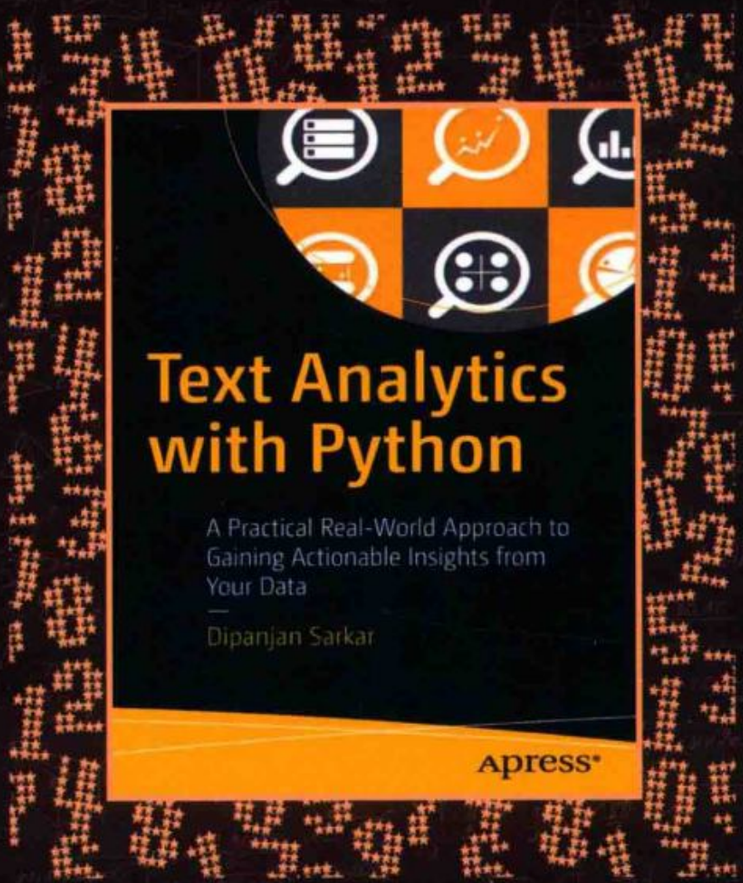
HZ BOOKS 华章IT
Apress®
数据科学与工程技术丛书

Python文本分析

[印度] 迪潘简·撒卡尔（Dipanjan Sarkar）著
闫龙川 高德荃 李君婷 译



TEXT ANALYTICS WITH PYTHON

A PRACTICAL REAL-WORLD APPROACH TO GAINING ACTIONABLE INSIGHTS FROM YOUR DATA

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

Python 文本分析 /（印）迪潘简·撒卡尔（Dipanjan Sarkar）著；闫龙川，高德荃，李君婷译 . —北京：机械工业出版社，2018.3
（数据科学与工程技术丛书）
书名原文：Text Analytics with Python: A Practical Real-World Approach to Gaining Actionable Insights from Your Data

ISBN 978-7-111-59324-9

I. P… II. ① 迪… ② 闫… ③ 高… ④ 李… III. 数据处理 IV. TP274

中国版本图书馆 CIP 数据核字（2018）第 043033 号

本书版权登记号：图字 01-2017-7335

Python 文本分析

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）
责任编辑：唐晓琳　　责任校对：李秋荣
印　　刷：中国电影出版社印刷厂　　版　　次：2018 年 4 月第 1 版第 1 次印刷
开　　本：185mm×260mm 1/16　　印　　张：17.75
书　　号：ISBN 978-7-111-59324-9　　定　　价：79.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379426 88361066　　投稿热线：（010）88379604
购书热线：（010）68326294 88379649 68995259　　读者信箱：hzit@hzbook.com

华章IT
HZBOOKS | Information Technology

译 者 序

自然语言处理和文本分析是当今人工智能研究和应用的重要方向，因其在人机交互方面的广泛应用和前景，吸引了学术界和产业界投入巨大的力量。目前，已经有一些产品陆续面世，在机器翻译、问答系统、语音助理、情感分析等方面取得了非常不错的进展，也给人们的生活带来了便利。

本书作者 Sarkar 是 Intel 公司的数据科学家，研究领域涉及数据科学与软件工程，有着丰富的文本分析研究和工程方面的经验，出版过多本 R 语言和机器学习方面的书籍。作者在 GitHub 上（https://github.com/dipanjanS/text-analytics-with-python）开源了本书相关的程序代码和数据集，感兴趣的读者可以下载研究。

本书首先介绍了与文本分析相关的自然语言基本概念以及 Python 语言的特点、特性和常用功能。然后，结合示例代码详细阐述了文本理解与处理、文本分类、文本摘要、文本相似性与聚类、语义与情感分析等内容，具有很强的实用性，内容覆盖了文本分析的重要方面，为相关应用的开发和研究提供了很好的参考借鉴。

本书是关于自然语言处理的实践教程，通过学习本书，读者可以全面地掌握文本分析的基础技术和机器学习的一些经典方法，包括 SVM、贝叶斯分类器、k-means 聚类、层次聚类等，为进一步的学习和研究奠定基础。感兴趣的读者可以继续研究和探索深度学习技术在文本分析中的应用，这是人工智能应用中发展非常迅速的领域，相信阅读本书打下的基础会对你大有帮助。

最后，感谢本书的作者和机械工业出版社华章公司的编辑，是他们的鼓励和支持使得本书能与读者见面。感谢我们家人的理解。尽管我们努力准确地表达作者的思想和方法，但仍难免有不当之处。译文中的错误，敬请指出，我们将非常感激，请将相关意见发往 yanlongchuan@iie.ac.cn。

闫龙川　高德荃　李君婷

2017 年 12 月

前　言

从高中开始接触数学和统计学以来，我就一直对数字着迷。分析学（analytics）、数据科学以及最近的文本分析技术均出现较晚，大概是在几年前，当时关于大数据（big data）和数据分析的炒作越来越猛烈，甚至有些疯狂。就个人而言，我认为其中很多都是过度炒作，但是也有一些令人兴奋的东西，因为这些技术在新工作、新发现以及解决人们先前认为不可能解决的问题方面展现了巨大的可能性。

自然语言处理（Natural Language Processing，NLP）一直深深吸引着我，因为人脑科学和人类认知能力确实令人着迷。如果尝试在机器中重塑这种传递信息、复杂思维和情绪的能力，那一定是令人惊喜的。当然，尽管我们在认知计算（cognitive computing）和人工智能（Artificial Intelligence，AI）方面的发展突飞猛进，但现在尚且无法实现这一点。仅通过图灵测试可能是不够的，机器真正能复制人的方方面面吗？

当今，对于NLP和文本分析应用，迫切需求从非结构化、原始文本数据中提取有用信息和可行见解的能力。到目前为止，我一直在努力解决各种问题，面临诸多挑战，并随着时间的推移吸取了各种各样的经验教训。本书涵盖了我在文本分析领域学到的大部分知识，仅仅从一堆文本文档中建立一个花哨的词云是不够的。

在学习文本分析方面，最大的问题或许不是信息缺乏，而是信息过多，通常这称为信息过载（information overload）。海量的资源、文档、论文、书籍和期刊包含了大量的理论资料、概念、技术和算法，它们常常使该领域的新手不知所措。解决问题的正确技术是什么？文本摘要如何真正有效？哪些才是解决多类文本分类的最佳框架？通过将数学和理论概念与现实用例的Python实现相结合，本书尝试解决这个问题，并帮助读者避免迄今为止我所遇到的一些急迫问题。

本书采用了全面的和结构化的介绍方法。首先，它在前几章中介绍了自然语言理解和Python结构的基础知识。熟悉了基础知识之后，其余章节将解决文本分析中的一些有趣问题，包括文本分类、聚类、相似性分析、文本摘要和主题模型。本书还将分析文本的结构、语义、情感和观点。对于每个主题，将介绍基本概念，并使用一些现实世界中的场景和数据来实现涵盖每个概念的技术。本书的构想是呈现一幅文本分析和NLP的蓝海，并提供必要的工具、技术和知识以处理和解决工作中遇到的问题。我希望你能觉得本书很有帮助，并祝你在文本分析的世界中旅途愉快！

目　录

译者序

前言

第 1 章　自然语言基础 …… 1

1.1　自然语言 …… 2

1.1.1　什么是自然语言 …… 2

1.1.2　语言哲学 …… 2

1.1.3　语言习得和用法 …… 4

1.2　语言学 …… 6

1.3　语言句法和结构 …… 7

1.3.1　词 …… 8

1.3.2　短语 …… 9

1.3.3　从句 …… 10

1.3.4　语法 …… 11

1.3.5　语序类型学 …… 17

1.4　语言语义 …… 17

1.4.1　词汇语义关系 …… 18

1.4.2　语义网络和模型 …… 20

1.4.3　语义表示 …… 21

1.5　文本语料库 …… 27

1.5.1　文本语料库标注及使用 …… 27

1.5.2　热门的语料库 …… 28

1.5.3　访问文本语料库 …… 29

1.6　自然语言处理 …… 33

1.6.1　机器翻译 …… 33

1.6.2　语音识别系统 …… 34

1.6.3　问答系统 …… 34

1.6.4　语境识别与消解 …… 34

1.6.5　文本摘要 …… 35

1.6.6　文本分类 …… 35

1.7　文本分析 …… 35

1.8　小结 …… 36

第 2 章　Python 语言回顾 …… 37

2.1　了解 Python …… 37

2.1.1　Python 之禅 …… 39

2.1.2　应用：何时使用 Python …… 40

2.1.3　缺点：何时不用 Python …… 41

2.1.4　Python 实现和版本 …… 42

2.2　安装和设置 …… 43

2.2.1　用哪个 Python 版本 …… 43

2.2.2　用哪个操作系统 …… 44

2.2.3　集成开发环境 …… 44

2.2.4　环境设置 …… 44

2.2.5　虚拟环境 …… 46

2.3　Python 句法和结构 …… 48

2.4　数据结构和类型 …… 50

2.4.1　数值类型 …… 51

2.4.2　字符串 …… 52

2.4.3　列表 …… 53

2.4.4　集合 …… 54

2.4.5 字典 …… 55
2.4.6 元组 …… 56
2.4.7 文件 …… 56
2.4.8 杂项 …… 57
2.5 控制代码流 …… 57
2.5.1 条件结构 …… 57
2.5.2 循环结构 …… 58
2.5.3 处理异常 …… 60
2.6 函数编程 …… 61
2.6.1 函数 …… 61
2.6.2 递归函数 …… 62
2.6.3 匿名函数 …… 63
2.6.4 迭代器 …… 63
2.6.5 分析器 …… 64
2.6.6 生成器 …… 66
2.6.7 itertools 和 functools 模块 …… 67
2.7 类 …… 67
2.8 使用文本 …… 69
2.8.1 字符串文字 …… 69
2.8.2 字符串操作和方法 …… 70
2.9 文本分析框架 …… 77
2.10 小结 …… 77

第 3 章 处理和理解文本 …… 79
3.1 文本切分 …… 80
3.1.1 句子切分 …… 80
3.1.2 词语切分 …… 83
3.2 文本规范化 …… 85
3.2.1 文本清洗 …… 85
3.2.2 文本切分 …… 86
3.2.3 删除特殊字符 …… 86
3.2.4 扩展缩写词 …… 87
3.2.5 大小写转换 …… 88
3.2.6 删除停用词 …… 89
3.2.7 词语校正 …… 89
3.2.8 词干提取 …… 95
3.2.9 词形还原 …… 97
3.3 理解文本句法和结构 …… 98
3.3.1 安装必要的依赖项 …… 99
3.3.2 机器学习重要概念 … 100
3.3.3 词性标注 …… 100
3.3.4 浅层分析 …… 106
3.3.5 基于依存关系的分析 …… 114
3.3.6 基于成分结构的分析 …… 118
3.4 小结 …… 123

第 4 章 文本分类 …… 124
4.1 什么是文本分类 …… 125
4.2 自动文本分类 …… 126
4.3 文本分类的蓝图 …… 128
4.4 文本规范化处理 …… 129
4.5 特征提取 …… 132
4.5.1 词袋模型 …… 133
4.5.2 TF-IDF 模型 …… 134
4.5.3 高级词向量模型 …… 139
4.6 分类算法 …… 143
4.6.1 多项式朴素贝叶斯 … 144
4.6.2 支持向量机 …… 145
4.7 评估分类模型 …… 147
4.8 建立一个多类分类系统 …… 150
4.9 应用 …… 158
4.10 小结 …… 158

第 5 章 文本摘要 …… 159
5.1 文本摘要和信息提取 …… 160
5.2 重要概念 …… 161
5.2.1 文档 …… 161
5.2.2 文本规范化 …… 161
5.2.3 特征提取 …… 161
5.2.4 特征矩阵 …… 161

5.2.5 奇异值分解 …… 162
5.3 文本规范化 …… 163
5.4 特征提取 …… 164
5.5 关键短语提取 …… 165
5.5.1 搭配 …… 165
5.5.2 基于权重标签的短语提取 …… 168
5.6 主题建模 …… 171
5.6.1 隐含语义索引 …… 172
5.6.2 隐含 Dirichlet 分布 …… 176
5.6.3 非负矩阵分解 …… 179
5.6.4 从产品评论中提取主题 …… 180
5.7 自动文档摘要 …… 183
5.7.1 隐含语义分析 …… 185
5.7.2 TextRank 算法 …… 187
5.7.3 生成产品说明摘要 … 190
5.8 小结 …… 191

第6章 文本相似度和聚类 …… 193
6.1 重要概念 …… 194
6.1.1 信息检索 …… 194
6.1.2 特征工程 …… 194
6.1.3 相似度测量 …… 194
6.1.4 无监督的机器学习算法 …… 195
6.2 文本规范化 …… 195
6.3 特征提取 …… 196
6.4 文本相似度 …… 197
6.5 词项相似度分析 …… 198
6.5.1 汉明距离 …… 200
6.5.2 曼哈顿距离 …… 201
6.5.3 欧几里得距离 …… 202
6.5.4 莱文斯坦编辑距离 … 202
6.5.5 余弦距离和相似度 … 206
6.6 文档相似度分析 …… 207
6.6.1 余弦相似度 …… 209
6.6.2 海灵格 - 巴塔恰亚距离 …… 210
6.6.3 Okapi BM25 排名 …… 212
6.7 文档聚类 …… 215
6.8 最佳影片聚类分析 …… 217
6.8.1 k-means 聚类 …… 219
6.8.2 近邻传播聚类 …… 224
6.8.3 沃德凝聚层次聚类 … 227
6.9 小结 …… 230

第7章 语义与情感分析 …… 232
7.1 语义分析 …… 233
7.2 探索 WordNet …… 233
7.2.1 理解同义词集 …… 234
7.2.2 分析词汇的语义关系 … 235
7.3 词义消歧 …… 240
7.4 命名实体识别 …… 241
7.5 分析语义表征 …… 244
7.5.1 命题逻辑 …… 244
7.5.2 一阶逻辑 …… 245
7.6 情感分析 …… 249
7.7 IMDb 电影评论的情感分析 … 249
7.7.1 安装依赖程序包 …… 250
7.7.2 准备数据集 …… 252
7.7.3 有监督的机器学习技术 …… 253
7.7.4 无监督的词典技术 … 256
7.7.5 模型性能比较 …… 271
7.8 小结 …… 272

第1章
自然语言基础

我们已迎来了大数据时代，组织和企业越来越难以管理由各种系统、过程和事务生成的所有数据。虽然，大数据的“3 V（Volume，Variety，Velocity）”（高容量，多样性，高速度）特征被广泛认可，但是其定义却比较模糊，导致了大数据（Big Data）这个术语常常被误用。这是因为人们有时很难准确地量化什么数据属于“大”数据。一些人可能把数据库中的10亿条记录看作大数据，但是与各种传感器甚至社交媒体生成的PB级数据相比，实际上该数据就显得量级相对较小。在所有组织，无论其从属何种行业领域，现今都有体量巨大的非结构化文本数据。仅举一些例子，我们有巨量的各种形式的数据，如推特、状态更新、评论、井号标签、文章、博客、维基信息和其他更多的社会媒体信息。即使在零售业和电子商务商店，也会基于新产品信息、客户评价和反馈元数据生成大量的文本数据。

与文本数据相关的挑战主要有两个方面。第一方面的挑战涉及数据的有效存储和数据管理。通常，文本数据是非结构化的，与关系数据库不同，其不遵循任何特定的预定义数据模型或模式。然而，基于数据语义，可以将数据要么存储在基于SQL的数据库管理系统（DBMS）中，如SQL Server，要么存储在基于NoSQL的系统中，如MongoDB。拥有海量文本数据集的组织通常使用基于文件的系统，例如Hadoop系统，其中所有数据以Hadoop分布式文件系统（Hadoop Distributed File System，HDFS）格式存储，并按需进行访问，这是数据湖（data lake）的主要原则之一。

第二方面的挑战是数据分析，以及尝试提取对于组织有价值、有意义的模式和有用的洞见。虽然有大量能任意使用的机器学习和数据分析技术，但其中大多数技术适用于数值型数据，所以必须借助于自然语言处理（Natural Language Processing，NLP）领域和专门的技术、变换和算法来分析文本数据，或者更具体地称为自然语言，它与机器容易理解的编程语言有着显著不同。请记住，高度非结构化的文本数据并不遵循结构化或规范化的语法和模式，因此不能直接使用数学模型或统计模型来分析它。

在深入文本数据分析的具体技术和算法之前，本章将讨论与文本数据特性相关的一些主要概念和理论基础。本章的主要目的是让你熟悉与自然语言理解、处理和文本分析相关的概念和领域。在本书中将使用Python编程语言，其主要用于访问和分析文本数据。本章中的示例将非常简单，而且易于理解。不过，如果想在阅读本章之前了解Python，你也可以快速浏览第2章的内容。本书中所有的例子可在原书作者的GitHub库（https://github.com/dipanjanS/text-analytics-withpython）中下载，其中包括程序、代码片段和数据集。本章介绍与自然语言、语言学、文本数据格式、句法、语义和语法相关的概念，然后再介绍高级的主

题，如文本语料库（text corpora）、NLP 和文本分析。

1.1 自然语言

虽然文本数据是非结构化数据，但它通常属于特定语言，遵循特定的语法和语义。任何文本数据片段（简单的单词、句子或文档）大多数情况下可追溯到一些自然语言。本节将讨论自然语言的定义、语言哲学、语言采集和语言的使用。

1.1.1 什么是自然语言

要理解文本分析和自然语言处理，我们需要了解到底是什么造就了语言的“自然性”。简单来说，自然语言是人类基于自然使用和交流而发展演化而来的语言，而不是像计算机编程语言那样由人工构造和创建的语言。

如英语、日语和梵语等人类语言都是自然语言。自然语言可以以不同的形式进行沟通和传递，包括言语、文字甚至符号。已经有大量的学术研究工作致力于理解语言的起源、本质和哲学。这将在下一节中简要讨论。

1.1.2 语言哲学

我们现在知道了自然语言是什么意思。但想想以下的问题：一门语言的起源是什么？

英国人的语言是因何形成“英语（English）”的？“fruit”这个词的意义是如何产生的？人们之间如何使用语言进行交流沟通？无疑，这些都是一些非常重大的哲学问题。

语言哲学主要解决以下四个问题，并探寻答案来解决它们：

- 语言中意义的本质。
- 语言用法。
- 语言认知。
- 语言与现实之间的关系。
- 语言中意义的本质涉及语言的语义和语意本身的特性。这里，语言哲学家或语言学试图找出语言对实际事物对象的意义——也就是说，任何词或句子的意义如何起源和产生，以及语言中不同的词语如何成为彼此的同义词并形成联系。另一个重要的研究内容是语言的结构和句法如何为语义奠定好基础，或者更具体地说，如何结构化地组织具有自身意义的词语以形成有意义的句子。语言学是对语言的科学研究，它是一个处理这些问题的专业领域，后面将对此进行更详细地讨论。句法、语义、语法和分析树是解决这些问题的一些方法。意义的语言学本质可以在两个人之间交流沟通时得以展现，这里的两个人分别记为发送者和接收者。对于发送者，意义是在向接收者发送消息时尝试表达或交流的内容，而对于接收者，意义是从接收消息的上下文中所理解或推断的内容。另外，从非语言的角度来看，诸如身体语言、既有经验和心理作用等都是语意的影响因素，考虑到这些因素，每个人都以自己的方式感知或推断出语言的意义。
- 语言用法更关心语言在各种场景和人类之间的交流中是如何使用的。这包括言语分析和说话时的语言用法，包括说话者的意图、语气、内容和表达消息时涉及的相关

动作。在语言学中这通常称为言语行为。语言创作的起源和人类认知活动——例如负责学习和使用语言的语言习得等更高级的概念也受到了重点关注。

- 语言认知侧重于研究人脑的认知功能如何实现理解和解释语言。考虑典型的发送者和接收者例子，涉及从消息沟通到解释的许多行为。语言认知试图发现在连接和关联特定词语成为句子以及构成一个有意义的消息时思维如何作用，当发送者和接收者使用语言来沟通消息时，语言与他们思维过程的关系是什么。
- 语言与现实之间的关系探讨语言表达的真实程度。通常，语言哲学家试图度量这些表达的事实符合度，以及它们如何与我们现实世界中的特定事件相关。这种关系可以用几种方式表示，我们将探讨其中的一些方式。

语义三角形模型是最流行的语义模型之一，其用于解释词语如何表达接收者心中的意义和想法，以及解释该意义如何与现实世界中的实体对象或事实联系起来。语义三角形模型由查尔斯·奥格登（Charles Ogden）和艾佛·理查德（Ivor Richards）在他们的著作《意义之意义》中提出，该书于1923年第一次出版，模型如图1-1所示。

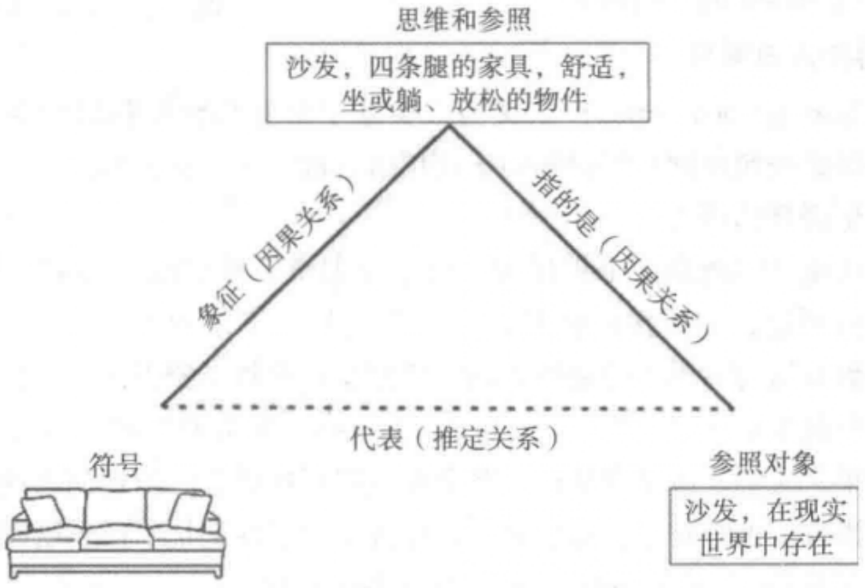


图1-1 语义三角形模型

语义三角形模型也称为意义之意义模型，图1-1中描述了一个真实例子，一个沙发置于某人面前，则可以被其感知。符号（symbol）表示语言符号，就像能唤起人脑中思想的词或物体。在这个例子中，符号是沙发，这唤起了什么是沙发的思想，一件家具，可以用于坐着或躺下和放松，它是让我们感到舒适的物体。这些思想称为参照，通过这个参照，人们能够将它与存在于现实世界中的称为参照对象的物体相关联。在本例中，参照对象是现实世界中的沙发。

找到语言和现实之间关系的第二种方式称为方向匹配（direction of fit），我们将在这里讨论两个主要方向。“词－世界”（word-to-world）的匹配方向谈论了语言使用能够反映现实的案例情况，即，使用词语来匹配或关联现实世界中正在发生的或已经发生的事情。举一个例子，句子“The Eiffel Tower is really big”，它强调了现实世界中的一个事实。另一个匹配方向，称为“世界－词”（world-to-word），它谈论语言使用可以改变现实的情况。举一个例子，句子“I am going to take a swim”，在这里人物“I”通过将要游泳正在改变现实，这通过正在交流的语句表达了相同的意义。图1-2显示了两种匹配方向之间的关系。

根据前面的描述，可以很清楚地看到，基于从现实世界感知的参照对象，一个人可以形成符号或词形式的表达，并且可以将其完全相同地传递给另一个人，这个人则基于所接收的符号建立对真实世界的画像表示，这样就构成一个循环。

1.1.3 语言习得和用法

到目前为止，我们已经掌握了自然语言所表示的意思和语言背后的概念、它的本质、意义和使用。本节将更深入地讨论如何使用人类的认知能力来感知、理解和学习语言，最后将讨论语言用法的主要形式，作为言语行为进行简要讨论。重要的是，不但要理解自然语言的含义，而且要理解人类如何解释、学习和使用相同的语言，以便我们尝试从文本数据中提取洞见时，能够在算法和技术中编程模拟其中的一些概念。

图 1-2 方向匹配表示图

1. 语言习得和认知学习

语言习得（language acquisition）定义为人类基于听觉和感知利用认知能力、知识和经验来理解语言并开始按照单词、短语和句子使用语言进行相互交流的过程。简单来说，语言习得是获取和产生语言的能力。

语言习得的历史可追溯到几个世纪前。哲学家和学者曾试图推理和理解语言习得的起源，并提出了几种理论，例如把语言看作一种代代相传下来的天赋的能力。柏拉图指出，词 - 义之间的映射在语言习得中起关键作用。许多学者和哲学家提出了现代理论，在一些流行理论中，最为著名的是 B. S. 斯金纳（B. S. Skinner）指出的知识、学习和语言的使用更多是一种行为结果。人类，或更具体地，儿童在使用任何语言的特定词语或符号时，会体验基于某些刺激的语言，由于对它们反复使用，导致了它们在记忆中得到增强。这个理论基于操作性条件反射（operant conditioning）或工具性条件反射（instrumentation conditioning），这是一种条件学习，其中基于其后果进行特定行为或行动强度的调整，例如奖励或惩罚，并且这些随后的刺激有助于增强或控制行为和学习。举一个例子，孩子们会学到由反复使用某个单词组成的一个特定的声音组合，这可能是通过他们的父母，或者通过他们正确地说这个字时得到的奖赏回报，或者说错时而被纠正。未来，这种反复的条件反射最终将强化孩子记忆中对该词的实际意义和理解。总而言之，孩子们基本上通过行为模仿和听成年人讲话来学习和使用语言。

然而，这种行为理论受到著名语言学家诺姆·乔姆斯基（Noam Chomsky）的挑战，他认为，孩子们不可能只通过模仿成人的一切来学习语言。这个假设在下面的例子中是成立的。虽然像 go 和 give 这样的词是有效的，但是孩子们常常最后会使用这个单词的无效形式，例如使用 goed 或 gived 来代替过去时态的 went 或 gave。可以确保他们的父母没有在孩子面前说出这些话，所以根据先前的 Skinner 理论，孩子们习得这些是不可能的。所以，乔姆斯基提出儿童不仅要模仿他们所听到的言语，还从相同的语言结构中提取模式、语法和规则，这与仅仅运用基于行为的通用认知能力是不同的。

按照乔姆斯基的观点，认知能力以及与特定语言相关的知识和能力，如句法、语义、言

语概念和语法，一起形成了他所称的语言习得机制，从而使人类能够具备语言习得的能力。除了认知能力之外，在语言学习中独特和重要的是语言本身的语法，这在他的名句“Colorless green ideas sleep furiously”中进行了强调。如果你观察这个句子并重复多次，会发现它是没有意义的。Colorless 与 green 矛盾，ideas 不能与 green 关联，它们也不能与 sleep furiously 搭配。然而，从语法上来讲，这个句子符合句法。这正是乔姆斯基试图解释的——句法和语法描述的信息独立于词的意义和语义。因此，他提出与其他认知能力相比，语言句法的学习和识别是人类的一种独特能力。这个假设也称为句法自主性（autonomy of syntax）。虽然这些理论仍然受到学者和语言学家的广泛争论，但是它对于探索人类心智如何习得和学习语言都非常有益。现在，我们将看看通常情况下语言使用的典型模式。

2. 语言用法

上一节讨论了言语行为以及如何使用方向匹配模型将词和符号与现实相关联。本节将介绍与言语行为相关的一些概念，重点讲述在交流中使用语言的不同方式。

言语行为主要分为三类：言内行为（locutionary）、言外行为（illocutionary）和言后行为（perlocutionary）。言内行为主要涉及当句子通过说话从一个人表达给另一个人时的实际传递行为。言外行为更关注于所表达句子的实际语义和意图。言后行为从心理或行为角度更注重言语表达对其接收者的实际影响。

举一个简单的例子，父亲对他孩子说的短语“Get me the book from the table”。这位父亲说出短语时则形成了言内行为。这句话的意图是一条指令，命令孩子替他去从桌上取书，它形成一种言外行为。孩子听到后所采取的行动，也就是说，如果他把书从桌子取给他的父亲，则形成了言后行为。

在上面这个例子中，言外行为是一个指令。根据哲学家约翰·塞尔（John Searle）的理论，总共有如下五种不同类型的言外行为。

- 断言（assertive）是说明事物已经存在于世界的言语行为。当发言者试图断言在现实世界中命题可能是真或假时，发言者说出的就是断言。这些断言可以是声明或陈述。举一个简单的例子，“The Earth revolves round the Sun”。这些信息表示前面讨论的“词 - 世界”的匹配方向。
- 指令（directive）是说话者向听话者表达请求或指挥他们做某事的言语行为。这表示听话者在接收到来自说话者的指令之后可能采取的自愿行动。既然指令是自愿性的，就可以遵从或不遵从它。这些指令可以是简单的请求，甚至命令或指令。“Get me the book from the table”是一个指令示例，我们先前谈论言语行为类型时讨论过。
- 承诺（commisive）是说话者或发言者按其所说，承诺一些未来自愿行为或行动的言语行为。诸如承诺、誓言、保证和宣誓等言语行为即为承诺，其匹配方向可以有两种。“I promise to be there tomorrow for the ceremony”是一个承诺类的例句。
- 表达（expressive）表明发言者或说话者对通过消息所传递的特定主张的意向和观点。它可以包含各种表达形式或情感类型，例如祝贺、讽刺等。“Congratulations on graduating top of the class”是一个表达类的例句。
- 宣告（declaration）是强大的言语行为，具有基于说话者/发送者表达消息中所宣称的主张来改变现实的能力。它的匹配方向通常是“世界 - 词”模式，但也可以是相反方向。“I hereby declare him to be guilty of all charges”是一个宣告类的例子。

这些言语行为是人类之间使用和传递语言的主要方式，我们往往并不会意识到，在一天中，我们可能会使用上述言语行为上百次。下面我们来看看语言学以及一些与它相关的主要研究领域。

1.2 语言学

我们已经介绍了是什么自然语言，如何学习和使用语言以及语言习得的起源。在语言学中，这些是语言学家等研究人员和学者们所研究的内容。严格来讲，*语言学*定义为对语言的科学研究，包括语言的形式和句法、意义和由语言用法和使用语境描述的语义。语言学的起源可追溯到公元前4世纪，当时的印度学者和语言学家帕尼尼（Panini）对梵语语言进行了形式化描述。1847年首先提出专业术语*语言学*（linguistics），用来表明对语言的科学研究，在此之前用于表达相同意思的专业术语是*文献学*（philology）。虽然文本分析不需要语言学的详细知识，但是了解语言学的不同领域还是十分有用的，因为其中一些领域在自然语言处理和文本分析算法中广泛使用。在语言学中，主要的专业研究领域如下。

- *语音学*（phonetics）：这是对在讲话过程中由人类声道产生的声音的声学性质的研究。它包括研究声音的特性，以及人类如何创造声音。在具体语言中，人类语音的最小个体单位称为*音位*（phoneme）。对于这个语音单位，一个更通用的跨语言术语是*音素*（phone）。
- *音韵学*（phonology）：这是对人类思维如何解释声音模式的研究，用于区分不同的音位，以找出哪些音位是重要的。通常，通过逐一考虑特定语言来详细研究音位的结构、组合和解释。英语由大约45个音位组成。音韵学通常不仅研究音位，还包括口音、语气和音节结构等内容。
- *句法*（syntax）：这通常是对句子、短语、词语及其结构的研究。它包括研究词语如何在语法上组合在一起以构成短语和句子。在短语或句子中，使用词语的句法顺序很重要，因为顺序会完全改变其含义。
- *语义*（semantics）：这涉及语言中意义的研究，并且可以进一步细分为词汇语义（lexical semantics）和成分语义（compositional semantics）。
  - *词汇语义*：使用形态和句法来研究词和符号的意义。
  - *成分语义*：研究词语和词语组合之间的关系，理解短语和句子的意义以及它们之间的相关性。
- *形态学*（morphology）：*语素*（morpheme）是具有区别性意义的最小语言单位。这包括诸如词语、前缀、后缀等具有各自不同含义的内容。形态学是对一门语言中这些不同单元或语素的结构和意义的研究。特定的规则和句法通常是控制语素组合在一起的方式。
- *词汇*（lexicon）：这是对语言中使用的单词和短语属性以及它们如何构建语言词汇的研究。这些包括何种声音与词语的含义相关联、词语所属的词性，以及它们的形态构成。
- *语用学*（pragmatics）：它研究语言和非语言因素（如上下文和场景）如何影响一条消息或一个话语所表达的意义。这包括设法推断在沟通交流中是否存在隐含或间接的意义。

- 话语分析（discourse analysis）：它通过人们的对话来分析语言和以句子形式的信息交换。这些对话可以通过口头、书面，甚至手势进行表达。
- 文体学（stylistics）：这是侧重于对语言写作风格的研究，包括语气、口音、对话、语法和语态类型。
- 符号学（semiotics）：这是对符号、标记和符号过程以及它们如何传达意义的研究。这个领域涵盖类比、隐喻和象征主义等内容。

上述内容是语言学研究探讨的主要领域，但语言学属于一个庞大的领域，其范围远远大于这里所提及的内容。然而，诸如语言句法和语义之类的内容是一些最重要的概念，这些概念基本上奠定了自然语言处理的基础。以下章节将更深入地介绍它们。

1.3 语言句法和结构

我们已经知道语言、句法和结构所表示的内涵。句法和结构通常是密切相关的，一组特定的规则、惯例和原则通常规定了单词组成短语、短语组成从句、从句组成句子的形式。本节专门讨论英语的句法和结构，因为在本书中我们将处理英文类型的文本数据。但是，这些概念也可以扩展到其他语言类型。关于语言结构和句法的知识对许多领域有所帮助，例如文本处理、文本标注，以及用于进一步的文本分析，如文本分类或摘要。

在英语中，单词常常组合在一起形成其他语言成分。这些成分要素包括单词、短语、从句和句子。所有这些成分在任何消息中一起存在，并且在层次结构中彼此相关。此外，句子是一种表达一组词汇的结构化格式，只要它们遵循某些句法规则，如语法。请看图 1-3 中所示的单词。

dog the over he
lazy jumping is the fox
and is quick brown

图 1-3 没有任何关系或结构的单词集合

在图 1-3 的单词集合中，很难确定它可能正在尝试传递什么或意味着什么。事实上，语言不仅仅是非结构化词汇的堆叠。正确句法不仅有助于我们获取正确的句子结构和相关词，还可以帮助句子根据单词的顺序或位置来表达意义。就我们先前的“句子→从句→短语→单词”的层次结构而论，我们能使用浅层分析（shallow parsing）构建图 1-4 中的分层句子树，浅层分析是一种用于查找句子成分的技术。

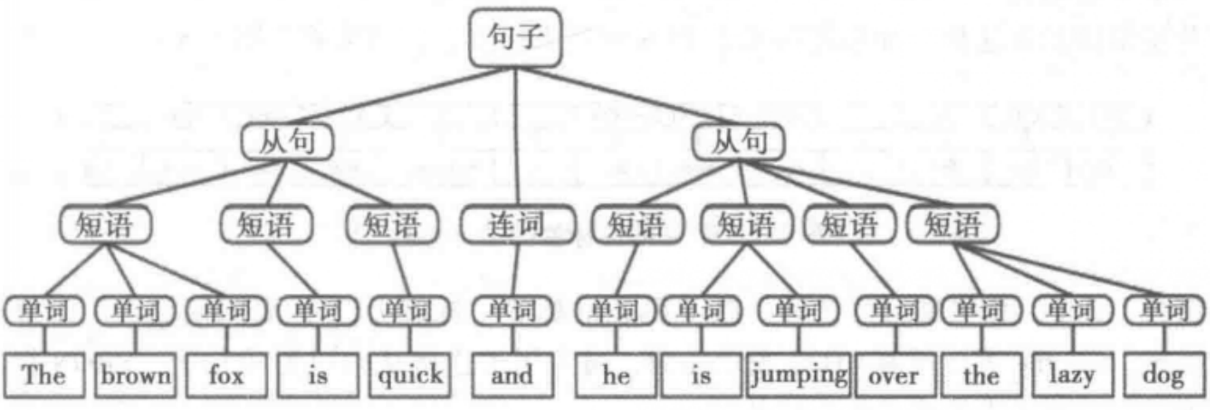


图 1-4 遵循分层句法的结构化语句

根据图1-4中的分层树，我们得到了句子“The brown fox is quick and he is jumping over the lazy dog”。可以看到，分层树的叶节点由单词组成，这些单词是树的最小单元，单词之间组合成短语，短语再依次形成从句。从句通过各种填充词或词语（如连词）连接在一起，形成最终的句子。下一节将进一步详细介绍这里提及的每种组成部分，了解如何分析它们，并找出主要的句法类别。

1.3.1 词

词（word）是一门独立语言中的最小单位，具有其特有的含义。虽然语素是有独特意义的最小语言单位，但语素与词不同，它不是独立的，而一个单词可以由几个语素组成。标注和标记单词并分析其词性（Parts Of Speech，POS）对于查看主要句法类别是有用的。这里将介绍各种POS标签的主要类别和意义。第3章将进一步详细研究它们，并学习编程生成POS标签的方法。

通常，单词可以归为以下主要的词类之一。

- 名词（noun）：通常表示描述一些可能有生命或无生命的物体或对象的单词。例如，fox、dog、book等。名词的POS标签是N。
- 动词（verb）：动词是用来描述某些动作、状态或事件的词。动词有各种各样的子类组成，如助（auxiliary）动词、反身（reflexive）动词和及物（transitive）动词等。动词的一些典型例子如running、jumping、read和write。动词的POS标签是V。
- 形容词（adjective）：形容词是用于描述或限定其他词——通常是名词和名词短语的词。短语beautiful flower有一个名词（N）flower，用形容词（ADJ）beautiful来描述或限定。形容词的POS标签是ADJ。
- 副词（adverb）：副词通常用作其他单词的修饰词，包括名词、形容词、动词或其他副词。短语very beautiful flower有副词（ADV）very，它修饰形容词（ADJ）beautiful，表明flower的beautiful程度。副词的POS标签是ADV。

除了上面这四个主要类别的词类外，还有其他词类在英语中经常出现。这些词类包括代词（pronoun）、介词（preposition）、感叹词（interjection）、连词（conjunction）和限定词（determiner）等。而且每个POS标签均可进一步细分，如名词（N）可以进一步细分为单数名词（NN）、单数专有名词（NNP）和复数名词（NNS）。第3章将更详细地介绍POS标签，到时我们将处理和解析文本数据并实现使用POS标签来标注文本。

考虑先前的例句（The brown fox is quick and he is jumping over the lazy dog），我们已经构建了分层结构的句法树，如果使用基本POS标签来标注它，结果将如图1-5所示。

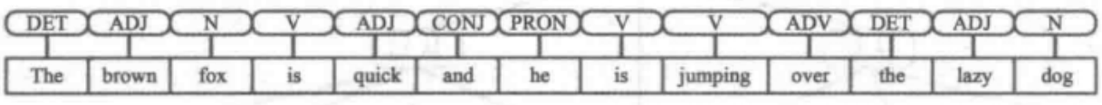


图1-5 带有词性标签的单词标注

在图1-5中，你可能会注意到一些不熟悉的标签。标签DET代表限定词，用于描述数量，如a、an、the等。标签CONJ表示连词，通常用于连接从句以形成句子。PRON标签代表代词，表示用于表达或替代名词的词。

标签N、V、ADJ和ADV是典型的开放性词类，代表属于开放性词汇表的词。开放性词

类（open class）是由一个无限的词集组成的词类，它通常接受向词汇表中添加人们创造的新词。词通常通过诸如形态派生（morphological derivation）、基于使用的发明和创建复合词（compound lexeme）等过程添加到开放性词类中。最近添加的一些流行名词包括 Internet（因特网）和 multimedia（多媒体）。封闭性词类（closed class）由一个封闭的和有限的词集组成，不接受添加新词。代词就属于封闭性的词类。

下一节将介绍分层结构中的下一个级别内容：短语。

1.3.2 短语

正如先前介绍的，词具有特有的词汇属性。使用这些词，可以按照词的意义来排序它们，使得每个词属于相应的短语类别，其中一个词是主词或中心词。在分层树中，词的组合构成短语，形成了句法树的第三级。原则上，按照“词 ← 短语 ← 从句 ← 句子”这样的级别排序，假定短语至少包含两个或多个单词。然而，在从句或句子中短语可以是基于句法和位置的单个单词或词组合。例如，句子“Dessert was good”只有三个单词，每个单词加起来就相当于三个短语。词“Dessert”是名词，也是名词短语，“is”充当动词以及动词短语，“good”代表形容词以及形容词短语，用来描述上述的“Dessert”。

短语有以下五个主要类别。

- 名词短语（Noun Phrase，NP）：这些短语以名词作为中心词。名词短语是作为动词的主语或对象。通常，名词短语是一组词，可以用代词代替而不会造成语句或从句在句法上不正确。例如“dessert”“the lazy dog”“the brown fox”。
- 动词短语（Verb Phrase，VP）：这些短语是以动词作为中心词的词汇单元。通常有两种形式的动词短语。一种形式包括动词要素以及其他词对象，如名词、形容词或副词，作为对象的一部分。这里的动词称为限定动词（finite verb）。它作为层次结构树中的独立单元，可以作为从句中的根。这种形式在结构语法（constituency grammar）中很重要。另一种形式中，以限定动词作为整个从句的根，这在依存语法（dependency grammar）中是重要的。这种形式的另一类派生词包括严格由动词组成的动词短语，包括主动词、辅动词、不定式和分词。句子“He has started the engine”可以用来说明形成两种类型的动词短语。短语“has started the engine”和“has started”就是基于刚才讨论的两种形式。
- 形容词短语（ADJective Phrase，ADJP）：这些短语是用形容词作为中心词的短语。它们的主要作用是在句子中描述或限定名词和代词，它们可以放在名词或代词之前或之后。“The cat is too quick”这句话有一个形容词短语“too quick”，用来修饰名词短语“cat”。
- 副词短语（ADVerb Phrase，ADVP）：这些短语类似副词，因为副词是作为短语中的中心词。副词短语用作名词、动词或副词本身的修饰词，以更多细节来描述或限定它们。在句子“The train should be at the station pretty soon”中，形容词短语“pretty soon”描述了火车何时到达。
- 介词短语（Prepositional Phrase，PP）：这些短语通常包含介词作为中心词和其他的词语成分，如名词、代词等。它的作用像形容词或副词，用来描述其他词或短语。“goingup the stairs”这个短语包含一个介词短语“up”，描述了“the stairs”的方向。

短语的五大主要句法类别可以从使用几个规则的词中产生，其中一些规则已讨论过，如利用不同类型的句法和语法。后面的章节将探讨一些流行的语法。*浅层句法分析*是一种流行的自然语言处理技术，用于提取这些组成成分，包括 POS 标签以及句子中的短语。对于句子“The brown fox is quick and he is jumping over the lazy dog”，根据浅层句法分析，获得了七个短语，如图 1-6 所示。

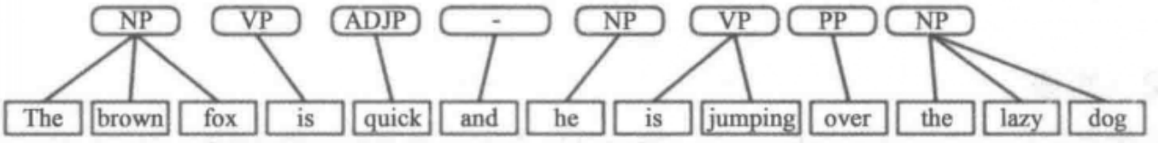


图 1-6 有标签的标注短语

短语标签属于前面讨论的类别，而“and”这个词是一个连词，通常用于将从句连接在一起。下一节将会讨论从句、它们的主要类别，以及用于从句子中提取从句的一些惯例和句法规则。

1.3.3 从句

本质上，从句可以作为独立的句子，或者几个从句可以组合在一起形成一个句子。*从句*由一组相互之间有某种关系的词所组成，它通常包含一个主语和一个谓语。有时主语并不存在，谓词通常具有动词短语或具有对象的动词。默认情况下，可以将从句分为两个不同的类别：*主从句*（main clause）和*从属从句*（subordinate clause）。主从句也称为*独立从句*，因为它可以本身形成一个句子，并可作为句子和从句。从属从句或*依存从句*本身不能独立存在，其取决于主从句的意义。它们往往通过使用从属词（如从属连词）与其他从句相连。

就语言的句法特性而言，从句可以根据句法细分为以下几个类别。

- *陈述类*（declarative）：这些从句通常很频繁地出现，表示没有与其相关联的特定语气的陈述。这些只是标准的使用中性语调的陈述，可能是事实或非事实的。例如“Grass is green”。
- *祈使类*（imperative）：这些从句通常以请求、命令、规定或建议的形式出现。这种情况下，语气可能是一个人向一个或多个人发出命令，以执行命令、请求或指令。例如“Please do not talk in class”。
- *关系类*（relative）：对*关系*从句的最简单解释是它们为从属从句，因此依赖于通常包含词、短语或从句句子的其他部分。该元素通常充当关系从句中一个词的先行词，并与之相关。举一个简单的例子，“John just mentioned that he wanted a soda”句子中的先行词是专有名词“John”，其在关系从句“he wanted a soda”中提及。
- *疑问类*（interrogative）：这些从句通常是以问题的形式出现。这些问题的类型可以是肯定的或者否定的。例如“Did you get my mail?”和“Didn’t you go to school?”。
- *感叹类*（exclamative）：这些从句用来表达惊讶、惊喜，甚至赞美。这些表达属于感叹语类，这类从句往往以感叹号结尾。例如“What an amazing race!”。

通常，大多数从句都以前面提到的句法形式之一表示，不过这个从句类别列表并不是详尽的列表，它可以进一步分为几种其他形式。对于例句“The brown fox is quick and he is jumping over the lazy dog”，如果你记得句法树，这里的并列连词“and”将句子分成两个从句：“The brown fox is quick”和“he is jumping over the lazy dog”。你能猜出它们可能属于什

么类别吗？（提示：请回顾陈述和关系从句的定义。）

1.3.4 语法

语法（grammar）有助于构建语言的句法和结构。语法主要包括一组规则，用来在为任何自然语言构造句子时确定如何定位词、短语和从句。语法不局限于书面文字——它在口头语言上也有同样作用。语法规则可以具体到一个地区、语言、方言或是比较普遍的主谓宾语（Subject-Verb-Object，SVO）模型。语法的起源有着悠久的历史，它从印度的梵文开始。在西方，语法研究起源于希腊语，最早的作品是由狄奥尼修斯·泰勒（Dionysius Thrax）撰写的《语法艺术》（Art of Grammar）。拉丁语语法模型是从希腊语模型开发而来，逐步跨越几个时代，创建了各种语言的语法。直到十八世纪，语法才被认为是语言学领域的重要分支。

语法在时间历程中发展演化，导致更新类型的语法诞生，而各种旧的语法慢慢失去了主导地位。因此，语法不仅仅是一套固定的规则，而且是基于人类在时间长河中使用语言的演变。在英语中，有几种方法可以对语法进行分类。我们将首先讨论两大类，最流行的语法框架可以据此分组。然后，我们将进一步探讨这些语法如何表达语言。

基于语言句法和结构的表示法，语法可以细分为两个主要类型：*依存语法*（dependency grammars）和*成分语法*（constituency grammars）。

1. 依存语法

依存语法并不关注词、短语和从句等组成部分，而是更加强调词。依存语法也称为*基于词的语法*。要理解依存语法，我们应该首先知道在这种情况下*依存性*是什么意思。在这种情况下的依存关系标记为通常非对称的词－词关联或连接。基于句子中词的定位，一个词与另一个词有关联或依赖于另一个词。因此，依存语法假定短语和从句更深一层的成分来自于词与词之间的依存关系结构。

依存语法背后的基本原理是，在某种语言的任意一个句子中，除了一个词以外，所有的词都与句子中的其他词有某种关系或依存关系。没有依存性的词称为句子的*根*（root）。在大多数情况下，将动词视为句子的根。所有其他的词都通过*连接*直接或间接地与根动词联系在一起，这就是依存性关系。虽然没有短语或从句的概念，但是了解词与其从属词的句法和关系，可以确定句子中必要的组成部分。

依存语法对句子中的每个词都有一一对应关系。这种语法表示有两个方面。一个是句子的句法或结构，另一个是从词之间表示的关系所获得的语义。词及其互连的句法或结构可以使用类似于前面章节所描述的句子语法或分析树来显示。考虑句子“The brown fox is quick and he is jumping over the lazy dog”，如果我们想为该句绘制依存关系的句法树，将得到如图 1-7所示的结构图。

图 1-7 显示了由依存关系构成的一棵树——或更准确地说是一个图，涵盖了句子中的所有词。该图连接的每个词至少有一条有向边以此作为连出或连入。该图也是有向的，因为两个词之间的每条边指向一个特定方向。本质上，依存关系树是有向非循环图（Directed Acyclic Graph，DAG）。除了根节点，树中的每个节点最多有一条入边。由于这是一个有向图，依存关系树自然不会描述句子中词的顺序，而是强调句子中词之间的关系。句子使用前面讨论的相关 POS 标签来标注，有向边则显示依存关系。现在，如果你还记得，我们刚刚讨论过，使用依存语法来表示句子有两个方面。每条有向边代表特定类型的有意义关系（也称

为句法功能）。我们能更进一步标注我们的句子，从而显示词之间具体的依存关系类型。

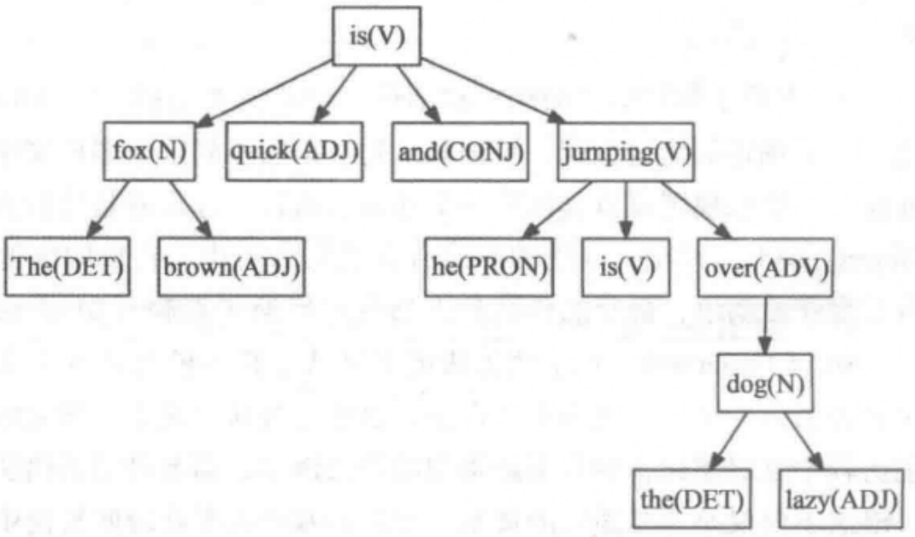


图 1-7 具有 POS 标签的基于句法树的依存语法

相同情形如图 1-8 所示。这里要记住的重要一点是，根据你使用的分析器，可能会存在该图的不同变体，因为它取决于分析器最初的训练方式、用于训练它的数据类型以及使用的标签系统的种类。

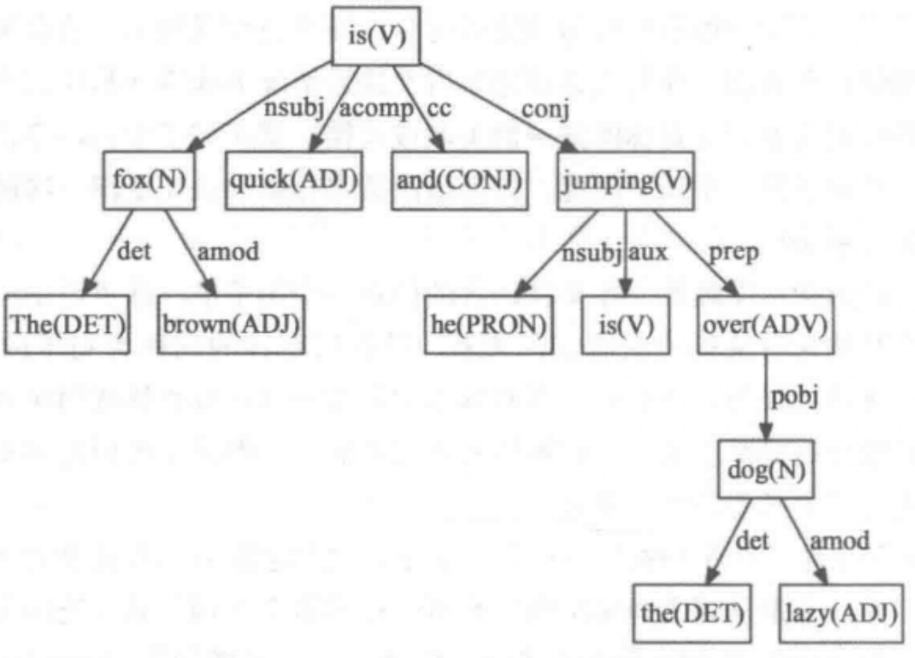


图 1-8 用依存关系类型注释的基于依存语法的句法树

每个依存关系都有自己的意义，并且是通用依存关系类型列表的一部分。这在原作论文（UniversalStanford Dependencies：A Cross-Linguistic Typology，de Marneffe et al，2014）中讨论。你可以在网址（http://universaldependencies.org/u/dep/index.html）查看依存关系类型及其含义的详尽列表。如果我们观察到这些依存关系，那就不难理解它们了。下面我们来看看在图 1-8 中句子依存关系所使用到的一些标签细节。

- 依存关系标签“det”非常直观——它表示一个名义上的中心词和限定词之间的限定关系。通常，带有 POS 标签 DET 的词也具有 det 的依存标签关系。例如（fox→the）和（dog→the）。

- 依存关系标签“amod”代表形容词修饰词（adjectival modifier），代表修饰名词意义的任何形容词。例如（fox→brown）和（dog→lazy）。
- 依存关系标签“nsubj”表示在从句中充当主语或替代词的实体。例如（is→fox）和（jumping→he）。
- 依存关系“cc”和“conj”更多的是与通过并列连词（coordinating conjunction）相关的词产生连接关系。例如（is→and）和（is→jumping）。
- 依存关系标签“aux”表示从句中的助动词或第二动词。例如（jumping→is）。
- 依存关系标签“acomp”代表语句中的形容词补语、作为补充的动作或动词的宾语。例如（is→quick）。
- 依存关系标签“prep”表示介词修饰符，经常用来修饰名词、动词、形容词或介词的含义。这种指代关系通常用于拥有名词或名词短语补语的介词。例如（jumping→over）。
- 依存关系标签“pobj”用于表示介词的宾语。在语句中，它通常是跟在介词后面的名词短语的前部。例如（over→dog）。

为了解释词语间不同的依存关系，在例句中，反复使用上述标签。现在既然你对于依存关系有了更好的理解，就可以懂得，当需要表示一个句子的依存关系时，通常并不会创作一个带有线性层级的树形结构，取而代之的是，你可以使用一个常规的图形来表示，因为在依存语法中，词汇并没有顺序的概念。图 1-9 同样阐述了依存语法。

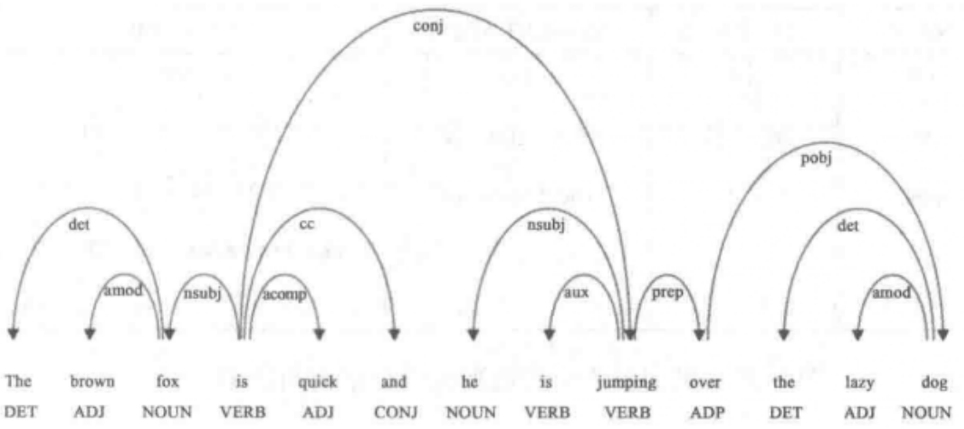


图 1-9 例句的依存关系语法示意图

图 1-9 由 spacy. io 生成，它拥有一些强大的自然语言处理（NLP）模型，这些模型所在的库是开源的。（当我们在下一节介绍基于成分结构的语法时，将会观察到依存关系语法中的节点数相较于它的成分结构副本数是更少的。）目前，基于依存关系语法建立了多种多样的语法架构。许多流行的语法架构包含代数句法（Algebraic Syntax）和算符文法（Operator Grammar）。

2. 成分语法

成分语法是建立在以下原则上的一类语法，即一条语句可以被多个由语句派生出的成分代表。这类语法能够以层次分明、有序的成分结构模拟或表征语句的内部结构。在实际用例中，每个单词通常都属于一个特定的词汇范畴，并形成不同短语的头单词。这些短语基于短语结构规则（phrase structure rule）构成。因此，成分语法通常也称为短语结构语法（phrase structure grammar）。短语结构语法最初由乔姆斯基在 19 世纪 50 年代提出。想要理解成分语法，我们必须清楚地明白成分的含义。为了唤醒你的记忆，这里再次说明，成分是指

拥有特定含义并且能够组合在一起构成独立或非独立单元的词或词组。它们也能够进一步结合在一起构成语句中的高阶结构，包括短语和从句。

短语结构规则构成了成分结构句法的核心，因为它们涉及管理语句中各种成分层级和排序的句法及规则。这些规则主要规定两件事。第一件（也是最重要的）是它们决定哪些词可以用来构成短语或成分。第二件是这些规则决定如何组合这些成分。如果我们希望分析短语结构，就应该清楚短语结构规则的典型架构模式。短语结构规则一般的表示方式是 $S \rightarrow AB$，它代表结构 S 由成分 A 和 B 构成，并且顺序是先 A 后 B。

目前，已有几种短语结构规则，我们将逐一探索它们，以理解到底如何在语句中对成分进行提取和排序。其中，最重要的规则是界定如何划分语句或从句。该短语结构规则用 S→NP VP 表示一个句子或分句的二分法，其中 *S* 表示语句或分句，它分为由名词短语（NP）表示的主语和由动词短语（VP）表示的谓语。

我们可以应用更多的规则以进一步分解每个成分，但是层级的顶端通常始于一个 NP 或 VP。名词短语的表示方式是 NP→［DET］［ADJ］N［PP］，其中方括号表示该内容是可选的。通常一个名词短语由一个肯定是作为头单词的名词（N）构成，并可选择性地包含定冠词（DET）、描述名词的形容词（ADJ）以及一个位于句法树右侧的介词短语（PP）。因此，一个名词短语可能包含另一个名词短语作为其中的成分。图 1-10 展现了一些由上述规则支配的名词短语的例子。

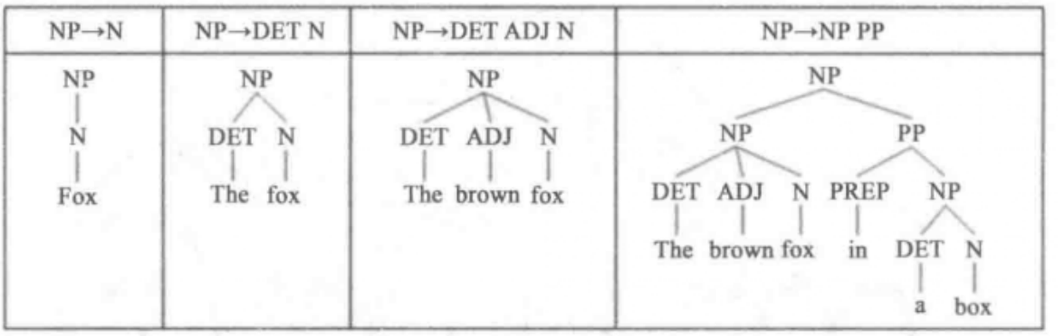


图 1-10 名词短语结构规则的成分结构句法树示意图

图 1-10 中的句法树为我们展示了一个名词短语中常包含的各种各样的成分。如前所述，在产生式规则左侧的使用 NP 表示的名词短语也可能会出现在产生式规则的右侧，正如前面的例子所示。这个特性称作递归（recursion），在本章的最后我们将会讨论递归。

接下来介绍动词短语的表示规则。规则的形式是 VP→V | MD［VP］［NP］［PP］［ADJP］［ADVP］，其中的第一个单词通常是一个动词（V）或者一个情态动词（MD）。情态动词本身是一个助动词，但是我们给予它一个不同的表达方式以区别于普通动词。头单词之后是一个可选的动词短语（VP）、名词短语（NP）、介词短语（PP）、形容词短语（ADJP）或者状语短语（ADVP）。当我们使用二分法切分语句的时候，动词短语总是第二组件，名词短语是第一组件。图 1-11 给出了一些例子以说明不同类型的动词短语通常是怎样构成的，以及它们的句法树形式。

如前所述，图 1-11 中的句法树展现了动词短语中各类成分的表示形式。采用递归特性，一个动词短语中同样可能包含另一个动词短语，你可以在第二个句法树中观察到这一现象。你还可以看到前面提到的层次结构，尤其是在句法树的第三层和第四层，其中 NP 和 PP 本身是 VP 下的成分，而它们还可以进一步拆分成为更小的成分。

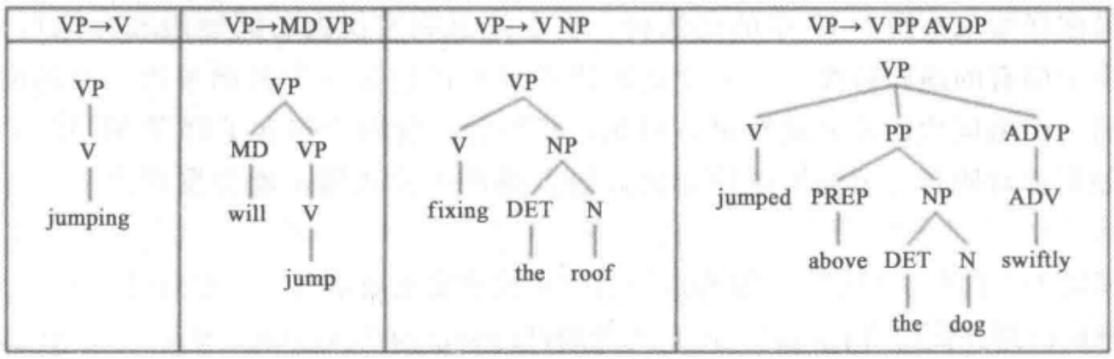


图 1-11 动词短语结构规则的成分结构句法树示意图

既然已经看过许多在例句中使用的介词短语，接下来，就让我们一起了解表示介词短语的产生式规则。基本原则的形式是 PP→PREP［NP］，其中 PREP 代表介词，它是头单词，后面是可选择性添加的名词短语（NP）。图 1-12 展示了介词短语的一些表现形式以及相应的句法树。

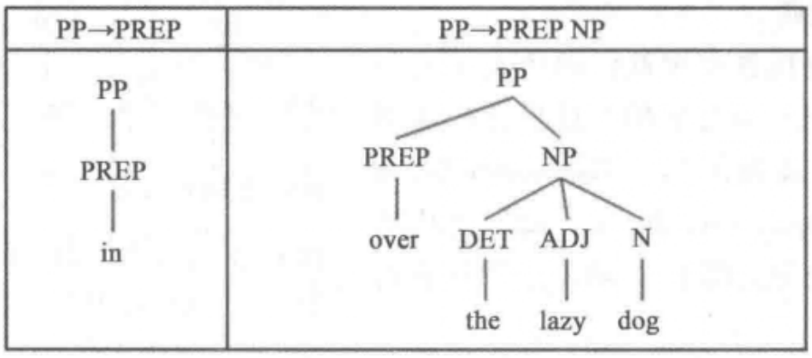


图 1-12 介词短语结构规则的成分结构句法树示意图

上图中的两例句法树展示了介词短语的不同表示方式。

递归作为语言的一个固有特性，允许成分嵌入在其他成分中，这一特性可以由同时出现在产生式规则两侧的短语类型说明。递归使我们能够从语句中创建出基于成分结构的长句法树。图 1-13 给出了一个简单的例子，句子“The flying monkey in the circus on the trapeze by the river”使用成分结构分析树的表示方式。

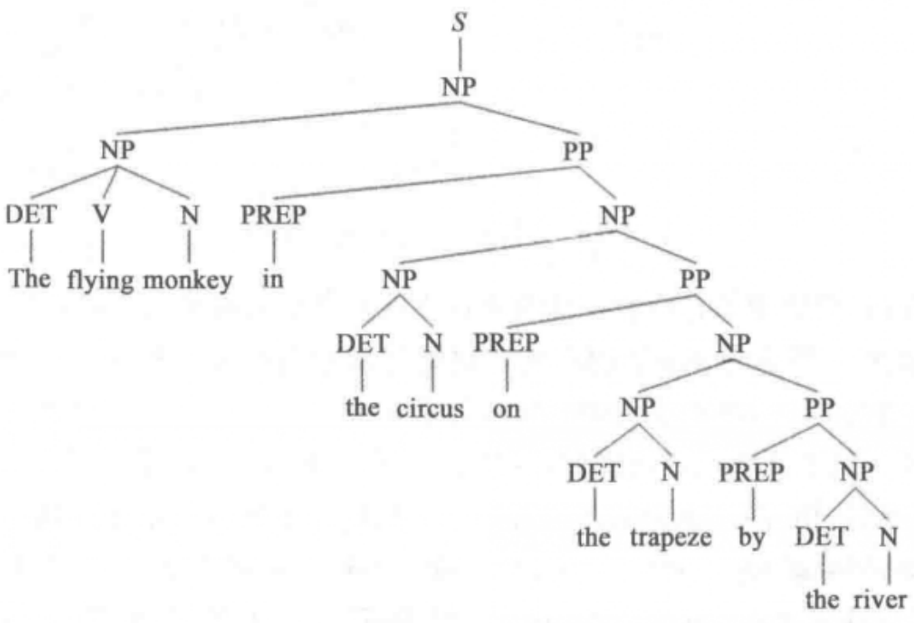


图 1-13 成分之间递归关系的成分结构句法树示意图

如果你仔细观察图 1-13 中的句法树，将会注意到它仅由名词短语和介词短语构成。然而，由于固有的递归特性，一个介词短语本身可以包含一个名词短语，而名词短语可以包含另一个名词短语或介词短语，因此，我们会发现图中包含了许多 NP 和 PP 的层级结构。如果你对所有的名词短语和介词短语重温产生式规则，就会发现树中的成分均符合规则。

连词用于连接句子和短语，是构成语言句法的重要组成部分。一般来说，词、短语，甚至从句都可以用连词组合在一起。产生式规则可以表示为 $S \rightarrow S$ conj $S \forall S \in \{S, \text{NP}, \text{VP}\}$，其中用 conj 表示将两个成分联结在一起的连接词。由两个名词短语和一个介词组成的名词短语的简单例子是“The brown fox and the lazy dog”。图 1-14 展示了该句的成分结构句法树，描述了固有的产生式规则。

图 1-14 展示了最上层的名词短语就是句子本身，它还拥有两个名词短语作为它自己的成分，它们通过连接词联结在一起，因此满足我们前述的产生式规则。

如果我们想要用连接词联结两个句子或分句会出现什么情况？通过采用上述所有的规则和规约，我们可以为例句“The brown fox is quick and he is jumping over the lazy dog”生成基于成分结构的句法树。图 1-15 展示了例句的句法表达。

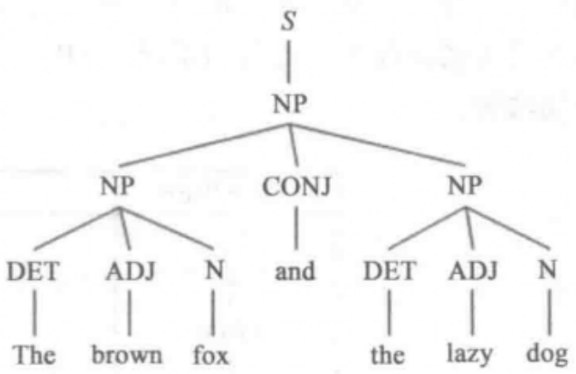


图 1-14 由连接词联结的名词短语的成分结构句法树示意图

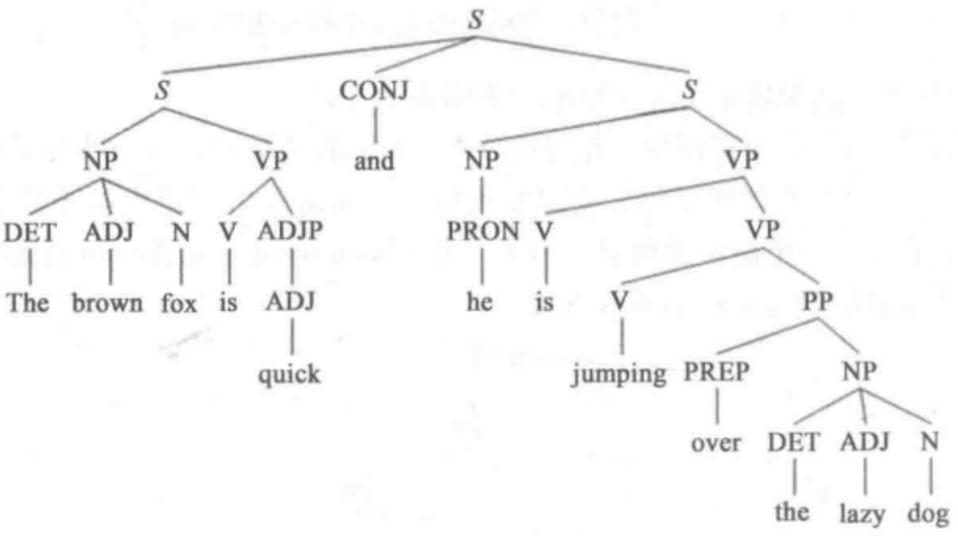


图 1-15 例句的成分结构句法树

由图 1-15 可以得出例句有两个主要的分句或成分（如前所述），它们由一个并列连接词（and）联结。此外，基于成分语法的产生式规则将最顶层的成分进一步切分成包含短语或者单词的成分。在图 1-15 的句法树中，你可以看出它确实展示了语句中单词的顺序，并且是一个层级分明、拥有无指向性边的树状结构。因此，相较于拥有无序单词和有指向性边的基于依存语法的句法树/图，成分语法大有不同。目前，由成分语法派生出了一些流行的语法架构，包括短语结构语法（Phrase Structure Grammar）、对弧语法（Arc Pair Grammar）、词汇功能语法（Lexical Functional Grammar），以及著名的上下文无关文法（Context Free

Grammar），它广泛适用于正式语言的描述。

1.3.5 语序类型学

语言学中的类型学（typology）是一个专门处理基于句法、结构、功能的语言分类的领域。语言有多种分类方法，最常见的模型之一是根据它们的主导词顺序分类，也称为语序类型学（word order typology）。分句中的主要语序包括主语、谓语和宾语。当然，并不是所有的分句都使用主语、谓语和宾语，并且在某些语言中，通常不会使用主语和宾语。但是，存在几种不同类型的语序，可用于分类各种各样的语言。Russell Tomlin 在 1986 年完成了一个调研，如表 1-1 所示，展示了一些来自于他的分析洞见。

表 1-1 基于语序的语言分类，由 Russell Tomlin 于 1986 年调研

序 号	语 序	语言占比	语言示例
1	主语 - 宾语 - 谓语	180（45%）	梵文、孟加拉语、哥特语、印地语、拉丁语
2	主语 - 谓语 - 宾语	168（42%）	英语、法语、中文、西班牙语
3	谓语 - 主语 - 宾语	37（9%）	希伯来语、爱尔兰语、菲律宾语、亚拉姆语
4	谓语 - 宾语 - 主语	12（3%）	Baure、马达加斯加语、Aneityan
5	宾语 - 谓语 - 主语	5（1%）	Apalai，希卡利亚纳语，Arecua
6	宾语 - 主语 - 谓语	1（0%）	瓦劳语

在表 1-1 中，我们可以观察到一共有六种主要的语序，例如英语遵循的是主语 - 谓语 - 宾语的语序。“He ate cake”是一个简单的例子，其中“He”是主语，“ate”是谓语，“cake”是宾语。表中大部分的语言遵循的是主语 - 宾语 - 谓语的语序。那么，如果翻译到这些语言中，“He cake ate”将会是正确的表达方式。图 1-16 说明了同一个句子从英语翻译到印地语中不同的表达方式。

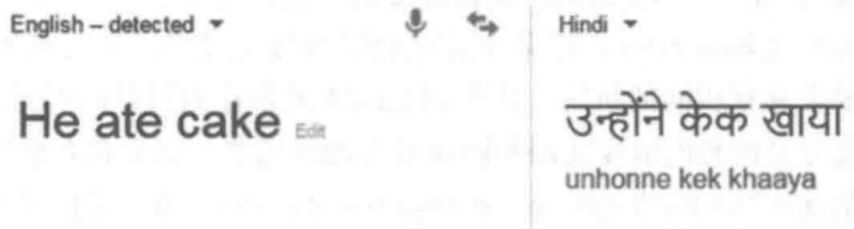


图 1-16 从英语翻译到印地语改变了例句“He ate cake”的语序类别（由谷歌翻译提供）

即使你不理解印地语，你也可以理解由谷歌提供的英语注释，其中 cake 一词（在印地语翻译的下方用 kek 表示）从右端移到了句子中间，用 khaaya 表示的动词 ate 从中间移到了句子末尾，使得语序类别变成了主语 - 宾语 - 谓语——印地语的正确语序。这个图示告诉了我们语序的重要性，以及在多种多样的语言中信息的表示方法在语法上可以大有不同。

关于句法和语言结构的讨论到此为止。接下来，将会研究语言语义学中的一些概念。

1.4 语言语义

语义学（semantics）最简单的定义是含义的研究。语言学有自身的语言语义学（linguistic semantics）的子域，研究涉及语言含义，词汇、短语以及符号之间关系，以及它们所表示知

识的暗示、含义和指代。简而言之，语义更关心的是面部表情、标志、符号、身体语言以及当信息从一个实体传递到另一个实体时所包含的知识。在句法和规则中也有很多类似的表示形式，包括前面章节讲述的各种各样的表示形式。使用正式规则或规约表示语义一直是语言学界的挑战。但是，还是有一些不同的方式以表示从语言中获得的含义和知识。本节关注语言中词汇单元（主要词汇和短语）的关系，并且探索关于知识和含义的形式化表达的一些表示形式和概念。

1.4.1 词汇语义关系

词汇语义学（lexical semantics）通常关注识别语言中词汇单元之间的语义关系，以及它们如何与句法和语言结构相关联。词汇单元通常是由语素表示，语素是有意义且语法正确的最小语言单元。词语本质上是这些语素的子集。每个词汇单元都有自己的句法、形式和含义。它们也从周围的语境（即短语、从句和句子的词汇单元）中派生出含义。词汇（lexicon）是这些词汇单元的一个完整的集合。我们将在本节中围绕词汇语义学探讨一些概念。

1. 词元和词形

词元（lemma）也称为一组词的规范或基本形式。词元通常是一组词（在本书中称为词位（lexeme））的基本形式。词元是一种特定的基本形式或头单词，它代表了词位。词形（wordform）是词元的变式，它是词位的一部分并且可以是文章中的一个词元。举个简单的例子，词位｛eating，ate，eats｝包含多种词形，它们的词元是单词 eat。

基于在句子中的位置，这些词拥有特定的含义。这也称为词的含义（sense），或者词义。词义给予了一个词所拥有的不同含义的具体表示。考虑下列句中“fair”一次的含义：“They are going to the annual fair”和“I hope the judgement is fair to all”。虽然两个句子中的“fair”一词完全一样，但是由于周围单词以及上下文的不同，词义发生了改变。

2. 同形同音异义词、同形异义词、同音异义词

同形同音异义词（homonyms）定义为具有相同拼写或发音，但具有不同含义的单词。一个替代的定义要求具有相同的拼写。这些词之间的关系称为同形同音异义（homonymy）。通常认为同形同音异义词是同形异义词和同音异义词的超集。以下例句说明了“bat”一词的同形同音异义情况：The bat hangs upside down from the tree（蝙蝠倒挂在树上）以及 That baseball bat is really sturdy（那个棒球棒真的很坚固）。

同形异义词（homogragh）是具有相同书面形式或拼写，但具有不同含义的单词。在几个替代定义中，其发音也可以不同。同形异义词的一些例子包括，“lead”一词在“I am usinga lead pencil（我正在使用一根铅笔）”和“Please lead the soldiers to the camp（请带领士兵到营地）”中的含义，以及“bass”一词在“Turn up the bass for the song（调高这首歌的低音）”和“ I just caught a bass today while I was out fishing（我今天在外面抓鱼的时候，抓到了一条鲈鱼）”中的含义。请注意，在这两种情况下，拼写均保持不变，但发音却根据句子中的上下文语境而发生变化。

同音异义词（homophone）是具有相同发音，但含义不同的单词，它们可以具有相同或不同的拼写。举例来说，“pair”（意为一对）和“pear”（意为梨）。它们听起来一样，但具有不同的含义和书面形式。通常这些词会在 NLP 中产生问题，因为使用机器智能很难找出实际的语境和含义。

3. 同形异音异义词和同音异形异义词

同形异音异义词（heteronym）是指具有相同书面形式或拼写，但具有不同发音和含义的单词。本质上，它们是同形异义词的一个子集。它们也称为异音词（heterophone），意思是拥有“不同的声音”。同形异音异义词的例子是“lead”（金属铅，命令）和“tear”（撕掉，泪水）。

同音异形异义词（heterograph）是具有相同发音，但含义和拼写不同的单词。本质上它们是同音异义词的一个子集。它们的书面表达可能会有所不同，但是它们的声音听起来很相似，或者说经常是完全一样的。例子包括单词“to”“too”和“two”，它们听起来相似，但却具有不同的拼写和含义。

4. 语义相近多义词

语义相近多义词（polysemes）是具有相同书面形式或拼写，同时具有虽然不同但却非常相关的含义的词。这与同形同音异义词非常相似，它们之间的差异是主观的，并且取决于上下文，因为这些词的含义彼此相关。一个很好的例子是“bank”这个词，它可以表示（1）一个金融机构；（2）河岸；（3）属于金融机构的建筑；（4）一个意为依赖的动词。这些例子使用相同单词“bank”，并且是同形同音异义的。但是，只有（1）、（3）和（4）是代表一个共同主题（象征着信任和安全的金融机构）的语义相近多义词。

5. 首字母大写异义词

首字母大写异义词（capitonyms）是具有相同书面形式或拼写，但是首字母大写时具有不同含义的词。它们可能有不同的发音。例子包括“march”和“may”，“March（三月）”表示月份，“march（行军）”则描绘行走的行动；“May（五月）”表示月份，“may”则是情态动词。

6. 同义词和反义词

同义词（synonym）是具有不同发音和拼写，但在某些或所有上下文中均具有相同含义的单词。如果两个词或词位是同义词，则它们可以在各种上下文中相互替代，并且表示它们具有相同的主题含义。同义词之间是彼此同义的，而同义词的状态被称为同义关系（synonymy）。然而，完美的同义词几乎不存在。因为同义词更多是感官和情境含义之间的关系，而不仅仅是词语。让我们细思同义词“big，huge，large”之间的区别。它们都非常相似并且可以在句子“That milkshake is really（big/large/huge）”中完美表达其意思。但是，对于句子“Bruce is my big brother”来说，如果我们用“huge”或“large”代替“big”的话，这句话就失去了意义。这是因为这里的“big”有一个描述长大、更老的语境或含义，而另外两个同义词缺少这个含义。同义词可以存在于所有的词性中，包括名词、形容词、动词、副词和介词。

反义词（antonym）是定义为拥有二元对立关系的一对单词。这些单词表示完全相反的具体感觉和含义。反义词的状态称为反义关系（antonym）。反义词有三种：分级反义词、互补反义词和关系反义词。正如其名字所暗示的，分级反义词是连续衡量时具有一定等级或层次的反义词，例如一对反义词“fat（胖的），skinny（极瘦的、皮包骨的）”。互补反义词是指具有相反意义的一对词，但它们不能以任何等级或尺度衡量。互补反义词的一个例子是“divide（分开、使……产生分歧），unite（团结、联合）”。关系反义词是拥有一定关系的一对词语，其反义关系是基于语境的，并由这种特定关系所表示。关系反义词的一个例子是“doctor（医生），patient（病人）”。

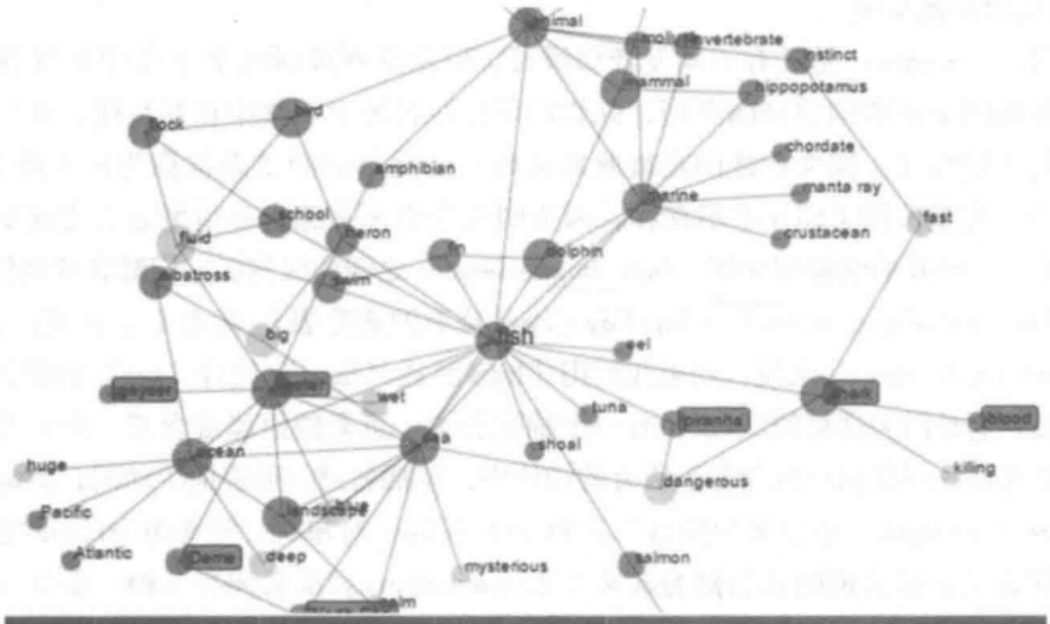
7. 下位词和上位词

下位词（hyponym）通常是另一个词的子类。在这种情况下，与它们的超类相比，下位词通常具有非常具体的含义和语境。上位词（hypernym）是下位词的超类，与下位词相比，它们具有更通用的含义。举例来说，单词“fruit”是一个上位词，“mango”“orange”和“pear”是这个词的下位词。这些词之间关系通常称为下位关系（hyponymy）和上位关系（hypernymy）。

1.4.2 语义网络和模型

我们已经看到了几种将词语与含义之间的关系具象化的方法。让我们思考词汇语义，现在已经有了一些方法可以找出每个词汇单元的含义，但是如果我们想要表示一些概念或理论的含义，而这些概念或理论需要将词汇单元结合在一起并根据它们的意义形成相互之间的联系呢？语义网络（semantic network）旨在使用网络或图来解决表示知识和概念的问题。

语义网络的基本单位是实体或概念。一个概念可以是有形或抽象的事物，比如一个想法。概念集合彼此拥有一定关系，这些关系可以用有向或无向边来表示。每条边都表示两个概念之间的特定关系。比如我们谈到“fish（鱼）”的概念，基于与鱼这一概念的关系，我们可以对鱼有不同的理解。例如，fish“is-a”animal（鱼是一种动物）和 fish“is-a”part of marine life（鱼是一种海洋生物）。这类关系称为“is-a”关系。其他类似的关系包括 has-a、part-of、related-to 等，这些关系取决于语境和语义。概念和关系合在一起形成了一个语义网络。目前，网上已经有了一些具有广阔的知识基础且横跨多个不同概念的语义模型。图 1-17 展示了一个与鱼有关的概念表示。该模型由 Nodebox（www. nodebox. net/perception/）提供，你可以在 Nodebox 中搜索各种概念，并以同样的展示形式看到其他相关概念。



IS-A	HAS-SPECIFIC	IS-PART-OF	HAS-PART	HAS-PROPERTY	IS-RELATED-TO
animal	eel	ocean	fin	wet	heron
marine	piranha	school			
	salmon	sea			
	shark	water			
	shoal				
	tuna				

图 1-17 关于鱼的概念的语义网络

在图 1-17 的网络中，我们可以看到前面讨论的一些关于鱼的概念，以及一些具体的鱼类，如鳗鱼、鲑鱼、鲨鱼等，这些是鱼这一概念的下位词。这些语义网络由使用图结构的语义数据模型表示，其中概念或实体由点表示，关系由线表示。语义网（Semantic Web）使用语义元数据注释，以及基于诸如资源描述框架（Resource Description Framework，RDF）和 Web 本体语言（Web Ontology Language，OWL）之类的数据建模技术来作为万维网的扩展。在语言学中，有一个丰富的词汇语料库和数据库，称为 WordNet，它提供了一个关于不同词汇实体集合基于相似语义（如同义词）映射到同义词集的详尽列表。这些同义词集和各种单词之间的语义关系可以在 WordNet 中进行探索，使其在本质上成为一种语义网络。后续章节中，在我们讲述文本语料库时，将更详细地讨论 WordNet。

1.4.3 语义表示

到目前为止，我们已经了解如何基于词汇单元来表示语义，以及词汇单元是如何借助语义网络相互关联的。但是，如果我们考虑通过消息传递的常规形式的通信，无论是书面还是口头，如果实体向另一实体发送消息，并且该实体基于该消息采取一些特定行动，则认为第二实体已经理解了被传达信息的含义。我们可能会想到的一个问题是，如何正式地表示一个简单句子所传达的含义或语义。虽然对我们来说，理解所传达的意义可能非常容易，但是正式的表示语义并不像看起来的那么简单。

让我们一起来看“Get me the book from the table（把书从桌上拿来）”这个例子。这句话本质上是一个指令，它指示听者做某事。理解这句话所传达的意义除了需要理解从桌子上拿出书的实际行为，还可能涉及语用学，比如具体是哪本书？具体是哪张桌子？虽然人类的思维是直观的，但是正式表达各种成分之间的含义和关系仍是一个挑战——我们可以使用诸如命题逻辑（Propositional Logic，PL）和一阶逻辑（First Order Logic，FOL）等技术来实现。使用这些表示方法，读者可以表示不同句子所包含的含义，从中推断，甚至可以发现一个句子基于其语义是否需要另一个句子。语义表示对于我们开展各种 NLP 操作使得机器能够使用适当的表达来理解消息背后的语义来说，是非常有用的，因为机器缺乏人类内在禀赋的认知能力。

1. 命题逻辑

命题逻辑（PL）也称为语句逻辑（sentential logic）或陈述逻辑（statement logic），定义为关于命题、陈述和句子研究的逻辑学科。它包括研究命题和陈述之间的逻辑关系和属性，将多个命题组合起来形成更复杂的命题，并观察命题的价值如何基于其成分和逻辑运算符的变化而改变。命题或陈述通常是声明性的，并且具有非真即假的布尔值。通常，陈述更具有语言特指性和具体性，而命题更倾向于传达的观念或概念。一个简单的例子就是下面两个陈述“The rocket was faster than the airship（火箭比飞艇快）”和“The airship was slower than the rocket（飞艇比火箭慢）”，它们是不同的句子，但是表达了同样的含义或命题。尽管如此，陈述和命题在命题逻辑中通常是可互换使用的。

命题逻辑的主要关注点是研究不同的命题以及如何将各种命题与逻辑运算符组合以改变整个命题的语义。这些逻辑运算符更多地用作连接词或并列连接词（如果你记得前文所讲述的）。运算符包括诸如“and”“or”“not”这样的词项，它们可以单独改变一个命题或者与几个组合在一起的命题的含义。一个简单的例子是下面两个命题，“The Earth is round

(地球是圆的)”和“The Earth revolves around the Sun (地球围绕太阳转动)”。它们可以通过逻辑运算符结合起来，并给出我们的命题“The Earth is round and it revolves around the Sun”，这表明了在运算符“and”两侧的命题必须都是真，组合命题才能够是真。

命题逻辑的优点是每个命题均有自己的真值，并且其真值与命题进一步细分后较小的语块的逻辑特征无关。每一个命题均可看作一个具有自身真值的不可分割的整体。逻辑运算符可以应用于一个或多个命题。在这里我们不考虑命题的细分部分，如分句或短语。为了代表命题逻辑的各种构成模块，我们使用一些规约和符号。大写字母用于表示单独的陈述或命题，例如 P 和 Q。表 1-2 中列出了命题逻辑使用的不同运算符及其相应符号，根据它们的优先顺序进行了排序。

表 1-2 逻辑运算符的符号及其优先级

序 号	运 算 符 号	操 作 含 义	优 先 级
1	¬	非 (not)	最高
2	∧	和 (and)	
3	∨	或 (or)	
4	→	若则 (if-then)	
5	↔	当且仅当 (iff——if and only if)	最低

可以看到总共有五个运算符，其中 not 运算符优先级最高，而 iff 运算符优先级最低。逻辑常量为 True 或 False。常量和符号称为基本单位——所有其他单位(更具体地说语句和陈述)都是复合单位。一段文字通常是一个基本陈述，或者是其使用了 not 运算符后的否定式。

我们来看一个简单的例子，两个句子 P 和 Q，对它们应用各种运算符。如下所示：

P：He is hungry (他饿了)。

Q：He will eat a sandwich (他会吃一个三明治)。

表达式 P∧Q 意味着“He is hungry and he will eat a sandwich (他饿了，并且他会吃一个三明治)”。这表示这个操作的结果本身也是一个语句或命题。这种是连接操作，其中 P 和 Q 是连接项。只有在 P 和 Q 都是 True 的时候，这个句子才是 True。

表达式 P∨Q 意味着“He is hungry or he will eat a sandwich (他饿了，或者他会吃三明治)”。这表明这个操作的结果也是一个命题，它由分离操作形成，其中 P 和 Q 是分离项。如果 P 和 Q 其中一项为 True，则该句子即为 True。

表达式 P→Q 表示“If he is hungry, then he will eat a sandwich (如果他饿了，那么他会吃三明治)”。这种是蕴含操作，它确定 P 是前提或先行条件，而 Q 是后果。它就像是一个规则，说明了只有 P 已经发生或者为 True，Q 才会发生。

表达式 P↔Q 意为“He will eat a sandwich if and only if he is hungry (当且仅当他饿了的时候，他会吃一个三明治)”，这基本上是“If he is hungry then he will eat a sandwich (P→Q) (如果他饿了，那么他会吃三明治)”和“If he will eat a sandwich, he is hungry (Q→P) (如果他要吃三明治，那么他是饿了)”两种表达的组合。当且仅当上述的两个蕴涵操作评估为 True 时，这种双值条件或等价操作才会评估为 True。

表达式¬ P 意为“He is not hungry (他不饿)”，这是否定操作，当且仅当 P 判断为 False 时，其结果才是 True。

这给了我们一个命题间基本操作和复杂操作的概念，复杂操作可以通过多个逻辑连接

符或者增加更多命题实现。举一个简单的例子，陈述 P：We will play football（我们准备去踢足球）、Q：The stadium is open（体育馆是开着的）、R：It will rain today（今天会下雨）可以组合并表示为 Q∧¬ R→P 来描绘复杂的命题：If the stadium is open and it does not rain today, then we will play football（如果体育场是开着的，并且今天不下雨，那么我们会去踢足球）。最终命题语义的真值或结果可以根据单个命题的真值和所使用的运算符进行评估。使用不同运算符所得出的真值结果如图 1-18 所示。

P	Q	¬P	P∧Q	P∨Q	P→Q	P↔Q
False	False	True	False	False	True	True
False	True	True	False	True	True	False
True	False	False	False	True	False	False
True	True	False	True	True	True	True

图 1-18 各种逻辑连接的真值

这样，使用图 1-18 中的表格，我们就可以通过将更加复杂的命题分解成简单的布尔运算，然后评估每个布尔运算的真值，再逐步合成结果，来实现复杂命题的评估。

除了这些结果，其他属性如结合性、交换性和分配性均有助于评估复杂命题的结果。检查每个操作和命题的有效性，并最终评估结果的行为也称为推理（inference）。然而，除了全面广泛地评估真值表，我们还可以利用一些推理规则来得出最终结果或结论。这样做的主要原因是，随着命题数量的增加，这些包含各种运算符的真值表的大小开始呈指数级增长。

此外，推理规则更容易被理解并且经过了良好的测试验证，其核心原理同样使用了真值表——但是我们不必为其内部原理而操心。通常，应用一系列推理规则后，可以得出一个结论，这个结论通常称为逻辑证明（logical proof）。推理规则的通用形式是 P⊢Q，表示 Q 可以通过由 P 表示的语句集合的某些推导操作得出。转向符号（⊢）表示 Q 是 P 的一些逻辑结论。最常用的推理规则如下所示。

- 取式推理（Modus Ponens）：可能是目前最流行的推理规则，它也称为蕴涵消除规则（Implication Elimination rule）。它可以表示为 {P→Q, P}⊢Q，即如果 P 意味着 Q，并且 P 被认定为 True，那么可以推断 Q 也是 True。你还可以用（(P→Q)∧P)→Q 表示上述关系，使用真值表便可以轻松评估该表达式。举一个简单的例子，“If it is sunny, we will play football（如果天晴，我们将踢足球）”以 P→Q 表示。现在如果我们说“天确实是晴朗的”，这表明 P 是 True，那么 Q 便自动推断为 True，即表明“我们将踢足球”。
- 拒式推理（Modus Tollens）：这与前面的规则非常相似，正式的表示是 {P→Q, ¬ Q}⊢¬ P，即如果 P 意味着 Q，并且 Q 实际上被断言为 False，则推断 P 也是 False。你还可以用（(P→Q)∧¬ Q)→¬ P 表示上述关系，使用真值表可以轻松评估该表达式。一个命题例子是“If he is a bachelor, he is not married（如果他是一个单身汉，那么他没有结婚）”，由 P→Q 表示。现在如果我们说“他已经结婚了”，由¬ Q 表示，那么我们可以推断¬ P，这意味着“他不是单身汉”。
- 选言推理（Disjunctive Syllogism）：也称为析取消去（Disjunction Elimination），其正式的表示是 {P∨Q, ¬ P}⊢Q，即 P 或 Q 是 True，并且 P 是 False，那么 Q 是 True。一个简单的例子是“He is a miracle worker or a fraud（他是一个奇迹创造者或者是个

骗子)”，由 P∨Q 表示，“他不是一个奇迹创造者”由¬ P 表示。那么，我们可以推断“他是一个骗子”，由 Q 表示。

- 假言推理（Hypothetical Syllogism）：这通常称为演绎链规则（Chain Rule of Deduction），正式的表示是 {P→Q，Q→R} ⊢ P→R，即如果 P 意味着 Q，并且 Q 意味着 R，我们可以推断 P 意味着 R。举一个有趣的例子来帮助读者理解，“If I am sick，I can’t go to work（如果我生病了，我就不能去工作）”由 P→Q 代表，“If I can’t go to work，the building construction will not be complete（如果我不能上班，建筑施工就无法完工）”，由 Q→R 表示。那么，我们可以推断出“如果我生病了，建筑施工就无法完工”，可以用 P→R 表示。
- 构造性二难推理（Constructive Dilemma）：该推理规则是取式推理的析取版本，正式的表示是 {(P→Q)∧(R→S)，P∨R} ⊢ Q∨S，即如果 P 意味着 Q，R 意味着 S，P 或 R 为 True，则可以推断 Q 或 S 为 True。考虑以下命题：“If I work hard，I will be successful（如果我努力工作，我将会取得成功）”以 P→Q 代表，“If I win the lottery，I will be rich（如果我赢得彩票，我将会变得富有）”以 R→S 代表。现在我们说我努力工作或赢得彩票是 True，由 P∨R 表示。那么我们可以推断出“我将会取得成功，或者我将会变得富有”，由 Q∨S 表示。这个规则的补集是破坏性二难推理规则（DestructiveDilemma），即拒式推理规则的析取版本。

以上内容应该可以给你一个关于推理规则的清晰直观的认识，当我们试图找出复杂命题的结论时，使用它们比使用多个真值表容易许多。从推理规则中得出的解释可以给我们语句或命题的语义。一个有效的陈述是一个在所有解释下都为 True 的语句，不论有何种逻辑运算符或各种陈述在其中。这通常称为重言式（tautology）。重言式的补集是在所有解释下都是 False 的矛盾陈述或前后不一致的陈述。请注意，前面的列表只是最流行的推理规则的参考列表，并不是详尽的列表。感兴趣的读者可以阅读更多关于推理规则和命题演算的资料，以了解本文以外的其他规则和公理。

