Is it Offensive? Detection of Offensive Tweets Using LSTM, Transformer and BERT

Charlie Albietz, S3058735
Panagiotis Ritas, S3152529
Richard Fischer, S3124908
Jan-Henner Roberg, S3228851

Abstract

In this paper three models were trained to distinguish offensive tweets from non-offensive tweets using the OLID dataset. The data was preprocessed and tokenized using the BERT tokenizer to facilitate performance. An LSTM was used due to its resistance to the vanishing gradient, as well as a transformer and the pre-trained BERT transformer. The results show that the pre-trained transformer performs the best, followed by the standard transformer and then the LSTM. Each model was able to outperform the baseline majority class choice even though the data is highly disproportionate. Improvements can be made by using pre-trained embeddings together with a pre-trained model.

1 Introduction

The internet has brought about a fast, worldwide, pseudo-anonymous communication network. The consequences of this invention, like those of many others, are not solely positive. One of those negative consequences is Online Harassment, which roughly 40% of Americans have experienced (Vogels, 2021). While companies like Twitter.com enable free speech wherever the website is accessible, the 500 million tweets sent every day contain a worrying amount of offensive language (Stats, 2021).

The vast amount of offensive content created on Social Media platforms every day requires an automated response. For instance, Twitter just implemented a new feature that detects possibly offensive tweets before they are sent and asks the user if he really wants to send it (Guardian, 2021). This feature is most likely built on deep neural network architecture, an active field of research.

Since languages contain many long term relations, a system with memory is necessary for this task. The long and short term memory (LSTM) model by Hochreiter and Schmidhuber (1997) has memory capabilities, and the novel idea of self-attention gave rise to transformer models (Vaswani et al., 2017). These attention models can be pre-trained on immense amounts of data, which led to the development of BERT (Devlin et al., 2019). Since the memory and self-attention capabilities of these models enable their usage in a Natural Language Processing task like offensive language detection, and the improvement of such systems is necessary, this paper answers the following research questions: **1)** 'Which architecture is better at offensive language classification, LSTM or transformer?' and **2)** 'Does the use of a pretrained model significantly increase the performance of an offensive language detection network?'.

2 Related Work

The codalab competition 'OffensEval 2019' by Harvard Zampieri et al. (2019) seeks to find the best approach to identify and classify offensive language. Oswal (2021) compared an LSTM to, among others, a support vector machine (SVM) and a Random Forest on the OLID dataset. In his experiments, the LSTM yielded the best F1 score (F1=0.72). Caselli et al. (2021) retrained the pre-trained BERT architecture to HateBERT for abusive language detection in English and compared the BERT and HateBERT on the OLID dataset. The HateBERT architecture yielded an F1

score of 0.715 for the offensive class. Lee et al. (2018) compared RNNs and CNNs on their performance on multiclass offensive language labelling based on a much bigger Twitter dataset than the OLID. In their experiment, the RNN architecture showed the best F1 score (F1=0.8).

3 Models

3.1 Long Short Term Memory RNN

Long Short Term Memory (LSTM) Recurrent Neural Networks (RNNs) were first proposed by Hochreiter and Schmidhuber (1997) as an alternative towards casual RNNs for tackling the vanishing gradient problem.

LSTMs expand on the main idea behind RNNs by adding a cell state, an input, output and forget gate inside a hidden state. For N time steps, LSTMs unfold an input sequence $x = (x1, ..., x_N)$ to an output sequence ($y = y_1, ...y_N$)with mapping equations that lead to a given hidden state h_t, $t \in T = 1, 2...N$. For the cell state C_t, $\widetilde{C_t}$ is used as the intermediate cell state used to later obtain C_t, which is then passed to the next hidden state h_{t+1}. The input gate i_t controls the flow of input activation inside the network, by multiplication in the [0,1] range of the input activation in the network. Similarly, the output of the network is controlled in a multiplicative manner by the output gate o_t. The forget gate f_t of the memory cell essentially decides how much of the information to keep inside the hidden layer and consequently scales the internal state of the cell. This allows for long term dependencies. The forget gate f_t is therefore applied recurrently as the hidden states of the model unfold temporally until the model reaches N steps in the sequence. The equations and process of how h_t is calculated can be shown in section B and Figure 1, respectively.

