SINGAPORE MANAGEMENT UNIVERSITY

IS460: Machine Learning & Applications

Final Project Report

Prepared For: Dr WANG Zhaoxia

Team Name	G1T2
Team Members	Evelyn PEH Ting Yu
	KOH Pei Ling
	LAU Wei Ting
	LUQMAN Juzaili Bin Muhammad Najib
	LYE Jian Yi
	SONG Yu Xiang

1. Project Title **1**
2. Introduction **1**
3. Motivation **1**
4. Literature Reviews **2**
4.1 Literature Review #1 - Sadfishing of #JusticeforAudrey (Hashtag) on Twitter 2
4.2 Literature Review #2 - A textual-based featuring approach for depression detection using machine learning classifiers and social media texts 3
4.3 Literature Review #3 - Applying Deep Learning Technique for Depression Classification in Social Media Text (Using Twitter Dataset) 7
4.4 Literature Review #4 - Assessing Emoji use in Modern Text Processing Tools 8
4.5 Literature Review #5 - A Comparative Study on Word Embeddings in Deep Learning for Text Classification 12
5. Tools and Resources Used **13**
6. Datasets **15**
6.1 Source of Data 15
6.1.1 Sentiment Analysis - Selected 15
6.1.2 Twitter Dataset (Social Media) 15
6.2 Exploratory Data Analysis 16
6.2.1 Meta Data of our Dataset 16
6.2.2 Dataset Analysis 17
6.3 Dataset Preprocessing 17
7. Methodology **26**
7.1 Natural Language Processing Background 26
7.2 Traditional Classifiers 26
7.3 Deep Learning Classifiers 27
7.4 Classification Metrics 27
7.5 General Process Flow for Traditional and Deep Learning Models 28
8. Word2Vec Natural Language Processing 30
8.1 Word2Vec Word Embeddings 30
8.2 Word2Vec Traditional Classification Results 31
8.3 Word2Vec Deep Learning Classification Results 32
8.4 Conclusion of Word2Vec 37
9. Embeddings from Language Models (ELMo) Natural Language Processing **38**
9.1 ELMo Word Embeddings 38
9.2 ELMo Traditional Classifiers Results 40
9.3 ELMo Deep Learning Implementation & Results 41
9.4 Conclusion of ELMo 50
10. Generative pre-trained transformer (GPT) 51
10.1 GPT Word Embeddings 51
10.2 GPT Traditional Classifiers Results 52
10.3 GPT Deep Learning Implementation & Results 52
10.4 Conclusion of GPT 64
11. Bidirectional Encoder Representations from Transformers (BERT) 65

11.1 BERT Word Embeddings 65
11.2 BERT Traditional Classifiers Results 67
11.3 BERT Deep Learning Implementation & Results 67
11.4 Conclusion of BERT 77
12. Results and Discussions **77**
13. Conclusion and Future Works **79**
14. References **80**

1. Project Title

Utilising NLP and Deep Learning to Detect Depression in Social Media Texts

2. Introduction

This project leverages social media (Eg, Twitter) datasets to predict if an individual is experiencing depression and to take prompt intervention measures such as therapy and medication control to help prevent the disorder from worsening and potentially leading to suicide.

3. Motivation

According to the Samaritans of Singapore (SOS), the number of suicide cases among youths aged 10 to 29 has remained consistently high in recent years. SOS has observed a rise from 22.3% in 2020 to 29.6% in 2021 in all suicides recorded nationwide. For youths aged 10 to 19, suicide has increased by 23.3%, from 30 deaths in 2020 to 37 in 2021. This upward trend of young people taking their own lives is a major concern. Common stressors experienced by these young people include interpersonal issues with family and friends, academic and achievement stress, and future job stress.

Studies have shown that mental disorders such as depression increase the risk of suicide compared to people without depression. Those living with depression often experience negative thoughts and overwhelming emotions that can become unbearable, leading them to feel like taking their own life is the only way to end the pain. With the lack of someone to express their feelings to and social media playing a large role in their lives, these young people tend to vent their problems on social media. A survey conducted found that depressed teens tend to turn to social media to cope. Many teens and young adults use social media as an alternative to therapy, not so much for receiving feedback but rather as a way to vent. In another article, "How 'Sadfishing' Became Our Favourite Coping Mechanism," a 22-year-old shared that whenever she faces a problem, she looks to Twitter to express her feelings rather than share with someone else.

As people tend to share their thoughts on social media platforms, social data contains valuable information that can be used to identify a user's psychological state. Analysing social media posts to predict the possibility of depression provides psychiatrists and psychologists with additional information before making decisions and opens up possibilities for early detection by using data obtained from the subject's social media platform. Depression is episodic, where individuals diagnosed with depression typically have multiple episodes. Therefore, this project aims to analyze social media posts for the early detection of depression, which can reduce the tragedy of suicide. (SOS,2022)

4. Literature Reviews

4.1 Literature Review #1 - Sadfishing of #JusticeforAudrey (Hashtag) on Twitter

This research study analyses the phenomenon of "sadfishing," a trending phenomenon used to describe the act of making exaggerated statements about one's emotional problems to gain sympathy. The term "sadfishing" is commonly used to refer to emotional posts expressed by people, especially teenagers, who seek support by discussing issues around anxiety and depression on social media. It is a way for someone to express their vulnerability and psychological condition through social media.

This phenomenon is common among teenagers worldwide, who use social media platforms as an avenue to confide in their personal problems. They may do so for attention or because therapy is inaccessible because they do not have an alternate outlet to express themselves, such as friends or family who understand what they are going through. In these scenarios, posting on social media becomes their therapy, a way to feel heard and understood and not bottle up their feelings.

With this ongoing trending phenomenon, where people increasingly turn to social media platforms such as Twitter, Reddit, and Facebook to express their opinions, communicate with others, and share their feelings, social media information greatly identifies people at risk of depression or other mental disorders.

Therefore, our project will analyse social media text, specifically posts from Twitter, to identify the possibility of depression in the user.

4.2 Literature Review #2 - A textual-based featuring approach for depression detection using machine learning classifiers and social media texts

This study was conducted with the aim of determining whether machine learning could be effectively used to detect signs of depression in social media users by analysing their social media posts, especially when those messages do not explicitly contain specific keywords such as ‘depression’ or ‘diagnosis’.The author uses two labelled Twitter datasets (Eye, Shen et al.) to train and test 8 machine learning models (Logistic Regression, Support Vector Machine, Multilayer Perceptron, Decision tree, Random Forest, Adaptive Boosting, Bagging Predictors and Gradient Boosting.) and thereafter tested the model against 3 other datasets (sourced from Facebook, Reddit, and an electronic diary) which contained only depression posts.

By utilising social media to detect depression, it is likely that these data are unstructured. Hence, Machine Learning is a good technique for dealing with such nonlinearity, a better option than traditional statistical methods. In this study, the features used for training were preprocessed and extracted by utilising various methods that were proven to have performed well in other studies.

Text Preprocessing Methods	- Tokenization - Stop Words Removal - Removal of Punctuation, Number, Website Links
Word Correction Techniques	- **Spelling Error Correction:** Utilised Peter Norvig’s code for spelling correction based on probability theory (https://norvig.com/spell-correct.html) - **Elongated Word Correction:** Words such as “yesss”, “fiiine”, “youuuu” were reformed back to its original form using the Peter Norvig’s code. - **Negative Word Correction:** Common goal is to introduce negation in the sentence, and there were reduced to their basic negative form, “not”
Part of Speech Tagging and Lemmetization	- **POS Tagging:** Assigning words to their syntactic functions, such as nouns, pronoun, adjective, verb, and adverb, putting the words in the context. - **Lemmatization:** Reducing a word to the correct context; E.g. Changing the word back to its basic form, which is an essential step in reducing word diversity and making recognition easier.
Bag of Words Feature	- Extraction combined with count vectorisation and n-gram words

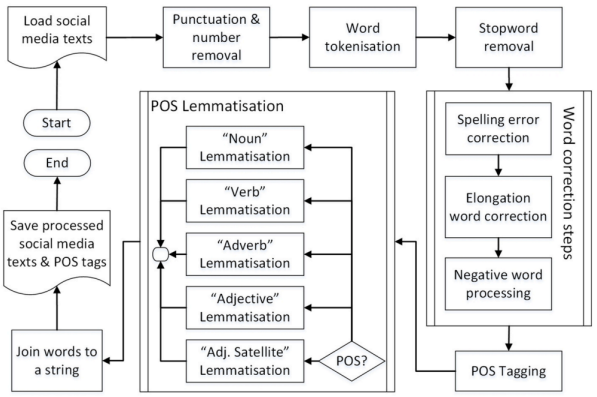


Figure 1: Process of Natural Language Processing

After processing the text, the author moved on to train 8 classifiers consisting of both single and ensemble models.

Single Classifier

Table 4
Results of the preliminary experiments.

Dataset	Classifier	Overall Measurement (%)				Detailed Accuracy (%)		Accuracy on Depression-class-only Datasets (%)		
		Acc	Prec	Rec[c]	F1	Dep[a,c]	Non-Dep[a]	Tan[b]	Koma[b]	Vira[b]
Eye	LR	99.80	100	99.09	99.54	99.09	100	1.61	40.54	16.30
	LSVM	99.78	99.87	99.14	99.50	99.14	99.96	1.77	30.32	18.20
	MLP	99.12	99.78	96.27	97.99	96.27	99.94	9.19	31.32	31.24
	DT	99.74	99.57	99.26	99.41	99.26	99.88	13.39	48.59	31.35
Shen et al.	LR	99.77	100	99.58	99.79	99.58	100	0	16.76	13.86
	LSVM	99.80	99.94	99.68	99.81	99.68	99.93	0.32	12.09	13.67
	MLP	99.57	99.94	99.26	99.60	99.26	99.93	0	4.73	13.35
	DT	99.71	99.96	99.51	99.73	99.51	99.95	6.45	49.86	25.86

[a] Dep = Depression; Non-Dep = Non-Depression.
[b] Tan = Tanwar's dataset; Koma = Komati's dataset; Vira = Virahonda's dataset; these three datasets are single-class datasets comprised only of depression records.
[c] Even though binary recall and accuracy in detecting positive (depression) classes will have the same values, both are presented here to show differences in the accuracy for depression and non-depression classes.

Table 5
Experimental results after the words 'depression' and 'diagnose' were deleted from the datasets.

Dataset	Classifier	Overall Measurement (%)				Detailed Accuracy (%)		Accuracy on Depression-class-only Datasets (%)		
		Acc	Prec	Rec[c]	F1	Dep[a,c]	Non-Dep[a]	Tan[b]	Koma[b]	Vira[b]
Eye	LR	92.61	93.32	72.21	81.38	72.21	98.51	69.19	90.40	69.65
	LSVM	91.57	84.84	76.04	80.17	76.04	96.07	60.00	85.92	70.44
	MLP	89.56	77.77	75.02	76.29	75.02	93.77	53.39	75.62	68.13
	DT	86.00	68.24	70.52	69.31	70.52	90.47	59.84	75.26	66.32
Shen et al.	LR	88.62	92.63	86.03	89.20	86.03	91.75	27.42	46.17	66.43
	LSVM	87.38	90.01	86.51	88.22	86.51	88.43	30.65	51.73	68.99
	MLP	85.13	86.65	86.07	86.35	86.07	84.01	28.39	52.70	64.30
	DT	82.23	83.46	84.20	83.82	84.20	79.86	47.90	52.62	66.93

Figure 2: Results Before and after removal of "Depression" and "Diagnose"

The single classifiers were trained on 2 separate scenarios - (1) with the original dataset and (2) with the words "depression" and "diagnose" deleted from the dataset.

A comparison of the results can be seen in the image above. We can see a reduction in the overall measurements of all classifiers. However, the detection accuracy significantly improved when the trained models were run on the other three depression-class-only datasets (Tanwar's, Komati's and Virahonda's datasets), with the improvement ranging from 27.42% to 67.58%.

These results imply that the models trained using the modified datasets are more general and can better detect depression in the text from datasets they were not trained on.

Imbalanced Dataset

It is worth noting that one of the datasets (Eye) used in the study is heavily imbalanced which could affect the prediction result. To overcome the problem of imbalanced data, the author modified the original dataset by oversampling the minority class of the training sets and undersample the majority class of the training sets. Both over and under-sampling improved the detection of the depression class. The rates of improvement in the case of over-sampling were 0.6% with the MLP, 1.5% with the LSVM, 8.7% with the LR, and 4.6% with the DT; whereas for under-sampling, they were 8% with the MLP, 6.4% with the LSVM, 10.4% with the LR, and 12.5% with the DT.

The detection of the non-depression class, however, worsened. From these results, we can conclude that in a heavily imbalanced dataset, dynamic sampling could increase the accuracy of the less populous class but, at the same time, decrease the accuracy of the more populous class. However, this can be beneficial if the goal is to detect depression and the less populous class is the depression class, which was the case in this study.

Ensemble Classifiers

Among the ensemble models, Random Forest provided the most balanced accuracy for depression and non-depression classes in both Twitter datasets. However, these results were not better than those provided by the single classifier Logistic Regression. The author guessed that such a result might be due to the choice of the base classifiers used by the ensemble models, which in this case is the Decision Tree. Furthermore, it yielded the worst result when comparing DT to the other single classifiers.

On the other hand, when tested on the three depression-class-only datasets, RF provided better results than all single and ensemble classifiers. Therefore, we can conclude that the RF model is better at depression detection using general texts than other classifiers.

Final Takeaways from Literature Review 2

Some takeaways from this study that can be applied to our project:

1. Since the preprocessing steps adopted in these studies have been proven to work effectively in other studies, it can also be served as a reference. Word correction techniques can also be explored because social media text often contains elongated work to emphasise emotion,
2. Ensemble learning methods can be adopted. However, it is important to choose the correct single classifier. One that perhaps achieves the highest accuracy.
3. In the case of having imbalanced datasets, adopting a method to balance it could improve the accuracy of detecting depression class which is definitely more important than not being able to detect non-depression class as the former could result in more serve consequences such as suicide.

4. One avenue that our project could also experiment with is to train our model on the Twitter dataset and see how well they can identify depression on other datasets from other social media platforms such as Facebook, Reddit as mentioned under Literature Review 1, that people share their feelings on those platforms, other than Twitter, too.

4.3 Literature Review #3 - Applying Deep Learning Technique for Depression Classification in Social Media Text (Using Twitter Dataset)

This literature review presents the authors, namely, Hussain Ahmad, Fahad M, Ibrahim Hameed and M.Zubair trying to use deep learning methodologies to classify Twitter tweets into "Normal" or "Depressed".

The literature review also states how several deep learning models have been applied, namely RNN, CNN, LSTM, GRU and BiLSTM, to classify a tweet as "normal" or "Depressed.

Table V. Comparative results of proposed model with other deep learning models.

Model	Accuracy	Precision	Recall	*F*1-Score
RNN	82.289%	0.83	0.82	0.83
CNN	0.87	0.84	0.86	0.82
LSTM	79.001%	0.81	0.77	0.79
GRU	0.84	0.81	0.84	0.83
Bi-LSTM	0.93	0.89	0.91	0.90

Figure 3: Comparative results of Proposed model with other Deep Learning models

The figure above shows that the Bi-LSTM model for depression classification outperformed similar Deep Learning Models.

Deep Learning, most notably Long Short-Term Memory (LSTM), is able to capture both semantic and syntactic information present in datasets. However, it is only able to preserve past information and is unable to utilise the future context and other models, like simple RNN (Recurrent Neural Network), can only retain contextual information for a short period of time.

Nevertheless, using BiLSTM, both past and future context can be considered, given an input text, with 2 hidden layers. The following briefly explains how BiLSTM works.

1) With a sequence of words w = [w1,w2,w3,w4….,wN] where N is the number of words, the BiLSTM layer calculates the forward and backwards hidden vectors.

2) Proceed to generate an output vector, and a tweet is classified into two classes: "normal" or "depressed."

3) Further tuning of parameters such as Vocab_size range, Batch Size range, BiLSTM unit range

The following image illustrates the general flow of how Bi-LSTM works.

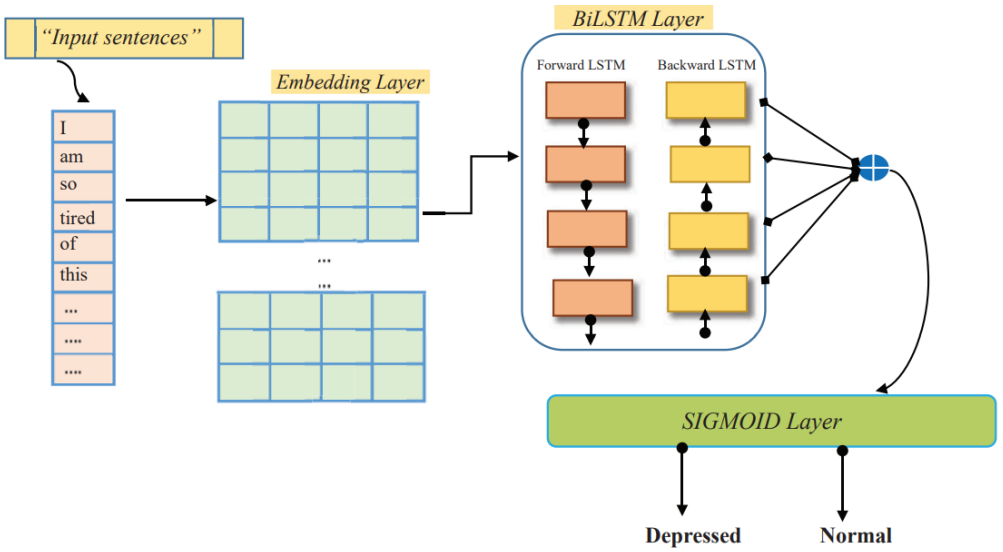


Figure 4: Process of Bi-LSTM

4.4 Literature Review #4 - Assessing Emoji use in Modern Text Processing Tools

In recent years, Emojis have been widely used to convey emotions or further enhance expression while communicating in textual messages. The use of emojis could potentially contribute to the overall sentiment of a message (e.g., positive, neutral, or negative). Although not covered in this paper, the use of emojis could appear in social media posts and might be a contributing factor in determining whether a post could exhibit signs of depression. Hence, exploring the different text processing tools in the study could be useful in pre-processing datasets.

More specifically, the different use cases of emojis are analysed in the study, and the aim is to find out which NLP tools are able to perform NLP tasks on the emojis without losing their meaning. Through this study, we can single out the NLP tools that can deal with emojis in text well.

The different use cases of emojis extend from exploring the different aspects of emoji use, such as the number of emojis, the position of emojis, the use of skin tone modifiers, etc.

Different Use Cases of Emojis

Case 1
Single Emojis
	- **Single Emojis with space**
	  - E.g. Emojis 🙂 are a new way of expressing emotions!
	- **Single Emojis without space**
	  - E.g. Emojis🙂are a new way of expressing emotions!

Case 2
Multiple Emojis
	- **Multi Emoji Multi Positions**
	  - E.g. Emojis 🙂 are a new way of expressing emotions! 😀
	- **Multi Emoji with Space**
	  - E.g. Another example is having multiple emojis 🙂 🙂 🙂 together in a tweet
	- **Multi Emoji Cluster**
	  - E.g. Another example is having multiple emojis 🙂🙂🙂 together in a tweet without having any spaces in between emojis.

Case 3 **Emojis with Skin Tone Modifiers**	- **Using the same emoji but in different skin tones** - E.g. Thumbs up with a light skin tone (👍🏻) / medium-light skin tone (👍🏼) / medium skin tone (👍🏽), etc.

Case 4 and 5
Basic and Supplemental Planes (BMP)
	- Depending on the emoji's characteristics, it could be encoded in Plane 0 or 1 (a group of code points in the Unicode standard).
	  - **Emojis encoded in Plane 0 (BMP)**
	  - Maximally require 16 bits to be encoded
	  - **Emojis in Plane 1 (Non-BMP)**
	  - Requires beyond 16 bits to be encoded
	- Need to find out whether the NLP tools can handle both types of emojis

Case 6 **Zero Width Joiner (ZWJ) Emojis**	- Ensure group emojis such as the family emoji (👩‍👩‍👧) are parsed as a single unit emoji rather than a combination of 4 emojis.

NLP Tools Used

AllenNLP	**Gensim**	**NLTK**
NLTK-TT	**PyNLP1**	**SpaCy**
SpaCtMoji	**Stanza**	**Textblob**

Retaining Emojis Semantics in NLP Tasks

Tokenization
The act of breaking a sequence of strings to words, keywords, phrases, symbols, etc.

NLP tools are expected to:

- Distinguish emoji from a word (e.g., treat emoji as a token)
- Distinguish emojis in the cluster (e.g., able to treat 3 emojis consecutively as 3 separate tokens)
- Distinguish emoji in a different skin tone as one emoji (e.g., "waving hand light skin emoji" should be split into "waving hand emoji" and "light skin modifier")
- Distinguish ZWJ emojis as a token and not a group of tokens

NLP Tools Performance:

	Task - Tokenization					
Tools	**Case 1 Single Emoji**	**Case 2 Multi Emoji**	**Case 3 Skin Tone Emoji**	**Case 4 BMP Plane 0**	**Case 5 non-BMP Other Planes**	**Case 6 Zero Width Joiner**
Gensim	0	0	0	0	0	0
NLTK	80	0	68	40	70	70
NLTK-TT	70	70	0	80	60	0
PyNLPI	50	0	38	30	20	70
SpaCy	100	100	0	100	100	0
SpaCyMoji	100	100	92	100	100	0
Stanza	90	10	68	50	90	40
TextBlob	80	0	68	40	70	70

Table 2: The percentage of success of tools covering all different cases of emojis in tokenization

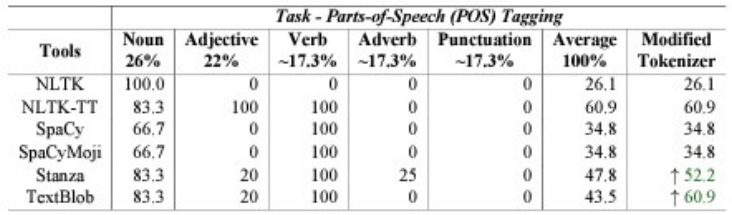
Figure 5: Tokenization

Parts of Speech
Assigning a label to each token to reflect a word class e.g., noun, adverb, verb, adjective etc.

