Fake News Detection System

University of Piraeus
Department of Digital Systems
Stamatios Orfanos
stamatisorfanos@gmail.com

Abstract

With the advancement of technology, digital news is more widely exposed to users globally and contributes to the increment of spreading of misinformation online. That problem created the need for one of the most relevant applications in the field of Pattern Recognition and Data Analysis today, the Fake-news Detection system. In this paper, we present an detailed analysis, overview and challenges, that we faced during the development of our fake-news detection system.

1 Introduction

The rapid growth of the internet and the social media have created the problem of "fake news" or more precisely "false information", which is the spread of misinformation about a diverse variety of topics including subjects like health, environment, politics, economics and many other topics.

Given the scale of the internet and social media platforms it was impossible to find each and every article or post that contained misinformation. At that moment computer scientist realised that an automated model had to be made in order to find those articles or posts and mark them as "Misinformation" or "Fake News".

Previous studies in computerized deception detection have relied only on shallow lexicosyntactic cues. Most are based on dictionary-based word counting using LIWC (Pennebakeret 2007) (e.g., Hancock, Vrij (2007)), while some recent ones explored the use of machine learning techniques using simple lexico-syntactic patterns, such as n-grams and part-of-speech (POS) tags (Mihalcea and Strapparava (2009)).

This paper investigates the use of Natural Language Processing (NLP) and embeddings as the way to use data representation for our goals.

2 Methodology

A rough outline of the methodology used to develop the fake-news detection system includes the creation of an embedding layer as the input of the model. Furthermore the model consists of one hidden layers before the output layer, in which we get the results of the training. While training we have a validation split of 10%, which is enough given the size of the data set that we are using in this paper.

Next we are going to save the model in order to predict the legitimacy of unknown data like articles, posts or tweets and lastly use it for our application.

2.1 Data

Initially the data set we are going to be using in this paper comes from the website Kaggle. More specifically we used the fake-news data set (found https://www.kaggle.com/c/fake-news/data). This data set was quite useful, since it included a sufficient number of records to use for our model.

The data included the id, the title and the author of the information, the text containing the information and the label of the each data record. The label attribute has either 0 for reliable or 1 for unreliable information as values.

Table 1: Data Record

Id	Title	Author	Text	Label
0	The Forgotten Man	Napolitano	...	0
1	F-35 Still Alive	Swanson	...	1
...	...	...	...	...
20800	...	...	...	...

In this data set we have 20800 total records we all the attributes above, but for our purposes we are going to use the attributes text and label in order to classify the reliability of the records. We are also going to drop any records that do not contain any text, since this is our only criterion to achieve our goal.

2.2 Data preprocessing

After we decided about the attributes we had a data set that had the following structure:

Table 2: Data Record

Text	Label
According to resources ...	0
The major problem ...	1
...	1
...	0

At this point we started the preprocessing procedure, in order to insert high quality data to our embedding and our model. We discovered that the text attributed had capital letters, punctuation, non-alphabetical terms and even XML, HTML and other markup tags. All those characters needed to be removed from the text since they did not offer any information for our model. When it comes to the removal of capital letters we realised that in case we did not lower every character the words "One" and "one" would be considered different increasing the vocabulary size while decreasing the effectiveness of our embedding.

Furthermore we had to remove all the stopwords from our text, due to their frequent presence in all of the text records that could possibly influence the result of our prediction. In other words if we had a unreliable text that contained multiple times the word "a" would influence our model to assume that any sequence with multiple occurrences of "a" would be unreliable, something that would hurt the performance of our model since the word "a" does not provide much information about information.

We achieved those removals by tokenizing our data and checking each individual token through our removal tests. Lastly we removed all the tokens/words that had length equal to 1, since we discovered that after all the removals we had some tokens with 1 character. For example the word "Jimmie's" would be transformed to "Jimmie s" creating a additional token with no use for our model.

Finishing our data preprocessing we detokenized the token sequences, creating a big sentences in order to insert our data to our embedding.

Example of our preprocessing procedure:
Before:Sad news, a plane crashed @Los Angeles
After:sad news plane crashed los angeles

2.3 Word2Vec

Word2vec is a technique for natural language processing. The word2vec algorithm uses a neural network model to learn word associations from a large corpus of text. Once trained, such a model can detect synonymous words or suggest additional words for a partial sentence. As the name implies, word2vec represents each distinct word with a particular list of numbers called a vector.

