Mattia Danese
Project 2
5/7/21

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

Problem 1

In order to produce a fixed vocabulary, V, by use of the `CountVectorizer` library, I did a fair amount of preprocessing on the training data beforehand. I first looped over every document and used the `.lower()` function to make each alphabetical character lowercase. Next, I looped over every character of every document in the training data and removed certain characters from each document by checking each character's ASCII value. A character was kept in the document if its ASCII value pertained to a lowercase letter, number, space, dash, or apostrophe; otherwise, the character was discarded. Then, I used the `CountVectorizer()` function and passed in the, now processed, training data, and a list of the top 60 most common words of the English language which I found online to be used as the stop words. I decided to use the top 60 words, and not a bigger list, because I manually looked through it and concluded that it took away appropriate words that did not add value to classifying a document. Bigger lists, on the other hand, also included words like "good" and "bad" which, yes, are very common, but still hold value in classifying documents, so I wanted to keep them in the documents. All this resulted in making a fixed vocabulary, V, with 4435 words. This concludes how I made the fixed vocabulary, V, but I did one last modification to the training data arrays that would be used to train future models. I looked at `y_train.csv` and noticed that there were pretty big groups of 1's and 0's throughout the whole file. This could lead to pretty biased training data, even with cross-validation techniques, so I shuffled the training data to make the training data more mixed and less grouped together.

Problem 2

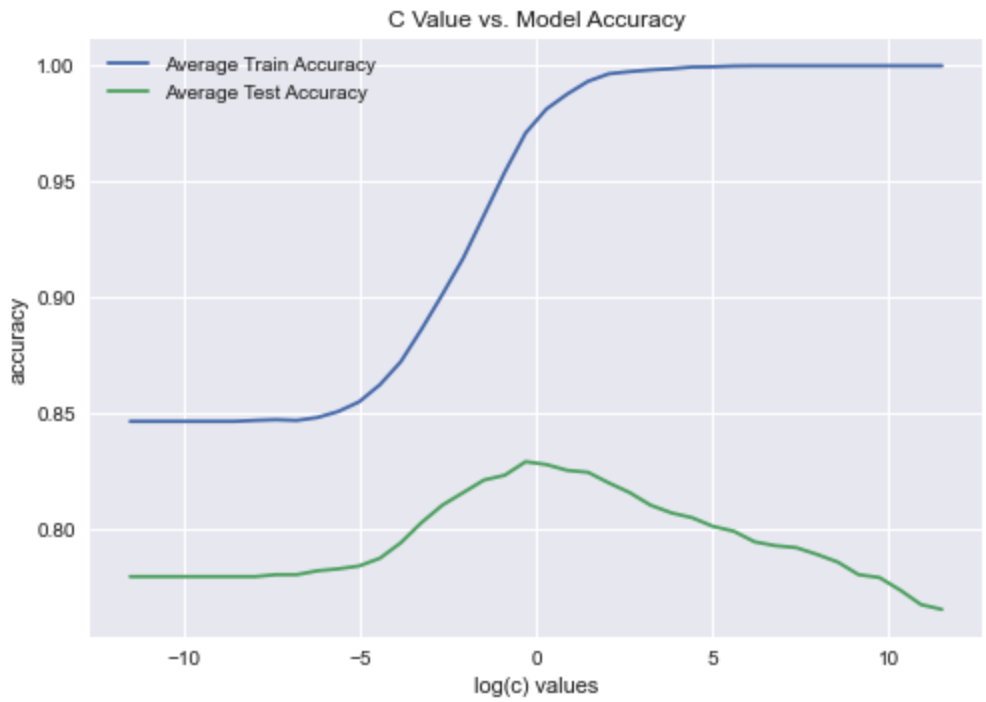
I made four different *logistic regression* models: a base model, a model with varying `c`, a model with varying `solver`, and a final model. I decided to make a base model to see how well the model performs with default parameters and to better gauge the performance of the models with varying parameters. The models with a varying parameter were chosen on the basis that they counteract overfitting best (`c`) or that they might optimize the model best given this specific type of input (`solver`). All models were trained and cross-validated (tested) by the `cross_validate()` function, where `k=5`. This function was very useful because it takes in the model and the training data and returns a dictionary with arrays for the testing and training errors across the `k` cross-validations.

```
Base Model:
Average Training Accuracy: 0.9770833333333334
Average Testing Accuracy: 0.8291666666666666
Testing Accuracy From Leaderboard: 0.88669
Testing Error From Leaderboard: 0.18667
```

Above are the testing accuracies of the base model across the 5 cross-validations performed, the average accuracy of all 5 cross-validation, and the accuracy and error on the actual testing data. These metrics will be compared with those of future models to indicate if updating the chosen parameters actually help the model.

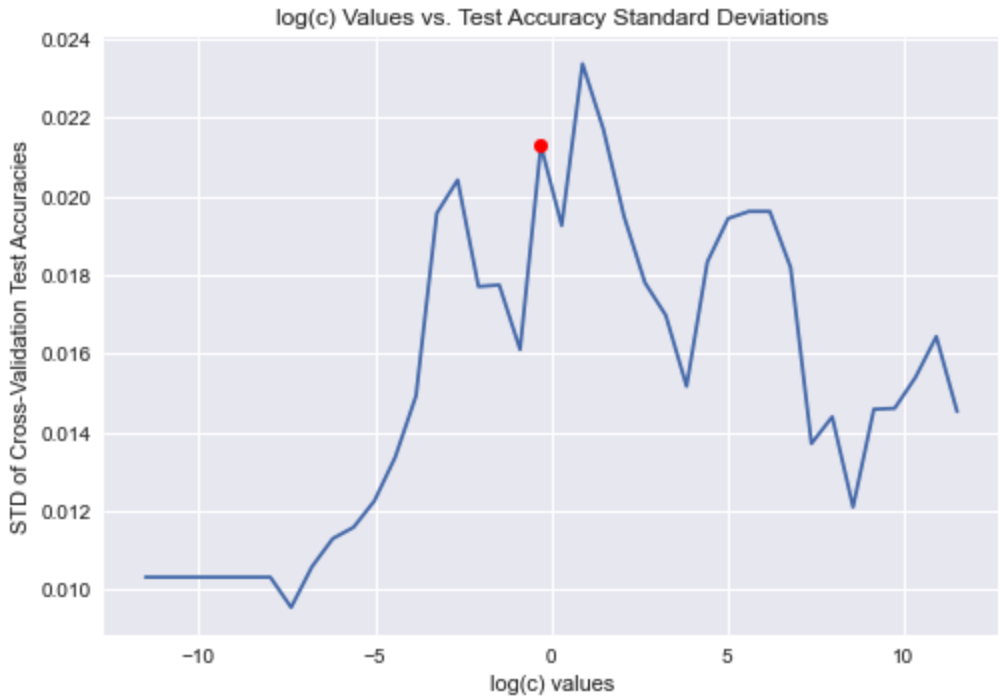
The first parameter I chose to vary is `c` because this parameter controls the regularization rate of the model, in other words how much stress is put on its complexity. I originally produced the `c` values that will be tested by running `np.logspace(-9, 6, 31)` which was given to us in previous assignments. The graph produced for the average testing accuracy was bell shaped around 0, so I thought it would be better to instead have more potential `c` values closer to and centered at 0, so I ran `np.logspace(-5, 5, 40)` to get the new potential `c` values and graphed the results.

```
C Value That Maximzes Average Test Accuracy: 0.7443803013251681
The Average Test Accuracy at Best C: 0.8291666666666668
```

