UNIVERSITY OF ATHENS
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

Deep Learning for NLP

Student name: *Nikitas Rafail Karachalios*
sdi: -

Course: *Artificial Intelligence II (M138, M226, M262, M325)*
Semester: *Fall Semester 2025*

Contents

1. Abstract

The aim of this project is to develop a sentiment classifier using Deep Learning (PyTorch) and pretrained Word2Vec embeddings (Google), on a provided English-language Twitter dataset. More specifically, the goal was to create a neural network that could classify each tweet as either positive or negative. To achieve this, I followed a pipeline consisting of data preprocessing, vectorization using Word2Vec, neural network implementation, and multiple experiments to improve performance (including the use of Optuna for hyperparameter tuning).

Compared to Project 1, this project introduced several challenges, including handling vector representations with pretrained embeddings and understanding deeper architectures (batch normalization, dropout, etc.). To deal with these, I experimented iteratively, tested variations in preprocessing and architecture, and analyzed performance through classification reports, learning curves, and ROC curves. It was pretty challenging since training a deep DNN isn't a walk in the park as highlighted in lecture slides! More hyperparameters, more complex models lead to more experiments.

2. Data processing

2.1. Pre-processing

Based on Homework slides, prior experience from Project 1, and experimental results, I decided to use an extended preprocessing pipeline with manual misclassification handling, as well, as mentioned from Yorgos Pantis. My baseline criterion for choosing each step was the validation accuracy and the overall impact on model performance and overfit avoidance. The full steps are summarized below:

1. **Lowercasing**: To reduce vocabulary size and unify words like "Happy" and "happy".
2. **URL removal**: URLs don't contribute to sentiment; they are just noise.
3. **User mentions removal**: Removing @mentions since usernames don't carry sentiment. We could replace it with @USER but the complete removal led to better results
4. **Hashtag symbol removal (keeping word)**: Hashtags can contain sentiment-related words, so I only removed the symbol # but kept the actual word.
5. **Emoticon replacement**: I replaced emoticons (like ":)" or ":(") with corresponding sentiment words (like "happy", "sad") so the model could capture their meaning through Word2Vec embeddings. This was crucial since Word2Vec only works with words, not symbols.
6. **Negation handling**: Converting contractions ("can't" $\rightarrow$ "can not") to their expanded form to preserve the negation structure, which affects sentiment. I was curious to try a model without them, and have an embedding "lol" for example, but this didn't improve performance in my model.

7. **Corrections**: Converting slang, abbreviations, and dataset-specific abbreviations (e.g., “lol” → “laugh out loud”) to their standard form, making sure Word2Vec embeddings exist for them.
8. **Repeated character handling**: I reduced sequences of more than 2 consecutive identical letters to 2 (e.g., “sooo happyyy” → “soo happyy”). This helped reduce noise while keeping some emphasis.
9. **Replace $ with “money” token**: To capture monetary references explicitly as a token (through manual misclassification, explained below).
10. **Tokenization**: I tokenized the text using TreebankWordTokenizer to split into words.
11. **Punctuation removal**: I removed all punctuation tokens, except for “!” and “?”, since these can express emotion and contribute to sentiment (through manual misclassification, explained below).

Some of the other approaches that didn’t help can be seen in preprocess appendices, submitted as a comment to my code (stopwords, lemmatization etc).

2.2. Manual error analysis

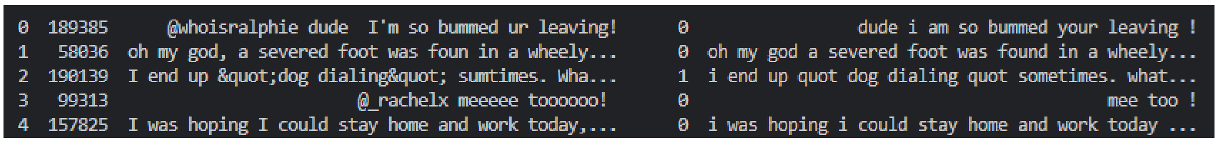
In order to further improve preprocessing and better understand the model’s behavior, I manually inspected misclassified tweets (true label vs prediction). Some representative examples are presented below, along with my interpretation of why they were misclassified:

- **TRUE: 1, PREDICTED: 0, TEXT:** *Want: Trip to Boston next month. Need: Addit’l motivation to save the $ to do so. Beloved daughter wants to go with, which = 2x the $.*
  This tweet contains positive planning and excitement, but the dollar symbols and the “need to save” phrasing might confuse the model into negative sentiment. This motivated me to explicitly replace $ with a “money” token in preprocessing to highlight its meaning.
- **TRUE: 1, PREDICTED: 0, TEXT:** *first day starts tomorrow!*
  Despite the exclamation mark and hopeful wording, the model misclassified it. This confirmed that keeping “!” in tokenization was a good choice since it signals excitement.
- **TRUE: 1, PREDICTED: 0, TEXT:** *No milk for breakfast, grrr. But Looks like a beach day!*
  The initial complaint (“no milk”) might have led to confusion despite the positive ending. This showed that mixed sentiments in tweets are hard for simple models, and highlights the need for context. One more example that shows the role of the "!".
- **TRUE: 1, PREDICTED: 0, TEXT:** *@USER no i didnt go, saving $$ for summer instead*
  Here again, the dollar signs and negation “no i didnt go” probably confused the model toward negative. Reinforces why I kept “$” replacement and negation handling.

These examples helped me justify specific preprocessing decisions (like keeping "!" and adding "money" token), and also revealed limitations in the architecture that could be explored in future work (e.g., using sequence models or attention mechanisms).

2.3. Preprocessing results-visualizations

We remind from the previous project that our datasets are perfectly balanced and not duplicate or empty tweets appear so no issues with data quality. Now, let's see our preprocessing in action by comparing the head of the training dataset before and after preprocessing.



```
0  189385      @whoisralphie dude  I'm so bummed ur leaving!      0                 dude i am so bummed your leaving !
1   58036  oh my god, a severed foot was foun in a wheely...      0  oh my god a severed foot was found in a wheely...
2  190139  I end up &quot;dog dialing&quot; sumtimes. Wha...      1  i end up quot dog dialing quot sometimes. what...
3   99313                        @_rachelx meeeee toooooo!      0                                         mee too !
4  157825  I was hoping I could stay home and work today,...      0  i was hoping i could stay home and work today ...
```

Above we can clearly see the effect of the preprocessing. "I" is same as "i" for the reduction of vocabulary size, spelling mistakes are corrected ("sumtimes" to "sometimes"), consecutive letters are reduced to 2, and punctuation is removed except for "!" and "?" for the reasons we explained above.

Most frequent words

Let's see the visualization of training and validation for most common words (excluding stop words) before and after preprocessing to feel even more confident about our text cleaning.

