UNIVERSITY OF ATHENS
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

Deep Learning for NLP

Student name: *<Ergina Dimitraina>*
sdi: <sdi2100031>

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

Contents

1. Abstract

<Briefly describe what's the task and how you will tackle it.>
The objective of this task was to develop a sentiment classifier using only the Logistic Regression model and the TF-IDF (Term Frequency-Inverse Document Frequency) method. We were provided with three datasets (test, train, and validation), each containing sentences that we needed to classify as either positive (1) or negative (0) based on their sentiment.

To begin, I studied the research papers on text classification and logistic regression provided by our professor to fully understand the assignment. Next, I focused on processing and analyzing the data. The detailed methodology and results will be discussed in the following sections.

After preprocessing the data, I implemented the TF-IDF transformation and applied the Logistic Regression model. Once the initial implementation was successful, I worked on improving my code to achieve a balance between a clean dataset and optimal accuracy. Finally, I generated various plots to visualize the results and evaluate the model's performance more effectively.

2. Data processing and analysis

2.1. Pre-processing

<In this step, you should describe and comment on the methods that you used for data cleaning and pre-processing. In ML and AI applications, this is the initial and really important step.

For example some data cleaning techniques are: Dropping small sentences; Remove links; Remove list symbols and other uni-codes.>
Pre-processing is a crucial step in ML and AI applications. It involves cleaning the dataset by removing non-essential elements, unusual patterns, and components that may hinder classification performance or lead to poor results.
In this project I tested all the following methods:
Pre-processing Methods Used:
Lowercasing
Removing special characters, numbers, URLs and extra spaces:
Tokenization
Lemmatization
Removing stopwords
Negation handling
Correcting spelling mistakes and slang words

I observed that using all the methods together did not improve the model's accuracy. In fact, it decreased it to 73%. So, I decided to test the methods one by one to achieve the best possible accuracy.

First, I applied lowercasing, which, as expected, helped. Then, I added the removal of special characters, URLs, numbers, and extra spacing to clean the dataset further. The most challenging part of the process was removing stopwords. Initially, I tried removing the default stopwords from the NLTK library, which was somewhat helpful but not very effective. Then, I combined the NLTK stopwords with those from the

sklearn library, which improved the accuracy to 75%. But again this was not the best option, because some words that were adding to the sentiment classification were removed, such as others pretty neutral words were still in the data set. To identify the best stopwords to remove for this specific dataset, I generated a diagram (Diagram 3) showing the most frequently used words before preprocessing. Based on this, I created a custom stopwords list tailored for this dataset. By using this list as an argument in the vectorizer, I was able to achieve 78% accuracy.

Finally, I attempted to correct some spelling mistakes in the dataset. However, the large size of the dataset made it impossible to manually correct all the errors. I tried using tools like TextBlob and SymSpell, but they were very time-consuming. Specifically, TextBlob took over 24 hours to process the three datasets and did not improve the model's accuracy. While SymSpell was faster, taking only 1–2 hours, it actually led to a decrease in accuracy.

Additionally, I experimented with lemmatization and negation handling methods, but they decreased the model's accuracy, so I did not include them in the final code.

Another important point is that I removed all punctuation, but this was a controversial decision. Some punctuation marks, such as $, &, and , did not add any meaning to the text, while others, like, could carry important sentimental information. I initially tried keeping, but it sometimes interfered with the preprocessing, ultimately reducing the model's accuracy. In the end, I decided to remove all punctuation, fully aware that this might lead to a slight loss in accuracy.

2.2. Analysis

<In this step, you should also try to visualize the data and their statistics (e.g., word clouds, tokens frequency, etc). So the processing should be done in parallel with an analysis. >

At this stage of the project, I began analyzing the data. I divided this part into two sections:

Analysis before pre-processing – Examining the raw dataset to understand its structure, common issues, and potential challenges.

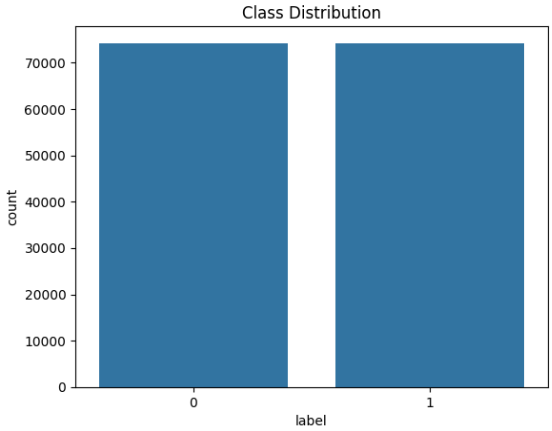
Analysis after pre-processing – Evaluating the cleaned dataset to assess improvements and how the applied transformations affected the data.

