

Deep Learning for NLP

Student name: *Dimitrios Tsiompikas*
sdi: 7115112300036

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

Contents

1	Abstract	2
2	Data processing and analysis	2
2.1	Pre-processing	2
2.2	Analysis	2
2.3	Data partitioning for train, test and validation	9
2.4	Embeddings	9
3	Algorithms and Experiments	10
3.1	Experiments	10
3.1.1	Table of trials	10
3.2	Hyper-parameter tuning	11
3.3	Optimization techniques	11
3.4	Evaluation	11
3.4.1	ROC Curve	12
3.4.2	Learning Curve	13
3.4.3	Confusion matrix	14
4	Results and Overall Analysis	15
4.1	Results Analysis	15
4.1.1	Best trial	18
4.2	Comparison with the first project	18
5	Bibliography	19

1. Abstract

Goal of the second assignment is to train and evaluate a Feed-forward Neural Network to identify sentiment (POSITIVE, NEUTRAL or NEGATIVE) of tweets based on a Tweet dataset from the 2023 Greek Elections. I will fulfill this task by following a classic ML process which is: Firstly clean the dataset from data that might harm our train, test and valid sets. Then create some visualizations to get a grasp of what I'm dealing with, using tokens, the sentiment column, word similarity and linear substructures. Train the model with the techniques mentioned below (Word2Vec, hyperparameter tuning, different types of activation functions and layers for my NNs) and finally evaluate the model with metrics and plots.

2. Data processing and analysis

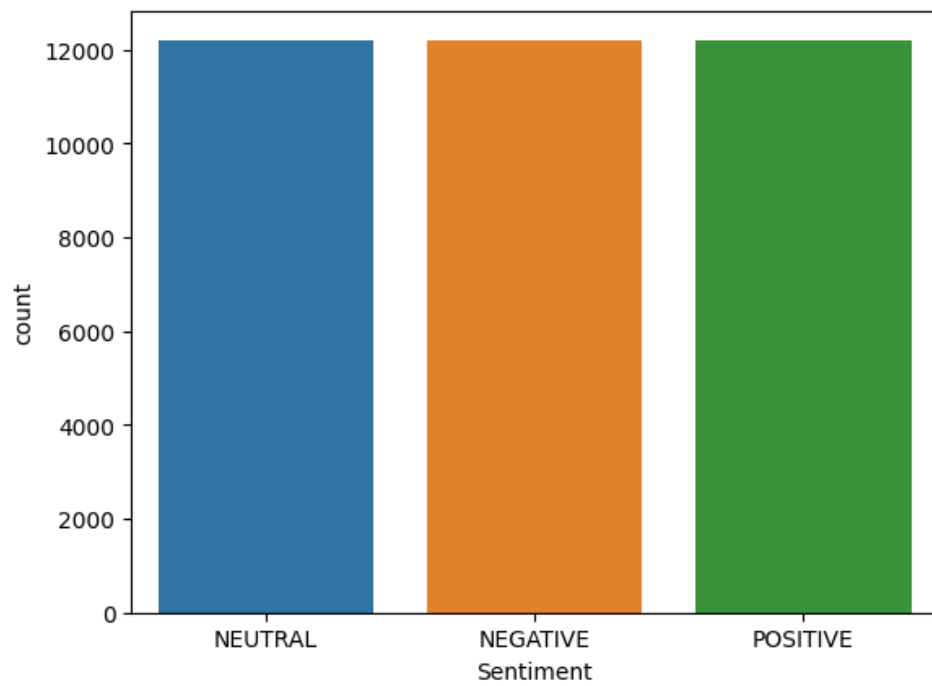
2.1. Pre-processing

For data pre-processing, I firstly checked for null values to clean them, there were none so I followed the cleanup with a hand-made tweet cleanup function that removes all mention signs, hashtags, links (URLs), punctuations and all words after mentions or hashtags (example: @akis18, I removed akis18) as they were in English and would cause trouble with model training. I also created a stopwords list in Greek, removed all useless Greek words, then I added English words in the list as well (taken from the tweets of all datasets) and removed them as well from the tweets in order to keep the model from having to deal with words in two languages. Lastly, I removed special symbols to further improve my model training and get the best possible results. In this part, I noticed that the dataset has some issues that cannot be fixed in such a large amount of data like wrong sentiment in specific tweets or sometimes even the wrong party. This will cause problems later on with overfitting and keeping the scores on low percentages.

