Classifying IMDB Movie Reviews

Dan Caley & Liza Minisci | NLP | 12/20/2021

Table of Contents

Overall 2

Reviewing the Data 3

Tokenized data 5

Filtered and tokenized data 7

Pre-processed and tokenized data 7

Feature Engineering 12

Bag of Words & Unigrams Features 12

Bigrams Features 13

Part-of-speech Features 13

Sentiment Lexicon (LIWC) 15

Subjectivity 15

Experiments 16

Bayes Classifier 16

Cross Validation 16

Sci-kit Learn - Advance Task 16

Results 17

Final Thoughts 20

Notes:

A copy of the juptyer notebook will be included in the submission. If you would like to also access the colab document please go to the link here.

Overall

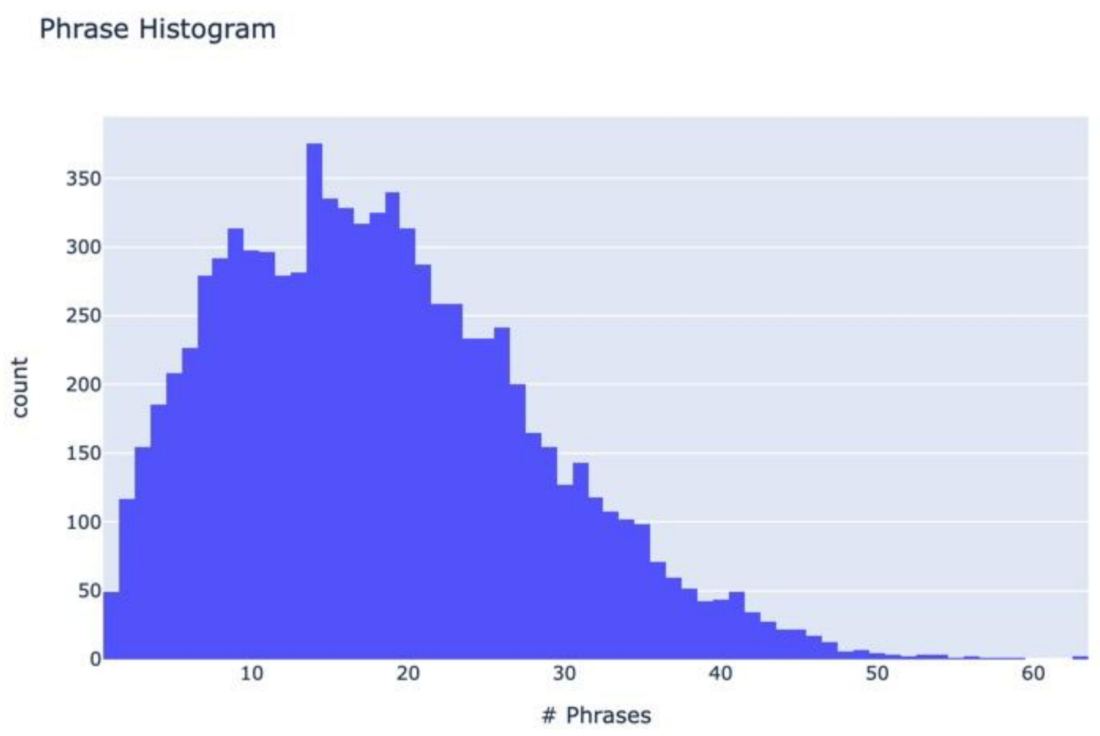
The goal we set out today is to predict the rating of a list of comments using classification algorithms and different cleaning techniques. The exhaustive list of items includes tokenizing, filtering, pre-processing the word list. Then apply feature engineering techniques that span Unigrams, Bigrams, and all the way to subjectivity. Then use multiple models like Bayes Classifier, Random Forest, and Support Vector Machine Classification. To round off the analysis, we used cross-validation to help prove the results will help in the prediction task.

No stone was left un-turn, with 30 different permutations of accuracy scores created based on this analysis's filtering, feature engineering, and models. Let’s jump in and see how well we can apply a rating to a comment?

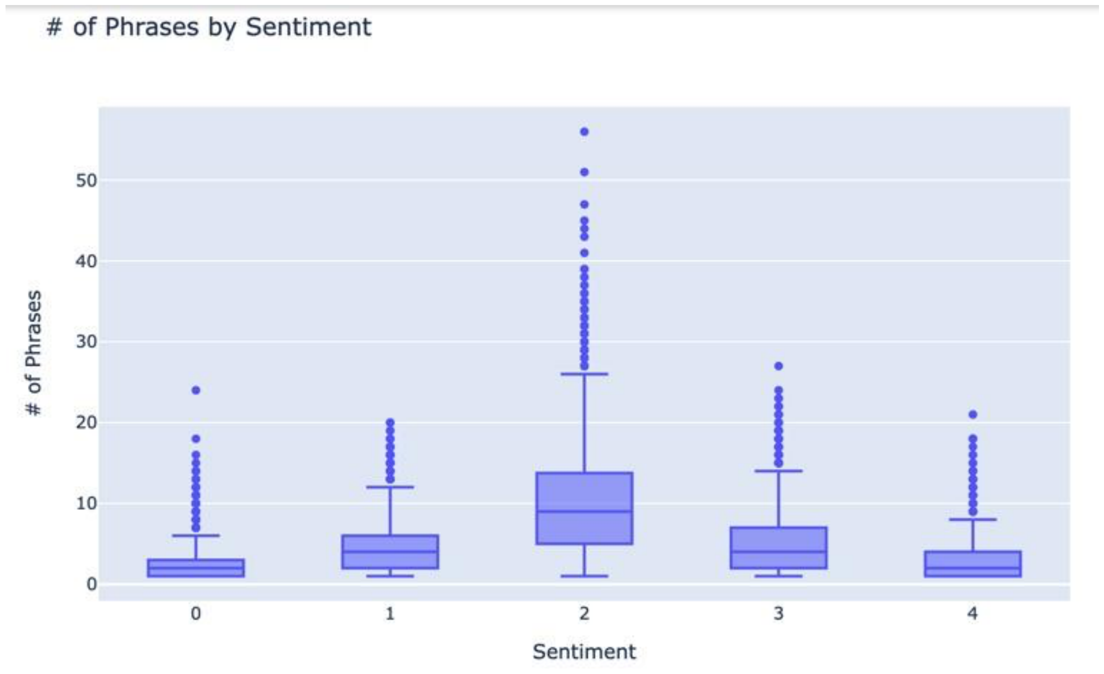
Reviewing the Data

The dataset we chose to review and analyze was the Kaggle movie review list. This dataset was produced for the Kaggle competition, which uses data from the sentiment analysis by Socher et al. The data was originally taken from the Pang and Lee movie review corpus based on reviews from Rotten Tomatoes website. Socher's group used crowd-sourcing to manually annotate all the sub-phrases of sentences with sentiment label scoring. The ranges included the following: “negative”, “somewhat negative”, “neutral”, “somewhat positive”, “positive”.