Before Preprocessing

Class Distribution:

The diagram plots the distribution of training data labeled as 1 (positive sentiment) and 0 (negative sentiment). The result confirms that our dataset is well-balanced, meaning we have approximately the same number of samples in both categories.

This balance is ideal because it ensures that the model is exposed to both sentiment classes equally, preventing bias toward one category and improving overall classification performance.

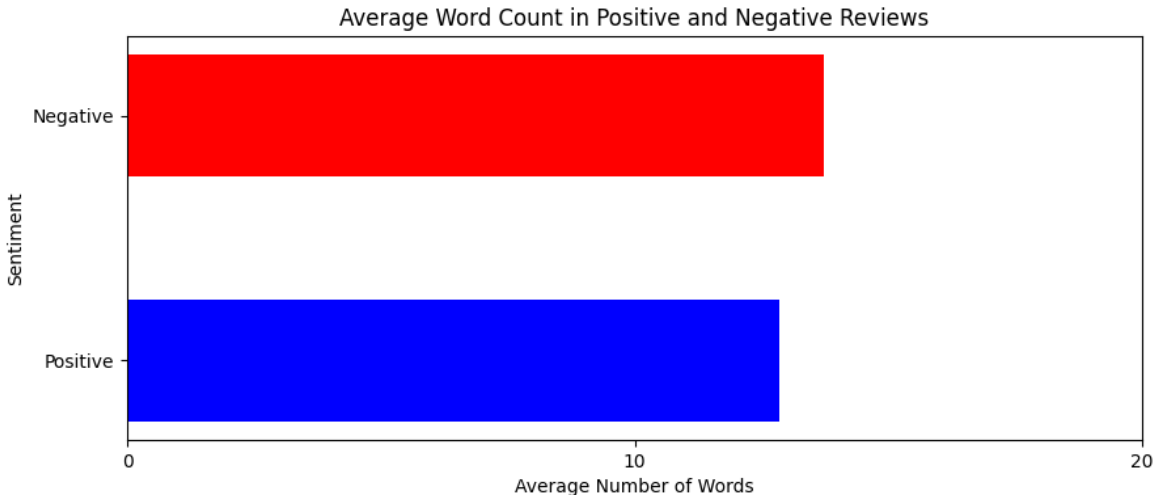


Average positive and negative word count before preprocess

This diagram visualizes the average length of texts with positive and negative sentiment. It helps us assess whether the data is balanced in terms of text length.

For example, if there were a significant difference in average word count between negative and positive sentiment texts, we could speculate that people tend to write more when they are unhappy, sad, or angry. However, this would only be an assumption.

The primary purpose of this diagram in this project is to confirm that the data is balanced, ensuring that both positive and negative sentiment texts are of similar length on average.

