第 6 章

文档过滤

Document Filtering

本章将向大家演示如何依据内容来对文档进行分类。文档分类是机器智能（machine intelligence）的一个应用，很有实用价值，而且现在越来越普及。关于文档过滤，最有价值也最为人们所熟知的应用，恐怕要数垃圾邮件过滤了。随着电子邮件的广泛普及与邮件发送的超低成本，人们面临的一大问题是：任何人的邮件地址只要落入不法者之手，便有可能会收到未经许可的商业邮件，致使我们无法阅读到真正感兴趣的邮件。

当然，垃圾信息的问题并非仅限于电子邮件。随着时间的推移，Web 网站已经越来越具有互动的特征了，它们或向用户征求意见，或请求用户提供原创内容，这些行为都会伴以垃圾信息侵扰的问题。例如像 Yahoo! Groups 和 Usenet 这样的公共留言板，就长期遭受着垃圾帖的侵扰。这些帖子或与留言板主题毫不相干，或者就是以兜售可疑产品为目的。现在，博客和维基也遭遇到了同样的问题。每当我们在构建一个允许普通大众一起参与的应用系统时，就始终应该考虑应对垃圾信息的策略。

本章中介绍的算法不是专门针对垃圾信息的。由于这些算法可以解决更为一般性的问题，即学习并鉴别文档所属的分类，因此我们还可以将其应用于一些相比垃圾信息而言不那么令人生厌的问题。一种可能的应用是，根据邮件正文自动将收件箱中的邮件划分为社交类邮件和工作相关类邮件。还有一种可能的应用是，识别出要求回复的邮件，并将其自动转发给最适合的人员进行处理。本章的最后一个例子，会为大家示范如何将来自某一 RSS 订阅源的内容项自动过滤到不同的分类之中。

过滤垃圾信息

Filtering Spam

早期尝试对垃圾信息进行过滤所用的都是基于规则的分类器（rule-based classifiers），使用时会有人事先设计好一组规则，用以指明某条信息是否属于垃圾信息。典型的规则包括：

英文大写字母的过度使用，与医学药品相关的单词，或是过于花哨的 HTML 用色等。基于

规则的分类器，其问题很快就显现了出来——垃圾信息制造者在知道了所有规则以后，为了绕开过滤器，其行为就会变得更加隐蔽；而且人们会发现，如果他们的父母不知道关闭大写锁定键（Caps Lock），一些正常的邮件也会被归类成垃圾邮件。

基于规则的过滤器还有另一个问题——是否被当作垃圾信息很大程度上因其所面对的读者和张贴位置的不同而不同。对于某一位特定用户、公告留言板或维基而言，那些可以用来明确指示是否垃圾信息的关键词，在其他场合下可能就会变得相当正常。为了解决这一问题，本章所要考查的程序会在开始阶段和逐渐收到更多消息之后，根据人们提供给它的有关哪些是垃圾邮件，哪些不是垃圾邮件的信息，不断地进行学习。通过这样的方式，我们可以分别为不同的用户、群组或网站建立起各自的应用实例和数据集，它们对垃圾信息的界定将逐步形成自己的观点。

文档和单词

Documents and Words

即将构造的分类器须要利用某些特征来对不同的内容项进行分类。所谓特征，是指任何可以用来判断内容中具备或缺失的东西。当考虑对文档进行分类时，所谓的内容即是文档，而特征则是文档中的单词。当将单词作为特征时，其假设是：某些单词相对而言更有可能会出现于垃圾信息中。这一假设是大多数垃圾信息过滤器背后所依赖的基本前提。不过，特征未必一定是一个个单词；它们也可以是词组或短语，或者任何可以归为文档中缺失或存在的其他东西。

请新建一个文件，取名 *docclass.py*，并在其中加入一个名为 `getwords` 的函数，以从文本中提取特征：

```
import re
import math

def getwords(doc):
  splitter=re.compile('\\W*')
  # 根据非字母字符进行单词拆分
  words=[s.lower() for s in splitter.split(doc)
         if len(s)>2 and len(s)<20]

  # 只返回一组不重复的单词
  return dict([(w,1) for w in words])
```

该函数以任何非字母类字符为分隔符对文本进行划分，将文本拆分成了一个个单词。这一

过程只留下了真正的单词，并将这些单词全都转换成了小写形式。

决定采用哪些特征颇具技巧性，也十分重要。特征必须具备足够的普遍性，即时常出现，但又不能普遍到每一篇文档里都能找到。理论上，整篇文档的文本都可以作为特征，但是

除非我们一再收到内容完全相同的邮件，否则这样的特征几乎肯定是毫无价值的。在另一种极端情况下，特征也可以是单个字符。但是由于每一封电子邮件中都有可能会出现所有这些字符，因此要想利用这样的特征将希望看到和不希望看到的文档区分开来是很困难的。即便选择使用单词作为特征，也依然还是会带来一些问题，包括如何正确划分单词，哪些标点符号应该被纳入单词，以及是否应该包含头信息（header information）等。

在根据特征进行判断时还有一点须要考虑，那就是如何才能更好地利用特征将一组文档划归到目标分类中去。例如，前述getwords函数的代码通过将单词转换为小写形式，从而减少了特征的总数。这意味着，程序会将位于句首以大写字母开头的单词与位于句中全小写形式的单词视为相同——这样做非常好，因为具有不同大小写形式的同一单词往往代表的含义是相同的。然而，上述函数完全没有考虑到被用于许多垃圾信息中的“SHOUTING风格”（译注1），而这一点可能对区分垃圾邮件和非垃圾邮件是至关重要的。除此以外，如果超过半数以上的单词都是大写时，那就说明必定会有其他的特征存在。

正如你所看到的，在选择特征集时须要做大量的权衡，而且还要不断地进行调整。不过眼下，可以暂且使用这个简单的 `getwords` 函数；在本章的后续部分，我们还将了解到有关特征提取的一些改进方法。

对分类器进行训练

Training the Classifier

本章中讨论的分类器可以通过接受训练的方式来学习如何对文档进行分类。本书中的许多其他算法，例如我们在第 4 章中见到过的神经网络，都是通过读取正确答案的样本进行学习的。如果分类器掌握的文档及其正确分类的样本越多，其预测的效果也就越好。人们专门设计分类器，其目的也就在于此，即：从极为不确定的状态开始，随着分类器不断了解到哪些特征对于分类而言更为重要，其确定性也在逐渐地增加。

我们要做的第一件事情，是编写一个代表分类器的类。这个类将对分类器到目前为止所掌握的信息进行封装。以这样的方式构造 Python 模块的好处在于，我们可以针对不同的用户、群组或查询，建立起多个分类器实例，并分别对它们加以训练，以响应特定群组的需求。请在 *docclass.py* 中新建一个名为 `classifier` 的类：

