Project Title: Improving Accuracy of Natural Language Processing in Yelp Reviews
Final Project Report for CS 175, Winter 2016
List of Team Members:
Efren Aguilar, 76323142, efrena@uci.edu
Monami Yang, 44194004, monamiy@uci.edu
Peter Truong, 64983222, pgtruong@uci.edu

1. Introduction and Problem Statement

In this project, we hope to improve review analyses of the Yelp dataset to overcome many of the setbacks from currently used algorithms. Our main goal is to take the Yelp dataset and find ways to accurately classify whether a review is positive or negative. Once we extract the raw data and convert it to usable training and testing datasets, we want to see if there are any ways we can modify the data points selected to help improve the accuracy on text classification. We classify text using sentiment analysis in which we see how words affect the positivity or negativity of a review. In our tests we use Multinomial Naive Bayes, Linear Regression, and Support Vector Machines as the algorithms for our classifiers. After figuring out the baseline accuracies we start with, we began our research into what different things we could do to improve on these baseline accuracies.

After many different tests and analyses we were able to come up with a few ways to improve the accuracy of all our classifiers on our dataset. By far, the biggest way we found to improve the accuracies was to remove reviews that were either very long or very short from our dataset, doing this improved our classifiers by around 10%. However, we also found that limiting the bag of words used for training to only include the most positive and negative words and including bigrams in our analysis led to improvements in classifications, although these improvements were less noticeable. Overall, we are happy to say that we ultimately succeeded in our end goal and that hopefully our findings may lead to an improved accuracy for the actual star ratings of reviews.

2. Related Work:

For our project, we've referenced the research paper, "Predicting Yelp Star Ratings Based on Text Analysis of User Reviews", from Yun Xu, Xinhui Wu, and QinXia Wang[1]. Similar to their approach, we've utilized the naive bayes algorithm for finding precision and recall and we split the texts into negative and positive reviews based from their star ratings. Other similarities between our projects for improving the overall accuracies include part of speech tagging to help in bigram collocations and splitting 70% of the data to be used as a training set and the remaining 30% for the testing set in cross validation. However, we find the vocabularies for our classifiers differently. Where we've used nltk libraries to create a bag of words, they've used a

[1] Xu, Yun, Xinhui Wu, and Qinxia Wang. *Sentiment Analysis of Yelp's Ratings Based on Text Reviews*. *Stanford.edu*. N.p., n.d. Web. <http://cs229.stanford.edu/proj2014/Yun%20Xu,%20Xinhui%20Wu,%20Qinxia%20Wang,%20Sentiment%20Analysis%20of%20Yelp's%20Ratings%20Based%20on%20Text%20Reviews.pdf>.

perceptron algorithm by considering the review as a set of sentences, not as a single unit of text. By using such an algorithm, they were able to use other methods in their classification such as stochastic gradient descent, Multi-class Support Vector Machines, and Nearest Neighbor methods from scikit-learn. From the comparisons of different algorithms, the group concluded that the Naive bayes classification method had the best overall accuracy compared to the Multi-class Support Vector Machine, Perceptron, and Nearest Neighbor methods for both precision and recall, which is one of the reasons why we made sure to include Naive Bayes in our classifications as well.

3. Data Sets

For this project use the textual content from the 2.2 million reviews provided by the Yelp dataset (https://www.yelp.com/dataset_challenge). We then extract the review information from Yelp's large dataset and use various amounts of reviews for classification. After we take the data, we split it into two sets: 75% being a training set and 25% being a testing set. We also create a bag of words, weighing certain words to be more impactful on the classification.

