第23章 实践：垃圾邮件过滤器

2002年，Paul Graham在把Viaweb卖给Yahoo之后腾出一些时间写了《一个处理垃圾邮件的计划》的随笔[①]，这引起了垃圾邮件过滤技术的一场小革命。在Graham的文章发表之前，多数垃圾邮件过滤器都基于手工编写的规则：如果标题中带有×××，那么它可能是垃圾邮件；如果正文的一行中有三个或更多词是全部大写的，那么它可能是垃圾邮件。Graham花了几个月时间试图编写这样一个基于规则的过滤器，并最终意识到这是一个非常折磨人的任务。

为了找出垃圾邮件的每一个特点，你必须站在垃圾邮件发送者的角度来思考。而且坦白地说，我想在这件事上花尽可能少的时间。

为了避免站在垃圾邮件发送者的角度来思考，Graham决定尝试从非垃圾邮件中区分垃圾邮件，通过统计哪些词出现在哪一类邮件中来做到这点。这个过滤器将持续跟踪特定单词出现在垃圾邮件和正常邮件中的频率，然后基于该频率分析新邮件中的单词，从而计算出该邮件是垃圾邮件还是正常邮件。他把这个方法称为贝叶斯（Bayesian）过滤，这种统计技术可以将个别词汇的频率组合成一个整体上的可能性。[②]

23.1 垃圾邮件过滤器的核心

在本章里，你将实现一个垃圾邮件过滤引擎的核心。我们不会编写完整的垃圾邮件过滤程序，而是会把精力集中在对新邮件进行分类以及训练过滤器上。

这个应用比较大，因此有必要定义一个包来避免名字冲突。在从本书网站下载到的源代码中，我使用了包名COM.GIGAMONKEYS.SPAM，这个包同时用到标准COMMON-LISP包和来自第15章的COM.GIGAMONKEYS.PATHNAMES包：

```
(defpackage :com.gigamonkeys.spam
  (:use :common-lisp :com.gigamonkeys.pathnames))
```

任何含有这个应用程序代码的文件都应当以下面这行开始：

① 此文可从http://www.paulgraham.com/spam.html找到，也收录在*Hackers & Painters:Big Ideas from the Computer Age*（O'Reilly，2004）一书中。

② 关于Graham所描述的技术是否真的是“贝叶斯”一直以来有些不同的看法。不过这个名字已经广为流传并成为谈论垃圾邮件过滤时“统计”的代名词。

```
(in-package :com.gigamonkeys.spam)
```

你可以使用相同的包名或者将com.gigamonkeys替换成你所控制的某个域。[1]

还可以通过在REPL中输入相同的形式来切换到这个包，从而测试你所编写的函数。在SLIME中这将使提示符从CL-USER>变成SPAM>，像下面这样：

```
CL-USER> (in-package :com.gigamonkeys.spam)
#<The COM.GIGAMONKEYS.SPAM package>
SPAM>
```

定义了包以后，就可以开始实际的编码工作了。你要实现的主函数有一个简单的任务——接受一封邮件的文本作为参数并将该邮件分类成垃圾邮件、有用信息或不确定。实现这个基本函数很简单，可以通过后面要编写的其他函数来定义它。

```
(defun classify (text)
  (classification (score (extract-features text))))
```

从里向外，分类邮件的第一步是从文本中提取出特征词并传递给score函数。score将计算出一个值，该值随后通过函数classification分成三类——垃圾邮件、有用信息或不确定中的一类。在这三个函数中，classification是最简单的。可以假设score在邮件为垃圾邮件时返回一个接近1的值，对正常邮件返回接近0的值，而在不确定时返回接近0.5的值。

因此你可以像下面这样实现classification：

```
(defparameter *max-ham-score* .4)
(defparameter *min-spam-score* .6)

(defun classification (score)
  (cond
    ((<= score *max-ham-score*) 'ham)
    ((>= score *min-spam-score*) 'spam)
    (t 'unsure)))
```

函数extract-features也几乎是一样直接，尽管它需要更多一些的代码。目前你所提取的特征是出现在文本中的单词。对于每个单词，需要跟踪它在垃圾邮件中出现的次数和在正常邮件中出现的次数。为方便地将这些数据与单词本身保存在一起，可以定义一个类word-feature，该类带有三个槽。

```
(defclass word-feature ()
  ((word
    :initarg :word
    :accessor word
    :initform (error "Must supply :word")
    :documentation "The word this feature represents.")
   (spam-count
    :initarg :spam-count
    :accessor spam-count
    :initform 0
    :documentation "Number of spams we have seen this feature in.")
   (ham-count
    :initarg :ham-count
```

① 尽管如此，并不推荐你使用一个以com.gigamonkeys开头的包来分发该应用，因为你并不控制那个域。

```
    :accessor ham-count
    :initform 0
    :documentation "Number of hams we have seen this feature in.")))
```

把所有的特征数据库保存在一个哈希表中，可以方便地查找代表给定特征的对象。可以定义一个特殊变量*feature-database*来保存对这个哈希表的引用。

```
(defvar *feature-database* (make-hash-table :test #'equal))
```

应当使用**DEFVAR**而不是**DEFPARAMETER**来定义它，因为你不希望你在开发过程中重新加载了含有该定义的文件后被*feature-database*重置，你不想丢失那些保存在*feature-database*中的数据。当然，这意味着如果你确实想要清空这个特征数据库，那你不能只是重新求值DEFVAR形式。所以应该定义一个clear-database函数。

```
(defun clear-database ()
  (setf *feature-database* (make-hash-table :test #'equal)))
```