接下来我们将讨论一阶逻辑，它可以解决命题逻辑中存在的一些不足。

2. 一阶逻辑

一阶逻辑（FOL）通常称为谓词逻辑（predicate logic）和一阶谓词演算（first order predicate calculus），定义为一种定义明确的形式系统的集合，其广泛用于演绎、推理和表征知识。FOL 允许我们在句子中使用数量词和变量，这使我们能够克服命题逻辑中的一些限制。如果我们要考虑命题逻辑（PL）的利弊，需要考虑到有利的方面，即 PL 是声明性的，并且允许我们使用格式良好的句法轻松地阐述事实。PL 还允许复杂表示，如联结、析取和否定的认知表示。这本质上使得 PL 组合中复合或复杂命题是由简单命题和逻辑连接词构成，而简单命题则是复杂命题的组成部分。然而，有几个领域是 PL 无法涉及的。在 PL 中想要表示事实绝非易事，因为对于每个可能的基本事实，我们都会需要一个唯一的符号表示。因此，由于这种限制，PL 具有非常有限的表现力。也正因为如此，FOL 背后的基本思想是不将命题视为基本实体。

与命题逻辑（PL）相比，FOL 具有更加丰富的句法和更多的必要组件。FOL 的基本组件如下。

- 对象：是具有独特身份的具体实体或事物，如人、动物等。

- 关联：也称为谓词，通常包含对象或对象集合，并表达某种形式的关系或连接，如 is_man、is_brother、is_mortal。关系通常对应于动词。
- 函数：是一些关联的子集，其中对于某些给定的输入，它们永远只有一个输出值或对象，如 height、weight、age_of。
- 属性：指有助于将其与其他对象区分开的特定属性，如 round、huge 等。
- 连接：指与 PL 中类似的逻辑连接，包括 not（¬）、and（∧）、or（∨）、implies（→）、iff（当且仅当↔）。
- 量词：包括两种类型的量词：全称量词（∀），代表“for all”或“all”；存在量词（∃）代表“there exists”或“exists”。它们以逻辑或数学的表达方式量化实体。
- 常量符号：用于表示具体的实体或对象。如 John、King、Red 和 7。
- 变量符号：用于表示变量，如 x、y 和 z。
- 函数符号：用于将功能映射到结果。如 age_of(John) = 25 或 color_of(Tree) = Green。
- 谓词符号：将具体实体以及它们之间的关系或函数映射到基于结果的真值。例如 color(sky, blue) = True。

这些是研究 FOL 逻辑表示和句法的主要组成部分。通常，对象由各种项表示，它们可以是上述不同组成部分中的函数、变量或者常量。这些项不需要定义，也不返回数值。借助谓词符号，使用谓词和项通常可以构建各种命题。一个 n 元谓词由一个关于 n 元词项的函数构成，它具有 True 或 False 结果。一个原子语句（atomic sentence）可以由一个 n 元谓词表示，并且结果是 True 或 False 取决于句子的语义——也就是说，取决于由词表示的对象在由谓词指定对象之间的关系是否正确。一个复杂语句或陈述是由几个原子语句和逻辑连接词形成。一个量化语句则在句子中包含前述的量词。

量词是一阶逻辑相对于命题逻辑的一个优势，它让我们可以表示关于全体对象集合的语句，而不需要使用不同的名字表示和列举每个对象。全称量词（∀）断言一个特定关系或者谓词对于某个变量的所有值取值均为真。∀x F(x) 表示 F 取值为 x 值域与 x 相关的所有值。对于∀ x cat(x) →animal(x) 这样一个例子，表示所有的猫都是动物。

全称量词经常与蕴含连接符（→）一起使用，形成规则或陈述。需要记住的是全称量词不和与连接符（∧）一起在陈述中使用来表示世界上涉及每个实体的关系。表示为∀x dog(x) ∧eats_meat(x)的例子，实际意思是世界上每个实体都是狗而且吃肉，这听起来有些荒诞！存在量词（∃）表示对于某个变量至少存在一些值使得一个特定的关系或预测取值为真。表达式∃x F(x) 表示对于一些 x 取值范围内的 x 值使得 F 取值为真。例如∃x student(x)∧pass_exam(x)，表示至少 1 个学生通过了考试。因为我们在不指定对象或实体名字的情况下，可以表达陈述，这个量词增加了一阶谓词表达能力。存在量词经常与与连接符（∧）一起使用来形成规则和陈述。需要注意的是存在量词几乎不与蕴含符连接符（→）一起在陈述中使用，因为其表达的语义一般也是错误的。例如∃ x student(x)→knowledgeable(x)，告诉我们如果你是个学生你就是知识渊博的，但事实上会有问题，如果你问不是学生的人知识就不渊博会怎么样？

量词的嵌套范围和量词相乘的顺序是否有影响取决于所使用的量词的类型。对于全称量词相乘，交换顺序不改变陈述的意义。这个可以描述为(∀x)(∀y)brother(x,y) ↔

(∀y)(∀x) brother(x,y)，x 和 y 作为变量符号表示两个人彼此是兄弟，而与顺序无关。同样，你可以交换存在量词的顺序，例如(∃x)(∃y) F(x,y) ↔ (∃y)(∃x)F(x,y)。交换陈述中混合量词的顺序影响和改变陈述的含义。下面的例子详细解释这一点，这些是一阶谓词中典型的例子：

- (∀x)(∃y) loves(x, y)意思是世界上每个人至少喜爱某个人。
- (∃y)(∀x) loves(x, y)意思是世界上的某个人被每个人喜爱。
- (∀y)(∃x) loves(x, y)意思是世界上的每个人至少有某个人喜爱他们。
- (∃x)(∀y) loves(x, y)意思是世界上至少有某个人喜爱每个人。

从上面的例子，你可以看到陈述基本相同，但是量词顺序的改变使语句含义产生了非常明显的改变。还有几个关于量词之间关系的特性。一些主要的量词恒等式和特性如下所示：

- (∀x) ¬ F(x) ↔ ¬ (∃ x) F(x)
- ¬ (∀x) F(x) ↔(∃ x) ¬ F(x)
- (∀x) F(x) ↔¬ (∃ x)¬ F(x)
- (∃x) F(x) ↔ ¬ (∀ x)¬ F(x)
- (∀x) (P(x) ∧Q(x)) ↔ ∀ x P(x) ∧ ∀ x Q(x)
- (∃x) (P(x) ∨Q(x)) ↔ ∃ x P(x) ∨ ∃ x Q(x)

谓词逻辑中存在几个重要的规则转换的概念。这些包括*例示规则*（instantiation）和*推广规则*（generalization）。*全称量词例示规则*（universal instantiation）也称为*全称量词消去规则*（universal elimination），是一种涉及全称量词的推理规则。它告诉我们，如果（∀x）F(x)是真的，则F(C)是真的，C是x值域的任意常数项。这里的变量符号可以被任何基项替换。例如（∀x）drinks (John, x) → drinks (John, Water)。

全称量词推广规则（universal generalization）又称为*全称量词引入规则*（universal introduction），这个推理规则告诉我们如果F(A)∧F(B)∧F(C)∧…为真，则我们可以推理（∀x）F(x)为真。*存在量词例示规则*（existential instantiation）也称为*存在量词消去规则*（existential elimination），是一个涉及存在量词的推理规则。它告诉我们，如果给定的表示（∃x）f(x)存在，对于一个新的常数或变量符号C我们可以推断出F(c)。这假设在这个规则中引入的常量或变量C应该是一个全新的常数，以前没有出现在这个证明或在我们现有的完整的知识基础中。这个过程也称为*斯科林化*（skolemization），常量C称为skolem常量。*存在量词一般化*（existenal generalization）也称为*存在量词引入规则*（existential introduction），是推理规则，告诉我们，假设f(C)是真的，C是一个常数项，那么我们可以推断（∃x)f(x)。这可以通过eats_fish(CAT)→(∃x)eats_fish(x)表示，可译为“猫吃鱼，因此至少存在某物或某人吃鱼”。

现在，我们将看一些如何用FOL来表示自然语言语句的例子，反之亦然。表1-3中的例子描述了一些一阶谓词（FOL）表示自然语言陈述的典型用法。

表1-3 使用一阶逻辑的自然语言陈述表达

SI序号	一阶谓词表示	自然语言陈述
1	¬ eats (John, fish)	约翰不吃鱼
2	is_hot(pie)∧is_delicious(pie)	这个饼是热的并且美味
3	is_hot(pie)∨ is_delicious(pie)	这个饼是热的或是美味的

（续）

SI 序号	一阶谓词表示	自然语言陈述
4	study (John, exam) →pass (John, exam)	如果约翰为考试学习，他将通过考试
5	∀ x student (x) → pass (x, exam)	所有学生通过了考试
6	∃ x student (x) ∧ fail (x, exam)	至少有一个学生没通过考试
7	(∃ x student (x) ∧ fail (x, exam) ∧ (∀ y fail (y, exam) → x = y))	就只有一个学生没通过考试
8	∀x (spider(x)∧black_widow(x))→poisonous(x)	所有黑寡妇蜘蛛都有毒

上述内容告诉我们一阶谓词的不同组成、使用和相对于命题逻辑的优点。但是一阶谓词也有自身的局限。就其本质而言，一阶谓词允许我们用量词限定变量和对象，但不能限定属性或关系。高阶谓词（Higher Order Logic，HOL）允许我们用量词限定关系、谓语和函数。更具体地说，二阶谓词允许我们用量词限定谓语和函数，三阶谓词允许我们用量词限定谓语的谓语。谓词的表达能力越强，越难确定高阶谓词所有陈述的正确性。

1.5 文本语料库

*文本语料库*是*文本语料*的复数形式。文本语料库是大量的结构化文本或文本数据集合，一般由书面或口语文本组成，常以电子形式存储。这包括将历史悠久的文本语料从物理形式转换成电子形式，以便于分析与处理。文本语料库的主要目的是便于语言学分析和统计分析，以及作为构建自然语言处理工具的数据。*单语言语料*由一种语言的文本数据组成，*多语言语料*则由多种语言的文本数据组成。

认识文本语料库的重要性，有助于理解语料库的起源以及其背后的原因。语料库与语言学一同出现，人们收集与语言相关的数据来学习语言的特性和结构。在 20 世纪 50 年代，统计和数量分析方法被用于分析收集到的数据。但由于缺乏统计方法有效应用所需的大量文本数据，这种努力很快到了尽头。此外，认知学习与行为科学获得了大量关注。这使得著名的语言学家乔姆斯基建立和形式化了一个复杂的基于规则的语言模型，该语言模型成为建立、标注、分析大规模文本语料库的基础。

1.5.1 文本语料库标注及使用

文本语料库使用了丰富的元数据进行标注，当使用文本语料进行自然语言处理和文本分析时，这对于获得有价值的洞见是非常有意义的。常见的语料标注包括词性标注（POS）、词干、词元等。下面是最常用的标注语料库的技术和方法。

- *词性（POS）标注*：这主要用词性标记表示每个单词与之相关的词性。
- *词干*：词干是词的一部分，各种词缀可以附加在词干上构成词。
- *词元*：词元是一组词的规范或基本形式，也称为中心词。
- *依存语法*：这包括查找句子成分之间的各种关系并标注这些依赖关系。
- *成分语法*：这是基于成分对句子增加句法标注，包括短语和从句。
- *语义类型和角色*：标注句子的不同成分包括单词和短语的语义类型和角色，这通常从一个表明它们做什么的本体中获得。这些包括地点、人物、时间、组织、代理、接受者、主题等。

高级的标注形式包括对文本增加句法和语义结构。这些是基于语法分析树的依存语法和成分语法。这些专业语料库，又称为树库，广泛应用于构建 POS 标注工具、句法和语义分析器。语料库也被语言学家广泛用于创建新字典和语法。语料库的属性，如一致性、搭配和频率计数使他们能找到词汇信息、模式、形态信息与语言学习。除了语言学外，语料库也广泛应用于开发自然语言处理工具，如文本标注工具、语音识别、机器翻译、拼写和语法检查器、文本到语音和语音到文本合成器、信息检索、对象识别和知识提取。

1.5.2 热门的语料库

目前，已经建成了一些流行的文本语料库资源，并随着时间的推移而不断演变。

本节列出了一些最著名和热门的语料库。你可以研究和了解你所感兴趣的文本语料库的更多细节。下面是一些长期积累下来的热门文本语料库。

- 上下文关键字：上下文关键字（KWIC）是一个 19 世纪 60 年代发明的方法，被语言学家在 20 世纪 50 年代广泛使用，以索引文件和创建一致性语料库。
- Brown 语料库：这是第一个百万级的英文语料库，也称为“当代美国英语标准语料库”，由 Kucera 和 Francis 于 1961 年发布。该语料库由来自不同来源和分类的文本组成。
- LOB 语料库：LOB 语料库编译于 20 世纪 70 年代，是兰卡斯特大学、奥斯陆大学和挪威卑尔根人文计算中心之间合作的结果。该项目的主要目的是提供一个与 Brown 语料库对应的英国语料库。该语料库也是一个百万单词级的语料库，由多种来源和类别的文本组成。
- Collins 语料库：柯林斯伯明翰大学国际语言数据库（COBUILD）是 1980 年由柯林斯出版社资助的、在伯明翰大学建立的一个大型的英语现代文本电子语料库。这也为未来的语料库（如英语银行和柯林斯 COBUILD 英语词典）铺平了道路。
- CHILDES：儿童语言数据交换系统（CHILDES）在 1984 年由 Brian 和 Catherine 创建，作为语言数据获取的存储库，包括来自超过 130 个不同的语料库的 26 种语言的文字稿、音频和视频。近来，该语料库已经合并了一个较大的语料库 TalkBank。其广泛用于幼儿语言与语音分析。
- WordNet：该语料库是一个面向语义的英语字典数据库，在 George Armitage 指导下于 1985 年在普林斯顿大学建立。该语料库由单词和同义词集（synset）组成。此外，该语料库有单词定义、关系以及单词和同义词集使用例子。整体来讲，它是一个字典和同义词字典的组合。
- Penn Treebank：该语料库包括标记和解析的英语句子，句子包含树库中常见的词性标注和语法分析树。它也定义为一个语言树的集合，由宾夕法尼亚大学创建，故称宾夕法尼亚大学树库。
- BNC：英国国家语料库（BNC）是最大的英语语料库之一，由多个来源的书面和口头文本样本的 1 亿个单词组成。该语料库是 20 世纪后期英国书面和口头英语的一个代表。
- ANC：ANC 是一个大型的美国英语语料库，该语料库由 20 世纪 90 年代以来的 2200 万口语和书面语单词组成。该语料拥有广阔的来源，包括新出现的而不在 BNC 中的来源，如邮件、推特和网页信息。

- COCA：美国当代英语语料库（COCA）由4.5亿个单词组成，是最大的美国英语文本语料库，包括来自不同的类别和来源的口语笔录和书面文本。
- 谷歌N-Gram语料库：谷歌N-Gram语料库由1万亿个单词组成，出自于包括书籍、网页等不同来源。该语料每种语言的n-gram文件达到了5-gram。
- 路透社语料库：该语料库专门为开展自然语言处理和机器学习而准备的，主要是2000年的路透社新闻和故事集。
- 网页、聊天记录、邮件和推特：这些都是随着社交媒体的崛起而快速发展起来的全新形式的文本语料库。这些语料库来自网上的不同来源，包括推特、Facebook、聊天室等。

上述内容让我们了解了一些最热门的文本语料库，以及它们随时间的演变情况。下一节讨论如何使用Python和自然语言工具包（NLTK）平台访问这些文本语料库。

1.5.3 访问文本语料库

我们对什么是文本语料库已经有了概念，并看到了现在还存在的几个热门的文本语料库的一个列表。在本节中，我们将使用Python和NLTK来连接和访问这些文本语料库。如果有语法或代码看起来不好理解，请不要担心，下一章会详细介绍Python和NLTK。本节的主要目的是让你了解到，你可以很容易地访问和利用文本语料库来满足自然语言处理和分析需求。

我们将使用IPython shell（https://ipython.org）运行Python代码，它为运行代码以及查看图表提供了一个功能强大的交互shell。我们也将使用NLTK函数库。你可以在www.nltk.org找到这个项目的更多细节信息，NLTK是一个用于访问文本资源的完整的平台和框架，包括各种NLP和机器学习功能的语料库和程序库。

开始之前，要先确认已经安装了Python。你可以单独安装Python或从连续分析网站www.continuum.io/downloads下载流行的Anaconda Python。这个版本包括NLTK的完整分析套件。如果你想深入了解Python以及哪个版本更适合你，第2章将更加详细地介绍这些主题。

假如现在已经安装好了Python，如果你安装的是Anaconda版本，NLTK则已经安装完毕。注意本书将使用Python 2.7，也欢迎你使用最新版本的Python，除了几个语法变化以外，大部分代码都可以在最新版的Python中复用。如果你没有安装Anaconda版本而仅仅是安装了Python，你可以打开终端或命令提示符，执行下面的命令来安装NLTK。

```
$ pip install nltk
```

该命令将会完成NLTK函数库安装，之后你就可以使用它了。然而，默认安装的NLTK不会包含本书所需要的全部组件。你可以打开Python shell输入下面的命令安装NLTK全部的组件和资源。你将会看到NLTK相关的各种资源被下载下来。部分输出如下面的代码片段所示。

```
In [1]: import nltk

In [2]: nltk.download('all')
[nltk_data] Downloading collection u'all'
[nltk_data]    |
[nltk_data]    | Downloading package abc to
[nltk_data]    |     C:\Users\DIP.DIPSLAPTOP\AppData\Roaming\nltk_data
[nltk_data]    |     ...
```

```
[nltk_data]    |   Package abc is already up-to-date!
[nltk_data]    | Downloading package alpino to
[nltk_data]    |     C:\Users\DIP.DIPSLAPTOP\AppData\Roaming\nltk_data
[nltk_data]    |     ...
```

上面的命令将下载 NLTK 所需要的全部资源。如果你不希望全部都下载，可以使用 nltk.download() 命令通过图形用户界面（GUI）选择必要的组件。必要的组件下载完毕后，你就可以开始访问文本语料库了！

1. 访问 Brown 语料库

我们已经初步介绍了布朗大学 1961 年开发的 Brown 语料库。该语料库由 500 多个来源的文本组成，并分为不同的类型。下面的代码片段将 Brown 语料库加载到内存，显示了不同的可用类型。

```
In [8]: # load the Brown Corpus
In [9]: from nltk.corpus import brown


In [10]: print 'Total Categories:', len(brown.categories())
Total Categories: 15

In [11]: print brown.categories()
[u'adventure', u'belles_lettres', u'editorial', u'fiction', u'government',
u'hobbies', u'humor', u'learned', u'lore', u'mystery', u'news', u'religion',
u'reviews', u'romance', u'science_fiction']
```

上面的输出告诉我们，该语料中共有 15 个类型，例如新闻（news）、推理小说（mystery）、传说（lore）等。下面的代码片段深入挖掘了 Brown 语料库中推理小说类型。

```
In [19]: # tokenized sentences
In [20]: brown.sents(categories='mystery')
Out[20]: [[u'There', u'were', u'thirty-eight', u'patients', u'on', u'the',
u'bus', u'the', u'morning', u'I', u'left', u'for', u'Hanover', u',',
u'most', u'of', u'them', u'disturbed', u'and', u'hallucinating', u'.'],
[u'An', u'interne', u',', u'a', u'nurse', u'and', u'two', u'attendants',
u'were', u'in', u'charge', u'of', u'us', u'.'], ...]

In [21]: # POS tagged sentences
In [22]: brown.tagged_sents(categories='mystery')
Out[22]: [[(u'There', u'EX'), (u'were', u'BED'), (u'thirty-eight', u'CD'),
(u'patients', u'NNS'), (u'on', u'IN'), (u'the', u'AT'), (u'bus', u'NN'),
(u'the', u'AT'), (u'morning', u'NN'), (u'I', u'PPSS'), (u'left', u'VBD'),
(u'for', u'IN'), (u'Hanover', u'NP'), (u',', u','), (u'most', u'AP'),
(u'of', u'IN'), (u'them', u'PPO'), (u'disturbed', u'VBN'), (u'and', u'CC'),
(u'hallucinating', u'VBG'), (u'.', u'.')], [(u'An', u'AT'), (u'interne',
u'NN'), (u',', u','), (u'a', u'AT'), (u'nurse', u'NN'), (u'and', u'CC'),
(u'two', u'CD'), (u'attendants', u'NNS'), (u'were', u'BED'), (u'in', u'IN'),
(u'charge', u'NN'), (u'of', u'IN'), (u'us', u'PPO'), (u'.', u'.')], ...]

In [28]: # get sentences in natural form
In [29]: sentences = brown.sents(categories='mystery')
In [30]: sentences = [' '.join(sentence_token) for sentence_token in
sentences]
In [31]: print sentences[0:5] # printing first 5 sentences
[u'There were thirty-eight patients on the bus the morning I left for
Hanover , most of them disturbed and hallucinating .', u'An interne , a
nurse and two attendants were in charge of us .', u"I felt lonely and
depressed as I stared out the bus window at Chicago's grim , dirty West Side
.", u'It seemed incredible , as I listened to the monotonous drone of voices
```

```
and smelled the fetid odors coming from the patients , that technically I
was a ward of the state of Illinois , going to a hospital for the mentally
ill .', u'I suddenly thought of Mary Jane Brennan , the way her pretty eyes
could flash with anger , her quiet competence , the gentleness and sweetness
that lay just beneath the surface of her defenses .']
```

从上面的片段我们可以看到，推理小说类型的书面内容，以及句子如何以标记和标注的格式存在。假设我们想看看推理小说类型中最常用的单词，我们可以使用下面的代码。请注意在词性标注中使用 NN 或是 NP 指示名词的不同形式。第 3 章将详细介绍词性标注。

```
In [81]: # get tagged words
In [82]: tagged_words = brown.tagged_words(categories='mystery')

In [83]: # get nouns from tagged words
In [84]: nouns = [(word, tag) for word, tag in tagged_words if any(noun_tag
in tag for noun_tag in ['NP', 'NN'])]

In [85]: print nouns[0:10] # prints the first 10 nouns
[(u'patients', u'NNS'), (u'bus', u'NN'), (u'morning', u'NN'), (u'Hanover',
u'NP'), (u'interne', u'NN'), (u'nurse', u'NN'), (u'attendants', u'NNS'),
(u'charge', u'NN'), (u'bus', u'NN'), (u'window', u'NN')]

In [85]: # build frequency distribution for nouns
In [86]: nouns_freq = nltk.FreqDist([word for word, tag in nouns])

In [87]: # print top 10 occuring nouns
In [88]: print nouns_freq.most_common(10)
[(u'man', 106), (u'time', 82), (u'door', 80), (u'car', 69), (u'room', 65),
(u'Mr.', 63), (u'way', 61), (u'office', 50), (u'eyes', 48), (u'hand', 46)]
```

该片段输出了出现次数最多的 10 个名词，例如 man、time、room 等。我们已经使用了一些高级的部件和技术，例如列表分析器、迭代器和元组。下一章将进一步详细地介绍这些内容，包括它们的工作原理和主要功能。目前，你需要知道的就是我们基于词性标注从其他词中过滤出名词，然后计算这些名词的出现频率，从而获得语料库中最常出现的名词。

2. 访问路透社语料库

路透社（Reuters）语料库由 90 个不同分类的 10 788 条路透社新闻组成，并分成了训练集和测试集。在机器学习术语中，训练集经常用于训练模型，测试集用于测试模型的性能。下面的代码片段给出了如何访问路透社语料库的数据。

```
In [94]: # load the Reuters Corpus
In [95]: from nltk.corpus import reuters

In [96]: print 'Total Categories:', len(reuters.categories())
Total Categories: 90
In [97]: print reuters.categories()
[u'acq', u'alum', u'barley', u'bop', u'carcass', u'castor-oil', u'cocoa',
u'coconut', u'coconut-oil', u'coffee', u'copper', u'copra-cake', u'corn',
u'cotton', u'cotton-oil', u'cpi', u'cpu', u'crude', u'dfl', u'dlr', u'dmk',
u'earn', u'fuel', u'gas', ...]

In [104]: # get sentences in housing and income categories
In [105]: sentences = reuters.sents(categories=['housing', 'income'])
In [106]: sentences = [' '.join(sentence_tokens) for sentence_tokens in
sentences]
In [107]: print sentences[0:5]  # prints the first 5 sentences
[u"YUGOSLAV ECONOMY WORSENED IN 1986 , BANK DATA SHOWS National Bank
```

```
economic data for 1986 shows that Yugoslavia ' s trade deficit grew , the
inflation rate rose , wages were sharply higher , the money supply expanded
and the value of the dinar fell .", u'The trade deficit for 1986 was 2 .
012 billion dlrs , 25 . 7 pct higher than in 1985 .', u'The trend continued
in the first three months of this year as exports dropped by 17 . 8 pct ,
in hard currency terms , to 2 . 124 billion dlrs .', u'Yugoslavia this year
started quoting trade figures in dinars based on current exchange rates ,
instead of dollars based on a fixed exchange rate of 264 . 53 dinars per
dollar .', u"Yugoslavia ' s balance of payments surplus with the convertible
currency area fell to 245 mln dlrs in 1986 from 344 mln in 1985 ."]

In [109]: # fileid based access
In [110]: print reuters.fileids(categories=['housing', 'income'])
[u'test/16118', u'test/18534', u'test/18540', u'test/18664', u'test/18665',
u'test/18672', u'test/18911', u'test/19875', u'test/20106', u'test/20116',
u'training/1035', u'training/1036', u'training/10602', ...]

In [111]: print reuters.sents(fileids=[u'test/16118', u'test/18534'])
[[u'YUGOSLAV', u'ECONOMY', u'WORSENED', u'IN', u'1986', u',', u'BANK',
u'DATA', u'SHOWS', u'National', u'Bank', u'economic', u'data', u'for',
u'1986', u'shows', u'that', u'Yugoslavia', u"'", u's', u'trade', u'deficit',
u'grew', u',', u'the', u'inflation', u'rate', u'rose', u',', u'wages',
u'were', u'sharply', u'higher', u',', u'the', u'money', u'supply',
u'expanded', u'and', u'the', u'value', u'of', u'the', u'dinar', u'fell',
u'.'], [u'The', u'trade', u'deficit', u'for', u'1986', u'was', u'2', u'.',
u'012', u'billion', u'dlrs', u',', u'25', u'.', u'7', u'pct', u'higher',
u'than', u'in', u'1985', u'.'], ...]
```

上述内容告诉了我们如何使用分类和文件标识符访问语料库数据。

3. 访问 WordNet 语料库

因为 WordNet 语料库由大量的单词和连接每个单词的语义同义词组成，它也许是最常使用的语料库之一。我们将探索 WordNet 语料库的一些基本特征，包括同义词和语料库数据访问方法。更多关于 WordNet 高级分析和覆盖情况请详见第 7 章，包括同义词、词元、上下义词、多义词，以及前面语义章节中提到的一些概念。下面的代码片段让你对如何访问 WordNet 语料库数据和同义词有个初步的了解。

```
In [113]: # load the Wordnet Corpus
In [114]: from nltk.corpus import wordnet as wn

In [127]: word = 'hike' # taking hike as our word of interest

In [128]: # get word synsets
In [129]: word_synsets = wn.synsets(word)
In [130]: print word_synsets
[Synset('hike.n.01'), Synset('rise.n.09'), Synset('raise.n.01'),
Synset('hike.v.01'), Synset('hike.v.02')]

In [132]: # get details for each synonym in synset
     ...: for synset in word_synsets:
     ...:     print 'Synset Name:', synset.name()
     ...:     print 'POS Tag:', synset.pos()
     ...:     print 'Definition:', synset.definition()
     ...:     print 'Examples:', synset.examples()
     ...:     print
     ...:
Synset Name: hike.n.01
POS Tag: n
Definition: a long walk usually for exercise or pleasure
```

```
Examples: [u'she enjoys a hike in her spare time']

Synset Name: rise.n.09
POS Tag: n
Definition: an increase in cost
Examples: [u'they asked for a 10% rise in rates']

Synset Name: raise.n.01
POS Tag: n
Definition: the amount a salary is increased
Examples: [u'he got a 3% raise', u'he got a wage hike']

Synset Name: hike.v.01
POS Tag: v
Definition: increase
Examples: [u'The landlord hiked up the rents']

Synset Name: hike.v.02
POS Tag: v
Definition: walk a long way, as for pleasure or physical exercise
Examples: [u'We were hiking in Colorado', u'hike the Rockies']
```

前面的代码片段描述了单词 hike 和它相关同义词的一个有趣的例子，这包括它的名词同义词以及具有不同的意义的动词。WordNet 使得单词间的同义词语义链接比较容易，可以很容易地检索不同词的含义和例子。前面的例子告诉我们，hike 具有“长时间的步行、工资或租金价格的上涨”的意思。请随时进行不同的实验，以找出它们的同义词、定义、例子和关系。

除了流行的语料库外，你还可以通过 nltk.corpus 模块检查和访问很多文本语料库。因此，在 Python 和 NLTK 的帮助下，你可以看到访问和使用来自任何文本语料库的数据都非常容易。

我们对文本语料库的探讨暂告一段落。下面的章节将介绍自然语言处理和文本分析相关的基础知识。

1.6 自然语言处理

在本章中，我们多次提到自然语言处理（NLP）这个词。到目前为止，你也许对自然语言处理已经有了一些理解。自然语言处理是以计算语言学为基础，属于计算机科学与工程和人工智能的一个专业领域。它主要涉及设计和构建让机器和人类所使用的自然语言间进行交互的应用和系统。这也使得自然语言处理与人机交互（Human-Computer Interaction，HCI）领域联系紧密。自然语言处理技术使得计算机可以处理和理解人类的语言并且进一步提供有用的输出。下面，我们将介绍自然语言处理的一些主要应用。

1.6.1 机器翻译

机器翻译也许是自然语言处理领域中最梦寐以求和最受欢迎的应用之一。其定义是为任意两种语言之间提供实现句法、语法、语义正确翻译的技术。机器翻译几乎是自然语言处理中第一个主要研究与开发的领域。简而言之，机器翻译就是使用机器实现自然语言的翻译。默认情况下，机器翻译的基本组成模块是将单词从一种语言简单地转换为另一种语言，但这种情况下我们忽略了如语法和短语的一致性。因此，经过一段时间的发展，进化出了更

加复杂的技术，包括将大规模的文本语料资源与统计技术和语言技术有机结合。谷歌翻译是最知名的机器翻译系统之一。在图 1-19 中，谷歌翻译成功地把“What is the fare to the airport?”从英语翻译到意大利语。

随着时间的推移，机器翻译系统正变得越来越好，当你说话或书写时可以在应用中提供实时的翻译。

图 1-19 谷歌翻译系统的机器翻译结果

1.6.2 语音识别系统

语音识别也许是自然语言处理中最难的应用。在人工智能系统中，最难的智能测试也许就是图灵测试。这个测试是测试计算机的智能程度。向计算机和人提出一个问题，如果不能说出哪个答案是由人给出的，则通过了测试。随着时间的推移，通过使用如语音合成、分析、句法分析、上下文推理等技术，这方面取得了很大的进步。但是语音识别系统主要的局限仍然存在：它们是特定域的，如果用户与系统期望的输入有一点偏差，将不能正常工作。现在，从桌面电脑到移动电话再到虚拟协助系统，语音识别系统已经在许多地方得到了应用。

1.6.3 问答系统

问答系统（Question Answering Systems，QAS）基于自然语言处理和信息检索（Information Retrieval，IR）技术构建问题回答规则。问答系统主要涉及的是建立鲁棒的、可扩展的系统，以自然语言形式为用户问题提供答案。设想一下，在外国用户通过自然语言向手机的个人助理提出一个问题，并从个人助理获得一个相似的回应。这就是研究人员和技术人员一直追求的理想的状态。像 Siri 和 Cortana 个人助理一样，这一领域已经取得了一系列成功，但由于它们只能理解整个人类自然语言关键句子和短语的一个子集，所以它们的应用范围还存在一定局限。

建立一个成功的问答系统需要有各个领域大量数据组成的知识库。对于这个知识库，问答系统将使用有效的检索系统，以自然语言的形式提供答案。创建和维护这样一个巨大的、可查询的知识库十分困难。因此，你会看到问答系统在一些诸如食品、健康、电子商务等特殊的领域出现。聊天机器人充分运用了问答系统的技术，这是一个新兴的研究趋势。

1.6.4 语境识别与消解

语境识别与消解（contextual recognition and resolution）是一个理解自然语言的广阔领域，包括基于语法和语义的推理。词义消歧是一个流行的应用，我们希望通过该应用找到给定句

子中单词的上下文意义。思考 book 这个单词。当用作名词时，它可以表示包含知识和信息的对象，当用作动词时，它也可以用来表示预订座位或桌位。通过上下文检测句子中单词的不同含义是词义消歧的主要前提，第 7 章将介绍这个艰巨的任务。

指代消解（coreference resolution）是语言学自然语言处理中正着力解决的另一个问题。根据定义，指代是在一个文本中两个或多个单词或表达指的是同一个实体。它们称为具有相同的指代。思考这句话“John just told me that he is going to the exam hall”。在这句话中，代词 he 指的就是约翰。解析这样的代词是指代消解的一部分，当我们在文本中具有多个指代时，这就成为了一个挑战。例如，“John just talked with Jim. He told me we have a surprise test tomorrow”。在这个文本中，代词“He”可以指“John”或“Jim”，因此这使得准确指出精确的指代十分困难。

1.6.5 文本摘要

文本摘要的主要目标是对一个可能包含文本、段落或句子的文本语料合理地缩减内容创建一个保留关键点信息的摘要。摘要可以通过浏览不同文档找到包含重要突出信息的关键字、关键短语和句子来实现。主要有两类文本摘要技术，分别是基于抽取的文本摘要技术和基于抽象概括的文本摘要技术。随着文本和非结构化数据的大量出现，对快速地获得有洞见的文本摘要的需求十分巨大。

文本摘要系统通常执行两类主要的操作。第一类是通用摘要，通过分析对收集到的文本尽力提供一个通用的摘要。第二类是基于查询的摘要，根据从查询中提取的特定的查询、相关的关键字和短语对语料进行过滤来提供一个查询相关的文本摘要。第 5 章将详细介绍这些内容。

1.6.6 文本分类

文本分类的主要目的是基于文档的内容识别出文档属于哪个类型或类别。这是自然语言处理和机器学习领域最受关注的应用之一，因为拥有正确的数据，就非常容易地理解内在的原则，并实现一个可以运转的文本分类系统。有监督的机器学习和无监督的机器学习都可以用来解决这个问题，有时两种方法也可以组合使用。这些技术已经帮助建立了很多成功的、实用的应用，包括垃圾邮件过滤、新闻文章分类。我们将在第 4 章构建我们自己的文本分类系统。

1.7 文本分析

如前所述，随着大量的计算资源、非结构化数据的出现，以及机器学习和统计分析技术的成功应用，文本分析很快获得了较多的关注。然而，相对于常规分析方法，文本分析仍存在一些挑战。自由流动的文本是高度非结构化的数据，很少像关系数据库中天气数据或结构化属性一样遵循任何特定的模式。因此，将标准的统计方法直接应用到非结构文本数据时是没有帮助的。本节涵盖了一些主要的文本分析概念，还介绍了文本分析的定义和范围，这将帮助你对下面章节的内容有一个广泛的认知。

文本分析也称为文本挖掘，是从文本数据中获得高质量和可操作信息和见解所遵循的方法和过程。这涉及使用自然语言处理、信息检索和机器学习技术从语法上把非结构化文本数据解析成更结构化的形式，并从这些数据中提取出对终端用户有帮助的模式和洞见。为满

足分析需要，文本分析由用于建模和从文本中提取信息的机器学习、语言学和统计技术组成，包括商务智能、探索性、描述性和预测性分析。以下是文本分析中一些主要的技术和操作：

- 文本分类。
- 文本聚类。
- 文本摘要。
- 情感分析。
- 实体抽取与识别。
- 相似性分析与关系建模。

有时候开展文本分析是一个比常规的统计分析或机器学习更加复杂的过程。在应用任何学习技术或算法之前，你必须将非结构化文本数据转换为这些算法所能接受的格式。根据定义，正文的分析通常是一个文档，我们通常通过应用各种技术将这个文档转换为词向量，词向量是一种数字数组，其值是每个词的特定权值，可以是词的频率、出现次数或是其他各种描述，我们将在第3章中讨论其中一些权值。文本经常需要进行清洗和处理，以去除噪声项和数据，这一过程称为文本预处理。

一旦拥有了机器能够阅读和理解的数据，我们就可以把相关算法应用到要着手解决的问题上。文本分析具有多方面的应用，以下内容包括了一些最流行的文本分析应用：

- 垃圾邮件检测。
- 新闻分类。
- 社交媒体分析与监视。
- 生物医疗。
- 安全智能。
- 市场营销和客户关系管理。
- 情感分析。
- 广告投放。
- 聊天机器人。
- 虚拟助理。

1.8 小结

恭喜你坚持读完了这冗长的一章！通过浏览自然语言的世界和与之相关的概念与领域，我们已经开始了 Python 文本分析之旅。你已经很好地掌握了自然语言在我们这个世界的意义以及其重要性。你也熟悉了语言方式、语言获取和语言使用的相关概念。这也是语言研究涉及的领域，研究自然语言的起源和随时间的演化方式。我们详细介绍了语言的语法和语义，包括一些便于理解和掌握的关键概念和有趣的例子。我们还介绍了语言资源，也就是文本语料库，以及一些如何使用 Python 和 NTLK 来连接和访问语料库的代码实例。最后，我们以讨论自然语言处理和文本分析不同方面的应用结束本章。下一章，我们将介绍如何使用 Python 进行文本分析。我们将涉及 Python 开发环境的安装、Python 的不同组成和如何使用 Python 进行文本处理。我们将会介绍本书中要用到的常用的库、框架和平台。

第2章
Python 语言回顾

在上一章中，我们开启了自然语言世界之旅，探讨了与之相关的几个有趣的概念和领域。现在，我们对自然语言处理（NLP）、语言学和文本分析的整体范畴有了更深入的了解。如果你还记得的话，我们已经借助于 NLTK 平台，初步尝试了运行 Python 代码来访问和使用文本语料库资源。

在本章，我们将介绍许多基础内容，涵盖 Python 的核心组件和功能以及与 NLP 和文本分析相关的一些重要库和框架。本章旨在进行 Python 回顾温习，并为开始进行文本分析提供至关重要的初始化构建模块。本书假定你对 Python 或其他编程语言有一定程度的了解。如果你是 Python 专业人员，可以速读本章，因为这里的内容是从建立 Python 开发环境开始，然后介绍 Python 的基础知识。

本章重点是讨论如何在 Python 中处理文本数据，包括与此相关的数据类型和函数。然后，我们还将讨论 Python 中的几个高级概念，包括列表分析器、生成器和修饰器，这使你在开发和编写高质量的、可复用的代码工作时能更加轻松。本章将采用比上一章更为实践性的方法，通过实例来介绍各种概念。

2.1 了解 Python

在深入熟悉 Python 生态系统以及了解与之相关的各种组件之前，我们必须要回顾 Python 背后的起源和哲学，看看随着时间的推移，它如何演变成为目前许多应用程序、服务器和系统都选择的语言。Python 是一种高级、开源、通用的编程语言，广泛用于脚本编写并跨领域使用。Python 源自于 Guido Van Rossum 的创意，在 20 世纪 80 年代后期作为 ABC 语言的继承者，两者都是由荷兰国家数学和计算机科学研究所（Centrum Wiskunde & Informatica，CWI）开发的。Python 最初设计为脚本和解释语言，而且到目前为止，它仍然是最流行的脚本语言之一。Python 使用面向对象编程（Object-Oriented Programming，OOP）和构造，你可以像使用任何其他面向对象的语言一样使用它，譬如 Java。Guido 将该语言起名为 Python 不是指蛇，而是意指电视喜剧片“蒙提·派森的飞行马戏团”（Monty Python's Flying Circus），因为他是一个超级粉丝。

如前所述，Python 是一种通用编程语言，支持多种编程范式，包括以下流行的编程范式：

- 面向对象编程。

- 函数式编程。
- 过程编程。
- 面向方面的编程。

Python 中具备很多 OOP 概念，包括类、对象、数据和方法。诸如抽象、封装、继承和多态的原则也可以使用 Python 来实现和展现。Python 中有几个高级功能，包括迭代器、生成器、列表分析器、lambda 表达式和几个模块（如 `itertools` 和 `functools`），它们提供了遵循函数编程范式编写代码的能力。

Python 的设计思想：简单而美丽的程序代码应该遵循更加优雅和易于使用的风格，而不是过早优化和编写难以解释的代码。Python 标准库功能强大，具有从低级硬件接口到处理文件和文本数据的各种功能和特性。易于扩展和集成使得在开发 Python 时，可以轻松地实现与现有应用程序集成——甚至可以创建*应用程序接口*（Application Programming Interface，API），以提供与其他应用程序和工具的接口。

Python 提供了很多优势和特色。下面是一些主要的特色。

- *友好易学*：Python 编程语言非常容易理解和学习。学校正在开始将 Python 选为教授孩子们编程的首选语言。它的学习曲线不是很陡峭，你可以在 Python 中做很多有趣的事情，从构建游戏到事务自动化（如阅读和发送电子邮件）。（事实上，在https://automatetheboringstuff.com 上，有一本专门用于“automating the boring stuff（自动化无聊东西）”的图书和网址。）Python 还有一个蓬勃发展和有用的开发人员社区，它确保在互联网上有大量有用的资源和文档。社区还在世界各地组织了各种培训班和研讨会。
- *高级抽象*：Python 是一种高级语言（High-Level Language，HLL），通过高级抽象来消除编写低层代码时所需的大量繁重的编码工作。Python 重点关注代码的简洁性和可扩展性，你可以使用比其他传统编译语言（如 C++ 和 C）更少的代码行数来执行各种简单或复杂的操作。
- *提高效率*：与 Java、C++ 和 C 等其他语言相比，Python 通过减少开发、运行、调试、部署和维护大型代码库所需的时间来提高工作效率。超过 100 行的大程序可以通过移植到 Python，实现减少到平均 20 行或更少的程度。高级抽象有助于开发人员专注于手头要解决的问题，而不必担心特定语言的细微差别。Python 也避开了编译和链接的障碍。因此，Python 通常是首选的编程语言，特别是快速原型开发，这对于在很短时间内解决重要问题显得至关重要。
- *完整的生态系统*：Python 的主要优点之一是它是一种多用途的编程语言，可以用于任何事情！从 Web 应用程序到智能系统，Python 支持各种各样的应用程序和系统。稍后我们将在本章讨论一些相关内容。除了作为多用途语言，使用 Python 开发的，以及为 Python 而开发的各种框架、库和平台，围绕 Python 形成了一个完整的、稳健的生态系统。这些库通过最少的代码为我们提供各种各样的能力和功能来执行各种任务，使得生活更加轻松方便。有一些库的例子，包括处理数据库、文本数据、机器学习、信号处理、图像处理、深度学习、人工智能等。
- *开源*：作为开源语言，Python 正在积极进行开发和持续改进、优化和新功能的更新。现在，Python 软件基金会（Python Software Foundation，PSF）拥有所有与 Python 相关

的知识产权（Intellectual Property，IP），并管理所有许可证相关的事务。开放源代码提升了 Python 生态系统，几乎所有的库也都是开源的，任何人都可以分享、贡献、提出改进和反馈意见。这有助于培育技术人员、工程师、研究人员和开发人员之间的健康合作。

- 易于移植、集成和部署：Python 支持所有的主流操作系统（OS），包括 Linux、Windows和 MacOS。在一个操作系统中编写的代码可以通过简单复制代码文件轻松地移植到另一个操作系统中，而且可无缝地工作。Python 也可以轻松地与现有应用程序进行集成和扩展，可以使用套接字（socket）、网络和端口与各种 API 和设备实现交互。可以用 Python 调用其他语言的程序代码，并且提供一些 Python 绑定用于从其他语言调用 Python 代码。这有助于在必要时能轻松地集成 Python 代码。最重要的优点是，无论你的代码库多么复杂，开发 Python 代码并进行部署都是非常容易的。你要遵循正确的持续集成（Continuous Integration，CI）流程，并合理地管理你的 Python 代码库，部署工作通常包含在生产环境中更新你的最新代码并启动必要的流程。Python 可以在最短的时间内非常容易地获得正确的工作代码，而这对于其他语言而言通常很难做到。

所有这些特色以及 Python 过去数年在各领域的迅速发展应用，使得 Python 广受欢迎。如果在编写代码时不遵循简单、优雅和极简主义的 Python 正确原则，则代码被称为不是“pythonic”。编写好的 Python 代码有一种众人熟知的风格和约定，许多文章和书籍教导了如何编写 pythonic 式的代码。在 Python 社区中，活跃用户和开发人员称其为 Pythonistas、Pythoneers以及其他更多有趣的名称。因为 Python 及其整个生态系统一直在积极地改进和发展，繁荣的 Python 社区使得这门语言更加鼓舞人心。

2.1.1 Python 之禅

你或许想知道 Python 之禅究竟是什么，在你对 Python 熟悉之后，这是你应首先了解的事情之一。Python 之美在于其简洁和优雅的风格。在程序设计中，Python 有 20 条有影响力的指导原则或格言。资深的 Python 开拓者（Pythoneer）Tim Peters 在 1999 年记录了其中 19 条，可以通过 https://hg.python.org/peps/file/tip/pep-0020.txt 访问它们，它们已作为 Python 增强建议（Python Enhancement Proposals，PEP）第 20 号（PEP 20）的一部分。如果你已经安装了 Python，这些原则部分内容可以随时通过在 Python 或 IPython shell 中运行以下代码来访问 Python 之禅：

```
In [5]: import this
The Zen of Python, by Tim Peters
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
```

```
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

上述输出的显示形成 Python 之禅的 19 条原则，其作为复活节彩蛋被包括在 Python 语言本身中。这些原则是用简单的英语编写的，即使你以前没有写过代码，很多都是很一目了然的，其中很多包含着圈内笑料！Python 专注于编写可读的、简洁的代码。Python 还旨在确保你能专注于错误处理和实现易于解释和理解的代码。本书作者最想要请你记住的一个原则是“简单胜过复杂（Simple is better than complex）”，它不仅适用于 Python，而且适用于解决世界上的很多事情。只要你知道在做什么，有时一个简单的方法比一个更复杂的会更好，因为它可以帮助你避免过度复杂的状况。

2.1.2 应用：何时使用 Python

Python 作为通用的多用途编程语言，能为不同领域构建应用程序和系统，并解决各种现实世界中的问题。Python 自带有一个标准库，它包括大量对于解决各种问题有用的库和模块。除了标准库，互联网上还有数以千计的第三方库随时可用，它们用于鼓励开源和积极开发。官方存储库是 Python 的程序包索引（Python Package Index，PyPI），用于托管第三方库以及 Python 增强开发的工具。你可以访问 https://pypi. python. org 并查看各种程序包。目前，可以安装和使用的程序包超过 80 000 个。

Python 可以用来解决大量问题，下面列出一些最受欢迎的应用领域。

- *脚本*（scripting）：Python 被称为脚本语言。它可以用于执行许多任务，例如，与网络、硬件的交互，处理文件和数据库，执行操作系统操作，以及接收和发送电子邮件。Python 也广泛用于服务器端脚本，甚至用于开发服务网页的整个 Web 服务器。许多 Python 脚本以 ad-hoc 方式用于自动化操作，譬如网络套接字通信、处理电子邮件、解析和提取网页、通过 FTP 进行文件共享和传输、通过不同协议进行通信以及其他多种操作。
- *Web 开发*（Web development）：有很多广泛用于 Web 开发的强大且稳定的 Python 框架，包括 Django、Flask、Web2Py 和 Pyramid。你可以使用它们来开发完整的企业 Web 应用程序，Python 支持各种架构风格，如 RESTful API 和 MVC 架构。Python 还提供数据库交互的 ORM 支持，并在其上使用 OOP。Python 甚至还有像 Kivy 这样的框架，可以支持跨平台开发，用于在 iOS、Android、Windows 和 OS X 等多个平台上开发应用程序。Python 也用于在 IronPython 中开发具有 Silverlight 框架支持的富互联网应用程序（Rich Internet Application，RIA），IronPython 是一个受欢迎的Microsoft. NET 框架和 pyjs 完美集成的 Python 版本，RIA 开发架构支持 Python 到 JavaScript 的编译器和 AJAX 框架。
- *图形用户界面*（Graphical User Interface，GUI）：使用 Python 可以轻松构建大量具有 GUI 的桌面应用程序。Tkinter、PyQt、PyGTK 和 wxPython 之类的库和 API 允许开发人员通过简单/复杂的接口开发基于 GUI 的应用程序。多样化的框架使得开发人员能够为不同的操作系统和平台开发基于 GUI 的应用程序。

- 系统编程（systems programming）：作为一门高级语言，Python 具有与低级 OS 服务和协议交互的大量接口，并且这些服务之上的抽象使得开发人员能够编写强大而可移植的系统监视和管理工具。我们可以使用 Python 执行操作系统操作，包括创建、处理、搜索、删除和管理文件和目录。Python 标准库（Python Standard Library，PSL）提供操作系统和 POSIX 绑定，可用于处理文件、多线程、多处理、环境变量、控制套接字、管道和进程。这也增强了 Python 脚本编写能力，以最少的工作和代码行来执行系统级的管理任务。
- 数据库编程（database programming）：Python 用于连接和访问来自不同类型数据库的数据，无论是 SQL 还是 NoSQL。MySQL、MSSQL、MongoDB、Oracle、PostgreSQL 和 SQLite 之类的数据库都有 API 和连接器。事实上，SQLite 是一个轻量级的关系数据库，现在它作为 Python 标准发布版的一部分。SQLAlchemy 和 SQLObject 这类的热门库提供了访问各种关系数据库的接口，并且还具备 ORM 组件来帮助在关系表之上实现 OOP 风格的类和对象。
- 科学计算（scientific computing）：Python 在数值和科学计算等领域展现出多用途的禀赋。你可以使用 Python 执行简单和复杂的数学运算，包括代数和微积分。诸如 SciPy 和 NumPy 这样的库能够帮助研究人员、科学家和开发人员利用高度优化的函数和接口进行数值和科学编程。这些库也是在机器学习等各个领域开发复杂算法的基础。
- 机器学习（machine learning）：Python 被视为当今最流行的机器学习语言之一。Python有一套广泛的库和框架，如 `scikit-learn`、`h2o`、`tensorflow`、`theano`，甚至还有 `numpy` 和 `scipy` 这样的核心库，不仅能够实现机器学习算法，而且还使用它们来解决现实世界中的高级分析问题。
- 文本分析（text analytics）：如上所述，Python 可以很好地处理文本数据，这方面产生了几个流行的库用来进行 NLP、信息检索和文本分析，如 `nltk`、`gensim` 和 `pattern`。你还可以应用标准机器学习算法来解决与文本分析相关的问题。Python 生态系统中易于使用的程序包可以减少开发的时间和工作量。我们将在本书中探讨其中的几个库。

尽管上述列表看起来已经非常强大了，但这些只不过是用 Python 可能解决的问题领域中的冰山一角。它还广泛应用于人工智能（AI）、游戏开发、机器人、物联网（IoT）、计算机视觉、多媒体处理以及网络和系统监控等多个领域，上面仅列举几例。想要阅读 Python 在诸如艺术、科学、计算机科学、教育等不同领域所取得的一些广泛成功案例，热心的程序员和研究人员可以访问 www. python. org/about/success/。要了解使用 Python 开发的各种流行应用程序，请参阅 www. python. org/about/apps/和 https://wiki. python. org/moin/Applications，你一定会找到你已经使用的一些应用程序——其中一些是不可或缺的。

2.1.3 缺点：何时不用 Python

到目前为止，作者一直是 Python 的吹鼓手，但或许你想知道它有什么缺点呢？像任何工具或语言一样，Python 有它的优缺点。是的，即使是 Python 也会有一些缺点，在这里我们将重点介绍其中的一些，以便你在 Python 中开发和编写代码时了解它们。

- 执行速度性能：性能是一个非常关键的方面，它可以包涵各种涵义，所以需要准确界定我们要谈论的范围是什么，这里性能就是指执行速度。因为 Python 并不是一个完全编译的语言，因此它总是比完全编译的低级编程语言（如 C 和 C++）慢些。你有几种方法可以优化代码，包括多线程和多处理。你也可以使用静态类型和 Python 的 C 语言扩展（称为Cython）。你还可以考虑使用 PyPy，它比普通 Python 快得多，因为它使用即时（Just-In-Time，JIT）编译器（参见 http://pypy.org），但是如果编写优化的代码，你通常可以在 Python 中很好地开发应用程序，而不需要依赖其他语言。请记住问题通常不在于工具，而是你编写的代码——所有开发人员和工程师都会随着时间和经验而意识到这一点。
- 全局解释器锁（Global Interpreter Lock，GIL）：GIL 是一个互斥锁，用于多个编程语言解释器，如 Python 和 Ruby。使用 GIL 的解释器只允许单个线程依次有效执行，即使它在多核处理器上运行时，从而有效限制了多线程实现的并行性，这取决于进程是 I/O 绑定还是 CPU 绑定，以及在解释器之外有多少个调用。
- 版本不兼容：如果你一直在跟踪 Python 的新闻，会知道 Python 在 2.7.x 版本之上发布了 3.x 版本，由于它在许多方面都是向后不兼容的，这确实会带来一大堆亟待解决的复杂问题。在 Python 2.7.x 中构建的几个主要库和程序包会在用户不经意更新 Python 版本时开始中断。因此，由于遗留代码问题，一大批企业和开发者社区仍然使用 Python 2.7.x，因为这些程序包和库的新版本从未建成。代码弃用和版本更改是系统崩溃中的一些最重要的因素。

上述这些问题中许多并不是 Python 特有的，也适用于其他语言，所以不要仅仅因为前面说的几点就不再鼓励你使用 Python 了——但是，你在编写代码和构建系统时一定要记住它们。

2.1.4 Python 实现和版本

Python 有几种不同版本的实现方式，因为它们正在积极开发中，版本会定期发布。本节讨论 Python 的实现和版本以及它们的重要性，这些应该让你了解你可能想要哪种 Python 来满足开发需求。目前，有四种产品完备的、强大和稳定的主流 Python 实现。

- CPython 是常规的老版本 Python，也是我们通常所称的 Python。它既是编译器也是解释器，有自己的一套全部用标准 C 语言编写的标准程序包和模块。该版本可以直接用于所有流行的当前平台。大多数的 Python 第三方程序包和库与此版本兼容。
- PyPy 是 Python 实现的一个更快实现，它使用 JIT 编译器来使代码运行速度比 CPython 实现的速度更快——有时提供达 10～100 倍的加速。PyPy 还有更高的内存效率，支持 greenlet 和 stackless 从而具有高并行性和并发性。
- Jython 是 Java 平台的 Python 实现，它支持 Java 虚拟机（Java Virtual Machine，JVM），适用于任何版本的 Java（版本最好是 7 以上）。通过使用 Jython，你可以用所有类型的Java库、包和框架来编写代码。当你比较了解 Java 语法和 Java 中广泛使用的 OOP 原则（如类、对象和接口）时，它的效果最好。
- IronPython 是流行的 Microsoft.NET 框架的 Python 实现，也称为通用语言运行时（Common Language Runtime，CLR）。你可以使用 IronPython 中的所有 Microsoft CLR 库和框架，即使你实质上并不需要在 C#中编写代码，更多地了解 C#的语法和构造有助于有效地使用 IronPython。

我们首先建议你使用默认的 Python 版本，即 CPython 实现，只有当你真的有兴趣与其他语言（如 C#和 Java）进行交互并需要在代码库中使用它们时，才需要去尝试其他版本。

Python 有两个主流版本：2. x 系列和 3. x 系列，其中 x 是一个数字。Python 2. 7 是 2010 年发布的 2. x 系列的最后一个主版本。从那以后，后续的版本包括错误修复和性能改进但没有新的功能。最新版本是于 2016 年 6 月发布的 Python 2. 7. 12。3. x 系列是从 Python 3. 0 开始，与 Python 2. x 系列相比，它引入了许多向后不兼容的更改。每一个发布的版本 3 系列不仅具有错误修复和改进，还引入了新功能，例如最近发布的 AsyncIO 模块。在撰写本书时，在 2016 年 6 月发布的 Python 3. 5. 2 是 3. x 系列中的最新版本。

关于使用何种版本的 Python 尚有许多争论。我们稍后会讨论其中一些，但最好的方法是你要考虑解决的问题以及需要使用的完整软件生态系统，从库、依赖关系和架构开始到实现和部署——同时也要考虑重用现有的旧代码库。

2. 2 安装和设置

既然你已经熟悉了 Python，了解到很多关于这门语言以及它的功能、实现和版本的信息，那么我们将讨论在本书中使用的 Python 版本，并讨论关于如何设置开发环境、处理包管理和虚拟环境的详细信息。本节将为你的后续工作奠定一个良好的开端，并提供我们将在本书中介绍的各种实际操作示例。

2. 2. 1 用哪个 Python 版本

如前所述，两个主要的 Python 版本是 2. x 系列和 3. x 系列。它们是非常相似的，但是在 3. x 版本中出现了几个向后不兼容的变化，这导致在使用 2. x 的人和使用 3. x 的人之间产生了极大的不便。PyPI 上的大多数遗留代码和大部分的 Python 包都是在 Python 2. 7. x 中开发的，因为所需的工作量不会小，许多程序包的所有者没有时间或意愿将所有代码库移植到 Python3. x。下面是 3. x 系列中的一些变化：

- 默认情况下，所有文本字符串均为 Unicode。
- `print` 和 `exec` 是函数，不再是语句。
- `range()`返回一个内存高效的 iterable，而不再是一个列表。
- 修改了类的风格。
- 基于惯例和类型冲突进行了库和名称变更。

要了解更多 Python 3. 0 所引入的变更情况，请查看 https://docs. python. org/3/whatsnew/3. 0. html，上面的官方文档列出了变更内容。如果你将代码从 Python 2 移植到 Python 3，关于哪些变化会破坏你的代码，这个链接应该会给你一个很好的解读。

对于选择哪个版本的问题，并没有绝对的答案。它纯粹取决于你正在试图解决的问题、现有代码和具有的基础设施、将来如何维护代码以及所有必要的依赖关系。如果你将开始一个全新项目，也非常清楚你并不需要任何仅依赖于 Python 2. x 的外部程序包和库，那么你可以使用 Python 3. x 并启动系统开发。但是，如果你有很多依赖于外部的程序包，并且可能会破坏 Python 3. x 或仅仅适用于 Python 2. x，那么你就别无选择，只能坚持使用 Python 2. x 了。除此之外，通常你必须处理很久以前的遗留代码，特别是大型公司和组织中会存有巨型的代

码库。在这种情况下，将整个代码移植到 Python 3. x 将会浪费力气——这有点像重新开发，因为你不会错过使用 Python 2. x 的主要功能和性能，实际上，你甚至没有意识到你可能会最终破坏现有的代码和功能。最后，这是一个留给读者的决定，你必须仔细考虑所有的场景。

仅是为了安全起见，我们将在本书中使用 Python 2. 7. 11，因为它是经过所有主要企业测试和验证的 Python 版本。当然，也非常欢迎你使用 Python 3. x——算法和技术将是相同的，但是你可能需要考虑一些版本变更情况，例如 `print` 语句在 Python 3. x 中作为函数使用，等等。

2. 2. 2 用哪个操作系统

目前有几种流行的操作系统，每个人都有自己的使用偏好。无疑，Python 的美丽之处在于可以在任何操作系统上无缝地运行。最受欢迎的三个操作系统包括：

- Windows。
- Linux。
- OS X（现称为 macOS）。

可以选择任何你希望使用的操作系统，并将其与本书中的示例一起使用。我们将在本书中使用 Windows 作为主要操作系统。这本书针对的是职场上的专业人士和从业者，其中大多数人所在的企业环境通常使用企业版的 Windows。此外，几个 Python 外部软件包在基于 UNIX 的操作系统（如 Linux 和 macOS）上能很容易地安装。但是，有时在 Windows 上安装它们会出现重大问题，所以我们想强调说明这样的情形，并确保解决它们，以便让你在这里执行任何代码片段和案例时会变得容易。再一次欢迎你按照本书中的示例使用你所选择操作系统。

2. 2. 3 集成开发环境

集成开发环境（Integrated Development Environment，IDE）是软件产品，通过提供编写、管理和执行代码所需的一整套工具和功能，使得开发人员能够高效地工作。IDE 的常见组成包括源代码编辑器、调试器、编译器、解释器、重构和构建工具。它们还具有其他功能，例如代码完成、语法高亮显示、错误突出显示和检查、对象和变量浏览器。可以使用 IDE 来管理整个代码库——这比在简单的文本编辑器中编写代码要好得多，但也需要更多的时间。也就是说，经验丰富的开发人员经常使用简单的纯文本编辑器来编写代码，尤其是如果他们在服务器环境中工作的话。你可以在 https://wiki. python. org/moin/IntegratedDevelopmentEnvironments 上找到专门用于 Python 的 IDE 列表。

我们将使用 Spyder IDE，通过与 Anaconda Python 发行版一起使用来编写和执行代码。

2. 2. 4 环境设置

本节将介绍如何以最少的工作量和主要组件来设置 Python 环境的详细信息。

首先，请访问 Python 官方网站，并从 www. python. org/downloads/下载 Python 2. 7. 11。或者从 Continuum Analytics（www. continuum. io/downloads）下载一个完整的 Python 发行版，它包含 700 多个包，称为 Anaconda Python 发行版，专门用于数据科学和分析。该软件包具有很多优点，特别是对于 Windows 用户来说，因为安装某些软件包如 `numpy` 和 `scipy` 有时会出

现问题。你可以访问 https://docs.continuum.io/anaconda/index 来获取有关 Anaconda 和 Continuum Analytics 的更多信息。Anaconda 提供了 conda——一个开源软件包和环境管理系统，以及 Spyder（Scientific Python Development Environment）——一个用于编写和执行代码的 IDE。

对于其他操作系统选项，请查看网站上的相关说明。

下载完 Python 之后，启动可执行文件，按照屏幕上的说明在每一步单击“下一步”按钮。但是，在开始安装之前，请记得要勾选图 2-1 所示的两个选项。

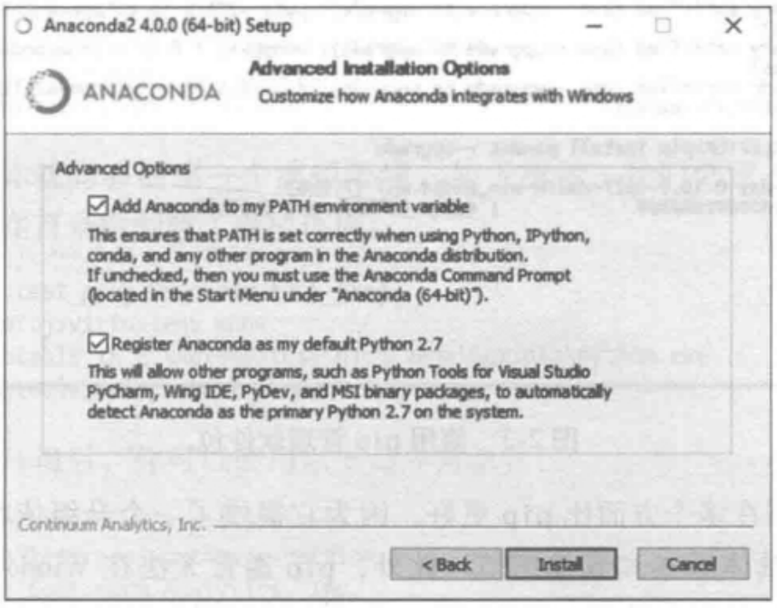


图 2-1 安装 Anaconda Python 发行版

完成安装之后，可以通过双击图标启动 Spyder，或者通过命令提示符启动 Python 或 IPython shell。Spyder 提供了一个完整的 IDE，可以在常规 Python 和 IPython shell 中编写和执行代码。图 2-2 显示如何从命令提示符运行 IPython。

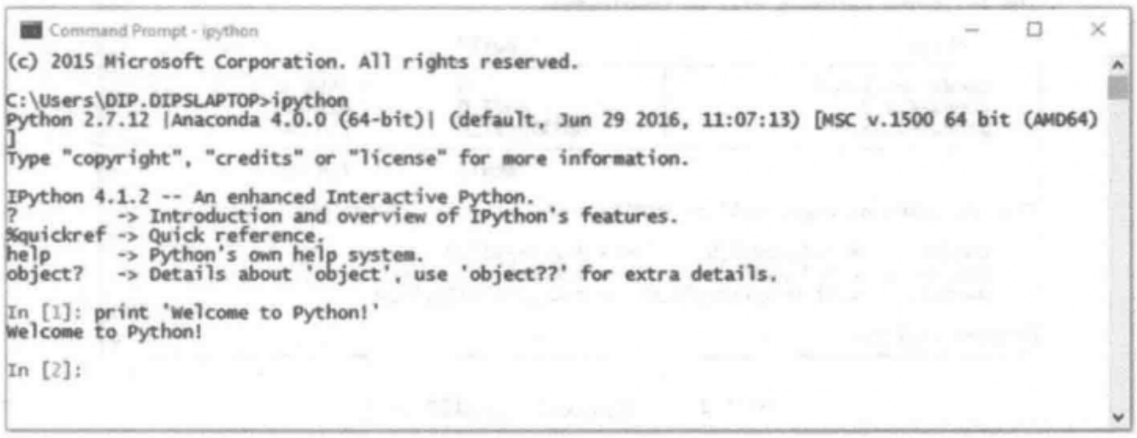


图 2-2 从命令提示符启动 IPython

图 2-2 展示了屏幕输出一个普通句子“Welcome to Python!”，只是为了告诉你 Python 正确安装并且正常工作。输入和输出执行历史记录保存在名为 `In` 和 `Out` 的变量中，如图中所示的提示编号 `In [1]`。IPython 拥有了很多优点，包括代码完成、内联执行和绘图以及代码段交互运行。我们将在 IPython shell 中运行大部分代码段，就像第 1 章中的示例一样。

现在你已经安装好了 Anaconda，可以开始运行本书中的代码示例了。在前往下一节之前，我们要简单介绍一下软件包管理。你可以使用 pip 或 conda 命令来安装、卸载和升级软件包。图 2-3 所示的 shell 命令描述了通过 pip 来安装 pandas 库。因为我们已经安装了该库，所以可以使用 --upgrade 标志。

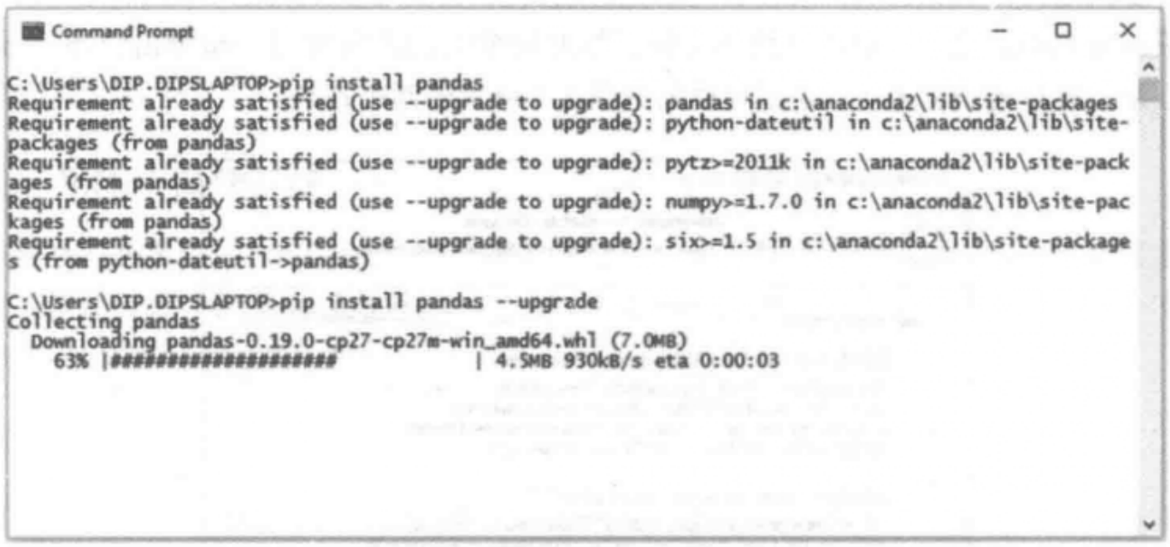


图 2-3 使用 pip 管理软件包

conda 包管理器在多个方面比 pip 更好，因为它提供了一个升级依赖关系的完整视图，以及安装过程中的具体版本和其他细节。此外，pip 通常无法在 Windows 中安装一些软件包，但 conda 在安装过程中没有这样的问题。图 2-4 描述了如何使用 conda 来管理包。

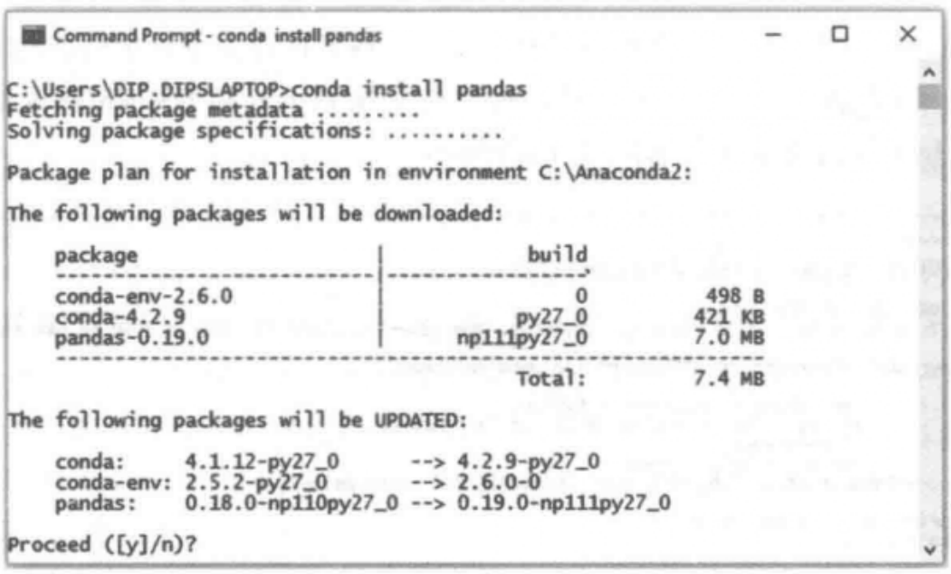


图 2-4 使用 conda 管理软件包

现在，你更加熟悉了如何在 Python 中安装外部程序包和库。后续当我们安装一些专门为文本分析而构建的库时，这将很有用。现在你应该完成了 Python 环境的设置，为执行代码做好了准备。在我们深入了解 Python 中的基本和高级概念之前，下面先来讨论虚拟环境。

2.2.5 虚拟环境

虚拟环境（virtual environment，或简称 venv）是一个完整的独立 Python 环境，它有自己

的 Python 解释器、库、模块和脚本。该环境是与其他虚拟环境和默认的系统级 Python 环境相隔离的独立环境。当你有多个项目或代码库并对相同软件包或库的不同版本具有依赖关系时，虚拟环境非常有用。例如，如果项目 TextApp1 依赖于 nltk 2.0，而另一个项目 TextApp2 依赖于 nltk 3.0，那么在同一个系统上运行这两个项目是不可能的。因此，需要提供环境完全隔离的虚拟环境，并且能够根据需要进行激活和停用。

要设置虚拟环境，你需要按如下所示安装 virtualenv 软件包：

```
E:\Apress>pip install virtualenv
Collecting virtualenv
  Downloading virtualenv-15.0.2-py2.py3-none-any.whl (1.8MB)
    100% |################################| 1.8MB 290kB/s
Installing collected packages: virtualenv
Successfully installed virtualenv-15.0.2
```

安装完成后，你就能够创建一个虚拟环境，如下所示，我们创建一个名为 test_proj 的新项目目录，并在目录中创建了虚拟环境：

```
E:\Apress>mkdir test_proj && chdir test_proj
E:\Apress\test_proj>virtualenv venv
New python executable in E:\Apress\test_proj\venv\Scripts\python.exe
Installing setuptools, pip, wheel...done.
```

成功安装虚拟环境后，你可以使用以下命令激活它：

```
E:\Apress\test_proj>venv\Scripts\activate
(venv) E:\Apress\test_proj>python --version
Python 2.7.11 :: Continuum Analytics, Inc.
```

对于其他操作系统平台，你可能需要使用命令 source venv/bin/activate 来激活虚拟环境。

一旦虚拟环境处于活动状态，你可以看到（venv）符号，如上面输出的代码所示，并且你安装的任何新软件包将被放置在与全局系统 Python 安装完全隔离的 venv 文件夹中。下面的代码说明了全局系统 Python 和虚拟环境 Python 中不同版本 pandas 软件包的差异：

```
C:\Users\DIP.DIPSLAPTOP>echo 'This is Global System Python'
'This is Global System Python'
C:\Users\DIP.DIPSLAPTOP>pip freeze | grep pandas
pandas==0.18.0

(venv) E:\Apress\test_proj>echo 'This is VirtualEnv Python'
'This is VirtualEnv Python'
(venv) E:\Apress\test_proj>pip install pandas
Collecting pandas
  Downloading pandas-0.18.1-cp27-cp27m-win_amd64.whl (6.2MB)
    100% |################################| 6.2MB 142kB/s
Installing collected packages: pandas
Successfully installed pandas-0.18.1
(venv) E:\Apress\test_proj>pip freeze | grep pandas
pandas==0.18.1
```

你可以从上面代码中看到，pandas 包在同一台机器中有不同的版本：全局 Python 为 0.18.0，虚拟环境 Python 为 0.18.1。因此，这些隔离的虚拟环境可以在同一系统上无缝运行。

在虚拟环境中完成工作后，你可以按如下方式再次停用它：

```
(venv) E:\Apress\test_proj>venv\Scripts\deactivate
E:\Apress\test_proj>
```

这让你回到了系统默认的 Python 解释器，它带有所有已安装的库。这让我们很好地了解到虚拟环境的效用和优点，一旦你开始研究几个项目，一定要考虑使用它。要了解有关虚拟环境的更多信息，请访问 http://docs.python-guide.org/en/latest/dev/virtualenvs/，这是 virtualenv 软件包的官方文档。

我们将结束安装和设置工作，接下来将使用实例来研究 Python 的概念、构造、语法和语义。

2.3 Python 句法和结构

在编写代码时你应该记住，Python 代码有一个清晰的分层语法。任何大型 Python 应用程序或系统都由多个模块构建，这些模块本身由 Python 语句组成。每条语句就像系统的命令或指令，指挥它应该执行什么操作，这些语句由表达式和对象组成。Python 中的所有东西都是对象——包括函数、数据结构、类型、类等。图 2-5 显示了这种层次结构。

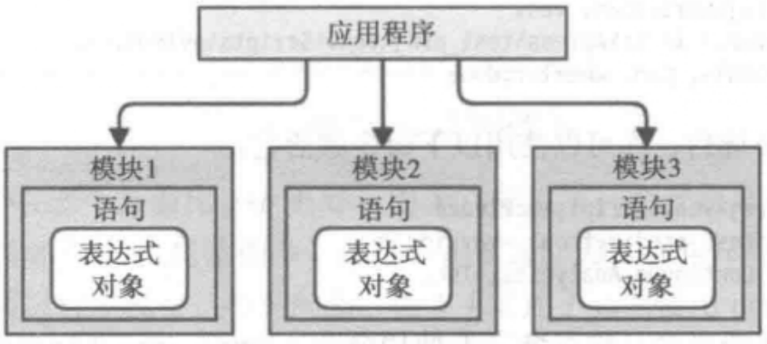


图 2-5 Python 程序层次结构

基本语句由对象、表达式组成，表达式通常使用对象并对其进行处理和执行操作。

对象可以是从简单数据类型和结构到复杂对象的任何东西，包括函数和具有自己特定作用的保留字。Python 有大约 37 个关键字或保留字，它们有自己设定的作用和功能。表 2-1 详细列出了每一个关键字及其示例，这些示例在你编程时会很方便有用。

表 2-1 Python 保留字

SI 序号	关键字	说　明	示　例
1	and	逻辑 AND 运算符	(5==5 and 1==2) == False
2	as	用作某些对象/引用的同义词	with open ('file.txt') as f
3	assert	断言/检查表达式是否为 True	assert 1==2, " Not Equal"
4	async	函数声明为异步（协同）	async def get_ data ():
5	await	用于调用协同程序	return await get_ data ()
6	break	跳出执行循环	while True: break
7	class	创建一个类（OOP）	class ClassifyText (object):
8	continue	继续循环的下一个迭代	while True: if a==1: continue
9	def	定义一个函数	def add (a, b): return a+b
10	del	删除引用	del arr
11	elif	else-if 条件语句	if num==1: print '1'elif num==2: print '2'

（续）

SI 序号	关键字	说　明	示　例
12	else	else 条件语句	if num = =1: print '1'else: print ' not 1'
13	except	捕捉异常	except ValueError, e: print e
14	exec	动态执行代码	exec 'print " Hello Python" '
15	False	布尔值假	False = = 0
16	finally	在 try-except 之后的最后执行语句	finally: print 'end of exception'
17	for	for 循环	for num in arr: print num
18	from	从模块导入特定组件	from nltk. corpus import brown
19	global	声明变量为全局变量	global var
20	if	if 条件	if num = =1: print '1'
21	import	导入现有模块	import numpy
22	in	对现有对象检查或循环	for num in arr \ if x in y
23	is	用于检查是否相等	type ('a') is str
24	lambda	创建一个匿名函数	lambda a: a* * a
25	None	表示无值或空值	num = None
26	nonlocal	在函数中修改外部的但非全局范围的变量值	nonlocal var
27	not	逻辑 NOT 运算符	not 1 = = 2
28	or	逻辑 OR 运算符	1 or 2 = = 1
29	pass	用作指示空块的占位符	if a = = 1: pass
30	print	打印字符串或其他对象	print 'Hello World! '
31	raise	引发异常	raise Exception ('overflow')
32	return	从函数退出后返回对象	return a, b
33	try	执行 try 语句的代码块，如果发生异常，执行 except 语句	try: read_ file () except Exception, e: print e
34	while	while 循环	while True: print value
35	with	对表达式中的对象执行操作	with open ('file. txt') as f: data = f. read ()
36	yield	生成器功能，暂停并返回给调用者	def generate_ func (arr): for num in arr: yieldnum + 1

表 2-1 显示了所有在语句中使用的 Python 关键字。但是，有几点事项需要记住。关键字 async 和 await 仅在 Python 3. 5. x 之后的版本中才可用。关键字 exec 和 print 仅用在Python 2. x 系列中的语句——从 Python 3. x 开始，它们是函数。关键字列表中的 False、True 和 nonlocal 是从 Python 3. x 系列开始引入的。

通常，Python 语句指示解释器在执行语句时应该做什么。一串语句通常形成一个逻辑的代码块。包括函数、循环以及条件的各种构造有助于分隔和执行基于用户决策逻辑和设计的代码块。Python 也非常关注代码的可读性——因此缩进是 Python 代码的重要组成部分。默认情况下，Python 不使用像分号的标点符号来表示语句结束。它还使用制表符或空格来显示和分隔特定的代码块，而不是像 C、C++ 、Java 等语言中使用传统的括号或关键字。Python 接受空格和制表符作为缩进，通常的规范是一个制表符或四个空格来表示每个特定的代码块。

未缩进的代码总是会引起语法错误，所以任何人在编写 Python 代码时都必须格外小心，要注意代码的格式和缩进。

Python 程序通常围绕前面提到的层次结构构建。每个模块通常都是一个带有_init_.py 文件的目录，该文件使目录成为一个包，它还可能有多个模块，每个模块都是一个单独的 Python（.py）文件。每个模块通常都有类和对象，例如其他模块和代码调用的函数。所有互连的模块最终构成了一个完整的 Python 程序、应用或系统。通常，你可以通过在 Python（.py）文件中编写必要的代码来启动任意的项目，并在项目因为添加更多组件而增大时要使其变得更模块化。

2.4 数据结构和类型

Python 有几种数据类型，其中许多数据类型被用作处理数据的数据结构。所有数据类型都是从 Python 中的默认对象数据类型派生出来的。该对象数据类型是 Python 用于管理和处理数据的抽象概念。所有的代码和数据都是由对象及对象之间的关系来存储和处理的。每个对象都有三个属性以将其与其他对象进行区分。

- 身份（identity）：该属性是唯一的，一旦对象创建就不会改变，通常由对象的内存地址表示。
- 类型（type）：该属性决定对象的类型（通常为数据类型，这又是基础对象类型的子类）。
- 值（value）：该属性表示对象存储的实际值。

假设一个变量是数据类型之一的字符串。要查看操作中的三个属性，你可以使用以下代码段中所描述的函数：

```
In [46]: new_string = "This is a String"  # storing a string

In [47]: id(new_string)  # shows the object identifier (address)
Out[47]: 243047144L

In [48]: type(new_string)  # shows the object type
Out[48]: str

In [49]: new_string  # shows the object value
Out[49]: 'This is a String'
```

Python 有几种数据类型，包括几种核心数据类型和复杂的数据类型（如函数和类）。在本节中，我们将讨论 Python 的核心数据类型，包括一些广泛作为数据结构处理数据的数据类型。这些核心数据类型如下：

- 数值（numeric）。
- 字符串（string）。
- 列表（list）。
- 集合（set）。
- 字典（dictionary）。
- 元组（tuple）。
- 文件（file）。
- 杂项（miscellaneou）。

虽然这不是一份详尽的列表，但是你90%以上的时间将使用这些对象编写 Python 语句。下面让我们更详细地逐一讨论它们，以更好地理解它们的属性和行为。

2.4.1 数值类型

数值数据类型（numeric data type）或许是 Python 中最常见和最基本的数据类型。各种应用程序最终都会以某种形式处理和使用数字。主要有三种数值类型：整数、浮点数和复数。整数是在小数点后没有小数部分或尾数的数字。整数可以如下表示和操作：

```
In [52]: # representing integers and operations on them
In [53]: num = 123

In [54]: type(num)
Out[54]: int

In [55]: num + 1000  # addition
Out[55]: 1123

In [56]: num * 2  # multiplication
Out[56]: 246

In [59]: num /  2  # integer division
Out[59]: 61
```

还有各种类型的整数，取决于它们的基数或数字。这些包括十进制、二进制、八进制和十六进制整数。数字的正常非零前导序列是十进制整数。以 0 或通常 0o 开头防止出错的整数是八进制整数。以 0x 开头的数字是十六进制数，而以 0b 开头的是二进制整数。你还可以使用 bin()、hex()和 oct()函数将十进制整数转换为相应的基数格式。

下面的代码段说明了各种形式的整数：

```
In [94]: # decimal
In [95]: 1 + 1
Out[95]: 2

In [96]: # binary
In [97]: bin(2)
Out[97]: '0b10'
In [98]: 0b1 + 0b1
Out[98]: 2
In [99]: bin(0b1 + 0b1)
Out[99]: '0b10'

In [100]: # octal
In [101]: oct(8)
Out[101]: '010'
In [102]: oct(07 + 01)
Out[102]: '010'
In [103]: 0o10
Out[103]: 8

In [104]: # hexadecimal
In [105]: hex(16)
Out[105]: '0x10'
In [106]: 0x10
Out[106]: 16
In [116]: hex(0x16 + 0x5)
Out[116]: '0x1b'
```

浮点数或浮点表示为数字序列，它包括小数点和跟着它的一些数字（尾数），指数部分（e或E后面是+/-符号，后面再跟着数字），或有时两个都有。以下是浮点数的一些示例：

```
In [126]: 1.5 + 2.6
Out[126]: 4.1

In [127]: 1e2 + 1.5e3 + 0.5
Out[127]: 1600.5

In [128]: 2.5e4
Out[128]: 25000.0

In [129]: 2.5e-2
Out[129]: 0.025
```

浮点数的范围和精度类似于C语言中的双精度数据类型。

复数具有两个分量，由浮点数表示的实部和虚部。虚部由数字和后面的字母j组成，文字末尾的符号j表示-1的平方根。以下代码段显示了复数的一些表达和操作：

```
In [132]: cnum = 5 + 7j

In [133]: type(cnum)
Out[133]: complex

In [134]: cnum.real
Out[134]: 5.0

In [135]: cnum.imag
Out[135]: 7.0

In [136]: cnum + (1 - 0.5j)
Out[136]: (6+6.5j)
```

2.4.2 字符串

字符串（string）是用于存储和表示文本型数据的字符序列或集合——这将是本书大多数示例中所选择的数据类型。字符串能用于将文本和字节存储为信息。有各种各样用于处理和操作字符串的方法，我们将在本章后面详细介绍。需要记住的一点是字符串是不可变的，并且对字符串执行的任何操作总会创建一个新的字符串对象（还记得一个对象的三个属性吗），而不是仅仅变化和更改现有字符串对象的值。

以下代码段显示了一些字符串的表示和基本操作：

```
In [147]: s1 = 'this is a string'
In [148]: s2 = 'this is "another" string'
In [149]: s3 = 'this is the \'third\' string'
In [150]: s4 = """this is a
     ...: multiline
     ...: string"""

In [151]: print s1, s2, s3, s4
this is a string this is "another" string this is the 'third'
string this is a
multiline
string
In [152]: print s3 + '\n' + s4
this is the 'third' string
```

```
this is a
multiline
string

In [153]: ' '.join([s1, s2])
Out[153]: 'this is a string this is "another" string'

In [154]: s1[::-1]  # reverses the string
Out[154]: 'gnirts a si siht'
```

2.4.3 列表

列表（list）是任意异构或相同类型对象的集合。列表还遵循基于列表中所存对象顺序的序列，并且每个对象都有可以访问它的索引。列表与其他语言中的数组类似，与之不同的是数组保存同一数据类型的同质项，而列表可以包含不同类型的对象。一个简单的例子就是包含数字、字符串甚至子列表的列表。如果一个列表中所包含的对象为列表，这些通常称为*嵌套列表*（nested list）。

以下代码段显示了列表的一些示例：

```
In [161]: l1 = ['eggs', 'flour', 'butter']
In [162]: l2 = list([1, 'drink', 10, 'sandwiches', 0.45e-2])
In [163]: l3 = [1, 2, 3, ['a', 'b', 'c'], ['Hello', 'Python']]

In [164]: print l1, l2, l3
['eggs', 'flour', 'butter'] [1, 'drink', 10, 'sandwiches', 0.0045] [1, 2, 3,
['a', 'b', 'c'], ['Hello', 'Python']]
```

你还可以对列表执行丰富的操作，包括索引（indexing）、切片（slicing）、追加（appending）、移除（popping）等。以下代码段描述了列表的一些典型操作：

```
In [167]: # indexing lists
In [168]: l1
Out[168]: ['eggs', 'flour', 'butter']
In [169]: l1[0]
Out[169]: 'eggs'
In [170]: l1[1]
Out[170]: 'flour'
In [171]: l1[0] +' '+ l1[1]
Out[171]: 'eggs flour'

In [171]: # slicing lists
In [172]: l2[1:3]
Out[172]: ['drink', 10]
In [173]: numbers = range(10)
In [174]: numbers
Out[174]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
In [175]: numbers[2:5]
Out[175]: [2, 3, 4]
In [180]: numbers[:]
Out[180]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
In [181]: numbers[::2]
Out[181]: [0, 2, 4, 6, 8]

In [181]: # concatenating and mutating lists
In [182]: numbers * 2
Out[182]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
In [183]: numbers + l2
```

```
Out[183]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 'drink', 10, 'sandwiches',
           0.0045]

In [184]: # handling nested lists
In [184]: l3
Out[184]: [1, 2, 3, ['a', 'b', 'c'], ['Hello', 'Python']]
In [185]: l3[3]
Out[185]: ['a', 'b', 'c']
In [186]: l3[4]
Out[186]: ['Hello', 'Python']

In [187]: l3.append(' '.join(l3[4]))  # append operation
In [188]: l3
Out[188]: [1, 2, 3, ['a', 'b', 'c'], ['Hello', 'Python'], 'Hello Python']

In [189]: l3.pop(3)  # pop operation
Out[189]: ['a', 'b', 'c']
In [190]: l3
Out[190]: [1, 2, 3, ['Hello', 'Python'], 'Hello Python']
```

2.4.4 集合

集合（set）是唯一且不可变对象的无序集合。你可以使用 set()函数或大括号{...}来创建一个新的集合。集合通常用于从列表中删除重复项、测试对象的成员资格并执行数学集合操作，包括并集（union）、交集（intersection）、差集（difference）和对称差集（symmetricdifference）。

以下代码段显示了一些集合的表达和操作：

```
In [196]: l1 = [1,1,2,3,5,5,7,9,1]

In [197]: set(l1)  # makes the list as a set
Out[197]: {1, 2, 3, 5, 7, 9}


In [198]: s1 = set(l1)

# membership testing
In [199]: 1 in s1
Out[199]: True
In [200]: 100 in s1
Out[200]: False

# initialize a second set
In [201]: s2 = {5, 7, 11}

# testing various set operations
In [202]: s1 - s2  # set difference
Out[202]: {1, 2, 3, 9}

In [203]: s1 | s2  # set union
Out[203]: {1, 2, 3, 5, 7, 9, 11}

In [204]: s1 & s2  # set intersection
Out[204]: {5, 7}

In [205]: s1 ^ s2  # elements which do not appear in both sets
Out[205]: {1, 2, 3, 9, 11}
```

2.4.5 字典

Python 中的字典（dictionary）是无序且可变的键值映射。它们通常称为哈希映射（hash-map）、联合数组（associative array）和联合内存（associative memory）。字典使用键（key）索引，它可以是任何不可变的对象类型，如数值类型或字符串，甚至是我们稍后会看到的元组。要记住键应该是一些不可变的数据类型。字典的值可以是不可变的或可变的对象，包括列表和含字典本身形成的嵌套字典。如果你之前已经使用过 JSON 对象，你会看到字典与它们有很多相似之处。字典通常在 Python 中称为 `dicts`，`dict()`函数也可用于创建新的字典。

下面的代码段显示了一些字典的表达和操作：

```
In [207]: d1 = {'eggs': 2, 'milk': 3, 'spam': 10, 'ham': 15}
In [208]: d1
Out[208]: {'eggs': 2, 'ham': 15, 'milk': 3, 'spam': 10}

# retrieving items based on key
In [209]: d1.get('eggs')
Out[209]: 2
In [210]: d1['eggs']
Out[210]: 2
# get is better than direct indexing since it does not throw errors
In [211]: d1.get('orange')
In [212]: d1['orange']
Traceback (most recent call last):
  File "<ipython-input-212-ebecbf415243>", line 1, in <module>
    d1['orange']
KeyError: 'orange'

# setting items with a specific key
In [213]: d1['orange'] = 25
In [214]: d1
Out[214]: {'eggs': 2, 'ham': 15, 'milk': 3, 'orange': 25, 'spam': 10}

# viewing keys and values
In [215]: d1.keys()
Out[215]: ['orange', 'eggs', 'ham', 'milk', 'spam']
In [216]: d1.values()
Out[216]: [25, 2, 15, 3, 10]

# create a new dictionary using dict function
In [219]: d2 = dict({'orange': 5, 'melon': 17, 'milk': 10})
In [220]: d2
Out[220]: {'melon': 17, 'milk': 10, 'orange': 5}

# update dictionary d1 based on new key-values in d2
In [221]: d1.update(d2)
In [222]: d1
Out[222]: {'eggs': 2, 'ham': 15, 'melon': 17, 'milk': 10, 'orange': 5,
          'spam': 10}

# complex and nested dictionary
In [223]: d3 = {'k1': 5, 'k2': [1,2,3,4,5], 'k3': {'a': 1, 'b': 2, 'c':
          [1,2,3]}}
In [225]: d3
Out[225]: {'k1': 5, 'k2': [1, 2, 3, 4, 5], 'k3': {'a': 1, 'b': 2, 'c':
          [1, 2, 3]}}
In [226]: d3.get('k3')
Out[226]: {'a': 1, 'b': 2, 'c': [1, 2, 3]}
In [227]: d3.get('k3').get('c')
Out[227]: [1, 2, 3]
```

2.4.6 元组

元组（tuple）也是类似列表的序列，但它们是不可变的。通常，元组用于表示对象或值的固定集合。元组使用括号封装、逗号分隔的值序列进行创建，并且也可以使用 `tuple()` 函数。

以下代码段显示了一些元组的表达和操作：

```
# creating a tuple with a single element
In [234]: single_tuple = (1,)
In [235]: single_tuple
Out[235]: (1,)

# original address of the tuple
In [239]: id(single_tuple)
Out[239]: 176216328L

# modifying contents of the tuple but its location changes (new tuple is
created)
In [240]: single_tuple = single_tuple + (2, 3, 4, 5)
In [241]: single_tuple
Out[241]: (1, 2, 3, 4, 5)
In [242]: id(single_tuple) # different address indicating new tuple with
          same name
Out[242]: 201211312L

# tuples are immutable hence assignment is not supported like lists
In [243]: single_tuple[3] = 100
Traceback (most recent call last):
  File "<ipython-input-247-37d1946d4128>", line 1, in <module>
    single_tuple[3] = 100
TypeError: 'tuple' object does not support item assignment

# accessing and unpacking tuples
In [243]: tup = (['this', 'is', 'list', '1'], ['this', 'is', 'list', '2'])
In [244]: tup[0]
Out[244]: ['this', 'is', 'list', '1']
In [245]: l1, l2 = tup
In [246]: print l1, l2
['this', 'is', 'list', '1'] ['this', 'is', 'list', '2']
```

2.4.7 文件

文件（file）是 Python 中特殊的对象类型，主要用于与文件系统中的外部对象进行交互，包括文本、二进制、音频和视频文件，以及文档、图像等。有些人可能不同意把它作为 Python 中的一种数据类型，但它实际上是一种特殊的数据类型，类型名称——文件，完全适合处理所有类型的外部文件。通常你可以使用 `open()`函数来打开一个文件，并且还有在函数中指定处理模式字符的各种模式，如读取和写入。

文件处理的一些示例显示在以下代码段中：

```
In [253]: f = open('text_file.txt', 'w')   # open in write mode
In [254]: f.write("This is some text\n")  # write some text
In [255]: f.write("Hello world!")
In [256]: f.close()  # closes the file

# lists files in current directory
```

```
In [260]: import os
In [262]: os.listdir(os.getcwd())
Out[262]: ['text_file.txt']

In [263]: f = open('text_file.txt', 'r')  # opens file in read mode
In [264]: data = f.readlines()  # reads in all lines from file
In [265]: print data  # prints the text data
['This is some text\n', 'Hello world!']
```

2.4.8 杂项

除了上面已经提到的数据类型和结构，还有其他几种 Python 数据类型：

- None 类型，表示无值/无数据或空对象。
- 布尔类型，包括 `True` 和 `False`。
- 十进制和分数类型，以更好的方式处理数字。

至此，我们就完成了 Python 核心数据类型和数据结构的学习，你在代码和实现中将会经常使用它们。现在我们将讨论用于控制代码流的一些常用结构。

2.5 控制代码流

*代码流*是非常重要的，其中很多基于业务逻辑和规则。它还取决于开发人员在构建系统和应用时所采用的实现决策类型。Python 提供了几个可用于控制代码流的控制流工具和实用程序。下面是最流行的几个：

- `if-elif-else` 条件。
- `for` 循环。
- `while` 循环。
- 循环中的 `break`、`continue` 和 `else`。
- `try-except`。

这些结构将帮助你理解几个概念，包括条件代码流、循环和处理异常。

2.5.1 条件结构

条件代码流的概念包括根据代码本身实现的一些用户定义逻辑而有条件地执行不同的代码块。当你不想要按照顺序依次执行一系列语句，而是基于满足或不满足某些条件来执行其中的部分代码时，这个结构非常有用。`if-elif-else` 语句帮助实现该目的。它的一般语法如下：

```
if <conditional check 1 is True>:    # the if conditional (mandatory)
    <code block 1>   # executed only when check 1 evaluates to True
        ...
    <code block 1>
elif <conditional check 2 is True>:  # the elif conditional (optional)
    <code block 2>   # executed only when check 1 is False and 2 is True
        ...
    <code block 2>
else:                                # the else conditional (optional)
    <code block 3>   # executed only when check 1 and 2 are False
        ...
    <code block 3>
```

上述语法中要记住的一个重点是相应的代码块只能在满足必要条件的基础上执行。此外，只有 if 语句是强制性的，除非有基于条件逻辑的需要，否则不需要提及 elif 和 else 语句。

下面的示例描述了条件代码流：

```
In [270]: var = 'spam'
In [271]: if var == 'spam':
     ...:     print 'Spam'
     ...:
Spam

In [272]: var = 'ham'
In [273]: if var == 'spam':
     ...:     print 'Spam'
     ...: elif var == 'ham':
     ...:     print 'Ham'
     ...:
Ham

In [274]: var = 'foo'
In [275]: if var == 'spam':
     ...:     print 'Spam'
     ...: elif var == 'ham':
     ...:     print 'Ham'
     ...: else:
     ...:     print 'Neither Spam or Ham'
     ...:
Neither Spam or Ham
```

2.5.2 循环结构

Python 中有两种主要的循环类型：for 和 while 循环。这些循环结构用于重复执行代码块，直到满足某些条件或循环基于某些其他语句或条件退出为止。

for 语句通常用于顺序循环项，并且通常循环遍历一个或多个迭代顺序，每轮执行相同的代码块。while 语句更多地用作条件通用循环，一旦满足特定条件就停止循环，或者运行循环直到满足一些条件。有趣的是，在循环结尾有一个可选的 else 语句，只有在循环正常退出而且没有任何 break 语句时才执行它。break 语句经常与条件语句一起使用，以立即停止执行循环中的所有语句，并退出最近一轮的循环。continue 语句将停止在循环中执行下面的所有语句，并将控制返回到下一轮循环的循环开始处。pass 语句仅仅用作空的占位符——它不做任何事情，并且通常用于指示一个空的代码块。这些语句构成了核心循环结构。

以下代码段显示了构建 for 和 while 循环时一般使用的典型语法：

```
# the for loop
for item in iterable:  # loop through each item in the iterable
    <code block>    # Code block executed repeatedly
else:                  # Optional else
    <code block>    # code block executes only if loop exits normally
                    without 'break'

# the while loop
while <condition>:  # loop till condition is satisfied
    <code block>    # Code block executed repeatedly
else:                  # Optional else
    <code block>    # code block executes only if loop exits normally
                    without 'break'
```

以下示例显示了循环语句如何与其他循环结构一起工作，包括 pass、break 和continue：

```
# illustrating for loops
In [280]: numbers = range(0,5)
In [281]: for number in numbers:
     ...:     print number
     ...:
0
1
2
3
4
In [282]: sum = 0
In [283]: for number in numbers:
     ...:     sum += number
     ...:
In [284]: print sum
10

# role of the trailing else and break constructs
In [285]: for number in numbers:
     ...:     print number
     ...: else:
     ...:     print 'loop exited normally'
     ...:
0
1
2
3
4
loop exited normally
In [286]: for number in numbers:
     ...:     if number < 3:
     ...:         print number
     ...:     else:
     ...:         break
     ...: else:
     ...:     print 'loop exited normally'
     ...:
0
1
2

