coreML'23

Twitter Project Report (Team #58)

Koushik Chowdhury (2572865), Enam Biswas (7015620), Daniel Hof (2568535)

August 29, 2024

Abstract– In this project, we explored three datasets containing training and two test sets. The training dataset has been labeled with both sentiment scores (-1 to 1) and sentiment types (positive, negative, and neutral). Towards getting the best possible result, we have applied our own pre-processing and individual pre-training of transformers. We applied various classification algorithms to the embeddings from the pre-trained model for both regression and classification tasks. For classification, LSTM in combination with MLP layers performed better, on the validation set (10% of the training set) and the test set 1, than the other classifiers discussed in the ML modeling section. This was sufficient to secure #1 position in the leaderboard with an average macro F1 score of 0.8372. Along with the classification algorithm, different regression analyses have been performed on the training dataset to find the most suitable regression algorithm and among them, just feeding the embeddings to a simple dense layer produced the best result (top 2 in the leaderboard for test set 1 with the RMSE score of 0.1781).

1 Introduction and Background

We live in a time where social media plays one of the most important roles in information sharing. People use social media to express their views or opinions on different kinds of things; thus, automatically deriving sentiment is an interesting yet challenging field. As mentioned above, this project involves two tasks: classification and regression analysis. The regression task is to predict the sentiment score, and the classification task is to predict the sentiment type of tweets on the Twitter platform.

Working on a Twitter dataset to predict the sentiment score of a tweet and the sentiment type of a tweet is not new in the machine learning field. There are several studies that have been done before. In this project, we went through some of the state-of-the-art research papers to learn more about the classification and regression approaches. Gautam et al. [3] have done sentiment analysis for customer reviews, which is useful for analyzing data in the form of the number of tweets where opinions are extremely unstructured. Tweets are either positive, negative or somewhere in between. More classical techniques for classifying data include Naive Bayes, Maximum Entropy, Support Vector Machines, and Semantic Analysis (derived from WordNet), where Semantic Analysis (WordNet) provides the highest accuracy [3]. Liu et al. [5] proposed an adaptable multiclass support vector machine to adjust sentiment categorization to various topics in the absence of enough labeled data. Huq et al. [4] also applied support vector machines along with kNN to classify sentiment labels. We also investigated how neural networks, such as convolutional neural networks, have been used for Twitter sentiment classification. In order to train the correct model without requiring further features, Severyn et al. [9] used a convolutional neural network. The BERT model is also introduced in Twitter data. Chanda [1] inspected BERT along with a traditional machine learning classifier, and BERT provided the best result in the prediction task.

In this project, we gathered knowledge and information that can be used to implement the classification and regression tasks from the types of research indicated above. For example Razeen et al. [7] applied different regression models such as linear, support vector, decision tree, ridge, lasso, and random forest to the movie revenue dataset. Razeen et al. [7] found that the random forest regression method provides the lowest RMSE, and Ridge and Lasso provide the same RMSE. We evaluted some of those methods on the task of Twitter sentiment regression along side a variaty of classification models.

2 Datasets and Pre-processing

2.1 Datasets

The datasets were obtained by the Machine Learning team at Saarland University. There are three datasets available, all of which were contributed by them. One is a training dataset, and the other two are test sets. The training set consists of 8000 instances and 16 variables. The training dataset has two target variables for the respective regression and classification tasks, such as 'score_compound' and 'sentiment'. 'score_compound' concerns the sentiment score of the tweets, and the 'sentiment' variable has 3 classes with 'positive' shares 27.85% of the data, 'negative' shares 6.02%, and neutral shares 66.12%. There are no missing values. Both test sets were used to evaluate the model that has been employed in the training set.

2.2 Pre-processing

It is required by the machine learning models for the data to be fed in a structured manner. Thus, the pre-processing of text data is crucial. The dataset is provided with words that were already sufficiently pre-processed. However, due to model tweaking and applying different methods, we have pre-processed our text using 1. We have tried pre-training our model using both the given and our version of pre-processing. Eventually, it seems that our version of the pre-processing was able to pick better contextual information.

