23. Practical: A Spam Filter

In 2002 Paul Graham, having some time on his hands after selling Viaweb to Yahoo, wrote the essay "A Plan for Spam"[1] that launched a minor revolution in spam-filtering technology. Prior to Graham's article, most spam filters were written in terms of handcrafted rules: if a message has *XXX* in the subject, it's probably a spam; if a message has a more than three or more words in a row in ALL CAPITAL LETTERS, it's probably a spam. Graham spent several months trying to write such a rule-based filter before realizing it was fundamentally a soul-sucking task.

> To recognize individual spam features you have to try to get into the mind of the spammer, and frankly I want to spend as little time inside the minds of spammers as possible.

To avoid having to think like a spammer, Graham decided to try distinguishing spam from nonspam, a.k.a. *ham*, based on statistics gathered about which words occur in which kinds of e-mails. The filter would keep track of how often specific words appear in both spam and ham messages and then use the frequencies associated with the words in a new message to compute a probability that it was either spam or ham. He called his approach *Bayesian* filtering after the statistical technique that he used to combine the individual word frequencies into an overall probability.[2]

The Heart of a Spam Filter

In this chapter, you'll implement the core of a spam-filtering engine. You won't write a soup-to-nuts spam-filtering application; rather, you'll focus on the functions for classifying new messages and training the filter.

This application is going to be large enough that it's worth defining a new package to avoid name conflicts. For instance, in the source code you can download from this book's Web site, I use the package name `COM.GIGAMONKEYS.SPAM`, defining a package that uses both the standard `COMMON-LISP` package and the `COM.GIGAMONKEYS.PATHNAMES` package from Chapter 15, like this:

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

Any file containing code for this application should start with this line:

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

You can use the same package name or replace `com.gigamonkeys` with some domain you control.[3]

You can also type this same form at the REPL to switch to this package to test the functions you write. In SLIME this will change the prompt from `CL-USER>` to `SPAM>` like this:

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

Once you have a package defined, you can start on the actual code. The main function you'll need to implement has a simple job--take the text of a message as an argument and classify the message as spam, ham, or unsure. You can easily implement this basic function by defining it in terms of other functions that you'll write in a moment.

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

Reading from the inside out, the first step in classifying a message is to extract features to pass to the `score` function. In `score` you'll compute a value that can then be translated into one of three classifications--spam, ham, or unsure--by the function `classification`. Of the three functions, `classification` is the simplest. You can assume `score` will return a value near 1 if the message is a spam, near 0 if it's a ham, and near .5 if it's unclear.

Thus, you can implement `classification` like this:

```
(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)))
```

The `extract-features` function is almost as straightforward, though it requires a bit more code. For the moment, the features you'll extract will be the words appearing in the text. For each word, you need to keep track of the number of times it has been seen in a spam and the number of times it has been seen in a ham. A convenient way to keep those pieces of data together with the word itself is to define a class, `word-feature`, with three slots.

```
(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
    :accessor ham-count
```

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

You'll keep the database of features in a hash table so you can easily find the object representing a given feature. You can define a special variable, `*feature-database*`, to hold a reference to this hash table.

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

You should use **`DEFVAR`** rather than **`DEFPARAMETER`** because you don't want `*feature-database*` to be reset if you happen to reload the file containing this definition during development--you might have data stored in `*feature-database*` that you don't want to lose. Of course, that means if you *do* want to clear out the feature database, you can't just reevaluate the **`DEFVAR`** form. So you should define a function `clear-database`.

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

To find the features present in a given message, the code will need to extract the individual words and then look up the corresponding `word-feature` object in `*feature-database*`. If `*feature-database*` contains no such feature, it'll need to create a new `word-feature` to represent the word. You can encapsulate that bit of logic in a function, `intern-feature`, that takes a word and returns the appropriate feature, creating it if necessary.