Before reviewing the original dataset we needed to import all the necessary packages needed for preprocessing and filtering. First we created training and test data frames using the given train.tsv and test.tsv files that were given to us. We wanted to review exactly how many unique sentences and compare them to the total number of sentences supplied in the training dataset. The total number of full sentences (also considered “unique_sentences”) were 8520, out of a total of 156,060 listed phrases (also considered “total_sentence”). That is approximately 18 times the amount of unique sentences. Next, we assign a “FullSentenceId” to each phrase, while grouping them by the SentenceId, this way we can see which smaller phrases are assigned to which completed full sentence. After putting the training data into a table with the new columns we could see that this dataframe was similar to a break down tree, similar to the grammar context previously learned in the course. To better review this discovery, we created a phrase histogram that compared the count along the y-axis and number of phrases along x-axis. The number exceeded above 350 count as its peak early on at approximately 15 phrases as you can see below.

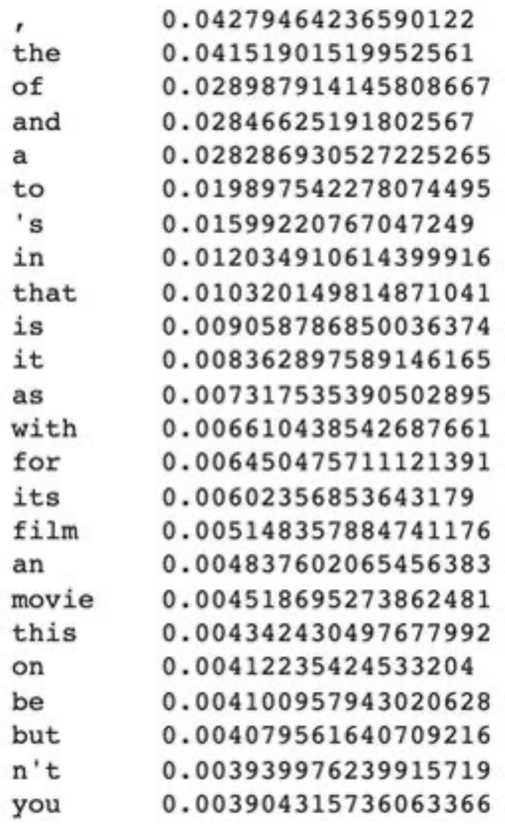


To continue our visualization analysis of the training data, we then compared the number of phrases to the sentiment scores that were assigned to them. As scores 1 and 4 were the least, the score of 2 had the largest upper and lower bounds with exceeding whiskers past the 50 count of assigned phrases as you can see below. This essentially confirmed that there was a lot of cleaning and filtering that needed to be done prior to creating a new model to run our experiments on.