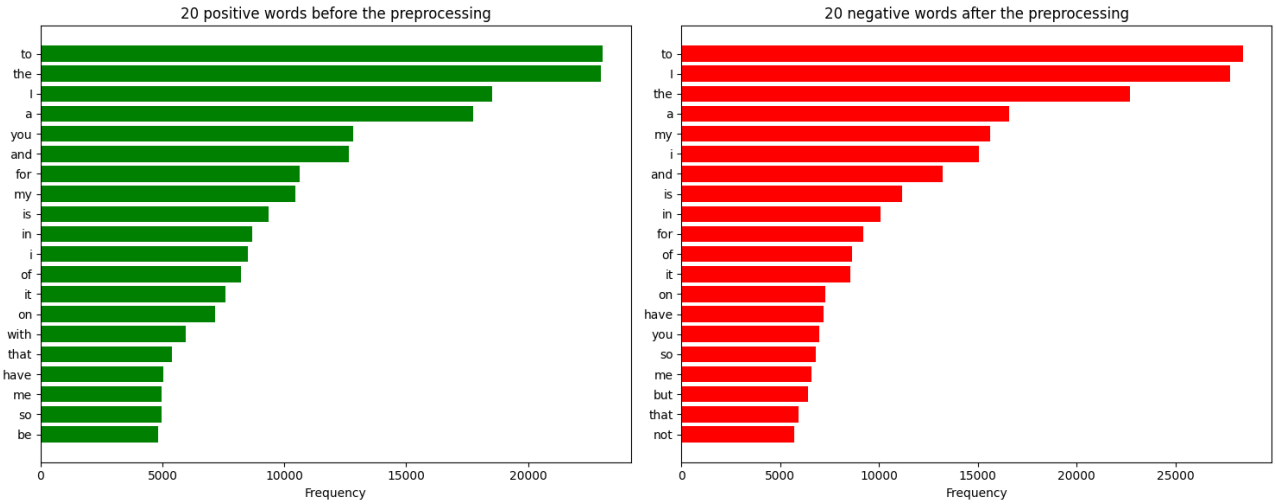


Most common positive and negative words before preprocess

This diagram visualizes the most frequently occurring positive and negative words in the dataset before pre-processing. However, as observed, the words appearing in both categories are identical.

This indicates that the dataset contains a large number of stopwords, which do not contribute meaningfully to sentiment classification. If left unprocessed, these words would very much reduce the model's accuracy.

At this point, we confirmed that removing stopwords is a crucial step in pre-processing to enhance the model's performance. We will see an update of this diagram after the pre-process.



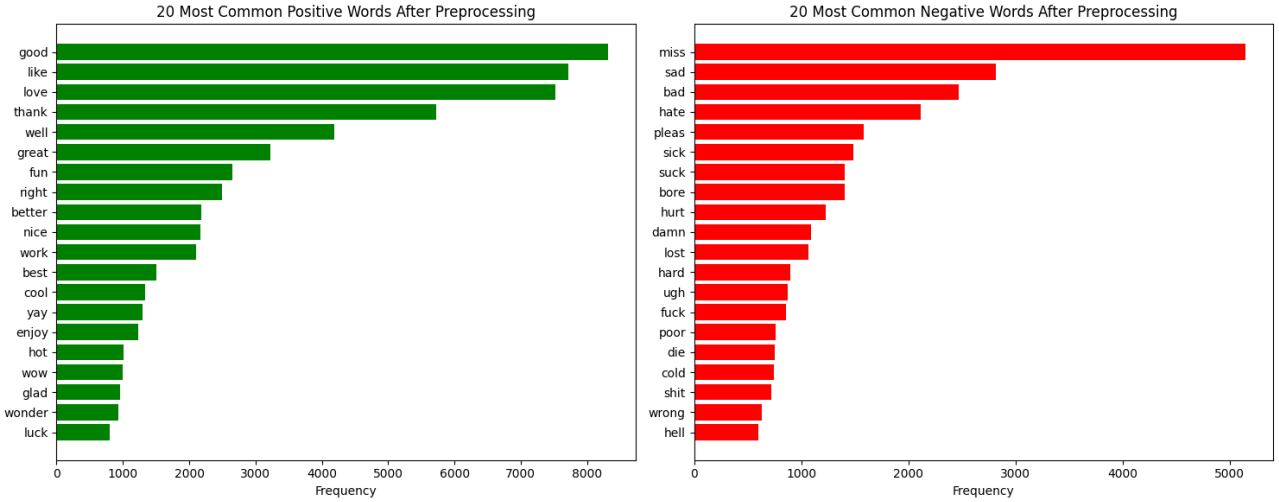
As observed, the analysis before pre-processing primarily provides reassurance about the necessary steps to follow in the process: pre-process, split, vectorize, and fit the model.

This initial analysis helps confirm the key areas that need attention, such as removing stopwords and ensuring the data is balanced, guiding us to the most effective approach for preparing the data and training the model.

After Preprocessing

Most common positive and negative words after preprocess:

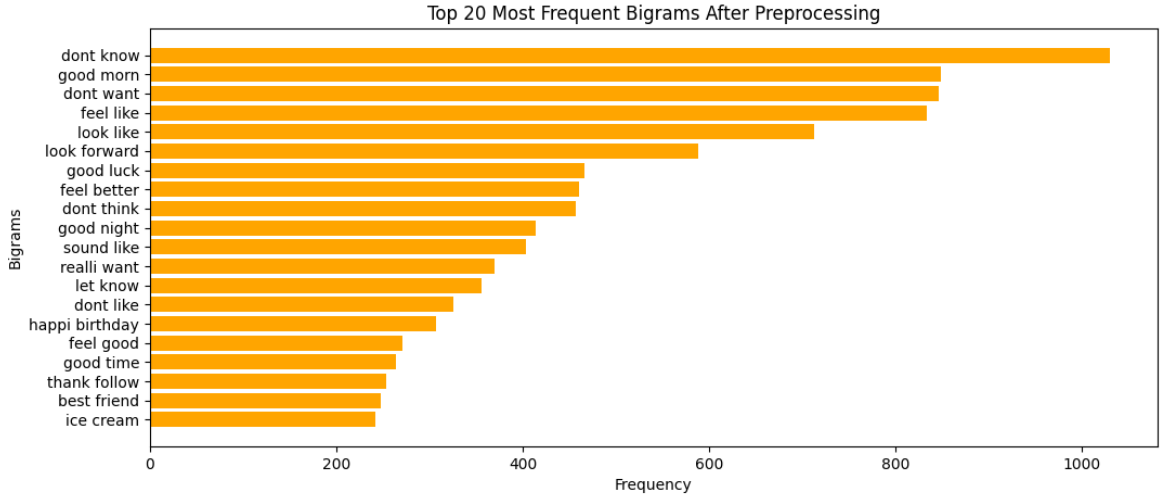
After pre-processing the data, we plotted the updated diagram, showing the most common positive and negative words. Now, we can observe that the most frequent words for each sentiment are those that, when used in a sentence, clearly indicate the sentiment—words with a direct and obvious positive or negative meaning.



Most frequent Bigrams after preprocess:

This diagram visualizes the most common bigrams in the dataset after pre-processing. The purpose of this analysis is to ensure that no useless patterns—such as "there is", "not the", etc.—dominate the data, as these do not contribute meaningful information for sentiment classification.

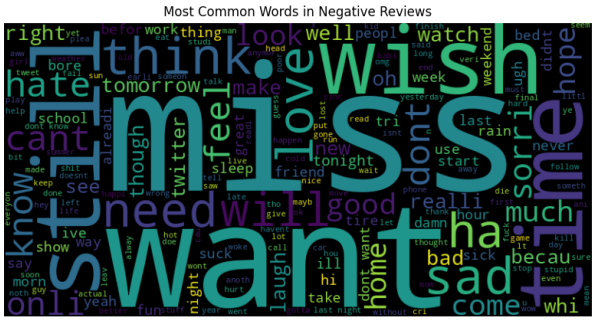
From the diagram, we observe that most bigrams clearly indicate a positive sentiment, which suggests that they can be highly useful for the model in correctly labeling tweets. This reinforces the effectiveness of our pre-processing steps in enhancing the dataset's quality for training.



Word Clouds: To visualize the clean and tidy dataset, we use a wordcloud .Wordcloud library provides a graphical representation in which the most common words are displayed in larger font sizes, while less frequent words are shown in smaller size. Bellow we see the wordcloud with the positive and negative sentiment words

The majority of the words observed are positive or neutral in nature, with "thank" and "love" being the most common. As a result, the frequently used words align well with the sentiment of positive tweets.

On the other hand, many of the words observed in the negative sentiment tweets again contribute to the overall negative sentiment. As a result, the frequently used words in these tweets do align well with the expected negative sentiment. But there are also some words such as "time" or "still" that they are preaty neutral. I tried to remove those words manualy, but the accuracy was decreased.



2.3. Data partitioning for train, test and validation

<Describe how you partitioned the dataset and why you selected these ratios>

Train and Testing Data Split:

At this point we have to split the data into training and testing.This following command is the one that does that job. Lets explain the reason between the ratios for each variable.

```
#splitting the data set
X_train, X_test, y_train, y_test = train_test_split(df["text"], df["label"], test_size=0.2, random_state=42, stratify=df["label"])
```

test size = 0.2

We selected 20% of the training data for testing the model at the final stage. This percentage was chosen carefully to prevent the model from overfitting (memorizing the training data too well) or underfitting (failing to capture meaningful patterns).

Through experimentation with different test sizes (10%, 15%, and 30%), we found that 20% provided the best accuracy in the final evaluation. This balance ensures that the model is trained on sufficient data while still having enough unseen data for a reliable performance assessment.

random state = 42

Setting the random state to 42% is a common practice in machine learning. The main reason for this choice is to ensure reproducibility allowing anyone who runs our model to obtain the same results. By fixing the random state, we make sure that data splitting and other randomized processes remain consistent across different runs, leading to reliable and comparable outcomes.

