COMP135: Project 02

Shaikat Islam

November 25, 2019

1 Introduction

This paper explores two datasets: 2,400 training examples derived from `yelp.com` , `amazon.com`, and `imdb.com`[1], as well as pre-trained embedding vectors for 400,000 vocabulary words. On both sets of data, sentiment analysis is performed using three different models. Part 1 follows the use of three different models with hyperparameter tuning, and Part 2 builds on that by using word embeddings to create feature vectors from reviews. Methodologies are explained in both sections, along with respective figures and images.

2 Part 1: Sentiment Analysis Using TF-IDF Representation

2.1 Data Preprocessing and Vectorization

Data was loaded using pandas and then preprocessed. I used the Natural Language ToolKit `(nltk)` to preprocess the data. First, I converted all the sentences to lowercase. After that, I handled contractions (words with apostrophes in them) by converting them into a form without apostrophes, as defined within the dictionary named 'appos' (taken from the comments section in a vidhy blog[2]. After this, I removed all the stop words (commonly occurring words which are not relevant in the context of the data), using nltk's stopwords dictionary. After this, I removed punctuation from the words. Finally, I used lemmatization to convert all the words in a sentence to their base form (lemma), which removes inflection and determines the part of speech.

For the vectorization process, I decided to use `sklearn's` `TfidVectorizer` and `TfidTransformer` classes. Before any preprocessing, the input training data set initially had 4510 unique words, and the test data set had 1921. After preprocessing, which included the removal of stop words, conversion of contractions, and removal of punctuation, the vocabulary size decreased to 3414 words. There is no real order to these words, as there is no link from the words in these reviews in the training and test sets to any parts of speech – further work would be required to determine the part of speech, perhaps using a Conditional Random Fields (CRF) model, which uses probability to determine part of speech. I do not exclude rare words, nor do I exclude common words. I do exclude stop words, which are words that are mostly commonly occurring, but not relevant in the context of the data, as determined by `nltk's` stopwords dictionary. Because I use the `TfidTransformer` and `TfidVectorizer`, common words are penalized, as I use the inverse document frequency (IDF) multiplied by the term frequency (TF) as the "count" for each feature vector. I also only use single features.

Term frequency, or tf, measures how often a term appears in a document, and is normalized by dividing it by the number of terms within the document. Inverse document frequency, or idf, determines how important a term is (neglecting words such as *it*, *as*, or any other articles) by getting the log of the total number of documents divided by the number of documents with the term t within it. The tf-idf is simply the term frequency multiplied by the inverse document frequency.

[1] D. Kotzias, M. Denil, N. De Freitas, and P. Smyth

[2] https://medium.com/@annabiancajones/sentiment-analysis-of-reviews-text-pre-processing-6359343784fb

2.2 Logistic Regression Model for Bag-of-Words

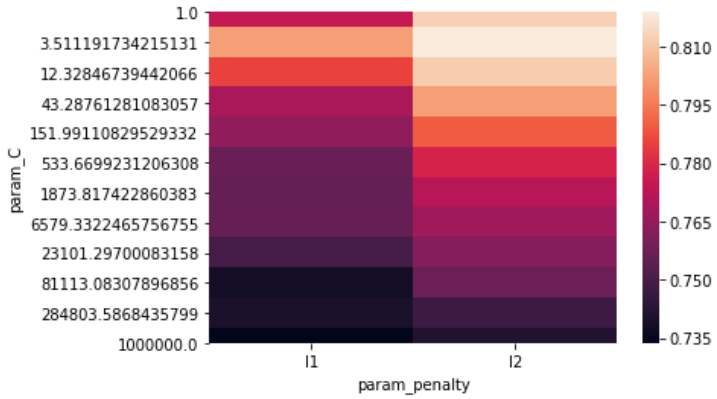

Figure 1: Heatmap for LogisticRegression() Classifier

```
Standard Deviations:  [0.03578 0.02763854 0.03165022 0.02871677 0.02825971
0.02977345 0.01615893 0.02904738 0.01214782 0.02822897 0.00926088 0.02190573
0.00775045 0.02138438 0.01635118 0.01929306 0.02049729 0.02021585 0.01528661
          0.01974666 0.01378153 0.01884697 0.01677051 0.01577841]
```

The first model used for the sentiment analysis was the logistic regression model. I used the LogisticRegression model from sklearn, using its default `'liblinear'` solver, as it seemed the best fit for the size of the data being used in this project, as well as it being relatively computationally inexpensive. During the running of this model, the model did fail to converge at one point, but continued to iterate until it did converge.

