第一章　机器学习的基础

“基础决定高度，而不是高度决定基础。”

机器学习脱胎于人工智能，自诞生开始，就充满了挑战，这个领域从来都吸引了全世界最顶尖的科学家，本书所讲述的正是他们多年的研究成果。对于普通读者而言，快速而全面的掌握各类算法技术并不是一件容易的事情。机器学习应用的领域方方面面：从交通运输、医疗诊断到自然语言处理等几乎各行各业都有。为了简化，多数书籍都偏重于数学理论，但即便完全形式化为数学概念，不仅不便于理解，所涉及到的知识也很庞杂。毕竟几十年来，机器学习这座大厦是靠一砖一瓦坚实的构筑起来的。

本章是全书的第一章，主要从程序编码、数据结构、数学理论、数据处理与可视化等几个方面阐述了机器学习相关的理论和技术实现。初看涉及内容比较多，为了避免混乱，我们以矩阵为中心贯穿本章各部分的知识讲解，然后扩展到概率论，数值分析，矩阵分析等知识来逐步引导读者进入机器学习的数学世界。

对大多数读者而言，理解数学原理和推导过程起初会存在一些障碍，这也是初学者们的一大壁垒，很多人虽有兴趣，但每到此处只能望而却步。因此，在数学理论方面，我们并没有罗列大量的公式和晦涩的术语，而是力求结合人们的日常生活，通过深入浅出的案例，使读者由浅入深、循序渐进地接触概念，最终真正领悟内涵。对象与维度、初识矩阵、理解随机性等章节完全不需要高等数学的基础，但是它们所阐述的方法和概念是整个机器学习大厦的基础。

而且幸运的是，由于软件编程方法的日新月异，矢量化编程方式（在第二节将详细介绍）能够将数学公式直观的转换为程序代码，这极大降低了程序设计的难度，多数公式的程序代码仅有 1~2 行。读者可以从 Numpy 矩阵运算和 Linalg 线性代数库两节逐步熟悉矢量化编程的风格，各类距离公式是矢量化编程的应用。除程序设计的之外，这两节也可看作是对线性代数中一些重要概念的回顾，如果读者对线性代数的概念生疏了，可以借此重温一下。总体而言，机器学习对程序设计的要求不高，除去矢量化编程，一般而言都是一些最基本的指令，只要掌握一定的编程技术和高等数学的基本概念，学好本书还是不难的。

各类距离公式、矩阵空间变换对于某些读者而言可能属于新知识。本章涉及的距离公式比较多，但总的来讲都不难，本书也都提供了程序代码。矩阵的空间变换是个重点，后续章节中多有涉及，需要读者认真领会。如果理解上有困难，具体算法上我们还会详细讲解。

总之，我们的目标是使程序设计变为一件轻松、快乐的事情。常言道：“千里之行，始于足下”。现在，就让我们开始激动人心的机器学习之旅吧！

1.1 编程语言与开发环境

1.1.1 搭建 python 开发环境

“工欲善其事，必先利其器”，还好 Python 语言还算锋利，一般具有程序设计基础的读者，几周就可掌握本书中所需的语言技术。因此，书中绝大多数程序代码都使用 Python 语言编写，原因有以下几点：

- 免费、开源：Python 语言是免费开源的脚本语言。这两个词几乎成为流行编程语言必不可少的特征
- Python 编程更简单，相比于编译语言(C,C++)而言，Python 是一种跨平台脚本语言，编写好的代码可以直接部署在各类操作系统上(例如，Linux、Windows、MAC OS X)；
- 开发和执行效率高，其各种库大多数都是基于 C 语言编写的(相对于 Java 而言)，并适用于 32 位和 64 位系统，性能损失小，适合大规模数据处理；
- 丰富的程序库，支持矢量编程，Python 在机器学习和自然语言处理方面提供了完备程序库，包括：机器学习、数学分析、可视化库、GPU 并行库等等；
- Python 支持网络编程，写好的代码可以直接发布到 Internet 上。

Python 开发环境可以搭建在 Linux 下，也可以搭建在 Windows 下，可以是 32 位的，也可以是 64 位的。这为开发者提供了很大的灵活性。为了便于初学者学习，本书在 Windows7 下部署 64 位的 Python 开发环境，同时在附录中也提供 Linux 下的部署方式。

Python 可在官方网站直接下载，网址：https://www.python.org/downloads/source/。本书使用的是 2.7-64 位版本，下载地址:https://www.python.org/downloads/release/python-279/，所装的库和代码也是以 2.7 版本为基础的，如果读者使用其他版本，需要做相应的修改。

下面给出在 Windows 下简要的安装步骤:

1．双击下载的安装程序：python-2.7.9.amd64.msi，执行安装（如图 1.1），如果其他用户不需要 python 的话，可以使用第二个，不过我们一般都是单用户，所以没差别。

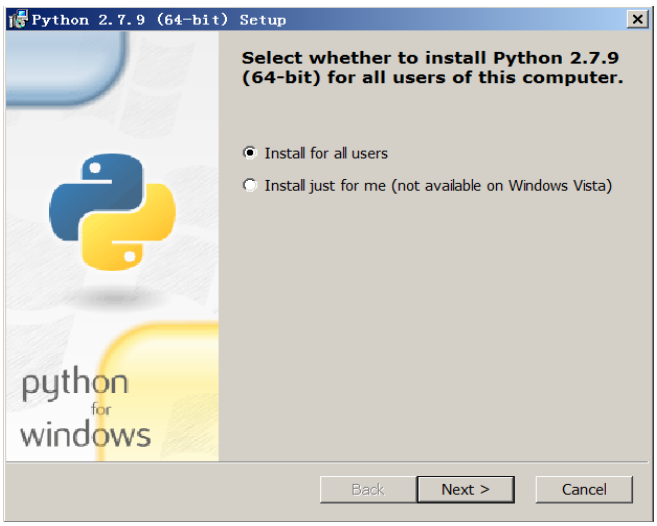


图 1.1　python 安装步骤 1

2．选择安装路径，可按默认路径安装，也可自己新建路径，新建路径一般用英文名比较好，方便命令行访问，本书使用的是 C:\python64 点击 Next（如图 1.2）。

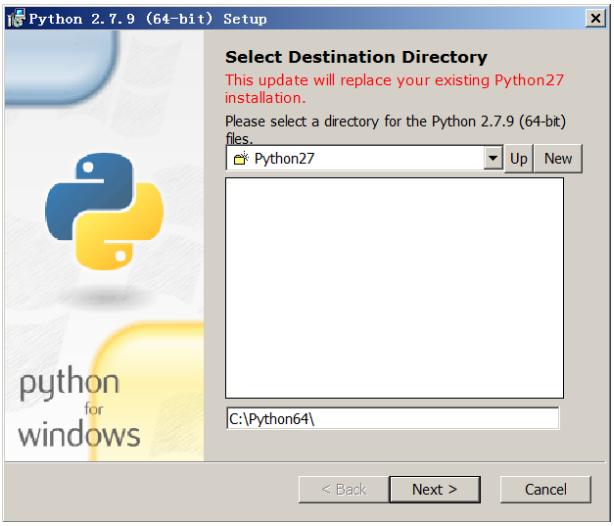


图 1.2　python 安装步骤 2

3．其他界面都一路 Next，程序开始复制安装文件，复制完文件后点击 finish，完成安装（如图 1.3）。

图 1.3　python 安装步骤 3

该界面出现，指示安装完成。

1.1.2　安装 python 算法库

最初，Python 社区仿照 Matlab 开发了类似的数学分析库，主要包括 NumPy 和 SciPy 来处理数据，Matplotlib 实现数据可视化。大多数 Python 数学和算法领域的应用都广泛地将其作为基本的程序库。很快，为了适应处理大规模数据的需求，Python 在此基础上开发了 Scikit-Learn 机器学习算法库(网址: http://scikit-learn.org/stable/)；同时，还提供了深度学习算法库 Theano(网址: http://deeplearning.net/software/theano/)，并支持 GPU 运算。迄今为止，Python 已经可以完整地提供基于 C/C++的全部机器学习开发包，而且都是开源版的，有兴趣的朋友还可以下载源码包学习。后面的章节将详细讲解这些库的使用，这里就不再赘述了。

除此之外，Python 也提供了大量的常用程序库，例如数据库 API(MysqlDB)、GUI 图形界面库(WxPython)、高并发协程库(gevent)、中文分词库(jieba)等外部库。所有这些库可以从下面两个网址查询到：

- 官方下载地址：https://pypi.python.org/pypi
- 非官方下载地址：http://www.lfd.uci.edu/~gohlke/pythonlibs/

最值得称道的是，这些算法库不仅安装非常简便，而且均提供 32 位和 64 位两个版本，并分别支持 2.6，2.7，3.3，3.4 不同版本的 Python 应用。

安装顺序与步骤：

- Python 算法库的安装非常简单:
  执行 C:\Python64\Scripts\pip install **库名**(小写字母不加后缀)
- Python 算法库的安装顺序:
  1. Numpy
  2. Scipy
  3. Matplotlib
  4. Scikit-Learn

上述每个库安装完成后，都会提示如下信息：

```
Successfully installed xxx(库名)

Cleaning up..
```

表示此库安装成功。

1.1.3 IDE 配置及其安装测试

Python 的各类算法库安装完成之后，一般需要选个开发平台(IDE)编写代码。本书的核心内容偏重算法设计，对 IDE 要求不高，推荐使用 Ultraedit 高级文本编辑器。主要考虑资源占用较小，功能较强，支持多种文件编码转换，并有支持远程开发等优势（本地可同步远程 Linux 上的源代码，本地编写，远程上运行）。网上可以下载安装绿色版的软件。下载完成之后，需要进行如下配置：

1. 在点击“高级”选项卡，点开“工具栏配置”（如图 1.4）：

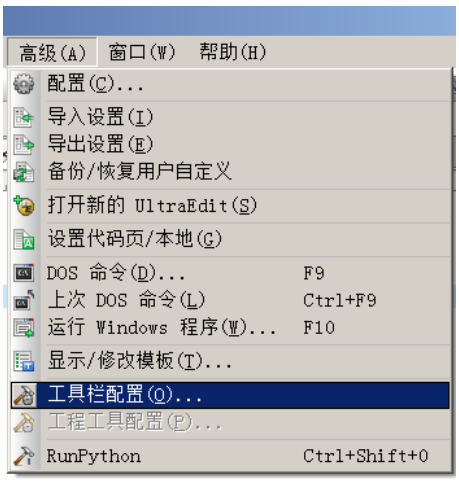


图 1.4　配置 Ultraedit 步骤 1

2. 按下图界面输入,并“应用”（如图 1.5）：

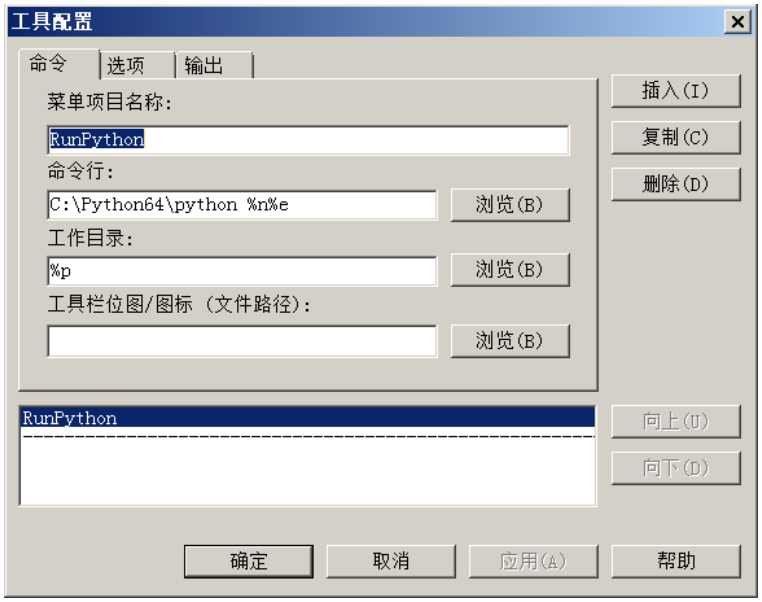


图 1.5　配置 Ultraedit 步骤 2

3. 按下图界面输入，并点击“确定”（如图 1.6）：

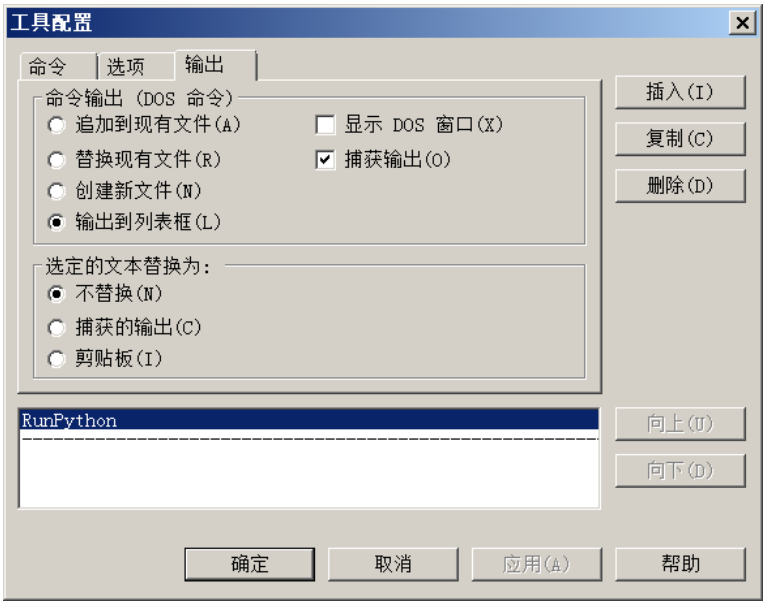


图 1.6　配置 Ultraedit 步骤 3

这样 Ultraedit 就算配置完成了。下面我们开始编写一些测试代码。因为安装的模

块比较多，简单的 Hello World 测试不能说明问题，过于复杂的程序又容易使人费解。为此，我们简单编写一个应用程序用于测试，运行下面的测试代码，可以检验安装的效果。

代码文件 mytest1.py：

```
# -*- coding: utf-8 -*-
# Filename : mytest1.py

import numpy as np                # 导入 numpy 库
from numpy import *               # 导入 numpy 库
import matplotlib.pyplot as plt   # 导入 matplotlib 库

# 测试数据集-二维 list
dataSet = [[-0.017612,14.053064],[-1.395634  ,4.662541],[-0.752157
      ,6.538620],[-1.322371     ,7.152853],[0.423363      ,11.054677],[0.406704
      ,7.067335],[0.667394      ,12.741452], [-2.460150   ,6.866805],[0.569411
      ,9.548755],[-0.026632     ,10.427743],[0.850433     ,6.920334],[1.347183
      ,13.175500],[1.176813     ,3.167020],[-1.781871     ,9.097953]]

dataMat = mat(dataSet).T                  # 将数据集转换为 numpy 矩阵，并转置
plt.scatter(dataMat[0],dataMat[1],c='red',marker='o')      # 绘制数据集散点图

# 绘制直线图形
X = np.linspace(-2,2,100) # 产生直线数据集
# 建立线性方程
Y = 2.8*X+9

plt.plot(X,Y)   # 绘制直线图
plt.show()      # 显示绘制后的结果
```

输出结果（如图 1.7）：

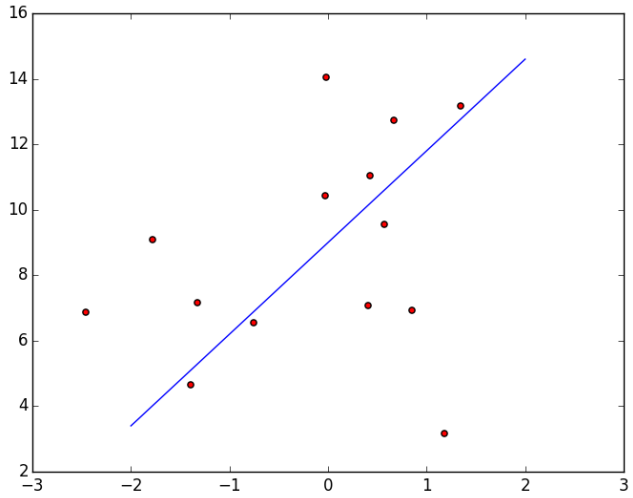


图 1.7　显示执行结果

说明主要模块安装成功。

1.2　对象、矩阵与矢量化编程

有了工具，很多事情就变得方便了。现在，我们正式进入机器学习的基础知识。简单回忆一下绪论部分所提出的三种对象类型：文本、表格、图。乍一看，有点眼花缭乱，仔细分析下来，三种结构虽各有千秋，却存在着共性。

1.2.1　对象与维度

对于大多数程序员而言，对象应该不是个陌生的概念。在面向对象的程序设计思想中，对象就是一个类的实例。机器学习中的对象与之很相似，在机器学习中，对象是指含有一组特征的行向量。而行向量的集合最容易构造的结构就是表。下面我们来观察一张表（如表 1.1），此表来源于现实中真实的统计数据：

表 1.1　大型动物与水果

实例	种属	重量(平均)	颜色(主)	生命周期/保质期
非洲象	动物	5吨	土灰色	70年
大白鲨	动物	3.2吨	灰白色	70年

苹果	植物	250克	红色	5~15天
梨	植物	300克	黄色	5~10天

表中第一行黑体字：种属、重量(平均)、颜色(主)、生命周期/保质期表示为特征名称。所有特征组合在一起构成一组行向量，也称为特征向量，我们为了区别线性代数中的特征值和特征向量引入对象这个名称。以非洲象、大白鲨等开头的数据行就是一组行向量，也是一个对象。对象的维度就是行向量的列数，上述数据集的维度为 5。

在实际计算中，除非特殊情况，特征名称不需列明；含有字符串的对象名称因无法直接参与运算，一般情况下可以编码为数字，我们将种属特征转换为是否动物（用布尔值 0,1 替代），颜色特征转换为十六进制。各列的特征值为了计算方便，应统一单位，区间值可以选择中间值。为了方便量化，表 1.1 删除第一列，大型动物与水果表就转换为（如表 1.2）：

表 1.2　大型动物与水果表

1	5000	#DCDCDC	70*365
1	3200	#9D9D9D	70*365
0	0.25	#FF0000	10
0	0.3	#FFFF00	7

此时，该表被转换为 4 行、4 列的数据表，维度为 4。有过一些线性代数知识的朋友可以很简单的将此表转换为下面的矩阵（如表 1.3）：

表 1.3　大型动物与水果矩阵

$$\begin{pmatrix} 1 & 5000 & \#DCDCDC & 70*365 \\ 1 & 3200 & \#9D9D9D & 70*365 \\ 0 & 0.25 & \#FF0000 & 10 \\ 0 & 0.3 & \#FFFF00 & 7 \end{pmatrix}$$

一般而言一个对象应被视作完整的个体，代表现实中有意义的事物，不能轻易拆分。它的表现形式可能多种多样，例如在图像识别中它可能是一张图片（如图 1.8）：

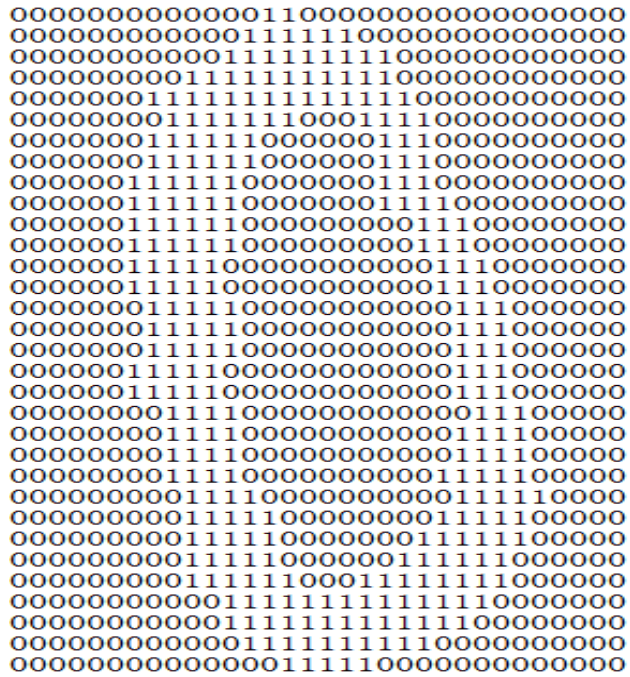


图 1.8　手写体识别中的“0”

上图是向量化的手写体图片。注意，上图的维度，即行向量的个数是 32*32 个，整张图被当作一行或者一个对象。因为，整张图仅代表一个完整的事物，表达了一个完整的意义。一般来说，图片数据集的维度都比较高。

文本类数据集有点特殊，需要首先生成词袋列表，再根据每个词出现的词频数值化，不过仔细想想也万变不离其宗。例如，如下文本数据集:

文本 1：My dog ate my homework。

文本 2：My cat ate the sandwich。

文本 3：A dolphin ate the homework。

根据上述三段短文本中出现的词汇，生成词袋列表，该列表记录了上述文本中出现的每个不重复的词：

[a, ate, cat, dolphin, dog, homework, my, sandwich, the]

各段文本根据生成的词袋列表转换为对象，其维度就是词袋列表的长度，计算结果是 32 维。向量化后的文本称为词向量。

各段文本向量化后的词向量，其中 0 表示词袋中对应的词未出现在文本中，1 表示该词出现在文本中 1 次，如果多次出现，则累加（如表 1.4）：

表 1.4　词向量空间

词袋 文本	a, ate, cat, dolphin, dog, homework, my, sandwich, the
文本1	0,1,0,0,1,1,2,0,0
文本2	0,1,1,0,0,0,1,1,1
文本3	1,1,0,1,0,1,0,0,1

1.2.2 初识矩阵

在线性代数中我们学过，矩阵是由 m×n 个数组成的一个 m 行 n 列的矩形表格，或者更深一点的定义，表示由方程组的系数及常数所构成的二维数据表。从机器学习的角度来看，这两个定义都合适，又都不合适。

现在，我们重新考察一下矩阵。上一节，我们有了表和对象的概念。对象是被特征化的客观事物，而表是容纳这些对象的容器。换句话说，对象是表中的元素，表是对象的集合。但这个集合有点特殊，即表中的每个对象都有相同的特征和维度，对象对于每个特征都有一定的取值。这里矩阵可以理解为，具有相同特征和维度的对象集合。

总结一下：

- 矩阵是具有相同特征和维度的对象集合，表现为一张二维数据表；
- 一个对象表示为矩阵中的一行，一个特征表示为矩阵中的一列，每个特征都有数值型的取值；
- 特征相同、取值相异的对象集合所构成的矩阵，使对象之间既相互独立，又相互联系；
- 由特征列的取值范围所有构成的矩阵空间，应具有完整性，即能够反映出事物的空间形式或变化。（再谈矩阵一节中详细说明）

关于第三点，我们再观察表 1.4：动物种属和植物种属是两个风马牛不相及的概念，但如果放到一张表中，我们必须使用一组特征向量来衡量它们，比如：种属、重量、颜色、生命周期(年龄有点牵强)等等。但不能用味道这个特征来衡量，因为大象和鲨鱼很难讲是酸是甜。这就是各类对象之间的联系。

如果不看实例名称，我们从是否动物、重量、颜色、生命周期几个特征也能很容易的分辨出两类对象的差异性。例如，大象重量和苹果重量的悬殊差异，或生命周期/保质期的悬殊差异。这反映了对象间的相互独立。

我们再扩展一下这个例子（如表 1.5）：

表 1.5　扩展表 1.2

实例	是否动物	重量	颜色	生命周期/保质期
大象	1	5000	#DCDCDC	70*365

鲨鱼	1	3200	#9D9D9D	70*365
苹果	0	0.25	#FF0000	10
梨	0	0.3	#FFFF00	7

除差异性之外，表 1.5 中还有一类情况，苹果和梨、大象和鲨鱼。在是否动物、重量和生命周期三个特征中，苹果和梨具有相似性，大象和鲨鱼具有相似性。很明显，大象与鲨鱼的重量较之苹果(或梨)的重量更相似；或者大象与鲨鱼的寿命较之苹果(或梨)的保质期更相似；由于这种相似性，我们很自然地可以将苹果和梨分为一类：水果（区别于植物）；大象和鲨鱼分为一类：大型动物（区别于动物）。

由此可见，分类或聚类可以看作是根据对象特征的相似性与差异性，对矩阵空间的一种划分。

下面，我们再看另一个例子(如表 1.6)：

表 1.6　2-18 岁正常男生体重身高对照表

年龄(岁)	身高(cm)	体重(kg)
2	88.5	12.54
3	96.8	14.65
4	104.1	16.64
5	111.3	18.98
6	117.7	21.26
7	124.0	24.06
8	130.0	27.33
9	135.4	30.46
10	140.2	33.74
11	145.3	37.69
12	151.9	42.49
13	159.5	48.08
14	165.9	53.37
15	169.8	57.08
16	171.6	59.35
17	172.3	60.68
18	172.7	61.40

这个表列举了 2~18 岁男生的正常体重、身高的变化。数值上呈现一种递增的趋势，每个对象都与上一行或下一行的对象在时间上相关，并且时间间隔相等，都为 1 年。这种时间上的相关性使矩阵反映出某一事物在时间上连续变化的过程。

由此可见，预测或回归可以看作根据对象在某种序列(时间)上的相关性，表现为特征取值变化的一种趋势。

分类、聚类和回归是机器学习最基本的主题。通过矩阵，可以构建客观事物的多维度数学模型，并通过条件概率分布、梯度、神经网络、协方差等等运算方式，多角度认识和分析事物。

具体来讲，矩阵有三个重要用途，第一是解线性方程组，比如二维矩阵可以理解为一个平面直角坐标系内的点集，通过计算点与点之间的距离，完成聚类、分类或预测，类似的运算完全可以扩展到多维的情况。第二个用途是方程降次，也就是利用矩阵的二次型，通过升维将线性不可分的数据集映射到高维中，转换为线性可分的情形，这是支持向量机的基本原理之一。第三个用途是变换，矩阵可以通过特征值和特征向量，完成维度约简，简化类似图片这种高维数据集的运算，主成分分析使用的就是这个原理。

在程序设计中，我们可以从形式上把矩阵理解为一个二维数组。以 Python 语言为例，矩阵就是嵌套着若干个 list 的一个大 list。内部的每个 list 都是等长的，其中每个元素都是整型或浮点型的数值。内部的 list 就是行向量，即一个对象。

Numpy 数据包提供了专门的矩阵数据结构和线性代数库。从下一节开始，我们将详细介绍 Numpy 的矩阵运算。

1.2.3 矢量化编程与 GPU 运算

传统的计算机设计语言，例如 C 语言，是针对标量的，虽然也提供诸如数组、二维数组的数据结构，但这些结构的赋值、加法、数乘、转置、矩阵相乘还是要通过循环语句完成，十分麻烦，导致程序设计的复杂度也比较高。

```
mylist = [1,2,3,4,5]
length = len(mylist)
a = 10
for indx in xrange(length):
    mylist[indx] = a*mylist[indx]
print mylist
```

输出结果：

```
[10, 20, 30, 40, 50]
```

而基于矩阵的算法都是针对向量的，这里也称为矢量。为了简化程序的逻辑，就需要一种新的编程方法，处理基于矩阵的基本运算，这就是所谓的矢量化编程。

随着程序设计的发展，使用计算机实现矩阵运算越来越方便。最早实现矢量化编程的语言是 Matlab 的脚本语言，它极大的降低了数学领域程序设计的复杂度。因此大量的人工智能算法最早都是用 Matlab 语言写的。

Python 自带的 List 结构，提供切片的功能可以部分实现矢量化编程。其扩展包 Numpy 提供了专门的矩阵数据结构和线性代数库，完全实现了矢量化编程。

```
import numpy as np

mylist = [1,2,3,4,5]
```

```
a = 10
mymatrix = np.mat(mylist)
print a*mymatrix
```

输出结果：

```
[[10 20 30 40 50]]
```

矢量化编程的一个重要特点就是可以直接将数学公式转换为相应的程序代码，极大方便了程序的阅读和调试，使复杂数学公式的实现变得简单和方便，本节和下一节所用的矢量化程序的代码一般只有一两行，即可完成复杂的数学运算。

