Programming Collective Intelligence

Building Smart Web 2.0 Applications

Toby Segaran

O'REILLY®

Beijing · Cambridge · Farnham · Köln · Sebastopol · Taipei · Tokyo

CHAPTER 6

Document Filtering

This chapter will demonstrate how to classify documents based on their contents, a very practical application of machine intelligence and one that is becoming more widespread. Perhaps the most useful and well-known application of document filtering is the elimination of spam. A big problem with the wide availability of email and the extremely low cost of sending email messages is that anyone whose address gets into the wrong hands is likely to receive unsolicited commercial email messages, making it difficult for them to read the messages that are actually of interest.

The problem of spam does not just apply to email, of course. Web sites have gotten more interactive over time, soliciting comments from users or asking them to create original content, which has compounded the spam problem. Public message boards like Yahoo! Groups and Usenet have long been victims of postings that are unrelated to the board's subject or that hawk dubious products. Blogs and Wikis are now experiencing the same problem. When building an application that allows the general public to contribute, you should always have a strategy for dealing with spam.

The algorithms described in this chapter are not specific to dealing with spam. Since they solve the more general problem of learning to recognize whether a document belongs in one category or another, they can be used for less unsavory purposes. One example might be automatically dividing your inbox into social and work-related email, based on the contents of the messages. Another possibility is identifying email messages that request information and automatically forwarding them to the most competent person to answer them. The example at the end of this chapter will demonstrate automatically filtering entries from an RSS feed into different categories.

Filtering Spam

Early attempts to filter spam were all rule-based classifiers, where a person would design a set of rules that was supposed to indicate whether or not a message was spam. Rules typically included things like overuse of capital letters, words related to pharmaceutical products, or particularly garish HTML colors. The problems with

rule-based classifiers quickly became apparent—spammers learned all the rules and stopped exhibiting the obvious behaviors to get around the filters, while people whose parents never learned to turn off the Caps Lock key found their good email messages being classified as spam.

The other problem with rule-based filters is that what can be considered spam varies depending on where it's being posted and for whom it is being written. Keywords that would strongly indicate spam for one particular user, message board, or Wiki may be quite normal for others. To solve this problem, this chapter will look at programs that *learn*, based on you telling them what is spam email and what isn't, both initially and as you receive more messages. By doing this, you can create separate instances and datasets for individual users, groups, or sites that will each develop their own ideas about what is spam and what isn't.

Documents and Words

The classifier that you will be building needs *features* to use for classifying different items. A feature is anything that you can determine as being either present or absent in the item. When considering documents for classification, the items are the documents and the features are the words in the document. When using words as features, the assumption is that some words are more likely to appear in spam than in nonspam, which is the basic premise underlying most spam filters. Features don't have to be individual words, however; they can be word pairs or phrases or anything else that can be classified as absent or present in a particular document.

Create a new file called *docclass.py*, and add a function called getwords to extract the features from the text:

```
import re
import math

def getwords(doc):
  splitter=re.compile('\\W*')
  # Split the words by non-alpha characters
  words=[s.lower( ) for s in splitter.split(doc)
         if len(s)>2 and len(s)<20]

  # Return the unique set of words only
  return dict([(w,1) for w in words])
```

This function breaks up the text into words by dividing the text on any character that isn't a letter. This leaves only actual words, all converted to lowercase.

Determining which features to use is both very tricky and very important. The features must be common enough that they appear frequently, but not so common that they appear in every single document. In theory, the entire text of the document

could be a feature, but that would almost certainly be useless unless you receive the exact same email message over and over. At the other extreme, the features could be individual characters, but since they would all likely appear in every email message, they would do a poor job of separating wanted from unwanted documents. Even the choice to use words as features poses questions of exactly how to divide words, which punctuation to include, and whether header information should be included.

The other thing to consider when deciding on features is how well they will divide the set of documents into the target categories. For example, the code for getwords above reduces the total number of features by converting them to lowercase. This means it will recognize that a capitalized word at the start of a sentence is the same as when that word is all lowercase in the middle of a sentence—a good thing, since words with different capitalization usually have the same meaning. However, this function will completely miss the SHOUTING style used in many spam messages, which may be vital for dividing the set into spam and nonspam. Another alternative might be to have a feature that is deemed present if more than half the words are uppercase.

As you can see, the choice of feature set involves many tradeoffs and is subject to endless tweaking. For now, you can use the simple getwords function given; later in the chapter, you'll see some ideas for improving the extraction of features.

