Classification of Drug Reviews Sentiment Using Natural Language Processing

Karim Al Zeer Alhusaini
CSC 7810

April 29, 2021

Abstract

Many strides and breakthroughs in the field of Natural Language Processing have been made as a result of recent advances in deep learning, with a broad variety of applications ranging from text classification, sentiment classification, text and speech recognition and generation, translation, to auto-correct. Artificial intelligence is used in Natural Language Processing (NLP) to allow machines to process, comprehend, and extract context and meaning from human language. Usually, this process entails structuring the input text, evaluating the data, and then generating an output. In this research, I used a variety of machine learning and deep learning techniques to classify the sentiment of user feedback for specific drugs, with varying degrees of success. I started by processing the dataset files, then preprocessing the text by removing any noise, punctuation, stop words, correcting for misspellings, and reducing the words to their lemmas. Following that, the terms were vectorized and tokenized in preparation for modeling. I used Naive Bayes, Logistic Regression, Gradient Boosting Classifier, XGBoost, and Support Vector Machines among other supervised learning models. I then developed a Bidirectional LSTM neural network, a CNN with LSTM layer, a Pooled GRU, and a Hierarchical Attention Network using two word embeddings (GloVe and fastText). Finally, I used the BERT advanced transformer system in TensorFlow, which yielded the most accurate results. Ultimately, I built a user-input area using the BERT model, where new feedback can be scored and graded into positive and negative sentiments.

Keywords: Natural Language Processing, Deep Learning, TF-IDF, Word Embeddings, GloVe, fastText Long Short-term Memory, Convolutional Neural Network, Hierarchical Attention Network, BERT

Contents

1 Introduction

It is critical to understand consumer opinion, particularly when it comes to drug reviews. Understanding the drug's user's sentiment and reaction to the medication is critical to the clinical process for a doctor or a pharmaceutical company. It can be difficult to choose the right drug for the right patient. For a variety of purposes, collecting data on the patient's feedback and response to the medication is important. The practitioner may evaluate the drug's efficacy and utility by monitoring the patient's experience. Patients are far more frank and forthcoming when researching items online than when speaking with a physician. This gives the physician the most accurate information about the product. Furthermore, this gives the manufacturer the ability to track the drug's non-clinical performance over time. Finally, consumer sentiment and reviews influence the decisions of other consumers. Patients may avoid a drug if it has received negative reviews, especially if an alternative is available. For these purposes, a reliable model that monitors the feedback posted on *Drugs.com* and *Druglib.com* will aid in predicting the patient's true opinion against the medication prescribed is essential.

While there are many valuable tools on the internet about text classification and natural language processing, this dataset has received little attention and is seldom checked. My aim for this project was to experiment with and eventually create an advanced supervised machine learning model that can categorize the sentiment of a review into two categories: positive and negative. Once the model is trained, it should classify new reviews into their respective sentiments with high accuracy. **Figure 1** below details the goal of the project.

Figure 1: Project Goal: Train a model to classify new data

Text classification is a supervised machine learning method used to classify sentences or text documents into one or more defined categories. It is a widely used natural language processing task playing an important role in spam filtering, sentiment analysis, categorization of news articles, and many other business-related issues. In this project, I focused on Sentiment analysis or classification which allows us to classify text or speech into pre-defined sentiments or class: whether positive, negative, neutral, etc. Most current state of the art approaches rely on two different techniques: the continuous bag-of-words approach, and the continuous skip gram approach. In this project, I applied the standard Term Frequency - Inverse Term Frequency (TF-IDF) vectorized, and the GloVe and fastText embeddings to generate word vectors. This project used Python libraries such as TensorFlow, Keras, NLTK, along with many others. The code was run using Google Colaboratory Pro to get the maximum GPU possible.

2 Literature Review

The chosen dataset is relatively understudied in academic literature. Nonetheless, there is a large number of research and academic papers that investigate the field of text classification. Below is a selection of some published academic and research work that was useful in implementing this project.

- Ye, Zhang, and Law studied a similar dataset on personal reviews for travel-related information on the internet. They used applied two machine learning models: Support Vector Machines, Naive Bayes, and the character N-gram model for sentiment classification. Based on their research, they found that the support vector machines and the N-Gram model performed better than the Naive Bayes model with over 80% accuracy. I was able to apply the Native Bayes and the Support Vector Machines models on my dataset to achieve similar accuracy.
- Huang, Ou, and Carley discussed a new variation of the Long-Short Term Memory model which is called Attention-Over-Attention (AOA) network for aspect level classification which learns the interaction between aspect and context. They applied their model on a dataset containing sentences with different classes: positive, neutral, and negative. In this project, I created a simple version

of this model called the Hierarchical Attention Network (HAN) using a Time-distributed layer to achieve high accuracy.

- Rao, Huang, Feng, and Cong studied sentiments in sentences housed in different documents which is a challenging task. While this does not directly apply to the topic of this project, their method works well. They created an Sentence-Representation LSTM with two layers, one containing the sentence level vectors, the other contains the relations between sentences in different documents. Along with rigorous data processing and cleaning, this SR-LSTM model was able to achieve high accuracy with over 90% correct predictions.
- Munikar, Shakya, and Shrestha focused on classify sentences with more than two sentiments. To achieve high accuracy on complex sentences, they applied the latest BERT transformer model on the movie reviews dataset. I was able to use this technique and apply the BERT model using TensorFlow to achieve over 98% accuracy.
- Sarma, Liang, and Sethares created Domain Adapted (DA) word embeddings, which unline generic word embeddings which are trained on large-scale data, are trained on data related to the domain of interest. I used a similar approach in this project where rather than apply the existing word embeddings from Google and Facebook, I used the TF-IDF vectorizer to create a word embedding that is unique to the dataset. They applied their data on a similar sentiment classification dataset and achieved a high accuracy.
- Lastly Zhang, Zheng, Jiang, Huang, and Chen created a unique architecture that combines Convolutional Neural Networks (CNN) with LSTM to learn the intrinsic semantic and emotional information in the text. The model begins with a CNN layer that is then pooled before applying an LSTM layer to capture the meaning. Their model produced high accuracy and I was able to create this architecture using the defined Keras layers to achieve a similarly high accuracy.

3 Methodology

I followed a step by step methodology in my project. **Figure 2** details the methodology followed in this project. First, I explored the dataset and prepared the text for analysis. This involved separating the text by white space, removing stop words, punctuation, and any symbols, unique characters, and HTML tags. The text was then lemmatized. Then, I extracted features from the text by implementing the Bag of Words (BoW) using the Term Frequency-Inverse Document Frequency (TF-IDF) vector value for the text. I created a pipeline that first converts the text into a matrix, then runs the TF-IDF normalization, followed by 5 different classification models: Naïve Bayes, Logistic Regression, Support Vector Machine, Gradient Boosting Classifier, and XGBoost. These models established a baseline understanding of the classification problem. Then, I applied different pretrained word embeddings, including Word2Vec, GloVe, and fastText. I used these embeddings to create Deep Learning models, including a Bi-Directional LSTM, CNN-LSTM, HAN, and a Pooled GRU model. My hyper parameter tuning for these models was mainly a trial and error as using an automated tuning method was not efficient on such a large dataset. I then experimented with advanced transformer models, mainly focusing on BERT.

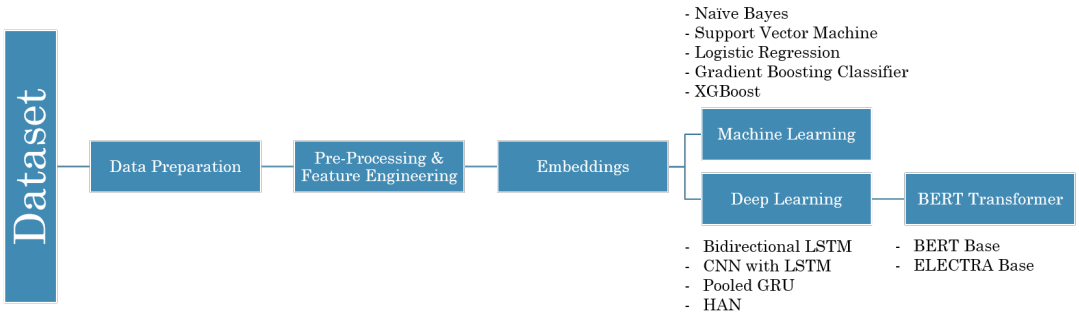


Figure 2: Proposed Methodology

3.1 Metrics

As this is a classification problem, I used the following classification metrics:

- Accuracy: accuracy of the classification which computes the number of correct predictions over the number of observations in the sample.
- Precision: this refers to the ability of the classifier not to label a negative observation as negative. In other words, it is the sum of true positive observation divided by the sum of the observations that were predicted as positive.
- Recall: which computes the classifier's ability to find all the positive samples. It is the same as the True Positive Rate, which is the sum of true positive observations over condition observations that are positive.
- F-1 Score: which is a harmonized weight combining the precision and recall values.

$$F_1 = 2 * \frac{precision * recall}{precision + recall}$$

- AUC: Area under the Curve of the Receiver Operating Characteristics plot.

4 Data

4.1 Data Source

The dataset contains patient reviews on specific drugs along with related conditions and a patient rating reflecting overall patient satisfaction. I pulled the data from the *UC Irvine Machine Learning Repository*. The original data was extracted using web crawl performed on *Drugs.com* and *Druglib.com*. One of the motivations for selecting this dataset is that it is relatively understudied in online literature, particularly in applications of advanced methods. This made it a good canvas to learn the challenges of building an effective text classification model. The native format was of the data sets was .tsv but I converted it to .csv before any analysis. The data was already split into training and test sets, with a 60:40 split. To my advantage, both sets were labeled, so I merged the two datasets into one using pandas. I later split the data into different training and testing sets depending on the different models I ran to optimize the models' training performance. Some reviews contained a single word review or a single character and a few that had a Null value. I dropped these reviews as they have no value in this analysis.

4.2 Data Dictionary

The combined dataset contained 121, 573 reviews in total with a median word count of 112. There was a total of 541 drugs related to 1808 different medical conditions. These input features seem to be user inputted and as a result, many of these conditions are redundant due to misspelling or other user input issues. These features, nonetheless, were dropped for this project. **Table 1** shows the different features that were included in the raw dataset.

Feature	Type	Description
DrugName	Categorical	Name of the drug reviewed
Condition	Categorical	Name of the medical condition
Review	Text	Review inputted by the customer
Rating	Categorical	Customer Rating of the drug
Date	Date	Date review was written
UsefulCount	Numeric	Number of users who found the review useful

Table 1: Raw Table Features

The two features that were used in this project are the text *Review* feature, which contains the customer review, and the *Rating* feature, which is the target variable: 0 for a negative review, 1 for positive.

