Assignment 5: Detecting Sentiment

Overview
The purpose of this assignment is to apply what you've learned about Supervised Machine Learning and practice using python dictionaries, type hints and the pickle library.

You will implement a Naïve Bayes Classifier that analyzes the *sentiment* conveyed in text. When given a selection of text as input, your system will return whether the text is positive, or negative. The system will use a corpus of movie reviews as training data, using the number of stars assigned to each review by its author as the truth data (1 star is negative, 5 stars is positive).

More Details
See the teams document for who you'll be working with. If all team members are present for the working sessions, you can simply have one team member submit for the team, with the other team members noted in a comment on canvas.

Here's a high level overview of a Bayes Classifier. Later in this document, we provide a more in-depth, mathier explanation.

Let's say we have a movie review called `mystery-?.txt`. The text of our review is `amazing movie`. Let's also say that for every word we know the probability of that word existing in a positive review and the probability of that word existing in a negative review. For example, we might know that the word `amazing` shows up 8 times in positive reviews out of 1032 total word occurrences and 2 times in negative reviews out of 743 total word occurrences.

P("amazing" | positive) = 8/1032 ~= .0077
P("amazing" | negative) = 2/743 ~= .0027

Okay, now we're ready to classify our document. We're going to calculate two probabilities, the probability that our document is positive and the probability that it is negative. Whichever probability is higher will be the output. To calculate the probability of the document being positive we will combine the probability of each individual word occurring, given that we are in a positive document. For now we'll just multiply the probabilities together, below we mention issues with this and a better way to combine them. For example here's a calculation of positive (made up data):

$$P(positive \mid mystery - ?.txt) = P("amazing"|positive) * P("movie"|positive) = .0077 * .0016$$

For anyone who hasn't seen the above notation, | means *given*, so P(some_event | some_data) means the probability of some_event given some_data. Here we want to calculate the probability of a positive review given the document. To do so we assume the document is positive and multiply the probability of each word given that knowledge. We would then repeat the same calculation for negative. Finally, we would output whichever class had the higher probability.

Important to note that we've simplified one or two things above that we'll explain more in the `Part 3: Start Classifying` section.

The bayes_classifier class

This class has the following attributes:

- `pos_freqs` - dictionary of frequencies of positive words. This is a dictionary of `str->int` where keys are words (tokens) and values are the count of that word across the training texts. For example if the positive reviews had `8` instances of the word `"Tomate"` then `pos_freqs["Tomate"]` would equal `8`.
- `neg_freqs` - dictionary of frequencies of negative words. See `pos_freqs` above except for negative words.
- `pos_filename` - name of positive dictionary cache (pickled) file
- `neg_filename` - name of negative dictionary cache (pickled) file
- `training_data_directory` - relative path to training directory
- `neg_file_prefix` - prefix of negative reviews
- `pos_file_prefix` - prefix of positive reviews

It should also contain the following methods (a few have already been provided, those are marked below):

- `__init__(self)` (**provided**): Initializes various attributes described again and begins training (by calling `train`) if it does not detect `pos.dat` and `neg.dat` data files
- `train(self)`: Trains the classifier by generating `pos_freqs` and `neg_freqs` dictionaries on all files in the folder given by `training_data_directory`.
- `classify(self, text)`: Provides a classification for given text. We'd hope `some_classifier.classify("I love Tomate")` would return `positive`.
- `load_file(self, filepath)` (**provided**): Loads file given by filepath, returning the text of the file.
- `save_dict(self, dict, filepath)` (**provided**): Saves (by pickling) given dictionary to the provided filepath.
- `load_dict(self, filepath)` (**provided**): Loads a picked dictionary at the given filepath, returning the dictionary
- `tokenize(self, text)` (**provided**): Splits given string into a list of individual words (tokens) in the text (all lowercase).

- `update_dict(self, words, freqs)`: Updates the frequency of words in the `freqs` dictionary with the given `words` list. This involves incrementing the count of each word in `words` in the `freqs` dictionary or adding words with a starting count of 1 if they don't already exist in `freqs`.

Your Job
Part 1: The Training Data

To train your system, you will be using data from a corpus of movie reviews collected from RateItAll. 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 we will not use the id number in this assignment so feel free to ignore it.

We will only be training using **movie** reviews that had either 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.

Two provided functions `load_file` and `tokenize` will be helpful here. 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 so that you don't create extra work for yourself.

Part 2: Train the System

Now that you're familiar with the training data, our next task will be to train the classifier (filling in the body of the train method).

Given a **target document** (a document to be classified), Naïve Bayes Classifiers calculate the probability of each **feature** (i.e. each individual word) in the target document occurring in a document of each **document class** (i.e. positive and negative) in order to find the class that is most probable. For example, in our review from before called "mystery-?.txt" with the text "amazing movie" the features would be "amazing" and "movie".

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 329 times in the positive documents, but only 53 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 adds three to the count).

These instructions use **frequency** but feel free to experiment with both.

Naïve Bayes Classifiers typically use a database to store all the counts. 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. You'll have two dictionaries, one holding all the words that occur in positive documents keyed to the number of times they occurred, and the corresponding dictionary for negative documents. If the positive reviews had 300 unique words the positive dictionary would have 300 keys, with the corresponding value equal to the number of times that word appeared across all positive 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.