Training the Classifier

The classifiers discussed in this chapter learn how to classify a document by being trained. Many of the other algorithms in this book, such as the neural network you saw in Chapter 4, learn by reading examples of correct answers. The more examples of documents and their correct classifications it sees, the better the classifier will get at making predictions. The classifier is also specifically designed to start off very uncertain and increase in certainty as it learns which features are important for making a distinction.

The first thing you'll need is a class to represent the classifier. This class will encapsulate what the classifier has learned so far. The advantage of structuring the module this way is that you can instantiate multiple classifiers for different users, groups, or queries, and train them differently to respond to a particular group's needs. Create a class called `classifier` in *docclass.py*:

```
class classifier:
  def __init__(self,getfeatures,filename=None):
    # Counts of feature/category combinations
    self.fc={}
    # Counts of documents in each category
    self.cc={}
    self.getfeatures=getfeatures
```

The three instance variables are fc, cc, and getfeatures. The fc variable will store the counts for different features in different classifications. For example:

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

This indicates that the word "the" has appeared in documents classified as bad three times, and in documents that were classified as good three times. The word "Python" has only appeared in good documents.

The cc variable is a dictionary of how many times every classification has been used. This is needed for the probability calculations that we'll discuss shortly. The final instance variable, getfeatures, is the function that will be used to extract the features from the items being classified—in this example, it is the getwords function you just defined.

The methods in the class won't use the dictionaries directly because this restricts potential options for storing the training data in a file or database. Create these helper methods to increment and get the counts:

```
# Increase the count of a feature/category pair
def incf(self,f,cat):
  self.fc.setdefault(f,{})
  self.fc[f].setdefault(cat,0)
  self.fc[f][cat]+=1

# Increase the count of a category
def incc(self,cat):
  self.cc.setdefault(cat,0)
  self.cc[cat]+=1

# The number of times a feature has appeared in a category
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

# The number of items in a category
def catcount(self,cat):
  if cat in self.cc:
    return float(self.cc[cat])
  return 0

# The total number of items
def totalcount(self):
  return sum(self.cc.values())

# The list of all categories
def categories(self):
  return self.cc.keys()
```

The train method takes an item (a document in this case) and a classification. It uses the getfeatures function of the class to break the item into its separate features. It then calls incf to increase the counts for this classification for every feature. Finally, it increases the total count for this classification:

```
def train(self,item,cat):
  features=self.getfeatures(item)
  # Increment the count for every feature with this category
  for f in features:
    self.incf(f,cat)

  # Increment the count for this category
  self.incc(cat)
```

You can check to see if your class is working properly by starting a new Python session and importing this module:

```
$ 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
```

At this point, it's useful to have a method to dump some sample training data into the classifier so that you don't have to train it manually every time you create it. Add this function to the start of *docclass.py*:

```
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')
```

Calculating Probabilities

You now have counts for how often an email message appears in each category, so the next step is to convert these numbers into probabilities. A probability is a number between 0 and 1, indicating how likely an event is. In this case, you can calculate the probability that a word is in a particular category by dividing the number of times the word appears in a document in that category by the total number of documents in that category.

Add a method called fprob to the classifier class:

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

```
        # The total number of times this feature appeared in this
        # category divided by the total number of items in this category
        return self.fcount(f,cat)/self.catcount(cat)
```

This is called *conditional probability*, and is usually written as *Pr(A | B)* and spoken "the probability of A given B." In this example, the numbers you have now are *Pr(word | classification)*; that is, for a given classification you calculate the probability that a particular word appears.

You can test this function in your Python session:

```
>>> reload(docclass)
<module 'docclass' from 'docclass.py'>
>>> cl=docclass.classifier(docclass.getwords)
>>> docclass.sampletrain(cl)
>>> cl.fprob('quick','good')
0.6666666666666663
```

You can see that the word "quick" appears in two of the three documents classified as good, which means there's a probability of *Pr(quick | good)* = 0.666 (a 2/3 chance) that a good document will contain that word.

Starting with a Reasonable Guess

The `fprob` method gives an accurate result for the features and classifications it has seen so far, but it has a slight problem—using only the information it has seen so far makes it incredibly sensitive during early training and to words that appear very rarely. In the sample training data, the word "money" only appears in one document and is classified as bad because it is a casino ad. Since the word "money" is in one bad document and no good ones, the probability that it will appear in the good category using `fprob` is now 0. This is a bit extreme, since "money" might be a perfectly neutral word that just happens to appear first in a bad document. It would be much more realistic for the value to gradually approach zero as a word is found in more and more documents with the same category.