For training dataset, before preprocess (with or without) stopwords we see noisy, imbalanced data with useless punctuation.

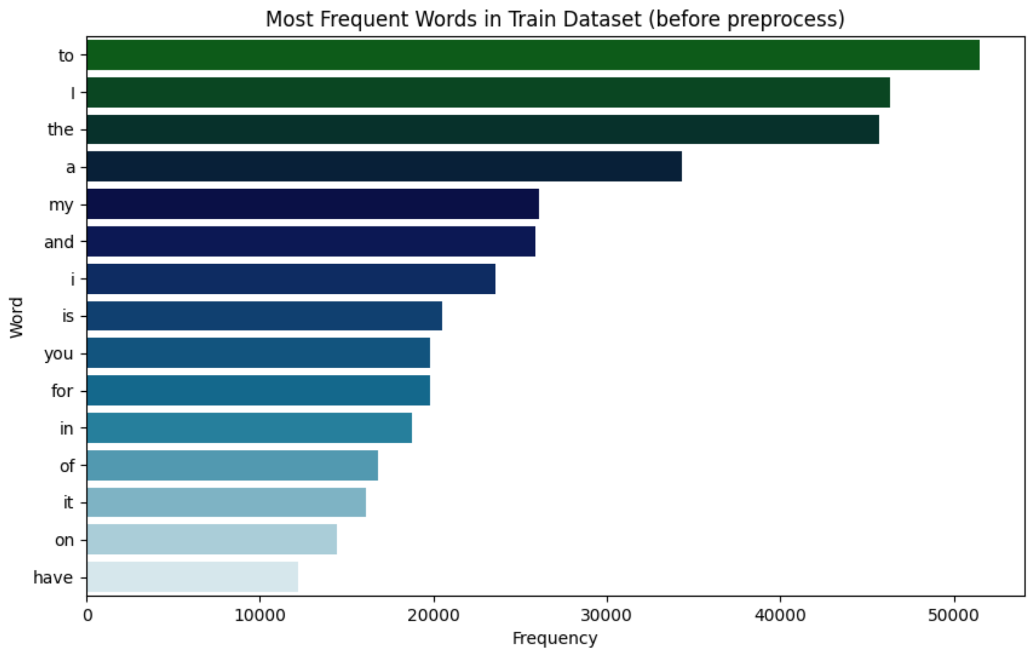


Figure 1: Train Dataset no preprocess, no stopword removal

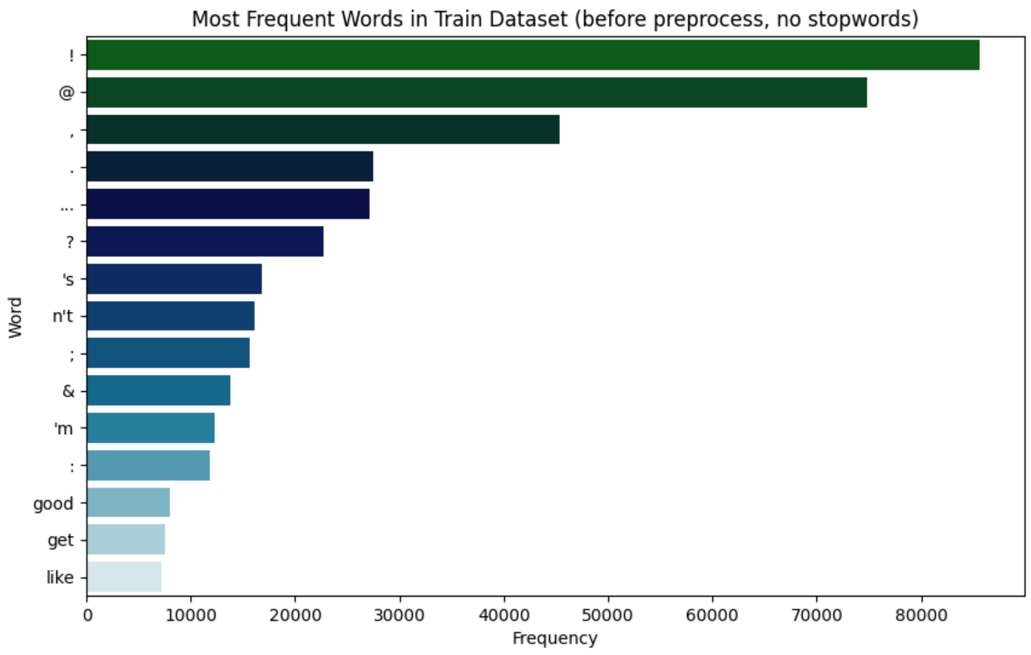


Figure 2: Train Dataset no preprocess, with stopword removal

Same observations for the validation dataset, as well:

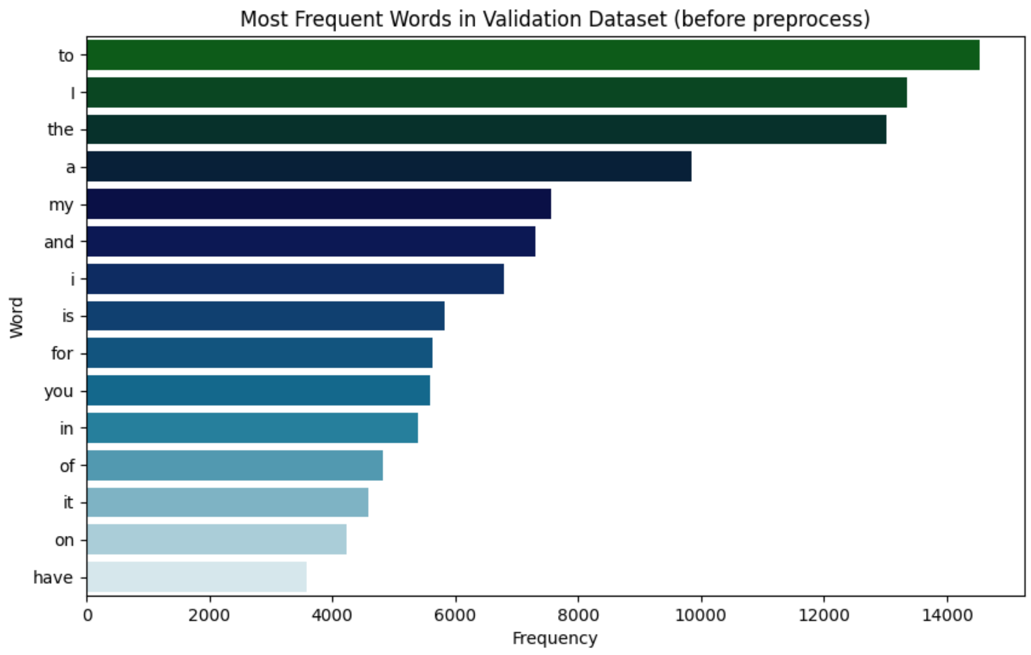


Figure 3: Validation Dataset no preprocess, no stopword removal

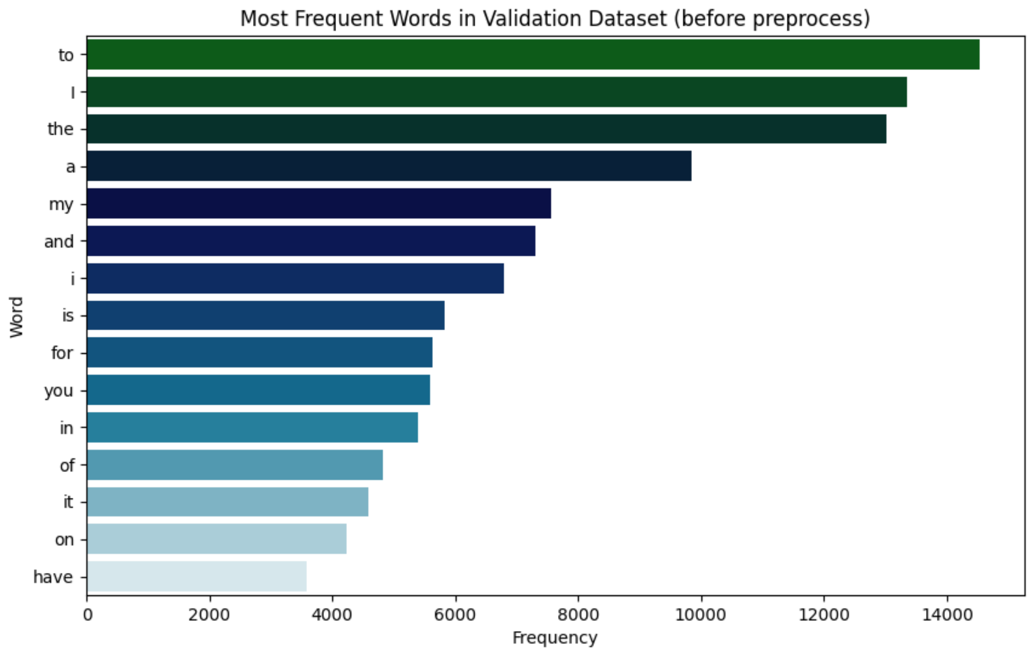


Figure 4: Validation Dataset no preprocess, with stopword removal

Now, it's time to see our preprocessed data for training and validation. Below, it is obvious (especially when stop words are removed and we have a better picture) that the data are less noisy, the useless punctuation is cleared, the meaningful one which contributes to sentiment ("!", "?") is kept. Also, "I" and "i" are treated in the same way. In figure 6, in which we removed stopwords, noisy symbols/words such as "@", ",", ".", "n't" (etc.) are cleared and crucial words for sentiment such as "good", "like", "laugh" (etc.) popped up. These key differnces can be seen, also, in the below validation set visualisations. Although data without stopwords seem more organised, the removal of stopwords led to worse performance so we decided to keep them.