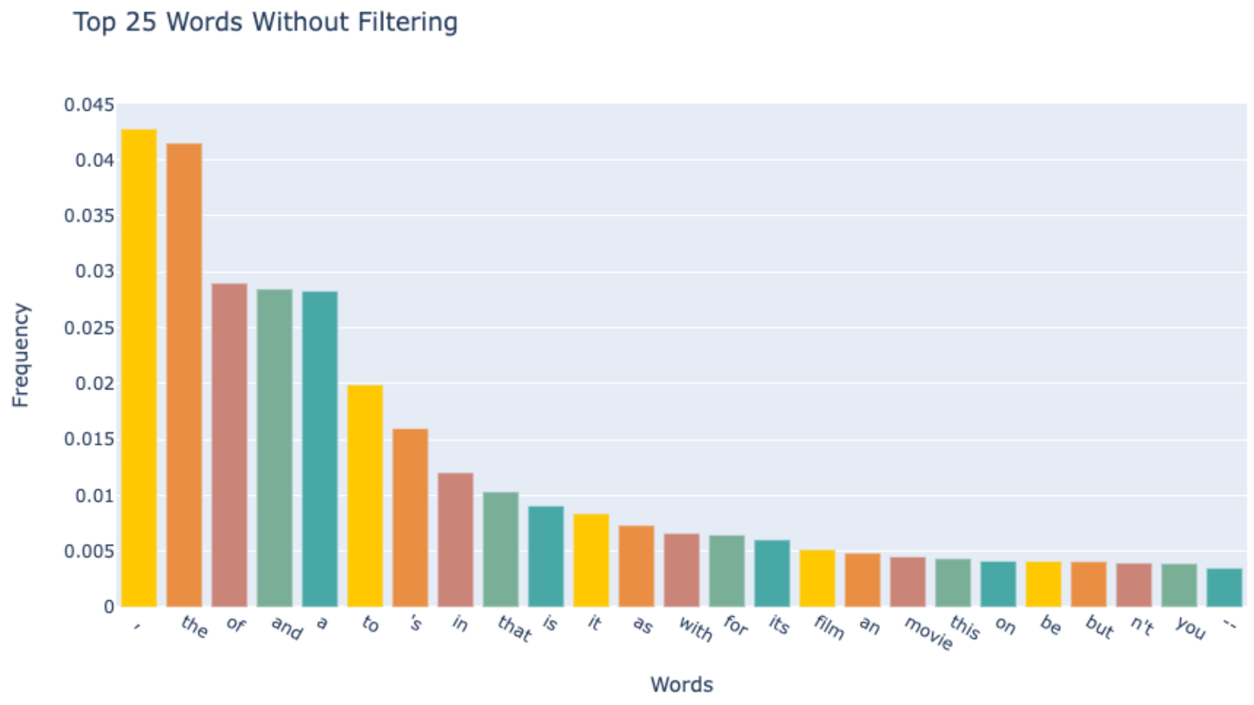
Tokenized data

In order to tokenize our data we first join the training data frame creating the “Phrase” column into ‘characters.’ Using a nltk.word function that tokenizes the testtext vector to make our tokens. And to make sure all of our characters are seen the same, we made sure they were all lowercase into a new vector named testwords. The length of testtokens outputted 981,478 total words without filtering. We then wanted to dive deeper reviewing our bag (“test words”) of words by using the FreqDist function. 25 of the most common words based on the frequency were displayed to see the depth of filtering that needs to be done.



```
,        0.04279464236590122
the      0.04151901519952561
of       0.028987914145808667
and      0.02846625191802567
a        0.028286930527225265
to       0.019897542278074495
's       0.01599220767047249
in       0.012034910614399916
that     0.010320149814871041
is       0.009058786850036374
it       0.008362897589146165
as       0.007317535390502895
with     0.006610438542687661
for      0.006450475711121391
its      0.00602356853643179
film     0.005148357884741176
an       0.004837602065456383
movie    0.004518695273862481
this     0.004342430497677992
on       0.00412235424533204
be       0.004100957943020628
but      0.004079561640709216
n't      0.003939976239915719
you      0.003904315736063366
```

As you can see below, one of the top results was a comma and other “words” such as “‘s” or “n’t” were considered in that top 25 of most common. Therefore, filtering needs to be done in order to get a better accuracy of our dataframe. Below is a better visualization of the word frequency without filtering:



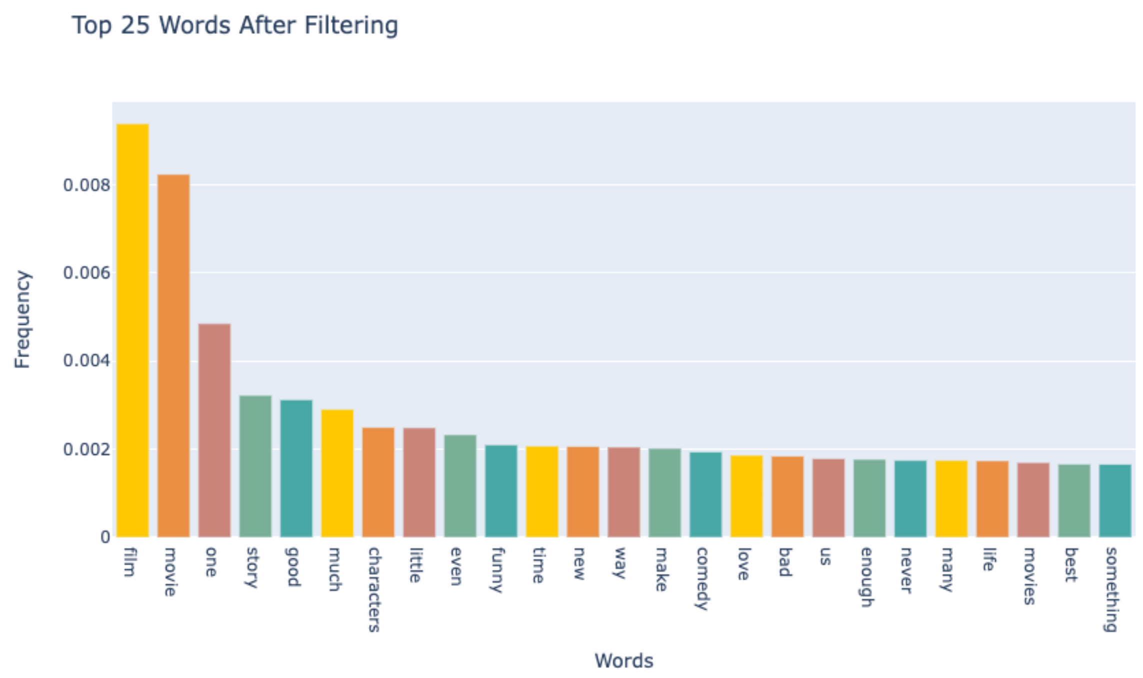
Top 25 Words Without Filtering
Frequency
0
0.005
0.01
0.015
0.02
0.025
0.03
0.035
0.04
0.045
,
the
of
and
a
to
's
in
that
is
it
as
with
for
its
film
an
movie
this
on
be
but
n't
you
--
Words

Filtered and tokenized data

First step we decided to do was remove punctuations from our dataframe using the list(puncuation) function and apply that to the testwords vector we already had stored. The length of words decreased to 9,30730. The frequency distribution of our testwords using the 25 most common were compared cause a slight change. Next, we remove any non-alphabetical characters in our test words with Regular Expression practices by using the re.compile('^[^a-z]+$') function within defining an alpha_filter. Doing this significantly causes 10% of our test words to decrease. However, we still need to remove periods as well. We repeat this re.compile('\ |\?|\.|\!|\/|\;|\:') function toward the newly created alphatestwords vector. This also decreases our number of testwords again.

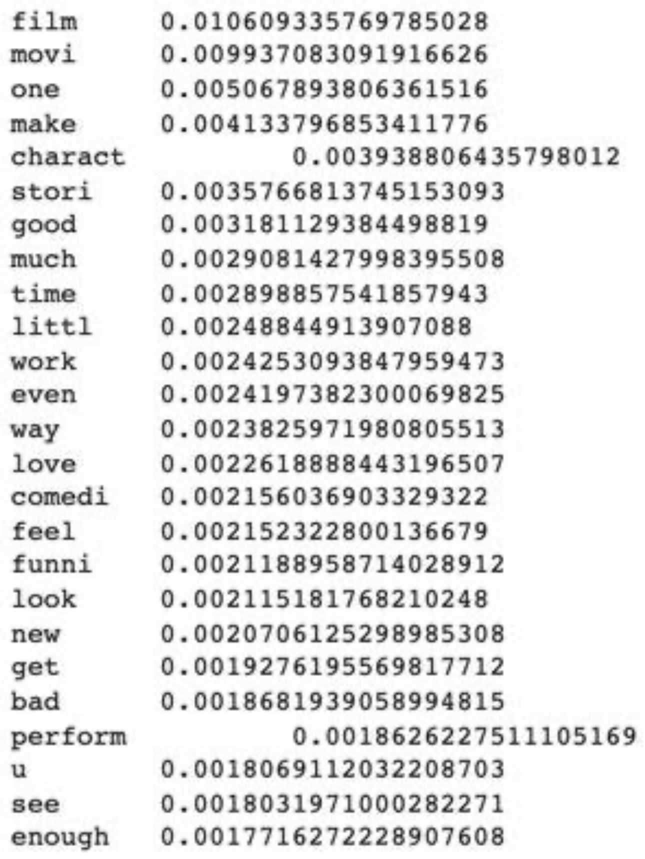
Pre-processed and tokenized data

Now that unnecessary characters have been removed from the training dataset, we have to focus on the words tokens themselves. A lot of the time, English stop words decrease accuracy or natural language processing so it is best to remove them to better our models. We obtained the list of English stop words from the NLTK corpus (total of 179). Then, we created a new vector of stoppedtestwords that removes the nltkstopwords from the alphatestwords. A drastic decrease in total words was applied here making a total of 568,580 words in the training dataset. Another frequency distribution was applied to the new total words vector. Some characters that are not considered valid words were displayed in the most common list. Taking another list of morestopwords taken from a previous lab was used in addition to other characters from the frequency distribution and then added to the nltkstopwords. Ultimately, this help increase the frequency distribution for a majority of the most common characters as you can see below:



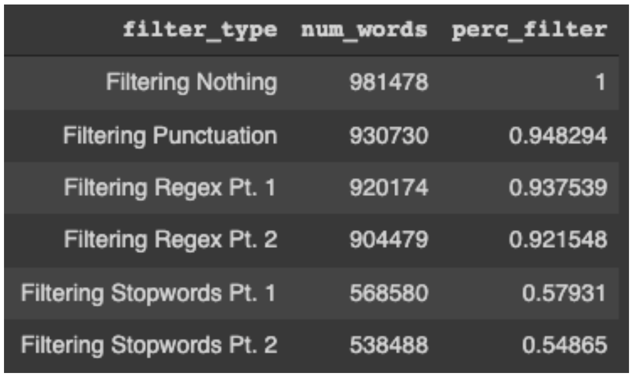
Another form of tokenizing data and pre-processing is stemming and lemmatizing words. This essentially is mering the words together in order to increase the frequency overall. For stemming we apply the porter.stem function toward the stoppedtestwords and join that with the total words vector. For lemmatizing we apply the word_lemma.lemmatize function as well to the previously created stemmedtestwords. This exercise did not decrease the number of words but did help improve the frequency. That is due to words being combined together and therefore increased the occurrence of the words.

The picture below illustrates how stemming and lemmatizing moved the word film from 0.008 to 0.01.

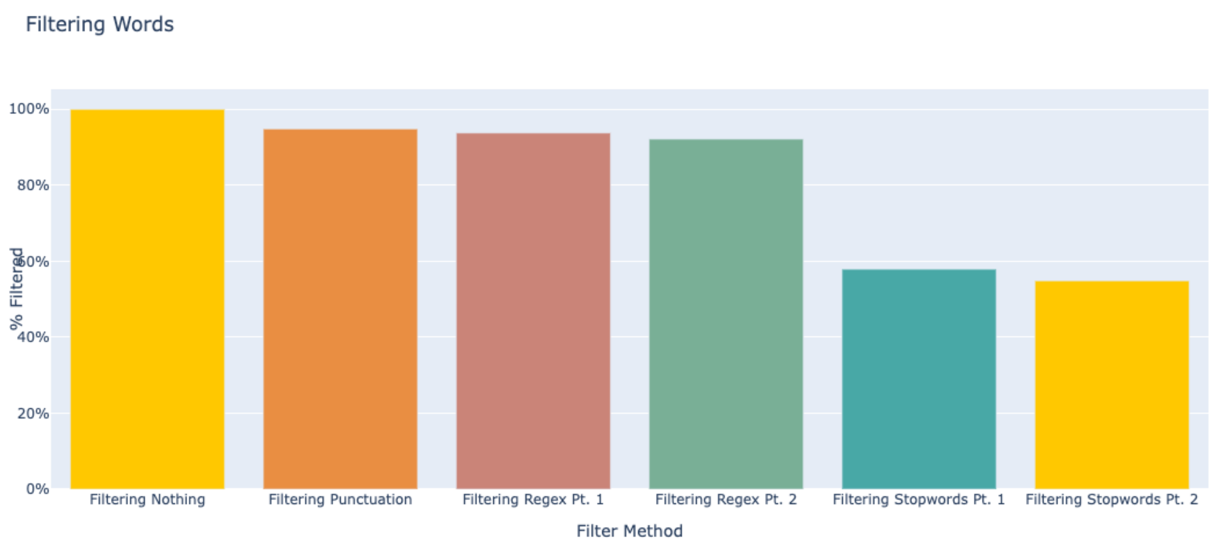


```
film	0.010609335769785028
movi	0.009937083091916626
one	0.005067893806361516
make	0.004133796853411776
charact		0.003938806435798012
stori	0.0035766813745153093
good	0.00318129384498819
much	0.0029081427998395508
time	0.00289885754185794З
littl	0.00248844913907088
work	0.0024253093847959473
even	0.0024197382300069825
way	0.0023825971980805513
love	0.0022618888443196507
comedi	0.002156036903329322
feel	0.002152322800136679
funni	0.0021188958714028912
look	0.002115181768210248
new	0.0020706125298985308
get	0.0019276195569817712
bad	0.0018681939058994815
perform		0.0018626227511105169
u	0.0018069112032208703
see	0.0018031971000282271
enough	0.0017716272228907608
```

In summary the chart and graph below shows the effect of each filtering technique on that data. What can be seen is that filtering stop words nearly cut the total words in half with the other technique cutting about 8% from the data.

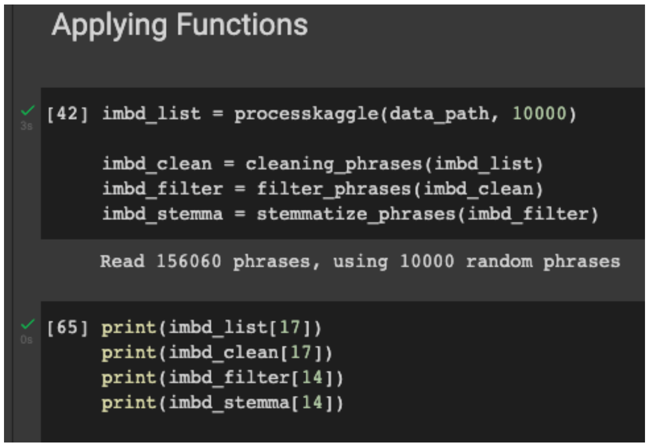


filter_type	num_words	perc_filter
Filtering Nothing	981478	1
Filtering Punctuation	930730	0.948294
Filtering Regex Pt. 1	920174	0.937539
Filtering Regex Pt. 2	904479	0.921548
Filtering Stopwords Pt. 1	568580	0.57931
Filtering Stopwords Pt. 2	538488	0.54865