无论是 Matlab 还是 Python 的矢量化编程都可以无缝的调用底层的 C 函数，还可以提高算法速度。为了提升特定数值运算操作，例如矩阵相乘、矩阵相加、矩阵-向量乘法、浮点运算的速度，数值计算和并行计算的研究人员已经努力了几十年。这个领域最出色的技术就是使用图形处理器的 GPU 运算。

英伟达（nVidia）公司在 1999 年发布 GeForce256 图形处理芯片时首先提出了 GPU 运算的概念。十几年的发展使单个 GPU 芯片在浮点运算、大规模并行计算方面，可以提供数十倍乃至上百倍于 CPU 的性能。GPU 的流处理器也由几十个增加到最新的三千多个，浮点运算 TFlops 值也达到 5 以上。本书的第十章的深度学习部分专门讲解了 GPU 运算的 Python 框架 Theano。

对 GPU 运算比较熟悉的读者也可以将本书中较大规模的矩阵运算交由 GPU 完成。

1.2.4 理解数学公式与 Numpy 矩阵运算

为了便于理解后续章节算法部分的讲解，本节将常用的矩阵数学公式和程序代码对应出来，读者可根据自己需求有选择的学习。因使用矢量编程的方法，矩阵的基本运算得到了较大的简化。

1.矩阵的初始化：

```
import numpy as np  # 导入 numpy 包
```

1.1 创建一个 3*5 的全零矩阵和全 1 矩阵

```
myZero = np.zeros([3,5]) # 3*5 的全零矩阵
print myZero
myOnes = np.ones([3,5]) # 3*5 的全 1 矩阵
print myOnes
```

输出结果：

```
[[ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]]
 [[ 1.  1.  1.  1.  1.]
```

```
 [ 1.   1.   1.   1.   1.]
 [ 1.   1.   1.   1.   1.]]
```

1.2 生成随机矩阵：

```
myRand = np.random.rand(3,4) # 3 行 4 列的 0~1 之间的随机数矩阵
print myRand
```

输出结果：

```
[[ 0.79473503   0.4682515    0.53933591   0.94568244]
 [ 0.52199975   0.81190881   0.41920671   0.16756969]
 [ 0.57211218   0.53727222   0.83488426   0.30227915]]
```

1.3 单位阵：

```
myEye = np.eye(3) # 3*3 的单位阵
print myEye
```

输出结果：

```
[[ 1.   0.   0.]
 [ 0.   1.   0.]
 [ 0.   0.   1.]]
```

2.矩阵的元素运算：是指矩阵在元素级别的加、减、乘、除运算

```
from numpy import*   #导入 numpy 包
```

2.1 元素相加和相减：条件，矩阵的行数和列数必须相同

数学公式：$(\mathbf{A} \pm \mathbf{B})_{i,j} = \mathbf{A}_{i,j} \pm \mathbf{B}_{i,j}$，

```
myOnes = ones([3,3])   # 3*3 的全 1 矩阵
myEye = eye(3)         # 3*3 的单位阵
print myOnes+myEye   #  矩阵相加
print myOnes-myEye    #  矩阵相减
```

输出结果：

```
[[ 2.   1.   1.]
 [ 1.   2.   1.]
 [ 1.   1.   2.]]
[[ 0.   1.   1.]
 [ 1.   0.   1.]
 [ 1.   1.   0.]]
```

2.2 矩阵数乘：一个数乘以一个矩阵

数学公式：$(c\mathbf{A})_{i,j} = c \cdot \mathbf{A}_{i,j}$.

```
mymatrix = mat( [[1,2,3],[4,5,6],[7,8,9]])
a = 10
print a*mymatrix
```

输出结果：

```
[[10 20 30]
 [40 50 60]
```

```
 [70 80 90]]
```

2.3 矩阵所有元素求和：

数学公式： $\mathrm{sum}(A)=\sum_{i=1}^{m}\sum_{j=1}^{n}A_{i,j}$ 其中1<i<m,1<j<n

```
mymatrix = mat([[1,2,3],[4,5,6],[7,8,9]] )
print sum(mymatrix)
```

输出结果：

```
45
```

2.4 矩阵各元素的积：矩阵的点乘同维对应元素的相乘。当矩阵的维度不相同时，会根据一定的广播规则将维数扩充到一致的形式，

数学公式： $(A.*B)_{ij} = A_{ij} * B_{ij}$

```
mymatrix = mat( [[1,2,3],[4,5,6],[7,8,9]])
mymatrix2 = 1.5*ones([3,3])
print multiply(mymatrix,mymatrix2)
```

输出结果：

```
[[  1.5   3.    4.5]
 [  6.    7.5   9. ]
 [ 10.5  12.   13.5]]
```

2.5 矩阵各元素的 n 次幂：n=2

数学公式： $A_{ij}^{2} = A_{ij} * A_{ij}$

```
mylist = mat( [[1,2,3],[4,5,6],[7,8,9]])
print power(mymatrix,2)
```

输出结果：

```
[[ 1  4  9]
 [16 25 36]
 [49 64 81]]
```

4. 矩阵的乘法：矩阵乘矩阵

数学公式： $[A,B]_{i,j}=A_{i,1}B_{1,j}+A_{i,2}B_{2,j}+\ldots+A_{i,n}B_{n,j}=\sum_{r=1}^{n}A_{i,r}B_{r,j}$

```
from numpy import *

mymatrix = mat([[1,2,3],[4,5,6],[7,8,9]])
mymatrix2 = mat([[1],[2],[3]])
print mymatrix*mymatrix2
```

输出结果：

```
[[14]
 [32]
 [50]]
```

5. 矩阵的转置：

数学公式：$(A^T)_{i,j} = A_{j,i}$.

```
from numpy import *

mymatrix = mat([[1,2,3],[4,5,6],[7,8,9]])
print mymatrix.T    # 矩阵的转置
mymatrix.transpose()  # 矩阵的转置
print mymatrix
```

输出结果：

```
[[1 4 7]
 [2 5 8]
 [3 6 9]]
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

5. 矩阵的其他操作：行列数、切片、复制、比较

```
from numpy import *

mymatrix = mat([[1,2,3],[4,5,6],[7,8,9]])
[m,n]=shape(mymatrix)    # 矩阵的行列数
print "矩阵的行数和列数:",m,n

myscl1 = mymatrix[0] #  按行切片
print "按行切片:",myscl1

myscl2 = mymatrix.T[0]   # 按列切片
print "按列切片:",myscl2

mycpmat = mymatrix.copy() #  矩阵的复制
print "复制矩阵:\n",mycpmat

#比较
print "矩阵元素的比较:\n",mymatrix < mymatrix.T
```

输出结果：

```
矩阵的行数和列数: 3 3
按行切片: [[1 2 3]]
按列切片: [[1 4 7]]
复制矩阵:
[[1 2 3]
 [4 5 6]
```

```
 [7 8 9]]
矩阵元素的比较:
[[False  True  True]
 [False False  True]
 [False False False]]
```

1.2.5 Linalg 线性代数库

在矩阵基本运算的基础之上，Numpy 的 linalg 库可以满足大多数的线性代数运算，本节所列的矩阵公式列表和代码如下：

- ❑ 矩阵的行列式
- ❑ 矩阵的逆
- ❑ 矩阵的对称
- ❑ 矩阵的秩
- ❑ 可逆矩阵求解线性方程组

读者可根据自己需求有选择的学习。因使用矢量编程的方法，矩阵的基本运算得到了较大的简化。

1.矩阵的行列式：

```
from numpy import *

# n 阶方阵的行列式运算
A = mat( [[1,2,4,5,7,],[9,12,11,8,2,],[6,4,3,2,1,],[9,1,3,4,5],[0,2,3,4,1]])
print "det(A):",linalg.det(A);   #  方阵的行列式
```

输出结果：

```
det(A): -812.0
```

2.矩阵的逆：

```
from numpy import *

A = mat( [[1,2,4,5,7,],[9,12,11,8,2,],[6,4,3,2,1,],[9,1,3,4,5],[0,2,3,4,1]])
invA = linalg.inv(A) #  矩阵的逆
print "inv(A):",invA
```

输出结果：

```
inv(A): [[ -7.14285714e-02   -1.23152709e-02    5.29556650e-02    9.60591133e-02
    -8.62068966e-03]
 [   2.14285714e-01   -3.76847291e-01    1.22044335e+00   -4.60591133e-01
     3.36206897e-01]
 [ -2.14285714e-01    8.25123153e-01   -2.04802956e+00    5.64039409e-01
    -9.22413793e-01]
```

```
[   1.66901077e-16   -4.13793103e-01    8.79310345e-01   -1.72413793e-01
     8.10344828e-01]
[   2.14285714e-01   -6.65024631e-02    1.85960591e-01   -8.12807882e-02
    -1.46551724e-01]]
```

3.矩阵的对称：

```
from numpy import *

A = mat([[1,2,4,5,7,],[9,12,11,8,2,],[6,4,3,2,1,],[9,1,3,4,5],[0,2,3,4,1]])
AT = A.T     #矩阵的对称
print A*AT
```

输出结果：

```
[[ 95 131   43   78   43]
 [131 414 153 168   91]
 [ 43 153   66   80   26]
 [ 78 168   80 132   32]
 [ 43   91   26   32   30]]
```

4.矩阵的秩：

```
from numpy import *

A = mat([[1,2,4,5,7,],[9,12,11,8,2,],[6,4,3,2,1,],[9,1,3,4,5],[0,2,3,4,1]])
print linalg.matrix_rank(A)   #矩阵的秩
```

输出结果：

```
5
```

5.可逆矩阵求解：

```
from numpy import *

A = mat([[1,2,4,5,7,],[9,12,11,8,2,],[6,4,3,2,1,],[9,1,3,4,5],[0,2,3,4,1]])
b = [1,0,1,0,1]
S = linalg.solve(A,T(b))
print S
```

输出结果：

```
[-0.0270936    1.77093596 -3.18472906   1.68965517   0.25369458]
```

1.3　机器学习的数学基础

上一节，我们建立了矩阵的概念，并介绍了矩阵基本运算的 Python 实现。从本节开始，我们学习有关机器学习的一些最基本的数学概念。一谈到数学，大多数人的感觉就一个字：晕。其实，数学让人觉得难的地方不外乎两点：一是语言符号非常简练，

二是理论描述比较抽象。长久以来数学研究的是客观世界的空间形式与数量性质，即事物在时空中的普遍存在与运动的规律。因为反映本质，所以才精炼，因为应用普遍，所有才抽象。

现代数学有三个重要的基石：概率论、数值分析、线性代数。概率论说明了事物可能会是什么样；数值分析揭示了它们为什么这样，以及如何变成这样；线性代数则告诉我们事物从来不只有一个样子，使我们能从多个角度来观察事物。

1.3.1 相似性的度量

继续以第一节的表 1.5 为例，我们做出如下的精简（如表 1.7）：

表 1.7 精简的大型动物和水果表

实例	重量	生命周期/保质期
大象	5000	70*365
鲨鱼	3200	70*365
苹果	0.25	15
梨	0.3	10

其中每个对象都是一个特征向量，从直觉上，我们可将其分为两大类：

- 大型动物:大象、鲨鱼
- 水果:苹果、梨

因为大象和鲨鱼都很大，生命周期也都很长，相比之下苹果和梨要小得多，保质期也都很短。很大、很长对很小、很短是在量上的比较，因此，利用初等数学的知识，这些给定数值的对象就可以看作一个 n 维坐标系下的点，并通过点与点之间的距离来度量。

两个向量之间的距离(此时向量作为 n 维坐标系中的点)计算，在数学上称为向量的距离(Distance)，也称为样本之间的相似性度量(Similarity Measurement)。 它反映为某类事物在距离上接近或远离的程度，直觉上，距离越近的就越相似，越容易归为一类，距离越远就越不同，但这个直觉的标准是什么呢？换句话说，这么划分的依据是什么呢？由此，我们引出向量间的各类距离公式，下面这些距离公式从不同角度对向量间的距离定义了衡量标准。

在引入距离公式之前，我们先看一个概念:

范数(来自百度百科): 向量的范数可以简单形象的理解为向量的长度，或者向量到坐标系原点的距离，或者相应空间内的两个点之间的距离。

向量的范数定义：向量的范数是一个函数||x||,满足非负性||x|| >= 0，齐次性||cx|| = |c| ||x|| ，三角不等式||x+y|| <= ||x|| + ||y||

L1 范数: ||x||为 x 向量各个元素绝对值之和。

L2 范数: ||x||为 x 向量各个元素平方和的开方，L2 范数又称 Euclidean 范数或者 Frobenius 范数

Lp 范数: ||x||为 x 向量各个元素绝对值 p 次方和的 1/p 次方

L∞范数: ||x||为 x 向量各个元素绝对值最大那个元素，如下：

$$\lim_{k\to\infty}(\sum_{i=1}^{n}|p_i-q_i|^k)^{1/k}$$

向量范数的运算：

```
A = [8,1,6]
# 手工计算
modA = sqrt(sum(power(A,2)))
print "modA:",modA
# 库函数
normA = linalg.norm(A)
print "norm(A):",normA
```

结果：

```
modA: 10.0498756211
norm(A): 10.0498756211
```

1.3.2 各类距离的意义与 Python 实现

本节所列的距离公式列表和代码如下：

- 闵可夫斯基距离(Minkowski Distance)
- 欧氏距离(Euclidean Distance)
- 曼哈顿距离(Manhattan Distance)
- 切比雪夫距离(Chebyshev Distance)
- 夹角余弦(Cosine)
- 汉明距离(Hamming distance)
- 杰卡德相似系数(Jaccard similarity coefficient)

读者可根据自己需求有选择的学习。因使用矢量编程的方法，距离计算得到了较大的简化。

1. 闵可夫斯基距离(Minkowski Distance)

严格意义上，闵氏距离不是一种距离，而是一组距离的定义。

(1)闵氏距离的定义:

两个 n 维变量 $A(x_{11},x_{12},\ldots,x_{1n})$与 $B(x_{21},x_{22},\ldots,x_{2n})$间的闵可夫斯基距离定义为：

$$d_{12}=\sqrt[p]{\sum_{k=1}^{n}(x_{1k}-x_{2k})^{p}}$$

其中 p 是一个变参数。

当 p=1 时，就是曼哈顿距离

当 p=2 时，就是欧氏距离

当 p→∞时，就是切比雪夫距离

根据变参数的不同，闵氏距离可以表示一类的距离。

2.欧氏距离(Euclidean Distance)

欧氏距离（L2 范数）是最易于理解的一种距离计算方法，源自欧氏空间中两点间的距离公式（如图 1.9）。

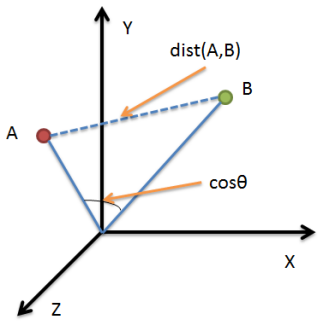


图 1.9　AB 间的欧式距离 dist(A,B)

(1)二维平面上两点 $a(x_1,y_1)$与 $b(x_2,y_2)$间的欧氏距离：

$$d_{12}=\sqrt{(x_1-y_1)^2+(x_2-y_2)^2}$$

(2)三维空间两点 $A(x_1,y_1,z_1)$与 $B(x_2,y_2,z_2)$间的欧氏距离：

$$d_{12}=\sqrt{(x_1-x_2)^2+(y_1-y_2)^2+(z_1-z_2)^2}$$

(3)两个 n 维向量 $A(x_{11},x_{12},\ldots,x_{1n})$与 $B(x_{21},x_{22},\ldots,x_{2n})$间的欧氏距离：

$$d_{12}=\sqrt{\sum_{k=1}^{n}(x_{1k}-x_{2k})^2}$$

表示成向量运算的形式：

$$d_{12}=\sqrt{(A-B)(A-B)^T}$$

(4) python 实现欧式距离公式的：

```
from numpy import *

vector1 = mat([1,2,3])
vector2 = mat([4,5,6])
print sqrt((vector1-vector2)*((vector1-vector2).T))
```

输出：

```
[[ 5.19615242]]
```

3.曼哈顿距离(Manhattan Distance)

从名字就可以猜出这种距离的计算方法了。想象你在曼哈顿要从一个十字路口开车到另外一个十字路口，驾驶距离是两点间的直线距离吗？显然不是，除非你能穿越大楼。实际驾驶距离就是这个“曼哈顿距离”(L1 范数)。而这也是曼哈顿距离名称的来源，曼哈顿距离也称为城市街区距离(City Block distance)（如图 1.10）。

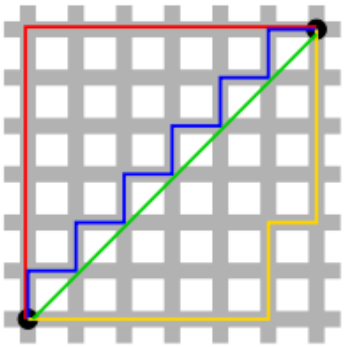


图 1.10　AB 间的曼哈顿距离为红色、蓝色、黄色线条

(1)二维平面两点 $A(x_1,y_1)$与 $B(x_2,y_2)$间的曼哈顿距离

$$d_{12} = |x_1 - x_2| + |y_1 - y_1|$$

(2)两个 n 维向量 $A(x_{11},x_{12},\ldots,x_{1n})$与 $B(x_{21},x_{22},\ldots,x_{2n})$间的曼哈顿距离

$$d_{12} = \sum_{k=1}^{n} |x_{1k} - x_{2k}|$$

(3)python 实现曼哈顿距离：

```
from numpy import *

vector1 = mat([1,2,3])
vector2 = mat([4,5,6])
print sum(abs(vector1-vector2))
```

输出：

```
9
```

4.切比雪夫距离(Chebyshev Distance)

国际象棋玩过么？国王走一步能够移动到相邻的 8 个方格中的任意一个（如图 1.11）。那么国王从格子(x_1,y_1)走到格子(x_2,y_2)最少需要多少步？自己走走试试。你会发现最少步数总是 max(| x_2-x_1| , |y_2-y_1|) 步。有一种类似的一种距离度量方法叫切比雪夫距离(L∞范数)。

图 1.11　国际象棋与切比雪夫距离

(1)二维平面两点 A(x_1,y_1)与 B(x_2,y_2)间的切比雪夫距离：

$$d_{12} = \max(|x_1 - x_2|, |y_1 - y_2|)$$

(2)两个 n 维向量 A (x_{11},x_{12},…,x_{1n})与 B(x_{21},x_{22},…,x_{2n})间的切比雪夫距离：

$$d_{12} = \max_i(|x_{1i} - x_{2i}|)$$

这个公式的另一种等价形式是：

$$d_{12} = \lim_{k\to\infty}(\sum_{i=1}^{n}|x_{1i} - x_{2i}|^k)^{1/k}$$

(3) Python 实现切比雪夫距离：

```
from numpy import*

vector1 = mat([1,2,3])
vector2 = mat([4,7,5])
print abs(vector1-vector2).max()
```

输出：

```
5
```

5. 夹角余弦(Cosine)

几何中夹角余弦可用来衡量两个向量方向的差异，机器学习中借用这一概念来衡量样本向量之间的差异（如图 1.12）。

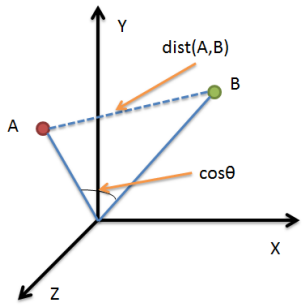


图 1.12　AB 间的夹角余弦 cos(θ)

(1)在二维空间中向量 A(x_1,y_1)与向量 B(x_2,y_2)的夹角余弦公式：

$$\cos(\theta)=\frac{x_1x_2+y_1y_2}{\sqrt{x_1^2+y_1^2}\sqrt{x_2^2+y_2^2}}$$

(2) 两个 n 维样本点 A (x_{11},x_{12},…,x_{1n})与 B(x_{21},x_{22},…,x_{2n})的夹角余弦

类似的，对于两个 n 维样本点 A(x_{11},x_{12},…,x_{1n})与 B(x_{21},x_{22},…,x_{2n})，可以使用类似于夹角余弦的概念来衡量它们间的相似程度。

$$\cos(\theta)=\frac{AB}{|A||B|}$$

即：

$$\cos(\theta)=\frac{\sum_{k=1}^{n}x_{1k}x_{2k}}{\sqrt{\sum_{k=1}^{n}x_{1k}^2}\sqrt{\sum_{k=1}^{n}x_{2k}^2}}$$

夹角余弦取值范围为[-1,1]。夹角余弦越大表示两个向量的夹角越小，夹角余弦越小表示两向量的夹角越大。当两个向量的方向重合时夹角余弦取最大值 1，当两个向量的方向完全相反夹角余弦取最小值-1。

(3)python 实现夹角余弦

```
from numpy import*

cosV12 = dot(vector1,vector2)/(linalg.norm(vector1)*linalg.norm(vector2))
print cosV12
```

输出：

```
0.92966968
```

6. 汉明距离(Hamming distance)

(1)汉明距离的定义

两个等长字符串 s1 与 s2 之间的汉明距离定义为将其中一个变为另外一个所需要作的最小替换次数。例如字符串“1111”与“1001”之间的汉明距离为 2。

应用：信息编码（为了增强容错性，应使得编码间的最小汉明距离尽可能大）。

(2) python 实现汉明距离：

```
from numpy import *
matV = mat([[1,1,0,1,0,1,0,0,1],[0,1,1,0,0,0,1,1,1]])
smstr = nonzero(matV[0]-matV[1]);
print shape(smstr[0])[1]
```

输出：

```
6
```

7. 杰卡德相似系数(Jaccard similarity coefficient)

(1) 杰卡德相似系数

两个集合 A 和 B 的交集元素在 A，B 的并集中所占的比例，称为两个集合的杰卡德相似系数，用符号 J(A,B)表示。

$$J(A,B)=\frac{|A\cap B|}{|A\cup B|}$$

杰卡德相似系数是衡量两个集合的相似度一种指标。

(2) 杰卡德距离

与杰卡德相似系数相反的概念是杰卡德距离(Jaccard distance)。杰卡德距离可用如下公式表示：

$$J_{\delta}(A,B)=1-J(A,B)=\frac{|A\cup B|-|A\cap B|}{|A\cup B|}$$

杰卡德距离用两个集合中不同元素占所有元素的比例来衡量两个集合的区分度。

(3) 杰卡德相似系数与杰卡德距离的应用

可将杰卡德相似系数用在衡量样本的相似度上。

样本 A 与样本 B 是两个 n 维向量，而且所有维度的取值都是 0 或 1。例如：A(0111)和 B(1011)。我们将样本看成是一个集合，1 表示集合包含该元素，0 表示集合不包含该元素。

P：样本 A 与 B 都是 1 的维度的个数

q：样本 A 是 1，样本 B 是 0 的维度的个数

r：样本 A 是 0，样本 B 是 1 的维度的个数

s：样本 A 与 B 都是 0 的维度的个数

那么样本 A 与 B 的杰卡德相似系数可以表示为：

这里 p+q+r 可理解为 A 与 B 的并集的元素个数，而 p 是 A 与 B 的交集的元素个数。

而样本 A 与 B 的杰卡德距离表示为：

$$J = \frac{p}{p+q+r}$$

(4) Python 实现杰卡德距离：

```
from numpy import *
import scipy.spatial.distance as dist   # 导入 scipy 距离公式
matV = mat([[1,1,0,1,0,1,0,0,1],[0,1,1,0,0,0,1,1,1]])
print "dist.jaccard:", dist.pdist(matV,'jaccard')
```

输出：

