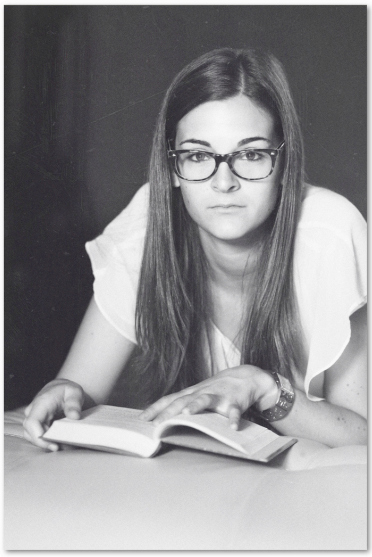
Let's say I want to create a system that can tell whether a person likes or dislikes various food products. We might come up with an idea of having a list of words that would provide evidence that a person likes the product and another list of words that provides evidence that the person doesn't like the product.

'Like' words:
delicious
tasty
good
love
smooth

'Dislike' words:
awful
bland
bad
hate
gritty

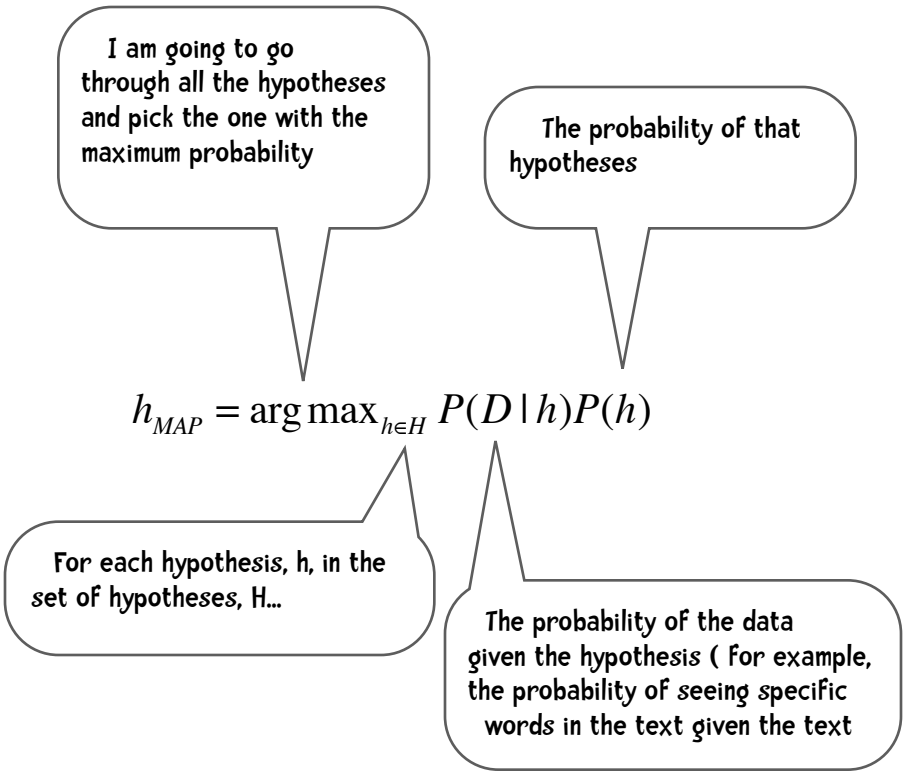
If we are trying to determine if a particular reviewer likes Chobani yogurt or not, we can just count the number of 'like' words and the number of 'dislike' words in their text. We will classify the text based on which number is higher. We can do this for other classification tasks. For example, if we want to decide whether someone is pro-choice or pro-life, we can base it on the words and phrases they use. If they use the phrase 'unborn child' then chances are they are pro-life; if they use fetus they are more likely to be pro-choice. It's not surprising that we can use the occurrence of words to classify text.



Rather than just using raw counts to classify text, let's use the naïve Bayes!!

$$h_{MAP} = \arg\max_{h \in H} P(D \mid h)P(h)$$

Let's dissect that formula!



We will use the naïve Bayes methods that were introduced in the previous chapter. We start with a training data set and, since we are now interested in unstructured text this data set is called **the training corpus**. Each entry in the corpus we will call a document even if it is a 140 character Twitter post. Each document is labeled with its class. So, for example, we might have a corpus of Twitter posts that rated movies. Each post is labeled in some way as a 'favorable' review or 'unfavorable' and we are going to train our classifier using this corpus of labeled documents. The $P(h)$ in the formula above is the probability of these labels. If we have 1,000 documents in our training corpus and 500 of them are favorable reviews and 500 unfavorable then

$$P(favorable) = 0.5 \qquad P(unfavorable) = 0.5$$

When we start with labeled training data it is called 'supervised learning.' Text classification is an example of supervised learning.

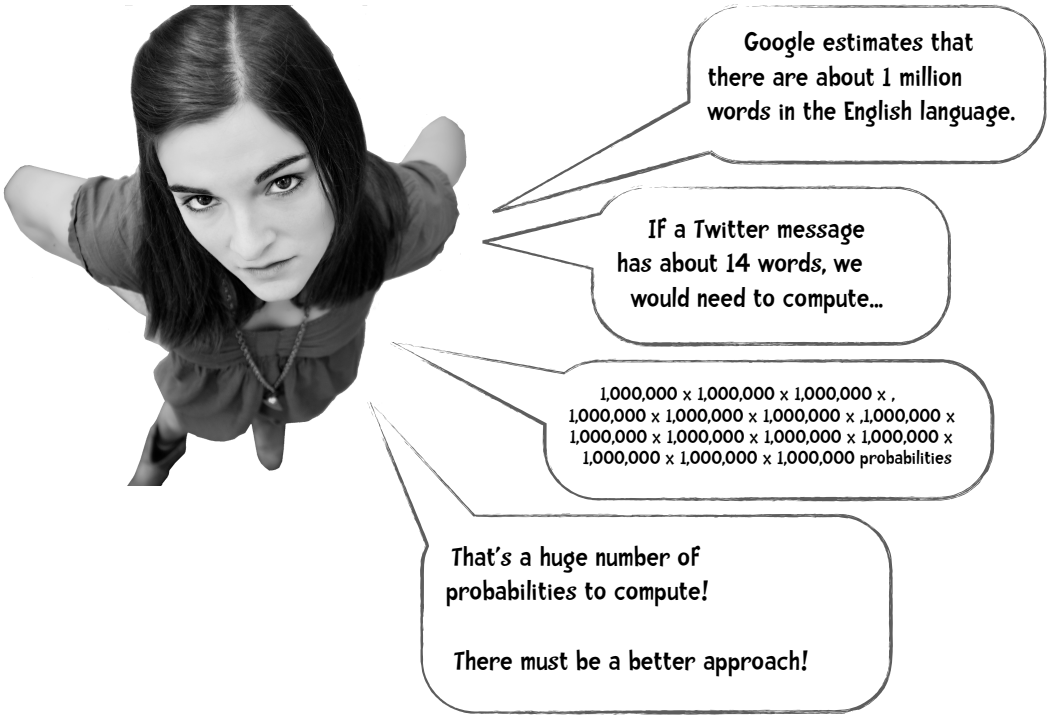
Learning from unlabeled text is called unsupervised learning. One example of unsupervised learning is clustering which we will cover in the next chapter.

There is also semi-supervised learning where the system learns from both labeled and unlabeled data. Often the system is bootstrapped using labeled data and then in subsequent learning makes use of unlabeled data.

Okay, back to

$$h_{MAP} = \arg\max_{h \in H} P(D \mid h)P(h)$$

Now let's examine the $P(D|h)$ part of the formula—the probability of seeing some evidence, some data D given the hypothesis h. The data D we are going to use is the words in the text. One approach would be to start with the first sentence of a document, for example, *Puts the Thrill back in Thriller*. And compute things like the probability that a 'like' document starts with the word *Puts*; what's the probability of a 'like' document having a second word of *the*; and the probability of the third word of a like document being *Thrill* and so on. And then compute the probability of a dislike document starting with the word *Puts*, the probability of the second word of a dislike document being *the* and so on.



Hmm. yeah. That is a huge number of probabilities which makes this approach unworkable. And, fortunately, there is a better approach. We are going to simplify things a bit by treating the documents as bags of unordered words. Instead of asking things like What's the probability that the third word is *thrill* given it is a 'like' document we will ask What's the probability that the word *thrill* occurs in a 'like' document. Here is how we are going to compute those probabilities.

Training Phase

First, we are going to determine the vocabulary—the unique words—of all the documents (both like and dislike documents). So, for example, even though *the* may occur thousands of times in our training corpus it only occurs once in our vocabulary. Let

$$|Vocabulary|$$

denote the number of words in the vocabulary. Next, for each word w_k in the vocabulary we are going to compute the probability of that word occurring given each hypothesis: $P(w_k \,|h_i)$.

