

Chapter 7: Naïve Bayes and Text

Classifying unstructured text

In previous chapters we've looked at recommendation systems that have people explicitly rate things with star systems (5 stars for Phoenix), thumbs-up/thumbs-down (Inception--thumbs-up!), and numerical scales. We've looked at implicit things like the behavior of people—did they buy the item, did they click on a link. We have also looked at classification systems that use attributes like height, weight, how people voted on a particular bill. In all these cases the information in the datasets can easily be represented in a table.

age	glucose level	blood pressure	diabetes?
26	78	50	1
56	111	72	1
23	81	78	0

mpg	cylinders	HP	sec. 0-60
30	4	68	19.5
45	4	48	21.7
20	8	130	12

This type of data is called “structured data”—data where instances (rows in the tables above) are described by a set of attributes (for example, a row in a table might describe a car by a set of attributes including miles per gallon, the number of cylinders and so on). Unstructured data includes things like email messages, twitter messages, blog posts, and newspaper articles. These types of things (at least at first glance) do not seem to be neatly represented in a table.

For example, suppose we are interested in determining whether various movies are good or not good and we want to analyze Twitter messages:

The image displays six separate Twitter posts arranged in two columns. The top row contains three tweets, and the bottom row contains three more. All tweets are related to the movie *Gravity*.

- Debra Murphy (@DebraSMurphy)** posted on 19 Oct: "I am so stunned by the hype over #Gravity. Please - save your \$\$\$." (with an image of her face).
- NayanD (@NayantharaU)** posted on 19 Oct: "The movie #Gravity might be in my top 10 of all time. Really incredible but the title is so misleading!" (with an image of her face).
- Phileena Heuertz (@phileena)** posted on 17 Oct: "If you haven't seen the movie #gravity don't miss it! Mind-blowing metaphor for #transformation &... instagram.com/p/1ksGX-" (with an image of her face).
- Andy Gavin (@asgavin)** posted on 19 Oct: "#Gravity – Puts the Thrill Back in Thriller - shar.es/EwNuL - visuals, excitement, good acting, what more do you want?" (with an image of his face).
- Tahmoh Penikett (@TahmohPenikett)** posted on 17 Oct: "I didn't think #gravity could possibly live up to the hype. It did, and then some. Game changer. Ground breaking, on edge of your seat, film!" (with an image of his face).
- Jack Mirkinson (@jackmirkinson)** posted on 17 Oct: "#scandal thoughts so far: ugh bomb scares. Plus The Counselor looks like one of the worst movies ever made." (with an image of a newspaper).
- The Dissolve (@thedissolve)** posted on 19 Oct: "The Schwarzenegger-Stallone team-up **ESCAPE PLAN** is small enough to make its diminished stars seem big again: bit.ly/19MeGWK" (with an image of a movie poster).

We, as speakers of English can see that Andy Gavin likes *Gravity*, since he said “puts the thrill back in thriller” and “good acting.” We know that Debra Murphy seems not so excited about the movie since she said “save your \$\$\$.” And if someone writes “I wanna see Gravity sooo bad, we should all go see it!!!” that person probably likes the movie even though they used the word *bad*.

Suppose I am at my local food co-op and see something called Chobani Greek Yogurt. It looks interesting but is it any good? I get out my iPhone, do a google search and find the following from the blog “Woman Does Not Live on Bread Alone”:

Chobani nonfat greek yogurt.

Have you ever had greek yogurt? If not, stop reading, gather your keys (and a coat if you live in New York) and get to your local grocery. Even when nonfat and plain, greek yogurt is so thick and creamy, I feel guilty whenever I eat it. It is definitely what yogurt is MEANT to be. The plain flavor is tart and fantastic. Those who can have it, try the honey version. There's no sugar, but a bit of honey for a taste of sweetness (or add your own local honey-- local honey is good for allergies!). I must admit, even though I'm not technically supposed to have honey, if I've had a bad day, and just desperately need sweetness, I add a teaspoon of honey to my yogurt, and it's SO worth it. The fruit flavors from Chobani all have sugar in them, but fruit is simply unnecessary with this delicious yogurt. If your grocery doesn't carry the Chobani brand, Fage (pronounced Fa-yeh) is a well known, and equally delicious brand.

