Review of Sentiment Classification Analysis on Hateful Speech on Twitter Report

Andri Setiawan Susanto 1002849
Chan Terng Tseng Nigel 1002027
Jayvan Leong Ghit 1002780
Information Systems Technology and Design - 2019
Singapore University of Technology and Design
Course: ISTD - 50.038 - Computational Data Science

I. Abstract

In recent years, hate speech and hate crimes have been on the rise on social media. Twitter, being one of the top social platforms, has been a medium for propagating these rampant prejudicial ideals. In light of this, our project aims to predict if a tweet contains hateful speech. The following is a report on the different Machine Learning models we have worked with to apply sentiment classification analysis on hateful speech in Twitter. It includes what we have learnt from the process of trial and error and it highlights several Machine Learning algorithms that can be applied in this area to improve the classification of the twitter texts.

II. Methodology

A. Dataset Collection and Extraction

We used Andy Fou's Twitter Data that consisted of 99799 different tweet ids and their associated labels. These labels comprised of hateful, abusive, normal and spam. To gather and put our initial dataset together, with the twitter ids in hand, we used a Python script together with Twitter's API to crawl and retrieve all the text to be used for our classification model. As several tweets were no longer existing or had null values, we removed those data accordingly. This resulted in a total of 62579 valid tweet ids out of 99799 tweet ids.

We performed data manipulation to extract features we found meaningful, such as timestamps for the tweet's creation, different hashtags, etc, that were used for visualization for us to observe whether certain trends could have an impact on how we should potentially tweak our process.

Fig. 1 displays the class distribution of the four different labels in the data set. From the numbers, we could see a clear imbalance in the data set where there was a significantly larger proportion of non-hateful class compared to the hateful class.

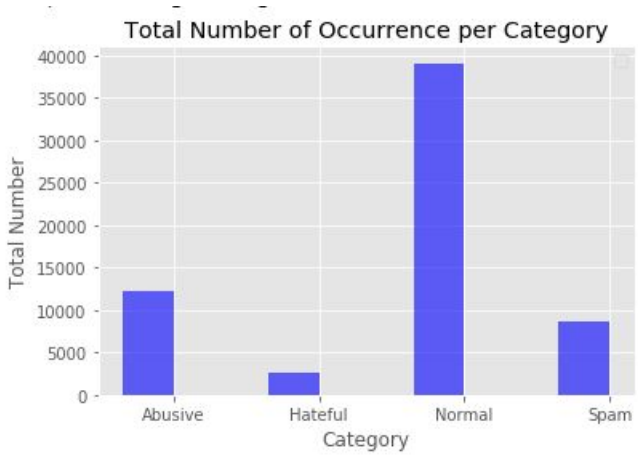

Fig. 1. Different category distribution

B. Dataset Pre-processing

With regards to the numbers seen in Fig. 1, we decided to combine Normal and Spam classes as class 0, and Hateful and Abusive classes as class 1. We justified putting Hateful and Abusive classes together as their content had relatively similar characteristics.

Given that there was still a huge imbalance at this point, we reduced the numbers of class 0 to

70 percent the size of class 1, resulting in a ratio of 7:10 between class 0 and class 1. This ratio was used to maintain some imbalance in the dataset to model it to real world circumstances where data is hardly ever balanced.

Next, some cleaning steps we took were the removal of hashtags, punctuations, stopwords and lowercasing of the text. We followed up with lemmatization of the text to group together the inflected forms of a word so that they could be analysed as a single item.

Fig. 2 shows the numbers for the resulting dataset.

