Approaching Tweet Classification With Different Neural Network Architectures

Gerard Cegarra Dueñas

April 2019

1 Introduction

In the era of social media as the main channel of news, content and opinions spreading, political campaigns are also conducted through platforms such as Tweeter. Fake news and contents are new weapons that have been increasingly used in the last years to manipulate the mass opinions and voting intention. In this paper, three different approaches of classification models are benchmarked in terms of accuracy, targeting the source identification of tweets from two opposite Spanish political parties during the last week before a national election. Solving this classification problem is a partial contribution to the main fight against mass manipulation over social media.

The whole code implementing the processes and models described in this paper can be found at the public repository https://github.com/gerardcd/nn_comparison_2. All the code has been implemented in the python programming language, using the Keras framework with Tensor Flow as backend to build the classification models.

2 The problem

The problem faced in this paper is the task of classifying the real source of a tweet. Given a set of tweets from two different accounts, we aim to build a classificator that takes as input the body text of a new tweet and outputs a class which identifies one of these two accounts. The data used for the final experimentation results is a fairly big set of tweets from the @podemos and @populares accounts posted during the week before the Spanish elections (28 Apr 2019)

This problem should be a feasible task as each party's style of writing and vocabulary is somehow constant through their repetitive campaign messages and slogans. This constantness should allow a model trained with a subset of the account's tweets (training set) to properly predict unseen tweets (testing set).

3 Data exploration

The data used to solve the presented classification problem is around 600 tweets coming approximately even from the Tweeter accounts @podemos and @populares. Any media like videos or images potentially contained in the tweets is not included in the used data. Hashtags, direct mentions to Tweeter users and urls are also removed form the tweets, as they conform two nearly exclusive sets of labels for each account, which would make the classification problem almost trivial, but would lead to useless models that doesn't work in the absence of this labeling. Finally, each word is stemmed to Spanish language.

After this cleansing first step, the data has to be encoded in a way that models can ingest tweets as their input. The encoding system used is to convert each word into an integer, and then convert the tweets into arrays of integers of a fixed length. Two decisions needs to be taken to implement this encoding: the size of the vocabulary used and the length of the tweets representation arrays.

In order to decide the length of the encoded tweets an histogram is performed on the clean tweets length (Figure 1). This histogram shows that the major part of the tweets have a length between 10 and 50 words. Actually, the 96.51% of the tweets have 50 words or less. Limiting the encoded tweet size to 50 words sounds reasonable as it will keep the most part of the information while avoiding some empty slots at the beginning of each encoded tweet.

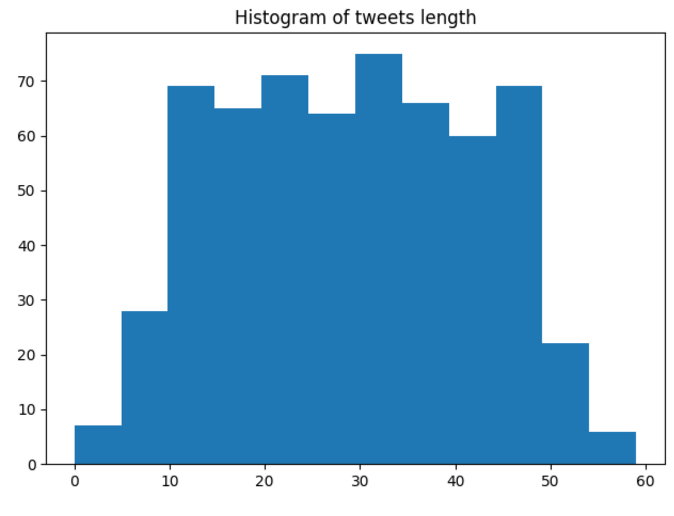

Figure 1: Histogram of tweets length

To chose a value for the vocabulary size, we start checking the number of different stemmed words that appear in the clean tweets set, which is 2063. In order to reduce this set for performance and accuracy improvement, we are also interested on the frequencies of these words, i.e. the number of times each word appears through the whole set of tweets. With these frequencies we can remove the less appearing ones. A model that learns with more common words in the specific domain will be more robust classifying unseen data.

In order to perform a cut in this set of 2063 words, the frequencies are calculated and sorted in descending order. Then, the values of this sorted array are accumulated (Figure 2). As we can see, a vocabulary conformed by the 1000 most repeated words is covering the 93.44% of word appearings in the tweets set.