4.3 Exploratory Data Analysis

Before any preprocessing of the reviews, I did an exploratory data analysis on the text to get a feel for the problem. First, I created a function to produce the word count, unique word count, character count, mean word length, and punctuation count for each review. **Figure 3** shows the distribution of the word count for each of the target variables. Most reviews are between 100 and 200 words long, with a few outliers reaching as many as 1000 words. These outliers were not excluded from the data as they still could have predictive capabilities. When I examine the count of unique words in each review, the mean word count drops compared with the original count. The mean unique word count before preprocessing was 112 words with a total of approximately 11 million words and characters. It can be seen from the figure that the distribution of the word count is not uniform, with the distribution having two peaks. This could be attributed to the fact that the data was pulled from two different sites which could have different word count limits.

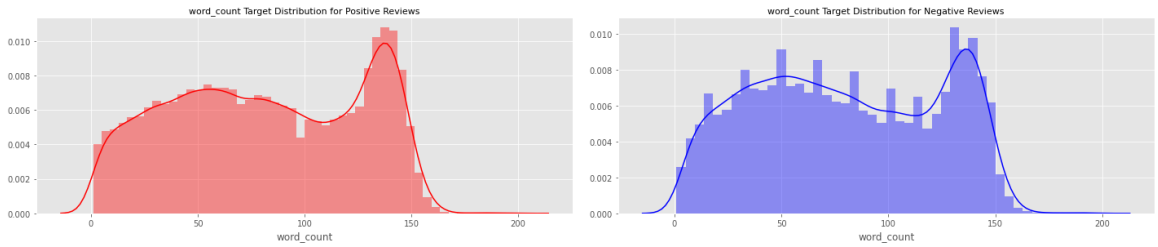


Figure 3: Distribution of word count by target variable (Red positive)

In terms of target variable distribution, although there were more positive reviews than negative, the distribution is approximately 60% positive and 40% negative. To cure the class imbalance, I first applied class weights using the following formula:

$$w_i = \frac{1}{class_i} * total * \frac{1}{2}$$

.

However, I found that oversampling from the under-represented class to reach a 50:50 class balance achieved better results when training the base models. To a human, the new values are repetitions of existing data points but to model, it is handled as a new data point for training. **Figure 4** plots the count plot for each of the target classes before oversampling. After oversampling the data, I split it into 60:20:20 training, validation, and testing split used across all models built in this project.

Figure 4: Distribution of the Target variable

In the process of Exploratory Data Analysis, I added more features to the dataset, such as word count, character count, and others. The purpose of these features was to aid in the EDA process. However, for future further study, I believe these features can be included in a future model as there could be distinguishing factors between the characteristics of the positive and negative reviews that are not linguistic or of textual nature.

4.4 Data Preprocessing

The first step to building an effective NLP machine learning model is to preprocess and analyze the text at hand. Once the preliminary assessment was over, I was able to perform some preprocessing on the data set that is necessary to build an effective text classification model. It is often difficult to begin modeling without having a preprocessed dataset. Having a well-processed dataset is important in the feature extraction process before modeling. Before making any changes to the text, I manually analyzed a random sample of the reviews to observe for any abnormalities that may be unique to this data set. I first noticed that the drug name is often mentioned in the review. I decided to leave the drug name included in the text rather than remove it as it could be an important feature, i.e. some drugs have more negative reviews than positive which could impact the prediction. Furthermore, as reviews tend to be very informal, there were many abbreviations, contractions, and punctuation that had to be dealt with. The following steps detail the preprocessing methods I followed in the dataset.

1. Splitting by White space: to make it easier to analyze the text, I split the text by white space in a list of words.
2. Use the PySpelling package in python to apply minimal spell checking and correction with a high threshold. This is to not over-correct some words and to only correct were necessary.
3. Remove punctuation: I used the *NLTK* package to remove the punctuation in the text. I also applied a list of additional punctuation and removed what the *NLTK* method may have missed
4. Applying lowercase: to ensure consistency in the text, I enforced lower case on all the words in the text.
5. Remove stop words: I used the *NLTK stopwords* library to remove all stop words that add noise to the data set.
6. Remove HTML/XML tags or markup: this step may have been unnecessary, but it is common for web text data to include some HTML/XML tags that could have been added in the scraping process. I used *BeautifulSoup* package to apply this rule.
7. Lemmatization: I used the *NLTK* package to convert the words into their lemmas where ever possible. I did not apply the stemming rule as I noticed the words became less recognizable in the word embedding process that followed. Additionally, lemmatization uses the context of the sentence whereas stemming considers only the one word.

Nevertheless, some of the models I used, such as BERT do not require thorough preprocessing, such as lemmatization, as some of the contextual value is lost when applying the model. Therefore, I created a separate data set for the BERT transformer model that did not include lemmatization.

Once I finished the preprocessing, I created N-Gram visualizations of the text that display the most

common word or n number words that are common in each class. **Figure 5** shows the Trigram for each of the classes. For instance, the phrase "feel much better" is more prevalent in positive reviews, while the phrase "horrible side effect" is more prevalent in negative reviews.

There are two models that are commonly used in literature. The Continuous Bag of Words (CBOW) model, which predicts a word given the word's neighboring context. The other is the Continuous Skip-Gram model, which predicts the context or neighboring words given the word itself. The Skip-Gram model runs slower than the CBOW model, but produces better results, especially in a text classification problem such as the one discussed in this project.

Lastly, I applied the word embedding models before and after the preprocessing to check the text and vocabulary coverage. Word Embeddings are learned representation for text where words that have the same meaning have a similar representation. I applied the GloVe (840B.300D), fastText(300d-2M), and Word2Vec word embeddings. Before the preprocessing, GloVe covered 31% of the vocabulary and 90% of the text, while fastText covered 35% of the vocabulary and 91% of the text. After I cleaned up the text, GloVe covered 78% of the vocabulary and 99% of the text, while fastText covered 77% of the vocabulary and 99% of the text, a significant improvement. The total number of words after preprocessing was approximately 4.7 million.

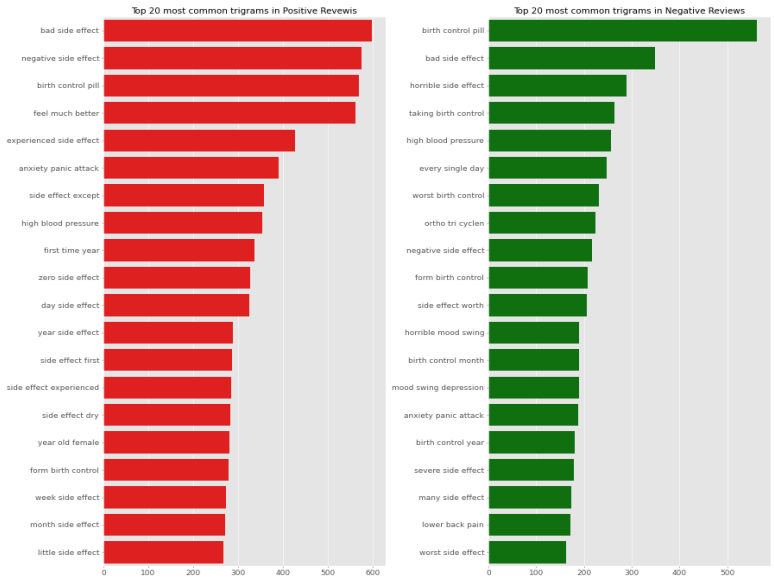


Figure 5: Trigram of the most common words in each class

I also used *gensim* library to implement the Word2Vec embedding. I trained the embedding on my data to check if it produces meaningful vocabulary similarities. **Figure 6** shows the similarities retrieved for the word "pain". These similarities are associated with a positive review. I also applied the t-SNE manifold method to visualize the word similarities for the Word2Vec embedding. **Appendix 1** plots the two t-SNE components.

```
w2v_model.wv.most_similar(positive=['pain'])
```

```
[('affect', 0.8622065782546997),
 ('drowsiness', 0.44682586193084717),
 ('benefit', 0.4073120355606079),
 ('note', 0.3956373929977417),
 ('tiredness', 0.37383872270584106),
 ('unpleasant', 0.3640897572040558),
 ('sleepiness', 0.33923178911209106),
 ('except', 0.3333709239959717),
 ('sexual', 0.3146399259567261),
 ('symptom', 0.30873697996139526)]
```
Figure 6: Word2Vec Word Similarities

5 Models & Results

5.1 Machine Learning Models

I first began by establishing baseline models for this classification problem. I created an automated pipeline that runs through 5 classification models. First, I applied the **Count Vectorizer** function which converts a collection of text documents to a matrix of token counts. This method produces a sparse matrix representation of the counts that is equal to the vocabulary size found by analyzing the data. I then applied the **TF-IDF Transformer** function which transforms the produced token matrix into a normalized TF-IDF representation. The TF-IDF (Term Frequency - Inverse Document Frequency) frequency is used rather than the raw frequencies because it produces a frequency that is scaled with the impact of the specific word in the text data. Once these functions were applied, I was able to run the classification models detailed below. The classification results below are all based on predictions made on the 20% untrained testing set.

5.1.1 Naïve Bayes Classifier

The first method I applied was the Naïve Bayes classifier. I used the sklearn's multinomial Naïve Bayes classifier as the input features were continuous. This classifier is ideal for text classification as it is suitable for the classification of discrete features, such as text word counts or text frequencies, such as the Tf-IDF. **Figure 7** shows the classification report for the NB method. The model produced a relatively high accuracy given its simplicity. The accuracy was 83% with an AUC score of 0.8 on the testing set. The advantage of such a classifier is its ease of use, efficient computation, and scalability.

```
accuracy 0.8323566952889302
ROC AUC 0.8081140675774485
              precision    recall  f1-score   support

           0       0.89      0.67      0.77     14104
           1       0.81      0.94      0.87     20368

    accuracy                           0.83     34472
   macro avg       0.85      0.81      0.82     34472
weighted avg       0.84      0.83      0.83     34472
```

Figure 7: Naïve Bayes Classifier - Report

5.1.2 Logistic Regression

I applied a Logistic regression with a ridge regularization value of 0.00001 and a cross-entropy loss. This model was very accurate and produced 90% accuracy with an AUC value of 0.89 on the testing set. **Figure 8** shows the classification report for the Logistic Regression model. The reason for the high accuracy could be due to the normalization of the features and their proper presentation. Nonetheless, logistic regression does not take into account the context in which the word is presented in and merely looks at the word as a feature.

```
accuracy 0.8980912044557902
ROC AUC 0.893286653761375
              precision    recall  f1-score   support

           0       0.88      0.87      0.87     14104
           1       0.91      0.92      0.91     20368

    accuracy                           0.90     34472
   macro avg       0.90      0.89      0.89     34472
weighted avg       0.90      0.90      0.90     34472
```

