Peter Werner (3259306)  
CS 165A Machine Problem 1

**Architecture**

My program consists of the following five classes:

* *NaiveBayesClassifier* is the main driver class. It accepts the files as command line arguments and verifies that they are indeed valid files. It then instantiates a *Classifier* and feeds the training and testing data to the *Classifier* instance. It also prints the accuracy statistics.
* *Classifier* does most of the work. It maintains several hashmaps that are populated during training – storing word counts, category counts, and word counts per category.  
  *Classifier* has methods to train one review or a file full of reviews. The main driver class is responsible for calling the file training method, which then populates the hashmaps.  
  The class also has methods to classify one review or a file full of reviews. The classification methods use the data in the hashmaps to choose which category any given review has the greatest probability of belonging to.  
  *Classifier* uses an instance of the *Parser* class to pre-process the reviews before training and classification (ie: removing stop words, etc.).  
  In calculating the probabilities, *Classifier* uses Baye’s rule:  
   P(category|review) = P(word1,word2,…|category) P(category) / P(word1,word2,…)  
  If any given word exists in the lists of known positive and negative words found in the *WordWeight* class, the probability is weighted using a constant factor.  
  Smoothing is handled very simply. If the training set contained N reviews, the smoothed probability is simply 1 / N. This tested surprisingly well in cross-validation testing.  
  *Classifier* also contains code (now commented out and unused) for post-processing, weighting reviews based on word count.
* *Parser* is used to pre-process the data (reviews). It splits a given review into a set of words (split by whitespace). It also does some manipulation on the word set.  
  It contains methods for:

shifting everything to lowercase, removing punctuation, splitting word-punctuation pairs into isolated words, removing neutral stop words, combining adjacent sets of words into word-pairs, stemming prefixes and suffixes

Ultimately, the only methods that performed well in cross-validation testing were splitting punctuation from words and removing neutral stop words.

* *WordWeight* maintains lists of known positive and negative words (lists taken from Hu & Liu, fully cited in the code comment at the top of the class). Given a word, the class will return a weight (either positive or negative). By default, words neutrally have 0 weight.
* *RandomTest* was my cross-validation helper. It is not used in the main program. It randomly partitions the training set (2/3 used for training, 1/3 used for testing). Given an integer N, it runs the program using random partitions N times; then, it prints out the average accuracy and standard deviation. This allowed to me test various techniques without too much worry of overfitting.

**Preprocessing**

As described in the *Parser* bullet point of the above section, my program splits reviews into sets (such that each word only occurs once) of words (splitting by whitespace). It also removes common stop words and adds word-punctuation pairs as separate words (ie: “great!” 🡪 “great”, “!”).

**Model Building**

Training consists of taking reviews, splitting them into sets of words, and incrementing counters in the following hashmaps:

* Category count: maps category 🡪 number of reviews in category
* Word instance count: maps word 🡪 number of reviews containing word
* Word category count: maps category 🡪 hashmap M  
   where M maps word 🡪 number of reviews in category containing word

This data is all we need to calculate the probability of any given review existing in any given category in constant time.

**Results**

Using the training and testing sets provided, training takes 1 second, testing takes 1 second, and the classification is 83.4% accurate.  
Using my *RandomTest* utility class and 100 random tests, the average accuracy is 87.5% with standard deviation of 1.17%.  
  
The top 10 most influential features in each category were as follows:

|  |  |
| --- | --- |
| Positive | Negative |
| Space, | YOU |
| Personally, | team, |
| really | /><br |
| ways, | sounding |
| only | like |
| Care | capture |
| film, | eh? |
| film. | walking |
| instance. | Thanks |
| Chacha | can |

*\* influence judged by highest P(category|feature)*

**Challenges**

A few of the challenges I faced:

* Best smoothing. I tried various smoothing formulas but ultimately (and surprisingly) I got the best results simply using probability 1 / N (where N is the number of reviews in training).
* Testing. I tried to avoid overfitting by using my *RandomTest* utility to eliminate overfitting and flukes as much as I could (using randomness, the law of large numbers, and by looking at standard deviation as well as average accuracy).

**Weaknesses**

The biggest weakness of my program is that it does nothing to consider ordering and position of words or length of reviews. It is as naïve as a naïve classifier gets. Possible solutions include: weighting probabilities based on some word positioning pattern and more intelligently weighting words based on frequency (think TF-IDF).