To get around this, you'll need to decide on an *assumed probability*, which will be used when you have very little information about the feature in question. A good number to start with is 0.5. You'll also need to decide how much to *weight* the assumed probability—a weight of 1 means the assumed probability is weighted the same as one word. The weighted probability returns a weighted average of `getprobability` and the assumed probability.

In the "money" example, the weighted probability for the word "money" starts at 0.5 for all categories. After the classifier is trained with one bad document and finds that "money" fits into the bad category, its probability becomes 0.75 for bad. This is because:

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

Add the method for weightedprob to your classifier class:

```
def weightedprob(self,f,cat,prf,weight=1.0,ap=0.5):
  # Calculate current probability
  basicprob=prf(f,cat)

  # Count the number of times this feature has appeared in
  # all categories
  totals=sum([self.fcount(f,c) for c in self.categories()])

  # Calculate the weighted average
  bp=((weight*ap)+(totals*basicprob))/(weight+totals)
  return bp
```

You can now test the function in your Python session. Reload the module and rerun the sampletrain method, since creating a new instance of the class will wipe out its existing training:

```
>>> 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
```

As you can see, rerunning the sampletrain method makes the classifier even more confident of the various word probabilities as they get pulled further from their assumed probability.

The assumed probability of 0.5 was chosen simply because it is halfway between 0 and 1. However, it's possible that you might have better background information than that, even on a completely untrained classifier. For example, one person who begins training a spam filter can use probabilities from other people's already-trained spam filters as the assumed probabilities. The user still gets a spam filter personalized for him, but the filter is better able to handle words that it has come across very infrequently.

A Naïve Classifier

Once you have the probabilities of a document in a category containing a particular word, you need a way to combine the individual word probabilities to get the probability that an entire document belongs in a given category. This chapter will consider two different classification methods. Both of them work in most situations, but they vary slightly in their level of performance for specific tasks. The classifier covered in this section is called a *naïve Bayesian classifier*.

This method is called *naïve* because it assumes that the probabilities being combined are *independent* of each other. That is, the probability of one word in the document being in a specific category is unrelated to the probability of the other words being in that category. This is actually a false assumption, since you'll probably find that documents containing the word "casino" are much more likely to contain the word "money" than documents about Python programming are.

This means that you can't actually use the probability created by the naïve Bayesian classifier as the actual probability that a document belongs in a category, because the assumption of independence makes it inaccurate. However, you can *compare* the results for different categories and see which one has the highest probability. In real life, despite the underlying flawed assumption, this has proven to be a surprisingly effective method for classifying documents.

Probability of a Whole Document

To use the naïve Bayesian classifier, you'll first have to determine the probability of an entire document being given a classification. As discussed earlier, you're going to assume the probabilities are independent, which means you can calculate the probability of all of them by multiplying them together.

For example, suppose you've noticed that the word "Python" appears in 20 percent of your bad documents—*Pr(Python | Bad)* = 0.2—and that the word "casino" appears in 80 percent of your bad documents *(Pr(Casino | Bad)* = 0.8). You would then expect the independent probability of both words appearing in a bad document—*Pr(Python & Casino | Bad)*—to be $0.8 \times 0.2 = 0.16$. From this you can see that calculating the entire document probability is just a matter of multiplying together all the probabilities of the individual words in that document.

In *docclass.py*, create a subclass of `classifier` called `naivebayes`, and create a `docprob` method that extracts the features (words) and multiplies all their probabilities together to get an overall probability:

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

    # Multiply the probabilities of all the features together
    p=1
    for f in features: p*=self.weightedprob(f,cat,self.fprob)
    return p
```

You now know how to calculate *Pr(Document | Category)*, but this isn't very useful by itself. In order to classify documents, you really need *Pr(Category | Document)*. In other words, given a *specific* document, what's the probability that it fits into this category? Fortunately, a British mathematician named Thomas Bayes figured out how to do this about 250 years ago.

A Quick Introduction to Bayes' Theorem

Bayes' Theorem is a way of flipping around conditional probabilities. It's usually written as:

$$Pr(A \mid B) = Pr(B \mid A) \times Pr(A)/Pr(B)$$

In the example, this becomes:

$$Pr(Category \mid Document) = Pr(Document \mid Category) \times Pr(Category) / Pr(Document)$$

The previous section showed how to calculate *Pr(Document | Category)*, but what about the other two values in the equation? Well, *Pr(Category)* is the probability that a randomly selected document will be in this category, so it's just the number of documents in the category divided by the total number of documents.

As for *Pr(Document)*, you could calculate it, but that would be unnecessary effort. Remember that the results of this calculation will not be used as a real probability. Instead, the probability for each category will be calculated separately, and then all the results will be compared. Since *Pr(Document)* is the same no matter what category the calculation is being done for, it will scale the results by the exact same amount, so you can safely ignore that term.

The prob method calculates the probability of the category, and returns the product of *Pr(Document | Category)* and *Pr(Category)*. Add this method to the naivebayes class:

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

Try this function in Python to see how the numbers vary for different strings and categories:

```
>>> 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
```

Based on the training data, the phrase "quick rabbit" is considered a much better candidate for the good category than the bad.

Choosing a Category

The final step in building the naïve Bayes classifier is actually deciding in which category a new item belongs. The simplest approach would be to calculate the probability of this item being in each of the different categories and to choose the category with the best probability. If you were just trying to decide the best place to put something, this would be a feasible strategy, but in many applications the categories can't be considered equal, and in some applications it's better for the classifier to admit that it doesn't know the answer than to decide that the answer is the category with a marginally higher probability.

In the case of spam filtering, it's much more important to avoid having good email messages classified as spam than it is to catch every single spam message. The occasional spam message in your inbox can be tolerated, but an important email that is automatically filtered to junk mail might get overlooked completely. If you have to search through your junk mail folder for important email messages, there's really no point in having a spam filter.

To deal with this problem, you can set up a minimum threshold for each category. For a new item to be classified into a particular category, its probability must be a specified amount larger than the probability for any other category. This specified amount is the *threshold*. For spam filtering, the threshold to be filtered to bad could be 3, so that the probability for bad would have to be 3 times higher than the probability for good. The threshold for good could be set to 1, so anything would be good if the probability were at all better than for the bad category. Any message where the probability for bad is higher, but not 3 times higher, would be classified as unknown.

To set up these thresholds, add a new instance variable to `classifier` by modifying the initialization method:

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

Add some simple methods to set and get the values, returning 1.0 as the default:

```
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]
```

Now you can build the `classify` method. It will calculate the probability for each category, and will determine which one is the largest and whether it exceeds the next largest by more than its threshold. If none of the categories can accomplish this, the method just returns the default values. Add this method to `classifier`:

```
def classify(self,item,default=None):
  probs={}
  # Find the category with the highest probability
```

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

    # Make sure the probability exceeds threshold*next best
    for cat in probs:
      if cat==best: continue
      if probs[cat]*self.getthreshold(best)>probs[best]: return default
    return best
```

You're done! You've now built a complete system for classifying documents. This can also be extended to classify other things by creating different methods for getting the features. Try out the classifier in your Python session:

```
>>> 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'
```

Of course, you can alter the thresholds and see how the results are affected. Some spam-filtering plug-ins give users control over the thresholds so they can be adjusted if they're letting too much spam into the inbox or categorizing good messages as spam. The thresholds will also be different for other applications that involve document filtering; sometimes all categories will be equal, or filtering to "unknown" will be unacceptable.

The Fisher Method

The *Fisher method*, named for R. A. Fisher, is an alternative method that's been shown to give very accurate results, particularly for spam filtering. This is the method used by *SpamBayes*, an Outlook plug-in written in Python. Unlike the naïve Bayesian filter, which uses the feature probabilities to create a whole document probability, the Fisher method calculates the probability of a category for each feature in the document, then combines the probabilities and tests to see if the set of

probabilities is more or less likely than a random set. This method also returns a probability for each category that can be compared to the others. Although this is a more complex method, it is worth learning because it allows much greater flexibility when choosing cutoffs for categorization.

Category Probabilities for Features

With the naïve Bayesian filter discussed earlier, you combined all of the *Pr(feature | category)* results to get an overall document probability, and then flipped it around at the end. In this section, you'll begin by calculating how likely it is that a document fits into a category given that a particular feature is in that document—that is, *Pr(category | feature)*. If the word "casino" appears in 500 documents, and 499 of those are in the bad category, "casino" will get a score very close to 1 for bad.