NLP tools are expected to:

- Identify the correct use of part-of-speech tags by verifying the correctness of its prediction

NLP Tools Performance:

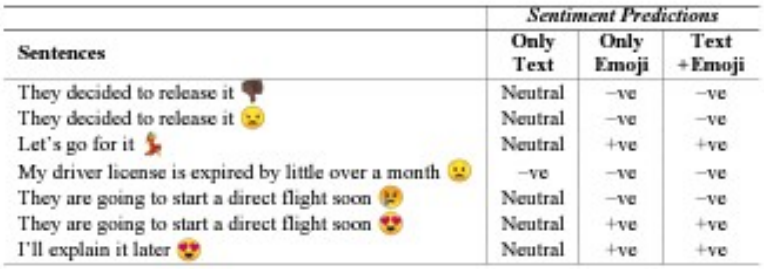
	

Tools	*Task - Parts-of-Speech (POS) Tagging*						
	Noun 26%	Adjective 22%	Verb ~17.3%	Adverb ~17.3%	Punctuation ~17.3%	Average 100%	Modified Tokenizer
NLTK	100.0	0	0	0	0	26.1	26.1
NLTK-TT	83.3	100	100	0	0	60.9	60.9
SpaCy	66.7	0	100	0	0	34.8	34.8
SpaCyMoji	66.7	0	100	0	0	34.8	34.8
Stanza	83.3	20	100	25	0	47.8	↑ 52.2
TextBlob	83.3	20	100	0	0	43.5	↑ 60.9

Table 3: The percentage of success of tools at labeling emojis with different parts-of-speech. The last column reports the average percentage of success when a modified tokenizer is used.

Figure 6: POS Tagging

***Note:** Gensim and PyNLPl are omitted as they do not offer POS tagging functionality

<u>Sentiment Analysis:</u> **Understand the sentiment effect emojis have on textual communication e.g., neutral texts such as "I'll explain it later" can have a different sentiment meaning depending on the emoji that is added to it e.g., " 😀" (positive) or " 👎" (negative).**	**NLP tools are expected to:** - Predict the polarity change (positive or negative) of texts with neutral or ambiguous sentiment **NLP Tools Performance:** 

Sentences	*Sentiment Predictions*		
	Only Text	Only Emoji	Text +Emoji
They decided to release it 🌪️	Neutral	–ve	–ve
They decided to release it 😒	Neutral	–ve	–ve
Let's go for it 💃	Neutral	+ve	+ve
My driver license is expired by little over a month 😐	–ve	–ve	–ve
They are going to start a direct flight soon 😢	Neutral	–ve	–ve
They are going to start a direct flight soon 😍	Neutral	+ve	+ve
I'll explain it later 😍	Neutral	+ve	+ve

Figure 7: Examples sentences with emojis that can moderate the overall sentiment of the sentence

<u>Final Takeaways from Literature Review 4</u>

1. Emojis should be considered when conducting sentiment analysis as they may affect the overall sentiment score of a message. This ensures a deeper understanding when processing social posts to detect signs of depression.
2. Various NLP tools can be explored to determine its emoji support level. Depending on the different cases of the emojis, the different appropriate NLP tools can be leveraged.

4.5 Literature Review #5 - A Comparative Study on Word Embeddings in Deep Learning for Text Classification

This research paper explores the effectiveness of different word embeddings in deep learning models for text classification tasks. The study compares the performance of different types of word embeddings - Word2Vec, GloVe, FastText, Character-Level Embedding, ELMo and BERT - in combination with deep learning models such as CNN and Bi-LSTM.

The table below shows the evaluation result, and we can see that the choice of word embedding method had a significant impact on the performance of the deep learning models. Among the different methods, BERT yields the best performance. The research study also mentioned that BERT generates conceptualised embeddings, meaning that the representation for each word depends on the context in which it appears. BERT is able to capture the nuances and complexities of language in a way that traditional word embedding methods, such as Word2Vec and GloVe, cannot, as they are only able to generate a fixed representation for each word in the vocabulary, not depending on the context in which the word appears. BERT being able to capture the context could be a potential reason why they are able to outperform other traditional methods.

	20NewsGroup				SST-2			
	CNN		BiLSTM		CNN		BiLSTM	
	Accuracy	macro-F1	Accuracy	macro-F1	Accuracy	macro-F1	Accuracy	macro-F1
Baseline	78.00±0.94	77.58±0.94	49.02±1.13	49.27±1.13	81.45±1.79	81.39±1.79	83.58±1.70	83.57±1.70
word2vec	82.67±0.85	82.14±0.87	62.45±1.09	62.43±1.09	82.14±1.76	82.07±1.76	84.09±1.68	84.02±1.68
G-50	79.98±0.90	79.44±0.91	69.03±1.04	68.98±1.04	80.78±1.81	80.75±1.81	83.03±1.72	83.00±1.73
G-100	81.03±0.89	80.28±0.90	72.80±1.00	72.58±1.01	81.47±1.78	81.40±1.79	84.15±1.68	84.07±1.68
G-200	82.95±0.85	82.26±0.86	75.47±0.97	75.16±0.98	82.80±1.73	82.76±1.73	85.04±1.64	85.02±1.64
G-300	82.61±0.86	81.85±0.87	74.24±0.99	73.97±0.99	82.03±1.76	81.98±1.77	84.88±1.65	84.86±1.65
F-300	82.16±0.86	81.57±0.88	62.37±1.09	62.40±1.09	81.44±1.79	81.30±1.79	84.33±1.67	84.31±1.67
F-300+G-300s	**84.39±0.82**	**83.65±0.84**	76.04±0.96	75.77±0.97	83.73±1.70	83.73±1.70	85.76±1.61	85.75±1.61
Char+G-100	82.11±0.87	81.31±0.88	79.31±0.91	78.78±0.92	82.46±1.75	82.41±1.75	84.79±1.65	84.75±1.65
ELMo	78.94±0.92	78.33±0.93	71.86±1.02	71.13±1.02	88.12±1.49	88.11±1.49	89.27±1.42	89.27±1.42
BERT	83.60±0.84	82.88±0.85	**81.25±0.88**	**80.54±0.89**	**90.01±1.38**	**90.00±1.38**	**90.04±1.38**	**90.04±1.38**

Figure 8: Results of different Word Embedding and Deep Learning Models

In the last section of the research, the author mentioned that it would be interesting to investigate these embeddings on other classification models, particularly the traditional machine learning methods (ML) such as Naïve Bayes, SVM, etc. Since these traditional approaches are more efficient in training and inference than the deep models used as downstream encoders, it would be interesting to investigate the balance between efficiency and potential performance tradeoffs.

Therefore, for our project, we will be leveraging the different word embedding techniques and thereafter feeding the output to both the traditional classifier and deep learning models to find out the best combination.

5. Tools and Resources Used

Data Exploration / Preprocessing	Modelling	Evaluation

Data Mining
- numpy
- Pandas
 	
Visualization
- matplotlib
- seaborn
- wordcloud
- PIL (Python Imaging Library)
 	
Regular Expression
- re
 	
Natural Language Toolkit
- nltk.corpus (stopwords)
- nltk.tokenize (Word_tokenize)
- nltk.stem (Wordnet Lemmatizer, Porter Stemmer)
- nltk (pos_tag, FreqDist)
- nltk.tokenize.casual (TweetTokenizer)
 	
Emoji
- unicode_emoji
	sklearn.model_selection
	- Train_test_split
	- GridSearchCV
	- Cross_val_score
	- KFold
	
	sklearn
	- MultinomialNB
	- GaussianNB
	- LogisticRegression
	- RandomForestClassifier
	- DecisionTreeClassifier
	- GradientBoostingClassifier
	- SVM.SVC
	
	pytorch
	- torch.nn
	  - nn.Linear
	  - nn.ReLU
	  - nn.Conv2d
	  - nn.lstm
	- torch.nn.functional
	- torch.utils.data (TensorDataset, DataLoader, RandomSampler, SequentialSampler)
	
	ELMo
	- tensorflow
	- tensorflow_hub
	
	BERT
	- transformers (BertModel, BertTokenizer, AdamW, get_linear_schedule_with_warmup)
	
	Word2Vec
	- Gensim.models
	
	GPT
	- Transformers (GPT2Tokenizer, TFGPT2Model)
	- TensorFlows (Keras.Models, Keras.Layers,
		sklearn.metrics
		- accuracy_score
		- precision_score
		- recall_score,
		- f1_score
		- confusion_matrix
		- classification_report

	Bidirectional, LSTM, Dense) - Torch (Torch.nn, Torch.utils.data, Dataloader, TensorDataset)	

6. Datasets

6.1 Source of Data

6.1.1 Sentiment Analysis - Selected

This dataset comprises around 10.1k rows of data extracted from Twitter. It consists of 3 columns. The id, tweets and labels with 1 indicating the person is suffering from depression and 0 for the non-depressed.

Source:
https://www.kaggle.com/datasets/gargmanas/sentimental-analysis-for-tweets

6.1.2 Twitter Dataset (Social Media)

This dataset comprises around 20k data extracted from Twitter. It consists of information regarding the post itself: Post_Text, Post_date, Number of retweets, favourites, and has a label as well.

Source:
https://www.kaggle.com/datasets/infamouscoder/mental-health-social-media?select=Mental-Health-Twitter.csv

6.2 Exploratory Data Analysis

6.2.1 Meta Data of our Dataset

Two labelled social media text datasets will be used to solve and reduce the depression problem rising on the internet. The two tables below show the original column and the description of the original datasets before any data cleaning.

Sentiment Analysis - Selected

Column Name	Description
Index	The ID value of a tweet
Message to Examine	The message on which the Sentimental Analysis needs to be performed
Label (Depression Results)	Does the person have depression? 0 stands for NO, and 1 stands for YES

Twitter Dataset (Social Media)

Column Name	Description
Index	Index of the data point
post_id	The Post ID of the tweet
post_created	The date and time the tweet was created
post_text	The uncleaned tweet that was posted
user_id	User Identification
followers	The number of followers
friends	The number of friends
favourites	The number of favourites
statuses	The total status count
retweets	The total number of retweets on the current tweet
label	Labels whether it is depression or not. 0 stands for NO, and 1 stands for YES

6.2.2 Dataset Analysis

As part of our EDA process, we have decided to check whether the datasets were imbalanced. However, before the team can analyse it, both datasets has been merged.

DF 1 - Sentiment Analysis Dataset
DF 2 - Twitter Dataset

As DF 2 contains more columns than DF 1, columns deemed unecessary to our project were dropped. Out of 11 columns in DF 2, the following 9 columns were dropped - index, post_id, prost_created, user_id, followers, friends, favourites, statuses, and retweets.

In addition, 1 column - index was also dropped from DF 1 as it is does not bring much value to our text classification analysis.

Once dropped and renamed, DF 1 and 2 now have 2 columns and can be merged. With that being said, the labels of the merged dataset were analysed and Figure 9 shows the results.

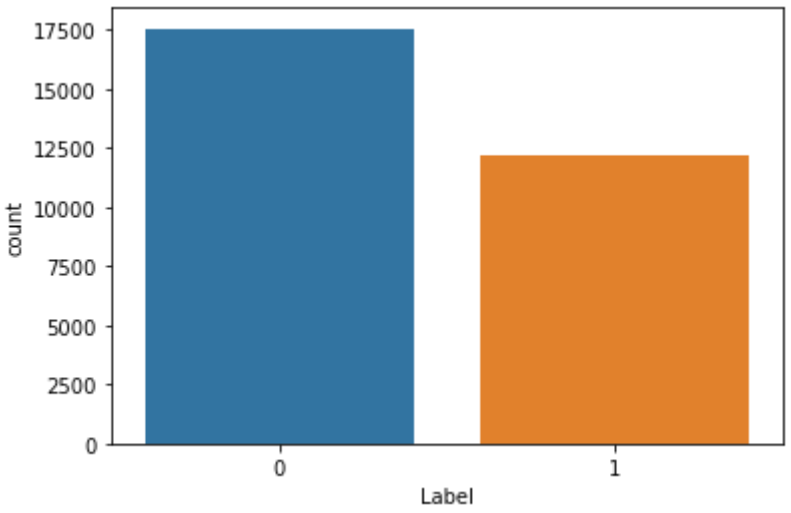


Figure 9: Barchart of “Depressed” and “Non-Depressed” Tweets

6.3 Dataset Preprocessing

Before Cleaning Dataset

Balanced Dataset

Overall, there are more Non-Depressed Tweets as compared to Depressed Tweets by roughly 17%. Thus, this data is a slight imbalance which prompts us that we might have to re-sample by oversampling instances of the minority class (generate synthetic data) or undersampling instances from the majority class. Perhaps, to

explore advanced techniques like SMOTE or GAN (Generative Adversarial Network). However, in this project, we did not as the data is only slightly imbalanced.

Percentage of Stop Words

Stop words are usually filtered out and they are considered to be less informative than other words in the text. The frequency of Stop Words in a sentence can indicate the quality of the text. For example, the high frequency of stop words suggests that the text is poorly written, lacks focus or is ungrammatical. From our dataset, most of the percentage of stop words falls between the 0.0 to 0.3 ranges, thus it indicates that our dataset has a rather high text quality.

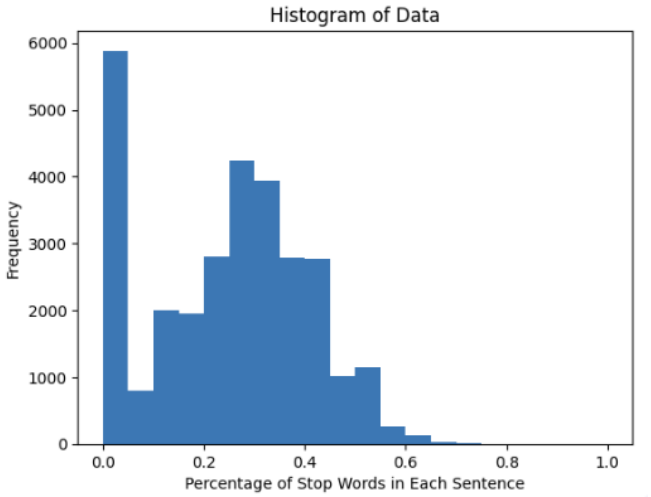


Figure 10: Percentage of Stop Words in Each sentence

Common Words in the Tweet

From the Word Cloud, we can see that:

1) There are internet links (Eg. https protocol)
2) There are Contractions (I'm, we're, he's)
3) There are Standalone Characters
4) There are short-forms like (lol, ly - love you)
5) There are also Stop words (The, That, etc)

However, this is not exhaustive, and we need a closer inspection of the dataset with sampling.

Figure 11: Word Cloud of Dataset

Other Words in the Tweet

Retweet/ Mentions of Users , HTML Entities, Hashtags, Emojis, Spelling Errors, and Other variations of alphabets. From this, there is a need for us to further clean our dataset.

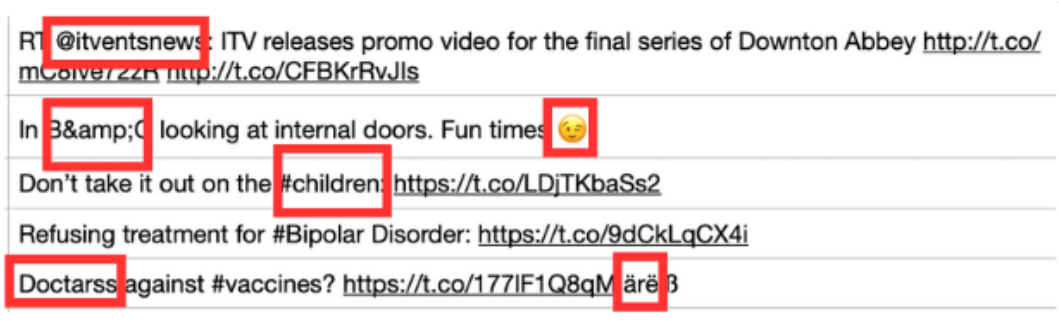


Figure 12: Checking of our Dataset

Steps to Cleaning Dataset

Before we started cleaning our dataset, we converted all the tweets to lowercase. The following few are the systematic steps for cleaning our dataset.

Step 1 : Removal of HTML Entities

```
# These are the HTML Entities
html_entities = {' ': '', '&iexcl;': '', '&cent;': '', '&pound;': '', '&curren;': '', '&yen;': '', '&brvbar;': '',
                 '&sect;': '', '&uml;': '', '&copy;': '', '&reg;': '', '&trade;': '', '&macr;': '', '&deg;': '',
                 '&plusmn;': '', '&sup2;': '', '&sup3;': '', '&acute;': '', '&micro;': '', '&para;': '', '&middot;': '',
                 '&cedil;': '', '&sup1;': '', '&ordm;': '', '&raquo;': '', '&frac14;': '', '&frac12;': '', '&frac34;': '
                 '&iquest;': '', '&Agrave;': '', '&Aacute;': '', '&Acirc;': '', '&Atilde;': '', '&Auml;': '',
                 '&Aring;': '', '&AElig;': '', '&Ccedil;': '', '&Egrave;': '', '&Eacute;': '', '&Ecirc;': '', '&Euml;':
                 '&Igrave;': '', '&Iacute;': '', '&Icirc;': '', '&Iuml;': '', '&ETH;': '', '&Ntilde;': '', '&Ograve;': '
                 '&Oacute;': '', '&Ocirc;': '', '&Otilde;': '', '&Ouml;': '', '&times;': '', '&Oslash;': '',
                 '&Ugrave;': '', '&Uacute;': '', '&Ucirc;': '', '&Uuml;': '', '&Yacute;': '', '&THORN;': '', '&szlig;':
                 '&agrave;': '', '&aacute;': '', '&acirc;': '', '&atilde;': '', '&auml;': '', '&aring;': '', '&aelig;':
                 '&ccedil;': '', '&egrave;': '', '&eacute;': '', '&ecirc;': '', '&euml;': '', '&igrave;': '', '&iacute;'
                 '&icirc;': '', '&iuml;': '', '&eth;': '', '&ntilde;': '', '&ograve;': '', '&oacute;': '', '&ocirc;': ''
                 '&otilde;': '', '&ouml;': '', '&divide;': '','&amp;': '', '&lt;': '', '&gt;': '', '&quot;': '', '&apos;
```

Figure 13: Dictionary of HTML entities

Step 2 : Removal of Hashtags

```
# Removal of HashTag
def remove_hashtags (text):
    text = re.sub(r"#\w+", '', text, flags=re.MULTILINE)
    return(text)

```

Figure 14: Codes to remove Hashtags

Step 3 : Conversion of Contractions to their full word form

```
contractions_dict = { "ain't": "are not","'s":" is","aren't": "are not",
                     "can't": "cannot","can't've": "cannot have",
                     "'cause": "because","could've": "could have","couldn't": "could not",
                     "couldn't've": "could not have", "didn't": "did not","doesn't": "does not",
                     "don't": "do not","hadn't": "had not","hadn't've": "had not have",
                     "hasn't": "has not","haven't": "have not","he'd": "he would",
                     "he'd've": "he would have","he'll": "he will", "he'll've": "he will have",
                     "how'd": "how did","how'd'y": "how do you","how'll": "how will",
                     "i'd": "i would", "i'd've": "i would have","i'll": "i will",
                     "i'll've": "i will have","i'm": "i am","i've": "i have", "isn't": "is not",
                     "it'd": "it would","it'd've": "it would have","it'll": "it will",
                     "it'll've": "it will have", "let's": "let us","ma'am": "madam",
                     "mayn't": "may not","might've": "might have","mightn't": "might not",
                     "mightn't've": "might not have","must've": "must have","mustn't": "must not",
                     "mustn't've": "must not have", "needn't": "need not",
                     "needn't've": "need not have","o'clock": "of the clock","oughtn't": "ought not",
                     "oughtn't've": "ought not have","shan't": "shall not","sha'n't": "shall not",
                     "shan't've": "shall not have","she'd": "she would","she'd've": "she would have",
                     "she'll": "she will", "she'll've": "she will have","should've": "should have",
                     "shouldn't": "should not", "shouldn't've": "should not have","so've": "so have",
                     "that'd": "that would","that'd've": "that would have", "there'd": "there would",
                     "there'd've": "there would have", "they'd": "they would",
                     "they'd've": "they would have","they'll": "they will",
                     "they'll've": "they will have", "they're": "they are","they've": "they have",
                     "to've": "to have","wasn't": "was not","we'd": "we would",
                     "we'd've": "we would have","we'll": "we will","we'll've": "we will have",
                     "we're": "we are","we've": "we have", "weren't": "were not","what'll": "what will",
                     "what'll've": "what will have","what're": "what are", "what've": "what have",
                     "when've": "when have","where'd": "where did", "where've": "where have",
                     "who'll": "who will","who'll've": "who will have","who've": "who have",
```

Figure 15: Dictionary of Contractions

Step 4 : Removal of URLs

```
def remove_urls (text):
    text = re.sub(r'(https|http)?:\/\/(\w|\.|\/|\?|\=|\&|\%)*\b', '', text, flags=re.MULTILINE)
    return(text)
```

```
df_uncleaned['Message'] = df_uncleaned['Message'].apply(lambda x:remove_urls(x))
```

```
df_uncleaned.sample(10)
```

Figure 16: Codes for removal of URL

Step 5 : Removal of Mentions/Retweets User Accounts

```
def remove_mentions_user (text):
    text = re.sub(r"@\w+", '', text, flags=re.MULTILINE)
    return(text)
```

Figure 17: Codes for removal of User Accounts

Step 6 : Removal of Punctuations

```
def remove_punctuations (text):
    text = re.sub(r"""[!"¦#$%&\'()*+,-./:;<=>?@[\\\]^_`{|}~•°…«“\":\"–”]""", '', text, flags=re.MULTILINE)
    return(text)

df_uncleaned['Message'] = df_uncleaned['Message'].apply(lambda x:remove_punctuations(x))
```

Figure 18: Codes for removal of Punctuations

Step 7 : Removal of Numbers

```
df_uncleaned['Message'] = df_uncleaned['Message'].str.replace('\d+', '')
```

Figure 19: Removal of Numbers

Step 8 : Emojis to textual description conversion

Ensure that the sentiment value that an emoji contains will be captured, prevent emojis from being removed through converting it to its textual description, and concatenate description with more than one word together.

