Ronen Reouveni - NLP Final Project

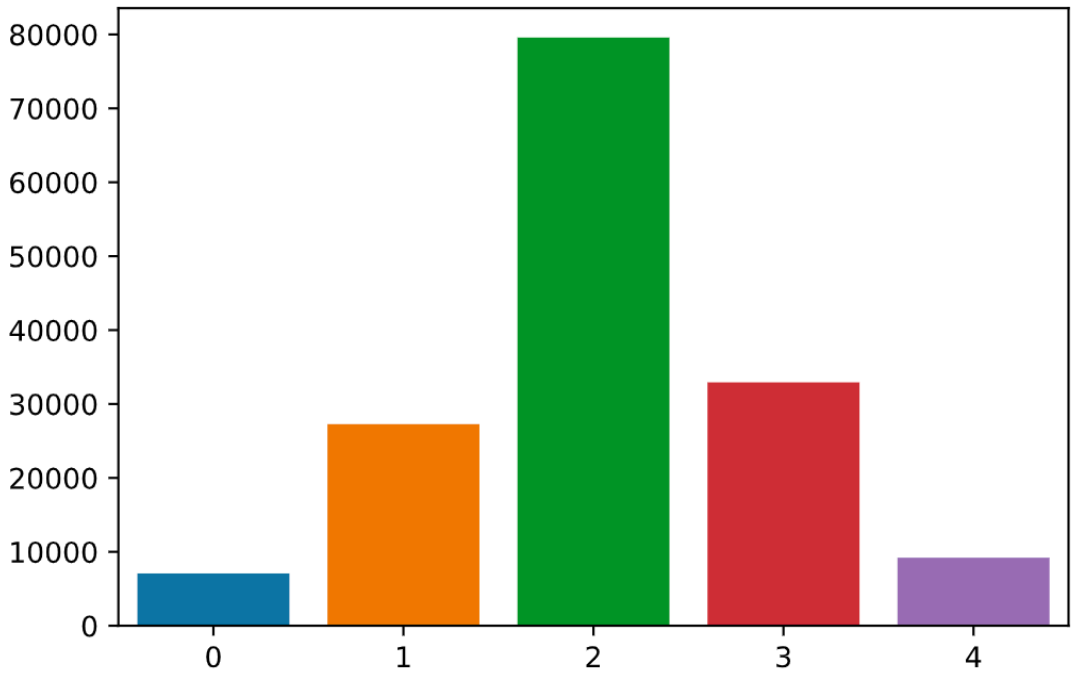
Introduction

An amazing and interesting part of the internet is the world of reviews. Almost everything we buy or experience for entertainment can be reviewed by anyone. There is a tremendous amount of business value in being able to label any text with the correct sentiment. The world of reviews gives us the perfect set of data to build and test classifiers. In this analysis, I will train a sentiment classifier on the Rotten Tomatoes Kaggle data set.

Explanation of Cleaning, Preprocessing

The first steps done in the preprocessing is loading in the dataset. It is a .tsv and I use Pandas to load it into a data frame. There are many columns in this data frame, however, the only two of interest in this case are the phrases and sentiment. In the case of this dataset, sentences are all split into different phrases. There are 156,060 phrases with an accompanying sentiment label. The sentiment labels are ,0 - negative, 1 - somewhat negative, 2 - neutral, 3 - somewhat positive, and 4 - positive. All in all we have 5 classes. The first set of preprocessing done is to set all the phrases to lowercase. This will ensure that the same words, regardless of capitalization, are truly seen as the same.

The next thing to notice about the data is the massive imbalance in class labels. This is a visualization showing the distribution of class labels.



Obviously, we can see that by far majority of the phrases are labeled as neutral. There is also a steep drop-off as we move into negative and positive sentiment. Kaggle competitions host two datasets, a train set and a test set. The train set has labels and the test set does not. We are to train on the training set and submit predictions on the test set. However, on Kaggle, the training and test sets tend to have the same distributions. The goal is to see how my classification rates can stack up against the Kaggle community. Therefore I will not throw out any samples to fix this distribution and I will not combine any labels.