The hyperparameters that I used to control the model include the inverse penalty `C` as well as the regularization. I determined the inverse penalty `C` using `np.logspace(0, 6, 12)`, which generated 12 samples of `C` from 1.0 (the default) to 1000000. I used these ranges of `C` after experimenting with different ranges – it turned out that smaller and larger values than those defined by the log space used resulted in greater losses in accuracy. The hyperparameter `C` applies a penalty to the parameter values in order to reduce overfitting using the function $C{=}1/\lambda$. Furthermore, I used different values for regularization (`L1` and `L2`) to see if there was any key differences in terms of overfitting. `L1` regularization uses ridge regularization, or the absolute value of magnitude of the coefficient as the penalty to the loss function, whereas `L2` regularization uses the lasso method, or the squared magnitude of the coefficient as its penalty terms[3] These values for the hyperparameters resulted in 24 different models.

In order to use cross-validation on these values for the hyperparameters, I used `sklearn's GridSearchCV`[4] with 5 folds (in order to reduce computation time), with a verbosity of 0 (`verbose = 0`).

The results are described within the heatmap, in Figure 1. From the results, it seems that the model does best using `L2` regularization with `C` values closest to the default value of 1.0. The model that had the highest accuracy in this case had a `C` value of 3.511 on and an `L2` regularization with an accuracy of 0.81875 and a standard deviation of 0.0287167682335213. The models did the worse with inverse C values greater than 150, on both `L1` and `L2` regularizations. Evidence of over-fitting may exist at `C` values close to the default of 1.0, using `L2` regularization, as these values performed the best with the training set – using the test set on the GradeScope leaderboard, this model gave an error rate of 0.16167 and an AUROC of 0.9061, putting me 10th place on the leaderboard at the time of writing (out of 57).

[3] https://towardsdatascience.com/l1-and-l2-regularization-methods-ce25e7fc831c
[4] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

2.3 Multilayer Perceptron (MLP) Model with Bag-Of-Words

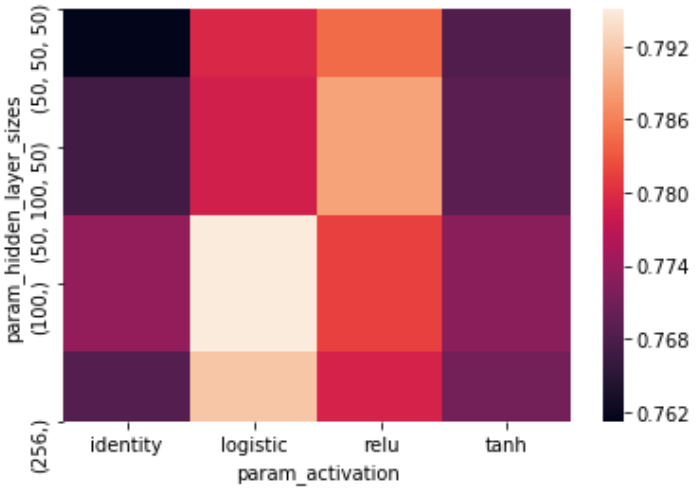

Figure 2: Result of parameter tuning on MLP using Bag-of-Words methodology

```
Standard Deviations:  [0.03578 0.02763854 0.03165022 0.02871677 0.02825971
0.02977345 0.01615893 0.02904738 0.01214782 0.02822897 0.00926088 0.02190573
0.00775045 0.02138438 0.01635118 0.01929306 0.02049729 0.02021585 0.01528661
           0.01974666 0.01378153 0.01884697 0.01677051 0.01577841]
```

