Personality Recognition – Neuroticism Classifier

By Tejasri Pavuluri & Faizah Tariq Pirzada

1. Background

The neuroticism classifier aim is to build a classifier that classifies a users' social media posts as neurotic, or not, using logistic regression and also boosts the performance of the classifier using different techniques. The report and the problem was based from the article "Workshop on Computational Personality Recognition: Shared Task" which discusses personality traits values and social media statuses from well-known Big 5 personality traits (also known as OCEAN for the 5 traits it defines: Openness, Conscientiousness, Extraversion, Agreeableness and Neuroticism). The article discussed two datasets (Essays and MyPersonality) with gold standard labeled user information used for personality recognition by 8 different groups. The goal of the workshop was for each group to go about finding personality recognition solutions on the same datasets in a way they saw most fit. This idea originated from the observation that much of the research being done in personality recognition was being done with varying resources and techniques that did not permit for an adequate comparison between colleagues. Thus, at the conclusion of the workshop, the groups would present their work with the performance increase or decrease that was obtained, and they would serve as a benchmark from which future projects in the similar field could compare themselves to.

Like the workshop, our classifier also utilizes the same gold standard labeled datasets. The two datasets: *Essay* and *myPersonality* were provided and will be used in this project to develop a model to predict 'neuroticism'. *Essay* is a large dataset of about 2400 texts produced by students who took the Big5 test collected between 1997 and 2004 with personality classes (nominal classes) converted from z-scores. *myPersonality* contains information about 9900 status updates(raw text) from individuals on social media, gold standard personality labels(both as scores and classes) and several social network measures, including: network size, betweenness centrality, density, brokerage and transitivity.

2. Explore Data/Initial Data Analysis

2.1. Jsonlines from csv file

After loading the myPersonality.csv data produced JSON lines which is the appealing form format for data streaming. JSON Lines is a convenient format for storing structured data that may be processed one record at a time.

2.2. Explore Data

Understanding the characteristics of the data beforehand will enable us to build a better model. This could simply mean obtaining a higher accuracy. It could also mean requiring less data for training, or fewer computational resources.

To run some checks on the data, we picked a few samples and manually checked if they are consistent with the expectations. For example, we printed a few random samples to see if the sentiment label (neurotic) corresponds to the sentiment of the user statuses. Here is a review we picked at random from the myPersonality dataset: “is tired and for some reason is looking forward to classes starting...” The expected sentiment (not neurotic) matches the sample’s label.

> “is celebrating her new haircut by listening to swinger music and generally looking like a doofus.”
> The expected sentiment (neurotic) matches the sample’s label.

2.3. Collect Key Metrics

Once we verified the data, we collected the following important metrics that can help characterize the text classification problem:

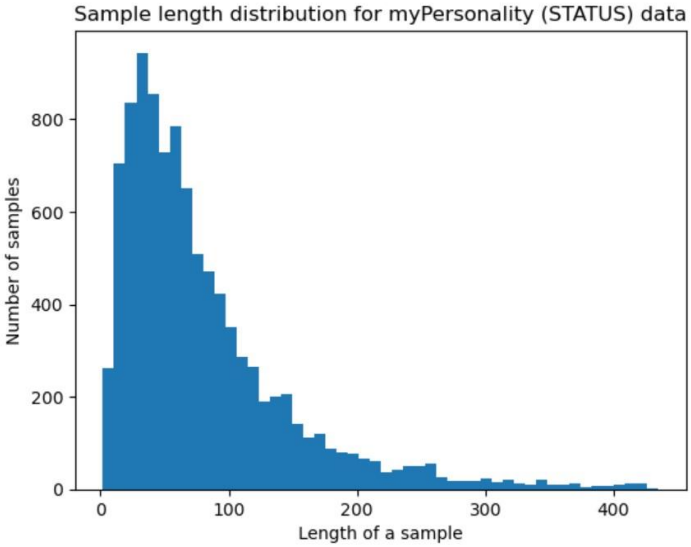
1. **Number of samples**: Total number of examples you have in the data.
2. **Number of classes**: Total number of topics or categories in the data.
3. **Number of samples per class**: Number of samples per class (topic/category). In a balanced dataset, all classes will have a similar number of samples; in an imbalanced dataset, the number of samples in each class will vary widely.
4. **Number of words per sample**: Median number of words in one sample.
5. **Frequency distribution of words**: Distribution showing the frequency (number of occurrences) of each word in the dataset.
6. **Distribution of sample length**: Distribution showing the number of words per sample in the dataset.