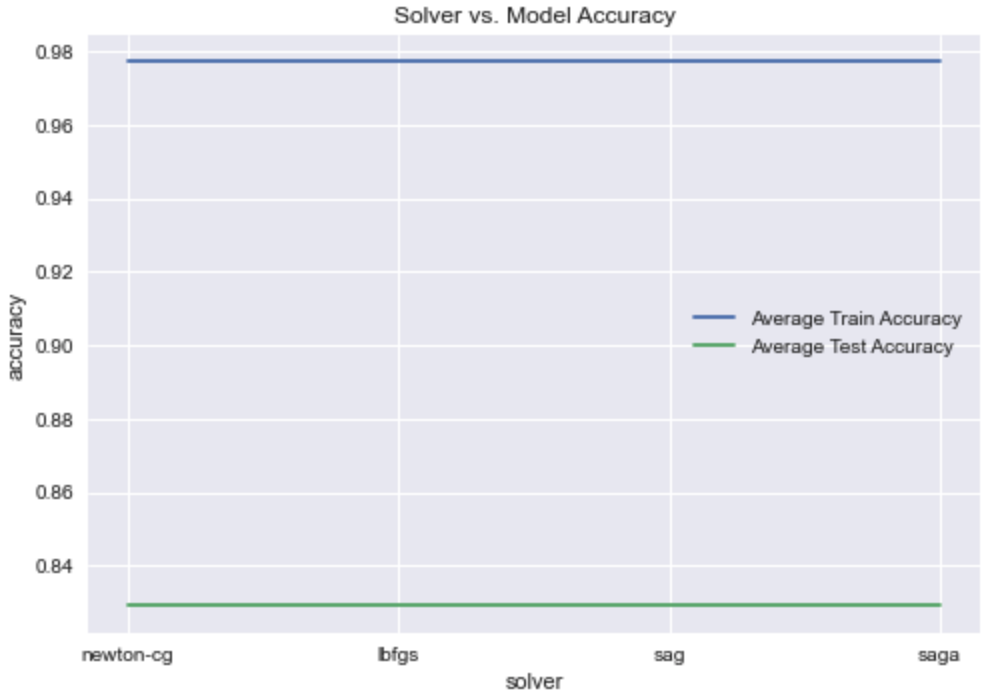


As the graph above shows, both average training and testing accuracies start out suboptimal for very low values of c, but then they both respectively increase. The average training accuracy resembles an x-cubed function, and quickly becomes overfit, while the testing accuracy resembles a bell shape; thus, I decided to use the c value that maximizes average test accuracy, about 0.744. It is important to note, however, that selecting a specific c value does not really produce a much better model relative to the base model. The training and testing accuracies of this model are either decently worse (underfitting or overfitting at extreme values of c) or just barely better than those of the base model.

The graph below of the testing accuracy standard deviations versus the log of the c values shows that there is a lot of variance for extreme values of c relative to c values closer in magnitude to 0. This supports the choice of 0.744 as the best c value because the variance of the testing accuracy across the k folds, at and around such a value, is very similar, so the true c value that maximized each fold is very probable to be close to 0.744.



The second parameter I varied for the *logistic model* is the model `solver`. We haven't seen this type of problem or input data before, so I was wondering if other `solver` types that we have not used in prior assignments could be better suited for this project. I made an array with all the solvers apart from the used in the base model (liblinear): newton-cg, lbfgs, sag, saga. Surprisingly, all the solvers had pretty much the same average training or testing accuracy, which is seen by the graph below, thus the `solver` parameter was deemed insignificant and the final model retained the `solver` of the base model.



Considering the two different parameters of the *logistic regression* model that I varied, I think it is safe to conclude that the `C` parameter is more important than `solver` because certain `C` values can dramatically decrease the accuracy of the model. In the grand scheme of things, however, none of the varied parameters had much, if any, positive effect on the model. It is also important to note that, with or without varied parameters, none of the models seem to be overfit. I made the final model by specifying the `C` value found before and left `solver` as it was in the base model (liblinear). Below are the metrics of the final model.

```
Final Model:
Average Training Accuracy: 0.9710416666666667
Average Testing Accuracy: 0.8291666666666668
Testing Accuracy From Leaderboard: 0.88702
Testing Error From Leaderboard: 0.19
```

The final model has slightly better metrics across the board than the base model, accuracies of the final model went up by about 0.01 and errors went down by 0.01. Specifying a `C` value during cross validation improved the model, but barely did, so it was expected that the final model would perform just slightly better than the base model.

Problem 3

I made five different *neural network* models: a base model, a model with varying `alpha` (regularization rate), a model with varying `learning_rate_init`, a model with varying `batch_size`, and a final model. I decided to make a base model to see how well the model performs with default parameters and to better gauge the performance of the models with varying parameters. The varying parameters were chosen on the basis that they can help fine tune the overall algorithm as they all deal with either model complexity or weight updating. Additionally, all models used the `relu` activation function as it is the least intricate activation function and, therefore, might help cut down the already hefty run-times of the models. All models were trained and cross-validated (tested) by the `cross_validate()` function, where `k=5`. This function was very useful because it takes in the model and the training data and returns a dictionary with arrays for the testing and training errors across the `k` cross-validations.

```
Base Model:
Average Training Accuracy: 1.0
Average Testing Accuracy: 0.7841666666666668
Testing Accuracy From Leaderboard: 0.86062
Testing Error From Leaderboard: 0.22167
```