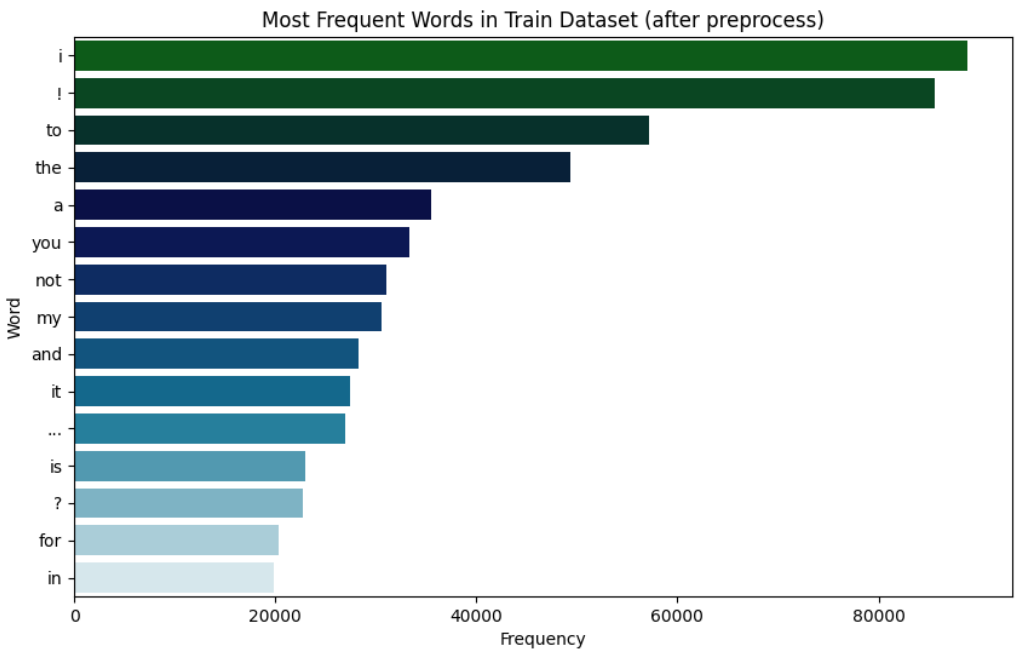


Figure 5: Train Dataset preprocessed, no stopword removal

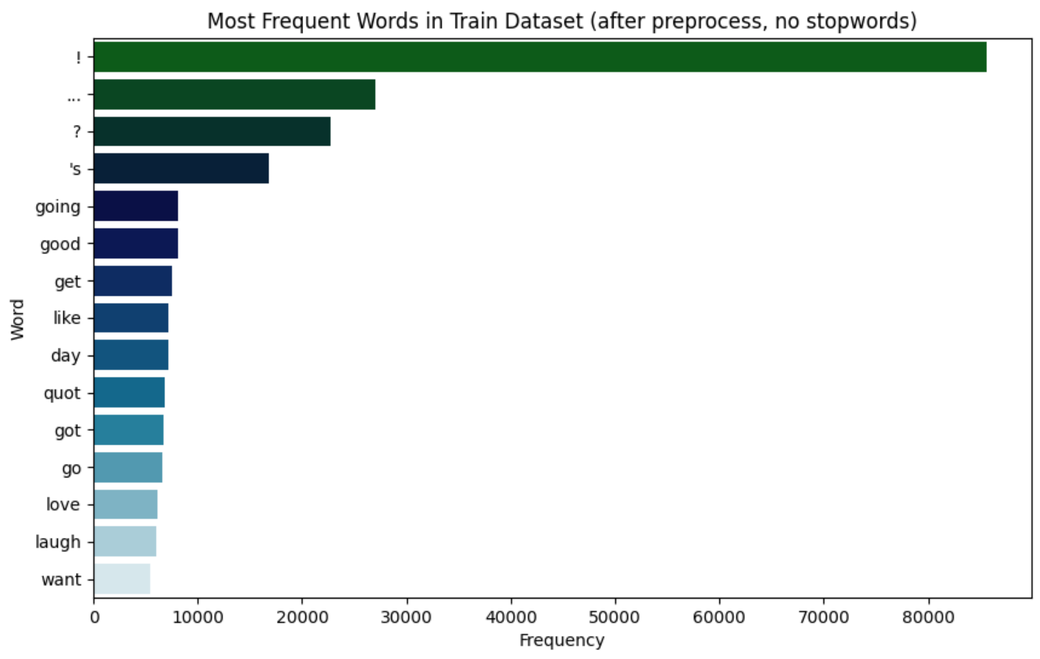


Figure 6: Train Dataset preprocessed, stopword removal

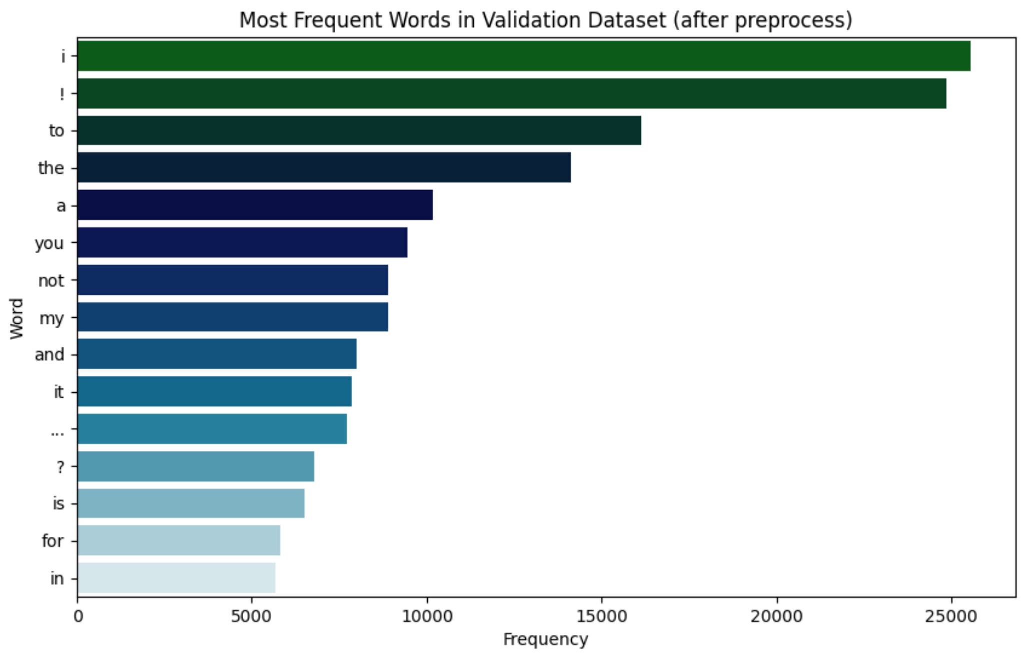


Figure 7: Validation Dataset preprocessed, no stopword removal

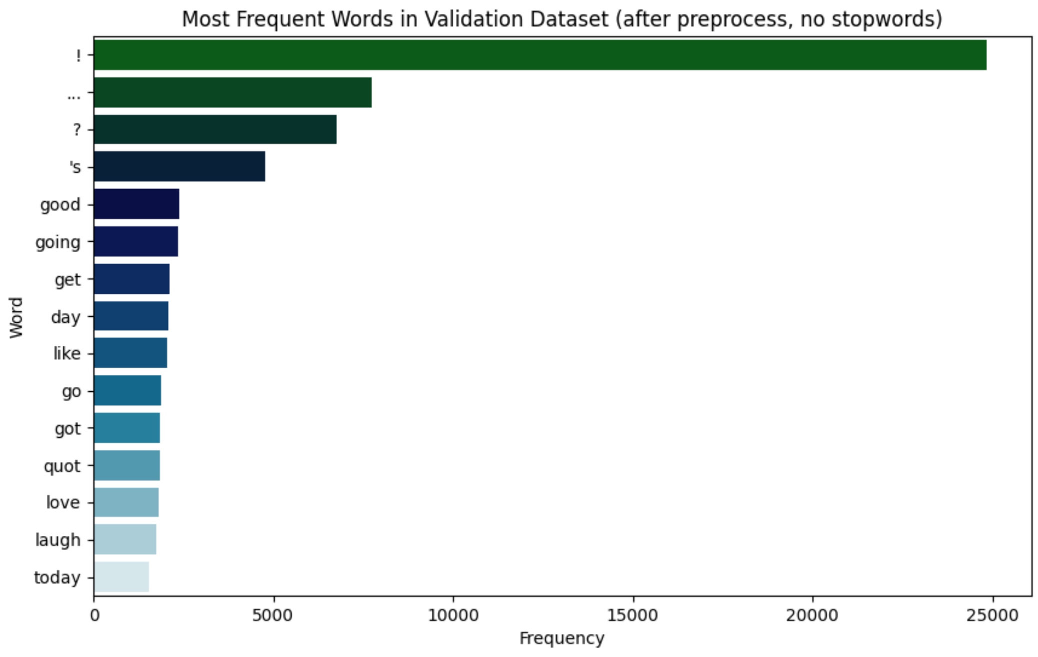


Figure 8: Validation Dataset preprocessed, stopword removal

2.4. Data partitioning for train, test and validation

In this project, as we have said, the datasets were provided pre-partitioned by the instructor into training, validation, and test sets. According to the instructions, these sets were used respectively for training, validation, and testing the model. As a result, no manual partitioning was needed. The main execution block in the code shows this by directly loading the datasets from separate .csv files for each set.

After loading the data, text preprocessing was applied to create a new column called "clean_text" in each dataset. This column contains the processed text data, (we just discussed) which was then used model training, and evaluation.

2.5. Vectorization

For the vectorization step, I used pretrained Word2Vec embeddings from Google News (300 dimensions), as instructed.

But a key question that came up (and it was also addressed on Piazza) was: once we have the embeddings for the words, how do we represent an entire tweet?

The solution was actually simple but clever: for each tweet, I took all the valid word embeddings (ignoring words that didn't exist in the Word2Vec vocabulary) and calculated their average vector. This way, every tweet was summarized into a single 300-dimensional vector. I followed this approach because, although we lose information about the order of the words and the sentence structure, this is the most fitting representation for a simple feedforward neural network like the one we implemented.