We are going to compute this as follows. For each hypothesis (in this case 'like' and dislike')

1. combine the documents tagged with that hypothesis into one text file.
2. count how many word occurrences there are in the file. This time, if there are 500 occurrences of *the* we are going to count it 500 times. Let's call this n.
3. For each word in the vocabulary w_k, count how many times that word occurred in the text. Call this n_k
4. For each word in the vocabulary w_k, compute

$$P(w_k \mid h_i) = \frac{n_k + 1}{n + |Vocabulary|}$$

Naïve Bayes Classification Phase

Once we have completed the training phase we can classify documents using the formula that was already presented:

$$h_{MAP} = \arg\max_{h \in H} P(D \mid h) P(h)$$

Let's say our training corpus consisted of 500 Twitter messages with positive reviews of movies and 500 negative. So

$$P(like) = 0.5 \qquad P(dislike) = 0.5$$

After training the probabilities are as follows:

word	P(word|like)	P(word|dislike)
am	0.007	0.009
by	0.012	0.012
good	0.002	0.0005
gravity	0.00001	0.00001
great	0.003	0.0007
hype	0.0007	0.002
I	0.01	0.01
over	0.005	0.0047
stunned	0.0009	0.002
the	0.047	0.0465

How should we classify:

I am stunned by the hype over gravity

We are going to compute

$$P(like) \times P(I \mid like) \times P(am \mid like) \times P(stunned \mid like) \times \ldots$$

and

$$P(dislike) \times P(I \mid dislike) \times P(am \mid dislike) \times P(stunned \mid dislike) \times \ldots$$

and chose the hypothesis associated with the highest probability.

word	P(word|like)	P(word|dislike)
	P(like) = 0.5	P(dislike) =0.05
I	0.01	0.01
am	0.007	0.009
stunned	0.0009	0.002
by	0.012	0.012
the	0.047	0.0465
hype	0.0007	0.002
over	0.005	0.0047
gravity	0.00001	0.00001
∏	6.22E-22	4.72E-21

So the probabilities are

```
like      0.00000000000000000000622

dislike   0.00000000000000000004720
```

The probability of dislike is larger than that for like so we classify the tweet as a dislike.

Just a reminder:

That e notation means how many places to move the decimal point. If the number is positive we move the decimal to the right, negative means move it to the left. So

1.23e-1 = 0.123
1.23e-2 = 0.0123
1.23e-3 = 0.00123

and so on

Here's an illustration of the problem. Let's say we have a 100 word document and the average probability of each word is 0.001 (words like *tell*, *reported*, *average*, *morning*, and *am* have a probability of around 0.001). If I multiply those probabilities in Python we get zero:

```
>>> 0.0001**100
0.0
```

However, if we add the log of the probabilities we do get a non-zero value:

```
>>> import math
>>> p = 0
>>> for i in range(100):
      p += math.log(0.0001)

>>> p
-921.034037197617
```

in case you forgot ... $b^n = x$

The logorithm (or log) of a number (the x above) is the exponent (the n above) that you need to raise a base (b) to equal that number. For example, suppose the base is 10,

$\log_{10}(1000) = 3$ since 1000 equals 10^3

The base of the Python log function is the mathematical constant e. We don't really need to know about e. What is of interest to us is:

1. logs compress the scale of a number (with logs we can represent smaller numbers in Python)
   for ex.,
   .0000001 x .000005 = .000000000005
   the logs of those numbers are:
   -16.11809 + -9.90348 = -26.02157

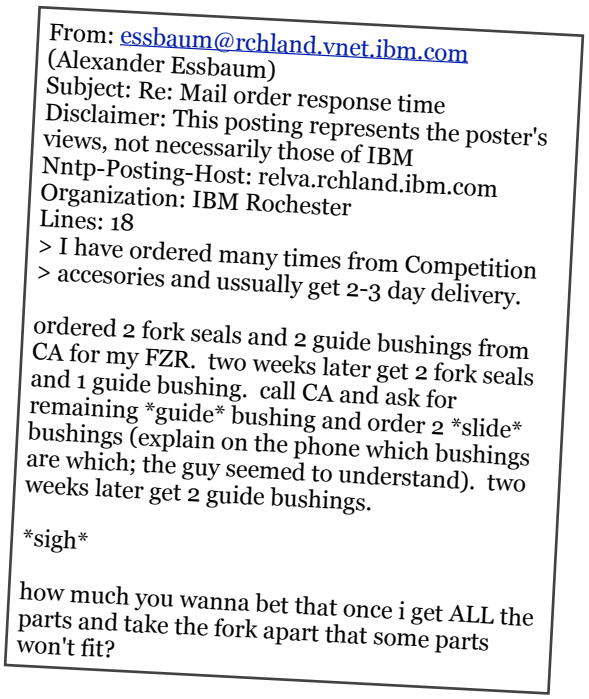
2. instead of multiplying the probabilities we are going to add the logs of the probabilities (as shown above).

Newsgroup Corpus

We will first investigate how this algorithm works by using a standard reference corpus of usenet newsgroup posts. The data consists of posts from 20 different newsgroups:

comp.graphics	misc.forsale	soc.religion.christian	alt.atheism
comp.os.ms-windows-misc	rec.autos	talk.politics.guns	sci.space
comp.sys.ibm.pc.hardware	rec.motorcycles	talk.politics.mideast	sci.crypt
comp.sys.mac.hardware	rec.sport.baseball	talk.politics.misc	sci.electronics
comp.windows.x	rec.sport.hockey	talk.religion.misc	sci.med

We would like to build a classifier that can correctly determine what group the post came from. For example, we would like to classify this post


From: essbaum@rchland.vnet.ibm.com
(Alexander Essbaum)
Subject: Re: Mail order response time
Disclaimer: This posting represents the poster's views, not necessarily those of IBM
Nntp-Posting-Host: relva.rchland.ibm.com
Organization: IBM Rochester
Lines: 18
> I have ordered many times from Competition
> accesories and ussually get 2-3 day delivery.

ordered 2 fork seals and 2 guide bushings from CA for my FZR. two weeks later get 2 fork seals and 1 guide bushing. call CA and ask for remaining *guide* bushing and order 2 *slide* bushings (explain on the phone which bushings are which; the guy seemed to understand). two weeks later get 2 guide bushings.

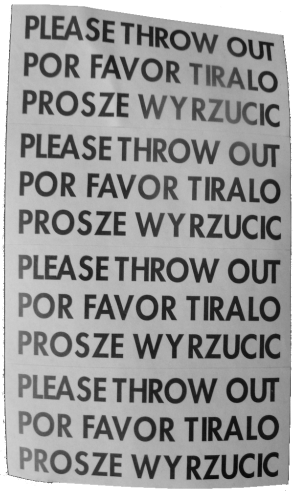
sigh

how much you wanna bet that once i get ALL the parts and take the fork apart that some parts won't fit?

as being from rec.motorcycles

Notice the misspellings (*accesories* and *ussually*). This might be challenging for a classifier!

The data is available at http://qwone.com/~jason/20Newsgroups/ (we are using the 20news=bydate dataset) . It is also available on the website for the book, http://guidetodatamining.com. The data consists of 18,846 documents and is already sorted into training (60% of the data) and test sets. The training and test data are in separate directories. Within each directory are subdirectories representing each newsgroup. Within those are the separate documents representing posts to that newsgroup.



Throwing things out!

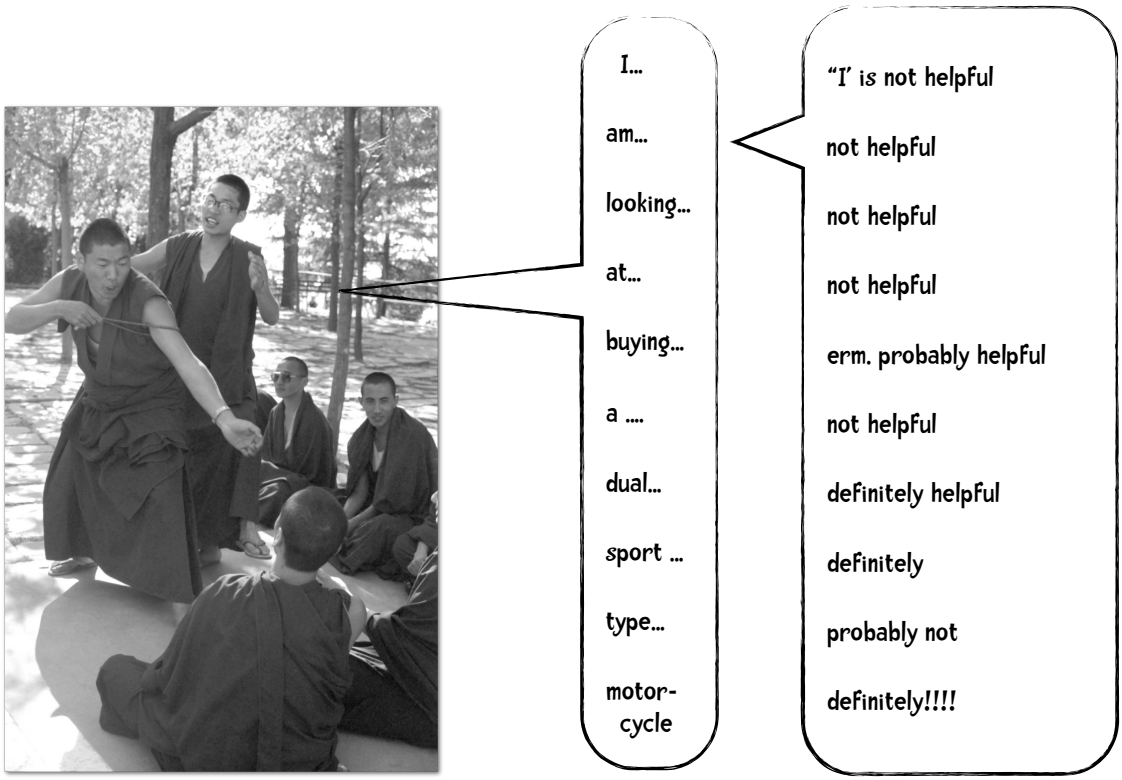
Before we start coding, let's think about this task in more detail.

