Cody Heffner
Project 4
CS 453 Winter 2015

Text Document Classification Using a Multinomial Naive Bayes Model

Introduction

This report details the implementation choices and experimental results of my Multinomial Naive Bayes (MNB) classifier. The assignment was to implement and apply the MNB model for text document classification and to perform feature selection on the vocabulary of a given document collection. The specifications were completed, and this report summarizes the various ways that features were implemented (see **Implementation Choices**) and the results of performing 5-fold cross validation accuracy tests across five feature (vocabulary) size values (see **Experimental Results**).

Implementation Choices

My implementation of the MNB classifier relied heavily on the use of Java's LinkedHashMap class; I used it in nearly all of my data structures. I chose to use this class because it is extremely good at finding random individual elements regardless of where they are or how they are ordered and because it is also extremely good at iterating through the structure thanks to its hashing and linked-list properties. For example, I used a LinkedHashMap to store the vocabulary and the vocabulary elements' corresponding Information Gain (IG) values. I was able to swiftly check if individual words were (or were not) in the vocabulary, but I was also able to swiftly iterate over the vocabulary to find the M highest-scoring IG words.

In implementing previous projects in this class, I used much simpler data structures such as plain arrays to keep it simple, reduce memory overhead, and was still able to access elements instantly through the use of clever macros. I found that performance was pretty good (not particularly better or worse than using Java's LinkedHashMap for this project), but the coding was a *nightmare*. By placing the burden of storing data on Java instead of myself, I found that this project was significantly simpler to create.

Although not mentioned in the project specifications, I chose to stem words because I felt it would help improve the ability of the MNB classifier to recognize similar documents.I found this cut my vocabulary size down from around 95,000 elements to about 85,000 elements (this number varies since the training documents chosen varies randomly).

The formula I used to calculate the probability of a document *d* being in class *c* is as follows:

$$P(d|c) = \sum_{c \in C} \sum_{w \in d} tf_{w,d} \times log_2(P(w|c))$$

where $tf_{w,d}$ is the term frequency of *w* in *d*, and *P(w|c)* is the Laplacian smoothed estimate:

$$P(w|c) = \frac{tf_{w,c} + 1}{|c| + |V|}$$

where $tf_{w,c}$ is the term frequency of *w* in *c*, |*c*| is the total term frequency in the class, and |*V*| is the size of the vocabulary. Note: |*c*| is not the count of unique words in the class.

Experimental Results

To measure accuracy of my MNB classifier, the classifier trained itself using a random 80% of the documents in the collection and reserved the other 20% as “test” documents. During the test phase each of the test documents were classified. Those classifications were compared with their pre-assigned classifications. The ratio of the total number of correctly classified documents over the total number of test documents was the value of a session’s accuracy. I performed 5 sessions (according to the required 5-fold cross validation approach) at each of the five levels of vocabulary size values M, where M ∈ {6200, 12400, 18600, 24800, Full}, then averaged the accuracy values as shown below in figure 1. Vocabulary elements were chosen based on their IG values - only the M highest elements were used.

