Assignment #1, Amazon Review Classification

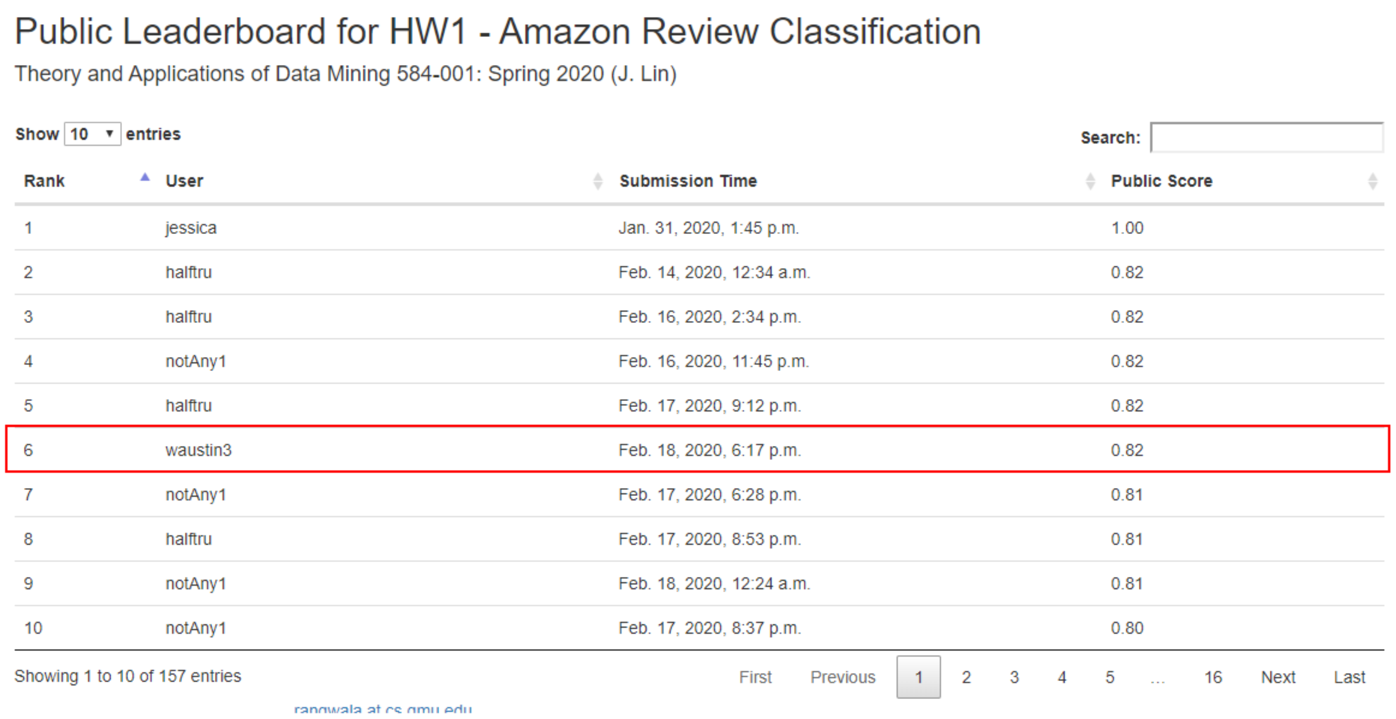
CS 584: Data Mining
Due February 18th, 2020
Submitted by William Austin

1. Miner Website Credentials

Registered Miner Username: **waustin3**

2. Rank & Accuracy of Submission

My third and final Miner submission for this assignment, on February 18th, had an accuracy of **0.82**. At the time of this writing, this value is tied for the best in the class with 2 other users (halftru and notAny1).