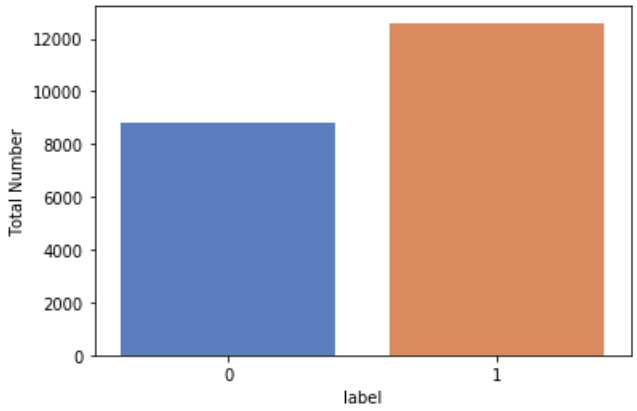

Fig. 2. Total Number of Occurance per Label with the ratio of 7:10

III. Sentiment Analysis using TextBlob

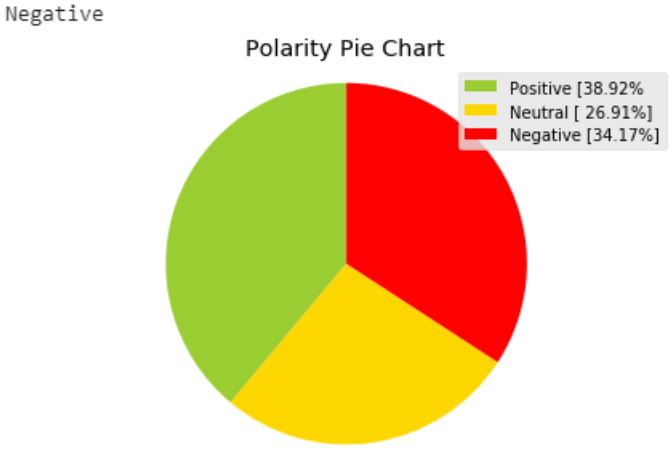

Fig. 3. Overall Sentiment of Unmodified Dataset

To analyse the sentiment of the unmodified dataset, we used TextBlob to plot a pie chart to get a better understanding of the overall sentiment of the dataset (Fig. 3). The same was done for our modified dataset (Fig. 4) so that we could get a rough gauge of this dataset's overall sentiment and to check if it was somewhat representative of the original dataset.

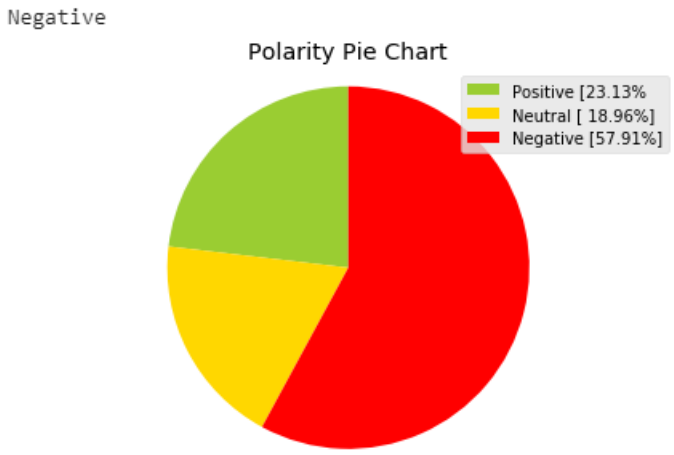

Fig. 4. Overall Sentiment of 7:10 Dataset

IV. Different Machine Learning Models Used for Classification

A. *GloVe*

GloVe is a log-bilinear model with a weighted least-squares objective. The main intuition for this model is the basic observation that ratios of word-word co-occurrence probabilities have the potential for encoding some form of meaning(Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014).

For the majority of the models we tested, we initialized the first layer with pre-trained GloVe word vectors from a 50 dimensional twitter glove word vector set. This was done where we took the word vectors of each word in our training data in sequence, and passed these vectors into an embedding matrix. This matrix was then passed as weights to initialize the embedding layer.

There are other models for word embeddings such as Word2Vec. Some research was done where we found that Word2Vec only accounted for local contexts and did not take the global context into consideration. This is where GloVe embeddings differ as they apply neural methods to decompose the co-occurence matrix into dense vectors that

are generally more expressive with vectors that are faster to train. Despite their differences, both should be evaluated together with the same dataset as neither of the two have shown to outperform the other(Aaron (Ari) Bornstein, 2018).

We chose to start with using GloVe embeddings first.

B. GloVe with SimpleRNN

NOTE In this subsection, material applied came from our Computational Data Science Course in Singapore University of Technology and Design taught by Professor Soujanya Poria and Professor Dorien Herremans.

Recurrent Neural Networks (RNN) is a model that processes a sequence of inputs which generates output sequences at different time steps.

We trained with the SimpleRNN model as a naive approach to investigate the kind of results that would come from vanilla RNN model. As such, the results we got were below average and we deduced that this could be attributed to the cons of RNN that we had learnt of, namely the vanishing/exploding gradient.

The problem of the vanishing/exploding arises during training as the gradients are being propagated back in time to the initial layer. Due to this, gradients from deeper layers have to undergo continuous matrix multiplications due to the chain rule. As these gradients approach the earlier layers, having a small value of less than 1 would result in them shrinking exponentially until they 'vanish', making it impossible for the model to learn. On the other hand, if the gradients have large values of greater than 1, they get larger and eventually 'explode' and crash the model (Eniola Alese, 2018). These two issues give rise to the vanishing and exploding gradient problems respectively.

With that problem in mind, our next step was to take a look at LSTM, a model that was designed to overcome the problems of vanilla RNN.

C. GloVe with LSTM Model

Long-short term memory (LSTM) network is a recurrent neural network that consists of gates that have the ability to remember patterns for long durations (Moawad, 2018). This model is essential in providing a more accurate prediction for the twitter analysis because each tweet contains a sequence of words. By using LSTM, we can retain the overall sequence and hence meaning of the sentence in a tweet. An important aspect of using this sequence modeller is such that it retains important information and discards the unimportant ones.

