Project 2
Hannah Doherty
CSO135

For this project, the task was to develop binary classifiers using supervised learning. Exactly 2400 reviews were supplied from imdb.com, amazon.com, and yelp.com that were labeled indicating the sentiment (1 for positive, 0 for negative). There were also 600 test reviews that were not labeled with a sentiment for the purpose of uploading to a leaderboard to test the accuracy of the model created.

Part One: Classifying Review Sentiment with Bag-of-Words Features

1. Generating Bag-of-Words Model

Part one of this project involves using a Bag-of-Words (BoW) model to determine the fixed vector length of each review. The vector length is the magnitude of the vocabulary used in all the reviews combined. An important part of natural language processing is the pre-processing step and the following is the process for generating my Bag-of-Words features. To build the BoW model, I used the training dataset and 5-fold cross-validation to develop a validation dataset. **CountVectorizer** was the class used to convert the text into a matrix of token counts. The text being fed into this would ideally not contain any spelling errors, non-English language, accents, non-uniform upper-case characters, and slang. In order to accomplish this I had to think about feature selection and pre-processing before turning the reviews into a vector.

I started by analyzing how big the vocabulary is and of those how many are nouns, verbs, adjectives, proper nouns, and adverbs. I printed out the dimensions of the **CountVectorizer** model and it came out to be 2400 reviews with 4510 features or vocabulary. I then used a library called **SpaCy** which is used for information extraction, natural language understanding systems, or pre-process text for deep learning to determine how many features I could remove if I just chose to focus on nouns, verbs, adjectives, etc. From this analysis, it is shown that there are initially 4510 words in the training set review and 2400 reviews. But after removing superfluous words, there are at most 4324 words left in the training dataset. This will be the new dataset used in future hyperparameter tuning. Plotting the **CountVectorizer** as a matrix table also showed that the vocabulary was organized alphabetically. This made sense from an organization standpoint so I did not change it.

	00	10	100	11	12	13	15	15g	15pm	17	...	youtube	yucky	yukon	yum	yummy	yun	z500a	zero	zillion	zombie
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	1	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
2395	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
2396	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
2397	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
2398	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
2399	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

2400 rows × 4510 columns

Continuing with the pre-processing, I looked at certain non-uniform features like accents and uppercase letters. **CountVectorizer** is a class that converts text into a matrix of token counts and has parameters that luckily take care of these two pre-processing steps. When the parameter **strip_accents** is set to its default 'None', then it will reduce dimensionality by making the input consistent when it comes to accented characters. When it comes to character cases, the parameter **lowercase** has a default of 'True' which automatically converts all the characters to lowercase before tokenizing.

The next step is looking at parameters that would allow for more feature-selection in this pre-processing stage. The parameters I focused on are **ngram_range**, **tokenizer**, **max_df**, **min_df**, and

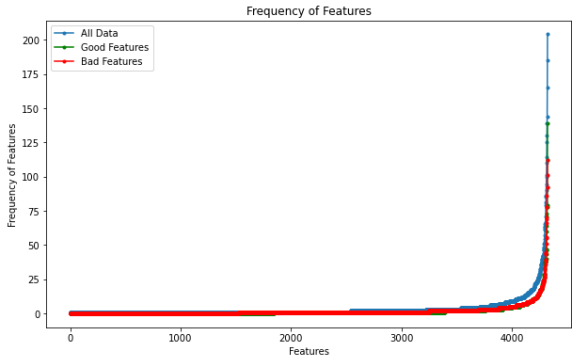
stop_words. The parameter **ngram_range** refers to the number of words to be grouped together when converting the text to tokens. To determine the best range, a comparison was made between (1,1) = unigrams, (2,2) = bigrams or two words, and (3,3) = trigrams or three words. The scores were calculated using Logistic Regression with default parameters. In the end, unigram had the best score for training and testing.

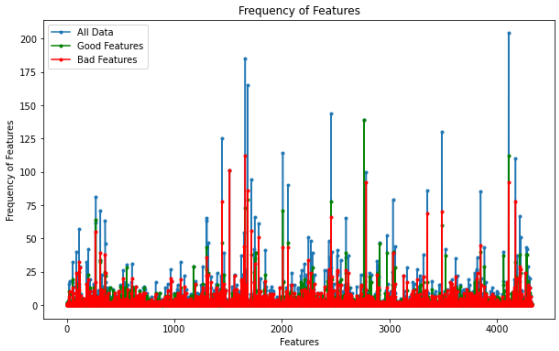
ngram_range Comparison Table		
	Training Score Average	Testing Score Average
Unigram	0.924375	0.79125
Bigram	0.959375	0.69583
Trigram	0.895208	0.55125

The **tokenizer** is important because it renames words to one singular new word. It also has the power to exclude very rare words or very common words by making lists of those words and equating them to a single word. A very rare word could be considered a word that only shows up one time in one review. While a very common word could be a word that shows up in all the reviews multiple times like the word “the” or “I”. In the case of good reviews, any word that is commonly used in good reviews and has a low frequency in bad reviews would be a great tokenizer. To help **tokenizer** common good words, I searched for a list of the most common words used in good reviews and wrote a function that would turn all of those words into the word ‘good’. Then I did the same search for bad review words and turned it into a function that would return the word ‘bad’ if it had been used in the review training dataset.

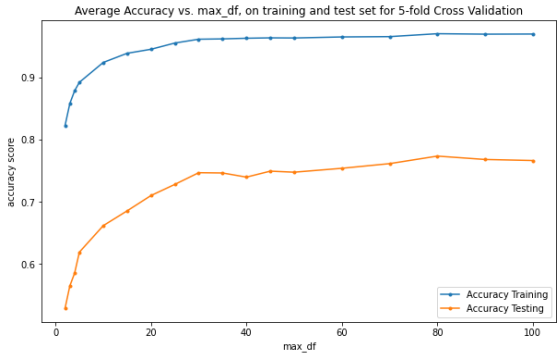
Another **tokenizer** list I used would change all the numbers to the string ‘num’. This reduced the number of features down to 3328. When I ran this, the accuracy scores were all the same except for the decision tree classifier model which had validation scores worse by 0.02.