译注 1：此处是指许多垃圾邮件中所采用的将单词以大写形式书写的手段。

```
class classifier:
  def __init_  _(self,getfeatures,filename=None):
    # 统计特征/分类组合的数量
    self.fc={}
    # 统计每个分类中的文档数量
    self.cc={}
    self.getfeatures=getfeatures
```

该类中有 3 个实例变量，它们分别是 fc、cc 和 getfeatures。变量 fc 将记录位于各分类中的不同特征的数量。例如：

```
{'python': {'bad': 0, 'good': 6}, 'the': {'bad': 3, 'good': 3}}
```

上述示例表明，单词“the”在被划归“bad”类的文档中已经出现了 3 次，而在被划归“good”类的文档中也出现了 3 次。而单词“Python”却只在“good”类的文档中出现过。

变量 cc 是一个记录各分类被使用次数的字典。这一信息是我们稍后即将讨论的概率计算所需的。最后一个实例变量，getfeatures，对应于一个函数，其作用是从即将被归类的内容项中提取出特征来——在本例中，就是我们刚才定义过的 getwords 函数。

类中定义的方法不会直接引用这些字典，因为这会有碍于将训练数据存入文件或数据库的潜在选择。请加入下列辅助函数，以实现计数值的增加和获取：

```
# 增加对特征/分类组合的计数值
def incf(self,f,cat):
  self.fc.setdefault(f,{})
  self.fc[f].setdefault(cat,0)
  self.fc[f][cat]+=1

# 增加对某一分类的计数值
def incc(self,cat):
  self.cc.setdefault(cat,0)
  self.cc[cat]+=1

# 某一特征出现于某一分类中的次数
def fcount(self,f,cat):
  if f in self.fc and cat in self.fc[f]:
    return float(self.fc[f][cat])
  return 0.0

# 属于某一分类的内容项数量
def catcount(self,cat):
  if cat in self.cc:
    return float(self.cc[cat])
  return 0

# 所有内容项的数量
def totalcount(self):
  return sum(self.cc.values())

# 所有分类的列表
def categories(self):
  return self.cc.keys()
```

train 方法接受一个内容项（本例中为文档）和一个分类作为参数。它利用 getfeatures 函数，将内容项拆分为彼此独立的各个特征。然后调用 incf 函数，针对该分类为每个特征增加计数值。最后，函数会增加针对该分类的总计数值：

```
def train(self,item,cat):
  features=self.getfeatures(item)
  # 针对该分类为每个特征增加计数值
  for f in features:
    self.incf(f,cat)

  # 增加针对该分类的计数值
  self.incc(cat)
```

请启动一个新的 Python 会话，并引入该模块，我们可以来检查一下这个类是否可用：

```
$ python
>>> import docclass
>>> cl=docclass.classifier(docclass.getwords)
>>> cl.train('the quick brown fox jumps over the lazy dog','good')
>>> cl.train('make quick money in the online casino','bad')
>>> cl.fcount('quick','good')
1.0
>>> cl.fcount('quick','bad')
1.0
```

此处，我们用一个函数将训练用的样本数据导入到分类器中是很有价值的，因为这样就无须在每次创建分类器的时候再对其进行手工训练了。请将该函数加入 *docclass.py* 的开始处：

```
def sampletrain(cl):
  cl.train('Nobody owns the water.','good')
  cl.train('the quick rabbit jumps fences','good')
  cl.train('buy pharmaceuticals now','bad')
  cl.train('make quick money at the online casino','bad')
  cl.train('the quick brown fox jumps','good')
```

计算概率

Calculating Probabilities

既然我们已经对一封电子邮件在每个分类中的出现次数进行了统计，那么接下来的工作就是要将其转换成概率了。所谓概率，是指一个介于 0 和 1 之间的数字，用以指示某一事件发生的可能性。在本例中，可以用一个单词在一篇属于某个分类的文档中出现的次数，除以该分类的文档总数，计算出单词在分类中出现的概率。

请将一个名为 fprob 的方法加入 classifier 的类中：

```
def fprob(self,f,cat):
  if self.catcount(cat)==0: return 0
```

```
    # 特征在分类中出现的总次数，除以分类中包含内容项的总数
    return self.fcount(f,cat)/self.catcount(cat)
```

我们称上述概率为条件概率，通常记为 *Pr*(A | B) ，读作“在给定 B 的条件下 A 的概率”。在本例中，目前我们所求得的值对应于 *Pr(word | classification)*，即：对于一个给定的分类，某个单词出现的概率。

可以在你的 Python 会话中尝试执行一下该函数：

```
>>> reload(docclass)
<module 'docclass' from 'docclass.py'>
>>> cl=docclass.classifier(docclass.getwords)
>>> docclass.sampletrain(cl)
>>> cl.fprob('quick','good')
0.66666666666666663
```

从执行结果中我们可以看到，在三篇被归类为“good”的文档中，有两篇文档出现了单词“quick”，即：一篇“good”分类的文档中包含该单词的概率为，*Pr(quick | good)* = 0.666（有 2/3 的机会）。

从一个合理的推测开始

Starting with a Reasonable Guess

`fprob` 方法针对目前为止见到过的特征与分类，给出了一个精确的结果。但是它有一个小小的问题——只根据以往见过的信息，会令其在训练的初期阶段，对那些极少出现的单词变得异常敏感。在训练用的样本数据中，单词“money”只在一篇文档中出现过，并且由于这是一则涉及赌博的广告，因此文档被划归为了“bad”类。由于单词“money”在一篇“bad”类的文档中出现过，而任何“good”类的文档中都没有该单词，所以此时利用 `fprob` 计算所得的单词“money”在“good”分类中出现的概率为 0。这样做有一些偏激，因为“money”可能完全是一个中性词，只是恰好先出现在了一篇 “bad”类的文档中而已。伴随着单词越来越多地出现在同属于一个分类的文档中，其对应的概率值也逐渐接近于 0，恐怕这样才会更合理一些。

为了解决上述问题，在我们手头掌握的有关当前特征的信息极为有限时，我们还须要根据一个假设的概率来作出判断。一个推荐的初始值是 0.5。我们还须要确定为假设的概率赋以多大的权重——权重为 1 代表假设概率的权重与一个单词相当。经过加权的概率值返回的是一个由 `getprobability` 与假设概率组成的加权平均。

在单词“money”的例子中，针对“money”的加权概率对于所有分类而言均是从 0.5 开始的。待到在 classifier 训练期间接受了一篇 “bad”分类的文档，并且发现“money”适合于“bad”分类时，

其针对“bad”分类的概率就会变为 0.75。这是因为：

```
(weight*assumedprob + count*fprob)/(count+weight)
= (1*1.0+1*0.5)/(1.0 + 1.0)
= 0.75
```