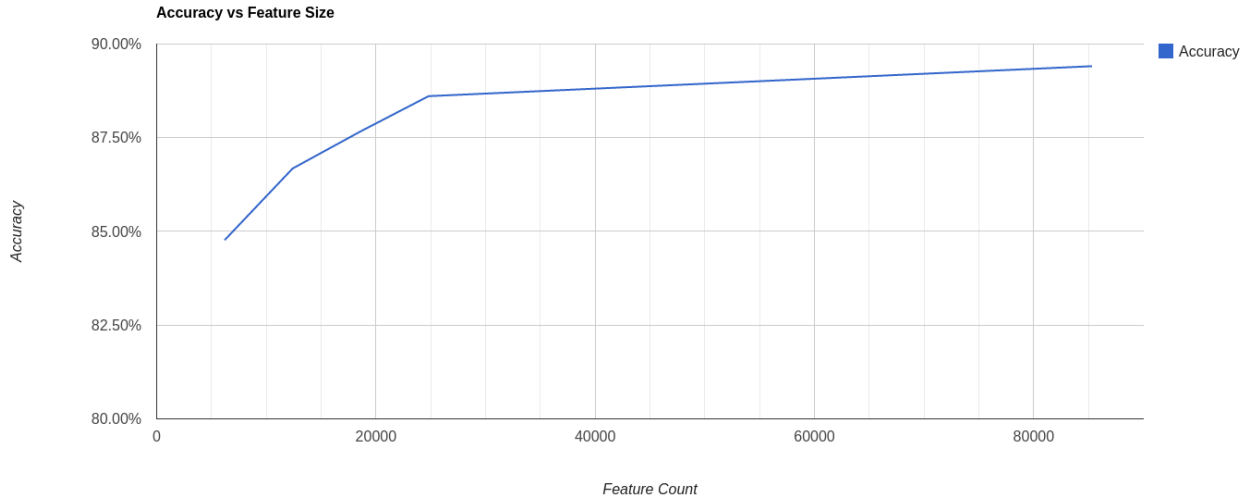

Figure 1 - Accuracy vs Feature Size

The accuracy values were all within a range of 5% for all tests. The lowest accuracy was obtained in the experiment that used the fewest number of features, 6200 elements, with an accuracy of 84.77%. The highest average accuracy was seen during the experiment that used the full vocabulary, 85,000 elements, with an accuracy of 89.41%.

Accuracy drops off quickly as the classifier begins using a smaller feature subset. I ran a few tests with extremely small feature counts and received terrible accuracies. When I ran the classifier with M=10, my classifier was only 38.80% accurate. This goes to show how much of an impact even a single word can have on the accuracy of the classifier. There are 85,000 words in the vocabulary, and by using only ten words, the classifier can put half the documents in their correct category. With ten different categories to choose from, this is not an insignificant result.

Execution time was constant in relation to feature size. This was because in order to determine which elements to include in the feature subset, every word count of every document in every class needed to be recorded. Each word needed to be assigned a value, then the feature subset was chosen based on the highest M values. The M most valuable words are not known before the classifier is trained. As seen in figure 2, the majority of my program's run time is the training time. As a result, total runtime was not affected significantly by feature size.

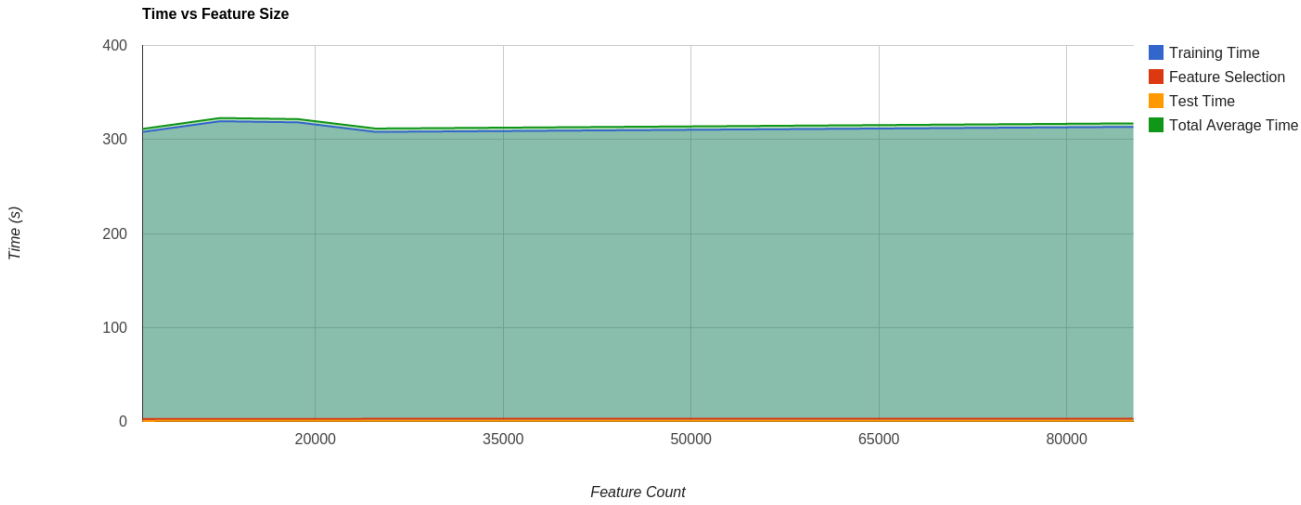

Figure 2 - Execution Time vs Feature Size

The values for the time spent training the document compared as a function of the feature count is shown in the following table:

Feature Count	6200	12400	18600	24800	85000
Training Time (s)	308	319	318	308	313

As you can see, there is no discernable pattern. This is because the classifier must ignore feature selection until the classifier is already trained. This is not particularly helpful or interesting.

A more interesting set of data is the time spent performing the feature selection as a function of the feature count. Feature selection time is proportional to the feature count, as expected. This is shown in the following table:

Feature Count	6200	12400	18600	24800	85000
Feature Selection Time (s)	2.65	2.74	2.73	2.86	2.94

The most interesting data, I discovered regarding feature selection was the average test time for 2000 documents as a function of feature count. If one wishes to use a text document classifier, training time and feature selection time will most likely be a non-issue - these are tasks that need to be performed once and then not again until you restart the classifier. The most important time for a classifier is how long it takes to classify a document.

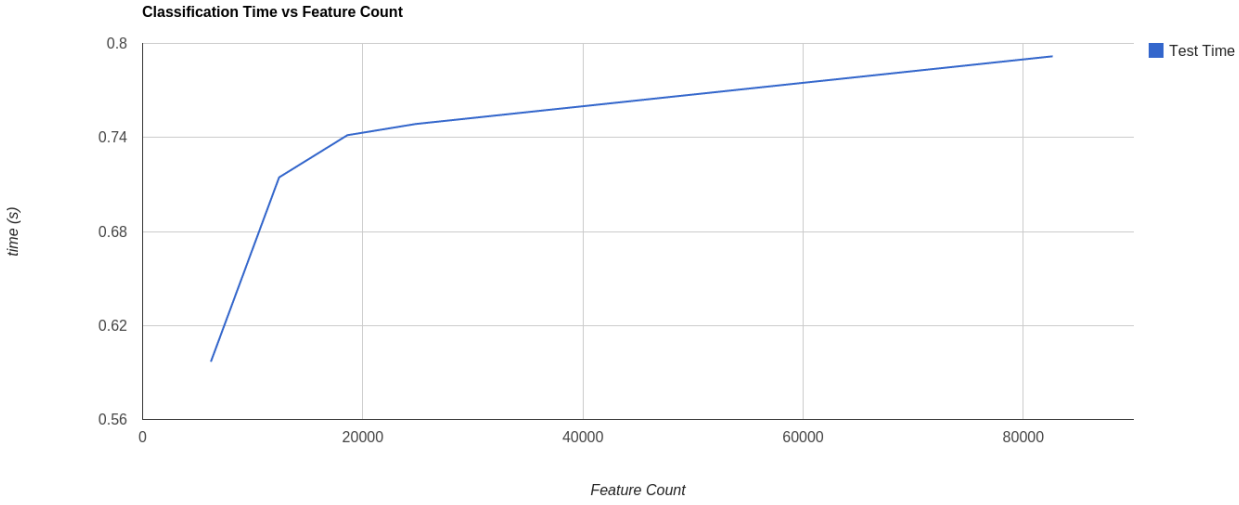

Figure 3 - Classification Time vs Feature Count

Figure 3 shows that the classifier can perform much faster when using less features - when using 6500 features, execution time was nearly 25% faster! However, as figure 1 shows, accuracy is also a function of feature count. So how does one balance accuracy vs. classification time? Behold, figure 4!