Figure 8: Logistic Regression Classifier - Report

5.1.3 Support Vector Machine

I used the Stochastic Gradient Descent (SGD) classier in Sci-kit learn which by default implements a Support Vector Machine (SVM) classifier with convex loss functions. This method is a conceptually simple but efficient approach to classification. It is also flexible and can be accurate even on natural

language data. I applied this classifier using a hinge loss with a loss parameter of 0.001. **Figure 9** shows the classification report for the SVM model. This model produced 80% accuracy with an AUC value of 0.76 on the testing set. However, it had a low recall value when predicting negative reviews. This low recall value could due to some words appearing frequently in both positive and negative reviews but within a different context.

```
accuracy 0.8022743095845903
ROC AUC 0.7682878094674551
              precision    recall  f1-score   support

           0       0.90      0.58      0.71     14104
           1       0.77      0.96      0.85     20368

    accuracy                           0.80     34472
   macro avg       0.83      0.77      0.78     34472
weighted avg       0.82      0.80      0.79     34472
```

Figure 9: SVM Classifier - Report

5.1.4 Gradient Boosting Classifier

Gradient Boosting Classifier is a built-in classifier in Sci-Kit learn. It is an ensemble boosting method in which several of predictors are aggregated to form a final prediction. In boosting methods, such as GBC, the predictors are trained sequentially, meaning the error of one stage is passed as an input to the next stage and so on. I ran this model which produced inadequate accuracy at 78% with an AUC of 0.75 on the testing set. The recall value produced impacted the overall accuracy of the model. **Figure 10** shows the classification report for the GBC method.

```
accuracy 0.7781387792991413
ROC AUC 0.7483215248948558
              precision    recall  f1-score   support

           0       0.82      0.58      0.68     14104
           1       0.76      0.91      0.83     20368

    accuracy                           0.78     34472
   macro avg       0.79      0.75      0.76     34472
weighted avg       0.79      0.78      0.77     34472
```

Figure 10: GBC Classifier - Report

5.1.5 XGBoost

The last classifier I applied is another ensemble boosting method. XGBoost, which is arguably the most powerful of all the boosting methods. It is a decision tree-based method that uses an optimized "extreme" gradient boosting through parallel processing, tree-pruning, and regularization to avoid over-fitting. XGBoost requires little feature engineering yet it can produce quick results with high performance and minimal overfitting. This model, however, produced similar results to the GBC method, with an accuracy of 78% and an AUC of 0.75 on the testing set. **Figure 11** shows the classification report for the XGBoost method.

```
accuracy 0.777993734045022
ROC AUC 0.7495070983188961
              precision    recall  f1-score   support

           0       0.81      0.59      0.69     14104
           1       0.76      0.91      0.83     20368

    accuracy                           0.78     34472
   macro avg       0.79      0.75      0.76     34472
weighted avg       0.78      0.78      0.77     34472
```

Figure 11: XGBoost Classifier - Report

5.1.6 Results

Out of all the baseline models, I found that the logistic regression model produced the highest accuracy of all. This was unexpected as I thought the ensemble methods GBC and XGBoost would yield the best results. I believe the low performance could be due to the highly featured data set. **Table 2** summarizes the results when the models were run on the untrained testing set.

Model	AUC	Precision	Recall	F1-Score
NB	0.81	0.84	0.84	0.83
SVM	0.77	0.82	0.80	0.79
Log Reg	**0.89**	**0.90**	**0.90**	**0.90**
GBC	0.75	0.79	0.78	0.77
XGBoost	0.75	0.78	0.78	0.77

Table 2: Baseline Model Metrics

5.2 Deep Learning with Word Embeddings

The models above establish a good baseline for this text classification problem. However, these models treat the text tokens solely as features and ignore the context in which the text appears. Advanced deep learning models can solve this problem and achieve better results. The success of these models relies on its capacity to model complex and non-linear relationships within the data.

In building the deep learning models, I used word embeddings. Word embeddings are a type of distributed word representation that allows words with similar meanings to have a similar representation using the cosine similarities between the different words. Word embeddings are a class of techniques where individual words are represented as real-valued vectors in a predefined vector space. Each word is mapped to one vector and the vector values are learned in a way that resembles a neural network, and hence the technique is often lumped into the field of deep learning. The learning process is usually joint with the neural network model on some tasks.

For each of the deep learning models, I used three different types of word embeddings. First, I used a 'learn from scratch' approach in which the text is converted into vectors with a shape that matches the chosen vocabulary size and embedding dimension size. This embedding is added as a layer within the deep learning models were the model has to compute the weights for each word from scratch. The second embedding I used was Stanford's GloVe. GloVe is an unsupervised learning algorithm for obtaining vector representations for words. Training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representations showcase interesting linear substructures of the word vector space. For this particular application, I manually downloaded and used the "6B - 100d" embedding which is trained on Wikipedia and Gigaword 5 words. The third embedding I used is Facebook's fastText which is trained using a Web Crawl method. The dimensions of the embedding are 1M - 300d. The dimensions for the fastText embedding are higher than those of GloVe, however, 300d is the lowest dimension that fastText provides, which made the performance of the fastText embedding more memory intensive. For the GloVe and fastText embedding, I computed the word weight matrix associated with each word present in the text. This matrix was then applied as an embedding layer with initialized weights for each of the deep learning models.

I built three different types of deep learning models. Recurrent Neural Networks, Convolutional Networks, and Attention Networks. Recurrent neural networks have the capability of processing sequential information. These networks perform the task over each instance of the sequence where the output is dependent on the previous results. Therefore, recurrent networks have a memory over the previous computations and use this memory in processing of the current sequence. RNNs are ideal for language-related tasks such as text classification as they can capture the sequential nature of language, however, if not designed carefully, they can suffer from quickly vanishing gradient. Similarly, Convolutional neural networks can also be used in text classification purposes. They work by extracting the n-gram features from the words. They typically have two feature extractors: a convolutional layer, and a pooling layer. Lastly, Hierarchical Attention Networks use stacked recurrent neural networks on word-level followed by an attention mode to extract words that are important to the meaning of the sentence and then aggregate

the representation of those words int vectors.

5.2.1 Long Short-Term Memory Network

LSTM models are a type of RNNs that can overcome the problem of vanishing gradient by allowing the error to propagate through an unlimited number of times. For my model, I used a Bidirectional LSTM layer. The difference being a unidirectional LSTM can only process the info from the past to the future (single direction) while a bidirectional LSTM can process the information for the past to the future and in reverse. For example, if a network is considering an input sentence, then the bidirectional network would go from the beginning of the sentence and the end of the sentence to compute the weights and predict. This gives the LSTM an advantage in processing contextual information. In essence, it is merging two RNN layers.

Before creating the networks, I had to convert the text into a format that is suitable for deep learning models. The text was tokenized then converted to sequences of a maximum length of 128, which is optimal for such text size. Any empty tokens were counted or padded as zeros. I also used a vocabulary size of 35,000 words for the embedding. For the first model, I did not use any pre-trained word embeddings, so no weights were initialized. The embedding dimensions were fixed at 100, which is plenty enough. Only the fastText embedding dimension had to be changed to 300 to match the minimum embedding dimensions. The embedding layer was followed by a bidirectional LSTM layer which was followed by one dense layer and an output layer with a Sigmoid activation function. I used the binary crossentropy loss, with the Adam optimizer. The models were set to run for 10 epochs, but I added an EarlyStopping mechanism that ends the training if the validation loss does not change by a magnitude of 0.0001 between epochs. **Figure 12** shows the architecture of the Bidirectional LSTM models. I kept 10% of the dataset for testing and prediction, while 20% was used for validation during the training process.

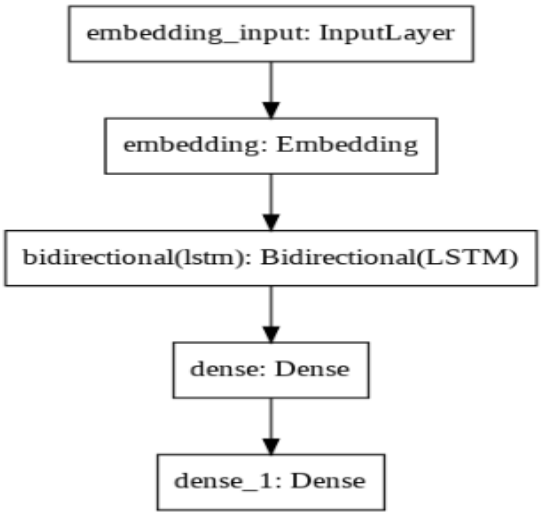
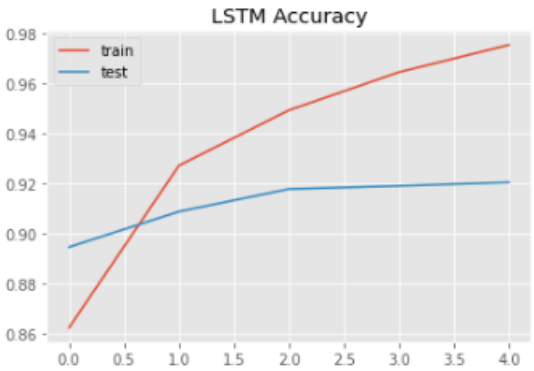


Figure 12: Architecture of the LSTM Model

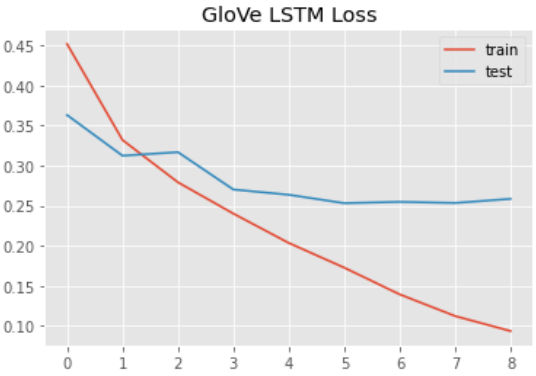
Figures 13, 14, & 15 show the loss and accuracy on the training and validation sets for the Bidirectional LSTM model with no pre-training word embeddings, with GloVe embeddings, and fastText embeddings respectively. The model automatically stopped for the no embedding and the fastText embedding models after 5 and 6 epochs respectively as the validation loss remained unchanged for 3 epochs. Meanwhile, for the GloVe model trained for 9 epochs.

