UNIVERSITY OF ATHENS
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

Deep Learning for NLP 2nd Assignment

Student name: *Dimokritos Kolitsos*
sdi: sdi1900085

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

Contents

1. Abstract

The task involves developing a sentiment classifier for tweets related to the Greek general elections. To tackle this task I will be implementing a deep neural network model. The end goal is for the model to be capable of classifying tweets into three sentiment categories: POSITIVE, NEUTRAL, and NEGATIVE. To achieve this, the process will be to first apply steps like data preprocessing, defining a neural network architecture, training the model and evaluating its performance but also experimenting with various techniques and methods to improve the model such as lemmatization, stemming, hyperparameter tuning etc. The point is to experiment and improve the model's performance as much as possible.

2. Data processing and analysis

2.1. Pre-processing

Since the dataset is the same one we used in the previous assignment, I know that there aren't any missing (NaN) or odd(for example not a label in the labels column) data in the dataset so there is no need to do something about missing or odd data.

Since I have already implemented several good pre-processing techniques that I'm happy with the results of them for the dataset, I will keep them for this assignment aswell. Some of them include removing special characters, URLs and usernames using regular expressions. This will help the model focus on words that can have an indication of sentiment and context and not on irrelevant words and characters. Before moving on I also converted sentiment labels to numbers which will also help in the training of the machine learning model.

The processed text was then tokenized using the Natural Language Toolkit library. Tokenization is necessary as it helps the model understand the structure of the text and allows for techniques like lowercasing, stop words removal, lemmatization etc. to be applied easily.

All tokens were then converted to lowercase characters to maintain consistency, limit extra characters from needing to be included and understood and prevent possible confusion from the model in general.

I then imported a txt file with Greek stopwords which was created by fellow classmate Giorgos - Marios Patelis which he kindly shared in discord for us to use and was uploaded in kaggle by another fellow classmate. I used the file to remove stopwords from the tweet's text since I saw in the analysis that irrelevant words such as articles (ο, η το) had the highest count in the tweets. This aswell significantly helps the model focus on words that have meaning and can hint sentiment.

Afterwards I also tried applying lemmatization to the text to see if it will improve the specifier's training. I contemplated whether to try stemming, lemmatization or both but since spaCy already has a Greek language model it made my decision easier. Lemmatization involves reducing words to their base or root form, contributing to a more refined and standardized representation of the text, meaning that the the model can focus more on the essence of the words and as with the previous steps, it helps leaving less margin for confusion.

I was not happy with the results and although the lemmatization gave my previous assignment's logistic regression model a better performance, this time applying the

technique dropped the neural net model's performance significantly overall. Also with spaCy's Greek lemmatization model the processed text had little to no visible changes and the words that were changed, made little to no sense (their format etc). To add to that Giorgos - Marios Patelis shared with us yet another helpful tool, a working Greek Stemmer, so I decided to use that instead. Although this initially didn't help with the performance (the results were almost similar to the ones with lemmatization and definitely lower than the ones I got without using any), the processed text looks visibly much better and more organized and the data analysis shows that, to me it looked that it would make so I decided to keep this step and work with the Neural net and the word2vec implementation to improve the performance of the model.

2.2. Word Embeddings

In addition to the pre-processing phase, it's crucial to highlight use of Word2Vec word embeddings to transform the processed tweet text into continuous vector representations suitable for input into a neural network model.

At first I used the keras tokenizer to simply convert the tweets text to sequences of word indices (tokenizer.texts_to_sequences) and then made sure that they have the same length by padding them. Although this worked and I could then convert the data to PyTorch tensors suitable to train my model, I was not happy with the training results and my model's performance overall so I decided to use a pretrained model.

The first one I tried was FastText's Greek pretrained model and although the results looked promising (printing the word embeddings and the tensors) I kept running out of run when I tried to train my neural net. Even after messing around with various different ways to implement the word2vec transformation with the word2vec model, different dimensions for it and even different hyperparameters for my model, I kept running into errors or out of memory in kaggle so after losing a lot of time on this I decided to try a different model.

To avoid future confusion I stuck with gensim library. I found a model at the NLPL word embeddings repository which it was specified that it worked with the Greek language. More specifically the pre-trained Greek Word2Vec model is trained on the Greek CoNLL17 corpus and provides vector representations of words in a 100-dimensional space. After messsing around with that model I found out that most of the tokens in my data was out of vocabulary data and I couldn't get it to work without exceeding memory limits or getting multiple errors in the training process anyway so I decided not to use this model either.