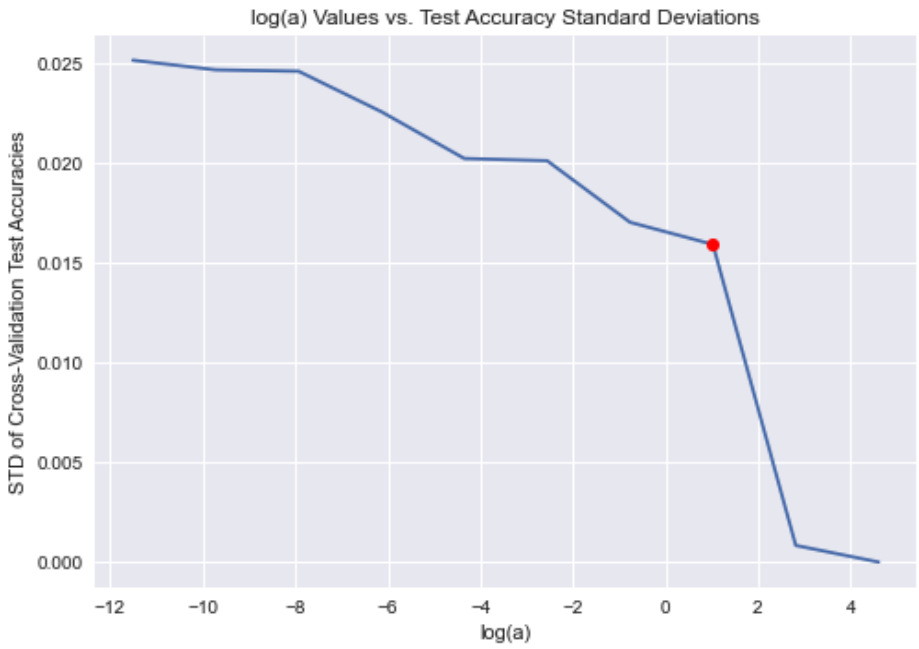
Above are the testing accuracies of the base model across the 5 cross-validations performed, the average accuracy of all 5 cross-validation, and the accuracy and error on the actual testing data. These metrics will be compared with those of future models to indicate if updating the chosen parameters actually help the model.

The first parameter I varied was `alpha` because it controls the L2 penalty of the model, in other words the model complexity. Since this is a pretty big project (the *neural network* model has 100 default hidden neurons, 2400 samples, 4435 features per sample), I thought it would be most beneficial to finetune the complexity of the model before doing anything else. Parameter `alpha` is usually a small number in magnitude, so I ran `np.logspace(-5,2,10)` and looped each value, below are my results.

```
The Best Alpha Value: 2.782559402207126
The Average Test Accuracy at Best Alpha: 0.8320833333333333
```

As shown by the graph above, I found the best `alpha` to be about 2.783 (where testing accuracy is greatest). The average test accuracy across the cross validations is significantly better than that of the base model. It is also a good sign that the training accuracy is no longer 1.0 when `alpha` is 2.783 as this means that the regularization rate not only helped in testing (made predictions more accurate) but also in training (made the model less overfit).

Below are the standard deviations of the `alpha` values across the 5 cross-validations. The red dot signifies the best `alpha` value found in the previous graph. This graph also helps to validate the best `alpha` value as it shows that the best regularization rate has the lowest standard of deviation before both testing and training accuracies plummeted. In other words, a regularization rate of 2.783 had the most consistent accuracies among the tested rates which still produced usable models.



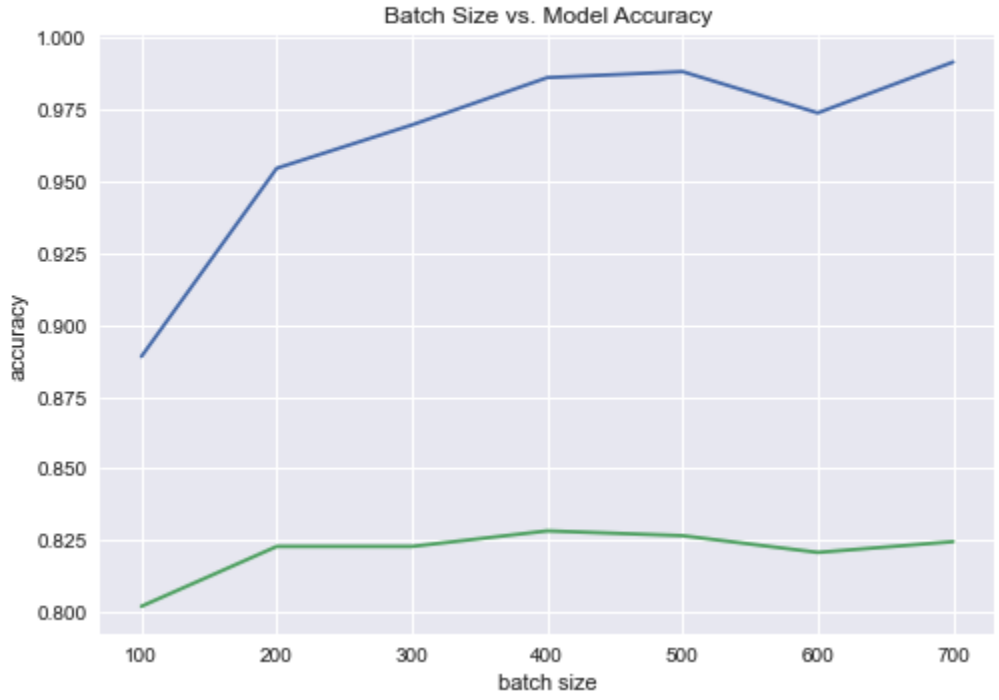
The second parameter I varied was `learning_rate_init` because it controls the step-size for weight updates and could definitely help not only get better performance but also faster runtimes. Similar to the prior regularization rates, I ran `np.logspace(-5,2,10)` to get an array of possible learning rates. I looped through them and got the following results below. It is important to note that this model also had the best regularization rate found previously.
(The blue line is Average Train Accuracy, the green line is Average Testing Accuracy)

```
The Best Learning Rate: 0.0021544346900318843
The Average Test Error at Best Learning: 0.8266666666666665
```

As the graph above shows, the model with the best `learning_rate_init` (and the best `a`) has a much greater testing accuracy across the 5 cross-validations than the base model, as expected, but surprisingly does slightly worse than the model with the just best `a` (and the default `learning_rate_init`). I suspect this is due to the fact that the best regularization rate was found by a model that also had a set learning rate, so the regularization rate is most likely optimal for models that have the exact same parameters (i.e. the default learning rate value).