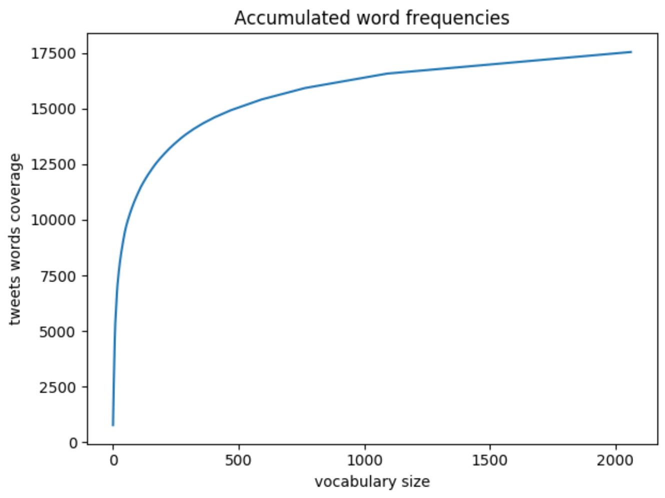

Figure 2: Accumulated word frequencies

The data cleansing process described in this section is implemented in the file *data.py* of the source code, and the data exploration is implemented in the *data_exploration.py* file.

4 Networks

In this section three different classification models are presented to solve the described problem (Section 2). These three models are built with three diferent neural network architectures: a Mutli-Layer Perceptron (NN), a Recurrent Neural Network (RNN) and a Long-Short Term Memory Network (LSTM).

The three presented models, NN, RNN and LSTM, are implemented in the files *nn.py*, *rnn.py* and *lstm.py* of the source code respectively.

4.1 Multi-Layer Perceptron

The first Neural Network implemented is a simple Multi-Layer perceptron. This network is conformed by 4 different layers. The first one is a regular densely-connected layer with a linnear activation function (no transformation applied to the output). The second layer is a dropout layer which will randomly set part of the inputs to 0 at each training update. The RLU activation function of this layer will also set the output values to its absolute value. This kind of layer is a common mechanism to avoid overfiting during the training process. The next layer is another regular dense layer that will take the input of the previous layers and output two values. This two values will be passed to a last activation layer that will apply a softmax activation function. The output of this final layer is the classification of the input. The output values represent the probability of the initial input to belong to each class.

Layer	**In, Out Shape**	**Activation**
Dense	1000, H	Linnear
Dropout	H, H	Rectified Linear Unit
Dense	H, 2	Linnear
Activation	2, 2	Softmax

Table 1: Multi-Layer Perceptron summary

Layer	**In, Out Shape**	Activation
Embedding	50, (50, E)	-
Simple RNN	(50, E), H	Hyperbolic tangent
Dropout	H, H	Rectified Linear Unit
Dense	H, 2	Linnear
Activation	2, 2	Softmax

Table 2: Recurrent Neural Network summary

The exact network architecture (Table 1) will depend on the parameter H, which indicate the size of the hidden layers.

The process of data encoding for this model is different from the one described previously (Section 3). In this kind of non-recurrent model, the order of the input data (words) is irrelevant. For this reason, and because the training process is fast enough to use big sized inputs, instead of using arrays of a fixed small size to represent each tweet, we are going to use a boolean array $t = \{t_i\}$ of size 1000 (the vocabulary size). Each position t_i of a tweet array is 1 if the word i belongs to the tweet t and 0 otherwise.

4.2 Recursive Neural Network

The second network built to approach the classification problem is a Recursive Neural Network. As opposite to the Multi-Layer Perceptron, in this kind of network architecture the position of the words in each tweet is taken into account for classification. For this reason we will be using this time the data encoded described before (Section 3). However, when each word of the vocabulary is converted to a integer, this integer represents a category from the statistical point of view, and the RNN expects numerical input data. To convert the categorical data into numerical data an embedding layer is introduced as the first layer of the network. This layer creates E new embedding dimensions and transform each word in a vector of size E. The coordinates of each word are calculated in a way that distances between word vectors make sense with the whole data context. The output of the embedding layer is a matrix of shape $(50, E)$: a vector of size E for each one of the 50 words of the input. Hence, the exact network architecture (Table 2) will depend on the parameters H and E this time.

The next layer of the model is the Recurrent one. The activation function used for this layer is the hyperbolic tangent function, which is the default option in the Keras framework. The rest of the layers are the same that in the NN model (Section 3)

4.3 Long-Short Term Memory Network

The last network proposed is a Long-Short Term Memory Network. This kind of network is very similar to the recurrent

Layer	In, Out Shape	Activation
Embedding	50, (50, E)	-
LSTM	(50, E), H	Hyperbolic tangent
Dropout	H, H	Rectified Linear Unit
Dense	H, 2	Linnear
Activation	2, 2	Softmax

Table 3: Long-Short Term Memory Network summary

one, but it implements a more complex mechanism in order to avoid the vanishing gradient problem that RNNs have. In our case, the network architecture will be exactly the same as the recurrent one (Section 4.2) but for the RNN layer, that will be replaced by an LSTM one, with also an hyperbolic tangent activation function. The embedding process and the three final layers remain the same. Hence, the exact architecture (Table 3) is again parametrized by the embedding size E and the hidden layers size H.

