Introduction

For our final Project, we could once again select amongst two different exercises. We chose exercise 1, a follow up to the previous Project's exercise 9, which gives us a chance to perform a side-by-side comparison of the previously investigated models and RNNs. This was a much deeper dive into Deep-Learning and its uses for NLP. This was a stepping stone in understanding the complicated nature of Deep Learning. New problems arose, such as the time and computational power limitations that posed few to no issues in our previous projects. As before, we will be using the models created in previous exercises as baselines for our new RNN model with a bidirectional LSTM layer. This time we were familiar with the majority of tools needed, however we would need to be very cautious in their usage under the immense pressure the complexity of our model would create to our limited computational resources.

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. The sheer volume of our dataset, which would normally help our models achieve better results, was actually a cause of concern in using RNNs. Our limited RAM could not handle training our LSTM for so many training examples and that would usually cause issues, such as our RAM crashing, a message we would get used to seeing. This led us to actually keeping only 1.000 of our samples as the total dataset (after extensively trying to keep 20.000 of our data, on which we finally gave up after multiple usage limits by google collab), after which we used to train-test-split twice. First, we kept 70% of our data to train our models on. Then 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 700 documents, with test and development data being 180 and 120 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 162 words per review. Our vocabulary came at 12712 words in total. After deciding on the final shape of our data, came data cleaning and preprocessing. To give them their final morphology, 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.

To tokenize our data, we used Fasttext pre-trained word embeddings. After downloading them, we also created a function to calculate the centroids of each sentence, as all of our models other than the LSTM, have been trained to use a sentence embedding as an input. To use our embeddings, we used a spacy tokenizer to tokenize our sentences, also removing stopwords using one of spacy's built in libraries. 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 0.33-0.34. 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.

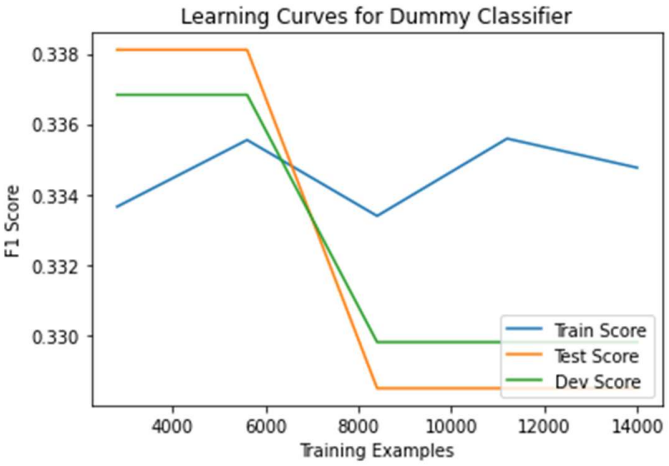


Figure 1 Learning Curves -Logistic Regression Model

Sets/Metrics	Dummy Classifier Model									
	Precision		Recall		F1 score		Macro Average			
	Positive	Negative	Positive	Negative	Positive	Negative	Precision	Recall	F1	AUC
Test	0	0,52	0	1	0	0,68	0,26	0,5	0,34	0,5
Dev	0	0,52	0	1	0	0,68	0,27	0,5	0,34	0,5
Train	0	0,5	0	1	0	0,67	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 0.72 ±2% on the validation, train set. The ROC-AUC were very varied, ranging from 0.82 to 0.86 on the test set and 0.69 to 0.74 on the validation set, with 0.82 to 0.84 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,

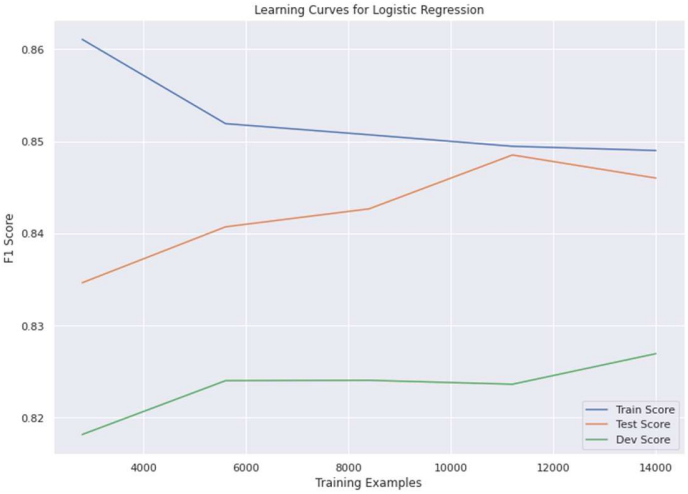


Figure 2 Learning Curves-Logistic Regression Model

perfectly explaining the plot. We specifically selected the "saga" solver as it seems to be uniformly the best in most situations.