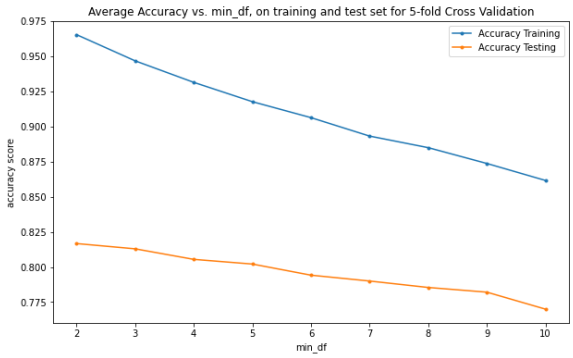
The next pre-processing task I looked at was feature-selection. It includes removing very common words or very uncommon words. This involves tuning 2 different parameters: the **max_df** and the **min_df**. The **max_df** parameter is key because it allows the **CountVectorizer** to ignore any terms that occur frequently in the dataset, for example: ‘the’, ‘and’, ‘is’, ‘he’, ‘she’, etc. To determine the features to keep, I plotted the frequency of the words appearing in all the reviews. The most common words used in the good review and also the bad review are shown below. The plot on the left is with the features sorted while the plot on the right is unsorted. You can see that in the sorted plot, 97.99% of the features have a frequency less than 25. These are charts using data that has parsed out anything that is not a noun, verb, adjective, etc. What is interesting in the sorted frequency feature plot is that there are more words that are of higher frequency in the good reviews versus the bad reviews. Bad reviews share more words with the good reviews it appears. But good reviews share less words (or features) with the bad reviews. You can see that there are words that have a higher frequency in good reviews versus bad reviews.





The plot below shows the different scores based on setting **max_df** and **min_df** to different values to help weed out any **stop_words** aka very common words or very rare words in the case of **min_df**. These plots are amazing because they show so much information. In the **max_df** plot, it shows that you start to plateau in your training and testing dataset with your accuracy scores when you hit **max_df** = 50. In the **min_df** plot, you see that the more words you cut out, the worse your accuracy score is for both testing and training data. These plots really helped me understand how these two parameters worked. It also took me some time to understand what the range of numbers should be so when I was finally able to get it to word I was really happy.





In the end, I set the **max_df** = 50 and kept the **min_df** at default which is 1.

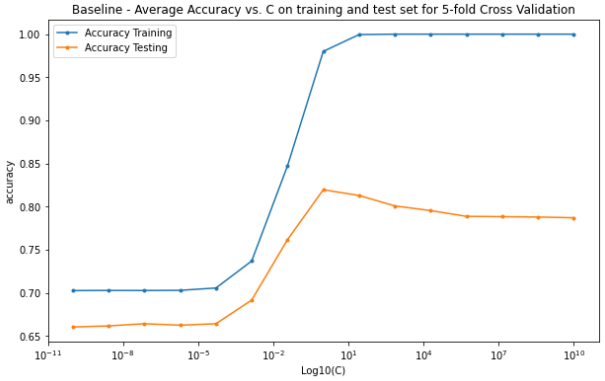
The last parameter to analyze is **stop_words**. This parameter is similar to **max_df** in that its objective is to ignore common words that would occur in both reviews and not aid in the process of classifying new data. The given list for **stop_words** is 'english' but when doing a search online, there were better lists that were recommended. I ended up using a list called "Long Stopword List". When I compared the results from using the 'english' list, the googled list, and no list, the two lists ended up having the same results and not having a list ended up having better results than both of them which was curious. I decided to use the googled stop word list since the description for the **stop_words** 'english' list looked problematic and it was better to have a **stop_words** list than not have one at all.

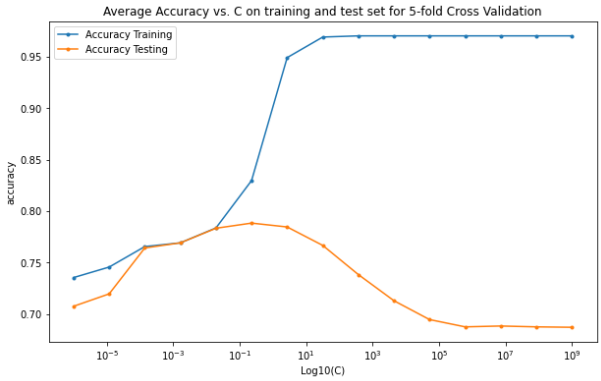
For comparison reasons, I wanted to try using the previous frequency results that were calculated, and see if I set **max_df** = 50 and **min_df** = 1, if that accuracy would perform better than incorporating a **stop_words** list. It performed worse when I traded out the **stop_words** list unfortunately. It was 0.1 worse on the testing data and 0.001 worse on the training data when it comes to a logistic regression model. (I did this test after creating the logistic regression model.) With this in mind, I decided to keep the **max_df** at default and use the custom **stop_words** lists instead.

Now that the pre-processing is done, there is a finished product for the Bag-of-Words feature representation which will convert the string of words into feature vectors. With the parameters set: **ngram_range** to (1,1), **tokenizer** set to three lists in total: two lists for each classifier and one for any numbers, **max_df** set to default, and **stop_words** set to a custom list. Throughout all these tests and analyses, the **random_state** was set to 1 to ensure consistent results. If I could do one more analysis, it would be to try out Term Frequency-Inverse document frequency (TF-IDF) to determine whether it performs better or worse than **CountVectorizer**.