为了查找给定邮件中的特征，代码要提取出每一个词，继而在*feature-database*中查找对应的word-feature对象。如果*feature-database*不含有这个特征，那么它将创建一个代表该词的新word-feature。可以将这些逻辑封装在一个函数intern-feature中，它接受一个单词并返回对应的特征，必要时会创建它。

```
(defun intern-feature (word)
  (or (gethash word *feature-database*)
      (setf (gethash word *feature-database*)
            (make-instance 'word-feature :word word))))
```

可以使用正则表达式从邮件文本中提取出某个词。例如，使用Edi Weitz编写的Common Lisp可移植的Perl兼容的正则表达式（CL-PPCRE）库，extract-words的代码可以这样写：[①]

```
(defun extract-words (text)
  (delete-duplicates
   (cl-ppcre:all-matches-as-strings "[a-zA-Z]{3,}" text)
   :test #'string=))
```

现在为了实现extract-features需要将extract-words和intern-feature放在一起。由于extract-words返回了一个字符串的列表，而你想要的是一个列表中的每个字符串都被转换成对应的word-feature对象，因此正好可以使用**MAPCAR**。

```
(defun extract-features (text)
  (mapcar #'intern-feature (extract-words text)))
```

可以在REPL中测试这些函数：

```
SPAM> (extract-words "foo bar baz")
("foo" "bar" "baz")
```

同时确保**DELETE-DUPLICATES**能够工作：

```
SPAM> (extract-words "foo bar baz foo bar")
("baz" "foo" "bar")
```

还可以测试extract-features。

① CL-PPCRE的一个版本包含于本书的源代码中，或者你可以从Weitz的站点http://www.weitz.de/cl-ppcre/下载。

```
SPAM> (extract-features "foo bar baz foo bar")
(#<WORD-FEATURE @ #x71ef28da> #<WORD-FEATURE @ #x71e3809a>
 #<WORD-FEATURE @ #x71ef28aa>)
```

正如你所看到的，打印任意对象的默认方法输出的信息太简单。对于这个程序来说，最好可以更清晰地打印出word-feature对象。第17章提到过，所有对象的打印都是由广义函数**PRINT-OBJECT**来实现的。因此，为了改变word-feature的打印方式，你只需定义一个特化在word-feature上的**PRINT-OBJECT**方法。为了让这样的方法实现起来更简单，Common Lisp提供了一个宏**PRINT-UNREADABLE-OBJECT**。[①]

PRINT-UNREADABLE-OBJECT基本形式如下所示：

```
(print-unreadable-object (object stream-variable &key type identity)
  body-form*)
```

其中的*object*参数是一个求值到被打印对象的表达式。在**PRINT-UNREADABLE-OBJECT**主体中，*stream-variable*被绑定到一个流，可以向其中打印你想要的任何东西。打印到该流中的任何东西都将被**PRINT-UNREADABLE-OBJECT**输出并封装在不可读对象的标准语法#<>中。[②]

通过关键字参数*type*和*identity*，**PRINT-UNREADABLE-OBJECT**还可以让你包含对象的类型和一个对象标识的指示。如果它们是非NIL的，那么输出将以对象类的名字开始并以对象的标识结束，这与**STANDARD-OBJECT**的默认**PRINT-OBJECT**方法所打印的形式相似。对于word-feature来说，需要定义一个**PRINT-OBJECT**方法来包含其类型而不是标识，并同时带有word、ham-count和spam-count等槽的值。这个方法如下所示：

```
(defmethod print-object ((object word-feature) stream)
  (print-unreadable-object (object stream :type t)
    (with-slots (word ham-count spam-count) object
      (format stream "~s :hams ~d :spams ~d" word ham-count spam-count))))
```

再在REPL中测试extract-features时，可以更清楚地看到那些被提取出的特征。

```
SPAM> (extract-features "foo bar baz foo bar")
(#<WORD-FEATURE "baz" :hams 0 :spams 0>
 #<WORD-FEATURE "foo" :hams 0 :spams 0>
 #<WORD-FEATURE "bar" :hams 0 :spams 0>)
```

23.2 训练过滤器

现在有了跟踪单独特征的方式，几乎可以开始实现score了。但首先你需要编写用来训练垃圾邮件过滤器的代码，这样score才会有数据可用。为此要定义一个函数train，它接受一些文本和一个指示邮件类型（ham或spam）的符号，然后递增文本中出现的所有特征的ham或spam

① 使用**PRINT-UNREADABLE-OBJECT**的主要原因是，在某人试图可读地打印你的对象时，它会负责报一个适当的错误，例如在使用**FORMAT**指令~S时。

② **PRINT-UNREADABLE-OBJECT**也会在打印控制变量***PRINT-READABLY***为真时报错。这样，一个完全由**PRINT-UNREADABLE-OBJECT**形式组成的**PRINT-OBJECT**方法将正确实现遵守***PRINT-READABLY***协议的**PRINT-OBJECT**。

计数器，它们代表目前所处理过的有用信息或垃圾邮件信息的全局计数。你可以再次采用自顶向下的方法通过其他尚不存在的函数来实现它。

```
(defun train (text type)
  (dolist (feature (extract-features text))
    (increment-count feature type))
  (increment-total-count type))
```

你已经编写了extract-features，因此下一个需要编写的是increment-count，它接受一个word-feature和一个邮件类型并递增该特征的相应槽。由于递增这些计数器的逻辑对于不同类型的对象不应变化，因此可以将它写成一个正规函数。[1]因为你将ham-count和spam-count都定义成带有一个:accessor选项，因此可以使用**INCF**和由**DEFCLASS**创建的访问函数来递增相应的槽。

```
(defun increment-count (feature type)
  (ecase type
    (ham (incf (ham-count feature)))
    (spam (incf (spam-count feature)))))
```