(a) LSTM No Embedding - Loss

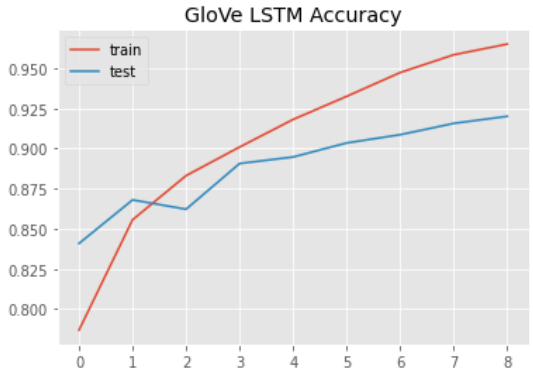


(b) LSTM No Embedding - Acc

Figure 13: Bidirectional LSTM Performance



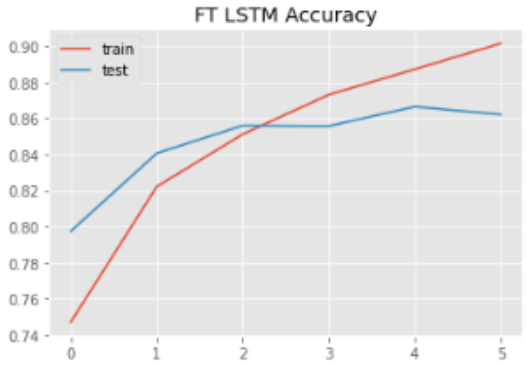
(a) LSTM GloVe - Loss



(b) LSTM GloVe - Acc

Figure 14: Bidirectional LSTM Performance using GloVe Embeddings

(a) LSTM fastText - Loss



(b) LSTM fastText - Acc

Figure 15: Bidirectional LSTM Performance using fastText Embeddings

Table 3 summarizes the classification metrics for each of the models. The GloVe embeddings model produced the best results out of the three models despite taking more epochs to train. The fastText model was the least efficient as it consumed high memory and trained for a few hours yet produced the least impressive results. The model with no embeddings produced good results while remaining efficient

Model	AUC	Precision	Recall	F1-Score
LSTM - Learn from Scratch	0.91	0.92	0.92	0.92
LSTM - GloVe	**0.92**	**0.92**	**0.92**	**0.92**
LSTM - fastText	0.85	0.86	0.86	0.86

Table 3: Bidirectional LSTM Model Metrics

5.2.2 Convolutional Neural Network with LSTM layer

The second deep learning model I build was a Convolutional neural network with an LSTM layer. The Bidirectional LSTM model worked well but it was inefficient as it consumed a lot of memory and the training was time-consuming. Adding a convolutional layer with a pooling layer can help alleviate this problem. The convolutional layer passes a filter over the data and computes a higher representation. By adding an LSTM layer at the end, this can increase the capability of the model by making it sequential. The model begins with an embedding layer, followed by a dropout, followed by a convolutional layer. Convolutional layers can be multi-dimensional, however, for this application, I used a one-dimensional convolutional layer with a one-dimensional maximum pooling layer that works well with temporal data. Higher dimensional layers are designed for spatial and image processing. The pooling layer downsamples the data extracted from the convolutional layer to reduce the dimensionality of the feature map. The model ends with the LSTM layer and another dense layer with a sigmoid activation function that summarizes the results of the model by performing pattern recognition on the features extracted from the pooling layer. **Figure 16** shows the architecture of the CNN-LSTM model. I used a Binary Crossentropy loss, with the Adam optimizer.

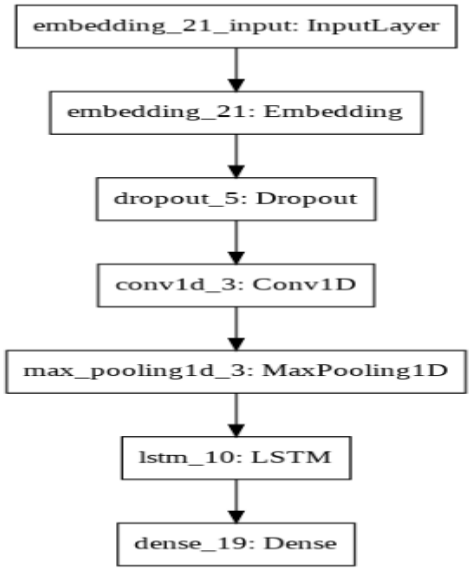
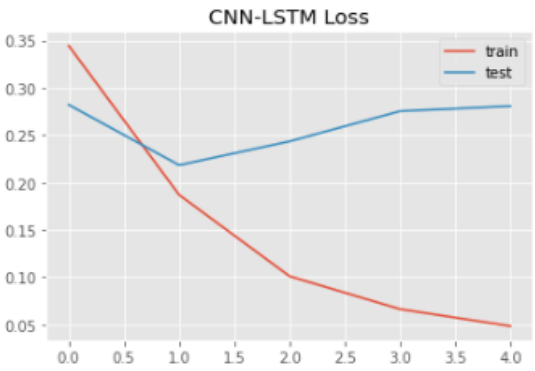
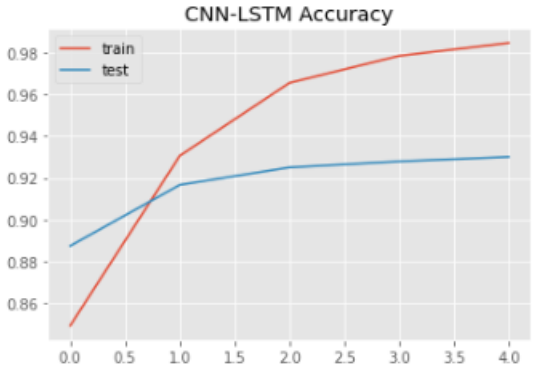


Figure 16: Architecture of the CNN-LSTM Model

For this model, the text had to be converted into sequences. I used a maximum sequence length of 128 with padding. I also used a maximum vocabulary size of 35,000 words. The embedding layer dimensions was 100 for the learn-from-scratch and GloVe models, and 300 for the fastText model. **Figures 17, 18, & 19** show the training loss and accuracy for each of the models. The models were set to run for 10 epochs with an EarlyStopping mechanism.

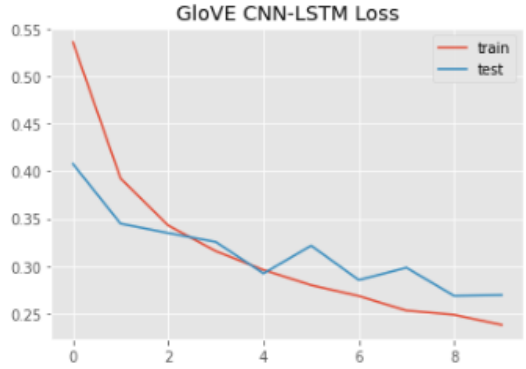


(a) CNN-LSTM No Embedding - Loss

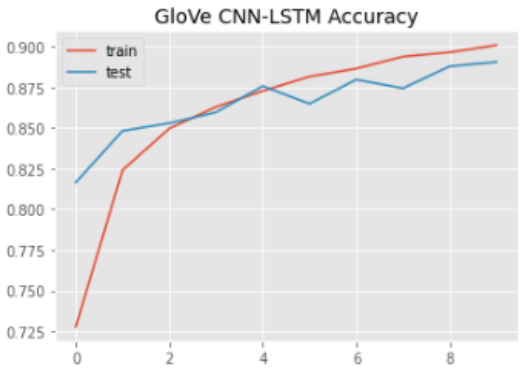


(b) CNN-LSTM No Embedding - Acc

Figure 17: CNN-LSTM Performance

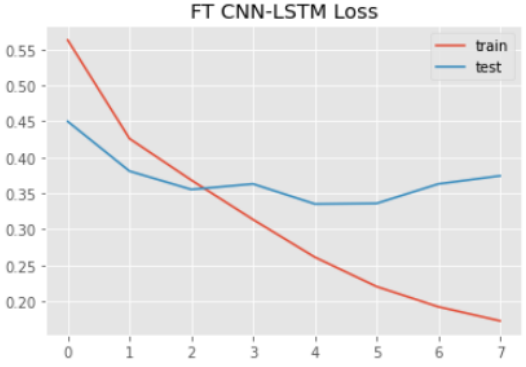


(a) CNN-LSTM GloVe - Loss

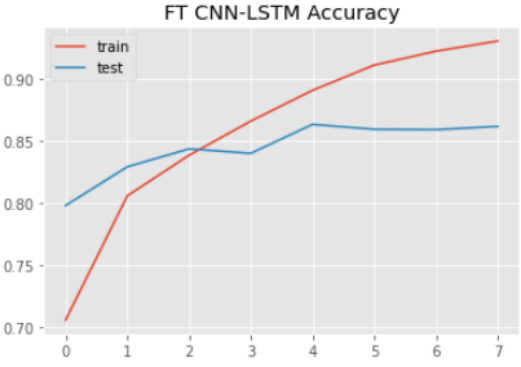


(b) CNN-LSTM GloVe - Acc

Figure 18: CNN-LSTM Performance using GloVe Embeddings



(a) CNN-LSTM fastText - Loss



(b) CNN-LSTM fastText - Acc

Figure 19: CNN-LSTM Performance using fastText Embeddings

Table 4 summarizes the classification metrics for each of the models. The learn-from-scratch model produced the best results at 93% accuracy with an AUC value of 0.92. The GloVe model had slightly less accuracy at 88% accuracy, and the fastText model was the least effective model at 85% accuracy. The first model ran for 5 epochs while the GloVe model ran for 10 epochs and the fastText model ran for 8 epochs. The CNN-LSTM was more efficient than the Bidirectional LSTM model. The training time was 50% faster yet the results were somewhat comparable.

Model	AUC	Precision	Recall	F1-Score
CNN-LSTM - Learn-from-scratch	**0.92**	**0.93**	**0.93**	**0.93**
CNN-LSTM - GloVe	0.88	0.89	0.89	0.89
CNN-LSTM - fastText	0.85	0.86	0.86	0.86

Table 4: CNN-LSTM Model Metrics

5.2.3 Pooled Gated Recurrent Unit

Gated Recurrent Units (GRU) are another improvement on the simple Recurrent Neural Networks. They are conceptually similar to LSTM models and they both attempt to cure the severe vanishing gradient problem that plagues RNNs. GRUs make use of update and reset gates. These are essentially vectors that control the flow of information that is passed to the output layer. Like the LSTM model, GRUs can store information from the past and pass it to the future output. Pooled GRUs, as the name implies, are GRU models that contain pooling layers. **Figure 20** shows the architecture for the Pooled GRU mode. I start with the standard embeddings layer, followed by a spatial dropout layer. A spatial dropout layer is the same as the regular dropout layer but rather than drop one feature map, it drops the 2D feature maps which is generated in the embedding layer. Then I add a bidirectional GRU layer followed by 2 pooling layers: maximum and average pooling which helps in reducing the dimensionality of the output before going into a dense layer with a Sigmoid activation function. The two pooling layers are then concatenated and compiled using a binary crossentropy loss, and Adam optimizer.