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

You can extract the individual words from the message text using a regular expression. For example, using the Common Lisp Portable Perl-Compatible Regular Expression (CL-PPCRE) library written by Edi Weitz, you can write `extract-words` like this:[4]

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

Now all that remains to implement `extract-features` is to put `extract-features` and `intern-feature` together. Since `extract-words` returns a list of strings and you want a list with each string translated to the corresponding `word-feature`, this is a perfect time to use **`MAPCAR`**.

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

You can test these functions at the REPL like this:

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

And you can make sure the **DELETE-DUPLICATES** is working like this:

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

You can also test `extract-features`.

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

However, as you can see, the default method for printing arbitrary objects isn't very informative. As you work on this program, it'll be useful to be able to print `word-feature` objects in a less opaque way. Luckily, as I mentioned in Chapter 17, the printing of all objects is implemented in terms of a generic function **PRINT-OBJECT**, so to change the way `word-feature` objects are printed, you just need to define a method on **PRINT-OBJECT** that specializes on `word-feature`. To make implementing such methods easier, Common Lisp provides the macro **PRINT-UNREADABLE-OBJECT**.[5]

The basic form of **PRINT-UNREADABLE-OBJECT** is as follows:

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

The *object* argument is an expression that evaluates to the object to be printed. Within the body of **PRINT-UNREADABLE-OBJECT**, *stream-variable* is bound to a stream to which you can print anything you want. Whatever you print to that stream will be output by **PRINT-UNREADABLE-OBJECT** and enclosed in the standard syntax for unreadable objects, #<>.[6]

PRINT-UNREADABLE-OBJECT also lets you include the type of the object and an indication of the object's identity via the keyword parameters *type* and *identity*. If they're non-**NIL**, the output will start with the name of the object's class and end with an indication of the object's identity similar to what's printed by the default **PRINT-OBJECT** method for **STANDARD-OBJECT**s. For `word-feature`, you probably want to define a **PRINT-OBJECT** method that includes the type but not the identity along with the values of the `word`, `ham-count`, and `spam-count` slots. Such a method would look like this:

```
(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))))
```

Now when you test `extract-features` at the REPL, you can see more clearly what features are being extracted.

```
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>)
```

Training the Filter

Now that you have a way to keep track of individual features, you're almost ready to implement `score`. But first you need to write the code you'll use to train the spam filter so `score` will have some data to use. You'll define a function, `train`, that takes some text and a symbol indicating what kind of message it is--`ham` or `spam`--and that increments either the ham count or the spam count of all the features present in the text as well as a global count of hams or spams processed. Again, you can take a top-down approach and implement it in terms of other functions that don't yet exist.

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

You've already written `extract-features`, so next up is `increment-count`, which takes a `word-feature` and a message type and increments the appropriate slot of the feature. Since there's no reason to think that the logic of incrementing these counts is going to change for different kinds of objects, you can write this as a regular function.[7] Because you defined both `ham-count` and `spam-count` with an `:accessor` option, you can use **`INCF`** and the accessor functions created by **`DEFCLASS`** to increment the appropriate slot.

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

The **`ECASE`** construct is a variant of **`CASE`**, both of which are similar to `case` statements in Algol-derived languages (renamed `switch` in C and its progeny). They both evaluate their first argument--the *key form*--and then find the clause whose first element--the *key*--is the same value according to **`EQL`**. In this case, that means the variable `type` is evaluated, yielding whatever value was passed as the second argument to `increment-count`.

The keys aren't evaluated. In other words, the value of `type` will be compared to the literal objects read by the Lisp reader as part of the **`ECASE`** form. In this function, that means the keys are the symbols `ham` and `spam`, not the values of any variables named `ham` and `spam`. So, if `increment-count` is called like this:

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

the value of `type` will be the symbol `ham`, and the first branch of the **`ECASE`** will be evaluated and the feature's ham count incremented. On the other hand, if it's called like this:

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

then the second branch will run, incrementing the spam count. Note that the symbols `ham` and `spam` are quoted when calling `increment-count` since otherwise they'd be evaluated as the

names of variables. But they're not quoted when they appear in **ECASE** since **ECASE** doesn't evaluate the keys.[8]

The *E* in **ECASE** stands for "exhaustive" or "error," meaning **ECASE** should signal an error if the key value is anything other than one of the keys listed. The regular **CASE** is looser, returning **NIL** if no matching clause is found.