3.2 Transformer

The Transformer model is a sequence-to-sequence architecture that was introduced in Vaswani et al. (2017). It incorporates a mechanism called "Self Attention". The model consists of several stacked transformer blocks that all perform sequence-to-sequence operations. Each block includes an attention layer, normalization and fully-connected layers (see Figure 2). The most important part is the attention layer, enabling the model to identify the most important words in sequence for a given task. The attention layer takes a series of vectors x as input and outputs the same number of vectors y, these are a weighted average of the input vectors: $y_i = \sum_j w_{ij} * x_j$. The attention weights are calculated as the dot product of two input vectors: $w_{ij} = x_i^T * x_j$, after all attention weights have been calculated in this way $softmax$ is used to transform the vector w into a pdf. Since the vectors x are learned, the model can decide which words (and word combinations) are important in a sequence. Layer normalization in the transformer block is similar to batch normalization, however, the normalization is performed element-wise instead of over the whole batch. For a more detailed description of Self Attention and the Transformer model please see Vaswani et al. (2017).

For our project we used an untrained transformer and BERT, a pre-trained model from Google (Devlin et al., 2019).

Without Pre-training To have a fair comparison to the untrained LSTM we trained a transformer model that was coded from scratch in `Pytorch` (implementation from Bloem (2019b)). This implementation uses two embedding layers, one for words (or tokens) and one for the positions of the tokens. The word-embedding layer learns how to present words in a vector of a given dimension. Positional embedding is needed as, unlike the LSTM, the Transformer is not trained sequentially and therefore needs this embedding to know where the word occurs in a sequence. Both embeddings have the same size, and are combined using summation.

After summation the embeddings are then fed through several transformer blocks (Figure 2). The number of transformer blocks is variable and can be optimised to obtain better results. After the final Transformer block, average or max pooling is performed to transform the sequence into a vector of the same length as the *embedding dimension*. This vector is then fed through a fully connected layer to get a 2-dimensional output vector. Finally, the cross-entropy loss is computed and backpropagation is performed.

BERT Bert, a widely used state-of-the-art language model, was initially developed from Google to understand search queries better. $TheBERT_{base}$ model used in this paper is available on huggingface (Huggingface, Huggingface) and consists of 12 encodes and 12 bidirectional self-attention heads. It has a total of 110 million learned parameters. It is pretrained for Masked Language Modeling and Next Sentence Prediction on the Books corpus (800 million words) and on English Wikipedia (2.500 million words). This gives it a contextual understanding of word meanings and therefore a significant advantage over non-pretrained models. In principle, BERT is a transformer network. However, there are some differences (for more information see Devlin et al. (2018)). The huggingface implementation allows for a lot of hyperparameter tuning, this will be further discussed below.

4 Experiments

In this paper, the performance of the three models will be compared by training and evaluating each model and subsequently comparing their accuracy and F1 score.

4.1 Data

We trained all models with the Offensive Language Identification Dataset (OLID) first used by Zampieri et al. (2019). The dataset contains 14099 tweets in total, of which 859 are used as a test set. Zampieri et al. (2019) labelled each tweet in three ways. First, the data was classified as being offensive or not offensive. The offensive tweets were further classified, distinguishing tweets that were offensive towards a specific target (i.e. individual or group) from tweets that were offensive in general and not directed at anyone in particular. The targeted tweets were further labelled as being targeted towards an individual, targeted towards a group of people or targeted towards others.

As we are not concerned with identifying in what way a tweet is offensive or not, we only made use of the offensive or not offensive labels. The distribution of the labels can be found in Table 1. Due to the uneven split of the data, the baseline model that will be used in this paper will be a model that will choose the majority class, thus the baseline accuracy for the test set is 0.72%

OLID	Train	Test
Offensive tweets	4400	239
Non-offensive tweets	8840	620
Total tweets	13240	859
Percentage offensive	0.33	0.278

Table 1: Overview of the data set label distribution.

4.2 Preprocessing

As each element from the training and testing set was a tweet taken directly from Twitter, the sentences themselves were rather difficult to work with. For this reason, a substantial amount of preprocessing was needed to be able to work with the data. The data was preprocessed in two steps. The first part aimed at making the dataset more manageable and reducing the level of abstractness that is used in day to day social media communication. As they were, the tweets contained many terms and characters that would make the subsequent tokenization step harder, such as the use of emojis, the use of hashtags combined with the lack of spaces between words, the tagging of other Twitter users and a large number of typos. The methods used were taken over from Dai et al. (2020) and will be described in the following section.

