UNIVERSITY OF ATHENS
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

Deep Learning for NLP

Student name: *Apostolos Koukouvinis*
sdi: 2000098

Course: *Artificial Intelligence II (M138, M226, M262, M325)*
Semester: *Fall Semester 2023*

Contents

1. Abstract

The task is to create a sentiment classifier (NEUTRAL, NEGATIVE, POSITIVE) for greek tweets using a bidirectional stacked RNN with LSTM/GRU cells. The inputs will be wor2vec embedding that properly represent each tweet. To tackle this we have to pre-process the data, create a Data Loader and a Neural Network, encode the Sentiments using Label Encoder, create a Word2vec model based on our dataset, represent the tweet based on word embeddings from Word2Vec, train the model and predict the test set Sentiments. To do that, we have dataset and validation set, based on these we will try to tune the hyperparameters using optuna framework to improve f1 score. We have chosen to improve f1 score since after experimenting a lot we found out that the most consistent score were that improve f1. This is normal since it uses both precision and recall to calculate its value. After we find the best model and the best hyperparameters, we will concatenate the validation and dataset to create the expanded word2vec model and predict the sentiments of the test set.

2. Data processing and analysis

2.1. Pre-processing

Basically the pre-process is simillar to the pre-process of the first and second assignment without removing special characters, the results we got with them included were better. First of all we lowercase all letters and remove stopwords (we have created a stopword list based on greek frequent words). Then we remove urls. Afterwards we remove acute tones and replace ς with σ and imply stemming using greek stemmer. We also have to add a space between words and special characters like @ and # to treat them as seperate words-tokens. Finally we remove extra spaces and save our clean dataset. We have to mention here that all the steps above were done for both data and validaiton set, and of course test set.

2.2. Analysis

Since the dataset is the same of the first assignment, the analysis is also the same. After analysis on the words frequency, we could not find any relationship between the 20 most frequent words and any sentiment, so we can drop these words or just dont mind them. It is also interesting, that after the data cleaning, the top 3 most frequent words per sentiment are common. So the dataset is prety balanced in terms of word frequency

2.3. Data partitioning for train, test and validation

We are keeping the original partition in train and validation set so the optuna studies are more simple.

2.4. Vectorization

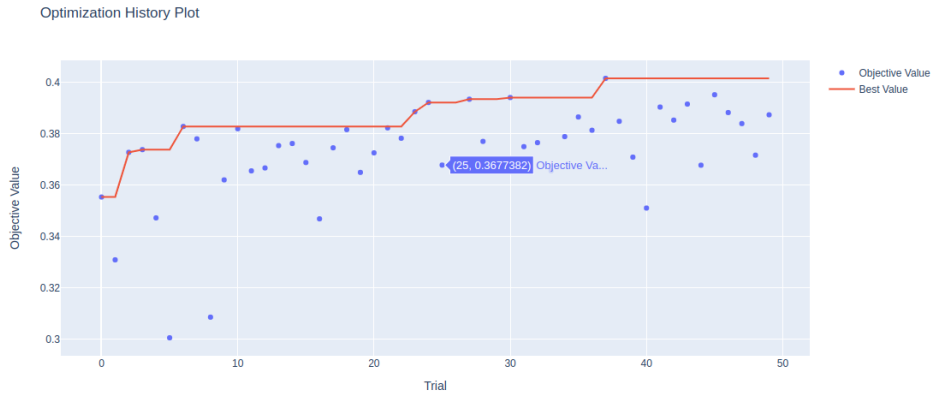
In this project the vectorization may be useful to represent each tweet. I tried Tfidf

and count vectorizers in the second project, and tried to tuned hyperparameters for them using optuna. However those representation did not yield results as good as the naive calculation of mean embedding. So in this project I also tried tfidf which was more competitive than count vectorizer but still the results with mean embedding are just better