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	0,71	0,78	0,79	0,70	0,75	0,74	0,82	0,86	0,75	0,75	0,74
Dev	0,67	0,73	0,74	0,66	0,70	0,69	0,69	0,74	0,70	0,70	0,70
Train	0,74	0,78	0,80	0,71	0,77	0,75	0,84	0,82	0,76	0,76	0,76

Table 2 Logistic Regression Classifier- Classification Report

Multi-Layer Perceptron (MLP)

It was also mandated that we have to use an MLP, this time as a baseline. 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 50-dimensional layer, as that is the dimensionality word sentence centroids 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 50 epochs with early stopping to our model, monitoring the validation set's loss. We gave early stopping a patience of 8 and a batch size of 64, as we have few data for this experiment.

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 0.85 0.83 and 0.84 weighted F1 scores for the train development and test sets respectively, we can see that the efficiency of our models is very similar.

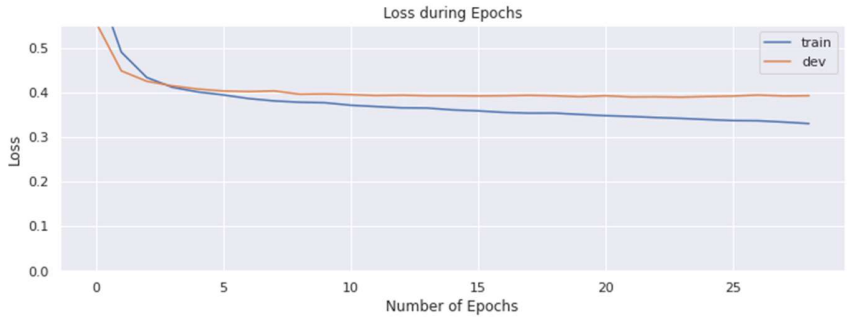


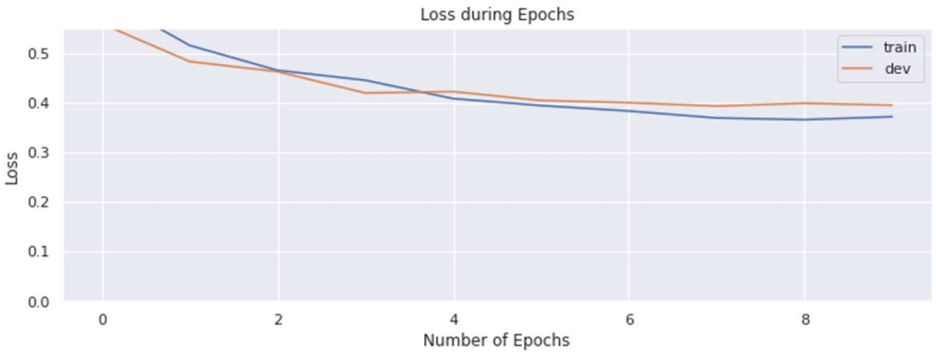
Figure 3 Curves showing the loss on training and def data as a function of epochs

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	0,85	0,75	0,86	0,73	0,80	0,79	0,87	0,88	0,80	0,79	0,79
Dev	0,72	0,74	0,72	0,74	0,72	0,74	0,78	0,85	0,73	0,73	0,73
Train	0,86	0,87	0,86	0,86	0,86	0,87	0,91	0,90	0,86	0,86	0,86

Figure 4 MLP Classifier Model Classification Report

Recurrent Neural Network (RNN)