5 Benchmarking

This section describes the experimentation process and the performance results of the three implemented networks: NN, RNN and LSTM. As we have seen before (Section 4), the NN network architecture is parametrized by the size of its hidden layers H, and the RNN and LSTM architectures by H and also its embedding sizes E. Then, in order to benchmark these three architectures, we are going to extract performance metrics for different combinations of values for H and E.

For every possible combination of H and E values in the set of the first 10 powers of 2 ($H, E \in \{2^i \mid 1 \leq i \leq 10\}$), every architecture will be instantiated 5 times, trained with a randomly selected 80% of the tweets and tested with the remaining 20%. Every instance's testing accuracy will be calculated and averaged at the end of the 5 runs (Figures 3, 4, 5, 6, 7).

Another way to collect the results would be to select the best result among the 10 instances of each parametrization. However, the objective here is to find an architecture that is reliable for a real world sized problem. The aim of this benchmarking is to find an architecture that will give a good test accuracy in an scenario where the training process can not be repeated an arbitrary number of times. Hence, we are looking for an architecture that gives good perfomance in most of the cases it is used.

The training processes will be performed with a batch size of 64 and during 10 epochs. The experimentation process is implemented in the file *analyzer.py* of the source code, and the results visualization in *results_visualization.py*. The files with the averaged results for each architecture can be found under the *results* directory in the source code. These results include training loss, training accuracy, test loss, test accuracy and training time.

6 Conclusions

There are some notable conclusions that can be extracted from the experimental results. The first one is that the RNN

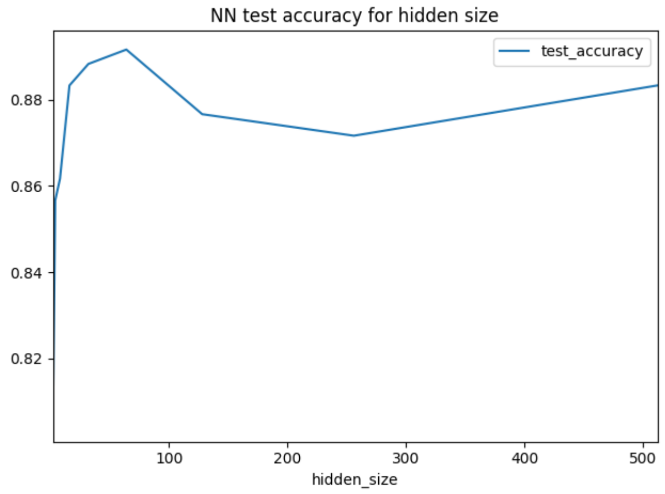

Figure 3: NN test accuracy for hidden size

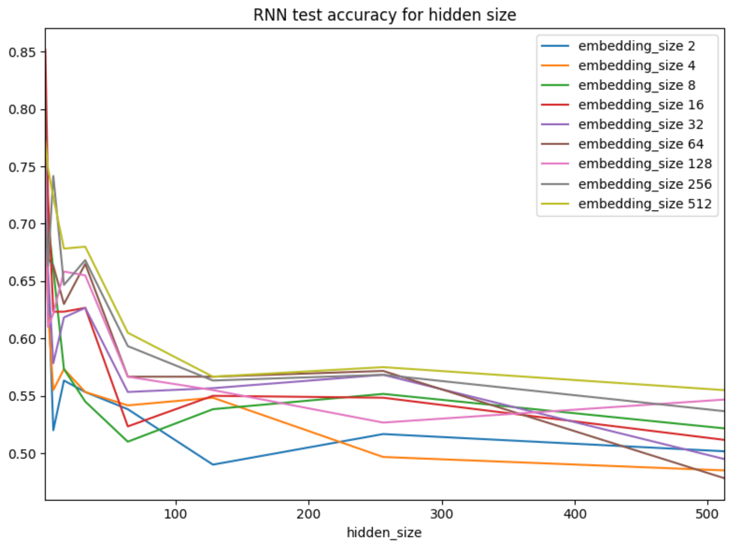

Figure 4: RNN test accuracy for hidden size

architecture has an unexpected behaviour with respect to both H and E parameters (Figures 4, 6). It seems to be working better for small values while bigger configurations should be able to detect better the patterns of the writing style. One reason could be that the bigger configurations need more epochs to get trained properly, but the reality is that this is only true for the case of $H = 256$ (this can be seen in the results file *results/rnn.csv* in the source code). The reason might be then, that the networks are overfitting, and the dropout strategy is not enough to avoid it.

Another notable phenomena is that in the case of the NN architecture, the testing accuracy seems to have a maximum at an H of 64 (Figure 3). Moreover, this maximum represents a 89.17% of test accuracy, averaged over 5 randomized runs, which is a good result. The LSTM test accuracy in relation to H (Figure 5) supports this conclusion. The lines representing LSTM's best Es (128, 256 and 512) have also a maximum at 64 H.