The LSTM model makes small modifications to the information provided by applying multiplications and additions and works in a way where information flows through a mechanism known as cell states. This allows LSTM to selectively remember or forget things(Srivastava, 2017). A LSTM cell/unit comprises the following:

1) Cell State : LSTM internal state stores and transfers relevant information across the sequence chain. It has the formula: $C_t = f_t * C_{t-1} + i_t * C'_t$
2) Hidden State : It stores information from the previous input.
3) Forget gate (see Fig. 5) : This gate is responsible for selecting the input data. It will decide whether the information needs to be kept or forgotten depending on the usefulness of the data. The forget gate will output 1 or 0 from the input feed. Data is kept when the output is 1 and forgotten if the data when the output is 0. It has the formula: $f_t = \sigma(W_f.[h_{t-1}, x_t] + b)$

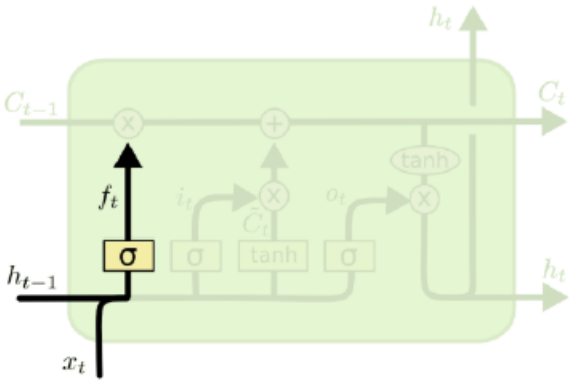

Fig. 5. LSTM Forget Gate

4) Input gate (Fig. 6): It is responsible for determining what new information will be written to the cell state where sigmoid and tanh functions are used. Sigmoid function produces 0 or 1, which determines what to

write while Tanh function produces values between -1 and 1, which generates the candidate values. It has the formulas: $i_t = \sigma(W_i .[h_{t-1},x_t] + b)$ and $C'_t = \tanh(W_C.[h_{t-1},x_t] + b_C)$

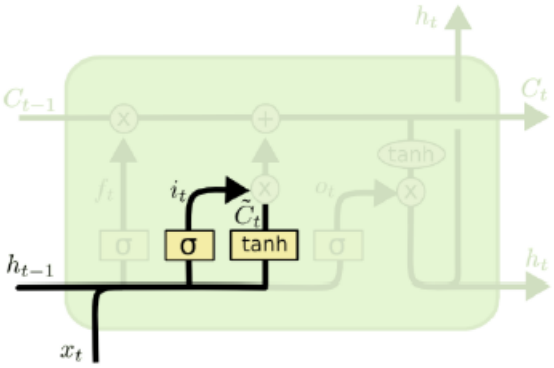

Fig. 6. LSTM Input Gate

5) Output gate (Fig. 7): The output gate outputs the next hidden state. This new hidden state will be passed to the next LSTM cell and/or used to generate a prediction at the end. It has the formula: $o_t = \sigma(W_o[h_{t-1},x_t] + b_o)$ and $h_t = o_t * \tanh(C_t)$

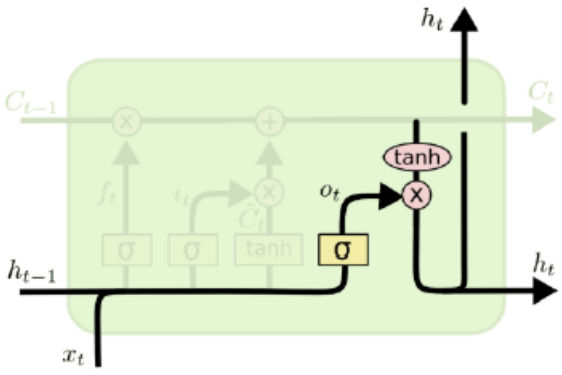

Fig. 7. LSTM Output Gate

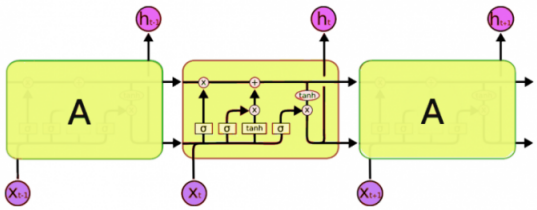

Fig. 8. LSTM Architecture

Before adding the LSTM layer as our sequence modeller, we have to apply some additional preprocessing to our dataset, these steps include:

1) Tokenization : Sentence is split into its individual words and where integers are assigned to their respective words.
2) Sequences Padding: Pads the sequence into equal length with respect to the text sequence with the greatest length in the dataset.
3) Embedding process: Each word in the dataset will be represented as a vector using a dense vector representation. In this case, Glove pre-trained word vectors were used. Word embeddings are used as they provide large sparse vectors to represent each word. A given word will be represented as a large vector of mostly zero values.The position of a word in the learned vector space is referred to as its embedding.