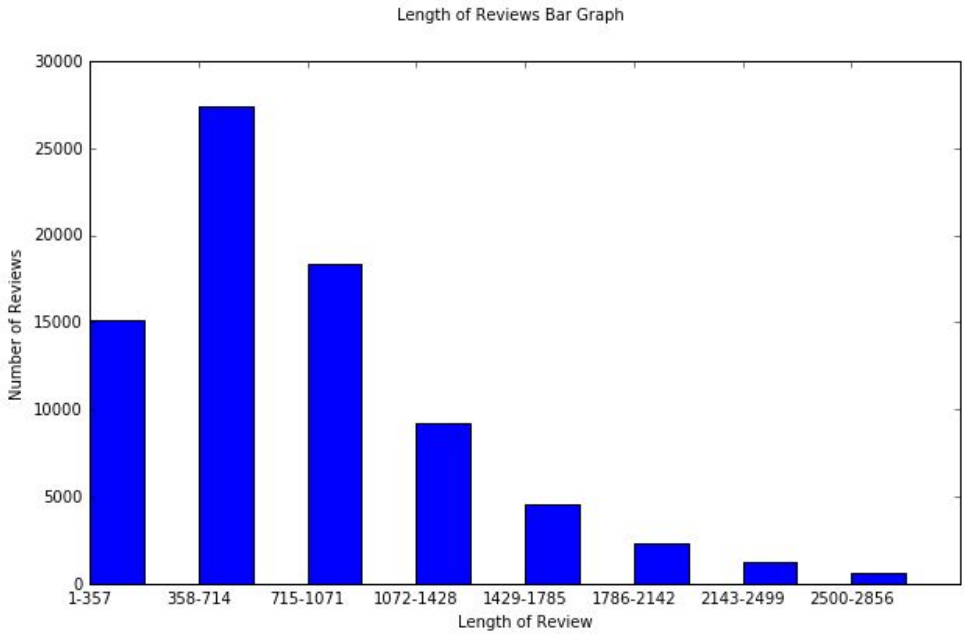

Figure 1

We made the graph in figure 1 using 40,000 reviews to see the average length of a review in order to determine and implement a cutoff for when reviews are too difficult to analyze for both short and long reviews. This would allow us to make our classifier more accurate and look at only reviews with a good length to analyze.

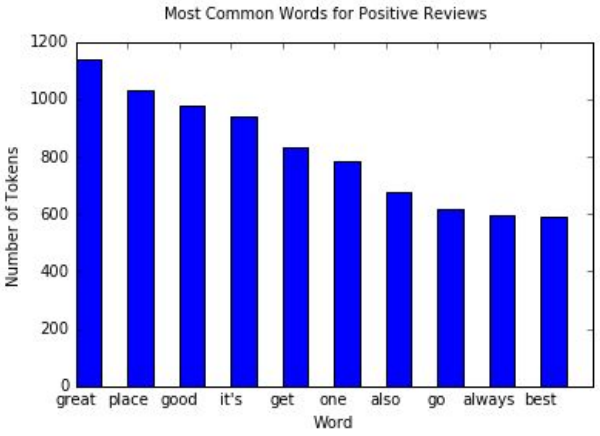

Figure 2

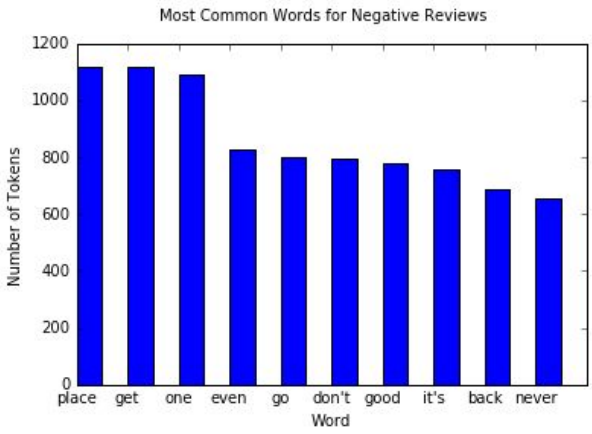

Figure 3

We extracted a dataset of 14,000 reviews and checked the 10 most common words using a frequency distribution. We did this to see what words we could eliminate and classify as stopwords to speed up our program. Figures 2 and 3 show this analysis.

Our program does not rely on any lexicon to weigh the words, rather we implement our own algorithm to take words within the reviews and find if they appear more in positive or negative reviews of the training set, thus training the classifier to weigh words that appear more often in positive or negative reviews. At one point, we did try to use the lexicon provided here (https://www.cs.uic.edu/~liub/FBS/sentiment-analysis.html#lexicon) to make an easier weight algorithm to find ratios of words based on the words within the lexicon. However, we found that datasets are easier to classify with our original algorithm, than with using this lexicon.