```
dist.jaccard: [ 0.75]
```

现在，我们有能力为矩阵中对象间的相似程度(接近与远离)提供各种度量方法，以及编码实现。通过计算对象间的距离，我们就可以轻松地得到表 2.8 中的四个对象所属的类别：以克、天为单位的苹果是水果类别的一个实例；以吨、年为单位鲨鱼是大型动物的一个实例。这种区别是明显的，

但是，如果我们考察颜色这个特征，情况可能会有所不同，苹果和梨都有黄色这个特征，像这种情况我们如何区分呢？

1.3.3 理解随机性

让我们改变一下视角，从整体上观察矩阵(集合)中的对象分布与矩阵整体的关系。这需要引入一个新的概念：概率论。概率论是整个数学大厦中比较难理解的一门学科。这多少与直觉有点差异，人们常把概率简单理解为事件发生的可能性。其实这只是概率的表面现象。想要真正理解概率需要弄清楚两个问题：

- 确定性与随机性
- 统计规律

《自然哲学之数学原理》是牛顿最重要的著作，书中构建了科学有史以来第一个完整的、科学的宇宙论和科学理论体系，并试图用统一的力学原因解释宇宙所有的运动和现象，它所造成的影响极其深远。

1986 年，这部划时代巨著出版整整三百周年，英国皇家学会专门举办了隆重的纪念大会。在这次大会上，著名的流体力学权威詹姆士.莱特希尔爵士却发表了令人震惊的道歉宣言。

“今天，我们深深意识到，我们的前辈对牛顿力学惊人成就的崇拜，促使他们认

为世界具有可预见性，的确，我们在 1960 年以前大都倾向于相信这个说法，但现在我们知道这是错误的。我们曾经误导了公众，向他们宣传说满足牛顿运动定律的系统是决定论的，可在 1960 年后这已被证明不是真的，为此，我们愿意向公众表示道歉。”

从认识论角度来看，这是一个划时代的进步，但从常识性的角度而言，这的确让人困惑。如果世界不是确定的，规律从哪里来，我们应该按照何种法则生活？

1963 年，美国气象学家洛仑兹建立了一个描述大气对流状况的数学模型，叫洛仑兹动力学方程。这是个简单确定性方程。因为它只有三个变量用来分别代表大气中的风速、温度和气压；说它是确定性的，因为不含任何随机项，初始值也可以给定。但就是由这个方程所描述的只有三个变量的简单确定性系统里，却让我们出乎意料地见到了随机性的一个典型特征：混沌，见识了混沌非比寻常的特性。

洛仑兹动力学方程描绘出的运动轨迹，它具有一种奇特的形状，像一只展开了双翼的蝴蝶，所以又称为蝴蝶效应（如图 1.13）。在这个奇妙精巧的蝴蝶上，确定性和随机性被统一在一起：一方面，运动的轨迹必然落在“蝴蝶”上，绝不会远离它们而去，这是确定性的表现，表明系统未来的所有运动都被限制在一个明确的范围之内。另一方面，运动轨迹变化缠绕的规则却是随机性的，任何时候你都无法准确判定下一次运动的轨迹将落在“蝴蝶”的哪一侧翅膀上的哪一点上。也就是说，这个系统运动大的范围是确定的，可预测的；但是运动的细节是随机的，不可预测的。

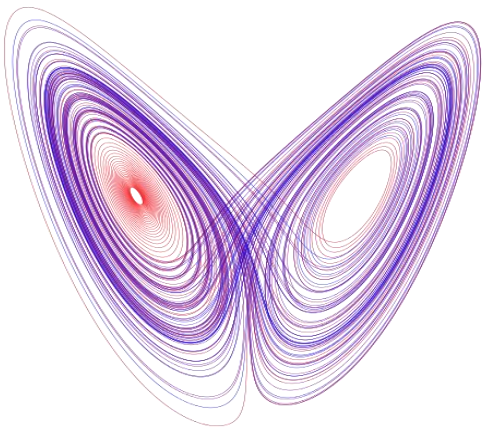


图 1.13　洛仑兹吸引子——蝴蝶效应的理论基础

如果从统计学的角度来看，蝴蝶效应说明了两方面的意义：一方面，样本总体（特征向量）的取值范围一般是确定的，所有样本对象（包括已经存在的和未出现的）的取值都位于此空间范围内。另一方面，无论搜集再多的样本对象，也不能使这种随机性降低或者消失。因此，随机性是事物的一种根本的、内在的、无法根除的性质，也是一切事物（概率）的本质属性。

世界中，绝大多数的规律都来源于统计学。地球围绕太阳旋转的周期大约是 365 天，但每次都有微小差别，否则为什么要制订历法；车辆每次停车的位置都在站台附近，但具体的位置都不一样；抛出硬币落下后不是正面就是反面，但每次实验的结果都很难预期。随机性是隐藏在我们生活中最普遍规律。

1.3.4 回顾概率论

对事物运动这种不确定性(随机性)的度量就是概率论，接下来我们考察一下概率的基本概念。衡量事物运动的随机性，必须从整体而不是局部来认知事物，因为从每个局部，事物可能看起来都是不同的(或相同的)。不像其他的数学体系，这需要概率论架构在一套完整的数学模型之上。

为了更好的理解，我们修改一下上面的例子，假设我们的集合中只有苹果和梨两大类的对象，苹果有 10 个，梨也有 10 个，这次我们仅考察颜色特征。苹果有两种颜色：红色或黄色，其中红色占了 8 个；梨也有两种颜色：黄色或绿色，其中黄色占 9 个，假如从这堆水果中挑出一个黄色水果，问这个水果属于梨的可能性。

概率论基础概念：

- 样本(样本点)：原指随机实验一个结果，可以理解为矩阵中的一个对象：苹果或梨；
- 样本空间：原指随机实验所有结果的集合，可以理解为矩阵的所有对象，引申为对象特征的取值范围：10 个苹果，10 个梨；
- 随机事件：是指样本空间的一个子集，可以理解为某个分类，它实际指向一种概率分布：苹果为红色；梨为黄色；
- 随机变量：可以理解为指向某个事件的一个变量：X{x=黄色}
- 随机变量的概率分布：给定随机变量的取值范围，导致某种随机事件出现的可能性，从机器学习的角度来看，就是符合随机变量取值范围的某个对象属于某个类别或服从某种趋势的可能性。

概率论妙处在于，由于随机性的存在，有时候无法直接讨论随机变量如何从样本空间映射到事件空间，因为随机变量的取值范围允许它可以映射到事件空间中的任一事件。此时，只有通过研究随机变量映射为每个事件的可能性，也就是说某个对象属于每个类别的可能性，才能做出合理的判断。理解了这一层也就理解了概率论的数学本质。

结合本例，我们不去研究黄色的苹果与黄色的梨有什么差别。而承认其统计规律：苹果是红色的概率是 0.8，苹果是黄色的概率就是 1-0.8=0.2，而梨是黄色的概率是 0.9，将其作为先验概率。有了这个先验概率，就可以利用抽样，即任取一个水果，前提是

抽样对总体的概率分布没有影响，通过它的某个特征来划分其所属的类别。黄色是苹果和梨共有的特征，因此，既有可能是苹果也有可能是梨，概率计算的意义在于得到这个水果更有可能的那一种。

这个问题的求解过程就是著名的贝叶斯公式：

$$P(B|A)=\frac{P(A|B)P(B)}{P(A)}$$

用数学的语言来表达，就是已知 P(苹果)=10/(10+10)，P(梨)=10/(10+10)，P(黄色|苹果)=20%，P(黄色|梨)=90%，求 P(梨|黄色)

可得：P(梨|黄色)=P(黄色,梨)/P(黄色)

=P(黄色|梨)P(梨)/P(黄色)

=81.8%

贝叶斯公式贯穿了机器学习中随机问题分析的全过程。从文本分类到概率图模型，其基本原理都是贝叶斯公式，是机器学习领域最重要的基础概念。

1.3.5 多元统计基础

理解了随机性和概率基础，下一步我们与之前介绍的矩阵结合起来，将它扩展到多维的情况。

矩阵是具有相同特征和维度的对象集合，其中每个对象，也称为行向量，都具有一个以上特征。如果，每个特征都用一个随机变量来表示，那么从概率论的角度，一个对象就可以表示为 n 个随机变量的整体，其中 X=(X1,X2, …，Xn)为 n 维随机变量或随机向量。每个对象就是随机向量的一组取值，矩阵中的所有对象构成了随机向量的联合和边缘概率分布。

继续上一节的例子，我们把水果的种类和颜色都看作对象的两个特征，那么这个随机向量的联合概率分布可以写为（如表 1.8）：

表 1.8　水果联合概率分布

颜色 种类	红色 (0)	黄色 (1)	绿色 (2)	颜色
苹果(0)	0.4	0.1	0	0.5
梨(1)	0	0.45	0.05	0.5
水果	0.4	0.55	0.05	1.0

表中间白色部分是水果和颜色两个特征的联合概率分布，灰色部分是两个特征各自取值的边缘概率分布。形式上，以二维随机变量为例，X，Y 的联合概率分布为：

$$P\{X=x_i,\ Y=y_j\}=p_{ij}$$

P{X=苹果, Y=红色}=p_{00}=0.4;　　P{X=梨, Y=红色}=p_{10}=0;

P{X=苹果, Y=黄色}=p_{01}=0.1;　　P{X=梨, Y=黄色}=p_{20}=0.45;

P{X=苹果, Y=绿色}=p_{02}=0;　　P{X=梨, Y=绿色}=p_{30}=0.05。

相应对象的边缘分布表示为：$P\{X=\mathrm{x}_1\}=\sum_{j=0}^{n}p_{1j}$

上例中，X=苹果的边缘概率分布为：$P\{X=\text{苹果}\}=\sum_{\mathrm{j}=0}^{\mathrm{m}}p_{0j}$=0.5；　Y=绿色的边缘概率分布为：$P\{Y=\text{绿色}\}=\sum_{\mathrm{i}=0}^{\mathrm{n}}p_{\mathrm{i}2}$=0.05

随机向量的联合概率分布和边缘概率分布描述了对象特征间的概率关系。机器学习中，对象以及对象构成的矩阵都是多元数据，因此，所有与概率相关的算法都以对象的联合概率分布和边缘概率分布为运算基础，本书中的多元统计算法有：朴素贝叶斯分析、回归分析、统计学习理论、聚类分析、主成分分析和概率图模型等等。

1.3.6　特征间的相关性

前面各章节，我们的分析着重于对矩阵行向量的划分，也就是分类过程。本节侧重于对特征列的研究，特征列的研究主要应用于预测活动，例如，在金融分析中，通过做两支股票价格波动的相关,来判断它们之间的关系，以期达到最大化收益同时最小化风险的目的。抽象一点说，就是在一个时间序列上观察两列数据之间的相关性。

关于预测，概率论提供了一套完整和有效的数学方法。我们继续讲解随机变量的一些重要特征。随机变量，一般是一个向量，可以包含不同取值范围的多个变量，有必要研究一下这些变量的分布情况，也就是随机变量的数字特征，从中发掘出一定的规律性：

- 期望：衡量样本某个特征列取值范围的平均值
- 方差：衡量样本某个特征列取值范围的离散程度
- 协方差矩阵和相关系数：衡量样本特征列之间线性相关性

1 相关系数 (Correlation coefficient)与相关距离(Correlation distance)

(1) 相关系数的定义

$$\rho_{XY}=\frac{Cov(\mathrm{X},\mathrm{Y})}{\sqrt{\mathrm{D}(\mathrm{X})}\sqrt{\mathrm{D}(\mathrm{Y})}}=\frac{E((\mathrm{X}-\mathrm{EX})(\mathrm{Y}-\mathrm{EY}))}{\sqrt{\mathrm{D}(\mathrm{X})}\sqrt{\mathrm{D}(\mathrm{Y})}}$$

相关系数是衡量两个特征列之间相关程度的一种方法，相关系数的取值范围是[-1，1]。相关系数的绝对值越大，则表明特征列 X 与 Y 相关度越高。当 X 与 Y 线性相关时，相关系数取值为 1（正线性相关）或-1（负线性相关）。

(2)相关距离的定义

$$D_{XY} = 1 - \rho_{XY}$$

(3)Python 实现相关系数：数据来源于表 2.6　2-18 岁正常男生体重身高对照表

```
from numpy import *

featuremat =
mat([[88.5,96.8,104.1,111.3,117.7,124.0,130.0,135.4,140.2,145.3,151.9,159.5,165.9,169.8,1
71.6,172.3,172.7],
[12.54,14.65,16.64,18.98,21.26,24.06,27.33,30.46,33.74,37.69,42.49,48.08,53.37,57.08,59.
35,60.68,61.40]])
# 计算均值
mv1 = mean(featuremat[0])# 第一列的均值
mv2 = mean(featuremat[1])# 第二列的均值
# 计算两列标准差
dv1 = std(featuremat[0])
dv2 = std(featuremat[1])
corref = mean(multiply(featuremat[0]-mv1,featuremat[1]-mv2))/(dv1*dv2)
print corref

# 使用 numpy 相关系数得到相关系数矩阵
print corrcoef(featuremat)
```

输出：

```
0.98160142576
[[ 1.          0.98160143]
 [ 0.98160143  1.        ]]
```

相关系数矩阵的含义是，如果把第一个特征列作为参照数据(自己与自己的相关程度为 1)，那么第二个与第一个的相关程度是 98%。

有了数字特征，下面我们可以发展欧氏距离公式为马氏距离公式：

2.马氏距离(Mahalanobis Distance)

（1）马氏距离定义

有 M 个样本向量 X_1~X_m，协方差矩阵记为 S，均值记为向量 μ，则其中样本向量 X 到 u 的马氏距离表示为：

$$D(\mathrm{X}) = \sqrt{(X-\mu)^T \mathrm{S}^{-1}(\mathrm{X}-\mu)}$$

而其中向量 X_i 与 X_j 之间的马氏距离定义为：

$$D(\mathrm{X}_i,\mathrm{X}_j)=\sqrt{(X_i-X_j)^T\,\mathrm{S}^{-1}(X_i-X_j)}$$

若协方差矩阵是单位矩阵（各个样本向量之间独立同分布）,则公式就成了：

$$D(\mathrm{X}_i,\mathrm{X}_j)=\sqrt{(X_i-X_j)^T(X_i-X_j)}$$

也就是欧氏距离了。

若协方差矩阵是对角矩阵，公式变成了标准化欧氏距离。

(2)马氏距离的优点：量纲无关，排除变量之间的相关性的干扰。

(3)马氏距离的 Python 计算：数据来源于表 2.6　2-18 岁正常男生体重身高对照表

```
from numpy import *
featuremat =
mat([[88.5,96.8,104.1,111.3,117.7,124.0,130.0,135.4,140.2,145.3,151.9,159.5,165.9,169.8,1
71.6,172.3,172.7],
[12.54,14.65,16.64,18.98,21.26,24.06,27.33,30.46,33.74,37.69,42.49,48.08,53.37,57.08,59.
35,60.68,61.40]])
covinv = linalg.inv(cov(featuremat))
tp = featuremat.T[0]-featuremat.T[1]
distma = sqrt(dot(dot(tp,covinv),tp.T))
print distma
```

输出：

```
[[ 0.92966634]]
```

1.3.7　再谈矩阵—空间的变换

到目前为止，读者应该已经理解我们的多数运算都是以矩阵为基础的，对象是矩阵中的一个向量，矩阵是由对象构成的集合。现在我们改变一下角度，将“集合”这个术语换成”空间”，来看看矩阵最重要的性质—空间变换。

- ❑ 由特征列的取值范围所有构成的矩阵空间，应具有完整性，即能够反映出事物的空间形式或变化规律。

这是我们在初始矩阵中概括的矩阵特点的第四点。为什么这么说呢？因为机器学习中，训练集构成的矩阵为了能反映事物的规律常常需要包含大量的样本，或因为建模的需要。矩阵中的向量具有很大的维度，这就是我们所说的完整性。虽然理论上，矩阵空间可以包含无限多个向量，但谁也不能对无限个向量计算，也没有这个必要。因此，我们希望能做到两点：

- ❑ 向量之间，以及向量与矩阵之间的运算到底有什么意义；为什么向量和矩阵的运算能够揭示出事物的空间形式。
- ❑ 矩阵能否通过运算抽取出事物最本质的特征，通过这些特征就能反映出

对象集合表达的形式和规律；

1. 向量

在揭示这些秘密之前，我们首先要澄清一些概念。以往，我们在距离公式中隐含了一个假设，这个假设扩展了初等数学中平面直角（正交）坐标系的概念，即把向量解释为 n 维直角坐标系中的一个质点，通过质点间的各种距离公式来进行度量，但这并不是向量的本义。其实无论在几何还是物理上，向量都是个有方向、有大小的量，而向量的点坐标不过表征了该向量与坐标系原点的距离，以及与坐标轴的夹角而已。

现在我们知道向量不是一个点，而是一个有向的线段，线段的长度是向量的大小，线段的指向是向量的方向。仔细想想，这么说还是有点晕，因为我们使用了两个数（点坐标）通过一个称为坐标系的东西来表示另外两个数（长度和角度），那么这个坐标系是什么呢，它从何而来呢？如果不解释这个问题，理解向量的长度和角度仍旧是无源之水，无本之木。

2. 向量张成空间

现在，考察一下我们熟悉的 n 维正交空间的性质：在零维空间中没有长度也没有方向，因此没有量的概念，也没有运算规则，我们形象地称其为原点；扩展到一维空间，数轴是个向量，其长度就是它的坐标（可以无限延伸），方向有两个，或正或负，其上点的运算规则是标量的运算规则，当然这不是我们研究的重点；再扩展到平面直角坐标系，x 轴与 y 轴也是两个向量，长度可以无限延伸，它们之间在方向上相互正交，定位其上的点需要考虑它与原点和坐标轴 x，y 之间的关系，因此是个二维向量，例如，直角坐标系中向量的长度可以通过它与原点的欧氏距离得到，方向可以通过它与坐标轴的夹角余弦得到；再推而广之，n 维直角坐标系的每个坐标轴都是向量，长度可以无限延伸，它们两两之间在方向上相互正交，定位其上的点需要考虑它与原点和每个坐标轴的关系，因此是个 n 维向量。

现在应该清楚了，从上面的研究中，我们得出一个重要的结论：向量的长度和方向都是相对于其他向量的量，长度相对于原点，方向相对于坐标轴。要想构成一个二维的空间（注意，不一定是正交空间），就必须有两个或两个以上存在交点（原点）的向量（或其延长线）。从线性代数的角度说，构成空间的两个向量之间必须线性无关，也就是说这两个向量的维度必须与它们构成的矩阵的秩相等，而这两个向量就称为这个空间的基或基底。基底是向量空间的一组很“结实”的向量集合，每一个基就像房梁和立柱一样支撑起整个空间大厦，并衍生出空间内的全部向量。

我们给一个向量空间找一个基底，本质上就是为了给这个空间定一个坐标系(注意，不一定是正交)，以方便定位和计算向量。基底就成为坐标系在线性空间中的推广。基底向量对应坐标系的坐标轴，有几个基底向量就有几个坐标轴，n 维空间的一个基底就需要有 n 个基底向量。不难看出，所谓 n 维正交空间是由 n 个彼此正

交的基底向量构成的空间（如图 1.14）。

由基底构成的空间内部的任何向量都可以由其基底来线性表示，也就是说，基底构成的空间内部的任何向量都可以通过基底数乘和加法来得到。我们来看一个例子：

向量组 $\boldsymbol{\alpha}_1$(3，1)和 $\boldsymbol{\alpha}_2$(1，3)构成一个基底空间，通过 $\boldsymbol{\alpha}_1+\boldsymbol{\alpha}_2$ 得到 $\boldsymbol{\alpha}_3$ (4，4)，它位于空间的内部，这就是向量加法的几何意义。数乘更简单，乘数相当于基底空间的内部向量长度的倍数。

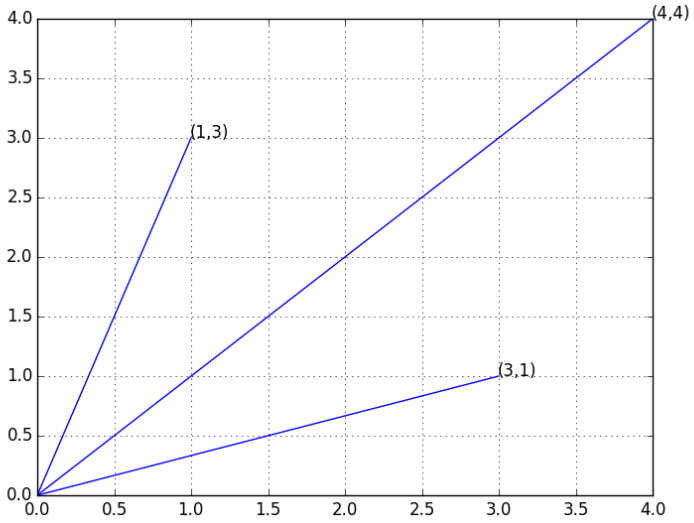


图 1.14　基底空间

3.向量的空间变换

再看一个例子，平面直角坐标系中有三个向量：$\boldsymbol{\alpha}_1$(3，1)，$\boldsymbol{\alpha}_2$ (1，3)和 $\boldsymbol{\alpha}_3$ (1，2)，注意，它们的基底是平面直角坐标系。我们让向量组 $\boldsymbol{\alpha}_1$ (3，1)和 $\boldsymbol{\alpha}_2$(1，3)构成的一个新的基底空间，这次我们让向量 $\boldsymbol{\alpha}_3$ (1，2) 线性变换到 $\boldsymbol{\alpha}_1$ (3，1)和 $\boldsymbol{\alpha}_2$(1，3)构成的新基底空间之中（如图 1.15）。

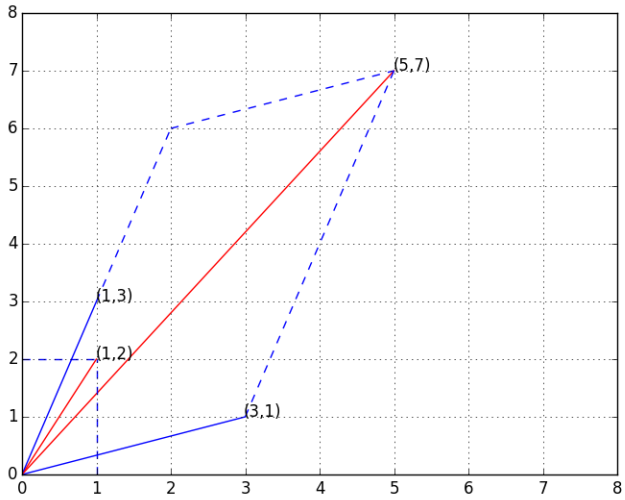


图 1.15　向量的空间变换

从原理上讲其实很简单。$\boldsymbol{\alpha}_3$(1，2)决定了一个边长分别为 1×2 的矩形空间，这个矩形空间的夹角为 90 度。现在我们要把这个矩形空间拉伸(旋转)延展为边长为$(1*\sqrt{10})$×$(2*\sqrt{10})$，夹角余弦为 0.6（大约 53 度）的平行四边形中，计算拉伸后这个平行四边形中对角线 $\boldsymbol{\alpha}_3$’的坐标，也就是上图中由实线和虚线构成的平行四边形的对角线坐标。

可以使用平行四边形对角线公式求解这个问题，但是，现在我们使用向量乘以矩阵的乘法公式：$\boldsymbol{\alpha}_3$* [$\boldsymbol{\alpha}_1$,$\boldsymbol{\alpha}_2$]=[1,2]*[[1,3],[3,1]] 得到[5,7],也可以写成[$\boldsymbol{\alpha}_1$,$\boldsymbol{\alpha}_2$] *$\boldsymbol{\alpha}_3^T$的形式，$\boldsymbol{\alpha}_3^T$是 $\boldsymbol{\alpha}_3$ 的转置。到这里就应该明白向量与矩阵的乘法含义了吧。

所谓向量与矩阵的乘法就是一个向量从一个线性空间(坐标系)，通过选取一个新的基底([$\boldsymbol{\alpha}_1$,$\boldsymbol{\alpha}_2$] 构成的新坐标系)，变换到这个新基底构成的另一个线性空间的过程。

要特别要说明的是，一个向量的线性变换在几何上表现为：缩放、旋转这两个动作，及其这两个动作的组合：拉伸、压缩等。注意，这些动作的一个共性是都要通过原点。

4.向量组与矩阵的变换--理解矩阵乘法

理解了向量与矩阵的线性变换，再看向量组与矩阵之间的变换就简单了。我们已经可以把一个向量变换到一个空间中，我们现在想把一组向量都变换到另一个空间中，这就需要通过矩阵乘法来计算，当然也就是矩阵乘法的意义。在变换时，左边是一组在原有空间中的向量组，而且数量可以是不定的，因此，矩阵乘法对乘号左边矩阵的行数不限制，但是列数要等于乘号右边矩阵的行数，这样的变换才有定义。那为什么乘法公式选择要左边的列数等于右边矩阵的行数呢，想来原因如下：

左边的矩阵被定义为一个向量组，其列数被认为是向量的维度；右边的矩阵被定义为一个线性空间，相乘的运算就是做将这个向量组线性变换到新的线性空间中。也就是说，右边矩阵的行数最少要满足是由基底向量构成的线性空间的维度，或方程组的秩。这样就保证了这个变换必然存在。

因此，矩阵乘法公式左右两边看是都是矩阵，但其意义是完全不同的，所以不满足交换律。

5.线性变换—特征值与特征向量

我们知道，矩阵乘法对应了一个变换，它可以把一个向量变成另一个方向或长度都不同的新向量。在这个变换的过程中，原向量主要发生旋转、伸缩的变化。但是线性变换的规则要求这个变换必须通过由基底向量构成的坐标系的原点。简单说，以平面直角坐标系为例，这种变换不过就是沿着通过原点的某条直线的缩放和旋转。

这种变换必然导致有这样一组向量存在，即被变换的线性空间内的某个向量或向量组只发生了伸缩变换，而没有产生旋转的效果，也就是说在线性变换下，在它们所在的直线上保持不变；那么这些向量就称为这个矩阵的特征向量，伸缩的比例就是特征值。

这里如果特征值变为负值，即特征向量旋转 180 度，也可看作方向不变，而伸缩比为负值。所以特征向量也叫线性不变量。特征向量的不变性使它们变成与其自身共线的向量，特征向量在线性变换后可能伸长或缩短，或反向伸长或反向缩短，甚至变成零向量（特征值为零时），但与它变换后的向量应在同一直线上。

矩阵的特征向量和特征值：

$$Av=\lambda v$$

这里 A 是原矩阵，v 是特征向量，λ 是特征值。下面我们使用 Python 求取矩阵的特征值和特征向量：

```
A = [[8,1,6],[3,5,7],[4,9,2]]
evals, evecs = linalg.eig(A)
print "特征值:",evals,"\n 特征向量:", evecs
```

结果：

```
特征值: [ 15.                4.89897949   -4.89897949]
特征向量: [[-0.57735027 -0.81305253 -0.34164801]
 [-0.57735027   0.47140452 -0.47140452]
 [-0.57735027   0.34164801   0.81305253]]
```

下面我们给出使用手工的方法求取矩阵的特征值：

```
#  手动计算特征值：
# Aeig = lambda*I-A= [[lambda-8,-1],[-6;-3,lambda-5,-7],[-4,-9,lambda-2]]
# (lambda-8)*(lambda-5)*(lambda-2)-190-24*(5-lambda)-3*(2-lambda)-63*(8-lambda)
equationA= [1,-15,-24,360]   #  得到系数方程矩阵
```

```
evals = roots(equationA)        # 计算矩阵方程的根
print "特征值:" ,evals
```

结果：

```
特征值: [ 15.                4.89897949   -4.89897949]
```

有了特征值和特征向量，我们可以还原出原矩阵：

$$A=Q\Sigma Q^{-1}$$

A 是原矩阵，Q 是特征向量，Σ是特征值构成的对角矩阵。下面是代码实现：

```
# 特征值和特征向量,还原原矩阵
sigma = evals*eye(m)
print evecs*sigma*linalg.inv(evecs)
```

结果：

```
[[ 8.   1.   6.]  [ 3.   5.   7.]  [ 4.   9.   2.]]
```

1.3.8 数据归一化

归一化是一种简化计算的方式，即将有量纲的表达式，经过变换，化为无量纲的表达式，成为标量，是机器学习的一项基础工作。

归一化方法有两种形式，一种是把数变为(0,1)之间的小数，一种是把有量纲表达式变为无量纲表达式。

在统计学中，归一化的具体作用是归纳统一样本的统计分布性。归一化在(0,1)之间是统计的概率分布，归一化在-1--+1 之间是统计的坐标分布。

从集合的角度来看，可以做维度的归一，即抽象化归一，把不重要的、不具可比性的集合中的元素的属性去掉，保留人们关心的那些属性，这样，本来不具有可比性的对象或是事物，就归一为一类，然后就可以比较了，并且，人们往往喜欢用相对量来比较，比如人和牛，身高体重都没有可比性，但身高/体重的值，就可能有了可比性，人吃多少，牛吃多少，可能也没有直接的可比性，但相对于体重，或是相对于一天的各自的能量提供需要的食量，就有了可比性；这些，从数学角度来看，可以认为是把有纲量变成了无纲量了。

数据标准化（Data Normalization Method）数据标准化是归一化的一种形式，本质上是也是消除量纲的差异，比较认识。数据的标准化的主要方法是按比例缩放，使之落入一个小的特定区间。由于特征向量的各个特征的度量单位不同，为了能够特征之间参与评价计算，需要对特征做规范化处理，通过函数变换或统计方法将其数值映射到某个数值区间。

对欧氏距离的标准化

(1)标准欧氏距离的定义

标准化欧氏距离是针对简单欧氏距离的缺点而作的一种改进方案。标准欧氏距离的思路：既然数据各维分量的分布不一样，先将各个分量都“标准化”到均值、方差相等。

而且标准化变量的均值为 0，方差为 1。因此样本集的标准化过程(standardization)用公式描述就是：

$$X^*=\frac{X-M}{S}$$

标准化后的值 = (标准化前的值 － 分量的均值)/分量的标准差

经过简单的推导就可以得到两个 n 维向量 A $(x_{11},x_{12},\ldots,x_{1n})$与 B$(x_{21},x_{22},\ldots,x_{2n})$间的标准化欧氏距离的公式：

$$d_{12}=\sqrt{\sum_{k=1}^{n}(\frac{X_{1k}-X_{2k}}{S_k})^2}$$

如果将方差的倒数看成是一个权重，这个公式可以看成是一种加权欧氏距离(Weighted Euclidean distance)。

(2)标准化欧氏距离 python 实现

```
vectormat = mat([[1,2,3],[4,5,6]])
v12 = vectormat[0]-vectormat[1]
print sqrt(v12*v12.T)
#norm
varmat = std(vectormat.T,axis=0)
normvmat = (vectormat-mean(vectormat))/varmat.T
normv12 = normvmat[0]-normvmat[1]
print sqrt(normv12*normv12.T)
```

输出：

