## 1. Introduction

This project focuses on sentiment analysis of tweets using Natural Language Processing (NLP) techniques. The goal is to build a classification model that can distinguish between positive and negative sentiments in tweets. The dataset used in this project was provided by the instructor and is sourced from the nltk.corpus.twitter\_samples, which contains pre-labeled tweets making it ideal for supervised machine learning tasks.

Sentiment analysis is an important application of NLP and is widely used in industry for brand monitoring, customer feedback analysis, and political opinion tracking. In this project, I performed several key steps including exploratory data analysis, preprocessing, vectorization, modeling, and evaluation to identify the best-performing sentiment classifier.

## 2. Exploratory Data Analysis (EDA)

After importing the necessary libraries and resources, I loaded the sentiment-labeled tweet data using the NLTK twitter\_samples corpus. This dataset provides a balanced set of tweets that are pre-labeled as either positive or negative, which makes it ideal for binary classification tasks.

Data Loading

The dataset was split as follows:

* 5,000 positive tweets
* 5,000 negative tweets

I combined the tweet texts into a single tweets list and created a corresponding labels list, where 1 indicates a positive sentiment and 0 indicates a negative sentiment. I combined these two sets into a single list of tweets and created a corresponding list of binary sentiment labels, where a value of 1 represents a positive tweet and 0 represents a negative one. This resulted in a balanced dataset of 10,000 tweets in total.

### Initial Inspection

To get a feel for the content, I printed out the first 10 tweets from both the positive and negative sets. This gave me insight into the tone, structure, and use of informal elements like emojis, hashtags, mentions, and URLs—common features in social media text.

* Positive tweets often contained emojis like ":)", expressions of excitement ("yeaaaah yippppy!!!"), gratitude, and compliments.
* Negative tweets were marked by sad emojis (":(", ":-("), expressions of disappointment, frustration, or melancholy.

### Label Distribution

To ensure that the dataset is balanced (which is crucial for model fairness), I used the Counter class to verify the label counts.

This confirmed that the dataset is perfectly balanced, which simplifies the modeling process since we don’t need to apply class rebalancing techniques like oversampling or undersampling.

# Preprocessing

Text preprocessing is a crucial step in Natural Language Processing, especially when working with social media data like tweets. Tweets often contain noise such as URLs, hashtags, mentions, emojis, and informal language, which can negatively affect model performance if not handled properly. Therefore, I began preprocessing the text to clean and standardize it for analysis.

### Step 1: Removing URLs

The preprocessing phase began with the removal of URLs from the tweet texts. Tweets often include hyperlinks, usually pointing to external websites or media content, which generally do not convey sentiment information relevant to classification tasks. To eliminate this noise, I applied a regular expression that targets all URL patterns starting with either "http" or "https" followed by any sequence of non-space characters. This expression ensured that both full and shortened links were removed from the data. For example, a tweet containing a promotional or news-related link was cleaned by extracting only the textual content, leaving out the URL. This step was essential to reduce irrelevant variation in the data and to prevent the model from learning patterns associated with frequently repeated or unique URLs, which hold little to no predictive value in determining sentiment.

### Step 2: Removing Punctuation

Following the removal of URLs, I proceeded to clean the tweets by eliminating punctuation marks. Punctuation, while useful for human readability and sometimes expressive tone, generally adds noise in text classification tasks unless explicitly handled. Since this project focused on traditional feature extraction methods like TF-IDF, I chose to remove all non-alphanumeric symbols to simplify the text and improve consistency across tokens. This was done using a regular expression that filters out any character that is not a word character or whitespace. As a result, characters such as exclamation marks, colons, hashtags, and even emoticons like ":)" were removed. For instance, a tweet like "CONGRATS :)" was transformed into "CONGRATS". While punctuation and emoticons may carry emotional meaning, they were not explicitly modeled in this project and were therefore excluded to maintain a clean and uniform input for downstream processing. This step helped standardize the dataset and ensured that the vectorizer would focus solely on meaningful textual content.