The text itself is littered with contractions. These are things like “I’ll”, “it’s”, etc. In order to accomplish my goal of building a custom sentiment calculator, there cannot be contractions. I use regex within a function to simply expand 8 common contractions. These are, ‘not’, ‘are’, ‘is’, ‘would’, ‘will’, ‘not’, ‘have’, and ‘am’. It will become apparent why this is important to do later in this report.

Finally, the text is tokenized using the NLTK tokenizer. This means that each phrase is broken up into words all held in a list. This is stored in a list of lists data structure where the outer list is the phrase and the inner list is the list of tokens that make up that phrase. It is important to note that phrases range from a single word to an entire sentences.

Feature Engineering

Polarity and Subjectivity - My Advanced features

My main goal in this entire analysis is building a custom sentiment polarity calculator. This would loop through the tokens in the phrase and calculate a custom polarity score. I build this on top of a lexicon and add parsing rules to get a better sentiment polarity score than provided by the lexicon itself. The lexicon I chose to build my custom sentiment calculator on is TextBlob. It has a great vocabulary and is seamless to use. Before implementing this I extract the subjectivity score of every phrase and save it in a list. This will eventually be used as a feature later on during classification. My custom sentiment polarity calculator expects a list of lists as input, where the outer list is the list of phrases and the inner list is the list of tokens within each phrase. I define two additional lists of words, I call the first list ‘flippers’, and the second list ‘negationwords’. ‘Flippers’ are the words ‘but’, ‘however’, and ‘although’. The ‘negationwords’ list are words like ‘not’, ‘neither', and ‘nor’, to name a few. At the beginning of each phrase I set a variable called ‘state’ to equal 0. As I loop through the tokens in a phrase, I pass the individual token into the lexicon and add its polarity score to the state. If the word however, is a ‘flipper’, it resets the state back to 0. This is amazing because this can now handle sentences like, ‘it seemed like it would be amazing, but it was terrible’. The lexicon on its own would not give this a negative polarity, but my custom calculator will. While iterating through the tokens if the token is

in the ‘negationwords’ list then the next token’s polarity will be subtracted from the state and not added to the state. This now allows the custom polarity calculator to handle negation and therefore sentences like ‘it was neither good not fun’. This is how my creation is able to handle negation. In helper.py I show a detailed analysis of what the output from my custom sentiment polarity calculator is and that of just using the lexicon on the same strings. Note, the TextBlob polarity is limited in the range -1 to 1, most negative to most positive. Mine is unbounded but we are most interested in the sign. Here are the 5 strings used, the simple lexicons score, my custom functions score, and my hand written label.

String	Simple Lexicon	My Function Custom Score	True Label
you are neither funny nor smart’	0.23	-0.46	negative
you are neither boring nor mean',	-0.65	1.3	positive
it seems very good, but its actually just awful',	-0.03	-1	negative
it seems really bad, but I loved it'	0	0.7	positive
unlike how terrible its predecessor was, this was actually amazing	-0.2	-0.4	positive

As is shown above, my custom sentiment polarity function does a much better job at handling the text. Both have a problem with the final sentence but that one is particularly difficult. These results were very encouraging. Therefore, I apply the function to every phrase. This calculates the custom sentiment score for each phrase to then be used as a predictor. It ends up being one of my best features for classifying the text.

There is an important additional note to mention about this. The type of Naive Bayes model used later does not accept negative numbers as input. Therefore I use unity-based normalization to force all sentiment scores to be non negative.

Bag of Words: Unigrams, Bigrams, and Trigrams

The next feature I create is Bag of Words. This is a very simple model that just builds a vocabulary from all the words in the corpus. Then a selection of the vocabulary set become the columns, and we have an indicator if a phrase contains that specific word. I use a CountVectorizor to accomplish this and more. This method allows us to pass in ngram_range, min_df, and max_df. The ngram_range sets if we want to look at only unigrams, bigrams, trigrams, or any combination of these. min_df sets the minimum number of documents the ngram needs to be in to be admitted as a feature. max_df is the same but it sets the maximum number of documents an ngram can be in to be admitted as a feature. Both min_df and max_df are used to limit the number of words used as predictors. Finally, we can also pass in the option to omit stop words. I show in the modeling the impact of removing stop words and a hypothesis as to why it does not work in this setting.