其中的**ECASE**构造是**CASE**的一个变体，两者都类似于源自Algol语言的case语句（在C和它的后裔中重命名成switch了）。它们都求值其第一个参数——键形式，然后找出第一个元素（键）**EQL**相等的子句。在本例中，这意味着当变量type被求值时，得到了作为increment-count的第二个参数所传递的值。

键不会被求值。换句话说，type的值将与Lisp读取器作为**ECASE**形式一部分所读取的字面对象进行比较。在这个函数中，这意味着键是符号ham和spam，而不是任何名为ham和spam的变量的值。因此，如果increment-count像这样

```
(increment-count some-feature 'ham)
```

被调用，那么type的值将是符号ham，而**ECASE**的第一个分支将被求值并且对应特性的ham计数将会递增。另一方面，如果它像这样

```
(increment-count some-feature 'spam)
```

被调用，那么第二个分支将运行，从而递增spam计数。这里符号ham和spam在调用increment-count时被加了引号，否则它们就会作为变量被求值。但是当它们出现在**ECASE**中时却没加引号，因为**ECASE**不对键求值。[2]

---

① 如果你以后决定需要为不同的类编写不同版本的increment-feature，那么你可以将increment-count重定义成一个广义函数，而将该函数定义成一个特化在word-feature上的方法。

② 从技术上来讲，一个**CASE**或**ECASE**的每个子句中的键都将被解释成一个列表指示符，一个指定了对象列表的对象。一个单一的非列表对象被当作列表指示符对待时相当于一个只含有一个对象的列表，而一个列表将指代它本身。这样，每个子句可以有多个键，**CASE**和**ECASE**将选择键列表中含有键形式的值的子句。例如，如果你想要把good作为ham的同义词而把bad作为spam的同义词，那么你可以像下面这样来编写increment-count：

```
(defun increment-count (feature type)
  (ecase type
    ((ham good) (incf (ham-count feature)))
    ((spam bad) (incf (spam-count feature)))))
```

ECASE中的E代表“无遗漏的”（exhaustive）或“错误”（error），意味着当键值是任何列出的键之外的东西时，ECASE应当报错。正常的CASE相对宽松，当没有匹配的子句时返回NIL。

为了实现increment-total-count，需要决定将计数保存在哪里。目前，使用两个特殊变量*total-spams*和*total-hams*会比较好。

```
(defvar *total-spams* 0)
(defvar *total-hams* 0)

(defun increment-total-count (type)
  (ecase type
    (ham (incf *total-hams*))
    (spam (incf *total-spams*))))
```

应当使用DEFVAR来定义这两个变量，理由与用在*feature-database*时相同——它们将在你运行程序期间始终保持其中的数据，你不想只是因为在开发过程中重新加载了你的代码就扔掉这些数据。但是你希望在重置*feature-database*之后可以顺便重置这两个变量。因此，应当按照如下方式在clear-database中添加几行：

```
(defun clear-database ()
  (setf
   *feature-database* (make-hash-table :test #'equal)
   *total-spams* 0
   *total-hams* 0))
```

23

23.3 按单词来统计

一个统计型垃圾邮件过滤器的核心当然是那些基于统计结果计算概率的函数。讨论这些计算的数学原理[①]超出了本书的范围，有兴趣的读者可以参考Gray Robinson的几篇论文。[②]我将把焦点集中在它们是怎样实现的。

统计计算的起点是测量值的集合——保存在*feature-database*、*total-spams*和*total-hams*中的频率数据。假设用于训练的邮件集合在统计上有代表性，那么可以将观察到的频率视为未来邮件中同样特征在有用信息和垃圾邮件信息中出现的概率。

分类邮件的基本方法是提取其中的特征，计算含有该特征的邮件是垃圾邮件的概率，然后再

① 从数学的角度来说，本章中有时对概率一词较宽松的用法可能会冒犯严肃的统计学家。不过，由于即便是该用法的赞成者，其中还进一步划分成贝叶斯论者和频率论者，也无法对概率究竟是什么达成统一意见，因此我不会担心这一点。毕竟这是一本关于编程而不是统计学的书。

② 本章参考的Robinson的文章有“A Statistical Approach to the Spam Problem”（发表在*Linux Journal*上并可从http://www.linuxjournal.com/article.php?sid=6467获得，一个简化版本发表在Robinson的博客http://radio. weblogs.com/0101454/stories/2002/09/16/spamDetection.html上）以及“Why Chi? Motivations for the Use of Fisher's Inverse Chi-Square Procedure in Spam Classification”（可从http://garyrob.blogs.com/whychi93.pdf获得）。另一篇可能有用的文章是“Handling Redundancy in Email Token Probabilities”（可从http://garyrob.blogs.com//handlingtokenredundancy94.pdf获得）。SpamBayes项目（http://spambayes.sourceforge.net/）的存档邮件列表里也含有许多关于测试垃圾过滤器的不同算法和思想的有用信息。

将所有这些概率综合成该邮件的一个整体评分，带有许多垃圾邮件特征和很少有用特征的邮件将得到接近于1的评分，而带有许多有用特征和很少垃圾邮件特征的邮件的评分会接近于0。

第一个统计函数用来计算一个含有给定特征的邮件是垃圾邮件的概率。从某种角度看，一个含有该特征的给定邮件是垃圾邮件的概率，就是含有该特征的垃圾邮件与含有该特征的所有邮件的比值。这样，就可以用如下方式来计算它：

```
(defun spam-probability (feature)
  (with-slots (spam-count ham-count) feature
    (/ spam-count (+ spam-count ham-count))))
```

该函数的值可能被任何邮件是一封垃圾邮件或有用邮件的总体概率所影响。例如，假设你通常获得的正常邮件是垃圾邮件的9倍。那么一个完全中立的特征将在每9个正常邮件后出现在一个垃圾邮件里，从而这个函数计算出1/10的垃圾邮件概率。