Ladies and Gentlemen. On the main stage ... Just based on the words in the text, we are going to attempt to tell which newsgroup the post is from

For example, we would like to build a system that would classify the following post as being from rec.motorcycle:

> I am looking at buying a Dual Sport type motorcycle. This is my first cycle as well. I am interested in any experiences people have with the following motorcycles, good or bad.
>
> Honda XR250L
> Suzuki DR350S
> Suzuki DR250ES
> Yamaha XT350
>
> Most XXX vs. YYY articles I have seen in magazines pit the Honda XR650L against another cycle, and the 650 always comes out shining. Is it safe to assume that the 250 would be of equal quality ?

Let's consider which words might be helpful in the classification task:



If we throw out the 200 most frequent words in English our document looks like this:

I am looking at buying a Dual Sport type motorcycle. This is my first cycle as well. I am interested in any experiences people have with the following motorcycles, good or bad.

Honda XR250L
Suzuki DR350S
Suzuki DR250ES
Yamaha XT350

Most XXX vs. YYY articles I have seen in magazines pit the Honda XR650L against another cycle, and the 650 always comes out shining. Is it safe to assume that the 250 would be of equal quality ?

Removing these words cuts down the size of our text by about half. Plus, it doesn't look like removing these words will have any impact on our ability to categorize texts. Indeed data miners have called such words *words without any content*, and *fluff words*. H.P. Luhn, in his seminal paper 'The automatic creation of literature abstracts' says of these words that they are “too common to have the type of significance being sought and would constitute 'noise' in the system.” That noise argument is interesting as it implies that removing these words will improve performance. These words that we remove are called 'stop words'. We have a list of such words, the 'stop word list', and remove these words from the text in a preprocessing step. We remove these words because 1) it cuts down on the amount of processing we need to do and 2) it does not negatively impact the performance of our system—as the noise argument suggests removing them might improve performance.

Common Words vs. Stop Words
While it is true that common words like ‘the’ and ‘a’ may not help us in our classification task, other common words such as ‘work’, ‘write’, and ‘school’ may help depending on our classification task. When we create a stop word list, we often omit common words that may be helpful. You can explore these differences by comparing stop word lists and frequent word lists found on the web.

The counter argument: the hazards of stop word removal

You whippersnapper. You shouldn’t be throwing away those common words!

While removing stop words may be useful in some situations, you should not just automatically remove them without thinking. For example, it turns out just using the most frequent words and throwing out the rest (the reverse technique of the above) provides

sufficient information to identify where Arabic documents were written. (If you are curious about this check out the paper *Linguistic Dumpster Diving: Geographical Classification of Arabic Text* I co-wrote with some of my colleagues at New Mexico State University. It is available on my website http://zacharski.org). In looking at online chats, sexual predators use words like *I*, *me*, and *you*, much more frequently than non-predators. If your task is to identify sexual predators, removing frequent words would actually hurt your performance.

Coding it – Python Style

Let us first consider coding the training part of the Naïve Bayes Classifier. Recall that the training data is organized as follows:

```
20news-bydate-train
      alt.atheism
            text file 1 for alt.atheism
            text file 2
            …
            text file n
      comp.graphics
            text file 1 for comp.graphics
            ...
```

So I have a directory (in this example called '`20news-bydate-train`'). Underneath this directory are subdirectories representing different classification categories (in this case `alt.atheism, comp.graphics`, etc). The names of these subdirectories match the category names. The test directory is organized in a similar way. So, in matching this structure, the Python code for training will need to know the training directory (for example, `/Users/raz/Downloads/20news-bydate/20news-bydate-train/`). The outline for the training code is as follows.

class BayesText

1. the init method:
    a. read in the words from the stoplist
    b. read the training directory to get the names of the subdirectories (in addition to being the names of the subdirectories, these are the names of the categories).
    c. For each of those subdirectories, call a method "train" that will count the occurrences of words in all the files of that subdirectory.
    d. compute the probabilities using

$$P(w_k \mid h_i) = \frac{n_k + 1}{n + |Vocabulary|}$$

Yet another reminder that all the code is available at
guidetodatamining.com

```
from __future__ import print_function
import os, codecs, math

class BayesText:

    def __init__(self, trainingdir, stopwordlist):
        """This class implements a naive Bayes approach to text
        classification
        trainingdir is the training data. Each subdirectory of
        trainingdir is titled with the name of the classification
        category -- those subdirectories in turn contain the text
        files for that category.
        The stopwordlist is a list of words (one per line) will be
        removed before any counting takes place.
        """
        self.vocabulary = {}
        self.prob = {}
        self.totals = {}
        self.stopwords = {}
        f = open(stopwordlist)
        for line in f:
            self.stopwords[line.strip()] = 1
        f.close()
        categories = os.listdir(trainingdir)
        #filter out files that are not directories
        self.categories = [filename for filename in categories
                           if os.path.isdir(trainingdir + filename)]
        print("Counting ...")
        for category in self.categories:
            print('    ' + category)
            (self.prob[category],
             self.totals[category]) = self.train(trainingdir, category)
        # I am going to eliminate any word in the vocabulary
        # that doesn't occur at least 3 times
        toDelete = []
        for word in self.vocabulary:
            if self.vocabulary[word] < 3:
                # mark word for deletion
                # can't delete now because you can't delete
                # from a list you are currently iterating over
                toDelete.append(word)
```

```
        # now delete
        for word in toDelete:
            del self.vocabulary[word]
        # now compute probabilities
        vocabLength = len(self.vocabulary)
        print("Computing probabilities:")
        for category in self.categories:
            print('    ' + category)
            denominator = self.totals[category] + vocabLength
            for word in self.vocabulary:
                if word in self.prob[category]:
                    count = self.prob[category][word]
                else:
                    count = 1
                self.prob[category][word] = (float(count + 1)
                                             / denominator)
        print ("DONE TRAINING\n\n")


    def train(self, trainingdir, category):
        """counts word occurrences for a particular category"""
        currentdir = trainingdir + category
        files = os.listdir(currentdir)
        counts = {}
        total = 0
        for file in files:
            #print(currentdir + '/' + file)
            f = codecs.open(currentdir + '/' + file, 'r', 'iso8859-1')
            for line in f:
                tokens = line.split()
                for token in tokens:
                    # get rid of punctuation and lowercase token
                    token = token.strip('\'".,?:-')
                    token = token.lower()
                    if token != '' and not token in self.stopwords:
                        self.vocabulary.setdefault(token, 0)
                        self.vocabulary[token] += 1
                        counts.setdefault(token, 0)
                        counts[token] += 1
                        total += 1
            f.close()
        return(counts, total)
```

The results of the training phase are stored in a dictionary (hash table) called `prob`. The keys of the dictionary are the different classifications (comp.graphics, rec.motorcycles, soc.religion.christian, etc); the values are dictionaries. The keys of these subdictionaries are the words and the values are the probabilities of those words. Here is an example:

```
bT = BayesText(trainingDir, stoplistfile)
>>>bT.prob["rec.motorcycles"]["god"]
0.00013035445075435553
>>>bT.prob["soc.religion.christian"]["god"]
0.004258192391884386
>>>bT.prob["rec.motorcycles"]["the"]
0.028422937849264914
>>>bT.prob["soc.religion.christian"]["the"]
0.039953678998362795
```

So, for example, the probability of the word 'god' occurring in a text in the rec.motorcycles newsgroup is 0.00013 (or one occurrence of *god* in every 10,000 words). The probability of the word 'god' occurring in a text in soc.religion.christian is .00424 (one occurrence in every 250 words).

Training also results in a list called categories, which, as you might predict, is simply a list of all the categories:

```
['alt.atheism', 'comp.graphics', 'comp.os.ms-windows.misc',
'comp.sys.ibm.pc.hardware', ...]
```

So that is the training phase. Let us now turn to classifying a document.

code it

Can you code a method called classify that will predict the classification of a document? For example:

```
>>> bT.classify("20news-bydate-test/rec.motorcycles/104673")
'rec.motorcycles'
>>> bT.classify("20news-bydate-test/sci.med/59246")
'sci.med'
>>> bT.classify("20news-bydate-test/soc.religion.christian/21424")
'soc.religion.christian'
```

As you can see, the classify method takes a filename as an argument and returns a string denoting the classification.