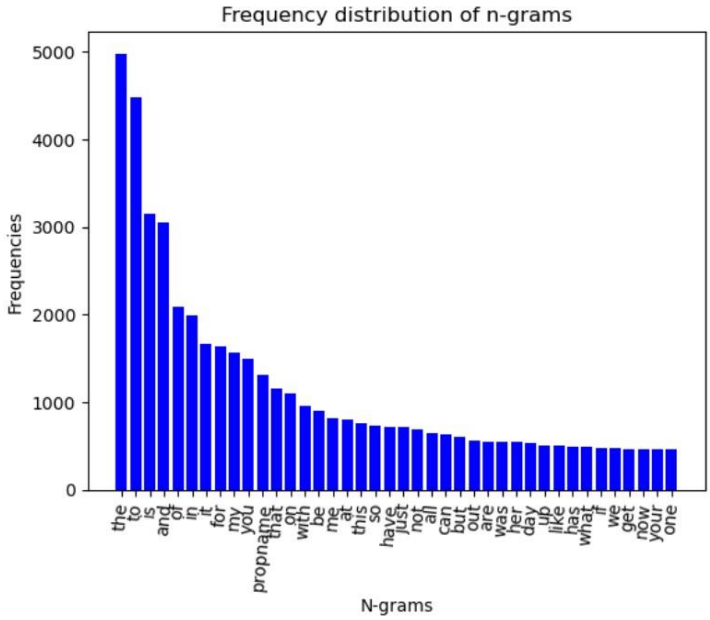
Below are the values of the metrics for myPersonality user dataset

Metric name	Metric value
Number of samples	9916
Number of classes	5
Number of samples per class	In below table
Number of words per sample	11

Number of samples per class				
cEXT	cNEU	cAGR	cCON	cOPN
0	2238	4811	2180	688

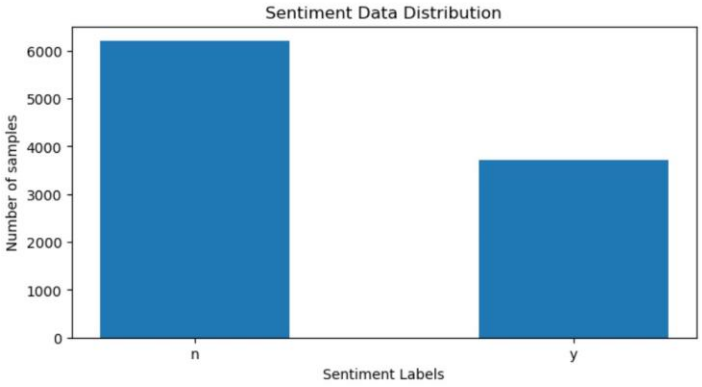
Below are the plots of the word-frequency and sample-length distribution





Since this project is focussed on sentiment classification of neurotic or non-neurotic sentiment analysis, a distribution of these labels would give us a better understanding (y - neurotic, n - non neurotic). As shown in the distribution figure

below, we can see that non-neurotic labels are higher than the neurotic which can conclude that the dataset is quite imbalanced.



3. Baseline

3.1. Train-Test Split Baseline model

After we gain insights into the key characteristics of the data, the next step is to think about the classification model to be used. Since the main requirement for this project was to develop a **baseline** model for the prediction of neuroticism using different tools and strategies, we chose the baseline model, a logistic regression model, which is a type of n-gram model that sees text as "bags" (sets) of words. Another main reason was to utilize our code from previous class assignments where a logistic regression model was used to predict the political parties speeches, and the speeches were encoded into Bag of Words representing the Unigrams and Bigrams, and the stopwords were filtered.

Before our data can be fed into the model, it needs to be transformed to a format the model can understand. Specifically, the user statuses are collected and encoded into Bag of Words representation (a sparse matrix) of unigrams and bigrams that the logistic regression model can process. There is an additional step in the process of encoding where a predetermined stop word list is utilized to filter the common words that hold little, to no, meaning in the prediction. The two pre-processing steps in the process of converting text into numerical vectors are tokenization and vectorization. CountVectorizer tokenizer is used to tokenize the text by removing the punctuation marks and converting all the words to lowercase. Stopwords list is also passed as an argument to count vectorizer.

