将可能成为当前题解。这是避免图 5-2 中局部最小值问题的一种尝试。

某些情况下，在我们能够得到一个更优的解之前转向一个更差的解是很有必要的。模拟退火算法之所以管用，不仅因为它总是会接受一个更优的解，而且还因为它在退火过程的开始阶段会接受表现较差的解。随着退火过程的不断进行，算法越来越不可能接受较差的解，直到最后，它将只会接受更优的解。更高成本的题解，其被接受的概率由下列公式给出：

$$p=e^{(-(highcost-lowcost)/temperature)}$$

因为温度（接受较差解的意愿）开始非常之高，指数将总是接近于 0，所以概率几乎为 1。随着温度的递减，高成本值和低成本值之间的差异越来越重要——差异越大，概率越低，因此该算法只倾向于稍差的解而不会是非常差的解。

请在 *optimization.py* 文件中创建一个名为 annealingoptimize 的新函数，实现上述算法：

```
def annealingoptimize(domain,costf,T=10000.0,cool=0.95,step=1):
  # 随机初始化值
  vec=[float(random.randint(domain[i][0],domain[i][1]))
       for i in range(len(domain))]

  while T>0.1:
    # 选择一个索引值
    i=random.randint(0,len(domain)-1)

    # 选择一个改变索引值的方向
    dir=random.randint(-step,step)
```

```
    # 创建一个代表题解的新列表，改变其中一个值
    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]

    # 计算当前成本和新的成本
    ea=costf(vec)
    eb=costf(vecb)

    # 它是更好的解吗？或者是趋向最优解的可能的临界解吗？
    if (eb<ea or random.random()<pow(math.e,-(eb-ea)/T)):
      vec=vecb

    # 降低温度
    T=T*cool
  return vec
```

为了退火，函数首先创建一个具有合适长度的随机解，其中的所有值都位于定义域参数指定的范围内。温度和冷却率是两个可选的参数。函数会在每次迭代时，将 i 设为题解的一个随机索引，并将 dir 设为介于-step 与 step 之间的某个随机数。该算法会计算当前函数的成本，以及以 dir 为增量对 i 处的值进行修改时的函数成本。

粗体显示的代码行是关于概率计算的，概率的值随着 T 的降低而降低。如果介于 0 和 1 之间的某个随机浮点数小于该值，或者如果新的题解更优，那么该函数将接受新的题解。函数中的循环过程直到温度几乎等于 0 为止，每次循环会将温度值与冷却率相乘。

现在，我们可以在自己的 Python 会话中尝试使用模拟退火算法来进行优化了：

```
>>> reload(optimization)
>>> s=optimization.annealingoptimize(domain,optimization.schedulecost)
>>> optimization.schedulecost(s)
2278
>>> 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 15:07-17:21 $129
```

这种优化算法在保持成本持续下降的同时，在减少总的执行时间方面也表现不俗。很显然，你得到的结果可能会有所不同，甚至有可能碰巧会得到一个较差的结果。对于任何一个给定的问题，不妨使用不同的参数（初始温度和冷却率）做一做试验。你还可以改变代表随机推进的 step 值的大小。

遗传算法

Genetic Algorithms

另一类优化技术也是受自然科学的启发，被称为**遗传算法**。这类算法的运行过程是先随机生成一组解，我们称之为**种群**（population）。在优化过程中的每一步，算法会计算整个种群的成本函数，从而得到一个有关题解的有序列表。示例如表 5-1 所示。

表 5-1：题解及成本的有序列表

题解	成本
[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

在对题解进行排序之后，一个新的种群——我们称之为下一代——被创建出来了。首先，我们将当前种群中位于最顶端的题解加入其所在的新种群中。我们称这一过程为**精英选拔法**（elitism）。新种群中的余下部分是由修改最优解后形成的全新解所组成的。

有两种修改题解的方法。其中较为简单的一种被称为**变异**（mutation），其通常的做法是对一个既有解进行微小的、简单的、随机的改变。在本例中，要完成变异只须从题解中选择一个数字，然后对其进行递增或递减即可。图 5-3 中给出了两个示例。

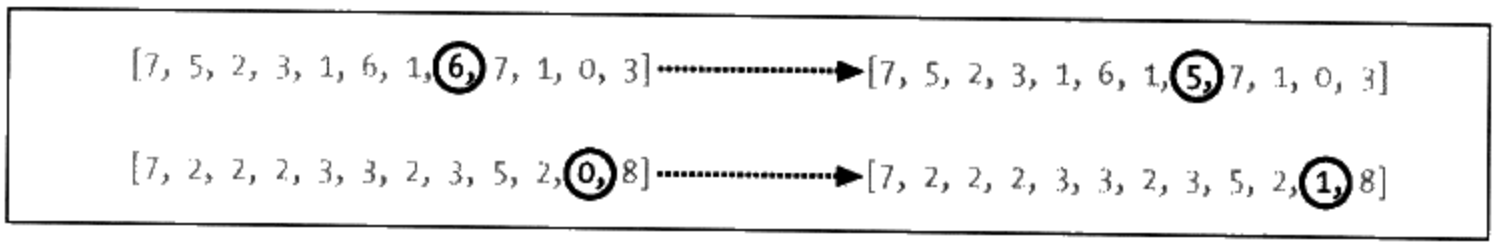


图 5-3：针对题解的变异示例

修改题解的另一种方法称之为**交叉**（crossover）或**配对**（breeding）。这种方法是选取最优解中的两个解，然后将它们按某种方式进行结合。在本例中，要实现交叉的一种简单方式是，从一个解中随机取出一个数字作为新题解中的某个元素，而剩余元素则来自另一个题解，如图 5-4 所示。

一个新的种群是通过对最优解进行随机的变异和配对处理构造出来的，它的大小通常与旧的种群相同。尔后，这一过程会一直重复进行——新的种群经过排序，又一个种群被构造出来。达到指定的迭代次数，或者连续经过数代后题解都没有得到改善，整个过程就结束了。

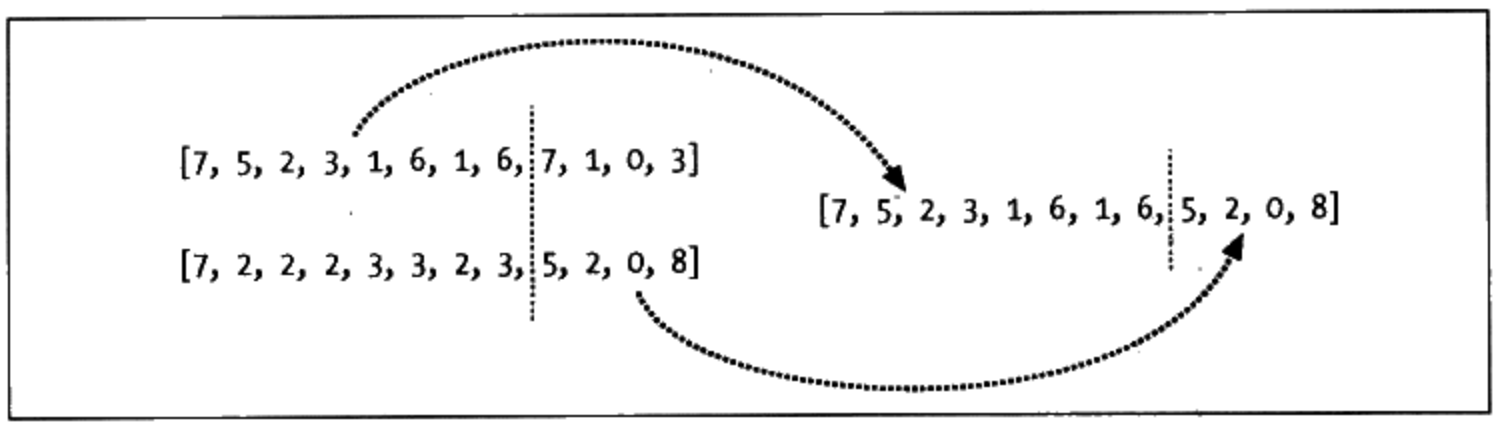


图 5-4：交叉示例

请将 geneticoptimize 添加到 *optimization.py* 文件中：

```
def geneticoptimize(domain,costf,popsize=50,step=1,
                    mutprob=0.2,elite=0.2,maxiter=100):
  # 变异操作
  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:]

  # 交叉操作
  def crossover(r1,r2):
    i=random.randint(1,len(domain)-2)
    return r1[0:i]+r2[i:]

  # 构造初始种群
  pop=[]
  for i in range(popsize):
    vec=[random.randint(domain[i][0],domain[i][1])
        for i in range(len(domain))]
    pop.append(vec)

  # 每一代中有多少胜出者?
  topelite=int(elite*popsize)

  # 主循环
  for i in range(maxiter):
    scores=[(costf(v),v) for v in pop]
    scores.sort()
    ranked=[v for (s,v) in scores]

    # 从纯粹的胜出者开始
    pop=ranked[0:topelite]

    # 添加变异和配对后的胜出者
    while len(pop)<popsize:
      if random.random()<mutprob:
```

```
            # 变异
            c=random.randint(0,topelite)
            pop.append(mutate(ranked[c]))
          else:

            # 交叉
            c1=random.randint(0,topelite)
            c2=random.randint(0,topelite)
            pop.append(crossover(ranked[c1],ranked[c2]))

        # 打印当前最优值
        print scores[0][0]

      return scores[0][1]
```

上述函数引入了几个可选的参数：

popsize

种群大小

mutprob

种群的新成员是由变异而非交叉得来的概率。

elite

种群中被认为是优解且被允许传入下一代的部分。

maxiter

须运行多少代。

请尝试在你的 Python 会话中，运用遗传算法优化一下旅行计划：

```
>>> 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
```

在第 11 章中，我们还将看到遗传算法的一个拓展，称为**遗传编程**（genetic programming），在那里，我们采用了类似的思路来完整构造新的程序。

提示：计算机科学家 John Holland 因其在 1975 年所撰写的《自然与人造系统的适应性（Adaptation in Natural and Artificial Systems）》一书（密歇根大学出版社），而被公认为是遗传算法之父。但是相关的研究工作还可以追溯到 20 世纪 50 年代，那时的生物学家们已经开始尝试在计算机上进行进化建模了。从那以后，遗传算法和其他优化方法已经被广泛应用于许多不同的问题。

- 寻找能够给出最佳音效的音乐厅外形。
- 为超音速飞机设计最佳的机翼。
- 给出最佳的化学制品库以供研发前沿药物参考之用。
- 自动化设计语音识别芯片。

我们可以将这些问题的潜在解转化成数字列表。这样我们就可以很容易地应用遗传算法或模拟退火算法了。

一种优化方法是否管用很大程度上取决于问题本身。模拟退火算法、遗传算法，以及大多数其他优化方法都有赖于这样一个事实：对于大多数问题而言，最优解应该接近于其他的优解。来看一个优化可能不起作用的例子，如图 5-5 所示。

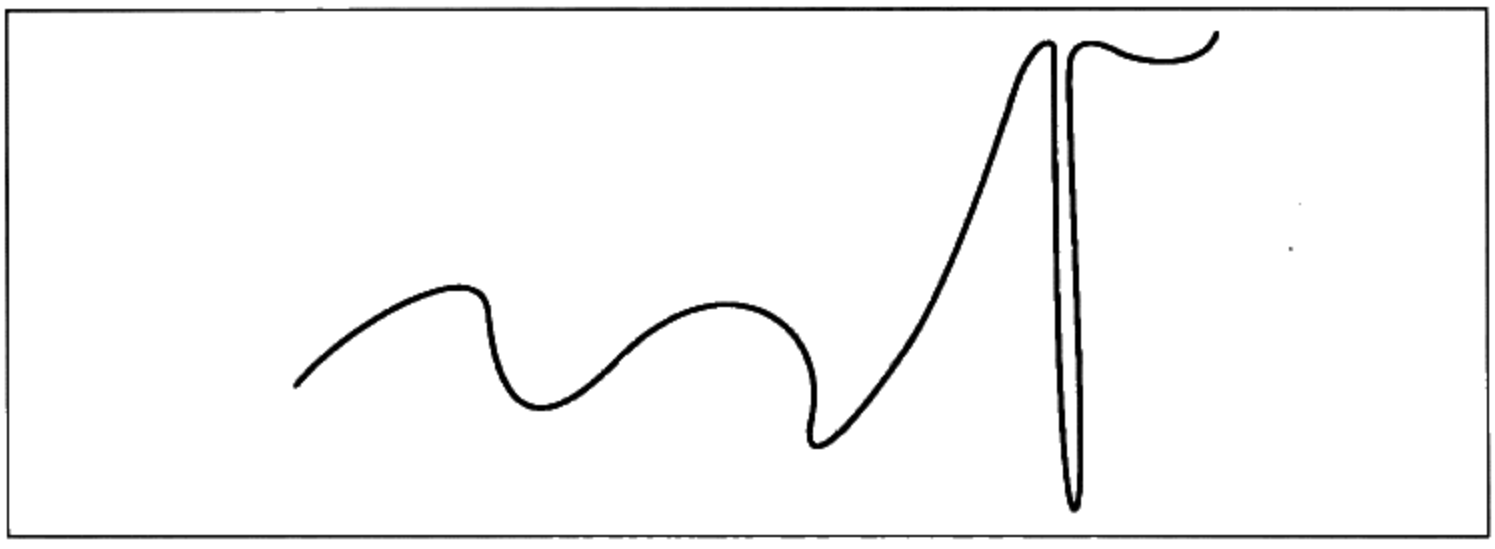


图 5-5：很难优化的问题

在图的最右边，成本的最低点实际上处在一个非常陡峭的区域。接近它的任何解都有可能被排除在外，因为这些解的成本都很高，所以我们永远都找不到通往全局最小值的途径。大多数算法都会陷入图中左边某个局部最小化的区域里。

优化算法对航班安排的例子之所以管用是因为，我们将一个人从当天的第二次航班转移到第三次航班，要比将他转移到第八次航班更有可能降低总成本。如果航班处于无序状态，那么优化方法的效果是不会比随机搜索好多少的——事实上，在这种情况下，没有任何一种优化方法一定会比随机搜索更加有效。

真实的航班搜索
Real Flight Searches

既然对于示例数据而言一切都没有问题了，那么是我们尝试利用真实的航班数据对前述优化算法的有效性进行考查的时候了。为此我们可以从 Kayak 网站下载数据，Kayak 提供了一套用于航班搜索的 API。真实的航班数据与你正在使用的示例数据的主要区别在于，在真实数据中，各大城市之间每天的航班远超过 9 次。

Kayak API

如图 5-6 所示，Kayak 是一个很受欢迎的旅游类**垂直搜索引擎**。尽管有许多在线的旅游站点，但是 Kayak 对本例而言还是很有价值的，因为它有一套非常不错的 XML API，我们可以利用这套 API 在 Python 程序中执行真实的旅游搜索。为了使用这套 API，我们必须去 *http://www.kayak.com/labs/api/search* 注册一个开发者密钥（developer key）。

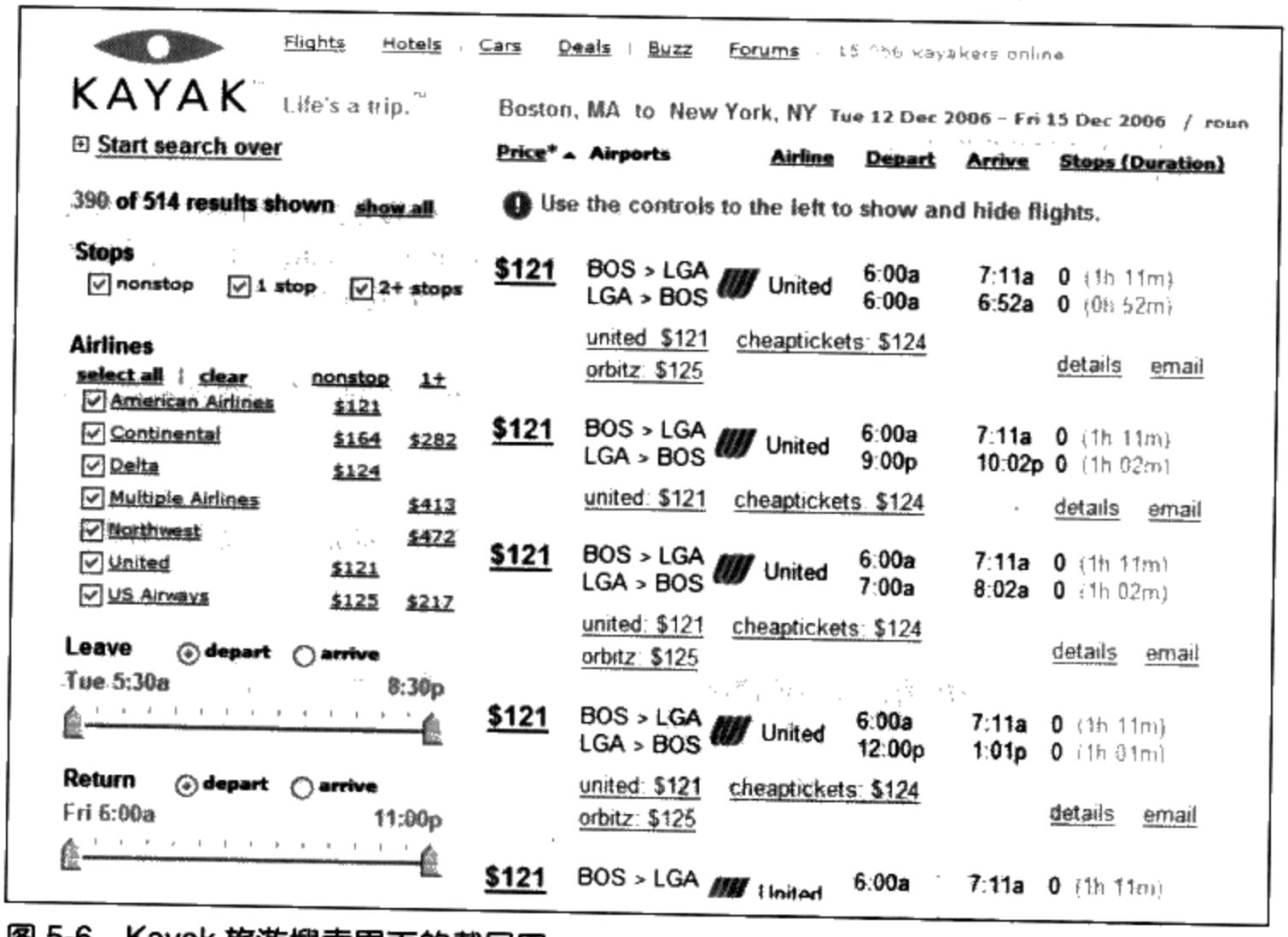


图 5-6：Kayak 旅游搜索界面的截屏图

开发者密钥是一个由数字与字母组合而成的长串，利用它我们可以在 Kayak 中进行航班搜索（我们也可以用它来搜索旅馆，不过这不是本章要讨论的话题）。在本书撰写期间，还没

有像 del.icio.us 那样的专用于 Kayak 的 Python API，不过 Kayak 的 XML 接口是很好理解的。本章将为你示范怎样用 Python 的 *urllib2* 包和 *xml.dom.minidom* 包来建立搜索，这两个包都位于标准的 Python 发布包中。

minidom 包

The minidom Package

minidom 是标准 Python 发布包的一部分。它是文档对象模型（DOM）接口的一个轻量级实现，DOM 是一种将 XML 文档当作对象树来看待的标准方式。这个包接受字符串或包含 XML 的开放文件作为输入，然后返回一个对象，我们可以利用该对象轻松地提取信息。例如，我们可以在 Python 会话中输入下面的代码：

```
>>> 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!'
```

由于许多 Web 站点现在都提供了利用 XML 接口来访问信息的方式，所以学习怎样使用 Python 的 XML 包对于集体智慧编程（collective intelligence programming）而言是非常有用的。下面这些是我们将要在 Kayak API 中用到的操作 DOM 对象的重要方法。

`getElementsByTagName(name)`

在整个文档范围内搜索标签名与 `name` 相匹配的元素，然后返回一个包含所有满足条件的 DOM 节点的列表。

`firstChild`

返回对象的首个子节点。在上面的例子中，`r` 的首个子节点就是代表文本“Hello”的节点

`data`

返回与对象有关的数据，大多数情况下这些数据就是该节点所内含的一个 Unicode 文本串。

航班搜索

Flight Searches

首先请新建一个名为 *kayak.py* 的文件，然后加入下面的代码：

```
import time
import urllib2
import xml.dom.minidom

kayakkey='YOURKEYHERE'
```

我们要做的第一件事情是通过编写代码利用开发者密钥来获得一个新的 Kayak 会话。实现此功能的函数会向 apisession 发送一个带有 token 参数（设有你的开发者密钥）的请求。由该 URL 所返回的 XML 中会包含一个 sid 标签，内有 session ID：

```
<sid>1-hX4lII_wS$8b06aO7kHj</sid>
```

下面的函数只须对 XML 进行解析，以得到 sid 标签的内容。请将该函数加入 *kayak.py* 文件中：

```
def getkayaksession():
  # 构造 URL 以开启一个会话
  url='http://www.kayak.com/k/ident/apisession?token=%s&version=1' % kayakkey

  # 解析返回的 XML
  doc=xml.dom.minidom.parseString(urllib2.urlopen(url).read())

  # 找到<sid>xxxxxxxx</sid>标签
  sid=doc.getElementsByTagName('sid')[0].firstChild.data
  return sid
```

下一步是新建一个函数，开始进行航班搜索。用于该搜索的 URL 会很长，因为它包含了供航班搜索之用的所有参数。这其中最为重要的参数包括 sid（getkayaksession 函数返回的会话 ID）、destination，以及 depart_date。

返回的 XML 有个名为 searchid 的标签，函数将采用与 getkayaksession 同样的方式来提取 XML 中的内容。因为搜索也许会花费很长的时间，所以该调用最后实际上不会返回任何结果——它仅仅是启动搜索，然后返回一个可以用来获得结果的 ID。

请将该函数加入 *kayak.py* 文件中：

```
def flightsearch(sid,origin,destination,depart_date):

  # 构造搜索用的 URL
  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)

  # 得到 XML
  doc=xml.dom.minidom.parseString(urllib2.urlopen(url).read())

  # 提取搜索用的 ID
  searchid=doc.getElementsByTagName('searchid')[0].firstChild.data

  return searchid
```

最后，我们还需要一个函数来不断地请求结果，直到没有任何新结果获得为止。Kayak 提供了另外一个 URL——flight，它会给出航班的查询结果。在返回的 XML 中有一个 morepending 标签，直到搜索过程完成为止，该标签中始终会包含一个“true”的字样。这个函数须要一直请求页面到 morepending 不再为真（true）为止，这样函数才能够得到完整的结果。

请将该函数加入 *kayak.py* 文件中：

```
def flightsearchresults(sid,searchid):

  # 删除开头的$和逗号，并把数字转化成浮点类型
  def parseprice(p):
    return float(p[1:].replace(',',''))

  # 遍历检测
  while 1:
    time.sleep(2)

    # 构造检测所用的URL
    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())

    # 寻找morepending标签，并等待其不再为true
    morepending=doc.getElementsByTagName('morepending')[0].firstChild
    if morepending==None or morepending.data=='false': break

  # 现在，下载完整的列表
  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())

  # 得到不同元素组成的列表
  prices=doc.getElementsByTagName('price')
  departures=doc.getElementsByTagName('depart')
  arrivals=doc.getElementsByTagName('arrive')

  # 用zip将它们连在一起
  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])
```

注意：函数在结尾处得到了所有的 price、depart 和 arrive 标签。对它们三者而言，各自有相同数量的一组数据——每三个数据对应一次航班——因此我们可以用 zip 函数将它们连在一起，形成一个大列表中的若干元组。由于出发和到达的信息是以空格分隔的日期加时间的形式给出的，因此我们可以用函数将字符串分隔以得到时间值。函数还将价格传递给了 parseprice，将其转化成浮点类型。

为了确保一切正常，现在我们可以在自己的 Python 会话中尝试一下实际的航班搜索（记住请把日期改成将来的某个时间）：

```
>>> 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)]
```

航班数据是按价格排序的方式返回的，对于价格相同的航班，则按时间排序。这样的输出结果非常不错，因为正如此前提到的那样，出现于结果中的相似解是聚集在一起的。为了将该函数和其余代码整合在一起，我们唯一要做的就是，利用原先已从文件中加载进来的相同结构，给 Glass 一家的不同成员建立一个完整的日程安排。为此我们只须遍历人员列表，然后对往返航班进行搜索。请将 `createschedule` 函数加入 *kayak.py* 文件中：

```
def createschedule(people,dest,dep,ret):
  # 得到搜索用的会话 id
  sid=getkayaksession()
  flights={}

  for p in people:
    name,origin=p
    # 往程航班
    searchid=flightsearch(sid,origin,dest,dep)
    flights[(origin,dest)]=flightsearchresults(sid,searchid)
    # 返程航班
    searchid=flightsearch(sid,dest,origin,ret)
    flights[(dest,origin)]=flightsearchresults(sid,searchid)

  return flights
```

现在，我们可以尝试利用实际的航班数据为这一家进行航班安排的优化了。Kayak 的搜索过程可能要花费一定的时间，因此开始时可以将搜索范围限定在头两名家庭成员。请在你的 Python 会话中输入下列代码：

```
>>> 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
```

恭喜你！刚刚你已经根据实时的航班数据进行了一次优化的过程。由于搜索的范围变大了，因而我们不妨借此机会试验一下算法的最大执行速度（maximum velocity）和学习速率（learning rate）。

有很多种方式可以去扩展这一函数。我们可以将它与天气搜索结合起来，以找到价格合理、在旅游期间气候温暖的优化方案，或者将之与旅馆搜索结合起来，找到航班和旅馆住宿价格都合理的目标解。Internet 上有数以千计的站点提供了旅游目的地的相关数据，这些数据都可以为优化算法所利用。

尽管在日常搜索（searches per day）方面 Kayak API 还是有局限的，但是它的确为我们返回了可以直接与任何航班或旅馆进行交易的链接，这意味着我们可以很方便地将这个 API 集成到任何应用程序中去。

涉及偏好的优化
Optimizing for Preferences

我们已经看到了一个可以用优化算法来解决问题的例子，但是还有许多表面上看似不相关的问题，也可以用同样的方法来解决。请记住，利用优化算法解决问题的基本要求是：问题本身有一个定义好的成本函数，并且相似的解会产生相似的结果。并非每一个具有此类特征的问题都能用优化算法来解决，但是优化算法很有可能会返回一些此前我们未曾考虑到的值得关注的结果。

本节我们将考查另一个不同的问题，这个问题很明显要借助优化算法来加以解决。其一般的表述是：如何将有限的资源分配给多个表达了偏好的人，并尽可能使他们都满意（或者根据他们的意愿，尽可能地满足他们的需要）。

学生宿舍优化问题
Student Dorm Optimization

本节中的示例问题是，依据学生的首选和次选，为其分配宿舍。尽管这是一个非常具体化的例子，但是将这种情况推广到其他问题是非常容易的——完全相同的代码可以用于在线纸牌游戏中玩家的牌桌分配，也可以用于大型编程项目中开发人员的 bug 分配，甚或用于家庭成员中的家务分配。须要再次说明的是，这类问题的目的是为了从个体中提取信息，并将其组合起来产生出优化的结果。

本例中有 5 间宿舍，每间宿舍有两个隔间，由 10 名学生来竞争住所。每一名学生都有一个首选和一个次选。我们新建一个名为 *dorm.py* 的文件，并添加宿舍列表和人员列表，以及每个人的两项选择：

```
import random
import math

# 代表宿舍，每个宿舍有两个可用的隔间
dorms=['Zeus','Athena','Hercules','Bacchus','Pluto']

# 代表学生及其首选和次选
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'))]
```

马上你就会发现，每个人都不可能满足各自的首选，因为 Bacchus 仅有两个隔间，而想要住进去的人却有三个。将这些人中的任何一位安置于他们的次选宿舍中，都将意味着 Hercules 中没有足够的空间留给选择它的人。

为了易于理解，我们有意将这个问题设计得很小巧，但在真实生活中，问题也许会涉及成百上千名学生在更大数量的宿舍范围内竞争更多的住所。因为这个例子仅有大约 100 000 个可能的解，所以将所有解都尝试一遍并从中找到最优解是可能的。但是当每间宿舍有 4 个隔间时，这一数字会快速增长到上万亿。

和航班问题相比，本题解在表达上更需要一点技巧。理论上，我们可以构造一个数字序列，让每个数字对应于一名学生，表示我们将学生安置在了某一间宿舍。问题在于，这种表达方式无法在题解中体现每间宿舍仅限两名学生居住的约束条件。一个全零序列代表将所有人都安置在了 Zeus 宿舍，这不是一个有效的解。

解决这一问题的一种办法是让成本函数返回一个很高的数值，用以代表无效解，但是这将使优化算法很难找到次优的解（better solutions），因为算法无法确定返回结果是否接近于其他优解（good solutions），甚或是有效的解。一般而言，我们最好不要让处理器的时钟周期浪费在无效解的搜索上。

解决这一问题的更好办法是寻找一种能让每个解都有效的题解表示法。有效解未必是优解；它仅代表恰有两名学生被安置于每间宿舍内。要达到这一目的，一种办法是设想每间宿舍都有两个“槽”，如此，在本例中共计有 10 个槽。我们将每名学生依序安置于各空槽内——第一位可置于 10 个槽中的任何一个内，第二位则可置于剩余 9 个槽中的任何一个内，依次类推。

搜索的定义域必须满足这一约束。请在 *dorm.py* 中加入如下代码行：

```
# [(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)]
```

打印题解的代码示范了槽的工作方式。该函数首先创建一个槽序列，每两个槽对应一间宿舍。然后遍历题解中的每个数字，并在槽序列中找到该数字对应的宿舍号，表示学生被安置的宿舍。此函数将学生和与之对应的宿舍打印输出，随后再将槽从序列中删除，如此，其余学生便不会再被安置于该槽中了。待最后一次迭代结束之后，槽列为空，且每名学生及其对应宿舍也都打印完毕。请将函数加入 *dorm.py* 中：

```
def printsolution(vec):
  slots=[]
  # 为每个宿舍建两个槽
  for i in range(len(dorms)): slots+=[i,i]

  # 遍历每一名学生的安置情况
  for i in range(len(vec)):
    x=int(vec[i])

    # 从剩余槽中选择
    dorm=dorms[slots[x]]
    # 输出学生及其被分配的宿舍
    print prefs[i][0],dorm
    # 删除该槽
    del slots[x]
```

可以在你的 Python 会话中导入上述文件，并试着打印一个题解如下：

```
>>> 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
```

如果你想改变数值查看不同的题解，请记住每个数值必须在合理的域值范围内。在序列中，第一项的值可以介于 0 到 9 之间，第二项的值则介于 0 到 8 之间，依次类推。如果我们设置的某个数值超出了合理的域值范围，则函数将会抛出异常。由于优化函数将会保证数值在域值范围之内（该域值通过定义域参数来指定），因此我们在优化期间不会遇到此类问题。

成本函数

The Cost Function

成本函数的工作方式与打印函数类似。构造一个槽序列，并将用过的槽删除。成本的计算值，是通过将学生的当前宿舍安置情况与他的两项选择进行对比而得到的。如果学生当前被安置的宿舍即是其首选宿舍，则总成本加 0；如果是次选宿舍则加 1；如果不在其选择之列，则加 3：

```
def dormcost(vec):
  cost=0
  # 建立一个槽序列
  slots=[0,0,1,1,2,2,3,3,4,4]

  # 遍历每一名学生
  for i in range(len(vec)):
    x=int(vec[i])
    dorm=dorms[slots[x]]
    pref=prefs[i][1]
    # 首选成本值为 0，次选成本值为 1
    if pref[0]==dorm: cost+=0
    elif pref[1]==dorm: cost+=1
    else: cost+=3
    # 不在选择之列则成本值为 3

    # 删除选中的槽
    del slots[x]

  return cost
```

在构造成本函数时有一条法则很有用，即：尽可能让最优解的成本为零（本例中的最优解就是将每个人都安置于其首选宿舍内）。在本例中，我们已经明确不存在最优解，但是知道最优解的成本为零，却可以使我们了解到目前与最优解的差距有多少。这一法则的另一个好处在于，当优化算法找到一个最优解时，我们可以让优化算法停止搜寻更优的解。

执行优化函数

Running the Optimization

有了题解的表示形式、成本函数，以及结果打印输出函数，我们就可以执行此前定义好的优化函数了。请在你的 Python 会话中输入如下内容：

```
>>> 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
```

同样，我们可以调整输入参数，看一看遗传优化算法是否能更快地找到一个优解。

网络可视化
Network Visualization

本章的最后一个例子将向大家展示优化算法的另一种用途，这与前述的其他问题丝毫没有任何关连性。我们要讨论的是网络的可视化问题。此处的网络，意指任何彼此相连的一组事物。像 MySpace、Facebook 或 LinkedIn 这样的社会网络便是在线应用领域中的一个极好的例子。在那里，人们因互为朋友或具备特定关系而彼此相连。网站的每一位成员可以选择与他们相连的其他成员，共同构筑一个人际关系网络。将这样的网络可视化输出，以明确人们彼此间的关系结构——例如寻找联络人（那些认识许多其他朋友的人，或是联系其他私人小圈子的人）——是一件颇有意义的事情。

布局问题
The Layout Problem

为了展示一大群人及其彼此间的关联，我们将网络绘制成图，绘制时会遇到一个问题，我们应该如何安置图中的每个人名（或头像）呢？以图 5-7 中的网络为例。

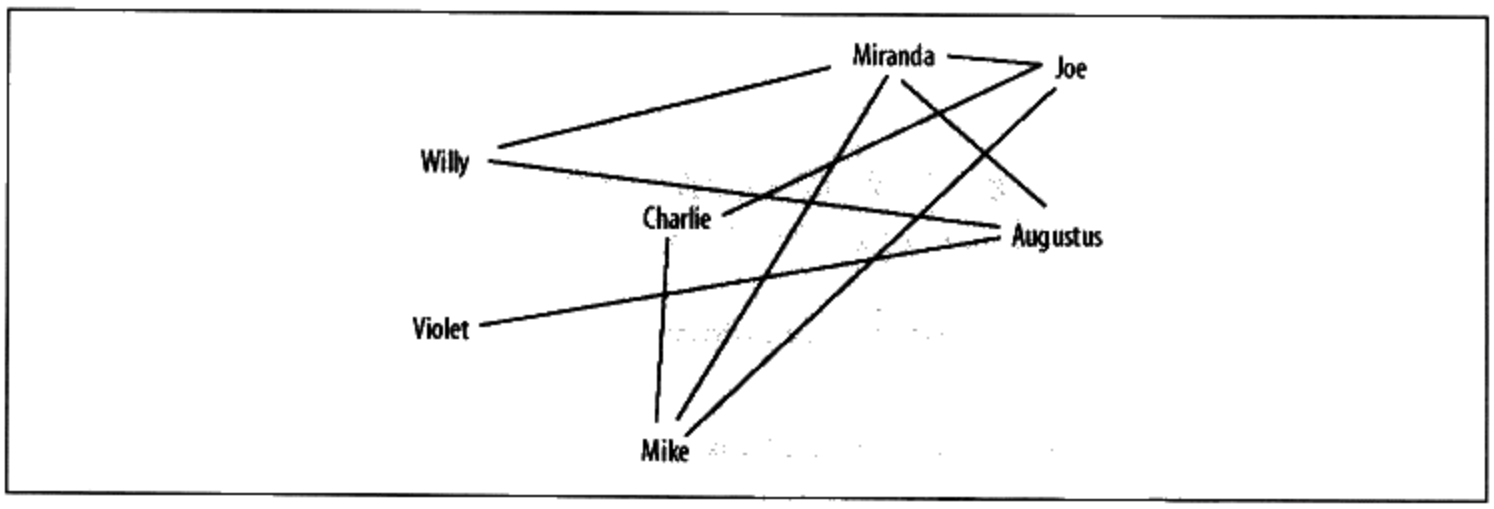


图 5-7：混乱的网络布局

在本图中，我们可以看到 Augustus 是 Willy、Violet 和 Miranda 的朋友。但是，网络的布局有点杂乱，而且增加更多的人会使布局非常地混乱不堪。一个更为清晰的布局如图 5-8 所示。

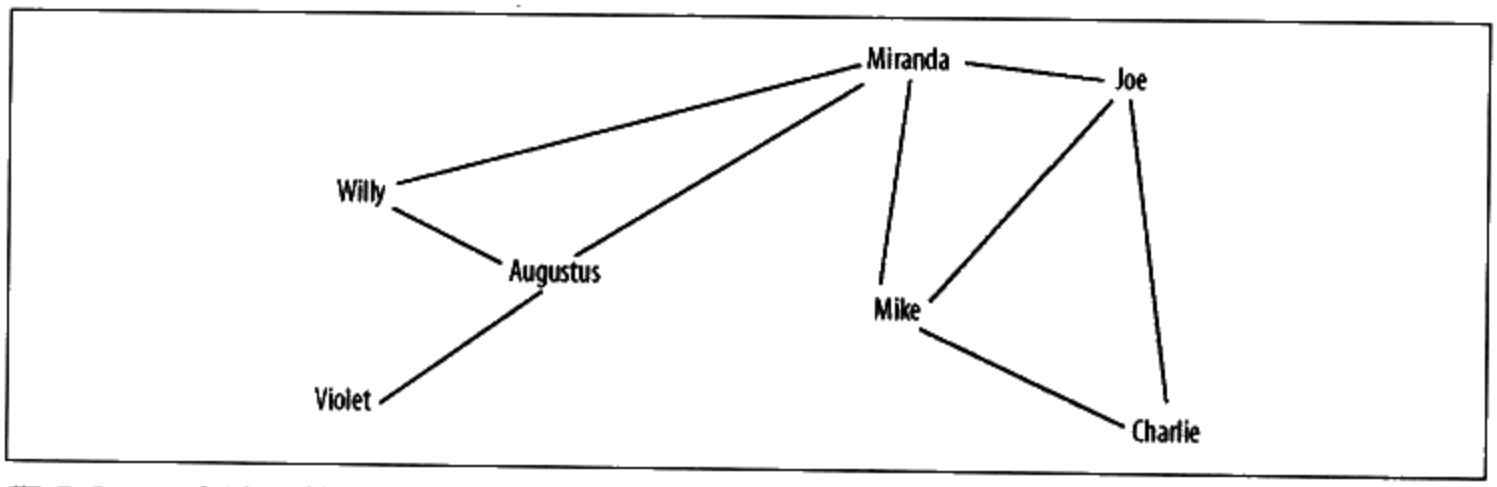


图 5-8：一个清晰的网络布局

本节我们将考虑如何运用优化算法来构建更好的而非杂乱无章的网络图。首先，我们新建一个名为 *socialnetwork.py* 的文件，并加入一些事实数据，这些数据代表着社会网络的某一个小部分：

```
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')]
```

此处，我们的目标是要建立一个程序，令其能够读取一组有关于谁是谁的朋友的事实数据，并生成一个易于理解的网络图。要完成这项工作，通常须要借助于**质点弹簧算法**（mass-and-spring algorithm）。这一算法是从物理学中建模而来的：各结点彼此向对方施以推力并试图分离，而结点间的连接则试图将关联结点彼此拉近。如此一来，网络便会逐渐呈现出这样一个布局：未关联的结点被推离，而关联的结点则被彼此拉近——却又不会靠得很拢。

遗憾的是，质点弹簧算法无法避免交叉线。这使得我们很难在一个拥有大量连接的网络中观察结点的关联情况，因为追踪彼此交叉的连线是颇具难度的。不过，假如使用优化算法

来构建布局的话，那么我们只须要确定一个成本函数，并尝试令它的返回值尽可能地小。在本例中，一个值得一试的成本函数是计算彼此交叉的连线数。

计算交叉线
Counting Crossed Lines

为了能够使用早先定义过的那些优化函数，我们须要将题解表示为一个数值序列。所幸的是，将这一特定问题表示成一个数值序列是非常容易的——每个结点都有 x 和 y 坐标，因此我们可以将所有结点的坐标放入一个长长的列表中：

```
sol=[120,200,250,125 ...
```

在上例中，Charlie 位于(120,200)，Augustus 位于(250,125)，凡此种种，不一而足。

随后，新的成本函数只须对彼此交叉的连线进行计数即可。有关两线交叉的公式出处，超越了本章讨论的范围，不过其基本思路就是计算线条的“分数值”（此处每条线都是“交叉”的）。如果两条线的分数值介于 0（表示线的一端）和 1（表示线的另一端）之间，则它们彼此交叉。反之，则不交叉。

该函数遍历每一对连线，并利用连线端点的当前坐标来判定它们是否交叉。如果交叉，则总分加 1。请将 crosscount 加入 *socialnetwork.py*：

```
def crosscount(v):
  # 将数字序列转换成一个 person:(x,y) 的字典
loc=dict([(people[i],(v[i*2],v[i*2+1])) for i in range(0,len(people))])
total=0

# 遍历每一对连线
for i in range(len(links)):
  for j in range(i+1,len(links)):

    # 获取坐标位置
    (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)

    # 如果两线平行，则 den==0
    if den==0: continue

    # 否则，ua 与 ub 就是两条交叉线的分数值
    ua=((x4-x3)*(y1-y3)-(y4-y3)*(x1-x3))/den
    ub=((x2-x1)*(y1-y3)-(y2-y1)*(x1-x3))/den
```

```
    # 如果两条线的分数值介于 0 和 1 之间，则两线彼此交叉
    if ua>0 and ua<1 and ub>0 and ub<1:
      total+=1
  return total
```

上述搜索算法的定义域就是每组坐标的域值范围。举例而言，假设我们将网络绘制于400*400 像素的图中，则为了留出一定的页边，定义域可以稍小于该范围值。请将下列代码行加入 *socialnetwork.py* 的末尾处：

```
domain=[(10,370)]*(len(people)*2)
```

现在，我们可以尝试利用前述的优化算法，以寻找极少有连线交叉情况的题解了。请将 *socialnetwork.py* 导入你的 Python 会话中，并尝试几种优化算法：

```
>>> 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]
```

利用模拟退火算法有可能会找到极少有连线交叉情况的题解，但是返回的结果却是难以理解的坐标序列。下一节中，我们将向大家展示如何编写程序来自动绘制网络。

绘制网络

Drawing the Network

绘制网络须要用到第 3 章中曾经使用过的 Python Imaging Library。如果还没有安装该库，请参考附录 A，按其指示获取该库的最新版本，并将之安装到你的 Python 实例中。

绘制网络的代码非常简单易懂。全部代码要做的工作包括：建立一个 image 对象，绘制介于不同人之间的连线，并为每个人绘制相应的结点。我们将人名放到最后绘制，这样就不会被连线遮盖了。请将该函数加入 *socialnetwork.py*：

```
def drawnetwork(sol):
  # 建立 image 对象
  img=Image.new('RGB',(400,400),(255,255,255))
  draw=ImageDraw.Draw(img)

  # 建立标示位置信息的字典
  pos=dict([(people[i],(sol[i*2],sol[i*2+1])) for i in range(0,len(people))])
```

```
# 绘制连线
for (a,b) in links:
  draw.line((pos[a],pos[b]),fill=(255,0,0))

# 绘制代表人的结点
for n,p in pos.items():
  draw.text(p,n,(0,0,0))

img.show()
```

要在你自己的 Python 会话中执行本函数，只须重新载入模块，调用该函数，并传入你的题解即可：

```
>>> reload(socialnetwork)
>>> drawnetwork(sol)
```

图 5-9 给出了一种可能的优化结果。

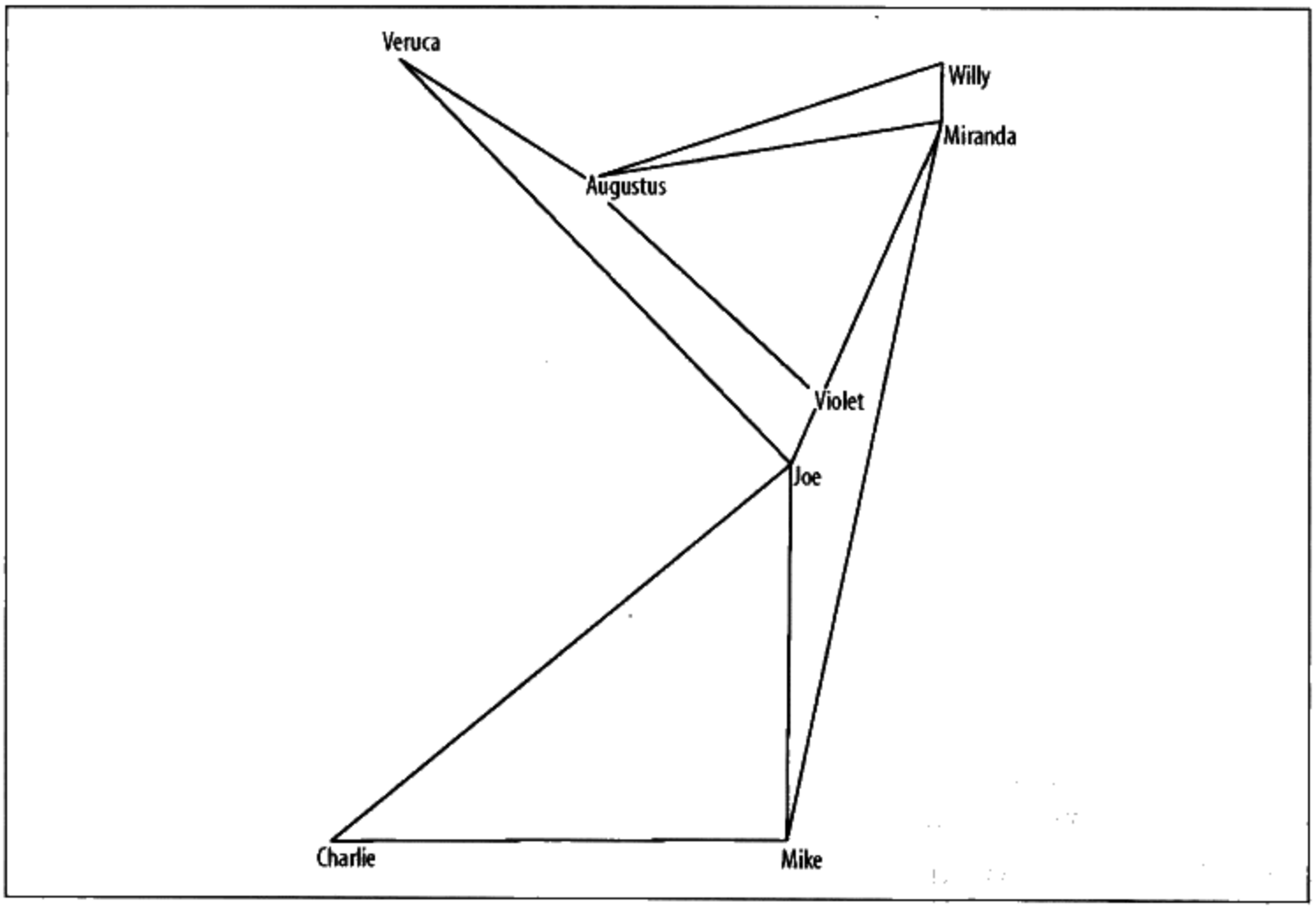


图 5-9：由无交叉连线（no-crossed-lines）优化算法形成的布局

当然，你的题解可能会有别于上述计算结果。有时题解可能会看起来非常的古怪——这是因为我们的目标只是令交叉线的数目最小化，成本函数并没有排除诸如两线夹角非常小，或两结点间距非常近这样的布局情况。就这一点而言，优化算法就像是一个忠实满足你愿

望的精灵（angles）一样。所以，清楚你到底想要什么是非常重要的。时常会有题解满足原本的“最优”条件，但却并非是我们想要的结果。

如果要对一个两结点放置太近的题解进行“判罚”（penalize），最简单的办法就是计算两结点间的距离并除以一个预期的最小距离。我们可以将如下代码添加至 crosscount 的末尾处（return 语句之前），以提供这一附加的判断。

```
for i in range(len(people)):
  for j in range(i+1,len(people)):
    # 获得两结点的位置
    (x1,y1),(x2,y2)=loc[people[i]],loc[people[j]]

    # 计算两结点的间距
    dist=math.sqrt(math.pow(x1-x2,2)+math.pow(y1-y2,2))
    # 对间距小于 50 个像素的结点进行判罚
    if dist<50:
      total+=(1.0-(dist/50.0))
```

当一对结点的彼此间距小于 50 个像素时，上述代码将产生一个比原来更高的成本值，该值与其距离的远近成比例。如果两结点恰好位置相重，则其值为 1。再次执行优化算法，看看能否形成一个分布较为开阔的布局。

其他可能的应用场合

Other Possibilities

本章向大家展示了优化算法的三种截然不同的应用，但这只是众多可能的应用场合中很小的一部分。正如本章一再重申的，关键步骤在于确定题解的表示法及成本函数。如果能做到这些，那么我们就有机会利用优化算法来对问题进行求解。

关于优化，有这样一项应用也许是值得关注的：或许我们会希望对一群人进行分组，让组员的技能得以均匀分布。在一个小型的竞赛活动中，我们可能希望将参赛者进行组队，使每个队都能在体育、历史、文学，以及电视方面具备足够的知识。另一种可能的应用场合是根据人们的技能搭配情况，为项目组分派任务。优化算法可以找到任务分解的最佳方案，从而使任务列表得以在最短时间内完成。

假设有一个标注了关键字的长长的网站列表，根据用户提供的关键字来寻找一组最佳网站可能也是一件很有意义的事情。最佳网站中所包含的网站，并不需要具备大量彼此公有的关键字，而是要尽可能多地体现由用户提供的关键字。

练习

Exercises

1. **组团旅行的成本函数** 请以飞机上每分钟 0.50 美元的成本将总飞行时间计入成本。然后再尝试追加 20 美元的罚款，以确保任何人都能在上午 8 点之前抵达机场。

2. **退火算法的初始值** 模拟退火算法的结果很大程度上取决于其初始值。请构造一个新的优化函数，用多个初始值来模拟退火，并返回最优解。

3. **遗传优化算法的结束条件** 本章中的函数是以固定迭代次数来进行遗传优化的。请改变算法的结束条件，使其在经过 10 次迭代之后，任一最优解都没有任何改善时，方才结束。

4. **往返定价** 此前通过 Kayak 获取航班数据的函数查找的仅是单程航班。购买往返机票的价格可能会更加便宜。请修改代码取得往返票价，并修改成本函数，令其针对某一特定往返航班进行票价查询，而不是只对单程票价进行求和运算。

5. **学生组对** 假设并非要求学生列出对宿舍的偏好，而是令其表达对同住舍友的偏好。那么你将如何表达学生组对的结果呢？成本函数又将如何定义呢？

6. **连线夹角的判断** 请在连接同一人的两线夹角非常小的时候，为网络布局算法的成本函数再增加一项成本。（提示：可以使用向量的叉乘。）

第6章

文档过滤

Document Filtering

本章将向大家演示如何依据内容来对文档进行分类。文档分类是机器智能（machine intelligence）的一个应用，很有实用价值，而且现在越来越普及。关于文档过滤，最有价值也最为人们所熟知的应用，恐怕要数垃圾邮件过滤了。随着电子邮件的广泛普及与邮件发送的超低成本，人们面临的一大问题是：任何人的邮件地址只要落入不法者之手，便有可能会收到未经许可的商业邮件，致使我们无法阅读到真正感兴趣的邮件。

当然，垃圾信息的问题并非仅限于电子邮件。随着时间的推移，Web 网站已经越来越具有互动的特征了，它们或向用户征求意见，或请求用户提供原创内容，这些行为都会伴以垃圾信息侵扰的问题。例如像 Yahoo! Groups 和 Usenet 这样的公共留言板，就长期遭受着垃圾帖的侵扰。这些帖子或与留言板主题毫不相干，或者就是以兜售可疑产品为目的。现在，博客和维基也遭遇到了同样的问题。每当我们在构建一个允许普通大众一起参与的应用系统时，就始终应该考虑应对垃圾信息的策略。

本章中介绍的算法不是专门针对垃圾信息的。由于这些算法可以解决更为一般性的问题，即学习并鉴别文档所属的分类，因此我们还可以将其应用于一些相比垃圾信息而言不那么令人生厌的问题。一种可能的应用是，根据邮件正文自动将收件箱中的邮件划分为社交类邮件和工作相关类邮件。还有一种可能的应用是，识别出要求回复的邮件，并将其自动转发给最适合的人员进行处理。本章的最后一个例子，会为大家示范如何将来自某一 RSS 订阅源的内容项自动过滤到不同的分类之中。

过滤垃圾信息

Filtering Spa

早期尝试对垃圾信息进行过滤所用的都是基于规则的分类器（rule-based classifiers），使用时会有人事先设计好一组规则，用以指明某条信息是否属于垃圾信息。典型的规则包括：英文大写字母的过度使用，与医学药品相关的单词，或是过于花哨的 HTML 用色等。基于

规则的分类器，其问题很快就显现了出来——垃圾信息制造者在知道了所有规则以后，为了绕开过滤器，其行为就会变得更加隐蔽；而且人们会发现，如果他们的父母不知道关闭大写锁定键（Caps Lock），一些正常的邮件也会被归类成垃圾邮件。

基于规则的过滤器还有另一个问题——是否被当作垃圾信息很大程度上因其所面对的读者和张贴位置的不同而不同。对于某一位特定用户、公告留言板或维基而言，那些可以用来明确指示是否垃圾信息的关键词，在其他场合下可能就会变得相当正常。为了解决这一问题，本章所要考查的程序会在开始阶段和逐渐收到更多消息之后，根据人们提供给它的有关哪些是垃圾邮件，哪些不是垃圾邮件的信息，不断地进行学习。通过这样的方式，我们可以分别为不同的用户、群组或网站建立起各自的应用实例和数据集，它们对垃圾信息的界定将逐步形成自己的观点。

文档和单词

Documents and Words

即将构造的分类器须要利用某些特征来对不同的内容项进行分类。所谓特征，是指任何可以用来判断内容中具备或缺失的东西。当考虑对文档进行分类时，所谓的内容即是文档，而特征则是文档中的单词。当将单词作为特征时，其假设是：某些单词相对而言更有可能会出现于垃圾信息中。这一假设是大多数垃圾信息过滤器背后所依赖的基本前提。不过，特征未必一定是一个个单词；它们也可以是词组或短语，或者任何可以归为文档中缺失或存在的其他东西。

请新建一个文件，取名 *docclass.py*，并在其中加入一个名为 `getwords` 的函数，以从文本中提取特征：

```
import re
import math

def getwords(doc):
  splitter=re.compile('\\W*')
  # 根据非字母字符进行单词拆分
  words=[s.lower() for s in splitter.split(doc)
         if len(s)>2 and len(s)<20]

  # 只返回一组不重复的单词
  return dict([(w,1) for w in words])
```

该函数以任何非字母类字符为分隔符对文本进行划分，将文本拆分成了一个个单词。这一过程只留下了真正的单词，并将这些单词全都转换成了小写形式。

决定采用哪些特征颇具技巧性，也十分重要。特征必须具备足够的普遍性，即时常出现，但又不能普遍到每一篇文档里都能找到。理论上，整篇文档的文本都可以作为特征，但是

除非我们一再收到内容完全相同的邮件，否则这样的特征几乎肯定是毫无价值的。在另一种极端情况下，特征也可以是单个字符。但是由于每一封电子邮件中都有可能会出现所有这些字符，因此要想利用这样的特征将希望看到和不希望看到的文档区分开来是很困难的。即便选择使用单词作为特征，也依然还是会带来一些问题，包括如何正确划分单词，哪些标点符号应该被纳入单词，以及是否应该包含头信息（header information）等。

在根据特征进行判断时还有一点须要考虑，那就是如何才能更好地利用特征将一组文档划归到目标分类中去。例如，前述 `getwords` 函数的代码通过将单词转换为小写形式，从而减少了特征的总数。这意味着，程序会将位于句首以大写字母开头的单词与位于句中全小写形式的单词视为相同——这样做非常好，因为具有不同大小写形式的同一单词往往代表的含义是相同的。然而，上述函数完全没有考虑到被用于许多垃圾信息中的“SHOUTING 风格”（译注 1），而这一点可能对区分垃圾邮件和非垃圾邮件是至关重要的。除此以外，如果超过半数以上的单词都是大写时，那就说明必定会有其他的特征存在。

正如你所看到的，在选择特征集时须要做大量的权衡，而且还要不断地进行调整。不过眼下，可以暂且使用这个简单的 `getwords` 函数；在本章的后续部分，我们还将了解到有关特征提取的一些改进方法。

对分类器进行训练

本章中讨论的分类器可以通过接受训练的方式来学习如何对文档进行分类。本书中的许多其他算法，例如我们在第 4 章中见到过的神经网络，都是通过读取正确答案的样本进行学习的。如果分类器掌握的文档及其正确分类的样本越多，其预测的效果也就越好。人们专门设计分类器，其目的也就在于此，即：从极为不确定的状态开始，随着分类器不断了解到哪些特征对于分类而言更为重要，其确定性也在逐渐地增加。

我们要做的第一件事情，是编写一个代表分类器的类。这个类将对分类器到目前为止所掌握的信息进行封装。以这样的方式构造 Python 模块的好处在于，我们可以针对不同的用户、群组或查询，建立起多个分类器实例，并分别对它们加以训练，以响应特定群组的需求。请在 *docclass.py* 中新建一个名为 `classifier` 的类：

```
class classifier:
  def __init_  _(self,getfeatures,filename=None):
    # 统计特征/分类组合的数量
    self.fc={}
    # 统计每个分类中的文档数量
    self.cc={}
    self.getfeatures=getfeatures
```

译注 1：此处是指许多垃圾邮件中所采用的将单词以大写形式书写的手段。

该类中有 3 个实例变量，它们分别是 fc、cc 和 getfeatures。变量 fc 将记录位于各分类中的不同特征的数量。例如：

```
{'python': {'bad': 0, 'good': 6}, 'the': {'bad': 3, 'good': 3}}
```

上述示例表明，单词“the”在被划归“bad”类的文档中已经出现了 3 次，而在被划归“good”类的文档中也出现了 3 次。而单词“Python”却只在“good”类的文档中出现过。

变量 cc 是一个记录各分类被使用次数的字典。这一信息是我们稍后即将讨论的概率计算所需的。最后一个实例变量，getfeatures，对应于一个函数，其作用是从即将被归类的内容项中提取出特征来——在本例中，就是我们刚才定义过的 getwords 函数。

类中定义的方法不会直接引用这些字典，因为这会有碍于将训练数据存入文件或数据库的潜在选择。请加入下列辅助函数，以实现计数值的增加和获取：

```
# 增加对特征/分类组合的计数值
def incf(self,f,cat):
  self.fc.setdefault(f,{})
  self.fc[f].setdefault(cat,0)
  self.fc[f][cat]+=1

# 增加对某一分类的计数值
def incc(self,cat):
  self.cc.setdefault(cat,0)
  self.cc[cat]+=1

# 某一特征出现于某一分类中的次数
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

# 属于某一分类的内容项数量
def catcount(self,cat):
  if cat in self.cc:
    return float(self.cc[cat])
  return 0

# 所有内容项的数量
def totalcount(self):
  return sum(self.cc.values())

# 所有分类的列表
def categories(self):
  return self.cc.keys()
```

train 方法接受一个内容项（本例中为文档）和一个分类作为参数。它利用 getfeatures 函数，将内容项拆分为彼此独立的各个特征。然后调用 incf 函数，针对该分类为每个特征增加计数值。最后，函数会增加针对该分类的总计数值：

```
def train(self,item,cat):
  features=self.getfeatures(item)
  # 针对该分类为每个特征增加计数值
  for f in features:
    self.incf(f,cat)

  # 增加针对该分类的计数值
  self.incc(cat)
```

请启动一个新的 Python 会话，并引入该模块，我们可以来检查一下这个类是否可用：

```
$ 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
```

此处，我们用一个函数将训练用的样本数据导入到分类器中是很有价值的，因为这样就无须在每次创建分类器的时候再对其进行手工训练了。请将该函数加入 *docclass.py* 的开始处：

```
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')
```

计算概率

Calculating Probabilities

既然我们已经对一封电子邮件在每个分类中的出现次数进行了统计，那么接下来的工作就是要将其转换成概率了。所谓概率，是指一个介于 0 和 1 之间的数字，用以指示某一事件发生的可能性。在本例中，可以用一个单词在一篇属于某个分类的文档中出现的次数，除以该分类的文档总数，计算出单词在分类中出现的概率。

请将一个名为 fprob 的方法加入 classifier 的类中：

```
def fprob(self,f,cat):
  if self.catcount(cat)==0: return 0
```

```
        # 特征在分类中出现的总次数，除以分类中包含内容项的总数
        return self.fcount(f,cat)/self.catcount(cat)
```

我们称上述概率为条件概率，通常记为 *Pr*(A | B) ，读作“在给定 B 的条件下 A 的概率”。在本例中，目前我们所求得的值对应于 *Pr(word | classification)*，即：对于一个给定的分类，某个单词出现的概率。

可以在你的 Python 会话中尝试执行一下该函数：

```
>>> reload(docclass)
<module 'docclass' from 'docclass.py'>
>>> cl=docclass.classifier(docclass.getwords)
>>> docclass.sampletrain(cl)
>>> cl.fprob('quick','good')
0.66666666666666663
```

从执行结果中我们可以看到，在三篇被归类为“good”的文档中，有两篇文档出现了单词“quick”，即：一篇“good”分类的文档中包含该单词的概率为，*Pr(quick | good)* = 0.666（有 2/3 的机会）。

从一个合理的推测开始

Starting with a Reasonable Guess

`fprob` 方法针对目前为止见到过的特征与分类，给出了一个精确的结果。但是它有一个小小的问题——只根据以往见过的信息，会令其在训练的初期阶段，对那些极少出现的单词变得异常敏感。在训练用的样本数据中，单词“money”只在一篇文档中出现过，并且由于这是一则涉及赌博的广告，因此文档被划归为了“bad”类。由于单词“money”在一篇“bad”类的文档中出现过，而任何“good”类的文档中都没有该单词，所以此时利用 `fprob` 计算所得的单词“money”在“good”分类中出现的概率为 0。这样做有一些偏激，因为“money”可能完全是一个中性词，只是恰好先出现在了一篇 “bad”类的文档中而已。伴随着单词越来越多地出现在同属于一个分类的文档中，其对应的概率值也逐渐接近于 0，恐怕这样才会更合理一些。

为了解决上述问题，在我们手头掌握的有关当前特征的信息极为有限时，我们还须要根据一个假设的概率来作出判断。一个推荐的初始值是 0.5。我们还须要确定为假设的概率赋以多大的权重——权重为 1 代表假设概率的权重与一个单词相当。经过加权的概率值返回的是一个由 `getprobability` 与假设概率组成的加权平均。

在单词“money”的例子中，针对“money”的加权概率对于所有分类而言均是从 0.5 开始的。待到在 classifier 训练期间接受了一篇 “bad”分类的文档，并且发现“money”适合于“bad”分类时，其针对“bad”分类的概率就会变为 0.75。这是因为：

```
(weight*assumedprob + count*fprob)/(count+weight)
= (1*1.0+1*0.5)/(1.0 + 1.0)
= 0.75
```

请将 weightedprob 方法加入 classifier 类中：

```
def weightedprob(self,f,cat,prf,weight=1.0,ap=0.5):
  # 计算当前的概率值
  basicprob=prf(f,cat)

  # 统计特征在所有分类中出现的次数
  totals=sum([self.fcount(f,c) for c in self.categories()])

  # 计算加权平均
  bp=((weight*ap)+(totals*basicprob))/(weight+totals)
  return bp
```

现在我们可以在自己的 Python 会话中尝试执行一下该函数了。由于新建一个 classifier 类的实例将清除其已有的训练数据，因此请重新加载模块，并再次运行 sampletrain 方法：

```
>>> 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
```

正如我们所看到的，随着单词的概率从假设的初始值开始被逐步地“拉动”，重新运行 sampletrain 方法后使 classifier 对各个单词的概率变得更加确信了。

选择 0.5 作为假设的概率初始值仅仅是因为它介于 0 和 1 的正中间。不过，也有可能我们已经掌握了更多的背景信息，从而使假设更加有据可依，这一点即便对于一个完全没有经过训练的分类器而言，也是有可能的。例如，一个准备对垃圾信息过滤器进行训练的人，可以利用他人训练过的垃圾过滤器，将其所得的概率值作为假设的概率初始值。使用者还可以专门为自己设计个性化的垃圾信息过滤器，只是不管怎样，对于一个过滤器而言，它最好应该有能力处理极少会出现的单词。

朴素分类器

A Naive Classifier

一旦我们求出了指定单词在一篇属于某个分类的文档中出现的概率，就需要有一种方法将各个单词的概率进行组合，从而得出整篇文档属于该分类的概率。本章将分别考查两种不同的分类方法。这两种方法在大多数场合下都是可以使用的，只不过它们在面对特定任务时，在算法的性能级别上有些微的不同。本节中我们要讨论的分类器被称为**朴素贝叶斯分类器**。

这种方法之所以被冠以**朴素**二字，是因为它假设将要被组合的各个概率是**彼此独立**的。即，一个单词在属于某个指定分类的文档中出现的概率，与其他单词出现于该分类的概率是不相关的。事实上这个假设是不成立的，因为你也许会发现，与有关 Python 编程的文档相比，包含单词“casino”的文档更有可能包含单词“money”。

这意味着，我们无法将采用朴素贝叶斯分类器所求得的结果实际用作一篇文档属于某个分类的概率，因为这种独立性的假设会使其得到错误的结果。不过，我们还是可以对各个分类的计算结果进行**比较**，然后再看哪个分类的概率最大。在现实中，若不考虑假设的潜在缺陷，朴素贝叶斯分类器将被证明是一种非常有效的文档分类方法。

整篇文档的概率

Probability of a Whole Document

为了使用朴素贝叶斯分类器，首先我们须要确定整篇文档属于给定分类的概率。正如此前讨论过的，我们须要假设概率的彼此独立性，即：可以通过将所有的概率相乘，计算出总的概率值。

例如，假设我们已经注意到有 20%的“bad”类文档中出现了单词“Python”——*Pr(Python | Bad)* = 0.2——同时有 80%的文档出现了单词“casino”（*Pr(Casino | Bad)* = 0.8）。那么，预期两个单词出现于同一篇“bad”类文档中的独立概率为——*Pr(Python & Casino | Bad)*——$0.8 \times 0.2 = 0.16$。从中我们会发现，计算整篇文档的概率，只须将所有出现与某篇文档中的各单词的概率相乘即可。

请在 *docclass.py* 中，新建一个 `classifier` 的子类，取名 `naivebayes`，并为其添加一个 `docprob` 方法，该方法的作用是提取特征（单词）并将所有单词的概率值相乘以求出整体概率：

```
class naivebayes(classifier):
  def docprob(self,item,cat):
    features=self.getfeatures(item)

    # 将所有特征的概率相乘
    p=1
    for f in features: p*=self.weightedprob(f,cat,self.fprob)
    return p
```

现在我们已经知道了如何计算 *Pr(Document | Category)*，不过只做到这一步还不行。为了对文档进行分类，我们真正需要的是 *Pr(Category | Document)*。换言之，就是对于一篇**给定**的文档，它属于某个分类的概率是多少？所幸的是，一位名叫 Thomas Bayes 的英国数学家早在大约 250 年前就已经找到了解决这一问题的办法。

贝叶斯定理简介

A Quick Introduction to Bayes' Theorem

贝叶斯定理是一种对条件概率进行调换求解（flipping around）（译注 2）的方法。它通常被写作：

Pr(A | B) = Pr(B | A) x Pr(A)/Pr(B)

在本例中，即为：

Pr(Category | Document) = Pr(Document | Category) x Pr(Category) / Pr(Document)

*Pr(Document | Category)*的计算方法上一节已经介绍过了，但是等式中的另两个值如何计算呢？*Pr(Category)*是随机选择一篇文档属于该分类的概率，因此就是属于该分类的文档数除以文档的总数。

至于 *Pr(Document)*，我们也可以计算它，但这将会是一项不必要的工作。请记住，我们不会将这一计算结果当作真实的概率值。相反，我们会分别计算每个分类的概率，然后对所有的计算结果进行比较。由于不论计算的是哪个分类，*Pr(Document)* 的值都是一样的，其对结果所产生的影响也完全是一样的，因此我们完全可以忽略这一项。

`prob` 方法用于计算分类的概率，并返回 *Pr(Document | Category)*与 *Pr(Category)*的乘积。请将该方法加入 `naivebayes` 类中：

```
def prob(self,item,cat):
  catprob=self.catcount(cat)/self.totalcount()
  docprob=self.docprob(item,cat)
  return docprob*catprob
```

请在 Python 的执行环境中尝试一下该函数，看看针对不同的字符串和分类，概率值是如何变化的：

```
>>> 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
```

根据训练的数据，我们认为相比于“bad”分类而言，短语“quick rabbit”更适合于“good”分类。

译注 2：根据后面的公式，此处 flipping around 的意思是通过 *P(B|A)*来求 *P(A|B)*，而 *B|A* 对 *A|B* 而言，二者的相对位置正好调了过来。

选择分类

Choosing a Category

构造朴素贝叶斯分类器的最后一个步骤是实际判定某个内容项所属的分类。此处最简单的方法，是计算被考查内容在每个不同分类中的概率，然后选择概率最大的分类。如果我们只是在试图判断“将内容放到哪里最合适”的问题，那么这不失为一种可行的策略，但是在许多应用中，我们无法将各个分类同等看待，而且在一些应用中，对于分类器而言，承认不知道答案，要好过判断答案就是概率值稍大一些的分类。

在垃圾信息过滤的例子中，避免将普通邮件错当成垃圾邮件要比截获每一封垃圾邮件更为重要。收件箱中偶尔收到几封垃圾邮件还是可以容忍的，但是一封重要的邮件则有可能会因为自动过滤到废件箱而被完全忽视。假如我们必须在废件箱中找回自己的重要邮件，那就真的没必要再使用垃圾信息过滤器了。

