第 12 章

算法总结

Algorithm Summary

本书向大家介绍了若干种不同的算法，假如读者研习过书中的示例，那么现在手头就应该有大多数算法的 Python 实现代码。之前的章节在内容组织上都是围绕着一个样例问题、解决该问题的相应算法，及其变体这样的形式来介绍的。本章可以作为相关算法的一个参考。因此，假如读者希望对某个新的数据集进行数据挖掘或是机器学习，那么不妨可以参看一下此处的算法，然后再择优而用；同时，我们还可以利用已经编写好的算法实现代码，帮助我们对数据进行分析。

为了让大家不至于因查找某种算法的细节，而重新回过头去翻阅整本书，笔者会在本章中提供针对于每种算法的算法描述、工作原理的扼要概述、适用的数据集类型，以及算法实现代码的使用方法。除此以外，笔者还会提供每一种算法的优缺点说明（或者，如果你愿意的话，也可以将其理解为：怎样将这些好方法推销给你的老板☺），偶尔还会借助示例来解释算法的某些特征。这些例子都被经过了极大的简化——其中的多数例子只要大家看一下数据就知道该如何解决——但是对于演示而言，它们都是非常有价值的。

我们首先从监督学习算法开始讲起，这类算法根据训练样本来推测某一分类或某个数值。

贝叶斯分类器

Bayesian Classifier

贝叶斯分类器是在第 6 章介绍的。在那一章中，我们已经学会了如何建立一个文档分类系统，将其用于垃圾邮件过滤，或是根据关键字的模糊搜索来对一组文档进行划分。

尽管所有的例子都是关于文档处理的，但是第 6 章的贝叶斯分类器实际上也可以适用于任何其他形式的数据集，只要我们能将其转换成一组特征列表。所谓特征，就是指一个给定项中存在或缺少的某种东西。在文档的例子中，特征就是文档中的单词，但它们也可以是

某个不明对象的特有属性、一种疾病的症状，或是其他任何形式的东西，只要我们能够称其是“存在的”或“缺少的”即可。

训练
Training

和所有监督算法一样，贝叶斯分类器是利用样本进行训练的。每个样本包含了一个特征列表和对应的分类。假定我们要对一个分类器进行训练，使其能够正确判断出：一篇包含单词“python”的文档究竟是关于编程语言的，还是关于蛇的。表 12-1 给出了一个样本训练集。

表 12-1：针对一组文档的特征和分类

特征	分类
Python 是以鸟和哺乳动物为食的大蟒	蛇
Python 最初是作为一门脚本语言被开发出来的	语言
在印度尼西亚发现了一条 14.935 米（49 英尺）长的 Python	蛇
Python 具有动态类型系统	语言
拥有鲜艳表皮的 Python	蛇
开源项目	语言

分类器记录了它迄今为止见过的所有特征，以及这些特征与某个特定分类相关联的数字概率。分类器逐一接受样本的训练。当经过某个样本的训练之后，分类器会更新该样本中特征与分类的概率，同时还会产生一个新的概率，即：在一篇属于某个分类的文档中，含有指定单词的概率。例如，经过如表 12-1 所示的一组文档的训练之后，也许我们最终会得到如表 12-2 所示的一组概率。

表 12-2：单词属于某个给定分类的概率

特征	语言	蛇
dynamic	0.6	0.1
constrictor	0.0	0.6
long	0.1	0.2
source	0.3	0.1
and	0.95	0.95

从上表中我们可以看到，经过训练之后，特征与各种分类的关联性更加明确了。单词“constrictor”属于蛇的分类概率更大，而单词“dynamic”属于编程语言的分类概率更大。

另一方面，有些特征的所属分类则并没有那么明确，比如：单词“and”出现在两个分类中的概率是差不多的（单词“and”几乎会出现在每一篇文档中，不管它属于哪一个分类）。分类器在经过训练之后，只会保留一个附有相应概率的特征列表，与某些其他的分类方法不同，此处的原始数据在训练结束之后，就没有必要再加以保存了。

分类
Classifying

当一个贝叶斯分类器经过训练之后，我们就可以利用它来对新的项目进行自动分类了。假定我们有一篇新的文档，包含了特征“long”、“dynamic”和“source”。表 12-2 列出了每个特征的概率，但这些概率只是针对于各个单词而言的。如果所有单词同属于一个分类的概率值更大，那么答案显然是很清楚的。然而在本例中，“dynamic”属于语言分类的概率更大，而“long”属于蛇分类的概率更大。因此，为了真正对一篇文档进行有效地分类，我们需要一种方法能将所有特征的概率组合到一起，形成一个整体上的概率。

解决这一问题的一种方法是利用我们在第 6 章中介绍过的朴素贝叶斯分类器。它是通过下面的公式将概率组合起来的：

*Pr(Category | Document) = Pr(Document | Category) * Pr(Category)/ Pr(Document)*

此处：

*Pr (Document | Category) = Pr(Word1 | Category) * Pr(Word2 | Category) *…*

*Pr(Word | Category)*的取值来自于上表，比如：*Pr (dynamic | Language)* = 0.6。*Pr(Category)*的取值则等于某个分类出现的总体几率。因为“language”有一半的机会都会出现，所以*Pr(Language)*的值为 0.5。无论是哪个分类，只要其 *Pr(Category | Document)*的值相对较高，它就是我们预期的分类。

代码使用说明
Using Your Code

为了利用第 6 章中构造的贝叶斯分类器对数据集进行分类，我们所要做的唯一一件事情就是定义一个特征提取函数，该函数的作用是将我们用以训练或分类的数据转化成一个特征列表。在第 6 章中我们处理的是文档，所以该函数将字符串拆分成了一个个单词，但是也可以采用任何其他形式的函数，只要它接受的是一个对象，并且返回一个列表：

```
>>> docclass.getwords('python is a dynamic language')
{'python': 1, 'dynamic': 1, 'language': 1}
```

上述函数可用于创建一个新的分类器，针对字符串进行训练：

```
>>> cl=docclass.naivebayes(docclass.getwords)
>>> cl.setdb('test.db')
>>> cl.train('pythons are constrictors','snake')
```

```
>>> cl.train('python has dynamic types','language')
>>> cl.train('python was developed as a scripting language','language')
```

然后进行分类：

```
>>> cl.classify('dynamic programming')
u'language'
>>> cl.classify('boa constrictors')
u'snake'
```

对于允许使用的分类数量，此处并没有任何的限制，但是为了使分类器有一个良好的表现，我们须要为每个分类提供大量的样本。

优点和缺点

Strengths and Weaknesses

朴素贝叶斯分类器与其他方法相比最大的优势或许就在于，它在接受大数据量训练和查询时所具备的高速度。即使选用超大规模的训练集，针对每个项目通常也只会有相对较少的特征数，并且对项目的训练和分类也仅仅是针对特征概率的数学运算而已。

尤其当训练量逐渐递增时则更是如此——在不借助任何旧有训练数据的前提下，每一组新的训练数据都有可能会引起概率值的变化。(你会注意到，贝叶斯分类器的算法实现代码允许我们每次只使用一个训练项，而其他方法，比如决策树和支持向量机，则须要我们一次性将整个数据集都传给它们。)对于一个如垃圾邮件过滤这样的应用程序而言，支持增量式训练的能力是非常重要的，因为过滤程序时常要对新到的邮件进行训练，然后必须即刻进行相应的调整；更何况，过滤程序也未必有权访问已经收到的所有邮件信息。

朴素贝叶斯分类器的另一大优势是，对分类器实际学习状况的解释还是相对简单的。由于每个特征的概率值都被保存了起来，因此我们可以在任何时候查看数据库，找到最适合的特征来区分垃圾邮件与非垃圾邮件，或是编程语言与蛇。保存在数据库中的这些信息都很有价值，它们有可能会被用于其他的应用程序，或者作为构筑这些应用程序的一个良好基础。

朴素贝叶斯分类器的最大缺陷就是，它无法处理基于特征组合所产生的变化结果。假设有如下这样一个场景，我们正在尝试从非垃圾邮件中鉴别出垃圾邮件来：假如我们构建的是一个 Web 应用程序，因而单词“online”时常会出现在你的工作邮件中。而你的好友则在一家药店工作，并且喜欢给你发一些他碰巧在工作中遇到的奇闻趣事。同时，和大多数不善于严密保护自己邮件地址的人一样，偶尔你也会收到一封包含单词“online pharmacy”的垃圾邮

件。

也许你已经看出了此处的难点——我们往往会告诉分类器“online”和“pharmacy”是出现在非垃圾邮件中的，因此这些单词相对于非垃圾邮件的概率会更高一些。当我们告诉分类

器有一封包含单词“online pharmacy”的邮件属于垃圾邮件时，则这些单词的概率又会进行相应的调整，这就导致了一个经常性的矛盾。由于特征的概率都是单独给出的，因此分类器对于各种组合的情况一无所知。在文档分类中，这通常不是什么大问题，因为一封包含单词“online pharmacy”的邮件中可能还会有其他特征可以说明它是垃圾邮件，但是在面对其他问题时，理解特征的组合可能是至关重要的。

决策树分类器

Decision Tree Classifier

决策树是在第 7 章中介绍的，在那一章我们学到了如何根据服务器日志来对用户的行为进行建模。决策树以其极易理解和解释的特点而著称。图 12-1 给出了一个决策树的例子。

