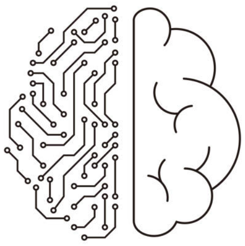
University of Athens
Department of Informatics and Telecommunications

TF-IDF and Logistic Regression in Deep Learning for NLP



Student name: Antonis Zikas

email: sdi2100038@di.uoa.gr

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

Contents

1 Data Processing and Analysis **2**
1.1 Data Pre-processing 2
1.1.1 General Cleaning 2
1.1.2 Stop-words Removal 2
1.2 Exploratory Data Analysis 3
1.2.1 Sentiment Correlation 3
1.2.2 Dataset Splitting 3
1.2.3 Sentiment Distribution 3
1.2.4 Analysis of Word Frequencies by Sentiment 4
1.2.5 Text Length Analysis 4
1.3 Data Vectorization 5
1.3.1 Introduction 5
1.3.2 Term Frequency (TF) 5
1.3.3 Inverse Document Frequency (IDF) 5
1.3.4 TF-IDF 6
1.4 Data Visualization with PCA 6

2 Model Training and Evaluation **7**
2.1 Model Foundations 7
2.1.1 Dummy Model 7
2.1.2 Gradient Descent 7
2.1.3 Base Model 9
2.2 Hyperparameter Tuning 9
2.2.1 Maximum Features Selection 9
2.2.2 N-Gram Analysis 10
2.2.3 Optimal Parameter Combination 10
2.2.4 Optimization Parameters: Solvers, Regularization, and Penalties 11
2.2.5 Hyperparameter Optimization via Grid Search 11
2.3 Appendices 11

3 Results and Overall Analysis **12**
3.1 Experimental Summary 12
3.2 Key Findings 12
3.3 Optimal Configuration Analysis 12

4 Bibliography **12**

ABSTRACT

In this article, we investigate the field of **Natural Language Processing** (NLP) by developing a machine learning model designed to classify **user comments** as either **positive** or **negative**. Our objective is to construct a **sentiment classifier** using a dataset of **Twitter** user comments. The model will leverage the **logistic regression** algorithm combined with the **TF-IDF** approach.

KEYWORDS. Machine Learning, Sentiment Classifier, Logistic Regression, TF-IDF

1 Data Processing and Analysis

The dataset is first **processed** and **cleaned**. In data analysis and machine learning, data preprocessing is considered the most critical initial phase of the workflow. Ensuring that the data is in a **suitable format** is essential for effective algorithmic applications.

1.1 Data Pre-processing

1.1.1. General Cleaning. The dataset underwent several preprocessing and cleaning techniques, including:

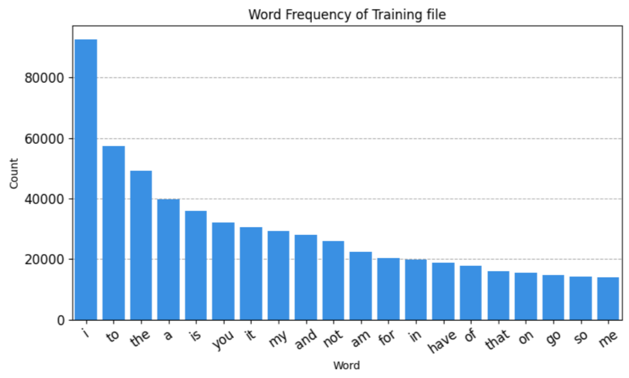
- **Removal of URLs, user mentions, hashtags, special characters, and numbers**,
- **Lowercasing all characters**,
- **Elimination of extra whitespaces**,
- **Stemming and lemmatization** of words, and
- **Removal of punctuation**

Applying these methods to the dataset yielded the results shown in Table 1, which displays a representative subset of transformed tweets. The table demonstrates how the preprocessing steps modified the text: user mentions were removed, characters were converted to lowercase, and other changes were applied.

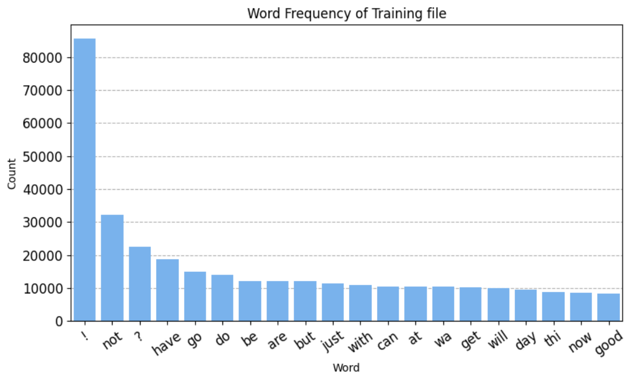
Notably, certain punctuation marks—**exclamation points** (!), **question marks** (?), and **ellipses** (...)—were intentionally retained due to their potential sentiment value. For instance, an **exclamation point** (!) may indicate enthusiasm, while **ellipses** (...) might suggest hesitation or irony.

1.1.2. Stop-words Removal. The subsequent preprocessing phase focuses on eliminating words with minimal sentiment value, known as **stop-words**. These terms provide negligible benefit for model training due to their limited semantic content. As such words frequently dominate text corpora without meaningful interpretation, their removal was prioritized.

A custom stop-word removal approach was implemented, differing from standard library-based methods. The processed dataset results are shown in Table 1, with the complete preprocessing workflow visualized in Figure 1 for the **training file**.



(a) General Cleaning word frequency



(b) Stop-words removal word frequency

Figure 1: Most frequent words across preprocessing steps

ID	Original Text	General Cleaning	Stop-words Removal
39015	how much i do love him	how much i do love him	how much do love him
15358	@Simpsdj Cool stuffs, sounds interesting	cool stuff sound interest	cool stuff sound interest
7666	@zackalltimelow dude what is it like	dude what is it like	dude what like
103061	on my way to METROBAR... teleserv anniv	on my way to metrobar... teleserv anniv	way metrobar... teleserv anniv

Table 1: Text Transformation after preprocessing. Second column shows the original text, the third column represents the text after general cleaning and fourth column shows the text after removing stop-words.

Antonis Zikas

email: sdi2100038@di.uoa.gr

1.2 Exploratory Data Analysis

We will now proceed to some deeper analysis of the dataset.

1.2.1. Sentiment Correlation. The **Exploratory Data Analysis** (EDA) commenced with **sentiment correlation analysis** to identify words most frequently associated with each sentiment. The dataset was divided into positive and negative **subsets**, followed by extraction of frequent terms. The results are presented in Figure 2.

1.2.2. Dataset Splitting. In machine learning applications, proper dataset partitioning is essential for model development. The standard approach divides data into three distinct subsets: **70% for training**, **20% for validation**, and **10% for testing**. This structured division facilitates effective model training, hyperparameter tuning, and final evaluation. The current dataset's file size distribution across these splits is presented in Table 2, with additional visualization provided in Figure 3.