This section uses the Multilayer Perceptron (MLP) model from `sklearn`.The hyperparameters used here control for the activation functions, the number of hidden layers, as well as the number of neurons composing the hidden layers. For the number of hidden layers, the size ranges from 1 to 3, as I've learned that for most learning tasks, the number of hidden layers for an MLP model is usually optimized for 1 or 2 hidden layers[5]. As for the number of neurons per layer, I used the default value of shape (100,), as well as shape (256,) and shapes (50,50,50), (50,100,50). The values for the number of neurons were derived by experimenting with different shapes and computation times, as well as the rule of thumb that the number of neurons per hidden layer should be between the range of neurons within the input and output layer (the number of features within the data). I also used four different activation functions, all defined on `sklearn's` documentation[6]: `'identity'`,`'logistic'`, `'relu'`, and `'tanh'`. The 'identity' function uses no-op activation, and it returns f(x) = x. As for the logistic function, this uses the sigmoid, defined by an S-shaped curve, but it does have a few problems, including the vanishing gradient problem, slow convergence, and uncentered gradient updates. This function returns f(x) = 1 / (1 + exp(-x)). The rectified-linear unit (reLU) function converges more often than the `'tanh'` activation function, and it remedies the vanishing gradient problem – its limitation lies in that it can only be used for hidden layers, and it can create weight updates that lead to the non-activation of neurons, leading to neuron death. This function returns f(x)=max(0,x). The `'tanh'` function optimizes faster than the logistic function, but also has the problem of a vanishing gradient. This function returns f(x)=tanh(x). I chose these parameters because these were all possible values for the activation function on `sklearn's` documentation, and I found it interesting to compare different ways of outputting inputs to forward layers. These sets of parameters resulted in 16 different models.

In order to use cross-validation on these values for the hyperparameters, I used `sklearn's GridSearchCV` with 5 folds (in order to reduce computation time), with a verbosity of 0 (`verbose = 0`). This matches what I did with the last `LogisticRegression model`, in order to keep results consistent.

The results are described within Figure 2. From the results, it seems that the model does best using a logistic activation function and a default value of one hidden layer with 100 neurons. This resulted in an accuracy of 0.795 with a standard deviation of 0.019211903138997517. This is

[5] https://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-layers-and-nodes-in-a-feedforward-n
[6] https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

close to the value outputted by the Logistic Regression model, but the Logistic Regression model does better in terms of accuracy. Overfitting may exist in association with the logistic activation function, and data seems to be underfitted when using the simplest identity function. Using the test set on the GradeScope leaderboard, this model gave an error rate of 0.20167 and an AUROC of 0.87983, putting me at 41st place on the leaderboard at the time of writing (out of 61).

2.4 K Nearest Neighbors with Bag-Of-Words

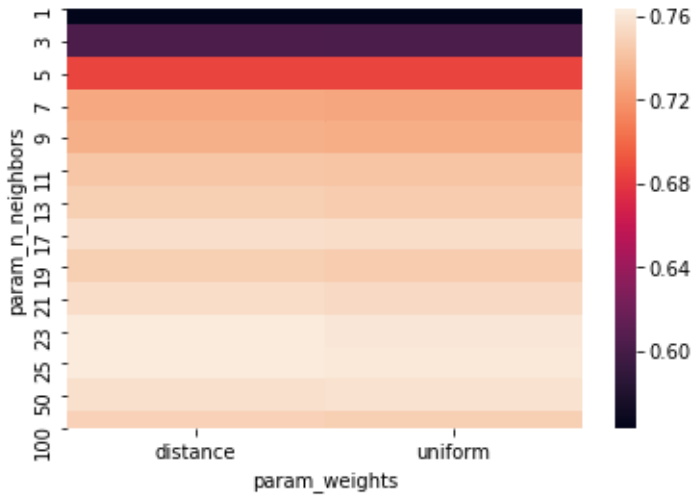

Figure 3: Result of parameter tuning on K-Nearest Neighbor using Bag-of-Words methodology

```
Standard Deviations:  [0.01144128 0.01144128 0.07474958 0.07526112 0.02698379
 0.02757565 0.00805795 0.00870026 0.00618017 0.00552771 0.01333333 0.0124024
 0.01502313 0.01427653 0.01803738 0.01743042 0.0164781 0.0152525 0.02060711
 0.01983438 0.02186607 0.02008662 0.02207154 0.02079162 0.03080517 0.02225952
                        0.0209165 0.01902119]
```