为了解决这一问题，我们可以为每个分类定义一个最小阈值。对于一封将要被划归到某个分类的新邮件而言，其概率与针对所有其他分类的概率相比，必须大于某个指定的数值才行。这一指定的数值就是阈值。以垃圾邮件过滤为例，假如过滤到“bad”分类的阈值为 3，则针对“bad”分类的概率就必须至少 3 倍于针对 “good”分类的概率才行。假如针对“good”分类的阈值为 1，则对于任何邮件，只要概率确实大于针对“bad”分类的概率，它就是属于“good” 分类的。任何更有可能属于“bad”分类，但概率并没有超过 3 倍以上的邮件，都将被划归到“未知”分类中。

为了定义阈值，请修改初始化方法，在 classifier 中加入一个新的实例变量：

```
def __init_  _(self,getfeatures):
  classifier.__init_  _(self,getfeatures)
  self.thresholds={}
```

请加入几个用于设值和取值的简单方法，令其默认返回为 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]
```

现在，我们可以构建 classify 方法了。该方法将计算每个分类的概率，从中得出最大值，并将其与次大概率值进行对比，确定是否超过了规定的阈值。如果没有任何一个分类满足上述条件，方法就返回默认值。请将该方法加入 classifier 中：

```
def classify(self,item,default=None):
  probs={}
  # 寻找概率最大的分类
```

```
    max=0.0
    for cat in self.categories():
      probs[cat]=self.prob(item,cat)
      if probs[cat]>max:
        max=probs[cat]
        best=cat

    # 确保概率值超出域值*次大概率值
    for cat in probs:
      if cat==best: continue
      if probs[cat]*self.getthreshold(best)>probs[best]: return default
    return best
```

大功告成！现在我们已经建立起了一个完整的文档分类系统。通过构造不同的特征提取方法，我们可以对该系统进行扩展，以实现对任何其他内容的分类。请在你的 Python 会话中试验一下这一分类器：

```
>>> 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'
```

当然，我们还可以修改一下阈值，看看对结果有何影响。一些垃圾信息过滤插件为用户提供了控制阈值的功能，这样一来，只要当前阈值令太多的垃圾邮件进入到收件箱中，或者有大量正常邮件被错归为了垃圾邮件，我们就可以对阈值进行调整。当然，对于另一些涉及文档过滤的应用而言，阈值的定义也可能有所不同；与上述情况不一样，有时，所有分类可能都是平等的，而有时，将内容过滤到“未知”分类则是不可接受的。

费舍尔方法

The Fisher Method

以 R. A. Fisher 的名字命名的**费舍尔方法**，是前面介绍的朴素贝叶斯方法的一种替代方案，它可以给出非常精确的结果，尤其适合垃圾信息过滤。*SpamBayes*，一个用 Python 编写的 Outlook 插件，便采用了这一方法。与朴素贝叶斯过滤器利用特征概率来计算整篇文档的概率不同，费舍尔方法为文档中的每个特征都求得了分类的概率，然后又将这些概率组合起

来，并判断其是否有可能构成一个随机集合。该方法还会返回每个分类的概率，这些概率彼此间可以进行比较。尽管这种方法更为复杂，但是因为它在为分类选择临界值（cutoff）时允许更大的灵活性，所以还是值得一学的。

针对特征的分类概率
Category Probabilities for Features

前面讨论过的朴素贝叶斯过滤器，将所有 *Pr(feature | category)*的计算结果组合起来得到了整篇文档的概率，然后再对其进行调换求解。在本节中，我们将直接计算当一篇文档中出现某个特征时，该文档属于某个分类的可能性，也就是 *Pr(category | feature)*。如果单词“casino”出现于 500 篇文档中，并且其中有 499 篇属于“bad”分类，则“casino”属于“bad”分类的概率将非常接近于 1。

计算 *Pr(category | feature)*的常见方法是：

(具有指定特征的属于某分类的文档数) / (具有指定特征的文档总数)

上述计算公式并没有考虑我们收到属于某一分类的文档可能比其他分类更多的情况。假如我们有许多“good”分类的文档，而“bad”分类的文档则很少，那么一个出现于所有“bad”类文档中的单词，即便邮件内容看上去可能没有问题，该单词属于“bad”分类的概率也依然会更大一些。如果我们假设“未来将会收到的文档在各个分类中的数量是相当的”，那么上述方法就会有更好的表现，因为这使得它们能更有效地利用特征来识别分类。

为了进行归一化计算，函数将分别求得 3 个量：

- 属于某分类的概率 *clf = Pr(feature | category)*
- 属于所有分类的概率 *freqsum = Pr(feature | category)*之和
- *cprob = clf / (clf+nclf)*

请在 *docclass.py* 中为 `classifier` 新建一个子类，取名 `fisherclassifier`，并加入如下方法：

```
class fisherclassifier(classifier):
  def cprob(self,f,cat):
    # 特征在该分类中出现的频率
    clf=self.fprob(f,cat)
    if clf==0: return 0

    # 特征在所有分类中出现的频率
    freqsum=sum([self.fprob(f,c) for c in self.categories()])

    # 概率等于特征在该分类中出现的频率除以总体频率
    p=clf/(freqsum)

    return p
```

基于各分类中所包含的内容项数量相当的假设，该函数返回的概率值，代表了具备指定特征的内容属于指定分类的可能性。可以在你的 Python 会话中看一下这些概率的实际计算结果：

```
>>> reload(docclass)
>>> cl=docclass.fisherclassifier(docclass.getwords)
>>> docclass.sampletrain(cl)
>>> cl.cprob('quick','good')
0.57142857142857151
>>> cl.cprob('money','bad')
1.0
```

上述方法告诉我们，包含单词“casino”的文档是垃圾邮件的概率为 0.9。这与训练数据是相符的，不过这种方法同样也会遇到前文提到的问题——因为算法接触单词的次数太少，所以它有可能会对概率值估计过高。因此，不妨像前文那样，对概率进行加权处理，即：所有概率值均以 0.5 作为初始值，而后伴随不断的训练过程，允许它们逐渐向其他概率值变化。

```
>>> cl.weightedprob('money','bad',cl.cprob)
0.75
```

将各概率值组合起来

Combining the Probabilities

现在，我们须要将对应各个特征的概率值组合起来，形成一个总的概率值。理论上，我们可以将它们连乘起来，利用相乘的结果在不同分类间进行比较。当然，由于特征不是彼此独立的，因此它们并不代表真实的概率，不过这已经比我们在前一节中构造的贝叶斯分类器要好不少了。由费舍尔方法返回的结果是对概率的一种更好的估计，这对于结果报告或临界值判断而言是非常有价值的。

费舍尔方法的计算过程是将所有概率相乘起来，然后取自然对数（Python 中的 *math.log*），再将所得结果乘以-2。请将下列方法加入 `fisherclassifier` 类中，以实现这一计算过程：

```
def fisherprob(self,item,cat):
  # 将所有概率值相乘
  p=1
  features=self.getfeatures(item)
  for f in features:
    p*=(self.weightedprob(f,cat,self.cprob))

  # 取自然对数，并乘以-2
  fscore=-2*math.log(p)

  # 利用倒置对数卡方函数求得概率
  return self.invchi2(fscore,len(features)*2)
```

费舍尔方法告诉我们，如果概率彼此独立且随机分布，则这一计算结果将满足**对数卡方分布**（chi-squared distribution）。也许我们会预料到，不属于某个分类的内容项中，可能会包含针对该分类的不同特征概率的单词（可能会随机出现）；或者，一个属于该分类的内容项中会包含许多概率值很高的特征。通过将费舍尔方法的计算结果传给倒置对数卡方函数，我们会得到一组随机概率中的最大值。

请将倒置对数卡方函数加入 `fisherclassifier` 类中：

```
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)
```

我们依然可以在自己的 Python 会话中试验该函数，看看费舍尔方法是如何对样本字符串进行评价的：

```
>>> 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
```

正如我们所看到的，结果总是介于 0 和 1 之间。这些结果本身即是衡量文档所属分类的一种很好的度量方法。正是由于这一点，分类器本身才有可能变得更为有效。

对内容项进行分类
Classifying Items

我们可以利用 `fisherprob` 的返回值来决定如何进行分类。不像贝叶斯过滤器那样须要乘以阈值，此处我们可以为每个分类指定下限。尔后，分类器会返回介于指定范围内的最大值。在垃圾信息过滤器中，我们可以将“bad”分类的下限值设得很高，比如 0.6；将“good”分类的下限值设置得很低，比如 0.2。这样做可以将正常邮件被错归到“bad”分类的可能性减到最小，同时也会允许少量垃圾邮件进入到收件箱中。任何针对“good”分类的分值低于 0.2，针对“bad”分类的分值低于 0.6 的邮件，都将被划归到“未知”分类中。

请在 fisherclassifier 类中新建一个 init 方法，再增加一个保存临界值的变量：

```
def __init_  _(self,getfeatures):
  classifier.__init_  _(self,getfeatures)
  self.minimums={}
```

请将下述两个用于设值和取值的方法加入类中，默认取值为 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]
```

最后，再添加一个方法，用以计算每个分类的概率，并找到超过指定下限值的最佳结果：

```
def classify(self,item,default=None):
  # 循环遍历并寻找最佳结果
  best=default
  max=0.0
  for c in self.categories():
    p=self.fisherprob(item,c)
    # 确保其超过下限值
    if p>self.getminimum(c) and p>max:
      best=c
      max=p
  return best
```

现在我们可以针对测试数据，利用费舍尔评价方法试验一下分类器了。请在你的 Python 会话中输入如下代码：

```
>>> 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')
>>>
```

此处的执行结果与朴素贝叶斯分类器的结果类似。人们相信，在实践中费舍尔分类器对垃圾信息的过滤效果会更好；只不过对于这样一小组训练数据而言，过滤的效果可能不太明显。应该使用何种分类器要取决于你的应用，没有一种简单方法可以预测出什么样的分类器会更好，或者我们应该使用多大的临界值。所幸的是，利用此处给出的代码，我们应该能够很容易地对两种算法以及各种不同的设置项进行试验。

将经过训练的分类器持久化

Persisting the Trained Classifiers

在任何真实世界的应用中，所有的训练和分类工作都不太可能完全在一次会话中完成。如果分类器被用作 Web 应用的一部分，那么我们就有可能须要将用户在使用系统期间所产生的任何与训练相关的数据保存起来，然后在下一次用户登录之后再将数据恢复。

使用 SQLite

Using SQLite

本节中我们将为大家示范如何利用数据库（本例中为 SQLite）将分类器的训练信息进行持久化。如果我们的应用涉及许多用户同时对分类器进行训练和查询，那么将计数值存入数据库可能是一个明智之举。SQLite 就是我们曾在第 4 章中使用过的数据库。如果你还没有用过 pysqlite，则须要先将其下载并安装；有关下载和安装的详细情况请见附录 A。通过 Python 访问 SQLite 与访问其他数据库是很类似的，因此如果要进行数据库移植应该也非常的容易。

为了将 pysqlite 引入进来，请将下列语句加入 *docclass.py* 的首部：

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

本节中的代码将当前 `classifier` 类中所用的字典结构都替换为了一个持久化的数据存储结构。请在 `classifier` 中添加一个方法，为该分类器打开数据库，并在必要时执行建表操作。这些数据表与它们所替换的字典在结构上是相匹配的：

```
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)')
```

如果我们正打算将分类器移植到另一个数据库上，为了能够在所使用的目标系统上正常运行，有可能须要修改相应的建表语句。

我们还须要替换所有用于获取和累加计数值的辅助函数：

```
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])
```

获取所有分类的列表与文档总数的方法也应该被替换掉：

```
  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]
```

最后，我们须要在训练结束之后添加一条提交语句，以便在所有计数值被更新之后程序能将数据存入数据库。请将下列代码行加入 classifier 中 train 方法的末尾处：

```
    self.con.commit()
```

大功告成！在对 classifier 初始化之后，我们须要调用 setdb 方法，并传入数据库文件的名称。所有训练数据都将被自动存入数据库中，并且能够为任何其他人所使用。我们甚至可以将取自某一分类器的训练数据用于另一种类型的分类器：

```
>>> 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'
```

过滤博客订阅源

Filtering Blog Feeds

为了在真实环境下试验分类器，也为了演示其不同的用途，我们可以将分类器应用于来自某个博客或RSS订阅源的内容项。为此，我们需要用到曾在第3章中介绍过的 Universal Feed Parser。如果你还没有下载相应的函数库，则可以通过访问 *http://feedparser.org* 进行下载。有关安装 Feed Parser 的更多信息请见附录 A。

尽管博客的内容中未必会包含垃圾信息，但是在众多博客所包含的文章中，并非所有的文章都是我们感兴趣的。这也许是因为我们只希望阅读属于某个分类的文章，或者某位作者所撰写的文章，不过通常而言实际情况要比这更为复杂。同样地，我们也可以针对自己感兴趣和不感兴趣的内容定义一些专门的规则——也许我们阅读了一个有关小件装置（gadget）的博客，并且对其中包含单词“cell phone”的内容不感兴趣——但是，假如利用前面已经构造好的分类器来为我们得出上述这些规则，其所需的工作量相对而言会更少一些。

对一个 RSS 订阅源中的内容项进行分类的好处在于，假如我们使用了像 Google Blog Search 这样的博客搜索工具，那么就可以在订阅源的阅读器中对搜索的结果进行定制了。许多人以此来追踪产品和他们感兴趣的内容，甚至还包括他们自己的名字。但是我们会发现，试图利用流量来赚钱的垃圾博客和那些毫无价值的博客也有可能会出现在这些搜索结果当中。

尽管许多订阅源因为拥有的内容项太少而无法进行任何有效的训练，不过在本例中，我们还是可以根据自己的喜好来选择任何的订阅源。在这个特定的例子里，我们使用 Google Blog Search 对单词“Python”进行搜索，其搜索结果都是 RSS 形式的。你可以从 *http://kiwitobes.com/feeds/python_search.xml* 处下载到这些结果。

请新建一个名为 *feedfilter.py* 的文件，并加入下列代码：

```
import feedparser
import re

# 接受一个博客订阅源的URL文件名并对内容项进行分类
def read(feed,classifier):
  # 得到订阅源的内容项并遍历循环
  f=feedparser.parse(feed)
  for entry in f['entries']:
    print
    print '-----'
    # 将内容项打印输出
    print 'Title:     '+entry['title'].encode('utf-8')
    print 'Publisher: '+entry['publisher'].encode('utf-8')
    print
    print entry['summary'].encode('utf-8')


    # 将所有文本组合在一起，为分类器构建一个内容项
    fulltext='%s\n%s\n%s' % (entry['title'],entry['publisher'],entry['summary'])
```

```
    # 将当前分类的最佳推测结果打印输出
    print 'Guess: '+str(classifier.classify(fulltext))

    # 请求用户给出正确分类，并据此进行训练
    cl=raw_input('Enter category: ')
    classifier.train(fulltext,cl)
```

该函数循环遍历所有内容项，并利用分类器得到有关分类的最佳推测结果。它向用户给出最佳推测，并接着询问正确的分类是什么。当我们使用一个新的分类器运行该程序时，起初的推测结果将会带有随机性，但是它们会随着时间的推移逐步得到改善。

上述构建好的分类器是完全通用的。尽管我们利用了垃圾信息过滤的例子来帮助说明每段代码的工作原理，但是分类的类别则可以是任何形式的内容。如果你正在使用*python_search.xml*，那么其中也许包含了4个分类——一个是关于编程语言的，一个是关于电影《Monty Python》的，一个是关于蟒蛇的，还有一个则是涉及任何其他内容的。请在你的 Python 会话中试着运行一下这个交互式的过滤器，设置好一个分类器，并将其传给`feedfilter`：

```
>>> import feedfilter
>>> cl=docclass.fisherclassifier(docclass.getwords)
>>> cl.setdb('python_feed.db')  # 仅当你使用的是 SQLite
>>> feedfilter.read('python_search.xml',cl)

-----
Title:     My new baby boy!
Publisher: 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...
Guess: None
Enter category: snake

-----
Title:     If you need a laugh...
Publisher: Kate&#39;s space

Even does 'funny walks' from Monty <b>Python</b>. He talks about all the ol'
Guess: snake
Enter category: monty

-----
Title:     And another one checked off the list..New pix comment ppl
Publisher: 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
Guess: snake
Enter category: snake
```

我们会发现，推测的结果随着时间的推移在逐渐的改善。由于没有太多有关于蛇的样本信息，尤其是它们被进一步划分成了宠物蛇和时尚一类的帖子，因此分类器对于这一分类的推测结果时常是错误的。当执行完整个训练之后，我们就可以得到针对于指定特征的概率值了——包括针对给定分类的单词概率，以及针对给定单词的分类概率：

```
>>> 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
```

从上述结果中我们可以看到，由于每个内容项都包含单词“python”，因此该单词的概率被等分了。在涉及《Monty Python》的内容项中，有 25%的文章包含了单词“Eric”，而其他内容项中则没有出现该单词。因此就“Eric”而言，对于给定分类的单词概率为 0.25，而对于给定单词的分类概率则为 1.0。

对特征检测的改进

Improving Feature Detection

目前为止的所有例子中，建立特征列表的函数只是简单地使用了非字母非数字类字符作为分隔符对单词进行拆分。函数还将所有单词都转换成了小写形式，因此我们没有办法检测大写单词的过度使用问题。有几种不同的方法可以对其加以改进。

- 不真正区分大写和小写的单词，而是将“含有许多大写单词”这样的现象作为一种特征。
- 除了单个单词以外，还可以使用词组。
- 捕获更多的元信息，如：是谁发送了电子邮件，或者一篇博客被提交到了哪个分类下，可以将这样的信息标示为元信息。
- 保持 URL 和数字原封不动，不对其进行拆分。

请记住，这不仅是让特征更有针对性这么简单。特征必须出现于多篇文档之中，因为它们对分类器而言起了很大的作用。

`classifier` 类可以接受任何形式的函数作为 `getfeatures`，它就传入的内容项运行该函数，并预期返回一个针对该内容项的包含所有特征的列表或字典。由于这种通用性，我们可以轻松地建立起一个函数，令其处理比简单的字符串而言更为复杂的类型。例如，当对一个博客订阅源的内容项进行分类时，我们可以编写一个函数，令其接受整篇文章的内容，而非从中提取出来的文本，然后标注出各个单词的来源。我们还可以从文本正文中找出词

组，而从主题中找出个别的单词。此外，对记录文章创建者的字段进行拆分可能也是毫无意义的，因为名叫“John Smith”的人所提交的内容，是不太可能会告诉我们任何有关其他叫 John 的人所提交的内容的。

请将这个新的特征提取函数加入 *feedfilter.py* 中。请注意，它需要的是一个订阅源的内容项作为参数，而非字符串：

```
def entryfeatures(entry):
  splitter=re.compile('\\W*')
  f={}

  # 提取标题中的单词并进行标示
  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

  # 提取摘要中的单词
  summarywords=[s.lower() for s in splitter.split(entry['summary'])
        if len(s)>2 and len(s)<20]

  # 统计大写单词
  uc=0
  for i in range(len(summarywords)):
    w=summarywords[i]
    f[w]=1
    if w.isupper(): uc+=1

    # 将从摘要中获得的词组作为特征
    if i<len(summarywords)-1:
      twowords=' '.join(summarywords[i:i+1])
      f[twowords]=1

  # 保持文章创建者和发布者名字的完整性
  f['Publisher:'+entry['publisher']]=1

  # UPPERCASE 是一个“虚拟”单词，用以指示存在过多的大写内容
  if float(uc)/len(summarywords)>0.3: f['UPPERCASE']=1

  return f
```

上述函数从文档和摘要中提取单词，就如同此前的 getwords 那样。它将所有位于标题中的单词标识出来，并将其作为特征。位于摘要中的单词以及前后连贯的词组，也被当作了特征。函数还将未经拆分的内容创建者和发布者当作了特征。最后，它统计了摘要中大写单词的出现次数。一旦有超过 30% 的单词为大写形式，函数就会在特征集中加入这一特征，并取名为“UPPERCASE”。与认为“大写单词代表着某种特殊情况”的规则不同，这只是一个附加的特征，分类器可以利用该特征来进行训练——而在某些场合下，分类器也可能会认为这一特征对于区分文档分类而言是完全没有用处的。

如果希望将这一新函数与 filterfeed 结合使用，我们就须要修改代码，将内容项而非全文作为参数传给分类器。请将函数的末尾处修改如下：

```
# 将当前分类的最佳推测结果打印输出
print 'Guess: '+str(classifier.classify(entry))

# 请求用户给出正确分类，并据此进行训练
cl=raw_input('Enter category: ')
classifier.train(entry,cl)
```

随后，我们就可以初始化分类器，并将 entryfeatures 用作特征提取函数了：

```
>>> reload(feedfilter)
<module 'feedfilter' from 'feedfilter.py'>
>>> cl=docclass.fisherclassifier(feedfilter.entryfeatures)
>>> cl.setdb('python_feed.db')  # 仅当你使用的是DB版的代码
>>> feedfilter.read('python_search.xml',cl)
```

关于特征，我们还有许多工作可做。前文构造的基本框架允许我们定义自己的特征提取函数，并设置分类器令其使用该函数。分类器将对任何传入的对象进行分类，只要我们指定的特征提取函数能够根据对象返回一组特征即可。

使用 Akismet

Using Akismet

Akismet 与本章介绍的有关文本分类算法的研究稍有些偏离，不过对于特定类型的应用而言，使用 Akismet 可以花费最小的代价满足你对垃圾信息过滤的需求，同时也免去了自己构造分类器的需要。

Akismet 是作为 WordPress 的一个插件发展而来的，它允许人们向其报告提交到各自博客上的垃圾评论，并与他人报告的垃圾评论进行相似度对比，对新提交的评论进行过滤。目前这些 API 是开放的，因此我们可以向 Akismet 发起任何字符串查询请求，以获知 Akismet 是否认为该字符串属于垃圾信息。

我们要做的第一件事情是获得一个 Akismet 的 API 密钥，可以从 *http://akismet.com* 获取到该密钥。这些密钥对于个人用途而言是免费的，此外还有一些针对商业用途的密钥可供选择。 Akismet API 是通过常规的 HTTP 请求进行调用的，相应的函数库已经被写成了各种不同的语言。本节中用到的函数库可以从 *http://kemayo.wordpress.com/2005/12/02/akismet-py* 处下载到。请下载 *akismet.py*，并将其与你的代码放入同一目录下，或者也可以将其放入 Python 库所在的目录下。

API 的用法非常简单。请新建一个名为 *akismettest.py* 的文件，并加入下列函数：

```
import akismet

defaultkey = "YOURKEYHERE"
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 'Invalid key'
      return False
  except akismet.AkismetError, e:
      print e.response, e.statuscode
      return False
```

现在，我们已经拥有了一个可以接受任何字符串的可供调用的方法，我们可以调用该函数来判断传入的字符串是否与博客评论中的内容相类似。请在你的 Python 会话中尝试一下：

```
>>> import akismettest
>>> msg='Make money fast! Online Casino!'
>>> akismettest.isspam(msg,'spammer@spam.com','127.0.0.1')
True
```

请以不同的用户名、代理和 IP 地址进行试验，观察结果如何变化。

由于 Akismet 的主要用途是对提交到博客上的垃圾评论进行判断，因此它也许并不适合处理其他类型的文档，如电子邮件。而且，与前述分类器不同的是，它不允许你对传入的参数做任何的调整，我们也无法洞悉其求解答案的具体计算过程。不过，Akismet 对于垃圾评论的过滤而言还是非常准确的，而且假如我们的应用正在不断地遭受到相似种类的垃圾信息的骚扰，那么 Akismet 是值得一试的，因为与我们可能搜集到的数据量相比，Akismet 拥有一个相当巨大的对比用文档集。

替代方法

Alternative Methods

本章中介绍的两个分类器都是**监督型学习方法**（supervised learning methods）的例子，这是一种利用正确结果接受训练并逐步作出更准确预测的方法。第 4 章中介绍过的用于对搜索结果进行排名的人工神经网络是另一个监督型学习的例子。通过将特征作为输入，并令输出代表每一种可能的分类，我们也可以将神经网络用于本章中的相同问题。同样地，第 9 章中介绍的**支持向量机**（support vector machines），也可以用于解决本章中的问题。

贝叶斯分类器之所以经常被用于文档分类的原因是，与其他方法相比它所要求的计算资源更少。一封电子邮件可能包含数百甚至数千个单词，与训练相应规模大小的神经网络相比，简单地更新一下计数值所需占用的内存资源和处理器时钟周期会更少。而且正如你所看到的，这些工作完全可以在一个数据库中完成。神经网络是否会成为一种可行的替代方案，取决于训练和查询所要求的速度，以及实际运行的环境。神经网络的复杂性导致了其在理解上的困难。在本章中，我们可以清楚地看到单词的概率，以及它们对最终分值的实际贡献有多大，而对于网络中两个神经元之间的连接强度而言，则并不存在同样简单的解释。

另一方面，与本章中所介绍的分类器相比，神经网络和支持向量机有一个很大的优势：它们可以捕捉到输入特征之间更为复杂的关系。在贝叶斯分类器中，每个特征都有一个针对各分类的概率值，将这些概率组合起来之后就得到了一个整体上概率值。在神经网络中，某个特征的概率可能会依据其他特征的存在或缺失而改变。也许你正在试图阻止有关在线赌博的垃圾信息，但是又对跑马很感兴趣，在这种情况下，只有当电子邮件中的其他地方没有出现单词“horse”时，单词 “casino”才被认为是“bad”的。朴素贝叶斯分类器无法捕获这样的相互依赖性，而神经网络却是可以的。

练习

Exercises

1. **改变假设概率** 请修改 `classifier` 类，使其能够支持针对不同特征的不同假设概率。修改 `init` 方法，使其能够接受其他分类器作为参数，并从一个更合理的假设概率推测值（而不是 0.5）开始。

2. **计算 *Pr(Document)*** 在朴素贝叶斯分类器中，*Pr(Document)*的计算被略过了，因为它对于比较概率值而言并不是必需的。在特征彼此独立的前提下，事实上利用 *Pr(Document)* 来计算整体概率值是可行的。应该如何计算 *Pr(Document)*呢？

3. **POP-3 电子邮件过滤器** Python 有一个用于下载电子邮件的库，叫做 *poplib*。请编写一段脚本，从服务器下载电子邮件，并尝试对其进行分类。一封电子邮件包含有哪些不同的属性？你将如何利用这些属性来构建特征提取函数呢？

4. **任意长度的短语** 本章为你示范了提取词组和单个单词的方法。请修改代码令特征提取过程变得可配置，使其能够一次提取出一组拥有指定数量的单词，并将之作为一个独立的特征。

5. **保留 IP 地址**　IP 地址、电话号码，以及其他数字信息可能有助于对垃圾信息的识别。请修改特征提取函数，使其将这些信息作为特征加以返回（IP 地址中包含有句号，但是你依然须要剔除句子间的句号）。

6. **其他虚拟特征**　有许多像 UPPERCASE 那样的虚拟特征，这些特征可能对文档分类很有帮助。篇幅过长的文档或长单词占据优势的情况也有可能是一种线索。请将这些情况也作为特征。你还能想到其他情况吗？

7. **神经网络分类器**　请修改第 4 章中的神经网络，利用它对文档进行分类。如何对神经网络的输出结果进行比较？请编写一个程序对文档进行分类，并对其进行上千次的训练。记录每一种算法执行所需的时间。如何对这些算法作出对比呢？

第 7 章

决策树建模

Modeling with Decision Trees

到目前为止，我们已经掌握了几种不同的自动分类器算法，本章我们将对此做进一步延伸，介绍一种非常有用的算法，叫做**决策树学习**。不同于其他大多数分类器，由决策树产生的模型具有易于解释的特点——贝叶斯分类器中的数字列表会告诉我们每个单词的重要程度，但是你必须经过计算才能够确知结果到底如何。理解神经网络的难度则更大，因为位于两个神经元之间的连接上的权重值本身并没有什么实际意义。而对于决策树，我们只须要通过观察就可以理解其推导的过程，我们甚至还可以将其转换成一系列简单的 if-then 语句。

本章将给出三个使用决策树的不同例子。在第一个例子中，我们向大家示范了如何预测一个网站上有多少用户有可能会愿意为了某些高级功能而支付费用。许多在线应用都是以会员订阅的方式或是在依用付费的基础上（on a per-use basis）进行定价的，它们为用户提供了一种付费之前先试用的方式。以会员订阅的方式为例，网站通常会提供有时间限制的免费试用版本，或者是功能受限的免费版本。采用依用付费定价方式的站点则有可能会为用户提供一个免费的会话连接（free session），或者是诸如此类的东西。

本章稍后介绍的其他几个例子，将利用决策树来为住房价格及来自 Hot or Not 网站的“热度（hotness）”评价进行建模。

预测注册用户

Predicting Sign-Ups

有时，当拥有很大访问量（high-traffic）的站点推出新的应用，并提供免费账号和会员账号时，这样的网站应用将会吸引数以千计的新用户。这些用户里有许多人都是受好奇心的驱使，而非真的在寻找某种特定类型的应用，因此他们成为付费用户的可能性是非常小的。这种情况令我们很难识别和跟踪潜在的网站用户，于是许多站点就采取了向所有注册用户群发邮件的方式，而没有采用更有针对性的方法。

为了解决上述问题，假如我们能够预测出一位用户成为付费顾客的可能性有多大，那将是一项非常有价值的工作。到目前为止，我们已经知道了，可以利用贝叶斯分类器或神经网络来完成这一功能。然而在这里，我们所要强调的是算法的清晰直观——如果我们知道有哪些因素可以表明用户将会成为付费顾客，那么就可以利用这些信息来指导我们的广告策略制定工作，让网站的某些功能具有更好的可用性，或者采取其他能够有效增加付费顾客数量的策略。

此处，假设我们有一个提供免费试用的在线应用。用户为了获得试用的机会而注册了账号，待使用了若干天之后，他们可以选择向基本服务或高级服务升级。因为用户为了免费试用须要注册账号，所以我们可以借此将用户的相关信息收集起来，并且在试用结束的时候，网站的所有者会记录下哪些用户选择了成为付费客户。

为了尽量减少用户的工作量，使其能够尽快地注册账号，网站不会过多地询问用户的个人信息，相反，它会从服务器的日志中收集这些信息，比如：用户来自哪个网站，所在的地理位置，以及他们在注册之前曾经浏览过多少网页，等等。假设我们收集到了这些数据，并且将它们填入一张表格中，其结果可能如表 7-1 所示。

表 7-1：针对某个 Web 站点的用户行为及其最终购买决策

来源网站	位置	是否阅读过 FAQ	浏览网页数	选择服务类型
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
(直接)	New Zealand	No	12	None
(直接)	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
Google	UK	Yes	18	Basic
Kiwitobes	France	Yes	19	Basic

我们将上述信息整理到一个由一行行数据组成的列表里，列表中的每一行都是由上述表格中各栏数据构成的一个序列。其中的最后一栏代表了用户是否已经注册；而这个“服务”栏，正是我们希望预测的内容。请新建一个名为 *treepredict.py* 的文件，我们会在本章的后续部分一直使用该文件。如果你想手工输入数据，请将下列代码加入文件的首部：

```
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'],
         ['google','UK','yes',18,'Basic'],
         ['kiwitobes','France','yes',19,'Basic']]
```

如果你希望下载事先准备好的数据集，那么也可以访问 *http://kiwitobes.com/tree/decision_tree_example.txt*。

为了将数据文件加载进来，请将下面这行代码加入 *treepredict.py* 的首部：

```
my_data=[line.split('\t') for line in file('decision_tree_example.txt')]
```

现在，我们已经掌握了用户相关的信息，包括：用户所在的位置，他们是通过哪些网站访问到这里的，以及他们在注册之前在这个网站上花费了多少时间；我们只须要找到一种方法，能够将一个合理的推测值填入“服务”栏即可。

引入决策树

Introducing Decision Trees

相比于其他方法，**决策树**是一种更为简单的机器学习方法。它是对被观测数据（observations）进行分类的一种相当直观的方法，决策树在经过训练之后，看起来就像是以树状形式排列的一系列 if-then 语句。图 7-1 展示了一个利用决策树对水果进行分类的例子。

一旦我们有了决策树，据此进行决策的过程就变得非常容易理解了。只要沿着树的路径一直向下，正确回答每一个问题，最终就会得到答案。沿着最终的叶节点向上回溯，我们就会得到一个有关最终分类结果的推理过程。

本章我们将着手考查一种决策树的表示法，我们会编写代码，利用真实数据来构造决策树，并对新遇到的观测数据进行分类。首先，我们来构造决策树的表达形式。请新建一个类，取名为 `decisionnode`，它代表树上的每一个节点：

```
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
```

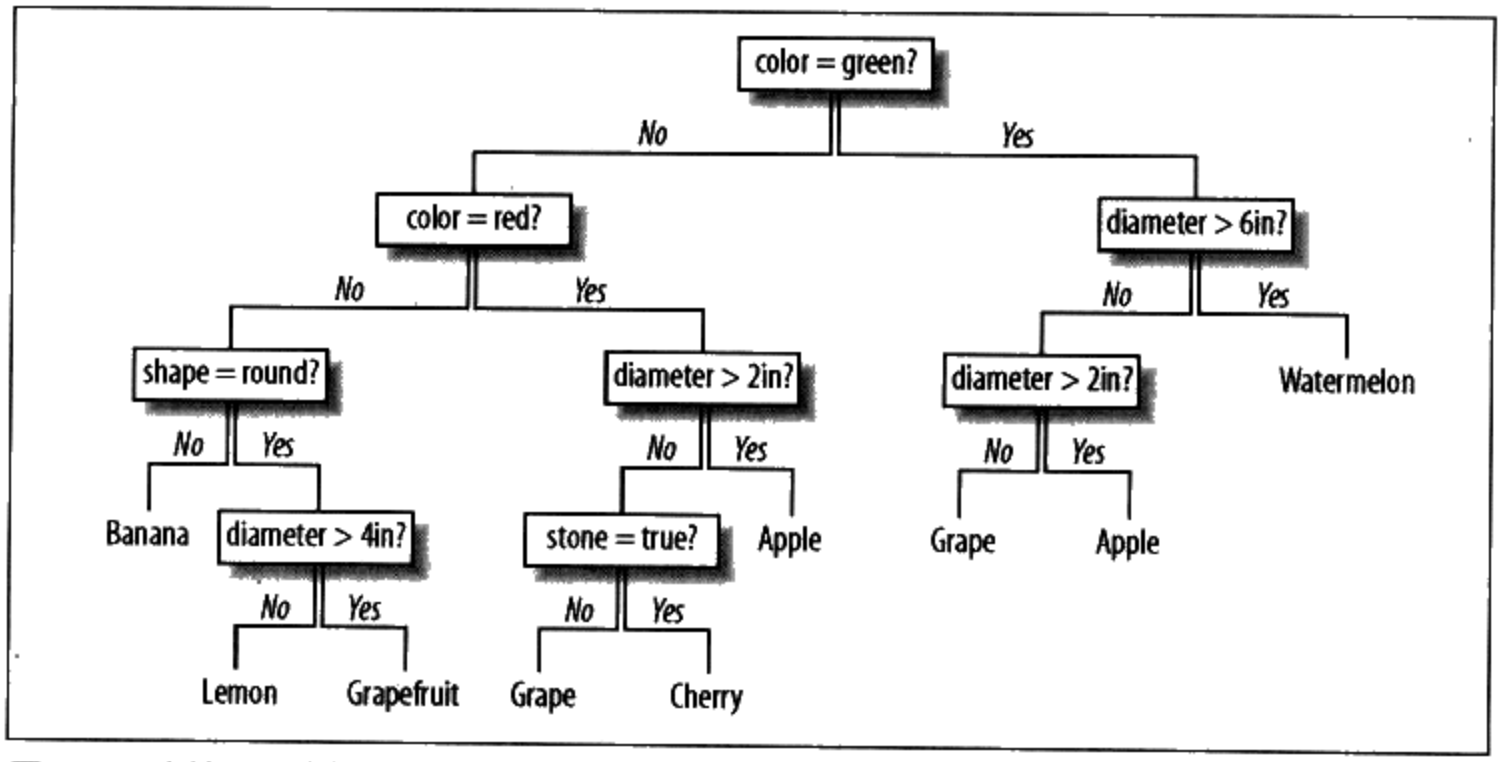


图 7-1：决策树示例

每一个节点都有 5 个实例变量，这 5 个变量都是在 initializer 中设置的。

- `col` 是待检验的判断条件（the criteria to be tested）所对应的列索引值。
- `value` 对应于为了使结果为 true，当前列必须匹配的值。
- `tb` 和 `fb` 也是 `decisionnode`，它们对应于结果分别为 true 或 false 时，树上相对于当前节点的子树上的节点。
- `results` 保存的是针对于当前分支的结果，它是一个字典。除叶节点外，在其他节点上该值都为 `None`。

构造决策树的函数将会返回一个根节点，我们可以沿着它的 True 分支或 False 分支一直遍历下去，直至到达最终结果为止。

对树进行训练

Training the Tree

本章我们将使用一种叫做 *CART*（Classification and Regression Trees 的缩写，即分类回归树）的算法。为了构造决策树，算法首先创建一个根节点。然后通过评估表中的所有观测变量，从中选出最合适的变量对数据进行拆分。为此，算法考查了所有不同的变量，然后从中选出一个条件（比如："用户是否读过了 FAQ？"）对结果数据进行分解，以使我们能更容易地推测出用户的意图来（用户会因哪一项服务而注册账号）。

函数 `divideset` 的作用是根据列表中某一栏的数据将列表拆分成两个数据集。该函数接受一个列表，一个指示表中列所在位置的数字，和一个用以对列进行拆分的参考值作为参数。以"Read FAQ"为例，可能的取值有 Yes 或 No，而对于"Referrer"而言，则会有很多可能的取值。随后，算法会返回两个列表：第一个列表所包含的数据行，其指定列中的数据将会与我们先前指定的参考值相匹配，而第二个列表，则包含了与参考值不相匹配的剩余数据行。

```
# 在某一列上对数据集合进行拆分，能够处理数值型数据或名词性数据。
def divideset(rows,column,value):
   # 定义一个函数，令其告诉我们数据行属于第一组(返回值为 true)还是第二组(返回值为 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

   # 将数据集拆分成两个集合，并返回
   set1=[row for row in rows if split_function(row)]
   set2=[row for row in rows if not split_function(row)]
   return (set1,set2)
```

上述代码创建了一个名为 split_function 的函数，该函数根据数据集的类型（是否是数值型），对其进行拆分。如果数据是数值型的，split_function 函数就会根据“true”的判断条件，判断指定列中的数值是否大于参考值。如果数据不是数值型的，则函数只会判断指定列中的数值是否与参考值相等。我们利用该函数将数据拆分成了两个集合，其中一个是 split_function 函数返回 true 时的集合，另一个则是返回 false 时的集合。

请启动 Python 会话，尝试按“Read FAQ”列对结果进行拆分：

```
$ python
>>> import treepredict
>>> treepredict.divideset(treepredict.my_data,2,'yes')
([['slashdot', 'USA', 'yes', 18, 'None'], ['google', 'France', 'yes', 23,
'Premium'],...]]
[['google the ', 'UK', 'no', 21, 'Premium'], ['(direct)', 'New Zealand', 'no',
12, 'None'],...])
```

表 7-2 给出了拆分的结果

表 7-2：基于“Read FAQ”列的拆分结果

True	False
None	Premium
Premium	None
Basic	Basic
Basic	Premium
None	None
Basic	None
Basic	None

目前看来，拆分结果所选用的变量并不是很理想，因为两边似乎都混杂了各种情况。我们需要一种方法来选择最合适的变量。

选择最合适的拆分方案

Choosing the Best Split

对于前述的观测数据而言，我们所选择的变量不是非常的理想。这一点也许并无大碍，但是从软件解决方案的角度而言，为了选择合适的变量，我们需要一种方法来衡量数据集合中各种因素的混合情况。我们所要做的，就是找出合适的变量，使得生成的两个数据集合在混杂程度上能够尽可能小。首先，我们需要一个函数来对数据集合中的每一项结果进行计数。请将下列函数加入 *treepredict.py* 中：

```
# 对各种可能的结果进行计数（每一行数据的最后一列记录了这一计数结果）
def uniquecounts(rows):
   results={}
   for row in rows:
      # 计数结果在最后一列
      r=row[len(row)-1]
      if r not in results: results[r]=0
      results[r]+=1
   return results
```

函数 uniquecounts 的作用是找出所有不同的可能结果，并返回一个字典，其中包含了每一项结果的出现次数。其他函数将利用该函数来计算数据集合的混杂程度。对于混杂程度的测度，有几种不同的度量方式可供选择，此处我们将考查其中的两种：基尼不纯度（Gini impurity）和熵（entropy）。

基尼不纯度

Gini Impurity

基尼不纯度，是指将来自集合中的某种结果随机应用于集合中某一数据项的预期误差率。如果集合中的每个数据项都属于同一分类，那么推测结果总会是正确的，因而此时的误差率为 0。如果有 4 种可能的结果均匀地分布在集合内，则推测有 75%的可能是不正确的，因而此时的误差率为 0.75。

基尼不纯度的计算函数如下所示：

```
# 随机放置的数据项出现于错误分类中的概率
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
```

该函数利用集合中每一项结果出现的次数除以集合的总行数来计算相应的概率，然后将所有这些概率值的乘积累加起来。这样就会得到某一行数据被随机分配到错误结果的总概率。

这一概率的值越高，就说明对数据的拆分越不理想。概率值为 0 则代表拆分的结果非常理想，因为这说明了每一行数据都已经被分配到了正确的集合中。

熵

Entropy

在信息理论中，**熵**代表的是集合的无序程度——基本上就相当于我们在此处所说的集合的混杂程度。请将下列函数加入 *treepredict.py* 中：

```
# 熵是遍历所有可能的结果之后所得到的 p(x)log(p(x))之和
def entropy(rows):
   from math import log
   log2=lambda x:log(x)/log(2)
   results=uniquecounts(rows)
   # 此处开始计算熵的值
   ent=0.0
   for r in results.keys():
      p=float(results[r])/len(rows)
      ent=ent-p*log2(p)
   return ent
```

函数 entropy 计算了每一项数据出现的频率（即数据项出现的次数除以集合的总行数），并使用了如下公式：

p(i) = frequency(outcome) = count(outcome) / count(total rows)

Entropy = 针对所有结果的 p(i) x log(p(i))之和

这是一种衡量结果之间差异程度的测度方法。如果所有结果都相同（比如说，如果我们够幸运的话，所有人最终都成为了付费订户），则熵为 0。群组（groups）越是混乱，相应的熵就越高。我们之所以要将数据拆分成两个新的组，其目的就是要降低熵。

请在你的 Python 会话中分别尝试一下基尼不纯度和熵这两种度量方法：

```
>>> 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
```

熵和基尼不纯度之间的主要区别在于，熵达到峰值的过程要相对慢一些。因此，熵对于混乱集合的“判罚”往往要更重一些。由于人们对熵的使用更为普遍，因此本章后续部分将选择熵作为度量标准，不过我们如果要想切换到基尼不纯度也是非常容易的。

以递归方式构造树

Recursive Tree Building

为了弄明白一个属性的好坏程度，我们的算法首先求出整个群组的熵，然后尝试利用每个属性的可能取值对群组进行拆分，并求出两个新群组的熵。为了确定哪个属性最适合用来拆分，算法会计算相应的**信息增益**（Information gain）。所谓信息增益，是指当前熵与两个新群组经加权平均后的熵之间的差值。算法会针对每个属性计算相应的信息增益，然后从中选出信息增益最大的属性。

待根节点处的判断条件确定之后，算法会根据该条件返回的 true 或 false，分别建立两个分支，如图 7-2 所示。

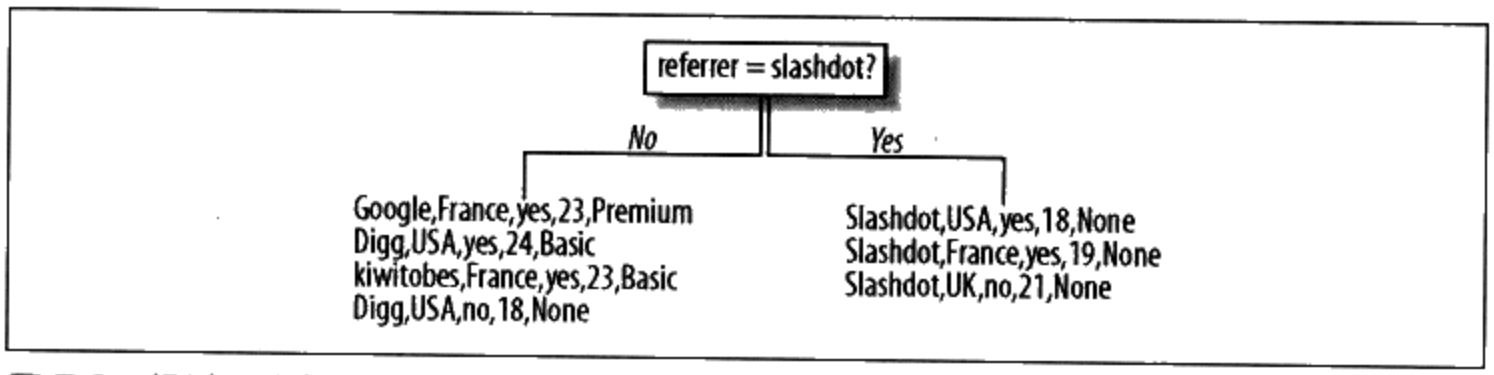


图 7-2：经过一次拆分之后的决策树

算法将观测数据拆分成了两个组，其中一组符合判断条件，另一组则与判断条件不符。对于每个分支，算法随后会判断是否要对其做进一步的拆分，或者我们已经获得了一个明确的结论而无须再行拆分了。如果某个新分支可以被继续拆分，算法就会使用与上面同样的方法来确定接下来到底应该选用哪一个变量。第二次拆分的情况如图 7-3 所示。

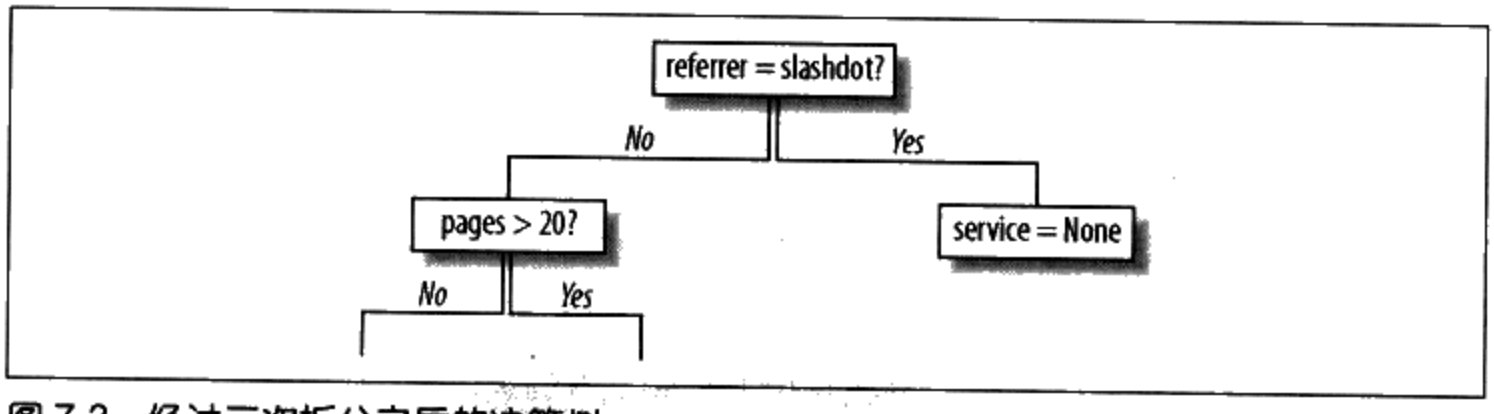


图 7-3：经过二次拆分之后的决策树

通过计算每个新生节点的最佳拆分属性，对分支的拆分过程和树的构造过程会不断地持续下去。当拆分某个节点所得的信息增益不大于 0 的时候，对分支的拆分才会停止。

请在 *treepredict.py* 中新建一个函数 `buildtree`。这是一个递归函数，它通过为当前数据集选择最合适的拆分条件来实现决策树的构造过程：

```
def buildtree(rows,scoref=entropy):
  if len(rows)==0: return decisionnode()
  current_score=scoref(rows)

  # 定义一些变量以记录最佳拆分条件
  best_gain=0.0
  best_criteria=None
  best_sets=None

  column_count=len(rows[0])-1
  for col in range(0,column_count):
    # 在当前列中生成一个由不同值构成的序列
    column_values={}
    for row in rows:
       column_values[row[col]]=1
    # 接下来根据这一列中的每个值，尝试对数据集进行拆分
    for value in column_values.keys():
      (set1,set2)=divideset(rows,col,value)

      # 信息增益
      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)
  # 创建子分支
  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))
```

上述函数首先接受一个由数据行构成的列表作为参数。它遍历了数据集中的每一列（最后一列除外，因为那是用来存放最终结果的），针对各列查找每一种可能的取值，并将数据集拆分成两个新的子集。通过将每个子集的熵乘以子集中所含数据项在原数据集中所占的比重（fraction），函数求出了每一对新生成子集的加权平均熵，并记录下熵值最低的那一对子集。

如果由熵值最低的一对子集求得的加权平均熵比当前集合的熵要大，则拆分过程就结束了，针对各种可能结果的计数所得将会被保存起来。否则，算法就会在新生成的子集上继续调用 `buildtree` 函数，并把调用所得的结果添加到树上。我们把针对每个子集的调用结果分别附加到节点的 True 分支和 False 分支上，最终整棵树就这样构造出来了。

现在，我们可以将算法最终应用到整个原始数据集上了。上述代码足够灵活，它可以同时处理文本型数据和数值型数据。代码还假定了数据集的最后一行（译注 1）对应于目标值，因此我们只要简单地将数据集传进去，就可以构造出决策树来：

```
>>> reload(treepredict)
<module 'treepredict' from 'treepredict.py'>
>>> tree=treepredict.buildtree(treepredict.my_data)
```

现在，变量 tree 中保存着一个经过训练的决策树。稍后我们将会学习如何对树进行浏览，以及如何借此来做出预测。

决策树的显示

Displaying the Tree

既然我们已经得到了一棵决策树，下一步应该如何处置它呢？或许有一件事你一定会想到要做的，那就是树的浏览。下面的 printtree 是一个以纯文本方式显示树的简单函数。虽然输出结果不是很美观，但是对于显示节点不太多的树而言，这不失为一种简单的办法：

```
def printtree(tree,indent=''):
  # 这是一个叶节点吗?
  if tree.results!=None:
    print str(tree.results)
  else:
    # 打印判断条件
    print str(tree.col)+':'+str(tree.value)+'? '

    # 打印分支
    print indent+'T->',
    printtree(tree.tb,indent+'  ')
    print indent+'F->',
    printtree(tree.fb,indent+'  ')
```

这又是一个递归函数。它接受 buildtree 返回的树作为参数，然后沿着树向下遍历，当函数到达一个包含结果信息的节点时，它就知道已经达到了一个分支的末端。此时，它就会打印出针对 True 分支和 False 分支的判断条件，并针对每个分支递归调用 printtree。每调用一次，缩排字符串就增加一格。

请针对我们前面刚刚构造好的树，调用此函数，我们将得到如下结果：

```
>>> 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}
```

译注 1：此处原文的意思是最后一行，但根据上下文译者认为是指最后一列。

这是决策树在尝试生成新的分类时执行推理过程的一个可视化表达。位于根节点处的判断条件是“Google 在第 0 列吗？”。如果这一条件满足，那么算法就会走 T->分支，并判断任何通过 Google 来到这里的使用者，如果他们浏览的网页个数已经达到或超过了 21 个，则将成为付费订户。如果条件不满足，那么算法就会跳到 F->分支，并对条件“Slashdot 在第 0 列吗？”进行评估。这一过程会一直持续下去，直到算法到达一个包含最终结果的分支为止。正如我们先前提到过的，能够直观地看到隐藏在推理过程背后的逻辑，是决策树的一大优势。

图形显示方式

Graphical Display

文本显示方式对于节点不太多的树而言是可行的，但是随着树的规模逐渐变大，以这样的可视化形式来跟踪我们在树上所走的路径可能是非常困难的。此处，我们将会看到树的一种图形化表现形式，对于浏览本章后续部分将要构造的决策树而言这种方式将会是非常有用的。

绘制树的代码与第 3 章中绘制树状图（dendrograms）的代码是类似的。两者都涉及了绘制具有任意深度节点的二叉树，因此我们首先须要编写函数来确定，一个给定节点要占据多少空间——包括所有子节点的总宽度，以及节点所要到达的深度值，后者告诉我们，为了能够容纳所有分支，节点在垂直方向上所需要的空间。一个分支的总宽度等于其所有子分支的宽度之和，而如果节点没有子分支的话，则对应宽度为 1：

```
def getwidth(tree):
  if tree.tb==None and tree.fb==None: return 1
  return getwidth(tree.tb)+getwidth(tree.fb)
```

一个分支的深度等于其最长子分支的总深度值加 1：

```
def getdepth(tree):
  if tree.tb==None and tree.fb==None: return 0
  return max(getdepth(tree.tb),getdepth(tree.fb))+1
```

为了将树真正绘制出来，我们还须要安装 Python Imaging Library。可以从 *http://pythonware.com* 处下载到该库，附录 A 包含有关于安装该库的更多信息。请将下列 `import` 语句添加到 *treepredict.py* 文件的首部：

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

函数 `drawtree` 为待绘制的树确定出一个合理的尺寸，并设置好画布（canvas）。然后将画布和树的根节点传递给 `drawnode`。请将该函数添加到 *treepredict.py* 中：

```
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')
```

函数 drawnode 实际用于绘制决策树的节点。它以递归的方式工作，首先绘制当前节点，并计算子节点的位置，然后在每个子节点上再次调用 drawnode。请将该函数添加到 *treepredict.py* 中：

```
def drawnode(draw,tree,x,y):
  if tree.results==None:
    # 得到每个分支的宽度
    w1=getwidth(tree.fb)*100
    w2=getwidth(tree.tb)*100

    # 确定此节点所要占据的总空间
    left=x-(w1+w2)/2
    right=x+(w1+w2)/2

    # 绘制判断条件字符串
    draw.text((x-20,y-10),str(tree.col)+':'+str(tree.value),(0,0,0))

    # 绘制到分支的连线
    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))

    # 绘制分支的节点
    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))
```

现在，我们可以尝试在自己的 Python 会话中将当前的树绘制出来了：

```
>>> reload(treepredict)
<module 'treepredict' from 'treepredict.pyc'>
>>> treepredict.drawtree(tree,jpeg='treeview.jpg')
```

上述命令的执行结果应该会生成一个名为 *treeview.jpg* 的新文件，如下页图 7-4 所示。

此处，代码并没有打印出 True 分支和 False 分支的标签，在更为复杂的图中，这些标签可能只会造成布局的混乱。由于在生成的图上，True 分支总是位于右侧，因此我们可以按图索骥，很容易地跟踪推断的过程。

对新的观测数据进行分类
Classifying New Observations

目前，我们还需要一个函数，接受新的观测数据作为参数，然后根据决策树对其进行分类。请将该函数添加到 *treepredict.py* 中：

```
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):
```

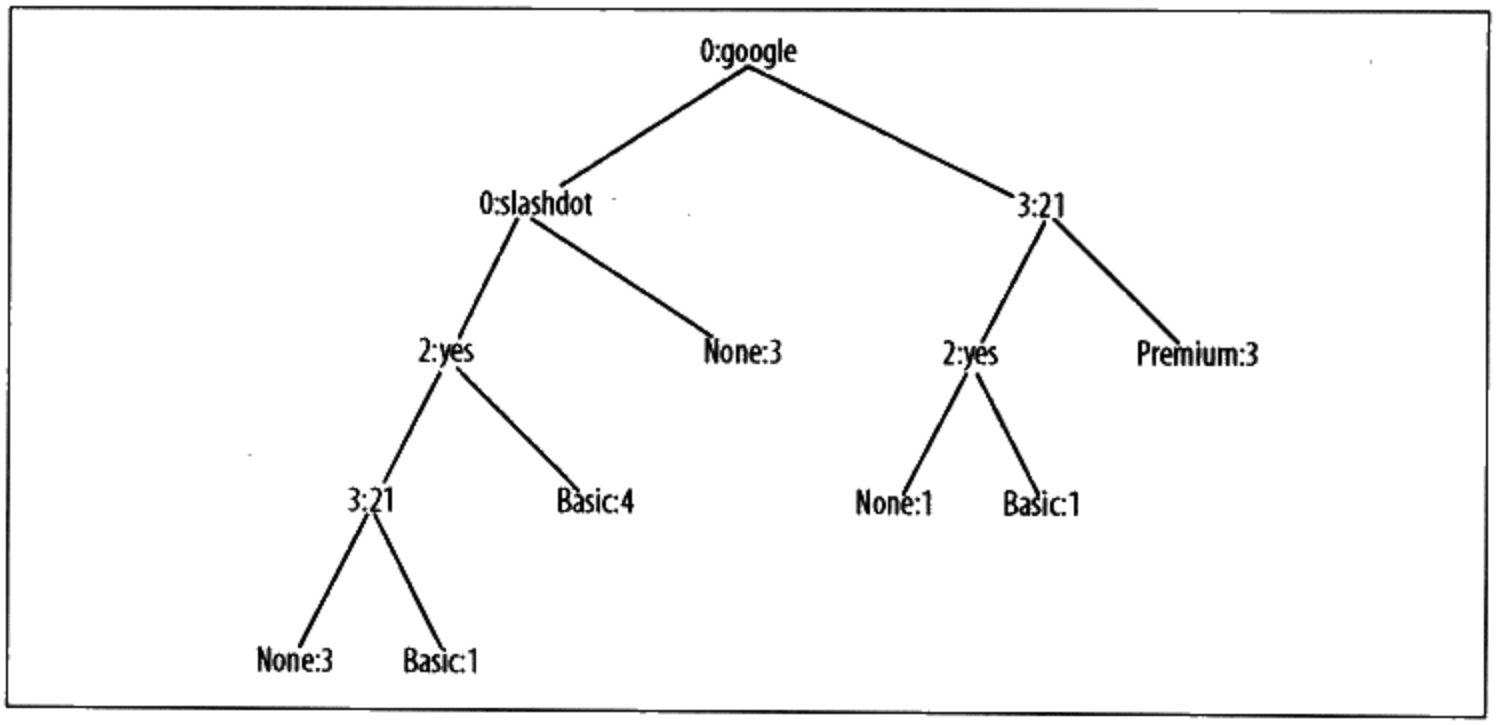


图 7-4：用于预测订户的决策树

```
    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)
```

该函数采用与 `printtree` 完全相同的方式对树进行遍历。在每次调用之后，函数会根据调用结果来判断是否到达分支的末端。如果尚未到达末端，它会对观测数据作出评估，以确认列数据是否与参考值相匹配。如果匹配，则会再次在 True 分支上调用 `classify`；如果不匹配，则会在 False 分支上调用 `classify`。

现在，我们可以调用 `classify` 函数，对新的观测数据进行预测了：

```
>>> reload(treepredict)
<module 'treepredict' from 'treepredict.pyc'>
>>> treepredict.classify(['(direct)','USA','yes',5],tree)
{'Basic': 4}
```

至此，我们已经拥有了能够从任何数据集中构造决策树的函数，显示和解释决策树的函数，以及对新的观测数据进行分类的函数。我们可以将这些函数应用到任何形式的数据集上，只要数据集是由多个数据行组成，并且每一行数据都包含一组观测变量和一个结果值即可。

决策树的剪枝

Pruning the Tree

利用前述方法来训练决策树会有一个问题，那就是决策树可能会变得**过度拟合**（overfitted）——也就是说，它可能会变得过于针对训练数据。专门针对训练集所创建出来的分支，其熵值与真实情况相比可能会有所降低，但决策树上的判断条件实际上是完全随意的，因此一棵过度拟合的决策树所给出的答案也许会比实际情况更具特殊性。

真实世界里的决策树

由于决策树具有易于解释的特点，因此它是商务分析、医疗决策和政策制定领域里应用最为广泛的数据挖掘方法之一。通常，决策树的构造是自动进行的，专家们可以利用生成的决策树来理解问题的某些关键因素，然后对其加以改进，以便更好地与他的观点相匹配。这一过程允许机器协助专家进行决策，并清晰地展示出推导的路径，从而我们可以据此来判断预测的质量。

如今，决策树以这样的形式被广泛运用于众多应用系统之中，其中就包括了顾客调查、金融风险分析、辅助诊断和交通预测。

因为前述算法直到无法再进一步降低熵的时候才会停止分支的创建过程，所以一种可能的解决办法是，只要当熵减少的数量小于某个最小值时，我们就停止分支的创建。这种策略时常被人们采用，但是它有一个小小的缺陷——我们有可能会遇到这样的数据集：某一次分支的创建并不会令熵降低多少，但是随后创建的分支却会使熵大幅降低。对此，一种替代的策略是，先构造好如前所述的整棵树，然后再尝试消除多余的节点。这个过程就是剪枝。

剪枝的过程就是对具有相同父节点的一组节点进行检查，判断如果将其合并，熵的增加量是否会小于某个指定的阈值。如果确实如此，则这些叶节点会被合并成一个单一的节点，合并后的新节点包含了所有可能的结果值。这种做法有助于避免过度拟合的情况，也使得根据决策树作出的预测结果，不至于比从数据集中得到的实际结论还要特殊。

请将用于剪枝的新函数添加到 *treepredict.py* 中：

```
def prune(tree,mingain):
  # 如果分支不是叶节点，则对其进行剪枝操作
  if tree.tb.results==None:
    prune(tree.tb,mingain)
  if tree.fb.results==None:
    prune(tree.fb,mingain)

  # 如果两个子分支都是叶节点，则判断它们是否须要合并
  if tree.tb.results!=None and tree.fb.results!=None:
    # 构造合并后的数据集
    tb,fb=[],[]
    for v,c in tree.tb.results.items():
      tb+=[[v]]*c
    for v,c in tree.fb.results.items():
      fb+=[[v]]*c

    # 检查熵的减少情况
    delta=entropy(tb+fb)-(entropy(tb)+entropy(fb)/2)
```

```
    if delta<mingain:
      # 合并分支
      tree.tb,tree.fb=None,None
      tree.results=uniquecounts(tb+fb)
```

当我们在根节点调用上述函数时，算法将沿着树的所有路径向下遍历到只包含叶节点的节点处。函数会将两个叶节点中的结果值组合起来形成一个新的列表，同时还会对熵进行测试。如果熵的变化小于 mingain 参数指定的值，则叶节点将会被删除，并且相应的结果值也将被全部移入父节点。随后，合并而成的新节点也可能成为删除对象，以及与其他节点的合并对象。

针对当前数据集，请尝试调用上述函数，看看是否有节点会被合并：

```
>>> 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}
```

在这个例子中，数据的拆分非常容易，因此根据一个合理的最小增益值（minimum gain）进行剪枝，实际上并没有多少工作量。只有当我们将最小增益值调得非常高的时候，某个叶节点才会被合并。正如我们稍后会看到的，现实中数据集的拆分往往不会像这个例子中那样的干脆利落，因此在那样的场合下，剪枝的效果往往会更加的明显。

处理缺失数据

Dealing with Missing Data

除了易于解释外，决策树还有一个优点，就是它处理缺失数据的能力。我们所使用的数据集也许会缺失某些信息——比如，在当前的例子中，用户的地理位置未必能够从其 IP 地址

中识别出来，所以这一字段也许会为空。为了使决策树能够处理这种情况，我们须要实现一个新的预测函数。

如果我们缺失了某些数据，而这些数据是确定分支走向所必需的，那么实际上我们可以选择两个分支都走。不过，此处我们不是平均地统计各分支对应的结果值，而是对其进行加权统计。在一棵基本的决策树中，所有节点都隐含有一个值为 1 的权重，即观测数据对于数据项是否属于某个特定分类的概率具有百分之百的影响。而如果要走多个分支的话，那么我们可以给每个分支赋以一个权重，其值等于所有位于该分支的其他数据行所占的比重。

实现上述功能的函数，`mdclassify`，是对 `classify` 的一个简单修改。请将它添加到 *treepredict.py* 中：

```
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():
        if k not in result: result[k]=0
        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)
```

`mdclassify` 与 `classify` 相比，唯一的区别在于末尾处：如果发现有重要数据缺失，则每个分支的对应结果值都会被计算一遍，并且最终的结果值会乘以它们各自的权重。

针对关键信息缺失的数据行，我们来尝试运行一下 `mdclassify`，看看最终的结果如何：

```
>>> 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}
```

正如我们所期望的那样，忽略 Pages 变量（即浏览网页数）的结果将导致 Premium 的概率偏大，而 Basic 的概率偏小。对 Read FAQ 变量的忽略则会导致另一种不同的分布。此处，每个概率最终都会乘以相应的权重值（即数据项位于各分支的比例）。

处理数值型结果
Dealing with Numerical Outcomes

用户行为的例子和水果树的例子都属于分类问题（因为最终的结果是分类而不是数字）。本章剩余的例子，住房价格和热度评价，都是处理数值型结果的问题。

有时，当我们在以数字作为输出结果的数据集上执行 `buildtree` 函数时，效果可能不一定非常的理想。如果我们将所有数字都看作是不同的分类，那么目前的算法将不会考虑这样一个事实：有些数字彼此非常的接近，而其他数字则相差很远；我们将这些数字完全看作成了绝对的离散。为了解决这个问题，当我们拥有一棵以数字作为输出结果的决策树时，我们可以使用方差（`variance`）作为评价函数来取代熵或基尼不纯度。请将函数 `variance` 加入到 *treepredict.py* 中：

```
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
```

该函数可以作为 `buildtree` 的一个参数，它的作用是计算一个数据集的统计方差。偏低的方差代表数字彼此都非常的接近，而偏高的方差则意味着数字分散得很开。当使用方差作为评价函数来构造决策树时，我们选择节点判断条件的依据就变成了：拆分之后令数字较大者位于树的一侧，数字较小者位于树的另一侧。以这种方式来拆分数据，就可以降低分支的整体方差。

对住房价格进行建模
Modeling Home Prices

决策树有许多潜在的应用，不过假如存在多个可能的观测变量，而且我们又对决策推导的过程很感兴趣的话，那么此时决策树的应用价值是最大的。在某些场合下，也许我们已经知道了最终的输出结果，而我们所关心的则是如何对输出结果进行建模，借此来理解背后的来龙去脉。决策树的一个潜在的非常受关注的应用领域是对商品价格的理解，尤其是针对那些拥有大量可测度的观测变量的商品价格。由于住房价格的变动非常大，况且又有许多数值型变量和名词性变量可以很容易地被测量到，因此本节我们将着手构造决策树，对房地产价格进行建模。

Zillow API

Zillow 是一个免费的 Web 服务，其作用是对房地产价格进行跟踪，并利用这些信息来评估住房的价格。Zillow 的工作原理是对样板住房（comps，即情况相似的住房）进行考查，并利用其价格信息来预测新的住房价格，而其所预测的价格则将接近于房产评估人员所给出的实际价格。图 7-5 给出了来自 Zillow 网站上的一个截屏片段，其中包含了房屋的有关信息及其评估值。

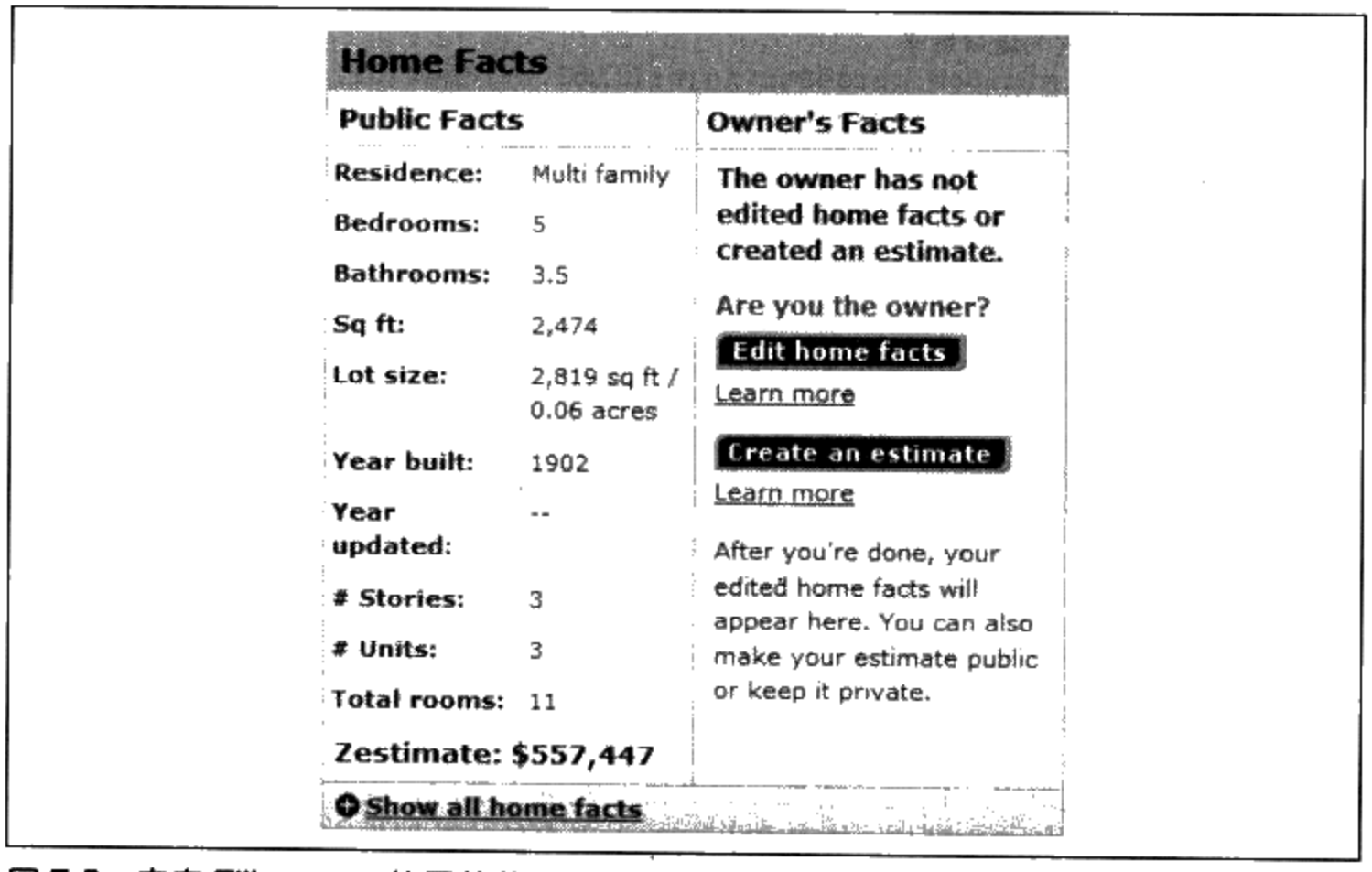


图 7-5：来自 Zillow.com 的屏幕截图

所幸的是，Zillow 还提供了一套 API，我们可以利用这套 API 获取到房屋的详细信息及其评估值。Zillow API 的网页地址为 *http://www.zillow.com/howto/api/APIOverview.htm*。