```
[[ 5.19615242]]
[[ 6.36396103]]
```

1.4 数据处理与可视化

本节，主要讲 Python 在数据处理和可视化的方面的一些技术，主要包含三大方面：

- 数据的导入和内存管理
- 基本数据结构与可视化:表，树，图
- 数据可视化

1.4.1 数据的导入和内存管理

机器学习一般要处理的都是海量的表格和文本，小数据集的几十 MB，大的几个 TB，如果处理不当，有时候是个比较麻烦的事情。本节主要讲解一下数据导入、持久化的一些方法。

1.数据表文件的读取：

机器学习常用到大量的数据表。一般从几 K 到几十 MB 不等。现在大多数 64 位的系统内存都在4GB以上，空闲内存最少也有2GB，因此这类数据表的处理比较简单，可以直接读入内存，并结构化：

数据文件样式（如图 1.16）：

香菜 2.80 4.00 4.00 4.00 2.20 山西汾阳市晋阳农副产品批发市场 山西 汾阳
大葱 2.80 2.80 2.80 2.80 2.60 山西汾阳市晋阳农副产品批发市场 山西 汾阳
葱头 1.60 1.60 1.60 1.60 1.60 山西汾阳市晋阳农副产品批发市场 山西 汾阳
大蒜 3.60 3.60 3.60 3.60 3.00 山西汾阳市晋阳农副产品批发市场 山西 汾阳
蒜苔 6.20 6.40 6.40 6.40 5.20 山西汾阳市晋阳农副产品批发市场 山西 汾阳
韭菜 5.60 5.60 5.60 5.60 4.60 山西汾阳市晋阳农副产品批发市场 山西 汾阳
青椒 5.20 5.00 5.00 5.00 4.80 山西汾阳市晋阳农副产品批发市场 山西 汾阳
茄子 5.40 4.40 4.40 4.40 5.40 山西汾阳市晋阳农副产品批发市场 山西 汾阳
西红柿 4.80 5.00 5.00 5.00 5.00 山西汾阳市晋阳农副产品批发市场 山西 汾阳
黄瓜 3.40 4.00 4.00 4.00 2.60 山西汾阳市晋阳农副产品批发市场 山西 汾阳
青冬瓜 1.60 1.60 1.60 1.60 1.50 山西汾阳市晋阳农副产品批发市场 山西 汾阳
西葫芦 2.80 3.00 3.00 3.00 2.60 山西汾阳市晋阳农副产品批发市场 山西 汾阳
白萝卜 1.20 1.20 1.20 1.20 0.80 山西汾阳市晋阳农副产品批发市场 山西 汾阳
胡萝卜 1.50 1.50 1.50 1.50 1.50 山西汾阳市晋阳农副产品批发市场 山西 汾阳
土豆 1.80 2.00 2.00 2.00 1.80 山西汾阳市晋阳农副产品批发市场 山西 汾阳
豆角 9.00 10.40 10.40 10.40 8.60 山西汾阳市晋阳农副产品批发市场 山西 汾阳
尖椒 5.40 5.40 5.40 5.40 4.40 山西汾阳市晋阳农副产品批发市场 山西 汾阳
面粉 3.44 3.44 3.44 3.44 3.44 山西汾阳市晋阳农副产品批发市场 山西 汾阳
大米 6.00 6.00 6.00 6.00 6.00 山西汾阳市晋阳农副产品批发市场 山西 汾阳
豆油 8.40 8.40 8.40 8.40 8.40 山西汾阳市晋阳农副产品批发市场 山西 汾阳
富士苹果 7.00 7.00 7.00 7.00 7.00 山西汾阳市晋阳农副产品批发市场 山西 汾阳
香蕉 4.20 4.00 4.00 4.00 4.20 山西汾阳市晋阳农副产品批发市场 山西 汾阳
鸡蛋 7.80 7.80 7.80 7.80 8.00 山西汾阳市晋阳农副产品批发市场 山西 汾阳
猪肉 21.00 21.00 21.00 21.00 20.60 山西汾阳市晋阳农副产品批发市场 山西 汾阳
牛肉 50.00 50.00 50.00 50.00 50.00 山西汾阳市晋阳农副产品批发市场 山西 汾阳

图 1.16　数据文件样式

Python 读取数据表例程:

```
# -*- coding: utf-8 -*-

import sys
import os
from numpy import *

#  配置 utf-8 输出环境
reload(sys)
sys.setdefaultencoding('utf-8')

#数据文件转矩阵
```

```
# path: 数据文件路径
# delimiter: 行内字段分隔符
def file2matrix(path,delimiter):
    recordlist = []
    fp = open(path,"rb") # 读取文件内容
    content = fp.read()
    fp.close()
    rowlist = content.splitlines()    # 按行转换为一维表
    # 逐行遍历，结果按分隔符分割为行向量
    recordlist=[map(eval, row.split(delimiter)) for row in rowlist if row.strip()]
    return mat(recordlist)    # 返回转换后的矩阵形式

root = "testdata"          # 数据文件所在路径
pathlist = os.listdir(root) # 获取路径下所有数据文件
for path in pathlist:
    recordmat = file2matrix(root+"/"+path,"\t")  # 文件到矩阵的转换
    print shape(recordmat)                       # 输出解析后矩阵的行、列数
```

输出结果:

```
(4103L, 9L)
(2737L, 9L)
```

2.对象的持久化：

有时候，我们希望数据以对象的方式保存。Python 提供了 cPicke 模块支持对象的读写：

继续上面的代码：

```
import cPickle as pickle # 导入 cPickle 库

file_obj = open(root+"/recordmat.dat", "wb")
pickle.dump(recordmat[0],file_obj) #将生成的矩阵对象保存到指定位置
file_obj.close()
```

此行代码可以将刚才转换为矩阵的数据持久化为对象的文件。

读取序列化后的文件:

```
read_obj = open(root+"/recordmat.dat", "rb")
readmat = pickle.load(read_obj) #从指定位置读取对象
print shape(readmat)
```

输出结果:

```
(4103L, 9L)
```

1.高效读取大文本文件：

笔者曾经处理过最大的单一文本文件有近 40G。 数据来自 DBPedia 的 RDF 实例数据集，格式上类似 XML 文档。这种情况可以使用如下函数逐行读取，逐行处理。

```
# 按行读文件，读取指定行数：nmax=0 按行读取全部
def readfilelines(path,nmax=0):
    fp = open(path,"rb")
    ncount = 0 # 已读取行
    while True:
        content = fp.readline()
        if content =="" or (ncount>=nmax and nmax!=0): # 判断到文件尾，或读完指定行数
            break
        yield content   # 返回读取的行
        if nmax != 0 :    ncount += 1
    fp.close()

path = "testdata/01.txt"                    # 数据文件所在路径
for line in readfilelines(path,nmax=10):   # 读取 10 行
    print line.strip()
```

1.4.2 表与线性结构的可视化

python 提供了四种容器结构 list, dict, set,tuple 来装载数据。其中线性结构有两种 list 和 tuple。由于 tuple 是只读结构，仅用于外部生成器生成的数据，所以最常用的线性结构就是 list。Numpy 的矩阵结构可在 matplotlib 中实现可视化，而且支持矢量编程，这样可视化的问题就变得简单很多。

```
import numpy as np
import matplotlib.pyplot as plt

# 曲线数据加入噪声
x = np.linspace(-5,5,200);
y = np.sin(x);    # 给出 y 与 x 的基本关系
yn = y+np.random.rand(1,len(y))*1.5 ;      # 加入噪声的点集
# 绘图
fig = plt.figure()
ax = fig.add_subplot(111)
ax.scatter(x,yn,c='blue',marker='o')
ax.plot(x,y+0.75,'r')
plt.show()
```

输出结果（如图 1.17）：

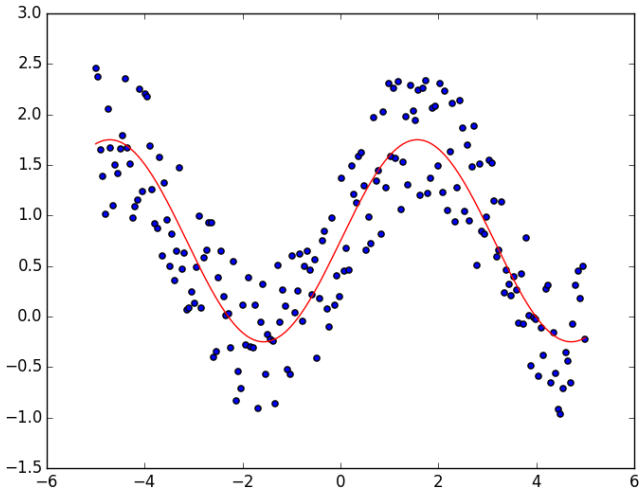


图 1.17　数据可视化 1

1.4.3　树与分类结构的可视化

树状结构是一种非线性结构，一般用于分类树算法，Python 使用 dict 字典型数据结构实现存储，但 matplotlib 没有提供专门绘制树的 API。Peter Harrington 提供了一个简单绘制树的模块。

```
import numpy as np
import matplotlib.pyplot as plt
import treePlotter as tp

# 配置 utf-8 输出环境
reload(sys)
sys.setdefaultencoding('utf-8')

# 绘制树
myTree = {'root': {0: 'leaf node', 1: {'level 2': {0: 'leaf node', 1: 'leaf node'}},2:{'level2': {0: 'leaf node', 1: 'leaf node'}}}}
tp.createPlot(myTree)
```

输出结果（如图 1.18）：

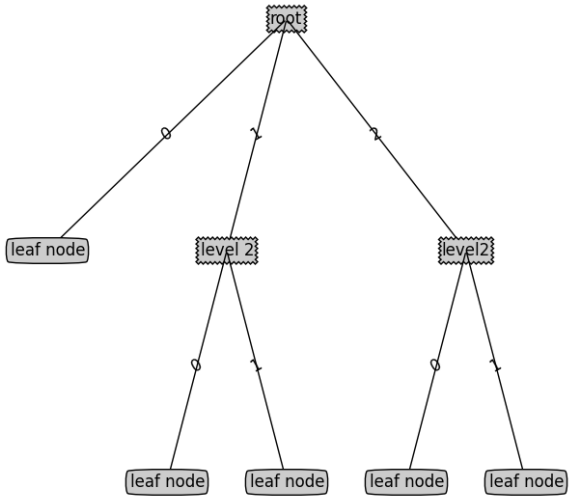


图 1.18　数据可视化 2

1.4.4　图与网络结构的可视化

图和网络结构是神经网络和贝叶斯网络中重要的数据结构。完整的结构一般使用 dict 加 list 进行存储：

```
Node = {'node name':node_info}
Arc = {'arc name':list[node1,node2,...]}
```

在算法中，经常简化存储为邻接矩阵的形式，使用 Numpy 的矩阵结构存储点坐标；弧的坐标使用使用距离公式计算。可视化时可以生成 x 轴的 list 和 y 轴的 list 显示在图片中。

```
# -*- coding: utf-8 -*-
from numpy import *
import matplotlib.pyplot as plt

dist = mat([[0.1,0.1],[0.9,0.5],[0.9,0.1],[0.45,0.9],[0.9,0.8],[0.7,0.9],[0.1,0.45],[0.45,0.1]])
m,n = shape(dist)

fig = plt.figure() # 绘图
ax = fig.add_subplot(111)
ax.scatter(dist.T[0],dist.T[1],c='blue',marker='o')
for point in dist.tolist():
    plt.annotate("("+str(point[0])+", "+str(point[1])+")",xy = (point[0],point[1]))
xlist = []; ylist = []
```

```
for px,py in zip(dist.T.tolist()[0],dist.T.tolist()[1]):
        xlist.append([px])
        ylist.append([py])
ax.plot(xlist,ylist,'r')
plt.show()
```

输出结果（如图 1.19）：

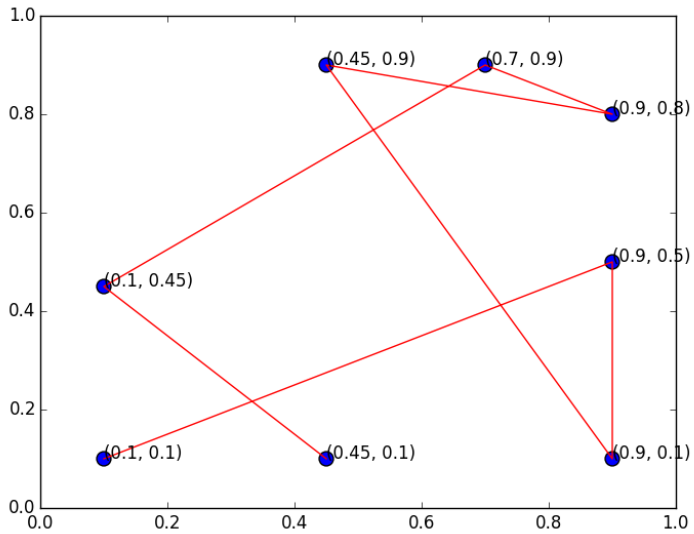


图 1.19　数据可视化 3

有关更多的可视化细节请查看: http://matplotlib.org/examples/index.html

1.5　附录: Linux 下部署 Python 机器学习开发环境

1.5.1　Linux 发行版的选择

对于操作系统的考虑是综合性的问题。经过多方面的测试和评估，最终定出以下几个方面作为生产环境的选择标准：

- 较小的系统资源占用
- 稳定的运行效率
- 高效的内存管理机制
- 支持 64 位应用
- 支持集群部署和分布式应用

- 混合编程开发:开发工具的统一管理

在实际项目中，CentOS 6.5 作为生产环境应用平台;Win7 作为试验环境的应用平台:

- Windows 7 64 位 ，Python 开发工具：UltraEdit
- Centos 6.5 64 位，Python 开发工具：Geany

从上述内容可见，在 Win7 中的开发工具与 CentOS 中的开发工具差别不大。仅在 Web 和 Python 中有所区别。大家对 UltraEdit 比较熟悉，这里就不具体介绍了。

Geany 最初是一个文本编辑器，同时支持 Linux 和 Windows 平台。随着版本的发展，也加入了很多编译和执行选项，以及终端配置，可用于基于文件的脚本语言的编译和开发。关于软件的源码下载和应用参见:http://www.geany.org/，下文将介绍它的具体配置。

从源代码编译安装 Geany IDE。从 http://www.geany.org/下载 geany 的源代码，进行安装。

```
./configure --prefix=/usr/local/geany
make && make install
```

注意编译选项:/configure –prefix=“/usr/local/geany”。引号内为选择的安装目录。安装过程中需要依赖 GTK2.0 的相关包，可以直接

```
yum install gtk2.0
```

获得。此安装非常简单。

1.5.2. Centos 部署多版本 Python 实例

不幸的是，CentOS 已经预装了 python2.66 的开发环境。这样我们必须要安装一套全新的 python2.7 环境，这个环境与默认环境在所有方面（配置文件、库文件、头文件、执行文件、扩展模块）都完全隔离，即不改变系统默认执行环境的优先级。在一个完全隔离的空间上安装的 numpy，scipy，matplotlib 等模块。系统任何 yum 级别的更新和扩展都只会作用于默认的系统版本，而不会作用于这个新版本。这样做的好处是不会影响系统级别的 python2.66 应用：yum，ibus 等。但也有个问题，任何扩展都需要从源代码级别进行安装。为此，我们仔细考虑，规划 python2.7 新版本的安装策略如下：

1.与系统默认安装的完全分离，不改变系统默认 python2.66 的任何配置以及调用的方式

2.执行 python2.7 程序或调用新安装的 python2.7 模块时，需要必须指定版本号：python2.7 XXX.py

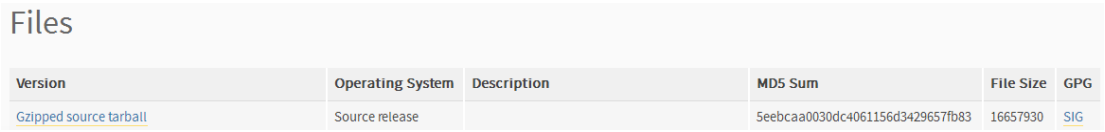
3.任何 python2.7 的扩展都使用 python 专门的扩展程序进行安装（ezinstall,pip 等）—减少源代码安装中包依赖的繁琐。

python2.7 的扩展中如需要 c/c++语言的源码包，例如 scipy 需要预装 blas，此时才

使用 yum 进行安装，安装到全局。有了这些原则，从源代码安装 python2.7 就变得清晰多了。

1.从源代码编译安装 python2.7

首先现在 python2.7 的源码包，我安装的是 python2.7.9，下载地址：https://www.python.org/downloads/source/



Files

Version	Operating System	Description	MD5 Sum	File Size	GPG
Gzipped source tarball	Source release		5eebcaa0030dc4061156d3429657fb83	16657930	SIG

下载后解压缩，执行安装程序：

```
./configure --prefix=/usr/local/python2.7
make && make install
```

执行之后的安装文件位于/usr/local/python2.7 目录下。进入这个目录，将/usr/local/python2.7/bin/python2.7 这个文件创建三个同名的链接文件，分别保存到 /bin；usr/bin;/usr/local/bin 的目录下。这三个目录决定了系统默认调用 python 的路径：

```
[root@localhost ~]#ln –s /usr/local/python2.7/bin/python2.7 /bin/python2.7
[root@localhost ~]#ln –s /usr/local/python2.7/bin/python2.7 /usr/bin/python2.7
[root@localhost ~]#ln –s /usr/local/python2.7/bin/python2.7 /usr/ local/bin/python2.7
```

此时运行 python 和 python2.7 检测版本信息

```
[root@localhost ~]# python2.7 -V
Python 2.7.9

[root@localhost ~]# python -V
Python 2.6.6
```

跟据上面的安装,配置 geany 开发环境：在 geany 的菜单“生成”项—>“设置生成命令”（如图 1.20）：

图 1.20　设置生成命令

输入以上内容。即可撰写脚本进行编译。

```
# -*- coding: utf-8 -*-
# Filename : cnstring.py

cnstr = "中文 test"
print cnstr , len(cnstr) # 10

#  中英文字符串混合遍历
utfstr = unicode (cnstr, "utf-8")
print utfstr,len(utfstr)
for c in utfstr:
    print c
```

可能执行后不会有输出。

选择“编辑”-->“首选项”-->工具，输入以下文字（如图 1.21）：

图 1.21　首选项输入文字

运行即可看见输出（如图 1.22）：

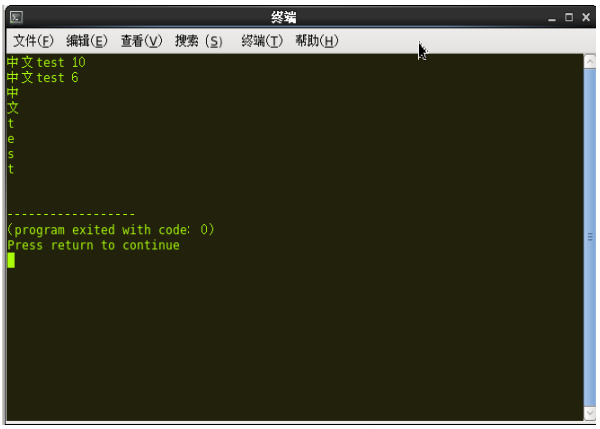


图 1.22　运行结果

这就是我们希望达到的效果。这种完全隔离式的安装方式，操作简单，维护方便，建议为各版本 python 的安装首选。

1.5.3. 安装 numpy, scipy,matplotlib 开发包

1.从源代码编译安装 Numpy

从 numpy 网站下载最新版的源代码：

http://sourceforge.net/projects/numpy/files/NumPy/1.9.0/ 解压缩之后，直接运行下面的指令进行安装：

```
[root@localhost numpy-1.9.0]# python2.7 setup.py build
[root@localhost numpy-1.9.0]# python2.7 setup.py install
```

测试安装：

```
import numpy
numpy.test(1,1)
```

应能成功执行。

2.从源代码编译安装 Scipy

从 Scipy 网站下载最新版的源代码：http://sourceforge.net/projects/scipy/files/scipy/0.14.0/ 解压缩之后，直接运行下面的指令进行安装：

```
[root@localhost scipy-0.14.0]# python2.7 setup.py build
```

安装过程中可能会遇到如下问题：

```
Blas ([url=http://www.netlib.org/blas/]http://www.netlib.org/blas/[/url]) sources not found.
Directories to search for the sources can be specified in the
numpy/distutils/site.cfg file (section [blas_src]) or by setting
the BLAS_SRC environment variable.
warnings.warn(BlasSrcNotFoundError.__doc__)
```

因为是 blas 是基于全局的库，直接 yum 安装：

```
yum install blas blas-devel
```

之后继续运行 python2.7 setup.py build。遇到下面问题：

```
Lapack ([url=http://www.netlib.org/lapack/]http://www.netlib.org/lapack/[/url]) sources not found.
Directories to search for the sources can be specified in the
numpy/distutils/site.cfg file (section [lapack_src]) or by setting
the LAPACK_SRC environment variable.
warnings.warn(LapackSrcNotFoundError.__doc__)
```

因为是 lapack 也是基于全局的库，直接 yum 安装：

```
yum install lapack lapack-devel
```

之后运行 python2.7 setup.py build 完成构建，接下来：

```
[root@localhost scipy-0.14.0]# python2.7 setup.py install
```

完成安装

测试安装：

```
import scipy
scipy.test(10)
```

应能成功执行。

3.从源代码编译安装 matplotlib

从 matplotlib 网站下载最新版的 matplotlib 源代码：

```
[root@localhost matplotlib-1.4.0]# python2.7 setup.py build
```

中间可能会有网络问题，需要多试几次

```
[root@localhost matplotlib-1.4.0]# python2.7 setup.py install
```

提示错误：

```
freetype: no[Requires freetype2 2.4 or later. Found 2.3.11.]
```

yum 中没有 freetype2 2.4 版本以上的库，需要下载升级。下载地址：http://sourceforge.net/projects/freetype/files/freetype2/2.4.12/，源码包下载后解压，配置，编译，安装

```
./configure
make && make install
```

继续。

```
[root@localhost matplotlib-1.4.0]# python2.7 setup.py install
```

成功完成。

1.5.4. 安装 scikit-learn 开发包

pip 下载安装 scikit-learn：scikit-learn 建议使用 pip 安装，这是 python 的自动安装方式：

```
[root@localhost ~]# wget [url=http://python-distribute.org/distribute_setup.py]http://python-distribute.org/distribute_setup.py[/url]
[root@localhost ~]# python2.7 distribute_setup.py
[root@localhost ~]# wget [url=https://github.com/pypa/pip/raw/master/contrib/get-pip.py]https://github.com/pypa/pip/raw/master/contrib/get-pip.py[/url]
[root@localhost ~]# python2.7 get-pip.py
```

输入 pip 命令如果能看到 pip 的介绍信息，那么证明安装成功了。

```
[root@localhost ~]# /usr/local/python2.7/bin/pip install -U scikit-learn
```

看到如下信息：

```
Successfully installed scikit-learn
Cleaning up..
```

终于 Scikit-Learn 在 Linux 安装成功。

1.6 结语

本章提纲挈领的介绍了机器学习常用的一些软件工具、数学理论和数据可视化的方法。

软件语言方面本书选择 Python 作为程序设计语言，简要介绍了 Python 解释器的安装、算法包的安装和 IDE 开发环境的搭建。附录部分还介绍了 Linux 操作系统下的 Python 环境安装。本书还介绍了 Python Numpy 包在矢量编程方面的应用。并用矢量编

程的方式实现了常用的线性代数和矩阵的大多数算法公式。

本章的第二个重点是系统的介绍了以矩阵为核心的机器学习必需的数学概念和原理。包括如何建立数据表，数据表如何转换为矩阵（初始矩阵一节），同时引入了对象的概念；接下来，以矢量编程的方式列举了机器学习常用的各类距离公式和程序实现；最后用一个案例详细介绍了矩阵的空间变换原理，以及一些较难理解的矩阵乘法的意义和特征值特征向量的意义。

在概率论方面，本章讲解了随机性和概率论的关系，通过一个实例讲解了贝叶斯公式和相关系数的概念。在数据归一化的一节，简要介绍了归一化的概念和标准化欧式距离的算法公式。

最后介绍了大规模数据处理的一些程序设计方法，以及计算结果可视化的方法。

第二章　中文文本分类

本章以实例的形式，系统的介绍了中文文本分类的整体流程和相关算法。内容从文本挖掘的大背景开始，以文本分类算法为中心，详细介绍了一个中文文本分类项目的流程及其周边方方面面的知识，知识点涉及中文分词、向量空间模型、TF-IDF 方法直到几个典型的文本分类算法和评价指标等等。因为是第一章，为便于读者的学习，所使用的算法技术并不复杂：

- 朴素的贝叶斯算法
- KNN 最近邻算法

代码讲解分为两大部分：

- 外部库如 jieba 分词、Scikit-Learning：提供详细的使用说明和案例代码
- 算法：提供详细的讲解和注释

与第一章相同，本章使用矢量编程，代码结构较很简单。阅读起来一目了然。通过学习本章，读者有能力实现小型的文本分类系统，并应用于实践之中。

2.1　文本挖掘与文本分类的概念

文本挖掘(Text Mining)是从非结构化文本信息中获取用户感兴趣或者有用的模式的过程。其中被普遍认可的文本挖掘定义如下:

文本挖掘是指从大量文本数据中抽取事先未知的、可理解的、最终可用的知识的过程，同时运用这些知识更好地组织信息以便将来参考。

简言之，文本挖掘就是从非结构化的文本中寻找知识的过程。

文本挖掘的七个主要领域：

- 搜索和信息检索（IR）：存储和文本文档的检索，包括搜索引擎和关键字搜索。
- 文本聚类：使用聚类方法，对词汇，片段，段落或文件进行分组和归类。
- 文本分类：对片段，段落或文件进行分组和归类，使用数据挖掘分类方法的基础上，经过训练的标记示例模型。
- Web 挖掘：在互联网上进行数据和文本挖掘，并特别关注在网络的规模和相互联系。
- 信息抽取（IE）：从非结构化文本中识别与提取有关的事实和关系;从非结构化和半结构化文本制作的结构化数据的过程。
- 自然语言处理（NLP）：将语言作为一种有意义、有规则的符号系统，在底层解析和理解语言的任务（例如，词性标注）;目前的技术主要从语法、语义的角度发现语言最本质的结构和所表达的意义。
- 概念提取：把单词和短语按语义分组成意义相似的组。

本章主要讲解文本分类的主要技术，第十三章会对词性标注的内容详细讲解。

可以说在分析机器学习的数据源中最常见的知识发现主题是把数据对象或事件转换为预定的类别，再根据类别进行专门的处理，这是分类系统的基本任务。文本分类也如此：其实就是为用户给出的每个文档找到所属的正确类别（主题或概念）。

想要实现这个任务，首先需要给出一组类别，然后根据这些类别收集相应的文本集合，构成训练数据集，训练集既包括分好类的文本文件也包括类别信息。今天，在互联网的背景下自动化的文本分类被广泛的应用于，包括文本检索，垃圾邮件过滤，网页分层目录，自动生成元数据，题材检测，以及许多其他的应用领域，是文本挖掘最基础也是应用最广范的核心技术。

目前，有两种主要的文本分类方法，一是基于模式系统（通过运用知识工程技术），二是分类模型（通过使用统计和/或机器学习技术）。专家系统的方法是将专家的知识以规则表达式的形式编码成分类系统。机器学习的方法是一个广义归纳过程，采用由一组预分类的例子，通过训练建立分类。由于文件数量以指数速度的增加和知识专家的可用性变得越来越小，潮流趋势正在转向机器学习 - 基于自动分类技术。

2.2 文本分类项目

本节开始，我们和读者一起完成一个文本分类项目的全过程。虽然从分类算法层

面来看，各类语言的文本分类技术都大同小异。但从整个流程来考察，不同语言的文本处理所用到的技术还是有差别的。下面给出中文语言的文本分类技术和流程，主要包括以下几个步骤：

（1）预处理：去除文本的噪声信息，例如 HTML 标签，文本格式转换，检测句子边界等等；

（2）中文分词：使用中文分词器为文本分词，并去除停用词；

（3）构建词向量空间：统计文本词频，生成文本的词向量空间；

（4）权重策略--TF-IDF 方法：使用 TF-IDF 发现特征词，并抽取为反映文档主题的特征；

（5）分类器：使用算法训练分类器；

（6）评价分类结果：分类器的测试结果分析。

建议使用 Windows7 环境开发，我们的项目根路径 root 为“X:\WorkSpace\TextClassification\”，以下简称为 root。

2.2.1 文本预处理

因为文本处理的核心任务要把非结构化和半结构化的文本转化为结构化的形式，即向量空间模型。这之前，必须要对不同类型的文本进行预处理。在大多数文本挖掘任务中，文本预处理的步骤都是相似的，基本步骤如下：

1.选择处理的文本的范围。

对于一个实际的文本挖掘任务，文本范围是比较容易确定的。例如，电子邮件或呼叫记录可以把每封邮件自然转换为一个文本。但是，对于较长的文件需要决定是否使用整个文档或切分文档分成各节、段落或句子。选择适当的范围取决于文本挖掘任务的目标：对于分类或聚类的任务，往往把整个文档作为处理单位；对于情感分析，文档自动文摘，或信息检索，段落或章节可能更合适。

2.建立分类文本语料库。

文本分类中所说的文本语料一般分为两大类：

训练集语料：这是指已经分好类的文本资源，在很长一段时间内，中文文本分类的研究没有公开的数据集，无论从教学角度还是实用角度，使得分类算法都难以比较。目前比较好的中文分类语料库有：复旦大学谭松波中文文本分类语料(下载地址：http://www.threedweb.cn/thread-1292-1-1.html)和搜狗新闻分类语料库（下载地址：http://www.sogou.com/labs/dl/c.html）等

相对于搜狗新闻分类语料库而言，复旦大学的中文文本分类语料库小一些，但质量很高，是用于学习教育比较专业的语料。鉴于大多数读者都在个人电脑上进行测试，

我们这里截取部分的中文分类语料库作为文本资源。

中文文本分类语料下载地址：http://www.threedweb.cn/thread-1288-1-1.html

未分词训练语料库路径：Root\ train_corpus_small\；语料目录结构（如图 2.1）：

图 2.1　目录结构

未分词训练语料(以下称原始语料)一共包含 10 个子目录，目录名称为语料类别，该类所属训练文本就位于子目录中，以连续自然数编号：

本项目测试语料随机选自训练语料之中，测试语料库路径：Root\ test_corpus\

测试集语料：所谓测试集就是待分类的文本语料，可以是训练集的一部分，也可以是外部来源的文本语料。外部来源比较自由，一般实际项目都是解决新文本的分类问题。

待分类文本资源的获取方式很多，例如通过公司、图书馆甚至商业渠道，当然最为广泛的来源是互联网。如果不是特殊的用途，来自互联网的语料有很多好处：

- 语料种类繁多，规模庞大，能够抽取出绝大多数的相关信息和特征词汇，使用互联网语料制作的分类系统最具代表性；
- 免费，只要不是特别专业的，一般互联网上的网页资源和文本资源都可以免费下载；大规模文本抽取的成本比较低廉。
- 易于抽取，html 网页文本是未加密的数据文件和应用程序的综合体，可用任何一种文本编辑器直接打开，使用一些专用程序还可以执行脚本文件获取额外的数据资源。Html 文件可以保留源文本的所有格式，这与 word 和 pdf 不同。保留的文本格式对基于模式的文本分类会收到意想不到的效果。

一般批量获取网络文本需要使用网络爬虫下载，这方面的技术已经比较成熟，Java 和 Python 都可提供一整套的解决方案。因这部分超出本书的范围，就不做详细介绍。

3.文本格式转换

不同格式的文本不论何种处理形式都要统一转换为纯文本文件，例如，网页文本，word 或 pdf 文件都要转换为纯文本格式。

以网页文本为例，由于下载后的网页保留了全部的格式信息，首先要观察，格式信息对于文本挖掘是否有帮助。因为无论我们的任务是分类、聚类还是信息抽取，其基本工作都是想办法从文本中发现知识，而所有的知识都是一种结构化的信息。而有些文本，例如 html，其表格<table></table>内的信息一般是结构化的。对于基于机器学习的分类系统没什么用，但是对于基于模式的系统却很有价值，如果此类系统参与分类，在去除 html 标签时，应该保留表格，或从中抽取出表格，作为文档的辅助分类依据。

过滤掉这些有意义的标签之后，就要去除 html 的其余所有标签，将文本转换为 txt 格式或 xml 格式的半结构化文本。笔者曾经处理过几千万的 html 文件，程序在执行文档格式转换时，最容易出错的地方不是字符集的不同：Python 可以通过 Encode，Decode，Unicode 等指令，自动统一将文档编码转换为指定的编码格式：UTF-8 或 GBK。而主要问题是乱码：一些文档在发布后，由于编辑问题或网络传输的问题，文档中常出现乱码。这类情况如果处理不好，很容易造成程序异常。

为了提高性能，一般 Python 去除 html 标签，较多的使用 lxml 库，是一个 C 编写的 xml 扩展库，比使用 re 正则表达式库的标签去除方式性能高很多，适用于海量的网络文本格式转换。

下面给出样例代码：