The combinations of features used within each model is explained in the following modeling and experimentation section.

Modeling and Experimentation

Disclaimer: The number one score on this dataset in the history of Kaggle is .76. This used extremely deep LSTM, GRU, and CNN combinations, as well as BERT and transfer learning.

Model 1

The first model is used to get some sort of baseline. It is important to remember that this is a 5 class classification problem that is very nuanced between classes. Expectations are low for any Bag of Words model without more advanced features. In this first model I use only unigrams and I do not remove stop words. The unigrams must show up in at least 100 phases and be in at most 750 phrases. This leaves the model with 1,128 words as predictors. Since we have many one word phrases, it is the case that with only using unigrams bigrams or trigrams there will be many observations with zero information available to make a prediction. If a word is not admitted as a feature, then we are asked to make a prediction on that word, our model will have nothing. The overall accuracy by taking the mean of 5 fold cross validation is .29. We can also the accuracy of each fold here.

[0.27527874 0.28463412 0.29347687 0.30241574 0.30626041]

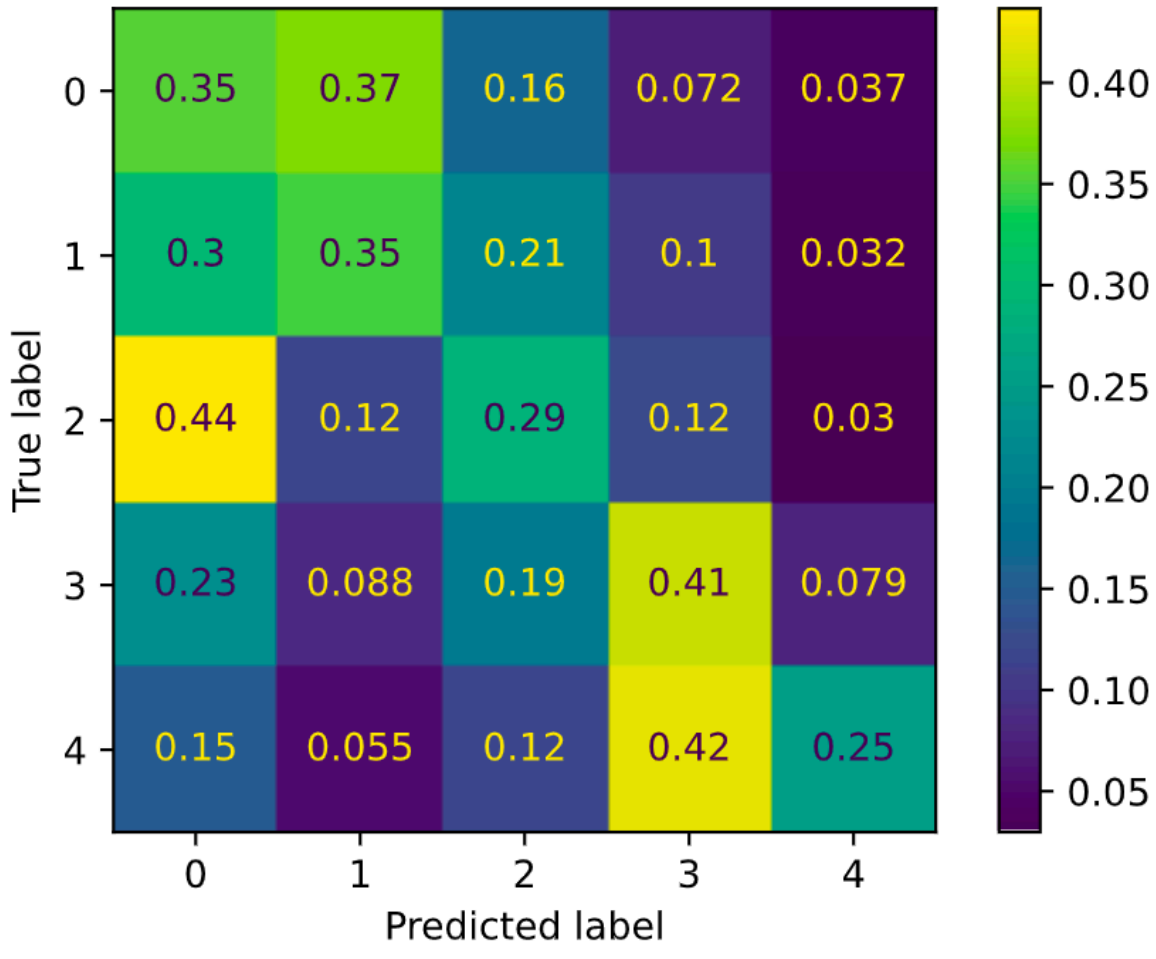
Although at first .29 seems terrible, it is realistic for using a simple Bag of Words on a dataset like this. It is also important to remember we did not throw out any difficult data and we did not collapse any of the 5 classes.