# illustrating while loops
In [290]: number = 5
In [291]: while number > 0:
     ...:     print number
     ...:     number -= 1  # important! else loop will keep running
     ...:
5
4
3
2
1
# role of continue construct
In [295]: number = 10
In [296]: while number > 0:
     ...:     if number % 2 != 0:
     ...:         number -=1 # decrement but do not print odd numbers
     ...:         continue  # go back to beginning of loop for next
iteration
     ...:     print number  # print even numbers and decrement count
     ...:     number -= 1
```

```
    ...:
10
8
6
4
2

# role of the pass construct
In [297]: number = 10
In [298]: while number > 0:
     ...:     if number % 2 != 0:
     ...:         pass # don't print odds
     ...:     else:
     ...:         print number
     ...:     number -= 1
     ...:
10
8
6
4
2
```

2.5.3 处理异常

异常（exception）是指某些非自然错误的发生或人为因素而触发的特殊事件。它们广泛用于错误处理、事件通知和控制代码流。使用类似 try-except-finally 的结构，当运行时无论何时发生何种错误，你都可以让 Python 在执行代码时触发异常。这使你能够捕获这些异常，并根据需要处理它们，或者完全忽略它们。在 Python 2.5.x 之前的版本中，通常有两种基于 try 结构的异常处理版本。一种是 try-finally，另一种是 try-except 以及可选的用来捕获异常的 else 子句。现在我们有一个包含它们全部的构造，即可以用于异常处理的 try-except-else-finally 结构。语法描述如下：

```
try:                            # The try statement
    <main code block>       # Checks for errors in this block
except <ExceptionType1>:        # Catch different exceptions
    <exception handler 1>
except <ExceptionType2>:
    <exception handler 2>
...
else:                           # Optional else statement
    <optional else block>   # Executes only if there were no exceptions
finally:                        # The finally statement
    <finally block>         # Always executes in the end
```

上面代码片段中的代码流从 try 语句和其中的代码块 < main code block > 开始，它首先执行并检查异常。如果发生任何异常，则根据上述代码段所示的异常类型进行匹配。假设匹配了 ExceptionType1，则执行 ExceptionType1 相应的异常处理程序 exception handler 1。如果没有异常触发，那么只执行 < optional else block > 。无论是否发生异常，始终会执行 < finally block > 。

下面的示例描述了 try-except-else-finally 结构的使用：

```
In [311]: shopping_list = ['eggs', 'ham', 'bacon']
# trying to access a non-existent item in the list
In [312]: try:
     ...:     print shopping_list[3]
```

```
     ...: except IndexError as e:
     ...:     print 'Exception: '+str(e)+' has occurred'
     ...: else:
     ...:     print 'No exceptions occurred'
     ...: finally:
     ...:     print 'I will always execute no matter what!'
     ...:
Exception: list index out of range has occurred
I will always execute no matter what!
# smooth code execution without any errors
In [313]: try:
     ...:     print shopping_list[2]
     ...: except IndexError as e:
     ...:     print 'Exception: '+str(e)+' has occurred'
     ...: else:
     ...:     print 'No exceptions occurred'
     ...: finally:
     ...:     print 'I will always execute no matter what!'
     ...:
bacon
No exceptions occurred
I will always execute no matter what!
```

有关 Python 代码控制流核心结构的讨论到此为止。下一节将介绍 Python 函数编程范式的一些核心概念和结构。

2.6 函数编程

函数编程范式是一种起源于 lambda（λ）演算的编程风格。它纯粹基于执行和评价函数来处理任何形式的计算。Python 并不是纯函数式编程语言，而是具有可用于函数编程的多个结构。在本节中，我们将讨论其中的几个结构，包括函数和一些诸如生成器（generator）、迭代器（iterator）和分析器（comprehension）的高级概念。我们还将介绍诸如 `itertools` 和 `functools` 之类的模块，其中包含基于 Haskell 和 Standard ML 概念思想实现的函数工具。

2.6.1 函数

函数（function）定义为只有通过请求调用它才能执行的代码块。函数包含一个函数定义，该函数定义有函数标识（函数名、参数）和被调用时执行的函数内部的一组语句。Python标准库提供了一套丰富的函数，用于执行不同类型的操作。此外，用户可以使用 `def` 关键字来定义自己的函数。

函数通常返回一些值，即使它们没有返回值，默认情况下会返回 `None` 类型。需要记住的一件重要事情是，通常你可能会看到可以互换使用的方法和函数，但是函数和方法两者之间的区别是方法作为在类语句中定义的函数。函数也是对象，这是因为 Python 中的每个类型和结构都是从基础对象类型派生而来的。这开启了一个全新的视角，你甚至可以将函数作为参数或变量传递给其他函数。此外，函数可以绑定到变量，甚至可以作为其他函数的返回结果。因此，函数通常称为 Python 中的一类对象。

以下代码段显示了 Python 中函数定义的基本结构：

```
def function(params):  # params are the input parameters
    <code block>       # code block consists of a group of statements
    return value(s)    # optional return statement
```

上面的 params 表示输入参数的列表，这些参数不是强制性的，实际上在许多函数中没有输入参数。你甚至可以将函数本身作为参数传递。在代码块中执行一些逻辑，这也许会/也许不会修改输入参数，最后可能会返回一些输出值，也可能完全不返回任何值。

下面的代码段展示了函数具有固定参数、变量参数和内置函数的一些基本示例：

```
# function with single argument
In [319]: def square(number):
     ...:     return number*number
     ...:

In [320]: square(5)
Out[320]: 25

# built-in function from the numpy library
In [321]: import numpy as np
In [322]: np.square(5)
Out[322]: 25

# a more complex function with variable number of arguments
In [323]: def squares(*args):
     ...:     squared_args = []
     ...:     for item in args:
     ...:         squared_args.append(item*item)
     ...:     return squared_args
     ...:
In [324]: squares(1,2,3,4,5)
Out[324]: [1, 4, 9, 16, 25]
```

上面的例子显示了如何动态地在一个函数中引入可变数量的参数。你还可以引用关键字参数，其中每个参数都有自己的变量名称和值，如以下代码段所示：

```
# assign specific keyword based arguments dynamically
In [325]: def person_details(**kwargs):
     ...:     for key, value in kwargs.items():
     ...:         print key, '->', value
     ...:
In [326]: person_details(name='James Bond', alias='007', job='Secret Service
          Agent')
alias -> 007
job -> Secret Service Agent
name -> James Bond
```

2.6.2 递归函数

递归函数运用*递归*（recursion）的概念，其中函数在其代码块内调用自身。应该注意函数要确保有一个停止条件，最终终止递归调用——否则，该函数将进入一个无限执行继续调用自身的循环。递归在每轮递归调用中利用调用堆栈，因此与常规函数相比，它们往往不是非常有效率；然而，它们却是非常强大的。

下面的例子描述了使用递归的平方（squares）函数：

```
# using recursion to square numbers
In [331]: def recursive_squares(numbers):
     ...:     if not numbers:
     ...:         return []
     ...:     else:
     ...:         return [numbers[0]*numbers[0]] + recursive_
```

```
                squares(numbers[1:])
    ...:
In [332]: recursive_squares([1, 2, 3, 4, 5])
Out[332]: [1, 4, 9, 16, 25]
```

2.6.3 匿名函数

匿名函数（anonymous function）是没有任何名称的函数，通常由单行表达式组成，表示使用 lambda 构造的函数。lambda 关键字用于定义内联函数对象，它们可以像正则函数一样使用，不过还有一些区别。如下代码段显示了 lambda 函数的通用语法：

```
lambda arg, arg2,... arg_n : <inline expression using args>
```

实际上这个表达式甚至能分配给变量，然后类似于使用 def 创建的函数，它可以作为普通函数被调用执行。然而，lambda 函数是表达式，而不是像 def 定义代码块中那样的语句，所以将复杂逻辑放在 lambda 函数中非常困难，因为它始终是一个单行的内联表达式。但是，lambda 函数非常强大，甚至可以在列表、函数和函数参数中使用。除了 lambda 函数之外，Python还提供了 map()、reduce()和 filter()等函数，它们广泛使用 lambda 函数，并将其应用于迭代量，以完成相应的转换、约简或过滤操作。

以下代码段描述了 lambda 函数的一些示例，其中使用到我们刚刚讨论的构造：

```
# simple lambda function to square a number
In [340]: lambda_square = lambda n: n*n
In [341]: lambda_square(5)
Out[341]: 25

# map function to square numbers using lambda
In [342]: map(lambda_square, [1, 2, 3, 4, 5])
Out[342]: [1, 4, 9, 16, 25]

# lambda function to find even numbers used for filtering
In [343]: lambda_evens = lambda n: n%2 == 0
In [344]: filter(lambda_evens, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
Out[344]: [2, 4, 6, 8, 10]

# lambda function to add numbers used for adding numbers in reduce function
In [345]: lambda_sum = lambda x, y: x + y
In [346]: reduce(lambda_sum, [1, 2, 3, 4, 5])
Out[346]: 15
# lambda function to make a sentence from word tokens with reduce function
In [347]: lambda_sentence_maker = lambda word1, word2: ' '.join([word1,
          word2])
In [348]: reduce(lambda_sentence_maker, ['I', 'am', 'making', 'a',
          'sentence', 'from', 'words!'])
Out[348]: 'I am making a sentence from words!'
```

前面的例子应该可以让你很好地了解 lambda 函数的工作原理以及其强大的功能。使用一行命令，你可以从单词标识（word token）中创建流畅的语句，并计算列表中的数字总和！lambda 函数的使用具有无限的可能性，你可以用它们来解决最复杂的问题。

2.6.4 迭代器

迭代器（Iterator）是用于对可迭代对象实现迭代计算的结构。可迭代对象（iterable）基本上是其他对象和数据的序列。for 循环是一个很好的例子，它实际上是一个循环遍历列

表或序列的可迭代对象。迭代器是使用 next()函数来遍历可迭代对象的对象或构造，它可以在每次调用时从可迭代对象返回下一项。一旦遍历了全部的可迭代对象，它返回一个 StopIteration 异常。我们已经了解 for 循环的一般工作原理，但是在抽象背后，for 循环实际上是对可迭代对象调用 iter()函数来创建一个迭代器对象，然后使用 next()函数来遍历它。

以下示例说明了迭代器的工作原理：

```
# typical for loop
In [350]: numbers = range(6)
In [351]: for number in numbers:
     ...:     print number
0
1
2
3
4
5

# illustrating how iterators work behind the scenes
In [352]: iterator_obj = iter(numbers)
In [353]: while True:
     ...:     try:
     ...:         print iterator_obj.next()
     ...:     except StopIteration:
     ...:         print 'Reached end of sequence'
     ...:         break
0
1
2
3
4
5
Reached end of sequence

# calling next now would throw the StopIteration exception as expected
In [354]: iterator_obj.next()
Traceback (most recent call last):
  File "<ipython-input-354-491178c4f97a>", line 1, in <module>
    iterator_obj.next()
StopIteration
```

2.6.5 分析器

分析器（comprehension）是有趣的构造，它与 for 循环类似但更有效率。无疑，它们属于遵循集合生成器符号（set builder notation）的函数编程范式。最初列表分析器（list comprehension）的思想来自 Haskell，经过一系列冗长的讨论后，最终添加了分析器，自那以后它已成为最常用的结构之一。目前有可以应用于现有数据类型的各种类型分析器，包括列表、集合和字符串的分析器。以下代码段显示了使用非常常见的列表分析器和 for 循环的解析语法，for 循环是分析器中的核心组件：

```
# typical comprehension syntax
[ expression for item in iterable ]

# equivalent for loop statement
for item in iterable:
```

```
    expression

# complex and nested iterations
[ expression for item1 in iterable1 if condition1
             for item2 in iterable2 if condition2 ...
             for itemN in iterableN if conditionN ]

# equivalent for loop statement
for item1 in iterable1:
    if condition1:
        for item2 in iterable2:
            if condition2:
                ...
                    for itemN in iterableN:
                        if conditionN:
                            expression
```

我们看到分析器与循环结构是很相似的。解析器的优势是它们比循环更有效率，性能表现更好。如果你还记得前面讲的语法，就会知道这里有一些注意事项，包括你不能在分析器中使用赋值语句，因为它们仅支持表达式而不支持语句。集合和字典分析器也使用与此相同的语法。

以下例子说明了不同分析器的用法：

```
In [355]: numbers = range(6)
In [356]: numbers
Out[356]: [0, 1, 2, 3, 4, 5]

# simple list comprehension to compute squares
In [357]: [num*num for num in numbers]
Out[357]: [0, 1, 4, 9, 16, 25]

# list comprehension to check if number is divisible by 2
In [358]: [num%2 for num in numbers]
Out[358]: [0, 1, 0, 1, 0, 1]

# set comprehension returns distinct values of the above operation
In [359]: set(num%2 for num in numbers)
Out[359]: {0, 1}

# dictionary comprehension where key:value is number: square(number)
In [361]: {num: num*num for num in numbers}
Out[361]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

# a more complex comprehension showcasing above operations in a single
comprehension
In [362]: [{'number': num,
            'square': num*num,
            'type': 'even' if num%2 == 0 else 'odd'} for num in numbers]
Out[362]:
[{'number': 0, 'square': 0, 'type': 'even'},
 {'number': 1, 'square': 1, 'type': 'odd'},
 {'number': 2, 'square': 4, 'type': 'even'},
 {'number': 3, 'square': 9, 'type': 'odd'},
 {'number': 4, 'square': 16, 'type': 'even'},
 {'number': 5, 'square': 25, 'type': 'odd'}]

# nested list comprehension - flattening a list of lists
In [364]: list_of_lists = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
In [365]: list_of_lists
Out[365]: [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
In [367]: [item for each_list in list_of_lists for item in each_list]
Out[367]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

2.6.6 生成器

生成器（generator）是用于创建和使用迭代器的功能强大、内存高效的结构。其存在两种变体：函数和表达式。生成器的工作思想称为惰性计算（lazy evaluation）——因此，它们在多数情况下内存效率更高、执行更好，因为它们不需要像列表分析器一次性对全部对象进行计算和加载。然而，需要注意的是，由于生成器是以特定方式一次性产生一项，所以与列表分析器相比，它们在执行时间方面可能会表现得较差，除非是处理含有许多元素的大对象。

生成器函数以 def 语句定义的常规函数。它们使用惰性计算的概念，并使用 `yield` 语句一次返回一个对象。生成器函数与具有 `return` 语句的正则函数不同，正则函数一旦执行结束将执行函数内部的代码块，而生成器将使用 `yield` 语句，该语句在生成并返回每个值或对象后，暂停或恢复执行和状态。更准确地说，生成器函数在每个步骤产生值，而不是返回它们。这确保了当前状态，包括关于其保留的本地代码块范围信息，并且使生成器能够从其停止的地方恢复。

以下代码段显示了生成器函数的一些示例：

```
In [369]: numbers = [1, 2, 3, 4, 5]

In [370]: def generate_squares(numbers):
     ...:     for number in numbers:
     ...:         yield number*number

In [371]: gen_obj = generate_squares(numbers)
In [372]: gen_obj
Out[372]: <generator object generate_squares at 0x000000000F2FC2D0>
In [373]: for item in gen_obj:
     ...:     print item
     ...:
1
4
9
16
25
```

这些生成器的优点在于内存效率和执行时间，特别是当可迭代对象和对象容量较大且占用大量内存时。你也无须将整个对象加载到主内存中，以对其执行各种操作。对于流数据它们经常处理得很好，在那种情况下你不能随时将所有流数据保存在内存中。这同样适用于生成器表达式，它们与分析器非常相似，除了它们是用括号括起来的，如下所示。

```
In [381]: csv_string = 'The,fox,jumps,over,the,dog'

# making a sentence using list comprehension
In [382]: list_cmp_obj = [item for item in csv_string.split(',')]
In [383]: list_cmp_obj
Out[383]: ['The', 'fox', 'jumps', 'over', 'the', 'dog']
In [384]: ' '.join(list_cmp_obj)
Out[384]: 'The fox jumps over the dog'

# making a sentence using generator expression
In [385]: gen_obj = (item for item in csv_string.split(','))
In [386]: gen_obj
Out[386]: <generator object <genexpr> at 0x000000000F2FC3F0>
In [387]: ' '.join(gen_obj)
Out[387]: 'The fox jumps over the dog'
```

生成器函数和表达式都会使用与迭代器相同的构造来创建生成器对象，并在每个阶段启动、停止和恢复函数或循环，一旦完成，它将触发 StopIteration 异常。

2.6.7 itertools 和 functods 模块

Python 标准库中提供各种各样的模块。一些主流的模块包括 collections、itertools 和 functools，它们提供各种构造和函数，可用于提升工作效率，并减少编写额外代码来解决问题所花费的时间。itertools 模块是专门用于构建和操作迭代器的完整模块。它提供支持不同操作的各种函数，包括切片、链接、分组和分割迭代器。有关 itertools 最全面的信息来源可以在 https://docs.python.org/2/library/itertools.html 上的官方 Python 文档中查到。该文档列出了每个函数及其功能示例。functools 模块提供 with 函数，其概念来源于函数编程，包括 wrapper 和 partial。这些函数通常作用于其他函数，作为输入参数并通常返回一个函数作为结果。https://docs.python.org/2/library/functools.html 上的官方文档提供了每个函数的详细信息。

2.7 类

Python 类（class）是确保人们能遵循 OOP 范例进行代码编写的构造。在范例中，会大量使用像对象、封装、方法、继承和多态等概念。如果你以前使用过任何 OOP 语言（如 C++ 或 Java），可能会发现使用 Python 类与在其他语言中使用类的情形比较类似。详细讨论每个概念将超出本书的范围，但我将简要介绍类的基本概念，并介绍不同类型的对象和继承。

类基本上是真实世界实体对象的软件模型或抽象。这种抽象导致了类被称为用户定义的类型，一旦你定义了一个类，就可以实例化和创建该类的实例或对象。每个对象都有自己的实例变量和方法，用来定义该对象的属性和行为。所有类都从基础对象类型继承，你可以创建自己的类，并从这些用户定义的类继承更多的类。类本身也是最终的对象，并且可以绑定到变量和其他结构。

以下代码段给出了类的基本语法：

```
class ClassName(BaseClass):
    class_variable  # shared by all instances\objects

    def __init__(self, ...): # the constructor
        # instance variables unique to each instance\object
        self.instance_variables = ...

    def __str__(self):  # string representation of the instance\object
        return repr(self)

    def methods(self, ...):  # instance methods
        <code block>
```

上面的代码段说明了名为 ClassName 的类继承自其父类 BaseClass。在参数中可以有逗号分隔的多个父类或基类。_init_()方法充当构造函数，使用 ClassName(...)来实例化和创建类的对象，该方法自动调用_init_()方法——该方法可以根据其定义来选择参数。_str_()方法是可选的，它打印对象的字符串表示。你可以使用自定义修改默认方法，并且

它经常用于打印对象变量的当前状态。class_variable 表示仅包含在类定义块中所定义的类变量，并且这些类变量由该类的所有对象或实例共享。instance_variables 是特定于每个对象或实例的变量。这些方法表示实例方法，定义了对象的具体行为。self 参数通常用作方法中的第一个参数，这更多是作为引用类 ClassName 的实例或对象来调用该方法的惯例。

以下示例描述了一个简单的类及其工作原理：

```
# class definition
In [401]: class Animal(object):
     ...:     species = 'Animal'
     ...:
     ...:     def __init__(self, name):
     ...:         self.name = name
     ...:         self.attributes = []
     ...:
     ...:     def add_attributes(self, attributes):
     ...:         self.attributes.extend(attributes) \
     ...:             if type(attributes) == list \
     ...:             else self.attributes.append(attributes)
     ...:
     ...:     def __str__(self):
     ...:         return self.name+" is of type "+self.species+" and has
                  attributes:"+str(self.attributes)
     ...:

# instantiating the class
In [402]: a1 = Animal('Rover')
# invoking instance method
In [403]: a1.add_attributes(['runs', 'eats', 'dog'])
# user defined string representation of the Animal class
In [404]: str(a1)
Out[404]: "Rover is of type Animal and has attributes:['runs', 'eats',
          'dog']"
```

这让我们了解到类的工作原理。但是，如果我们想要针对特定的动物，比如 dogs 和 foxes呢？我们可以应用继承的概念，使用 super()方法来访问每个定义中 Animal 基类的构造函数。下面的示例说明了继承的概念：

```
# deriving class Dog from base class Animal
In [413]: class Dog(Animal):
     ...:     species = 'Dog'
     ...:
     ...:     def __init__(self, *args):
     ...:         super(Dog, self).__init__(*args)

# deriving class Fox from base class Animal
In [414]: class Fox(Animal):
     ...:     species = 'Fox'
     ...:
     ...:     def __init__(self, *args):
     ...:         super(Fox, self).__init__(*args)

# creating instance of class Dog
In [415]: d1 = Dog('Rover')
In [416]: d1.add_attributes(['lazy', 'beige', 'sleeps', 'eats'])
In [417]: str(d1)
Out[417]: "Rover is of type Dog and has attributes:['lazy', 'beige',
          'sleeps', 'eats']"
```

```
# creating instance of class Fox
In [418]: f1 = Fox('Silver')
In [419]: f1.add_attributes(['quick', 'brown', 'jumps', 'runs'])
In [420]: str(f1)
Out[420]: "Silver is of type Fox and has attributes:['quick', 'brown',
          'jumps', 'runs']"
```

2.8 使用文本

在前面的章节中，我们已经了解了与 Python 相关的大部分构造、数据类型、结构、概念和编程范例。本节简要介绍适合处理文本数据的特定数据类型，并展示这些数据类型及其相关实用工具在后续章节中的用途。Python 中用于处理文本数据的主要数据类型是字符串，其可以是常规字符串、存储二进制信息的字节或 Unicode。默认情况下，Python 3. x 中的所有字符串都是 Unicode，但在 Python 2. x 中并不是这样，因此你在处理不同 Python 发行版中的文本时应该牢记这一点。字符串是 Python 中的字符序列，类似于数组和代码，其可以利用一组属性和方法来轻松地对文本数据进行操作处理，这使得 Python 成为许多场景中进行文本分析的首选语言。我们将在下一节中讨论各种类型的字符串及其例子。

2.8.1 字符串文字

如前所述，Python 有各种类型的字符串，在前面关于数据类型的几节中你已经看到了一些例子。以下 BNF（Backus-Naur Form）为我们提供了如 Python 官方文档中所示的生成字符串的通用词汇定义：

```
stringliteral   ::=  [stringprefix](shortstring | longstring)
stringprefix    ::=  "r" | "u" | "ur" | "R" | "U" | "UR" | "Ur" | "uR"
                     | "b" | "B" | "br" | "Br" | "bR" | "BR"
shortstring     ::=  "'" shortstringitem* "'" | '"' shortstringitem* '"'
longstring      ::=  "'''" longstringitem* "'''" | '"""' longstringitem*
'"""'
shortstringitem ::=  shortstringchar | escapeseq
longstringitem  ::=  longstringchar | escapeseq
shortstringchar ::=  <any source character except "\" or newline or the
quote>
longstringchar  ::=  <any source character except "\">
escapeseq       ::=  "\" <any ASCII character>
```

上述规则说明存在不同类型的字符串前缀，可以使用不同的字符串类型来生成字符串文字。简单地说，下列类型的字符串使用最多。

- 短字符串（short string）：这些字符串通常用单个（'）或双引号（"）把字符括起来。例如'Hello'和" Hello"。
- 长字符串（long string）：这些字符串通常用三个单个（'''）或三个双引号（"""）把字符括起来。例如""" Hello, I'm a long string""" 或者'''Hello I\'m a long string '''。注意（\'）表示下面所讨论的转义序列。
- 字符串转义序列（escape sequences in strings）：这些字符串通常具有嵌入的转义序列，其中转义序列的规则以反斜杠（\）开始，随后是任意的 ASCII 字符。因此，它们执行退格插补。流行的转义序列包括（\n）——表示新行的字符，和(\t)——表示的是一个 tab。

- 字节（bytes）：用来表示字节串（bytestring），从而创建数据类型 bytes 的对象。这些字符串可以创建为 bytes（'...'），或使用 b'...'表示。例如 bytes('hello')和 b'hello'。
- 原始字符串（raw strings）：这些字符串最初专门用于正则表达式（regex）和创建 regex模式。这些字符串可以使用 r'...'表示法创建，并将字符串保持为原始或原生形式。因此，它不执行任何退格插补，并关闭了转义序列。例如：r'Hello'。
- Unicode：这些字符串在文本中支持 Unicode 字符，通常是非 ASCII 字符序列。字符串用 u'...'表示。除了字符串表示之外，还有几种具体的方法来表示字符串中的特殊 Unicode 字符。通常包括十六进制字节值转义序列，格式为'\xVV'。除此之外，我们还有'\uVVVV'和'\uVVVVVVVV'格式的 Unicode 转义序列，其中第一个格式使用 4 个十六进制数字来编码 16 位字符，第二个格式使用 8 个十六进制数字来编码 32 位字符。例如 u'H\xe811o'和 u'H\u00e811o'，表示字符串'Hèllo'。

以下代码段描述了这些不同类型的字符串及其输出：

```
# simple string
In [422]: simple_string = 'hello' + " I'm a simple string"
In [423]: print simple_string
hello I'm a simple string

# multi-line string, note the \n (newline) escape character automatically
created
In [424]: multi_line_string = """Hello I'm
     ...: a multi-line
     ...: string!"""
In [425]: multi_line_string
Out[425]: "Hello I'm\na multi-line\nstring!"
In [426]: print multi_line_string
Hello I'm
a multi-line
string!
# Normal string with escape sequences leading to a wrong file path!
In [427]: escaped_string = "C:\the_folder\new_dir\file.txt"
In [428]: print escaped_string  # will cause errors if we try to open a file
          here
C:      he_folder
ew_dirile.txt

# raw string keeping the backslashes in its normal form
In [429]: raw_string = r'C:\the_folder\new_dir\file.txt'
In [430]: print raw_string
C:\the_folder\new_dir\file.txt

# unicode string literals
In [431]: string_with_unicode = u'H\u00e8llo!'
     ...: print string_with_unicode
Hèllo!
```

2.8.2 字符串操作和方法

字符串是可迭代的序列，这意味着可以对它们执行大量操作，特别是当将文本数据处理和解析为易于使用（easy-to-consume）的格式时尤其有用。可以对字符串执行几种操作，它们分类为以下几个部分：

- 基本操作。
- 索引和切片。
- 方法。
- 格式化。
- 正则表达式。

这些将涵盖使用字符串时最常用的技术，并为下一章开始所需的工作打好基础（我们将根据前两章中学到的概念来理解和处理文本数据）。

1. 基本操作

你可以对字符串执行几个基本操作，包括连接和检查子字符串、字符和长度。以下代码段通过一些示例说明了这些操作：

```
# Different ways of String concatenation
In [436]: 'Hello' + ' and welcome ' + 'to Python!'
Out[436]: 'Hello and welcome to Python!'
In [437]: 'Hello' ' and welcome ' 'to Python!'
Out[437]: 'Hello and welcome to Python!'
# concatenation of variables and literals
In [438]: s1 = 'Python!'
In [439]: 'Hello ' + s1
Out[439]: 'Hello Python!'
# we cannot concatenate a variable and a literal using this method
In [440]: 'Hello ' s1
  File "<ipython-input-440-2f801ddf3480>", line 1
    'Hello ' s1
             ^
SyntaxError: invalid syntax

# some more ways of concatenating strings
In [442]: s2 = '--Python--'
In [443]: s2 * 5
Out[443]: '--Python----Python----Python----Python----Python--'
In [444]: s1 + s2
Out[444]: 'Python!--Python--'
In [445]: (s1 + s2)*3
Out[445]: 'Python!--Python--Python!--Python--Python!--Python--'

# concatenating several strings together in parentheses
In [446]: s3 = ('This '
     ...:       'is another way '
     ...:       'to concatenate '
     ...:       'several strings!')
In [447]: s3
Out[447]: 'This is another way to concatenate several strings!'

# checking for substrings in a string
In [448]: 'way' in s3
Out[448]: True
In [449]: 'python' in s3
Out[449]: False
# computing total length of the string
In [450]: len(s3)
Out[450]: 51
```

2. 索引和切片

如前所述，字符串是可迭代对象——有序字符的序列。因此，字符串可以与其他可迭代

对象（例如列表）类似地进行索引、切片和迭代。在字符串中，每个字符都有一个特定的位置，称为它的*索引*（index）。使用索引，我们就可以访问字符串的指定部分。使用字符串中的特定位置或索引访问单个字符称为*索引*（indexing），而访问字符串的一部分，例如使用开始和结束索引的子字符串则称为*切片*（slicing）。Python 支持两种索引类型。一种是从 0 开始，每个字符每次增加 1，直到字符串的结尾。另一种是从字符串末尾的 -1 开始，每个字符每次减 1，直到字符串的开头。图 2-6 显示了字符串 'PYTHON' 的两种索引类型。

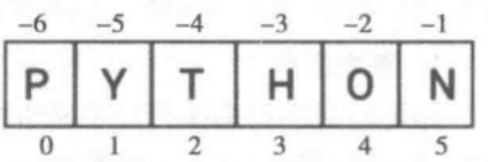


图 2-6 字符串索引语法

要访问字符串中的任何特定字符，你必须使用相应的索引，可以使用语法 var [start: stop] 来提取切片，该语法提取字符串 var 中从索引 start 直到索引 stop 的所有字符，但不包括位于 stop 索引上的字符。

以下示例显示如何对字符串进行索引、切片和迭代：

```
# creating a string
In [460]: s = 'PYTHON'

# depicting string indexes
In [461]: for index, character in enumerate(s):
     ...:     print 'Character', character+':', 'has index:', index
Character P: has index: 0
Character Y: has index: 1
Character T: has index: 2
Character H: has index: 3
Character O: has index: 4
Character N: has index: 5

# string indexing
In [462]: s[0], s[1], s[2], s[3], s[4], s[5]
Out[462]: ('P', 'Y', 'T', 'H', 'O', 'N')
In [463]: s[-1], s[-2], s[-3], s[-4], s[-5], s[-6]
Out[463]: ('N', 'O', 'H', 'T', 'Y', 'P')

# string slicing
In [464]: s[:]
Out[464]: 'PYTHON'   # prints whole string when no indexes are specified
In [465]: s[1:4]
Out[465]: 'YTH'
In [466]: s[:3]
Out[466]: 'PYT'
In [467]: s[3:]
Out[467]: 'HON'
In [468]: s[-3:]
Out[468]: 'HON'
In [469]: s[:3] + s[3:]
Out[469]: 'PYTHON'
In [470]: s[:3] + s[-3:]
Out[470]: 'PYTHON'

# string slicing with offsets
In [472]: s[::1]  # no offset
Out[472]: 'PYTHON'
In [473]: s[::2]  # print every 2nd character in string
Out[473]: 'PTO'

# strings are immutable hence assignment throws error
In [476]: s[0] = 'X'
```

```
Traceback (most recent call last):
  File "<ipython-input-476-2cd5921aae94>", line 1, in <module>
    s[0] = 'X'
TypeError: 'str' object does not support item assignment

# creates a new string
In [477]: 'X' + s[1:]
Out[477]: 'XYTHON'
```

3. 方法

字符串和 Unicode 为你提供了一个庞大的内置方法（method），可用于对字符串执行各种转换、控制和操作。详细讨论每个方法将超出当前的范围，在https://docs.python.org/2/library/stdtypes.html#string-methods 上的官方 Python 文档提供了你所需要的每个方法的所有信息，以及语法和定义。方法是非常有用的，能提高你的工作效率，因为你不必再花费额外的时间编写样板代码来处理和操作字符串。

以下代码段展示了一些流行的字符串操作方法示例：

```
# case conversions
In [484]: s = 'python is great'
In [485]: s.capitalize()
Out[485]: 'Python is great'
In [486]: s.upper()
Out[486]: 'PYTHON IS GREAT'

# string replace
In [487]: s.replace('python', 'analytics')
Out[487]: 'analytics is great'


# string splitting and joining
In [488]: s = 'I,am,a,comma,separated,string'
In [489]: s.split(',')
Out[489]: ['I', 'am', 'a', 'comma', 'separated', 'string']
In [490]: ' '.join(s.split(','))
Out[490]: 'I am a comma separated string'

# stripping whitespace characters
In [497]: s = '    I am surrounded by spaces    '
In [498]: s
Out[498]: '    I am surrounded by spaces    '
In [499]: s.strip()
Out[499]: 'I am surrounded by spaces'

# coverting to title case
In [500]: s = 'this is in lower case'
In [501]: s.title()
Out[501]: 'This Is In Lower Case'
```

前面的例子中，对于许多可能用于控制和操作字符串的介绍仅仅是浅尝即止。请使用文档中提到的不同方法尝试其他操作。我们将在后面的章节中使用其中的几个。

4. 格式化

字符串的格式化（formatting）用于替换字符串中的特定数据对象和类型。当向用户显示文本时，格式化最常用。字符串主要有两种不同类型的格式化：

- 格式化表达式（formatting expressions）：这些表达式通常的语法是'...%s...%s...'%(values)，其中%s 表示用于从 values 中描绘的字符串列表中替换字符串的占

位符。这与C语言风格的printf模块非常相似，Python自始至今都保留着。你可以用%符号加上后面的相应字母来替换其他类型的值，如整数的%d和浮点数的%f。

- 格式化方法（formatting methods）：这些字符串采用'... {} ... {} ... '.format(values)的形式，其使用大括号{}作为占位符，通过format方法从values中放置字符串。Python自2.6.x版本以来，这个方法一直存在。

下面的代码段用了几个示例描述了两种类型的字符串格式化：

```
# simple string formatting expressions
In [506]: 'Hello %s' %('Python!')
Out[506]: 'Hello Python!'


In [507]: 'Hello %s' %('World!')
Out[507]: 'Hello World!'

# formatting expressions with different data types
In [508]: 'We have %d %s containing %.2f gallons of %s' %(2, 'bottles', 2.5,
          'milk')
Out[508]: 'We have 2 bottles containing 2.50 gallons of milk'
In [509]: 'We have %d %s containing %.2f gallons of %s' %(5, 'jugs', 10.867,
          'juice')
Out[509]: 'We have 5 jugs containing 10.87 gallons of juice'

# formatting using the format method
In [511]: 'Hello {} {}, it is a great {} to meet you'.format('Mr.', 'Jones',
          'pleasure')
Out[511]: 'Hello Mr. Jones, it is a great pleasure to meet you'
In [512]: 'Hello {} {}, it is a great {} to meet you'.format('Sir',
          'Arthur', 'honor')
Out[512]: 'Hello Sir Arthur, it is a great honor to meet you'

# alternative ways of using format
In [513]: 'I have a {food_item} and a {drink_item} with me'.format(drink_
          item='soda', food_item='sandwich')
Out[513]: 'I have a sandwich and a soda with me'
In [514]: 'The {animal} has the following attributes: {attributes}'.
          format(animal='dog', attributes=['lazy', 'loyal'])
Out[514]: "The dog has the following attributes: ['lazy', 'loyal']"
```

从上述示例中，你可以看到格式化字符串并没有硬性规定，因此可以继续尝试使用不同的格式，并使用最适合你任务的格式。

5. 正则表达式

正则表达式（regular expressions）也称为regexes，允许你创建字符串模式，并使用它们来搜索和替换文本数据中的特定匹配模式。Python提供了一个名为re的富模块，用于创建和使用正则表达式。我们整本书都是在讨论这个话题，因为它易于使用，但很难掌握。通过寥寥数页来讨论正则表达式的各个方面是不可能的，但是本书将提供丰富的案例来介绍其主要领域。

正则表达式是通常使用原始字符串符号表示的特定模式。这些模式基于模式表达规则与特定的一组字符串相匹配。然后，这些模式通常被编译成字节码，再通过使用匹配的引擎执行匹配的字符串。re模块还提供了几个可以改变模式匹配执行方式的标识（flag）参数。一些重要的标识参数包括以下内容：

- re.I或re.IGNORECASE用于匹配忽略区分大小写的模式。

- re. S 或 re. DOTALL 使句点（.）字符匹配包括新行的任何字符。
- re. U 或 re. UNICODE 有助于匹配基于 Unicode 的字符（在 Python 3. x 中不推荐使用）。

对于模式匹配，在正则表达式中使用了各种各样的规则。一些流行的规则如下：

- . 用于匹配任意的单个字符。
- ^用于匹配字符串的首字符。
- $ 用于匹配字符串的末尾字符。
- * 用于在模式中的* 符号之前，匹配之前所述正则表达式的零个或多个情况。
- ? 用于在模式中的? 符号之前，匹配之前所述正则表达式的零个或一个情况。
- [...] 用于匹配方括号内的任意一组字符。
- [^...] 用于匹配在方括号中^符号之后不存在的一个字符。
- | 表示 OR 运算符，用于匹配前一个或下一个正则表达式。
- + 用于在模式中的 + 符号之前，匹配之前所述正则表达式的一个或多个情况。
- \d 用于匹配十进制数字，也描述为 [0-9]。
- \D 用于匹配非数字，也描述为 [^0-9]。
- \s 用于匹配空格字符。
- \S 用于匹配非空格字符。
- \w 用于匹配字母数字字符，也描述为 [a-zA-Z0-9_]。
- \W 用于匹配非字母数字字符，也描绘为 [^a-zA-Z0-9_]。

正则表达式可以编译为模式对象，然后使用各种方法对字符串进行模式搜索和替换。re 模块提供执行这些操作的主要方法，如下所示。

- re. compile()：该方法将指定的正则表达式模式编译为可用于匹配和搜索的正则表达式对象。如前所述，需要采用一个模式和可选标识作为输入参数。
- re. match()：该方法用于匹配字符串开始的模式。
- re. search()：该方法用于匹配出现在字符串中任意位置的模式。
- re. findall()：该方法返回字符串中对于指定的正则表达式模式的所有非重叠匹配项。
- re. finditer()：当从左到右扫描时，该方法以迭代器形式返回字符串中特定模式的所有匹配实例。
- re. sub()：该方法用于以替换串来替代字符串中指定的正则表达式模式。它仅仅替换字符串中最左侧出现的模式。

以下代码段描述了刚才讨论的一些方法，以及在处理字符串和正则表达式时通常使用的步骤：

```
# importing the re module
In [526]: import re

# dealing with unicode matching using regexes
In [527]: s = u'H\u00e8llo'
In [528]: s
Out[528]: u'H\xe8llo'
In [529]: print s
Hèllo
# does not return the special unicode character even if it is alphanumeric
```

```
In [530]: re.findall(r'\w+', s)
Out[530]: [u'H', u'llo']
# need to explicitly specify the unicode flag to detect it using regex
In [531]: re.findall(r'\w+', s, re.UNICODE)
Out[531]: [u'H\xe8llo']

# setting up a pattern we want to use as a regex
# also creating two sample strings
In [534]: pattern = 'python'
     ...: s1 = 'Python is an excellent language'
     ...: s2 = 'I love the Python language. I also use Python to build
          applications at work!'

# match only returns a match if regex match is found at the beginning of the
string
In [535]: re.match(pattern, s1)
# pattern is in lower case hence ignore case flag helps
# in matching same pattern with different cases
In [536]: re.match(pattern, s1, flags=re.IGNORECASE)
Out[536]: <_sre.SRE_Match at 0xf378308>
# printing matched string and its indices in the original string
In [537]: m = re.match(pattern, s1, flags=re.IGNORECASE)
In [538]: print 'Found match {} ranging from index {} - {} in the string
"{}"'.format(m.group(0), m.start(), m.end(), s1)
Found match Python ranging from index 0 - 6 in the string "Python is an
excellent language"

# match does not work when pattern is not there in the
beginning of string s2
In [539]: re.match(pattern, s2, re.IGNORECASE)

# illustrating find and search methods using the re module
In [540]: re.search(pattern, s2, re.IGNORECASE)
Out[540]: <_sre.SRE_Match at 0xf378920>

In [541]: re.findall(pattern, s2, re.IGNORECASE)
Out[541]: ['Python', 'Python']

In [542]: match_objs = re.finditer(pattern, s2, re.IGNORECASE)
In [543]: print "String:", s2
     ...: for m in match_objs:
     ...:     print 'Found match "{}" ranging from index {} - {}'.format(m.
              group(0), m.start(), m.end())
String: I love the Python language. I also use Python to build applications
at work!
Found match "Python" ranging from index 11 - 17
Found match "Python" ranging from index 39 - 45

# illustrating pattern substitution using sub and subn methods
In [544]: re.sub(pattern, 'Java', s2, flags=re.IGNORECASE)
Out[544]: 'I love the Java language. I also use Java to build applications
          at work!'
In [545]: re.subn(pattern, 'Java', s2, flags=re.IGNORECASE)
Out[545]: ('I love the Java language. I also use Java to build applications
          at work!', 2)
```

至此，我们将结束有关字符串的各个方面内容以及如何将它们用于处理文本数据的讨论。字符串形成处理文本的基础，这是文本分析中的重要组成内容。下一节将简要讨论一些流行的文本分析框架。

2.9 文本分析框架

就像本书之前提到的，Python 生态系统非常多样化，支持许多领域中各种各样的库、框架和模块。因为我们将要分析文本数据并对其进行各种操作，所以你需要了解专用的框架和文本分析库，你可以安装和开始使用它们——就像 Python 标准库中的任何其他内置模块一样。这些框架已经存在很长一段时间，包含各种方法、功能和特点，用于在文本上操作，以获取洞察力，并为进一步的分析做好数据准备，例如对预处理后的文本数据应用机器学习算法。

利用这些框架可以节省大量的精力和时间，否则在编写样板代码来加工、处理和操作文本数据时将花费很多时间。因此，这些框架可使开发人员和研究人员能更多地关注解决实际问题以及这样做所需的必要逻辑和算法。在第 1 章我们已经了解一些 NLTK 库。以下列出了一些最流行的文本分析库和框架，我们将在学习本书的整个过程中应用到其中的几个。

- NLTK：该自然语言工具包是一个包含 50 多个语料库和词汇资源的完整平台。它还提供必要的工具、接口和方法来处理和分析文本数据。
- `pattern`：`pattern` 项目开始于比利时安特卫普大学计算语言与心理语言学研究中心的研究项目。它为 Web 挖掘、信息检索、NLP、机器学习和网络分析提供了工具和接口。`pattern.en` 模块包含文本分析的大多数实用工具。
- `gensim`：`gensim` 库具有一套丰富的语义分析功能，包括主题建模和相似性分析。而其最重要的是包含一个谷歌非常流行的 word2vec 模型（最初可用作 C 语言包）的 Python 接口，这个神经网络模型实现了学习词语的分布式表达，其中相似词（语义）彼此相邻出现。
- `textblob`：它是另一个提供多种功能的库，包括文本处理、短语提取、分类、POS 标注、文本翻译和情感分析。
- `spacy`：这是一个较新的库，它声称通过每种技术和算法的最佳实现来提供工业级 NLP 功能，使得 NLP 任务在性能和实现方面高效。

除了这些，还有其他几个不是专门用于文本分析的框架和库，但是当你想对文本数据使用机器学习技术时，这些框架和库是有用的。以上这些库包括 `scikit-learn`、`numpy` 和 `scipy stack`。除此之外，如果你想要构建基于深度神经网络、ConvNet 和基于 LSTM 的模型的高级深度学习（deep learning）模型，深度学习和基于张量的库如 `theano`、`tensorflow` 和 `keras` 也会用得上的。你可以使用命令提示符或终端上的 `pip install <library>` 命令来安装大多数库。当我们使用这些库时，我们将在接下来的章节中讨论出现的有关注意事项。

2.10 小结

本章提供了整个 Python 生态系统的鸟瞰和详细视图，以及该语言所提供的功能情况。你阅读了 Python 语言的起源，了解到它的演变过程。该语言具有开放源代码的优点，这让积极的开发人员社区持续不断努力，以改进语言并添加新功能。到目前为止，你还知道什么

时候应该使用 Python 以及与此语言相关的缺点——这是每个开发人员在构建系统和应用程序时都应该牢记的。本章还讨论了如何设置 Python 环境并处理多个虚拟环境。

我们从基础开始，深入了解了 Python 语言的各种结构和构造，包括数据类型、使用循环和条件的控制代码流。我们也探讨了各种编程范例中的概念，包括 OOP 和函数编程。在编写高质量的 Python 代码时，类、函数、lambda、迭代器、生成器和分析器等构造是可以在很多场景中派上用场的工具。你还看到了如何使用字符串的数据类型及其各种语法、方法、操作和格式来处理文本数据。我们还讨论了正则表达式的强大功能，以及它们在模式匹配和替换中的详细用处。最后，我们研究了各种流行的文本分析框架，这些框架在解决与 NLP 有关的问题和任务中特别有用，能从文本数据中分析和提取洞察力。

以上所述的都应该可以让你开始使用 Python 编程了。下一章我们将在本章的基础上，开始理解、处理和解析可用格式的文本数据。

第3章
处理和理解文本

到目前为止，我们已经介绍了自然语言处理（NLP）和文本分析的主要概念和相关领域。在上一章中，我们还对 Python 编程语言有了一定程度的了解，特别是关于不同的结构和语法以及如何使用字符串来管理文本数据。想要执行不同的操作并分析文本，你需要将文本数据处理和解析为更容易解读的格式。

所有机器学习（ML）算法，无论是有监督的还是无监督的，通常都会使用数值格式的输入特征。虽然这是特征工程下的一个独立主题，但是我们仍然将详细地探讨它。为了实现数值格式的特征输入，你需要清洗、规范化和预处理初始文本数据。通常，文本语料库和原始文本的数据格式既非准确的，也非规范的，当然，我们应该可以预料到这些——毕竟文本数据是高度非结构化的。文本处理，或者更具体地说——文本预处理，涉及使用各种技术将原始文本转换成定义良好的语言成分序列，这些序列具有标准的结构和标记。

通常，额外的元数据也会以注释的形式存在，以给文本组件（如标签）添加更多的含义。以下是将在本章中探讨的一些主流文本预处理技术：

- 切分（tokenization）。
- 标注（tagging）。
- 分块（chunking）。
- 词干提取（stemming）。
- 词形还原（lemmatization）。

除了这些技术之外，你还需要执行一些基本操作，例如处理拼写错误的文本、删除停用词，以及根据需要处理其他不相关的成分。你需要记住一件重要的事情，一个强大的文本预处理系统始终是 NLP 和文本分析中所有程序的重要组成部分。其主要原因是在预处理之后获得的所有文本组件——无论是单词、短语、句子还是标识——形成了下一阶段程序输入的基本构件，这些程序执行更复杂的分析，包括学习模式和提取信息。因此，俗话说“垃圾输入，垃圾输出”在这里非常中肯，因为如果无法正确地处理文本，最终只能从应用程序和系统中获得多余和不相关的结果。

文本处理还有助于文本的清洗和标准化，这在文本分析系统中大有裨益，如可以提高分类器的准确度。我们还可以以注释的形式获取附加信息和元数据，它们可以提供有关文本的更多信息。在本章中，我们将使用清洗、删除无用标识、词干和词根的各种技术来研究文本规范化。

另一个重要内容是理解经过处理和规范化后的文本数据。这涉及重新审视第 1 章中所介

绍的语言语法和结构的相关概念，包括句子、短语、词性、浅层分析和语法。在本章中，我们将介绍实现这些概念并将其用于实际数据的方法。我们将遵循一个规整和确定的路径，从文本处理开始，逐步探索与之相关的各种概念和技术，并最终理解文本结构和语法。本书针对实际的开发人员，书中的代码和示例可以帮助你实现必要的工具和框架，也便于在实际处理过程中理解相关概念。

3.1 文本切分

第1章讨论了文本结构、成分和标识。具体来说，标识（token）是具有一定的句法语义且独立的最小文本成分。一段文本或一个文本文件具有几个组成部分，包括可以进一步细分为从句、短语和单词的语句。最流行的文本切分技术包括句子切分和词语切分，用于将文本语料库分解成句子，并将每个句子分解成单词。因此，文本切分可以定义为将文本数据分解或拆分为具有更小且有意义的成分（即标识）的过程。在下一节中，我们将介绍一些将文本切分为句子的方法。

3.1.1 句子切分

句子切分（sentence tokenization）是将文本语料库分解成句子的过程，这些句子是组成语料库的第一级切分结果。这个过程也称为*句子分割*，因为我们尝试将文本分割成有意义的句子。任何文本语料库都是文本的集合，其中每一段落都包含多个句子。

执行句子切分有多种技术，基本技术包括在句子之间寻找特定的分隔符，例如句号（.）、换行符（\n）或者分号（;）。我们将使用NLTK框架进行切分，该框架提供用于执行句子切分的各种接口。我们将主要关注以下句子切分器：

- `sent_tokenize`。
- `PunktSentenceTokenizer`。
- `RegexpTokenizer`。
- 预先训练的句子切分模型。

在将文本分割成句子之前，需要一些测试该操作的文本。我们将加载一些示例文本，以及部分在NLTK中可用的古腾堡（Gutenberg）语料库。你可以使用以下代码段加载必要的依存项：

```
import nltk
from nltk.corpus import gutenberg
from pprint import pprint

alice = gutenberg.raw(fileids='carroll-alice.txt')
sample_text = 'We will discuss briefly about the basic syntax, structure and
design philosophies. There is a defined hierarchical syntax for Python code
which you should remember when writing code! Python is a really powerful
programming language!'
```

我们可以使用以下代码段查看“*Alice in Wonderland*”语料库的长度及其前几行内容：

```
In [124]: # Total characters in Alice in Wonderland
     ...: print len(alice)
144395
```

```
In [125]: # First 100 characters in the corpus
     ...: print alice[0:100]
[Alice's Adventures in Wonderland by Lewis Carroll 1865]

CHAPTER I. Down the Rabbit-Hole

Alice was
```

nltk.sent_tokenize 函数是 nltk 推荐的默认的句子切分函数。它内部使用了一个 PunktSentenceTokenizer 类的实例。然而，它不仅仅是一个普通的对象或实例——它已经在几种语言模型上完成了预训练，并且在除英语外的许多主流语言上取得了良好的运行效果。

以下代码段展示了该函数在示例文本中的基本用法：

```
default_st = nltk.sent_tokenize
alice_sentences = default_st(text=alice)
sample_sentences = default_st(text=sample_text)

print 'Total sentences in sample_text:', len(sample_sentences)
print 'Sample text sentences :-'
pprint(sample_sentences)
print '\nTotal sentences in alice:', len(alice_sentences)
print 'First 5 sentences in alice:-'
pprint(alice_sentences[0:5])
```

运行上述代码段，你将得到以下输出，该输出给出句子总数以及这些句子在文本语料库中的模样：

```
Total sentences in sample_text: 3
Sample text sentences :-
['We will discuss briefly about the basic syntax, structure and design
philosophies.',
 'There is a defined hierarchical syntax for Python code which you should
remember when writing code!',
 'Python is a really powerful programming language!']

Total sentences in alice: 1625
First 5 sentences in alice:-
[u"[Alice's Adventures in Wonderland by Lewis Carroll 1865]\n\nCHAPTER I.",
 u"Down the Rabbit-Hole\n\nAlice was beginning to get very tired of sitting
by her sister on the\nbank, and of having nothing to do: once or twice she
had peeped into the\nbook her sister was reading, but it had no pictures
or conversations in\nit, 'and what is the use of a book,' thought Alice
'without pictures or\nconversation?'",
 u'So she was considering in her own mind (as well as she could, for the\nhot
day made her feel very sleepy and stupid), whether the pleasure\nof making
a daisy-chain would be worth the trouble of getting up and\npicking the
daisies, when suddenly a White Rabbit with pink eyes ran\nclose by her.',
 u"There was nothing so VERY remarkable in that; nor did Alice think it so\
nVERY much out of the way to hear the Rabbit say to itself, 'Oh dear!",
 u'Oh dear!']
```

现在，你应该可以看出，句子切分器其实是非常智能的，它不仅会使用句号来划分语句。它还会考虑到其他标点符号以及单词大小写。

我们也可以对其他语言的文本进行句子切分。如果正在处理德语文本，可以使用已经训练好的 sent_tokenize，或者在德语文本中加载一个预先训练好的切分模型到一个 PunktSentenceTokenizer 实例中并执行相同的操作。以下代码段显示了德语中的句子切分过程。

我们首先加载德语文本语料库并检查它：

```
In [4]: from nltk.corpus import europarl_raw
   ...:
   ...: german_text = europarl_raw.german.raw(fileids='ep-00-01-17.de')
   ...: # Total characters in the corpus
   ...: print len(german_text)
   ...: # First 100 characters in the corpus
   ...: print german_text[0:100]
157171

Wiederaufnahme der Sitzungsperiode Ich erkläre die am Freitag , dem 17.
Dezember unterbrochene Sit
```

然后，使用默认的 sent_tokenize 切分器和一个从 nltk 源加载的预训练的德语切分器来将文本语料库分割成句子：

```
In [5]: german_sentences_def = default_st(text=german_text,
language='german')
   ...:
   ...: # loading german text tokenizer into a PunktSentenceTokenizer
        instance
   ...: german_tokenizer = nltk.data.load(resource_url='tokenizers/punkt/
        german.pickle')
   ...: german_sentences = german_tokenizer.tokenize(german_text)
   ...:
   ...: # verify the type of german_tokenizer
   ...: # should be PunktSentenceTokenizer
   ...: print type(german_tokenizer)
<class 'nltk.tokenize.punkt.PunktSentenceTokenizer'>
```

由此可以看出 german_tokenizer 是 PunktSentenceTokenizer 的一个实例，它专门用来处理德语。

接下来，我们对比从默认切分器获得的句子是否与从预训练切分器获得的句子相同，理想情况下应为 True。之后，显示部分示例句子的切分结果：

```
In [9]: print german_sentences_def == german_sentences
   ...: # print first 5 sentences of the corpus
   ...: for sent in german_sentences[0:5]:
   ...:     print sent
True

Wiederaufnahme der Sitzungsperiode Ich erkläre die am Freitag , dem 17.
Dezember unterbrochene Sitzungsperiode des Europäischen Parlaments für
wiederaufgenommen , wünsche Ihnen nochmals alles Gute zum Jahreswechsel und
hoffe , daß Sie schöne Ferien hatten .
Wie Sie feststellen konnten , ist der gefürchtete " Millenium-Bug " nicht
eingetreten .
Doch sind Bürger einiger unserer Mitgliedstaaten Opfer von schrecklichen
Naturkatastrophen geworden .
Im Parlament besteht der Wunsch nach einer Aussprache im Verlauf dieser
Sitzungsperiode in den nächsten Tagen .
Heute möchte ich Sie bitten - das ist auch der Wunsch einiger Kolleginnen
und Kollegen - , allen Opfern der Stürme , insbesondere in den verschiedenen
Ländern der Europäischen Union , in einer Schweigeminute zu gedenken .
```

从结果可以看出前面的假设是正确的，你可以用两种方式来切分英语之外的语言句子。使用默认的 PunktSentenceTokenizer 类也能很方便地实现句子切分，如下所示：

```
In [11]: punkt_st = nltk.tokenize.PunktSentenceTokenizer()
    ...: sample_sentences = punkt_st.tokenize(sample_text)
    ...: pprint(sample_sentences)
['We will discuss briefly about the basic syntax, structure and design
philosophies.',
 'There is a defined hierarchical syntax for Python code which you should
remember when writing code!',
 'Python is a really powerful programming language!']
```

可以看到，我们得到了与预期一致的输出。在句子切分这部分知识中，我们要介绍的最后一个是使用 RegexpTokenizer 类的实例将文本切分为句子，我们将使用基于正则表达式的模式来切分句子。如果你需要唤醒之前的记忆，那么请回想一下上一章的正则表达式（regex）。以下代码段显示了如何使用正则表达式来分割句子：

```
In [29]: SENTENCE_TOKENS_PATTERN = r'(?<!\w\.\w.)(?<![A-Z][a-z]\.)
(?<![A-Z]\.)(?<=\.|\?|\!)\s'
    ...: regex_st = nltk.tokenize.RegexpTokenizer(
    ...:             pattern=SENTENCE_TOKENS_PATTERN,
    ...:             gaps=True)
    ...: sample_sentences = regex_st.tokenize(sample_text)
    ...: pprint(sample_sentences)
['We will discuss briefly about the basic syntax, structure and design
philosophies.',
 ' There is a defined hierarchical syntax for Python code which you should
remember  when writing code!',
 'Python is a really powerful programming language!']
```

通过上面的输出可以看出，我们获得的切分结果与使用其他切分器进行切分的结果相同。以上介绍给予了关于使用不同的 nltk 接口将文本切分成句子的大致概念。在下一节中，我们将使用几种技术将这些句子分割成单词。

3.1.2 词语切分

词语切分（word tokenization）是将句子分解或分割成其组成单词的过程。句子是单词的集合，通过词语切分，在本质上，将一个句子分割成单词列表，该单词列表又可以重建句子。词语切分在很多过程中都是非常重要的，特别是在文本清洗和规范化时，诸如词干提取和词形还原这类基于词干、标识信息的操作会在每个单词上实施。与句子切分类似，nltk 为词语切分提供了各种有用的接口。本节将介绍以下主流接口：

- word_tokenize。
- TreebankWordTokenizer。
- RegexpTokenizer。
- 从 RegexpTokenizer 继承的切分器 。

我们将使用例句“The brown fox wasn’ t that quick and he couldn’ t win the race”作为各种切分器的输入。nltk. word_tokenize 函数是 nltk 默认并推荐的词语切分器。该切分器实际上是 TreebankWordTokenizer 类的一个实例或对象，并且是该核心类的一个封装。以下代码段可以说明其用法：

```
 In [114]: sentence = "The brown fox wasn't that quick and he couldn't win
the race"
     ...:
     ...: default_wt = nltk.word_tokenize
```

```
     ...: words = default_wt(sentence)
     ...: print words
['The', 'brown', 'fox', 'was', "n't", 'that', 'quick', 'and', 'he', 'could',
"n't", 'win', 'the', 'race']
```

TreebankWordTokenizer 基于 Penn Treebank，并使用各种正则表达式来分割文本。当然，这里的一个主要假设是我们已经预先执行了句子切分。Penn Treebank 使用的原始切分器是一个 sed 脚本，你可以在 https://catalog. ldc. upenn. edu/ldc99t42 上下载它，从而了解句子切分为单词的简要模式。该切分器的一些主要功能包括：

- 分割和分离出现在句子末尾的句点。
- 分割和分离空格前的逗号和单引号。
- 将大多数标点符号分割成独立标识。
- 分割常规的缩写词——例如将“don’ t”分割成“do”和“n’ t”。

以下代码段展示了 TreebankWordTokenizer 在词语切分中的用法：

```
In [117]: treebank_wt = nltk.TreebankWordTokenizer()
     ...: words = treebank_wt.tokenize(sentence)
     ...: print words
['The', 'brown', 'fox', 'was', "n't", 'that', 'quick', 'and', 'he', 'could',
"n't", 'win', 'the', 'race']
```

可以看出，正如所预期的那样，上述代码段的输出与 word_tokenize()的输出相似，因为它们使用了相同的分词机制。

现在来看看如何使用正则表达式和 RegexpTokenizer 类切分句子。请记住，在词语切分中有两个主要参数：pattern 参数和 gaps 参数。pattern 参数用于构建切分器；gaps 参数如果设置为 True，用于查找标识之间的间隙。否则，它用于查找标识本身。

以下代码段显示了一些使用正则表达式执行词语切分的示例：

```
# pattern to identify tokens themselves
In [127]: TOKEN_PATTERN = r'\w+'
     ...: regex_wt = nltk.RegexpTokenizer(pattern=TOKEN_PATTERN,
     ...:                                 gaps=False)
     ...: words = regex_wt.tokenize(sentence)
     ...: print words
['The', 'brown', 'fox', 'wasn', 't', 'that', 'quick', 'and', 'he', 'couldn',
't', 'win', 'the', 'race']
# pattern to identify gaps in tokens
In [128]: GAP_PATTERN = r'\s+'
     ...: regex_wt = nltk.RegexpTokenizer(pattern=GAP_PATTERN,
     ...:                                 gaps=True)
     ...: words = regex_wt.tokenize(sentence)
     ...: print words
['The', 'brown', 'fox', "wasn't", 'that', 'quick', 'and', 'he', "couldn't",
'win', 'the', 'race']
# get start and end indices of each token and then print them
In [131]: word_indices = list(regex_wt.span_tokenize(sentence))
     ...: print word_indices
     ...: print [sentence[start:end] for start, end in word_indices]
[(0, 3), (4, 9), (10, 13), (14, 20), (21, 25), (26, 31), (32, 35), (36, 38),
(39, 47), (48, 51), (52, 55), (56, 60)]
['The', 'brown', 'fox', "wasn't", 'that', 'quick', 'and', 'he', "couldn't",
'win', 'the', 'race']
```

除了基础的 RegexpTokenizer 类之外，还有几个派生类可以执行不同类型的词语切分。

WordPunktTokenizer 使用 r'\w + |[^\w\s] +'模式将句子切分成独立的字母和非字母标识。WhitespaceTokenizer 基于诸如缩进符、换行符及空格的空白字符将句子分割成单词。

以下代码说明了上述派生类的用法：

```
In [132]: wordpunkt_wt = nltk.WordPunctTokenizer()
     ...: words = wordpunkt_wt.tokenize(sentence)
     ...: print words
['The', 'brown', 'fox', 'wasn', "'", 't', 'that', 'quick', 'and', 'he',
'couldn', "'", 't', 'win', 'the', 'race']
In [133]: whitespace_wt = nltk.WhitespaceTokenizer()
     ...: words = whitespace_wt.tokenize(sentence)
     ...: print words
['The', 'brown', 'fox', "wasn't", 'that', 'quick', 'and', 'he', "couldn't",
'win', 'the', 'race']
```

以上就是关于词语切分的介绍。现在我们知道了如何将原始文本分割成句子和单词，下一节中将以此为基础，将这些标识规范化，以获得更容易理解、解释以及在 NLP 和机器学习中使用的清晰、标准的文本数据。

3.2 文本规范化

文本规范化定义为这样的一个过程，它包含一系列步骤，依次是转换、清洗以及将文本数据标准化成可供 NLP、分析系统和应用程序使用的格式。通常，文本切分本身也是文本规范化的一部分。除了文本切分以外，还有各种其他技术，包括文本清洗、大小写转换、词语校正、停用词删除、词干提取和词形还原。文本规范化也常常称为文本清洗或转换。

本节将讨论在文本规范化过程中使用的各种技术。在探索各种技术之前，请使用以下代码段来加载基本的依存关系以及将使用的语料库：

```
import nltk
import re
import string
from pprint import pprint

corpus = ["The brown fox wasn't that quick and he couldn't win the race",
          "Hey that's a great deal! I just bought a phone for $199",
          "@@You'll (learn) a **lot** in the book. Python is an amazing
          language!@@"]
```

3.2.1 文本清洗

通常，我们要使用或分析的文本数据都包含大量无关和不必要的标识和字符，在进行其他操作（如切分和其他规范化操作）之前，应该先删除它们。这包括从如 HTML 之类的数据源中提取有意义的文本，数据源中可能包含不必要的 HTML 标记，甚至是来自 XML 和 JSON feed 的数据。解析并清洗这些数据的方法很多，以删除不必要的标签。你可以使用 nltk 的 clean_html()函数，甚至是 BeautifulSoup 库来解析 HTML 数据。你还可以使用自定义的逻辑，包括正则表达式、xpath 和 lxml 库来解析 XML 数据。从 JSON 获取数据较为容易，因为它具有明确的键值注释。

3.2.2 文本切分

通常，在删除数据中多余字符和符号操作的前后，进行文本切分操作。文本切分和删除多余字符的顺序取决于你要解决的问题和你正在处理的数据。上一节已经介绍了各种切分技术，这里会定义一个通用的切分函数，并在前面提到的语料库中运行它。

以下代码段定义了文本切分函数：

```
def tokenize_text(text):
    sentences = nltk.sent_tokenize(text)
    word_tokens = [nltk.word_tokenize(sentence) for sentence in sentences]
    return word_tokens
```

这个函数的功能是接收文本数据，再从中提取句子，最后将每个句子划分成标识，这些标识可以是单词、特殊字符或标点符号。以下代码段说明了该函数的功能：

```
In [297]: token_list = [tokenize_text(text)
     ...:               for text in corpus]
     ...: pprint(token_list)
[[['The', 'brown', 'fox', 'was', "n't", 'that', 'quick', 'and', 'he',
          'could', "n't",
   'win', 'the', 'race']],
 [['Hey', 'that', "'s", 'a', 'great', 'deal', '!'],
  ['I', 'just', 'bought', 'a', 'phone', 'for', '$', '199']],
 [['@', '@', 'You', "'ll", '(', 'learn', ')', 'a', '**lot**', 'in', 'the',
       'book', '.'],
  ['Python', 'is', 'an', 'amazing', 'language', '!'],
  ['@', '@']]]
```

现在，你可以看出每个语料库中的文本是如何被切分的，你也可以在更多的文本数据上尝试一下切分函数，看看能不能改进它！

3.2.3 删除特殊字符

文本规范化中的一个重要任务是删除多余和特殊的字符，诸如特殊符号或标点符号。这个步骤通常在切分操作前后进行。这样做的主要原因是，当我们分析文本并提取基于 NLP 和机器学习的特征或信息时，标点符号或特殊字符往往没有多大的意义。我们将在切分前后删除这两类特殊的字符。以下代码段显示了如何在切分之后删除特殊字符：

```
def remove_characters_after_tokenization(tokens):
    pattern = re.compile('[{}]'.format(re.escape(string.punctuation)))
    filtered_tokens = filter(None, [pattern.sub('', token) for token in tokens])
    return filtered_tokens
In [299]: filtered_list_1 =  [filter(None,[remove_characters_after_
                             tokenization(tokens)
     ...:                                  for tokens in sentence_tokens])
     ...:                     for sentence_tokens in token_list]
     ...: print filtered_list_1
[[['The', 'brown', 'fox', 'was', 'nt', 'that', 'quick', 'and', 'he',
'could', 'nt', 'win', 'the', 'race']], [['Hey', 'that', 's', 'a', 'great',
'deal'], ['I', 'just', 'bought', 'a', 'phone', 'for', '199']], [['You',
'll', 'learn', 'a', 'lot', 'in', 'the', 'book'], ['Python', 'is', 'an',
'amazing', 'language']]]
```

本质上，我们在这里使用的是 `string.punctuation` 属性，它由所有可能的特殊字符/符号组成，并从中创建一个正则表达式模式。我们使用它来匹配并删除符号和字符标识。使

用正则表达式 sub 算法删除特殊字符以后，可以使用 filter 函数删除空标识。

以下代码段展示了如何在文本切分之前删除特殊字符：

```
def remove_characters_before_tokenization(sentence,
                                          keep_apostrophes=False):
    sentence = sentence.strip()
    if keep_apostrophes:
        PATTERN = r'[?|$|&|*|%|@|(|)|~]' # add other characters here to
        remove them
        filtered_sentence = re.sub(PATTERN, r'', sentence)
    else:
        PATTERN = r'[^a-zA-Z0-9 ]' # only extract alpha-numeric characters
        filtered_sentence = re.sub(PATTERN, r'', sentence)
    return filtered_sentence

In [304]: filtered_list_2 = [remove_characters_before_tokenization(sentence)
    ...:                     for sentence in corpus]
    ...: print filtered_list_2
['The brown fox wasnt that quick and he couldnt win the race', 'Hey thats a
great deal I just bought a phone for 199', 'Youll learn a lot in the book
Python is an amazing language']

In [305]: cleaned_corpus = [remove_characters_before_tokenization(sentence,
                           keep_apostrophes=True)
    ...:                    for sentence in corpus]
    ...: print cleaned_corpus
["The brown fox wasn't that quick and he couldn't win the race", "Hey that's
a great deal! I just bought a phone for 199", "You'll learn a lot in the
book. Python is an amazing language!"]
```

上面的输出显示了在切分前删除特殊字符的两种不同的方式——一种是删除所有特殊字符，另一种是保留撇号[⊖]和句号——这两种方法均使用正则表达式。至此，你一定已经意识到，正如第 2 章提到的，正则表达式是非常强大的工具。通常，在删除这些字符后，你就可以对干净的文本进行切分或进行其他规范化操作了。有时候，我们会想要保留句子中的撇号作为跟踪文本的方式，并在需要的时候扩展它们，下一节将探讨这些内容。

3.2.4 扩展缩写词

缩写词（contraction）是词或音节的缩短形式。它们既在书面形式中存在，也在口语中存在。现有单词的缩短版本可以通过删除特定的字母和音节获得。在英语的缩写形式中，缩写词通常是从单词中删除一些元音来创建的。举例来说“is not”缩写成“isn’t”，“will not”缩写成“won’t”，你应该已经注意到了，缩写词中撇号用来表示缩写，而一些元音和其他字母则被删除了。通常，在正式书写时会避免使用缩写词，但在非正式情况下，它们被广泛使用。英语中存在各种形式的缩写词，这些形式的缩写词与助动词的类型相关，不同的助动词给出了常规缩写词、否定缩写词和其他特殊的口语缩写词（其中一些可能并不涉及助动词）。

缩写词确实为 NLP 和文本分析制造了一个难题，首先因为在该单词中有一个特殊的撇号字符。此外，我们有两个甚至更多的单词由缩写词表示，当尝试执行词语切分或者词语标准化时，这就会引发一连串复杂的问题。因此，在处理文本时，需要一些确切的步骤来处理缩写词。理想情况下，你可以对缩写词和对应的扩展词语进行适当的映射，然后使用映射关

⊖ 英文中的上撇号如 isn't 中的撇号。——译者注

系扩展文本中的所有缩写词。我创建了一个缩写词及其扩展形式的词汇表，你可以在 Python 库中的 contractions.py 文件中访问它们（可与本章的代码文件一起使用）。部分缩写词汇表显示在以下代码段中：

```
CONTRACTION_MAP = {
"isn't": "is not",
"aren't": "are not",
"can't": "cannot",
"can't've": "cannot have",
.
.
.
"you'll've": "you will have",
"you're": "you are",
"you've": "you have"
}
```

请记住，一些缩写词可以对应多种形式。举例来说，缩写词“you' ll”可以表示“you will”或者“you shall”。为了简化，我为每个缩写词选取了一个扩展形式。下一步，想要扩展缩写词，你需要使用以下代码段：

```
from contractions import CONTRACTION_MAP

def expand_contractions(sentence, contraction_mapping):
    contractions_pattern = re.compile('({})'.format('|'.join(contraction_
    mapping.keys())),
                                      flags=re.IGNORECASE|re.DOTALL)
    def expand_match(contraction):
        match = contraction.group(0)
        first_char = match[0]
        expanded_contraction = contraction_mapping.get(match)\
                               if contraction_mapping.get(match)\
                               else contraction_mapping.get(match.lower())
        expanded_contraction = first_char+expanded_contraction[1:]
        return expanded_contraction

    expanded_sentence = contractions_pattern.sub(expand_match, sentence)
    return expanded_sentence
```

上面代码段中的主函数 expand_ contractions 使用了 expand_match 函数来查找与正则表达式模式相匹配的每个缩写词，这些正则表达式模式由 CONTRACTION_MAP 库中的缩写词构成。匹配缩写词后，我们用相应的扩展版本替换它，并保留非缩写形式的单词。

在上节获得的文本 cleaned_corpus 上应用该函数，如下所示：

```
In [311]: expanded_corpus = [expand_contractions(sentence, CONTRACTION_MAP)
     ...:                    for sentence in cleaned_corpus]
     ...: print expanded_corpus
['The brown fox was not that quick and he could not win the race', 'Hey that
is a great deal! I just bought a phone for 199', 'You will learn a lot in
the book. Python is an amazing language!']
```

从结果可以看出，正如所预期的那样，每个缩写词都被正确地扩展了。你可以构建一个更好的缩写词扩展器吗？尝试一下！这是一个非常有趣又有难度的课题。

3.2.5 大小写转换

通常，我们会希望修改单词或句子的大小写，以使诸如特殊单词或标识匹配的工作更加

容易。通常有两种类型的大小写转换操作，即小写转换和大写转换，通过这两种操作可以将文本正文完全转换为小写或大写。当然也有其他大小写形式，如句式大小写（sentence case）或单词首字母大小写（proper case）。小写字体是一种格式，其中所有文本的字母都是小写字母，大写字体格式则全部是大写字母。

以下代码段说明了以上概念：

```
# lower case
In [315]: print corpus[0].lower()
the brown fox wasn't that quick and he couldn't win the race
# upper case
In [316]: print corpus[0].upper()
THE BROWN FOX WASN'T THAT QUICK AND HE COULDN'T WIN THE RACE
```

3.2.6 删除停用词

停用词（stopword，有时也拼写成 stop word）是指没有或只有极小意义的词语。通常在处理过程中将它们从文本中删除，以保留具有最大意义及语境的词语。如果你基于单个标识聚合语料库，然后检查词语频率，就会发现停用词的出现频率是最高的。类似“a”“the”“me”和“and so on”这样的单词就是停用词。目前还没有普遍或已穷尽的停用词列表。每个领域或语言可能都有一系列独有的停用词。以下代码段展示了一种过滤和删除英语停用词的方法：

```
def remove_stopwords(tokens):
    stopword_list = nltk.corpus.stopwords.words('english')
    filtered_tokens = [token for token in tokens if token not in stopword_
    list]
    return filtered_tokens
```

在前面的函数中，我们使用了 nltk，它有一个英文的停用词列表。我们使用它来过滤掉所有与停用词相对应的标识。我们使用 tokenize_text 函数来分割在上一节中获得的 expanded_corpus，然后使用前面的函数删除停用词：

```
In [332]: expanded_corpus_tokens = [tokenize_text(text)
     ...:                           for text in expanded_corpus]
     ...: filtered_list_3 =  [[remove_stopwords(tokens)
     ...:                      for tokens in sentence_tokens]
     ...:                      for sentence_tokens in expanded_corpus_
                               tokens]
     ...: print filtered_list_3
[[['The', 'brown', 'fox', 'quick', 'could', 'win', 'race']], [['Hey',
'great', 'deal', '!'], ['I', 'bought', 'phone', '199']], [['You', 'learn',
'lot', 'book', '.'], ['Python', 'amazing', 'language', '!']]]
```

与之前的输出结果相比，本次输出的标识明显减少了，通过比较，你可以发现被删除的标识都是停用词。想要查看 nltk 库中所有的英文停用词汇列表，请打印 nltk.corpus.stopwords.words('english') 的内容。请记住一个重要的事情，就是在上述情况下（在第一句话中），删除了诸如“no”和“not”这样的否定词，通常，这类词语应当保留，以便于在诸如情绪分析等应用中句子语意不会失真，因此在这些应用中你需要确保此类词语不会被删除。

3.2.7 词语校正

文本规范化面临的主要挑战之一是文本中存在不正确的单词。这里不正确的定义包括

拼写错误的单词以及某些字母过多重复的单词。举例来说，“finally”一词可能会被错误地写成“fianlly”，或者被想要表达强烈情绪的人写成“finallllyyyyyy”。我们的主要目标是将这些单词标准化为正确形式，使我们不至于失去文本中的重要信息。本节将介绍处理重复字符以及校正拼写错误的方法。

1. 校正重复字符

在这里，我们将介绍一种语法和语义组合使用的拼写校正方法。首先，从校正这些单词的语法开始，然后转向语义。

算法的第一步是，使用正则表达式来识别单词中的重复字符，然后使用置换来逐个删除重复字符。考虑前面例子中“finalllyyy”一词，可以使用模式 `r'(\w* )(\w)\2(\w* )'`来识别单词中在两个不同字符之间的重复字符。通过利用正则表达式匹配组（组1、2和3）并使用模式 `r'\1\2\3'`，能够使用置换方法消除一个重复字符，然后迭代此过程，直到消除所有重复字符。

以下代码段说明了上述过程：

```
In [361]: old_word = 'finalllyyy'
     ...: repeat_pattern = re.compile(r'(\w*)(\w)\2(\w*)')
     ...: match_substitution = r'\1\2\3'
     ...: step = 1
     ...:
     ...: while True:
     ...:     # remove one repeated character
     ...:     new_word = repeat_pattern.sub(match_substitution,
     ...:                                   old_word)
     ...:     if new_word != old_word:
     ...:         print 'Step: {} Word: {}'.format(step, new_word)
     ...:         step += 1 # update step
     ...:         # update old word to last substituted state
     ...:         old_word = new_word
     ...:         continue
     ...:     else:
     ...:         print "Final word:", new_word
     ...:         break
     ...:
Step: 1 Word: finalllyy
Step: 2 Word: finallly
Step: 3 Word: finally
Step: 4 Word: finaly
Final word: finaly
```

上面的代码段显示了重复字符如何逐步被删除，直到我们得到最终的单词“finaly”。然而，在语义上，这个词是不正确的——正确的词是“finally”，即在步骤3中获得的单词。现在将使用WordNet语料库来检查每个步骤得到的单词，一旦获得有效单词就立即终止循环。这就引入了算法所需的语义校正，如下面的代码段所示：

```
In [363]: from nltk.corpus import wordnet
     ...: old_word = 'finalllyyy'
     ...: repeat_pattern = re.compile(r'(\w*)(\w)\2(\w*)')
     ...: match_substitution = r'\1\2\3'
     ...: step = 1
     ...:
     ...: while True:
     ...:     # check for semantically correct word
     ...:     if wordnet.synsets(old_word):
```

```
     ...:         print "Final correct word:", old_word
     ...:         break
     ...:     # remove one repeated character
     ...:     new_word = repeat_pattern.sub(match_substitution,
     ...:                                   old_word)
     ...:     if new_word != old_word:
     ...:         print 'Step: {} Word: {}'.format(step, new_word)
     ...:         step += 1 # update step
     ...:         # update old word to last substituted state
     ...:         old_word = new_word
     ...:         continue
     ...:     else:
     ...:         print "Final word:", new_word
     ...:         break
     ...:
Step: 1 Word: finalllyy
Step: 2 Word: finallly
Step: 3 Word: finally
Final correct word: finally
```

从上面的代码段中可以看出，在第3步后代码终止了，我们获得了正确的、符合语法和语义的单词。

可以通过将该逻辑编写到函数中来构建一个更好的代码段，以便使其在校正词语时变得更为通用，如下面的代码段所示：

```
from nltk.corpus import wordnet

def remove_repeated_characters(tokens):
    repeat_pattern = re.compile(r'(\w*)(\w)\2(\w*)')
    match_substitution = r'\1\2\3'
    def replace(old_word):
        if wordnet.synsets(old_word):
            return old_word
        new_word = repeat_pattern.sub(match_substitution, old_word)
        return replace(new_word) if new_word != old_word else new_word

    correct_tokens = [replace(word) for word in tokens]
    return correct_tokens
```

如前所述，该代码段使用内部函数`replace()`来实现我们的算法，然后在外部函数`remove_repeated_characters()`中对句子中的每个标识重复调用它。

可以在下面的代码段中看到上述代码的实际运行情况，以下代码段包含了一个实际的例句：

```
In [369]: sample_sentence = 'My schooool is reallllllyyy amaaazingggg'
     ...: sample_sentence_tokens = tokenize_text(sample_sentence)[0]
     ...: print sample_sentence_tokens
['My', 'schooool', 'is', 'reallllllyyy', 'amaaazingggg']

In [370]: print remove_repeated_characters(sample_sentence_tokens)
['My', 'school', 'is', 'really', 'amazing']
```

从上面的输出可以看出，函数执行过程符合预期，它替换了每个标识中的重复字符，然后按照要求给出了正确的标识。

2. 校正拼写错误

我们面临的另一个问题是由人为错误导致的拼写错误，甚至是由于自动更正文本等功

能导致的机器拼写错误。有多种处理拼写错误的方法，其最终目标都是获得拼写正确的文本标识。本节将介绍最为著名的算法之一，它由谷歌研究主管 Peter Norvig 开发。你可以在 http://norvig.com/spell-correct.html 上找到完整详细的算法说明。

我们的主要目标是，给出一个单词，找到这个单词最有可能的正确形式。我们遵循的方法是生成一系列类似输入词的候选词，并从该集合中选择最有可能的单词作为正确的单词。我们使用标准英文单词语料库，根据语料库中单词的频率，从距离输入单词最近的最后一组候选词中识别出正确的单词。这个距离（即一个单词与输入单词的测量距离）也称为编辑距离（edit distance）。我们使用的输入语料库包含 Gutenberg 语料库书籍、维基词典和英国国家语料库中的最常用单词列表。你可以在本章的代码资源中找到这个命名为 big.txt 的文件，或者从 http://norvig.com/big.txt 下载它。

可以使用以下代码段来生成英文中最常出现的单词及其计数：

```
import re, collections

def tokens(text):
    """
    Get all words from the corpus
    """
    return re.findall('[a-z]+', text.lower())

WORDS = tokens(file('big.txt').read())
WORD_COUNTS = collections.Counter(WORDS)

# top 10 words in the corpus
In [407]: print WORD_COUNTS.most_common(10)
[('the', 80030), ('of', 40025), ('and', 38313), ('to', 28766), ('in',
22050), ('a', 21155), ('that', 12512), ('he', 12401), ('was', 11410),
('it', 10681)]
```

拥有了自己的词汇之后，就可以定义三个函数，计算出与输入单词的编辑距离为 0、1 和 2 的单词组。这些编辑距离由插入、删除、添加和调换位置等操作产生。以下代码段定义了实现该功能的函数：

```
def edits0(word):
    """
    Return all strings that are zero edits away
    from the input word (i.e., the word itself).
    """
    return {word}

def edits1(word):
    """
    Return all strings that are one edit away
    from the input word.
    """
    alphabet = 'abcdefghijklmnopqrstuvwxyz'
    def splits(word):
        """
        Return a list of all possible (first, rest) pairs
        that the input word is made of.
        """
        return [(word[:i], word[i:])
                for i in range(len(word)+1)]
    pairs      = splits(word)
    deletes    = [a+b[1:]           for (a, b) in pairs if b]
```

```
    transposes = [a+b[1]+b[0]+b[2:] for (a, b) in pairs if len(b) > 1]
    replaces   = [a+c+b[1:]         for (a, b) in pairs for c in alphabet
                                    if b]
    inserts    = [a+c+b             for (a, b) in pairs for c in alphabet]
    return set(deletes + transposes + replaces + inserts)

def edits2(word):
    """Return all strings that are two edits away
    from the input word.
    """
    return {e2 for e1 in edits1(word) for e2 in edits1(e1)}
```

我们还可以定义一个 known() 函数，该函数根据单词是否存在于词汇词典 WORD_COUNTS 中，从 edit 函数得出的候选词组中返回一个单词子集。这使我们可以从候选词组中获得一个有效单词列表：

```
def known(words):
    """
    Return the subset of words that are actually
    in our WORD_COUNTS dictionary.
    """
    return {w for w in words if w in WORD_COUNTS}
```

从下面的代码段中可以看出，这些函数完成了输入单词的拼写校正。基于与输入单词之间的编辑距离，它给出了最有可能的候选词：

```
# input word
In [409]: word = 'fianlly'

# zero edit distance from input word
In [410]: edits0(word)
Out[410]: {'fianlly'}
# returns null set since it is not a valid word
In [411]: known(edits0(word))
Out[411]: set()

# one edit distance from input word
In [412]: edits1(word)
Out[412]:
{'afianlly',
 'aianlly',
  .
  .
 'yianlly',
 'zfianlly',
 'zianlly'}
# get correct words from above set
In [413]: known(edits1(word))
Out[413]: {'finally'}

# two edit distances from input word
In [417]: edits2(word)
Out[417]:
{'fchnlly',
 'fianjlys',
   .
   .
 'fiapgnlly',
 'finanlqly'}
# get correct words from above set
In [418]: known(edits2(word))
Out[418]: {'faintly', 'finally', 'finely', 'frankly'}
```

上面的输出显示了一组能够替换错误输入词的候选词。通过赋予编辑距离更小的单词更高的优先级，可以从前面的列表中选出候选词，如下列代码段所示：

```
In [420]: candidates = (known(edits0(word)) or
     ...:               known(edits1(word)) or
     ...:               known(edits2(word)) or
     ...:               [word])

In [421]: candidates
Out[421]: {'finally'}
```

假如在前面的候选词中两个单词的编辑距离相同，则可以通过使用 `max (candidates, key = WORD_COUNTS. get)` 函数从词汇字典 `WORD_COUNTS` 中选取出现频率最高的词来作为有效词。现在，我们使用上述逻辑定义拼写校正函数：

```
def correct(word):
    """
    Get the best correct spelling for the input word
    """
    # Priority is for edit distance 0, then 1, then 2
    # else defaults to the input word itself.
    candidates = (known(edits0(word)) or
                  known(edits1(word)) or
                  known(edits2(word)) or
                  [word])
    return max(candidates, key=WORD_COUNTS.get)
```

可以对拼写错误的单词使用上述函数来直接校正它们，如下面的代码段所示：

```
In [438]: correct('fianlly')
Out[438]: 'finally'

In [439]: correct('FIANLLY')
Out[439]: 'FIANLLY'
```

可以看出这个函数对大小写比较敏感，它无法校正非小写的单词，因此我们编写下列函数，以使其能够同时校正大写和小写的单词。该函数的逻辑是存储单词的原始大小写格式，然后将所有字母转换为小写字母，更正拼写错误，最后使用 `case_of` 函数将其重新转换回初始的大小写格式：

```
def correct_match(match):
    """
    Spell-correct word in match,
    and preserve proper upper/lower/title case.
    """

    word = match.group()
    def case_of(text):
        """
        Return the case-function appropriate
        for text: upper, lower, title, or just str.:
            """
        return (str.upper if text.isupper() else
                str.lower if text.islower() else
                str.title if text.istitle() else
                str)
    return case_of(word)(correct(word.lower()))

def correct_text_generic(text):
```

```
    """
    Correct all the words within a text,
    returning the corrected text.
    """
    return re.sub('[a-zA-Z]+', correct_match, text)
```

现在，上述函数既可以用来校正大写单词，也可以用来校正小写单词，如下面的代码段所示：

```
In [441]: correct_text_generic('fianlly')
Out[441]: 'finally'
In [442]: correct_text_generic('FIANLLY')
Out[442]: 'FINALLY'
```

当然，这种方法并不总是准确的，如果单词没有出现在词汇字典中，就有可能无法被校正。使用更多的词汇表数据以涵盖更多的词语可以解决这个问题。在 pattern 库中也有类似的、开箱即用的算法，如下面的代码段所示：

```
from pattern.en import suggest

# test on wrongly spelt words
In [184]: print suggest('fianlly')
 [('finally', 1.0)]

In [185]: print suggest('flaot')
 [('flat', 0.85), ('float', 0.15)]
```

除此之外，Python 中还提供了几个强大的库，包括 PyEnchant 库，它基于 enchant 库（更多信息请参见 http://pythonhosted.org/pyenchant/），以及 aspell-python 库，它是目前很流行的 GNU Aspell 的一个 Python 封装。欢迎查看并使用它们来校正单词拼写！

3.2.8 词干提取

想要理解词干提取的过程需要先理解词干（stem）的含义。第 1 章中谈到词素，它是任何自然语言中最小的独立单元。词素由词干和词缀（affixe）组成。词缀是指前缀、后缀等词语单元，它们附加到词干上以改变其含义或创建一个新单词。词干也经常称为单词的基本形式，我们可以通过在词干上添加词缀来创建新词，这个过程称为词形变化。相反的过程是从单词的变形形式中获得单词的基本形式，这称为词干提取。

以“JUMP”一词为例，你可以对其添加词缀形成新的单词，如“JUMPS”“JUMPED”和“JUMPING”。在这些情况下，基本单词“JUMP”是词干。如果对这三种变形形式中的任一种进行词干提取，都将得到基本形式，如图 3-1 所示。

上图显示了词干在所有变形中是如何存在的，它构建了一个基础，每个词形变化都是在其上添加词缀构成的。词干提取帮助我们将词语标准化到其基础词干而不用考虑其词形变化，这对于许多应用程序大有裨益，如文本分类或聚类以及信息检索。搜索引擎广泛使用这些技术来提供更好、更准确的结果，而无需考虑单词的形式。

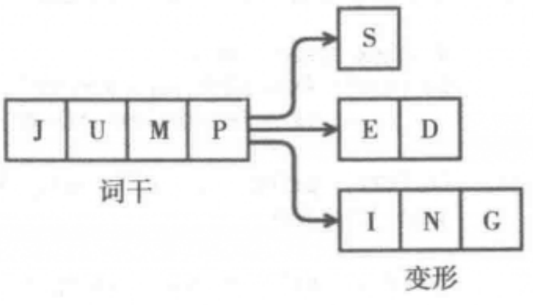


图 3-1 词干及词形变化

对于词干提取器，nltk 包有几种实现算法。这些

词干提取器包含在 stem 模块中，该模块继承了 nltk.stem.api 模块中的 StemmerI 接口。你甚至可以使用这个类（严格来说，它是一个接口）作为你的基类来创建自己的词干提取器。目前，最受欢迎的词干提取器之一是波特词干提取器，它基于其发明人马丁·波特（Martin Porter）博士所开发的算法。其原始算法拥有 5 个不同的阶段，用于减少变形和提取词干，其中每个阶段都有自己的一套规则。此外，还有一个 Porter2 词干提取算法，它是波特博士在原始算法基础上提出的改进算法。以下代码段展示了波特词干提取器：

```
# Porter Stemmer
In [458]: from nltk.stem import PorterStemmer
     ...: ps = PorterStemmer()

In [459]: print ps.stem('jumping'), ps.stem('jumps'), ps.stem('jumped')
jump jump jump

In [460]: print ps.stem('lying')
lie

In [461]: print ps.stem('strange')
strang
```

兰卡斯特词干提取器（Lancaster stemmer）基于兰卡斯特词干算法，通常也称为佩斯/哈斯科词干提取器（Paice/Husk stemmer），由克里斯·D·佩斯（Chris D. Paice）提出。该词干提取器是一个迭代提取器，具有超过 120 条规则来具体说明如何删减或替换词缀以获得词干。以下代码段显示了兰卡斯特词干提取器的用法：

```
# Lancaster Stemmer
In [465]: from nltk.stem import LancasterStemmer
     ...: ls = LancasterStemmer()

In [466]: print ls.stem('jumping'), ls.stem('jumps'), ls.stem('jumped')
jump jump jump

In [467]: print ls.stem('lying')
lying

In [468]: print ls.stem('strange')
strange
```

你可以看出，这个词干提取器的行为与波特词干提取器的行为是不相同的。

还有一些其他的词干提取器，包括 RegexpStemmer，它使你可以根据用户定义的规则构建自己的词干提取器，还有 SnowballStemmer，它除了支持英语外，还支持其他 13 种不同的语言。

以下代码段显示了如何使用 RegexpStemmer 执行词干提取。RegexpStemmer 使用正则表达式来识别词语中的形态学词缀，并且删除与之匹配的任何部分：

```
# Regex based stemmer
In [471]: from nltk.stem import RegexpStemmer
     ...: rs = RegexpStemmer('ing$|s$|ed$', min=4)

In [472]: print rs.stem('jumping'), rs.stem('jumps'), rs.stem('jumped')
jump jump jump

In [473]: print rs.stem('lying')
ly
```

```
In [474]: print rs.stem('strange')
strange
```

你可以看出上面的词干提取结果与之前的词干提取结果之间的差异，上面的词干提取结果完全取决于我们自定义的提取规则（该规则基于正则表达式）。以下代码段展示了如何使用 SnowballStemmer 来对其他语言的单词进行词干提取（你可以在 http:// snowballstem. org 上找到有关 Snowball Project 的更多详细信息）：

```
# Snowball Stemmer
In [486]: from nltk.stem import SnowballStemmer
     ...: ss = SnowballStemmer("german")
In [487]: print 'Supported Languages:', SnowballStemmer.languages
Supported Languages: (u'danish', u'dutch', u'english', u'finnish',
u'french', u'german', u'hungarian', u'italian', u'norwegian', u'porter',
u'portuguese', u'romanian', u'russian', u'spanish', u'swedish')

# stemming on German words
# autobahnen -> cars
# autobahn -> car
In [488]: ss.stem('autobahnen')
Out[488]: u'autobahn'

# springen -> jumping
# spring -> jump
In [489]: ss.stem('springen')
Out[489]: u'spring'
```

波特词干提取器是目前最常用的词干提取器，但是在实际执行词干提取时，你还是应该根据具体问题来选择词干提取器，并经过反复试验以验证提取器效果。如果需要的话，你也可以使用自定义的规则来构建自己的提取器。

3.2.9 词形还原

词形还原（lemmatization）的过程与词干提取非常相似，去除词缀以获得单词的基本形式。但在这种情况下，这种基本形式称为根词（root word），而不是词干。它们的不同之处在于，词干不一定是标准的、正确的单词。也就是说，它可能不存在于词典中。根词也称为词元（lemma），始终存在于词典中。

词形还原的过程比词干提取慢很多，因为它涉及一个附加步骤，当且仅当该词元存在于词典中时，才通过去除词缀形成根形式或词元。nltk 包有一个强大的词形还原模块，它使用 WordNet、单词的句法和语义（如词性和语境）来获得根词或词元。还记得第 1 章的词性吗？它主要包含三个实体——名词、动词和形容词——最常见于自然语言。

以下代码段显示了如何对每类词语执行词形还原：

```
In [514]: from nltk.stem import WordNetLemmatizer
     ...:
     ...: wnl = WordNetLemmatizer()

# lemmatize nouns
In [515]: print wnl.lemmatize('cars', 'n')
     ...: print wnl.lemmatize('men', 'n')
car
men
# lemmatize verbs
In [516]: print wnl.lemmatize('running', 'v')
```

```
    ...: print wnl.lemmatize('ate', 'v')
run
eat

# lemmatize adjectives
In [517]: print wnl.lemmatize('saddest', 'a')
    ...: print wnl.lemmatize('fancier', 'a')
sad
fancy
```

上述代码段展示了每个单词是如何使用词形还原变回其基本格式的。词形还原有助于我们进行词语标准化。上述代码利用了 `WordNetLemmatizer` 类，它使用 `WordNetCorpusReader` 类的 `morphy()`函数。该函数使用单词及其词性，通过比对 WordNet 语料库，并采用递归技术删除词缀直到在词汇网络中找到匹配项，最终获得输入词的基本形式或词元。如果没有找到匹配项，则将返回输入词（输入词不做任何变化）。

在这里，词性非常重要，因为如果词性是错误的，那么词形还原就会失效，如下面的代码段所示：

```
# ineffective lemmatization
In [518]: print wnl.lemmatize('ate', 'n')
    ...: print wnl.lemmatize('fancier', 'v')
ate
fancier
```

至此，我们就结束了处理和规范化文本技术的讨论。到目前为止，你已经学到了很多关于如何处理、规范化和标准化文本的内容。在下一节中，我们将介绍如何分析和理解文本数据的句法属性和结构。

3.3 理解文本句法和结构

第 1 章详细介绍了语言的句法和结构。如果你已经忘记了，请翻到 1.3 节，并快速浏览以了解分析和理解文本数据句法和结构的各种方法。在本节中，我们将会介绍和实现一些用于理解文本句法和结构的概念和技术。这些算法在 NLP 中非常有用，它们通常在文本处理和标准化之后执行。本节，我们主要关注以下技术：

- 词性（POS）标签。
- 浅层分析。
- 基于依存关系的解析。
- 基于成分结构的解析。

本书针对的读者是文本分析实践人员，可以执行并注重在实际问题中使用技术和算法的最佳解决方案。所以，下面将介绍利用现有库（如 `nltk` 和 `spacy`）来实现和执行一些技术的最佳方法。此外，由于许多读者可能对技术的内部构件感兴趣，并且可能会尝试自己实现部分技术，我们也会介绍如何做到这一点。请记住，我们主要关注的是以实际的例子来研究实现概念的方法，而不是重写方法。在进一步介绍之前，我们将看看所需库的必要依存关系和安装细节，因为这可并不简单。

3.3.1 安装必要的依赖项

在本章中，我们将会使用如下库和依赖项：

- nltk 库，最好是 3.1 或 3.2.1 版。
- spacy 库。
- pattern 库。
- 斯坦福分析器（Stanford parser）。
- Graphviz 及必要库。

我们在第 1 章中介绍了安装 nltk 的相关内容。你可以直接通过终端机或命令提示符安装它，只要输入 pip install nltk，就可以自动下载并安装它。请注意，最好不要安装 3.2.0 版本的库，因为该版本中的部分函数存在问题，比如 pos_tag()函数。

完成 nltk 下载和安装后，请记得下载我们在第 1 章中讨论过的语料库。关于下载和安装 nltk 的更多信息，请参见 www.nltk.org/install.html 和 www.nltk.org/data.html，其上有详细的安装说明。你也可以启动 Python 解释器并执行以下代码段完成相同任务：

```
import nltk
# download all dependencies and corpora
nltk.download('all', halt_on_error=False)
# OR use a GUI based downloader and select dependencies
nltk.download()
```

想要安装 pattern 库，请输入命令 pip install pattern，下载并安装库及其必要依赖项。关于 pattern 库的更多信息请参见网址 www.clips.ua.ac.be/pages/ pattern-en。对于 spacy 库，你需要先安装软件包，然后单独安装其依赖项（也称为语言模型）。请在终端输入 pip install spacy 来安装 spacy 库。安装完成后，请使用命令 python-m spacy.en.download 从终端下载英文语言模型（大约 500 MB），存储于 spacy 包的根目录下。更多详细信息，请参见 https://spacy.io/docs/#getting-started，其上包含 spacy 库的使用说明。接下来，我们将使用 spacy 库进行标记，并说明基于依赖的解析。

斯坦福分析器是由斯坦福大学开发的基于 Java 的语言分析器，它能够帮助我们解析句子以了解其底层结构。我们将使用斯坦福分析器和 nltk 来执行基于依存关系的解析以及基于成分结构的解析。nltk 提供了一个出色的封装，它可以利用 Python 本身的分析器，因而无需在 Java 中编程。你可以参考网址 https://github.com/nltk/nltk/wiki/Installing-Third Party-Software，其上介绍了如何下载和安装斯坦福分析器并将其与 nltk 集成。就个人而言，我在安装过程中遇到过几次问题（特别是在 Windows 系统中），所以在这里我会提供安装斯坦福分析器和必要依赖项的最普遍的方法。

首先，请确保你已下载并安装了 Java Development Kit（不仅仅是 JRE，即 Java Runtime Environment），可从官方网站 www.oracle.com/technetwork/java/javase/downloads/index.html?ssSourceSiteId=otnjp 下载。Java SE 8u101/8u102 是本书编写时的最新版本——我本人就使用过 8u102。安装后，请确保通过将其添加到 Path 系统环境变量中来设置 Java 的“Path”。你还可以创建一个 JAVA_HOME 环境变量指向属于 JDK 的 java.exe 文件。根据我的经验，这两种常用的方法在运行 Python 代码时效果都不好，最好是直接使用 Python os 库来设置环境变量，当我们进一步深入介绍实施细节时我会展示如何进行环境变量设置。安装完 Java 后，

请从 http://nlp.stanford.edu/software/stanford-parser-full-2015-04-20.zip 下载官方的斯坦福分析器。你可以通过访问 http://nlp.stanford.edu/software/lex-parser.shtml#Download 来查看“发布历史”信息，并尝试更新的版本。下载完成后，请将解压缩文件添加到文件系统中的一个已知位置。完成以上步骤后，你就可以使用 nltk 的分析器了，这就是我们接下来要来探索学习的。