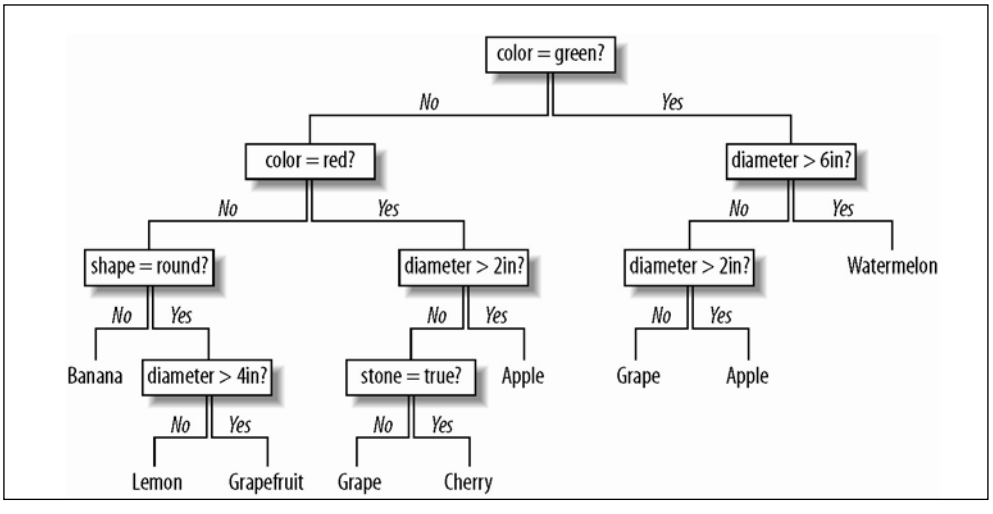

图 12-1：决策树的例子

图中清楚地给出了，当对新项目进行分类时，决策树所要做的工作。从树的根节点开始，我们会对每个节点的判断条件进行检查——如果节点的判断条件满足，就走 Yes 分支，否则，就走 No 分支。这一过程会一直重复进行，直至到达代表预测分类的那个叶节点为止。

训练

Training

利用决策树进行分类非常的简单；但对决策树进行训练则需要更多的技巧。第 7 章介绍的

算法从根部开始构造决策树，在每一步中它都会选择一个属性，利用该属性以最佳的可能方式对数据进行拆分。为了说明这一点，请看如表 12-3 所示的水果数据集。我们将它作为初始数据集。

表 12-3：水果数据

直径	颜色	水果
4	Red	Apple
4	Green	Apple
1	Red	Cherry
1	Green	Grape
5	Red	Apple

为了创建树的根节点，有两个可能的变量可用于对数据进行拆分，它们分别是直径和颜色。第一步就是要对每个变量都进行尝试，从中找出拆分数据效果最好的一个。按颜色拆分数据集得到的结果如表 12-4 所示。

表 12-4：按颜色拆分后的水果数据

Red	Green
Apple	Apple
Cherry	Grape
Apple	

上述数据仍然显得非常的混乱。但是，假如我们按直径（小于 4 英寸、大于或等于 4 英寸）进行拆分，则拆分得到的结果就会变得非常清晰（我们将左侧的数据称为 Subset 1，右侧的数据称为 Subset 2）。此时的拆分情况如表 12-5 所示。

表 12-5：按直径拆分后的水果数据

直径<4 英寸	直径≥4 英寸
Cherry	Apple
Grape	Apple

这显然是一种更好的拆分结果，因为 Subset 2 包含来自初始集合中的所有“Apple”条目。尽管在本例中，哪个变量的拆分效果更好是很明确的；但是当面对规模更大一些的数据集时，就不会总是有如此清晰的拆分结果了。为了衡量一个拆分的优劣，第 7 章引入了熵的概念（代表一个集合中无序的程度）：

- *p(i) = frequency(outcome) = count(outcome) / count(total rows)*

- *Entropy* = 针对所有结果的 *p(i)* * *log(p(i))*之和

集合中的熵偏小，就意味着该集合中的大部分元素都是同质的（homogeneous）；而熵等于 0，则代表集合中的所有元素都是同一类型的。表 12-5 中 Subset 2（直径≥4）的熵即为 0。每个集合中的熵都是用来计算信息增益（information gain）的，信息增益的定义如下：

- *weight1 = subset1* 的大小 / 原始集合的大小
- *weight2 = subset2* 的大小 / 原始集合的大小
- *gain = entropy(original) - weight1*entropy(set1) - weight2*entropy(set2)*

因此对于每一种可能的拆分，我们都会计算出相应的信息增益，并以此来确定拆分用的变量。一旦用以拆分的变量被选定，第一个节点就创建出来了，如图 12-2 所示。

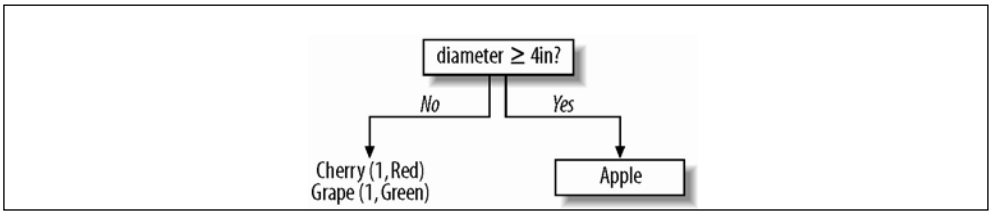

图 12-2：水果决策树的根节点

此处，我们将判断条件显示在了节点的位置，如果数据不满足判断条件，就会沿着 No 分支向下走，如果满足，则会沿 Yes 分支向下走。因为 Yes 分支现在仅有一种可能的结果，因此它就成了叶节点。No 分支依然显得有些混乱，因此可以采用与选择根节点时完全相同的办法对其做进一步的拆分。在本例中，颜色现在成为了拆分数据的最佳变量。这一过程会一直重复下去，直到在某个分支上拆分数据时不会再有信息增益为止。

决策树分类器使用说明

Using Your Decision Tree Classifier

第 7 章中决策树的算法实现代码是基于一个列表的列表进行训练的，每个内部列表都包含了一组值，其中的最后一个值代表分类。我们可以按如下所示的方法来创建前面那个简单的水果数据集

```
>>> fruit=[[4,'red','apple'],
... [4,'green','apple'],
... [1,'red','cherry'],
... [1,'green','grape'],
... [5,'red','apple']]
```

现在，我们可以对这棵决策树进行训练，并利用它对新的样本进行分类：

```
>>> import treepredict
>>> tree=treepredict.buildtree(fruit)
>>> treepredict.classify([2,'red'],tree)
{'cherry': 1}
>>> treepredict.classify([5,'red'],tree)
{'apple': 3}
>>> treepredict.classify([1,'green'],tree)
{'grape': 1}
>>> treepredict.classify([120,'red'],tree)
{'apple': 3}
```

很显然，直径 10 英尺且颜色为紫色的水果不应该是苹果，但决策树受制于其所见过的样本。最后，我们可以将树打印或绘制出来，以理解其决策的推导过程：

```
>>> treepredict.printtree(tree)
0:4?
T-> {'apple': 3}
F-> 1:green?
  T-> {'grape': 1}
  F-> {'cherry': 1}
```

优点和缺点

Strengths and Weaknesses

决策树最为显著的优点在于，利用它来解释一个受训模型是非常容易的，而且算法将最为重要的判断因素都很好地安排在了靠近树的根部位置。这意味着，决策树不仅对分类很有价值，而且对决策过程的解释也很有帮助。和贝叶斯分类器一样，可以通过观察内部结构来理解它的工作方式，同时这也有助于在分类过程之外进一步做出其他的决策。例如，第 7 章中的模型对哪些用户最终会成为付费客户进行了预测，而有了决策树，就可以清晰地显示出哪些变量是最适合用于拆分数据的，这对于规划广告策略，以及判断还应该收集哪些数据而言，都是非常有价值的。

因为决策树要寻找能够使信息增益达到最大化的分界线，因此它也可以接受数值型数据作为输入。能够同时处理分类（categorical）数据和数值数据，对于许多问题的处理都是很有帮助的——这些问题往往是传统的统计方法（比如回归）所难以应对的。另一方面，决策树并不擅长于对数值结果进行预测。一棵回归树可以将数据拆分成一系列具有最小方差的均值，但是如果数据非常复杂，则树就会变得非常庞大，以至于我们无法借此来做出准确的决策。

与贝叶斯分类器相比，决策树的主要优点是它能够很容易地处理变量之间的相互影响。一个用决策树构建的垃圾邮件过滤器可以很容易地判断出“online”和“pharmacy”在分开时并不代表垃圾信息，但当它们组合在一起时则为垃圾信息。

遗憾的是，利用第 7 章中的算法所实现的垃圾邮件过滤器是不实用的。很简单，因为它不支持增量式的训练。（目前，支持增量式训练的决策树替代算法是一个活跃的研究领域）我们可以接受一大堆邮件，并构建一棵用于垃圾邮件过滤的决策树，但是无法对新收到的一封邮件单独进行训练——每次训练都必须重新开始。因为许多人都有成千上万封邮件，所以每次都重新开始是不切实际的。另外，因为节点的数量有可能会非常庞大（每一个特征

都有可能存在或缺失），这些树有可能会变得异常庞大而复杂，这会导致分类效率的降低。

神经网络

Neural Networks

第 4 章向大家介绍了如何根据用户以往点击的链接简单构建一个神经网络，用以对搜索结果的排名进行调整。神经网络可以识别出哪些单词的组合是重要的，以及哪些单词对于某次查询是不重要的。我们不仅可以将神经网络用于分类，还可以将其用于数值预测问题。

第 4 章中的神经网络被当作了分类器——它针对每个链接给出一个数字，并预测数字最大者将会是用户要点击的链接。由于算法为每个链接都给出了一个数字，因此可以利用所有这些数字来改变搜索结果的排名。

有许多不同种类的神经网络。本书涉及的神经网络被称为多层感知器网络（multilayer perceptron network），之所以这样命名是因为它包含一层输入神经元（input neurons），这些神经元会将输入传递给一层或多层隐藏神经元（hidden neurons）。其基本结构如图 12-3 所示。