Dataset File	Records Count	Size Rate
Training	593.552	≈ 70.00%
Validating	169.584	≈ 20.00%
Testing	63.597	≈ 10.00%

Table 2: Dataset training, validating and testing files size

1.2.3. Sentiment Distribution. A **Sentiment Distribution Analysis** was conducted for both training and validation datasets. This analysis quantified the proportion of positive versus negative comments in each dataset. The results, presented in Table 3 and Figure 3, reveal near-equal representation of both sentiment classes. While **positive comments** slightly **outnumber** negative ones in both datasets, the marginal difference confirms their **balanced** distribution. This equilibrium suggests neither sentiment class dominates the training or validation data.

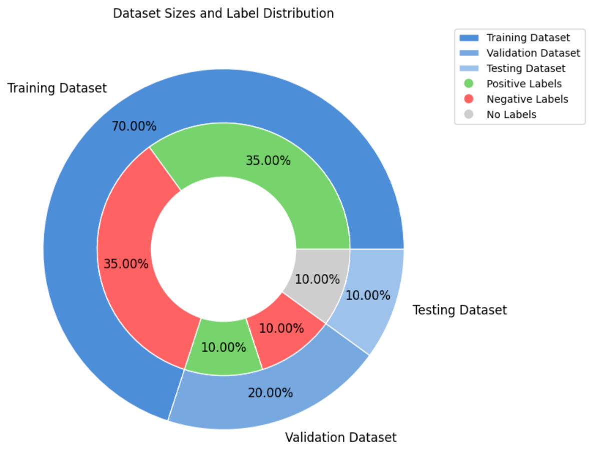


Figure 3: Training, Validating and Testing Datasets sizing and sentiment distribution analysis

Data Balance. The balanced dataset ensures a **fair learning process**, where the model is not biased toward any particular class due to unequal representation. This equilibrium promotes better generalization across all classes and yields more reliable validation performance. Consequently, evaluation metrics including **accuracy**, **precision**, **recall**, and **F1-score** provide meaningful assessments of the model's predictive capabilities.

Dataset File	Positive Tweets	Negative Tweets
Training	74.196	74.192
Validating	21.199	21.197
Testing	-	-

Table 3: Sentiment Distribution across the files. How many positive and negative tweets are there.

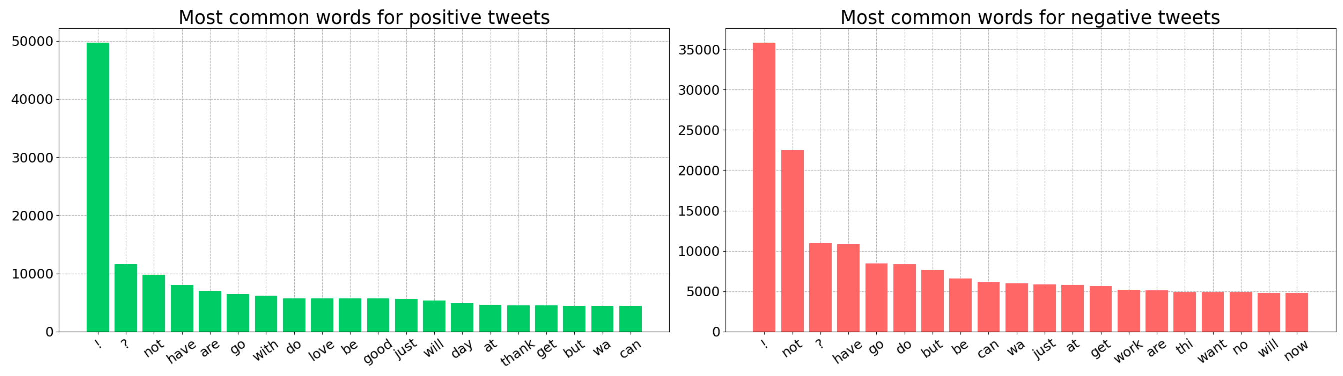


Figure 2: Sentiment Correlation: Most frequent words for positive and negative tweets file

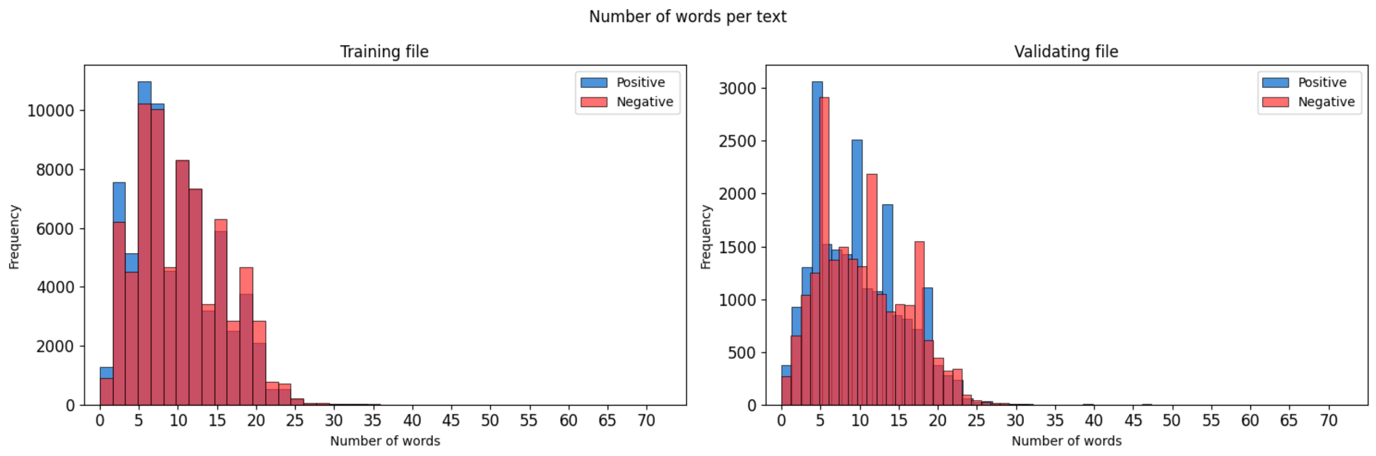


Figure 4: Sentiment Text Length Analysis for both training and validation files. X-axis represents the number of words and y-axis is the frequency of these numbers inside the files.

1.2.4. Analysis of Word Frequencies by Sentiment. Let's take a closer look at which words appear most often in our dataset and how they differ between positive and negative tweets. Figure 1 gives us an overview of the most common words overall, while Figure 5 specifically compares how these top 20 words appear in positive versus negative tweets.

Some interesting patterns stand out when we examine these results. Positive tweets tend to use certain features much more frequently - for example, **exclamation marks** (!) appear more often in positive tweets compared to negative ones. On the other hand, words that express negation like **'not'** show up significantly more in negative tweets. Other words like "good" and "great" naturally appear more in positive tweets, while terms like "bad" and "worst" dominate in negative ones. These patterns make perfect sense when we think about how people express positive versus negative feelings in their writing. The clear differences in word usage help our model learn to distinguish between these two sentiment categories.

