ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

Text Analytics
Exercise 3 Report

Papastamos Konstantinos – DS3617011

Kanellis Georgios – DS3617006

Tsolas Leonidas– DS3617017

Exercise

The exercise given is described below:

Repeat exercise 17 of Part 2 (Text Classification with mostly linear classifiers), now using an MLP implemented (by you) in Keras, TensorFlow, PyTorch, or DyNet. 3 You are allowed to use a different (e.g., larger) text classification dataset than the one you used in exercise 17 of Part 2. If you use a different dataset, it would be interesting to compare the results of your two classifiers (from exercise 17 of Part 2 and this exercise) on both datasets (e.g., a simpler model may work better when the training set is small). Tune the feature set (if applicable) and hyper-parameters (e.g., number of hidden layers) on a held-out part of the training data or using a cross-validation on the training data. It is better to monitor the performance of the MLP on a separate development subset during training to decide how many epochs to use. Include in your report examples of frequent types of mistakes your classifier makes.

For our implementation we used python, keras and the nltk package. Below all steps taken will be described in detail along with the necessary result screenshots.

The full code used for the exercise can be found here.

1. Data Loading and Preprocessing

First the file of kaggle's imdb reviews dataset was downloaded as found here. The file contains 100000 reviews, of which only 50000 are labeled. After the file was read, the script iterated over the reviews, and used the Clean_Text() function on each individual sentence. The mentioned function does the following:

1. Turns everything to lower case.
2. Removes certain unwanted characters, like punctuation, slashes etc.
3. Reduces all whitespace down to one.

The result of the loading and preprocessing steps is two lists, "data" which is a list of strings containing the sentences of all files and "labels" which is a list containing 1s and 0s indicating positive and negative sentiment respectively.

2. Feature Extraction

On this step, the data features were created. For this project we decided to use TF-IDF features.

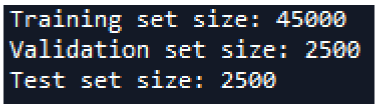
Unigram_vectorizer

The unigram vectorizer created a sparse matrix of Tf-Idf features. We decided to remove stopwords in order to reduce dimensionality and also because we made the assumption that stopwords do not add information on the sentiment of a sentence when working with unigrams (since tf-idf features do not retain the text direction and context information). Finally unigrams with document frequency less than 10 were ignored.

3. Data Shaping and Preprocessing

3.1. Set Formation

The next step was to randomly split the corpus into training, validation and test sets. The splitting was done using sklearn's train_test_split function. A 90 - 5 - 5 split strategy was used to form the training, validation and test sets resulting in the following sizes:

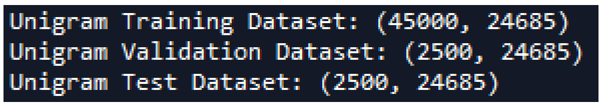


```
Training set size: 45000
Validation set size: 2500
Test set size: 2500
```

3.2. Dimensionality Reduction

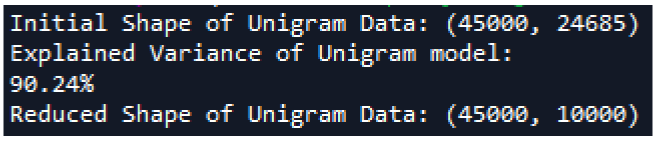
TF-IDF Features

However after inspecting the resulting datasets as seen below, we decided that the number of created Tf-Idf features is too large. In order to tackle the issue, we used truncated SVD to reduce the dimensions as much as possible.



```
Unigram Training Dataset: (45000, 24685)
Unigram Validation Dataset: (2500, 24685)
Unigram Test Dataset: (2500, 24685)
```

We used the TruncatedSVD module from the sklearn.decomposition package and performed SVD on the unigram training, validation and test sets. After several tries, due to resource limitation and by consulting the explained variance ratio for different number of components we ended up keeping 10000 components. The resulting sets have an explained variance ratio as seen below:



```
Initial Shape of Unigram Data: (45000, 24685)
Explained Variance of Unigram model:
90.24%
Reduced Shape of Unigram Data: (45000, 10000)
```

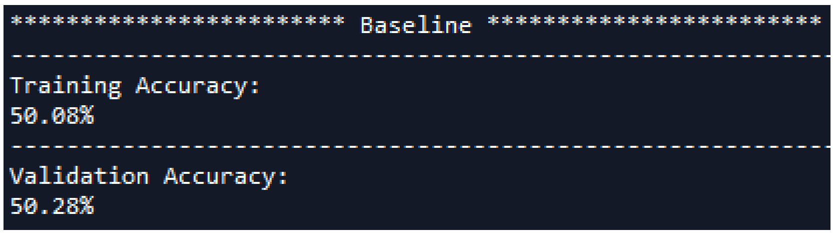
4. Modelling

On the next step we used the keras package on tensorflow to train different Models and evaluate the results. First we created a baseline model to use as a frame of reference.