2.2. Analysis

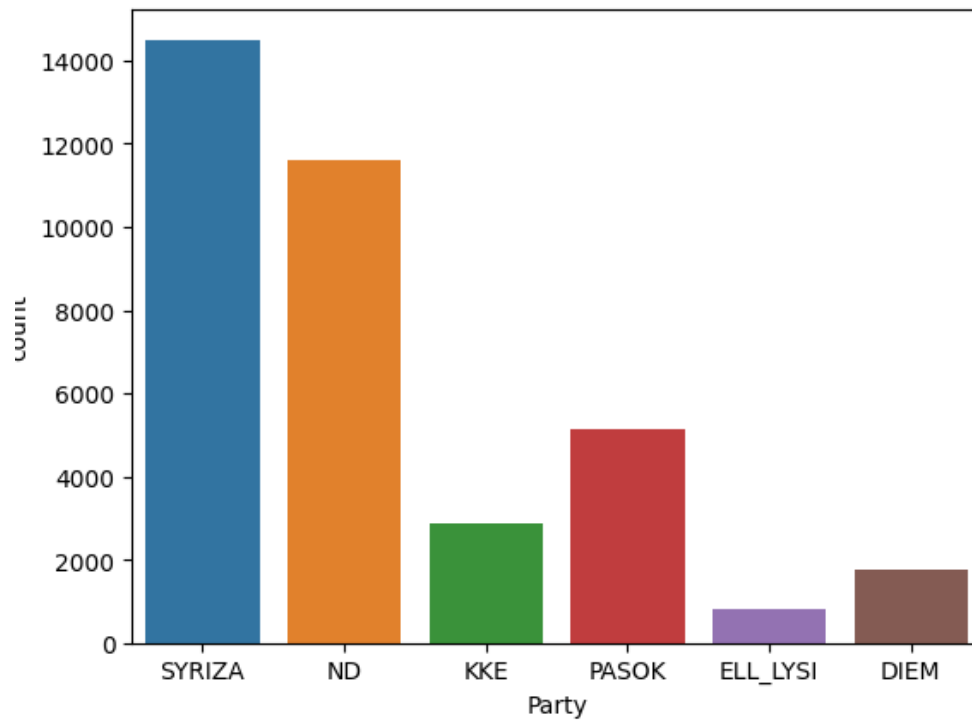
Firstly, I began with analyzing the dataset columns:

Figure 1: Sentiment countplot



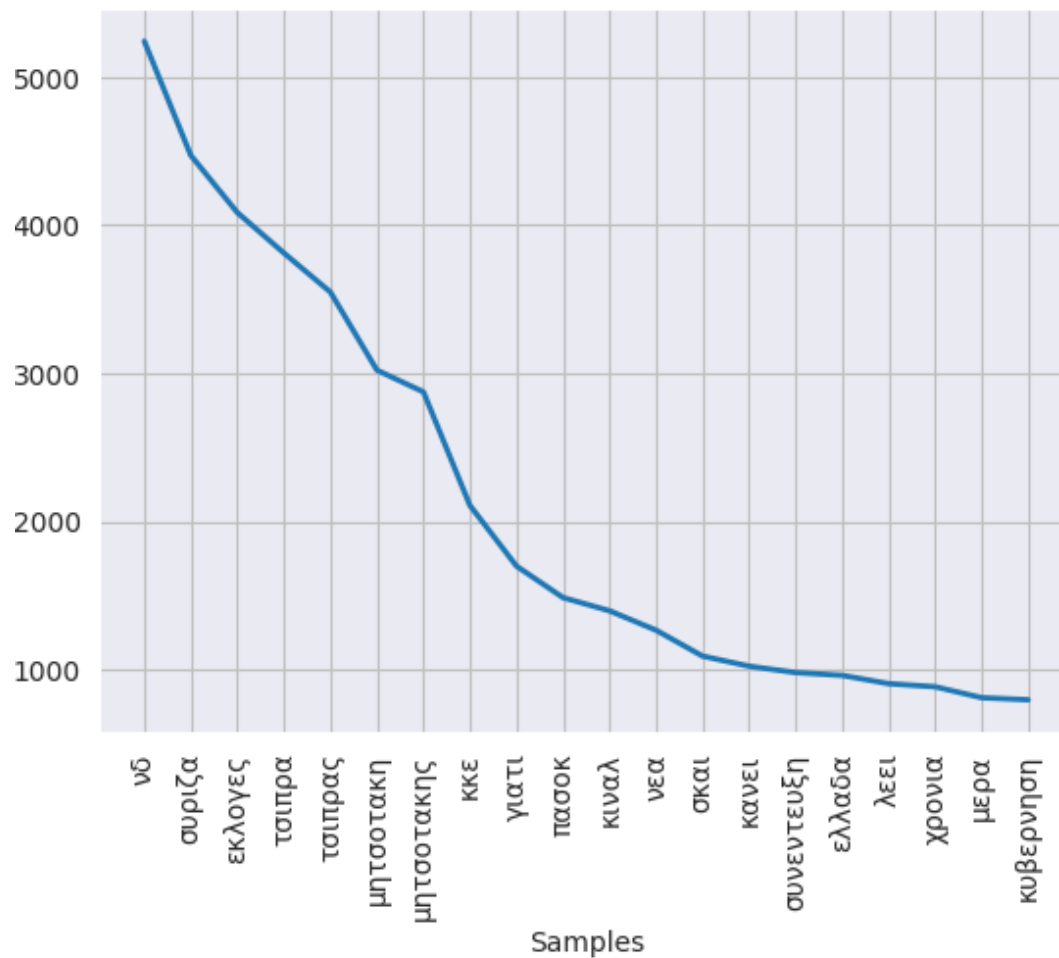
We have the same count for all sentiments.

