The first part of this paper will summarize the work I have done on building a classifier for IMDB Review’s by predicting the expected score of each IMDB review. The data was retrieved from Kaggle as listed in our project description (<https://www.kaggle.com/iarunava/imdb-movie-reviews-dataset/version/1>) and all accuracy figures for my classifier will be based on training on the 25,000 training examples and classifying on the 25,000 test examples. The dataset for the classifier (supervised learning) contains 50,000 reviews which are split evenly into 25,000 positive and 25,000 negative reviews. Positive reviews are classified as having a score that is greater than or equal to 7 out of 10 and negative reviews are classified as having a score of less than or equal to 4 out of 10. It is also important to note that reviews with more neutral ratings are not included in the supervised learning dataset.

The second part of this paper will briefly cover what I have done with unsupervised learning to mine patterns in the data.

**Loading the Data**

The first task of my classifier was to load the data. To do this I iterated through each directory with my python script and loaded the positive and negative data into a 2D array. Inside the 2D array, the first column has the ID, the second has the expected rating, and the third column has the text data (review). While fetching the review data, I also cleaned up the review text as it contained some html code in it (I removed “<br />). I also implemented a load and save feature for this as opening each file creates more overhead and takes longer for the script, so once all data is loaded into an array, I use the *pickle* library of python to save all the data into a file to use as a cache for the next steps, significantly reducing testing time.

**Pre-Processing**

The next step was then to convert this array into a *pandas* dataframe which would allow for more data manipulation. Thus, I imported the *pandas* library and created columns containing the “id”, “rating”, and “data”. I then loaded the python library *nltk* (a natural language processing library) which would allow me to tokenize my “data” column. By tokenizing the data column, I am able to then create a vector of all the words in the review and count how many times each word appears in the review. It is also important to note that the amount of vocabulary that is selected can also be limited in the count vectorizer (i.e. I can limit vocabulary saved to be 3,000 words instead of every word in all the reviews.)

Another approach I tried for pre-processing the data was to also clean the tokens of the words before applying any machine learning algorithm on them. I first lowered all the tokens to lowercase and removed all non-alphabetical characters. I then removed all stopwords from the array/vector of tokenized words. Stopwords can essentially be classified as extremely common words that do not play a vital meaning to the sentence, this can include “The”, “Is”, “at”, “on” etc. and in my case, the stopwords come from the *nltk* library’s stopwords set. After removing stopwords, I also stemmed the words by using a SnowballStemmer which according to the nltk website, removes “morphological affixes from words, leaving only the word stem” (nltk.org). Or in more simple terms, it removes tenses or endings of words to only leave the stem (i.e. *flying* becomes *fly*). This is important because it helps the classifier models to spot similar words that it previously would not have identified as the same.

After attaining the counts of each word in the vector (tokenized review), I then implement a TF-IDF (Term Frequency-Inverse Document Frequency) transformer which basically takes the term frequency and accounts for the frequency in comparison to how many total words there were in the review. This ensures that some counts/values of words will not be skewed as a result of a review being longer and mentioning the word multiples times due to an extra-long length.

My next step after applying all this processing was to create my target values which would be the ratings, of which I simply extracted it out of the “rating” column of the dataframe I had created earlier. I then used scikit learn’s *train\_test\_split* feature to split the dataset into 70% training data and 30% validation/testing data. I allowed a *random\_state* equal to 5 in order to ensure that the data would not be in the order I had loaded it in (reducing overfitting for weird patterns).

**Applying ML Models**

The models that I applied to this dataset are the Naïve Bayes Multinomial NB, Logistic Regression, Linear Passive-Aggressive-Classifier, Linear Perceptron, and an SGD Classifier all from the sci-kit learn library. The Naïve\_Bayes MultinomialNB algorithm should be the best theoretically to classify the text data that I passed it. This is due to the fact that it is great for classification with discrete features such as word counts (provided by the pre-processing earlier and tf-idf). When I initially tested the results, I got an accuracy of 33.04%, this low accuracy was mainly due to the fact that I was classifying for data for scores of 1 – 10. Seeing as this was not the most optimal way of looking at the problem, I then concluded it would make more sense to classify data as positive and negative reviews, as that was how the data was classified when given to us. Thus, using the guidelines given to us by Kaggle, all predictions above or equal to 7 were considered positive reviews, and all predictions below or equal to 4 were classified as negative reviews. I have put the results of the Naïve Bayes Multinomial NB in a table below with varying parameters and approaches that I have tried:

|  |  |  |  |
| --- | --- | --- | --- |
| Naïve Bayes Multinomial NB | | | |
| Score | Word Limit | SnowballStemmer | Stopwords |
| 77.92% | no | no | no |
| 84.18% | yes | no | no |
| 83.46% | no | yes | yes |
| 80.10% | yes | yes | yes |
| 80.92% | no | no | yes |
| 84.29% | yes | no | yes |
| 79.63% | no | yes | no |
| 84.29% | yes | yes | no |

As can be seen, the highest accuracy was given (84.288%) when the total number of words to be vectorized was limited to 3,000 (I had tried with 1,000, 2,000 and 4,000 but they all scored lower than 3,000 and the gains after 3,000 were minimal for Naïve Bayes, For logistic regression 3,000 scored the most so I decided on using 3,000 as my standard for word limitations) and either stopwords were removed or stemming was applied. The general trend in this data set would be an improvement in the score once a word limit was put in place.