Algorithm 1 Text Preprocessing

```
1: function PreprocessText($text$)
2:     $text \leftarrow$ convert $text$ to lowercase
3:     Remove occurrences of "RT"
4:     Remove URLs and mentions (e.g., @username)
5:     Remove hyphens surrounded by letters (e.g., time-to-time $\rightarrow$ time to time)
6:     Remove non-alphabetic characters
7:     Tokenize $text$ into $tokens$
8:     Remove $stop\_words$ from $tokens$
9:     Lemmatize $tokens$ using the lemmatizer
10:    Remove tokens with length $\leq 1$
11:    $processed\_text \leftarrow$ join $tokens$ into a single string
12:    return $processed\_text$
13: end function
```

3 Methodologies

In this section, we outline the architecture and training details of the various regression, classification, and neural network models used in our analysis. We discuss traditional regression models including Linear Regression, Lasso Regression, and Ridge regression, followed by traditional classifiers like Logistic Regression, Support Vector Machines, Random Forests, and Extreme Gradient Boosting. We then delve into neural network models, covering LSTM, Multilayer Perceptron (MLP), and Convolutional Neural Networks (CNN) along with MLP. Lastly, we present our approach for employing DistillBERT, an optimized variant of BERT, for the classification task. Our discussion encompasses model structures, hyperparameters, and training strategies, shedding light on the diverse methodologies employed to achieve the best classification results.

3.1 Experimental Setup

For ML models, we have used the pre-trained embeddings that have been engineered by the last layer of 3.2. To mention, we didn't fully utilize the training dataset since the leaderboard updates were once per week and we always needed some validity before submitting the results. So, we have used 10% of the training set as a constant validation set (with a 'random seed' of 42) while stratifying on the sentiment labels. As a result, all of our models are trained on 7200 examples and validated on 800 examples.

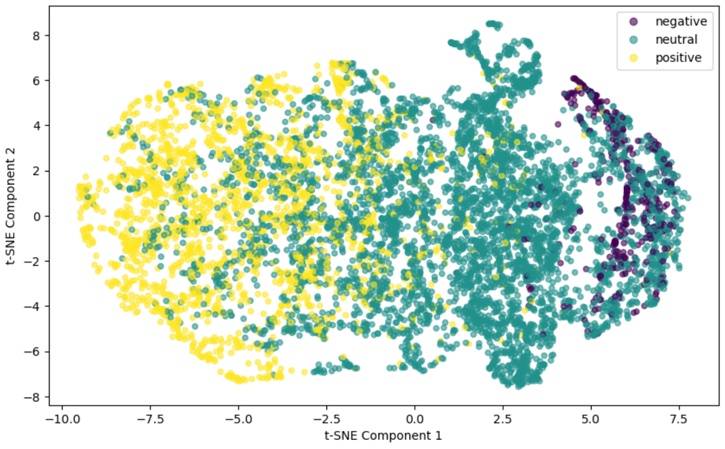
3.2 Transformer-based Pre-training

As already stated in the introduction and as can be seen in the benchmark[1], BERT-based models[2] seem to perform better than more classical machine learning techniques. Yet since the available training data is limited and the base BERT model has many parameters, a smaller model called DistillBERT was chosen. According to the Author DistillBERT is 40% smaller but still retains about 97% of its language understanding capabilities[8]. The decrease in size of the model helps to prevent it from overfitting on our small dataset, by limiting the function class to a smaller set of functions.