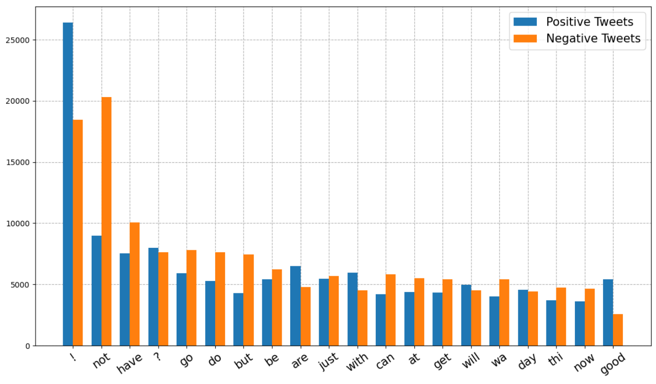


Figure 5: Comparative Analysis of Word Frequencies by Sentiment. How often does a word appear in positive and negative tweets.

1.2.5. Text Length Analysis. Now let's examine how long the tweets are in our dataset. We analyzed the number of words in both our **training set** (which contains 70% of our data) and our **validation set** (containing 20%). Looking at Figure 6, we can see that most tweets are quite short, with the majority falling between **0 to 30** words.

In the training data, the most common tweet lengths are **5-6** words, while in the validation set it's slightly shorter at **4-5** words. The validation set shows smaller numbers overall, but this is exactly what we'd expect since it's a smaller sample of data. When we separate tweets by sentiment (shown in Figure 4), we notice that positive and negative tweets have almost identical length distributions. Whether someone is expressing something positive or negative doesn't seem to affect how long their tweet is - both types average around the same word count. This tells us that text length probably won't be a useful feature for our sentiment classifier, since it doesn't correlate with sentiment.

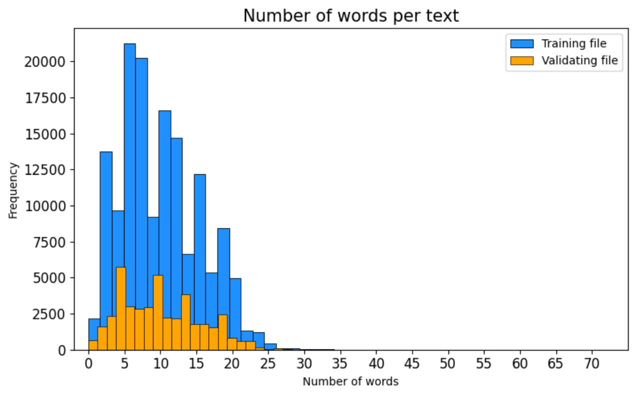


Figure 6: Text Length Analysis for training and validation dataset files. X-axis represents the number of words and y-axis is the frequency of these numbers inside the files.

1.3 Data Vectorization

To enable sentiment classification, text data must be converted to numerical form. The **TF-IDF** vectorizer transforms text by weighting words based on both their frequency in individual tweets and their rarity across the entire dataset. This creates meaningful numerical representations while filtering out less important terms.

1.3.1. Introduction. The TF-IDF vectorization method is formally introduced below.

TF-IDF (Term Frequency-Inverse Document Frequency) is a statistical measure that evaluates a word's relevance in a document relative to a corpus. This metric is fundamental in information retrieval and natural language processing applications.

The TF-IDF value combines two components: **Term Frequency (TF)**, quantifying a word's occurrence in a document, and **Inverse Document Frequency (IDF)**, assessing the word's distinctiveness across the entire document collection.

1.3.2. Term Frequency (TF). Term Frequency measures how often a word appears in a document, normalized by the document's length. The TF calculation is given by:

$$TF(t,d) = \frac{f_{t,d}}{N_d} \quad (1)$$

where t is the target term (word), d is a specific document, $f_{t,d}$ is the raw count of term t in document d and N_d is the total word count in document d. The frequency calculation process is implemented in Algorithm 1.

Algorithm 1: Frequency(t, d)

```
Input  : A term $t$, and a document $d$
Output: The frequency of $t$ in $d$
1 Let $n$ be the total number of terms in document $d$
2 Let $c$ be the number of times the term $t$ appear in
    document $d$
3 return d/n
```

More commonly, we squash the raw frequency a bit, by using the $\log_{10}$ of the frequency instead. The intuition is that a word appearing 100 times in a document does not make that word 100 times more likely to be relevant to the meaning of the document. We also need to do something with counts of 0, since log of 0 is undefined. Then the calculation of the TF is simplified and shown in Formula 2.

$$TF(t,d) = \begin{cases} 1 + \log_{10}(f_{t,d}) & \text{if } f_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

Algorithm 2 explains how the Term Frequency (TF) method works by applying Formula 2 on each term inside a given collection of documents.

Algorithm 2: TF(D)

```
Input  : A collection $D$ of documents
Output: Result 2D matrix $TF$
1  $TF \leftarrow$ empty matrix of size $(|W|, |D|)$
2  $W \leftarrow$ {all terms in $D$ sorted lexicography}
3  for $i \in \{1, 2, \dots, |W|\}$ do
4      $t \leftarrow W_i$
5      for $j \in \{1, 2, \dots, |D|\}$ do
6          $d \leftarrow D_j$
7          $f_{t,d} \leftarrow$ Frequency$(t, d)$
8          if $f_{t,d} > 0$ then
9              $TF_{ij} \leftarrow 1 + \log_{10}(f_{t,d})$
10         else
11             $TF_{ij} \leftarrow 0$
12 return $TF$
```

The result of Algorithm 2 will be a $M \times N$ matrix that contains the TF values for every term in every document and will have the following form:

$$TF = \begin{bmatrix} TF_{11} & TF_{12} & \dots & TF_{1N} \\ TF_{21} & TF_{22} & \dots & TF_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ TF_{M1} & TF_{M2} & \dots & TF_{MN} \end{bmatrix} \in \mathbb{R}^{M \times N}$$

1.3.3. Inverse Document Frequency (IDF). The Inverse Document Frequency reduces the weight of terms that appear frequently across all documents because these words are less informative. It is calculated using the Formula 3.

$$IDF(t,D) = \log_{10}\left(\frac{|D|}{1 + |D_t|}\right),\ D_t = \{d \in D \mid t \in d\} \quad (3)$$

where $|D|$ is the total number of documents in the collection, and $|D_t|$ is the number of ducuments that contain term t. The fewer documents in which a term occurs, the higher its weight is. The lowest weight of 1 is assigned to terms that occur in all the documents. Because of the large number of documents in many collections, this measure is usually squashed with a log function too. This is why we use the $\log_{10}$ in Formula 3. The pseudocode of the IDF method is represented in Algorithm 3.