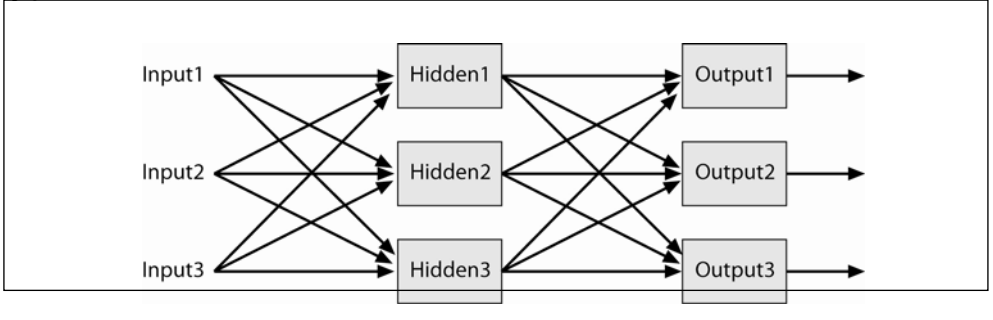

图 12-3：基本的神经网络结构

上述网络有两层神经元。层与层之间通过突触（synapse）彼此相连，每个突触都有一个与之关联的权重。一组神经元的输出是通过突触传入下一层的。如果从一个神经元指向下一个神经元的突触权重越大，那么它对神经元输出的影响也就越大。

作为一个简单的例子，我们再来回顾一下前述“贝叶斯分类器”一节中提到的垃圾邮件过滤问题。在简化了的邮件场景中，一封邮件可能会包含单词“online”、“pharmacy”，或二者的组合。为了判断哪些邮件是垃圾邮件，我们有可能要用到一个如下页图 12-4 所示的神经网络。

在上图中，为了解决垃圾邮件的问题，我们已将突触的权重都设置好了（我们将在下一节中了解到这些突触的设置方法）。位于第一层中的神经元对于用作输入的单词给予响应——如果某个单词存在于邮件信息中，则与该单词关联最强的神经元就会被激活。第二层神经

元接受第一层神经元的输入，因此它会对单词的组合给予响应。最后，这些神经元会将结

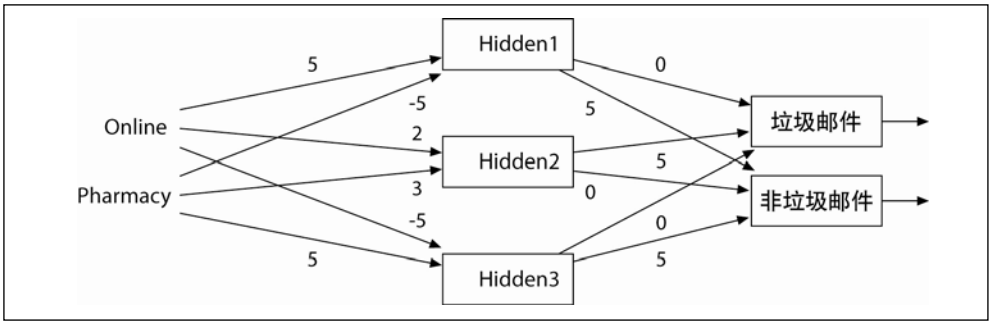

图 12-4：用于垃圾分类的神经网络

果输出，而某些单词的组合则或许会与所有可能的结果形成“强”关联或“弱”关联。最终的决策结论，就是要判定哪一个输出最强。图 12-5 给出了单词“online”在没有单词“pharmacy”跟随的情况下，神经网络对其反应的情况。

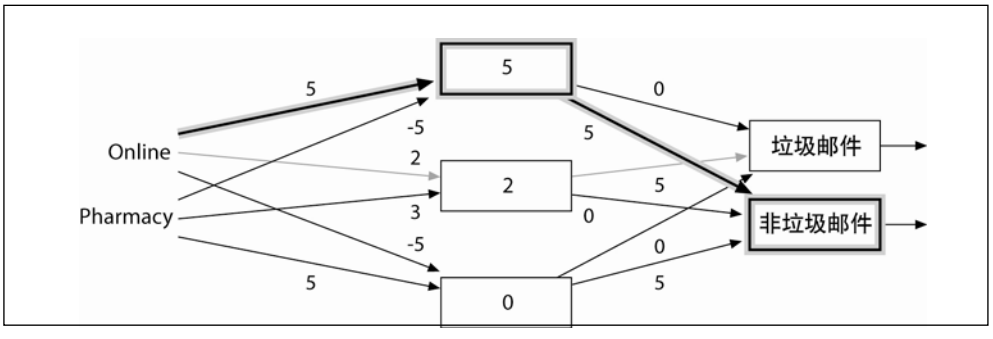

图 12-5：神经网络对单词“online”作出的反应

位于第一层中的某个神经元对“online”给予了响应，并且将其输出至第二层；而位于第二层中的某个神经元则已经学会了识别仅包含单词“online”的邮件。该神经元有一个指向非垃圾邮件的突触，其所拥有的权重值要比指向垃圾邮件的突触更大，因此邮件最终被归类为非垃圾邮件。下页图 12-6 给出了单词“online”在和“pharmacy”一起被传入神经网络时所发生的状况。

因为位于第一层中的神经元是对单个单词给予响应的，所以这次有两个神经元都被激活了。到了第二层，情况变得更有意思。“pharmacy”的存在对“online”产生了消极的影响——第一层中两个神经元的共同作用会激活第二层中间的那个神经元，该神经元经过训练之后会对“online”和“pharmacy”出现在一起的情况给予响应。由于该神经元非常明确地指向垃圾邮件分类，因此邮件就被归类成了垃圾邮件。这个例子说明了，多层神经网络可以非常轻松地处理代表不同事物的各种特征的不同组合。

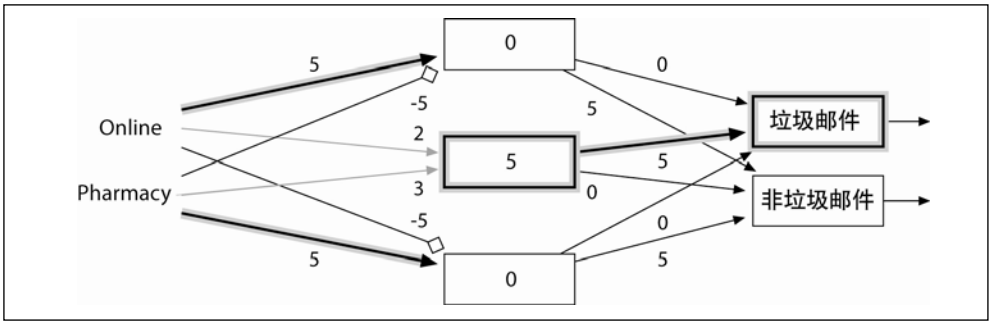

图 12-6：神经网络对“online pharmacy”作出的反应

训练神经网络

Training a Neural Network

在前述的例子中，神经网络已经为每个突触都设置了相应的权重。神经网络的真正威力在于，它们可以从随机的权重值开始，然后通过训练不断地从样本中得到学习。训练多层感知器网络最为常见的方法，也是在第 4 章中介绍的方法，被称为反向传播法（backpropagation）。

为了利用反向传播法对网络展开训练，首先从一个样本（比如单词“online”）及其正确答案（本例中为非垃圾邮件）开始。随后，将样本送入神经网络，观察其当前的推测结果。

开始的时候，网络也许会为垃圾邮件赋予一个比非垃圾邮件更高一些的权重，这是不正确的。为了修正这一错误，我们告诉网络，垃圾邮件的权重应该更接近于 0，而非垃圾邮件则应该更接近于 1。指向垃圾邮件的突触权重，会根据每个隐藏层节点的贡献程度相应地做向下微调，而指向非垃圾邮件的权重则会做向上微调。介于输入层与隐藏层之间的突触权重，也会根据其对输出层中的重要节点的贡献程度进行相应的调整。

关于上述调整的实际公式已在第 4 章中给出。为了防止网络在接受有噪声干扰或不确定性数据作为训练数据时所作出的过度补偿反应（overcompensate），整个训练过程将会缓慢地进行。如此，网络看到某个样本的次数越多，对其分类的效果就会越好。

神经网络代码使用说明

Using Your Neural Network Code

用第 4 章中的代码来解决上述问题是非常简单的。唯一的技巧在于，代码不直接接受单词

作为输入，而是使用数字 ID。因此我们须要为每一个可能的输入赋予一个数字。代码使用一个数据库来保存训练数据，因此它只要打开一个指定名称的数据文件，就可以开始训练了。

```
>>> import nn
>>> online,pharmacy=1,2
>>> spam,notspam=1,2
>>> possible=[spam,notspam]
>>> neuralnet=nn.searchnet('nntest.db')
>>> neuralnet.maketables()
>>> neuralnet.trainquery([online],possible,notspam)
>>> neuralnet.trainquery([online,pharmacy],possible,spam)
>>> neuralnet.trainquery([pharmacy],possible,notspam)
>>> neuralnet.getresult([online,pharmacy],possible)
[0.7763, 0.2890]
>>> neuralnet.getresult([online],possible)
[0.4351, 0.1826]
>>> neuralnet.trainquery([online],possible,notspam)
>>> neuralnet.getresult([online],possible)
[0.3219, 0.5329]
>>> neuralnet.trainquery([online],possible,notspam)
>>> neuralnet.getresult([online],possible)
[0.2206, 0.6453]
```

你会发现，神经网络接受训练的次数越多，其给出的结果就越准确。而且它还能处理偶尔出现的错误样本，并依然保持良好的预测能力。

优点和缺点

Strengths and Weaknesses

神经网络的主要优点是它们能够处理复杂的非线性函数，并且能发现不同输入间的依赖关系。尽管前面的例子只给出了 1 或 0（代表存在或缺失）这样的数字输入，但是任何数字都是可以用作输入的，并且网络也可以将评估所得的数字作为输出。

神经网络也允许增量式训练，并且通常不要求大量空间来保存训练模型，因为它们须要保存的仅仅是一组代表突触权重的数字而已。同时，也没有必要保留训练后的原始数据，这意味着，可以将神经网络用于不断有训练数据出现的应用之中。

