Text Analytics
4th Assignment

Galanos Vasileios - p3351902

January 19, 2021

This report is one of the two deliverables which we provide as a solution to the 4th Assignment for the course of Text Analytics of the MSc in Data Science at Athens University of Economics and Business. Accompanied with this file comes also the following link.

- Google Colab Link.

Here, you can find the code written for the assignment. It is hosted in Github and Google Drive and can be opened through Google Colab. Below we provide an overview of what we have created, explanations for methods and patterns used, descriptions and opinions on tools, algorithms and practices.

Exercise 1 - Sentiment classifier using RNN

For the purpose of this excercise we have implemented a sentiment classifier for tweet texts extracted from the Twitter social network using a Recursive Neural Network. We decided to occupy ourselves with Exercise 1 instead of Exercise 2, as this fourth assignment is the natural development of the third assignment and as the previous time we completed exercise 9 using MLPs, this time using RNNs we could examine, how better our sentiment classifier could become. The dataset that we used, is the same as that we have used in all four exercises and is named "Sentiment140" containing 1.6 million tweets.

Packages and Datasets

We start this assignment by installing all missing packages and then we mount our Google Drive in order to load and save data without losing them between runtime sessions. We then check the GPU that is assigned to us from Colab in case we want to restart our session in order to get something better. Moving on, we import all the libraries we will later use in our code and download, unzip and load our fastext word embeddings. These embeddings are pre-trained word vectors for 157 languages, trained on Common Crawl and Wikipedia using fastText. These models were trained using CBOW with position-weights, in

300 dimensions , with character n-grams of length 5, a window of size 5 and 10 negatives. Next we download and unzip our dataset from the following link provided by the Stanford University.

- Sentiment140 Link.

Moving on, we load the data to a dataframe choosing "ISO-8859-1" for decoding as there were major failures when using Unicode, or "UTF-8" . After having a closer look at our data, we drop the extra columns, that are not usefull for our assignment, as "text" and "sentiment" are the only columns needed in our case. Our dataset uses the "sentiment" column to denote the sentiment of the tweet, using "0" for negative, "2" for neutral and "4" for positive sentiment. As we discovered, although our dataset supports three labels, it contains only two of them (positive/negative). We change the value of sentiment "4" to "1" in order to give our results a binary look. We then, add a block of code, where we can lighten our colab storage by removing files that we won't be needing in the following sections. In the following code block, we print the count of the column "sentiment" and check how many values there are from each category. As we can see, in our case we only have negative and positive values equally divided, thus a balanced dataset.

Data Preparation

Next, we create two dictionaries which will later help us clean and manipulate our dataset better. The (*load_dict_smileys*) maps most commonly used smiley faces with their corresponding meaning and (*load_dict_contractions*) maps most commonly used contraction of words to the corresponding non-shortened phrases.

In the following code block, we create a function (*tweet_cleaner*) that cleans our text from characters that are not usefull for the purposes of our assignment. The method is mainly based on regular expressions which help us achieve our goal. Some examples of cases and characters we want to remove are: references to other users via @ character, links to websites, single character words, hashtags, number and finally multiple spaces. We also occupy ourselfs with cases of html tags left in our texts and we care about removing faulty characters which might have been the outcome of the decoding process of the text. Next, we utilize the two dictionaries we created to map contractions/slang and smiley faces in actual words with the same meaning. Furthermore, we replace three or more repetitions of letters with only two in order for multiple misspelled words to map in a unique one. Finally, we replace emojis with their corresponding word meaning and lematize each word to merge multiple versions of the same words into one. [1]. For example the tweet:

[1] https://towardsdatascience.com/fasttext-sentiment-analysis-for-tweets-a-straightforward-guide-9a8c070449a2

Figure 1: Raw tweet

will be mapped to the following word sequence:

```
lot of rock thumb up fell fromm the sky smiley
he will consider taken an umbrella love
```

Moreover, after taken a subset of our data we shuffle it, because we have noticed that the positive sentiment tweets are placed lower on the dataset and we don't want that messing with our outcome [2]. For each of the tweets in our sample, we apply our cleaning function and then we keep only the rows which containt text after the process described (empty tweets will be dropped).