Algorithm 3: IDF(D)

```
Input  : A collection $D$ of documents
Output: Result vector $IDF$
1 $W \leftarrow$ {all terms in $D$ sorted lexicography}
2 $IDF \leftarrow$ empty column vector of size $|W|$
3 for $i \in \{1, 2, \dots, |W|\}$ do
4     $t \leftarrow W_i$
5     $D_t \leftarrow \{d \in D \mid t \in d\}$
6     $IDF_i \leftarrow \log_{10}\left(\frac{|D|}{1+|D_t|}\right)$
7 return $IDF$
```

$$TF\text{-}IDF = \begin{bmatrix} TF_{11} & TF_{12} & \dots & TF_{1N} \\ TF_{21} & TF_{22} & \dots & TF_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ TF_{M1} & TF_{M2} & \dots & TF_{MN} \end{bmatrix} \circ \begin{bmatrix} IDF_1 \\ IDF_2 \\ \vdots \\ IDF_M \end{bmatrix} = \begin{bmatrix} TF_{11} \cdot IDF_1 & TF_{12} \cdot IDF_1 & \dots & TF_{1N} \cdot IDF_1 \\ TF_{21} \cdot IDF_2 & TF_{22} \cdot IDF_2 & \dots & TF_{2N} \cdot IDF_2 \\ \vdots & \vdots & \ddots & \vdots \\ TF_{M1} \cdot IDF_M & TF_{M2} \cdot IDF_M & \dots & TF_{MN} \cdot IDF_M \end{bmatrix} \in \mathbb{R}^{M \times N} \quad (5)$$

The IDF of a collection of documents D is a column vector that contains the specific IDF value for every term in D and has the following form:

$$IDF = \begin{bmatrix} IDF_1 \\ IDF_2 \\ \vdots \\ IDF_M \end{bmatrix} \in \mathbb{R}^M$$

1.3.4. TF-IDF. By combining the Term Frequency and the Inverse Document Frequency, we can define the **TF-IDF** for a collection of documents. Let us have a collection of documents D. The TF-IDF score for a term t in a specific document d is given by the Formula 4.

$$TF\text{-}IDF(t, d, D) = TF(t, d) \times IDF(t, D) \quad (4)$$

Algorithm 4 describes how the TF-IDF vectorizer works by using Algorithm 2 and Algorithm 3 for a specific collection of documents D.

Algorithm 4: TF-IDF(D)

```
    Input   : A collection $D$ of documents
    Output: Result 2D matrix $TF\text{-}IDF$
1   $W \leftarrow \{$all terms in $D$ sorted lexicography$\}$
2   $TF\text{-}IDF \leftarrow$ empty matrix of size $(|W|, |D|)$
3   for $i \in \{1, 2, \dots, |W|\}$ do
4       for $j \in \{1, 2, \dots, |D|\}$ do
5           Let $TF \leftarrow \mathrm{TF}(D)$
6           Let $IDF \leftarrow \mathrm{IDF}(D)$
7           $TF\text{-}IDF_{ij} \leftarrow TF_{ij} \times IDF_i$
8   return $TF\text{-}IDF$
```

Algorithm 4 performs an **element-wise multiplication** between the TF matrix and IDF vector, and outputs a $M \times N$ matrix containing the TF-IDF values for every term t in every document d and has the form described in Formula 5. The resulting matrix shown in Formula 5 will be the training data of the future model we are going to implement.

1.4 Data Visualization with PCA

After vectorization, the data are **visualized** in a 2D space to provide an initial understanding of their structure. However, the vectorized training data have dimensions of 148.388×72.713, necessitating dimensionality reduction to 148.388×2 for effective visualization.

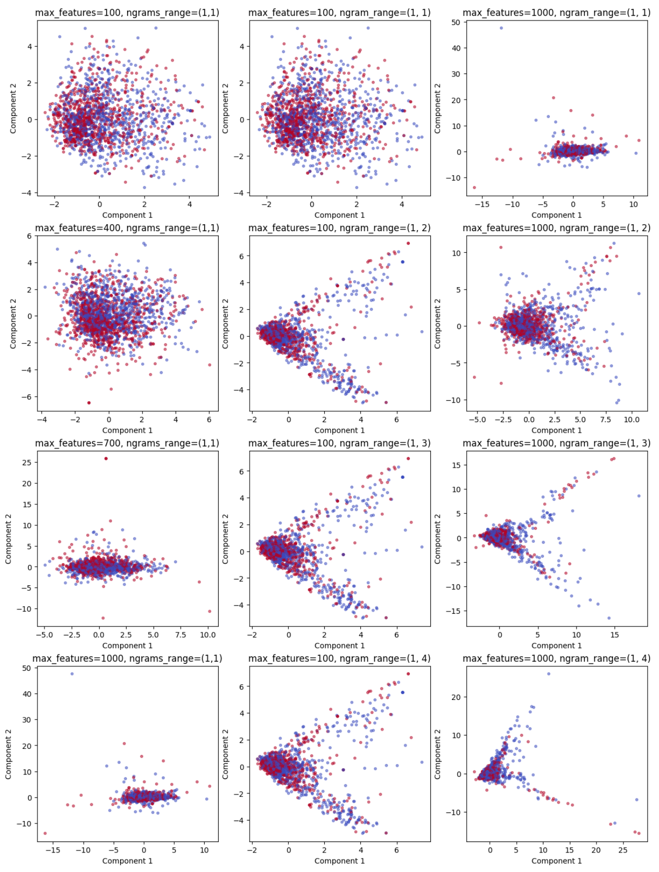


Figure 7: Training Data visualization using PCA and different combinations of max_features and ngram_range parameters of TF-IDF

What is PCA. **PCA** (Principal Component Analysis) is a dimensionality reduction technique that identifies orthogonal axes (principal components) maximizing variance. The method standardizes data, computes a covariance matrix, extracts eigenvectors/values, and projects data onto dominant components.

PCA will be used to reduce the data dimensions. Additionally, different values of **max_features** and **ngram_range** - parameters of Scikit Learn's TF-IDF - will be tested. These parameters will be briefly discussed later. After vectorizing the data with different combinations and applying PCA, the visualization results are shown in Figure 7.

2 Model Training and Evaluation

Following data vectorization using the TF-IDF algorithm implemented via the **Scikit Learn** Python library, model training can commence and experimental results can be extracted.

2.1 Model Foundations

2.1.1. Dummy Model. The experimentation process was initiated with a baseline model that generates **random guesses** for tweet sentiments in the validation dataset. This *dummy* model requires no training prior to prediction. After multiple iterations, an average accuracy of **0.5** was observed, which aligns with expectations given the dataset's **balanced** nature (as shown in Figure 3). This outcome demonstrates that by increasing random predictions, the accuracy naturally converges to **0.5**.

2.1.2. Gradient Descent. While the *dummy* model provides rapid predictions, its limitations necessitate pursuing more accurate alternatives. This study focuses on implementing training procedures using the **logistic regression** model, which will serve as our primary approach moving forward.