Train and Validation Data Split:

Now we split again our testing data and keep some for validation.

```
#splitting the training data set to validation
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.1, random_state=42, stratify=y_train)
```

From the 80% remain train data we keep the 10% for validation.
Now we have the 3 data sets:
Train set: 72%
Validation set: 8%
Testing set: 20%

2.4. Vectorization

<Explain the technique used for vectorization>
For text vectorization, we were required to use the TF-IDF Vectorizer. The importance of each word is determined by two key factors: Term Frequency (TF) – Measures how often a word appears in a given text. Inverse Document Frequency (IDF) – Evaluates the significance of a word based on its occurrence across the entire dataset. Words that appear frequently in many documents receive lower importance, while rare but meaningful words are given more weight. In the implementation, I used the TF-IDF Vectorizer from the sklearn library to transform the text data into numerical representations for the machine learning model.

```
#TF-IDF Method
vectorizer = TfidfVectorizer(max_df=0.7, min_df=10, ngram_range=(1,2), stop_words=list(custom_stopwords))
X_train_tfidf = vectorizer.fit_transform(X_train)
X_test_tfidf = vectorizer.transform(X_test)
X_val_tfidf = vectorizer.transform(X_val)
```

Let now explain the parameters used:
max_df = 0.7: This means that the words that appear 70% of the times in the texts we ignore them
min_df = 10: This means that the words that appear in less than 10 texts we ignore them
stop_words=list(custom_stopwords): We ignore all the stopwords in the custom list. As mentioned earlier, I created this list based on the diagram showing the most frequent words in both positive and negative sentiment before preprocessing (Diagram 3). By using these custom stopwords, I aimed to maximize the model's accuracy, as the excluded words were specifically tailored to this dataset. This approach ensured that only the most relevant words were considered in the classification process.
The parameters for TF-IDF Vectorization were carefully chosen to improve the performance of the sentiment classifier. Words that appear too frequently or too rarely do not contribute significantly to classification and may introduce noise. Therefore, we chose to ignore them to enhance model efficiency.

Additionally, we tested different max-min frequency thresholds (e.g., 80-20 and 90-10), but these settings significantly increased the runtime without improving the model's accuracy. Based on these observations, we selected the most optimal parameters to achieve a balance between performance and efficiency.

3. Algorithms and Experiments

3.1. Experiments

<Describe how you faced this problem. For example, you can start by describing a first brute-force run and afterwords showcase techniques that you experimented with. **Caution:** we want/need to see your experiments here either they increased or decreased scores. At the same time you should comment and try to explain why an experiment failed or succeeded. You can also provide plots (e.g., ROC curves, Learning-

curves, Confusion matrices, etc) showing the results of your experiment. Some techniques you can try for experiments are cross-validation, data regularization, dimension reduction, batch/partition size configuration, data pre-processing from 2.1, gradient descent>

Initial Brute-Force Approach

Initially, I started with a brute-force approach, which involved: Loading the datasets, splitting the data into training, validation, and testing sets, applying the TF-IDF Vectorizer,training a Logistic Regression model and printing the accuracy. As expected, this brute-force technique was not very effective, as it did not involve any optimization or fine-tuning. The model achieved an accuracy of 74%, which indicated that further improvements were necessary.

Pre-processing Function Development

To improve the model's performance, I systematically refined the pre-processing function through multiple steps:

Lowercasing – Converted all text to lowercase to ensure uniformity (e.g., treating "data" and "Data" as the same word).Removing special characters, numbers, extra spaces, and URLs – Cleaned the dataset by eliminating unnecessary elements, the accuracy was increased.

Tokenization and Lemmatization – Tokenized the text into individual words and applied lemmatization to reduce words to their base form (e.g., "running" $\rightarrow$ "run"). Despite these improvements into the data set, the accuracy by using the lemmantization was decreased to 74% so I only used the tokenization, which left the accuracy of the model unchanged.

Removing Stopwords – Eliminated common stopwords that do not contribute to sentiment classification. This was implemented using the NLTK library, which provides a predefined list of basic English stopwords.