请将 weightedprob 方法加入 classifier 类中：

```
def weightedprob(self,f,cat,prf,weight=1.0,ap=0.5):
  # 计算当前的概率值
  basicprob=prf(f,cat)

  # 统计特征在所有分类中出现的次数
  totals=sum([self.fcount(f,c) for c in self.categories()])

  # 计算加权平均
  bp=((weight*ap)+(totals*basicprob))/(weight+totals)
  return bp
```

现在我们可以在自己的 Python 会话中尝试执行一下该函数了。由于新建一个 classifier 类的实例将清除其已有的训练数据，因此请重新加载模块，并再次运行 sampletrain 方法：

```
>>> reload(docclass)
<module 'docclass' from 'docclass.pyc'>
>>> cl=docclass.classifier(docclass.getwords)
>>> docclass.sampletrain(cl)
>>> cl.weightedprob('money','good',cl.fprob)
0.25
>>> docclass.sampletrain(cl)
>>> cl.weightedprob('money','good',cl.fprob)
0.16666666666666666
```

正如我们所看到的，随着单词的概率从假设的初始值开始被逐步地“拉动”，重新运行 sampletrain 方法后使 classifier 对各个单词的概率变得更加确信了。

选择 0.5 作为假设的概率初始值仅仅是因为它介于 0 和 1 的正中间。不过，也有可能我们已经掌握了更多的背景信息，从而使假设更加有据可依，这一点即便对于一个完全没有经过训练的分类器而言，也是有可能的。例如，一个准备对垃圾信息过滤器进行训练的人，可以利用他人训练过的垃圾过滤器，将其所得的概率值作为假设的概率初始值。使用者还可以专门为自己设计个性化的垃圾信息过滤器，只是不管怎样，对于一个过滤器而言，它最好应该有能力处理极少会出现的单词。

朴素分类器

A Naïve Classifier

一旦我们求出了指定单词在一篇属于某个分类的文档中出现的概率，就需要有一种方法将各个单词的概率进行组合，从而得出整篇文档属于该分类的概率。本章将分别考查两种不同的分类方法。这两种方法在大多数场合下都是可以使用的，只不过它们在面对特定任务

时，在算法的性能级别上有些微的不同。本节中我们要讨论的分类器被称为朴素贝叶斯分类器。

这种方法之所以被冠以朴素二字，是因为它假设将要被组合的各个概率是彼此独立的。即，一个单词在属于某个指定分类的文档中出现的概率，与其他单词出现于该分类的概率是不相关的。事实上这个假设是不成立的，因为你也许会发现，与有关 Python 编程的文档相比，包含单词“casino”的文档更有可能包含单词“money”。

这意味着，我们无法将采用朴素贝叶斯分类器所求得的结果实际用作一篇文档属于某个分类的概率，因为这种独立性的假设会使其得到错误的结果。不过，我们还是可以对各个分类的计算结果进行比较，然后再看哪个分类的概率最大。在现实中，若不考虑假设的潜在缺陷，朴素贝叶斯分类器将被证明是一种非常有效的文档分类方法。

整篇文档的概率

Probability of a Whole Document

为了使用朴素贝叶斯分类器，首先我们须要确定整篇文档属于给定分类的概率。正如此前讨论过的，我们须要假设概率的彼此独立性，即：可以通过将所有的概率相乘，计算出总的概率值。

例如，假设我们已经注意到有 20%的“bad”类文档中出现了单词“Python”——*Pr(Python | Bad)* = 0.2——同时有 80%的文档出现了单词“casino”（*Pr(Casino | Bad)* = 0.8）。那么，预期两个单词出现于同一篇“bad”类文档中的独立概率为——*Pr(Python & Casino | Bad)*——0.8×0.2 = 0.16。从中我们会发现，计算整篇文档的概率，只须将所有出现与某篇文档中的各单词的概率相乘即可。

请在 *docclass.py* 中，新建一个 `classifier` 的子类，取名 `naivebayes`，并为其添加一个 `docprob` 方法，该方法的作用是提取特征（单词）并将所有单词的概率值相乘以求出整体概率：

```
class naivebayes(classifier):
  def docprob(self,item,cat):
    features=self.getfeatures(item)

    # 将所有特征的概率相乘
    p=1
    for f in features: p*=self.weightedprob(f,cat,self.fprob)
    return p
```

现在我们已经知道了如何计算 *Pr(Document | Category)*，不过只做到这一步还不行。为了对文档进行分类，我们真正需要的是 *Pr(Category | Document)*。换言之，就是对于一篇给定的

文档，它属于某个分类的概率是多少？所幸的是，一位名叫 Thomas Bayes 的英国数学家早在大约 250 年前就已经找到了解决这一问题的办法。

贝叶斯定理简介

A Quick Introduction to Bayes' Theorem

贝叶斯定理是一种对条件概率进行调换求解（flipping around）（译注2）的方法。它通常被写作：

Pr(A | B) = Pr(B | A) x Pr(A)/Pr(B)

在本例中，即为：

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

*Pr(Document | Category)*的计算方法上一节已经介绍过了，但是等式中的另两个值如何计算呢？*Pr(Category)*是随机选择一篇文档属于该分类的概率，因此就是属于该分类的文档数除以文档的总数。

至于 *Pr(Document)*，我们也可以计算它，但这将会是一项不必要的工作。请记住，我们不会将这一计算结果当作真实的概率值。相反，我们会分别计算每个分类的概率，然后对所有的计算结果进行比较。由于不论计算的是哪个分类，*Pr(Document)* 的值都是一样的，其对结果所产生的影响也完全是一样的，因此我们完全可以忽略这一项。

`prob` 方法用于计算分类的概率，并返回 *Pr(Document | Category)*与 *Pr(Category)*的乘积。请将该方法加入 `naivebayes` 类中：

```
def prob(self,item,cat):
  catprob=self.catcount(cat)/self.totalcount()
  docprob=self.docprob(item,cat)
  return docprob*catprob
```

请在 Python 的执行环境中尝试一下该函数，看看针对不同的字符串和分类，概率值是如何变化的：

译注 2：根据后面的公式，此处 flipping around 的意思是通过 *P(B|A)*来求 *P(A|B)*，而 *B|A* 对 *A|B* 而言，二者的相对位置正好调了过来。

```
>>> reload(docclass)
<module 'docclass' from 'docclass.pyc'>
>>> cl=docclass.naivebayes(docclass.getwords)
>>> docclass.sampletrain(cl)
>>> cl.prob('quick rabbit','good')
0.15624999999999997
>>> cl.prob('quick rabbit','bad')
0.050000000000000003
```

