Exercises on text classification with Multi-Layer Perceptrons (MLPs)

Koutsomarkos Alexandros[1], Lakkas Ioannis[2], Tzoumezi Vasiliki[3], Tsoukalelis Nikolaos[4], and Chalkiopoulos Georgios[5]

[1]p3352106 , [2]p3352110 , [3]p3352121 , [4]p3352123 , [5]p3352124
Emails: akoutsomarkos@aueb.gr , ilakkas@aueb.gr , vtzoumezi@aueb.gr , ntsoukalelis@aueb.gr , gchalkiopoulos@aueb.gr

June 19, 2022

Google Colab Link

1 MLP classifier implemented in Keras/TensorFlow

Develop a sentiment classifier for a kind of texts of your choice (e.g., tweets, product or movie reviews). Use an existing sentiment analysis dataset with at least two classes (e.g., positive/negative or positive/negative/neutral).2 The classes should be mutually exclusive, i.e., this is a single-label multi-class classification problem.

1.1 Introduction

Implementing classifiers to make a sentiment analysis on tweets/posts.

1.2 Dataset

For the purpose of the exercise we used the Twitter Sentiment Analysis Dataset. It contains tweets and the corresponding label (0 - negative, 1 - positive). In Figure (1) we can see how these tweets are distributed to each class for the entirety of the dataset. It is worth mentioning that this time we downloaded only the train dataset provided in Kaggle, due to the fact that the Google Colab, even with an enabled GPU, could not handle such a large dataset, especially after the TF-IDF vectorization using 5,000 features.

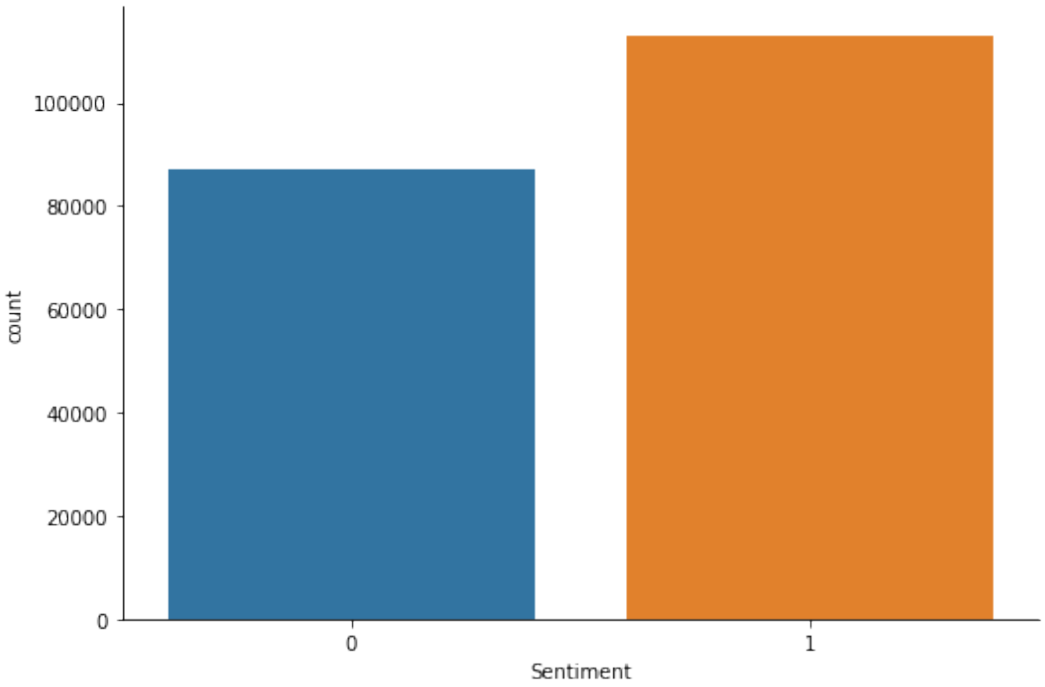

Figure 1: Class distribution for the whole dataset of twits.