2. Logistic Regression Model

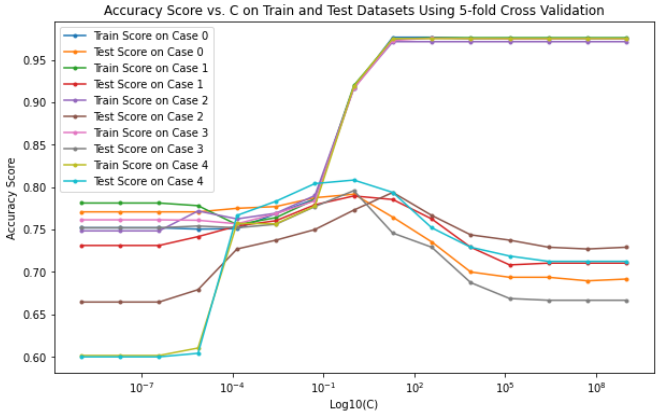
In this part of the project, the goal is to train a logistic regression model for feature data in order to classify training data. I used 5-fold cross-validation again and set the **random_state** to 1. To determine what the best complexity setting is, I chose to vary the **C** parameter and plot the results. Below are graphs that show the ideal setting for **C** = 1 which is also the default value. The plot on the left is closer to how an accuracy vs C plot would look since the training and testing data are almost aligned when they increase and plateau as **C** increases. The right plot is after the **CountVectorizer**'s parameter tuning is incorporated. I'm not sure why the testing data in the right plot decreases in accuracy at high values of **C**. A higher value in **C** would usually have the training and testing data act in parallel and increase until they plateau together. This is something to keep in mind.





Another parameter I checked out was **max_iter**. Since there is less data than features, I also chose to set **max_iter** to a higher iteration number of 10,000.

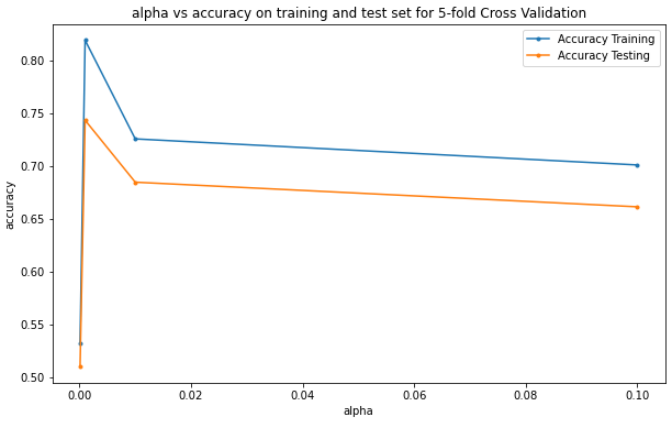
In the plot to the right, you can see how much the testing scores vary compared to the training scores. The training scores are much more consistent.



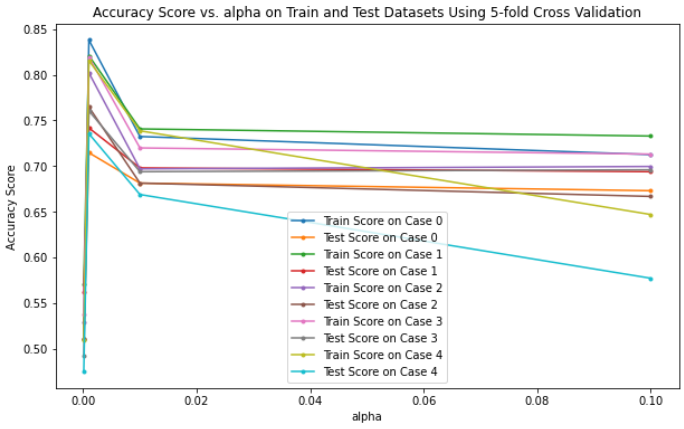
Just to be sure, I ran GridSearchCV on a range of parameters and values: {**penalty** = 'l1', 'l2', 'none'}, {**C** = 0.001, 0.01, 0.1, 1, 10, 100, 1000}, and {**solver** = 'liblinear', 'lbfgs'}. These were the best parameters recommended: **C** = 1, **penalty** = 'l2', **solver** = 'liblinear' which are the parameters I used in the end. This resulted in an average accuracy for the training data of 0.98 and average accuracy for the testing data of

0.82 when using the **CountVectorizer** with parameters set to their default values. But if the **CountVectorizer** has the parameters tuned then the scores decrease to training: 0.92 and testing: 0.799. This is also something to keep in mind since tuning the hyperparameters would ideally increase its accuracy and performance. I'm not sure why it would decrease the scores in this case. But my best bet is that I had overfit the data with my hyperparameter tuning in the **CountVectorizer**.

3. Neural Network (MLP) Model



The next model to generate is a neural network (or MLP) model. Generating this for the feature-data involves similar hyperparameter tuning. To speed up the process, **GridSearchCV** let me set a range of parameters to analyze over a default of 5-fold cross validation. I changed the cross-validation to 3-fold and set **n_jobs** = -1 because this calculation took the longest on my laptop. The hyperparameters I looked into were {**alpha** = 0.001, 0.01, 0.1}, {**activation** = logistic, relu, tanh}, and {**solver** = lbfgs, sgd, adam}. The best parameters were found to be **activation** = logistic, **alpha** = 0.001, **solver** = sgd. To see the visual representation of how the varying degrees of alpha affect the accuracy, I plotted the results of alpha vs accuracy as shown below. You can see that the training and testing data both peak at **alpha** = 0.001 and then the scores decrease because the regularization term is penalizing the complexity of the model. As it increases, it is supposed to discourage a complex, flexible, and overfit model. The plot below shows a similar pattern that is seen in most accuracy plots for training and validation data. The values increase and then plateau at a certain point together in parallel. I'm not sure why the data decides to jack-knife as soon as it reaches **alpha** = 0.001 but my best guess is that it could be because there were not enough values in the **alpha** range or that it picked up on some extra noise. I don't think it has to do with overfitting or underfitting since the training and testing scores both stay in parallel to each other. Since I used **GridSearchCV** to double check the results, I feel confident setting **alpha** to 0.001.



