**Sentiment analysis**

**Tahsin Raihan**

**Dept of CSE**

**Rajshahi University of Engineering & Technology**

**Introduction**

Sentiment analysis is a vital field within Natural Language Processing (NLP) that delves into extracting subjective information from text data. It involves categorizing and identifying opinions that are expressed in a text to determine the writer's thought and attitude , it is often categorized as positive, negative, or neutral. Usually we categorize it into these major three categories but we can modify it into more categories as sad , happy , angry , excited and so on . The increasing reliance on digital communication in modern era has made sentiment analysis an essential tool for businesses, researchers, and decision-makers. Its applications range from monitoring social media trends & analyzing customer feedback to enhancing user experiences .It also helps to extract features from data to categorise the customer and finding the outlier .

This project offers advanced machine learning techniques to build a robust sentiment analysis system. By employing a stacked ensemble model, we combine the strengths of various algorithms to deliver higher predictive accuracy. We have analysed different machine learning algorithms and analysed the accuracy . After complementing this model, a user-friendly streamlit interface enables real-time sentiment prediction. This will give the users a real time experience about how the sentiment analysis engine is working . This documentation explores every aspect of the project, from data preprocessing and modeling to deployment and evaluation, highlighting the practical applications and challenges of sentiment analysis.

.

**Objectives**

The key aims of this project are as follows:

1. Developing a robust sentiment analysis model using stacked ensemble learning.
2. Leverage multiple base classifiers to enhance predictive accuracy and reliability.
3. Design an intuitive front-end interface using Streamlit for real-time user interaction.
4. Demonstrate the practical applicability of machine learning techniques for text analysis in natural language processing field .

**Dataset Description**

We have used the dataset from Kaggle .Basically , the dataset is collected from twitter . We have labels that follows corresponding text . The dataset used in this project comprises text samples labeled in three category as **positive**, **negative**, or **neutral**. Each entry contains a short text snippet and its corresponding sentiment label.

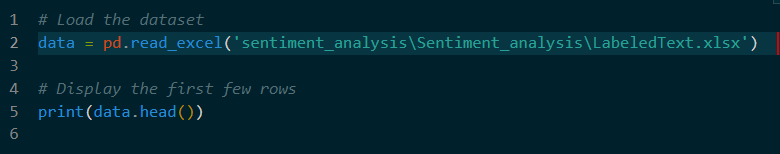
Key properties of the dataset include as follows :

* There is a balanced representation of all three sentiment classes( approximately all three category is equal in the dataset ) .
* The text in the dataset ( natural language) , requiring significant preprocessing to remove noises .
* A total of **4070 samples**, split into training and testing sets to evaluate model performance
* We have divided the total dataset into two segment , training (80%) and testing (20%) datasets to evaluate model performance.

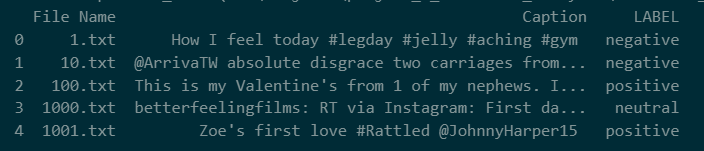
**Implementation Steps**

**1. Data Loading**

The dataset is loaded using Python's Pandas library. A try-except block ensures that appropriate error handling is in place in case of file-related issues. This makes the code robust and user-friendly, providing clear feedback when the file is missing or incorrectly specified.



The output of this segment shows a few rows which demonstrates the columns in the dataset as follows :