The third parameter I varied was `batch_size` because, like the previous parameter, this controls how frequently the weights are updated and, again, could definitely help with performance and runtime. I looked at the documentation for the MLPClassifier and noticed that the default for `batch_size` is 200, with this in mind I made an array of `batch_size` values to be tested from 100 to 700 with a 100 step size. I looped through them and got the following results. It is important to note that this model also has the optimal `alpha` and `learning_rate_init` found previously.
(The blue line is Average Train Accuracy, the green line is Average Testing Accuracy)

```
The Best Batch Size: 400
The Average Test Error at Best Batch Size: 0.8283333333333334
```



The graph above shows that the best `batch_size` was found to be 400 and resulted in an average testing accuracy across the 5 cross-validations of 0.828. This average testing accuracy is still a lot better than that of the base model and slightly better than the model with the optimal `a` and `learning_rate_init`, but still less than the model with just the optimal `alpha`. Thus,

I varied `batch_size` again but now the model only has optimal `alpha` (and the default `learning_rate_init`) and got a slightly better average testing accuracy of 0.829583. I think this is still worse than the model with just optimal a for the same reason as before: the regularization rate was optimized when all other parameters were set to default values, so changing these parameters will not produce the same or a better result as the regularization rate is no longer optimized relative to the current model parameters.

Considering the three different parameters of the *neural network* model that I varied, I think it is safe to conclude that `a` is more important than `learning_rate_init` and `batch_size` on the basis that optimizing the regularization rate helped the model's performance the most. It is important to note that for high values of `a` and `learning_rate_init`, the model seemed very underfit (basically just guessing at that point) and for very low values of `a` the model seemed very overfit.

Even though the model with just the optimal regularization rate had the best average testing accuracy out of all the *neural network* models, I found that including all the optimal parameter values actually performed best on Leaderboard (still has a lower average testing accuracy than the model with just optimal `alpha`), thus the final model includes all the optimal parameters and its metrics are below.

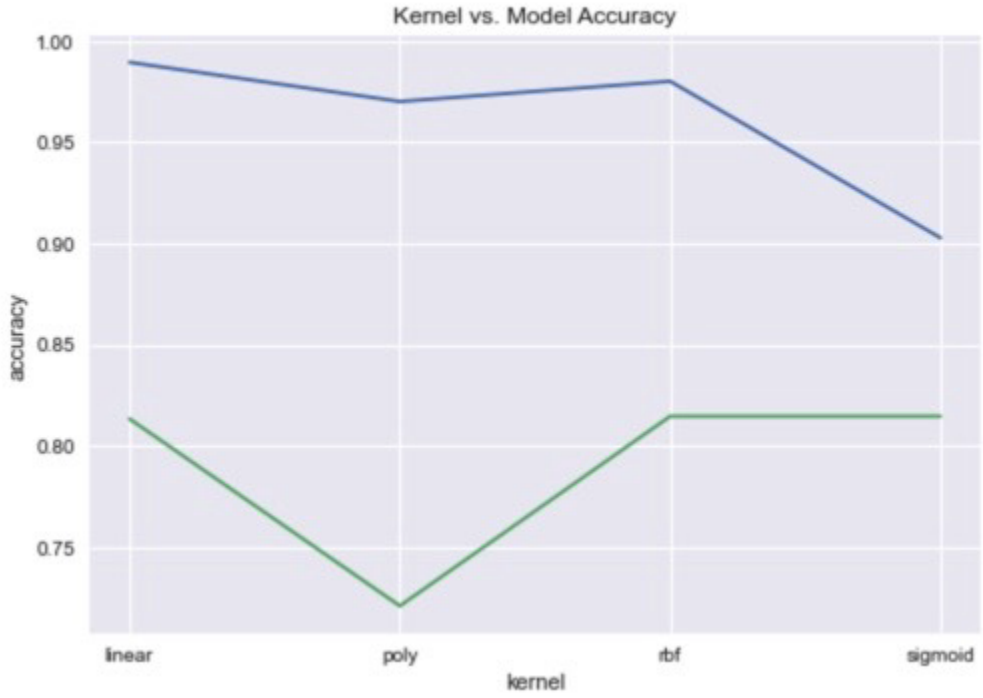
```
Final Model:
Average Training Accuracy: 0.9566666666666667
Average Testing Accuracy: 0.8320833333333333
Testing Accuracy From Leaderboard: 0.89113
Testing Accuracy From Leaderboard: 0.18
```

Problem 4

For the third model, I chose to make *SVM* Classifiers and vary the `kernel` and `C` parameters. The reasoning for varying `kernel` is that we never worked with these models in a previous assignment before, so I was a bit unfamiliar on how the different kernels operated. Additionally, I chose to vary the C parameter because it controls model complexity and could help with model performance and runtime. All the models that were built used `random_state=0`, leaving all other parameters as their fault values, and were trained and cross-validated (tested) by the `cross_validate()` function, where `k=5`. This function was very useful because it takes in the model and the training data and returns a dictionary with arrays for the testing and training errors across the `k` cross-validations.

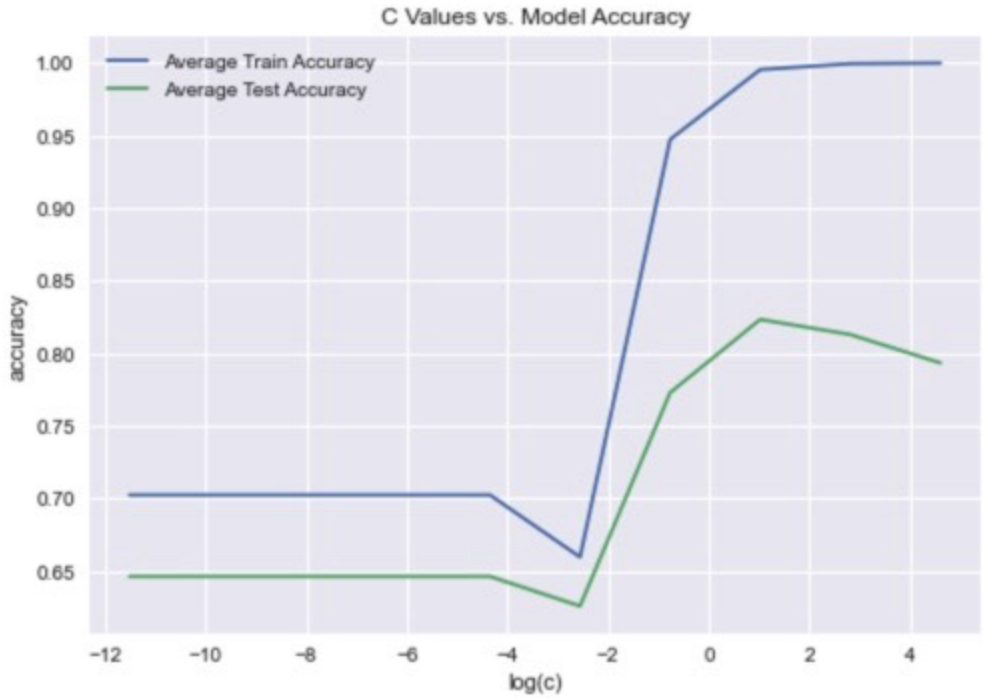
The first parameter I varied was `kernel` as I needed to figure out which kernel I would use before I optimize any other parameter. I made an array with all the possible kernel values, looped over them, and plotted their training and testing accuracies over the 5 cross-validations performed. According to the graph below, the rbf kernel produced the greatest average test accuracy. Even though the linear kernel had very similar results to the rbf kernel, I decided to continue with the rbf kernel because more parameters of the *SVM* Classified applied to it rather than the linear kernel.
(The blue line is Average Train Accuracy, the green line is Average Testing Accuracy)

```
The Best Kernel: rbf
The Average Test Error at Best Kernel: 0.8145833333333332
```



