Project Report By:

Introduction

For this Project, we could select amongst two different exercises. We chose exercise 9, a follow up to the previous Project, which gives us a chance to perform a side-by-side comparison of the previously investigated (mostly) linear models and MLPs. The approach was straightforward but that is also what made us avoid focusing on technicalities and achieving a clearer understanding of the material. Having worked with Logistic Regression before, it would now be used as a baseline to evaluate our MLPs. This was a baseline both for building the model, and by a more "academic" approach. Each time a different tool or technique was used in training and evaluating our MLP, it was also inevitably compared with the much more familiar (mostly) Linear Model Classifiers, stressing the similarities and making the differences even more apparent.

Data preprocessing

Even before preprocessing our data, it was already split into training and test subsets, equally large, at 25.000 reviews each. The reviews themselves were also split exactly in half, with 12.500 being positive and 12.500 being negative. Originally the sentiment column had the values "neg" and "pos", which were then replaced with 0s and 1s respectively, since our models can only recognize numbers. Although the dataset was evenly split, we thought that 25.000 test cases would be a bit of an overkill for our models. Thus, we concatenated the train and test data into a single Data Frame which we then split again into 3 subsets. We used to train-test-split twice. First, we kept 70% of our data to train our models on. After that we performed a 60%-40% split on the remaining 30%, the larger part of which we used as test and the smaller as development data.

After the rearranging performed above, the size of our train data is 35000 documents, with test and development data being 9000 and 6000 documents respectively. Furthermore, to get a better overview of our data we then estimated the average document length in our training dataset, which came at 232 words per review. After deciding on the final shape of our data, came data cleaning and preprocessing. To give them their final morphology that would be transformed through the TFIDF vectorizer, we created a preprocessing function which did the following: It lowercases all text, removes html tags through regex, removes punctuation, numbers, single characters and multiple spaces, in that order. Finally, it tokenized the different reviews, used a Lemmatizer on them and reconstructed them giving us our final document form. We ran our preprocessing function on all of our

subsets, and then initialized our vectorizer setting the n-grams hyperparameter to unigrams and bigrams, removing English stop words, as well as setting maximum features to 7.500.

As we wanted to further investigate our data, we initialized another TFIDF vectorizer object, using only unigrams and removing stop words, that we used to calculate our vocabulary size (number of features) which ended up being 78098 words in total. After fit-transforming the vectorizer on our training data, and also using it to transform our development and test data as well, only a few steps remained before we could start testing our models. We created a new Data Frame that only contained the train and development subsets, which would later be used in the grid search function to tune our parameters for the Logistic Regression. Also, we used a mutual info classifier, to further reduce the number of features selected from 7500 to 2500, specifically the ones that present the highest information gain. The number of features kept was selected through several tests of our models' performance on the development subset, with the number of features ranging from 500 to 3750. The tests were not kept in the code, although the results were taken into consideration as we concluded that it would be time inefficient. Finally, we created a function that plots the learning curves for a specific (linear) model given the current state of our datasets. Essentially, given a specific model it breaks the training set into 5 parts, namely 20%, 40%, 60%, 80% and 100% of the whole set. Then, it trains the model on those subsets, testing it on the subset itself as well as the entire test and development sets for each loop. It plots the resulting macro average F1 scores, giving us the 3 learning curves as requested.

Dummy Classifier

In selecting our model, we first needed a baseline. Using a classic majority classifier as a baseline, namely Dummy classifier, we ran the necessary training and testing, as well as plotted its learning curves. Due to the training, test and development sets being balanced amongst the two classes, we got pretty much the exact same macro average F1 scores in all of them at 0.33. The learning curves are a bit more chaotic. Given that this is a majority classifier, this is absolutely logical and caused by the slight difference in balance between the sentiment values in the train, test and development sets. As we can see, the fluctuation is only evident in a very small scale, since they all range from 0.3325 to 0.3350. A majority classifier however is still a solid baseline in any single-label two-class classification problem so we decided it to be an appropriate standard.