Here, I used a K-Nearest Neighbors clustering model[7] in order to determine sentiment analysis, which gave me the worst performance out of all three models thus far. I optimized on the number of neighbors, as well as the weight function used in training the model. In determining the optimal value for the k number of neighbors, I decided that I should have values lower than the default (5), and a few higher than the default – these higher values (from 7 to 100), were determined practically by running each iteration of the model. I found that there was a decrease in performance once the model reaches n=50, which is where a shift can be seen from n=25 (the increase in gradient, signifying lower performance accuracy). I selected the weight function as I believed it would have a significant effect on how the data points in each neighborhood (cluster) would be calculated, but that cannot be seen in the results, as seen in Figure (insert), as there is a slight difference between the 'distance' and 'uniform' parameter as can be seen in the lack of a clear gradient marker. A `'uniform'` weight function weights all points in each cluster equally, while a `'distance'` weight function weights all points by the inverse of their distance (closer neighbors of a query point have greater weight than neighbors that are further away). This resulted in 28 different models.

As for cross-validation for the hyperparameter tuning, I used 5 folds and a verbosity of 0, as I have with the two previous models.

The results are described in Figure 3. From the results, it seems that the model does best from n=17 to n=25, with the best model having an accuracy of 0.7629166666666667 and a standard deviation of 0.02008661798865659. It is interesting that the weight function did not have as much of an effect on the performance as I thought it would; I believe that the preprocessing of the data, in tandem with the TF-IDF vectorization may have normalized feature vectors a bit too much, leading to a lesser variance in performance based on weight functions. This model ranked 57th

[7] https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

place out of 62, with an error rate of 0.23333 and an AUROC of 0.86578. It is the worst performing model, thus far.

2.5 Summary of Model Results

The best performing classifier has been the logistic regression classifier, to my surprise. I initially thought that it would be very unlikely that the logistic regression model would give me the best results, as it seemed very straightforward – that seemed to work in its favor, however. More complicated models such as the multilayer perceptron classifier performed worse, and the K-Nearest Neighbor model also performed the worst of them all. I believe a large part of this success has to do with the feature tuning performed – by so rigorously trying to normalize the input data into feature vectors using TF-IDF, it seems to me that the Logistic Regression classifier worked the best as it usually performs the best when attributes unrelated to the output variable, as well as closely related attributes are removed from the input set. This was most perceptible within the steps of removing stop words (removing unrelated attributes to the feature set), and the TF-IDF vectorization (penalizing closely related attributes). In combination with the regularization performed on the model, it seems that normalizing the data did make the output classes rather separable.

Using a KNN model may have not been as suited for this task as the number of output classes was limited to 2. Possible reasons for why the KNN model and MLP model may have not worked as best, or in conjunction with my original hypothesis, may also be as a consequence of underfitting, and a lack of greater parameter tuning (which may have resulted in better performance.)

The logistic regression model did best on predicting positive values with data sourced from Amazon, with a true positive rate of .9675, compared to .9475 and .935 for Yelp and IMDb, respectively. The model predicted true negatives especially from Yelp reviews, with a true negative rate of 0.98, compared to .9775 and .9725 for Amazon and IMDb, respectively. The model also had a higher false positive rate for Yelp reviews, at a false positive rate of 0.0525, compared to 0.0325 and 0.065 for Amazon and IMDb reviews. Interestingly, it also had the lowest true negative rates for Yelp reviews, at 0.02, compared to 0.0225 and 0.275 for Amazon and IMDb, respectively. Overall, in terms of total accuracy, the model performed best on Amazon reviews, at an accuracy of 0.9725, compared to 0.96375 and 0.95375 for Yelp and IMDb, respectively. Possible reasons for this may be due to the number of sentimental (positive/negative) words within the feature sets for each type of review – Amazon customers may have better use of language in line with how the model calculates an output class for sentiment, but conversely, my model may also overfit for Amazon reviews.