根据训练的数据，我们认为相比于“bad”分类而言，短语“quick rabbit”更适合于“good”分类。

选择分类

Choosing a Category

构造朴素贝叶斯分类器的最后一个步骤是实际判定某个内容项所属的分类。此处最简单的方法，是计算被考查内容在每个不同分类中的概率，然后选择概率最大的分类。如果我们只是在试图判断“将内容放到哪里最合适”的问题，那么这不失为一种可行的策略，但是在许多应用中，我们无法将各个分类同等看待，而且在一些应用中，对于分类器而言，承认不知道答案，要好过判断答案就是概率值稍大一些的分类。

在垃圾信息过滤的例子中，避免将普通邮件错当成垃圾邮件要比截获每一封垃圾邮件更为重要。收件箱中偶尔收到几封垃圾邮件还是可以容忍的，但是一封重要的邮件则有可能会因为自动过滤到废件箱而被完全忽视。假如我们必须在废件箱中找回自己的重要邮件，那就真的没必要再使用垃圾信息过滤器了。

为了解决这一问题，我们可以为每个分类定义一个最小阈值。对于一封将要被划归到某个分类的新邮件而言，其概率与针对所有其他分类的概率相比，必须大于某个指定的数值才行。这一指定的数值就是阈值。以垃圾邮件过滤为例，假如过滤到“bad”分类的阈值为 3，则针对“bad”分类的概率就必须至少 3 倍于针对 “good”分类的概率才行。假如针对“good”分类的阈值为 1，则对于任何邮件，只要概率确实大于针对“bad”分类的概率，它就是属于“good”分类的。任何更有可能属于“bad”分类 ,但概率并没有超过 3 倍以上的邮件 ,都将被划归到“未知”分类中。

为了定义阈值，请修改初始化方法，在 `classifier` 中加入一个新的实例变量：

```
def __init_  _(self,getfeatures):
  classifier.__init_  _(self,getfeatures)
  self.thresholds={}
```

请加入几个用于设值和取值的简单方法，令其默认返回为 1.0：

```
def setthreshold(self,cat,t):
  self.thresholds[cat]=t

def getthreshold(self,cat):
  if cat not in self.thresholds: return 1.0
  return self.thresholds[cat]
```

现在，我们可以构建 `classify` 方法了。该方法将计算每个分类的概率，从中得出最大值，并将其与次大概率值进行对比，确定是否超过了规定的阈值。如果没有任何一个分类满足上述条件，方法就返回默认值。请将该方法加入 `classifier` 中：

```
def classify(self,item,default=None):
  probs={}
  # 寻找概率最大的分类
```

```
    max=0.0
    for cat in self.categories():
      probs[cat]=self.prob(item,cat)
      if probs[cat]>max:
        max=probs[cat]
        best=cat

    # 确保概率值超出域值*次大概率值
    for cat in probs:
      if cat==best: continue
      if probs[cat]*self.getthreshold(best)>probs[best]: return default
    return best
```

大功告成！现在我们已经建立起了一个完整的文档分类系统。通过构造不同的特征提取方法，我们可以对该系统进行扩展，以实现对任何其他内容的分类。请在你的 Python 会话中试验一下这一分类器：

```
>>> reload(docclass)
<module 'docclass' from 'docclass.pyc'>
>>> cl=docclass.naivebayes(docclass.getwords)
>>> docclass.sampletrain(cl)
>>> cl.classify('quick rabbit',default='unknown')
'good'
>>> cl.classify('quick money',default='unknown')
'bad'
>>> cl.setthreshold('bad',3.0)
>>> cl.classify('quick money',default='unknown')
'unknown'
>>> for i in range(10): docclass.sampletrain(cl)
...
>>> cl.classify('quick money',default='unknown')
'bad'
```

当然，我们还可以修改一下阈值，看看对结果有何影响。一些垃圾信息过滤插件为用户提供了控制阈值的功能，这样一来，只要当前阈值令太多的垃圾邮件进入到收件箱中，或者有大量正常邮件被错归为了垃圾邮件，我们就可以对阈值进行调整。当然，对于另一些涉及文档过滤的应用而言，阈值的定义也可能有所不同；与上述情况不一样，有时，所有分类可能都是平等的，而有时，将内容过滤到“未知”分类则是不可接受的。

费舍尔方法

The Fisher Method

以 R. A. Fisher 的名字命名的费舍尔方法，是前面介绍的朴素贝叶斯方法的一种替代方案，它可以给出非常精确的结果，尤其适合垃圾信息过滤。*SpamBayes*，一个用 Python 编写的 Outlook 插件，便采用了这一方法。与朴素贝叶斯过滤器利用特征概率来计算整篇文档的概

率不同，费舍尔方法为文档中的每个特征都求得了分类的概率，然后又将这些概率组合起

来，并判断其是否有可能构成一个随机集合。该方法还会返回每个分类的概率，这些概率彼此间可以进行比较。尽管这种方法更为复杂，但是因为它在为分类选择临界值（cutoff）时允许更大的灵活性，所以还是值得一学的。

针对特征的分类概率

Category Probabilities for Features

前面讨论过的朴素贝叶斯过滤器，将所有 *Pr(feature | category)*的计算结果组合起来得到了整篇文档的概率，然后再对其进行调换求解。在本节中，我们将直接计算当一篇文档中出现某个特征时，该文档属于某个分类的可能性，也就是 *Pr(category | feature)*。如果单词“casino”出现于 500 篇文档中，并且其中有 499 篇属于“bad”分类，则“casino”属于“bad”分类的概率将非常接近于 1。

计算 *Pr(category | feature)*的常见方法是：

> （具有指定特征的属于某分类的文档数）/（具有指定特征的文档总数）

上述计算公式并没有考虑我们收到属于某一分类的文档可能比其他分类更多的情况。假如我们有许多“good”分类的文档，而“bad”分类的文档则很少，那么一个出现于所有“bad”类文档中的单词，即便邮件内容看上去可能没有问题，该单词属于“bad”分类的概率也依然会更大一些。如果我们假设“未来将会收到的文档在各个分类中的数量是相当的”，那么上述方法就会有更好的表现，因为这使得它们能更有效地利用特征来识别分类。

为了进行归一化计算，函数将分别求得 3 个量：

- 属于某分类的概率 *clf = Pr(feature | category)*
- 属于所有分类的概率 *freqsum = Pr(feature | category)*之和
- *cprob = clf / (clf+nclf)*

请在 *docclass.py* 中为 `classifier` 新建一个子类，取名 `fisherclassifier`，并加入如下方法：

```
class fisherclassifier(classifier):
  def cprob(self,f,cat):
    # 特征在该分类中出现的频率
    clf=self.fprob(f,cat)
    if clf==0: return 0

    # 特征在所有分类中出现的频率
    freqsum=sum([self.fprob(f,c) for c in self.categories()])

    # 概率等于特征在该分类中出现的频率除以总体频率
    p=clf/(freqsum)

    return p
```