Finally, looking at LSTM test accuracy results in relation to E (Figure 7) we can also see that its best result is found

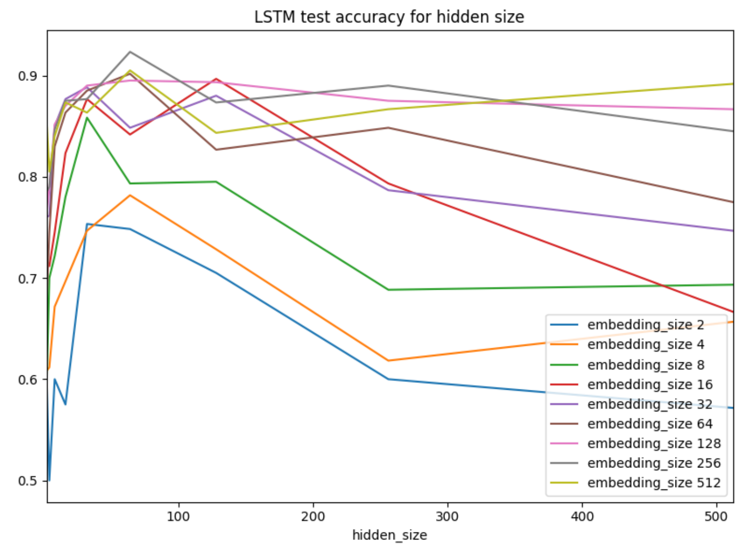

Figure 5: LSTM test accuracy for hidden size

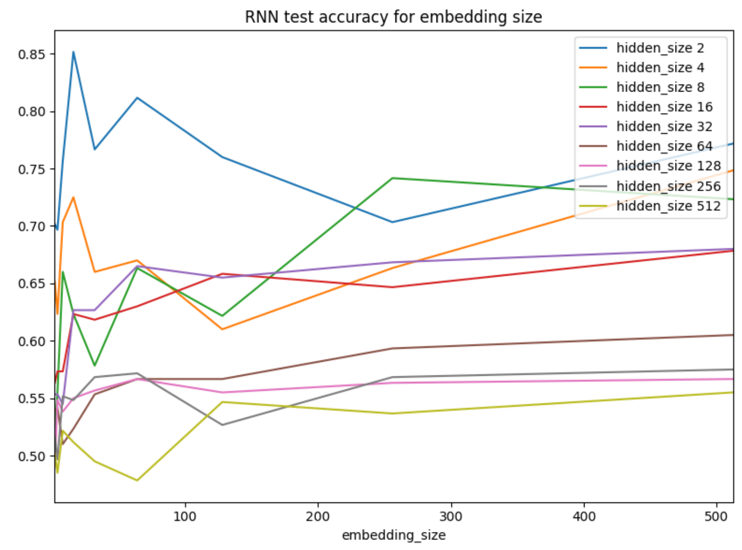

Figure 6: RNN test accuracy for embedding size

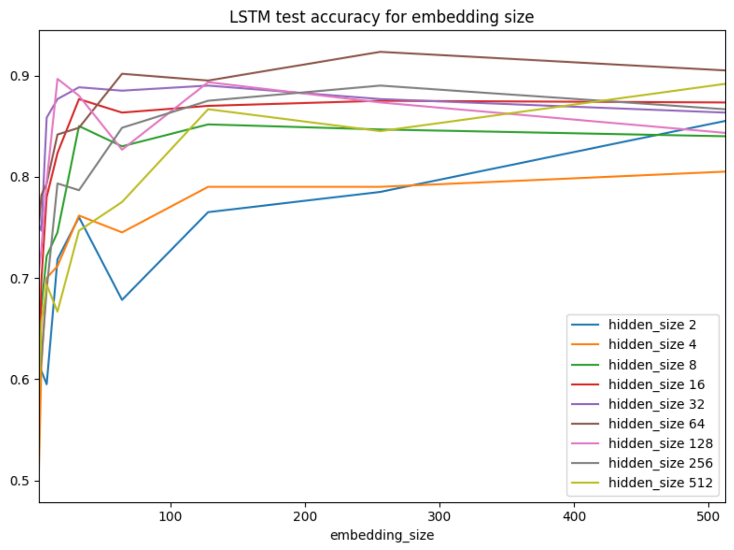

Figure 7: LSTM test accuracy for embedding size

with a H value of 64 and an E value of 256. This result represents a test accuracy of 92.33% which is better than the 89.17% of the NN.

Taking all these facts in consideration, its reasonable to claim that the best architecture for this particular problem is a configuration of $H = 64$ and $E = 256$.