```
from lxml import etree,html

# htm 文件路径，以及读取文件
path = "1.htm"
content = open(path,"rb").read()
page = html.document_fromstring(content) # 解析文件
text = page.text_content() # 去除所有标签
print text    # 输出去除标签后解析结果
```

4.检测句子边界：标记句子的结束。

句子边界检测是分解整个文档，并转换成单独句子的过程。对于中文文本，它就是寻找像“。”“？”或“！”等标点符号，作为断句的依据。然而，随着英语的普及，也有使用“.”作为句子的结束标志。这容易与某些词语出现的缩写或缩写词的一部分混淆。如果从这里断句，容易发生错误。这种情况可以使用一些简单的启发式规

则或统计分类技术，正确识别大多数句子边界。

最奇葩的现象是，某些网站使用图片作为标点符号。图片的样式以及所占的空间与实际标点符号完全一样，如果不下载文本，肉眼几乎看不出差别。这种文档下载后，必须用字符型的标点符号替换掉图片式的标点符号。

2.2.2 中文分词介绍

中文分词 (Chinese Word Segmentation) 指的是将一个汉字序列（句子）切分成一个一个单独的词。分词就是将连续的字序列按照一定的规范重新组合成词序列的过程。我们知道，在英文的行文中，单词之间是以空格作为自然分界符的，而中文只是字、句和段能通过明显的分界符来简单划界，唯独词没有一个形式上的分界符，虽然英文也同样存在短语的划分问题，不过在词这一层上，中文比之英文要复杂的多、困难的多。中文分词，不仅是中文文本分类的一大问题，也是中文自然语言处理的核心问题之一。

在一定意义上，中文分词不完全是个技术问题。中文分词的难点也不完全是算法问题。因为关于到底什么是词这个概念在中国理论界已经争论了很多年。一个著名的例子就是：“鸡蛋”、“牛肉”、“下雨”是词吗，如果是那么“鸭蛋”、“驴肉”、“下雪”、“鸟蛋”、“鱼肉”、“下雾”也应该是词，按照这样规则组合下去会产生很多让人费解的结论。如果不是，这些字符串在我们日常生活中使用的频率非常高，而且都有独立的意义。

一般像这种最后谁也想不明白的问题，最后都交给概率论。原因在第一章已经讲得很清楚，这里就不再重复了。最终完全解决中文分词的算法是基于概率图模型的条件随机场(CRF)。这个算法由 Lafferty 等人于 2001 年提出，估计当时设计这个算法不是为了解决中文分词问题。这件事对中文自然语言处理而言是件大事情，但是…

闲话不讲了，我们回到正题。分词是自然语言处理中最基本、最底层的模块，分词精度对后续应用模块影响很大，纵观整个自然语言处理领域，文本或句子的结构化表示是语言处理最核心的任务。目前，文本的结构化表示简单分为四大类：词向量空间模型、主题模型、依存句法的树表示、RDF 的图表示。以上这四种文本表示都以分词为基础的。

一般专业化大型的文本分类系统，为了提高精度，常常订制开发自己的分词系统。一般算法都使用 CRF，语料资源则各有不同。现在开放出来的、比较成熟的、有商用价值的分词工具有：理工大学张华平博士开发的中文分词系统：http://ictclas.nlpir.org/（一年免费试用权）；哈工大的语言云系统：http://www.ltp-cloud.com/intro/(开源)，Ansj 的中文分词系统：http://www.nlpcn.org/（开源）等等。这类分词系统完整性强，稳定

性好、精度也不错。

即使如此也不能满足搜索引擎类公司对超大规模分词的需求，一般像新浪和百度这些公司的分词系统，仅扩展词汇要到五百多万，估计基础词汇不少于五十万。

以上这些分词系统与 Python 整合都比较麻烦，占用资源也大。因为分词不是本书讲解的重点，为了方便，我们尽量使用小巧高效，而且原生支持 Python 的分词系统。这里我们推荐使用 jieba 分词，它是专门使用 Python 语言开发的分词系统，占用资源较小，常识类文档的分词精度较高。对于非专业文档绰绰有余。

Jieba 分词已经作为 Python 的官方外部库，上传到 pypi 上，可以通过 pip 直接下载使用：

```
C:\python64\scripts\pip install jieba
```

下载安装最新版。本书使用的是 jieba-0.36.2 版，分词算法使用的是 CRF，编写语言的是 Python，基础词库 349046 个。

下面给出 jieba 分词简单的样例代码：

```
# -*- coding: utf-8 -*-

import sys
import os
import jieba

# 设置 utf-8 unicode 环境
reload(sys)
sys.setdefaultencoding('utf-8')

seg_list = jieba.cut("小明 1995 年毕业于北京清华大学", cut_all=False)
print "Default Mode:", " ".join(seg_list)  # 默认切分

seg_list = jieba.cut("小明 1995 年毕业于北京清华大学")
print "    ".join(seg_list)

seg_list = jieba.cut("小明 1995 年毕业于北京清华大学", cut_all=True)
print "Full Mode:", "/ ".join(seg_list)  # 全切分

seg_list = jieba.cut_for_search("小明硕士毕业于中国科学院计算所，后在日本京都大学深造")  # 搜索引擎模式
print "/    ".join(seg_list)
```

输出结果：（省略了无关的提示信息）

```
Default Mode: 小明 1995 年 毕业 于 北京 清华大学
小明  1995  年  毕业  于  北京  清华大学
Full Mode: 小/ 明/ 1995/ 年/ 毕业/ 于/ 北京/ 清华/ 清华大学/ 华大/ 大学
```

```
小明/ 硕士/ 毕业/ 于/ 中国/ 科学/ 学院/ 科学院/ 中国科学院/ 计算/ 计算所/ ，/ 后/
在/ 日本/ 京都/ 大学/ 日本京都大学/ 深造
```

Jieba 分词支持的分词模式包括：默认切分、全切分、搜索引擎切分几种。应用范围比较广。中文文本分词之后，连续的字序列就变成了以词为单位的向量。向量中的每个分量都是一个有独立意义的词。通过分词中文文本实现了最基础的结构化。

本项目中创建分词后语料路径：Root\ train _corpus_seg\

1.设置字符集，并导入 jieba 分词包

```
# -*- coding: utf-8 -*-

import sys
import os
import jieba

# 配置 utf-8 输出环境
reload(sys)
sys.setdefaultencoding('utf-8')
```

2.定义创建两个函数，用于处理读取和保存文件：

```
def savefile(savepath,content): # 保存至文件
    fp = open(savepath,"wb")
    fp.write(content)
    fp.close()

def readfile(path):   # 读取文件
    fp = open(path,"rb")
    content = fp.read()
    fp.close()
    return content
```

3.接上面的部分，以下是整个语料库的分词主程序：

```
corpus_path = "train_corpus_small/"  # 未分词分类语料库路径
seg_path = "train_corpus_seg/"       # 分词后分类语料库路径

catelist = os.listdir(corpus_path)  # 获取 corpus_path 下的所有子目录

# 获取每个目录下所有的文件
for mydir in catelist:
    class_path = corpus_path+mydir+"/"      # 拼出分类子目录的路径
    seg_dir = seg_path+mydir+"/"            # 拼出分词后语料分类目录
    if not os.path.exists(seg_dir):         # 是否存在目录，如果没有创建
            os.makedirs(seg_dir)
    file_list = os.listdir(class_path)      # 获取类别目录下的所有文件
```

```
    for file_path in file_list:                          # 遍历类别目录下文件
        fullname = class_path + file_path    # 拼出文件名全路径
        content = readfile(fullname).strip()        # 读取文件内容
        content = content.replace("\r\n","").strip() # 删除换行和多余的空格
        content_seg = jieba.cut(content)     # 为文件内容分词
        savefile(seg_dir+file_path," ".join(content_seg))   # 将处理后的文件保存到分词后语料
目录

print "中文语料分词结束！！！"
```

简单吧，区区几行代码就完成了整个语料库的全部分词任务。最后，我们看一下分词前后的结果：

分词前：292.txt

第七届全国美展中国画获奖作者新作展在京开幕
新华社北京5月22日电（记者邵建武）第七届全国
美展中国画获奖作者新作展今天在京开幕。
26位曾经在第七届全国美展中获奖的中国画家展出
的近百幅新作，较为集中地反映了他们的创作状况与新的
探索，其中有写意人物与山水，也有工笔人物与花鸟，或
洒脱，或凝重，或纵逸，或拙雅，风姿各异。
这次展览是由中国美术家协会、香港深圳博雅艺术公
司和北京艺术博物馆联合举办的。（完）

分词后：292.txt

第七届 全国 美展 中国画 获奖 作者 新作 展 在京开幕 新华社 北京 5月 22 日电 （ 记者 邵建武 ） 第七届 全国 美展 中国画 获奖 作者 新作展 今天 在京开幕 。 26 位 曾经 在 第七届 全国 美展 中 获奖 的 中国 画家 展出 的 近 百幅 新作 ， 较为 集中地 反映 了 他们 的 创作 状况 与 新 的 探索 ， 其中 有 写意 人物 与 山水 ， 也 有 工笔 人物 与 花鸟 ， 或 洒脱 ， 或 凝重 ， 或 纵逸 ， 或 拙雅 ， 风姿 各异 。 这次 展览 是 由 中国美术家协会 、 香港 深圳 博雅 艺术 公司 和 北京 艺术 博物馆 联合 举办 的 。 （ 完 ）

当然，在实际应用中，为了后续生成向量空间模型的方便，这些分词后的文本信息还要转换为文本向量信息并对象化。这里需要引入一个 Scikit-Learning 库的 Bunch 数据结构：

```
from sklearn.datasets.base import Bunch   # 导入 Bunch 类

# Bunch 类提供一种 key,value 的对象形式
# target_name:所有分类集名称列表
```

```
# label:每个文件的分类标签列表
# filenames:文件路径
# contents:分词后文件词向量形式
bunch = Bunch(target_name=[],label=[],filenames=[],contents=[])
```

将分好词的文本文件转换并持久化为 Bunch 类形式代码如下：

```
wordbag_path = "train_word_bag/train_set.dat"   # 分词语料 Bunch 对象持久化文件路径
seg_path = "train_corpus_seg/"          #  分词后分类语料库路径

catelist = os.listdir(seg_path)
bunch.target_name.extend(catelist) # 将类别信息保存到 bunch 对象中
for mydir in catelist:
    class_path = seg_path+mydir+"/"
    file_list = os.listdir(class_path)
    for file_path in file_list:
        fullname = class_path + file_path
        bunch.label.append(mydir)    #   保存当前文件的分类标签
        bunch.filenames.append(fullname)   # 保存当前文件的文件路径
        bunch.contents.append(readfile(fullname).strip())   # 保存文件词向量

#bunch 对象持久化
file_obj = open(wordbag_path, "wb")
pickle.dump(bunch,file_obj)
file_obj.close()

print "构建文本对象结束！！！"
```

这样在目录下生成了一个 train_set.dat 文件。此文件保存所有训练集文件的所有分类信息，以及每个文件的文件名，文件所属分类和词向量。

2.2.3 Scikit-Learn 库简介

后面的部分，将使用 Scikit-Learn 库中的算法模块处理生成的训练集文本向量，在讲解之前，我们先简单介绍一下 Scikit-Learn 算法库。这是有关了解机器学习各类算法很重要的资源，本来想把这部分放到第一章来讲，但是考虑篇幅问题，最终还是移到这里。

Scikit-Learn 网站的主页截图(http://scikit-learn.org/stable/)。Scikit-Learn 是一个用于机器学习的 Python 库，建立在 SciPy 基础之上，获得 3-Clause BSD 开源许可证。这个网站是由 David Cournapeau 在 2007 年发起的一个 Google Summer of Code 项目，从那时起这个项目就已经拥有很多的贡献者了，而且目前该网站也是由一志愿者团队

在维护着（如图 2.2）。

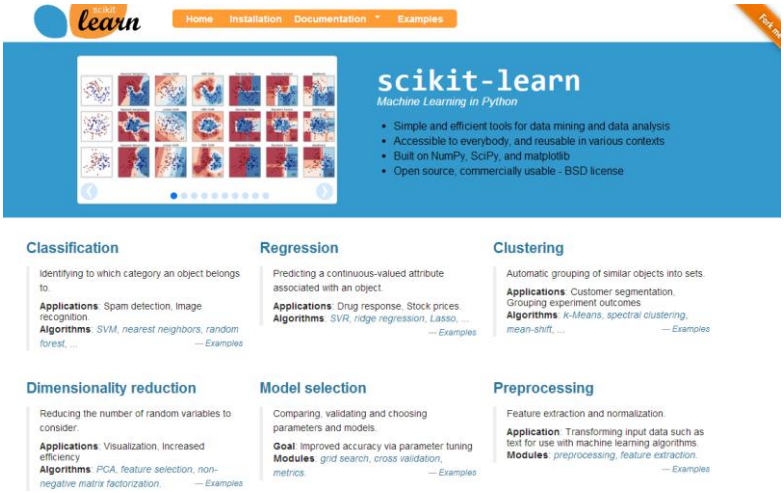


图 2.2　Scikit-Learn 主页

模块分类：

- ❑ 分类和回归算法：广义线性模型，线性和二次判别分析，岭回归，支持向量机，随机梯度下降，KNN，高斯过程，交叉分解，朴素贝叶斯，决策树，集成方法，多细粒度的算法，特征选择，半监督，保序回归，概率校准；
- ❑ 聚类算法：K-means，仿射传播，均值漂移，谱聚类，分层聚类，DBSCAN，Birch；
- ❑ 维度约简：PCA，潜在语义分析（截断奇异值分解），字典学习，因子分析，ICA，非负矩阵分解；
- ❑ 模型选择：交叉验证，评价估计性能，网格搜索：搜索参数估计，模型的预测质量的量化评价，模型的持久化，验证曲线：绘制分数评价模型；
- ❑ 数据预处理：标准化，去除均值率和方差缩放，正规化，二值化，编码分类特征，缺失值的插补

主要特点：

- ❑ 操作简单、高效的数据挖掘和数据分析
- ❑ 无访问限制，在任何情况下可重新使用
- ❑ 建立在 NumPy、SciPy 和 matplotlib 基础上
- ❑ 使用商业开源协议——BSD 许可证

重要链接：

- ❑ 官方源代码报告：https://github.com/scikit-learn/scikit-learn
- ❑ HTML 文档（稳定本）：http://scikit-learn.org

- HTML 文档（开发版本）：http://scikit-learn.org/dev/
- 下载版本：http://sourceforge.net/projects/scikit-learn/files/
- 问题跟踪：https://github.com/scikit-learn/scikit-learn/issues
- 邮箱列表：https://lists.sourceforge.net/lists/listinfo/scikit-learn-general

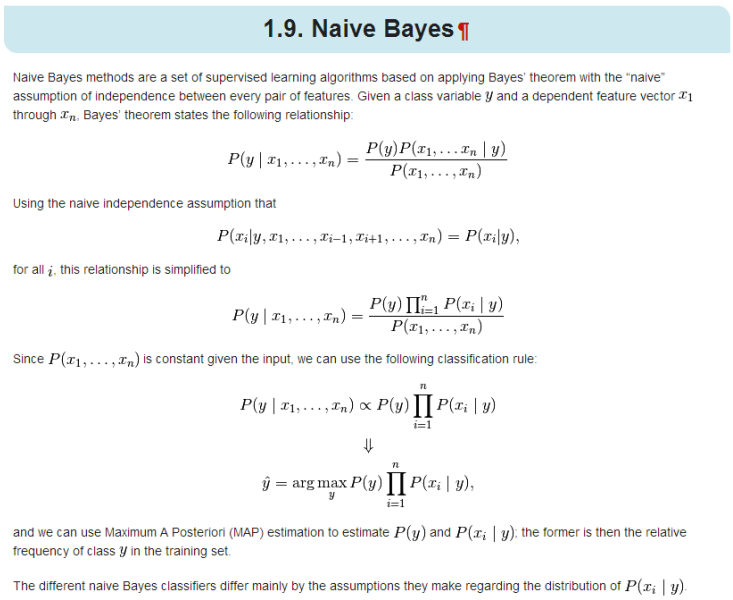
相关性

- scikit-learn 经过测试之后可以运行在 Python 2.6、Python 2.7 和 Python 3.4 平台上。除此之外，它还要适应运行在 Python 3.3 平台上。

以上部分来自是官方介绍的中文翻译。本书中代码分为两大部分：原理级别的算法模块和注释由本书作者提供；大多数功能级别的算法模块都来自 Scikit-Learn，诸如本章的朴素贝叶斯，KNN，以及后面的支持向量机等等。

之所以我们专门选取一个小节来介绍 Scikit-Learn，是因为这个网站是学习和应用机器学习算法的最重要的工具之一。以朴素贝叶斯算法为例，网站提供了一整套算法学习的教程资源和源代码，网址： http://scikit-learn.org/stable/modules/naive_bayes.html：

- 首先，网站提供了算法的数学公式以及简单的推导过程（如图 2.3）：



1.9. Naive Bayes¶

Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of independence between every pair of features. Given a class variable y and a dependent feature vector x_1 through x_n, Bayes' theorem states the following relationship:

$$P(y \mid x_1, \dots, x_n) = \frac{P(y)P(x_1, \dots x_n \mid y)}{P(x_1, \dots, x_n)}$$

Using the naive independence assumption that

$$P(x_i|y, x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n) = P(x_i|y),$$

for all i, this relationship is simplified to

$$P(y \mid x_1, \dots, x_n) = \frac{P(y) \prod_{i=1}^{n} P(x_i \mid y)}{P(x_1, \dots, x_n)}$$

Since $P(x_1, \dots, x_n)$ is constant given the input, we can use the following classification rule:

$$P(y \mid x_1, \dots, x_n) \propto P(y) \prod_{i=1}^{n} P(x_i \mid y)$$

$$\Downarrow$$

$$\hat{y} = \arg\max_y P(y) \prod_{i=1}^{n} P(x_i \mid y),$$

and we can use Maximum A Posteriori (MAP) estimation to estimate $P(y)$ and $P(x_i \mid y)$; the former is then the relative frequency of class y in the training set.

The different naive Bayes classifiers differ mainly by the assumptions they make regarding the distribution of $P(x_i \mid y)$.

图 2.3　朴素贝叶斯公式的推导过程

- 接下来，提供了算法应用于不同情况的几个变种（如图 2.4）：

1.9.1. Gaussian Naive Bayes ¶

GaussianNB implements the Gaussian Naive Bayes algorithm for classification. The likelihood of the features is assumed to be Gaussian:

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right)$$

The parameters σ_y and μ_y are estimated using maximum likelihood.

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> from sklearn.naive_bayes import GaussianNB
>>> gnb = GaussianNB()
>>> y_pred = gnb.fit(iris.data, iris.target).predict(iris.data)
>>> print("Number of mislabeled points out of a total %d points : %d"
...       % (iris.data.shape[0],(iris.target != y_pred).sum()))
Number of mislabeled points out of a total 150 points : 6
```

图 2.4　高斯朴素贝叶斯算法

如图 2.3 所示，不仅提供了变种的说明，还提供了例子程序。

❑　第三，提供了变种类(高斯朴素贝叶斯)的参数说明（如图 2.5/2.6）：

http://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html#sklearn.naive_bayes.GaussianNB

Attributes:

class_prior_ : array, shape (n_classes,)

probability of each class.

class_count_ : array, shape (n_classes,)

number of training samples observed in each class.

theta_ : array, shape (n_classes, n_features)

mean of each feature per class

sigma_ : array, shape (n_classes, n_features)

variance of each feature per class

图 2.5　高斯朴素贝叶斯属性说明

Methods

fit(X, y)	Fit Gaussian Naive Bayes according to X, y
get_params([deep])	Get parameters for this estimator.
partial_fit(X, y[, classes])	Incremental fit on a batch of samples.
predict(X)	Perform classification on an array of test vectors X.
predict_log_proba(X)	Return log-probability estimates for the test vector X.
predict_proba(X)	Return probability estimates for the test vector X.
score(X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
set_params(**params)	Set the parameters of this estimator.

图 2.6　高斯朴素贝叶斯方法说明

- 第四，还提供了该算法的一些应用实例（如图 2.7）：

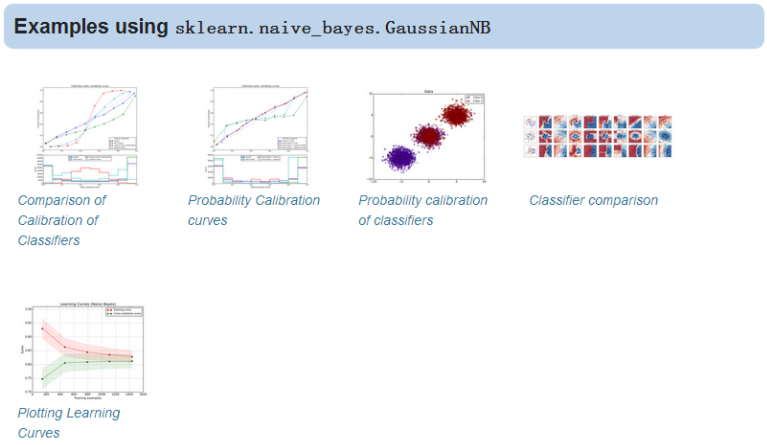


图 2.7　高斯朴素贝叶斯实例

- 最后，可从这里 https://github.com/scikit-learn/scikit-learn 下载整个项目的源代码

通过教程的学习，读者可以对各类算法有一个全面的了解。并通过案例学习加深对各类算法的认识，便于应用到实际项目之中去。

2.2.4　向量空间模型

文本分类的结构化方法就是向量空间模型，虽然越来越多的实践已经证明，这种模型存在着的局限，但是迄今为止，它仍是在文本分类中应用最广泛、最为流行的数据结构，也是很多相关技术的基础，例如：推荐系统、搜索引擎等。

向量空间模型把文本表示为一个向量，其中该向量的每个特征表示为文本中出现的词。通常，把训练集中出现的每个不同的字符串都作为一个维度，包括常用词、专有词、词组和其他类型模式串，如电子邮件地址和 URL。目前，大多数文本挖掘系统，都把文本存储为向量空间的表示，因为它便于运用机器学习算法。这类算法适用并能有效处理高维空间的文本情况。但是，对于大规模文本分类，这会导致极高维的空间，即使是中等大小的文本文件集合，向量的维度也很轻易就达到数十万维。

由于文本在存储为向量空间时，维度比较高。为节省存储空间和提高搜索效率，在文本分类之前会自动过滤掉某些字或词，这些字或词即被称为停用词。这类词一般都是意义模糊的常用词，还有一些语气助词，通常它们对文本起不了分类特征的意义。

这些停用词都是人工输入、非自动化生成的，生成后的停用词会形成一个停用词表。各类停用词表大同小异，读者可以从这个网址下载停用词表：http://www.threedweb.cn/thread-1294-1-1.html。

读取停用词列表：

```
# 读取文件
# 1. 读取停用词表
stopword_path = "train_word_bag/hlt_stop_words.txt"
stpwrdlst = readfile(stopword_path).splitlines()
```

2.2.5 权重策略：TF-IDF 方法

对于已经看过第一章的读者，对向量空间模型应该并不陌生，也就是前文所说的词袋模型，它将文本中的词和模式串转换为数字，而整个文本集也都转换为维度相等的词向量矩阵。假设文本仍旧是第一章的三个文本：

文本 1：My dog ate my homework。

文本 2：My cat ate the sandwich。

文本 3：A dolphin ate the homework。

生成的词袋中不重复的词还是 9 个，注意，这里增加了词频信息：

a (1), ate (3), cat (1), dolphin (1), dog (1), homework (2), my (3), sandwich (1), the (2).直观上，文本的词向量表示可以使用二元表示：

文本 1：0,1,0,0,1,1,1,0,0（注意，尽管“my”出现了两次，但二元向量表示中仍然是“1”）

文本 2：0,1,1,0,0,0,1,1,1

文本 3：1,1,0,1,0,1,0,0,1

这种方式的问题是忽略了一个句子中出现多个相同词的词频信息，我们增加这些词频信息，就变成了整型计数方式，下表是使用整形计数方式的词向量表示：

文本 1：0,1,0,0,1,1,2,0,0（请注意，“my”在句子中出现的两次）

文本 2：0,1,1,0,0,0,1,1,1

文本 3：1,1,0,1,0,1,0,0,1

接下来，我们对整型计数方式进行归一化，归一化可以避免句子长度不一致问题，便于算法计算，而且对于基于概率算法，词频信息就变为了概率分布，这就是文档的 TF 信息：

文本 1：0,1/5,0,0,1/5,1/5,2/5,0,0（请注意，“my”在句子中出现的两次）

文本 2：0,1/5,1/5,0,0,0,1/5,1/5,1/5

文本 3：1/5,1/5,0,1/5,0,1/5,0,0,1/5

但是这里还有个问题，如何体现生成的词袋中的词频信息呢？

原信息：a (1), ate (3), cat (1), dolphin (1), dog (1), homework (2), my (3), sandwich (1), the (2).注意：由于词袋收集了所有文档中的词，这些词的词频是针对所有文档的词频，因此，词袋的统计基数是文档数