To deal with the shortcomings of vanilla RNN, LSTM is deisgned in such a way whereeach LSTM cell uses the previous cell and hidden states.

$C_t = f_t * C_{t-1} + i_t * C'_t$ (1)
$C_{t-1} = f_{t-1} * C_{t-2} + i_{t-1} * C'_{t-1}$ (2)
$C_{t-2} = f_{t-2} * C_{t-3} + i_{t-2} * C'_{t-2}$ (3)

.

.

Thus,

$C_t = C_{t-1} \text{ X } \sigma(W_f.[h_{t-1},x_t]) + tanh(W_c.[h_{t-1},x_t]) \text{ X } \sigma(W_i.[h_{t-1},x_t])$

By taking the differentiation of C_t:

$$\frac{\partial C_t}{\partial C_{t-1}} = \sigma(W_f \cdot [H_{t-1}, X_t]) + \frac{d}{dC_{t-1}}(tanh(W_c \cdot [H_{t-1}, X_t]) \otimes \sigma(W_i \cdot [H_{t-1}, X_t]))$$

$$\frac{\partial C_t}{\partial C_{t-1}} \approx \sigma(W_f \cdot [H_{t-1}, X_t])$$

substituting this equation into:

$$\frac{\partial E_k}{\partial W} = \frac{\partial E_k}{\partial H_k}\frac{\partial H_k}{\partial C_k}\frac{\partial C_k}{\partial C_{k-1}} \cdots \frac{\partial C_2}{\partial C_1}\frac{\partial C_1}{\partial W} = \frac{\partial E_k}{\partial H_k}\frac{\partial H_k}{\partial C_k}\left(\prod_{t=2}^{k}\frac{\partial C_t}{\partial C_{t-1}}\right)\frac{\partial C_1}{\partial W}$$

and the forget gate to be approximately 1

$$\sigma(W_f \cdot [H_{t-1}, X_t]) \approx \vec{1}$$

$$\frac{\partial C_t}{\partial C_{t-1}} \nrightarrow 0$$

$$\frac{\partial E_k}{\partial W} \approx \frac{\partial E_k}{\partial H_k}\frac{\partial H_k}{\partial C_k}\left(\prod_{t=2}^{k}\sigma(W_f \cdot [H_{t-1}, X_t])\right)\frac{\partial C_1}{\partial W} \nrightarrow 0$$

Hence, the gradient does not vanish, (Arbel, 2018) solving the problem of the vanishing/exploding gradient in RNN.

D. GloVe with Bidirectional LSTM

Bidirectonal LSTM (Fig. 9) is a model that is built on top of the existing LSTM model by using two LSTM cells that come from the duplication of the first recurrent layer in the network, for sequence classification where they now run as two layers side by side. One layer handles the normal input sequence while the other handles the reverse of the input sequence.

For the normal LSTM, inputs are run only from past to future(forward), whereas for bidirectional LSTM, inputs are run both from past to future(forward) and future to past(backward).

This functionality of bidirectional LSTM is where it differs from the regular LSTM as it allows preservation of both the past and future information. By combining the hidden states from both the forward and backward passes, we are able to gain context from both past and future at any given point in time of a specific time frame (Jason Brownlee, 2017).

The bidirectional LSTM is able to understand context better than the regular LSTM and hence can possibly learn better.

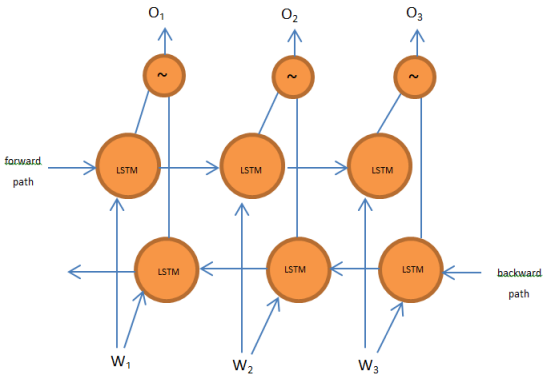

Fig. 9. BidirectionalLSTM Architecture

E. BERT

Bidirectional Encoder Representations from Transformers (BERT) is a state-of-the-art natural language processing model. BERT pretrains on deep bidirectional representations from unlabelled text data by conditioning on both left and right context in conjunction in all layers. This creates a deep bidirectional representation as compared to the Bidirectional LSTM model with GloVe embeddings model.