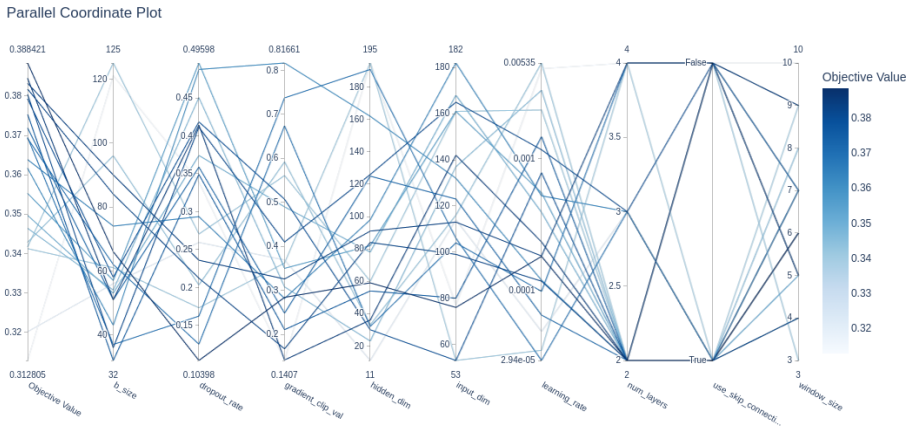
3. Algorithms and Experiments

3.1. Experiments

Fisrt of all we need to decide the representation of each tweet which will be the input of the Neural Network. The first main approach is to represent each tweet as a sequence of word embeddings, with each word representing a timestep. The second approach is to represent each tweet bu getting the mean its word embeddings. The first two optuna studies are used in order to come up with the best representation. After some reruns of the project, it is clear that the mean embedding is the best approach, with more consistently better scores, so we are using this approach. The experiment start with the optuna trial which is using mean embedding representation and LSTM cells for the RNN. For the Sentiment Classifer class we have we are using input_dim, hidden_dim, output_dim, num_layers, dropout_rate, use_skip_connections parameters for the initialization of the classifier/RNN. Where :
input_dim: The size of the input features. (size of embedding)
hidden_dim: The number of features in the hidden state of the RNN.
output_dim: The size of the output, the number of classes in the classification task (always 3 in our project).
num_layers: The number of recurrent layers in the RNN.
dropout_rate: The dropout rate, used to regularize the network.
use_skip_connections: A boolean that indicates whether or not to use skip connections .
In the code the self.fc line creates a fully connected (linear) layer that maps the output of the LSTM layers to the output size. The hidden_dim * 2 is used because the LSTM is bidirectional, so the output features from both directions are concatenated, doubling the feature size. The experments consist of LSTM and GRU with mean embedding, LSTM and GRU with tfidf embedding. After these experiments we keep the best one which is LSTM with mean embedding and try to properly use the attention method, fist we a simple class and then with a more complex one. We use 4 differrent plots to visualize each optuna study. The first one is the history plot which does not give much information, just shows us how fast the study found the best solution and it is like this for LSTM with mean embedding :

