Twitter sentiment analysis using LSTM

Adam Lewandowski, Ivan Sladkov, Patrick Cormac English
School of Computer Science
University College Dublin
Belfield, Dublin 4, Ireland
`adam.lewandowski@ucdconnect.ie`
`ivan.sladkov@ucdconnect.ie`
`patrick.english@ucdconnect.ie`

Abstract

In the advent of social media, forums, blogs, product review pages and other streams of human generated e-content, internet has become the de facto standard for human interactions and outlet for their opinions and emotions. This makes it a perfect resource for sentiment analysis, which can serve as critical information in the decision making process for businesses and politicians alike [1]. In this report we are especially focusing on tweet sentiment analysis. We discuss the challenges and findings associated with it, as well as comparing the performance of modern LSTM architecture against previous state of the art. Moreover, we discuss necessary text pre-processing techniques and describe the process of LSTM model construction.

1 Introduction

As micro-blogging websites, such as Facebook and Twitter, have proliferated in both size and popularity, there has been a concomitant growth in interest in data-extraction of the audiences and user-bases of these platforms.The data available on these platforms is both high-volume and complex,presenting challenges for both pre-existing and novel-model based analysis.One aspect of this data in particular, "Sentiment", has been the focus of extensive research in the field of model-based analysis. Financial, commercial, and political applications exist for the effective analysis of user-sentiment, and there has been particular emphasis on Twitter as a "real-time" source of user sentiment in relation to events.

This work will assess a variety of model-based approaches to assessing user sentiment, in particular the performance distinctions that arise between what might be termed "Deep Learning" approaches (Neural Networks – Recursive and Convolutional), and a variety of non-Deep Learning approaches. A shared pre-processing stage was developed, with a number of operations applied uniformly to a Twitter dataset ("Tweets") in anticipation of model training and prediction.

In addition to the initial assessment, an additional generalisation test is included as part of the experimental framework. This measures the performance of models on a "newer" Twitter dataset (with the same pre-processing stages), to identify model robustness across deviations in language and user tendencies.

2 Related work

This area has been subject to a variety of approaches using the models identified above, and a broad survey of existing works has been carried out as a preparatory step for this report.

Initial vectorisation of tweet data was considered with respect to the work undertaken by Hitesh et al [3]. The need to reduce/transform input features in a manner most conducive to the dataset was appreciated at the outset of this work, and Word2Vec and TFIDF were both identified as viable candidates. TFIDF assigns word scores based on document appearance, and has been identified as a useful basis for establishing word importance, in addition broad categorisation of documents. Word2Vec is a shallow-Neural Network approach that maps input words to a vector space, identifying common linguistic context. A number of considerations were identified with respect to the work by Hitesh et al [3]. and their choice to employ Word2Vec. Key motivating factors in their choice were a) Dimensionality reduction and b) semantic relationship identification. In the corpus utilised for this work, it was identified that a) tweets were short, and shared significant vocabulary components (once pre-processing steps were executed) and b) topic identification provided, in the authors' view, a more readily accessible mode of sentiment grouping than semantic contextualisation. The two Deep-learning models employed in this study also allowed for inherent semantic encoding, and so specific pre-processing in this regard was judged extraneous. Further, as computational resources were limited for this survey, TFIDF provided a more manageable process. Lastly, it was considered less desirable to introduce a NN-based feature extraction into the experimental environment for non-NN methods. On that basis, TFIDF was identified as a more desirable vectorisation process, and was employed for this assessment.In addition, a more compact "hash vectorisation" approach was also compared, to serve as a benchmark for the TFIDF approach in non-NN modelling.