为了能够访问 API，我们须要得到一个开发者密钥，该密钥是免费的，并且可以从 Zillow 网站获得。这套 API 本身相当的简单——其使用方法是，构造一个包含全部搜索参数的 URL 并发起查询请求，然后对返回的 XML 结果进行解析，以得到诸如卧室数量和评估价格这样的详细查询结果。请新建一个名为 *zillow.py* 的文件并加入下列代码：

```
import xml.dom.minidom
import urllib2

zwskey="X1-ZWz1chwxis15aj_9skq6"
```

就如同我们在第 5 章中所做的那样，此处我们将采用 *minidom* API 对查询返回的 XML 结果进行解析。函数 `getaddressdata` 以地址和城市作为输入参数，并构造 URL 向 Zillow 查询

房产信息。函数会对查询返回的结果进行解析，并从中提取重要的信息，最后函数会将这些信息以多元组的形式返回。请将该函数加入 *zillow.py* 中：

```
def getaddressdata(address,city):
  escad=address.replace(' ','+')

  # 构造URL
  url='http://www.zillow.com/webservice/GetDeepSearchResults.htm?'
  url+='zws-id=%s&address=%s&citystatezip=%s' % (zwskey,escad,city)

  # 解析XML形式的返回结果
  doc=xml.dom.minidom.parseString(urllib2.urlopen(url).read())
  code=doc.getElementsByTagName('code')[0].firstChild.data

  # 状态码为0代表操作成功，否则代表有错误发生
  if code!='0': return None

  # 提取有关该房产的信息
  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)
```

由于该函数返回的元组中，代表“结果”的价格一栏位于最后，因此我们可以将该元组直接放入一个列表中作为观测数据。要使用此函数生成完整的数据集，我们还需要一个地址列表。你可以选择自己生成列表，也可以去 *http://kiwitobes.com/addresslist.txt* 下载一个随机生成的马塞诸塞州剑桥地区附近的地址列表。

请新建一个名为 `getpricelist` 的函数，以读取该文件并构造一个数据列表：

```
def getpricelist():
  l1=[]
  for line in file('addresslist.txt'):
    data=getaddressdata(line.strip(),'Cambridge,MA')
    l1.append(data)
  return l1
```

现在，我们可以利用上述函数生成数据集并构造决策树了。可以在你的 Python 会话中尝试执行下列命令：

```
>>> import zillow
>>> housedata=zillow.getpricelist()
>>> reload(treepredict)
>>> housetree=treepredict.buildtree(housedata,scoref=treepredict.variance)
>>> treepredict.drawtree(housetree,'housetree.jpg')
```

最终生成的文件，*housetree.jpg*，可能如图 7-6 所示。

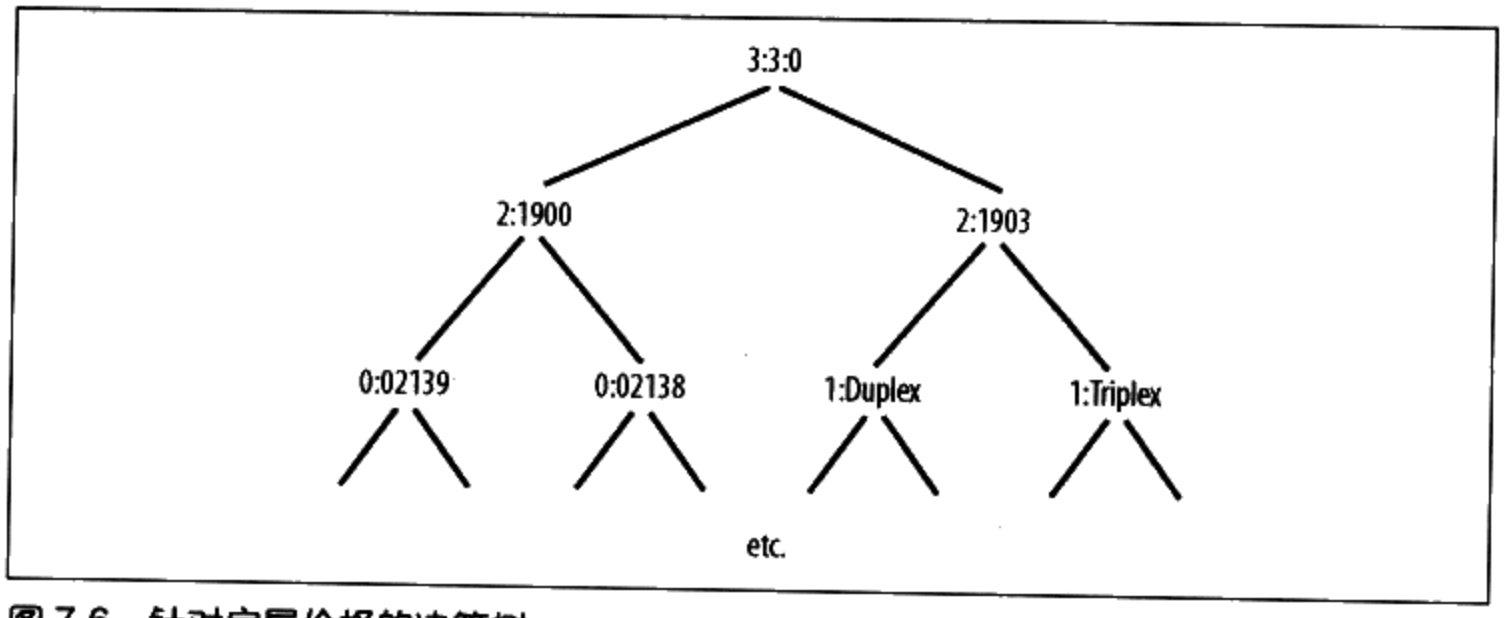


图 7-6：针对房屋价格的决策树

当然，假如我们只是对推测房产的价格感兴趣，那么只须调用 Zillow API 得到估价结果就可以了。此处值得注意的是，我们实际上已经为决定房屋价格所要考虑的因素建立起了一个模型。请注意，这棵树的根节点是浴室，这意味着为了最大限度的降低方差，我们是根据浴室的总量来拆分数据集的。在剑桥地区，住房价格的主要决定因素是看它是否有 3 个或更多的浴室（通常这表明该房产是一所供多户家庭居住的大房子）。

决策树的使用在此处有一个显著的缺陷：我们必须建立大量的价格数据才行，这是因为住房的价格千差万别，而且为了能够得出有价值的结论，我们须要以某种方式对其进行有效的分组。要找到一种针对实际价格数据的更加有效的预测技术并不是没有可能。第 8 章我们将讨论另一种价格预测的方法。

对“热度”评价进行建模
Modeling "Hotness"

Hot or Not 是一个允许用户上传自己照片的站点。网站的初衷是让用户能够根据他人的外表形象对其进行评价，然后网站会将评价结果综合起来，并为每个人设立一个介于 1 到 10 之间的评分。Hot or Not 后来逐渐演变成了一个约会交友的网站，现在它还提供了一套开放的 API，允许你获取网站成员的个人信息以及他们的“热度”评价情况。这使得 Hot or Not 成为实验决策树建模的一个很好的案例，因为它拥有一组输入变量和一个输出变量，还有一个可能颇为有趣的推理过程。Hot or Not 站点本身也是集体智慧的一个很好的应用案例。

为了访问 Hot or Not API，我们须要得到一个应用密钥。可以在 *http://dev.hotornot.com/signup* 上注册并获得一个相应的密钥。

Hot or Not API 与前述其他 API 的工作机理是完全一样的。我们只要向某个 URL 传送查询所需的参数，并对返回的 XML 结果进行解析就可以了。首先，请新建一个名为 *hotornot.py* 的文件，并将下列 import 语句以及对密钥的定义加入其中：

```
import urllib2
import xml.dom.minidom

api_key="479NUNJHETN"
```

接下来，是获取一个随机的人员列表，用于构造数据集。所幸的是，Hot or Not 提供了一个 API 调用，能够返回符合指定条件的人员列表。在本例中，唯一的查询条件就是有否“meet me”描述信息，因为我们只有从这些描述信息中才能获取到其他诸如家庭住址和兴趣爱好这样的信息。请将该函数添加到 *hotornot.py* 中：

```
def getrandomratings(c):
  # 为 getRandomProfile 构造 URL
  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')

  # 将 emids 和 ratings 组合到一个列表中
  result=[]
  for e,r in zip(emids,ratings):
    if r.firstChild!=None:
      result.append((e.firstChild.data,r.firstChild.data))
  return result
```

当我们生成了包含用户 ID 和评价信息的列表之后，还须要一个函数来下载人员信息——在本例中，包括性别、年龄、住址和关键字。将所有 50 个州都作为住址变量的取值范围有可能会导致过多的分支出现。为了减少住址的可能取值，我们不妨将各州划分成几个地区。请添加下列代码以指定地区信息：

```
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']}
```

Hot or Not API 提供了一个下载个人详细信息的方法调用，因此函数 getpeopledata 只须遍历第一遍搜索得到的所有结果，并依次调用 API 即可查询到人员的详细信息。请将这个函数添加到 *hotornot.py* 中：

```
def getpeopledata(ratings):
  result=[]
  for emid,rating in ratings:
    # 对应于 MeetMe.getProfile 方法调用的 URL

    url="http://services.hotornot.com/rest/?app_key=%s" % api_key
    url+="&method=MeetMe.getProfile&emid=%s&get_keywords=true" % emid

    # 得到所有关于此人的详细信息
    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]

      # 将州转换成地区
      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
```

现在，我们可以将上述模块导入到自己的 Python 会话中，生成数据集了：

```
>>> import hotornot
>>> l1=hotornot.getrandomratings(500)
>>> len(l1)
442
>>> pdata=hotornot.getpeopledata(l1)
>>> pdata[0]
(u'female', 28, 'West', 9)
```

上述列表中包含了每个用户的个人信息，最后一个字段是他们的评价情况。我们可以将这个数据结构直接传给 buildtree 方法来构造决策树：

```
>>> hottree=treepredict.buildtree(pdata,scoref=treepredict.variance)
>>> treepredict.prune(hottree,0.5)
>>> treepredict.drawtree(hottree,'hottree.jpg')
```

最终决策树的可能输出结果如下页图 7-7 所示。

图上的根节点对应于性别，据此我们得到了数据集的一个最佳拆分。树的剩余部分非常复杂而且难以阅读。不过，毫无疑问我们还是可以利用这棵树来对以前从未谋面的人进行预测的。另外，因为算法能够处理数据缺失的情况，所以我们还可以依据为数众多的变量对人员进行聚类。例如，也许我们想对比一下南部地区（South）与中大西洋地区（Mid-Atlantic），看看这两个地区人们的热度如何：

```
>>> south=treepredict2.mdclassify((None,None,'South'),hottree)
>>> midat=treepredict2.mdclassify((None,None,'Mid Atlantic'),hottree)
```

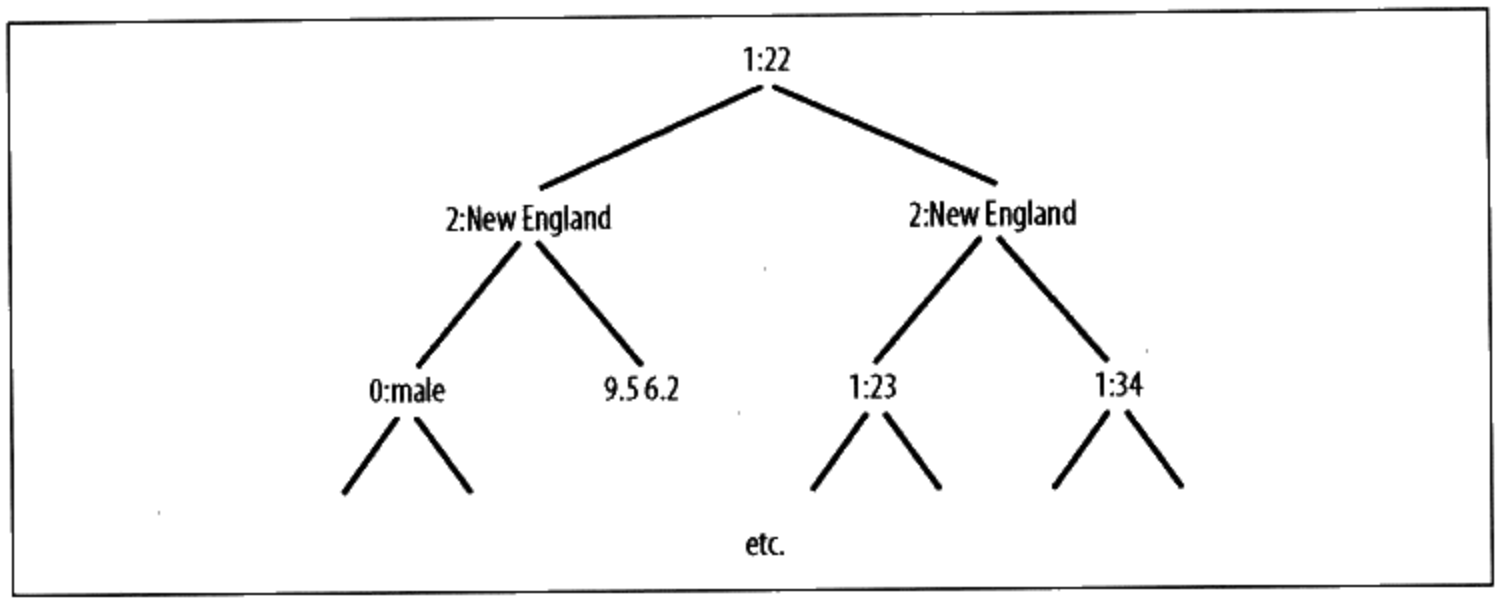


图 7-7：关于“热度”评价的决策树模型

```
>>> south[10]/sum(south.values())
0.0558208151832617 35
>>> midat[10]/sum(midat.values())
0.0489727973206008 64
```

从上述数据集中我们发现，南方地区的高“热度”者（super-hot people）相对而言要稍多一些。我们还可以尝试一下其他形式的预测，比如看一看年龄的分组情况，或者检验一下男士是否赢得了比女士更高的评价。

什么时候使用决策树

When to Use Decision Trees

或许决策树最大的优势就在于它可以轻易地对一个受训模型给予解释。在本章的例子里，执行完算法程序之后，我们不仅可以得到一棵用以预测新用户的决策树，而且还可以得到一个有助于我们做出判断的问题列表。从中我们可以发现，比如那些通过 Slashdot 找到该站点的用户从来都不会成为付费订户，而那些通过 Google 找到该站点并且至少浏览了 20 个网页的用户，则很有可能会成为付费订户。根据这一情况，我们也许就会对广告策略作出相应的调整，使其更加倾向于那些能够为我们带来更高访问流量的目标站点。除此以外，我们还发现了某些变量，比如用户的出生国籍，对于最终输出结果的确定并没有起到多大的作用。假如有些数据难以收集或收集的代价高昂，而且我们又知道这些数据无关痛痒时，那么我们完全可以不对它们进行收集。

与其他几种机器学习算法不同，决策树可以同时接受分类（categorical）数据和数值（numerical）数据作为输入。在本章的第一个例子中，我们的输入数据除了“浏览网页数”是数值数据外，其他几个都属于分类数据。不仅如此，许多算法在运行之前都要求我们必须对输入数据做预处理，或是归一化处理，而本章中的代码却可以接受包括分类数据和数值数据在内的任何数据列表，并据此构造出相应的决策树来。

决策树还允许数据的不确定性分配（译注 2）。由于种种原因，我们不一定总是能掌握足够的信息来做出正确的分类——在一棵决策树上也许会存在一部分节点，它们具有多种可能的结果值，但是又无法再进一步拆分。本章中的代码会返回一个字典对象，其中包含了针对不同结果的统计量，借助这一信息我们可以判断出结果的可信度。要知道，并不是所有算法都能够评估出一个不确定结果的概率来的。

不过，此处所使用的决策树算法的确还是有缺陷的。虽然对于只包含少数几种可能结果的问题而言，算法处理起来非常有效，但是当面对拥有大量可能结果的数据集时，算法就变得不那么有效了。在第一个例子中，仅有的输出结果包括了 none、basic 和 premium。而当输出结果有上百个的时候，决策树就会变得异常复杂，而且预测的效果也可能会大打折扣。

本章介绍的决策树还有另一个较大的缺陷，尽管它可以处理简单的数值型数据，但是它只能创建满足“大于/小于”条件的节点。对于某些数据集，当我们对其进行分类的时候，决定分类的因素往往取决于更多变量的复杂组合，此时要根据前述的决策树进行分类就比较困难了。例如，假设结果值是由两个变量的差来决定的，那么这棵树就会变得非常庞大，而且预测的准确性也会迅速下降。

总之，对于有大量数值型输入和输出的问题，决策树未必是一个好的选择；如果数值型输入之间存在许多错综复杂的关系，比如当我们进行金融数据分析或影像分析的时候，决策树同样也不一定是很好的选择。决策树最适合用来处理的，是那些带分界点（breakpoints）的、由大量分类数据和数值数据共同组成的数据集。如果对决策过程的理解至关重要，那么采用决策树就再合适不过了；就如同你已经看到的那样，明白推理过程有可能和知道最终的预测结果同样的重要。

练习

Exercises

1. **针对结果的概率**　目前，`classify` 和 `mdclassify` 函数都是以总计数值的形式给出最终结果的。请对它们进行修改，以给出最终结果属于某个分类的概率。

2. **缺失数据的范围**　`mdclassify` 允许我们使用“None”来表示一个值的缺失。对数值型数据而言，其结果未必是绝对未知的，也许相应的取值会落在某个已知的范围内。请修改 `mdclassify` 函数，允许使用一个如`(20,25)`这样的元组来代替原来的单一值，并且如果有必要的话，对两个分支都进行遍历。

3. **提早停止向下拆分**　不同于对决策树的剪枝，`buildtree` 可以在它到达某个节点，而该节点处对应熵的下降幅度又没有达到足够量的时候，停止继续向下拆分。有时这种做法未必能够达到理想的效果，但是它的确省去了额外的剪枝工作。请修改 `buildtree` 函数，令其接受一个代表最小增益的参数，一旦最小增益条件不满足，就停止继续向下拆分。

译注 2：即允许数据的缺失。

4. **数据有缺失的决策树构造**　我们编写的函数能够对一个有缺失的数据行进行分类，但是如果训练集中也有数据缺失的现象，那又该怎么办呢？请修改 `buildtree` 函数，令其检查数据是否有缺失的情况，并且当我们无法将结果沿某个分支向下传递的时候，令其同时沿两个分支向下传递。

5. **多路径拆分**（有难度）　本章中构造的所有树都是严格的二叉决策树。然而，有些数据集却允许我们可以将一个节点拆分成两个以上的分支，根据这些数据集构造出来的决策树也许会更加的简单。如果是这样的话，那么你将如何表达此类决策树？又如何对其加以训练呢？

第 8 章

构建价格模型

Building Price Models

迄今为止，我们已经考查过了一部分分类器，其中大多数都非常适合于对未知数据的所属分类进行预测。但是，在利用多种不同属性（比如价格）对数值型数据进行预测时，贝叶斯分类器、决策树，以及支持向量机（将在下一章中见到）都不是最佳的算法。本章我们将对一系列算法进行考查：这些算法可以接受训练，根据之前见过的样本数据作出数值类的预测，而且它们还可以显示出预测的概率分布情况，以帮助用户对预测过程加以解释。

在本章的后续部分，我们将考查如何利用这些算法来构造价格预测模型。经济学家认为，价格（尤其是拍卖价格）是一种利用集体智慧来决定商品真实价值的非常好的方法；在一个拥有众多买家和卖家的大型市场中，通常对于交易双方而言，商品的价格最终将会达到一个最优值。与此同时，对价格进行预测也是测试此类算法的一种很好的手段，因为在确定价格时，通常有许多不同的因素须要考虑。例如，当我们打算竞价一台膝上型电脑时，须要考虑处理器的速度、RAM 的容量、硬盘的大小、屏幕的分辨率，以及其他许多因素。

进行数值型预测的一项关键工作是确定哪些变量是重要的，以及如何将它们组合在一起。在膝上型电脑的例子中，可能有一些变量即便对价格会产生一定的影响，其影响也几乎是微乎其微的，例如：免费的附件或是某些捆绑销售的软件。而且与硬盘大小相比，可能屏幕尺寸对最终价格所产生的影响会更大一些。我们将利用第 5 章中介绍过的优化技术，自动确定各个变量的最佳权重。

构造一个样本数据集

Building a Sample Dataset

一个极富挑战性的测验数值型预测算法的数据集应该具备某些特征，这些特征的存在会使算法更加难以根据数据集作出预测。举例来说，如果你正打算购买电视机，那么很容易得出尺寸越大越好的结论，而这样的问题利用传统的统计技术来加以解决可能会更加容易一

些。正因为如此，所以我们去考查那些价格并非简单地按照商品尺寸或特征数量的增长而成比例增长的数据集，会更加地有意义。

本节中，我们将根据一个人为假设的简单模型来构造一个有关葡萄酒价格的数据集。酒的价格是根据酒的等级及其储藏的年代共同来决定的。该模型假设葡萄酒有“峰值年（peak age)”的现象，即：较之峰值年而言，年代稍早一些的酒的品质会比较好一些，而紧随其后的则品质稍差些。一瓶高等级的葡萄酒将从高价位开始，尔后价格逐渐走高直至其“峰值年”；而一瓶低等级的葡萄酒则会从一个低价位开始，价格一路走低。

为了对这一现象进行建模，我们新建一个名为 *numpredict.py* 的文件，并加入 `wineprice` 函数：

```
from random import random,randint
import math

def wineprice(rating,age):
  peak_age=rating-50

  # 根据等级来计算价格
  price=rating/2
  if age>peak_age:
    # 经过“峰值年”，后继 5 年里其品质将会变差
    price=price*(5-(age-peak_age))
  else:
    # 价格在接近“峰值年”时会增加到原值的 5 倍
    price=price*(5*((age+1)/peak_age))
  if price<0: price=0
  return price
```

我们还需要一个函数来构造表示葡萄酒价格的数据集。下列函数“生产”了 200 瓶葡萄酒，并根据模型求出了这些葡萄酒的价格。紧接着，函数还在原有价格的基础上随机地加减了 20%，以此来表现诸如税收和价格局部变动这样的情况，这同时也是为了增加数值型预测的难度。请将 `wineset1` 加入 *numpredict.py* 中：

```
def wineset1():
  rows=[]
  for i in range(300):
    # 随机生成年代和等级
    rating=random()*50+50
    age=random()*50

    # 得到一个参考价格
    price=wineprice(rating,age)

    # 增加“噪声”
    price*=(random()*0.4+0.8)

    # 加入数据集
    rows.append({'input':(rating,age),
                 'result':price})
  return rows
```

请开启一个 Python 会话，然后测试一下葡萄酒的价格，并据此构造出一个新的数据集：

```
$ 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}
```

在如上所示的数据集中我们可以看到，第二瓶酒的年代太久了，而且还过了期，而第一瓶酒的年代则正好。变量间的相互作用，使得这一数据集很适合于对算法进行测试。

k-最近邻算法

k-Nearest Neighbors

对于我们的葡萄酒定价问题而言，最简单的做法与人们尝试手工进行定价时所采用的做法是一样的——即，找到几瓶情况最为相近的酒，并假设其价格大体相同。算法通过寻找与当前所关注的商品情况相似的一组商品，对这些商品的价格求均值，进而作出价格预测。这种方法被称为 k-最近邻算法（k-nearest neighbors，kNN）。

近邻数

Number of Neighbors

kNN 中的 k 代表的，是为了求得最终结果而参与求平均运算的商品数量。对于理想情况下的数据集，我们可以令 k=1，这意味着我们只会选择距离最近的邻居，并将其价格作为最终的答案。不过在现实世界里，总是没有那么理想化的。在本例中，我们故意引入了“噪声”，来模拟这样的情况（随机加减了 20%）。由于有了这些噪声，一部分顾客可能会因此大赚一笔，而也有消息闭塞的顾客，可能会为这样的“最近邻者”多支付很多的钱。基于这样的原因，我们最好多选取一些近邻，然后对它们取平均，以此来减少噪声。

为了形象地说明选择过少近邻的问题，我们以年代为例来考虑一下只有一个描述性变量（descriptive variable）的情形。下页图 8-1 所示的是一幅反映价格（y 轴）与年代（x 轴）之间关系的图。图上还标出了当只使用一个最近邻时所得到的曲线。

请注意此处所预测的价格是怎样过度依赖于曲线的随机变化的。如果我们打算利用图中的曲线进行预测，那么当我们真的只关注一瓶 15 年的葡萄酒和一瓶 16 年的葡萄酒在价格上的差异时，我们就会得出结论，认为这两瓶葡萄酒在价格上存在一个大的跳跃。

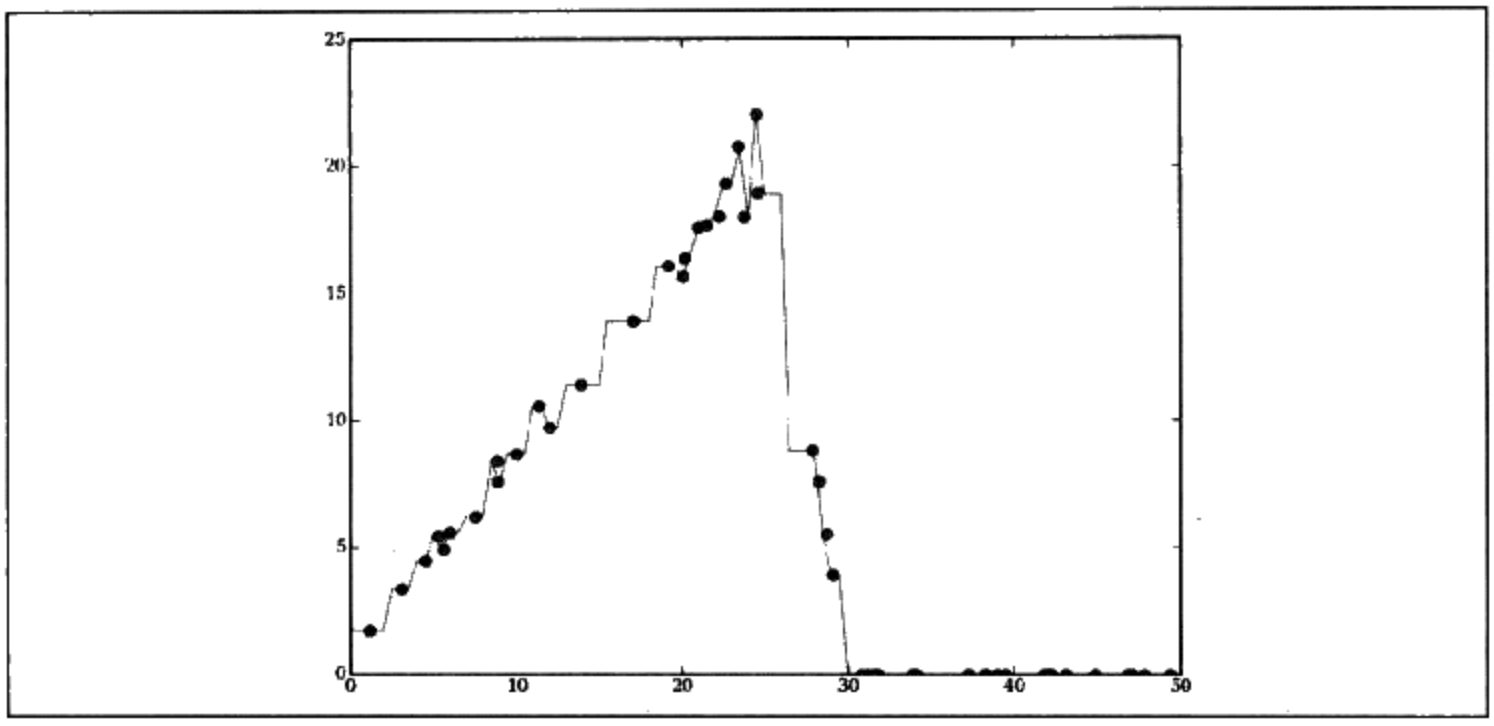


图 8-1：使用过少近邻的 kNN

另一方面，选择过多的近邻同样也会降低准确性，因为算法会对那些与被查询的商品根本没有任何相似性的商品求平均。图 8-2 给出了同样的数据集，以及对 20 个近邻取平均后得到的曲线。

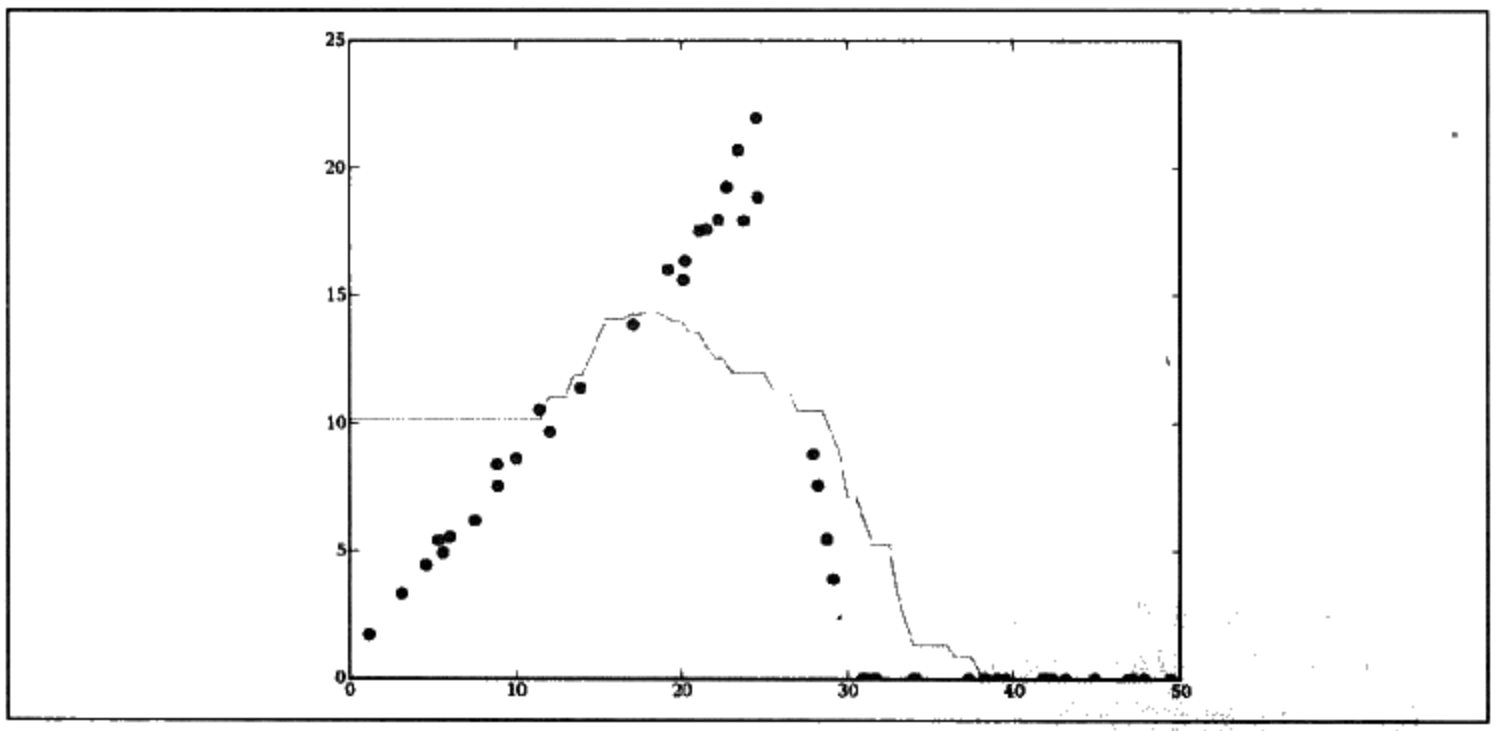


图 8-2：使用过多近邻的 kNN

很显然，对太多的葡萄酒价格取平均，就会极大地低估 25 年左右的葡萄酒的价格对结果所产生的影响。为了选择适当的近邻数，我们可以针对不同的数据集加以手工选择，或者也可以采用一些优化措施。

定义相似度

Defining Similarity

对于 kNN 算法，我们首先要做的一件事情是，寻找一种衡量两件商品之间相似程度的方法。我们已经在本书中学到过各种不同的度量方法。此处，我们将选用欧几里德距离算法，相应的函数我们已经在前几章中介绍过了。请将 `euclidian` 函数加入 *numpredict.py* 中：

```
def euclidean(v1,v2):
  d=0.0
  for i in range(len(v1)):
    d+=(v1[i]-v2[i])**2
  return math.sqrt(d)
```

请在你的 Python 会话中，选择数据集中的某几个数据点，尝试运行上述函数，并得到一个新的数据点：

```
>>> 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
```

你会注意到，该函数在计算距离时对年代和等级是同等看待的，但是现实情况是，某些变量对最终价格所产生的影响往往会比其他变量更大。这是 kNN 众所周知的一个缺点，而解决这一问题的方法将在本章的后续部分讲到。

k-最近邻算法的代码

Code for k-Nearest Neighbors

kNN 是一种实现起来相对简单的算法。虽然这种算法的计算量很大（computationally intensive），但是其优点在于每次有新数据加入时，都无须重新进行训练。请将 `getdistances` 函数加入 *numpredict.py* 中，我们利用该函数来计算给定商品与原数据集中任一其他商品间的距离：

```
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
```

该函数针对给定向量，以及数据集中的任何一个其他的向量，调用距离函数，并将结果置入一个大的列表中。为了让距离最近者位于最前端，我们对列表进行了排序。

kNN 函数利用了上述距离列表，并对其中的前 `k` 项结果求出了平均值。请将 `knnestimate` 加入 *numpredict.py* 中：

```
def knnestimate(data,vec1,k=5):
  # 得到经过排序的距离值
  dlist=getdistances(data,vec1)
  avg=0.0

  # 对前 k 项结果求平均
  for i in range(k):
    idx=dlist[i][1]
    avg+=data[idx]['result']
  avg=avg/k
  return avg
```

现在，我们可以对一件新商品进行估价了：

```
>>> 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)  # Get the actual price # 得到实际价格
30.306122448979593
>>> numpredict.knnestimate(data,(99.0,5.0),k=1) # 尝试更少的近邻
38.078819347238685
```

请尝试不同的参数和不同的 k 值，看一看它们对结果是如何产生影响的。

为近邻分配权重

Weighted Neighbors

目前我们所用的算法有可能会选择距离太远的近邻，对于这样的情况，一种补偿的办法是根据距离的远近为其赋以相应的权重。这种方法与我们在第 2 章中所采用的方法是类似的，在那里我们根据某一位寻求推荐的用户与其他人在偏好上的相近程度，为那些人的偏好赋予了相应的权重。

商品越是相近，彼此间的距离就越小，我们须要一种方法来将距离转换为权重。要完成这项工作可以有多种不同的方法，每一种方法都各有优缺点。本节我们将介绍 3 种方法。

反函数

Inverse Function

在第 4 章中，我们将距离转换为权重所使用的是一个反函数。图 8-3 给出了这种方法的图示，其中一个坐标轴代表的是权重，另一个则代表价格。

该函数最为简单的一种形式是返回距离的倒数。不过有时候，完全一样或非常接近的商品，会使权重值变得非常之大，甚至是无穷大。基于这样的原因，我们有必要在对距离求倒数之前先加上一个小小的常量。

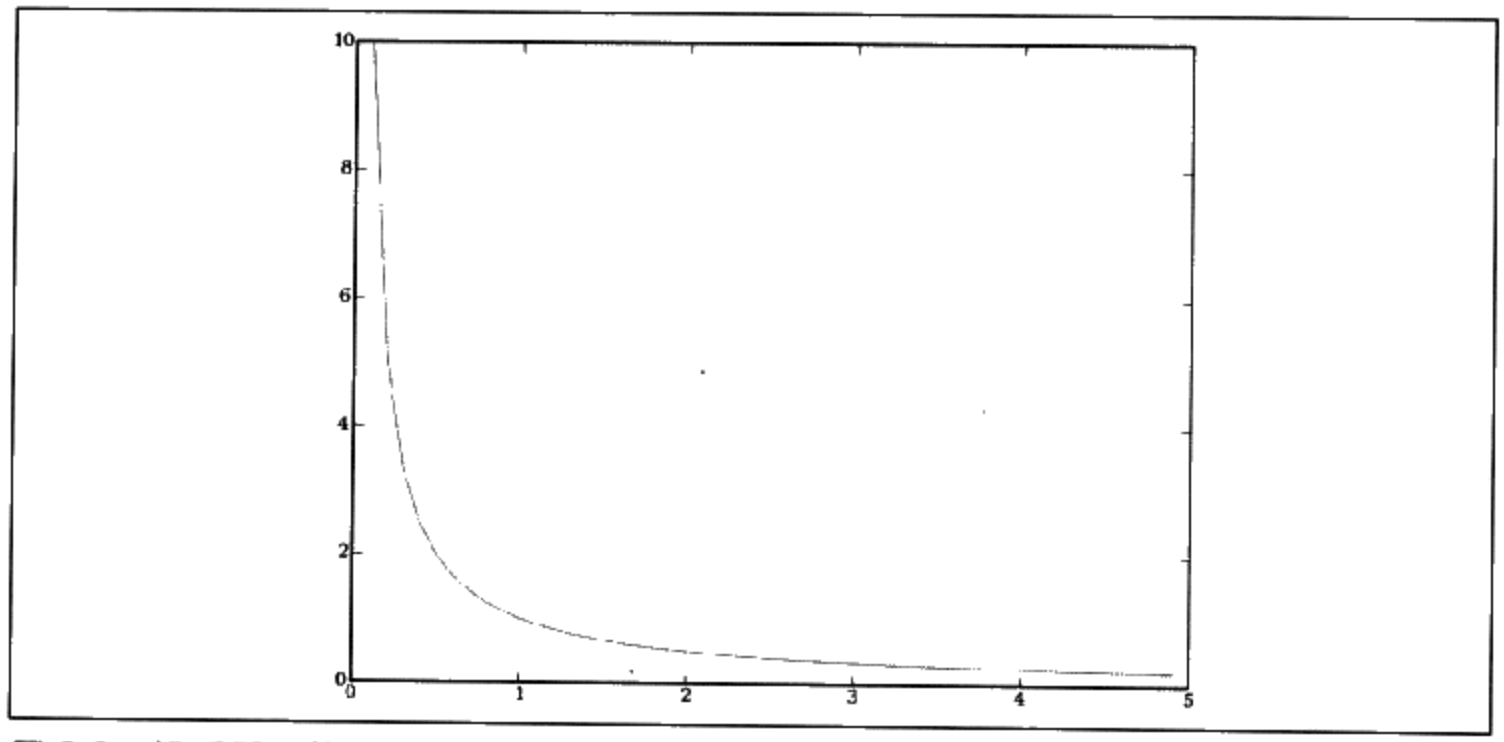


图 8-3：权重反函数

请将 `inverseweight` 函数加入 *numpredict.py* 中：

```
def inverseweight(dist,num=1.0,const=0.1):
  return num/(dist+const)
```

该函数的执行速度很快，也很容易实现，我们还可以尝试一下不同的 `num` 值，看看哪个效果更好。这种方法最为主要的潜在缺陷在于，它会为近邻项赋以很大的权重，而稍远一点的项，其权重则会“衰减”得很快。这种情况也许正是我们所期望的，但有的时候，这也会使算法对噪声变的更加敏感。

减法函数
Subtraction Function

除了反函数外，我们的另一个选择是减法函数，算法示意图如下页图 8-4 所示。

这是一个很简单的函数，它用一个常量值减去距离。如果相减的结果大于 0，则权重为相减的结果；否则，结果为 0。请将 `subtractweight` 函数加入 *numpredict.py* 中：

```
def subtractweight(dist,const=1.0):
  if dist>const:
    return 0
  else:
    return const-dist
```

该函数克服了前述对近邻项权重分配过大的潜在问题，但是它也有自身的局限。由于权重值最终会跌至 0，因此我们有可能找不到距离足够近的项，将其视作近邻，即：对于某些项，算法根本就无法作出预测。

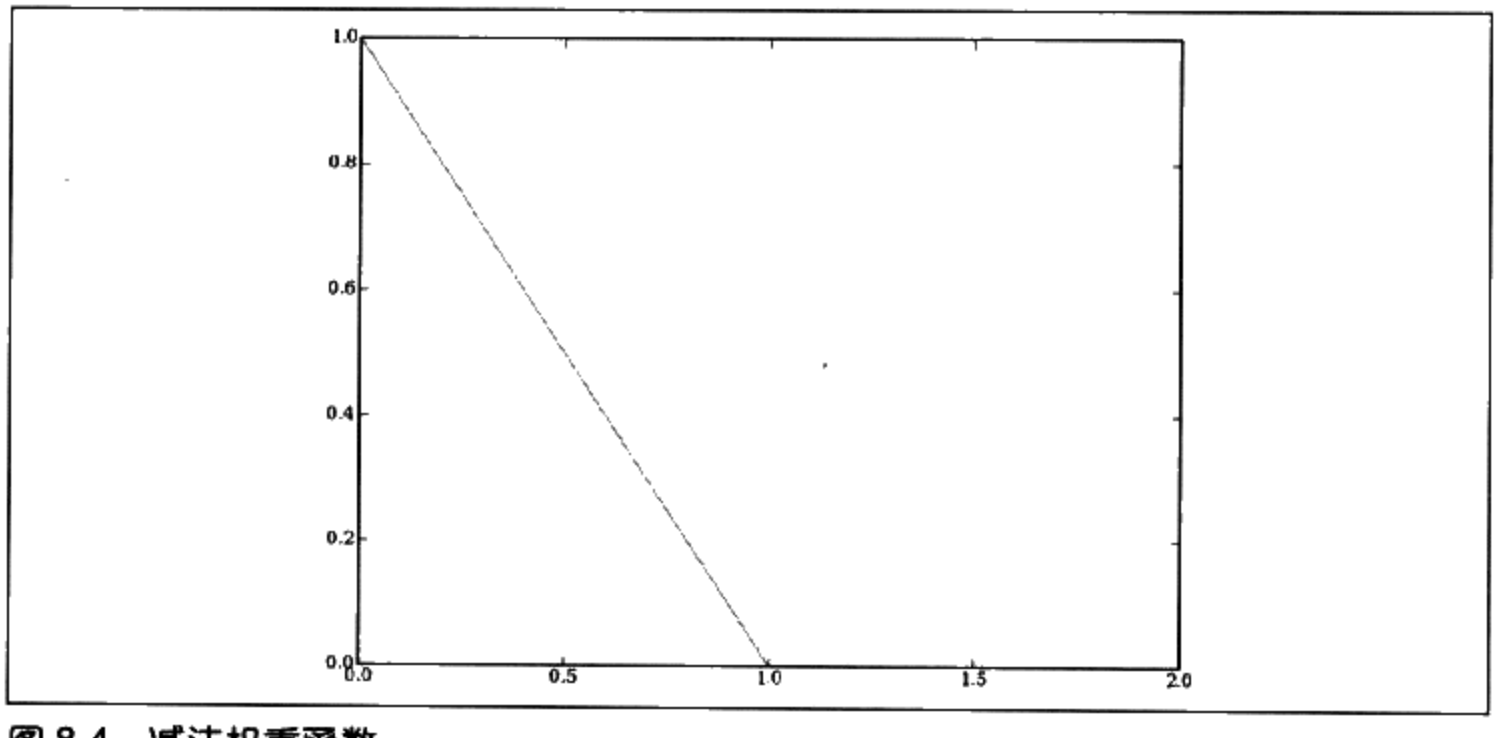


图 8-4：减法权重函数

高斯函数

Gaussian Function

最后我们来看一下高斯函数，也有人称其为“钟型曲线”。该方法要比此前提到的其他函数稍复杂一些，不过你会发现，这种方法克服了前述方法的局限。高斯函数如图 8-5 所示。

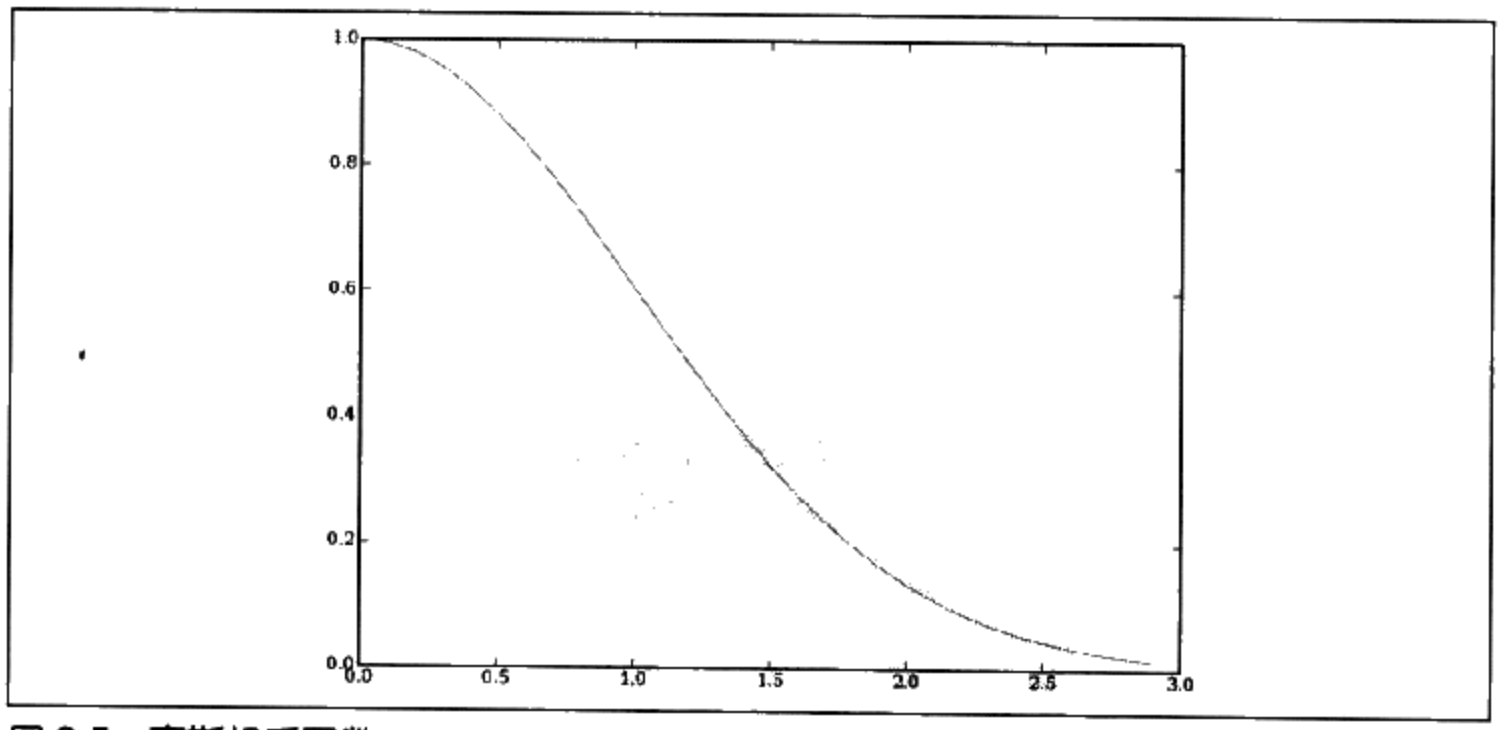


图 8-5：高斯权重函数

该函数在距离为 0 的时候所得的权重值为 1，并且权重值会随着距离的增加而减少。不过，和减法函数不同的是，这里的权重值始终都不会跌至 0，因此该方法总是可以做出预测的。高斯函数的代码有些复杂，而且其执行速度不会像其他两个函数那样快。

请将 gaussian 加入 *numpredict.py* 中：

```
def gaussian(dist,sigma=10.0):
  return math.e**(-dist**2/(2*sigma**2))
```

现在，请针对部分数据项尝试使用上述不同的函数，并传入不同的参数值，看一看这些方法彼此间的差异如何：

```
>>> 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
```

我们可以看到，所有函数都是在距离为 0.0 处得到最大的权重值，然后从那一点开始以不同的方式逐渐递减。

加权 kNN

Weighted kNN

实现加权 kNN 算法的代码与普通的 kNN 函数在执行过程上是相同的，函数首先获得经过排序的距离值，然后取距离最近的 k 个元素。与普通 kNN 相比，加权 kNN 算法最重要的区别在于，它并不是对这些元素简单地求平均，它求的是加权平均。加权平均的结果是通过将每一项的值乘以对应权重，然后将所得结果累加得到的。待求出总和以后，我们再将其除以所有权重值之和。

请将 weightedknn 加入 *numpredict.py* 中：

```
def weightedknn(data,vec1,k=5,weightf=gaussian):
  # 得到距离值
  dlist=getdistances(data,vec1)
  avg=0.0
  totalweight=0.0

  # 得到加权平均值
  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
```

上述函数循环遍历距离最近的 k 个近邻，并将各个距离值传入预先定义好的某个权重函数。变量 avg 的值是通过将权重乘以对应近邻的数值而求得的。变量 totalweight 是权重值的总和。最后，我们将 avg 除以 totalweight。

我们可以在自己的 Python 会话中尝试执行一下上述函数，然后和普通的 kNN 函数做一下性能上的对比：

```
>>> reload(numpredict)
<module 'numpredict' from 'numpredict.py'>
>>> numpredict.weightedknn(data,(99.0,5.0))
32.640981119354301
```

在本例中，通过计算所得的结果我们可以看出，weightedknn 比 knnestimate 更接近正确的答案。不过，这只是针对两组样例而言的。更为严格的测验过程须要涉及数据集中的大量不同项，我们可以利用这些项来决定最优算法和最佳参数。接下来，你将会看到我们是如何进行这样的测试的。

交叉验证
Cross-Validation

交叉验证是将数据拆分成训练集与测试集的一系列技术的统称。我们将训练集传入算法，随着正确答案的得出（在本章的例子中即为价格），我们就得到了一组用以进行预测的数据集。随后，我们要求算法对测试集中的每一项数据都作出预测。其所给出的答案，将与正确答案进行对比，算法会计算出一个整体分值，以评估其所做预测的准确程度。

通常这一过程会执行若干次，每次对数据的拆分都不相同。典型的情况下，测试集只会包含一小部分数据，大概是所有数据的 5%，剩下的 95%则构成了训练集。要实现这一算法，请先在 *numpredict.py* 中新建一个名为 dividedata 的函数，该函数会根据你给定的比率，将原有数据集拆分为两个较小的集合：

```
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
```

接下来，我们为算法提供训练集，并针对测试集中的每一项数据来调用算法，以此对算法加以测试。下面给出的函数会求得差值，并将这些差值组合起来建立起一个评分值的聚合结果，以此来评估预测结果与正确结果大体上的差距。为此，我们通常需要将所有差值的平方累加起来。

请将一个新的函数，testalgorithm，加入 *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)
```

testalgorithm 的 algf 接受一个算法函数作为参数，而该算法函数则接受一个数据集和一个查询项作为参数。testalgorithm 会循环遍历测试集中的每一行，并利用 algf 得出最佳的猜测结果。随后，它会从实际结果中减去猜测所得的结果。

对数字求平方是一种常见的做法，因为它会突显较大的差值。这意味着，一个在大多数时候都非常接近于正确值，但是偶尔会有较大偏离的算法，要比始终都比较接近于正确值的算法稍逊一些。一般而言，这种情况是我们所期望的，不过有时也有例外，那就是：如果算法在余下的大多数时候准确度都非常的高，那么偶尔犯一个大错误还是可以接受的。如果是这种情况，那么我们可以对函数稍做修改，只要将差值的绝对值累加起来即可。

实现交叉验证算法的最后一个步骤是编写一个函数，对数据采取若干不同的划分，并在每个划分上执行 testalgorithm 函数，然后再将所有的结果都累加起来，以求得最终的评分值。我们将 crossvalidate 加入 *numpredict.py* 中：

```
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
```

目前为止所编写的代码有许多可以调整的地方，我们可以针对不同的调整方式进行比较。例如，我们可以利用不同的 k 值来试验 knnestimate 函数。

```
>>> 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
```

正如我们所预料的，选择太少或太多的近邻都会导致效果不彰。在本例中，值为 3 的效果要比 1 或 5 好。你也可以试一下此前曾为加权 kNN 算法定义过的各种不同的权重函数，看看哪一个函数能够给出最佳的结果：

```
>>> 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
```

待我们正确设置好参数之后，加权 kNN 算法似乎能够针对上述数据集给出更好的结果来。选择正确的参数也许会花费一定的时间，但是对于一个特定的训练集而言，这样的工作我们只须要做一次即可，或许，随着训练集内容的增长，偶尔我们还须要对其进行再次的更新。在本章稍后的“对缩放结果进行优化”一节中，我们将会考查自动确定部分参数的方法。

不同类型的变量

Heterogeneous Variables

我们在本章开始处所构造的数据集是特地做了人为简化的，用来预测价格的所有变量大致上都是可比较的，而且这些变量对最终的结果都很重要。

因为所有变量都位于同一值域范围内，因此利用这些变量一次性算出距离值是有意义的。不过，假设我们引入了一个对价格产生影响的新变量，诸如以毫升为单位的酒瓶尺寸。与我们目前已经使用过的变量不同（那些变量的取值均介于 0 和 100 之间），这些变量的值域范围可能会达到 1500。请见下页图 8-6，看看这种情况是如何对最近邻或加权距离的计算结果构成影响的。

很显然，和原先的变量相比，这个新的变量对距离计算所产生的影响更为显著——其影响将超过任何其他变量对距离计算所构成的影响，这意味着，在计算距离的过程中其他变量根本就未被考虑在内。

除此以外，我们可能会遇到的另一个问题是数据集中引入了完全不相关的变量。如果前面的数据集中还包含了安放葡萄酒的通道号，那么这一变量也将会被纳入距离计算之中。如此一来，对于其他方面都完全一样，唯独所在通道不一样的两瓶葡萄酒而言，算法也会将其认为是不同的，这种情况会导致算法预测的准确度大打折扣。

加入数据集

Adding to the Dataset

为了模仿上述效果，我们须要将一些新的变量加入到数据集中。我们可以照搬 `wineset1` 中的代码，新建一个名为 `wineset2` 的函数，并对其进行修改，加入下列以黑体显示的部分：

```
def wineset2():
  rows=[]
  for i in range(300):
    rating=random()*50+50
    age=random()*50
```

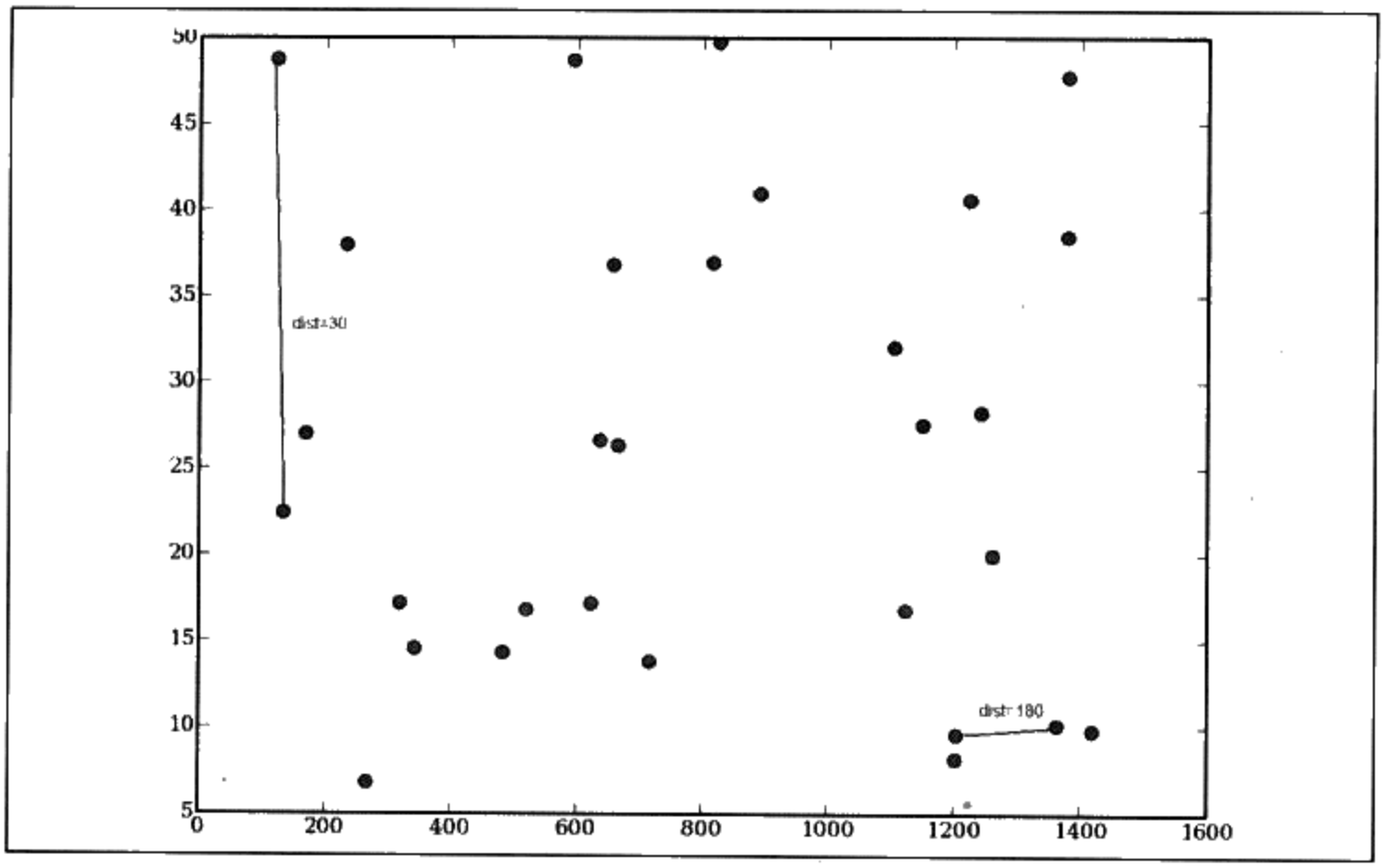


图 8-6：不同类型的变量导致距离的计算出现问题

```
    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
```

现在，我们可以构造一个带有通道信息和酒瓶尺寸的新数据集了：

```
>>> reload(numpredict)
<module 'numpredict' from 'numpredict.py'>
>>> data=numpredict.wineset2()
```

为了看到新数据集对 kNN 预测算法的影响情况，请按照我们此前得到的最佳参数值，试验一下这个新的数据集：

```
>>> numpredict.crossvalidate(knn3,data)
1427.3377833596137
>>> numpredict.crossvalidate(numpredict.weightedknn,data)
1195.0421231227463
```

我们会发现，即使数据集现在包含了更多的信息，而且噪声也比以前更少了（理论上，这应该会得到更好的预测结果），但是 crossvalidate 函数实际返回的结果较之以前却更为糟糕。其原因就在于，算法现在还不知道如何对不同的变量加以区别对待。

按比例缩放
Scaling Dimensions

此处，我们所需要的并不是一种根据变量的实际值来计算距离的方法，而是需要一种对数值进行归一化处理的方法，从而使所有变量都位于相同的值域范围之内。这样做也有助于找到减少多余变量的方法，或者至少能够降低其对计算结果的影响。为了达成上述两个目标，一种办法就是在进行任何计算之前先对数据重新按比例进行缩放。

按比例重新进行缩放的最简单形式是将每个维度上的数值乘以一个在该维度上的常量。如图 8-7 所示。

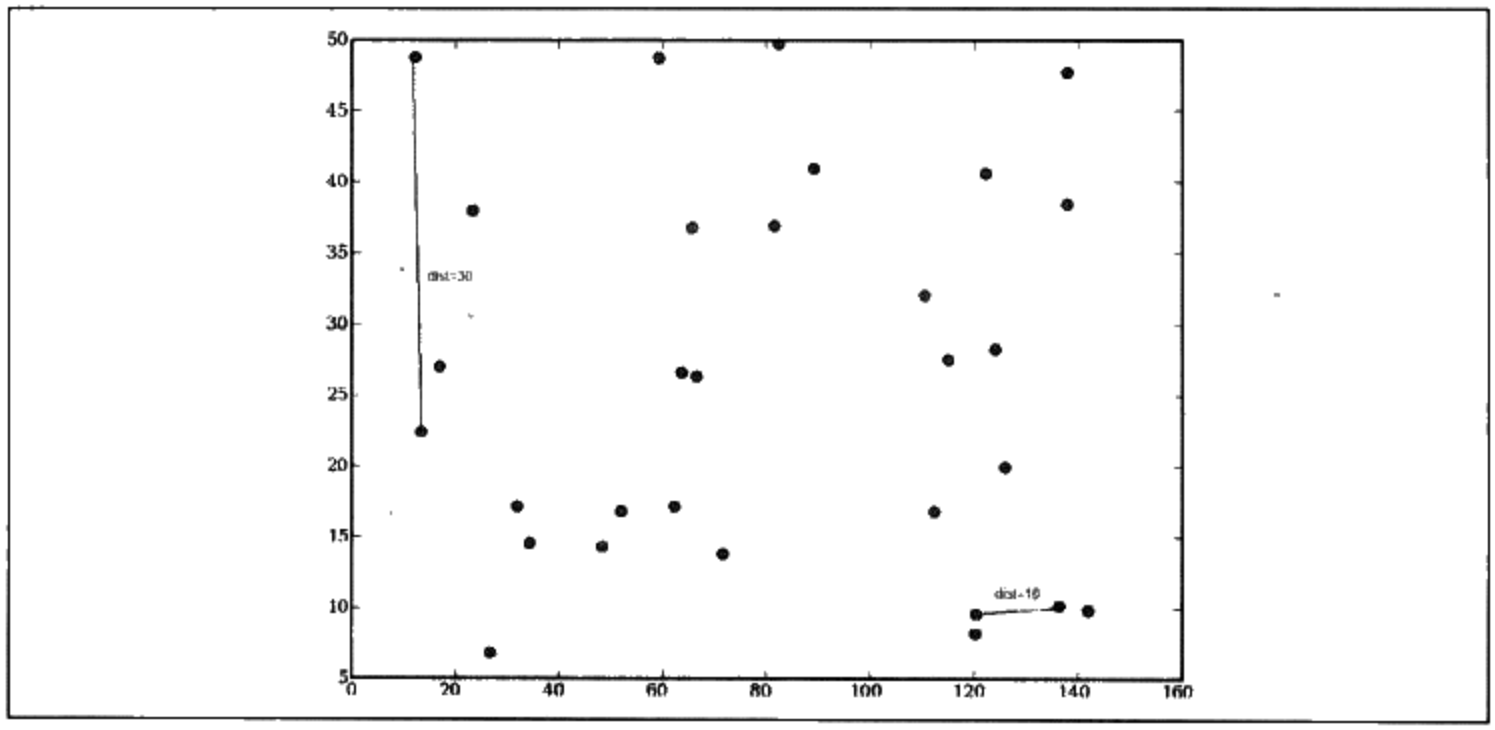


图 8-7：对各个维度进行缩放解决了距离计算的问题

我们可以看到，代表酒瓶尺寸的维度被值为 10 的比例因子缩小了，相应地，一些项的最近邻也产生了变化。这种做法解决了一部分变量天生比其他变量更“强势” 的问题，但是对于那些重要程度不高的变量，又如何处置呢？设想一下，如果将某个维度上的每一项数值都乘以 0 会是怎样呢，如下页图 8-8 所示。

请注意，在代表通道的维度上，每一项数据所处的地位都是相同的，因此两项之间的距离就完全取决于其在年代维度上所处的位置了。也就是说，在计算最近邻的过程中，通道这一变量已经变得毫无意义，并且被彻底忽略了。如果所有无关紧要的变量都被缩减到了 0，那么算法将会变得更加准确。

函数 `rescale` 接受一个列表项和一个名为 `scale` 的实数列作为参数。该函数返回一个新的数据集，其中的所有数据都乘了 `scale` 参数中的对应值。请将 `rescale` 加入 *numpredict.py* 中：

```
def rescale(data,scale):
  scaleddata=[]
  for row in data:
```

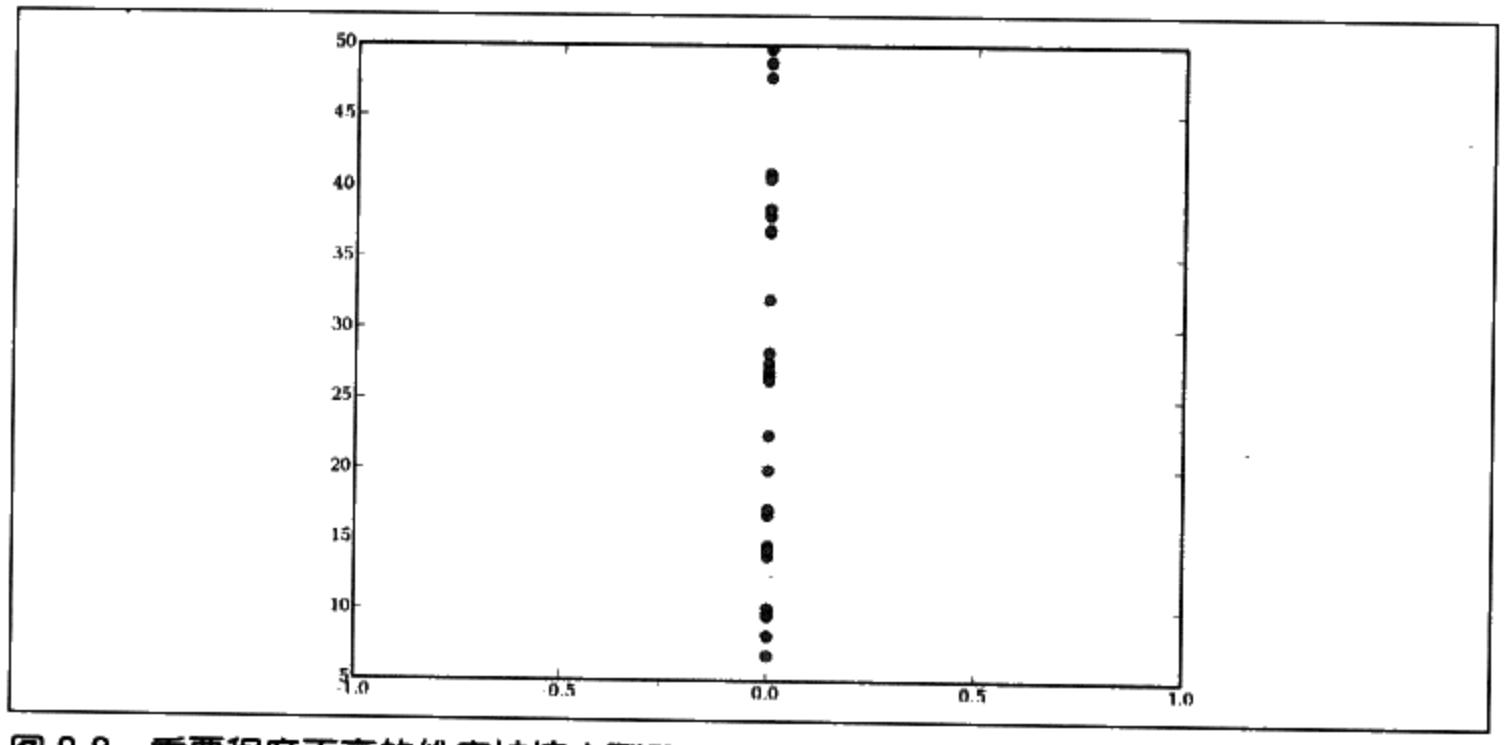


图 8-8：重要程度不高的维度被缩小到了 0

```
    scaled=[scale[i]*row['input'][i] for i in range(len(scale))]
    scaleddata.append({'input':scaled,'result':row['result']})
  return scaleddata
```

我们可以精心挑选一些参数，试着对数据集按比例重新进行缩放，看一看是否能够得到满意的预测结果：