Below we can see output of a confusion matrix, precision, recall, f1-score, and support for each class, as well as a normalized confusion matrix.

```
Model 1
[[ 2140  2673  1372   612   275]
 [ 8394  8335  6371  3216   957]
 [35130 10353 21038 10292  2769]
 [ 7798  3290  6683 12196  2960]
 [ 1377   626  1296  3982  1925]]
              precision    recall  f1-score   support

         neg       0.04      0.30      0.07      7072
    some_neg       0.33      0.31      0.32     27273
     neutral       0.57      0.26      0.36     79582
     som_pos       0.40      0.37      0.39     32927
         pos       0.22      0.21      0.21      9206

    accuracy                           0.29    156060
   macro avg       0.31      0.29      0.27    156060
weighted avg       0.45      0.29      0.34    156060
```



People explain precision as the percentage of results that are related to that class. The problem with looking at overall accuracy is it does not account for class imbalance. Therefore, we have precision and recall to help understand our models performance. In this first model, it is all pretty bad. However, it will be very beneficial to compare the changes in precision and recall as we move on to more advanced models with more advanced features. The normalized confusion matrix is more visually pleasing than a standard confusion matrix. The lighter the color, the more density. We would like to see a strong pattern with a visible diagonal. Currently it is all over the place, but hopefully this will also improve with more advanced modeling. Just to show some of the results provided above, the precision for the 'negative' class is .04. This means we almost never predict 'negative' class correctly. The beautiful thing is that F1 scores combine both precision and recall. Therefore, we can monitor the change in F1 scores across all models and see how much class specific improvement we get.

Model 2

The second model uses Bag of Words, but this time it contains unigrams, bigrams, and trigrams. For a ngram to be used as a feature it must be in a minimum of 100 phrases and maximum of 750. We also do not remove stop words. This leaves the model with 1,669 words. We will still have the same problem as before. Since many phrases in the dataset are single word, that word may not be a part of the feature set. The classifier will then have 0 information on how to classify that phrase. The overall accuracy is .31 and here are the results of the 5 cross folds.