4.1. Baseline

The baseline-dummy classifier generates always the class 0, since it is the most frequent class on the training set, although the dataset is balanced, so the class frequency is almost the same for both classes.

Thus the baseline scores are seen below:

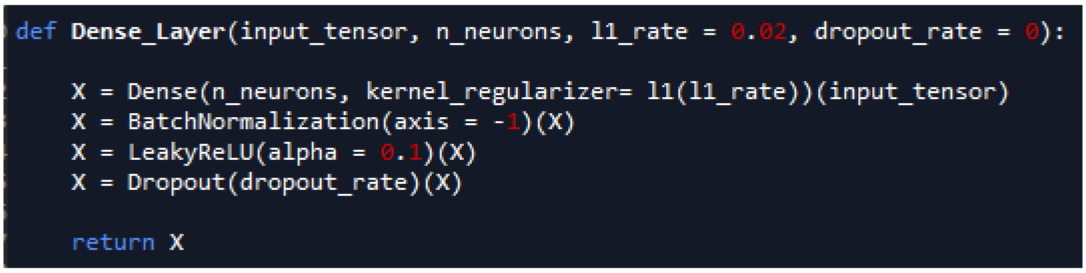


```
************************ Baseline ************************
------------------------------------------------------------
Training Accuracy:
50.08%
------------------------------------------------------------
Validation Accuracy:
50.28%
```

As expected for a balanced dataset, a model that generates always the same class will get almost 50% accuracy.

4.2. Multi Layer Perceptron in Keras

In order to make model development easier, a function was implemented called Dense_Layer() that given an input tensor, the number of neurons for the layer, the l1 regularization rate and the dropout rate it creates a Dense Layer Block with a fixed structure as seen below:



```python
def Dense_Layer(input_tensor, n_neurons, l1_rate = 0.02, dropout_rate = 0):

    X = Dense(n_neurons, kernel_regularizer= l1(l1_rate))(input_tensor)
    X = BatchNormalization(axis = -1)(X)
    X = LeakyReLU(alpha = 0.1)(X)
    X = Dropout(dropout_rate)(X)

    return X
```

The block has the following structure:

- First a dense layer is applied on the input with the given number of neurons and a regularization rate on the kernel.
- Then a batch normalization layer normalizes the output of the dense layer to make training faster and more stable.
- After that, a Leaky Relu activation with a small alpha value is applied on the data.
- Finally a dropout layer is applied in order to regularize the model with the given dropout rate.

Based on this logic, a function is created called Create_Model() that given a list called structure, a l1 rate, a dropout rate, an optimizer and the input size, returns the model to be trained. The structure list contains one item per layer of the model, and each item is the number of neurons for that layer, e.g. if given the list structure = [50,40,30,20] it will create a model with 4 dense layers, the first of which will have 50 neurons, the second will have 40 and so on. After the desired dense layers are created the function adds a batch normalization layer on the output of the final dense layer, and finally one dense layer with the sigmoid function as activation that outputs the model's prediction.

Moreover a class called Metrics was created that implements a callback, in order to evaluate the model on the validation set on the end of each epoch and to also implement the precision, recall and f1_scores which are not provided by keras.

4.3. Training and Tuning

Since there are many hyperparameters to tune, we decided to start with the model structure having the rest of the hyper parameters fixed. Each time we will conclude to a value for a hyper parameter we will tune the next one. So, when we conclude to a certain structure that achieves the best validation accuracy, we will experiment with different optimizers and when the best one has been chosen, the regularization technique and rate will be also tuned. In order to test different structures, we trained 10 different models with the following fixed characteristics:

Optimizer: **Adam** | Loss: **Binary Cross Entropy** | Epochs: **15** | Dropout: **30%** | L1 Regularization: **0.005**