This is facilitated by employing an unsupervised fine-tuning approach on the self-attention layers of the Transformer (Vaswani et al., 2017) also used in OpenAI GPT (Radford et al., 2018) which formerly achieved state-of-the-art results on many sentence-level tasks from the GLUE benchmark. (Wang et al., 2018) The key difference is that BERT Transformer uses bidirectional self-attention, while the GPT Transformer uses constrained self-attention where every token can only attend to context to its left like LSTM models. as shown in Fig 10. (Devlin, Chang, Lee Toutanova, 2019)

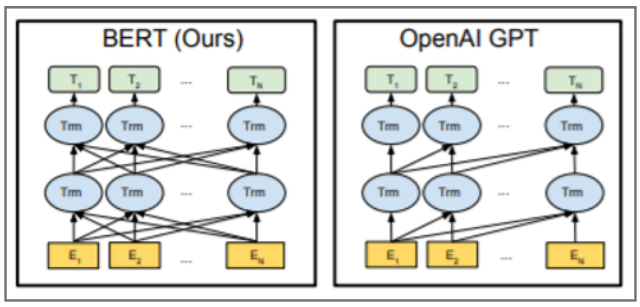

Fig. 10. Bert vs GPT transformer

A visual representation of the architecture the model is shown in Fig. 11, where input embeddings denoted as E and the final hidden vector for the 1^{th} input token as $T_i \in R^H$

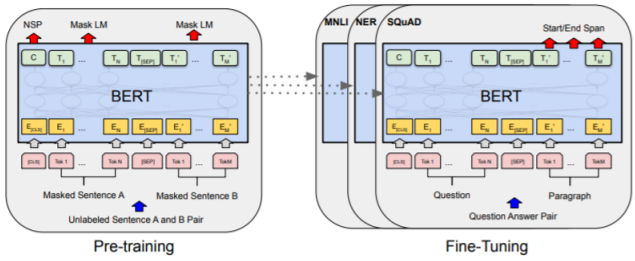

Fig. 11. Model Architecture

Fig. 11 depicts the overall pre-training and fine-tuning procedures for BERT. The same algorithm structures are used in both pre-training and fine-tuning. [CLS] is a special symbol added in front of every input example, and [SEP] is a special separator token added to the end of every sentence.

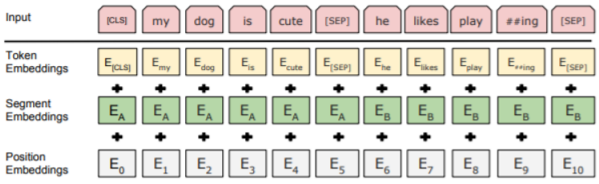

Fig. 12.

BERT uses the WordPiece embeddings with a 30,000-token vocabulary for its input embeddings. (Devlin et al., 2019) In Fig. 12 we observe an additional learned embedding being applied to each token to indicate which sentence it belongs to. For every token, its input representation is the summation of its token, segment and position embeddings.

The model is pretrained using two unsupervised tasks; Masked Language Model (MLM) and Next Sentence Prediction (NSP). This is shown in the "Pre-training" diagram in Fig. 11. BERT implements MLM, also known as Cloze task (Taylor, 1953), by randomly masking 15% of all tokens in each input sequence and then predict those masked tokens. This is done by feeding the final hidden vector of the masked tokens (T) into an output SoftMax function over the WordPiece vocabulary. (Devlin et al., 2019)

The second unsupervised task NSP is implemented by feeding in a pair of sentences as inputs with a special separator token [SEP] in between. The relationship between the two sentences are binarized during pretraining where a label of either IsNext when paired with the correct sentence or NotNext when paired with a randomly incorrect sentence in equal proportions. This prove effective in further task applications such as the Stanford Question Answering Dataset (SQuAD v1.1).

V. Critical Finding

A. Comparing Results (SimpleRNN, LSTM, Bidirectional LSTM)

The results of our classification with our first few models are shown in this section where batch size of 128 and learning rate of 0.01 were used.

Model	Accuracy	Null Accuracy	F1 Score	Precision	Recall	Misclassification Rate	Specificity
SimpleRNN	0.5996	0.5916	0.7463	0.5969	0.9953	0.4004	0.0263
LSTM	0.9071	0.587	0.9201	0.9283	0.9121	0.0929	0.8994
BiLSTM	0.8966	0.587	0.9103	0.9274	0.8938	0.1034	0.9005
CuDNNLSTM	0.8868	0.5786	0.9051	0.8788	0.933	0.1132	0.8233
BiCuDNNLSTM	0.8903	0.5786	0.9063	0.8959	0.9169	0.1097	0.8537

Fig. 13. Result from Different Model Without Considering Dropout

From the table (Fig. 13), without considering the F1 score, CuDNNLSTM model gave the highest accuracy.

We reevaluated the model by implementing dropout at certain locations in our training model. Dropout is where nodes are dropped randomly. This reduces overfitting of our training model and improves generalization error in deep neural networks. To find which layer where dropout would give a better accuracy, we trained the different models with varying dropout rates and layers. The result is shown below (Fig. 14).

From our findings, we found out that placing dropout rate of 0.2 after embedding layer had the best improvement in accuracy for our prediction.

We noticed that in all our iterations, placing the dropout after the LSTM layers always resulted in overfitting where the validation loss would decrease a little and then increase exponentially. This led us to believe that the location where dropout is used must be done so selectively as an incorrect location would result in decreasing the model's performance.