Before

buddy stop 😂	0

After

Message	Label
buddy stop facewithtearsofjoy	0

Figure 20: Conversion of Emoji to Textual Description

Step 9 : Remove Stop Words

Remove common English words such as "The", "is", "are" etc. Helps remove low-level information to focus on other words of higher importance.

Before

going to the doc and after it to the studio flo ri woulda right round

After

going doc studio flo ri woulda right round

Figure 21: Removal of Stop words

Step 10 : Tokenization

Break up paragraphs and sentences into words. Helps with text processing by assigning meanings to smaller units of text.

Before

Message	Label
someone struggles severe depression perfect	1
really forgetting depression actual chemical ambalance feeling huh	1
boys flowers awesome specially yijung cha gaeul story try watch great	0

After

Tokenised
[someone, struggles, severe, depression, perfect]
[really, forgetting, depression, actual, chemical, ambalance, feeling, huh]
[boys, flowers, awesome, specially, yijung, cha, gaeul, story, try, watch, great]

Figure 22: Word Tokenization

Step 11 : Lemmatization

A text normalisation technique that helps to convert words to its base root form and helps group words of similar meaning with different form together e.g. "Playing", "Played" and "Plays" to "Play"

Before

Message	Label
stays using puppy filter	0
quite endorsement request key golden palace	0
concept	0

After

Tokenised
[stays, using, puppy, filter]
[quite, endorsement, request, key, golden, palace]
[concept]

Figure 23: Lemmatization

Data Analysis after cleaning Dataset

After we have cleaned our dataset such that we only keep the necessary information and attributes, we have proceeded to conduct EDA on the cleaned dataset. This is to further be sure that our data preparation and cleaning has been done correctly.

Common Words in overall tweets

Find the top 20 most common words in the dataset to provide insights of what words are more significant than the others in detecting depression. We need to know how many of these words are from depressed labelled tweets.

	Common_words	count
0	depression	2965
1	would	2140
2	like	1353
3	face	1232
4	good	1042
5	ame	995
6	ti	994
7	get	976
8	love	954
9	one	932
10	know	902
11	wi	855
12	day	819
13	u	745
14	go	723
15	people	691
16	got	659
17	see	619
18	today	596
19	new	594

Figure 24: Common Words in overall tweets

Common Words in the depressed labelled tweets

After taking a deeper look at only depressed labelled tweets, we are able to notice extra words such as "treatments" and "help" among others. However, some of these words may also appear in non-depressed label tweets.

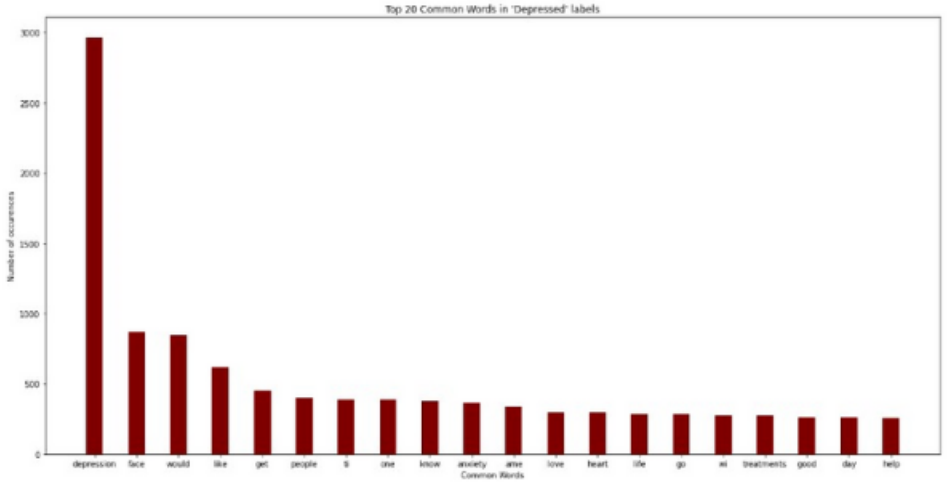


Figure 25: Histogram of Common words in Depressed Labelled tweets

Common Words in the non depressed labelled tweets

Looking at the common words in the non depressed label tweets, we are able to see repetitive words e.g., "would" and "like" that appeared in both depressed and non depressed labelled tweets. Such words that appeared in both labelled tweets will not help us in determining depressed tweets.

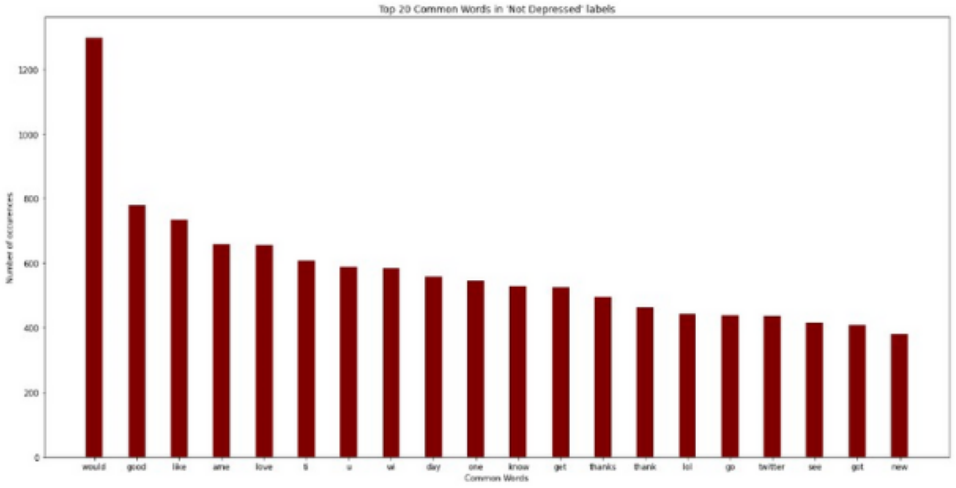


Figure 26: Histogram of Common words in non-depressed labelled tweets

Unique Common Words in the depressed labelled tweets

After obtaining the unique words for the depressed label tweets, we are able to identify 8 unique words that only exist in the depressed labelled tweets.

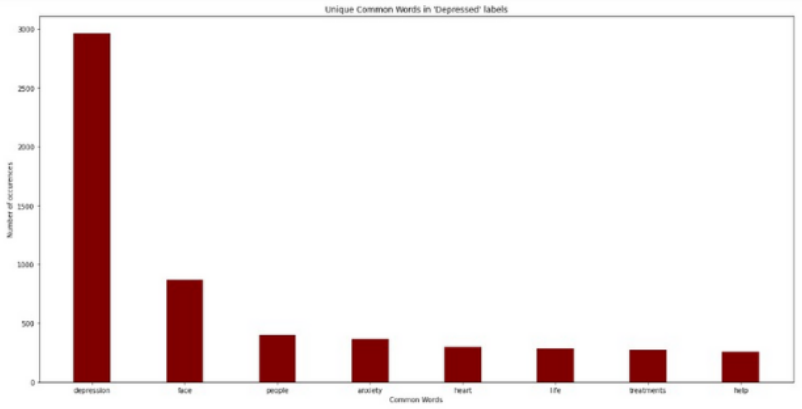


Figure 27: Histogram of Unique common words in depressed labelled tweets

7. Methodology

7.1 Natural Language Processing Background

Understanding language has always been a forte of human beings, whereas it is not as straightforward for computers. For instance, humans can effortlessly comprehend the connection between words such as king and queen, man and woman, tiger and tigress, but computers may find it challenging to do so. (Great Learning Team, 2020)

In order to tackle this problem, word embedding was introduced and it is done so by tokenizing each word in a sentence and converting them into a vector space. These embeddings are text representations in an n-dimensional space, where words with similar meanings have comparable representations. In other words, it means that analogous words are represented by almost identical vectors that are located in close proximity to each other in a vector space. However, it is important to note that each Word Embedding technique is created differently which produces word embeddings with different properties and levels of effectiveness for different tasks.

There are different type of Word Embedding Methods and some examples include Term Frequency-Inverse Document Frequency (TF-IDF), Bag of Word (BOW), Word2Vec, GloVe, BERT Embeddings from Language Models (ELMo), Generative Pre-Trained Transformer (GPT). As seen from Literature Review 5 above, different embedding methods, coupled with different deep learning models, achieves different results. For our project, we will be exploring 4 different word embedding methods, namely Word2Vec, ELMo, GPT and BERT

7.2 Traditional Classifiers

Our embedded output will be fed into these Classifiers as a follow up to the future work mentioned in literature review 5 as shown in the table below.

Traditional Classifiers	
Support Vector Machine (SVM)	Finds the unique optimal hyperplane to separate classes in binary classification.
Gradient Boosting (GB)	An ensemble learning algorithm that combines multiple Learners to improve classification results
Random Forest (RF)	An ensemble learning algorithm that creates multiple decision trees and combines their results to improve classification accuracy.
Logistic Regression (LR)	A statistical model that estimates the probability of binary outcome based on the relationship between the independent and dependent variables.
Decision Tree (DT)	A tree based model that makes predictions by following a sequence of binary decisions based on the features of the input data.

Gaussian Naives Bayes (GNB)	A probabilistic classifier that applies Bayes Theorem with the assumption of independence between features and a Gaussian distribution for continuous features.

Thus, in our general flow of the project, it will include the respective Word Embedding and classified using the Traditional Classification model as shown above.

7.3 Deep Learning Classifiers

Furthermore, our team also further explored with Deep Learning models that are stated in Literature Review 3 to classify whether the tweets are classified as Depressed or Non Depressed. The table below shows the Deep Learning Models that we will be using.

Deep Learning Models	
Convolutional Neural Network (CNN)	It uses convolutional layers to extract features, followed by pooling layers to reduce the dimensionality of the features, then fully connecting the layers for classification.
Recurrent Neural Network (RNN)	It uses loops to allow information to persist, making them well-suited for processing sequences of data such as text or Audio.
Long Short Term Memory (LSTM)	LSTMs have a more complex architecture that includes memory cells and gates that control the flow of information.
Bidirectional LSTM (Bi-LSTM)	Bi-LSTM processes input sequences in both directions (forward and backward), which allows the network to access past and future information concurrently.

Thus, in our general flow of the project, it will also include Word Embedding used in the Deep Learning Model as shown above.

7.4 Classification Metrics

At the end of the project, we aim to discover the best combination of word embedding methods together with their deep learning and traditional machine learning models. We also intend to figure out if deep learning approaches are able to yield better results than traditional classifiers as the former traded training efficiency with potential performance improvements. However, in order to determine the best performance model, we need to have a set of evaluation metrics. Henceforth, for our model evaluation, we will base off 4 metrics - accuracy, recall, precision and f1-score.

Accuracy in machine learning is a performance metric that measures how well a machine learning model correctly predicts the outcome of a given task. It is the ratio of the number of correct predictions to the total number of predictions made by the model. Our team felt that accuracy itself might not be enough to evaluate a model because the cost of misclassification is not uniform across the both classes. The cost of not being

able to accurately identify individual with depression is much higher than not being able to accurately identify individual without depression as the former can lead to death through the means of suicide.

That being said, recall is an important metric and useful in our context as it answers the question of what proportion of actual positives was identified correctly, whereas positive in our given problem refers to having depression. As mentioned, we want this group of individuals to be identified correctly so that actions can be taken to heal their mental illness because if left unattended could lead to death.

A metric that would take into account both precision and recall would be the f1-score which is the harmonic mean of both scores. This metric in particular would be important for us to determine the best performing model.

Hence for our implementations below, we have first considered **F1-Score then Recall**. However there may be variations and those will be explained in their respective sections.

7.5 General Process Flow for Traditional and Deep Learning Models

General Flow for Traditional Classifiers

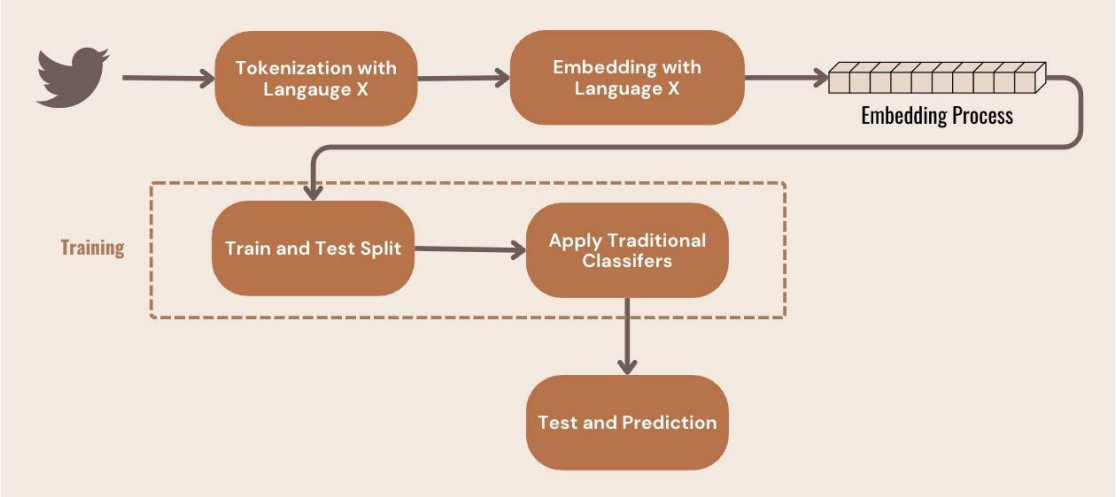


Figure 28: General Flow of Traditional Classifier Process

In the Figure 28 above, it shows the general flow of how we process the Twitter Data and prepare it for Traditional Classification. After cleaning the data which was mentioned in the previous section, we use the cleaned data and Tokenize with Language Model X and proceed to embed it with Language X. After tokenization and having the embeddings generated, the data was then split into train and test sets in which the train set contains 75% of the data and the test set contains 25% of the data. These data were then trained on the 6 traditional classifier models mentioned in part 7.2.

Apart from just fitting the data into the machine learning model, K-fold cross-validation was also used to further improve the baseline model's prediction as well as to prevent the problem of overfitting. The training

dataset will be split into 10 fold, where one fold will be considered for testing and the rest will be for training the model during every iteration. The average training accuracy score will be calculated based on all the accuracy scores from each iteration.

General Flow for Deep Learning Models

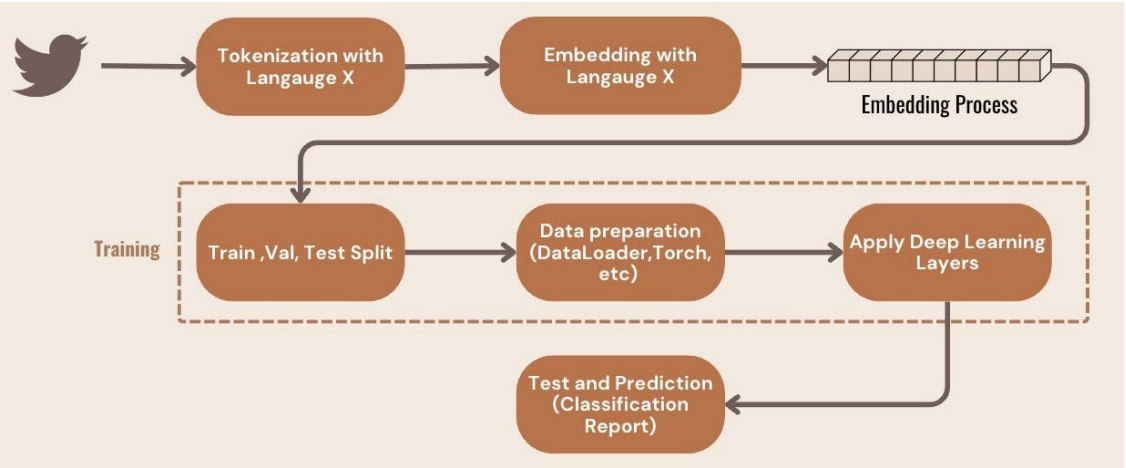


Figure 29: General flow of Deep Learning Models

While the flow is generally similar to the Traditional Classifier, some additional steps include validation split alongside train and test split, data preparation steps to prepare the embedded data in the format that can be used by the Deep Learning Models, and applying Deep Learning layers instead of Traditional Classifier Models. We then run each of the Deep Learning Models listed in part 7.3 and display the Test and Prediction Results in a Classification Report.

We attempted to standardise most of the model parameters and hyperparameters of the Deep Learning models across all 4 of the language models. While the actual code implementation of these Deep Learning models may be different due to certain constraints of the language model it is paired with, the parameter configurations are mostly identical. The reason for this is to provide a fair comparison of the accuracies between the language models when applying Deep Learning. Listed below are the common configurations for the 4 Deep Learning models:

CNN:

Number of Convolution Layers = 3

Kernel Size for first Conv Layer = 3

Kernel Size for second Conv Layer = 4

Kernel Size for third Conv Layer = 5

Stride = 1

Activation Function for each Conv Layer = ReLU

Dropout Rate = 0.4

Activation Function for output Layer = Sigmoid

Optimizer = Adam / AdamW

Loss function = Cross-Entropy / Binary Cross-Entropy

Learning Rate = 5e-5

Epsilon = 1e-8

Batch size = 32

Number of epochs = 10

RNN:

Second layer activation function = ReLU

Optimizer = Adam / AdamW

Loss function = Cross-Entropy / Binary Cross-Entropy

Learning Rate = 5e-5

Epsilon = 1e-8

Batch size = 32

Number of epochs = 10

BiLSTM:

Optimizer = Adam / AdamW

Loss function = Cross-Entropy / Binary Cross-Entropy

Learning Rate = 5e-5

Epsilon = 1e-8

Batch size = 32

Number of epochs = 10

LSTM:

Optimizer = Adam / AdamW

Loss function = Cross-Entropy / Binary Cross-Entropy

Learning Rate = 5e-5

Epsilon = 1e-8

Batch size = 32

Number of epochs = 10

8. Word2Vec Natural Language Processing

8.1 Word2Vec Word Embeddings

Word2Vec is a neural network-based learning language model that is used to represent words as vectors, allowing us to perform mathematical operations on words and analyse their similarities and relationships.

Being a non-contextual based method for generating word embeddings, Word2Vec does not consider the context in which a word appears when creating its vector representation (ChatGPT, 2023). Hence, each word has a fixed representation based on its occurrence statistics across the entire dataset.

Word2vec is an effective word embedding methodology and its strength comes from its ability to group together vectors of similar words, while vectors with different meanings will have vectors that are far apart. With a large enough dataset, Word2Vec can make relatively strong estimates about a word's meaning. For example, words like "King" and "Queen" will have word vector representations that are similar to one another (ChatGPT, 2023). These word embeddings can then be used as inputs to other machine learning models for various natural language processing tasks.

In Word2Vec, there are two different architectures: Continuous Bag of Words (CBOW) and Continuous Skip-Gram Model. Referencing to the diagram below, CBOW is a variation of Word2Vec model that learns word embedding by predicting the target word from its context words, hence it tries to predict a target word based on the words that appear around it. Meanwhile, skip-gram is a type of Word2Vec model that learns word embeddings by predicting the context words given a target word, hence it tries to predict the words that appear in the context of a given word.