基于各分类中所包含的内容项数量相当的假设，该函数返回的概率值，代表了具备指定特征的内容属于指定分类的可能性。可以在你的 Python 会话中看一下这些概率的实际计算结果：

```
>>> reload(docclass)
>>> cl=docclass.fisherclassifier(docclass.getwords)
>>> docclass.sampletrain(cl)
>>> cl.cprob('quick','good')
0.57142857142857151
>>> cl.cprob('money','bad')
1.0
```

上述方法告诉我们，包含单词“casino”的文档是垃圾邮件的概率为 0.9。这与训练数据是相符的，不过这种方法同样也会遇到前文提到的问题——因为算法接触单词的次数太少，所以它有可能会对概率值估计过高。因此，不妨像前文那样，对概率进行加权处理，即：所有概率值均以 0.5 作为初始值，而后伴随不断的训练过程，允许它们逐渐向其他概率值变化。

```
>>> cl.weightedprob('money','bad',cl.cprob)
0.75
```

将各概率值组合起来

Combining the Probabilities

现在，我们须要将对应各个特征的概率值组合起来，形成一个总的概率值。理论上，我们可以将它们连乘起来，利用相乘的结果在不同分类间进行比较。当然，由于特征不是彼此独立的，因此它们并不代表真实的概率，不过这已经比我们在前一节中构造的贝叶斯分类器要好不少了。由费舍尔方法返回的结果是对概率的一种更好的估计，这对于结果报告或临界值判断而言是非常有价值的。

费舍尔方法的计算过程是将所有概率相乘起来，然后取自然对数（Python 中的 *math.log*），再将所得结果乘以-2。请将下列方法加入 `fisherclassifier` 类中，以实现这一计算过程：

```
def fisherprob(self,item,cat):
  # 将所有概率值相乘
  p=1
  features=self.getfeatures(item)
  for f in features:
    p*=(self.weightedprob(f,cat,self.cprob))

  # 取自然对数，并乘以-2
  fscore=-2*math.log(p)

  # 利用倒置对数卡方函数求得概率
  return self.invchi2(fscore,len(features)*2)
```

费舍尔方法告诉我们，如果概率彼此独立且随机分布，则这一计算结果将满足对数卡方分布（chi-squared distribution）。也许我们会预料到，不属于某个分类的内容项中，可能会包含针对该分类的不同特征概率的单词（可能会随机出现）；或者，一个属于该分类的内容项中会包含许多概率值很高的特征。通过将费舍尔方法的计算结果传给倒置对数卡方函数，我们会得到一组随机概率中的最大值。

请将倒置对数卡方函数加入 `fisherclassifier` 类中：

```
def invchi2(self,chi,df):
  m = chi / 2.0
  sum = term = math.exp(-m)
  for i in range(1, df//2):
     term *= m / i
     sum += term
  return min(sum, 1.0)
```

我们依然可以在自己的 Python 会话中试验该函数，看看费舍尔方法是如何对样本字符串进行评价的：

```
>>> reload(docclass)
>>> cl=docclass.fisherclassifier(docclass.getwords)
>>> docclass.sampletrain(cl)
>>> cl.cprob('quick','good')
0.57142857142857151
>>> cl.fisherprob('quick rabbit','good')
0.78013986588957995
>>> cl.fisherprob('quick rabbit','bad')
0.35633596283335256
```

正如我们所看到的，结果总是介于 0 和 1 之间。这些结果本身即是衡量文档所属分类的一种很好的度量方法。正是由于这一点，分类器本身才有可能变得更为有效。

对内容项进行分类

Classifying Items

我们可以利用 `fisherprob` 的返回值来决定如何进行分类。不像贝叶斯过滤器那样须要乘以阈值，此处我们可以为每个分类指定下限。尔后，分类器会返回介于指定范围内的最大值。在垃圾信息过滤器中，我们可以将“bad”分类的下限值设得很高，比如 0.6；将“good”分类的下限值设置得很低，比如 0.2。这样做可以将正常邮件被错归到“bad”分类的可能性减到最小，同时也会允许少量垃圾邮件进入到收件箱中。任何针对“good”分类的分值低于 0.2，针对“bad”分类的分值低于 0.6 的邮件，都将被划归到“未知”分类中。

请在 fisherclassifier 类中新建一个 init 方法，再增加一个保存临界值的变量：

```
def __init_  _(self,getfeatures):
  classifier.__init_  _(self,getfeatures)
  self.minimums={}
```

请将下述两个用于设值和取值的方法加入类中，默认取值为 0：

```
def setminimum(self,cat,min):
  self.minimums[cat]=min

def getminimum(self,cat):
  if cat not in self.minimums: return 0
  return self.minimums[cat]
```

最后，再添加一个方法，用以计算每个分类的概率，并找到超过指定下限值的最佳结果：

```
def classify(self,item,default=None):
  # 循环遍历并寻找最佳结果
  best=default
  max=0.0
  for c in self.categories():
    p=self.fisherprob(item,c)
    # 确保其超过下限值
    if p>self.getminimum(c) and p>max:
      best=c
      max=p
  return best
```

现在我们可以针对测试数据，利用费舍尔评价方法试验一下分类器了。请在你的 Python 会话中输入如下代码：

```
>>> reload(docclass)
<module 'docclass' from 'docclass.py'>
>>> docclass.sampletrain(cl)
>>> cl.classify('quick rabbit')
'good'
>>> cl.classify('quick money')
'bad'
>>> cl.setminimum('bad',0.8)
>>> cl.classify('quick money')
'good'
>>> cl.setminimum('good',0.4)
>>> cl.classify('quick money')
>>>
```

此处的执行结果与朴素贝叶斯分类器的结果类似。人们相信，在实践中费舍尔分类器对垃圾信息的过滤效果会更好；只不过对于这样一小组训练数据而言，过滤的效果可能不太明显。应该使用何种分类器要取决于你的应用，没有一种简单方法可以预测出什么样的分类器会更好，或者我们应该使用多大的临界值。所幸的是，利用此处给出的代码，我们应该

能够很容易地对两种算法以及各种不同的设置项进行试验。

将经过训练的分类器持久化

Persisting the Trained Classifiers

在任何真实世界的应用中，所有的训练和分类工作都不太可能完全在一次会话中完成。如果分类器被用作 Web 应用的一部分，那么我们就有可能须要将用户在使用系统期间所产生的任何与训练相关的数据保存起来，然后在下一次用户登录之后再将数据恢复。

使用 SQLite

Using SQLite

本节中我们将为大家示范如何利用数据库（本例中为 SQLite）将分类器的训练信息进行持久化。如果我们的应用涉及许多用户同时对分类器进行训练和查询，那么将计数值存入数据库可能是一个明智之举。SQLite 就是我们曾在第 4 章中使用过的数据库。如果你还没有用过 pysqlite，则须要先将其下载并安装；有关下载和安装的详细情况请见附录 A。通过 Python 访问 SQLite 与访问其他数据库是很类似的，因此如果要进行数据库移植应该也非常的容易。