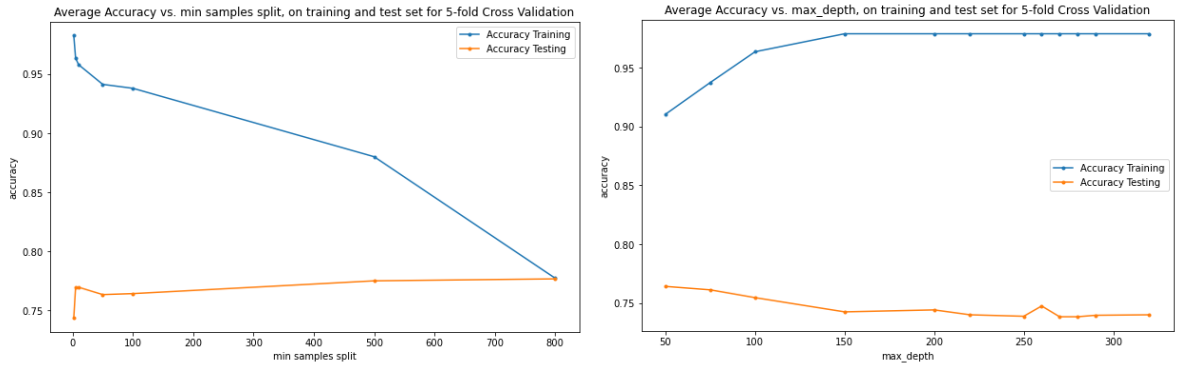
In the plot to the right, you see that the alpha score is pretty consistent throughout the 5 cross validation runs. This helps give confidence in the model knowing that they won't stray from the average. It looks like the last case does drop off a little but not a significant amount.

The MLP model's scored had an opposite affect compared to the logistic regression model in that the parameter tuning in the

CountVectorizer increased the scores. With no tuning, the scores are training: 0.53 and testing: 0.51. With tuning to the **CountVectorizer** led to training: 0.779 and testing: 0.778.

4. Decision Tree Classifier Model

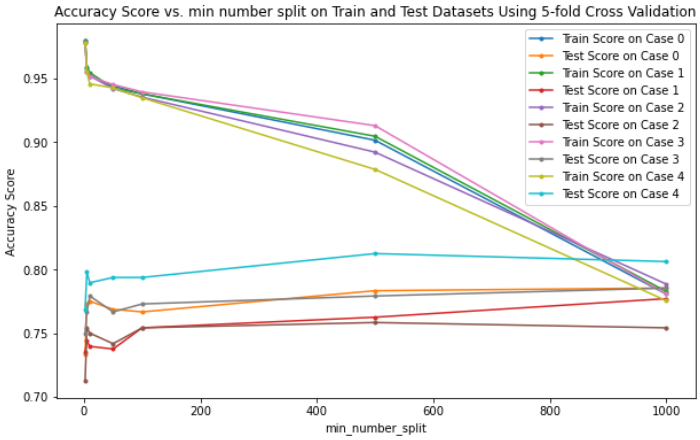
The last model I chose to generate of model of was a Decision Tree classifier. I chose this because I wanted to learn how to use sklearn's **DecisionTreeClassifier** function and I was curious how the results would compare to the other models. The hyperparameters I chose to vary the values of and plot were the **min_samples_split** and **max_depth**. The parameter **min_samples_split** is significant because it is the minimum number of samples required to split an internal node. The **max_depth** parameter is the maximum depth of the tree. Both of these are important in determining the complexity of the tree and it's shape. In the plot shown below, as the **min_samples_split** increases, the training accuracy decreases while the testing accuracy increases a little but mostly levels off. This makes sense because as there is an increase the number of values needed to be in a node before a split, then there would be an increase in the number of values required to be a leaf too. Which means the leaves will have more datapoints and may be less accurate in the case of the training data. I'm not quite sure why the validation data increases in accuracy as the **min_samples_split** increases. It is possible because the data is becoming too underfit as the **min_samples_split** increases compared to the data being too overfit when the **min_samples_split** is too small.



Continuing with the model, for the **max_depth** plot, as the values increase, the training accuracy decreases while the testing accuracy increases slightly and levels off. As the values increase, it appears that the data becomes underfit and when the values decrease, the data becomes overfit. There is a sweet spot for the depth of the tree where the model does not have to compromise the accuracy of the training data for the testing data. It shows that I need to be careful not to overfit or underfit the model. For the **min_samples_split**, I chose the value of 5 as the best possible way to increase the testing accuracy while not loosing too much of the training data accuracy. For the **max_depth**, the values start to plateau at 150. To reduce complexity, I chose the value of 200 though.

When I ran these parameters as they were tuned to the values calculated in the previous paragraph, the testing score was higher without the **max_depth** parameter by 0.01. The scores ended up being training: 0.968 and testing: 0.739 when the **CountVectorizer** had default parameters. And with parameters set to values, it was training: 0.949 and testing: 0.766.