神经网络的主要缺点在于它是一种黑盒方法。此处给出的例子设计简单，也很容易理解。但在现实中，一个网络也许会有数百个节点和上千个突触，这使我们很难确知网络何以得到最终的答案。可是无法确知推导的过程对于某些应用而言，也许是一个很大的阻碍。

神经网络的另一个缺点是，在选择训练数据的比率及与问题相适应的网络规模方面，并没有明确的规则可以遵循。最终的决定往往须要依据大量的试验。选择过高的训练数据比率，有可能会导致网络对噪音数据产生过度归纳（overgeneralize）的现象，而选择过低的训练比率，则意味着除了我们给出的已知数据外，网络有可能就不会再进一步学习了。

支持向量机

Support-Vector Machines

支持向量机（SVM）是在第 9 章介绍的，它有可能是本书中所提到的最为复杂的一种分类方法。SVM 接受数据集作为数字输入，并尝试预测这些数据属于哪个分类。例如，也许我们希望通过一组有关身高和跑动速度的数据来确定一支篮球队的队员站位。为了简化起见，此处我们只考虑两种情况——前场的位置需要身材高大的队员，而后场的位置则需要跑得更快的队员。

SVM 是通过寻找介于两个分类间的分界线来构建预测模型的。如果将一组身高与速度的对应值以及每个人的最佳站位在图上绘制出来，就会得到如图 12-7 所示的结果。其中，前场队员以 X 显示，后场队员以 O 显示。除此以外，图上还有若干条直线，将数据拆成了两个分类。

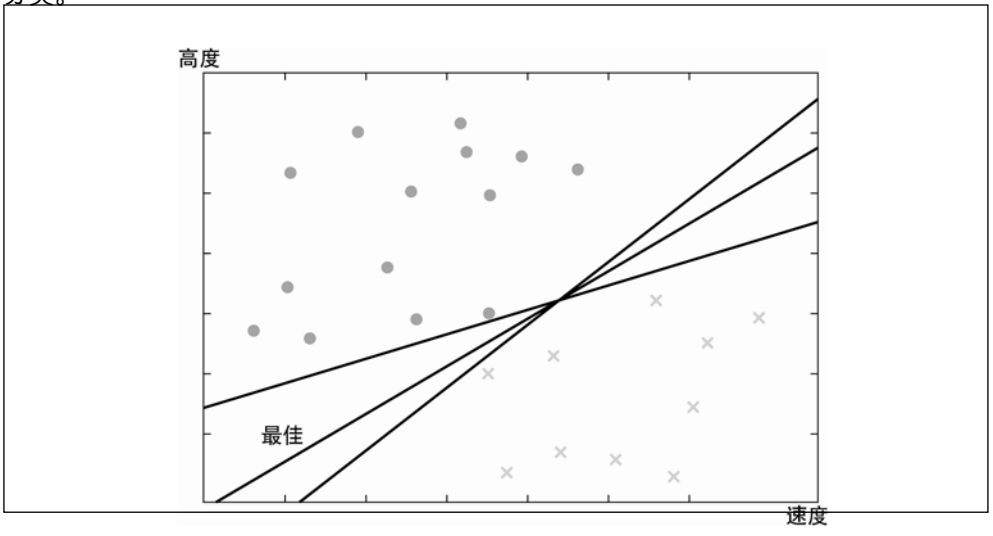

图 12-7：篮球队员及分界线的示意图

通过支持向量机找到的分界线，能够非常清晰地对数据进行划分，这意味着分界线与处于其附近的坐标点彼此间达到了最大可能距离。在图 12-7 中，尽管所有线条都可以对数据进行划分，但其中表现最好的则是标记为“最佳”的那条直线。确定这条分界线所在位置唯一需要的坐标点，是距离它最近的那些点，这些点被称为支持向量。

当分界线找到以后，对新项目的分类只须简单地将其绘制在图上，并观察其落在线的哪一侧即可。而且，一旦找到了这条分界线，对新坐标点的分类就不必再去考查训练数据了，

因此分类的速度非常快。

核技法

The Kernel Trick

与其他借助于向量点积运算的线性分类器一样，支持向量机通常会借助于一种叫做核技法（kernel trick）的技术。为了理解这一点，假设我们所要预测的分类不是队员的站位，而是要判断队员是否适合于一支站位经常不定的业余球队，此时的情况又会如何呢？这个问题更加值得关注，因为此时的划分已不再是线性的了。我们不想要身材太高或速度太快的队员，因为他们的存在会使比赛对其他人而言难度太大，但是我们同样也不希望队员的身材太矮或速度太慢。如图 12-8 所示，图中 O 表示队员对球队是合适的，X 则表示不合适。

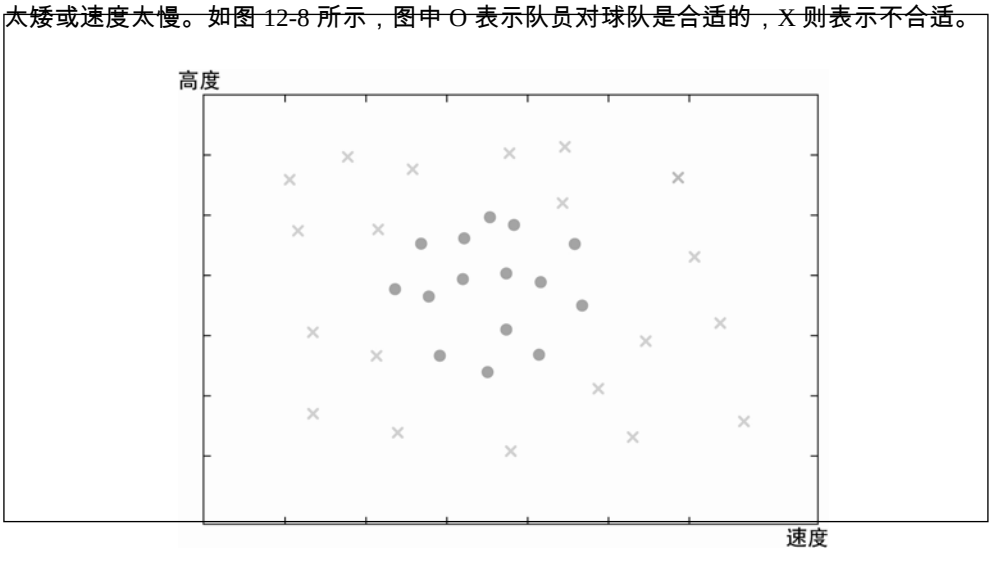

图 12-8：业余篮球队的队员示意图

在图上我们找不到任何可以划分数据的直线，因此在没有按某种方式对数据采取变换之前，我们是无法利用线性分类器来找到有效划分的。解决这一问题的一种方法是，通过在各个轴向上施以不同的函数，将数据变换到另一个不同的空间中——也许是一个超过二维的空间。在本例中，我们可以令身高和速度减去各自的平均值，然后再对两者求平方，以此来构造一个新的空间。如下页图 12-9 所示。

上述方法被称作多项式变换（polynomial transformation），它在不同轴向上对数据进行了变换。现在，我们很容易就能分辨出，在适合球队与不适合球队的队员之间存在着一条分界线，这条线是可以用线性分类器找到的。此时，对于新坐标点的分类，只须将坐标点变换到这一空间，然后再观察其落在线的哪一侧即可。

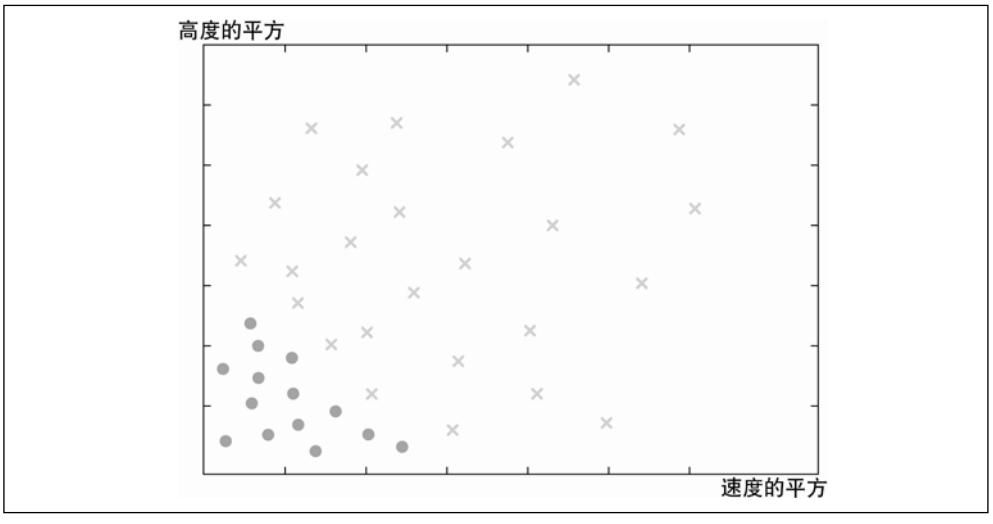

图 12-9：位于多项式空间中的篮球队员

在本例中，坐标点的变换非常成功；但是在大多数时候，为了找到分界线，往往要将坐标点变换到更为复杂的空间。这些空间的维度有的是上千维，有的甚至是无限维的，因此在现实中进行这样的变换并不总是可行的。这便是核技法的用武之地——不再进行空间的变换，而是用一个新的函数来取代原来的点积函数，该函数会在数据被变换到另一个不同的空间之后，返回相应的点积结果。例如，我们不再进行如上所示的多项式变换，而是将：

```
dotproduct(A,B)
```