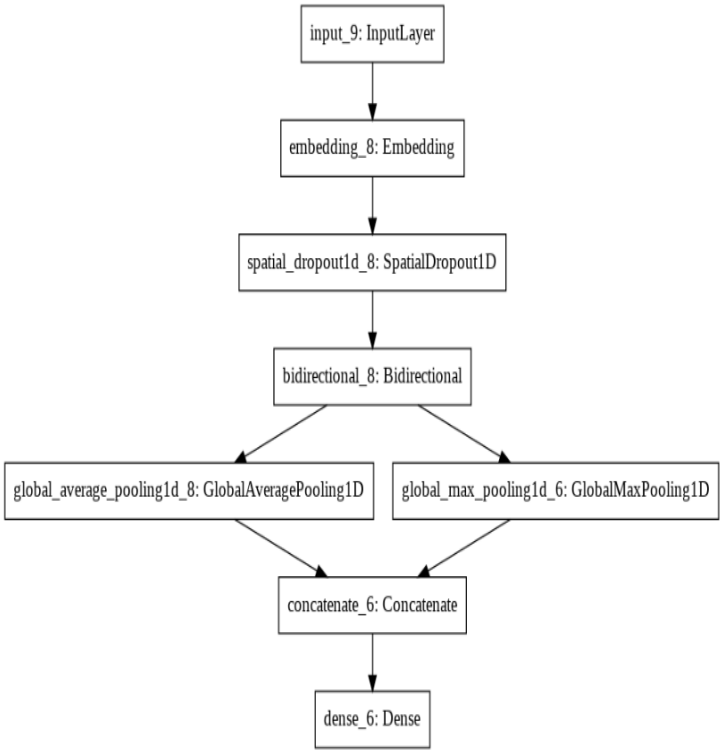
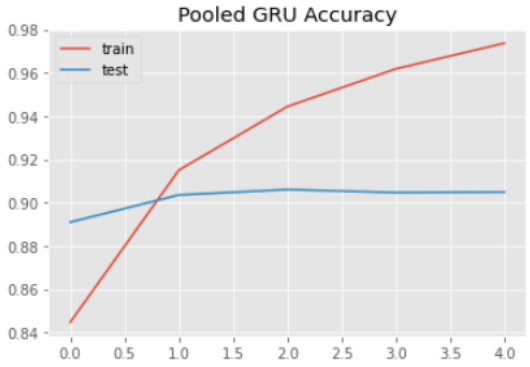


Figure 20: Architecture of the Pooled GRU Model

One drawback to the Pooled GRU model is that Keras does not produce predicted classes on the testing set. Therefore, I was not able to produce a classification report for this model. Instead, I used the *fit* function in keras to fit the model on the testing dataset to produce loss and accuracy results. **Figures 21, 22, & 23** show the performance of the Pooled GRU models using 3 different embedding weights.

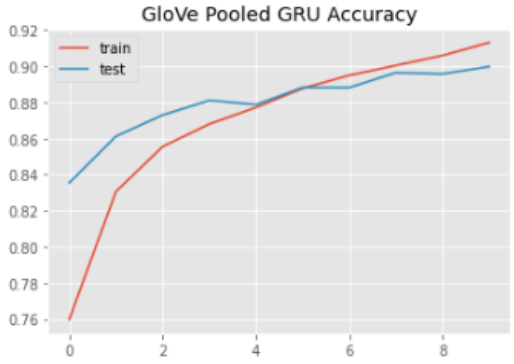
(a) Pooled GRU No Embedding - Loss



(b) Pooled GRU No Embedding - Acc

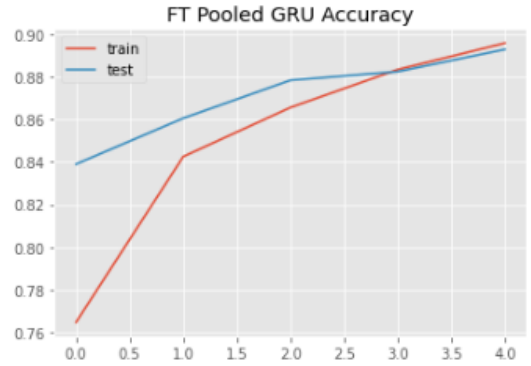
Figure 21: Pooled GRU Performance

(a) Pooled GRU GloVe - Loss

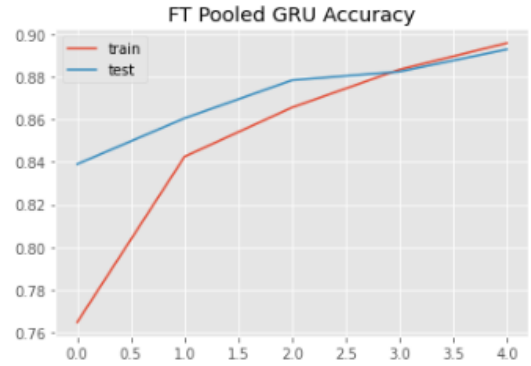


(b) Pooled GRU GloVe - Acc

Figure 22: Pooled GRU Performance using GloVe Embeddings



(a) Pooled GRU fastText - Loss



(b) Pooled GRU fastText - Acc

Figure 23: Pooled GRU Performance using fastText Embeddings

The Pooled GRU model was more efficient than the LSTM and CNN models despite containing two pooling layers. I ran the model for 10 epochs but stopped early at 5 for the learn-from-scratch and the fastText models. It ran for 10 epochs on the GloVe model. **Table 5** shows the performance results for the 3 models on the testing dataset. The GloVe model produced the best results, at 90% accuracy with 0.25 loss. The learn-from-scratch model had equally good accuracy results but produced higher loss at 0.31. The fastText produced slightly lower accuracy at 89% and a good loss at 0.26.

Model	Accuracy	Loss
Pooled GRU - Learn-from-scratch	**0.91**	0.31
Pooled GRU - GloVe	0.90	**0.25**
Pooled GRU - fastText	0.89	0.26

Table 5: Pooled GRU Model Metrics

5.2.4 Hierarchical Attention Network

Hierarchical Attention Network is a type of temporal network which I came across while reviewing literature for this project. It is similar to Recurrent Neural Networks but adds an attention mechanism that identifies the most important parts of the text it considers, such as the most important parts of the paragraphs or sentences. It is hierarchical because it builds an RNN and an attention mechanism hierarchically. For example, if a document is fed to the network, then it would break it down to paragraphs, then sentences, then words. At each step, it considers the most important parts of the document, paragraph, sentence, and word in order. HANs are particularly useful for document classification due to their hierarchical nature. For this model, I used an existing Keras layer *Timedistributed* to build the HAN architecture. Furthermore, rather than tokenize only words within the text, I had to tokenize sentences within the text, which I did not have to do for the previous models. In addition to the time distributed layer, I also added a Bidirectional LSTM layer which allows the HAN to move in both directions of the text. **Figure 24** shows the architecture of the HAN model.

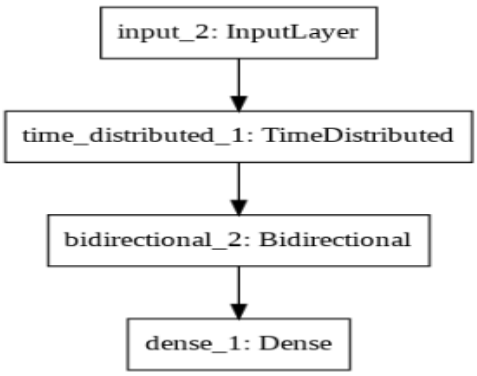
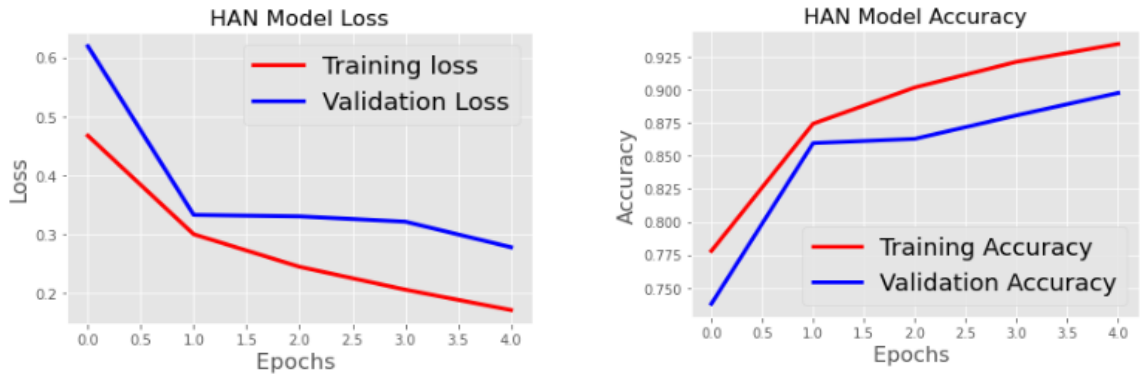


Figure 24: Architecture of the HAN Model

I attempted to run HAN in the same way I did for the previous models, however, it was extremely computationally expensive. I was able to run it only on the GloVe embedding alone. **Figure 25** show the performance of the HAN model using the GloVe embedding



(a) HAN No Embedding - Loss

(b) HAN No Embedding - Acc

Figure 25: HAN Model Performance

Like the Pooled GRU model, the HAN model contained a 'model' layer in the architecture to combine

the input and output, and I was unable to compute the classification report. The HAN model produced very similar results to the previous models. The best accuracy achieved was 90% with a 0.28 loss. **Table 6** details the results of the model.

Model	**Accuracy**	**Loss**
HAN - Learn-from-scratch	0.90	0.28

Table 6: HAN Model Metrics

5.3 BERT Model & User-Input Classification

Bidirectional Encoder Representations from Transformers or BERT is an advanced method developed by Google in late 2018. According to Google AI:

> 'BERT is a method of pre-training language representations, meaning that I train a general-purpose "language understanding" model on a large text corpus (like Wikipedia), and then use that model for downstream NLP tasks that I care about (like question answering). BERT outperforms previous methods because it is the first unsupervised, deeply bidirectional system for pre-training NLP.' **[4]**

The reason BERT is unsupervised is that it was trained using plain text corpus from text data available on the web, so it can be both contextual and non-contextual. According to Google API, BERT should outperform other methods due to its deep bidirectionality, unlike existing methods such as ELMo and ULMFit which are unidirectional. BERT is a Transformer model in that it allows for parallel processing of text or words in sentences which is not possible in other methods, such as LSTM. BERT can also be Transfer Learning model as I could use the pre-trained model and tune it to the data.

According to Google, these are the current BERT models and their descriptions:

- **BERT-Base**, Uncased and seven more models with trained weights released by the original BERT authors.
- **Small BERTs** have the same general architecture but fewer and/or smaller Transformer blocks, which lets you explore trade-offs between speed, size and quality.
- **ALBERT**: four different sizes of "A Lite BERT" that reduces model size (but not computation time) by sharing parameters between layers.
- **BERT Experts**: eight models that all have the BERT-base architecture but offer a choice between different pre-training domains, to align more closely with the target task.
- **Electra** has the same architecture as BERT (in three different sizes), but gets pre-trained as a discriminator in a set-up that resembles a Generative Adversarial Network (GAN).
- **BERT with Talking-Heads Attention and Gated GELU** [base, large] has two improvements to the core of the Transformer architecture.

For my models, I ran the Small BERT base and the Electra Base models using Tensorflow high GPU. Unfortunately, these model are extremely computationally expensive with each epoch project lasting for an hour. The models had to run GPU rather than CPU due to the parallel processing nature of BERT. **Figure 26** shows the architecture of the model I built using BERT. I used the binary crossentropy loss and the Adam optimizer. **Appendix 5** shows the base BERT architecture.