Figure 2: Party countplot



Here, we can see that the most prevalent party in the tweets is SYRIZA which makes sense since SYRIZA was the political opposition for ND.

Figure 3: Word frequency diagram from Text column



From this diagram we can see the most frequent tokens in the Text column. We can see that the first ones are related to Alexis Tsipras, Kyriakos Mitsotakis, ND and SYRIZA as they were the two main competitors for the 2023 elections. Afterwards, parties with lower percentages follow such as KINAL and KKE and some other words related to elections or politics.

Figure 5: Most similar words to the word Mitsotakis


Most semantically similar to word Mitsotakis:

word	similarity score
=====	
μητσοτακης...	0.9551254510879517
ζητησε	0.9532033205032349
ντιξον	0.9503481984138489
αποδειχθει	0.9468908309936523
προσφυγικο	0.945105791091919
συνθεση	0.9412325024604797
ταξιδευοντας	0.9361863732337952
χιουγκο	0.9341863393783569
μειωσουμε	0.932820200920105
τολμη	0.9320380091667175

Here we can see some words that are similar to the word Mitsotakis using the word embeddings, the first word is indeed similar since it's the same thing, the rest might correspond to something that Mitsotakis has done, or words that follow the word Mitsotakis right after in tweets.

Figure 6: Most similar words to the word Tsipras

Most semantically similar to word Tsipras:

word	similarity score
=====	
σκαϊτσιπρας	0.9553589820861816
	0.9534194469451904
τηλεθεασης	0.9530816078186035
ζαγορα	0.9527417421340942
τσιπραςσκαϊ	0.9513566493988037
κιν	0.9511148929595947
σπαει	0.9500080347061157
πρωινες	0.9495740532875061
σκαϊξεφτιλες	0.948449432849884
ευχαριστη	0.9474217295646667

Here we can see words that are similar to the word Tsipras , again, using the word embeddings. Apart from every word that contains the same substring (tsipras) the rest of the words seem to correlate to a TV channel named SKAI in Greece which is known to be more ND-favored among Greek citizens and probably gets backlash to all tweets regarding Alexis Tsipras hence the similarity.

Figure 7: Linear substructure visualization using Word Embeddings



Here I used the words "μητσοτακης, τσιπρας, συριζα, νδ , κιναλ, κει" as my vocabulary for the Linear substructure visualization because they're very common words in the tweets. The distancing between words is apparent due to issues in the dataset.

2.3. Data partitioning for train, test and validation

I've used the train set for training , test set to predict the labels and valid set to create the metrics.

2.4. Embeddings

For this assignment we had to use word embeddings to create the input tensors for our neural networks. I used Word2Vec library from gensim, I tokenized each tweet (Text column of each dataset) after preprocessing, created sentences for each dataset , trained my word2vec model with them and then created word embeddings for each word (with vectors of size 100). Afterwards, I convert the tweets into embeddings , one-hot encode the labels of the train and valid data sets to add them to the tensors and also created a small dictionary to map the one hot vectors to numbers 0,1,2 so I can create the submission.csv later on. Finally, I add the tweet embeddings to torch tensors in order to feed them to the NNs.