Further to this, a survey was conducted of viable model options for the non-Neural Network paradigm assessment. In this regard, the above work by Hitesh et al. was persuasive in setting out a basis for the use of Random Forest model, albeit in that case it was paired with Word2Vec, and so deviation in performance was anticipated. Tyagi et al. established a similar basis for the use of K-nn classifiers [4]. It should be noted that performance deviations were, once again, anticipated, as that study did not make use of vectorisation and instead employed a "bag of words" approach. A similar issue was identified with the use of Naieve Bayes classifiers, as employed by Suppala et al, and the decision was made not to include that model-set [5]. It was identified however that those models were not incompatible with a vectorisation step, and indeed may derive some benefits from it. Two regression-based methods were identified as suitable for this study based on analysis of the literature: Logistic Regression classification, as per Tyagi et al. and Support Vector Machines, as based upon assessment of Ahmad et al [6].

Having considered the non-Neural Network baseline models, an assessment was made of both Recursive and Convolutional approaches, identified as desirable in Van et al.[7] and Severyn et al [8]. RNNs presented several favourable characteristics for this task, notably capture of sequential information, and shared parameters. Given the repetitive nature, and context dependency, of the Tweet data, RNNs were assessed as suitable for this survey. Further to this, issue with gradients ("vanishing") were assessed as unlikely in this context, given the brevity of input data. CNNs were also identified as suitable, based on the assessment of Severyn et al. The ability to capture word sequences within the vectorised data was identified as desirable, given the dependencies inherent in certain sentiment-sensitive terms.

3 Experimental setup

Following steps identified in Hitesh et al. and Burns, a number of steps were taken before vectorisation in order to maximise information density within the input data.

3.1 Punctuation removal

A straightforward string check was employed to remove a variety of punctuation symbols. Given the level at which the data will be provided to the models for training, punctuation encodes no significant semantic information, and can be removed without detriment to model performance.

3.2 Stop-word removal

Stop-words are those words which do not convey "content" in the sense of specific references within the data, and instead represent common, intra-sentence data such as particles ("to"), prepositions ("on"), etc. These words were identified through the NLTK package, and replaced using the same methodology as in step a). While it is possible for data to be encoded within stop-words, given the aforementioned level at which data will be provided to the models (high-level – topics, key-words), these words can be removed without detriment to model performance.

3.3 POS-tagging

POS-tagging refers to identifying word roles within a sentence (noun, verb, etc.). This is necessary where, as in part d), lemmatizing is employed for word reduction.

3.4 Lemmatizing

Many words can be reduced to some simpler form (ie. “fish” from “fishing” and “fished”). Given the relatively small dataset, it was decided to perform lemmatizing (reduction of words to root, as above), rather than stemming. Stemming is a simpler method, where reductions are made on the basis of string-based rules (eg. remove all “ing” strings). This is a quick method, but can result in illegal word truncation. Lemmatizing, the removal of morphological affixes (the “ing”, “ed” in the example above), was chosen as providing a more suitable base for the task.

3.5 Emoticon encoding

It is common knowledge that social media users heavily use emoticons in their messages. That information seems to be particularly useful in terms of sentiment analysis. That is because emoticons are the very definition of sentiment - they are a substitute for facial mimicry and other things that are otherwise impossible to express in a text. In order to accommodate data to binary classification task (positive/negative), emoticons have been generalized into four categories: postive, negative, surprised, tounge emoticons. For example: emoticons such as ;] , :-}, 8-) will be changed into :).

3.6 Minuscule generalizations

Originally we wanted to employ spelling correction in order to increase data density. Unfortunately this turned out to be extremely hard in terms of tweets. They are full of slang expressions which are not part of most official dictionaries. That means that those expressions would be changed into words with different meaning. Due to that fact, we have created a small internal list of common missplellings and strings that have equal meaning (haha vs hehehe).