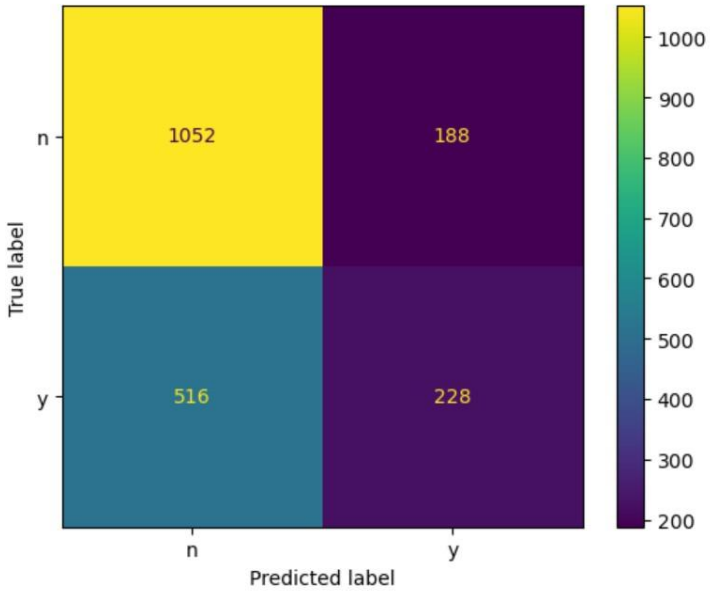
Some of the features(tokens) from the vector of tokens are discarded to calculate the important features (own features), by dropping certain tokens that occurred extremely rarely across the dataset using stopwords. The features extracted included non-stopwords,

non-punctuation unigrams, and bigrams features that were collected. Combining these methods the dataset is split into 80-20 train and test sets, and the vectorizer feature string functions were applied on the training set to create the training feature vector. The following results were obtained from the logistic regression classifier after creating the test set feature vectors.

```
Accuracy  = 0.6451612903225806
Classification report for baseline:
              precision    recall  f1-score   support

           n       0.85      0.67      0.75      1568
           y       0.31      0.55      0.39       416

    accuracy                           0.65      1984
   macro avg       0.58      0.61      0.57      1984
weighted avg       0.73      0.65      0.67      1984
```



To ensure the model is not affected by the data order and the significant imbalance of labels, we applied simple best practices like **shuffle=True and stratified**. To evaluate the performance on the testing set, we used the "classification_report" and "confusion_matrix" functions from scikit-learn.

3.2. K-Fold Stratified Baseline model

We understand there is a label imbalance in the classes(neurotic or not-neurotic) values and implemented 'StratifiedKFold' method on X_features and y labels meaning and implemented the above same sparse matrix with stopwords, tokenization and vectorization on the entire statues messages and split the data into 5 folds. By stratifying the folds, we can ensure that the class proportions are similar between train and test sets, which helps to avoid overfitting.

The ‘cross_val_score()’ method was used to evaluate the score by cross-validation. The accuracy of the logistic regression model was estimated on the data by splitting the data, fitting the model and computing 5 scores consecutively. The performance measure is the average of the 5-folds computed in the loop. The mean score using this method is shown below

```
Scores for each fold are: [0.63508065 0.63961694 0.63287948 0.63489662 0.65658094]
Average score: 0.64
```

We can see that the both baseline models (Train-test split model & Stratified K-Fold model) resulted in the same accuracy of 64%.

4. Improved Model Classification

For the next step to improve the performance of the base model, we set up to maintain the logistic regression model and enhance the performance through additional processing steps. Furthermore, we will implement a deep learning model using keras.

4.1. Improved Logistic Regression Model

The baseline model was focused on maintaining the regression model on Bag of Words encoding and the stop words filtering. The enhancements attempted were done on top of existing functionality. These enhancements included the addition of n-grams (trigrams) to sparse matrices, no n-grams(not including any n-grams), using TfidfVectorizer, lemmatization with extra pre-processing, adding lexical features, using different classifiers like Naive Bayes, SVM and Random Forest, and also parameter tuning logistic regression via grid search. The performance varied for each of these implementations, many causing the performance to worsen, some stood out contributing to a notable improvement and a few to a very minimum improvement from the baseline model. Each implementation is discussed below along with the insights.