4. Description of Technical Approach

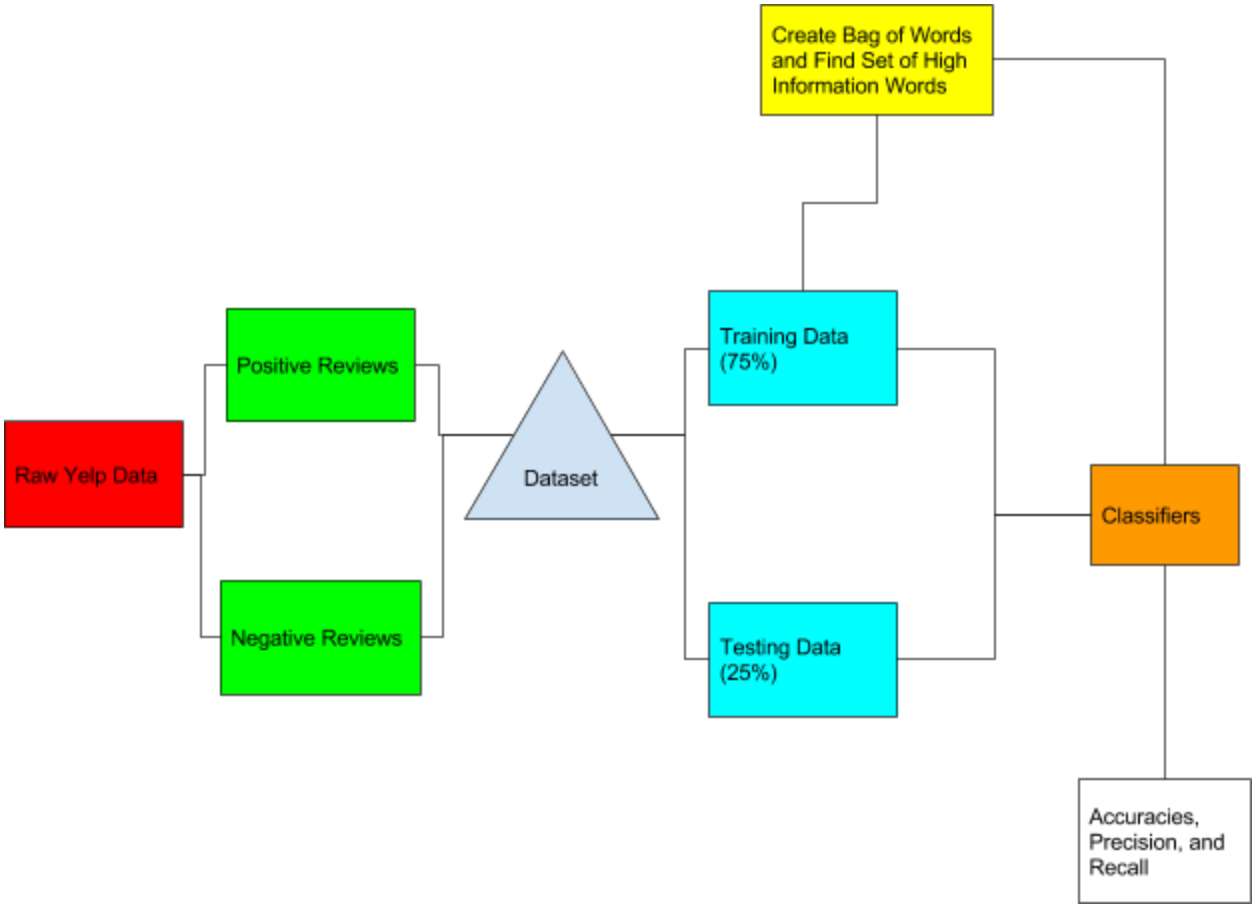

Figure 3

We first take the raw yelp data and split it based on the star rating of the reviews. Then, we create our own dataset with x reviews (x being how many reviews we want) and we split the reviews to 75% being training data and 25% being testing data. We then create our bag of words and find most informative words using our own algorithm along with some help from nltk to find bigrams. The algorithm we created uses nltk's bigram collocation finder, FreqDist, and Conditional FreqDist to find the most informative words and bigrams to help with weighing words for our classifiers. We improve our bag of words using these high information words and bigrams to help improve accuracies on our classifiers.