Filtering Words
100%
80%
60%
40%
20%
0%
% Filtered
Filtering Nothing
Filtering Punctuation
Filtering Regex Pt. 1
Filtering Regex Pt. 2
Filtering Stopwords Pt. 1
Filtering Stopwords Pt. 2
Filter Method

Pre-processed, filtered, and tokenized data

The Final Step was to build a function that leverages the pre-processing and filtering techniques discussed above to then apply to each individual sentence. First, we defined a processkaggle function to read the kaggle training file, train and test a classifier. This incorporated a limiting argument, looping over lines, picking a random sample of length, and then printing the phrase data. A cleaning_phrases function was then defined by creating a list of phrase documents that first tokenizes the list of words, then makes those tokens lowercase, removes punctuations, and removes any tokens that are less than 1 at length. The next function created was filter_phrases which incorporated tokenizing again, removing non-alphabetical characters, removing words under three characters, removing all stopwords, and again removing any tokens that are less than 1 at length. Lastly, the stemming and lemmatize techniques were combined into one stemmatize_phrase function which tokens the phrases again, stemming, lemmatizing, and returning a final token list. Below are screenshots of applying these condensed functions to our imdb dataset and printing out the final breakdown of these words.


Applying Functions

```
[42] imbd_list = processkaggle(data_path, 10000)

     imbd_clean = cleaning_phrases(imbd_list)
     imbd_filter = filter_phrases(imbd_clean)
     imbd_stemma = stemmatize_phrases(imbd_filter)

     Read 156060 phrases, using 10000 random phrases

[65] print(imbd_list[17])
     print(imbd_clean[17])
     print(imbd_filter[14])
     print(imbd_stemma[14])
```

Output:

```
['see a study in contrasts ; the wide range of one actor , and the limited range of a comedian', '2']
(['see', 'a', 'study', 'in', 'contrasts', 'the', 'wide', 'range', 'of', 'one', 'actor', 'and', 'the',
'limited', 'range', 'of', 'a', 'comedian'], 2)
(['see', 'study', 'contrasts', 'wide', 'range', 'one', 'actor', 'limited', 'range', 'comedian'], 2)
(['see', 'studi', 'contrast', 'wide', 'rang', 'one', 'actor', 'limit', 'rang', 'comedian'], 2)
```

What can be seen here is that the first sentence is not tokenized. Each of these sentences have the rating at the far right. The next 3 in the output above shows the sentence tokenized with everything lowercase and punctuations removed. The third sentence filters any stop words from NLTK and the custom list created. The last sentence shows the Stemmatize and Lemmatize technique. This can be seen with the word “study” changing to “studi”.

Feature Engineering

The objective for feature engineering is to create the featuresets by passing the sentences into a function that will add a true or false value for every unigram, bigram, Part of Speech etc. A lexicon of different features like part of speech, sentiment, or subjectivity can be applied. More of this will be discussed further in the document.

Bag of Words & Unigrams Features

This type of feature tokenizing every word in the sentence applies a True or False rating. At the end of Unigram dictionary contains the sentiment. This can be seen in the chart below.

```
({'V_film': False,
  'V_movi': False,
  'V_one': False,
  'V_stori': False,
  'V_make': False,
  'V_charact': False,
  'V_time': False,
  'V_good': False,
  'V_even': False,
  'V_work': False},
 1)
```

To begin our feature engineering we first get all the words from our dataset imbd_stemma and put it into a frequency distribution. The output of the printed length returns 7056. Out of that total, we take 1000 of the most common word items and create a word features vector. Next, we define a document_features function that takes the keywords of a document for bag of words and unigram baseline. Imbd_clean is stored back into a document vector to store so we can replace this instead of all the individual features. After creating a new dataframe called feturesets that includes the previously stored documents, keyword features and category features. Doing this increases the length up to 9998.

Bigrams Features

A bigram group two words together in a phrase and then assigns a rating of True or False. For example “Dan is the man” would translate to “Dan is”, “is the”, “the man”. Then a chi squared measure is used to find the top 500 bigrams. The feature set will then include just these word phrases.

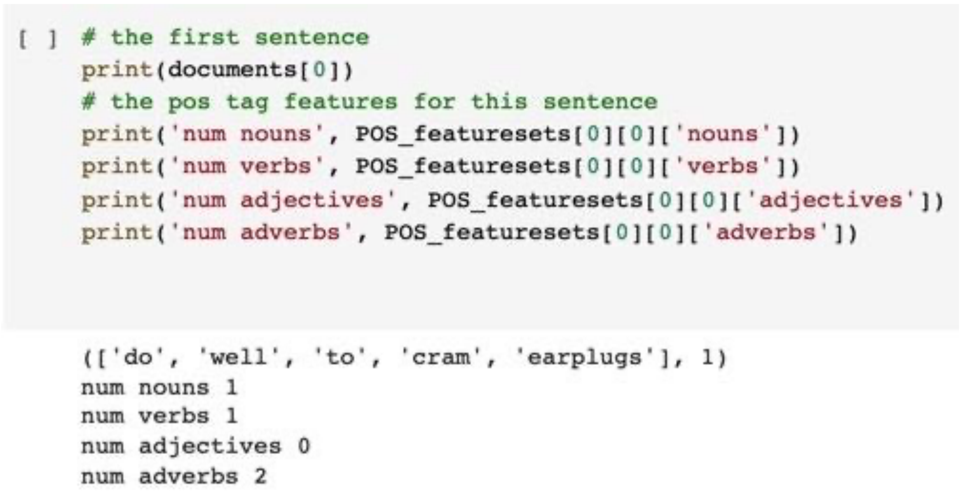
We used the nltk.collaction.BigramAssocMeasures function on all the words in our sequence and stored that into a finder vector. Next, we define the top 500 bigrams using the chi squared measure and store that into a bigram_features vector. We then define a bigram_document_feature using both document, word_features, and bigram_features described beforehand. Using this function will create feature sets for all the sentences. To test these bigrams we use the train_set again to test a classifier and report the accuracy at 45.6%

```
({'B_10-year_delay': False,
'B_18-year-old_mistress': False,
'B_22-year-old_girlfriend': False},1)
```

Part-of-speech Features

A lexicon was used here to understand how the part of speech might play a role in a sentence and in a model. In the second picture below we can see the number of nouns, verbs, adjectives, and adverbs for the first sentence.