```
>>> 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
```

对于上述少量样例数据而言，这样的结果已经相当不错了；肯定要比之前所得的结果好。请尝试更改一下 scale 参数的值，看看是否能够得到更好的结果。

对缩放结果进行优化
Optimizing the Scale

在本章的例子里，要选择一个合适的参数进行缩放并不困难，因为我们事先已经知道了哪个变量是重要的。但是，大多数时候我们所面对的数据集都不会是自己构造的，而且我们也未必知道，到底哪些变量是不重要的，而哪些变量又对计算结果有着重大的影响。

理论上，我们可以尝试大量不同数值的组合，直到发现一个足够好的结果为止，不过也许会有数以百计的变量须要考查，并且这项工作可能会非常地乏味。所幸的是，假如你通读过第 5 章，想必已经知道了，如何在有许多输入变量须要考查的情况下，利用优化算法自动寻找最优解的办法。

也许你还记得，优化的过程只要求我们提供一个指定变量个数和值域范围的定义域参数，和一个成本函数即可。由于函数 crossvalidate 对于较差的给定题解，会返回一个较高的数值结果，因此它已经是一个天然的成本函数了。我们唯一要作的事情，就是将它封装起来，令其接受一组数值作为参数，然后对数据按比例进行缩放，并计算交叉验证的误差。请将 createcostfunction 加入 *numpredict.py*：

```
def createcostfunction(algf,data):
  def costf(scale):
    sdata=rescale(data,scale)
    return crossvalidate(algf,sdata,trials=10)
  return costf
```

此处的定义域就是每一个维度上的权重范围。在本例中，由于负数只会得到原数值的一个镜像（mirror image），这对距离计算不会有任何影响，因此我们将定义域的最小可能值设置为 0。理论上，我们想将权重设成多大的值都是可以的，但从实际应用的角度出发，眼下我们只须将其限制在 20 即可。请将如下代码行加入 *numpredict.py*：

```
weightdomain=[(0,20)]*4
```

现在，我们已经为自动优化权重值做好了一切准备。请确保 *optimization.py*（我们在第 5 章中建立的文件）位于当前目录下，然后在 Python 会话中尝试一下退火优化算法：

```
>>> 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]
```

非常好！算法不但明确了通道是一个没有价值的变量，并将其按比例缩小到了几乎为 0，而且它还指出了，相比于其对结果的影响，酒瓶的尺寸被不成比例的放大了；除此以外，它还将另外两个变量做了相应地放大。

我们还可以尝试一下执行速度更慢但是一般来说更加精确的 geneticoptimize 函数，看看它是否会返回类似的结果：

```
>>> optimization.geneticoptimize(numpredict.weightdomain,costf,popsize=5,\\
    lrate=1,maxv=4,iters=20)
[20,18,0,12]
```

以这种方式对变量缩放进行优化的一个好处在于，我们很快就能发觉哪些变量是重要的，并且其重要程度有多大。有的时候，有些数据很难收集到，或者收集的代价高昂，此时如果能够确定这些数据不是很有价值，那就可以将其忽略以避免额外的成本投入。此外，特别是在制定价格策略的时候，知道哪些变量是重要的，有可能会影响到我们所关注的方向，而这些方向将会成为市场营销工作中相当重要的一部分内容；另外，这样做也有可能会为我们揭示出，如何将商品设计得与众不同，才能赢得最高的价格。

不对称分布

Uneven Distributions

到目前为止，我们已经假设了，如果你对数据求平均或加权平均，那么就会得到一个有关最终价格的合理估计。在许多场合下，这样做是没有问题的，但有些时候，也可能会存在一些无法测定的变量，它们会对结果产生很大的影响。假设在本章的例子中，葡萄酒购买者分别来自两个彼此独立的群组：一部分人是从小酒馆购得的葡萄酒，而另一部分人则是从折扣店购得，并且后者得到了50%的折扣。不幸的是，这些信息在数据集中并没有被记录下来。

函数 `createhiddendataset` 构造了一个数据集，用以模拟这样的一系列特征。它去除了某些复杂变量，并将注意力集中在了某几个基本的变量上。请将该函数加入 *numpredict.py* 中：

```
def wineset3():
  rows=wineset1()
  for row in rows:
    if random()<0.5:
      # 葡萄酒是从折扣店购得的
      row['result']*=0.5
  return rows
```

试想一下，假如我们使用 kNN 算法或加权 kNN 算法，对不同的葡萄酒进行价格预估，会发生什么样的情况。由于事实上数据集中并不包含任何有关购买者是从小酒馆还是从折扣店购买葡萄酒的信息，因此算法无法将这一情况考虑在内，其所得到的最近邻结果也将不会考虑购买源这一因素。最终的结果是：算法给出的平均值将同时涉及两组人群，这就相当于可能有 25%的折扣。我们可以在自己的 Python 会话中做一下试验，验证一下这种情况：

```
>>> 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
>>> 599.51654107008562
```

如果你只想得到一个简单的数字，那么这不失为一种预测的办法，但是它并不能准确地反应出某人实际最终的购买情况。为了不只是简单地得到一个平均值，我们需要一种方法能够在某些方面更近一步地对数据进行考查。

估计概率密度

Estimating the Probability Density

除了取近邻的加权平均并得到一个价格预估外，知道某瓶葡萄酒落入指定价格区间的概率

也有可能是一件颇值得关注的事情。在本章的例子中，假设输入条件为99%和20年，那么我们需要一个函数来告诉我们，价格介于40美元和80美元之间的几率是50%，而价格介于80美元和100美元之间的几率也是50%。

为了达成这一点，我们需要一个函数能够返回一个介于0和1之间的值，用以代表概率。函数首先计算位于该范围内近邻的权重值，然后计算所有近邻的权重值。最终的概率等于在指定范围内的近邻权重之和除以所有权重之和。为了实现这一计算过程，请在*numpredict.py*中新建一个名为probguess的函数：

```
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']

    # 当前数据点位于指定范围内吗?
    if v>=low and v<=high:
      nweight+=weight
    tweight+=weight
  if tweight==0: return 0

  # 概率等于位于指定范围内的权重值除以所有权重值
  return nweight/tweight
```

和kNN算法一样，该函数根据与vec1距离的远近对数据进行排序，并确定最近邻的权重值。函数将所有近邻的权重加在一起得到tweight。它还会判断每个近邻的价格是否位于指定范围内（介于low和high之间）；如果是，则将权重计入nweight。针对vec1而言，价格介于low和high之间的概率，等于nweight与tweight相除的结果。

现在，请针对前面的数据集，尝试执行一下该函数：

```
>>> reload(numpredict)
<module 'numpredict' from 'numpredict.py'>
>>> numpredict.probguess(data,[99,20],40,80)
0.623059884514972 96
>>> 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
```

函数给出了一个合理的执行结果。位于实际价格以外的区间对应概率为 0，而覆盖全部区间的概率则接近于 1。通过将区间拆分成更小的区段，我们可以确定出每一瓶葡萄酒倾向于集中分布的实际值域范围。不过，这要求我们不断地猜测范围区间，并将其作为输入，直到对数据的整体结构有了一个清晰的认识为止。在下一节中，我们将会学到一种能够获得概率分布整体视图的方法。

绘制概率分布

Graphing the Probabilities

为了避免胡乱猜测范围区间，我们可以建立概率密度的图形化表达。有一个用于数学图形绘制的优秀函数库，名叫 *matplotlib*，该函数库是用 Python 语言编写的，而且是完全免费的，我们可以从 *http://matplotlib.sourceforge.net* 处下载到它。

在 matplotlib 的网站上有相应的安装说明文档，附录 A 中还有关于 matplotlib 的更多信息。该函数库功能非常强大，且特性很多，本章中我们将只使用其中的一小部分。在安装完毕之后，我们可以在自己的 Python 会话中试着绘制一个简单的图形：

```
>>> 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()
```

上述代码绘制出了一个简单的图形，如下页图 8-9 所示。与函数 `arange` 类似，函数 `arange` 构造了一个数组，其中包含了一组数字。在本例中，我们所绘制的是一条从 0 到 10 的正弦曲线。

本节将为大家介绍两种不同的查看概率分布的方法。第一种方法被称为**累积概率**（cumulative probability）。累积概率图显示的是结果小于给定值的概率分布情况。以价格为例，图形从概率为 0 开始（对应于价格小于 0 的概率），尔后随着商品在某一价位处命中的概率值而逐级递增。直到最高价位处，图形对应的概率值达到 1（因为实际价格小于或等于最高价格的可能性必为 100%）。

为累积概率图构造数据是非常简单的，我们只须循环遍历价格的值域范围，并以 0 为下限，以某个指定价格为上限，调用 `probabilityguess` 函数。我们可以将这些调用的结果数据传入 `plot` 函数中，以生成对应的图形。请将 `cumulativegraph` 加入 *numpredict.py* 中：

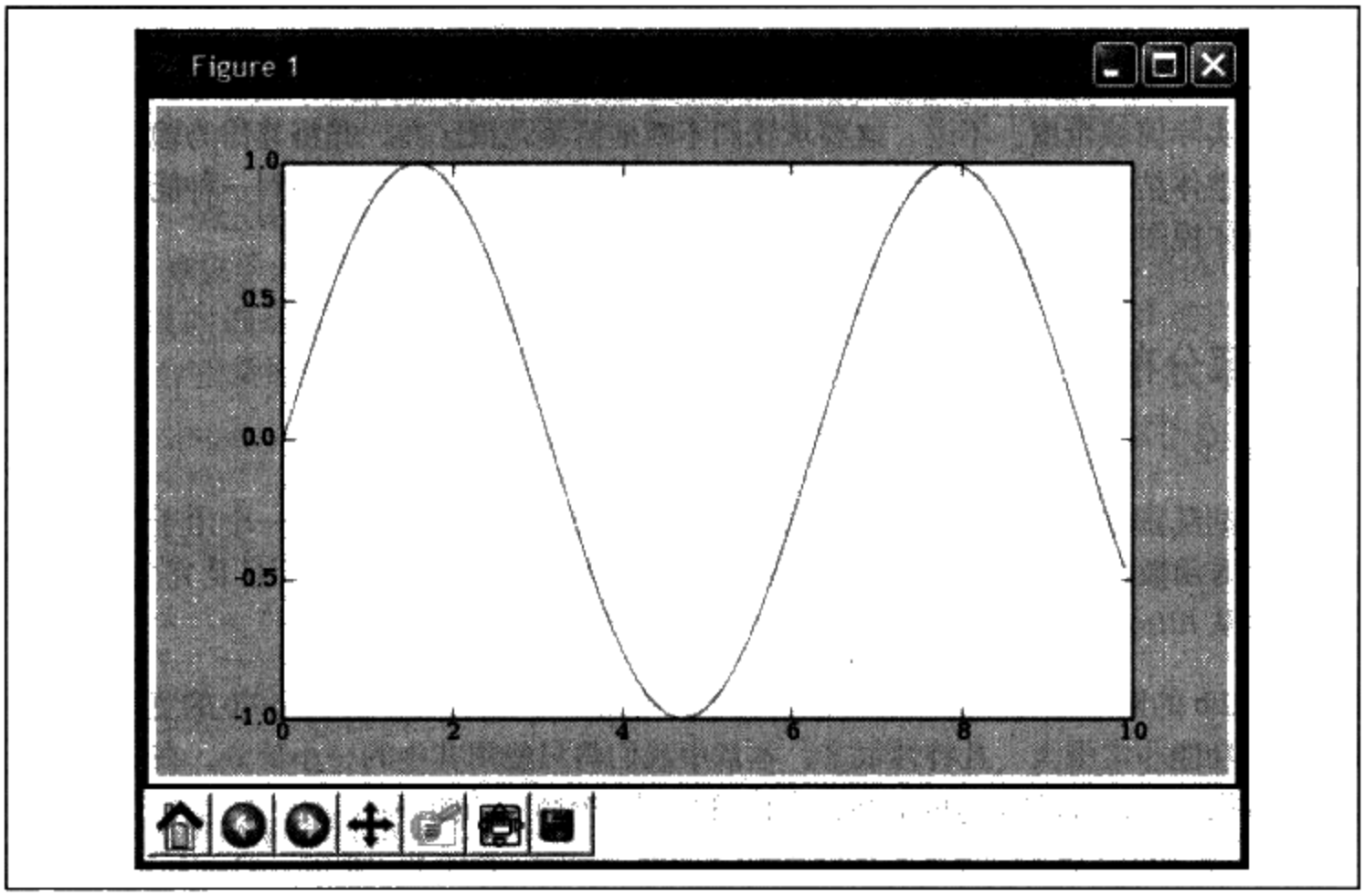


图 8-9：使用 matplotlib 的例子

```
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()
```

现在，我们可以在自己的 Python 会话中调用该函数，并生成相应的图形了：

```
>>> reload(numpredict)
<module 'numpredict' from 'numpredict.py'>
>>> numpredict.cumulativegraph(data,(1,1),120)
```

利用 cumulativegraph 函数绘制出来的图形类似于下页图 8-10 所示。正如我们所期望的，累积概率从 0 开始，并一路递增至 1。这幅图中值得注意的地方在于概率递增的方式：概率值在到达 50 美元附近之前始终都处于 0，随后就以极快的速度攀升至 0.6，尔后这一状态一直维持到 110 美元价位处，在那里又再一次跳跃。

通过观察图形，我们可以很清楚地看到，此处的概率值集中在 60 美元和 110 美元之间，因为那段区间是累积概率发生跳跃的地方。预先得知这一情况将使我们能够在不依靠猜测的情况下进行概率的计算。

除了累积概率外，还有一种绘制概率分布的方法，那就是尝试将处于不同价位点的实际概率值绘制出来。由于任何一瓶葡萄酒准确位于某一价位的概率都是非常低的，因此这种方法相比而言所要求的技巧性会更高一些。这样绘制出来的图形，在我们预测的价格附近会形成一个个小小的突起，而其余地方则几乎都会是 0。这样的结果并不是我们想要的，我们需要的是有一种方法能够在某些“窗口”（windows）范围内将概率值组合起来。为了达成

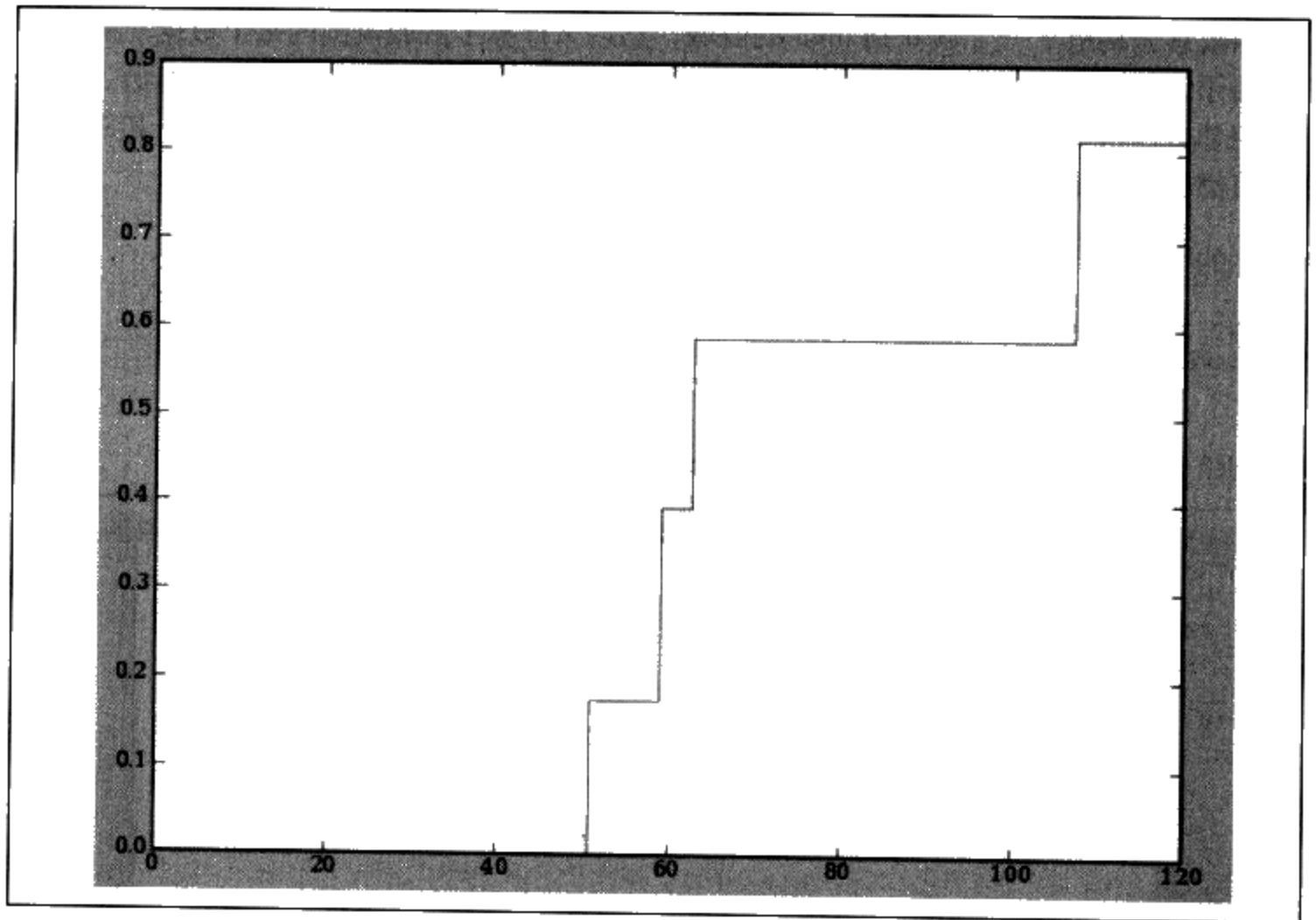


图 8-10：累积概率图

这一目的，有一种办法就是，假设每个价位点的概率都等于其周边概率的一个加权平均，这与加权 kNN 算法非常类似。

要想马上看到效果，请将 probabilitygraph 加入 *numpredict.py*：

```
def probabilitygraph(data,vec1,high,k=5,weightf=gaussian,ss=5.0):
  # 建立一个代表价格的值域范围
  t1=arange(0.0,high,0.1)

  # 得到整个值域范围内的所有概率
  probs=[probguess(data,vec1,v,v+0.1,k,weightf) for v in t1]

  # 通过加上近邻概率的高斯计算结果，对概率值做平滑处理
  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()
```

上述函数首先构造了一个从 0 到 high 的值域，然后又计算了值域范围内每个数据点的概率值。由于这样做通常会导致图形出现明显的锯齿，因此函数又对数组做了循环遍历，通过追加相邻概率值的方法对数组进行了平滑处理。经过平滑处理后的每个数据点，其对应的概率值都是邻近概率的高斯加权和。参数 ss 指定了概率应被平滑处理的程度。

请在你的 Python 会话中尝试执行一下该函数：

```
>>> reload(numpredict)
<module 'numpredict' from 'numpredict.py'>
>>> numpredict.probabilitygraph(data,(1,1),6)
```

执行的结果应该得到一个类似图 8-11 所示的图形。

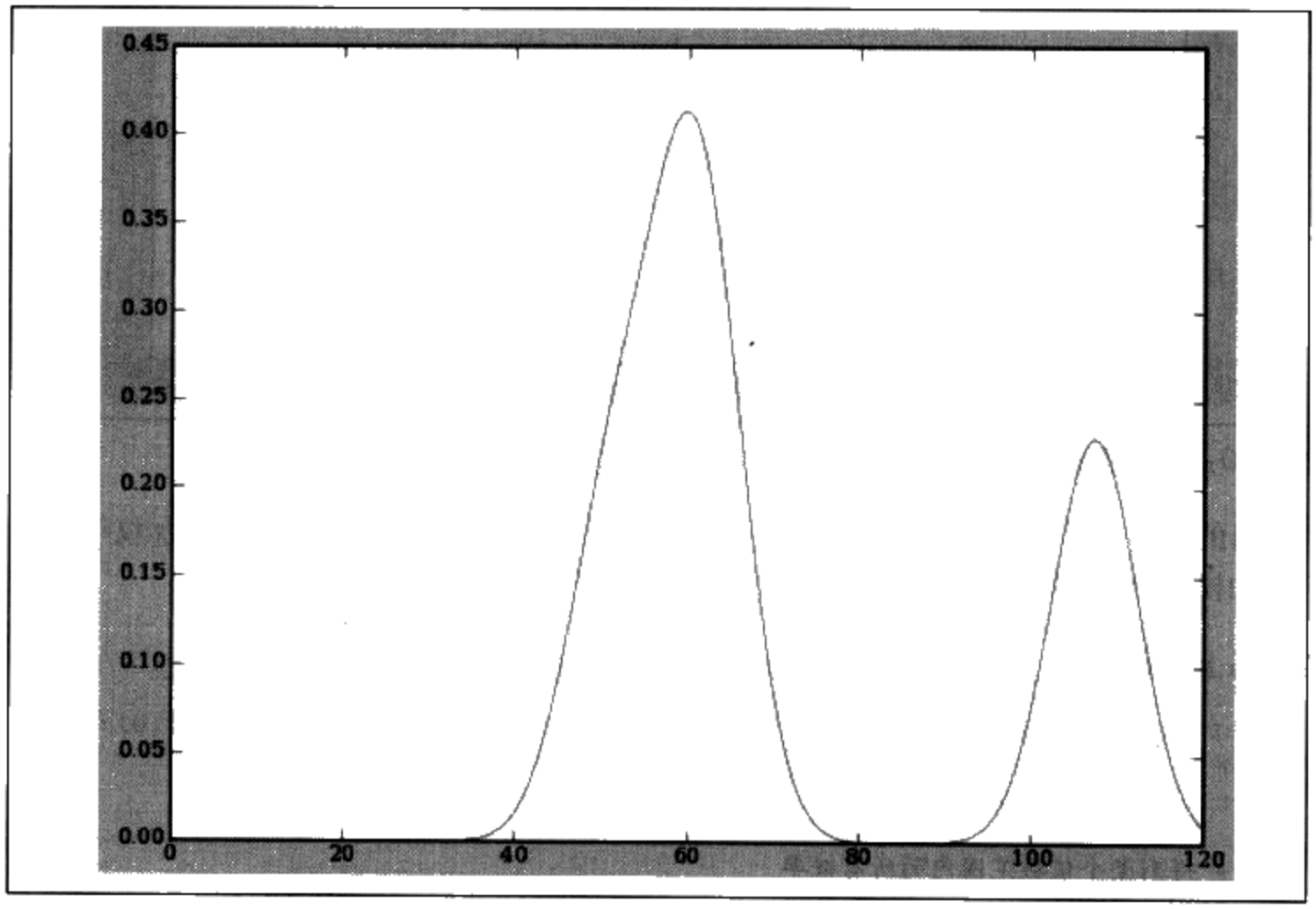


图 8-11：一幅概率密度图

通过上图我们可以更加清楚地看到结果数据集中分布的区域。请尝试不同的窗口参数 ss，看看相应结果的变化情况。这样的概率分布清楚地反映出：我们在预测葡萄酒价格时缺少了一部分关键数据，那就是有些人的葡萄酒生意何以比其他人做得更好的原因。有的时候，我们能够明确地指出这些数据是什么，但有的时候，我们只会发现自己须要在更低的价格范围内销售葡萄酒才行。

使用真实数据——eBay API

Using Real Data—the eBay API

eBay 是一个在线的拍卖网站，也是互联网上最受欢迎的站点之一。它拥有数以百万计的商品名目，还有数以百万计的用户参与竞拍，并一起制定价格，这些都使得 eBay 成为了集体智慧的一个绝佳的应用案例。恰好，eBay 也有免费的基于 XML 的 API，我们可以利用它来进行搜索，取得商品的详细信息，并提交供出售的商品。在本节中，我们将会看到如何利用 eBay API 来获得商品的价格数据，并对数据进行转换，以便能够使用本章中的算法来进行预测。

获取开发者密钥

Getting a Developer Key

访问 eBay API 的流程需要若干个步骤，不过这是相当简单和自动化的。有关流程的一个很好的综述性文档，请见在线的“快速指南”，该指南位于 *http://developer.ebay.com/quickstartguide* 处。

该指南将向你介绍整个流程的执行过程，包括：建立开发者账号、获得生产密钥（production keys）、创建令牌（token）。待上述工作完成之后，我们手中应该拥有 4 个字串，它们是运行本章示例所必需的。

- 开发者密钥
- 应用密钥
- 证书密钥
- 长长的认证令牌（authentication token）

请新建一个名为 *ebaypredict.py* 的文件，并加入如下代码，这些代码引入了某些外部模块，并将上述字串包含了进来：

```
import httplib
from xml.dom.minidom import parse, parseString, Node

devKey = 'developerkey'
appKey = 'applicationkey'
certKey = 'certificatekey'
userToken = 'token'
serverUrl = 'api.ebay.com'
```

虽然，目前还没有官方的针对 eBay 的 Python API 出现，不过 eBay 提供了 XML 的 API，我们可以利用 *httplib* 和 *minidom* 对 eBay API 进行访问。本节我们将只介绍 eBay API 中的两个调用，`GetSearchResults` 和 `GetItem`，不过此处给出的大多数代码都可以被复用于其他调用。有关这套 API 所支持的全部调用的更多信息，可以查看位于 *http://developer.ebay.com/DevZone/XML/docs/WebHelp/index.htm* 处的完整文档。

建立连接
Setting Up a Connection

一旦得到了密钥，我们就可以建立一个指向 eBay API 的连接。eBay API 要求我们向它传递大量的头信息（headers），其中包括密钥和即将要发起的调用。为此，请建立一个名为 `getHeaders` 的函数，接受一个调用名称作为参数，并返回一个包含头信息的字典，我们可以将这个字典传递给 `httplib`。请将该函数加入 *ebaypredict.py* 中：

```
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
```

除了头信息外，eBay API 还要求针对我们所发起的请求，发送一段带有参数信息的 XML 数据。它会相应地返回一个 XML 文档，我们可以利用 minidom 库中的 *parseString* 对其进行解析。

发送请求的函数会打开一个指向服务器端的连接，并提交带参数信息的 XML，然后再对返回结果进行解析。请将 `sendrequest` 加入 *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 "Error sending request:" + response.reason
  else:
    data = response.read()
    connection.close()
  return data
```

我们可以利用上述函数对 eBay API 发起任何形式的调用。对于不同的 API 调用，我们须要生成用于发起请求的 XML，并对解析后的结果加以解释。

由于解析 DOM 的过程有些索然无味，所以我们还应该建立一个简单的辅助函数，`getSingleValue`，该函数用于查找节点并返回节点对应的内容：

```
def getSingleValue(node,tag):
  nl=node.getElementsByTagName(tag)
  if len(nl)>0:
    tagNode=nl[0]
    if tagNode.hasChildNodes():
      return tagNode.firstChild.nodeValue
  return '-1'
```

执行搜索

Performing a Search

执行一次搜索，就是为 GetSearchResults 的 API 调用构造 XML 参数，并将其传入此前定义好的 sendrequest 函数。XML 参数的格式如下所示：

```
<GetSearchResultsRequest xmlns="urn:ebay:apis:eBLBaseComponents">
<RequesterCredentials><eBayAuthToken>token</eBayAuthToken></RequesterCredentials>
<parameter1>value</parameter1>
<parameter2>value</parameter2>
</GetSearchResultsRequest>
```

我们可以为该 API 调用传入许多参数，不过在本章的例子里，我们将只考查其中的两个参数：

`Query`

这是一个包含搜索词条的字符串。使用该参数就如同在 eBay 的主页上手工进行搜索一样。

`CategoryID`

这是一个数字，它指定了我们想要搜索的分类。eBay 有一个巨大的分类层次结构，你可以利用 GetCategories API 调用来请求这些信息。我们可以单独使用这一参数，也可以与参数 Query 结合使用。

函数 doSearch 接受上述两个参数，并执行一次搜索。随后，它会返回一个商品 ID 的列表（稍后我们将在 GetItem 调用中用到它），以及这些商品的描述信息和当前的价格。请将 doSearch 加入 *ebaypredict.py* 中：

```
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
```

为了使用分类参数，我们还需要一个获取分类层次信息的函数。这又是一个简单的 API 调用，只不过涉及全部分类数据的 XML 文件非常庞大，下载须要花费很长的时间，并且解析难度也非常的大。因此，我们打算将分类数据的范围限制在人们通常所关注的那一部分内容。

函数 getCategory 接受一个字符串和一个父分类 ID 作为参数，并返回位于父分类下的包含该字符串的所有分类。如果父 ID 缺失，则函数只会给出包含所有顶层分类的列表。请将该函数加入 *ebaypredict.py* 中：

```
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
```

现在，我们可以在自己的 Python 会话中尝试执行该函数了：

```
>>> 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'),
...
```

哦，看起来搜索“laptop”的返回结果中包含了各种与膝上型电脑关系不是很大的附件（accessories）。所幸的是，我们可以搜索“Laptops, Notebooks”分类，将搜索范围限制在与膝上型电脑真正相关的内容之中。为此，我们首先须要获取顶层的分类列表，然后在“Computers & Networking”范围内进行搜索，找到涉及膝上型电脑的分类 ID，然后我们就可以在正确的分类范围内搜索“laptop”了：

```
>>> 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'),...
```

在本书撰写期间，eBay 上有关“Laptops, Notebooks”分类的 ID 一共有 51148 个。你可以看到，我们通过只查询“laptop”，将搜索范围限制在了“laptop”分类范围内，从而减少了大量无关的搜索结果。这种搜索行为与搜索结果的高度一致性，使得我们的数据集非常适合为价格模型所用。

获取商品明细

Getting Details for an Item

搜索结果中的列表给出了商品的名目和价格，也许我们可以从描述商品名目的文字中提取出诸如“容量”或“颜色”这样的详细信息来。eBay 也提供了针对不同商品类型的各种属性信息。一台膝上型电脑可以列举的属性包括处理器类型和所安装的 RAM 等，而一个 iPod 则可以包含诸如体积这样的属性。除了这些细节信息之外，我们还可以获取到诸如卖家评价情况、参与竞价数，以及起步价等信息。

为了得到这些细节信息，我们须要发起一个针对 GetItem 的 eBay API 调用，并提供一个由搜索函数返回的商品 ID 作为参数。为此，请在 *ebaypredict.py* 中新建一个名为 getItem 的函数：

```
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
```

上述函数利用 sendrequest 取到商品的 XML 数据，然后从中解析出感兴趣的部分。由于每件商品的属性都不尽相同，所以它们将被包含在一个字典中一并返回。我们可以针对自己搜索得到的某件商品，尝试执行一下该函数：

```
>>> 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'}
```

从上述结果中我们可以看出，属性 26 444 代表处理器速度，26 446 代表屏幕尺寸，12 代表安装的 RAM，而 25 710 则代表了硬盘尺寸。除了卖家评价情况外，还有参与竞价数，以及起步价，这些内容共同构成了一个具有潜在吸引力的数据集，我们可以将其用于价格预测。

构造价格预测程序
Building a Price Predictor

为了利用我们在本章中构造好的预测程序，我们须要选取一组来自 eBay 的商品数据，并将它们转换成一系列数值列表，以数据集的形式传入交叉验证函数中。为此，下面的 makeLaptopDataset 函数首先调用 doSearch 获取到一个涉及膝上型电脑的商品清单，然后针对其中的每一件商品单独发起请求。利用我们在上一节中确立的商品属性，该函数构造了一个预测用的数值列表，并将这些数据放入了一个 kNN 函数所要求的数据结构中。

请将 makeLaptopDataset 加入 *ebaypredict.py* 中：

```
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
```

在上面的函数里，任何商品如果没有包含必须具备的属性，都将被函数所忽略。整个下载和处理的过程可能会花去一定的时间，但是我们将得到一个很有意思的包含真实价格信息与属性信息的数据集，以供预测之用。为了取得数据，请在你的 Python 会话中调用上述函数：

```
>>> reload(ebaypredict)
<module 'ebaypredict' from 'ebaypredict.py'>
>>> set1=ebaypredict.makeLaptopDataset()
...
```

现在，我们可以选择不同的配置参数，尝试利用 kNN 进行估价了：

```
>>> 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
```

上面给出了一些估价的结果，这些结果受到了不同的 RAM 数量、处理器速度，以及反馈评分的影响。我们可以选择不同的变量值进行试验，对数据按比例缩放，并绘制概率分布。

何时使用 k-最近邻算法

When to Use k-Nearest Neighbors

k-最近邻算法也存在一些不足之处。因为算法须要计算针对每个点的距离，因此预测过程的计算量很大。而且，在一个包含有许多变量的数据集中，我们可能很难确定合理的权重值，也很难决定是否应该去除某些变量。优化可能有助于解决这一问题，但是对于大数据集而言，寻找一个优解可能会花费非常长的时间。

尽管如此，正如你在本章中看到的，kNN 较之其他方法还是有一定优势的。关于预测计算量非常大这一特点，也有其好的一面，那就是我们可以在无须任何计算开销的前提下将新的观测数据加入到数据集中。要正确地解释这一点也很容易，因为我们知道算法是在使用其他观测数据的加权值来进行预测的。

尽管确定权重可能是需要技巧的，但是一旦确定了最佳的权重值，我们就可以凭借这些信息更好地掌握数据集所具备的特征。最后，当我们怀疑数据集中还有其他无法度量的变量时，我们还可以建立概率函数。

练习

Exercises

1. **对近邻数进行优化** 构造一个优化用的成本函数，为简单数据集确定一个理想的近邻数。

2. **留一式（Leave-one-out）交叉验证** 留一式交叉验证是另一种计算预测误差的方法，它将数据集中的每一行单独看作一个测试集，并将数据集的剩余部分都看作训练集。请实现一个完成此功能的函数。将其与本章中介绍的方法试做对比。

3. **削减变量** 除了面对一大群有可能毫无用处的变量，试图去优化它们的缩放比例外，我们还可以在事前先试着消除一些可能会导致预测效果不彰的变量。你能否找到相应的方法呢？

4. **调整概率图形的 ss 值** `probabilityguess` 中的 `ss` 参数指示了概率图的平滑程度。如果这一参数的取值过高会怎样？过低又会如何？你能在不看图形的前提下找到一种办法来确定 `ss` 的合理值吗？

5. **膝上型电脑的数据集** 请尝试为取自 eBay 网站的膝上型电脑的数据集做一下优化。看看哪些变量是重要的？请尝试运行一下绘制概率密度的相关函数。看看是否存在任何值得注意的波峰？

6. **其他商品种类** eBay 上还有哪些其他具有适宜的数值型属性的商品呢？iPod、移动电话，还有汽车，这些商品都包含了许多值得关注的信息。请试着再构造一个数据集，来进行数值型预测。

7. **搜索属性** 许多 eBay API 所具备的功能都没有在本章中介绍到。`GetSearchResults` 调用包含了许多选项，其中有一项是将搜索限制在个别属性范围内。请修改相应的函数以支持这一功能，并试着将查询范围限定在酷睿（Core Duo）膝上型电脑。

第 9 章

高阶分类：核方法与 SVM

Advanced Classification: Kernel Methods and SVMs

前面几章已经探讨了若干种分类器，包括决策树、贝叶斯分类器和神经网络。本章将引入线性分类器和核方法的概念，并以此为铺垫进而向大家介绍一种最为高阶的分类器，同时这也是目前仍然处于活跃状态的一个研究领域，我们称之为**支持向量机**（SVMs）。

本章中出现的数据集所涉及的，几乎都是关于如何为约会网站的用户寻找配对。给定两人的信息，我们能否预测出他们将会成为一对好朋友呢？这是一个值得关注的问题，因为其中包含了许多变量，既有数值型的，也有名词性的，还有大量的非线性关系。我们将选用这样的数据集来为大家示范前述几种分类器的某些缺陷，以及怎样对数据集进行调整才能更好地适应前述这些算法。通过本章我们还会了解到这样一个重要的事实，那就是：将一个复杂数据集扔给一个算法，然后寄希望于它能够学会如何进行精确分类，这几乎是不可能的。选择正确的算法，然后对数据进行适当地预处理，这是要获得满意的分类结果所必需的。笔者希望通过对本章中出现的数据集所做的调整，能够为大家日后调整其他数据集提供有益的启示。

在本章的结尾处，我们将会学习到如何构造一个包含真实人员信息的数据集，这些数据来自 *Facebook*，这是时下流行的一个社区网站。此外还将利用算法来预测：具备某些性格特征的人们是否可能成为好朋友。

婚介数据集

Matchmaker Dataset

本章中将要使用的数据集基于一个假想中的在线约会站点。大多数约会站点都会收集大量与成员有关的有趣信息，其中包括人员基本状况、兴趣爱好，以及行为举止。假设站点收集如下信息。

- 年龄
- 是否吸烟？

- 是否要孩子?
- 兴趣列表
- 家庭住址

不仅如此，这个站点所收集的信息还包括：两个人是否已经成功配对，他们是否已经开始交往，以及是否决定见面。我们就是利用这些数据来构造婚介数据集的。这里有两个文件可供下载：

http://kiwitobes.com/matchmaker/agesonly.csv
http://kiwitobes.com/matchmaker/matchmaker.csv

matchmaker.csv 文件如下所示：

```
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
```

上面每一行数据都对应于一位男士和女士的信息，最后一列用 1 或 0 来代表是否认为两人配对成功。(笔者很清楚此处的题设是经过了简化的；计算机模型总是不及真实生活那么复杂)。对于一个拥有大量用户信息的网站而言，或许可以利用这些信息来构造一个预测算法，帮助用户寻找可以与之配对的其他人。算法或许还可以找出站点目前正欠缺的某些特定类型的人群，这些人在面向新成员的网站推广策略中，将会起到很大的作用。因为两个变量更容易将问题解释清楚，所以 *agesonly.csv* 文件只包含了基于年龄的配对信息，接下来我们将利用这一信息来演示分类器的工作原理。

第一步我们来编写一个加载数据集的函数。该函数的工作仅仅是将所有字段读入一个列表而已，但是出于实验的目的，函数还提供了一个可选的参数，用以有选择地加载个别字段。请新建一个名为 *advancedclassify.py* 的文件，然后将 `matchrow` 和 `loadmatch` 函数加入其中：

```
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
```

loadmatch 的作用是构造一个 matchrow 类的列表，列表中的每一项元素均包含了原始数据，以及有关是否配对成功的信息。下面我们分别利用该函数来加载只包含年龄信息和包含完整信息的婚介数据集：

```
>>> import advancedclassify
>>> agesonly=advancedclassify.loadmatch('agesonly.csv',allnum=True)
>>> matchmaker=advancedclassify.loadmatch('matchmaker.csv')
```

数据中的难点
Difficulties with the Data

上述数据集有两个值得注意的地方，那就是变量的相互作用和非线性特点。如果已经安装了第 8 章中的 *matplotlib*(*http://matplotlib.sourceforge.net*)，那么就可以利用 advancedclassify，对某些变量进行可视化，并从中生成两个包含坐标值信息的列表。（这一步骤对于完成本章剩余部分的内容并不是必需的。）请在你的 Python 会话中尝试之：

```
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()
```

请在你的 Python 会话中调用上述方法：

```
>>> reload(advancedclassify)
<module 'advancedclassify' from 'advancedclassify.py'>
>>> advancedclassify.plotagematches(agesonly)
```

执行上述命令将会生成一个涉及男性与女性年龄对比情况的**散布图**。如果两两匹配，则相应坐标点将标以 O，否则就标以 X。我们将得到一个如下页图 9-1 所示的“窗口”（window）。

尽管很显然还有许多其他因素会对两个人是否成功匹配构成影响，但上图却是根据简化了的只包含年龄信息的数据集绘制而成的，并且它还给出了一条明显的边界，表明人们不会去寻找远远超出其年龄范围内的人进行配对。图上的边界看上去似乎还有些曲折，并且年龄越大边界就越不清晰，这表明人们的年龄越见长就越能忍受更大的年龄差距。

决策树分类器
Decision Tree Classifier

第 7 章提到了决策树分类器，我们利用树来尝试对数据进行自动分类。在那一章中，我们所介绍的决策树算法是根据数值边界来对数据进行划分的。当我们可以借助带有两个变量的函数来更精确地表达分界线（dividing line）时，问题也就随之而来了。在本例中，选择

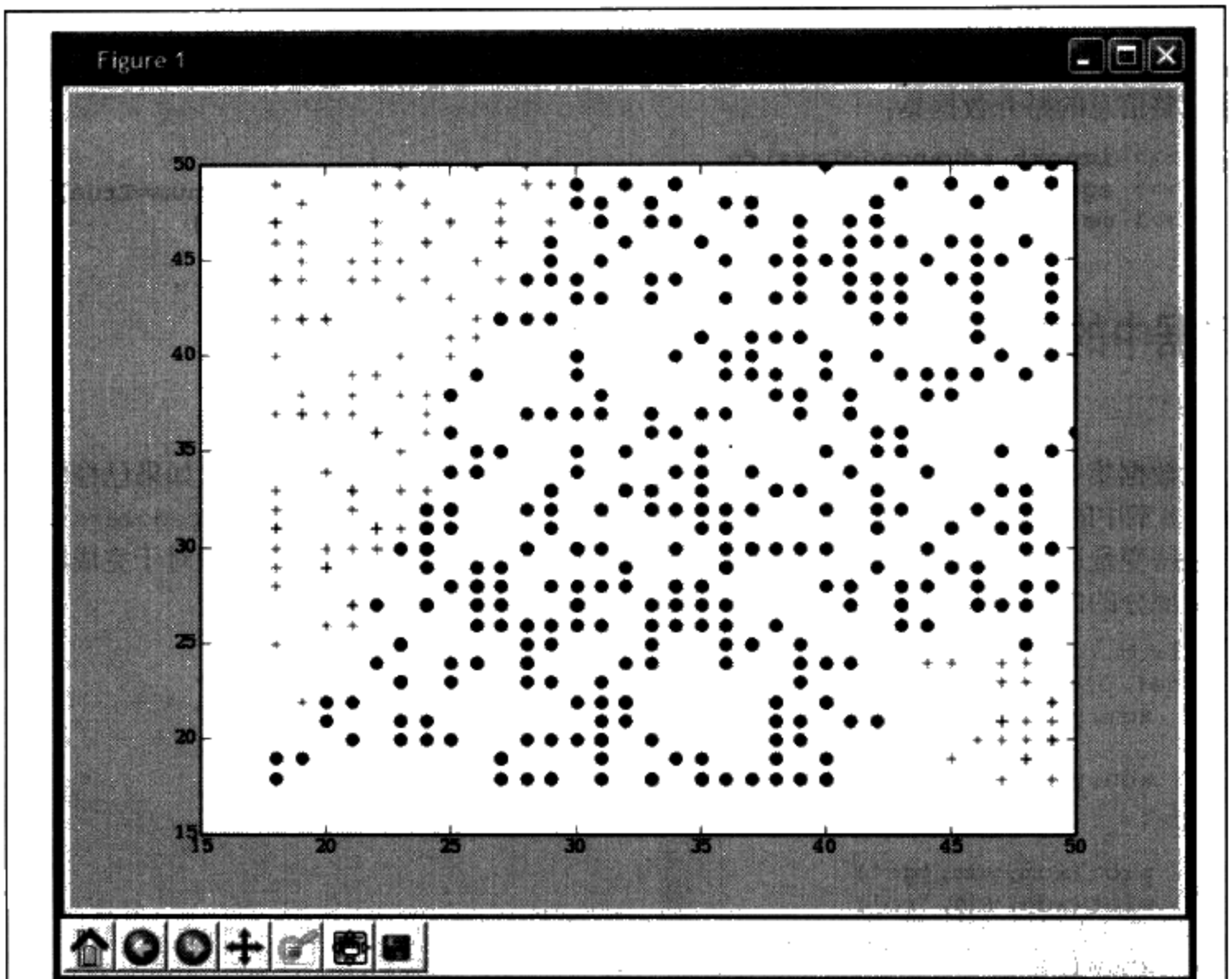


图 9-1：生成的年龄散布图

两人的年龄差作为变量进行预测会更加地稳妥。试想，直接对数据进行决策训练将会得到如图 9-2 所示的结果。

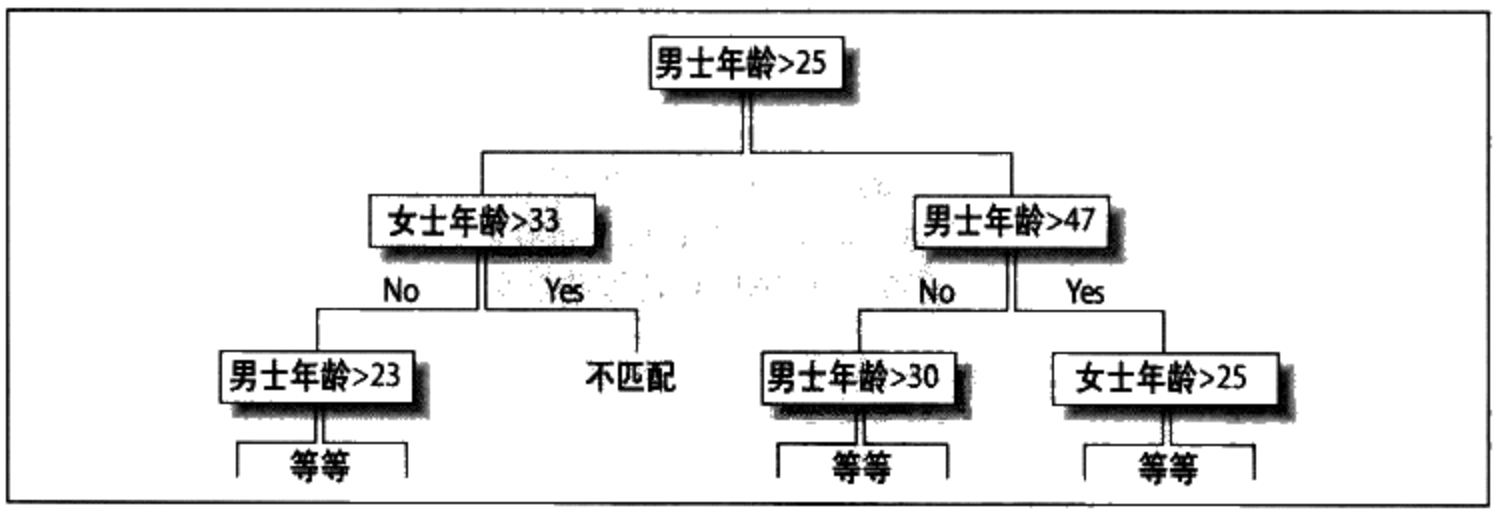


图 9-2：反映曲折边界的决策树

上述结果对于解释决策的过程显然没有任何用处。这棵决策树也许对自动分类会有帮助，但是这样做太麻烦也太死板了。假如我们考虑除年龄之外的其他变量，结果甚至有可能会变得令人更加难以理解。为了明白决策树到底做了些什么，让我们来看一下散布图，以及根据决策树生成的决策边界，如图 9-3 所示。

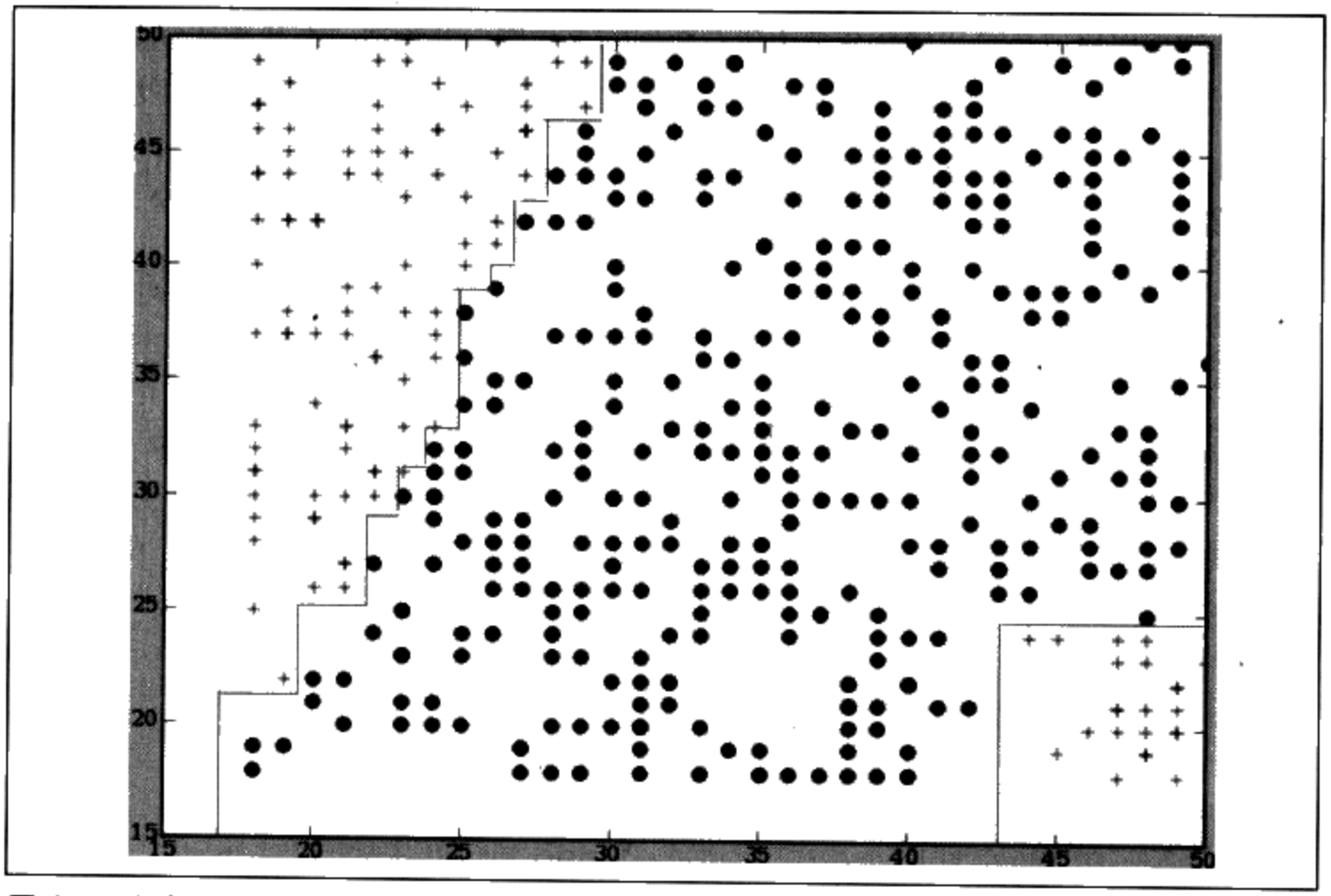


图 9-3：根据决策树生成的边界线

决策边界是这样一条线：位于这条线一侧的每一个点会被赋予某个分类，而位于另一侧的每一个点会被赋予另一个分类。从图中我们可以清晰地看到，决策树的约束条件使边界线呈现出垂直或水平向的分布。

此处有两个要点。其一是，在没有弄清楚数据本身的含义及如何将其转换成更易于理解的形式之前，轻率地使用提供给我们的数据是错误的。建立散布图有助于我们找到数据真正的划分方式。其二是，尽管第 7 章介绍的决策树有其自身的优势，但是在确定问题的分类时，如果存在多个数值型输入，且这些输入彼此间所呈现的关系并不简单，决策树则常常不是最有效的方法。

基本的线性分类

Basic Linear Classification

虽然线性分类是分类器中最简单的一种，但是这对于我们的后续讨论是一个很好的基础。线性分类的工作原理是寻找每个分类中所有数据的平均值，并构造一个代表该分类中心位置的点。然后我们就可以通过判断距离哪个中心点位置最近来对新的坐标点进行分类了。

为了实现上述功能，我们首先需要一个函数来计算分类的**均值点**（average point）。在本例中，所谓的分类就是 0 和 1。请将 `lineartrain` 加入 *advancedclassify.py* 中：

```
def lineartrain(rows):
  averages={}
  counts={}

  for row in rows:
    # 得到该坐标点所属的分类
    cl=row.match

    averages.setdefault(cl,[0.0]*(len(row.data)))
    counts.setdefault(cl,0)

    # 将该坐标点加入 averages 中
    for i in range(len(row.data)):
      averages[cl][i]+=float(row.data[i])

    # 记录每个分类中有多少坐标点
    counts[cl]+=1

  # 将总和除以计数值以求得平均值
  for cl,avg in averages.items():
    for i in range(len(avg)):
      avg[i]/=counts[cl]

  return averages
```

我们可以在自己的 Python 会话中运行上述函数，以求得平均值：

```
>>> reload(advancedclassify)
<module 'advancedclassify' from 'advancedclassify.pyc'>
>>> avgs=advancedclassify.lineartrain(agesonly)
```

为了明白线性分类所起的作用，我们再来看一看年龄数据的分布图，如下页图 9-4 所示。

图中的 X 表示由 `lineartrain` 计算求得的均值点。划分数据的直线位于两个 X 的中间位置。这意味着，所有位于直线左侧的坐标点都更接近于表示“不相匹配（no match）”的均值点，而所有位于右侧的坐标点则都更接近于表示“相匹配（match）”的均值点。任何时候当我们遇到一对新的年龄数据时，如果想要推测二者是否相匹配，我们只须将其想象成上图中的一个坐标点，并判断其更接近于哪个均值点即可。

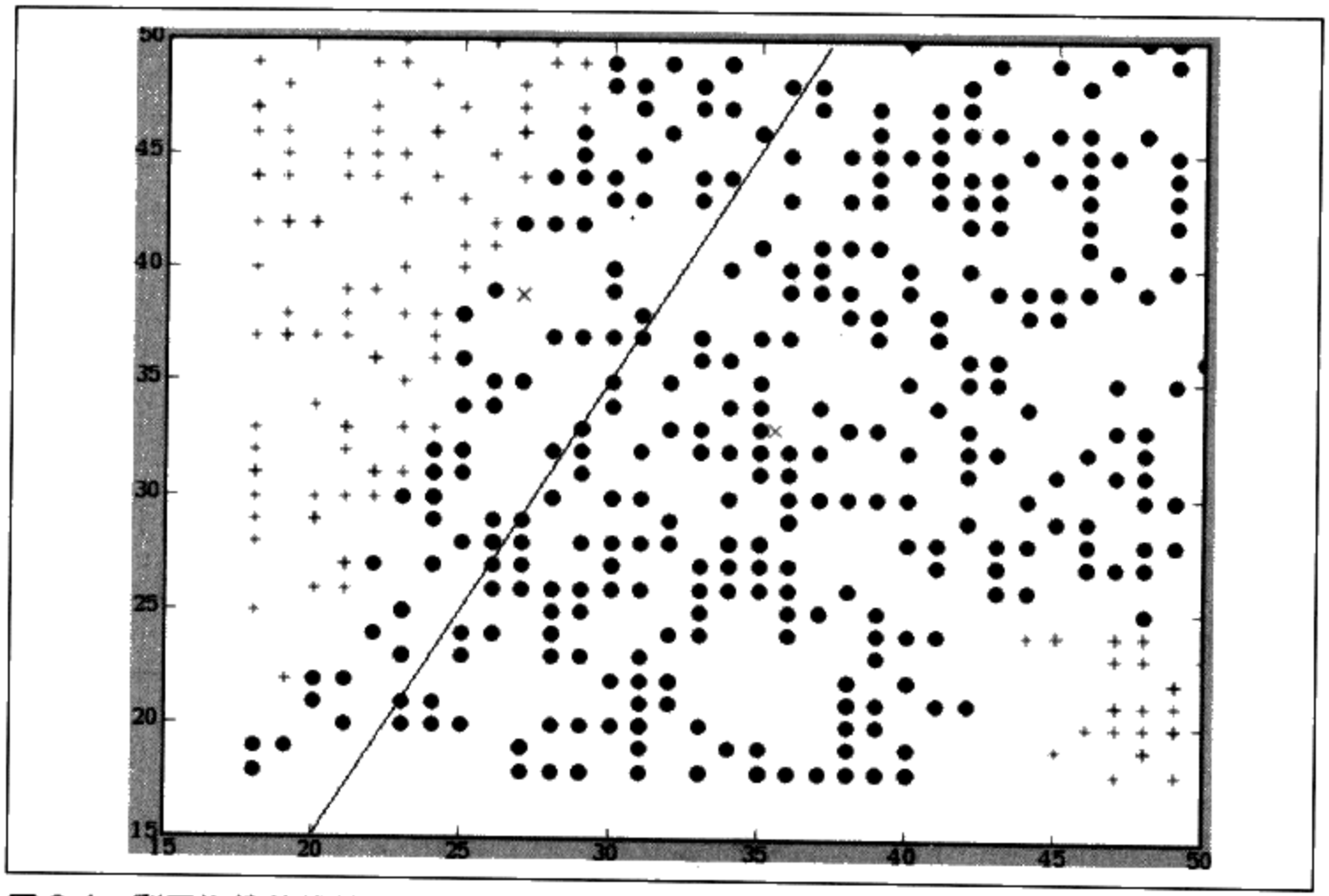


图 9-4：利用均值的线性分类器

有多种方法可以判定一个坐标点距离均值点的远近程度。前面几章介绍了欧几里德距离的相关知识；一种方法就是，先计算坐标点到每个分类的均值点的距离，然后从中选择距离较短者。尽管可以将该方法用于此处的分类器，但是为了方便后面的扩展，我们将采用另一种不同的方法：使用向量和点积。

向量具有大小和方向，我们时常将其表示成平面上的一个箭头，或者记作一个数字集合。下页图 9-5 给出了一个向量的例子。图中还示范了如何用一个点减去另一个点，以得到连接两者的向量。

所谓点积是指，针对两个向量，将第一个向量中的每个值与第二个向量中的对应值相乘，然后再将所得的每个乘积相加，最后得到一个总的结果。请在 *advancedclassify.py* 中新建一个名为 dotproduct 的函数：

```
def dotproduct(v1,v2):
  return sum([v1[i]*v2[i] for i in range(len(v1))])
```

点积也可以利用两个向量的长度乘积，再乘以两者夹角的余弦求得。最重要的是，如果夹角的度数大于 90 度，则夹角余弦值为负，这意味着此时的点积结果也为负值。为了明白我们是如何利用点积的这一计算特征的，请看下页图 9-6。

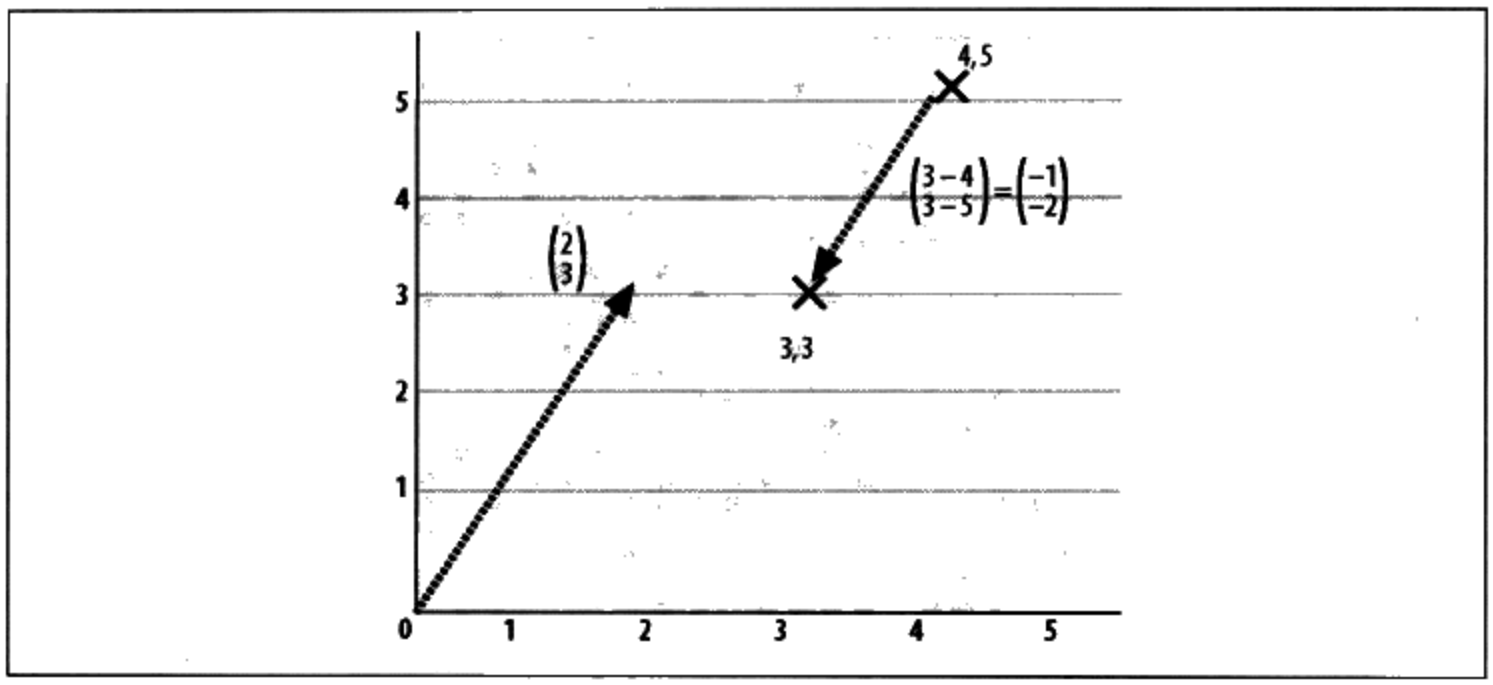


图 9-5：向量的例子

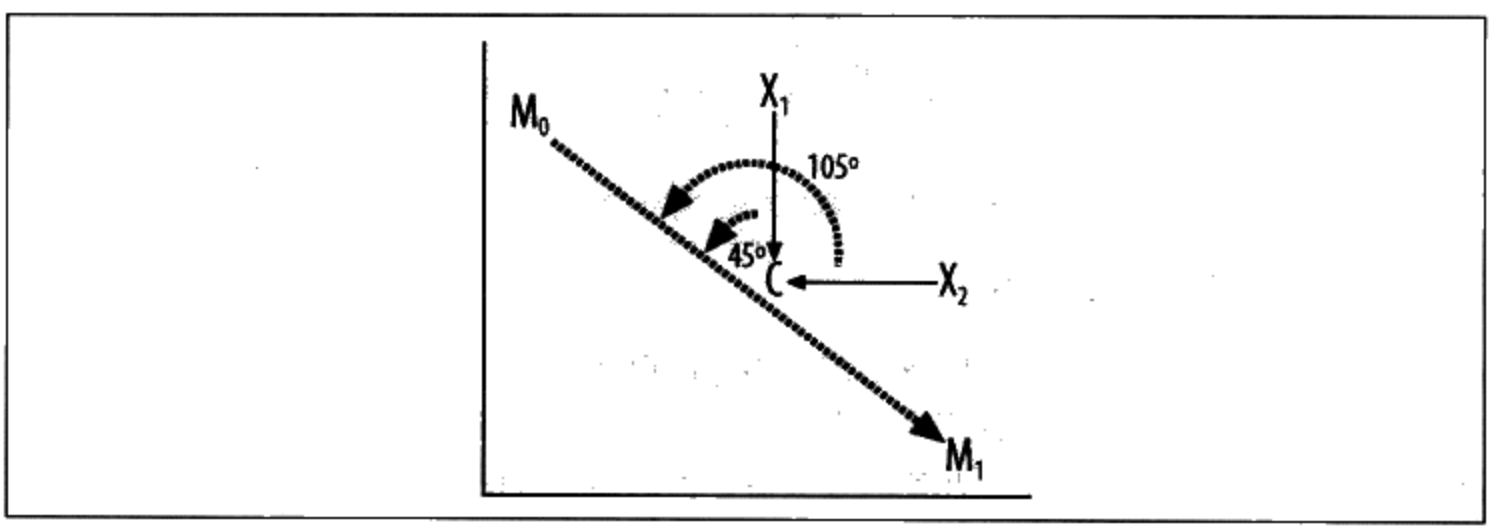


图 9-6：利用点积来确定距离

在图中，我们看到有两个均值点，分别对应于“相匹配”（M_0）和“不相匹配”（M_1）两种情况，以及一个位置介于 M_0 与 M_1 中间的 C。另外还有两个点，X_0 和 X_1，它们是即将要被分类的两个例子。除此以外，图上还显示了连接 M_0 到 M_1 的向量，以及连接 X_1 到 C 和 X_2 到 C 的两个向量。

在图中，X_1 更接近于 M_0，因此它应该被划归为“相匹配”。我们注意到，介于向量 $X_1 \rightarrow C$ 和 $M_0 \rightarrow M_1$ 的夹角为 45 度，小于 90 度，因此 $X_1 \rightarrow C$ 与 $M_0 \rightarrow M_1$ 的点积结果为正数。

而由于向量 $X_2 \rightarrow C$ 和 $M_0 \rightarrow M_1$ 的指向相反，因此介于两者间的夹角大于 90 度。即 $X_2 \rightarrow C$ 和 $M_0 \rightarrow M_1$ 的点积结果为负数。

夹角大者点积为负，夹角小者点积为正，因此只须通过观察点积结果的正负号，就可以判断出新的坐标点属于哪个分类。

因为点 C 是 M_0 和 M_1 的均值点，亦即（M_0+M_1）/2，因此寻找分类的公式如下：

```
class=sign((X - (M0+M1)/2) . (M0-M1))
```

相乘后的结果为：

```
class=sign(X.M0 - X.M1 + (M0.M0 - M1.M1)/2)
```

我们将利用上述公式来确定分类。请将一个名为dpclassify的新函数加入*advancedclassify.py*中：

```
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
```

现在，我们可以在自己的 Python 会话中利用线性分类器，从前述数据中尝试得出一些结论了：

```
>>> 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
```

请记住这是一个线性分类器，所以它只找出了一条分界线来。这意味着，如果找不到一条划分数据的直线来，或者如果实际存在多条直线时，就如同前面那个年龄对比的例子中所呈现的那样，那么此时分类器将会得到错误的答案。在那个例子中，48 岁与 20 岁的年龄对比实际上应该是“不相匹配”的结果，但是因为我们只找到了一条直线，而相应的坐标点又落在了这条直线的右侧，所以函数得出了“相匹配”的结论。在本章稍后的“理解核方法”一节中，将会学习如何对这一分类器进行改进，使其能够处理非线性分类。

分类特征
Categorical Features

婚介数据集中既包含有数值数据，也包含有分类数据（categorical data）。一些像决策树这样的分类器不须要对数据作任何预处理，就可以同时应对这两种类型的数据，但是本章接下来要介绍的分类器只能处理数值型数据。为了解决这一问题，我们需要一种能够将数据转换成数值类型的方法，以使得分类器能够处理这些数据。

是/否问题

Yes/No Questions

将数据转换成数值类型的一个最简单的例子就是“是/否”问题了，因为可以将“是”转换为 1，将“否”转换为-1。这样做还为我们留出了一个选择：可以将缺失的或模棱两可的数据（比如“我不清楚”）转换为 0。请将转换函数 `yesno` 加入 *advancedclassify.py*：

```
def yesno(v):
  if v=='yes': return 1
  elif v=='no': return -1
  else: return 0
```

兴趣列表

Lists of Interests

有多种不同的方法可以在数据集中记录下人们的兴趣爱好。其中最为简单的方法就是将每一种可能的兴趣爱好都视作单独的数值变量，如果人们具备某一项兴趣就令其为 0，否则就令其为 1。假如要处理的是一个一个的人，那么这样做是最为合理的。然而在本例中，我们所处理的是一对一对的人，因此一种更为直观的方法是将具备共同兴趣爱好的数量视作变量。

请将一个名为 `matchcount` 的新函数加入 *advancedclassify.py* 中，它以浮点数的形式返回列表中匹配项的数量：

```
def matchcount(interest1,interest2):
  l1=interest1.split(':')
  l2=interest2.split(':')
  x=0
  for v in l1:
    if v in l2: x+=1
  return x
```

将具备共同兴趣爱好的数量作为变量是很有意义的，但是这样做的确也忽视了某些富有价值的潜在信息。某些不同兴趣爱好的组合也有可能会配对成功，比如：滑雪与滑板、饮酒与跳舞。一个分类器假如不是在未经处理的原始数据上进行训练，它是无法“学习”到这样的组合的。

如果为每一项兴趣都新建一个变量，则会导致为数众多的变量出现，从而使分类器变得更加复杂，改变这一情况的一种办法是将兴趣爱好按层级排列。比如说，滑雪和滑板都属于雪地运动，而雪地运动又属于体育运动的子分类；如果两个人都对雪地运动感兴趣，但具体感兴趣的并不是同一个项目，那么其 `matchcount` 值也许会加上 0.8，而不是满分 1。如果为寻找匹配而遍历的层级越靠近上层，那么相应赢取的分值也就越小。尽管婚介数据集并没有这样一个层级关系，但是当有类似问题出现的时候，不妨也可以考虑一下采用这样的办法。

利用 Yahoo! Maps 来确定距离

Determining Distances Using Yahoo! Maps

数据集中最难处理的莫过于家庭住址了。两个人住得越近是否就越有可能匹配成功，这一点很值得考量，但是数据文件中的住址信息是以地址和邮编的混合形式给出的。解决这一问题的一个非常简单的办法，是将“居住地邮编是否相同”视作一个变量，但是这样做有非常大的局限性——居住在邻近地区的人，其邮编很有可能是不同的。理想情况下，我们可以根据距离值来建立一个新的变量。

当然，在没有额外信息补充的情况下，我们是无法计算两地间的距离的。所幸的是，Yahoo! Maps 提供了一个叫做 *Geocoding* 的 API 服务，它可以接受一个美国范围内的地址，然后返回相应的经度和纬度。针对两个不同的地址调用该项服务，就能计算出两地间的大概距离了。

假如有任何原因使我们无法使用 Yahoo! API，就请将一个名为 `milesdistance` 的空函数加入到 *advancedclassify.py* 中：

```
def milesdistance(a1,a2):
  return 0
```

获取 Yahoo!的应用密钥

为了使用 Yahoo! API，首先须要获得一个应用密钥，我们将该密钥用于发起的查询请求中，以此来标识我们的应用程序。通过访问 *http://api.search.yahoo.com/webservices/register_application* 并回答一系列问题，就可以拿到一个密钥。如果你还没有 Yahoo!账号，那就必须先注册一个。待注册完毕之后就可以立即得到一个密钥，而不必等待 E-mail 的回应。