Next, we use nltk's scikit wrapper to use our classifiers: Multinomial Naive Bayes, Logistic Regression, and Support Vector Machine. The classifiers train on the training data and then test using the testing data. Afterwards, the classifiers will return the accuracies, then calculate precision and recall using nltk's libraries that include the calculations for accuracy, precision, and recall. We then are able to compare these classifiers to see which did the best. At the same time, the dataset also calculates 3-fold cross-validation of the bag of words. As it loops 3 times, the train set and the validation set will be splitted from the current number of fold, which is one

at first, then train the train set in the naive bayes classifier. Afterward, it returns three accuracies from the three validation set.

In terms of what we tried to improve on, logistic regression did the best with our improvements such as our cutoff points for review lengths and our bigram information. On the other hand, the original simple bag of words put into the classifiers showed that support vector machine did the best purely because they use their own algorithms for calculations and adding certain features can sometimes drop the support vector machine's accuracy.

4. Software

a.) Code and scripts written by us

- Loading json files and appending set amounts of data to a data list
- Tokenization of simple texts along with replacing punctuation
- Removing stopwords from the tokens
- Multiple bag of word functions to analyze accuracies on our classifier
- Splitting the data into positive and negative reviews by checking the reviews' score is greater than 4 stars or less than 2 stars and also has a cap to give a max length for the list of data.
- Classifying recall and precision for knowing the percentage of Type I error (false positive) and Type II error (false negative) in the data
- Getting frequency distribution from yelp_dataset bag of words
- Taking the dataset list and plots the review by its word lengths
- Plotting the top 20 most common words for positive and negative
- Removing the most infrequent word from the review
- Implementation of bi-grams for vocabularies
- Creating positive and negative ratios for weighting the reviews
- Creating word features dictionary of both negative and positive data
- Calculating 3-fold cross validation using the word features

b.) Outside code and scripts used in our project

- yelp_academic_dataset_reviews json files
- Nltk stopwords, Frequency Distribution, POS(part of speech) tagging, bigram collector
- Nltk classifiers and scikit wrapper for classifiers
- matplotlib.pyplot
- numpy
- sklearn Multinomial Naive Bayes, Bernoulli Naive Bayes, Logistic Regression
- CountVectorizer, TfidfTransformer

- math library
- Positive-words.txt and negative-words.txt for sentiment analysis

5. Experiments and Evaluation

For our initial experimentation, we used scikit's built in classifiers, such as Multinomial Naive Bayes, Bernoulli Naive Bayes, and Logistic Regression, on a tfidf document matrix to get some starting accuracies as a base to improve on. This was done using an array of data, being 0's and 1's for negative and positive reviews. Once this was done, we decided the best starting point was to analyze the yelp data itself. We created three graphical analyses that ran on the training data set for our classifiers. One was the most common words for positive reviews and their frequencies, another was the same thing but for negative words, the third was a graph showing the distribution of review length for all of the reviews.

An interesting thing that we noticed from the length graph, shown in Figure 1, was that the majority of the reviews seemed to fall with a certain range length. Once we observed this, we decided to try and find a good estimate for this range and see if by removing all the reviews that did not have a length within this range would improve our accuracy at all. After doing so we found that it actually led to a massive improvement in accuracy! Our classifiers went up an average of around 10%! After this we began to take a look at the other two graphs we had previously generated to see if they could lead to further improvements for our classifiers' accuracies (Figures 2 and 3).

The outputted graph for positive words seemed to be on the right track; however, the negative graph had many words that weren't very negative at all. Taking this into account, our next idea was to find ways to improve the vocabulary used to train our classifiers to give a better interpretation on what exactly classifies as a positive or negative word. To do this we tried two things, using a pre-made lexicon of positive and negative words, giving these words more weight in determining the sentiment of a review, and implementing bigrams into our vocabulary.

We decided to try some online lexicons to see if our accuracy would improve instead of just using a bag of words and trying the classifier. We used the lexicons stated earlier to have positive and negative words already labelled. We started by having positive words count for a score of one and negative words count for a score of one. We noticed a decrease in accuracy and noted that we needed to weigh higher information words to be weighed higher. Although we attempted to, we noticed a decrease in our accuracy and crossed it off for later.