Graphviz 并不是一个必需库，我们仅仅使用它来查看斯坦福分析器生成的依存关系分析树。你可以从其官方网站 www.graphviz.org/download/下载并安装 Graphviz 库。然后，安装 pygraphviz，可以根据你的系统架构和 Python 版本从 www.lfd.uci.edu/~gohlke/pythonlibs/#pygraphviz 网站上的下载 wheel 文件。接下来，使用命令 pip install pygraphviz-1.3.1-cp27-none-win_amd64.whl 安装它（适用于 64 位系统 Python 2.7.x 环境）。安装完成后，pygraphviz 就应该可以正常工作了。可能有些读者在安装过程中会遇到其他问题，这种情况下，使用的如下代码依次完成 pydot-ng 和 graphviz 的安装可能会有所帮助：

```
pip install pydot-ng
pip install graphviz
```

至此，我们就安装完了必要的依赖项，可以开始执行和查看示例了。但是，实际上我们没有完全准备好。在研究代码和示例之前，我们还需要了解机器学习的一些基本概念。

3.3.2 机器学习重要概念

我们将在下一节中使用语料库，并利用一些预先构建好的标签器来训练我们自己的标签器。为了更好地理解实现过程，你必须知道如下与数据分析和机器学习相关的重要概念。

- 数据准备：通常包含特征提取以及训练前的数据预处理。
- 特征提取：从原始数据中提取出有用特征以训练机器学习模型的过程。
- 特征：数据的各种有用属性（以个人数据为例，可以是年龄、体重等）。
- 训练数据：用于训练模型的一组数据。
- 测试/校验数据：一组数据，经过预先训练的模型使用该组数据进行测试和评估，以评估模型优劣。
- 模型：使用数据/特征组合构建，一个机器学习算法可以是有监督的，也可以是无监督的。
- 准确率：模型预测的准确程度（还有其他详细的评估指标，如精确率、召回率和 F1-score）。

知道了这些术语应该足以让你开始接下来的学习了。更详细的知识介绍超出了本书的范围，但是如果你对于探索机器学习领域感兴趣，你可以在网络上找到大量关于机器学习的资料。后面的章节将会介绍与文本数据有关的有监督机器学习和无监督机器学习。

3.3.3 词性标注

词性（POS）是基于语法语境和词语作用的具体词汇分类。第 1 章介绍了 POS 的一些基础，并提到了主要的 POS，包括名词、动词、形容词和副词。对单词进行分类并标记 POS 标签称为词性标注或 POS 标注。POS 标签用于标注单词并描述其词性，当我们需要在基于 NPL 的程序中使用注释文本时，这是非常有用的，因为我们可以通过特定的词性过滤数据并利用

该信息来执行具体的分析，例如将词汇范围缩小至名词，分析哪些是最突出的词语，消除歧义并进行语法分析。

我们将使用 Penn Treebank 进行 POS 标注。你可以在 www. cis. uni-muenchen. de/ ~ schmid/tools/TreeTagger/data/Penn-Treebank-Tagset. pdf 中找到有关各种 POS 标签及其标注的更多信息，其上包含了详细的说明文档，举例说明了每一项标签。Penn Treebank 项目是宾夕法尼亚大学的一个项目，该项目网站 www. cis. upenn. edu/提供了更多的相关信息。目前，有各种各样的标签以满足不同的应用场景，例如 POS 标签是分配给单词标记词性的标签，语块标签通常是分配给短语的标签，还一些标签是用于描述关系的次级标签。表 3-1 给出了词性标签的详细描述及其示例，如果你并不想费力气去查看 Penn Treebank 标签的详细文档，你可以随时用它作为参考，以便更好地了解 POS 标签和分析树。

表 3-1 词性标签

序 号	TAG	描 述	示 例
1	CC	条件连接词	and, or
2	CD	基本数量词	five, one, 2
3	DT	限定词	a, the
4	EX	存在量词	there were two cars
5	FW	外来词	d'hoevre, mais
6	IN	介词/从句连接词	of, in, on, that
7	JJ	形容词	quick, lazy
8	JJR	比较级形容词	quicker, lazier
9	JJS	最高级形容词	quickest, laziest
10	LS	列表项标记符	2)
11	MD	情态动词	could, should
12	NN	单数或不可数名词	fox, dog
13	NNS	复数名词	foxes, dogs
14	NNP	专有名词单数	John, Alice
15	NNPS	专有名词复数	Vikings, Indians, Germans
16	PDT	前置限定词	both the cats
17	POS	所有格	boss's
18	PRP	人称代词	me, you
19	PRP $	所有格代词	our, my, your
20	RB	副词	naturally, extremely, hardly
21	RBR	比较级副词	better
22	RBS	最高级级副词	best
23	RP	副词小品词	about, up
24	SYM	符号	%, $
25	TO	不定词	how to, what to do
26	UH	感叹词	oh, gosh, wow
27	VB	动词原型	run, give
28	VBD	动词过去式	ran, gave
29	VBG	动名词	running, giving
30	VBN	动词过去分词	given

（续）

序 号	TAG	描 述	示 例
31	VBP	动词非第三人称一般现在时	I think, I take
32	VBZ	动词第三人称一般现在时	he thinks, he takes
33	WDT	WH 限定词	which, whatever
34	WP	WH 人称代词	who, what
35	WP $	WH 物主代词	whose
36	WRB	WH 副词	where, when
37	NP	名词短语	the brown fox
38	PP	介词短语	in between, over the dog
39	VP	动词短语	was jumping
40	ADJP	形容词短语	warm and snug
41	ADVP	副词短语	also
42	SBAR	主从句连接词	whether or not
43	PRT	小品词	up
44	INTJ	语气词	hello
45	PNP	介词名词短语	over the dog, as of today
46	-SBJ	主句	the fox jumped over the dog
47	-OBJ	从句	the fox jumped over the dog

该表显示了 Penn Treebank 中主要的 POS 标签集，也是各类文本分析和自然语言处理程序中使用最广泛的 POS 标签集合。在下一节中，我们将推荐一些 POS 标签器，并且介绍如何构建自己的标签。

1. POS 标签器推荐

我们将在这里讨论一些标记句子的推荐方法。第一种方法是使用 `nltk` 推荐的`pos_tag()`函数，它基于 Penn Treebank。我们将在这里再次使用第 1 章中用过的有趣的例句。以下代码段展示了使用 `nltk` 获取句子 POS 标签的方法：

```
sentence = 'The brown fox is quick and he is jumping over the lazy dog'

import nltk
tokens = nltk.word_tokenize(sentence)
tagged_sent = nltk.pos_tag(tokens, tagset='universal')
In [13]: print tagged_sent
[('The', u'DET'), ('brown', u'ADJ'), ('fox', u'NOUN'), ('is', u'VERB'),
('quick', u'ADJ'), ('and', u'CONJ'), ('he', u'PRON'), ('is', u'VERB'),
('jumping', u'VERB'), ('over', u'ADP'), ('the', u'DET'), ('lazy', u'ADJ'),
('dog', u'NOUN')]
```

上面的输出显示了句子中每个单词的 POS 标签。你可以发现其与表 3-1 所示的标签非常相似。其中一些作为通用/普遍标签在第 1 章中也提到过。你还可以使用 `pattern` 模块通过以下代码段获取句子的 POS 标签：

```
from pattern.en import tag
tagged_sent = tag(sentence)
In [15]: print tagged_sent
 [(u'The', u'DT'), (u'brown', u'JJ'), (u'fox', u'NN'), (u'is', u'VBZ'),
```

```
(u'quick', u'JJ'), (u'and', u'CC'), (u'he', u'PRP'), (u'is', u'VBZ'),
(u'jumping', u'VBG'), (u'over', u'IN'), (u'the', u'DT'), (u'lazy', u'JJ'),
(u'dog', u'NN')]
```

该输出为我们提供了严格遵循 Penn Treebank 格式的标签，指出了形容词、名词或动词并给出了详细信息。

2. 建立自己的 POS 标签器

在本节中，我们将探讨一些构建自己的 POS 标签器的技术，并利用 nltk 提供的一些类来实现它们。为了评估我们的标签器性能，我们会使用 nltk 中 treebank 语料库的一些测试数据。我们还将使用一些训练数据来训练标签器。首先，通过读取已标记的 treebank 语料库，我们可以获得训练和评估标签器的必要数据：

```
from nltk.corpus import treebank
data = treebank.tagged_sents()
train_data = data[:3500]
test_data = data[3500:]
# get a look at what each data point looks like
In [17]: print train_data[0]
 [(u'Pierre', u'NNP'), (u'Vinken', u'NNP'), (u',', u','), (u'61', u'CD'),
(u'years', u'NNS'), (u'old', u'JJ'), (u',', u','), (u'will', u'MD'),
(u'join', u'VB'), (u'the', u'DT'), (u'board', u'NN'), (u'as', u'IN'), (u'a',
u'DT'), (u'nonexecutive', u'JJ'), (u'director', u'NN'), (u'Nov.', u'NNP'),
(u'29', u'CD'), (u'.', u'.')]

# remember tokens is obtained after tokenizing our sentence
tokens = nltk.word_tokenize(sentence)
In [18]: print tokens
['The', 'brown', 'fox', 'is', 'quick', 'and', 'he', 'is', 'jumping', 'over',
'the', 'lazy', 'dog']
```

我们将使用测试数据来评估标签器，并使用例句的标识作为输入来验证标签器的工作效果。我们在 nltk 中使用的所有标签器均来自 nltk.tag 包。每个标签器都是基类 TaggerI 类的子类，并且每个标签器都执行一个 tag()函数，它将一个句子的标识列表作为输入，返回带有 POS 标签的相同单词列表作为输出。除了标记外，还有一个 evaluate()函数用于评估标签器的性能。它通过标记每个输入测试语句，然后将输出结果与句子的实际标签进行对比来完成评估。我们将使用该函数来测试我们的标签器在 test_data 上的性能。

首先，我们看看从 SequentialBackoffTagger 基类继承的 DefaultTagger，并为每个单词分配相同的用户输入 POS 标签。这可能看起来很简单，但它是构建 POS 标签器基准的好方法：

```
from nltk.tag import DefaultTagger
dt = DefaultTagger('NN')

# accuracy on test data
In [24]: print dt.evaluate(test_data)
0.145415819537
# tagging our sample sentence
In [25]: print dt.tag(tokens)
 [('The', 'NN'), ('brown', 'NN'), ('fox', 'NN'), ('is', 'NN'), ('quick',
'NN'), ('and', 'NN'), ('he', 'NN'), ('is', 'NN'), ('jumping', 'NN'),
('over', 'NN'), ('the', 'NN'), ('lazy', 'NN'), ('dog', 'NN')]
```

从上面的输出可以看出，在树库（treebank）测试数据集中，我们已经获得了 14% 的

单词正确标记率——这个结果并不是很理想，并且正如预期的那样，例句中的输出标签都为名词，因为我们给标签器输入的都是相同的标签。

现在，我们将使用正则表达式和 RegexpTagger 来尝试构建一个性能更好的标签器：

```
from nltk.tag import RegexpTagger
# define regex tag patterns
patterns = [
        (r'.*ing$', 'VBG'),               # gerunds
        (r'.*ed$', 'VBD'),                # simple past
        (r'.*es$', 'VBZ'),                # 3rd singular present
        (r'.*ould$', 'MD'),               # modals
        (r'.*\'s$', 'NN$'),               # possessive nouns
        (r'.*s$', 'NNS'),                 # plural nouns
        (r'^-?[0-9]+(.[0-9]+)?$', 'CD'),  # cardinal numbers
        (r'.*', 'NN')                     # nouns (default) ... ]
rt = RegexpTagger(patterns)

# accuracy on test data
In [27]: print rt.evaluate(test_data)
0.240391131765

# tagging our sample sentence

In [28]: print rt.tag(tokens)
 [('The', 'NN'), ('brown', 'NN'), ('fox', 'NN'), ('is', 'NNS'), ('quick',
'NN'), ('and', 'NN'), ('he', 'NN'), ('is', 'NNS'), ('jumping', 'VBG'),
('over', 'NN'), ('the', 'NN'), ('lazy', 'NN'), ('dog', 'NN')]
```

该输出显示现在的准确性已经提高到了 24%。我们可以做得更好吗？我们现在将训练一些 n 元分词标签器。n 元分词是来自文本序列或语音序列的 n 个连续项。这些项可以由单词、音素、字母、字符或音节组成。Shingles 是只包含单词的 n 元分词。我们将使用大小为 1、2 和 3 的 n 元分词，它们分别也称为一元分词（unigram）、二元分词（bigram）和三元分词（trigram）。UnigramTagger、BigramTagger 和 TrigramTagger 继承自基类 NGramTagger，NGramTagger 类则继承自 ContextTagger 类，该类又继承自 SequentialBackoffTagger 类。我们将使用 train_data 作为训练数据，根据语句标识及其 POS 标签来训练 n 元分词标签器。然后我们将在 test_data 上评估训练后的标签器，并查看例句的标记结果：

```
from nltk.tag import UnigramTagger
from nltk.tag import BigramTagger
from nltk.tag import TrigramTagger

ut = UnigramTagger(train_data)
bt = BigramTagger(train_data)
tt = TrigramTagger(train_data)

# testing performance of unigram tagger
In [31]: print ut.evaluate(test_data)
0.861361215994
In [32]: print ut.tag(tokens)
[('The', u'DT'), ('brown', None), ('fox', None), ('is', u'VBZ'), ('quick',
u'JJ'), ('and', u'CC'), ('he', u'PRP'), ('is', u'VBZ'), ('jumping', u'VBG'),
('over', u'IN'), ('the', u'DT'), ('lazy', None), ('dog', None)]

# testing performance of bigram tagger
In [33]: print bt.evaluate(test_data)
0.134669377481
In [34]: print bt.tag(tokens)
```

```
 [('The', u'DT'), ('brown', None), ('fox', None), ('is', None), ('quick',
None), ('and', None), ('he', None), ('is', None), ('jumping', None),
('over', None), ('the', None), ('lazy', None), ('dog', None)]

# testing performance of trigram tagger
In [35]: print tt.evaluate(test_data)
0.0806467228192
In [36]: print tt.tag(tokens)
 [('The', u'DT'), ('brown', None), ('fox', None), ('is', None), ('quick',
None), ('and', None), ('he', None), ('is', None), ('jumping', None),
('over', None), ('the', None), ('lazy', None), ('dog', None)]
```

上面的输出清楚地表明，我们仅使用 UnigramTagger 标签器就可以在测试集上获得86%的准确率，这个结果与我们前一个标签器相比要好很多。标签 None 表示标签器无法标记该单词，因为它在训练数据中未能获取类似的标识。二元分词和三元分词模型的准确性远远不及一元分词模型，因为在训练数据中观察到的二元词组和三元词组不一定会在测试数据中以相同的方式出现。

现在，通过创建一个包含标签列表的组合标签器以及使用 backoff 标签器，我们将尝试组合运用所有的标签器。本质上，我们将创建一个标签器链，对于每一个标签器，如果它不能标记输入的标识，则标签器的下一步将会回退到 backoff 标签器：

```
def combined_tagger(train_data, taggers, backoff=None):
    for tagger in taggers:
        backoff = tagger(train_data, backoff=backoff)
    return backoff

ct = combined_tagger(train_data=train_data,
                     taggers=[UnigramTagger, BigramTagger, TrigramTagger],
                     backoff=rt)

# evaluating the new combined tagger with backoff taggers
In [38]: print ct.evaluate(test_data)
0.910155871817
In [39]: print ct.tag(tokens)
 [('The', u'DT'), ('brown', 'NN'), ('fox', 'NN'), ('is', u'VBZ'), ('quick',
u'JJ'), ('and', u'CC'), ('he', u'PRP'), ('is', u'VBZ'), ('jumping', 'VBG'),
('over', u'IN'), ('the', u'DT'), ('lazy', 'NN'), ('dog', 'NN')]
```

我们现在在测试数据上获得了91%的准确率，效果非常好。另外我们也看到，这个新标签器能够成功地标记例句中的所有标识（即使它们中的一些不正确，比如 brown 应该是一个形容词）。

对于最终的标签器，我们将使用有监督的分类算法来训练它。ClassifierBasedPOSTagger 类使我们能够使用 classifier_builder 参数中的有监督机器学习算法来训练标签器。该类继承自 ClassifierBasedTagger，并拥有构成训练过程核心部分的feature_detector()函数。该函数用于从训练数据（如单词、前一个单词、标签、前一个标签，大小写等）中生成各种特征。实际上，在实例化 ClassifierBasedPOSTagger 类对象时，你也可以构建自己的特征检测器函数，将其传递给 feature_detector 参数。在这里，我们使用的分类器是 NaiveBayesClassifier，它使用贝叶斯定理构建概率分类器，假设特征之间是独立的。相关算法细节超出了本书的讨论范围，如果你想了解更多，请访问 https://en. wikipedia. org/wiki/Naive_Bayes_classifier。

以下代码段展示了如何基于分类方法构建 POS 标签器并对其进行评估：

```
from nltk.classify import NaiveBayesClassifier
from nltk.tag.sequential import ClassifierBasedPOSTagger

nbt = ClassifierBasedPOSTagger(train=train_data,
                               classifier_builder=NaiveBayesClassifier.
                               train)

# evaluate tagger on test data and sample sentence
In [41]: print nbt.evaluate(test_data)
0.930680607997
In [42]: print nbt.tag(tokens)
 [('The', u'DT'), ('brown', u'JJ'), ('fox', u'NN'), ('is', u'VBZ'),
('quick', u'JJ'), ('and', u'CC'), ('he', u'PRP'), ('is', u'VBZ'),
('jumping', u'VBG'), ('over', u'IN'), ('the', u'DT'), ('lazy', u'JJ'),
('dog', u'VBG')]
```

使用上面的标签器，我们在测试数据上的准确率达到了93%——这在我们所有的标签器中是最高的。此外，如果仔细观察例句的输出标签，你会发现它们不仅是正确的，并且是完全合理的。这让我们了解到基于分类器的POS标签器是多么强大和有效。你也可以尝试使用其他的分类器，如`MaxentClassifier`，并将其性能与此标签器性能进行比较。此外，还有几种使用`nltk`和其他程序包构建或使用POS标签器的方法。以上内容应该可以满足你对于POS标签器的需求，在这里，我们没有必要再去了解其他的算法，但是如果你很感兴趣的话，也可以继续探索其他算法并与本章介绍的方法进行对此，以满足你的好奇心。

3.3.4 浅层分析

浅层分析（shallow parsing）也称为浅分析（light parsing）或组块分析（chunking），是分析句子结构的一种技术，它将句子分解成最小的组成部分（它们是标识，如单词），然后将它们组合成更高级的短语。在浅层分析中，主要的关注焦点是识别这些短语或语块，而不是挖掘每个块内句法和语句关系的深层细节，正如我们在基于深度分析获得的分析树中看到的。浅层分析的主要目的是获得语义上有意义的短语，并观察它们之间的关系。

请参阅1.3节，来帮助你回忆由一连串单词所组成的句子是如何被词语或短语赋予结构的。在该节中，还展示了例句的浅层分析树。

接下来，我们将从一些值得推荐的、简单易用的浅层分析器开始，研究进行浅层分析的各种方法。我们还将使用例如正则表达式、分块、加缝隙和基于标签的训练等技术，来实现我们自己的浅层分析器。

1. 浅层分析器推荐

在这里，我们将使用`pattern`包创建一个浅层分析器，用以从句子中提取有意义的语块。以下代码段展示了如何在我们的例句上执行浅层分析：

```
sentence = 'The brown fox is quick and he is jumping over the lazy dog'

from pattern.en import parsetree
tree = parsetree(sentence)

# print the shallow parsed sentence tree
In [5]: print tree
   ...:
[Sentence('The/DT/B-NP/O brown/JJ/I-NP/O fox/NN/I-NP/O is/VBZ/B-VP/O quick/
JJ/B-ADJP/O and/CC/O/O he/PRP/B-NP/O is/VBZ/B-VP/O jumping/VBG/I-VP/O over/
IN/B-PP/B-PNP the/DT/B-NP/I-PNP lazy/JJ/I-NP/I-PNP dog/NN/I-NP/I-PNP')]
```

上面的输出就是例句的原始浅层分析语句树。如果将它们与之前的 POS 标签表进行对比，你会发现许多标签是非常相似的。你还会注意到上面的输出中有一些新的符号——前缀 I、O 和 B，即分块技术领域里十分流行的 IOB 标注，I、O 和 B 分别表示内部、外部和开头。标签前面的 B - 前缀表示它是块的开始，而 I - 前缀则表示它在块内。O 标签表示标识不属于任何块。当后续标签跟当前语块的标签类型相同，并且它们之间不存在 O 标签时，则对当前块使用 B - 标签。

以下代码段显示了如何简单易懂地获得语块：

```
# print all chunks
In [6]: for sentence_tree in tree:
   ...:     print sentence_tree.chunks
[Chunk('The brown fox/NP'), Chunk('is/VP'), Chunk('quick/ADJP'), Chunk('he/
NP'), Chunk('is jumping/VP'), Chunk('over/PP'), Chunk('the lazy dog/NP')]

# Depict each phrase and its internal constituents
In [9]: for sentence_tree in tree:
   ...:     for chunk in sentence_tree.chunks:
   ...:         print chunk.type, '->', [(word.string, word.type)
   ...:                                  for word in chunk.words]
NP -> [(u'The', u'DT'), (u'brown', u'JJ'), (u'fox', u'NN')]
VP -> [(u'is', u'VBZ')]
ADJP -> [(u'quick', u'JJ')]
NP -> [(u'he', u'PRP')]
VP -> [(u'is', u'VBZ'), (u'jumping', u'VBG')]
PP -> [(u'over', u'IN')]
NP -> [(u'the', u'DT'), (u'lazy', u'JJ'), (u'dog', u'NN')]
```

上面的输出是例句的浅层分析结果，该结果十分简单明了，其中的每个短语及其组成部分都被清楚地显示出来。

我们可以构建一些通用函数，更好地解析和可视化浅层分析的语句树，还可以在分析常见句子时重复使用它们，如下列代码所示：

```
from pattern.en import parsetree, Chunk
from nltk.tree import Tree

# create a shallow parsed sentence tree
def create_sentence_tree(sentence, lemmatize=False):
    sentence_tree = parsetree(sentence,
                              relations=True,
                              lemmata=lemmatize) # if you want to lemmatize
                              the tokens
    return sentence_tree[0]

# get various constituents of the parse tree
def get_sentence_tree_constituents(sentence_tree):
    return sentence_tree.constituents()

# process the shallow parsed tree into an easy to understand format
def process_sentence_tree(sentence_tree):

    tree_constituents = get_sentence_tree_constituents(sentence_tree)
    processed_tree = [
                      (item.type,
                       [
                           (w.string, w.type)
                           for w in item.words
                       ]
```

```
                    )
                    if type(item) == Chunk
                    else ('-',
                          [
                              (item.string, item.type)
                          ]
                         )
                          for item in tree_constituents
                ]

    return processed_tree

# print the sentence tree using nltk's Tree syntax
def print_sentence_tree(sentence_tree):

    processed_tree = process_sentence_tree(sentence_tree)
    processed_tree = [
                    Tree( item[0],
                         [
                             Tree(x[1], [x[0]])
                             for x in item[1]
                          ]
                        )
                        for item in processed_tree
                ]

    tree = Tree('S', processed_tree )
    print tree
# visualize the sentence tree using nltk's Tree syntax
def visualize_sentence_tree(sentence_tree):

    processed_tree = process_sentence_tree(sentence_tree)
    processed_tree = [
                    Tree( item[0],
                         [
                             Tree(x[1], [x[0]])
                             for x in item[1]
                          ]
                        )
                        for item in processed_tree
                ]
    tree = Tree('S', processed_tree )
    tree.draw()
```

执行以下代码段，我们可以看出上述函数是如何在例句中发挥作用的：

```
# raw shallow parsed tree
In [11]: t = create_sentence_tree(sentence)
    ...: print t
Sentence('The/DT/B-NP/O/NP-SBJ-1 brown/JJ/I-NP/O/NP-SBJ-1 fox/NN/I-NP/O/NP-
SBJ-1 is/VBZ/B-VP/O/VP-1 quick/JJ/B-ADJP/O/O and/CC/O/O/O he/PRP/B-NP/O/NP-
SBJ-2 is/VBZ/B-VP/O/VP-2 jumping/VBG/I-VP/O/VP-2 over/IN/B-PP/B-PNP/O the/
DT/B-NP/I-PNP/O lazy/JJ/I-NP/I-PNP/O dog/NN/I-NP/I-PNP/O')

# processed shallow parsed tree
In [16]: pt = process_sentence_tree(t)
    ...: pt
Out[16]:
[(u'NP', [(u'The', u'DT'), (u'brown', u'JJ'), (u'fox', u'NN')]),
 (u'VP', [(u'is', u'VBZ')]),
 (u'ADJP', [(u'quick', u'JJ')]),
 ('-', [(u'and', u'CC')]),
```

```
 (u'NP', [(u'he', u'PRP')]),
 (u'VP', [(u'is', u'VBZ'), (u'jumping', u'VBG')]),
 (u'PP', [(u'over', u'IN')]),
 (u'NP', [(u'the', u'DT'), (u'lazy', u'JJ'), (u'dog', u'NN')])]

# print shallow parsed tree in an easy to understand format using nltk's
Tree syntax
In [17]: print_sentence_tree(t)
(S
  (NP (DT The) (JJ brown) (NN fox))
  (VP (VBZ is))
  (ADJP (JJ quick))
  (- (CC and))
  (NP (PRP he))
  (VP (VBZ is) (VBG jumping))
  (PP (IN over))
  (NP (DT the) (JJ lazy) (NN dog)))

# visualize the shallow parsed tree
In [18]: visualize_sentence_tree(t)
```

上面的输出显示了从例句中创建、表示和可视化浅层分析树的方法。对于同一个例句，图 3-2 所示的可视化结果与第 1 章的树形表示非常类似。最低一级表示实际的标识值；上一级别表示每个标识的 POS 标签；而再上一级表示语块短语的标签。你可以继续在其他的句子上尝试这些函数，并比较它们的结果。在下一节中，我们将尝试构建自己的浅层分析器。

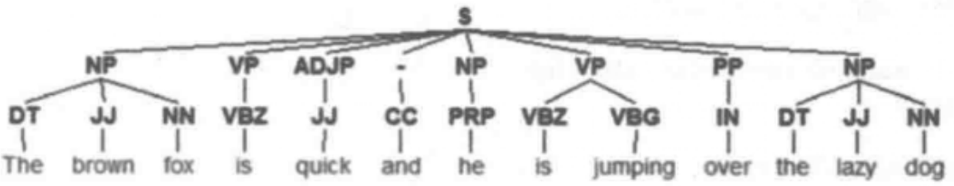


图 3-2 例句的浅层分析树的可视化表示

2. 构建自己的浅层分析器

我们将会使用正则表达式、基于标签的学习器等技术构建自己的浅层分析器。与之前的 POS 标签类似，如果需要的话，我们会使用一些训练数据来训练分析器，然后使用测试数据和例句对分析器进行评估。在 `nltk` 中，可以使用 `treebank` 语料库，它带有语块标注。首先，加载语料库，并使用以下代码段准备训练数据集和测试数据集：

```
from nltk.corpus import treebank_chunk
data = treebank_chunk.chunked_sents()
train_data = data[:4000]
test_data = data[4000:]

# view what a sample data point looks like
In [21]: print train_data[7]
 (S
  (NP A/DT Lorillard/NNP spokewoman/NN)
  said/VBD
  ,/,
  ``/``
  (NP This/DT)
  is/VBZ
  (NP an/DT old/JJ story/NN)
  ./.)
```

从上面的输出可以看出，我们的数据点是使用短语和 POS 标签完成标注的句子，这将有助于训练浅层分析器。我们将从使用正则表达式开始进行浅层分析，同时还会使用分块和加缝隙的概念。通过分块，我们可以使用并指定特定的模式来识别想要在句子中分块或分段的内容，例如一些基于特定元数据（如每个标识的 POS 标签）的短语。加缝隙过程则与分块过程相反，在该过程中，我们指定一些特定的标识使其不属于任何语块，然后形成除这些标识以外的必要语块。让我们一起来看一个简单的例句，通过使用 RegexpParser类，我们可以利用正则表达式创建浅层分析器，以说明名词短语的分块和加缝隙过程，如下所示：

```
simple_sentence = 'the quick fox jumped over the lazy dog'

from nltk.chunk import RegexpParser
from pattern.en import tag
# get POS tagged sentence
tagged_simple_sent = tag(simple_sentence)
In [83]: print tagged_simple_sent
[(u'the', u'DT'), (u'quick', u'JJ'), (u'fox', u'NN'), (u'jumped', u'VBD'),
(u'over', u'IN'), (u'the', u'DT'), (u'lazy', u'JJ'), (u'dog', u'NN')]

# illustrate NP chunking based on explicit chunk patterns
chunk_grammar = """
NP: {<DT>?<JJ>*<NN.*>}
"""
rc = RegexpParser(chunk_grammar)
c = rc.parse(tagged_simple_sent)

# view NP chunked sentence using chunking
In [86]: print c
(S
  (NP the/DT quick/JJ fox/NN)
  jumped/VBD
  over/IN
  (NP the/DT lazy/JJ dog/NN))

# illustrate NP chunking based on explicit chink patterns
chink_grammar = """
NP: {<.*>+} # chunk everything as NP
}<VBD|IN>+{
"""
rc = RegexpParser(chink_grammar)
c = rc.parse(tagged_simple_sent)

# view NP chunked sentence using chinking
In [89]: print c
(S
  (NP the/DT quick/JJ fox/NN)
  jumped/VBD
  over/IN
  (NP the/DT lazy/JJ dog/NN))
```

从上面的输出中可以看出，我们在试验性 NP（名词短语）浅层分析器上使用分块和加缝隙方法得到了相同的结果。请记住，语块是包含在组块（语块）集合中的标识序列，缝隙则是被排除在语块之外的标识或标识序列。

现在，我们要训练一个更为通用的基于正则表达式的浅层分析器，并在我们的测试 treebank 数据上检测其性能。在程序内部，需要执行几个步骤来完成此分析器。首选，需

要将 nltk 中用于表示被解析语句的 Tree 结构转换为 ChunkString 对象。然后，使用定义好的分块和加缝隙规则创建一个 RegexpParser 对象。最后，使用 ChunkRule 和 ChinkRule 类及其对象创建完整的、带有必要语块的浅层分析树。以下代码段展示了基于正则表达式的浅层分析器：

```
# create POS tagged tokens for sample sentence
tagged_sentence = tag(sentence)
In [90]: print tagged_sentence
[(u'The', u'DT'), (u'brown', u'JJ'), (u'fox', u'NN'), (u'is', u'VBZ'),
(u'quick', u'JJ'), (u'and', u'CC'), (u'he', u'PRP'), (u'is', u'VBZ'),
(u'jumping', u'VBG'), (u'over', u'IN'), (u'the', u'DT'), (u'lazy', u'JJ'),
(u'dog', u'NN')]

# create the shallow parser
grammar = """
NP: {<DT>?<JJ>?<NN.*>}
ADJP: {<JJ>}
ADVP: {<RB.*>}
PP: {<IN>}
VP: {<MD>?<VB.*>+}

"""
rc = RegexpParser(grammar)
c = rc.parse(tagged_sentence)

# view shallow parsed sample sentence
In [99]: print c
(S
  (NP The/DT brown/JJ fox/NN)
  (VP is/VBZ)
  quick/JJ
  and/CC
  he/PRP
  (VP is/VBZ jumping/VBG)
  (PP over/IN)
  (NP the/DT lazy/JJ dog/NN))

# evaluate parser performance on test data
In [100]: print rc.evaluate(test_data)
ChunkParse score:
    IOB Accuracy:  54.5%
    Precision:     25.0%
    Recall:        52.5%
    F-Measure:     33.9%
```

我们可以看出，上面输出的例句分析树非常类似于上一节内容中分析器给出的分析树。此外，测试数据的整体准确率达到了 54.5%，这是一个很不错的开头。更多关于性能指标的详细信息，请参阅 4.7 节。

还记得我之前提到的带注释的文本标记元数据在许多方面都是很有用的吗？接下来，我们将使用分好块并标记好的 treebank 训练数据，构建一个浅层分析器。我们会用到两个分块函数：一个是 tree2conlltags 函数，它可以为每个词元获取三组数据——单词、标签和块标签；另一个是 conlltags2tree 函数，它可以从上述三元组数据中生成分析树。稍后，我们将使用这些函数来训练分析器。首先，一起来看看这两个函数是如何工作的。请记住，块标签使用前面提到的 IOB 格式：

```
from nltk.chunk.util import tree2conlltags, conlltags2tree

# look at a sample training tagged sentence
In [104]: train_sent = train_data[7]
     ...: print train_sent
(S
  (NP A/DT Lorillard/NNP spokewoman/NN)
  said/VBD
  ,/,
  ``/``
  (NP This/DT)
  is/VBZ
  (NP an/DT old/JJ story/NN)
  ./.)

# get the (word, POS tag, Chunk tag) triples for each token
In [106]: wtc = tree2conlltags(train_sent)
     ...: wtc
Out[106]:
[(u'A', u'DT', u'B-NP'),
 (u'Lorillard', u'NNP', u'I-NP'),
 (u'spokewoman', u'NN', u'I-NP'),
 (u'said', u'VBD', u'O'),
 (u',', u',', u'O'),
 (u'``', u'``', u'O'),
 (u'This', u'DT', u'B-NP'),
 (u'is', u'VBZ', u'O'),
 (u'an', u'DT', u'B-NP'),
 (u'old', u'JJ', u'I-NP'),
 (u'story', u'NN', u'I-NP'),
 (u'.', u'.', u'O')]

# get shallow parsed tree back from the WTC triples
In [107]: tree = conlltags2tree(wtc)
     ...: print tree
(S
  (NP A/DT Lorillard/NNP spokewoman/NN)
  said/VBD
  ,/,
  ``/``
  (NP This/DT)
  is/VBZ
  (NP an/DT old/JJ story/NN)
  ./.)
```

现在，我们已经知道了这些函数是如何工作的，接下来，我们定义一个函数 conll_tag_chunks()从分块标注好的句子中提取 POS 和块标签。从 POS 标注到使用组合标签器（包含 backoff 标签器）训练数据的过程中，我们还可以再次使用 combined_taggers()函数，如以下代码段所示：

```
def conll_tag_chunks(chunk_sents):
  tagged_sents = [tree2conlltags(tree) for tree in chunk_sents]
  return [[(t, c) for (w, t, c) in sent] for sent in tagged_sents]

def combined_tagger(train_data, taggers, backoff=None):
    for tagger in taggers:
        backoff = tagger(train_data, backoff=backoff)
    return backoff
```

现在，我们定义一个 NGramTagChunker 类，将标记好的句子作为训练输入，获取它们

的 WTC 三元组，即单词（word）、POS 标签（POS tag）和块标签（Chunk tag）三元组，并使用 UnigramTagger 作为 backoff 标签器训练一个 BigramTagger。我们还将定义一个 parse()函数来对新的句子执行浅层分析：

```
from nltk.tag import UnigramTagger, BigramTagger
from nltk.chunk import ChunkParserI

class NGramTagChunker(ChunkParserI):

  def __init__(self, train_sentences,
               tagger_classes=[UnigramTagger, BigramTagger]):
    train_sent_tags = conll_tag_chunks(train_sentences)
    self.chunk_tagger = combined_tagger(train_sent_tags, tagger_classes)

  def parse(self, tagged_sentence):
    if not tagged_sentence:
        return None
    pos_tags = [tag for word, tag in tagged_sentence]
    chunk_pos_tags = self.chunk_tagger.tag(pos_tags)
    chunk_tags = [chunk_tag for (pos_tag, chunk_tag) in chunk_pos_tags]
    wpc_tags = [(word, pos_tag, chunk_tag) for ((word, pos_tag), chunk_tag)
                    in zip(tagged_sentence, chunk_tags)]
    return conlltags2tree(wpc_tags)
```

在上述类中，构造函数__init__()使用基于语句 WTC 三元组的 n 元分词标签训练浅层分析器。在程序内部，它将一列训练语句作为输入，这些训练句使用分好块的分析树元数据作标注。该函数使用我们之前定义的 conll_tag_chunks()函数来获取所有分块分析树的 WTC 三元组数据列表。然后，该函数使用这些三元组数据来训练一个 Bigram 标签器，它使用 Unigram 标签器作为 backoff 标签器，并且将训练模型存储在 self. chunk_tagger 中。请记住，你可以在训练中使用 tagger_classes 参数来分析其他 n 元分词标签。完成训练后，可以使用 parse()函数来评估测试数据上的标签并对新的句子进行浅层分析。在程序内部，该函数使用经 POS 标注的句子作为输入，从句子中分离出 POS 标签，并使用我们训练完的 self. chunk_tagger 获取句子的 IOB 块标签。然后，将其与原始句子标识相结合，并使用 conlltags2tree()函数获取最终的浅层分析树。

以下代码段展示了我们的分析器：

```
# train the shallow parser
ntc = NGramTagChunker(train_data)

# test parser performance on test data
In [114]: print ntc.evaluate(test_data)
ChunkParse score:
    IOB Accuracy:  99.6%
    Precision:     98.4%
    Recall:       100.0%
    F-Measure:     99.2%

# parse our sample sentence
In [115]: tree = ntc.parse(tagged_sentence)
     ...: print tree
(S
  (NP The/DT brown/JJ fox/NN)
  is/VBZ
  (NP quick/JJ)
  and/CC
```

```
  (NP he/PRP)
  is/VBZ
  jumping/VBG
  over/IN
  (NP the/DT lazy/JJ dog/NN))
```

从以上输出可以看出，在 treebank 测试集数据上，我们的分析器总体准确率达到了 99.6%，性能非常出色！

现在，我们一起在 conll2000 语料库上对我们的分析器进行训练和评估。conll2000 语料库是一个更大的语料库，它包含了“华尔街日报”摘录。我们将在前 7 500 个句子上训练我们的分析器，并在其余 3 448 个句子上进行性能测试。如下面的代码段所示：

```
from nltk.corpus import conll2000
wsj_data = conll2000.chunked_sents()
train_wsj_data = wsj_data[:7500]
test_wsj_data = wsj_data[7500:]
# look at a sample sentence in the corpus
In [125]: print train_wsj_data[10]
(S
  (NP He/PRP)
  (VP reckons/VBZ)
  (NP the/DT current/JJ account/NN deficit/NN)
  (VP will/MD narrow/VB)
  (PP to/TO)
  (NP only/RB #/# 1.8/CD billion/CD)
  (PP in/IN)
  (NP September/NNP)
  ./.)

# train the shallow parser
tc = NGramTagChunker(train_wsj_data)

# test performance on the test data
In [126]: print tc.evaluate(test_wsj_data)
ChunkParse score:
    IOB Accuracy:  66.8%
    Precision:     77.7%
    Recall:        45.4%
    F-Measure:     57.3%
```

上面的程序输出显示，我们的分析器整体准确率大致是 67%，略低于在 treebank 测试数据上的结果，因为 conll2000 语料库要远远大于 treebank 语料库。你还可以尝试使用其他技术，比如 ClassifierBasedTagger 类的有监督分类器，来实现浅层分析器。

3.3.5 基于依存关系的分析

在基于依存关系的分析中，我们会使用依存语法来分析和推断语句中每个标识在结构上和语义上的关系（如果你需要唤醒之前的记忆，请参阅 1.3.4 节下的“依存语法”内容）。基于依存关系的语法可以帮助我们使用依存标签标注句子。依存标签是标识之间的一对一的映射，表示标识之间的依存关系。基于依存语法的分析树是一个有标签且有方向的树或图，可以更加精确地表示语句。分析树中的节点始终是词汇类型的标识，有标签的边表示起始点及其从属项（依赖起始点的标识）的依存关系。有向边上的标签表示依存关系中的语法角色。你应该还记得我们的例句“The brown fox is quick and he is jumping over the lazy dog”，我们在第 1 章中给出了该句的一种依存关系表示，如图 3-3 所示。

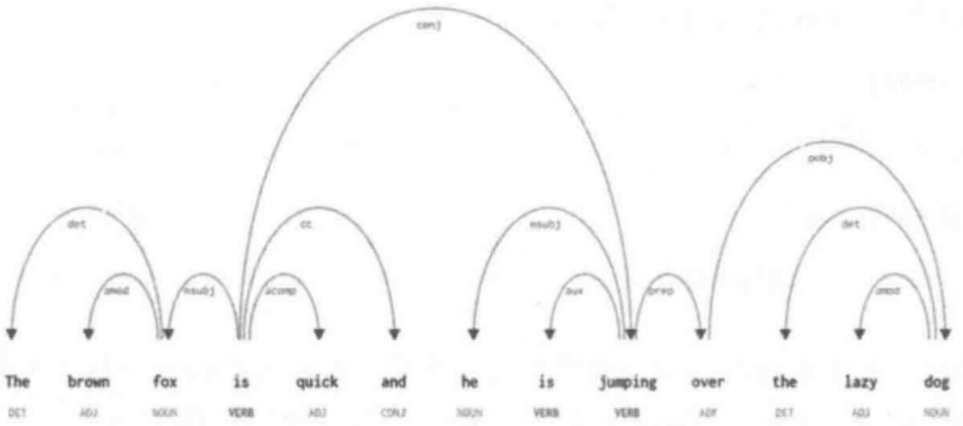


图 3-3 例句的依存关系语法示意图

在本节中，我们将介绍一些基于依存语法的分析，使我们进一步了解文本标识之间的句法和语义。

1. 依存关系分析器推荐

我们将使用几个库来生成基于依存关系的分析树，并对我们的例句进行测试。首先，我们将使用 spacy 库来分析我们的例句，生成所有标识及其依存关系。图 3-3 是使用 spacy 的输出结果生成的，其中添加了一些漂亮的 CSS 以使依存关系看起来更加清晰易懂。

以下代码段展示了如何从例句中获取每个标识的依存关系：

```
sentence = 'The brown fox is quick and he is jumping over the lazy dog'

# load dependencies
from spacy.en import English
parser = English()
parsed_sent = parser(unicode(sentence))

# generate dependency parser output
In [131]: dependency_pattern = '{left}<---{word}[{w_type}]---
>{right}\n--------'
     ...: for token in parsed_sent:
     ...:     print dependency_pattern.format(word=token.orth_,
     ...:                                     w_type=token.dep_,
     ...:                                     left=[t.orth_
     ...:                                           for t
     ...:                                           in token.lefts],
     ...:                                     right=[t.orth_
     ...:                                            for t
     ...:                                            in token.rights])
[]<---The[det]--->[]
--------
[]<---brown[amod]--->[]
--------
[u'The', u'brown']<---fox[nsubj]--->[]
--------
[u'fox']<---is[ROOT]--->[u'quick', u'and', u'jumping']
--------
[]<---quick[acomp]--->[]
--------
[]<---and[cc]--->[]
--------
[]<---he[nsubj]--->[]
--------
[]<---is[aux]--->[]
--------
```

```
[u'he', u'is']<---jumping[conj]--->[u'over']
--------
[]<---over[prep]--->[u'dog']
--------
[]<---the[det]--->[]
--------
[]<---lazy[amod]--->[]
--------
[u'the', u'lazy']<---dog[pobj]--->[]
--------
```

上面的输出显示了所有的标识及其依存关系类型，左箭头指向其左侧的依存项，右箭头指向其右侧的依存项。如果你将输出结果中的每一行与前面的依存关系树示意图进行比较，你会发现它们有很多相似之处。如果你忘记了每个依存标签所代表的含义，可以快速回顾一下第 1 章中的相关内容。

接下来，我们将使用 `nltk` 和斯坦福分析器为例句生成依存关系树，如下面的代码段所示：

```
# set java path
import os
java_path = r'C:\Program Files\Java\jdk1.8.0_102\bin\java.exe'
os.environ['JAVAHOME'] = java_path

# perform dependency parsing
from nltk.parse.stanford import StanfordDependencyParser
sdp = StanfordDependencyParser(path_to_jar='E:/stanford/stanford-parser-
full-2015-04-20/stanford-parser.jar',
                               path_to_models_jar='E:/stanford/stanford-
parser-full-2015-04-20/stanford-parser-3.5.2-models.jar')
result = list(sdp.raw_parse(sentence))

# generate annotated dependency parse tree
In [134]: result[0]
Out[134]:
```

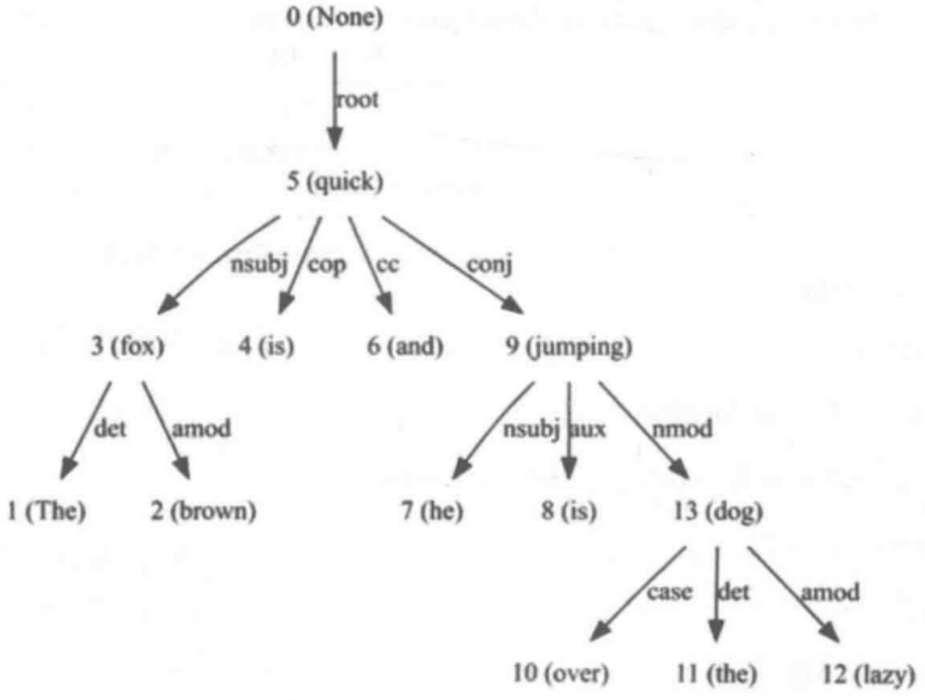


图 3-4 附带标注的例句依存关系分析树

```
# generate dependency triples
Out[136]:
[((u'quick', u'JJ'), u'nsubj', (u'fox', u'NN')),
 ((u'fox', u'NN'), u'det', (u'The', u'DT')),
 ((u'fox', u'NN'), u'amod', (u'brown', u'JJ')),
 ((u'quick', u'JJ'), u'cop', (u'is', u'VBZ')),
 ((u'quick', u'JJ'), u'cc', (u'and', u'CC')),
 ((u'quick', u'JJ'), u'conj', (u'jumping', u'VBG')),
 ((u'jumping', u'VBG'), u'nsubj', (u'he', u'PRP')),
 ((u'jumping', u'VBG'), u'aux', (u'is', u'VBZ')),
 ((u'jumping', u'VBG'), u'nmod', (u'dog', u'NN')),
 ((u'dog', u'NN'), u'case', (u'over', u'IN')),
 ((u'dog', u'NN'), u'det', (u'the', u'DT')),
 ((u'dog', u'NN'), u'amod', (u'lazy', u'JJ'))]
# print simple dependency parse tree
In [137]: dep_tree = [parse.tree() for parse in result][0]
     ...: print dep_tree
(quick (fox The brown) is and (jumping he is (dog over the lazy)))

# visualize simple dependency parse tree
In [140]: dep_tree.draw()
Out [140]:
```

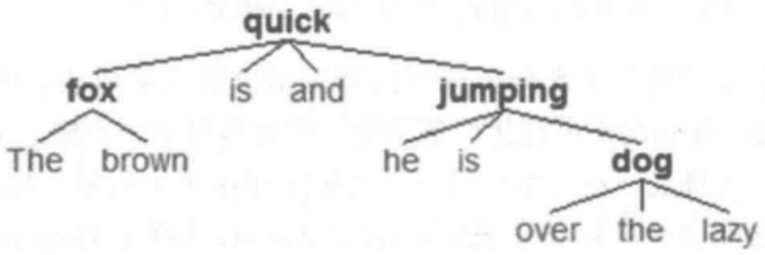


图 3-5 简化的例句依存关系分析树

上面的输出结果展示了我们如何轻松地为例句生成依存分析树，并分析和理解标识间的关系。斯坦福分析器是十分强大且稳定的，它能够很好地与 nltk 集成。在这里，有一点需要说明，那就是你需要安装 graphviz 才能够生成如图 3-4 所示的带标注的依存关系树。

2. 建立自己的依存关系分析器

从头开始构建自己的依存关系分析器并不容易，因为你需要大量的、充足的数据，而且仅仅按照语法产生式规则来检查并不总是能够很好地评估分析器效果。下面的代码段展示了如何构建自己的依存关系分析器。我们首先利用 nltk 的 DependencyGrammar 类从用户输入语法获得产生式规则。然后，我们使用 ProjectiveDependencyParser（一个映射式的、基于产生式规则的依存关系分析器）来执行基于依存关系的分析：

```
import nltk
tokens = nltk.word_tokenize(sentence)

dependency_rules = """
'fox' -> 'The' | 'brown'
'quick' -> 'fox' | 'is' | 'and' | 'jumping'
'jumping' -> 'he' | 'is' | 'dog'
'dog' -> 'over' | 'the' | 'lazy'
"""
dependency_grammar = nltk.grammar.DependencyGrammar.fromstring(dependency_
rules)
# print production rules
In [143]: print dependency_grammar
```

```
Dependency grammar with 12 productions
  'fox' -> 'The'
  'fox' -> 'brown'
  'quick' -> 'fox'
  'quick' -> 'is'
  'quick' -> 'and'
  'quick' -> 'jumping'
  'jumping' -> 'he'
  'jumping' -> 'is'
  'jumping' -> 'dog'
  'dog' -> 'over'
  'dog' -> 'the'
  'dog' -> 'lazy'

# build dependency parser
dp = nltk.ProjectiveDependencyParser(dependency_grammar)

# parse our sample sentence
res = [item for item in dp.parse(tokens)]
tree = res[0]

# print dependency parse tree
In [145]: print tree
 (quick (fox The brown) is and (jumping he is (dog over the lazy)))
```

你可以看出，上面的依存关系分析树与由斯坦福分析器生成的分析树是相同的。事实上，你可以使用 `tree. draw()`来可视化树形结构，并将其与上一个树形结构进行比较。分析器的扩展性一直是一个挑战，在大型项目中，大量工作用来生成基于依存语法规则的系统。一些例子包括词汇功能语法（Lexical Functional Grammar，LFG）Pargram 项目和词汇化树形联结语法（Lexicalized Tree Adjoining Grammar）XTAG 项目。

3.3.6 基于成分结构的分析

基于成分结构的语法常用来分析和确定语句的组成成分。此外，这种语法的另一个重要用途是找出这些组成成分的内部结构以及它们之间的关系。对于不同类型的短语，根据其包含的组件类型，通常会有几种不同的短语规则，我们可以使用它们来构建分析树。如果你需要温习相关内容并查阅一些分析树示例，请翻看 1.3.4 节下的“成分语法”内容。

基于成分结构的语法可以帮助我们将句子分解成各种成分。然后，我们可以进一步将这些成分分解成更细的细分项，并且重复这个过程直至将各种成分分解成独立的标识或单词。这些语法具有各种产生式规则，一般而言，一个与上下文语境无关的语法（CFG）或短语结构语法就足以完成上述操作。

一旦我们拥有了一套语法规则，就可以构建一个成分结构分析器，它根据这些规则处理输入的语句，并辅助我们构建分析树。分析器是为语法赋予生命的东西，也可以说是语法的程序语言解释。目前，有各种类型的分析算法，包括如下几类：

- 递归下降解析（Recursive Descent parsing）。
- 移位归约解析（Shift Reduce parsing）。
- 图表解析（Chart parsing）。
- 自下而上解析（Bottom-up parsing）。
- 自上而下解析（Top-down parsing）。
- PCFG 解析（PCFG parsing）。

受本书篇幅限制，我们不可能去详细介绍所有的分析算法。nltk 在其官方说明书中提供了关于它们的详细信息，你可参见 http://www.nltk.org/book/ch08.html。稍后，当我们执行自己的分析器时，我会简要地介绍部分分析器，并重点介绍 PCFG 解析。*递归下降解析*通常遵循自上而下的解析方法，它从输入语句中读取标识，然后尝试将其与语法产生式规则中的终结符进行匹配。它始终超前一个标识，并在每次获得匹配时，将输入读取指针前移。

*移位归约解析*遵循自下而上的解析方法，它找出与语法产生式规则右侧一致的标识序列（单词或短语），然后用该规则左侧的标识替换它。这个过程一直持续，直到整个句子只剩下形成分析树的必要项。

*图表解析*采用动态规划，它存储中间结果，并在需要时重新使用这些结果，以获得显著的效能提升。这种情况下，图表分析器存储部分解决方案，并在需要时查找它们以获得完整的解决方案。

1. 成分结构分析器推荐

在这里，我们将使用 nltk 和 StanfordParser 来生成分析树。在运行代码以解析例句之前，首先设置 Java 路径，然后将显示并可视化分析树，它将会与第 1 章中基于成分语法的分析树非常相似。下面的代码段说明了上述过程：

```
# set java path
import os
java_path = r'C:\Program Files\Java\jdk1.8.0_102\bin\java.exe'
os.environ['JAVAHOME'] = java_path

sentence = 'The brown fox is quick and he is jumping over the lazy dog'

from nltk.parse.stanford import StanfordParser
# create parser object
scp = StanfordParser(path_to_jar='E:/stanford/stanford-parser-
full-2015-04-20/stanford-parser.jar', path_to_models_jar='E:/stanford/
stanford-parser-full-2015-04-20/stanford-parser-3.5.2-models.jar')

# get parse tree
result = list(scp.raw_parse(sentence))

# print the constituency parse tree
In [150]: print result[0]
     ...:
(ROOT
  (NP
    (S
      (S
        (NP (DT The) (JJ brown) (NN fox))
        (VP (VBZ is) (ADJP (JJ quick))))
      (CC and)
      (S
        (NP (PRP he))
        (VP
          (VBZ is)
          (VP
            (VBG jumping)
            (PP (IN over) (NP (DT the) (JJ lazy) (NN dog)))))))))

# visualize constituency parse tree
In [151]: result[0].draw()
Out [151]:
```

上面的代码段向我们展示了如何为句子构建基于成分语法的分析树。请注意，图 3-6 中所示的分析树与依存关系分析树是显著不同的，它与第 1 章中展示的成分结构分析树是相匹配的。请注意树中显示的嵌套和层次结构，它们是成分结构分析树的典型特征。

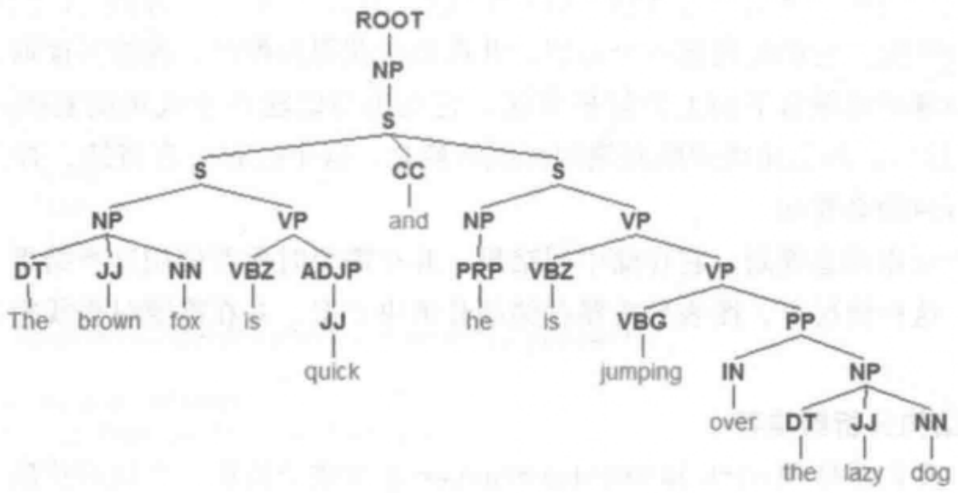


图 3-6 例句的成分结构分析树

2. 构建自己的成分结构分析器

构建自己的成分结构分析器有各种各样的方法，包括创建 CFG 产生式规则，然后使用该语法规则构建分析器等。想要构建自己的 CFG，你可以使用 `nltk.CFG.fromstring` 函数来输入产生式规则，然后再使用 `ChartParser` 或 `RecursiveDescentParser` 分析器（它们均属于 `nltk` 包）。欢迎构建一些语法规则，并试用这些分析器。

我们将会考虑构建一个扩展性良好且运行高效的成分结构分析器。常规 CFG 分析器（如图表分析器、递归下降分析器）的问题是解析语句时很容易被大量的解析工作量所压垮，导致运行速度非常缓慢。而这正是像 PCFG（概率上下文无关语法，Probabilistic Context Free Grammar）这样的加权语法和像维特比分析器这样的概率分析器在运行中被证明更有效的地方。PCFG 是一种上下文无关的语法，它将每个产生式规则与一个概率值相关联。从 PCFG 产生一个分析树的概率是每一个产生式规则概率的乘积。

我们将使用 `nltk` 的 `ViterbiParser` 来训练 `treebank` 语料库中的分析器，`treebank` 语料库为每个句子提供了带注释的分析树。这个分析器是一个自下而上的 PCFG 分析器，它使用动态规划来查找每个步骤中最有可能的解析结果。我们可以通过加载必要的训练数据和依存关系来开始构建我们的分析器：

```
import nltk
from nltk.grammar import Nonterminal
from nltk.corpus import treebank

# get training data
training_set = treebank.parsed_sents()

# view a sample training sentence
In [161]: print training_set[1]
(S
  (NP-SBJ (NNP Mr.) (NNP Vinken))
  (VP
    (VBZ is)
    (NP-PRD
```

```
      (NP (NN chairman))
      (PP
        (IN of)
        (NP
          (NP (NNP Elsevier) (NNP N.V.))
          (, ,)
          (NP (DT the) (NNP Dutch) (VBG publishing) (NN group))))))
  (. .))
```

现在，我们从标记和注释完的训练句子中获取规则，并构建我们语法的产生式规则：

```
# extract the productions for all annotated training sentences
treebank_productions = list(
                        set(production
                            for sent in training_set
                            for production in sent.productions()
                        )
                    )

# view sample productions
In [166]: treebank_productions[0:10]
Out[166]:
[VBZ -> 'cites',
 VBD -> 'spurned',
 PRN -> , ADVP-TMP ,,
 NNP -> 'ACCOUNT',
 JJ -> '36-day',
 NP-SBJ-2 -> NN,
 JJ -> 'unpublished',
 NP-SBJ-1 -> NNP,
 JJ -> 'elusive',
 NNS -> 'Lids']

# add productions for each word, POS tag
for word, tag in treebank.tagged_words():
        t = nltk.Tree.fromstring("("+ tag + " " + word  +")")
        for production in t.productions():
                treebank_productions.append(production)

# build the PCFG based grammar
treebank_grammar = nltk.grammar.induce_pcfg(Nonterminal('S'),
                                            treebank_productions)
```

现在我们有了必要的语法与产生式规则，我们将使用以下代码段，通过语法训练创建自己的分析器，然后使用例句对分析器进行评估：

```
# build the parser
viterbi_parser = nltk.ViterbiParser(treebank_grammar)

# get sample sentence tokens
tokens = nltk.word_tokenize(sentence)

# get parse tree
In [170]: result = list(viterbi_parser.parse(tokens))
Traceback (most recent call last):
  File "<ipython-input-170-c2cdab3cd56c>", line 1, in <module>
    result = list(viterbi_parser.parse(tokens))
  File "C:\Anaconda2\lib\site-packages\nltk\parse\viterbi.py", line 112, in
parse
    self._grammar.check_coverage(tokens)
ValueError: Grammar does not cover some of the input words: u"'brown',
'fox', 'lazy', 'dog'".
```

不幸的是，在尝试用新建的分析器解析例句的标识时，我们收到了一个错误提示。错误的原因很明显：例句中的一些单词不包含在基于 treebank 的语法中，因为这些单词并不在我们的 treebank 语料库中。由于该语法使用 POS 标签和短语标签来构建基于训练数据的分析树，我们将在语法中为例句添加标识和 POS 标签，然后重新构建分析器：

```
# get tokens and their POS tags
from pattern.en import tag as pos_tagger
tagged_sent = pos_tagger(sentence)
# check the tokens and their POS tags
In [172]: print tagged_sent
    ...:
[(u'The', u'DT'), (u'brown', u'JJ'), (u'fox', u'NN'), (u'is', u'VBZ'),
(u'quick', u'JJ'), (u'and', u'CC'), (u'he', u'PRP'), (u'is', u'VBZ'),
(u'jumping', u'VBG'), (u'over', u'IN'), (u'the', u'DT'), (u'lazy', u'JJ'),
(u'dog', u'NN')]

# extend productions for sample sentence tokens
for word, tag in tagged_sent:
    t = nltk.Tree.fromstring("("+ tag + " " + word  +")")
    for production in t.productions():
                treebank_productions.append(production)

# rebuild grammar
treebank_grammar = nltk.grammar.induce_pcfg(Nonterminal('S'),
                                            treebank_productions)
# rebuild parser
viterbi_parser = nltk.ViterbiParser(tbank_grammar)

# get parse tree for sample sentence
result = list(viterbi_parser.parse(tokens))

# print the constituency parse tree
In [178]: print result[0]
(S
  (NP-SBJ-163 (DT The) (JJ brown) (NN fox))
  (VP
    (VBZ is)
    (PRT (JJ quick))
    (S
      (CC and)
      (NP-SBJ (PRP he))
      (VP
        (VBZ is)
        (PP-1
          (VBG jumping)
          (NP (IN over) (DT the) (JJ lazy) (NN dog))))))) (p=2.02604e-48)

# visualize the constituency parse tree
In [179]: result[0].draw()
Out [179]:
```

现在，我们成功地为例句生成了分析树。你可以在图 3-7 中看到成分结构树的可视化结果。请记住，这是一个概率 PCFG 分析器，你可以在之前的输出结果中看到这棵树的总体概率。这里的标签注释均基于我们前面讨论过的树库注释。到此为止，就介绍完了构建成分结构分析器的方法。

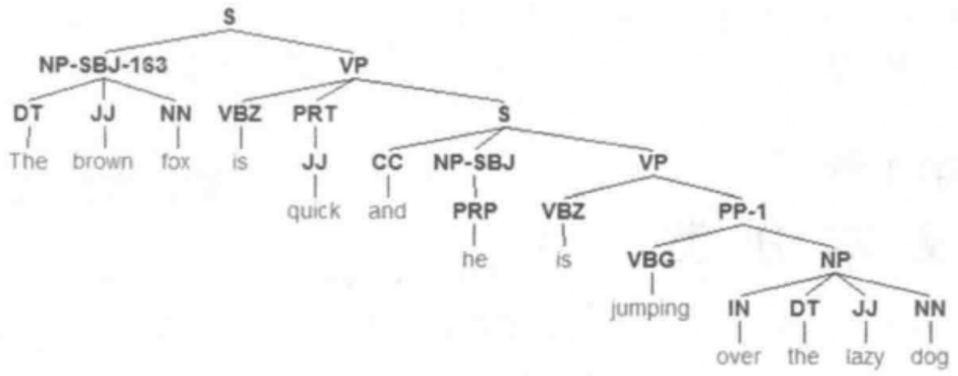


图 3-7 基于树库注释的例句成分结构树

3.4 小结

祝贺你完成了本章的学习。在本章中我们介绍了一些关于文本处理、语义分析和文本理解的概念、技术和实现方法。现在，你应该对于第 1 章中介绍的许多概念有了更清晰的理解，因为我们实际上已经在本章的实际案例中应用了它们。

本章主要介绍了两方面的内容，我们研究了与文本处理和规范化有关的概念。你现在应该已经知道了文本处理及规范化的重要性，随着本书内容的继续深入，在后续的章节中，你将会理解拥有处理良好且经过规范化的文本数据的重要性。我们介绍了各种相关概念，并应用了词语切分和文本规范化的技术，包括清洗和校正文本（如拼写和缩写词）。我们还在同一个上下文语境中建立了自己的拼写修正器和缩略词扩展器，提出了一种利用 WordNet 和重复字符校正单词的方法。之后，我们研究了一些词干提取和词形还原的概念和技术。最后，我们聚焦于分析和理解文本语法和结构，并再次涉及了第 1 章中的概念，包括 POS 标注、浅层解析、基于依存关系的分析和基于成分结构的分析。

现在，你已经知道了如何在现实世界的文本数据上使用标签器和分析器，以及构建你自己的标签器和分析器。在接下来的几章中，我们将使用各种机器学习技术（包括分类、聚类和摘要），深入研究如何从文本中分析和解读洞见。

第4章
文本分类

学习处理和理解文本是踏上从文本中获取有意义的洞见之旅的第一步。尽管理解如何组织语言和特殊的文本句法模式非常重要，但这对于想要得到有用模式和洞见、最大化使用海量文本数据的企业和组织来讲还是不够充分。语言处理的知识以及分析和机器学习的概念有助于构建使用文本数据的系统，该系统可以帮助解决实际问题，给企业带来商业利益。

机器学习包括有监督学习、无监督学习、强化学习和最新的深度学习技术。每个机器学习的概念涉及多个技术和算法，可以在文本数据上应用这些技术和算法，建立一个可以自学习而不需要太多人工监督的系统。在第3章通过建立我们自己的分析器和标签器，你可以体会到机器学习模式是数据和算法的组合。机器学习的优势是一旦我们训练好了模型，就可以直接在新的以前未见到的数据上应用这个模型，以获得有意义的洞见和期望的结果。

最相关和最有挑战的问题之一是*文本分类或归类*，这涉及基于每个文件的内在属性或特性，尽量将文本文件划分为不同的类别。该技术可用在不同的领域，包括垃圾邮件识别和新闻分类。如果你只有少量的文档，你可以查看每个文档，知道其内容，尽量去标识它们，分类这个概念也许就比较简单。基于这些知识，你可以将相似的文档归到不同的类型。当需要分类的文档数量增加到几十万或百万时，这将更具挑战。像特征提取、有监督学习和无监督学习这些技术都能派上用场。文档分类是一个通用的问题，不仅仅局限在文本，还可以拓展到其他方面，像音乐、图像、视频和其他媒体。

为了更清楚地把问题形式化，我们将使用一组给定的类或类别及若干文本文件。请记住，文件基本上由句子或段落文本组成。这就形成了一个语料库。我们的任务是确定每个文档属于哪一类或哪些类。整个过程包括几个步骤，这将在本章后面详细讨论。简言之，对于有监督分类问题，我们需要一些标记的数据用来训练文本分类模型。这些数据基本上是组织好的文件，已经预先分配给某些特定的类或类别。使用这些数据，我们可以从每个文档提取特征和属性，通过将这些数据送入一个有监督学习算法，使模型学习这些与每个文档对应的属性和分类或类别。当然，在建立模型前，数据需要预处理和规范化处理。一旦处理完毕，我们将对新文档采用同样的规范化处理和特征提取方法，然后将这些特征送入模型，预测这些文档的分类或类型。然而，对于无监督学习问题，我们基本上没有预先标注好的训练文档。基于文档内在的特性，我们将使用像聚类、文档相似性度量技术实现文档聚类，并为文档分配类型。

本章将讨论文本分类的概念，以及如何将其形式化为一个有监督机器学习问题。我们还将讨论各种形式的分类和它们所说明的内容。我们将清晰描述完成文本分类工作流程的关

键步骤，这将涉及前面还没讨论到的一些关键步骤，包括特征提取、分类器、模型评估，最后我们将这些关键步骤整合在一起，建立一个基于真实数据的文本分类系统。

4.1 什么是文本分类

在定义文本分类以前，我们需要理解文本数据的范围，以及分类的真实含义。这里的文本数据可以是短语、句子或者包含文本段落的整篇文档等任何形式，这些数据可以从语料库、博客或互联网的任何地方获得。文本分类也经常称为*文档分类*，*文档*这个词概括了任何形式的文本内容。*文档*这个词可以定义为思想或事件的一些具体的表示，这些表示可以是书面、语言记录、绘画或演讲等形式。这里，使用*文档*这个词来表示文本数据，例如英语中的句子或段落。

文本分类也为称为*文本归类*，这里使用*文本分类*这个词有两个原因。第一个原因是我们要分类文档，文本分类和文本归类具有相同的本质。第二个原因是我们将用分类或有监督机器学习方法来分类或归类文档。文本分类具有很多方法。我们将会集中精力解释用于分类的有监督方法。分类过程不只局限于文本，还广泛用于其他领域，包括科学、健康、天气预测和技术等。

假设有一个预定义的类集合，文本或文档分类是将文档指定到一个或多个分类或类型的过程。这里的文档就是文本文档，每个文档包含单词组成的句子或段落。一个文本分类系统基于文档的内置属性，能够成功地将每个文档分类到正确的类别中。数学上，可以做如下定义：假设 d 是文档 D 的描述或属性，$d \in D$，我们具有一组预先定义的类别或分类 $C = \{c_1, c_2, c_3, \cdots, c_n\}$。真实的文档 D 可能拥有很多内在的属性，这使得 D 成为高维空间的一个实体。使用这个空间的一个子集，其是包含一组有限的描述或特征的集合，表示为 d，可以使用文本分类系统 T 成功地将原始文档 D 划分到正确的类型 C_x。这可以表示为 T：$D \rightarrow C_x$。

本章后面详细介绍文本分类系统。图 4-1 是文本分类过程高层次的概念表示。

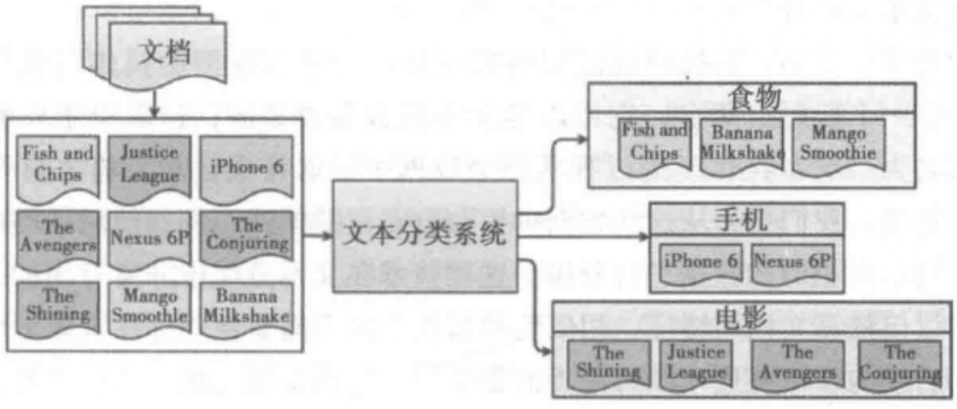


图 4-1 文本分类概念图

如图 4-1 所示，我们看到表示产品的几个文档分为食物、移动电话和电影几个类别。最初，正如文本语料库自身包含不同的文档一样，这些文档放置在一起。通过一个文本分类系统之后，这里用通过黑色的方框表示，我们看到每个文档属于预先定义的一个类或类别中。这里使用文档的名称来代表文档，但是实际数据中，文档可能包含丰富的信息使得文档识别与分类更加容易，这些信息包括电影的类型、产品的说明与组成，以及很多可以作为分类系统特征的属性。

文本分类具有很多划分方法，本章主要介绍两种基于文档内容类型的分类：

- 基于内容的分类。
- 基于请求的分类。

这两类的差异在于文本文档分类方法背后的思想或理念，而不在于具体的技术算法与过程。基于内容的分类是根据文本内容主题或题目的属性或权重来进行文档分类的。举一个概念性的例子，一本书有30%以上的内容是关于食物准备的，这本书可以归为烹饪/识谱类。基于请求的分类受到用户需求的影响，其目标是特定的用户群和读者。这类分类受到特殊策略和思想的控制。

4.2 自动文本分类

现在我们已经对文本分类的定义和范围有所了解。当我们提到“文本分类系统”可以将文本文件划分到它们代表的类或类别时，我们也从该概念和数学上对文本分类进行了正式的定义。假设几个人通过浏览每个文档并进行分类完成文本分类任务，那么他们就是我们所讨论的文档分类系统的一部分。然而，一旦文档数量超过百万并且需要快速进行分类处理时，该方法则不能很好地扩展。为了使文本分类的过程更加高效和快速，我们需要思考文本分类任务的自动化，这给我们带来了自动文本分类。

为实现自动文本分类，可以充分利用一些机器学习的技术和概念。这里主要有两类与解决该问题相关的技术：

- 有监督机器学习。
- 无监督机器学习。

此外，还有一些其他的机器学习算法家族，例如强化学习和半监督学习。接下来，让我们更加深入地了解有监督机器学习和无监督机器学习算法，从机器学习方面了解如何利用这些算法进行文本文件分类。

无监督学习指的是不需要提前标注训练数据样本来建立模型的具体的机器学习技术或算法。通常，有一个数据点集合，它可以是文本或数字类型的，这取决于要解决的具体问题。我们通过名为“特征提取”的过程从每个数据中提取特征，然后将来自于每个数据的特征集合输入算法。我们尽力从这些数据中提取有意义的模式，例如使用聚类或基于主题模型的文本摘要技术对相似的数据进行分组。这项技术在文本分类中非常有用的，也称为文档聚类，即我们仅仅依靠文档的特征、相似度和属性，而不需要使用标注数据训练任何模型进行文档分组。后续的章节将进一步讨论无监督学习，包括主题建模、文档摘要、相似性分析和聚类。

有监督学习指的是训练预标注数据样本（也称为训练数据）的具体机器学习技术或算法。使用特征提取从数据中提取特征或属性，对于每个数据点，我们将拥有特征集和对应的类型/标签。算法从训练数据中学习每个分类的不同模式。学习完成后，我们得到一个训练好的模型。一旦我们将未来测试数据样本的特征送入这个模型，模型就可以预测这些测试数据样本的分类。这样机器就学会了如何基于训练的数据样本预测未知的新数据样本的分类。

目前，有两种主要的有监督学习算法。

- 分类：当预测的输出是离散的类型时，有监督学习的过程称为分类，因此这种情况下输出变量是类别的变量。这样的例子有新闻分类或电影分类。
- 回归：当我们希望输出的结果是连续的数值变量时，有监督机器学习算法称为回归算法。这样的例子有房屋价格或人的体重。

如本章章名所述，我们将主要关注分类问题，尽力将文本文件划分为离散的类别或分类。我们将在后面的实现中继续讨论有监督学习方法。

现在，我们已经准备好从数学上对自动或基于机器的文本分类过程进行定义。有一个文档集合，集合中文档带有相应的类别或分类标签。这个集合可以用 TS 表示，这是一个文档和标签对的集合，$TS=\{(d_1,c_1),(d_2,c_2),\cdots,(d_n,c_n)\}$，其中 d_1，d_2，…，d_n 是文本列表，c_1，c_2，…，c_n 是这些文本对应的类型。这里 $c_x \in C=\{c_1, c_2, \cdots, c_n\}$，其中 c_x 表示文档 x 对应的类型，C 表示所有可能离散分类的集合，集合中任何元素可能是文档的一个或多个类型。假设我们已经拥有了训练数据集，我们可以定义一个有监督学习算法 F，当算法在训练数据 TS 集上训练之后，我们得到训练好的分类器 γ，可以表示为 $F(TS)=\gamma$。因此，有监督学习算法 F 使用输入集（document，class）对 TS，得到训练的分类器 γ——这就是我们的模型。上述过程就称之为训练过程。

这个模型输入一个新的、未知的文档 ND，可以预测文档的类型 c_{ND}，使得 $c_{ND} \in C$。这一过程称为预测过程，可以表示为 γ：$TD \rightarrow c_{ND}$。这样我们看到有监督文本分类过程有两个主要的过程：

- 训练。
- 预测。

一个需要记住的关键点就是有监督文本分类也需要一些手工标注的训练数据，即使我们谈的是自动文本分类，我们也需要一些手工的工作以启动我们的自动处理。当然，这一点的收益也是多方面的，我们可以使用较少的努力和人力监督来不停地进行新文档的预测与分类。

下面的章节将讨论不同的学习方法或算法。这些算法不仅针对文本数据，它们是通用的机器学习算法，可以应用到各类经过预处理和特征提取的数据上。我们将涉及很多有监督的机器学习算法，并使用它们解决真实的文本分类问题。这些算法通常在训练数据集上进行训练，在一个可选的验证数据集上进行验证以避免模型在训练数据上过拟合。过拟合基本的意思是对于新的文本实例，模型不能很好地推广和准确地预测。通常，我们会使学习算法多次调优模型内部参数，使用性能度量（如验证集的准确率）或使用交叉验证来评估性能。交叉验证时，使用随机采样将训练数据分为训练集和验证集。这些构成了训练过程，其输出是完全训练好的模型，可以进行预测。在预测阶段，一般使用测试数据集中新的数据。在规范化处理和特征提取之后，将它们送入训练好的模型，然后通过评估预测性能来观察模型执行的好坏程度。

基于预测类型的数量和预测的本质，有多种文本分类。这些类型基于数据集、与数据集相关的类型或类别数量、数据点上可以预测的类型数量。

- 二元分类是当离散类型或类别的数量是 2 时，任何预测可以是二者之一。
- 多类分类也称为多元分类，指的是当一个问题中类型的数量超过 2 时，每个预测给出这些类型中的一个类或类别。当全部类型数量超过 2 时，这是二元分类问题的一个扩展。

- 多标签分类指的是对于任何数据，每个预测结果可以产生多个结果/预测类型。

4.3 文本分类的蓝图

现在我们已经了解了自动文本分类的基本范围，本节将看一看建立自动文本分类系统完整流程的蓝图。这包括在前面章节提到的训练和测试阶段必须要完成的一系列步骤。为建立文本分类系统，需要确认我们已经拥有数据来源并获取了这些数据，可以开始将这些数据送入系统。假设已经下载了数据集，并且准备好了数据，下面给出一个文本分类系统典型工作流程的主要步骤：

（1）准备训练和测试数据。

（2）文本规范化处理。

（3）特征抽取。

（4）模型训练。

（5）模型预测与评估。

（6）模型部署。

为建立文本分类器，需要按照顺序执行这些步骤。图 4-2 显示了文本分类系统详细的工作流程，主要处理过程突出地显示在训练和预测部分。

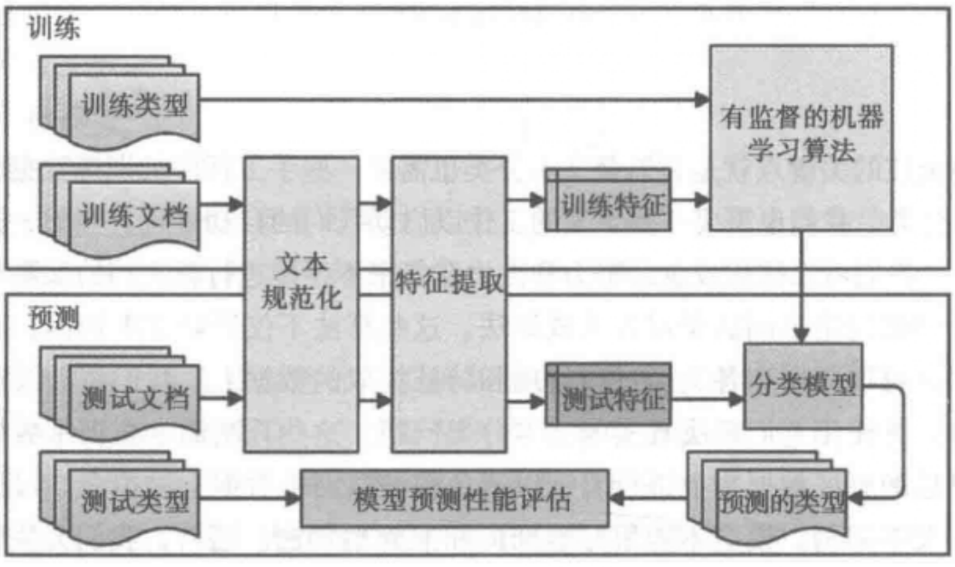


图 4-2 构建一个自动文本分类系统的蓝图

注意，这里有训练和预测两个主要的矩形框，它们表示的是建立文本分类器的两个主要过程。通常情况下，将数据集划分为两个或三个部分，分别称为训练集、验证集（可选）和测试集。在图 4-2 中，可以看到两个过程都用到了文本规范化处理和特征提取模块，这说明无论我们想对哪个文档进行分类或预测它的类型，这个文档在训练和预测阶段都必须执行相同的转换处理。首先对每个文档进行预处理和规范化处理，然后提取与该文档有关的特征。这些过程在训练过程和预测过程总是保持一致，以确保分类模型保持预测的一致性。

在训练过程中，每个文档都有对应的分类或类型，这些分类或类型是提前手工标注和组织的。这些训练文档在文本规范化模块中处理和规范化，输出整齐和标准化的文档。接着，将它们送入特征提取模块，这一模块使用不同的特征提取技术从文档中提取有意义的特征。

下面的章节会介绍主流的特征提取技术。因为标准的机器学习算法处理的是数字向量，所以这些特征通常是数组或向量。一旦获得特征后，就可以选择有监督的学习方法并训练模型了。

训练模型过程需要将文档的特征向量和每个文档对应的标签送入，使得算法可以学习每个分类或类型对应的不同模式，可以重用这些学习到的知识预测未来新文档的分类。一般情况下，使用一个可选的验证数据集来评估分类算法的性能，以确保算法使用训练过程中的数据获得较好的推广能力。训练过程结束后，这些特征和机器学习算法的组合产生了分类模型。通常情况下，会使用不同的参数对这个模型进行调优以获得一个性能更好的模型，这一过程称为*超参数调优*。

图 4-2 中显示的预测过程包括预测新文档的类型或评估在测试数据上预测的工作原理两个部分。测试数据集文档经过同样的规范化处理和特征提取过程后，这些文档的特征被送到训练好的分类模型，这个模型根据前期训练好的模式预测每个文档可能的类标签。如果有手工标注的这些文档的真实类标签，你可以通过使用不同度量标准（比如准确率）比较真实标签和预测标签，评估这个模型的性能。这将反映模型对于新文档的预测性能。

一旦获得了一个稳定的、可工作的模型，最后一步通常是部署这个模型，这包括存储这个模型和相关依赖的文件，将模型部署为一个服务或者可执行程序，它批量预测新文档的类型，或以 Web 服务的形式满足用户请求。这里有很多不同的机器学习模型部署方法，这通常取决于你后续如何访问这些模型。

我们将讨论前面蓝图中的一些主要模块，并且实现这些模块，以便我们可以将这些模块集成在一起，建立一个真实的文本分类器。

4.4 文本规范化处理

第 3 章详细阐述了文本处理和规范化，参阅第 3 章中各种可用方法与技术。本节中，我们将定义一个规范化模块以处理文本文档规范化，并在后面建立分类器时使用这个处理模块。尽管有许多可用的技术，但是我们将坚持简化与直接原则，以便于很容易地一步一步参照这里的实现。我们将在模块中实现和使用下面的规范化技术：

- 扩展缩写词。
- 通过词形还原实现文本处理规范化。
- 去除特殊字符与符号。
- 去除停用词。

我们不再更多地关注拼写纠正及其他高级的技术，但如果你感兴趣，你可以集成这些前面章节实现的功能。我们实现了规范化处理模块，可以在本章代码文件夹的 `normalization.py` 文件中找到它。首先从载入一些依赖的模块开始。记住，你需要在第 3 章自定义的名为 `contractions.py` 的映射文件，来实现缩写词扩展。

下面的代码段显示了必要的引用和依赖项：

```
from contractions import CONTRACTION_MAP
import re
import nltk
import string
from nltk.stem import WordNetLemmatizer
stopword_list = nltk.corpus.stopwords.words('english')
wnl = WordNetLemmatizer()
```

上面的代码中，我们载入了英文的停用词、出自 CONTRACTION_MAP 的缩写映射和 WordNetLemmatizer 的一个实例来实现词形还原。现在，我们定义一个函数实现文本的切分，它将用在其他的规范化函数中。下面的函数实现词语切分，并去除分割后符号中的多余空格。

```
def tokenize_text(text):
    tokens = nltk.word_tokenize(text)
    tokens = [token.strip() for token in tokens]
    return tokens
```

定义扩展缩写词的函数。该函数与第 3 章实现的函数相似，输入文本，如果有匹配的缩写，则返回包含扩展缩写词后的文本。下面的代码段有助于实现这些：

```
def expand_contractions(text, contraction_mapping):

    contractions_pattern = re.compile('({})'.format('|'.join(contraction_
    mapping.keys())),
                                      flags=re.IGNORECASE|re.DOTALL)
    def expand_match(contraction):
        match = contraction.group(0)
        first_char = match[0]
        expanded_contraction = contraction_mapping.get(match)\
                               if contraction_mapping.get(match)\
                               else contraction_mapping.get(match.lower())
        expanded_contraction = first_char+expanded_contraction[1:]
        return expanded_contraction

    expanded_text = contractions_pattern.sub(expand_match, text)
    expanded_text = re.sub("'", "", expanded_text)
    return expanded_text
```

既然我们已经有了扩展缩写词的函数，接下来就可以实现一个使用词形还原函数把单词变换为词基或词根形式的函数以对文本进行规范化处理。下面的函数有助于实现这些：

```
from pattern.en import tag
from nltk.corpus import wordnet as wn

# Annotate text tokens with POS tags
def pos_tag_text(text):
    # convert Penn treebank tag to wordnet tag
    def penn_to_wn_tags(pos_tag):
        if pos_tag.startswith('J'):
            return wn.ADJ
        elif pos_tag.startswith('V'):
            return wn.VERB
        elif pos_tag.startswith('N'):
            return wn.NOUN
        elif pos_tag.startswith('R'):
            return wn.ADV
        else:
            return None

    tagged_text = tag(text)
    tagged_lower_text = [(word.lower(), penn_to_wn_tags(pos_tag))
                         for word, pos_tag in
                         tagged_text]
    return tagged_lower_text

# lemmatize text based on POS tags
def lemmatize_text(text):
```

```
    pos_tagged_text = pos_tag_text(text)
    lemmatized_tokens = [wnl.lemmatize(word, pos_tag) if pos_tag
                         else word
                         for word, pos_tag in pos_tagged_text]
    lemmatized_text = ' '.join(lemmatized_tokens)
    return lemmatized_text
```

上面的代码片段描述了两个词形还原函数。主函数是 lemmatize_text，该函数接受文本数据，基于每个词性标签还原词形，接着给用户返回词形还原处理后的文本。为实现这个功能，需要标注每个文本符号的词性标签。使用 pattern 函数库中的 tag 函数对每个符号标注词性标签，因为 WordNetLemmatizer 基于 WordNet 格式对词性标签进行检查，所以需要将词性标签从 Penn treebank 语法格式转换为 WordNet 语法格式。将每个单词符号转换为小写，纠正拼写，转换为 WordNet 词性标签，返回这些标注好的单词符号，最后将这些符号最后送入 lemmatize_text 函数。

下面的函数帮助我们实现了特殊符号和字符的去除：

```
def remove_special_characters(text):
    tokens = tokenize_text(text)
    pattern = re.compile('[{}]'.format(re.escape(string.punctuation)))
    filtered_tokens = filter(None, [pattern.sub('', token) for token in
    tokens])
    filtered_text = ' '.join(filtered_tokens)
    return filtered_text
```

我们通过文本切分去除了一些特殊字符，因此可以去除一些实际上是缩写的标识，但无法在第一步中去除“s”“re”等。我们将在去除停用词时去除它们。然而，也可以不通过文本切分来去除这些特殊字符。通过正则表达式匹配来去除 string. punctuation 中定义的特殊字符。下面的函数有助于去除文本数据中的停用词。

```
def remove_stopwords(text):
    tokens = tokenize_text(text)
    filtered_tokens = [token for token in tokens if token not in
    stopword_list]
    filtered_text = ' '.join(filtered_tokens)
    return filtered_text
```

既然已经定义了全部的函数，就可以通过将所有函数一个接一个地连接在一起的方式建立文本处理流水线。下面的函数实现上述功能，输入文本文档语料，进行规范化处理，返回规范化处理后的文本文档语料。

```
def normalize_corpus(corpus, tokenize=False):

    normalized_corpus = []
    for text in corpus:
        text = expand_contractions(text, CONTRACTION_MAP)
        text = lemmatize_text(text)
        text = remove_special_characters(text)
        text = remove_stopwords(text)
        normalized_corpus.append(text)
        if tokenize:
            text = tokenize_text(text)
            normalized_corpus.append(text)

    return normalized_corpus
```

至此，我们完成了文本规范化处理模块所需的全部函数的讨论和实现。下面来看一看特征提取的概念和实际实现。

4.5 特征提取

有很多特征提取技术可以应用到文本数据上，但在深入学习之前，先思考特征的意义。为什么我们需要这些特征？它们又如何发挥作用？数据集中通常包含很多数据。一般情况下，数据集的行和列是数据集的不同特征或属性，每行或者每个观测值都是特殊的值。在机器学习术语中，特征是独一无二的，是数据集中每个观测值或数据的可度量的属性或性质。特征通常具有数值的性质，可能是绝对值或者是列表中每个分类进行二进制编码的分类特征，这一过程称为一位有效（one-hot）编码过程。特征的提取和选择过程既是一门科学，也是一门艺术，这个过程也称为特征提取或特征工程。

通常情况下，为获取洞见，把提取到的特征送入机器学习算法以学习可以应用到新数据特征上的模式。因为每个算法的核心是数学上的优化操作，当算法从数据和观测值上学习模式时，是一个最小化误差和错误的过程，所以这些算法一般都期望特征是数值向量的形式。因此，处理文本数据增加的挑战就是如何转换文本数据并从中提取数值特征。

现在，我们看一些与文本数据有关的特征提取概念和技术。

向量空间模型是处理文本数据非常有用的概念和模型，并在信息检索与文档排序中广泛使用。向量空间模型也称为词向量模型，定义为文本文档转换与表示的数学或代数模型，作为形成向量维度的特定词项的数字向量。数学上定义如下，假设在文档向量空间 VS 中有一个文档 D。每个文档维度和列数量将是向量空间中全部文档中不同词项或单词的总数量。因此，向量空间可以表示为

$$VS = \{W_1, W_2, \cdots, W_n\}$$

其中，n 是全部文档中不同单词的数量。现在，可以把文档 D 在向量空间表示为

$$D = \{w_{D1}, w_{D2}, \cdots, w_{Dn}\}$$

其中，w_{Dn}表示文档 D 中第 n 个词的权重。这个权重是一个数量值，可以表示任何事，可以是文档中单词的频率、平均的出现频率，或者是 TF-IDF 权重（稍后介绍）。

下面将介绍和实现如下特征提取技术：

- 词袋模型。
- TF-IDF 模型。
- 高级词向量模型。

对于特征提取，需要记住的一个关键问题是，一旦建立一个使用一些转换和数学操作的特征提取器，就需要确保从新文档提取特征时重用同样的过程，不需要对新文档重新建立整个算法。对于每项技术，我们都将使用一个例子进行说明。请注意，对于本节中例子的具体实现，我们将使用 `nltk`、`genism` 和 `scikit-learn` 等函数库，你可以使用前面讨论的 `pip` 命令进行安装（如果你没有安装这些软件）。

特征提取的实现可以分为两个模块。`feature_extractors.py` 包括后面建立分类器时使用的通用函数。在 `feature_extraction_demo.py` 中通过一些实际的例子使用同样的函数说明每项技术如何工作。可以在代码文件中访问这些模块，我会在本节中一直使用相同的代码

以便于理解。我们将使用 CORPUS 变量中描述的以下文档提取特征，并建立一些向量化模型。为说明如何从新文档中提取特征（作为测试数据集的一部分），我们将在下面的代码段中使用 new_doc 变量中独立的文档。

```
CORPUS = [
'the sky is blue',
'sky is blue and sky is beautiful',
'the beautiful sky is so blue',
'i love blue cheese'
]

new_doc = ['loving this blue sky today']
```

4.5.1 词袋模型

词袋模型也许是从文本文档中提取特征最简单但又最有效的技术。这个模型的本质是将文本文档转化成向量，从而将每个文档转化成一个向量，这个向量表示在文档空间中全部不同的单词在该文档中出现的频率。因此，根据前面的数学定义，这里的例子向量记为 D，每个单词的权重与该词在文档中出现的频率相等。

有意思的事情是可以为单个单词出现频率和 n 元分词出现频率建立同样的模型，该模型就是 n 元分词词袋模型，它计算不同的 n 元分词在每个文档中的出现频率。

下面的代码片段给出了一个函数，实现了基于词袋的特征提取模块，该模块也接受 ngram_range 参数作为 n 元分词的特征。

```
from sklearn.feature_extraction.text import CountVectorizer

def bow_extractor(corpus, ngram_range=(1,1)):

    vectorizer = CountVectorizer(min_df=1, ngram_range=ngram_range)
    features = vectorizer.fit_transform(corpus)
    return vectorizer, features
```

上面的函数使用 CountVectorizer 类。可以在 http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html#sklearn.feature_extraction.text.CountVectorizer 地址访问详细的 API（应用程序接口）文档，根据你要提取的特征类型，该函数有一系列不同参数用于调优。我们使用默认的配置，这对大多数场景都是足够的，其中 min_df 设置为 1 表示在整个文档空间中最小频率为 1 的词项都会被考虑。可以设置 ngram_range 为不同的参数值，如（1，3），将建立包括所有 unigram、bigram 和 trigram 的向量空间。下面的代码片段显示函数在样本语料，即 4 个训练文档和 1 个测试文档中的执行情况。

```
# build bow vectorizer and get features
In [371]: bow_vectorizer, bow_features = bow_extractor(CORPUS)
     ...: features = bow_features.todense()
     ...: print features
[[0 0 1 0 1 0 1 0 1]
 [1 1 1 0 2 0 2 0 0]
 [0 1 1 0 1 0 1 1 1]
 [0 0 1 1 0 1 0 0 0]]

# extract features from new document using built vectorizer
In [373]: new_doc_features = bow_vectorizer.transform(new_doc)
     ...: new_doc_features = new_doc_features.todense()
```

```
    ...: print new_doc_features
[[0 0 1 0 0 0 1 0 0]]

# print the feature names
In [374]: feature_names = bow_vectorizer.get_feature_names()
    ...: print feature_names
[u'and', u'beautiful', u'blue', u'cheese', u'is', u'love', u'sky', u'so',
u'the']
```

上述输出显示每个文档如何转换为向量。每行代表语料库中的一个文档，我们对两个语料库均执行相同操作。使用 CORPUS 变量中的文档建立了向量生成器。用它来提取特征，使用这个建立的向量生成器从全新的文档中提取特征。向量的每一列描述的单词在 feature_names 变量中描述，每列的值是该词在文档中的频率。第一次看到它时可能难于理解，因此我们准备了下面的函数，它有助于更好地理解特征向量。

```
import pandas as pd

def display_features(features, feature_names):
    df = pd.DataFrame(data=features,
                      columns=feature_names)
    print df
```

现在可以将特征名字和向量送入这个函数，以一种比较易于理解的结构查看特征矩阵，如下所示：

```
In [379]: display_features(features, feature_names)
   and  beautiful  blue  cheese  is  love  sky  so  the
0    0          0     1       0   1     0    1   0    1
1    1          1     1       0   2     0    2   0    0
2    0          1     1       0   1     0    1   1    1
3    0          0     1       1   0     1    0   0    0

In [380]: display_features(new_doc_features, feature_names)
   and  beautiful  blue  cheese  is  love  sky  so  the
0    0          0     1       0   0     0    1   0    0
```

这使得事情变得更加清楚，对吧？考虑一下 CORPUS 的第二个文档，在上面的第一个表的第 1 行中表示。可以看到，‘sky is blue and sky is beautiful’ 这句话中，特征 sky 的值为 2，beautiful 值为 1，等等。文档中未出现的单词的值为 0。请注意，对于新的文档变量 new_doc，这句话中没有 today、this 或 loving 这些单词，因此没有这些词的特征。前面提到过这个原因，就是特征提取过程、模型和词汇总是基于训练数据，将不随着新文档变化或受其影响，这将用于后面的测试或其他语料的预测。你或许已经猜到这是因为一个模型总是基于训练数据进行训练，除非重新建立模型，否则模型不会受到新文档的影响。因此，这个模型的特征总是受限于训练语料的文档向量空间。

你已经了解了如何从文本数据中提取基于向量的有意义的特征，在从前看来这是不可能的。试着使用上面的函数，把 ngram_range 参数设置为（1，3），观察输出结果。

4.5.2 TF-IDF 模型

词袋模型还不错，但向量完全依赖于单词出现的绝对频率。这存在一些潜在的问题，语料库全部文档中出现次数较多的单词将会拥有较高的频率，这些词将会影响其他一些出现不如这些词频繁但对于文档分类更有意义和有效的单词。这就是 TF-IDF 的来源。TF-IDF 代

表的是词频 - 逆文档频率，是两个度量的组合：词频和逆文档频率。该技术最初作为显示搜索引擎用户查询结果排序函数的一个度量，现在已经成为信息检索和文本特征提取的一部分。

现在正式定义 TF-IDF，开始实现之前，看一下它的数学表示。数学上，TF-IDF 是两个度量的乘积，可以表示为 $tfidf = tf \times idf$，其中词频（*tf*）和逆文档频率（*idf*）是两个度量。

词频由 *tf* 表示，由词袋模型计算得出。任何文档的词频是该词在特定文档出现的原始频率值。数学上，词频可以表示为 $tf(w,D) = f_{w,D}$，其中 $f_{w,D}$ 表示单词 w 在文档 D 中的频率，这就是词频（*tf*）。有一些其他的词频表示与计算方法，例如将频率转化为二进制频率，其中 1 代表单词在文档中出现过，0 则代表没有出现过。有时，也可以通过对数运算或频率平均值将原始频率标准化。我们将在具体实现中使用原始频率。

逆文档频率由 *idf* 表示，是每个单词的文档频率的逆。该值由语料库中全部文档数量除以每个单词的文档频率，然后对结果应用对数运算变换其比例。在这里的实现中，将对每个单词的文档频率加 1，意味着词汇表中每个单词至少包含在一个语料库文档之中。这是为了避免被 0 除的错误，平滑逆文档频率。也对 *idf* 的计算结果加 1，避免被忽略单词拥有 0 值的 *idf*。数学上，*idf* 实现表示如下：

$$idf(t) = 1 + \log \frac{C}{1 + df(t)}$$

其中，$idf(t)$ 表示单词 t 的 *idf*，C 表示语料库中文档的总数量，$df(t)$ 表示包含单词 t 的文档数量频率。

因此，词频 - 逆文档频率可以通过把两个度量乘在一起来计算。最终将要使用的 TF-IDF 度量是 *tfidf* 矩阵的归一化版本，矩阵是 *tf* 和 *idf* 的乘积。将 *tfidf* 矩阵除以矩阵的 L2 范数来进行矩阵归一化，L2 范数也称为欧几里得范数，它是每个单词 *tfidf* 权重平方和的平方根。数学上，将最终的 *tfidf* 特征向量表示为 $tfidf = \frac{tfidf}{||\ tfidf\ ||}$，其中，$||\ tfidf\ ||$ 表示 *tfidf* 矩阵的欧几里得 L2 范数。