The second parameter I varied is the regularization parameter, `C`. Much like the *neural network* model, I made an array of possible `C` values by running `np.logspace(-5,2,10)` and then looped through them. Below are my results.

```
The Best C Value: 2.782559402207126
The Average Test Error at Best C Value: 0.8233333333333333
```



As the graph above shows, the regularization value that produced the greatest average test accuracy across the 5 cross-validations performed is 2.783. For very low values of `C`, the model is visibly underfit while for very high values of `C`, the model is visibly overfit; thus, it makes sense for the optimal regularization rate to have a "neutral" magnitude in the lower single digits.

The validity of this optimal regularization rate is also supported by the standard deviations of the test accuracies across the 5 cross-validations performed. As seen in the graph below, the optimal regularization rate is the global minimum of the graph; this means that no other regularization rate tested produced more consistent results.

Considering the two different parameters of the *SVM* Classifier model that I varied, I think it is safe to conclude both `kernel` and `C` helped model performance. It is important to note that the poly kernel did significantly worse in average testing accuracies and the sigmoid kernel did significantly worse in the training accuracies across the 5 cross-validation performed. Additionally, very small or very large regularization rates produced underfit and overfit models, respectively.

The final model that I built included both optimized parameters found previously. This particular model had the greatest average training accuracy across the 5 cross-validations performed and also performed the best on the Leaderbaord. The metrics of the model are below.

```
Final Model:
Average Training Accuracy: 0.9851041666666667
Average Testing Accuracy: 0.8300000000000001
Testing Accuracy From Leaderboard: 0.9083
Testing Error From Leaderboard: 0.18
```

Problem 5

Out of the three classifiers that I built, the final model *SVM* Classifier produces the greatest average testing accuracy across the 5 cross-validations performed. One of the main reasons why I think this model performed the best is because *SVM* Classifiers are usually insensitive to

outliers of data and there are probably many words that, across all 2400 documents, only appear a handful of times. Additionally, *SVM* Classifiers are known to give robust class boundaries to separate two classes which would definitely be helped for the documents that the classifier is less "sure" about. At the same time, though, this could be the reason why the classifier misclassified some of the documents that had a prediction probability close to the threshold/boundaries. Some documents that the classifier may have been less "sure" about might have just happened to be on the wrong side of the robust boundaries, leading the classifier to declare them part of the wrong class.

Problem 6

The best classifier from the previous steps (the final *SVM* Classifier) performed pretty well on the Leaderboard. Both the *SVM* Classifier and the *neural network* model had the same error rate (0.18) but the *SVM* Classifier had a slightly better testing accuracy (0.9083) than the *neural network* (0.89113). This was slightly unexpected because even though the *SVM* Classifier performed the best across the cross-validations performed, *neural network* models tend to be very good at generalizing over new data due to their internal neurons. Additionally, the robust *SVM* Classifier boundaries previously mentioned in Problem 5 I would expect be better for scenarios where most data are clearly on one side of the boundary line or the other, but in this case there is a lot of room for ambiguity. The logistic regression model was just barely the worst with an error rate of 0.19.

Part Two: Classifying Review Sentiment with Word Embeddings

Problem 1

The first step I took in the feature-generation pipeline was to preprocess all the training and testing data. Just like in Part One, I first looped over every document and used the `.lower()` function to make each alphabetical character lowercase. Next, I looped over every character of every document in the training data and removed certain characters from each document by checking each character's ASCII value. A character was kept in the document if its ASCII value pertained to a lowercase letter, number, space, dash, or apostrophe; otherwise, the character was discarded. After this, I looped over every word of every document and took out any words that were also included in my *stop words* list. My second step was to make a dictionary from the *glove* data. I looped over each line of the *glove* file, stored the word on each line as a key to the dictionary, and stored the accompanying array as the corresponding value in the dictionary. I did this for all 400,000 words. Next, I looped through each, now preprocessed, document, used each word in the document as a key inquiry to the *glove* dictionary, and averaged the embedded vector of each word to create one final embedded vector for the entire document. I decided to average all the word embedded vectors into one document embedded vector for two main reasons: words are represented equally (no words overshadow others) and so that each document had the same number of features (this would make fitting and predicting a lot easier). I additionally allowed for multiple copies of the same vector to be included in the averaged vector (in the case a word appears more than once in a document) under the assumption that a word that is repeated is probably significant to the overall sentiment/meaning of the document.

Problem 2

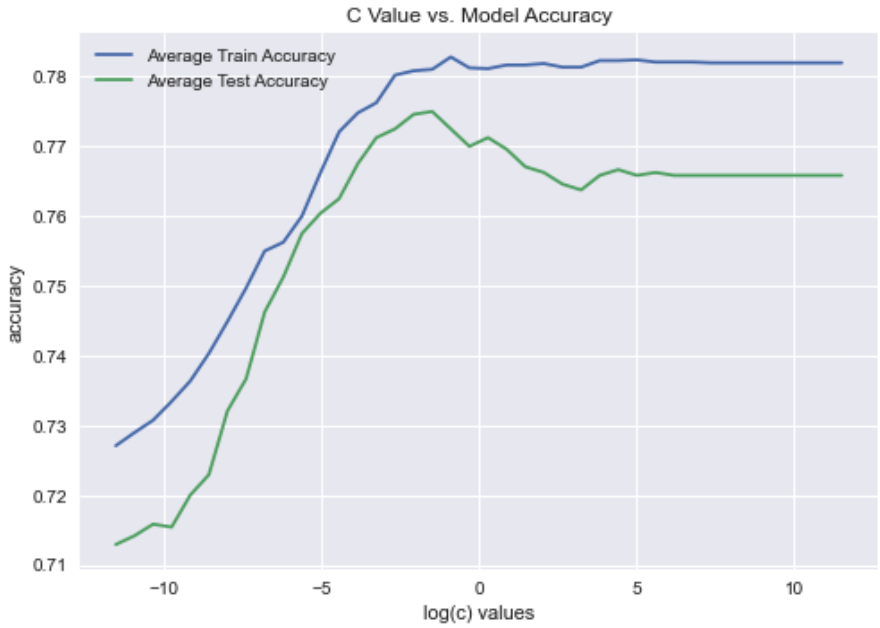
I made two different *logistic regression* models: a base model and a model with varying `c`. I decided to not vary the `solver` parameter, like I did in Part One, as all the different types of solvers produced very similar results, there was no clear winner. I decided to make a base model to see how well the model performs with default parameters and to better gauge the performance of the model with a varying `c`. I chose to vary `c` because, much like in Part One, this parameter regulates the complexity of the function and could reduce overfitting. All models were trained and cross-validated (tested) by the `cross_validate()` function, where `k=5`. This function was very useful because it takes in the model and the training data and returns a dictionary with arrays for the testing and training errors across the `k` cross-validations. Additionally, all models had the `random_state` parameter set to 0 in order to reproduce the same randomness each time.

```
Base Model:
Average Training Accuracy: 0.7814583333333334
Average Testing Accuracy: 0.7704166666666665
Testing Accuracy From Leaderboard: 0.82939
Testing Error From Leaderboard: 0.25167
```