To end my troubles and stop losing countless days on not even the main focus of this assignment I decided not to use a pretrained model and just train my own using gensim's Word2Vec function. To do that, I first combined all the preprocessed text from all train,validation and test sets to one to use for the training of my model to have better training results. I then made sure that the input requirements of gensim's Word2Vec model were met by spliting the joined tokens of the preprecessed text. Finally I trained the model on what semeed to be good parameters, I saved it so I wouldn't need to run all that again if I wanted to reuse it in the same session and finally I created a function and used it to convert my dataset's preprocessed texts to word embeddings.

2.3. Analysis

For the analysis of the data I implemented a simple word cloud graph to get a rough idea of how the dataset looks and also which are the dominant words in the dataset. Otherwise I already had a rough idea already from printing the dataset each time to check if the preprocessing techniques I implemented were working correctly and also see the difference they made each time.

2.4. Data partitioning for train, test and validation

I found out that repartitioning the dataset and trying out different configurations from the one already given didn't yield any significant differences in the results so I left the dataset split as given from the beginning

3. Algorithms and Experiments

3.1. Experiments

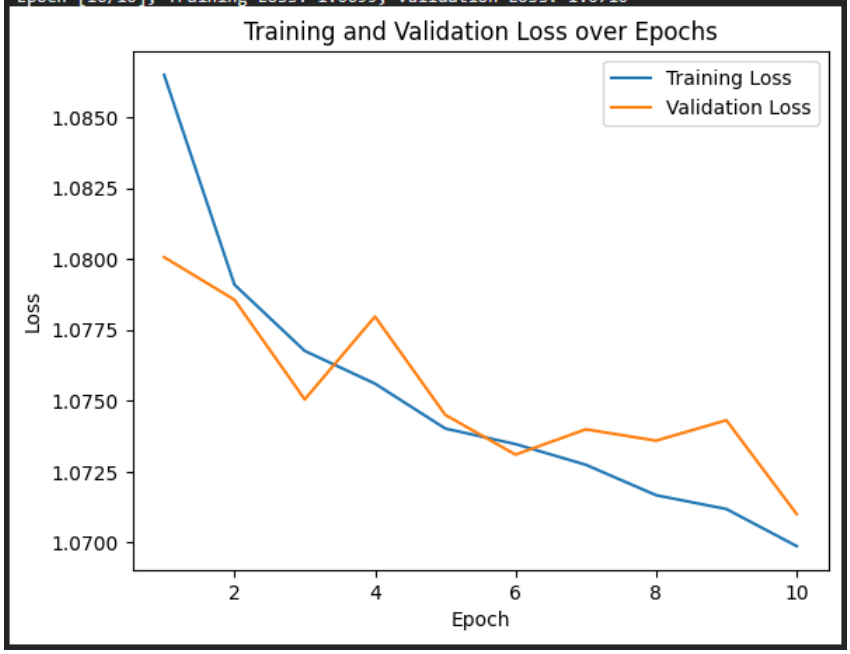
At first, after some basic data exploration (basically realizing the dataset was the same one as the previous assignment..) and after setting up a basic word2vec implementation (using keras sequences) and a basic model without worrying too much about details such as adjusting hyperparameters etc. I decided to run a training session without any preprocessing (except for transforming the data to word embeddings which is necessary) to see how a deep neural net model performs with this particular dataset unprocessed and also have a base point to compare my results and see how each new technique and feature I implement impacts the model's performance. I knew from the start that with this particular dataset I couldn't expect very good results and that also the results most of the time will not really be representative of the quality of the implementation and cannot really be used as a measure point and that was proven because, at least for me, the results of the neural net model with the raw unprocessed text were worse than the ones my logistic regression model had achieved when I tried something similar. The loss over 10 epochs was averaging at about 1.1 - 1.09 and on the validation set I got somewhere around 0.35 accuracy.

At this point I started experimenting with the various preprocessing methods that most of them were mentioned in the previous section of the report. I tried using all the methods I used for the previous assignment and then some and I occasionally run a training and evaluation session to check what impact the data preprocessing had on the model's performance and, to my surprise, the results were mostly worse by the ones I got without preprocessing. Even when I implemented teqniques like stop words removal and stemming that, to me at least, seemed like a significant improvement of the processed text's quality, there were significant drops in the results aswell (around 1.1+ loss and 0.33 accuracy).