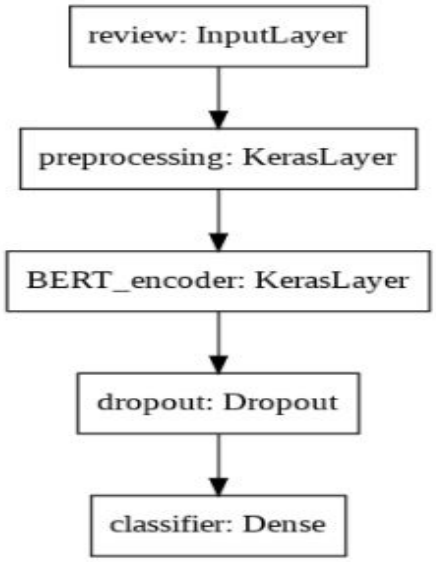
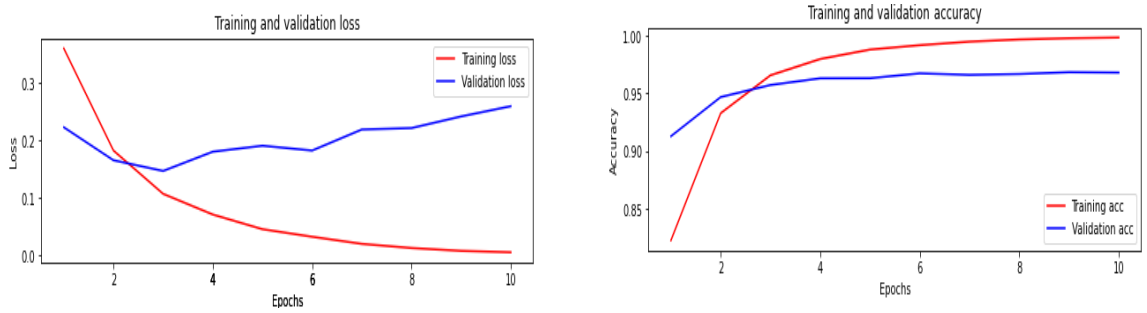


Figure 26: Architecture of the BERT Model

To build the model, I used the original un-processed dataset. I only removed unique characters and any HTML/XML tags. BERT is capable of handling minimally data which can be useful while training the model to perceive the context of the sentence. I ran the model for 10 epochs using Google Colab Pro which allows for uninterrupted run times that can handle long runs. **Figure 27** shows the loss and accuracy results for the BERT model.



(a) BERT Model Loss (b) BERT Model Acc

Figure 27: BERT Model Performance

The BERT model produced the best results out of all the models I built. It had an accuracy of 99% on the training set with a loss of 0.1, and an accuracy of 95% on the 20% testing set with 0.36 loss. **Figure 28** shows the confusion matrix of the model on the testing set.

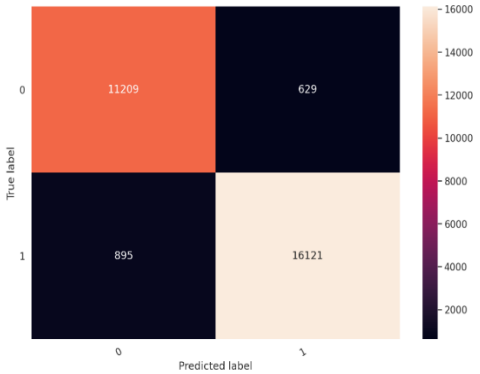


Figure 28: BERT Model Confusion Matrix

5.3.1 BERT Application

I saved the model from BERT run since it produced the best results. I then called the model in the jupyter notebook and allowed for user input. The user input can then be scored and assigned a class. The figure below shows two examples that were tested using the model.

```
examples = [
    input("Enter a review:")
]

reloaded_results = tf.sigmoid(reloaded_model(tf.constant(examples)))
original_results = tf.sigmoid(classifier_model(tf.constant(examples)))

print('Results from the saved model:')
print_my_examples(examples, reloaded_results)
print('Results from the model in memory:')
print_my_examples(examples, original_results)
```

```
Enter a review:I do not recommend this product.
Results from the saved model:
input: I do not recommend this product. : score: 0.000031

Results from the model in memory:
input: I do not recommend this product. : score: 0.000031
```

(a) User input - Negative

```
examples = [
    input()
]

reloaded_results = tf.sigmoid(reloaded_model(tf.constant(examples)))
original_results = tf.sigmoid(classifier_model(tf.constant(examples)))

print('Results from the saved model:')
print_my_examples(examples, reloaded_results)
print('Results from the model in memory:')
print_my_examples(examples, original_results)
```

```
I recommend this drug. It has no side effects and it works well.
Results from the saved model:
input: I recommend this drug. It has no side effects and it works well. : score: 0.999720

Results from the model in memory:
input: I recommend this drug. It has no side effects and it works well. : score: 0.999720
```

(b) User input - Positive

Figure 29: User Input Examples

This application can be further refined and added into a GUI that can assess any new user review.

6 Conclusion & Further Analysis

This project demonstrated my first hands-on project on advanced NLP models. I used a relatively understudied dataset on drug reviews and created multiple models using different techniques to classify the text predictions. I began by exploring the dataset, cleaning the text, and preprocessing the data to convert it into a usable input. I established five baseline classification models using the TF-IDF tokenization method: Naïve Bayes, Support Vector Machines, XGBoost, Gradient Boosting Classifier, and XGBoost. The logistic regression model was highly accurate despite the nature of the dataset. I moved to build advanced deep learning models: a bidirectional LSTM, a CNN with LSTM layer, a pooled GRU, and a Hierarchical Attention Network using three different word embeddings: learn-from-scratch, GloVe, and fastText. Overall the CNN-LSTM model produced the best results with the highest performance efficiency. The learn-from-scratch and GloVe embeddings were equally good embeddings but the fastText embedding was more memory-intensive yet less accurate than the other embeddings. This can be attributed to the high dimensionality of the embeddings. It may also be due to the nature of the embedding's architecture.

I also built an advanced BERT model which produced the best accuracy results. This model used the state of the art Google-developed BERT. While it was computationally expensive, it was a successful model and a good learning experience. I attempted to use two other transfer learning methods: ULMFit, and Flair, however, the computing capability that was available to us did not permit.

For future analysis, I hope to apply additional BERT models such as ALBERT, and DistilBERT which combine efficiency and accuracy and produce good results. Furthermore, I would like to expand the model to include multiple classes (sentiments), perhaps a 1 - 5 or a 1 - 10 score. This is a more challenging problem but running the BERT model may produce good results. Lastly, I would like to incorporate more features into the classification such as the drug name and the condition it treats. This was not possible in this dataset due to the imbalance in the drugs covered and the conditions mentioned. With a better dataset, these studies can be achieved.