Dataset Splitting

In the following code block, using the dataset we have manipulated until this point, we split it into three different sets:

- The largest set, will be our training set, consisting of the **80%** of the dataset.
- The next part, will be the development set, which will be used for hyper-parameter tuning and consists of the **10%** of the dataset.
- The final part, will be our test set consisting of the remaining **10%** of our dataset.

Some statistics on the dataset

In order to understand our dataset better, we continue with extracting some statistics from our dataset. First, we get the documents contained in each of our sets.

[2] shuffling the sentences makes the training more unbiased

Documents per set

Training set	80000
Development set	10000
Test set	10000

Moving on, we also present some statistics on the length of the documents for each set. These statistics contain the mean, standar deviation and max length for each set.

X dev mean: 12.95 with std: 7.14 and max: 34

Statistics on length of documents pes set

data set	**mean**	**std**	**max**
Training set	12.91	7.14	46
Development set	12.95	7.14	34
Test set	12.98	7.16	41

Tokenizing - Sequences

In this section, we initialy turn our sets to lists to make them easier to handle in the following processes. Then, we utilize functions provided by tensorflow to tokenize and create text sequences. Initially we import and use the Tokenizer provided for us from (*tensorflow.keras*) and fit it on our texts which updates the internal vocabulary of the tokenizer based on a list of texts we provide. Next using the same tokenizer, we turn our texts (in each set) into sequences using the (*texts_to_sequences*) which transforms each list of texts, into a sequence of integers. Finally, we use the (*pad_sequences*) function, on our sequences. This function transforms a list of sequences (integers) into a 2D Numpy array of shape we provide.

Embedding matrix with fasttext pre-trained embeddings

Moving on, we create an embedding matrix, using fasttext pre-trained embeddings and the unique tokens we can find through the (*word_index*) list of our tokenizer. The matrix we end up with, has 50002 * 300 dimension. This embedding matrix, will be later used as input to the weights of the embedding layer of our model.

Transforming y vectors

Next, we prepare two dimension y vectors to work with Keras. In order to have 2 values for (*predict_proba*) we have DENSE 2 output in keras and so the method expects a two dimensional y vector in any case. We create it by stacking vertically the suplement of y.

Classification metrics

We created our own function (*get_scores*) that calculates the following metrics, for user-defined classification thresholds as well as for all the classes:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \quad \text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \quad \text{F}_1 = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \quad [3]$$

Similarly we use macro-averaging to get a more robust estimate for all the classes

$$\text{Macro - Precision} = \frac{1}{n} \sum_{i=1}^{n} \text{Precision}_i$$

$$\text{Macro - Recall} = \frac{1}{n} \sum_{i=1}^{n} \text{Recall}_i$$

$$\text{Macro - F}_1 = \frac{1}{n} \sum_{i=1}^{n} \text{F1}_i$$

In this section we also declare the (*print_scores*) which based on the classifier used, utilizes a different prediction method, gathers our scores using the affomentioned function and then prints them in a presentable way. Moving on, we also create another method named (*print_classifier_scores*) which simply uses the previous method and prints our scores for each of our sets, formatted.

Turning indexes to centroids

In this next code block, we create a new custom class named (*Indexes_to_Centroids*). With this class, we transform dataset with indexes to a dataset with centroids. We use this in pipelines, thus the **sklearn** classifiers can be fitted with the same data that our RNN is.

Classifiers

Moving on to the main part of the assignment, we now provide a Recursive Neural Network we found through experimentation, but before developing the RNN, we present two linear classifiers used in the second assignment. The first is a baseline majority classifier and the second is our best probabilistic classifier from the second assignment. The purpose of presenting these classifiers is to see how well, we can classify our tweets to the correct sentiment and how can our RNN increase our scores when classifying.