使用 Geocoding API

Geocoding API 要求我们以指定的 URL 格式来发起请求。可以采用形如 *http://api.local.yahoo.com/MapsService/V1/geocode?appid=appid&location=location* 这样的格式来构造 URL。

其中的 location 是一段随意的文本串，它可以是一个地址、一个邮编，甚或是一个城市名加一个州名。请求所返回的结果是一个如下所示的 XML 文件：

```
<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>
```

此处我们所关注的字段是 longitude 和 latitude。为了解析这个文件，我们将使用前面章节中曾经用到过的 minidom API。请将 `getlocation` 添加到 *advancedclassify.py* 中：

```
yahookey="Your Key Here"
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]
```

上述函数利用我们的应用密钥和位置信息构造了一个 URL，然后提取并返回了相应的经纬度结果。尽管这一结果是计算距离所需的唯一信息，但还是可以利用 Yahoo! Geocoding API 来做许多其他事情的，比如：可以确定给定地址的邮编，或是找出某个邮编的所处位置。

计算距离

如果要非常精确地将两个坐标点的经纬度转换成以英里计量的距离值，这其实是一项极具技巧性的工作。不过，本章例子中涉及的距离值非常小，而计算距离的目的只是为了做比较，因此完全可以用近似值来替代。这个近似值类似于在前几章中见到的欧几里德距离，不同之处在于将纬度之间的差值乘以 69.1，将经度之间的差值乘以 53。

请将 `milesdistance` 函数添加到 *advancedclassify.py* 中：

```
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
```

该函数针对两个不同的地址分别调用了此前定义的 `getlocation` 函数，随后函数计算了这两个地址间的距离。如果愿意的话，也可以在自己的 Python 会话中尝试执行一下上述函数：

```
>>> 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
```

利用近似法求得的距离值一般会有小于 10%的误差，这对于本章的应用而言应该是可以接受的。

构造新的数据集

Creating the New Dataset

现在已经做好了所有的准备工作，这些都是构造用于训练分类器所需的数据集所必需的。我们需要一个函数将所有这些部件组装起来。该函数将利用 `loadmatch` 函数从数据文件中加载数据集，并对各列数据进行适当的变换处理。请将 `loadnumerical` 加入 *advancedclassify.py* 中：

```
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
```

上述函数针对原始数据集的每一行生成一个新的数据行。它调用我们先前定义好的函数，将所有数据转换成新的值，其中包括距离值的计算，以及相同兴趣爱好的统计。

请在你的 Python 会话中调用上述函数，以构造新的数据集：

```
>>> reload(advancedclassify)
>>> numericalset=advancedclassify.loadnumerical()
>>> numericalset[0].data
[39.0, 1, -1, 43.0, -1, 1, 0, 0.90110601059793416]
```

同样，此处可以很容易地通过指定自己感兴趣的列来构造子数据集。这对于数据的可视化呈现，以及理解分类器针对不同变量的执行过程是很有帮助的。

对数据进行缩放处理

Scaling the Data

当我们只是根据人们的年龄进行对比时，保留数据的原有状态并使用均值和距离值，是没有任何问题的，因为将代表同一事物的变量放在一起对比是很合乎情理的。然而，现在已经引入了一些新的变量，这些变量与年龄事实上没有什么可比性，相对而言它们的值要小很多。双方对于是否要小孩所持的不同观点——介于 1 与-1 之间，最大差值为 2——在现实中也许要比一个 6 岁的年龄差距更有意义得多，但是如果使用目前的数据，年龄之差将会被视为 3 倍于观念之差。

为了解决这一问题，一种推荐的做法是，将所有数据都缩放为同一尺度，从而使每个变量上的差值都具有可比性。通过确定每个变量的最大最小值，并对数据进行相应的缩放，以使最小值为 0，最大值为 1，其他值介于 0 和 1 之间，就可以在相同尺度上对数据进行比较了。

请将 scaledata 添加到 *advancedclassifier.py* 中：

```
def scaledata(rows):
  low=[999999999.0]*len(rows[0].data)
  high=[-999999999.0]*len(rows[0].data)
  # 寻找最大值和最小值
  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]

  # 对数据进行缩放处理的函数
  def scaleinput(d):
    return [(d.data[i]-low[i])/(high[i]-low[i])
            for i in range(len(low))]

  # Scale all the data
  # 对所有数据进行缩放处理
  newrows=[matchrow(scaleinput(row.data)+[row.match])
          for row in rows]

  # 返回新的数据和缩放处理函数
  return newrows,scaleinput
```

上述函数定义了一个内部函数，scaleinput，其作用是找出最小值，并从所有数值中减去该最小值，从而将值域范围调整至以 0 为起点。函数随后又将调整后的结果除以最大最小值的差，从而将所有数据都转换成了介于 0 和 1 之间的值。该函数针对数据集中的每一行数据分别调用了 scaleinput 函数，并将新生成的数据集与 scaleinput 函数一并返回，从而使我们对于查询得到的结果也可以作出同样的调整。

现在，可以针对更大规模的变量组合，来尝试线性分类器了：

```
>>> 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
```

请注意，须要先对样例数据进行缩放处理，将其调整至新的值域空间。尽管此处所用的线性分类器对于某些例子是有效的，但是只试图寻找一条分界线的局限性现在已经变得越来越明显了。为了有所改进，我们需要一种能够超越线性分类的新的分类方法。

理解核方法

Understanding Kernel Methods

试想一下，假如我们尝试对如图 9-7 所示的数据集施以线性分类会如何。

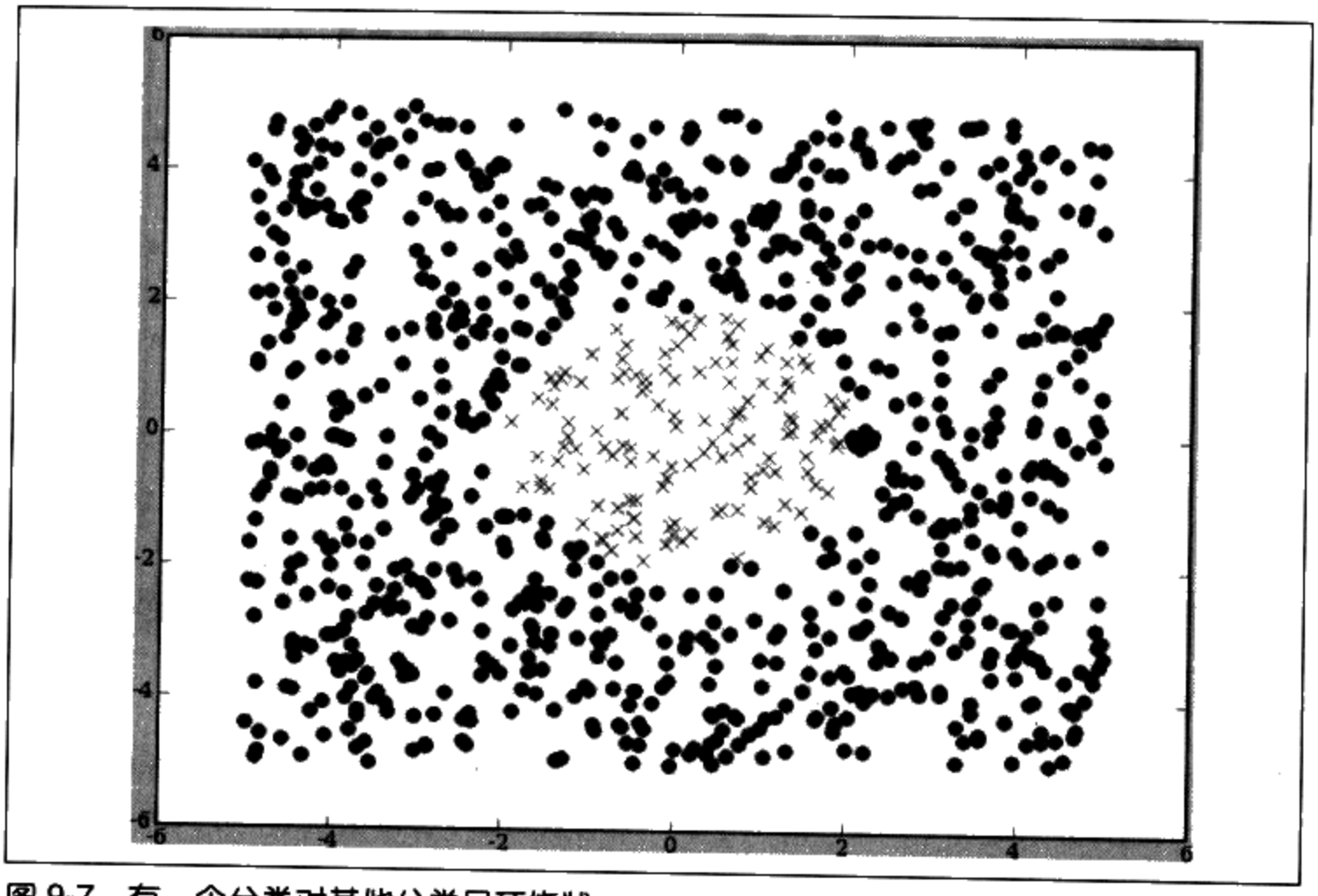


图 9-7：有一个分类对其他分类呈环绕状

每个分类的均值点在哪儿呢？它们恰好都位于相同的位置！尽管大家都很清楚，任何位于圆圈内的都是 X，位于圆圈外的都是 O，但是线性分类器却无法识别这两个分类。

不过请试想一下，假如先对每一个 x 值和 y 值求平方，情况又会如何呢。原来位于(-1,2)处的坐标点现在将变成(1,4)，原来位于(0.5,1)处的坐标点现在将变成(0.25,1)，依此类推。新的坐标点分布情况如下页图 9-8 所示。

所有的 X 现在都已经偏移到了图上的角落处，所有的 O 都则位于角落以外的区域。现在，要用一条直线来划分 X 和 O 已经变得非常容易了，并且任何时候只要有待分类的新数据，我们只须对 x 值和 y 值求平方，并观察其落于直线的哪一侧即可。

上述例子告诉我们，通过预先对坐标点进行变换，构造一个只用一条直线就可以进行划分的新数据集是完全有可能的。然而，在这里之所以选择这样一个例子，是因为它非常容易变换；在面对真实问题时，变换的方法可能要复杂许多，其中就包括数据的多维变换。例

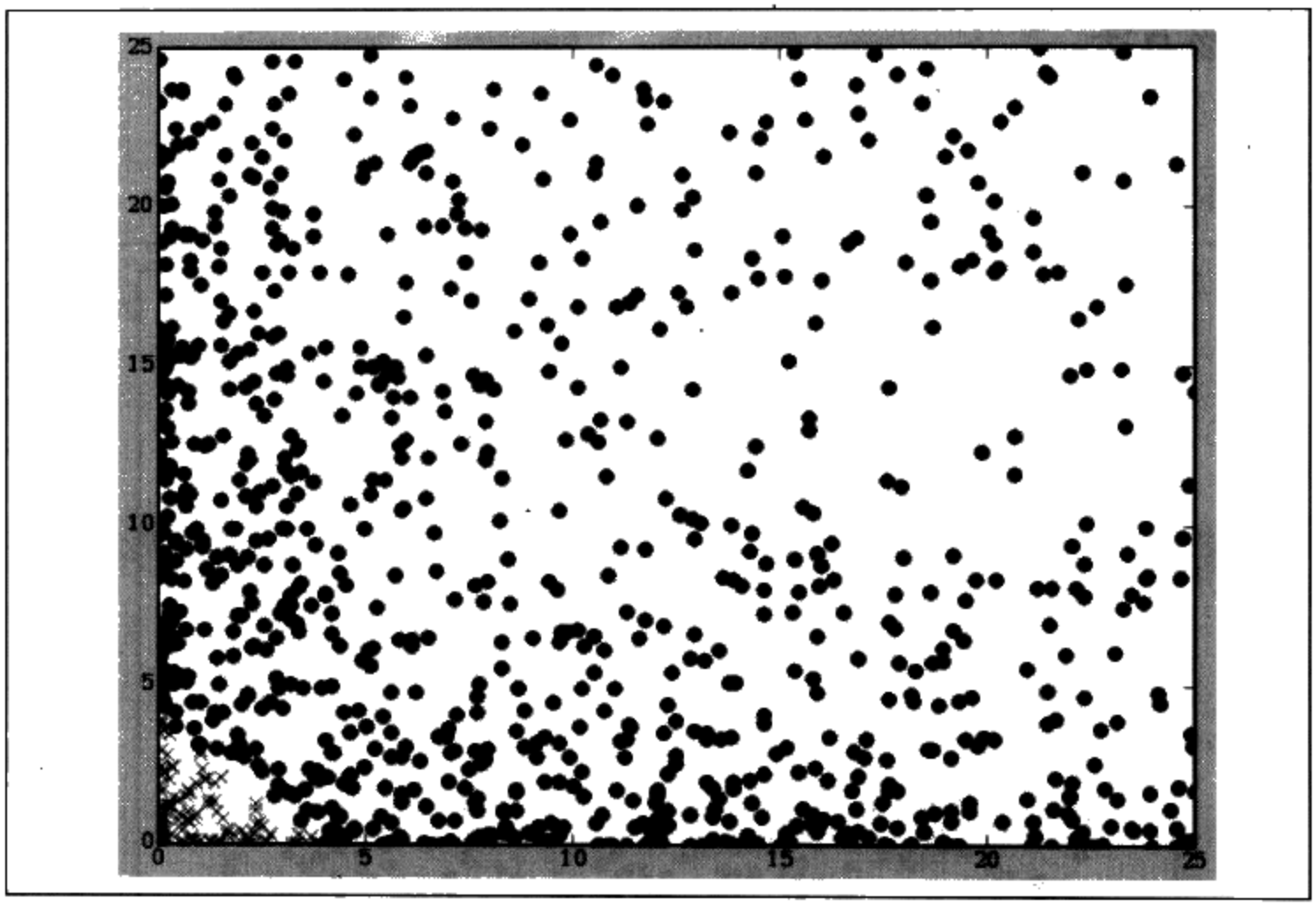


图 9-8：将坐标点移至新的坐标空间

如，或许我们会将一个包含 x 和 y 坐标的数据集，变换成一个由 a、b、c 三个坐标构成的新数据集，其条件分别是 $a=x^2$、$b=x*y$、$c=y^2$。一旦将数据置于多维空间中，寻找两个分类间的分界线就容易多了。

核技法

The Kernel Trick

尽管可以编写代码将数据依照如上方法变换到新的坐标空间中，但实际上我们通常不会这样去做，因为要找到一条与实际数据集相匹配的分界线，必须要将数据投影到成百上千个维度上，要实现这样的功能是非常不切实际的。然而，对于任何用到了点积运算的算法——包括线性分类器——我们可以采用一种叫做**核技法**的技术。

核技法的思路是用一个新的函数来取代原来的点积函数，当借助某个映射函数将数据第一次变换到更高维度的坐标空间时，新函数将会返回高维度坐标空间内的点积结果。此处，就变换方法的数量而言并没有任何的限制，但是在现实中我们其实只会采用少数几种变换方法。其中备受人们推崇的一种方法（也是将要在此处采用的方法）被称为**径向基函数**（radial-basis function）。

径向基函数与点积类似，它接受两个向量作为输入参数，并返回一个标量值。与点积不同的是，径向基函数是非线性的，因而它能够将数据映射到更为复杂的空间中。请将 rbf 加入 *advancedclassify.py* 中：

```
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)
```

该函数还接受一个 gamma 参数，我们可以对该参数进行调整，以得到一个针对给定数据集的最佳线性分离。

现在，我们需要一个新的函数，用以计算坐标点在变换后的空间中与均值点间的距离。遗憾的是，目前的均值点是在原始空间中计算得到的，因此无法在此处直接使用它们——事实上，根本无法计算均值点，因为实际上不会在新的坐标空间中计算坐标点的位置。所幸的是，先对一组向量求均值，然后再计算均值与向量 A 的点积结果，与先对向量 A 与该组向量中的每一个向量求点积，然后再计算均值，在效果上是完全等价的。

因此，我们不再对尝试分类的两个坐标点求点积，也不再计算某个分类的均值点了，取而代之的是，计算出某个坐标点与分类中其余每个坐标点之间的点积或径向基函数的结果，然后再对它们求均值。请将 nonlinearclassify 加入 *advancedclassify.py* 中：

```
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
```

此处的偏移量（offset）参数，在转换后的空间中也会发生改变，而其计算过程可能会有些费时。因此，我们应该预先为某个数据集计算一次偏移量，然后在每次调用 nlclassify 时将其传入。

下面让我们来尝试一下新的分类器，只考虑年龄因素，看看它是否解决了之前提到的问题：

```
>>> 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
```

非常棒！对数据的变换处理，使我们的分类器能够识别出一个年龄匹配的环状分布，位于该区域内的年龄彼此都非常的接近，而该区域外的任何一侧，这样的匹配都是极不可能的。现在，我们的分类器已经可以识别出：48 岁与 20 岁是不会成功配对的。接下来，请将除年龄以外的其他数据也包含进来，再试一试：

```
>>> 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] # 男士不想要小孩，而女士想要
>>> advancedclassify.nlclassify(scalef(newrow),scaledset,ssoffset)
0
>>> newrow=[28.0,-1,1,26.0,-1,1,2,0.8] # 双方都想要小孩
>>> advancedclassify.nlclassify(scalef(newrow),scaledset,ssoffset)
1
```

此处分类器的性能改善了不少。从中我们可以看到，假如男士不想要小孩而女士想要的话，那么一段好姻缘就会被断送，即便双方的年龄都很接近，即便双方还拥有不少的共同爱好也无济于事。请改用其他的变量试一试，看看对结果的影响如何。

支持向量机
Support-Vector Machines

让我们再次回到前面的话题，当寻找一条划分两个分类的直线时，我们所面对的困境。图9-9给出了这样的一个例子。每个分类的均值点及隐含的分界线如图所示。

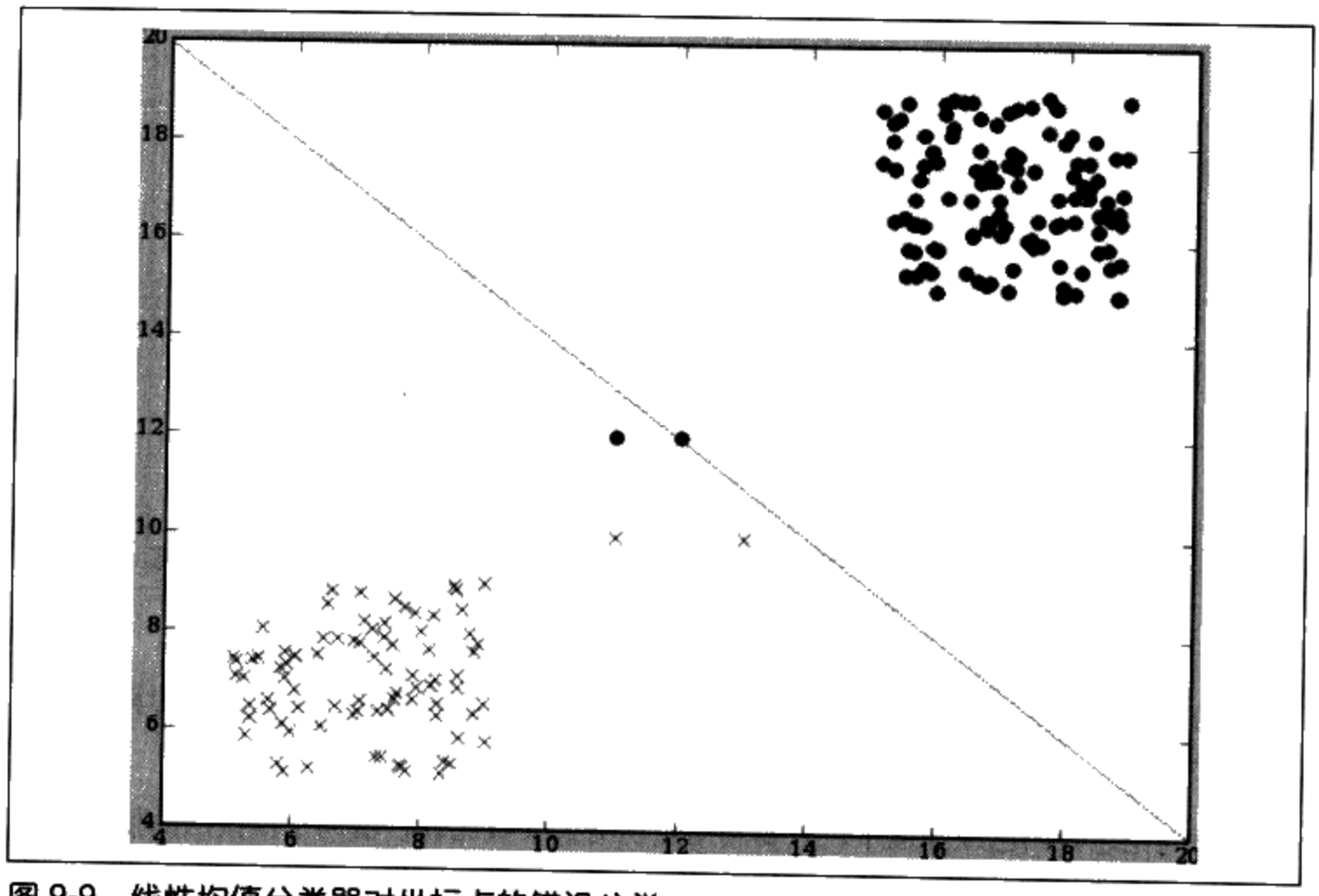


图 9-9：线性均值分类器对坐标点的错误分类

我们注意到，有两个坐标点因为与其他大多数数据相比，都更加接近于利用均值点计算得到的分界线，所以它们被划归到了错误的分类。此处的问题在于，因为大多数数据都是远离分界线的，所以判断坐标点的分类与是否位于直线的某一侧并没有太大的关系。

支持向量机是广为人知的一组方法的统称，借助于它我们可以构造出解决上述问题的分类器。其思路就是，尝试寻找一条尽可能远离所有分类的线，这条线被称为**最大间隔超平面**（maximum-margin hyperplane），如下页图9-10所示。

此处选择分界线的依据是：寻找两条分别经过各分类相应坐标点的平行线，并使其与分界线的距离尽可能的远。同样，对于新的数据点，可以通过观察其位于分界线的哪一侧来判断其所属的分类。请注意，只有位于间隔区边缘的坐标点才是确定分界线位置所必需的；我们可以去掉其余所有的数据，而分界线依然还会处于相同的位置。我们将位于这条分界

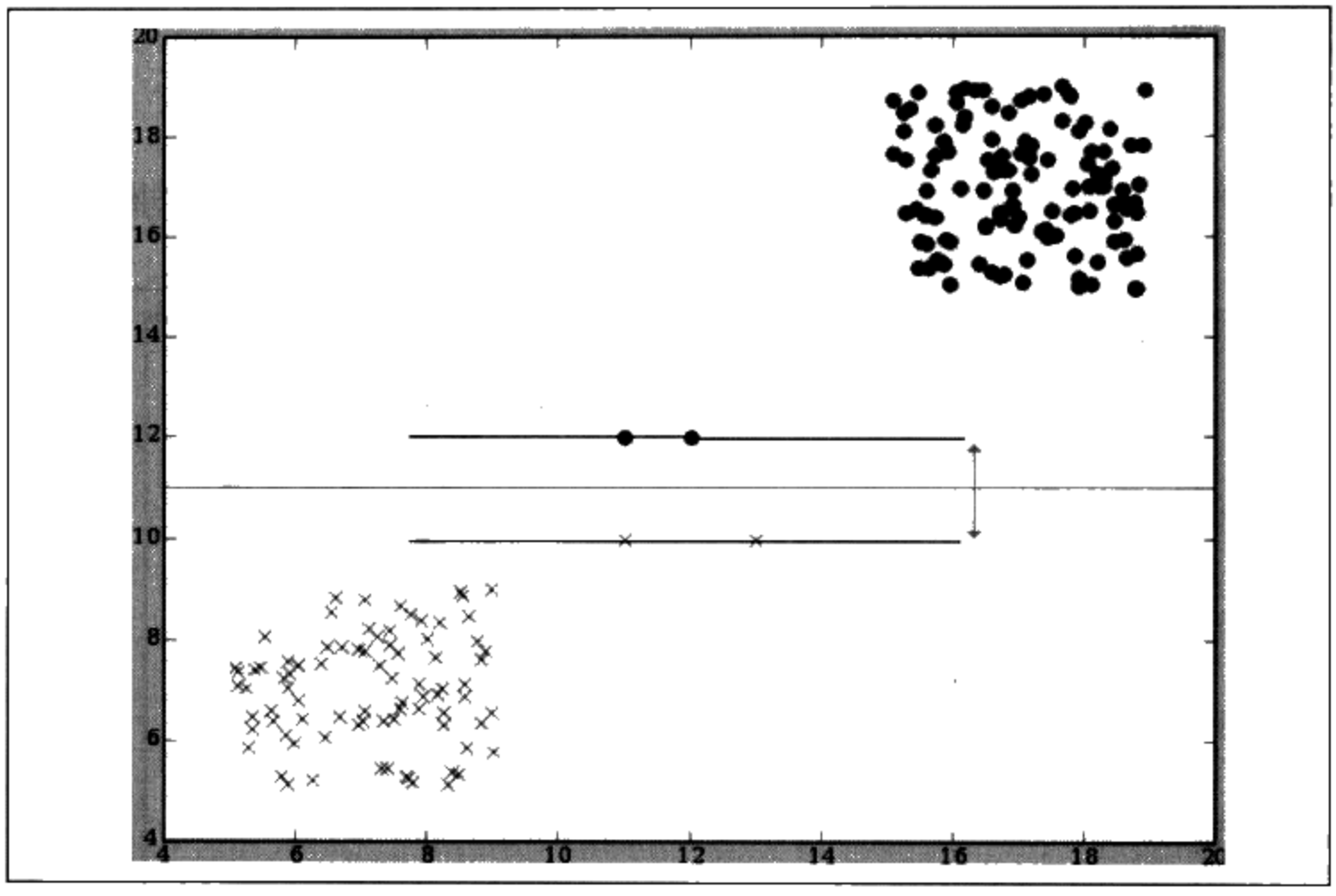


图 9-10：寻找最佳分界线

线附近的坐标点称做**支持向量**。寻找支持向量，并利用支持向量来寻找分界线的算法便是支持向量机。

通过前面的论述我们已经知道了，只要利用点积结果来做比较，借助于核技法的使用，就可以将一个线性分类器转换成非线性分类器。支持向量机所使用的也是点积的结果，因此同样也可以利用核技法将其用于非线性分类。

支持向量机的应用

因为支持向量机在高维数据集上有不错的表现，因此它们时常被用于解决数据量很大（data-intensive）的科学问题，以及其他须要处理极复杂数据集的问题。其中的一些例子如下所示。

- 对面部表情进行分类。
- 使用军事数据侦测入侵者。
- 根据蛋白质序列预测蛋白质结构。
- 笔迹识别。
- 确定地震期间的潜在危害。

使用 LIBSVM

Using LIBSVM

相信前一节的讨论对于大家理解支持向量机的工作方式和工作原理会有很大的帮助，但是训练支持向量机的算法所涉及的数学概念，其计算量是非常庞大的，而且也超出了本章讨论的范畴。因此，在这一节中我们将引入一个叫做 *LIBSVM* 的开源库，它能够对一个 SVM 模型进行训练、给出预测，并利用数据集对预测结果进行测试。LIBSVM 甚至还提供了针对径向基函数和许多其他核方法的支持。

获取 LIBSVM

Getting LIBSVM

我们可以从 *http://www.csie.ntu.edu.tw/~cjlin/libsvm* 处下载到 LIBSVM。

LIBSVM 是用 C++编写而成的，它还有一个 Java 的版本。该下载包中包含了一个 Python 的封装程序，叫做 *svm.py*。为了使用 *svm.py*，我们须要根据自己所使用的平台选择合适的 LIBSVM 编译版本。如果你使用的是 Windows 平台，则须要将一个名为 *svmc.dll* 的 DLL 文件包含进来。(Python 2.5 要求我们将该文件重命名为 *svmc.pyd*，因为它无法导入带 DLL 扩展名的库)。LIBSVM 的文档里还详细说明了如何针对其他平台来生成函数库的编译版本。

一个 Python 会话的例子

A Sample Session

一旦得到了 LIBSVM 的编译版本，就可以将其与 *svm.py* 一起置于 Python 的安装路径或工作目录之下。我们可以在自己的 Python 会话中将该库导入进来，并尝试解决一个简单的问题：

```
>>> from svm import *
```

首先让我们来构造一个简单的数据集。LIBSVM 从一个包含两个列表的元组中读取数据。其中，第一个列表含有分类数据，第二个列表含有输入数据。请尝试用两个分类来构造一个简单的数据集：

```
>>> prob = svm_problem([1,-1],[[1,0,1],[-1,0,-1]])
```

此外，我们还须要通过构造 svm_parameter 来指定所选用的核方法：

```
>>> param = svm_parameter(kernel_type = LINEAR, C = 10)
```

接下来，就可以对模型进行训练了：

```
>>> 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
```

最后，利用该模型来预测新的分类：

```
>>> m.predict([1, 1, 1])
1.0
```

上述例子为我们展示了 LIBSVM 的所有相关功能，这些都是我们根据训练数据构造模型，并借此给出预测所必需的。此外，LIBSVM 还提供了另一项很方便的功能，那就是加载和保存构造好的受训模型：

```
>>> m.save(test.model)
>>> m=svm_model(test.model)
```

将 SVM 用于婚介数据集

Applying SVM to the Matchmaker Dataset

为了将 LIBSVM 用于婚介数据集，必须先将数据集 `scaledset` 转换成 `svm_model` 所要求的列表元组。转换的过程非常简单，我们在 Python 会话中用一行代码就可以完成：

```
>>> answers,inputs=[r.match for r in scaledset],[r.data for r in scaledset]
```

同样，为了避免过高估计某些变量所起的作用，此处我们使用了经缩放处理的数据。同时，这对于算法性能的改善也起到了一定的作用。请使用前述的新函数来构造数据集，并选择径向基函数作为核方法，构造模型：

```
>>> 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
```

现在，我们已经可以对“具备给定特征集的人是否能够成功匹配”进行预测了。我们须要利用缩放处理函数，将待预测的数据按比例进行缩放，从而使相应的变量与我们构造模型时所用的变量处于相同的尺度范围内：

```
>>> newrow=[28.0,-1,-1,26.0,-1,1,2,0.8] # 男士不想要小孩，而女士想要
>>> m.predict(scalef(newrow))
0.0
>>> newrow=[28.0,-1,1,26.0,-1,1,2,0.8] # 双方都想要小孩
>>> m.predict(scalef(newrow))
1.0
```

尽管上述做法似乎可以给出合理的预测结果，但是假如我们能够确知预测结果的好坏程度，那么对于为基函数选择最佳的参数取值而言，将会是非常有帮助的。LIBSVM 也提供了对模型进行交叉验证（cross-validating）的功能。我们在第 8 章中已经讨论过交叉验证的工作原理——数据集被自动划分为训练集和测试集。训练集的作用是构造模型，而测试集的作用则是对模型进行测试，以评估预测结果的好坏程度。

可以利用交叉验证函数来检验模型的质量。该函数接受一个参数 n，并将数据集拆分为 n 个子集。然后，函数每次将一个子集作为测试集，并利用所有其他子集对模型展开训练。函数最后会返回一个结果列表，我们可以将该结果列表与最初的列表进行对比。

```
>>> 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
```

answers 与 guesses 之间的差异数为 116。由于初始数据集中有 500 行数据，因此这意味着算法得到了 384 项正确的匹配。如果你愿意的话，也可以在 LIBSVM 文档中找一找其他的核方法和对应的参数值，看看是否可以在此基础上通过改变参数值，令结果获得更进一步的改善。

基于 Facebook 的匹配

Matching on Facebook

Facebook 是时下流行的一个社会化网络站点，它最初面向的是大学生，但现在已经向更大范围内的受众开放了。和其他社会化网络站点一样，Facebook 允许用户定义个人简历，输入关于他们自身的个人详细信息，并通过网站与他们的好友进行联系。Facebook 还提供了一套 API，允许你查询用户的个人信息，查明两人是否好友。借助 Facebook 的 API，我们可以利用真实的人员信息，构造出一个与前述婚介数据集相类似的数据集合。

截至本书撰写期间，Facebook 仍然非常忠于个人隐私，因此我们只能查看好友的个人资料。其 API 的应用规则，要求用户必须先登录，并且只允许查询。因此很遗憾，如果你拥有一个 Facebook 的账户，并且已经与至少 20 个人建立了联系，那么你将只能在这一范围内调用 Facebook 的 API。

获得开发者密钥

Getting a Developer Key

如果你有 Facebook 的账号，那就可以在它的开发者站点 *http://developers.facebook.com* 上注册一个开发者密钥。

我们一共会得到两个字串，一个是 API 密钥，另一个是“私密（secret）”密钥（或称私钥）。API 密钥的作用是识别你的身份，而私钥的作用，则是在稍后将要看到的散列函数中，对发起的调用请求进行加密处理。首先，我们来新建一个名为 *facebook.py* 的文件，导入所需的模块，并定义好若干常量：

```
import urllib,md5,webbrowser,time
from xml.dom.minidom import parseString
```

```
apikey="Your API Key"
secret="Your Secret Key"
FacebookSecureURL = "https://api.facebook.com/restserver.php"
```

还有两个辅助函数须要加入：getsinglevalue，根据指定的具名节点（named node）得到其子节点的对应值；callid，根据系统当前时间返回一个相应的数字。

```
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))
```

一些 Facebook 的调用要求我们提供序列号，该序列号可以是任意数字，只要它大于我们上一次所用的序列号即可。利用系统时间生成的序列号，可以保证我们得到的数字始终比前一次的更大，同时，这种方法也非常简便。

建立会话

Creating a Session

建立一个指向 Facebook 的会话，其目的实际上是为了帮助我们建立一个可供他人使用的应用程序，人们有了这一应用程序之后，就不须要知道他们的登录信息了。建立会话的过程包含如下几个步骤。

1. 调用 Facebook API，请求一个令牌（token）；
2. 向用户发送一个指向 Facebook 登录页面的 URL，URL 中携带了令；
3. 一直等待，直到用户成功登录为止；
4. 利用令牌请求一个调用 Facebook API 的会话。

因为有几个变量，所有的 Facebook 调用都会用到，所以我们最好还是将 Facebook 的功能封装到一个类里。请在 *facebook.py* 中新建一个名为 fbsession 的类，并添加一个__init__方法，该方法实现了上面所列的几个步骤：

```
class fbsession:
  def __init_ _(self):
    self.session_secret=None
    self.session_key=None
    self.createtoken()
    webbrowser.open(self.getlogin())
    print "Press enter after logging in:",
    raw_input()
    self.getsession()
```

__init__用到了几个方法，为了使其能够正常工作，我们须要将这几个方法加入类中。首先，我们须要发送请求到 Facebook API。sendrequest 方法的作用便是建立一个指向 Facebook

的连接，并提交请求所需的参数。对返回的 XML 文件进行解析的工作，是由 minidom 解析器完成的。请将该方法加入类中：

```
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)
```

以粗体显示的代码行生成了相应的 URL 请求字串。此处我们调用了 makehash 方法，该函数将所有参数连接成一个字符串，然后用私钥对其进行散列化处理。当得到一个会话之后，我们会发现私钥马上就会改变，因此该方法还会检查我们是否已经持有了一个会话私钥。请将 makehash 加入类中：

```
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()
```

现在，我们可以开始编写实际的 Facebook API 调用了。首先是 createtoken，该函数的作用是创建并保存登录页面中将会用到的令牌：

```
def createtoken(self):
  res = self.sendrequest({'method':"facebook.auth.createToken"})
  self.token = getsinglevalue(res,'token')
```

然后是 getlogin，该函数的作用只是返回用户登录页面的 URL：

```
def getlogin(self):
  return "http://api.facebook.com/login.php?api_key="+apikey+\
        "&auth_token=" + self.token
```

待用户登录之后，我们应该调用 getsession 得到一个会话密钥和一个会话私钥（session secret key），后者是对日后发起的调用请求进行散列处理时所必需的。请将该函数加入类中：

```
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')
```

设置一个 Facebook 会话须要做大量的工作，不过一旦完成这项工作之后，将来的调用就会变得非常简单。本章我们只涉及人员信息的获取，但是如果你阅读相关的文档就会发现，像下载照片和事件信息这样的方法调用，其使用方法也是相当简单的。

下载好友数据

Download Friend Data

有了前面的铺垫，现在可以开始编写一些具有实用价值的方法了。getfriend 方法的作用是下载当前登录用户的好友 ID，并以一个列表的形式将其返回。请将该方法加入 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
```

因为 getfriends 只返回 ID，所以我们还须要另一个方法来下载实际的人员信息。getinfo 方法以用户 ID 列表作为参数，调用 Facebook 的 getInfo API。该函数只请求了少量选定的字段，不过我们可以在此基础上进行扩展，只要向 fields 中加入更多的字段，并修改从返回结果中解析并提取相关信息的代码即可。在 Facebook 的开发者文档中，有一个完整的字段列表介绍：

```
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):
  # 得到家庭住址的信息
  locnode=n.getElementsByTagName('hometown_location')[0]
  loc=getsinglevalue(locnode,'city')+', '+getsinglevalue(locnode,'state')

  # 得到就读学校的信息
  college=''
  gradyear='0'
  affiliations=n.getElementsByTagName('affiliations_elt')
  for aff in affiliations:
    # 类型为 1 代表学校
    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
```

上述函数以字典的形式返回最终的结果，它将一个用户 ID 与相应的信息子集建立起了映射关系。我们可以利用这一数据来构造新的婚介数据集。如果愿意的话，我们也可以在自己的 Python 会话中尝试一下新的 Facebook 类：

```
>>> import facebook
>>> s=facebook.fbsession()
登录后请按回车：
>>> friends=s.getfriends()
>>> friends[1]
u'iY5TTbS-0fvs.'
>>> s.getinfo(friends[0:2])
{u'iA810MUfhfsw.': {'gender': u'Female', 'location': u'Atlanta, '},
 u'iY5TTbS-0fvs.': {'gender': u'Male', 'location': u'Boston, '}}
```

构造匹配数据集

Building a Match Dataset

在我们的练习中，最后一个用到的 Facebook API 调用，是判断两人是否为好友。我们将该判断结果用作新数据集中的“答案”。这一 API 调用要求我们提供两个大小相同的 ID 列表，并返回一个新的列表，其中每一个配对对应一个数字——如果两人是好友则取值 1，否则就取值 0。请将该方法加入类中：

```
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
```

最后，我们将所有这些成果组织起来，构造一个可供 LIBSVM 使用的数据集。这一过程涉及：得到一个包含登录用户所有好友的列表，下载他们的个人信息，为每一个可能的配对建立一个数据行，然后判断该配对中两人是否为好友。请将 makedataset 加入类中：

```
def makedataset(self):
  from advancedclassify import milesdistance
  # 得到有关于我的所有好友的全部信息
  friends=self.getfriends()
  info=self.getinfo(friends)
  ids1,ids2=[],[]
  rows=[]

  # 以嵌套方式遍历，判断每两个人彼此间是否为好友
  for i in range(len(friends)):
    f1=friends[i]
    data1=info[f1]
```

```
    # 因为从 i+1 开始，所以不会重复
    for j in range(i+1,len(friends)):
      f2=friends[j]
      data2=info[f2]
      ids1.append(f1)
      ids2.append(f2)

      # 根据对原有数据的判断生成一些新的值
      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)
  # 针对每两个人，批量调用 arefriends
  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
```

上述方法将性别和状态转换成了数字，从而使该数据集能够直接为 LIBSVM 所用。在最后的循环中，因为 Facebook 限制了单次请求的数据规模，因此我们是以批量的方式请求每两个人的好友状态的（即判断是否为好友）。

构造 SVM 模型

Creating an SVM Model

为了尝试在前述数据的基础上构造 SVM，请重新载入 `facebook` 类，新建一个会话，并构造相应的数据集：

```
>>> reload(facebook)
<module 'facebook' from 'facebook.pyc'>
>>> s=facebook.fbsession()
登录后请按回车：
>>> answers,data=s.makedataset()
```

我们应该可以在此基础上直接运行 `svm` 方法：

```
>>> param = svm_parameter(kernel_type = RBF)
>>> prob = svm_problem(answers,data)
>>> m=svm_model(prob,param)
>>> m.predict([1,2003,1,2003,1]) # 两人均为男士，同一年毕业，就读于同一所学校
1.0
>>> m.predict([1,2003,1,1996,0]) # 不同年毕业，就读于不同的学校
0.0
```

当然，你所得到的最终结果可能会有所不同，但是作为一个典型的例子，模型极有可能会得到类似这样的结论：进入同一所大学或来自同一家乡的人，很有可能会成为好朋友。

练习

Exercises

1. **贝叶斯分类器** 我们在第 6 章中构造的贝叶斯分类器是否可以用于婚介数据集呢？能否找到一个恰当的例子加以说明？

2. **优化分界线** 我们是否可以利用第 5 章中学到的优化方法，而不是仅用求均值的方法来选择分界线呢？你会选用什么样的成本函数呢？

3. **选择最佳核参数** 请编写一个函数，以遍历的方式为 `gamma` 参数选择不同的取值，并针对给定数据集找出最佳取值。

4. **兴趣爱好的层级排列** 请为兴趣爱好设计一个简单的层级排列，并用一个数据结构来描述它。请修改 `matchcount` 函数，使其能够利用这一层级结构来给出相应的匹配分值。

5. **其他的 LIBSVM 核方法** 请阅读 LIBSVM 的文档，看看还有哪些核方法可以使用。请尝试一下多项式核方法（polynomial kernel）。看看预测的效果是否有所改进。

6. **基于 Facebook 的其他字段进行预测** 请查阅 Facebook API 的文档，找出所有可供使用的字段。我们可以为一位来自 Facebook 的用户构造什么样的数据集？我们是否可以利用 SVM 模型来做这样的预测：根据某人就读的学校来判断其是否会将某部影片列为收藏。我们还可以做其他形式的预测吗？

第 10 章

寻找独立特征

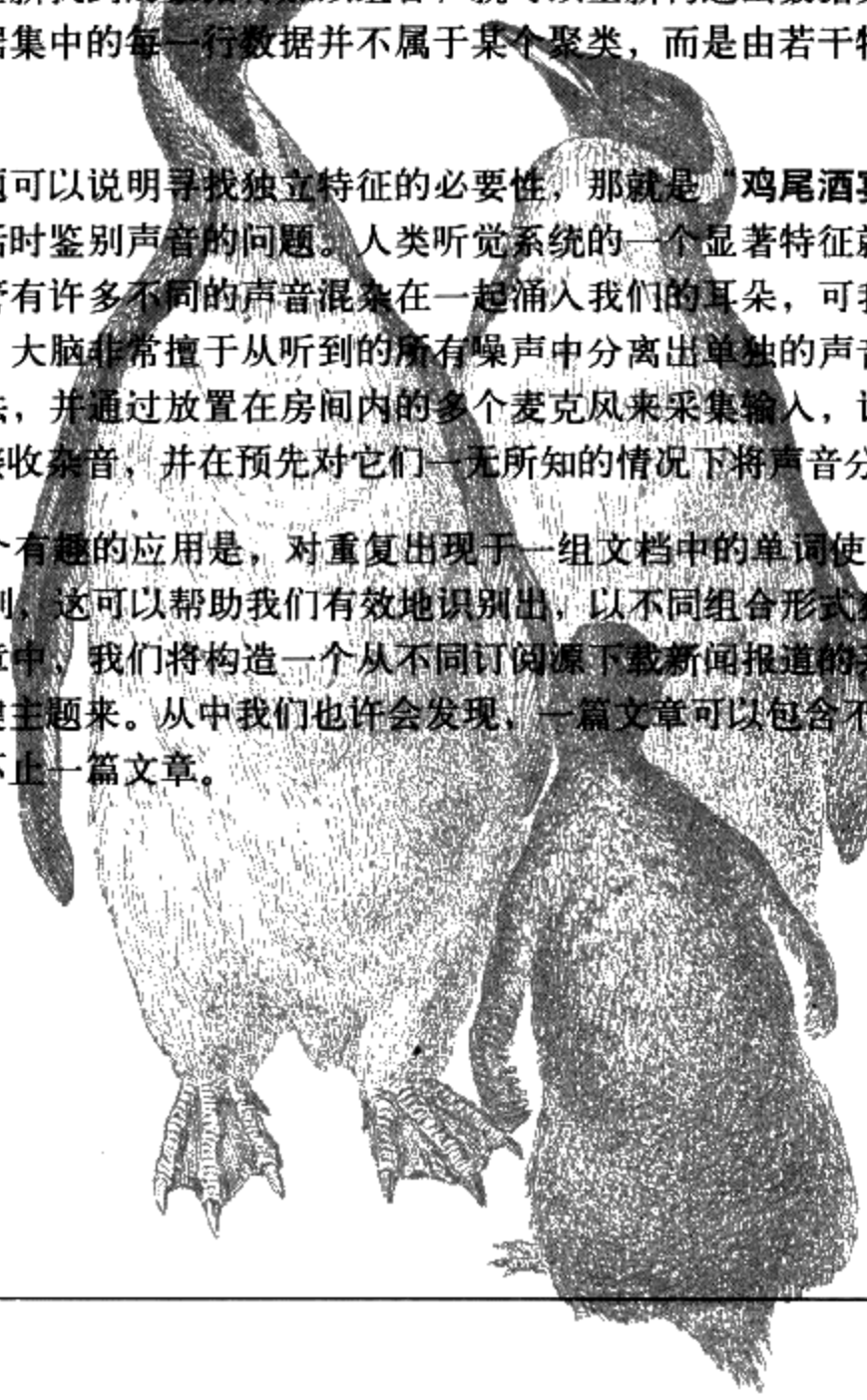
Finding Independent Features

迄今为止，除了第 3 章介绍的**聚类算法**属于**非监督**技术外，其余大部分章节都主要集中在**监督**分类器的讨论上。本章将研究如何在数据集并未明确标识结果的前提下，从中提取出重要的潜在特征来。和聚类一样，这些方法的目的不是为了预测，而是要尝试对数据进行特征识别，并且告诉我们值得关注的重要信息。

回顾第 3 章我们还记得，聚类算法将数据集中的每一行数据分别分配给了层级结构中的某个组（group）或某个点（point）——每一项数据都精确对应于一个组，这个组代表了组内成员的平均水平。**特征提取**是这种思想更为一般的表现形式；它会尝试从数据集中寻找新的数据行，将这些新找到的数据行加以组合，就可以重新构造出数据集。和原始数据集不一样，位于新数据集中的每一行数据并不属于某个聚类，而是由若干特征的组合构造而成的。

有一个经典的问题可以说明寻找独立特征的必要性，那就是**“鸡尾酒宴会”问题**，这是一个如何在多人谈话时鉴别声音的问题。人类听觉系统的一个显著特征就是：在一个人声鼎沸的屋子里，尽管有许多不同的声音混杂在一起涌入我们的耳朵，可我们还是能够从中鉴别出某个声音来。大脑非常擅于从听到的所有噪声中分离出单独的声音。通过使用类似本章中所描述的算法，并通过放置在房间内的多个麦克风来采集输入，计算机就有可能完成同样的任务——接收杂音，并在预先对它们一无所知的情况下将声音分离出来。

特征提取的另一个有趣的应用是，对重复出现于一组文档中的单词使用模式（word-usage patterns）进行识别，这可以帮助我们有效地识别出，以不同组合形式独立出现于各个文档中的主题。在本章中，我们将构造一个从不同订阅源下载新闻报道的系统，并从一组报道文章中识别出关键主题来。从中我们也许会发现，一篇文章可以包含不止一个主题；当然，主题也可以用于不止一篇文章。



本章的第二个例子是关于股票市场数据的，我们假定这些数据背后潜藏着诸多原因，正是这些原因共同组合的结果，导致了证券市场的格局。我们可以将同样的算法用于这些数据，寻找数据背后的原因，以及它们各自对结果所构成的影响。

搜集一组新闻

A Corpus of News

首先，我们需要一组供试验用的新闻报道文章。这些文章最好来自不同的地方，如此，所要讨论的主题就会出现于不同的位置，这样更容易为算法所识别。所幸的是，大多数主流的新闻服务和 Web 站点都提供了 RSS 或 Atom 订阅源，这些订阅源有的针对所有文章，也有的针对某个分类。在前几章中，我们已经用 *Universal Feed Parser* 解析过博客的 RSS 和 Atom 订阅源，我们也可以用同样的解析器去下载新闻报道。如果你手头还没有这个解析器，请从 *http://feedparser.org* 上下载。

选择新闻来源

Selecting Sources

从主流新闻电台和报纸到政论博客，有数以千计的新闻来源可供我们选择。这其中就包括：

- 路透社
- 美联社
- 纽约时报
- Google News
- Salon.com
- Fox News
- 福布斯杂志
- 美国有线新闻网

这里所列举的，仅仅是众多可能来源中的一小部分。假如将来自政治领域的持不同观点的新闻源，与那些采用不同写作风格的其他新闻源组合在一起，那将是对算法的一个更好的测试，因为算法应该能够从中找到关键特征，并忽略掉不相关的部分。此外，如果所给的数据集恰到好处，那么我们的特征提取算法就有可能从一系列带有特定政治倾向的故事中识别出特征来，并将这一特征连同描述同一主题的其他特征，一起赋予相应的故事。

请新建一个名为 *newsfeatures.py* 的文件，并加入下列代码以导入所需的函数库，并给出一个包含新闻来源的列表：

```
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']
```

上述订阅源列表包含了各种不同的新闻源，这些新闻源主要来自于各大新闻服务提供商的头条新闻（Top News）、世界新闻（World News）和美国新闻（U.S. News）等栏目。你也可以根据自己的喜好，对订阅源列表做相应的修改，但是必须要保证存在内容重叠的主题。如果没有任何文章存在共同之处，那么算法将很难从中提取出重要的特征，最终得到的特征将会变得没有任何的实际意义。

下载新闻来源

Downloading Sources

特征提取算法和聚类算法一样，接受一个大型的数字矩阵，其中的每一行代表一个数据项，而每一列则代表数据项的一个属性。在本例中，行对应于各类文章，列对应于文章中的单词。而矩阵中的每一个数字则代表了某个单词在一篇给定文章中出现的次数。因此，下面的矩阵告诉我们：单词“hurricane”在文章 A 中出现了三次，单词“democrats”在文章 B 中出现了两次，等等。

```
articles = ['A','B','C',...
words = ['hurricane','democrats','world',...
matrix = [[3,0,1,...]
          [1,2,0,...]
          [0,0,2,...]
                 ...]
```

为了从一个订阅源中得到这样的矩阵，与前面几章类似，我们须要用到几个方法。第一个方法是删除文章中所有的图片和 HTML 标记。请将 `stripHTML` 添加到 *newsfeatures.py* 中：

```
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
```

正如前几章中所做的那样，我们还需要一个方法来拆分文本中的单词。如果你事先已经写好了一个单词拆分方法，而且比本章所提供的简单包含文字与数字的正则表达式更为复杂，那么在此处复用一下该函数即可；否则，就请将下列函数添加到 *newsfeatures.py* 中：

```
def separatewords(text):
  splitter=re.compile('\\W*')
  return [s.lower() for s in splitter.split(text) if len(s)>3]
```

接下来这个函数所要实现的功能是对所有订阅源进行遍历，用 feedparser 进行解析，去除 HTML 标记，并利用先前定义好的函数逐个提取单词。该函数还记录了每个单词在所有文章中总共被使用的次数，以及在每篇文章中被使用的次数。

请将该函数添加到 *newsfeatures.py* 中：

```
def getarticlewords():
  allwords={}
  articlewords=[]
  articletitles=[]
  ec=0
  # 遍历每个订阅源
  for feed in feedlist:
    f=feedparser.parse(feed)

    # 遍历每篇文章
    for e in f.entries:
      # 跳过标题相同的文章
      if e.title in articletitles: continue
      # 提取单词
      txt=e.title.encode('utf8')+stripHTML(e.description.encode('utf8'))
      words=separatewords(txt)
      articlewords.append({})
      articletitles.append(e.title)

      # 在 allwords 和 articlewords 中增加针对当前单词的计数
      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
The function has three variables:
```

上述函数有三个变量。

- allwords 记录了单词在所有文章中被使用的次数。我们将利用该变量来判断，哪些单词应该被视作特征的一部分。
- articlewords 是单词在每篇文章中出现的次数。
- articletitles 是一个文章标题的列表。

转换成矩阵

Converting to a Matrix

现在我们手头已经有了两个字典，分别记录了单词在所有文章和每篇文章中出现的次数，这些数据必须被转换成先前介绍过的矩阵形式。首先，我们来建立一个单词列表，用作矩阵的列。为了缩减矩阵的大小，我们可以去掉几个只在少数几篇文章中出现过的单词（这样做对于寻找特征可能不会有多大影响），也可以去掉一些在过多文章中出现的单词。

此处，我们只考虑满足如下要求的单词：在超过三篇文章中出现过，但在所有文章中出现的比例则小于 60%。随后，我们就可以利用一个嵌套结构的列表推导式来构造矩阵了，目前这个列表推导式只是一个包含列表的列表。每个内嵌的列表都是通过遍历 `wordvec`，并查找字典中的单词构造而成的——如果单词不存在，就加 0；否则，就将单词在文章中出现的次数及单词本身加入列表。

请将 `makematrix` 函数添加到 *newsfeatures.py* 中：

```
def makematrix(allw,articlew):
  wordvec=[]

  # 只考虑那些普通的但又不至于非常普通的单词
  for w,c in allw.items():
    if c>3 and c<len(articlew)*0.6:
      wordvec.append(w)

  # 构造单词矩阵
  l1=[[(word in f and f[word] or 0) for word in wordvec] for f in articlew]
  return l1,wordvec
```

请开启一个 Python 会话，并导入 `newsfeatures`。然后尝试解析订阅源，并构造矩阵：

```
$ 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]
```

在上述例子中，我们列出了单词向量中的前 10 个单词。并列出了第二篇文章的标题，后面跟着的是该篇文章在单词矩阵中对应数据行的前 10 个值。从中我们可以看到，单词“men”在这篇文章中出现了一次，而其他几个单词则都没有出现过。

先前的方法

Previous Approaches

在前面几章中，我们已经讨论过几种不同的针对文本数据的单词统计方法。为了做个比较，我们首先来尝试一下先前的这些方法，看看会得到什么样的结果，然后将它们与特征提取得到的结果进行对比，相信这样做是很有意义的。如果我们手头有相关章节的代码，就请将对应模块导入进来，并针对目前的订阅源数据逐一进行尝试。如果没有这些代码也不要担心——在本节中我们将会告诉大家，如何将这些方法用于样本数据。

贝叶斯分类

Bayesian Classification

如你所知，**贝叶斯分类**是一种监督学习法。如果我们想要尝试使用第 6 章中构造的贝叶斯分类器，首先必须要对某几个样本故事进行分类，以供分类器训练之用。随后，分类器才能将后续故事放入先前定义好的分类中。除了须要在开始阶段接受训练这个明显的缺点外，这种方法还有一个局限：开发人员必须确定所有不同的分类。迄今为止我们见过的所有分类器，如决策树和支持向量机，在面对这样的数据集时都存在同样的限制。

如果想针对这一数据集尝试运行贝叶斯分类器，就须要将第 6 章中编写的模块放入工作目录中。此处我们可以借助 `articlewords` 字典，该字典原本就是用于获取每篇文章的特征集的。

请在你的 Python 会话中尝试输入如下命令：

```
>>> 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'
>>> # 将其作为‘iraq’故事加以训练
>>> classifier.train(wordmatrix[0],'iraq')
>>> artt[1]
u'Bush signs U.S.-India nuclear deal'
>>> # 将其作为‘india’故事加以训练
>>> classifier.train(wordmatrix[1],'india')
>>> artt[2]
u'Fatah, Hamas men abducted freed: sources'
>>> # 这个故事该属于哪个分类呢?
>>> classifier.classify(wordmatrix[1])
u'iraq'
```

在我们所使用的样本数据中，包含了许多可能的主题，而每个主题中仅有几个故事。最终，贝叶斯分类器会掌握所有的主题，但是因为它要求在每个主题上都要训练若干个样本，所以这种分类器更加适合于类别较少，而每个类别包含的样本数较多的情况。

聚类

Clustering

聚类是迄今为止我们看到过的唯一一种非监督方法，第 3 章中已经对此有过论述。

第 3 章所用的数据是放在一个矩阵中的，这个矩阵与我们刚才构造的矩阵完全一样。同样地，如果我们手头有那一章的代码，就请将其导入到自己的 Python 会话中，然后针对前面的单词矩阵运行聚类算法：

```
>>> import clusters
>>> clust=clusters.hcluster(wordmatrix)
>>> clusters.drawdendrogram(clust,artt,jpeg='news.jpg')
```

图 10-1 给出了执行该聚类算法的一个可能的结果，我们将结果保存在一个名为 *news.jpg* 的文件中。

不出所料，主题相近的新闻故事被分到了一起。此处得到的结果甚至要比第 3 章中的那个博客的例子更加令人满意。因为，不同的新闻出版机构往往都倾向于使用相近的语言来报道完全相同的东西。不过，下页图 10-1 中的个别例子也点出了一个问题：并非按部就班地将新闻故事逐一放入各个“桶 (buckets)”中，就一定能够得到准确的结果。例如，《The Nose Knows Better》原本是一篇健康类的文章，而此处我们却将它与另一篇报道《Suffolk Strangler》（译注 1）的文章放在了一起。有时，新闻文章和人一样也是不可拆分的，我们必须将其视作独一无二的整体才行。

如果愿意的话，我们还可以将矩阵旋转，看看故事中的单词如何分组。在本例中，像单词“station”、“solar”和“astronauts”将会被分在一起。

非负矩阵因式分解

Non-Negative Matrix Factorization

从数据中提取重要特征的技术，被称为**非负矩阵因式分解**（NMF）。这是整本书中涉及的最为复杂的技术之一，为了理解这项技术，我们有必要在此做进一步的解释，并向大家简要介绍一下线性代数方面的相关知识。本节涵盖了你所须掌握的全部内容。

矩阵数学简介

A Quick Introduction to Matrix Math

为了理解 NMF 的工作原理，首先我们须要了解一点有关矩阵乘法方面的知识。如果你已经学习过线性代数，就请放心地跳过本节。

矩阵乘法的例子如第 234 页图 10-2 所示。

译注 1：此处所指的，是近年来发生在英国萨福克郡的一起连环杀人案，这起案件在英国广受关注，媒体将被告与 1888 年出现于东伦敦的妓女连环杀手“开膛手杰克”相提并论，称之为“萨福克扼杀者 (Suffolk Strangler)”。

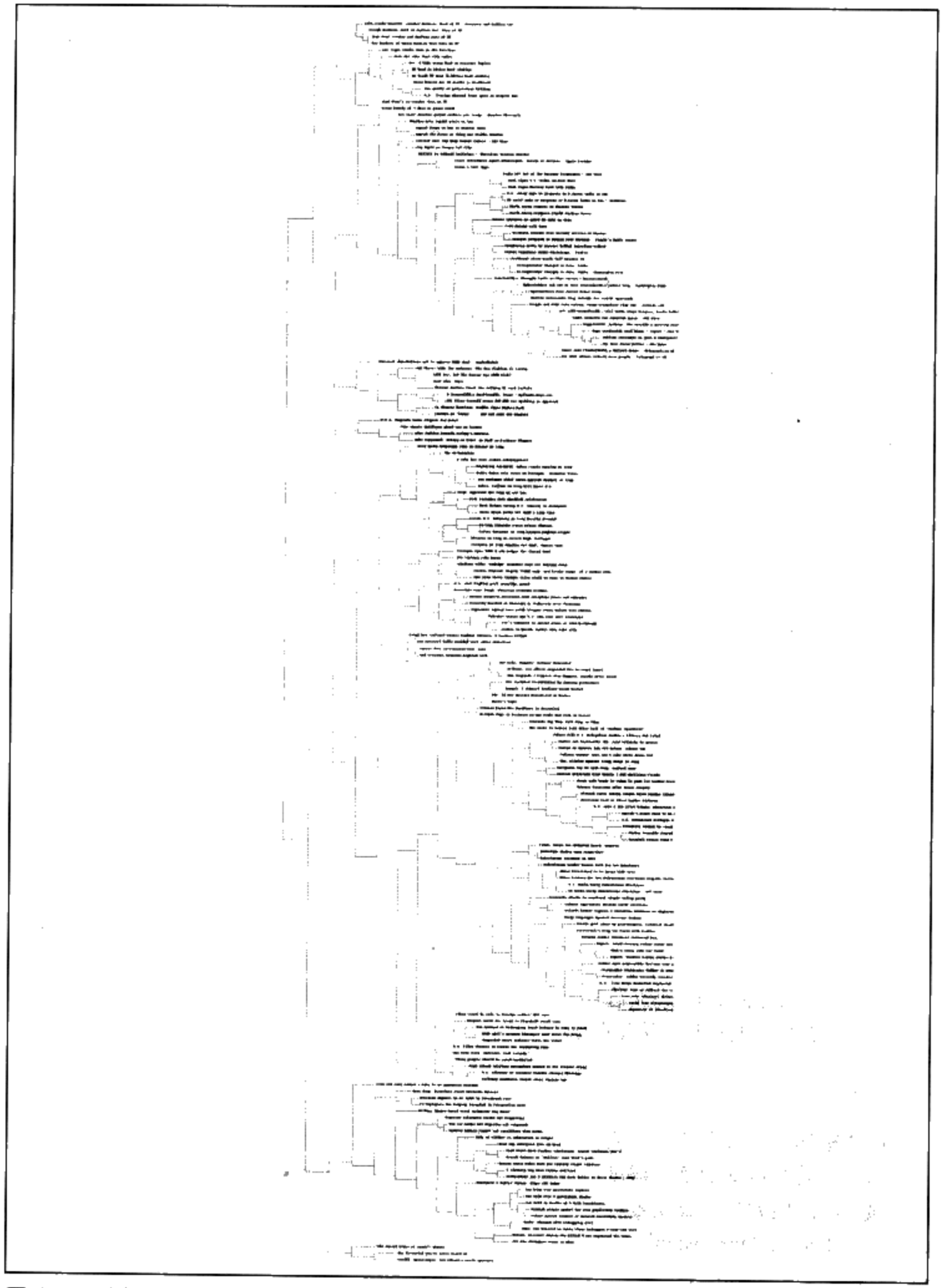


图 10-1：树状图给出了聚类过后的新闻故事

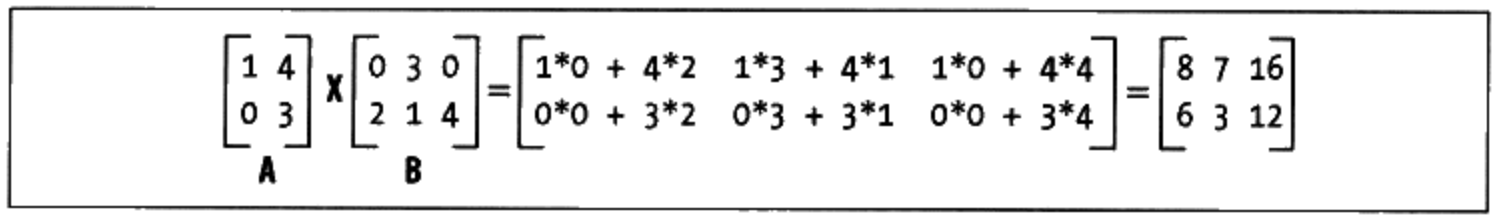


图 10-2：矩阵乘法的例子

该图给出了两个矩阵相乘的示意图。当两个矩阵相乘时，第一个矩阵（图中矩阵 A）的列数必须与第二个矩阵（矩阵 B）的行数相等。在本例中，矩阵 A 有两列，矩阵 B 有两行。结果矩阵（矩阵 C）的行数将与矩阵 A 的行数相等，列数将与矩阵 B 的列数相等。

矩阵 C 中每个元素的取值，是通过将矩阵 A 中相同行上的值与矩阵 B 中相同列上的值相乘，然后再将乘积相加得到的。以矩阵 C 左上角的值为例，其计算方法就是将矩阵 A 第一行上的值与矩阵 B 第一列上的对应值相乘，然后再将所得的乘积加在一起，这样就得到了最终的结果。矩阵 C 中的其他元素都是以相同方式计算得到的。

除乘法以外，另一个常用的矩阵操作是转置。所谓转置是指，将矩阵的列变成行，行变成列。它通常用符号“T”来标示，如图 10-3 所示。

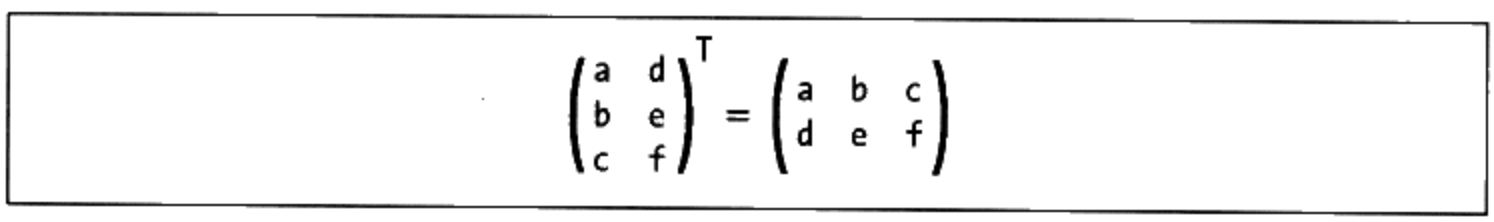


图 10-3：转置一个矩阵

我们将在后面的 NMF 算法实现中用到矩阵的转置和乘法操作。

这与文章矩阵有何关系？

What Does This Have to Do with the Articles Matrix?

目前为止，我们手中所持有的是一个带单词计数信息的文章矩阵。而我们的目标是要对这个矩阵进行因式分解，即：找到两个更小的矩阵，使得二者相乘以得到原来的矩阵。这两个矩阵分别是特征矩阵和权重矩阵。

特征矩阵

在该矩阵中，每个特征对应一行，每个单词对应一列。矩阵中的数字代表了某个单词相对于某个特征的重要程度。由于每个特征都应该对应于在一组文章中出现的某个主题，因此假如有一篇文章报道了一个新的电视秀节目，那么也许我们会期望这篇文章相对于单词“television”能够有一个较高的权重值。

权重矩阵

> 该矩阵的作用是将特征映射到文章矩阵。其中每一行对应于一篇文章，每一列对应于一个特征。矩阵中的数字代表了，将每个特征应用于每篇文章的程度。

和文章矩阵一样，在特征矩阵中，每个单词对应一列。由于该矩阵的每一行都对应着一个不同的特征，因而在此处用一个单词权重的列表来表示一个特征。图 10-4 给出了一个特征矩阵的例子。

	hurricane	democrats	florida	elections
特征 1	2	0	3	0
特征 2	0	2	0	1
特征 3	0	0	1	1

图 10-4：一个特征矩阵

由于矩阵中的每一行都对应于一个由若干单词联合组成的特征，因此很显然，只要将不同数量的特征矩阵按行组合起来，就有可能重新构造出文章矩阵来。而权重矩阵，如图 10-5 所示，则将特征映射到了文章。矩阵中的每一列对应一个特征，每一行对应一篇文章。

	特征1	特征2	特征3
hurricane in Florida	10	0	0
Democrats sweep elections	0	8	1
Democrats dispute Florida ballots	0	5	6

图 10-5：一个权重矩阵

图 10-6 给出了一个文章矩阵的构造过程，只要将权重矩阵与特征矩阵相乘，就可以重新构造出文章矩阵。

	特征1	特征2	特征3		hurricane	democrats	florida	elections		hurricane	democrats	florida	elections
hurricane...	10	0	0	特征1	2	0	3	0	hurricane...				
...sweep...	0	8	1	X 特征2	0	2	0	1	= ...sweep...				
Florida ballots	0	5	6	特征3	0	0	1	1	Florida ballots				

图 10-6：将权重矩阵与特征矩阵相乘

如果特征数量与文章数量恰好相等，那么最理想的结果就是能够为每一篇文章都找到一个与之完美匹配的特征。不过，在此处使用矩阵因式分解的目的，是为了缩减观测数据（本例中为文章）的集合规模，并且保证缩减之后足以反映某些共性特征。理想情况下，这个

相对较小的特征集能够与不同的权重值相结合，从而完美地重新构造出原始的数据集。但在现实中这种可能性是非常小的，因此算法的目标是要尽可能地重新构造出原始数据集来。

非负矩阵因式分解之所以如此称呼，是因为其所返回的特征和权重都是非负值。在现实中，这意味着所有的特征值都必须是正数或零，这一点对于我们的例子而言是毫无疑问的，因为单词在一篇文章中出现的次数是不可能为负的。同时，这也意味着特征是不能做减法的（即从某些特征中去掉一部分其他的特征）——如果明确排除掉某些单词，则 NMF 将无法找到有效解。尽管这样的约束也许会阻碍算法得到最佳的因式分解，但是其结果却往往更易于理解。

使用 NumPy
Using NumPy

Python 的标准库中并没有提供支持矩阵操作的函数。尽管我们可以自己编写这样的函数，但是还有一个更好的选择，那就是安装一个名为 *NumPy* 的包，这个包不仅提供了一个矩阵对象和所有必要的矩阵操作，而且其性能堪比商业数学软件。我们可以从 *http://numpy.scipy.org* 处下载到 NumPy。

更多关于安装 NumPy 的信息，请见附录 A。

NumPy 提供了一个矩阵对象，我们可以利用一个嵌套的列表对其进行初始化，同时该矩阵对象的结构与此前我们构造的文章矩阵非常的类似。为了立刻看到效果，我们可以将 NumPy 导入到自己的 Python 会话中，并创建一个矩阵试试：