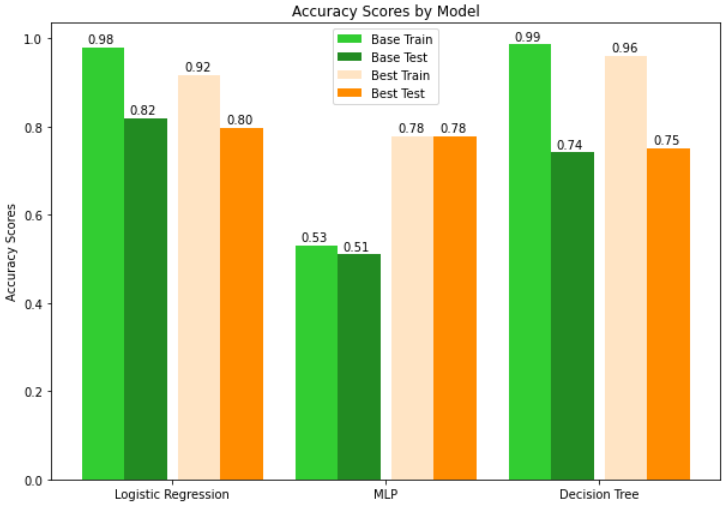
The plot on the right shows what the cross validation results look like for each of the runs. This pretty much matches what we saw in the previous plot of averages results for each of the 5 folds of the parameter, min_samples_split. We can see that the training data is very dense and close together while the testing/validation data is a little more spread out. This could be due to the fact that we have a better opportunity of getting good training data scores because we fit the model to that data.



5. Summarize

To summarize the findings, I'll step through the process of creating the models. Initially, **CountVectorizer** was used to create a Bag-of-Words model. This required a pre-processing phase where I looked at tuning hyperparameters until I had settings that increased the model's accuracy score to the best it could be. I learned that the parameters **strip_accents** and **lowercase** had default values that would naturally help with feature selection. Then I looked at several other parameters and gained insights from that analysis. A full analysis was run on parameters **ngram_range** and **max_df** which showed in the end that their default values would be the best choice. I looked two more parameters that took in lists of words and helped with more feature selection. These parameters were **tokenizer** and **stop_words**.

Then I took a look at three different classifier models. The first was a logistic regression model. In this model, I used **GridSearchCV** to determine the best hyperparameters. I also analyzed a chart of a range of **C** values to determine any oddities or overfitting. One of the plots showed a strange result that was unfortunately inconclusive. The next model generated was a neural net, the MLP model. This model was great because the **GridSearchCV** gave the best parameters for many values as well as a solid plot. This time, the graph was of the **alpha** value, another regularization term. This conclusive evidence made it easier to feel confident about the parameter settings. The last model was the Decision Tree classifier. Two analyses were done on two parameters: **min_samples_split** and **max_depth**. The plots helped determine the best value to set those parameters to as well as evidence of potential overfitting and underfitting.



The classifier model that was the best was the Logistic Regression model because it had the best training and testing accuracy score out of the three

models. The comparison between scores of the final three models are shown in the bar chart below. The "Base" model scores refer to default parameter tuning in the CountVectorizer while the "Best" model scores refer to parameter tuning. When I uploaded the results to the leaderboard on Gradescope, the best performing model was the logistic regression model (as shown in the table in section 6. Leaderboard). I learned that I should go off of the validation score or the halfway point between the training and validation score instead of just the highest accuracy of a single training set. A surprising part of this bar chart is that the MLP model performed better with the hyperparameter tuning done to the **CountVectorizer**. That was a happy surprise (because that doesn't show up in Part Two at all).

6. Leaderboard

The table below shows the results of the leaderboard for each of the models. The highlighted model, Logistic Regression – Baseline, was the best performing model. What is amazing about these results is that they match pretty well with the validation data's results. If I rank the models from best to worst in the leaderboard table, then they have the same ranking with the validation data's ranking results.

Table: Part One Leaderboard Results

		Error Rate	ROC	Score
Logistic Regression	Baseline	0.175	0.89106	3/3
	Parameter Tuning	0.20167	0.86526	2.8833
MLP	Baseline	0.445	0.57056	0.5514
	Parameter Tuning	0.235	0.8357	2.55
Decision Tree	Baseline	0.25833	0.74932	2.321
	Parameter Tuning	0.25333	0.77076	2.3684

Part Two: Classifying Review Sentiment with Word Embeddings

1. Generating Word Embedding Model

In part two of the project there is a focus on word embeddings. This time, each vocabulary word is replaced with a complex feature-vector. These feature-vectors or embedding vectors are given in an archived file full of 400,000 possible vocabulary words. These vectors are 50-values in length. We have the GloVe project at Stanford to thank for these pre-trained vectors. Using these vectors, the next step is to take all the vectors for each word in a review and combine them into one vector. I tried a few different ways of combining the vectors to see which one performs that best. At first, I summed them together which resulted in scores much lower than the BoW scores, as shown in the plots below. Then I tried averaging the vectors together since this would at least consider the length (number of words) of each of the reviews. Before getting too much into the scores, I started to investigate the pre-processing step.