The next stage of experiments were about the word2vec implementation. I thought that the reason the results were worse when I was improving the dataset with the various preprocessing teqniques was because of it's transformation to word embeddings and so I wanted to try a much different implementation. Here I lost countless time messing around with pretrained models (that I thought that would yield the better results I wanted since I was looking for models that would presumably work well with Greek). After having tried a few and not having much success to even get a result I scrapped the idea of using a pre trained model. The route I took was to train a

simple word2vec model myself using all of the dataset's text combined with genism's Word2Vec function. I remodified the rest of the code with the model architecture and the training once again to suit my implementation and it worked giving me much better results than all the ones I had achieved before:

```
Epoch [1/10], Training Loss: 1.0865, Validation Loss: 1.0801
Epoch [2/10], Training Loss: 1.0791, Validation Loss: 1.0786
Epoch [3/10], Training Loss: 1.0768, Validation Loss: 1.0750
Epoch [4/10], Training Loss: 1.0756, Validation Loss: 1.0780
Epoch [5/10], Training Loss: 1.0740, Validation Loss: 1.0745
Epoch [6/10], Training Loss: 1.0735, Validation Loss: 1.0731
Epoch [7/10], Training Loss: 1.0727, Validation Loss: 1.0740
Epoch [8/10], Training Loss: 1.0717, Validation Loss: 1.0736
Epoch [9/10], Training Loss: 1.0712, Validation Loss: 1.0743
Epoch [10/10], Training Loss: 1.0699, Validation Loss: 1.0710
```



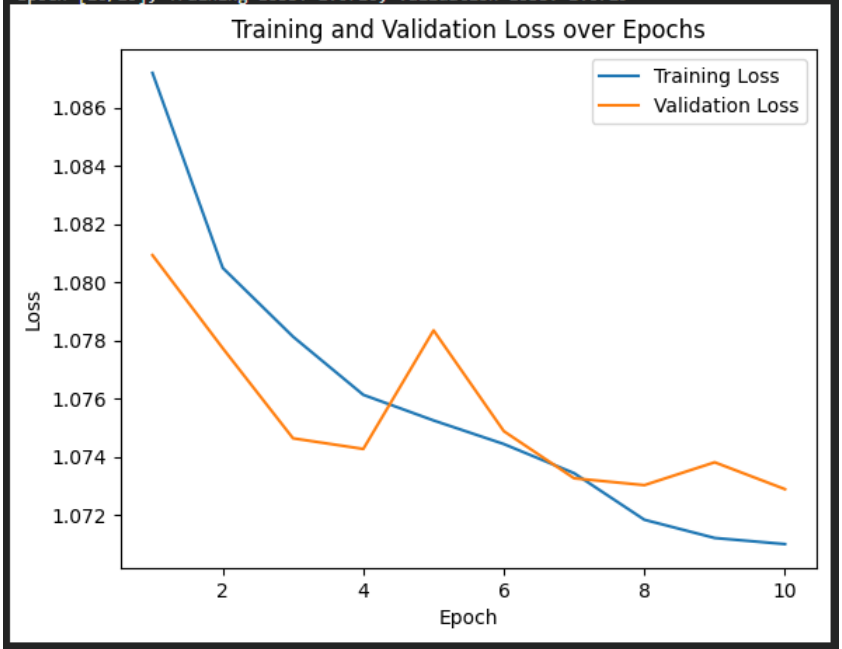
Validation Precision: 0.4115, Recall: 0.4002, F1-score: 0.3811