The POS feature takes a document list of words and returns a feature dictionary in order to run the default POS tagger (Standord tagger) on the document. This way the feature can count 4 types of POS tags to use as features, such as, nouns, verbs, adjectives, and adverbs. The length of POS_features outputted 1004. Below is an example of the first sentence.



```
[ ] # the first sentence
    print(documents[0])
    # the pos tag features for this sentence
    print('num nouns', POS_featuresets[0][0]['nouns'])
    print('num verbs', POS_featuresets[0][0]['verbs'])
    print('num adjectives', POS_featuresets[0][0]['adjectives'])
    print('num adverbs', POS_featuresets[0][0]['adverbs'])
```

```
    (['do', 'well', 'to', 'cram', 'earplugs'], 1)
    num nouns 1
    num verbs 1
    num adjectives 0
    num adverbs 2
```

Sentiment Lexicon (LIWC)

This feature is also considered linguistic inquiry and word count. This form of text analysis essentially calculates the degree to which the various categories in our kaggle training dataset of words are used in text. This lexicon assigned a positive or negative rating then adds to either a pos or negative list in the dictionary. A sample of data can be seen in the jupyter notebook and is very similar to the other examples above.

Using the sentiment_read_LIWC_pos_neg_words.read_words function was stored into a path vector, which was then separated into a positive and negative path. We created a LIWC_feature that includes word counts of the subjectivity words. The negative features will have a number of weakly negative words and 2 times the number of strongly negative words. The final length of LIWC_featuresets outputs 9998.

Subjectivity

To begin engineering subjectivity, we imported in the necessary readSubjectivity libraries from the Kaggle movie reviews. Once the SLpath was stored into a vector, we defined the features that included word counts of the subjectivity words plus 2 times the number of strongly positive and negative words. Keeping in mind that positive features have similar definitions, not counting the neutral words. If the word was tagged as “weaksubj” or “strongsubj” it would now be considered positive or negative. The final length of SL_features outputted 1002.

Experiments

Bayes Classifier

Once our condensed features were created, we decided to test out our training_set and test_set using naive Bayesian Classifier. After taking 500 random samples of featuresets within the train_set and test_set, we used the NLTK Naive Bayes Classifier function and evaluated the accuracy and it came out to be 53.8%. Using the bigram_featuresets the accuracy resulted in 54.4%. Using the POS_featuresets to train and test the classifier our accuracy increased slightly to 54.6%. Lastly, the subjectivity lexicon resulted in a 56.4% accuracy and sentiment LWIC featuresets resulted in 53.8%.

Cross Validation

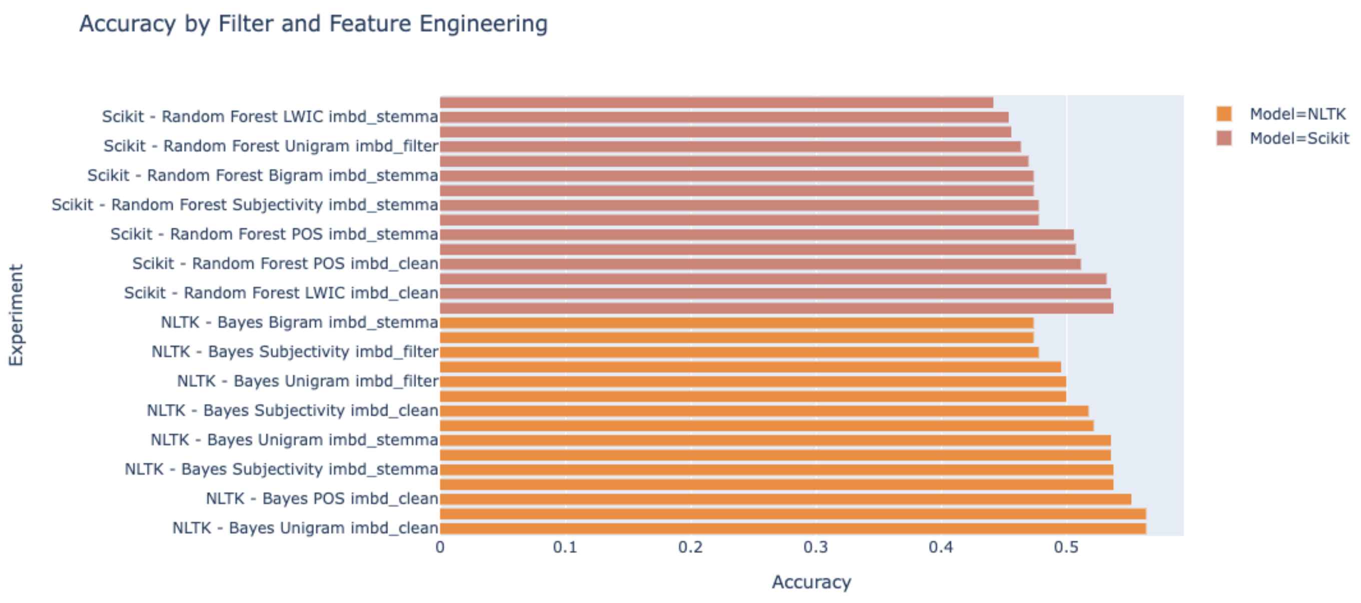
In the cross validation experiment we created a cross_validation_accuracy function that the number of folds and feature sets, then iterates over the folds. By using different sections for training and testing, this function will print the accuracy for each fold and the average accuracy at the end of the output. When using the unigram word features and using a number fold of 3 the mean accuracy resulted in 48.0% with each fold size being 3332. When using the bigram_featuresets and the same number of folds, the mean accuracy also resulted in 52.1%. Using the POS_featuresets within the cross_validation_accuracy function, the mean accuracy resulted in 53.9%. Lastly, using the subjectivity lexicon, the mean accuracy resulted in 53.1%.

Sci-kit Learn - Advance Task

Sciki-Learn is another machine learning program used for word efficiency models like NLTK. However, it was used within our program to see if the applications and accuracy of our models were in comparison to our other classifier and feature combinations. The first classifier we used was a random forest using the POS_featuresets and that accuracy result was higher than the NLTK with 57.8%. The next one we used was the Support Vector Machine classifier using the same POS_featuresets and that resulted in 54.6%, similarly to the NLTK.

