Predicting Movie Ratings Based On Reviews

Josh Balingit, Luc Chen, Hengyuan (David) Liu, Kathy Mo, Ka Wai Sit

University of California, Davis

STA 141C

March 23, 2023

Table of Contents

1 Abstract

For this project, we have 2000 training and 500 test observations, containing information on movie reviews (words) and rating labels (1 for positive and 0 for negative). Our main goal is to effectively implement algorithms that can predict whether or not a movie receives a positive rating based on its review. After constructing a standardized uni-gram TF-IDF matrix, we had roughly 17,000 stem features. To improve efficiency, we decided to reduce the dimensionality of our problem. Our initial approach to dimension reduction was through PCA via the power method; however, due to running time issues, we ultimately reduced dimensionality by selecting the top n stem terms from positive and negative reviews. We then implemented logistic regression through gradient descent. When conducting k-folds cross-validation for this algorithm, we found that $n = 1000$ returned the lowest cross-validation error rate of 5.35%. This process took 20 minutes after using parallel computing. After running logistic gradient descent with $n = 1000$, our team was left with a final test error rate of 20%. This process took 1 to 2 minutes after implementing step-halving, setting the appropriate initial guess, and selecting the largest standard tolerance of 0.001. Using K-NN and cross-validation, we identified $n = 100$ and $k = 8$ as optimal hyperparameters with the lowest cross-validation error of 23.70%. When applied to the test set, this yielded an error of 26.20% in 18 seconds using parallel computing. To speed up the process of calculating Euclidean distances, we used the default R function norm(), leveraging LAPACK written in a lower-level programming language (Anderson et al.).

2 Introduction and Motivation

When releasing blockbuster films, producers often want to know how it is being received by test audiences. This is because an overall negative reception may require them to expend additional resources on editing movie scenes and scheduling interview events in order to avoid controversy and generate excitement. However, manually reading through thousands of reviews and determining whether each one is "positive" or "negative" can be expensive and time-consuming. It also leaves room for a lot of human error given the sheer scale of reading and documentation involved. To assist producers with this dilemma, our team has focused research on effectively implementing machine learning algorithms to predict movie ratings based on reviews. However, this does not mean that things have become any easier, as there are numerous factors to consider when developing algorithms from scratch. This includes, but is not limited to, iterative initialization and parallel computing. This report describes, in detail, our encounters with these types of technical details as we implemented two major classification algorithms: logistic regression and k-nearest neighbors.

3 Research Question

How can we effectively implement algorithms to predict whether or not a movie receives a positive rating based on its review?

4 Data Description

The original dataset utilized in this project comprises 50,000 IMDB reviews, which are equally divided into 25,000 samples for the training and test sets. The labels are distributed evenly between positive and negative reviews (25k each). However, only a random subset of this dataset will be used, consisting of 2,000 training set samples and 500 test set samples, with a total of 2,500 movie reviews. The positive and negative reviews are balanced in this subset. Additionally, to mitigate the influence of correlated ratings, no more than 30 reviews are included for each movie in the entire collection. The train and test sets contain disjoint sets of movies, negating any significant performance improvement resulting from the memorization of unique terms associated with specific movies and their observed labels. Negative reviews have a score less than or equal to 4 out of 10, while positive reviews have a score greater than or equal to 7 out of 10. Reviews with neutral ratings are not included in the train/test sets. The dataset includes information on unique IMDB IDs, movie reviews, and ratings. For the purpose of this project, a binary column is created, with negative reviews labeled as 0 (rating of 4 or below) and positive reviews as 1 (rating of 7 or above).

5 Exploratory Analysis

5.1 Word Length Histograms

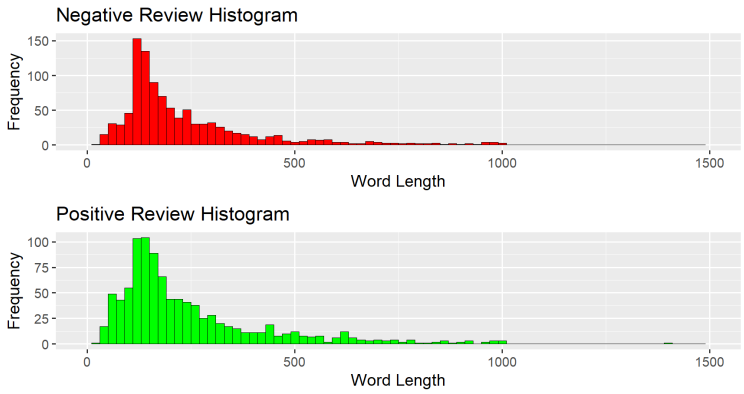

Figure 1: Word Length Histograms for Positive and Negative Reviews

The distributions of word length for both positive and negative reviews were plotted, disregarding typos and special characters. The resulting histograms were found to be skewed to receive some outlier positive reviews having word lengths approaching 1500. These unusually lengthy reviews may have been written by professional movie reviewers or promoters, who may have been required to adhere to a word count or provide a comprehensive review highlighting all aspects of the movie. The mean word length for negative reviews was found to be 237, while for positive reviews it was 245. This similarity in mean word length can be explained by the fact that people are generally more willing to write longer reviews expressing their enjoyment of a movie, while they may be less motivated to write longer reviews to express dissatisfaction, or may not even bother to write a review for a movie they found unsatisfactory.

However, the histograms of the positive and negative word lengths are very similar. We used the raw data since cleaning every review would be time inefficient and have a lot of room for human error. We initially attempted to clean the data, however, it was too time-consuming. Nonetheless, through our attempts, we noticed several types of special characters in both negative and positive reviews, such as (or), $< br/ >< br/ >$, and $< U + 0085 >$, which makes the review become longer than expected.

The histogram uses an algorithm that splits the reviews based on spaces. This means that special characters were counted as words, along with any stray letters. For example. reviews may have typos such as "tim e" which would count as two words. Because of this and many other factors in the reviews, it is simply not possible to fully clean the data set nor consider every possible typo and special character. Perhaps after cleaning, the histograms would change, but it is not possible to do so since there is a lot of room for human error. If the typos and special characters were not considered words, the average review length for negative and positive reviews would become lower.

There are three possibilities for the histograms. Typos and special characters are equally prevalent in both reviews, which would mean the average positive review length is similar to the negative review length, and therefore positive and negative reviewers are equally passionate. The second possibility is that positive reviews have more typos and special characters, which would mean they were less passionate. Since these reviewers liked the movie, they possibly wanted to say some kind words about the movie and not much more, while negative reviewers needed to describe exactly why they disliked the movie. The third possibility is that negative reviews have more special characters and typos, possibly from their anger.

Nonetheless, since the histogram for positive and negative reviews are basically the same, we will not use review word length to differentiate between positive and negative reviews.

5.2 Word Clouds

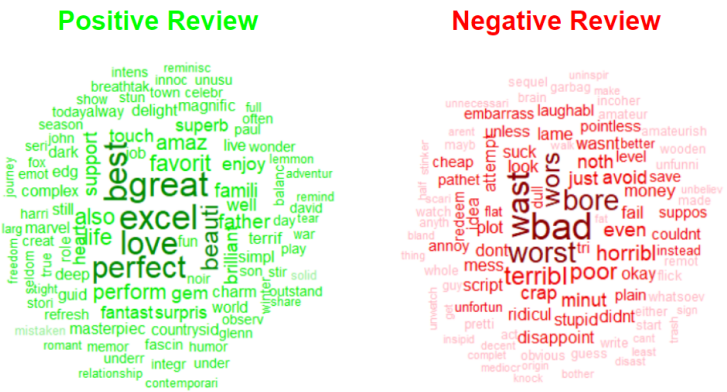

Figure 2: Word Clouds for Positive and Negative Reviews

In addition to examining word length, we generated separate word clouds, with the green one on the left representing positive reviews and the red one on the right representing negative reviews. Before discussing the meaning behind these word clouds, we must first describe the generative process.

It began with us pre-processing all of the training reviews. In our case, pre-processing involved removing punctuations, numbers, and stop words. This comes from the assumption that these things do not contribute as much meaning and will not help distinguish between positive and negative reviews. It is also worth noting that we are considering "film", "films", "movie", and "movies" as stop words. This is because our data is movie-related, which makes it reasonable to assume that the aforementioned terms will appear across most positive and negative reviews. With these terms being prevalent in both groups, we can further assume that they will not be helpful in distinguishing between positive from negative and vice versa.

We then proceeded to stem each review. Our team did initially experiment with lemmatization; however, it often led to many setbacks. For instance, when lemmatizing "amaze", "amazing", and "amazingly", we'd hope that all three terms would be converted to the same term. Unfortunately, only "amaze" and "amazing" were converted to "amaze" while "amazingly" remained as "amazingly". Although lemmatization would've led to more interpretable terms, our main goal is to effectively implement an algorithm that returns good predictions. As such, stemming still served as a viable option to summarize raw text review. We will also later illustrate in this section how stemming does not completely remove interpretability and how it can still yield fairly comprehensible results.

After this, we generated a uni-gram term-frequency inverse-document-frequency (TF-IDF) matrix. The reason why we implemented TF-IDF weighting is that we wanted to add more weight to "exclusive" stem terms that either appeared often in positive reviews or often in negative reviews. TF-IDF weighting also assigns less weight to stem terms that are prevalent across both types of reviews. Ultimately, this emphasizes any underlying stem differences between positive and negative reviews.

We then scaled this matrix so that each column had mean 0 and standard deviation 1. The details surrounding this will be further elaborated on in later sections, but it is primarily to have the proper matrix setup for principal component analysis (PCA). It has also been empirically shown to speed up convergence for iterative algorithms such as gradient descent (Balasubramanian).

Using this standardized matrix, we then generated the word clouds shown in Figure 2. With these word clouds, the color simply indicates which word cloud belongs to which rating group. As mentioned before, the green word cloud represents positive reviews while the red word cloud represents negative reviews. Meanwhile, the hue, size, and location of each stem represent its prominence. For instance, dark, large stem terms towards the center such as "excel", "great", and "love" are the most prominent terms in positive reviews. Similarly, we can interpret "bad", "worst", and "bore" as the most prominent terms in negative reviews.

This provides us with a lot of powerful insight because it not only highlights how distinct positive and negative reviews are but how intuitive the division is. In retrospect, many of the prominent terms revealed by these word clouds are not just terms that we'd used to describe movie reviews but positive and negative life experiences overall. For instance, when landing a job, we often say that it was "excellent", "great", or "lovely". Meanwhile, when failing a class, we often say that it was "bad", "the worst", or "boring".

