Classifying Reviews with NLP

CS135 Project 2

Joseph Egan

December 2022

Introduction

The dataset for this project contains 3000 reviews evenly distributed between Amazon, Yelp, and IMDB. Labels of 0 and 1 for negative and positive sentiment were given for 2400 samples, while 600 were left for a final testing set. In this project, I evaluated four models with varying hyperparameters, along with several options for pre-processing the data before modeling. Ultimately, the final model selected was able to achieve 91.5% accuracy on the testing set with almost 97% AUROC.

Data Exploration

To begin, I examined both training and test data frame to see what the distributions of reviews were:

website_name	is_positive_sentiment	
amazon	0	400
	1	400
imdb	0	400
	1	400
yelp	0	400
	1	400

Number of reviews for each website and sentiment value in the training set

Both the training and testing set had equal amounts of Amazon, Yelp, and IMDB reviews, and the training set had equal amounts of positive and negative reviews.

Feature Engineering & Text Processing

I attempted many modifications/adjustments in this section, some of which did not have significant or positive impact in the results and thus were not used in creating the final models. In fact, for the best model, I made just about no modifications to the text, and that is described further in later sections. I have included a chart at the end of this section showing what was and what wasn't included as a helpful reference.

Adding Features

Before preparing the text data for modeling, I tried to harvest as much information from the other parts of the training data as I could. To keep things simple, I one-hot encoded the website labels (Amazon, Yelp, IMDB) and created a variable that was the length of each review. This added four features in total.

Text Processing

Character and Word Removal

From each review, I removed special characters, numbers, and punctuation using the 'string' library. Additionally, I used a modified version of the nltk 'english' stop-words dictionary to remove common words from each string that likely would not provide classification benefit. I created the modified dictionary by removing certain stop-words from the nltk version that I thought would be useful (and therefore I wanted to retain them). These mostly included negative modifiers such as "not" and "doesn't". The full list is available in the appendix.

Word Modification

Next, I used the textblob library to spell check and replace each remaining word in the text. I also used nltk's WordNetLemmatizer to modify words to their 'base' version. Some examples of this would be modifying 'rocks' to 'rock or 'corpora' to 'corpus'.

TF_IDF Transformation

Finally, I converted each sample in the dataset into a vector using sklearn's TfidVectorizer. This method creates a vocabulary of every word used in the body of texts. Each review is converted into a vector with

dimensionality equal to the size of the vocabulary. The value of element 'j' in a review's vector is equal to the tf_idf value for the word that element 'j' represents. The tf_idf value is calculated in such a way that more frequently used words in the entire body of texts are weighted less than less frequently used words. Removing stop-words and using tf_idf instead of regular bag-of-words both have the effect of assigning weight to less common words.

I then added in the created features (one-hot encoded websites, text lengths). While performing these modifications, I examined samples of the data to see what the modified text looked like. Some of the modifications led to new texts or vectors that I was unsure about, and so I evaluated some of the modifications using a simple logistic regression model with 5-fold cross validation. After comparing the validation accuracy of each modification, I generated the final tf_idf feature matrix using these modifications:

Modification/Feature	Level of Inclusion	Reasoning	Other
Meta features	All removed	Led to extremely high overfitting on training data, negligible impact on validation accuracy, and horrible testing accuracy	I'm not sure why these features caused such epic devastation to the testing accuracy
Character/Word Removal	All included	My modified stop-words list gave a slight improvement to accuracy	I always included removal of non-letter characters.
Word Modification	Textblob (spell check) kept, Lemma removed	Lemma seemed to modify words incorrectly just as often as it did correctly, and using it led to no noticeable improvement in validation accuracy.	Textblob mostly made correct changes, with incorrect changes mostly occurring on stop-words which I removed anyways.
Vocabulary Size	All words remaining after the above modifications were retained. >4000 words	Best testing accuracy*	Tested 500, 1200, and all words (>4000), along with .9 cutoff for term frequency

*My biggest fear was overfitting since I had more features than I did samples. However, validation and testing accuracy were generally higher for the models that used the full vocabulary size. As expected, training accuracy was also extremely high (~99%).