### Step 3: Tokenization

With the text cleaned of both URLs and punctuation, I moved on to the tokenization process. Tokenization involves breaking down each tweet into individual words, known as tokens, which serve as the fundamental units for analysis in most natural language processing tasks. Given the brevity of tweets, I opted to tokenize at the word level rather than at the sentence level. For this purpose, I utilized NLTK’s TweetTokenizer, which is specifically designed to handle the informal and idiosyncratic nature of Twitter content. The tokenizer was configured to convert all characters to lowercase (preserve\_case=False), remove Twitter handles (strip\_handles=True), and normalize elongated words (reduce\_len=True) by collapsing repeated characters. These settings ensured that tokens such as “Sooooo” and “sooo” would be treated uniformly, and user mentions like “@user123” would be removed as they do not contribute to sentiment. The output of this step was a list of tokenized tweets, where each tweet was represented as a list of lowercase words, stripped of usernames and repetitive characters. This stage was crucial for preparing the data for the subsequent removal of stopwords and for ensuring consistency across the dataset.

### Step 4: Removing Stopwords

After tokenization, I proceeded to remove stopwords from each tweet. Stopwords are commonly used words in the English language such as “the”, “is”, “in”, “at”, and “and”, which typically carry little meaningful information in sentiment analysis. These words appear frequently across all types of text and do not contribute significantly to determining the emotional tone of a sentence. Using NLTK’s built-in list of English stopwords, I filtered out these low-value terms from each tokenized tweet. The removal of stopwords not only improved the clarity of the data by retaining only sentiment-bearing words but also reduced the overall dimensionality of the feature space. As a result, each tweet was distilled into a more concise and informative list of keywords. For instance, phrases such as “please call our contact centre and we will be able to assist you” were reduced to essential terms like “please”, “call”, “contact”, “centre”, “able”, and “assist”. This process sharpened the focus of the dataset and enhanced its suitability for downstream tasks such as stemming and feature extraction.

### Step 5: Stemming

Once stopwords were removed, I applied stemming to further normalize the text data. Stemming is the process of reducing words to their root or base form, which helps group together different variations of a word that carry the same core meaning. For this task, I used the Porter Stemmer provided by the NLTK library, which is one of the most commonly used stemming algorithms due to its efficiency and simplicity. Each token in the filtered tweets was passed through the stemmer, resulting in words like “engaged”, “engaging”, and “engagement” being reduced to the common root “engag”. This reduction significantly decreased the vocabulary size and improved the model’s ability to generalize across similar word forms. For example, the phrase “amazing track” was transformed to “amaz track”, and “verified profile” became “verifi profil”. While stemming can sometimes lead to non-standard word forms, it proved useful in standardizing the textual data for feature extraction. This step completed the normalization of the tweet content and laid the groundwork for the final transformation into model-ready input.

### Step 6: Reconstructing Cleaned Tweets

After completing the stemming process, the final step in the preprocessing pipeline was to reconstruct each tweet from its list of processed tokens back into a single string of text. This was accomplished by joining the stemmed tokens using whitespace, resulting in a list of cleaned tweets represented as space-separated strings. This step was essential because many machine learning vectorization tools, such as TF-IDF, expect input in the form of raw text rather than lists of individual tokens. Reconstructing the text preserved the sentence-like structure of each tweet while retaining only the most meaningful, normalized words. For example, the processed list ['amaz', 'track', 'scotland'] became the cleaned string “amaz track scotland”. By the end of this stage, each tweet had been fully standardized and stripped of noise, including URLs, punctuation, stopwords, and morphological variation. This prepared the dataset for the next phase of feature extraction and modeling.

## 5. Modeling

With the tweets fully preprocessed and reconstructed into cleaned strings, I moved on to preparing the data for model training and evaluation by performing a train-test split. This step involved dividing the dataset into two subsets: one for training the models and another for testing their performance on unseen data. Using the train\_test\_split function from scikit-learn, I allocated 70 percent of the data to the training set and 30 percent to the test set. This division ensured that the models would learn from a substantial portion of the data while still being evaluated on a separate, unbiased sample. A random seed of 42 was used to make the results reproducible across multiple runs. The split preserved the label distribution, ensuring that both the training and test sets contained a balanced number of positive and negative tweets. This separation was crucial for assessing how well the models would generalize beyond the data they were trained on and provided a robust foundation for building and validating sentiment classifiers.