下面的代码片段是考虑我们已经有了前面章节的词袋特征向量的情况下，获得基于 *tfidf* 的特征向量的具体实现。

```
from sklearn.feature_extraction.text import TfidfTransformer

def tfidf_transformer(bow_matrix):

    transformer = TfidfTransformer(norm='l2',
                                   smooth_idf=True,
                                   use_idf=True)
    tfidf_matrix = transformer.fit_transform(bow_matrix)
    return transformer, tfidf_matrix
```

你可能看到，在参数中使用了 L2 范数选项，并且对一些单词可能存在 *idf* 为 0 的情况以增加权重的方式对 *idf* 进行平滑处理，而没有忽略它们。在下面的代码片段观察这个函数的执行情况。

```
import numpy as np
from feature_extractors import tfidf_transformer
feature_names = bow_vectorizer.get_feature_names()

# build tfidf transformer and show train corpus tfidf features
```

```
In [388]: tfidf_trans, tdidf_features = tfidf_transformer(bow_features)
     ...: features = np.round(tdidf_features.todense(), 2)
     ...: display_features(features, feature_names)
    and  beautiful  blue  cheese    is  love   sky    so   the
0  0.00       0.00  0.40    0.00  0.49  0.00  0.49  0.00  0.60
1  0.44       0.35  0.23    0.00  0.56  0.00  0.56  0.00  0.00
2  0.00       0.43  0.29    0.00  0.35  0.00  0.35  0.55  0.43
3  0.00       0.00  0.35    0.66  0.00  0.66  0.00  0.00  0.00

# show tfidf features for new_doc using built tfidf transformer
In [389]: nd_tfidf = tfidf_trans.transform(new_doc_features)
     ...: nd_features = np.round(nd_tfidf.todense(), 2)
     ...: display_features(nd_features, feature_names)
   and  beautiful  blue  cheese   is  love   sky   so  the
0  0.0       0.0  0.63     0.0  0.0   0.0  0.77  0.0  0.0
```

上面的输出显示了全部例子文档的 *tfidf* 特征向量。使用 TfidfTransformer 类有助于计算每个文档基于前面方程描述的 *tfidf* 值。

现在，我们将看看该类内部如何工作。你也会看到如何实现前面描述的数学方程以计算基于 *tfidf* 的特征向量。本节主要面向机器学习专家（和对这个场景背后如何工作感兴趣的读者）。我们将载入必要的依赖，并通过重用词袋模型特征计算样例语料的单词频率（*TF*），该词频也可作为训练语料 CORPUS 的词频。

```
import scipy.sparse as sp
from numpy.linalg import norm
feature_names = bow_vectorizer.get_feature_names()

# compute term frequency
tf = bow_features.todense()
tf = np.array(tf, dtype='float64')

# show term frequencies
In [391]: display_features(tf, feature_names)
   and  beautiful  blue  cheese   is  love  sky   so  the
0  0.0       0.0   1.0     0.0  1.0   0.0  1.0  0.0  1.0
1  1.0       1.0   1.0     0.0  2.0   0.0  2.0  0.0  0.0
2  0.0       1.0   1.0     0.0  1.0   0.0  1.0  1.0  1.0
3  0.0       0.0   1.0     1.0  0.0   1.0  0.0  0.0  0.0
```

我们将基于出现某单词的文档数量计算每个单词的文档频率（*DF*）。下面的代码片段显示如何从词袋模型特征矩阵获得 *DF*。

```
# build the document frequency matrix
df = np.diff(sp.csc_matrix(bow_features, copy=True).indptr)
df = 1 + df # to smoothen idf later

# show document frequencies
In [403]: display_features([df], feature_names)
   and  beautiful  blue  cheese  is  love  sky  so  the
0    2         3     5       2   4     2    4   2    3
```

上述输出向我们展示了每个单词的文档频率（*DF*），可以使用 CORPUS 中的文档来验证它。假设有一个文档，其中所有单词都出现一次，请记住，我们已经对每个频率值加 1 以平滑 *idf* 值，避免被 0 除的错误。因此，如果验证 CORPUS，我们将会看到 blue 出现 4（+1）次，sky 出现 3（+1）次等，考虑到我们使用了（+1）来进行平滑。

现在我们拥有了文档频率，就可以使用前面的公式计算逆文档频率（*idf*）。记住，对语

料库中文档的总数加 1，因为早先假设平滑 *idf* 包含所有单词至少 1 次。

```
# compute inverse document frequencies
total_docs = 1 + len(CORPUS)
idf = 1.0 + np.log(float(total_docs) / df)

# show inverse document frequencies
In [406]: display_features([np.round(idf, 2)], feature_names)
    and  beautiful  blue  cheese    is  love   sky    so   the
0  1.92       1.51   1.0    1.92  1.22  1.92  1.22  1.92  1.51

# compute idf diagonal matrix
total_features = bow_features.shape[1]
idf_diag = sp.spdiags(idf, diags=0, m=total_features, n=total_features)
idf = idf_diag.todense()

# print the idf diagonal matrix
In [407]: print np.round(idf, 2)
[[ 1.92  0.    0.    0.    0.    0.    0.    0.    0.  ]
 [ 0.    1.51  0.    0.    0.    0.    0.    0.    0.  ]
 [ 0.    0.    1.    0.    0.    0.    0.    0.    0.  ]
 [ 0.    0.    0.    1.92  0.    0.    0.    0.    0.  ]
 [ 0.    0.    0.    0.    1.22  0.    0.    0.    0.  ]
 [ 0.    0.    0.    0.    0.    1.92  0.    0.    0.  ]
 [ 0.    0.    0.    0.    0.    0.    1.22  0.    0.  ]
 [ 0.    0.    0.    0.    0.    0.    0.    1.92  0.  ]
 [ 0.    0.    0.    0.    0.    0.    0.    0.    1.51]]
```

你现在看到了 *idf* 矩阵，这个矩阵是基于数学公式计算得到的，我们把它转换为对角矩阵。当计算词频乘积时，这将非常有用。

既然有了 *tf* 和 *idf*，就可以使用矩阵乘积计算 *tfidf* 特征矩阵了，如以下代码所示：

```
# compute tfidf feature matrix
tfidf = tf * idf

# show tfidf feature matrix
In [410]: display_features(np.round(tfidf, 2), feature_names)
    and  beautiful  blue  cheese    is  love   sky    so   the
0  0.00       0.00   1.0    0.00  1.22  0.00  1.22  0.00  1.51
1  1.92       1.51   1.0    0.00  2.45  0.00  2.45  0.00  0.00
2  0.00       1.51   1.0    0.00  1.22  0.00  1.22  1.92  1.51
3  0.00       0.00   1.0    1.92  0.00  1.92  0.00  0.00  0.00
```

我们现在已经得到了 *tfidf* 特征矩阵，但要等一等，现在还没有结束。如果你还记得前面描述的方程，就会知道我们还需要把它除以 L2 范数。下面的代码片段计算每个文档的 *tfidf* 范数，使用这个范数除以 *tfidf* 权重得到我们最终想要的 *tfidf* 矩阵。

```
# compute L2 norms
norms = norm(tfidf, axis=1)

# print norms for each document
In [412]: print np.round(norms, 2)
[ 2.5   4.35  3.5   2.89]

# compute normalized tfidf
norm_tfidf = tfidf / norms[:, None]

# show final tfidf feature matrix
In [415]: display_features(np.round(norm_tfidf, 2), feature_names)
    and  beautiful  blue  cheese    is  love   sky    so   the
```

```
0  0.00       0.00  0.40    0.00  0.49  0.00  0.49  0.00  0.60
1  0.44       0.35  0.23    0.00  0.56  0.00  0.56  0.00  0.00
2  0.00       0.43  0.29    0.00  0.35  0.00  0.35  0.55  0.43
3  0.00       0.00  0.35    0.66  0.00  0.66  0.00  0.00  0.00
```

比较上面得到的 CORPUS 矩阵中文档的 *tfidf* 特征矩阵和前面使用 TfidfTransformer 得到的特征矩阵。注意，它们是相同的，因此验证了我们的数学实现是正确的——事实上，scikit-learn 的 TfidfTransformer 采用同样的数学实现，在后台进行了一些优化。现在，假设我们想计算新文档 new_doc 基于 *tfdif* 的特征矩阵，可以使用下面的代码片段计算它。在计算词频前，复用 new_doc_features 词袋向量：

```
# compute new doc term freqs from bow freqs
nd_tf = new_doc_features
nd_tf = np.array(nd_tf, dtype='float64')

# compute tfidf using idf matrix from train corpus
nd_tfidf = nd_tf*idf
nd_norms = norm(nd_tfidf, axis=1)
norm_nd_tfidf = nd_tfidf / nd_norms[:, None]

# show new_doc tfidf feature vector
In [418]: display_features(np.round(norm_nd_tfidf, 2), feature_names)
   and  beautiful  blue  cheese   is  love   sky   so  the
0  0.0        0.0  0.63     0.0  0.0   0.0  0.77  0.0  0.0
```

上面的输出描述了 new_doc 基于 *tfidf* 的特征向量，可以看到它与通过 TfidfTransformer 计算得到的相同。

掌握了 *tfidf* 计算内部的工作原理之后，我们将实现一个通用的函数，可以直接从原始文档中计算文档基于 *tfidf* 的特征向量。下面的代码描述了相同的过程。

```
from sklearn.feature_extraction.text import TfidfVectorizer

def tfidf_extractor(corpus, ngram_range=(1,1)):

    vectorizer = TfidfVectorizer(min_df=1,
                                 norm='l2',
                                 smooth_idf=True,
                                 use_idf=True,
                                 ngram_range=ngram_range)
    features = vectorizer.fit_transform(corpus)
    return vectorizer, features
```

上面的函数直接使用了 TfidfVectorizer，通过把原始文档作为输入，在内部计算词频和逆文档频率，直接计算 *tfidf* 向量，避免了使用 CountVectorizer 计算基于词袋模型的词频。它也支持将 n 元分词加入特征向量中。可以在下面的片段中看到函数的执行情况：

```
# build tfidf vectorizer and get training corpus feature vectors
In [425]: tfidf_vectorizer, tdidf_features = tfidf_extractor(CORPUS)
     ...: display_features(np.round(tdidf_features.todense(), 2), feature_
          names)
    and  beautiful  blue  cheese    is  love   sky    so   the
0  0.00       0.00  0.40    0.00  0.49  0.00  0.49  0.00  0.60
1  0.44       0.35  0.23    0.00  0.56  0.00  0.56  0.00  0.00
2  0.00       0.43  0.29    0.00  0.35  0.00  0.35  0.55  0.43
3  0.00       0.00  0.35    0.66  0.00  0.66  0.00  0.00  0.00
# get tfidf feature vector for the new document
In [426]: nd_tfidf = tfidf_vectorizer.transform(new_doc)
```

```
    ...: display_features(np.round(nd_tfidf.todense(), 2), feature_names)
   and  beautiful  blue  cheese   is  love   sky   so  the
0  0.0        0.0  0.63     0.0  0.0   0.0  0.77  0.0  0.0
```

你从上面的输出可以看到 *tfidf* 特征向量与前面得到的一致。到此就要结束使用 *tfidf* 进行特征提取的讨论了。现在，我们将要看一看高级词向量技术。

4.5.3 高级词向量模型

为从文档中提取特征，有各种各样的方法创建更高级的词向量模型。这里将讨论一些词向量模型，它们使用了主流的谷歌 word2vec 算法。word2vec 模型由谷歌公司 2013 年发布，它是一个基于神经网络的实现，使用 CBOW（Continuous Bag of Words）和 skip-gram 两种结构学习单词的分布式向量表示。word2vec 模型相对于其他神经网络实现运行速度更快，而且不需要手工标签来创建单词的意义标识。

可以在下面的网址查阅谷歌 word2vec 项目更加详细的信息 https://code.google.com/archive/p/word2vec/。如果感兴趣，你可以尝试自己做一些实现。

在代码实现中，将使用 gensim 库，该库是 word2vec 的 Python 实现，提供了几个高级的接口使得建模非常容易。基本思想是提供一些文档语料作为输入，会得到词向量特征作为输出。在模型内部，建立基于输入文档的词汇表，通过前面提到的各种技术学习单词的向量表示，一旦学习完成，就建立了一个可用于从文档中提取单词向量的模型。使用如平均值和 *tfidf* 加权等方法，可以使用词向量计算文档的平均向量。可以在 http://radimrehurek.com/gensim/models/word2vec.html 上获得有关 gensim 库接口更详细的信息。

在训练语料建立模型时，我们将主要关注下面的参数。

- size：该参数用于设定词向量的维度，可以是几十到几千。可以尝试不同的维度，以获得最好的效果。
- window：该参数用于设定语境或窗口尺寸，指定了训练时对算法来说可算做上下文的单词窗口长度。
- min_count：该参数指定单词表中单词在语料中出现的最小次数。这个参数有助于去除一些文档中出现次数较少的不重要的单词。
- sample：该参数用于对单词出现的频率进行下采样，其理想值在 0.01 到 0.0001 之间。

建立模型之后，将基于一些加权策略来定义和实现两种词向量与文档结合的技术。接下来将实现下面两个技术：

- 平均词向量。
- TF-IDF 加权词向量。

在进一步实现之前，使用训练语料建立 word2vec 模型，开始特征提取的过程。下面的代码显示了如何实现：

```
import gensim
import nltk

# tokenize corpora
TOKENIZED_CORPUS = [nltk.word_tokenize(sentence)
                    for sentence in CORPUS]
tokenized_new_doc = [nltk.word_tokenize(sentence)
```

```
                for sentence in new_doc]

# build the word2vec model on our training corpus
model = gensim.models.Word2Vec(TOKENIZED_CORPUS, size=10, window=10,
                               min_count=2, sample=1e-3)
```

如你所见，我们使用前面描述的参数建立了模型。你可以尝试调整这些参数，也可以查看文档其他参数，以改变结构类型、worker 数量等，训练不同的模型。现在我们有了自己的模型，可以开始实现特征提取技术。

1. 平均词向量

上面的模型为单词表中每个单词创建一个向量表示，我们可以输入下面的代码来查看它们。

```
In [430]: print model['sky']
[ 0.01608407 -0.04819566  0.04227461 -0.03011346  0.0254148   0.01728328
  0.0155535   0.00774884 -0.02752112  0.01646519]

In [431]: print model['blue']
[-0.0472235   0.01662185 -0.01221706 -0.04724348 -0.04384995  0.00193343
 -0.03163504 -0.03423524  0.02661656  0.03033725]
```

基于前面指定的参数尺寸，每个单词向量的长度为 10。但是当我们处理语句和文档时，必须执行一些合并和聚合操作，以确保无论文本长度、单词数量等情况如何，最后特征的维度是相同的。在这个技术中，我们使用平均加权词向量，对于每个文档我们将提取文档中所有单词，获得单词表中每个单词的词向量。我们将全部词向量加在一起，除以单词表中匹配单词的总数，最后得到文档的平均词向量表示结果。上述描述可以使用下面的数学公式表示

$$AVW(D) = \frac{\sum_{1}^{n} wv(w)}{n}$$

其中 $AVW(D)$ 表示文档 D 的平均词向量，文档 D 中包括单词 w_1，w_2，…，w_n，$wv(w)$ 表示的是单词 w 的词向量。

该算法的伪代码描述如下：

```
model := the word2vec model we built
vocabulary := unique_words(model)
document := [words]
matched_word_count := 0
vector := []

for word in words:
        if word in vocabulary:
                vector := vector + model[word]
                matched_word_count :=  matched_word_count + 1

averaged_word_vector := vector / matched_word_count
```

这个伪代码以一种较好的、易于理解的方式显示了操作流程。现在，我们要使用下面的代码在 Python 中实现这个算法。

```
import numpy as np

# define function to average word vectors for a text document
def average_word_vectors(words, model, vocabulary, num_features):
```

```
    feature_vector = np.zeros((num_features,),dtype="float64")
    nwords = 0.

    for word in words:
        if word in vocabulary:
            nwords = nwords + 1.
            feature_vector = np.add(feature_vector, model[word])
    if nwords:
        feature_vector = np.divide(feature_vector, nwords)

    return feature_vector

# generalize above function for a corpus of documents
def averaged_word_vectorizer(corpus, model, num_features):
    vocabulary = set(model.index2word)
    features = [average_word_vectors(tokenized_sentence, model, vocabulary,
    num_features)
                for tokenized_sentence in corpus]
    return np.array(features)
```

你一定对 `average_word_vectors()`函数十分熟悉，它是我们前面使用伪代码表示的算法的完整实现。我们也创建了一个通用的函数 `averaged_word_vectorizer()`来实现语料库中多个文档平均词向量的计算。下面代码显示了我们这个函数在示例语料库上的执行情况。

```
# get averaged word vectors for our training CORPUS
In [445]: avg_word_vec_features = averaged_word_vectorizer(corpus=TOKENIZED_
CORPUS,
     ...:                                                  model=model,
     ...:                                                  num_features=10)
     ...: print np.round(avg_word_vec_features, 3)
[[ 0.006 -0.01   0.015 -0.014  0.004 -0.006 -0.024 -0.007 -0.001  0.   ]
 [-0.008 -0.01   0.021 -0.019 -0.002 -0.002 -0.011  0.002  0.003 -0.001]
 [-0.003 -0.007  0.008 -0.02  -0.001 -0.004 -0.014 -0.015  0.002 -0.01 ]
 [-0.047  0.017 -0.012 -0.047 -0.044  0.002 -0.032 -0.034  0.027  0.03 ]]

# get averaged word vectors for our test new_doc
In [447]: nd_avg_word_vec_features = averaged_word_
vectorizer(corpus=tokenized_new_doc,
     ...:                                                    model=model,
     ...:                                                    num_
                                                             features=10)
     ...: print np.round(nd_avg_word_vec_features, 3)
[[-0.016 -0.016  0.015 -0.039 -0.009  0.01  -0.008 -0.013  0.     0.023]]
```

你可以从上面的输出看到，我们已经获得了语料库中每个文档的维度尺寸统一的平均词向量，这些词向量将在后面送入机器学习算法进行分类。

2. TF-IDF 加权平均词向量

前面的向量生成器基于模型单词表中的单词，简单地对任何文档中相关的词向量进行求和，通过除以匹配的单词的数量计算一个简单的平均值。本节介绍一项创新的加权技术，使用单词的 TF-IDF 评分对每个匹配的词向量进行加权，对它们进行求和并除以文档中匹配的单词数量。我们将得到每个文档的一个 TF-IDF 加权平均词向量。

上述描述可以使用下面的数学公式表示

$$TWA(D) = \frac{\sum_{1}^{n} wv(w) \times tfidf(w)}{n}$$

其中 $TWA(D)$ 表示的是 TF-IDF 文档 D 加权平均词向量，文档 D 中包括单词 w_1，w_2，…，w_n，$wv(w)$ 表示的是单词 w 的词向量，$tfidf(w)$ 是单词 w 的 TF-IDF 权重。下面给出该算法的伪代码：

```
model := the word2vec model we built
vocabulary := unique_words(model)
document := [words]
tfidfs := [tfidf(word) for each word in words]
matched_word_wts := 0
vector := []

for word in words:
        if word in vocabulary:
                word_vector := model[word]
                weighted_word_vector := tfidfs[word] x word_vector
                vector := vector + weighted_word_vector
                matched_word_wts :=  matched_word_wts + tfidfs[word]

tfidf_wtd_avgd_word_vector := vector / matched_word_wts
```

这段伪代码给出了算法的结构，显示了如何实现前面定义算法的数学公式。下面的代码段在 Python 中实现了这个算法，因此我们可以使用它提取特征。

```
# define function to compute tfidf weighted averaged word vector for a document
def tfidf_wtd_avg_word_vectors(words, tfidf_vector, tfidf_vocabulary, model,
num_features):

    word_tfidfs = [tfidf_vector[0, tfidf_vocabulary.get(word)]
                   if tfidf_vocabulary.get(word)
                   else 0 for word in words]
    word_tfidf_map = {word:tfidf_val for word, tfidf_val in zip(words, word_
    tfidfs)}

    feature_vector = np.zeros((num_features,),dtype="float64")
    vocabulary = set(model.index2word)
    wts = 0.
    for word in words:
        if word in vocabulary:
            word_vector = model[word]
            weighted_word_vector = word_tfidf_map[word] * word_vector
            wts = wts + word_tfidf_map[word]
            feature_vector = np.add(feature_vector, weighted_word_vector)
    if wts:
        feature_vector = np.divide(feature_vector, wts)

    return feature_vector

# generalize above function for a corpus of documents
def tfidf_weighted_averaged_word_vectorizer(corpus, tfidf_vectors,
                                    tfidf_vocabulary, model, num_features):

    docs_tfidfs = [(doc, doc_tfidf)
                   for doc, doc_tfidf
                   in zip(corpus, tfidf_vectors)]
    features = [tfidf_wtd_avg_word_vectors(tokenized_sentence, tfidf, tfidf_
    vocabulary,
                                    model, num_features)
                for tokenized_sentence, tfidf in docs_tfidfs]
    return np.array(features)
```

tfidf_wtd_avg_word_vectors()函数帮助我们获得每个文档的 TF-IDF 加权平均词向量。我们也创建一个函数 tfidf_weighted_averaged_word_vectorizer()实现语料库中多个文档 TF-IDF 加权平均词向量的计算。使用下面代码看看我们实现的这个函数在示例语料库上的执行情况：

```
# get tfidf weights and vocabulary from earlier results and compute result
In [453]: corpus_tfidf = tdidf_features
     ...: vocab = tfidf_vectorizer.vocabulary_
     ...: wt_tfidf_word_vec_features = tfidf_weighted_averaged_word_
vectorizer(corpus=TOKENIZED_CORPUS, tfidf_vectors=corpus_tfidf,
     ...:                                tfidf_vocabulary=vocab, model=model,
                                         num_features=10)
     ...: print np.round(wt_tfidf_word_vec_features, 3)
[[ 0.011 -0.011  0.014 -0.011  0.007 -0.007 -0.024 -0.008 -0.004 -0.004]
 [ 0.    -0.014  0.028 -0.014  0.004 -0.003 -0.012  0.011 -0.001 -0.002]
 [-0.001 -0.008  0.007 -0.019  0.001 -0.004 -0.012 -0.018  0.001 -0.014]
 [-0.047  0.017 -0.012 -0.047 -0.044  0.002 -0.032 -0.034  0.027  0.03 ]]

# compute avgd word vector for test new_doc
In [454]: nd_wt_tfidf_word_vec_features = tfidf_weighted_averaged_word_
vectorizer(corpus=tokenized_new_doc, tfidf_vectors=nd_tfidf, tfidf_
vocabulary=vocab, model=model, num_features=10)
     ...: print np.round(nd_wt_tfidf_word_vec_features, 3)
[[-0.012 -0.019  0.018 -0.038 -0.006  0.01  -0.006 -0.011 -0.003  0.023]]
```

从上面的结果，你看到了我们如何将每个文档转换为 TF-IDF 加权平均词向量。在我们实现基于 TF-IDF 的文档特征提取时，也使用到了我们之前获得的 TF-IDF 权重和单词表。

现在，你已经很好地掌握了如何从文档数据中提取用于训练分类器的特征。

4.6 分类算法

分类算法是有监督的机器学习算法，基于其对过去的观测情况，对数据点进行分类、归类或加标签。作为一个有监督的机器学习算法，每个分类算法都需要训练数据。训练数据由一组训练值组成，其中每个观测值是一对输入数据点（通常是特征向量，如我们前面看到的）和输入观测值对应的输出结果。分类算法整体有三个过程：

- 训练是有监督学习算法分析和尝试推理训练数据的模式，让算法可以识别出产生具体输出模式的过程。这些输出一般是类标签/类变化/响应变量。训练之前，我们通常执行特征提取过程从原始数据中提取出有意义的特征。这些特征集被送入我们所选择的算法，这个算法尽力从这些数据和对应的输出中识别和学习模式。算法的结果是一个推断函数，称为模型或分类模型。期望该模型从训练集的学习模式中得到足够的推广能力，使之能够预测未来新数据点的类别或结果。
- 评估包括测试模型的预测性能，检验它在训练数据集上训练和学习效果如何。为此，我们通常使用验证数据集，通过预测该数据集，并对比预测结果与真实类标签的差异来测试模型的性能，真实类标签也称为真实值（ground truth）。通常，我们还会使用交叉验证，其中把数据划分为 2 份，其中一份用于训练，剩下的用于验证被训练的模型。注意，我们还会根据验证结果对模型进行调整，以获得最佳的配置，从而产生最高的精度和最小的误差。我们也会使用保留数据集或测试数据集评估我们的

模型，但我们不会使用这个数据集调整模型，因为这将导致模型有偏差或者把数据过拟合到非常具体的特征。保留数据集或测试数据集是新的、有代表性的样本点，真实的数据样本可以看到模型产生预测的情况，以及模型在这些新的数据样本上的效果如何。稍后，我们将研究用来评估和度量模型性能的各种常用的度量标准。

- 调优，也称为超参数调优或优化，我们致力于优化模型以最大限度地发挥其预测能力并减少错误。每个模型的核心是一个带着几个参数的数学公式，这些参数决定了模型的复杂度、学习能力等。这些参数也称为超参数，因为它们不能从数据中直接学习，必须在运行前设置和训练这个模型。因此，选择一组可选的模型超参数使得模型产生良好预测准确性的过程称为模型调优，我们可以使用多种方法进行模型调优，包括随机搜索和网格搜索。因为参数调优更倾向于机器学习核心内容，目前超出了本书的范围，并且我们的模型使用默认超参数配置就能很好工作，因此我们在实现中不再涉及这些内容。如果你对模型调优和优化感兴趣，互联网上可以找到很多资源。

虽然有各种类型的分类算法，但我们不会深入到每一个细节。我们的重点仍然是文本分类，我不想让每个人都过多地对算法进行数学推导。不过，我会介绍一些对文本分类非常有效的算法，尽量对它们进行一些说明，并保留一些核心的数学公式。这些算法如下：

- 多项式朴素贝叶斯（Multinomial Naïve Bayes）。
- 支持向量机（Support Vector Machines）。

此外你还可以查阅一些其他的算法，包括逻辑回归、决策树和神经网络。集成技术使用一组或集成多个模型学习和预测输出，包括随机森林和梯度提升，但这些算法因为容易过拟合，所以在文本分类上表现不是很好。我建议你在实践这些方法之前认真思考。除此之外，最近基于深度学习的技术变得十分活跃。它们使用多个隐层，结合一些神经网络模型建立复杂的分类模型。

在使用多项式朴素贝叶斯和支持向量机解决分类问题之前，我们将简要介绍一些与它们相关的概念。

4.6.1 多项式朴素贝叶斯

该算法是主流的朴素贝叶斯算法的一个特例，用于超过两类的预测和分类任务。在学习多项式朴素贝叶斯算法之前，让我们先看看朴素贝叶斯算法的定义和公式。朴素贝叶斯算法是一个将贝叶斯定理应用于实践的有监督学习算法。然而，这里有一个“朴素”的假设，也就是每个特征与其他特征之间相互独立。数学上，我们可以公式化表示如下：假设类型变量 y 和一组由 n 个特征组成的特征向量 $\{x_1, x_2, \cdots, x_n\}$，使用贝叶斯定理，我们可以在给定这些特征的情况下，把 y 发生的概率表示为

$$P(y|x_1,x_2,\cdots,x_n) = \frac{P(y) \times P(x_1,x_2,\cdots,x_n|y)}{P(x_1,x_2,\cdots,x_n)}$$

在假设 $P(x_i|y,x_1,x_2,\cdots,x_{i-1},x_{i+1},\cdots,x_n) = P(x_i|y)$ 情况下，对于所有的 i 我们可以表示为

$$P(y|x_1,x_2,\cdots,x_n) = \frac{P(y) \times \prod_{i=1}^{n} P(x_i|y)}{P(x_1,x_2,\cdots,x_n)}$$

其中，i 的取值范围从 1 到 n。为了简化，这个公式可以写作后验概率 = $\frac{\text{步验} \times \text{似然值}}{\text{证据}}$，因为 $P(x_1, x_2, \cdots, x_n)$ 是常量，模型可以表示为如下形式

$$P(y|x_1, x_2, \cdots, x_n) \propto P(y) \times \prod_{i=1}^{n} P(x_i|y)$$

这意味着在前面特征独立性假设下，每个特征与其他特征条件独立，将被预测的类型变量的条件分布 y 可用下面的数学方程表示

$$P(y|x_1, x_2, \cdots, x_n) = \frac{1}{Z} P(y) \times \prod_{i=1}^{n} P(x_i|y)$$

其中证据度量 $Z = p(x)$ 是依赖于特征变量的常数比例因子。从这个方程，我们通过结合 MAP（Maximum A Posteriori）决策规则就可以建立贝叶斯分类器，MAP 代表最大后验概率。在当前的范围内，深入统计细节是不现实的。但通过使用它，分类器可以表示为一个数学函数，该函数赋予一个预测类标签 $\hat{y} = C_k$，对于 k 使用下面的表示：

$$\hat{y} = \underset{k \in \{1,2,\cdots,K\}}{\operatorname{argmax}} P(C_k) \times \prod_{i=1}^{n} P(x_i|C_k)$$

一般认为这个分类器是简单的，很明显因为它的名字，也因为我们对数据和特征的几个假设，在现实世界中数据和特征其实并不独立。然而，该算法在许多分类相关的用例中仍然十分有效地工作，包括多类文档分类、垃圾邮件过滤等。相对于其他分类器，该分类器训练速度快，当我们训练数据不充足时也能很好地工作。当有很多特征时，模型往往表现不好，这种现象称为维数灾难。朴素贝叶斯通过解耦类变量 - 相关条件的特征分布的方法解决这个问题，从而使各分布作为单一维度的分布独立估计。

多项式朴素贝叶斯是上面预测和分类数据算法的一个扩展，这里分类类型或输出的数量大于 2。本例中，通常假设特征向量为词袋模型的单词数量，但是基于 TF-IDF 的加权特征也可以工作。存在一个局限就是负的加权特征不能送入这个算法。对于每一个类标签 y，其分布可以表示为 $p_y = \{p_{y1}, p_{y2}, \cdots, p_{yn}\}$，特征总数量是 n，这可以表示文本分析中词汇表中的不同单词。根据上面的公式，$p_{yi} = P(x_i|y)$ 表示输出结果或类型为 y 的任何观测例子特征 i 的概率。可以使用平滑版的最大似然估计方法（使用相对出现频率）估计参数 p_y，可表示如下

$$\hat{p}_{yi} = \frac{F_{yi} + \alpha}{F_y + \alpha n}$$

其中 $F_{yi} = \sum_{x \in TD} x_i$ 是训练数据集 TD 中类标签 y 例子中特征 i 出现的频率，$F_y = \sum_{i=1}^{|TD|} F_{yi}$ 是类标签 y 所有特征的总频率。对于在学习数据中未出现的特征的计数，这里有一些利用先验值 $\alpha \geqslant 0$ 平滑方法帮助消除零概率相关的问题。通常该参数也有一些常用的特定的设置。$\alpha = 1$ 称为 Laplace 平滑，$\alpha < 1$ 称为 Lidstone 平滑。`scikit-learn` 函数库的 `MultinomialNB` 类中提供了一个多项式朴素贝叶斯的优秀实现，我们将在后面建立文本分类器时使用。

4.6.2 支持向量机

在机器学习中，支持向量机（SVM）可以用于分类、回归、异常值或离群点检测。考虑到二元分类问题，如果我们有训练数据，使每个数据点或观察属于一个特定的类，SVM 算法可以基于此数据进行训练，使得它可以把未来的数据点分配到两个类之一。该

算法将在空间中表示训练数据样本的点，这样的点属于哪两个类之一可以由它们之间的一个宽的间隔分隔，称为*超平面分离*。被预测的新的数据点属于哪个类取决于新数据点落到超平面哪一侧。这个过程是一个典型的线性分类过程。然而，通过一个有趣的称为*内核技巧*的方法，SVM 也可以执行非线性分类。通常，在特征空间中的数据点之间的内积有助于实现这一点。

SVM 算法输入一组数据点，为高维特征空间建立一个或一组超平面。超平面的间隔越大，分隔效果就越好，分类器就可以得到更低的推广误差。让我们正式地使用数学表示这些。假设训练数据集包括 n 个数据点 $(\vec{x}_1, y_1), \cdots, (\vec{x}_n, y_n)$，类变量 $y_i \in \{-1, 1\}$ 表示数据点 $\vec{x}_i$ 对应的类。每个数据点 $\vec{x}_i$ 是一个特征向量。SVM 算法的目标是找到能把类标签 $y_i = 1$ 的数据点集合和 $y_i = -1$ 数据点集合最大间隔的超平面，使得超平面和与超平面最近的样本数据点的距离最大化。这些样本数据点称为支持向量。图 4-3 来自维基百科，显示了带有超平面的向量空间的情况。

你在图中可以清楚地看到超平面和支持向量的情况。超平面定义为满足 $\vec{\omega} \cdot \vec{x} + b = 0$ 的点 $\vec{x}$ 的集合，这里 $\vec{\omega}$ 是到超平面的法线向量，如图 4-3 所示，$\frac{b}{||\vec{\omega}||}$ 是从原点到图中突出显示的支持向量的超平面偏移量。将数据点分隔为所属的不同类型，这里有两种主要的间隔类型。

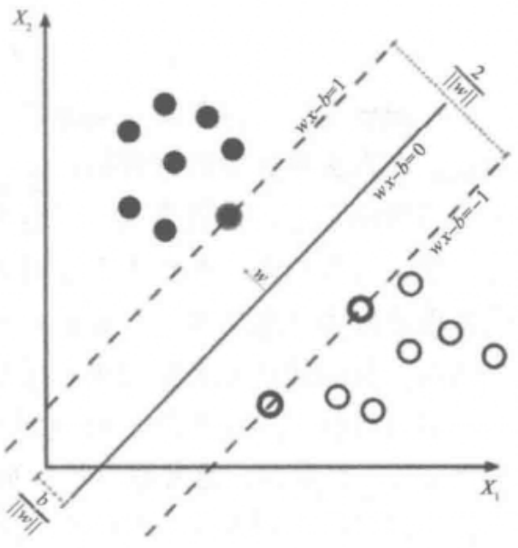


图 4-3 两类分类 SVM 超平面和支持向量（源自：Wikipedia）

如图 4-3 所示，当数据是线性可分的时候，我们有两个平行超平面表示的一个硬间隔边界，由图中虚线表示，这有助于将属于不同类型的数据点分开。分隔数据点时，要考虑使它们之间的距离尽可能大。由两个超平面作为边界的区域形成了位于中间的最大间隔超平面。图中显示的超平面方程为 $\vec{\omega} \cdot \vec{x} + b = 1$ 和 $\vec{\omega} \cdot \vec{x} + b = -1$。

通常，数据点是线性不可分的，对此我们可以使用 `hinge` 损失函数，该函数可以表示为 $max\ (0,\ 1 - y_i\ (\vec{\omega} \cdot \vec{x} + b))$，实际上可以在 `SVC`、`LinearSVC`、或 `SGDClassifier` 类中找到 SVM 的 `scikit-learn` 实现，这里我们使用了前面定义的 `hinge` 损失函数（默认设置）来优化和建立模型。这个损失函数帮助我们获得软间隔，称为*软间隔 SVM*。

对于多类分类问题，如果我们有 n 个类，可以把每个类作为二分类进行训练，帮助实现该类与其他 $n-1$ 个类的分离。预测过程就是计算每个分类器的得分（到超平面的距离），将最大得分的分类器选为类标签的过程。在 SVM 算法中，随机梯度下降法也用于最小化损失函数。图 4-4 显示了如何训练 `iris` 数据集上的三类分类问题的三个分类器。该图使用 `scikit-learn` 模块生成，可以从 http://scikit-learn.org 的官方文档中获得。

从图 4-4 中，你可以清楚地看到对于 3 个类，共计训练了 3 个 SVM 分类器，一起组成了最终的预测结果，以便准确地计算数据点的所属类别。有很多详尽介绍有监督机器学习和分类的资源和书籍。感兴趣的读者可以进一步去查阅，更加深入地了解这些技术如何工作以及如何应用到分析各类问题上。

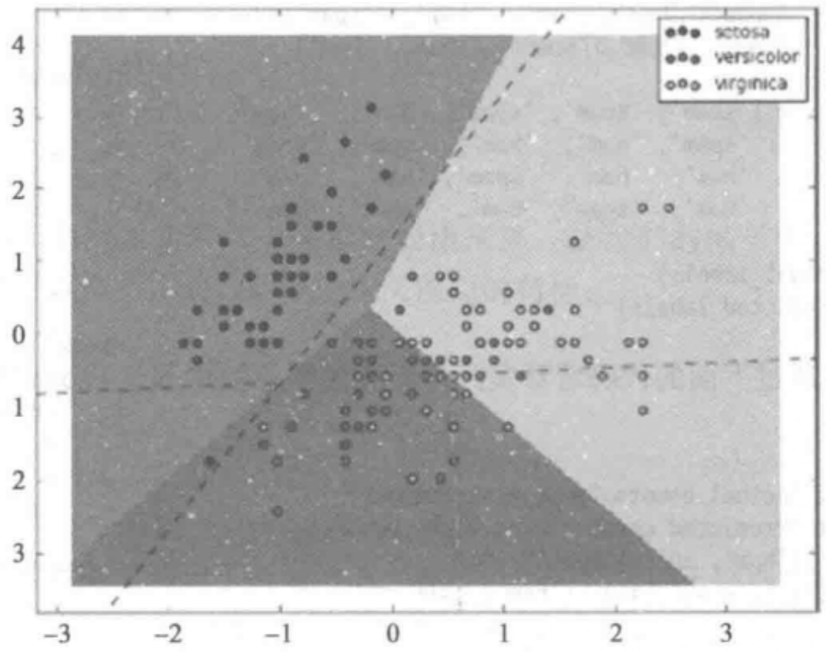


图 4-4 用于三类分类的多类 SVM 分类器（源自：scikit-learn. org）

4.7 评估分类模型

训练、调优和建立模型是整个分析生命周期的重要部分，但更重要的是知道这些模型的性能如何。分类模型的性能一般基于模型对新数据的预测输出结果。通常情况下，使用测试数据集或保留数据集对性能进行测量，数据集中的数据不会以任何方式影响或训练分类器。测试数据集一般包括一些观测值和对应的标签。

我们使用与训练模型时一样的方法提取特征。这些特征被送入训练好的模型，我们就获得了每个数据的预测结果。接着，预测结果与实际标签进行匹配以观察模型预测结果如何或其准确情况。

有多个指标可以判定模型的预测性能，但我们将主要关注下面几个指标：

- 准确率（accuracy）。
- 精确率（precision）。
- 召回率（recall）。
- F1 score。

让我们看一下实际的例子，看看如何计算这些指标。考虑一个将邮件分类为'Spam'（垃圾邮件）或'ham'（正常邮件）之一的二分类问题。假设我们共有 20 封邮件，对于每封邮件我们已经有了实际的手工标签，我们将这些邮件送入已经建好的分类器，就得到了每封邮件的预测标签。现在，我们想通过比较每个预测值与实际标签来衡量分类器的性能。下面的代码设置初始化依赖项、实际标签与预测的标签。

```
from sklearn import metrics
import numpy as np
import pandas as pd
from collections import Counter

actual_labels = ['spam', 'ham', 'spam', 'spam', 'spam',
                 'ham',  'ham', 'spam', 'ham',  'spam',
```

```
                'spam', 'ham', 'ham',  'ham',  'spam',
                'ham',  'ham', 'spam', 'spam', 'ham']

predicted_labels = ['spam', 'spam', 'spam', 'ham',  'spam',
                    'spam', 'ham',  'ham',  'spam', 'spam',
                    'ham',  'ham',  'spam', 'ham',  'ham',
                    'ham',  'spam', 'ham',  'spam', 'spam']

ac = Counter(actual_labels)
pc = Counter(predicted_labels)
```

现在，让我们通过下面的代码看看实际标签和预测标签中属于'spam'或'ham'类的邮件总数。

```
In [517]: print 'Actual counts:', ac.most_common()
     ...: print 'Predicted counts:', pc.most_common()
Actual counts: [('ham', 10), ('spam', 10)]
Predicted counts: [('spam', 11), ('ham', 9)]
```

因此，我们看到共有10封电子邮件是'spam'和10封电子邮件是'ham'。我们的分类器预测共有11封电子邮件是'spam'和9个邮件是'ham'。现在如何比较哪些电子邮件是真正的'spam'，它的归类是什么？混淆矩阵是衡量两个类分类性能的很好的方法。*混淆矩阵*是一个表格结构，它有助于可视化分类器的性能。矩阵中的每一列代表预测的分类实例，矩阵的每一行代表实际类标签的分类实例（如果需要的话，也可将其反转）。我们通常定义一个类标签为*正类*，这是我们感兴趣的类。图4-5给出了一个典型的两类分类混淆矩阵，其中p代表正类，n代表负类。

	P' （预测）	n' （预测）
P （实际）	真阳性	假阳性
n （实际）	假阳性	真阳性

图4-5 两类分类问题的混淆矩阵

在图4-5中，你可以看到矩阵中的一些术语。真阳性（TP）表示正确的命中数或预测为正类。假阴性（FN）表示我们错误地将这个类的实例预测为错误的负类。假阳性（FP）是我们错误地预测为正类时，实际上它不是。真阴性（TN）是我们正确预测为负类的实例数。

下面的代码为我们的数据建立一个混淆矩阵：

```
In [519]: cm = metrics.confusion_matrix(y_true=actual_labels,
     ...:                               y_pred=predicted_labels,
     ...:                               labels=['spam','ham'])
     ...: print pd.DataFrame(data=cm,
     ...:                    columns=pd.MultiIndex(levels=[['Predicted:'],
     ...:                                                  ['spam','ham']],
     ...:                                          labels=[[0,0],[0,1]]),
     ...:                    index=pd.MultiIndex(levels=[['Actual:'],
```

```
    ...:                                                ['spam','ham']],
    ...:                                           labels=[[0,0],[0,1]]))
           Predicted:
                spam ham
Actual: spam       5   5
        ham        6   4
```

我们现在得到一个与上面图片中类似的混淆矩阵。在我们的例子中，我们假设'spam'是正类。现在我们可以在下面的代码段中定义前面的指标：

```
positive_class = 'spam'

true_positive = 5.
false_positive = 6.
false_negative = 5.
true_negative = 4.
```

既然我们从混淆矩阵中得到了必要的值，就可以逐一计算四个性能指标。为了有助于除法计算，我们把这些值作为浮点数。我们将使用 scikit-learn 的指标（metriss）模块，该模块非常强大，有助于在一个函数内计算这些指标。我们将手动定义和计算这些指标，以便你可以清楚地了解它们，并从 metriss 模块中看到这些函数的内部执行情况。

准确率定义为模型整体的准确性或正确预测的比例，可由下面公式表示

$$\text{准确率} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{FN} + \text{TN}}$$

这里我们在分子和分母中使用正确预测数量除以全部输出数量。下面的代码给出了准确率的计算：

```
In [522]: accuracy = np.round(
    ...:                 metrics.accuracy_score(y_true=actual_labels,
    ...:                                        y_pred=predicted_labels),2)
    ...: accuracy_manual = np.round(
    ...:                     (true_positive + true_negative) /
    ...:                       (true_positive + true_negative +
    ...:                        false_negative + false_positive),2)
    ...: print 'Accuracy:', accuracy
    ...: print 'Manually computed accuracy:', accuracy_manual
Accuracy: 0.45
Manually computed accuracy: 0.45
```

精确率定义为正类准确预测的数量与正类相关所有预测的比值，可用公式描述为

$$\text{精确率} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

这里使用正类正确预测的数量除以正类的所有预测数量，包括预测为假阳性的数量。下面的代码给出了精确率的计算：

```
In [523]: precision = np.round(
    ...:                 metrics.precision_score(y_true=actual_labels,
    ...:                                         y_pred=predicted_labels,
    ...:                                         pos_label=positive_
                                                 class),2)
    ...: precision_manual = np.round(
    ...:                         (true_positive) /
    ...:                         (true_positive + false_positive),2)
    ...: print 'Precision:', precision
    ...: print 'Manually computed precision:', precision_manual
Precision: 0.45
Manually computed precision: 0.45
```

召回率的定义是正类中被正确预测的实例数量，也称之为命中率、覆盖率或灵敏度，可用公式描述为

$$召回率 = \frac{TP}{TP + FN}$$

其中，我们使用正类中正确预测的数量除以正确与未正确预测的数量之和，得到命中率。下面的代码给出了召回率的计算：

```
In [524]: recall = np.round(
     ...:             metrics.recall_score(y_true=actual_labels,
     ...:                                  y_pred=predicted_labels,
     ...:                                  pos_label=positive_class),2)
     ...: recall_manual = np.round(
     ...:                     (true_positive) /
     ...:                     (true_positive + false_negative),2)
     ...: print 'Recall:', recall
     ...: print 'Manually computed recall:', recall_manual
Recall: 0.5
Manually computed recall: 0.5
```

F1 score 是另一个准确性指标，通过计算精确率和召回率的调和平均值得到，其用公式表示如下：

$$F1\ score = \frac{2 \times 精确率 \times 召回率}{精确率 + 召回率}$$

我们可以使用下面的代码计算一样的 F1 score：

```
In [526]: f1_score = np.round(
     ...:                 metrics.f1_score(y_true=actual_labels,
     ...:                                  y_pred=predicted_labels,
     ...:                                  pos_label=positive_class),2)
     ...: f1_score_manual = np.round(
     ...:                     (2 * precision * recall) /
     ...:                     (precision + recall),2)
     ...: print 'F1 score:', f1_score
     ...: print 'Manually computed F1 score:', f1_score_manual
F1 score: 0.48
Manually computed F1 score: 0.47
```

这里给出了几个评估分类模型最常用的主要指标，我们将用来评估模型的性能的正是以上指标，而且你应该还记得在第 3 章建立标签器和分析器时见过这几个指标。

4.8 建立一个多类分类系统

从规范化到特征提取、建模和评估，我们已经完成了建立分类系统的全部必要的步骤。本节将所有的东西组装在一起，应用到真实数据上以建立一个多类文本分类系统。对于此项工作，我们将使用 scikit-learn 下载的 20 个新闻组数据集。这 20 个新组闻数据集包括分散在 20 个不同类别或主题的 18 000 个新闻组帖子，这就构建了 20 类分类问题！请记住类的数量越多，尝试建立正确分类器就越复杂或者越困难。为防止模型因为文件头或者邮件地址而过拟合或泛化能力不强，推荐的做法是从文档中去除文件头、文件尾和引用，因此需要确保我们考虑到了这一点。对于去除上述三项内容后的空文档或没有内容的文档，我们也将予以剔除，因为尝试从空文档中提取特征是毫无意义的。

让我们开始下载所需的数据集以及为建立训练和测试数据集所用的函数：

```
from sklearn.datasets import fetch_20newsgroups
from sklearn.cross_validation import train_test_split
def get_data():
    data = fetch_20newsgroups(subset='all',
                              shuffle=True,
                              remove=('headers', 'footers', 'quotes'))
    return data

def prepare_datasets(corpus, labels, test_data_proportion=0.3):
    train_X, test_X, train_Y, test_Y = train_test_split(corpus, labels,
                                                        test_size=0.33,
random_state=42)
    return train_X, test_X, train_Y, test_Y

def remove_empty_docs(corpus, labels):
    filtered_corpus = []
    filtered_labels = []
    for doc, label in zip(corpus, labels):
        if doc.strip():
            filtered_corpus.append(doc)
            filtered_labels.append(label)

    return filtered_corpus, filtered_labels
```

现在我们已经获得了数据，查看了数据集中分类的数量，使用下面的代码将数据集分为测试数据集和训练数据集（如果你还没有下载数据集，请连接互联网，花费一些时间下载全部语料）：

```
# get the data
In [529]: dataset = get_data()

# print all the classes
In [530]: print dataset.target_names
['alt.atheism', 'comp.graphics', 'comp.os.ms-windows.misc', 'comp.sys.ibm.
pc.hardware', 'comp.sys.mac.hardware', 'comp.windows.x', 'misc.forsale',
'rec.autos', 'rec.motorcycles', 'rec.sport.baseball', 'rec.sport.hockey',
'sci.crypt', 'sci.electronics', 'sci.med', 'sci.space', 'soc.religion.
christian', 'talk.politics.guns', 'talk.politics.mideast', 'talk.politics.
misc', 'talk.religion.misc']

# get corpus of documents and their corresponding labels
In [531]: corpus, labels = dataset.data, dataset.target
     ...: corpus, labels = remove_empty_docs(corpus, labels)

# see sample document and its label index, name
In [548]: print 'Sample document:', corpus[10]
     ...: print 'Class label:',labels[10]
     ...: print 'Actual class label:', dataset.target_names[labels[10]]
Sample document: the blood of the lamb.
This will be a hard task, because most cultures used most animals
for blood sacrifices. It has to be something related to our current
post-modernism state. Hmm, what about used computers?

Cheers,
Kent
Class label: 19
Actual class label: talk.religion.misc

# prepare train and test datasets
```

```
In [549]: train_corpus, test_corpus, train_labels, test_labels = prepare_
datasets(corpus,
     ...:                                                    labels, test_
data_proportion=0.3)
```

从上面的代码你可以看到文档和标签的情况。每个文档拥有自己的类标签，这些标签是需要进行分类的20个主题之一。这些标签是数字形式的，如果需要，我们可以使用上面的代码容易地将它们映射回原来的类别名字。我们已经把数据分为训练数据集和测试数据集，测试数据集占总数据的30%。我们将使用训练数据集建立模型，使用测试数据集测试模型的性能。下面的代码将使用前面建立的规范化模块对数据集进行规范化处理：

```
from normalization import normalize_corpus

norm_train_corpus = normalize_corpus(train_corpus)
norm_test_corpus = normalize_corpus(test_corpus)
```

记住，对语料库中每个文档进行规范化处理需要很多步骤（我们在前面已经介绍了这些步骤的实现方法），所以这将会耗费一些时间才能完成。完成文档规范化处理后，我们将使用前面建立的特征提取模块从文档中提取特征。我们将分别建立词袋模型、TF-IDF模型、平均词向量模型和TF-IDF加权平均词向量模型，并比较它们的性能。

下面的代码基于不同技术提取必要的特征：

```
from feature_extractors import bow_extractor, tfidf_extractor
from feature_extractors import averaged_word_vectorizer
from feature_extractors import tfidf_weighted_averaged_word_vectorizer
import nltk
import gensim

# bag of words features
bow_vectorizer, bow_train_features = bow_extractor(norm_train_corpus)
bow_test_features = bow_vectorizer.transform(norm_test_corpus)
# tfidf features
tfidf_vectorizer, tfidf_train_features = tfidf_extractor(norm_train_corpus)
tfidf_test_features = tfidf_vectorizer.transform(norm_test_corpus)

# tokenize documents
tokenized_train = [nltk.word_tokenize(text)
                   for text in norm_train_corpus]
tokenized_test = [nltk.word_tokenize(text)
                  for text in norm_test_corpus]
# build word2vec model
model = gensim.models.Word2Vec(tokenized_train,
                               size=500,
                               window=100,
                               min_count=30,
                               sample=1e-3)

# averaged word vector features
avg_wv_train_features = averaged_word_vectorizer(corpus=tokenized_train,
                                                 model=model,
                                                 num_features=500)
avg_wv_test_features = averaged_word_vectorizer(corpus=tokenized_test,
                                                model=model,
                                                num_features=500)

# tfidf weighted averaged word vector features
vocab = tfidf_vectorizer.vocabulary_
```

```
tfidf_wv_train_features =
tfidf_weighted_averaged_word_vectorizer(corpus=tokenized_train,
tfidf_vectors=tfidf_train_features,
tfidf_vocabulary=vocab, model=model,
num_features=500)
tfidf_wv_test_features =
tfidf_weighted_averaged_word_vectorizer(corpus=tokenized_test,
tfidf_vectors=tfidf_test_features,
tfidf_vocabulary=vocab, model=model,
num_features=500)
```

使用上面的特征提取器从文本文档中提取了全部必要的特征之后，基于前面讨论的四个指标，我们定义一个函数用来评估分类模型，函数如下面代码段所示：

```
from sklearn import metrics
import numpy as np

def get_metrics(true_labels, predicted_labels):
print 'Accuracy:', np.round(
                    metrics.accuracy_score(true_labels,
                                           predicted_labels),
                    2)
print 'Precision:', np.round(
                    metrics.precision_score(true_labels,
                                           predicted_labels,
                                           average='weighted'),
                    2)
print 'Recall:', np.round(
                    metrics.recall_score(true_labels,
                                           predicted_labels,
                                           average='weighted'),
                    2)
print 'F1 Score:', np.round(
                    metrics.f1_score(true_labels,
                                           predicted_labels,
                                           average='weighted'),
                    2)
```

现在我们定义一个函数使用机器学习算法和训练数据来训练模型，使用训练的模型在测试数据上执行预测，接着使用上面的函数评估模型预测性能：

```
def train_predict_evaluate_model(classifier,
                                 train_features, train_labels,
                                 test_features, test_labels):
    # build model
    classifier.fit(train_features, train_labels)
    # predict using model
    predictions = classifier.predict(test_features)
    # evaluate model prediction performance
    get_metrics(true_labels=test_labels,
                predicted_labels=predictions)
    return predictions
```

现在引入了 2 个机器学习算法（前面已经讨论），我们开始使用已经提取的特征建立模型。我们将使用前面提到的 scikit-learn 引入必要的分类算法，以节省我们花费在重写代码上的时间和精力：

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

mnb = MultinomialNB()
svm = SGDClassifier(loss='hinge', n_iter=100)
```

现在下面的代码将使用多项式朴素贝叶斯和支持向量机以及全部不同类型的特征进行模型训练、预测和评估：

```
# Multinomial Naive Bayes with bag of words features
In [558]: mnb_bow_predictions = train_predict_evaluate_model(classifier=mnb,
     ...:                                                    train_features=bow_
                                                             train_features,
     ...:                                                    train_labels=train_
                                                             labels,
     ...:                                                    test_features=bow_test_
                                                             features,
     ...:                                                    test_labels=test_
                                                             labels)
Accuracy: 0.67
Precision: 0.72
Recall: 0.67
F1 Score: 0.65

# Support Vector Machine with bag of words features
In [559]: svm_bow_predictions = train_predict_evaluate_model(classifier=svm,
     ...:                                                    train_features=bow_
                                                             train_features,
     ...:                                                    train_labels=train_
                                                             labels,
     ...:                                                    test_features=bow_test_
                                                             features,
     ...:                                                    test_labels=test_
                                                             labels)
Accuracy: 0.61
Precision: 0.66
Recall: 0.61
F1 Score: 0.62

# Multinomial Naive Bayes with tfidf features
In [560]: mnb_tfidf_predictions = train_predict_evaluate_
model(classifier=mnb,
     ...:                                                    train_features=tfidf_
                                                             train_features,
     ...:                                                    train_labels=train_
                                                             labels,
     ...:                                                    test_features=tfidf_
                                                             test_features,
     ...:                                                    test_labels=test_
                                                             labels)
Accuracy: 0.72
Precision: 0.78
Recall: 0.72
F1 Score: 0.7

# Support Vector Machine with tfidf features
In [561]: svm_tfidf_predictions = train_predict_evaluate_
model(classifier=svm,
     ...:                                                    train_features=tfidf_
                                                             train_features,
     ...:                                                    train_labels=train_
                                                             labels,
```

```
    ...:                                                    test_features=tfidf_
                                                            test_features,
    ...:                                                    test_labels=test_
                                                            labels)
Accuracy: 0.77
Precision: 0.77
Recall: 0.77
F1 Score: 0.77

# Support Vector Machine with averaged word vector features
In [562]: svm_avgwv_predictions = train_predict_evaluate_
model(classifier=svm,
    ...:                                                    train_features=avg_wv_
                                                            train_features,
    ...:                                                    train_labels=train_
                                                            labels,
    ...:                                                    test_features=avg_wv_
                                                            test_features,
    ...:                                                    test_labels=test_
                                                            labels)
Accuracy: 0.55
Precision: 0.55
Recall: 0.55
F1 Score: 0.52

# Support Vector Machine with tfidf weighted averaged word vector features
In [563]: svm_tfidfwv_predictions = train_predict_evaluate_model(classifier
=svm,
    ...:
train_features=tfidf_wv_train_features,
    ...:
train_labels=train_labels, test_features=tfidf_wv_test_features,
    ...:                          test_labels=test_labels)
Accuracy: 0.53
Precision: 0.55
Recall: 0.53
F1 Score: 0.52
```

我们使用不同类型的特征建立了 6 个模型，使用测试数据评估了模型的性能。从上面的结果我们可以看到使用 TF-IDF 特征的 SVM 模型获得了最好的结果，准确率、精确率、召回率和 F1 score 均为 77%。我们可以建立 SVM TF-IDF 模型的混淆矩阵，以便了解模型性能不好的具体分类的情况：

```
In [597]: import pandas as pd
    ...: cm = metrics.confusion_matrix(test_labels, svm_tfidf_predictions)
    ...: pd.DataFrame(cm, index=range(0,20), columns=range(0,20))
Out[597]:
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	157	3	0	1	1	0	2	3	4	1	4	4	1	4	5	34	3	7	7	22
1	1	225	8	7	8	14	8	0	2	1	0	2	5	4	4	1	4	0	3	0
2	1	20	219	19	9	18	8	1	0	0	0	3	5	2	3	2	1	1	2	0
3	1	11	25	223	9	5	8	2	1	1	1	2	6	3	1	0	1	0	0	0
4	0	4	7	15	228	6	5	2	3	1	0	3	9	3	3	1	1	0	1	0
5	0	21	18	1	2	272	0	1	1	0	0	0	4	3	1	0	0	1	0	0
6	0	2	7	11	12	2	269	10	3	2	1	1	10	1	4	0	2	1	1	0
7	1	5	2	2	2	3	4	247	19	1	3	2	9	3	2	0	3	3	4	1
8	3	1	0	4	2	2	5	27	252	3	4	2	1	4	1	3	2	2	4	0
9	2	1	1	0	2	3	4	3	6	277	12	2	1	1	2	4	2	0	2	0
10	0	0	0	0	0	0	1	3	2	4	282	1	2	1	4	1	0	1	1	0
11	3	5	3	3	1	2	2	2	2	3	0	259	6	2	0	1	5	2	5	0
12	1	6	6	15	7	2	13	10	8	4	4	2	211	4	5	1	1	1	0	1
13	2	4	0	1	2	4	3	0	2	1	1	1	7	268	4	2	3	0	3	0
14	0	5	3	0	2	4	2	5	4	1	2	0	8	3	264	2	4	1	3	1
15	11	1	0	0	1	1	0	0	4	1	2	2	1	7	5	291	4	4	3	5
16	4	1	0	0	0	4	2	1	7	2	2	11	3	2	4	2	227	3	13	3
17	6	0	1	0	1	3	0	2	3	2	4	6	1	3	1	6	5	259	10	2
18	10	1	2	1	0	1	2	1	5	3	3	7	0	9	6	4	33	7	164	3
19	21	5	0	1	0	2	4	3	7	2	1	1	0	11	3	57	22	7	3	63

图 4-6　基于 SVM 模型的 20 类分类的混淆矩阵

从图4-6混淆矩阵上，我们可以看到很多类标签为0的文档被错误地分类到类标签15里面，同样对于类标签18的很多文档被错误地分类到类标签16里面。很多类型标识19的文档被错误地分类到类型标识15里面。打印类型名字，我们可以看到如下输出：

```
In [600]: class_names = dataset.target_names
     ...: print class_names[0], '->', class_names[15]
     ...: print class_names[18], '->', class_names[16]
     ...: print class_names[19], '->', class_names[15]
alt.atheism -> soc.religion.christian
talk.politics.misc -> talk.politics.guns
talk.religion.misc -> soc.religion.christian
```

从前面的输出可以看到错误分类与实际分类并没有显著的不同。Christian、religion和atheism都是与上帝和宗教存在有关的概念，可能会有相似的特征。杂项问题和枪支都与政治有关，必然有相似的特征。我们可以使用下面的代码，进一步地详细查看和分析被错误分类的文档（由于受篇幅限制，仅仅给出每种情况前几个被错误分类的文档）：

```
In [621]: import re
     ...: num = 0
     ...: for document, label, predicted_label in zip(test_corpus, test_
labels, svm_tfidf_predictions):
     ...:     if label == 0 and predicted_label == 15:
     ...:         print 'Actual Label:', class_names[label]
     ...:         print 'Predicted Label:', class_names[predicted_label]
     ...:         print 'Document:-'
     ...:         print re.sub('\n', ' ', document)
     ...:         print
     ...:         num += 1
     ...:         if num == 4:
     ...:             break
     ...:
     ...:
Actual Label: alt.atheism
Predicted Label: soc.religion.christian
Document:-
I would like a list of Bible contadictions from those of you who dispite
being free from Christianity are well versed in the Bible.

Actual Label: alt.atheism
Predicted Label: soc.religion.christian
Document:-
  They spent quite a bit of time on the wording of the Constitution.  They
picked words whose meanings implied the intent.  We have already looked in
the dictionary to define the word.  Isn't this sufficient?  But we were
discussing it in relation to the death penalty.  And, the Constitution need
not define each of the words within.  Anyone who doesn't know what cruel is
can look in the dictionary (and we did).

Actual Label: alt.atheism
Predicted Label: soc.religion.christian
Document:-
Our Lord and Savior David Keresh has risen!     He has been seen
alive!         Spread the word!     ----------------------------
-----------------------------------------------------------------

Actual Label: alt.atheism
Predicted Label: soc.religion.christian
Document:-
  "This is your god" (from John Carpenter's "They Live," natch)

In [623]: num = 0
```

```
    ...: for document, label, predicted_label in zip(test_corpus, test_
labels, svm_tfidf_predictions):
    ...:     if label == 18 and predicted_label == 16:
    ...:         print 'Actual Label:', class_names[label]
    ...:         print 'Predicted Label:', class_names[predicted_label]
    ...:         print 'Document:-'
    ...:         print re.sub('\n', ' ', document)
    ...:         print
    ...:         num += 1
    ...:         if num == 4:
    ...:             break
    ...:
    ...:
Actual Label: talk.politics.misc
Predicted Label: talk.politics.guns
Document:-
After the initial gun battle was over, they had 50 days to come out
peacefully. They had their high priced lawyer, and judging by the posts here
they had some public support. Can anyone come up with a rational explanation
why the didn't come out (even after they negotiated coming out after the
radio sermon) that doesn't include the Davidians wanting to commit suicide/
murder/general mayhem?

Actual Label: talk.politics.misc
Predicted Label: talk.politics.guns
Document:-
Yesterday, the FBI was saying that at least three of the bodies had gunshot
wounds, indicating that they were shot trying to escape the fire.  Today's
paper quotes the medical examiner as saying that there is no evidence of
gunshot wounds in any of the recovered bodies.  At the beginning of this
siege, it was reported that while Koresh had a class III (machine gun)
license, today's paper quotes the government as saying, no, they didn't have
a license.  Today's paper reports that a number of the bodies were found
with shoulder weapons next to them, as if they had been using them while
dying -- which doesn't sound like the sort of action I would expect from a
suicide.  Our government lies, as it tries to cover over its incompetence
and negligence.  Why should I believe the FBI's claims about anything else,
when we can see that they are LYING?  This system of government is beyond
reform.

Actual Label: talk.politics.misc
Predicted Label: talk.politics.guns
Document:-
  Well, for one thing most, if not all the Dividians (depending on whether
they could show they acted in self-defense and there were no illegal
weapons), could have gone on with their life as they were living it. No one
was forcing them to give up their religion or even their legal weapons. The
Dividians had survived a change in leadership before so even if Koresch
himself would have been convicted and sent to jail, they still could have
carried on.   I don't think the Dividians were insane, but I don't see a
reason for mass suicide (if the fire was intentional set by some of the
Dividians.) We also don't know that, if the fire was intentionally set from
inside, was it a generally know plan or was this something only an inner
circle knew about, or was it something two or three felt they had to do
with or without Koresch's knowledge/blessing, etc.? I don't know much about
Masada. Were some people throwing others over? Did mothers jump over with
their babies in their arms?

Actual Label: talk.politics.misc
Predicted Label: talk.politics.guns
Document:-
rja@mahogany126.cray.com (Russ Anderson) writes...      The fact is that
```

```
Koresh and his followers involved themselves  in a gun battle to control
the Mt Carmel complex. That is not  in dispute. From what I remember of the
trial, the authories  couldn't reasonably establish who fired first, the
big reason  behind the aquittal. Mitchell S Todd
```

你可以看到我们是如何分析和查看被错误分类的文档的，然后回到前面步骤，调整优化特征提取方法，通过删除特定的单词或调整单词权重来减少或增加影响程度。

至此，就要结束对文本分类系统的讨论和实施了。你可以自由地使用其他创新性特征提取技术或有监督学习算法实现更多的模型，并比较它们的性能。

4.9 应用

文本分类和归类可以用于在很多真实的场景和应用中，包括：

- 新闻文章的归类。
- 垃圾邮件的过滤。
- 音乐或电影的分类。
- 情感分析。
- 语言检测。

文本应用具有无限的可能，经过一些小小的努力，你就可以使用分类来解决各类问题，把一些费时的操作和场景实现自动化。

4.10 小结

文本分类的确是一个强大的工具，本章已经介绍了与之相关的几乎所有方面。从文本分类的定义和应用开始我们的学习之旅，接着我们将自动文本分类定义为有监督学习问题并且查看了不同类型的文本分类。我们也简要介绍了一些与各类算法相关的机器学习概念。一个典型的文本分类系统路线图（蓝图）定义描述了不同的模块和建立端到端的文本分类器包含的步骤。我们详细叙述了蓝图中的每个模块。第3章详细阐述了规范化，并为文本分类建立了规范化模块。在本章我们详细介绍了不同的特征提取技术，包括词袋、TF-IDF、高级词向量技术。

你现在不仅清楚地掌握了有监督机器学习的数学表示和概念，而且掌握了使用示例代码实现它们的方法。我们讨论了不同的机器学习方法，主要有多项式朴素贝叶斯和支持向量机，我们浏览了评估分类模型性能的方法，并实现了这些度量方法。最后，我们将所学的东西放在一起，在真实数据上建立了一个鲁棒的20类文本分类系统，并详细分析了文本分类模型的性能。我们通过简述文本分类经常使用的领域完美地结束了我们的讨论。

这里只是通过分类粗略地研究了文本分析的表面。在后续的章节中，我们将更多地着眼于分析和获取文本数据洞见的方法。

第5章
文本摘要

我们已在文本分析和自然语言处理的世界中走过了漫长的路。你已经明白如何处理和注释文本数据以将其用于各种应用。我们还涉足了机器学习世界，并利用各种特征提取技术和有监督机器学习算法构建了多类文本分类系统。

在本章，我们将在文本分析领域处理一个稍微不同的问题。世界在科技、贸易、商业和媒体方面迅速发展。昔日我们等待报纸到家，从而了解更新世界各种活动信息的日子已经过去。现在有了互联网和各种形式的社交媒体，我们利用它们以保持日常事件信息的及时更新，并与外界以及亲戚朋友保持联系。通过短信和状态，像 Facebook 和 Twitter 这样的社交媒体网站开辟了完全不同的信息分享和应用维度。人类的注意力往往比较短暂，这导致我们在应用或阅读大文本文档和文章时会感到厌倦无聊。因此，产生了*文本摘要*（text summarization）技术，这是文本分析中极其重要的概念，它被商业企业和分析公司用来缩短和总结大文本文档，从而使它们仍然保留其主要内容或主题，并可以向消费者和客户展示这些摘要信息。这类似于*电梯推销*（elevator pitch），执行摘要可以描述过程、产品、服务或业务，同时它可以在乘坐电梯所需时间内保留其核心的重要主题和价值观。

假设你有一整套文本文档语料库，其范围从句子到段落，你的任务是尝试从中获得有意义的见解。乍一看，这似乎很困难，因为你甚至不知道如何处理这些文件，更不用说对数据使用一些分析或机器学习技术。好的方法是使用一些专门针对文本摘要和信息提取的无监督学习方法。以下是你可以对文本文档进行的一些操作：

- 提取文档中的关键影响短语。
- 提取文档中存在的各种不同的概念或主题。
- 总结文件，以提供保留着整个语料库重要部分的要点。

本章将介绍执行上述三个操作方法的概念、技术和实际实现。现在正式描述在本章中将要尝试解决的问题，以及与之相关的一些概念。给定一组文档，文本摘要旨在缩减语料库中的一个文档或一组文档，以总结成为用户指定长度的摘要，且保留语料库中的关键重要概念和主题。我们还将讨论其他概括文档和信息提取的方法，包括主题模型和关键短语提取。

本章将讨论文本摘要以及从文本文档中提取信息，它们抓住并总结了文档语料库的主要主题或概念。下面将首先详细讨论各种类型的摘要和信息提取技术，其后讨论一些理解实际实现的基本概念。本章还将简要介绍与文本处理和特征提取相关的一些背景知识，然后再转到每种技术。我们将讨论三种主要的概念和技术：关键短语提取、主题模型和自动文摘。

5.1 文本摘要和信息提取

文本摘要和信息提取处理试图从巨大的文本语料库中提取关键的重要概念和主题，本质上是在此过程中对它们进行缩减。在深入了解概念和技术之前，我们应该首先了解对文本摘要的需求。信息过载（information overload）的概念是文本摘要需求背后的主要原因之一。由于印刷和口头媒介占据主导，有了大量的书籍、文章、音频和视频。这一切在公元前三或四世纪就开始了，当时人们查阅大量的书籍，因为书籍的生产似乎没有尽头，而且这种信息的过载常常遭遇到反对。文艺复兴时期，大约在公元1440年Gutenberg发明了印刷术，使书籍、手稿、文章和小册子得以大量生产。这大大增加了信息过载，为此学者控诉了这样的信息过剩情形，它使信息变得非常难以使用、处理和管理。

在20世纪，计算机和技术的进步迎来了数字时代，并最终产生了互联网。互联网为社会媒体、新闻网站、电子邮件、即时通信功能开启了充满生产和消费信息的各种可能性的窗口。反过来这又导致了信息量的爆炸式增长和不需要的垃圾邮件信息、无用的状态和推文——乃至导致在网络上发布更多不需要的内容。

那么，信息过载就意味着存在过多的数据或信息，消费者在做出知情决策时会觉得难以处理。一旦系统输入的信息量超过系统的处理能力时，便会发生过载。我们人类具有有限的认知处理能力，并且还以这样一种方式进行连接，因为思维常常会随时徘徊游离，使得我们不能花很长时间来阅读单个的信息或数据。因此，当我们获得信息后做出定性决策时信息会减少。

到目前为止，你可能已经猜到会在哪里用到这个概念以及为什么我们需要总结和提取信息。企业在做出关键和明智的决策时会蓬勃发展，通常它们拥有大量的数据和信息。但从中获得洞察力不是一件非常容易之事，因为往往不清楚所有数据的处理方式，所以自动化是困难的。管理人员很少有时间听长篇大论，或者浏览重要事件和重要信息页面。摘要和信息提取的思想是得到大量信息文档的重要论题和主题，并将其总结为可以轻松阅读、理解和解读的简短内容，从而简化了在更短的时间内做出良好决策的过程。我们需要能对文本数据执行此操作的有效和可扩展的流程和技术，而最流行的技术是*关键短语提取*（keyphrase extraction）、*主题建模*（topic modeling）和*自动文档摘要*（automated document summarization）。前两种技术更多的是从文档中以概念、标题和主题的形式提取关键信息，从而可以缩略文档；最后一种技术是将大文本文档总结成数行，从而提供该文件正在试图传达的关键内容或信息。我们将在未来几节中详细介绍各种技术及其实际的例子，但现在将简要介绍一下每种技术所涉及的要求及其范围。

- *关键短语提取*也许是三种技术之中最简单的。它涉及从包含其主要概念或主题的文本文档或语料库中提取关键字或短语。它可以说是主题建模的一种简单形式。你可能已经在研究论文中或者甚至在网络商店上的一些产品中看到过所描述的关键字或短语，它们用几个单词或短语来描述对象，突出其主要思想或概念。
- *主题建模*通常涉及使用统计和数学建模技术从文档语料库中提取核心主题、题材或概念。请注意，这里强调文档语料库，是因为你拥有的文档集更多样，你就可以生成更多的主题或概念——与单个文档不同，如果谈及的是单一概念，你将不会收到太

多的主题或概念。主题模型通常也称为概率统计模型（probabilistic statistical model），其使用特定的统计技术，包括奇异值分解和隐含 dirichlet 分布来发现在产生主题和概念的文本数据中的连接潜语义结构。它们广泛用于文本分析甚至生物信息学。

- 自动文档摘要是使用基于统计和机器学习技术的计算机程序或算法来概括文档或文档语料库的过程，以便我们可以获得包含原始文档或语料库的所有基本概念和主题的简短摘要。可用各种各样的技术构建自动化文本摘要器，包括各种基于提取和概括的技术。所有这些算法背后的关键思想是找到原始数据集的代表性子集，使得从语义和概念角度来看数据集的核心要素包含在该子集中。文档摘要通常涉及从单个文档中提取和构建执行摘要。但是，相同的算法可以扩展到多个文档，虽然通常不将多个不同文档结合在一起，这可能将违背算法的初衷。相同的概念不仅应用于文本分析，还适用于图像和视频摘要。

在进一步详细介绍各种技术之前，下一节将讨论一些重要的数学和机器学习概念、文本规范化和特征提取过程。

5.2 重要概念

掌握几个重要的数学和机器学习的概念在日后非常有用，因为我们将立足于这些概念的几个实现。有些你会很熟悉，但为了完整起见，本节将再次简要介绍它们，以便你可以重温内容。本节还将介绍自然语言处理的一些概念。

5.2.1 文档

文档通常是一个包含完整文本数据的实体，包含可选的标题和其他元数据信息。语料库通常由一系列文档组成。这些文档可以是简单的句子或完整的文本信息段落。分词语料库指的是每个文档被分词化或分解成标识的语料库，其中标识通常是单词。

5.2.2 文本规范化

文本规范化是通过技术来清洗、规范化和标准化文本数据的过程，譬如删除特殊符号和字符、去除多余的 HTML 标签、移除停用词、校正拼写、词干提取和词形还原。

5.2.3 特征提取

特征提取是我们从原始文本数据中提取有意义的特征或属性，以将其提供给统计或机器学习算法的过程。这个过程也称为向量化（vectorization），因为该过程的转换结果通常是来自于原始文本标识的数值向量。其原因是常规算法可以对数值向量奏效，并不能直接在原始文本数据上处理。有各种不同的特征提取方法，包括：基于词袋的二进制特征可以告诉我们文档中是否存在一个单词或一组单词，基于词袋的频率特征可以告诉我们文档中一个单词或者一组单词的出现频次，以及词频和逆文档频率或 TF-IDF 加权特征在计算每个词项权重时考虑了词频和逆文档频率。有关特征提取的更多信息，请参阅第 4 章。

5.2.4 特征矩阵

特征矩阵通常是指从文档集合到特征的映射，其中每行表示文档，每列表示具体特征，

特征通常是一个单词或一组单词。我们将通过特征提取后的特征矩阵来表达文档或句子的集合，并且将在后面的实例中经常在统计和机器学习技术之中应用这些矩阵。

5.2.5 奇异值分解

奇异值分解（Singular Value Decomposition，SVD）是线性代数的一种技术，它在摘要算法中经常使用。SVD 是实数或复数矩阵的因子分解的过程。正式地，我们可以定义 SVD 如下。考虑维度为 $m \times n$ 的矩阵 M，其中 m 表示行数，n 表示列数。在数学上，矩阵 M 可以使用 *SVD* 作为因式分解，使得

$$M_{m \times n} = U_{m \times m} S_{m \times n} V^{T}_{n \times n}$$

其中，

- U 是 $m \times m$ 酉矩阵，使得 $U^{T}U = I_{m \times m}$，其中 I 是单位矩阵。U 列表示左奇异向量。
- S 是在矩阵的对角线上具有正实数的对角线 $m \times n$ 矩阵。这通常也表示为显示奇异值的 m 值向量。
- V^{T} 是 $n \times n$ 酉矩阵，使得 $V^{T}V = I_{n \times n}$，其中 I 是单位矩阵。V 行表示右侧奇异向量。

这表明 U 和 V 是正交的。S 的奇异值在摘要算法中尤为重要。我们使用 SVD 是为了进行低秩矩阵逼近，其中我们用矩阵 M 来近似原始矩阵，使得该新矩阵是具有秩为 k 的原始矩阵 M 的截尾矩阵，并且可以由 SVD 表示为 $M = USV^{T}$，其中 S 是原始 S 矩阵的缩减版本，其现在仅由前 k 个最大奇异值组成，其他奇异值表示为零。我们将使用源自 `scipy` 的一个很好的实现来提取顶部的 k 个奇异值，并返回相应的 U、S 和 V 矩阵。在 `utils.py` 文件中我们将使用以下代码段：

```
from scipy.sparse.linalg import svds

def low_rank_svd(matrix, singular_count=2):

    u, s, vt = svds(matrix, k=singular_count)
    return u, s, vt
```

将在主题建模以及有关文本摘要的章节中使用此实现。图 5-1 给出了上述过程的良好描述，其从原始 SVD 分解产生 k 个奇异向量，并显示出了如何从相同的原始矩阵获得低秩矩阵逼近。

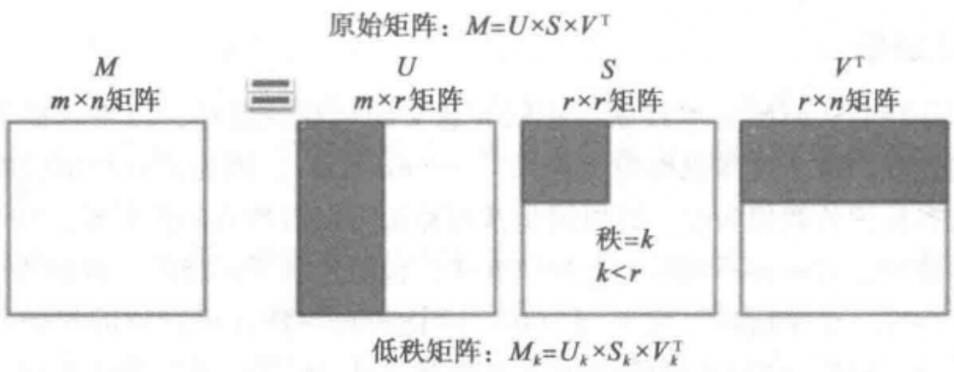


图 5-1 低秩矩阵逼近的奇异值分解

你可以清楚地看到 k 个奇异值保留在低秩矩阵逼近中，以及原始矩阵 M 如何使用 SVD 分解为 U、S 和 V。在我们的计算中，通常矩阵 M 的行将表示词项，列将表示文档。该矩阵也称为词项 - 文档矩阵（term-document matrix），通常是在特征提取之后在应用 SVD 之前将

文档 - 词项矩阵转换为其转置来获得，除非需要了解算法是如何工作的，本章的其余部分将努力使数学相关内容保持在最低限度。以下部分将简要介绍文本规范化和特征提取，以突出本章中将要使用的技术和方法。

5.3 文本规范化

第3章详细介绍了文本规范化，第4章构建了自己的规范化模块。本章将重复使用相同的模块，但将会为我们的算法特别增加几个增强功能。你可以在 normalization.py 文件中找到所有与文本规范化相关的代码。在文本规范化中执行的主要步骤包括：

（1）句子提取。

（2）取消 HTML 转义序列。

（3）扩展缩写词。

（4）文本还原。

（5）删除特殊字符。

（6）删除停用词。

步骤 3 ~ 6 与第4章中的相同，除了步骤5，我们通过用代码 pattern.sub ('', token) 所描绘的空格而不是第4章中的空字符串来替换每个特殊字符。

步骤1是一个新的函数，其中，收到一个文本文档时，删除其换行符，解析文本，将其转换为 ASCII 格式，并将其分解成其句子成分。该函数如以下代码段所描述：

```
def parse_document(document):
    document = re.sub('\n', ' ', document)
    if isinstance(document, str):
        document = document
    elif isinstance(document, unicode):
        return unicodedata.normalize('NFKD', document).encode('ascii',
        'ignore')
    else:
        raise ValueError('Document is not string or unicode!')
    document = document.strip()
    sentences = nltk.sent_tokenize(document)
    sentences = [sentence.strip() for sentence in sentences]
    return sentences
```

步骤2涉及被转义或编码的非转义特殊 HTML 字符。在 www.theukwebdesigncompany.com/articles/entity-escape-characters.php 上的完整列表基本上显示了一些特殊符号甚至常规字符如何转义为不同的代码，例如 & 转义为 &。因此我们使用以下函数来取消它们的转义，并将它们恢复到原来的未转义形式，这样便可在后续阶段规范化它们：

```
from HTMLParser import HTMLParser

html_parser = HTMLParser()
def unescape_html(parser, text):
    return parser.unescape(text)
```

我们还在最终规范化函数中用参数表示词形还原操作，以使其可选，因为在某些情况下，它可以正常工作，而在其他情况下，我们可能不想使用词形还原。完整的规范化函数如下所示：

```
def normalize_corpus(corpus, lemmatize=True, tokenize=False):

    normalized_corpus = []
    for text in corpus:
        text = html_parser.unescape(text)
        text = expand_contractions(text, CONTRACTION_MAP)
        if lemmatize:
            text = lemmatize_text(text)
        else:
            text = text.lower()
        text = remove_special_characters(text)
        text = remove_stopwords(text)
        if tokenize:
            text = tokenize_text(text)
            normalized_corpus.append(text)
        else:
            normalized_corpus.append(text)

    return normalized_corpus
```

我们将使用这个函数来满足大部分的规范化需要。有关第4章讨论的规范化文本的所有详细的辅助函数，请参考 normalization.py 文件。

5.4 特征提取

使用泛型函数（generic function）从文本数据中执行各种类型的特征提取。将要使用的特征类型如下：

- 基于词项次数的二值特征。
- 基于词袋模型的频率特征。
- TF-IDF 权重特征。

我们将在以后的大部分实例中使用以下函数，实现从文本文档中提取特征。你也可以在与本章相关代码文件的 utils.py 模块中找到该函数：

```
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer

def build_feature_matrix(documents, feature_type='frequency'):

    feature_type = feature_type.lower().strip()

    if feature_type == 'binary':
        vectorizer = CountVectorizer(binary=True, min_df=1,
                                     ngram_range=(1, 1))
    elif feature_type == 'frequency':
        vectorizer = CountVectorizer(binary=False, min_df=1,
                                     ngram_range=(1, 1))
    elif feature_type == 'tfidf':
        vectorizer = TfidfVectorizer(min_df=1,
                                     ngram_range=(1, 1))
    else:
        raise Exception("Wrong feature type entered. Possible values:
        'binary', 'frequency', 'tfidf'")

    feature_matrix = vectorizer.fit_transform(documents).astype(float)

    return vectorizer, feature_matrix
```

现在，已经介绍了本章所需的必要背景概念和依赖项，下面将详细深入介绍每个文本摘要和信息提取技术。

5.5 关键短语提取

从非结构化文本文档中提取重要信息的最简单但最强大的技术之一是关键短语提取。关键短语提取也称为词项提取（terminology extraction），定义为从非结构化文本体提取关键重要和相关词项或短语的过程或技术，使得文本文档的核心论题或主题涵盖在这些关键短语中。这种技术属于信息检索和提取的广泛领域。关键短语提取可以应用在许多领域，包括：

- 语义网。
- 基于查询的搜索引擎和爬虫。
- 推荐系统。
- 标注系统。
- 文档相似性。
- 翻译。

在文本分析或 NLP 中，关键短语提取通常是执行更复杂任务的起点，并且其输出本身可以作为更复杂系统的特征。有各种各样的关键短语提取方法。下面将介绍两种：

- 搭配。
- 基于权重标签的短语提取。

这里要记住一个重要事情，我们将提取的短语通常是词汇集，尽管有时可能包括一个单词。如果你正在提取的是关键词，这也称为关键词提取，它是关键短语提取的一个子集。

5.5.1 搭配

词项搭配（collocation）实际上是从分析语料库和语言学中借鉴的一个概念。搭配是一种趋向于频繁发生的序列或一组词，使得该发生频率倾向于超过可称为随机或偶然发生的频率。可以根据诸如名词、动词等各种词项的词性形成各种类型的搭配。有各种方法来提取搭配，最好方法之一是使用一种 n 元分词分组或分割方法，我们从语料库中构造 n 元分词分组，计算每个 n 元分词的频率，并根据它们的出现频率进行排序以得到最频繁的 n 元分词搭配。

这个思想是有一个可以是段落或句子的文档语料库，标记它们以形成句子，拼合句子列表以组成一个大的句子或字符串，在此基础上，基于 n 元分词范围滑动大小为 n 的窗口，并计算整个字符串的 n 元分词。计算完成后，可以根据其发生频率对每个 n 元分词进行计数，然后根据其频率进行排序。这样可以在频率基础上得到最频繁的搭配。

我们将从头开始实现这一点，以便你可以更好地理解算法，然后使用一些 `nltk` 内置的功能来展示相同的内容。首先加载一些必要的依赖关系以及将要计算搭配的一个语料库。将使用 `nltk` Gutenberg 语料库的书——Lewis Carroll 的“Alice in Wonderland”作为我们的语料库。还使用之前指定的规范化模块，来归一化语料库以实现文本内容的规范化：

```
from nltk.corpus import gutenberg
from normalization import normalize_corpus
import nltk
from operator import itemgetter
```

```
# load corpus
alice = gutenberg.sents(fileids='carroll-alice.txt')
alice = [' '.join(ts) for ts in alice]
norm_alice = filter(None, normalize_corpus(alice, lemmatize=False))
# print first line
In [772]: print norm_alice[0]
alice adventures wonderland lewis carroll 1865
```

现在已经加载了语料库，我们将定义一个函数来将语料库压缩成1个大的文本串。以下的函数将有助于对文档语料库实现这一点：

```
def flatten_corpus(corpus):
    return ' '.join([document.strip()
                     for document in corpus])
```

将根据一些词例输入列表和参数n来定义一个函数从而计算n元分词，参数决定了n元分词的程度，如unigram、bigram等。以下代码段为输入序列计算n元分词：

```
def compute_ngrams(sequence, n):
    return zip(*[sequence[index:]
                 for index in range(n)])
```

该函数基本上接受标识序列，并计算所有带序列的列表，其中每个列表都包含来自前一列表的所有项，除了从先前列表中删除的首项。它构造 n 个这样的列表，然后将它们全部连在一起，从而给我们必要的n元分词。可以在下面代码段中看到函数在示例序列中的作用：

```
In [802]: compute_ngrams([1,2,3,4], 2)
Out[802]: [(1, 2), (2, 3), (3, 4)]

In [803]: compute_ngrams([1,2,3,4], 3)
Out[803]: [(1, 2, 3), (2, 3, 4)]
```

上面的输出显示了输入序列的bigrams和trigrams。现在我们将利用这个函数，在它基础上构建产生根据发生频率最大的n元分词。以下代码段可帮助我们获得最大的n元分词：

```
def get_top_ngrams(corpus, ngram_val=1, limit=5):

    corpus = flatten_corpus(corpus)
    tokens = nltk.word_tokenize(corpus)

    ngrams = compute_ngrams(tokens, ngram_val)
    ngrams_freq_dist = nltk.FreqDist(ngrams)
    sorted_ngrams_fd = sorted(ngrams_freq_dist.items(),
                              key=itemgetter(1), reverse=True)
    sorted_ngrams = sorted_ngrams_fd[0:limit]
    sorted_ngrams = [(' '.join(text), freq)
                     for text, freq in sorted_ngrams]

    return sorted_ngrams
```

利用nltk的FreqDist类基于频率创建一个所有n元分词的计数器，进而根据它们的频率对它们进行分类，并根据用户指定的限制返回最大的n元分词。现在将使用以下代码段来计算语料库中的最大bigram和trigram：

```
# top 10 bigrams
In [805]: get_top_ngrams(corpus=norm_alice, ngram_val=2,
     ...:                limit=10)
Out[805]:
[(u'said alice', 123),
```

```
 (u'mock turtle', 56),
 (u'march hare', 31),
 (u'said king', 29),
 (u'thought alice', 26),
 (u'said hatter', 22),
 (u'white rabbit', 22),
 (u'said mock', 20),
 (u'said gryphon', 18),
 (u'said caterpillar', 18)]

# top 10 trigrams
In [806]: get_top_ngrams(corpus=norm_alice, ngram_val=3,
    ...:                 limit=10)
Out[806]:
[(u'said mock turtle', 20),
 (u'said march hare', 10),
 (u'poor little thing', 6),
 (u'white kid gloves', 5),
 (u'little golden key', 5),
 (u'march hare said', 5),
 (u'certainly said alice', 5),
 (u'mock turtle said', 5),
 (u'mouse mouse mouse', 4),
 (u'join dance join', 4)]
```

上面的输出显示了依据 n 元分词生成的两个和三个单词的序列以及它们在整个语料库中出现的次数。我们可以看到大部分的搭配都指向那些说话的人，如“said < person >”；也看到在“Alice in Wonderland”中受欢迎的人物，像假乌龟、国王、兔子、帽匠、当然爱丽丝自己也描绘在上述搭配中。

现在来看看 nltk 的搭配查找器，这使我们能够使用诸如原始频率、点互信息（pointwise mutual information，PMI）等各种度量法来找到搭配。只是为了简要解释一下，可以将两个事件或词项的点互信息按各自概率的乘积计算为它们一起发生概率比率的对数，这里假设它们彼此相互独立。数学上，可以这样表达：

$$\text{pmi}(x,y) = \log \frac{p(x,y)}{p(x)p(y)}$$

这个度量是对称的。以下代码段显示了如何使用这些度量法来计算这些搭配：

```
# bigrams
from nltk.collocations import BigramCollocationFinder
from nltk.collocations import BigramAssocMeasures

finder = BigramCollocationFinder.from_documents([item.split()
                                                 for item
                                                 in norm_alice])
bigram_measures = BigramAssocMeasures()
# raw frequencies
In [813]: finder.nbest(bigram_measures.raw_freq, 10)
Out[813]:
[(u'said', u'alice'),
 (u'mock', u'turtle'),
 (u'march', u'hare'),
 (u'said', u'king'),
 (u'thought', u'alice'),
 (u'said', u'hatter'),
 (u'white', u'rabbit'),
 (u'said', u'mock'),
 (u'said', u'caterpillar'),
 (u'said', u'gryphon')]
# pointwise mutual information
In [814]: finder.nbest(bigram_measures.pmi, 10)
```

```
Out[814]:
[(u'abide', u'figures'),
 (u'acceptance', u'elegant'),
 (u'accounting', u'tastes'),
 (u'accustomed', u'usurpation'),
 (u'act', u'crawling'),
 (u'adjourn', u'immediate'),
 (u'adoption', u'energetic'),
 (u'affair', u'trusts'),
 (u'agony', u'terror'),
 (u'alarmed', u'proposal')]

# trigrams
from nltk.collocations import TrigramCollocationFinder
from nltk.collocations import TrigramAssocMeasures

finder = TrigramCollocationFinder.from_documents([item.split()
                                                  for item
                                                  in norm_alice])
trigram_measures = TrigramAssocMeasures()
# raw frequencies
In [817]: finder.nbest(trigram_measures.raw_freq, 10)
Out[817]:
[(u'said', u'mock', u'turtle'),
 (u'said', u'march', u'hare'),
 (u'poor', u'little', u'thing'),
 (u'little', u'golden', u'key'),
 (u'march', u'hare', u'said'),
 (u'mock', u'turtle', u'said'),
 (u'white', u'kid', u'gloves'),
 (u'beau', u'ootiful', u'soo'),
 (u'certainly', u'said', u'alice'),
 (u'might', u'well', u'say')]
# pointwise mutual information
In [818]: finder.nbest(trigram_measures.pmi, 10)
Out[818]:
[(u'accustomed', u'usurpation', u'conquest'),
 (u'adjourn', u'immediate', u'adoption'),
 (u'adoption', u'energetic', u'remedies'),
 (u'ancient', u'modern', u'seaography'),
 (u'apple', u'roast', u'turkey'),
 (u'arithmetic', u'ambition', u'distraction'),
 (u'brother', u'latin', u'grammar'),
 (u'canvas', u'bag', u'tied'),
 (u'cherry', u'tart', u'custard'),
 (u'circle', u'exact', u'shape')]
```

现在你了解了如何使用 n 元分词生成方法来计算语料库的搭配。我们将在下一节中介绍基于词性标注和词项权重的更好方法来生成关键短语。

5.5.2 基于权重标签的短语提取

这是一个稍微不同的提取关键短语的方法。这种方法参考了一些论文，即 K. Barker 和 N. Cornachhia 的“Using Noun Phrase Heads to Extract Document Keyphrases”和 Ian Witten 等人的“KEA：Practical Automatic Keyphrase Extraction”，你可以参考他们的实验和方法以进一步了解细节。这里，在算法中将遵循以下两个步骤：

（1）使用浅层分析提取所有名词短语词块。

（2）计算每个词块的 TF-IDF 权重并返回最大加权短语。

对于第一步，我们将使用基于词性（POS）标签的简单模式来提取名词短语词块。你可以从第 3 章熟悉这部分内容，那里探讨了分块和浅层分析。在讨论算法之前，先定义一下语

料库，我们将要以此来测试我们的实现。使用从维基百科获取的关于大象的示例描述，如以下代码所示：

```
toy_text = """
Elephants are large mammals of the family Elephantidae
and the order Proboscidea. Two species are traditionally recognised,
the African elephant and the Asian elephant. Elephants are scattered
throughout sub-Saharan Africa, South Asia, and Southeast Asia. Male
African elephants are the largest extant terrestrial animals. All
elephants have a long trunk used for many purposes,
particularly breathing, lifting water and grasping objects. Their
incisors grow into tusks, which can serve as weapons and as tools
for moving objects and digging. Elephants' large ear flaps help
to control their body temperature. Their pillar-like legs can
carry their great weight. African elephants have larger ears
and concave backs while Asian elephants have smaller ears
and convex or level backs.
"""
```

现在已经准备好了语料库，我们将使用模式"NP:{<DT>? <JJ>* <NN.*>+}"来从文档/句子的语料库中提取所有可能的名词短语。你后续可以随时尝试更复杂的模式，包括动词、形容词或者副词短语。但是简便起见，我们集中关注核心逻辑。一旦有了自己的模式，便可使用以下代码段定义一个函数来解析和提取这些短语（现在也就此加载了必要的依赖项）：

```
from normalization import parse_document
import itertools
import nltk
from normalization import stopword_list
from gensim import corpora, models

def get_chunks(sentences, grammar = r'NP: {<DT>? <JJ>* <NN.*>+}'):
    # build chunker based on grammar pattern
    all_chunks = []
    chunker = nltk.chunk.regexp.RegexpParser(grammar)

    for sentence in sentences:
        # POS tag sentences
        tagged_sents = nltk.pos_tag_sents(
                            [nltk.word_tokenize(sentence)])
    # extract chunks
    chunks = [chunker.parse(tagged_sent)
              for tagged_sent in tagged_sents]
    # get word, pos tag, chunk tag triples
    wtc_sents = [nltk.chunk.tree2conlltags(chunk)
                 for chunk in chunks]

    flattened_chunks = list(
                        itertools.chain.from_iterable(
                            wtc_sent for wtc_sent in wtc_sents)
                       )
    # get valid chunks based on tags
    valid_chunks_tagged = [(status, [wtc for wtc in chunk])
                    for status, chunk
                    in itertools.groupby(flattened_chunks,
                                         lambda (word,pos,chunk): chunk
                                         != 'O')]
    # append words in each chunk to make phrases
    valid_chunks = [' '.join(word.lower()
                             for word, tag, chunk
                             in wtc_group
                                 if word.lower()
                                     not in stopword_list)
                             for status, wtc_group
```

```
                                in valid_chunks_tagged
                                  if status]
        # append all valid chunked phrases
        all_chunks.append(valid_chunks)

    return all_chunks
```

上述函数中的注释简单易懂。基本上，我们有一个已定义的语法模式来分块或提取名词短语。我们在同一模式中定义一个分块器，对于文档中的每个句子，首先用它的 POS 标签来标注它（因此，不应该对文本进行规范化），然后构建一个具有名词短语的浅层分析树作为词块和其他全部基于 POS 标签的单词作为缝隙，缝隙是不属于任何词块的部分。完成此操作后，我们使用 `tree2conlltags` 函数来生成（`w`，`t`，`c`）三元组，它们是第 3 章中介绍的单词、POS 标签和 IOB 格式的词块标签。删除所有词块带有`'O'`标签的标签，因为它们基本上是不属于任何词块的单词或词项（如果你还记得第 3 章中浅层分析的讨论）。最后，从这些有效的词块中，组合分块的词项，并从每个词块分组中生成短语。可以在下面的代码段中看到这个函数对语料库的操作：

```
sentences = parse_document(toy_text)
valid_chunks = get_chunks(sentences)
# print all valid chunks
In [834]: print valid_chunks
 [['elephants', 'large mammals', 'family elephantidae', 'order
proboscidea'], ['species', 'african elephant', 'asian elephant'],
['elephants', 'sub-saharan africa', 'south asia', 'southeast asia'],
['male african elephants', 'extant terrestrial animals'], ['elephants',
'long trunk', 'many purposes', 'breathing', 'water', 'grasping objects'],
['incisors', 'tusks', 'weapons', 'tools', 'objects', 'digging'],
['elephants', 'large ear flaps', 'body temperature'], ['pillar-like legs',
'great weight'], ['african elephants', 'ears', 'backs', 'asian elephants',
'ears', 'convex', 'level backs']]
```

上面的输出显示了文档中每个句子的所有有效关键短语。你已经看到，由于我们针对的是名词短语，因此所有的短语都是涉及基于名词的对象。现在将通过实现第二步的必要逻辑来构建 `get_chunks()`函数，在这里将使用 `gensim` 来构建一个基于 TF-IDF 的关键短语模型，然后根据每个关键短语在语料库中的发生频率来计算它的 TF-IDF 权重。最后，将根据它们的 TF-IDF 权重对这些关键短语进行分类排序，并根据用户指定的参数 `n` 来显示前 `n` 个关键短语：

```
def get_tfidf_weighted_keyphrases(sentences,
                                  grammar=r'NP: {<DT>? <JJ>* <NN.*>+}',
                                  top_n=10):
    # get valid chunks
    valid_chunks = get_chunks(sentences, grammar=grammar)
    # build tf-idf based model
    dictionary = corpora.Dictionary(valid_chunks)
    corpus = [dictionary.doc2bow(chunk) for chunk in valid_chunks]
    tfidf = models.TfidfModel(corpus)
    corpus_tfidf = tfidf[corpus]
    # get phrases and their tf-idf weights
    weighted_phrases = {dictionary.get(id): round(value,3)
                        for doc in corpus_tfidf
                        for id, value in doc}
    weighted_phrases = sorted(weighted_phrases.items(),
                              key=itemgetter(1), reverse=True)
    # return top weighted phrases
    return weighted_phrases[:top_n]
```