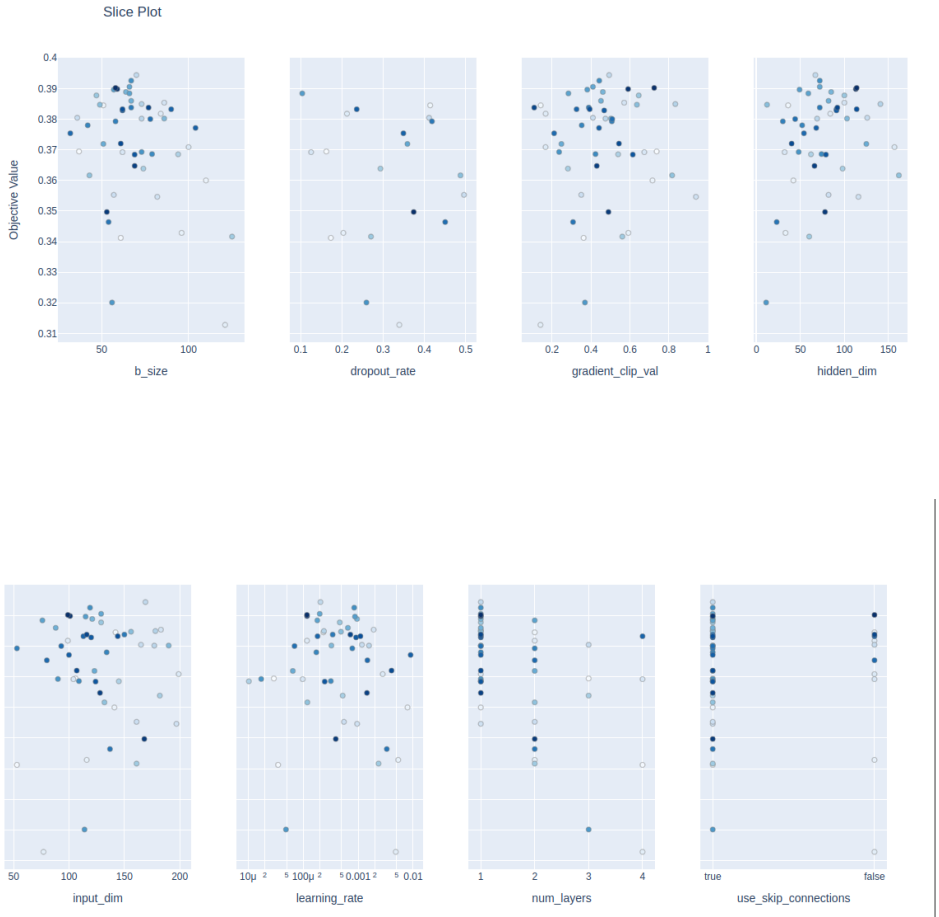


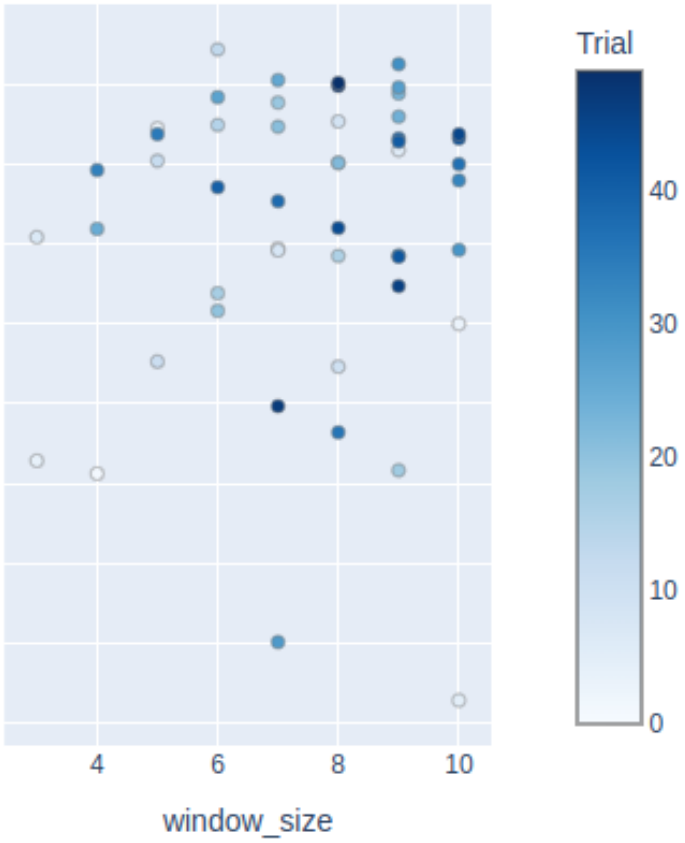
From this plot we can deduct that more trials means better objective value since the optuna study converges to the optimal. This can be said for all optuna studies of the project so it will not be commented any further.
The second image for some reason it can not be properly diplayed in Kaggle and needs to be downloaded as png, this is what we get for the LSTM with mean embedding :