Public Leaderboard for HW1 - Amazon Review Classification

Theory and Applications of Data Mining 584-001: Spring 2020 (J. Lin)

Show 10 entries

Search:

Rank	User	Submission Time	Public Score
1	jessica	Jan. 31, 2020, 1:45 p.m.	1.00
2	halftru	Feb. 14, 2020, 12:34 a.m.	0.82
3	halftru	Feb. 16, 2020, 2:34 p.m.	0.82
4	notAny1	Feb. 16, 2020, 11:45 p.m.	0.82
5	halftru	Feb. 17, 2020, 9:12 p.m.	0.82
6	waustin3	Feb. 18, 2020, 6:17 p.m.	0.82
7	notAny1	Feb. 17, 2020, 6:28 p.m.	0.81
8	halftru	Feb. 17, 2020, 8:53 p.m.	0.81
9	notAny1	Feb. 18, 2020, 12:24 a.m.	0.81
10	notAny1	Feb. 17, 2020, 8:37 p.m.	0.80

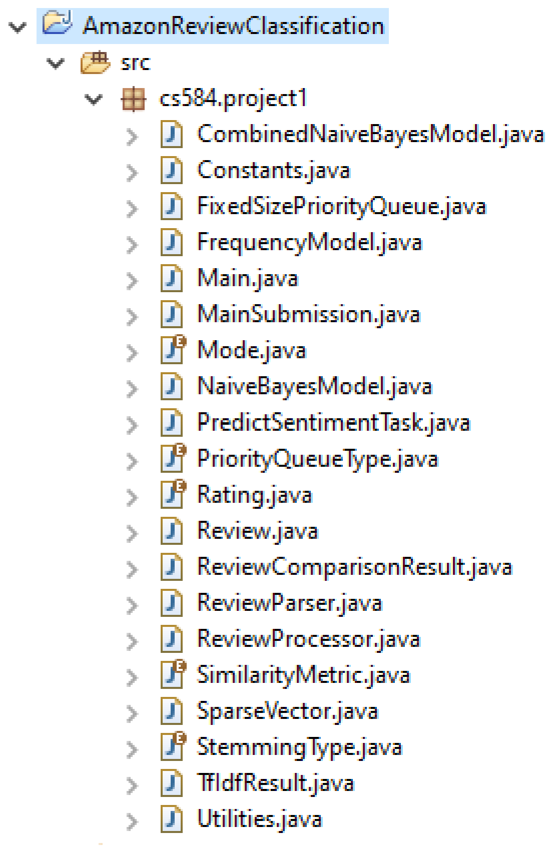
Showing 1 to 10 of 157 entries

My other 2 older submissions for the project were:

- Submission #2. February 8th, Accuracy = 0.79
- Submission #1. February 8th, Accuracy = 0.69

3. Instructions for Running the Code

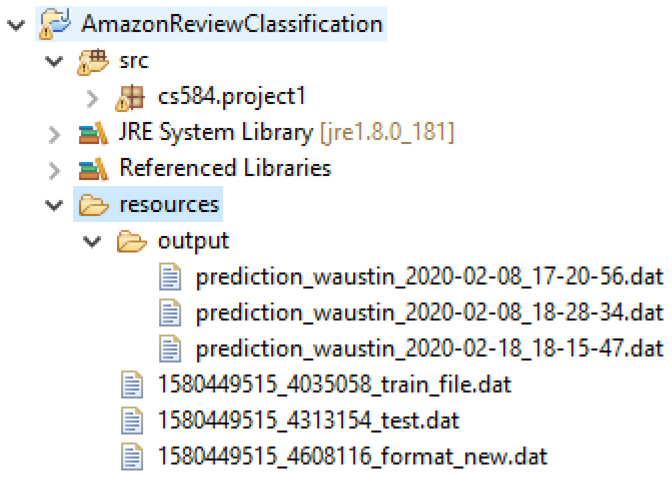
I implemented my application in Java, and my JDK version is 1.8. I chose Java because it's the language I am most comfortable with, and I had never implemented a project like this before in Java. The project structure is typical for Java applications, with the **`src/`** directory being the class path directory for the code. All my code is in the **`cs584.project1`** package. The Eclipse structure is shown below:



```
AmazonReviewClassification
  src
    cs584.project1
      CombinedNaiveBayesModel.java
      Constants.java
      FixedSizePriorityQueue.java
      FrequencyModel.java
      Main.java
      MainSubmission.java
      Mode.java
      NaiveBayesModel.java
      PredictSentimentTask.java
      PriorityQueueType.java
      Rating.java
      Review.java
      ReviewComparisonResult.java
      ReviewParser.java
      ReviewProcessor.java
      SimilarityMetric.java
      SparseVector.java
      StemmingType.java
      TfIdfResult.java
      Utilities.java
```

To look at the code, start with **MainSubmission.java** and **Constants.java**.