After removing stopwords, the model's accuracy increased to 75%.But as observed, the predefined NLTK stopwords list was too limited for a dataset of 2000 lines. Removing only these words had minimal impact on improving accuracy.To enhance the effectiveness of stopword removal, I tried to combined multiple stopword libraries that contain custom stopwords beyond the basic NLTK list.The I also tried to manually remove frequently occurring words that I noticed appeared too often but did not contribute meaningfully to sentiment classification. Lets now see the trials and the results from each one:

Trial	**Score**
nltk	75%
ntlk+sklearn	73%
ntlk+sklearn+spaCy	73%
ntlk+custom	73%
sklearn+custom	73%
custom	78%

Table 1: Trials

3.1.1. Table of trials. From these results, we can conclude that removing too many words led to a decrease in the model's accuracy. However, when using the custom-made stopwords list, we achieved the best accuracy. This is because the custom list was specifically created based on our dataset and included only words that did not contribute to meaningful classification.

3.2. Hyper-parameter tuning

<Describe the results and how you configured the model. What happens with under-over-fitting??>
For hyper-parameter tuning, we used GridSearchCV with a Logistic Regression model. The parameter grid was:

```
#hyperparameter with GridSearchCV and Logistic Regreation Model
param_grid = {'C': [0.01, 0.1, 1, 10], 'solver': ['lbfgs', 'saga'], 'max_iter': [3000]}
kfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
model = GridSearchCV(LogisticRegression(), param_grid, cv=kfold, scoring='accuracy')
model.fit(X_train_tfidf, y_train)
```

We performed 5-fold Stratified Cross-Validation, ensuring balanced class distributions across folds. The best parameters found were:

```
Best Parameters: {'C': 1, 'max_iter': 3000, 'solver': 'lbfgs'}
Accuracy:  0.78
```

Underfitting and Overfitting With low C values (e.x. 0.001), the model was underfitting, meaning it failed to capture important patterns. At C = 1, the model balanced bias-variance well, preventing underfitting. Higher C values (e.x., 10, 100) slightly increased training accuracy but caused overfitting, where the model learned noise from training data and performed worse on test data.

3.3. Optimization techniques

<Describe the optimization techniques you tried. Like optimization frameworks you used.>
To improve model performance and efficiency, several optimization techniques were applied during hyperparameter tuning and training.
Grid Search for Hyperparameter Tuning
Used GridSearchCV to systematically test different hyperparameter values. The goal for this was to achieve a balanced bias-variance tradeoff to avoid underfitting and overfitting.
Cross-Validation (Stratified K-Fold)
Used Stratified K-Fold (K=5) to ensure class distribution consistency across folds.This helps to avoid overfitting by validating performance on different splits of data.
Feature Selection
Adjusted max_df and min_df in TF-IDF to remove uninformative words.
Tested different solvers

Tested 'lbfgs' and 'saga'. Chose the best solver based on accuracy and training speed.
Scaling for Stability
Tried StandardScaler to normalize the TF-IDF matrix before training. Although this does not improve the accuracy of the model and also decreased its speed, so at the end I decided not to use this technique.

3.4. Evaluation

<How will you evaluate the predictions? Detail and explain the scores used (what's fscore?). Provide the results in a matrix/plots>
<Provide and comment diagrams and curves>
Those are the final results of my implementation of the sentiment Classifier.

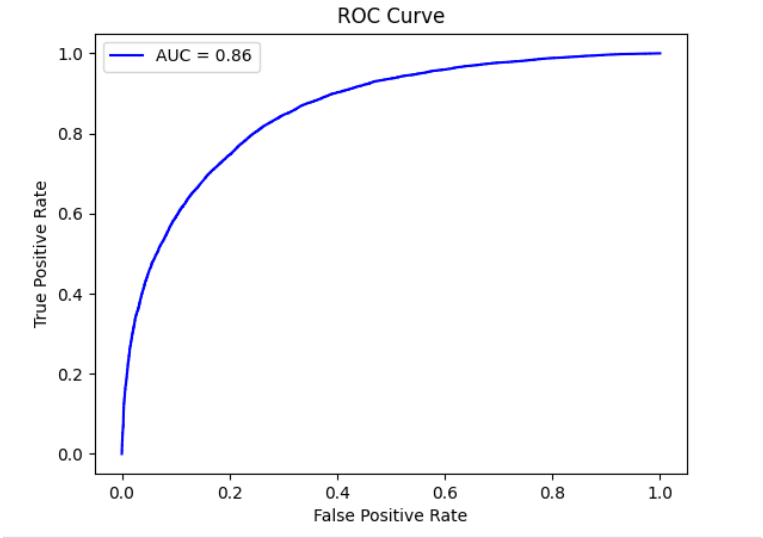
```
Best Parameters: {'C': 1, 'max_iter': 3000, 'solver': 'lbfgs'}
Accuracy:  0.78
Classification Report:
              precision    recall  f1-score   support

           0       0.79      0.76      0.77     14839
           1       0.77      0.80      0.78     14839

    accuracy                           0.78     29678
   macro avg       0.78      0.78      0.78     29678
weighted avg       0.78      0.78      0.78     29678

Cross-validation Accuracy: 0.78 ± 0.00
Train Loss: 0.4335
Validation Loss: 0.4725
```