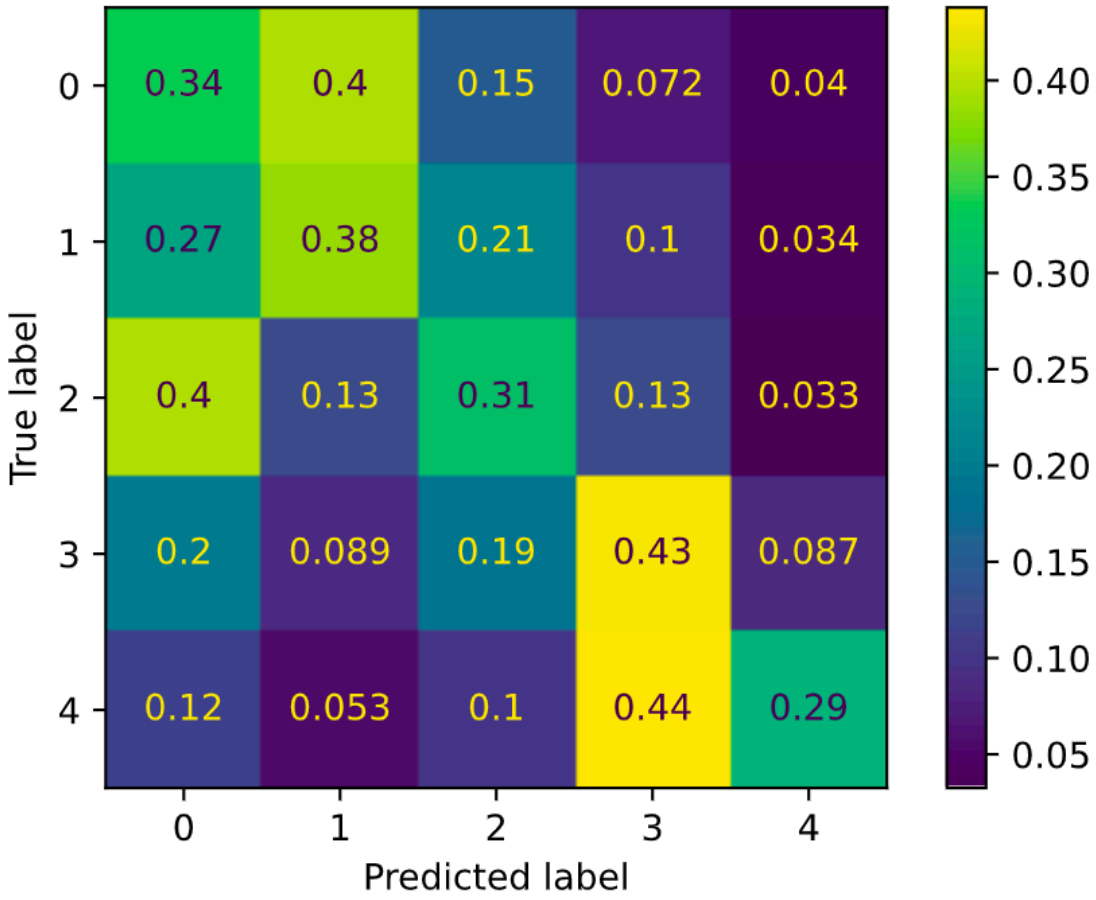
[0.29760989 0.30664488 0.31532744 0.31984493 0.32596437]

On the next page we will see output of the confusion matrix, precision, recall, f1-score, and support for each class, as well as a normalized confusion matrix.

```
Model 2
[[ 1975  2851  1329   576   341]
 [ 7448  9107  6567  3111  1040]
 [31840 11141 22728 10939  2934]
 [ 6743  3390  6772 12775  3247]
 [ 1127   596  1124  4085  2274]]
              precision    recall  f1-score   support

         neg       0.04      0.28      0.07      7072
    some_neg       0.34      0.33      0.34     27273
     neutral       0.59      0.29      0.38     79582
     som_pos       0.41      0.39      0.40     32927
         pos       0.23      0.25      0.24      9206

    accuracy                           0.31    156060
   macro avg       0.32      0.31      0.29    156060
weighted avg       0.46      0.31      0.36    156060
```



Like I explained above, if I do not remove observations to fix the class imbalance and I do no collapse the labels, a Bag of Words model will not cut it in getting a competitive result. We need a more advanced feature to give the classifier more information. In going from model 1 to model 2 we were only able to improve the model by a percent. To take a closer look let us analyze the F1 scores.

F1 scores	Model 1	Model 2
Negative	0.07	0.07
Somewhat negative	0.32	0.34
Neutral	0.36	0.38
Somewhat positive	0.39	0.4
Positive	0.21	0.24

As we can see, the F1 scores, which is a combination of both precision and recall barely improve. It will take more advanced feature engineering to make a significant improvement.

Model 3

This is the model with massive improvement. First, it uses unigrams, bigrams, and trigrams. To be admitted as a feature, the ngram must now be in at least 75 phrases and at most 800 phrases. I do not remove stop words. To solve this repeated problem I have had in the past two models I use my advanced feature, the custom sentiment polarity calculator. The beautiful thing about it is that it maintains the order of the words and gives an easy to use polarity score that can just be inserted into the model. This model has 2,392 words as predictors, and it has the custom polarity scores for each phrase, as well as a simple subjectivity score for each phrase. Remember, the custom polarity scores handle negation very well, as well as words that reset the polarity to zero, like ‘but’. There is a famous saying that nothing said before the ‘but’ has any meaning. Amazingly, this model has a mean accuracy of .512. Here are the results for each of the 5 folds.