1.3 Pre-Processing

Before proceeding with the development of the models, we had to do some data-prepossessing in order to clear the text of the twits and only keep words. The steps we followed may be found below:

1. Lower Case: using the `.lower()` method we converted the input string to the equivalent, only having lower case words. This ensures that there will be no differences in calculations due to differences in lower and capital letters.
2. Clean the text: using regular expressions we removed the following and kept only the words in each tweet.
   (a) underscores
   (b) twitter usernames
   (c) websites
   (d) non words
   (e) single characters

(f) numbers

(g) multiple whitespaces

3. Lemmatization: We split each tweet on whitspaces to get words and then we used the lematize methond from nltk to return the lemma of each word.

4. Finally, the words are put together and returned as a sentence.

In Table (1) we see some samples of the orginal tweets and the corresponding processed tweets.

Original text	Processed text
is so sad for my APL friend.....	is so sad for my apl friend
I missed the New Moon trailer...	missed the new moon trailer
omg its already 7:30 :O	omg it already
.. Omgaga. Im sooo im gunna CRy	omgaga im sooo im gunna cry
i think mi bf is cheating on me!!! T_T	think mi bf is cheating on me tt

Table 1: Sample tweets and the corresponding processed text

1.4 Train/Dev/Test split

In order to create our model we had to split the input into three parts. We set a seed for reproducibility purposes and shuffle the tweets.

1. Train: 70% of the input tweets were used to train the models.

2. Development: 30% of the train tweets were used for feature tuning and hyper-parameters tuning.

3. Test: 30% of the input tweets were used in order to test the performance of the model, using previously unseen data.

1.5 Features

In this exercise we choose to experiment with TF-IDF features and BOW(bag of words).

1.5.1 TF-IDF

We used uni-gram and bi-gram TF-IDF features, which were produced by the TfidfVectorizer. In total we produced 5000 features by excluding any stopword.

1.5.2 Bag of Words

We used uni-gram and bi-gram features, which were produced by the CountVectorizer. In total we produced 5000 features by excluding any stopword.

1.6 Model Training

For this part we chose to follow three main pillars in order to implement our analysis. We used the same dummy classifier and logistic regression models with the previous assignment and then we created various neural network models, starting from the single layer perceptron and then experimenting with more complicated models (e.g. adding layers, neurons and choosing different hyper-parameters).

Dummy Classifier - Logistic Regression - Single Layer Perceptron - Neural Networks

1.6.1 Dummy Classifier

Beginning with the Dummy Classifier we get a "baseline" classification and we can have a first picture. This classifier serves as a simple baseline to compare against other more complex classifiers and this is exactly how we planned to use it in our assignment. In detail we used the 'most frequent' strategy. As a result we made a classifier that assigns the most frequent class of the training data. Finally we printed the train/test accuracy and the confusion matrix (see below).

Train accuracy: 56.46%

Test accuracy: 56.46%

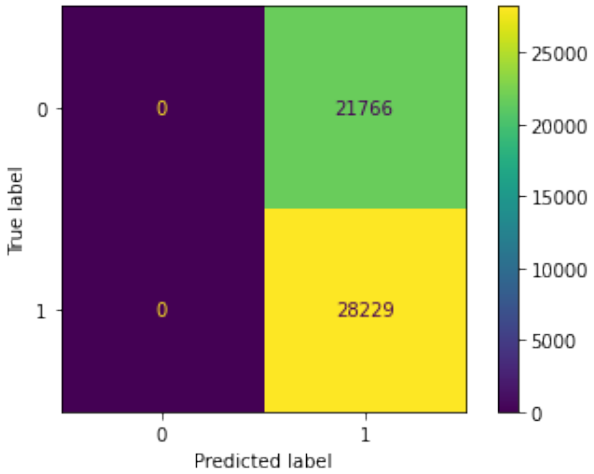

Figure 2: Confusion matrix for the test data using the Dummy Classifier

1.6.2 Logistic Regression

This is the model that was tuned (from our previous exercise, as well) in order to obtain the best possible result.