For pooling the generated word embeddings into a sentence embedding, we employed attention-based pooling, in which the model uses a dense layer with a softmax activation to learn the attention weights which then get used in weighted sum to reduce the sequence dimension of the word embeddings. In the traditional implementation, this pooling is either done by summing all the word embeddings or using the [CLS]-token as a sentence embedding. In comparison, attention-based pooling allows for the model to specifically learn weights

[1]https://github.com/cardiffnlp/tweeteval

Figure 1: t-SNE plot with PCA



which tells it how important each word is for the regression/classification task. The equation for this can be seen below in which x are the input matrix of dimension BxNxD with B being the batch size N the sequence length and the dimensions of each word embedding and w being the weights of the dense layer. Thus as an output we get a matrix of the shape BxD with the sequence dimension be sum pooled.

$$Sentence\ Embedding(x) = \sum_{i=1}^{N} softmax(w^T x)_i * x_i \tag{1}$$

For each of the two tasks, a separate model was then trained by appending a dense layer to the output of the pooling layer. For the regression task the mean squared error was used as the loss function, while it was found that for the classification task, it helped to also attach a dense layer for regression and jointly learning the the weighted cross entropy loss of the classification and the mean squared error of the regression task. It was found while helping the classification task the joined backpropagation did negatively impact the regression performance, thus only the classification task uses the joined model. The weighting of the cross entropy loss was done in order to mitigate the large class imbalance presented in the dataset. The weights were calculated based on the class distribution, in a way that the loss of each class contributed equally.

Both models were trained by using the AdamW optimizer, which already incorporates a form of regularization by using the technique of weight decay[6]. An additional regularization dropout was employed with a dropout rate of 40%. Due to the outlier sensitivity of mean squared error, it was also found that some of the gradients were exceptionally large and thereby negatively impacted the training by overfitting the model on these samples. To avoid this a technique called gradient clipping was employed, more specifically clipping by the norm of the gradient vector. The value of the clipping was clipped in order to achieve a unit norm.

Lastly, the batch size and learning rate were determined based on multiple cross-validation-based experiments. It was found that a lower batch size of 16 works better than a higher batch size, which is most likely to the outliers contained in the set having less of an impact with a lower batch size and thus more iterations per epoch. As for the learning rate, we found that the learning rate of $5e-5$ works well. For fine-tuning a transformer model this is already a relatively high learning rate so as a future regularization technique to prevent the model from overfitting to the dataset, we employed a linear rate decay with a warm-up period. The warm-up period helps the model not make too drastic initial changes and thus better preserve the valuable knowledge gained in the pre-training of the model. The then following decay of the learning rate can be seen as the reverse process in which we want the model first to learn more general features and then make smaller and smaller changes to find the local minimum. Both of the models were trained for 100 epochs but by employing cross-validation and early stopping both the training processes were stopped around the 50 epoch mark. The output features are of shape 768 that are fed into downstreaming models. To mention, the model to produce embeddings were trained first and then quality of embeddings were than justified based on the performances of downstreaming models.

The Figure 1 (3D space view in A.1) is showing the visualization of embeddings from the last layers of our pre-training process. We can see that the neutral sentiment is so diverse that it's overpowering the negative sentiment by a large margin and positive sentiments are also getting shadowed as well. This is one of the reasons why deriving automatic sentiment is a challenging task.

3.3 Traditional Regression Model

In this section, we have examined various regression models. The reason for applying many regression models to the training dataset is to check how the dataset performed with different model biases, and thus determine the best overall model. The initial regression model is a **Linear Regression** model, that has been applied to the training data with two different settings, such as 'fit_intercept: True or False'. 'fit_intercept: True' indicates that an intercept term would be included in the linear equation, allowing the regression line to start anywhere other than the origin (0, 0). As a result, the correlations between variables in our experiments are better captured. 'fit_intercept: False' tells the opposite and does not include any intercept term in the equation.

The second regression model is the **Lasso regression**. Two parametric settings, such as 'selection: cyclic' and 'selection: random', have been used for both 'fit_intercept: True' and 'fit_intercept: False' where all of the cases, 'alpha' is respectively 0.01, 0.1, 1.0, 10.0. To regularize 'alpha: 0.01' has been used, where 0.01 indicates a mild level of regularization and 10 indicates a high level of regularization. The alpha value is sequentially increased to check the behavior of lasso regression. In our experiments, the selection of 'cyclic' indicates that the coefficients are updated in a sequential manner, whereas 'random' indicates that the coefficients are updated in a random manner for each iteration of optimization.