3. Algorithms and Experiments

3.1. Experiments

Firstly, I have created a brute force Feed-forward NN to begin the experiments in order to get a grasp of what I will deal with. The NN consists of 1 input layer (which is of size 100 to take in the vectors of the word embeddings from the tensors), 4 hidden layers and an output layer (which has 3 columns since we have 3 labels). For the initial experiment, I used 4 loss functions (MSELoss, BCELoss, BCELossWithLogits, Cross Entropy Loss) and optimizers (Adam, AdamW, SGD). I found out that the best loss function was cross entropy loss and the best optimizer (for all NNs) was Adam, with the best reduction in loss (both validation and training) for each epoch, SGD was by far the worst out of all three, with terrible performance in all NNs (lowest scores) and very low reduction of loss with each epoch. I also created the tensor datasets and put the train and valid datasets in tensor dataloaders. For training, I used the train set dataloader and for evaluation of the model I used the valid set dataloader. Scores, due to creating batches, fluctuate for all NNs and for the brute force one I got 36-37% F1-score which is pretty low.

Furthermore, I took the initiative to test several Feed-forward NN models to check which one would have the best performance for the experiment. The NNs were: a Softmax layer NN which has ReLU applied to all hidden layers and a dropout layer for the final hidden layer and then the softmax activation function is applied to the output layer. A Sigmoid NN which has a sigmoid activation function applied to the final layer at the end and finally a brute force NN upgraded with ReLU and dropout layers (which I called ReLU / Dropout NN). Loss functions used for each NN were: MSELoss for Softmax NN, BCELoss for Sigmoid NN (converts my results to binary) and BCELossWithLogits / CrossEntropyLoss for ReLU / Dropout NN. In the table below you will find all scores for my trials with these NNs. The best one was the Softmax NN which was also used with Optuna for hyperparameter tuning. All tests were done with 16 epochs as I found out with the learning curve that this number of epochs gave out the best results.

Lastly, as I will mention below, I used Optuna to calculate the best hyperparameters and find the optimal metric score. Using it I found the best hyperparameters for the learning rate, dropout rate and optimizer. You can see more about these in subsection 3.2.

3.1.1. Table of trials.

Trial	Learning rate	optimizer	Loss function	Score
Brute-force NN	1e-3	SGD	CrossEntropyLoss	36.6%
Brute-force NN	1e-3	Adam	CrossEntropyLoss	37.2%
Brute-force NN	1e-3	AdamW	CrossEntropyLoss	37.8%
ReLU / Dropout NN	1e-3	SGD	CrossEntropyLoss	37.1%
ReLU / Dropout NN	1e-3	Adam	CrossEntropyLoss	39%
ReLU / Dropout NN	1e-3	AdamW	CrossEntropyLoss	38%
ReLU / Dropout NN	1e-3	SGD	BCEWithLogitsLoss	32.3%
ReLU / Dropout NN	1e-3	Adam	BCEWithLogitsLoss	38.3%
ReLU / Dropout NN	1e-3	AdamW	BCEWithLogitsLoss	38.2%
Sigmoid NN	1e-3	SGD	BCELoss	37.6%
Sigmoid NN	1e-3	Adam	BCELoss	38.8%
Sigmoid NN	1e-3	AdamW	BCELoss	38.7%
Softmax NN	1e-3	SGD	MSELoss	33.8%
Softmax NN	1e-3	Adam	MSELoss	38.8%
Softmax NN	1e-3	AdamW	MSELoss	38.1%
Softmax NN (best params)	1e-3	SGD	MSELoss	33.8%
Softmax NN (best params)	1e-3	Adam	MSELoss	39.3%
Softmax NN (best params)	1e-3	AdamW	MSELoss	38.8%

Table 1: Trials

3.2. Hyper-parameter tuning

I used the Optuna library to tune the model's hyperparameters. I created an objective function that trains a Softmax NN with learning rate, dropout rate and optimizers in mind. However, Optuna, due to batches, also gives different values each time it's ran, so I got the ones that I saw best results with. The best values I found for these hyperparameters are:

Learning rate = 1e-3

Dropout rate = 0.3

Optimizer = Adam

Providing the best score which was 39.3% (40% is reachable though, it's just a matter of the fluctuations due to batches, with a proper seed).