Emoji to Word Emojis were transformed into words representing their depiction using the python library `emoji`. By doing so the emoji: 😆 becomes *laughing*.

Hashtag Segmentation In this step, hashtags were segmented into individual words. This is a necessary step as hashtags, usually used to help categorize tweets and connect a tweet to a particular topic, are written in a single word that can consist of multiple words. The hashtags were segmented by splitting them up when they contained a capital letter. For instance, the hashtag: *#JusticeForOliver* would become: *justice for oliver*.

In the case that a hashtag is spelt in all caps, then, instead of splitting up each letter, the algorithm will change the entire hashtag to lower case. From this follows that *#KAREN* becomes *karen*.

User Mention Replacement When on Twitter it is custom to tag another user to make sure that the user can see the tweet. In the dataset, these tags have been replaced by *@USER*. If multiple users are being tagged in a single tweet these tags often occur successively. If one were to leave them the way they are and tokenize these multiple user tags to then vectorize them, a tweet with a large number of tags would be heavily biased by the tags although not carrying much meaning. To circumvent this issue, whenever a series of multiple tagged users are located within a tweet, this series is replaced by the new token: *@USERS*

BERT Tokenization After the previous prepossessing, the tweets are tokenized using the pre-trained BERT tokenizer. In this process, the sentences are first split into individual words according to the word piece approach. Here, words that contain word-parts such as suffixes are split up into the root word and then the suffix. For instance the word *running* would be split up into *run* and *##ing*. The double hash indicates that whatever follows (in this case the 'ing') will be appended to the previous word.

After the words have been split up, each token is assigned an index according to the vocabulary in the BERT tokenizer. Any word that is not recognised by the tokenizer is given the index corresponding to the unknown word token. As there are more words in the BERT tokenizer vocabulary than needed for our model, the indexes were redistributed to avoid having to make an unnecessarily large matrix of vectors in later steps.

4.3 Evaluation

As previously mentioned the evaluation of the models is done based on accuracy and F1 score. The accuracy of the models will be calculated by obtaining the ratio of correctly predicted tweets over the total amount of tweets in the test set. This measure is usually a good place to start in evaluating the performance of a model, however, due to the uneven split of the data, an accuracy of 72% can be achieved by predicting the majority class alone. Thus, F1 will also be taken into consideration.

F1 score is a performance measure used for binary classification that is calculated based on the models recorded precision and recall obtained from a confusion matrix. F1 is calculated as follows:

$$F1 = \frac{tp}{tp + \frac{1}{2}(fp + fn)} \tag{1}$$

Where tp is the number of true positive results, fp is the number of false-positive results and fn is the number of false-negative results. This calculation is done for both the offensive tweets and the non-offensive tweets being the true positive result. This leads to two F1 scores of which the weighted average is used to obtain the models' ability to classify between the classes.

4.4 Training

To obtain the best performance of each model, both the LSTM and the transformer will be optimized in several ways. The process in which the best settings was found was manual, as incorporating more systematic methods such as using K-Fold cross-validation and checking different settings for each fold would have been too time-consuming. Despite this, manual optimisation improved the models. After finding the best hyperparameter combination for each model, training was repeated over 5 times for these settings to obtain our evaluation metrics.

LSTM In the LSTM, we modified the learning rate, L2 weight-decay, hidden and embedding size, and dropout. The best combination of hyperparameters was used to provide the accuracies in 2. We used binary cross-entropy function as well as Adam for optimizer, as the combination of these two is usually very competent for model performance. We also used RMSprop for experiments but Adam proved to be better for this task. For the learning rate, we tried values between the range [0.00002, 0.0004] from which the value 0.0004 seems to work the best. We tried several L2 values but we decided to keep it to 0, as we used a dropout of 0.2% for regularization. We tried a small amount of LSTM layers for the model but it seemed that 4 layers a good approximation in terms of variance-bias trade-off.