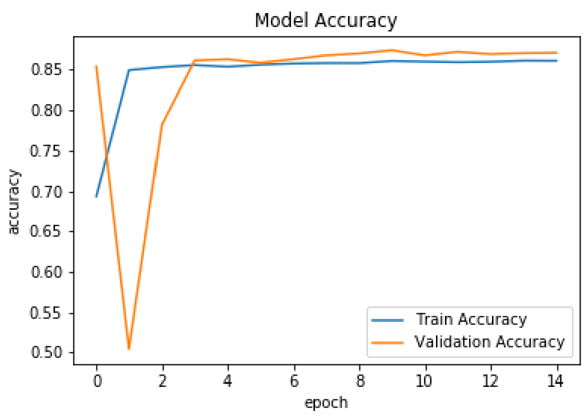
4.3.1. Structure tuning

In order to optimize the model structure we trained 10 different models based on the following scenarios:

- Scenario 1: 1 Hidden Layer with 10 units
- Scenario 2: 1 Hidden Layer with 50 units
- Scenario 3: 2 Hidden Layers with 10 units each.
- Scenario 4: 2 Hidden Layers with 50 units each.
- Scenario 5: 3 Hidden Layers with 10 units each.
- Scenario 6: 3 Hidden Layers with 50 units each.
- Scenario 7: 4 Hidden Layers with 10 units each.
- Scenario 8: 4 Hidden Layers with 50 units each.
- Scenario 9: 5 Hidden Layers with 10 units each.
- Scenario 10: 5 Hidden Layers with 50 units each.

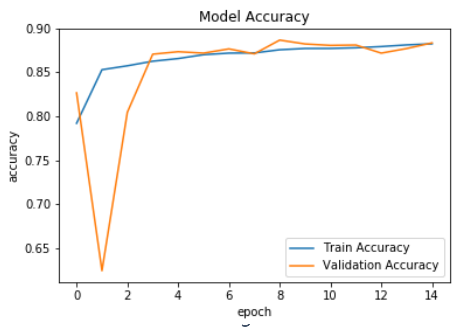
The results can be found plotted in the structure tuning for TF-ID folder **here**.

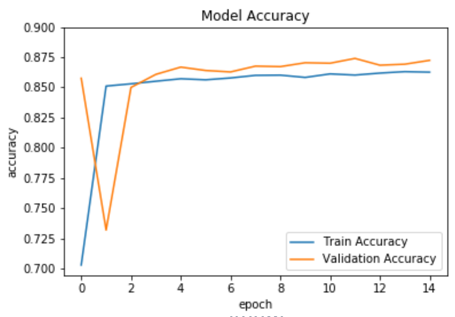
From the results we concluded that all structures achieved almost the same accuracy levels, between 86% and 88%. This can mean that the data are easy enough to be learned, so the extra complexity does not add much on the overall performance of the model. After consulting the results, we chose scenario 10 since it seems to have the most stable training, without many oscillations after the first epoch as seen below:



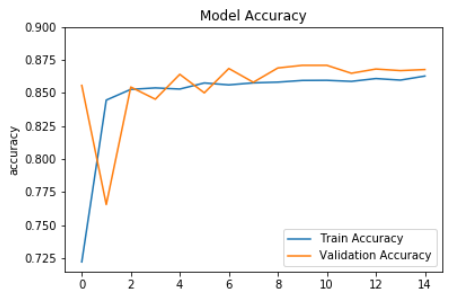
4.3.2. Optimizer tuning

Then we will proceed to try different optimizers. For the purpose of this experiment we will try using adagrad and Rmsprop optimizers in addition to adam already implemented. The rest of the parameters will remain fixed on the aforementioned values. From the results of the experimentation we concluded that the best performance is achieved by adagrad since it has more stable training than rmsprop and at the same time it achieves a constant 87%+ validation accuracy after 4 epochs in contract with adam as seen below:





Adam



Rmsprop

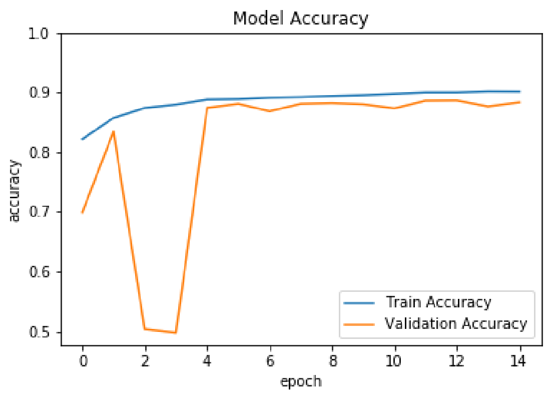
So adagrad will be used for the rest of the tuning. The optimization results can be found in the Optimizer Tuning for TF-IDF folder **here**.