```
>>> 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]])
```

NumPy 提供的矩阵对象支持若干数学运算，其中包括了以标准运算符表示的乘法运算和加法运算。而矩阵的转置操作则是通过 `transpose` 函数实现的：

```
>>> m2=matrix([[1,2],[3,4],[5,6]])
>>> m2
matrix([[1, 2],
        [3, 4],
        [5, 6]])
>>> m1*m2
matrix([[22, 28],
        [49, 64]])
```

此处，shape 函数所返回的是矩阵的行数和列数，这对于循环遍历矩阵中的所有元素是很有帮助的：

```
>>> shape(m1)
(2, 3)
>>> shape(m2)
(3, 2)
```

最后，NumPy 还提供了一个高效的数组对象（fast array object）。和矩阵一样，数组对象也可以是多维的。我们可以很容易地将一个矩阵转换成数组，反之亦然。当进行乘法运算时，数组的行为与矩阵有所不同；数组仅当彼此拥有完全相同的形式时（译注 2）才可以相乘，并且其运算规则是将一个数组中的每个值与另一数组中的对应值相乘。如下所示：

```
>>> 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]])
```

NumPy 的高性能优势，对于我们后面即将要看到的 NMF 算法实现而言，是很有必要的，因为 NMF 算法需要大量的矩阵运算。

算法实现

The Algorithm

我们即将要讨论的对矩阵进行因式分解的算法最早是在 20 世纪 90 年代后期公布的，因此本书所介绍的这一算法实现目前是最新的。有证据表明，该算法实现在某些问题的处理上表现出了很高的效率，比如从一组照片中自动判断出不同的面部特征就是一个例子。

通过计算最佳的特征矩阵和权重矩阵，算法尝试尽最可能大地来重新构造文章矩阵。在本例中，我们很有必要找到一种办法来对最终结果与理想结果的接近程度加以衡量。为此我们定义了 difcost 函数，该函数针对两个同样大小的矩阵遍历其中的每一个值，并将两者间差值的平方累加起来。

请新建一个名为 *nmf.py* 的文件，并将 difcost 函数加入其中：

```
from numpy import *

def difcost(a,b):
  dif=0
  # 遍历矩阵中的每一行和每一列
  for i in range(shape(a)[0]):
    for j in range(shape(a)[1]):
      # 将差值相加
      dif+=pow(a[i,j]-b[i,j],2)
  return dif
```

译注 2：即行数和列数都相同。

现在，我们需要一种方法能够逐步地更新矩阵，以使成本函数的计算值逐步降低。假如你已经阅读过第 5 章，就会明白此处我们需要的是一个成本函数，并且我们一定可以借助退火优化算法或群（swarm）优化算法搜索到一个满意的题解。然而，搜索最优解的一种更为行之有效的方法，是使用**乘法更新法则**（multiplicative update rules）。

有关这一组法则的来历已经超出了本章讨论的范围，但是假如你有兴趣了解更多的背景资料，可以在 *http://hebb.mit.edu/people/seung/papers/nmfconverge.pdf* 处找到原始的论文。

这些法则产生了 4 个新的更新矩阵（update matrices）。在下面的说明中，我们将最初的文章矩阵称为数据矩阵。

hn

经转置后的权重矩阵与数据矩阵相乘得到的矩阵。

hd

经转置后的权重矩阵与原权重矩阵相乘，再与特征矩阵相乘得到的矩阵。

wn

数据矩阵与经转置后的特征矩阵相乘得到的矩阵。

wd

权重矩阵与特征矩阵相乘，再与经转置后的特征矩阵相乘得到的矩阵。

为了更新特征矩阵和权重矩阵，我们首先将上述所有矩阵都转换成数组。然后将特征矩阵中的每一个值与 *hn* 中的对应值相乘，并除以 *hd* 中的对应值。类似地，我们再将权重矩阵中的每一个值与 *wn* 中的对应值相乘，并除以 *wd* 中的对应值。

函数 `factorize` 的作用就是完成上述计算任务的。请将 `factorize` 添加到 *nmf.py* 中：

```
def factorize(v,pc=10,iter=50):
  ic=shape(v)[0]
  fc=shape(v)[1]

  # 以随机值初始化权重矩阵和特征矩阵
  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)])

  # 最多执行 iter 次操作
  for i in range(iter):
    wh=w*h

    # 计算当前差值
    cost=difcost(v,wh)

    if i%10==0: print cost

    # 如果矩阵已分解彻底，则立即终止
    if cost==0: break
```

```
        # 更新特征矩阵
        hn=(transpose(w)*v)
        hd=(transpose(w)*w*h)

        h=matrix(array(h)*array(hn)/array(hd))

        # 更新权重矩阵
        wn=(v*transpose(h))
        wd=(w*h*transpose(h))

        w=matrix(array(w)*array(wn)/array(wd))

    return w,h
```

上述函数要求我们指定希望找到的特征数。有时，我们的确清楚须要寻找的特征数量（比如，在一段录音中的两种声音，或是当天的五大新闻主题）；但有时，我们却无法得知到底要指定多少特征。没有一种通用的方法可以自动确定正确的特征数目，但是借助实验手段，可以找到一个合理的范围。

请开启会话，针对矩阵 m1*m2 尝试运行上述函数，看一看该算法是否能够找到一个逼近原始矩阵的解：

```
>>> 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]])
```

上述算法成功地找到了权重矩阵和特征矩阵，使得这两个矩阵相乘的结果与原始矩阵完美匹配。我们也可以尝试将该算法用于前面的文章矩阵，看一看它提取关键特征的表现如何（这也许会耗费一些时间）：

```
>>> v=matrix(wordmatrix)
>>> weights,feat=nmf.factorize(v,pc=20,iter=50)
1712024.47944
2478.13274637
2265.75996871
2229.07352131
2211.42204622
```

变量 feat 现在保存着新闻报道的特征，而保存在变量 weights 中的则是将各特征应用于每篇文章的权重。不过通过观察我们发现，最后得到的矩阵并不是十分的理想。我们需要一种方法来呈现最终的结果，并对其加以解释。

结果呈现

Displaying the Results

如何恰如其分地将最终结果呈现出来呢，要回答这个问题稍有一些复杂。特征矩阵中的每个特征都有一个权重，它是用来指示每个单词应用到该特征的程度的，因此可以尝试列出每一个特征中的前 5 或前 10 个单词来，看看在该特征中哪几个单词的重要程度是最高的。在权重矩阵里对应列上的数字告诉我们的，是该特征应用于每一篇文章的权重值。因此，假如我们列出前三篇文章的权重，借此来考查该项特征应用于所有文章的情况，同样也是很有意义的。

请将一个名为 showfeatures 的新函数添加到 *newsfeatures.py* 中：

```
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=[]

  # 遍历所有特征
  for i in range(pc):
    slist=[]
    # 构造一个包含单词及其权重数据的列表
    for j in range(wc):
      slist.append((h[i,j],wordvec[j]))
    # 将单词列表倒序排列
    slist.sort()
    slist.reverse()
    # 打印开始的 6 个元素
    n=[s[1] for s in slist[0:6]]
    outfile.write(str(n)+'\n')
    patternnames.append(n)

    # 构造一个针对该特征的文章列表
    flist=[]
    for j in range(len(titles)):
      # 加入文章及其权重数据
      flist.append((w[j,i],titles[j]))
      toppatterns[j].append((w[j,i],i,titles[j]))

    # 将该列表倒序排列
    flist.sort()
    flist.reverse()

    # 显示前 3 篇文章
    for f in flist[0:3]:
      outfile.write(str(f)+'\n')
    outfile.write('\n')

  outfile.close()
  # 返回模式名称，以供后续使用
  return toppatterns,patternnames
```

上述函数遍历每一个特征，并构造了一个包含所有单词权重及其对应单词（取自单词向量）的列表。函数对列表还进行了排序，从而使得相对于某一特征拥有最大权重的单词出现在列表的最前面。随后，函数将这些单词中的前 10 个打印输出（译注 3）。由此，我们应该会对这一特征所要表达的主题有一个较为准确的把握了。函数最后返回了排在最前面的模式（patterns）及其名称，因此我们仅须计算一次，就可以将其再次用于后面的 showarticles 函数。

在列出特征之后，showfeatures 函数又循环遍历了文章的标题，并根据权重矩阵中文章与特征间形成的权重值，对结果进行了排序。随后，函数从中选出了与特征关联最为紧密的 3 篇文章，连同其在权重矩阵中的对应值一起打印输出。我们会发现，有时一个特征可能会对多篇相关的文章都很重要，而有时却只会对某篇文章构成影响。

现在，我们可以调用 showfeatures 函数对特征进行考查了：

```
>>> reload(newsfeatures)
<module 'newsfeatures' from 'newsfeatures.py'>
>>> topp,pn= newsfeatures.showfeatures(weights,feat,artt,wordvec)
```

因为结果非常冗长，所以代码将之保存到了一个文本文件中。我们要求函数生成 20 个不同的特征。很显然，在数以百计的文章中，存在的主题肯定不止 20 个。但是，我们希望找到的是那些最为关键的特征。例如，请看下面的例子：

```
[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')
```

上述特征清晰地列出了一组与巴勒斯坦选举有关的单词，而且还列出了一组与特征所要表达的主题关系密切的文章。由于生成这些结果的依据来自于文章的标题和一部分正文，因此我们可以看到，第 1 篇文章和第 3 篇文章都与上述特征紧密相关，即便两者的标题中并没有任何单词是一样的。另外，因为单词的重要性是根据其在大量文章中被引用的次数而得到的，所以单词“palestinian”和“elections”位于最前列。

有些特征并没有这样一组明确无疑的文章与之关联，但是它们仍然为我们提供了颇有价值的结果。请看下面的例子：

```
[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')
```

很显然，此处的特征与第 1 篇介绍乳癌的文章有着极为紧密的关系。然而，后面还有几篇关系不是那么紧密的文章是与健康相关的，这些文章中有一部分单词与第一篇文章是相同的。

译注 3：上述函数代码中实际打印的是前 6 个，原文此处有误。

以文章的形式呈现
Displaying by Article

关于结果数据的呈现方式，还有一种可供选择的方法，就是列出每篇文章及应用于该篇文章的前三项特征。借此可以判断出，一篇文章是由相同数量的几个主题共同构成的，还是由某个权重很高的主题单独构成的。

请将新函数，showarticles，添加到 *newsfeatures.py* 中：

```
def showarticles(titles,toppatterns,patternnames,out='articles.txt'):
  outfile=file(out,'w')

  # 遍历所有的文章
  for j in range(len(titles)):
    outfile.write(titles[j].encode('utf8')+'\n')

    # 针对该篇文章，获得排位最靠前的几个特征
    # 并将其按倒序排列
    toppatterns[j].sort()
    toppatterns[j].reverse()

    # 打印前 3 个模式
    for i in range(3):
      outfile.write(str(toppatterns[j][i][0])+' '+
                    str(patternnames[toppatterns[j][i][1]])+'\n')
    outfile.write('\n')

  outfile.close()
```

因为针对每篇文章的前几项特征已经由 showfeatures 函数计算得到，所以上述函数只是遍历了所有的文章标题，将其打印输出，然后再列出每篇文章的前几个模式。

为了运行上述函数，请重新加载 *newsfeatures.py*，并为其传入文章标题和 showfeatures 返回的结果：

```
>>> reload(newsfeatures)
<module 'newsfeatures' from 'newsfeatures.py'>
>>> newsfeatures.showarticles(artt,topp,pn)
```

上述代码的执行将会产生一个名为 *articles.txt* 的文件，其中包含了一系列文章，以及与之关系最为紧密的模式。例如下面就是一个很好的例子，一篇文章中包含了两个内容相同的特征：

```
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']
```

很显然，这两个特征都是与 Iraq 相关的，但这篇文章又不是非常的典型，因为文中并没有涉及“oil”或“gates”。通过构造出可以组合使用的，同时又不是专门针对于某篇文章裁剪得来的模式，我们的算法能够以较少的模式来覆盖更多的文章。

下面这篇文章有一个权重值很高的特征，但是这一特征却无法应用于其他任何文章：

```
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']
```

因为我们所使用的模式非常少，所以也有可能会出现少量与其他任何文章都没有什么相似性的文章，并且也得不出有关于它们自身的模式来。此处就有一个这样的例子：

```
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']
```

我们可以发现，在这个例子中位于最前面的几个特征与文章并没有什么相关性，而且看上去几乎都是随机产生的。所幸的是，此处的权重值都非常小，因而很明显我们不会将这些特征真正应用于该篇文章中。

利用股票市场的数据
Using Stock Market Data

NMF 除了能够处理一些像单词计数这样的带有名词性数据的问题外，还适用于包含真正的数值型数据的问题。本节我们将告诉大家如何利用同样的算法，借助从 Yahoo! Finance 下载得到的数据，对美国股票市场的**交易量**进行分析。通过对数据的分析，我们可以找出反映重要交易日的模式，以及潜在因素如何得以驱动多支股票交易量的原因。

金融市场被认为是集体智慧的一个典型的例子，因为它们拥有为数众多的参与者，这些参与者们根据掌握的各种信息和行情，彼此独立地采取行动，并产生少量的输出，比如：价格和成交量。已经证明在价格预测方面，群体相对于个体而言有着更大的优势。有大量的学术研究表明，在金融市场的价格制定方面，群体比任何个体都更有可能获得成功。

什么是成交量
What Is Trading Volume?

对于某支股票而言，成交量就是指在某一给定时间段内（通常是 1 天）所买卖的股票份数。下页图 10-7 显示的是一张 Yahoo!股票的走势图，股票代码（ticker symbol）为 *YHOO*。位于最上方的线代表**收盘价**，就是当天最后一次交易的价格。下面的柱状图则给出了成交量。

你会发现，当股票价格有较大变化的时候，成交量在那几天往往就会变得很高。这通常会发生在公司发表重要声明或发布财务报告的时候。此外，当有涉及公司或业界的新闻报道时，也会导致价格出现“突变”（spikes）。在缺少外部事件影响的情况下，对于某支股票而言，成交量通常是（但不总是）保持不变的。

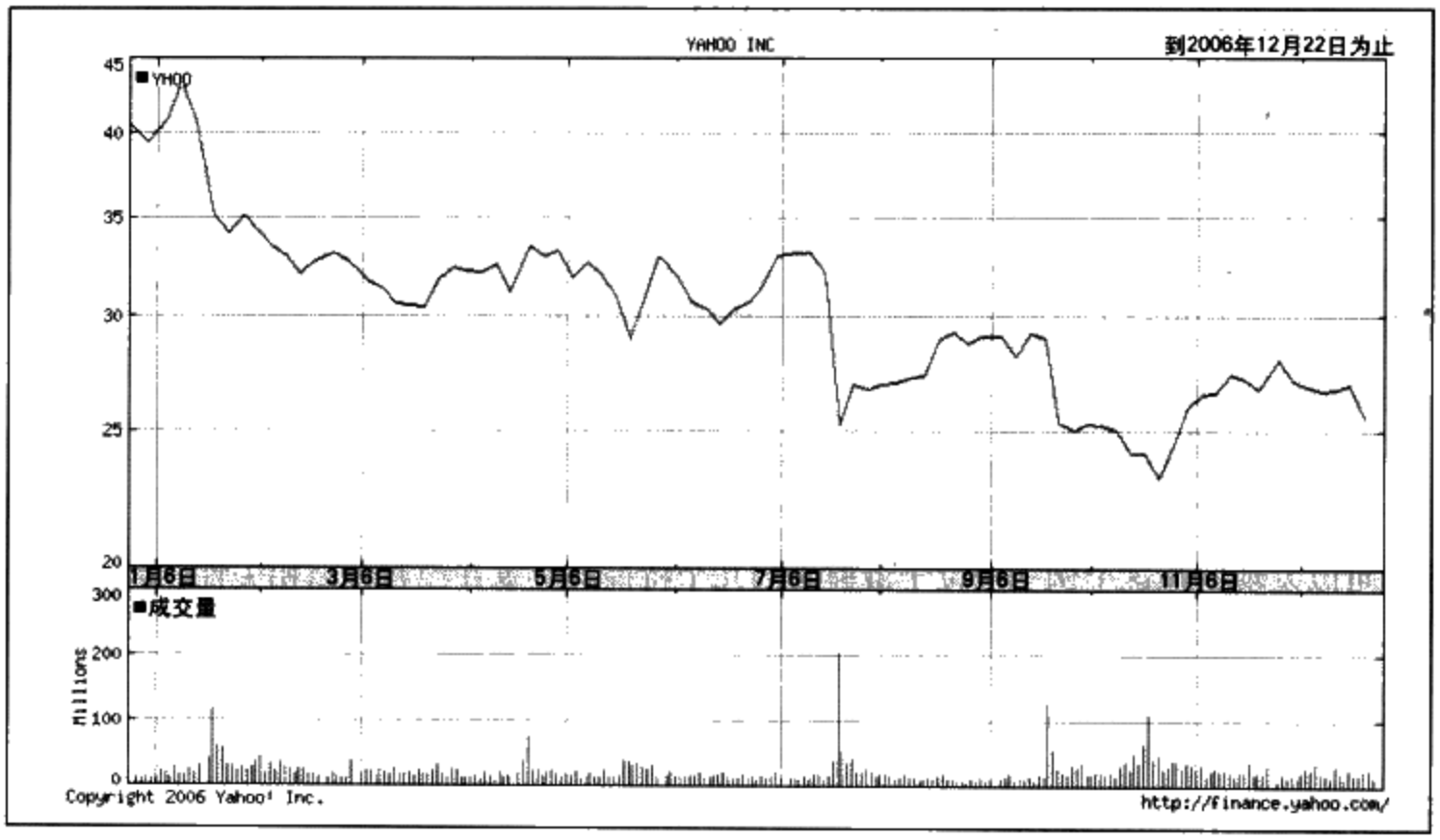


图 10-7：显示了价格和成交量的股票走势图

在本节中，我们将会在时间序列上对一组股票的成交量进行考查。从中找到即刻会对多支股票构成影响的反映成交量变化的模式，以及足以帮助我们判断特征的极具影响力的新闻事件。此处之所以选择成交量而不是收盘价作为考查对象，是因为 NMF 试图寻找的特征是可以相加在一起的正数；价格通常会受事件的影响而向下走，而 NMF 找到的特征是不会为负的。相反，成交量更容易被建模，因为它有一个可以受外部影响而递增的基准水平，这使得我们可以很容易地根据成交量来构造出正数矩阵来。

从 Yahoo! Finance 下载数据

Downloading Data from Yahoo! Finance

Yahoo! Finance 是一个汇集了各种财经数据的绝佳资源，它包括了股票价格、期权、货币汇率，以及债券利率。而且它还允许人们以一种很易于处理的 CSV 格式下载历史股票成交量和价格数据。通过访问如下的 URL 地址：*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*，我们可以下载到一个以逗号分隔的文件，其中包含了股票的一组每日数据，文件的开始几行如下所示：

```
交易日期,  开盘价,最高价,最低价,收盘价,成交量,  调整后的收盘价*
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
```

数据中的每一行都包含了交易日期、开盘价和收盘价、最高价和最低价、成交量，以及调整后的收盘价。调整后的收盘价考虑了股票除权或除息的情况，假如你在两个不同的交易日之间拥有这支股票，那么我们就可以利用它来准确计算出你所得的收益。

对于本例，我们将会得到一组股票的成交量数据。请新建一个名为 *stockvolume.py* 的文件，然后将下列代码加入其中。这段代码的作用是，根据一组股票代码将一系列以逗号分隔的文件下载到本地，并将其放入一个字典中。它还记下了这些股票中拥有交易记录的最小交易日天数，以此作为观测矩阵的行数：

```
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:
  # 打开 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()


  # 从每一行中提取成交量
  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()!='']
```

上述代码针对每支股票打开 URL 并下载数据。然后以逗号作为分隔符对每一行数据进行拆解，并从中取出第 5 项对应的浮点值，即该支股票的成交量，以此来构造列表。

准备矩阵

Preparing a Matrix

下一步我们要将上述数据转换成一个观测矩阵，这是调用 NMF 函数所必需的。为此我们只须要构造一个嵌套列表即可，其中的每一个内部列表代表了一组股票的每日总成交量。例如，请看下面的例子：

```
[[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,...]
 ...]
```

上述列表表明了，最近一日 AMGN 的交易量为 4 453 500，AVP 的交易量为 842 400，等等。而在前一日，两支股票的交易量分别为 5 000 100 和 1 486 900。对比于新闻故事的例子：原来的文章现在变成了交易日，原来的单词现在变成了股票份数，而原来的单词计数现在则变成了交易量。

利用一个列表推导式，可以很容易地将矩阵构造出来。其中，内层循环作用于股票代码，而外层循环则作用于由观测数据（天）构成的列表。请将下列代码添加到 *stockvolume.py* 的末尾处：

```
l1=[[prices[tickers[i]][j]
     for i in range(len(tickers))]
    for j in range(shortest)]
```

运行 NMF

Running NMF

现在，我们要做的唯一一件事情就是调用 `nmf` 模块中的因式分解函数。我们须要指定希望寻找的特征数目；对于为数不多的几支股票而言，特征数取 4 或 5 可能就差不多了。

请将下列代码添加到 *stockvolume.py* 的末尾处：

```
w,h= nmf.factorize(matrix(l1),pc=5)

print h
print w
```

现在，可以在命令行状态下运行上述代码，看看它是否能够正确执行：

```
$ python stockvolume.py
```

我们得到的矩阵分别对应于权重和特征。特征矩阵中的每一行对应一个特征，这一特征代表了一组股票的成交量，可以将之与其他特征相结合，以重新构造当日的交易量数据。权重矩阵中的每一行对应一个具体的日期，相应的数值代表了每个特征对于当日的贡献程度。

结果呈现

Displaying the Results

很显然，要直接解释上述矩阵是有难度的，因此我们须要编写代码，以更好的方式来呈现这些特征。我们希望看到的是每一个特征对于各支股票成交量的贡献度，以及与这些特征关系最为紧密的那一日。

请将下列代码添加到 *stockvolume.py* 的末尾处：

```
# 遍历所有特征
for i in range(shape(h)[0]):
  print "Feature %d" %i
```

```
# 得到最符合当前特征的股票
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

# 显示最符合当前特征的交易日期
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
```

由于有大量的文本产生，因此将结果输出到一个文件可能是最好的选择了。请在命令行输入如下命令：

```
$ python stockvolume.py > stockfeatures.txt
```

文件 *stockfeatures.txt* 现在包含着一个特征列表，其中还包括了与之关系最为紧密的股票及交易日期。下面是我们从文件中摘选出来的一个例子，因为从中我们发现，有一支股票在某个交易日显示出了非常高的权重：

```
特征 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')]
```

从数据中我们可以看到，此处的特征几乎是完全针对于 YHOO 的，尤其是 2006 年 7 月 19 日。碰巧那一天 Yahoo!的交易量出现了大幅地变动（a massive spike），当天也是 Yahoo!公布业绩预报（earnings guidance）的日子。

另有一个特征则同时对两家公司都产生了相同的影响，如下所示：

```
特征 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'),
```

此处的特征表明，Google 的交易量出现了大幅的变动，排在最前面的 3 组数据与当日的新闻事件有关。权重值最大的一天，即 1 月 20 日，Google 对外宣布了不会将搜索引擎的使用信息提交给政府。这一特征真正引人注目的地方在于，影响 Google 交易量的事件似乎对 Yahoo!的交易量也有很大的贡献，即使这与 Yahoo!并没有什么关系。上述列表中的第二个交易日呈现出交易量的大幅变动，这一情况出现在 Google 首席财务官宣布公司增长速度正在放缓的时候。图表显示，Yahoo!当天的交易量也有相应的增长，这有可能是因为，人们认为这一消息也会影响到 Yahoo!。

我们在此处所得到的信息与仅仅找出股票交易量之间的相关性有着很大的不同，明白这一点是很重要的。上述两个特征表明了，在某一段时期，Google 和 Yahoo!的股票交易量会呈现出相似的模式（patterns），而在其他时候，两支股票的走势则完全不同。仅仅考虑相关性，会将所有这些关系都“抹平”掉，同时又无法“抹杀”这样一个事实：仅仅只有几天 Yahoo!就发布了对外通告，称其受到了较大的影响。

上述这个例子只利用少数几支股票就揭示出了这样一个简单的道理，而如果我们选择更多的股票，并寻找更多的模式，那将会揭示出各种股票之间更为复杂的相互影响关系来。

练习

Exercises

1. **改变新闻来源** 本章例子里所选用的主要都是纯新闻来源。请尝试添加一些排名靠前的政论性博客（*http://technorati.com* 是寻找博客的一个好去处）。这样做对结果会有怎样的影响？是否存在对政治评论有显著影响的特征呢？是否可以将新闻故事和相关的评论很容易地分在一组呢？

2. **K 均值聚类** 本章我们将分级聚类算法用于了文章矩阵，如果我们使用 K 均值聚类算法，情况又会怎样呢？我们究竟须要多少个聚类才能有效地划分各个不同的故事？如何将这一结果与提取所有主题所需的特征数进行比较呢？

3. **优化因式分解**　我们能否将第 5 章中实现优化功能的代码用于对矩阵的因式分解呢？这样做会使因式分解的执行速度变得更快还是更慢呢？如何对结果进行比较？

4. **终止条件**　本章中的 NMF 算法在成本值降为 0 或迭代次数达到最大值时就会终止执行。有时，当我们找到了一个较为满意但还不是最理想的题解时，成本值就几乎不会再有任何改变了。请修改代码，当成本值在每次迭代中的变化量不超过 1%时，就终止算法的执行。

5. **其他的结果呈现方法**　本章给出的结果呈现函数只是列出了最为重要的特征，但它们却丢失了很多上下文信息。你能够想到其他的结果呈现方式吗？请尝试编写一个函数列出文章的原始正文及根据各关键特征得到的关键词，或者编写函数在成交量走势图中清晰地标注出关键的交易日。

第 11 章

智能进化

Evolving Intelligence

通观本书，我们已经遇见过许多不同的问题，而且在面对每一个问题时，都采用了一种最适合于解决该问题的算法。在某些例子中，我们还须要对参数进行调整，或者借助于优化手段来找出一个满意的参数集来。本章将考查一种截然不同的问题解决方法。与先前遇到一个问题就选择一种算法的思路不同，我们将编写一个程序，尝试自动构造出解决某一问题的最佳程序来。因而从本质上看，我们即将要构造的是一个能够构造算法的算法。

为了做到这一点，我们将采用一种称为**遗传编程**（genetic programming）的技术。因为本章是我们学习新算法的最后一章，所以笔者在此处选择了一个崭新的议题，一个激动人心的、目前仍旧处于研究当中的议题。本章与其他章节有稍许的不同，这是因为我们将不再使用任何开放的 API 或公共的数据集，同时也是因为，能够根据与大量人群的交互对自身作出修改的程序，是一种极为有趣且类型迥异的集体智慧。由于遗传编程是一个非常大的议题，已经有大量与此有关的书籍出现，因此在这里只提供一个引介，但是笔者希望它能引起大家的兴趣，并且能激发大家亲自动手进行研究和尝试。

本章涉及两个问题，分别是：根据给定的数据集重新构造一个数学函数；在一个简单的棋类游戏中自动生成一个 AI（人工智能）玩家。这仅仅是展示遗传编程能力的一个非常小的例子——计算能力（computational power）才是真正制约遗传编程问题解决能力的唯一因素。

什么是遗传编程

What Is Genetic Programming?

遗传编程是受到生物进化理论的启发而形成的一种机器学习技术。其通常的工作方式是：以一大堆程序（被称为**种群**）开始——这些程序可以是随机产生的，也可以是人为设计的（hand-designed），并且它们被认为是在某种程度上的一组优解（good solutions）。随后，这些程序将会在一个由用户定义的任务（user-defined task）中展开竞争。此处所谓的任务或许

是一种竞赛（game），各个程序在竞赛中彼此直接展开竞争，或者也有可能是一种个体测试，其目的是要测出哪个程序的执行效果更好。待竞争结束之后，我们会得到一个针对所有程序的评价列表，该列表按程序的表现成绩从最好到最差顺次排列。

接下来——也正是进化得以体现的地方——算法可以采取两种不同的方式对表现最好的程序实施复制和修改。比较简单的一种方式称为**变异**（mutation），算法会对程序的某些部分以随机的方式稍作修改，希望借此能够产生一个更好的题解来。另一种修改程序的方式称为**交叉**（crossover），有时也称为**配对**（breeding），其做法是：先将某个最优程序的一部分去掉，然后再选择其他最优程序的某一部分来替代之。这样的复制和修改过程会产生出许多新的程序来，这些程序基于原来的最优程序，但又不同于它们。

在每一个复制和修改的阶段，算法都会借助于一个适当的函数对程序的质量作出评估。因为种群的大小始终是保持不变的，所以许多表现极差的程序都会从种群中被剔除出去，从而为新的程序腾出空间。新的种群被称为“下一代”，而整个过程则会一直不断地重复下去。因为最优秀的程序一直被保留了下来，而且算法又是在此基础上进行复制和修改的，所以我们有理由相信，每一代的表现都会比前一代更加出色；这在很大程度上有点类似于人类世界中，年轻一代比他们的父辈更聪明。

创建新一代的过程直到终止条件满足才会结束，具体问题的不同，可能的终止条件也不同。

- 找到了最优解。
- 找到了表现足够好的解。
- 题解在历经数代之后都没有得到任何改善。
- 繁衍的代数达到了规定的限制。

对于某些问题而言——比如确定一个数学函数，令其将一组输入正确地映射到某个输出——要找到最优解是有可能的。但是对于其他问题——比如棋类游戏——也许根本就不存在最优解，这是因为题解的质量依赖于程序的对抗者所采取的策略。

下页图 11-1 以流程图的形式给出了遗传编程算法执行的一个大体过程。

遗传编程与遗传算法
Genetic Programming Versus Genetic Algorithms

我们在第 5 章曾经介绍过一组相关算法，被称为**遗传算法**（genetic algorithms）。遗传算法是一种优化技术，它汲取了生物进化中优胜劣汰的思想。就优化技术而言，不论是何种形式的优化，算法或度量都是事先选择好了的，而我们所要做的工作只是尝试为其找到最佳参数。

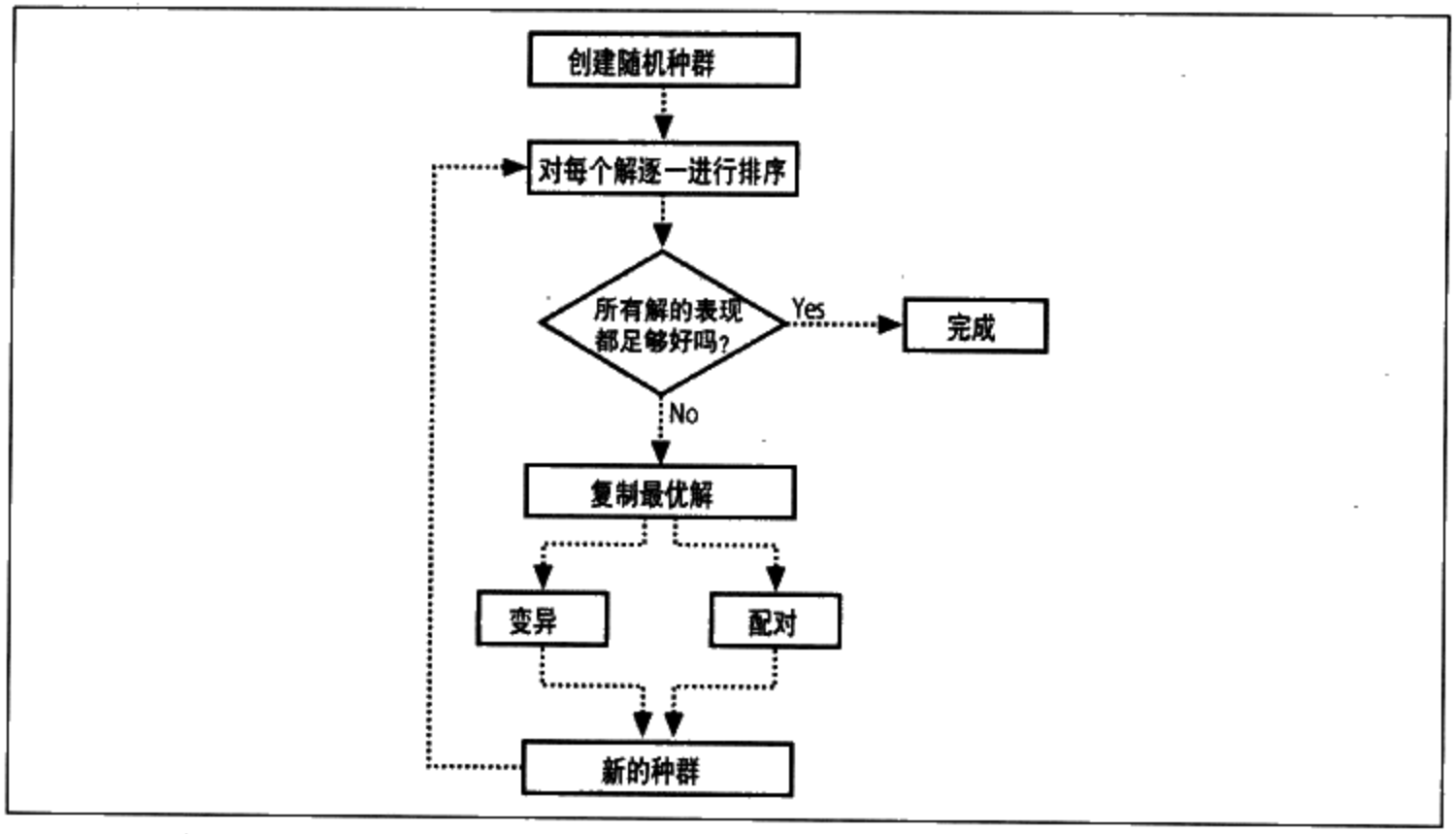


图 11-1：遗传编程的大体执行过程

遗传编程的成功之处

遗传编程自 20 世纪 80 年代以来就一直存在着，但是它的计算量非常庞大，而且以那个时候可以获得的计算能力而言，人们是不可能将其用于稍复杂一些的问题的。然而，随着计算机的执行速度越来越快，人们已经逐渐能够将遗传编程应用到复杂问题上了。正因为如此，许多以前的专利发明，借助遗传编程得到了再次挖掘和进一步的改善，而近年来也有不少可以获得专利的新发明，都是借助计算机利用遗传编程设计出来的。

人们已经将遗传编程技术广泛应用于许多领域，比如 NASA 的天线设计、光子晶体领域、光学领域、量子计算系统，以及其他科学发明领域。遗传编程也被应用于许多竞技类游戏程序的开发上，比如：国际象棋和西洋双陆棋。1998 年，来自卡耐基梅隆大学的研究者率领一支机器人队伍闯入了 RoboCup 机器人世界杯赛，并且在众多参赛者中排名居中，这支队伍就是完全利用遗传编程技术打造的。

和优化算法一样，遗传编程也需要一种方法来度量题解的优劣程度；但与优化算法不同的是，此处的题解并不仅仅是一组用于给定算法的参数而已。相反，在遗传编程中，连同算法本身及其所有的参数，都是按照优胜劣汰的进化规律（evolutionary pressure）自动设计得到的。

将程序以树形方式表示
Programs As Trees

为了构造出能够用以测试、变异和配对的程序，我们需要一种方法能够在 Python 代码中描述和运行这些程序。这种描述方法自身必须是易于修改的，而且更重要的一点是，它必须保证所描述的是一个实实在在的程序——这意味着，试图将随机生成的字符串作为 Python 代码的做法是行不通的。为了描述遗传编程中的程序，研究者已经提出了各种不同的方法，而这其中应用最为普遍的是树形表示法。

大多数编程语言，在编译或解释时，首先都会被转换成一棵**解析树**，这棵树非常类似于此处我们将要用到的树。(Lisp 编程语言及其变体，本质上就是一种直接访问解析树的方法)。图 11-2 给出了一个解析树的例子。

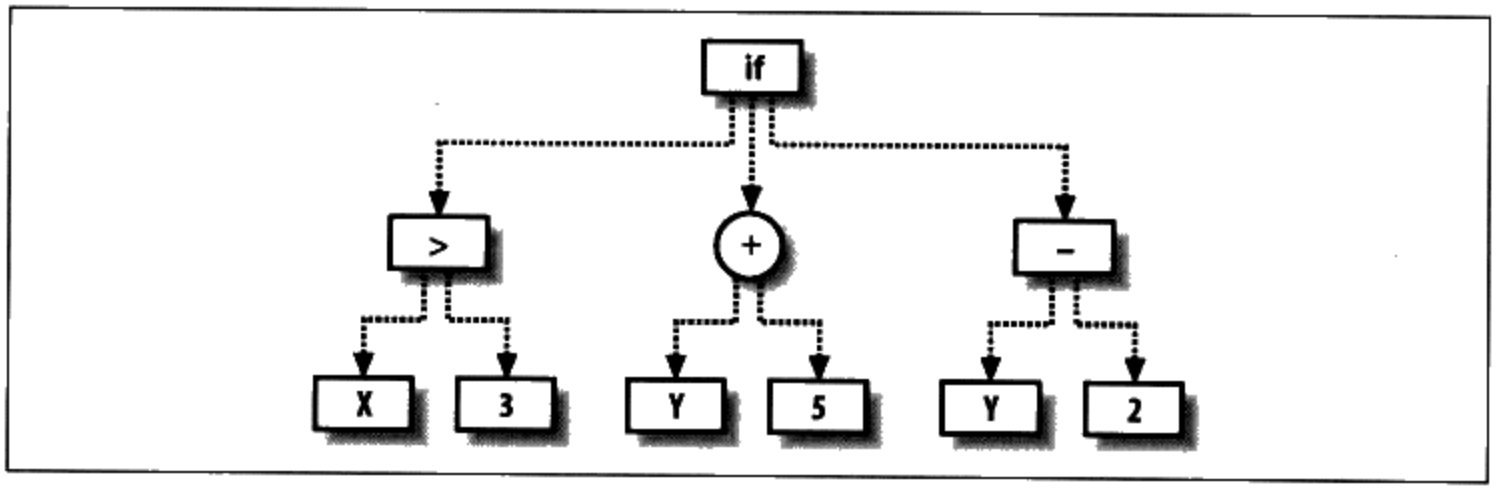


图 11-2：“程序”树的例子

树上的节点有可能是枝节点，代表了应用于其子节点之上的某一项操作；也有可能是叶节点，比如一个带常量值的参数。例如，图上的圆形节点代表了应用于两个分支（本例中为 Y 值和 5）之上的求和操作。一旦我们求出了此处的计算值，就会将计算结果赋予上方的节点处，相应地，这一计算过程会一直向下传播。同时你还会注意到，树上有一个节点的操作为“if”，这表示：如果该节点左侧分支的计算结果为真，则它将返回中间的分支；如果不为真，则返回右侧的分支。

对整棵树进行遍历，你就会发现它相当于下面这个 Python 函数：

```
def func(x,y)
  if x>3:
    return y + 5
  else:
    return y - 2
```

乍看起来，这样的树似乎只能用来构造非常简单的函数。不过，此处我们有两点须要考虑——第一，构成这棵树的节点可以是非常复杂的函数，比如距离度量或高斯分布。第二，

通过引用树上位置相对较高的节点，我们可以用递归的方式来构造树。采用这样的方式来构造树可以实现循环及其他更为复杂的控制结构。

在 Python 中表现树
Representing Trees in Python

现在，我们可以开始在 Python 中构造“树状程序（tree programs）”了。这棵树是由若干节点组成的，根据与之关联的函数的不同，这些节点又可以拥有一定数量的子节点。有些节点将会返回传递给程序的参数，另一些节点则会返回常量，而那些最值得关注的节点则会返回应用于其子节点之上的操作。

请新建一个名为 *gp.py* 的文件，并新建 4 个类：`fwrapper`、`node`、`paramnode` 和 `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
The classes here are:
```

下面是这些类的说明。

`fwrapper`

一个封装类，对应于“函数型”节点上的函数。其成员变量包括了函数名称、函数本身，以及该函数接受的参数个数。

`node`

对应于函数型节点（即带子节点的节点）。我们以一个 `fwrapper` 类对其进行初始化。当 `evaluate` 被调用时，我们会对各个子节点进行求值运算，然后再将函数本身应用于求得的结果。

`paramnode`

这个类对应的节点只返回传递给程序的某个参数。其 `evaluate` 方法返回的是由 `idx` 指定的参数。

`constnode`

返回常量值的节点。其 `evaluate` 方法仅返回该类被初始化时所传入的值。

此外，我们还会用到一些针对节点的操作函数。为此，须要构造一组函数，然后利用 `fwrapper` 类赋予它们名称和参数个数。请将下列函数列表添加到 *gp.py* 中：

```
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]
```

一些较为简单的函数，比如 `add` 和 `subtract`，可以利用 lambda 以内联方式来定义；而其他函数则要求我们必须在一个单独的语法块中进行定义。不论是哪种情况，它们都会被封装在一个 `fwrapper` 类中，并附以函数名称和要求的参数个数。最后一行创建了一个包含所有函数的列表，这样我们就可以在稍后对它们进行随机选择了。

树的构造和评估
Building and Evaluating Trees

现在，我们可以利用刚刚创建的节点类来构造如图 11-2 所示的程序树了。为此，请将 `exampletree` 添加到 *gp.py* 中：

```
def exampletree():
  return node(ifw,[
                  node(gtw,[paràmnode(0),constnode(3)]),
                  node(addw,[paramnode(1),constnode(5)]),
                  node(subw,[paramnode(1),constnode(2)]),
                  ]
              )
```

请启动一个 Python 会话来测试我们的程序：

```
>>> import gp
>>> exampletree=gp.exampletree()
>>> exampletree.evaluate([2,3])
1
>>> exampletree.evaluate([5,3])
8
```

上述程序成功地完成了与前面的代码块完全相同的功能。至此，我们已经成功地在 Python 中构造出了一个以树为基础的迷你语言和解释器。通过加入更多的节点类型，我们可以很方便地对这种语言进行扩展，同时它也是理解本章遗传编程概念的基础。为了确保你理解了程序树的工作原理，请尝试再构造一些简单的程序树。

程序的展现

Displaying the Program

因为程序树是自动构造而成的，我们对树的结构一无所知。所以寻找一种程序树的展现方法，从而使我们能够很容易地对其加以解释是很重要的。幸运的是，节点类的设计考虑了这一点：每个节点都带有一个描述函数名称的字符串，因此一个 display 函数只须返回该字符串及其子节点的显示字符串就可以了。为了使之更易于阅读，display 函数还应该将子节点缩进，这样我们一眼就可以辨别出树中的父子关系了。

请在node类中新建一个名为display的方法，其功能就是显示出整棵树的字符串表示形式：

```
def display(self,indent=0):
  print (' '*indent)+self.name
  for c in self.children:
    c.display(indent+1)
```

我们还须要为 paramnode 类新建一个 display 方法，其功能很简单，只须打印出该节点返回参数的对应索引即可：

```
def display(self,indent=0):
  print '%sp%d' % (' '*indent,self.idx)
```

最后是 constnode 类的 display 方法：

```
def display(self,indent=0):
  print '%s%d' % (' '*indent,self.v)
```

请利用上述方法打印出整棵树：

```
>>> reload(gp)
<module 'gp' from 'gp.py'>
>>> exampletree=gp.exampletree()
>>> exampletree.display()
if
 isgreater
  p0
  3
 add
  p1
  5
 subtract
  p1
  2
```

如果你已经读过了第 7 章，就会注意到此处所用的方法与那一章中决策树的显示方法非常的类似。为了获得更加清晰，更易于阅读的输出效果，第 7 章还给出了一种树的图形化显示方法。如果愿意的话，我们也可以采用同样的方法为树状程序提供一种图形化的表达形式。

构造初始种群

Creating the Initial Population

尽管为遗传编程手工构造程序是可行的，但是通常的初始种群都是由一组随机程序构成的。这样做可以使我们的起点变得更低，因为我们没有必要去设计一组几乎已经将问题完全解决了的程序。而且，这样做还可以在初始种群中引入更加丰富的多样性——由某位编程人员为了解决特定问题而专门设计的一组程序，彼此间很可能会非常相似，尽管这些程序也许会给出几乎正确的答案，但是最终的理想题解很有可能会截然不同。很快，我们就会了解到更多有关多样性重要价值的知识。

创建一个随机程序的步骤包括：创建根结点并为其随机指定一个关联函数，然后再随机创建尽可能多的子节点；相应地，这些子节点也可能会有它们自己的随机关联子节点。和大多数对树进行操作的函数一样，这一过程很容易以递归的形式进行定义。请将新函数 makerandomtree 添加到 *gp.py* 中：

```
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))
```

该函数首先创建了一个节点并为其随机选择了一个函数，然后它查看了随机选中的函数所需的子节点数。针对每一个子节点，函数通过调用自身来创建新的节点。通过这样的方式，一棵完整的树就被构造出来了，仅当被随机选中的函数不再要求新的子节点时（即，如果函数返回的是一个常量或输入参数时），向下创建分支的过程才会结束。此处的参数 pc 是我们在本章中将会一直使用的参数，它给出了程序树所需输入参数的个数。参数 fpr 给出了新建节点属于函数型节点的概率，而 ppr 则给出了当新建节点不是函数型节点时，其属于 paramnode 节点的概率。

请在你的 Python 会话中尝试执行上述函数，构造一组程序，然后看一看采用不同的变量值会得到什么样的结果：

```
>>> 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
```

如果一个程序的所有叶节点都是常量，则该程序实际上根本不会接受任何形式的输入参数，因此无论我们传给它什么样的输入，其结果都是一样的。我们可以利用前面定义好的函数来显示此处随机生成的这棵树：

```
>>> random1.display()
p0
>>> random2.display()
subtract
 7
 multiply
  isgreater
   p0
   p1
  if
   multiply
    p1
    p1
   p0
   2
```

我们会发现，有时生成的树会非常深，这是因为每个分支都会一直不断地增长下去，直到它遇到一个没有任何子节点的节点为止。这就是为什么增加一个最大深度的约束条件尤为重要的原因；否则，树就有可能会变得非常庞大，并且有可能会导致堆栈溢出。

测试题解

Testing a Solution

如果我们可以持续不断地产生随机程序，直至找到一个正确解为止，那么现在我们就已经具备了自动构造程序所需的所有条件了。但是很显然，这样做是荒谬而不切实际的，因为生成的程序有无穷多种可能，并且在任何一个合理的时间范围内，碰巧遇到一个正确解几乎是不可能的事情。不过在此处，我们寻找测试题解正确与否的方法还是有必要的，如果题解不正确，我们还可以确知它与正确答案的差距。

一个简单的数学测试

A Simple Mathematical Test

测试遗传编程算法的一个最为简单的例子，是尝试重新构造一个简单的数学函数。假设我们有一张包含输入和输出的表，如表 11-1 所示。

表 11-1：一个未知函数的输入数据和输出结果

X	Y	结果
26	35	829
8	24	141
20	1	467
33	11	1215
37	16	1517

的确存在一些函数可以将 X 和 Y 映射到上述输出结果一栏，但是没有人告诉我们这个函数到底是什么。统计学家看到上述表格也许会尝试做一个回归分析，但是这样做要求我们首先要推测出公式的结构。另一种选择是利用第 8 章中介绍过的 k-最近邻技术来构造一个预测模型，但是那样做我们就得保留所有的数据。有时，我们需要的仅仅是一个公式而已，或许我们会将其编入另一个更为简单的程序中，又或者我们是为了向他人解释正在发生的情况。

相信读者一定很想知道这个公式到底是什么，下面笔者就来为大家揭开谜底。请将 `hiddenfunction` 加入 *gp.py* 中：

```
def hiddenfunction(x,y):
   return x**2+2*y+3*x+5
```

我们将利用上述函数来构造一个数据集，借助得到的数据集，我们可以开始对生成的程序进行测试了。请新加一个函数，`buildhiddenset`，用以构造数据集：

```
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
```

请在你的 Python 会话中利用上述函数来创建一个数据集：

```
>>> reload(gp)
<module 'gp' from 'gp.py'>
>>> hiddenset=gp.buildhiddenset()
```

我们当然知道用来生成上述数据集的函数到底是什么，但是此处真正要测试的，是遗传编程是否能够在不知情的前提下重新构造出这一函数来。

衡量程序的好坏
Measuring Success

与优化技术一样，此处必须要找到一种衡量题解优劣程度的方法。在本例中，我们是在一个数值型结果的基础上对程序进行测试，因此一个简单的测试办法，是看这个程序与代表正确答案的数据集之间的接近程度。请将 `scorefunction` 加入 *gp.py* 中：

```
def scorefunction(tree,s):
  dif=0
  for data in s:
    v=tree.evaluate([data[0],data[1]])
    dif+=abs(v-data[2])
  return dif
```

上述函数将检查数据集中的每一行数据，计算函数的输出结果，并将其与实际结果进行比较。函数将所有差值累加起来，题解的表现越好，累加得到的值就越小——累加值为 0 则表示该程序得到的每一项结果都是正确的。现在，我们可以开始在自己的 Python 会话中对自动生成的程序进行测试了。我们来看一看它们的累加结果：

```
>>> reload(gp)
<module 'gp' from 'gp.py'>
>>> gp.scorefunction(random2,hiddenset)
137646
>>> gp.scorefunction(random1,hiddenset)
125489
```

由于我们只生成了一小部分程序，而且完全是随机产生的，因此这其中存在正确解的概率是非常小的。（假如你的程序中恰好有一个是正确答案，那么笔者建议你放下书本，去买彩票☺）不过，在这个预测数学函数的例子中，我们现在已经有办法能够测试出程序表现的优劣来了，这对于决定哪些程序将进入下一代是至关重要的。

对程序进行变异
Mutating Programs

当表现最好的程序被选定之后，它们就会被复制并修改以进入到下一代。如前所述，变异的做法是对某个程序进行少许的修改。一个树状程序可以有多种修改方式——我们可以改变节点上的函数，也可以改变节点的分支。图 11-3 是一个改变了所需子节点数目的函数，为此，我们可以删除旧的分支，也可以增加新的分支。

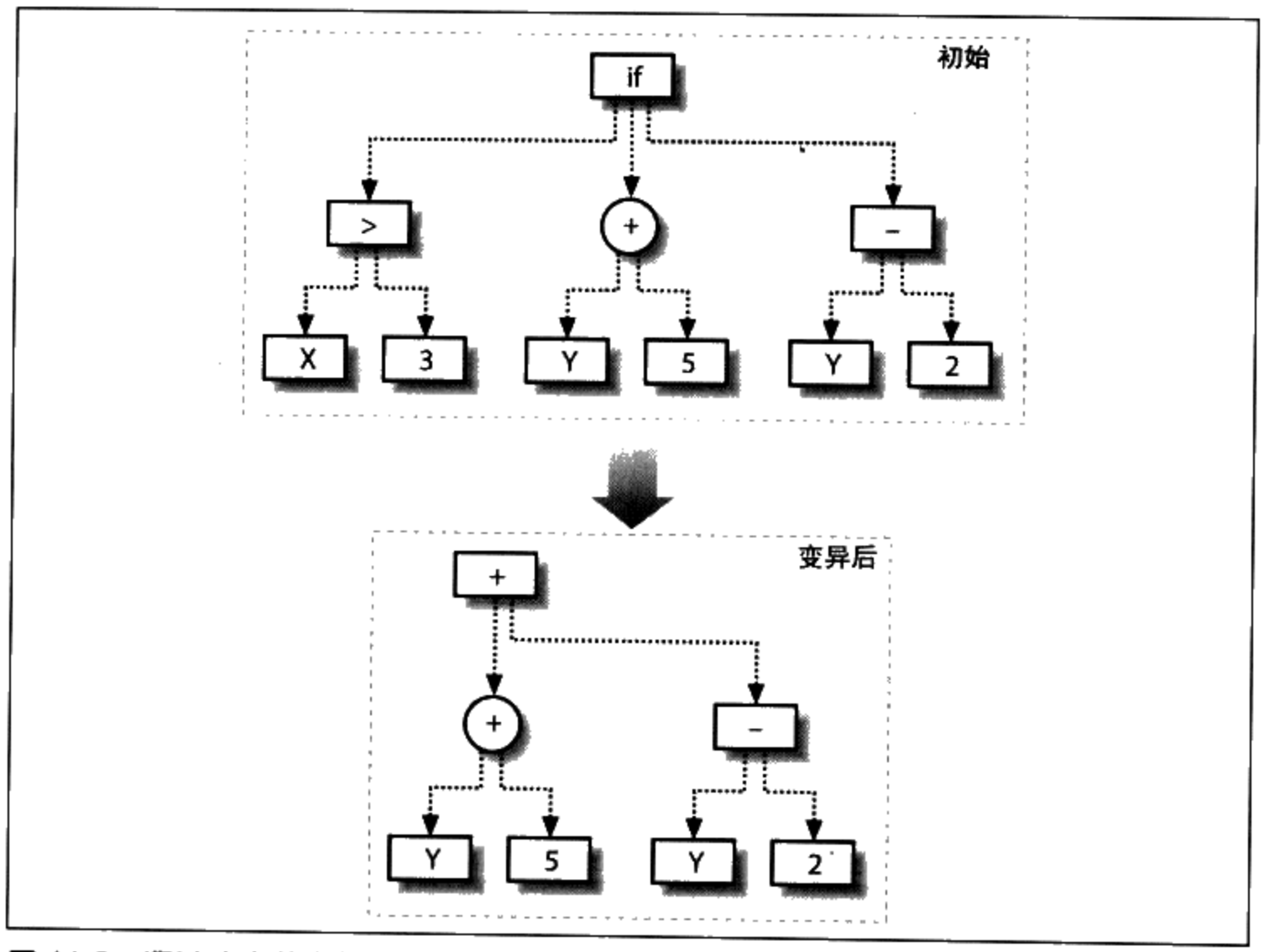


图 11-3：通过改变节点处的函数进行变异

另一种变异的方式，是利用一棵全新的树来替换某一子树，如下页图 11-4 所示。

变异采用的次数不宜过多。例如，我们不宜对整棵树上的大多数节点都实施变异。相反，我们可以为任何须要进行修改的节点定义一个相对较小的概率。从树的根节点开始，如果每次生成的随机数小于该概率值，就以如上所述的某种方式对节点进行变异；否则，就再次对子节点进行测试。

为了简单起见，此处给出的代码只实现了第二种变异方式。请新建一个名为 mutate 的函数以完成这项操作：

```
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
```

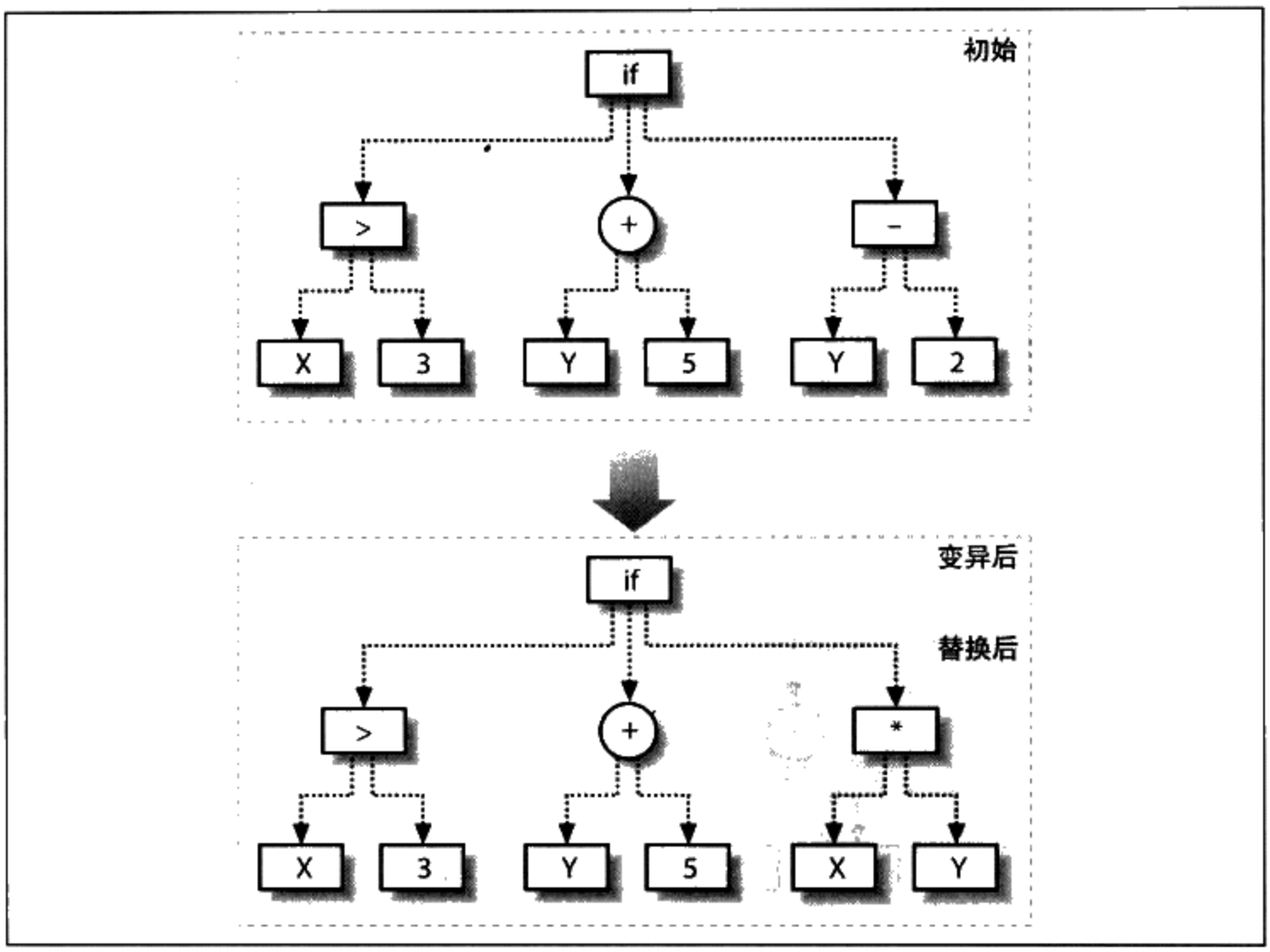


图 11-4：通过替换子树进行变异

上述函数从树的根节点开始，逐一判断节点是否应该被修改。如果不用修改，函数便会在树的子节点上再次调用自身。最终，我们有可能会对整棵树都进行变异，同样也有可能只遍历而不做任何改变。

请尝试对此前随机生成的程序执行若干次 `mutate` 函数，看一看该函数是如何对树进行修改的：

```
>>> 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
```

当对树成功实施变异之后，我们来看一看 scorefunction 的结果是否有了较大的改变，是变好了还是变差了：

```
>>> gp.scorefunction(random2,hiddenset)
125489
>>> gp.scorefunction(muttree,hiddenset)
125479
```

请记住，变异是随机进行的，而且不必非得朝着有利于改善题解的方向进行。我们只是希望其中的一部分变异能够对最终的结果有所改善。这种变化过程会一直持续下去，并且在经历过数代之后，我们终将找到最优解。

交叉
Crossover

除了变异外，另一种修改程序的方法称为交叉或配对。其做法是：从众多程序中选出两个表现优异者，并将其组合在一起构造出一个新的程序，通常的组合方式是用一棵树的分支取代另一棵树的分支。图 11-5 给出了一个说明其工作方式的例子。

执行交叉操作的函数以两棵树作为输入，并同时开始向下遍历。当到达某个随机选定的阈值时，该函数便会返回前一棵树的一份拷贝，树上的某个分支会被后一棵树上的一个分支所取代。通过同时对两棵树的即时遍历，函数会在每棵树上大致位于相同层次的节点处实施交叉操作。请将 crossover 函数添加到 *gp.py* 中：

```
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
```

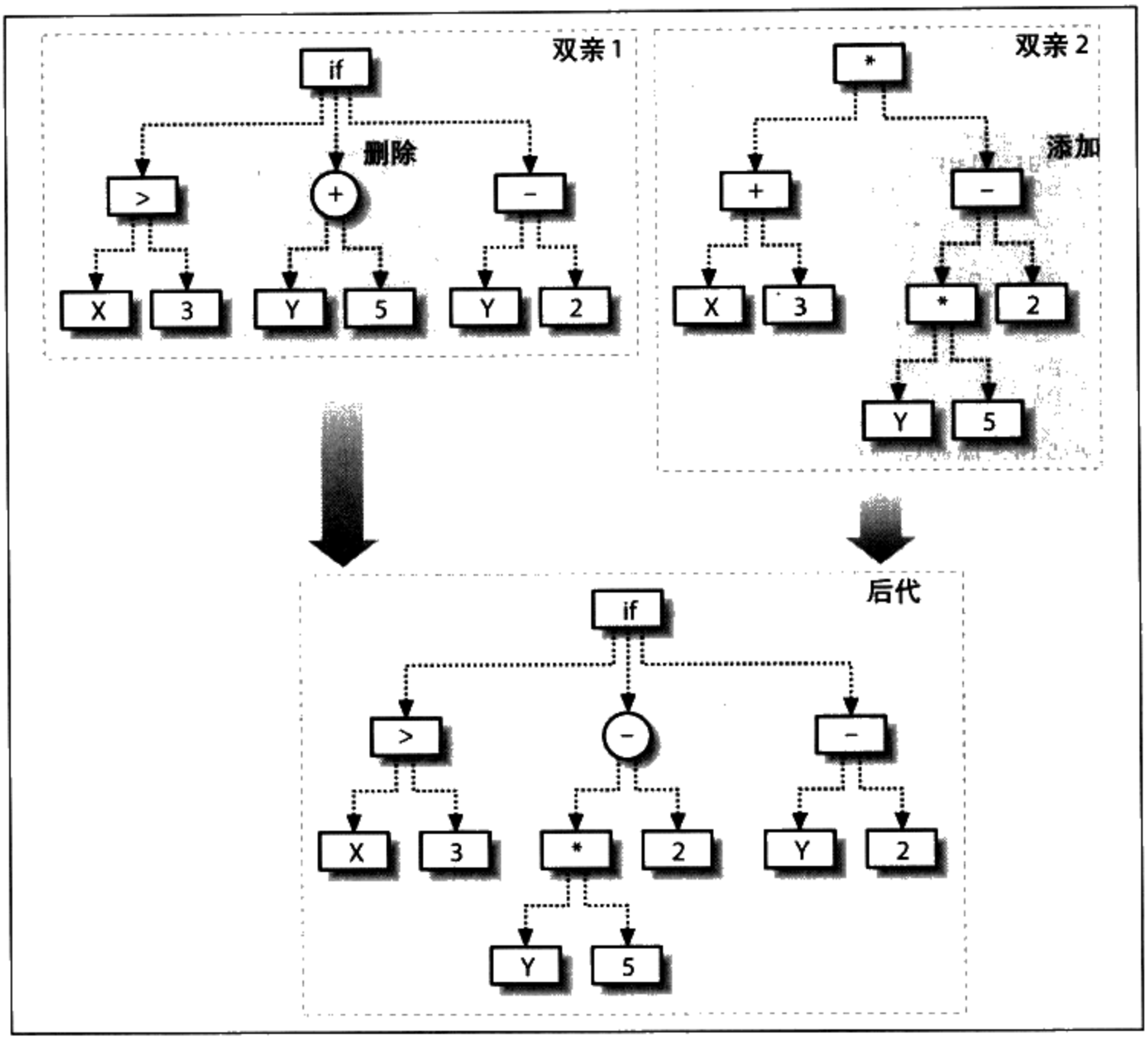


图 11-5：交叉操作

请针对一部分随机生成的程序尝试执行一下 crossover 函数。看看交叉之后的结果如何，同时也可以看一下，如果偶尔对两个最优的程序实施交叉操作，是否会得到一个更优的程序：

```
>>> 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
```

你也许会注意到，交换两个分支可能会完全改变程序的行为。或许你还会注意到，导致各个程序最终接近于正确答案的原因可能是五花八门的，因此，将两个程序合并后得到的结果可能会与前两者都截然不同。同样，此处我们的希望是，某些交叉操作会对题解有所改进，并且这些题解会被保留到下一代。

构筑环境

Building the Environment

有了度量程序优劣的办法和修改最优程序的两种方法，我们现在可以开始构筑供程序进化用的竞争环境了。相应的操作步骤如图 11-1 中的流程图所示。本质上，我们的思路是要生成一组随机程序并择优复制和修改，然后一直重复这一过程直到终止条件满足为止。

请新建一个名为 evolve 的函数，用以执行上述过程：

```
def evolve(pc,popsize,rankfunction,maxgen=500,
           mutationrate=0.1,breedingrate=0.4,pexp=0.7,pnew=0.05):
  # 返回一个随机数，通常是一个较小的数
  # pexp 的取值越小，我们得到的随机数就越小
  def selectindex():
    return int(log(random())/log(pexp))

  # 创建一个随机的初始种群
  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

    # 总能得到两个最优的程序
    newpop=[scores[0][1],scores[1][1]]
```

```
    # 构造下一代
    while len(newpop)<popsize:
      if random()>pnew:
        newpop.append(mutate(
                      crossover(scores[selectindex()][1],
                                scores[selectindex()][1],
                               probswap=breedingrate),
                     pc,probchange=mutationrate))
      else:
      # 加入一个随机节点，以增加种群的多样性

        newpop.append(makerandomtree(pc))

      population=newpop
    scores[0][1].display()
    return scores[0][1]
```

上述函数首先创建一个随机种群。然后循环至多 maxgen 次，每次循环都会调用 rankfunction 对程序按表现从优到劣的顺序进行排列。表现最优者会不加修改地自动进入到下一代，有时我们称这样的方法为**精英选拔法**（elitism）。至于下一代中的其他程序，则是通过随机选择排名靠前者，再经交叉和变异之后得到的。这一过程会一直重复下去，直到某个程序达到了完美的 0 分值，或者重复次数达到了 maxgen 次为止。

evolve 函数有多个参数，用以从各个不同的方面对竞争环境加以控制，说明如下。

rankfunction

对应于一个函数，即：将一组程序按从优到劣的顺序进行排列的函数。

mutationrate

代表了发生变异的概率，该参数会被传递给 mutate。

breedingrate

代表了发生交叉的概率，该参数会被传递给 crossover。

popsize

初始种群的大小

probexp

表示在构造新的种群时，“选择评价较低的程序”这一概率的递减比率。该值越大，相应的筛选过程就越严格，即：只选择评价最高者作为复制对象的概率就越大。

probnew

表示在构造新的种群时，“引入一个全新的随机程序”的概率。参数 probexp 和 probnew 会在接下来的“多样性的重要价值”一节中得到进一步讨论。

在我们的程序真正开始进化之前，还有最后一件事情须要做，即：根据 scorefunction 得到的结果对程序进行排序。请在 *gp.py* 中新建一个名为 getrankfunction 的函数，该函数将会返回一个针对给定数据集的排序函数：

```
def getrankfunction(dataset):
  def rankfunction(population):
    scores=[(scorefunction(t,dataset),t) for t in population]
    scores.sort()
    return scores
  return rankfunction
```

现在，我们可以开始为前述数据集自动生成代表数学公式的程序了。请在你的 Python 会话中尝试执行如下语句：

```
>>> 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
```

此处的数字变化得很慢，但是它最终应该会逐步减到 0。有意思的是，尽管这里给出的解是完全正确的，但是它显然比我们此前构造数据集时所用的函数要复杂得多。（自动生成得到的题解极有可能会比我们所设想的还要复杂。）不过，只需一点代数知识我们就会发现，这些函数实际上都是等价的——记住此处的 p0 为 X，p1 为 Y。下面第一行对应的函数，便是根据前面生成的树得到的：

```
 (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
```

上述例子告诉我们遗传编程的一个重要特征：算法找到的题解也许是完全正确的，亦或是非常不错的，但是由于其构造方式的独特性，通常这些题解远比编程人员手工设计出来的答案复杂得多。在这样的程序中，我们时常会发现有大段的内容不做任何工作，或者对应的是形式复杂、但始终都只返回同一结果的公式。请注意，在上面的例子中，节点(10>5)只不过是 1 的一种奇怪的表达方式而已。

要让程序保持简单是可以的，但是多数情况下这样做会增加我们寻找优解的难度。解决这一问题的一种更好的方法是：允许程序不断进化以形成优解，然后再删除并简化树中不必要的部分。我们可以手工完成这项工作，也可以借助剪枝算法自动进行。

多样性的重要价值
The Importance of Diversity

evolve 函数中有一部分代码会对程序按表现从优到劣的顺序进行排列，因而只选择两到三个排在最前列的程序，然后对其进行复制和修改以形成新的种群，这样的做法很具有吸引力。毕竟，为什么我们要不厌其烦地将表现一般的程序继续保留到下一代呢？

问题在于，仅仅选择表现优异的少数几个题解很快就会使种群变得极端同质化（homogeneous，或称为近亲交配，如果大家更习惯于这样称呼的话）：尽管种群中所包含的题解，表现都非常不错，但是它们彼此间不会有太大的差异，因为在这些题解间进行的交叉操作最终会导致种群内的题解变得越来越相似。我们称这一现象为达到**局部最大化**（local maxima）。对于种群而言，局部最大化是一种不错的状态，但还称不上是最佳的状态。在处于这种状态的种群里，任何细小的变化都不会对最终的结果产生多大的改变。

事实证明，将表现极为优异的题解和大量成绩尚可的题解组合在一起，往往能够得到更好的结果。基于这一原因，evolve 函数提供了两个额外的参数，允许我们对筛选进程中的多样性进行调整。通过降低 probexp 的值，我们允许表现较差的题解进入最终的种群之中，从而将“适者生存（survival of the fittest）”的筛选进程调整为“最适合者及最幸运者生存（survival of the fittest and luckiest）”。通过增加 probnew 的值，我们还允许全新的程序被随机地加入到种群中。这两个参数都会有效地增加进化进程中的多样性，同时又不会对进程有过多的扰乱，因为，表现最差的程序最终总是会被剔除掉的。