4.1.1. Trigrams (3-grams)

We expanded the model to include 3-grams on the base model. After testing each n-gram on by themselves as well as different combinations, the performance boost wasn’t as significant and could only increase the accuracy by 0.2% maximum. Looking at the results of this implementation the accuracy was 0.66 with an F1score of 0.58 for the macro-average. Concluding, we learned that the combination of unigram, bigram and trigram together led to a minor improvement, and this could be because of the short strings of text from the status which leads to low number of tokens.

```
Classification report for baseline:
              precision    recall  f1-score   support

           n       0.88      0.67      0.76      1619
           y       0.29      0.59      0.39       365

    accuracy                           0.66      1984
   macro avg       0.59      0.63      0.58      1984
weighted avg       0.77      0.66      0.69      1984
```

4.1.2. No Grams Model

Since we learned that there wasn't a significant performance increase by increasing the grams in the model, we wanted to try implementing a regression model without n-grams, and still continue the baseline model of tokenizing, vectorization and using stopwords with shuffle & stratification methods. This gave an unexpected result with a similar accuracy as the n-gram method which is 64.61%. This led to a conclusion that including n-grams has no impact on this dataset and a reasonable performance can be achieved without including the grams.

4.1.3. TfidfVectorizer

We understand that CountVectorizer creates a dictionary based on the occurrence(frequency) of each word that occurs in the entire text. We can cross **TfidfVectorizer** that determines the importance of a particular token and applied on this data to see if there is an improvement in performance. Shown below is the comparison report of the two vectorizers applied to the model with no grams, and included removing stopwords.

```
Classification report for CountVectorizer:
              precision    recall  f1-score   support

           n       0.80      0.66      0.73      1489
           y       0.33      0.49      0.40       495

    accuracy                           0.62      1984
   macro avg       0.56      0.58      0.56      1984
weighted avg       0.68      0.62      0.64      1984

Classification report for tdidfVectorizer:
              precision    recall  f1-score   support

           n       0.93      0.65      0.77      1776
           y       0.17      0.60      0.26       208

    accuracy                           0.65      1984
   macro avg       0.55      0.63      0.51      1984
weighted avg       0.85      0.65      0.71      1984
```

Since we have an imbalance of labels, looking at **macro** average results where all classes are equally important, show that the countvectorizer performed well compared to TfidfVectorizer. By using TfidfVectorizer, the

improvement wasn't remarkable but we learned that the reason might be because the data is shorter and/or the data contains fewer unique words (status).

4.1.4. Lemmatization & extra pre-processing

To improve the bag of words functionality, we applied lemmatization along with extra pre-processing to see if the base model would provide a better performance enhancement. A function 'clean_text' was created such as it removes HTML tags, accented characters and numbers, tokenize, use stopwords and also lemmatize using WordNetLemmatizer. Sadly, there isn't any effect on the performance.

4.1.5. Adding lexical features

Another classification improvement method we used was adding lexical features beginning with extracting the 'STATUS' and 'cNEU' columns as features, respectively. CountVectorizer was used to add lexical features to the text data in the 'STATUS' column. CountVectorizer converts text data into numerical feature vectors. Adding lexical features to a dataset can improve classification by providing additional information to the model about the language and context used in the text data. In the myPersonality dataset, the 'STATUS' column contains text data in the form of social media updates, which may contain specific words or phrases that are indicative of a person's neuroticism. By using CountVectorizer to extract lexical features from the text data, we can transform the unstructured text into a structured numerical representation that can be fed into a machine learning model. This can help the model learn more accurately about the relationship between the text data and the target variable, which in this case is neuroticism. Overall, adding lexical features to the myPersonality dataset can improve classification by providing additional information about the text data, potentially helping the model make more accurate predictions. Here we have the classification report for adding lexical features for the logistic regression classifier using k-fold cross validation:

	precision	recall	f1-score	support
n	0.66	0.82	0.73	6200
y	0.50	0.31	0.39	3717
accuracy			0.63	9917
macro avg	0.58	0.56	0.56	9917
weighted avg	0.60	0.63	0.60	9917

The classification report shows that the precision and recall for the "y" class, indicating high neuroticism, are 0.50 and 0.31. This indicates that when the classifier predicts that an individual has high neuroticism, it is correct 50% of the time, and it identifies only 31% of the individuals with

high neuroticism in the dataset. The precision and recall for the "n" class (indicating low neuroticism) are 0.66 and 0.82, respectively. This means that when the classifier predicts that an individual has low neuroticism, it is correct 66% of the time, and it identifies 82% of the individuals with low neuroticism in the dataset. The F1-score for the "n" class is 0.73. The macro-average F1-score is 0.56, which takes into account the balance between the F1-scores of the two classes.

