Introduction to Artificial Intelligence – CS 510
Machine Learning Programming Assignment – Sentiment Analysis

The purpose of this assignment is to apply what you've learned about approaches in machine learning to a real world problem. In particular, you will implement a Naïve Bayes Classifier in Python that analyzes the *sentiment* conveyed in text. When given a selection of text as input, your system will return whether the text is positive, negative, or neutral. The system will use a corpus of movie reviews as training and testing data, using the number of stars assigned to the each review by its author as the truth data (1 star is negative, 5 stars is positive). Depending on time, we may hold a short tournament to test the accuracy of your systems.

Part 1: The Training Data

To train your system, you will be using data from a corpus of movie and product reviews collected from www.rateitall.com. The reviews come from a variety of domains, e.g., movies, music, books, and politicians and are stored in individual text files where the file name is of the form *domain- number_of_stars-id.txt.* For example, the file "movies-1-32.txt" contains a movie review that was assigned one star by its author and has an id of 32. Please note that the id number will be of no use to you for this assignment.

For your system, you will only be using reviews that had one or five stars, where the reviews with one star are the "negative" training data and the reviews with five stars are the "positive" training data.

I've provided you with two Python functions (in the template file) to help you in using this training data.

1. 1) loadFile – a simple Python function that takes a filename and returns a string of the file's contents.
2. 2) tokenize – takes a string of text and returns a list of the individual words that occur in the text.

Familiarize yourself with these functions by using them at the python interpreter to read reviews from the training data files. Take a moment to explore the code provided in the template file so that you don't create extra work for yourself.

Part 2: Train the System

Now that you're familiar with the training data, your next task is to train the classifier. As you know from your reading and class, given a **target document** (a document to be classified), Naïve Bayes Classifiers calculate the probability of each **feature** in the target document (i.e. individual word) occurring in a document in each **document class** (i.e. positive, negative and neutral) in order to find the class that is most probable. In order to make these calculations, the system must store the number of times that each feature occurs in each document class in the training data. For example, given a word like "good," it may occur 1500 times in the positive documents, but only 200 times in the negative documents. There are two different types of tallies that are commonly used: **presence** refers to the number of documents in the training data that the feature occurs in and **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).

Given all of the counts that are to be stored, Naïve 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 two dictionaries (like hashtables), one which holds all of the words that occur in positive documents and the frequencies at which they occurred, and the corresponding dictionary for

negative documents. 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.

Write a function called *train* that takes no parameters, and trains the system by:

- Getting the names of all of the files in the "reviews" directory using *os.walk("reviews/")* in the following manner:
    - lFileList = []
    - for fFileObj in os.walk("reviews/"):
    - lFileList = fFileObj[2]
      break

  Following the execution of the above code, the list of filenames will be stored in lFileList.

- For each file name, parse the file name and determine if it is a positive or negative review.
- For each word in the file, update the frequency for that word in the appropriate dictionary.
- Save these dictionaries using "pickle" so that the system only has to calculate them once. I've provided you with two functions (*save* and *load*) to help with this step.

Write the initialization function for the *Bayes_Classifier* class (*__init__*) that does the following:

- Initializes the two dictionaries as attributes of the class.
- If the *pickled* files exist, then load the dictionaries into memory.
- If the *pickled* files do not exist, then train the system.

Part 3: Start Classifying!

At this point, you now have all the data you need stored in memory and can start classifying text! Given a piece of text, the goal is for your system to output the correct document class (i.e. positive, negative, or neutral). In particular, you'll calculate the conditional probability of each document class given the features in the target document (e.g. P(positive | word1, word2, ...)) and return the document class of the highest probability.

Naïve Bayes Classifiers find the conditional probability of a document *d* being a member of a class *c* given the features in *d* by calculating the product of all of the **conditional probabilities** of each of the individual features (i.e. words) of document *d* occurring given that a document is of class *c*, multiplied by the **prior probability** of any document being a member of class *c*. The conditional probability of a feature *f* occurring given that a document is of class *c* is equal to the training frequency of feature *f* in class *c* divided by the sum of all of frequencies of features in class *c*. The prior probability of a class *c* is simply equal to the fraction of training documents from class *c.*

In case you do better with equations, assume we are trying to calculate the probability that a target document *d* is positive given the set of *n* features of *d,* which we'll call *f.*

$$P(positive \mid f) = P(positive) * \prod_{i=1}^{n} P(f_i \mid positive)$$

The first term on the right side of the equation is the prior probability of any document being positive. The second term on the right side of the equation is the product of the conditional probabilities of each of the features occurring given that a document is positive.

Here is an expanded version of this calculation:

$$P(positive|f) = P(positive) * P("I"|positive) * P("love"|positive) * P("school"|positive)$$

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

1. 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.
2. Smoothing – This problem is more subtle as it won't produce a bug, but will throw off your calculations. Suppose a feature *f* occurs once in the training data for class *c1* and not at all in the training data for class *c2*. If a document *d* contains feature *f*, the probability for class *c2* (all else equal) will be higher than the probability for class *c1,* which intuitively should not be the case. Read online about "add one smoothing" as a solution this problem.

Write a function *classify* that takes as input a string of text and returns a string of the classification – either "positive," "negative," or "neutral." Note that in building your classifier, you only have data to calculate the probability of a document being positive or negative, so you'll have to come up with some way to determine if a document is neutral (keep it simple).

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

```
>>> execfile("bayes.py")
>>> bc = Bayes_Classifier()
>>> result = bc.classify("I love my AI class!")
>>> print result
positive
```

Now, write a function that takes the address of a folder as an input, which contains a bunch of test files (without the label in the file name), and returns the classifier's performance on the terminal. For your testing, you can use a part of the trainset without using the labels.

Code it by command line arguments, so that the test folder address can be passed when calling the runnable object on terminal.

Reflection Question: Take a moment to celebrate your success. 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 (or 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?

Part 4: Improve your Classifier!

Congratulations! Once you reach this point, you will have built your first Naïve Bayes Classifier! If 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*. 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 me to grade these parts separately. We expect the improved classifier to return a higher performance in classifying the same test files.

The most important research problem with respect to Naïve Bayes Classifiers is the choice of features used in classification. 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 one other feature in your classification.

Part 5: Evaluate Your System

I’d now like you to evaluate how well your system works. In particular, I’d like you to compare your base system (that you completed in part 3) to *your best classifier* (that you completed in part 4). Not only should you compare the two systems based on their **precision**, **recall**, and **f- measure**, but also reflect on why you think the systems performed well (or poorly). Outline ways that you would extend the system to improve future performance. Consider and present other ways that you might approach this task of classifying sentiment in text.

Part 6: Turn it in

Submit the following three files via BBLearn (1) your naïve classifier bayes.py, (2) your improved classifier bayesbest.py, and (3) A document including your answers to the reflection question, an explanation of the feature(s) you used to improve your system, and your evaluation.

Item	Points
Naïve Classifier (proper algorithm, well commented, correct input and output format)	35
Improved Classifier (proper algorithm, well commented, correct input and output format)	25
Evaluation documentation	30
Readme (including your name, what the files are and what they contain, also the compilation and execution instructions on tux.)	10

This assignment is based on an assignment designed by Sara Sood of Pomona College. Thanks Sara!