There is also a resources directory that contains the input files, output files, and required library initialization files. The image below shows this directory, with the **`src/`** directory minimized.



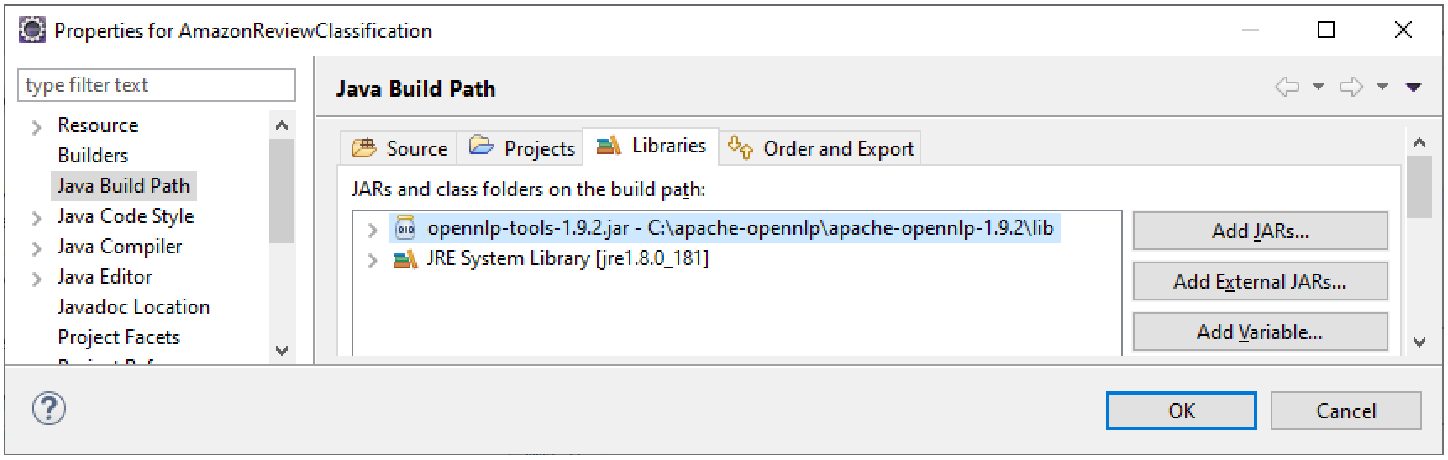
```
AmazonReviewClassification
  src
    cs584.project1
  JRE System Library [jre1.8.0_181]
  Referenced Libraries
  resources
    output
      prediction_waustin_2020-02-08_17-20-56.dat
      prediction_waustin_2020-02-08_18-28-34.dat
      prediction_waustin_2020-02-18_18-15-47.dat
    1580449515_4035058_train_file.dat
    1580449515_4313154_test.dat
    1580449515_4608116_format_new.dat
```

Note that in my submission, I am including the contents of my resources/ directory, with the exception of the training, test, and original format files, as these are large, and have already been provided.

As a side note, I had originally included the models and dependencies for the Stanford NLP API so that I could experiment with feature reduction techniques based on part-of-speech tagging, but I was not getting good results, so I abandoned that effort, and the submitted code does not require this dependency.

Therefore, the only dependency that needs to be set up for the project is for Apache OpenNLP, version 1.9.2. Because the jar filess are quite large, I am omitting them from the submission, but it is quite easy to set up. The steps are:

1. Download it from here: https://opennlp.apache.org/download.html
2. Extract the jar file from the zip and add the **`lib/`** directory to the classpath, as shown here:


Properties for AmazonReviewClassification

Java Build Path

JARs and class folders on the build path:

opennlp-tools-1.9.2.jar - C:\apache-opennlp\apache-opennlp-1.9.2\lib
JRE System Library [jre1.8.0_181]

After these steps, the project will compile and be ready for execution.