现在在下面的代码段中可以用前面的小语料库来测试这个函数，以生成前十个关键短语：

```
# top 10 tf-idf weighted keyphrases for toy_text
In [836]: get_tfidf_weighted_keyphrases(sentences, top_n=10)
Out[836]:
[(u'pillar-like legs', 0.707),
 (u'male african elephants', 0.707),
 (u'great weight', 0.707),
 (u'extant terrestrial animals', 0.707),
 (u'large ear flaps', 0.684),
 (u'body temperature', 0.684),
 (u'ears', 0.667),
 (u'species', 0.577),
 (u'african elephant', 0.577),
 (u'asian elephant', 0.577)]
```

有趣的是，我们看到了在关键短语中所描绘的各种类型的大象，像亚洲和非洲大象，还有大象的典型属性，如“great weight”“large ear flaps”和“pillar like legs”。因此，你可以了解关键短语提取如何从文本文档中提取关键重要的概念，并对其进行总结。在其他语料库中尝试这些函数可以看到有趣的结果！

5.6 主题建模

我们已经了解使用几种可以提取关键短语的技术。虽然这些短语从文档或语料库中指明了关键点，但它是简单的，特别是在文档语料库中有相互区别的主题或概念时，往往没有描述语料库中的各种主题或概念，主题模型专门设计用于从包含各种类型文档的大型语料库中提取各种不同概念或主题，其中每个文档涉及一个或多个概念。这些概念可以是从思想到意见、事实、展望、陈述等。主题建模的主要目的是使用数学和统计技术来发现语料库中的隐藏和潜在语义结构。

主题建模涉及从文档词项中提取特征，并使用矩阵分解和 SVD 等数学结构和框架来生成彼此不同的词簇或词组，并且这些词簇形成主题或概念。这些概念可以用来解释语料库的主要主题，也可以在各种文档中频繁共同出现的单词之间进行语义连接。构建主题模型有各种框架和算法。我们将介绍以下三种方法：

- 隐含语义索引。
- 隐含 Dirichlet 分布。
- 非负矩阵分解。

上面前两种方法颇受欢迎，已经存在很长时间了。最后一种技术——非负矩阵分解，是一种很新的技术，它非常有效并且取得了佳绩。我们将利用 gensim 和 scikit-learn 来进行实际的实现，并且还会介绍如何基于隐含语义索引来构建自己的主题模型。这将让你了解这些技术如何工作，以及如何将数学框架转化为实际的实现。最初将使用以下的小语料库来测试我们的主题模型：

```
toy_corpus = ["The fox jumps over the dog",
"The fox is very clever and quick",
"The dog is slow and lazy",
"The cat is smarter than the fox and the dog",
"Python is an excellent programming language",
"Java and Ruby are other programming languages",
```

```
"Python and Java are very popular programming languages",
"Python programs are smaller than Java programs"]
```

你可以看到我们在前面的语料库中有8个文档：前四个是关于各种动物的，后四个是关于编程语言的。因此，这表明了语料库中有两个不同的主题。我们可以使用我们的大脑来概括，但以下部分将尝试使用计算方法来提取相同的信息。一旦构建了一些主题建模框架，我们将使用相同的方式来生成源自Amazon实际产品评论的主题。

5.6.1 隐含语义索引

我们的第一种技术是隐含语义索引（latent Semantic Indexing，LSI），自20世纪70年代以来一直在使用中，起初它作为一种统计技术，用于关联并找出语料库中的语义链接词。LSI不仅用于文本摘要，还用于信息检索和搜索。LSI使用了5.2节中所介绍的非常流行的SVD技术。LSI背后的主要原理是相似词趋向于在相同语境中使用，因此往往会更多地共同出现。LSI术语来自这样的事实，即这种技术有能力发现潜在的隐含词，而它们在语义上相互关联，形成主题。

现在我们将尝试通过利用gensim实现LSI，并从那个小语料库中提取主题。首先，我们加载必要的依赖关系，并使用以下代码段对小语料库进行规范化：

```
from gensim import corpora, models
from normalization import normalize_corpus
import numpy as np

norm_tokenized_corpus = normalize_corpus(toy_corpus, tokenize=True)
# view the normalized tokenized corpus
In [841]: norm_tokenized_corpus
Out[841]:
[[u'fox', u'jump', u'dog'],
 [u'fox', u'clever', u'quick'],
 [u'dog', u'slow', u'lazy'],
 [u'cat', u'smarter', u'fox', u'dog'],
 [u'python', u'excellent', u'programming', u'language'],
 [u'java', u'ruby', u'programming', u'language'],
 [u'python', u'java', u'popular', u'programming', u'language'],
 [u'python', u'program', u'small', u'java', u'program']]
```

我们现在建立一个字典或词汇，gensim使用它将每个唯一词映射到一个数值。构建完成后，我们将前面的分词语料库转换成一个数值型的词袋向量表示，其中句子中的每个词及其频率由元组（词，频率）描绘，如以下代码段所示：

```
# build the dictionary
dictionary = corpora.Dictionary(norm_tokenized_corpus)
# view the dictionary mappings
In [846]: print dictionary.token2id
{u'program': 17, u'lazy': 5, u'clever': 4, u'java': 13, u'programming': 10,
u'language': 11, u'python': 9, u'smarter': 7, u'fox': 1, u'dog': 2, u'cat':
8, u'jump': 0, u'popular': 15, u'slow': 6, u'excellent': 12, u'quick': 3,
u'small': 16, u'ruby': 14}

# convert tokenized documents into bag of words vectors
corpus = [dictionary.doc2bow(text) for text in norm_tokenized_corpus]
# view the converted vectorized corpus
In [849]: corpus
Out[849]:
[[(0, 1), (1, 1), (2, 1)],
 [(1, 1), (3, 1), (4, 1)],
 [(2, 1), (5, 1), (6, 1)],
```

```
[(1, 1), (2, 1), (7, 1), (8, 1)],
[(9, 1), (10, 1), (11, 1), (12, 1)],
[(10, 1), (11, 1), (13, 1), (14, 1)],
[(9, 1), (10, 1), (11, 1), (13, 1), (15, 1)],
[(9, 1), (13, 1), (16, 1), (17, 2)]]
```

我们现在将对这个语料库建立一个 TF-IDF 加权模型，其中每个文档中的每个词将包含其 TF-IDF 权重。这类似于特征提取或向量空间转换，其中每个文档由其词的 TF-IDF 向量表示，就如我们之前所做的那样。完成之后，我们将在这些特征之上构建一个 LSI 模型，并输入我们想要生成的主题数量。这个数字是基于直觉和试错，所以当你在语料库上建立主题模型时，可以随意尝试这个参数。根据我们期望小语料库所包含的主题数量将此参数设置为 2：

```
# build tf-idf feature vectors
tfidf = models.TfidfModel(corpus)
corpus_tfidf = tfidf[corpus]

# fix the number of topics
total_topics = 2

# build the topic model
lsi = models.LsiModel(corpus_tfidf,
                      id2word=dictionary,
                      num_topics=total_topics)
```

现在我们建立了主题建模框架，我们可以在以下代码段中看到生成的主题：

```
In [855]: for index, topic in lsi.print_topics(total_topics):
     ...:     print 'Topic #'+str(index+1)
     ...:     print topic
     ...:     print
Topic #1
-0.459*"language" + -0.459*"programming" + -0.344*"java" + -0.344*"python" +
-0.336*"popular" + -0.318*"excellent" + -0.318*"ruby" + -0.148*"program" +
-0.074*"small" + -0.000*"clever"

Topic #2
0.459*"dog" + 0.459*"fox" + 0.444*"jump" + 0.322*"smarter" + 0.322*"cat" +
0.208*"lazy" + 0.208*"slow" + 0.208*"clever" + 0.208*"quick" + -0.000*"ruby"
```

让我们花一点时间来理解这些结果。

首先，忽略权重，你可以看到第一个主题包含与编程语言相关的词，第二个主题包含与动物有关的词，这符合我们前面提到的小语料库的两个主要概念。如果看看权重，对于每个主题都有贡献的词则存在较高的权重和相同的符号。第一个主题具有负权重的相关词，第二个主题具有正权重的相关词。符号只是表明主题的方向，也就是说，主题中相似的关联词将具有相同的符号或方向。下面的函数有助于在有阈值或无阈值的情况下以更好的方式显示主题：

```
def print_topics_gensim(topic_model, total_topics=1,
                        weight_threshold=0.0001,
                        display_weights=False,
                        num_terms=None):

    for index in range(total_topics):
        topic = topic_model.show_topic(index)
        topic = [(word, round(wt,2))
                 for word, wt in topic
                 if abs(wt) >= weight_threshold]
        if display_weights:
            print 'Topic #'+str(index+1)+' with weights'
```

```
            print topic[:num_terms] if num_terms else topic
        else:
            print 'Topic #'+str(index+1)+' without weights'
            tw = [term for term, wt in topic]
            print tw[:num_terms] if num_terms else tw
        print
```

可以使用以下代码段对小语料库的主题模型测试这个函数，以了解如何获取主题并调整参数：

```
# print topics without weights
In [860]: print_topics_gensim(topic_model=lsi,
     ...:                     total_topics=total_topics,
     ...:                     num_terms=5,
     ...:                     display_weights=False)
Topic #1 without weights
[u'language', u'programming', u'java', u'python', u'popular']
Topic #2 without weights
[u'dog', u'fox', u'jump', u'smarter', u'cat']

# print topics with their weights
In [861]: print_topics_gensim(topic_model=lsi,
     ...:                     total_topics=total_topics,
     ...:                     num_terms=5,
     ...:                     display_weights=True)
Topic #1 with weights
[(u'language', -0.46), (u'programming', -0.46), (u'java', -0.34),
(u'python', -0.34), (u'popular', -0.34)]

Topic #2 with weights
[(u'dog', 0.46), (u'fox', 0.46), (u'jump', 0.44), (u'smarter', 0.32),
(u'cat', 0.32)]
```

我们已经成功地使用 LSI 构建了一个主题建模框架，其可以从文档语料库中区分和显示主题。现在我们通过本章开始讨论的数学概念，将使用 SVD 从头开始构建自己的 LSI 主题模型框架。我们将首先建立一个 TF-IDF 特征矩阵，其实际上是一个文档 - 词项矩阵（如果你记得我们在第 4 章的分类练习）。在使用以下代码段计算 SVD 之前，我们将转置矩阵，形成词项 - 文档矩阵。此外，我们还修正了想要生成的主题数量，并从特征中提取词项的名称，以便我们可以将它们与权重进行对照：

```
from utils import build_feature_matrix, low_rank_svd

# build the term document tf-idf weighted matrix
norm_corpus = normalize_corpus(toy_corpus)
vectorizer, tfidf_matrix = build_feature_matrix(norm_corpus,
                                    feature_type='tfidf')
td_matrix = tfidf_matrix.transpose()
td_matrix = td_matrix.multiply(td_matrix > 0)

# fix total topics and get the terms used in the term-document matrix
total_topics = 2
feature_names = vectorizer.get_feature_names()
```

完成后，使用 `low_rank_svd()`函数计算我们的词项 - 文档矩阵的 SVD，以便我们构建一个只取前 k 个奇异向量的低秩矩阵逼近，这将等于我们在此情况下的主题数量。通过使用 S 和 U 分量，我们将它们一起相乘以生成每个主题的每个词项及其权重，类似于你之前所见，这给我们每个主题所需的权重：

```
u, s, vt = low_rank_svd(td_matrix, singular_count=total_topics)
weights = u.transpose() * s[:, None]
```

现在我们有了词项的权重，需要将它们连接回到我们的词项。我们定义两个效用函数，用于通过连接词项与权重来生成这些主题，然后使用具有可配置参数的函数来打印这些主题：

```
# get topics with their terms and weights
def get_topics_terms_weights(weights, feature_names):
    feature_names = np.array(feature_names)
    sorted_indices = np.array([list(row[::-1])
                           for row
                           in np.argsort(np.abs(weights))])
    sorted_weights = np.array([list(wt[index])
                              for wt, index
                              in zip(weights,sorted_indices)])
    sorted_terms = np.array([list(feature_names[row])
                             for row
                             in sorted_indices])

    topics = [np.vstack((terms.T,
                     term_weights.T)).T
              for terms, term_weights
              in zip(sorted_terms, sorted_weights)]

    return topics

# print all the topics from a corpus
def print_topics_udf(topics, total_topics=1,
                     weight_threshold=0.0001,
                     display_weights=False,
                     num_terms=None):

    for index in range(total_topics):
        topic = topics[index]
        topic = [(term, float(wt))
                 for term, wt in topic]
        topic = [(word, round(wt,2))
                 for word, wt in topic
                 if abs(wt) >= weight_threshold]

        if display_weights:
            print 'Topic #'+str(index+1)+' with weights'
            print topic[:num_terms] if num_terms else topic
        else:
            print 'Topic #'+str(index+1)+' without weights'
            tw = [term for term, wt in topic]
            print tw[:num_terms] if num_terms else tw
        print
```

现在，我们已经准备好查看运行的函数。以下代码段利用先前定义的函数来生成主题，该函数使用了 LSI 实现，其中采用了 SVD 连接每个主题词项及其权重：

```
In [871]: topics = get_topics_terms_weights(weights, feature_names)
     ...: print_topics_udf(topics=topics,
     ...:                  total_topics=total_topics,
     ...:                  weight_threshold=0,
     ...:                  display_weights=True)
Topic #1 with weights
[(u'dog', 0.72), (u'fox', 0.72), (u'jump', 0.43), (u'smarter', 0.34),
(u'cat', 0.34), (u'slow', 0.23), (u'lazy', 0.23), (u'quick', 0.23),
(u'clever', 0.23), (u'program', 0.0), (u'java', 0.0), (u'excellent', -0.0),
(u'small', 0.0), (u'popular', 0.0), (u'python', 0.0), (u'programming',
-0.0), (u'language', -0.0), (u'ruby', 0.0)]

Topic #2 with weights
[(u'programming', -0.73), (u'language', -0.73), (u'python', -0.56),
(u'java', -0.56), (u'popular', -0.34), (u'ruby', -0.33), (u'excellent',
```

```
-0.33), (u'program', -0.21), (u'small', -0.11), (u'fox', 0.0), (u'dog',
0.0), (u'jump', 0.0), (u'clever', 0.0), (u'quick', 0.0), (u'lazy', 0.0),
(u'slow', 0.0), (u'smarter', 0.0), (u'cat', 0.0)]
```

从上面的输出我们看到，两个主题都有所有的词项，但更加关注权重。你可以看到有什么区别吗？当然，与编程有关主题中的词项的值为零，表明它们对该主题没有贡献。下面让我们给出适当的阈值，并且只得到每个主题的相关词：

```
# applying a scoring threshold
In [874]: topics = get_topics_terms_weights(weights, feature_names)
     ...: print_topics_udf(topics=topics,
     ...:                  total_topics=total_topics,
     ...:                  weight_threshold=0.15,
     ...:                  display_weights=True)
Topic #1 with weights
[(u'dog', 0.72), (u'fox', 0.72), (u'jump', 0.43), (u'smarter', 0.34),
(u'cat', 0.34), (u'slow', 0.23), (u'lazy', 0.23), (u'quick', 0.23),
(u'clever', 0.23)]

Topic #2 with weights
[(u'programming', -0.73), (u'language', -0.73), (u'python', -0.56),
(u'java', -0.56), (u'popular', -0.34), (u'ruby', -0.33), (u'excellent',
-0.33), (u'program', -0.21)]

 In [875]: topics = get_topics_terms_weights(weights, feature_names)
     ...: print_topics_udf(topics=topics,
     ...:                  total_topics=total_topics,
     ...:                  weight_threshold=0.15,
     ...:                  display_weights=False)
Topic #1 without weights
[u'dog', u'fox', u'jump', u'smarter', u'cat', u'slow', u'lazy', u'quick',
u'clever']

Topic #2 without weights
[u'programming', u'language', u'python', u'java', u'popular', u'ruby',
u'excellent', u'program']
```

这让我们可以更好地描述这些主题，类似于之前得到的那些主题，其中每个主题都明确地具有与另一个不同的概念。因此，你可以看到简单的矩阵计算如何帮助我们实现强大的主题模型框架！我们使用 LSI 来定义下面的函数作为通用可重用的主题建模框架：

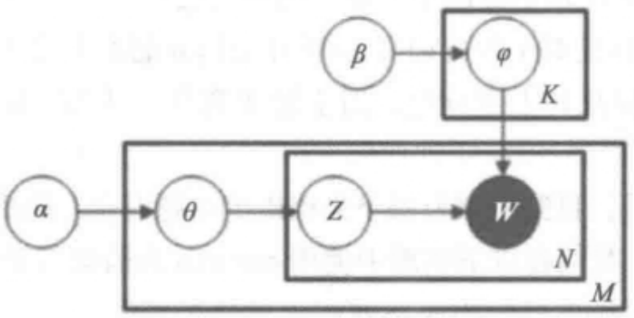
```
def train_lsi_model_gensim(corpus, total_topics=2):

    norm_tokenized_corpus = normalize_corpus(corpus, tokenize=True)
    dictionary = corpora.Dictionary(norm_tokenized_corpus)
    mapped_corpus = [dictionary.doc2bow(text)
                     for text in norm_tokenized_corpus]
    tfidf = models.TfidfModel(mapped_corpus)
    corpus_tfidf = tfidf[mapped_corpus]
    lsi = models.LsiModel(corpus_tfidf,
                          id2word=dictionary,
                          num_topics=total_topics)
    return lsi
```

稍后将使用上述函数从产品评论中提取主题。现在让我们来看下一个构建主题模型的技术——使用隐含 Dirichlet 分布。

5.6.2 隐含 Dirichlet 分布

隐含 Dirichlet 分布（Latent Dirichlet Allocation，LDA）技术是一种概率生成模型，其中假定每个文档具有类似于概率隐含语义索引模型的主题组合——但是在此情况下，隐含主

题包含它们的 Dirichlet 先验分布。这项技术背后的数学知识比较复杂，因为它的具体细节将超出当前范围，所以我们会设法总结一下。建议读者查阅 Christine Doig 在 http://chdoig.github.io/pygotham-topic-modeling/#/上发表的优秀演讲，从中我们将借鉴一些优秀的图形化表示。LDA 模型的盘子表示法（plate notation）如图 5-2 所示。



- K是主题数量
- N是文档中的单词数量
- M是待分析的文档数量
- α是每个文档主题分布的Dirichlet先验集中参数
- β是每个主题词分布的相同参数
- $\varphi(k)$是对于主题k的词分布
- $\theta(i)$是对于文档i的主题分布
- $z(i,j)$是对于$w(i,j)$的主题分配
- $w(i,j)$是第i个文档中的第j个词
- φ和θ是Dirichlet，z和w是多项式

图 5-2 LDA 的盘子表示法（由 C. Doig 提供，Python 中的主题建模简介）

图 5-3 显示了每个参数如何连接到文本文档和词的良好表示。假设我们有 M 个文档，N 个文档中的单词，以及 K 个想要生成的主题数量。

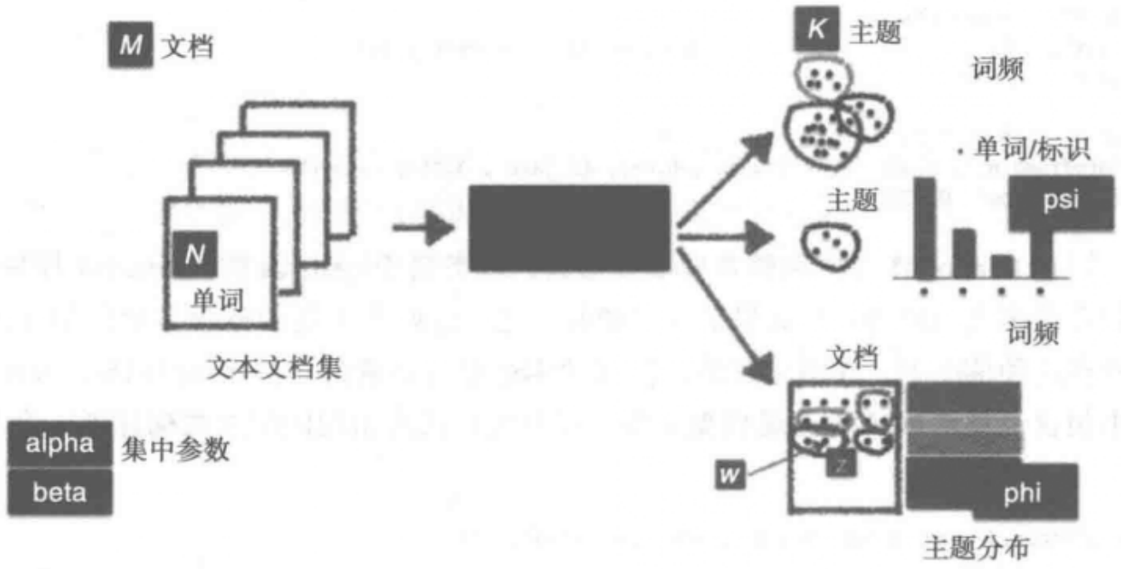


图 5-3 端到端的 LDA 框架（由 C. Doig 提供，Python 中的主题建模简介）

图中的黑盒子表示利用前面提到的参数从文档中提取 K 个主题的核心算法。为大家着想，以下步骤就算法的过程给出了一个非常简明的解释：

（1）初始化必要的参数。

（2）对于每个文档，随机将每个单词初始化为 K 个主题之一。

(3) 开始如下的一个迭代过程，重复几次。

(4) 对于每个文档 D：

a. 对于文档中的每个单词 W：

- 对于每个主题 T：
- 计算 $P(T \mid D)$，其是 D 中分配给主题 T 的词的比例。
- 计算 $P(W \mid D)$，其是对于含有词 W 的所有文档分配给主题 T 的比例。
- 考虑所有其他单词及其主题分配，用主题 T 和概率 $P(T \mid D) \times P(W \mid D)$ 重新分配词 W。

运行了几次迭代之后，我们应该为每个文档提供主题混合，然后从指向该主题的词中生成每个主题的组成部分。我们在以下实现中使用 gensim 来构建基于 LDA 的主题模型：

```
def train_lda_model_gensim(corpus, total_topics=2):

    norm_tokenized_corpus = normalize_corpus(corpus, tokenize=True)
    dictionary = corpora.Dictionary(norm_tokenized_corpus)
    mapped_corpus = [dictionary.doc2bow(text)
                     for text in norm_tokenized_corpus]
    tfidf = models.TfidfModel(mapped_corpus)
    corpus_tfidf = tfidf[mapped_corpus]
    lda = models.LdaModel(corpus_tfidf,
                          id2word=dictionary,
                          iterations=1000,
                          num_topics=total_topics)
    return lda

# use the function to generate topics on toy corpus
In [922]: lda_gensim = train_lda_model_gensim(toy_corpus,
     ...:                                     total_topics=2)
     ...:
     ...: print_topics_gensim(topic_model=lda_gensim,
     ...:                     total_topics=2,
     ...:                     num_terms=5,
     ...:                     display_weights=True)
Topic #1 with weights
[(u'fox', 0.08), (u'dog', 0.08), (u'jump', 0.07), (u'clever', 0.07),
(u'quick', 0.07)]

Topic #2 with weights
[(u'programming', 0.08), (u'language', 0.08), (u'java', 0.07), (u'python',
0.07), (u'ruby', 0.07)]
```

你可以在 LdaModel 类中调整各种模型参数，该类属于 gensim 的 ldamodel 模块。该实现对于包含许多文档的语料库效果最好。如前所述，这两个主题的概念是如何相互区分的，但是注意在这种情况下，权重是正的，使其比 LSI 更容易解读。甚至 scikit-learn 最终也在其库中包含了基于 LDA 的主题模型实现。以下代码段使用相同的方法构建了一个 LDA 主题模型：

```
from sklearn.decomposition import LatentDirichletAllocation

# get tf-idf based features
norm_corpus = normalize_corpus(toy_corpus)
vectorizer, tfidf_matrix = build_feature_matrix(norm_corpus,
                                    feature_type='tfidf')
# build LDA model
total_topics = 2
lda = LatentDirichletAllocation(n_topics=total_topics,
                                max_iter=100,
```

```
                                learning_method='online',
                                learning_offset=50.,
                                random_state=42)
lda.fit(tfidf_matrix)

# get terms and their weights
feature_names = vectorizer.get_feature_names()
weights = lda.components_

# generate topics from their terms and weights
topics = get_topics_terms_weights(weights, feature_names)
```

在该段代码中，将 LDA 模型应用于文档 - 词项的 TF-IDF 特征矩阵，其被分解成两个矩阵，即一个文档 - 主题矩阵和一个主题 - 词项矩阵。我们使用存储在 `lda.components_`中的主题 - 词项矩阵来检索每个主题每个词的权重。得到这些权重后，我们使用 LSI 建模中的 `get_topics_terms_weights()`函数根据每个主题的词项和权重来构建主题。我们现在可以使用之前实现的 `print_topics_udf()`函数查看主题：

```
In [926]: topics = get_topics_terms_weights(weights, feature_names)
     ...: print_topics_udf(topics=topics,
     ...:                  total_topics=total_topics,
     ...:                  num_terms=8,
     ...:                  display_weights=True)
Topic #1 with weights
[(u'fox', 1.86), (u'dog', 1.86), (u'jump', 1.19), (u'clever', 1.12),
(u'quick', 1.12), (u'lazy', 1.12), (u'slow', 1.12), (u'cat', 1.06)]

Topic #2 with weights
[(u'programming', 1.8), (u'language', 1.8), (u'java', 1.64), (u'python',
1.64), (u'program', 1.3), (u'ruby', 1.11), (u'excellent', 1.11),
(u'popular', 1.06)]
```

我们现在可以看到两个可区别主题的相似结果，其中第一个主题是关于前四个文档中的动物及其特征，而第二个主题是关于后四个文档中的编程语言及其属性。

5.6.3 非负矩阵分解

我们将看到的最后一项技术是非负矩阵分解（Non-Negative Matrix Factorization，NNMF），它是另一种类似于 SVD 的矩阵分解技术，虽然 NNMF 是对非负矩阵操作运算，并也可适用于多变量数据。NNMF 可以正式定义为：给定非负矩阵 V，目标是找到两个非负矩阵因子 W 和 H，使得它们相乘时，它们可以近似重构 V。数学上这表示为

$$V \approx WH$$

使得所有三个矩阵都为非负。为了实现这个近似，我们通常使用一个成本函数，如两个矩阵之间的欧几里得距离或 L2 范数，或是 L2 范数略微修改的 Frobenius 范数。这可以表示为

$$\underset{W,H}{\operatorname{argmin}} \frac{1}{2} \| V - WH \|^2$$

其中，我们有三个非负矩阵 V、W 和 H。这可以进一步简化为：

$$\frac{1}{2} \sum_{i,j} (V_{ij} - WH_{ij})^2$$

在本节使用的 `scikit-learn decomposition` 模块的 NMF 类中可获得该实现。

可以在我们的小语料库上使用以下代码段构建一个基于 NNMF 的主题模型，它给出了与 LDA 一样的特征名称和权重：

```
from sklearn.decomposition import NMF
# build tf-idf document-term matrix
norm_corpus = normalize_corpus(toy_corpus)
vectorizer, tfidf_matrix = build_feature_matrix(norm_corpus,
                                    feature_type='tfidf')
# build topic model
total_topics = 2
nmf = NMF(n_components=total_topics,
          random_state=42, alpha=.1, l1_ratio=.5)
nmf.fit(tfidf_matrix)
# get terms and their weights
feature_names = vectorizer.get_feature_names()
weights = nmf.components_
```

现在我们有了词项及其权重，可以使用我们以前定义的函数来打印主题，如下所示：

```
In [928]: topics = get_topics_terms_weights(weights, feature_names)
     ...: print_topics_udf(topics=topics,
     ...:                  total_topics=total_topics,
     ...:                  num_terms=None,
     ...:                  display_weights=True)
Topic #1 with weights
[(u'programming', 0.55), (u'language', 0.55), (u'python', 0.4), (u'java',
0.4), (u'popular', 0.24), (u'ruby', 0.23), (u'excellent', 0.23),
(u'program', 0.09), (u'small', 0.03)]

Topic #2 with weights
[(u'dog', 0.57), (u'fox', 0.57), (u'jump', 0.35), (u'smarter', 0.26),
(u'cat', 0.26), (u'quick', 0.13), (u'slow', 0.13), (u'clever', 0.13),
(u'lazy', 0.13)]
```

我们观察到的是，与其他方法相比，即使是包含较少文档的小语料库，非负矩阵分解的工作效果最好，但是，这又取决于你正在处理的数据类型。

5.6.4 从产品评论中提取主题

现在，我们将利用早期的函数，并使用三种技术对一些实际数据构建主题模型。为此，我们已经从亚马逊提取了一些特定产品的评论。数据发烧友可以从 http://jmcauley.ucsd.edu/data/amazon/上获取有关此数据源的更多信息，其中包含基于产品类型和类别的各种产品评论。我们感兴趣的产品是由 Bethesda Softworks 开发的非常流行的视频游戏“The Elder Scrolls V：Skyrim”。这或许是最好的角色扮演游戏之一。（如果你有兴趣，可以在 www.amazon.com/dp/B004HYK956 上查看该产品信息及其在亚马逊的评论。）在我们的例子中，可以在名为 amazon_skyrim_reviews.csv 的 CSV 文件中获取提取的评论，并附有本章的代码文件。在提取主题之前让我们先加载评论：

```
import pandas as pd
import numpy as np
# load reviews
CORPUS = pd.read_csv('amazon_skyrim_reviews.csv')
CORPUS = np.array(CORPUS['Reviews'])

# view sample review
In [946]: print CORPUS[12]
I base the value of a game on the amount of enjoyable gameplay I can get out
of it and this one was definitely worth the price!
```

现在我们已经加载了产品评论的语料库，我们将主题数量设置为5，并使用前面几节中实现的所有三种技术来提取主题。以下代码段实现相同效果：

```
# set number of topics
total_topics = 5

# Technique 1: Latent Semantic Indexing
In [958]: lsi_gensim = train_lsi_model_gensim(CORPUS,
     ...:                                       total_topics=total_topics)
     ...: print_topics_gensim(topic_model=lsi_gensim,
     ...:                     total_topics=total_topics,
     ...:                     num_terms=10,
     ...:                     display_weights=False)
Topic #1 without weights
[u'skyrim', u'one', u'quest', u'like', u'play', u'oblivion', u'go', u'get',
u'time', u'level']

Topic #2 without weights
[u'recommend', u'love', u'ever', u'best', u'great', u'level', u'highly',
u'play', u'elder', u'scroll']

Topic #3 without weights
[u'recommend', u'highly', u'fun', u'love', u'ever', u'wonderful', u'best',
u'everyone', u'series', u'scroll']

Topic #4 without weights
[u'fun', u'scroll', u'elder', u'recommend', u'highly', u'wonderful', u'fan',
u'graphic', u'series', u'cool']

Topic #5 without weights
[u'fun', u'love', u'elder', u'scroll', u'highly', u'5', u'dont', u'hour',
u'series', u'hundred']

# Technique 2a: Latent Dirichlet Allocation (gensim)
In [959]: lda_gensim = train_lda_model_gensim(CORPUS,
     ...:                                       total_topics=total_topics)
     ...: print_topics_gensim(topic_model=lda_gensim,
     ...:                     total_topics=total_topics,
     ...:                     num_terms=10,
     ...:                     display_weights=False)
Topic #1 without weights
[u'quest', u'good', u'skyrim', u'love', u'make', u'best', u'time', u'go',
u'play', u'every']

Topic #2 without weights
[u'good', u'play', u'get', u'really', u'like', u'one', u'hour', u'buy',
u'go', u'skyrim']

Topic #3 without weights
[u'fun', u'gameplay', u'skyrim', u'best', u'want', u'time', u'one', u'play',
u'review', u'like']

Topic #4 without weights
[u'love', u'play', u'one', u'much', u'great', u'ever', u'like', u'fun',
u'recommend', u'level']

Topic #5 without weights
[u'great', u'long', u'love', u'scroll', u'elder', u'oblivion', u'play',
u'month', u'never', u'skyrim']

# Technique 2b: Latent Dirichlet Allocation (scikit-learn)
In [960]: norm_corpus = normalize_corpus(CORPUS)
     ...: vectorizer, tfidf_matrix = build_feature_matrix(norm_corpus,
     ...:                                                 feature_type='tfidf')
     ...: feature_names = vectorizer.get_feature_names()
     ...:
     ...:
     ...: lda = LatentDirichletAllocation(n_topics=total_topics,
     ...:                                 max_iter=100,
     ...:                                 learning_method='online',
     ...:                                 learning_offset=50.,
```

```
     ...:                                  random_state=42)
     ...: lda.fit(tfidf_matrix)
     ...: weights = lda.components_
     ...: topics = get_topics_terms_weights(weights, feature_names)
     ...: print_topics_udf(topics=topics,
     ...:                  total_topics=total_topics,
     ...:                  num_terms=10,
     ...:                  display_weights=False)
Topic #1 without weights
[u'statrs', u'expression', u'demand', u'unnecessary', u'mining', u'12yr',
u'able', u'snowy', u'shopkeepers', u'arpg']

Topic #2 without weights
[u'game', u'play', u'get', u'one', u'skyrim', u'great', u'like', u'time',
u'quest', u'much']

Topic #3 without weights
[u'de', u'pagar', u'cr\xe9dito', u'momento', u'responsabilidad', u'compras',
u'para', u'futuras', u'recomiendo', u'skyrimseguridad']

Topic #4 without weights
[u'booklet', u'proudly', u'ending', u'destiny', u'estatic', u'humungous',
u'chirstmas', u'bloodthey', u'accolade', u'scaled']

Topic #5 without weights
[u'game', u'play', u'fun', u'good', u'buy', u'one', u'whatnot', u'titles',
u'haveseen', u'best']

# Technique 3: Non-negative Matrix Factorization
In [961]: nmf = NMF(n_components=total_topics,
     ...:           random_state=42, alpha=.1, l1_ratio=.5)
     ...: nmf.fit(tfidf_matrix)
     ...:
     ...: feature_names = vectorizer.get_feature_names()
     ...: weights = nmf.components_
     ...:
     ...: topics = get_topics_terms_weights(weights, feature_names)
     ...: print_topics_udf(topics=topics,
     ...:                  total_topics=total_topics,
     ...:                  num_terms=10,
     ...:                  display_weights=False)
Topic #1 without weights
[u'game', u'get', u'skyrim', u'play', u'time', u'like', u'quest', u'one',
u'go', u'much']

Topic #2 without weights
[u'game', u'best', u'ever', u'fun', u'play', u'hour', u'great', u'rpg',
u'definitely', u'one']

Topic #3 without weights
[u'write', u'review', u'describe', u'justice', u'word', u'game', u'simply',
u'try', u'period', u'really']

Topic #4 without weights
[u'scroll', u'elder', u'series', u'always', u'love', u'pass', u'buy',
u'franchise', u'game', u'best']

Topic #5 without weights
[u'recommend', u'love', u'game', u'highly', u'great', u'play', u'wonderful',
u'like', u'oblivion', u'would']
```

上面的输出显示了每种技术的五个主题。如果你仔细观察它们，会注意到，主题之间总是会有一些重叠，但是它们呈现了评论与概念的区别。我们可以得出几点观察结果：

- 所有主题建模技术提出的概念会与使用形容词来描述该游戏的人有关，如 wonderful、great 和 highly recommendable。

- 他们还将游戏的类型描述为 RPG（Role-Playing Game，角色扮演游戏）或 ARPG（Action Role-Playing Game，动作角色扮演游戏）。
- 游戏特征如玩法（gameplay）和画面（graphics），会与正面的词语相关联，如 good、great、fun 和 cool。
- 词“oblivion”出现在许多主题模型中。这个可以参考 Elder Scrolls series 之前的游戏版本，称为“The Elder Scrolls IV：Oblivion”。它表明客户在评论中将该游戏与上一版进行比较。

请继续使用这些函数和数据。你甚至可以尝试在新的数据源上建立主题模型。请记住，主题建模通常是作为更深层次挖掘数据的工作起点，它通过查询特定主题概念，或者甚至聚类和分组文本文档并分析其相似性以发掘模式。

5.7 自动文档摘要

在本章开头，我们简要介绍了文档摘要，试图从大型文档或语料库中提取精要，使其保留语料库的核心精华或要义。文档摘要的思路与关键短语提取或主题建模会有些不同。其最终的结果仍然是某种文档形式，但它可能是我们想要的那种基于摘要长度的几个句子。这类似于有一篇含有摘要或执行摘要的研究论文。自动化文档摘要的主要目标是不包括人工输入的执行此摘要，除了运行任何计算机程序。数学和统计模型有助于通过观察其内容和上下文来构建和自动化概括文档的任务。

应用自动化技术进行文档摘要主要有两大类做法。

- 基于提取的技术（extraction-based technique）：这些方法使用数学和统计学概念（如 SVD）从原始文档中提取内容的一些关键子集，使得该内容子集包含核心信息，并作为整个文档的重点。这个内容可以是单词、短语或句子。这种方法的最终结果是从原始文档中采集或提取了几行简短的执行摘要。在这种技术中不产生新的内容——因此，这个名称是基于提取的。
- 基于概括的技术（abstraction-based technique）：这些方法更加复杂和精准，并利用语言语义来产生表示。它们还利用 NLG 技术，其中机器使用知识库和语义表达来自己生成文本，并像人类编写一样来创建摘要。

目前，大多数研究只是基于提取的技术，因为构建基于概括的摘要生成器比较难。但是在这方面已经取得了一些进展，就是创建模仿人类的概括型概要。我们通过利用 gensim 的摘要模块来看看文档摘要的实现。我们将使用维基百科关于大象的描述作为我们将测试所有摘要技术的文档。首先加载必要的依赖项和语料库，如下所示：

```
from normalization import normalize_corpus, parse_document
from utils import build_feature_matrix, low_rank_svd
import numpy as np

toy_text = """
Elephants are large mammals of the family Elephantidae
and the order Proboscidea. Two species are traditionally recognised,
the African elephant and the Asian elephant. Elephants are scattered
throughout sub-Saharan Africa, South Asia, and Southeast Asia. Male
African elephants are the largest extant terrestrial animals. All
elephants have a long trunk used for many purposes,
```

```
particularly breathing, lifting water and grasping objects. Their
incisors grow into tusks, which can serve as weapons and as tools
for moving objects and digging. Elephants' large ear flaps help
to control their body temperature. Their pillar-like legs can
carry their great weight. African elephants have larger ears
and concave backs while Asian elephants have smaller ears
and convex or level backs.
"""
```

我们现在定义一个函数将输入文档总结到其原始大小的一小部分，这将作为下面函数中的用户输入参数 summary_ratio。输出将是摘要后的文件：

```
from gensim.summarization import summarize, keywords

def text_summarization_gensim(text, summary_ratio=0.5):

    summary = summarize(text, split=True, ratio=summary_ratio)
    for sentence in summary:
        print sentence
```

现在我们将解析输入文档以删除换行符并提取句子，然后将整个文档传递给前面的函数，其中 gensim 负责处理规范化，并对该文档进行汇总，如下面的代码段所示：

```
In [978]: docs = parse_document(toy_text)
     ...: text = ' '.join(docs)
     ...: text_summarization_gensim(text, summary_ratio=0.4)
Two species are traditionally recognised,  the African elephant and the
Asian elephant.
All  elephants have a long trunk used for many purposes,  particularly
breathing, lifting water and grasping objects.
African elephants have larger ears  and concave backs while Asian elephants
have smaller ears  and convex or level backs.
```

如果观察上述输出并与原始文档进行比较，在原始文档中共有 9 个句子，而在总结后总共有 3 个句子。但是，如果你阅读了总结文档，将看到文档的核心意义和主题已被保留，其中包括两种大象，它们如何区分彼此，以及它们的共同特征。这个源自 gensim 的摘要实现是基于一种流行的称为 TextRank 的算法。

现在我们已经了解了非常有趣的文本摘要，让我们来看看几个基于提取的摘要算法。我们将主要关注以下两种技术：

- 隐含语义分析。
- TextRank。

我们首先将探讨每种技术背后的概念和数学表达，然后使用 Python 实现。最后，我们将对之前的小文档进行测试。在深入了解技术之前，让我们解析和规范化我们的小文档，如下所示：

```
# parse and normalize document
sentences = parse_document(toy_text)
norm_sentences = normalize_corpus(sentences,lemmatize=True)
# check total sentences in document
In [992]: total_sentences = len(norm_sentences)
     ...: print 'Total Sentences in Document:', total_sentences
Total Sentences in Document: 9
```

一旦有了一个可运用的摘要算法，我们将为每种技术构建一个通用函数，并在下一节中对维基百科的真实产品描述进行测试。

5.7.1 隐含语义分析

在这里，我们将要了解如何总结文本文档，这可通过将 SVD 应用在文档句子、文档每个句子中的词，以及使用某种特征权重（如词袋或 TF-IDF 权重）上来实现。隐含语义分析（LSA）的核心原则是，在任何文件中，在词语的相关语境中存在隐含的结构，因此也应该在相同的奇异空间中相关。我们在实现过程中采用的方法来自于 J. Steinberger 和 K. Jezek 在 2004 年发表的热门论文“Using latent semantic analysis in text summarization and summary evaluation”，其中他们基于 Y. Gong 和 X. Liu 在 2001 年出版的“Generic Text Summarization Using Relevance Measure and Latent Semantic Analysis”中所做的优秀工作之上提出了一些改进。如果你有兴趣了解这项技术，我们建议你阅读这两篇论文。

我们实现的主要思想是使用 SVD，如果你记得 SVD 中的方程式，其中 $M = USV^T$，使得 U 和 V 是正交矩阵，S 是对角矩阵，其也可以表示为奇异值向量。原始矩阵可以表示为词项 – 文档矩阵，其中行将是词，每 1 列将是 1 个文档，也就是说，在这种情况下是我们文档中的 1 个句子。这些值可以是任何类型的加权，例如基于词袋模型的频率、TF-IDF 或出现次数二值特征。我们将使用 `low_rank_svd()` 函数根据概念数量 k 创建 M 的低秩矩阵近似，k 将是奇异值的数量。来自矩阵 U 的相同的 k 列将指向 k 个概念中的每一个词向量，并且对于矩阵 V，基于前 k 个奇异值的 k 行指向句子向量。从基于概念数量 k 的前 k 个奇异值的 SVD 中得到 U、S 和 V^T之后，我们执行以下计算。请记住，需要的输入参数是我们预期最终摘要包含的概念数量 k 和句子数 n。

- 从矩阵 V（k 行）获取句子向量。
- 从 S 获得前 k 个奇异值。
- 应用基于阈值的方法，删除小于最大奇异值一半的奇异值（如果有的话）。这是启发式的，你可以按照需要调整这个值。在数学上，$S_i = 0$ iff $S_i < \frac{1}{2}\max(S)$。
- 将来自 V 平方的每个词句子列乘以 S 平方相对应的奇异值，以获得每个主题的句子权重。
- 计算主题之间的句子权重之和，并取最终分数的平方根来获得文档中每个句子的凸显度分数。

每个句子前面的凸显度分数的计算可以在数学上表示为

$$SS = \sqrt{\sum_{i=1}^{k} S_i V_i^T}$$

其中 SS 表示每个句子的凸显度分数，其通过采用奇异值和 V^T句子向量之间的点积获得。得到这些分数后，按降序对它们进行排序，选择与最高分数相对应的前 n 个句子，并根据它们在原始文档中出现的顺序将它们组合起来形成最终的摘要。让我们使用以下代码段在代码中实现上述步骤：

```
# set the number of sentences and topics for summarized document
num_sentences = 3
num_topics = 3

# build document term matrix based on bag of words features
vec, dt_matrix = build_feature_matrix(sentences,
                                      feature_type='frequency')
```

```
# convert to term document matrix
td_matrix = dt_matrix.transpose()
td_matrix = td_matrix.multiply(td_matrix > 0)

# get low rank SVD components
u, s, vt = low_rank_svd(td_matrix, singular_count=num_topics)

# remove singular values below threshold
sv_threshold = 0.5
min_sigma_value = max(s) * sv_threshold
s[s < min_sigma_value] = 0

# compute salience scores for all sentences in document
salience_scores = np.sqrt(np.dot(np.square(s), np.square(vt)))

# print salience score for each sentence
In [996]: print np.round(salience_scores, 2)
[ 2.93  3.28  1.67  1.8   2.24  4.51  0.71  1.22  5.24]

# rank sentences based on their salience scores
top_sentence_indices = salience_scores.argsort()[-num_sentences:][::-1]
top_sentence_indices.sort()
# view top sentence index positions
In [997]: print top_sentence_indices
[1 5 8]

# get document summary by combining above sentences
In [998]: for index in top_sentence_indices:
     ...:     print sentences[index]
Two species are traditionally recognised,  the African elephant and the
Asian elephant.
Their  incisors grow into tusks, which can serve as weapons and as
tools  for moving objects and digging.
African elephants have larger ears  and concave backs while Asian elephants
have smaller ears  and convex or level backs.
```

可以看到一些矩阵操作为我们提供了一个简明、优秀的总结文档，涵盖了大象文档中的主要主题。将其与之前使用 gensim 生成的进行比较。你可以在这些概要之间看到一些相似之处吗？

我们现在将使用之前的算法为 LSA 构建一个通用的可重用函数，以便我们可以在后续产品描述文档中使用它，你也可以对自己的数据使用此函数：

```
def lsa_text_summarizer(documents, num_sentences=2,
                        num_topics=2, feature_type='frequency',
                        sv_threshold=0.5):

    vec, dt_matrix = build_feature_matrix(documents,
                                          feature_type=feature_type)

    td_matrix = dt_matrix.transpose()
    td_matrix = td_matrix.multiply(td_matrix > 0)

    u, s, vt = low_rank_svd(td_matrix, singular_count=num_topics)
    min_sigma_value = max(s) * sv_threshold
    s[s < min_sigma_value] = 0

    salience_scores = np.sqrt(np.dot(np.square(s), np.square(vt)))
    top_sentence_indices = salience_scores.argsort()[-num_sentences:][::-1]
    top_sentence_indices.sort()

    for index in top_sentence_indices:
        print sentences[index]
```

至此，我们完成了 LSA 的介绍，接下来我们将介绍下一项基于提取的文档摘要技术。

5.7.2 TextRank 算法

TextRank 摘要算法内部使用了热门的 PageRank 算法，PageRank 算法被谷歌用于网站和网页排名，并评估它们的重要性。当谷歌搜索引擎根据搜索查询提供相关网页时将使用它。为了更好地理解 TextRank，我们需要了解 PageRank 有关的一些概念。

PageRank 中的核心算法是一种基于图的评分或排名算法，其中页面根据其重要性进行评分或排名。网站和网页包含了嵌入其中的进一步链接，它们可以链接到含有更多链接的更多页面，这可以在互联网上不断延续。这可以表达为基于图的模型，其中顶点表示网页，边表示它们之间的链接。这点可以用于形成投票或推荐系统，使得当图中一个顶点链接到另一个顶点时，它相当于投了一票。顶点重要性不仅取决于投票数或边数，还取决于与其相连顶点的重要性以及它们的重要性。这有助于确定每个顶点或页面的分数或等级。从图 5-4 中可以看出，它显示了具有重要性的页面示例。

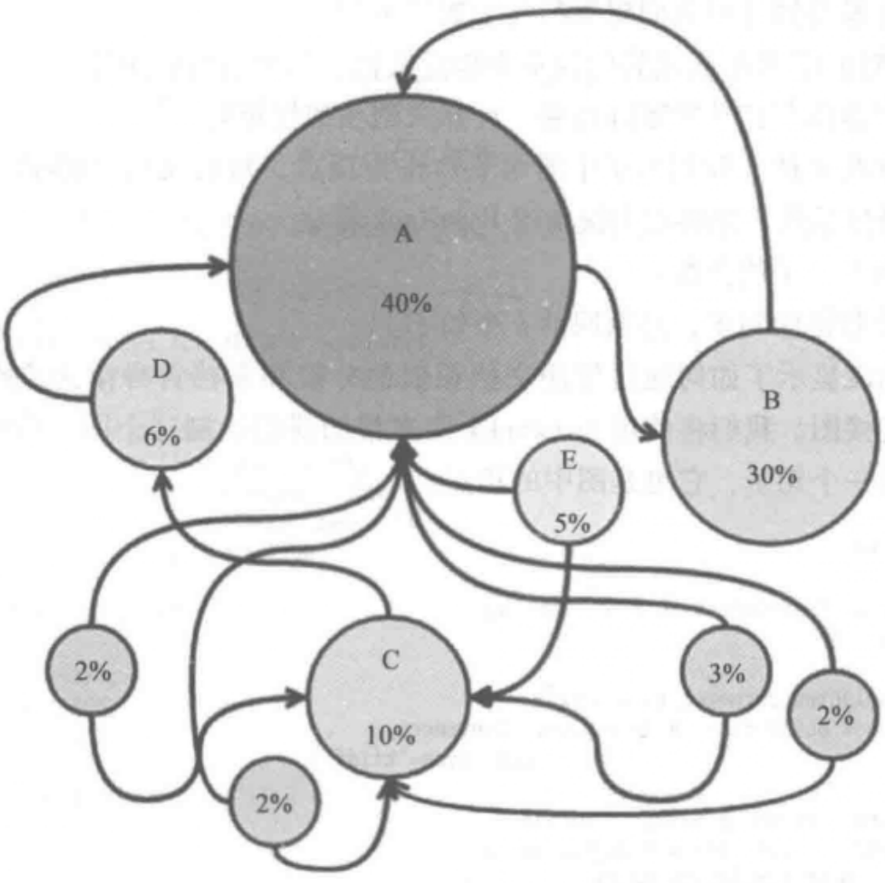


图 5-4 一个简单网络的 PageRank 分数

在图 5-4 中，我们可以看到，表示页面 B 的顶点的分数要高于页面 C，即使它相对于页面 C 有较少的边，这是因为页面 A 是重要页面，而其与页面 B 是相连的。因此，我们现在可以正式定义 PageRank 如下。

给定一个有向图，其表示为 $G=(V,E)$，使得 V 表示为顶点或页面的集合，E 表示为边或链接的集合，E 是 $V\times V$ 的子集。假设我们有一个给定的页面 V_i，想要计算 PageRank，我们可以在数学上将其定义为

$$PR(V_i) = (1-d) + d \times \sum_{j\in \ln(V_i)} \frac{PR(V_j)}{|\mathrm{Out}(V_j)|}$$

其中，对于顶点/页面 V_i，我们有表示 PageRank 分数的 $PR(V_i)$，$\text{In}(V_i)$ 表示指向该顶点/页面的页面集合，$\text{Out}(V_i)$ 表示顶点/页面 V_i 指向的页面集合，d 是通常值在 0 至 1 之间的阻尼因子——理想情况下，设置其为 0.85。

回到 TextRank 算法，当我们总结文档时，将根据我们尝试建立的摘要类型，将句子、关键词或短语作为算法的顶点。我们可能在这些顶点之间有多个链接，并且我们从原始 PageRank 算法进行的修改是在连接两个顶点 V_i 和 V_j 的边之间有一个权重系数 W_{ij}，该权重表示它们之间这种连接关系的强度。这样，我们现在正式定义了用于计算顶点 TextRank 的新函数

$$TR(V_i) = (1 - d) + d \times \sum_{V_j \in \text{In}(V_i)} \frac{w_{ji} TR(V_j)}{\sum_{V_k \in \text{Out}(V_j)} w_{jk}}$$

其中，TR 表示顶点的加权 PageRank 得分，现在其定义为该顶点的 TextRank。因此，现在可以规划算法，并确定我们将要遵循的主要步骤：

（1）从待总结的文档中标记和提取句子。

（2）确定在最终摘要中我们想要的句子数量 k。

（3）使用诸如 TF-IDF 或词袋的权重来构建文档 - 词项的特征矩阵。

（4）通过将矩阵与其转置矩阵相乘，计算文档相似性矩阵。

（5）使用这些文档（我们例子中的句子）作为顶点，每对文档之间的相似性作为前面提到的权重或得分系数，并将它们提供给 PageRank 算法。

（6）获得每个句子的分数。

（7）根据分数排序句子，并返回前 k 个句子。

下面的代码段显示了如何通过使用文档相似性分数和文档自身作为顶点来构建小文档中所有句子的连接图。我们将使用 networkx 库来帮助我们绘制这个图。请记住，每个文档是我们例子中的一个句子，它也是图中的顶点：

```
import networkx

# define number of sentences in final summary
num_sentences = 3

# construct weighted document term matrix
vec, dt_matrix = build_feature_matrix(norm_sentences,
                                      feature_type='tfidf')

# construct the document similarity matrix
similarity_matrix = (dt_matrix * dt_matrix.T)
# view the document similarity matrix
In [1011]: print np.round(similarity_matrix.todense(), 2)
[[ 1.    0.    0.03  0.05  0.03  0.    0.15  0.    0.06]
 [ 0.    1.    0.    0.07  0.    0.    0.    0.    0.11]
 [ 0.03  0.    1.    0.03  0.02  0.    0.03  0.    0.04]
 [ 0.05  0.07  0.03  1.    0.03  0.    0.04  0.    0.11]
 [ 0.03  0.    0.02  0.03  1.    0.07  0.03  0.    0.04]
 [ 0.    0.    0.    0.    0.07  1.    0.    0.    0.  ]
 [ 0.15  0.    0.03  0.04  0.03  0.    1.    0.    0.05]
 [ 0.    0.    0.    0.    0.    0.    0.    1.    0.  ]
 [ 0.06  0.11  0.04  0.11  0.04  0.    0.05  0.    1.  ]]

# build the similarity graph
similarity_graph = networkx.from_scipy_sparse_matrix(similarity_matrix)
# view the similarity graph
In [1013]: networkx.draw_networkx(similarity_graph)
Out [1013]:
```

在图 5-5 中，我们可以看到小文档中的句子现在如何根据文档的相似性相互链接。该图展示了一些句子如何与其他句子进行连接。

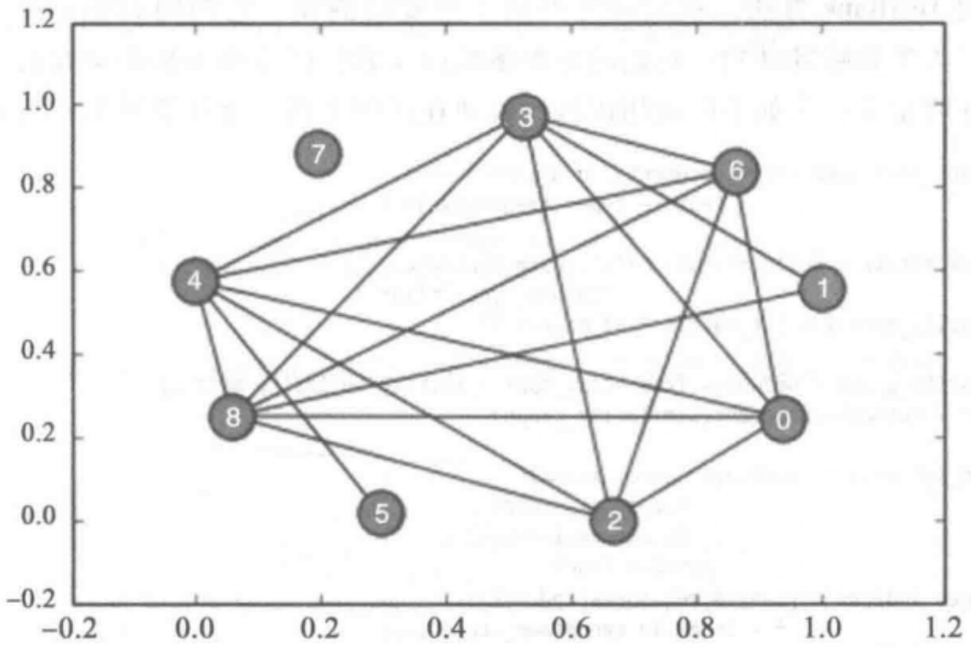


图 5-5 显示句子之间连接关系的相似图

现在，我们将计算所有句子的 PageRank 得分，对它们进行排名，并使用前三个句子构建摘要：

```
# compute pagerank scores for all the sentences
scores = networkx.pagerank(similarity_graph)

# rank sentences based on their scores
ranked_sentences = sorted(((score, index)
                           for index, score
                           in scores.items()),
                          reverse=True)
# view the ranked sentences
In [1030]: ranked_sentences
Out[1030]:
[(0.11889477617125277, 8),
 (0.11456045476451866, 3),
 (0.11285293843138654, 0),
 (0.11210156056437962, 6),
 (0.11139550507847462, 4),
 (0.1111111111111111, 7),
 (0.10709498606197024, 5),
 (0.10610242758495998, 2),
 (0.10588624023194664, 1)]

# get the top sentence indices for our summary
top_sentence_indices = [ranked_sentences[index][1]
                        for index in range(num_sentences)]
top_sentence_indices.sort()

# view the top sentence indices
In [1032]: print top_sentence_indices
 [0, 3, 8]

# construct the document summary
In [1033]: for index in top_sentence_indices:
      ...:     print sentences[index]
Elephants are large mammals of the family Elephantidae  and the order
Proboscidea.
```

```
Male  African elephants are the largest extant terrestrial animals.
African elephants have larger ears  and concave backs while Asian elephants
have smaller ears  and convex or level backs.
```

通过使用 TextRank 算法，我们最终得到了想要的摘要。它的内容也是非常有意义的，其中它讲述了大象是哺乳动物、它们的分类系统以及如何区分亚洲和非洲大象。

我们现在将定义一个如下的通用函数，以便在任何文档上计算基于 TextRank 的摘要：

```
def textrank_text_summarizer(documents, num_sentences=2,
                             feature_type='frequency'):

    vec, dt_matrix = build_feature_matrix(norm_sentences,
                                          feature_type='tfidf')
    similarity_matrix = (dt_matrix * dt_matrix.T)

    similarity_graph = networkx.from_scipy_sparse_matrix(similarity_matrix)
    scores = networkx.pagerank(similarity_graph)

    ranked_sentences = sorted(((score, index)
                               for index, score
                               in scores.items()),
                              reverse=True)
top_sentence_indices = [ranked_sentences[index][1]
                        for index in range(num_sentences)]
top_sentence_indices.sort()

for index in top_sentence_indices:
    print sentences[index]
```

我们已经介绍了两种文档摘要的技术，并且还构建了通用的可重用函数来计算任意文本文档的自动文档摘要。在下一节中，我们将从一个维基页面来生成产品说明摘要。

5.7.3 生成产品说明摘要

基于5.6.4节中谈到的内容，我们在这里将总结相同产品的描述——一个名为“The Elder Scrolls V：Skyrim”的角色扮演视频游戏。我们从维基百科页面收集了几行包含该产品的详细说明。在本节中，我们将利用上一节的函数，对该产品描述执行自动化文档摘要。我们将从加载产品说明和内容规范化开始：

```
# load the document
DOCUMENT = """
The Elder Scrolls V: Skyrim is an open world action role-playing video game
developed by Bethesda Game Studios and published by Bethesda Softworks.
It is the fifth installment in The Elder Scrolls series, following
The Elder Scrolls IV: Oblivion. Skyrim's main story revolves around
the player character and their effort to defeat Alduin the World-Eater,
a dragon who is prophesied to destroy the world.
The game is set two hundred years after the events of Oblivion
and takes place in the fictional province of Skyrim. The player completes
quests
and develops the character by improving skills.
Skyrim continues the open world tradition of its predecessors by allowing the
player to travel anywhere in the game world at any time, and to
ignore or postpone the main storyline indefinitely. The player may freely roam
over the land of Skyrim, which is an open world environment consisting
of wilderness expanses, dungeons, cities, towns, fortresses and villages.
Players may navigate the game world more quickly by riding horses,
or by utilizing a fast-travel system which allows them to warp to previously
Players have the option to develop their character. At the beginning of the game,
players create their character by selecting one of several races,
including humans, orcs, elves and anthropomorphic cat or lizard-like
creatures,
```

```
and then customizing their character's appearance.discovered locations. Over the
course of the game, players improve their character's skills, which are
numerical
representations of their ability in certain areas. There are eighteen skills
divided evenly among the three schools of combat, magic, and stealth.
Skyrim is the first entry in The Elder Scrolls to include Dragons in the game's
wilderness. Like other creatures, Dragons are generated randomly in the world
and will engage in combat.
"""

# normalize the document
In [1045]: sentences = parse_document(DOCUMENT)
      ...: norm_sentences = normalize_corpus(sentences,lemmatize=True)
      ...: print "Total Sentences:", len(norm_sentences)
Total Sentences: 13
```

我们可以看到该说明书总共有 13 个句子。现在，在下面的代码段中使用我们的函数来生成文档摘要：

```
# LSA document summarization
In [1053]: lsa_text_summarizer(norm_sentences, num_sentences=3,
      ...:                     num_topics=5, feature_type='frequency',
      ...:                     sv_threshold=0.5)
The Elder Scrolls V: Skyrim is an open world action role-playing video
game  developed by Bethesda Game Studios and published by Bethesda
Softworks.
Players may navigate the game world more quickly by riding horses,  or
by utilizing a fast-travel system which allows them to warp to
previously  Players have the option to develop their character.
At the beginning of the game,  players create their character by selecting
one of several races,  including humans, orcs, elves and anthropomorphic
cat or lizard-like creatures,  and then customizing their character's
appearance.discovered locations.

# TextRank document summarization
In [1054]: textrank_text_summarizer(norm_sentences, num_sentences=3,
      ...:                          feature_type='tfidf')
The Elder Scrolls V: Skyrim is an open world action role-playing video
game  developed by Bethesda Game Studios and published by Bethesda
Softworks.
Players may navigate the game world more quickly by riding horses,  or
by utilizing a fast-travel system which allows them to warp to
previously  Players have the option to develop their character.
Skyrim is the first entry in The Elder Scrolls to include Dragons in the
game's  wilderness.
```

从上述输出可以看出，我们能够成功地将产品描述从 13 行总结缩略到 3 行，而这个简短摘要描述了产品描述的核心内容，比如游戏的名称以及与玩法和人物有关的各种特征。

至此，我们就结束了自动文本摘要的介绍。我们鼓励你对更多文档尝试这些技术，并去测试各种不同的参数，例如更多的主题数量，不同的特征类型，如 TF-IDF、词袋、出现次数二值特征，甚至词向量。

5.8 小结

在本章中，我们介绍了 NLP 和文本分析中与信息提取、文档摘要和主题建模有关的一些有趣的领域。我们首先概述了信息的发展，并让大家了解了信息过载等概念，正是信息过载促发了对文本摘要和信息检索的需要。我们探讨了从文本数据中提取关键信息的各种方法以及大型文档生成文摘的方式。本章涵盖了重要的数学概念，如 SVD 和低秩矩阵近似，

并在我们的几个算法中得到应用。我们主要介绍了减少信息过载的三种方法，包括关键短语提取、主题模型和自动文档摘要。关键短语提取包括一些方法，例如搭配和基于加权标签词的方法，用于从语料库获取关键短语或词。我们建立了几种主题建模技术，包括隐含语义索引、隐含 Dirichlet 分布和最新引入的非负矩阵分解。最后，我们研究了两种基于提取的自动文档摘要技术：LSA 和 TextRank。我们编程实现了每种方法并观察实际数据执行的结果，从而深入了解这些方法的工作原理，以及简单的数学运算如何能够有效地产生可操作的洞见。

第6章
文本相似度和聚类

前面的章节介绍了一些用于分析文本和提取有趣洞见的技术。前面研究了有监督的机器学习（ML）技术，它用于将文本文档分类为几种预先假定的类别。此外，还介绍了无监督的机器学习技术，如主题模型和文档摘要，它们尝试从大型文本文档和语料库中提取和检索关键主题和信息。本章将讨论其他一些利用无监督学习和信息检索概念的技术及其用例。

如果你还记得第4章的内容，就会明白文本分类其实是一个十分有趣的问题，它有几个应用场景，最常见的是对新闻文章和电子邮件分类。但是文本分类中的一个限制是我们需要手动标记一些数据作为训练数据，因为我们使用有监督的学习算法构建分类模型。构建这样一个数据集绝对不容易，因为要构建一个好的模型，需要大量的训练数据。为此，需要花时间和精力来标注数据，构建模型，然后最终使用它们来分类新的文档。可以让机器代替我们实现这些吗？当然，事实上，这样是可行的。本章专门讨论文本文档的内容，使用各种技术手段分析其相似度，并将相似文档聚类在一起。

文本数据是非结构化的和高噪声的。在执行文本分类时，拥有标记合理的训练数据和有监督学习大有裨益。但是，文档聚类是一个无监督的学习过程，我们将尝试通过让机器学习各种各样的文本文档及其特征、相似度以及它们之间的差异，来将文档分割和分类为单独的类别。这使得文档聚类更具挑战性，也更有意思。考虑一个涉及各种不同的概念和想法的文档语料库。人类以这样的方式将它们联系在一起，即使用过去学习的各种知识，并应用它们来区分不同的文档。例如，相对于句子“Python is an excellent programming language（Python是一种优秀的编程语言）”，句子“The fox is smarter than the dog（狐狸比狗更聪明）”和“The fox is faster than the dog（狐狸比狗要快）”更为相似。可以轻松并直观地找出Python、fox（狐狸）、dog（狗）、programming（编程）等特定的关键短语，这有助于确定哪些句子或文档更相似。那么，可以通过编程来实现以上过程吗？本章将重点介绍与文本相似度、距离度量和无监督ML算法相关的几个概念，以回答以下问题：

- 如何衡量文档之间的相似度？
- 如何使用距离测量值来找出最相关的文档？
- 什么时候距离测量值称为度量？
- 如何聚类或组合类似的文档？
- 可以可视化文档聚类吗？

尽管聚焦于回答以上问题，还是要先介绍解决这些问题所需的各种技巧的基本概念和信息。我们还将使用一些实例来说明与文本相似度、距离度量和文档聚类相关的概念。此

外，在这些技术中，许多都可以与我们以前学习的一些技术相结合，反之亦然。例如，采用距离度量的文本相似度概念也可以用于构建文档聚类。还可以使用主题模型中的特征来衡量文本相似度。此外，聚类通常是培养你对于数据可能的分组或分类的，甚至是可视化聚类的一个很好的切入点。聚类可以插入其他系统（如有监督的分类系统）中，还可以将几种技术组合在一起，并构建加权分类器。这些可能性是无止境的。

本章将首先介绍与距离测量值、度量和无监督学习相关的一些重要概念，并对文本规范化和特征提取进行梳理。介绍完基础知识之后，我们的目标将是理解和分析词项相似度、文档相似度，并最终完成文档聚类。

6.1 重要概念

本章的主要目的是了解文本相似度和聚类。在介绍实际的技术和算法之前，本节将讨论与信息检索、文档相似度度量和机器学习相关的一些重要概念。虽然这些概念中的部分可能在前几章中你已经比较熟悉，但是随着我们逐渐深入本章，所有这些概念对我们都是有用的。

6.1.1 信息检索

信息检索（Information Retrieval，IR）是根据某些需求从存储信息的语料库或实体中检索或获取相关信息源的过程。例如，它可以是用户在搜索引擎中输入查询或搜索，然后获取与其查询相关的搜索项的过程。实际上，搜索引擎是 IR 最受欢迎的应用。

文档和信息与用户需求的相关性可以通过几种方式进行衡量。它包括从搜索文本中查找特定关键字，或使用一些相似度度量来查看文档与输入查询的相似度排名或得分。这与字符串匹配或正则表达式匹配完全不同，因为相较于文档（实体）集合中的字符串，搜索字符串中的单词常常具有不同的顺序、上下文和语义，而这些单词的含义基于同义词、反义词和否定修饰符可以有多种可能的结果。

6.1.2 特征工程

到目前为止你已经熟悉了*特征工程*或*特征提取*知识。词袋、TF-IDF 和词向量化模型等技术通常用于以数值向量的形式表示文档或对文档建模，以便于更加方便地应用数学或机器学习技术。通过这些特征提取技术，可以得出各种文档的数字表示，甚至可以将每个字母或单词映射到与之相应的唯一数字标识符。

6.1.3 相似度测量

在文本相似度分析和聚类中经常使用*相似度测量*。相似度或距离测量值通常是用来衡量两个实体之间的接近程度的，其中实体可以是任何文本形式，例如文档、句子甚至是短语。这种相似度测量在识别类似实体并将不同实体加以区分时十分有用。相似度测量是非常有效的，有时选择正确的度量方式可能会对最终分析系统的性能产生很大的影响。基于距离测量，人们还发明了各种评分或排名算法。实体之间的相似程度由两个主要因素决定：

- 实体的固有属性或特征。
- 测量公式及其特性。

后面几节将介绍一些衡量相似度的距离测量值。请记住一个重要的知识点，那就是不是所有的距离测量值都是相似度的距离度量（distance metric）。A. Huang 在一篇优秀的论文“Similarity Measures for Text Document Clustering（文本文档聚类的相似度测量）”中详细介绍了这一点。我们可以考虑距离测量值 d 和两个实体（如文档）x 和 y。x 和 y 之间的距离用于确定它们之间的相似程度，可以表示为 $d(x,y)$，当且仅当满足以下四个条件时，测量值 d 才可以称为一个相似度的距离度量：

（1）任何两个实体之间的测量距离（如 x 和 y 的距离）必须始终为非负数，即 $d(x,y) \geqslant 0$。

（2）当且仅当两个实体相同时，距离为零，即 $d(x,y) = 0$ iff $x = y$。

（3）这个距离测量值应该是对称的，这意味着从 x 到 y 的距离总是和从 y 到 x 的距离相同，数学上表示为 $d(x,y) = d(y,x)$。

（4）该距离测量值应满足三角不等式特性，在数学上可以表示为 $d(x,z) \leqslant d(x,y) + d(y,z)$。

以上条件是重要的衡量标准，也是一个良好的框架，可以用它来检查距离测量方法是否可以用作测量相似度的距离度量。在本书中，没有更多的篇章可以对它进行详细的阐述。你也许会想要了解流行的 KL 散度测量（KL-divergence measure），它也称为库尔贝克 - 莱布勒散度（Kullback-Leibler divergence），它是一个不满足上述第三个条件的距离测量方法，这个测量是非对称的。因此，使用它作为文本文档的相似度测量是没有意义的，但是另一方面，它在区分各种分布和模式方面很有价值。

6.1.4 无监督的机器学习算法

无监督的机器学习算法属于 ML 算法系列，它们尝试从数据的各种属性和特征中发现其中潜在的、隐藏的结构和模式。此外，一些无监督学习算法也用来减少特征空间，通常是将高维度的特征空间转变为低维度的特征空间。这些算法所运行的数据基本上是没有预先分类的未标记数据。应用这些算法的目的是寻找模式并识别特征，这有助于将各种数据点分成组或类。这些算法通常称为聚类算法。第 5 章介绍的主题模型也属于无监督的学习算法。

以上就是本章所需的重要概念和背景知识。接下来，简要介绍文本规范化和特征提取。

6.2 文本规范化

在进一步开展分析或 NLP 之前，首先需要规范文本文档和语料库。为此，将再次使用第 5 章中的规范化模块，此外还需要应用一些专门针对本章内容的新技术。本章的代码文件 `normalization.py` 中提供了完整的规范化模块，为了便于读者学习，我会在本节的规范化模块中突出显示新增内容。

在分析了许多语料库后，经过精心挑选了一些新词，并将它们更新进了停用词名单，如下代码段所示：

```
stopword_list = nltk.corpus.stopwords.words('english')
stopword_list = stopword_list + ['mr', 'mrs', 'come', 'go', 'get', 'tell',
'listen', 'one', 'two', 'three', 'four', 'five',
'six', 'seven', 'eight',
'nine', 'zero', 'join', 'find', 'make', 'say', 'ask',
'tell', 'see', 'try', 'back', 'also']
```

可以看出新添加的单词大多数是通用的、没有多大意义的动词或名词。将它们更新进停用词列表对于文本聚类中的特征提取是十分有用的。还在规范化 pipeline 中添加了一个新函数，它使用正则表达式从文本主体中提取文本标识，如下所示：

```
import re

def keep_text_characters(text):
    filtered_tokens = []
    tokens = tokenize_text(text)
    for token in tokens:
        if re.search('[a-zA-Z]', token):
            filtered_tokens.append(token)
    filtered_text = ' '.join(filtered_tokens)
    return filtered_text
```

我们将新函数连同在前几章中反复使用的函数（包括扩展缩写词，解码 HTML，词语切分，删除停用词及特殊字符，词形还原）一起添加到最终的规范化函数中。更新后的规范化函数如下所示：

```
def normalize_corpus(corpus, lemmatize=True,
                     only_text_chars=False,
                     tokenize=False):

    normalized_corpus = []
    for text in corpus:
        text = html_parser.unescape(text)
        text = expand_contractions(text, CONTRACTION_MAP)
        if lemmatize:
            text = lemmatize_text(text)
        else:
            text = text.lower()
        text = remove_special_characters(text)
        text = remove_stopwords(text)
        if only_text_chars:
            text = keep_text_characters(text)

        if tokenize:
            text = tokenize_text(text)
            normalized_corpus.append(text)
        else:
            normalized_corpus.append(text)

    return normalized_corpus
```

你可以看出上述函数非常类似于第 5 章中的函数，只是添加了 keep_text_characters() 函数来保留文本字符，该函数通过将 only_text_chars 参数设置为 True 来执行。

6.3 特征提取

我们还将使用特征提取函数，该函数与第 5 章中的特征提取函数类似。函数代码也与之前的特征提取器代码类似，但是在本章中将添加一些新的参数。该函数可以在 utils.py 文件中找到，函数代码如下所示：

```
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer

def build_feature_matrix(documents, feature_type='frequency',
                         ngram_range=(1, 1), min_df=0.0, max_df=1.0):

    feature_type = feature_type.lower().strip()

    if feature_type == 'binary':
        vectorizer = CountVectorizer(binary=True, min_df=min_df,
                                     max_df=max_df, ngram_range=ngram_range)
    elif feature_type == 'frequency':
        vectorizer = CountVectorizer(binary=False, min_df=min_df,
                                     max_df=max_df, ngram_range=ngram_range)
    elif feature_type == 'tfidf':
        vectorizer = TfidfVectorizer(min_df=min_df, max_df=max_df,
                                     ngram_range=ngram_range)
    else:
        raise Exception("Wrong feature type entered. Possible values:
'binary', 'frequency',
                        'tfidf'")

    feature_matrix = vectorizer.fit_transform(documents).astype(float)

    return vectorizer, feature_matrix
```

从函数定义可以看出，它可以提取词袋模型频率、出现次数以及基于 TF-IDF 的特征。此函数新增了 min_df、max_dfC 和 ngram_range 参数，并将其设为可选参数。当要添加二元分词、三元分词等作为附加特征时，ngram_range 参数将会十分有用。min_df 参数可以由［0.0，1.0］范围内的阈值表示，并将忽略文档频率低于输入阈值的特征。max_df 参数也可以由［0.0，1.0］范围内的阈值表示，并且将忽略文档频率高于输入阈值的特征。这样做的原因是，如果这些词语出现在几乎所有的文件中，那么它们对于区分不同文件的类型往往没有多少价值。接下来，将深入了解文本相似度分析的各种技术。

6.4 文本相似度

文本相似度分析的主要目的是分析和测量两个文本实体彼此距离的远近。这些文本实体可以是简单的标识或词项，例如单词，也可以是包含句子和文本段落的整个文档。目前，有各种各样的文本相似度分析方法，文本相似度分析的目的大致分为以下两个方面。

- 词汇相似度：通过句法、结构和内容研究文本文档的内容，并根据这些参数测量其相似度。
- 语义相似度：首先找出文档的语义、含义和上下文，然后找出它们彼此的距离。在这方面，依存语法和实体识别是很有用的工具。

目前最流行的研究领域是词汇相似度分析，因为这些技术更简单、更易于实现，还可以使用简单的模型（如词袋模型）实现语义相似度的一些分析。通常，距离度量用来衡量文本实体之间的相似度。接下来，将主要介绍以下两个领域的文本相似度。

- 词项相似度：在这里，将测量每个标识或单词之间的相似度。
- 文档相似度：在这里，将测量整个文本文档之间的相似度。

我们的思路是实现并使用几个距离度量，看看如何测量和分析只具有简单单词的实体之间的相似度，然后再看看当测量由复杂词组组成的文档之间的相似度时，会发生什么变化。

6.5 词项相似度分析

我们将从分析词项相似度入手，或者更准确地说，将从分析单独的单词标识相似度入手。虽然词项相似度分析没有在实际应用中大量使用，但是仍可以作为理解文本相似度分析的一个很好的出发点。当然，一些应用程序和用例（如自动填充程序、拼写检查和文本校正器）也会使用词项相似度分析中的部分技术来纠正拼写错误的词项。在这里，将选取一些单词并计算它们之间的相似度，然后应用不同的单词表示方法和距离度量进行相似度分析。我们将使用如下单词表示方法：

- 字符向量化。
- 字符袋（Bag of Character）向量化。

字符向量化是非常简单的过程，它将词项的每个字符映射到唯一的对应数字。可以使用以下代码段来实现：

```
import numpy as np

def vectorize_terms(terms):
    terms = [term.lower() for term in terms]
    terms = [np.array(list(term)) for term in terms]
    terms = [np.array([ord(char) for char in term])
                for term in terms]
    return terms
```

该函数输入一列单词或词项，并返回相应的字符向量。字符袋向量化与词袋模型非常类似，区别是在这里我们会计算单词中每个字符的频率。字符袋向量化过程中不考虑字符序列或单词顺序。以下函数可以帮助我们完成上述过程：

```
from scipy.stats import itemfreq

def boc_term_vectors(word_list):
    word_list = [word.lower() for word in word_list]
    unique_chars = np.unique(
                        np.hstack([list(word)
                        for word in word_list]))
    word_list_term_counts = [{char: count for char, count in
itemfreq(list(word))}
                                for word in word_list]

    boc_vectors = [np.array([int(word_term_counts.get(char, 0))
                                for char in unique_chars])
                   for word_term_counts in word_list_term_counts]
    return list(unique_chars), boc_vectors
```

在这个函数中，输入的是单词或词项，然后从所有单词中提取出特殊字符。与词袋模型类似，这就是特征列表，不同的是，在词袋模型中特殊词（而不是字符）才是特征。有了 unique_chars 的这个列表之后，就可以得到所有单词中每个字符的计数，并构建字符袋向量。

可以在下面的代码段中看到前述函数的应用。我们将使用四个示例词项，并计算它们之间的相似度：

```
root = 'Believe'
term1 = 'beleive'
term2 = 'bargain'
term3 = 'Elephant'
```

```
terms = [root, term1, term2, term3]

# Character vectorization
vec_root, vec_term1, vec_term2, vec_term3 = vectorize_terms(terms)
# show vector representations
In [103]: print '''
     ...: root: {}
     ...: term1: {}
     ...: term2: {}
     ...: term3: {}
     ...: '''.format(vec_root, vec_term1, vec_term2, vec_term3)
root: [ 98 101 108 105 101 118 101]
term1: [ 98 101 108 101 105 118 101]
term2: [ 98  97 114 103  97 105 110]
term3: [101 108 101 112 104  97 110 116]

# Bag of characters vectorization
features, (boc_root, boc_term1, boc_term2, boc_term3) = boc_term_
vectors(terms)
# show features and vector representations
In [105]: print 'Features:', features
     ...: print '''
     ...: root: {}
     ...: term1: {}
     ...: term2: {}
     ...: term3: {}
     ...: '''.format(boc_root, boc_term1, boc_term2, boc_term3)
Features: ['a', 'b', 'e', 'g', 'h', 'i', 'l', 'n', 'p', 'r', 't', 'v']

root: [0 1 3 0 0 1 1 0 0 0 0 1]
term1: [0 1 3 0 0 1 1 0 0 0 0 1]
term2: [2 1 0 1 0 1 0 1 0 1 0 0]
term3: [1 0 2 0 1 0 1 1 1 0 1 0]
```

可以看到我们如何轻松地将文本词项转换为数字向量表示形式。现在，要使用几个距离度量来计算根词和前面代码段中其他三个词之间的相似度。有很多距离指标可以用来计算和测量相似度。本节将介绍以下五个度量：

- 汉明距离（Hamming distance）。
- 曼哈顿距离（Manhattan distance）。
- 欧几里得距离（Euclidean distance）。
- 莱文斯坦编辑距离（Levenshtein edit distance）。
- 余弦距离（Cosine distance）和相似度。

下面将介绍每个距离度量的概念，并使用 numpy 数组来实现必要的计算和数学公式。之后，通过计算示例词项的相似度来具体介绍每个距离度量概念。首先，设置一些必要的变量以存储根词项、其他词项以及它们的向量化表示，如下代码段所示：

```
root_term = root
root_vector = vec_root
root_boc_vector = boc_root

terms = [term1, term2, term3]
vector_terms = [vec_term1, vec_term2, vec_term3]
boc_vector_terms = [boc_term1, boc_term2, boc_term3]
```

现在，已经做好了计算相似度度量的准备，我们将使用前述的词项及其向量表示来测量相似度。

6.5.1 汉明距离

汉明距离在信息和通信领域中广泛使用，它是非常流行的距离度量方法。汉明距离是两个长度相等的字符串之间的测量距离。它的正式定义是两个长度相等的字符串之间互异字符或符号的位置的数量。考虑长度为 n 的两个词项 u 和 v，汉明距离的数学表达式为：

$$hd(u,v) = \sum_{i=1}^{n} (u_i \neq v_i)$$

可以通过将不匹配的数目除以词项的总长度来获得归一化的汉明距离，如下所示：

$$norm\_hd(u,v) = \frac{\sum_{i=1}^{n} (u_i \neq v_i)}{n}$$

其中，n 表示词项的长度。

以下函数可以计算两个词项之间的汉明距离，还可以计算归一化距离：

```
def hamming_distance(u, v, norm=False):
    if u.shape != v.shape:
        raise ValueError('The vectors must have equal lengths.')
    return (u != v).sum() if not norm else (u != v).mean()
```

现在，使用以下代码段来衡量根词项和其他词项之间的汉明距离：

```
# compute Hamming distance
In [115]: for term, vector_term in zip(terms, vector_terms):
     ...:     print 'Hamming distance between root: {} and term: {} is {}'.
format(root_term,
     ...:                            term, hamming_distance(root_vector, vector_
term, norm=False))

Hamming distance between root: Believe and term: believe is 2
Hamming distance between root: Believe and term: bargain is 6
Traceback (most recent call last):
  File "<ipython-input-115-3391bd2c4b7e>", line 4, in <module>
    hamming_distance(root_vector, vector_term, norm=False))
ValueError: The vectors must have equal lengths.

# compute normalized Hamming distance
In [117]: for term, vector_term in zip(terms, vector_terms):
     ...:     print 'Normalized Hamming distance between root: {} and term:
{} is
     ...:                                                 {}'.format(root_term,
     ...:                                                 term,
     ...:                 round(hamming_distance(root_vector, vector_term,
                          norm=True), 2))

Normalized Hamming distance between root: Believe and term: believe is 0.29
Normalized Hamming distance between root: Believe and term: bargain is 0.86
Traceback (most recent call last):
  File "<ipython-input-117-7dfc67d08c3f>", line 4, in <module>
    round(hamming_distance(root_vector, vector_term, norm=True), 2))
ValueError: The vectors must have equal lengths
```

从前面的输出可以看出，在忽略大小写的情况下，'Believe'和'beleive'最为相似，其汉明距离为 2 或 0.29（归一化汉明距离），与'bargain'一词的汉明距离值为 6 或 0.86（这里，数值越小，词项越相似）。词项'Elephant'则引发了程序异常，因为该词的长度是 8，而根词'Believe'的长度为 7，所以不能计算汉明距离，因为汉明距离的假设前提是距离长度相等。

6.5.2 曼哈顿距离

曼哈顿距离度量在概念上类似于汉明距离，区别是曼哈顿距离计算两个字符串的每个位置上对应字符之间的差值，而不是计算不匹配字符的数量。曼哈顿距离也称为城市街区距离、L1 范数、计程车度量，正式定义是基于严格水平或垂直路径的网格中两个点之间的距离，而非通常计算的欧几里得对角距离，其数学表示为：

$$md(u,v) = \| u - v \|_1 = \sum_{i=1}^{n} | u_i - v_i |$$

其中，u 和 v 是长度为 n 的两个词项。与汉明距离相同，这里的假设前提也是两个词项的长度相同。还可以通过绝对差的和除以词项长度来计算曼哈顿归一化距离。表达式如下：

$$norm\_md(u,v) = \frac{\| u - v \|_1}{n} = \frac{\sum_{i=1}^{n} | u_i - v_i |}{n}$$

其中，n 是词项 u 和 v 的长度，以下函数有助于计算曼哈顿距离，也可以计算曼哈顿归一化距离：

```
def manhattan_distance(u, v, norm=False):
    if u.shape != v.shape:
        raise ValueError('The vectors must have equal lengths.')
    return abs(u - v).sum() if not norm else abs(u - v).mean()
```

现在，使用上述函数计算根词项和其他词项之间的曼哈顿距离，如下代码段所示：

```
# compute Manhattan distance
In [120]: for term, vector_term in zip(terms, vector_terms):
     ...:     print 'Manhattan distance between root: {} and term: {} is
              {}'.format(root_term,
     ...:                        term, manhattan_distance(root_vector,
                                 vector_term, norm=False))

Manhattan distance between root: Believe and term: believe is 8
Manhattan distance between root: Believe and term: bargain is 38
Traceback (most recent call last):
  File "<ipython-input-120-b228f24ad6a2>", line 4, in <module>
    manhattan_distance(root_vector, vector_term, norm=False))
ValueError: The vectors must have equal lengths.

# compute normalized Manhattan distance
In [122]: for term, vector_term in zip(terms, vector_terms):
     ...:     print 'Normalized Manhattan distance between root: {} and
              term: {} is {}'.format(root_term,
     ...:                term,
     ...:                round(manhattan_distance(root_vector, vector_term,
                         norm=True),2))
     ...:
     ...:
Normalized Manhattan distance between root: Believe and term: believe is 1.14
Normalized Manhattan distance between root: Believe and term: bargain is 5.43
Traceback (most recent call last):
  File "<ipython-input-122-d13a48d56a22>", line 4, in <module>
    round(manhattan_distance(root_vector, vector_term, norm=True),2))
ValueError: The vectors must have equal lengths.
```

从这些结果可以看出，正如预期的那样，在忽略大小写的情况下，'Believe'和'believe'之间的相似度最高，距离度量得分为 8 或 1.14，与'bargain'一词的距离度量得分为 38 或 5.43（这里得分越小，词越相似）。词项'Elephant'则在运行中产生错误，因为它与根词项的长度不同，就像我们之前在计算汉明距离时注意到的那样。

6.5.3 欧几里得距离

在上一节介绍曼哈顿距离时，曾简要地提到欧几里得距离。欧几里得距离也称为欧几里得范数、L2 范数或 L2 距离，它的正式定义是两点之间最短的直线距离。数学上可以表示为：

$$ed(u,v) = \| u - v \|_2 = \sqrt{\sum_{i=1}^{n} (u_i - v_i)^2}$$

其中，u 和 v 两点是场景中的向量化文本词项，每个的长度都为 n。以下函数有助于计算两个词项之间的欧几里得距离：

```
def euclidean_distance(u, v):
    if u.shape != v.shape:
        raise ValueError('The vectors must have equal lengths.')
    distance = np.sqrt(np.sum(np.square(u - v)))
    return distance
```

现在，使用上述函数计算词项之间的欧几里得距离，如下代码段所示：

```
# compute Euclidean distance
In [132]: for term, vector_term in zip(terms, vector_terms):
     ...:     print 'Euclidean distance between root: {} and term: {} is
{}'.format(root_term,
     ...:                             term, round(euclidean_distance(root_
                                      vector, vector_term),2))
Euclidean distance between root: Believe and term: believe is 5.66
Euclidean distance between root: Believe and term: bargain is 17.94
Traceback (most recent call last):
  File "<ipython-input-132-90a4dbe8ce60>", line 4, in <module>
    round(euclidean_distance(root_vector, vector_term),2))
ValueError: The vectors must have equal lengths.
```

从上述输出可以看出，'Believe'和'beleive'这两个词最相似，得分为 5.66，'bargain'得分为 17.94，'Elephant'则给出了一个 ValueError，因为前面提到的基本假设（即被比较的字符串应该具有相同的长度）也适用于该距离度量。

到目前为止，我们使用的所有距离度量都只适用于相同长度的字符串或词项，并且在长度不等时计算失败。那么如何应对这个问题呢？接下来，介绍几个即使在字符串长度不相等的情况下也能够计算相似度的距离度量。

6.5.4 莱文斯坦编辑距离

莱文斯坦编辑距离通常也称为莱文斯坦距离，属于基于编辑距离的度量，它用于根据字符差异来测量两个字符串之间的距离——类似于汉明距离的度量概念。两个词项之间的莱文斯坦编辑距离可以定义为通过添加、删除或替换将一个词项转变成另一个词项所需的最少编辑次数。这些替代是基于字符的替代，其中单次操作可以编辑单个字符。此外，如前所述，这两个词项的长度在这里不必相等。在数学上，可以将两个词项之间的莱文斯坦编辑距离表示为 $ld_{u,v}(|u|,|v|)$，其中，u 和 v 是词项，而 $|u|$ 和 $|v|$ 是其长度。该距离可以用以下公式表示

$$ld_{u,v}(i,j) = \begin{cases} \max(i,j) & \text{如果 } \min(i,j) = 0 \\ \min\begin{cases} ld_{u,v}(i-1,j) + 1 \\ ld_{u,v}(i,j-1) + 1 \\ ld_{u,v}(i-1,j-1) + C_{u_i \neq v_j} \end{cases} & \text{其他} \end{cases}$$

其中，i 和 j 是词项 u 和 v 的索引。上面最小项的第三个方程包含一个由 $C_{u_i \neq v_j}$ 表示的成本函数，它具有如下限制条件：

$$C_{u_i \neq v_j} = \begin{cases} 1, & u_i \neq v_j \\ 0, & u_i = v_j \end{cases}$$

上式为指示函数，它说明了匹配两个词项的对应字符所需的成本（该方程式代表了匹配操作或不匹配操作）。上一个最小值中的第一个方程表示删除操作，第二个方程表示插入操作。这样，函数 $ld_{u,v}(i,j)$ 就涵盖了前面提到的插入、删除和添加的所有操作，它表示在词项 u 的第 i 个字母到词项 v 的第 j 个字母之间的莱文斯坦编辑距离。对于莱文斯坦编辑距离，还有几个有趣的边界条件：

- 两个词项之间编辑距离的最小值是两个词项长度的差值。
- 两个词项之间编辑距离的最大值可以是较大词项的长度。
- 如果两个词项完全一样，则编辑距离为零。
- 当且仅当两个词项具有相同的长度时，两个词项之间莱文斯坦编辑距离的上限为汉明距离。
- 莱文斯坦编辑距离作为距离度量也满足三角不等式特性（在讨论距离度量时讨论过三角不等式特性）。

有很多种计算莱文斯坦编辑距离的方法，在这里，以两个词项为例介绍莱文斯坦编辑距离计算。考虑根词项'believe'和另一个词项'beleive'（在计算中忽略大小写）。两者的编辑距离应为2，因为需要进行以下两个操作完成'beleive'到根词项的变换：

- 'beleive'→'beliive'（将 e 替换为 i）。
- 'beliive'→'believe'（替换为 e）。

为了实现这一点，需要构建一个矩阵，它可以通过比较第一词项的每个字符和第二词项的每个字符，计算两个词项所有字符之间的莱文斯坦距离。为了便于计算，遵循一种动态规划方法，根据最终计算结果得到两个词项之间的编辑距离。对于给定的两个词项，这里的算法应生成的莱文斯坦编辑距离矩阵如图 6-1 所示。

	b	e	l	i	e	v	e
b	0	1	2	3	4	5	6
e	1	0	1	2	3	4	5
l	2	1	0	1	2	3	4
e	3	2	1	1	1	2	3
i	4	3	2	1	2	2	3
v	5	4	3	2	2	2	3
e	6	5	4	3	2	3	2

图 6-1 两个词项之间的莱文斯坦编辑距离矩阵

由图 6-1 可见，对词项中的每对字符都计算编辑距离，如前所述，图中突出显示的最终编辑距离值即为两个词项之间的实际编辑距离。该算法也称为瓦格纳 - 费舍尔（Wagner-Fischer）算法，如果你想了解更多细节，可以参考瓦格纳（R. Wagner）和费舍尔（M. Fischer）的“字符串到字符串校正问题（The String-to-String Correction Problem）”一文。该算法的伪代码如下所示（由该文章提供）：

```
function levenshtein_distance(char u[1..m], char v[1..n]):
# for all i and j, d[i,j] will hold the Levenshtein distance between the
first i characters of
# u and the first j characters of v, note that d has (m+1)*(n+1) values
int d[0..m, 0..n]

# set each element in d to zero
```

```
d[0..m, 0..n] := 0

# source prefixes can be transformed into empty string by dropping all
characters
for i from 1 to m:
   d[i, 0] := i

# target prefixes can be reached from empty source prefix by inserting every
character
for j from 1 to n:
    d[0, j] := j

# build the edit distance matrix
for j from 1 to n:
    for i from 1 to m:
        if s[i] = t[j]:
            substitutionCost := 0
        else:
            substitutionCost := 1
            d[i, j] := minimum(d[i-1, j] + 1,                  # deletion
                               d[i, j-1] + 1,                  # insertion
                               d[i-1, j-1] + substitutionCost)  # substitution

# the final value of the matrix is the edit distance between the terms
return d[m, n]
```

从上述的函数伪代码可以看出是如何获得定义莱文斯坦编辑距离的必要公式的。现在将在 Python 中实现上述伪代码。上述算法使用的是 $O(mn)$ 空间，因为虽然它包含整个距离矩阵，但是只需存储前一行和当前的距离行便足以获得最终结果。我们将在代码中执行相同的操作，同时也会将结果存储在一个矩阵中，以便于在最后将其显示出来。以下函数可以实现莱文斯坦编辑距离计算：

```
import copy
import pandas as pd

def levenshtein_edit_distance(u, v):
    # convert to lower case
    u = u.lower()
    v = v.lower()
    # base cases
    if u == v: return 0
    elif len(u) == 0: return len(v)
    elif len(v) == 0: return len(u)
    # initialize edit distance matrix
    edit_matrix = []
    # initialize two distance matrices
    du = [0] * (len(v) + 1)
    dv = [0] * (len(v) + 1)
    # du: the previous row of edit distances
    for i in range(len(du)):
        du[i] = i
    # dv : the current row of edit distances
    for i in range(len(u)):
        dv[0] = i + 1
        # compute cost as per algorithm
        for j in range(len(v)):
            cost = 0 if u[i] == v[j] else 1
            dv[j + 1] = min(dv[j] + 1, du[j + 1] + 1, du[j] + cost)
        # assign dv to du for next iteration
        for j in range(len(du)):
            du[j] = dv[j]
        # copy dv to the edit matrix
        edit_matrix.append(copy.copy(dv))
    # compute the final edit distance and edit matrix
```

```
    distance = dv[len(v)]
    edit_matrix = np.array(edit_matrix)
    edit_matrix = edit_matrix.T
    edit_matrix = edit_matrix[1:,]
    edit_matrix = pd.DataFrame(data=edit_matrix,
                               index=list(v),
                               columns=list(u))
    return distance, edit_matrix
```

该函数返回两个词项 u 和 v 之间最终的莱文斯坦编辑距离以及完整的编辑矩阵，其中 u 和 v 是我们的输入。请记住，我们需要以原始字符串格式，而不是以它们的向量表示来传递词项。此外，在这里我们不考虑字符串的大小写，并将所有字符全部转换为小写。

以下代码段使用上述函数来计算示例词项之间的莱文斯坦编辑距离：

```
In [223]: for term in terms:
     ...:     edit_d, edit_m = levenshtein_edit_distance(root_term, term)
     ...:     print 'Computing distance between root: {} and term: {}'.
              format(root_term,
     ...:                                                          term)
     ...:     print 'Levenshtein edit distance is {}'.format(edit_d)
     ...:     print 'The complete edit distance matrix is depicted below'
     ...:     print edit_m
     ...:     print '-'*30

Computing distance between root: Believe and term: beleive
Levenshtein edit distance is 2
The complete edit distance matrix is depicted below
   b  e  l  i  e  v  e
b  0  1  2  3  4  5  6
e  1  0  1  2  3  4  5
l  2  1  0  1  2  3  4
e  3  2  1  1  1  2  3
i  4  3  2  1  2  2  3
v  5  4  3  2  2  2  3
e  6  5  4  3  2  3  2
------------------------------
Computing distance between root: Believe and term: bargain
Levenshtein edit distance is 6
The complete edit distance matrix is depicted below
   b  e  l  i  e  v  e
b  0  1  2  3  4  5  6
a  1  1  2  3  4  5  6
r  2  2  2  3  4  5  6
g  3  3  3  3  4  5  6
a  4  4  4  4  4  5  6
i  5  5  5  4  5  5  6
n  6  6  6  5  5  6  6
------------------------------
Computing distance between root: Believe and term: Elephant
Levenshtein edit distance is 7
The complete edit distance matrix is depicted below
   b  e  l  i  e  v  e
e  1  1  2  3  4  5  6
l  2  2  1  2  3  4  5
e  3  2  2  2  2  3  4
p  4  3  3  3  3  3  4
h  5  4  4  4  4  4  4
a  6  5  5  5  5  5  5
n  7  6  6  6  6  6  6
t  8  7  7  7  7  7  7
------------------------------
```

从上述输出可以看出，'Believe'和'beleive'彼此最接近，编辑距离为 2，'Believe'与'bargain''Elephant'之间的距离为 6，表示总共需要 6 个编辑操作。通过编辑距离矩阵

可以更详细地了解算法如何通过每次迭代计算距离。

6.5.5 余弦距离和相似度

余弦距离是一个可以从余弦相似度推导得出的度量，反之亦然。考虑两个词项，它们以向量化形式表示，余弦相似度给出了当它们在内积空间中为非零正向量时，它们之间角度的余弦值。因此具有相似方向的词项向量将拥有更接近于1（cos0°）的相似度得分，表示向量在相同方向上彼此非常接近（它们之间的夹角接近零度）。相似度得分接近0（cos90°）则表示两个词项向量的夹角近似直角，它们是无关词项。相似度得分接近-1（cos180°）表示彼此是完全相反的词项。图6-2给出了详细的说明，其中 u 和 v 是我们在向量空间中的词项向量。