Drop out	After		
	Embedding	LSTM	dense
0.1	0.9042	0.91244	0.9035
0.2	0.9157	0.9094	0.9038
0.5	0.9138	0.9064	0.8989
0.8	0.8984	0.914	0.9087
1	0.9054	0.9003	0.9047

Fig. 14. Accuracy from different drop out location

1) Right after LSTM cells. This will cause the model to lose some important information that it has learnt inside the LSTM cells, reducing the model's ability to predict correctly.
2) Right before the activation layer. This will prevent our model to adjust its learning before doing the classification.
3) When dealing with small datasets, there is no need for regularisation as the model has already limited amounts of information to train with. By applying dropout, it will decrease the performance of the model in predicting.

Fig. 15 shows the results for the different models where dropout is performed after the embedding layer.

Early stopping			
Monitor	Mode	Verbose	Patience
val_loss	min	1	10

Model	Accuracy	F1 Score	Precision	Recall	Misclassification Rate	Specificity
SimpleRNN	0.7952	0.8426	0.7728	0.9262	0.2048	0.6054
LSTM	0.9068	0.9219	0.9146	0.9294	0.0932	0.8742
BiLSTM	0.9113	0.9254	0.921	0.9298	0.08872	0.8845
CuDNNLSTM	0.908	0.9217	0.928	0.9155	0.092	0.8971
BiCuDNNLSTM	0.9038	0.9166	0.9196	0.9136	0.0962	0.8903

Fig. 15. Dropout after Embedding Layer

* CuDNNLSTM is another version LSTM with GPU enabled capability

From applying the dropout, we see an increase in the accuracy as the model is less prone to overfitting due to reducing the number of nodes at random, within the neural network.

B. Evaluation (SimpleRNN, LSTM, Bidirectional LSTM)

We have used different metrics to assess the accuracy of our prediction for the different models.

By looking at accuracy alone, it might not give us the complete picture of our prediction. For example, if our data has 90% class 1, our prediction will get 90% accuracy. This shows that our model did not learn well during training phase.

Model	Accuracy	Null Accuracy	F1 Score	Precision	Recall	Misclassification Rate	Specificity
SimpleRNN	0.7952	0.5916	0.8426	0.7728	0.9262	0.2048	0.6054
LSTM	0.9068	0.5916	0.9219	0.9146	0.9294	0.0932	0.8742
BiLSTM	0.9113	0.5916	0.9254	0.921	0.9298	0.08872	0.8845
CuDNNLSTM	0.908	0.5916	0.9217	0.928	0.9155	0.092	0.8971
BiCuDNNLSTM	0.9038	0.5786	0.9166	0.9196	0.9136	0.0962	0.8903

Fig. 16. With Null Accuracy Results

We need to check the null accuracy which is achieved by always predicting the most frequent class. If the null accuracy is about the same as the accuracy that we get, we should not trust that result. Fig. 16 shows the null accuracies for each of the models.

Next, we need to consider other metrics:

1) Recall value. It is also known as the sensitivity value. The higher the recall value the better the prediction.
2) Specificity value. It tells us how often the prediction is correct when the actual value is negative. A high specificity value shows that the model is not very sensitive to new datasets and very specific.
3) Precision value. It tells us how often our model can predict correctly when a positive value is predicted.
4) F1 Score. It shows the balance between Precision and Recall. For an imbalanced dataset, higher F1 score is preferred in order to ensure that the accuracy score is accurate.

Below is the result by ranking the accuracy with F1 Score

From these results, we found that overall, Bidirectional LSTM (BiLSTM) had the highest accuracy.

Model	Accuracy	Null Accuracy	F1 Score	Precision	Recall	Misclassification Rate	Specificity
BiLSTM	0.9113	0.5916	0.9254	0.921	0.9298	0.08872	0.8845
LSTM	0.9068	0.5916	0.9219	0.9146	0.9294	0.0932	0.8742
CuDNNLSTM	0.908	0.5916	0.9217	0.928	0.9155	0.092	0.8971
BiCuDNNLSTM	0.9038	0.5786	0.9166	0.9196	0.9136	0.0962	0.8903
SimpleRNN	0.7952	0.5916	0.8426	0.7728	0.9262	0.2048	0.6054

Fig. 17. Rank Accuracy with F1 Score

C. Comparing Results(BERT)

Like the previous models, we fed our binarized pre-processed dataset with a 7:10 class 0 to class 1 ratio of tweets into the model with a ratio of 1:4 of test and training data to fine-tune it for the prediction task of detecting hateful tweets. Each input sequence of the model is an independent tweet that was randomised and uniquely tokenised as shown in Fig. 18 as the "Single Sentence" input.