Skip-gram and Negative Sampling
In our project we implemented the Skip-gram model, that predicts the context (or neighbors) of a word, given the word itself. The model is trained on skip-grams, which are n-grams that allow tokens to be skipped. The context of a word can be represented through a set of skip-gram pairs of (targetWord, contextWord) where contextWord appears in the neighboring context of targetWord.

In this paper we used the Gensim library for our Word2vec model, that has four basic parameters. The first parameter is the sequences that takes a list of sentences, that we tokenized with the getReviewLines() method.

The second parameter is the embedding size, which indicates how many values we are going to use in order to uniquely represent a word. The embedding size defines the size of the embedding vector and in our case we are using an embedding size of 128, which means that each word is going to be represented by 128 values.

The window is the third parameter and it gives us the number of words before and after the target word of the sentence. The fourth and last parameter is the minimum number of words that we are going to use for our model, in case a sentence is smaller than the window. We decided that the window of the model is going to be 5, since we have big enough sentences most of the time with the minimum count being 1.

Table 3: Word2Vec saved model

trump	-1.8746284	5.4467015	...	-1.8335162
clinton	-2.0395045	3.4139013	...	-1.8159785
state	-0.9414992	-0.008999801	...	1.5200808
...	...	...	...	...

Note: The Word2Vec model is saved in the project folder

2.4 Embedding

The embedding layer is the input of our model, so we have to make the weights of the model firstly. In this step we use the Tokenizer function from Tensorflow Keras preprocessing in aim of transforming every word to an integer representation, step necessary for our model. Before using the Tokenizer function we use the getReviewLines() function to tokenize the sentences. Given the sentence below, the procedure goes as follows:

sentence = trump won state over clinton
reviewLine = getReviewLines(sentence)
reviewLine = "trump", "won", "state", "clinton"
Tokenizer()
wordIndex = ["trump": 1,"won": 2, "state": 3, "clinton": 4]
sequence = [1,2,3,4]

Table 4: Tokenizer Word Index

trump	1
won	2
state	3
clinton	4

The key advantage of using the Tokenizer() function is the fact that frequent words get smaller values in the word index, which is quite helpful computationally. Given the results of the Word2Vec model and the Tokenizer index and sequences we are able to complete the embedding layer.

In particular we use the function getWeightMatrix() to make the weights of the input layer. This function takes the word index made by the Tokenizer and replaces the words of the Word2Vec model with the integer counterparts that we just made. This means that the embedding has the following structure:

Table 5: Weight Matrix for Input Layer

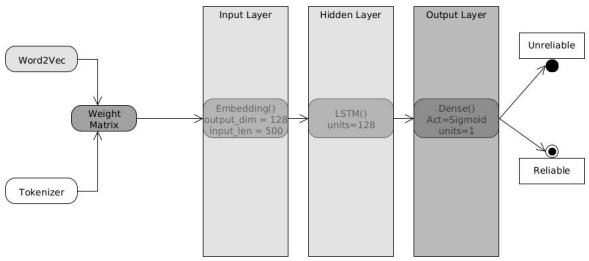
1	-1.8746284	5.4467015	...	-1.8335162
4	-2.0395045	3.4139013	...	-1.8159785
3	-0.9414992	-0.008999801	...	1.5200808
...	...	...	...	...

A decisive detail that had a major positive influence in the performance of the model is the maximum size of the vocabulary, decided in the Tokenizer() function. After some experiments we realised that the maximum vocabulary used was more than 180.000 different words, so we decided to use slightly less than half of the initial vocabulary at about 90.000 words. The difference was significant, because without that vocabulary limitation our model accuracy was falling dramatically after the second epoch.

2.5 Model

Having made the weight matrix we start by defining our input layer, using the Embedding() function provided by Tensorflow Keras. As parameters we pass the vocabulary size, the weight matrix and the maximum length of each sequence. As a maximum sequence of tokens we choose 500 tokens, since the average sequence is about 475 tokens with the majority of the sequences of the data being around 450-550 tokens long.

Next was the first hidden layer that used a LSTM() architecture, with 128 units. LSTM is a recurrent neural network (RNN) architecture that has the ability to "remember" the values over arbitrary intervals. As we already mentioned LSTM() is a RNN, that is quite more complicated in comparison to a simple Dense() layer, which means that the time required to train the model is a bit longer, but we are able to minimize the number of epochs we need to sufficiently train our model.



2.3.1: Model Architecture

Lastly for the output layer we used a Dense() layer with 1 units, since the problem given has two possible results.