2.6 Applying Best Classifier to Leaderboard

Using the test set on the GradeScope leaderboard, the logistic regression model gave an error rate of 0.16167 and an AUROC of 0.9061, putting me 11th place on the leaderboard at the time of writing (out of 64). This matches up with what the training set performance eluded to, given that both the training set and the test set had the best performance compared to the other classfiers. This may suggest that the testing data may be similar in terms of TF-IDF values with the training data, and the Logistic Regression model may have overfit on the training data, leading to an increased performance on the testing data (which is similar, in this case). It could also mean that the other models overfit on training data, which in this scenario, is not as similar in comparison to the testing data.

3 Part 2: Word Embeddings

This section explores another technique useful for NLP and sentiment analysis – word embeddings. Here, the GloVe dataset[8] is being used, with 50 dimensions for each feature vector.

3.1 Pipeline for Vectorization using GloVe

First, I preprocessed the data in the same way as described, in detail, in section 1.1.

[8] https://nlp.stanford.edu/projects/glove/1

Since GloVE provides a 50 dimension feature vector for each word, I figured that a good way to vectorize reviews would be to average all the words within the reviews. I thought it best to use the average of all the feature vectors in order to standardize the input data based on varying input feature length. By simply summing and concatenating feature vectors, discrepancies in length are not handled properly. After averaging all the values, I felt that it was relevant to weigh the feature vectors by multiplying the feature vectors by the TF-IDF weight, calculated using `sklearn's TfidfVectorizer()`. This time, each feature vector has 50 dimensions instead of 3414, due to the limit in dimensionality of the word embedding dataset being used.

3.2 Logistic Regression and Word Embeddings

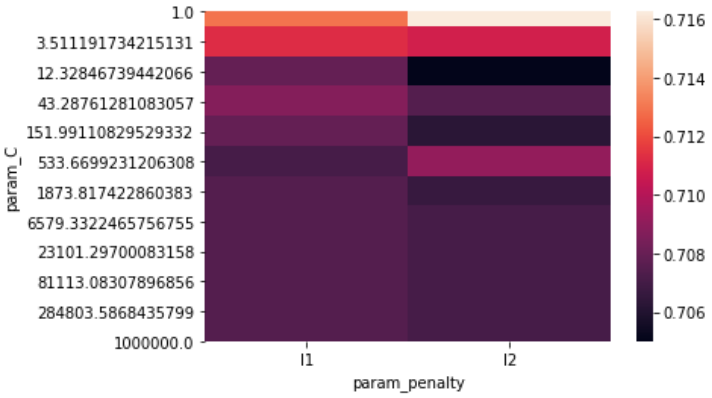

Figure 4: Results of the Logistic Regression Classifier With Hyperparameter Tuning

```
Standard Deviations:  [0.02214222 0.01942757 0.02519369 0.02138438 0.02354812
 0.02624008 0.02206367 0.02322146 0.02409472 0.02169069 0.02348167 0.02402978
 0.02325881 0.02283698 0.02325881 0.02322146 0.02325881 0.02322146 0.02325881
            0.02322146 0.02325881 0.02322146 0.02325881 0.02322146]
```

Here, I used the LogisticRegression model from sklearn, using its default `'liblinear'` solver. The hyperparameters that I used to control the model include the inverse penalty `C` as well as the regularization. These values are the same as described in section 2.2. I determined the inverse penalty `C` using `np.logspace(0, 6, 12)`, which generated 12 samples of `C` from 1.0 (the default) to 1000000. Comparing these results with those evaluated in section 2.2 using the Bag-of-Words model, it seems that there is a sharper divide in model accuracies based on parameter penalties and inverse penalties. For the previous model, using the Bag-of-Words approach, there was a gradual range of accuracies, visible in the gradient observed on Figure 1's `l2` column. These values for accuracy are also far narrower than those observed in section 2.2; the range of these values are .10 – those in section 2.2 have a range of .8. These values for the hyperparameters resulted in 24 different models.