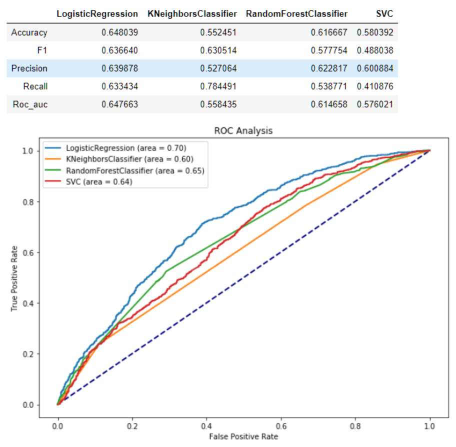
3.7 Vectorization

After the application of pre-processing steps to the data, a TFIDF matrix was developed using the scikit_learn package (TfidfVectorizer). This implicitly involves the following operations: creation of a term frequency matrix (identifying the occurrence or not of a given term within a document), a document frequency matrix (identifying the occurrence or not of a given term across the documents), and finally an inverse document frequency (log N/O, where N is the number of documents, and O was the number of documents in which a term occurred). The TF-IDF score can then be obtained by obtaining the product of a term frequency and IDF score.
TFIDF turned out to be an extremely memory hungry method. Full TFIDF matrix for training data requires approximately 1TB of memory, so it was necessary reduce it. This resulted in matrix reduction to approximately 300 most common words. In order to mitigate that issue, an alternative Hashing vector was developed using the scikit_learn package (HashingVectorizer). It is much faster and requires way less memory since it doesn't need to store vocabulary dictionary in memory.

3.8 Non-NN environment setup

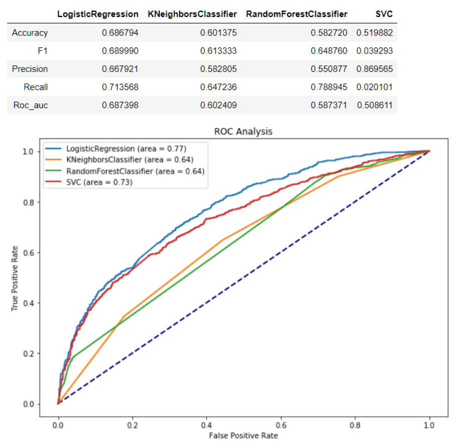
The substantive setup for the Non-NN setup has been described in the above section. Four models were selected for this section, based on the findings outlined in section 2: Logistic Regression, K-NN, Random Forest, and an RBF SVM.

In order to provide a benchmark, both TFIDF vectorisation and Hash vectorisation methods were employed, on the same data subsets. This subset comprised a representative sampling of 10K items from the greater dataset, to allow for training of multiple models. The initial results, comprising an Accuracy, F1, Precision, Recall, and ROC-AUC curve, are as follows:



	LogisticRegression	KNeighborsClassifier	RandomForestClassifier	SVC
Accuracy	0.648039	0.552451	0.616667	0.580392
F1	0.636640	0.630514	0.577754	0.488038
Precision	0.639878	0.527064	0.622817	0.600884
Recall	0.633434	0.784491	0.538771	0.410876
Roc_auc	0.647663	0.558435	0.614658	0.576021

(a) Hash Vectorisation



	LogisticRegression	KNeighborsClassifier	RandomForestClassifier	SVC
Accuracy	0.686794	0.601375	0.582720	0.519882
F1	0.689990	0.613333	0.648760	0.039293
Precision	0.667921	0.582805	0.550877	0.869565
Recall	0.713568	0.647236	0.788945	0.020101
Roc_auc	0.687398	0.602409	0.587371	0.508611

(b) TFIDF Vectorisation

Figure 1: Hash vectorisation vs TFIDF vectorisation

3.9 Pre-processing for deep learning models

For the deep learning models, we decided to use the embedding layers provided by the Keras library to generate the word vectors for every word in the dataset. Before training the models using the data, we had to apply some extra pre-processing. The embedding layers in Keras require arrays of integers of a fixed size as an input, so the first thing we had to do was transform the tweets from single strings to arrays of integers representing every word in the sentence. The first step was creating a dictionary of the words in the dataset. It is important to limit the words used in the vocabulary. This is because there are a lot of words that only appear once in all 1600000 tweets, leaving these words out would not impact the training of the model. Out of the 311787 unique words used throughout the dataset, only the top 2365 words occur more than 500 times in the dataset. Training vectors for every word would take a very long time for an insignificant gain. For these reasons, we decided to limit our vocabulary size to 25000.