My next approach was to apply logistic regression to the data. Surprisingly, this approach performed substantially better than Naïve Bayes. In order for the regression model to work, I had to round values to the closest whole number, or in this case, if it was greater than 7 I would classify it as a positive review and if it was less than 4 I would classify it as a negative review.

|  |  |  |  |
| --- | --- | --- | --- |
| Logistic Regression | | | |
| Score | Word Limit | SnowballStemmer | Stopwords |
| 87.43% | no | no | no |
| 87.00% | yes | no | no |
| 86.25% | no | yes | yes |
| 86.42% | yes | yes | yes |
| 86.98% | no | no | yes |
| 86.56% | yes | no | yes |
| 80.92% | no | yes | no |
| 86.56% | yes | yes | no |

My results showed that without pre-processing the data, the algorithm performed better with just tokenized words before pre-processing. In this set of data, results were mixed when a word limit was put in place, with half of the tests improving and the other half having a detrimental affect.

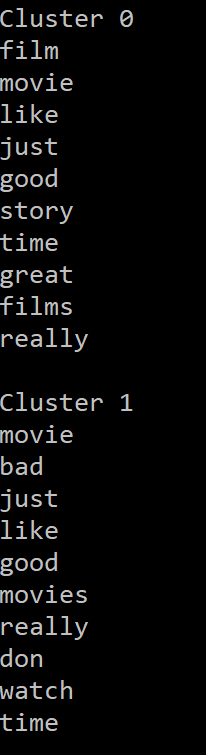
Next, I tried to us a Stochastic Gradient Descent Classifier, a Passive Aggressive Classifier, and a Linear Perceptron Classifier all from the scikit modules. The results are as shown below:

|  |  |  |  |
| --- | --- | --- | --- |
| Stochastic Gradient Descent Classifier | | | |
| Score | Word Limit | SnowballStemmer | Stopwords |
| 85.40% | no | no | no |
| 84.28% | yes | no | no |
| 83.44% | no | yes | yes |
| 83.68% | yes | yes | yes |
| 84.13% | no | no | yes |
| 83.52% | yes | no | yes |
| 86.98% | no | yes | no |
| 83.45% | yes | yes | no |
|  |  |  |  |
| Passive Aggressive Classifier | | | |
| Score | Word Limit | SnowballStemmer | Stopwords |
| 81.53% | no | no | no |
| 81.38% | yes | no | no |
| 80.63% | no | yes | yes |
| 78.45% | yes | yes | yes |
| 80.40% | no | no | yes |
| 80.73% | yes | no | yes |
| 80.33% | no | yes | no |
| 80.46% | yes | yes | no |
|  |  |  |  |
| Linear Perceptron | | | |
| Score | Word Limit | SnowballStemmer | Stopwords |
| 81.10% | no | no | no |
| 79.94% | yes | no | no |
| 79.70% | no | yes | yes |
| 78.21% | yes | yes | yes |
| 79.42% | no | no | yes |
| 78.74% | yes | no | yes |
| 79.42% | no | yes | no |
| 78.74% | yes | yes | no |

As can be seen in the data, the Linear Perceptron scored noticeably Lower than the other Classifiers, while Stochastic Gradient Descent Scored the best. The general trend for these models was a decrease in performance when a Word Limit was put in place. A reason for this could be how the Linear Models in Scikit were made, as all of the above models I tested aside from the Naïve Bayes model took a negative impact when a word Limit was put in place. The highest scores for all of these models were also when the least amount of pre-processing was applied (only tokenization). Thus, I believe that the overall trend with using linear models for this type of data would be best if data was given without any form of pre-processing and a word cap was not put in place for the tokenizer.

**Unsupervised Learning**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Cluster 1 | Cluster 2 | Cluster 3 | Cluster 4 | Cluster 5 | Cluster 6 | Cluster 7 | Cluster 8 | Cluster 9 | Cluster 10 |
| Action | Book | Bad | Disney | Horror | Funny | Series | Women | Sci | Game |
| Movie | Read | Movie | Flynn | Movie | Comedy | Episode | Gay | Fi | Games |
| Film | Movie | Acting | Bambi | Film | Movie | Episodes | Men | Movie | Graphics |
| Good | Film | Worst | Animation | Gore | Jokes | Season | Film | Film | Play |

For unsupervised learning, I applied tokenization for my pre-processing then removed stopwords from the scikit library of stopwords and then applied TF-IDF Vectorizer which returns the importance of a word based on how many time it appears in the sentence. Once this was done, I put the data through a KMeans cluster with K = 2. Meaning that I would only have 2 clusters. This was the most reasonable cluster to me as I was trying to classify reviews based on being positive or negative reviews of the movie. The top ten words of each cluster can be seen here to the right. As you can see, Cluster 0 is what the model classified as a positive review, while Cluster 1 is what the model classified as a negative review. As I increased the clusters (K=20), each cluster began to lean towards genres with 1 cluster being representative of negative reviews. Here is a table with 10 of the following 20 clusters and their top 4 words in each cluster:

As can be seen, these clusters generally lean towards a type of movie rather than the types of reviews (except for Cluster 3) and a clear trend in the data can be seen by using the K means algorithm. I had also tried to use Agglomerative clustering but had failed to do so as the data was considered to “sparse” for the model itself.

**Conclusion**

Overall, I learned various techniques from this project for text classification/prediction. A lot of what I had learned involved pre-processing data and the majority of my difficulties involved finding models that would work with the data that I was giving it. This is due to the fact that many of the models I had tried for supervised learning (such as Multilayer Perceptron Classifier, Lasso Regression etc.) ran out of memory or wanted “dense” data, and various models I had tried for unsupervised leraning such as Feature Agglomeration required “dense” data (when I tried to convert to dense data, my computer would run out of memory). Thus, I also learned a good deal about the limits of computing on limited resources from this assignment. Each test case would take multiple minutes to run too due to the sheer amount of text data that was being processed.