3.3. Optimization techniques

I used Optuna mostly to find the best hyperparameters, added extra layers in the NNs, made trials with different batch numbers, created new neural networks. There was difference between every NN but it wasn't that big. However, Softmax NN outperformed them all.

3.4. Evaluation

The predictions were evaluated using the valid set sentiment column, I put all NNs to evaluation mode, compared my predicted results with the actual values from the y

batch of the valid dataset and I used the following metrics:

Classification report: this produces several metrics all-in-one , namely Recall,Precision,F1-score and Accuracy.

Recall: this shows how many true results are returned from the predictions.

Precision: this shows how relevant are the results from the predictions.

F1-score: this uses recall and precision to find out how accurate the model is with its predictions. It ranges from 0 to 1 and the closer it is to 1 the better.

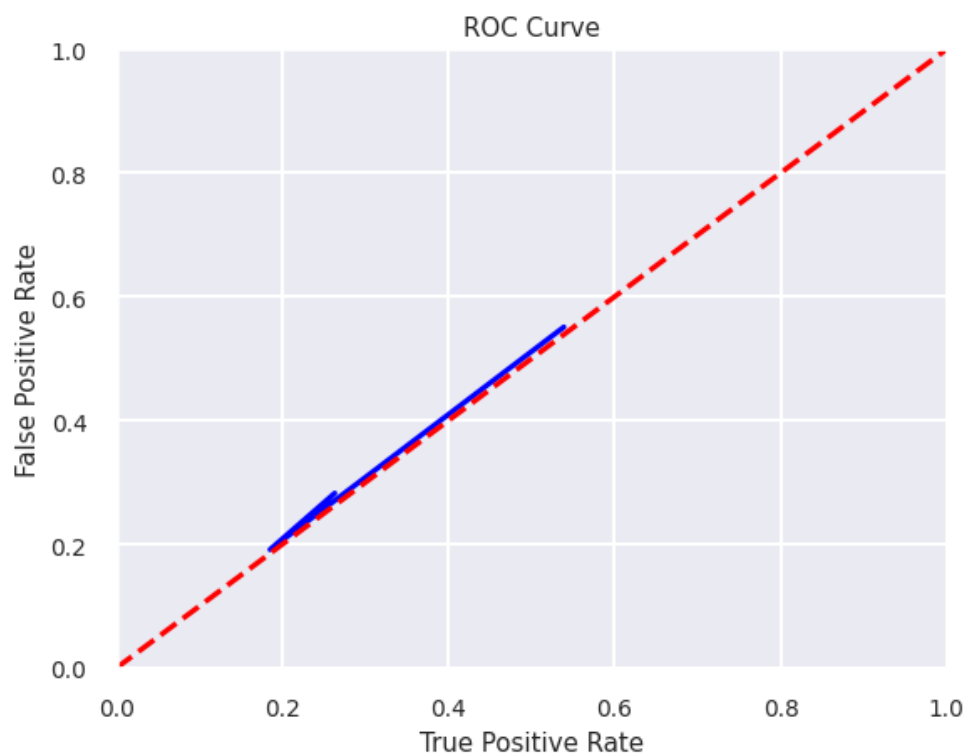
Accuracy: this calculates the accuracy of the model by calculating the fraction of number of true results to the number of total results.

Confusion matrix: This is a table that shows the True positive , true negative, false positive and false negative counts of values that the model has found. It's used to calculate the metrics above.

Below I will add figures showcasing each of the results. Also I will add the results of the brute-force experiments:

3.4.1. ROC Curve.

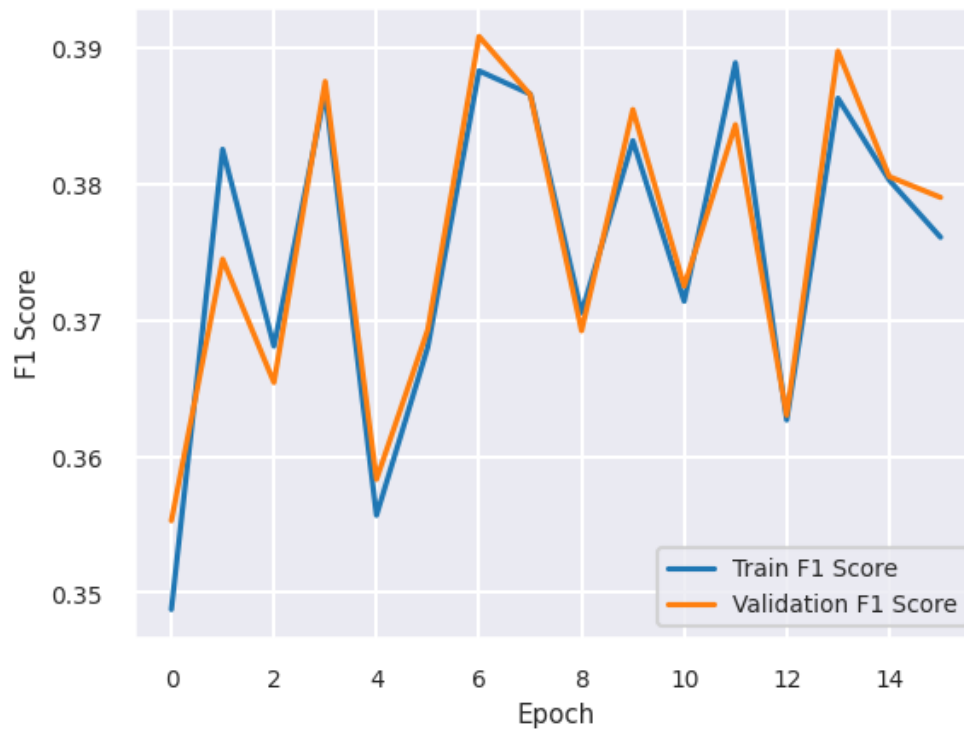
Figure 8: ROC Curve with best hyperparams



3.4.2. Learning Curve.

Here we observe that at 6 epochs the model reaches its peak F1 score, and then has fluctuations due to overfitting. The learning curve can differ from batch to batch but overall F1-score remains the same (38-40% with best hyperparameters).

Figure 9: Learning curve with best hyperparameters



3.4.3. Confusion matrix.

Figure 10: Confusion Matrix with best hyperparameters (best trial)

```
[[ 961  329  454]
 [ 946  334  464]
 [ 934  317  493]]
```

4. Results and Overall Analysis

4.1. Results Analysis

In conclusion, the results of the experiment were pretty underwhelming again. Since the dataset problems were beyond repair the model only produced results up to 40%. On average, Neural networks usually produce around 65-75% with proper tuning and data cleanup. So the expected results would be in that amount of percentages. I also noticed that the training loss and validation loss were fluctuating (going up and down) due to overfitting after 7-8 epochs, I tried ReLU for regularization to fix as much as I could, but it would still overfit. Some more techniques I could have used include: Used GLoVe pre-trained embeddings, used some kind of seed to lock the batches to one set of batches so I could measure changes more properly. I believe I did a handful of experiments taking note of each percentage so the only thing I'd do additionally is using the techniques mentioned previously.

Figure 11: Brute force NN scores

```
===== BRUTE FORCE NEURAL NETWORK SCORES =====
      precision    recall  f1-score   support

   NEGATIVE       0.38       0.47       0.42       1744
    NEUTRAL       0.40       0.21       0.28       1744
   POSITIVE       0.36       0.45       0.40       1744

 accuracy                   0.38       5232
 macro avg       0.38       0.38       0.37       5232
weighted avg       0.38       0.38       0.37       5232

accuracy:  0.3765290519877676
f1:  0.3765290519877676
total f1:  [0.42148338 0.27578391 0.39979524]
```

Figure 12: Softmax NN scores (no hyperparameters)

```

===== SOFTMAX NN SCORES =====
      precision    recall  f1-score   support

    NEGATIVE      0.38      0.53      0.44      1744
     NEUTRAL      0.40      0.26      0.32      1744
    POSITIVE      0.39      0.37      0.38      1744

 accuracy          0.39          5232
  macro avg       0.39      0.39      0.38      5232
 weighted avg     0.39      0.39      0.38      5232

accuracy: 0.3885703363914373
f1: 0.3885703363914373
total f1: [0.44481766 0.31626298 0.38109219]