但你更感兴趣的是一个给定特征出现在一封垃圾邮件中的概率，与收到垃圾邮件或正常邮件的整体概率无关。这样，你需要将垃圾邮件数量除以接受训练的垃圾邮件总数，将有用邮件数量除以有用邮件总数。为了避免发生除零错误，如果*total-spams*或*total-hams*两者任何一个为零，那么你应当将相应的频率视为零。（很明显，如果垃圾邮件或有用消息的任一总数为零，那么相应的每特征计数也将是零，因此你可以将结果频率视为零而不会带来不良影响。）

```
(defun spam-probability (feature)
  (with-slots (spam-count ham-count) feature
    (let ((spam-frequency (/ spam-count (max 1 *total-spams*)))
          (ham-frequency (/ ham-count (max 1 *total-hams*))))
      (/ spam-frequency (+ spam-frequency ham-frequency)))))
```

这个版本还有另一个问题——它没有在每单词概率上计入到达并分析的消息数量。假设你已经训练了2000条消息，一半是垃圾邮件而另一半正常。现在考察两个只出现在垃圾邮件中的特征。一个出现在所有1000条邮件中，而另一个仅出现一次。根据当前spam-probability的定义，两个特征的出现预测一个邮件是垃圾邮件的概率是相等的，都是1。

尽管如此，那个仅出现一次的特征很可能实际上是一个中性的特征，它很明显在无论是垃圾邮件还是有用信息中都很罕见，在2000条消息中仅出现一次。如果你训练了另外2000条邮件，它很可能又出现了一次，这次出现在一条正常邮件中，使得它突然成为了垃圾邮件可能性为0.5的一条中性特征。

所以看起来你想要计算一个概率，它以某种方式影响了进入到每个特征中的数据点数。Robinson在他的论文中推荐了一个基于贝叶斯概念的函数，将观察到的数据与先验知识或假设相合并。基本上，你以一个假设的先验概率开始计算新的概率并在添加新信息之前给假设的概率一个权重。Robinson的函数如下所示：

```
(defun bayesian-spam-probability (feature &optional
                                  (assumed-probability 1/2)
                                  (weight 1))
  (let ((basic-probability (spam-probability feature))
        (data-points (+ (spam-count feature) (ham-count feature))))
    (/ (+ (* weight assumed-probability)
```

```
    (* data-points basic-probability))
 (+ weight data-points))))
```

Robinson建议把1/2作为assumed-probability的值，把1作为weight的值。使用这些值，一个出现在一条垃圾邮件中而没有出现在有用邮件中的特征具有0.75的bayesian-spam-probability，一个出现在10条垃圾邮件而没有出现在有用邮件中的特征具有大约0.955的bayesian-spam-probability，而一条匹配了1000条垃圾邮件却没有有用邮件的特征将具有大约0.9995的垃圾邮件概率。

23.4 合并概率

现在你可以计算在一条消息中所找到的每一个单独特征的bayesian-spam-probability，实现score函数的最后一步是找出一种方式，将大量的概率个体合并成介于0和1之间的单个值。

如果单独的特征概率是彼此无关的，那么从数学上来讲可以将它们相乘从而得到一个合并的概率。但是它们实际上不可能是彼此无关的，特定的特征很可能会一起出现，而其他一些却从不这样。[①]

Robinson提议使用由统计学家R. A. Fisher发明的概率组合方法。在不讨论为什么它的技术可以奏效的具体细节的前堤下，方法是这样的：首先你通过将所有概率相乘来把它们组合在一起。这给了你一个接近于0的远低于最初概率集合中概率的值，然后取该值的对数并乘以-2。Fisher在1950年证明，如果这些单独的概率是彼此无关的并且来自于0和1之间的统一分布，那么得到的值将满足卡方（chi-square，χ^2）分布。该值和概率数量的两倍可以输入到一个反向卡方分布函数中，然后返回一个反映了通过组合相同数量的随机选择概率得到越来越大的值的可能性。当这个反向卡方分布函数返回一个较低的概率时，这意味着在单独的概率中存在相当多的低概率。（要么是许多相对的低概率，要么是少量非常低的概率。）

为了使用这个概率来检测一个给定的邮件是否是垃圾邮件，你从一个空假设（null hypothesis）开始，一个你想要击倒的稻草人。这个空假设是被分类的消息事实上只是一个特性的随机集合。如果它是的话，那么单独的概率，即每个特征出现在一个垃圾邮件中的可能性也将是随机的。这就是说，一个特征的随机选择将通常含有一些经常出现在垃圾邮件中的特征和另一些很少出现在垃圾邮件中的特征。如果你根据Fisher的方法合并这些随机选择的概率，那么你将得到一个中间的合并值，然后反向卡方分布函数将很可能返回一个比较高的值，事实也正是如此。但如果反向卡方分布返回了一个非常低的概率，这意味着产生该合并值的那些概率不太可能是随机选择的，而可能是里面有太多的低概率值。因此你可以拒绝这个空假设而采纳另一个替代假设：所有引入的特征来自一个有偏的样本——一个带有少量高垃圾邮件概率特征和许多低垃圾邮件特征的样本。换句话说，它一定是条有用的邮件。

尽管如此，Fisher方法并不是对称的，因为反向卡方分布函数对于由给定数量的随机选择的

① 从技术上来讲，对一些事实上无关的概率进行非无关的概率合并，这称为原生贝叶斯（Naive Bayesian）。Graham最初发表的建议本质上是一个原生贝叶斯分类器，其中带有一些“经验驱动”的常量因子。