The model was formed after experimentation with all the possible different hyper-parameters and a lot of testing with architectures.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 500, 128)          21030016
_________________________________________________________________
lstm (LSTM)                  (None, 128)               131584
_________________________________________________________________
dense (Dense)                (None, 1)                 129
=================================================================
Total params: 21,161,729
Trainable params: 131,713
Non-trainable params: 21,030,016
_________________________________________________________________
```

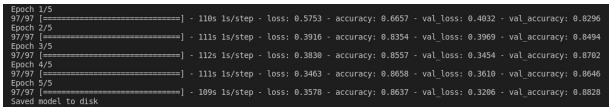
2.3.2: Model Summary

For the training of our model we used the Adam optimiser, with the loss function being Binary Crossentropy, because we have to possible results. We also used the metric parameter where we wanted to get the accuracy and the validation accuracy of our model.

3 Training Results

The main goal of this paper was to try to achieve a descent accuracy, avoiding any chance of over-fitting, while minimizing the loss metric in the same manner.

We trained the model for 5 epochs and with a batch size of 128, since any bigger batch size would create some performance issues, especially with the limited hardware we had in our possession.



3.1: Training Results

3.1 Accuracy Metric

According to the results of the training we were able to achieve the accuracy goal of 85% or more.

Firstly the accuracy of the model increases as the epochs pass. After countless experiments we discovered that after 6-8 epochs we had a continuous slight decrease of the accuracy, a sign of over-fitting in our data leading us to the choice of 5 epochs.

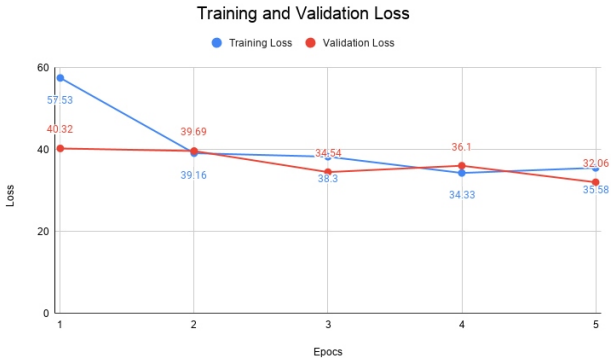
Similarly the validation accuracy is slightly better than our training accuracy as expected, due to the fact that we use all the nodes of our model to calculate the validation accuracy.

3.1.1: Accuracy Training Results

3.2 Loss Metric

The loss of the model is pretty high at a first glance, but after a couple epochs the loss metric decreases substantially

Again there are some positive aspects we can get from our experiment. The positive aspect of our model is the fact that the loss metric is steadily decreasing as the epochs go by.



3.1.2: Loss Training Results

As we can understand from the graphs, we were able to avoid the problems of over-fitting and under-fitting, since we have no steep increase of the loss metric or big drop of the accuracy metric in contrast to the first iteration of the model.

4 Evaluation

4.1 Predictions Accuracy

In the test.py file we load the model, and use the evaluate() function to find the accuracy of the model for totally new data, from the testSet.csv made in the preprocessing procedure. The calculated loss is 79%, while the accuracy of the model is 67%. Initially the results were quite bizarre, but after a quick investigation we realised that the vast majority of the test data had a length of less than 100. This piece of information indicates that the initial split between the train and test data send the tweets and the social media posts to the test set. Unfortunately after some in depth experimenting we found out that if a text has less than 200 tokens the accuracy falls dramatically at 50%.

4.2 Predictions Time

The time needed was just over 3 sec for 1000 predictions.

4.3 Training Time

In this paper we were able to track the time of each procedure. Firstly the time needed to complete the preprocessing of the data was 173.23 sec or 2.8 minutes, a descent result when taking into consideration the amount of text data we had to process.

Secondly the time to train the Word2Vec model, make the weight matrix and train our model came to 646.32 sec or around 10 minutes, which makes the training quite fast given the size of the training data.

Note: In our project there is a folder Time, in which we automatically are storing the time needed to complete a procedure.

5 Conclusion

In this paper we were able to achieve good results in both the accuracy and loss metrics in the training and the validation data. Unfortunately we were not able to make accurate and consistent predictions for texts with less than 100 tokens. The main reason behind that limitation of our system is the big padding added to the sequence, giving the model close to zero information about the text.

In conclusion we were able to achieve our initial goals of making a descent Fake-News Detection System.

References

[1] Tensorflow : https://www.tensorflow.org/tutorials/text/word2vec

[2] Michael Pi *https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21*

Published 24 September 2018

[3] https://adventuresinmachinelearning.com/gensim-word2vec-tutorial/