To implement `increment-total-count`, you need to decide where to store the counts; for the moment, two more special variables, `*total-spams*` and `*total-hams*`, will do fine.

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

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

You should use **DEFVAR** to define these two variables for the same reason you used it with `*feature-database*`--they'll hold data built up while you run the program that you don't necessarily want to throw away just because you happen to reload your code during development. But you'll want to reset those variables if you ever reset `*feature-database*`, so you should add a few lines to `clear-database` as shown here:

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

Per-Word Statistics

The heart of a statistical spam filter is, of course, the functions that compute statistics-based probabilities. The mathematical nuances[9] of why exactly these computations work are beyond the scope of this book--interested readers may want to refer to several papers by Gary Robinson.[10] I'll focus rather on how they're implemented.

The starting point for the statistical computations is the set of measured values--the frequencies stored in `*feature-database*`, `*total-spams*`, and `*total-hams*`. Assuming that the set of messages trained on is statistically representative, you can treat the observed frequencies as probabilities of the same features showing up in hams and spams in future messages.

The basic plan is to classify a message by extracting the features it contains, computing the individual probability that a given message containing the feature is a spam, and then combining all the individual probabilities into a total score for the message. Messages with many "spammy"

features and few "hammy" features will receive a score near 1, and messages with many hammy features and few spammy features will score near 0.

The first statistical function you need is one that computes the basic probability that a message containing a given feature is a spam. By one point of view, the probability that a given message containing the feature is a spam is the ratio of spam messages containing the feature to all messages containing the feature. Thus, you could compute it this way:

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

The problem with the value computed by this function is that it's strongly affected by the overall probability that *any* message will be a spam or a ham. For instance, suppose you get nine times as much ham as spam in general. A completely neutral feature will then appear in one spam for every nine hams, giving you a spam probability of 1/10 according to this function.

But you're more interested in the probability that a given feature will appear in a spam message, independent of the overall probability of getting a spam or ham. Thus, you need to divide the spam count by the total number of spams trained on and the ham count by the total number of hams. To avoid division-by-zero errors, if either of `*total-spams*` or `*total-hams*` is zero, you should treat the corresponding frequency as zero. (Obviously, if the total number of either spams or hams is zero, then the corresponding per-feature count will also be zero, so you can treat the resulting frequency as zero without ill effect.)

```
(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)))))
```

This version suffers from another problem--it doesn't take into account the number of messages analyzed to arrive at the per-word probabilities. Suppose you've trained on 2,000 messages, half spam and half ham. Now consider two features that have appeared only in spams. One has appeared in all 1,000 spams, while the other appeared only once. According to the current definition of `spam-probability`, the appearance of either feature predicts that a message is spam with equal probability, namely, 1.

However, it's still quite possible that the feature that has appeared only once is actually a neutral feature--it's obviously rare in either spams or hams, appearing only once in 2,000 messages. If you trained on another 2,000 messages, it might very well appear one more time, this time in a ham, making it suddenly a neutral feature with a spam probability of .5.

So it seems you might like to compute a probability that somehow factors in the number of data points that go into each feature's probability. In his papers, Robinson suggested a function based on the Bayesian notion of incorporating observed data into prior knowledge or assumptions. Basically, you calculate a new probability by starting with an assumed prior probability and a