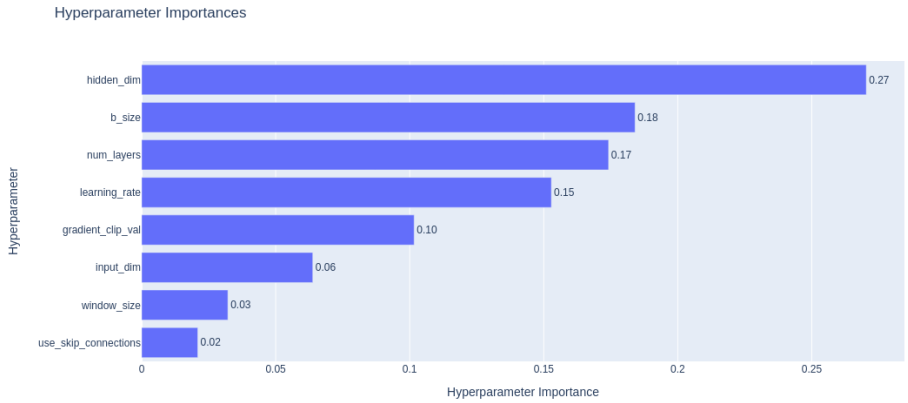
The rightmost axis represents the objective value, with a range from approximately 0.312 to 0.388. The darker the blue, the better the performance based on the objective value. To find the parameters that yield the best results, locate the line that corresponds to the darkest blue on the objective value axis. In this plot, the darkest blue line seems to hover around the 0.38 mark on the objective value scale. The plot shows a wide range of values for each hyperparameter, indicating that the optimization process explored a diverse set of configurations. It is interesting to note that the learning rate seems to have a wide range of values associated with higher performance (darker blues), which suggests that the optimal learning rate may be dependent on the combination with other hyperparameters. Similarly, the number of layers varies between 2 and 4, indicating that both shallow and deeper networks were among the top performers. The use_skip_connections parameter is binary (True/False), and lines corresponding to both values reach into the higher performance range. This means that skip connections may not be a deciding factor in performance for this specific optimization or that its effect is context-dependent on other hyperparameters. The batch_size appears to have a range from around 32 to 125. The darker lines, which indicate better performance, do not show a strong preference for a specific batch size. This could mean that the model's performance is relatively insensitive to batch size within this range,

or that the optimal batch size depends on the combination with other hyperparameters. The hidden_dim shows a wide range from around 11 to 195. The performance seems to be spread across this range, indicating that both smaller and larger hidden dimensions can result in good performance, depending on the other hyperparameter settings. The input_dim has values stretching roughly from 53 to 182. Similar to hidden_dim, there isn't a clear trend indicating a preference for performance, suggesting the model's performance is robust to a range of input dimensions, or the optimal input dimension is contingent on other factors. The window_size ranges from 3 to 10. The plot does not clearly show a strong preference for a particular window size, indicating that performance may not be highly sensitive to this parameter within the explored range.
The next plot we are displaying is the slice Plot, this is an interseting plot that helps to identify which hyperparameters have a significant impact on the model's performance and get an insight of the optimal value ranges. This is the plot for the first experiment :





3.1.1. Table of trials. These plots are hard to display in the pdf, however there is a lot information in it. We can see that for batch size the best sizes are around 60-70, dropout rates are evenly distributed, gradient clip values are better between 0.4-0.6 and hidden dim is better between 50-100. Input dim is evenly distributed an learing rate as well. Number of layers yield the best result with 1 or 2 values, while the use of skip connection seems irrelevant to the objective value and so does window size.
However the most straightforward way to get an insight of the importance of each hyperparameter is the last plot which is parameter importance plot and for the first experiment this is the plot :