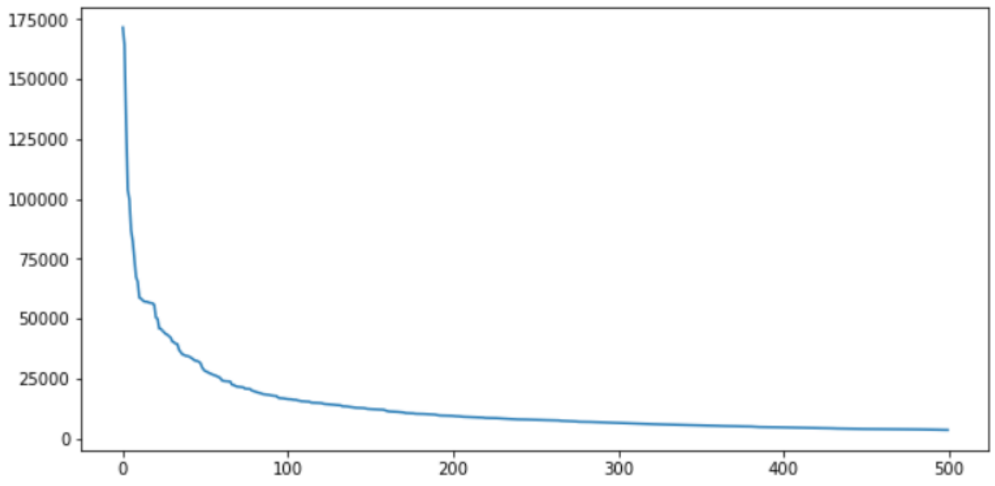


Figure 2: Frequency of the top 500 words

To create the dictionary out of the most frequently occurring words, we had to start by tokenising each tweet. We used the tokens to count the occurrence of every unique word, sort the words based on the frequency and only keep the top 25000 words. While counting the frequency of the tokens, we replace all twitter user handles with the tag "<usr>". We do this because naming another user in a tweet can have a lot of meaning, but each user handle is unique, and they all handles will be represented by a different token. We tried to negate this by replacing all usernames with the same token. After creating the dictionary, we replace every word with its index in the dictionary. If a word is not found, it is replaced by a "<unk>" tag, which shows that the word is unknown. Now the input data consist of arrays of varying sizes. The embedding layer requires all input data to have the same shape. To decide the length of a tweet we did some extra data analysis, we plotted the distribution of tweet lengths as seen in figure 2.