Note that my code for this project is also available on GitHub. The URL is:

https://github.com/william-r-austin/AmazonReviewClassification

4. Approach and Parameter Selection

I spent significant amounts of time on this project tuning parameters and experimenting with different methods that may yield improved accuracy. I'll highlight the main techniques that I experimented with, mentioning what did and did not work, and if applicable, where to find the implementation in my code.

1. **Support for K-Fold Cross Validation**
   The first thing I did as I was starting this project was to parse the training and test files and build out some of the necessary classes for the computations, like the **`Review`** class. However, I immediately noticed that I could only do a limited amount of testing against Miner, so I added support for K-Fold cross validation. This would allow me to test as much as possible against the training set, in a statistically valid way. I implemented this by associating each sample with a "fold" and then being careful to leave out the correct folds during training. This code to leave out the correct fold during training, (but not testing) is in **PredictSentimentTask.java.**

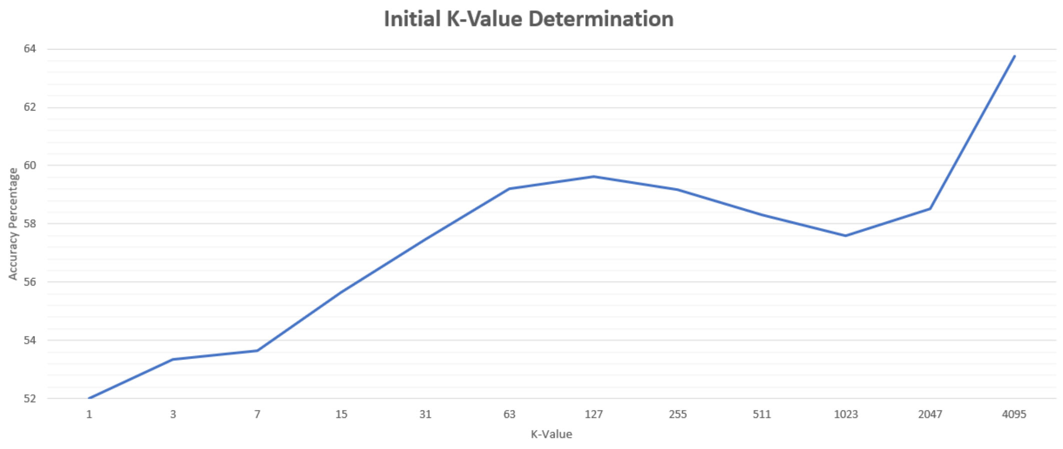
```
if(Mode.CROSS_VALIDATION.equals(mode)) {
    Integer predictDataGroupId = predictReview.getDataGroupId();
    if(predictDataGroupId.equals(compareReview.getDataGroupId())) {
        validComparison = false;
    }
}
```

This allows us to keep our training set together, but not risk training against the same sample.

2. **Implement TF-IDF, using a sparse data structure.**
   As Dr. Lin pointed out, it is not feasible to store a full feature matrix for this problem on normal desktop hardware. Indeed, the first time I tried implementing TF-IDF, I immediately encountered an OutOfMemory error, so I was forced to write a sparse data structure instead in the **SparseVector** class. Fortunately, this trick worked out well, and I was able to use this implementation for the remainder of the assignment.

   My TF-IDF implementation is mostly contained in the **FrequencyModel** class, and the main computation is in the **computeWeightVector()** method.