### Feature Extraction using TF-IDF

Following the data split, I transformed the raw textual data into numerical features using Term Frequency–Inverse Document Frequency (TF-IDF) vectorization. TF-IDF is a widely used technique in text mining that assigns weights to words based on their frequency within a document relative to their frequency across the entire corpus. This method helps highlight words that are both frequent in individual tweets and distinctive compared to the broader dataset. I employed scikit-learn’s TfidfVectorizer with the parameter max\_features set to 5000, which limited the feature space to the 5,000 most informative tokens. This not only improved computational efficiency but also helped reduce overfitting by ignoring infrequent or irrelevant words. The vectorizer was fit on the training data and subsequently used to transform both the training and testing sets into sparse matrices, where each row represented a tweet and each column corresponded to a weighted word feature. This representation captured the semantic importance of terms while preserving the structure required for input into machine learning models. At this stage, the data was fully transformed into a format suitable for classification.

### Logistic Regression

With the data successfully vectorized, I evaluated four different machine learning models for tweet sentiment classification. The first model tested was Logistic Regression, a widely-used baseline classifier for binary tasks due to its simplicity, interpretability, and effectiveness in high-dimensional feature spaces like those created by TF-IDF. The model learned to associate word weights with sentiment labels using a logistic loss function. When evaluated on the test set, Logistic Regression achieved an accuracy of 75.03%, with balanced precision, recall, and F1-scores of 0.75 across both positive and negative classes. While it occasionally misclassified tweets with ambiguous or minimal text, such as “one day,” it handled clear sentiment indicators reliably. Overall, its consistent performance across metrics made it a strong candidate for further use.

A blue squares with white numbers

AI-generated content may be incorrect.

### Naïve Bayes

The second model explored was Multinomial Naive Bayes, a probabilistic classifier well-suited for text classification tasks, particularly when working with sparse input data like TF-IDF vectors. This model achieved an accuracy of 74.5%, closely matching that of Logistic Regression. It showed slightly higher recall on negative tweets and higher precision on positive ones, with F1-scores again around 0.74–0.75. Sample predictions demonstrated that Naive Bayes handled polarized tweets well but was more likely to misclassify tweets with complex or subtle sentiment. Despite its marginally lower accuracy, Naive Bayes remained an efficient and computationally lightweight option.

Model and vectorizer saved.

A blue squares with white text

AI-generated content may be incorrect.

### Support Vector Machine

The third model evaluated was Linear Support Vector Machine (SVM), implemented using scikit-learn’s LinearSVC. SVMs are known for their strong performance in high-dimensional settings and work by maximizing the decision boundary between classes. The LinearSVC model achieved an accuracy of 73.63%, slightly below that of Logistic Regression and Naive Bayes. Precision and recall values were balanced, each producing F1-scores of 0.74, but the model showed a slight tendency to overpredict the negative class. While Linear SVM proved to be a reliable and robust classifier, it did not surpass the performance of the simpler logistic model, possibly due to the nature of the sparse TF-IDF features.

A blue squares with white text

AI-generated content may be incorrect.

### Extreme Gradient Boosting (XGBoost)

The fourth and final model tested was Extreme Gradient Boosting (XGBoost), a powerful ensemble technique known for its superior performance on structured data. Implemented using the XGBClassifier, this model achieved an accuracy of 72.27%, the lowest among all models tested. It demonstrated high precision (0.79) for the positive class but lower recall (0.62), and the inverse pattern was observed for the negative class. The confusion matrix revealed that the model leaned toward overpredicting negative sentiment. These results suggest that while XGBoost is typically a strong performer, its effectiveness in this context may have been limited by the linear nature of the data and the sparsity introduced by TF-IDF vectorization.

A blue squares with numbers and a graph

AI-generated content may be incorrect.

## Conclusion

After evaluating all four models using the same dataset and feature representation, Logistic Regression was selected as the most reliable and balanced model for this sentiment classification task. It offered the highest overall accuracy and exhibited stable performance across all core metrics, without favoring one sentiment class over the other. To preserve this model for future use and support deployment, I saved both the trained Logistic Regression classifier and the associated TF-IDF vectorizer using Python’s joblib library. The model was stored in a file named sentiment\_model.pkl, and the vectorizer in tfidf\_vectorizer.pkl. This ensures that any new tweet can be processed and classified using the exact same pipeline established during development. This step marked the final transition of the project from experimental analysis to a deployable, reusable NLP application.