In order to use cross-validation on these values for the hyperparameters, I used `sklearn's GridSearchCV` with 5 folds with a verbosity of 0 (`verbose = 0`).

The results are described within the heatmap in Figure 4. From the results, it seems that the model does best using `L2` regularization with `C` values closest to the default value of 1.0 with a regularization of `l2`. This is in accordance with the results from section 2.2, but are surprising nonetheless. The model that had the highest accuracy in this case had a `C` value of 1.0 on and an `L2` regularization with an accuracy of 0.71683 and a standard deviation of 0.0227684. Evidence of over-fitting does seem to exist: the narrow range of accuracy for this model is a great signifier that there has been overfitting on this dataset, perhaps due to the complex vectorization of the feature inputs. Compared to the models described in part 2.2, however, this model underfits the training data greatly in comparison, perhaps due to the lack of proper input normalization. Using the test set on the GradeScope leaderboard, this model gave an error rate of 0.295 and an AUROC of 0.77871, putting me 55th place on the leaderboard at the time of writing (out of 62).

3.3 Multilayer Perceptrons (MLP) and Word Embeddings

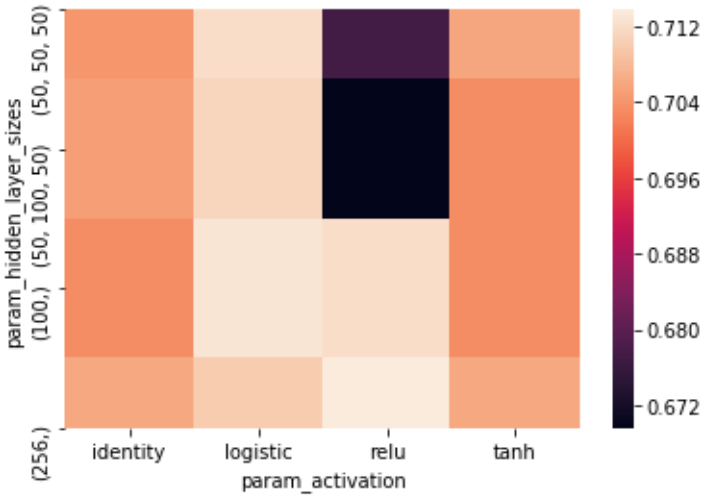

Figure 5: Results of the MLP Classifier With Hyperparameter Tuning

```
Standard Deviations:  [0.0295804 0.02498611 0.02344467 0.02729087 0.02166667
0.02417385 0.02391391 0.02 0.03105215 0.02445943 0.0353406 0.03721186 0.02332589
0.02402978 0.02318405 0.02422407]
```

This section uses the Multilayer Perceptron (MLP) model from `sklearn`. The hyperparameters used here control for the activation functions, the number of hidden layers, as well as the number of neurons composing the hidden layers. I used the default value of shape (100,), as well as shape (256,) and shapes (50,50,50), (50,100,50) for the number of neurons. I also used four different activation functions, (described in section 2.3). I used the same set of hyperparameters in order to compare the Bag-of-Words approach with the Word Embedding approach, as I have done with the previous section. These sets of parameters resulted in 16 different models.

In order to use cross-validation on these values for the hyperparameters, I used `sklearn's GridSearchCV` with 5 folds, with a verbosity of 0 (`verbose = 0`).

The results are described within Figure 5. These results differ greatly from those described in Figure 4, but the range of values for the accuracy are within a reasonable range (.03 - .04 for both models). What stands out within these results is the accuracy of the models using a rectified linear unit activation function with the (50,100,50) and (50,50,50) neuron/hidden layer structure. These two models performed the worst out of the 16, with an accuracy within the range of .672 to .688. The best MLP model had an accuracy of 0.71375 with a standard deviation of 0.03721185593627686. This is surprisingly similar to the accuracy reported by the logistic regression model described previously. I believe that this is a sign of overfitting – combined with the two outliers with accuracies within the range of .672 to .688, as well as the fact that most of the models described in Figure 5 have extremely similar accuracies, there must be overfitting going on, which is confirmed by my results on the leaderboard: I had an error rate of 0.27667 and an AUROC of 0.79791, placing me at 55th place out of 64 (at the time of writing). Again, in comparison to the models described in section 2, the models in section 3 have thus far underfitted more in comparison (performing worse on the training data and testing data).