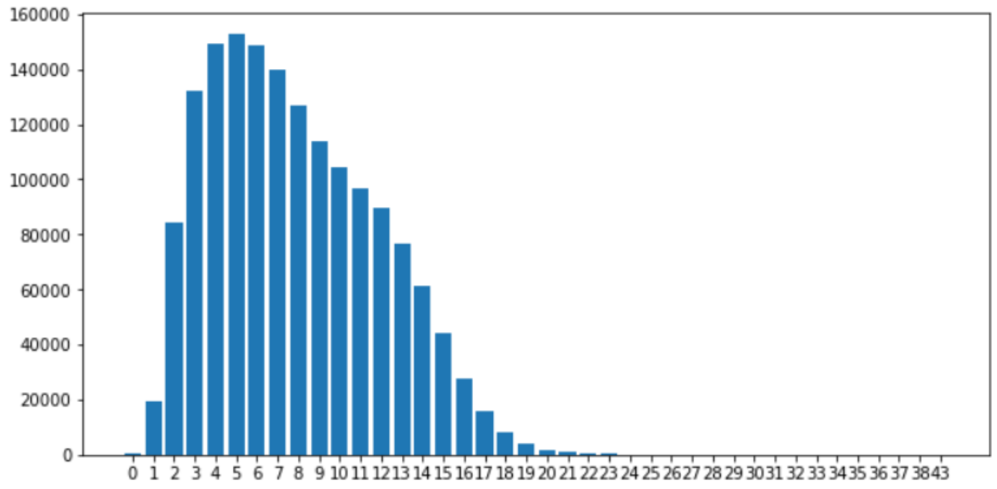


Figure 3: Distribution of the number of tokens tweets

The longest tweet has 43 tokens, but only one tweet has that many tokens. Most tweets have between 1 and 20 tokens. We chose a length of 25 tokens, all tweets with more than 25 tokens get reduced and all tweets with less than 25 tokens get padded up to 25 tokens using "<pad>". Now that the data is tokenised, padded and transformed using a dictionary, we split it into training and test data using an 80%-20% split. Later while training the actual models, the training data is split further into 80% training and 20% validation data .

4 Deep learning models

4.1 CNN

We experimented using two different types of deep learning models. First, we tried to train a Convolutional Neural Network (CNN). A CNN uses convolutional layers, the main goal of the convolutional layer is to extract pattern or specific features that occur often throughout the tweets in the training data. This is combined with a pooling layer that aggregates the information gathered from the previous layers and reduces the number of parameters [1]. A representation of the architecture we used in our CNN can be seen in Figure 3.

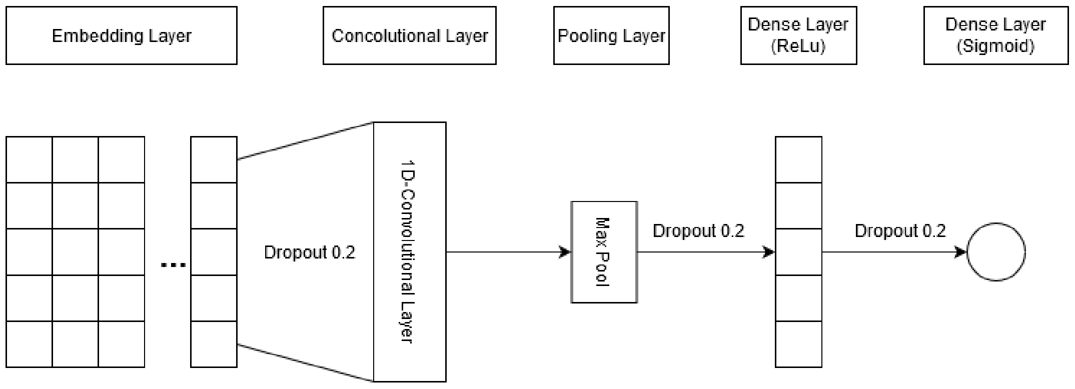


Figure 4: CNN model

Our model starts with an embedding layer that trains word embeddings with 300 features. A one-dimensional convolutional layer with a kernel size of 3 and a dimension size of 300. The results of the convolutional layer get pooled through a Max Pooling layer with a pool size of 2. The results get flattened into a one-dimensional array. The features extracted by the convolutional layer are piped to a Dense layer with 150 nodes and finally to a Dense layer with one node that makes the final prediction. To reduce overfitting, we have added a Dropout of 0.2 after the embeddings and before each dense layer.

4.2 RNN

The other model is a Recurrent Neural Network (RNN) using long short-term memory layers (LSTM). Unlike CNNs, when an RNN is trained using sequential data, the internal weights are shared across the sequence [2]. A tweet is a sequence of words, the LSTM layer looks at each word embedding in the sequence individually. The input is not only a single word embedding, but it also receives an extra input from the previous word. But sometimes there are words important to the context that occur later in the sentence. This will have a small impact if the LSTM only trains in one direction. That is why we used a Bidirectional LSTM, this means that there is a layer that goes over the sequence normally and a layer that goes over the sequence backwards. A representation of the architecture we used in our RNN can be seen in Figure 3.