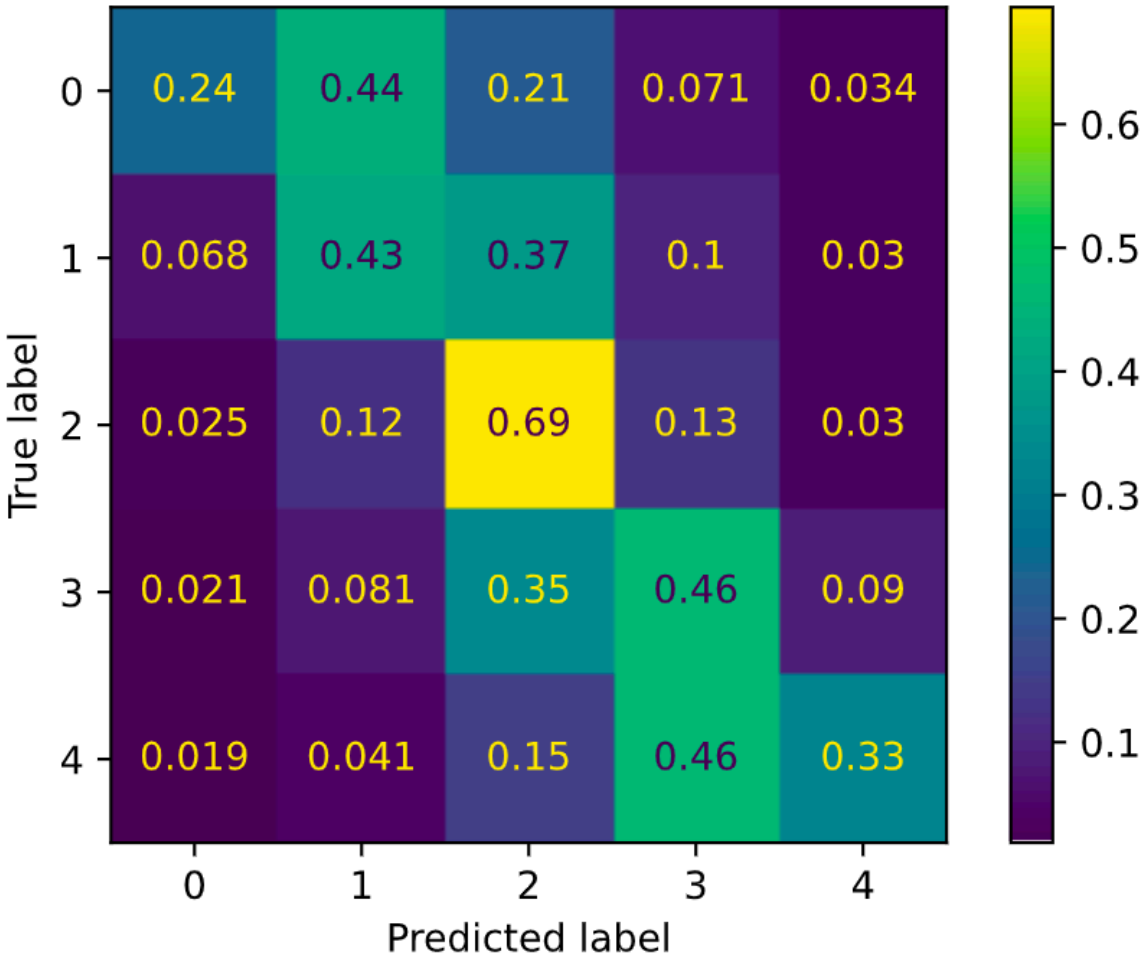
[0.52454184 0.50903499 0.51070101 0.50826605 0.50727284]