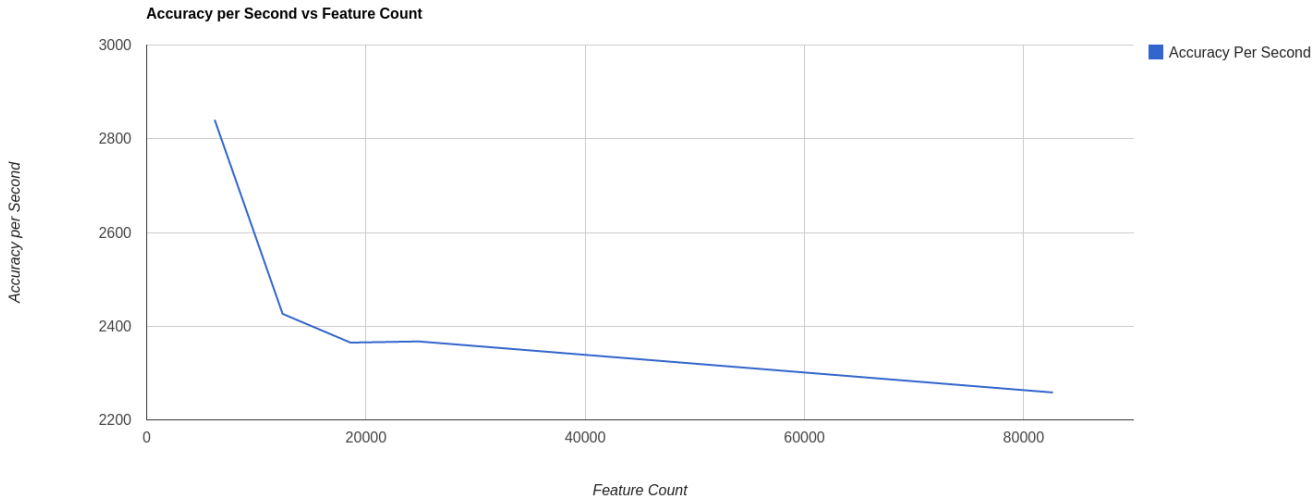

Figure 4 - Accuracy per Second as a function of Feature Count

Figure 4 scores each data set based on accuracy *and* execution time. It is desirous to have classifiers that perform accurately (the higher the accuracy, the better - accuracy is proportional to classifier "score") and fast (the lower the execution time, the better - time is inversely proportional to classifier score). To calculate a final score for the classifier using specific feature counts, I divided the accuracy values by the time spent classifying per document. Figure 4 shows that the classifier is able to classify more correct documents per second when using a smaller feature count.

There are at least two things to consider when scoring a classifier like this. First: what is an incorrectly-classified document worth? Second: when using a score like this, classifiers can simply spend one cycle to classify a document in the first category it sees and achieve a ridiculously high score. To genuinely use a scoring method like this, one would need to subtract the weight of an incorrectly-classified document multiplied by the rate of incorrectly-classified documents. I hypothesize that an equation like this would do the trick:

$$score = \frac{A}{t} - (1 - A) \times v$$

where *A* is the accuracy of the classifier, *t* is the time spent classifying per document, and *v* is the inconvenience caused by an incorrectly classified document. A typical value for *v* could be ~(*1/t)* to keep both parts of the equation equally weighted.

Conclusion

Another experiment that could be performed is the accuracy rate of a stemming vs non-stemming MNB classifier. In my implementation section, I stated that I felt stemming would be beneficial to the classifier to recognize similar documents. By stemming all words, relationships can be drawn where there were none previously. While there is specific basis on which to change my implementation, an area for experimentation would be the event where similar words stem to the same value, in which case a false positive would occur. For example, the words “policy” and “police” stem to the same value. These words are not particularly related but the classifier will think they are the same word. This has an effect on the way the classifier categorizes documents. The classification process could prove to be an excellent arena in which to judge stemming functions - one could split the exact same documents into the training and testing sets, stem and not stem the words, classify the documents, then compare the accuracy of the stemming vs. non-stemming classifier.

There are many areas of my classifier that could have been improved to provide a performance or accuracy boost. All in all, I wanted to keep this project simple to get a feel for how the basics of text classification work. Were I to build on this project, I would first focus on the formula used to classify a document. I feel that refining it to implement more advanced techniques in information retrieval could provide a significant boost to the classification accuracy.