概率返回的合并后的概率，将比你从合并实际概率中得到的值大得多。这种非对称性的用法对你是有利的，因为当你拒绝空假设时你知道更好的假设是什么。当你用Fisher方法合并单独的垃圾邮件概率时，而它告诉你有很高的概率表明空假设是错误的——邮件并不是一个单词的随机集合，那么这意味着该邮件很可能是有用的。返回的数值就算并非该邮件是有用邮件的字面概率，至少也是对它有用程度的一个好的衡量。相反，对于单独的有用概率的Fisher合并可以给你关于该邮件垃圾邮件程度的一个衡量。

为了得到一个最终的评分，你需要将这两个指标合并成单一的值，从而给一个范围是从0到1的组合的有用程度-垃圾邮件程度评分。Rabinson所推荐的方法是将有用程度和垃圾邮件程度之间差异的一半与1/2相加，换句话说，就是垃圾邮件程度和1减去有用程度的平均值。这在两个评分相反（高的垃圾邮件程度和低的有用程度，或者反过来）时可以带来很好的效果，这时你将得到一个接近0或1的强烈的指示值。但是当垃圾邮件程度和有用程度的评分都高或都低时，你将得到一个接近1/2的最终值，从而得到一个“不确定”的分类。

实现这一模型的score函数如下所示：

```
(defun score (features)
  (let ((spam-probs ()) (ham-probs ()) (number-of-probs 0))
    (dolist (feature features)
      (unless (untrained-p feature)
        (let ((spam-prob (float (bayesian-spam-probability feature) 0.0d0)))
          (push spam-prob spam-probs)
          (push (- 1.0d0 spam-prob) ham-probs)
          (incf number-of-probs))))
    (let ((h (- 1 (fisher spam-probs number-of-probs)))
          (s (- 1 (fisher ham-probs number-of-probs))))
      (/ (+ (- 1 h) s) 2.0d0))))
```

你接受一个特征的列表并循环它们，构建起两个概率的列表，一个列出含有每个特征的消息是垃圾邮件的概率，而另一个列出含有每个特征的邮件是有用邮件的概率。作为一项优化，你也可以在循环过程中统计概率的数量，然后将这个计数传给fisher从而避免在fisher本身再次对它们计数。当单独的概率中含有许多来自随机文本的低概率值时，由fisher返回的值也将非常低。这样，一个低的fisher垃圾邮件概率评分意味着存在许多有用的特征。将这个评分减去1就得到该邮件是有用邮件的概率。相反，从有用概率中减去fisher评分将得到该邮件是垃圾邮件的概率。将这两个概率组合在一起就可以给你一个介于0和1之间的整体垃圾邮件程度评分。

在循环内部，你可以使用函数untrained-p来跳过那些从邮件中提取出的从未在训练中出现过的特征。这些特征将具有值为0的垃圾邮件计数和有用计数。untrained-p函数非常简单。

```
(defun untrained-p (feature)
  (with-slots (spam-count ham-count) feature
    (and (zerop spam-count) (zerop ham-count))))
```

剩下的唯一一个新的函数是fisher本身。假设你已经有了一个inverse-chi-square函数，那么fisher在概念上很简单。

```
(defun fisher (probs number-of-probs)
  "The Fisher computation described by Robinson."
```

```
(inverse-chi-square
 (* -2 (log (reduce #'* probs)))
 (* 2 number-of-probs)))
```

不幸的是，在这个相当直接的实现中有一个小问题。尽管使用REDUCE是一个将数字列表相乘的简洁方法，但在这个特定应用中乘积将会过小而无法表示成一个浮点数。在这种情况下，结果将会下溢到0。而且，因为概率的乘积下溢，所有努力都将白费，因为对0求LOG要么报错，要么在某些实现中得到一个特殊的负无穷大值，这将使得所有后续的计算在本质上都变成无意义的。这在本函数中尤其不幸，因为当输入的概率值较低（接近0）时，fisher方法最为敏感，并且因此非常容易导致乘积下溢。

幸运的是，你可以运用一点儿高中数学知识来避免这个问题。一个乘法的对数等价于所有因数的对数之和，因此不用将所有概率相乘然后取对数，你可以将每个概率的对数相加。并且，由于REDUCE接受一个:key关键字参数，你可以用它来完成整个计算。把下面的写法：

```
(log (reduce #'* probs))
```

写成这样：

```
(reduce #'+ probs :key #'log)
```

23.5 反向卡方分布函数

本节中的inverse-chi-square实现是Rebinson所写的一个Python版本的相当直接的转换。讲解该函数的确切数学含义超出了本书的范围，但通过思考你传递给fisher的值将怎样影响结果可以得到一个关于其作用的直观印象：你传给fisher的低概率值越多，概率的乘积将会越小。一个小的乘积的对数将会是一个绝对值较大的负数。这样，传递给fisher的低概率值越多，它传递给inverse-chi-square的值就越大。当然，引入的概率数量也会影响传递给inverse-chi-square的值。由于概率的定义是小于或等于1的，一个乘积中的概率越多，它的结果就会越小并且传给inverse-chi-square的值就会越大。这样，在fisher合并值相比进入它的概率数量异乎寻常地大时，inverse-chi-square将返回一个较低的概率。下面的函数精确地做到了这点：

```
(defun inverse-chi-square (value degrees-of-freedom)
  (assert (evenp degrees-of-freedom))
  (min
   (loop with m = (/ value 2)
      for i below (/ degrees-of-freedom 2)
      for prob = (exp (- m)) then (* prob (/ m i))
      summing prob)
   1.0))
```

回忆第10章里**EXP**计算e的给定参数次方。这样，value的值越大，prob的初始值将会越小。但只要m大于自由度的数量，这个初始值随后就将不断地被每个自由度微调。由于inverse-chi-square返回的值应当是另一个概率，有必要用**MIN**来固定返回值，因为乘法和指数计算中的边界错误可能导致**LOOP**返回一个稍大于1的和。