Model Selection and Results

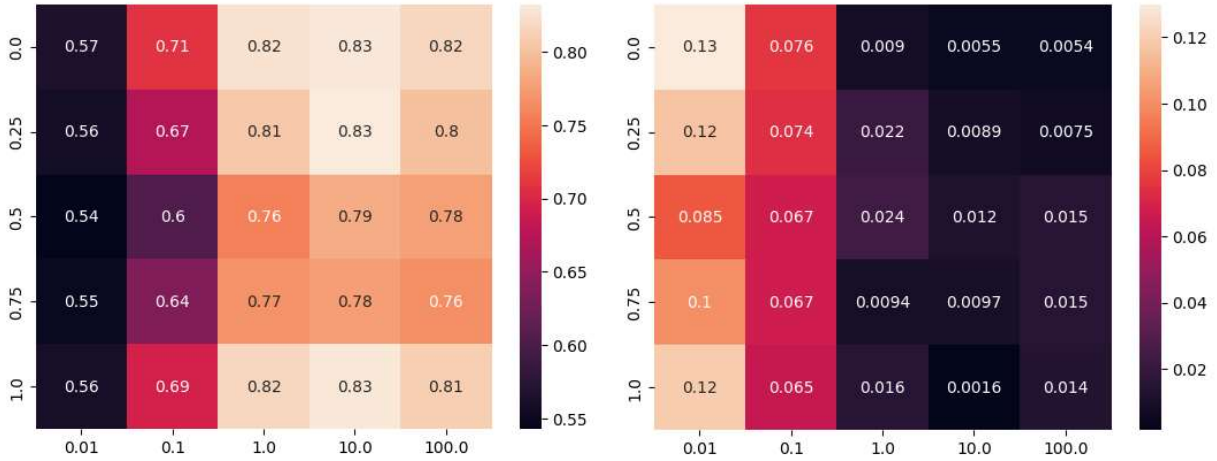
Logistic Regression

After creating the tf_idf vector, I used sklearn's LogisticRegressionCV class to create several logistic regression models with varying hyperparameters. I used the 'saga' solver so that I could use 'elasticnet'. This allowed me to use both l1 and l2 regularization, and also tune the weighting of both. I used 5-fold cross validation across these hyperparameters to generate a total of 125 models.

- L1_ratio had values of [0, 0.25, 0.50, 0.75, 1.00]
    - L1_ratio allows me to set what the weight is on the L1 penalty vs the L2 penalty where the sum of the two equals 1.

    - Both penalties have the effect of shrinking learned model weights. I was expecting L1 to be more useful since it is able to set model weights to 0.
    - In this way, I thought it would 'turn off' words that were not important.
- C (inverse regularization strength) had values [0.01, 0.1, 1.0, 10.0, 100.0]
    - This controls the level of impact that the regularization penalties are allowed to have in the loss function compared to the error term.
    - I chose these values to get a wide selection of levels.
    - After finding the optimal combination of these two, I explored even stronger regularization values (lower Cs), but found validation and testing accuracy did not improve.

Below are the mean and standard deviations of validation accuracy for each combination of hyperparameters. The columns are 'C' and the rows are 'l1_ratio':



Heatmap of average (left) and standard deviation (right) of validation accuracy for each hyperparameter value

Given the high number of features, there was likely overfitting across all hyperparameter values. The LogisitcRegressionCV class from sklearn does not provide you with a record of training accuracy, so I was unable to display it here. However, I was able to spot check several training data samples with some of the hyperparameter values, and training accuracy was >95%, clearly overfit. As stated earlier, reducing the number of features to mitigate this overfitting had a negative impact on validation and testing accuracy, so I chose to retain all features despite the disparity in training and validation accuracy.

It was clear from these results that some regularization helped, while too much was harmful. Interestingly, higher accuracy was concentrated at the extremes of the l1_ratio, meaning that either one or the other was preferred, but not a mixture of the two. The nice thing about the above results is that the hyperparameter combinations with the highest average validation accuracy also have the lowest standard deviation.