Below we will see output of the confusion matrix, precision, recall, f1-score, and support for each class, as well as a normalized confusion matrix.

```
[[ 1199  3210  1737   615   311]
 [ 2119 10063 10863  3208  1020]
 [ 2424 10748 52529 10998  2883]
 [  834  3263 11823 13549  3458]
 [  253   539  1556  4301  2557]]
              precision    recall  f1-score   support

         neg       0.18      0.17      0.17      7072
    some_neg       0.36      0.37      0.37     27273
     neutral       0.67      0.66      0.66     79582
     som_pos       0.41      0.41      0.41     32927
         pos       0.25      0.28      0.26      9206

    accuracy                           0.51    156060
   macro avg       0.37      0.38      0.38    156060
weighted avg       0.51      0.51      0.51    156060
```



Now, I have massive improvement in the modeling. The presence of the custom polarity score for each phrase has improved the accuracy tremendously. However, we can see that the improvements are class specific. By looking at the precision and recall, the new model is able to better classify negative and neutral sentiment. Between Model 2 and 3 we see mainly an improvement with the negative class and neutral class. We can very easily directly compare the metrics by using F1 table I created below.

F1 scores	Model 2	Model 3
Negative	0.07	0.17
Somewhat negative	0.34	0.37
Neutral	0.38	0.66
Somewhat positive	0.4	0.41
Positive	0.24	0.26

It is very interesting to note that F1 scores improved a huge amount for Neutral. My hypothesis is that this is because the model now has some sort of information for every observation. We can also see here that it was really only the ‘neutral’ and ‘negative’ classes that improved significantly. Something else that I am very happy about is how the model misclassified things. For example, if we look at any true label in either confusion matrices, the most mistakes are made on the adjacent class. For example, ‘negative’ is mostly confused with ‘somewhat negative’. This holds true throughout the classes. This is pretty understandable. If a human had to try and classify the difference between somewhat negative and negative there could be a lot of disagreement. The differences between the classes are very nuanced. Another way of seeing this is that in the normalized confusion matrix, we see the bright color make a diagonal from the top left to the bottom right. This classification algorithm is able to classify correctly 51% of the time, but majority of the misses are the next class over. Which is a very understandable mistake. When we think about how unbalanced this dataset is, the fact that we have five classes, and the fact that there are very small differences between the classes means that a score of 51% is pretty remarkable. There are many things that can arbitrarily increase the score, collapsing classes and getting rid of all phrases under a certain length. However, this is would be an oversimplification of the problem.

Model 3.5

This model uses the exact same data as the last model expect it now removes stop words from the word features. The purpose of this experiment was to see if there is a major difference between the model with and without stop words. However, there is almost no difference. In fact the classification rate when removing stop words is slightly worse, 50.9%. The reason for this is that the massive jump in classification accuracy was the custom polarity feature.

Model 4

For this next model, I use the data frame from model 3, the best model. However, instead of using Naive Bayes I use a Random Forest. The Random Forest had a classification rate of 56%, a great improvement on the 51% in Model 3. The only different thing between the two is the algorithm used, they both are trained on the exact same data. The idea here is to simply show that we can maybe get an extra few percent in classification accuracy by simply choosing a different model. Since the time to train the model is much longer with a Random Forest, I reduce the number of folds to 3. This makes the time to run the entire script more manageable. In the code submission it is commented out. Many times running the random forest my computer would crash. Therefore, it is commented out and anyone looking at the code can uncomment it to verify results.

Model 5

I love machine learning and deep learning for the beauty in the algorithms. The problem with the analysis up to now is that creating a bag of words destroys the order of the words in the phrases, and in doing so it looses its beauty. Other than the custom polarity score, there is no idea of the phrase remaining whole. There are however, deep learning algorithms that maintain the order of the words. One of these is a LSTM or a Long Short-Term Memory network. These networks are built to handle sequential data, like text. The architecture of my LSTM implemented is shown below, along with the validation accuracy on each epoch.