In practice, the process was:

tweet → tokens → embeddings → average embedding → input to neural network

I also briefly experimented with using GloVe embeddings instead of Google, but in my case, they didn't improve model performance (accuracy and f1 - the dataset is balanced so these two behave in a similar way). Of course, I believe that if I had spent more time fine-tuning the architecture or adjusting hyperparameters for GloVe, they might have worked better (even better than the Google ones). But since Google Word2Vec gave me solid results early on, I decided to stick with it and focus on improving other parts of the pipeline, like the network's architecture, batch normalization, and tuning with Optuna for optimization.

3. Algorithms and Experiments

3.1. Experiments

In this section, we will analyze some key experiments of all the ones we tried. In other words, the experiments timeline. More about each hyperparameter will be explained in the next section of "Hyperparameter tuning".

NOTE: When we say combinations, we mean different architecture of the NN with same number of hidden layers. The validation accuracy is the accuracy of the combination with the best performance (eg. 256, 64, 32)

Timeline + Plots

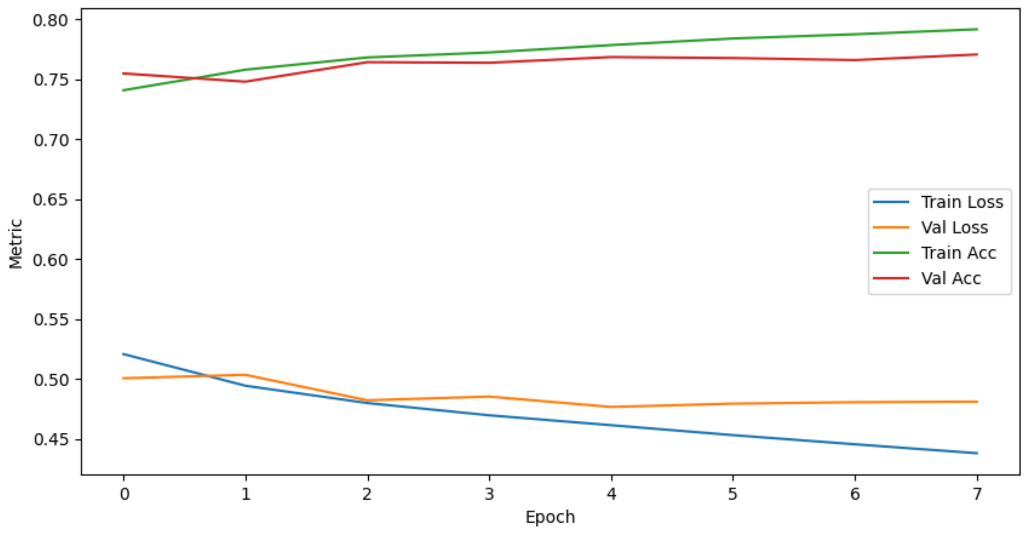
Trial	H.Layers	Dropout	Optimizer	Val Acc
Baseline (no preprocessing)	1	0	Adam	0.579
Preprocessed	1	0	Adam	0.7678
combinations + ReLU	2	0	Adam	0.7408
combinations + ReLU	3	0	Adam	0.7671
4 H layers + ReLU	3	0	Adam	0.0.7589
combinations + SeLU	3	0	Adam	interrupt
combinations + Tanh	3	0	Adam	0.7601
combinations + ReLU	3	0.5	Adam	0.7710
3 layers (+comb) + ReLU + Batch Norm	3	0.3	AdamW	0.7754
3 layers + ReLU + BN + scheduler	3	0.3	AdamW	0.7793
Optuna tuned	3	0.306	AdamW	0.7746

Table 1: Summary of key experiments and validation accuracies.

To start with, we did **baseline experiment without any preprocessing**, as a brute-force approach to gauge how much impact preprocessing would have. The results were poor, with a validation accuracy of only 0.579, and we immediately noticed that the noisy, unprocessed text (full of links, mentions, emojis, etc.) made it difficult for the model to learn meaningful patterns. This performance is expected a bit since without any preprocessing steps and experimenting - only with optimizer and default learning rate, we expect almost random guessing from the model (close to 0.5).

Switching to preprocessed text boosted our accuracy to 0.7678, confirming the importance of cleaning and normalizing the data.

After verifying the importance of preprocessing, it was time to decide **network architecture** - the structure of our model (number of layers, number of neurons per layer). In this section, we could add activation function, as well, but in the early stages we used only ReLU, because we know from class that ReLU is the most used activation function (by far), many libraries provide ReLU-specific optimizations; therefore, if speed is your priority, ReLU might still be the best choice. Later, as you see in the table of trials, we tried different ones. SeLU, which based on lecture slides, considered to be the best in general, was keyboard interrupted to Epoch 12 since it was time-consuming and the results weren't better. The way we decided the model architecture was to **keep each time the last 2 layers (eg. 64,32) and comparing the results with the full model (128, 64, 32)**. With this method, and **always experimenting and retuning the learning rate**, according to slides, we've concluded to (300, 256, 128, 64, 2). More about the combinations in "Hyper-Parameter Tuning" section. In this stage, the learning curve looks like this:



The generalization gap is obvious, but we haven't used any techniques to reduce overfit. Now, it's time for **dropout**. We tried different combinations with dropout as far its application to the network structure is concerned, of course excluding the output layer. Interesting was the relation of **learning rate and dropout**, so as to push our model for better performance, but have acceptable generalization gap. Below, you can see the comparison of different learning rates with dropout enabled. With less learning rate, the model is better for **steadier** improvements (we could push a little more the learning rate later).