6 Dimension Reduction

6.1 Principal Component Analysis (PCA)

Despite the work done to explore and understand our dataset, we still faced one significant problem: dimensionality. As mentioned earlier, we transformed 2000 raw text reviews into a uni-gram TF-IDF matrix; however, this matrix has roughly 17,000 columns, with each one representing a unique stem/feature. This was a major issue because the default R function glm() was unable to fit a model within a reasonable time frame. We know that it would take more than 5 minutes given that we waited that long and received no results. In order for us to implement an efficient algorithm that could return results within a reasonable time frame, dimension reduction ultimately became our next priority.

Now, our initial approach to dimension reduction was through PCA. With PCA, it is crucial that each column of the data matrix is, at the very least, centered at 0 in order to use singular value decomposition (SVD) to extract singular values and principal components. Given how we have already standardized our uni-gram TF-IDF matrix, we know that we satisfy this condition and can proceed with SVD. However, a major issue occurred when using the default R function svd(). Similar to glm(), we know that svd() would take more than 5 minutes given that we waited that long and received no results. Like before, this is likely due to the sheer scale of our data matrix.

And so, our team decided to utilize an iterative algorithm known as the power method in order to extract the first set of singular values for our standardized uni-gram TF-IDF matrix. The basic idea behind this algorithm is that we extract the first set of eigenvectors for XX^T where X is our standardized uni-gram TF-IDF matrix. We then use these eigenvectors to obtain the first set of eigenvalues for XX^T. From there, we then take the square root of these eigenvalues. This is because the square root of the eigenvalues for XX^T are the singular values for X.

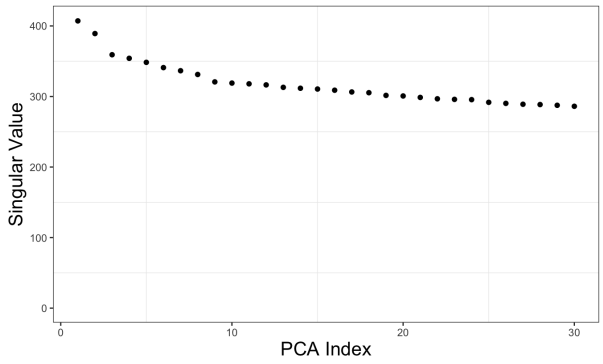

Figure 3: Singular Values for first 30 Principal Components

Figure 3 shows the results of this algorithm, with the x-values representing principal component indices and the y-values representing singular values. As we'd expect, the singular values decrease as we extract more principal components; however, the issue with this particular graph is that we are not seeing any significant dip in the singular values. The reason why we'd like to see a significant dip is that it signifies the lack of spread/information being retained from any following principal component. Therefore, we could simply disregard any principal component that proceeds this significant dip. However, as mentioned before, we are not seeing any significant dip. Instead, we are seeing a significant amount of spread/information being retained from the first 30 principal components.

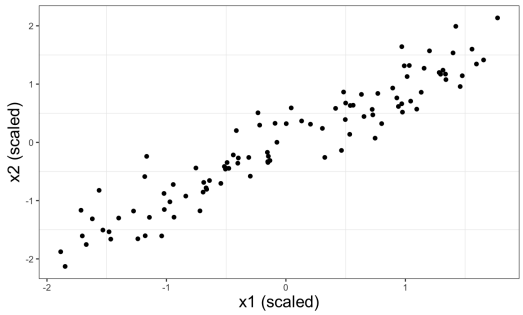

Figure 4:
Scatterplot of Strongly Correlated Data

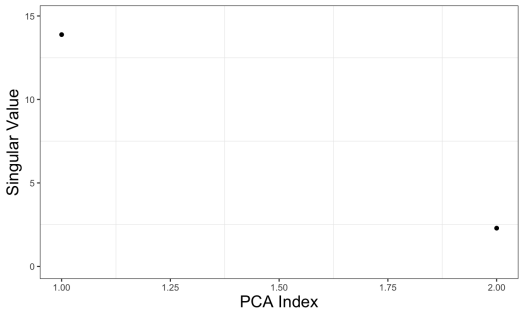

Figure 5:
Singular Values from Strongly Correlated Data

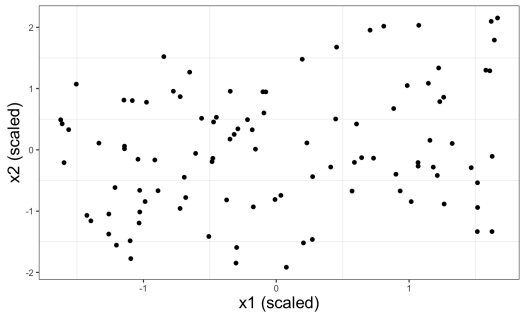

Figure 6:
Scatterplot of Weakly Correlated Data

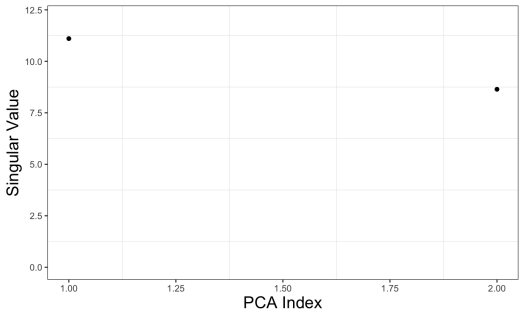

Figure 7:
Singular Values from Weakly Correlated Data

In order to gain some understanding as to why we are not seeing a significant dip, we must turn to our simulation study. As a way of testing our self-defined R function for the power method, we generated two types of datasets. The first dataset has 100 observations where variables x_1 and x_2 are strongly correlated, as shown in Figure 4. When applying our function to this dataset, we see a significant dip between the first and second singular values, as shown in Figure 5. This is to be expected, given the strong correlation prevalent in the first dataset. As we can see in Figure 4, most of the spread/information can be retained along the vector $(1, 1)$, while a small amount is retained along the perpendicular vector $(-1, 1)$.

Meanwhile, the second dataset has 100 observations designed so that variables x_1 and x_2 were weakly correlated, as shown in Figure 6. When applying our function to this dataset, we have a gradual dip between the first and second singular values, as shown in Figure 7. This is to be expected, given the weak correlation prevalent in the second dataset. As we can see in Figure 6, the approximate direction where the spread is maximized is $(1, 1)$. Although less spread/information is retained along the perpendicular vector $(-1, 1)$, the difference is not significant.

Based on the results from this simulation study, it is possible that there are many weakly correlated variables in our standardized uni-gram TF-IDF matrix. Given the sheer scale of this matrix, it is difficult to extract the correlation value between each feature; however, it is likely that many of these values are small in magnitude, and it is possible that this is one reason why we are not seeing a significant dip with the first 30 singular values. Although it is possible for a significant dip to occur after the 30th singular value, it would not be efficient to continue using the power method. This is because extracting the first 30 singular values still took the algorithm roughly 5 to 6 minutes.

6.2 Word Clouds

Given the discouraging results we were seeing with PCA, we decided to reexamine our word clouds. As mentioned earlier, a major takeaway from these visuals is that they not only reveal a clear divide between positive and negative reviews but a distinction that is rather intuitive. Prominent stem terms such as "excel", "great", "love", "bad", "worst", and "bore" are not only terms that we'd use to describe movie reviews but positive and negative life experiences overall.

From this, our team concluded that one possible, albeit crude, way of reducing the dimensionality is to select the top n stem terms from each rating group. If $n = 3$, "excel", "great", and "love" would be the top stem terms that we'd have from positive reviews while "bad", "worst", and "bore" would be the terms from negative reviews. We'd then expect positive reviews to have some terms from our "top positive stem bank" and to have very few terms from our "top negative stem bank". In other words, if $n = 3$, we'd expect most positive reviews to have "excel", "great", and/or "love" while lacking "bad", "worst", and "bore". Meanwhile, we'd anticipate the opposite pattern for negative reviews.

As we will illustrate in later sections, this method of dimension reduction can yield very promising results and provides several benefits when it comes to implementation and interpretation.

7 Logistic Regression

7.1 Description

$$P(Y_i = 1|X_i = x_i) = \frac{1}{1 + e^{-(\beta_0 + \beta^T x_i)}} \tag{1}$$

In this case, Y_i represents some rating label, which takes values 1 and 0. The value 1 represents a positive rating label while 0 represents a negative rating label. In addition to this, X_i represents some text review. Now, if we assume that the probability of Y_i being 1 given some X_i is equal to Equation 1, we can model the log odds as follows

$$log(\frac{h(x_i)}{1 - h(x_i)}) = \beta_0 + \beta^T x_i \tag{2}$$

In this case, $h(x_i)$ is simply a compact way of reexpressing Equation 1. Given that we are selecting the top n stem terms from positive and negative reviews, we can reexpress the log odds as follows

$$log(\frac{h(x_i)}{1 - h(x_i)}) = \beta_0 + \beta_1 excel_i + \beta_2 great_i + ... \beta_n love_i + \beta_{n+1} bad_i + \beta_{n+2} worst_i + ... \beta_{n+n} bore_i \tag{3}$$

In this case, β_1, β_2, ... β_n, β_{n+1}, β_{n+2}, ... β_{n+n} are the entries of our β vector. Given that we have a standardized uni-gram TF-IDF matrix, each stem feature can take any real values. As such, it is important to emphasize that we cannot technically think of each stem feature as a simple indicator variable. However, there are still some notable patterns worth observing. For instance, when a stem feature has little to no presence in a review, we see that it is often small in magnitude. However, when the presence is much more significant, we see that it is often large in magnitude.

$$log(\frac{h(x_i)}{1 - h(x_i)}) = \beta_0 + \beta_1 excel_i + \beta_2 great_i + ... \beta_n love_i \tag{4}$$

$$log(\frac{h(x_i)}{1 - h(x_i)}) = \beta_0 + \beta_{n+1} bad_i + \beta_{n+2} worst_i + ... \beta_{n+n} bore_i \tag{5}$$

As a result, we expect most positive reviews to effectively have the log odds shown in Equation 4. This is because, as mentioned before, we expect most positive reviews to have some terms from our "top positive stem bank" and to have very few terms from our "top negative stem bank". And so, we should see some stem features from our "top positive stem bank" to be large in magnitude and most stem features from our "top negative stem bank" to be small in magnitude. Although "small in magnitude" does not technically mean 0, it is worth noting that we will effectively be treating it as such, which is why Equation 4 only has stem features from our "top positive stem bank". The reason why we are doing this is that it reveals a powerful trick that we can exploit for our gradient descent algorithm, which we will further elaborate on later in this section. When it comes to negative reviews, we'd expect to effectively have the log odds shown in Equation 5 and essentially the opposite pattern as positive reviews.