替换成：

```
dotproduct(A,B)**2
```

在第 9 章，我们构造过一个利用群组均值的简单线性分类器。后来我们还对分类器进行了改造，将点积函数替换为支持向量组合的其他函数，从而使非线性问题也得到了解决。

LIBSVM 使用说明

Using LIBSVM

第 9 章曾经介绍过一个叫做 LIBSVM 的函数库。我们可以利用它对一个数据集进行训练(即在经过变换的空间中找到分界线)，然后再对新的观测数据进行分类：

```
>>> from random import randint
>>> # 随机生成 200 个坐标点
>>> d1=[[randint(-20,20),randint(-20,20)] for i in range(200)]
>>> # 对其进行分类，如果坐标点位于圆内则为 1，否则为 0
>>> result=[(x**2+y**2)<144 and 1 or 0 for (x,y) in d1]
>>> from svm import *
```

```
>>> prob=svm_problem(result,d1)
>>> param=svm_parameter(kernel_type=RBF)
>>> m=svm_model(prob,param)
>>> m.predict([2,2])
1.0
>>> m.predict([14,13])
0.0
>>> m.predict([-18,0])
0.0
```

LIBSVM 支持许多不同的核函数，并且对于一个给定的数据集，可以轻松地选择不同的参数，以尝试各种不同的方法，并从中找出表现最佳者。为了测试一个模型的表现，可以试一试 cross_validation 函数，该函数接受一个参数 n，并将数据集划分成 n 个子集。随后函数会分别将每个子集当作测试集，并利用所有剩余的子集对模型加以训练。该函数最后会返回一个包含答案的列表，可以将其与原始列表进行比较：

```
>>> guesses=cross_validation(prob,param,4)
>>> sum([abs(guesses[i]-result[i]) for i in range(len(guesses))])
28.0
```

当面对一个新的问题时，可以利用不同的参数来尝试不同的核函数，以此来寻找能够给出最佳结果的方案。具体情况可能会因数据集的不同而不同，在我们得到结论之后，就可以利用相应的参数来构造模型，对新的观测数据进行分类。在实践中，我们也许会建立一些嵌套循环以尝试不同的参数值，并记录下给出最佳结果的参数组合。

优点和缺点

Strengths and Weaknesses

支持向量机是一种功能强大的分类器，一旦得到了正确的参数，这种分类器的执行效果，与本书中所提及的其他任何一种分类方法相比，有可能会不相上下或更胜一筹。而且在接受训练之后，它们在对新的观测数据进行分类时速度极快，这是因为分类时只须判断坐标点位于分界线的哪一侧即可。通过将分类输入（categorical inputs）转换成数值输入，可以令支持向量机同时支持分类数据和数值数据。

支持向量机的一个缺点在于，针对每个数据集的最佳核变换函数及其相应的参数都是不一样的，而且每当遇到新的数据集时都必须重新确定这些函数及其参数。在可能的取值范围内进行循环遍历会有助于这一问题的解决，但是这要求我们有足够大的数据集来完成可靠的交叉检验。一般而言，SVM 更适合于那些包含大量数据的问题；而其他方法，如决策树，则更适合于小规模的数据集，并且这些方法还能从数据集中得到很有价值的信息。

如同神经网络，SVM 也是一种黑盒技术——实际上，由于存在向高维空间的变换，SVM 的分类过程甚至更加难于解释。SVM 也许会给出很好的答案，但是我们永远都无法得知找到答案的真正原因。

k-最近邻

k-Nearest Neighbors

在第 8 章我们讨论了，如何利用一种称为 k-最近邻（kNN）的算法进行数值预测，并利用这一算法示范了，如何针对一组给定的样本来构造价格预测模型。我们在第 2 章中介绍过的，用于预测人们对影片或链接的喜好程度的推荐算法，同样也是 kNN 算法的一个简化版本。

kNN 的工作原理，是接受一个用以进行数值预测的新数据项，然后将其与一组已经赋过值的数据项进行比较。算法会从中找出与待预测数据项最为接近的若干项，并对其求均值以得到最终的预测结果。表 12-6 列出了一组数码相机及其百万像素数、变焦能力、销售价格等的信息。

表 12-6：数码相机及其价格

相机	百万像素数	变焦能力	价格
C1	7.1	3.8x	$399
C2	5.0	2.4x	$299
C3	6.0	4.0x	$349
C4	6.0	12.0x	$399
C5	10.0	3x	$449

假定我们想推测一款拥有 6 百万像素和 6 倍变焦透镜的新相机的价格。首先要做的第一件事情就是寻找一种方法能够对两个数据项的近似程度进行度量。第 8 章曾经用过欧几里德距离，而且我们也曾在本书中见过许多其他的距离度量方法，比如皮尔逊相关度和 Tanimoto 分值。本例所采用的是欧几里德距离，借此我们发现表中最为接近的一项是 C3。为了使这一结论可视化，假设将这些数据项绘制到一张以百万像素数为 x 轴，以焦距为 y 轴的二维图上。每个数据项本身则以其相应的价格加以标识，结果如下页图 12-10 所示。

也许你可以接受 349 美元这样的价格作为答案（毕竟这是最为接近的匹配），但是我们无法得知这是否只是一个例外情况。有鉴于此，最好选择多个最佳匹配，然后再对它们求均值。k-最近邻算法中的 k，指的就是用于求均值的最佳匹配数。例如，如果选择三个最佳匹配，并对它们求均值，那么 kNN 中的 k 即为 3。

对基本均值运算的一个扩展，是根据近邻之间距离远近的程度进行加权平均。距离非常接

近的近邻会比距离稍远者拥有更高的权重值。权重与总的距离成正比。第 8 章曾经介绍过确定权重的各种不同的函数。在本例中，我们也许会对 349 美元的价格赋以最大的权重，而两个 399 美元则会赋以相对较小的权重。例如：

```
price = 0.5 * 349 + 0.25 * 399 + 0.25 * 399 = 374
```

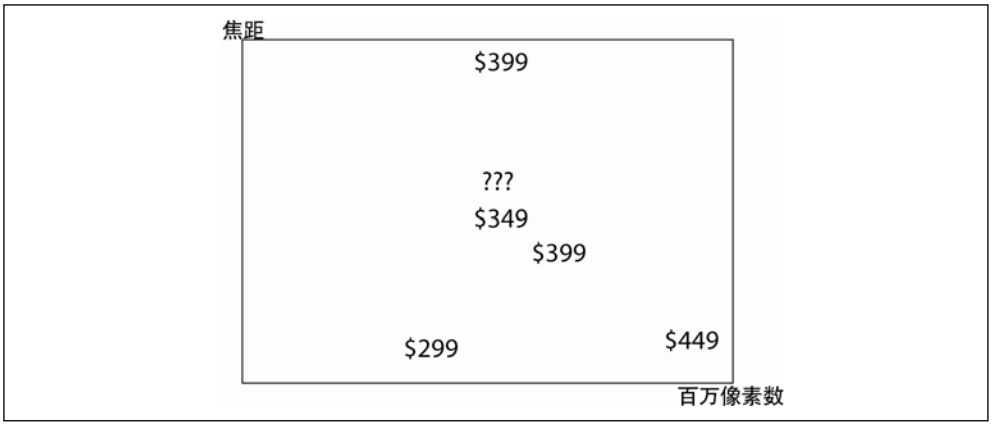

图 12-10：在焦距-百万像素数空间中的相机价格

变量缩放及多余变量

Scaling and Superfluous Variables

迄今为止我们讨论的 kNN 算法存在着一个很大的问题，那就是它在计算距离时考虑了所有的变量。这意味着，如果这些变量衡量的是不同的事物，而其中某个变量的取值又比其他变量大很多时，则该变量会对“接近”的含义形成非常大的影响。试想一下，如果上述数据集是按像素数而非百万像素数给出分辨率——相差 10 倍的变焦对于相机价格造成的影响，理应要比相差 10 个像素的分辨率大许多，但是现在它们会被视作效果相当。除此以外，有时数据集中还会包含一些对预测结果完全不起任何作用的变量，但是它们仍然会对距离的计算构成影响。

上述问题可以通过在计算距离之前对数据进行调整来加以解决。在第 8 章中，我们给出了一种数据调整的方法，该方法对某些变量的数值进行了放大，而对其他变量则做了缩小。由于完全不起作用的变量都被乘以了 0，因而这些变量对结果不再构成任何的影响。那些有价值但值域范围差别很大的变量，则被缩放到了更具可比性的程度——或许我们会将 2 000 像素的差异与 1 倍变焦的差异视作等同。

因为对数据的缩放量取决于具体的应用，所以可以通过对预测算法实施交叉验证，来判断一组缩放因子的优劣程度。交叉验证的做法是，先从数据集中去除一部分数据，然后再利用剩余数据来推测出这部分数据，算法会尝试对推测的效果进行评估。下页图 12-11 给出了交叉验证的工作原理。

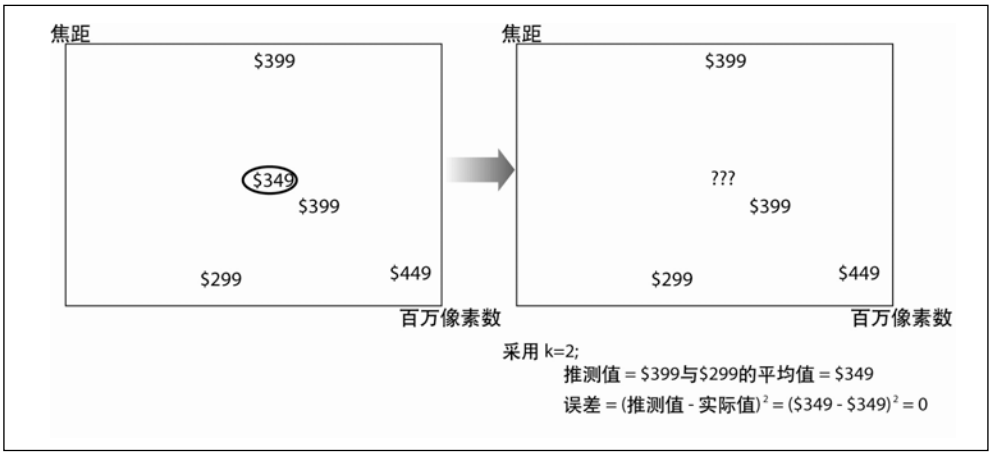

图 12-11：针对某项数据的交叉验证

通过对许多不同的缩放因子进行交叉验证，我们可以得到针对每一项数据的误差率，利用这一误差率，可以判断出：哪些缩放因子应该被用于对新数据的预测。

kNN 代码使用说明

Using Your kNN Code

在第 8 章中，我们给出了实现 kNN 和加权 kNN 算法的函数。将这些代码用于第 293 页表 12-6 给出的示例数据集是非常简单的。

```
>>> cameras=[{'input':(7.1,3.8),'result':399},
... {'input':(5.0,2.4),'result':299},
... {'input':(6.0,4.0),'result':349},
... {'input':(6.0,12.0),'result':399},
... {'input':(10.0,3.0),'result':449}]
>>> import numpredict
>>> numpredict(cameras,(6.0,6.0),k=2)
374.0
>>> numpredict.weightedknn(cameras,(6.0,6.0),k=3)
351.52666892719458
```