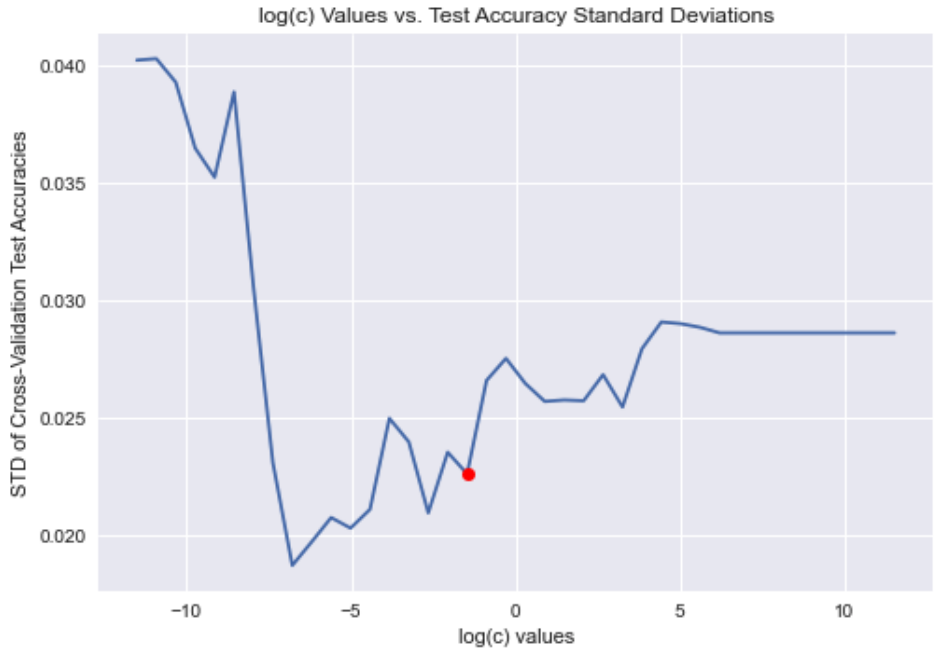
Above are the testing accuracies of the base model across the 5 cross-validations performed, the average accuracy of all 5 cross-validation, and the accuracy and error on the actual testing data. These metrics will be compared with those of future models to indicate if updating the chosen parameters actually help the model.

In order to variate c, I ran `np.logspace(-5, 5, 40)` to get a list of regularization rates to test. I noticed that having just 50 features per word made the model run very fast, compared to Part One, so I was able to test 40 such values. The results of this model can be seen in the graph below. It is evident that as the regularization rate increased in magnitude, the average training and testing accuracies increased with it. But around a rate of near 1, average training accuracy seems to even out while average testing accuracy decreases slightly and then it, too, evens out. Since the average training accuracy is modestly just above 0.78 at its maximum, I do not think that the models became overfit as the regularization rate increased. This is also supported by the average testing accuracy never plummeting down.

```
C Value That Maximzes Average Test Accuracy: 0.22854638641349884
The Average Test Accuracy at Best C: 0.7750000000000001
```



Below are the standard deviations of the `c` values across the 5 cross-validations. The red dot signifies the best `c` value found in the previous graph. This graph also helps to validate the best `c` value as it shows that the best regularization rate has a relatively low standard of deviation which shows that the higher training and testing averages for this particular regularization rate were rather consistent.



The final model that I built included just the optimized regularization rate previously found and kept all other parameters as their default values. The metrics of the model are below.

```
Final Model:
Average Training Accuracy: 0.7811458333333334
Average Testing Accuracy: 0.7704166666666665
Testing Accuracy From Leaderboard: 0.82947
Testing Accuracy From Leaderboard: 0.25667
```

Problem 3

I made four different *neural network* models: a base model, a model with varying `alpha` (regularization rate), a model with varying `batch_size`, and a final model. I decided to not vary learning_rate because, in Part One, it actually made cross validation accuracies worse, so I skipped it all together. I decided to make a base model to see how well the model performs with default parameters and to better gauge the performance of the models with varying parameters. The varying parameters were chosen on the basis that they control two crucial components of the model: complexity and weight updating Additionally, all models used the `relu` activation function for the same reason stated in Part One. All models were trained and cross-validated (tested) by the `cross_validate()` function, where `k=5`. This function was very useful because it takes in the model and the training data and returns a dictionary with arrays for the

testing and training errors across the `k` cross-validations. Lastly, all models had the `random_state` parameter set to 0 in order to reproduce the same randomness each time.

```
Base Model:
Average Training Accuracy: 0.9345833333333333
Average Testing Accuracy: 0.75
Testing Accuracy From Leaderboard: 0.86151
Testing Error From Leaderboard: 0.22
```

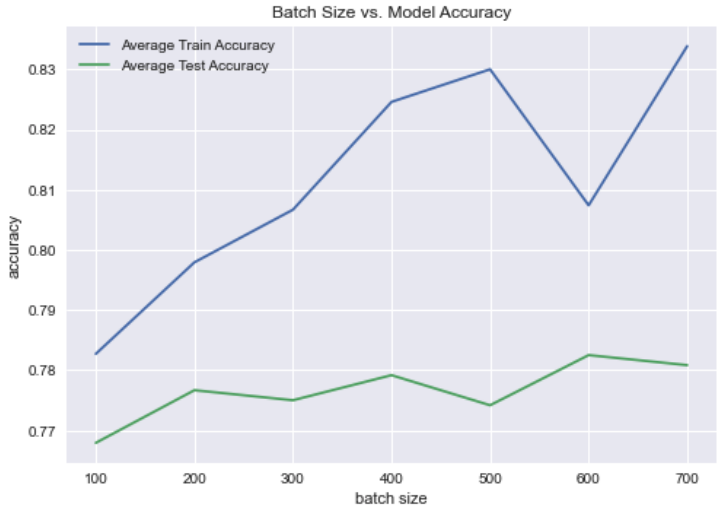
Above are the testing accuracies of the base model across the 5 cross-validations performed, the average accuracy of all 5 cross-validation, and the accuracy and error on the actual testing data. These metrics will be compared with those of future models to indicate if updating the chosen parameters actually help the model.