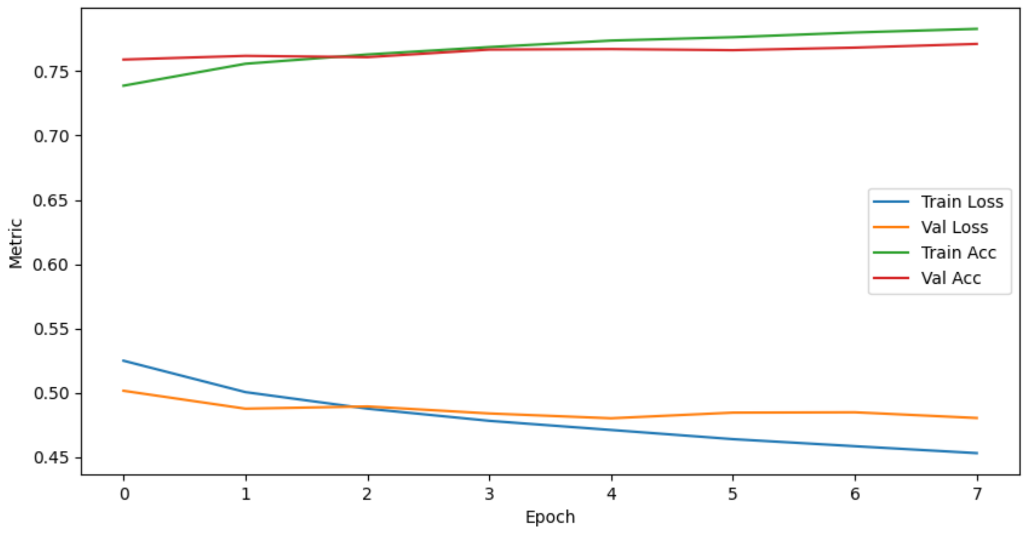


Figure 9: Learning Rate: 0.0005, Dropout 0.5

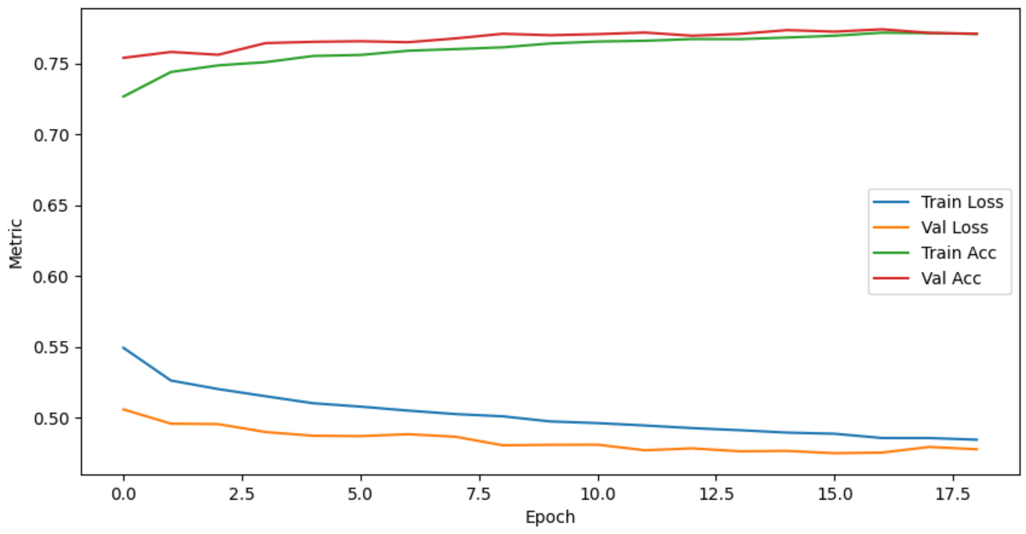


Figure 10: Learning Rate: 0.001, Dropout 0.5

Once dropout showed promising results, we further explored optimization techniques to improve the model's robustness. We added **batch normalization**, which, based on the lecture slides, can accelerate convergence and stabilize training. Although it increases per-epoch time slightly, the benefit was clear: it made learning smoother, and validation accuracy climbed to 0.7754. The smoothness is visible below:

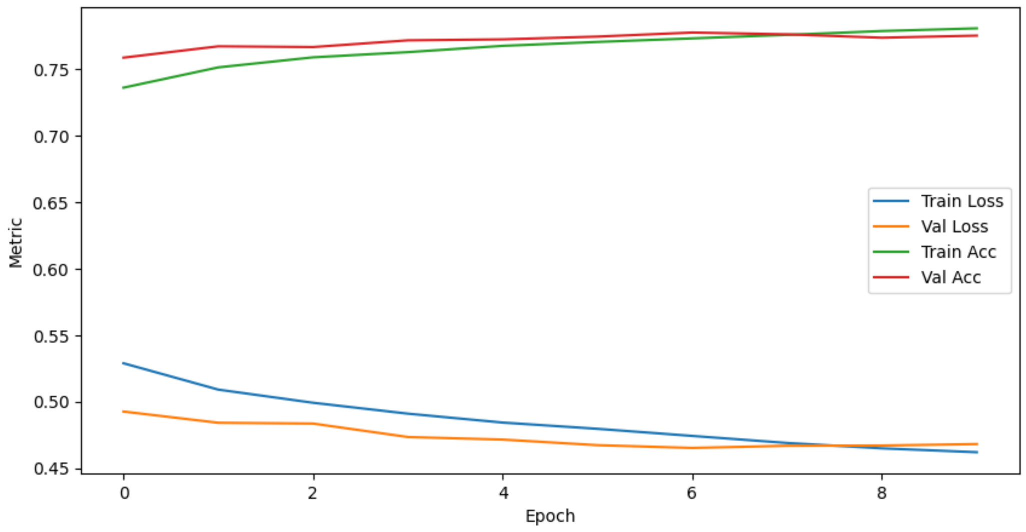


Figure 11: Addition of batch norm

To further fine-tune, we introduced a **learning rate scheduler** (ReduceLROnPlateau). As discussed in class, schedulers help slow down learning when validation loss stops improving, reducing overfitting and achieving a better balance across classes. Adding the scheduler pushed validation accuracy to **0.7793**, making it our best result. In the learning curve above, we notice that adding the scheduler allowed the model to train for more epochs without significant overfitting. this indicates that the scheduler slowed learning at the right time, letting the model settle into a more balanced state. Compared to batch normalization alone, the scheduler extends training without losing control over validation metrics, leading to our best generalization performance.

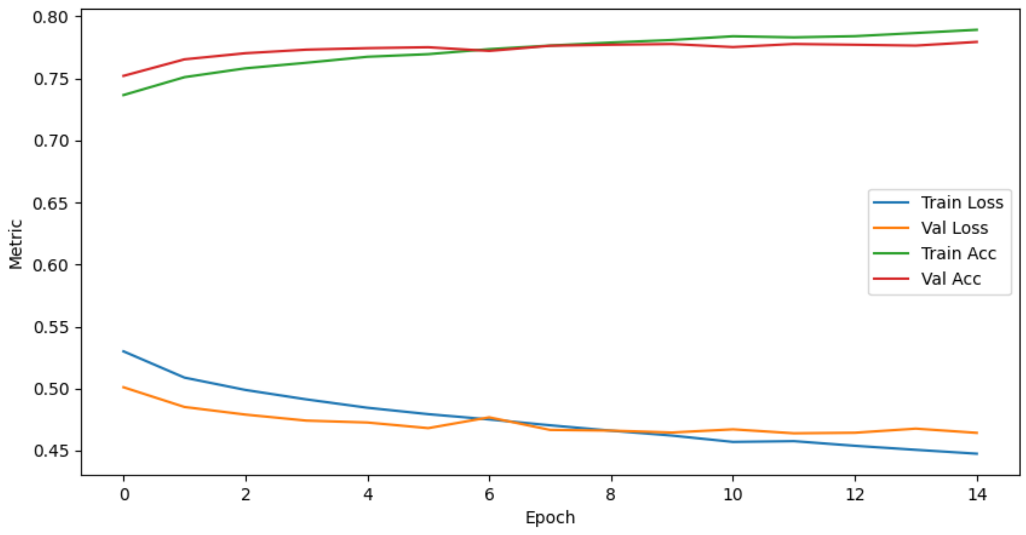


Figure 12: Addition of scheduler

At this point, manual tuning had plateaued: improvements were smaller, and each experiment required multiple epochs to assess. Therefore, we decided to adopt an automated hyperparameter search. so time for **Optuna**, to explore a broader range of learning rates, dropout values, and optimizers systematically. Later, why we'll see optuna results.