Transformer The following hyperparameters can be optimized: learning rate, L2 weight-decay, depth, embedding size, number of attention heads and dropout. The depth was set to 6 stacked transformer blocks, the number of heads was set to 8 both of these were recommended in Bloem (2019a). Embedding sizes of 128, 256 and 512 were tried, 128 lead to the best performance presumably because there was not enough data to properly train larger embeddings. For the learning rate values in the range of [0.0001, 0.005] were tried and after some experimentation 0.0005 seemed to give the best results. For the L2 weight-decay values between [0, 0.01] were tested and 0.0008 gave the best results. Tuning these parameters was crucial for the model to not only predict the majority class. Especially the learning rate was very important since even with all other parameters at their optimum the model would perform badly with a bad learning rate.

Pre-Trained BERT Transformer The hyperparameter optimization in BERT involved the following parameters: learning rate, number of epochs, weight decay and class weights. Generally, the BERT fine-tuning process quickly leads to overfitting. Devlin et al. (2019) recommends only training for 2-4 epochs. Figure 3 shows signs of the model overfitting on the data after the second epoch. The overfitting could not be delayed with any L2 weight decay setting [0.01, 0.5]. Therefore, the model was trained for 2 epochs only.

Devlin et al. (2019) recommends a learning rate of 0.00002. The empirical hyperparameter optimization showed that 0.0005 was the best learning rate. Furthermore, class weights were implemented to deal with class imbalance. Unfortunately, this did not lead to any improvements of the model.

5 Results

The results that were obtained after training and evaluating the models can be seen in Table 2. We can see that each model performs better than the baseline and overall, the pre-trained BERT transformer performed the best showing the highest accuracy and the highest F1 score. This was expected, as being a pre-trained model, much time could have been spent on the optimisation. Still, given the model was only trained for 1 epoch using the current dataset it performed rather well compared to the other models.

Model	Accuracy	Weighted F1 score
Baseline	0.72	0
LSTM	0.75	0.76
T	0.8	0.79
PBT	**0.86**	**0.84**

Table 2: Results obtained after training and evaluating the three models: LSTM, Transformer and the pre-trained BERT Transformer compared to the baseline

Looking at the models that were trained from scratch, we can see that the transformer outperforms the LSTM slightly in both accuracy and F1 score. The LSTM only manages to achieve an accuracy that is slightly above the accuracy of the baseline. However, Looking at the F1 scores, we can see that this is not the case as the LSTM shows a high F1 score (see Table 3 for a more detailed view of the F1 scores).

6 Conclusions and Future Work

In this project, we compared and evaluated three different methods, namely an LSTM, a transformer from scratch and a pre-trained BERT model for identifying which is the best model for identifying offensive language on the internet. We also aimed to answer whether a pre-trained model would perform better in an offensive language identification task. BERT performed the best in terms of both evaluation metrics, while the other two models compared similarly in both scores, the transformer seemed to do better. Our project, therefore, finds evidence that a transformer is the better choice for offensive language detection and that using a pre-trained model is the best choice overall.

One limitation that came with the study is that the nature of the comparison is unfair, as, for example, BERT has more parameters than the other models which naturally makes the capacity of the model bigger. Furthermore, no pre-trained LSTM was used to be a direct comparison in performance to BERT. Another limitation was that in this project we

used trained embeddings from scratch which potentially undermined the performance of all models, as our dataset was small and it did not allow for sufficient embedding training. Future research should therefore investigate the effect of pre-trained embeddings on the accuracy of models other than BERT, as they might provide promising or even higher results than hateBERT (Caselli et al., 2021).

References

Bloem, P. (2019a). Transformers from scratch.

Bloem, P. (2019b). `https://tinyurl.com/59jsprdj`.

Caselli, T., V. Basile, J. Mitrović, and M. Granitzer (2021, Feb). Hatebert: Retraining bert for abusive language detection in english.

Dai, W., T. Yu, Z. Liu, and P. Fung (2020). Kungfupanda at semeval-2020 task 12: Bert-based multi-task learning for offensive language detection.

Devlin, J., M. Chang, K. Lee, and K. Toutanova (2018). BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR abs/1810.04805*.

Devlin, J., M.-W. Chang, K. Lee, and K. Toutanova (2019). Bert: Pre-training of deep bidirectional transformers for language understanding.

Guardian, T. (2021). Twitter launches prompt asking users to rethink abusive tweets.

Hochreiter, S. and J. Schmidhuber (1997, 12). Long short-term memory. *Neural computation 9*, 1735–80.