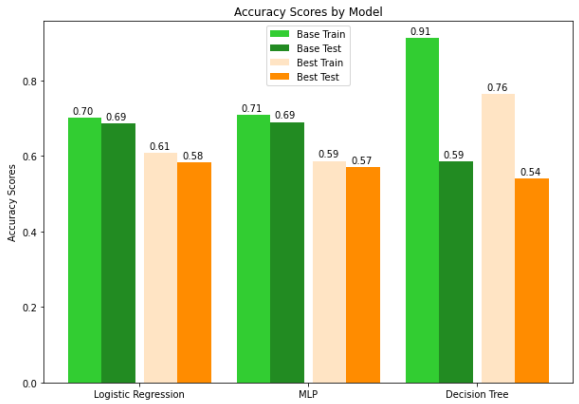


Figure 1: Summed Word Vectors

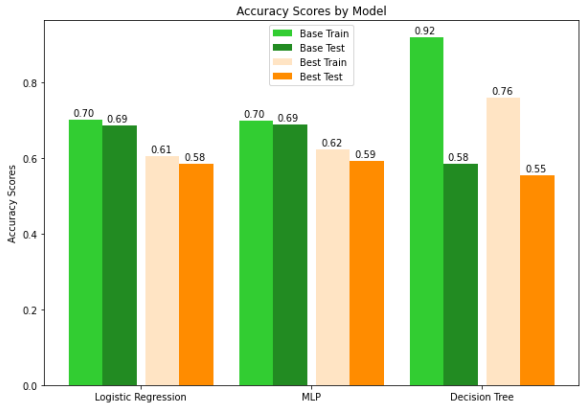


Figure 2: Averaged Word Vectors

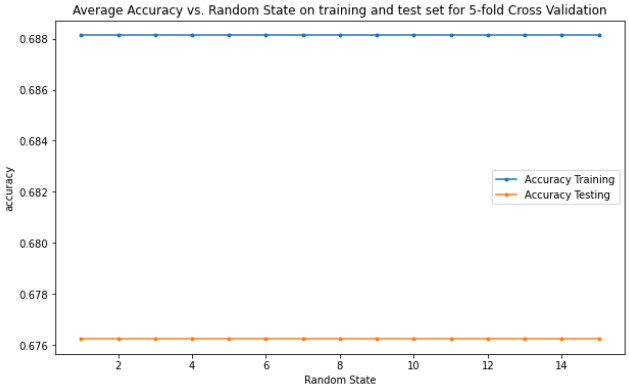
When it comes to pre-processing, I think it's important to keep repeats of words in a review because can be the difference between a neutral review and a easily classifiable review. The pre-processing feature selection in **CountVectorizer** can remove features like 'and', 'to', 'the', 'it', etc. which helps remove any noise and confusion. This can help refine the calculation. The last pre-processing step determining which feature-vectors to keep. I ran **CountVectorizer** with the same **stop_word** list and the same **tokenizer** as in Part One to help remove any very common or very rare words. The last step was removing any duplicates. I removed any words that were duplicates in a review in order to streamline the word vector and remove any noise.

Now getting back to the accuracy scores, I decided to calculate the models in order to determine full comparisons. All of the testing data accuracy scores were either better or the same as the BoW testing data accuracy scores. I decided to keep this as model after confirming the results.

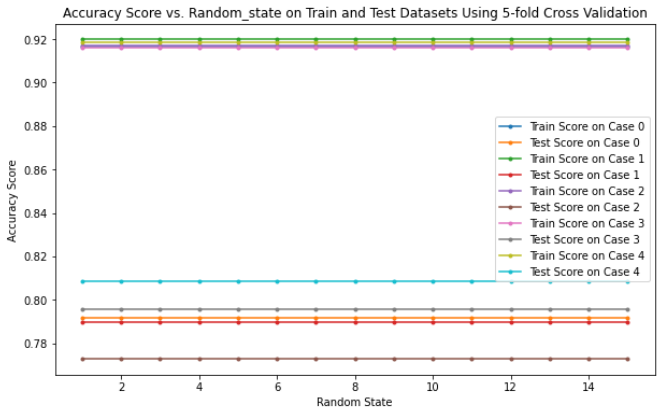
2. Logistic Regression Model

In this section of the project, I generated a logistic regression model for the feature-data and used it to classify the training data. To best determine the right hyperparameter settings, I used **GridSearchCV**

again. I set the parameters to the values as follow: {**penalty** = l1, l2, none}, {**C** = 0.001, 0.01, 0.1, 1, 10, 100, 1000}, and {**solver** = liblinear, lbfgs}. The best parameters were found to be **penalty** = l2, **C** = 1, and **solver** = liblinear. It turns out that the parameters penalty and C would be at their default values.



One of the ways I wanted to show how varying a hyperparameter affects the accuracy score is by plotting the results of varying **random_state** over a range of 1 to 15. The parameter **random_state** controls the random number generation for weights and bias initialization and train-test split sometimes. But the plot shows that the **random_state** did not actually affect the accuracy score at all. I was expecting a little variation but this has showed that the data is very uniform and consistent across all **random_state**s. To summarize, the parameters used in this model were the ones found from the grid search.

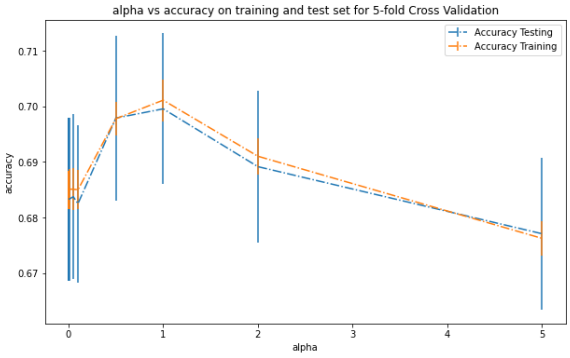


The plot on the right is great because it shows that while the training scores don't vary much between cross validation models, the testing score actually do vary by almost 0.4. This is interesting because it shows that depending on how the data is folded, we may get better validation scores. It seems this is a common theme for all the models and all the cross-validation plots produced for them. It's basically saying that we should take the validation results lightly because they can vary between the different 5 folds.