$$loss = -\sum_{i=1}^{2000} y_i * log[h(x_i)] + (1 - y_i) * log[1 - h(x_i)] \tag{6}$$

And so, our goal now is to find the optimal β_0 value and β vector. This can be attained by minimizing the training logistic loss shown in Equation 6 (Balasubramanian). It is worth noting that other sources may describe different methods such as maximum likelihood estimation; however, it can be shown that maximizing the likelihood is the same as minimizing the logistic loss (refer to the appendix). With that said, we will continue the report by describing our process strictly in terms of minimizing logistic loss to avoid any unnecessary confusion. Now, when it comes to minimizing logistic loss, there is no closed-form solution. As a result, our team had to implement an iterative algorithm known as gradient descent.

$$\theta \leftarrow \theta - H^{-1} \nabla loss \tag{7}$$

We did initially consider other iterative algorithms such as the Newton-Raphson method, with its update procedure shown in Expression 7. In this case, θ represents a vector with its first entry being β_0 and its remaining entries being the entries of our β vector. H represents the Hessian matrix associated with our loss function while $\nabla loss$ represents the gradient of our loss function with respect to θ (refer to the appendix for the derivation of $\nabla loss$). With this said, we quickly turned down the Newton-Raphson method given that it conflicted with our main goal. Recall, our goal is to implement an efficient, predictive algorithm. However, as shown in Expression 7, the Newton-Raphson method requires the inverse of a Hessian matrix, which would be especially large if $n = 1000$. Now, taking the inverse of a matrix is computationally expensive as it is; however, if $n = 1000$, it is likely that the Newton-Raphson algorithm would have run for countless minutes before returning any results.

$$\theta \leftarrow \theta - \alpha \nabla loss \tag{8}$$

To avoid this potential issue, we chose to implement gradient descent, with its update procedure shown in Expression 8. Rather than spending a significant amount of time finding H^{-1}, we set a learning rate α that controls how quickly we descend to our minimum loss point. Like with other iterative algorithms, we must also set the appropriate tolerance level and initial guess to avoid prolonged convergence.

Now, when setting an initial guess for the β_0 value and β vector that minimizes logistic loss, our team admittedly assigned the value 0 to each entry. This was primarily inspired by the gradient descent algorithm implemented by Youtube user Carestonee, as they also had zeroes as their initial guess (Carestonee). However, we eventually discovered a powerful trick that could improve efficiency and cut the total number of iterations in half. Rather than having zeroes as our initial guess, we initially set β_0 as 0, the first n entries of β as 1, and the last n entries of β as -1. As we alluded to earlier, this powerful trick comes from the fact that we are effectively treating stem features that are "small in magnitude" as 0. This then allows us to have simpler log odds equations for positive and negative reviews, as shown in Equation 4 and Equation 5 respectively. By examining these equations, we can quickly observe notable patterns surrounding the log odds.

Recall, a log odds greater than 0 signifies that a review is more likely to be positive than negative; therefore, it would be sensible to classify the said review as positive. Based on Equation 4, we can see that one way for positive reviews to have positive log odds is when the first n entries of β, AKA β_1, β_2, ... β_n, are positive values. Conversely, a log odds less than 0 signifies that a review is more likely to be negative than positive; therefore, it would be sensible to classify the said review as negative. Based on Equation 5, we can see that one way for negative reviews to have negative log odds is when the last n entries of β, AKA β_{n+1}, β_{n+2}, ... β_{n+n}, are negative values. Based on these statements, one way for our algorithm to correctly classify as many reviews as possible is to have the first n entries of β be a positive value and the last n entries of β be a negative value. And so, one reasonable initial guess is to have the first n entries of β be positive 1 and the last n entries of β be negative 1.

Now, all that we are left with is β_0. In this case, β_0 represents the log odds when stem features from our "top positive stem bank" and "top negative stem bank" is exactly 0. However, there is no occurrence of this throughout our entire standardized uni-gram TF-IDF matrix. And so, it is difficult to extract any meaningful interpretation of β_0 that can be used to improve convergence. With no better initial guess for β_0, our team settled for a base initial guess of 0.

In addition to this, we set our tolerance level at the largest standard value of 0.001. This not only allows us to return fairly acceptable results but to return results within a quicker time frame than smaller tolerance levels. Although smaller tolerance levels often lead to more accurate results, our primary goal is efficiency. And so, for this project, we are willing to sacrifice some accuracy as long as our algorithm, as mentioned before, returns acceptable results within a quick time frame.

Through trial and error, we also found that one of the best initial learning rates was set at 0.01. However, this was not perfect and still caused minor issues. For instance, a 0.01 learning rate occasionally increased the logistic loss during certain iterations, causing the convergence process to take much longer. In order to speed up convergence, our team decided to implement step-halving, which involved us scaling the learning rate by $\frac{1}{2}$ each time the logistic loss increased.

After setting the appropriate initial guess, tolerance level, and learning rate, our team carried out k-folds cross-validation in order to find the optimal n AKA the number of top stem terms from positive and negative reviews. According to machine learning writer Rukshan Pramoditha, 5 and 10 are good standard folds to use (Pramoditha). However, when taking into account the sheer scale of our dataset along with our goal of efficiency, we ultimately opted for 5-folds. By doing so, we would be fitting fewer models, which would result in shorter running times.

Despite n taking any positive integer, we only tested 400, 600, 800, and 1000. We have 400 as our minimum possible n because we assume any logistic model with less than that would have a small "stem bank". This would likely result in many positive and negative reviews having identical log odds as neither group would contain anything from a stem bank with very few terms. With observations from different rating groups looking the same, classification becomes much more difficult and unclear. The reason why we have increments of 200 and why we have 1000 as our maximum possible n is purely from a computational perspective. Even when utilizing parallel computing to cross-validate 400, 600, 800, and 1000, our run time was still as high as 20 minutes. This is especially large given how we are excluding an entire set of n values larger than 1000. It is here where we must emphasize the limitation of our algorithm as it is possible that the n which minimizes logistic loss is larger than 1000. With this said, it is equally important to emphasize our primary goal of efficiency, which is difficult to achieve when our algorithm takes too long to test many large n values.

7.2 Results

After conducting k-folds cross-validation, we found that $n = 1000$ returned the lowest cross-validation error rate of 5.35%. When utilizing parallel computing techniques, k-folds cross-validation took roughly 20 minutes. When using $n = 1000$ to fit a logistic model on our training set, we were left with a final test error rate of 20%. Altogether, logistic model fitting and testing took roughly 1 to 2 minutes.

7.3 Discussion

One thing that our team found rather surprising was the difference between the cross-validation error and the final test error. Recall, k-folds cross-validation is a means of estimating the test error; however, it appears that we have greatly underestimated this value. One possible explanation for this is the small number of folds that we selected. Although fewer folds speed up computation, they also create a smaller sample of errors, which often results in unstable, unreliable mean values. It may also be because our training set is too small. When this occurs, patterns from the population may not be accurately represented. For instance, the divide between positive and negative reviews might be more clear in our training set than in the population, which would cause our cross-validation error to be smaller than our final test error.

For our gradient descent algorithm, it is also worth noting that β_1, β_2, ... β_{1000} were not all positive, with 1.9% being negative. Furthermore, β_{1001}, β_{1002}, ... β_{2000} were not all negative, with 2.2% being positive. Recall, when $n = 1000$, β_1, β_2, ... β_{1000} represent the coefficients for the top 1000 positive stem terms while β_{1001}, β_{1002}, ... β_{2000} represent the coefficients for the top 1000 negative stem terms. As mentioned before, we expected the first 1000 coefficients to be positive and the last 1000 coefficients to be negative. One possible explanation as to why this is not apparent could be that our tolerance level is too large. With a large tolerance, our gradient descent algorithm may have stopped too early and returned a result that is close yet slightly distant from the true minimum point. And so, it is possible that β_1, β_2, ... β_{1000} would have all reached positive values and that β_{1001}, β_{1002}, ... β_{2000} would have all reached negative values if our tolerance level was smaller.

$$P(Y_i = 1|X_i^{(1)} = x_i^{(1)}, X_i^{(2)} = x_i^{(2)}) = \frac{1}{1 + e^{-[1+2x_i^{(1)}+4x_i^{(2)}]}} \tag{9}$$

$$P(Y_i = 1|X_i^{(1)} = x_i^{(1)}, X_i^{(2)} = x_i^{(2)}) = \frac{arctan[-4 + 2e^{x_i^{(1)}} - x_i^{(2)}] + \frac{\pi}{2}}{\pi} \tag{10}$$

Additionally, our team conducted a simulation study to test our self-defined R algorithm for logistic gradient descent. This first involved generating two distinct datasets. The first dataset has 1000 observations and Y_i was generated using Equation 9, satisfying the logistic regression assumption. The second dataset also has 1000 observations but Y_i was generated using Equation 10, violating the logisitic regression assumption.

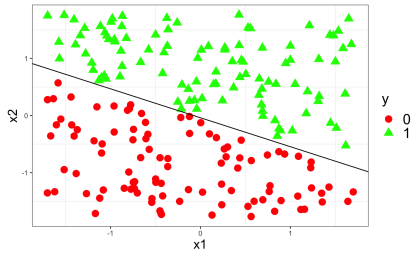

Figure 8:
Simulated Test Data and Logistic DB where Logistic Regression Assumption Satisfied

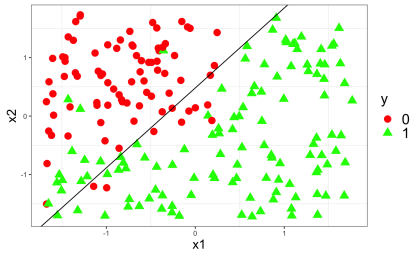

Figure 9:
Simulated Test Data and Logistic DB where Logistic Regression Assumption Violated

We then carried out an 80-20 train-test split for each dataset. Figure 8 shows our test observations from the first dataset along with the decision boundary (DB) that our logistic gradient descent algorithm fits. In this scenario, we are left with a final test error rate of 0%. This low value makes sense since we generated the first dataset in a way that satisfies the logistic regression assumption. Meanwhile, Figure 9 shows our test observations from the second dataset along with the DB that our logistic gradient descent algorithm fits. In this scenario, we are left with a final test error rate of 11%. Now, it makes sense for this value to be larger given that we generated the second dataset in a way that violates the logistic regression assumption. The reason why it is important to discuss these findings is that they provide an explanation behind our final test error rate of 20%. Based on our simulation study, one can argue that 20% is quite high and that it could be because we violated the logistic regression assumption. In these situations, we can lower our final test error rate by implementing other algorithms such as K-Nearest Neighbors (K-NN).