为了将 pysqlite 引入进来，请将下列语句加入 *docclass.py* 的首部：

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

本节中的代码将当前 `classifier` 类中所用的字典结构都替换为了一个持久化的数据存储结构。请在 `classifier` 中添加一个方法，为该分类器打开数据库，并在必要时执行建表操作。这些数据表与它们所替换的字典在结构上是相匹配的：

```
def setdb(self,dbfile):
  self.con=sqlite.connect(dbfile)
  self.con.execute('create table if not exists fc(feature,category,count)')
  self.con.execute('create table if not exists cc(category,count)')
```

如果我们正打算将分类器移植到另一个数据库上，为了能够在所使用的目标系统上正常运行，有可能须要修改相应的建表语句。

我们还须要替换所有用于获取和累加计数值的辅助函数：

```
def incf(self,f,cat):
  count=self.fcount(f,cat)
  if count==0:
    self.con.execute("insert into fc values ('%s','%s',1)"
                     % (f,cat))
  else:
    self.con.execute(
      "update fc set count=%d where feature='%s' and category='%s'"
      % (count+1,f,cat))

def fcount(self,f,cat):
  res=self.con.execute(
    'select count from fc where feature="%s" and category="%s"'
    %(f,cat)).fetchone()
```

```
    if res==None: return 0
    else: return float(res[0])

  def incc(self,cat):
    count=self.catcount(cat)
    if count==0:
      self.con.execute("insert into cc values ('%s',1)" % (cat))
    else:
      self.con.execute("update cc set count=%d where category='%s'"
                  % (count+1,cat))

  def catcount(self,cat):
    res=self.con.execute('select count from cc where category="%s"'
                    %(cat)).fetchone()
    if res==None: return 0
    else: return float(res[0])
```

获取所有分类的列表与文档总数的方法也应该被替换掉：

```
  def categories(self):
    cur=self.con.execute('select category from cc');
    return [d[0] for d in cur]

  def totalcount(self):
    res=self.con.execute('select sum(count) from cc').fetchone();
    if res==None: return 0
    return res[0]
```

最后，我们须要在训练结束之后添加一条提交语句，以便在所有计数值被更新之后程序能将数据存入数据库。请将下列代码行加入 `classifier` 中 `train` 方法的末尾处：

```
    self.con.commit()
```

大功告成！在对 `classifier` 初始化之后，我们须要调用 `setdb` 方法，并传入数据库文件的名称。所有训练数据都将被自动存入数据库中，并且能够为任何其他人所使用。我们甚至可以将取自某一分类器的训练数据用于另一种类型的分类器：

```
>>> reload(docclass)
<module 'docclass' from 'docclass.py'>
>>> cl=docclass.fisherclassifier(docclass.getwords)
>>> cl.setdb('test1.db')
>>> docclass.sampletrain(cl)
>>> cl2=docclass.naivebayes(docclass.getwords)
>>> cl2.setdb('test1.db')
>>> cl2.classify('quick money')
u'bad'
```

过滤博客订阅源

Filtering Blog Feeds

为了在真实环境下试验分类器，也为了演示其不同的用途，我们可以将分类器应用于来自某个博客或RSS订阅源的内容项。为此，我们需要用到曾在第3章中介绍过的 Universal Feed Parser。如果你还没有下载相应的函数库，则可以通过访问 *http://feedparser.org* 进行下载。有关安装 Feed Parser 的更多信息请见附录 A。

尽管博客的内容中未必会包含垃圾信息，但是在众多博客所包含的文章中，并非所有的文章都是我们感兴趣的。这也许是因为我们只希望阅读属于某个分类的文章，或者某位作者所撰写的文章，不过通常而言实际情况要比这更为复杂。同样地，我们也可以针对自己感兴趣和不感兴趣的内容定义一些专门的规则——也许我们阅读了一个有关小件装置（gadget）的博客，并且对其中包含单词“cell phone”的内容不感兴趣——但是，假如利用前面已经构造好的分类器来为我们得出上述这些规则，其所需的工作量相对而言会更少一些。

对一个 RSS 订阅源中的内容项进行分类的好处在于，假如我们使用了像 Google Blog Search 这样的博客搜索工具，那么就可以在订阅源的阅读器中对搜索的结果进行定制了。许多人以此来追踪产品和他们感兴趣的内容，甚至还包括他们自己的名字。但是我们会发现，试图利用流量来赚钱的垃圾博客和那些毫无价值的博客也有可能会出现在这些搜索结果当中。

尽管许多订阅源因为拥有的内容项太少而无法进行任何有效的训练，不过在本例中，我们还是可以根据自己的喜好来选择任何的订阅源。在这个特定的例子里，我们使用 Google Blog Search 对单词“Python”进行搜索，其搜索结果都是 RSS 形式的。你可以从 *http://kiwitobes.com/feeds/python_search.xml* 处下载到这些结果。

请新建一个名为 *feedfilter.py* 的文件，并加入下列代码：

```
import feedparser
import re

# 接受一个博客订阅源的 URL 文件名并对内容项进行分类
def read(feed,classifier):
  # 得到订阅源的内容项并遍历循环
  f=feedparser.parse(feed)
  for entry in f['entries']:
    print
    print '-----'
    # 将内容项打印输出
    print 'Title:     '+entry['title'].encode('utf-8')
    print 'Publisher: '+entry['publisher'].encode('utf-8')
    print
    print entry['summary'].encode('utf-8')


    # 将所有文本组合在一起，为分类器构建一个内容项
    fulltext='%s\n%s\n%s' % (entry['title'],entry['publisher'],entry['summary'])
```

```
    # 将当前分类的最佳推测结果打印输出
    print 'Guess: '+str(classifier.classify(fulltext))

    # 请求用户给出正确分类，并据此进行训练
    cl=raw_input('Enter category: ')
    classifier.train(fulltext,cl)
```

该函数循环遍历所有内容项，并利用分类器得到有关分类的最佳推测结果。它向用户给出最佳推测，并接着询问正确的分类是什么。当我们使用一个新的分类器运行该程序时，起初的推测结果将会带有随机性，但是它们会随着时间的推移逐步得到改善。