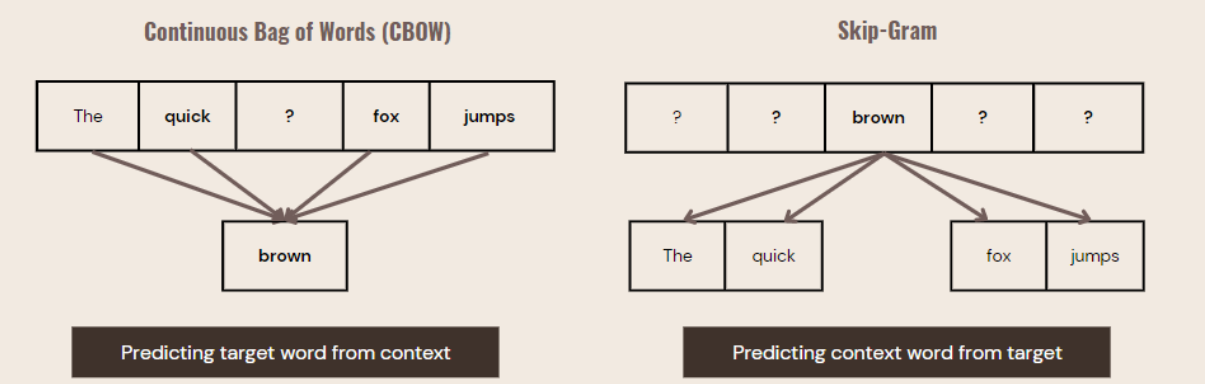


Figure 30: Word2Vec Architecture

For our implementation, we adopted the CBOW model as it is generally faster and requires less memory to train. It is also more effective at capturing the meaning of frequent worlds and hence would be the ideal choice for sentiment analysis.

8.2 Word2Vec Traditional Classification Results

Due to the time limit of 6 hours for SMU GPU, we had to run the machine learning models in batches, with SVM itself taking 3 hours 18 minutes out of the total 7 hours of training the machine learning algorithms.

Model/Metric	Train Accuracy Mean	Test Accuracy	Precision	Recall	F1-Score
SVM	0.75	0.75	0.48	0.82	0.60
GB	0.75	0.75	0.45	0.84	0.59
LR	0.73	0.72	0.54	0.69	0.61
RF	0.75	0.75	0.45	0.83	0.59
GNB	0.63	0.62	0.06	0.89	0.10
DT	0.66	0.65	0.58	0.56	0.57

From the table above, we can see that while logistic regression may have the best F1-Score of 0.61, it has a relatively lower Recall score than SVM which has a similar F1-Score. Hence we will conclude that generally, the **best performing Machine Learning model would be Word2Vec embedding with SVM** among the 6 traditional ML models that we have explored. Support Vector Machine (SVM) achieved the highest train accuracy mean and test accuracy as well. A reason why we think SVM is able to outperform other models is its ability to find the best hyperplane that separates the data points into different classes. The hyperplane is chosen to maximise the margin, which is the distance between the hyperplane and the nearest data points from each class, also called the support vectors. However, one trade-off is it usually takes longer to train the SVM model.

8.3 Word2Vec Deep Learning Classification Results

The Deep Learning implementations follow the process flow as stated in part 7.5 for Deep Learning Models. The data was split into training and testing datasets, with the testing set consisting of 25% of the original dataset and the validation set consisting of 20% of the remaining training set.

Word2Vec RNN

By combining Word2Vec and RNN, the hybrid model can benefit from the non-contextualized word embeddings learned by Word2Vec, while also leveraging the ability of RNNs to model sequential data. They have a feedback loop that will allow for storing and using information from the previous steps, hence making RNN effective at modelling dependencies and long-term dependencies in sequential data. To use Word2Vec with RNN, the input layer is initialised with the pre-trained Word2Vec embeddings and this will provide for a dense and meaningful representation of words, which can help RNN better capture the semantics of the data (ChatGPT, 2023). The screenshot below will show how we configured our RNN layers and the activation functions used.

```
from keras.optimizers import Adam

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=88)

# Define the RNN model architecture
model = Sequential()
model.add(LSTM(units=64, input_shape=(max_len, embedding_size)))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dense(y.shape[1], activation='softmax'))

# Compile the model
optimizer = Adam(lr=5e-5, epsilon=1e-8)
model.compile(loss=binary_crossentropy, optimizer=optimizer, metrics=['accuracy'])

# Train the model
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_split=0.2)
```

```
Train on 15876 samples, validate on 3970 samples
Epoch 1/10
15876/15876 [==============================] - 14s 864us/step - loss: 0.6292 - acc: 0.6669 - val_loss: 0.5437 - val_acc: 0.7453
Epoch 2/10
15876/15876 [==============================] - 13s 803us/step - loss: 0.5280 - acc: 0.7508 - val_loss: 0.5198 - val_acc: 0.7504
Epoch 3/10
15876/15876 [==============================] - 13s 804us/step - loss: 0.5130 - acc: 0.7591 - val_loss: 0.5125 - val_acc: 0.7554
Epoch 4/10
15876/15876 [==============================] - 13s 804us/step - loss: 0.5031 - acc: 0.7619 - val_loss: 0.5038 - val_acc: 0.7572
Epoch 5/10
15876/15876 [==============================] - 13s 803us/step - loss: 0.4942 - acc: 0.7654 - val_loss: 0.4995 - val_acc: 0.7579
Epoch 6/10
15876/15876 [==============================] - 13s 807us/step - loss: 0.4862 - acc: 0.7693 - val_loss: 0.4960 - val_acc: 0.7605
Epoch 7/10
15876/15876 [==============================] - 13s 803us/step - loss: 0.4795 - acc: 0.7708 - val_loss: 0.4909 - val_acc: 0.7605
Epoch 8/10
15876/15876 [==============================] - 13s 802us/step - loss: 0.4732 - acc: 0.7736 - val_loss: 0.4878 - val_acc: 0.7632
Epoch 9/10
15876/15876 [==============================] - 13s 817us/step - loss: 0.4667 - acc: 0.7769 - val_loss: 0.4865 - val_acc: 0.7610
Epoch 10/10
15876/15876 [==============================] - 13s 823us/step - loss: 0.4618 - acc: 0.7792 - val_loss: 0.4846 - val_acc: 0.7630
```

Figure 31: Code for Word2Vec RNN

In addition to the model hyperparameters stated in part 7.5 RNN, the model includes a single LSTM layer with 64 units, an input shape of (max_len, embedding size) and a drop out rate of 0.5. The dropout layer is included to prevent overfitting by randomly dropping out some of the neurons during training. The final output layer is a dense layer with a softmax activation function that will produce the predicted class probabilities. Upon training completion, the final epoch accuracy was 0.7792, training loss was 0.4618, validation accuracy was 0.7630, and validation loss was 0.4846. This indicates that there was no overfitting.

The test results from our RNN implementation are shown below:

Test Loss: 0.4920854500732514 **Test Classification Report**				
	Precision	**Recall**	**F1-Score**	**Support**
0	0.74	0.91	0.82	4004
1	0.79	0.52	0.63	2612
Accuracy			0.76	6616
Macro Average	0.77	0.71	0.72	6616
Weighted Average	0.76	0.76	0.74	6616

Figure 32: Classification Report Word2Vec RNN

The relevant results to observe are from the “1” row and the “Accuracy” row.

Word2Vec LSTM

The following code shows how we have implemented LSTM.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=88)

# Define the LSTM model architecture
model = Sequential()
model.add(LSTM(units=64, input_shape=(max_len, embedding_size)))
model.add(Dropout(0.5))
model.add(Dense(y.shape[1], activation='sigmoid'))

# Compile the model
optimizer = Adam(lr=5e-5, epsilon=1e-8)
model.compile(loss=binary_crossentropy, optimizer=optimizer, metrics=['accuracy'])

# Train the model
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_split=0.2)
```

```
Train on 15876 samples, validate on 3970 samples
Epoch 1/10
15876/15876 [==============================] - 40s 2ms/step - loss: 0.6297 - acc: 0.6691 - val_loss: 0.5447 - val_acc: 0.7385
Epoch 2/10
15876/15876 [==============================] - 33s 2ms/step - loss: 0.5319 - acc: 0.7492 - val_loss: 0.5239 - val_acc: 0.7479
Epoch 3/10
15876/15876 [==============================] - 33s 2ms/step - loss: 0.5170 - acc: 0.7598 - val_loss: 0.5143 - val_acc: 0.7531
Epoch 4/10
15876/15876 [==============================] - 20s 1ms/step - loss: 0.5055 - acc: 0.7638 - val_loss: 0.5070 - val_acc: 0.7547
Epoch 5/10
15876/15876 [==============================] - 13s 813us/step - loss: 0.4949 - acc: 0.7671 - val_loss: 0.4995 - val_acc: 0.7589
Epoch 6/10
15876/15876 [==============================] - 13s 806us/step - loss: 0.4858 - acc: 0.7705 - val_loss: 0.4934 - val_acc: 0.7579
Epoch 7/10
15876/15876 [==============================] - 13s 812us/step - loss: 0.4765 - acc: 0.7724 - val_loss: 0.4884 - val_acc: 0.7617
Epoch 8/10
15876/15876 [==============================] - 13s 805us/step - loss: 0.4688 - acc: 0.7763 - val_loss: 0.4893 - val_acc: 0.7592
Epoch 9/10
15876/15876 [==============================] - 13s 804us/step - loss: 0.4642 - acc: 0.7782 - val_loss: 0.4829 - val_acc: 0.7572
Epoch 10/10
15876/15876 [==============================] - 13s 806us/step - loss: 0.4581 - acc: 0.7805 - val_loss: 0.4814 - val_acc: 0.7584
```

Figure 33: Code for Word2Vec LSTM

In addition to the model hyperparameters stated in part 7.5 for LSTM, the LSTM model architecture consists of a single LSTM later with 64 units and the input shape of the LSTM layer is defined by “max_len” and “embedding_size” which are the maximum sequence length and the size of the word embedding respectively. To prevent overfitting, a dropout layer has been incorporated after the LSTM layer where it will randomly drop out a portion of the neurons during training, thereby decreasing the impact of any single neuron. Additionally, a dense layer with sigmoid activation function is introduced as the last layer to generate binary output. Upon training completion, the final epoch training accuracy was 0.7805, training loss was 0.4581, validation accuracy was 0.7584, and validation loss was 0.4814. This indicates that there was no overfitting.

The test results from our LSTM implementation is shown below:

Test Loss: 0.4834320670964496 **Test Classification Report**				
	Precision	**Recall**	**F1-Score**	**Support**
0	0.74	0.94	0.83	4004
1	0.84	0.50	0.63	2612
Accuracy			0.76	6616
Macro Average	0.79	0.72	0.73	6616
Weighted Average	0.78	0.76	0.75	6616

Figure 34: Classification Report for Word2Vec LSTM

The relevant results to observe are from the "1" row and the "Accuracy" row.

Word2Vec Bi-LSTM

The following code shows how we have implemented Bi-LSTM.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=88)

# Define the Bi-LSTM model architecture
model = Sequential()
model.add(Bidirectional(LSTM(units=64), input_shape=(max_len, embedding_size)))
model.add(Dropout(0.5))
model.add(Dense(y.shape[1], activation='sigmoid'))

# Compile the model
optimizer = Adam(lr=5e-5, epsilon=1e-8)
model.compile(loss=binary_crossentropy, optimizer=optimizer, metrics=['accuracy'])

# Train the model
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_split=0.2)
```

```
Train on 15876 samples, validate on 3970 samples
Epoch 1/10
15876/15876 [==============================] - 22s 1ms/step - loss: 0.6193 - acc: 0.6706 - val_loss: 0.5478 - val_acc: 0.7403
Epoch 2/10
15876/15876 [==============================] - 19s 1ms/step - loss: 0.5392 - acc: 0.7497 - val_loss: 0.5278 - val_acc: 0.7481
Epoch 3/10
15876/15876 [==============================] - 19s 1ms/step - loss: 0.5198 - acc: 0.7594 - val_loss: 0.5155 - val_acc: 0.7547
Epoch 4/10
15876/15876 [==============================] - 19s 1ms/step - loss: 0.5065 - acc: 0.7652 - val_loss: 0.5103 - val_acc: 0.7589
Epoch 5/10
15876/15876 [==============================] - 19s 1ms/step - loss: 0.4954 - acc: 0.7705 - val_loss: 0.5011 - val_acc: 0.7630
Epoch 6/10
15876/15876 [==============================] - 19s 1ms/step - loss: 0.4850 - acc: 0.7736 - val_loss: 0.4963 - val_acc: 0.7594
Epoch 7/10
15876/15876 [==============================] - 19s 1ms/step - loss: 0.4772 - acc: 0.7779 - val_loss: 0.4935 - val_acc: 0.7589
Epoch 8/10
15876/15876 [==============================] - 19s 1ms/step - loss: 0.4703 - acc: 0.7801 - val_loss: 0.4939 - val_acc: 0.7599
Epoch 9/10
15876/15876 [==============================] - 19s 1ms/step - loss: 0.4650 - acc: 0.7833 - val_loss: 0.4880 - val_acc: 0.7610
Epoch 10/10
15876/15876 [==============================] - 19s 1ms/step - loss: 0.4600 - acc: 0.7865 - val_loss: 0.4915 - val_acc: 0.7620
```

Figure 35: Codes for Word2Vec Bi-LSTM

In addition to the model hyperparameters stated in part 7.5 for Bi-LSTM, the architecture includes a Bidirectional LSTM layer with 64 units which operates by processing the input sequence through two LSTM layers, one in the forward direction and the other in the backward direction. By concatenating the outputs from both directions, the model was able to capture information from both the past and future context. A

dropout layer with a rate of 0.5 was also added to the model to prevent overfitting by randomly dropping out some of the neurons during training while a dense layer with sigmoid activation was added to produce the binary output. The final epoch training accuracy was 0.7865, training loss was 0.4600, validation accuracy was 0.7620, and validation loss was 0.4915. This indicates that there was no overfitting.

The test results from our Bi-LSTM implementation are shown below:

Test Loss: 0.5001656930204459 **Test Classification Report**				
	Precision	**Recall**	**F1-Score**	**Support**
0	0.75	0.88	0.81	4004
1	0.76	0.56	0.64	2612
Accuracy			0.76	6616
Macro Average	0.76	0.72	0.73	6616
Weighted Average	0.76	0.76	0.75	6616

Figure 36: Classification Report for Word2Vec Bi-LSTM

The relevant results to observe are from the "1" row and the "Accuracy" row.

Word2Vec CNN

The following code shows how we have implemented CNN with Word2Vec.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten, Conv1D, MaxPooling1D
from sklearn.model_selection import train_test_split
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.losses import binary_crossentropy

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=88)

# Define the CNN model architecture
model = Sequential()
model.add(Conv1D(filters=128, kernel_size=3, activation='relu', padding='same', input_shape=(X.shape[1], X.shape[2])))
model.add(Dropout(0.4))
model.add(MaxPooling1D(pool_size=2))
model.add(Conv1D(filters=128, kernel_size=4, activation='relu', padding='same'))
model.add(Dropout(0.4))
model.add(MaxPooling1D(pool_size=2))
model.add(Conv1D(filters=128, kernel_size=5, activation='relu', padding='same'))
model.add(Dropout(0.4))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(y.shape[1], activation='sigmoid'))

# Compile the model
optimizer = Adam(lr=5e-5, epsilon=1e-8)
model.compile(loss=binary_crossentropy, optimizer=optimizer, metrics=['accuracy'])

# Train the model
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_split=0.2)
```

Figure 37: Codes for Word2Vec CNN

In addition to the model hyperparameters stated in part 7.5 for CNN, a few "MaxPooling1D" layers with a pool size of 2 are added. A Flatten layer is added to convert the output to a 1D array. The final epoch training accuracy was 0.7597, training loss was 0.4841, validation accuracy was 0.7538, and validation loss was 0.5103. This indicates that there was no overfitting.

The test results from our CNN implementation is shown below:

Test Loss: 0.5065721661897495 **Test Classification Report**				
	Precision	**Recall**	**F1-Score**	**Support**
0	0.74	0.93	0.82	4004
1	0.82	0.50	0.62	2612
Accuracy			0.76	6616
Macro Average	0.78	0.71	0.72	6616
Weighted Average	0.77	0.76	0.74	6616

Figure 38: Classification Report for Word2Vec CNN

The relevant results to observe are from the "1" row and the "Accuracy" row.

In conclusion, for deep learning models, taking into account F1-Score and Recall as our main metrics, we can conclude that for Word2Vec, **the best performing Deep Learning model with the Word2Vec embedding would be Bi-LSTM, with the highest F1-score of 0.64.** This could be because Word2Vec embedding provides good initialization for the Bi-LSTM model, allowing it to better capture the complex relationships between words in the input sequence. Bi-LSTM itself is also a relatively powerful deep learning model with its ability to process the input in both directions, allowing for the algorithm to have a more comprehensive understanding of the context in which each word appears.

8.4 Conclusion of Word2Vec

Model/Metric	**Accuracy**	**Precision**	**Recall**	**F1-Score**
RNN	0.76	0.79	0.52	0.63
LSTM	0.76	0.84	0.50	0.63
CNN	0.76	0.82	0.50	0.62
BI-LSTM	0.76	0.76	0.56	0.64
LR	0.72	0.54	0.69	0.61

SVM	0.75	0.48	0.82	0.60
GNB	0.75	0.45	0.84	0.59
GB	0.75	0.45	0.84	0.59
RF	0.75	0.45	0.83	0.59
DT	0.65	0.58	0.56	0.59

The table above gives an overview of the performance of Word2Vec embeddings together with the 6 machine learning models and 4 deep learning models implemented. We can conclude that the best performing combination with Word2Vec will be Bi-LSTM as it has the highest F1-score.

9. Embeddings from Language Models (ELMo) Natural Language Processing

9.1 ELMo Word Embeddings

Embeddings from Language Models (ELMo) is a contextualised word embedding method that takes into account the different meaning of similar words according to the way it is being used in a sentence. ELMo trains a large text corpus by leveraging the bidirectional language model (biLM) to come up with pre-trained embeddings that can be applied to datasets (Jain, n.d.).

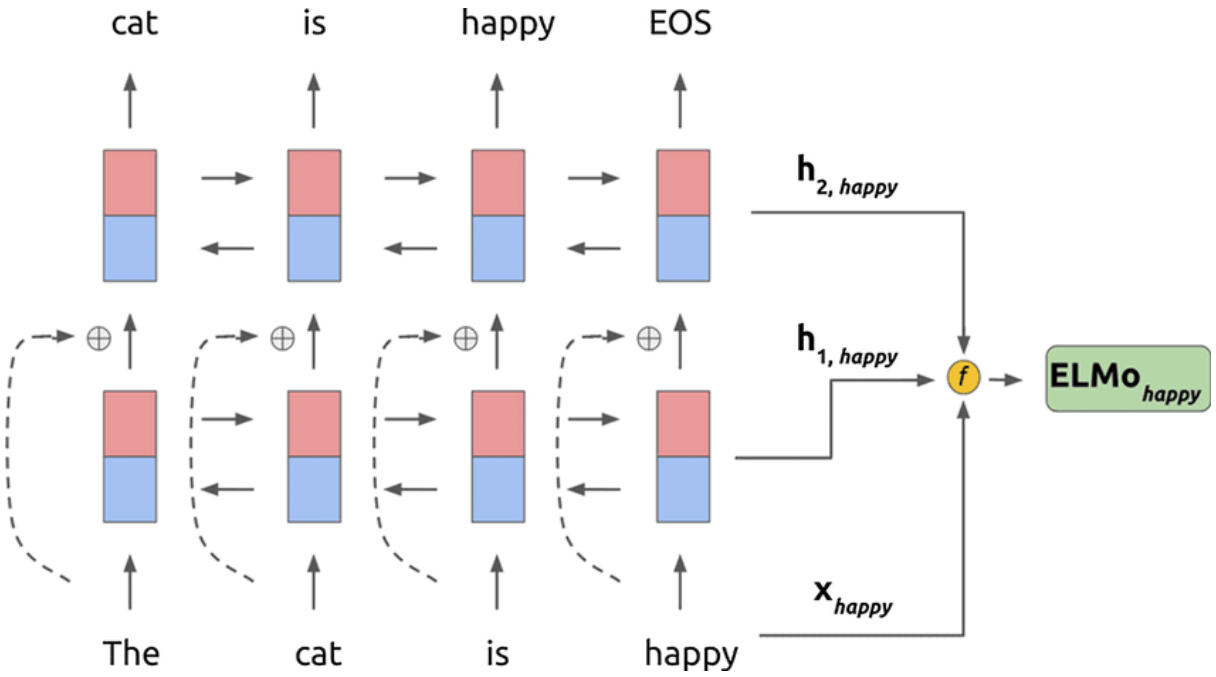


Figure 39: ELMo Architecture (Lee, 2021)

The ELMo architecture consists of a two-layer bidirectional language model (biLM) which contains a forward pass and backward pass in each layer and are being stacked together. The biLM model first

generates raw word vectors from the input words and introduces it to the first layer of biLM. The input data is then passed into the forward pass to learn about each word and the context through learning the word before it. The same thing happens in the backward pass, with the only difference being that the context is learnt from the word after it. When put together, the word vectors of a word will be formed and then processed into the second layer of biLM. Hence, the final ELMo word embedding will be the result of the weighted sum of the initial raw word vectors and the word vectors produced by the two layers in the model (Joshi, 2019).

We decided to utilise ELMo embeddings on top of Word2Vec embeddings due to the consideration of context in determining the word vectors in ELMo models. In contrast to Word2Vec, which are context-independent word embeddings that do not take into account how a word is used. This means that a word would still share the same vector representation when being conveyed differently in sentences (Jain, n.d.). For example, given the sentences, “Here is your present” and “Do not dwell in the past, live in the present”, we are able to see the word “present” being used twice but with different meanings. The former suggests that it is a gift whereas the latter would use it to describe the current state of time. In a Word2Vec model, the difference in this information does not get captured as only one word embedding will be generated for the word “present”. On the other hand, an ELMo model would understand the context and generate two different word embeddings for As a result, this would potentially affect the accuracy of the model when classifying texts.