对数据的缩放可以有效改善最终的结果。`rescale` 函数可以完成这一任务：

```
>>> scc=numpredict.rescale(cameras,(1,2))
>>> scc
[{'input': [7.1, 7.6], 'result': 399}, {'input': [5.0, 4.8], 'result': 299},
{'input': [6.0, 8.0], 'result': 349}, {'input': [6.0, 24.0], 'result': 399},
{'input': [10.0, 6.0], 'result': 449}]
```

通过使用 `crossvalidate`，我们可以找出表现最好的缩放因子：

```
>>> numpredict.crossvalidate(knn1,cameras,test=0.3,trials=2)
3750.0
>>> numpredict.crossvalidate(knn1,scc,test=0.3,trials=2)
2500.0
```

随着数据集拥有的变量越来越多，寻找合适的缩放因子也会变得越来越乏味，因此我们可以通过循环遍历所有数值来寻找最佳的结果，或者，如第 8 章那样，借助于某种优化算法。

优点和缺点
Strengths and Weaknesses

能够利用复杂函数进行数值预测，同时又保持简单易懂的特点，k-最近邻技术便是少数几种具备如是特征的算法之一。kNN 算法的推导过程很容易理解，并且只要对代码稍做修改，就可以清晰地观察到计算过程中到底使用了哪些近邻。神经网络也是利用复杂函数进行数值预测的，但是它们显然无法为我们提供类似的例子来帮助理解推导的过程。

不仅如此，确定合理的数据缩放量不但可以改善预测的效果，而且还可以告诉我们预测过程中各个变量的重要程度。任何被缩小至 0 的变量都会被舍弃。有些时候，数据的收集也许是十分困难或代价高昂的，而此时，知道有些变量对结果没有任何影响，也许会在日后为我们省去不少的时间和金钱。

kNN 是一种在线（online）技术，这意味着新的数据可以在任何时候被添加进来，这一点不同于以支持向量机为代表的一类技术，后者在数据改变之后必须重新进行训练。而对于 kNN 而言，添加新的数据根本不须要进行任何的计算，只要将数据添加到集合中即可。

kNN 主要的缺点在于，为了完成预测，它要求所有的训练数据都必须缺一不可。面对拥有上百万样本的数据集，这不仅在空间上会有问题，在时间上也是一个问题——为了找到最为接近的数据项，每一项待预测的数据都必须和所有其他数据项进行比较。这一过程对于某些应用而言也许会显得非常低效。

kNN 的另一个缺点是，寻找合理的缩放因子可能是一项乏味的工作。尽管有不少方法可以令这一过程更趋于自动化，但是当面对一个规模庞大的数据集时，为数以千计的缩放因子实施评估和交叉验证，是一项计算量非常巨大的工作。如果有许多不同的变量等待考查，那么我们也许就须要对数百万个不同的缩放因子进行尝试，直到找到正确的因子为止。

聚类
Clustering

分级聚类和 K-均值聚类都属于非监督学习技术，即：它们不要求训练用的数据样本，因为

这些方法不是用来做预测的。在第 3 章中，我们讨论了如何选择一组热门博客并自动对其进行聚类，从中可以发现哪些博客被理所当然地划归到了一类：这些博客或具有相似的描写主题，或使用了相近的词汇。

分级聚类

Hierarchical Clustering

聚类算法可以用于任何具有一个或多个数值属性的数据集。虽然第 3 章的例子中，我们使用了针对不同博客的单词计数，但是任何形式的数字集合都是可以用于聚类算法的。为了说明分级聚类算法的工作原理，请看表 12-7 中的数据项（其中包含了字母表中的一部分字母）及其数值属性。

表 12-7：用于聚类的一个简单表格

项	P1	P2
A	1	8
B	3	8
C	2	6
D	1.5	1
E	4	2

下页图 12-12 给出了针对上述数据项实施聚类的完整过程。在第一幅图中，所有数据项以二维形式分布，P1 对应于 x 轴，P2 对应于 y 轴。分级聚类的工作方式是，寻找两个距离最接近的数据项，然后将它们合二为一。在第二幅图中，我们可以看到两个最靠近的项，A 和 B，已经被合在了一起。新聚类的“位置”等于原来两个数据项位置的均值。在接下来的图中，距离最为接近的两项则变成了 C 和新的 A-B 聚类。这一过程会一直持续下去，直到如最后一幅图所示，每个数据项都被包含在了一个大的聚类中为止。

上述过程形成了一个层级结构，该层级结构可以显示为树状图的形式（dendrogram），树状图是一种类似于树的结构，它可以显示出哪些项和群组是紧靠在一起的。上述样例数据集的树状图如下页图 12-13 所示。

此处，距离最为接近的两项，A 和 B，最终关联在了一起。而 C 则与 A 和 B 的组合关联了起来。从该树状图中，我们可以选出任意一个枝节点，并判断其是否为一个有价值的群组。在第 3 章，我们曾经看到过几乎全部由政治类博客构成的分支，还有由技术类博客构成的分支，诸如此类。

K-均值聚类

K-Means Clustering

另一种对数据进行聚类的方法被称作 K-均值聚类。与分级聚类构造一棵由所有数据项构成的树不同，K-均值聚类实际上是将数据拆分到不同的群组中。它还要求我们在开始执行算法之前给出想要的群组数量。299 页图 12-14 给出了一个 K-均值聚类的实际例子。在这一例子中，我们尝试从一个数据集中寻找两个聚类，此处所用的数据集与分级聚类例子中用到的数据集稍有不同。

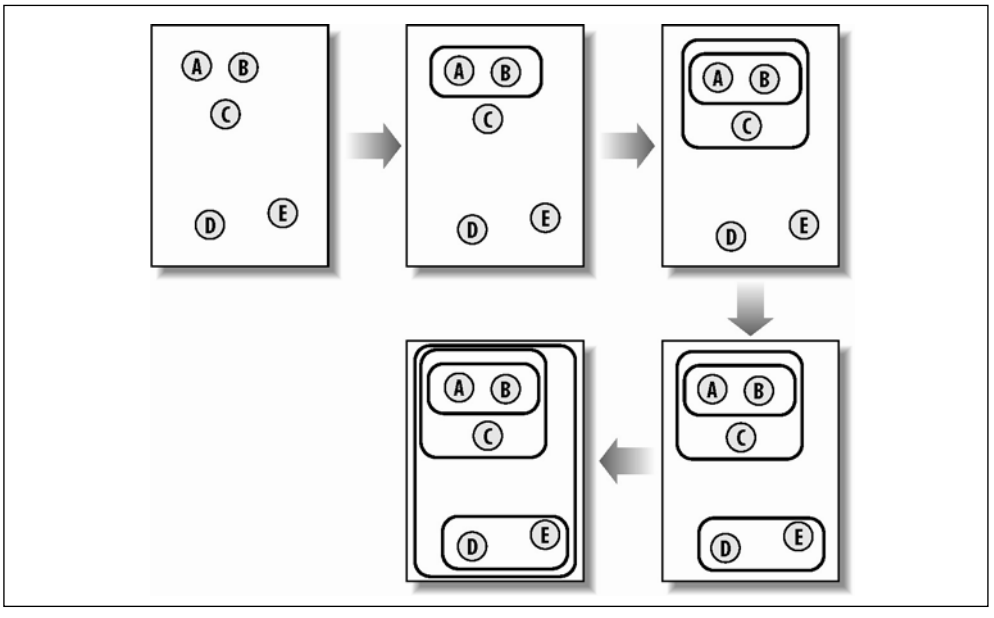

图 12-12：分级聚类的过程

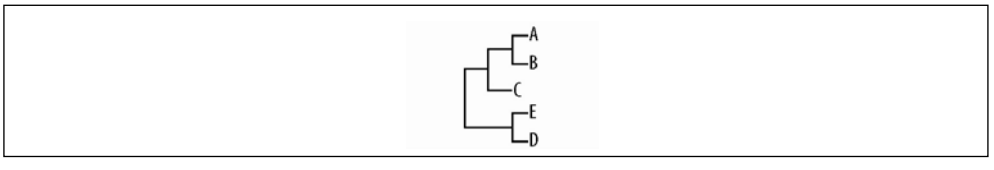

图 12-13：字母经过聚类后得到的树状图

在第一幅图中，两个中心点（以黑圈标示）的位置是随机产生的。在第二幅图中，每个数据项都被分配给了距离最近的中心点——在本例中，A 和 B 被分配给了上方的中心点，而 C、D 和 E 则被分配给了下方的中心点。在第三幅图中，中心位置已经移到了分配给原中心点的所有项的平均位置处。当再次进行分配时，我们可以看到 C 现在距离上方的中心点较之以前更为接近了，而 D 和 E 则依然是距离下方中心点最近的两项。如此，最终所得的结果是：A、B、C 在一个聚类中，而 D 和 E 则在另一个聚类中。