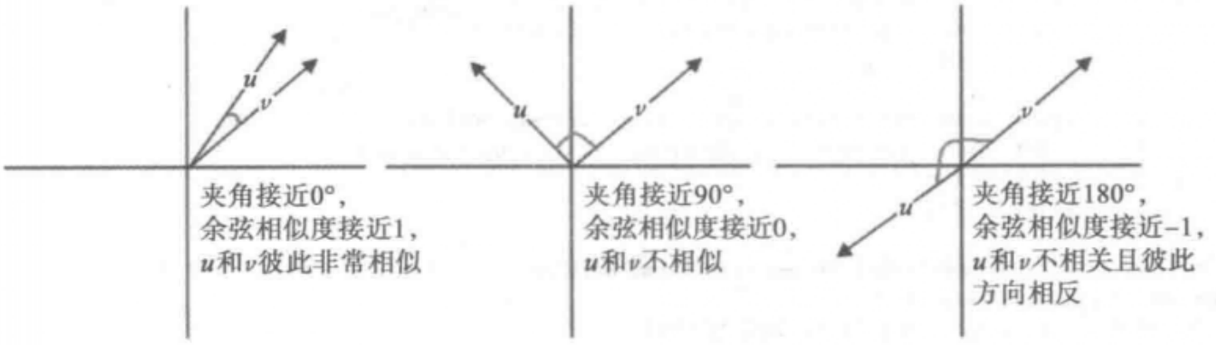


图6-2 词项向量的余弦相似度

你可以从向量的位置看出它们的关系，这些图可以更清楚地显示出向量是彼此靠近还是彼此远离的，它们之间夹角的余弦值给出了余弦相似度度量。我们可以将余弦相似度正式定义为两个词项向量 u 和 v 的点积除以它们的L2范数的乘积。在数学上，我们可以用如下表达式表示两个向量之间的点积：

$$u \cdot v = \| u \| \| v \| \cos(\theta)$$

其中 θ 是 u 和 v 之间的角度，$\| u \|$ 表示向量 u 的L2范数，$\| v \|$ 表示向量 v 的L2范数。我们可以从上面的公式导出余弦相似度的表达式：

$$cs(u,v) = \cos(\theta) = \frac{u \cdot v}{\| u \| \| v \|} = \frac{\sum_{i=1}^{n} u_i v_i}{\sqrt{\sum_{i=1}^{n} u_i^2} \sqrt{\sum_{i=1}^{n} v_i^2}}$$

其中 $cs(u,v)$ 是 u 和 v 之间的余弦相似度得分，这里 u_i 和 v_i 是两个向量的各类特征或组件，这些特征或组件的总数为 n。在我们的示例中，将使用字符包向量化来构建这些词项向量，n 表示待分析的所有词项的特殊字符的数量。在这里需要注意一件很重要的事情，那就是通常余弦相似度得分的范围从-1到+1，但是如果我们使用字符袋（基于字符频率）或词袋（基于词频），那么分数的范围将会是从0到1，因为频率向量永远不会是负的，所以两个向量之间的角度不能超过90°。我们可用如下公式来表示余弦距离：

$$cd(u,v) = 1 - cs(u,v) = 1 - \cos(\theta) = 1 - \frac{u \cdot v}{\| u \| \| v \|} = 1 - \frac{\sum_{i=1}^{n} u_i v_i}{\sqrt{\sum_{i=1}^{n} u_i^2} \sqrt{\sum_{i=1}^{n} v_i^2}}$$

其中 $cs(u,v)$ 表示词项向量 u 和 v 之间的余弦距离。以下函数给出了基于上述公式实现余弦距离的计算过程：

```
def cosine_distance(u, v):
    distance = 1.0 - (np.dot(u, v) /
                      (np.sqrt(sum(np.square(u))) * np.sqrt(sum(np.
                      square(v))))
                     )
    return distance
```

接下来，我们将使用之前创建的字符袋表示来测试示例词项之间的相似度，如下代码段所示：

```
In [235]: for term, boc_term in zip(terms, boc_vector_terms):
     ...:     print 'Analyzing similarity between root: {} and term: {}'.
format(root_term,
     ...:                                                          term)
     ...:     distance = round(cosine_distance(root_boc_vector, boc_term),2)
     ...:     similarity = 1 - distance
     ...:     print 'Cosine distance  is {}'.format(distance)
     ...:     print 'Cosine similarity  is {}'.format(similarity)
     ...:     print '-'*40
Analyzing similarity between root: Believe and term: believe
Cosine distance  is -0.0
Cosine similarity  is 1.0
----------------------------------------
Analyzing similarity between root: Believe and term: bargain
Cosine distance  is 0.82
Cosine similarity  is 0.18
----------------------------------------
Analyzing similarity between root: Believe and term: Elephant
Cosine distance  is 0.39
Cosine similarity  is 0.61
----------------------------------------
```

这些向量表示不考虑字符的顺序，因此"Believe"一词和"beleive"的相似度为 1.0 或 100% 完全相似，因为它们有着具有相同频率的相同字符。你可以看出将其与 WordNet 这样的语义字典组合使用的好处，即可以在用户键入拼写错误的单词时，通过测量单词之间的相似度来提供语义上和语法上正确的单词，从而提供正确的拼写建议。在这里，你也可以尝试使用不同的特征，如同时取两个字符并计算其频率来构建词项向量，而非示例中所示的使用单个字符的频率构建向量。这样就对词语中的部分字符顺序加入了考量。你可以尽情地尝试不同的可能性并比较结果！当测量大文件或长句子之间的相似度时，这种距离度量方法十分有效，在我们下一节讨论文档相似度时，你将会看出其效用。

6.6 文档相似度分析

在上一节中，我们使用了各种相似度和距离度量来分析词项之间的相似度。我们还看到了向量化是如何发挥效用的，它使得数学计算变得更加容易，特别是在计算向量之间的距离时。在本节中，我们将尝试分析文档之间的相似之处。到目前为止，你想必已经知道了文档的定义是可以由句子或文本段落组成的文本体。为了分析文档相似度，我们将使用 `utils` 模块的 `build_feature_ matrix()` 函数从文档中提取特征。我们将使用文档的 TF-IDF 相似度对文档进行向量化，在之前的分类文本文档和归纳整个文档时我们曾使用过该方法。有了各种文档的向量表示之后，我们将使用几个距离或相似度度量来计算文档之间的相似度。本节

将介绍如下度量：

- 余弦相似度。
- 海灵格 - 巴塔恰亚（Hellinger-Bhattacharya）距离。
- Okapi BM25 排名。

像以往一样，我们将介绍每个度量背后的概念，其数学表达形式和定义，然后使用 Python 实现它们。在本节中，我们还将使用一个包含九个文档的小语料库和一个包含三个文档（也是我们的查询文档）的独立语料库进行测试。对于这三个文档中的每一个，我们都将尝试从包含九个文档的语料库中找出最相似的文档，小语料库将作为我们的查询索引。我们可以把这一过程想象成对搜索过程的一个微型模拟——当你使用句子进行搜索时，最相关的结果将从搜索引擎的网页索引中返回给你。在我们的用例中，待检索的内容是三个文档，我们将根据相似度度量返回九个文档中相关文档的索引。

接下来，我们从加载必要的依存关系和用来测试各种度量指标的文档语料库开始着手，如下面的代码段所示：

```
from normalization import normalize_corpus
from utils import build_feature_matrix
import numpy as np

# load the toy corpus index
toy_corpus = ['The sky is blue',
'The sky is blue and beautiful',
'Look at the bright blue sky!',
'Python is a great Programming language',
'Python and Java are popular Programming languages',
'Among Programming languages, both Python and Java are the most used in
Analytics',
'The fox is quicker than the lazy dog',
'The dog is smarter than the fox',
'The dog, fox and cat are good friends']

# load the docs for which we will be measuring similarities
query_docs = ['The fox is definitely smarter than the dog',
              'Java is a static typed programming language unlike Python',
              'I love to relax under the beautiful blue sky!']
```

从该代码段可以看出，在我们的语料库索引中有各种各样的文档，涉及天空、程序语言和动物。此外，我们还有三个查询文档，我们希望根据相似度计算从 toy_corpus 索引中获取与其最相关的文档。在开始介绍度量之前，我们首先要规范化文档并通过提取 TF-IDF 特征将其向量化，如下代码段所示：

```
# normalize and extract features from the toy corpus
norm_corpus = normalize_corpus(toy_corpus, lemmatize=True)
tfidf_vectorizer, tfidf_features = build_feature_matrix(norm_corpus,
                                                        feature_
                                                        type='tfidf',
                                                        ngram_range=(1, 1),
                                                        min_df=0.0, max_
                                                        df=1.0)
# normalize and extract features from the query corpus
norm_query_docs =  normalize_corpus(query_docs, lemmatize=True)
query_docs_tfidf = tfidf_vectorizer.transform(norm_query_docs)
```

现在，我们已经完成了文档规范化并使用基于 TF-IDF 的向量表示方式实现了文档向量化，接下来我们将探究如何计算本节伊始介绍的每个度量的相似度值。

6.6.1 余弦相似度

我们已经介绍余弦相似度计算的相关概念，并对词项应用了余弦相似度度量。我们将继续使用相同的概念来计算文档的余弦相似度得分，采用基于词袋模型的文档向量，并用 TF-IDF 数值代替词频。在这里，我们同样只采用一元分词形式，但是你也可以在向量化过程中尝试采用二元分词等方式，并将其作为文档特征。对于三个查询文档中的每一个，我们都将使用 `toy_corpus` 中的九个文档计算其相似度，并返回 n 个最相似的文档，其中 n 为用户输入参数。

我们将定义一个函数，它的输入是向量化的语料库和需要计算相似度的文档语料库。与前几节类似，我们使用点积运算获得相似度得分，并以倒序的方式对文档进行排序，以获得相似度最高的 n 个文档。下面的函数实现了上述功能：

```
def compute_cosine_similarity(doc_features, corpus_features,
                              top_n=3):
    # get document vectors
    doc_features = doc_features.toarray()[0]
    corpus_features = corpus_features.toarray()
    # compute similarities
    similarity = np.dot(doc_features,
                        corpus_features.T)
    # get docs with highest similarity scores
    top_docs = similarity.argsort()[::-1][:top_n]
    top_docs_with_score = [(index, round(similarity[index], 3))
                           for index in top_docs]
    return top_docs_with_score
```

在该函数中，`corpus_features` 是位于 `toy_corpus` 索引中的向量化文档。这些文件将根据与 `doc_features` 的相似度得分进行抓取，`doc_features` 代表了属于每个 `query_doc` 的向量化文档，如下代码段所示：

```
# get Cosine similarity results for our example documents
In [243]: print 'Document Similarity Analysis using Cosine Similarity'
     ...: print '='*60
     ...: for index, doc in enumerate(query_docs):
     ...:
     ...:     doc_tfidf = query_docs_tfidf[index]
     ...:     top_similar_docs = compute_cosine_similarity(doc_tfidf,
     ...:                                                  tfidf_features,
     ...:                                                  top_n=2)
     ...:     print 'Document',index+1 ,':', doc
     ...:     print 'Top', len(top_similar_docs), 'similar docs:'
     ...:     print '-'*40
     ...:     for doc_index, sim_score in top_similar_docs:
     ...:         print 'Doc num: {} Similarity Score: {}\nDoc: {}'.
                  format(doc_index+1,
     ...:
sim_score, toy_corpus[doc_index])
     ...:         print '-'*40
     ...:     print

Document Similarity Analysis using Cosine Similarity
============================================================
Document 1 : The fox is definitely smarter than the dog
Top 2 similar docs:
----------------------------------------
Doc num: 8 Similarity Score: 1.0
Doc: The dog is smarter than the fox
----------------------------------------
```

```
Doc num: 7 Similarity Score: 0.426
Doc: The fox is quicker than the lazy dog
----------------------------------------

Document 2 : Java is a static typed programming language unlike Python
Top 2 similar docs:
----------------------------------------
Doc num: 5 Similarity Score: 0.837
Doc: Python and Java are popular Programming languages
----------------------------------------
Doc num: 6 Similarity Score: 0.661
Doc: Among Programming languages, both Python and Java are the most used in
Analytics
----------------------------------------

Document 3 : I love to relax under the beautiful blue sky!
Top 2 similar docs:
----------------------------------------
Doc num: 2 Similarity Score: 1.0
Doc: The sky is blue and beautiful
----------------------------------------
Doc num: 1 Similarity Score: 0.72
Doc: The sky is blue
----------------------------------------
```

基于余弦相似度得分，上面的输出给出了与每个查询文档最相关的两个文档，你可以看到输出是符合预期的。关于动物的文档与提及狐狸和狗的文档相似；关于 Python 和 Java 的文档与提及这两种程序语言的查询文档最相似；美丽的蓝天也确实类似于谈论天空是蓝色而美丽的文档！

还要注意前面输出中的余弦相似度得分，其中 1.0 表示完全相似，0.0 表示不相似，它们之间的分数表示不同的相似度水平（基于得分多少）。例如，在最后一个例子中，主要的文档向量是 ['sky'、'blue'、'beautiful']，因为它们都与小语料库中的第一个文档相匹配，所以我们获得了 1.0 或 100% 的相似度得分，只有 ['sky'、'blue'] 与第二个最相似的文档匹配，因为我们得到 0.72 或 72% 的相似度得分。你应该还记得在之前的内容里，我简要地提及了余弦相似度使用基于词袋的向量，仅仅考虑标识的权重，而不考虑词项的顺序。这在大型文档中是非常可取的，因为相同的内容可能会以不同的方式描绘，所以捕获词语序列可能会导致信息丢失，从而导致我们不希望看到的误匹配。

我们建议使用 scikit-learn 的 cosine_similarity()函数，你可以在 sklearn.metrics.pairwise 模块中找到它。它使用类似的逻辑实现以上功能，但是相比之下性能更优，并在大型文档上表现良好。你还可以直接使用 gensim.matutils 模块中提供的 gensim 相似度（similarities）模块或 cossim() 函数。

6.6.2 海灵格 - 巴塔恰亚距离

海灵格 - 巴塔恰亚（Hellinger-Bhattacharya）距离（HB 距离）也称为海灵格距离或巴塔恰亚距离。巴塔恰亚距离由巴塔恰亚（A. Bhattacharya）提出，用于测量两个离散或连续概率分布之间的相似度。海灵格（E. Hellinger）在 1909 年提出了海灵格积分，用于计算海灵格距离。总的来说，海灵格 - 巴塔恰亚距离是一个 f 散度（f-divergence），f 散度在概率论中定义为函数 $D_f(P \parallel Q)$，可用于测量 P 和 Q 概率分布之间的差异。有多种 f 散度的实例，包括 KL 散度和 HB 距离。请记住，KL 散度不是一个距离度量，因为它不符合将距离测量值作为度量所需的四个条件。

对于连续和离散的概率分布，均可以计算 HB 距离。在我们的例子中，我们将会使用基于 TF-IDF 的向量作为文档的概率分布。该分布为离散分布，因为对于特定的特征项有特定的 TF-IDF 值，即数值不连续。海灵格 - 巴塔恰亚距离的数学定义为：

$$hbd(u,v) = \frac{1}{\sqrt{2}} \parallel \sqrt{u} - \sqrt{v} \parallel_2$$

其中 $hbd(u,v)$ 表示文档向量 u 和 v 之间的海灵格 - 巴塔恰亚距离，并且它等于向量的平方根差的欧几里得或 L2 范数除以 2 的平方根。考虑到文档向量 u 和 v 是具有 n 个特征的离散量，我们可以进一步扩展上式为：

$$hbd(u,v) = \frac{1}{\sqrt{2}} \sqrt{\sum_{i=1}^{n} (\sqrt{u_i} - \sqrt{v_i})^2}$$

其中 $u = (u_1, u_2, \cdots, u_n)$ 和 $v = (v_1, v_2, \cdots, v_n)$ 是长度为 n 的文档向量，n 表示有 n 个特征，它们是文档中各类词项的 TF-IDF 权重。与前面的余弦相似度计算类似，我们以相同的原理建立函数；我们会将文档向量语料库和单个文档向量作为输入，这些单个文档向量正是我们希望基于 HB 距离从语料库获取 n 个最相似文档的文档向量。如下函数使用 Python 语言实现了上述概念：

```
def compute_hellinger_bhattacharya_distance(doc_features, corpus_features,
                                            top_n=3):
    # get document vectors
    doc_features = doc_features.toarray()[0]
    corpus_features = corpus_features.toarray()
    # compute hb distances
    distance = np.hstack(
                    np.sqrt(0.5 *
                            np.sum(
                                np.square(np.sqrt(doc_features) -
                                          np.sqrt(corpus_features)),
                                axis=1)))
    # get docs with lowest distance scores
    top_docs = distance.argsort()[:top_n]
    top_docs_with_score = [(index, round(distance[index], 3))
                           for index in top_docs]
    return top_docs_with_score
```

从上述实现过程中你可以看出，我们按照得分对文档进行了升序排列，因为与余弦相似度不同（其中 1.0 表示完全相似），这里是分布之间的距离度量，得分为 0 表示完全相似，而较高的数值则表示存在一些不相似之处。我们现在可以将此函数应用于示例语料库计算 HB 距离，你可以在如下代码段中看到打分结果：

```
# get Hellinger-Bhattacharya distance based similarities for our example
documents
In [246]: print 'Document Similarity Analysis using Hellinger-Bhattacharya
distance'
     ...: print '='*60
     ...: for index, doc in enumerate(query_docs):
     ...:
     ...:     doc_tfidf = query_docs_tfidf[index]
     ...:     top_similar_docs = compute_hellinger_bhattacharya_
              distance(doc_tfidf,
     ...:                                              tfidf_features,
     ...:                                              top_n=2)
     ...:     print 'Document',index+1 ,':', doc
     ...:     print 'Top', len(top_similar_docs), 'similar docs:'
     ...:     print '-'*40
     ...:     for doc_index, sim_score in top_similar_docs:
```

```
...:         print 'Doc num: {} Distance Score: {}\nDoc: {}'.
             format(doc_index+1,
...:                                          sim_score, toy_corpus[doc_
                                              index])
...:         print '-'*40
...:     print
...:
...:
Document Similarity Analysis using Hellinger-Bhattacharya distance
============================================================
Document 1 : The fox is definitely smarter than the dog
Top 2 similar docs:
----------------------------------------
Doc num: 8 Distance Score: 0.0
Doc: The dog is smarter than the fox
----------------------------------------
Doc num: 7 Distance Score: 0.96
Doc: The fox is quicker than the lazy dog
----------------------------------------

Document 2 : Java is a static typed programming language unlike Python
Top 2 similar docs:
----------------------------------------
Doc num: 5 Distance Score: 0.53
Doc: Python and Java are popular Programming languages
----------------------------------------
Doc num: 4 Distance Score: 0.766
Doc: Python is a great Programming language
----------------------------------------

Document 3 : I love to relax under the beautiful blue sky!
Top 2 similar docs:
----------------------------------------
Doc num: 2 Distance Score: 0.0
Doc: The sky is blue and beautiful
----------------------------------------
Doc num: 1 Distance Score: 0.602
Doc: The sky is blue
----------------------------------------
```

上述输出可以看出，具有较低 HB 距离得分的文档与查询文档更为相似，输出文档结果与使用余弦相似度获得的输出文档非常相似。请比较结果，并使用更大的语料库验证这些函数！在构建大型的相似度分析系统时，推荐使用在 `gensim.matutils` 模块（它的逻辑与前述函数相同）中的 `hellinger()`函数。

6.6.3 Okapi BM25 排名

目前，在信息检索和搜索引擎领域中，有几种非常受欢迎的技术，包括 PageRank 和 Okapi BM25，缩写词 BM 代表最佳匹配。这种技术也称为 BM25，但是为了完整起见，我将其称为 Okapi BM25，因为最初 BM25 函数的概念只存在于理论上，伦敦城市大学在 20 世纪 80 年代至 90 年代建立了 Okapi 信息检索系统，才真正实现了这种技术，并用来检索现实世界里真实的文件数据。这种技术也称为基于概率相关性的框架或模型，并由 20 世纪 70 年代至 80 年代由几位科学家提出，包括计算机科学家 S·罗伯森（S. Robertson）和 K·琼斯（K. Jones）。有一些函数可以根据不同的因素对文档进行排名，BM25 是其中之一，其较新的变体是 BM25F，其他变体包括 BM15 和 BM25 +。

Okapi BM25 的正式定义是采用一个基于词袋的模型，根据用户输入检索相关文档的文档排名和检索函数。该查询本身可以是包含句子或句子集合的文档，也可以只是几个单词。

实际上，Okapi BM25 不仅仅是一个函数，而是由一整套评分功能组合在一起构成的一个框架。假设我们有一个查询文档 QD，其中 $QD=(q_1,q_2,\cdots,q_n)$ 包含 n 个词项或关键字，同时我们在文档语料库中有一个语料库文档 CD，我们希望使用相似度得分从中获取与查询文档最相关的文档，正如我们之前所做的那样。那么，我们可以在数学上定义这两个文档之间的 BM25 得分：

$$bm25(CD,QD) = \sum_{i=1}^{n} idf(q_i) \cdot \frac{f(q_i,CD)\cdot(k_1+1)}{f(q_i,CD)+k_1\cdot(1-b+b\cdot\frac{|CD|}{avgdl})}$$

上式中函数 $bm25(CD,QD)$ 基于查询文档 QD 计算文档 CD 的 BM25 排名或得分。函数 $idf(q_i)$ 给出了在包含 CD 的语料库（我们希望从其中检索相关文档的语料库）中的词项 q_i 的逆文档频率（IDF）。不知你是否还记得，我们在第 4 章中实现 TF-IDF 特征提取器时计算过 IDF，它的表达式如下：

$$idf(t) = 1 + \log\frac{C}{1+df(t)}$$

其中 $idf(t)$ 表示词项 t 的 idf，C 表示我们语料库中的总文档数，$df(t)$ 表示包含词项 t 的文档数量的频率。实现 idf 还有其他各种各样的方法，但是在这里我们将使用这种方法，需要说明的是，不同实现方法最终得出的结果是非常相似的。函数 $f(q_i, CD)$ 给出了语料库文档 CD 中词项 q_i 的频率。$|CD|$ 表示通过字数测量得到的文档 CD 的总长度，$avgdl$ 表示待检索文档的语料库中文档的平均长度。此外，你还会观察到有两个自由参数 k_1 和 b，k_1 的取值范围通常为 [1.2, 2.0]，b 则通常取 0.75。我们将在实际执行中将 k_1 的值设为 1.5。

我们通过以下几个步骤来计算文档的 BM25 得分：

（1）建立一个函数以获得语料库中词项的逆文档频率（IDF）值。

（2）构建一个计算查询文档和语料库文档的 BM25 得分的函数。

（3）为语料库文档和查询文档获取基于词袋的特征。

（4）使用第 1 步中的函数计算语料库文档的平均长度和语料库文档中词项的 IDF 值。

（5）使用第 2 步中的函数计算 BM25 得分、为相关文档排序并为每个查询文档获取前 n 个最相关的文档。

我们从实现一个提取和计算文档语料库中所有词项的逆文档频率的函数开始着手，该函数使用包含词频的词袋特征，然后使用前述公式将其转换为 IDF。如下函数所示：

```
import scipy.sparse as sp

def compute_corpus_term_idfs(corpus_features, norm_corpus):

    dfs = np.diff(sp.csc_matrix(corpus_features, copy=True).indptr)
    dfs = 1 + dfs # to smoothen idf later
    total_docs = 1 + len(norm_corpus)
    idfs = 1.0 + np.log(float(total_docs) / dfs)
    return idfs
```

现在，我们要实现基于查询文件的、对语料库中所有文档的 BM25 得分进行计算的主要函数，并根据文档的 BM25 得分从语料库中检索前 n 个最相关的文档。以下函数实现了 BM25 评分框架：

```
def compute_bm25_similarity(doc_features, corpus_features,
                            corpus_doc_lengths, avg_doc_length,
                            term_idfs, k1=1.5, b=0.75, top_n=3):
```

```
    # get corpus bag of words features
    corpus_features = corpus_features.toarray()
    # convert query document features to binary features
    # this is to keep a note of which terms exist per document
    doc_features = doc_features.toarray()[0]
    doc_features[doc_features >= 1] = 1

    # compute the document idf scores for present terms
    doc_idfs = doc_features * term_idfs
    # compute numerator expression in BM25 equation
    numerator_coeff = corpus_features * (k1 + 1)
    numerator = np.multiply(doc_idfs, numerator_coeff)
    # compute denominator expression in BM25 equation
    denominator_coeff =  k1 * (1 - b +
                               (b * (corpus_doc_lengths /
                                     avg_doc_length)))
    denominator_coeff = np.vstack(denominator_coeff)
    denominator = corpus_features + denominator_coeff
    # compute the BM25 score combining the above equations
    bm25_scores = np.sum(np.divide(numerator,
                                   denominator),
                         axis=1)
    # get top n relevant docs with highest BM25 score
    top_docs = bm25_scores.argsort()[::-1][:top_n]
    top_docs_with_score = [(index, round(bm25_scores[index], 3))
                           for index in top_docs]
    return top_docs_with_score
```

函数里的注释十分简单明了，它们解释了函数如何实现 BM25 的评分功能。简单来说，我们首先计算 BM25 数学表达式中的分子，然后计算其分母。最后，我们将分子除以分母，获得所有语料库文档的 BM25 得分。最后我们按降序排序，并返回前 n 个具有最高 BM25 得分的相关文档。在下面的代码段中，我们将在示例语料库中对函数进行测试，并查看它对每个查询文档的执行情况：

```
# build bag of words based features first
vectorizer, corpus_features = build_feature_matrix(norm_corpus,
                                                   feature_type='frequency')
query_docs_features = vectorizer.transform(norm_query_docs)

# get average document length of the corpus (avgdl)
doc_lengths = [len(doc.split()) for doc in norm_corpus]
avg_dl = np.average(doc_lengths)
# Get the corpus term idfs
corpus_term_idfs = compute_corpus_term_idfs(corpus_features,
                                            norm_corpus)

# analyze document similarity using BM25 framework
In [253]: print 'Document Similarity Analysis using BM25'
     ...: print '='*60
     ...: for index, doc in enumerate(query_docs):
     ...:
     ...:     doc_features = query_docs_features[index]
     ...:     top_similar_docs = compute_bm25_similarity(doc_features,
     ...:                                                corpus_features,
     ...:                                                doc_lengths,
     ...:                                                avg_dl,
     ...:                                                corpus_term_idfs,
     ...:                                                k1=1.5, b=0.75,
     ...:                                                top_n=2)
     ...:     print 'Document',index+1 ,':', doc
     ...:     print 'Top', len(top_similar_docs), 'similar docs:'
     ...:     print '-'*40
     ...:     for doc_index, sim_score in top_similar_docs:
     ...:         print 'Doc num: {} BM25 Score: {}\nDoc: {}'.format(doc_
```

```
                index+1,
    ...:                                    sim_score, toy_corpus[doc_
                                            index])
    ...:         print '-'*40
    ...:     print

Document Similarity Analysis using BM25
============================================================
Document 1 : The fox is definitely smarter than the dog
Top 2 similar docs:
----------------------------------------
Doc num: 8 BM25 Score: 7.334
Doc: The dog is smarter than the fox
----------------------------------------
Doc num: 7 BM25 Score: 3.88
Doc: The fox is quicker than the lazy dog
----------------------------------------

Document 2 : Java is a static typed programming language unlike Python
Top 2 similar docs:
----------------------------------------
Doc num: 5 BM25 Score: 7.248
Doc: Python and Java are popular Programming languages
----------------------------------------
Doc num: 6 BM25 Score: 6.042
Doc: Among Programming languages, both Python and Java are the most used in
Analytics
----------------------------------------

Document 3 : I love to relax under the beautiful blue sky!
Top 2 similar docs:
----------------------------------------
Doc num: 2 BM25 Score: 7.334
Doc: The sky is blue and beautiful
----------------------------------------
Doc num: 1 BM25 Score: 4.984
Doc: The sky is blue
----------------------------------------
```

现在，你可以看出，对于每个查询文档，我们是如何获得与查询文档内容相似的文档的。你可以看到该结果与前几节的结果非常相似，当然了，因为它们都是相似度和排名指标，并且我们的预期就是会返回类似的结果。请注意，相关文件的 BM25 得分越高，文档越相关。不幸的是，我无法在 `nltk` 或 `scikit-learn` 中找到任何成熟的、可扩展的 BM25 排名框架实现方法。但是，在 `gensim.summarization` 包下，`gensim` 似乎有一个 `bm25` 模块，如果你感兴趣的话，可以尝试一下。该模块算法的核心仍然是基于前述过程的，所以应该能够取得不错的效果。

你可以尝试加载更大的文档语料库，并在一些示例查询字符串和示例文档上测试这些函数。事实上，诸如 Solr 和 Elasticsearch 这样的信息检索框架是建立在 Lucene 之上的，Lucene使用这类的排名算法从存储文档的索引中返回相关文档，你也可以使用排名算法构建你自己的搜索引擎！有兴趣的读者可查阅 Elastic 公司的博客文章，网址是 www. elastic. co/blog/found-bm-vs-lucene-default-similarity，该公司是广受欢迎的 Elasticsearch 产品的母公司，该文章向我们展示了 BM25 的表现要远远优于 Lucene 的默认相似度排名方法。

6.7 文档聚类

文档聚类或聚类分析是 NLP 和文本分析中的一个有趣的领域，它应用了无监督的 ML 概念和技术。文档聚类的主要前提类似于文档分类，你从文档的完整语料库开始，并根据文档

的一些独特的特性、属性和特征将它们分为不同的组。文档分类需要预先标记的培训数据来构建模型，然后对文档进行分类。文档聚类则使用无监督的 ML 算法将文档分组成各种类。这些类的特性是相较于和其他类的文档之间，一个类内的文档之间更相似、相互关联更紧密。图 6-3 由 scikit-learn 提供，是一个将数据点基于特征聚类成三个类的示例的可视化展现。

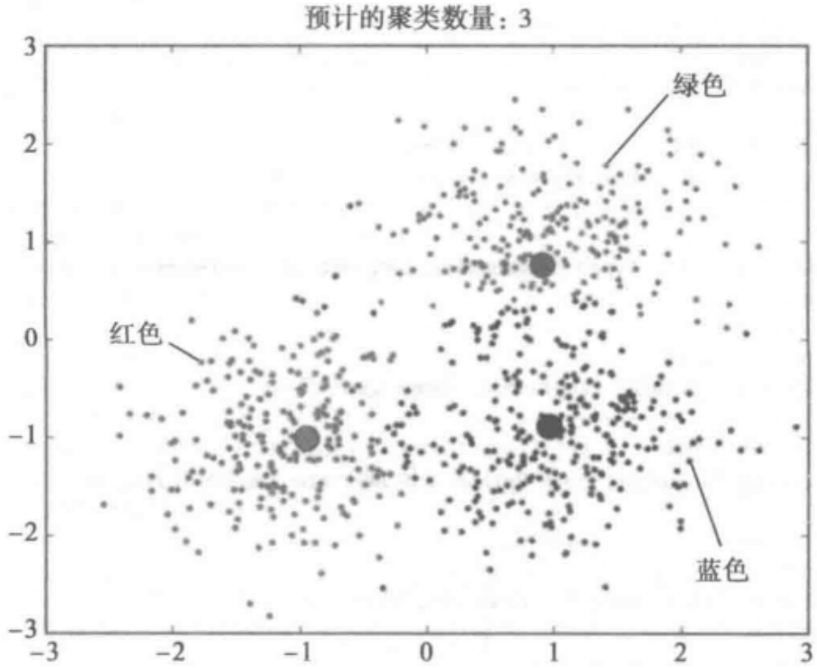


图 6-3 聚类分析结果样例（由 scikit-learn 提供）

图 6-3 中的聚类分析使用不同颜色描绘数据点中的三个聚类。这里要记住一个重要的事情就是聚类是一种无监督的学习技术，从图 6-3 可以清楚地看到，类之间总会有一些重叠，因为并没有一个完美的聚类这样的定义。所有这些技术都是基于数学、启发式算法以及一些聚类生成过程的固有属性，它们从来都不是 100% 完美的。目前，有一些发现聚类的技术或方法，几种流行的聚类算法简要描述如下。

- *层次聚类模型*：这些聚类模型也称为*基于连接的聚类方法*，它基于一个概念，即相似对象在向量空间中应更接近于相关对象，而不是无关对象，即它们距离无关对象更远。聚类通过基于距离的连接对象形成，可以采用*树形图*来可视化。这些模型的输出是完整的、层次结构详尽的聚类。该类模型主要分为凝聚式和分裂式聚类模型。
- *基于质心的聚类模型*：这些模型以这样的方式构建聚类，即每个聚类都有一个中心的、有代表性的成员，它能够代表整个聚类，并具有将特定聚类和其他聚类区分开来的特征。基于质心的聚类模型包含多种算法，如 k-means、k-medoids 算法等，此类算法中我们需要提前设置聚类 k 的数量，并且最小化距离度量（如每个数据点到质心的距离的平方）。这些模型的缺点是你需要提前指定 k 值，而这可能会导致局部极小值，使你无法获得数据的真实聚类表示。
- *基于分布的聚类模型*：这类模型利用概率分布的概念来聚类数据点。它的理念是具有相似分布的对象可以聚类成相同的组或聚类。*高斯混合模型*（Gaussian Mixture Model，GMM）使用诸如预期最大化算法来构建这些聚类。特征、属性相关性和依存关系也可以使用这些模型来捕获，但是这类模型容易过拟合。

- 基于密度的聚类模型：这种聚类模型使用聚集在较密集区域的数据点生成聚类，与较密集区域的数据点相比，其他数据点可能会随机地出现在向量空间的稀疏区域。将这些稀疏区域视为噪声，并作为边界来隔离聚美。该领域的两个流行算法是 DBSCAN 算法和 OPTICS 算法。

最近还出现了一些其他聚类模型，包括 BIRCH 和 CLARANS 算法等。目前，有很多专门研究聚类的书籍和杂志——因为聚类是一个非常有趣且很有价值的话题。我们无法在这里详细介绍每一种算法，所以我们将主要介绍三种不同的聚类算法，并用真实数据说明它们，以便你能更好地理解：

- k-means 聚类。
- 近邻传播（Affinity Propagation，AP）聚类。
- 沃德凝聚层次聚类（Ward's agglomerative hierarchical clustering）。

对于每个算法，我们将介绍其理论概念，正如我们介绍其他算法那样。我们还将通过将每个聚类算法应用于与电影和电影简介相关的一些真实数据来说明每种算法的工作原理。我们还将查看详细的聚类统计数据，并着重关注使用经过验证的算法可视化聚类，因为通常聚类结果很难可视化，而从业人员往往又面临着这一挑战。

6.8 最佳影片聚类分析

我们将使用电影简介作为我们的原始数据，将总共 100 部流行电影进行聚类分析。IMDb 也称为互联网电影数据库（www. imdb. com），是一个在线的数据库，它提供有关电影、电子游戏和电视节目的大量详细信息。它聚集了电影和电视节目的评论以及简介，并有几个精选影片清单。我们感兴趣的影片清单可在网址 www. imdb. com/list/ls055592025/上获取，名为“100 部最佳影片（终极清单）”。我们使用 IMDb 概要和每部电影的描述将这些电影进行聚类。

在我们开始分析之前，我要感谢布兰登·罗斯（Brandon Rose）帮助我获得这个数据，他亲自检索和筛选数据，并给予了一些关于可视化聚类的优秀指标。他对这些数据进行了详尽的聚类分析。如果你感兴趣，你可以获取原始数据，并在他的数据存储中查看他的文档聚类分析，网址为 https://github. com/brandomr/document_cluster，在他的个人博客中也有进一步的详细描述，网址为 http://brandonrose. org。

我们已经从前面提到的存储库中下载了有关前 100 部电影的标题及 IMDb 简介数据。我们解析并清洗了数据，并为在原始数据中缺少简介的几部电影添加了影片介绍。这些简介和电影描述来自维基百科。数据解析完成后，我们将它们存储在数据框中，并将其保存至 `movie_data. csv` 文件，你可以在本章的代码文件中找到该文件。我们将在聚类分析中加载并使用该文件中的数据，首先，我们需要加载并查看电影数据的内容，如下代码所示：

```
import pandas as pd
import numpy as np

# load movie data
movie_data = pd.read_csv('movie_data.csv')

# view movie data
In [256]: print movie_data.head()
```

```
                    Title                                        Synopsis
0             The Godfather  In late summer 1945, guests are gathered...
1  The Shawshank Redemption  In 1947, Andy Dufresne (Tim Robbins),...
2           Schindler's List  The relocation of Polish Jews from...
3                Raging Bull  The film opens in 1964, where an older...
4                 Casablanca  In the early years of World War II...

# print sample movie and its synopsis
In [268]: print 'Movie:', movie_titles[0]
     ...: print 'Movie Synopsis:', movie_synopses[0][:1000]
     ...:
Movie: The Godfather
Movie Synopsis: In late summer 1945, guests are gathered for the wedding
reception of Don Vito Corleone's daughter Connie (Talia Shire) and Carlo
Rizzi (Gianni Russo). Vito (Marlon Brando), the head of the Corleone Mafia
family, is known to friends and associates as "Godfather." He and Tom Hagen
(Robert Duvall), the Corleone family lawyer, are hearing requests for favors
because, according to Italian tradition, "no Sicilian can refuse a request
on his daughter's wedding day." One of the men who asks the Don for a favor
is Amerigo Bonasera, a successful mortician and acquaintance of the Don,
whose daughter was brutally beaten by two young men because she refused
their advances; the men received minimal punishment. The Don is disappointed
in Bonasera, who'd avoided most contact with the Don due to Corleone's
nefarious business dealings. The Don's wife is godmother to Bonasera's
shamed daughter, a relationship the Don uses to extract new loyalty from the
undertaker. The Don agrees to have his men punish
```

你可以看到我们已有了电影标题和相应的内容简介，我们将其加载到数据框中，然后存储在变量里。前面的输出也给出了一个电影样本及其部分摘要。我们的核心思路是使用这些电影简介作为原始输入来聚类电影并完成分组。我们将从这些简介中提取特征，并使用无监督的学习算法将它们进行聚类。电影标题则是用于表征数据，当我们想要可视化并展示聚类及其统计信息时，这些电影标题将会很有用。聚类算法的数据输入是从电影简介中提取的特征。在我们介绍每个聚类算法之前，将执行与前几节类似的规范化和特征提取过程：

```
from normalization import normalize_corpus
from utils import build_feature_matrix

# normalize corpus
norm_movie_synopses = normalize_corpus(movie_synopses,
                                       lemmatize=True,
                                       only_text_chars=True)

# extract tf-idf features
vectorizer, feature_matrix = build_feature_matrix(norm_movie_synopses,
                                                  feature_type='tfidf',
                                                  min_df=0.24, max_df=0.85,
                                                  ngram_range=(1, 2))
# view number of features
In [275]: print feature_matrix.shape
(100, 307)

# get feature names
feature_names = vectorizer.get_feature_names()
# print sample features
In [277]: print feature_names[:20]
[u'able', u'accept', u'across', u'act', u'agree', u'alive', u'allow',
u'alone', u'along', u'already', u'although', u'always', u'another',
u'anything', u'apartment', u'appear', u'approach', u'arm', u'army',
u'around']
```

我们在规范后的文本中保留了文本标识，并提取了基于 TF-IDF 的一元分词和二元分词

特征，以保证每个特征至少在25%的文档中出现，以及至多85%的文档使用词项 `min_df` 和 `max_df`。可以看出，对于100部电影，我们共有100行数据并且每部电影有307个特征，一些示例特征也显示在了上述代码段中。接下来，既然已经准备好了特征和文档，我们将开始进行聚类分析。

6.8.1 k-means 聚类

k-means 聚类算法是一种基于质心的聚类模型，它尝试将数据聚类成等方差的组或聚类。该算法尝试将标准或度量——惯量（inertia）最小化，惯量也称为聚类内平方和。这种算法的一个主要缺点是，和其他所有基于质心的聚类模型一样，它需要事先指定聚类 k 的数量。该算法可能是目前最流行的聚类算法，由于其易用性以及可扩展大量数据而被广泛使用。

现在，我们可以使用数学表达式正式地定义 k-means 聚类算法。假设我们有一个具有 N 个数据点或样本的数据集 X，并且我们希望将它们分组为 K 个聚类，其中 K 是一个用户指定的参数。k-means 聚类算法会将 N 个数据点分离为 K 个不相交的分离聚类 C_k，并且每一个聚类均可以被样例聚类的平均值描述。这些平均值就是聚类的质心 μ_k，它们不必受质心必须是 X 的 N 个样本中的实际数据点这个条件的限制。该算法选择这些质心并以这样一种方式构建聚类——即惯量或聚类内平方和需要最小化，其数学表达式为：

$$\min \sum_{i=1}^{K} \sum_{x_n \in C_i} \| x_n - \mu_i \|^2$$

聚类 C_i 和质心 μ_i 中 $i \in \{1,2,\cdots,k\}$。如果你是一个算法爱好者，这个优化将会是你们喜爱的 NP 难题。劳埃德（Lloyd）算法是解决这个问题的一个方案，它是包含以下步骤的迭代过程。

（1）通过从数据集 X 中选取 k 个随机样本，选择 k 个初始质心 μ_k。

（2）通过将每个数据点或样本分配到离其最近的质心点来更新聚类。在数学上，我们可以将其表示为 $C_k = \{x_n : \| x_n - \mu_k \| \leqslant \text{all} \| x_n - \mu_l \| \}$，其中 C_k 表示聚类。

（3）根据从步骤2获得的每个聚类的新聚类数据点重新计算并更新聚类。在数学上，这可以表示为

$$\mu_k = \frac{1}{C_k} \sum_{x_n \in C_k} x_n$$

其中 μ_k 表示质心。

以迭代方式重复上述步骤，直到步骤2和步骤3得到的结果不再发生变化。使用这种方法需要注意一点，那就是即使优化过程确保是收敛的，它仍然可能存在局部最小值，因此，在实际使用中，该算法会在多个时期和不同迭代次数中运行多次，并且如果需要的话，可以从多个迭代结果中取平均值。收敛和局部最小值的发生高度依赖于步骤1中最开始质心的初始化。一种方式是进行多次迭代，并进行多次随机初始化，然后取平均值。另一种方法是使用 `scikit-learn` 中的 `kmeans++` 方案，它在初始化质心时将质心彼此远离，这被证明是十分有效的。现在我们将使用 k-means 算法聚类前面的电影数据，如下列代码段所示：

```
from sklearn.cluster import KMeans
# define the k-means clustering function
def k_means(feature_matrix, num_clusters=5):
```

```
    km = KMeans(n_clusters=num_clusters,
                max_iter=10000)
    km.fit(feature_matrix)
    clusters = km.labels_
    return km, clusters
# set k = 5, lets say we want 5 clusters from the 100 movies
num_clusters = 5

# get clusters and assigned the cluster labels to the movies
km_obj, clusters = k_means(feature_matrix=feature_matrix,
                           num_clusters=num_clusters)
movie_data['Cluster'] = clusters
```

该代码段使用之前实现的 k-means 函数，根据电影简介中的 TF-IDF 特征对电影进行聚类，我们使用聚类分析的结果为每个电影分配聚类标签，并将其存储在'Cluster'列的movie_data数据帧中。你可以看到，在分析中我们将 k 设置成了 5。以下代码段可以查看 5 个聚类的电影总数：

```
In [284]: from collections import Counter
     ...: # get the total number of movies per cluster
     ...: c = Counter(clusters)
     ...: print c.items()
[(0, 29), (1, 5), (2, 21), (3, 15), (4, 30)]
```

你可以看到，正如我们前面所说的，共有 5 个聚类标签，从 0 到 4，并且对于每个标签都有一些电影属于该聚类，该聚类中的电影数为上一列表中每个数组中的第二个元素。除了仅仅看到聚类计数，我们能够做更多的事情吗？当然可以！接下来，我们将会定义一些函数来提取详细的聚类分析信息，显示结果并可视化聚类。我们首先定义一个函数，从聚类分析中提取重要信息：

```
def get_cluster_data(clustering_obj, movie_data,
                     feature_names, num_clusters,
                     topn_features=10):

    cluster_details = {}
    # get cluster centroids
    ordered_centroids = clustering_obj.cluster_centers_.argsort()[:, ::-1]
    # get key features for each cluster
    # get movies belonging to each cluster
    for cluster_num in range(num_clusters):
        cluster_details[cluster_num] = {}
        cluster_details[cluster_num]['cluster_num'] = cluster_num
        key_features = [feature_names[index]
                        for index
                        in ordered_centroids[cluster_num, :topn_features]]
        cluster_details[cluster_num]['key_features'] = key_features

        movies = movie_data[movie_data['Cluster'] == cluster_num]['Title'].
        values.tolist()
        cluster_details[cluster_num]['movies'] = movies

    return cluster_details
```

上述函数非常简单明了。它所做的是提取每个聚类的关键特征，这些特征对于定义聚类很重要。它还能够检索每个聚类的电影标题，并将所有内容存储在字典中。

接下来，我们将定义一个使用此数据结构并能够清晰展示结果的函数：

```
def print_cluster_data(cluster_data):
    # print cluster details
    for cluster_num, cluster_details in cluster_data.items():
```

```
        print 'Cluster {} details:'.format(cluster_num)
        print '-'*20
        print 'Key features:', cluster_details['key_features']
        print 'Movies in this cluster:'
        print ', '.join(cluster_details['movies'])
        print '='*40
```

在分析 k-means 聚类算法结果之前，我们还需要定义一个函数来实现聚类可视化。如果你还记得的话，我们之前提到有关可视化聚类的挑战。这是因为我们要处理的是多维特征空间和非结构化文本数据。如果直接可视化数字特征向量，那么这可能对读者是没有任何意义的。目前有一些技术，如主成分分析（PCA）或多维缩放（MDS）可以减少维度，我们可以在二维或三维图中可视化这些聚类。在具体实现中，我们将使用 MDS 来可视化聚类。

MDS 是一种减少非线性维度的方法，可以在低维度系统中更好地显现结果。它的核心思想是使用一个距离矩阵，以便捕获各种数据点之间的距离。在这里，我们将使用余弦相似度。MDS 尝试使用向量空间中的高维特征构建数据的低维表示，这样使得在高维特征空间中使用余弦相似度获得的各类数据点的距离在使用较低维表示时仍然大致相同。

MDS 的 scikit-learn 实现有两类算法：度量和非度量算法。在这里，我们将使用度量算法，因为我们将使用基于余弦相似度的距离度量来构建各种电影之间的输入相似度矩阵。在数学上，MDS 可以定义为：假设 S 是在特征矩阵上使用余弦相似度获得的各种数据点（电影）之间的相似度矩阵，X 是 n 个输入数据点（电影）的坐标，差异性（disparity）由 $\hat{d}_{ij} = t(S_{ij})$ 表示，其通常是相似度值的一些最佳变换，甚至可能是原始相似值本身。MDS 的目标函数称为*应力函数*，其定义为 $\text{sum}_{i<j} d_{ij}(X) - \hat{d}_{ij}(X)$。我们使用以下函数实现基于 MDS 的聚类可视化：

```
import matplotlib.pyplot as plt
from sklearn.manifold import MDS
from sklearn.metrics.pairwise import cosine_similarity
import random
from matplotlib.font_manager import FontProperties

def plot_clusters(num_clusters, feature_matrix,
                  cluster_data, movie_data,
                  plot_size=(16,8)):
    # generate random color for clusters
    def generate_random_color():
        color = '#%06x' % random.randint(0, 0xFFFFFF)
        return color
    # define markers for clusters
    markers = ['o', 'v', '^', '<', '>', '8', 's', 'p', '*', 'h', 'H', 'D', 'd']
    # build cosine distance matrix
    cosine_distance = 1 - cosine_similarity(feature_matrix)
    # dimensionality reduction using MDS
    mds = MDS(n_components=2, dissimilarity="precomputed",
              random_state=1)
    # get coordinates of clusters in new low-dimensional space
    plot_positions = mds.fit_transform(cosine_distance)
    x_pos, y_pos = plot_positions[:, 0], plot_positions[:, 1]
    # build cluster plotting data
cluster_color_map = {}
cluster_name_map = {}
for cluster_num, cluster_details in cluster_data.items():
    # assign cluster features to unique label
    cluster_color_map[cluster_num] = generate_random_color()
    cluster_name_map[cluster_num] = ', '.join(cluster_details['key_
    features'][:5]).strip()
# map each unique cluster label with its coordinates and movies
```

```
cluster_plot_frame = pd.DataFrame({'x': x_pos,
                                   'y': y_pos,
                                   'label': movie_data['Cluster'].
                                   values.tolist(),
                                   'title': movie_data['Title'].values.
                                   tolist()
                                    })
grouped_plot_frame = cluster_plot_frame.groupby('label')
# set plot figure size and axes
fig, ax = plt.subplots(figsize=plot_size)
ax.margins(0.05)
# plot each cluster using co-ordinates and movie titles
for cluster_num, cluster_frame in grouped_plot_frame:
    marker = markers[cluster_num] if cluster_num < len(markers) \
             else np.random.choice(markers, size=1)[0]
    ax.plot(cluster_frame['x'], cluster_frame['y'],
            marker=marker, linestyle='', ms=12,
            label=cluster_name_map[cluster_num],
            color=cluster_color_map[cluster_num], mec='none')
    ax.set_aspect('auto')
    ax.tick_params(axis= 'x', which='both', bottom='off', top='off',
                   labelbottom='off')
    ax.tick_params(axis= 'y', which='both', left='off', top='off',
                   labelleft='off')
fontP = FontProperties()
fontP.set_size('small')
ax.legend(loc='upper center', bbox_to_anchor=(0.5, -0.01),
fancybox=True,
          shadow=True, ncol=5, numpoints=1, prop=fontP)
#add labels as the film titles
for index in range(len(cluster_plot_frame)):
    ax.text(cluster_plot_frame.ix[index]['x'],
            cluster_plot_frame.ix[index]['y'],
            cluster_plot_frame.ix[index]['title'], size=8)
# show the plot
plt.show()
```

函数的代码比较长，但是注释相当清楚地说明了函数的每一步。我们首先使用文档之间的余弦相似度建立相似度矩阵，获得余弦距离，然后使用 MDS 将高维特征空间转换为二维。然后，我们使用 `matplotlib` 绘制聚类结果，并使用一些必要的格式来使展现效果更好。该函数是一个通用函数，可用于任何具有动态聚类数量的聚类算法。在显示结果中，每个聚类都会拥有自己的颜色、符号和标签，以区分图例框中的特征值。实际的可视化结果将绘制每部电影及其具有自己颜色和符号的聚类标签。

现在，我们可以使用上述函数来分析 k-means 算法的聚类结果。以下代码段描述了 k-means 聚类的详细分析结果：

```
# get clustering analysis data
cluster_data =  get_cluster_data(clustering_obj=km_obj, movie_data=movie_
data,
                                 feature_names=feature_names, num_
                                 clusters=num_clusters,
                                 topn_features=5)

# print clustering analysis results
In [294]: print_cluster_data(cluster_data)

Cluster 0 details:
------------------
Key features: [u'car', u'police', u'house', u'father', u'room']
Movies in this cluster:
Psycho, Sunset Blvd., Vertigo, West Side Story, E.T. the Extra-Terrestrial,
2001: A Space Odyssey, The Silence of the Lambs, Singin' in the Rain, It's
```

```
a Wonderful Life, Some Like It Hot, Gandhi, To Kill a Mockingbird, Butch
Cassidy and the Sundance Kid, The Exorcist, The French Connection, It
Happened One Night, Rain Man, Fargo, Close Encounters of the Third Kind,
Nashville, The Graduate, American Graffiti, Pulp Fiction, The Maltese
Falcon, A Clockwork Orange, Rebel Without a Cause, Rear Window, The Third
Man, North by Northwest
=========================================
Cluster 1 details:
------------------
Key features: [u'water', u'attempt', u'cross', u'death', u'officer']
Movies in this cluster:
Chinatown, Apocalypse Now, Jaws, The African Queen, Mutiny on the Bounty
=========================================
Cluster 2 details:
------------------
Key features: [u'family', u'love', u'marry', u'war', u'child']
Movies in this cluster:
The Godfather, Gone with the Wind, The Godfather: Part II, The Sound of
Music, A Streetcar Named Desire, The Philadelphia Story, An American in
Paris, Ben-Hur, Doctor Zhivago, High Noon, The Pianist, Goodfellas, The
King's Speech, A Place in the Sun, Out of Africa, Terms of Endearment,
Giant, The Grapes of Wrath, Wuthering Heights, Double Indemnity, Yankee
Doodle Dandy
=========================================
Cluster 3 details:
------------------
Key features: [u'apartment', u'new', u'woman', u'york', u'life']
Movies in this cluster:
Citizen Kane, Titanic, 12 Angry Men, Rocky, The Best Years of Our Lives, My
Fair Lady, The Apartment, City Lights, Midnight Cowboy, Mr. Smith Goes to
Washington, Annie Hall, Good Will Hunting, Tootsie, Network, Taxi Driver
=========================================
Cluster 4 details:
------------------
Key features: [u'kill', u'soldier', u'men', u'army', u'war']
Movies in this cluster:
The Shawshank Redemption, Schindler's List, Raging Bull, Casablanca, One
Flew Over the Cuckoo's Nest, The Wizard of Oz, Lawrence of Arabia, On the
Waterfront, Forrest Gump, Star Wars, The Bridge on the River Kwai, Dr.
Strangelove or: How I Learned to Stop Worrying and Love the Bomb, Amadeus,
The Lord of the Rings: The Return of the King, Gladiator, From Here to
Eternity, Saving Private Ryan, Unforgiven, Raiders of the Lost Ark, Patton,
Braveheart, The Good, the Bad and the Ugly, The Treasure of the Sierra
Madre, Platoon, Dances with Wolves, The Deer Hunter, All Quiet on the
Western Front, Shane, The Green Mile, Stagecoach
=========================================

# visualize the clusters
In [295]: plot_clusters(num_clusters=num_clusters,
     ...:               feature_matrix=feature_matrix,
     ...:               cluster_data=cluster_data,
     ...:               movie_data=movie_data,
     ...:               plot_size=(16,8))
```

上面的输出显示了每个聚类及其中电影的关键特征，你可以在图 6-4 的可视化结果中看到相同的内容（该图包含很多内容，如果图中文字太小，请查看 kmeans_clustering.png 文件，本章的代码文件也可供查阅）。每个聚类由其主题描绘，主题通过最主要的特征定义该聚类。你可以看到流行的电影，如《教父》、《教父 2》和《宾虚》等电影在同一个类中，它们都涉及“家庭”“爱”“战争”等。诸如《星球大战》、《指环王》、《鹿猎人》、《角斗士》、《阿甘正传》等电影与“杀人”“士兵”“军队”“战争”等主题的电影聚类在一起。考虑到用于聚类的数据只是每部电影简介的几个段落，我们得出的结果绝对会非常有趣。仔细观察聚类和可视化结果，你能注意到其他有趣的模式吗？

图 6-4 k-means 聚类算法应用于 IMDb 电影数据上的可视化结果

6.8.2 近邻传播聚类

k-means 算法虽然非常流行，但是有一个缺点，就是用户必须预先定义聚类数量。如果实际情况中有更多或更少的聚类呢？我们有一些检查聚类质量并计算最佳 k 值的方法。有兴趣的读者可以查看肘部法则（elbow method）和轮廓系数（silhouette coefficient），它们是确定最佳 k 值的流行方法。在这里，我们将讨论另一种算法，它基于数据的固有属性来构建聚类，无需对聚类数量进行任何预先假设。近邻传播（Affinity Propagation，AP）算法基于待聚类各数据点中的“消息传递”的概念，并且不需要关于聚类数量的预先假设。

AP 算法通过在数据点之间传递消息直至达到收敛而创建聚类。整个数据集由少数作为样本代表的样本表示。这些样本类似于你从 k-means 或 k-medoids 算法中获得的质心。在数据点之间发送的消息表示一个数据点是否适合作为样本代表表示其他数据点。它在每次迭代中不断更新直到收敛，最终的样本是每个聚类的代表。请记住，这种方法的一个缺点是计算强度较大，因为消息在整个数据集之间的每一对数据点之间传递，在应用于大量数据时，可能需要相当长的时间才能达到收敛。

现在，我们可以定义 AP 算法的步骤（由维基百科和 `scikit-learn` 提供）。假设我们有一个具有 n 个数据点的数据集 X，$X=\{x_1,x_2,\cdots,x_n\}$，假设 $\mathrm{sim}(x,y)$ 是衡量两点 x 和 y 之间相似度的相似度函数。在这个过程中，我们将再次使用余弦相似度。AP 算法通过执行两个消息传递步骤完成迭代过程，如下所示：

（1）发送吸引度更新，其数据表达式为

$$r(i,k) \leftarrow \mathrm{sim}(i,k) - \max_{k' \neq k}\{a(i,k') + \mathrm{sim}(i,k')\}$$

其中吸引度矩阵为 R，$r(i,k)$ 是与其他数据点相比 x_k 表示可以代表 x_i 程度的测量值。

（2）然后，发送归属度更新，其数据表达式为

$$a(i,k) \leftarrow \min\Big(0, r(k,k) + \sum_{i' \notin \{i,k\}} \max(0, r(i',k))\Big),\ i \neq k$$

$i=k$ 时，归属度表示为

$$a(k,k) \leftarrow \sum_{i' \neq k} \max(0, r(i',k))$$

其中归属度矩阵为 A，而 $a(i,k)$ 表示在考虑了其他所有的点选取 x_k 作为样本代表的基础之上，对于 x_i 来说选取 x_k 作为样本代表的契合程度。

这两个步骤不断迭代，直至收敛。以下函数可以实现 AP 算法，它的输入是一个特征矩阵，能够根据样本特征和其他样本情况返回每个样本的必要聚类：

```
from sklearn.cluster import AffinityPropagation

def affinity_propagation(feature_matrix):
    sim = feature_matrix * feature_matrix.T
    sim = sim.todense()
    ap = AffinityPropagation()
    ap.fit(sim)
    clusters = ap.labels_
    return ap, clusters
```

现在，我们将使用此函数对电影数据按照影片简介进行分组，然后显示每个聚类中的电影数量以及此算法形成的聚类总数：

```
# get clusters using affinity propagation
ap_obj, clusters = affinity_propagation(feature_matrix=feature_matrix)
movie_data['Cluster'] = clusters

# get the total number of movies per cluster
In [299]: c = Counter(clusters)
     ...: print c.items()
[(0, 5), (1, 6), (2, 12), (3, 6), (4, 2), (5, 7), (6, 10), (7, 7), (8, 4),
(9, 8), (10, 3), (11, 4), (12, 5), (13, 7), (14, 4), (15, 3), (16, 7)]

# get total clusters
In [300]: total_clusters = len(c)
     ...: print 'Total Clusters:', total_clusters
Total Clusters: 17
```

从前面的结果可以看出，AP 算法在我们的电影数据中共创建了 17 个聚类，包含 100 部电影。每个聚类都有最少 2 部、最多 12 部电影。接下来，与之前在 k-means 聚类分析中的做法类似，我们将提取详细的聚类信息、显示聚类统计信息并可视化聚类：

```
# get clustering analysis data
cluster_data =  get_cluster_data(clustering_obj=ap_obj, movie_data=movie_
data,
                                 feature_names=feature_names, num_
                                 clusters=total_clusters,
                                 topn_features=5)

# print clustering analysis results
In [302]: print_cluster_data(cluster_data)
     ...:
Cluster 0 details:
--------------------
Key features: [u'able', u'always', u'cover', u'end', u'charge']
Movies in this cluster:
The Godfather, The Godfather: Part II, Doctor Zhivago, The Pianist,
Goodfellas
========================================
Cluster 1 details:
--------------------
Key features: [u'alive', u'accept', u'around', u'agree', u'attack']
Movies in this cluster:
Casablanca, One Flew Over the Cuckoo's Nest, Titanic, 2001: A Space Odyssey,
The Silence of the Lambs, Good Will Hunting
========================================
Cluster 2 details:
--------------------
```

```
Key features: [u'apartment', u'film', u'final', u'fall', u'due']
Movies in this cluster:
The Shawshank Redemption, Vertigo, West Side Story, Rocky, Tootsie,
Nashville, The Graduate, The Maltese Falcon, A Clockwork Orange, Taxi
Driver, Rear Window, The Third Man
========================================
Cluster 3 details:
----------
Key features: [u'arrest', u'film', u'evening', u'final', u'fall']
Movies in this cluster:
The Wizard of Oz, Psycho, E.T. the Extra-Terrestrial, My Fair Lady, Ben-Hur,
Close Encounters of the Third Kind
========================================
Cluster 4 details:
----------
Key features: [u'become', u'film', u'city', u'army', u'die']
Movies in this cluster:
12 Angry Men, Mr. Smith Goes to Washington
========================================
Cluster 5 details:
----------
Key features: [u'behind', u'city', u'father', u'appear', u'allow']
Movies in this cluster:
Forrest Gump, Amadeus, Gladiator, Braveheart, The Exorcist, A Place in the
Sun, Double Indemnity
========================================
Cluster 6 details:
----------
Key features: [u'body', u'allow', u'although', u'city', u'break']
Movies in this cluster:
Schindler's List, Gone with the Wind, Lawrence of Arabia, Star Wars, The
Lord of the Rings: The Return of the King, From Here to Eternity, Raiders of
the Lost Ark, The Best Years of Our Lives, The Deer Hunter, Stagecoach
========================================
Cluster 7 details:
----------
Key features: [u'brother', u'bring', u'close', u'although', u'car']
Movies in this cluster:
Gandhi, Unforgiven, To Kill a Mockingbird, The Good, the Bad and the Ugly,
Butch Cassidy and the Sundance Kid, High Noon, Shane
========================================
Cluster 8 details:
----------
Key features: [u'child', u'everyone', u'attempt', u'fall', u'face']
Movies in this cluster:
Chinatown, Jaws, The African Queen, Mutiny on the Bounty
========================================
Cluster 9 details:
----------
Key features: [u'continue', u'bring', u'daughter', u'break', u'allow']
Movies in this cluster:
The Bridge on the River Kwai, Dr. Strangelove or: How I Learned to Stop
Worrying and Love the Bomb, Apocalypse Now, Saving Private Ryan, Patton,
Platoon, Dances with Wolves, All Quiet on the Western Front
========================================
Cluster 10 details:
----------
Key features: [u'despite', u'drop', u'family', u'confront', u'drive']
Movies in this cluster:
The Treasure of the Sierra Madre, City Lights, Midnight Cowboy
========================================
Cluster 11 details:
----------
Key features: [u'discover', u'always', u'feel', u'city', u'act']
Movies in this cluster:
Raging Bull, It Happened One Night, Rain Man, Rebel Without a Cause
========================================
Cluster 12 details:
```

```
--------------------
Key features: [u'discuss', u'alone', u'drop', u'business', u'consider']
Movies in this cluster:
Singin' in the Rain, An American in Paris, The Apartment, Annie Hall,
Network
========================================
Cluster 13 details:
--------------------
Key features: [u'due', u'final', u'day', u'ever', u'eventually']
Movies in this cluster:
On the Waterfront, It's a Wonderful Life, Some Like It Hot, The French
Connection, Fargo, Pulp Fiction, North by Northwest
========================================
Cluster 14 details:
--------------------
Key features: [u'early', u'able', u'end', u'charge', u'allow']
Movies in this cluster:
A Streetcar Named Desire, The King's Speech, Giant, The Grapes of Wrath
========================================
Cluster 15 details:
--------------------
Key features: [u'enter', u'eventually', u'cut', u'accept', u'even']
Movies in this cluster:
The Philadelphia Story, The Green Mile, American Graffiti
========================================
Cluster 16 details:
--------------------
Key features: [u'far', u'allow', u'apartment', u'anything', u'car']
Movies in this cluster:
Citizen Kane, Sunset Blvd., The Sound of Music, Out of Africa, Terms of
Endearment, Wuthering Heights, Yankee Doodle Dandy
========================================

# visualize the clusters
In [304]: plot_clusters(num_clusters=num_clusters, feature_matrix=feature_
matrix,
     ...:               cluster_data=cluster_data, movie_data=movie_data,
     ...:               plot_size=(16,8))
```

前面的输出显示了不同聚类的内容及其可视化结果。如果图 6-5 中的文字太小，你也可以参考 affinity_prop_clustering.png 文件，它的分辨率较高。从结果中可以看到，我们现在共有 17 个聚类，你还可以看出，在 k-means 聚类分析中类似的电影在这里也在相似的聚类里，但是这里也有一些显著的差异——许多电影现在属于新的聚类。这些聚类结果是否比以前的更好呢？这很大程度上取决于个人主观判断，由于我还没有全部看完这些电影，所以，亲爱的读者，我决定把这个问题留给你！在这里，需要注意一个重要的事情，那就是每个聚类的样本或质心中的几个关键字可能并不总是能够描述该聚类的本质或主题，有个好的解决办法是在每个聚类上构建主题模型，并查看你能从每个聚类中提取的主题，从而更好地表示每个聚类（可以看出，这是如何将各种文本分析技术结合在一起的另一个例子）。

6.8.3 沃德凝聚层次聚类

层次聚类系列算法与我们讨论的其他聚类模型有点不同。层次聚类尝试通过合并或拆分聚类来构建嵌套层次结构。层次聚类有两个主要的策略。

- 凝聚：这些算法遵循自下而上的方法，最初所有数据点都属于自己的单个聚类，然后从这个底层开始，我们将聚类合并在一起，准备向上构建聚类的层次结构。
- 分裂：这些算法遵循自上而下的方法，最初所有的数据点都属于同一个巨大的聚类，然后我们逐渐向下开始递归分割，从而产生自上而下的聚类层次结构。

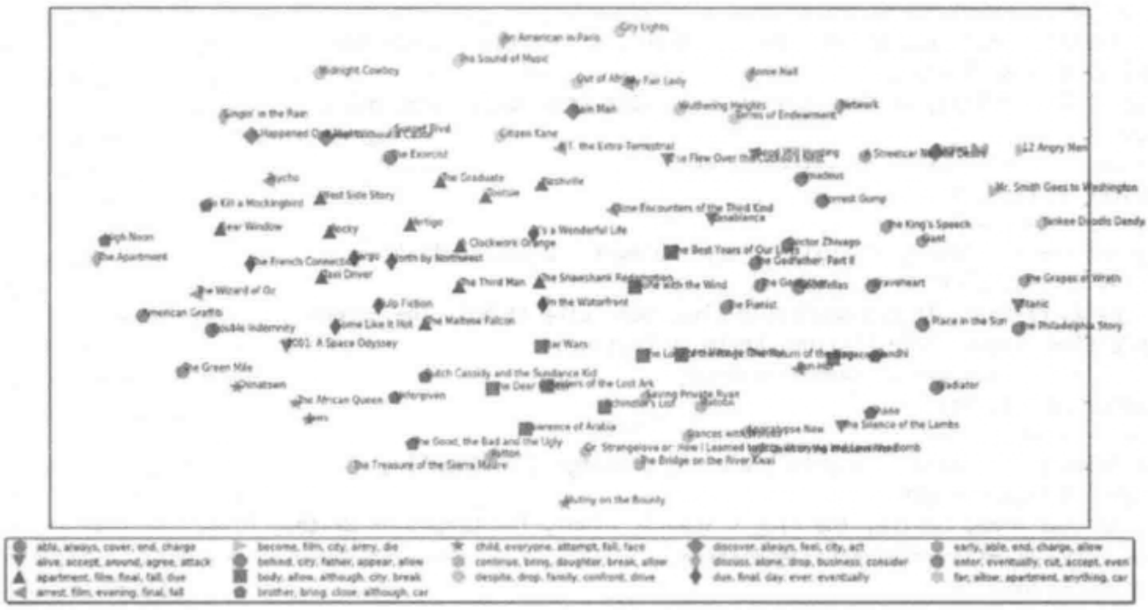


图6-5 AP聚类算法应用于IMDb电影数据上的可视化结果

通常，层次聚类使用一个贪心算法（greedy algorithm）进行合并和拆分，聚类层次结构的最终结果可以可视化为树状结构，称为树形图。图6-6显示了一个使用凝聚层次聚类方法为文档样本构建的树状图示例。

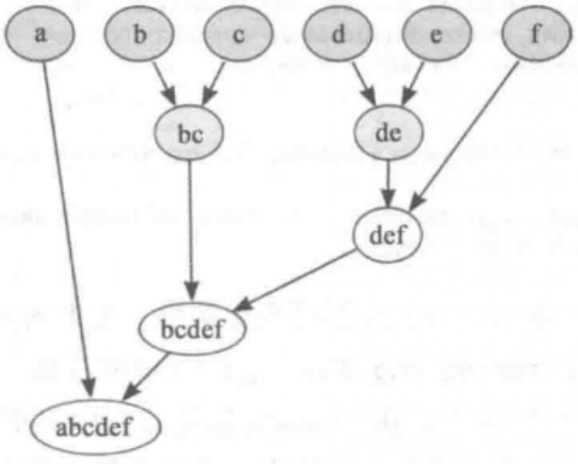


图6-6 凝聚层次聚类示意图

图6-6清楚地显示了六个独立的数据点如何从六个聚类开始，然后按照自下而上的方法逐步分组。在本节中，我们将使用凝聚层次聚类算法。在凝聚聚类中，为了决定在单个数据点集合开始时，应合并哪些聚类，我们需要两个度量标准。

- 距离度量：测量数据点之间的相似度或不相似度。在这里，我们将使用余弦距离/相似度。
- 合并标准：确定用于聚类合并策略的度量。在这里，我们将使用沃德的方法。

沃德的连接标准将所有聚类中的平方差的总和最小化，即方差最小化方法。这也称为沃德最小方差方法，最初由J·沃德（J. Ward）提出。其核心思想是使用诸如两点之间的L2范数距离这样的目标函数，来最小化每个聚类内的方差。我们可以使用公式计算每对数据点之间的初始聚类距离

$$d_{ij} = d(\{C_i, C_j\}) = \| C_i - C_j \|^2$$

其中初始C_i表示具有一个文档的聚类i，在每次迭代中，我们寻找合并之后的方差数值增加

最小的聚类对。如前面公式所示的加权平方欧几里得距离或 L2 范数就足以满足该算法的需求。我们使用余弦相似度来计算数据集中每对电影之间的余弦距离。以下函数实现了沃德凝聚层次聚类：

```
from scipy.cluster.hierarchy import ward, dendrogram

def ward_hierarchical_clustering(feature_matrix):

    cosine_distance = 1 - cosine_similarity(feature_matrix)
    linkage_matrix = ward(cosine_distance)
    return linkage_matrix
```

想要查看层次聚类的结果，我们需要使用前面的连接矩阵绘制一个树形图，执行以下函数可以从层次聚类连接矩阵中构建和绘制树形图：

```
def plot_hierarchical_clusters(linkage_matrix, movie_data, figure_
size=(8,12)):
    # set size
    fig, ax = plt.subplots(figsize=figure_size)
    movie_titles = movie_data['Title'].values.tolist()
    # plot dendrogram
    ax = dendrogram(linkage_matrix, orientation="left", labels=movie_titles)
    plt.tick_params(axis= 'x',
                    which='both',
                    bottom='off',
                    top='off',
                    labelbottom='off')
    plt.tight_layout()
    plt.savefig('ward_hierachical_clusters.png', dpi=200)
```

现在，我们已经准备好对电影数据进行层次聚类了！以下代码段显示了沃德聚类的执行过程：

```
In [307]:# build ward's linkage matrix
     ...:linkage_matrix = ward_hierarchical_clustering(feature_matrix)
     ...: # plot the dendrogram
     ...: plot_hierarchical_clusters(linkage_matrix=linkage_matrix,
     ...:                            movie_data=movie_data,
     ...:                            figure_size=(8,10))
```

图 6-7 中的树形图显示了聚类分析结果。图中颜色表示有三个主要的聚类，它们可以进一步细分为包含层次结构的、粒度更细的聚类。（如果你无法看清图中的文字，请查看本章代码文件中提供的文件 ward_hierachical_clusters. png）。你可以看出图中显示的结果与之前聚类算法的结果有很多相似之处。

图中绿色的电影，如《夺宝奇兵》、《指环王》、《星球大战》、《教父》、《教父 2》、《低俗小说》、《发条橙》和《野战排》绝对是一些顶级的电影，实际上它们都是经典的行动、冒险、战争和基于犯罪的流派。

图中红色的电影包括喜剧电影，如《城市之光》、《公寓春光》和《窈窕淑女》，以及几部剧情类电影，包括《叛舰喋血记》、《十二怒汉》、《安妮·霍尔》、《午夜牛郎》、《泰坦尼克号》和《一个美国人在巴黎 》，其中几部也有浪漫的情节。有一些电影甚至是音乐剧，包括《胜利之歌》、《一个美国人在巴黎》、《雨中曲》和《窈窕淑女》。只需要电影简介，我们的算法就能够将具有相似属性或相同类型的电影聚类在一起，这真是十分有趣的事情！

图中蓝色的电影也给出了类似的结果，其中《勇敢的心》和《角斗士》是动作、剧情和战争经典影片。还有一些剧情、浪漫和传记类的经典电影，如《音乐之声》、《呼啸山

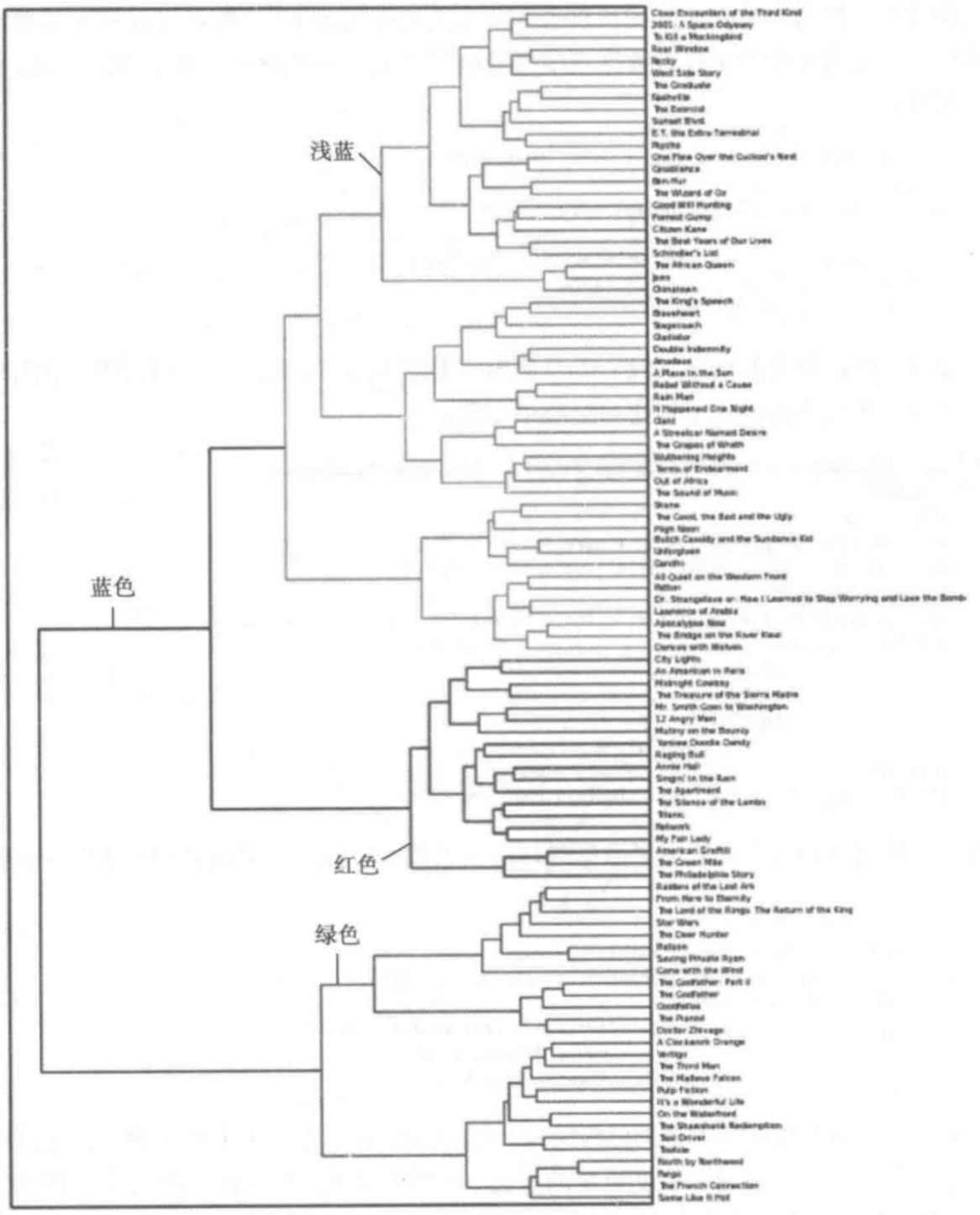


图 6-7 IMDb 电影数据的沃德聚类树形图

庄》、《母女情深》和《走出非洲》。在树形图的顶部，是与科幻有关的电影，如《2001 太空漫游》、《第三类接触》以及《外星人 E. T》，它们彼此十分类似。

你能找到更有趣的地方吗？你认为哪些电影不属于同一个聚类？我们能建立聚类效果更好的群集吗？我们可以根据电影聚类推荐类似的电影吗？这是一些可供探讨的有趣问题，我将把这些问题留给你们去进一步探索。

6.9 小结

我衷心地赞赏你一直努力地和我们一起学完本章的内容。在本章中，我们介绍了很多知识，包括无监督机器学习领域的几个课题，它们虽然有一定挑战性，但是非常有趣。你现在

知道了如何计算文本相似度，并且了解了各种距离测量值和度量。我们还研究了与距离度量和测量值相关的重要概念，以及一个测量值可以作为度量的条件。我们探讨了与无监督 ML 相关的概念，并了解了如何将这些技术融合到文档聚类中。各种测量词项和文档相似度的方法在本章中也进行了介绍，我们通过使用 Python 语言和几个开源库成功地将数学表达式转换为代码，实现了其中的几种技术。我们详细介绍了文档聚类，研究了聚类模型的各种概念和类型。最后，我们采用了一个现实世界里的真实例子，使用 IMDb 电影简介数据对有史以来的前 100 部最佳电影进行了聚类，并使用不同的聚类模型，如 k-means、AP 聚类和沃德层次聚类来构建、分析并可视化聚类。这应该足以让你开始分析文档的相似度并进行文档聚类了，你甚至可以开始尝试组合前面各章节中介绍的各种技术。（提示：可以通过聚类、组合有监督学习和无监督学习构建分类器、使用文档类簇的增强推荐系统建立主题模型——仅举几例供你参考！）

第7章
语义与情感分析

在过去的十几年里，随着机器学习（ML）的出现和深度学习与人工智能的进一步发展，自然语言理解已经变得十分重要。计算机和其他可以编程学习事物和执行特殊操作的机器，其能力的主要局限就是没有像人类一样的感知、理解事物的能力。随着神经网络的再次兴起和计算机架构的进步，深度学习和人工智能迅速地演进，我们做了一些努力尝试，让机器自己学习、感知、理解和执行动作。你或许已经看到或听到这些努力，诸如自动驾驶汽车，计算机在如围棋和国际象棋等比赛中击败了经验丰富的人类选手，以及在网络上广泛使用的聊天机器人。

在第4～6章中，我们研究了各类用于文本分类、聚类和摘要的计算、语言处理和机器学习技术。第3章中开发了分析、理解文本语法和结构的方法与程序。本章将讨论一些尝试回答下面问题的方法，*我们能分析和理解文本背后的意义和情感吗？*

自然语言处理（NLP）有着广泛的应用，它尝试使用自然语言理解来推理文本背后的意义和内容，并解决一些问题。我们在第1章中已经简要讨论了这些应用。下面的一些应用需要从语义方面深入理解文本。

- 问答系统（question answering system）。
- 语境识别（contextual recognition）。
- 语音识别（speech recognition）。

*文本语义学*专门研究理解文本或语言的意义。单词在结合成语句之后，单词之间有了语法关系和上下文关系，形成各种关系和结构，语义处在分析和理解这些关系以及从这些关系中推理意义的关键位置。我们将探讨自然语言中不同类型的语义关系，研究一些自然语言处理技术来推理和提取文本中有意义的语义信息。语义仅与内容和意思相关，文本的结构或格式发挥很小的作用。但是有时，语法和词的排列有助于我们推理词的内容、区分不同的事物，例如“lead”一词何时意为金属铅，何时意为电影主演。

*情感分析*也许是文本分析中最主流的应用，有大量关注不同文本资源情感分析的手册、网站和应用，其内容涵盖从企业调查到电影评论。情感分析的关键是分析文本以理解其表达的观点，以及情绪和情态等其他因素。通常相比在客观内容上，情感分析能够在主观内容上更好地工作。这是因为当一个文本拥有客观的上下文或视角时，文本通常描述一些正常的陈述或事实而不表达任何情感、感觉或情绪。主观性文本包含的文本通常由一个具有典型情绪、情感和感受的人来表达。情感分析有着广泛的应用，尤其是作为社交媒体分析的一部分应用于如商业、近期电影或产品发布等任何领域，以理解人们对这些事物的接受情况，通过

他们的观点或者情感，或者你猜测到的情况，理解他们的看法。

本章将要讨论文本数据的语义分析和情感分析的几个方面。我们将从对一个字典数据库 WordNet 的探索开始，并介绍一个称为同义词集的新概念。我们也会探索不同的语义关系和自然语言的表示方法，包括语义消歧和命名实体识别。在情感分析方面，我们将研究如何使用有监督机器学习技术分析情感，同时也使用几个基于字典的无监督技术深入研究自然语言的情感、情绪和情态。

7.1 语义分析

我们已经知道单词如何组成短语，进一步形成从句，到最后形成句子。第 3 章介绍了自然语言不同的结构组成，包括词性标注（POS）、组块分析（chunking）和语法。这些概念都属于文本数据的句法和结构分析。然而，我们研究单词、短语、从句之间的语法关系，纯粹是基于它们的位置、句法和结构。语义分析更多的是关于文本中单词背后真实内容和意思的理解，和它们如何互相联系作为整体来传达一些信息。

如第 1 章提到的，语义学本身的定义就是意义的研究，语言语义学是语言学下面的一个完整分支，其主要研究自然语言的意义，包括研究不同单词、短语以及符号之间的关系。此外，也有各种表示与语句和命题相关联语义的方法。我们将在语义分析的基础上广泛讨论以下主题：

- 探索 WordNet 和同义词集。
- 分析词汇语义关系。
- 语义消歧。
- 命名实体识别。
- 分析语义的表示方法。

讨论这些主题的目的是让你对语义分析的资源情况，以及如何使用这些资源有一个清晰的认识。我们将通过实际例子，研究第 1 章提到的与语义分析相关的各种概念。你可以回顾 1.4 节相关内容。别再拖延了，让我们开始吧！

7.2 探索 WordNet

WordNet 是大型的英语词汇数据库。该数据库属于普林斯顿大学，你可以通过 https://wordnet.princeton.edu 网站了解关于 WordNet 的更多信息。在 G. A. Miller 教授的指导下，WordNet 于 1985 年在普林斯顿大学建立。该词汇数据库由名词、形容词、动词和副词组成，而且基于相同的概念将相关的单词分为一组，称之为认知同义词集或同义词集。每个同义词集表达了独特的、不同的概念。从更高的层面来看，WordNet 可以比作能提供单词和同义词的词汇表和词典。从较低的层面来看，WordNet 能够提供的不止如此，WordNet 同义词集合和与之相关的词汇具有基于语义、相似概念的详细的关系和结构。作为一个词汇数据库，WordNet 广泛使用在文本分析、自然语言处理以及基于人工智能的应用中。

WordNet 数据库由 155 000 个单词组成，含有超过 117 000 个同义词集和 206 000 个词 - 义对。数据库大概有 12MB，可以支持各种接口和 API 的访问。官方网站提供一个 Web 应用

程序接口，可以访问与所输入单词相关的各种详细的单词、同义词和概念。你可以在 http://wordnetweb.princeton.edu/perl/webwn 地址访问它，也可以在 https://wordnet.princeton.edu/wordnet/download/地址下载它。下载的内容包括 WordNet 的各种程序包、文件和工具。我们将使用 nltk 程序包访问 WordNet。现在开始探索同义词以及使用同义词的不同语义关系吧。

7.2.1 理解同义词集

同义词是将每个事物联系在一起的非常重要的概念和结构之一，所以我们从研究同义词集开始探索 WordNet。通常，基于自然语言处理和信息检索的概念，一个同义词集是一组语义相似的数据实体的集合。这并不意味着它们完全相同，而是它们聚焦于相似的内容或概念。从 WordNet 的具体内容来讲，一个同义词集是一组同义词，它们在一个具体的概念上是可以替换或替代的。同义词集不仅由简单的单词组成，也可由词组组成。多义词（读音和组成相同但拥有不同相关意思的单词）会基于它们的意思被分配到不同的同义词集。同义词集通过语义关系与其他同义词集进行连接，下面的章节将对其进行介绍。一般情况下，每个同义词集拥有解释其意思的词汇、一些可选的例子以及相关的主旨（同义词集合）。一些词汇拥有多个与之相关的同义词集，每个同义词集拥有具体的内容。

让我们使用 nltk 的 WordNet 接口研究一个真实的例子，使用'fruit'这个单词探索同义词集，具体可以使用下面的代码来实现：

```
from nltk.corpus import wordnet as wn
import pandas as pd

term = 'fruit'
synsets = wn.synsets(term)
# display total synsets
In [75]: print 'Total Synsets:', len(synsets)
Total Synsets: 5
```

我们可以看到'fruit'这个单词一共有 5 个与之相联系的同义词集，这些同义词集表明了什么内容？可以使用下面的代码，深入挖掘每个同义词集及其组成。

```
In [76]: for synset in synsets:
    ...:     print 'Synset:', synset
    ...:     print 'Part of speech:', synset.lexname()
    ...:     print 'Definition:', synset.definition()
    ...:     print 'Lemmas:', synset.lemma_names()
    ...:     print 'Examples:', synset.examples()
    ...:     print
    ...:
    ...:
Synset: Synset('fruit.n.01')
Part of speech: noun.plant
Definition: the ripened reproductive body of a seed plant
Lemmas: [u'fruit']
Examples: []

Synset: Synset('yield.n.03')
Part of speech: noun.artifact
Definition: an amount of a product
Lemmas: [u'yield', u'fruit']
Examples: []

Synset: Synset('fruit.n.03')
Part of speech: noun.event
Definition: the consequence of some effort or action
Lemmas: [u'fruit']
```

```
Examples: [u'he lived long enough to see the fruit of his policies']

Synset: Synset('fruit.v.01')
Part of speech: verb.creation
Definition: cause to bear fruit
Lemmas: [u'fruit']
Examples: []
Synset: Synset('fruit.v.02')
Part of speech: verb.creation
Definition: bear fruit
Lemmas: [u'fruit']
Examples: [u'the trees fruited early this year']
```

上面的代码输出显示了与'fruit'相关的每个同义词集的细节，这些定义给出了每个同义词集的意义和与它相关的主旨。同时提到了每个同义词的语音部分，包括名词和动词。上面的输出也给出了一些例子，显示这些词汇在实际句子中如何使用。既然已经更好地认识了同义词集，现在开始探索前面提到的各种语义关系吧。

7.2.2 分析词汇的语义关系

文本语义学指的是意思和内容的研究。同义词集给出了各种词汇间一个很好的抽象，提供了有用的信息，例如定义、例子、词性和主旨。但是，我们可以使用同义词集探索实体间的语义关系吗？答案是肯定的。我们将讨论与语义相关的诸多概念（内容涵盖 1.4.1 节）。当我们在这里使用实例分析概念时，你最好再回顾一下那一节的内容，以便更好地理解它们。我们将使用 nltk 的 wordnet 资源，但是你可以使用 pattern 程序包中的 WordNet，其包含与 nltk 相似的接口。

1. 蕴含

通常蕴含指的是同样事件或行为，这些事件或行为在逻辑上涉及或者与其他已经发生或将要发生的行为或事件相关联。理想情况下，这适用于表示某些特定行为的动词。以下代码说明了如何获得蕴涵：

```
# entailments
In [80]: for action in ['walk', 'eat', 'digest']:
    ...:     action_syn = wn.synsets(action, pos='v')[0]
    ...:     print action_syn, '-- entails -->', action_syn.entailments()
Synset('walk.v.01') -- entails --> [Synset('step.v.01')]
Synset('eat.v.01') -- entails --> [Synset('chew.v.01'),
Synset('swallow.v.01')]
Synset('digest.v.01') -- entails --> [Synset('consume.v.02')]
```

从输出中，你可以看到相关的同义词描述蕴涵的概念。蕴涵描述了相关的行为，就像走路这样的行为涉及或需要迈步，吃需要咀嚼和吞咽一样。

2. 同音词和同形异义词

从较高的水平来看，同音词指的是单词或词项具有相同的书面形式或发音但意义不同。同音词是同形异义词的超集，同形异义词指的是具有同样拼写的单词，但可能具有不同的读音和意义。下面的代码片段为显示了如何得到同音词/同形异义词：

```
In [81]: for synset in wn.synsets('bank'):
    ...:     print synset.name(),'-',synset.definition()
    ...:
    ...:
bank.n.01 - sloping land (especially the slope beside a body of water)
depository_financial_institution.n.01 - a financial institution that accepts
```

```
deposits and channels the money into lending activities
bank.n.03 - a long ridge or pile
bank.n.04 - an arrangement of similar objects in a row or in tiers
...
...
deposit.v.02 - put into a bank account
bank.v.07 - cover with ashes so to control the rate of burning
trust.v.01 - have confidence or faith in
```

上面的输出显示了单词'bank'不同的同形异义词的一部分结果。你可以看到单词'bank'有很多不同意思，这很好地解释了同形异义词。

3. 同义词和反义词

如你所知，同义词有相似的意思或内容，反义词则是指具有相反或不同意思的单词。下面的代码显示了同义词和反义词：

```
In [82]: term = 'large'
    ...: synsets = wn.synsets(term)
    ...: adj_large = synsets[1]
    ...: adj_large = adj_large.lemmas()[0]
    ...: adj_large_synonym = adj_large.synset()
    ...: adj_large_antonym = adj_large.antonyms()[0].synset()
    ...: # print synonym and antonym
    ...: print 'Synonym:', adj_large_synonym.name()
    ...: print 'Definition:', adj_large_synonym.definition()
    ...: print 'Antonym:', adj_large_antonym.name()
    ...: print 'Definition:', adj_large_antonym.definition()
Synonym: large.a.01
Definition: above average in size or number or quantity or magnitude or
extent
Antonym: small.a.01
In [83]: term = 'rich'
    ...: synsets = wn.synsets(term)[:3]
    ...: # print synonym and antonym for different synsets
    ...: for synset in synsets:
    ...:     rich = synset.lemmas()[0]
    ...:     rich_synonym = rich.synset()
    ...:     rich_antonym = rich.antonyms()[0].synset()
    ...:     print 'Synonym:', rich_synonym.name()
    ...:     print 'Definition:', rich_synonym.definition()
    ...:     print 'Antonym:', rich_antonym.name()
    ...:     print 'Definition:', rich_antonym.definition()
Synonym: rich_people.n.01
Definition: people who have possessions and wealth (considered as a group)
Antonym: poor_people.n.01
Definition: people without possessions or wealth (considered as a group)

Synonym: rich.a.01
Definition: possessing material wealth
Antonym: poor.a.02
Definition: having little money or few possessions

Synonym: rich.a.02
Definition: having an abundant supply of desirable qualities or substances
(especially natural resources)
Antonym: poor.a.04
Definition: lacking in specific resources, qualities or substances
```

上面的输出显示了单词'large'和'rich'的同义词和反义词。此外，我们还探索了与单词或概念'rich'相关的几个同义词集，它们正确地给出了确切的同义词和对应的反义词。

4. 下位词和上位词

同义词集表示具有独立语义和概念的词、基于相似性和内容而联系或关联在一起。除了

具体的实体外，一些同义词集代表抽象的和一般性的概念。通常情况下，它们以一种层次结构连接的一起，表示一种 is-a 关系。下位词和上位词帮助我们在层次结构中探索相关的概念。具体来讲，下位词所指的概念和实体是高层概念或实体的子类，与其超类相比，下位词具有更具体的意义和语境。下面的片段给出了下位词的实体'tree'：

```
term = 'tree'
synsets = wn.synsets(term)
tree = synsets[0]
# print the entity and its meaning
In [86]: print 'Name:', tree.name()
    ...: print 'Definition:', tree.definition()
Name: tree.n.01
Definition: a tall perennial woody plant having a main trunk and branches
forming a distinct elevated crown; includes both gymnosperms and angiosperms
# print total hyponyms and some sample hyponyms for 'tree'
In [87]: hyponyms = tree.hyponyms()
    ...: print 'Total Hyponyms:', len(hyponyms)
    ...: print 'Sample Hyponyms'
    ...: for hyponym in hyponyms[:10]:
    ...:     print hyponym.name(), '-', hyponym.definition()

Total Hyponyms: 180
Sample Hyponyms
aalii.n.01 - a small Hawaiian tree with hard dark wood
acacia.n.01 - any of various spiny trees or shrubs of the genus Acacia
african_walnut.n.01 - tropical African timber tree with wood that resembles
mahogany
albizzia.n.01 - any of numerous trees of the genus Albizia
alder.n.02 - north temperate shrubs or trees having toothed leaves and
conelike fruit; bark is used in tanning and dyeing and the wood is rot-
resistant
angelim.n.01 - any of several tropical American trees of the genus Andira
angiospermous_tree.n.01 - any tree having seeds and ovules contained in the
ovary
anise_tree.n.01 - any of several evergreen shrubs and small trees of the
genus Illicium
arbor.n.01 - tree (as opposed to shrub)
aroeira_blanca.n.01 - small resinous tree or shrub of Brazil
```

上面的输出告诉我们这个'tree'有 180 个下位词，我们可以看到一些下位词例子和它们的定义。如预期的那样，可以看到每个下位词是一类具体的树。作为超类的下位词具有更一般的意义和语境。下面的代码显示了这个'tree'的直接超类。

```
In [88]: hypernyms = tree.hypernyms()
    ...: print hypernyms
[Synset('woody_plant.n.01')]
```

你可以使用下面的代码在描述了'tree'的下位词或父类的整个实体/概念结构上进行向上导航：

```
# get total hierarchy pathways for 'tree'
In [91]: hypernym_paths = tree.hypernym_paths()
    ...: print 'Total Hypernym paths:', len(hypernym_paths)
Total Hypernym paths: 1

# print the entire hypernym hierarchy
In [92]: print 'Hypernym Hierarchy'
    ...: print ' -> '.join(synset.name() for synset in hypernym_paths[0])
Hypernym Hierarchy
entity.n.01 -> physical_entity.n.01 -> object.n.01 -> whole.n.02 -> living_
thing.n.01 -> organism.n.01 -> plant.n.02 -> vascular_plant.n.01 -> woody_
plant.n.01 -> tree.n.01
```

从上面的输出可以看到'entity'是'tree'上最一般性的概念，完整的上位词层次结构显示了每个层次对应的上位词或超类。当你继续向下导航，你将获得更加具体的概念/实体，如果反方向导航，你将获得更加一般的概念/实体。

5. 整体词和部分词

整体词是一些实体，包含我们感兴趣的特定实体。基本上，整体词指的是单词或实体间的关系，表示一个整体或整体的具体部分。下面的代码片段显示了'tree'的整体词。

```
In [94]: member_holonyms = tree.member_holonyms()
    ...: print 'Total Member Holonyms:', len(member_holonyms)
    ...: print 'Member Holonyms for [tree]:-'
    ...: for holonym in member_holonyms:
    ...:     print holonym.name(), '-', holonym.definition()
Total Member Holonyms: 1
Member Holonyms for [tree]:-
forest.n.01 - the trees and other plants in a large densely wooded area
```

从输出可以看到，'forest'是'tree'的整体词，其语义上是正确的，因为森林是由树组成。部分词表示一个单词或实体作为其他单词组成或一部分的语义关系。下面的代码给出了'tree'的不同类型的部分词。

```
# part based meronyms for tree
In [95]: part_meronyms = tree.part_meronyms()
    ...: print 'Total Part Meronyms:', len(part_meronyms)
    ...: print 'Part Meronyms for [tree]:-'
    ...: for meronym in part_meronyms:
    ...:     print meronym.name(), '-', meronym.definition()
Total Part Meronyms: 5
Part Meronyms for [tree]:-
burl.n.02 - a large rounded outgrowth on the trunk or branch of a tree
crown.n.07 - the upper branches and leaves of a tree or other plant
limb.n.02 - any of the main branches arising from the trunk or a bough of a
tree
stump.n.01 - the base part of a tree that remains standing after the tree
has been felled
trunk.n.01 - the main stem of a tree; usually covered with bark; the bole is
usually the part that is commercially useful for lumber

# substance based meronyms for tree
In [96]: substance_meronyms = tree.substance_meronyms()
    ...: print 'Total Substance Meronyms:', len(substance_meronyms)
    ...: print 'Substance Meronyms for [tree]:-'
    ...: for meronym in substance_meronyms:
    ...:     print meronym.name(), '-', meronym.definition()
Total Substance Meronyms: 2
Substance Meronyms for [tree]:-
heartwood.n.01 - the older inactive central wood of a tree or woody plant;
usually darker and denser than the surrounding sapwood
sapwood.n.01 - newly formed outer wood lying between the cambium and the
heartwood of a tree or woody plant; usually light colored; active in water
conduction
```

上面的输出显示了不同的部分词，包括树的不同组成，如树桩和树干，以及衍生的其他物体，如树的心材和边材。

6. 语义关系与相似度

在前面的章节中，我们已经研究了各种与词汇语义关系相关的概念。现在将研究基于语义关系连接相似实体的方法，以及它们的相似性度量。语义相似性不同于第6章所讨论的传统的相似性度量。我们将使用一些与生活实体相关的同义词集例子，如下面分析代码所示。

```
tree = wn.synset('tree.n.01')
lion = wn.synset('lion.n.01')
tiger = wn.synset('tiger.n.02')
cat = wn.synset('cat.n.01')
dog = wn.synset('dog.n.01')
# create entities and extract names and definitions
entities = [tree, lion, tiger, cat, dog]
entity_names = [entity.name().split('.')[0] for entity in entities]
entity_definitions = [entity.definition() for entity in entities]

# print entities and their definitions
In [99]: for entity, definition in zip(entity_names, entity_definitions):
    ...:     print entity, '-', definition
tree - a tall perennial woody plant having a main trunk and branches forming
a distinct elevated crown; includes both gymnosperms and angiosperms
lion - large gregarious predatory feline of Africa and India having a tawny
coat with a shaggy mane in the male
tiger - large feline of forests in most of Asia having a tawny coat with
black stripes; endangered
cat - feline mammal usually having thick soft fur and no ability to roar:
domestic cats; wildcats
dog - a member of the genus Canis (probably descended from the common wolf)
that has been domesticated by man since prehistoric times; occurs in many
breeds
```

