Alok Gupta
Professor Yan
CS 165A

MP1 Report

Architecture

The architecture of the program is quite simple. In fact, I probably should have modularized it more. There's merely one file, NaiveBayesClassifier.py, that performs all pre-processing, training, classification, and analysis of the results.

Pre-Processing

In order to pre-process the text files for the training and classification, I merely read one review at a time. I then tokenized each review, breaking it into the individual words consisting of that review. Each element of this tokenized array was a word in the review, with the last index containing the classification of the review. I performed text normalization during this pre-processing phase, where the text was all converted to lowercase and stop words were converted to empty strings. I noticed that this helped improve accuracy during the classification portion of the program.

Model Building

In order to train my naïve bayes classifier, I counted the frequency of the words that showed up in the two categories of our movie reviews, positive and negative. If a term had a negative term directly before it (i.e. not, can't, shouldn't), I processed the term as its negative + the term itself. This was all stored in a list of lists with a dictionary for each term within these list to contain its metadata. With this data, I then calculated the probability of a positive review and a negative review. Using bayes theorem, I calculated the probability of a word given a certain category. To do so, I first get the frequency of each word and separate by positive and negative reviews. I get the positive conditional probability first, and then I do negative words, but only those that were not present in positive. Using this, I'm able to calculate word probabilities based on the word count data collected during the training process.

Results

My naïve bayes classifier achieved the following accuracy on the given datasets:

- Training Data: 96.12% accuracy
- Testing Data: 84%

The running time for training is $O(D*Ld + |C| |V|)$ where D represents the number of reviews and Ld is the average length of these reviews. The category is represented as C, and V represents the total size of the vocabulary of the provided data sets.

Top 10 words for positive reviews:
TODO: <Insert top 10 pos words>

Top 10 words for negative reviews:
TODO: <Insert top 10 neg words>

Challenges

The primary challenge I found with this project was beginning it. It seemed quite intimidating and I was unaware of where to begin. This likely led to the poor code architecture as I wrote code as I thought of it without thinking of separating logic amongst various files and functions. After reviewing the lecture slides and several online resources, I got a better idea of how to approach the assignment and how to calculate the various probabilities required to complete it. I was unsure of how to assign additional weightage to terms that should be considered more strongly than others (i.e. “horrible”, “awful”, “garbage”, “amazing”, “delightful”, “hilarious”).

Weaknesses

There are several weaknesses to my program. First, the architecture is quite terrible and if I had more time I would separate training and classification into different files and have functions. Currently, everything runs in one file in a static environment without any OO design or modularity whatsoever. This also introduces redundancy in my code and makes it more difficult to read and re-use. In addition, besides a few negation terms, my classifier assumes every word is independent from each other in the review. In other words, it doesn’t account for context whatsoever, even though words and sentences make a difference based on where in the review they are. For instance, the first sentence often sums up the review while the body explains in more detail the critic’s thoughts and ideas. My classifier does not account for any of this whatsoever.