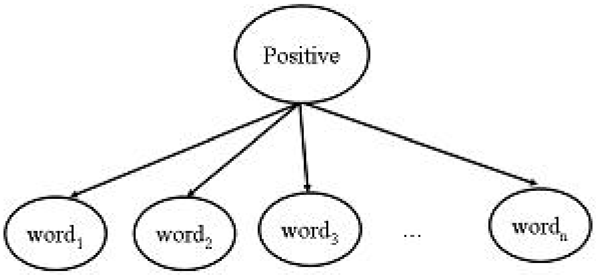
CS151 - Assignment 4
Due: Wednesday, Oct. 20 at midnight

The purpose of this assignment is to program a simple Bayesian classifier: you will implement a Naive Bayes Classifier that analyzes a number of text classification tasks. You may work in pairs on this assignment if you'd like. As always, read through the **entire** document before starting.

Overview

Recall that a Naive Bayes classifier is just a simple Bayes net that has one root node with some number of children. The children are connected only to the root, not to each other, as in the network shown below:



This is the Naive Bayes network we will use for this assignment. In this network, all variables are boolean. The variable "*Positive*" represents whether the document is a positive review ($Positive = false$ means the document is a negative review), and the variables "$word_1, word_2, ..., word_n$" represent the presence or absence of different words in the document.

The basic idea is that certain words are more likely to occur in postive documents, while others are more likely to occur in negatives ones. However, the Naive Bayes assumption we make is that once we know whether the document is positive or negative, the occurance of the words in the document become independent from one another (this of course is not true, but works reasonably well in practice).

Your task in this assignment will be to learn then apply the CPTs for the network above for a few different problem domains, e.g. $P(Positive)$ and $P(word\_i|Positive)$ for $i = 1$ to n.

Data

We have three different data sets we'll be playing with (two required and a third, optional):

- **Reviews:** This data set consists of approximately 14K reviews collected from `www.rateitall.com`. The reviews come from a variety of domains including movies, music, books, and politicians. The actual data has ratings from 1 to 5, but I have cleaned these up and left only 1s and 5s, denoted those with a 5 score as "positive" and those with a 1 as "negative", resulting in a binary sentiment prediction task. The data resides in the file `reviews.data`

- **UseNet posts:** Obtained from `http://www.cs.umass.edu/~mccallum/code-data.html`, this data set consists of approximately 75K UseNet articles from four discussion groups for simulated auto racing, simulated aviation, real autos and real aviation. I have created two binary classification tasks from this data set. `sraa.auto_aviation.data` has documents labeled with whether they are related to auto or aviation. `sraa.real_sim.data` has documents labeled with whether they are related to real or simulated discussion. Notice that these two data sets contain the same articles, they just have different labels depending on the task.

- **20 Newsgroups:** Both of the above data sets are binary classification problems. We can also classify between more than two classes (picking the most probable under the models). For extra credit, you can implement a classifier that handles multiple classes. This data set was obtained from `http://people.csail.mit.edu/jrennie/maintain.html` and consists of newsgroup postings on 20 different topics. The data resides in the files `20news.data`.

Getting Started

Because of the size of the data I have put all of the starter code and data on the CS network at Pomona. It can be found at:

```
/common/cs/cs151/assignment4/
```

which you can access remotely via sshing to vpn.cs.pomona.edu.

I have provided you with two python files to get you started:

- `DataReader.py`: This file contains some basic functionality for reading and processing the data files. To read the documents from a data file, create a new DataReader and iterate over it:

  ```
  reader = DataReader(dataFile)

  for label, tokens in reader:
      print label
      print tokens
  ```

  Notice that tokens is a list of tokens.

  Also in this file are two other functions that could be useful. `tokenize` takes a string and returns a list containing the words in that string (you probably won't need this, but just in case). `split` takes a data file and an output label and splits the data with 80% going into a training file, 10% into a development file and 10% a test file. This will prove useful when you start trying to evaluate your system.

- `BayesClassifier.py`: This contains some basic starter code to get you going.

1 Training

Now that youre familiar with the data and starter code, your next task is to "train" the classifier. As you know from your reading and from class, a Naive Bayes classifier is just a simple Bayes Net, in this case the one shown above.

We will use maximum likelihood (ML) parameter estimation to estimate each parameter in the Bayes net. To simplify the notation below, I'll assume we're looking at the task of sentiment analysis with two classes, positive and negative, though the training will be general regardless. As a reminder, the parameters that we need to estimate are all of the conditional and prior probabilities for the network:

$P(Positive)$ and $P(Negative)$: The prior probabilities of the classes
$P(word_i|Positive)$ and $P(word_i|Negative)$: The conditional probability that $word_i$ appears, given that a document has a particular label/class, for each $word_i$.

For the 20 newsgroup data set, you have 20 different labels and therefore 20 sets of prior probabilities and conditional probabilities!