8 K-Nearest Neighbors (K-NN)

8.1 Description

It is important to note that we will continue to use the variable 'n' to denote stem size throughout the rest of this section. We decided to use K-NN after running two simulations to determine what algorithms give the lowest test errors. We wanted to see if K-NN can obtain a low test error rate to determine whether K-NN is an appropriate method for our project. For the first simulation (Figure 8), we generated the data in a way where the decision boundary would be linear; this data would be more appropriate for logistic regression. Using K-NN, it obtained an error rate of 0 with the appropriate k = 4,5,6. In the second simulation, we intentionally created data with a nonlinear decision boundary, which is better suited for K-NN since it does not require linear assumptions of the data. The scatter plot (Figure 9) for this simulation revealed some red and green points mixed within the opposing group, suggesting that error rates would likely increase regardless of the method used. Nonetheless, using K-NN with an optimal k of 5 or 7, we achieved a low error rate of 5.5%, which is considered excellent. It is also lower than our error rate using logistic regression. In general, K-NN performs better than logistic regression when the decision boundary is non-linear, as expected. These simulations covered linear and nonlinear decision boundaries for binary classification. After both simulations achieved low error rates with appropriate choices of k, we concluded that K-NN is a suitable method to be included in our project.

Before using K-NN, we first performed cross-validation using k-fold with 5 folds to optimize the hyperparameters for K-NN. Cross-validation divides our training data into train and validation subsets. These subsets are called folds, and we chose to use 5 folds because the number balances between bias and variance. This ensured that it would not underfit nor overfit our data.

Afterwards, we used the training data to determine which n gives the lowest test error. To determine the optimal value of k for our K-NN algorithm, we tested the range of k values from 1 to 10. We chose this range because it strikes a balance between computational efficiency and error rate, which is particularly important given the high-dimensional nature of our dataset. It is the most optimal for large and high dimensional data since testing more than 10 k values often becomes computationally expensive and time-consuming, while testing fewer k values does not provide enough information. There are limitations of K-NN in high-dimensional settings, but testing using this range of k is reasonable (Ning). Therefore, we used k values from 1 to 10 for the number of stems words ranging from 100 to 1000. We chose to go up to 1000 rather than all 2000 of the dataset because going any higher than 1000 would be extremely computationally expensive.

Our dataset consists of a training set and a test set, where the number of columns corresponds to the number of stem words used plus one column for the binary label of each observation. The number of rows indicates the number of observations included in either the training or test set. When using the K-NN algorithm, the distance between instances plays a crucial role in determining the output for a new instance.

One commonly used distance metric in K-NN is the Euclidean distance, which measures the distance between two points in a multidimensional space. Specifically, the Euclidean distance between two rows, denoted as (x_{11}, x_{12}, ..., y_1) and (x_{21}, x_{22}, ..., y_2), is computed (Anderson et al.). In K-NN, the algorithm searches for the k nearest neighbors to the new instance based on the Euclidean distance. Once the k nearest neighbors are identified, the majority class or average value of the k neighbors is assigned to the new instance as its predicted class or regression output. Thus, Euclidean distance is used in K-NN to evaluate the similarity between instances in a multidimensional space and to identify the K nearest neighbors for prediction.

Due to the high-dimensional nature of our dataset, we decided to optimize our code by switching from a for-loop to a sapply function. This is because matrix multiplication is faster than for-loops. However, the computational speed was still slow due to the curse of dimensionality. To address this issue, we decided to implement parallel computing wherever applicable in our K-NN algorithm. As a result, we were able to significantly increase the computational speed.

8.2 Results

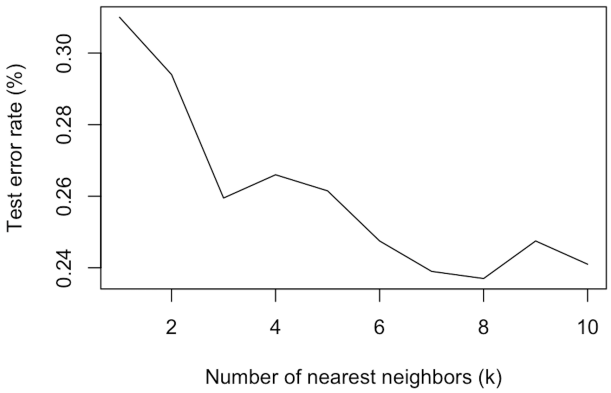

Figure 10: Relationship between k and test errors at n=100

After performing cross-validation by splitting the training set into validation and training subsets, we determined that a n of 100 and a value of k = 8 produced the lowest error rate. Figure 10 displays the performance of the K-NN algorithm for n=100 with different k values. Notably, we observed the minimum train error to be 23.70% with a running time of 315 seconds.

We then applied these parameters to the test set, resulting in an error rate of 26.20% and a computational time of approximately 18 seconds. Based on these findings, our algorithm performed relatively well, achieving an accuracy of approximately 74%. In practical terms, if we receive a new movie review, we will analyze its word usage to identify any matches with our positive or negative stem words. Depending on the number of positive or negative stem words identified, we will classify the review as either positive or negative. On average, our algorithm misclassified 26% of new movie reviews.

8.3 Discussion

Since the smallest n performed best, we determined that it is related to the curse of dimensionality, which occurs when working with high-dimensional data and can also negatively affect algorithms such as K-NN. Since this algorithm works by finding the k closest neighbors to a given data point based on the distance metric used, in high-dimensional spaces, the distance between points becomes increasingly large, and the difference between the nearest and farthest neighbors becomes smaller. As the number of dimensions increases, the volume of the space increases exponentially, leading to the sparsity of the data, and making it challenging to find meaningful patterns and relationships. This also leads to a significant increase in computational complexity and the risk of overfitting.

It is also worth noting that the k-fold cross-validation underestimated the test error. Cross-validation gave us a test error rate of 23.70%, while the actual test error rate was 26.20%. The difference between the two is not too significant, however, the relatively minor underestimation could be because of the small training set size, which results in the population not being accurately represented. Additionally, the number of folds we selected for the cross-validation may have been too small, resulting in unstable and unreliable mean values. We chose to use 5 folds as a trade-off between computational power and error rate, although ideally, leave-one-out cross-validation would be preferred because it would give us more training and test sets, which would give us more samples, and therefore a higher accuracy. Nonetheless, we decided not to use it since it would be too computationally expensive.

9 Conclusion

In summary, we found that $n = 1000$ returned the lowest cross-validation error rate (5.35%) for logistic regression, which took roughly 20 minutes. After running logistic gradient descent with $n = 1000$, we were left with a final test error rate of 20%, which took roughly 1 to 2 minutes. For K-NN, we found that $n = 100$ and $k = 8$ were the optimal hyperparameters, yielding the lowest cross-validation error rate of 23.70%. When applied to the test set, we received a final test error rate of 26.20% in just 18 seconds.

Based on these results, there are several things worth reflecting on. For instance, the computation time for our K-NN algorithm was shorter than our logistic algorithm. This is likely connected to the fact that our K-NN algorithm utilizes parallel computing while our logistic algorithm uses a simple for loop. However, it is worth noting that this is the best that we could do since we are minimizing logistic loss through iterative techniques, which use information from previous steps. Despite its computational advantage, our K-NN algorithm yields a larger test error than our logistic algorithm. One possible explanation could be the curse of dimensionality. In other words, K-NN tends to perform worse when the dimensions of a problem are high.

In addition to logistic regression and K-NN, other classification algorithms that we considered implementing include support vector machines, random forests, and neural networks. Support Vector Machines (SVMs) would be applicable because they are effective at finding the best hyperplane that separates positive and negative examples in the feature space, which is the goal of binary classification. SVMs can handle high-dimensional data, such as text data, and can capture nonlinear relationships between features using a variety of kernel functions. Random Forests would also be applicable because they can handle correlated features and are less prone to overfitting than individual decision trees. This is important when dealing with text data, which can have many correlated features (words) that may not be informative on their own but can be useful in combination. Neural Networks, such as multilayer perceptrons (MLPs) and convolutional neural networks (CNNs), are effective for text classification tasks because they can learn complex nonlinear relationships between the input features and the target variable. This is important in text classification because the meaning of a sentence or phrase can be highly dependent on the context and may not be captured by individual words alone. However, neural networks can be computationally expensive to train, especially for large datasets.

With these things in mind, it is important to emphasize that our goal is effective algorithm implementation rather than selection. And so, we leave the debate open on which algorithm is "better". The only thing that we stress is for people to carefully consider what they value. Do they value computation time or accuracy? What is a "good enough" computation time? What is a "good enough" accuracy?

10 Appendix

$$
\begin{aligned}
\theta_j &= \beta_{j-1} \text{ for } j = 1, 2, ...n, n+1, n+2, ...n+n \\
P_\theta(Y_i = y_i | X_i = x_i) &= \frac{1}{1 + e^{-y_i(\beta_0 + \beta^T x_i)}} \\
L(\theta) &= \prod_{i=1}^{m} P_\theta(Y_i = y_i | X_i = x_i) \prod_{i=1}^{m} P(X_i = x_i) \\
loss(\theta) &= -\sum_{i=1}^{m} y_i * log[P_\theta(Y_i = 1 | X_i = x_i)] + (1 - y_i) * log[1 - P_\theta(Y_i = 1 | X_i = x_i)] \\
\hat{\theta} &= \underset{\theta}{\text{argmax}} \ L(\theta) \\
&= \underset{\theta}{\text{argmax}} \ log[L(\theta)] \\
&= \underset{\theta}{\text{argmax}} \ log[\prod_{i=1}^{m} P_\theta(Y_i = y_i | X_i = x_i) \prod_{i=1}^{m} P(X_i = x_i)] \\
&= \underset{\theta}{\text{argmax}} \ \sum_{i=1}^{m} log[P_\theta(Y_i = y_i | X_i = x_i)] + \sum_{i=1}^{m} log[P(X_i = x_i)] \\
&= \underset{\theta}{\text{argmax}} \ \sum_{i=1}^{m} log[P_\theta(Y_i = y_i | X_i = x_i)] \\
&= \underset{\theta}{\text{argmax}} \ \sum_{i=1}^{m} y_i * log[P_\theta(Y_i = 1 | X_i = x_i)] + (1 - y_i) * log[P_\theta(Y_i = 0 | X_i = x_i)] \\
&= \underset{\theta}{\text{argmax}} \ \sum_{i=1}^{m} y_i * log[P_\theta(Y_i = 1 | X_i = x_i)] + (1 - y_i) * log[1 - P_\theta(Y_i = 1 | X_i = x_i)] \\
&= \underset{\theta}{\text{argmin}} \ -\sum_{i=1}^{m} y_i * log[P_\theta(Y_i = 1 | X_i = x_i)] + (1 - y_i) * log[1 - P_\theta(Y_i = 1 | X_i = x_i)] \\
&= \underset{\theta}{\text{argmin}} \ loss(\theta)
\end{aligned}
$$