Logistic Regression. Logistic regression is a supervised learning algorithm for binary classification problems. While derived from linear regression, it differs by predicting class **probabilities** through application of the **sigmoid function** [1]:

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

The linear predictor z combines input features as:

$$z = w_1x_1 + \cdots + w_dx_d + b = W^TX + b$$

where x_i are features, w_i their corresponding weights, and b the bias term. The sigmoid transforms z into the range (0,1), yielding the class probability:

$$P(y=1 \mid X) = \hat{y} = \sigma(z) = \frac{1}{1+e^{-(W^TX+b)}}$$

Here, $\hat{y}$ represents the probability of class 1, with $1-\hat{y}$ for class 0.

In machine learning, model training aims to maximize accuracy while minimizing loss. **Gradient descent** [1] is an optimization algorithm designed to minimize a model's loss function. The algorithm iteratively adjusts the model's parameters (weights and bias) to identify the configuration that yields minimal loss, consequently achieving optimal accuracy. Model performance is quantified through a **loss function**, which numerically evaluates the discrepancy between predictions and ground truth. For gradient descent in classification tasks, the **Cross-Entropy Loss** [1] (Formula 6) is commonly employed.

$$J(W,b) = -\frac{1}{m}\sum_{i=1}^{m}\left(y_i \cdot log(\hat{y}_i) + (1-y_i)\cdot log(1-\hat{y}_i)\right) \tag{6}$$

where X is the training data matrix, W is the weight vector, b is the bias, y is the true label vector, $\hat{y} = \sigma(XW+b)$ is the predicted label vector. In order to minimize the loss function represented in Formula 6, we need to find its **gradient**. The gradient of the loss function is shown below:

$$\nabla J(W,b) = \left(\frac{\partial J}{\partial W}, \frac{\partial J}{\partial b}\right)$$

Using calculus [2] to find the individual derivatives in the upper equation, we have the following.

$$\begin{aligned}
\frac{\partial J}{\partial W} &= -\frac{1}{m}\sum_{i=1}^{m}\frac{\partial}{\partial W}\left(y_i \cdot log(\hat{y}_i) + (1-y_i)\cdot log(1-\hat{y}_i)\right) \Rightarrow \\
&= -\frac{1}{m}\sum_{i=1}^{m}\left(\frac{\partial}{\partial W}y_i \cdot log(\hat{y}_i) + \frac{\partial}{\partial W}(1-y_i)\cdot log(1-\hat{y}_i)\right) \Rightarrow \\
&= -\frac{1}{m}\sum_{i=1}^{m}\left(y_i \cdot \frac{1}{\hat{y}_i}\cdot\frac{\partial \hat{y}_i}{\partial W} + (1-y_i)\cdot\frac{1}{1-\hat{y}_i}\cdot\frac{\partial(1-\hat{y}_i)}{\partial W}\right)
\end{aligned} \tag{7}$$

In Formula 7 we have two terms to compute. The first term is $\partial\hat{y}_i/\partial W$, where $\hat{y}_i$ is the sigmoid function $\sigma(z)$ evaluated at $z = W^Tx_i + b$. We take as given that the partial derivative of the sigmoid function $\sigma(x)$ is: $\frac{\partial}{\partial x}\sigma(x) = \sigma(x)\,(1-\sigma(x))$. Given that, we can compute the first term of Formula 7 and we have that

$$\frac{\partial \hat{y}_i}{\partial W} = \hat{y}_i(1-\hat{y}_i)\cdot\frac{\partial}{\partial W}\left(W^Tx_i + b\right) = \hat{y}_i(1-\hat{y}_i)x_i \tag{8}$$

The second term of Formula 7, which is $\partial(1-\hat{y}_i)/\partial W$ is computed in a similar way and it is

$$\frac{\partial(1-\hat{y}_i)}{\partial W} = -\hat{y}_i(1-\hat{y}_i)x_i \tag{9}$$

Combining Formulas 8 and 9 we can simplify Formula 7 and we have that the first partial derivative of the cross entropy function with respect to the weights is

$$\begin{aligned}
\frac{\partial J}{\partial W} &= -\frac{1}{m}\sum_{i=1}^{m}\left(y_i \cdot (1-\hat{y}_i)x_i - (1-y_i)\cdot\hat{y}_ix_i\right) \Rightarrow \\
\frac{\partial J}{\partial W} &= -\frac{1}{m}\sum_{i=1}^{m}x_i(y_i - \hat{y}_iy_i - \hat{y}_i + \hat{y}_iy_i) \Rightarrow \\
\frac{\partial J}{\partial W} &= -\frac{1}{m}\sum_{i=1}^{m}x_i(y_i - \hat{y}_i)
\end{aligned}$$

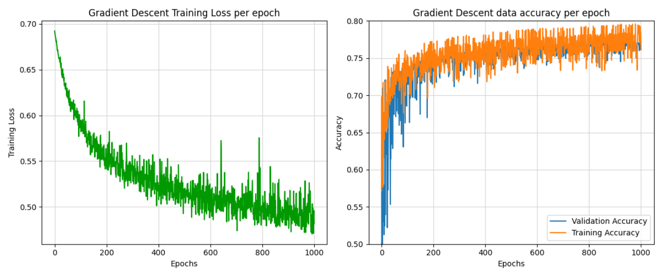


Figure 8: Training loss, and accuracies after a default gradient descent execution (1000 epochs, 10 batches, learning rate of 1)

Similarly we compute the second derivative in Formula 7 and we have that the partial derivative of the cross entropy function with respect to the bias is

$$\frac{\partial J}{\partial b} = -\frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)$$

The gradient computation enables loss minimization by identifying the optimal parameters. The gradients indicate the **steepest ascent** direction at a given point (current weights and bias). By negating these gradients, we obtain the **steepest descent** direction required for optimization. Using a learning rate α to control step size, the parameter update rules become:

$$W \leftarrow W - \alpha\frac{\partial J}{\partial W} \text{ and } b \leftarrow b - \alpha\frac{\partial J}{\partial b}$$

These update rules are fundamental to gradient descent, driving the iterative loss reduction. The complete procedure is formalized in Algorithm 5. Through careful **batch size configuration** [1], we successfully implemented this optimization approach in our codebase.

```
Algorithm 5: GradientDescent($X, y, \alpha, k$)
   Input  : Training data $X \in \mathbb{R}^{m\times n}$, labels $y \in \mathbb{R}^{m\times 1}$,
            learning rate $\alpha$ and max iterations $k$
   Output : Final weights $W \in \mathbb{R}^{n\times 1}$, and bias $b \in \mathbb{R}$
1  Initialize $W$ with small random values
2  Initialize $b$ with a small random value
3  for $i \in \{1, 2, \ldots, k\}$ do
4      Let $z \leftarrow W^T X + b$
5      Let $\hat{y} \leftarrow \sigma(z)$
6      Update $W \leftarrow W - \alpha\frac{\partial J}{\partial W}$
7      Update $b \leftarrow b - \alpha\frac{\partial J}{\partial b}$
8  return $W, b$
```

Several experiments were conducted with varying **learning rates**, **epochs**, and **batch sizes**. The baseline configuration (1000 epochs, 10 batches [size=14,838], learning rate=1) produced the results shown in Figure 8. The loss curve demonstrates successful convergence, while the accuracy plots show both training and validation performance stabilizing near 0.76. However, significant noise in both loss and accuracy metrics may indicate potential optimization challenges.