Let now analyse them:
Accuracy: The 78% is a good baseline, but could be improved. The classifier generalizes reasonably well, with a balanced performance across both classes.
Precision: Class 0 (Negative) has 79% precision, meaning when the model predicts negative, it's correct 79% of the time. Class 1 (Positive) has 77% precision, slightly lower, meaning there are slightly more false positives.
Recall: Class 0 recall = 76% .The model identifies 76% of actual negatives correctly.Class 1 recall = 80% .The model captures 80% of actual positives, slightly better.

The model is slightly imbalanced, as it performs better at capturing positive sentiments than negative ones. This is expected because, as shown in Diagram 2, negative tweets tend to contain more words than positive ones. The higher word count increases the likelihood of encountering words that might confuse the model, leading to a slight imbalance in its performance.
F1 score:0.77 (Negative) vs. 0.78 (Positive). Since precision and recall are close, the F1-score is balanced for both classes.78% F1-score overall suggests a strong balance between false positives and false negatives.
Cross-Validation Accuracy: Loss Analysis: The Train Loss is 0.4335. Low train loss indicates the model is learning well on training data. The Validation Loss is 0.4725 which

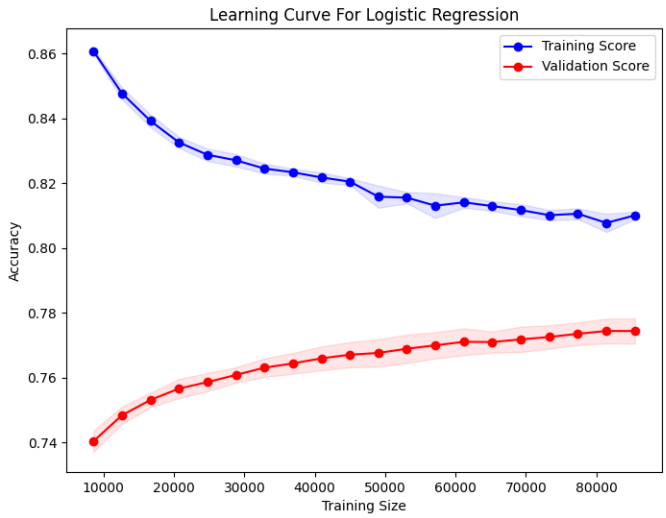
is slightly higher than train loss, indicating some generalization gap. This means that the model learns well on training data but performs slightly worse on unseen data.

3.4.1. ROC curve.



AUC = 0.86. This indicates that the model has a good discriminative ability.The curve rises quickly, showing that the model can distinguish between the two classes well. The AUC of 0.86 is strong, meaning that the model performs well in separating positive and negative classes. However, it is not perfect (AUC = 1.0 would be ideal), meaning that there is still some misclassification.

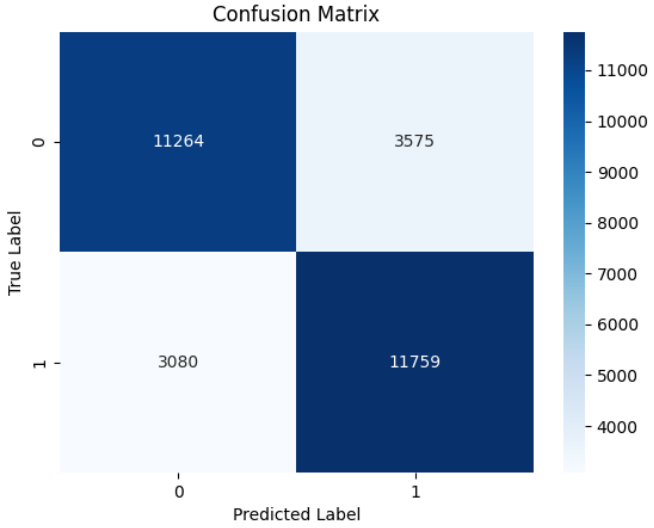
3.4.2. Learning Curve.