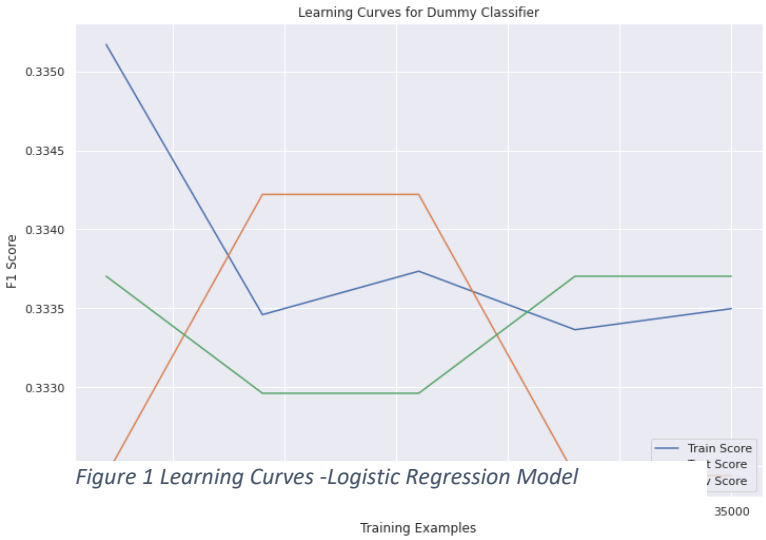

Figure 1 Learning Curves -Logistic Regression Model

Sets/Metrics	Dummy Classifier Model									
	Precision		**Recall**		**F1 score**		Macro Average			
	Positive	Negative	Positive	Negative	Positive	Negative	**Precision**	**Recall**	**F1**	**AUC**
Test	0,5	0	1	0	0,66	0	0,25	0,5	0,33	0,5
Dev	0,5	0	1	0	0,67	0	0,25	0,5	0,33	0,5
Train	0,5	0	1	0	0,67	0	0,25	0,5	0,33	0,5

Table 1 Dummy Classifier Classification Report

Logistic Regression

Up next was the model that we were instructed to use, that being Logistic Regression. Logistic Regression in general procured very encouraging results. Running its grid search on both "C" and "penalty" parameters, we ended up with a C value of 1 and an "l2" penalty as the best approach for our data. We then fit our model as before, returning the relevant metrics as well as learning curves. We had a very stable macro averaged F1 score, ranging from 0.89 ±1% on the validation, train set. The ROC-AUC scores are also quite high, with 0.88 on both test and dev sets and 0.9 on the train set. The behavior of the learning curves is quite orthodox as well, giving us an overall very satisfactory model. It should be noted that the slight drop in the training curve is mathematically consistent. As we add more points to our training set, the logistic function we try to fit to our data moves slightly to a better fit, still "missing" quite a few points. Those slight moves however further generalize the function, giving us a better result in the test and development sets, perfectly explaining the plot. We specifically selected the "saga" solver as it seems to be uniformly the best in most situations, especially in sparse datasets such as ours, since we have used tfidf.

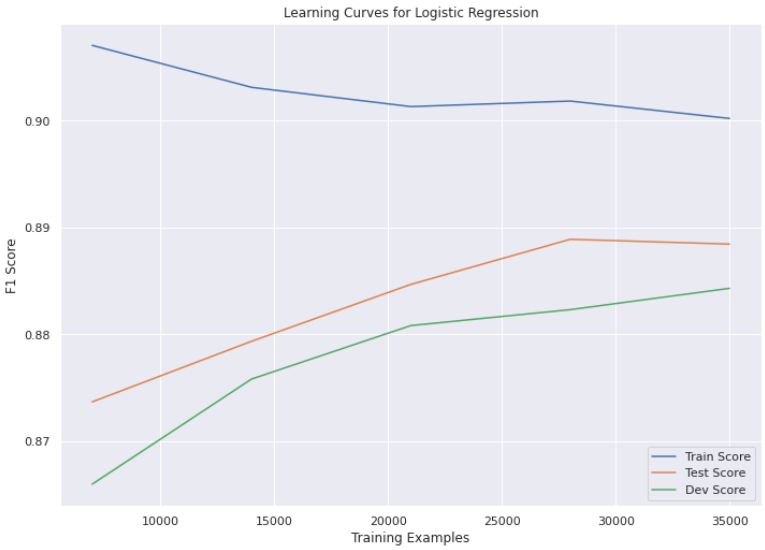

Figure 2 Learning Curves-Logistic Regression Model

Logistic Regression Classifier Model											
Sets/Metrics	**Precision**		**Recall**		**F1 score**		**AUC(macro avg)**		Macro Average		
	Positive	Negative	Positive	Negative	Positive	Negative	Positive	Negative	**Precision**	**Recall**	**F1**
Test	0,87	0,89	0,9	0,87	0,88	0,88	0,94	0,95	0,88	0,88	0,88
Dev	0,87	0,9	0,9	0,87	0,89	0,88	0,95	0,95	0,89	0,89	0,89
Train	0,91	0,93	0,93	0,91	0,92	0,92	0,96	0,96	0,92	0,92	0,92

Table 2 Logistic Regression Classifier- Classification Report

Multi-Layer Perceptron (MLP)

It was also mandated that we have to use an MLP, that being the core learning material of this Project. As in every model, the first thing we needed to do, was to tune all relevant hyperparameters. Given the complexity of a Perceptron Model and all the different hyperparameters it holds, an extensive hyperparameter tuning would be too expensive in computational resources. Because of that, oftentimes the approach selected is not the optimal, but just a computationally cheaper step to the right direction.