```

Figure 13: Sigmoid NN scores

```

===== SIGMOID NN SCORES (BCE LOSS) =====
      precision    recall  f1-score   support

    NEGATIVE      0.38      0.56      0.46      1744
     NEUTRAL      0.45      0.14      0.22      1744
    POSITIVE      0.38      0.46      0.42      1744

 accuracy          0.39          5232
  macro avg       0.40      0.39      0.36      5232
 weighted avg     0.40      0.39      0.36      5232

accuracy: 0.389717125382263
f1: 0.389717125382263
total f1: [0.45545013 0.21837088 0.41877445]

```


Figure 14: ReLU/Dropout NN scores (Cross Entropy Loss)

```

===== RELU AND DROPOUT NN SCORES (CROSS ENTROPY LOSS) =====
      precision    recall  f1-score   support

   NEGATIVE       0.39      0.49      0.43      1744
    NEUTRAL       0.42      0.16      0.23      1744
   POSITIVE       0.37      0.50      0.42      1744

 accuracy          0.38      5232
  macro avg       0.39      0.38      0.36      5232
 weighted avg     0.39      0.38      0.36      5232

accuracy: 0.3841743119266055
f1: 0.3841743119266055
total f1: [0.4341637 0.23485477 0.42378641]

```

Figure 15: ReLU/Dropout NN scores (BCEWithLogits Loss)

```

===== RELU AND DROPOUT NN SCORES (BCE LOSS WITH LOGITS) =====
      precision    recall  f1-score   support

   NEGATIVE       0.42      0.32      0.36      1744
    NEUTRAL       0.41      0.20      0.27      1744
   POSITIVE       0.36      0.63      0.46      1744

 accuracy          0.38      5232
  macro avg       0.40      0.38      0.36      5232
 weighted avg     0.40      0.38      0.36      5232

accuracy: 0.38245412844036697
f1: 0.38245412844036697
total f1: [0.35955787 0.27203065 0.4575136 ]

```

Here we see the difference of performance between the NNs. Softmax NN was better in this experiment so I used it to run my best trial.

4.1.1. Best trial. Below, you will find my metrics for my best trial. Essentially, after testing the problem thoroughly, I concluded in using the following techniques: Soft-max NN, with 0.001 learning rate, dropout rate = 0.3 and Adam as the optimizer, after hyperparameter tuning. Best f1-score resulted in 39.3%.

Figure 16: Best trial classification report, with micro f1 score and accuracy

```

===== BEST HYPERPARAMETER SOFTMAX NN SCORES =====
              precision    recall  f1-score   support

   NEGATIVE         0.38      0.62      0.47      1744
    NEUTRAL         0.41      0.23      0.30      1744
   POSITIVE         0.41      0.33      0.37      1744

 accuracy          0.39          5232
  macro avg         0.40      0.39      0.38      5232
 weighted avg       0.40      0.39      0.38      5232

accuracy:  0.3927752293577982
f1:  0.39277522935779824
total f1:  [0.46848419 0.29735683 0.36513471]

```

Figure 17: Confusion Matrix with best hyperparameters (best trial)

```

[[961 329 454]
 [946 334 464]
 [934 317 493]]

```

4.2. Comparison with the first project

The results in this project are pretty much the same, due to the false labeling in the dataset, even after conducting several experiments with Optuna and different types of Feed-Forward NNs.

5. Bibliography

References

- [1] Rohit Agrawal. Using fine-tuned Gensim Word2Vec Embeddings with Torchtext and Pytorch. <https://rohit-agrawal.medium.com/using-fine-tuned-gensim-word2vec-embeddings-with-torchtext-and-pytorch-17eea2883cd>, 2020.
 - [2] lucidv01d. Calculate metrics through confusion matrix. <https://stackoverflow.com/questions/31324218/scikit-learn-how-to-obtain-true-positive-true-negative-false-positive-and-fal>, 2017.
 - [3] Neri Van Otten. Word2Vec for text classification. <https://spotintelligence.com/2023/02/15/word2vec-for-text-classification/#Conclusion>, 2023.
 - [4] Milind Soorya. Introduction to Word Frequency in NLP using python. <https://www.milindsoorya.com/blog/introduction-to-word-frequencies-in-nlp#data-processing>, 2021.
- [4] [2] [1] [3]
 [Word frequency with NLTK] [Metric calculation for ROC curve using confusion matrix] [Using fine-tuned Gensim Word2Vec Embeddings with Torchtext and Pytorch] [Word2vec text classification methods]