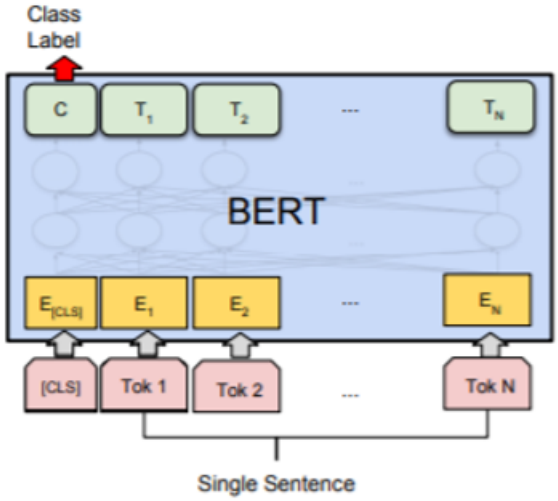

Fig. 18. Model architecture for fine-tuning process

We chose BERT-Base, Uncased: 12-layer, 768-hidden, 12-heads, 110M parameters with a batch size of 32 and fine-tune for 3 epochs over the data for our prediction task. We ran the model with 3 learning rates (4e-5, 3e-5, and 2e-5) to identify the best fine-tuning learning rate as shown in Fig. 19.

To ensure consistency in the baseline of the experiment, the model is fully restarted, and an empty output directory is initialised for each iteration. We then performed fine-tuning data shuffling and classifier layer initialization.

Model	Accuracy	Null Accuracy	F1 Score	Precision	Recall	Misclassification Rate	Specificity
BERT (2e-5)	0.9228	0.4284	0.932	0.938	0.9262	0.0772	0.9015
BERT (3e-5)	0.9169	0.4131	0.9297	0.9229	0.9366	0.0831	0.91
BERT (4e-5)	0.92156	0.4066	0.9343	0.9287	0.94	0.0784	0.9124

Fig. 19. Results from BERT with different learning rates.

D. Evaluation(BERT)

From Fig.19 , we observed that the learning rate of 4e-5 provides the overall best result as it is 1st in three metrics (F1 score, Recall and Specificity) and 2nd in the other two (Precision and Accuracy). We also observe that it is 2nd lowest in both Misclassification rate.

VI. Discussions

A. Observations Made

1) Dataset Duplicates: When we started training our models with our modified datasets, overfitting occurred quite frequently. We attempted to fix this first by looking at our datasets once more where we found an intersting trend that we had not taken into consideration earlier on.

In the initial stages, we only did a check on the twitter ids for our tweets to ensure there was no duplication. However, we failed to check the texts for duplicates as we had assumed that there would be none due to the unique twitter ids.

Upon checking this out, we found that about 10% of our tweets had duplicate texts. This could possibly be due to the nature of humans tweeting the same things, or perhaps the usage of bots to post multiple tweets with the same texts.

Removing these duplicates led to a significant reduction in our overfitting problem.

2) Dropout Location: While we were experimenting and trying to find which layers where dropout would work best, we found a general trend where placing the dropout after the LSTM, SimpleRNN or BiLSTM layers would result in overfitting.

The observation of this trend allowed us to determine that our best approach would be to place the dropout after the embedding layer in our training model.

B. Rational behind using Tahn function in LSTM

Tanh function is used in between hidden and output states because it has an anti-symmetric function. This function reduces the systematic bias for neurons at positions after the first neural layer. If Sigmoid function is used instead, it will introduce the systematic bias as it forces the output to be between 0 or 1. If we are dealing with a large network, Tahn function will converge faster for back-propagation learning.

C. Area affecting LSTM Weight

As all cells in LSTM are identical, the weight depends on the input dimensions and hyperparameters for the output dimensions. It does not depend on sequence length. The dimension of weights depends on the concatenated vector. $[h_{t-1},x_t]$

D. Usage of Dropout

For this project, our use of dropout was a means to investigate if it could enhance our model's performance. On further research, we found that the general use of dropout was done so in intermediate layers, where it attempts to stop the higher layers from being too dependent on any one neuron. This would encourage the neurons to encode more useful information, and hence possibly improve model performance.

In light of that, we realised that our way of implementing the dropout was a slightly different approach. Applying dropout after the embedding layer augmented the word embeddings by randomly zeroing some of the information. This encourages the model to be more generalizable and hence can possibly perform better against unseen data outside of our training data.

E. Possible Improvements

1) Comparison with Word2Vec: We did not use Word2Vec for our initialization this time round due to time constraints. However, we feel that one way we could have improved our findings would have been to initialize the same models we used with Word2Vec, and train them with the same given set of hyper-parameters. We would then take the results to compare them and see if one was better than the other in terms of overall performance.

2) Doc2Vec: Initially we did not venture into Doc2Vec as its sentence vectors are word order independent, meaning that word sequence is not taken into account using this model. Hence, as the input to the model loses its context, we felt that this might not give a good representation of the tweet's overall meaning. However, we could have possibly tested out this model to look for some meaningful observations.