For hyperparameter tuning, we selected keras tuner, as it is a relatively convenient tool, specifically created for Neural Network hyperparameters. To do that, we created a "build model" function, that runs a loop over the different hyperparameter choices for each hidden or dropout layer. Of course, the input and output layers are not tweaked at all, as they are set in stone by the formulation of our problem.

Since we know that at least theoretically, a 2-hidden-layer MLP can solve almost any problem of this nature, we had it run from 2 to 4 hidden dense layers, with a dropout in between each one ranging from 0.1 to 0.5. We also had it tune the solver's learning rate, giving it 0.001 and 0.0001 as options to tone down how much RAM the tuner would need. Our input layer was a 2500-dimensional layer, as that is the dimensionality of our tfidf vectors and our output layer was a single node with a sigmoid function as we have a binary mutually exclusive classification. We used Adam as a solver and Relu as the activation function for each hidden layer, even though ideally, we could have also set different options for our tuner to select from. Finally, the tuner would check the validation accuracy each time to select the model.

How the tuner's algorithm works is described perfectly in the documentation, so to quote: "The Hyperband tuning algorithm uses adaptive resource allocation and early-stopping to quickly converge on a high-performing model. This is done using a sports championship style bracket. The algorithm trains a large number of models for a few epochs and carries forward only the top-performing half of models to the next round. Hyperband determines the number of models to train in a bracket by computing 1 + log factor(max_epochs) and rounding it up to the nearest integer." This way the tuning is performed in a much more efficient fashion than a Grid Search would have, which is especially important given the larger number of hyperparameters.

Before we could finally run the tuner to establish our model, we also added 100 epochs with early stopping to our model, monitoring the validation set's loss. We gave early stopping a patience of 2, after manually trying a few different patience values ranging from 1 to 10. Because the model never reached more than 20 epochs, when rerunning the model at a later time, we reset the max number of epochs to 20.

The tuner returned a 2 hidden layer MLP, with the first hidden layer having 224 nodes and the second one having 320. After each hidden layer it decided on a dropout layer with 0.3 dropout probability. Because using the "get_best_hyperparameters" command would require to rerun the tuner every time we wanted to tweak the code, we also manually created a model with the best hyperparameters as selected by the tuner and used that for most of our tests. The tuned MLP returned satisfactory scores overall, however not significantly better than those of the Logistic model. At 0.92, 0.89 and 0.88 weighted F1 scores for the train development and test sets respectively, we can see that the efficiency of our models is very similar.

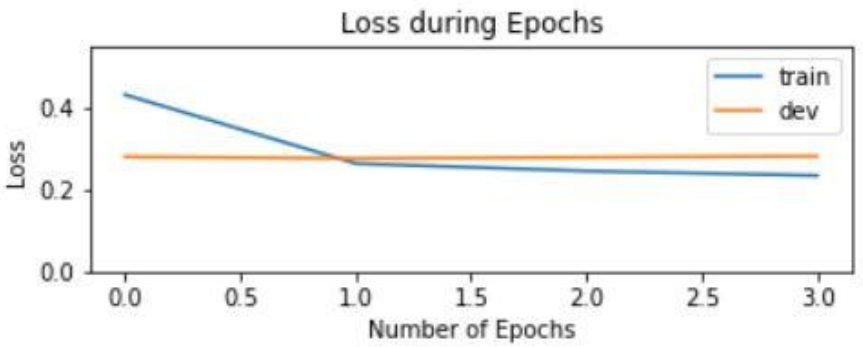

Figure 3 Curves showing the loss on training and def data as a function of epochs

MLP Classifier Model											
Sets/Metrics	**Precision**		**Recall**		**F1 score**		**AUC(macro avg)**		Macro Average		
	Positive	Negative	Positive	Negative	Positive	Negative	Positive	Negative	**Precision**	**Recall**	**F1**
Test	0,87	0,89	0,9	0,87	0,88	0,88	0,95	0,94	0,88	0,88	0,88
Dev	0,87	0,9	0,9	0,87	0,89	0,88	0,95	0,95	0,89	0,89	0,89
Train	0,91	0,93	0,93	0,91	0,92	0,92	0,97	0,97	0,92	0,92	0,92

Figure 4 MLP Classifier Model Classification Repo

Conclusion

After getting to experiment extensively with MLPs, and better understanding the extent that they are customizable and thus can adapt better to any given problem, we reached a final model that had a surprising similarity in results to our Logistic Regression baseline. Though they were both quite accurate and efficient, the fact that such a tailor-made model did not outperform a simple Logistic fit was a bit surprising. This reinforces the belief that a more complex model does not necessarily outperform a less complex one, and those simpler models should always be tried or at least kept as baselines in solving any problem. It is however important to note that due to computational restrictions we did not fully utilize the capacity of our MLP. We only used 2500 features, as we did for Logistic Regression, even though a perceptron could have easily handled the original 7500 features derived from our tfidf transformation, and perhaps even more. Ideally the number of features kept would also be tuned.