A Python file you can use as a template, bayesText-ClassifyTemplate.py, is available on our website.

```
class BayesText:

  def __init__(self, trainingdir, stopwordlist):
    self.vocabulary = {}
    self.prob = {}
    self.totals = {}
    self.stopwords = {}
    f = open(stopwordlist)
    for line in f:
        self.stopwords[line.strip()] = 1
    f.close()
    categories = os.listdir(trainingdir)
    #filter out files that are not directories
    self.categories = [filename for filename in categories
                       if os.path.isdir(trainingdir +
filename)]
    print("Counting ...")
    for category in self.categories:
        print('    ' + category)
        (self.prob[category],
         self.totals[category]) = self.train(trainingdir,
category)
    # I am going to eliminate any word in the vocabulary
```

code it - one possible solution

```
def classify(self, filename):
  results = {}
  for category in self.categories:
     results[category] = 0
  f = codecs.open(filename, 'r', 'iso8859-1')
  for line in f:
     tokens = line.split()
     for token in tokens:
       token = token.strip('\'".,?:-').lower()
       if token in self.vocabulary:
         for category in self.categories:
            if self.prob[category][token] == 0:
               print("%s %s" % (category, token))
            results[category] += math.log(
                        self.prob[category][token])
  f.close()
  results = list(results.items())
  results.sort(key=lambda tuple: tuple[1], reverse = True)
  # for debugging I can change this to give me the entire list
  return results[0][0]
```

Finally, let's have a method that classifies every document in the test directory and prints out the percent accuracy of this method.

```
    def testCategory(self, directory, category):
        files = os.listdir(directory)
        total = 0
        correct = 0
        for file in files:
            total += 1
            result = self.classify(directory + file)
            if result == category:
                correct += 1
        return (correct, total)

    def test(self, testdir):
        """Test all files in the test directory--that directory is
        organized into subdirectories--each subdir is a classification
        category"""
        categories = os.listdir(testdir)
        #filter out files that are not directories
        categories = [filename for filename in categories if
                      os.path.isdir(testdir + filename)]
        correct = 0
        total = 0
        for category in categories:
            (catCorrect, catTotal) = self.testCategory(
                testdir + correct += catCorrect
            total += catTotal
        print("Accuracy is  %f%%  (%i test instances)" %
              ((float(correct) / total) * 100, total))
```

When I run this code using an empty stoplist file I get:

```
DONE TRAINING
Running Test ...
.....................
Accuracy is  77.774827%  (7532 test instances)
```

code it

Can you run the classifier with a few stop word lists? Does performance improve? Which is most accurate? (You will need to search the web to find these lists)

stop list size	accuracy
0	77.774827
list 1	
list 2	

code it - some results

I found a 25 word stop word list at: http://nlp.stanford.edu/IR-book/html/htmledition/dropping-common-terms-stop-words-1.html
And a 174 word one at http://www.ranks.nl/resources/stopwords.html

(these word lists are available on our website)

Here are the results:

stop list size	accuracy
0	77.774827%
25 word list	78.757302%
174 word list	79.938927%

So in this case, it looks like having a 174 word stop word list improved performance about 2% over having no stop word list? Does this match your results?

Naïve Bayes and Sentiment Analysis

The goal of sentiment analysis is to determine the writer's attitude (or opinion).

One common type of sentiment analysis is to determine the **polarity** of a review or comment (positive or negative) and we can use a Naïve Bayes Classifier for this task. We can try this out by using the polarity movie review dataset first presented in Pang and Lee 2004[1]. Their dataset consists of 1,000 positive and 1,000 negative reviews. Here are some examples:

the second serial-killer thriller of the month is just awful . oh , it starts deceptively okay , with a handful of intriguing characters and some solid location work . ...

when i first heard that romeo & juliet had been " updated " i shuddered . i thought that yet another of shakespeare's classics had been destroyed .
fortunately , i was wrong . baz luhrman has directed an " in your face " , and visually

[1] Pang, Bo and Lillian Lee. 2004. A sentimental education: Sentiment analysis using subjectivity summarization based on minimum cuts. Proceedings of ACL.

You can download the original dataset from http://www.cs.cornell.edu/People/pabo/movie-review-data/. I have organized the data into 10 buckets (folds) with the following directory structure:

```
review_polarity_buckets
    txt_sentoken
        neg
            0
                files in fold 0
            1
                files in fold 1
            ...
            9
                files in fold 9
        pos
            0
                files in fold 0
            ...
```

This re-organized dataset is available on our website.

code it

Can you modify the Naive Bayes Classifier code to do 10-fold cross validation of the classifier on this data set. The output should look something like:

```
       Classified as:
          neg   pos
        +-----+-----+
 neg    |   1 |   2 |
 pos    |   3 |   4 |
        +-----+-----+
12.345 percent correct
total of 2000 instances
```

Also compute the kappa coefficient.

Obvious Disclaimer

You won't become proficient in data mining by reading this book anymore than reading a book about piano playing will make you proficient at piano playing. You need to practice!

Woman practicing Brahms

Practice makes the heart grow fonder!

Woman practicing Naïve Bayes

code it – my results

Here are the results I got:

```
       Classified as:
          neg   pos
         +-----+-----+
 neg     | 845 | 155 |
 pos     | 222 | 778 |
         +-----+-----+

81.150 percent correct
total of 2000 instances
```

Also compute the kappa coefficient.

$$\kappa = \frac{P(c) - P(r)}{1 - P(r)} = \frac{.8115 - 0.5}{1 - 0.5} = \frac{.3115}{.5} = 0.623$$

So we have good performance of the algorithm on this data.

My code is on the following page!

Yet another reminder:
The code is available for download on the book's website http://guidetodatamining.com/

```
from __future__ import print_function
import os, codecs, math

class BayesText:

    def __init__(self, trainingdir, stopwordlist, ignoreBucket):
        """This class implements a naive Bayes approach to text
        classification
        trainingdir is the training data. Each subdirectory of
        trainingdir is titled with the name of the classification
        category -- those subdirectories in turn contain the text
        files for that category.
        The stopwordlist is a list of words (one per line) will be
        removed before any counting takes place.
        """
        self.vocabulary = {}
        self.prob = {}
        self.totals = {}
        self.stopwords = {}
        f = open(stopwordlist)
        for line in f:
            self.stopwords[line.strip()] = 1
        f.close()
        categories = os.listdir(trainingdir)
        #filter out files that are not directories
        self.categories = [filename for filename in categories
                           if os.path.isdir(trainingdir + filename)]
        print("Counting ...")
        for category in self.categories:
            #print('    ' + category)
            (self.prob[category],
             self.totals[category]) = self.train(trainingdir, category,
                                                 ignoreBucket)
        # I am going to eliminate any word in the vocabulary
        # that doesn't occur at least 3 times
        toDelete = []
```

```
        for word in self.vocabulary:
            if self.vocabulary[word] < 3:
                # mark word for deletion
                # can't delete now because you can't delete
                # from a list you are currently iterating over
                toDelete.append(word)
        # now delete
        for word in toDelete:
            del self.vocabulary[word]
        # now compute probabilities
        vocabLength = len(self.vocabulary)
        #print("Computing probabilities:")
        for category in self.categories:
            #print('    ' + category)
            denominator = self.totals[category] + vocabLength
            for word in self.vocabulary:
                if word in self.prob[category]:
                    count = self.prob[category][word]
                else:
                    count = 1
                self.prob[category][word] = (float(count + 1)
                                             / denominator)
        #print ("DONE TRAINING\n\n")


    def train(self, trainingdir, category, bucketNumberToIgnore):
        """counts word occurrences for a particular category"""
        ignore = "%i" % bucketNumberToIgnore
        currentdir = trainingdir + category
        directories = os.listdir(currentdir)
        counts = {}
        total = 0
        for directory in directories:
            if directory != ignore:
                currentBucket = trainingdir + category + "/" +  \
                                directory
                files = os.listdir(currentBucket)
                #print("   " + currentBucket)
                for file in files:
```

```
                f = codecs.open(currentBucket + '/' + file, 'r',
                                'iso8859-1')
                for line in f:
                    tokens = line.split()
                    for token in tokens:
                        # get rid of punctuation
                        # and lowercase token
                        token = token.strip('\'".,?:-')
                        token = token.lower()
                        if token != '' and not token in \
                           self.stopwords:
                            self.vocabulary.setdefault(token, 0)
                            self.vocabulary[token] += 1
                            counts.setdefault(token, 0)
                            counts[token] += 1
                            total += 1
                f.close()
        return(counts, total)


    def classify(self, filename):
        results = {}
        for category in self.categories:
            results[category] = 0
        f = codecs.open(filename, 'r', 'iso8859-1')
        for line in f:
            tokens = line.split()
            for token in tokens:
                #print(token)
                token = token.strip('\'".,?:-').lower()
                if token in self.vocabulary:
                    for category in self.categories:
                        if self.prob[category][token] == 0:
                            print("%s %s" % (category, token))
                        results[category] += math.log(
                            self.prob[category][token])
        f.close()
        results = list(results.items())
        results.sort(key=lambda tuple: tuple[1], reverse = True)
```

```
        # for debugging I can change this to give me the entire list
        return results[0][0]

    def testCategory(self, direc, category, bucketNumber):
        results = {}
        directory = direc + ("%i/" % bucketNumber)
        #print("Testing " + directory)
        files = os.listdir(directory)
        total = 0
        correct = 0
        for file in files:
            total += 1
            result = self.classify(directory + file)
            results.setdefault(result, 0)
            results[result] += 1
            #if result == category:
            #               correct += 1
        return results

    def test(self, testdir, bucketNumber):
        """Test all files in the test directory--that directory is
        organized into subdirectories--each subdir is a classification
        category"""
        results = {}
        categories = os.listdir(testdir)
        #filter out files that are not directories
        categories = [filename for filename in categories if
                      os.path.isdir(testdir + filename)]
        correct = 0
        total = 0
        for category in categories:
            #print(".", end="")
            results[category] = self.testCategory(
                testdir + category + '/', category, bucketNumber)
        return results

def tenfold(dataPrefix, stoplist):
    results = {}
    for i in range(0,10):
```