Our baseline classifier like the last time is, the DummyClassifier from sklearn, which we parameterize to always return the most frequent result. We fit our

[3] We checked whenever the denominator was zero, and set the entire score to zero, in order to avoid arithmetic errors

training data on the classifier and then utilizing the methods we have previously created, we get our results. We present our results below:

All the scores presented bellow are measured for a classificiton threshold $t = 0.5$:

Baseline classifier

Training set			
	Precision	**Recall**	$\mathbf{F}_1$
Class 0	0	0	0
Class 1	0.501	1	0.668
Macro	0.251	0.5	0.334
Development set			
	Precision	**Recall**	$\mathbf{F}_1$
Class 0	0	0	0
Class 1	0.504	1	0.671
Macro	0.252	0.5	0.335
Test set			
	Precision	**Recall**	$\mathbf{F}_1$
Class 0	0	0	0
Class 1	0.488	1	0.656
Macro	0.244	0.5	0.328

For our next classifier we chose the SGDClassifier from sklearn. This is a set of linear classifiers with SGD training. For the loss parameter of the classifier we choose "log" which gives us logistic regression and so in total we have basically Logistic regression as the classifier, but using SGD (Stohastic Gradient Descent) training. This classifier as it was also mentioned above was the one to give the best classification results in the second assignment. Then, we train our model and again utilizing our method, we get the scores of the model on all three of our sets. We present our results below:

Logistic Regression classifier

Training set			
	Precision	**Recall**	$\mathbf{F}_1$
Class 0	0.715	0.787	0.75
Class 1	0.765	0.688	0.724
Macro	0.74	0.738	0.737

Development set			
	Precision	**Recall**	$\mathbf{F}_1$
Class 0	0.712	0.789	0.749
Class 1	0.768	0.687	0.725
Macro	0.74	0.738	0.737

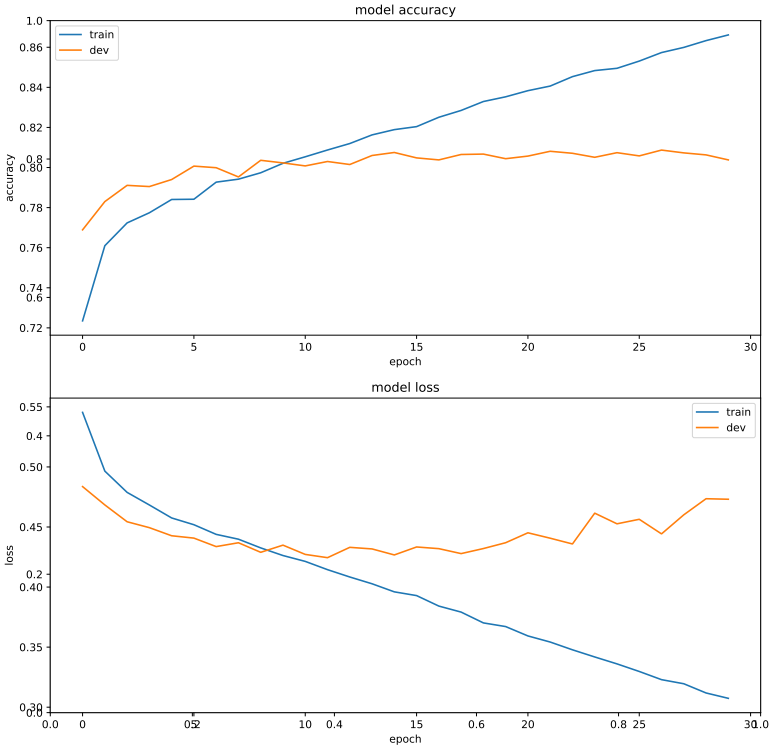
Test set			
	Precision	**Recall**	$\mathbf{F}_1$
Class 0	0.725	0.797	0.76
Class 1	0.762	0.682	0.72
Macro	0.744	0.74	0.74

RNN classifier

Next, we create a new class which we will be using later on when training our neural network. This class will be passed as a callback and will return us metrics like precision, recall and f1 score. Moving on, this time, instead of taking the last hidden state, we use self attention in order to take a weighted sum of all hidden states of our RNN. In order to do that we add implementation for a custom class used later as a layer on Keras which was provided in the class labs. The first function here is the (*dot_product*) which given the input and the weights, as its name suggests, returns the product of the two. Next, we have the linear attention class which basically returns a weighted sum of all the hidden states of the RNN. Similar is the behaviour of the deep attention class. In this case basically we use an MLP instead of a simple linear layer.