3.2. Hyper-parameter tuning and Optimization Techniques

For DNNs, there is a wide variety of Hyper-Paramater Tuning and optimization techniques, and we tried almost all of them:

1. **Learning rate:** As emphasized in class, the learning rate is arguably the most critical hyperparameter. Initially, we used the default 0.001 for Adam, but after early experiments, we reduced it progressively (e.g., 0.0005, 0.0001) to achieve steadier improvements. We observed that lower learning rates improved validation stability, especially in combination with dropout and batch normalization. Interesting was its relation with batch size as well, explained 5 parameters under. Also, the importance of learning rate was the reason that Learning Rate was one of the hyperparameters used in Optuna.

2. **Optimizer:** We started with Adam, due to its adaptive learning rate and good defaults. Later, we switched to AdamW. We found AdamW slightly outperformed Adam in validation accuracy, especially when combined with a scheduler.

3. **Number of hidden layers:** We progressively increased layers from 1 to 4. Following the lecture's suggestion to "start bigger and regularize," we initially tested deeper architectures, then monitored for overfitting. We observed that increasing beyond 3 layers gave worse returns or higher overfitting without added techniques.

4. **Number of neurons per layer:** Similar to layers, we experimented with different neuron counts per layer (e.g., 256, 128, 64). We analysed the way we worked for this model structure in the timeline.

5. **Activation functions:** We mostly used ReLU, aligning with the lecture recommendation that it's the fastest and most optimized in libraries. We tested SeLU and Tanh, but they either underperformed or increased training time without meaningful gains. Additionally, for the output layer, we experimented between a single output neuron with sigmoid + BCELoss versus 2 output neurons with CrossEntropyLoss. We found no practical difference for our task, so we kept `D_out=2` and CrossEntropyLoss.

6. **Batch size:** We tested batch sizes of 32, 64, and 128. We followed the strategy: **try to use a large batch size, using learning rate warmup, and if training is unstable or the final performance is disappointing, then try using a small batch size instead**. 64 was the best for all!

7. **Scheduler:** We used a ReduceLROnPlateau scheduler to reduce the learning rate when validation loss stopped improving. This slowed training adaptively, making the model working for more epochs without being interrupted by early stopping.

8. **Batch normalization:** We added batch normalization after each hidden layer to accelerate convergence and stabilize training, following lecture guidance. Although it increased computation per epoch, it reduced the total number of epochs needed for convergence. We noticed smoother learning curves and a reduced generalization gap, with validation accuracy improving. The model became more stable, especially in combination with dropout.

9. **Dropout:** We added dropout progressively (tested rates from 0.1 to 0.5 through optuna as well), applying it to the hidden layers but excluding the output layer, as recommended in class. Dropout played a vital role at improving generalization, narrowing the generalization gap.

10. **Early stopping:** To prevent unnecessary overfitting a, we implemented early stopping with `patience=3`, meaning training would stop if validation loss did not improve for 3 consecutive epochs. This allowed us to avoid overfitting while still giving the model enough room to escape small plateaus. We found early stopping especially useful when experimenting with deeper networks or slower learning rates, preventing wasted epochs once performance had plateaued.

For optimising the selection of hyperparameters we used Optuna. We did 30 trials with (0.1 to 0.5 dropout, 1e-4 to 1e-2 learning rate and Adam, AdamW optimizers. The result was:

```
Best trial params: {'learning_rate': 0.00014634185967178962,
'dropout_rate': 0.3064771962749029, 'optimizer': 'AdamW'}
Best validation loss: 0.4599183649387
```

3.3. Evaluation

According to slides, for a balanced binary classification task like this, confusion matrix may be the best metric. Except for it, we used ROC Curve and of course learning curves as we have seen.

3.3.1. Confusion Matrix. Below are the confusion matrices of three key models: baseline, best manually tuned model (with scheduler), and Optuna-tuned model.

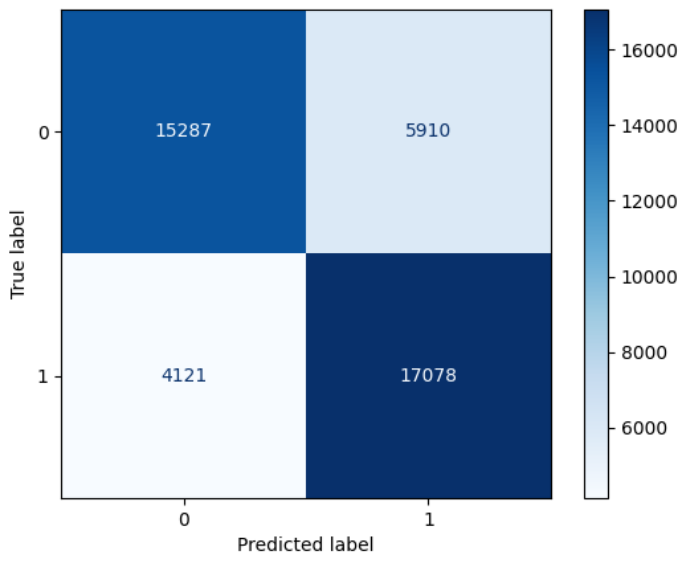


Figure 13: Baseline confusion matrix (no preprocessing)