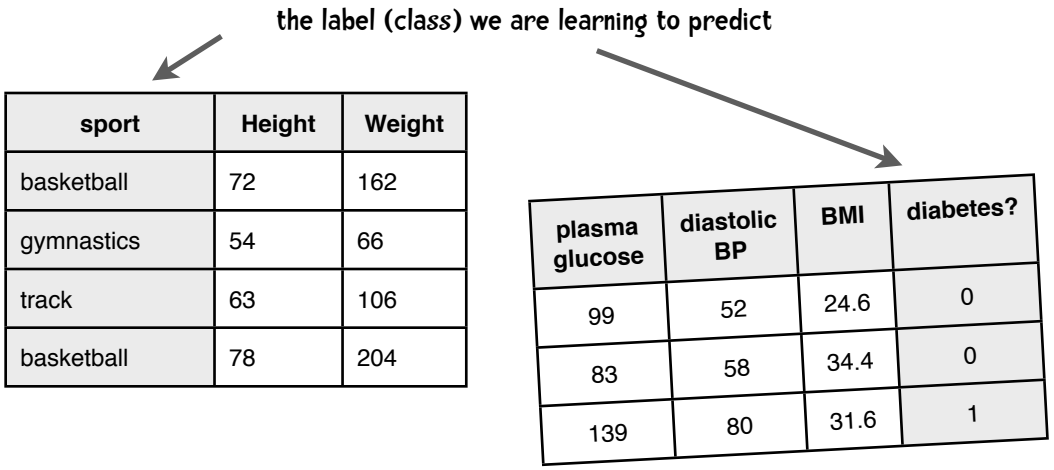
```
        bT = BayesText(dataPrefix, stoplist, i)
        r = bT.test(theDir, i)
        for (key, value) in r.items():
            results.setdefault(key, {})
            for (ckey, cvalue) in value.items():
                results[key].setdefault(ckey, 0)
                results[key][ckey] += cvalue
                categories = list(results.keys())
    categories.sort()
    print(   "\n       Classified as: ")
    header =    "          "
    subheader = "        +"
    for category in categories:
        header += "% 2s   " % category
        subheader += "-----+"
    print (header)
    print (subheader)
    total = 0.0
    correct = 0.0
    for category in categories:
        row = " %s    |" % category
        for c2 in categories:
            if c2 in results[category]:
                count = results[category][c2]
            else:
                count = 0
            row += " %3i |" % count
            total += count
            if c2 == category:
                correct += count
        print(row)
    print(subheader)
    print("\n%5.3f percent correct" %((correct * 100) / total))
    print("total of %i instances" % total)

# change these to match your directory structure
theDir = "/Users/raz/Downloads/review_polarity_buckets/txt_sentoken/"
stoplistfile = "/Users/raz/Downloads/20news-bydate/stopwords25.txt"
tenfold(theDir, stoplistfile)
```

Chapter 8: Clustering

Discovering Groups

In previous chapters we have been developing classification systems. In these systems we train a classifier on a set of labeled examples.



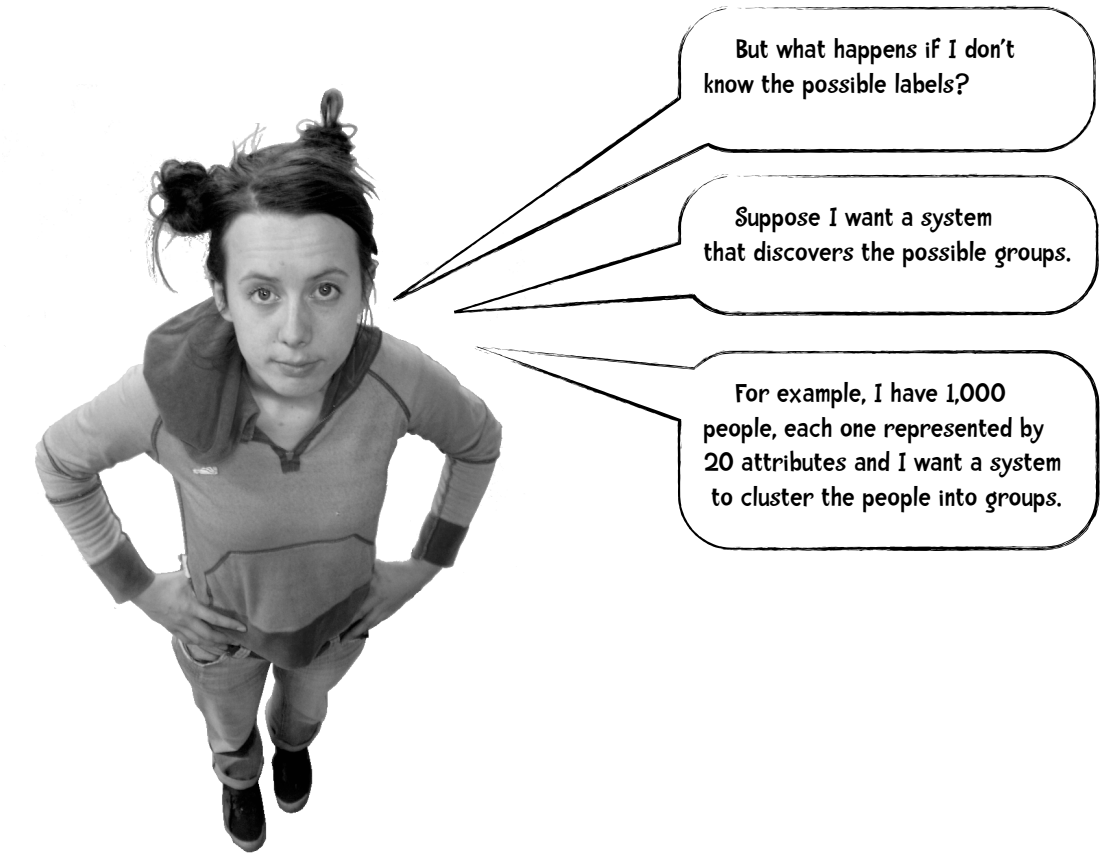
sport	Height	Weight
basketball	72	162
gymnastics	54	66
track	63	106
basketball	78	204

plasma glucose	diastolic BP	BMI	diabetes?
99	52	24.6	0
83	58	34.4	0
139	80	31.6	1

After we train the classifier, we can use it to label new examples.

> *This person looks like a basketball player. That one a gymnast. That person is unlikely to get diabetes in 3 years*.

and so on. In other words, the classifier selects a label from a set of labels it acquired during the training phase—it knows the possible labels.



This task is called clustering. The system divides a set of instances into clusters or groups based on some measure of similarity. There are two main types of clustering algorithms.

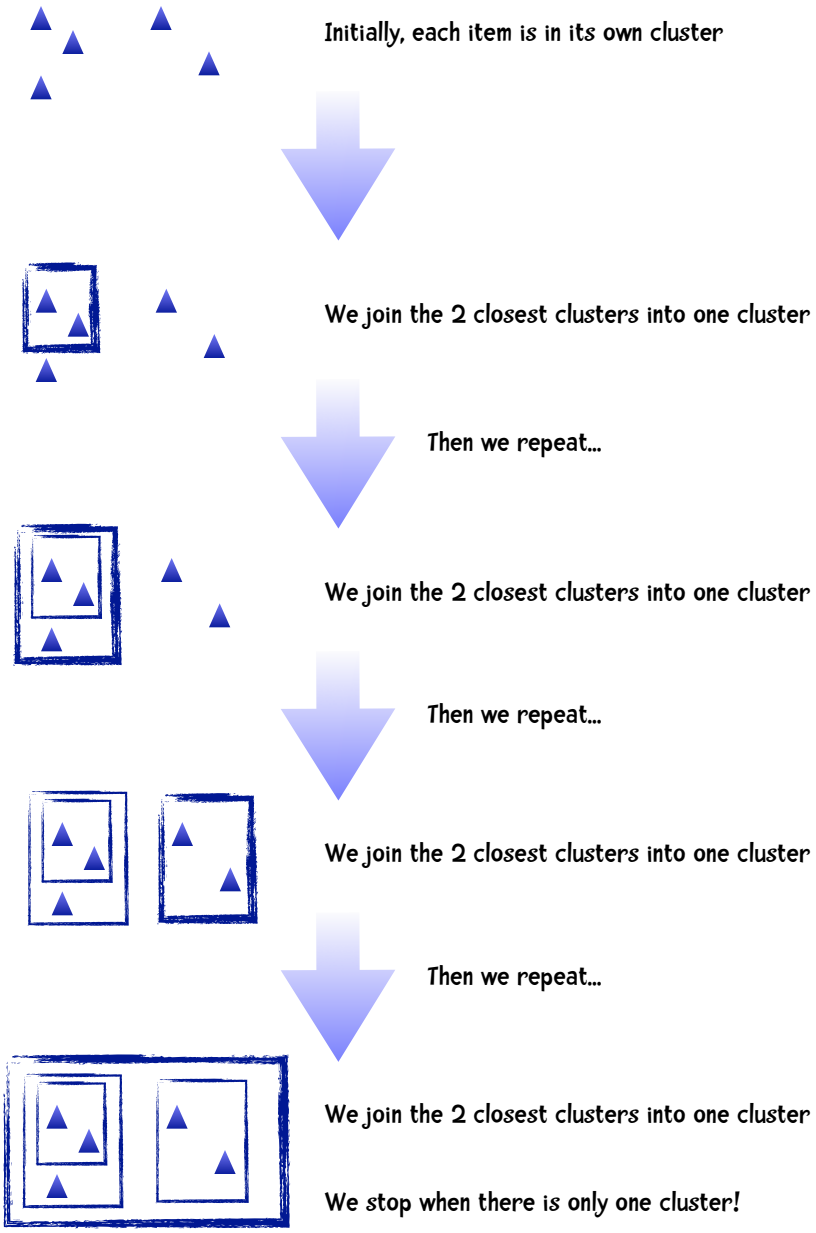
k-means clustering

For one type, we tell the algorithm how many clusters to make. *Please cluster these 1,000 people into 5 groups. Please classify these web pages into 15 groups.* These methods go by the name of k-means clustering algorithms and we will discuss those a bit later in the chapter.