I dont think that this plot needs explanation, hidden dim is by far the most important hyperparameter while use skip connections, window size and input dim seems irrelevant. All the others hyperparameters are kinda relevant to the objecti value.
So there are all the plot we are going to use. All plots are in the kaggle project, we will not display them here again because it will make the pdf too exhausting, however we will comment the results.
As it comes for the GRU cells with mean embedding the hidden dim and number of layers are kinda irrelevant while learning rate and window size are by far the most important hyperparameters. There is no clear explanation to this phenomena and may be just random or due to the dataset. A possible explanation is that generally since GRUs are simpler with fewer parameters, the learning rate might become more critical due to their simpler structure. The model's ability to converge and find optimal solutions might be more sensitive to the rate at which it learns. In contrast, with LSTMs, the size of the hidden layer (hidden_dim) can be more influential. This is because LSTMs have a more complex architecture, and the capacity to capture and retain information over longer sequences is more dependent on the size of the hidden layers. The hidden dimension size directly impacts the LSTM's ability to model complex features and interactions in the data. A larger hidden layer size can capture more nuanced patterns, which is crucial in tasks like sentiment analysis. For the GRU the objective function is better when batch size is at 70-100, gradient clip 0.2-0.3 and all others remain the same with LSTM. There is also an experiment with naive RNN and mean embedding, is it interesting that the objective function for that model is not that bad, even when we are not dealing with the vanishing gradient problem. For LSTM and GRU we will also try tfidf vectorization to represent each tweet instead of mean embedding, howver the results are not that good. For the tfidf with LSTM cells the most important parameter is gradient clipping. This happens perhaps because when training with high-dimensional, sparse data like TF-IDF, the LSTM may experience more significant fluctuations in gradients, making training unstable. Gradient clipping helps in stabilizing the training process by limiting the gradients to a defined range. This prevents the model parameters from making too large updates, which can cause the model to diverge. The importance of gradient clipping in this scenario underscores the challenges of training LSTMs with non-sequential, high-dimensional data. It becomes crucial to manage the training stability and ensure that the model learns effectively. For the TF-IDF with GRU cells the learning rate remains the most important parameter. Overall, we can say that hidden dim, batch size and learning rate are the most important pa-

rameters, while use skip connections is not that important. For the most part, there are not many outliers and the values are evenly distributed.

3.2. Attention

We will try to experiment with the attention mechanism. The idea behind attention is to allow the model to focus on different parts of the input sequence when predicting each part of the output sequence, similar to how humans pay attention to different parts of the visual field or a sentence when processing information. The attention mechanism calculates a score between the query and each of the keys. This score determines how much focus to put on each part of the input sequence. The scoring function can be a simple dot product or a more complex function like a small neural network. The raw scores are passed through a softmax function to normalize them into a probability distribution. The softmax function ensures that all the scores are positive and sum to one. The normalized scores are then used to create a weighted sum of the values. During training, the attention weights (the parameters of the scoring function) are learned along with the rest of the model parameters. The model learns to adjust the attention weights to focus on the most informative parts of the input sequence for predicting each part of the output sequence.
For the first approach of the attention mechanism we create a class called Attention. The Attention class defines a linear layer that projects the concatenated hidden states from the bidirectional RNN to a single attention score per time step. The forward method of the Attention class takes the rnn_output (the sequence of hidden states from the RNN), computes attention scores, applies a softmax to get attention weights, and then computes a context vector as the weighted sum of the hidden states. In the SentimentClassifier, the attention mechanism is applied to the output of the RNN layer. If use_skip_connections is set to True and the input dimension matches the RNN output dimension, a skip connection is added by element-wise adding the input x to the rnn_out. The attention mechanism as implemented is a simple global attention where all time steps of the RNN outputs are used to compute a single context vector. This context vector is then used to make a classification decision for the entire sequence.