Now it is time to a create our model using Tensorflow and Keras. We initially add an Input layer with size equal to the max sequence length that we found. Each layer is passed as an input to the next one in order to forge together one united model. The next layer is our embedding layer taking as input the embedding dimesion and the embedding matrix we have created, followed by a Dropout layer with rate 0.4. The numbers applied here are found through hyperparameter tuning using RandomizedGridSearch and can be found in the end of the report at Apendix A. Moving on, we add a bidirectional LSTM giving our network the recursive feature now, with the size of the LSTM being 300. The next layer is again a Dropout layer using 0.33 rate, followed by a LinearAttention layer as it proved to have slightly better results than the DeepAttention layer. Next, we add a new Dropout layer with 0.33 rate, followed by a Dense layer with 400 units and then another Dropout layer with its rate being 0.4. The last layer is Dense layer with 2 units and a softmax activation function, in order to convert the numbers to probabilities, giving us the final results. We then, compile our model using categorical crossentropy as loss function and the Adam optimizer and save the best weights for our model for later use. Finally, we fit our model with our training data using the callbacks we have previously created and set it to run for 30 epochs, evaluating on the development set. After training, we get our model evalution scores which we present below for all of our sets. Below we also present our model history graph and also our results below:

Figure 2: Keras model evaluation curves



Model Evaluation scores

Metrics	**Value**
Test precision	0.81
Test f1	0.81
Test accuracy	0.81

RNN

Training set			
	Precision	**Recall**	$\mathbf{F}_1$
Class 0	0.9	0.902	0.901
Class 1	0.902	0.901	0.902
Macro	0.901	0.901	0.901
Development set			
	Precision	**Recall**	$\mathbf{F}_1$
Class 0	0.803	0.814	0.808
Class 1	0.815	0.804	0.809
Macro	0.809	0.809	0.809
Test set			
	Precision	**Recall**	$\mathbf{F}_1$
Class 0	0.816	0.821	0.818
Class 1	0.811	0.805	0.808
Macro	0.813	0.813	0.813

Precision-recall curves

Using our own function, we implemented plot_precision_recall_curve that, given a list of thresholds, calculates **precision** and **recall** and stores them to lists. Then we calculate the area under curve and plot a figure with those metrics for each threshold value.

Figure 3: Precision Recall Curves: RNN

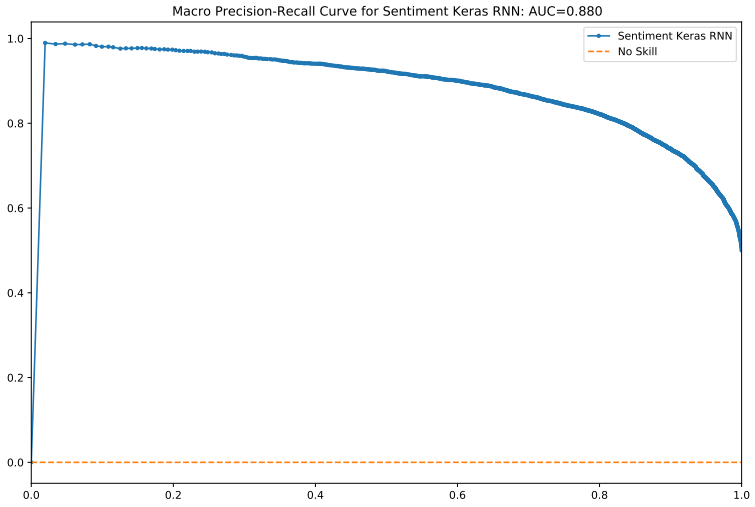
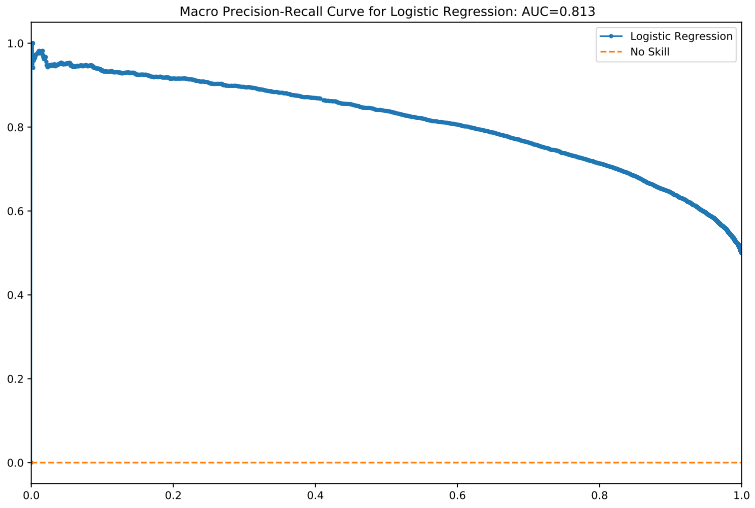