23.6 训练过滤器

你编写classify和train来接受一个字符串参数，因此你可以轻松地在REPL中测试它们。如果你还没有这样做过，那么你应当通过在REPL中求值一个**IN-PACKAGE**形式，或是使用SLIME的快捷命令change-package将当前包切换到你编写这些代码所在的包中。在你输入包名的时候，按Tab将会根据你的Lisp所知道的包来自动补全包名。现在你可以调用任何属于垃圾邮件应用一部分的函数了。你应当首先确保数据库为空。

```
SPAM> (clear-database)
```

现在你可以用一些文本来训练过滤器。

```
SPAM> (train "Make money fast" 'spam)
```

然后看分类器是怎样判断的。

```
SPAM> (classify "Make money fast")
SPAM
SPAM> (classify "Want to go to the movies?")
UNSURE
```

尽管最终你所关心的只是那个分类，但可以看到原始的评分也是很有用的。得到两个值而不会干扰任何其他代码的最简单方法是改变classification从而返回多个值。

```
(defun classification (score)
  (values
   (cond
     ((<= score *max-ham-score*) 'ham)
     ((>= score *min-spam-score*) 'spam)
     (t 'unsure))
   score))
```

你可以做出这个改变，然后只重新编译这一个函数。classify返回classification所返回的任何东西，因此它也将返回两个值。但由于主返回值和以前相同，这两个函数的那些只需要一个值的调用者将不会受到影响。现在当你测试classify时，能够精确地看到进入到分类中的评分。

```
SPAM> (classify "Make money fast")
SPAM
0.863677101854273D0
SPAM> (classify "Want to go to the movies?")
UNSURE
0.5D0
```

现在你可以看到，如果你用更多的一些有用邮件来训练过滤器的话将发生什么。

```
SPAM> (train "Do you have any money for the movies?" 'ham)
1
SPAM> (classify "Make money fast")
SPAM
0.7685351219857626D0
```

它仍然是垃圾邮件，只是不太确定，因为“money”在有用邮件中也出现了。

```
SPAM> (classify "Want to go to the movies?")
HAM
0.17482223132078922D0
```

而现在它被清楚地识别成有用的邮件，这要感谢单词“movies”的存在，这是一个有用的特征。

不过，你可能并不真的想手工训练这个过滤器，你真正喜欢的是一种简单的方式来指向一堆文件并在其上训练它。除此之外，因为你想要测试该过滤器实际工作的效果，你会希望随后使用它来分类另外一些已知类型的文件并查看分类的效果。因此，在本章中你将要编写的最后一点代码是一套测试系统，它在一个已知类型的邮件库上测试该过滤器，使用其中的一定比例用于训练，然后在其余部分测量该过滤器在分类时的精度。

23.7 测试过滤器

为了测试该过滤器，你需要一个已知类型的邮件库。你可以使用邮箱中的邮件，或者从Web上获得一个可用的邮件库。例如，SpamAssassin邮件库[①]含有手工分类成垃圾邮件、稍微有用和十分有用的几千条邮件。为了更容易地使用你所拥有的文件，可以定义一个驱动在一个文件/类型对的数组上的测试平台。你可以定义一个函数来接受文件名和类型，并像下面这样将其添加到消息库中：


```
(defun add-file-to-corpus (filename type corpus)
  (vector-push-extend (list filename type) corpus))
```

其中corpus的值应当是一个带有填充指针的可调整向量。例如，你可以像这样创建一个新的库：

```
(defparameter *corpus* (make-array 1000 :adjustable t :fill-pointer 0))
```

如果你的有用邮件和垃圾邮件已经分别放在了不同的目录中，那么你可能想要一次性将一个目录中的所有文件作为相同的类型添加到库中。下面这个函数使用了第15章的list-directory函数，它实现了上述想法：

```
(defun add-directory-to-corpus (dir type corpus)
  (dolist (filename (list-directory dir))
    (add-file-to-corpus filename type corpus)))
```

例如，假设你有一个mail目录，它含有两个子目录spam和ham，分别包含相应类型的消息。你可以像下面这样把这两个目录中的所有文件添加到*corpus*中：

```
SPAM> (add-directory-to-corpus "mail/spam/" 'spam *corpus*)
NIL
SPAM> (add-directory-to-corpus "mail/ham/" 'ham *corpus*)
NIL
```

① 包括SpamAssassin邮件库在内的几个垃圾邮件库可以从http://nexp.cs.pdx.edu/~psam/cgi-bin/view/PSAM/CorpusSets获得。

现在你需要一个函数来测试分类器。基本的策略是选择库中的一个随机片段来训练，然后通过把库的其余部分分类来测试这个库，将由classify返回的分类与已知的分类进行比较。你主要想知道的是分类器的精度——究竟有多少百分比的消息被正确分类了？但你还可能对错误分类的消息以及错误的方向感兴趣——究竟是假阳性更多还是假阴性更多？为了方便对分类器的行为不断地进行分析，你应当定义测试函数来构造一个原始结果的列表，随后你可以用任何方法来分析它。

主测试函数如下所示：

```
(defun test-classifier (corpus testing-fraction)
  (clear-database)
  (let* ((shuffled (shuffle-vector corpus))
         (size (length corpus))
         (train-on (floor (* size (- 1 testing-fraction)))))
    (train-from-corpus shuffled :start 0 :end train-on)
    (test-from-corpus shuffled :start train-on)))
```