weight to give that assumed probability before adding new information. Robinson's function is this:

```
(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 suggests values of 1/2 for `assumed-probability` and 1 for `weight`. Using those values, a feature that has appeared in one spam and no hams has a `bayesian-spam-probability` of 0.75, a feature that has appeared in 10 spams and no hams has a `bayesian-spam-probability` of approximately 0.955, and one that has matched in 1,000 spams and no hams has a spam probability of approximately 0.9995.

Combining Probabilities

Now that you can compute the `bayesian-spam-probability` of each individual feature you find in a message, the last step in implementing the `score` function is to find a way to combine a bunch of individual probabilities into a single value between 0 and 1.

If the individual feature probabilities were independent, then it'd be mathematically sound to multiply them together to get a combined probability. But it's unlikely they actually are independent--certain features are likely to appear together, while others never do.[11]

Robinson proposed using a method for combining probabilities invented by the statistician R. A. Fisher. Without going into the details of exactly why his technique works, it's this: First you combine the probabilities by multiplying them together. This gives you a number nearer to 0 the more low probabilities there were in the original set. Then take the log of that number and multiply by -2. Fisher showed in 1950 that if the individual probabilities were independent and drawn from a uniform distribution between 0 and 1, then the resulting value would be on a chi-square distribution. This value and twice the number of probabilities can be fed into an inverse chi-square function, and it'll return the probability that reflects the likelihood of obtaining a value that large or larger by combining the same number of randomly selected probabilities. When the inverse chi-square function returns a low probability, it means there was a disproportionate number of low probabilities (either a lot of relatively low probabilities or a few very low probabilities) in the individual probabilities.

To use this probability in determining whether a given message is a spam, you start with a *null hypothesis*, a straw man you hope to knock down. The null hypothesis is that the message being classified is in fact just a random collection of features. If it were, then the individual probabilities--the likelihood that each feature would appear in a spam--would also be random. That is, a random selection of features would usually contain some features with a high

probability of appearing in spam and other features with a low probability of appearing in spam. If you were to combine these randomly selected probabilities according to Fisher's method, you should get a middling combined value, which the inverse chi-square function will tell you is quite likely to arise just by chance, as, in fact, it would have. But if the inverse chi-square function returns a very low probability, it means it's unlikely the probabilities that went into the combined value were selected at random; there were too many low probabilities for that to be likely. So you can reject the null hypothesis and instead adopt the alternative hypothesis that the features involved were drawn from a biased sample--one with few high spam probability features and many low spam probability features. In other words, it must be a ham message.

However, the Fisher method isn't symmetrical since the inverse chi-square function returns the probability that a given number of randomly selected probabilities would combine to a value as large or larger than the one you got by combining the actual probabilities. This asymmetry works to your advantage because when you reject the null hypothesis, you know what the more likely hypothesis is. When you combine the individual spam probabilities via the Fisher method, and it tells you there's a high probability that the null hypothesis is wrong--that the message isn't a random collection of words--then it means it's likely the message is a ham. The number returned is, if not literally the probability that the message is a ham, at least a good measure of its "hamminess." Conversely, the Fisher combination of the individual ham probabilities gives you a measure of the message's "spamminess."

To get a final score, you need to combine those two measures into a single number that gives you a combined hamminess-spamminess score ranging from 0 to 1. The method recommended by Robinson is to add half the difference between the hamminess and spamminess scores to 1/2, in other words, to average the spamminess and 1 minus the hamminess. This has the nice effect that when the two scores agree (high spamminess and low hamminess, or vice versa) you'll end up with a strong indicator near either 0 or 1. But when the spamminess and hamminess scores are both high or both low, then you'll end up with a final value near 1/2, which you can treat as an "uncertain" classification.

The `score` function that implements this scheme looks like this:

```
(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))))
```

You take a list of features and loop over them, building up two lists of probabilities, one listing the probabilities that a message containing each feature is a spam and the other that a message containing each feature is a ham. As an optimization, you can also count the number of

probabilities while looping over them and pass the count to `fisher` to avoid having to count them again in `fisher` itself. The value returned by `fisher` will be low if the individual probabilities contained too many low probabilities to have come from random text. Thus, a low `fisher` score for the spam probabilities means there were many hammy features; subtracting that score from 1 gives you a probability that the message is a ham. Conversely, subtracting the `fisher` score for the ham probabilities gives you the probability that the message was a spam. Combining those two probabilities gives you an overall spamminess score between 0 and 1.

Within the loop, you can use the function `untrained-p` to skip features extracted from the message that were never seen during training. These features will have spam counts and ham counts of zero. The `untrained-p` function is trivial.

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

The only other new function is `fisher` itself. Assuming you already had an `inverse-chi-square` function, `fisher` is conceptually simple.