Consequently we implemented the Randomized Search Cross Validation on these (TF-IDF, Bag of Words) using the development set. Having found our best hyperparameters we implemented our classifier using these hyperparameters on the train and test dataset (fit, predict). Below are the results of the test set, for the TF-IDF and BOW models.

TF-IDF test accuracy: 74.3%

BOW test accuracy: 73.7%

1.6.3 Neural Network Models

Before presenting the various models that we created and how they performed we will explain the functions we created for the model compilation and training.

First of all we created a function ("model_configuration") in order to perform the aforementioned processes and the plotting of the training history and testing scores. In our approach we wanted to check how the different architectures would impact the model's evaluation, keeping the inputs, the optimizer, the metrics and any hyperparameters constant.

Loss Function
During the training of our models we used "binary cross-entropy" as loss function. It compares each of the predicted probabilities to actual class output which can be either 0 or 1. It then calculates the score that penalizes the probabilities based on the distance from the expected value. That means how close or far from the actual value.

Optimizer
Furthermore, for optimization we chose the Adam optimizer, which is different to classical stochastic gradient descent. The latter maintains a single learning rate (termed alpha) for all weight updates and the learning rate does not change during training. However, in Adam optimizer a learning rate is maintained for each network weight (parameter) and separately adapted as learning unfolds. The method computes individual adaptive learning rates for different parameters from estimates of first and

second moments of the gradients. Specifically, the algorithm calculates an exponential moving average of the gradient and the squared gradient, and the parameters beta1 and beta2 control the decay rates of these moving averages.

Metrics
A metric is a function that is used to judge the performance of our model. Metric functions are similar to loss functions, except that the results from evaluating a metric are not used when training the model. Taking into account the assignment instructions we monitored the following metrics during training: precision, recall, F1, precision-recall AUC scores.

Models' architectures
Finally, before proceeding with the models' specific architectures, i.e. the number of hidden layers and neurons per layer, we should mention that all our models are Sequential. This type of model is appropriate for a plain stack of layers where each layer has exactly one input tensor and one output tensor. Also, the layers we choose are Dense, meaning that they are fully connected layers in which every output depends on every input. Regarding the activation functions, for all hidden layers we used the rectified linear activation function or ReLU for short, which is a piecewise linear function that will output the input directly if it is positive, otherwise, it will output zero. For the output layer we used the SoftMax function, in order to return the probability of each class.

1.6.4 Dummy Neural Network

Firstly, we implement a Dummy Neural Network (NN) in order to get a "baseline" classification and we can have a first picture. This NN serves as a simple baseline to compare against other more complex NNs and this is exactly how we planned to use it in our assignment. In detail, a NN with 1 hidden layer and 1 neuron, i.e. a single layer perceptron which is a linear classifier. This is the simplest model we are going to examine. Finally we printed the test accuracy (see below).

Test accuracy: 73.35%

1.6.5 BOW Neural Network

After some implementations and estimations, we came to a conclusion that BOW NN is the best model to apply, for which we tried different "setups" for the NN.

Finally we implemented the BOW NN, for which we chose the following setup:

model with 1 hidden layer and 16 neurons.

For this we ran the predictions and generated the requested metrics.

BOW NN train accuracy: 77 %

	precision	recall	f1-score	AUC
0	0.85	0.59	0.69	
1	0.74	0.92	0.82	
accuracy			0.77	0.753
macro avg	0.80	0.75	0.76	
weighted avg	0.79	0.77	0.77	

Table 2: Score table for the train set.

BOW NN dev accuracy: 73 %

	precision	recall	f1-score	AUC
0	0.77	0.53	0.63	
1	0.71	0.88	0.79	
accuracy			0.73	0.705
macro avg	0.74	0.70	0.71	
weighted avg	0.74	0.73	0.72	

Table 3: Score table for the development set.