Results

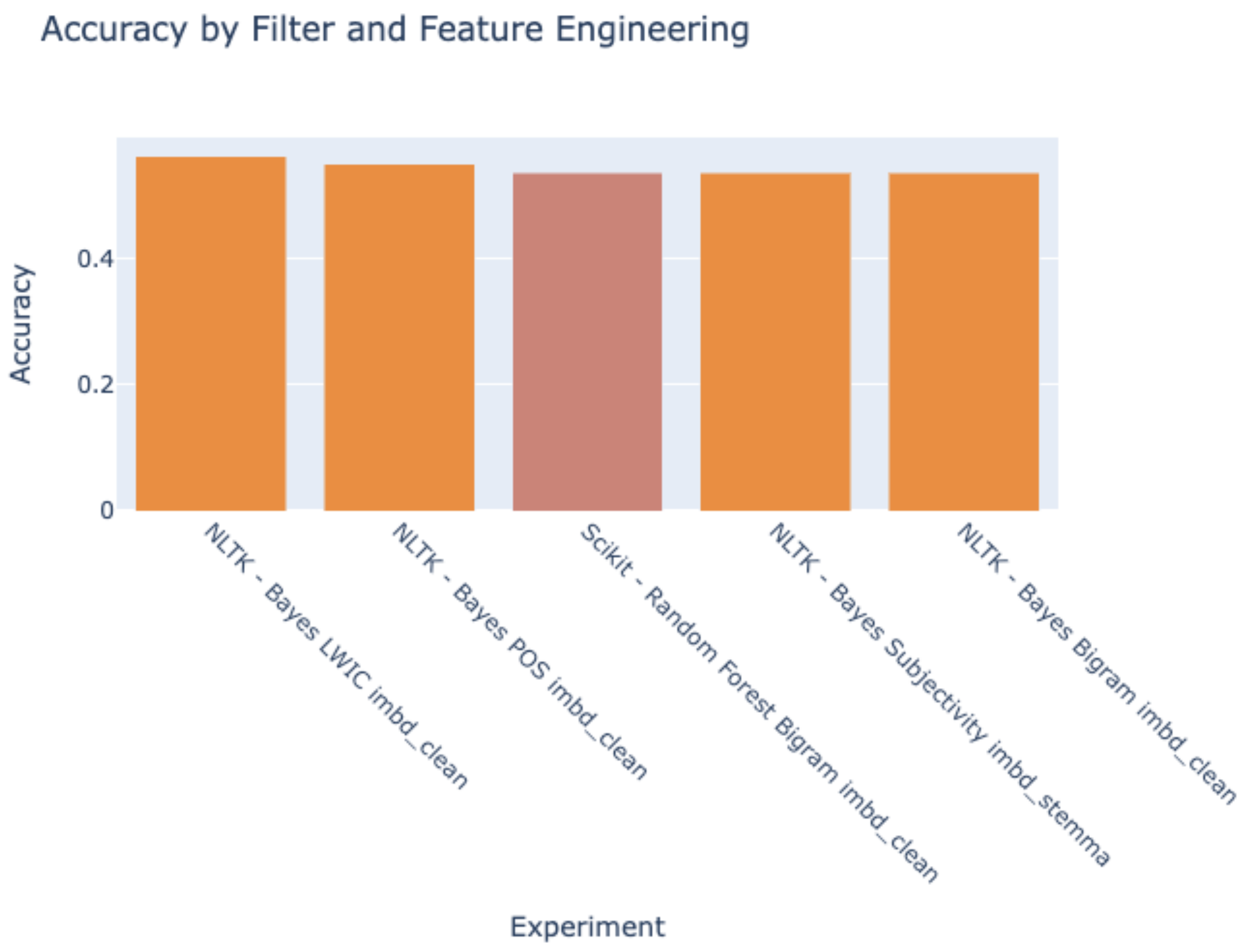
After building 3 filters, 5 feature engineering, and 2 models ran every permutation of possible experiments. This included 30 different experiments between Bayes NLTK and Random Forest Scikit learn. The results ranged from 56% to 44% with each new experiment incrementally improving from the other. The overall results are still poor with accuracy scores equating to nearly 50/50 shot in predicting the right rating based on comments.

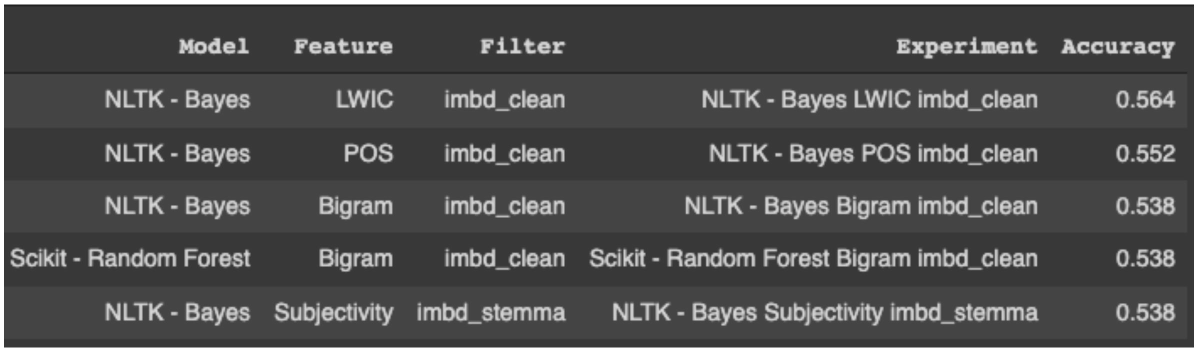


Regardless, drilling down into the top 5 experiments which includes the following:

1. Bayes LWIC using the clean feature
2. Bayes Unigram Clean feature
3. Bayes Part of Speech clean feature
4. Scikit Random Forest Bigram Clean feature
5. Bayes Subjectivity clean data feature