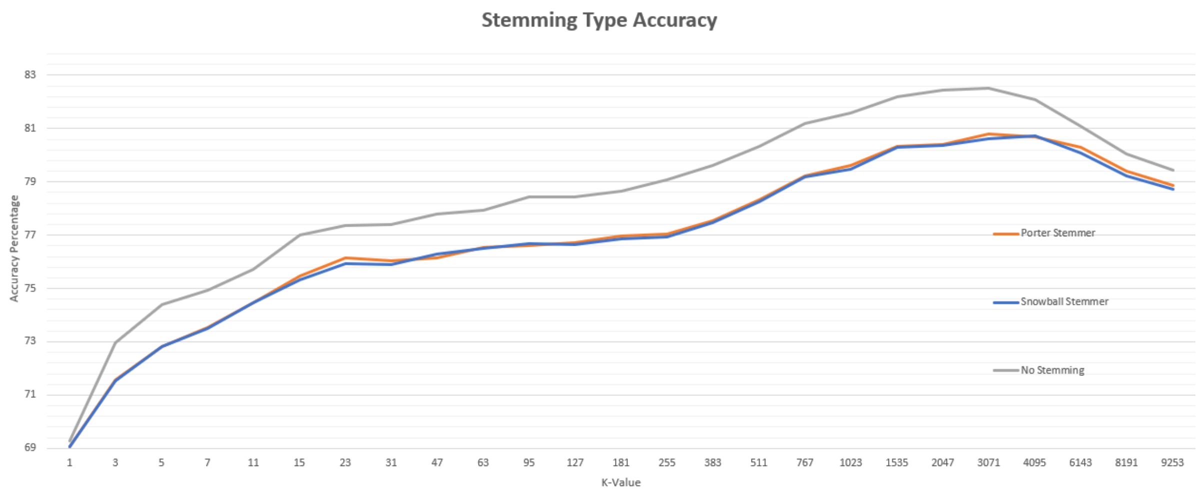
3. **Optimize K-Value for K Nearest Neighbor Algorithm**
   After achieving a basic implementation for TF-IDF, with K = 1, I naturally decided to experiment with different values of K to improve my accuracy. Naturally, I choose roughly exponential values for K because we are getting less precise matching information as K increases, so we must increase K by larger amounts to see a noticeable trend that (hopefully) does not contain a lot of noise. The graph below shows my initial results:



   After realizing how tedious this testing would be, the next optimization that I made was to be able to compute the accuracies for multiple values of K in the same execution by fully sorting the matches and then repeatedly looking at the top-K values. Given that the expensive part of this operation is the matching and sorting of the results, making predictions for numerous values of K is actually very cheap and did not even noticeably slow down the runtime. This feature is made possible by the **FixedSizePriorityQueue** class (based on **TreeSet**), and the **ReviewComparisonResult** bean. As we will see below, this allows us to get a much better picture of how to choose optimal K.

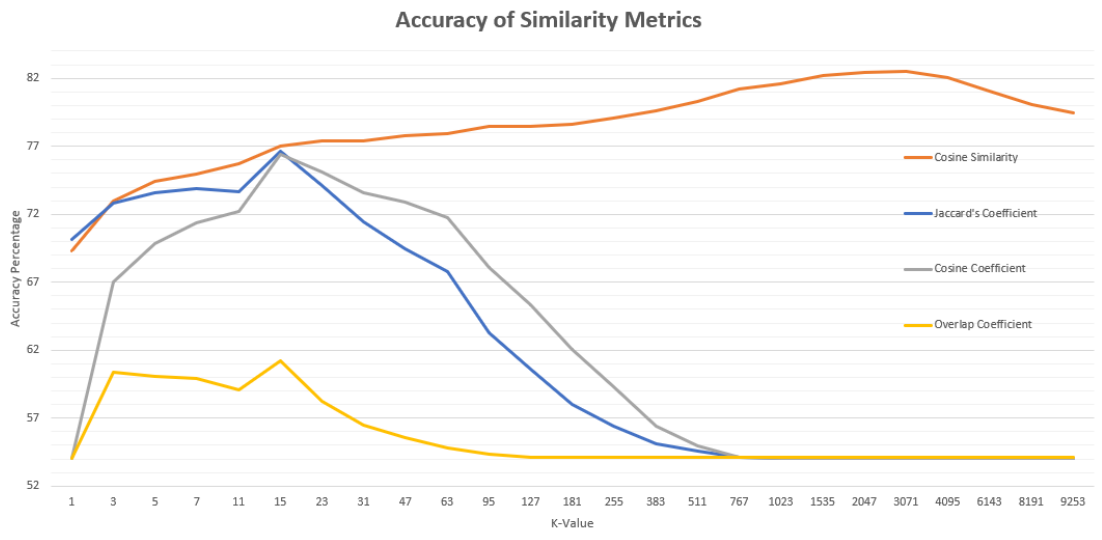
4. **Choose Stemming Technique**
Initially, I was simply splitting words on whitespace and creating tokens. However, non-letter symbols do not add much value for sentiment prediction, so my final code uses this regex:
```
String[] initialTokens = input.replaceAll("[^a-zA-Z' ]", " ").toLowerCase().trim().split("\\s+");
```
This removes all non-letter characters except single quotes, converts to lower case and splits the result at the whitespace. This caused my accuracy to jump to over 68% as compared to the 64% that we saw in the graph above.