You're now ready to start implementing training. Before filling out the ‘train’ method, implement the ‘update_dict’ method as you will want to use this in your implementation of ‘train’. ‘update_dict’ takes a list of words and a dictionary and updates the counts in the dictionary with the new words in the given words list (see the docstring in the code for more details).

Once you've implemented ‘update_dict’ you're now ready to fill in the body of the `train` method. Notice that we've started it for you by giving you a list of the training files. Here is a rough overview of the steps you'll need to take:

- For each file name, parse the file name and determine if it's a positive or negative review.
- For each word in the file, update the frequency for that word in the appropriate dictionary. **Hint:** ‘update_dict’ might come in handy here.
- Save these dictionaries using ‘pickle’ so that the system only has to calculate them once. We've provided you with a ‘save_dict’ function to help with this step (the complimentary function ‘load_dict’ is used in ‘__init__’).

More detailed instructions are provided in the comments in the provided code. Be sure to notice the attributes provided in the ‘__init__’ method. In particular, ‘self.pos_freqs’ and ‘self.neg_freqs’ are the dictionaries you'll use to store the data.

After writing the ‘train’ method, you can test it by creating an instance of a ‘bayes_classifier’ because the constructor (‘__init__’) method calls the ‘train’ method. Try the following…

```
b = bayes_classifier()
```

If you change your code and want it to retrain, you'll have to delete the ‘.dat’ files that have been produced/pickled (‘pos.dat’ and ‘neg.dat’) so that they can be reproduced.

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 document with some text, the goal is for your system to output the correct document class (i.e. 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, ...)) and return the document class with the highest probability.

IMPORTANT: In the `Overview` section above we made a few simplifications. Your classifier **must** classify according to the instruction in the following paragraphs, not the simplified version.

Underflow
The first important simplification we made was using the product of conditional probabilities. This causes an issue called underflow. Because we are multiplying so many fractions, the product becomes too small to be represented. The standard solution to this problem (which we're going to implement) is to calculate the sum of the logs (base e) of the probabilities, as opposed to the product of the probabilities themselves. Our earlier example with ` mystery-?.txt ` would now look like this:

$$P(positive \mid mystery-?.txt) =$$

$$log(P(\text{"}amazing\text{"}|positive)) + log(P(\text{"}movie\text{"}|positive)) = log(.0077) + log(.0016) \ = -2.11 + -2.80$$

Note: this results in negative probabilities. We will still output whichever class has the larger probability, in this case, the least negative one.

Smoothing
Another important piece is called smoothing (specifically we'll be doing "add one smoothing"). This problem is more subtle as it won't produce a bug, but will throw off your calculations. Suppose a feature **f** (say "amazing") occurs some number of times in the training data for class **positive** and not AT ALL in the training data for class **negative**. If the document we're trying to classify contains feature **f**, the probability for class **negative** (assuming all else equal) will be higher than the probability for class **positive**, which intuitively should not be the case. Additionally, we run into issues with trying to take the log of 0. To solve this problem we pretend that the feature has a count of 1 in the class where it doesn't exist in the training data. To account for the fact that we added 1 to one class we need to also add 1 to the count in each other class. We'll discuss this in class more; you can also read more online by searching "add one smoothing".

In case you prefer 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**:

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

The term on the right side of the equation is the sum of the conditional probabilities of each of the features occurring given that a document is positive.

Here is an expanded version of this calculation (for our "amazing movie" example):

$$P(positive \mid f) = log(P(\text{"amazing"} \mid positive)) + log(P(\text{"movie"} \mid positive))$$

IMPORTANT: The equations above don't show "add one smoothing" but you *must* implement it for an accurate classifier.

Fill in the body of the `classify` method which takes as input a string of text - like "amazing movie" (not a file name). It should return a string of the classification -- either "positive" or "negative". More details are provided in comments inside the method.

Once you've implemented it you should now have a classifier that can be used in the following manner:

```
bc = bayes_classifier()
print(bc.classify("I love python!")) # should print positive
print(bc.classify("I hate python!")) # should print negative
```

OPTIONAL Extensions

Congratulations! Once you reach this point, you will have built your first Naïve Bayes Classifier! In the previous part, you probably realized some ways in which you can improve upon your system. In this part, try and implement these ideas and build **your best classifier**. For this part, it might make sense to create a copy of your code and store it in a separate file so that you don't break your existing classifier.

The most important 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. You could also remove frequently occurring words (stop_words) that are often thought of as noise. See `sorted_stoplist.txt` (provided) or store the frequencies of the stems of the words instead of the words themselves.

Additionally, you could also account for any imbalances in the number of documents in each class. In an extreme example, if we had 1,000,000 positive reviews and 1 negative review we

would want to bias our classifier towards a positive classification. To do so, we could add the prior probability of the document being a given class to each class calculation. That is, when calculating the probability that our text is positive, we would add the log of the probability of a document being positive based on the number of documents, in this case 1,000,000 / 1,000,001 ~= 0.9999. To the negative probability we'd add the log of 1 / 1,000,001 ~= 0.00001. In practice, the prior probability can be somewhat artificial, which is why we did not include it in the initial classifier.