The same 'alpha' and 'fit_intercept' parametric configurations have also been used in **Ridge Regression**. Eight different methods of solvers have been used to solve the optimization problem associated with the ridge. A solver is needed to find the optimal solution, and each solver uses a different decomposition. The reason for using these multiple solvers is that it aids in determining potential problems that might occur with a certain solver and accessing the stability of the outcomes.

KNN regression has been used by changing the 'n_neighbors (3, 5, 7, 10)' and weights ('uniform', 'distance'). Four kNN algorithm settings, such as auto, ball tree, kd tree, and brute, were employed. Ball tree performs kNN search in high-dimensional spaces, whereas kd-tree is in lower-dimensional spaces. Brute is helpful for tiny datasets and estimates distances between all pairs of data points for kNN search, and if we set 'auto', the algorithm determined the optimum approach based on the data and other factors. Another regression model named AdaBoost regressor used two hyperparameters, such as 'learning_rate' and 'n_estimators'. 'learning_rate' regulates how much each weak learner contributes to the final prediction, while 'n_estimators' is concerned with the model's complexity. A small 'n_estimators' number indicates a simpler model with a lesser danger of overfitting, whereas a big 'n_estimators' value indicates the opposite.

The last traditional regression model is **Decision Tree Regression**, which has four hyperparameters. The first setting concerns the depth of the decision tree, which can be controlled by assigning a depth value. The second setting deals with sample leaf, and the third setting is sample split. The final setting is a splitter, which specifies the approach for selecting the split at each node. While trying out different settings, we ran over 201 different experiments (on top of transformer engineering on the downstream task). We have utilized the scikit-learn libraries for these tasks and selected the model with the lowest RMSE score.

3.4 Traditional Classifers

In order to select many methods, we initially experimented with basic classifiers and their default settings. (unlike trying every parameter as in 3.2). Our initial experimental set was - **Logistic Regression**, **Support Vector Classifier (SVC)**, **Random Forest Classifier**, **k-Nearest Neighbors Classifier**, and **Extreme Gradient Boosting (XGBoost)**. Then we have selected SVC and XGBoost for hyperparameter optimization.

To find the best setup for the **SVC**, we have used the regularization parameter to control the trade-off between maximizing the margin and still generalizing to new unseen data. Additionally, we used a variety of kernels, including linear, polynomial, radial basis, sigmoid, etc. In particular, these kernels help to model linear and more importantly non-linear data connections. We also consider gamma, which is specific to some kernels, such as the RBF and polynomial kernels. It outlines the significance of one training example that delivered unexpectedly good results.

Lastly, we went through XGBoost while exploring 'max_depth' (3, 5, and 7), 'learning_rate' (0.1, 0.01, and 0.001), and 'n_estimator' (100, 200, and 300) to check whether it can outperform SVC. For the implementation of XGBoost, we have used XGBoost along with SKlearn for other implementations.

3.5 Neural Nets for Classification and Regression

For the regression task, we have experimented with feeding the embeddings to multiple MLP models. We tried several combinations of neurons (512, 256, 128, and 64), while introducing different dropout rates (0.1, 0.2, 0.3, and 0.4) and various optimizers (SGD, SGD with Momentum, Adam, and RMSPro). We observed that, the simpler MLP models are, the better they performed. The reason behind this is clear from the observation of

BERT pre-training. We have trained our models up to 20 epochs and all results are presenting the best RMSE value while applying early-stopping with a patience of 2.

In the case of the classification task, we have fed the embeddings to various combinations of deep neural networks. Similar to regression tasks, we have fed the embeddings MLP models (up to 4 layers), first. These seem be lacking compared to our other models (however, they were still performing better than the traditional models, except for SVC). To improve it even further, we have introduced 1D convolutional layers (up to 2 layers, with filter sizes of 256 and 128 and a fixed kernel size of 3) on top of one dense layer and then the predictor dense layer. This proved to be better then most of the larger MLP models (more about it in the result section later). Our best model, an LSTM-based model (2), was found by experimenting with several main components (of size 128 and 256) but a fixed neuron size of 64 for the dense layers.