Huggingface. Huggingface bertforclassification.

Lee, Y., S. Yoon, and K. Jung (2018, Aug). Comparative studies of detecting abusive language on twitter.

Oswal, N. (2021, Apr). Identifying and categorizing offensive language in social media.

Stats, I. L. (2021). Twitter usage statistics.

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, u. Kaiser, and I. Polosukhin (2017). Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, Red Hook, NY, USA, pp. 6000–6010. Curran Associates Inc.

Vogels, E. A. (2021, May). The state of online harassment.

Zampieri, M., S. Malmasi, P. Nakov, S. Rosenthal, N. Farra, and R. Kumar (2019). Predicting the Type and Target of Offensive Posts in Social Media. In *Proceedings of NAACL*.

7 Appendix

A Figures

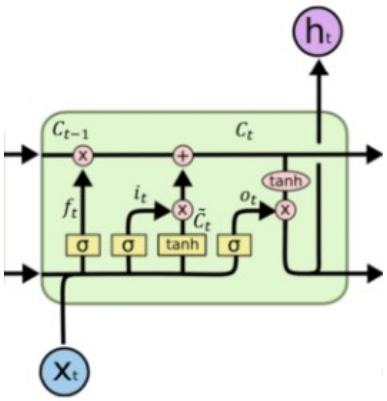

Figure 1: Illustration of an LSTM memory block.The figure illustrates the calculation of the forget, inpupt and output gate, as well as the calculation of cell states $\tilde{C_t}$ and C_t. The C_t eventually is passed through a tanh activation and its product is multiplied with the output gate to give h_t, $t \in T$

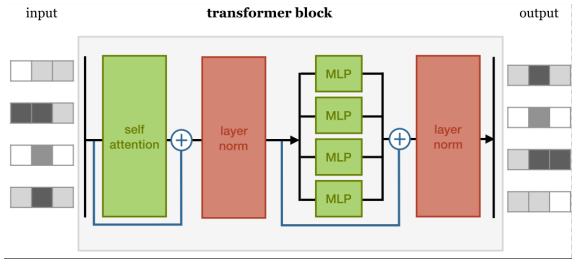

Figure 2: Illustration of a Transformer block, taken from `http://peterbloem.nl/blog/transformers`

B LSTM Equations

$$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \quad (2)$$

$$f_t = \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \quad (3)$$

$$\widetilde{C_t} = tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \quad (4)$$

$$o_t = \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \quad (5)$$

$$C_t = \sigma(f_t W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \quad (6)$$

$$h_t = o_t \odot tanh(c_t) \quad (7)$$

In the formulas shown above, W are the matrices that compute the transition of the current hidden state for the input or previous hidden state $ht - 1$, b are the bias parameters for each matrix multiplication, σ is the sigmoid activation applied at the end of the forget gate equation, input and output gates to scale the gates to [0,1], tanh is the activation applied for the first update of the cell state in timestep t,which brings the cell state to the range [-1,1], and $\odot$ is the Hadamard product (matrix dot product multiplication).

C BERT loss

Figure 3: The training loss compared to the validation loss for the BERT model across epochs. Overfitting can be observed after the second epoch.

D Tables

Model	F1 (1)	F1 (0)	Weighted F1 Score
LSTM	0.57	0.83	0.76
T	0.6	0.86	0.79
PBT	**0.72**	**0.88**	**0.84**

Table 3: Different F1 scores depending on which class was used (Offensive: 1, non offensive: 0) as true positive and then the weighted average F1 score

E Code

The code for this project can be found at `https://github.com/CharlieAprog/LTPoffensiveLanguage`.

F Group Contribution

Panos:

- created the LSTM model and fine-tuned the LSTM model, did part of the preprocessing (tokenization, lowering, split)
- wrote the LSTM, parts of training, and conclusion, as well as the LSTM figure,

hyperparameter table, and LSTM equations part of the appendix

Charlie:

- Data preprocessing, LSTM training and research also helped with LSTM model creation.
- wrote data, preprocessing, experiments and results.

Richard:

- BERT implementation, BERT tokenizer, BERT hyperparameter tuning (+ class weights)
- wrote Introduction + Related Work, wrote BERT training part

Jan:

- Data preprocessing, adopted the transformer implementation, training and optimizing the transformer
- wrote transformer model description section and transformer training section