Now that my model looks much better (at least for this particular dataset) I know that the data I 'feed' to it is fine so I shift my focus on to the model. The first thing I implemented is hyperparameter tuning which, from the previous assignment, I know that it can significantly improve the model's performance. I implemented a grid search technique which let me test multiple different hyperparameter combinations in a short time. This showed me my model's potential on precision, recall and f1-score (I used the evaluation function I made to test the hyperparameters with the validation set) and also it found the values that yield the best results for my model:

```
Validation Precision: 0.4051, Recall: 0.3964, F1-score: 0.3862
Validation Precision: 0.3991, Recall: 0.3903, F1-score: 0.3791
Validation Precision: 0.4036, Recall: 0.3989, F1-score: 0.3872
Validation Precision: 0.4271, Recall: 0.4056, F1-score: 0.3794
Validation Precision: 0.4032, Recall: 0.4006, F1-score: 0.3938
Validation Precision: 0.3971, Recall: 0.3966, F1-score: 0.3910
Validation Precision: 0.4324, Recall: 0.4088, F1-score: 0.3805
Validation Precision: 0.4117, Recall: 0.3985, F1-score: 0.3757
Validation Precision: 0.4141, Recall: 0.3943, F1-score: 0.3651
Validation Precision: 0.4607, Recall: 0.3911, F1-score: 0.3392
Validation Precision: 0.4625, Recall: 0.3836, F1-score: 0.3076
Validation Precision: 0.4518, Recall: 0.3740, F1-score: 0.2906
Validation Precision: 0.4458, Recall: 0.3891, F1-score: 0.3342
Validation Precision: 0.4391, Recall: 0.4039, F1-score: 0.3686
Validation Precision: 0.4837, Recall: 0.3825, F1-score: 0.3257
Validation Precision: 0.4198, Recall: 0.3878, F1-score: 0.3528
Validation Precision: 0.4726, Recall: 0.3867, F1-score: 0.3165
Validation Precision: 0.4254, Recall: 0.3958, F1-score: 0.3621
Validation Precision: 0.7778, Recall: 0.3333, F1-score: 0.1667
Validation Precision: 0.7778, Recall: 0.3333, F1-score: 0.1667
Validation Precision: 0.7778, Recall: 0.3333, F1-score: 0.1667
Validation Precision: 0.7778, Recall: 0.3333, F1-score: 0.1667
Validation Precision: 0.7778, Recall: 0.3333, F1-score: 0.1667
Validation Precision: 0.7778, Recall: 0.3333, F1-score: 0.1667
Validation Precision: 0.7778, Recall: 0.3333, F1-score: 0.1667
Validation Precision: 0.7778, Recall: 0.3333, F1-score: 0.1667
Best Hyperparameters: {'Learning Rate': 0.001, 'Hidden Size': 128, 'Batch Size': 64}
```

I played around with various different hyperparameters and values but since the results were different between session to session and even between run to run in a single session even when everything else was the same, changing hyperparameters was also not the most consistent thing in terms of results so it was a little bit difficult to keep track of everything and record the best result I got (even because other stuff was also changing/improving while I was hunting for the best hyperparameters). At last these are the best (most consistent) results I could achieve with my model just by playing with the hyperparameters:

```
Epoch [1/10], Training Loss: 1.0872, Validation Loss: 1.0809
Epoch [2/10], Training Loss: 1.0805, Validation Loss: 1.0777
Epoch [3/10], Training Loss: 1.0781, Validation Loss: 1.0746
Epoch [4/10], Training Loss: 1.0761, Validation Loss: 1.0743
Epoch [5/10], Training Loss: 1.0752, Validation Loss: 1.0783
Epoch [6/10], Training Loss: 1.0744, Validation Loss: 1.0749
Epoch [7/10], Training Loss: 1.0734, Validation Loss: 1.0733
Epoch [8/10], Training Loss: 1.0718, Validation Loss: 1.0730
Epoch [9/10], Training Loss: 1.0712, Validation Loss: 1.0738
Epoch [10/10], Training Loss: 1.0710, Validation Loss: 1.0729
```




```
Validation Precision: 0.4160, Recall: 0.3955, F1-score: 0.3654
```

3.2. Hyper-parameter tuning

The hyperparameter tuning process involves playing with various parameter combinations to optimize the neural net model's performance. After finding some parameters and possible combinations for each and playing with them I decided to do a grid search implementation which tests all possible combinations of the parameters you provide it and gives you the ones that yield the best results in the particular instance of the model depending on the training and the data it trains on. I noticed that there wasn't a standard set of parameters that would always be better and that grid search would result with different parameters each time depending the data. Also having a lot of different parameters takes a lot of time and can sometimes 'break' the model and certainly not yield consistent and desireable resaults so after multiple experiments with a dozen of different parameters and values I only kept the combination of parameters that gave the best and most consistent results.

With my experimenting I came to the conclusion that changing the parameters won't usually change the performance of the model in a drastic way or nearly as much as it can with trying different techniques but on the other hand it is a consistent way to gain the maximum performance out of the particular model instance and in the end it makes a significant difference keeping in mind that with the nature of the particular dataset possitive changes might sometimes not even improve but have the oposite effect on the performanse of the model. Overall playing with the parameters helped reduce the apparent overfitting and enhance the model's generalizability as the results from the training and validation dataset are now much closer indicating that the model is as good at analyzing unknown tweets that it is at analyzing tweets that it has trained on.