聚类代码使用说明

Using Your Clustering Code

为了进行聚类，我们需要一个数据集和一个距离度量方法。该数据集由一组数字列表构成，其中的每个数字代表一个变量。虽然我们在第 3 章使用了皮尔逊相关度和 Tanimoto 分值作

为距离度量的方法，但是使用其他度量方法也是非常容易的，比如欧几里德距离。

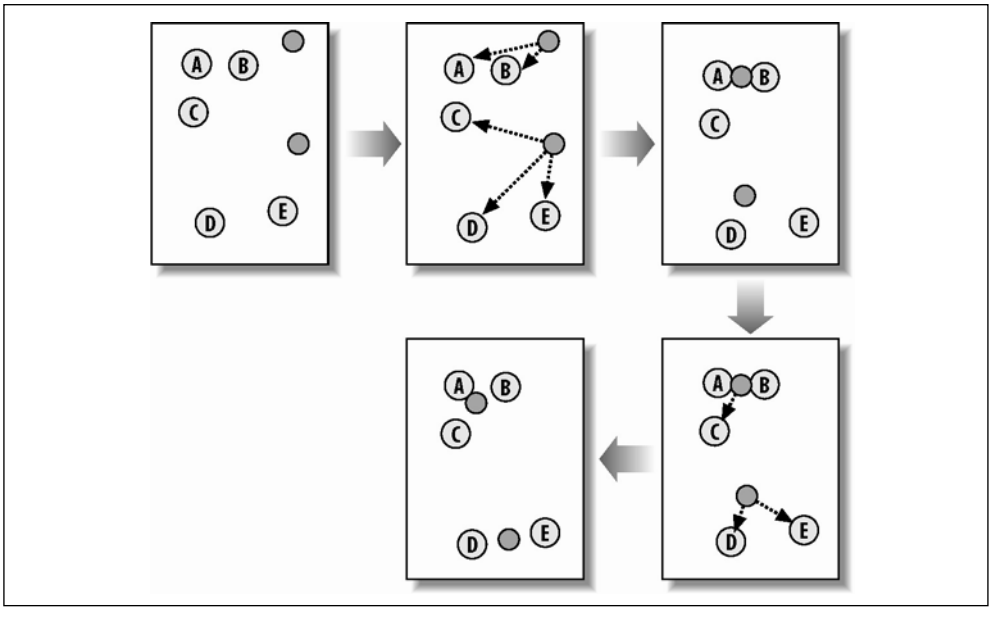

图 12-14：K-均值聚类的过程

```
>>> data=[[1.0,8.0],[3.0,8.0],[2.0,7.0],[1.5,1.0],[4.0,2.0]]
>>> labels=['A','B','C','D','E']
>>> def euclidean(v1,v2): return sum([(v1[i]-v2[i])**2 for i in range(len(v1))])
>>> import clusters
>>> hcl=clusters.hcluster(data,distance=euclidean)
>>> kcl=clusters.kcluster(data,distance=euclidean,k=2)
Iteration 0
Iteration 1
```

对于 K-均值聚类，我们可以轻松地打印出各个聚类中包含的数据项：

```
>>> kcl
[[0, 1, 2], [3, 4]]
>>> for c in kcl: print [labels[l] for l in c]
...
['A', 'B', 'C']
['D', 'E']
```

分级聚类的结果不太适合打印输出，不过我们在第 3 章所给的代码中包含了一个绘制分级聚类树状图的函数：

```
>>> clusters.drawdendrogram(hcl,labels,jpeg='hcl.jpg')
```

到底选择哪一种算法完全取决于所要处理的问题。将数据拆分到不同的群组中——比如通过 K-均值聚类得到的群组——常常是很有价值的，因为这样做更易于打印输出，也更易于识别群组的特征。而另一方面，面对一个全新的数据集，我们也许并不知道自己想要多少

群组，也许只想了解其中有哪些群组彼此最为接近。在这种情况下，采用分级聚类或许是更好的选择。

此外，还可以同时借助于两种方法：先利用 K-均值聚类建立起多个群组，然后再根据中心点的间距对群组实施分级聚类。这样就会得到一系列以树型方式组织的群组，它们位于树上的不同层次，从中我们可以观察到所有群组间的关联关系。

多维缩放

Multidimensional Scaling

在第 3 章的博客例子中，我们还用到了另一种方法，被称为多维缩放。和聚类算法一样，多维缩放也是一种非监督技术，它的作用并不是要做预测，而是要使不同数据项之间的关联程度更易于理解。多维缩放会为数据集构造一个低维度的表达形式，并令距离值尽可能接近于原数据集。对于屏幕或纸张的打印输出，多维缩放通常意味着将数据从多维降至 2 维。

设想一下，比如，我们有一个如表 12-8 所示的 4 维数据集（每一项数据有 4 个相关的值）

表 12-8：一个待缩放的简单的 4 维表格

A	0.5	0.0	0.3	0.1
B	0.4	0.15	0.2	0.1
C	0.2	0.4	0.7	0.8
D	1.0	0.3	0.6	0.0

利用欧几里德距离公式，我们可以得到每两项间的距离值。例如，A和B之间的距离为 $sqrt(0.1^2+0.15^2+0.1^2+0.0^2) = 0.2$。所有数据项两两之间的距离矩阵如表 12-9 所示。

表 12-9：上述示例的距离矩阵

	A	B	C	D
A	0.0	0.2	0.9	0.8
B	0.2	0.0	0.9	0.7
C	0.9	0.9	0.0	1.1
D	0.8	0.7	1.1	0.0

我们的目标是要将所有数据项绘制在一张 2 维图上，从而使各数据项在 2 维空间中的距离尽可能接近于其在 4 维空间中的距离。我们将所有数据项随机放置于图中，并且对各项之间的当前距离进行了计算，如下页图 12-15 所示。

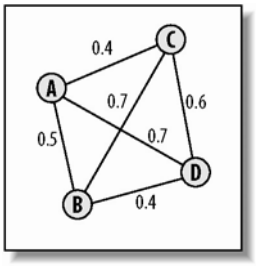

图 12-15：各项之间的距离

针对每一组数据项，我们会将目标距离与当前距离进行比较，并求出误差，然后再根据两者间的误差，将每个数据项的所在位置按比例移近或移远少许量。图 12-16 给出了我们对数据项 A 的施力情况。图中 A 与 B 之间的距离是 0.5，而两者的目标距离则仅为 0.2，因此我们必须将 A 朝 B 的方向移近一点才行。与此同时，我们还将 A 推离了 C 和 D，因为它距离 C 和 D 都太近了。

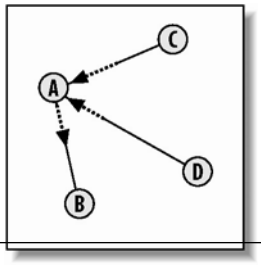

图 12-16：对数据项 A 的施力情况

每个节点的移动，都是所有其他节点施加在该节点上的推或拉的合力造成的。节点每移动一次，其当前距离和目标距离之间的差距就应该会减少一些。这一过程会不断地重复多次，直到无法再通过移动节点来减少总的误差值为止。

多维缩放代码使用说明

Using Your Multidimensional Scaling Code

在第 3 章中，我们提供了两个涉及多维缩放的函数，其中一个是用来实际运行算法的，而另一个则是用来显示结果的。第一个函数，`scaledown`，接受一个以多维形式表达的数据项列表，然后它以相同次序返回同一列表，并将所有数据的维度都降到了 2 维：

```
>>> labels=['A','B','C','D']
>>> scaleset=[[0.5,0.0,0.3,0.1],
... [0.4,0.15,0.2,0.1],
... [0.2,0.4,0.7,0.8],
```

```
... [1.0,0.3,0.6,0.0]]
>>> twod=clusters.scaledown(scaleset,distance=euclidean)
>>> twod
[[0.45, 0.54],
 [0.40, 0.54],
 [-0.30, 1.02],
 [0.92, 0.59]]
```

另一个函数，draw2d，则接受经过降维的列表，并生成一幅图片：

```
>>> clusters.draw2d(twod,labels,jpeg='abcd.jpg')
```

执行上述函数调用将会生成一个名为 *abcd.jpg* 的文件，其中包含了最终的结果。除此以外，我们也可以利用其他程序，比如电子表格软件，对 scaledown 产生的列表进行解读，从而以不同的方式将结果加以可视化。

非负矩阵因式分解

Non-Negative Matrix Factorization

第 10 章我们讨论的是一种被称为非负矩阵因式分解（NMF）的高阶技术，这一技术可以将一组数值型的观测数据拆解成不同的组分。借助这一方法，可以观察到构成新闻故事的各种不同的主题，还可以借此来了解，如何将股票交易量分解成一系列会对单支或多支股票即刻构成影响的新闻事件。同样，这也是一种非监督算法，因为其作用并非预测分类或数值，而是帮助我们识别数据的特征。

为了理解 NMF 的原理，请见表 12-10 所示的这组数据：

表 12-10：用于 NMF 的一个简单表格

观测值序号	A	B
1	29	29
2	43	33
3	15	25
4	40	28
5	24	11
6	29	29
7	37	23
8	21	6

假定观测值 A 和 B 是由两对数字（即特征）的某种组合构成的，但是我们并不知道这些数对到底是多少，也不知道每个数对在构造观测值时的贡献度（即权重）有多大。NMF 能够为我们找到特征和权重的可能取值。对于第 10 章中的新闻故事而言，观测值便是故事，表中的列则是故事中的单词。而对于股票交易量而言，观测值便是交易日期，表中的列则是

各支股票的交易代码。无论是哪一种情况，算法都会试图从中找出一小部分组分，将这些组分以不同数量结合在一起便可以得到此前的观测值。

针对上表中的数据，一种可能的答案是：两个数对分别为(3, 5)和(7, 2)。

借助于得到的这些组分，我们会发现，通过将各个数对以不同的数量加以组合，可以重新构造出原来的观测值，如下所示：

```
5*(3, 5) + 2*(7, 2) = (29, 29)
5*(3, 5) + 4*(7, 2) = (43, 33)
```