We originally tried to implement bigrams ourselves using part of speech tagging. First, we would tag all the words, then we would use these tags to get bigrams (for things like adjectives and adverbs). Our code worked, and we got great bigrams from our functions (as shown in FIgure 4), but our analysis was way too slow for the size of our dataset. For our code to work, we had to do a part of speech analysis on all of the review text, then we had to iterate over all these

tagged words to find bigrams. Once we tested our code, it took way too long to run and we could only do it on very small datasets. So, we turned to find other ways of getting bigrams.

```
Positive bigram:
[('make sure', 76), ('one best', 61), ('really good', 43), ("it's great", 42), ('highly
recommend', 36), ('pretty good', 35), ('pretty much', 34), ('really good.', 34), ('really
nice', 32), ('one favorite', 32), ('well worth', 28), ('always good', 24), ('definitely
worth', 22), ('always great', 20), ("it's nice", 19), ('staff friendly', 19), ("it's good",
18), ('go wrong', 18), ('great little', 17), ('really great', 17)]
```

Figure 4: Good bigrams, but way too slow

After some research, we found that nltk has various functions for analyzing bigram collocations along with a scikit wrapper to use the scikit classifiers. We decided to use nltk's library for these classifiers and the bigram collocation finder. We started with a simple bag of words with each word given a boolean to state whether they are in the review or not, essentially a very basic document matrix. We used three classifiers to test our data on: Multinomial Naive Bayes, Logistic Regression, and Support Vector Machine. It is important to note that we used two different algorithms for Support Vector Machine (Linear vs Nu) to see how the accuracies would be affected. Afterwards, we tested our classifiers with the bag of words and decided to implement a weight function. Our weight function, called high_words, took 10000 of the most informative words and only accounted for those words in our newly created bag of words. We found great improvements in our accuracy. We went from an average of 66%-70% percent accuracy using from where we started to a whopping 90% with these features and the use of better classifiers.

Description	Multinomial Naïve Bayes	Logistic Regression	Linear Support Vector Machine	Nu Support Vector Machine
Bag of Words	0.8752, 0.8432, 0.8791	0.9148, 0.9134, 0.9161	0.8932, 0.8908, 0.8954	0.8928, 0.8907, 0.8947
Added High Information + Cutoff	0.8964, 0.8792, 0.9136	0.9196, 0.9048, 0.9207	0.9048, 0.9031, 0.9063	0.894, 0.872, 0.916
Added Bigrams	0.8896, 0.8851, 0.8936	0.916, 0.9151, 0.9167	0.9076, 0.9061, 0.9090	0.902, 0.9018, 0.9021
TRAINED ON 10000 REVIEWS				
7500 training, 2500 testing				
Data sorted by:				
accuracy, pos f-measure, neg f-measure				

Figure 5: One example of our analyses with a dataset. Of course, larger datasets yield different results and these results do not justify the classifiers accuracies as a whole.

With this is mind, we wanted to find a way to increase the accuracies even more. We then used the bigram code from nltk to implement a bag of words function that also included bigrams and noticed a small increase (around 1% increases) to our classifiers. We were happy with our results in the end and calculated our precision, recall, and f-measure to check and make sure our classifiers were running correctly with a good balance of type I and II errors.

We also want to note that experimenting with larger datasets took significantly more time but yielded greater accuracies. However, when we started our experimentation we learned some errors in our algorithms that caused our program to run slower. Because scikit was a little difficult for us as well as our old weight function and our bigram collector, we noticed a huge impact on our speed and decided to use nltk's library more to help us with processing time. We are happy with the speed it runs at considering the large datasets we use.

6. Discussion and Conclusion

In this project, we learned that sentiment analysis is extremely complex and difficult given what we know with artificial intelligence. The course setup allowed us to gain insight on how to start our project and we feel that the way the course was structured helped tremendously in our progress. We learned about the most simple algorithms like tokenizing and creating a bag of words or finding bigrams to some more complex ones such as the calculation for precision and recall. Surprisingly, we also learned that something as simple as creating a length cutoff range can also make as big of an impact as many of these complex algorithms. The biggest limitation we found in this project was our lack of knowledge in many of the different features of nltk and scikit. If we had a more in depth understanding of how they work, we may have been able to tweak the algorithms for further improvements. If we were in charge of a research lab, this is exactly what we would do, try to find ways to tweak algorithms to make improvements on things like bigram collocation and the accuracies of individual classifiers. After this, we would also try and improve our classifier to estimate the actual star rating of a review, rather than just if it is of positive or negative sentiment.