Now, for Greek yogurt, you will pay about 50 cents to a dollar more, and there are about 20 more calories in each serving. But it's worth it, to me, to not feel deprived and saddened over an afternoon snack!

<http://womandoesnotliveonbreadalone.blogspot.com/2009/03/sugar-free-yogurt-reviews.html>

Is that a positive or negative review for Chobani? Even based on the second sentence: *If not, stop reading, gather your keys ... and get to your local grocery store*, it seems that this will be a positive review. She describes the flavor as *fantastic* and calls the yogurt *delicious*. It seems that I should buy it and check it out. I will be right back...



An automatic system for determining positive and negative texts.



John, that looks like a positive tweet for Gravity!

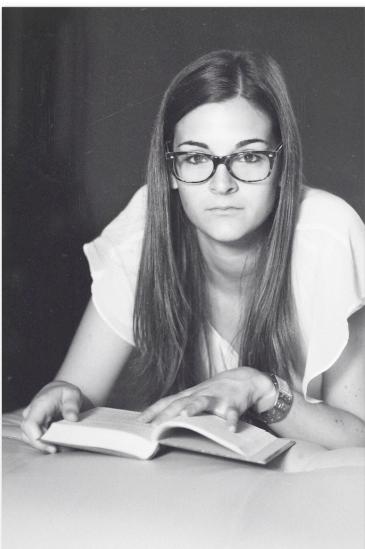
Let's imagine an automatic system that can read some text and decide whether it is a positive or negative report about a product. Why would we want such a system? Suppose there is a company that sells health monitors, they might want to know about what people are saying about their products. Are what people say mostly positive or negative? They release an ad campaign for a new product. Are people favorable about the product (*Man, I sooo want this!*) or negative (*looks like crap*). Apple has a press conference to talk about the iPhone problems. Is the resulting press coverage positive? A Senate candidate delivers a major policy speech—do the political bloggers view it favorably? So an automatic system does sound useful.



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.



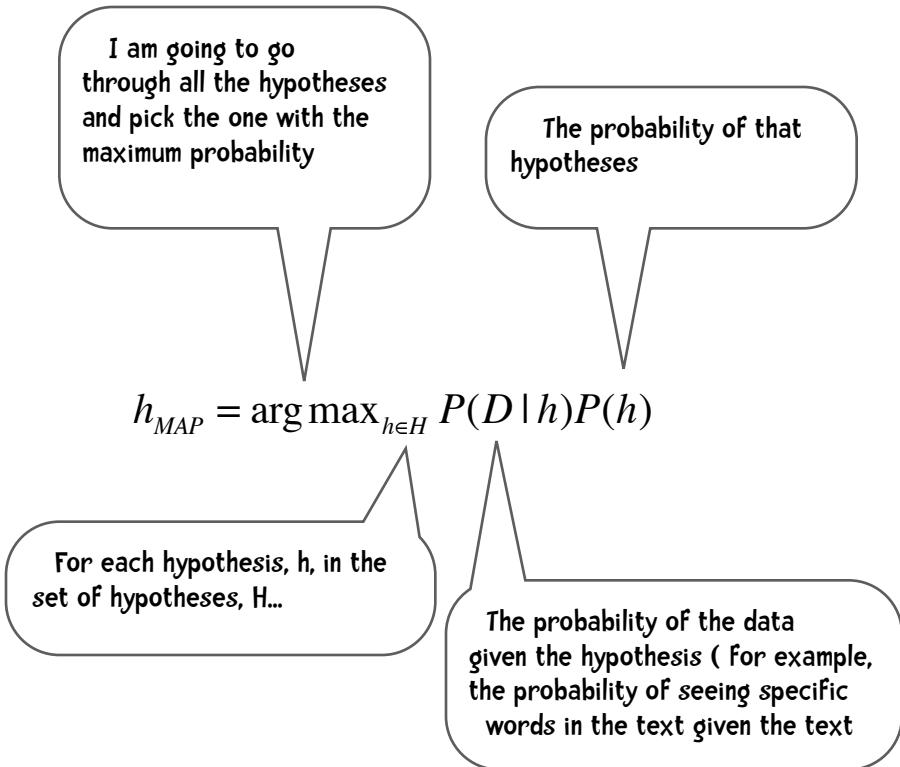
If we are trying to determine if a particular reviewer likes Chobani yogurt or not, we can just count the number of 'like' words and the number of 'dislike' words in their text. We will classify the text based on which number is higher. We can do this for other classification tasks. For example, if we 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!!