3. Neural Net (MLP) Model

Instead of using a logistic regression model, another option is to analyze how a Neural Net or MLP model will perform with this word embedding data. Similar to part one, I used **GridSearchCV** to help determine the best parameter settings. I input the following parameters and value ranges: {**alpha** = 0.001, 0.01, 0.1}, {**activation** = logistic, relu, tanh}, and {**solver** = lbfgs, sgd, and adam}. I set the **cv** = 3 and **n_jobs** = -1 this time to help speed up the process because it took longer to calculate this **GridSeachCV** compared to the logistic regression model. The best parameters that the function returned were **activation** = relu, **alpha** = 0.1, and **solver** = sgd. To help display the variation in accuracy score, I plotted **alpha** over a range of 0.001 to 5. You see to the right a smooth curve where the training and testing data stay parallel to each other. It is

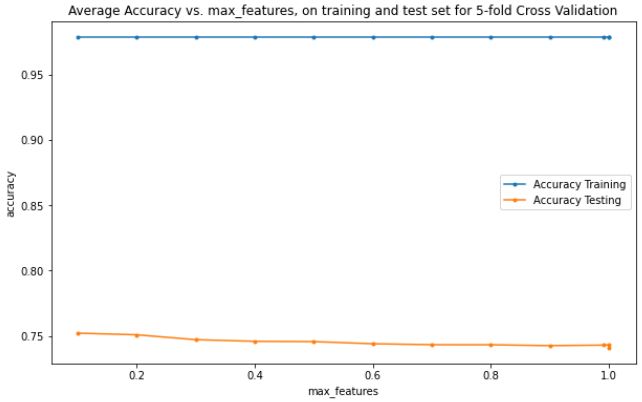
typical to see this because as **alpha** increases, so would the accuracy score since **alpha** is a regularization term that would penalize any complexity and overfitting. But as alpha gets too big, training should decrease while testing increases in accuracy. This plot does not show that possibly because the range was too narrow.



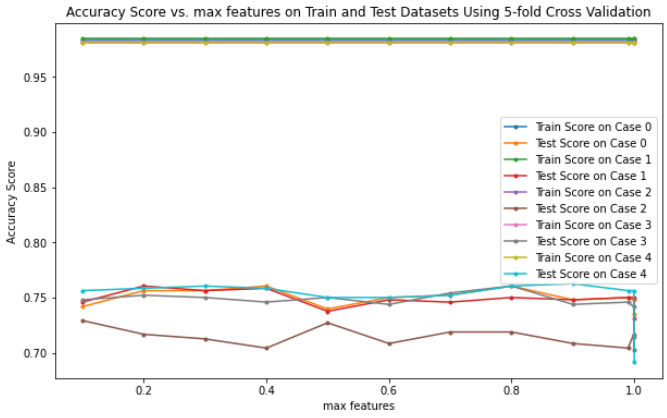
The model to the right is of the 5-fold cross validation for each of the folds. This graph shows the standard deviation bars which show that the testing scores have a larger range of values than the training bars. This means that the training data is denser while the testing data is a little more sparse in comparison.

4. Decision Tree Classifier Model

The last model to look at is the Decision Tree Classifier model. I chose this model so that I could have an easier time comparing results and determining where performs better. For this decision tree model I looked two new parameters that I was curious to learn more about. The first one is called **splitter**. The parameter **splitter** is supposed to be a parameter that chooses the best way to split a node but unfortunately the method of going about finding the best node was not explained. When this parameter is set to random it performed better than **splitter** = best. Possibly, the algorithm they use to choose the best split is not as accurate on larger data or data of this size. It's hard to tell without a better idea of how the algorithm works. When I run a 5-fold cross-validation model while only changing **splitter**, it shows that the default value (best) has the same training score as when **splitter** = random, but the 'best's test score was 0.005 worse than the **splitter** = random's test score.



For the parameter, I chose to analyze **max_features** over a range of 0.1 to 1. The parameter **max_features** puts an upper bound on the amount of features that can exist in the model. The plot shows that as the number of features increases to 100%, the testing accuracy decreases by 0.01 while the training accuracy stays the same up to the 17th decimal place. Based on these results, I kept this parameter set to it's default, 'None' which means it's equal to the number of features already in the model and not limited in any way.



The plot to the right is of the five folds of cross validation. It's similar to when we plotted random_state in that the training scores were all very close together with little variation while the testing scores had a little more variation.

5. Summarize

To summarize, Part Two of the project begins by implementing a word embedding model. This model still uses **CountVectorizer** to attain the list of features but instead creates a model based on the average of vectors included in the project. The difficult part of this was determining how to combine all the word vectors in a way that improves the accuracy score of the final model. Analysis showed that an average of the word vectors had better performance than a sum of them. I didn't have enough time to calculate the third type of word embed model which would have been the min/max vector. This vector would have taken the min and max values at each index of the vector and then calculated the distance between them. It would have been interesting to see because this could have shown a true relationship between the words in a review and possibly showed a pattern between bad reviews versus good reviews.

After forming the word embedding model, the next step is to train a classifier model with this data. The first model to be trained was a logistic regression model. This model **GridSearchCV** was used to determine the best hyperparameters. Then an analysis was done the parameter **random_state** to see if there was any variation in the accuracy score based on the **random_state** value. The analysis showed there was definitively no variation in the accuracy score.

After training the logistic regression model, the neural net MLP model was generated. The **GridSearchCV** function was used again to determine the best parameters. This helped to conclusively show the best values for the parameters of this model. After determining these, a plot was created to show the correlation between **alpha** and the accuracy score again. This was done on purpose a second time because the plot from Part One had an odd sharp drop in the beginning and I wanted to see if that would happen again to the same data in a different model. But this time, the plot of the average cross-validation was cleaner and easier to analyze thankfully. The second plot showing the five folds with standard deviation bars showed that there is a clear best value for **alpha** which was double checked with the grid search function.

Lastly, the Decision Tree model was created to train the data. This model showed that the parameter **max_features** was best set at it's default because the training data didn't change but the testing data improved slightly the more features were allowed.