Algorithm 2 Classification Model with LSTM

```
1: procedure ClassificationModel($input\_size, hidden\_size, num\_classes, use\_dropout$)
2:     $lstm \leftarrow$ LSTM($input\_size, hidden\_size$, batch_first=True)
3:     $fc1 \leftarrow$ Linear($hidden\_size$, 64)
4:     $dropout \leftarrow$ Dropout(0.5)
5:     $fc2 \leftarrow$ Linear(64, $num\_classes$)
6:     self.hidden_size $\leftarrow$ $hidden\_size$
7:     self.use_dropout $\leftarrow$ $use\_dropout$
8: end procedure
```

We explored the above architectures with different number of epochs (up to 20) and dropout rates (0.1, 0.2, 0.3, 0.4, and 0.5). In most of our experiments, we managed to limit overfitting. Adam optimizer worked the best for us with a learning rate of 0.01 (although we have experimented with lower learning rates, this seemed to have worked very nicely). All these models were implemented using PyTorch along with the model in 3.2.

4 Model Selection and Results

All our results are based on the validation set in 3.1. For the regression task, we have tried to minimize the RMSE score and for classification, our main concern was to maximize the average macro F1 score. The validation scores almost represent the results in the test set 1. For example - the best performance of our regression model is 0.1557 while getting 0.1781 REG-leaderboard and best AMF1 was 0.87 while getting 0.8372 CLF-leaderboard on the test set 1.

4.1 Regression results

From Table 1, after intensive hyperparameter searches, we were able to see the performance of our best regression models. There is no significant difference in RMSE scores between the Linear Regression and Ridge Regression models, indicating that both are consistent in their ability to predict unknown data. These models have been overfitted, based on the low train RMSE values. The validation RMSE for Lasso Regression is 0.183, suggesting a less accurate fit. It indicates that there may be overfitting and suboptimal feature selection due to the higher train root mean square error (0.074). Similar to linear models, the K-Nearest Neighbors Regressor has a validation RMSE of 0.16. In contrast, its train RMSE of 0.0 indicates significant overfitting that would require further regularization or hyperparameter adjustment.

Decision Tree models performed similar to the linear models with a validation RMSE of 0.167, but with a bit more variation. Due to its low train RMSE (0.0125), it may be overfitting, thus requiring tree depth control. A competitive validation RMS error of 0.162 indicates stable performance for the Adaboost Regressor.Its train RMSE (0.0275) implies that it has a good degree of generalization, but there may be space for improvement.

When looking at the train RMSE score, our worst performing model, the MLP models, has a high validation RMSE of 0.2712, indicating potential overfitting. The extraordinarily low train RMSE (0.002) suggests memorizing. The single dense layer model, our best model, has the lowest validation RMSE of 0.1557, demonstrating good predictive power. Its train RMSE (0.0330) indicates good generalization, emphasizing the utility DistillBERT for feature extraction.

4.2 Classification Results

For the classification task (Table 2), the traditional machine learning models didn't improve significantly while shortlisting them for the best hyperparameters search. The gain in average macro F1 (AMF1) score is low compared to the default parameters. One possible reason might be that the embeddings were already enriched to the point that only the mechanism behind the models contributed towards the end result. SVC performed

Model	**Best Parameters**	**Val. RMSE**	**Train RMSE**
Linear Regression	'fit_intercept': False	0.1602	0.0187
Ridge Regression	'alpha': 0.1, 'fit_intercept': False, 'solver': 'saga'	0.1602	0.0190
Lasso Regression	'alpha': 0.01, 'fit_intercept': True, 'selection': 'cyclic'	0.1830	0.0735
K Neighbors Regressor	'algorithm': 'ball_tree', 'n_neighbors': 10, 'weights': 'distance'	0.1605	0.0
Decision Tree Regressor	'max_depth': 10, 'min_samples_leaf': 2, 'min_samples_split': 10, 'splitter': 'best'	0.1665	0.0125
Adaboost Regressor	'learning_rate': 1.0, 'n_estimators': 250	0.1620	0.0275
MLP (multiple layers)	dense = [128, 64], LR = 0.01, optimizer = Adam, dropout = 0.5	0.2712	0.002
Single dense layer	LR = 0.0005, optimizer = AdamW, dropout = 0.4	0.1557	0.0330