一个简单的游戏
A Simple Game

关于遗传编程，还有一个更为有趣的应用，那就是为游戏引入人工智能。通过彼此竞争以及真人对抗，为表现优异的程序提供更多的进入下一代的机会，我们可以让程序不断地得到进化。在本节中，我们将编写一个非常简单的游戏模拟程序，称为 Grid War，如下页图 11-6 所示。

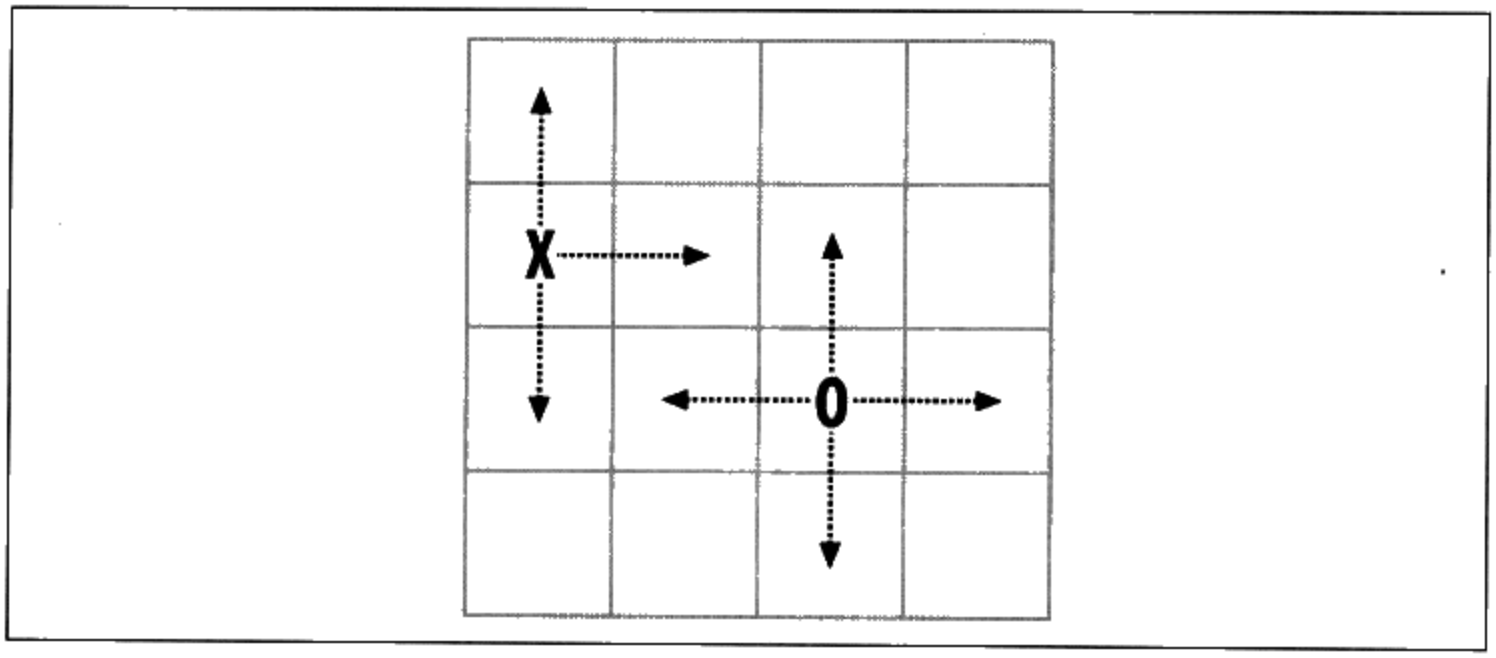


图 11-6：Grid War 的例子

该游戏有两位玩家参与，每个人轮流在一系列小网格中移动。每位玩家可以选择 4 个方向中的任何一个进行移动，并且游戏区域是受限的，如果其中一位玩家企图移到边界以外，他就丢掉了这一局。游戏的目标是要在自己这一局中捉住对方，相应的方法是，只要将自己移至对方所在的区域即可。唯一的附加条件是，如果你试图在一行的同一个方向上移动两次，那就自动认输了。这个游戏虽然非常简单，但是由于它要求玩家彼此相互博弈，因此我们会从中了解到更多有关进化在竞争性方面的细节。

首先，我们来创建一个函数，该函数涉及两位玩家，并在双方之间模拟一场游戏。函数将玩家及其对手的所在位置，连同他走的上一步，依次传给每一个程序，并根据返回的结果决定下一步该如何移动。

我们用数字 0 到 3 来代表移动的方向，即 4 个可能方向中的一个，不过由于所要处理的随机程序是可以返回任意整数的，所以函数必须对超出值域范围的情况进行处理。为此，我们以 4 为模对结果进行求模运算。此外，我们的随机程序所提供的方案，还有可能会让玩家在一个圆周范围内不停地移动，或者是诸如此类的方案，因此我们将移动的步数限制在 50 步，超过 50 就认为是打成了平局。

请将 gridgame 添加到 gp.py 中：

```
def gridgame(p):
  # 游戏区域的大小
  max=(3,3)

  # 记住每位玩家的上一步
  lastmove=[-1,-1]

  # 记住玩家的位置
  location=[[randint(0,max[0]),randint(0,max[1])]]

  # 将第二位玩家放在离第一位玩家足够远的地方
  location.append([(location[0][0]+2)%4,(location[0][1]+2)%4])
```

```
# 打成平局前的最大移动步数为 50
for o in range(50):

  # 针对每位玩家
  for i in range(2):
    locs=location[i][:]+location[1-i][:]
    locs.append(lastmove[i])
    move=p[i].evaluate(locs)%4

    # 如果在一行中朝同一个方向移动了两次，就判定为你输
    if lastmove[i]==move: return 1-i
    lastmove[i]=move
    if move==0:
      location[i][0]-=1
      # 限制游戏区域
      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]

    # 如果抓住了对方玩家，就判定为你赢
    if location[i]==location[1-i]: return i
return -1
```

如果玩家 1 是赢家，程序就返回 0；如果玩家 2 是赢家，程序就返回 1；如果打成了平局，就返回-1。我们可以尝试构造两个随机程序，并让它们彼此展开竞争：

```
>>> reload(gp)
<module 'gp' from 'gp.py'>
>>> p1=gp.makerandomtree(5)
>>> p2=gp.makerandomtree(5)
>>> gp.gridgame([p1,p2])
1
```

上述程序是完全没有经过进化的，因此它们可能会因为在一行上朝同一方向移动两次而输掉游戏。理想情况下，一个进化过的程序会学会避免这种情形。

循环赛

A Round-Robin Tournament

与通常的集体智慧应用一样，我们也希望通过与真人的对抗来测试程序的适应性，并以这样的方式来迫使其得到进化。这可能会是一种非常好的方法，因为我们可以捕获到数以千计的人的行为，并借此开发出更加智能的程序来。然而，随着种群数和代数的激增，这

样做就可能很快使比赛的数量增加到千百万场，并且对于这其中的大多数比赛而言，对手的实力可能都是非常弱的。而这与我们的目标是不相符的，因此，我们不妨先让这些程序在一场比赛中彼此展开竞争，借此得以进化。

下面的 tournament 函数接受一个玩家列表作为输入，并让每位玩家与其他玩家一一进行对抗，同时它还会记录每个程序在游戏中失败的次数。如果一个程序输掉了比赛，就得 2 分，如果打成平局，则得 1 分。请将 tournament 添加到 *gp.py* 中：

```
def tournament(pl):
  # 统计失败的次数
  losses=[0 for p in pl]

  # 每位玩家都将和其他玩家一一对抗
  for i in range(len(pl)):
    for j in range(len(pl)):
      if i==j: continue

      # 谁是胜利者?
      winner=gridgame([pl[i],pl[j]])

      # 失败得 2 分，打平得 1 分
      if winner==0:
        losses[j]+=2
      elif winner==1:
        losses[i]+=2
      elif winner==-1:
        losses[i]+=1
        losses[i]+=1
      pass

  # 对结果排序并返回
  z=zip(losses,pl)
  z.sort()
  return z
```

在函数的末尾处，我们对结果进行了排序，并且将排名最靠前的、失败次数最少的程序作为函数的返回值。这一返回类型是前述的 evolve 函数对程序进行评估时所必需的，也就是说，我们可以将 tournament 函数作为参数传递给 evolve，同时这也意味着，我们现在可以开始进行游戏，对程序实施进化了。请在你的 Python 会话中尝试执行上述函数（这也许会花费一定的时间）：

```
>>> reload(gp)
<module 'gp' from 'gp.py'>
>>> winner=gp.evolve(5,100,gp.tournament,maxgen=50)
```

随着程序的进化，我们会注意到，此处代表失败次数的数字并没有像先前的数学函数中那样严格递减。请想一下这是为什么呢——毕竟，我们总是让最优秀的玩家进入到了下一代，难道不是吗？恰好，因为下一代种群完全是由新进化而来的程序构成的，所以在上一代中表现最为优异的程序，在下一代中也许会表现得极为糟糕。

真人对抗
Playing Against Real People

当我们通过进化得到了一个程序，如果该程序在与其他机器人竞争者进行对抗时表现得十分优异，那么是时候进行真人对抗了。为此，我们可以再新建一个类，同样也定义一个evaluate方法，用以向用户显示游戏的区域，并询问下一步打算如何走。请将humanplayer类添加到*gp.py*中：

```
class humanplayer:
  def evaluate(self,board):

    # 得到自己的位置和其他玩家的位置
    me=tuple(board[0:2])
    others=[tuple(board[x:x+2]) for x in range(2,len(board)-1,2)]

    # 显示游戏区域
    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

    # 显示上一步，作为参考
    print 'Your last move was %d' % board[len(board)-1]
    print ' 0'
    print '2 3'
    print ' 1'
    print 'Enter move: ',

    # 不论用户输入什么内容，均直接返回
    move=int(raw_input())
    return move
```

请在你的Python会话中，尝试执行上述函数：

```
>>> reload(gp)
<module 'gp' from 'gp.py'>
>>> gp.gridgame([winner,gp.humanplayer()])
. O . .
. . . .
. . . .
. . . X
Your last move was -1
 0
2 3
 1
Enter move:
```

或许你会发现，我们的程序不堪一击，或者也有可能会极难对付，这取决于程序的进化程度。这些程序几乎肯定会掌握“不能在一行里朝同一方向移动两次”的策略，因为那样会即刻导致死亡，但是它掌握其他对抗策略的程度则要视每一次进化情况的不同而定。

更多可能性
Further Possibilities

本章只是遗传编程的一个引介，遗传编程是一个巨大的、飞速发展的领域。截至目前，我们已经利用这一方法解决了一些简单的问题，在这些问题的解决过程中，自动构建程序所花费的时间是按分钟而非天来计算的，但是其中的原则同样适用于更为复杂的问题。与更为复杂的问题相比，本章中种群所包含的程序数已经算是非常小了——一个更为典型的种群规模差不多有数千或数百万之多。我们鼓励大家去尝试难度更大的问题，并尝试更大的种群规模，只是当程序运行时，也许我们须要等上数小时或数天的时间才能得到最终的结果。

下面几个小节为我们简要列出了几种方法，借助这些方法我们可以将简单的遗传编程模型扩展到各种不同的应用领域。

更多数值型函数
More Numerical Functions

到目前为止，我们为了构造随机程序，已经采用过若干个数值型的函数。不过，这不足以体现一个简单程序所能应用的范围——面对更为复杂的问题，我们须要着力增加可供选用的函数，以帮助我们成功构造出程序树来。下面是一些可供选择的函数。

- 三角函数，比如正弦、余弦和正切函数。
- 其他数学函数，比如乘方、平方根和绝对值。
- 统计分布，比如高斯分布。
- 距离度量，比如欧几里德距离和 Tanimoto 距离。
- 一个包含 3 个参数的函数，如果第一个参数介于第二个和第三个之间，则返回 1。
- 一个包含 3 个参数的函数，如果前两个参数的差小于第三个，则返回 1。

正如我们所期望的，这些函数算是较为复杂的，我们在利用这些函数解决具体问题时，往往须要对其进行裁减。在解决信号处理领域中的问题时，三角函数也许是必要的，不过在本章我们所构建的游戏中，三角函数的用处就不大了。

记忆力

Memory

本章中构造的程序几乎完全是“反应式”的（reactive），它们只会根据输入来给出结果。尽管这种做法对于解决数学函数的问题是有效的，但是它却使我们的程序在执行时缺少了长期策略（longer-term strategy）。在前面的“追逐”游戏（chasing game）中，我们会将上一步操作传递给程序——因而大多数情况下，程序会明白，它们不可以在一行里朝相同方向移动两次——但这仅仅是程序的一种输出而已，并非它们自己做出的决策。

如果一个程序要发展出更为长期的策略，那么就须要有一种方法，能够将下一回合中需要用到的信息保存起来。实现这一功能的一种简单办法是：建立一种新类型的节点，用以将信息存入预先定义好的内存槽（slot）内，或从槽中取出。一个用于**存入信息**的节点（store node），包含一个子节点和一个指向记忆槽的索引；它会从子节点中取出结果，并存入内存槽中，然后再将结果一同传递给其父节点。一个用于**取回信息**的节点（recall node），不包含任何子节点，并且它只返回位于相应槽中的结果值。如果一个存入节点位于树的根部，则树上的任何一个部位，只要具有相应的取回节点，就可以获取到最终的结果值。

除了独享内存（individual memory）外，我们还可以设置共享内存，以供所有程序读写之用。共享内存除了有一组可供所有程序读写的内存槽外，与独享内存并没有多大的区别。共享内存为更高级别的协作与竞争创造了条件。

不同数据类型

Different Datatypes

本章所讨论的框架是完全针对于接受整型参数并返回整型结果的程序的。因为浮点数的操作和整型是类似的，所以我们只要稍作修改就可以将其用于浮点数。为此，我们只须修改`makerandomtree`，以随机的浮点值而非整型值来生成常量节点。

假如我们要让程序能够处理其他类型的数据，则还须做更大范围的修改，大部分修改都是针对节点上的函数进行的。对基础框架进行修改之后，我们可以令其支持如下类型的数据。

字符串

字符串的相关操作包括连接（concatenate）、分割（split）、索引（indexing）和求子串（substrings）。

列表

列表操作与字符串操作差不多。

字典

字典的相关操作包括替换和添加。

对象

任何一个自定义对象都可以用于一棵树的输入，树上节点处的函数就对应于该对象的方法调用。

从上述这些例子中我们可以得出一个很重要的结论：我们时常会要求树上的节点处理不止一种类型的返回值。比如，一个求子串的操作就需要一个字符串和两个整数作为输入，这意味着：子节点中必须有一个返回字符串，而另两个则分别返回整数。

针对上述情况，一种最为简单的做法是：随机地生成树上的节点，对节点施以变异和配对，并剔除那些数据类型不匹配的节点。但是这在计算上可能是一种浪费，何况我们已经掌握了为树的构造增加限制条件的方法——在前面处理整型值的树中，每个节点处的函数都知道自己需要多少子节点；同样，我们可以如法炮制，对子节点的类型及其返回类型加以限制。例如，我们可以按照如下方式重新定义 fwrapper 类，此处的 params 是一个字符串列表，它为每个参数指定了所使用的数据类型：

```
class fwrapper:
  def __init_  _(self,function,params,name):
    self.function=function
    self.childcount=param
    self.name=name
```

同时，我们或许还希望将 flist 设置为一个带返回值类型的字典。例如：

```
flist={'str':[substringw,concatw],'int':[indexw,addw,subw]}
```

然后，我们可以将 makerandomtree 的开始部分按如下方式进行修改：

```
def makerandomtree(pc,datatype,maxdepth=4,fpr=0.5,ppr=0.5):
  if random()<fpr and maxdepth>0:
    f=choice(flist[datatype])
    # 依据所有针对于 f 的参数类型，逐一调用 makerandomtree
    children=[makerandomtree(pc,type,maxdepth-1,fpr,ppr)
              for type in f.params]
    return node(f,children)
  诸如此类……
```

同样，crossover 函数也可能须要进行相应的修改，以确保实施交换的节点具有相同的返回类型。

有关如何将遗传编程从一个简单模型扩展到更为复杂的情形，本节从概念上为我们提供了一些思路，同时我们也鼓励大家进一步对其加以完善，并尝试将自动生成的程序用于更为复杂的问题。尽管程序的生成过程可能会耗费相当长的时间，但是一旦找到了一个表现优异的程序，我们就可以对其反复地加以利用。

练习

Exercises

1. **更多的函数类型**　前面我们只给出了少数几个函数。你还能想到其他函数吗？请实现一个带 4 个参数的欧几里德距离节点。

2. **替换变异**　请实现如下变异过程：选择树上的一个随机节点并对其加以修改。请确保其可以处理：函数、常量，以及参数型节点。假如我们利用该函数取代原来的分支替换方式，那么对进化的效果会有怎样的影响呢？

3. **随机交叉**　目前 crossover 函数的做法是从两棵树上的相同层级处选择分支。请再编写一个 crossover 函数，允许对任意两个随机分支进行交叉操作。这对进化的效果会有怎样的影响？

4. **终止进化过程**　请为进化过程再添加一个条件：如果最佳分值在经历过 X 代之后没有得到任何改善，则进化过程结束，并返回最终结果。

5. **隐函数**（hidden functions）　请尝试再设计几个数学函数，供程序进行推测。其中，哪些函数更容易被程序发现？而哪些则更难于发现呢？

6. **Grid War 的玩家**　请尝试自己手工设计一个能够在 Grid War 游戏中有出色发挥的程序。假如你认为设计这样一个程序不费吹灰之力，那就请尝试再手工编写一个完全不同的程序吧。与此前完全随机的初始种群不同，此处我们让种群中的大部分程序都随机生成，同时将手工设计的程序也包含进来。如何将其与随机程序进行比较？这对于进化的效果会有改善吗？

7. **井字游戏**（tic-tac-toe）　请构造一个井字游戏的模拟程序。设立一个与 Grid War 类似的比赛。看一看程序的表现如何？它们能否在学习中逐渐成长为“高手”？

8. **带数据类型的节点**　本章就如何实现带多个数据类型的节点为我们提供了一些思路。请将这些思路付诸实现，看看我们是否能对一个程序实施进化，使其能够学会返回一个字符串的第 2、第 3、第 6 和第 7 个字符（比如，“genetic”会变成“enic”）。

第 12 章

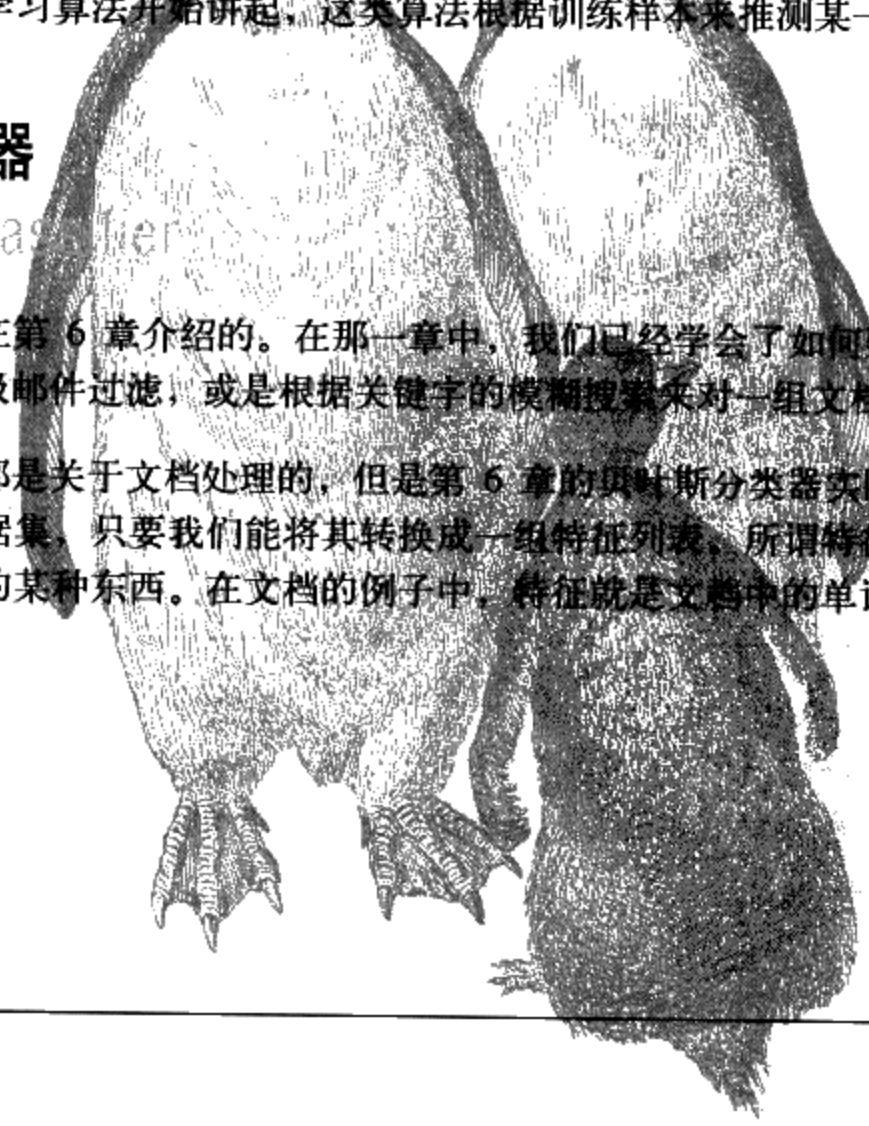
算法总结

Algorithm Summary

本书向大家介绍了若干种不同的算法，假如读者研习过书中的示例，那么现在手头就应该有大多数算法的 Python 实现代码。之前的章节在内容组织上都是围绕着一个样例问题、解决该问题的相应算法，及其变体这样的形式来介绍的。本章可以作为相关算法的一个参考。因此，假如读者希望对某个新的数据集进行数据挖掘或是机器学习，那么不妨可以参看一下此处的算法，然后再择优而用；同时，我们还可以利用已经编写好的算法实现代码，帮助我们对数据进行分析。

为了让大家不至于因查找某种算法的细节，而重新回过头去翻阅整本书，笔者会在本章中提供针对于每种算法的算法描述、工作原理的扼要概述、适用的数据集类型，以及算法实现代码的使用方法。除此以外，笔者还会提供每一种算法的优缺点说明（或者，如果你愿意的话，也可以将其理解为：怎样将这些好方法推销给你的老板☺），偶尔还会借助示例来解释算法的某些特征。这些例子都被经过了极大的简化——其中的多数例子只要大家看一下数据就知道该如何解决——但是对于演示而言，它们都是非常有价值的。

我们首先从监督学习算法开始讲起，这类算法根据训练样本来推测某一分类或某个数值。



贝叶斯分类器

Bayesian Classifier

贝叶斯分类器是在第 6 章介绍的。在那一章中，我们已经学会了如何建立一个文档分类系统，将其用于垃圾邮件过滤，或是根据关键字的模糊搜索来对一组文档进行划分。

尽管所有的例子都是关于文档处理的，但是第 6 章的贝叶斯分类器实际上也可以适用于任何其他形式的数据集，只要我们能将其转换成一组特征列表。所谓特征，就是指一个给定项中存在或缺少的某种东西。在文档的例子中，特征就是文档中的单词，但它们也可以是

某个不明对象的特有属性、一种疾病的症状，或是其他任何形式的东西，只要我们能够称其是“存在的”或“缺少的”即可。

训练

Training

和所有监督算法一样，贝叶斯分类器是利用样本进行训练的。每个样本包含了一个特征列表和对应的分类。假定我们要对一个分类器进行训练，使其能够正确判断出：一篇包含单词“python”的文档究竟是关于编程语言的，还是关于蛇的。表 12-1 给出了一个样本训练集。

表 12-1：针对一组文档的特征和分类

特征	分类
Python 是以鸟和哺乳动物为食的大蟒	蛇
Python 最初是作为一门脚本语言被开发出来的	语言
在印度尼西亚发现了一条 14.935 米（49 英尺）长的 Python	蛇
Python 具有动态类型系统	语言
拥有鲜艳表皮的 Python	蛇
开源项目	语言

分类器记录了它迄今为止见过的所有特征，以及这些特征与某个特定分类相关联的数字概率。分类器逐一接受样本的训练。当经过某个样本的训练之后，分类器会更新该样本中特征与分类的概率，同时还会产生一个新的概率，即：在一篇属于某个分类的文档中，含有指定单词的概率。例如，经过如表 12-1 所示的一组文档的训练之后，也许我们最终会得到如表 12-2 所示的一组概率。

表 12-2：单词属于某个给定分类的概率

特征	语言	蛇
dynamic	0.6	0.1
constrictor	0.0	0.6
long	0.1	0.2
source	0.3	0.1
and	0.95	0.95

从上表中我们可以看到，经过训练之后，特征与各种分类的关联性更加明确了。单词“constrictor”属于蛇的分类概率更大，而单词“dynamic”属于编程语言的分类概率更大。

另一方面，有些特征的所属分类则并没有那么明确，比如：单词“and”出现在两个分类中的概率是差不多的（单词“and”几乎会出现在每一篇文档中，不管它属于哪一个分类）。分类器在经过训练之后，只会保留一个附有相应概率的特征列表，与某些其他的分类方法不同，此处的原始数据在训练结束之后，就没有必要再加以保存了。

分类

Classifying

当一个贝叶斯分类器经过训练之后，我们就可以利用它来对新的项目进行自动分类了。假定我们有一篇新的文档，包含了特征“long”、“dynamic”和“source”。表 12-2 列出了每个特征的概率，但这些概率只是针对于各个单词而言的。如果所有单词同属于一个分类的概率值更大，那么答案显然是很清楚的。然而在本例中，“dynamic”属于语言分类的概率更大，而“long”属于蛇分类的概率更大。因此，为了真正对一篇文档进行有效地分类，我们需要一种方法能将所有特征的概率组合到一起，形成一个整体上的概率。

解决这一问题的一种方法是利用我们在第 6 章中介绍过的朴素贝叶斯分类器。它是通过下面的公式将概率组合起来的：

$$Pr(Category \mid Document) = Pr(Document \mid Category) * Pr(Category) / Pr(Document)$$

此处：

$$Pr(Document \mid Category) = Pr(Word1 \mid Category) * Pr(Word2 \mid Category) * \cdots$$

*Pr(Word | Category)*的取值来自于上表，比如：*Pr (dynamic | Language)* = 0.6。*Pr(Category)*的取值则等于某个分类出现的总体几率。因为“language”有一半的机会都会出现，所以*Pr(Language)*的值为 0.5。无论是哪个分类，只要其 *Pr(Category | Document)*的值相对较高，它就是我们预期的分类。

代码使用说明

Using Your Code

为了利用第 6 章中构造的贝叶斯分类器对数据集进行分类，我们所要做的唯一一件事情就是定义一个特征提取函数，该函数的作用是将我们用以训练或分类的数据转化成一个特征列表。在第 6 章中我们处理的是文档，所以该函数将字符串拆分成了一个个单词，但是也可以采用任何其他形式的函数，只要它接受的是一个对象，并且返回一个列表：

```
>>> docclass.getwords('python is a dynamic language')
{'python': 1, 'dynamic': 1, 'language': 1}
```

上述函数可用于创建一个新的分类器，针对字符串进行训练：

```
>>> 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')
```

然后进行分类：

```
>>> cl.classify('dynamic programming')
u'language'
>>> cl.classify('boa constrictors')
u'snake'
```

对于允许使用的分类数量，此处并没有任何的限制，但是为了使分类器有一个良好的表现，我们须要为每个分类提供大量的样本。

优点和缺点

Strengths and Weaknesses

朴素贝叶斯分类器与其他方法相比最大的优势或许就在于，它在接受大数据量训练和查询时所具备的高速度。即使选用超大规模的训练集，针对每个项目通常也只会有相对较少的特征数，并且对项目的训练和分类也仅仅是针对特征概率的数学运算而已。

尤其当训练量逐渐递增时则更是如此——在不借助任何旧有训练数据的前提下，每一组新的训练数据都有可能会引起概率值的变化。（你会注意到，贝叶斯分类器的算法实现代码允许我们每次只使用一个训练项，而其他方法，比如决策树和支持向量机，则须要我们一次性将整个数据集都传给它们。）对于一个如垃圾邮件过滤这样的应用程序而言，支持增量式训练的能力是非常重要的，因为过滤程序时常要对新到的邮件进行训练，然后必须即刻进行相应的调整；更何况，过滤程序也未必有权访问已经收到的所有邮件信息。

朴素贝叶斯分类器的另一大优势是，对分类器实际学习状况的解释还是相对简单的。由于每个特征的概率值都被保存了起来，因此我们可以在任何时候查看数据库，找到最适合的特征来区分垃圾邮件与非垃圾邮件，或是编程语言与蛇。保存在数据库中的这些信息都很有价值，它们有可能会被用于其他的应用程序，或者作为构筑这些应用程序的一个良好基础。

朴素贝叶斯分类器的最大缺陷就是，它无法处理基于特征组合所产生的变化结果。假设有如下这样一个场景，我们正在尝试从非垃圾邮件中鉴别出垃圾邮件来：假如我们构建的是一个 Web 应用程序，因而单词“online”时常会出现在你的工作邮件中。而你的好友则在一家药店工作，并且喜欢给你发一些他碰巧在工作中遇到的奇闻趣事。同时，和大多数不善于严密保护自己邮件地址的人一样，偶尔你也会收到一封包含单词“online pharmacy”的垃圾邮件。

也许你已经看出了此处的难点——我们往往会告诉分类器“online”和“pharmacy”是出现在非垃圾邮件中的，因此这些单词相对于非垃圾邮件的概率会更高一些。当我们告诉分类

器有一封包含单词“online pharmacy”的邮件属于垃圾邮件时，则这些单词的概率又会进行相应的调整，这就导致了一个经常性的矛盾。由于特征的概率都是单独给出的，因此分类器对于各种组合的情况一无所知。在文档分类中，这通常不是什么大问题，因为一封包含单词“online pharmacy”的邮件中可能还会有其他特征可以说明它是垃圾邮件，但是在面对其他问题时，理解特征的组合可能是至关重要的。

决策树分类器
Decision Tree Classifier

决策树是在第 7 章中介绍的，在那一章我们学到了如何根据服务器日志来对用户的行为进行建模。决策树以其极易理解和解释的特点而著称。图 12-1 给出了一个决策树的例子。

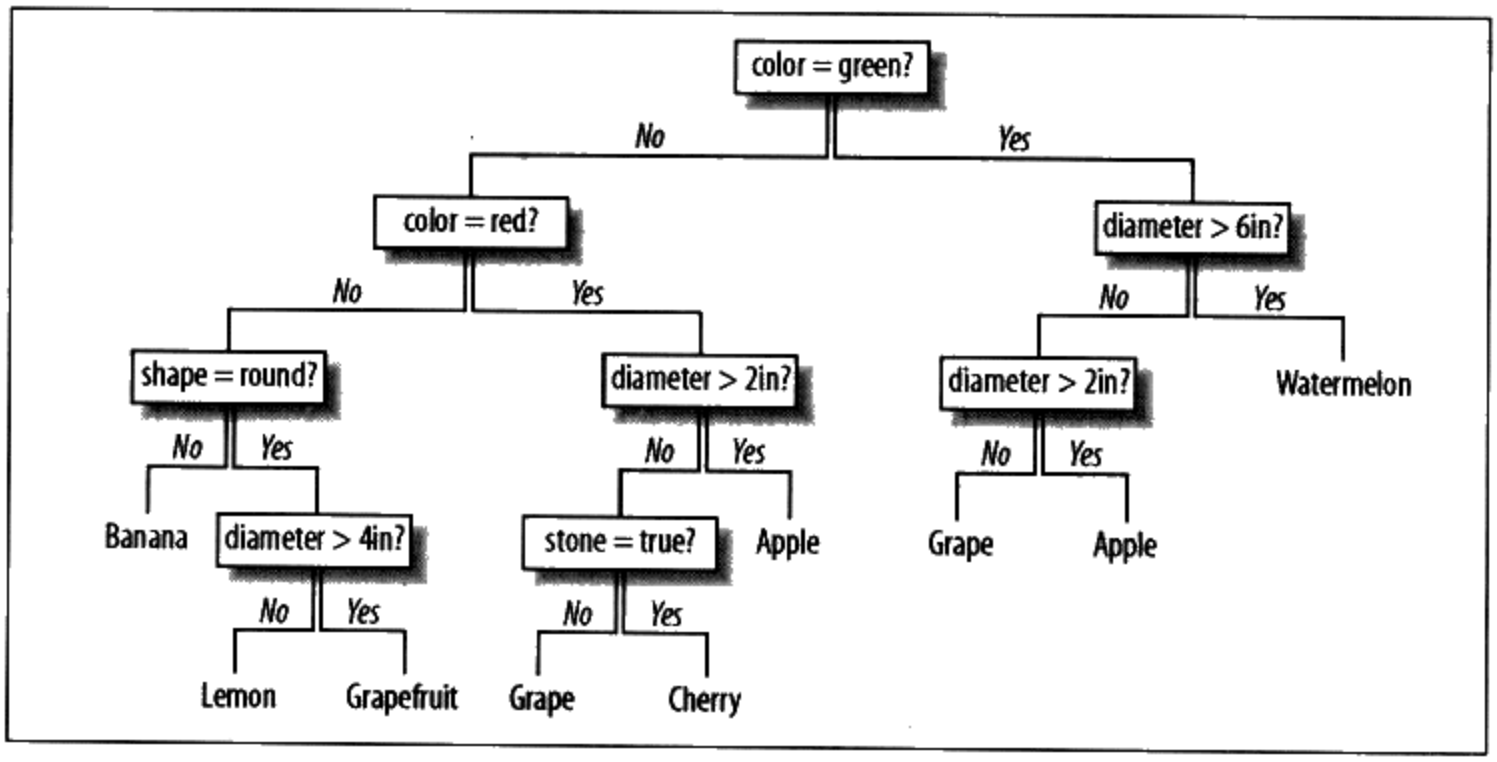


图 12-1：决策树的例子

图中清楚地给出了，当对新项目进行分类时，决策树所要做的工作。从树的根节点开始，我们会对每个节点的判断条件进行检查——如果节点的判断条件满足，就走 Yes 分支，否则，就走 No 分支。这一过程会一直重复进行，直至到达代表预测分类的那个叶节点为止。

训练
Training

利用决策树进行分类非常的简单；但对决策树进行训练则需要更多的技巧。第 7 章介绍的算法从根部开始构造决策树，在每一步中它都会选择一个属性，利用该属性以最佳的可能方式对数据进行拆分。为了说明这一点，请看如表 12-3 所示的水果数据集。我们将它作为初始数据集。

表 12-3：水果数据

直径	颜色	水果
4	Red	Apple
4	Green	Apple
1	Red	Cherry
1	Green	Grape
5	Red	Apple

为了创建树的根节点，有两个可能的变量可用于对数据进行拆分，它们分别是直径和颜色。第一步就是要对每个变量都进行尝试，从中找出拆分数据效果最好的一个。按颜色拆分数据集得到的结果如表 12-4 所示。

表 12-4：按颜色拆分后的水果数据

Red	Green
Apple	Apple
Cherry	Grape
Apple	

上述数据仍然显得非常的混乱。但是，假如我们按直径（小于 4 英寸、大于或等于 4 英寸）进行拆分，则拆分得到的结果就会变得非常清晰（我们将左侧的数据称为 Subset 1，右侧的数据称为 Subset 2）。此时的拆分情况如表 12-5 所示。

表 12-5：按直径拆分后的水果数据

直径<4 英寸	直径≥4 英寸
Cherry	Apple
Grape	Apple

这显然是一种更好的拆分结果，因为 Subset 2 包含来自初始集合中的所有“Apple”条目。尽管在本例中，哪个变量的拆分效果更好是很明确的，但是当面对规模更大一些的数据集时，就不会总是有如此清晰的拆分结果了。为了衡量一个拆分的优劣，第 7 章引入了熵的概念（代表一个集合中无序的程度）：

- *p(i) = frequency(outcome) = count(outcome) / count(total rows)*
- *Entropy* = 针对所有结果的 *p(i) * log(p(i))*之和

集合中的熵偏小，就意味着该集合中的大部分元素都是同质的（homogeneous），而熵等于 0，则代表集合中的所有元素都是同一类型的。表 12-5 中 Subset 2（直径≥4）的熵即为 0。每个集合中的熵都是用来计算**信息增益**（information gain）的，信息增益的定义如下：

- *weight1 = subset1* 的大小 / 原始集合的大小
- *weight2 = subset2* 的大小 / 原始集合的大小
- *gain = entropy(original) - weight1*entropy(set1) - weight2*entropy(set2)*

因此对于每一种可能的拆分，我们都会计算出相应的信息增益，并以此来确定拆分用的变量。一旦用以拆分的变量被选定，第一个节点就创建出来了，如图 12-2 所示。

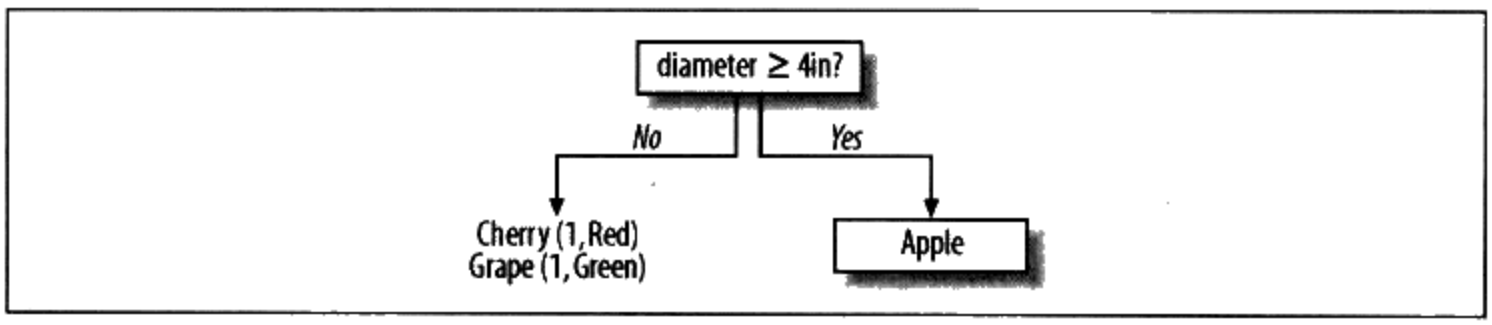


图 12-2：水果决策树的根节点

此处，我们将判断条件显示在了节点的位置，如果数据不满足判断条件，就会沿着 No 分支向下走，如果满足，则会沿 Yes 分支向下走。因为 Yes 分支现在仅有一种可能的结果，因此它就成了叶节点。No 分支依然显得有些混乱，因此可以采用与选择根节点时完全相同的办法对其做进一步的拆分。在本例中，颜色现在成为了拆分数据的最佳变量。这一过程会一直重复下去，直到在某个分支上拆分数据时不会再有信息增益为止。

决策树分类器使用说明

Using Your Decision Tree Classifier

第 7 章中决策树的算法实现代码是基于一个列表的列表进行训练的，每个内部列表都包含了一组值，其中的最后一个值代表分类。我们可以按如下所示的方法来创建前面那个简单的水果数据集

```
>>> fruit=[[4,'red','apple'],
... [4,'green','apple'],
... [1,'red','cherry'],
... [1,'green','grape'],
... [5,'red','apple']]
```

现在，我们可以对这棵决策树进行训练，并利用它对新的样本进行分类：

```
>>> 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}
```

很显然，直径 10 英尺且颜色为紫色的水果不应该是苹果，但决策树受制于其所见过的样本。最后，我们可以将树打印或绘制出来，以理解其决策的推导过程：

```
>>> treepredict.printtree(tree)
0:4?
T-> {'apple': 3}
F-> 1:green?
  T-> {'grape': 1}
  F-> {'cherry': 1}
```

优点和缺点

Strengths and Weaknesses

决策树最为显著的优点在于，利用它来解释一个受训模型是非常容易的，而且算法将最为重要的判断因素都很好地安排在了靠近树的根部位置。这意味着，决策树不仅对分类很有价值，而且对决策过程的解释也很有帮助。和贝叶斯分类器一样，可以通过观察内部结构来理解它的工作方式，同时这也有助于在分类过程之外进一步做出其他的决策。例如，第 7 章中的模型对哪些用户最终会成为付费客户进行了预测，而有了决策树，就可以清晰地显示出哪些变量是最适合用于拆分数据的，这对于规划广告策略，以及判断还应该收集哪些数据而言，都是非常有价值的。

因为决策树要寻找能够使信息增益达到最大化的分界线，因此它也可以接受数值型数据作为输入。能够同时处理分类（categorical）数据和数值数据，对于许多问题的处理都是很有帮助的——这些问题往往是传统的统计方法（比如回归）所难以应对的。另一方面，决策树并不擅长于对数值结果进行预测。一棵回归树可以将数据拆分成一系列具有最小方差的均值，但是如果数据非常复杂，则树就会变得非常庞大，以至于我们无法借此来做出准确的决策。

与贝叶斯分类器相比，决策树的主要优点是它能够很容易地处理变量之间的相互影响。一个用决策树构建的垃圾邮件过滤器可以很容易地判断出："online" 和 "pharmacy" 在分开时并不代表垃圾信息，但当它们组合在一起时则为垃圾信息。

遗憾的是，利用第 7 章中的算法所实现的垃圾邮件过滤器是不实用的。很简单，因为它不支持增量式的训练。（目前，支持增量式训练的决策树替代算法是一个活跃的研究领域）我们可以接受一大堆邮件，并构建一棵用于垃圾邮件过滤的决策树，但是无法对新收到的一封邮件单独进行训练——每次训练都必须重新开始。因为许多人都有成千上万封邮件，所以每次都重新开始是不切实际的。另外，因为节点的数量有可能会非常庞大（每一个特征都有可能存在或缺失），这些树有可能会变得异常庞大而复杂，这会导致分类效率的降低。

神经网络

Neural Networks

第 4 章向大家介绍了如何根据用户以往点击的链接简单构建一个神经网络，用以对搜索结果的排名进行调整。神经网络可以识别出哪些单词的组合是重要的，以及哪些单词对于某次查询是不重要的。我们不仅可以将神经网络用于分类，还可以将其用于数值预测问题。

第 4 章中的神经网络被当作了分类器——它针对每个链接给出一个数字，并预测数字最大者将会是用户要点击的链接。由于算法为每个链接都给出了一个数字，因此可以利用所有这些数字来改变搜索结果的排名。

有许多不同种类的神经网络。本书涉及的神经网络被称为多层感知器网络（multilayer perceptron network），之所以这样命名是因为它包含一层输入神经元（input neurons），这些神经元会将输入传递给一层或多层隐藏神经元（hidden neurons）。其基本结构如图 12-3 所示。

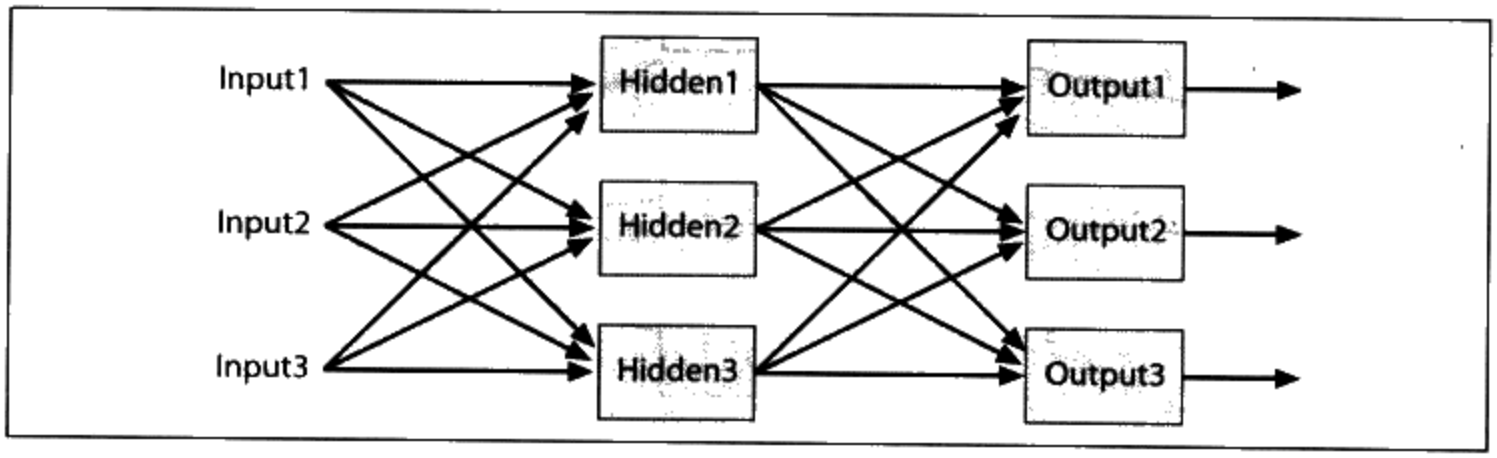


图 12-3：基本的神经网络结构

上述网络有两层神经元。层与层之间通过**突触**（synapse）彼此相连，每个突触都有一个与之关联的权重。一组神经元的输出是通过突触传入下一层的。如果从一个神经元指向下一个神经元的突触权重越大，那么它对神经元输出的影响也就越大。

作为一个简单的例子，我们再来回顾一下前述“贝叶斯分类器”一节中提到的垃圾邮件过滤问题。在简化了的邮件场景中，一封邮件可能会包含单词“online”、“pharmacy”，或二者的组合。为了判断哪些邮件是垃圾邮件，我们有可能要用到一个如下页图 12-4 所示的神经网络。

在上图中，为了解决垃圾邮件的问题，我们已将突触的权重都设置好了（我们将在下一节中了解到这些突触的设置方法）。位于第一层中的神经元对于用作输入的单词给予响应——如果某个单词存在于邮件信息中，则与该单词关联最强的神经元就会被激活。第二层神经元接受第一层神经元的输入，因此它会对单词的组合给予响应。最后，这些神经元会将结

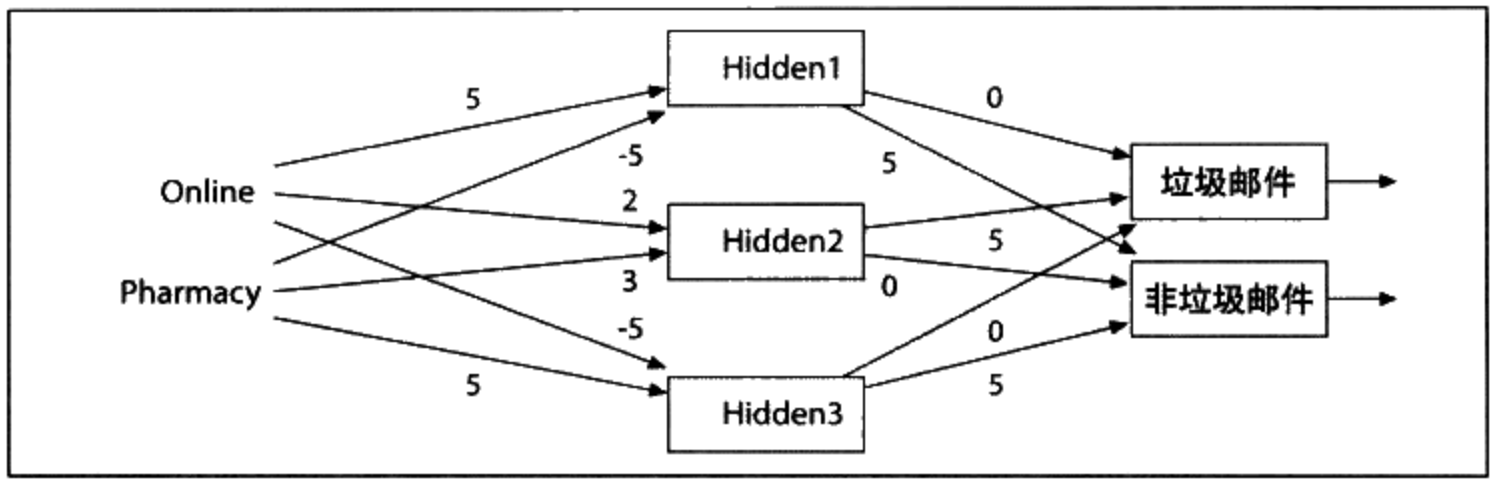


图 12-4：用于垃圾分类的神经网络

果输出，而某些单词的组合则或许会与所有可能的结果形成“强”关联或“弱”关联。最终的决策结论，就是要判定哪一个输出最强。图 12-5 给出了单词“online”在没有单词“pharmacy”跟随的情况下，神经网络对其反应的情况。

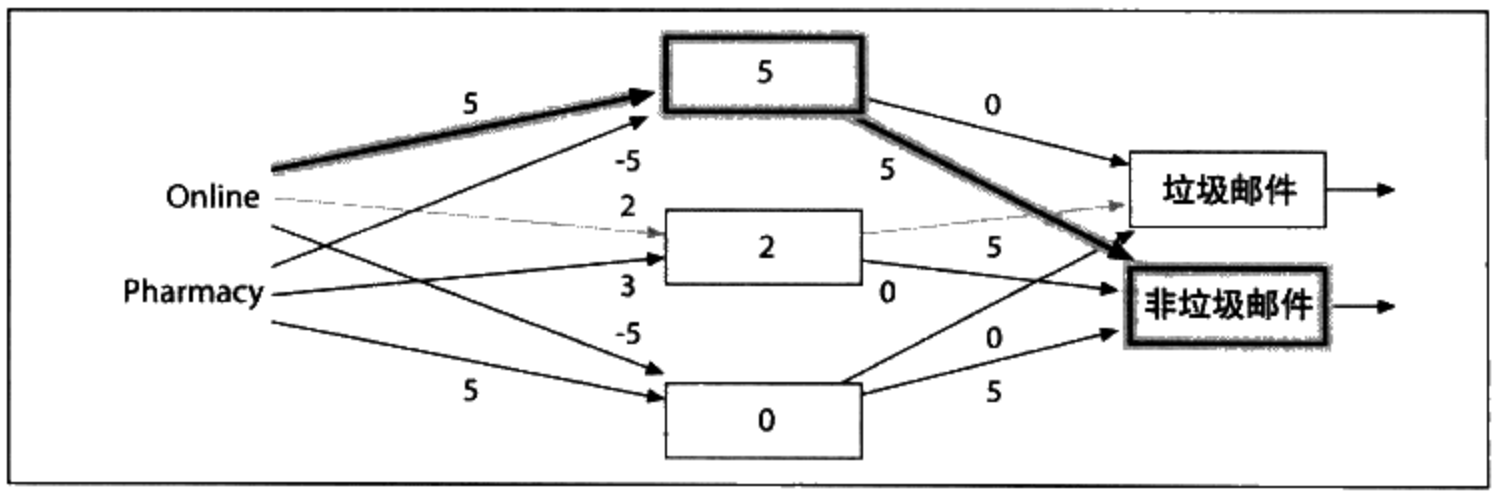


图 12-5：神经网络对单词“online”作出的反应

位于第一层中的某个神经元对“online”给予了响应，并且将其输出至第二层；而位于第二层中的某个神经元则已经学会了识别仅包含单词“online”的邮件。该神经元有一个指向非垃圾邮件的突触，其所拥有的权重值要比指向垃圾邮件的突触更大，因此邮件最终被归类为非垃圾邮件。下页图 12-6 给出了单词“online”在和“pharmacy”一起被传入神经网络时所发生的状况。

因为位于第一层中的神经元是对单个单词给予响应的，所以这次有两个神经元都被激活了。到了第二层，情况变得更有意思。“pharmacy”的存在对“online”产生了消极的影响——第一层中两个神经元的共同作用会激活第二层中间的那个神经元，该神经元经过训练之后会对“online”和“pharmacy”出现在一起的情况给予响应。由于该神经元非常明确地指向垃圾邮件分类，因此邮件就被归类成了垃圾邮件。这个例子说明了，多层神经网络可以非常轻松地处理代表不同事物的各种特征的不同组合。

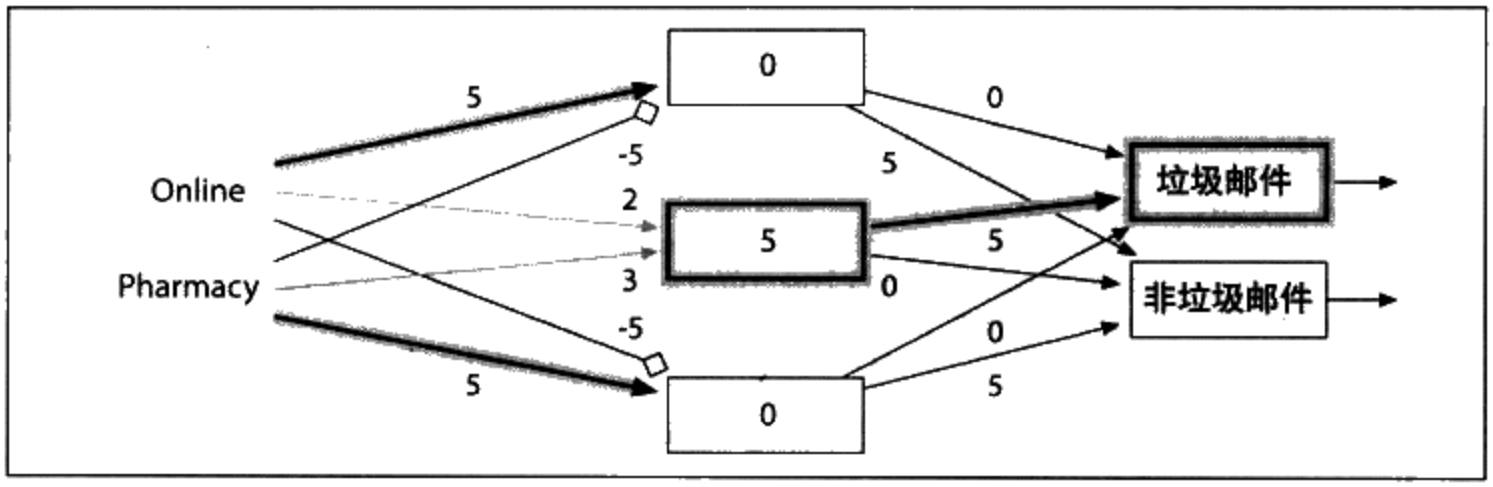


图 12-6：神经网络对“online pharmacy”作出的反应

训练神经网络

Training a Neural Network

在前述的例子中，神经网络已经为每个突触都设置了相应的权重。神经网络的真正威力在于，它们可以从随机的权重值开始，然后通过训练不断地从样本中得到学习。训练多层感知器网络最为常见的方法，也是在第 4 章中介绍的方法，被称为反向传播法（backpropagation）。

为了利用反向传播法对网络展开训练，首先从一个样本（比如单词“online”）及其正确答案（本例中为非垃圾邮件）开始。随后，将样本送入神经网络，观察其当前的推测结果。

开始的时候，网络也许会为垃圾邮件赋予一个比非垃圾邮件更高一些的权重，这是不正确的。为了修正这一错误，我们告诉网络，垃圾邮件的权重应该更接近于 0，而非垃圾邮件则应该更接近于 1。指向垃圾邮件的突触权重，会根据每个隐藏层节点的贡献程度相应地做向下微调，而指向非垃圾邮件的权重则会做向上微调。介于输入层与隐藏层之间的突触权重，也会根据其对输出层中的重要节点的贡献程度进行相应的调整。

关于上述调整的实际公式已在第 4 章中给出。为了防止网络在接受有噪声干扰或不确定性数据作为训练数据时所作出的过度补偿反应（overcompensate），整个训练过程将会缓慢地进行。如此，网络看到某个样本的次数越多，对其分类的效果就会越好。

神经网络代码使用说明

Using Your Neural Network Code

用第 4 章中的代码来解决上述问题是非常简单的。唯一的技巧在于，代码不直接接受单词作为输入，而是使用数字 ID。因此我们须要为每一个可能的输入赋予一个数字。代码使用一个数据库来保存训练数据，因此它只要打开一个指定名称的数据文件，就可以开始训练了。

```
>>> 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]
```

你会发现，神经网络接受训练的次数越多，其给出的结果就越准确。而且它还能处理偶尔出现的错误样本，并依然保持良好的预测能力。

优点和缺点

Strengths and Weaknesses

神经网络的主要优点是它们能够处理复杂的非线性函数，并且能发现不同输入间的依赖关系。尽管前面的例子只给出了 1 或 0（代表存在或缺失）这样的数字输入，但是任何数字都是可以用作输入的，并且网络也可以将评估所得的数字作为输出。

神经网络也允许增量式训练，并且通常不要求大量空间来保存训练模型，因为它们须要保存的仅仅是一组代表突触权重的数字而已。同时，也没有必要保留训练后的原始数据，这意味着，可以将神经网络用于不断有训练数据出现的应用之中。

神经网络的主要缺点在于它是一种黑盒方法。此处给出的例子设计简单，也很容易理解。但在现实中，一个网络也许会有数百个节点和上千个突触，这使我们很难确知网络何以得到最终的答案。可是无法确知推导的过程对于某些应用而言，也许是一个很大的阻碍。

神经网络的另一个缺点是，在选择训练数据的比率及与问题相适应的网络规模方面，并没有明确的规则可以遵循。最终的决定往往须要依据大量的试验。选择过高的训练数据比率，有可能会导致网络对噪音数据产生过度归纳（overgeneralize）的现象，而选择过低的训练比率，则意味着除了我们给出的已知数据外，网络有可能就不会再进一步学习了。

支持向量机
Support-Vector Machines

支持向量机（SVM）是在第 9 章介绍的，它有可能是本书中所提到的最为复杂的一种分类方法。SVM 接受数据集作为数字输入，并尝试预测这些数据属于哪个分类。例如，也许我们希望通过一组有关身高和跑动速度的数据来确定一支篮球队的队员站位。为了简化起见，此处我们只考虑两种情况——前场的位置需要身材高大的队员，而后场的位置则需要跑得更快的队员。

SVM 是通过寻找介于两个分类间的分界线来构建预测模型的。如果将一组身高与速度的对应值以及每个人的最佳站位在图上绘制出来，就会得到如图 12-7 所示的结果。其中，前场队员以 X 显示，后场队员以 O 显示。除此以外，图上还有若干条直线，将数据拆成了两个分类。

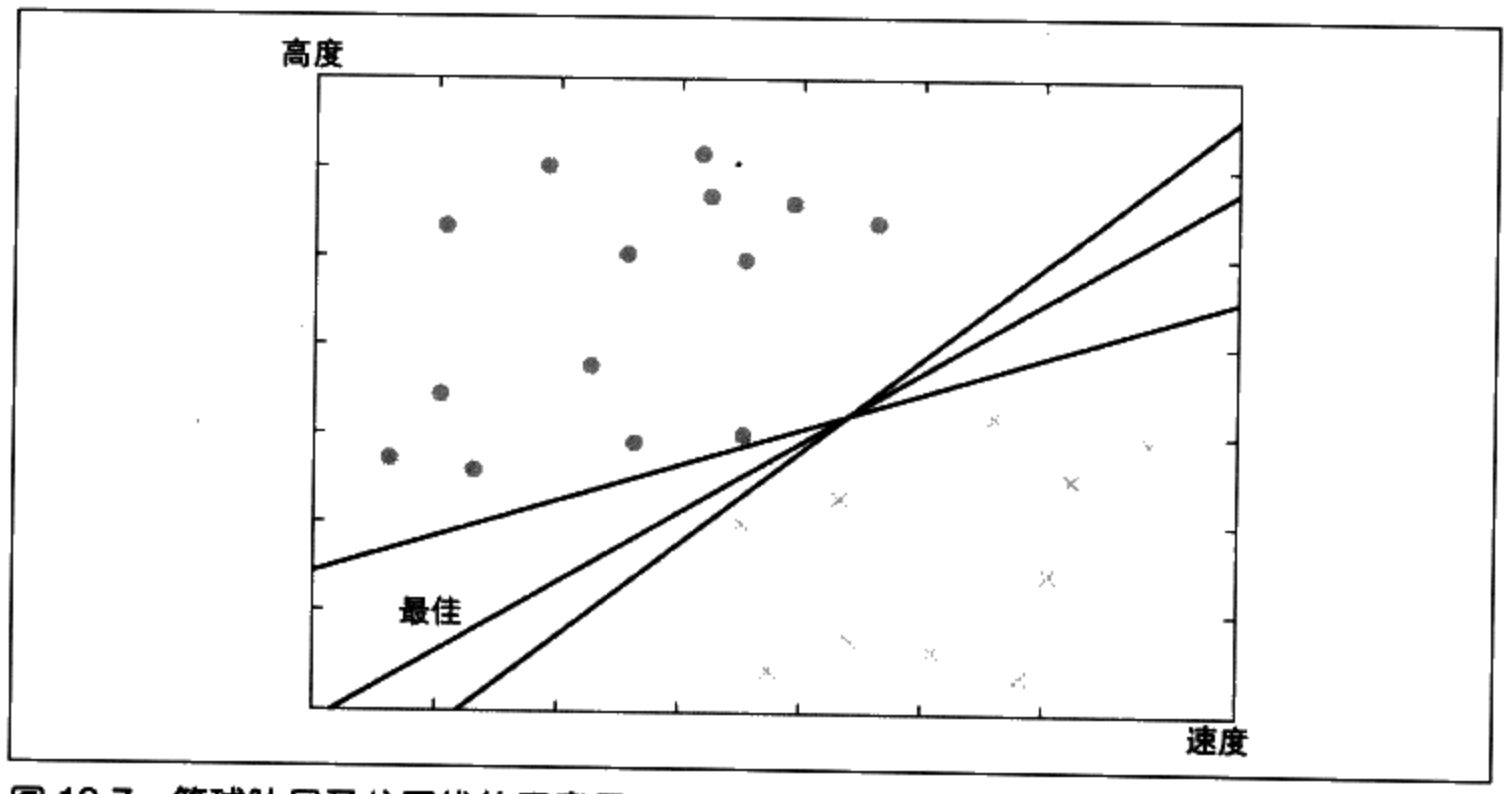


图 12-7：篮球队员及分界线的示意图

通过支持向量机找到的分界线，能够非常清晰地对数据进行划分，这意味着分界线与处于其附近的坐标点彼此间达到了最大可能距离。在图 12-7 中，尽管所有线条都可以对数据进行划分，但其中表现最好的则是标记为“最佳”的那条直线。确定这条分界线所在位置唯一需要的坐标点，是距离它最近的那些点，这些点被称为支持向量。

当分界线找到以后，对新项目的分类只须简单地将其绘制在图上，并观察其落在线的哪一侧即可。而且，一旦找到了这条分界线，对新坐标点的分类就不必再去考查训练数据了，因此分类的速度非常快。

核技法

The Kernel Trick

与其他借助于向量点积运算的线性分类器一样，支持向量机通常会借助于一种叫做**核技法**（kernel trick）的技术。为了理解这一点，假设我们所要预测的分类不是队员的站位，而是要判断队员是否适合于一支站位经常不定的业余球队，此时的情况又会如何呢？这个问题更加值得关注，因为此时的划分已不再是线性的了。我们不想要身材太高或速度太快的队员，因为他们的存在会使比赛对其他人而言难度太大，但是我们同样也不希望队员的身材太矮或速度太慢。如图 12-8 所示，图中 O 表示队员对球队是合适的，X 则表示不合适。

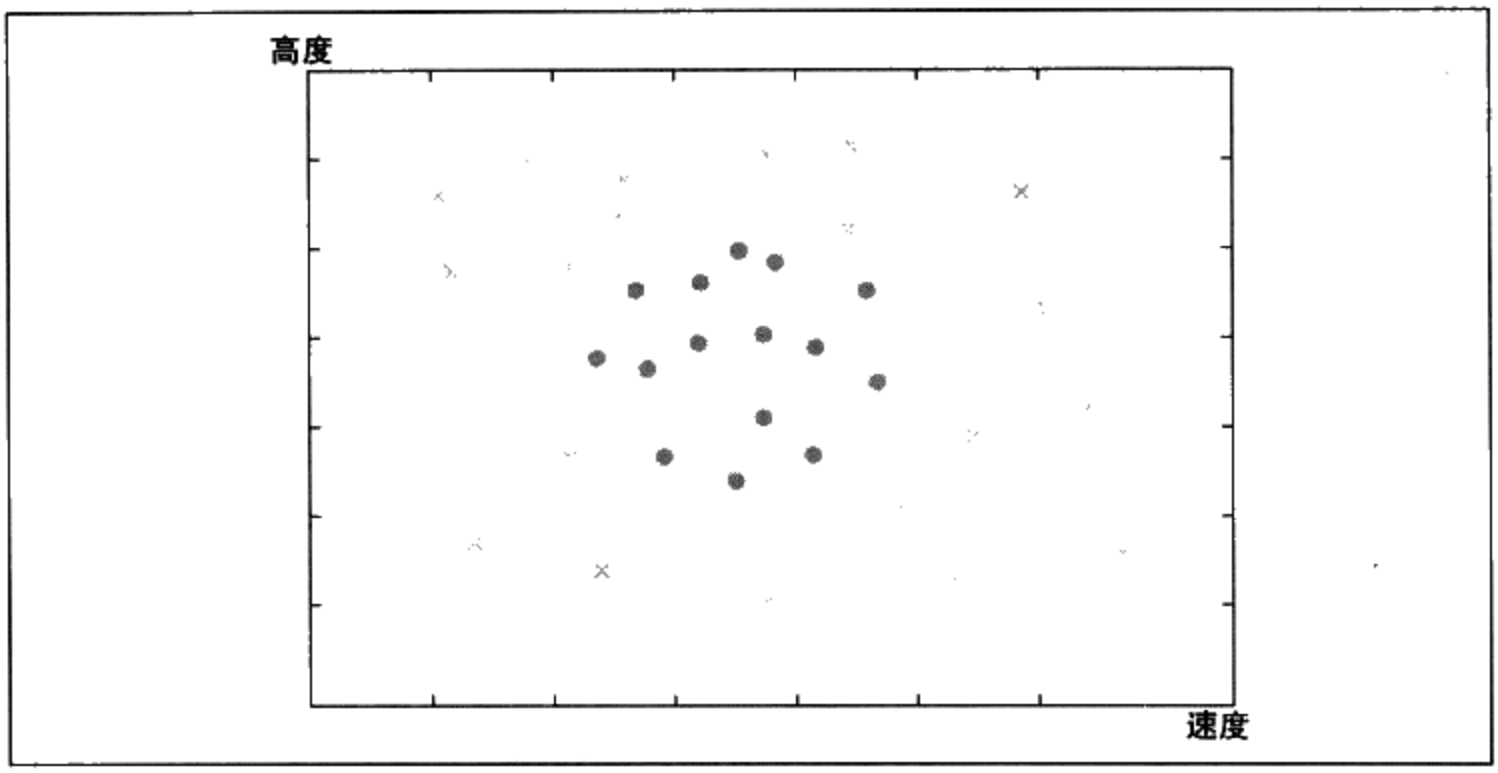


图 12-8：业余篮球队的队员示意图

在图上我们找不到任何可以划分数据的直线，因此在没有按某种方式对数据采取变换之前，我们是无法利用线性分类器来找到有效划分的。解决这一问题的一种方法是，通过在各个轴向上施以不同的函数，将数据变换到另一个不同的空间中——也许是一个超过二维的空间。在本例中，我们可以令身高和速度减去各自的平均值，然后再对两者求平方，以此来构造一个新的空间。如下页图 12-9 所示。

上述方法被称作**多项式变换**（polynomial transformation），它在不同轴向上对数据进行了变换。现在，我们很容易就能分辨出，在适合球队与不适合球队的队员之间存在着一条分界线，这条线是可以用线性分类器找到的。此时，对于新坐标点的分类，只须将坐标点变换到这一空间，然后再观察其落在线的哪一侧即可。

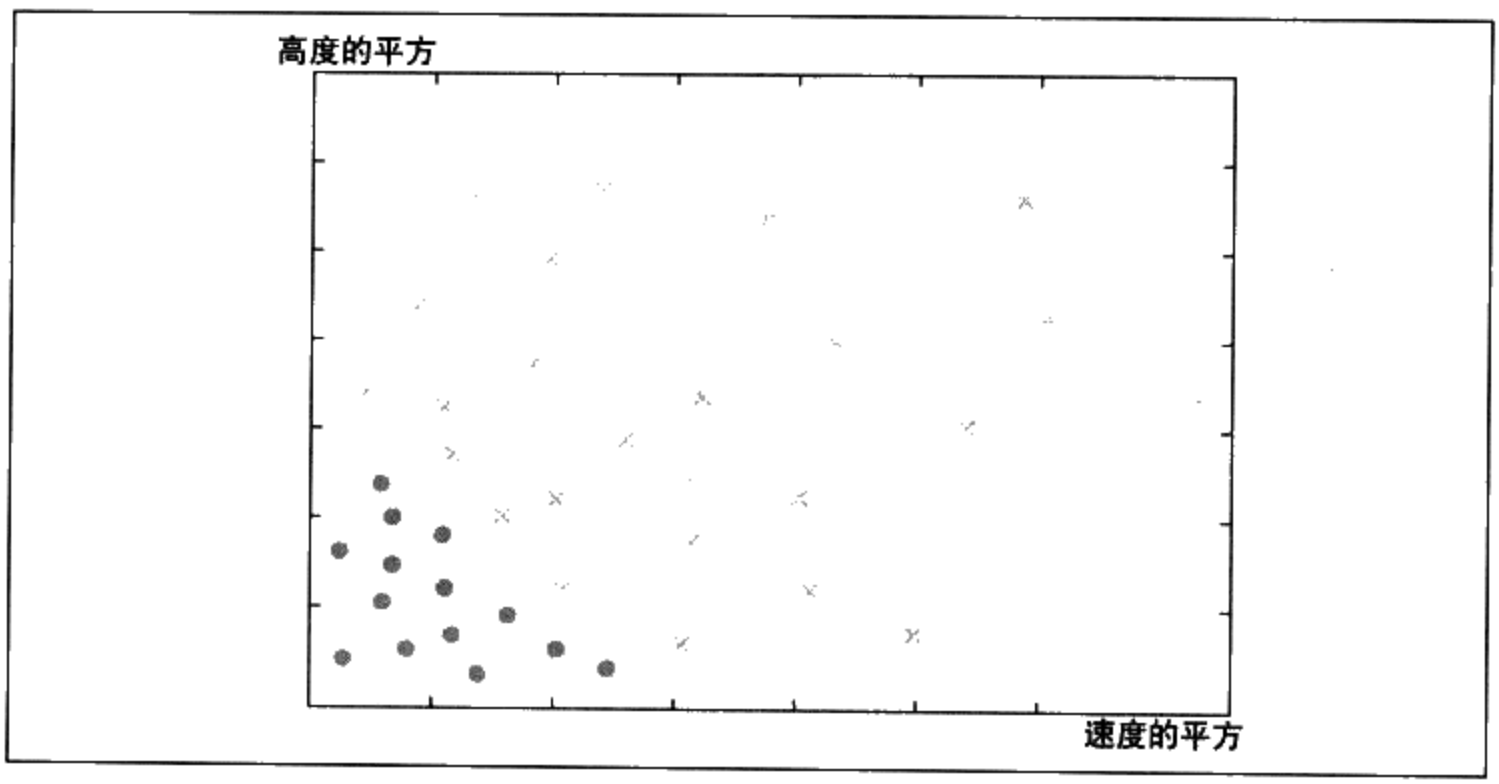


图 12-9：位于多项式空间中的篮球队员

在本例中，坐标点的变换非常成功；但是在大多数时候，为了找到分界线，往往要将坐标点变换到更为复杂的空间。这些空间的维度有的是上千维，有的甚至是无限维的，因此在现实中进行这样的变换并不总是可行的。这便是核技法的用武之地——不再进行空间的变换，而是用一个新的函数来取代原来的点积函数，该函数会在数据被变换到另一个不同的空间之后，返回相应的点积结果。例如，我们不再进行如上所示的多项式变换，而是将：

```
dotproduct(A,B)
```