我们知道我们的实体相比于这些定义可以更好地解释内容，我们将基于这些实体共同的上位词来联系它们。对于每一对实体，我们将尽力寻找关系层次结构树中最低级别的上位词。相关联的实体应该拥有非常具体的上位词，不相关的实体则拥有非常抽象或通用的上位词。具体说明如下面的代码片段所示：

```
common_hypernyms = []
for entity in entities:
    # get pairwise lowest common hypernyms
    common_hypernyms.append([entity.lowest_common_hypernyms(compared_entity)[0]
                                        .name().split('.')[0]
                             for compared_entity in entities])
# build pairwise lower common hypernym matrix
common_hypernym_frame = pd.DataFrame(common_hypernyms,
                                     index=entity_names,
                                     columns=entity_names)
# print the matrix
In [101]: print common_hypernym_frame
     ...:
           tree      lion      tiger        cat        dog
tree       tree   organism   organism   organism   organism
lion   organism       lion    big_cat     feline  carnivore
tiger  organism    big_cat      tiger     feline  carnivore
cat    organism     feline     feline        cat  carnivore
dog    organism  carnivore  carnivore  carnivore        dog
```

对于每对实体，忽略矩阵的主对角线，我们看到它们最低级的共同上位词描述了它们之间的自然关系。除了它们都是生物之外，树（'tree'）与其他动物之间没有关系。因此我们得到了它们之间的生物（'organism'）关系。猫和狮子、老虎有关，都是猫科动物，从上面的输出同样看到了这一点。老虎和狮子彼此通过大猫（'big cat'）相互联系。最后，因为它们通常都吃肉，我们可以看到狗与其他动物是食肉动物（'carnivore'）的关系。

也可以使用各种语义概念来度量实体之间的语义相似性。我们将使用路径相似性('path similarity')，它基于连接两个词的上位词/下位词的最短路径返回一个数值，其范围是[0，1]。下面的代码显示了如何生成这个相似性矩阵：

```
similarities = []
for entity in entities:
    # get pairwise similarities
    similarities.append([round(entity.path_similarity(compared_entity), 2)
                         for compared_entity in entities])
# build pairwise similarity matrix
similarity_frame = pd.DataFrame(similarities,
                                index=entity_names,
                                columns=entity_names)
# print the matrix
print similarity_frame
```

```
       tree  lion  tiger   cat   dog
tree   1.00  0.07   0.07  0.08  0.13
lion   0.07  1.00   0.33  0.25  0.17
tiger  0.07  0.33   1.00  0.25  0.17
cat    0.08  0.25   0.25  1.00  0.20
dog    0.13  0.17   0.17  0.20  1.00
```

如预期的一样，从上面的输出可以看到狮子和老虎最相似，其值为0.33，紧接着是猫，猫与它们的语义相似性值为0.25。与动物相比，树的语义相似性最低。

到这里就结束了对词汇语义关系的分析的讨论。我鼓励你尝试使用不同的例子，利用WordNet探索更多的概念。

7.3 词义消歧

在前面的章节，我们研究了同形异义词和同音异义词，这些词的拼写或发音相似，但意义不同。词义基于单词如何使用且依赖于单词的语义，由语境决定。假设单词用于多个意思，基于单词使用情况，识别单词正确语义和意义称为语义消歧。语义消歧是自然语言处理中的典型问题，在各领域均有使用，如提高搜索引擎结果的相关性、一致性等。

这里可用多种方法解决这个问题，包括基于词汇和字典的方法、有监督机器学习和无监督机器学习的方法。详细论述每个方面将超出本书的范围，在这里我将使用Lesk算法介绍语义消歧，该算法是一个经典的算法，由M. E. Lesk于1986年发明。该算法的基本原理是使用字典或词汇的定义为我们消除文本的歧义，把我们感兴趣单词周围的一段文字与这些定义中的文字进行比较。我们将使用WordNet的定义代替单词词典。我们的主要目的是返回上下文的句子和我们要消歧的单词同义词集的定义之间的最大数量的重叠词或词项。下面的代码片段描述了如何使用`nltk`进行语义消歧：

```
from nltk.wsd import lesk
from nltk import word_tokenize

# sample text and word to disambiguate
samples = [('The fruits on that plant have ripened', 'n'),
           ('He finally reaped the fruit of his hard work as he won the
           race', 'n')]
word = 'fruit'
# perform word sense disambiguation
In [106]: for sentence, pos_tag in samples:
     ...:     word_syn = lesk(word_tokenize(sentence.lower()), word, pos_tag)
     ...:     print 'Sentence:', sentence
     ...:     print 'Word synset:', word_syn
     ...:     print 'Corresponding definition:', word_syn.definition()
     ...:     print
Sentence: The fruits on that plant have ripened
```

```
Word synset: Synset('fruit.n.01')
Corresponding definition: the ripened reproductive body of a seed plant

Sentence: He finally reaped the fruit of his hard work as he won the race
Word synset: Synset('fruit.n.03')
Corresponding definition: the consequence of some effort or action

# sample text and word to disambiguate
samples = [('Lead is a very soft, malleable metal', 'n'),
           ('John is the actor who plays the lead in that movie', 'n'),
           ('This road leads to nowhere', 'v')]
word = 'lead'
# perform word sense disambiguation
In [108]: for sentence, pos_tag in samples:
     ...:     word_syn = lesk(word_tokenize(sentence.lower()), word,
              pos_tag)
     ...:     print 'Sentence:', sentence
     ...:     print 'Word synset:', word_syn
     ...:     print 'Corresponding definition:', word_syn.definition()
     ...:     print
Sentence: Lead is a very soft, malleable metal
Word synset: Synset('lead.n.02')
Corresponding definition: a soft heavy toxic malleable metallic element;
bluish white when freshly cut but tarnishes readily to dull grey

Sentence: John is the actor who plays the lead in that movie
Word synset: Synset('star.n.04')
Corresponding definition: an actor who plays a principal role

Sentence: This road leads to nowhere
Word synset: Synset('run.v.23')
Corresponding definition: cause something to pass or lead somewhere
```

我们在上面的例子中尝试对不同文本文档中的两个单词'fruit'和'lead'进行语义消歧。你可以看到我们利用 Lesk 算法基于每个文档的使用和内容获得了单词的正确语义。这告诉你，fruit 有两个意思，是可以消费的实体或是一些行为的结果。我们也看到 lead 的意思是一种软金属、让某物/某人去某地，或是戏剧或电影中的主演。

7.4 命名实体识别

在文档中，有一些特别的词汇与文本中的其他词汇比起来代表更多的信息，并拥有上下文一致的内容。这些实体称为*命名实体*，这些实体特指代表真实世界的对象的词汇，如人物、地点、组织等，一般通过专有名词表示。通常我们可以通过查询文本中的名词短语来找到这些实体。*命名实体识别*也称为*实体识别/提取*，经常用于信息提取来识别和分割命名实体，或在不同预定义类型下进行分类。一些最经常使用的类型如图 7-1 所示（图片源自 nltk 和斯坦福 NLP 研究组）。

GPE 和 LOCATION 的实体类型之间存在一些重叠。GPE 实体通常更通用，代表地缘政治方面的实体，如城市、州、国家和洲。LOCATION 也可以指这些实体（它因不同的 NER 系统而变化），以及非常具体的位置，如山、河或山间避暑小镇。另一方面，FACILITY 通常是指人造的流行纪念碑或文物。图 7-1 中，剩余的类别通过名字和例子可以很好地对其进行解释说明。

德甲联赛是德国最流行的高水平职业足球联赛，拜仁慕尼黑足球俱乐部是德甲联赛中具有全球声誉的俱乐部之一。现在我们将以这个俱乐部在维基百科的样本描述为例，尝试提

命名实体类型	例 子
PERSON	President Obama, Franz Beckenbauer
ORGANIZATION	WHO, ISRO, FC Bayern
LOCATION	Germany, India, USA, Mt. Everest
DATE	December, 2016-12-25
TIME	12:30:00 AM, one thirty pm
MONEY	Twenty dollars, Rs. 50, 100 GBP
PERCENT	20%, forty five percent
FACILITY	Stonehenge, Taj Mahal, Washington Monument
GPE	Asia, Europe, Germany, North America

图 7-1 常用的命名实体及例子

取命名实体。这一节，我们将重用规范化模块（访问源代码文件 normalization.py），在最后解析文件，删除不必要的新行。下面将使用 nltk 进行命名实体抽取：

```
# sample document
text = """
Bayern Munich, or FC Bayern, is a German sports club based in Munich,
Bavaria, Germany. It is best known for its professional football team,
which plays in the Bundesliga, the top tier of the German football
league system, and is the most successful club in German football
history, having won a record 26 national titles and 18 national cups.
FC Bayern was founded in 1900 by eleven football players led by Franz John.
Although Bayern won its first national championship in 1932, the club
was not selected for the Bundesliga at its inception in 1963. The club
had its period of greatest success in the middle of the 1970s when,
under the captaincy of Franz Beckenbauer, it won the European Cup three
times in a row (1974-76). Overall, Bayern has reached ten UEFA Champions
League finals, most recently winning their fifth title in 2013 as part
of a continental treble.
"""

import nltk
from normalization import parse_document
import pandas as pd

# tokenize sentences
sentences = parse_document(text)
tokenized_sentences = [nltk.word_tokenize(sentence) for sentence in
sentences]

# tag sentences and use nltk's Named Entity Chunker
tagged_sentences = [nltk.pos_tag(sentence) for sentence in tokenized_
sentences]
ne_chunked_sents = [nltk.ne_chunk(tagged) for tagged in tagged_sentences]

# extract all named entities
named_entities = []
for ne_tagged_sentence in ne_chunked_sents:
    for tagged_tree in ne_tagged_sentence:
        # extract only chunks having NE labels
        if hasattr(tagged_tree, 'label'):
                entity_name = ' '.join(c[0] for c in tagged_tree.leaves()) #
                get NE name
                entity_type = tagged_tree.label() # get NE category
                named_entities.append((entity_name, entity_type))
# get unique named entities
named_entities = list(set(named_entities))
# store named entities in a data frame
entity_frame = pd.DataFrame(named_entities,
                            columns=['Entity Name', 'Entity Type'])
# display results
In [116]: print entity_frame
```

```
          Entity Name   Entity Type
0              Bayern        PERSON
1          Franz John        PERSON
2   Franz Beckenbauer        PERSON
3              Munich  ORGANIZATION
4            European  ORGANIZATION
5          Bundesliga  ORGANIZATION
6              German           GPE
7             Bavaria           GPE
8             Germany           GPE
9           FC Bayern  ORGANIZATION
10               UEFA  ORGANIZATION
11             Munich           GPE
12             Bayern           GPE
13            Overall           GPE
```

命名实体识别器从上面的文档中识别出命名实体，然后我们再从完成标签标注的句子中提取这些命名实体，如上所示使用数据框显示它们。尽管出现了几个错误的识别，你还是可以清楚地看到上述代码是如何识别出人物（PERSON）、组织（ORGANIZATION）和地理位置（GPE）相关的命名实体的。

我们将在同样的文本中使用斯坦福 NER 标记器，并与上面结果进行比较。为实现这一过程，你需要安装 Java 并在 http://nlp. stanford. edu/software/stanford-ner-2014-08-27. zip 地址下载斯坦福 NER 的资源。将这些资源解压缩到指定的位置（在我的系统中，使用 E:/stanford）。完成之后，你可以使用 nltk 接口访问它们，类似于第 3 章所做的关于选区和从属关系解析的工作。关于斯坦福 NER 的详细信息，请访问官方网站 http://nlp. stanford. edu/software/CRF-NER. shtml，该网站包含了命名实体识别器的最新版本（我使用的是老版本）：

```
from nltk.tag import StanfordNERTagger
import os

# set java path in environment variables
java_path = r'C:\Program Files\Java\jdk1.8.0_102\bin\java.exe'
os.environ['JAVAHOME'] = java_path

# load stanford NER
sn = StanfordNERTagger('E:/stanford/stanford-ner-2014-08-27/classifiers/
english.all.3class.distsim.crf.ser.gz',
                       path_to_jar='E:/stanford/stanford-ner-2014-08-27/
                       stanford-ner.jar')
# tag sentences
ne_annotated_sentences = [sn.tag(sent) for sent in tokenized_sentences]

# extract named entities
named_entities = []
for sentence in ne_annotated_sentences:
    temp_entity_name = ''
    temp_named_entity = None
    for term, tag in sentence:
        # get terms with NE tags
        if tag != 'O':
            temp_entity_name = ' '.join([temp_entity_name, term]).strip() #
            get NE name
            temp_named_entity = (temp_entity_name, tag) # get NE and its
            category
        else:
            if temp_named_entity:
                named_entities.append(temp_named_entity)
                temp_entity_name = ''
                temp_named_entity = None

# get unique named entities
named_entities = list(set(named_entities))
```

```
# store named entities in a data frame
entity_frame = pd.DataFrame(named_entities,
                            columns=['Entity Name', 'Entity Type'])

# display results
In [118]: print entity_frame
         Entity Name   Entity Type
0         Franz John        PERSON
1  Franz Beckenbauer        PERSON
2            Germany      LOCATION
3             Bayern  ORGANIZATION
4            Bavaria      LOCATION
5             Munich      LOCATION
6          FC Bayern  ORGANIZATION
7               UEFA  ORGANIZATION
8      Bayern Munich  ORGANIZATION
```

上面的输出给出了我们从文档中获得的各种命名实体。你可以将这个结果与 nltk NER 识别器的结果进行比较。这个结果明显好一些，这里没有错误的分类，所有命名实体都被赋予了正确的类型。有一些非常有趣的点：它已经正确地将 Munich 识别为 LOCATION，将 Mayern Munich 识别为 ORGANIZATION。这是否意味着第二个命名实体识别器更好呢？不完全是。这依赖于你所分析的语料库类型，你可以使用与第 3 章相似的方法通过有监督的机器学习来训练预标记的语料库，从而建立自己的 NER 识别器。实际上，上述讨论的两个标记器已经使用预标记的语料库如 CoNLL、MUC 和 Penn Treebank 进行了训练。

7.5 分析语义表征

我们通过口语或书面语的形式与其他人或接口进行交流。这些信息一般是单词、短语和句子的集合，它们各自拥有自己的语义和内容。目前为止，我们一直在讨论不同单词单元间的语义和关系。但是，我们如何表示一个或多个消息所传递的语义呢？人们如何理解其他人正在和他们讲述的内容呢？我们如何相信陈述与命题，评估其结果，并最终确定采取什么行动？因为大脑帮助我们进行了逻辑和推理，感觉这个比较容易，但是在计算上我们能一样做到吗？

答案是可以。如命题逻辑和一阶逻辑的框架可以帮助我们进行语义表示。我们在 1.4.3 节已经讨论了这一点。我支持你再次阅读该节以加强记忆。在下面的章节中，我们将会研究命题逻辑和一阶逻辑的表示方法，用实际的例子和代码证明或反证命题、语句和谓词。

7.5.1 命题逻辑

当研究命题、陈述和语句时，我们已经讨论了命题逻辑（PL）。一个命题通常是一个声明，其值是二进制值真或假。这里还有各种不同的逻辑运算符，如与、或、蕴含和等价。我们还研究了这些运算符在多个命题上的应用效果，以了解它们的行为和结果。

让我们思考第 1 章关于 2 个命题 P 和 Q 的例子，它们可以如下表示：

P: He is hungry
Q: He will eat a sandwich

我们现在将尝试基于 1.4.3 节（请参考“命题逻辑”这一部分）所讨论的各种逻辑运算符，使用 nltk 建立这些命题上不同运算符的真值表并导出计算结果：

```
import nltk
import pandas as pd
import os
```

```
# assign symbols and propositions
symbol_P = 'P'
symbol_Q = 'Q'
proposition_P = 'He is hungry'
propositon_Q = 'He will eat a sandwich'
# assign various truth values to the propositions
p_statuses = [False, False, True, True]
q_statuses = [False, True, False, True]
# assign the various expressions combining the logical operators
conjunction = '(P & Q)'
disjunction = '(P | Q)'
implication = '(P -> Q)'
equivalence = '(P <-> Q)'
expressions = [conjunction, disjunction, implication, equivalence]

# evaluate each expression using propositional logic
results = []
for status_p, status_q in zip(p_statuses, q_statuses):
    dom = set([])
    val = nltk.Valuation([(symbol_P, status_p),
                          (symbol_Q, status_q)])
    assignments = nltk.Assignment(dom)
    model = nltk.Model(dom, val)
    row = [status_p, status_q]
    for expression in expressions:
        # evaluate each expression based on proposition truth values
        result = model.evaluate(expression, assignments)
        row.append(result)
    results.append(row)
# build the result table
columns = [symbol_P, symbol_Q, conjunction,
           disjunction, implication, equivalence]
result_frame = pd.DataFrame(results, columns=columns)

# display results
In [125]: print 'P:', proposition_P
     ...: print 'Q:', propositon_Q
     ...: print
     ...: print 'Expression Outcomes:-'
     ...: print result_frame
P: He is hungry
Q: He will eat a sandwich

Expression Outcomes:-
       P      Q (P & Q) (P | Q) (P -> Q) (P <-> Q)
0  False  False   False   False     True      True
1  False   True   False    True     True     False
2   True  False   False    True    False     False
3   True   True    True    True     True      True
```

上面的输出描述了这两个命题的不同真值，当我们使用不同的逻辑运算符将它们结合在一起时，你会发现这些结果与第 1 章中人工评估的结果是一致的。例如，P&Q 表示当且仅当两个独立的命题为真时，“He is hungry and he will eat a sandwich（他饿了并且他想吃三明治）”为 True。我们使用 nltk 的 Valuation 类创建命题字典和它们不同的输出状态。我们使用 Model 类评估每个表达式，其中 evaluate() 函数内部调用递归函数 satisfy()，这有助于在所赋真值的基础上评估每个带有命题的表达式的结果。

7.5.2 一阶逻辑

命题逻辑有几个局限，如没有能力表示事实或复杂的关系和推理。对于每个新的命题，我们需要统一的符号表示，这使得产生事实非常困难，因此命题逻辑表现能力十分有限。一

阶逻辑（FOL）具有如函数、量词、关系、连接词和符号等特征，这是其能够很好工作的原因。它提供了更加丰富和有力的语义信息表示方法。1.4.3 节的“一阶逻辑”部分提供了 FOL 如何工作的详细信息。

本节将建立几个与第 1 章手工建立的数学陈述类似的 FOL 陈述。这里我们将使用相似的语法构建它们，并类似于 PL 所做的事情，我们基于预先定义的条件和关系，使用 nltk 和一些定理证明器来证明不同表达式的输出。从这一节中获得的主要内容应该是了解如何在 Python 中表示 FOL 表达式，以及如何使用基于某些目标和预定义规则和事件的证据来执行后续推理。这里有几个定理证明器（prover）可以用于评价表达式和证明定理。nltk 程序包有三个主要的不同类型的定理证明器：Prover9、TableauProver 和 ResolutionProver。Prover9 是一个可以免费使用的定理证明器，可在 www. cs. unm. edu/ ~ mccune/prover9/download/上下载。你可以选择一个位置来解压缩这些内容（我使用 E:/prover9）。在下面的例子中，我们将使用 ResolutionProver 和 Prover9。下面的代码片段帮助我们建立 FOL 表达式和评估所依赖的环境：

```
import nltk
import os
# for reading FOL expressions
read_expr = nltk.sem.Expression.fromstring
# initialize theorem provers (you can choose any)
os.environ['PROVER9'] = r'E:/prover9/bin'
prover = nltk.Prover9()
# I use the following one for our examples
prover = nltk.ResolutionProver()
```

既然已经准备好了依赖的软件，让我们对几个 FOL 表达式进行评估吧。考虑一个简单的表述，If an entity jumps over another entity, the reverse cannot happen（如果一个实体跳过另外一个实体，那么相反的情况不能发生）。假设实体是 x 和 y，我们可以使用 FOL ∀x ∀y (jumps_over(x, y) →¬ jumps_over(y, x)) 表示这个情况，这意味着对于所有的 x 和 y，如果 x 跳过 y，y 就不能跳过 x。考虑我们有两个实体 fox 和 dog，fox 跳过了 dog 是一个已经发生的事件，可以用 jumps_over(fox, dog) 表示。我们的最终目标是在考虑上面的表达式和事件已经发生的情况下，评估 jumps_over(dog, fox) 的输出。下面的片段显示了如何实现它们：

```
# set the rule expression
rule = read_expr('all x. all y. (jumps_over(x, y) -> -jumps_over(y, x))')
# set the event occured
event = read_expr('jumps_over(fox, dog)')
# set the outcome we want to evaluate -- the goal
test_outcome = read_expr('jumps_over(dog, fox)')

# get the result
In [132]: prover.prove(goal=test_outcome,
     ...:              assumptions=[event, rule],
     ...:              verbose=True)
[1] {-jumps_over(dog,fox)}                    A
[2] {jumps_over(fox,dog)}                     A
[3] {-jumps_over(z4,z3), -jumps_over(z3,z4)}  A
[4] {-jumps_over(dog,fox)}                    (2, 3)

Out[132]: False
```

上面的输出描述了我们的目标 test_outcome 的最后结果是 Flase。基于我们的规则和在证明器中已经赋值给假设参数的事件，如果 fox 已经跳过了 dog，dog 就不能跳过 fox。

输出显示了推导结果的顺序步骤。让我们考虑另外一个 FOL 表达式规则 ∀ x studies(x, exam)→pass(x, exam)，这个表达式告诉我们对于所有实例 x，如果 x 为考试学习了，他/她就可以通过考试。让我们表示这个规则，并考虑两个学生，John 和 Pierre，John 没有学习，Pierre 学习了。我们可以找到输出，基于给定的表达式规则，他们会通过考试吗？下面的片段显示了如何实现：

```
# set the rule expression
rule = read_expr('all x. (studies(x, exam) -> pass(x, exam))')
# set the events and outcomes we want to determine
event1 = read_expr('-studies(John, exam)')
test_outcome1 = read_expr('pass(John, exam)')
event2 = read_expr('studies(Pierre, exam)')
test_outcome2 = read_expr('pass(Pierre, exam)')

# get results
In [134]: prover.prove(goal=test_outcome1,
     ...:              assumptions=[event1, rule],
     ...:              verbose=True)
[1] {-pass(John,exam)}                  A
[2] {-studies(John,exam)}               A
[3] {-studies(z6,exam), pass(z6,exam)}  A
[4] {-studies(John,exam)}               (1, 3)

Out[134]: False

In [135]: prover.prove(goal=test_outcome2,
     ...:              assumptions=[event2, rule],
     ...:              verbose=True)
[1] {-pass(Pierre,exam)}                A
[2] {studies(Pierre,exam)}              A
[3] {-studies(z8,exam), pass(z8,exam)}  A
[4] {-studies(Pierre,exam)}             (1, 3)
[5] {pass(Pierre,exam)}                 (2, 3)
[6] {}                                  (1, 5)
```

因此，你可以从上面的评估中看到 Pierre 通过了考试，因为他为考试进行了学习，不像 John 因为没有学习所以没有通过考试。

让我们考虑更复杂的多个实体的例子，它们执行以下几个动作：

- 有 2 只狗 rover(r) 和 alex(a)。
- 有 1 只猫 garfield(g)。
- 有 1 只狐狸 felix(f)。
- 2 个动物 alex (a) 和 felix(f) 正在奔跑，记作函数 runs()。
- 2 个动物 rover(r) 和 garfield(g) 睡着了，记作 sleeps()。
- 2 个动物 felix(f) 和 alex(a) 可以跳过其他 2 个动物，记作函数 jumps_over()。

考虑到所有这些假设，下面的代码片段基于前面提到的域和基于实体和函数的赋值，构建了一个基于 FOL 的模型。建立模型之后，我们评估各种不同的基于 FOL 的表达式，以决定它们的输出，并像我们前面所做的一样证明一些定理：

```
# define symbols (entities\functions) and their values
rules = """
    rover => r
    felix => f
    garfield => g
    alex => a
    dog => {r, a}
    cat => {g}
```

```
    fox => {f}
    runs => {a, f}
    sleeps => {r, g}
    jumps_over => {(f, g), (a, g), (f, r), (a, r)}
    """
val = nltk.Valuation.fromstring(rules)
# view the valuation object of symbols and their assigned values
(dictionary)
In [143]: print val
{'rover': 'r', 'runs': set([('f',), ('a',)]), 'alex': 'a', 'sleeps':
set([('r',), ('g',)]), 'felix': 'f', 'fox': set([('f',)]), 'dog':
set([('a',), ('r',)]), 'jumps_over': set([('a', 'g'), ('f', 'g'), ('a',
'r'), ('f', 'r')]), 'cat': set([('g',)]), 'garfield': 'g'}

# define domain and build FOL based model
dom = {'r', 'f', 'g', 'a'}
m = nltk.Model(dom, val)

# evaluate various expressions
In [148]: print m.evaluate('jumps_over(felix, rover) & dog(rover) &
runs(rover)', None)
False

In [149]: print m.evaluate('jumps_over(felix, rover) & dog(rover) &
-runs(rover)', None)
True

In [150]: print m.evaluate('jumps_over(alex, garfield) & dog(alex) &
cat(garfield) & sleeps(garfield)', None)
True

# assign rover to x and felix to y in the domain
g = nltk.Assignment(dom, [('x', 'r'), ('y', 'f')])

# evaluate more expressions based on above assigned symbols
In [152]: print m.evaluate('runs(y) & jumps_over(y, x) & sleeps(x)', g)
True

In [153]: print m.evaluate('exists y. (fox(y) & runs(y))', g)
True
```

前面的代码片段基于不同规则和域的符号的赋值，描述了各种表达式的评价。我们基于预定义的假设，创建了各种基于 FOL 的表达式并查看了它们的结果。例如，第一个表达式给我们结果是 `Flase`，因为 `rover` 无法 `runs()`，第二个和第三个表达式为 `True`，因为它们满足所有条件，如 `felix` 和 `alex` 可以跳过 `rover` 或者 `garfield`，`rover` 是一只不能跑的狗，`garfield` 是一只猫。第二组表达式的评估是基于我们的领域（`dom`），为 `felix` 和 `rover` 指派特定的符号，进而进行评估，在评估表达式时我们传递这个变量（`g`）。我们可以使用 `satisfiers()` 函数，匹配公式或表达式，如下所示：

```
# who are the animals who run?
In [154]: formula = read_expr('runs(x)')
     ...: print m.satisfiers(formula, 'x', g)
set(['a', 'f'])

# animals who run and are also a fox?
In [155]: formula = read_expr('runs(x) & fox(x)')
     ...: print m.satisfiers(formula, 'x', g)
set(['f'])
```

前面的输出简单易懂的，在上述代码中我们评估了开放式的问题，如哪些动物会奔跑？还有哪些动物可以跑，同时还是狐狸？我们在输出中获得了相关的符号，这样你可以映射回实际动物名字（提示：`a` 是 `alex`，`f` 是 `felix`）。我鼓励你通过构建自己的假设、领域和规则

来尝试更多的命题和 FOL 表达式。

7.6 情感分析

我们现在将要讨论本章与第二个主题情感分析相关的概念、技术和实例。即使是非结构化的文本数据，也主要可以分为两大类：基于事实类（客观）和基于观念类（主观）。在本章的开始，我们介绍情感分析的概念时简要地介绍了这两个分类以及如何对含有主观内容的文本较好地进行情感分析。通常，社交媒体、调研和反馈数据都有明显的倾向性，表达着人类的信念、判断、情绪和情感。情感分析也称为意见分析/挖掘，定义为使用如 NLP、字典资源、语言学和机器学习等技术进行主观意见相关的信息提取（这些主观意见包括情绪、态度、心情、情态等），并尝试用这些来计算文本文档所表示的极性的过程。所谓极性，是指文件是否表示积极、消极或中性的情绪。更高级的分析包括寻找更复杂的情绪，如悲伤、快乐、愤怒和讽刺。

通常情况下，可以在几个层次上开展文本数据的情感分析，包括单个语句层次、段落层次和整篇文档作为一个整体。情感分析一般基于整篇文档计算，或是逐句计算之后累加在一起。语义的极性分析通常为文档所表达的正面和负面情感赋予一些分值，然后基于累计的分值为文档赋予一个标签。我们这里将介绍情感分析的两种主要技术：

- 有监督的机器学习。
- 无监督的机器学习。

学习各种用于解决情感分析问题的技术的关键是你可以应用它们解决你自己的问题。我们在这里将看到如何再次把第 4 章的有监督机器学习概念应用到文档所表达的情感的分类上。我们将使用词典（专门用于情感分析的字典或是词汇表）采用无监督的技术计算情感。我们将在与电影评论有关的大型真实数据集上进行实验，这使得该项任务更加有意义。我们将对比不同算法的性能，在极性分析的基础上尝试一些详细的分析，包括评论的主题、情绪、情感。别再迟疑了，让我们开始吧！

7.7 IMDb 电影评论的情感分析

我们将使用一个从互联网电影数据库（IMDb）中获得的电影评论数据集。该数据集包含 50 000 多条电影评论，可以从 http://ai.stanford.edu/~amaas/data/sentiment/上下载，该数据集出自斯坦福大学以及 A. L. Maas、R. E. Daly、P. T. Pham、D. Huang、Andrew Ng 和 C. Potts 等人，并在他们的论文“Learning Word Vectors for Sentiment Analysis”中使用了该数据集。我们将使用该数据集中的 50 000 条电影评论，其中包括评论和对应的正面或负面的情感极性标签。基本上，正面评论是 IMDb 中 6 星以上的电影评论，负面评论则是 IMDb 中低于 5 星的电影评论。在我们开始之前，需要记住的一个事实是即使是很多标记为正面或负面的评论，其中仍分别包含一些负面或正面元素。因此，很多评论中存在一些交集，这使得任务更加困难。情感不是一个可以在数学上计算和证明的数值。情感表达了复杂的情绪、感觉和判断，因此你不要尝试建立一个百分比上完美的模型，而是建立一个数据推广能力良好，能够很好工作的模型。在研究各种技术之前，我们将从一些必要依赖和部件安装开始。

7.7.1 安装依赖程序包

在我们开始情感分析之前，需要安装几个公用函数、数据和依赖的程序包。我们会使用电影评论数据库、一些在我们的实现中会使用到的特殊的程序包，我们会定义几个与前几章使用的函数相似的公用函数，用于文本规范化、特征提取和模型评估。

1. 数据获取与格式化

我们将使用 IMDb 电影评论数据库，每个数据集（训练和测试）原始文本文件的官方地址为 http://ai.stanford.edu/~amaas/data/sentiment/。你可以下载和解压缩这些文件到指定的位置，使用 `review_data_extractor.py` 文件以及本章的代码文件从解压缩目录中提取每个评论，解析它们，整齐地将它们格式化到数据框，将该数据存储为 CSV 文件格式，命名为 `movie_reviews.csv`。另外，你可以从 https://github.com/dipanjanS/text-analytics-with-python/tree/master/Chapter-7 上直接下载解析和格式化文件，这些文件包括所有数据集和代码，是本书的官方资料库。对于每个数据点，数据帧包括两列——评论（`review`）和情感（`sentiment`），其表示一个电影的评论和对应情感（正面或负面）。

2. 文本规范化

我们将使用第 6 章中类似的方法对文本数据进行规范化和标准化处理。为做到这一点，我们将重用第 6 章的 `normalization.py` 模块，并增加一些内容。这里主要增加了 HTML 去除器，用于从文本文档中去除不必要的 HTML 字符，如下所示：

```
from HTMLParser import HTMLParser

class MLStripper(HTMLParser):
    def __init__(self):
        self.reset()
        self.fed = []
    def handle_data(self, d):
        self.fed.append(d)
    def get_data(self):
        return ' '.join(self.fed)

def strip_html(text):
    html_stripper = MLStripper()
    html_stripper.feed(text)
    return html_stripper.get_data()
```

我们也增加了一个新的函数以对特殊的重音字符规范化，并将它们转换为规则的 ASCII 字符，规范化文档中所有的文本。下面的代码实现了这个功能：

```
def normalize_accented_characters(text):
    text = unicodedata.normalize('NFKD',
                                 text.decode('utf-8')
                                 ).encode('ascii', 'ignore')
    return text
```

下面的代码片段描述了整个文本规范化函数，它重用了前几章的规范化模块中的缩写词扩展、词形还原、去除 HTML 封装、去除特殊字符、去除停用词等函数：

```
def normalize_corpus(corpus, lemmatize=True,
                     only_text_chars=False,
                     tokenize=False):

    normalized_corpus = []
```

```
        for index, text in enumerate(corpus):
            text = normalize_accented_characters(text)
            text = html_parser.unescape(text)
            text = strip_html(text)
            text = expand_contractions(text, CONTRACTION_MAP)
            if lemmatize:
                text = lemmatize_text(text)
            else:
            text = text.lower()
        text = remove_special_characters(text)
        text = remove_stopwords(text)
        if only_text_chars:
            text = keep_text_characters(text)

        if tokenize:
            text = tokenize_text(text)
            normalized_corpus.append(text)
        else:
            normalized_corpus.append(text)

    return normalized_corpus
```

为重用此代码，你可以使用 normalization. py 和 contractions. py 文件以及本章提供的代码文件。

3. 特征提取

我们可以重用第 6 章所使用的特征提取函数，该函数是 utils. py 模块的一部分。出于完整性的考虑，函数如下所示：

```
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer

def build_feature_matrix(documents, feature_type='frequency',
                         ngram_range=(1, 1), min_df=0.0, max_df=1.0):

    feature_type = feature_type.lower().strip()

    if feature_type == 'binary':
        vectorizer = CountVectorizer(binary=True, min_df=min_df,
                                     max_df=max_df, ngram_range=ngram_range)
    elif feature_type == 'frequency':
        vectorizer = CountVectorizer(binary=False, min_df=min_df,
                                     max_df=max_df, ngram_range=ngram_range)
    elif feature_type == 'tfidf':
        vectorizer = TfidfVectorizer(min_df=min_df, max_df=max_df,
                                     ngram_range=ngram_range)
    else:
        raise Exception("Wrong feature type entered. Possible values:
        'binary', 'frequency', 'tfidf'")

    feature_matrix = vectorizer.fit_transform(documents).astype(float)
    return vectorizer, feature_matrix
```

你可以尝试该函数提供的不同的特征，包括基于词袋模型的频率、出现次数和基于 TF-IDF 的特征。

4. 模型性能评估

我们使用与第 4 章文本分类相似的指标（准确率、精确率、召回率和 F1 score）评估我们的模型。此外，我们会查看混淆矩阵和每一类详细的分类报告，也就是正例和反例，以评估模型的性能。你可以参考 4.7 节，加深一下对于不同模型评估度量标准的记忆。下面的代码可以帮助我们获得模型的准确率、精确率、召回率和 F1-score：

```
from sklearn import metrics
import numpy as np
import pandas as pd

def display_evaluation_metrics(true_labels, predicted_labels, positive_
class=1):
    print 'Accuracy:', np.round(
                        metrics.accuracy_score(true_labels,
                                               predicted_labels),
                        2)
    print 'Precision:', np.round(
                        metrics.precision_score(true_labels,
                                               predicted_labels,
                                               pos_label=positive_class,
                                               average='binary'),
                        2)
    print 'Recall:', np.round(
                        metrics.recall_score(true_labels,
                                               predicted_labels,
                                               pos_label=positive_class,
                                               average='binary'),
                        2)
    print 'F1 Score:', np.round(
                        metrics.f1_score(true_labels,
                                               predicted_labels,
                                               pos_label=positive_class,
                                               average='binary'),
                        2)
```

我们还将定义一个函数，帮助我们建立混淆矩阵，以评估对实际评论情感标识的模型预测。下面的函数帮助我们实现这一点：

```
def display_confusion_matrix(true_labels, predicted_labels, classes=[1,0]):
    cm = metrics.confusion_matrix(y_true=true_labels,
                                  y_pred=predicted_labels,
                                  labels=classes)
    cm_frame = pd.DataFrame(data=cm,
                            columns=pd.MultiIndex(levels=[['Predicted:'],
                            classes],
                                                  labels=[[0,0],[0,1]]),
                            index=pd.MultiIndex(levels=[['Actual:'],
                            classes],
                                                labels=[[0,0],[0,1]]))
    print cm_frame
```

最后，我们再定义一个函数，通过显示每一类的精确率、召回率、F1 score 和支撑指标（评论的数量）以获得一个详细的分类报告（正例和反例）：

```
def display_classification_report(true_labels, predicted_labels,
classes=[1,0]):
    report = metrics.classification_report(y_true=true_labels,
                                           y_pred=predicted_labels,
                                           labels=classes)
    print report
```

你可以在 utils.py 模块以及其他代码文件中找到上面的函数，你可以随需重用这些函数。此外，需要确认你已经安装了 nltk 和 pattern，因为我们在前面的章节中已经多次使用了它们。

7.7.2 准备数据集

我们将载入电影评论数据，与第 4 章类似，准备两个数据集，分别是训练数据集和测试

数据集。我们将使用训练数据对有监督的模型进行训练，使用测试数据评估模型的性能。对于无监督的模型，我们将直接在测试数据上评估它们，以便与有监督的模型进行性能对比。此外，我们将选择一些正面和负面评论的例子来观察不同模型的表现：

```
import pandas as pd
import numpy as np
# load movie reviews data
dataset = pd.read_csv(r'E:/aclImdb/movie_reviews.csv')
# print sample data
In [235]: print dataset.head()
                                              review sentiment
0  One of the other reviewers has mentioned that ...  positive
1  A wonderful little production. <br /><br />The...  positive
2  I thought this was a wonderful way to spend ti...  positive
3  Basically there's a family where a little boy ...  negative
4  Petter Mattei's "Love in the Time of Money" is...  positive

# prepare training and testing datasets
train_data = dataset[:35000]
test_data = dataset[35000:]

train_reviews = np.array(train_data['review'])
train_sentiments = np.array(train_data['sentiment'])
test_reviews = np.array(test_data['review'])
test_sentiments = np.array(test_data['sentiment'])

# prepare sample dataset for experiments
sample_docs = [100, 5817, 7626, 7356, 1008, 7155, 3533, 13010]
sample_data = [(test_reviews[index],
                test_sentiments[index])
                  for index in sample_docs]
```

我们将在 50 000 条评论数据中选取 35 000 条作为训练数据集，使用剩余的 15 000 条评论评估模型和测试它们。构建训练和测试数据集一般按照行数量 70：30 的比例划分。我们从测试数据集中抽取了 8 条评论，将在下面的章节中仔细研究这些文档，并在完整的测试数据集上评估这些模型。

7.7.3 有监督的机器学习技术

如前所述，在本节我们将使用有监督机器学习的方法构建情感分析模型。该模型从训练数据集中已有的评论和对应的情感中学习，用于对测试数据集中新的评论进行情感预测。这里基本的原则是使用与我们文本分类中相同的概念，被预测的类型包括与电影评论对应的正面情感和负面情感。

我们将遵循 4.3 节中同样的工作流程（见图 4-2）。这些步骤总结如下：

（1）模型训练。

- 训练数据规范化处理。
- 特征提取以及建立特征集和特征向量。
- 使用有监督的机器学习算法（SVM）建立预测模型。

（2）模型测试。

- 测试数据规范化处理。
- 使用训练特征向量生成器提取特征。
- 实现训练好的模型预测测试评论的情感。
- 评估模型性能。

我们将使用第1点中的步骤训练模型，使用前面章节讨论的规范化和特征提取模块：

```
from normalization import normalize_corpus
from utils import build_feature_matrix

# normalization
norm_train_reviews = normalize_corpus(train_reviews, lemmatize=True, only_
text_chars=True)
# feature extraction
vectorizer, train_features = build_feature_matrix(documents=norm_train_
reviews,
                                                  feature_type='tfidf',
                                                  ngram_range=(1, 1),
                                                  min_df=0.0, max_df=1.0)
```

我们现在使用支持向量机（SVM）算法构建模型，该算法已经在第4章文本分类中使用过。具体内容请参考4.6.2节：

```
from sklearn.linear_model import SGDClassifier
# build the model
svm = SGDClassifier(loss='hinge', n_iter=200)
svm.fit(train_features, train_sentiments)
```

上面的代码用于训练分类器并建立模型，该模型存储在 svm 变量中。模型可以用于测试数据集（而不是训练数据集）中新电影评论的情感预测。让我们进行规范化处理，按照工作流程第2步的内容，从测试数据集中提取特征：

```
# normalize reviews
norm_test_reviews = normalize_corpus(test_reviews, lemmatize=True, only_
text_chars=True)
# extract features
test_features = vectorizer.transform(norm_test_reviews)
```

现在我们已经有了整个测试数据集的特征，在进行整个数据集的情感预测和模型预测性能测量之前，让我们看看前面提取的例子文档的预测情况：

```
# predict sentiment for sample docs from test data
In [253]: for doc_index in sample_docs:
     ...:     print 'Review:-'
     ...:     print test_reviews[doc_index]
     ...:     print 'Actual Labeled Sentiment:', test_sentiments[doc_index]
     ...:     doc_features = test_features[doc_index]
     ...:     predicted_sentiment = svm.predict(doc_features)[0]
     ...:     print 'Predicted Sentiment:', predicted_sentiment
     ...:     print
     ...:
     ...:
Review:-
Worst movie, (with the best reviews given it) I've ever seen. Over the top
dialog, acting, and direction. more slasher flick than thriller.With all the
great reviews this movie got I'm appalled that it turned out so silly. shame
on you martin scorsese
Actual Labeled Sentiment: negative
Predicted Sentiment: negative

Review:-
I hope this group of film-makers never re-unites.
Actual Labeled Sentiment: negative
Predicted Sentiment: negative

Review:-
no comment - stupid movie, acting average or worse... screenplay - no sense
at all... SKIP IT!
```

```
Actual Labeled Sentiment: negative
Predicted Sentiment: negative

Review:-
Add this little gem to your list of holiday regulars. It is<br /><br
/>sweet, funny, and endearing
Actual Labeled Sentiment: positive
Predicted Sentiment: positive

Review:-
a mesmerizing film that certainly keeps your attention... Ben Daniels is
fascinating (and courageous) to watch.
Actual Labeled Sentiment: positive
Predicted Sentiment: positive

Review:-
This movie is perfect for all the romantics in the world. John Ritter has
never been better and has the best line in the movie! "Sam" hits close to
home, is lovely to look at and so much fun to play along with. Ben Gazzara
was an excellent cast and easy to fall in love with. I'm sure I've met
Arthur in my travels somewhere. All around, an excellent choice to pick up
any evening.!:-)
Actual Labeled Sentiment: positive
Predicted Sentiment: positive

Review:-
I don't care if some people voted this movie to be bad. If you want the
Truth this is a Very Good Movie! It has every thing a movie should have. You
really should Get this one.
Actual Labeled Sentiment: positive
Predicted Sentiment: negative

Review:-
Worst horror film ever but funniest film ever rolled in one you have got
to see this film it is so cheap it is unbeliaveble but you have to see it
really!!!! P.s watch the carrot
Actual Labeled Sentiment: positive
Predicted Sentiment: negative
```

从上面的输出，你可以看到每条评论、标记的情感以及我们预测的情感，你会看到包括一些负面的和正面的评论。除了最后2个例子外，我们的模型可以正确地识别出大多数例子评论的情感。如果你详细看看最后2个例子，会发现评论中包含一些负面的情感（“worst horror film最恐怖的电影)”，“(voted this movie to be bad给电影差评)”），但是填写评论的人的情感或意见趋向正面。这就我们前面提到的一些正面与负面情感交叉的例子，这使得模型对真实的情感预测变得困难！

现在让我们预测全部测试数据集中的评论，评估一下模型的性能：

```
# predict the sentiment for test dataset movie reviews
predicted_sentiments = svm.predict(test_features)

# evaluate model prediction performance
from utils import display_evaluation_metrics, display_confusion_matrix,
display_classification_report

# show performance metrics
In [270]: display_evaluation_metrics(true_labels=test_sentiments,
     ...:                            predicted_labels=predicted_sentiments,
     ...:                            positive_class='positive')
Accuracy: 0.89
Precision: 0.88
Recall: 0.9
F1 Score: 0.89
```

```
# show confusion matrix
In [271]: display_confusion_matrix(true_labels=test_sentiments,
     ...:                           predicted_labels=predicted_sentiments,
     ...:                           classes=['positive', 'negative'])
                  positive negative
Actual: positive      6770      740
        negative       912     6578

# show detailed per-class classification report
In [272]: display_classification_report(true_labels=test_sentiments,
     ...:                                predicted_labels=predicted_
                                         sentiments,
     ...:                                classes=['positive', 'negative'])
              precision    recall  f1-score   support

   positive       0.88      0.90      0.89      7510
   negative       0.90      0.88      0.89      7490

avg / total       0.89      0.89      0.89     15000
```

上面的输出显示了不同的性能指标，这些指标描述了我们的 SVM 模型对于电影评论情感预测的性能。我们得到了 89% 的情感预测平均准确率，如果将它与有监督学习的文本分类标准基线进行比较，这个结果还是相当不错的。分类报告还显示了每个分类的详细报告，我们可以看到正面和负面情感的 F1-score（精确率和召回率的调和平均值）均是 89%。支撑指标显示了正面评论的数量（7510）和负面评论的数量（7490）。混淆矩阵显示了情感预测准确的评论数量（正面：6770/7510，负面：6578/7490），以及情感预测错误的评论数量（正面：740/7510，负面：912/7490）。尝试不同的特征和有监督学习算法建立更多的模型（第 4 章讨论了不同的特征提取技术）。你可以训练一个更好的模型来获得更加准确的情感预测结果吗？

7.7.4 无监督的词典技术

目前为止，我们使用有标记的训练数据、从电影评论中提取的特征和它们对应的情感来进行模式学习模式。然后，我们将所学习到的知识应用到新的电影评论（测试数据集）上预测它们的情感。通常情况下，你并没有标注好的训练数据集。在这些情况下，你需要采用无监督技术使用专为情感分析组织和准备的知识库、本体、数据库和词典来预测情感。

如上所述，词典是一个字典、一个词汇表或一本书。在我们的例子中，词典就是为情感分析而建立的特殊的字典或词汇表。大多数词典拥有一个包含正面极性和负面极性的单词的词汇表，单词上赋予了相应的分值，使用各种技术（如词的位置、周围词、内容、词性和短语等）为我们想要计算情感的文本赋予分值。汇总这些得分后，我们得到了最终的情感。也可以执行更高级的分析，包括主观性检测、语气检测和模式检测。用于情感分析的各种流行的词典如下所示：

- AFINN 词典。
- Bing Liu 词典。
- MPQA 主观词典。
- SentiWordNet。
- VADER 词典。
- Pattern 词典。

这里无法穷尽全部可用于情感分析的词典，还有其他一些词典在互联网上比较容易获

取。我们将简要介绍这几个词典，并且使用最后三个词典在测试数据集上详细地进行情感分析。尽管这些是无监督的方法，你也可以在训练数据集上使用它们进行情感分析与评估。但是，由于一致性方面的原因，为了与有监督机器学习进行比较，我们将在测试数据集上进行情感分析。

1. AFINN 词典

AFINN 词典由 Finn Årup Nielsen 组织和创建，更加详细的内容参见他的文章“A New ANEW: Evaluation of a Word List for Sentiment Analysis in Microblogs”。最新版本 AFINN-111 由 2477 个单词和短语以及基于情感极性的得分组成。极性基本上意味着正面的、负面的或中性的程度如何，及其对应的数值。你可以从下面地址 www2. imm. dtu. dk/pubdb/views/publication_details. php? id = 6010 上下载。该地址详细介绍了该词典。该词典的作者为词典建立了一个 Python 的封装，你可以直接使用它进行文本情感预测。这些文档存放在 GitHub 上，地址为 https://github. com/fnielsen/afinn。你可以直接安装 `afinn` 函数库，开始情感分析。该函数库甚至包括情感符号和笑容符号。下面是 AFINN-111 词典的例子：

```
abandon        -2
abandoned      -2
abandons       -2
abducted       -2
abduction      -2
...
...
youthful        2
yucky          -2
yummy           3
zealot         -2
zealots        -2
zealous         2
```

该词典基本的思想是将词典中的极性单词和短语以及对应得分的列表（如上所示）载入内存，在文本中查找相同的单词/短语和得分。最后，将得分加在一起，得到了文档最终的情感和得分。下面是官方文档中的一个例子：

```
from afinn import Afinn
afn = Afinn(emoticons=True)

In [281]: print afn.score('I really hated the plot of this movie')
-3.0
In [282]: print afn.score('I really hated the plot of this movie :(')
-5.0
```

你可以使用 `score()` 函数直接评估文本的情感，从上面的输出可以看到该函数给出了情感的合适权重，这些可以广泛用于社交媒体分析，如 Twitter 和 Facebook。

2. Bing Liu 词典

该词典由 Bing Liu 在几年前开发，在 Nitin Jindal 和 Bing Liu 的论文“Identifying Comparative Sentences in Text Documents”中有详细的讨论。你可以在网页 https://www. cs. uic. edu/ ~ liub/FBS/sentiment-analysis. html#lexicon 上获得该词典更加详细的信息，该页面上包括一个下载该文件（RAR 格式）的链接。该词典由 6800 个单词组成，分为两个文件，`positive-words. txt` 文件包含大约 2000 个单词/短语，`negative-words. txt` 文件包含大约 4800 个单词/短语。该词典的核心思想是当识别出文档中的单词时，使用这些单词就可以确定任何文档的正面或负面的极性。考虑到社交媒体网站上经常有一些拼写错误，该词典也包括了一些

拼写错误的单词。

3. MPQA 主观词典

MPQA 代表的是多视角的问题回答，它包括了由匹兹堡大学维护的大量资源，有观点语料库、主观词典、词性标注、基于参数的词汇以及辩论数据集。它们中的很多资源可以用于人类情绪和情感的复杂分析。主观词典由 Theresa Wilson、Janyce Wiebe 和 Paul Hoffmann 维护，并在论文“Recognizing Contextual Polarity in Phrase-Level Sentiment Analysis”中进行了详细的讨论，内容主要集中在语境的极性上。你可以在 http://mpqa.cs.pitt.edu/lexicons/subj_lexicon/上下载这个主观词典，这是它的官方网站。主观性的提示词语可以在档案解压后的 subjclueslen1-HLTEMNLP05.tff 中找到。数据集中的一些示例如下所示：

```
type=weaksubj len=1 word1=abandoned pos1=adj stemmed1=n
priorpolarity=negative
type=weaksubj len=1 word1=abandonment pos1=noun stemmed1=n
priorpolarity=negative
type=weaksubj len=1 word1=abandon pos1=verb stemmed1=y
priorpolarity=negative
type=strongsubj len=1 word1=abase pos1=verb stemmed1=y
priorpolarity=negative
...
...
type=strongsubj len=1 word1=zealously pos1=anypos stemmed1=n
priorpolarity=negative
type=strongsubj len=1 word1=zenith pos1=noun stemmed1=n
priorpolarity=positive
type=strongsubj len=1 word1=zest pos1=noun stemmed1=n priorpolarity=positive
```

你可以参考该数据集附带的 `readme` 文件，以便理解这些数据。

该数据集中的提示性词语由上面提到的该项目的维护人员手工整理和收集。上面提到的各种参数的详细解释如下所示。

- `type`：该参数值是强主观的（`strongsubj`）或是弱主观的（`weaksubj`），强主观表示内容具有较强的主观性，弱主观表示内容具有较弱的主观性。
- `len`：该参数指的是提示词中单词的数量（目前均为长度为 1 的单个单词）。
- `word1`：作为真实单词的一个符号或词干。
- `pos1`：该词的词性，可以是名词（`noun`）、动词（`verb`）、形容词（`adj`）、副词（`adverb`）或是任意词性（`anypos`）。
- `stemmed1`：该参数表示该提示词是（`y`）否（`n`）被词干化，如果词干化，它可以与相同词性的其他变量进行匹配。
- `priorpolarity`：该参数值为负面、正面或中性，表示的是与该提示词（词干）相关情感极性。

将该词典载入数据库或内存（提示：Python 目录可以很好地工作），然后采用与前面词典相似的方法分析任何文本文档相关的情感。

4. SentiWordNet

我们知道 WordNet 也许是最流行的英语语料库之一，它广泛用于语义分析，并引入了同义词集的概念。SentiWordNet 词典是一个用于情感分析和意见挖掘的词典资源。对于 WordNet 中的每个同义词集，SentiWordNet 词典赋予三个情感分值，包括正面极性分值、负面极性分值和客观分值。你可以在官方网站 http://sentiwordnet.isti.cnr.it 上了解更多详细信息，

网站上的研究论文详细地介绍了该词典，同时给出了一个下载该词典的连接。Python 语言的 nltk 程序包提供了直接访问该词典的接口，我们将使用该接口分析电影评论的情感。下面的代码给出了同义词集的示例，以及该示例使用 SentiWordNet 获得的情感分值。

```
import nltk
from nltk.corpus import sentiwordnet as swn
# get synset for 'good'
good = swn.senti_synsets('good', 'n')[0]
# print synset sentiment scores
In [287]: print 'Positive Polarity Score:', good.pos_score()
     ...: print 'Negative Polarity Score:', good.neg_score()
     ...: print 'Objective Score:', good.obj_score()
Positive Polarity Score: 0.5
Negative Polarity Score: 0.0
Objective Score: 0.5
```

既然已经知道了如何使用 sentiwordnet 接口，接下来我们定义一个函数，该函数可以输入文本（在我们的例子里是电影评论）并使用 sentiwordnet 进行情感分析：

```
from normalization import normalize_accented_characters, html_parser, strip_
html

def analyze_sentiment_sentiwordnet_lexicon(review,
                                           verbose=False):
    # pre-process text
    review = normalize_accented_characters(review)
    review = html_parser.unescape(review)
    review = strip_html(review)
    # tokenize and POS tag text tokens
    text_tokens = nltk.word_tokenize(review)
    tagged_text = nltk.pos_tag(text_tokens)
    pos_score = neg_score = token_count = obj_score = 0
    # get wordnet synsets based on POS tags
    # get sentiment scores if synsets are found
    for word, tag in tagged_text:
        ss_set = None
    if 'NN' in tag and swn.senti_synsets(word, 'n'):
        ss_set = swn.senti_synsets(word, 'n')[0]
    elif 'VB' in tag and swn.senti_synsets(word, 'v'):
        ss_set = swn.senti_synsets(word, 'v')[0]
    elif 'JJ' in tag and swn.senti_synsets(word, 'a'):
        ss_set = swn.senti_synsets(word, 'a')[0]
    elif 'RB' in tag and swn.senti_synsets(word, 'r'):
        ss_set = swn.senti_synsets(word, 'r')[0]
    # if senti-synset is found
    if ss_set:
        # add scores for all found synsets
        pos_score += ss_set.pos_score()
        neg_score += ss_set.neg_score()
        obj_score += ss_set.obj_score()
        token_count += 1

# aggregate final scores
final_score = pos_score - neg_score
norm_final_score = round(float(final_score) / token_count, 2)
final_sentiment = 'positive' if norm_final_score >= 0 else 'negative'
if verbose:
    norm_obj_score = round(float(obj_score) / token_count, 2)
    norm_pos_score = round(float(pos_score) / token_count, 2)
    norm_neg_score = round(float(neg_score) / token_count, 2)
    # to display results in a nice table
    sentiment_frame = pd.DataFrame([[final_sentiment, norm_obj_score,
                                     norm_pos_score, norm_neg_score,
                                     norm_final_score]],
                                     columns=pd.MultiIndex(levels
```

```
                                    =[['SENTIMENT STATS:'],
                                                ['Predicted Sentiment',
                                                 'Objectivity',
                                                'Positive', 'Negative',
                                                 'Overall']],
                                                labels=[[0,0,0,0,0],
                                                [0,1,2,3,4]]))
                                                print sentiment_frame

    return final_sentiment
```

上述函数的注释非常清晰。我们以文本（电影评论）作为输入，做一些初始化的预处理，然后进行单词切分和词性标注。对于每对单词和标记，我们检查是否存在与该单词和标记相同的语义同义词集。如果匹配成功，我们将采用第一个语义同义词集，并在对应的变量中记录情感的得分，最后我们将这些得分累计在一起。现在可以从下面的代码中看看上面的函数在评论示例中的执行情况（已经从测试数据中创建了变量 sample_data）：

```
# detailed sentiment analysis for sample reviews
In [292]: for review, review_sentiment in sample_data:
     ...:     print 'Review:'
     ...:     print review
     ...:     print
     ...:     print 'Labeled Sentiment:', review_sentiment
     ...:     print
     ...:     final_sentiment = analyze_sentiment_sentiwordnet_
              lexicon(review,
     ...:
              verbose=True)
     ...:     print '-'*60
     ...:
     ...:
Review:
Worst movie, (with the best reviews given it) I've ever seen. Over the top
dialog, acting, and direction. more slasher flick than thriller.With all the
great reviews this movie got I'm appalled that it turned out so silly. shame
on you martin scorsese

Labeled Sentiment: negative

     SENTIMENT STATS:
  Predicted Sentiment Objectivity Positive Negative Overall
0            negative        0.83     0.08     0.09   -0.01
------------------------------------------------------------
Review:
I hope this group of film-makers never re-unites.

Labeled Sentiment: negative

     SENTIMENT STATS:
  Predicted Sentiment Objectivity Positive Negative Overall
0            negative        0.71     0.04     0.25   -0.21
------------------------------------------------------------
Review:
no comment - stupid movie, acting average or worse... screenplay - no sense
at all... SKIP IT!

Labeled Sentiment: negative

     SENTIMENT STATS:
  Predicted Sentiment Objectivity Positive Negative Overall
0            negative        0.81     0.04     0.15   -0.11
------------------------------------------------------------
Review:
Add this little gem to your list of holiday regulars. It is<br /><br
/>sweet, funny, and endearing
```

```
Labeled Sentiment: positive

     SENTIMENT STATS:
  Predicted Sentiment Objectivity Positive Negative Overall
0           positive        0.76     0.18     0.06    0.13
-----------------------------------------------------------
Review:
a mesmerizing film that certainly keeps your attention... Ben Daniels is
fascinating (and courageous) to watch.

Labeled Sentiment: positive

     SENTIMENT STATS:
  Predicted Sentiment Objectivity Positive Negative Overall
0           positive        0.84     0.14     0.03    0.11
-----------------------------------------------------------
Review:
This movie is perfect for all the romantics in the world. John Ritter has
never been better and has the best line in the movie! "Sam" hits close to
home, is lovely to look at and so much fun to play along with. Ben Gazzara
was an excellent cast and easy to fall in love with. I'm sure I've met
Arthur in my travels somewhere. All around, an excellent choice to pick up
any evening.!:-)

Labeled Sentiment: positive

     SENTIMENT STATS:
  Predicted Sentiment Objectivity Positive Negative Overall
0           positive        0.75      0.2     0.05    0.15
-----------------------------------------------------------
Review:
I don't care if some people voted this movie to be bad. If you want the
Truth this is a Very Good Movie! It has every thing a movie should have. You
really should Get this one.

Labeled Sentiment: positive

     SENTIMENT STATS:
  Predicted Sentiment Objectivity Positive Negative Overall
0           positive        0.73     0.21     0.06    0.15
-----------------------------------------------------------
Review:
Worst horror film ever but funniest film ever rolled in one you have got
to see this film it is so cheap it is unbeliaveble but you have to see it
really!!!! P.s watch the carrot

Labeled Sentiment: positive
     SENTIMENT STATS:
  Predicted Sentiment Objectivity Positive Negative Overall
0           positive        0.79     0.13     0.08    0.05
-----------------------------------------------------------
```

从上面的输出你可以看到与每个情感得分和整体得分相关的详细统计数据，可以与每个评论标准的情感进行比较。非常有意思的是，相对于有监督的机器学习技术，我们准确预测了全部示例评论。但是，该技术在完整测试数据集上表现如何呢？下面将给出答案！

```
# predict sentiment for test movie reviews dataset
sentiwordnet_predictions = [analyze_sentiment_sentiwordnet_lexicon(review)
                            for review in test_reviews]

from utils import display_evaluation_metrics, display_confusion_matrix,
display_classification_report

# get model performance statistics
In [295]: print 'Performance metrics:'
     ...: display_evaluation_metrics(true_labels=test_sentiments,
```

```
...:                                predicted_labels=sentiwordnet_
                                    predictions,
...:                                positive_class='positive')
...: print '\nConfusion Matrix:'
...: display_confusion_matrix(true_labels=test_sentiments,
...:                               predicted_labels=sentiwordnet_
                                   predictions,
...:                               classes=['positive', 'negative'])
...: print '\nClassification report:'
...: display_classification_report(true_labels=test_sentiments,
...:                                    predicted_labels=sentiwordnet_
                                        predictions,
...:                                    classes=['positive', 'negative'])
Performance metrics:
Accuracy: 0.59
Precision: 0.56
Recall: 0.92
F1 Score: 0.7

Confusion Matrix:
                Predicted:
                  positive negative
Actual: positive      6941      569
        negative      5510     1980

Classification report:
             precision    recall  f1-score   support

   positive       0.56      0.92      0.70      7510
   negative       0.78      0.26      0.39      7490

avg / total       0.67      0.59      0.55     15000
```

该模型的情感预测准确率在60%左右，F1 score 大约是70%。如果研究一些详细的分类报告和混淆矩阵，你会看到我们将6941/7510的正面评论正确地分类到正面类别，但我们错误地将5510/7490的负面评论分类到正面类别，这错误率太高了！解决这个问题的方法是在函数中稍微修改一下逻辑，调整一下决定文本是正面或是负面的整体情感评分的阈值，从0调整到0.1或更高。请在该参数下做试验，看看能得到什么结果。

5. VADER 词典

VADER 的意思是 Valence Aware Dictionary and S entiment Reasoner。该词典是一个基于规则的情感分析框架，专为社交媒体分析情感而建立。该词典由 C. J. Hutto 和 Eric Gilbert 开发，你可以在文章“VADER: A Parsimonious Rule-based Model for Sentiment Analysis of Social Media Text”中找到详细信息。你可以在 https://github.com/cjhutto/vaderSentiment 中了解更多信息、下载数据集或安装函数库，该地址包含了与 VADER 词典相关的全部资源。`vader_sentiment_lexicon.txt` 文件中包含了各种词必要的情感分数，包括单词、表情符号，甚至是俚语的缩写（如 `lol`、`wtf`、`nah` 等）。该词典有超过9000个词汇特征，并进一步整理成为7500个带有正确验证得分的词汇特征。每个特征的范围从-4（非常负面）到4（非常正面），允许0（代表中性）或没有（N/A）。这种综合处理是保留所有的词汇特征有一个非零的平均评级，其标准差小于2.5，这是由十个独立的等级评定者评分累加得到的。VADER 词典的示例如下：

```
)-:<   -2.2   0.4     [-2, -2, -2, -2, -2, -2, -3, -3, -2, -2]
)-:{   -2.1   0.9434  [-1, -3, -2, -1, -2, -2, -3, -4, -1, -2]
):     -1.8   0.87178 [-1, -3, -1, -2, -1, -3, -1, -3, -1, -2]
...
```

```
...
resolved      0.7    0.78102  [1, 2, 0, 1, 1, 0, 2, 0, 0, 0]
resolvent     0.7    0.78102  [1, 0, 1, 2, 0, -1, 1, 1, 1, 1]
resolvents    0.4    0.66332  [2, 0, 0, 1, 0, 0, 1, 0, 0, 0]
...
...
}:-(   -2.1   0.7      [-2, -1, -2, -2, -2, -4, -2, -2, -2, -2]
}:-)    0.3   1.61555  [1, 1, -2, 1, -1, -3, 2, 2, 1, 1]
```

每一行代表词典中一个独一无二的词，可以是一个单词或是一个表情符号。第一列表示单词/表情符号，第二列表示平均得分，第三列是标准差，最后一列是十个独立评分者给出的得分。nltk 提供了一个使用 VADER 词典的接口，下面的函数可以对任何文档执行同样的情感分析：

```
from nltk.sentiment.vader import SentimentIntensityAnalyzer

def analyze_sentiment_vader_lexicon(review,
                                    threshold=0.1,
                                    verbose=False):
    # pre-process text
    review = normalize_accented_characters(review)
    review = html_parser.unescape(review)
    review = strip_html(review)
    # analyze the sentiment for review
    analyzer = SentimentIntensityAnalyzer()
    scores = analyzer.polarity_scores(review)
    # get aggregate scores and final sentiment
    agg_score = scores['compound']
    final_sentiment = 'positive' if agg_score >= threshold\
                                 else 'negative'
    if verbose:
        # display detailed sentiment statistics
        positive = str(round(scores['pos'], 2)*100)+'%'
        final = round(agg_score, 2)
        negative = str(round(scores['neg'], 2)*100)+'%'
        neutral = str(round(scores['neu'], 2)*100)+'%'
        sentiment_frame = pd.DataFrame([[final_sentiment, final, positive,
                                         negative, neutral]],
                columns=pd.MultiIndex(levels=[['SENTIMENT STATS:'],
                                              ['Predicted Sentiment',
                                               'Polarity Score',
                                               'Positive', 'Negative',
                                               'Neutral']],
                                      labels=[[0,0,0,0,0],[0,1,2,3,4]]))
        print sentiment_frame

    return final_sentiment
```

该函数帮助计算任何文档（本例中使用电影评论）的情感和各种相关的统计值。注释解释了该函数的主要部分，包括文本预处理、使用 VADER 词典获得必要的情感评分、对它们进行累加、使用我们前面讨论过的一个特定的阈值计算最终的情感（正面/负面）。平均来看，阈值取值 0.1 似乎工作效果最好，但你可以尝试对该参数做进一步的试验。下面的代码告诉我们如何在我们的测试电影评论示例上使用该功能：

```
# get detailed sentiment statistics
In [301]: for review, review_sentiment in sample_data:
     ...:     print 'Review:'
     ...:     print review
     ...:     print
     ...:     print 'Labeled Sentiment:', review_sentiment
     ...:     print
     ...:     final_sentiment = analyze_sentiment_vader_lexicon(review,
```

```
    ...:                                                    threshold=0.1,
    ...:                                                    verbose=True)
    ...:     print '-'*60

Review:
Worst movie, (with the best reviews given it) I've ever seen. Over the top
dialog, acting, and direction. more slasher flick than thriller.With all the
great reviews this movie got I'm appalled that it turned out so silly. shame
on you martin scorsese

Labeled Sentiment: negative

     SENTIMENT STATS:
  Predicted Sentiment Polarity Score Positive Negative Neutral
0           negative          0.03    20.0%    18.0%   62.0%
-----------------------------------------------------------
Review:
I hope this group of film-makers never re-unites.

Labeled Sentiment: negative

     SENTIMENT STATS:
  Predicted Sentiment Polarity Score Positive Negative Neutral
0           positive          0.44    33.0%     0.0%   67.0%
-----------------------------------------------------------
Review:
no comment - stupid movie, acting average or worse... screenplay - no sense
at all... SKIP IT!

Labeled Sentiment: negative

     SENTIMENT STATS:
  Predicted Sentiment Polarity Score Positive Negative Neutral
0           negative          -0.8     0.0%    40.0%   60.0%
-----------------------------------------------------------
Review:
Add this little gem to your list of holiday regulars. It is<br /><br />sweet,
funny, and endearing
Labeled Sentiment: positive

     SENTIMENT STATS:
  Predicted Sentiment Polarity Score Positive Negative Neutral
0           positive          0.82    40.0%     0.0%   60.0%
-----------------------------------------------------------
Review:
a mesmerizing film that certainly keeps your attention... Ben Daniels is
fascinating (and courageous) to watch.

Labeled Sentiment: positive

     SENTIMENT STATS:
  Predicted Sentiment Polarity Score Positive Negative Neutral
0           positive          0.71    31.0%     0.0%   69.0%
-----------------------------------------------------------
Review:
This movie is perfect for all the romantics in the world. John Ritter has
never been better and has the best line in the movie! "Sam" hits close to
home, is lovely to look at and so much fun to play along with. Ben Gazzara
was an excellent cast and easy to fall in love with. I'm sure I've met
Arthur in my travels somewhere. All around, an excellent choice to pick up
any evening.!:-)

Labeled Sentiment: positive

     SENTIMENT STATS:
  Predicted Sentiment Polarity Score Positive Negative Neutral
0           positive          0.99    37.0%     2.0%   61.0%
-----------------------------------------------------------
```

```
Review:
I don't care if some people voted this movie to be bad. If you want the
Truth this is a Very Good Movie! It has every thing a movie should have. You
really should Get this one.

Labeled Sentiment: positive

    SENTIMENT STATS:
  Predicted Sentiment Polarity Score Positive Negative Neutral
0           negative          -0.16    17.0%    14.0%   69.0%
-----------------------------------------------------------
Review:
Worst horror film ever but funniest film ever rolled in one you have got
to see this film it is so cheap it is unbeliaveble but you have to see it
really!!!! P.s watch the carrot
Labeled Sentiment: positive

    SENTIMENT STATS:
  Predicted Sentiment Polarity Score Positive Negative Neutral
0           positive           0.49    11.0%    11.0%   77.0%
-----------------------------------------------------------
```

除了 Positive、Negative 和 Neutral 三列，上面的统计与前面的类似，这三列表示文档是正面、负面或中性的比例，最终的得分基于极性得分和阈值来决定。下面的代码给出了该模型对整个电影评论测试数据集的情感预测性能：

```
# predict sentiment for test movie reviews dataset
vader_predictions = [analyze_sentiment_vader_lexicon(review, threshold=0.1)
                     for review in test_reviews]

# get model performance statistics
In [302]: print 'Performance metrics:'
     ...: display_evaluation_metrics(true_labels=test_sentiments,
     ...:                            predicted_labels=vader_predictions,
     ...:                            positive_class='positive')
     ...: print '\nConfusion Matrix:'
     ...: display_confusion_matrix(true_labels=test_sentiments,
     ...:                          predicted_labels=vader_predictions,
     ...:                          classes=['positive', 'negative'])
     ...: print '\nClassification report:'
     ...: display_classification_report(true_labels=test_sentiments,
     ...:                               predicted_labels=vader_predictions,
     ...:                               classes=['positive', 'negative'])
Performance metrics:
Accuracy: 0.7
Precision: 0.65
Recall: 0.86
F1 Score: 0.74

Confusion Matrix:
                Predicted:
                  positive negative
Actual: positive      6434     1076
        negative      3410     4080

Classification report:
             precision    recall  f1-score   support

   positive       0.65      0.86      0.74      7510
   negative       0.79      0.54      0.65      7490

avg / total       0.72      0.70      0.69     15000
```

上面的指标说明该模型在情感预测方面的准确率是70%左右，F1-socre 接近75%，这明显好于我们前面的模型。还要注意到在7510个正面电影评论中我们可以正确地预测到6434

个，在7490个负面电影评论中可以正确预测4080个。

6. Pattern 词典

pattern是一个完整的自然语言处理、文本分析和信息检索的函数包。在前面的章节中已经对它进行了详细讨论，并且已经使用它解决了一些问题。该函数包由CLiPS（Computational Linguistics & Psycholinguistics）开发，该研究中心隶属于安特卫普大学艺术学院语言学系。该函数包有一个情感分析模块，以及情绪分析和文本模式分析模块。

对于情感分析，该模块通过将文本分为句子、词干化、对词干进行词性标注等步骤分析任何文本。该函数包使用自己的主观情感词典，你可以访问官方资料库 https://github.com/clips/pattern/blob/master/pattern/text/en/en-sentiment.xml。该词典包含极性、主观、强度、自信的评分，以及词性标签、WordNet 标识等。函数库使用该词典计算文本的整体极性和主观评分。pattern 推荐的阈值为0.1，高于该值时情感为正面，低于该值时为负面。

你可以使用 pattern 函数包提供的情绪和情态函数分析文本中的情绪和情态。情绪函数帮助确定一个具体文本文档所表达的情绪。该函数基于其内容返回 INDICATIVE、IMPERATIVE、CONDITIONAL 或 SUBJUNCTIVE。图7-2中的表格详细地讨论了每类情绪，免费的官方文档由 CLiPS pattern 提供。USE 列讨论了每种情绪典型使用模式，EXAMPLE 列提供了一些英语中的实例。

MOOD	FORM	USE	EXAMPLE
INDICATIVE	none of the below	fact, belief	*It rains.*
IMPERATIVE	infinitive without *to*	command, warning	*Don't rain!*
CONDITIONAL	*would, could, should, may,* or *will, can* + *if*	conjecture	*It might rain.*
SUBJUNCTIVE	*wish, were,* or *it is* + infinitive	wish, opinion	*I hope it rains.*

图7-2 不同类型的情绪和例子（图片来自于 CLiPS pattern）

任意文本的情态代表着文本整体上所表示的确定性的程度。它是0到1之间的数值。如果值大于0.5表示真实文本具有高确定性，值小于0.5表示的是祝愿、希望，具有低确定性。我们现在将定义一个使用 Pattern 词典来分析文本情感的函数：

```
from pattern.en import sentiment, mood, modality

def analyze_sentiment_pattern_lexicon(review, threshold=0.1,
                                      verbose=False):
    # pre-process text
    review = normalize_accented_characters(review)
    review = html_parser.unescape(review)
    review = strip_html(review)
    # analyze sentiment for the text document
    analysis = sentiment(review)
    sentiment_score = round(analysis[0], 2)
    sentiment_subjectivity = round(analysis[1], 2)
    # get final sentiment
    final_sentiment = 'positive' if sentiment_score >= threshold\
                                 else 'negative'
    if verbose:
        # display detailed sentiment statistics
        sentiment_frame = pd.DataFrame([[final_sentiment, sentiment_score,
                                         sentiment_subjectivity]],
                                         columns=pd.MultiIndex(levels
                                         =[['SENTIMENT STATS:'],
                                                   ['Predicted Sentiment',
```

```
                                                'Polarity Score',
                                                'Subjectivity Score']],
                                                labels=[[0,0,0],
                                                [0,1,2]]))
        print sentiment_frame
        assessment = analysis.assessments
        assessment_frame = pd.DataFrame(assessment,
                                columns=pd.MultiIndex(levels=[['DETAILED
                                ASSESSMENT STATS:'],
                                                ['Key Terms', 'Polarity
                                                Score',
                                                'Subjectivity Score',
                                                'Type']],
                                                labels=[[0,0,0,0],
                                                [0,1,2,3]]))
                                                print assessment_frame
                                                print

    return final_sentiment
```

现在将测试一下我们定义的用于测试电影评论例子的函数，并观察其结果。基于累积的情感极性得分和官方文档的几个实验和推荐，将阈值设为0.1作为文档正面和负面情感的临界点：

```
# get detailed sentiment statistics
In [303]: for review, review_sentiment in sample_data:
     ...:     print 'Review:'
     ...:     print review
     ...:     print
     ...:     print 'Labeled Sentiment:', review_sentiment
     ...:     print
     ...:     final_sentiment = analyze_sentiment_pattern_lexicon(review,
     ...:
threshold=0.1,
     ...:
verbose=True)
     ...:     print '-'*60

Review:
Worst movie, (with the best reviews given it) I've ever seen. Over the top
dialog, acting, and direction. more slasher flick than thriller.With all the
great reviews this movie got I'm appalled that it turned out so silly. shame
on you martin scorsese

Labeled Sentiment: negative

     SENTIMENT STATS:
  Predicted Sentiment Polarity Score Subjectivity Score
0            negative           0.06               0.62
  DETAILED ASSESSMENT STATS:
                  Key Terms Polarity Score Subjectivity Score  Type
0                   [worst]          -1.0             1.000  None
1                    [best]           1.0             0.300  None
2                     [top]           0.5             0.500  None
3                  [acting]           0.0             0.000  None
4                    [more]           0.5             0.500  None
5                   [great]           0.8             0.750  None
6                [appalled]          -0.8             1.000  None
7                   [silly]          -0.5             0.875  None

------------------------------------------------------------
Review:
I hope this group of film-makers never re-unites.
Labeled Sentiment: negative

     SENTIMENT STATS:
```

```
  Predicted Sentiment Polarity Score Subjectivity Score
0            negative            0.0                0.0
Empty DataFrame
Columns: [(DETAILED ASSESSMENT STATS:, Key Terms), (DETAILED ASSESSMENT
STATS:, Polarity Score), (DETAILED ASSESSMENT STATS:, Subjectivity Score),
(DETAILED ASSESSMENT STATS:, Type)]
Index: []

---------------------------------------------------------
Review:
no comment - stupid movie, acting average or worse... screenplay - no sense
at all... SKIP IT!

Labeled Sentiment: negative

     SENTIMENT STATS:
  Predicted Sentiment Polarity Score Subjectivity Score
0            negative          -0.36                0.5
  DETAILED ASSESSMENT STATS:
                 Key Terms Polarity Score Subjectivity Score  Type
0                [stupid]          -0.80                1.0  None
1                [acting]           0.00                0.0  None
2               [average]          -0.15                0.4  None
3              [worse, !]          -0.50                0.6  None

---------------------------------------------------------
Review:
Add this little gem to your list of holiday regulars. It is<br /><br
/>sweet, funny, and endearing

Labeled Sentiment: positive

     SENTIMENT STATS:
  Predicted Sentiment Polarity Score Subjectivity Score
0            positive          0.19               0.67
  DETAILED ASSESSMENT STATS:
                 Key Terms Polarity Score Subjectivity Score  Type
0                [little]       -0.1875                0.5  None
1                 [funny]        0.2500                1.0  None
2             [endearing]        0.5000                0.5  None

---------------------------------------------------------
Review:
a mesmerizing film that certainly keeps your attention... Ben Daniels is
fascinating (and courageous) to watch.
Labeled Sentiment: positive

     SENTIMENT STATS:
  Predicted Sentiment Polarity Score Subjectivity Score
0            positive           0.4               0.71
  DETAILED ASSESSMENT STATS:
                 Key Terms Polarity Score Subjectivity Score  Type
0            [mesmerizing]       0.300000           0.700000  None
1              [certainly]       0.214286           0.571429  None
2            [fascinating]       0.700000           0.850000  None

---------------------------------------------------------
Review:
This movie is perfect for all the romantics in the world. John Ritter has
never been better and has the best line in the movie! "Sam" hits close to
home, is lovely to look at and so much fun to play along with. Ben Gazzara
was an excellent cast and easy to fall in love with. I'm sure I've met
Arthur in my travels somewhere. All around, an excellent choice to pick up
any evening.!:-)

Labeled Sentiment: positive

     SENTIMENT STATS:
```

```
  Predicted Sentiment Polarity Score Subjectivity Score
0            positive           0.66               0.73
   DETAILED ASSESSMENT STATS:
                  Key Terms Polarity Score Subjectivity Score  Type
0                 [perfect]       1.000000           1.000000  None
1                  [better]       0.500000           0.500000  None
2                 [best, !]       1.000000           0.300000  None
3                  [lovely]       0.500000           0.750000  None
4               [much, fun]       0.300000           0.200000  None
5               [excellent]       1.000000           1.000000  None
6                    [easy]       0.433333           0.833333  None
7                    [love]       0.500000           0.600000  None
8                    [sure]       0.500000           0.888889  None
9            [excellent, !]       1.000000           1.000000  None
10                   [:-)]       0.500000           1.000000  mood

------------------------------------------------------------
Review:
I don't care if some people voted this movie to be bad. If you want the
Truth this is a Very Good Movie! It has every thing a movie should have.
You really should Get this one.
Labeled Sentiment: positive

    SENTIMENT STATS:
  Predicted Sentiment Polarity Score Subjectivity Score
0            positive           0.17               0.55
  DETAILED ASSESSMENT STATS:
                  Key Terms Polarity Score Subjectivity Score  Type
0                     [bad]          -0.7           0.666667  None
1           [very, good, !]           1.0           0.780000  None
2                  [really]           0.2           0.200000  None

------------------------------------------------------------
Review:
Worst horror film ever but funniest film ever rolled in one you have got
to see this film it is so cheap it is unbeliaveble but you have to see it
really!!!! P.s watch the carrot

Labeled Sentiment: positive

    SENTIMENT STATS:
  Predicted Sentiment Polarity Score Subjectivity Score
0            negative          -0.04               0.63
  DETAILED ASSESSMENT STATS:
                  Key Terms Polarity Score Subjectivity Score  Type
0                   [worst]      -1.000000               1.0  None
1                   [cheap]       0.400000               0.7  None
2      [really, !, !, !, !]       0.488281               0.2  None

------------------------------------------------------------
```

上面的分析显示了每个评论示例的情感、极性和主观评分。此外，我们也看到了关键词和情感以及它们的极性评分，这些决定了每个评论的整体情感。你可以看到，当计算情感和极性时，声明和表情符号也被赋予重要性的和权重。下面的片段描述了测试电影评论例子的情绪和情态。

```
In [304]: for review, review_sentiment in sample_data:
     ...:     print 'Review:'
     ...:     print review
     ...:     print 'Labeled Sentiment:', review_sentiment
     ...:     print 'Mood:', mood(review)
     ...:     mod_score = modality(review)
     ...:     print 'Modality Score:', round(mod_score, 2)
     ...:     print 'Certainty:', 'Strong' if mod_score > 0.5 \
     ...:                                   else 'Medium' if mod_score > 0.35 \
     ...:                                                 else 'Low'
```

```
...:     print '-'*60
Review:
Worst movie, (with the best reviews given it) I've ever seen. Over the top
dialog, acting, and direction. more slasher flick than thriller.With all the
great reviews this movie got I'm appalled that it turned out so silly. shame
on you martin scorsese
Labeled Sentiment: negative
Mood: indicative
Modality Score: 0.75
Certainty: Strong
------------------------------------------------------------
Review:
I hope this group of film-makers never re-unites.
Labeled Sentiment: negative
Mood: subjunctive
Modality Score: -0.25
Certainty: Low
------------------------------------------------------------
Review:
no comment - stupid movie, acting average or worse... screenplay - no sense
at all... SKIP IT!
Labeled Sentiment: negative
Mood: indicative
Modality Score: 0.75
Certainty: Strong
------------------------------------------------------------
Review:
Add this little gem to your list of holiday regulars. It is<br /><br
/>sweet, funny, and endearing
Labeled Sentiment: positive
Mood: imperative
Modality Score: 1.0
Certainty: Strong
------------------------------------------------------------
Review:
a mesmerizing film that certainly keeps your attention... Ben Daniels is
fascinating (and courageous) to watch.
Labeled Sentiment: positive
Mood: indicative
Modality Score: 0.75
Certainty: Strong
------------------------------------------------------------
Review:
This movie is perfect for all the romantics in the world. John Ritter has
never been better and has the best line in the movie! "Sam" hits close to
home, is lovely to look at and so much fun to play along with. Ben Gazzara
was an excellent cast and easy to fall in love with. I'm sure I've met
Arthur in my travels somewhere. All around, an excellent choice to pick up
any evening.!:-)
Labeled Sentiment: positive
Mood: indicative
Modality Score: 0.58
Certainty: Strong
------------------------------------------------------------
Review:
I don't care if some people voted this movie to be bad. If you want the
Truth this is a Very Good Movie! It has every thing a movie should have. You
really should Get this one.
Labeled Sentiment: positive
Mood: conditional
Modality Score: 0.28
Certainty: Low
------------------------------------------------------------
Review:
Worst horror film ever but funniest film ever rolled in one you have got
to see this film it is so cheap it is unbeliaveble but you have to see it
really!!!! P.s watch the carrot
Labeled Sentiment: positive
```

```
Mood: indicative
Modality Score: 0.75
Certainty: Strong
-----------------------------------------------------------
```

上面的输出描述了每个评论的情绪、情态得分以及确定性因子。看到"Add this little gem…"被正确地赋予了情绪，它是命令语气，"I hope this…"被正确地与虚拟语气关联，也是很有趣的。其他更多的评论是陈述语气，这是显而易见的，因为它们表达了电影评论人的观点。当使用如"hope"和"if"单词时，确定性是比较低的，对于观点比较强烈的评论，确定性就较高。

最后，与前面的模型一样，我们要在整个电影评论数据集上评估一下这个模型的情感预测性能。下面的片段实现了同样功能：

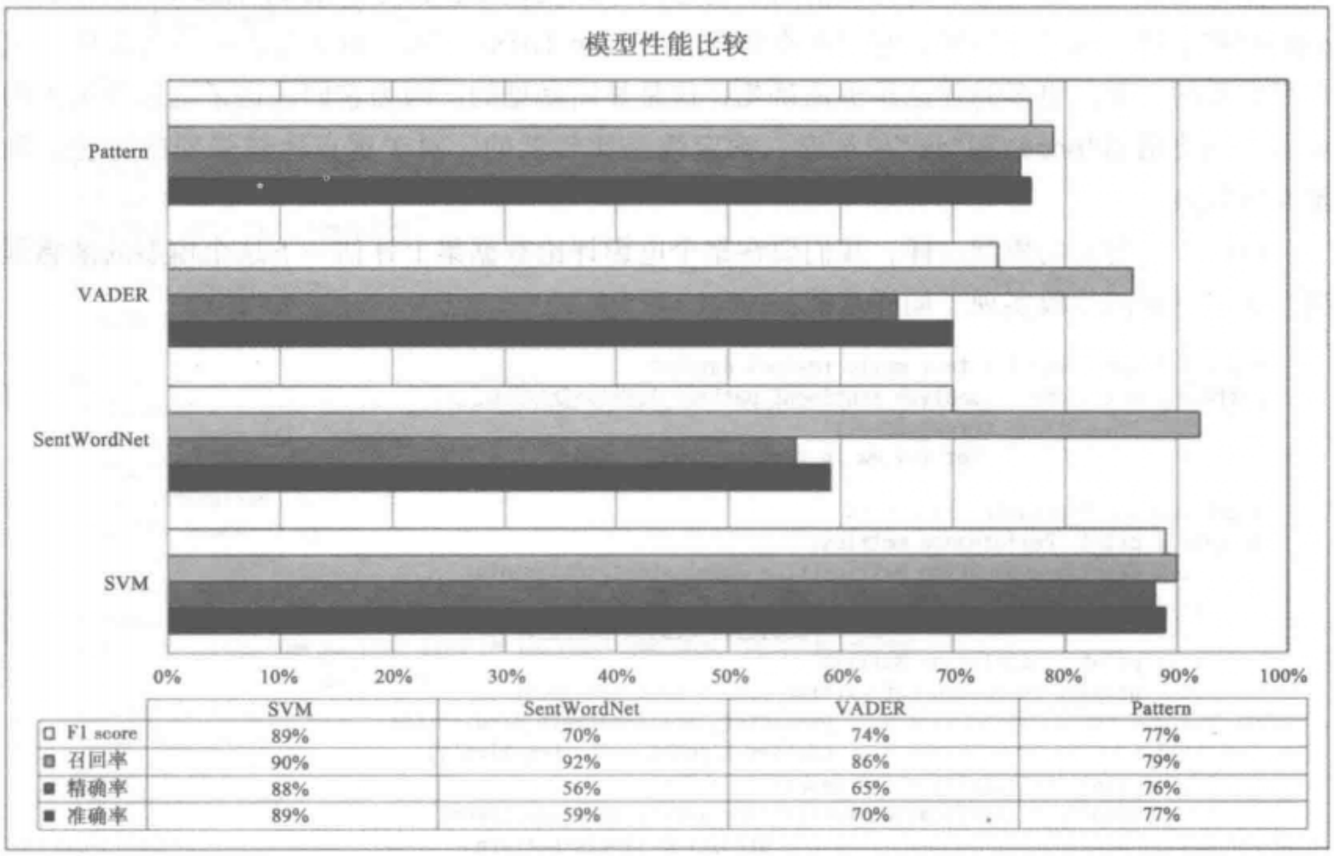
```
# predict sentiment for test movie reviews dataset
pattern_predictions = [analyze_sentiment_pattern_lexicon(review,
                       threshold=0.1)
                        for review in test_reviews]

# get model performance statistics
In [307]: print 'Performance metrics:'
     ...: display_evaluation_metrics(true_labels=test_sentiments,
     ...:                            predicted_labels=pattern_predictions,
     ...:                            positive_class='positive')
     ...: print '\nConfusion Matrix:'
     ...: display_confusion_matrix(true_labels=test_sentiments,
     ...:                          predicted_labels=pattern_predictions,
     ...:                          classes=['positive', 'negative'])
     ...: print '\nClassification report:'
     ...: display_classification_report(true_labels=test_sentiments,
     ...:                               predicted_labels=pattern_
                                        predictions,
     ...:                               classes=['positive', 'negative'])
Performance metrics:
Accuracy: 0.77
Precision: 0.76
Recall: 0.79
F1 Score: 0.77

Confusion Matrix:
                Predicted:
                  positive negative
Actual: positive      5958     1552
        negative      1924     5566

Classification report:
             precision    recall  f1-score   support

   positive       0.76      0.79      0.77      7510
   negative       0.78      0.74      0.76      7490

avg / total       0.77      0.77      0.77     15000
```

该模型给出了更好的且更加平衡的正面和负面情感类型预测的性能。对于该模型，我们得到了平均77%的情感预测准确率和77%的F1-score。尽管与我们前面的模型相比，正确的正面评论预测下降到5958/7510，但负面评论的正确预测数量明显地提升到5566/7490。

7.7.5 模型性能比较

我们建立了一个有监督的分类模型和三个无监督的词典模型来预测电影评论的情绪。为了计算情绪，对于每个模型，我们研究了它的详细分析和统计过程。我们还评估了每个模

型的标准指标，如精确率、召回率、准确率和 F1-score 等。在本节中，我们将简要地查看一下每个模型与其他模型之间的性能比较。图 7-3 显示了模型的性能度量和所有模型指标的可视化对比。



	SVM	SentWordNet	VADER	Pattern
F1 score	89%	70%	74%	77%
召回率	90%	92%	86%	79%
精确率	88%	56%	65%	76%
准确率	89%	59%	70%	77%

图 7-3 情感分析模型性能比较

从图 7-3 的视图和表格中，非常明显地看到使用 SVM 的有监督模型的结果最好，这是可以预测的，主要是因为使用了 35 000 个电影评论对它进行训练。在测试数据集上，无监督的技术中 Pattern 字典的性能最好。这意味着该模型会一直性能最好吗？绝对不是。这取决于你正在分析的数据。请记住，在评估任何模型时，要考虑各种模型，也要评估所有的度量标准，而不仅仅是一两个指标。图表中的一些模型具有很高的召回率，但精确率不高，这意味着这些模型倾向于做出更错误的预测或误报。当试验不同的特征、词典和技术时，你可以重用这些基准，评估更多的情感分析模型。

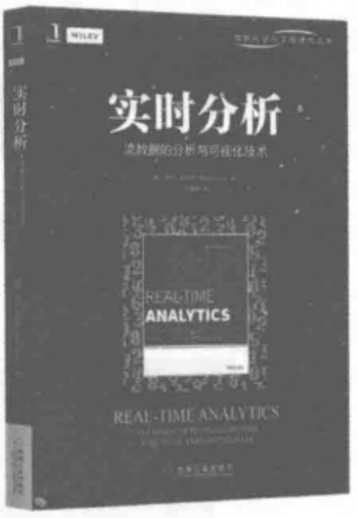
7.8 小结

在最后一章，我们重点阐述了文本数据语义和情感分析方面的一些主题。我们回顾了第 1 章中与语言语义相关的几个概念。我们详细研究了 WordNet 语料库，通过实际例子探索了同义词集概念。我们也使用同义词集和真实世界的例子，分析了第 1 章里各种词汇的语义关系。我们研究了一些关系，包括蕴含、同音词、同形异义词、同义词、反义词、下位词、上位词、整体词和部分词。通过使用不同同义词集上共同上位词的例子，详细讨论了语义关系和相似性计算技术。通过实例对语义和信息提取方面广泛使用的主流技术进行了介绍，包括语义消歧和命名实体识别。除了语义关系外，我们也重温了语义表示的相关概念，即谓词逻辑和一阶逻辑。我们使用定理证明并从计算上评估了命题和逻辑表示。

接下来，我们介绍了情感分析和意见挖掘的概念，看到它在诸如社交媒体、调查和反馈数据等不同领域如何应用。我们举了一个对 IMDb 真实电影评论进行情感分析的实际例子，建立了几个模型，包括有监督机器学习模型和基于词典的无监督模型。我们详细研究了每种技术及其结果，并比较了所有模型的性能。

本书即将结束。我希望这里讨论的各种概念和技术对你有所帮助，希望你在解决文本分析和自然语言处理世界中遇到有挑战的问题时，可以用到本书的知识和技术。到目前为止，你也许会看到，在非结构化文本分析的世界里还有很多未被探索的领域。我祝你一切顺利，送给你源于“奥卡姆的剃刀”的临别赠言：“有时候最简单的解决办法就是最好的”。

推荐阅读



统计学习导论——基于R应用

作者：加雷斯·詹姆斯 等 ISBN：978-7-111-49771-4 定价：79.00元

应用预测建模

作者：马克斯·库恩 等 ISBN：978-7-111-53342-9 定价：99.00元

实时分析：流数据的分析与可视化技术

作者：拜伦·埃利斯 ISBN：978-7-111-53216-3 定价：79.00元

数据挖掘与商务分析：R语言

作者：约翰尼斯·莱道尔特 ISBN：978-7-111-54940-6 定价：69.00元

R语言市场研究分析

作者：克里斯·查普曼 等 ISBN：978-7-111-54990-1 定价：89.00元

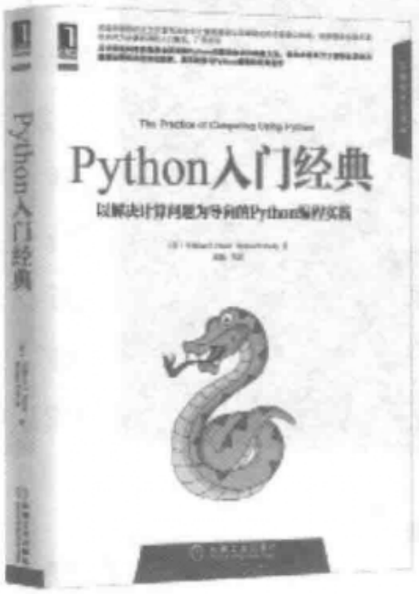
高级R语言编程指南

作者：哈德利·威克汉姆 ISBN：978-7-111-54067-0 定价：79.00元

推荐阅读

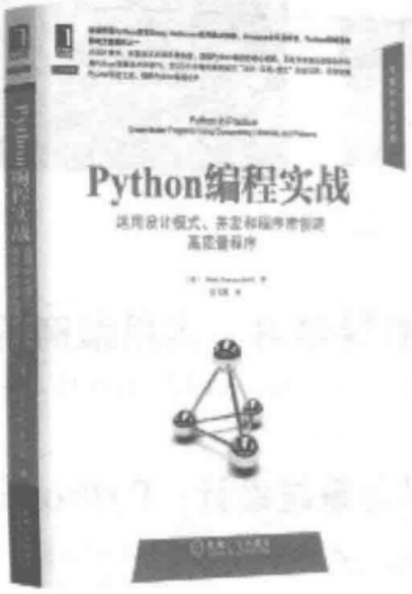
Python学习手册（原书第4版）

作者：Mark Lutz ISBN：978-7-111-32653-3 定价：119.00元



Python入门经典：以解决计算问题为导向的Python编程实践

作者：William F. Punch ISBN：978-7-111-39413-6 定价：79.00元



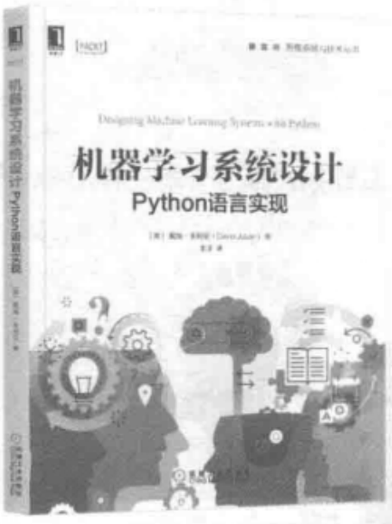
Python编程实战：运用设计模式、并发和程序库创建高质量程序

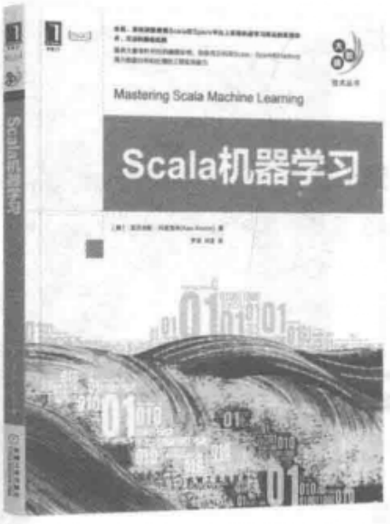
作者：Mark Summerfield ISBN：978-7-111-47394-7 定价：69.00元

Effective Python：编写高质量Python代码的59个有效方法

作者：Brett Slatkin ISBN：978-7-111-52355-0 定价：59.00元

推荐阅读





Python机器学习

作者：Sebastian Raschka，Vahid Mirjalili ISBN：978-7-111-55880-4 定价：79.00元

机器学习：实用案例解析

作者：Drew Conway，John Myles White ISBN：978-7-111-41731-6 定价：69.00元

面向机器学习的自然语言标注

作者：James Pustejovsky，Amber Stubbs ISBN：978-7-111-55515-5 定价：79.00元

机器学习系统设计：Python语言实现

作者：David Julian ISBN：978-7-111-56945-9 定价：59.00元

Scala机器学习

作者：Alexander Kozlov ISBN：978-7-111-57215-2 定价：59.00元

R语言机器学习：实用案例分析

作者：Dipanjan Sarkar，Raghav Bali ISBN：978-7-111-56590-1 定价：59.00元

Python文本分析

TEXT ANALYTICS WITH PYTHON

A PRACTICAL REAL-WORLD APPROACH TO GAINING ACTIONABLE INSIGHTS FROM YOUR DATA

使用Python从数据中发掘有用的洞见，了解自然语言处理和文本分析相关的技术，理解和获得解决特定问题的最佳技能。

本书全方位讲解文本、语言语法、结构和语义等基本概念和高级概念，并详细阐释文本分类、聚类、主题建模和文本摘要等算法和技术。

本书遵循结构化和综合性的讲解方式，即使缺少相关经验，你也不会感到茫然无措。从自然语言和Python的基础知识开始，进而介绍先进的分析理念和机器学习概念。你将了解每种技术和算法的概况，理解它们如何使用，同时从微观视角理解相关数学概念并应用它们来解决现实问题。

本书特色：

- 全面介绍自然语言处理（NLP）和文本分析的主要概念与技术。
- 包含丰富的真实案例实现技术，例如，构建分类新闻文章的文本分类系统，使用主题建模和文本摘要分析app或游戏评论，进行热门电影概要的聚类分析和电影评论的情感分析。
- 介绍基于Python及一些流行NLP和文本分析开源库（如自然语言工具包（nltk）、gensim、scikit-learn、spaCy和Pattern）的实现。

Apress®

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

上架指导：计算机/数据分析
ISBN 978-7-111-59324-9
9 787111 593249 >
定价：79.00元