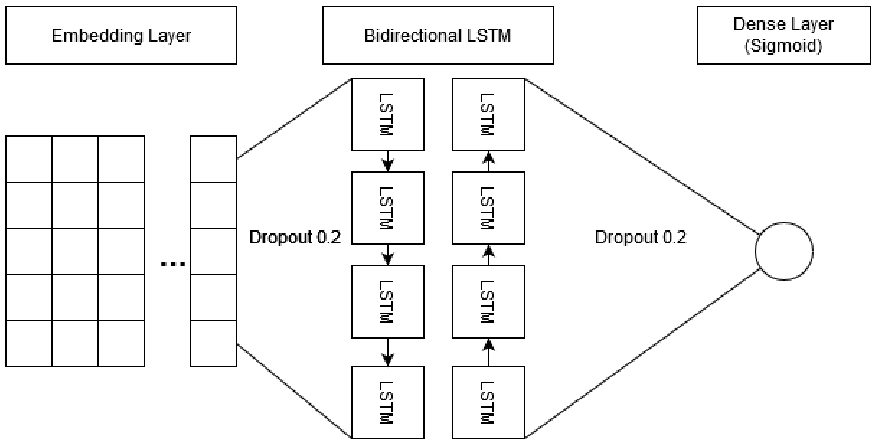


Figure 5: RNN model

Our model starts with an embedding layer that trains word embeddings with 150 features. The word vectors are passed to a Bidirectional LSTM layer that has 80 nodes in each direction, making it a total of 160 nodes. Finally, the results of the LSTM layer get passed to a Dense layer with 1 node and a sigmoid activation layer to get the prediction. To reduce overfitting, we have added a dropout layer of 0.2 after the embedding and before the dense layer.

5 Experimentation

We tried different variations for both neural networks. In both cases, we trained multiple models with different configurations e.g. Features in the embedding layer, more dense layers of varying sizes. That is how we found out that an embedding layer with 150 features gives better results in the RNN while the CNN model gave a better accuracy using an embedding layer with 300 features.

In the CNN we experimented with the dimension size and kernel size of the convolutional layer. We trained the model using high and low values and chose to use the configuration that gave us the best results. The first version of the model did not have a hidden dense layer. But we found that adding it improves the accuracy by 2%. To choose the size of the hidden layer, we trained the model %using 50, 100, 150, 200 and 250 nodes and we found that a %Dense layer with 150 nodes gave the best results.

We did the same thing with the LSTM layer in the RNN. Adding a hidden Dense layer in this model made the model too complex and helped it overfit.

5.1 Pre-trained word embeddings

The RNN was the best out of the two models. We wanted to see if it was possible to improve the model using pre-trained word embeddings. At the moment, the embedding layer generated word vectors using our own data. Pre-trained word embeddings are vectors that have been trained using a large dataset and are saved so they can be reused in other NLP tasks. Examples of popular pre-trained words embeddings are word2vec and GloVe These vectors are generated using bigger datasets and more advanced methods than our embedding layer. We wanted to find out if it would improve the accuracy of our models. We used the GloVe [9] word vectors with 100 features, which was trained using Wikipedia dumps and Gigaword corpora, by taking the embeddings of the words that appear in our dictionary and using them to create an embedding matrix. The words that appear in both our dictionary and the GloVe embeddings will have vectors, while the words that only appear in our dictionary will have a vector of zeroes. The matrix then replaces the weights of the embedding layer. We trained this model twice, first with a disabled embedding layer so the vectors aren't modified. Then with the weights disabled so the embedding layer modifies the vectors based on our data.