这个函数从清空特征数据库开始。[①]然后，它对整个库进行“洗牌”，使用一个你将很快实现的函数，基于其testing-fraction参数来找出用于训练的消息和将被保留用来测试的消息。两个辅助函数train-from-corpus和test-from-corpus都将带有关键字参数:start和:end，从而允许它们对给定消息库的一个子序列进行操作。

train-from-corpus函数相当简单——简单地在库的适当部分中循环，它使用**DESTRCUTURING-BIND**从每个元素中解出文件名和类型，然后将命名文件的文本和类型传给train。由于某些邮件消息，尤其是那些带有附件的，通常会比较大，你应当限制它从消息中获取的字符数量。它使用一个你将很快实现的函数start-of-file来获取文本，该函数接受一个文件名和一个最大的字符数来返回相应的文本。train-from-corpus如下所示：

```
(defparameter *max-chars* (* 10 1024))

(defun train-from-corpus (corpus &key (start 0) end)
  (loop for idx from start below (or end (length corpus)) do
        (destructuring-bind (file type) (aref corpus idx)
          (train (start-of-file file *max-chars*) type))))
```

除了要返回一个含有每个分类结果的列表，从而可以稍后来分析它们之外，函数test-from-corpus和上述函数相似。这样，你应当同时捕捉由classify返回的分类和评分数据，然后将文件名、实际类型、由classify返回的类型以及评分收集在一个列表中。为了使结果更好理解，你可以在列表中放置一些关键字来指示每个值的含义。

```
(defun test-from-corpus (corpus &key (start 0) end)
  (loop for idx from start below (or end (length corpus)) collect
        (destructuring-bind (file type) (aref corpus idx)
          (multiple-value-bind (classification score)
```

① 如果你想要进行一个测试而不想干扰已有的数据库，那么你可以用一个**LET**绑定*feature-database*、*total-spams*和*total-hams*，但这样的话在测试结束之后，你就没有办法查看数据库了，除非你把所用到的这些值返回到函数中。

```
      (classify (start-of-file file *max-chars*))
    (list
     :file file
     :type type
     :classification classification
     :score score)))))
```

23.8 一组工具函数

为了完成test-classifier的实现，你还需要编写两个事实上跟垃圾邮件过滤没有特别关系的工具函数，shuffle-vector和start-of-file。

实现shuffle-vector的简单有效的方法是使用Fisher-Yates算法[①]。你可以从实现一个函数nshuffle-vector开始，它可以就地重排一个向量。这个名字遵循与诸如**NCONC**和**NREVERSE**等其他破坏性函数相同的命名规则。如下所示：

```
(defun nshuffle-vector (vector)
  (loop for idx downfrom (1- (length vector)) to 1
        for other = (random (1+ idx))
        do (unless (= idx other)
             (rotatef (aref vector idx) (aref vector other))))
  vector)
```

非破坏性版本简单地复制最初的向量，然后将它传给破坏性版本。

```
(defun shuffle-vector (vector)
  (nshuffle-vector (copy-seq vector)))
```

另一个工具函数start-of-file也是非常直接的，只有一点特别。把一个文件的内容读取到内存中最有效的方式是创建一个适当大小的数组并使用**READ-SEQUENCE**来填充其内容。因此，你应该创建一个字符数组，其长度要么是文件的大小要么是你想要读取的字符的最大数量，后者相对小一些。不幸的是，如同第14章里提到的，在处理字符流时，函数**FILE-LENGTH**完全没有很好地定义，因为一个文件中编码的字符个数可能同时取决于使用的字符编码和文件中的特定文本。在最坏的情况下，精确测量文件中字符数的唯一方法是实际读取整个文件。这样，在处理字符流时**FILE-LENGTH**存在歧义。而在多数实现中，**FILE-LENGTH**总是返回文件的字节数，这可能大于可从文件中读取的字符数。

不过，**READ-SEQUENCE**可以返回实际读取的字符数。因此，你可以尝试读取由**FILE-LENGTH**报告的字符数，并在实际读取的字符数较少时返回一个子串。

```
(defun start-of-file (file max-chars)
  (with-open-file (in file)
    (let* ((length (min (file-length in) max-chars))
           (text (make-string length))
```

① 这个算法以发明了概率合并方法的同一个Fisher和Frank Yates来命名，后者是Fisher的*Statistical Tables for Biological, Agricultural and Medical Research*（Oliver & Boyd, 1938）一书的共同作者。根据Knuth的说法，他们首次公开发表了对该算法的描述。

```
        (read (read-sequence text in)))
    (if (< read length)
      (subseq text 0 read)
      text))))
```

23.9　分析结果

现在编写一些代码来分析由test-classifier生成的结果。回顾一下，test-classifier返回了由test-from-corpus所返回的列表，其中每个元素是一个代表了文件分类结果的plist。这个plist含有该文件的名字、文件的实际类型、分类以及由classify所返回的评分。编写分析性代码的第一步是编写一个函数，该函数可以返回一个符号来指明一个给定结果究竟是正确的、假阳性的、假阴性的、错过的有用消息或错过的垃圾邮件消息。你可以使用**DESTRUCTURING-BIND**从一个单独的结果列表中取出:type和:classification元素（使用&allow-other-keys来告诉**DESTRUCTURING-BIND**忽略任何其他的键值对），然后使用嵌套的**ECASE**将不同的配对转换成单一符号。

```
(defun result-type (result)
  (destructuring-bind (&key type classification &allow-other-keys) result
    (ecase type
      (ham
       (ecase classification
         (ham 'correct)
         (spam 'false-positive)
         (unsure 'missed-ham)))
      (spam
       (ecase classification
         (ham 'false-negative)
         (spam 'correct)
         (unsure 'missed-spam))))))
```