4.1.6. Different classifiers

We turned our attention to implement other traditional models like SVM, Naive Bayes and Random Forest for neurotic classification. However, the improvement wasn't remarkable. The accuracies for all 3 models were shown below

Summary of classifiers:

	Accuracy
Naive Bayes	0.640625
SVM	0.624496
RandomForest	0.603327

The main idea behind including these traditional classifiers was for a better classification of a user as neurotic or not, but the dataset was so disproportionate of neurotic instances and non-neurotic instances (almost double as neurotic instances) that the anticipated accuracy results of other classifiers were down. Below we can see the **macro** average wasn't as high as our baseline model and there wasn't any improvement.

```
Classification report for MultinomialNB():
              precision    recall  f1-score   support

           n       0.82      0.67      0.74      1509
           y       0.34      0.53      0.42       475

    accuracy                           0.64      1984
   macro avg       0.58      0.60      0.58      1984
weighted avg       0.71      0.64      0.66      1984

Classification report for LinearSVC(max_iter=100000):
              precision    recall  f1-score   support

           n       0.75      0.68      0.71      1373
           y       0.41      0.50      0.45       611

    accuracy                           0.62      1984
   macro avg       0.58      0.59      0.58      1984
weighted avg       0.65      0.62      0.63      1984

Classification report for RandomForestClassifier():
              precision    recall  f1-score   support

           n       0.76      0.66      0.70      1423
           y       0.35      0.46      0.40       561

    accuracy                           0.60      1984
   macro avg       0.55      0.56      0.55      1984
weighted avg       0.64      0.60      0.62      1984
```

4.1.7. Parameter tuning

In this improvement, we used a dictionary of hyperparameters to search over, which include the regularization parameter C and the penalty. We then create a grid search object using GridSearchCV() with 5-fold cross-validation. The best parameters found by the grid were as below for the base logistic regression model and the test data performance after evaluating the final model was again the same which is approximately 64% accuracy.

```
Best hyperparameters:  {'C': 0.1, 'penalty': 'l2'}
Test accuracy:  0.6370967741935484
```

4.2. Improved Model using Bidirectional LSTM

We also implemented a Bidirectional LSTM model for the performance improvement of the baseline classification. Since the data we are dealing with consists of text and sequential data we want to try a neural network model for better predictions and learned that this bidirectional LSTM model would use two LSTM networks; one feed with a forward sequence and another reversed sequence.

Furthermore, with little familiarity ourselves with the LSTM architecture implementing this model was appealing.

After splitting the dataset into train(70%) and test(30%) sets the data was tokenized and padded so that it is feasible to train with a sequential model and LSTM model as shown below was implemented. Finally, since this is a classification problem, we used a Dense output layer with a single neuron and a sigmoid activation function to make 0 or 1 predictions for the two classes in the project. To compile the model log loss is used as the loss function (binary_crossentropy in Keras). The efficient ADAM optimization algorithm is used. The model is fit for only 10 epochs.