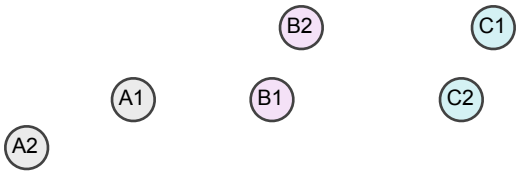
hierarchical clustering

For the other approach we don't specify how many clusters to make. Instead the algorithm starts with each instance in its own cluster. At each iteration of the algorithm it combines the two most similar clusters into one. It repeatedly does this until there is only one cluster. This

is called hierarchical clustering and its name makes sense. The running of the algorithm results in one cluster, which consists of two sub-clusters. Each of those two sub-clusters in turn, consist of 2 sub-sub clusters and so on.



Again, at each iteration of the algorithm we join the two closest clusters. To determine the 'closest clusters' we use a distance formula. But we have some choices in how we compute the distance between two clusters, which leads to different clustering methods. Consider the three clusters (A, B, and C) illustrated below each containing two members. Which pair of clusters should we join? Cluster A with B, or cluster C with B?



Single-linkage clustering

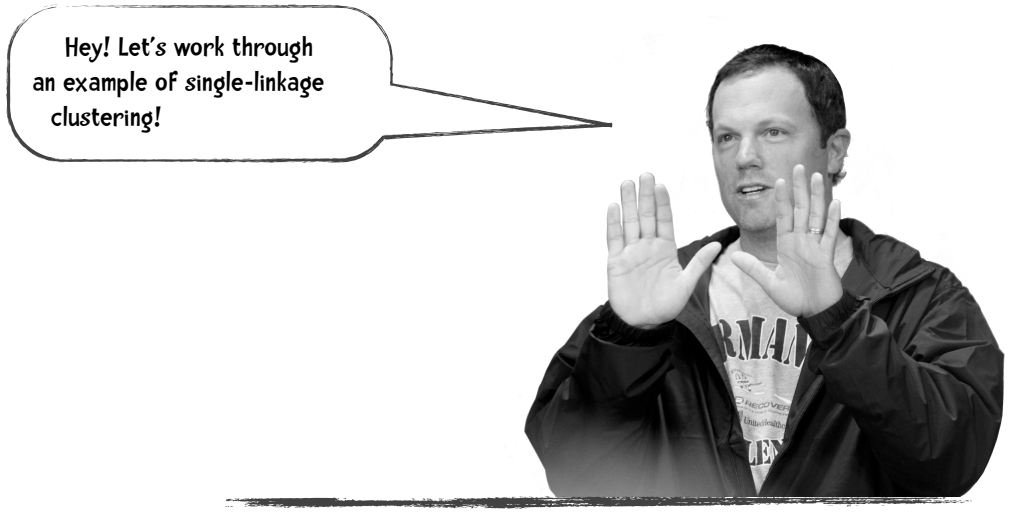
In single-linkage clustering we define the distance between two clusters as the **shortest** distance between any member of one cluster to any member of the other. With this definition, the distance between Cluster A and Cluster B is the distance between A1 and B1, since that is shorter than the distances between A1 and B2, A2 and B1, and A2 and B2. With single-linkage clustering, Cluster A is closer to Cluster B than C is to B, so we would combine A and B into a new cluster.

Complete-linkage clustering

In complete-linkage clustering we define the distance between two clusters as the **greatest** distance between any member of one cluster to any member of the other. With this definition, the distance between Cluster A and Cluster B is the distance between A2 and B2. With complete-linkage clustering, Cluster C is closer to Cluster B than A is to B, so we would combine B and C into a new cluster.

Average-linkage clustering

In average-linkage clustering we define the distance between two clusters as the **average** distance between any member of one cluster to any member of the other. In the diagram above, it appears that the average distance between Clusters C and B would be less than the average between A and B and we would combine B and C into a new cluster.



Good idea! Let's practice by clustering dog breeds based on height and weight!

breed	height (inches)	weight (pounds)
Border Collie	20	45
Boston Terrier	16	20
Brittany Spaniel	18	35
Bullmastiff	27	120
Chihuahua	8	8
German Shepherd	25	78
Golden Retriever	23	70
Great Dane	32	160
Portuguese Water Dog	21	50
Standard Poodle	19	65
Yorkshire Terrier	6	7

Psst! I think we are forgetting something. Isn't there something we should do before computing distance?

Normalization!

Let's change those numbers to Modified Standard Scores

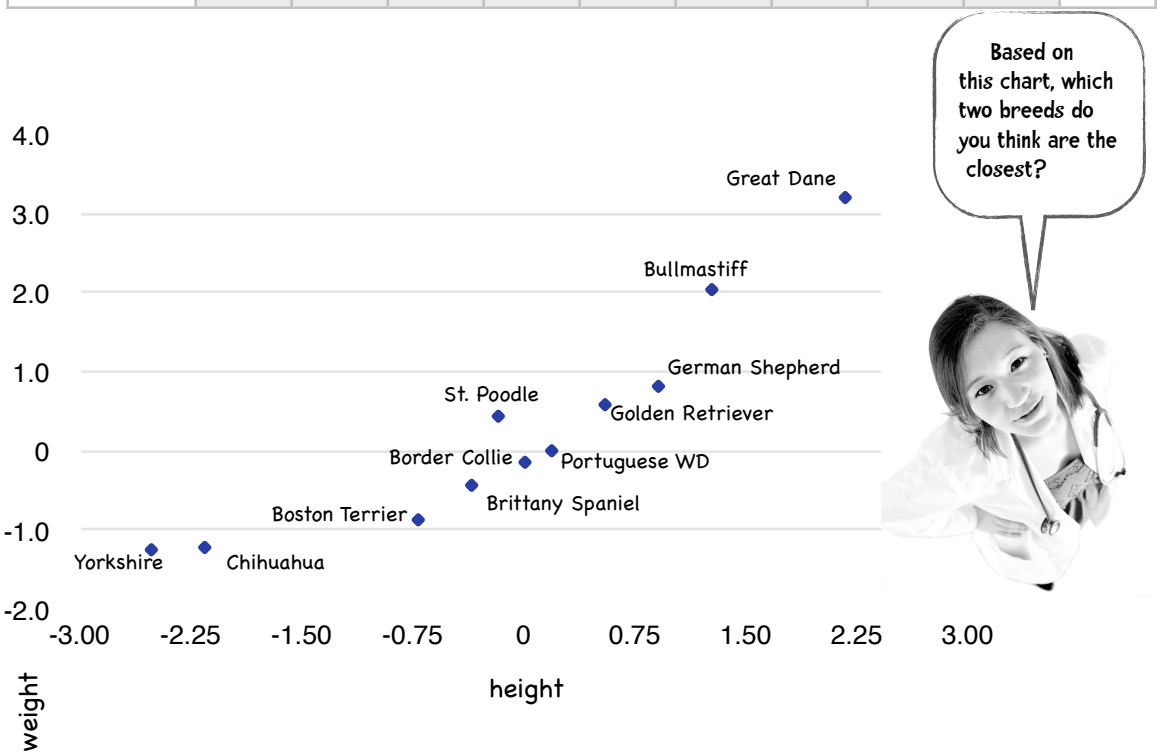
Modified Standard Scores

breed	height	weight
Border Collie	0	-0.1455
Boston Terrier	-0.7213	-0.873
Brittany Spaniel	-0.3607	-0.4365
Bullmastiff	1.2623	2.03704
Chihuahua	-2.1639	-1.2222
German Shepherd	0.9016	0.81481
Golden Retriever	0.541	0.58201
Great Dane	2.16393	3.20106
Portuguese Water Dog	0.1803	0
Standard Poodle	-0.1803	0.43651
Yorkshire Terrier	-2.525	-1.25132

Next we are going to compute the Euclidean distance between breeds!