Recall from class that the ML estimate for these parameters are:

$P(positive)$ = # of positive docs / total number of docs
$P(word_i|positive)$ = (# $word_i$ in positive docs) / (# words in all positive docs)

When training the system, instead of storing the actual parameter estimates, just calculate the frequencies needed to calculate the parameters on the fly. In calculating these parameter estimates, you will need to keep track of a few things:

- how often each word appears in the positive documents
- how often each word appears in the negative documents
- how many positive documents there are
- how many negative documents there are

For the word occurrences, there are two different types of tallies that are commonly used: presence or frequency. Presence refers to the number of documents in the training data that the feature occurs in (A word either occurs or it doesn't in a document. If it appears multiple times, it still just counts as occurring.) while frequency refers to the number of times that the feature occurs in the training set (i.e. if it occurs three times in one document, that counts as three). We will use frequency counts for this part of the assignment to estimate parameters, but you might experiment in part 4 with using presence counts instead.

Given all of the counts that are to be stored, Nave Bayes Classifiers typically use a database. However, since you are writing this code in Python, we'll instead use a handy Python utility called pickle. Using pickle, one can write a data structure to a file, and then load it back into memory later. In particular, you'll have one or more dictionaries (like hashtables) holding the above counts. Since these dictionaries are time consuming to construct, it is useful to only calculate them once, pickle them, and then load them into memory the next time you need to use them.

To do this, the following steps would be a good approach:

1. Create any dictionaries, etc. in the init method that you will need.
2. Fill in the details of the `train` method. This method should fill in the counts based on the file provided. You'll probably need to use the `DataReader` class here. Do not change the input parameters!
3. Fill in the details of the `save` method. This method should use pickle to save all of your dictionaries, etc. to a file. You can call `p.dump` in sequence to write multiple objects to a single file. Again, do not change the parameters.
4. Fill in the details of the `load` method. This method will again use pickle to load into the current object all of the stored dictionaries, etc. You should call `u.load` for each variable you dumped in save, in the same order and save it back to this object.

2 Classifying

At this point, you now have a trained model/set of counts and are ready to classify. Given a piece of text, the goal is for your system to output the correct document class (e.g. positive or negative). In particular, you'll calculate the conditional probability of each document class given the features in the target document (e.g. $P(positive|word_1, word_2, ..., word_n)$) and return the document class of the highest probability.

To classify a document, you need to find:
$P(positive|word_1, word_2, ..., word_k)$

That is, the probability that a document is postive, given the words in that document. Like any Bayes Net, the Naive Bayes classifier calculates this probability as the product of the probability of each node in the network, given the value for its parents. In this case:

$$P(positive|word_1, ..., word_k) = \alpha P(positive)P(word_1|positive)..P(word_k|positive)$$

Note, however, that this probability ignores some information that we are given. In particular, it only considers words that you HAVE seen in the document, and it does not factor in the words that appearn in the Bayes net that you HAVE NOT seen. What we're really after is:

$$P(positive|word_1, ..., word_k, \neg word_{k+1}, ..., \neg word_n) =$$
$$\alpha P(positive)P(word_1|positive)...P(word_k|positive)P(\neg word_{k+1}|positive)...P(\neg word_n|positive)$$

where $word_{k+1}, ..., word_n$ are words that are in the full Bayes Net (i.e., words seen in some document in the training set), but do not happen to appear in this document. We'll use the simplification above and only take into account the words that did occur. Effectively this simplification states that we have not observed $word_{k+1}, ..., word_n$, i.e., we haven't seen them, but we haven't NOT seen them either.

Reflection question #1: In your "reflection_evaluation.txt" file, answer the following question:
Give at least one reason why we ignore words we have not see in the document when classifying the document (i.e., why not use the full specification that includes words that are NOT in the document?).

Furthermore, we don't *really* care about α, since it is the same for $P(positive|[words\_in\_doc])$ and $P(negative|[words\_in\_doc])$. Thus, you can ignore it and just compare the unnormalized probabilities for positive and negative directly.

You will find that two problems will arise in building this basic classifier. I suggest that you build the classifier first, and then address the problems:

- Underflow – Because you are multiplying so many fractions, the product becomes too small to be represented. The standard solution to this problem is to calculate the sum of the logs of the probabilities, as opposed to the product of the probabilities themselves.

- Smoothing – This problem is more subtle as it wont produce a bug, but will throw off your calculations. Suppose a feature (word) f does not occur at all in the training data for negative (or positive) documents. Then our estimate for $P(f|negative) = 0$. This one missing word essentially negates the influence of all other words in the document! One word should not be given that much power in our decisions. See our class notes for a discussion of smoothing. A popular solution is "add one smoothing", setting $\lambda = 1$, though this tends to be too large in practice. This is one thing you can experiment with when developing your "best" classifier.

Fill in the details for the `classify` method that takes a string as input (note you'll probably need to use the `tokenize` method here. The function should return the label that you think the text should have (positive vs. negative in the reviews case).