Table 1: Regression Report

Model	**Best Parameters**	**Val. AMF1**	**Train AMF1**
Logistic Regression	Default	0.80	0.89
SVC	'C': 10, 'gamma': 'scale', 'kernel': 'rbf'	0.82	0.98
Random Forests	Default	0.79	0.89
KNN	Default	0.77	0.97
XGBoost	'max_depth': 10, 'min_samples_leaf': 2, 'min_samples_split': 10, 'splitter': 'best'	0.77	1.00
MLP	dense = [256, 128], LR = 0.001, optimizer = Adam, dropout = 0.3	0.66	0.92
CNN + MLP	CNN = [(128, 3), (128, 3)], dense = [64], LR = 0.001, optimizer = Adam, dropout = 0.2	0.78	0.87
LSTM + MLP *****	LSTM = 128, dense = [64], LR = 0.01, optimizer = Adam, dropout = 0.5	0.87	0.88
Single dense layer	LR =0.0005, optimizer = AdamW, dropout = 0.4	0.84	0.98

Table 2: Classification Report

extremely well on our validation set with an AMF1 of 0.82, however, it overfitted greatly on the training set. KNN and XGBoost performed similarly on the validation set, however, both the models overfitted largely on the training set. Since our features are of very high dimensions, this kind of pattern is commonly observed.

On the other hand, the hyperparameters chosen for the SVC likely perform better due to a moderate regularization level (10), the adaptively controlled influence of data points (gamma set to 'scale'), and the use of a non-linear kernel ('rbf'). This combination helps handle varying feature variations, and captures complex patterns, thereby contributing to improving the AMF1 score compared to other traditional models.

While exploring MLP models, we have noticed that the simpler the MLP models are the better. However, while experimenting on various combinations and several hyperparameters (after almost several hundred runs) we were able to get the best AMF1 of 0.66, but resulted in a very overfitted model in most of the cases. It is due to the overfitting vulnerability of MLP model and the model eventually forgetting important contextual information. CNNs alongside MLPs performed nicely, and were less prone to overfitting compared to traditional and MLP models. However, our best model the LSTM model alongside a simple MLP layer with 64 neurons outperformed other models. The most impressive thing about it is the balance between train AMF1 and validation AMF1 score. It almost performed similarly on train and validation set. We have tried running it on several epochs, even after reaching the optimal epoch, but the model overfitted. Thus we have applied an early stopping patience of 3. The classification for this model is present on A.3 and also the learning curve in A.2.

5 Conclusion and Further Work

Our best models performed well based on the last leaderboard update (regression RMSE of 0.1781 on 14.08.2023 and classification average macro F1 score of 0.8372 on 16.08.2023), considering we only used 90% data for training. Although we have managed to gain impressive results on such a small dataset using custom pre-trained transformers, we could benefit from using more data. However, we plan to explore even further while tweaking the larger pre-trained model (citizenlab), which was trained on millions of sentiment data. Also, we intent further experiment with the usage of word embeddings and see what improvements could be gained. Theoretically, these changes should lead to significant performance gains, due to further enriching the extracted contextual information available to the predictors.

References

[1] A. K. Chanda. Efficacy of bert embeddings on predicting disaster from twitter datas. *arXiv preprint*, arXiv:2108.10698, 2021.

[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. pages 4171–4186, Minneapolis, Minnesota, June 2019.

[3] G. Gautam and D. Yadav. Sentiment analysis of twitter data using machine learning approaches and semantic analysis. *Seventh international conference on contemporary computing (IC3). IEEE, 2014*, 2014.

[4] A. A. Huq, Mohammad Rezwanul and A. Rahman. Sentiment analysis on twitter data using knn and svm. *International Journal of Advanced Computer Science and Applications*, 8.6, 2017.