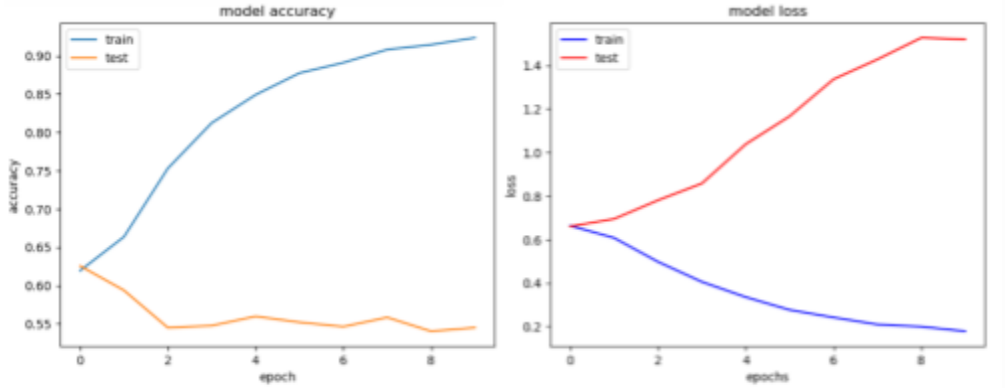
```
Model: "sequential_11"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_11 (Embedding)    (None, None, 32)          160000

 bidirectional_11 (Bidirect  (None, 64)                16640
 ional)

 dense_9 (Dense)             (None, 1)                 65

=================================================================
Total params: 176705 (690.25 KB)
Trainable params: 176705 (690.25 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Unfortunately, this model achieved good accuracy for the train set but gave a poor performance on the test set as shown below. We tried different approaches to work on this problem by changing the train-test split percentages, parameter tuning the model using dropout, number of nodes and hidden layers, and also number of units in the dense layer.



Model accuracy and Loss for LSTM

5. Results - Evaluating on new dataset

The performance of the trained model was assessed using a range of evaluation metrics, including accuracy, precision, recall, and F1-score. Accuracy gauges the overall correctness of sentiment label predictions. Precision is calculated as the ratio of true positive predictions to the total positive predictions, while recall is determined by the ratio of true positive predictions to the total actual positive instances. The F1-score, a balanced measure, is the harmonic mean of precision and recall, providing an overall assessment of the model's performance. From our findings, we can conclude that the improved classifier is train-test split model using CountVectorizer to extract features and preprocess by removing stopwords, along with stratified and shuffle method worked better with an accuracy of 0.65 and with an F1-score of 0.57 for the macro-average.

The *Essays* dataset with about 2700 records is used to evaluate and predict neurotic labels, and also test our saved model. Here the new dataset (which is called evaluate dataset) was preprocessed and the same vectorizer (as the first dataset) was used to feature extract. The pre-trained logistic regression model was used to predict the labels for the new dataset. After training and evaluating the sentiment analysis model, we obtained the following results:

Classification report of saved model on unseen data:

	precision	recall	f1-score	support
n	0.66	0.51	0.58	1607
y	0.36	0.52	0.43	861
accuracy			0.51	2468
macro avg	0.51	0.51	0.50	2468
weighted avg	0.56	0.51	0.52	2468

The model achieved an accuracy of 0.51 indicating that it was able to correctly predict the sentiment labels for the text reviews in the dataset. The F1 macro score of 0.50 indicates a good balance between precision and recall, suggesting that the model performed well in both predicting positive and negative sentiment labels.

6. Conclusion

In this project, we implemented a sentiment analysis model using multiple classification improvement techniques and machine learning algorithms in Python. There are limitations to our approach, including the reliance on labeled data and the limitations of Word2Vec embeddings. Further research can be done to explore other word embedding techniques and contextualized embeddings to improve the performance of the sentiment analysis model. Overall, our approach can be a useful tool for sentiment analysis in various applications, such as market research, customer feedback analysis, and social media sentiment analysis.

7. Challenges & Insights

Throughout the development of this project we faced many challenges to enhance the performance of the classifier. We noticed that most of the changes we implemented to the text processing caused a decrease in the performance and the most accuracy was obtained from the train-test split being done based on the user.

The results obtained from the sentiment analysis model are promising, indicating that the Bag Of Words embeddings combined with other algorithms and methods are effective in capturing the semantic meaning and context of words for sentiment analysis. The accuracy of 65% is considered good for a sentiment analysis task, and the precision, recall, and F1-score values also indicate that the model performed well in terms of both positive and negative sentiment prediction. One of the strengths of our approach is the use of n-grams, which capture the semantic meaning and context of words in a continuous vector format. This allows the model to understand the subtle nuances and context of words, which is important for accurate sentiment analysis. Meanwhile, stratification had a pronounced improvement in accuracy where the model would constantly reach its highest accuracies

Another advantage of our approach is the use of machine learning algorithms, which are known for their ability to learn patterns and make predictions from data. Even though the deep learning model accuracy wasn't that great we experimented with several parameters and gained some knowledge on LSTM models. We also implemented traditional algorithms including Logistic Regression, SVM, and Random Forests, and achieved good performance with all of them. This indicates that our approach is flexible and can be easily adapted to different machine learning algorithms depending on the specific requirements of the sentiment analysis task.

However, there are some limitations to our approach. One limitation is the reliance on labeled data for training the sentiment analysis model. The availability of labeled data may vary depending on the domain or industry for which sentiment analysis is being performed. Obtaining a large labeled dataset may be challenging and time-consuming, and the performance of the model may be affected by the quality and quantity of the labeled data.

Another limitation is that the use of certain classification improvement techniques may not capture all the semantic nuances of words, especially for rare or out-of-vocabulary words. Additionally, the vectorization process may lose some sequential information of the words in the text reviews, which can be important for understanding the context and meaning of a sentence. Further exploration of other word embedding techniques or contextualized word embeddings such as BERT or ELMO may improve the performance of the sentiment analysis model.