The first parameter I varied was `alpha` because it controls the L2 penalty of the model, in other words the model complexity. Even though there are not nearly as many features per word as in Part One, I felt it was still important and beneficial to finetune model complexity before all else. I ran `np.logspace(-5,2,20)` to make an array of possible `a` values, looped over each `a`, and the graph of the results is below. As the graph shows, the `alpha` value that produced the greatest average testing accuracy across the 5 cross-validations performed was1.438. The graph also clearly shows that for values low in magnitude, the model isn't really affected (average training and testing accuracies vary very little), and for values high in magnitude, both accuracies drastically decrease. The model doesn't seem to become overfit for any tested value of `alpha` but becomes extremely underfit as `alpha` increases. It should be noted that this graph is very similar to the corresponding graph in Part One even though word embeddings were used instead of making a fixed vocabulary . It makes sense that they look similar because it is still the same data being trained (and tested on), though the average testing accuries of Part One are a bit higher.

```
The Best Alpha Value: 1.438449888287663
The Average Test Accuracy at Best Alpha: 0.7754166666666668
```

The third parameter I varied was `batch_size` for similar reasons stated in Part One, it controls how frequently the weights are updated. Using the same rationale as in Part One, I made an array of `batch_size` values to be tested from 100 to 700 with a 100 step size. I looped through these values and got the following results. It is important to note that this model also has the optimal `alpha`.

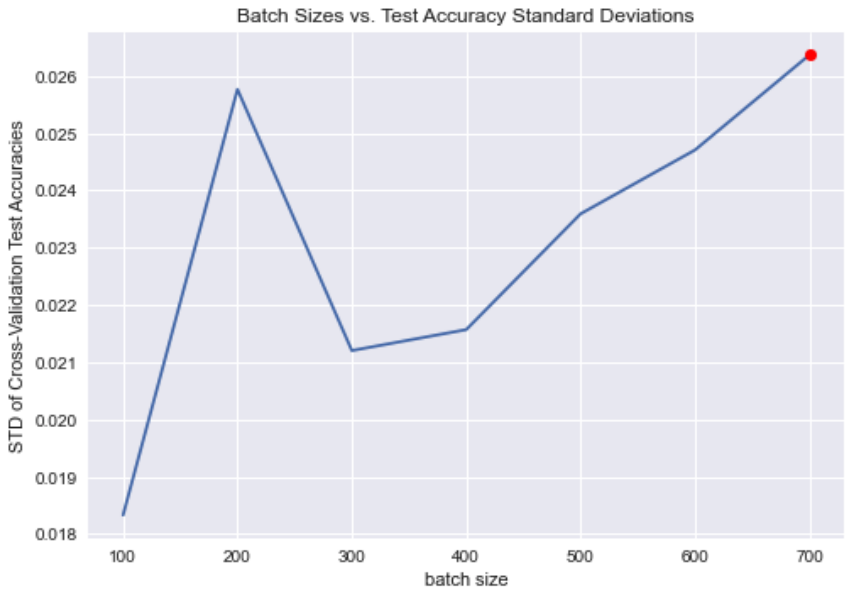
```
The Best Batch Size: 600
The Average Test Error at Best Batch Size: 0.7825
```



The graph above shows the results of varying `batch_size`. As shown, a size of 600 produces the greatest average testing accuracy across the 5 cross-validations performed. Due to how low average training accuracy is at this size, however, I chose instead to go with a size of 700 as this size produced the second highest average testing accuracy and the highest training accuracy. It is a bit puzzling that a size of 700 produces the highest average training accuracy and second highest average testing accuracy because a size this large would mean that weights would only

be updated 3 times (since there are only 2400 samples); I would think that so few updates would produce pretty bad accuracies.

The validity of this optimal `batch_size` is questioned a bit by the below graph of the standard deviations of the test accuracies across the 5 cross-validations performed. As seen in the graph below, the optimal `batch_size` (red dot) has the highest standard deviation which means that, relative to the other sizes, its accuracies across the 5 cross-validations were not very consistent.



The final model that I built included both optimized parameters previously found and kept all other parameters as their default values. The metrics of the model are below.

```
Final Model:
Average Training Accuracy: 0.8339583333333334
Average Testing Accuracy: 0.775
Testing Accuracy From Leaderboard: 0.85814
Testing Accuracy From Leaderboard: 0.225
```

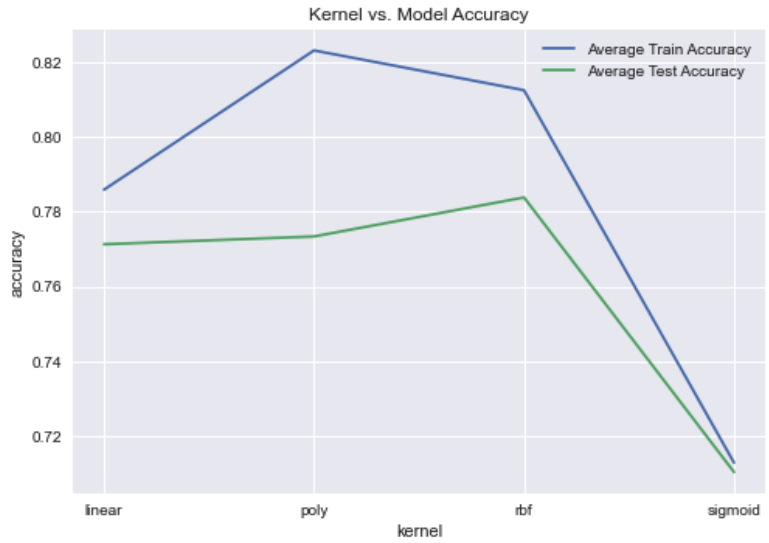
Problem 4

For the third model, I chose to make *SVM* Classifiers and vary the `kernel` and `C` parameters just like in Part One. I chose to vary `kernel` again because I am still not very familiar with the different kernels and I am interested to see if word embeddings would have a certain effect on a particular kernel. Similar to the previous problems, I chose to vary the `C` parameter because it controls model complexity and optimizing it could definitely help with model performance and

runtime. All the models that were built used `random_state=0`, leaving all other parameters as their fault values, and were trained and cross-validated (tested) by the `cross_validate()` function, where `k=5`. This function was very useful because it takes in the model and the training data and returns a dictionary with arrays for the testing and training errors across the `k` cross-validations.