我们也可以将其看做是一个矩阵的乘法，如图 12-17 所示。

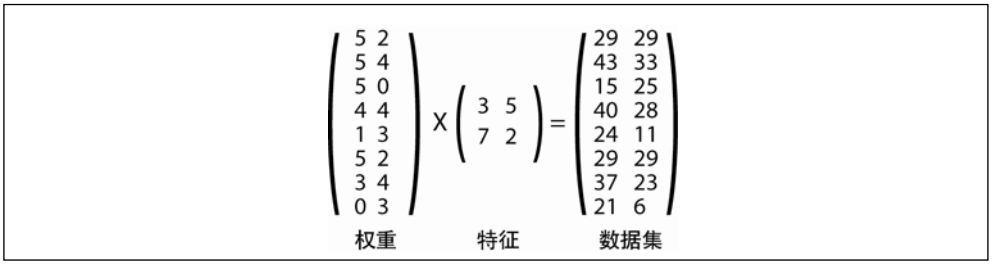

图 12-17：将一个数据集因式分解为权重和特征两个组分

NMF 的目标是要自动找到特征矩阵和权重矩阵。为此，它以随机矩阵开始，并根据一系列更新法则对这些矩阵加以更新。根据法则我们得到了 4 个更新矩阵。在下面的说明中，我们将初始矩阵称为数据矩阵。

hn

经转置后的权重矩阵与数据矩阵相乘得到的矩阵。

hd

经转置后的权重矩阵与原权重矩阵相乘，再与特征矩阵相乘得到的矩阵。

wn

数据矩阵与经转置后的特征矩阵相乘得到的矩阵。

wd

权重矩阵与特征矩阵相乘，再与经转置后的特征矩阵相乘得到的矩阵。

为了更新特征矩阵和权重矩阵，我们首先将上述所有矩阵都转换成数组。然后将特征矩阵中的每一个值与 *hn* 中的对应值相乘，并除以 *hd* 中的对应值。类似地，我们再将权重矩阵中的每一个值与 *wn* 中的对应值相乘，并除以 *wd* 中的对应值。

这一过程会一直重复下去，直到特征矩阵和权重矩阵的乘积与数据矩阵足够接近为止。特征矩阵可以告诉我们潜藏在数据背后的诸多因素，比如：新闻的主题，还有股票市场所发生的重大事件，这些因素的共同组合可以重新构造出我们的数据集来。

NMF 代码使用说明

Using Your NMF Code

要使用 NMF 的代码，我们只须调用 `factorize` 函数，并传入一组观测数据，以及希望找到的潜在特征的数量：

```
>>> from numpy import *
>>> import nmf
>>> data=matrix([[ 29.,  29.],
... [ 43.,  33.],
... [ 15.,  25.],
... [ 40.,  28.],
... [ 24.,  11.],
... [ 29.,  29.],
... [ 37.,  23.],
... [ 21.,   6.]])
>>> weights,features=nmf.factorize(data,pc=2)
>>> weights
matrix([[ 0.64897525,  0.75470755],
        [ 0.98192453,  0.80792914],
        [ 0.31602596,  0.70148596],
        [ 0.91871934,  0.66763194],
        [ 0.56262912,  0.22012957],
        [ 0.64897525,  0.75470755],
        [ 0.85551414,  0.52733475],
        [ 0.49942392,  0.07983238]])
>>> features
matrix([[ 41.62815416,   6.80725866],
        [  2.62930778,  32.57189835]])
```

函数最终会返回权重和特征。或许每次返回的结果都不尽相同，因为对于一个规模不是很大的观测数据集而言，有效的特征可能不止一个。观测数据集的规模越大，返回一致性结果的可能也就越大，尽管这些特征的返回次序也许稍有不同。

优化

Optimization

优化是在第 5 章中介绍的，和其他方法稍有不同，优化不是要处理数据集，而是要尝试找到能够使成本函数的输出结果达到最小化的值。第 5 章给出了成本函数的几个例子，比如：

根据价格和候机时间的组合来安排组团旅游，为学生分配最为适宜的宿舍，以及优化简单网络图的布局。一旦设计好成本函数，我们就可以利用同样的算法来解决上述这三个不同的问题。此处我们讨论其中的两种算法：模拟退火和遗传算法。

成本函数

The Cost Function

成本函数接受一个经推测得到的题解，并返回一个数值结果，该值越大就表示题解的表现越差，该值越小就表示题解的表现越好。优化算法利用该函数对各种题解进行检验，并从中找出最优解。通常，用于优化的成本函数有许多变量须要考虑，而且有时我们并不清楚到底要修改其中的哪个变量，才能使最终结果的改善效果达到最好。不过，为了说明问题，此处我们只考虑包含一个变量的函数，定义如下：

```
y = 1/x * sin(x)
```

图 12-18 给出了该函数的图示。

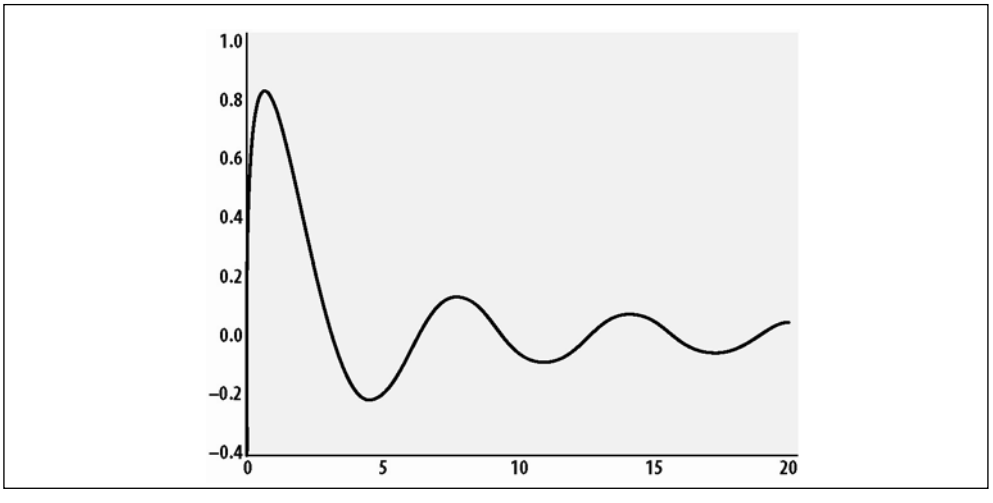

图 12-18：1/x * sin x 的图示

因为上述函数仅有一个变量，所以从图中我们很容易就可以找到函数的最低点。我们将以此来说明优化算法的工作原理；在现实中，当面对一个带有多个变量的复杂函数时，寄希望于将其简单绘制出来以寻找最低点这样的做法是行不通的。

该函数值得关注的一个地方是，它有多个局部最小值。这些点所处的位置要低于周围所有的点，但它们未必是全局意义上的最低点。这意味着，尝试随机选择题解并沿斜坡向下的方法未必一定能够找到最优解，因为我们有可能会陷入一个包含局部最小值的区域，而永远无法找到全局范围内的最小值。

模拟退火

Simulated Annealing

模拟退火，是受物理学领域中合金冷却的启发而提出的，它以一个随机推测的题解开始，然后以此为基准随机选择一个方向，并就近找到另一个近似解，判断其成本值。算法希望借此来改善题解的表现：如果题解的成本变小，则新的题解将取代原来的题解。如果成本较之原来变大了，则新题解取代旧题解的概率就取决于当前的温度值。此处的温度，会以一个相对较高的数值开始缓慢下降。正因如此，算法在执行的早期阶段会更容易接受表现相对较差的题解，这样我们就有效地避免了陷入局部最小值的可能。

当温度到达 0 时，算法便返回当前的题解。

遗传算法

Genetic Algorithms

遗传算法是受进化理论启发而提出的。它以一组被称为种群的随机题解开始。种群中表现最为优异的成员——即成本最低者——会被选中并通过稍事改变（即变异）或特征组合（即交叉或配对）的方式加以修改。随后，我们会得到一个新的种群，称之为下一代。经过连续数代之后，题解最终将会得到相应的改善。

上述过程会一直持续下去，只有当达到了某个阈值，或种群在经历数代之后没有得到任何改善，又或是遗传的代数达到了最大值时，这一过程才会就此终止。算法最终将返回在任何一代种群中发现的最优解。

优化代码使用说明

Using Your Optimization Code

不论上述哪一种算法，我们都须要定义一个成本函数，并确定题解的值域范围。此处的值域就是每个变量可能的取值范围。在这个简单的例子中，我们不妨使用[(0,20)]，即变量取值介于 0 和 20 之间。随后，我们便可以选择调用任何一个优化函数，并传入相应的成本函数及值域范围作为参数：

```
>>> import math
>>> def costf(x): return (1.0/(x[0]+0.1))*math.sin(x[0])
>>> domain=[(0,20)]
>>> optimization.annealingoptimize(domain,costf)
[5]
```

无论处理何种问题，为了对参数进行调整，并在执行时间和题解质量之间取得平衡，或许我们都有必要将优化算法执行多次。当为一组彼此相近的问题构造优化程序时——比如前述的旅游规划，这些问题目标一致但底层细节（此处为飞行时间和价格）却有所不同，可以针对相关参数进行实验，并借此确定出适合此类问题的参数设置，随后便可以始终保持这些参数设置固定不变，只要遇到的问题都同属于一类。

将机器学习、开放 API，以及面向公众的参与模式结合在一起，将会迸发出各种各样的可能性。不仅如此，随着算法的不断精炼，开放 API 的不断涌现，以及越来越多的人们成为在线应用的积极参与者，这种可能性在不久的将来还会持续扩大。希望本书能够对读者构建集体智慧的相关应用有所帮助，并且能够启迪大家寻找更多新的机遇！