3.3. Evaluation

For the evaluation of the model's performance I plotted learning curves and run evaluated it's performance with the validation set. The results are all showcased in section 3.1

<Provide and comment diagrams and curves>

3.3.1. ROC curve.

3.3.2. Learning Curve.

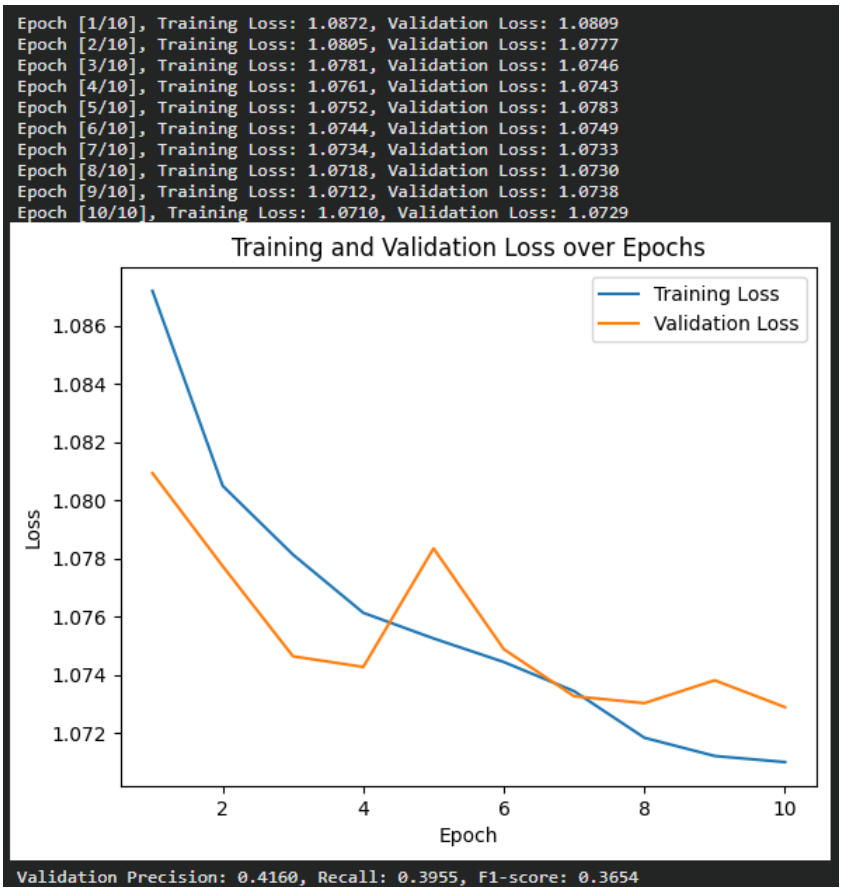
3.3.3. Confusion matrix.

4. Results and Overall Analysis

4.1. Results Analysis

This section was pretty much covered by section 3.1

4.1.1. Best trial.



4.2. Comparison with the first project

In comparison to the first assignment the results seemed more over the place. By that I mean that at first they were much lower than the logistic regression model which was weird but then they got up to par or even a bit better overall which is what I expected due to deep neural networks being a much more powerful and sophisticated method to implement a sentiment classifier than the "simpler" logistic regression. Although I believe that the nature of the dataset did not do the neural network model justice

4.3. Comparison with the second project

<Use only for projects 3,4>
<Comment the results. Why the results are better/worse/the same?>

4.4. Comparison with the third project

<Use only for project 4>
<Comment the results. Why the results are better/worse/the same?>

5. Bibliography

References

[1] Data Magic (by Sunny Kusawa). How to use kaggle notebook | kaggle tutorial | machine learning | data magic.

[2] codebasics. What is word2vec? a simple explanation | deep learning tutorial 41 (tensorflow, keras & python).

[3] codebasics. Word2vec part 2 | implement word2vec in gensim | | deep learning tutorial 42 with python.

[4] DigitalSreeni. 189 - hyperparameter tuning for dropout, neurons, batch size, epochs, and weight constraint.

[5] Pradip Nichite. Build text classification model using word2vec | gensim | nlp | python | code.

[6] utsav aggarwal. Text preprocessing | tokenization | cleaning | stemming | stopwords | lemmatization.

[1] How to use Kaggle Notebook | Kaggle Tutorial | Machine Learning | Data Magic

[6] Text Preprocessing | tokenization | cleaning | stemming | stopwords | lemmatization

[2] What is Word2Vec? A Simple Explanation | Deep Learning Tutorial 41 (Tensorflow, Keras & Python)

[3] Word2Vec Part 2 | Implement word2vec in gensim | | Deep Learning Tutorial 42 with Python

[5] Build Text Classification Model using Word2Vec | Gensim | NLP | Python | Code

[4] 189 - Hyperparameter tuning for dropout, neurons, batch size, epochs, and weight constraint

StackOverflow What is the best way to remove accents (normalize) in a Python unicode string?

Greek Stemmer

FastText pretrainned models

NLPL word embeddings repository