The normal way to calculate *Pr(category | feature)* would be:

> *(number of documents in this category with the feature) / (total number of documents with the feature)*

This calculation doesn't take into account the possibility that you may have received far more documents in one category than in another. If you have many good documents and only a few bad ones, a word that appears in all your bad documents will likely have a high probability for bad, even though the message is just as likely to be good. The methods perform better when they assume that in the future you will receive equal numbers of documents in each category, because this allows them to take advantage of the features that distinguish the categories.

To perform this normalization, the method calculates three things:

- *clf = Pr(feature | category) for this category*
- *freqsum = Sum of Pr(feature | category) for all the categories*
- *cprob = clf / (clf+nclf)*

Create a new subclass of `classifier` called `fisherclassifier` in *docclass.py* and add this method:

```
class fisherclassifier(classifier):
  def cprob(self,f,cat):
    # The frequency of this feature in this category
    clf=self.fprob(f,cat)
    if clf==0: return 0

    # The frequency of this feature in all the categories
    freqsum=sum([self.fprob(f,c) for c in self.categories()])

    # The probability is the frequency in this category divided by
    # the overall frequency
    p=clf/(freqsum)

    return p
```

This function will return the probability that an item with the specified feature belongs in the specified category, assuming there will be an equal number of items in each category. You can see what these numbers actually look like in your Python session:

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

This method shows us that documents containing the word "casino" have a 0.9 probability of being spam. This matches the training data, but again it suffers from the problem of only having been exposed to the words a small number of times, and it might be greatly overestimating the probabilities. So, like last time, it's better to use the weighted probability, which starts all probabilities at 0.5 and allows them to move toward other probabilities as the class is trained.

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

Combining the Probabilities

You now have to combine the probabilities of the individual features to come up with an overall probability. In theory, you can just multiply them all together, which gives you a probability that you can use to compare this category with other categories. Of course, since the features aren't independent, this won't be a real probability, but it works much like the Bayesian classifier that you built in the previous section. The value returned by the Fisher method is a much better estimate of probability, which can be very useful when reporting results or deciding cutoffs.

The Fisher method involves multiplying all the probabilities together, then taking the natural log (*math.log* in Python), and then multiplying the result by –2. Add this method to `fisherclassifier` to do this calculation:

```
def fisherprob(self,item,cat):
  # Multiply all the probabilities together
  p=1
  features=self.getfeatures(item)
  for f in features:
    p*=(self.weightedprob(f,cat,self.cprob))

  # Take the natural log and multiply by -2
  fscore=-2*math.log(p)

  # Use the inverse chi2 function to get a probability
  return self.invchi2(fscore,len(features)*2)
```

Fisher showed that if the probabilities were independent and random, the result of this calculation would fit a *chi-squared distribution*. You would expect an item that doesn't belong in a particular category to contain words of varying feature probabilities for that category (which would appear somewhat random), and an item that does belong in that category to have many features with high probabilities. By feeding the result of the Fisher calculation to the *inverse chi-square function*, you get the probability that a random set of probabilities would return such a high number.

Add the inverse chi-square function to the `fisherclassifier` class:

```
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)
```

Again, you can try this function in your Python session and see how the Fisher method scores some example strings:

```
>>> 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
```

As you can see, these results are always between 0 and 1. They are, on their own, a good measure of how well a document fits into a category. Because of this, the classifier itself can be more sophisticated.

Classifying Items

You can use the values returned by `fisherprob` to determine the classification. Rather than having multiplication thresholds like the Bayesian filter, you can specify the lower bounds for each classification. The classifier will then return the highest value that's within its bounds. In the spam filter, you might set the minimum for the bad classification quite high, perhaps 0.6. You might set the minimum for the good classification a lot lower, perhaps 0.2. This would minimize the chance of good email messages being classified as bad, and allow a few spam email messages into the inbox. Anything that scores lower than 0.2 for good and lower than 0.6 for bad would be classified as unknown.

Create an init method in fisherclassifier with another variable to store the cutoffs:

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

Add a couple of methods for getting and setting these values, with a default value of 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]
```

Finally, add a method to calculate the probabilities for each category and determine the best result that exceeds the specified minimum:

```
def classify(self,item,default=None):
  # Loop through looking for the best result
  best=default
  max=0.0
  for c in self.categories( ):
    p=self.fisherprob(item,c)
    # Make sure it exceeds its minimum
    if p>self.getminimum(c) and p>max:
      best=c
      max=p
  return best
```

Now you can try the classifier on the test data using the Fisher scoring method. Enter the following in your Python session:

```
>>> 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')
>>>
```

The results are similar to those of the naïve Bayesian classifier. The Fisher classifier is believed to perform better for spam filtering in practice; however, this is unlikely to be apparent with such a small training set. The classifier you should use depends on your application, and there's no easy way to predict in advance which will perform better or what cutoffs you should use. Fortunately, the code given here should make it very easy to experiment with the two algorithms and with different settings.

Persisting the Trained Classifiers

In any real-world application, it's unlikely that all the training and classification will be done entirely in one session. If the classifier is used as part of a web-based application, it will probably have to save any training that the user does while using the application, and then restore the training data the next time the user logs on.

Using SQLite

This section will show you how to persist the training information for your classifier using a database, in this case, SQLite. If your application involves many users concurrently training and querying the classifier, it's probably wise to store the counts in a database. SQLite is the same database we used in Chapter 4. You'll need to download and install pysqlite if you haven't already; details on how to do this are in Appendix A. Accessing SQLite from Python is similar to accessing other databases, so this should adapt quite easily.

To import pysqlite, add this statement to the top of *docclass.py*:

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

The code in this section will replace the dictionary structures currently in the `classifier` class with a persistent data store. Add a classifier method that opens a database for this classifier and creates tables if necessary. The tables match the structure of the dictionaries that they replace:

```
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)')
```

If you're planning to adapt this for another database, you may need to modify the `create table` statements to work with the system you're using.

You'll have to replace all the helper methods for getting and incrementing the counts:

```
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])
```

The methods that get the list of all the categories and the total number of documents should also be replaced:

```
  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]
```

Finally, you'll need to add a `commit` after training so that the data is stored after all the counts have been updated. Add this line to the end of the `train` method in `classifier`:

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

That's it! After you initialize a classifier, you need to call the `setdb` method with the name of a database file. All training will be automatically stored and can be used by anyone else. You can even use the training from one type of classifier to do classifications in another type:

```
>>> 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'
```

Filtering Blog Feeds

To try out the classifier on real data and show the different ways it can be used, you can apply it to entries from a blog or other RSS feed. To do this, you'll need to get the Universal Feed Parser, which we used in Chapter 3. If you haven't already downloaded it, you can get it from *http://feedparser.org*. More information on installing the Feed Parser is given in Appendix A.

Although a blog will not necessarily contain spam in its entries, many blogs contain some articles that interest you and some that don't. This can be because you only want to read articles in a certain category or by a certain writer, but it's often more complicated than that. Again, you can set up specific rules for things that do and do not interest you—maybe you read a gadget blog and are not interested in entries that contain the word "cell phone"—but it's much less work to use the classifier you've built to figure out these rules for you.

A benefit of classifying entries in an RSS feed is that if you use a blog-searching tool like Google Blog Search, you can set up the results of your searches in a feed reader. Many people do this to track products, things that interest them, even their own names. You'll find, though, that spam-based or useless blogs trying to make money from blog traffic can also appear in these searches.

For this example, you can use any feed you like, although many feeds have too few entries to do any effective training. This particular example uses the results of a Google Blog Search for the word "Python" in RSS format. You can download these results from *http://kiwitobes.com/feeds/python_search.xml*.

Create a new file called *feedfilter.py* and add the following code:

```
import feedparser
import re

# Takes a filename of URL of a blog feed and classifies the entries
def read(feed,classifier):
  # Get feed entries and loop over them
  f=feedparser.parse(feed)
  for entry in f['entries']:
    print
    print '-----'
    # Print the contents of the entry
    print 'Title:     '+entry['title'].encode('utf-8')
    print 'Publisher: '+entry['publisher'].encode('utf-8')
    print
    print entry['summary'].encode('utf-8')


    # Combine all the text to create one item for the classifier
    fulltext='%s\n%s\n%s' % (entry['title'],entry['publisher'],entry['summary'])
```

```
    # Print the best guess at the current category
    print 'Guess: '+str(classifier.classify(fulltext))

    # Ask the user to specify the correct category and train on that
    cl=raw_input('Enter category: ')
    classifier.train(fulltext,cl)
```

This function loops over all the entries and uses the classifier to get a best guess at the classification. It shows this best guess to the user and then asks what the correct category should have been. When you run this with a new classifier, the guesses will at first be random, but they should improve over time.

The classifier you have built is completely generic. Although spam filtering was used as an example to help explain what each piece of code does, the categories can be anything. If you're using *python_search.xml*, you might have four categories—one for the programming language, one for Monty Python, one for python snakes, and one for everything else. Try running the interactive filter in your Python session by setting up a classifier and passing it to `feedfilter`:

```
>>> import feedfilter
>>> cl=docclass.fisherclassifier(docclass.getwords)
>>> cl.setdb('python_feed.db')  # Only if you implemented 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
```

You'll see the guesses improving over time. There aren't many samples for snakes, so the classifier often gets them wrong, especially since they are further divided into pet snakes and fashion-related posts. After you've run through the training, you can get probabilities for a specific feature—both the probability of a word given a category and the probability of a category given a word:

```
>>> 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
```

The probabilities for the word "python" are evenly divided, since every entry contains that word. The word "Eric" occurs in 25 percent of entries related to Monty Python and does not occur at all in other entries. Thus, the probability of the word given the category is 0.25, and the probability for the category given the word is 1.0.

Improving Feature Detection

In all the examples so far, the function for creating the list of features uses just a simple nonalphanumeric split to break up the words. The function also converts all words to lowercase, so there's no way to detect the overuse of uppercase words. There are several different ways this can be improved:

- Without actually making uppercase and lowercase tokens completely distinct, use the fact that there are many uppercase words as a feature.
- Use sets of words in addition to individual words.
- Capture more metainformation, such as who sent an email message or what category a blog entry was posted under, and annotate it as metainformation.
- Keep URLs and numbers intact.

Remember that it's not simply a matter of making the features more specific. Features have to occur in multiple documents for them to be of any use to the classifier.

The `classifier` class will take any function as `getfeatures` and run it on the items passed in, expecting a list or dictionary of all the features for that item to be returned. Because it is so generic, you can easily create a function that works on types more complicated than just strings. For example, when classifying entries in a blog feed, you can use a function that takes the whole entry instead of the extracted text and annotates where the different words come from. You can also pull out word pairs from the body of the text and only the individual words from the subject. It's

also probably pointless to tokenize the creator field, since the postings of someone named "John Smith" will not likely tell you anything about the postings of someone else with the first name John.

Add this new feature-extraction function to *feedfilter.py*. Notice that it expects a feed entry and not a string as its parameter:

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

  # Extract the title words and annotate
  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

  # Extract the summary words
  summarywords=[s.lower( ) for s in splitter.split(entry['summary'])
          if len(s)>2 and len(s)<20]

  # Count uppercase words
  uc=0
  for i in range(len(summarywords)):
    w=summarywords[i]
    f[w]=1
    if w.isupper( ): uc+=1

    # Get word pairs in summary as features
    if i<len(summarywords)-1:
      twowords=' '.join(summarywords[i:i+1])
      f[twowords]=1

  # Keep creator and publisher whole
  f['Publisher:'+entry['publisher']]=1

  # UPPERCASE is a virtual word flagging too much shouting
  if float(uc)/len(summarywords)>0.3: f['UPPERCASE']=1

  return f
```

This function extracts the words from the title and the summary, just like getwords did earlier. It marks all the words in the title as such and adds them as features. The words in the summary are added as features, and then pairs of consecutive words are added as well. The function adds the creator and publisher as features without dividing them up, and finally, it counts the number of words in the summary that are uppercase. If more than 30 percent of the words are uppercase, the function adds an additional feature called UPPERCASE to the set. Unlike a rule that says uppercase words mean a particular thing, this is just an additional feature that the classifier can use for training—in some cases, it may decide it's completely useless to distinguish document categories.

If you want to use this new version with filterfeed, you'll have to change the function to pass the entries to the classifier rather than to fulltext. Just change the end to:

```
# Print the best guess at the current category
print 'Guess: '+str(classifier.classify(entry))

# Ask the user to specify the correct category and train on that
cl=raw_input('Enter category: ')
classifier.train(entry,cl)
```

You can then initialize the classifier to use entryfeatures as its feature-extraction function:

```
>>> reload(feedfilter)
<module 'feedfilter' from 'feedfilter.py'>
>>> cl=docclass.fisherclassifier(feedfilter.entryfeatures)
>>> cl.setdb('python_feed.db')  # Only if using the DB version
>>> feedfilter.read('python_search.xml',cl)
```

There's a lot more you can do with features. The basic framework you've built allows you to define a function for extracting features and set up the classifier to use the function. It will classify any object you pass to it as long as the feature-extraction function you specify can return a set of features from the object.

Using Akismet

Akismet is a slight detour from the study of text-classification algorithms, but for a specific class of applications, it may solve your spam-filtering needs with minimal effort and eliminate the need for you to build your own classifier.