The second approach of the attention method implement a MultiHeadAttention class. The MultiHeadAttention class defines a multi-head attention mechanism, which allows the model to jointly attend to information from different representation subspaces at different positions. This is done by projecting the hidden states into num_heads different attention heads and then performing scaled dot-product attention for each head. The attention score is calculated using a scaled dot-product of the queries and keys, and scaled by the square root of the depth of the attention heads to stabilize gradients during training. After computing the attention, the outputs of all heads are concatenated and then linearly transformed with self.dense. The context vector returned by the Attention mechanism is then fed into a fully connected layer (self.fc) to produce the final output. For the last approach of the attention method we will try to implement a mechanism called Bahdanau attention, which is a popular method used in various natural language processing tasks. This type of attention was introduced by Dzmitry Bahdanau in the context of neural machine translation and is designed to help the model focus on specific parts of the input sequence when generating each part of the output

sequence. The class takes hidden_dim as an argument, which is the dimension of the hidden states in the LSTM. key_layer and query_layer are linear layers that transform the encoder outputs and the hidden state, respectively. Since LSTM is bidirectional, the hidden state's size is hidden_dim * 2. energy_layer computes a scalar value (the alignment score) from the combination of key and query. Softmax is used to normalize the alignment scores across the sequence length. The energy_layer processes this result to produce a scalar energy score for each time step, and squeeze is used to remove the last dimension, leaving a 2D tensor. The energy scores are normalized using softmax to create attention weights. These weights sum up to 1 and indicate the importance of each time step in the sequence.
For the trials with Attention mechanism the most important hyperparameter seems to be learning rate. This is normal since, the introduction of the attention mechanism adds complexity and sensitivity to the model's learning process, making the learning rate a critical hyperparameter to ensure effective learning and generalization. This underscores the importance of hyperparameter tuning, especially in sophisticated models like those involving attention mechanisms.
The results will be presented in the table below. Keep in mind that the actual values in the notebook in submission differ, this table consists of the mean values, measured from 5 different executions of the program

Trial	Cell	Representation	Attention	Score
1	LSTM	Mean Embedding	-	0.399
2	GRU	Mean Embedding	-	0.395
3	Simple RNN	Mean Embeding	-	0.394
4	LSTM	TF-IDF	-	0.397
5	GRU	TF-IDF	-	0.393
6	LSTM	Mean Embedding	Simple-Global	0.389
6	LSTM	Mean Embedding	Multihead	0.398
6	LSTM	Mean Embedding	Bahdanau	0.394

Table 1: Trials

3.3. Hyper-parameter tuning

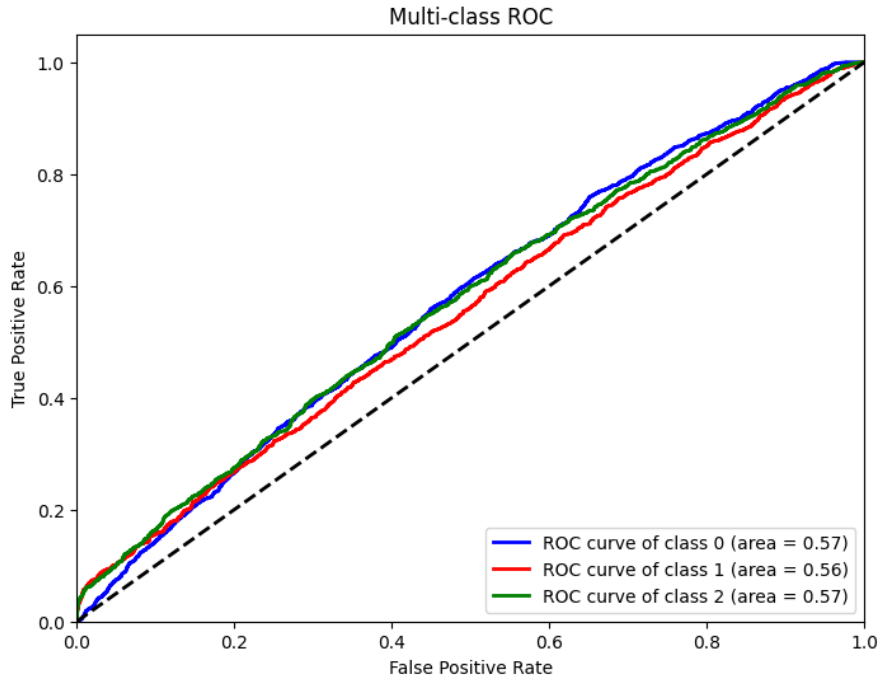
For the hyperparameters tuning we are using optuna to maximize f1-score and keep the best hyperparameters

3.4. Optimization techniques

The framework used for optimization is optuna.