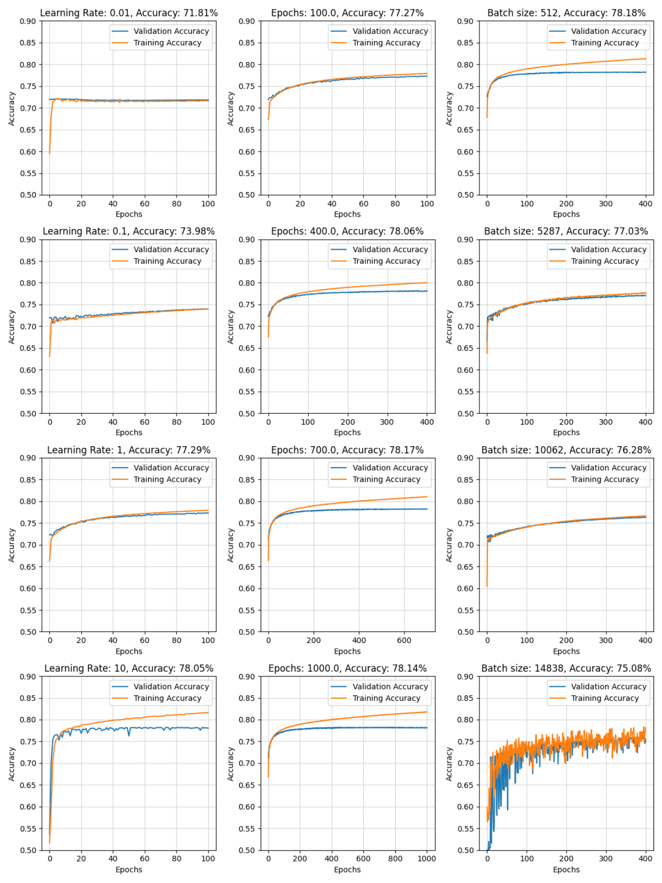


Figure 9: Gradient Descent Experimentation with learning rates, epochs and batch sizes

This prompted systematic experimentation with various hyperparameters, whose results are shown in Figure 9. The experimental design followed three configurations: (1) learning rate variation (100 epochs, 1024 batch size), (2) epoch variation (learning rate=1, 1024 batch size), and (3) batch size variation (learning rate=1, 400 epochs).

Key observations reveal that larger batch sizes increase result noise, explaining the noisy patterns in Figure 8. The optimal configuration achieved **0.7818** accuracy with **400 epochs**, **learning rate=1**, and **batch size=512** and it is represented in the upper right diagram. However, this diagram shows a **5%** gap between training and validation accuracy, suggesting potential overfitting. While this discrepancy is currently acceptable, it warrants monitoring in future iterations.

2.1.3. Base Model. The subsequent phase involved training a default Logistic Regression model using **Scikit-Learn**. This achieved a validation accuracy of **0.7819**, marginally outperforming our previous implementation. Additional evaluation metrics were computed (precision, recall, f1-score) [1], yielding **precision=0.7775**, **recall=0.7899**, and **F1-score=0.7837**.

The model's performance was further analyzed through its **confusion matrix** (Table 4), **learning curve**, and **ROC curve** (Figure 10). The learning curves demonstrate convergence between training and validation accuracies, maintaining a consistent **0.05** gap. This indicates generally good generalization with minor overfitting, as evidenced by slightly better performance on training data.

The ROC analysis revealed an **AUC** of **0.78**, confirming the model's robust discriminative capability. While the observed overfitting is statistically insignificant, it warrants monitoring in subsequent iterations.

AUC refers to the Area Under the Receiver Operating Characteristic (ROC) Curve. It measures the ability of a classification model to distinguish between positive and negative classes. The AUC value ranges from 0 to 1 and the model behaves differently according to the AUC value:

- AUC $= 0.5 \rightarrow$ No better than random gueassing
- AUC $> 0.5 \rightarrow$ Has some predictive power
- AUC $= 1 \rightarrow$ Perfectly distinguishes

The AUC value can be computed by the following formula:

$$\text{AUC} = \sum_{i=1}^{n-1} (x_{i+1} - x_i) \frac{(y_{i+1} + y_i)}{2}$$

where x_i is the **False Positive Rate** and y_i is the **True Positive Rate** at threshold i.

True/Predicted Label	Negative (0)	Positive (1)
Negative (0)	16.406	4.791
Positive (1)	4.452	16.747

Table 4: Confusion Matrix of the Base Model

Base Model Significance. It is crucial to emphasize that these results were obtained using the **default** configurations of both the TF-IDF vectorizer and Logistic Regression model, without any parameter modifications. This unmodified version constitutes our *base* model, which serves two critical purposes in the development process. First, it establishes a reliable performance baseline against which all subsequent modifications can be objectively evaluated. Second, it provides empirical evidence to determine whether specific parameter adjustments actually lead to meaningful improvements, ensuring that optimization efforts are data-driven rather than speculative. The base model's value lies in this comparative framework, which grounds all experimental modifications in measurable performance differences.

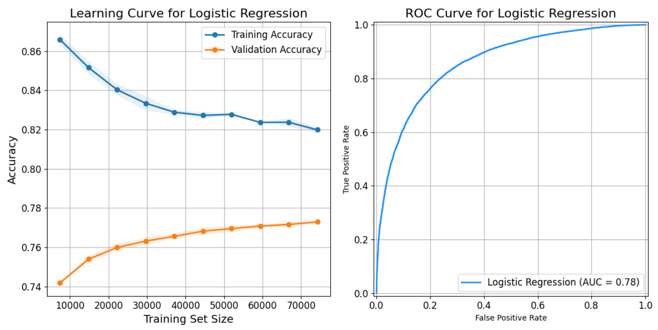


Figure 10: Learning Curves and ROC curve of the Base Model

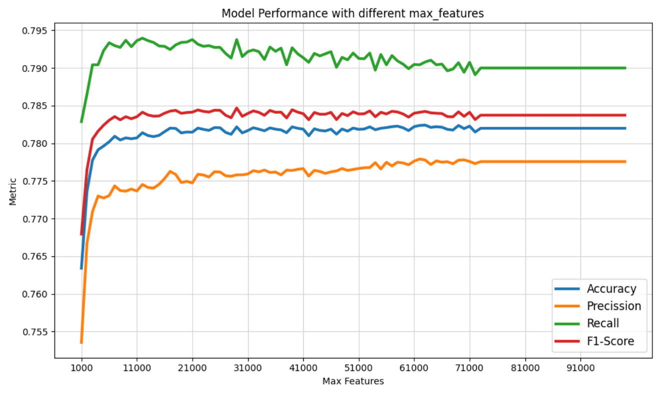


Figure 11: Model Performance with different values for max_features in the TF-IDF vectorizer