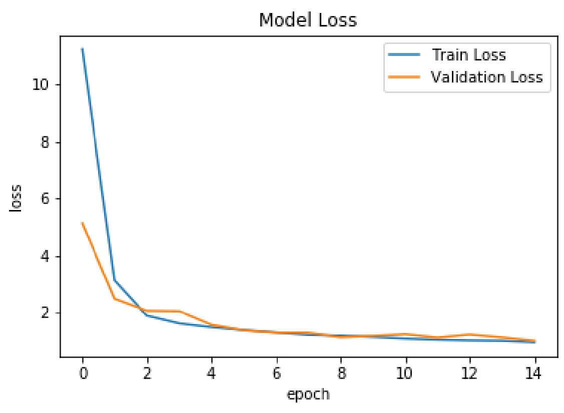
4.3.3. Regularization tuning

First we will try to use only dropout and then decide to add or completely switch to L1 regularization based on the results. The tuning results can be found in the Regularization Tuning for TF-IDF folder **here**.

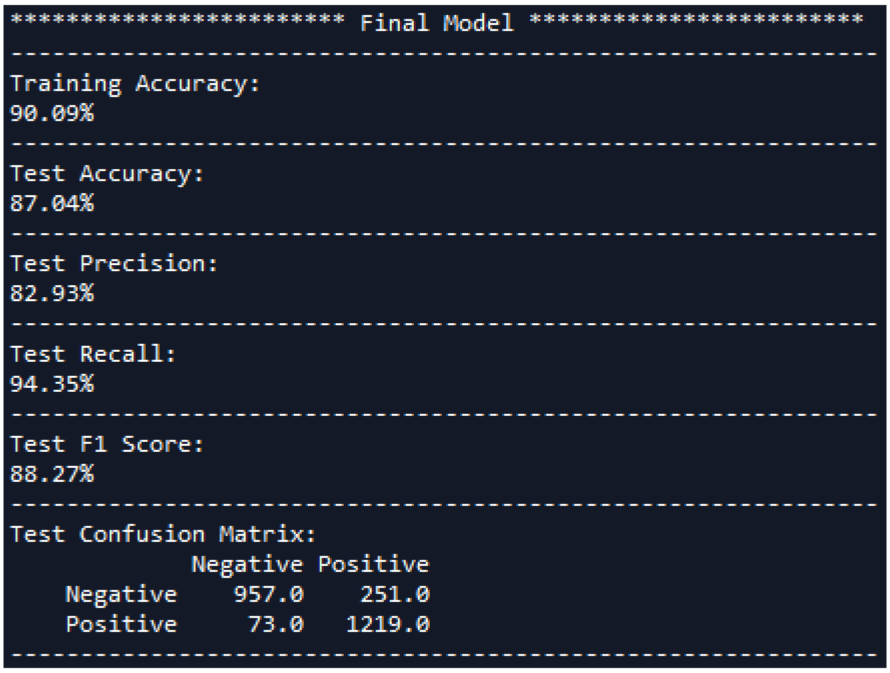
It's clear that dropout does not provide the best balance between high performance and low overfitting. In low levels (<0.6), it improves generalization a bit, but still after a few epochs the model seems to be overfitting, as seen in the 0.3, 0.5 and 0.6 images in the aforementioned folder. Above 0.6, the model can not learn making validation accuracy stay significantly lower than training.

Given these results, we concluded to use only L1 regularization with a value of 0.011, since is proved to keep both training and validation accuracy high after a few epochs with an insignificant difference between the two scores. The training results for this level of regularization can be seen below:





Thus, after tuning, the resulting model will have 5 hidden layers with 50 units each, will use adagrad as its optimizer, will be trained for 15 epochs and will use L1 regularization with a rate of 0.011. The model achieved the following results when evaluated on the test set:



```
************************ Final Model ************************
--------------------------------------------------------------
Training Accuracy:
90.09%
--------------------------------------------------------------
Test Accuracy:
87.04%
--------------------------------------------------------------
Test Precision:
82.93%
--------------------------------------------------------------
Test Recall:
94.35%
--------------------------------------------------------------
Test F1 Score:
88.27%
--------------------------------------------------------------
Test Confusion Matrix:
             Negative Positive
    Negative    957.0    251.0
    Positive     73.0   1219.0
--------------------------------------------------------------
```

4.4. Model Misclassification Overview

In this part we will review some of the errors that our model makes and the possible causes. The overview will be conducted on the test set.