3) BERT: Improvements to the performance of the BERT model could be made by pretraining the model on labelled datasets for emotions and sentiments for feature extraction before fine-tuning the model with our own dataset. This would also reduce computation time as the model can be fine-tuned with a small sample from our dataset.

VII. Conclusion

The best result we got from our testings with our first few models (SimpleRNN, LSTM, Bidirectional LSTM) was from our bidirectional LSTM model which had an accuracy of 0.9113 and F1 score of 0.9254. We believe that this is due to this model's ability to take into account both the past and future context which makes it able to learn context very well.

We recognise that the Bidirectional LSTM with GloVe embeddings is a feature-based model in which contextual representation of each token is the concatenation of independently trained left-to-right and right-to-left representations. However, research showed that deep bidirectional model is not as powerful as BERT as the latter can use both left and right context at every layer as shown in Fig 10. As such, from our findings we saw a better performance from the BERT model as expected with an accuracy of 0.9216 and F1 score of 0.9343.

Overall from our research and through the process of this project, we learnt that each of the models have their own strengths and weaknesses. Under certain circumstances, some models perform better than the others. If we want to predict shorter sentences for example, SimpleRNN could be a good enough model to use as the model only need to know nearby words to make the prediction. For longer sentences, LSTM, Bidirectional LSTM and BERT would give better predictions as these models can learn the context better and do not suffer from the vanishing/exploding gradient problem.

When assessing the performance of the model, we should not focus on one metric, but consider all the different metrics instead to get a full picture of the model's performance. A good example in this case would be the overfitting issue where our accuracy, precision and F1 scores were really good but, on further inspection, looking at the validation loss graph allowed us to detect overfitting due to the general increase we witnessed in some of the findings.

References

[1] Andy Fou, (2018). Hate and Abusive Speech on Twitter. https://github.com/ENCASEH2020/hatespeech-twitter

[2] Assad Moawad, (2018). The magic of LSTM neural networks. https://medium.com/datathings/the-magic-of-lstm-neural-networks-6775e8b540cd

[3] Christopher Olah, (2015). Understanding LSTM Networks. http://colah.github.io/posts/2015-08-Understanding-LSTMs/

[4] Jeffrey Pennington, Richard Socher, Christopher D. Manning. (2014). GloVe: Global Vectors for Word Representation. https://nlp.stanford.edu/projects/glove/

[5] Eniola Alese, 2018, (June 6). The curious case of the vanishing exploding gradient https://medium.com/learn-love-ai/the-curious-case-of-the-vanishing-exploding-gradient-bf58ec6822eb

[6] Jason Brownlee, (2017). How to Use Word Embedding Layers for Deep Learning with Keras. https://machinelearningmastery.com/use-word-embedding-layers-deep-learning-keras/

[7] Natasha Bernal, (2019). Police to 'predict' hate crimes through Twitter for the first time. https://www.telegraph.co.uk/technology/2019/10/19/police-predict-hate-crimes-twitter-first-time/

[8] Nir Arbel, (2018). How LSTM networks solve the problem of vanishing gradients.https://medium.com/datadriveninvestor/how-do-lstm-networks-solve-the-problem-of-vanishing-gradients-a6784971a577

[9] Pranjal Srivastava, (2017). Essentials of Deep Learning : Introduction to Long Short Term Memory. https://www.analyticsvidhya.com/blog/2017/12/fundamentals-of-deep-learning-introduction-to-lstm/

[10] Radford, A., Narasimhan, K., Salimans, T., Sutskever, I. (2019, March 9). Improving Language Understanding with Unsupervised Learning. Retrieved December 4, 2019, from https://openai.com/blog/language-unsupervised/

[11] Taylor, W. L. (1953). "Cloze Procedure": A New Tool for Measuring Readability. Journalism Bulletin, 30(4), 415–433. doi: 10.1177/107769905303000401

[12] Vaswani, A. N., Shazeer, N. N., Parmar, N. N., Uszkoreit, J. N., Jones, L. N., Gomez, A. N., ... Polosukhin, I. N. (2017, December 6). Attention Is All You Need - arXiv. Retrieved December 4, 2019, from https://arxiv.org/pdf/1706.03762.pdf

[13] Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., Bowman, S. (2019, February 22). GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. Retrieved December 4, 2019, from https://arxiv.org/abs/1804.07461

[14] Wu, Mike, Chen, Zhifeng, Mohammad, Wolfgang, ... Hughes. (2016, October 8). Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. Retrieved December 4, 2019, from https://arxiv.org/abs/1609.08144

[15] Aaron (Ari) Bornstein, (2018, October 18). Beyond Word Embeddings Part 2. https://towardsdatascience.com/beyond-word-embeddings-part-2-word-vectors-nlp-modeling-from-bow-to-bert-4ebd4711d0ec

[16] Jason Brownlee, (2017, June 16). How to Develop a Bidirectional LSTM For Sequence Classification in Python with Keras. https://machinelearningmastery.com/develop-bidirectional-lstm-sequence-classification-python-keras/