Kernel Support Vector Machine

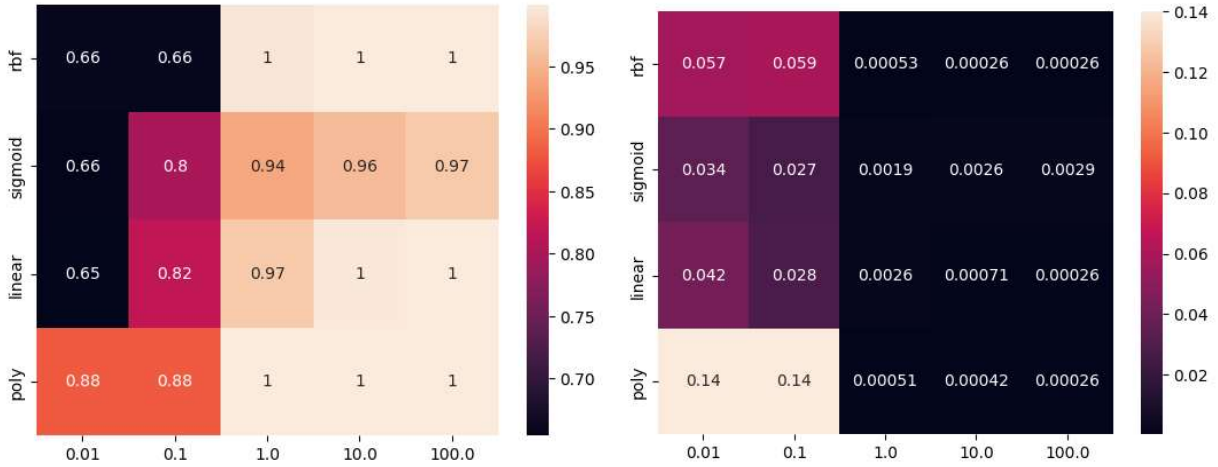
The next model type that I trained was the SVM. Since SVM does not come with a cross validation method built in like LogisticRegressionCV, I manually did cross validation using the StratifiedKFold class from sklearn with 5 folds. This allowed me to split the data into 80% training 20% validation 5 times, with each sample being used in the validation set once, and with both sets balanced evenly between

negative and positive sentiments. I then looped over four different kernels and five different regularization strengths:

- I tried the RBF, Sigmoid, Polynomial, and Linear kernels
    - These are all the available kernels from sklearn
- C (inverse regularization strength) had values [0.01, 0.1, 1.0, 10.0, 100.0]
    - I chose these values to get a wide selection of regularization strengths

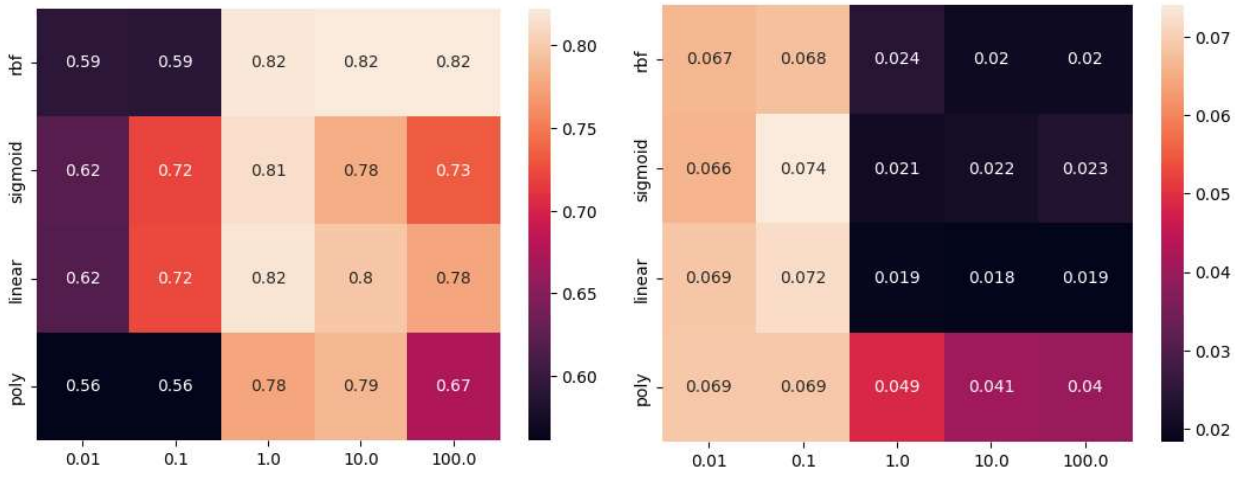
After training 100 models across each fold and hyperparameter value, I averaged the accuracies across each fold and obtained these results. Rows are the kernel functions and columns are squared-l2 inverse regularization strengths:

Training Accuracy and Standard Deviation:



Average accuracy (left) and standard deviation (right) across five folds for training data

Validation Accuracy and Standard Deviation:



Average accuracy (left) and standard deviation (right) across five folds for validation data

There are several similarities between the results here and the results from the logistic regression models. Both model types generated significantly higher training accuracy than validation accuracy. Again, this is likely due to the large number of features used. Another similarity is that the model configurations for the SVM models that led to the highest accuracies also led to the smallest standard deviations across the five folds. On the training data, all the kernels were able to obtain perfect or near perfect accuracy with very tight standard deviations when the regularization was weak. However, performance across kernels diverged when they were applied to the validation data. The RBF kernel was able to retain performance the best, while the polynomial kernel retained the least amount of accuracy, likely due to overfitting.

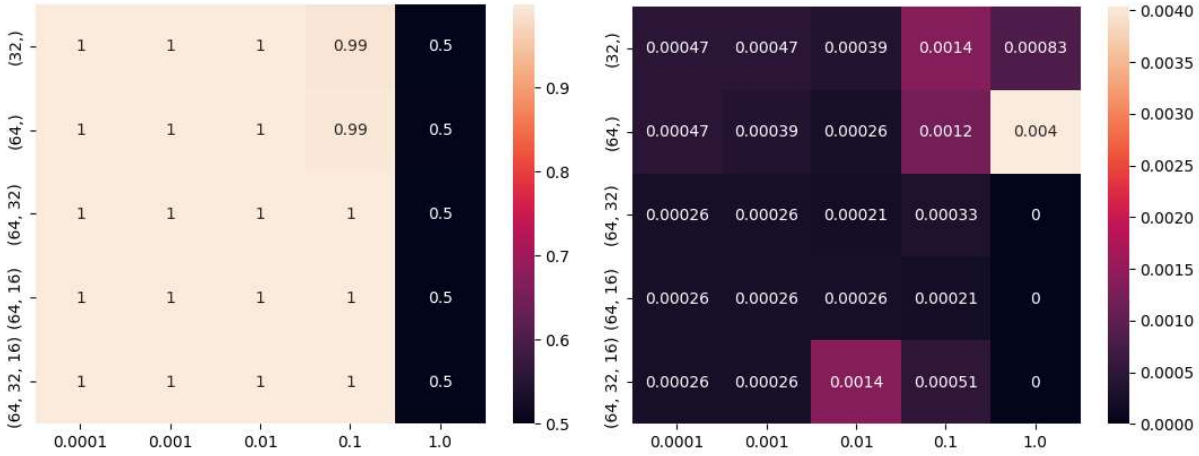
Multi-Layered Perceptron

The third model type that I tested was the sklearn multi-layered perceptron neural network. I used the GridSearchCV class to check hyperparameters across five-fold cross validation. The hyperparameters I varied were:

- Hidden layer sizes – this hyperparameter allows for the control of the network size
    - The parameter is structured as a tuple where the i-th element corresponds to the number of neurons in the i-th hidden layer
    - I tested 5 network sizes: two 1 hidden layer, two 2 hidden layer, and one three hidden layer
    - [(32,), (64,), (64,32), (64, 16), (64, 32, 16)]
- Alpha – similar to C in previous models, it controls the strength of regularization
    - These models took a lot longer to train and I was initially unsure of what values to use
    - I ran a search initially using higher values (100, 10, 1), since they were used in the code demo on GridSearchCV
    - However, the higher alpha values led to very bad results, so I dropped to a lower selection [0.0001, 0.001, 0.01, 0.1, 1.0]

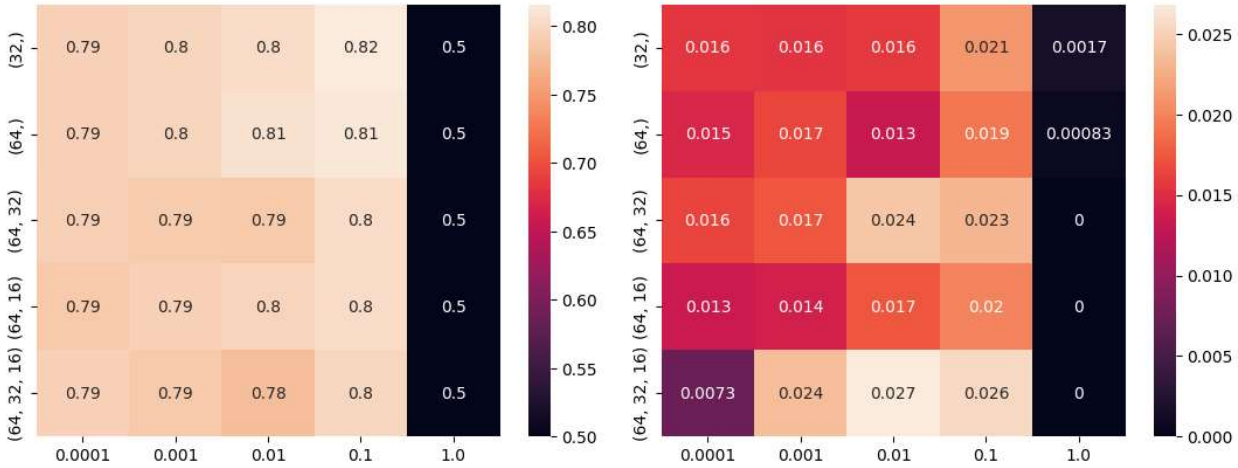
I trained 125 models and obtained these results. Rows are the network sizes while columns are the inverse regularization strengths:

Training Accuracy and Standard Deviation:



Average accuracy (left) and standard deviation (right) across five folds for training data

Validation Accuracy and Standard Deviation:



Average accuracy (left) and standard deviation (right) across five folds for validation data

The first observation from reviewing the results was again there was clear evidence of overfitting, likely due to the large number of features relative to the size of the training set. Regularization was helpful for both validation and training accuracy, and it was clear that too much regularization led to worse results with slightly tighter standard deviations. This was a departure from the logistic and SVM models, where the higher validation accuracies had tighter standard deviations. However, the differences between the standard deviations at the strongest regularization (0.0001) and second to weakest (0.1) were much smaller here than the differences across the regularization parameters for the logistic and SVM models.

Comparison of Original Three

After evaluating the three model types using cross validation, I took the best hyperparameters from each and re-trained one model of each type using those hyperparameters. I trained the models on a 90/10 split of the labelled training data using the StratifiedKFold class from sklearn. I observed these accuracies and confusion matrices:

Model	**Accuracy on Labelled Test Data**
Logistic Regression	0.8625
RBF SVM	0.8625
MLP	0.8500

```
Final Logistic Reg Model
Predicted    0    1
True
0          104   16
1           17  103
```

```
Final SVM Model
Predicted    0    1
True
0          106   14
1           19  101
```

```
Final MLP Model
Predicted    0    1
True
0          103   17
1           19  101
```

The SVM and logistic model had the same accuracy, which was higher than that of the MLP model. It's difficult to say why the MLP model was slightly worse, the difference could be insignificant and a result of the nature of the data in the training vs testing splits. When I ran the models again on a different

split, they had similar overall performance, but the SVM was now slightly lower than the MLP. If the training data contains better (that is, leading to better generalization) support vectors for the SVM, it's possible that it could then outperform the other models, since the SVM only relies on its support vector samples.