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

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

$$P(\text{favorable}) = 0.5$$

$$P(\text{unfavorable}) = 0.5$$



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

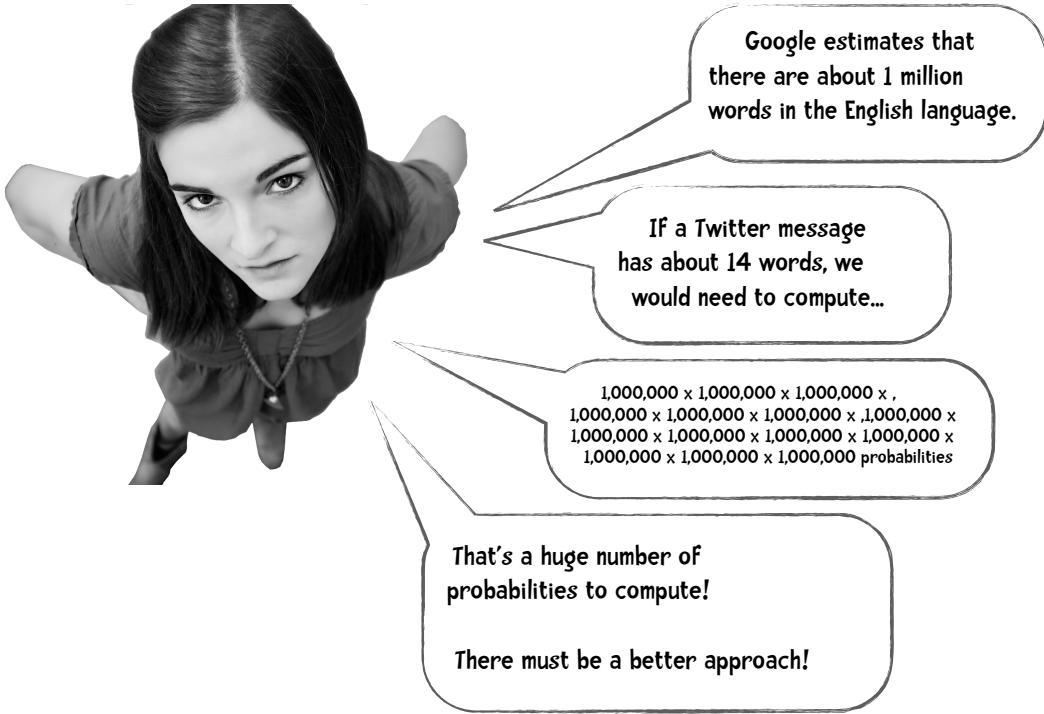
Learning From unlabeled text is called unsupervised learning. One example of unsupervised learning is clustering which we will cover in the next chapter.

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

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

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

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

Naïve Bayes Classification Phase

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

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



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

$$P(\text{like}) = 0.5$$

$$P(\text{dislike}) = 0.5$$

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

$$P(\text{like}) \times P(I \mid \text{like}) \times P(\text{am} \mid \text{like}) \times P(\text{stunned} \mid \text{like}) \times \dots$$

and

$$P(\text{dislike}) \times P(I \mid \text{dislike}) \times P(\text{am} \mid \text{dislike}) \times P(\text{stunned} \mid \text{dislike}) \times \dots$$

and chose the hypothesis associated with the highest probability.

word	$P(\text{word} \text{like})$	$P(\text{word} \text{dislike})$
	$P(\text{like}) = 0.5$	$P(\text{dislike}) = 0.05$
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
Π	6.22E-22	4.72E-21

So the probabilities are

like 0.0000000000000000000000000622

dislike 0.00000000000000000000000004720

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 logarithm (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,

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

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.