Next, I used the Apache OpenNLP implementation of the Porter and Snowball stemmers to try to improve the accuracy further. However, I was unable to get better results using stemming than without, and my final submission does not use stemming – only the tokenization mentioned above. My accuracy chart for different stemming methods is shown here:



Note that all stemming and tokenization code is in my **ReviewParser** class.

5. **Explore Similarity Measures**
To supplement the TF-IDF method, I also implemented the similarity measures introduced in class. These methods are located in the **Utilities** class, and the active method can be changed by updating the **Constants** class. As expected, the TF-IDF method performed the best. Given that this method reduces each sample to a floating-point vector, it makes sense that it would be somewhat more accurate than the other methods, that use set operations instead. Note that these sets can be represented as boolean vectors as well. In addition, these methods do not incorporate the IDF (inverse document frequency) portion of the computation, which is valuable context to have for prediction. Below are my results for the different similarity measures:



6. **Balance Positive and Negative Results**

One key insight that I made while testing my code was that all my models seemed to overpredict positive sentiment and underpredict negative sentiment. After digging a bit further, I realized that the bias existed because there are 10000 positive samples in the training set and 8506 negative samples – it's not an even split. This is a problem, specifically for KNN, because each of the training samples carry equal weight, but there are more of one class than the other, so a bias emerges, as we observed. I made the assumption that we want our model to be balanced, so to fix this, I simply computed a weight for each class and used that during our KNN summation. More specifically, each positive match receives a weight of:

$$\frac{Total\ Documents}{2 * Positive\ Documents} = \frac{18506}{2 * 10000} \approx 0.9253$$

And the negative match weight is:

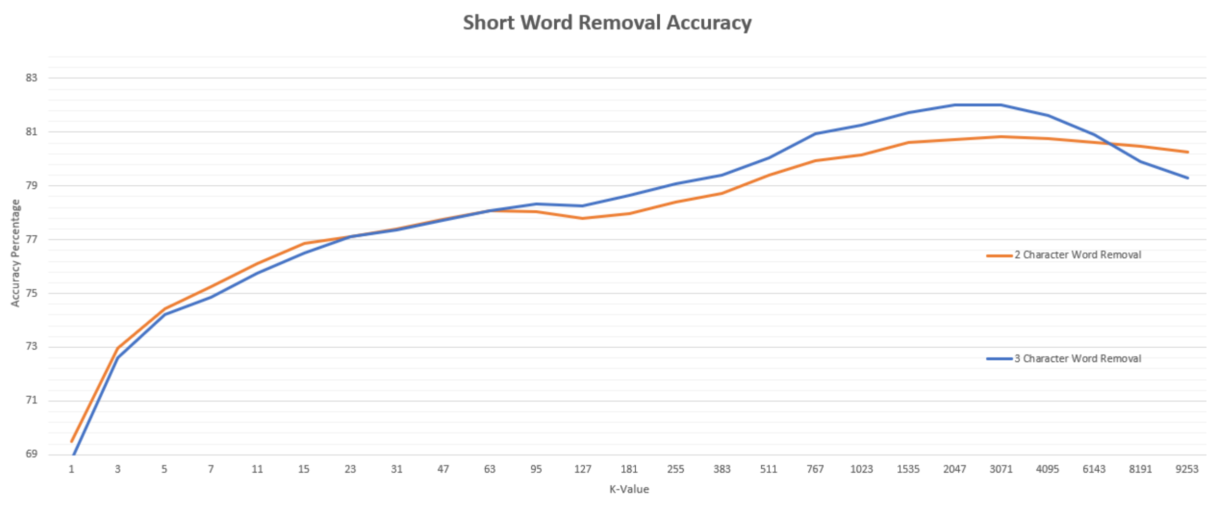
$$\frac{Total\ Documents}{2 * Negative\ Documents} = \frac{18506}{2 * 8506} \approx 1.0878$$

Making this change immediately raised my accuracy to almost 80%.