I then looked at common false positives and false negatives across the three:

False Positives:

['oh i forgot also mention weird color effect phone'

'none three sizes sent heads would stay ears'

'give bluetoooth heads time still not comfortable way fits ear']

These are results that were classified positive but were actually negative. To a human, these all seem intuitively negative, but it's possible that the classifiers were not able to associate 'weird', 'none', and 'not' as negatives in the contexts of the sentences.

False Negatives:

['their research development division obviously knows there doing'

'plan ordering again'

'little weeks think that sex toast rocks oozes sex right battery embedded sleek stylish leather case']

These are results classified as negative but that were actually positive. Again, to a human it is obvious that these are positive. It's possible that 'obvious' was weighted more negatively since it is stronger language that people might use in a negative review. I have the same hypothesis for the word 'again', as in people would say 'I'm never coming here again" or "I will never watch this again".

I also then looked at the false negative and false positive counts for each classifier across website types:

	Logistic		**SVM**		**MLP**	
	False Pos	**False Neg**	**False Pos**	**False Neg**	**False Pos**	**False Neg**
Amazon	5	2	5	2	7	2
Yelp	5	9	4	9	5	9
IMDB	6	6	5	8	5	8
Total	**16**	**17**	**14**	**19**	**17**	**19**

Interestingly, Yelp and IMDB had higher rates of error than Amazon, especially when it came to false negatives. My suspicion is that people are willing to use more extreme language that is easier for the classifiers to identify when reviewing objects and purchases, whereas they might holdback a bit more when reviewing something more personal like someone's business or movie.

Other Neural Networks

The more I read into NLP techniques, the more interested I became in learning about more complicated neural networks. This led me to building RNN's, GRU's, and LSTM's, using tokenization and the tensorflow and keras packages. All of them had decent but not amazing performances. I then checked out the transformer neural network called BERT, and used huggingface's transformer package to load in two versions:

- BERT Base which had >100 million parameters
- BERT Large which had > 300 million parameters

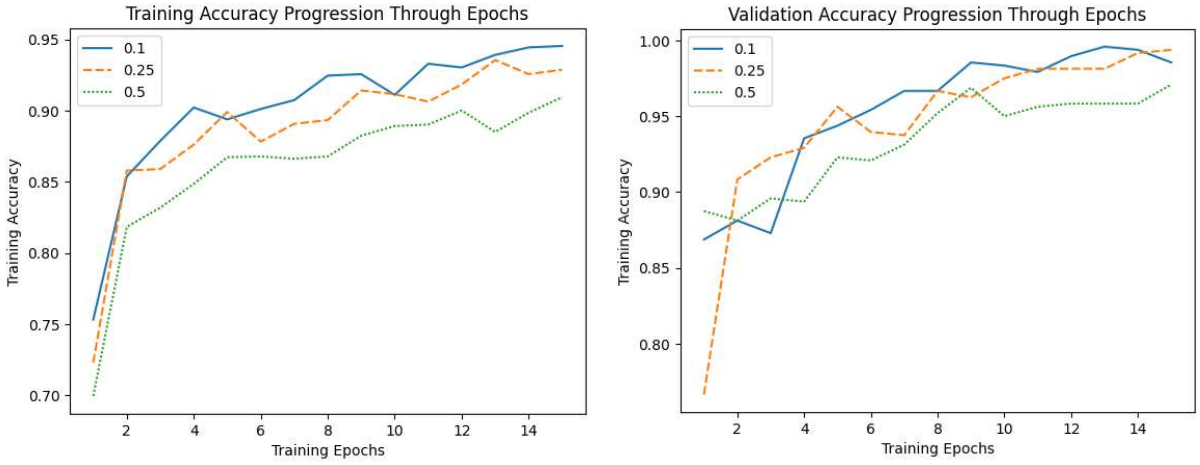
There were two sets of inputs to the model for each sample:

1. The tokens
    a. Each word in the corpus was assigned a number, and so each review became an array of numbers, padded with 0s at the end so each sample array was the same length
2. Attention mask
    a. Same length as the token arrays, but the elements were 1 when there was a word in that spot, and 0 otherwise

I used the BERT models to generate the word embeddings from the tokens, and then stacked several layers on top of those embeddings. So, while there were millions of parameters, I ended up only training around 100,000 so that BERT would learn to model this dataset.