The models below are when the **wordEmbed** is used to make a single 50 value vector that is the sum of all the words in the review. The green and orange plots just show Part 2's data while the blue and purple plots showed the overall data comparison for the whole project. The top two plots are when we calculate the word embed review vectors by summing up the individual word vectors. While the bottom two plots are when we calculate the word embed review vectors by averaging up the individual word vectors. There is an increase in score when averaging versus summing. There is better performance in training and testing for all the models when we don't tune the **CountVectorizer** classifier. I'm not sure why this is unless I chose words that are misclassifying the reviews. Ideally, I would use more time to play around with different lists to try to find the one that improves the score. Another trait of these plots to notice is that the training data is always performing better than the

testing data. This is a sign that the model could be overfitting to the training data and the alpha or C value should be increased.

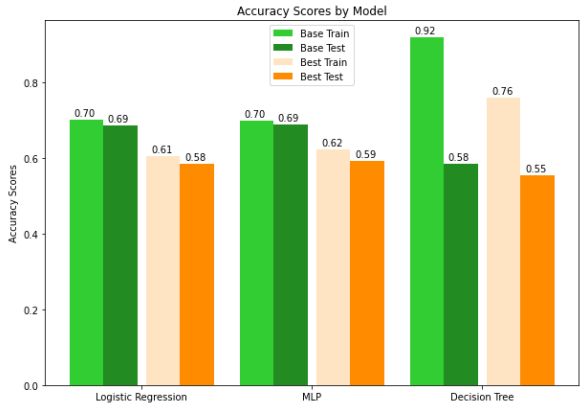


Figure 3: Using CountVectorizer

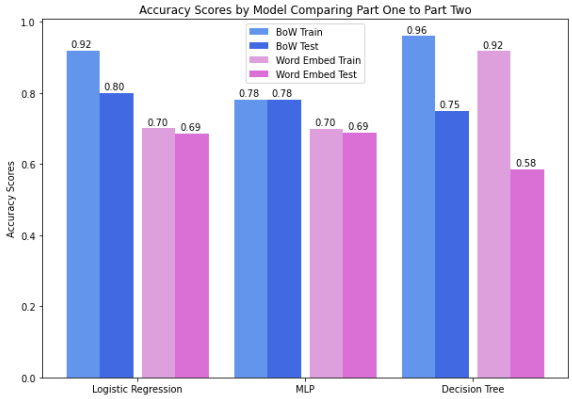


Figure 4: Using CountVectorizer

Since the results from these models were not spectacular, I thought I would try a last-ditch effort and switch all the **CountVectorizer**'s to **TfidfVectorizor**'s. But as the results show below, there was sadly no difference in the accuracy scores. Another attempt of increasing the validation accuracy score was by removing porter stemmer and using TfidfVectorizer in the pre-processing improved the hyperparamter-tuned scores by 0.03-0.05 but not the best scores of the two for each model. Another change I tried was fitting a new classifier, the **RandomForestClassifier**, increased the validation accuracy score to 0.67 compared to the Decision Tree's score of 0.58. But the change didn't affect the other two model's validation accuracy score. One more item I wish I could have tried is normalizing the data. That would have helped scale it to the same range and removed some of the noise.

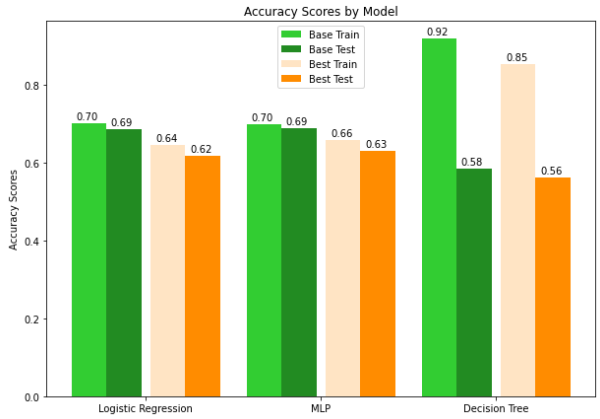


Figure 5: Using Tfidf without Porter Stemmer

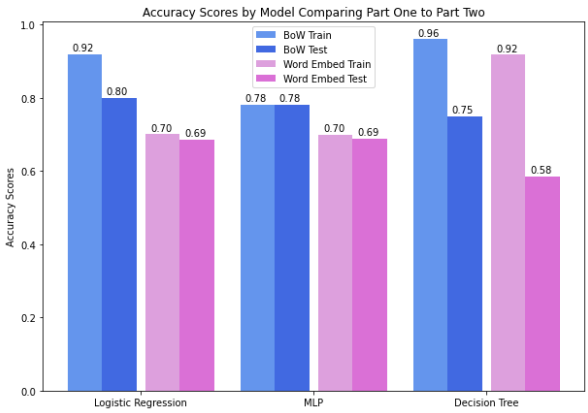


Figure 6: Using Tfidf without Porter Stemmer

6. Leaderboard

The leaderboard results showed that the best performing model was the MLP model when the **CountVectorizer** used default values on all its parameters. Based on the scores shown in the table in section 6. Leaderboard, I wish I would have tried to generate a model for SVM instead of a Decision Tree Classifier.

Table: Part Two's Leaderboard Results				
		Error Rate	ROC	Score
Logistic Regression	Baseline	0.34667	0.73837	2.1766/3
	Parameter Tuning	0.44	0.60054	0.87
MLP	Baseline	0.33833	0.73833	2.2937
	Parameter Tuning	0.43833	0.59918	0.8934
Decision Tree	Baseline	0.445	0.57426	0.8
	Parameter Tuning	0.44167	0.57731	0.8466