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_3 (Embedding)      (None, 200, 32)           489024
_________________________________________________________________
spatial_dropout1d_2 (Spatial (None, 200, 32)           0
_________________________________________________________________
lstm_8 (LSTM)                (None, 50)                16600
_________________________________________________________________
dense_2 (Dense)              (None, 5)                 255
=================================================================
Total params: 505,879
Trainable params: 505,879
Non-trainable params: 0
```

```
Epoch 1/5
3902/3902 [==============================] - 535s 136ms/step - loss: 1.0983 - accuracy: 0.5660 - val_loss: 0.9932 - val_accuracy: 0.5968
Epoch 2/5
3902/3902 [==============================] - 524s 134ms/step - loss: 0.8078 - accuracy: 0.6720 - val_loss: 0.9861 - val_accuracy: 0.6073
Epoch 3/5
3902/3902 [==============================] - 524s 134ms/step - loss: 0.7567 - accuracy: 0.6936 - val_loss: 0.9885 - val_accuracy: 0.6044
Epoch 4/5
3902/3902 [==============================] - 540s 138ms/step - loss: 0.7189 - accuracy: 0.7090 - val_loss: 1.0068 - val_accuracy: 0.6075
Epoch 5/5
3902/3902 [==============================] - 564s 145ms/step - loss: 0.6903 - accuracy: 0.7208 - val_loss: 1.0105 - val_accuracy: 0.6006
```

The network has 505,879 trainable parameters. This is a massive increase in trainable parameters than any classical machine learning algorithm like Naive Bayes. The preprocessing in an LSTM is involved. The phrases in the dataset need to be transformed into numbered representations, where we have a key value pair of words and their corresponding number. Once the phrases are turned into numbers, the LSTM will treat the numbers as a sequence, thus preserving the meaning from word to word. We will also set a sort of window size, this is 200 in our case. This means we consider sequences of 200 words, which is more than enough for this task. Up to here, this is the preprocessing and embedding layer of the network. Next, our LSTM has a layer of dropout, this randomly turns off neurons so the LSTM does not overfit. Then, these sequences are actually passed into a LSTM with 50 units. Finally, we have a fully connected dense layer at the end with 5 nodes and ‘sigmoid’ activation function. It has 5 nodes because this is a 5 class problem. It also trains for 5 epochs, an epoch is a full iteration over the data. The implementation in tensorflow keeps a holdout set that the LSTM is not exposed to, this is how the validation accuracy differs from the accuracy. As can be seen above, the accuracy goes up with each epoch but the validation accuracy stops improving after the second epoch. It would be interesting to let the LSTM run for 10-15 epochs and see if the validation accuracy converges to anything

greater than 61%. Either way 61% is an unbelievable result and improvement on our previous best result of 51%.

Subsequent LSTM Modeling

A general idea is that the wider a neural network the more features it can understand, the deeper the more nuanced features it can understand. Therefore, I trained two more neural nets. In the first I increased the number of units from 50 to 75, in the second I reduced the units from 50 to 24 but added 2 more layers. The goal of this was to compare the 50 unit LSTM discussed in the preceding paragraph with a wider neural net and deeper neural net. Both of these neural nets are commented out in code, to run them simply uncomment them. All of the neural nets perform between .605 and .61. None of these neural nets showed a significant difference between one another. My assumption is that more data is needed to have a deeper or wider neural net, or simply more time and patience to let it train for 24 hours plus. Either way, my result of .6073 after simply 2 epochs is incredible. It does however, point to the fact that it can be improved, either with more epochs or a deeper or wider network.

Areas of Improvement

As a main area of improvement, I have a lot of ideas to change the custom sentiment polarity calculator. A simple improvement is that I can add more words to the ‘flippers’ and ‘negationwords’ lists. They do a good job as it is, but there are more potential words to add. Another idea is to add amplifiers. This would be a new list of words like ‘very’, ‘extremely’ etc. If the calculator encountered one of these words in the sequence it would multiply the next words polarity by a factor x, where $x>1$. This would make the custom polarity calculator more dynamic and able to differentiate between more nuanced language. Another area of improvement for this would be POS tagging. Instead of making a mathematical operation on the next word after a word in a ‘negationwords’ list, it would apply the mathematical operation to the next adjective. This may be difficult, but it could drastically improve the results.

Conclusion

Sentiment is a domain specific task. Although we can rely on pretrained models, at a very advanced level we would need to use pretrained neural nets in conjunction with our own. For example, a pre trained neural network might predict that the sentence, ‘I was terrified the whole time’ as negative. However, if this were a review for a horror movie it would probably be five stars. Learning how to train our own sentiment classifiers for a domain specific task is essential in the field of NLP. There is so much crowd sourced review data that a task like this is essential in many business fields.