Euclidean Distances (a few of the shortest distances are highlighted):

	BT	BS	B	C	GS	GR	GD	PWD	SP	YT
Border Collie	1.024	0.463	2.521	2.417	1.317	0.907	3.985	0.232	0.609	2.756
Boston Terrier		0.566	3.522	1.484	2.342	1.926	4.992	1.255	1.417	1.843
Brittany Spaniel			2.959	1.967	1.777	1.360	4.428	0.695	0.891	2.312
Bullmastiff				4.729	1.274	1.624	1.472	2.307	2.155	5.015
Chihuahua					3.681	3.251	6.188	2.644	2.586	0.362
German Shphrd						0.429	2.700	1.088	1.146	4.001
Golden Retriever							3.081	0.685	0.736	3,572
Great Dane								3.766	3.625	6.466
Portuguese WD									0.566	2.980
Standard Poodle										2.889

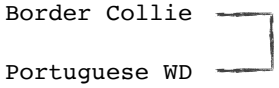


If you said Border Collie and Portuguese Water Dog, you would be correct!

The algorithm.

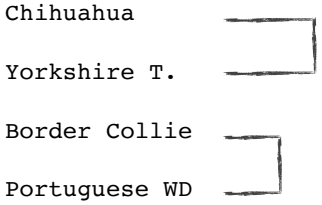
Step 1.

Initially, each breed is in its own cluster. We find the two closest clusters and combine them into one cluster. From the table on the preceding page we see that the closest clusters are the Border Collie and the Portuguese Water Dog (distance of 0.232) so we combine them.



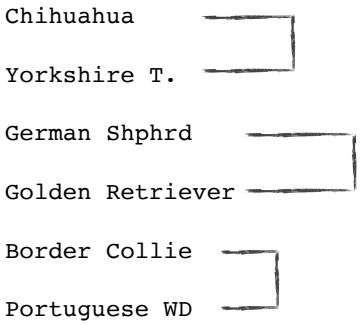
Step 2.

We find the two closest clusters and combine them into one cluster. From the table on the preceding page we see that these are the Chihuahua and the Yorkshire Terrier (distance of 0.362) so we combine them.



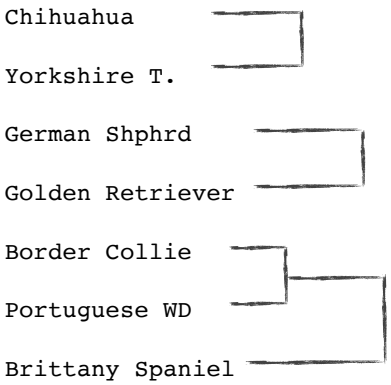
Step 3.

We repeat the process again. This time combining the German Shepherd and the Golden Retriever.



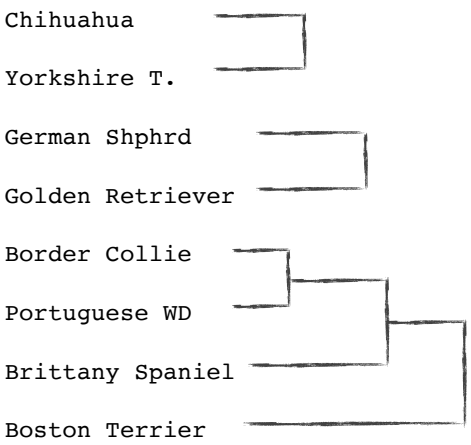
Step 4.

We repeat the process yet again. From the table we see that the next closest pair is the Border Collie and the Brittany Spaniel. The Border Collie is already in a cluster with the Portuguese Water Dog which we created in Step 1. So in this step we are going to combine that cluster with the Brittany Spaniel.



This type of diagram is called a dendrogram. It is basically a tree diagram that represents clusters.

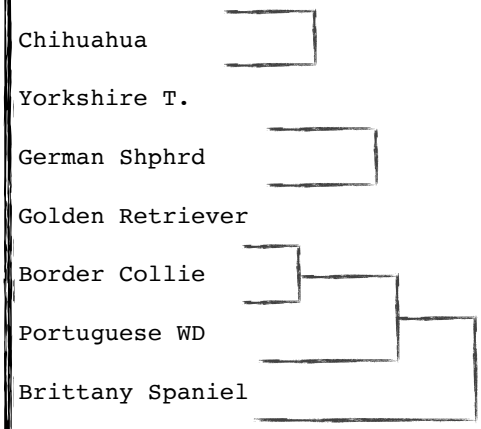
And we continue:



sharpen your pencil

Finish the clustering of the dog data!

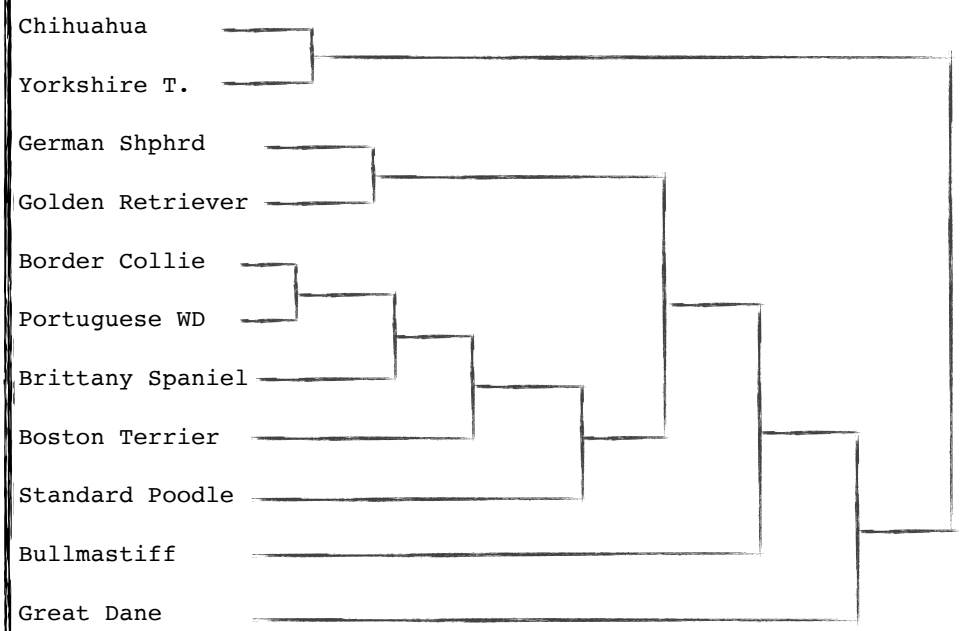
To help you in this task, there is a sorted list of dog breed distances on this chapter's webpage (https://raw.githubusercontent.com/zacharski/pg2dm-python/0684ec677a1a1baaecb47bc0f8f21ec121e83339/data/ch8/dogDistanceSorted.txt).



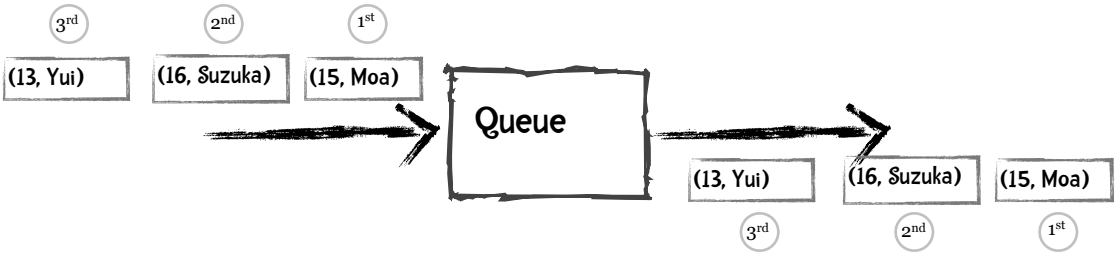
sharpen your pencil solution

Finish the clustering of the dog data!

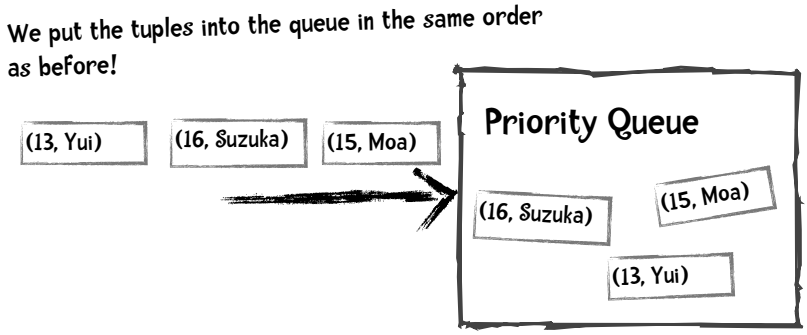
To help you in this task, there is a sorted list of dog breed distances on this chapter's webpage (http://guidetodatamining.com/guide/ch8/dogDistanceSorted.txt).

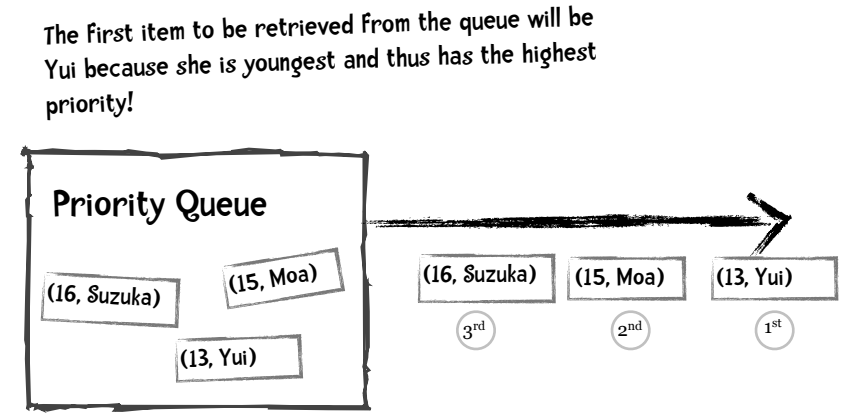


coding a hierarchical clustering algorithm



In a priority queue each item put into the queue has an associated priority. The order in which items are retrieved from the queue is based on this priority. Items with a higher priority are retrieved before items with a lower one. In our example data, suppose the younger a person is, the higher their priority.