3.4 Support Vector Machines and Word Embedding

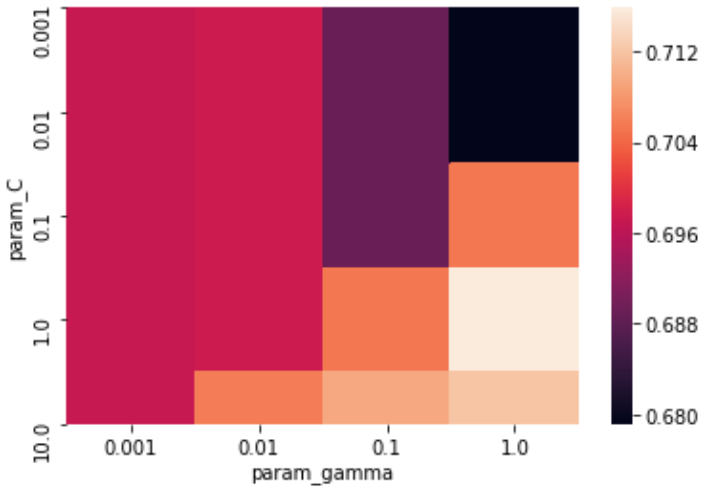

Figure 6: Results of the SVC Classifier With Hyperparameter Tuning

```
Standard Deviations:  [0.02666667 0.02811805 0.02968001 0.03013281 0.02666667
 0.02811805 0.02968001 0.03013281 0.02666667 0.02811805 0.02968001 0.02827813
 0.02666667 0.02811805 0.02428134 0.02276846 0.02666667 0.02365845 0.0225
 0.02462919]
```

This section uses the C-Support Vector Classification model from `sklearn`. The hyperparameters used here control for the inverse penalty `C` as well as the `gamma`. For `C`, I used values with 5 different orders of magnitude, ranging from 0.0001 to 10 in order to study the effect of lower and higher inverse penalties on the regularization of the model. This model uses a non-linear hyperplane by default (I believed that the success of the non-linear logistic regression function used in section 2.1 would carry on to another classifier, which is why I didn't change it). Because the model uses a nonlinear `'rbf'` kernel, it also uses the `gamma` parameter to regularize – the higher the `gamma`, the more the model fits to the training set.

In order to use cross-validation on these values for the hyperparameters, I used `sklearn's GridSearchCV` with 5 folds, with a verbosity of 0 (`verbose = 0`) (as I've done throughout the entire project).

The results are described within Figure 6. The left half of the figure, describing `gamma` parameters from 0.001 to 0.01 are on the lower end of the spectrum, and have similar accuracy values for all parameters of `C`. I believe this a result of underfitting to the training data – at higher values of `gamma`, as can be seen where `gammma`=1.0, there does seem to be increase in accuracy, suggesting that `gamma` regularization may have had a tightening effect on the model. Furthermore, for lower inverse penalty values of `C`, from 0.001 to 0.01, there seems to be lower accuracy values, suggesting that increasing `C` also pushes the model to further fit the training set. The range of these values is within .3, similar to the MLP models, but for word embedding, it seems that SVMs show the greatest variety of results for performance. Compared to all the other models, however, it does rather poorly: the best model scored 0.71583 for accuracy with a standard deviation of 0.0227684. These results are also strikingly similar to the values reported previously for this part of the project, further suggesting an overfitting of data.

These results differ greatly from those described in Figure 4, but the range of values for the accuracy are within a reasonable range (.03 - .04 for both models). What stands out within these results is the accuracy of the models using a rectified linear unit activation function with the (50,100,50) and (50,50,50) neuron/hidden layer structure. These two models performed the worst out of the 16, with an accuracy within the range of .672 to .688. The best MLP model had an accuracy of 0.71375 with a standard deviation of 0.03721185593627686. This is surprisingly similar to the accuracy reported by the logistic regression model described previously. I believe that this is a sign of overfitting – combined with the two outliers with accuracies within the range of .672

to .688, as well as the fact that most of the models described in Figure 5 have extremely similar accuracies, there must be overfitting going on, which is confirmed by my results on the leaderboard: I had an error rate of 0.27667 and an AUROC of 0.79791, placing me 55th out of 64 (at the time of writing). Again, in comparison to the models described in section 2, the models in section 3 have thus far underfitted more in comparison (performing worse on the training data and testing data).