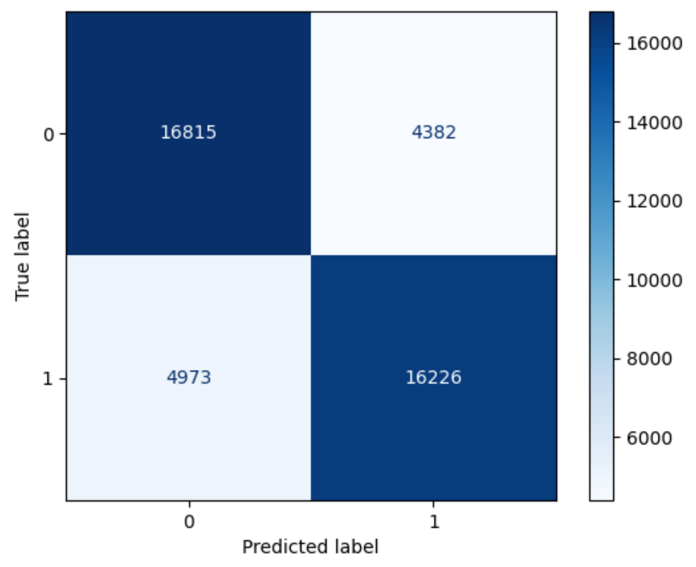


Figure 14: Best manual model confusion matrix (after scheduler)

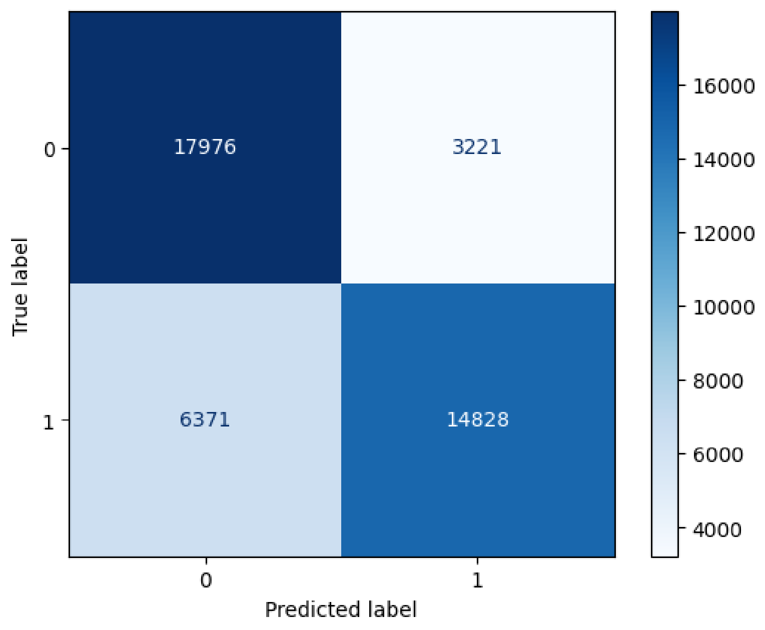


Figure 15: Final Optuna-tuned model confusion matrix

The baseline model shows poor separation between classes, with many misclassifications. The manually tuned model (with batch normalization, dropout, and scheduler) achieves a well-balanced confusion matrix, showing similar false positives and false negatives across classes.

Interestingly, the Optuna-tuned model—while minimizing validation loss—resulted in a less balanced confusion matrix, favoring the majority class slightly more. This highlights an important lesson: **optimizing for validation loss does not necessarily optimize for confusion matrix balance**. Despite this, I submitted the Optuna model in Kaggle as the final one, since it followed a systematic hyperparameter search and achieved stable validation loss, but looking back, I think the manually tuned model was probably more balanced overall as a classifier.

In summary, this evaluation shows how different metrics may lead to different "best" models depending on the goal (minimizing loss vs balancing class errors), and it's based on the tasks needs.

3.3.2. ROC Curve. Roc Curves of the Optuna and the manual model were almost identical. 0.86 AUC Score is a solid number and shows the ability of the model to distinguish between positive and negative classes.

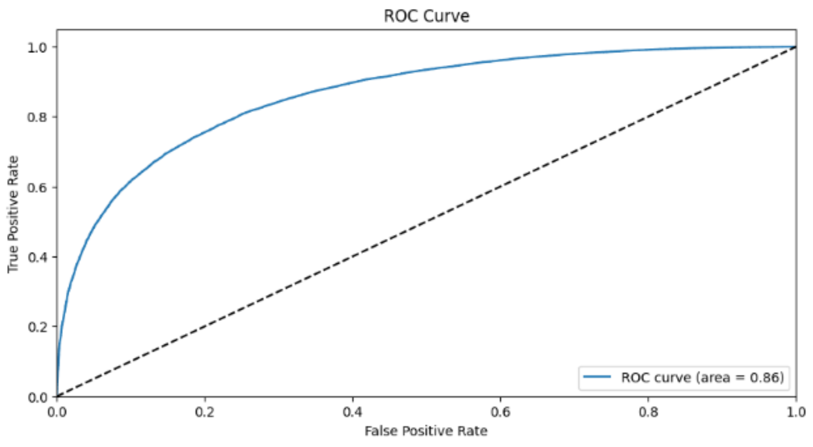


Figure 16: Final ROC Curve

3.3.3. Learning Curve. Except for the learning curve of the final manual model we saw, below we can see the Optuna Learning Curve minimizing validation loss.

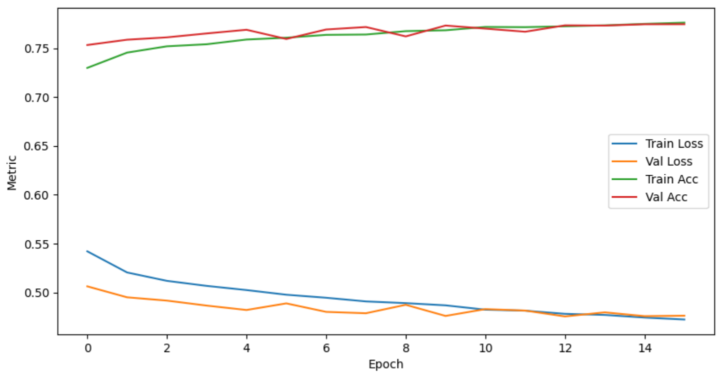


Figure 17: Optuna Learning curve

4. Overall Analysis

4.1. Comparison with the first project

In the first project, where we used a TF-IDF representation combined with a Logistic Regression classifier, I achieved a validation accuracy of **0.804**. In this second project, the performance was significantly lower (not only in accuracy but in F1-Score as well). The reason behind that I think is the more hyperparameters that NNs introduce. Also, if I spent more time with different pretrained word embeddings, maybe the performance could be higher.

4.2. Final thoughts

I found the project interestingly challenging, because of the variety of hyperparameters and combinations with DNNs. For first time into Deep Learning, I tried to apply lecture theory in code and I believe it was a vital step my knowledge in Data Science field.

5. Bibliography

References

[1] Manolis Koumparakis. Artificial intelligence ii, deep learning for natural language processing, advanced artificial intelligence. https://cgi.di.uoa.gr/~ys19/#homework, 2025.

[2] Myrto Tsokanaridou. Tutorials for homework 2. 2025.

[1] [2]