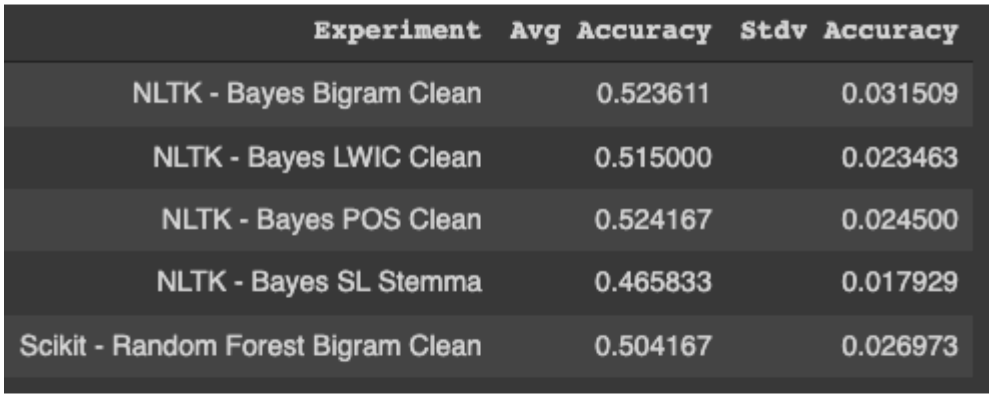
The top 5 ranged from 56% to 52% from an accuracy score. With Scikit Random forest as the only non NLTK model.





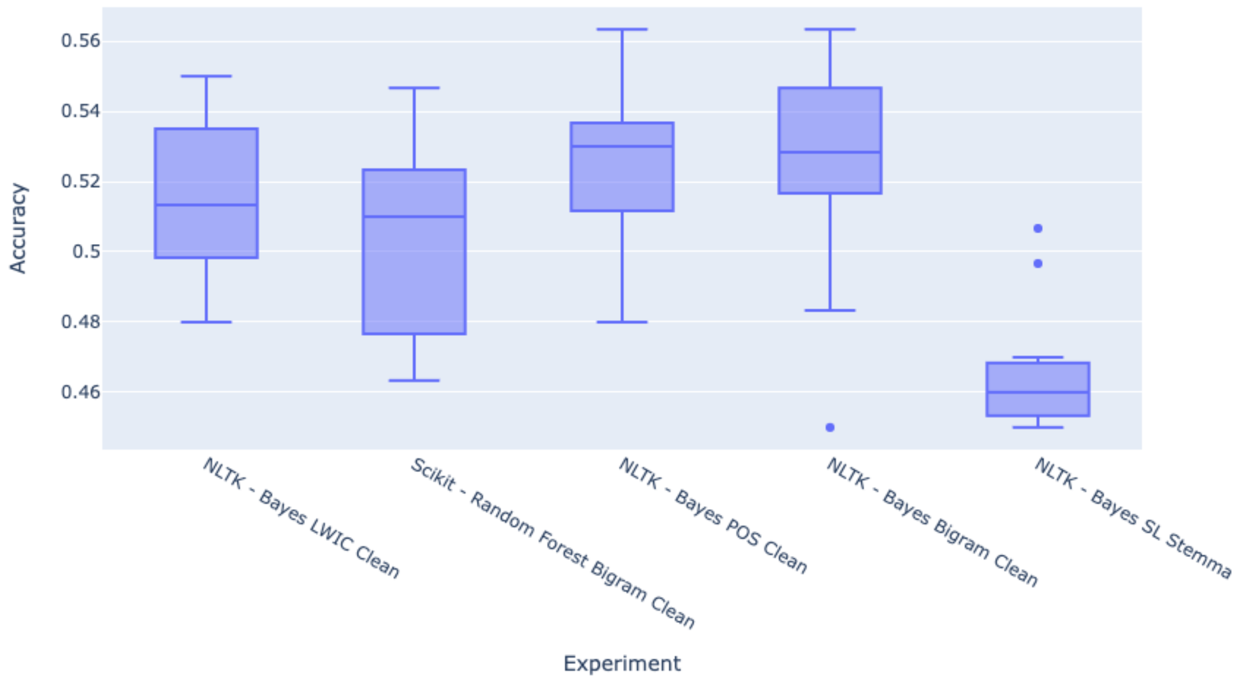
Model	Feature	Filter	Experiment	Accuracy
NLTK - Bayes	LWIC	imbd_clean	NLTK - Bayes LWIC imbd_clean	0.564
NLTK - Bayes	POS	imbd_clean	NLTK - Bayes POS imbd_clean	0.552
NLTK - Bayes	Bigram	imbd_clean	NLTK - Bayes Bigram imbd_clean	0.538
Scikit - Random Forest	Bigram	imbd_clean	Scikit - Random Forest Bigram imbd_clean	0.538
NLTK - Bayes	Subjectivity	imbd_stemma	NLTK - Bayes Subjectivity imbd_stemma	0.538

Taking the top 5 results and running them each through cross validation but with 3,600 sentences and 12 folds instead of the original 1000 and 3 folds shown above.



Experiment	Avg Accuracy	Stdv Accuracy
NLTK - Bayes Bigram Clean	0.523611	0.031509
NLTK - Bayes LWIC Clean	0.515000	0.023463
NLTK - Bayes POS Clean	0.524167	0.024500
NLTK - Bayes SL Stemma	0.465833	0.017929
Scikit - Random Forest Bigram Clean	0.504167	0.026973

The results from the cross validation shows that the average Accuracy is different then what we saw with the test and train data set. Overall the results are about 4% less than before. The top Accuracy is now the NLTK Bayes Part of Speech with 52%. When looking at the standard deviation, 2.4% is much better than the second best accuracy score, Bayes Bigram Clean. Meaning that POS with 52.4% and a standard deviation of 2.5% is the most optimal from the list above.



Looking at the 12 folds of the cross validation results above Bayes POS Clean has a much higher average and the best median value than the rest. What is interesting is that the Bayes Subjectivity Lexicon using Stigmatizing and Lematizing has the tightest box plot of the bunch but the worst in Accuracy scores.

Final Thoughts

The path to predicting a rating based on comments was long and arduous. Through filtering, transforming the data, applying multiple feature engineering techniques, and then modeling the featuresets, a conclusion can be seen. The power to predict comes at a great price.   Throughout the process we only achieved an accuracy score of around 55%. Though the journey was hard, the exercises learned were invaluable. From the numerous cleaning and transforming techniques to ingest into a model. To the excitement of seeing results. Then to use methods to help prove the results, by cross validating the predictions. Although only 3 models were explored, additional models can be applied, and more data could be used. At what cost depends on the machine and time necessary to continue striving for a higher accuracy score. In summary the techniques in this paper illustrate what is needed to succeed in modeling text data. In addition, it serves as a terrific blueprint to begin applying Natural Language Processing in the real world.