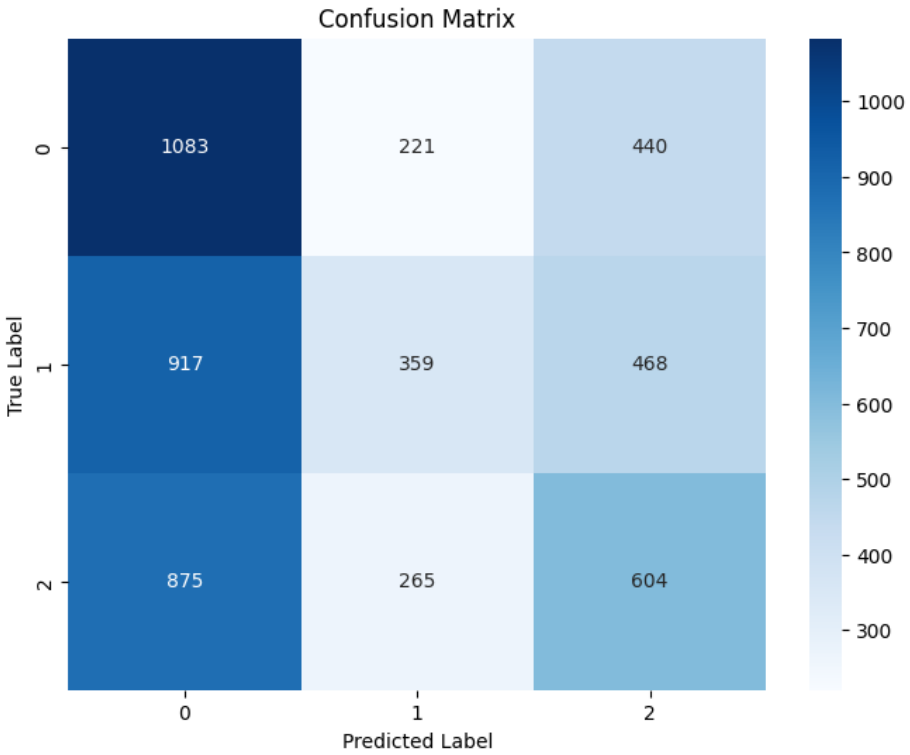
3.5. Evaluation

The predictions are around 0.39, while the best fscore I got was 0.404 with LSTM and mean embedding. Overall the best method is LSTM with mean embedding, and since we dont really have time limitations, this will be used to submit the predictions. There is no overftting or underffiting in the model and this can be proven from he diagrams and curves.



3.5.1. ROC curve. The AUC for each class is around 0.56 to 0.57. The AUC value ranges from 0 to 1, where 1 indicates a perfect model and 0.5 represents a model with no discriminatory power, equivalent to random guessing. The AUC values here are only slightly better than what would be expected from random guessing, suggesting that the model is not performing well at discriminating between the positive class and the negative class for any of the classes. The performance across the three classes seems to be roughly similar, with none of the classes showing a significantly different AUC from the others. This indicates that the model does not favor any particular class and is equally challenged in distinguishing each class from the others. The curves are close to the diagonal line, which is the line of no-discrimination. This suggests that the model's ability to correctly identify true positives over false positives is not much better than random chance, indicating that the model needs significant improvement. With such bad result we might say that the model is underfitting, but this is due to the dataset.