7. **Dimensionality Reduction**

I made three enhancements to implement dimensionality reduction. These were:

- Remove all words that are less than 3 or 4 characters long. The intuition behind this is that 1 and 2 character words like "a", "to", "of", "at", etc. do not contribute meaningful information for sentiment prediction. However, I found that removing 3 characters like "the" is also beneficial, even though some words like "joy" might be omitted. My test results are below:



By doing this step, we reduce the total number of terms ("features") from **23751** to **22557**. In addition, we see a slight increase to our overall accuracy. The implementation is contained in **ReviewProcessor.filterByTermLength()**.

- The next change I made to reduce the number of features is to remove terms based on their predictive value. To do this, I constructed a Naïve Bayes model that counts how many times each term is used in positive and negative samples, and then removes the term if these numbers match. In that case, the word has no value for sentiment prediction, so we remove it from the term set. This removes about another 10% of the feature set, to take our total feature count from **22557** to **20738**. In addition, our accuracy, is roughly maintained at 82%, as before. This is implemented in **ReviewProcessor.filterByPredictiveValue()**.

- Lastly, I decided to remove all terms from the feature set that were only used once in the training set. The rationale for this is that words used very infrequently should not be used for prediction because they may be outliers. To accomplish this, we instantiate a dummy **FrequencyModel** and use it to remove all the terms with a frequency of 1. This is by far the most effective method we have for dimensionality reduction and it takes our feature set size from **20738** to **10692**. From our starting feature set size, this is over 50% smaller, and in addition, our accuracy is again maintained! This is implemented in **ReviewProcessor.filterByFrequency()**.

8. Techniques that did not work

Briefly, I'll outline a few of the failed approaches that I tried throught the course of this project

- KNN Weighting. This would cause the closer matches to have a higher weight compared to more distant matches. However, I could not get the scaling to work, and my results were poor.

- Part-Of-Speech-Tagging. I installed the models for both the Stanford NLP tagger and the Apache OpenNLP tagger, but the results seemed inconsistent, and after toying with a few examples that mislabled verbs and adjectives, I abandoned the effort.
- “Bottom K-Means”. In theory, if we sort all matches for a particular sample to predict, looking at the top-K matches should give the correct class, and looking at the *bottom-K* matches, should yield the opposite class. This worked well enough, but there are enough samples near the bottom with no similarity, so that the fidelity is much less, and sticking with traditional KNN gave better results.

5. Algorithm Efficiency and Runtime

The main improvement that I made to improve performance is to farm out work to multiple threads so that the prediction tasks can be done in parallel. This can be accomplished because each thread gets a copy of the training set and model, as it is pre-computed, and if we are careful, we can combine the results from each thread into a single result for the whole data set. This task is made much easier by the utilities provided in the **java.util.concurrent.*** package. Namely, the **PredictSentimentTask** implements the **Callable** interface and allows us to manage the thread pool independently of this implementation.

On my laptop (Intel Core i5 with 2 cores), this change dropped the run time from about 10 minutes to around 7 minutes. With the dimensionality reduction changes, I was consistently running cross validation on the training set in about 6 minutes.

Overall, where n is the total number of input samples, and k is the total number of terms, reading the runtime is $O(n)$ for reading the input, $O(k)$ for dimensionality reduction because we need to go through all the terms to check their frequency and predictive value. However, building the frequency model is $O(nk)$ because some samples may have every term included. During the prediction phase, we must sort all input samples (implemented by TreeSet) which is $O(n \log n)$, so our full computation is $O(k\ m\ n \log n)$.

There a few improvements that I made earlier that also contribute to the application performace as well. Namely,

1. Obviously, implementing the **SparseVector** class to model term frequencies without having a $O(n)$ space requirement made these compuations possible.
2. The dimensionality reduction techniques obviously contribute to performance as well because there are fewer operations to perform during the prediction stage.
3. Computing KNN predictions for multiple values of K without recomputing the underlying information everytime. As expected, this code is in the **PredictSentimentTask** class.
4. Running K-Fold Cross Validation without reloading the data set and building a new model for each trial. This is done by making the models specific to a particular group and combining them dynamically.