4.4.1. False Positive

First we will review an example of a negative review that was misclassified as positive.

Review text:

"anyone who has a remote interest in science fiction should start at the basics everyone says star wars and star trek are the best science fiction films to begin at which is fine but the truth is the terminator and this movie soylent green are far better choices than those series soylent is probably science fiction ' s best kept secret it remains one of the biggest yet most forgotten films but the impact of its setting is becoming more a reality with each passing day charlton heston overdramatizes his role yet it works edward g robinson in his final role makes the most out of it in soylent green more than anyone else and his final scenes are touching it is manhattan in 2022 the world is overcrowded and food is an unbelievable fortune a small jar of strawberry jam costs $ 150 a big executive for the soylent company is murdered and police detective thorn is on the case the secret of soylent green is not a mystery if you do research on the movie soylent is enjoyable to watch but the whole screenplay is a joke it is just as cheap as the entire production the screenplay and the over dramatics of the actors made the movie yet were completely hilarious everyone seems to be a moron and no one knows the rules specifically cop thorn who likes to just waltz into people ' s apartments peruse around shamelessly and steal anything he wants the character ' s interactions keeps your attention on the movie but still you realize that soylent green sucks an enjoyable piece if you have the time but do not expect anything more"

In this review it's clear that the user does not like the movie. However the use of very positive words (highlighted in blue) confuses the model. The user uses only a few negative words (highlighted in red), which in addition are not inherently bad, but work as negative when used in a certain context. However TF-IDF features are not context-aware so the combination of a high amount of positive words and the lack of context-awareness makes the model misclassify the review as positive.

4.4.2. False Negative

Now we will review an example of a positive review that was misclassified as negative.

Review text:

*"okay i'll say it this movie made me laugh so hard that it hurt this statement may offend some of you who may think that this movie is nothing more than a **waste** of film but the thing that most people don't get is that this movie was intended to be **bad** and **cheezy** i mean did people actually think that a movie about a killer snowman was intended to be a **masterpiece** just look at the scary hologram on the jacket of the movie and you'll find your answer instead like the original jack frost which i thought was just as **funny** this movie turned out to be a side splitting journey into the depths of **corny** dialogue **bad** one liners and **horrible** special effects and it's all made to deliver laughter to us viewers it certainly worked for me for example anne tiler to her troubled husband what makes you **frown** so heavily darling if that chunk of dialogue doesn't make you laugh then you have serious **issues** who in their right mind would utter those words in real life of course no one because it was meant to sound **ridiculous** ! just take one viewing of this movie with an open mind and low expectations and hopefully you'll see what's so damn **funny** about jack frost 2"*

Similarly to the false positive case, in this review the use of mostly negative words is apparent. The user is trying to make the point that the movie has some negative points because it is intended to be that way. Thus in his description he uses a lot of high-emotional negative words (horrible, ridiculous), while in reality when reading the context, it's clear that the actual review expresses a positive sentiment. Since TF-IDF features are not good in detecting contradiction in the sentences, the model understands it as a clearly negative review giving it an output probability of 0.2 .

4.4.3. Conclusion

Even though Tf-Idf features perform good enough for sentiment classification, they fail to capture certain ways of expressing sentiment like contradiction, references to other movies than the target of the review, irony, sarcasm etc. In order to improve, more sophisticated features and model architectures need to be explored, as well as larger training datasets.

4.5. Comparison with Linear Model

There are some interesting points to be made on the differences of this model and the linear classifier implemented on exercise 2. The linear classifier was trained on almost half the data, and achieved a test accuracy of 88% on the Unigram dataset, having reached 89% in the training set. The MLP achieved very similar results with only 1% lower test accuracy. At the same time, even after applying regularization, it's clear that the MLP model is overfitting slightly more compared to the linear model, which is to be expected since MLPs, and especially the architecture chosen here, are significantly more complex than linear classifiers like logistic regression. However it seems that with a dataset of this size, and considering the results of the previous exercise, it would be better to either

- Use a linear classifier, and compromise to an accuracy level of ~88% with a simpler model than use a much more complex model like an MLP to achieve the same results.
- Directly use a more complex model, like an RNN, along with more sophisticated features, like word embeddings, in order to tackle the lack of context awareness issue explained in the previous chapters.