The steps above show how maximizing $L(\theta)$, AKA the logistic likelihood, is the same as minimizing $loss(\theta)$, AKA the logistic loss. Recall, θ represents a vector where the first entry is β_0 and the remaining entries are the entries of our β vector. Additionally, recall that n represents the number of top stem features that we select from positive and negative reviews. In this case, $P_\theta(Y_i = y_i | X_i = x_i)$ is the logistic probability function, m is sample size, and $\hat{\theta}$ is the θ which maximizes logistic likelihood / minimizes logistic loss.

$$h_\theta(x_i) = [1 + exp[-(\beta_0 + \beta_1 x_i^{(1)} + \beta_2 x_i^{(2)} + ...\beta_n x_i^{(n)} + \beta_{n+1} x_i^{(n+1)} + \beta_{n+2} x_i^{(n+2)} + ...\beta_{n+n} x_i^{(n+n)})]]^{-1}$$

$$\tilde{x}_i^{(j)} = \begin{cases} 1 & \text{if } j = 1 \\ x_i^{(j)} & \text{otherwise} \end{cases}$$

$$\frac{\partial h_\theta(x_i)}{\partial \theta_j} = h_\theta(x_i) * [1 - h_\theta(x_i)] * \tilde{x}_i^{(j)}$$

$$\begin{aligned}
(\nabla loss)_j &= \frac{\partial loss}{\partial \theta_j} \\
&= -\left[\sum_{i=1}^{m}\left[y_i * \frac{1}{h_\theta(x_i)} * \frac{\partial h_\theta(x_i)}{\partial \theta_j}\right] + \sum_{i=1}^{m}\left[(1-y_i) * \frac{1}{1-h_\theta(x_i)} * \frac{\partial [1 - h_\theta(x_i)]}{\partial \theta_j}\right]\right] \\
&= -\left[\sum_{i=1}^{m}\left[y_i * \frac{1}{h_\theta(x_i)} * \frac{\partial h_\theta(x_i)}{\partial \theta_j}\right] + \sum_{i=1}^{m}\left[(1-y_i) * \frac{1}{1-h_\theta(x_i)} * -\frac{\partial [h_\theta(x_i)]}{\partial \theta_j}\right]\right] \\
&= -\left[\sum_{i=1}^{m}\left[y_i * \frac{1}{h_\theta(x_i)} * h_\theta(x_i) * [1 - h_\theta(x_i)] * \tilde{x}_i^{(j)}\right] + \sum_{i=1}^{m}\left[(1-y_i) * \frac{1}{1-h_\theta(x_i)} * -h_\theta(x_i) * [1 - h_\theta(x_i)] * \tilde{x}_i^{(j)}\right]\right] \\
&= -\left[\sum_{i=1}^{m}\left[y_i * [1 - h_\theta(x_i)] * \tilde{x}_i^{(j)}\right] + \sum_{i=1}^{m}\left[(1-y_i) * -h_\theta(x_i) * \tilde{x}_i^{(j)}\right]\right] \\
&= -\sum_{i=1}^{m} y_i * [1 - h_\theta(x_i)] * \tilde{x}_i^{(j)} + (1-y_i) * -h_\theta(x_i) * \tilde{x}_i^{(j)} \\
&= -\sum_{i=1}^{m} [y_i * [1 - h_\theta(x_i)] + (1-y_i) * -h_\theta(x_i)] * \tilde{x}_i^{(j)} \\
&= -\sum_{i=1}^{m} [y_i - y_i h_\theta(x_i) - h_\theta(x_i) + y_i h_\theta(x_i)] * \tilde{x}_i^{(j)} \\
&= -\sum_{i=1}^{m} [y_i - h_\theta(x_i)] * \tilde{x}_i^{(j)}
\end{aligned}$$

The steps above show how to derive $\nabla loss$, AKA the logistic loss gradient, element-wise. For simplicity, we have $h_\theta(x_i)$ represent the logistic probability function when $Y_i = 1$. Recall, n represents the number of top stem features that we select from positive and negative reviews. In addition to this, we define $\tilde{x}_i^{(j)}$, which takes 1 if j is 1 and $x_i^{(j)}$ otherwise. Given that $h_\theta(x_i)$ is a sigmoid function, we also utilized a chain rule trick when calculating $\frac{\partial h_\theta(x_i)}{\partial \theta_j}$(Thavanani). Lastly, recall that θ_j is β_{j-1} for $j = 1, 2, ...n, n+1, n+2, ...n+n$ and m is sample size.

11 Sources

[1] Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., ... Sorensen, D. (1999). LAPACK Users' Guide (Third Edition). Society for Industrial and Applied Mathematics. DOI: https://doi.org/10.1137/1.9780898719604

[2] Balasubramanian, K. (2023, March). STA 142A Statistical Learning I. Lecture, University of California, Davis.

[3] Carestonee. (2020, July 25). YouTube. Retrieved March 22, 2023, from https://www.youtube.com/watch?v=ROJDtZeLqNo.

[4] Gunjan, P. K. (2022, November 18). IMDB Review. Kaggle. Retrieved March 22, 2023, from https://www.kaggle.com/datasets/pawankumargunjan/imdb-review.

[5] Ning, B. (2023, March 14). STA 141C Big Data and High-Performance Statistical Computing Office Hour. University of California, Davis.

[6] Pramoditha, R. (2020, December 20). K-Fold Cross Validation Explained in Plain English. https://towardsdatascience.com/k-fold-cross-validation-explained-in-plain-english-659e33c0bc0.

[7] Thavanani, S. (2020, July 8). Derivative of log-loss function for logistic regression. Medium. Retrieved March 23, 2023, from https://medium.com/analytics-vidhya/derivative-of-log-loss-function-for-logistic-regression-9b832f025c2d.

12 Code

For PCA via the power method, we first utilized matrix-vector multiplication rather than matrix-matrix multiplication to speed up the computation. We used parallel computing when generating the training data set, test set, and corpus. Also, we used parallel computing for K-folds cross-validation and K-NN from one to ten. Furthermore, for logistic gradient descent, we selected an appropriate initial guess and the largest standard tolerance, as mentioned in the logistic regression description section, and we used the crossprod(x,y) function to speed up convergence. Lastly, for the Euclidean Distance in our K-NN algorithm, we utilized the norm(degree = 2), which uses the LAPACK package.