[5] L. F. L. F. C. X. Liu, S. and H. Shen. Adaptive co-training svm for sentiment classification on tweets. *In Proceedings of the 22nd ACM international conference on Information Knowledge Management*, (pp. 2079-2088), 2013, October.

[6] I. Loshchilov and F. Hutter. Fixing weight decay regularization in adam. *CoRR*, abs/1711.05101, 2017.

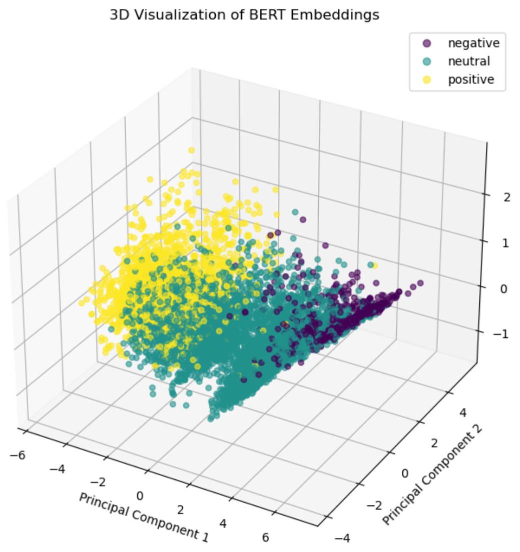
[7] S. S. B. W. Razeen, F. and S. Magesh. Predicting movie success using regression techniques. *In Intelligent Computing and Applications: Proceedings of ICICA 2019*, (pp. 657-670). Springer Singapore, 2021.

[8] V. Sanh, L. Debut, J. Chaumond, and T. Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter, 2020.

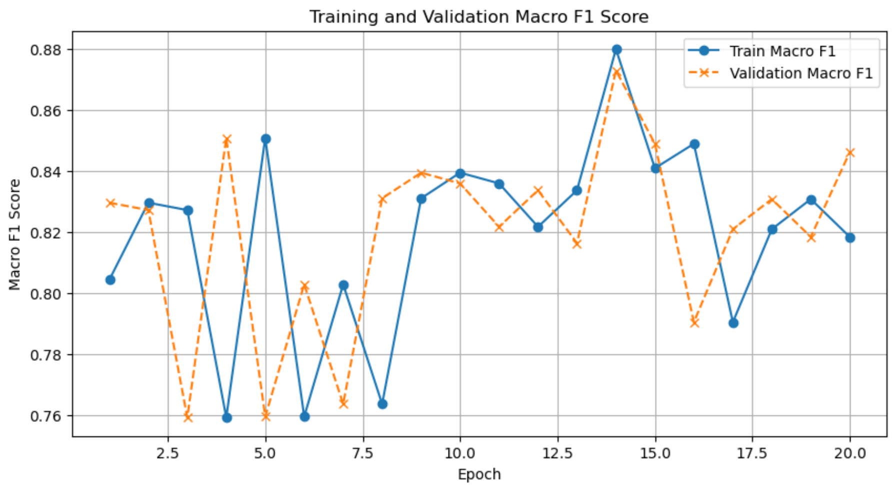
[9] A. Severyn and A. Moschitti. Unitn: Training deep convolutional neural network for twitter sentiment classification. *Proceedings of the 9th international workshop on semantic evaluation*, 2015.

A APPENDIX

A.1 3D t-SNE plot on Embeddings



A.2 LSTM + MLP Learning Curve



A.3 LSTM + MLP Classification Report

The F1 score on each class is considerably impressive (as we have a very imbalanced dataset). We managed to extract proper embedding space in differentiating classes.

Class	**Precision**	**Recall**	**F1-Score**	**Support**
Neutral	0.84	0.77	0.80	48
Positive	0.92	0.95	0.94	529
Negative	0.92	0.84	0.88	223
Accuracy			0.91	800
Macro Avg	0.89	0.86	0.87	800
Weighted Avg	0.91	0.91	0.91	800

Table 3: Classification Report