```
(defun fisher (probs number-of-probs)
  "The Fisher computation described by Robinson."
  (inverse-chi-square
   (* -2 (log (reduce #'* probs)))
   (* 2 number-of-probs)))
```

Unfortunately, there's a small problem with this straightforward implementation. While using **`REDUCE`** is a concise and idiomatic way of multiplying a list of numbers, in this particular application there's a danger the product will be too small a number to be represented as a floating-point number. In that case, the result will *underflow* to zero. And if the product of the probabilities underflows, all bets are off because taking the **`LOG`** of zero will either signal an error or, in some implementation, result in a special negative-infinity value, which will render all subsequent calculations essentially meaningless. This is particularly unfortunate in this function because the Fisher method is most sensitive when the input probabilities are low--near zero--and therefore in the most danger of causing the multiplication to underflow.

Luckily, you can use a bit of high-school math to avoid this problem. Recall that the log of a product is the same as the sum of the logs of the factors. So instead of multiplying all the probabilities and then taking the log, you can sum the logs of each probability. And since **`REDUCE`** takes a `:key` keyword parameter, you can use it to perform the whole calculation. Instead of this:

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

write this:

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

Inverse Chi Square

The implementation of `inverse-chi-square` in this section is a fairly straightforward translation of a version written in Python by Robinson. The exact mathematical meaning of this function is beyond the scope of this book, but you can get an intuitive sense of what it does by thinking about how the values you pass to `fisher` will affect the result: the more low probabilities you pass to `fisher`, the smaller the product of the probabilities will be. The log of a small product will be a negative number with a large absolute value, which is then multiplied by -2, making it an even larger positive number. Thus, the more low probabilities were passed to `fisher`, the larger the value it'll pass to `inverse-chi-square`. Of course, the number of probabilities involved also affects the value passed to `inverse-chi-square`. Since probabilities are, by definition, less than or equal to 1, the more probabilities that go into a product, the smaller it'll be and the larger the value passed to `inverse-chi-square`. Thus, `inverse-chi-square` should return a low probability when the Fisher combined value is abnormally large for the number of probabilities that went into it. The following function does exactly that:

```
(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))
```

Recall from Chapter 10 that **`EXP`** raises *e* to the argument given. Thus, the larger `value` is, the smaller the initial value of `prob` will be. But that initial value will then be adjusted upward slightly for each degree of freedom as long as `m` is greater than the number of degrees of freedom. Since the value returned by `inverse-chi-square` is supposed to be another probability, it's important to clamp the value returned with **`MIN`** since rounding errors in the multiplication and exponentiation may cause the **`LOOP`** to return a sum just a shade over 1.

Training the Filter

Since you wrote `classify` and `train` to take a string argument, you can test them easily at the REPL. If you haven't yet, you should switch to the package in which you've been writing this code by evaluating an **`IN-PACKAGE`** form at the REPL or using the SLIME shortcut `change-package`. To use the SLIME shortcut, type a comma at the REPL and then type the name at the prompt. Pressing Tab while typing the package name will autocomplete based on the packages your Lisp knows about. Now you can invoke any of the functions that are part of the spam application. You should first make sure the database is empty.

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

Now you can train the filter with some text.

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

And then see what the classifier thinks.

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

While ultimately all you care about is the classification, it'd be nice to be able to see the raw score too. The easiest way to get both values without disturbing any other code is to change `classification` to return multiple values.

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

You can make this change and then recompile just this one function. Because `classify` returns whatever `classification` returns, it'll also now return two values. But since the primary return value is the same, callers of either function who expect only one value won't be affected. Now when you test `classify`, you can see exactly what score went into the classification.

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

And now you can see what happens if you train the filter with some more ham text.

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

It's still spam but a bit less certain since *money* was seen in ham text.

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

And now this is clearly recognizable ham thanks to the presence of the word *movies*, now a hammy feature.

However, you don't really want to train the filter by hand. What you'd really like is an easy way to point it at a bunch of files and train it on them. And if you want to test how well the filter actually works, you'd like to then use it to classify another set of files of known types and see how it does. So the last bit of code you'll write in this chapter will be a test harness that tests the filter on a corpus of messages of known types, using a certain fraction for training and then measuring how accurate the filter is when classifying the remainder.

Testing the Filter

To test the filter, you need a corpus of messages of known types. You can use messages lying around in your inbox, or you can grab one of the corpora available on the Web. For instance, the SpamAssassin corpus[12] contains several thousand messages hand classified as spam, easy ham, and hard ham. To make it easy to use whatever files you have, you can define a test rig that's driven off an array of file/type pairs. You can define a function that takes a filename and a type and adds it to the corpus like this:

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

The value of `corpus` should be an adjustable vector with a fill pointer. For instance, you can make a new corpus like this:

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

If you have the hams and spams already segregated into separate directories, you might want to add all the files in a directory as the same type. This function, which uses the `list-directory` function from Chapter 15, will do the trick:

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

For instance, suppose you have a directory `mail` containing two subdirectories, `spam` and `ham`, each containing messages of the indicated type; you can add all the files in those two directories to `*corpus*` like this:

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

Now you need a function to test the classifier. The basic strategy will be to select a random chunk of the corpus to train on and then test the corpus by classifying the remainder of the corpus, comparing the classification returned by the `classify` function to the known classification. The main thing you want to know is how accurate the classifier is--what percentage of the messages are classified correctly? But you'll probably also be interested in what messages were misclassified and in what direction--were there more false positives or more false negatives? To make it easy to perform different analyses of the classifier's behavior, you should define the testing functions to build a list of raw results, which you can then analyze however you like.

The main testing function might look like this:

```
(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)))
```

This function starts by clearing out the feature database.[13] Then it shuffles the corpus, using a function you'll implement in a moment, and figures out, based on the `testing-fraction` parameter, how many messages it'll train on and how many it'll reserve for testing. The two helper functions `train-from-corpus` and `test-from-corpus` will both take `:start` and `:end` keyword parameters, allowing them to operate on a subsequence of the given corpus.

The `train-from-corpus` function is quite simple--simply loop over the appropriate part of the corpus, use **`DESTRUCTURING-BIND`** to extract the filename and type from the list found in each element, and then pass the text of the named file and the type to `train`. Since some mail messages, such as those with attachments, are quite large, you should limit the number of characters it'll take from the message. It'll obtain the text with a function `start-of-file`, which you'll implement in a moment, that takes a filename and a maximum number of characters to return. `train-from-corpus` looks like this:

```
(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))))
```

The `test-from-corpus` function is similar except you want to return a list containing the results of each classification so you can analyze them after the fact. Thus, you should capture both the classification and score returned by `classify` and then collect a list of the filename, the actual type, the type returned by `classify`, and the score. To make the results more human readable, you can include keywords in the list to indicate which values are which.

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

A Couple of Utility Functions

To finish the implementation of `test-classifier`, you need to write the two utility functions that don't really have anything particularly to do with spam filtering, `shuffle-vector` and `start-of-file`.

An easy and efficient way to implement `shuffle-vector` is using the Fisher-Yates algorithm.[14] You can start by implementing a function, `nshuffle-vector`, that shuffles a

vector in place. This name follows the same naming convention of other destructive functions such as **NCONC** and **NREVERSE**. It looks like this:

```
(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)
```

The nondestructive version simply makes a copy of the original vector and passes it to the destructive version.

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

The other utility function, `start-of-file`, is almost as straightforward with just one wrinkle. The most efficient way to read the contents of a file into memory is to create an array of the appropriate size and use **READ-SEQUENCE** to fill it in. So it might seem you could make a character array that's either the size of the file or the maximum number of characters you want to read, whichever is smaller. Unfortunately, as I mentioned in Chapter 14, the function **FILE-LENGTH** isn't entirely well defined when dealing with character streams since the number of characters encoded in a file can depend on both the character encoding used and the particular text in the file. In the worst case, the only way to get an accurate measure of the number of characters in a file is to actually read the whole file. Thus, it's ambiguous what **FILE-LENGTH** should do when passed a character stream; in most implementations, **FILE-LENGTH** always returns the number of octets in the file, which may be greater than the number of characters that can be read from the file.

However, **READ-SEQUENCE** returns the number of characters actually read. So, you can attempt to read the number of characters reported by **FILE-LENGTH** and return a substring if the actual number of characters read was smaller.

```
(defun start-of-file (file max-chars)
  (with-open-file (in file)
    (let* ((length (min (file-length in) max-chars))
           (text (make-string length))
           (read (read-sequence text in)))
      (if (< read length)
        (subseq text 0 read)
        text))))
```

Analyzing the Results

Now you're ready to write some code to analyze the results generated by `test-classifier`. Recall that `test-classifier` returns the list returned by `test-from-corpus` in which each element is a plist representing the result of classifying one file. This plist contains the name of the file, the actual type of the file, the classification, and the score returned by `classify`. The first bit of analytical code you should write is a function that returns a symbol indicating

whether a given result was correct, a false positive, a false negative, a missed ham, or a missed spam. You can use **DESTRUCTURING-BIND** to pull out the `:type` and `:classification` elements of an individual result list (using **&allow-other-keys** to tell **DESTRUCTURING-BIND** to ignore any other key/value pairs it sees) and then use nested **ECASE** to translate the different pairings into a single symbol.

```
(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))))))
```

You can test out this function at the REPL.

```
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
```

Having this function makes it easy to slice and dice the results of `test-classifier` in a variety of ways. For instance, you can start by defining predicate functions for each type of result.

```
(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))
```

With those functions, you can easily use the list and sequence manipulation functions I discussed in Chapter 11 to extract and count particular kinds of results.

```
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))
```

You can also use the symbols returned by `result-type` as keys into a hash table or an alist. For instance, you can write a function to print a summary of the counts and percentages of each type of result using an alist that maps each type plus the extra symbol `total` to a count.

```
(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))))))
```

This function will give output like this when passed a list of results generated by `test-classifier`:

```
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
```

And as a last bit of analysis you might want to look at why an individual message was classified the way it was. The following functions will show you:

```
(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))
```

What's Next

Obviously, you could do a lot more with this code. To turn it into a real spam-filtering application, you'd need to find a way to integrate it into your normal e-mail infrastructure. One approach that would make it easy to integrate with almost any e-mail client is to write a bit of code to act as a POP3 proxy--that's the protocol most e-mail clients use to fetch mail from mail servers. Such a proxy would fetch mail from your real POP3 server and serve it to your mail client after either tagging spam with a header that your e-mail client's filters can easily recognize or simply putting it aside. Of course, you'd also need a way to communicate with the filter about misclassifications--as long as you're setting it up as a server, you could also provide a Web interface. I'll talk about how to write Web interfaces in Chapter 26, and you'll build one, for a different application, in Chapter 29.

Or you might want to work on improving the basic classification--a likely place to start is to make `extract-features` more sophisticated. In particular, you could make the tokenizer smarter about the internal structure of e-mail--you could extract different kinds of features for words appearing in the body versus the message headers. And you could decode various kinds of message encoding such as base 64 and quoted printable since spammers often try to obfuscate their message with those encodings.

But I'll leave those improvements to you. Now you're ready to head down the path of building a streaming MP3 server, starting by writing a general-purpose library for parsing binary files.

---

[1]Available at `http://www.paulgraham.com/spam.html` and also in *Hackers & Painters: Big Ideas from the Computer Age* (O'Reilly, 2004)

[2]There has since been some disagreement over whether the technique Graham described was actually "Bayesian." However, the name has stuck and is well on its way to becoming a synonym for "statistical" when talking about spam filters.

[3]It would, however, be poor form to distribute a version of this application using a package starting with `com.gigamonkeys` since you don't control that domain.

[4]A version of CL-PPCRE is included with the book's source code available from the book's Web site. Or you can download it from Weitz's site at `http://www.weitz.de/cl-ppcre/`.

[5]The main reason to use **`PRINT-UNREADABLE-OBJECT`** is that it takes care of signaling the appropriate error if someone tries to print your object readably, such as with the `~S` **`FORMAT`** directive.

[6]**`PRINT-UNREADABLE-OBJECT`** also signals an error if it's used when the printer control variable **`*PRINT-READABLY*`** is true. Thus, a **`PRINT-OBJECT`** method consisting solely of a **`PRINT-UNREADABLE-OBJECT`** form will correctly implement the **`PRINT-OBJECT`** contract with regard to **`*PRINT-READABLY*`**.

[7]If you decide later that you do need to have different versions of `increment-feature` for different classes, you can redefine `increment-count` as a generic function and this function as a method specialized on `word-feature`.

[8]Technically, the key in each clause of a **`CASE`** or **`ECASE`** is interpreted as a *list designator*, an object that designates a list of objects. A single nonlist object, treated as a list designator, designates a list containing just that one object, while a list designates itself. Thus, each clause can have multiple keys; **`CASE`** and **`ECASE`** will select the clause whose list of keys contains the value of the key form. For example, if you wanted to make `good` a synonym for `ham` and `bad` a synonym for `spam`, you could write `increment-count` like this:

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

[9]Speaking of mathematical nuances, hard-core statisticians may be offended by the sometimes loose use of the word *probability* in this chapter. However, since even the pros, who are divided between the Bayesians and the frequentists, can't agree on what a probability is, I'm not going to worry about it. This is a book about programming, not statistics.

[10]Robinson's articles that directly informed this chapter are "A Statistical Approach to the Spam Problem" (published in the *Linux Journal* and available at `http://www.linuxjournal.com/ article.php?sid=6467` and in a shorter form on Robinson's blog at `http://radio.weblogs.com/ 0101454/stories/2002/09/16/spamDetection.html`) and "Why Chi? Motivations for the Use of Fisher's Inverse Chi-Square Procedure in Spam Classification" (available at `http://garyrob.blogs.com/ whychi93.pdf`). Another article that may be useful is "Handling Redundancy in Email Token Probabilities" (available at `http://garyrob.blogs.com//handlingtokenredundancy94.pdf`). The archived mailing lists of the SpamBayes project (`http://spambayes.sourceforge.net/`) also contain a lot of useful information about different algorithms and approaches to testing spam filters.

[11]Techniques that combine nonindependent probabilities as though they were, in fact, independent, are called *naive Bayesian*. Graham's original proposal was essentially a naive Bayesian classifier with some "empirically derived" constant factors thrown in.

[12]Several spam corpora including the SpamAssassin corpus are linked to from `http://nexp.cs.pdx.edu/~psam/cgi-bin/view/PSAM/CorpusSets`.

[13]If you wanted to conduct a test without disturbing the existing database, you could bind `*feature-database*`, `*total-spams*`, and `*total-hams*` with a **`LET`**, but then you'd have no way of looking at the database after the fact--unless you returned the values you used within the function.

[14]This algorithm is named for the same Fisher who invented the method used for combining probabilities and for Frank Yates, his coauthor of the book *Statistical Tables for Biological, Agricultural and Medical Research* (Oliver & Boyd, 1938) in which, according to Knuth, they provided the first published description of the algorithm.