Akismet started out as a WordPress plug-in that allowed people to report spam comments posted on their blogs, and to filter new comments based on their similarity to spam reported by other people. Now the API is open and you can query Akismet with any string to find out if Akismet thinks the string is spam.

The first thing you'll need is an Akismet API key, which you can get at *http://akismet.com*. These keys are free for personal use and there are several options available for commercial use. The Akismet API is called with regular HTTP requests, and libraries have been written for various languages. The one used in this section is available at *http://kemayo.wordpress.com/2005/12/02/akismet-py*. Download *akismet.py* and put it in your code directory or in your Python Libraries directory.

Using the API is very simple. Create a new file called *akismettest.py* and add this function:

```
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
```

You now have a method you can call with any string to see if it is similar to those in blog comments. Try it in your Python session:

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

Experiment with different usernames, agents, and IP addresses to see how the results vary.

Because Akismet is primarily used for spam comments posted on blogs, it may not perform well on other types of documents, such as email messages. Also, unlike the classifier, it doesn't allow any tweaking of parameters, nor does it let you see the calculations it uses to come up with the answer. It is, however, very accurate for spam-comment filtering, and it's worth trying on your applications if you are receiving a similar kind of spam because Akismet has a far larger collection of documents for comparison than you are likely to have gathered.

Alternative Methods

Both of the classifiers built in this chapter are examples of *supervised learning methods*, methods that are trained with correct results and gradually get better at making predictions. The *artificial neural network* described in Chapter 4 for weighting search results for ranking purposes was another example of supervised learning. That neural network can be adapted to work on the same problems in this chapter by using the features as inputs and having outputs representing each of the possible classifications. Likewise, *support vector machines*, which are described in Chapter 9, can be applied to the problems in this chapter.

The reason Bayesian classifiers are often used for document classification is that they require far less computing power than other methods do. An email message might have hundreds or even thousands of words in it, and simply updating the counts takes vastly less memory and processor cycles than training a neural network of that size does; as shown, it can be done entirely within a database. Depending on the speed required for training and querying, and on the environment in which it is run, a neural network may be a viable alternative. The complexity of a neural network also brings with it a lack of interpretability; in this chapter you were able to look at the word probabilities and see exactly how much they contribute to the final score, while the connection strengths between the neurons in a network has no equally simple interpretation.

On the other hand, neural networks and support-vector machines have one big advantage over the classifiers presented in this chapter: they can capture more complex relationships between the input features. In a Bayesian classifier, every feature has a probability for each category, and you combine the probabilities to get an overall likelihood. In a neural network, the probability of a feature can change depending on the presence or absence of other features. It may be that you're trying to block online-casino spam but you're also interested in horse betting, in which case the word "casino" is bad unless the word "horse" is somewhere else in the email message. Naïve Bayesian classifiers cannot capture this interdependence, and neural networks can.

Exercises

1. *Varying assumed probabilities.* Change the `classifier` class so it supports different assumed probabilities for different features. Change the `init` method so that it will take another classifier and start with a better guess than 0.5 for the assumed probabilities.
2. *Calculate Pr(Document).* In the naïve Bayesian classifier, the calculation of *Pr(Document)* was skipped since it wasn't required to compare the probabilities. In cases where the features are independent, it can actually be used to calculate the overall probability. How would you calculate *Pr(Document)*?
3. *A POP-3 email filter.* Python comes with a library called *poplib* for downloading email messages. Write a script that downloads email messages from a server and attempts to classify them. What are the different properties of an email message, and how might you build a feature-extraction function to take advantage of these?
4. *Arbitrary phrase length.* This chapter showed you how to extract word pairs as well as individual words. Make the feature extraction configurable to extract up to a specified number of words as a single feature.

5. *Preserving IP addresses*. IP addresses, phone numbers, and other numerical information can be helpful in identifying spam. Modify the feature-extraction function to return these items as features. (IP addresses have periods embedded in them, but you still need to get rid of the periods between sentences.)
6. *Other virtual features*. There are many virtual features like UPPERCASE that can be useful in classifying documents. Documents of excessive length or with a preponderance of long words may also be clues. Implement these as features. Can you think of any others?
7. *Neural network classifier*. Modify the neural network from Chapter 4 to be used for document classification. How do its results compare? Write a program that classifies and trains on documents thousands of times. Time how long it takes with each of the algorithms. How do they compare?