```
tweets_elmo_embedding=ELMoEmbedding(df_cleaned.Message)
```

```
2023-04-01 02:41:23.212979: I tensorflow/core/platform/cpu_feature_guard.cc:142] Your CPU supports instructions that this
TensorFlow binary was not compiled to use: AVX2 FMA
2023-04-01 02:41:23.219671: I tensorflow/core/platform/profile_utils/cpu_utils.cc:94] CPU Frequency: 2194900000 Hz
2023-04-01 02:41:23.220676: I tensorflow/compiler/xla/service/service.cc:168] XLA service 0xb219760 initialized for platf
orm Host (this does not guarantee that XLA will be used). Devices:
2023-04-01 02:41:23.220701: I tensorflow/compiler/xla/service/service.cc:176]   StreamExecutor device (0): Host, Default
Version
2023-04-01 02:41:23.401444: I tensorflow/compiler/xla/service/service.cc:168] XLA service 0x994e390 initialized for platf
orm CUDA (this does not guarantee that XLA will be used). Devices:
2023-04-01 02:41:23.401489: I tensorflow/compiler/xla/service/service.cc:176]   StreamExecutor device (0): NVIDIA GeForce
RTX 2080 Ti, Compute Capability 7.5
2023-04-01 02:41:23.401604: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1159] Device interconnect StreamExecutor w
ith strength 1 edge matrix:
2023-04-01 02:41:23.401620: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1165]
2023-04-01 02:41:32.990179: W tensorflow/core/framework/cpu_allocator_impl.cc:81] Allocation of 45957120000 exceeds 10% o
f system memory.
```

Figure 40: System Memory Error ELMo Word Embedding

```
%%time
df_depressed = df_cleaned[df_cleaned['Label'] == 1]
df_nondepressed = df_cleaned[df_cleaned['Label'] == 0]

tweets_list_1 = df_depressed['Message'].iloc[:2500].tolist()
tweets_list_2 = df_nondepressed['Message'].iloc[:2500].tolist()
tweets_list = tweets_list_1 + tweets_list_2


tweets_elmo_embedding=ELMoEmbedding(tweets_list)

tweets_elmo_embedding.shape
```

Figure 41: Extracting subset of the dataset

Before we moved on to the application of ELMo in our project, we would like to highlight a limitation in our application of the ELMo model. While generating the word embeddings for our dataset, we received a "Allocation of 45957120000 exceeds 10% of system memory" error. We then found out that the number '45957120000' refers to the number of bytes of a block of memory which would amount to 45GB. Due to the system's memory limit of allowing only 10% of the total system memory to be allocated at any time, it was unable to cater to such a big memory block (ChatGPT, 2023). Therefore, in the interest of time and complexity, our group decided to perform only on 5000 rows of the dataset as it is the maximum number of rows that an ELMo embedding could be generated for.

9.2 ELMo Traditional Classifiers Results

The general flow of the ELMo embedding implementation is similar to the Traditional Classifiers process in part 7.5. When generating the ELMo embeddings of the dataset, dimensionality reduction via max pooling was done to reshape the dimension of the ELMo embeddings from 3D to 2D.

Model/Metric	Train Accuracy Mean	Test Accuracy	Precision	Recall	F1-Score
SVM	0.94	0.93	0.88	0.98	0.93
GB	0.93	0.92	0.88	0.96	0.92
LR	0.92	0.91	0.91	0.91	0.91
RF	0.93	0.92	0.86	0.98	0.92
GNB	0.76	0.76	0.75	0.76	0.75
DT	0.86	0.84	0.85	0.84	0.84

Figure 42: Traditional Classifier Evaluation Score

From the table above, we can observe that SVM is the best performing traditional ML model with a F1-Score of 0.93. Although it is worth noting that it has a slightly lower Recall score of 0.98 than that of Random

Forest of 0.98, we would still prioritise F1-Score first. Hence we will conclude that generally, SVM performs the best among the 6 traditional ML models that we have explored. SVM achieves the highest train accuracy mean and test accuracy as well. Once again, we feel that SVM's ability to find the best hyperlane to separate the data point proved to be the best method to classify the tweets.

9.3 ELMo Deep Learning Implementation & Results

The Deep Learning implementations for ELMo follow the process flow as stated in part 7.5 for Deep Learning Models.After generating the word embeddings for the data, the dimensions of the data will be handled accordingly to fit the respective deep learning models. We will be looking at how specifically the data will be handled in their respective parts. The data was split into training and testing datasets, with the testing set consisting of 25% of the original dataset and the validation set consisting of 20% of the remaining training set.

ELMo RNN

To use ELMo with RNN, the input layer is initialised with the pre-trained ELM embeddings and this will provide for a dense and meaningful representation of words, which can help RNN better capture the semantics of the data. The screenshot below will show how we configured our RNN layers and the activation functions used.

```
from keras.optimizers import Adam
from tensorflow.keras.losses import binary_crossentropy

model = Sequential()
model.add(LSTM(units=64, input_shape=(max_len, embedding_size)))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))


# Compile the model
optimizer = Adam(lr=5e-5, epsilon=1e-8)
model.compile(loss=binary_crossentropy, optimizer=optimizer, metrics=['accuracy'])

# Train the model
model.fit(X_train, y_encoded, epochs=10, batch_size=32, validation_split=0.2)
Epoch 5/10
3000/3000 [==============================] - 6s 2ms/step - loss: 0.5608 - acc: 0.7210 - val_loss: 0.2874 - val_acc: 0.916
0
Epoch 6/10
3000/3000 [==============================] - 6s 2ms/step - loss: 0.2559 - acc: 0.9237 - val_loss: 0.1989 - val_acc: 0.942
7
Epoch 7/10
3000/3000 [==============================] - 6s 2ms/step - loss: 0.2012 - acc: 0.9413 - val_loss: 0.2037 - val_acc: 0.932
0
Epoch 8/10
3000/3000 [==============================] - 6s 2ms/step - loss: 0.1839 - acc: 0.9473 - val_loss: 0.1871 - val_acc: 0.945
3
Epoch 9/10
3000/3000 [==============================] - 6s 2ms/step - loss: 0.1652 - acc: 0.9540 - val_loss: 0.1861 - val_acc: 0.945
3
Epoch 10/10
3000/3000 [==============================] - 6s 2ms/step - loss: 0.1590 - acc: 0.9580 - val_loss: 0.1985 - val_acc: 0.945
3

<keras.callbacks.History at 0x7f95f0d657d0>
```

Figure 43: Codes for ELMO RNN

For our RNN implementation, the model has the exact same configurations as the general hyperparameters from part 7.5 for RNN. Some additions include the ELMo word embeddings that will be fed into the input layer, a single LSTM layer with 64 units, an input shape of (max_len, embedding size) and a drop out rate of 0.5. The dropout layer is included to prevent overfitting by randomly dropping out some of the neurons during training. The final output layer is a dense layer with a softmax activation function that will produce the predicted class probabilities. The model was then trained on the training and test sets. The final epoch training accuracy was 0.9580, training loss was 0.1590, validation accuracy was 0.945, and validation loss was 0.1985. A high validation accuracy indicates that there was no overfitting as the model was able to generalise well on the validation data..

The test results from our RNN implementation is shown below:

Test Loss: 0.2745239432297647 **Test Classification Report**				
	Precision	**Recall**	**F1-Score**	**Support**
0	0.90	0.94	0.92	634
1	0.94	0.89	0.92	616
Accuracy			0.92	1250
Macro Average	0.92	0.92	0.92	1250
Weighted Average	0.92	0.92	0.92	1250

Figure 44: Classification Report for ELMO RNN

The relevant results to observe are from the “1” row and the “Accuracy” row.

ELMo LSTM

The following code shows how we have implemented LSTM for the ELMo word embeddings.

```python
import tensorflow as tf
import tensorflow_hub as hub
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Dense, LSTM, Dropout

class ELMoLSTMModel:
    def __init__(self, input1, input2):

        # Define the LSTM model architecture
        self.model = Sequential()
        self.model.add(LSTM(128, input_shape=(input1, input2)))
        self.model.add(Dropout(0.5))
        self.model.add(Dense(1, activation="sigmoid"))

    def compile_model(self):
        self.optimizer = Adam(lr=5e-5, epsilon=1e-8)
        self.model.compile(loss="binary_crossentropy", optimizer=self.optimizer, metrics=["accuracy"])

    def train_model(self, X_train, y_train, X_val, y_val, epochs=10, batch_size=32):
        self.model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=epochs, batch_size=batch_size)

    def evaluate_model(self, X_test, y_test):
        loss, accuracy = self.model.evaluate(X_test, y_test)
        print('Test loss:', test_loss)

    def predict(self, X_new):
        # Make a prediction for a new input
        prediction = self.model.predict(X_new)
        return prediction
```

Figure 45: Codes for ELMO LSTM

```python
# Split data into train, validation, and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=88)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.2, random_state=42)

# Initialize the model, loss function, and optimizer
model = ELMoLSTMModel(X_train.shape[1],X_train.shape[2])
model.compile_model()
batch_size = 10
epochs = 32
model.train_model(X_train, y_train, X_val, y_val, batch_size, epochs)
```

```
WARNING:tensorflow:From /common/home/users/j/jianyi.lye.2020/jupyterlab-venv-tf/lib/python3.7/site-packages/keras/backend/tensorflow_backend.py:3075: The name tf.log is deprecated. Please use tf.math.log instead.

WARNING:tensorflow:From /common/home/users/j/jianyi.lye.2020/jupyterlab-venv-tf/lib/python3.7/site-packages/keras/backend/tensorflow_backend.py:3075: The name tf.log is deprecated. Please use tf.math.log instead.

Train on 3000 samples, validate on 750 samples
Epoch 1/10
3000/3000 [==============================] - 10s 3ms/step - loss: 0.6958 - acc: 0.4947 - val_loss: 0.6930 - val_acc: 0.5027
Epoch 2/10
3000/3000 [==============================] - 9s 3ms/step - loss: 0.6928 - acc: 0.5083 - val_loss: 0.6930 - val_acc: 0.5027
Epoch 3/10
3000/3000 [==============================] - 9s 3ms/step - loss: 0.6905 - acc: 0.5343 - val_loss: 0.6817 - val_acc: 0.6187
Epoch 4/10
3000/3000 [==============================] - 9s 3ms/step - loss: 0.4670 - acc: 0.7783 - val_loss: 0.2390 - val_acc: 0.9320
Epoch 5/10
3000/3000 [==============================] - 9s 3ms/step - loss: 0.2255 - acc: 0.9320 - val_loss: 0.2039 - val_acc: 0.9373
Epoch 6/10
3000/3000 [==============================] - 9s 3ms/step - loss: 0.1940 - acc: 0.9440 - val_loss: 0.1893 - val_acc: 0.9387
Epoch 7/10
3000/3000 [==============================] - 9s 3ms/step - loss: 0.1835 - acc: 0.9473 - val_loss: 0.1975 - val_acc: 0.9400
Epoch 8/10
3000/3000 [==============================] - 9s 3ms/step - loss: 0.1758 - acc: 0.9493 - val_loss: 0.1926 - val_acc: 0.9360
Epoch 9/10
3000/3000 [==============================] - 9s 3ms/step - loss: 0.1660 - acc: 0.9513 - val_loss: 0.1940 - val_acc: 0.9400
Epoch 10/10
3000/3000 [==============================] - 9s 3ms/step - loss: 0.1540 - acc: 0.9577 - val_loss: 0.2089 - val_acc: 0.9333
```

Figure 46: Epoch for ELMO LSTM

In addition to the model hyperparameters stated in part 7.5 for LSTM, the LSTM model architecture consists of a single LSTM later with 128 units and the input shape of the LSTM layer is defined by “max_len” and

“embedding_size” which are the maximum sequence length and the size of the word embedding respectively. The use of more units in the LSTM layer for the ELMo implementation compared to Word2Vec is due to the need to capture more complex patterns in the ELMo word embeddings as compared to the Word2Vec word embeddings. To prevent overfitting, a dropout layer has been incorporated after the LSTM layer where it will randomly drop out a portion of the neurons during training, thereby decreasing the impact of any single neuron. Additionally, a dense layer with sigmoid activation function is introduced as the last layer to generate binary output. The model was then trained on the training, validation, and test sets. The final epoch training accuracy was 0.9577, training loss was 0.1540, validation accuracy was 0.9333, and validation loss was 0.2089. A high validation accuracy indicates that there was no overfitting as the model was able to generalise well on the validation data..

The test results from our LSTM implementation is shown below:

Test Loss: 0.2745239432297647 **Test Classification Report**				
	Precision	**Recall**	**F1-Score**	**Support**
0	0.88	0.98	0.93	634
1	0.98	0.87	0.92	616
Accuracy			0.92	1250
Macro Average	0.93	0.92	0.92	1250
Weighted Average	0.93	0.92	0.92	1250

Figure 47: Classification Report for ELMO LSTM

The relevant results to observe are from the “1” row and the “Accuracy” row.

ELMo Bi-LSTM

The following code shows how we have implemented Bi-LSTM for the ELMo word embeddings.

```python
import tensorflow as tf
import tensorflow_hub as hub
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Dense, LSTM, Bidirectional, Dropout


class BiLSTMModel:
    def __init__(self, input1, input2):

        # Define the BiLSTM model architecture
        self.model = Sequential()
        self.model.add(Bidirectional(LSTM(128), input_shape=(input1, input2)))
        self.model.add(Dropout(0.5))
        self.model.add(Dense(1, activation="sigmoid"))

    def compile_model(self):
        self.optimizer = Adam(lr=5e-5, epsilon=1e-8)
        self.model.compile(loss="binary_crossentropy", optimizer=self.optimizer, metrics=["accuracy"])

    def train_model(self, X_train, y_train, X_val,y_val, epochs=10, batch_size=32):
        self.model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=10, batch_size=32)

    def evaluate_model(self, X_test, y_test):
        loss, accuracy = self.model.evaluate(X_test, y_test)
        print('Test loss:', test_loss)

    def predict(self, X_new):
        # Make a prediction for a new input
        prediction = self.model.predict(X_new)
        return prediction
```

Figure 48: ELMO Bi-LSTM

```python
# Split data into train, validation, and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=88)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.2, random_state=42)
```

```python
# Initialize the model, loss function, and optimizer
model = BiLSTMModel(X_train.shape[1],X_train.shape[2])
model.compile_model()
batch_size = 10
epochs = 32
model.train_model(X_train, y_train, X_val,y_val, batch_size, epochs)
```

```
Train on 3000 samples, validate on 750 samples
Epoch 1/10
3000/3000 [==============================] - 18s 6ms/step - loss: 0.5521 - acc: 0.7840 - val_loss: 0.4013 - val_acc: 0.8960
Epoch 2/10
3000/3000 [==============================] - 20s 7ms/step - loss: 0.2917 - acc: 0.9153 - val_loss: 0.2121 - val_acc: 0.9387
Epoch 3/10
3000/3000 [==============================] - 17s 6ms/step - loss: 0.2010 - acc: 0.9317 - val_loss: 0.1797 - val_acc: 0.9453
Epoch 4/10
3000/3000 [==============================] - 23s 8ms/step - loss: 0.1731 - acc: 0.9387 - val_loss: 0.1687 - val_acc: 0.9453
Epoch 5/10
3000/3000 [==============================] - 18s 6ms/step - loss: 0.1599 - acc: 0.9437 - val_loss: 0.1648 - val_acc: 0.9453
Epoch 6/10
3000/3000 [==============================] - 22s 7ms/step - loss: 0.1458 - acc: 0.9493 - val_loss: 0.1610 - val_acc: 0.9467
Epoch 7/10
3000/3000 [==============================] - 23s 8ms/step - loss: 0.1360 - acc: 0.9520 - val_loss: 0.1653 - val_acc: 0.9453
Epoch 8/10
3000/3000 [==============================] - 23s 8ms/step - loss: 0.1272 - acc: 0.9577 - val_loss: 0.1732 - val_acc: 0.9427
Epoch 9/10
3000/3000 [==============================] - 23s 8ms/step - loss: 0.1151 - acc: 0.9613 - val_loss: 0.1735 - val_acc: 0.9427
Epoch 10/10
3000/3000 [==============================] - 23s 8ms/step - loss: 0.1059 - acc: 0.9630 - val_loss: 0.1611 - val_acc: 0.9467
```

Figure 49: Epoch of ELMO Bi-LSTM

In addition to the model hyperparameters stated in part 7.5 for Bi-LSTM, the architecture includes a Bidirectional LSTM layer with 128 units which operates by processing the input sequence through two LSTM layers, one in the forward direction and the other in the backward direction. By concatenating the outputs from both directions, the model was able to capture information from both the past and future context. A dropout layer with a rate of 0.5 was also added to the model to prevent overfitting by randomly dropping out some of the neurons during training while a dense layer with sigmoid activation was added to produce the binary output. The final epoch training accuracy was 0.9630, training loss was 0.1059, validation accuracy was 0.9467, and validation loss was 0.1611. A high validation accuracy indicates that there was no overfitting as the model was able to generalise well on the validation data..

The test results from our Bi-LSTM implementation is shown below:

Test Loss: 0.2745239432297647
Test Classification Report

	Precision	Recall	F1-Score	Support
0	0.90	0.96	0.93	634
1	0.96	0.89	0.92	616
Accuracy			0.93	1250
Macro Average	0.93	0.93	0.93	1250
Weighted Average	0.93	0.93	0.93	1250

Figure 50: Classification Report for ELMO Bi-LSTM

The relevant results to observe are from the “1” row and the “Accuracy” row.

ELMo CNN

The following code shows how we have implemented CNN for the ELMo word embeddings.

```python
import torch
import torch.nn as nn
import torch.optim as optim
import os
from torch.utils.data import DataLoader, TensorDataset
from sklearn.metrics import accuracy_score

# Define the CNN model
class ELMoEmbeddingCNN(nn.Module):
    def __init__(self):
        super(ELMoEmbeddingCNN, self).__init__()
        self.linear = nn.Linear(768, 1024)
        self.conv1 = nn.Conv1d(1024, 128, kernel_size=3, stride=1, padding=2)
        self.conv2 = nn.Conv1d(1024, 128, kernel_size=4, stride=1, padding=3)
        self.conv3 = nn.Conv1d(1024, 128, kernel_size=5, stride=1, padding=4)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.4)
        self.max_pool = nn.AdaptiveMaxPool1d(1)
        self.fc = nn.Linear(128 * 3, 1)

    def forward(self, x):
        if x.dim() == 2:
            x = x.unsqueeze(1)  # Add the sequence dimension if it's missing
        x = x.permute(0, 2, 1)  # Swap the feature and sequence dimensions
        x1 = self.conv1(x)
        x1 = self.relu(x1)
        x1 = self.dropout(x1)
        x1 = self.max_pool(x1).squeeze()

        x2 = self.conv2(x)
        x2 = self.relu(x2)
        x2 = self.dropout(x2)
        x2 = self.max_pool(x2).squeeze()

        x3 = self.conv3(x)
        x3 = self.relu(x3)
        x3 = self.dropout(x3)
        x3 = self.max_pool(x3).squeeze()

        x = torch.cat([x1, x2, x3], dim=1)
        x = self.fc(x)
        x = torch.sigmoid(x)
        return x.squeeze()
```

Figure 51: Code sample for ELMO CNN

In addition to the model hyperparameters stated in part 7.5 for CNN, a few “AdaptiveMaxPool1D” layers with a pool size of 1 are added.

```python
# Split the training data into training and validation sets
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size=0.2, random_state=42)


# Create train and test datasets
train_dataset = TensorDataset(torch.tensor(x_train, dtype=torch.float32), torch.tensor(y_train, dtype=torch.float32))
val_dataset = TensorDataset(torch.tensor(x_val, dtype=torch.float32), torch.tensor(y_val, dtype=torch.float32))
test_dataset = TensorDataset(torch.tensor(x_test, dtype=torch.float32), torch.tensor(y_test, dtype=torch.float32))

# Create data loaders
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)

# Initialize the model, loss function, and optimizer
model = ELMoEmbeddingCNN()
criterion = nn.BCELoss()
optimizer = optim.AdamW(model.parameters(), lr=5e-5, eps=1e-8)
```

Figure 52: Data Preparation for ELMO CNN

We will then prepare the data using for training and testing the model. It splits the training data into training and validation sets using scikit-learn's train_test_split function, creates PyTorch datasets and data loaders for the training, validation, and testing sets, and initialises the model, loss function, and optimizer.