6 Results

Table 1: Deep learning model results

Model	Accuracy
CNN	0.78724375
RNN	0.795815625
RNN with GloVe (not trainable)	0.782503125
RNN with GloVe (trainable)	0.7985625

Each model was trained using a batch size of 50 to speed it up while not losing performance and a validation split of 10%. The models kept training until the validation loss stopped improving. This was different for every model. The dataset is completely balanced with 50% positive and 50% negative sentences, this allows us to use the accuracy metric because the balanced data will not create a bias towards one class, it can accurately show the performance of the model. Here we will give an overview of the results of all four models. The results in Table 1 were gathered by evaluating the model using the unseen test set we split off during pre-processing. The table shows the accuracy of each model, we can see that the results are very close to each other, there is a difference of 1.6%. between the best and worst-performing model.

We started by training the CNN and RNN without pre-trained weights. Both models were trained using three epochs, as more than three would cause the models to overfit and severely reduce the accuracy on unseen data. The RNN got an accuracy of 79.58%, which outperformed the CNN by 0.85%. That is why we used it to train the next two models.

The RNN with the disabled embedding layer took a lot longer to train. The validation accuracy after the first epoch was very low, and it took 8 epochs in total to get the best possible results. This model had the worst performance out of all four. This is because GloVe is trained using Wikipedia and other corpora that use a similar language structure. The language used in Wikipedia is a lot more formal than the one used in tweets. There are a lot of informal words used in the Sentiment140 tweets that do not appear in the GloVe embedding data, these words get assigned a vector of zeroes that can't get modified because the embedding layer is disabled. These gaps in the embedding layer can confuse the model while training because all these words would be considered similar to each other. The model with the enabled embedding layer trained a lot faster, only 4 epochs were required to get the best performance. Here these empty gaps of informal words were also modified while training which boosted the performance. This model gave the best performance out of all four with an accuracy of 79.86

7 Conclusions and Future Work

Based on the findings here, it is evident that both RNN and CNN models outperform their non-NN counterparts sizeably. Non-NN models did not break the 70% accuracy threshold, while all NN models detailed here saw significantly higher performance.

Future work in this vein might assess the underlying data, and identify further insights that could be identified there to better support the functioning of both the NN and non-NN models used here. While Word2Vec was out of scope for this inquiry, it would be interesting to examine if the additional data-insights provided by it would allow the non-NN models to perform closer to the NN benchmark identified here.

Further to this, greater emphasis on regularisation of the underlying data may have ramifications for generalisation of the model. Applying normalisation techniques (such as L2) to the dataset may reduce "noise", and improve model performance generally. In this vein, it may be interesting to assess how model performance varies across time, as the dataset used here was primarily from circ. 2009. Assessing linguistic trends was out of scope for this project, but might yield significant findings vis a vis the models ability to detect sentiment as distinct from contemporary mannerisms.

8 Footnotes

[1] Severyn, A. and Moschitti, A., 2015, August. Twitter sentiment analysis with deep convolutional neural networks. In Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval (pp. 959-962).

[2] Cliche, M., 2017. Bb_twtr at semeval-2017 task 4: Twitter sentiment analysis with cnns and lstms. arXiv preprint arXiv:1704.06125.

[3] Hitesh, MSR. et al. (2019). Real-Time Sentiment Analysis of 2019 Election Tweets using Word2vec and Random Forest Model. Jaipur: MU

[4] Tyagi, A. (2018). Sentiments Analysis of Twitter Data using K- Nearest Neighbour Classifier. IN: SRM

[5] Suppala, K. (2019). Sentiment Analysis Using Naïve Bayes Classifier. IJITEE: V8I8

[6] Ahmad, M. (2017). Sentiment Analysis of Tweets using SVM .IJCA: V177N5

[7] Van, V. (2018). Combining Convolution and Recursive Neural Networks for Sentiment Analysis. VI: HCM

[8] Severyn, A. (2015). Twitter Sentiment Analysis with Deep Convolutional Neural Networks. Santiago: CH

[9] Pennington, J., Socher, R. and Manning, C.D., 2014, October. Glove: Global vectors for word representation. In Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP) (pp. 1532-1543).