替换成：

```
dotproduct(A,B)**2
```

在第 9 章，我们构造过一个利用群组均值的简单线性分类器。后来我们还对分类器进行了改造，将点积函数替换为支持向量组合的其他函数，从而使非线性问题也得到了解决。

LIBSVM 使用说明
Using LIBSVM

第 9 章曾经介绍过一个叫做 LIBSVM 的函数库。我们可以利用它对一个数据集进行训练(即在经过变换的空间中找到分界线)，然后再对新的观测数据进行分类：

```
>>> from random import randint
>>> # 随机生成 200 个坐标点
>>> d1=[[randint(-20,20),randint(-20,20)] for i in range(200)]
>>> # 对其进行分类，如果坐标点位于圆内则为 1，否则为 0
>>> 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
```

LIBSVM 支持许多不同的核函数，并且对于一个给定的数据集，可以轻松地选择不同的参数，以尝试各种不同的方法，并从中找出表现最佳者。为了测试一个模型的表现，可以试一试 cross_validation 函数，该函数接受一个参数 n，并将数据集划分成 n 个子集。随后函数会分别将每个子集当作测试集，并利用所有剩余的子集对模型加以训练。该函数最后会返回一个包含答案的列表，可以将其与原始列表进行比较：

```
>>> guesses=cross_validation(prob,param,4)
>>> sum([abs(guesses[i]-result[i]) for i in range(len(guesses))])
28.0
```

当面对一个新的问题时，可以利用不同的参数来尝试不同的核函数，以此来寻找能够给出最佳结果的方案。具体情况可能会因数据集的不同而不同，在我们得到结论之后，就可以利用相应的参数来构造模型，对新的观测数据进行分类。在实践中，我们也许会建立一些嵌套循环以尝试不同的参数值，并记录下给出最佳结果的参数组合。

优点和缺点

Strengths and Weaknesses

支持向量机是一种功能强大的分类器，一旦得到了正确的参数，这种分类器的执行效果，与本书中所提及的其他任何一种分类方法相比，有可能会不相上下或更胜一筹。而且在接受训练之后，它们在对新的观测数据进行分类时速度极快，这是因为分类时只须判断坐标点位于分界线的哪一侧即可。通过将分类输入（categorical inputs）转换成数值输入，可以令支持向量机同时支持分类数据和数值数据。

支持向量机的一个缺点在于，针对每个数据集的最佳核变换函数及其相应的参数都是不一样的，而且每当遇到新的数据集时都必须重新确定这些函数及其参数。在可能的取值范围内进行循环遍历会有助于这一问题的解决，但是这要求我们有足够大的数据集来完成可靠的交叉检验。一般而言，SVM 更适合于那些包含大量数据的问题；而其他方法，如决策树，则更适合于小规模的数据集，并且这些方法还能从数据集中得到很有价值的信息。

如同神经网络，SVM 也是一种黑盒技术——实际上，由于存在向高维空间的变换，SVM 的分类过程甚至更加难于解释。SVM 也许会给出很好的答案，但是我们永远都无法得知找到答案的真正原因。

k-最近邻

k-Nearest Neighbors

在第 8 章我们讨论了，如何利用一种称为 k-最近邻（kNN）的算法进行数值预测，并利用这一算法示范了，如何针对一组给定的样本来构造价格预测模型。我们在第 2 章中介绍过的，用于预测人们对影片或链接的喜好程度的推荐算法，同样也是 kNN 算法的一个简化版本。

kNN 的工作原理，是接受一个用以进行数值预测的新数据项，然后将其与一组已经赋过值的数据项进行比较。算法会从中找出与待预测数据项最为接近的若干项，并对其求均值以得到最终的预测结果。表 12-6 列出了一组数码相机及其百万像素数、变焦能力、销售价格等的信息。

表 12-6：数码相机及其价格

相机	百万像素数	变焦能力	价格
C1	7.1	3.8x	$399
C2	5.0	2.4x	$299
C3	6.0	4.0x	$349
C4	6.0	12.0x	$399
C5	10.0	3x	$449

假定我们想推测一款拥有 6 百万像素和 6 倍变焦透镜的新相机的价格。首先要做的第一件事情就是寻找一种方法能够对两个数据项的近似程度进行度量。第 8 章曾经用过欧几里德距离，而且我们也曾在本书中见过许多其他的距离度量方法，比如皮尔逊相关度和 Tanimoto 分值。本例所采用的是欧几里德距离，借此我们发现表中最为接近的一项是 C3。为了使这一结论可视化，假设将这些数据项绘制到一张以百万像素数为 x 轴，以焦距为 y 轴的二维图上。每个数据项本身则以其相应的价格加以标识，结果如下页图 12-10 所示。

也许你可以接受 349 美元这样的价格作为答案（毕竟这是最为接近的匹配），但是我们无法得知这是否只是一个例外情况。有鉴于此，最好选择多个最佳匹配，然后再对它们求均值。k-最近邻算法中的 k，指的就是用于求均值的最佳匹配数。例如，如果选择三个最佳匹配，并对它们求均值，那么 kNN 中的 k 即为 3。

对基本均值运算的一个扩展，是根据近邻之间距离远近的程度进行加权平均。距离非常接近的近邻会比距离稍远者拥有更高的权重值。权重与总的距离成正比。第 8 章曾经介绍过确定权重的各种不同的函数。在本例中，我们也许会对 349 美元的价格赋以最大的权重，而两个 399 美元则会赋以相对较小的权重。例如：

```
price = 0.5 * 349 + 0.25 * 399 + 0.25 * 399 = 374
```

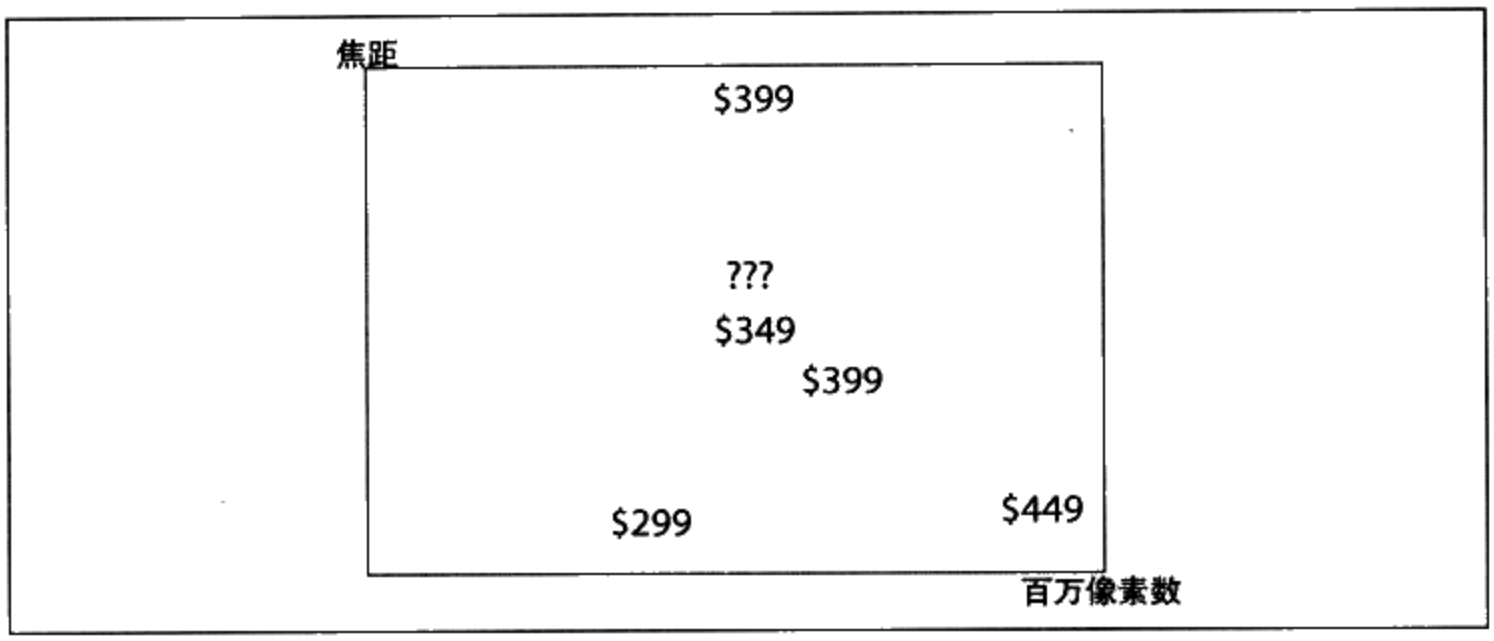


图 12-10：在焦距-百万像素数空间中的相机价格

变量缩放及多余变量

Scaling and Superfluous Variables

迄今为止我们讨论的 kNN 算法存在着一个很大的问题，那就是它在计算距离时考虑了所有的变量。这意味着，如果这些变量衡量的是不同的事物，而其中某个变量的取值又比其他变量大很多时，则该变量会对“接近”的含义形成非常大的影响。试想一下，如果上述数据集是按像素数而非百万像素数给出分辨率——相差 10 倍的变焦对于相机价格造成的影响，理应要比相差 10 个像素的分辨率大许多，但是现在它们会被视作效果相当。除此以外，有时数据集中还会包含一些对预测结果完全不起任何作用的变量，但是它们仍然会对距离的计算构成影响。

上述问题可以通过在计算距离之前对数据进行调整来加以解决。在第 8 章中，我们给出了一种数据调整的方法，该方法对某些变量的数值进行了放大，而对其他变量则做了缩小。由于完全不起作用的变量都被乘以了 0，因而这些变量对结果不再构成任何的影响。那些有价值但值域范围差别很大的变量，则被缩放到了更具可比性的程度——或许我们会将 2 000 像素的差异与 1 倍变焦的差异视作等同。

因为对数据的缩放量取决于具体的应用，所以可以通过对预测算法实施交叉验证，来判断一组缩放因子的优劣程度。交叉验证的做法是，先从数据集中去除一部分数据，然后再利用剩余数据来推测出这部分数据，算法会尝试对推测的效果进行评估。下页图 12-11 给出了交叉验证的工作原理。

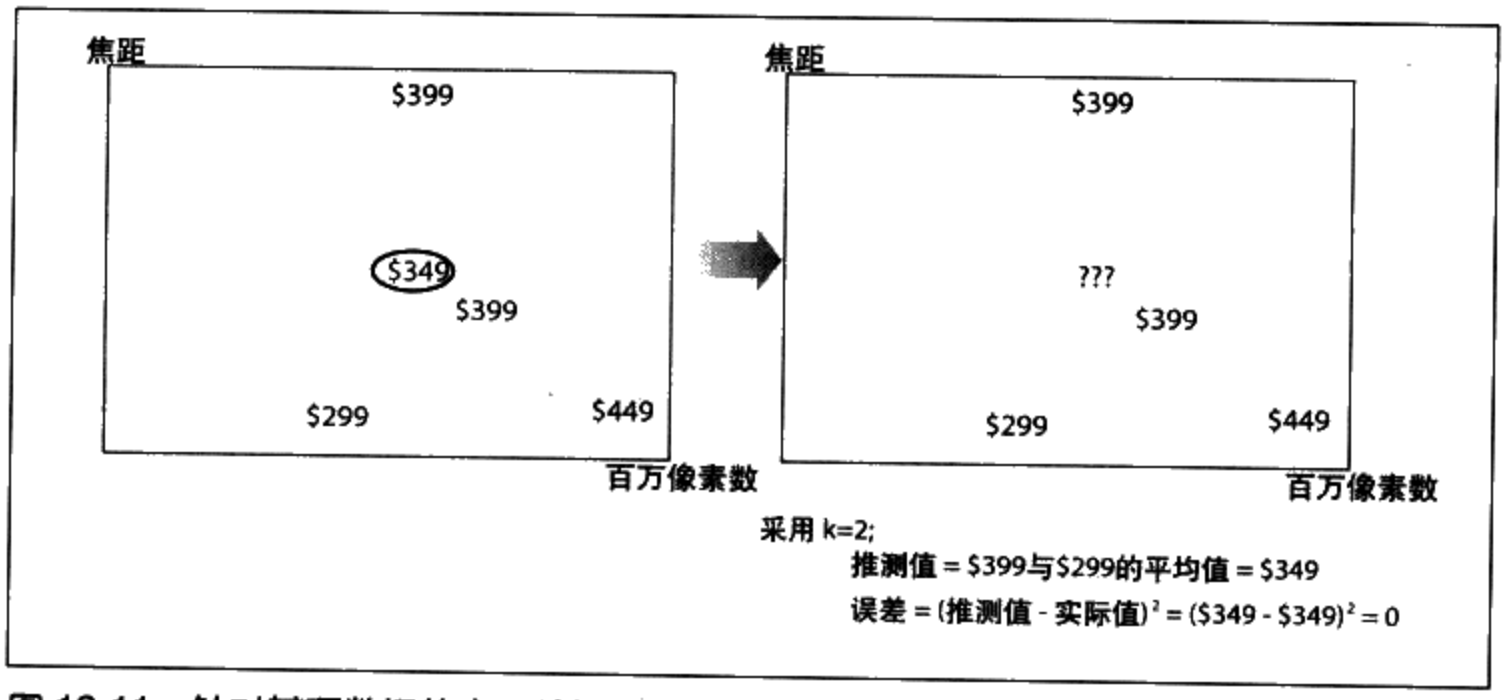


图 12-11：针对某项数据的交叉验证

通过对许多不同的缩放因子进行交叉验证，我们可以得到针对每一项数据的误差率，利用这一误差率，可以判断出：哪些缩放因子应该被用于对新数据的预测。

kNN 代码使用说明

Using Your kNN Code

在第 8 章中，我们给出了实现 kNN 和加权 kNN 算法的函数。将这些代码用于第 293 页表 12-6 给出的示例数据集是非常简单的。

```
>>> 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
```

对数据的缩放可以有效改善最终的结果。rescale 函数可以完成这一任务：

```
>>> 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}]
```

通过使用 crossvalidate，我们可以找出表现最好的缩放因子：

```
>>> numpredict.crossvalidate(knn1,cameras,test=0.3,trials=2)
3750.0
>>> numpredict.crossvalidate(knn1,scc,test=0.3,trials=2)
2500.0
```

随着数据集拥有的变量越来越多，寻找合适的缩放因子也会变得越来越乏味，因此我们可以通过循环遍历所有数值来寻找最佳的结果，或者，如第 8 章那样，借助于某种优化算法。

优点和缺点
Strengths and Weaknesses

能够利用复杂函数进行数值预测，同时又保持简单易懂的特点，k-最近邻技术便是少数几种具备如是特征的算法之一。kNN 算法的推导过程很容易理解，并且只要对代码稍做修改，就可以清晰地观察到计算过程中到底使用了哪些近邻。神经网络也是利用复杂函数进行数值预测的，但是它们显然无法为我们提供类似的例子来帮助理解推导的过程。

不仅如此，确定合理的数据缩放量不但可以改善预测的效果，而且还可以告诉我们预测过程中各个变量的重要程度。任何被缩小至 0 的变量都会被舍弃。有些时候，数据的收集也许是十分困难或代价高昂的，而此时，知道有些变量对结果没有任何影响，也许会在日后为我们省去不少的时间和金钱。

kNN 是一种在线（online）技术，这意味着新的数据可以在任何时候被添加进来，这一点不同于以支持向量机为代表的一类技术，后者在数据改变之后必须重新进行训练。而对于 kNN 而言，添加新的数据根本不须要进行任何的计算，只要将数据添加到集合中即可。

kNN 主要的缺点在于，为了完成预测，它要求所有的训练数据都必须缺一不可。面对拥有上百万样本的数据集，这不仅在空间上会有问题，在时间上也是一个问题——为了找到最为接近的数据项，每一项待预测的数据都必须和所有其他数据项进行比较。这一过程对于某些应用而言也许会显得非常低效。

kNN 的另一个缺点是，寻找合理的缩放因子可能是一项乏味的工作。尽管有不少方法可以令这一过程更趋于自动化，但是当面对一个规模庞大的数据集时，为数以千计的缩放因子实施评估和交叉验证，是一项计算量非常巨大的工作。如果有许多不同的变量等待考查，那么我们也许就须要对数百万个不同的缩放因子进行尝试，直到找到正确的因子为止。

聚类
Clustering

分级聚类和 K-均值聚类都属于非监督学习技术，即：它们不要求训练用的数据样本，因为这些方法不是用来做预测的。在第 3 章中，我们讨论了如何选择一组热门博客并自动对其进行聚类，从中可以发现哪些博客被理所当然地划归到了一类：这些博客或具有相似的描写主题，或使用了相近的词汇。

分级聚类

Hierarchical Clustering

聚类算法可以用于任何具有一个或多个数值属性的数据集。虽然第 3 章的例子中，我们使用了针对不同博客的单词计数，但是任何形式的数字集合都是可以用于聚类算法的。为了说明分级聚类算法的工作原理，请看表 12-7 中的数据项（其中包含了字母表中的一部分字母）及其数值属性。

表 12-7：用于聚类的一个简单表格

项	P1	P2
A	1	8
B	3	8
C	2	6
D	1.5	1
E	4	2

下页图 12-12 给出了针对上述数据项实施聚类的完整过程。在第一幅图中，所有数据项以二维形式分布，P1 对应于 x 轴，P2 对应于 y 轴。分级聚类的工作方式是，寻找两个距离最接近的数据项，然后将它们合二为一。在第二幅图中，我们可以看到两个最靠近的项，A 和 B，已经被合在了一起。新聚类的“位置”等于原来两个数据项位置的均值。在接下来的图中，距离最为接近的两项则变成了 C 和新的 A-B 聚类。这一过程会一直持续下去，直到如最后一幅图所示，每个数据项都被包含在了一个大的聚类中为止。

上述过程形成了一个层级结构，该层级结构可以显示为**树状图的形式**（dendrogram），树状图是一种类似于树的结构，它可以显示出哪些项和群组是紧靠在一起的。上述样例数据集的树状图如下页图 12-13 所示。

此处，距离最为接近的两项，A 和 B，最终关联在了一起。而 C 则与 A 和 B 的组合关联了起来。从该树状图中，我们可以选出任意一个枝节点，并判断其是否为一个有价值的群组。在第 3 章，我们曾经看到过几乎全部由政治类博客构成的分支，还有由技术类博客构成的分支，诸如此类。

K-均值聚类

K-Means Clustering

另一种对数据进行聚类的方法被称作 K-均值聚类。与分级聚类构造一棵由所有数据项构成的树不同，K-均值聚类实际上是将数据拆分到不同的群组中。它还要求我们在开始执行算法之前给出想要的群组数量。299 页图 12-14 给出了一个 K-均值聚类的实际例子。在这一例子中，我们尝试从一个数据集中寻找两个聚类，此处所用的数据集与分级聚类例子中用到的数据集稍有不同。

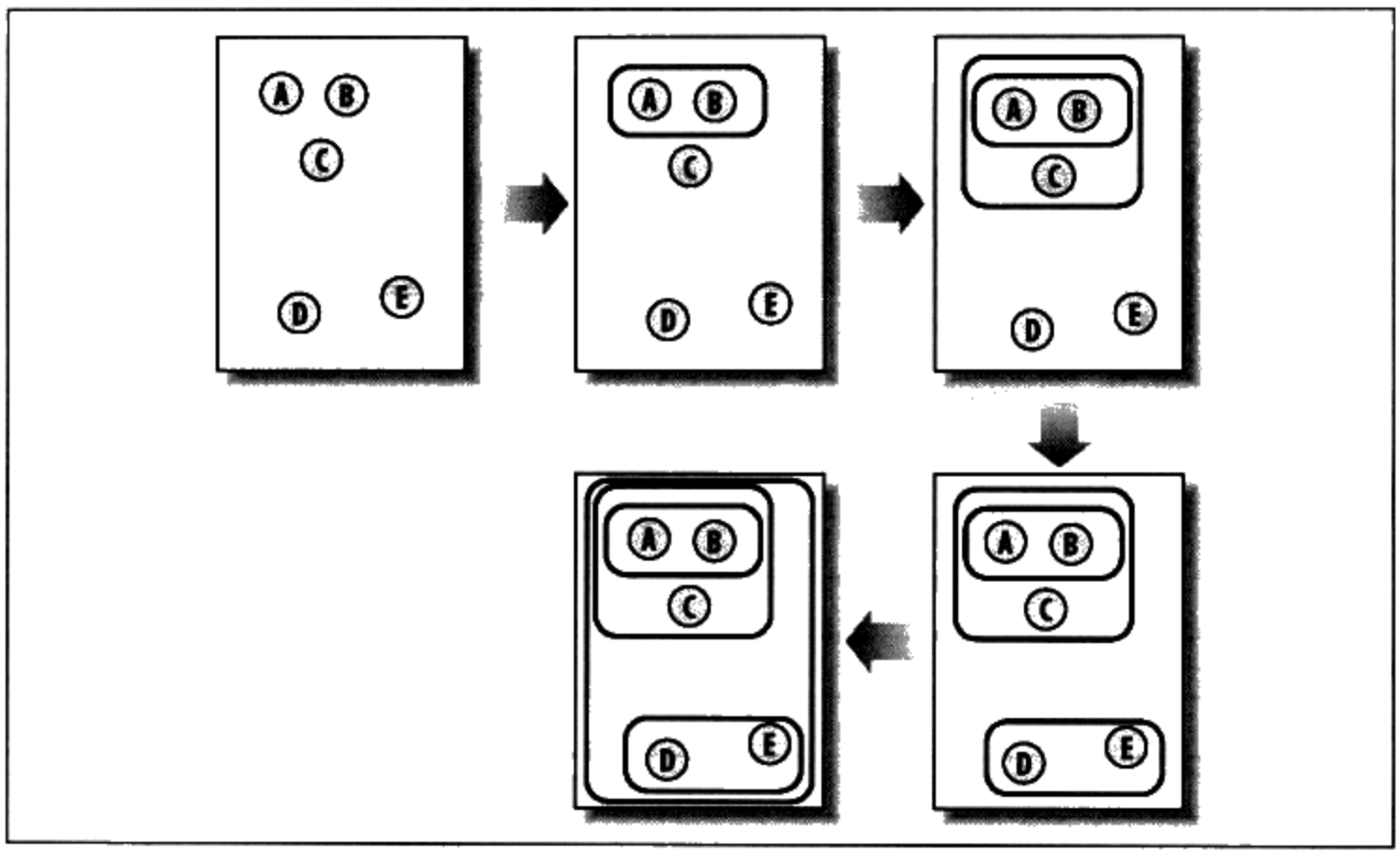


图 12-12：分级聚类的过程

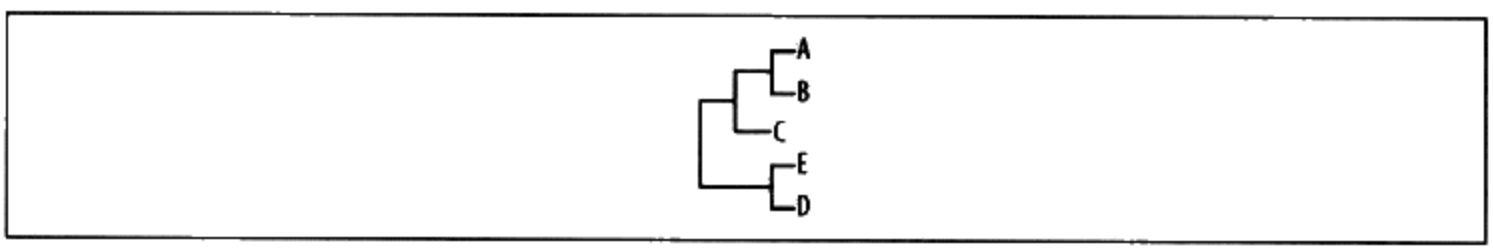


图 12-13：字母经过聚类后得到的树状图

在第一幅图中，两个中心点（以黑圈标示）的位置是随机产生的。在第二幅图中，每个数据项都被分配给了距离最近的中心点——在本例中，A 和 B 被分配给了上方的中心点，而 C、D 和 E 则被分配给了下方的中心点。在第三幅图中，中心位置已经移到了分配给原中心点的所有项的平均位置处。当再次进行分配时，我们可以看到 C 现在距离上方的中心点较之以前更为接近了，而 D 和 E 则依然是距离下方中心点最近的两项。如此，最终所得的结果是：A、B、C 在一个聚类中，而 D 和 E 则在另一个聚类中。

聚类代码使用说明

Using Your Clustering Code

为了进行聚类，我们需要一个数据集和一个距离度量方法。该数据集由一组数字列表构成，其中的每个数字代表一个变量。虽然我们在第 3 章使用了皮尔逊相关度和 Tanimoto 分值作为距离度量的方法，但是使用其他度量方法也是非常容易的，比如欧几里德距离。

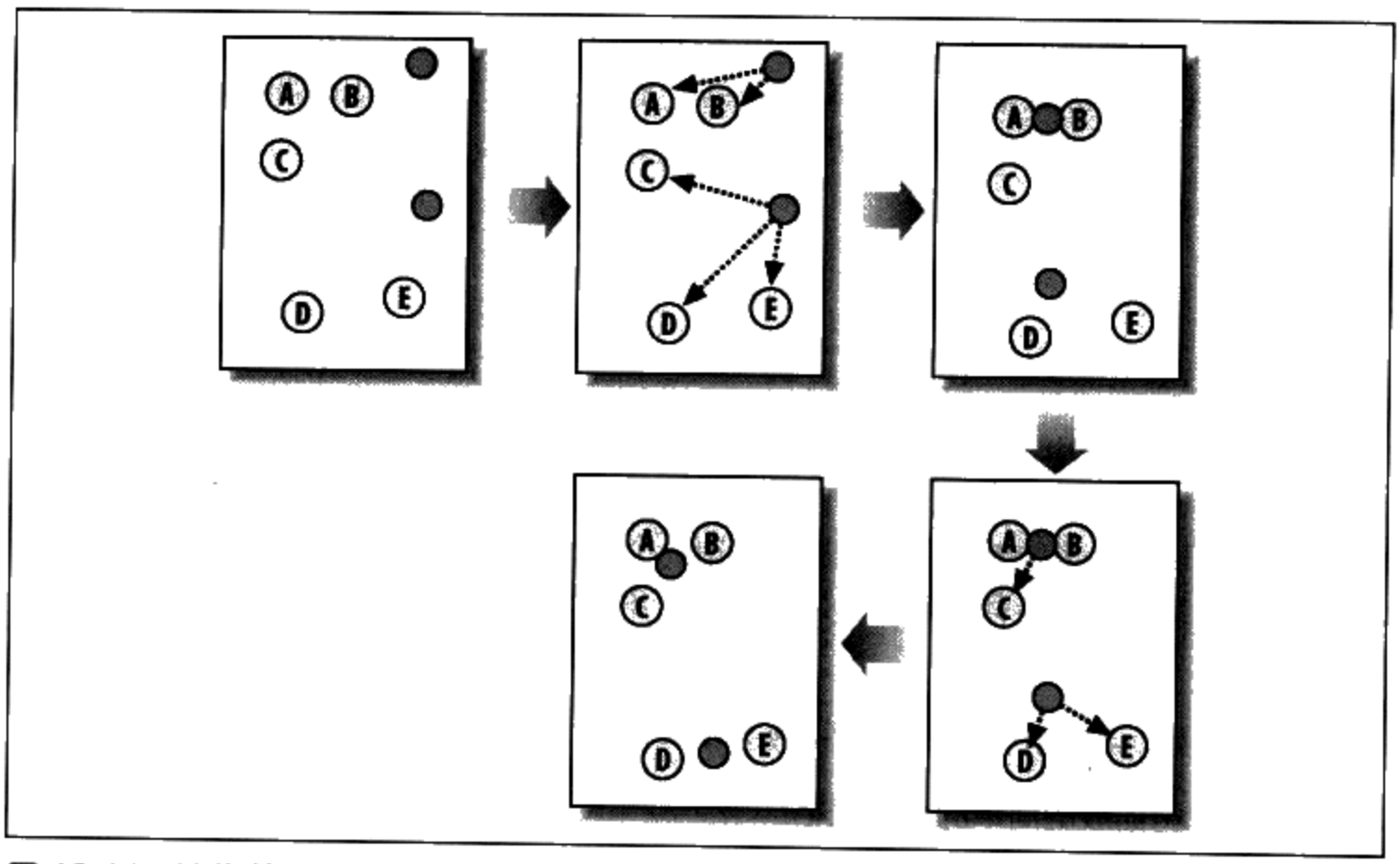


图 12-14：K-均值聚类的过程

```
>>> 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)
Iteration 0
Iteration 1
```

对于 K-均值聚类，我们可以轻松地打印出各个聚类中包含的数据项：

```
>>> kcl
[[0, 1, 2], [3, 4]]
>>> for c in kcl: print [labels[l] for l in c]
...
['A', 'B', 'C']
['D', 'E']
```

分级聚类的结果不太适合打印输出，不过我们在第 3 章所给的代码中包含了一个绘制分级聚类树状图的函数：

```
>>> clusters.drawdendrogram(hcl,labels,jpeg='hcl.jpg')
```

到底选择哪一种算法完全取决于所要处理的问题。将数据拆分到不同的群组中——比如通过 K-均值聚类得到的群组——常常是很有价值的，因为这样做更易于打印输出，也更易于识别群组的特征。而另一方面，面对一个全新的数据集，我们也许并不知道自己想要多少群组，也许只想了解其中有哪些群组彼此最为接近。在这种情况下，采用分级聚类或许是更好的选择。

此外，还可以同时借助于两种方法：先利用 K-均值聚类建立起多个群组，然后再根据中心点的间距对群组实施分级聚类。这样就会得到一系列以树型方式组织的群组，它们位于树上的不同层次，从中我们可以观察到所有群组间的关联关系。

多维缩放
Multidimensional Scaling

在第 3 章的博客例子中，我们还用到了另一种方法，被称为多维缩放。和聚类算法一样，多维缩放也是一种非监督技术，它的作用并不是要做预测，而是要使不同数据项之间的关联程度更易于理解。多维缩放会为数据集构造一个低维度的表达形式，并令距离值尽可能接近于原数据集。对于屏幕或纸张的打印输出，多维缩放通常意味着将数据从多维降至 2 维。

设想一下，比如，我们有一个如表 12-8 所示的 4 维数据集（每一项数据有 4 个相关的值）

表 12-8：一个待缩放的简单的 4 维表格

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

利用欧几里德距离公式，我们可以得到每两项间的距离值。例如，A 和 B 之间的距离为 $sqrt(0.1^2+0.15^2+0.1^2+0.0^2) = 0.2$。所有数据项两两之间的距离矩阵如表 12-9 所示。

表 12-9：上述示例的距离矩阵

	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

我们的目标是要将所有数据项绘制在一张 2 维图上，从而使各数据项在 2 维空间中的距离尽可能接近于其在 4 维空间中的距离。我们将所有数据项随机放置于图中，并且对各项之间的当前距离进行了计算，如下页图 12-15 所示。

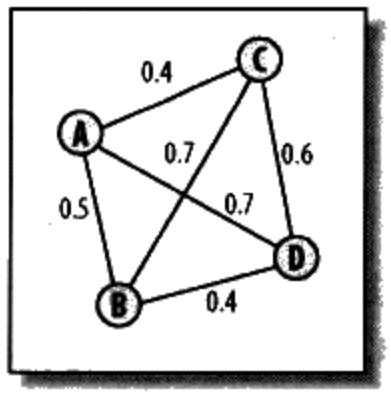


图 12-15：各项之间的距离

针对每一组数据项，我们会将目标距离与当前距离进行比较，并求出误差，然后再根据两者间的误差，将每个数据项的所在位置按比例移近或移远少许量。图 12-16 给出了我们对数据项 A 的施力情况。图中 A 与 B 之间的距离是 0.5，而两者的目标距离则仅为 0.2，因此我们必须将 A 朝 B 的方向移近一点才行。与此同时，我们还将 A 推离了 C 和 D，因为它距离 C 和 D 都太近了。

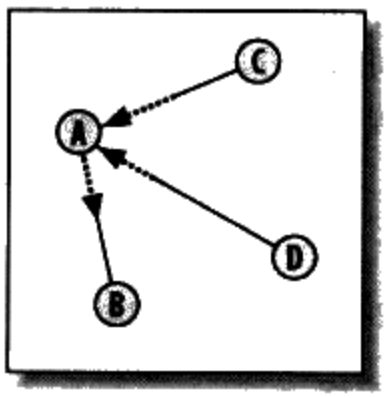


图 12-16：对数据项 A 的施力情况

每个节点的移动，都是所有其他节点施加在该节点上的推或拉的合力造成的。节点每移动一次，其当前距离和目标距离之间的差距就应该会减少一些。这一过程会不断地重复多次，直到无法再通过移动节点来减少总的误差值为止。

多维缩放代码使用说明
Using Your Multidimensional Scaling Code

在第 3 章中，我们提供了两个涉及多维缩放的函数，其中一个是用来实际运行算法的，而另一个则是用来显示结果的。第一个函数，scaledown，接受一个以多维形式表达的数据项列表，然后它以相同次序返回同一列表，并将所有数据的维度都降到了 2 维：

```
>>> 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]]
```

另一个函数，draw2d，则接受经过降维的列表，并生成一幅图片：

```
>>> clusters.draw2d(twod,labels,jpeg='abcd.jpg')
```

执行上述函数调用将会生成一个名为 *abcd.jpg* 的文件，其中包含了最终的结果。除此以外，我们也可以利用其他程序，比如电子表格软件，对 scaledown 产生的列表进行解读，从而以不同的方式将结果加以可视化。

非负矩阵因式分解

Non-Negative Matrix Factorization

第 10 章我们讨论的是一种被称为非负矩阵因式分解（NMF）的高阶技术，这一技术可以将一组数值型的观测数据拆解成不同的组分。借助这一方法，可以观察到构成新闻故事的各种不同的主题，还可以借此来了解，如何将股票交易量分解成一系列会对单支或多支股票即刻构成影响的新闻事件。同样，这也是一种非监督算法，因为其作用并非预测分类或数值，而是帮助我们识别数据的特征。

为了理解 NMF 的原理，请见表 12-10 所示的这组数据：

表 12-10：用于 NMF 的一个简单表格

观测值序号	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

假定观测值 A 和 B 是由两对数字（即特征）的某种组合构成的，但是我们并不知道这些数对到底是多少，也不知道每个数对在构造观测值时的贡献度（即权重）有多大。NMF 能够为我们找到特征和权重的可能取值。对于第 10 章中的新闻故事而言，观测值便是故事，表中的列则是故事中的单词。而对于股票交易量而言，观测值便是交易日期，表中的列则是

各支股票的交易代码。无论是哪一种情况，算法都会试图从中找出一小部分组分，将这些组分以不同数量结合在一起便可以得到此前的观测值。

针对上表中的数据，一种可能的答案是：两个数对分别为(3, 5)和(7, 2)。

借助于得到的这些组分，我们会发现，通过将各个数对以不同的数量加以组合，可以重新构造出原来的观测值，如下所示：

5*(3, 5) + 2*(7, 2) = (29, 29)
5*(3, 5) + 4*(7, 2) = (43, 33)

我们也可以将其看做是一个矩阵的乘法，如图 12-17 所示。

$$\begin{pmatrix} 5 & 2 \\ 5 & 4 \\ 5 & 0 \\ 4 & 4 \\ 1 & 3 \\ 5 & 2 \\ 3 & 4 \\ 0 & 3 \end{pmatrix} \times \begin{pmatrix} 3 & 5 \\ 7 & 2 \end{pmatrix} = \begin{pmatrix} 29 & 29 \\ 43 & 33 \\ 15 & 25 \\ 40 & 28 \\ 24 & 11 \\ 29 & 29 \\ 37 & 23 \\ 21 & 6 \end{pmatrix}$$

权重　　特征　　数据集

图 12-17：将一个数据集因式分解为权重和特征两个组分

NMF的目标是要自动找到特征矩阵和权重矩阵。为此，它以随机矩阵开始，并根据一系列更新法则对这些矩阵加以更新。根据法则我们得到了 4 个更新矩阵。在下面的说明中，我们将初始矩阵称为数据矩阵。

hn

经转置后的权重矩阵与数据矩阵相乘得到的矩阵。

hd

经转置后的权重矩阵与原权重矩阵相乘，再与特征矩阵相乘得到的矩阵。

wn

数据矩阵与经转置后的特征矩阵相乘得到的矩阵。

wd

权重矩阵与特征矩阵相乘，再与经转置后的特征矩阵相乘得到的矩阵。

为了更新特征矩阵和权重矩阵，我们首先将上述所有矩阵都转换成数组。然后将特征矩阵中的每一个值与 *hn* 中的对应值相乘，并除以 *hd* 中的对应值。类似地，我们再将权重矩阵中的每一个值与 *wn* 中的对应值相乘，并除以 *wd* 中的对应值。

这一过程会一直重复下去，直到特征矩阵和权重矩阵的乘积与数据矩阵足够接近为止。特征矩阵可以告诉我们潜藏在数据背后的诸多因素，比如：新闻的主题，还有股票市场所发生的重大事件，这些因素的共同组合可以重新构造出我们的数据集来。

NMF 代码使用说明
Using Your NMF Code

要使用 NMF 的代码，我们只须调用 `factorize` 函数，并传入一组观测数据，以及希望找到的潜在特征的数量：

```
>>> 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]])
```

函数最终会返回权重和特征。或许每次返回的结果都不尽相同，因为对于一个规模不是很大的观测数据集而言，有效的特征可能不止一个。观测数据集的规模越大，返回一致性结果的可能也就越大，尽管这些特征的返回次序也许稍有不同。

优化
Optimization

优化是在第 5 章中介绍的，和其他方法稍有不同，优化不是要处理数据集，而是要尝试找到能够使成本函数的输出结果达到最小化的值。第 5 章给出了成本函数的几个例子，比如：根据价格和候机时间的组合来安排组团旅游，为学生分配最为适宜的宿舍，以及优化简单网络图的布局。一旦设计好成本函数，我们就可以利用同样的算法来解决上述这三个不同的问题。此处我们讨论其中的两种算法：模拟退火和遗传算法。

成本函数

The Cost Function

成本函数接受一个经推测得到的题解，并返回一个数值结果，该值越大就表示题解的表现越差，该值越小就表示题解的表现越好。优化算法利用该函数对各种题解进行检验，并从中找出最优解。通常，用于优化的成本函数有许多变量须要考虑，而且有时我们并不清楚到底要修改其中的哪个变量，才能使最终结果的改善效果达到最好。不过，为了说明问题，此处我们只考虑包含一个变量的函数，定义如下：

```
y = 1/x * sin(x)
```

图 12-18 给出了该函数的图示。

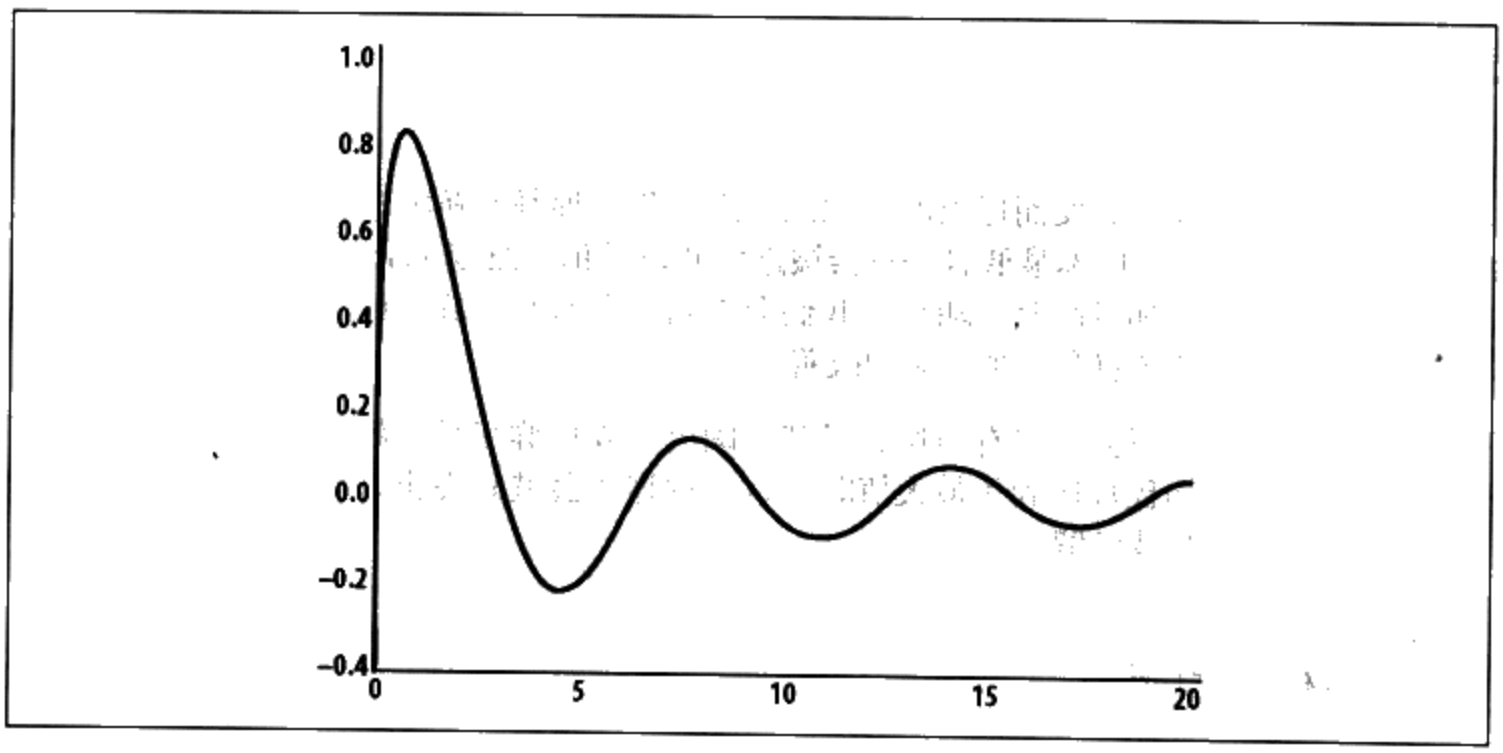


图 12-18：1/x * sin x 的图示

因为上述函数仅有一个变量，所以从图中我们很容易就可以找到函数的最低点。我们将以此来说明优化算法的工作原理；在现实中，当面对一个带有多个变量的复杂函数时，寄希望于将其简单绘制出来以寻找最低点这样的做法是行不通的。

该函数值得关注的一个地方是，它有多个局部最小值。这些点所处的位置要低于周围所有的点，但它们未必是全局意义上的最低点。这意味着，尝试随机选择题解并沿斜坡向下的方法未必一定能够找到最优解，因为我们有可能会陷入一个包含局部最小值的区域，而永远无法找到全局范围内的最小值。

模拟退火

Simulated Annealing

模拟退火，是受物理学领域中合金冷却的启发而提出的，它以一个随机推测的题解开始，然后以此为基准随机选择一个方向，并就近找到另一个近似解，判断其成本值。算法希望借此来改善题解的表现：如果题解的成本变小，则新的题解将取代原来的题解。如果成本较之原来变大了，则新题解取代旧题解的概率就取决于当前的温度值。此处的温度，会以一个相对较高的数值开始缓慢下降。正因如此，算法在执行的早期阶段会更容易接受表现相对较差的题解，这样我们就有效地避免了陷入局部最小值的可能。

当温度到达 0 时，算法便返回当前的题解。

遗传算法

Genetic Algorithms

遗传算法是受进化理论启发而提出的。它以一组被称为种群的随机题解开始。种群中表现最为优异的成员——即成本最低者——会被选中并通过稍事改变（即变异）或特征组合（即交叉或配对）的方式加以修改。随后，我们会得到一个新的种群，称之为下一代。经过连续数代之后，题解最终将会得到相应的改善。

上述过程会一直持续下去，只有当达到了某个阈值，或种群在经历数代之后没有得到任何改善，又或是遗传的代数达到了最大值时，这一过程才会就此终止。算法最终将返回在任何一代种群中发现的最优解。

优化代码使用说明

Using Your Optimization Code

不论上述哪一种算法，我们都须要定义一个成本函数，并确定题解的值域范围。此处的值域就是每个变量可能的取值范围。在这个简单的例子中，我们不妨使用[(0,20)]，即变量取值介于 0 和 20 之间。随后，我们便可以选择调用任何一个优化函数，并传入相应的成本函数及值域范围作为参数：

```
>>> import math
>>> def costf(x): return (1.0/(x[0]+0.1))*math.sin(x[0])
>>> domain=[(0,20)]
>>> optimization.annealingoptimize(domain,costf)
[5]
```

无论处理何种问题，为了对参数进行调整，并在执行时间和题解质量之间取得平衡，或许我们都有必要将优化算法执行多次。当为一组彼此相近的问题构造优化程序时——比如前述的旅游规划，这些问题目标一致但底层细节（此处为飞行时间和价格）却有所不同，可以针对相关参数进行实验，并借此确定出适合此类问题的参数设置，随后便可以始终保持这些参数设置固定不变，只要遇到的问题都同属于一类。

将机器学习、开放 API，以及面向公众的参与模式结合在一起，将会迸发出各种各样的可能性。不仅如此，随着算法的不断精炼，开放 API 的不断涌现，以及越来越多的人们成为在线应用的积极参与者，这种可能性在不久的将来还会持续扩大。希望本书能够对读者构建集体智慧的相关应用有所帮助，并且能够启迪大家寻找更多新的机遇！

第三方函数库
Third-Party Libraries

本书提到了许多第三方函数库，我们利用这些函数库来收集、存储和分析数据。本附录包含了有关这些函数库的下载和安装说明，还有一些使用的示例。

Universal Feed Parser

Universal Feed Parser 是由 Mark Pilgrim 编写的一个 Python 库，它可以用以分析 RSS 和 Atom 的订阅源。在本书中，我们利用该函数库从在线的新闻站点下载博客帖子和文章。 Universal Feed Parser 的首页地址是 *http://feedparser.org*。

在所有平台上安装
Installation for All Platforms

该函数库的下载地址是 *http://code.google.com/p/feedparser/downloads/list*。请下载文件 *feedparser*-X.Y.zip 的最新版本。

将 zip 文件解压到一个空目录下。在命令行输入：

```
c:\download\feedparser>python setup.py install
```

上述命令将定位我们的 Python 安装路径，并将函数库安装到该路径下。待安装完毕之后，就可以在自己的 Python 提示符下输入 `import feedparser`，开始使用该函数库了。

http://feedparser.org 上还提供了一些有关函数库的使用示例。

Python Imaging Library

Python Imaging Library（PIL）是一个开源的函数库，它为 Python 增加了图像生成和处理能力。它支持各种图形绘制操作和文件格式。其首页地址是 *http://www.pythonware.com/products/pil*。

在 Windows 下安装
Installation on Windows

PIL 有一个 Windows 安装程序可供下载。在其首页上，请滚动鼠标至下载区，然后根据我们所安装的 Python 版本选择下载最新的 Windows 可执行版本。运行下载后的文件，并按照屏幕上的指示一步步完成安装。

在其他平台上安装
Installation on Other Platforms

对于其他非 Windows 平台而言，我们须要从源文件中编译生成相应的函数库。函数库的源代码可以从其首页下载到，这些代码可以运行在近期发布的任何一个 Python 版本之上。

在安装之前，请先下载最新版本的源代码，在命令行中输入如下命令，并将 1.1.6 字样替换为我们下载时所选择的版本：

```
$ gunzip Imaging-1.1.6.tar.gz
$ tar xvf Imaging-1.1.6.tar
$ cd Imaging-1.1.6
$ python setup.py install
```

上述命令将对解压缩后的文件实施编译，并将函数库安装到 Python 所在的目录下。

简单的使用示例
Simple Usage Example

在这一示例中，我们创建了一个小小的图片，在上面绘制了几条直线，并写入了一行文本。然后又将图像保存成了 JPEG 文件。

```
>>> from PIL import Image,ImageDraw
>>> img=Image.new('RGB',(200,200),(255,255,255))  # 200×200 的白色背景
>>> draw=ImageDraw.Draw(img)
>>> draw.line((20,50,150,80),fill=(255,0,0))       # 红色线条
>>> draw.line((150,150,20,200),fill=(0,255,0))     # 绿色线条
>>> draw.text((40,80),'Hello!',(0,0,0))            # 黑色文本
>>> img.save('test.jpg','JPEG')                    # 保存到 test.jpg 文件
```

更多的示例可以访问 *http://www.pythonware.com/library/pil/handbook/introduction.htm*。

Beautiful Soup

Beautiful Soup 是一个 HTML 和 XML 文档的 Python 解析器。它可以用以对书写不规范的网页进行处理。在本书中，我们利用 Beautiful Soup，从不提供 API 调用的 Web 站点获取网页并构造数据集，还利用它在网页中查找索引所需的文本。Beautiful Soup 的首页地址是 *http://www.crummy.com/software/BeautifulSoup*。

在所有平台上安装
Installation on All Platforms

Beautiful Soup 可以以单个源文件的形式下载。在首页靠近底部的位置，有一个 *BeautifulSoup.py* 的下载链接。只要将其下载下来，并放入我们的工作目录或 Python/Lib 目录下即可。

简单的使用示例
Simple Usage Example

在这一示例中，我们对 Google 首页的 HTML 文本进行了解析，并示范了从 DOM 树上提取元素和查找链接的方法。

```
>>> 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'
```

更多的示例可以访问 *http://www.crummy.com/software/BeautifulSoup/documentation.html*。

pysqlite

pysqlite 是嵌入式数据库 SQLite 的 Python 接口。不同于传统的数据库，嵌入式数据库不在单独的服务器进程中运行，因此其安装和设置的过程非常简单。SQLite 将整个数据库保存在了一个单独的文件中。在本书中，我们为大家示范了如何利用 pysqlite 将收集到的数据持久化到数据库中。

pysqlite 的首页地址是 *http://www.initd.org/tracker/pysqlite/wiki/pysqlite*。

在 Windows 下安装
Installation on Windows

在 pysqlite 的首页上有针对 Windows 平台的二进制安装程序的下载链接。只要将其下载下来，然后双击运行。安装程序会询问我们的 Python 的安装目录，并将 pysqlite 安装到我们所指定的目录下。

在其他平台上安装
Installation on Other Platforms

对于其他非 Windows 平台而言，我们须要利用源文件来安装 pysqlite。可以从 pysqlite 的首

页下载到 tarball 形式的源代码。请下载文件的最新版本，在命令行中输入如下命令，并将 2.3.3 字样替换为我们下载时所选择的版本：

```
$ 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
```

简单的使用示例

Simple Usage Example

在这一示例中，我们新建了一张表，并在表中新添了一条记录，将这两项操作提交之后，又对刚插入表中的记录进行了查询：

```
>>> 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')
```

请注意，对于 SQLite 而言，字段的类型是可以省略的。为了使上述代码也能适用于更为传统的数据库，或许我们应该在建表时将字段类型加入到 SQL 声明语句中。

NumPy

NumPy 是一个 Python 的数学函数库，它提供了一个数组对象、一组与线性代数相关的函数，以及傅立叶变换函数。利用 Python 来进行科学计算是时下很流行的一种做法，而且这种做法很受大家的欢迎，在某些情况下甚至已经逐渐在取代像 MATLAB 这样的专业化的科学计算工具。在第 10 章中，我们利用 NumPy 库实现了 NMF 算法。NumPy 的首页地址是 *http://numpy.scipy.org*。

在 Windows 下安装

Installation on Windows

NumPy 有一个 Windows 版的二进制安装程序，我们可以在 *http://sourceforge.net/project/showfiles.php?group_id=1369&package_id=175103* 处下载到这一文件。

请选择与我们的 Python 版本相匹配的*.exe* 文件，下载并运行。安装程序会询问我们 Python 的安装目录，并将 NumPy 安装到我们所指定的目录下。

在其他平台上安装

Installation on Other Platforms

在其他平台上，我们可以使用源文件来安装 NumPy，NumPy 的源代码可以在 *http://sourceforge.net/project/showfiles.php?group_id=1369&package_id=175103* 处下载到。

请下载与我们所安装的 Python 版本相匹配的 *tar.gz* 文件。要从源文件进行安装，请输入如下命令，并将 1.0.2 字样替换为我们下载时所选择的版本：

```
$ gunzip numpy-1.0.2.tar.gz
$ tar xvf numpy-1.0.2.tar.gz
$ cd numpy-1.0.2
$ python setup.py install
```

简单的使用示例

Simple Usage Example

在这一示例中，我们为大家示范了构造矩阵与矩阵联乘的方法，还有转置（transpose）和平滑（flatten）操作的使用。

```
>>> 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 是一个 Python 的 2 维图形库，用它绘制数学图形的效果要比 Python Imaging Library 更为出色。利用 matplotlib 生成的图形，其质量非常之高，完全可以做出版之用。

安装

Installation

在安装 matplotlib 之前，我们须要事先安装 NumPy，关于 NumPy 的安装前文已经介绍过了。matplotlib 提供了适用于所有主流平台的二进制安装包，其中包括了 Windows 的、Mac OS X 的、基于 RPM 的 Linux 发布包和基于 Debian 的 Linux 发布包。我们可以在 *http://matplotlib.sourceforge.net/installing.html* 处找到在任何平台上安装 matplotlib 的详细说明。

简单的使用示例

Simple Usage Example

在这一示例中，我们将使用橙色小圆圈分别在坐标为(1,1)、(2,4)、(3,9)和(4,16)的地方绘制4个点。然后再将输出的图形保存成文件，并将结果显示在屏幕上的一个窗口中。

```
>>> from pylab import *
>>> plot([1,2,3,4], [1,4,9,16], 'ro')
[<matplotlib.lines.Line2D instance at 0x01878990>]
>>> savefig('test1.png')
>>> show()
```

我们还可以在 *http://matplotlib.sourceforge.net/tutorial.html* 处找到大量 matplotlib 的使用范例。

pydelicious

pydelicious 是一个从社会化书签网站 del.icio.us 获取数据的函数库。del.icio.us 本身有一个官方的 API，提供了一部分方法调用，而 pydelicious 则在此基础上增加了一些附加特性，在第 2 章中，我们利用这些特性构造了推荐引擎。pydelicious 目前位于 Google code 上，我们可以在 *http://code.google.com/p/pydelicious/source* 处访问到它。

在所有平台上安装

Installation for All Platforms

如果我们已经安装了 Subversion 版本控制软件，那么获取 pydelicious 的最新版本就相当容易了。我们只须在命令行输入如下命令即可：

```
svn checkout http://pydelicious.googlecode.com/svn/trunk/pydelicious.py
```

如果没有安装 Subversion，则可以在 *http://pydelicious.googlecode.com/svn/trunk* 处下载 pydelicious 的文件。

待文件下载完毕之后，只须在下载目录下运行 **python setup.py install** 即可。运行的结果会将 pydelicious 安装到 Python 的安装目录下。

简单的使用示例

Simple Usage Example

pydelicious 提供了许多方法调用，我们可以利用这些调用来获取热门书签，或者特定用户的书签。pydelicious 还允许我们将新书签添加到自己的账号中。

```
>> 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
```

附录 B

数学公式

Mathematical Formulas

在本书中，笔者向大家介绍了许多数学概念。本附录从中选择了一部分概念给予说明，并给出了相关公式和实现代码。

欧几里德距离

Euclidean Distance

欧几里德距离是指多维空间中两点间的距离，这是一种用直尺测量出来的距离。如果我们将两个点分别记作$(p_1, p_2, p_3, p_4, \cdots)$ 和$(q_1, q_2, q_3, q_4, \cdots)$，则欧几里德距离的计算公式如方程式 B-1 所示。

$$\sqrt{(p_1-q_1)^2+(p_2-q_2)^2+\ldots+(p_n-q_n)^2} = \sqrt{\sum_{i=1}^{n}(p_i-q_i)^2}$$

方程式 B-1：欧几里德距离

下面是上述公式的一个明确实现：

```
def euclidean(p,q):
  sumSq=0.0

  # 将差值的平方累加起来
  for i in range(len(p)):
    sumSq+=(p[i]-q[i])**2

  # 求平方根
  return (sumSq**0.5)
```

在本书中，我们曾经多次利用欧几里德距离来判断事项两两间的相似程度。

皮尔逊相关系数

Pearson Correlation Coefficient

皮尔逊相关系数是一种度量两个变量间相关程度的方法。它是一个介于 1 和-1 之间的值，其中，1 表示变量完全正相关，0 表示无关，-1 则表示完全负相关（译注 1）。

方程式 B-2 给出了皮尔逊相关系数的计算公式。

$$r = \frac{\sum XY - \frac{\sum X \sum Y}{N}}{\sqrt{\left(\sum X^2 - \frac{(\sum X)^2}{N}\right)\left(\sum Y^2 - \frac{(\sum Y)^2}{N}\right)}}$$

方程式 B-2：皮尔逊相关系数

上述公式可以用下列代码加以实现：

```
def pearson(x,y):
  n=len(x)
  vals=range(n)

  # 简单求和
  sumx=sum([float(x[i]) for i in vals])
  sumy=sum([float(y[i]) for i in vals])

  # 求平方和
  sumxSq=sum([x[i]**2.0 for i in vals])
  sumySq=sum([y[i]**2.0 for i in vals])

  # 求乘积之和
  pSum=sum([x[i]*y[i] for i in vals])

  # 计算皮尔逊评价值
  num=pSum-(sumx*sumy/n)
  den=((sumxSq-pow(sumx,2)/n)*(sumySq-pow(sumy,2)/n))**.5
  if den==0: return 1

  r=num/den

  return r
```

在第 2 章中，我们曾经利用皮尔逊相关系数来计算人们在偏好方面的相似级别。

译注 1：负相关是指，一个变量的值越大，则另一个变量的值反而会越小。

加权平均
Weighted Mean

加权平均是这样一类求平均的运算：参与求平均运算的每一个观测变量都有一个对应的权重值。在本书中，我们曾经根据相似度评价值，利用加权平均来进行数值型预测。加权平均的计算公式如方程式 B-3 所示，其中的 $x_1 \cdots x_n$ 是观测变量，$w_1 \cdots w_n$ 是权重值。

$$\bar{x} = \frac{w_1 x_1 + w_2 x_2 + \ldots + w_n x_n}{w_1 + w_2 + \ldots + w_n}$$

方程式 B-3：加权平均

上述公式的一个简单实现如下所示，函数接受一个数值列表和一个权重列表作为参数：

```
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
```

在第 2 章中，我们曾经利用加权平均来预测人们对一部影片的喜好程度。这是通过以其他用户与当前用户在品味上的相似度为权重，对这些人的影片评分情况求加权平均而得到的。在第 8 章，我们还曾利用加权平均来进行价格方面的预测。

Tanimoto 系数
Tanimoto Coefficient

Tanimoto 系数是一种度量两个集合之间相似程度的方法。在本书中，我们曾经根据一组属性列表，利用 Tanimoto 系数来计算两个事项之间的相似度。如果我们有两个集合，分别是 A 和 B：

A = [shirt, shoes, pants, socks]
B = [shirt, skirt, shoes]

那么两个集合之间的交集（即重叠部分），我们称其为 C，等于[shirt,shoes]。Tanimoto 系数的计算公式如方程式 B-4 所示，其中 N_a 是集合 A 的元素个数，N_b 是集合 B 的元素个数，N_c 是集合 C 的元素个数。在本例中，Tanimoto 系数等于 2/(4+3-2) = 2/5 = 0.4。

$$T = \frac{N_c}{(N_a + N_b - N_c)}$$

方程式 B-4：Tanimoto 系数

下列函数非常简单，它接受两个列表作为输入参数，并求得 Tanimoto 系数：

```
def tanimoto(a,b):
  c=[v for v in a if v in b]
  return float(len(c))/(len(a)+len(b)-len(c))
```

在第 3 章中，我们曾经在处理聚类时利用 Tanimoto 系数来计算两位用户之间的相似程度。

条件概率

Conditional Probability

概率是一种衡量某事件发生几率的方法。通常记做 *Pr(A)=x*，其中的 A 代表事件。例如，我们可能会说今天有 20%的可能性会下雨，相应地，我们将其记为 *Pr(rain)* = 0.2。

如果我们发现目前已经是乌云密布了，那么也许就会推断，今天有雨的几率会比原来更高。这就是**条件概率**，即：假定我们在知道 B 的情况下，A 发生的几率。我们将其记作 *Pr(A|B)*，在本例中，即为：*Pr(rain|cloudy)*。

条件概率的计算公式等于，两事件同时发生的概率，除以其中作为先决条件的那个事件所发生的概率，如方程式 B-5 所示。

$$Pr(A|B) = \frac{Pr(A \cap B)}{Pr(B)}$$

方程式 B-5：条件概率

因此，如果早上是阴天而且后来又下雨的概率为 10%，而早上阴天的概率为 25%，那么 *Pr(rain|cloudy)=0.1/0.25=0.4*。

由于只是一个简单的除法运算，因而此处就不给出函数了。在第 6 章中，我们曾经将条件概率用于对文档的过滤处理。

基尼不纯度

Gini Impurity

基尼不纯度是一种度量集合有多纯的方法。假设我们有一个集合，比如[A, A, B, B, B, C]，那么基尼不纯度会告诉我们：如果从集合中选出一项，并随机猜测其标识（译注 2），则猜测错误的概率为多少。如果集合中的所有元素都是 A，我们便总会猜到是 A，而且从来都不会出错，那么该集合就是绝对纯的。

方程式 B-6 给出了基尼不纯度的计算公式。

$$I_G(i) = 1 - \sum_{j=1}^{m} f(i,j)^2 = \sum_{j \neq k} f(i,j) f(i,k)$$

方程式 B-6：基尼不纯度

译注 2：在本例中，即为 A、B 或 C。

下列函数接受一个事项列表作为参数，并求得基尼不纯度：

```
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
```

我们在第 7 章进行决策树建模时，曾利用基尼不纯度来判断：如何划分一个集合才能令其变得更纯。

熵

Entropy

熵是判断集合内部混乱程度的另一种方法。它出自信息理论，是用以度量一个集合中的无序情况的。 如果要给熵下一个不是很严格的定义，我们可以认为熵代表了，从集合中随机抽中某一元素的意外程度。如果集合中的所有元素都是 A，那么当我们抽中元素 A 的时候是绝对不会出现意外的，此时的熵即为 0。熵的计算公式如方程式 B-7 所示。

$$H(X) = \sum_{i=1}^{n} p(x_i)\log_2\left(\frac{1}{p(x_i)}\right) = -\sum_{i=1}^{n} p(x_i)\log_2 p(x_i)$$

方程式 B-7：熵

下列函数接受一个事项列表作为参数，并求得熵的值：

```
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
```

我们在第 7 章进行决策树建模时，曾利用熵来判断：如何划分一个集合才能减少无序的情况。

方差

Variance

方差是用来度量一组数值与其均值之间的差距的。它经常被用于统计学中，用以测量集合中各个数值之间的差。方差的计算方法是，先求出每个数值与均值之间的差，然后再对这些差值的平方和求平均，如方程式 B-8 所示。

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2$$

方程式 B-8：方差

下面是一个简单的函数实现：

```
def variance(vals):
  mean=float(sum(vals))/len(vals)
  s=sum([(v-mean)**2 for v in vals])
  return s/len(vals)
```

我们在第 7 章进行回归树建模时，曾利用方差来判断：如何对集合做划分才能使划分后的子集分布得更加紧密。

高斯函数

Gaussian Function

高斯函数是正态曲线的概率密度函数。由于它具有“从高位开始快速下降，但从不会降到 0”的特点，因此在本书中，我们将它用作加权 k-最近邻算法的权重函数。

如方程式 B-9 所示的，是一个带差值变量 σ 的高斯函数计算公式。

$$\frac{1}{\sigma\sqrt{2\pi}}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

方程式 B-9：高斯函数

上述公式可以直接被翻译成一个只有两行代码的函数：

```
import math
def gaussian(dist,sigma=10.0):
  exp=math.e**(-dist**2/(2*sigma**2))
  return (1/(sigma*(2*math.pi)**.5))*exp
```

我们在第 8 章构造数值型预测算法时，曾将高斯函数作为一个可行的权重函数。

点积

Dot-Products

点积是两个向量的一种相乘运算。假设我们已有两个向量，$a = (a_1,a_2,a_3,\cdots)$，$b = (b_1,b_2,b_3,\cdots)$，那么点积的定义就如方程式 B-10 所示。

$$\mathbf{a} \bullet \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \ldots + a_n b_n$$

方程式 B-10：利用向量中的各个元素来计算点积

我们可以很容易地利用下列函数来实现点积的计算：

```
def dotproduct(a,b):
  return sum([a[i]*b[i] for i in range(len(a))])
```

假设 θ 是两个向量间的夹角，那么点积的定义也可以如方程式 B-11 所示。

$$\mathbf{a} \bullet \mathbf{b} = |\mathbf{a}||\mathbf{b}|\cos\theta$$

方程式 B-11：利用夹角来计算点积

这意味着，我们可以利用点积来计算两个向量间的夹角：

```
from math import acos

# 计算一个向量的大小

def veclength(a):
  return sum([a[i] for i in range(len(a))])**.5

# 计算两个向量间的夹角

def angle(a,b):
  dp=dotproduct(a,b)
  la=veclength(a)
  lb=veclength(b)
  costheta=dp/(la*lb)
  return acos(costheta)
```

我们在第 9 章对事物进行分类时，曾利用点积来计算向量的夹角。

索引
Index

We'd like to hear your suggestions for improving our indexes. Send email to *index@oreilly.com*.

C

D

E

F

G

O

P

Q

R

S

T

O'REILLY®

集体智慧编程

“太棒了！对于初学这些算法的开发者而言，我想不出有比这本书更好的选择了，而对于像我这样学过AI的老朽而言，我也想不出还有什么更好的办法能够让自己重温这些知识的细节。”

——Dan Russell，资深技术经理，Google

“Toby的这本书非常成功地将机器学习算法这一复杂的议题拆分成了一个个既实用又易懂的例子，我们可以直接利用这些例子来分析当前网络上的社会化交互作用。假如我早两年读过这本书，就会省去许多宝贵的时间，也不至于走那么多的弯路了。”

——Tim Wolters，CTO，Collective Intellect

想了解蕴藏在搜索排名、商品推荐、社会化书签以及在线婚介应用背后的巨大威力吗？本书的内容引人入胜，它将会告诉我们如何构造Web 2.0应用，使其能够挖掘有大量用户参与的互联网应用所产生的海量数据。利用书中介绍的这些复杂算法，可以编写出智能程序、访问其他Web站点的数据集、从我们自己的应用程序中搜集用户数据，进而分析和理解这些数据。

本书将引领我们进入机器学习与计算统计的世界，并解释如何得出有关用户体验、市场营销、个人品味以及我们和他人每天搜集的用户行为方面的结论。书中对每一个算法都进行了详细的描述，并附以简洁的代码，这些代码可以直接用于我们的Web站点、博客、维基，或者其他特定的应用。

本书向读者介绍了：

- 令在线零售商向用户提供商品或媒体推荐的协作型过滤技术；
- 在一个大型数据集中检测相似项群组的聚类方法；
- 在针对某一问题的数以百万计的可能题解中进行搜索，并从中选出最优解的优化算法；
- 用于垃圾过滤技术的贝叶斯过滤器，如何根据单词类型及其他特征对文档进行分类；
- 用于对在线约会站点的用户进行配对的支持向量机；
- 用于问题求解的智能进化技术——随着玩游戏的次数逐渐增多，计算机玩家如何通过改进自身代码的方式来发展技能。

本书的每一章后都有练习，这些练习对算法进行了扩展，使其变得更加强大。让我们超越以数据库为后端的简单应用系统，挖掘互联网数据的价值，为我所用！

图书分类：Web开发

责任编辑：王继花

项目管理：梁晶

www.oreilly.com

Broadview® WWW.BROADVIEW.COM.CN

网上订购：www.dearbook.com.cn

第二书店 · 第一服务

www.phei.com.cn

ISBN 978-7-121-07539-1

9 787121 075391 >

定价：59.80元