3.5.2. Learning Curve. Both the training and validation loss are decreasing, which indicates that the model is learning and improving its predictions over time. The gap between the training and validation loss is relatively small, which suggests that the model is generalizing well to unseen data. There is no clear sign of overfitting, as overfitting would be characterized by a large gap where the training loss continues to decrease significantly while the validation loss starts to increase or plateaus. The losses are relatively high and seem to be decreasing at a slow rate, which might suggest underfitting. This would mean that the model has not yet captured the underlying patterns in the data sufficiently, perhaps because those patterns do not exist in this dataset.



3.5.3. Confusion matrix. For class 0, there are 1083 true positives, meaning 1083 instances were correctly identified as class 0. For class 1, 359 instances were correctly identified. For class 2, 604 instances were correctly identified Class 0 seems to be relatively well-classified, although there is a considerable number of instances (440) misclassified as class 2. Class 1 and class 2 have a high number of misclassifications. Many instances of class 1 are misclassified as class 0 (917) or class 2 (468), and many instances of class 2 are misclassified as class 0 (875). There seems to be confusion between class 1 and the other two classes, which is evident from the relatively high number of misclassifications. Similarly, class 2 is often confused with class 0.In conclusion, the confusion matrix suggests that the model is better at identifying class 0 than class 1 or class 2. There is a notable amount of confusion between the classes, especially for classes 1 and 2, indicating that the model's ability to distinguish between these classes could be improved.

4. Results and Overall Analysis

4.1. Results Analysis

The scores we are getting are all around 0.39 with the deviation to be smaller than 0.01 most of the time. Introducing the Attention mechanism does not yeild better results. Furthermore, the scores are not consistent, with some sets of hyperparameters the score drops at less than 0.3 which is worse than random guessing. While the mechanism works, the results are not better, this is perhaps dueo to the dataset. So, for the final predictions, it will not be used. The best approach of Attention was the multi-head Attention mechanism, with results close to LSTM with mean embedding but not better. So the final model consists of LSTM with mean embedding, the exact hyperparameters are given in the subsection below. After coming up with the final model, we are concatenating train and validation set so to get an expanded Word2Vec model. In the last trials with the final model at the validation phase, we have also introduced an early stopping mechanism, that most of the times stops at epoch 25. Since the model is expanded, we will chose to stop at epoch 30 when predicting the test set.

4.1.1. Best trial. Best trial was with LSTM and mean embedding with fscore = 0.403. The values of the hyperparameters were :

- input_dim = 190
- hidden_dim = 20
- num_layers = 3
- window_size = 7
- learning_rate = 0.0006802710564956603
- dropout_rate = 0.33171542389215286
- b_size = 55
- use_skip_connections = False
- gradient_clip_val = 0.5743447268798871

4.2. Comparison with the first project

The results compared to the first project that was using Logistic Regression are worse. This is perhaps the sentiment within tweets may be determined by specific keywords or phrases that are easily captured by logistic regression, indicating that the relationship between the features and the target variable may be linear or close to linear. While LSTMs typically require a large amount of data to perform well since they have more parameters to train. If the dataset is small, simpler models like logistic regression can outperform more complex models.

4.3. Comparison with the second project

The results compared to the second project that was using a deep Neural Network are mostly the same. This happens perhaps because the sentiment classification task may not require capturing long-term dependencies within the text, which is where LSTMs excel. Simple patterns or cues might be sufficient to determine sentiment, which can be captured by a DNN. The DNN may be able to capture enough contextual information without the need for LSTM's sequence modeling capabilities. LSTMs are designed to handle sequential data and can capture temporal dynamics. If the sequence in which words appear in tweets does not hold significant information for sentiment classification, an LSTM's advantage may not be evident, and a DNN could perform similarly.

5. Bibliography

1. https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html
2. https://pytorch.org/docs/stable/generated/torch.nn.GRU.html
3. https://scikit-learn.org/0.21/documentation.html
4. https://arxiv.org/abs/2112.05561
5. https://machinelearningmastery.com/global-attention-for-encoder-decoder-recurrent-neural
6. https://d2l.ai/chapter_attention-mechanisms-and-transformers/multihead-attention.html
7. https://d2l.ai/chapter_attention-mechanisms-and-transformers/bahdanau-attention.html