Figure 4: Precision Recall Curves: Logistic Regression

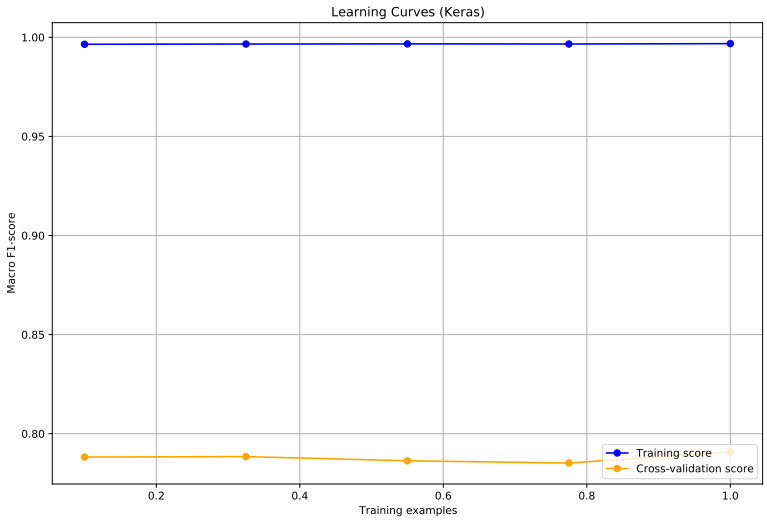


As we can see, in terms of **AUC** (area under curve), the **RNN classifier** is the best among the others and also as we saw above, it scored better than all other classifiers in all previous sections metrics. This proves that these kind of figures have great value when one evaluates different classifiers and also, combined with other metrics, can provide userfull information for safe conclusions. As we know, just because one classifier outscored the other on a certain threshold, that doesn't mean that it is the overall best classifier, but in our case all data point to that direction.

Learning curves

Moving on, although it was not a deliverable for the exercise, we thought it was interesting to examine the learning curves for the RNN. Utilizing the graph below, we can comment on the following. The model we have created seems to have great capacity and really overfits our data. We chose this specific model as it gave the best results on our test set among many others we tried. Another reason for choosing this model is, that it was the best suggestion from our randomized grid search. It is obvious, that what we can set as future objective for our model, is to drop its major overfit.

Figure 5: Learning Curves: RNN



Bootstrap statistical significance tests

In order to derive a safe conclusion on which classifier outperforms the others, we use **Bootstrap Hypothesis Tests**. Let A the best classifier in terms of some metric on a test set and B another classifier that we want to compare it with. The main idea here is that we assume the null hypothesis (H_0) is that the difference between the two classifiers, in the chosen metric, would be zero. The alternative hypothesis (H_1), is that indeed, there is a significant difference between the evaluation scores not a random fluke.
Let $\delta(x) = \text{Macro-F1}_\text{A} - \text{Macro-F1}_\text{B}$, be the difference calculated on all the test set.