```
from sklearn.metrics import classification_report

# Create a directory for saving the model
model_dir = "models"
os.makedirs(model_dir, exist_ok=True)

# Define a function to calculate metrics and print the classification report
def get_classification_report(data_loader, loader_type):
    model.eval()
    y_true = []
    y_pred = []

    with torch.no_grad():
        for batch_x, batch_y in data_loader:
            outputs = model(batch_x)
            predictions = (outputs > 0.5).int()
            y_true.extend(batch_y.tolist())
            y_pred.extend(predictions.tolist())

    report = classification_report(y_true, y_pred, output_dict=True, zero_division=0)
    print(f"{loader_type} Classification Report (Epoch {epoch+1}):")
    print(classification_report(y_true, y_pred, output_dict=False, zero_division=0))
    return report

# Train the model and calculate metrics for each epoch
num_epochs = 10
best_val_loss = float("inf")
best_epoch = 0
for epoch in range(num_epochs):
    # Training phase
    model.train()
    train_loss = 0

    for batch_x, batch_y in train_loader:
        optimizer.zero_grad()
        outputs = model(batch_x)
        loss = criterion(outputs, batch_y)
        loss.backward()
        optimizer.step()

                    loss.item() * batch_x.size(0)

                    train_loader.dataset)
                    ɔch+1}/{num_epochs}] Train Loss: {train_loss:.4f}")


    model.eval()
    val_loss = 0

    with torch.no_grad():
        for batch_x, batch_y in val_loader:
            outputs = model(batch_x)
            loss = criterion(outputs, batch_y)

            val loss += loss.item() * batch x.size(0)
            val_loss /= len(val_loader.dataset)

            # Print the classification reports for training and validation sets
            train_report = get_classification_report(train_loader, "Train")
            print(f"Epoch [{epoch+1}/{num_epochs}] Val Loss: {val_loss:.4f}")
            val_report = get_classification_report(val_loader, "Validation")

            # Save the model with the best validation accuracy
            if val_loss < best_val_loss:
                best_val_loss = val_loss
                best_epoch = epoch + 1
                torch.save(model.state_dict(), os.path.join(model_dir, "best_ELMo_+_CNN_model.pt"))
                print(f"Model saved for epoch {best_epoch}")
        print(f"Best model saved at epoch {best_epoch} with validation loss {best_val_loss:.4f}")
```

Figure 53: Codes for Classification Report for ELMO CNN

We will then proceed to train the CNN model on the training data and evaluate its performance on the validation and testing sets. The classification metrics (precision, recall, F1-score, and accuracy) of the training results will be printed for each epoch. The training loop iterates over a fixed number of epochs, updating the model weights using backpropagation and saving the model with the best validation loss.

The model was then trained on the training, validation, and test sets. The final epoch training accuracy was 0.94, training loss was 0.1841, validation accuracy was 0.94, and validation loss was 0.2297. A high validation accuracy indicates that there was no overfitting as the model was able to generalise well on the validation data..

Train Loss: 0.1841 **Train Classification Report**				
	Precision	**Recall**	**F1-Score**	**Support**
0	0.92	0.96	0.94	1493
1	0.96	0.92	0.94	1507
Accuracy			0.94	3000
Macro Average	0.94	0.94	0.94	3000
Weighted Average	0.94	0.94	0.94	3000

Figure 54: Classification Report for Train ELMO CNN

Validation Loss: 0.2297 **Validation Classification Report**				
	Precision	**Recall**	**F1-Score**	**Support**
0	0.93	0.94	0.94	373
1	0.94	0.93	0.94	377
Accuracy			0.94	750
Macro Average	0.94	0.94	0.94	750
Weighted Average	0.94	0.94	0.94	750

Figure 55: Classification Report for Validation ELMO CNN

The test results from our CNN implementation is shown below:

Test Loss: 0.2416 **Test Classification Report**				
	Precision	**Recall**	**F1-Score**	**Support**
0	0.91	0.93	0.92	634
1	0.93	0.91	0.92	616
Accuracy			0.92	1250
Macro Average	0.92	0.92	0.92	1250
Weighted Average	0.92	0.92	0.92	1250

Figure 56: Classification Report for Test ELMO CNN

The relevant results to observe are from the "1" row and the "Accuracy" row.

From the above discussion and taking into account F1-Score and Recall as our main metrics, we can conclude that for ELMo, the best performing Deep Learning model with the ELMo embedding would be CNN with a highest F1-Score of 0.92 and Recall score of 0.91. This could be because ELMo embedding provides complex representations for the CNN model, allowing it to better capture the patterns or relationships between nearby/further words or phrases in a sentence with richer information. CNN itself is also less sensitive to the order of words as compared to other models. As ELMo embeddings capture the context of the entire sentence, rather than only previous or next words, the order becomes irrelevant. Hence, it is easier for CNN to capture such information due to the lack of sensitivity as compared to the other models.

9.4 Conclusion of ELMo

Model/Metric	**Accuracy**	**Precision**	**Recall**	**F1-Score**
RNN	0.92	0.94	0.89	0.92
LSTM	0.92	0.98	0.87	0.92
CNN	0.92	0.93	0.91	0.92
BI-LSTM	0.93	0.96	0.89	0.92
LR	0.91	0.91	0.91	0.91
SVM	0.93	0.88	0.98	0.93
GNB	0.76	0.75	0.76	0.75
GB	0.92	0.88	0.96	0.92

RF	0.92	0.86	0.98	0.92
DT	0.84	0.85	0.84	0.84

The table above gives an overview of the performance of ELMo embeddings together with the 6 machine learning models and 4 deep learning models implemented. Upon comparing the results of all the traditional ML models and deep learning models, we found out that SVM is the best performing model with a F1-Score of 0.93 and recall score of 0.98.

10. Generative pre-trained transformer (GPT)

10.1 GPT Word Embeddings

Generative Pre-Trained Transformer is a type of deep learning model that was introduced by OpenAI in 2018, it uses a variant of the Transformer architecture originally introduced by Vaswani et al. There are multiple versions of GPT, such as GPT-1, GPT-2, GPT-3, and currently OpenAI is in the midst of developing GPT-4. However, our team has shortlisted the usage of GPT-2 due to a few reasons. GPT-2 is available for free while GPT-3 requires users to pay to access GPT-3 through OpenAI's API. In terms of Model Size, GPT-2 is still a large and powerful model with 1.5 billion parameters as compared to GPT-3 which would be less computationally expensive. With these two reasons, our team will be using GPT-2 for this project.

Figure 57: Simplified GPT-2 Architecture

The general flow of GPT-2 architecture works is that the input is a sequence of tokens which the Tokenizer will convert the input text into a sequence of tokens that the model can understand. Each token is then embedded into a high-dimensional vector space, which allows the model to learn representations of the input text.

The embedded tokens are then fed into a series of transformer layers. Each transformer layer consists of two sublayers: a self-attention layer and a feedforward neural network. The self-attention layer allows the model to weigh the importance of each token in the sequence based on the context, while the feedforward network applies a non-linear transformation to the attention outputs.

The transformer layer's output is then passed to the decoder which predicts the next token in the sequence based on the previous tokens. Thus, the output of the decoder will return a vector representation of the input. (ChatGPT,2023)

10.2 GPT Traditional Classifiers Results

Model/Metric	Train Accuracy Mean	Test Accuracy	Precision	Recall	F1-Score
SVM	0.76	0.77	0.47	0.90	0.62
GB	0.75	0.75	0.48	0.81	0.60
LR	0.71	0.73	0.35	0.94	0.50
RF	0.74	0.75	0.43	0.88	0.58
GNB	0.60	0.60	0.30	0.49	0.38
DT	0.64	0.64	0.55	0.55	0.55

Figure 58: GPT-2 Traditional Classifier Results

Based on the table above, we observed that SVM is the best performing model with a F1-Score of 0.62 and a Recall Score of 0.90. It is also worth noting that Logistic Regression has a slightly higher Recall Score of 0.94 but however in terms of F1 Score, it has performed worse than SVM.

One possible reason why we think that SVM is able to outperform other models is due to the fact that it is able to find the best hyperplane that separates the data points into different classes, enabling the most optimal solution to be found. However, the hyperplane is chosen to maximise the margins, which is one of the reasons why SVM is more computationally expensive than other models during training.

10.3 GPT Deep Learning Implementation & Results

Our deep learning implementations follow the process flow as described in part 7.5 for Deep Learning, with some additional steps for different models.

GPT-2 RNN

Model Configuration & Initialization

```python
import torch
import torch.nn as nn

class GPT_RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers):
        super(GPT_RNN, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.rnn = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)
        self.dropout = nn.Dropout(0.5)
        self.fc1 = nn.Linear(hidden_size, 64)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(64, output_size)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)
        c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.device)
        out, _ = self.rnn(x, (h0, c0))
        out = self.dropout(out[:, -1, :])
        out = self.fc1(out)
        out = self.relu(out)
        out = self.fc2(out)
        out = self.softmax(out)
        return out
```

```python
import torch.optim as optim
from sklearn.metrics import accuracy_score

# Define hyperparameters
input_size = 768
hidden_size = 128
output_size = 2
num_layers = 1
lr = 5e-5
num_epochs = 10

# Instantiate the model and optimizer
model = GPT_RNN(input_size, hidden_size, output_size, num_layers)
optimizer = optim.AdamW(model.parameters(), lr=lr, eps=1e-8)
```

Figure 59: Codes for GPT RNN

Our RNN implementation uses a Long Short-Term Memory (LSTM) unit to process input data of size 768. The hidden_size and num_layers are the number of hidden units and LSTM layers in the RNN, respectively. The forward method of the class takes an input tensor x and applies the LSTM unit to it. The output of the LSTM is then passed through a linear layer, another linear layer, and a softmax activation function, producing an output tensor of size 2. The hyperparameters stated in part 7.5 for RNN are defined and the model is then instantiated.

Train, Validation, Test Split and Data Preparation

```python
# Split data into train, validation, and test sets
x_train, x_test, y_train, y_test = train_test_split(X_features, y, test_size=0.25, random_state=88)
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size=0.2, random_state=42)
```

```python
# Convert training data to PyTorch tensors
x_train = torch.from_numpy(x_train).float()
y_train = torch.from_numpy(y_train.to_numpy()).long()
x_train = x_train.unsqueeze(1)
```

```python
# Convert validation data to PyTorch tensors
x_val = torch.from_numpy(x_val).float()
y_val = torch.from_numpy(y_val.to_numpy()).long()
x_val = x_val.unsqueeze(1)
```

Figure 60: Codes for Dataset Splitting

The data was split into training and testing datasets, with the testing set consisting of 25% of the original dataset and the validation set consisting of 20% of the remaining training set. The training and validation data are then converted to PyTorch tensors using the torch.from_numpy function, which converts NumPy arrays to PyTorch tensors. The x_train and x_val tensors are first cast to float tensors and then unsqueezed to add an additional dimension. The y_train and y_val tensors are cast to long tensors since they contain labels.

Model Training

```
from sklearn.metrics import accuracy_score, precision_recall_fscore_support, classification_report

# Train the model
best_val_loss = float("inf")
for epoch in range(num_epochs):
    model.train()
    optimizer.zero_grad()
    outputs = model(x_train)
    loss = nn.CrossEntropyLoss()(outputs, y_train)
    loss.backward()
    optimizer.step()

    # Compute training accuracy
    with torch.no_grad():
        _, predicted = torch.max(outputs.data, 1)
        train_report = classification_report(y_train.numpy(), predicted.numpy(), digits=4, zero_division=1)

        # Compute validation accuracy and loss
        model.eval()
        val_outputs = model(x_val)
        _, val_predicted = torch.max(val_outputs.squeeze().data, 1)
        val_acc = accuracy_score(val_predicted.numpy(), y_val.numpy())
        val_loss = nn.CrossEntropyLoss()(val_outputs.squeeze(), y_val)
        val_report = classification_report(y_val.numpy(), val_predicted.numpy(), digits=4, zero_division=1)

        print('Epoch [{}/{}], Training Loss: {:.4f}'.format(epoch+1, num_epochs, loss.item()))
        print('Training Classification Report:\n', train_report)

        print('Validation Loss: {:.4f}'.format(val_loss.item()))
        print('Validation Classification Report:\n', val_report)

        # Save the model with the best validation loss
        if val_loss < best_val_loss:
            best_val_loss = val_loss
            best_epoch = epoch + 1
            torch.save(model.state_dict(), os.path.join(model_dir, "best_GPT_+_RNN_model.pt"))
            print(f"Model saved for epoch {best_epoch}")

print(f"Best model saved at epoch {best_epoch} with validation loss {best_val_loss:.4f}")
```

Figure 61: Model Training for GPT RNN

Model training and evaluation are performed using PyTorch and scikit-learn metrics. During each epoch, the training loss is computed and printed to the console. The code then computes the training classification report for each class. After computing the training metrics, the model is switched to evaluation mode, and the validation accuracy and loss are computed. The validation classification report is also computed and printed to the console. During training, the best model is saved based on the lowest validation loss. The training results from our RNN implementation are shown below:

Train Loss: 0.6956 **Train Classification Report**				
	Precision	**Recall**	**F1-Score**	**Support**
0	1.00	0.00	0.00	9654
1	0.39	1.00	0.56	6222
Accuracy			0.39	15876
Macro Average	0.70	0.50	0.28	15876
Weighted Average	0.76	0.39	0.22	15876

Figure 62: Classification Report for Train GPT RNN

Validation Loss: 0.6955 **Validation Classification Report**				
	Precision	**Recall**	**F1-Score**	**Support**
0	1.00	0.00	0.00	2387
1	0.40	1.00	0.57	1583
Accuracy			0.40	3970
Macro Average	0.70	0.50	0.29	3970
Weighted Average	0.76	0.40	0.23	3970

Figure 63: Classification Report for Validation GPT RNN

Testing the Model

```
# Convert test data to PyTorch tensors
x_test = torch.from_numpy(x_test).float()
y_test = torch.from_numpy(y_test.to_numpy()).long()
x_test = x_test.unsqueeze(1)
```

```
# Load the best model
model.load_state_dict(torch.load(os.path.join(model_dir, "best_GPT_+_RNN_model.pt")))

# Test the model
model.eval()
with torch.no_grad():
    test_outputs = model(x_test)
    _, test_predicted = torch.max(test_outputs.data, 1)
    test_acc = accuracy_score(test_predicted.numpy(), y_test.numpy())
    test_loss = nn.CrossEntropyLoss()(test_outputs, y_test)
    test_report = classification_report(y_test.numpy(), test_predicted.numpy(), digits=4, zero_division=1)

    print('Test Loss: {:.4f}'.format(test_loss.item()))
    print('Test Classification Report:\n', test_report)
```

Figure 64: Codes for Testing the GPT RNN Model

The test set is prepared by converting it to PyTorch tensors and unsqueezing the data to match the dimensions required by the model. Then, the saved best model with the lowest validation loss is loaded and evaluates its performance on the test set. The test_outputs are obtained by feeding the test data to the trained model, the outputs are converted to predictions using torch.max() and the accuracy score of the predictions is computed using accuracy_score() from scikit-learn. The test loss is computed using nn.CrossEntropyLoss() and a classification report of the test set is generated using classification_report() from scikit-learn.

The test results from our RNN implementation are shown below:

Test Loss: 0.6956 **Test Classification Report**				
	Precision	**Recall**	**F1-Score**	**Support**
0	1.00	0.00	0.00	4004
1	0.40	1.00	0.57	2612
Accuracy			0.39	6616
Macro Average	0.70	0.50	0.28	6616
Weighted Average	0.76	0.40	0.22	6616

Figure 65: Classification Report for Test GPT RNN

The relevant results to observe are from the “1” row and the “Accuracy” row.

GPT-2 CNN

The data was split into training and testing datasets, with the testing set consisting of 25% of the original dataset

Model Configuration

```python
import torch
import torch.nn as nn
import torch.optim as optim
import os
from torch.utils.data import DataLoader, TensorDataset
from sklearn.metrics import accuracy_score

# Define the CNN model
class GPTEmbeddingCNN(nn.Module):
    def __init__(self):
        super(GPTEmbeddingCNN, self).__init__()
        self.conv1 = nn.Conv1d(768, 128, kernel_size=3, stride=1, padding=2)
        self.conv2 = nn.Conv1d(768, 128, kernel_size=4, stride=1, padding=3)
        self.conv3 = nn.Conv1d(768, 128, kernel_size=5, stride=1, padding=4)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.4)
        self.max_pool = nn.AdaptiveMaxPool1d(1)
        self.fc = nn.Linear(128 * 3, 1)

    def forward(self, x):
        if x.dim() == 2:
            x = x.unsqueeze(1)  # Add the sequence dimension if it's missing
        x = x.permute(0, 2, 1)  # Swap the feature and sequence dimensions
        x1 = self.conv1(x)
        x1 = self.relu(x1)
        x1 = self.dropout(x1)
        x1 = self.max_pool(x1).squeeze()

        x2 = self.conv2(x)
        x2 = self.relu(x2)
        x2 = self.dropout(x2)
        x2 = self.max_pool(x2).squeeze()

        x3 = self.conv3(x)
        x3 = self.relu(x3)
        x3 = self.dropout(x3)
        x3 = self.max_pool(x3).squeeze()

        x = torch.cat([x1, x2, x3], dim=1)
        x = self.fc(x)
        x = torch.sigmoid(x)
        return x.squeeze()
```

Figure 66: Codes for GPT CNN

In addition to the model hyperparameters stated in part 7.5 for CNN, our implementation also has an adaptive max pooling layer. The adaptive max pooling layer pools the output of each convolutional layer across the time dimension (which is the second dimension) to get the maximum value across all time steps, resulting in three feature vectors of size 128. These three feature vectors are concatenated along the feature dimension to get a feature vector of size 384, which is then fed into a fully connected layer (fc) with one output neuron.

Data Preparation & Model Initialization

```
# Split the training data into training and validation sets
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size=0.2, random_state=42)

# Create train and test datasets
train_dataset = TensorDataset(torch.tensor(x_train, dtype=torch.float32), torch.tensor(y_train.values, dtype=torch.float32))
val_dataset = TensorDataset(torch.tensor(x_val, dtype=torch.float32), torch.tensor(y_val.values, dtype=torch.float32))
test_dataset = TensorDataset(torch.tensor(x_test, dtype=torch.float32), torch.tensor(y_test.values, dtype=torch.float32))

# Create data loaders
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)

# Initialize the model, loss function, and optimizer
model = GPTEmbeddingCNN()
criterion = nn.BCELoss()
optimizer = optim.AdamW(model.parameters(), lr=5e-5, eps=1e-8)
```

Figure 67: Dataset Preparation for GPT CNN Model

Firstly, we split the training data into two sets, a training set and a validation set, using the train_test_split() function from the scikit-learn library. Then, we create PyTorch TensorDataset objects for the training, validation, and test datasets using the torch.tensor() function, and put them into data loaders using DataLoader. The data loaders make it easier to handle the data by creating batches of data to be fed into the model during training. Finally, we initialise the model using the hyperparameters stated for CNN in part 7.5.

Model Training

```
from sklearn.metrics import classification_report

# Create a directory for saving the model
model_dir = "models"
os.makedirs(model_dir, exist_ok=True)

# Define a function to calculate metrics and print the classification report
def get_classification_report(data_loader, loader_type):
    model.eval()
    y_true = []
    y_pred = []

    with torch.no_grad():
        for batch_x, batch_y in data_loader:
            outputs = model(batch_x)
            predictions = (outputs > 0.5).int()
            y_true.extend(batch_y.tolist())
            y_pred.extend(predictions.tolist())

    report = classification_report(y_true, y_pred, output_dict=True, zero_division=0)
    print(f"{loader_type} Classification Report (Epoch {epoch+1}):")
    print(classification_report(y_true, y_pred, output_dict=False, zero_division=0))
    return report

# Train the model and calculate metrics for each epoch
num_epochs = 10
best_val_loss = float("inf")
best_epoch = 0

for epoch in range(num_epochs):
    # Training phase
    model.train()
    train_loss = 0

    for batch_x, batch_y in train_loader:
        optimizer.zero_grad()
        outputs = model(batch_x)
        loss = criterion(outputs, batch_y)
        loss.backward()
        optimizer.step()

        train_loss += loss.item() * batch_x.size(0)

    train_loss /= len(train_loader.dataset)
    print(f"Epoch [{epoch+1}/{num_epochs}] Train Loss: {train_loss:.4f}")
```

```
    # Validation phase
    model.eval()
    val_loss = 0

    with torch.no_grad():
        for batch_x, batch_y in val_loader:
            outputs = model(batch_x)
            loss = criterion(outputs, batch_y)

            val_loss += loss.item() * batch_x.size(0)

    val_loss /= len(val_loader.dataset)

    # Print the classification reports for training and validation sets
    train_report = get_classification_report(train_loader, "Train")
    print(f"Epoch [{epoch+1}/{num_epochs}] Val Loss: {val_loss:.4f}")
    val_report = get_classification_report(val_loader, "Validation")

    # Save the model with the best validation accuracy
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        best_epoch = epoch + 1
        torch.save(model.state_dict(), os.path.join(model_dir, "best_GPT_+_CNN_model.pt"))
        print(f"Model saved for epoch {best_epoch}")
print(f"Best model saved at epoch {best_epoch} with validation loss {best_val_loss:.4f}")
```

Figure 68: Model Training for GPT CNN

We define a function for calculating classification metrics and printing the classification report. We then train the model for 10 epochs and calculate the training and validation losses. The function gets called twice to calculate classification metrics and print the classification report for both the training and validation sets. The best model based on validation loss is saved in a separate file in the "models" directory. The classification report function calculates the classification report for the given data loader, where the model is set to evaluation mode to disable gradient calculation. The classification report contains metrics such as precision, recall, and F1 score for each class, as well as the macro and micro average metrics for the overall performance. The training results from our RNN implementation are shown below:

Train Loss: 0.5183 **Train Classification Report**				
	Precision	**Recall**	**F1-Score**	**Support**
0	0.74	0.93	0.82	9654
1	0.81	0.49	0.61	6222
Accuracy			0.75	15876
Macro Average	0.77	0.71	0.71	15876
Weighted Average	0.77	0.75	0.74	15876

Figure 69: Classification Report for Train GPT CNN

Validation Loss: 0.5435 **Validation Classification Report**				
	Precision	**Recall**	**F1-Score**	**Support**
0	0.73	0.92	0.81	2387
1	0.81	0.48	0.60	1583
Accuracy			0.75	3970
Macro Average	0.77	0.70	0.71	3970
Weighted Average	0.76	0.75	0.73	3970

Figure 70: Classification Report for Validation GPT CNN

Testing the Model

```
# Load the best model
model.load_state_dict(torch.load(os.path.join(model_dir, "best_GPT_+_CNN_model.pt")))

# Define a function to calculate metrics and print the classification report
def get_classification_report(data_loader, loader_type):
    model.eval()
    y_true = []
    y_pred = []

    with torch.no_grad():
        for batch_x, batch_y in data_loader:
            outputs = model(batch_x)
            predictions = (outputs > 0.5).int()
            y_true.extend(batch_y.tolist())
            y_pred.extend(predictions.tolist())

    report = classification_report(y_true, y_pred, output_dict=True, zero_division=0)
    print(f"{loader_type} Classification Report:")
    print(classification_report(y_true, y_pred, output_dict=False, zero_division=0))
    return report

# Print the classification report for the test dataset
test_loss = 0
with torch.no_grad():
    for batch_x, batch_y in test_loader:
        outputs = model(batch_x)
        loss = criterion(outputs, batch_y)
        test_loss += loss.item() * batch_x.size(0)

test_loss /= len(test_loader.dataset)

print(f"Test Loss: {test_loss:.4f}")
test_report = get_classification_report(test_loader, "Test")
```

Figure 71: Codes for GPT CNN

We load the best model that was saved during the training phase. Then, we define a function that calculates the classification report for a given data loader and loader type (e.g., train, validation, test). This function evaluates the model on the given data loader, calculates the predictions and true values, and then generates the classification report using the classification_report function from scikit-learn. The classification_report function calculates precision, recall, F1-score, and support for each class and for all classes combined. The function then prints the classification report and returns it as a dictionary. Finally, we evaluate the loaded model on the test dataset, calculate the test loss, and print the classification report for the test set using the get_classification_report function.

Test Loss: 0.5411 **Test Classification Report**				
	Precision	**Recall**	**F1-Score**	**Support**
0	0.73	0.92	0.82	4004
1	0.80	0.48	0.60	2612
Accuracy			0.75	6616
Macro Average	0.77	0.70	0.71	6616
Weighted Average	0.76	0.75	0.73	6616

Figure 72: Classification Report for Test GPT CNN

The relevant results to observe are from the “1” row and the “Accuracy” row.

GPT-2 LSTM

Our GPT-2 LSTM does not follow the hyperparameters stated in part 7.5 due to certain constraints of its datatype.