7 Appendix

7.1 Appendix 1: Word2Vec t-SNE Visualization

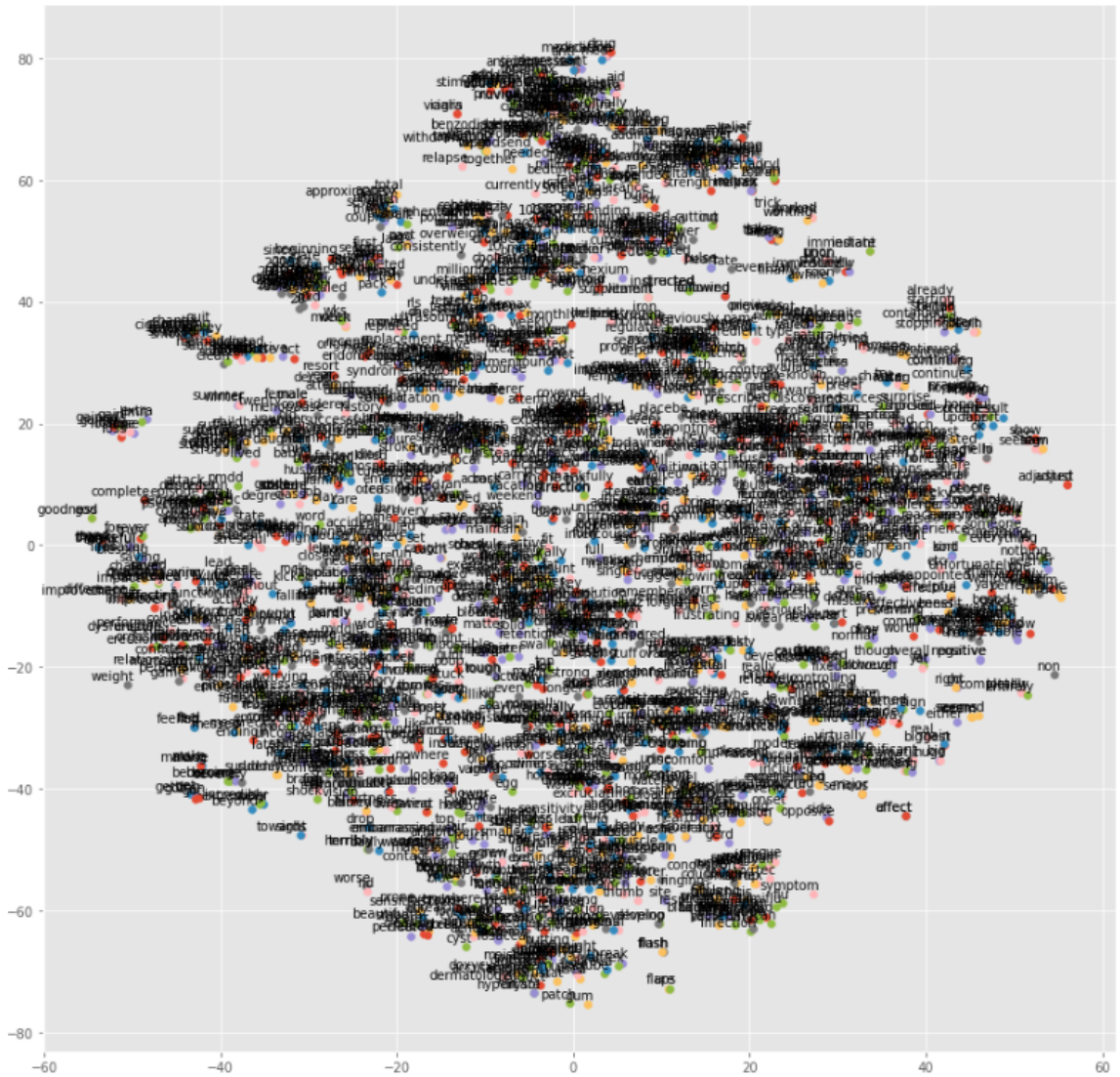


Figure 30: Word2Vec t-SNE Visualization

7.2 Appendix 2: (Bag of Words) Single Value Decomposition Plot using CountVectorizer

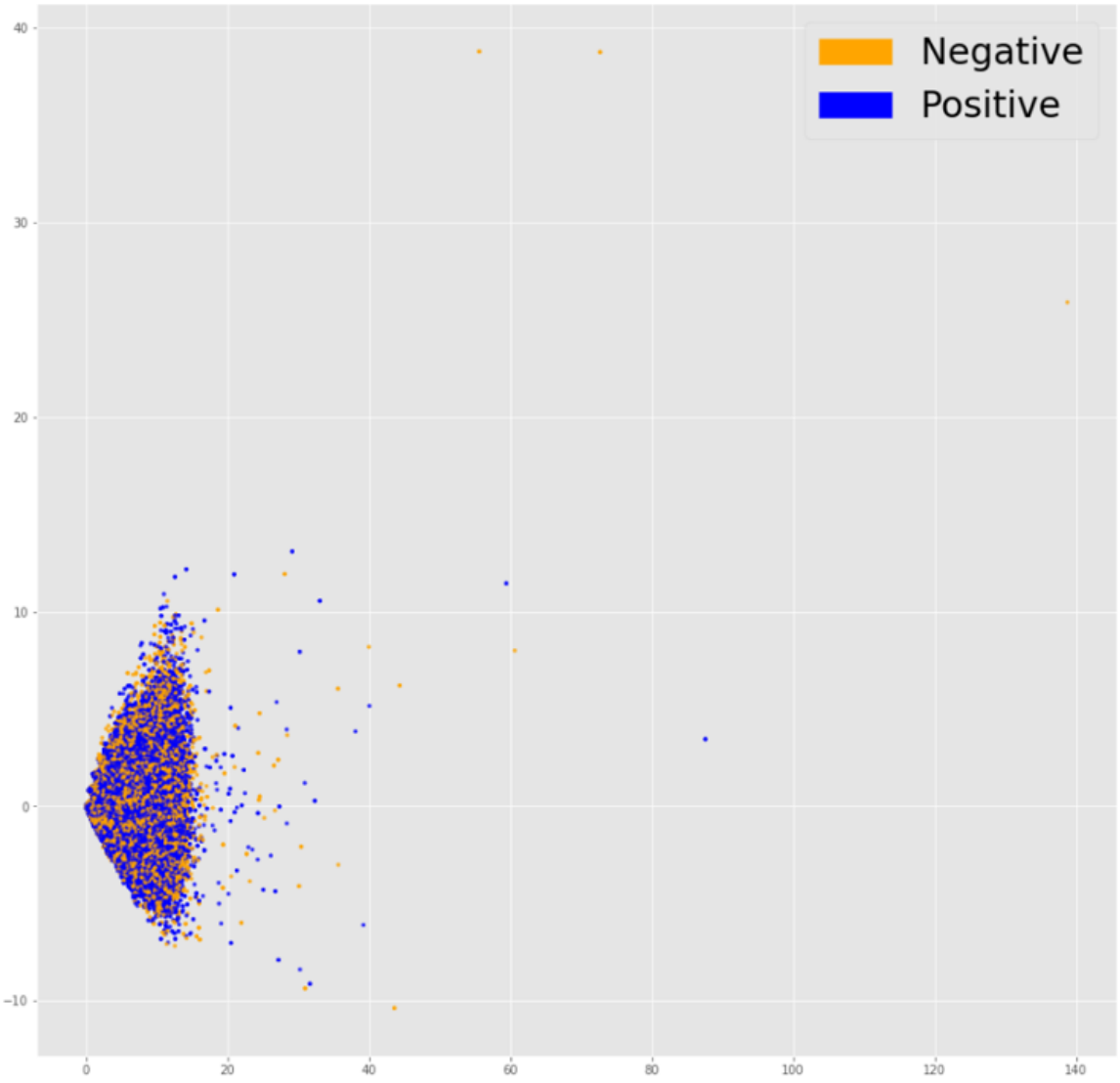


Figure 31: SVD CV Visualization

7.3 Appendix 3: (Bag of Words) Single Value Decomposition Plot using TF-IDF

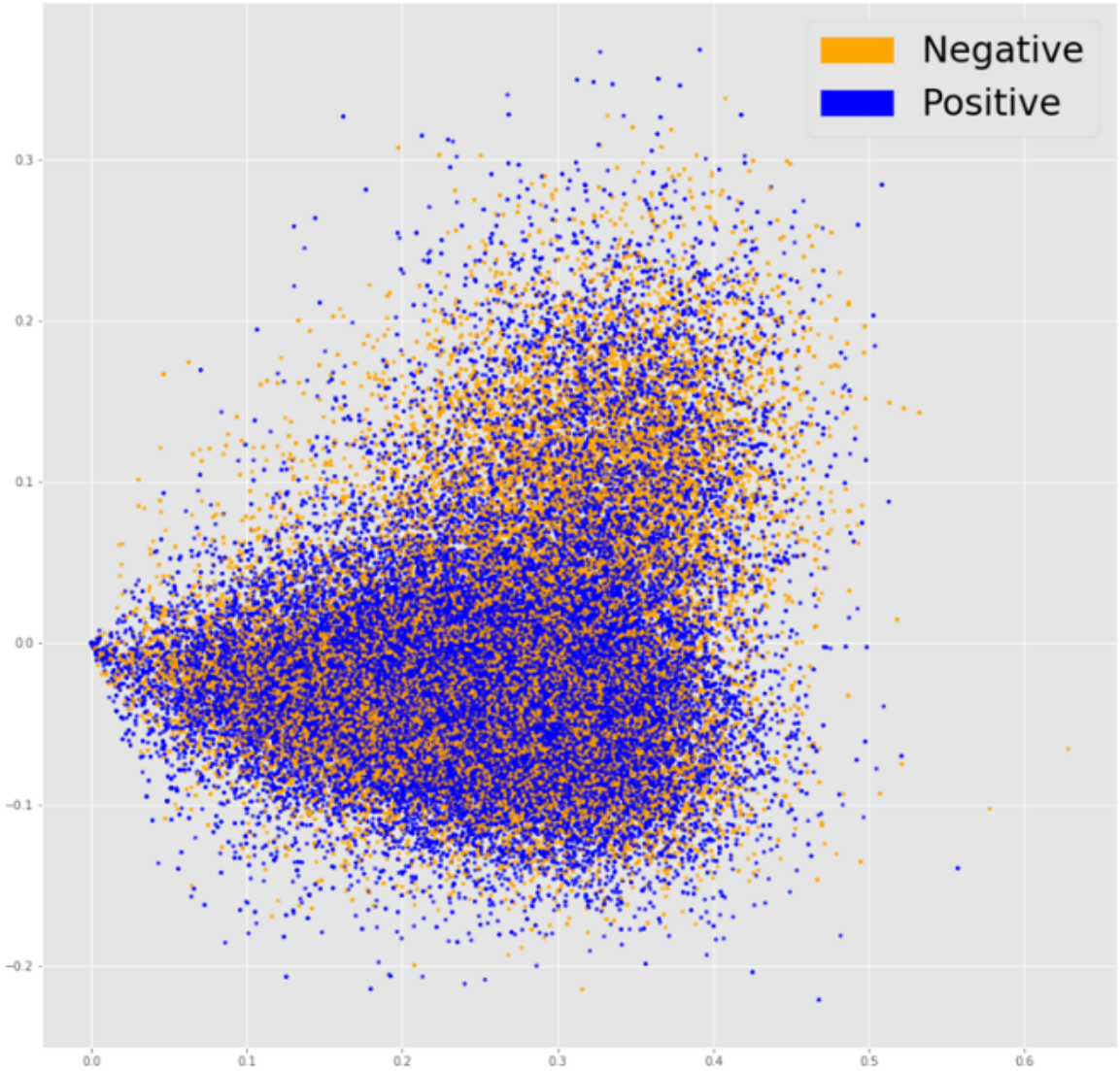


Figure 32: SVD TF-IDF Visualization

7.4 Appendix 4: Countvectorizer Plot using UMAP Features

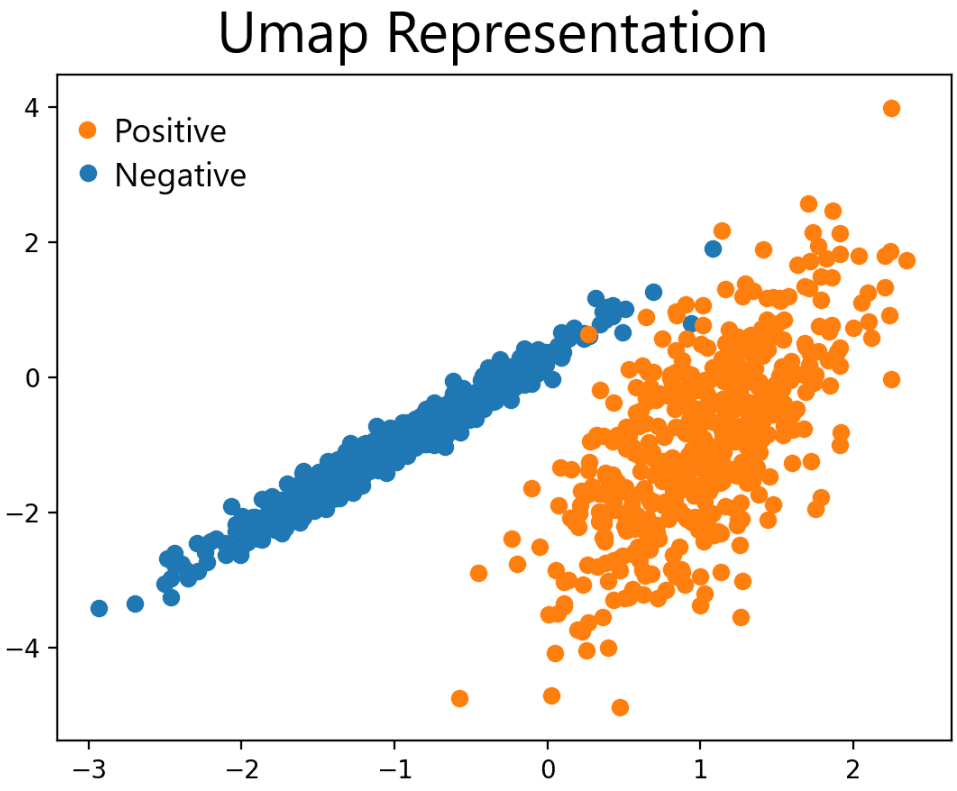


Figure 33: Countvectorizer Plot using UMAP Features

7.5 Appendix 5: BERT Transformer Architecture

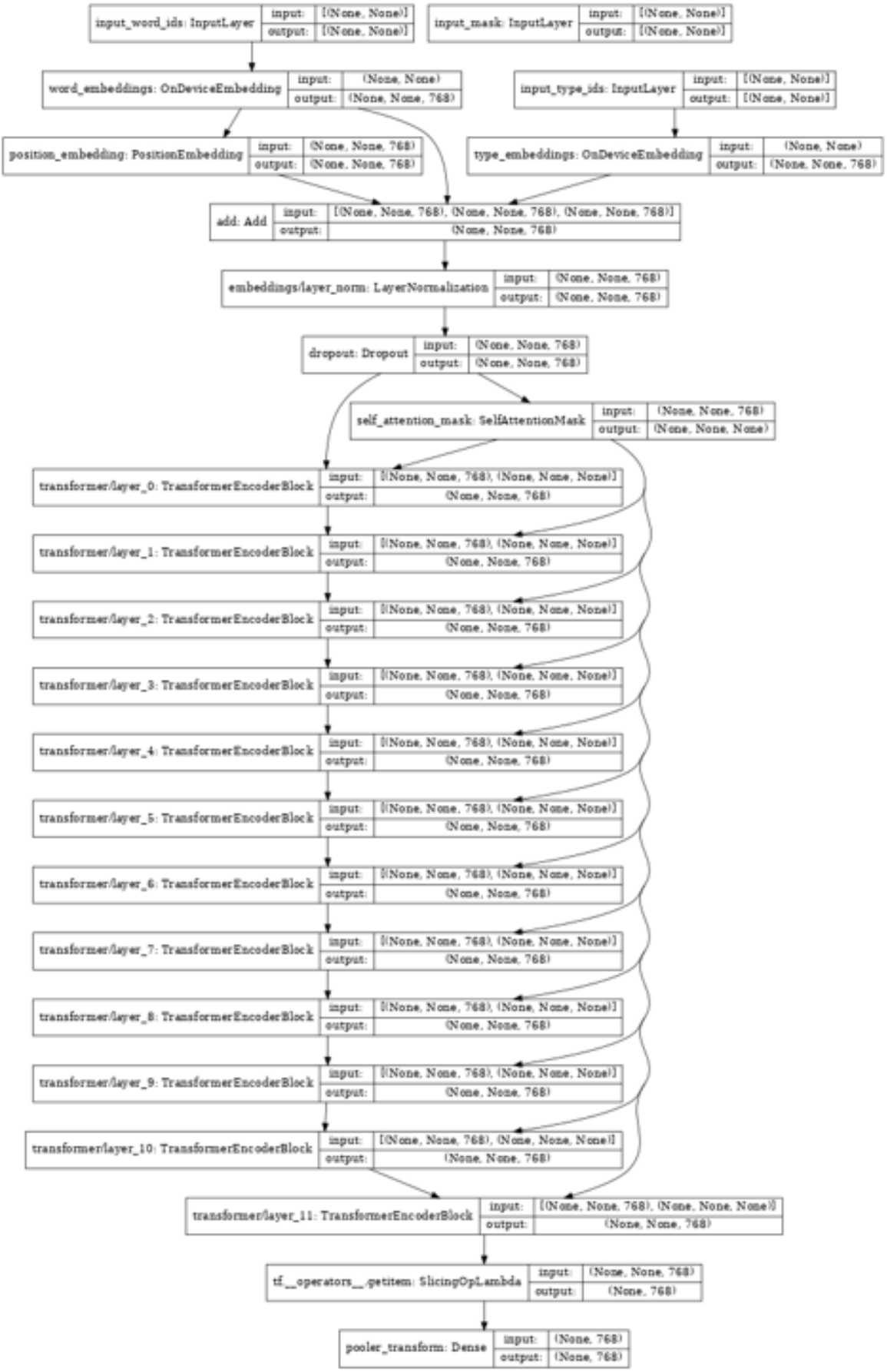


Figure 34: BERT Transformer Architecture

References

[1] Aaron Kub. *Sentiment Analysis with Python.*
https://bit.ly/2W5XsmT.

[2] Arun Maiya. *ktrain: A Lightweight Wrapper for Keras to Help Train Neural Networks.*
https://bit.ly/3eVvpiG.

[3] Arun Maiya. *Text Classification with Hugging Face Transformers in TensorFlow 2.*
https://bit.ly/3bIUlIp.

[4] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.*
https://arxiv.org/pdf/1810.04805.pdf.

[5] Ashok Chilakapati. *Word Bags vs Word Sequences for Text Classification.*
https://bit.ly/2VHXBOA.

[6] Diego Lopez Yse. *Your Guide to Natural Language Processing (NLP).*
https://bit.ly/3cNls4Y.

[7] Edward Ma. *Interpreting your deep learning model by SHAP.*
https://bit.ly/2xUjgu0.

[8] Heet Sankesara. *Hierarchical Attention Networks.*
https://bit.ly/35aJ6G1.

[9] Himanshu Kriplani. *Understanding language modeling(NLP) and Using ULMFIT.*
https://bit.ly/3bI77XJ.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova. *Attention Is All You Need.*
https://arxiv.org/abs/1706.03762.

[11] Jason Brownlee. *How to Use Word Embedding Layers for Deep Learning with Keras.*
https://bit.ly/35aRrJS.

[12] Javaid Nabi. *Machine Learning — Text Processing.*
https://bit.ly/2VIiN7i.

[13] Julien Chaumond. *State-of-the-art Natural Language Processing for TensorFlow 2.0 and PyTorch.*
https://bit.ly/2Sdu1yb.

[14] Karan Bhanot. *Different techniques to represent words as vectors (Word Embeddings).*
https://bit.ly/2SeDsgO.

[15] Marco Tulio Correia Ribeiro . *Lime: Explaining the predictions of any machine learning classifier.*
https://github.com/marcotcr/lime.

[16] Maria Kränkel, Hee-Eun Lee. *Text Classification with Hierarchical Attention Networks.*
https://bit.ly/2yPsOGq.

[17] Michael Galarnyk. *PCA using Python (scikit-learn).*
https://bit.ly/2XsRFdz

[18] Miguel Fernández Zafra. *Text Classification in Python.*
https://bit.ly/2Sg480W.

[19] Nagesh Singh Chauhan. *A beginner's guide to Linear Regression in Python with Scikit-Learn.*
https://bit.ly/3a8y7y0

[20] Nikolai Janakiev *Practical Text Classification With Python and Keras.*
https://realpython.com/python-keras-text-classification/.

[21] Rachel Koeing. *NLP for Beginners: Cleaning & Preprocessing Text Data.*
https://bit.ly/2VHa8lr.

[22] Radu Soricut and Zhenzhong Lan (Google Research). *ALBERT: A Lite BERT for Self-Supervised Learning of Language Representations.*
https://bit.ly/3eZ6rPt.

[23] Rani Horev. *BERT Explained: State of the art language model for NLP.*
https://bit.ly/2ScjNyd.

[24] Robert Tibshirani, Trevor Hastie, Gareth James, and Daniela Witten. *An Introduction to Statistical Learning.* Springer, New York, New York, 2013.

[25] Sabber Ahamed. *[NLP] Performance of Different Word Embeddings on Text Classification.*
https://bit.ly/2VI4Sy0.