BOW NN test accuracy: 73 %

	precision	recall	f1-score	AUC
0	0.78	0.52	0.63	
1	0.71	0.88	0.78	
accuracy			0.73	0.704
macro avg	0.74	0.70	0.71	
weighted avg	0.74	0.73	0.72	

Table 4: Score table for the test set.

Next we produced the curves for the train and development dataset. Apart from the BOW NN curves for the loss on training and development data, we also included f1 and accuracy curves.

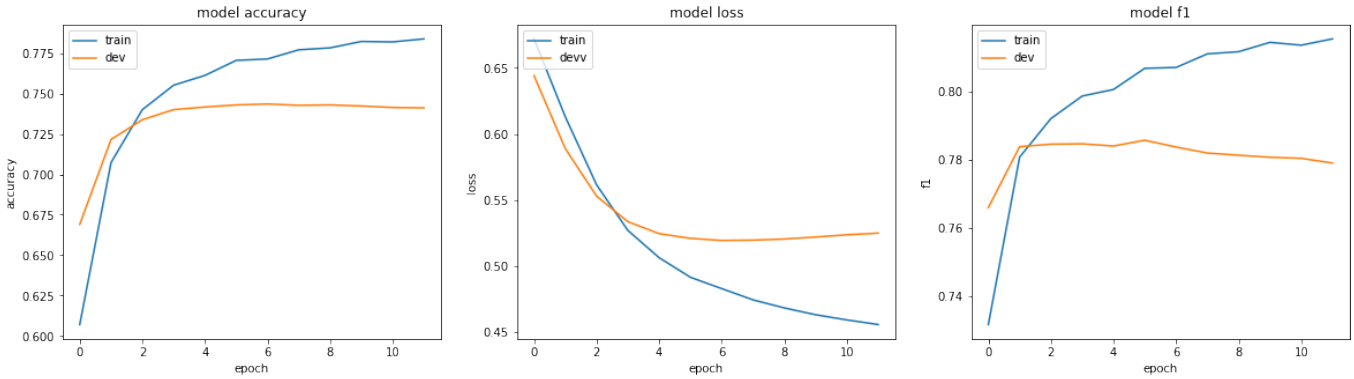

Figure 3: Curves showing the loss on training and development data

Finally we produced the architecture of the best NN model we implemented.

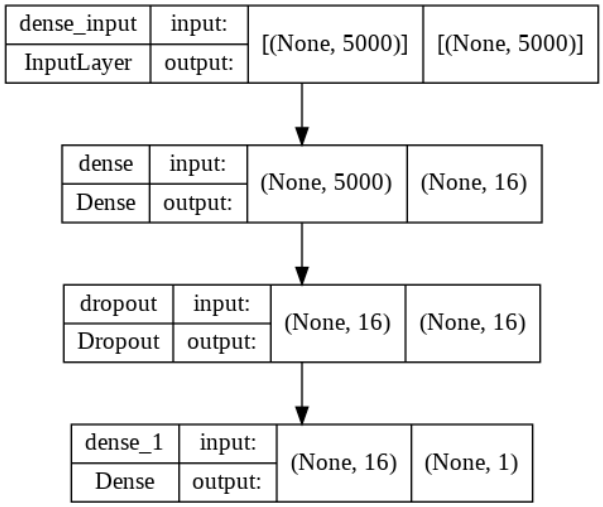

Figure 4: Best NN architecture

1.6.6 Talos - attempt

For our last implementation we used Talos. Talos, is used as an extra way, for estimating and finding the best model. In detail, it assisted us in avoiding, parameter hopping and optimization solutions that add complexity instead of reducing it. Without learning any new syntax, Talos allowed us to configure, perform, and evaluate hyperparameter optimization experiments that yield state-of-the-art results across a wide range of prediction tasks.

Although, we noticed that our implementation for the best model, using BOW NN, still provided better results compared to Talos' results. So, this could be considered as an extra way of figuring out and estimating "a best model", that showed some good results (detailed results can be found in our ipynb file, containing the code and the output).