3.5 Summary of Model Results

The best performing classifier in this section has again been the Logistic Regression model, with an accuracy of 0.71683 and a standard deviation of 0.0227684. This confirms the data reported in section 2 of this report, where the logistic regression model also performed the best, but in this case, the margins of accuracy are extremely low, as all three models in this section scored within the range of .71375-.71683, which heavily suggests that there has been some overfitting, with respect to the testing data, and underfitting, with respect to the models described in section 2. It seems that the TF-IDF weighting in conjunction with the averaging of the feature vectors may have not been the best way to normalize the input features. One reason for this may be due to the equality of words within word-embedding algorithms: words already have context attached to them with respect to the corpus of the entire text – further adjustment by weighting them with the TD-IDF weights may have severely overfitted the data with respect to the testing set, which explains the very poor performance on the leaderboard. Using a simpler method for vectorization may have resulted in better fitted data – using averages and averages only would still provide a good source of normalization as it adjusts for the size of the feature vector – using the TD-IDF may definitely have been overkill.

The logistic regression model did best on predicting positive values with data sourced from Yelp, with a true positive rate of .7275, compared to .6875 and .695 for Amazon and IMDb, respectively. The model predicted the best true negatives rates for Amazon reviews, with a true negative rate of 0.7975, compared to .7025 and .735 for Amazon and IMDb, respectively. The model also had a higher false positive rate for Amazon reviews, at a false positive rate of 0.3125, compared to 0.2725 and 0.305 for Amazon and IMDb reviews. Interestingly, Amazon also had the lowest true negative rates for at 0.2025, compared to 0.2975 and 0.265 for Yelp and IMDb, respectively. Overall, in terms of total accuracy, the model performed best on Amazon reviews, at an accuracy of 0.7425, compared to 0.715 and 0.715 for Yelp and IMDb, respectively. This further suggests that Amazon reviews have the best sentiment indicators, perhaps due to the breadth and scope of the data that is available on their website, as their website is definitely more popular, and thus subject to greater values of variance, which can result in a healthier input set.

3.6 Applying Best Model to Leaderboard

Using the test set on the GradeScope leaderboard, the logistic regression model gave an error rate of 0.295 and an AUROC of 0.77871, putting me at 55th place on the leaderboard at the time of writing (out of 62). This matches up with what the training set performance eluded to, given that both the training set and the test set had the best performance compared to the other classfiers.

However, in this part of the project, the ranges for the accuracy were extraordinarily lower, and the accuracies were also poorer, which is definitely not what I expected. I believed that using word-embeddings would have had a net positive increase in accuracy due to the decrease in sparsity provided by the word embeddings – instead 3414 dimensioned feature vectors, there were now 50, which I believe would play well with both the MLP and SVM model, as they were simpler.

Now, reflecting back on the results, they do make sense. The vectorization of the features did result in a net loss of information (300 dimensioned feature vectors are recommended usually). What the word-embeddings lack in sentiment semantics, they make up for in more compact input formats and easy computation.

The results of the leaderboard suggest that my preprocessing and pipeline definitely overfitted the data – just comparing my results in part 1 with these results is an indicator of that. Perhaps using something as simple as the average of the word-embedding vectors as the vectorization process would reduce overfitting greatly.

References

1 D.Kotzias,M.Denil,N.DeFreitas,andP.Smyth

2 https://medium.com/@annabiancajones/sentiment-analysis-of-reviews-text-pre-processing-6359343

3 https://towardsdatascience.com/l1-and-l2-regularization-methods-ce25e7fc831c

4 https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.
html

5 https://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-layers-and-r

6 https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.
html

7 https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.
html

8 https://nlp.stanford.edu/projects/glove/l