上述构建好的分类器是完全通用的。尽管我们利用了垃圾信息过滤的例子来帮助说明每段代码的工作原理，但是分类的类别则可以是任何形式的内容。如果你正在使用*python_search.xml*，那么其中也许包含了 4 个分类——一个是关于编程语言的，一个是关于电影《Monty Python》的，一个是关于蟒蛇的，还有一个则是涉及任何其他内容的。请在你的 Python 会话中试着运行一下这个交互式的过滤器，设置好一个分类器，并将其传给`feedfilter`：

```
>>> import feedfilter
>>> cl=docclass.fisherclassifier(docclass.getwords)
>>> cl.setdb('python_feed.db')  # 仅当你使用的是 SQLite
>>> feedfilter.read('python_search.xml',cl)

-----
Title:     My new baby boy!
Publisher: Shetan Noir, the zombie belly dancer! - MySpace Blog

This is my new baby, Anthem. He is a 3 and half month old ball <b>python</b>,
orange shaded normal pattern. I have held him about 5 times since I brought him
home tonight at 8:00pm...
Guess: None
Enter category: snake

-----
Title:     If you need a laugh...
Publisher: Kate&#39;s space

Even does 'funny walks' from Monty <b>Python</b>. He talks about all the ol'
Guess: snake
Enter category: monty

-----
Title:     And another one checked off the list..New pix comment ppl
Publisher: And Python Guru - MySpace Blog

Now the one of a kind NERD bred Carplot male is in our possesion. His name is Broken
(not because he is sterile) lol But check out the pic and leave one
Guess: snake
Enter category: snake
```

我们会发现，推测的结果随着时间的推移在逐渐的改善。由于没有太多有关于蛇的样本信息，尤其是它们被进一步划分成了宠物蛇和时尚一类的帖子，因此分类器对于这一分类的推测结果时常是错误的。当执行完整个训练之后，我们就可以得到针对于指定特征的概率值了——包括针对给定分类的单词概率，以及针对给定单词的分类概率：

```
>>> cl.cprob('python','prog')
0.33333333333333331
>>> cl.cprob('python','snake')
0.33333333333333331
>>> cl.cprob('python','monty')
0.33333333333333331
>>> cl.cprob('eric','monty')
1.0
>>> cl.fprob('eric','monty')
0.25
```

从上述结果中我们可以看到，由于每个内容项都包含单词“python”，因此该单词的概率被等分了。在涉及《Monty Python》的内容项中，有 25%的文章包含了单词“Eric”，而其他内容项中则没有出现该单词。因此就“Eric”而言，对于给定分类的单词概率为 0.25，而对于给定单词的分类概率则为 1.0。

对特征检测的改进

Improving Feature Detection

目前为止的所有例子中，建立特征列表的函数只是简单地使用了非字母非数字类字符作为分隔符对单词进行拆分。函数还将所有单词都转换成了小写形式，因此我们没有办法检测大写单词的过度使用问题。有几种不同的方法可以对其加以改进。

- 不真正区分大写和小写的单词，而是将“含有许多大写单词”这样的现象作为一种特征。
- 除了单个单词以外，还可以使用词组。
- 捕获更多的元信息，如：是谁发送了电子邮件，或者一篇博客被提交到了哪个分类下，可以将这样的信息标示为元信息。
- 保持 URL 和数字原封不动，不对其进行拆分。

请记住，这不仅是让特征更有针对性这么简单。特征必须出现于多篇文档之中，因为它们对分类器而言起了很大的作用。

`classifier` 类可以接受任何形式的函数作为 `getfeatures`，它就传入的内容项运行该函

数，并预期返回一个针对该内容项的包含所有特征的列表或字典。由于这种通用性，我们可以轻松地建立起一个函数，令其处理比简单的字符串而言更为复杂的类型。例如，当对一个博客订阅源的内容项进行分类时，我们可以编写一个函数，令其接受整篇文章的内容，而非从中提取出来的文本，然后标注出各个单词的来源。我们还可以从文本正文中找出词

组，而从主题中找出个别的单词。此外，对记录文章创建者的字段进行拆分可能也是毫无意义的，因为名叫“John Smith”的人所提交的内容，是不太可能会告诉我们任何有关其他叫John 的人所提交的内容的。

请将这个新的特征提取函数加入 *feedfilter.py* 中。请注意，它需要的是一个订阅源的内容项作为参数，而非字符串：

```
def entryfeatures(entry):
  splitter=re.compile('\\W*')
  f={}

  # 提取标题中的单词并进行标示
  titlewords=[s.lower() for s in splitter.split(entry['title'])
        if len(s)>2 and len(s)<20]
  for w in titlewords: f['Title:'+w]=1

  # 提取摘要中的单词
  summarywords=[s.lower() for s in splitter.split(entry['summary'])
        if len(s)>2 and len(s)<20]

  # 统计大写单词
  uc=0
  for i in range(len(summarywords)):
    w=summarywords[i]
    f[w]=1
    if w.isupper(): uc+=1

    # 将从摘要中获得的词组作为特征
    if i<len(summarywords)-1:
      twowords=' '.join(summarywords[i:i+1])
      f[twowords]=1

  # 保持文章创建者和发布者名字的完整性
  f['Publisher:'+entry['publisher']]=1

  # UPPERCASE 是一个“虚拟”单词，用以指示存在过多的大写内容
  if float(uc)/len(summarywords)>0.3: f['UPPERCASE']=1

  return f
```

上述函数从文档和摘要中提取单词，就如同此前的 `getwords` 那样。它将所有位于标题中的单词标识出来，并将其作为特征。位于摘要中的单词以及前后连贯的词组，也被当作了特征。函数还将未经拆分的内容创建者和发布者当作了特征。最后，它统计了摘要中大写单词的出现次数。一旦有超过 30% 的单词为大写形式，函数就会在特征集中加入这一特征，并取名为“UPPERCASE”。与认为“大写单词代表着某种特殊情况”的规则不同，这只是一个附加的特征，分类器可以利用该特征来进行训练——而在某些场合下，分类器也可能会认为

这一特征对于区分文档分类而言是完全没有用处的。

如果希望将这一新函数与 filterfeed 结合使用，我们就须要修改代码，将内容项而非全文作为参数传给分类器。请将函数的末尾处修改如下：

```
# 将当前分类的最佳推测结果打印输出
print 'Guess: '+str(classifier.classify(entry))

# 请求用户给出正确分类，并据此进行训练
cl=raw_input('Enter category: ')
classifier.train(entry,cl)
```

随后，我们就可以初始化分类器，并将 entryfeatures 用作特征提取函数了：

```
>>> reload(feedfilter)
<module 'feedfilter' from 'feedfilter.py'>
>>> cl=docclass.fisherclassifier(feedfilter.entryfeatures)
>>> cl.setdb('python_feed.db')  # 仅当你使用的是 DB 版的代码
>>> feedfilter.read('python_search.xml',cl)
```

关于特征，我们还有许多工作可做。前文构造的基本框架允许我们定义自己的特征提取函数，并设置分类器令其使用该函数。分类器将对任何传入的对象进行分类，只要我们指定的特征提取函数能够根据对象返回一组特征即可。