你可以在REPL中测试这个函数。

```
SPAM> (result-type '(:FILE #p"foo" :type ham :classification ham :score 0))
CORRECT
SPAM> (result-type '(:FILE #p"foo" :type spam :classification spam :score 0))
CORRECT
SPAM> (result-type '(:FILE #p"foo" :type ham :classification spam :score 0))
FALSE-POSITIVE
SPAM> (result-type '(:FILE #p"foo" :type spam :classification ham :score 0))
FALSE-NEGATIVE
SPAM> (result-type '(:FILE #p"foo" :type ham :classification unsure :score 0))
MISSED-HAM
SPAM> (result-type '(:FILE #p"foo" :type spam :classification unsure :score 0))
MISSED-SPAM
```

有了这个函数，你就可以方便地以多种方式切分test-classifier的结果了。例如，你可以从为每种结果类型定义谓词函数开始。

```
(defun false-positive-p (result)
  (eql (result-type result) 'false-positive))
```

```
(defun false-negative-p (result)
  (eql (result-type result) 'false-negative))

(defun missed-ham-p (result)
  (eql (result-type result) 'missed-ham))

(defun missed-spam-p (result)
  (eql (result-type result) 'missed-spam))

(defun correct-p (result)
  (eql (result-type result) 'correct))
```

有了这些函数，你可以轻易地使用我在第11章里讨论的列表和序列操作函数来解出并统计特定类型的结果。

```
SPAM> (count-if #'false-positive-p *results*)
6
SPAM> (remove-if-not #'false-positive-p *results*)
((:FILE #p"ham/5349" :TYPE HAM :CLASSIFICATION SPAM :SCORE 0.9999983107355541d0)
 (:FILE #p"ham/2746" :TYPE HAM :CLASSIFICATION SPAM :SCORE 0.6286468956619795d0)
 (:FILE #p"ham/3427" :TYPE HAM :CLASSIFICATION SPAM :SCORE 0.9833753501352983d0)
 (:FILE #p"ham/7785" :TYPE HAM :CLASSIFICATION SPAM :SCORE 0.9542788587998488d0)
 (:FILE #p"ham/1728" :TYPE HAM :CLASSIFICATION SPAM :SCORE 0.684339162891261d0)
 (:FILE #p"ham/10581" :TYPE HAM :CLASSIFICATION SPAM :SCORE 0.9999924537959615d0))
```


你还可以使用result-type返回的符号作为哈希表或alist中的键。例如，你可以编写一个函数来打印结果中每种类型的个数和比例，该函数使用alist将每种类型和一个额外的符号total映射到一个计数上。

```
(defun analyze-results (results)
  (let* ((keys '(total correct false-positive
                 false-negative missed-ham missed-spam))
         (counts (loop for x in keys collect (cons x 0))))
    (dolist (item results)
      (incf (cdr (assoc 'total counts)))
      (incf (cdr (assoc (result-type item) counts))))
    (loop with total = (cdr (assoc 'total counts))
          for (label . count) in counts
          do (format t "~&~@(~a~):~20t~5d~,5t: ~6,2f%~%"
                     label count (* 100 (/ count total))))))
```

当给该函数传递一个由test-classifier生成的结果列表时，它将给出下面的输出：

```
SPAM> (analyze-results *results*)
Total:               3761 : 100.00%
Correct:             3689 :  98.09%
False-positive:         4 :   0.11%
False-negative:         9 :   0.24%
Missed-ham:            19 :   0.51%
Missed-spam:           40 :   1.06%
NIL
```

而在分析的最后，你可能想要了解一个单独的消息被分类成某种类型的原因。下面的函数将

为你显示这点：

```
(defun explain-classification (file)
  (let* ((text (start-of-file file *max-chars*))
         (features (extract-features text))
         (score (score features))
         (classification (classification score)))
    (show-summary file text classification score)
    (dolist (feature (sorted-interesting features))
      (show-feature feature))))

(defun show-summary (file text classification score)
  (format t "~&~a" file)
  (format t "~2%~a~2%" text)
  (format t "Classified as ~a with score of ~,5f~%" classification score))

(defun show-feature (feature)
  (with-slots (word ham-count spam-count) feature
    (format
     t "~&~2t~a~30thams: ~5d; spams: ~5d;~,10tprob: ~,f~%"
     word ham-count spam-count (bayesian-spam-probability feature))))

(defun sorted-interesting (features)
  (sort (remove-if #'untrained-p features) #'< :key #'bayesian-spam-probability))
```

23.10 接下来的工作

很明显，你可以用这些代码做更多的事。为了将它变成一个真正的垃圾邮件过滤应用程序，你需要找到一种方式来将它集成到你正常的电子邮件基础服务框架中。一种使它方便地与几乎任何电子邮件客户端相集成的思路是，编写一点儿代码来使它成为一个POP3代理。这是多数电子邮件客户端从邮件服务器上获取邮件所使用的协议，这样一个代理将从你的实际POP3服务器中获取邮件并为你的电子邮件客户端提供服务，在这个过程中它要么将垃圾邮件标记一个你的电子邮件客户端过滤器可以理解的信头，要么直接把它放在一边。当然你可能还需要一种方式来与过滤器沟通有关错误分类的信息。只要你把它设置成一个服务器，你就可以提供一个Web接口。第26章将谈及如何编写Web接口，第29章将为不同的应用构建Web接口。

或者你可能想要改进基本分类——一个可能的起点是令extract-features更加专业。特别地，你可以使分词器更聪明地处理电子邮件的内部结构，即可以为出现在消息体和消息头中的单词解出不同类型的特征。你还可以解码包括Base64和Quoted Printable在内的多种类型的消息编码，因为垃圾邮件发送者经常使用这些编码来扰乱它们的消息。

但是我将把这些改进留给你。现在你已准备好继续前进来构建一个流式MP3服务器，先从编写一个解析二进制文件的通用库开始。