词条的文档频率: a (1/3), ate (3/3), cat (1/3), dolphin (1/3), dog (1/3), homework (2/3), my (2/3), sandwich (1/3), the (2/3)

词袋模型的 IDF 权重：

IDF: a log(3/1), ate log(3/3), cat log(3/1), dolphin log(3/1), dog log(3/1), homework log (3/2), my log (3/2), sandwich log (3/1), the log (3/2)

TF-IDF 权重策略：

计算文本的权重向量，应该选择一个有效的权重方案。最流行的方案是对 TF-IDF 权重的方法。TF-IDF 的含义是词频--逆文档频率，其含义是如果某个词或短语在一篇文章中出现的频率 TF 高，并且在其他文章中很少出现，则认为此词或者短语具有很好的类别区分能力，适合用来分类。在前面的例子中，在文本 1 的词频“my”是 2，因为它在文档中出现了两次。而在这三个文件集合中，“my”的词条的文档频率也是 2。TF-IDF 的假设是，高频率词应该具有高权重，除非它也是高文档频率。“my”这个词在文本中是最经常出现的词汇之一。它不仅很多次发生在单一的文本中，但几乎也发生在每个文档中。逆文档频率就是使用词条的文档频率来抵消该词的词频对权重的影响，而得到一个较低的权重。

下面我们给出定义和公式(来自百度百科)：

词频（term frequency，TF）指的是某一个给定的词语在该文件中出现的频率。这个数字是对词数(term count)的归一化，以防止它偏向长的文件。对于在某一特定文件里的词语来说，它的重要性可表示为：

$$\mathrm{TF_{ij}} = \frac{n_{i,j}}{\sum_{k} n_{\mathrm{k},j}}$$

以上式子中分子是该词在文件中的出现次数，而分母则是在文件中所有字词的出现次数之和。

逆向文件频率（inverse document frequency，IDF）是一个词语普遍重要性的度量。某一特定词语的 IDF，可以由总文件数目除以包含该词语之文件的数目，再将得到的商取对数得到：

$$\mathrm{IDF_i} = \log \frac{|\mathrm{D}|}{|\{\mathrm{j}: \mathrm{t}_i \in \mathrm{d}_j\}|}$$

其中

|D|：语料库中的文件总数；

j：包含词语的文件数目，如果该词语不在语料库中，就会导致分母为零，因此一般情况下使用 1+|{d∈D:t∈d}|作为分母；

然后再计算 TF 与 IDF 的乘积。

某一特定文件内的高词语频率，以及该词语在整个文件集合中的低文件频率，TFIDFij=TFijI×IDFij 可以产生出高权重的 TF-IDF。因此，TF-IDF 倾向于过滤掉常见的词语，保留重要的词语。

最后介绍代码的实现：

1．导入所需的 Scikit-Learn 包：

```
# -*- coding: utf-8 -*-

import sys
import os
from sklearn.datasets.base import Bunch   # 引入 Bunch 类
import cPickle as pickle                  # 引入持久化类
from sklearn import feature_extraction
from sklearn.feature_extraction.text import TfidfTransformer   # Tfidf 向量转换类
from sklearn.feature_extraction.text import TfidfVectorizer    # Tfidf 向量生成类

# 配置 utf-8 输出环境
reload(sys)
sys.setdefaultencoding('utf-8')
```

2.读取和写入 bunch 对象的函数：

```
#读取 bunch 对象
def readbunchobj(path):
    file_obj = open(path, "rb")
    bunch = pickle.load(file_obj)
    file_obj.close()
    return bunch

#写入 bunch 对象
def writebunchobj(path,bunchobj):
    file_obj = open(path, "wb")
    pickle.dump(bunchobj,file_obj)
    file_obj.close()
```

3.从训练集生成 tfidf 向量词袋：

```
# 2. 导入分词后的词向量 bunch 对象
path = "train_word_bag/train_set.dat"          # 词向量空间保存路径
bunch     =   readbunchobj(path)
```

```
# 3. 构建 tf-idf 词向量空间对象
tfidfspace                                                                    =
Bunch(target_name=bunch.target_name,label=bunch.label,filenames=bunch.filenames,    tdm=[],
vocabulary={})

# 4. 使用 TfidfVectorizer 初始化向量空间模型
vectorizer = TfidfVectorizer(stop_words=stpwrdlst,sublinear_tf = True,max_df = 0.5)
transformer=TfidfTransformer()    # 该类会统计每个词语的 tf-idf 权值
# 文本转为词频矩阵,单独保存字典文件
tfidfspace.tdm = vectorizer.fit_transform(bunch.contents)
tfidfspace.vocabulary= vectorizer.vocabulary_
```

4.持久化 tfidf 向量词袋：

```
# 5. 创建词袋的持久化
space_path = "train_word_bag/tfdifspace.dat"          # 词向量词袋保存路径
writebunchobj(space_path,tfidfspace)
```

2.2.6 使用朴素贝叶斯分类模块

最常用的文本分类方法有 kNN 最近邻算法，朴素贝叶斯算法和支持向量机算法。这三类算法一般而言 kNN 最近邻算法的原理最简单，分类精度尚可，但是速度最慢；朴素贝叶斯对于短文本分类的效果最好，精度很高；支持向量机的优势是支持线性不可分的情况，精度上取中，有关支持向量机的算法参见后面章节的相关案例。

本节选择 Scikit-Learn 的朴素贝叶斯算法进行文本分类，测试集随机抽取自训练集中的文档集合，每个分类取 10 个文档，过滤掉 1k 以下的文档。

训练步骤与训练集相同，首先是分词，之后生成文件词向量文件，直至生成词向量模型。**不同的是在训练词向量模型时，需要加载训练集词袋，将测试集产生的词向量映射到训练集词袋的词典中，生成向量空间模型。**

代码如下：

```
# 2. 导入分词后的词向量 bunch 对象
path = "test_word_bag/test_set.dat"          # 词向量空间保存路径
bunch     = readbunchobj(path)

# 3. 构建测试集 tfidf 向量空间
testspace                                                                    =
Bunch(target_name=bunch.target_name,label=bunch.label,filenames=bunch.filenames,tdm=[],voc
abulary={})
```

```
# 4. 导入训练集的词袋
trainbunch = readbunchobj("train_word_bag/tfdifspace.dat")

# 5. 使用 TfidfVectorizer 初始化向量空间模型
vectorizer = TfidfVectorizer(stop_words=stpwrdlst,sublinear_tf = True,max_df = 0.5,vocabulary=trainbunch.vocabulary)  # 使用训练集词带向量
transformer=TfidfTransformer()
testspace.tdm = vectorizer.fit_transform(bunch.contents)
testspace.vocabulary = trainbunch.vocabulary

# 创建词袋的持久化
space_path = "test_word_bag/testspace.dat"          # 词向量空间保存路径
writebunchobj(space_path,testspace)
```

执行多项式贝叶斯算法进行测试文本分类，并返回分类精度：

1.导入多项式贝叶斯算法包：

```
from sklearn.naive_bayes import MultinomialNB #导入多项式贝叶斯算法
```

2.执行预测

```
# 导入训练集向量空间
trainpath = "train_word_bag/tfdifspace.dat"
train_set = readbunchobj(trainpath)

# 导入测试集向量空间
testpath = "test_word_bag/testspace.dat"
test_set = readbunchobj(testpath)

# 应用朴素贝叶斯算法
# alpha:0.001 alpha 越小，迭代次数越多，精度越高
clf = MultinomialNB(alpha = 0.001).fit(train_set.tdm, train_set.label)

# 预测分类结果
predicted = clf.predict(test_set.tdm)
total = len(predicted);rate = 0
for flabel,file_name,expct_cate in zip(test_set.label,test_set.filenames,predicted):
    if flabel != expct_cate:
        rate += 1
        print file_name,": 实际类别:",flabel," -->预测类别:",expct_cate
# 精度
print "error rate:",float(rate)*100/float(total),"%"
```

3.预测精度：

```
test_corpus_seg/art/3143.txt :  实际类别: art   -->预测类别: education
```

```
error rate: 0.990099009901 %
```

需要指出的是，多项式贝叶斯算法的分类精度非常高，测试集实际分类的精度是 100%，出错的 3143.txt 文件是笔者从 education 复制到 art 下的。

2.2.7 分类结果评估

一般机器学习领域的算法评估有三个基本的指标：

召回率：(Recall Rate,也叫查全率)是检索出的相关文档数和文档库中所有的相关文档数的比率，衡量的是检索系统的查全率；

召回率(**R**ecall)=系统检索到的相关文件/系统所有相关的文件总数

准确率(**P**recision,也称为精度)：是检索出的相关文档数与检索出的文档总数的比率，衡量的是检索系统的查准率。

准确率(**P**recision) =系统检索到的相关文件/系统所有检索到的文件总数

注意：准确率和召回率是互相影响的，理想情况下肯定是做到两者都高，但是一般情况下准确率高、召回率就低，召回率高、准确率就低（如图 2.8）。

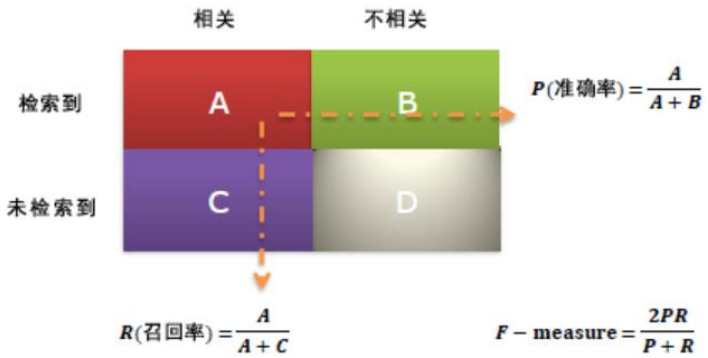


图 2.8　召回率与准确率的关系

Fβ-Measure(又称为 F-Score**)：**是机器学习领域的常用的一个评价标准，计算公式为：

$$F_\beta = \frac{(\beta^2+1)PR}{\beta^2 P + R}$$

其中 β 是参数，P 是准确率(Precision)，R 是召回率(Recall)。

F1-Measure：

当参数 β=1 时，就是最常见的 F1-Measure 了：

$$F_1 = \frac{2PR}{P+R}$$

文本分类项目的分类结果评估：

```
import numpy as np
from sklearn import metrics

# 定义分类精度函数：
def metrics_result(actual,predict):
    print '精度:{0:.3f}'.format(metrics.precision_score(actual,predict))
    print '召回:{0:0.3f}'.format(metrics.recall_score(actual,predict))
    print 'f1-score:{0:.3f}'.format(metrics.f1_score(actual,predict))

metrics_result(test_set.label,predicted)
```

输出结果：

```
精度:0.991
召回:0.990
f1-score:0.990
```

2.3 分类算法：朴素贝叶斯

前面一节，我们完成使用 Scikit-Learn 实现了中文文本分类的全过程，本节和后面一节，我们开始分析项目最核心的部分分类算法。本节主要讨论朴素贝叶斯算法的基本原理和简单的 Python 实现。

2.3.1 贝叶斯公式推导

经典数学理论的创立者们，诸如牛顿、高斯等，往往对所研究的领域抱有非常谨慎的态度，在提出一个学说或理论之前要从多个不同的角度考察这些理论的意义和正确性。现在看来，他们提出理论虽然简单，却常常揭示出事物最本质的规律，贝叶斯也是这样一位数学家。

这个简单的定理在第一章已经做了介绍：已知某条件概率，如何得到两个事件交换后的概率，也就是在已知 P(A|B)的情况下如何求得 P(B|A)：

$$P(B|A) = \frac{P(A|B)P(B)}{P(A)}$$

朴素贝叶斯分类是一种十分简单的分类算法，叫它朴素是因为其思想基础的简单

性：就文本分类而言，它认为词袋中的两两词之间的关系是相互独立的，即一个对象的特征向量中每个维度都是相互独立的。例如，黄色是苹果和梨共有的属性，但苹果和梨是相互独立的。这是朴素贝叶斯理论的思想基础。现在我们将它扩展到多维的情况：

朴素贝叶斯分类的正式定义如下：

1．设 $x=\{a_1,a_2,\ldots,a_m\}$为一个待分类项，而每个 a 为 x 的一个特征属性。

2．有类别集合 $C=\{y_1,y_2,\ldots,y_n\}$。

3．计算 $P(y_1|x), P(y_2|x),\ldots, P(y_n|x)$。

4．如果 $P(y_k|x)=\max\{P(y_1|x),P(y_2|x),\ldots,P(y_n|x)\}$，则 $x\in y_k$。

那么现在的关键就是如何计算第 3 步中的各个条件概率。我们可以这么做：

1．找到一个已知分类的待分类项集合，也就是训练集。

2．统计得到在各类别下各个特征属性的条件概率估计。即：

$P(a_1|y_1), P(a_2|y_1),\ldots, P(a_m|y_1)$;

$P(a_1|y_2), P(a_2|y_2),\ldots, P(a_m|y_2)$;

$P(a_m|y_n), P(a_m|y_n),\ldots, P(a_m|y_n)$。

3、如果各个特征属性是条件独立的(或者我们假设它们之间是相互独立的)，则根据贝叶斯定理有如下推导：

$$P(y_i|x)=\frac{P(x|y_i)P(y_i)}{P(x)}$$

因为分母对于所有类别为常数，只要将分子最大化皆可。又因为各特征属性是条件独立的，所以有：

$$P(x|y_i)P(y_i)=P(a_1|y_i)P(a_2|y_i)\ldots P(a_m|y_i)P(y_i)=P(y_i)\prod_{j=1}^{m}P(a_j|y_i)$$

这就是在 Scikit-Learn(http://scikit-learn.org/stable/modules/naive_bayes.html)中的公式推导过程的说明。

根据上述分析，朴素贝叶斯分类的流程可以表示如下：

第一阶段：训练数据生成训练样本集：TF-IDF

第二阶段：对每个类别计算 P(yi)

第三阶段：对每个特征属性计算所有划分的条件概率

第四阶段：对每个类别计算 $P(x|y_i)P(y_i)$

第五阶段：以 $P(x|y_i)P(y_i)$ 的最大项作为 x 的所属类别

2.3.2 朴素贝叶斯算法实现

我们首先创建名为 Nbayes_lib.py 的文件，这个文件用来编写导入的数据和朴素贝叶斯类的代码：

为了将主要精力都集中在算法本身，我们使用简单的英文语料作为数据集：

```
def loadDataSet():
    postingList=[['my', 'dog', 'has', 'flea', 'problems', 'help', 'please'],
                 ['maybe', 'not', 'take', 'him', 'to', 'dog', 'park', 'stupid'],
                 ['my', 'dalmation', 'is', 'so', 'cute', 'I', 'love', 'him','my'],
                 ['stop', 'posting', 'stupid', 'worthless', 'garbage'],
                 ['mr', 'licks', 'ate', 'my', 'steak', 'how', 'to', 'stop', 'him'],
                 ['quit', 'buying', 'worthless', 'dog', 'food', 'stupid']]
    classVec = [0,1,0,1,0,1]      #1 is abusive, 0 not
    return postingList,classVec
```

postList 是训练集文本，classVec 是每个文本对应的分类。

根据上节的步骤，逐步实现贝叶斯算法的全过程:

1.编写一个贝叶斯算法类，并创建默认的构造方法：

```
class NBayes(object):
    def __init__(self):
        self.vocabulary = [] # 词典
        self.idf=0                  # 词典的 idf 权值向量
        self.tf=0                   # 训练集的权值矩阵
        self.tdm=0                  # P(x|yi)
        self.Pcates = {}            # P(yi)--是个类别字典
        self.labels=[]              # 对应每个文本的分类，是个外部导入的列表
        self.doclength = 0          # 训练集文本数
        self.vocablen = 0           # 词典词长
        self.testset = 0            # 测试集
```

2.导入和训练数据集，生成算法必须的参数和数据结构：

```
    def train_set(self,trainset,classVec):
        self.cate_prob(classVec)    # 计算每个分类在数据集中的概率：P(yi)
        self.doclength = len(trainset)
        tempset = set()
        [tempset.add(word) for doc in trainset for word in doc] # 生成词典
        self.vocabulary = list(tempset)
        self.vocablen = len(self.vocabulary)
        self.calc_wordfreq(trainset)  # 计算词频数据集
        self.build_tdm()                  # 按分类累计向量空间的每维值：P(x|yi)
```

3.cate_prob 函数：计算在数据集中每个分类的概率：P(yi)

```
def cate_prob(self,classVec):
    self.labels = classVec
    labeltemps = set(self.labels)# 获取全部分类
    for labeltemp in labeltemps:
        # 统计列表中重复的分类：self.labels.count(labeltemp)
        self.Pcates[labeltemp] = float(self.labels.count(labeltemp))/float(len(self.labels))
```

4.calc_wordfreq 函数：生成普通的词频向量 ：

```
# 生成普通的词频向量
def calc_wordfreq(self,trainset):
    self.idf = np.zeros([1,self.vocablen])# 1*词典数
    self.tf = np.zeros([self.doclength,self.vocablen])# 训练集文件数*词典数
    for indx in xrange(self.doclength):     # 遍历所有的文本
        for word in trainset[indx]:             # 遍历文本中的每个词
            self.tf[indx,self.vocabulary.index(word)] +=1   # 找到文本的词在字典中的位
置+1
        for signleword in set(trainset[indx]):
            self.idf[0,self.vocabulary.index(signleword)] +=1
```

5.build_tdm 函数：按分类累计计算向量空间的每维值：P(x|yi)

```
#按分类累计向量空间的每维值：P(x|yi)
def build_tdm(self):
    self.tdm = np.zeros([len(self.Pcates),self.vocablen])#类别行*词典列
    sumlist = np.zeros([len(self.Pcates),1])  # 统计每个分类的总值
    for indx in xrange(self.doclength):
        self.tdm[self.labels[indx]] += self.tf[indx]   # 将同一类别的词向量空间值加总
        # 统计每个分类的总值--是个标量
        sumlist[self.labels[indx]]= np.sum(self.tdm[self.labels[indx]])
    self.tdm = self.tdm/sumlist   # 生成 P(x|yi)
```

6.map2vocab 函数：将测试集映射到当前词典

```
def map2vocab(self,testdata):
    self.testset = np.zeros([1,self.vocablen])
    for word in testdata:
        self.testset[0,self.vocabulary.index(word)] +=1
```

7.predict 函数：预测分类结果，输出预测的分类类别

```
def predict(self,testset):
    if np.shape(testset)[1] != self.vocablen:# 如果测试集长度与词典不相等，退出程序
        print "输入错误"
        exit(0)
    predvalue = 0       # 初始化类别概率
    predclass = ""      # 初始化类别名称
    for tdm_vect,keyclass in zip(self.tdm,self.Pcates):
```

```
            # P(x|yi) P(yi)
            temp = np.sum(testset*tdm_vect*self.Pcates[keyclass]) # 变量 tdm，计算最大分类
值
            if temp > predvalue:
                predvalue = temp
                predclass = keyclass
        return predclass
```

2.3.3 算法的改进

此算法的改进是为普通的词频向量使用 TF-IDF 策略，使之有能力修正多种偏差。

4.calc_tfidf 函数：以 tf-idf 方式生成向量空间：

```
# 生成 tf-idf
def calc_tfidf(self,trainset):
self.idf = np.zeros([1,self.vocablen])
self.tf = np.zeros([self.doclength,self.vocablen])
for indx in xrange(self.doclength):
    for word in trainset[indx]:
        self.tf[indx,self.vocabulary.index(word)] +=1
    # 消除不同句长导致的偏差
    self.tf[indx] = self.tf[indx]/float(len(trainset[indx]))
    for signleword in set(trainset[indx]):
        self.idf[0,self.vocabulary.index(signleword)] +=1
self.idf = np.log(float(self.doclength)/self.idf)
self.tf = np.multiply(self.tf,self.idf) # 矩阵与向量的点乘 tf x idf
```

2.3.4 评估分类结果

执行我们创建的朴素贝叶斯类，获取执行结果：

```
# -*- coding: utf-8 -*-

import sys
import os
from numpy import *
import numpy as np
from Nbayes_lib import *

dataSet,listClasses = loadDataSet()  # 导入外部数据集
# dataset: 句子的词向量，
```

```
# listClass 是句子所属的类别 [0,1,0,1,0,1]
nb = NBayes()                          # 实例化
nb.train_set(dataSet,listClasses)    # 训练数据集
nb.map2vocab(dataSet[0])             # 随机选择一个测试句
print nb.predict(nb.testset)          # 输出分类结果
```

分类结果：

```
1
```

2.4 分类算法:kNN

前面，我们使用朴素贝叶斯算法实现了文本分类。下面我们考虑通过计算向量间的距离衡量相似度来进行文本分类。这就是 kNN 算法，一种基于向量间相似度的分类算法。

2.4.1 kNN 算法原理

k 最近邻（k-Nearest Neighbor）算法是比较简单的机器学习算法。它采用测量不同特征值之间的距离方法进行分类。它的思想很简单：如果一个样本在特征空间中的 k 个最邻近（最相似）的样本中的大多数都属于某一个类别，则该样本也属于这个类别。第一个字母 k 可以小写，表示外部定义的近邻数量。这句话不难理解，但有点拗口，下面通过一个事例来讲解一下。

首先我们准备一个数据集，数据集很简单是二维空间上的四个点构成的一个矩阵（如表 2.1）：

表 2.1 数据集

分类	X	Y
A	1.0	1.1
A	1.0	1.0
B	0	0
B	0	1

其中前两个点构成一个类别 A，后两个点构成一个类别 B。

我们用 Python 把这四个点在坐标系中绘制出来：

1. 产生数据集的函数代码：

```
# -*- coding: utf-8 -*-

import sys
```

```
import os
from numpy import *
import numpy as np
import matplotlib.pyplot as plt
import operator

# 配置 utf-8 输出环境
reload(sys)
sys.setdefaultencoding('utf-8')
#
def createDataSet():
    group = array([[1.0,1.1],[1.0,1.0],[0,0],[0,0.1]])
    labels = ['A','A','B','B']
    return group, labels
```

2. 绘制图形的函数代码：

```
dataSet,labels = createDataSet()
# 绘图
fig = plt.figure()
ax = fig.add_subplot(111)
indx = 0
for point in dataSet:
    if labels[indx] == 'A' :
        ax.scatter(point[0],point[1],c='blue',marker='o',linewidths=0, s=300)
        plt.annotate("("+str(point[0])+","+str(point[1])+")",xy = (point[0],point[1]))
    else:
        ax.scatter(point[0],point[1],c='red',marker='^',linewidths=0, s=300)
        plt.annotate("("+str(point[0])+", "+str(point[1])+")",xy = (point[0],point[1]))
    indx += 1
plt.show()
```

从图形中可以清晰的看到由四个点构成的训练集。它被分为两个类别： A 类--蓝色圆圈、B 类--红色三角形。因为红色区域内的点距比它们到蓝色区域内的点距要小得多。这种分类也很自然（如图 2.9）。

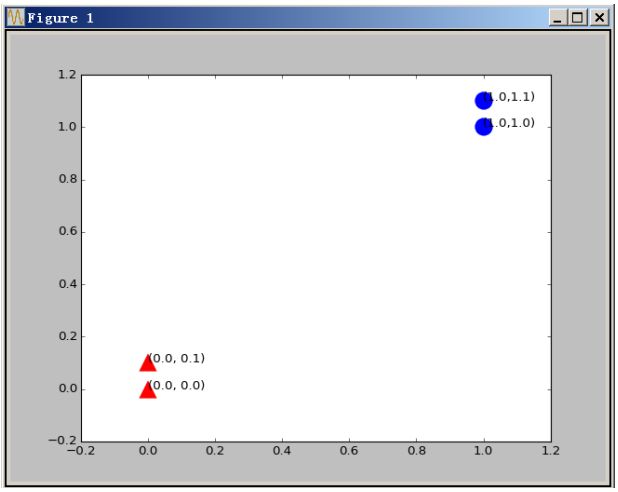


图 2.9　四个点构成的训练集

下面我们给出测试集，只有一个点，我们把它加入到刚才的矩阵中去（如表 2.2）：

表 2.2　测试集

分类	X	Y
A	1.0	1.1
A	1.0	1.0
B	0	0
B	0	1
?	0.2	0.2

我们想知道给出的这个测试集，它应该属于哪个分类呢？我想最简单的方法还是画图，我们把新加入的点加入图中。

3.　绘制测试集图形代码：

```
testdata=[0.2,0.2]    # 绘图
…
ax.scatter(testdata[0],testdata[1],c='green',marker='^',linewidths=0, s=300)
plt.annotate("("+str(testdata[0])+", "+str(testdata[1])+")",xy = (testdata[0],testdata[1]))
…
plt.show()
```

很清晰，从距离上看，它更接近红色三角的范围，应该归入 B 类（如图 2.10）。这就是 kNN 算法的基本原理。

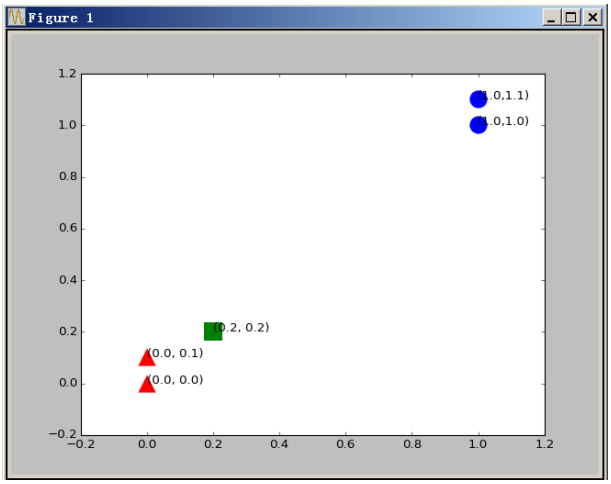


图 2.10　kNN 算法的基本原理

由此可见 kNN 算法应由以下步骤构成：

第一阶段：首先我们事先定下 k 值（就是指最近邻居的个数）。一般是个奇数，我们因为测试样本有限取 k 值为 3；

第二阶段：确定的距离度量公式—文本分类一般使用夹角余弦，得出待分类数据点和所有已知类别的样本点中，选择距离最近的 k 个样本。

夹角余弦：$\cos(\theta) = \dfrac{AB}{|\mathrm{A}||\mathrm{B}|}$

第三阶段：统计这 k 个样本点中，各个类别的数量。如上图，如果我们选定 k 值为 3，则 B 类样本（三角形）有 2 个，A 类样本（圆形）有 1 个，那么我们就把这个方形数据点定为 B 类；即，根据 k 个样本中，数量最多的样本是什么类别，我们就把这个数据点定为什么类别。

2.4.2　KNN 的 Python 实现

1.第一阶段：导入所需的库，数据的初始化

```
# -*- coding: utf-8 -*-

import sys
import os
from numpy import *
import numpy as np
import operator
```

```
from Nbayes_lib import*

# 配置 utf-8 输出环境
reload(sys)
sys.setdefaultencoding('utf-8')

k=3
```

2.第二阶段：实现夹角余弦的距离公式

```
# 夹角余弦距离公式
def cosdist(vector1,vector2):
    return dot(vector1,vector2)/(linalg.norm(vector1)*linalg.norm(vector2))
```

3.第三阶段：kNN 实现分类器：

```
# kNN 分类器
# 测试集：testdata；训练集：trainSet；类别标签：listClasses；k:k 个邻居数
def classify(testdata, trainSet, listClasses, k):
    dataSetSize = trainSet.shape[0]     # 返回样本集的行数
    distances = array(zeros(dataSetSize))
    for indx in xrange(dataSetSize):  # 计算测试集与训练集之间的距离：夹角余弦
     distances[indx] = cosdist(testdata,trainSet[indx])
    # 根据生成的夹角余弦按从大到小排序,结果为索引号
    sortedDistIndicies = argsort(-distances)
    classCount={}
    for i in range(k):            # 获取角度最小的前 k 项作为参考项
       # 按排序顺序返回样本集对应的类别标签
        votellabel = listClasses[sortedDistIndicies[i]]
        # 为字典 classCount 赋值,相同 key，其 value 加 1
         classCount[votellabel] = classCount.get(votellabel,0) + 1
    # 对分类字典 classCount 按 value 重新排序
    # sorted(data.iteritems(), key=operator.itemgetter(1), reverse=True)
    # 该句是按字典值排序的固定用法
    # classCount.iteritems()：字典迭代器函数
    # key：排序参数；operator.itemgetter(1)：多级排序
    sortedClassCount = sorted(classCount.iteritems(), key=operator.itemgetter(1), reverse=True)
    return sortedClassCount[0][0]     # 返回序最高的一项
```