PLEASE THROW OUT
POR FAVOR TIRALO
PROSZE WYRZUCIC
PLEASE THROW OUT
POR FAVOR TIRALO
PROSZE WYRZUCIC
PLEASE THROW OUT
POR FAVOR TIRALO
PROSZE WYRZUCIC
PLEASE THROW OUT
POR FAVOR TIRALO
PROSZE WYRZUCIC

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

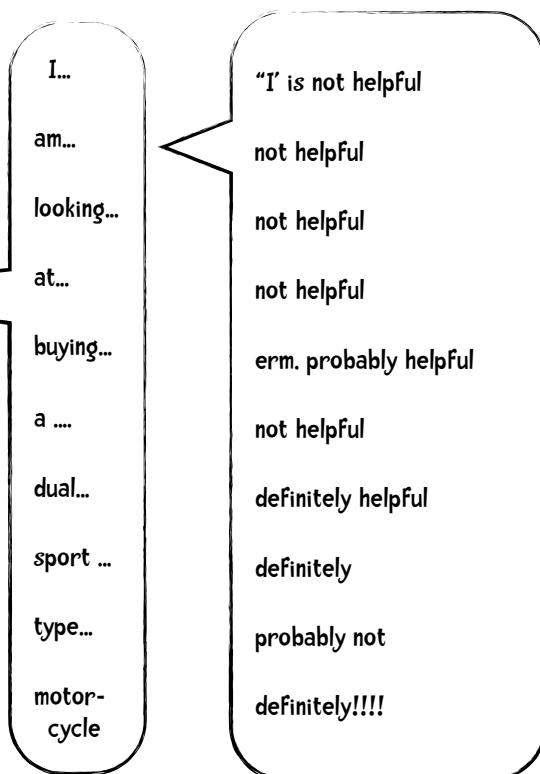
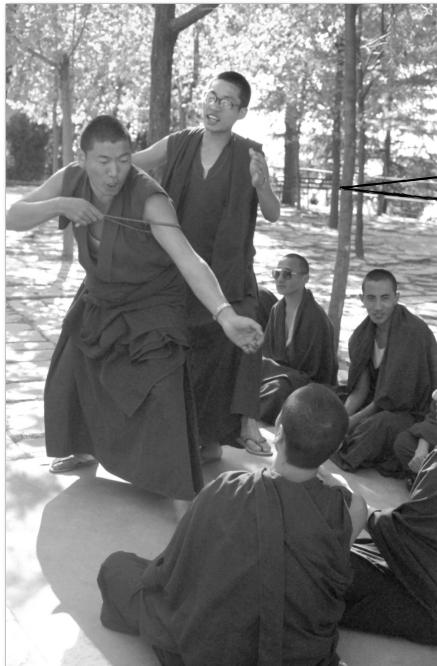
For example, we would like to build a system that would classify the following post as being from rec.motorcycle:

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.



Don't blindly remove stop words.
Think First.

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

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

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 + category)
        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)

71.71% accuracy is pretty good...
I wonder what the accuracy would be
if we used a stoplist?

Only one way to find out ...



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	71.714827
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 2004¹. 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

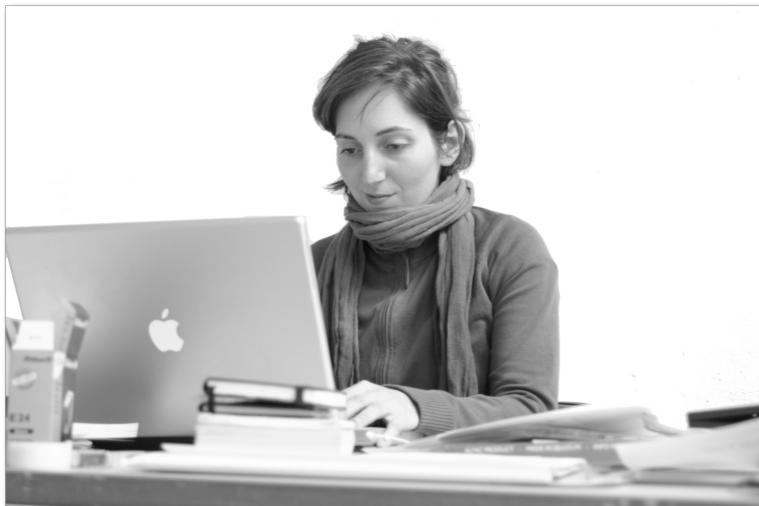
Also compute the kappa coefficient.

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)

```