使用 Akismet

Using Akismet

Akismet 与本章介绍的有关文本分类算法的研究稍有些偏离，不过对于特定类型的应用而言，使用 Akismet 可以花费最小的代价满足你对垃圾信息过滤的需求，同时也免去了自己构造分类器的需要。

Akismet 是作为 WordPress 的一个插件发展而来的，它允许人们向其报告提交到各自博客上的垃圾评论，并与他人报告的垃圾评论进行相似度对比，对新提交的评论进行过滤。目前这些 API 是开放的，因此我们可以向 Akismet 发起任何字符串查询请求，以获知 Akismet 是否认为该字符串属于垃圾信息。

我们要做的第一件事情是获得一个 Akismet 的 API 密钥，可以从 *http://akismet.com* 获取到该密钥。这些密钥对于个人用途而言是免费的，此外还有一些针对商业用途的密钥可供选择。 Akismet API 是通过常规的 HTTP 请求进行调用的，相应的函数库已经被写成了各种不同的语言。本节中用到的函数库可以从 *http: //kemayo.wordpress.com/2005/12/02/akismet-py* 处下载到。请下载 *akismet.py*，并将其与你的代码放入同一目录下，或者也可以将其放入

Python 库所在的目录下。

API 的用法非常简单。请新建一个名为 *akismettest.py* 的文件，并加入下列函数：

```
import akismet

defaultkey = "YOURKEYHERE"
pageurl="http://yoururlhere.com"
```

```
defaultagent="Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.0.7) "
defaultagent+="Gecko/20060909 Firefox/1.5.0.7"

def isspam(comment,author,ipaddress,
          agent=defaultagent,
          apikey=defaultkey):
  try:
    valid = akismet.verify_key(apikey,pageurl)
    if valid:
      return akismet.comment_check(apikey,pageurl,
        ipaddress,agent,comment_content=comment,
        comment_author_email=author,comment_type="comment")
    else:
      print 'Invalid key'
      return False
  except akismet.AkismetError, e:
      print e.response, e.statuscode
      return False
```

现在，我们已经拥有了一个可以接受任何字符串的可供调用的方法，我们可以调用该函数来判断传入的字符串是否与博客评论中的内容相类似。请在你的 Python 会话中尝试一下：

```
>>> import akismettest
>>> msg='Make money fast! Online Casino!'
>>> akismettest.isspam(msg,'spammer@spam.com','127.0.0.1')
True
```

请以不同的用户名、代理和 IP 地址进行试验，观察结果如何变化。

由于 Akismet 的主要用途是对提交到博客上的垃圾评论进行判断，因此它也许并不适合处理其他类型的文档，如电子邮件。而且，与前述分类器不同的是，它不允许你对传入的参数做任何的调整，我们也无法洞悉其求解答案的具体计算过程。不过，Akismet 对于垃圾评论的过滤而言还是非常准确的，而且假如我们的应用正在不断地遭受到相似种类的垃圾信息的骚扰，那么 Akismet 是值得一试的，因为与我们可能搜集到的数据量相比，Akismet 拥有一个相当巨大的对比用文档集。

替代方法

Alternative Methods

本章中介绍的两个分类器都是监督型学习方法（supervised learning methods）的例子，这是一种利用正确结果接受训练并逐步作出更准确预测的方法。第 4 章中介绍过的用于对搜索结果进行排名的人工神经网络是另一个监督型学习的例子。通过将特征作为输入，并令输出代表每一种可能的分类，我们也可以将神经网络用于本章中的相同问题。同样地，第 9

章中介绍的支持向量机（support vector machines），也可以用于解决本章中的问题。

贝叶斯分类器之所以经常被用于文档分类的原因是，与其他方法相比它所要求的计算资源更少。一封电子邮件可能包含数百甚至数千个单词，与训练相应规模大小的神经网络相比，简单地更新一下计数值所需占用的内存资源和处理器时钟周期会更少。而且正如你所看到的，这些工作完全可以在一个数据库中完成。神经网络是否会成为一种可行的替代方案，取决于训练和查询所要求的速度，以及实际运行的环境。神经网络的复杂性导致了其在理解上的困难。在本章中，我们可以清楚地看到单词的概率，以及它们对最终分值的实际贡献有多大，而对于网络中两个神经元之间的连接强度而言，则并不存在同样简单的解释。

另一方面，与本章中所介绍的分类器相比，神经网络和支持向量机有一个很大的优势：它们可以捕捉到输入特征之间更为复杂的关系。在贝叶斯分类器中，每个特征都有一个针对各分类的概率值，将这些概率组合起来之后就得到了一个整体上概率值。在神经网络中，某个特征的概率可能会依据其他特征的存在或缺失而改变。也许你正在试图阻止有关在线赌博的垃圾信息，但是又对跑马很感兴趣，在这种情况下，只有当电子邮件中的其他地方没有出现单词“horse”时，单词 “casino”才被认为是“bad”的。朴素贝叶斯分类器无法捕获这样的相互依赖性，而神经网络却是可以的。

练习

Exercises

1. 改变假设概率　请修改 `classifier` 类，使其能够支持针对不同特征的不同假设概率。修改 `init` 方法，使其能够接受其他分类器作为参数，并从一个更合理的假设概率推测值（而不是 0.5）开始。

2. 计算 *Pr(Document)*　在朴素贝叶斯分类器中，*Pr(Document)*的计算被略过了，因为它对于比较概率值而言并不是必需的。在特征彼此独立的前提下，事实上利用 *Pr(Document)* 来计算整体概率值是可行的。应该如何计算 *Pr(Document)*呢？

3. POP-3 电子邮件过滤器　Python 有一个用于下载电子邮件的库，叫做 *poplib*。请编写一段脚本，从服务器下载电子邮件，并尝试对其进行分类。一封电子邮件包含有哪些不同的属性？你将如何利用这些属性来构建特征提取函数呢？

4. 任意长度的短语　本章为你示范了提取词组和单个单词的方法。请修改代码令特征提取过程变得可配置，使其能够一次提取出一组拥有指定数量的单词，并将之作为一个独立的特征。

5. 保留 IP 地址　IP 地址、电话号码，以及其他数字信息可能有助于对垃圾信息的识别。请修改特征提取函数，使其将这些信息作为特征加以返回（IP 地址中包含有句号，但是你依然须要剔除句子间的句号）。

6. 其他虚拟特征　有许多像 UPPERCASE 那样的虚拟特征，这些特征可能对文档分类很有帮助。篇幅过长的文档或长单词占据优势的情况也有可能是一种线索。请将这些情况也作为特征。你还能想到其他情况吗？

7. 神经网络分类器　请修改第 4 章中的神经网络，利用它对文档进行分类。如何对神经网络的输出结果进行比较？请编写一个程序对文档进行分类，并对其进行上千次的训练。记录每一种算法执行所需的时间。如何对这些算法作出对比呢？