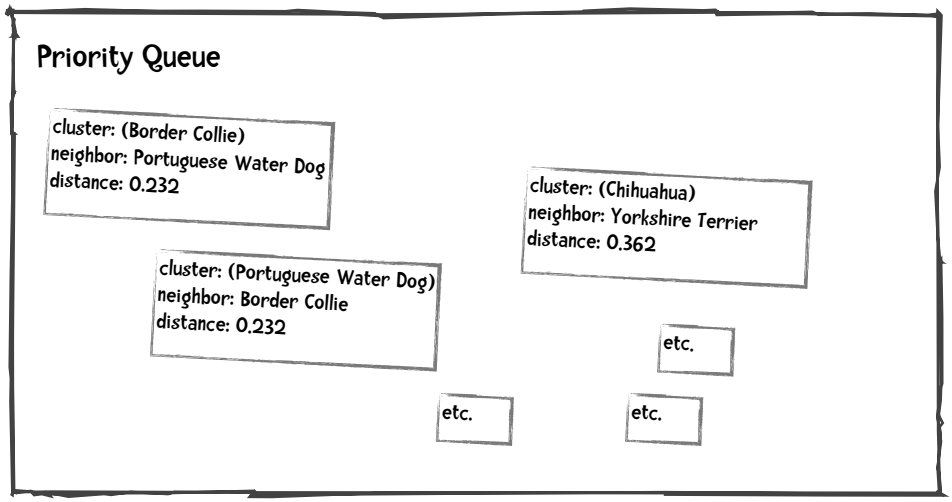


Let's see how this works in Python

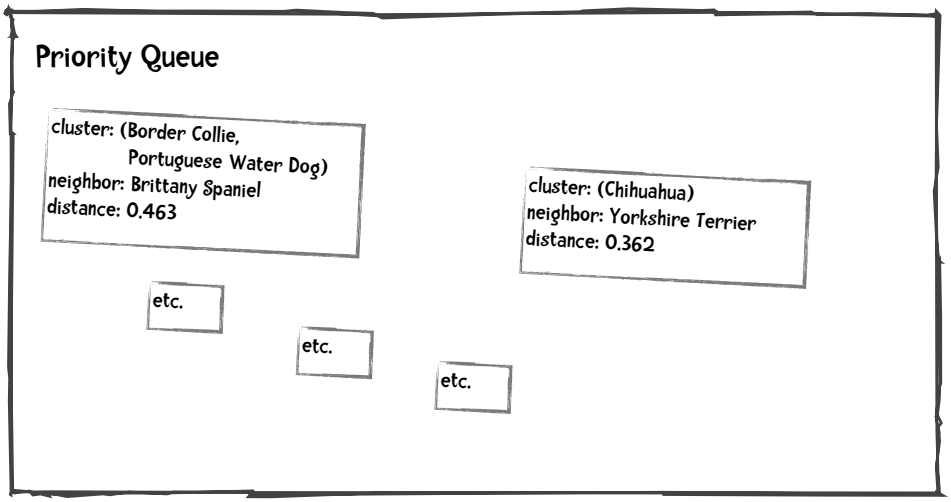
```
>>> from queue import PriorityQueue          # load the PriorityQueue library
>>> singersQueue = PriorityQueue()           # create a PriorityQueue called
                                             # singersQueue
>>> singersQueue.put((16, 'Suzuka Nakamoto')) # put a few items in the queue
>>> singersQueue.put((15, 'Moa Kikuchi'))
>>> singersQueue.put((14, 'Yui Mizuno'))
```

```
>>> singersQueue.put((17, 'Ayaka Sasaki'))
>>> singersQueue.get()                          # The first item retrieved
(14, 'Yui Mizuno')                             # will be the youngest, Yui.
>>> singersQueue.get()
(15, 'Moa Kikuchi')
>>> singersQueue.get()
(16, 'Suzuka Nakamoto')
>>> singersQueue.get()
(17, 'Ayaka Sasaki')
```

For our task of building a hierarchical clusterer, we will put the clusters in a priority queue. The priority will be the shortest distance to a cluster's nearest neighbor. Using our dog breed example, we will put the Border Collie in our queue recording that it's nearest neighbor is the Portuguese Water Dog at a distance of 0.232. We put similar entries into the queue for the other breeds:



We will get the two entries with the shortest distance, making sure we have a matching pair. In this case we get the entries for Border Collie and Portuguese Water Dog. Next, we join the clusters into one cluster. In this case, we create a Border Collie - Portuguese Water Dog cluster. And put that cluster on the queue:



And repeat until there is only one cluster on the queue. The entries we will put on the queue need to be slightly more complex than those used in this example. So let's look at this example in more detail.

Reading the data from a file

The data will be in a CSV (comma separated values) file where the first column is the name of the instance and the rest of the columns are the values of various attributes. The first line of the file will be a header that describes these attributes:

```
breed,height (inches),weight (pounds)
Border Collie,20,45
Boston Terrier,16,20
Brittany Spaniel,18,35
Bullmastiff,27,120
Chihuahua,8,8
German Shepherd,25,78
Golden Retriever,23,70
Great Dane,32,160
Portuguese Water Dog,21,50
Standard Poodle,19,65
Yorkshire Terrier,6,7
```

The data in this file is read into a list called, not surprisingly, `data`. The list `data` saves the information by column. Thus, `data[0]` is a list containing the breed names (`data[0][0]` is the string 'Border Collie, `data[0][1]` is 'Boston Terrier' and so on). `data[1]` is a list

containing the height values, and `data[2]` is the weight list. All the data except that in the first column is converted into floats. For example, `data[1][0]` is the float 20.0 and `data[2][0]` is the float 45. Once the data is read in, it is normalized. Throughout the description of the algorithm I will use the term *index* to refer to the row number of the instance (for example, Border Collie is index 0, Boston Terrier is index 1, and Yorkshire Terrier is index 10).

Initializing the Priority Queue

At the start of the algorithm, we will put in the queue, entries for each breed. Let's consider the entry for the Border Collie. First, we calculate the distance of the Border Collie to all other breeds and put that information into a Python dictionary:

```
{1: ((0, 1), 1.0244),      the distance between the Border Collie (index 0) and the Boston Terrier
                           (index 1), is 1.0244
 2: ((0, 2), 0.463),       the distance between the Border Collie the Brittany Spaniel is 0.463
 ...
 10: ((0, 10), 2.756)}     the Border Collie -- Yorkshire Terrier distance is 2.756
```

We will also keep track of the Border Collie's nearest neighbor and the distance to that nearest neighbor:

```
closest distance: 0.232
nearest pair: (0, 8)
```

The closest neighbor to the Border Collie (index 0) is the Portuguese Water Dog (index 8) and vice versa.

The problem of identical distances and what is with all those tuples.

You may have noticed that in the table on page 8-7, the distance between the Portuguese Water Dog and the Standard Poodle and the distance between the Boston Terrier and the Brittany Spaniel are the same—0.566. If we retrieve items from the priority queue based on distance there is a possibility that we will retrieve Standard Poodle and Boston Terrier and join them in a cluster, which would be an error. To prevent this error we will use a tuple containing the indices (based on the `data` list) of the two breeds that the distance represents. For example, Portuguese Water Dog is entry 8 in our data and the Standard

Poodle is entry 9, so the tuple will be (8,9). This tuple is added to the nearest neighbor list. The nearest neighbor for the poodle will be:

```
['Portuguese Water Dog', 0.566, (8,9)]
```

and the nearest neighbor for the Portuguese Water Dog will be:

```
['Standard Poodle', 0.566, (8,9)]
```

By using this tuple, when we retrieve items from the queue we can see if they are a matching pair.

Another thing to consider about identical distances.

When I introduced Python Priority Queues a few pages ago, I inserted into the queue, tuples representing the ages and names of Japanese Idol performers. These entries were retrieved based on age. What happens if some of the entries have the same age (the same priority)? Let's try:

```
>>> singersQueue.put((15,'Suzuka Nakamoto'))
>>> singersQueue.put((15,'Moa Kikuchi'))
>>> singersQueue.put((15, 'Yui Mizuno'))
>>> singersQueue.put((15, 'Avaka Sasaki'))
>>> singersQueue.put((12, 'Megumi Okada'))
>>> singersQueue.get()
(12, 'Megumi Okada')
>>> singersQueue.get()
(15, 'Avaka Sasaki')
>>> singersQueue.get()
(15, 'Moa Kikuchi')
>>> singersQueue.get()
(15, 'Suzuka Nakamoto')
>>> singersQueue.get()
(15, 'Yui Mizuno')
>>>
```

You can see that if the first items in the tuples match, Python uses the next item to break the tie. In the case of all those 15 year olds, the entries are retrieved based on the next item, the person's name. And since these are strings, they are ordered alphabetically. Thus the entry for Avaka Sasaki is retrieved before Moa Kikuchi and Moa is retrieved before Suzuka, which is retrieved before Yui.