We state the following hypotheses as the null and alternative one:

$$H_0 : \quad \delta(x) = 0 \quad VS \quad H_1 : \quad \delta(x) > 0$$

Here, we define $\delta(x_i^*)$ as the differrence between the classifiers on a simulated dataset, sampled from the test set with replacement. We calculate the p-value as $p - value \approx \frac{s}{b}$ where s are the times that $\delta(x_i^*) > 2\delta(x)$, as the algorithm of Berg & Kirkpatrick states.

if $p - value < 0.01$, we have strong statistical evidence to reject the Null Hypothesis, thus embracing the alternative hypothesis as the correct one (i.e. accept that A's victory was real and not just a random fluke).

Applying this test using our **RNN** as our best classifier and comparing it to all the others we got the following results:

RNN VS		
	Baseline	**Logistic Regression**
p-value	0	0

As we saw in the results, we reject the null hypothesis in all cases. Thus, the **RNN classifier** has significant difference with the **Baseline classifier** and the **Logistic Regression classifier**.

Appendix A: Random Gridsearch

We used various different architectures for our RNN classifier. We also experimented with dropout rates and also the sizes for some layers. We found a pain-less way to test all those different set-ups using Randomized GridSearch. We first applied a wrapper to define our keras RNN to sklearn. Then we run it for various values in order to find the best drop out rates and the optimal sizes for our LSTM and also our Dense layer. One last thing we wanted to check was whether our model would get better results using LinearAttention layer or DeepAttention. Since we could not change the whole model and running the same process two times would be cost worthy, we set up a new variable which is given to the Randomized GridSearch and determines whether the next run would be done using the one or the other layer. We also, set up our search to check 8 possible models and use 3 fold cross validation on each one to be sure about each of our results. Finally in order to speed up the process, we used only one third of our original data.

```
Best: 0.784576 using {'lstm_sz': 300, 'is_deep': True, 'drp4': 0.4, 'drp3': 0.33, 'drp2': 0.33, 'drp1': 0.4, 'dns_sz': 400}
0.783971 (0.003611) with: {'lstm_sz': 300, 'is_deep': True, 'drp4': 0.33, 'drp3': 0.33, 'drp2': 0.4, 'drp1': 0.33, 'dns_sz': 200}
0.782409 (0.002170) with: {'lstm_sz': 300, 'is_deep': True, 'drp4': 0.33, 'drp3': 0.33, 'drp2': 0.33, 'drp1': 0.33, 'dns_sz': 300}
0.779431 (0.005318) with: {'lstm_sz': 300, 'is_deep': False, 'drp4': 0.4, 'drp3': 0.4, 'drp2': 0.33, 'drp1': 0.4, 'dns_sz': 200}
0.784576 (0.002749) with: {'lstm_sz': 300, 'is_deep': True, 'drp4': 0.4, 'drp3': 0.33, 'drp2': 0.33, 'drp1': 0.4, 'dns_sz': 400}
0.778975 (0.004581) with: {'lstm_sz': 300, 'is_deep': False, 'drp4': 0.4, 'drp3': 0.33, 'drp2': 0.33, 'drp1': 0.33, 'dns_sz': 300}
0.783625 (0.000469) with: {'lstm_sz': 400, 'is_deep': True, 'drp4': 0.4, 'drp3': 0.4, 'drp2': 0.4, 'drp1': 0.33, 'dns_sz': 200}
0.782437 (0.001632) with: {'lstm_sz': 400, 'is_deep': True, 'drp4': 0.33, 'drp3': 0.4, 'drp2': 0.33, 'drp1': 0.33, 'dns_sz': 300}
0.782400 (0.001341) with: {'lstm_sz': 300, 'is_deep': True, 'drp4': 0.33, 'drp3': 0.4, 'drp2': 0.33, 'drp1': 0.4, 'dns_sz': 400}
```