[26] Sabber Ahamed. *Text Classification Using CNN, LSTM and visualize Word Embeddings.*
https://bit.ly/2KDQb8L.

[27] Sabber Ahamed. *Text Classification Using CNN, LSTM and visualize Word Embeddings.*
https://bit.ly/35hzJ7K.

[28] Scott Lundberg. *SHAP: A game theoretic approach to explain the output of any machine learning model..*
https://github.com/slundberg/shap.

[29] Shervin Minaee, Nal Kalchbrenner, Erik Cambria, Narjes Nikzad, Meysam Chenaghlu, Jianfeng Gao. *Deep Learning Based Text Classification: A Comprehensive Review.*
https://arxiv.org/pdf/2004.03705.pdf.

[30] Simon Haykin. *Neural Networks and Learning Machines.* Pearson, Upper Saddle River, New Jersey, 2019.

[31] Steve Mutuvi. *Using Transfer Learning and Pre-trained Language Models to Classify Spam.*
https://bit.ly/357OdHd.

[32] Susan Li. *Explain NLP models with LIME & SHAP.*
https://bit.ly/2SbWCns.

[33] Susan Li. *Multi-Class Text Classification Model Comparison and Selection.*
https://bit.ly/3aCUk7q.

[34] Tadej Magajna. *Text Classification with State of the Art NLP Library — Flair.*
https://bit.ly/3bJkTt4.

[35] Thilina Rajapakse. *A Simple Guide On Using BERT for Binary Text Classification.*
https://bit.ly/2YbJJxQ.

[36] Thu Dinh, Oklahoma State University. *Detecting Side Effects and Evaluating Effectiveness of Drugs from Customers' Online Reviews using Text Analytics and Data Mining Models.*

[37] Tom Lin. *Text Classification Using CNN, LSTM and visualize Word Embeddings.*
https://bit.ly/2KDQb8L.

[38] Tom Young, Devamanyu Hazarika, Soujanya Poria Erik, Cambria. *Recent Trends in Deep Learning Based Natural Language Processing.*

[39] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning.* Springer, New York, New York, 2008.

[40] Unnamed Author (Stack Overflow Research). *Predicting Stack Overflow Tags with Google's Cloud AI.*
https://bit.ly/3eTOTnX.

[41] Venelin Valkov. *Intent Recognition with BERT using Keras and TensorFlow 2.*
https://bit.ly/3eW3gYZ.

[42] Victor Sanh. *Smaller, faster, cheaper, lighter: Introducing DistilBERT, a distilled version of BERT.*
https://bit.ly/358AqjH.

[43] Will Koehrsen. *How to Visualize a Decision Tree from a Random Forest in Python using Scikit-Learn.*
https://bit.ly/2ww5oFl

[44] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, Eduard Hovy. *Hierarchical Attention Networks for Document Classification.*
https://www.cs.cmu.edu/ ./hovy/papers/16HLT-hierarchical-attention-networks.pdf.

[45] Qiang Ye, Ziqiong Zhang, Rob Law. *Sentiment classification of online reviews to travel destinations by supervised machine learning approaches.*
https://www.sciencedirect.com/science/article/abs/pii/S0957417408005022.

[46] Binxuan Huang, Yanglan Ou, Kathleen M. Carley. *Aspect Level Sentiment Classification with Attention-over-Attention Neural Networks.* https://bit.ly/3u3TXNR2.

[47] Guozheng Rao, Weihang Huang, Zhiyong Feng, Qiong Cong. *LSTM with sentence representations for document-level sentiment classification.*
https://bit.ly/3dXmv5Y.

[48] Manish Munikar, Sushil Shakya, Aakash Shrestha. *Fine-grained Sentiment Classification using BERT.*
https://ieeexplore.ieee.org/abstract/document/8947435.

[49] Prathusha K Sarma, YIngyu Liang, William A Sethares. *Domain Adapted Word Embeddings for Improved Sentiment Classification .*
https://arxiv.org/abs/1805.04576.

[50] Yangsen Zhang, Jia Zheng, Yuru Jiang, Gaijuan Huang, Ruoyu Chen. *A Text Sentiment Classification Modeling Method Based on Coordinated CNN-LSTM-Attention Model.*
https://digital-library.theiet.org/content/journals/10.1049/cje.2018.11.004.