If you've followed my instructions, you should now have a classifier that can be used in the following manner:

```
>>> execfile("BayesClassifier.py")
>>> bc = BayesClassifier()
>>> bc.train("reviews.data")
>>> bc.classify("I love my AI class!")
positive
```

Reflection Question #2: Take a moment to celebrate your success (yay!). Try out your classifier on a few sentences and see when it performs well and when it performs poorly. What are three examples of sentences (of any length of text) on which you think your classifier failed? For each example, why do you think it failed? What was hard about the target text? Write your answers to these questions in your file "reflection_evaluation.txt".

3 Improve your Classifier

Hopefully in the previous part, you probably realized some ways in which you can improve upon your system. In this part, I would like you to be creative and implement these ideas and build your best classifier. Time permitting, we'll hold a competition in class to see whose classifier does best on a set of documents that I'll choose randomly. The outcome of this competition will not have an impact on your grade.

For this part, **make a copy of your python file** and add the word "best" to the end of the filename. This will make it easier for us to grade these parts separately.

Much of the research with respect to Nave Bayes Classifiers is the choice of features used in classification (i.e., the children nodes in the Bayes net). Thus far, you've only used "unigrams" (i.e. single words) as features. Another common feature is "bigrams" (i.e. two word phrases). Another feature might be the length of the document, or amount of capitalization or punctuation used. For your best classifier, you must include at least two other features in your classification. Another thing to experiment with is smoothing, which can have a big impact of performance.

Reflection #3: Briefly explain what changes you made to your improved classifier.

4 Evaluation

Now that you have a working system, you can evaluate how well your system works. To do this, you should split the data into train and test using the `split` function. You should **never** train and test on the same data! If you want to be extra legitimate, when you're developing your approach, you use a development data set for testing incremental improvements and only when you're finally done, do you use the test data set.

We'll look at four different evaluation measures:

- Accuracy: number correct / total
- Precision: number correctly classified as positive / total classified as positive (similarly you can calculate the precision for other classes, such as negative, etc.). Measures how good we are at predicted one class.
- Recall: number correctly classified as positive / total number of positive. Measure how good we are at finding/identifying positive documents.

- F1-measure: 2 * (precision * recall) / (precision + recall). This is one approach for summarizing both precision and recall.

Notice that only accuracy is measured over all classes. The other measures give you a per class measure, i.e. your results for just the positive class or just the negative class.

Once you have your evaluation measures working, play with the different systems and classification tasks. Include a table summarizing your results and write a paragraph or two analyzing them. This will be good practice for when you're working on your own research project.

- Compare your base system and your best classifier on the reviews task. How do they compare? Any differences? Do you think your improvements had a significant impact.
- Now compare your systems out on **both** of the UseNet tasks (real vs. simulated and auto vs. aviation). What do your results imply about the difficulty of these tasks? Why do you think one task is harder/easier than the other? Did your improved system have a larger impact on any one of the data sets? Compare both the UseNet tasks as well as the review tasks.
- Another thing you might investigate (though it's not required) is how the amount of training data affects your performance.

In additional to analyzing your results, also include a "Future Work" section that outlines ways that you would extend the system to improve future performance. Include thise in your "reflection_evaluation.txt".

5 Extra Credit: Multi-class classification

This portion is optional. The 20 newsgroups data set includes 20 classes instead of just two. Modify your code to support multiple classes. One way to do this is instead of having separate dictionaries for each class, use one dictionary keyed off of the label/class whose value is another dictionary with the word counts. In your classify method, you'll then need to calculate log probabilities for all of these classes and pick the label with the largest of all 20.

If you implement this, also include a section in your writeup discussing your results.

When you're done

When you're all done, follow the directions on the course web page for submitting your work. Make sure that your code compiles, that your files are named as specified and that all your functions have the same name and number of parameters. If you get an error, try changing the name of the folder to include a version number and resubmit.

If you worked with a partner, put both people's last names on the submitted directory, but only submit one copy.

What to submit

- `BayesClassifier.py` supporting the functionality described above.
- `BayesClassifier.best.py` which includes your improved (hopefully) version of the classifier.
- `reflection_evaluation.txt` which should contain your answers to the three reflections (make it clear where your response to each question is), a section showing your evaluation results and analyzing them and a section describing future improvements.

Commenting and code style

Your code should be commented appropriately (though you don't need to go overboard). The most important things:

- Your name (or names) and the assignment number should be at the top of each file
- Each class and method should have a short "docstring"
- If anything is complicated, put a short note in there to help the graders out if there are any issues.

There are many possible ways to approach this problem, which makes code style and comments very important here so that the grader and I can understand what you did. For this reason, you will lose points for poorly commented or poorly organized code.

Grading

Part	points
train	15
load/save	5
classify	30
improve	10
evaluation/write-up	30
style/commenting	10
extra credit	10
total	100 + 10 extra