```python
class GPTLSTMModel:
    def __init__(self, input1, input2):

        # Define the LSTM model architecture
        self.model = Sequential()
        self.model.add(LSTM(128, input_shape=(input1, input2)))
        self.model.add(Dropout(0.5))
        self.model.add(Dense(1, activation="sigmoid"))

    def compile_model(self):
        self.model.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"])

    def train_model(self, X_train, y_train, X_val, y_val, epochs=10, batch_size=32):
        self.model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=epochs, batch_size=batch_size)

    def evaluate_model(self, X_test, y_test):
        loss, accuracy = self.model.evaluate(X_test, y_test)
        score = (accuracy * 100)
        return score

    def predict(self, X_new):
        # Make a prediction for a new input
        prediction = self.model.predict(X_new)
        return prediction
```

Figure 73: Configuration of GPT LSTM Model

Based on this code, we add a LSTM layer to the model with 128 units and the input shape of the input data. In addition, the model uses the binary cross entropy loss function with Adam Optimizer.

```python
model = GPTBiLSTMModel(X_train.shape[1], X_train.shape[2])
model.compile_model()

batch_size = 10
epochs = 50
model.train_model(X_train, y_train, X_val, y_val, epochs=epochs, batch_size=batch_size)

# Evaluate the model on the test set
score = model.evaluate_model(X_test, y_test)
print(f"Test accuracy: {score:.2f}%")
```

Figure 74: Initialisation of GPT LSTM Model

In addition, we also initialise a batch size of 10 and run the model for 50 epochs. After training the model, we then predict the result on X_Test to obtain Y_Pred. We then obtain the classification report based on Y_Test and Y_Pred in the figure below.

Test Classification Report				
	Precision	**Recall**	**F1-Score**	**Support**
0	0.65	0.88	0.75	3160
1	0.62	0.30	0.41	2133
Accuracy			0.64	5293
Macro Average	0.64	0.59	0.58	5293
Weighted Average	0.64	0.64	0.61	5293

Figure 75: Classification Report for GPT LSTM

The relevant results to observe are from the "1" row and the "Accuracy" row.

GPT-2 Bi-LSTM

```
class GPTBiLSTMModel:
    def __init__(self, input1, input2):

        # Define the BiLSTM model architecture
        self.model = Sequential()
        self.model.add(Bidirectional(LSTM(128), input_shape=(input1, input2)))
        self.model.add(Dropout(0.5))
        self.model.add(Dense(1, activation="sigmoid"))

    def compile_model(self):
        self.model.compile(loss="binary_crossentropy", optimizer="adam", metrics=["accuracy"])

    def train_model(self, X_train, y_train, X_val, y_val, epochs=10, batch_size=32):
        self.model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=epochs, batch_size=batch_size)

    def evaluate_model(self, X_test, y_test):
        loss, accuracy = self.model.evaluate(X_test, y_test)
        score = (accuracy * 100)
        return score

    def predict(self, X_new):
        # Make a prediction for a new input
        prediction = self.model.predict(X_new)
        return prediction
```

Figure 76: Codes for GPT Bi-LSTM

Based on the code, it is similar to the LSTM code; however, we add the Bi-LSTM with 128 units of the input and output data. The other configurations remain constant. Thus, by training on X Test and test on X Test, we obtain the following results as shown in the figure below.

Test Classification Report				
	Precision	**Recall**	**F1-Score**	**Support**
0	0.63	0.97	0.76	3160
1	0.75	0.14	0.24	2133
Accuracy			0.64	5293
Macro Average	0.69	0.59	0.58	5293
Weighted Average	0.68	0.64	0.55	5293

Figure 77: Classification Report of Test GPT Bi-LSTM

The relevant results to observe are from the "1" row and the "Accuracy" row.

Based on the results obtained from the 4 Deep Learning models, we can conclude that CNN performs best with an F1-score of 0.60. We can also note that RNN performs strongly as well because it has a perfect Recall score of 1 and Recall is one of the key metrics we use to evaluate model performance. However, even though RNN has a recall of 1, it has a significantly worse precision. Hence we find that F1-score is still a better measure in this case because it balances the tradeoff between precision and recall for overall performance. Therefore, **the best Deep Learning model to implement with GPT is CNN.** One possible reason that the GPT language model produces best results with a CNN deep learning model is that the CNN architecture is good at learning local patterns in data, while the GPT model is good at capturing global patterns and dependencies in language. By combining the strengths of both models, the CNN can effectively identify and extract relevant features from the input data, while the GPT model can use these features to generate more accurate predictions.

10.4 Conclusion of GPT

Model/Metric	Accuracy	Precision	Recall	F1-Score
RNN	0.39	0.39	1.0	0.5661
LSTM	0.64	0.62	0.30	0.41
CNN	0.75	0.80	0.48	0.60
BI-LSTM	0.64	0.75	0.14	0.24
LR	0.73	0.35	0.94	0.51
SVM	0.77	0.47	0.90	0.62
GNB	0.60	0.30	0.49	0.38

GB	0.75	0.48	0.81	0.60
RF	0.75	0.43	0.88	0.58
DT	0.64	0.55	0.55	0.55

The table above gives us an overview of performance of GPT-2 embedding together with the 6 machine learning models and the 4 deep learning models implemented. We can conclude that the best performing combination with Word2Vec will be SVM as it has the highest F1-score of 0.62.

11. Bidirectional Encoder Representations from Transformers (BERT)

11.1 BERT Word Embeddings

BERT was developed by Google in 2018 with an aim to improve contextual understanding of unlabeled text access to a broad range of tasks by learning to predict text that might come before and after other text.

As opposed to other word embedding method such as Word2Vec, GloVe etc, which are directional in nature, BERT learns information from both the left and the right side of a token's context during the training phase. The bi-directionality of a model is important in order to truly understand the meaning of language as there are situations where the same word holds different meanings, depending on the context. Given the examples below, Word2Vec will generate the same single vector for the word bank for both of the sentences but BERT will generate two different vectors for the word bank due to it having different context. (Sanad, 2019)

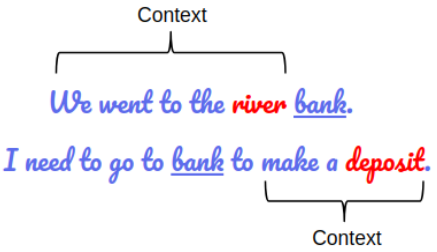


Figure 78: Contextual Word

For this project, we leveraged on Hugging Face which is a popular open-source library for natural language processing (NLP) tasks and it provides several pre-trained models for different NLP tasks, including BERT. The two pretrained models that were used are BertTokenizer and BertModel.

BertModel is used to obtain the corresponding embedding. However, the data need to be first converted into an appropriate format before passing it into the BertModel. BertModel requires input text to be tokenized, which means that it needs to be split into individual tokens or subwords. The BertTokenizer from the Hugging Face library is a tool that performs this tokenization step specifically for BERT models.

In hindsight, the BertTokenizer takes input text and returns a list of tokens, where each token is a subword or word that has been split into smaller pieces to form a vocabulary that the BERT model can understand. The BertTokenizer also performs some additional preprocessing steps, such as adding special tokens to the beginning and end of the input sequence to indicate the start and end of a sentence. This will be further elaborated in the subsequent paragraphs.

(Prakash, 2021)

```
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased', do_lower_case=True)
```

```
def bert_tokenizer(data):
    input_ids = []
    attention_masks = []
    for sent in data:
        encoded_sent = tokenizer.encode_plus(
            text=sent,
            add_special_tokens=True,        # Add `[CLS]` and `[SEP]` special tokens
            max_length=MAX_LEN,             # Choose max length to truncate/pad
            pad_to_max_length=True,         # Pad sentence to max length
            return_attention_mask=True      # Return attention mask
            )
        input_ids.append(encoded_sent.get('input_ids'))
        attention_masks.append(encoded_sent.get('attention_mask'))

    # Convert lists to tensors
    input_ids = torch.tensor(input_ids)
    attention_masks = torch.tensor(attention_masks)

    return input_ids, attention_masks
```

Figure 79: BertTokenizer

The process includes (1) breaking down input text into tokens, (2) adding special tokens, such as [CLS] and [SEP], to indicate the beginning and end of the text, (3) encoding the tokens into numerical values which are obtained by looking up each token in BERT's vocabulary and mapping it to a unique integer ID, (4) padding the input sequence to ensure that all input sequences have the same length, and (4) creating an attention mask to differentiate between padding tokens and input text, The attention mask is important for allowing the BERT model to ignore the padding tokens during training and inference. By setting the attention scores to 0 for padding tokens, the model learns to focus only on the relevant input tokens when making predictions. (Au Yeung, 2020)

The output of tokenizer is a dictionary containing two keys, (1) input ids and (2) attention mask, which will then be passed into BERTModel from Hugging Face pretrained “bert-based-uncased” model.

The training process for BERT involves training the BERT model on the corpus of text data in an unsupervised manner. This is done by using a masked language modelling (MLM) task and a next sentence prediction (NSP) task to predict the masked words and to determine whether two sentences are related or not, respectively. During MLM, 15% of the tokens from the sequence were randomly masked and then the model was trained to predict the missing words based on the context of the surrounding words. The NSP task

involves labelling half the data as “isNext” where Sentence B of the input sequence is just the next sentence of Sentence A from the dataset corpus and labelling the other half of the data is “notNext” where Sentence B is not next to Sentence A but any random sentence from the corpus dataset. The model is trained to determine whether two sentences are adjacent or not, based on the text data. (Devlin et al., 2019)

It is worth noting that BERT does not directly output a prediction for the specific task. Instead, it generates a sequence of contextualised embeddings that encode the meaning of the input text. Therefore, in the section 7.3.1 and 7.3.2 below, we will be discussing how this embedding is being passed into Traditional Classifiers and Deep Learning Model to generate prediction.

11.2 BERT Traditional Classifiers Results

Model/Metric	Train Accuracy Mean	Test Accuracy	Precision	Recall	F1-Score
SVM	0.77	0.77	0.52	0.84	0.64
GB	0.75	0.75	0.50	0.79	0.61
LR	0.75	0.75	0.57	0.72	0.64
RF	0.75	0.75	0.46	0.82	0.59
GNB	0.70	0.70	0.62	0.62	0.62
DT	0.65	0.65	0.55	0.55	0.55

As we can see from the table above, Support Vector Machine (SVM) achieves the highest train accuracy mean, test accuracy as well as Recall score. It can also be seen that Logistic Regression has a slightly higher f1-score than SVM. However, SVM has a significantly higher Recall which as aforementioned, was more important as the consequences of not being able to correctly identify someone with depression is much higher.

One possible reason why we think SVM is able to outperform other models is due to it finding the best hyperplane that separates the data points into different classes. The hyperplane is chosen to maximise the margin, which is the distance between the hyperplane and the nearest data points from each class, also called the support vectors. However, one trade-off is it usually takes longer to train the SVM model.

11.3 BERT Deep Learning Implementation & Results

Splitting Data

After tokenization, the dataset was splitted into 3 sets - train, validation and test. With the training proportion being 0.72 to 0.08 to 0.2.

Train - Validation - Test

```
X = df_cleaned.Message
y = df_cleaned.Label
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=88)
```

```
X_train, X_valid, y_train, y_valid = train_test_split(X_train, y_train, test_size=0.1, stratify=y_train, random_state=88)
```

Figure 80: Train-Validation-Test split

It's important to split the data into three parts because using the same data for both training and testing can lead to overfitting, which occurs when the model performs well on the training data but poorly on new, unseen dataset. The validation set helps prevent overfitting by providing a way to measure the model's performance on data that it has not been trained on, and the test set provides a final check on the model's generalisation performance. (Agrawal, 2021)

Data preparation

After the data has been splitted and has been tokenized, we move on to prepare the data in a format that is suited for PyTorch which includes creating a data loader for each dataset as seen from the image below.

```
# Convert target columns to pytorch tensors format
train_labels = torch.from_numpy(y_train.to_numpy())
val_labels = torch.from_numpy(y_valid.to_numpy())
test_labels = torch.from_numpy(y_test.to_numpy())
```

```
batch_size = 32
```

```
# Create the DataLoader for our train set
train_data = TensorDataset(train_inputs, train_masks, train_labels)
train_sampler = RandomSampler(train_data)
train_dataloader = DataLoader(train_data, sampler=train_sampler, batch_size=batch_size)

# Create the DataLoader for our validation set
val_data = TensorDataset(val_inputs, val_masks, val_labels)
val_sampler = SequentialSampler(val_data)
val_dataloader = DataLoader(val_data, sampler=val_sampler, batch_size=batch_size)

# Create the DataLoader for our test set
test_data = TensorDataset(test_inputs, test_masks, test_labels)
test_sampler = SequentialSampler(test_data)
test_dataloader = DataLoader(test_data, sampler=test_sampler, batch_size=batch_size)
```

Figure 81: Codes for Data Preparation for BERT

The dataloader takes a dataset object as input, along with various parameters such as batch size which we set to 32. It then loads data from the dataset in batches of size 32 and returns them as a tuple of inputs and targets. During training, these batches of data are passed to the model in order to compute the loss and update the parameters.

In deep learning, large datasets are often used to train models, and these datasets can be too large to fit in memory all at once. Therefore, to address this issue, the Dataloaders provide a way to load data in batches, allowing for efficient use of memory during training. (Nolet, 2022)

Training and Validation process

Once the data has been readily prepared, the next time would be defining the training and validation process.

```
for epoch_i in range(epochs):
    print("-"*10)
    print("Epoch : {}".format(epoch_i+1))
    print("-"*10)
    print("-"*38)
    print(f"{'BATCH NO.':^7} | {'TRAIN LOSS':^12} | {'ELAPSED (s)':^9}")
    print("-"*38)

    # Measure the elapsed time of each epoch
    t0_epoch, t0_batch = time.time(), time.time()

    # Reset tracking variables at the beginning of each epoch
    total_loss, batch_loss, batch_counts = 0, 0, 0

    ###TRAINING###

    # Put the model into the training mode
    model.train()

    for step, batch in enumerate(train_dataloader):
        batch_counts +=1

        b_input_ids, b_attn_mask, b_labels = tuple(t.to(device) for t in batch)

        # Zero out any previously calculated gradients
        model.zero_grad()

        # Perform a forward pass and get logits.
        logits = model(b_input_ids, b_attn_mask)

        # Compute loss and accumulate the loss values
        loss = loss_fn(logits, b_labels)
        batch_loss += loss.item()
        total_loss += loss.item()

        # Perform a backward pass to calculate gradients
        loss.backward()

        # Clip the norm of the gradients to 1.0 to prevent "exploding gradients"
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

        # Update model parameters:
        # fine tune BERT params and train additional dense layers
        optimizer.step()
        # update learning rate
        scheduler.step()
```

Figure 82: Training Process code

The above image shows the code of the training process. During training, the model is presented with batches of input examples where the process of forward pass will be executed. During the forward pass, the input data is fed through the layers of the neural network, and the output of each layer is calculated and passed as input to the next layer until the final output is produced.

Next, the loss will be calculated and loss.backward() is called. “loss.backward()” is a backward propagation step in PyTorch, which is used to compute the gradients of the loss function with respect to the model parameters. The backward propagation step is a key component of neural network training, as it allows the gradients to flow backwards through the model and compute the necessary updates to the parameters that minimise the loss.

After computing, optimizer.step() is called which then uses the computed gradients to update the model parameters. The optimizer is also responsible for selecting the learning rate and controlling the optimization algorithm used to update the parameters. Different optimizers have different update rules, such as gradient descent, Adam, RMSProp, etc. As seen from the image below, we have defined the optimization algorithm to be Adam and the initial learning rate to be “5e-5”.

```python
def initialize_bert_model(epochs):
    # Instantiate Bert Classifier
    bert_classifier = Bert_Classifier(freeze_bert=False)

    bert_classifier.to(device)

    # Set up optimizer
    optimizer = AdamW(bert_classifier.parameters(),
                      lr=5e-5,    # learning rate, set to default value
                      eps=1e-8    # decay, set to default value
                      )

    ### Set up learning rate scheduler ###

    # Calculate total number of training steps
    total_steps = len(train_dataloader) * epochs

    # Defint the scheduler
    scheduler = get_linear_schedule_with_warmup(optimizer,
                                                num_warmup_steps=0, # Default value
                                                num_training_steps=total_steps)
    return bert_classifier, optimizer, scheduler
```

Figure 83: Optimizer

At the end of each training epoch, we will call the ‘scheduler.step()” function which is a method used to update the learning rate based on the current epoch. The learning rate is a hyperparameter that determines how much the model's parameters are updated during each step of training. (Brownlee, 2019) A higher learning rate can cause the model to converge too quickly to a suboptimal solution, while a lower learning rate can lead to slower convergence but more stable training. Learning rate scheduling is a technique used to adjust the learning rate over the course of training. There are various strategies for learning rate scheduling, such as stepwise decay, Linear Warmup, exponential decay, or cyclic scheduling etc. As seen from the image above, we have defined a Linear Warmup where the learning rate increases linearly from a low rate to a constant rate thereafter. This reduces volatility in the early stages of training. (Linear Warmup Explained, n.d.)

The training process continues for 10 epochs, during which the weights are updated multiple times to improve the performance of the model on the task-specific dataset. In each epoch, we will also test the model against the validation set and the final model that will be applied to the test set will be the one that yields the best performance on a held-out validation dataset. In this case, we define best performance as the one with the lowest validation loss. This process can be seen from the image below.

```
for batch in val_dataloader:
    batch_input_ids, batch_attention_mask, batch_labels = tuple(t.to(device) for t in batch)

    # We do not want to update the params during the evaluation,
    # So we specify that we dont want to compute the gradients of the tensors
    # by calling the torch.no_grad() method
    with torch.no_grad():
        logits = model(batch_input_ids, batch_attention_mask)

    loss = loss_fn(logits, batch_labels)

    val_loss.append(loss.item())

    # Get the predictions starting from the logits (get index of highest logit)
    preds = torch.argmax(logits, dim=1).flatten()

    # Calculate the validation accuracy
    accuracy = (preds == batch_labels).cpu().numpy().mean() * 100
    val_accuracy.append(accuracy)

# Compute the average accuracy and loss over the validation set
val_loss = np.mean(val_loss)
val_accuracy = np.mean(val_accuracy)

valid_stats.append(
    {
        'Val Loss': val_loss,
        'Val Accur.': val_accuracy,
    }
)


# Print performance over the entire training data
time_elapsed = time.time() - t0_epoch
print("-"*61)
print(f"{'AVG TRAIN LOSS':^12} | {'VAL LOSS':^10} | {'VAL ACCURACY (%)':^9} | {'ELAPSED (s)':^9}")
print("-"*61)
print(f"{avg_train_loss:^14.6f} | {val_loss:^10.6f} | {val_accuracy:^17.2f} | {time_elapsed:^9.2f}")
print("-"*61)
print("\n")

if valid_stats[epoch_i]['Val Loss'] < best_valid_loss:
    best_valid_loss = valid_stats[epoch_i]['Val Loss']
    # save best model for use later
    torch.save(model, f'bert-{model_name}-model.pt')  # torch save
```

Figure 84: Validation Process code

In the following sub section, we will be diving more into details of how 3 deep learning layers have been defined.

Recurrent Neural Network (RNN)

By combining BERT and RNNs in this way, the hybrid model can benefit from the contextualised word embeddings learned by BERT, while also leveraging the ability of RNNs to model sequential data.