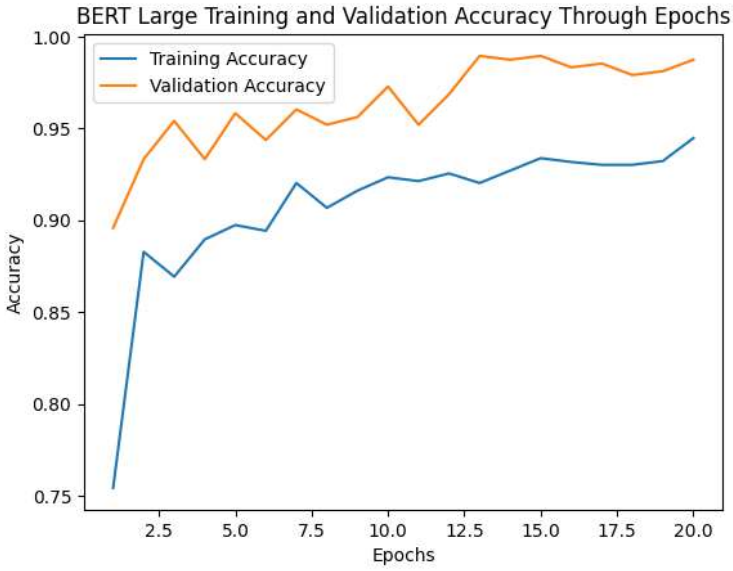
Transformer models are a vast improvement over GRU's/LSTM's. While the latter are able to develop some context in sentences sequentially (forwards and backwards too), they cannot correlate words across sentences/text. The transformer model uses the concept of 'attention' to develop context between all the words in a sentence or text. BERT Base I ran with three different dropout rates ([0.1, 0.25, 0.5]) after the first dense layer, and for 15 epochs. Models trained to 20 and 25 epochs were able to achieve 100% validation accuracy, but performance on the test set did not improve commensurately.

Here are the results for BERT Base:



Training (left) and validation (right) accuracy for BERT Base models over 15 epochs with three different dropout rates

BERT Large I ran with only 0.5 dropout rate and for 20 epochs given time constraints:



These models were able to obtain 100% validation accuracy given enough training, and were trending towards 100% training accuracy. It is interesting that validation accuracy is higher than training. My guess is that even though I shuffled the data before sending it to the network, the validation set contained samples that were 'easier' to classify. If I had more time and GPUs, I could try running on several different splits/folds.

Final Comparison

Here are the results from submitting the test predictions from the best model as determined by validation accuracy in each section:

Model	**Hyperparameters**	**Test Accuracy**	**Test AUROC**
Logistic Regression	C = 10 L1_ratio = 0	0.8100	0.8100
Kernel SVM	Kernel = RBF C = 10	**0.8233**	**0.8233**
Multi-Layer Perceptron	Hidden Layer = (32,) Alpha = 0.1	0.8183	0.8183
BERT Base	Dropout = 0.1 Epochs = 15	0.8850	0.9498
BERT Large	Dropout = 0.5 Epochs = 15	**0.9150**	**0.9679**

These results are what I expected – a few points lower than the validation results. For the three original classifiers, this was encouraging to me that even though I was overfitting, the results were still good on data that even the hyperparameters were not trained on.