After using a MLP in the last exercise, we were tasked with solving the same problem, this time however using a bi-directional stacked RNN with LSTM cells and a self-attention MLP using Keras/Tensorflow. Our model of choice was a functional model and consisted of an embedding layer as input. To be precise we used a padded sequence of our documents with a maximum length of 50 words per document, which was a number selected to conserve space in our RAM. The padding process was used as a way of dimensionality reduction to retain a set number of words given a threshold of our choice. In case the document length was less than the desired threshold the document was padded with zeros until the aforementioned limit was met. Following the embedding layer was a bi-directional RNN with LSTM cells, allowing our input to be processed both linearly and backwards and rendering our RNN able to be aware of future context as well. We chose to use 64 as our batch size in order to avoid the high variance in the gradient that could hamper the convergence of the optimizer. On top of the BILSTM was a Deep Attention layer, letting our model selectively focus on valuable parts of the input sequence and hence, learn the association between them, enabling easier learning and a higher quality. The final layer of our model was the output layer, which was responsible for classifying our documents' sentiment. Each respective layer, was followed by a dropout layer because a fully connected layer occupies most of the parameters, and hence, neurons develop co-dependency amongst each other during training which curbs the individual power of each neuron leading to over-fitting of training data. Also, this way we can also avoid vanishing gradients caused by the number of nodes and layers.



RNN 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	0,75	0,79	0,78	0,75	0,76	0,77	0,81	0,83	0,77	0,77	0,77
Dev	0,71	0,76	0,76	0,71	0,73	0,73	0,79	0,80	0,73	0,73	0,73
Train	0,89	0,89	0,89	0,89	0,89	0,89	0,91	0,91	0,89	0,89	0,89

Figure 5 RNN Classifier Model Classification Report

After the complete structure of the model was designed, the next step was tuning the hyperparameters. The first problem that occurred, was how to tune the number of layers in the Deep-Attention MLP. To solve that, we changed the Deep-Attention class, creating an option for an addition of a third layer which, if set to "True", initializes and creates another table "W2" and thus another dense layer in our Deep-Attention MLP. We then selected different options for our different hyperparameters as we did above with Keras tuner, defined a functional model in the required function and finally running the tuner test on our model. The model returned is shown in Figure 6. Our RNN's macro F1 scores were 0.89 for the train set, 0.73 for the development and 0.77 for the test sets. Overall, not the most satisfactory model, but due to data scale limitations, worse results were expected. Pretrained fastext embeddings although might have contributed to the unexpected success of the RNN model.

```
[64]  1 best_model.summary()

Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 50)]              0

 embedding (Embedding)       (None, 50, 300)           150600

 dropout (Dropout)           (None, 50, 300)           0

 bidirectional (Bidirectiona  (None, 50, 200)          320800
 l)

 dropout_1 (Dropout)         (None, 50, 200)           0

 deep_attention (DeepAttenti  [(None, 200),            40401
 on)                          (None, 50, 1)]

 dropout_2 (Dropout)         (None, 200)               0

 dense (Dense)               (None, 1)                 201

=================================================================
Total params: 512,002
Trainable params: 361,402
Non-trainable params: 150,600
_________________________________________________________________
```

Figure 6 Best performing model after tuning

There was also an attempt to optimize the inputs provided to our model using SVD reduction, to keep the best possible features, which unfortunately yielded unsatisfactory results leading us to settle for our original approach.

Conclusion

For this final project, we had to solve the same classification problem, this time, with an even more complicated model than before. After using a simple linear model and a simple MLP, this time we had to use an RNN with a self-attention layer, which was, until recently, considered state-of-the-art in NLP. By steadily escalating the complexity of our models throughout the past few projects, we got to experiment with and get the hang of many different models and tools, from intuitive straightforward run-of-the-mill regressors to Neural Networks that required extensive study of the documentation just to set up. What became obvious after working with all those different models, is that even though we can get satisfactory results with a simpler approach, it really takes a lot of work, setting up a more complex model, in order to achieve that extra percent which could possibly make the difference between a good and the best possible model. It was also prevalent that more complex models really do take up an extraordinary amount of computational power, so a failed attempt at a model could be a lot of time and resources wasted. Those further accents how important tuning the hyper-parameters is, so we get the max possible value for each run. In conclusion, in experimenting with different kinds of models and delving deeper into the concept of deep learning we managed to attain a relative ease in handling all kinds of Neural Networks using Keras and python libraries in general.