2.4.3 评估分类结果

最后是使用 kNN 实现文本分类：

```
dataSet,listClasses = loadDataSet()
nb = NBayes()
```

```
nb.train_set(dataSet,listClasses)
# 使用之前贝叶斯分类阶段的数据集，以及生成的 tf 向量进行分类
print classify(nb.tf[3], nb.tf, listClasses, k)
```

结果：

```
1
```

准确率 100%

2.5 结语

本章通过中文文本分类的实例讲解了机器学习中的两个比较简单的算法：朴素贝叶斯算法和最近邻算法。

第二节我们详细介绍了中文文本分类的六个主要步骤：（1） 文本预处理；（2）中文分词；（3） 构建词向量空间；（4）权重策略--TF-IDF 方法；（5）朴素贝叶斯分类器；（6）评价分类结果。每个步骤都提供了完整的代码实现。因为是本书开始的章节，所以整个项目都是基于 scikit-learn 机器学习库实现的，因此，这些代码都可运用实际的工程项目，具有很强的实用性。

从第三节开始，我们进入算法讲解和源码实现部分。第三节重点讲解了朴素贝叶斯的数学推导，以及基本原理。之后，我们给出了算法的 Python 的代码实现，并且将 Python 实现算法用于实际的一个小样本的分类，最后评估了分类的结果。

第四节，我们使用 Python 实现了 kNN 算法的，并评估了分类的结果。

第三章　决策树的发展

决策树算法是最早的机器学习算法之一。早在 1966 年 Hunt,Marin 和 Stone 提出的 CLS 学习系统就有了决策树算法的概念。但到了 1979 年，J.R. Quinlan 才给出了 ID3 算法的原型，1983 年和 1986 年他对 ID3 算法进行了总结和简化，正式确立了决策树学习的理论。从机器学习的角度来看，这是决策树算法的起点。到 1986 年，Schlimmer 和 Fisher 在此基础上进行改造，引入了节点缓冲区，提出了 ID4 算法。在 1993 年，Quinlan 进一步发展了 ID3 算法，改进成 C4.5 算法，成为机器学习的十大算法之一。ID3 的另一个分支是分类回归决策树算法(Classification Regression Tree)，与 C4.5 不同的是，CART 的决策树主要用于预测，这样决策树理论完整地覆盖了机器学习中分类和回归两个领域了。

- 决策树的算法思想
- 信息熵与 ID3
- C4.5 算法
- Scikit-Learn 与回归树

3.1 决策树的基本思想

决策树的思想来源非常朴素，每个人大脑中都有类似 if-then 这样的判断逻辑，其中 if 表示条件，then 就是选择或决策。程序设计中，最基本的语句条件分支结构就是 if-then 结构。而最早的决策树就是利用这类结构分隔数据的一种分类学习方法。下面我们从一个实例开始讲解一下最简单的决策树的生成过程。

3.1.1 从一个实例开始

假定某间 IT 公司销售笔记本电脑产品，为了提高销售收入，公司对各类客户建立了统一的调查表，统计了几个月销售数据之后收集到（如表 3.1）中的数据，为了提高销售的效率，公司希望通过上表对潜在客户进行分类，并根据上述特征制作简单的销售问卷。以利于销售人员的工作。这就出现两个问题：

1. 如何对客户分类？
2. 如何根据分类的依据，并给出销售人员指导的意见？

表 3.1 销售调查表

计数	年龄	收入	学生	信誉	是否购买
64	青	高	否	良	不买
64	青	高	否	优	不买
128	中	高	否	良	买
60	老	中	否	良	买
64	老	低	是	良	买
64	老	低	是	优	不买
64	中	低	是	优	买
128	青	中	否	良	不买

64	青	低	是	良	买
132	老	中	是	良	买
64	青	中	是	优	买
32	中	中	否	优	买
32	中	高	是	良	买
64	老	中	否	优	不买

问题分析：

我们从第一列开始看这个表格，表格不大，一共 15 行，每行表示列特征取不同值时的统计人数，第一列是统计人数值；第二列是年龄特征，取三个值：老、中、青；第三列是收入，同样取三个值：高、中、低；第四列是学生，取两个值：是、否；第五列是信誉，取两个值：优、良。最后一列是销售结果，可以理解为分类标签，取两个值：买、不买。

那么对于取任意给定特征值的一个客人（测试样例），算法需要帮助公司将这位客人归类，即预测这位客人是属于“买”计算机的那一类，还是属于“不买”计算机的那一类。并且给出判断的依据。

下面我们引入 CLS 算法(Concept Learning System)的思想，CLS 是早期的决策树学习算法。它是各类决策树学习算法的基础。为了便于理解，先用手工实现上例的决策树。我们将决策树设计为三类节点，根节点、叶子节点和内部节点。如果从一棵空决策树开始，任意选择第一个特征就是根节点；我们按照某种条件进行划分，如果划分到某个子集为空，或子集中的所有样本已经归为同一个类别标签，那么该子集为叶结点，否则这些子集就对应于决策树的内部结点，如果是内部节点就需要选择一个新的类别标签继续对该子集进行划分，直到所有的子集都为叶子节点，即为空或者属于同一类。

接下来，就按照上述的规则进行划分，我们选年龄特征作为根节点，这个特征取三个值：老、中、青。我们将所有的样本分为老、中、青三个集合，构成决策树的第一层：

现在我们暂时忽略其他特征，仅关注年龄的取值，将表格变为（如表 3.2）的形式。得出以下结果：

1. 年龄=青，是否购买：不买、买
2. 年龄=中，是否购买：买
3. 年龄=老，是否购买：不买、买

此时，年龄为中年时，是否购买标签都一致的变为买。此时的中年就称为决策树

的叶子节点。当年龄为青年和老年，是否购买有两个选择，可以继续分解。

表 3.2　按年龄划分的表格

计数	年龄	收入	学生	信誉	是否购买
64	青	高	否	良	不买
64	青	高	否	优	不买
128	青	中	否	良	不买
64	青	低	是	良	买
64	青	中	是	优	买
128	中	高	否	良	买
64	中	低	是	优	买
32	中	中	否	优	买
32	中	高	是	良	买
60	老	中	否	良	买
64	老	低	是	良	买
64	老	低	是	优	不买
132	老	中	是	良	买
64	老	中	否	优	不买

现在，将年龄特征等于青年的选项剪切出一张表格，选择第二个特征：收入，并根据收入排序（如表 3.3）：

表 3.3　按青年-收入划分的表格

计数	年龄	收入	学生	信誉	是否购买
64	青	高	否	良	不买
64	青	高	否	优	不买
128	青	中	否	良	不买
64	青	中	是	优	买

64	青	低	是	良	买

其中，高收入和低收入的特征值只有一个类别标签买。将其作为叶子节点。然后继续划分中等收入的下一个特征：学生，于是有了下表（如表 3.4）：

表 3.4　按青年-中-学生划分的表格

计数	年龄	收入	学生	信誉	是否购买
128	青	中	否	良	不买
64	青	中	是	优	买

学生特征只有两个取值，当取否时，对应的标签为不买；当取是时，对应的标签为买。此时，学生特征就生成了决策树左侧分支的所有节点。

如图 3.1 所示中的圆角矩阵为根节点或内部节点：就是可以继续划分的节点；图中的椭圆形节点就是叶子节点，即不能在划分的节点，一般叶子节点都指向一个分类标签，即产生一种决策。

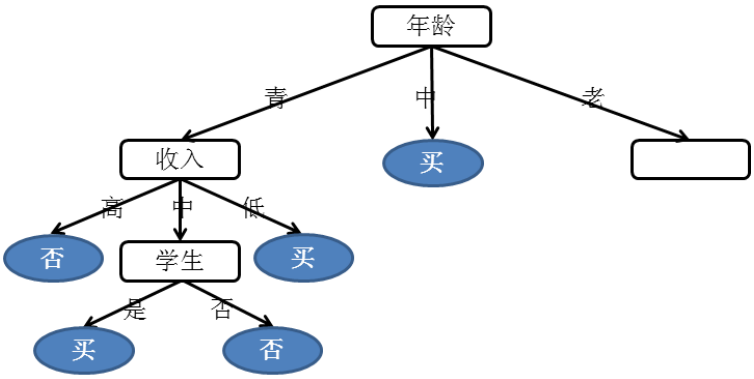


图 3.1　按年龄-收入-学生划分的决策树分支

接下来，我们继续右侧的分支的划分，但在划分时我们做一个简单的变化，划分的顺序为信誉→收入→学生→计数。

这样整个划分过程就变得简单了，当信誉取值为良时，类别标签仅有一个选项就是买。那么信誉为良就是叶子节点。当信誉取值为优时，类别标签仅有一个选项就是不买（如表 3.5）。

表 3.5　按老年-信誉划分的表格

计数	年龄	收入	学生	信誉	是否购买
60	老	中	否	良	买
132	老	中	是	良	买
64	老	低	是	良	买
64	老	中	否	优	不买
64	老	低	是	优	不买

划分最终结果如下（如图 3.2）：

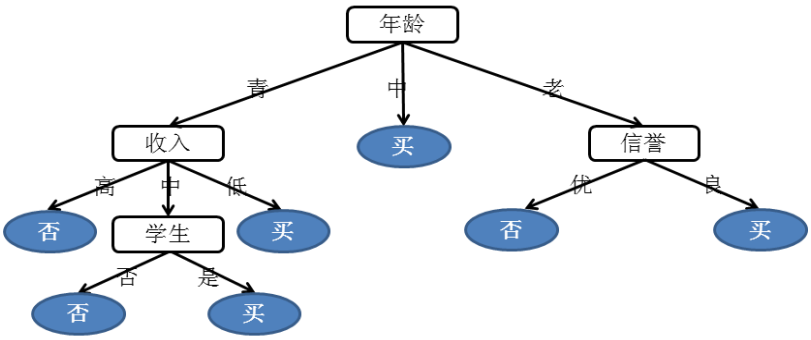


图 3.2　销售调查表完整的决策树

图 3.2 我们就得到了关于例子问题的一棵完整的决策树。为了便于做出判断，我们对这棵树的方向做了调整，即将所有确定购买的叶子节点都放到了树的右侧，不买的节点都放到了树的左侧，这样我们就能够方便的回答上例提出的问题：

对于任意用户，当出现从内部节点向左到叶子节点路径时，就是不购买的用户，反之，即出现从内部节点向右到叶子节点的路径时，就是购买的用户。

对于任意的一个用户，我们先考察其年龄特征，如果是中年人，一般会购买本公司的产品；如果是青年人，高收入和低收入层都会购买，中等收入层还需要做进一步判断，如果不是学生就会购买。如果是老年客户，那么首先看一下他们的信誉，如果是良就会购买，如果信誉为优，多数不会购买。

这是从定性的角度对潜在用户的判断，下面我们给出定量上的判断（如图 3.3）：

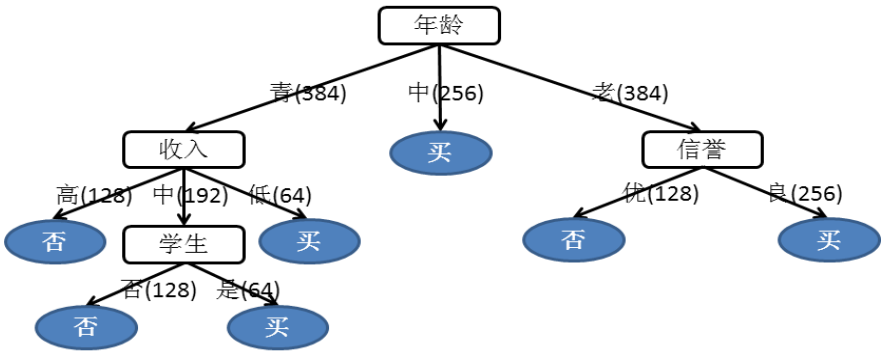


图 3.3　销售调查决策树的概率计算

我们知道，全部的计数特征总数为 1024，将图 3.3 中路径边的数据除以 1024 这个总数，就得到了每个节点的购买概率。有关概率的问题我们下一节具体来讲。

3.1.2　决策树的算法框架

- ❑ 决策树主函数：各种决策树的主函数都大同小异，本质上是个递归函数。该函数主要的功能是按照某种规则生长出决策树的各个分支节点，并根据终止条件结束算法。一般来讲，主函数需要完成如下几个功能：

1. 输入需要分类的数据集和类别标签；
2. 根据某种分类规则得到最优的划分特征，并创建特征的划分节点—计算最优特征子函数；
3. 按照该特征的每个取值划分数据集为若干部分—划分数据集子函数；
4. 根据划分子函数的计算结果构建出新的节点，作为树生长出的新分支；
5. 检验是否符合递归的终止条件
6. 将划分的新节点包含的数据集和类别标签作为输入，递归执行上述步骤

- ❑ 计算最优特征子函数，该函数是继主函数外最重要的函数。每种决策树之所以不同一般都因为最优特征选择的标准上有所差异，不同的标准导致不同类型的决策树，例如 ID3 的最优特征选择标准是信息增益、C4.5 是信息增益率，CART 是节点方差的大小等等。后面所讲的理论部分，都是对特征选择标准而言的。

  算法逻辑上，一般选择最优特征需要遍历整个数据集，评估每个特征，找到最优的那一个返回。

- ❑ 划分数据集：划分数据集函数的主要功能是分隔数据集，有的需要删除某个特征轴所在的数据列，返回剩余的数据集。有的干脆就将数据集一份为二。虽然实现有所不同，但基本含义都是一致的。

- 分类器：所有的机器学习算法要用于分类或回归预测。决策树的分类器就是将测试遍历整个生成的决策树，并找到最终的叶子节点的类别标签。这个标签就是返回的结果。

上述四大部分就构成了决策树算法的基本框架。

3.1.3 信息熵测度

虽然我们手工实现了上例的决策过程，但是将这种实现方法使用编程形式自动计算还存在一些问题。首先，特征集中的数据常常表现为定性字符串数据，称为标称数据，使用这些数据的算法缺乏泛化能力，在实际计算中需要将这些数据定量化为数字，也就是所谓的离散化。

（如图 3.3），我们可以将年龄、收入、学生、信誉这些特征的特征值转换为 0,1,2,…,n 的形式。这样，年龄={0（青）,1（中）,2（老）}；收入={0（高）,1（中）,2（低）}；学生={0（是）,1（否）}；信誉={0（优）,1（良）}

完成了特征离散化，回顾一下前面的手工计算过程，我们可以总结出这样一条规律，数据特征的划分过程是一个将数据集从无序变为有序的过程。这样我们就可以处理特征的划分依据问题，即对于一个有多维特征构成的数据集，如何优选出某个特征作为根节点。进一步扩展这个问题，如何每次都选择出特征集中无序度最大的那列特征作为划分节点。

为了衡量一个事物特征取值的有(无)序程度，下面我们引入一个重要的概念：信息熵。从这个词诞生开始，它就不是一个很容易理解的概念。为了便于理解，我们将这个词拆分为两部分：“信息”和“熵”。

所谓“熵”（Entropy）是德国物理学家克劳修斯在 1850 年创造的一个术语，他用它来表示任何一种能量在空间中分布的均匀程度。能量分布得越均匀，熵就越大。例如，在一个物理系统中，假如这个系统仅由两个物体构成，它们可以有两种情况：第一种情况是一个高温物体和一个低温物体；另一种情况是两个等温物体。我们就认为第二种情况的熵比第一种情况要高。因为第二种情况中能量的分布更加均匀。

1948 年，美国信息学家香农(Shannon)在他的《信息论》中借用了熵的概念，提出了著名的信息熵。在定义信息熵之前，香农首先定义了信息的概念：信息就是对不确定性的消除。现实中，信息可以理解为系统从信源的“消息”(网络、电话、电视、广播等)，转换的“状态”（天气的变化、温度的增减）等。在概率中我们可以称为它是一个随机事件。通常，一个信源发送出什么事件是不确定的，衡量它可以根据其出现的概率来度量。概率大，出现机会多，不确定性小；反之就大。

不确定性函数 I 就称为事件的信息量，是事件 U 发生概率 p 的单调递减函数；两个独立事件所产生的不确定性应等于各自不确定性之和，即 I(P1,P2) = I (P1)+ I (P2)，这称为可加性。同时满足这两个条件的函数 I 是对数函数，即

$$I(U) = \log(\frac{1}{p}) = -\log(p) \qquad \text{（公式 1）}$$

在一个信源中，不能仅考虑某一单个事件发生的不确定性，而需要考虑信源所有可能情况的平均不确定性。若信源事件有 n 种取值：$U_1 \ldots U_i \ldots Un$，对应概率为：$P_1 \ldots P_i \ldots Pn$，且各个事件的出现彼此独立。这时，信源的平均不确定性应当为单个符号不确定性-logP_i 的统计平均值（E），可称为信息熵，即

$$\mathrm{H(U)} = E[-\log p_i] = -\sum_{i=1}^{n} p_i \log p_i \qquad \text{（公式 2）}$$

上式中若对数一般取 2 为底，就是我们平常所说的信息单位：bit(比特)。

信息熵是事物不确定性的度量标准，也称为信息的单位或“测度”。在决策树中它不仅能用来度量类别的不确性，可以来度量包含不同特征的数据样本与类别的不确定性。即某个特征列向量的信息熵越大，就说明该向量的不确定性程度越大，即混乱程度越大，就应优先考虑从该特征向量着手来进行划分。信息熵为决策树的划分提供最重要的依据和标准。

首先，我们使用信息熵来度量类别标签对样本整体的不确定性。设 S 是 s 个数据样本的集合。假定类别标签具有 m 个不同值，定义 m 个不同类 C_i(i＝1，2，…，m)。设 s_i 是类 C_i 中的样本数。对一个给定的样本分类所需要的信息熵由下式给出：

$$I\left(s_1, s_2, \cdots, s_m\right) = -\sum_{i=1}^{m} p_i \log_2\left(p_i\right) \qquad \text{（公式 3）}$$

其中 p_i 是任意样本属于 C_i 的概率，并用 $p_i = \frac{s_i}{|S|}$ 估计。

接下来，我们使用信息熵来度量每种特征不同取值的不确定性。 设 A 具有 v 个不同值｛a_1，a_2，…，a_v｝。可以用特征 A 将 S 划分为 v 个子集｛S_1，S_2，…，S_V｝；其中，S_j 包含 S 中这样一些样本，它们在 A 上具有值 a_j。如果选 A 作测试特征，即最优划分特征，那么这些子集就是 S 节点中生长出来的决策树分枝。设 s_{ij} 是子集 S_j 中类 C_i 的样本数。根据由 A 划分成子集的熵或期望信息由下式给出：

$$E\left(A\right) = \sum_{j=1}^{v} \frac{s_{1j} + s_{2j} + \cdots + s_{mj}}{s} I\left(s_{1j}, s_{2j}, \cdots, s_{mj}\right) \qquad \text{（公式 4）}$$

其中，$\frac{s_{1j} + s_{2j} + \cdots + s_{mj}}{s}$ 是第 j 个子集的权，并且等于子集（即 A 值为 a_j）中的样本个数除以 S 中的样本总数。其信息熵值越小，子集划分的纯度越高。注意，对于

给定的子集 S_j 为

$$I\left(s_{1j},s_{2j},\cdots,s_{mj}\right)=-\sum_{i=1}^{m}p_{ij}\log_2\left(p_{ij}\right) \quad \text{（公式 5）}$$

其中，$p_{ij}=\frac{s_{ij}}{\left|S_j\right|}$ 是 S_j 中的样本属于类 C_i 的概率。

最后，我们使用信息增益来确定决策树分支的划分依据。它是决策树某个分支上整个数据集信息熵与当前节点信息熵的差值，用 Gain(A)表示，那么在 A 上分枝将获得的信息 增益就是：

$$Gain\left(A\right)=I\left(s_1,s_2,\cdots,s_m\right)-E\left(A\right) \quad \text{（公式 6）}$$

它是由于知道属性 A 的值而导致的熵的期望压缩。具有最高信息增益的特征就可选作给定集合 S 的测试属性。创建一个节点，并以该特征标记，对特征的每个值创建分枝，并据此划分样本。

3.2 ID3 决策树

3.2.1 ID3 算法

有了上面的这些概念，我们就可以来手工实现以下 ID3 算法的决策树生成过程。

1. 计算对给定样本分类所需的信息熵：

如表 3.1，类别标签 S 被分两类：买/不买。其中 S1(买)=640 ；S2(不买)= 384。那么总 S=S1+S2=1024。S1 的概率 P1=640/1024=0.625；S2 的概率 P2=384/1024=0.375。根据公式 3：

$$\begin{aligned} I(S_1,S_2) &= I(640,384)\\ &= -P_1LogP_1-P_2LogP_2\\ &= -\left(P_1LogP_1+P_2LogP_2\right)\\ &= 0.9544 \end{aligned}$$

2. 计算每个特征的信息熵：

2.1 先计算 “年龄”特征的熵。年龄共分三组：青年（0）、中年（1）、老年（2）。其中青年占总样本的概率为：P(0)= 384/1024=0.375; 青年中买/不买比例为: 128/256 ，S_1(买)=128， P_1=128/384；S_2(不买)= 256，P_2=256/384；S=S_1+S_2=384；

根据公式 3：$I(S_1,S_2)=I(128,256)$= 0.9183

其中中年占总样本的概率为：P(1)= 256/1024=0.25;中年买/不买比例为：256/0 ，

S_1(买)=256， P_1=1；S_2(不买)= 0，P_2=0；S=S_1+S_2=256；
根据公式 3：$I(S_1,S_2)=I(256,0)$= 0
其中老年占总样本的概率为：P(2)= 384/1024=0.375；老年买/不买比例为：257/127，
S_1(买)= 257， P_1=257/384；S_2(不买)= 127，P_2=127/384；S=S_1+S_2=384；
根据公式 3：$I(S_1,S_2)=I(257,127)$= 0.9157
那么，年龄的平均信息期望：
E（年龄）=0.375*0.9183+ 0.25*0+ 0.375*0.9157 =0.6877
G（年龄）=0.9544-0.6877 =0.2667
2.2 计算“学生”特征的熵。年龄共分两组：是（0）、否（1）。中间步骤省略，那么，学生的平均信息期望：
E（学生）=0.7811
年龄信息增益=0.9544-0.7811 =0.1733
2.3 计算“收入”特征的熵。年龄共分三组：高（0）、中（1）、低（2）。中间步骤省略，E（收入）=0.9361
收入信息增益=0.9544-0.9361 =0.0183
2.4 计算“信誉”特征的熵。年龄共分两组：优（0）、良（1）。中间步骤省略，
E（信誉）= 0.9048
信誉信息增益=0.9544-0.9048 =0.0496
3.从所有的特征列中选择出信息增益最大的那个作为根节点或内部节点—划分节点，划分整列，首次递归选择年龄列（G=0.2660）来划分。。
4.根据划分节点的不同取值来拆分数据集为若干个子集，然后删去当前的特征列，再计算剩余特征列的信息熵，如果有信息增益，就重复第二步直至划分结束。首次划分后，青年和老年内含有多个标签，所以可以继续划分，中年选项就只剩一种标签，就作为叶子节点。
5.划分结束的标志为：子集中只有一种类别标签，停止划分。
按照这样的逻辑产生的决策树结果如下：

图 3.4 中可以看到使用信息熵生成的决策树要比图 3.2 的决策树层数少。如果数据集的特征很多，那么使用信息熵创建决策树在结构上明显要比其他方法更优，形成最优的决策树结构。

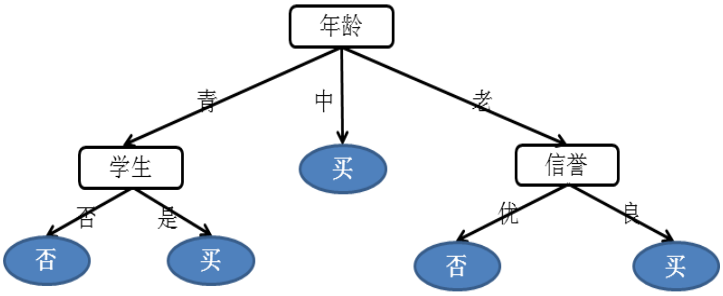


图 3.4　使用信息熵生成决策树

3.2.2　ID3 的实现

上一节，我们使用手工计算实现了一棵决策树，接下来，我们将其转换为编码进行实现。我们定义一个 ID3DTree 类来封装算法：

```
# -*- coding: utf-8 -*-

from numpy import *
import math
import copy
import cPickle as pickle
class ID3DTree(object):
    def __init__(self):        # 构造方法
        self.tree={}           # 生成的树
        self.dataSet=[]        # 数据集
        self.labels=[]         # 标签集
```

1.数据导入函数

```
    def loadDataSet(self,path,labels):
        recordlist = []
        fp = open(path,"rb")  # 读取文件内容
        content = fp.read()
        fp.close()
        rowlist = content.splitlines()      # 按行转换为一维表
        recordlist=[row.split("\t") for row in rowlist if row.strip()]
        self.dataSet = recordlist
        self.labels = labels
```

2.执行决策树函数

```
    def train(self):
        labels = copy.deepcopy(self.labels)
        self.tree = self.buildTree(self.dataSet,labels)
```

3.2.3 决策树主方法

1. 构建决策树：创建决策树主程序

```
def buildTree(self,dataSet,labels):
    cateList = [data[-1] for data in dataSet] # 抽取源数据集的决策标签列
    # 程序终止条件 1 ：如果 classList 只有一种决策标签，停止划分，返回这个决策标签
    if cateList.count(cateList[0]) == len(cateList):
        return cateList[0]
    # 程序终止条件 2: 如果数据集的第一个决策标签只有一个 返回这个决策标签
    if len(dataSet[0]) == 1:
        return self.maxCate(cateList)
    # 算法核心：
    bestFeat = self.getBestFeat(dataSet) # 返回数据集的最优特征轴：
    bestFeatLabel = labels[bestFeat]
    tree = {bestFeatLabel:{}}
    del(labels[bestFeat])
    # 抽取最优特征轴的列向量
    uniqueVals = set([data[bestFeat] for data in dataSet]) # 去重
    for value in uniqueVals: # 决策树递归生长
        subLabels = labels[:] #将删除后的特征类别集建立子类别集
        # 按最优特征列和值分割数据集
        splitDataset = self.splitDataSet(dataSet, bestFeat, value)
        subTree = self.buildTree(splitDataset,subLabels) # 构建子树
        tree[bestFeatLabel][value] = subTree
    return tree
```

2. 计算最多的类别标签：

```
def maxCate(self,catelist):      # 计算出现最多的类别标签
    items = dict([(catelist.count(i), i) for i in catelist])
    return items[max(items.keys())]
```

3. 计算最优特征：

```
def getBestFeat(self,dataSet):
    # 计算特征向量维，其中最后一列用于类别标签，因此要减去
    numFeatures = len(dataSet[0]) - 1                    # 特征向量维数= 行向量维度-1
    baseEntropy= self.computeEntropy(dataSet)        # 基础熵：源数据的香农熵
    bestInfoGain = 0.0;                                  # 初始化最优的信息增益
    bestFeature = -1                                     # 初始化最优的特征轴
    # 外循环：遍历数据集各列,计算最优特征轴
    # i 为数据集列索引：取值范围 0~(numFeatures-1)
    for i in xrange(numFeatures):          # 抽取第 i 列的列向量
        uniqueVals = set([data[i] for data in dataSet])   # 去重：该列的唯一值集
```

```
            newEntropy = 0.0                    # 初始化该列的香农熵
            for value in uniqueVals:            # 内循环：按列和唯一值计算香农熵

                subDataSet = self.splitDataSet(dataSet, i, value) # 按选定列 i 和唯一值分隔数据集
                prob = len(subDataSet)/float(len(dataSet))
                newEntropy += prob * self.computeEntropy(subDataSet)
            infoGain = baseEntropy - newEntropy     # 计算最大增益
            if (infoGain > bestInfoGain):           # 如果信息增益>0;
                bestInfoGain = infoGain             # 用当前信息增益值替代之前的最优增益值
                bestFeature = i                     # 重置最优特征为当前列
        return bestFeature
```