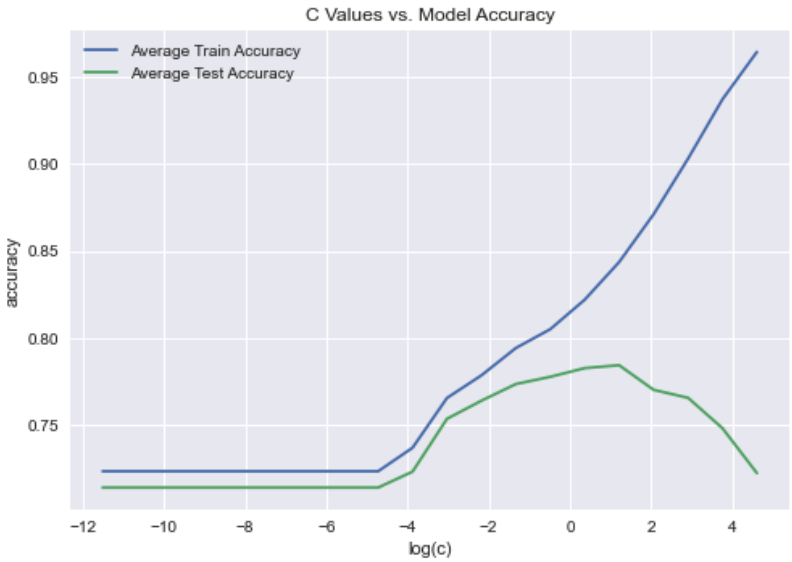
I varied `kernel` the same way I did in Part One: I made an array with all the possible kernel values, looped over them, and plotted their training and testing accuracies over the 5 cross-validations performed. According to the graph below, the rbf kernel produced the greatest average test accuracy while the poly kernel produced the greatest average train accuracy (and is not overfit). I was unsure on which kernel to go with and questioned if sacrificing a little generalization for precision (poly) was better than being a bit more general but less precise (rbf). In the end, I reasoned that the goal of optimizing a model is to make it be able to generalize on new data, so I decided to go with the rbf kernel.

```
The Best Kernel: rbf
The Average Test Error at Best Kernel: 0.78375
```

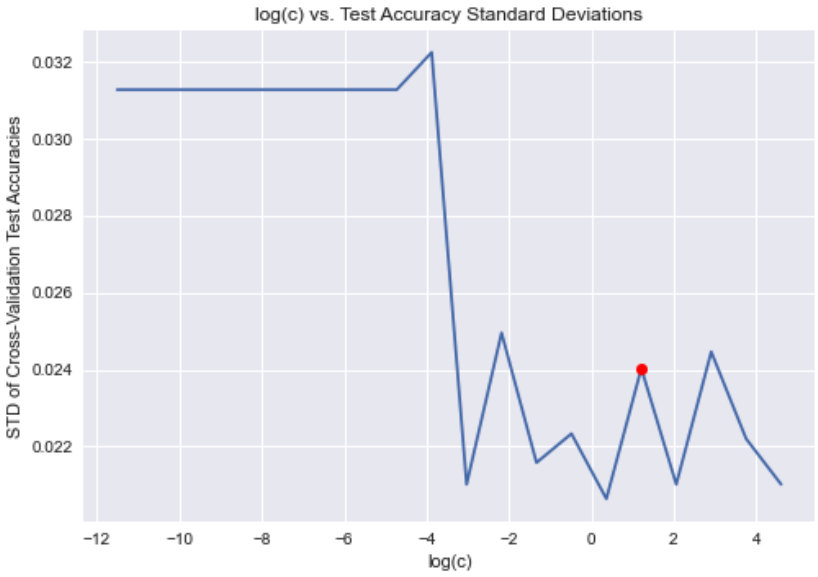


The second parameter I varied is the regularization parameter, `C`. I again made an array of possible `C` values by running `np.logspace(-5,2,20)` and then looped through them. According to the graph below, the `C` value that produced the greatest average testing accuracy across the 5 cross-validations performed was 3.3598. It is evident that for `C` values low in magnitude, the training and testing accuracies stay relatively constant and for `C` values too big in magnitude the model becomes overfit and the average training accuracy shoots up while the average testing accuracy decreases.

```
The Best C Value: 3.359818286283781
The Average Test Accuracy at Best C Value: 0.7841666666666667
```



Below are the standard deviations of the `C` values across the 5 cross-validations. The red dot signifies the best `C` value found in the previous graph. This graph also helps to validate the best `C` value as it shows that the best regularization rate has a relatively low standard of deviation. In other words, a regularization rate of 3.3598 produced consistent accuracies relative to the other tested rates.



The final model that I built included both optimized parameters previously found and kept all other parameters as their default values. The metrics of the model are below.

```
Final Model:
Average Training Accuracy: 0.8104166666666666
Average Testing Accuracy: 0.7770833333333333
Testing Accuracy From Leaderboard: 0.85575
Testing Error From Leaderboard: 0.225
```

Problem 5

Out of the three classifiers that I built, the final model *SVM* Classifier produces the greatest average testing accuracy across the 5 cross-validations performed. One of the main reasons why I think this model performed the best is because *SVM* Classifiers are usually insensitive to outliers of data and there are probably many words that, across all 2400 documents, only appear a handful of times. Additionally, *SVM* Classifiers are known to give robust class boundaries to separate two classes which would definitely be helped for the documents that the classifier is less "sure" about. At the same time, though, this could be the reason why the classifier misclassified some of the documents that had a prediction probability close to the threshold/boundaries. Some documents that the classifier may have been less "sure" about might have just happened to be on the wrong side of the robust boundaries, leading the classifier to declare them part of the wrong class.

Problem 6

The best classifier from the previous steps (the final *SVM* Classifier) performed pretty well on the Leaderboard, but fell just short of the *neural network* model. While both models have the same error rate of 0.225, the *neural network* model has a slightly higher accuracy (0.85814) than the *SVM* Classifier (0.85575). This was slightly expected because *neural network* models tend to be very good at generalizing over new data, due to their internal neurons, while the robust *SVM* Classifier boundaries may do more harm than good if samples are very close to them (same reasoning as Part One Problem 6). The logistic regression model was objectively the worst with an error rate of 0.25667.