2.2 Hyperparameter Tuning

2.2.1. Maximum Features Selection. The default configuration of Scikit-Learn's TF-IDF vectorizer utilizes all document features, resulting in a vocabulary containing 72,713 unique terms in our case. This behavior is controlled by the **max_features** parameter, which constrains the maximum vocabulary size. The complete training data matrix consequently has dimensions:

$$X = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{bmatrix} \in \mathbb{R}^{m \times n} \tag{10}$$

where m represents the number of documents and n the number of features ($\mathbf{n} = \mathbf{72{,}713}$ by default). This yields a training data matrix with 72,713 columns. We systematically evaluate different **max_features** values, specifically testing:

$$\text{max\_features} \in \{1.000, 2.000, 3.000, \ldots, 100.000\}$$

The experimental results for different **max_features** values are shown in Figure 11, which compares all evaluation metrics (accuracy, precision, recall, F1-score). While accuracy generally improves with more features, this relationship is not strictly monotonic. The peak accuracy of **0.7824** occurs at **63,000** features, suggesting an optimal trade-off between feature cover-

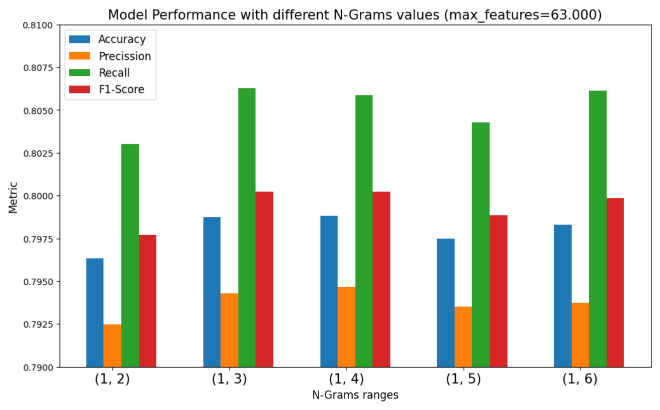


Figure 12: Model Performance with different values for n-gram ranges in the TF-IDF vectorizer

age and model performance. Notably, the marginal gains diminish beyond this point, indicating potential diminishing returns from additional features.

Also the metrics seem to remain constant when **max_features** approaches the vocabulary size of 72,713, as demonstrated in Figure 11. This convergence occurs because the vectorizer's feature space becomes saturated - additional features beyond this point merely represent zero-frequency terms in our corpus. The performance plateau indicates that the model's capacity to extract meaningful patterns is fully utilized at the natural dimensionality of the dataset. Any further increase in **max_features** would constitute redundant parameterization without substantive impact on model performance.

2.2.2. N-Gram Analysis. The **ngram_range** parameter significantly impacts the TF-IDF vectorizer's performance by controlling the word sequence extraction. This tuple parameter defines the minimum and maximum n-gram sizes, where:

- **(1, 1)** extracts only unigrams (single words)
- **(1, 2)** combines unigrams and bigrams (two-word sequences)
- **(1, 3)** adds trigrams (three-word sequences)

While n-grams enhance contextual understanding, they exponentially increase the feature space dimensionality. This expansion leads to:

- Higher space complexity
- Greater computational resource demands
- Longer training times

Our systematic evaluation tested the n-gram ranges:

$$\text{ngram\_range} \in \bigcup_{i=1}^{6} \{(1, i)\}$$

The experimental results for different n-gram ranges are presented in Figure 12. While expanding the n-gram range might suggest potential accuracy gains, our analysis reveals a non-monotonic relationship, with peak performance of **0.7988** accuracy achieved specifically at the **(1, 4)** configuration.

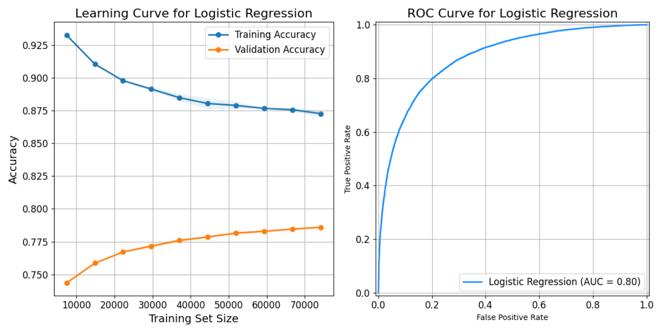


Figure 13: Learning Curves and ROC curve of the Medium Model (401.000 max_features and (1, 4) as an n-grams range)

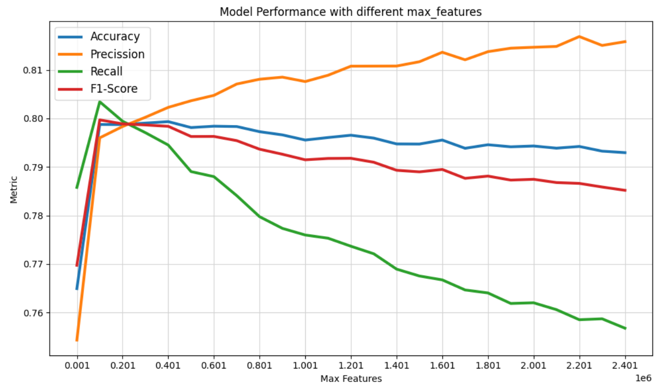


Figure 14: Model Performance with different values for max_features in the TF-IDF vectorizer. Fixed value (1, 4) is used for the n-grams range parameter.

This optimal point emerges from competing factors: intermediate n-grams (1-4) capture meaningful contextual patterns, while higher-order n-grams (5-6) appear to introduce either noise or overfitting without commensurate benefits. The results demonstrate a clear trade-off between contextual information capture and model complexity, where simply increasing the n-gram window size proves counterproductive beyond a certain point. The **(1, 4)** configuration appears to strike the ideal balance for this particular task and dataset.

2.2.3. Optimal Parameter Combination. Building upon our previous findings with optimal n-gram range **(1,4)** and feature count **63,000** (yielding **0.7988** validation accuracy), we investigate potential improvements through parameter combination. The expanded n-gram range increases the feature space dimensionality substantially, with the training matrix now containing **2,481,903** columns ($n = 2{,}481{,}903$ in Formula 10).

Holding the n-gram range fixed at **(1,4)**, we evaluate the following **max_features** values:

$$\text{max\_features} \in \{1.000, 101.000, 201.000, \ldots, 2.482.000\}$$

This systematic exploration tests whether constraining the expanded feature space can maintain accuracy while improving computational efficiency.

The training results using fixed n-gram range **(1,4)** are presented in Figure 14. The metrics demonstrate a characteristic pattern where all measures except precision peak at relatively low feature counts (below 500,000) before gradually declining. This trend is particularly pronounced in recall, which exhibits a sharp decrease beyond its optimal point.