Code Appendix

```
# GENERATE DATA FUNCTION
library("foreach")      # Parallel Computing
library("doParallel")   # Parallel Computing
registerDoParallel(detectCores())
generate_data = function(wd,size){
  review_files_all = list.files(wd)
  review_files = sample(review_files_all, size)
  mat = foreach(i = review_files, .combine="rbind", .packages = "readr") %dopar% {
    # rbind returns matrix
    # readr package used for read_file()
    rating_score = as.numeric(strsplit(gsub(".txt","",i),"_")[[1]][2])
    if(rating_score >= 7){
      c(read_file(paste(wd,i,sep = "/")),1) # 1 is pos
    }else{
      c(read_file(paste(wd,i,sep = "/")),0) # 0 is neg
    }
  }
  df_raw = as.data.frame(mat)
  rownames(df_raw) = 1:size
  colnames(df_raw) = c("review", "rating")
  df_raw$rating = as.integer(df_raw$rating)
  return(df_raw)
}
# TRAIN DATA
#train_wd = "C:/Users/Josh Balingit/OneDrive/Desktop/STA 141C Train Set Combine"
  # This contains all the training positive and negative reviews
#train_size = 2000
#set.seed(24)
#df_train_raw = generate_data(train_wd,train_size)
# TEST DATA
#test_wd = "C:/Users/Josh Balingit/OneDrive/Desktop/STA 141C Test Set Combine"
    # This contains all the test positive and negative reviews
#test_size = 500
#set.seed(12)
#df_test_raw = generate_data(test_wd,test_size)
# FOR TEAMMATES TO ACCESS TRAIN AND TEST DATA
#setwd("C:/Users/Josh Balingit/OneDrive/Desktop/STA 141C Final Project")
#write.csv(df_train_raw,"df_train_raw.csv")
#write.csv(df_test_raw,"df_test_raw.csv")
# TRAIN SUMMARY
df = read.csv("df_train_raw.csv")
df = df[,!(colnames(df) %in% "X")]
mean(df$rating == 1) # Percent of Positive Reviews
mean(df$rating == 0) # Percent of Negative Reviews
# GENERATE CORPUS
library("tm")          # Text Mining
library("SnowballC")  # Text Stemming (Text Lemmatizing has issues with adverbs ending with "ly")
generate_corpus = function(reviews){
  review_no_special = foreach(i = reviews, .combine = "rbind") %dopar% {
    no_apostophe = gsub("'", "", i)
      # Possessive  : Jack's to Jacks
```

```
      # Contraction : would've to wouldve
      # Title       : 'Harry Potter' to Harry Potter
    no_punctuation = gsub("[[:punct:]]", " ", no_apostophe)
  }
  corpus = Corpus(VectorSource(review_no_special))
  corpus = tm_map(corpus, content_transformer(tolower))
  corpus = tm_map(corpus, content_transformer(removeNumbers))
  corpus = tm_map(corpus, removeWords, c(stopwords("english"), "film", "films", "movie", "movies"))
    # Assume "film", "films", "movie", "movies" WON'T help distinguish
    # They appear frequently across POS and NEG
    # This is b/c data has reviews about film/movies in general
  corpus = tm_map(corpus, stemDocument)
  return(corpus)
}
# DOCUMENT TERM MATRIX WITH TF-IDF WEIGHT ON TRAIN DATA
corpus = generate_corpus(df$review)
td = DocumentTermMatrix(corpus)
train_terms = Terms(td)
td =  td[,sort(train_terms)]
train_idf = log2(nDocs(td)/colSums(as.matrix(td>0)))
mat_td = t(t(as.matrix(td))*train_idf)
mat_td_std = scale(mat_td)
  # Standardizing columns with mean 0 and sd 1 is necessary for PCA
    # With mean 0, we can use SVD decomposition (A. Chandler MAT 167)
    # With sd 1, we prevent variance of each variable from being inflated/deflated (E. Furfaro STA 1
    # Empirically shown to lead to faster convergence for gradient descent (K. Balasubramanian STA 1
# WORD CLOUD ON TRAIN DATA
library("wordcloud")    # Word Cloud Graph
library("RColorBrewer") # Word Cloud Color Palettes
word_cloud_visual = function(rating,max_words,color_gradient){
  mat_td_rating = mat_td_std[df$rating == rating,]
  stem = colnames(mat_td_rating)
  stem_size = colSums(mat_td_rating)
    # In this case, size does NOT refer to stem frequency across documents
    # We are taking column sums as a measure to compare stem given some rating
      # LARGE column sums => Stem is connected to rating
      # SMALL column sums => Stem is NOT connected to rating
  wc_plot = wordcloud(words = stem,
                      freq = stem_size,
                      max.words=max_words,
                      random.order=F,
                      colors = color_gradient,
                      scale = c(2,.1))
  print(wc_plot)
}
word_cloud_visual(1,100,c("lightgreen","green2","green4"))
word_cloud_visual(0,100,c("lightpink","red2","red4"))
# POWER METHOD FUNCTION TO FIND SINGULAR VALUES
singular_vals_power_method = function(num_pc,num_iter,q0,X){
  A = tcrossprod(X)
  u_vecs = c()
  eigenval = c()
  for(j in 1:num_pc){
```

```
    q = as.matrix(q0,nrow = u_length)
    for(i in 1:num_iter){
      z = A%*%q[,i]
      q = cbind(q,z/norm(z,type = "2"))
      if(min(norm(q[,i+1]-q[,i],type = "2"),
             norm(q[,i+1]+q[,i],type = "2")) < 1e-6){
        cat("Convergence at trial", i, "\n")
        break
      }
    }
    u_vecs = cbind(u_vecs,q[,ncol(q)])
    eigenval = append(eigenval,crossprod(u_vecs[,j],X)%*%crossprod(X,u_vecs[,j]))
      # (A%*%B)%*%x cost more flops than A%*%(B%*%x)
      # eigenval of tcrossprod(mat_td_std)
      # sqrt(eigenval) of tcrossprod(mat_td_std) = singval of mat_td_std
    A = A - eigenval[j]*tcrossprod(u_vecs[,j])
  }
  return(sqrt(eigenval))
}
  # Power Method is used to find eigenvalues of A
  # If A is Xt_X, then square root of eigenvalues of A are singular values of X
## GGPLOT FOR DATA VISUALIZATION
library("ggplot2")
# SIMULATION FOR PCA
  # TO TEST IF POWER METHOD FUNCTION WORKS
  # TO ILLUSTRATE ISSUE OF PCA WITH TRAIN DATA
pca_sim = function(X_sim){
  # Plot of x1 vs x2
  plot_x1_x2 = ggplot(data=as.data.frame(X_sim))+
    geom_point(aes(x=x1_sim,y=x2_sim))+
    labs(x="x1 (scaled)",y="x2 (scaled)")+
    theme_bw()
  num_pc_sim = 2
  num_iter_sim = 10000
  u_length_sim = nrow(X_sim)
  z0_sim = rep(1,times=u_length_sim)
  q0_sim = z0_sim/norm(z0_sim,type = "2")
  singular_vals_sim = singular_vals_power_method(num_pc_sim,num_iter_sim,q0_sim,X_sim)
  # Plot of i vs ith singular value
  plot_i_sv = ggplot(data=data.frame("num_pc"=1:num_pc_sim,"singular_vals"=singular_vals_sim))+
    geom_point(aes(x=num_pc,y=singular_vals))+
    ylim(c(0,max(singular_vals_sim)+1))+
    labs(x="index i",y="ith singular value")+
    theme_bw()
  print(plot_x1_x2)
  print(plot_i_sv)
}
# CASE 1: HIGH CORRELATED COVARIATES
set.seed(1)
x1_sim = runif(100,min=-10,max=10)
x2_sim = 0.5*x1_sim + rnorm(100,mean=0,sd=1)
X_sim = scale(cbind(x1_sim,x2_sim))
pca_sim(X_sim)
```

```r
# CASE 2: WEAK CORRELATED COVARIATES
set.seed(2)
x1_sim = runif(100,min=-10,max=10)
x2_sim = 0.05*x1_sim + rnorm(100,mean=0,sd=1)
X_sim = scale(cbind(x1_sim,x2_sim))
pca_sim(X_sim)
# PCA (NOTE: TOO LONG TO RUN)
num_pc = 30
num_iter = 10000
u_length = nrow(mat_td_std)
z0 = rep(1, times = u_length)
q0 = z0/norm(z0,type = "2")
singular_vals = singular_vals_power_method(num_pc,num_iter,q0,mat_td_std)
ggplot(data=data.frame("num_pc"=1:num_pc,"singular_vals"=singular_vals))+
    geom_point(aes(x=num_pc,y=singular_vals))+
    ylim(c(0,max(singular_vals)+1))+
    labs(x="principal component index",y="singular value")+
    theme_bw()
  # Based on simulation for PCA,
    # Weak Correlated Variables in mat_td_std
    # Points of mat_td_std "look less correlated" given Dimension of mat_td_std is large
    # Correlation is likely "hidden" by addition "noise" stem terms
# GRADIENT DESCENT FUNCTION FOR LOGISTIC REGRESSION
exp_neg_X_beta = function(X,beta){
  exp = exp(-X%*%beta)

  # To prevent underflow
  smallest_double = .Machine$double.eps
  exp = ifelse(exp < smallest_double, smallest_double,exp)

  return(exp)
}
h_beta = function(X,beta){
  return(1/(1+exp_neg_X_beta(X,beta)))
}
loss = function(y,X,beta){
   return(-sum(y*log(h_beta(X,beta))+
               (1-y)*log(1-h_beta(X,beta))))
}
gradient_descent = function(X,y,num_iter,tol,alpha,beta_vecs){
  for(i in 1:num_iter){
    h_y = h_beta(X,beta_vecs[,i]) - y
    gradient = crossprod(X,h_y)
    beta_vec_new = beta_vecs[,i] - alpha*gradient

    # While loss increase, descrease learning rate (Assistance From ChatGPT)
    if(loss(y,X,beta_vec_new) > loss(y,X,beta_vecs[,i])) {
      alpha = alpha / 2
    }
    beta_vecs = cbind(beta_vecs,beta_vec_new)

    if(norm(beta_vecs[,i+1]-beta_vecs[,i],type = "2") < tol){
      converge_trial = i
```

```
      return(list("beta" = beta_vecs[,ncol(beta_vecs)],
                  "converge_trial" = converge_trial))
    }
  }
  return(list("beta" = beta_vecs[,ncol(beta_vecs)],
              "converge_trial" = "NO CONVERGE"))
}
# TEST RESULTS FOR LOGISTIC REGRESSION
test_results_logistic = function(X_test,y_test,beta_hat_train,n){
  log_odds_pred = X_test%*%beta_hat_train
  y_pred = ifelse(log_odds_pred >= 0, 1, 0)
  o_v_p = table("observed" = y_test,"predicted" = y_pred)
  error_rate = mean(y_test != y_pred)
  return(list("o_v_p" = o_v_p,
              "error_rate" = error_rate))
}
# SIMULATION FOR GRADIENT DESCENT FOR LOGISTIC REGRESSION
  # TO TEST IF GRADIENT DESCENT FUNCTION FOR LOGISTIC REGRESSION WORKS
gd_logisitic_sim = function(X_scale_sim,y_sim){
  n = nrow(X_scale_sim)
  p = ncol(X_scale_sim)
  # CREATE 80% TRAIN AND 20% TEST SPLIT
  test_index_sim = sample(1:n,size=n/5)
  X_test_sim = X_scale_sim[test_index_sim,]
  y_test_sim = y_sim[test_index_sim]
  train_index_sim = -test_index_sim
  X_train_sim = X_scale_sim[train_index_sim,]
  y_train_sim = y_sim[train_index_sim]
  # FIT LOGISTIC MODEL USING GRADIENT DESCENT WITH TRAIN
  num_iter_sim = 100000
  tol_sim = 1e-6
  alpha_sim = 0.1
  beta_vec_0_sim = rep(0,times=p)
  beta_vecs_sim = as.matrix(beta_vec_0_sim)
  gd_results_sim = gradient_descent(X_train_sim,y_train_sim,num_iter_sim,tol_sim,alpha_sim,beta_vecs
  beta_hat_train_sim = gd_results_sim$beta