4. 计算信息熵：

```
    def computeEntropy(self,dataSet):                       # 计算香农熵
        datalen = float(len(dataSet))
        cateList = [data[-1] for data in dataSet]           # 从数据集中得到类别标签
        items = dict([(i,cateList.count(i)) for i in cateList]) # 得到类别为 key，出现次数 value 的字典
        infoEntropy = 0.0                                   # 初始化香农熵
        for key in items:    # 香农熵：= - p*log2(p) --infoEntropy = -prob * log(prob,2)
            prob = float(items[key])/datalen
            infoEntropy -= prob * math.log(prob,2)
        return infoEntropy
```

5. 划分数据集：分隔数据集：删除特征轴所在的数据列，返回剩余的数据集

```
    # dataSet：数据集;   axis：特征轴;   value：特征轴的取值
    def splitDataSet(self, dataSet, axis, value):
        rtnList = []
        for featVec in dataSet:
            if featVec[axis] == value:
                rFeatVec = featVec[:axis] # list 操作 提取 0~(axis-1)的元素
                rFeatVec.extend(featVec[axis+1:]) # list 操作 将特征轴（列）之后的元素加回
                rtnList.append(rFeatVec)
        return rtnList
```

3.2.4 训练决策树

下面我们使用上例中的数据训练出对应的决策树。本章提供的数据集可从 http://www.threedweb.cn/thread-1458-1-1.html 下载。该数据集来源于表 3.1 销售调查表，我们将其序列化为一个文本文件，计数列中所有的计数值都扩展为对应的行向量，

行向量一共 1024 条。其中，前四列为年龄、收入、学生、信誉四个特征，我们将特征值做了离散化，所有的特征值都转换为 0,1,2,…,n 的形式。年龄={0(青), 1(中), 2(老)}；收入={0(高), 1(中), 2(低)}；学生={0(否), 1(是) }；信誉={ 0(良), 1(优)}。最后一项是类别标签，yes 表示“买”，no 表示“不买”。

代码如下：

```
# -*- coding: utf-8 -*-

from numpy import *
from ID3DTree import *

dtree = ID3DTree()
# ["age","revenue","student","credit"]  对应  年龄、收入、学生、信誉  四个特征
dtree.loadDataSet("dataset.dat",["age","revenue","student","credit"])
dtree.train ()
print dtree.tree
```

执行结果：

```
 {'age': {'1': 'yes', '0': {'student': {'1': 'yes', '0': 'no'}}, '2': {'credit': {'1': 'no', '0': 'yes'}}}}
```

输出的结果是个数据字典，我们将此字典转换为树状的形式，就是（如图 3.5）的形式。

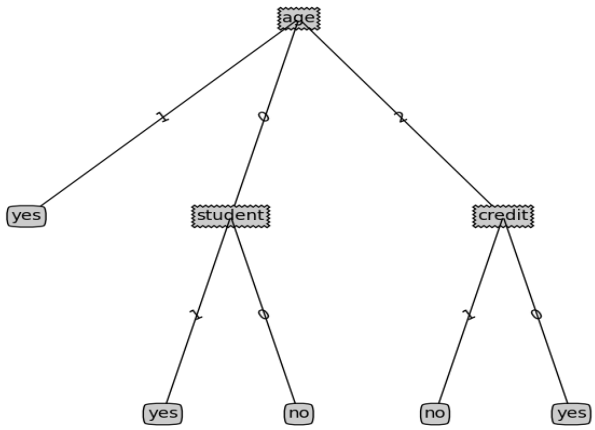


图 3.5　ID3 算法生成的决策树

可以看到图 3.5 创建的决策树结构与图 3.4 相同。说明算法正确。

3.2.5　持久化决策树

ID3 类也提供了专门的方法用于保存决策树到文件，并可从文件读取决策树到内存：

```
    def storeTree(self,inputTree,filename):  # 存储树到文件
```

```
        fw = open(filename,'w')
        pickle.dump(inputTree,fw)
        fw.close()

    def grabTree(self,filename):  # 从文件抓取树
        fr = open(filename)
        return pickle.load(fr)
```

执行代码：

```
dtree.storeTree(dtree.tree,"data.tree")
mytree = dtree.grabTree("data.tree")
print mytree
```

执行结果：

```
{'age': {'1': 'yes', '0': {'student': {'1': 'yes', '0': 'no'}}, '2': {'credit': {'1': 'no', '0': 'yes'}}}}
```

3.2.6 决策树分类

最后我们给出决策树的分类器代码：

```
    def predict(self,inputTree,featLabels,testVec):  # 分类器
        root = inputTree.keys()[0]      # 树根节点
        secondDict = inputTree[root]  # value-子树结构或分类标签
        featIndex = featLabels.index(root)  # 根节点在分类标签集中的位置
        key = testVec[featIndex] # 测试集数组取值
        valueOfFeat = secondDict[key]#
        if isinstance(valueOfFeat, dict):
            classLabel = self.predict(valueOfFeat, featLabels, testVec)#  递归分类
        else: classLabel = valueOfFeat
        return classLabel
```

下面我们随机给出一个潜在客户，即一个行向量，使用学习出的决策树进行分类。执行预测代码如下：

```
dtree = ID3DTree()

labels = ["age","revenue","student","credit"]
vector = ['0','1','0','0']       # ['0','1','0','0','no']
mytree = dtree.grabTree("data.tree")
print "真实输出  ","no","->","决策树输出",dtree.predict(mytree,labels,vector)
```

执行结果：

```
真实输出   no ->  决策树输出  no
```

3.2.7 算法评估

ID3 是比较早的机器学习算法，于 1979 年 Quinlan 就提出了算法的思想。它以信息熵为度量标准，划分出决策树特征节点，每次优先选取信息量最多的属性，即使信息熵变为最小的属性，以构造一棵信息熵下降最快的决策树。

但在另一方面，ID3 在使用中也暴露除了一些问题：

- ID3 算法的节点划分度量标准采用的是信息增益，信息增益偏向于选择特征值个数较多的特征，而取值个数较多的特征并不一定是最优的特征。所以需要改进选择属性的节点划分度量标准。
- ID3 算法递归过程中依次需要计算每个特征值的，对于大型数据会生成比较复杂的决策树：层次和分支都很多，而其中某些分支的特征值概率很小，如果不加忽略就造成了过拟合的问题。即决策树对样本数据的分类精度较高，但在测试集上，分类的结果受决策树分支的影响很大。

3.3 C4.5 算法

针对 ID3 算法存在的一些问题，1993 年，Quinlan 将 ID3 算法改进为 C4.5 算法。该算法成功的解决了 ID3 遇到的诸多问题。在业界得到广泛的应用，并发展成为机器学习的十大算法之一。

3.3.1 信息增益率

C4.5 并没有改变 ID3 的算法逻辑，基本的程序结构仍与 ID3 相同，但在节点的划分标准上做了改进。C4.5 使用信息增益率（GainRatio）来替代信息增益（Gain）进行特征的选择，克服了信息增益选择特征时偏向于特征值个数较多的不足，其中信息增益率定义如下：

$$\text{GainRatio}(S,A)=\frac{Gain(S,A)}{SplitInfo(S,A)} \qquad \text{（公式 7）}$$

其中 Gain(*S,A*)就是 ID3 算法中的信息增益，而划分信息 SplitInfo(*S,A*)代表了按照特征 A 划分样本集 S 的广度和均匀性。

$$SplitInfo(S,A)=-\sum_{i=1}^{c}\frac{|s_i|}{|S|}\log_2(\frac{|S|}{|s_i|}) \qquad \text{（公式 8）}$$

其中，Si 到 Sc 是特征 A 的 C 个不同值，而构成的样本子集。

1.使用信息增益率划分最优节点的方法：

```
def getBestFeat(self, dataSet):
Num_Feats = len(dataSet[0][:-1])
totality = len(dataSet)
BaseEntropy= self.computeEntropy(dataSet)
ConditionEntropy= [] # 初始化条件熵
slpitInfo = []    # for C4.5, calculate gain ratio
allFeatVList=[]
for f in xrange(Num_Feats):
        featList = [example[f] for example in dataSet]
        [splitI,featureValueList] = self.computeSplitInfo(featList)
        allFeatVList.append(featureValueList)
        slpitInfo.append(splitI)
        resultGain = 0.0
        for value in featureValueList:
                subSet = self.splitDataSet(dataSet, f, value)
                appearNum = float(len(subSet))
                subEntropy= self.computeEntropy(subSet)
                resultGain += (appearNum/totality)*subEntropy
        ConditionEntropy.append(resultGain) # 总条件熵
infoGainArray = BaseEntropy*ones(Num_Feats)-array(ConditionEntropy)
infoGainRatio = infoGainArray/array(slpitInfo) # c4.5，信息增益的计算
bestFeatureIndex=    argsort(-infoGainRatio)[0]
return bestFeatureIndex, allFeatVList[bestFeatureIndex]
```

2.计算划分信息(SplitInfo)：

```
def computeSplitInfo(self, featureVList):
        numEntries = len(featureVList)
        featureVauleSetList = list(set(featureVList))
        valueCounts = [featureVList.count(featVec) for featVec in featureVauleSetList]
        # caclulate shannonEnt
        pList = [float(item)/numEntries for item in valueCounts ]
        lList = [item*math.log(item,2) for item in pList]
        splitInfo = -sum(lList)
        return splitInfo, featureVauleSetList
```

3.3.2 C4.5 的实现

参照 ID3DTree 类，我们仅需要修改两个方法即可实现 C4.5 决策树。我们先建立 C45DTree 类，这个类的绝大多数代码来源于 ID3DTree 决策树。下面我们仅需要修改决策树的主方法一些代码片段即可：

修改后的决策树主方法：

```
    def buildTree(self,dataSet,labels):
    cateList = [data[-1] for data in dataSet]
    if cateList.count(cateList[0]) == len(cateList):
        return cateList[0]
    if len(dataSet[0]) == 1:
        return self.maxCate(cateList)
    bestFeat, featValueList = self.getBestFeat(dataSet)
    bestFeatLabel = labels[bestFeat]
    tree = {bestFeatLabel:{}}
    del(labels[bestFeat])
    for value in featValueList:
        subLabels = labels[:]
        splitDataset = self.splitDataSet(dataSet, bestFeat, value)
        subTree = self.buildTree(splitDataset,subLabels)
        tree[bestFeatLabel][value] = subTree
    return tree
```

3.3.3 训练决策树

下面我们仍使用上例中的数据训练出对应的决策树。本章提供的数据集可从 http://www.threedweb.cn/thread-1458-1-1.html 下载：

```
# -*- coding: utf-8 -*-

from numpy import *
from C45DTree import *

dtree = C45DTree()
dtree.loadDataSet("dataset.dat",["age","revenue","student","credit"])
dtree.train()
```

执行结果：

```
{'student': {'1': {'credit': {'1': {'age': {'1': 'yes', '0': 'yes', '2': 'no'}}, '0': 'yes'}}, '0': {'age': {'1': 'yes', '0': 'no',
    '2': {'credit': {'1': 'no', '0': 'yes'}}}}}}
```

绘制成决策树为下面的样子（如图 3.6）：

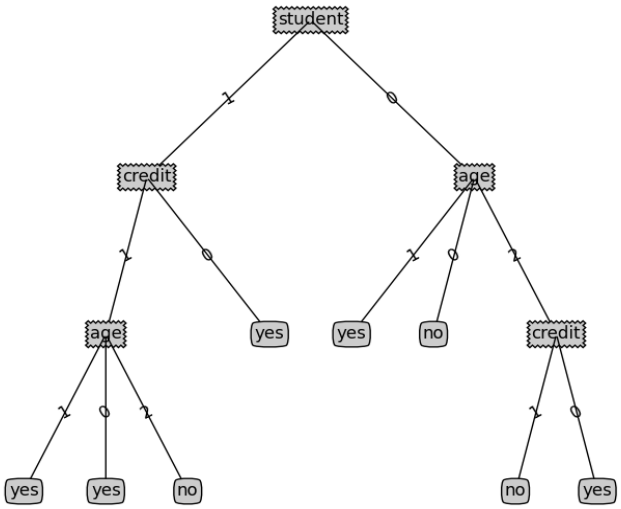


图 3.6　C4.5 算法生成的决策树

3.3.4　分类数据

执行我们创建的朴素贝叶斯类，获取执行结果：

```
# -*- coding: utf-8 -*-

from numpy import *
dtree = C45DTree()
labels = ["age","revenue","student","credit"]
dtree.loadDataSet("dataset.dat",labels)
dtree.train()
vector = ['0','1','0','0'] # ['0','1','0','0','no']
print "真实输出  ","no","->","决策树输出",dtree.predict(dtree.tree,labels,vector)
```

分类结果：

```
真实输出  no -> 决策树输出 no
```

3.4　Scikit-Learn 与回归树

3.4.1　回归算法原理

CART（Classification And Regression Tree）算法是目前决策树算法中最为成熟的

一类算法，应用范围也比较广泛。它既可用于分类，也可用于预测。因为分类的情况前面讲解得已经比较多了，我们下面主要考虑 CART 在预测方面的应用。有关预测方面的知识，建议大家可以先看一下第七章前几节的内容。

西方预测理论一般都是基于回归的，CART 是一种通过决策树方法实现回归的算法，它有很多其他全局回归算法不具有的特性。

在创建回归模型时，样本的取值分为观察值和输出值两种，观察值和输出值都是连续的，不像分类函数那样有分类标签，只有根据数据集的数据特征来创建一个预测的模型，反应曲线的变化趋势。这种情况下，原有分类树的最优划分规则就不再起作用了。在预测中，CART 使用最小剩余方差(Squared Residuals Minimization)来判定回归树的最优划分，这个准则期望划分之后的子树与样本点的误差方差最小。这样决策树将数据集切分成很多子模型数据，然后利用线性回归技术来建模。如果每次切分后的数据子集仍然难以拟合就继续切分。在这种切分方式下，创建出的预测树，每个叶子节点都是一个线性回归模型。这些线性回归模型反应了样本集合（观测集合）中蕴含的模式，也被称为模型树。因此，CART 不仅支持整体预测，也支持局部模式的预测，并有能力从整体中找到模式，或根据模式组合成一个整体。整体与模式之间的相互结合，对于预测分析非常有价值。因此 CART 决策树算法在预测中的应用非常广泛。

下面给出 CART 的算法流程：

- 决策树主函数：决策树的主函数是个递归函数。该函数主要的功能是按照 CART 的规则生长出决策树的各个分支节点，并根据终止条件结束算法：

1. 输入需要分类的数据集和类别标签；
2. 使用最小剩余方差判定回归树的最优划分，并创建特征的划分节点—最小剩余方差子函数；
3. 在划分节点划分数据集为两部分—二分数据集子函数；
4. 根据二分数据的结果构建出新的左、右节点，作为树生长出的两个分支；
5. 检验是否符合递归的终止条件
6. 将划分的新节点包含的数据集和类别标签作为输入，递归执行上述步骤

- 使用最小剩余方差子函数，计算数据集各列的最优划分方差、划分列、划分值。
- 二分数据集：根据给定的分割列和分割值将数据集一份为二，分别返回

代码实现：# 二元切分数据集

```
# dataSet,输入的数据集 feature,特征列  value 二分点的取值
def binSplit(dataSet, feature, value):
    # 数据集第 feature 列大于 value 的所有行向量
    mat0 = dataSet[nonzero(dataSet[:,feature] > value)[0],:][0]
    # 数据集第 feature 列小于等于 value 的所有行向量
```

```
    mat1 = dataSet[nonzero(dataSet[:,feature] <= value)[0],:][0]
    return mat0,mat1
```

- 剪枝策略：使用前剪枝和后剪枝策略剪枝计算的决策树。

3.4.2 最小剩余方差法

回归树中，数据集均为连续型的，连续数据的处理方法与离散数据不同，离散数据是按每个特征的取值来划分，而连续特征则要计算出最优划分点。但在连续数据集上计算线性相关度非常简单，算法思想来源于最小二乘法，有关最小二乘法的讲解，在第七章有详细的说明。

我们这里先给出 CART 选择最优划分点的方法，最小剩余方差法。首先求取划分数据列的均值和总方差，总方差的计算方法有两种：

- 求取均值 std，计算每个数据点与 std 的方差，然后 n 个点求和。
- 求取方差 var，然后 var_sum = var*n，n 为数据集数据数目。

那么，每次最佳分支特征的选取过程为：

1．先令最佳方差为无限大 bestVar = inf。

2．依次遍历所有特征列以及每个特征列的所有样本点（这是一个二重循环），在每个样本点上二分数据集。

3. 计算二分数据集后的总方差 currentVar（划分后左右子数据集的总方差之和），如果 currentVar < bestVar，则 bestVar = currentVar.

4．返回计算的最优分支特征列、分支特征值（连续特征则为划分点的值），以及左右分支子数据集到主程序。

代码精选，选自《机器学习实战》

```
# 选择最优分割点
# leafType: 叶子节点线形回归函数
# errType: 最小剩余方差实现函数
# ops:允许的方差下降值,最小切分样本数
def getBestFeat(dataSet, leafType=regLeaf, errType=regErr, ops=(1,4)):
    tolS = ops[0]; # 允许的方差下降值
    tolN = ops[1]# 最小切分样本数
    #---- 算法终止条件 1 开始 ----#
    splitdataSet = set(dataSet[:,-1].T.tolist()[0])
    if len(splitdataSet) == 1:
        return None, leafType(dataSet)    # 返回值: leafType(dataSet):树的叶子节点
    #---- 计算 dataSet 各列的最优划分方差,划分列,划分值 ----#
    m,n = shape(dataSet)# 返回数据集的行数和列数
    S = errType(dataSet)     # 计算整个数据集的回归方差,S
```

```
    bestS = inf; bestIndex = 0; bestValue = 0    #初始化最优参数: 最大方差、最优划分列、最优划
     分值
    for featIndex in xrange(n-1):# 按列循环
        for splitVal in set(dataSet[:,featIndex]):    # 按行循环—去重
            mat0, mat1 = binSplit(dataSet, featIndex, splitVal) # 二元划分数据集
            # mat0 的行数 小于 tolN 或 mat1 的行数 小于 tolN
            if (shape(mat0)[0] < tolN) or (shape(mat1)[0] < tolN): continue
            newS = errType(mat0) + errType(mat1)    # 计算最小方差和
            if newS < bestS:
                bestIndex= featIndex    # 最优索引 <- 特征索引
                bestValue = splitVal     # 最优值 <- 分割值
                bestS = newS             # bestS <- newS
    #---- DataSet 的最优划分参数：方差、划分列、划分值计算结束 ----#

    #----  算法终止条件 2 开始:返回的是值节点类型 ----#
    if (S - bestS) < tolS:
        return None, leafType(dataSet)
    #----  算法终止条件 3 开始 ----#
    # 二元划分数据集:按划分列和划分值分隔 dataSet
    mat0, mat1 = binSplitDataSet(dataSet, bestIndex, bestValue)
    # mat0 的行数小于 tolN 或 mat1 的行数小于 tolN
    if (shape(mat0)[0] < tolN) or (shape(mat1)[0] < tolN):
     return None, leafType(dataSet)
    # 算法终止的前 3 个条件的划分列为 None,说明为叶子节点,本枝分类树划分结束
    #----  算法终止条件 4 开始:返回的是子树节点类型 ----#
    # 返回最优特征的划分列和划分值,但回归树还需递归划分
    return bestIndex,bestValue
```

3.4.3 模型树

使用 CART 做预测是把叶子节点设定为一系列的分段线性函数，这些分段线性函数是对源数据曲线的一种模拟，每个线性函数都被称为一个模型树。模型树具有很多优秀的性质，它包含了如下的特征：

- 一般而言，样本总体的重复性不会很高，但局部模式经常重复，也就是我们所说的历史不会简单的重复，但会重演。模式比总体对未来的预测而言更有用。
- 模式给出了数据的范围，它可能是个时间范围，也可能是个空间范围；而且模型还给出了变化的趋势，可以是曲线也可以是直线，这依赖于使用的回归算法。这些因素使模型具有很强的可解释性。

- 传统的回归方法，无论是线性回归还是非线性回归，都不如模型树包含的信息丰富，因此模型树也具有更高的预测准确度。

也可以用 CART 单独来创建模型树。它的创建过程大体上与回归树是一样的，这里就不细说了。

3.4.4 剪枝策略

因为使用了连续型数据，CART 可以生长出大量的分支树，为了避免过拟合的问题，预测树采用了剪枝的方法。剪枝方法有很多，主流的方法有两类，先剪枝和后剪枝。先剪枝给出一个预定义的划分阈值，当节点的划分子集某个标准低于预定义的阈值时，子集划分将终止。但是选取适当的阈值比较困难，过高会导致过拟合而过低会导致欠拟合，因此需要人工反复的训练样本才能得到很好的效果。预剪枝也有优势，由于它不必生成整棵决策树，且算法简单，效率高，适合大规模问题的粗略估计。

另一种剪枝策略是后剪枝，也称为悲观剪枝。后剪枝是指在完全生成的决策树上，根据一定的规则标准，剪掉树中不具备一般代表性的子树，使用叶子结点取而代之，进而形成一棵规模较小的新树。后剪枝递归估算每个内部节点所覆盖样本节点的误判率。也就是计算决策树内部节点的误判率，如果内部节点低于这个误判率就将其变成叶子节点，该叶子节点的类别标签由原内部节点的最优叶子节点所决定。那么如何估计这个分类树内部节点的误判率呢？

现在我们考虑一个叶子节点，它覆盖了 N 个样本，其中有 E 个错误，那么该叶子节点的错误率为(E+0.5)/N。这个 0.5 就是惩罚因子。现在考虑一棵子树，如果一棵树错误分类一个样本值为 1，正确分类一个样本值为 0，该树错误分类的概率（误判率）为 e（e 为分布的固有属性,可以通过 $(\sum E_i + 0.5 * L) / \sum N_i$ 统计出来），那么树的误判次数就是伯努利分布，我们可以估计出该树的误判次数均值和标准差：

$$E(substree\_err\_count) = N * e \qquad \text{（公式 9）}$$

$$\text{var}(substree\_err\_count) = \sqrt{N * e * (1-e)} \qquad \text{（公式 10）}$$

把子树替换成叶子节点后，该叶子的误判次数也是一个伯努利分布，其概率误判率 e 为(E+0.5)/N，因此叶子节点的误判次数均值为

$$E(leaf\_err\_count) = N * e \qquad \text{（公式 11）}$$

使用训练数据，子树总是比替换为一个叶节点后产生的误差小，但是使用校正后有误差计算方法却并非如此，当子树的误判个数大过对应叶节点的误判个数一个标准差之后，就决定剪枝：

$E(substree\_err\_count) - \text{var}(substree\_err\_count) > E(leaf\_err\_count)$（公式 12）

这个条件就是剪枝的标准。

代码精选，选自《机器学习实战》

```
def prune(tree, testData):
    if shape(testData)[0] == 0: return getMean(tree) # 如果没有测试数据输入,运行 getMean,程序退出
    # 如果左、右子节点是树
    if (isTree(tree['right']) or isTree(tree['left'])):
        # 对测试集按划分列和树的划分值进行二元分割
        lSet, rSet = binSplitDataSet(testData, tree['spInd'], tree['spVal'])
    if isTree(tree['left']): tree['left'] = prune(tree['left'], lSet) # 如果左节点是树，对测试集递归剪枝
    if isTree(tree['right']): tree['right'] =  prune(tree['right'], rSet) # 如果右节点是树，对测试集递归剪枝
    # 如果左右节点都不是树
    if not isTree(tree['left']) and not isTree(tree['right']):
        lSet, rSet = binSplitDataSet(testData, tree['spInd'], tree['spVal']) # 对测试集按划分列和划分值进行二元分割
        # 计算左右子树的方差
        errorNoMerge = sum(power(lSet[:,-1] - tree['left'],2)) + sum(power(rSet[:,-1] - tree['right'],2))
        # 执行合并：树节点均值
        treeMean = (tree['left']+tree['right'])/2.0
        # 计算合并的方差
        errorMerge = sum(power(testData[:,-1] - treeMean,2))
        # 如果合并后的方差小于不合并方差
        if errorMerge < errorNoMerge:
            # print "merging"
            return treeMean  # 返回节点均值,执行合并
        else: return tree # 否则直接返回,不进行合并
    else: return tree
```

3.4.5 Scikit-Learn 实现

CART 的实现有很多种，源码在很多地方都可以找到，相信读者在阅读完前面的部分之后，有能力看懂，并且实现出 CART 的算法，本书就不提供源码的讲解了。下面，我们使用 Scikit-Learn 中的决策树算法来看一下 CART 的预测效果，使读者有一个直观的认识。

1．数据可视化函数

```
def plotfigure(X,X_test,y,yp):
    plt.figure()
    plt.scatter(X, y, c="k", label="data")
    plt.plot(X_test, yp, c="r", label="max_depth=5", linewidth=2)
    plt.xlabel("data")
    plt.ylabel("target")
    plt.title("Decision Tree Regression")
    plt.legend()
    plt.show()
```

2．执行决策树预测：

```
# -*- coding: utf-8 -*-

import numpy as np
from numpy import *
from sklearn.tree import DecisionTreeRegressor
import matplotlib.pyplot as plt

x = np.linspace(-5,5,200)
siny = np.sin(x)      #  给出 y 与 x 的基本关系
X = mat(x).T
y = siny+np.random.rand(1,len(siny))*1.5 #  加入噪声的点集
y = y.tolist()[0]
# Fit regression model
clf = DecisionTreeRegressor(max_depth=4) # max_depth 选取最大的树深度，类似前剪枝
clf.fit(X, y)

# Predict
X_test = np.arange(-5.0, 5.0, 0.05)[:, np.newaxis]
yp = clf.predict(X_test)

plotfigure(X,X_test,y,yp)
```

执行结果（如图 3.7）：

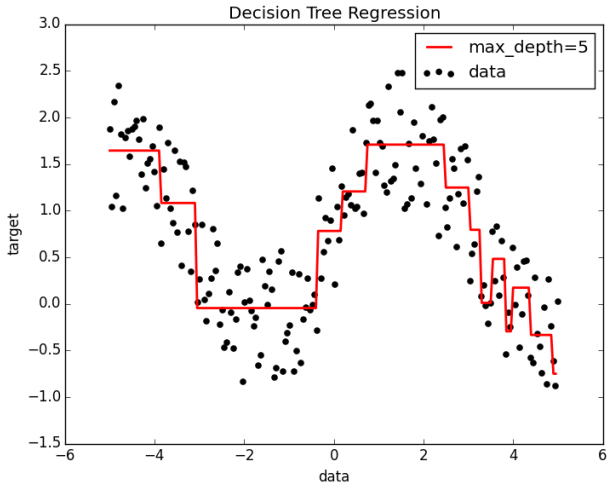


图 3.7　scikit-learn 生成的决策树

3.5　结语

本章的第一节从一个实例引入了决策树算法的基本思想，并以手工的方式构建了一棵最简单的决策树。接下来提出了决策树算法的基本框架。最后提出了数据集各个特征维度的信息熵测度，信息熵的概念和计算。

第二节，详细介绍了 ID3 算法的基本原理和 Python 实现。首先介绍了部分的使用手工方式实现了 ID3 算法的基本逻辑，之后给出了 Python 算法代码，最后通过一个实例完成了 ID3 决策树算法的过程。

第三节，我们在 ID3 的基础上，发展了 ID4.5 算法，并引入了信息熵的改进测度，信息增益率。之后使用 python 实现了 ID4.5 算法，并进行实验数据的训练。

第四节，我们简单介绍了 CART 回归树算法的主要内容以及前后剪枝的概念，对于最少剩余方差法和剪枝策略，我们还给出了样例代码。在第四节的结束，我们使用 Scikit-Learn 训练了一棵 CART 回归树。