Conclusions

When using the older classification models, it's clear that the data pre-processing and feature engineering are the keys to good results. I spent a ton of time tweaking the modifications that I made to the texts, and I still don't think that I found the right mixture/methods. Looking at the leaderboard for the testing predictions, unless my classmates also used more advanced neural networks, it's clear that some of them found a better way to process the texts before converting them to td_idf vectors. I think this is where experience is helpful. This was my first time doing an NLP project like this, and while it took me a while to work through everything, the second time will definitely be much quicker and probably better.

That being said, the BERT models required no modifications prior to the embedding and, once they were setup, required about the same amount of time to train and predict as the classical models. Combined with the fact that their results were by far best-in-class, it makes more sense to just spend time on improving the architecture and tweaking the hyperparameters on those.

Appendix

Words Removed from Stop-Word List

retain_words = ['not', 'nor','no','don',"don't",'very','aren', "aren't", 'couldn', "couldn't", 'didn', "didn't", 'doesn', "doesn't", 'hadn', "hadn't", 'hasn', "hasn't", 'haven', "haven't", 'isn', "isn't", 'mightn', "mightn't", 'mustn', "mustn't", 'needn', "needn't", 'shan', "shan't", 'shouldn', "shouldn't", 'wasn', wasn't", 'weren', "weren't", 'won', "won't", 'wouldn', "wouldn't"]

BERT Base Network

```
Model: "model"
_________________________________________________________________________________________________
 Layer (type)                   Output Shape              Param #     Connected to
=================================================================================================
 inputs_ids (InputLayer)        [(None, 61)]              0           []

 attention_mask (InputLayer)    [(None, 61)]              0           []

 tf_bert_model (TFBertModel)    TFBaseModelOutputWi       108310272   ['inputs_ids[0][0]',
                                thPoolingAndCrossAt                    'attention_mask[0][0]']
                                tentions(last_hidde
                                n_state=(None, 61,
                                768),
                                 pooler_output=(Non
                                e, 768),
                                 past_key_values=No
                                ne, hidden_states=N
                                one, attentions=Non
                                e, cross_attentions
                                =None)

 global_max_pooling1d (GlobalMa  (None, 768)              0           ['tf_bert_model[0][0]']
 xPooling1D)

 batch_normalization (BatchNorm  (None, 768)              3072        ['global_max_pooling1d[0][0]']
 alization)

 dense (Dense)                  (None, 128)               98432       ['batch_normalization[0][0]']

 dropout_37 (Dropout)           (None, 128)               0           ['dense[0][0]']

 dense_1 (Dense)                (None, 32)                4128        ['dropout_37[0][0]']

 outputs (Dense)                (None, 1)                 33          ['dense_1[0][0]']

=================================================================================================
Total params: 108,415,937
Trainable params: 104,129
Non-trainable params: 108,311,808
_________________________________________________________________________________________________
```

BERT Large Network

```
Model: "model"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 inputs_ids (InputLayer)        [(None, 100)]        0           []

 attention_mask (InputLayer)    [(None, 100)]        0           []

 tf_bert_model (TFBertModel)    TFBaseModelOutputWi  335141888   ['inputs_ids[0][0]',
                                thPoolingAndCrossAt               'attention_mask[0][0]']
                                tentions(last_hidde
                                n_state=(None, 100,
                                 1024),
                                 pooler_output=(Non
                                e, 1024),
                                 past_key_values=No
                                ne, hidden_states=N
                                one, attentions=Non
                                e, cross_attentions
                                =None)

 global_max_pooling1d (GlobalMa  (None, 1024)        0           ['tf_bert_model[0][0]']
 xPooling1D)

 batch_normalization (BatchNorm  (None, 1024)        4096        ['global_max_pooling1d[0][0]']
 alization)

 dense (Dense)                  (None, 128)          131200      ['batch_normalization[0][0]']

 dropout_73 (Dropout)           (None, 128)          0           ['dense[0][0]']

 dense_1 (Dense)                (None, 32)           4128        ['dropout_73[0][0]']

 outputs (Dense)                (None, 1)            33          ['dense_1[0][0]']

==================================================================================================
Total params: 335,281,345
Trainable params: 137,409
Non-trainable params: 335,143,936
__________________________________________________________________________________________________
```