  # CALCULATE TEST ERROR RATE USING TRAIN LOGISTIC MODEL
  test_results_sim = test_results_logistic(X_test_sim,y_test_sim,beta_hat_train_sim,n=length(y_test_
  error_rate_sim = test_results_sim$error_rate
  # VISUAL OF TEST DATA POINTS WITH TRAIN DECISION BOUNDARY
  plot_sim = ggplot(data=data.frame("x1"=X_test_sim[,2],"x2"=X_test_sim[,3],"y"=as.factor(y_test_sim
    geom_point(aes(x=x1,y=x2,col=y,shape=y))+
    scale_color_manual(values = c("red","green"))+
    geom_abline(intercept = -beta_hat_train_sim[1]/beta_hat_train_sim[3],
                slope=-beta_hat_train_sim[2]/beta_hat_train_sim[3])+
    labs(x="x1 (scaled)",y="x2 (scaled)")+
    theme_bw()
  # SUMMARY
  cat("Train Beta is", beta_hat_train_sim, "\n")
  cat("Test Error Rate using Train Model is", error_rate_sim)
  print(plot_sim)
}
# CASE 1: GENERATE DATA BASED ON LINEAR DECISION BOUNDARY AND LOGISITIC PROBABILITIES
```

```r
set.seed(3)
beta_sim = c(1,2,4)
xs_sim = sapply(1:2,function(i){
  return(runif(1000,min=-10,max=10))
})
X_sim = cbind(1,xs_sim[,1],xs_sim[,2])
prob_y_1_sim = h_beta(X_sim,beta_sim)
y_sim = sapply(prob_y_1_sim,function(i){
  return(rbinom(n=1,size=1,prob=i))
})
X_scale_sim = cbind(1,scale(X_sim[,2:ncol(X_sim)]))
gd_logisitic_sim(X_scale_sim,y_sim)
  # TEST error is LOW
    # This makes sense because our data was generated based on logistic regression assumption
  # TRAIN beta does NOT equal TRUE beta
    # This makes sense because we scaled our data
    # This is fine because our goal is to NOT estimate TRUE beta
    # This is fine because our goal is to have LOW TEST error
# CASE 2: GENERATE DATA BASED ON EXPONENTIAL DECISION BOUNDARY AND ARCTAN PROBABILITES
set.seed(4)
xs_sim = sapply(1:2,function(i){
  return(runif(1000,min=-10,max=10))
})
model_sim = -4+2*exp(xs_sim[,1])-xs_sim[,2]
prob_y_1_sim = (atan(model_sim)+pi/2)/pi
y_sim = sapply(prob_y_1_sim,function(i){
  return(rbinom(n=1,size=1,prob=i))
})
X_scale_sim = cbind(1,scale(cbind(xs_sim[,1],xs_sim[,2])))
gd_logisitic_sim(X_scale_sim,y_sim)
  # TEST error is HIGHER
    # This makes sense because our data was NOT generated based on logistic regression assumption
# GENERATE STANDARDIZED DATA WITH RATING COLUMN (REDUCED TO TOP POS AND NEG STEM)
generate_df_std_red = function(stem_size_each,mat_td_std,rating){
  top_stem_each = sapply(0:1,function(i){
    mat_td_rating = mat_td_std[df$rating == i,]
    stem = colnames(mat_td_rating)
    stem_size = colSums(mat_td_rating)
    stem_top = head(stem[order(stem_size,decreasing = T)],stem_size_each)
    stem_top
  })
  stem1_top = top_stem_each[,2]
  stem0_top = top_stem_each[,1]
  top_stem = c(stem1_top[!(stem1_top %in% stem0_top)],
               stem0_top[!(stem0_top %in% stem1_top)])
    # Remove intersection between two ratings
  stem1_num = sum(stem1_top %in% top_stem)
  stem0_num = sum(stem0_top %in% top_stem)
  mat_td_std_red = mat_td_std[,top_stem]
    # Head stem_size_each columns are pos
    # Tail stem_size_each columns are neg
  df_std_red = as.data.frame(mat_td_std_red)
  df_std_red = cbind("beta0" = 1,df_std_red)
```

```
  df_std_red$rating = rating
  return(list("df_std_red" = df_std_red,
              "top_stem" = top_stem,
              "stem1_num" = stem1_num,
              "stem0_num" = stem0_num))
}
# TRAIN STANDARDIZED DATA WITH RATING COLUMN (REDUCED TO TOP POS AND NEG STEM)
start = proc.time()
test_error_rate_estimates = foreach(stem_size_each = c(400,600,800,1000),.combine = "rbind", .packag
  df_std_red_results = generate_df_std_red(stem_size_each,mat_td_std,df$rating)
  df_std_red = df_std_red_results$df_std_red
  top_stem = df_std_red_results$top_stem
  stem1_num = df_std_red_results$stem1_num
  stem0_num = df_std_red_results$stem0_num
    # Extract stem size for each rating again b/c function removes intersection between two ratings
  num_betas = 1 + stem1_num + stem0_num
    # 1 b/c of intercept
  # K-FOLDS CROSS VALIDATION WITH GRADIENT DESCENT FOR LOGISTIC REGRESSION
  k_folds = 5
  k_sizes = nrow(df_std_red)/k_folds
  k_labels = rep(1:k_folds,each=k_sizes)
  k_df = split(df_std_red, k_labels)
    # By converting to mat_td_std_red to df_std_red, I preserve row and column names
  k_error_rate_valid = foreach(i=1:k_folds,.combine = "rbind")%dopar%{
    df_train_valid = k_df[[i]]
    X_train_valid = as.matrix(df_train_valid[,-(num_betas+1)])
    y_train_valid = df_train_valid[,(num_betas+1)]
    df_train_valid_index = as.integer(rownames(k_df[[i]]))
    df_train_fit = df_std_red[-df_train_valid_index,]
    X_train_fit = as.matrix(df_train_fit[,-(num_betas+1)])
    y_train_fit = df_train_fit[,(num_betas+1)]
    num_iter = 10000
    tol = 1e-3
    alpha = 0.01
    beta_vec_0 = c(0,rep(c(1,-1),times=c(stem1_num,stem0_num)))
      # log(p1/p0) = beta_0 + beta_1*x_1 + ... beta_(stem1_num+stem0_num)*x_(stem1_num+stem0_num)
      # log(p1/p0) > 0 <=> Pos Review <=> beta_i for i = 1, ... stem1_num are Pos
      # log(p1/p0) < 0 <=> Neg Review <=> beta_i for i = stem1_num+1, ... stem1_num+stem0_num are Ne
    beta_vecs = as.matrix(beta_vec_0)
    gd_results = gradient_descent(X_train_fit,y_train_fit,num_iter,tol,alpha,beta_vecs)
    beta_hat_fit = gd_results$beta
    test_results = test_results_logistic(X_train_valid,y_train_valid,beta_hat_fit,k_sizes)
    error_rate = test_results$error_rate
    error_rate
  }
  test_error_rate_estimate = mean(k_error_rate_valid)
  test_error_rate_estimate
}
end = proc.time()
end-start
# FIT LOGISTIC REGRESSION WITH TRAIN SET WITH CERTAIN NUM OF STEM
stem_size_each = 1000
  # Yield lowest CV error
```

```
df_std_red_results = generate_df_std_red(stem_size_each,mat_td_std,df$rating)
df_std_red = df_std_red_results$df_std_red
top_stem = df_std_red_results$top_stem
stem1_num = df_std_red_results$stem1_num
stem0_num = df_std_red_results$stem0_num
  # Extract stem size for each rating again b/c function removes intersection between two ratings
num_betas = 1 + stem1_num + stem0_num
df_train = df_std_red
X_train = as.matrix(df_train[,-(num_betas+1)])
y_train = df_train[,(num_betas+1)]
num_iter = 10000
tol = 1e-3
alpha = 0.01
beta_vec_0 = c(0,rep(c(1,-1),times=c(stem1_num,stem0_num)))
  # log(p1/p0) = beta_1*x_1 + ... beta_(stem1_num+stem0_num)*x_(stem1_num+stem0_num)
  # log(p1/p0) > 0 <=> Pos Review <=> beta_i for i = 1, ... stem1_num are Pos
  # log(p1/p0) < 0 <=> Neg Review <=> beta_i for i = stem1_num+1, ... stem1_num+stem0_num are Neg
beta_vecs = as.matrix(beta_vec_0)
start = proc.time()
gd_results = gradient_descent(X_train,y_train,num_iter,tol,alpha,beta_vecs)
end = proc.time()
time_info = end - start
time_passed = time_info["elapsed"]
print(time_passed)
beta_hat_train = gd_results$beta
# TEST SUMMARY
df_test = read.csv("df_test_raw.csv")
df_test = df_test[,!(colnames(df_test) %in% "X")]
mean(df_test$rating == 1)
mean(df_test$rating == 0)
# CONVERTING TEST SET IN TERMS OF TRAIN SET
# Code between was created with assitance from ChatGPT
corpus_test = generate_corpus(df_test$review)
td_test = DocumentTermMatrix(corpus_test, control = list(dictionary = train_terms))
td_test = td_test[,sort(train_terms)]
mat_td_test = t(t(as.matrix(td_test))*train_idf)
mat_td_std_test = scale(mat_td_test, center = colMeans(mat_td), scale = apply(mat_td, 2, sd))
# Code between was created with assitance from ChatGPT
X_test = cbind("beta0" = 1,mat_td_std_test[,top_stem])
  # Head stem_size_each columns are pos
  # Tail stem_size_each columns are neg
y_test = df_test$rating
# LOGISTIC REGRESSION EVALUATION
test_results = test_results_logistic(X_test,y_test,beta_hat_train,test_size)
error_rate = test_results$error_rate
print(error_rate)
#Function 1: Euclidean Distance
#Calculate the euclidean distance between a single row in test data (1x2, excluding the binary label
#to a single row in the train data (1x2, excluding the binary label)
euclidean_distance <- function(row, train_data) {
  apply(train_data, 1, function(train_row) norm(row - train_row, type = '2'))
}
```

```r
#Function 2: KNN
#This function takes a row of the test matrix and the entire training data matrix,
#calculate the distance between that row to the entire training data matrix,
#we should have a vector with length of 800 for distance. Then, we look at the
#label of the nearest neighbors, and make the decision
knn_predict <- function(train_data, test_data_point, k, cl){
  # Calculate distance between test data and all train data
  distances <- apply(test_data_point,1,euclidean_distance,train_data[,-3]) #800
  # Sort distance to find the closest ones (number of k)
  neighbors <- train_data[order(distances), ][1:k,3] #k x 3
  # Predict the class of the test data point as the majority class among the neighbors
  ifelse(sum(neighbors) > k/2, 1, 0) #use the nearest neighbor to make prediction, length of 1
}
#Case 1 Stimulation / Logistic
set.seed(3)
#Data generate section
#Functions for data generating process
h_beta = function(X,beta){
  return(1/(1+exp_neg_X_beta(X,beta)))
}
exp_neg_X_beta = function(X,beta){
  exp = exp(-X%*%beta)
  # To prevent underflow
  smallest_double = .Machine$double.eps
  exp = ifelse(exp < smallest_double, smallest_double,exp)
  return(exp)
}

#Data generate
beta_sim = c(1,2,4)
xs_sim = sapply(1:2,function(i){
  return(runif(1000,min=-10,max=10))
})
X_sim = cbind(1,xs_sim[,1],xs_sim[,2])
prob_y_1_sim = h_beta(X_sim,beta_sim)
y_sim = sapply(prob_y_1_sim,function(i){
  return(rbinom(n=1,size=1,prob=i))
})
X_scale_sim = scale(X_sim[,2:ncol(X_sim)])

#Dotpot that display the distribution of the stimulation data
ggplot(data = data.frame("x1"=X_scale_sim[,1],"x2"=X_scale_sim[,2],"y"=as.factor(y_sim)))+
  geom_point(aes(x=x1,y=x2,col=y,shape=y))+
  scale_color_manual(values = c("red","green"))+
  labs(x="x1",y="x2")+
  theme_bw()