Our primary focus on accuracy reveals a maximum value of **0.7993** at **401,000** features, representing a measurable improvement over previous configurations. However, this performance gain necessitates careful evaluation for potential overfitting, as the declining trends in other metrics beyond certain feature thresholds may indicate deteriorating model generalization. The observed patterns suggest the existence of an optimal feature subset size that balances predictive performance with model complexity.

Figure 13 displays the **learning curves** and **ROC curve** for the current optimal configuration (401,000 max_features, (1,4) n-gram range). The learning curves exhibit behavior qualitatively similar to previous iterations, though with an increased minimum divergence of **0.094** between training and validation accuracy (compared to 0.05 in Figure 10). While this suggests marginally greater susceptibility to overfitting, the absolute difference remains sufficiently small to maintain confidence in the model's generalization capability.

The ROC analysis reveals an improved **AUC** of **0.80** (versus 0.78 previously), demonstrating enhanced discriminative power in classifying tweet sentiment. This 2.6% relative improvement in AUC confirms the effectiveness of our parameter optimization while still operating within acceptable generalization bounds.

2.2.4. Optimization Parameters: Solvers, Regularization, and Penalties. The final experimental phase examines core **Logistic Regression** hyperparameters in Scikit-Learn: **solver** selection, regularization strength (**C**) [1], and **penalty** type. These parameters collectively control the optimization process:

- **Solver**: Determines the numerical optimization algorithm for loss minimization
- **C**: Inverse regularization strength ($C = 1/\lambda$), where lower values increase regularization
- **Penalty**: Specifies the norm (L1/L2) used for regularization

Despite extensive testing across parameter combinations, the model accuracy remained stable at **0.7993**, suggesting either:

- The current configuration already operates near the model's performance ceiling
- These parameters primarily affect training dynamics rather than final performance
- The chosen evaluation metric may be insensitive to these optimizations

2.2.5. Hyperparameter Optimization via Grid Search. While our previous experiments achieved a peak accuracy of **0.7993**, we conducted comprehensive parameter space exploration using Scikit-Learn's **Grid Search** to evaluate potential synergistic effects between parameters. The grid search examined:

Preprocessing	**GradientDescent**	**Scikit-Learn**
Stem+Lem+SWR	0.7993	0.7993
No preprocessing	0.8050	0.8021

Table 5: Accuracy comparison with different preprocessing pipelines

- Regularization strength: $C \in \{0.01, 0.02, \ldots, 1\}$ (20 logarithmically spaced values)
- Penalty types: $\{\text{l1}, \text{l2}, \text{elasticnet}, \text{None}\}$
- Optimizers: $\{\text{lbfgs}, \text{liblinear}, \text{sag}, \text{saga}, \text{newton-cg}\}$

The search employed 2-fold cross-validation with fixed preprocessing parameters (401,000 features, (1,4) n-gram range). The optimal configuration identified was:

- Solver: **lbfgs** (limited-memory BFGS optimization)
- Regularization: $C = \mathbf{1}$ with **l2** penalty

Notably, the best cross-validation score (**0.7858**) and final test accuracy (**0.7993**) suggest that:

- Our previous manual optimization had already identified near-optimal parameters
- The model's performance ceiling may be constrained by fundamental dataset characteristics
- The l2 penalty with moderate regularization provides the best bias-variance tradeoff

2.3 Appendices

Our comprehensive parameter optimization achieved a peak accuracy of **0.7993**, representing the model's performance ceiling under standard preprocessing. However, subsequent analysis revealed unexpected gains through preprocessing simplification:

The improved performance (**+0.57%-0.70%**) without stemming, lemmatization, or stop-word removal (SWR) suggests that:

- These techniques may eliminate meaningful sentiment signals in tweet-length text
- The original preprocessing pipeline could be overly aggressive for this domain
- The model benefits from raw lexical features in short-form social media content

This finding highlights the importance of domain-specific preprocessing optimization, particularly for social media sentiment analysis where conventional NLP techniques may not always apply.

3 Results and Overall Analysis

3.1 Experimental Summary

Steady performance improvements were observed across successive iterations of the model. A baseline dummy model was initially employed, achieving 50% accuracy, which aligned with expectations for random predictions in a balanced dataset. Subsequent enhancements were implemented, beginning with a custom gradient descent optimization method, which resulted in a significant increase to 78.18% accuracy. This served as the first meaningful benchmark. Further refinement was achieved through the adoption of Scikit-Learn's logistic regression implementation, with the base configuration reaching 78.19% accuracy.

The most substantial improvements were realized during systematic hyperparameter optimization. Methodical testing of TF-IDF vectorizer parameters—including maximum features and n-gram ranges—was combined with logistic regression hyperparameter tuning. Through this process, an accuracy of 79.93% was attained. This 1.74 percentage point increase over the initial models highlighted the critical role of careful parameter selection.

3.2 Key Findings

The most insightful discovery was made by revisiting the text preprocessing pipeline. Contrary to standard NLP practices, the removal of stemming, lemmatization, and stop-word elimination was found to produce the highest accuracies: 80.50% with the custom gradient descent method and 80.21% using Scikit-Learn's implementation. A 0.57–0.70 percentage point improvement was observed, suggesting that raw lexical features may retain stronger sentiment signals than normalized text in tweet analysis.

The custom gradient descent method was found to slightly outperform Scikit-Learn's version (80.50% vs. 80.21%). However, comparable overfitting characteristics were observed in both approaches, with training-validation gaps of 10% and 7% respectively.

3.3 Optimal Configuration Analysis

The peak-performing configuration was identified by combining several optimized parameters: a TF-IDF vectorizer with 401,000 maximum features and a (1,4) n-gram range, paired with gradient descent training using a learning rate of 1, a batch size of 512, and 1,000 epochs. The effectiveness of this configuration—particularly when minimal text preprocessing was used—suggested three key insights for short-form social media text analysis:

- The full n-gram spectrum (1–4 words) was found to capture meaningful sentiment patterns
- The 401,000 feature threshold was identified as an optimal balance between contextual relevance and noise reduction
- Raw word forms were shown to preserve critical sentiment signals that might be lost during normalization

Model Configuration	**Accuracy (%)**
Random predictions (baseline)	50.00
Custom Gradient Descent (initial)	78.18
Scikit-Learn LR (base)	78.19
Optimized TF-IDF + LR	79.93
Simplified preprocessing pipeline	80.50

Table 6: Model performance progression through experimentation

While clear progress was demonstrated, potential for improvement remained through methods like error analysis, ensemble techniques, or alternative feature representations. A consistent 7–10% performance gap between training and validation accuracy was noted, indicating opportunities for stronger regularization strategies.

4 Bibliography

References

[1] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models*. 3rd edition, 2025. Online manuscript released January 12, 2025.

[2] Kaare Brandt Petersen, Michael Syskind Pedersen, et al. The matrix cookbook. *Technical University of Denmark*, 7(15):510, 2008.