```
class Bert_Classifier(nn.Module):
    def __init__(self, freeze_bert=False):
        super(Bert_Classifier, self).__init__()

        input_dim = 768
        hidden_dim = 50
        layer_dim =1
        output_dim =2

        # Instantiate BERT model
        self.bert = BertModel.from_pretrained('bert-base-uncased')
        self.rnn = nn.RNN(input_dim, hidden_dim, layer_dim, batch_first=True, nonlinearity='relu')
        self.linear = nn.Linear(hidden_dim, output_dim)

        # Add possibility to freeze the BERT model
        # to avoid fine tuning BERT params (usually leads to worse results)
        if freeze_bert:
            for param in self.bert.parameters():
                param.requires_grad = False

    def forward(self, input_ids, attention_mask):
        # Feed input data to BERT
        outputs = self.bert(input_ids=input_ids,
                            attention_mask=attention_mask)

        sequence_output = outputs[0]
        sequence_output, _ = self.rnn(sequence_output)
        linear_output = self.linear(sequence_output[:, -1])

        return linear_output
```

Figure 85: Codes for BERT RNN

The output of the BERT model is then fed into an RNN-based model that takes into account the context of neighbouring words in the sequence and then fed into the linear layer to produce a predicted label. The result can be seen below.

Train Classification Report (BERT + RNN)			
Batch No.	**Train Loss**	**Elapsed (s)**	
100	0.010640	16.86	
200	0.018153	16.91	
300	0.019014	17.04	
400	0.020871	16.89	
500	0.018794	16.93	
595	0.023659	15.82	
Average Train Loss	**Validation Loss**	**Validation Accuracy**	**Elapsed (s)**
0.018465	1.714300	80.39	103.79

Classification Report for BERT + RNN				
	Precision	**Recall**	**F1-Score**	**Support**
0	0.82	0.82	0.82	3209
1	0.73	0.73	0.73	2084
Accuracy			0.78	5293
Macro Average	0.77	0.77	0.77	5293
Weighted Average	0.78	0.78	0.78	5293

Figure 86: Output of BERT RNN

Long-Short Term Memory (LSTM)

LSTM stands for Long Short-Term Memory, which is a type of recurrent neural network (RNN) architecture designed to handle the vanishing gradient problem of traditional RNNs.

```python
class BERT_LSTM(nn.Module):
    def __init__(self, freeze_bert=False):
        super(BERT_LSTM, self).__init__()
        # Specify hidden size of BERT, hidden size of our classifier, and number of labels
        D_in, H, D_out = 768, 50, 2

        # Instantiate BERT model
        self.bert = BertModel.from_pretrained('bert-base-uncased')
        self.lstm = nn.LSTM(D_in, H, batch_first=True, bidirectional=False)
        self.linear = nn.Linear(H , D_out)

        # Freeze the BERT model
        if freeze_bert:
            for param in self.bert.parameters():
                param.requires_grad = False

    def forward(self, input_ids, attention_mask):
        # Move input tensors to CUDA device with index 0
        input_ids = input_ids.to('cuda:0')
        attention_mask = attention_mask.to('cuda:0')

        # Feed input to BERT
        outputs = self.bert(input_ids=input_ids, attention_mask=attention_mask)
        sequence_output = outputs[0]
        #print("sequence_output size", sequence_output.size())

        # Move sequence_output tensor to CUDA device with index 0
        sequence_output = sequence_output.to('cuda:0')

        sequence_output, _ = self.lstm(sequence_output)
        #print("lstm size", sequence_output.size())

        linear_output = self.linear(sequence_output[:, -1])
        #print("linear_output size", linear_output.size())

        return linear_output
```

Figure 87: Codes of BERT LSTM

On top of the linear layer seen in the previous section, we added a LSTM layer with bidirectional being set to false. Similarly, the output from BERT would be fit into the LSTM layer and then the linear layer to produce a predicted label. The result can be seen below.

Train Classification Report (BERT + LSTM)			
Batch No.	**Train Loss**	**Elapsed (s)**	
100	0.025580	13.52	
200	0.016974	13.35	
300	0.022433	13.36	
400	0.015029	13.37	
500	0.025836	13.38	
595	0.020117	12.63	
Average Train Loss	**Validation Loss**	**Validation Accuracy**	**Elapsed (s)**
0.021010	1.050817	0.80	82.17

Classification Report for BERT + LSTM				
	Precision	**Recall**	**F1-Score**	**Support**
0	0.82	0.87	0.84	3209
1	0.78	0.70	0.74	2084
Accuracy			0.80	5293
Macro Average	0.80	0.79	0.79	5293
Weighted Average	0.80	0.80	0.80	5293

Figure 88:Output of BERT LSTM

Bidirectional Long-Short Term Memory (Bi-LSTM)

BERT is a purely transformer-based model that only captures contextual information in the forward direction. However, adding a Bi-LSTM to BERT allows the model to capture contextual information in both forward and backward directions, which can potentially improve its ability to understand the relationships between words in the input text.

```
class BERT_BILSTM(nn.Module):
    def __init__(self, freeze_bert=False):
        super(BERT_BILSTM, self).__init__()
        # Specify hidden size of BERT, hidden size of our classifier, and number of labels
        D_in, H, D_out = 768, 50, 2

        # Instantiate BERT model
        self.bert = BertModel.from_pretrained('bert-base-uncased')
        self.lstm = nn.LSTM(D_in, H, batch_first=True, bidirectional=True)
        self.linear = nn.Linear(H*2 , D_out)

        # Freeze the BERT model
        if freeze_bert:
            for param in self.bert.parameters():
                param.requires_grad = False

    def forward(self, input_ids, attention_mask):
        # Feed input to BERT
        outputs = self.bert(input_ids=input_ids,attention_mask=attention_mask)
        sequence_output = outputs[0]
        sequence_output, _ = self.lstm(sequence_output)
        linear_output = self.linear(sequence_output[:, -1])

        return linear_output
```

Figure 89:Codes of BERT Bi-LSTM

On top of the linear layer seen in the previous section, we added a LSTM layer with bidirectional being set to true to indicate that it is a Bi-LSTM layer. Similarly, the output from BERT would be fit into the Bi-LSTM layer and then the linear layer to produce a predicted label. The result can be seen below.

Train Classification Report (BERT + BILSTM)			
Batch No.	**Train Loss**	**Elapsed (s)**	
100	0.023651	13.73	
200	0.019016	13.57	
300	0.015615	13.56	
400	0.019833	13.56	
500	0.015792	13.58	
595	0.019105	12.81	
Average Train Loss	**Validation Loss**	**Validation Accuracy**	**Elapsed (s)**
0.018841	1.133959	0.81	0.83

Classification Report for BERT + Bi-LSTM				
	Precision	**Recall**	**F1-Score**	**Support**
0	0.79	0.89	0.84	3209
1	0.80	0.64	0.71	2084
Accuracy			0.80	5293
Macro Average	0.80	0.77	0.78	5293
Weighted Average	0.80	0.80	0.79	5293

Figure 90: Output of BERT Bi-LSTM

Convolutional Neural Network (CNN)

Adding a CNN (Convolutional Neural Network) to BERT can potentially improve its performance in several ways.

BERT is a sequential model that takes into account the context of each word in the input text, but it does not consider the global context of the entire sentence. CNNs, on the other hand, are designed to capture global patterns in the input data, making them well-suited for capturing higher-level features in the text. By adding a CNN to BERT, it allows us to capture both the local context of individual words and the global context of the entire sentence, thereby potentially improving the model's ability to understand and classify the text.

```
class Bert_CNN(nn.Module):

    def __init__(self, freeze_bert=False):
        super().__init__()
        self.bert = BertModel.from_pretrained('bert-base-uncased',output_hidden_states=True).cuda()
        output_channel = 16  # number of kernels
        num_classes = 2  # number of targets to predict
        dropout = 0.4  # dropout value
        embedding_dim = 768  # length of embedding dim

        ks = 3  # three conv nets here

        # input_channel = word embeddings at a value of 1; 3 for RGB images
        input_channel = 4  # for single embedding, input_channel = 1

        # 3 convolutional nets
        self.conv1 = nn.Conv2d(input_channel, output_channel, (3, embedding_dim), padding=(2, 0), groups=4)
        self.conv2 = nn.Conv2d(input_channel, output_channel, (4, embedding_dim), padding=(3, 0), groups=4)
        self.conv3 = nn.Conv2d(input_channel, output_channel, (5, embedding_dim), padding=(4, 0), groups=4)

        # apply dropout
        self.dropout = nn.Dropout(dropout)

        # fully connected layer for classification
        # 3x conv nets * output channel
        self.fc1 = nn.Linear(ks * output_channel, num_classes)

        # Freeze the BERT model
        if freeze_bert:
            for param in self.bert.parameters():
                param.requires_grad = False

    def forward(self,inputs, mask):
        outputs = self.bert(input_ids=inputs,attention_mask=mask)

        x = outputs[2]  # get hidden layers
        x = torch.stack(x, dim=1)  # stack the layers
        x = x[:, -4:]

        x = [F.relu(self.conv1(x)).squeeze(3), F.relu(self.conv2(x)).squeeze(3), F.relu(self.conv3(x)).squeeze(3)]
        # max-over-time pooling; # (batch, channel_output) * ks
        x = [F.max_pool1d(i, i.size(2)).squeeze(2) for i in x]
        # concat results; (batch, channel_output * ks)
        x = torch.cat(x, 1)
        # add dropout
        x = self.dropout(x)
        # generate logits (batch, target_size)
        logit = self.fc1(x)
        return logit
```

Figure 90: Codes of BERT CNN

On top of the linear layer seen in the previous section, three convolutional nets were added. After getting the output from the BERT model, the last four hidden layers are selected and passed into the convolutional layer. Each kernel takes the output of the last four hidden layers of BERT as 4 different channels and applies convolution operation on it. After that, the output is passed through ReLU and Max-Pooling operation in which the output is concatenated and passed through the Linear layer to get the final binary label. This model was trained for 10 epochs with a learning rate of 5e-5. The result can be seen below.

Train Classification Report (BERT + CNN)			
Batch No.	**Train Loss**	**Elapsed (s)**	
100	0.015276	56.57	
200	0.010901	56.01	
300	0.012848	55.99	
400	0.016137	55.98	
500	0.019463	55.98	
595	0.014301	52.89	
Average Train Loss	**Validation Loss**	**Validation Accuracy**	**Elapsed (s)**
0.014826	1.627959	0.81	337.16

Classification Report for BERT + CNN				
	Precision	**Recall**	**F1-Score**	**Support**
0	0.81	0.87	0.84	3209
1	0.77	0.68	0.72	2084
Accuracy			0.80	5293
Macro Average	0.79	0.78	0.78	5293
Weighted Average	0.79	0.80	0.79	5293

Figure 91: Output of BERT CNN

Based on the test results of the 4 Deep Learning models, both RNN and LSTM have the highest and very similar f1-score but in this case, RNN will be deemed as the best performing model due to it being able to achieve a higher recall score. One possible reason for this could be that the two models complement each other's strengths. BERT is trained on a large corpus of text and is able to capture complex linguistic patterns and relationships, while RNNs can effectively model sequential dependencies and extract contextual information. By combining these models, we can leverage the strengths of both to improve performance on the NLP tasks.

11.4 Conclusion of BERT

Model/Metric	Accuracy	Precision	Recall	F1-Score
RNN	0.78	0.73	0.73	0.73
LSTM	0.80	0.78	0.70	0.74
CNN	0.80	0.77	0.68	0.72
BI-LSTM	0.80	0.79	0.69	0.74
LR	0.75	0.57	0.72	0.64
SVM	0.77	0.53	0.84	0.64
GNB	0.70	0.61	0.62	0.62
GB	0.75	0.50	0.79	0.61
RF	0.75	0.46	0.82	0.59
DT	0.65	0.55	0.55	0.55

The above table shows the evaluation result for both Traditional Classifiers and Deep Learning Models. Few key observations that can be made are that generally, the deep learning models were able to achieve a higher precision, f1-score and accuracy score than the traditional classifier but not in the case for recall.

Since the focus of model evaluation is based on f1-score, it is fair to say that the deep learning models perform significantly better than the traditional classifiers as the former achieve significantly higher f1-score than the latter. These deep learning models poof to be able to improve performance significantly by having a longer training time.

We can conclude that the best performing combination with BERT will be RNN as mentioned in Part 11.3 because even though both RNN and LSTM have the highest and very similar f1-score, RNN will be deemed as the best performing model due to it being able to achieve a higher recall score.

12. Results and Discussions

The following table illustrates our overall best pairing of both deep learning and machine learning models respectively.

Deep Learning Models				
Model/Metric	**Accuracy**	**Precision**	**Recall**	**F1-Score**
GPT + CNN	0.75	0.80	0.48	0.60

Word2Vec + Bi-LSTM	0.76	0.76	0.56	0.64
ELMo + CNN (small data sample)	0.92	0.93	0.91	0.92
BERT + RNN	0.78	0.73	0.73	0.73
Traditional Machine Learning Models				
ELMo + SVM	0.93	0.88	0.98	0.93
GPT + SVM	0.77	0.47	0.90	0.62
Word2Vec + SVM	0.75	0.48	0.82	0.60
BERT + SVM	0.77	0.53	0.84	0.64

From the above deep learning results table, we can see that the best combination at a glance would be ELMo + CNN. Based on our in depth analysis, ELMo works well with most Deep Learning Models as they capture rich, context dependent information about each word in the text, using a bidirectional network which was explained previously, unlike traditional word embeddings. Furthermore, upon further research, ELMo embeddings have been shown to improve the performance of a wide range of natural language processing tasks, including sentiment analysis. One reason for this is that the contextualised information captured by ELMo embeddings can help to address the problem of polysemy, where a word can have multiple meanings depending on the context in which it appears. For our implementation, we have tested and trained it with randomised sample data to account for the small sample that we used (due to the lack of memory allocation). Hence, it would most likely work as the best, even when we scale the size of the dataset. However, as the sample is only about 20% of the full data, we are unable to be 100% sure that ELMo + CNN will be **the eventual best performer and have settled for BERT + RNN as the final best combination**.

From the machine learning results table, we can distinctly say that SVM is the best performing machine learning model with the four word embedding models that we have used, with the best combination at a glance being ELMo + SVM. However, similar to the reason provided above, we are unable to be 100% sure that ELMo + SVM will be the eventual best performer, hence we have settled for **BERT + SVM as the final best combination for machine learning**. SVM does well on classification tasks as it aims to find the hyperplane that maximises the margin between two classes, giving a unique solution. This results in a classifier that has good generalisation properties, meaning that it can perform well on new, unseen data. Furthermore, SVM is robust to noises and outliers, which can be present in real-world data.

From these final best combinations, we can **conclude that BERT embeddings often result in more accurate text classifications and that the overall best combination for detecting depression will be**

BERT + RNN. This is possible due to the fact that BERT undergoes pre-training on a vast corpus of textual data, enabling it to acquire knowledge of intricate linguistic structures and associations among words and phrases in a contextualised manner. This pre-training process facilitates the capture of such complex patterns, subsequently enhancing its efficacy in text classification and other downstream tasks. Furthermore, deep learning models perform better than machine learning models as they are able to learn complex representations of the text data, better generalisation capacity for unseen data as well as the ability to take on a transfer learning approach.

13. Conclusion and Future Works

Moving forward, there are a few aspects that we can improve and further develop our project on.

Firstly, we can explore running ELMo embeddings on the entire dataset with the use of Cache with AWS S3 Bucket to see the actual accuracies of ELMo output on the dataset as we are currently limited by the lack of CPU memory to store all our ELMO embeddings.

Secondly, with the exacerbating numbers of depression cases in recent years, there is a need to provide avenues where the public is able to detect if a tweet or comment is depressive or not. If detected as depressive, precautionary measures can be taken to prevent further dangerous thoughts. Hence, we are proposing the deployment of our top-performing model into a web application for the public to detect if a tweet is depressive or not.

Lastly, due to the limited time that we have throughout the project runtime, we were unable to explore Doc2Vec and Topic Modelling in full scale to be shown as part of our project deliverables, though we had the foundations completed. This was because there was insufficient data from other sources such as from forums like Reddit which will provide us with lengthier posts for analysis and the detection of keywords used, which could ultimately be used by Topic Modeling to give us the top few categories of depression. Furthermore, as seen from our EDA, we only had 8 unique words that were filtered out from the depressive tweets, hence with large corpus and longer text being fed into our model, more keywords and categories can be filtered out which will aid in the detection of depressive tweets. Hence, in the future, we could collate datasets from different platforms such as Reddit, Youtube comments, instagram comments and so on to have more accurate and inclusive predictions of depressive sentiments.

14. References

Ahmad, Hussain & Asghar, Dr. Muhammad & Alotaibi, Fahad & Hameed, Ibrahim. (2020). Applying Deep Learning Technique for Depression Classification in Social Media Text. Journal of Medical Imaging and Health Informatics. 10. pp. 2446-2451(6). 10.1166/jmihi.2020.3169.

Agrawal, S. (2021, May 19). How to split data into three sets (train, validation, and test) And why? Medium. https://towardsdatascience.com/how-to-split-data-into-three-sets-train-validation-and-test-and-why-e50d22d3e54c

Au Yeung. (2020, June 19). BERT - Tokenization and Encoding. Albert Au Yeung. Retrieved April 9, 2023, from https://albertauyeung.github.io/2020/06/19/bert-tokenization.html/

Brownlee. (2019, January 25). Understand the Impact of Learning Rate on Neural Network Performance. Machine Learning Mastery. Retrieved April 9, 2023, from https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/

ChatGPT, personal communication, April 2023

Dataquest. (2023, February 8). How to clean and analyze your data, using feedback of student projects. Dataquest. Retrieved February 18, 2023, from https://www.dataquest.io/blog/how-to-clean-and-prepare-your-data-for-analysis/

Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019). BERT: Explained - State of the Art Language Model for NLP. Towards Data Science. https://towardsdatascience.com/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270

Farruque, N., Zaïane, O. R., Goebel, R., & Sivapalan, S. (2022, May). DeepBlues@ LT-EDI-ACL2022: Depression level detection modelling through domain specific BERT and short text depression classifiers. In Proceedings of the Second Workshop on Language Technology for Equality, Diversity and Inclusion (pp. 167-171).

Fox, M. (2018, July 31). Depressed teens turn to social media to cope, survey finds. NBCNews.com. Retrieved February 18, 2023, from https://www.nbcnews.com/health/health-news/depressed-teens-turn-social-media-cope-survey-finds-n895951

Great Learning Team. (2020, July 12). What is Word Embedding | Word2Vec | GloVe. Great Learning. Retrieved April 9, 2023, from https://www.mygreatlearning.com/blog/word-embedding/

Jain, A. (n.d.). How to Use ELMo Embedding in Bidirectional LSTM Model Architecture. Insofe Insights. Retrieved April 20, 2023, from https://www.insofe.edu.in/insights/how-to-use-elmo-embedding-in-bidirectional-lstm-model-architecture/

Joshi, P. (2019, March 11). Learn to Use ELMo to Extract Features from Text. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2019/03/learn-to-use-elmo-to-extract-features-from-text/

Lee, Y. (2021, February). Illustration of ELMo architecture [Figure]. ResearchGate. https://www.researchgate.net/figure/Illustration-of-ELMo-architecture_fig1_346273417

Nolet, B. (2022). DataLoader. In Deep Learning (1st ed.). Manning Publications. https://livebook.manning.com/concept/deep-learning/dataloader#:~:text=data%20module%20has%20a%20class,choose%20from%20different%20sampling%20strategies.

Papers with Code - Linear Warmup Explained. (n.d.). Papers With Code. Retrieved April 9, 2023, from https://paperswithcode.com/method/linear-warmup

Prakash, P. (2021, September 9). An Explanatory Guide to BERT Tokenizer. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2021/09/an-explanatory-guide-to-bert-tokenizer/

Putri, Citra & Damayanti, Novita & Hamzah, Radja. (2020). Sadfishing Phenomenon of #Justiceforaudrey (Hashtag) on Twitter. Mediator: Jurnal Komunikasi. 13. 10.29313/mediator.v13i1.5598.

Raymond Chiong, Gregorius Satia Budhi, Sandeep Dhakal, Fabian Chiong, A textual-based featuring approach for depression detection using machine learning classifiers and social media texts, Computers in Biology and Medicine, Volume 135, 2021,104499, ISSN 0010-4825, https://doi.org/10.1016/j.compbiomed.2021.104499.

When life gets to you, we provide a safe space. SOS. (2022, July). Retrieved April 21, 2023, from https://www.sos.org.sg/pressroom/singapores-suicide-rates-decrease-yet-rising-youth-numbers-cause-for-concern

Sanad, M. (2019, September 25). Demystifying BERT: A Comprehensive Guide to the Groundbreaking NLP Framework. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2019/09/demystifying-bert-groundbreaking-nlp-framework/

Shoeb, A. A. M., & de Melo, G. (2021). Assessing emoji use in modern text processing tools. arXiv preprint arXiv:2101.00430.

The Healthline Editorial Team. (2019, October 15). Why teens turn to 'sadfishing' on social media. Healthline. Retrieved February 18, 2023, from https://www.healthline.com/health-news/why-teens-turn-to-sadfishing-on-social-media