#Rescale data
#X_scale_sim = scale(X_sim)
test_index_sim = sample(1:1000,size = 200)
X_test_sim = X_scale_sim[test_index_sim,] #test dataset
y_test_sim = y_sim[test_index_sim] #test dataset
train_index_sim = -test_index_sim
```

```r
X_train_sim = X_scale_sim[train_index_sim,] #train dataset
y_train_sim = y_sim[train_index_sim] #train dataset

#Combine x and y
train_data_sim = cbind(X_train_sim,y_train_sim)
test_data_sim=cbind(X_test_sim,y_test_sim)

#This is a for loop to check the error rates for the k we want
error_rates <- c() # create an empty vector to store the error rates
for (k in 1:10) {
  knn_predictions <- sapply(seq_len(nrow(test_data_sim)), function(i) {
    knn_predict(train_data = train_data_sim, test_data_sim[i, 1:2, drop = FALSE], k, cl)
  })
  actual <- test_data_sim[,3]
  error_rate <- mean(knn_predictions != actual)
  error_rates[k] <- error_rate # store the error rate for this k value in the error_rates vector
}
plot(1:10, error_rates, type="l", xlab="k", ylab="Test error")
#Case 2 Stimulation / KNN
set.seed(4)

#Data generate section
xs_sim = sapply(1:2,function(i){
  return(runif(1000,min=-10,max=10))
})
model_sim = -4+2*exp(xs_sim[,1])-xs_sim[,2]
prob_y_1_sim = (atan(model_sim)+pi/2)/pi
y_sim = sapply(prob_y_1_sim,function(i){
  return(rbinom(n=1,size=1,prob=i))
})
X_sim = cbind(xs_sim[,1],xs_sim[,2])

#Dotpot that display the distribution of the stimulation data
ggplot(data = data.frame("x1"=X_sim[,1],"x2"=X_sim[,2],"y"=as.factor(y_sim)))+
  geom_point(aes(x=x1,y=x2,col=y,shape=y))+
  scale_color_manual(values = c("red","green"))+
  labs(x="x1",y="x2")+
  theme_bw()

#Rescale data
X_scale_sim = scale(X_sim)
test_index_sim = sample(1:1000,size = 200)
X_test_sim = X_scale_sim[test_index_sim,] #test dataset
y_test_sim = y_sim[test_index_sim] #test dataset
train_index_sim = -test_index_sim
X_train_sim = X_scale_sim[train_index_sim,] #train dataset
y_train_sim = y_sim[train_index_sim] #train dataset

#Combine x and y
train_data_sim = cbind(X_train_sim,y_train_sim)
test_data_sim =cbind(X_test_sim,y_test_sim)

#This is a for loop to check the error rates for the k we want
```

```r
error_rates <- c() # create an empty vector to store the error rates
for (k in 1:10) {
  knn_predictions <- sapply(seq_len(nrow(test_data_sim)), function(i) {
    knn_predict(train_data = train_data_sim, test_data_sim[i, 1:2, drop = FALSE], k, cl)
  })
  actual <- test_data_sim[,3]
  error_rate <- mean(knn_predictions != actual)
  error_rates[k] <- error_rate # store the error rate for this k value in the error_rates vector
}
plot(1:10, error_rates, type="l", xlab="k", ylab="Test error")
#KNN functions 3
euclidean_distance <- function(row, train_data) {
#uses the apply function to apply a function that computes the Euclidean distance
#between row and each row of train_data using the norm function with type = '2'.
  apply(train_data, 1, function(train_row) norm(row - train_row, type = '2'))
}

#KNN function 4
knn_predict <- function(train_data, test_data_point, k){
  # Calculate distance between test data and all train data
  distances <- apply(test_data_point,1,euclidean_distance,train_data[,-train_data_rating_index]) #80
  # Sort distance to find the closest ones (number of k)
  neighbors <- train_data[order(distances), ][1:k,201] #k x 3
  # Predict the class of the test data point as the majority class among the neighbors
  ifelse(sum(neighbors) > k/2, 1, 0) #use the nearest neighbor to make prediction, length of 1
}
#Start: Data set up
#Document term matrix with TF-IDF Weight on Train Data
df = read.csv("/Users/lucchen/Desktop/STA 141C/Final project/df_train_raw.csv")
df = df[,!(colnames(df) %in% "X")]

#Corpus
corpus = generate_corpus(df$review)
td = DocumentTermMatrix(corpus)
train_terms = Terms(td)
td = td[,sort(train_terms)]
train_idf = log2(nDocs(td)/colSums(as.matrix(td>0)))
mat_td = t(t(as.matrix(td))*train_idf)
mat_td_std = scale(mat_td)

# TRAIN STANDARDIZED DATA WITH RATING COLUMN (REDUCED TO TOP POS AND NEG STEM)
stem_size_each = 100
df_std_red_results = generate_df_std_red(stem_size_each,mat_td_std,df$rating)
df_std_red = df_std_red_results$df_std_red[,-1]
top_stem = df_std_red_results$top_stem
stem1_num = df_std_red_results$stem1_num
stem0_num = df_std_red_results$stem0_num
# Extract stem size for each rating again b/c function removes intersection between two ratings
num_of_stems = stem1_num + stem0_num
# 1 b/c of intercept

y_train_index = which(names(df_std_red) == "rating") #column index of the rating column or the y_tra
X_train = as.matrix(df_std_red[,-y_train_index]) #X_train as KNN input
```

```
y_train = as.matrix(df_std_red[,y_train_index]) #y_train as KNN input
colnames(y_train) = "rating"
train_data = cbind(X_train,y_train)
train_data_rating_index = which(colnames(train_data)=="rating")
#### Cross Validation ####
k_folds=5
k_sizes=nrow(df_std_red)/k_folds
k_labels = rep(1:k_folds,each=k_sizes)
k_df = split(df_std_red, k_labels)
error_list = c()

# Set up a parallel backend with multiple cores
cl <- makeCluster(detectCores())
registerDoParallel(cl)

start = proc.time()
k_error_rate_list <- foreach(k=1:10, .combine="c", .packages = c("foreach","doParallel")) %dopar% {
  k_error_rate_valid_list <- foreach(i=1:k_folds, .combine="c", .packages = c("foreach","doParallel"
    # retrieves the training and validation data for the i-th fold from a list k_df.
    df_train_valid <- k_df[[i]]
    # split the training and validation data into predictors (X_train_valid) and response (y_train_v
    X_train_valid <- as.matrix(df_train_valid[,-(num_of_stems+1)])
    y_train_valid <- df_train_valid[,(num_of_stems+1)]
    # extracts the row indices of the training and validation data.
    df_train_valid_index <- as.integer(rownames(k_df[[i]]))
    # creates the training data by removing the validation data from the full data set df_std_red.
    df_train_fit <- df_std_red[-df_train_valid_index,]
    # split the training data into predictors (X_train_fit) and response (y_train_fit).
    X_train_fit <- as.matrix(df_train_fit[,-(num_of_stems+1)])
    y_train_fit <- df_train_fit[,(num_of_stems+1)]
    # apply the knn_predict function to each row of the validation data df_train_valid by parallel c
    # train_data is the training data, test_data_point is a single row of the validation data, and k
    # nearest neighbors to consider.
    knn_predictions_list <- foreach(j = seq_len(nrow(df_train_valid)), .combine = "c") %dopar% {
      knn_predict(train_data = df_train_fit, test_data_point = df_train_valid[j, -ncol(df_train_vali
    }
    # compute the error rate for the current fold by comparing the predicted
    # ratings (knn_predictions) to the actual ratings (actual) and taking the mean of the resulting
    actual <- df$rating[df_train_valid_index]
    error <- mean(knn_predictions_list != actual)
    error #error list
  }
  mean(k_error_rate_valid_list) #mean error rate for k-fold
}

stopCluster(cl)

# plot the 10 error rates
plot(1:10, k_error_rate_list, type="l",  xlab="Number of nearest neighbors (k)", ylab="Test error ra
end = proc.time()
time_info = end - start
time_passed = time_info["elapsed"]
print(time_passed)
```

```
#Start: Data set up
#Document term matrix with TF-IDF Weight on Train Data
df = read.csv("/Users/lucchen/Desktop/STA 141C/Final project/df_train_raw.csv")
df = df[,!(colnames(df) %in% "X")]
df_test = read.csv("/Users/lucchen/Desktop/STA 141C/Final project/df_test_raw.csv")
df_test = df_test[,!(colnames(df_test) %in% "X")]

#Corpus
registerDoParallel(detectCores())
corpus = generate_corpus(df$review)
td = DocumentTermMatrix(corpus)
train_terms = Terms(td)
td = td[,sort(train_terms)]
train_idf = log2(nDocs(td)/colSums(as.matrix(td>0)))
mat_td = t(t(as.matrix(td))*train_idf)
mat_td_std = scale(mat_td)

#Stems
stem_size_each = 100 #This is our hyperparameter
df_std_red_results = generate_df_std_red(stem_size_each,mat_td_std,df$rating)
df_std_red = df_std_red_results$df_std_red
df_std_red = df_std_red[,-1] #This one is the X_train and y_train, sepearate them into X_train,y_tra
top_stem = df_std_red_results$top_stem

y_train_index = which(names(df_std_red) == "rating") #column index of the rating column or the y_tra
X_train = as.matrix(df_std_red[,-y_train_index]) #X_train as KNN input
y_train = as.matrix(df_std_red[,y_train_index]) #y_train as KNN input
colnames(y_train) = "rating"

#Converting test set in terms of Train Set
corpus_test = generate_corpus(df_test$review)
td_test = DocumentTermMatrix(corpus_test,control = list(dictionary = train_terms))
td_test = td_test[,sort(train_terms)]
mat_td_test = t(t(as.matrix(td_test))*train_idf)
mat_td_std_test = scale(mat_td_test,center = colMeans(mat_td),scale = apply(mat_td,2,sd))
X_test = as.matrix(mat_td_std_test[,top_stem]) #X_test as KNN input
y_test = as.matrix(df_test$rating) #y_test as KNN input
colnames(y_test) = "rating"
# Set up a parallel backend with multiple cores
cl <- makeCluster(detectCores())
registerDoParallel(cl)

#Apply KNN to our dataest (Starting of the algorithm)
train_data = cbind(X_train,y_train)
test_data = cbind(X_test,y_test)

train_data_rating_index = which(colnames(train_data)=="rating")
test_data_rating_index = which(colnames(test_data)=="rating")

##############Stemming size =100, K=8###################################
start = proc.time()
knn_predictions_list <- foreach(j = seq_len(nrow(test_data)), .combine = "c") %dopar% {
      knn_predict(train_data = train_data, test_data_point = test_data[j, -ncol(test_data), drop = F
```

```
}
actual <- test_data[, ncol(test_data)]
mean(knn_predictions_list != actual)
end = proc.time()
time_info = end - start
time_passed = time_info["elapsed"]
print(time_passed)
```