Training Score starts high (0.86) and decreases. Initially, when the model is trained on a small dataset, it fits the data almost perfectly, resulting in a high training score. As more data is added, training accuracy gradually decreases and stabilizes around 82%. Validation Score starts Low (0.74) and increases with more data When training

with a small dataset, the validation score is relatively low (74%) because the model isn't exposed to enough diverse data. As training size increases, validation accuracy steadily improves and stabilizes at 78%. This suggests that the model benefits from having more training data. The gap between Training and Validation scores is almost 4%. If the gap were larger , the model would be overfitting, meaning it memorizes training data but fails to generalize. Since the gap remains stable, it suggests the model has learned a good balance between bias and variance. Because now the validation score increases with more data we do not observe underfitting. Also, because the train score is not much higher than the validation one, we do no also observe underfitting.

3.4.3. Confusion matrix.



Class 0 (Negative Sentiment):
Correctly classified (True Negatives): 11,264
Misclassified as Positive (False Positives): 3,575
Class 1 (Positive Sentiment):
Correctly classified (True Positives): 11,759
Misclassified as Negative (False Negatives): 3,080

There is a reasonable balance in errors, meaning the model does not favor one class significantly.

4. Results and Overall Analysis

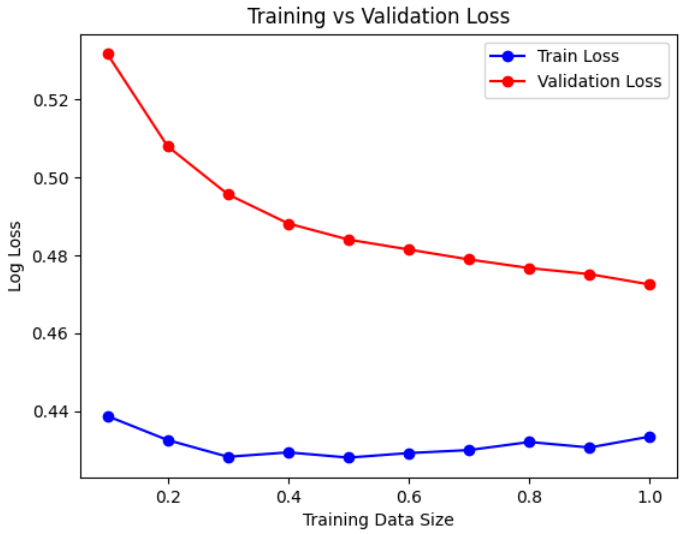
4.1. Results Analysis

<Comment your results so far. Is this a good/bad performance? What was expected? Could you do more experiments? And if yes what would you try?>

<Provide and comment diagrams and curves>

The model achieved an accuracy of 78%, which is decent but leaves room for improvement. The classification report shows a balanced precision and recall, while the ROC curve (AUC = 0.86) indicates good discriminatory power. The learning curve suggests that adding more data may help but won't drastically improve performance.

Some misclassifications are observed, particularly 3,578 false positives and 3,027 false negatives in the confusion matrix. I tried refining text preprocessing by preserving sentiment-relevant punctuation to improve the accuracy of the model and I manually corrected some very common words like "haha", but again the results was 78%. Also, I experimented with different parameters in the Vectorizer and the GridSearch, but again the best result I could achieve is 78%. For those results the diagram forn the training vs validation loss is:



The model is not overfitting because the gap between training and validation loss is not excessively large.

4.1.1. Best trial. <Showcase best trial>

My best trial was the one indicated below.

```
Best Parameters: {'C': 1, 'max_iter': 3000, 'solver': 'lbfgs'}
Accuracy:  0.78
Classification Report:
              precision    recall  f1-score   support

           0       0.79      0.76      0.77     14839
           1       0.77      0.80      0.78     14839

    accuracy                           0.78     29678
   macro avg       0.78      0.78      0.78     29678
weighted avg       0.78      0.78      0.78     29678

Cross-validation Accuracy: 0.78 ± 0.00
Train Loss: 0.4335
Validation Loss: 0.4725
```

4.2. Comparison with the first project

<Use only for projects 2,3,4>
<Comment the results. Why the results are better/worse/the same?>

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] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

<You should link and cite whatever you used or "got inspired" from the web. Use the / cite command and add the paper/website/tutorial in refs.bib>
<Example of citing a source is like this:> [1] <More about bibtex>