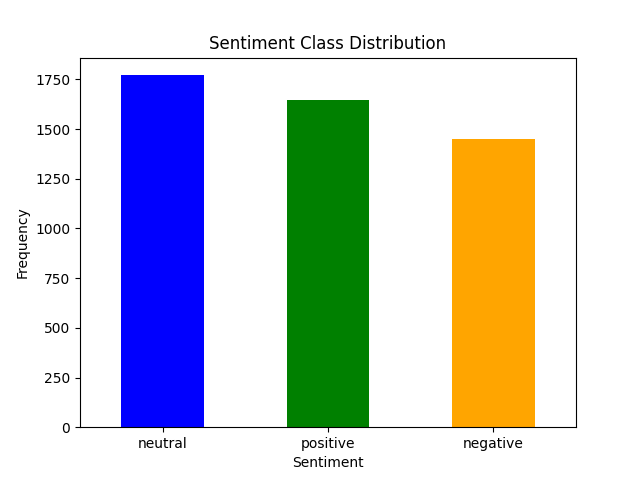


**Detailed Analysis of the data :**

**Counting the occurrences of each sentiment class :**

This code segment visualizes the distribution of sentiment classes in the dataset using a bar chart, providing insights into the frequency of each category ( positive , negative , neutral ). The code begins by importing necessary libraries as libraries are essential for creating and customizing visualizations. It computes the count of each unique sentiment class in the LABEL column of the pandas DataFrame named data using the value\_counts() function, which returns a series where the index represents the sentiment labels and values represent their respective counts. A bar chart is generated with class\_counts.plot(kind='bar'), where the bars are styled with colors (blue, green, and orange). The chart is enhanced with a title, labeled axes (Sentiment for the x-axis and Frequency for the y-axis), and horizontal x-axis labels for readability. The chart is saved as an image file, sentiment\_distribution.png, using plt.savefig() and displayed on the screen with plt.show(). This ensures accessibility in both graphical and non-graphical environments. The code requires a pandas DataFrame containing a column named LABEL and assumes that the matplotlib library is installed. The bar chart effectively highlights class imbalances, offering valuable insights for further data preprocessing or model evaluation in sentiment analysis tasks.





**2. Text Preprocessing**

Preprocessing is a critical step to clean and standardize the raw text data. As we have observed in the text in natural language , there are a lots of special characters , stop words , mixture of lower case and upper case . Therefore , in this project, a custom preprocess\_text ( ) function was implemented to:

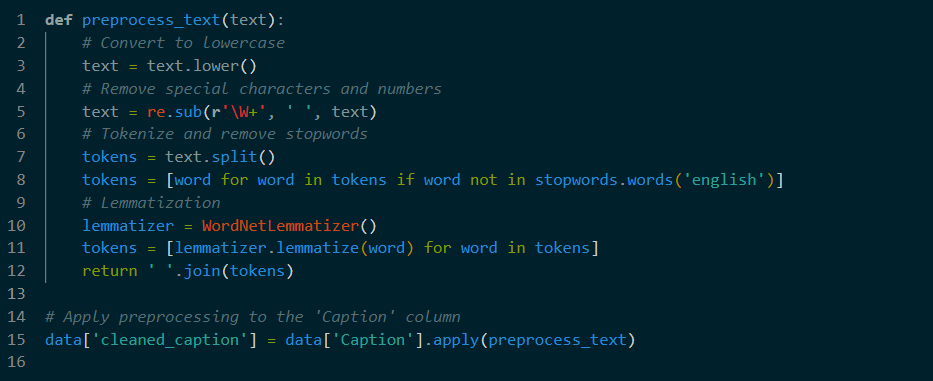
 **Convert to Lowercase:** At first , we standardized the text by converting it to lowercase.

 **Remove Special Characters:** Used regular expressions to eliminate symbols and digits as they are not relevant in our project .

 **Tokenization:** We split the text into individual words to analyse it more accurately .

 **Stop-word Removal:** We have filtered out common words like "is," "the," and "and" using NLTK's stopword list as they are meaningless in terms of analysing sentiment .

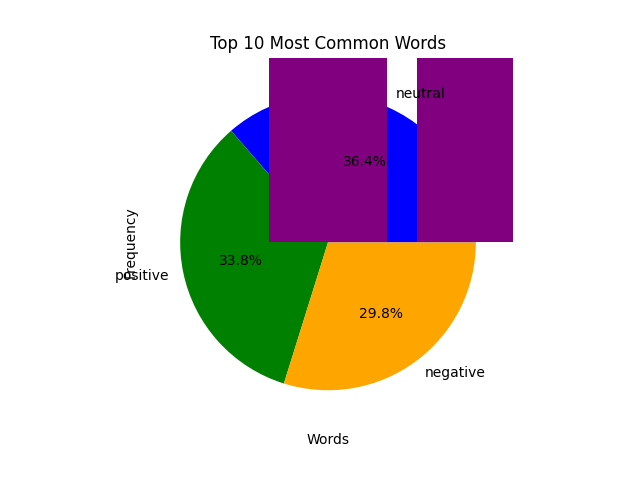
 **Lemmatization:** In this step , reduced words to their base forms using WordNetLemmatizer. Like swimming to swim .



**Top 10 most common words:**

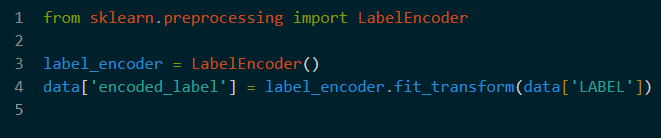


This code analyzes a text dataset to identify and visualize the top 10 most common words using a bar chart. It begins by importing Counter from the collections module for counting word occurrences and word\_tokenize from the nltk library for tokenizing text. The Caption column of the data DataFrame is processed to combine all text into a single string, which is then split into individual words. The Counter function calculates the frequency of each word, and the most\_common(10) method retrieves the 10 most frequently occurring words and their counts. These results are unpacked into two variables, words and counts. Using matplotlib.pyplot (plt), a bar chart is plotted with the words on the x-axis and their frequencies on the y-axis, styled with a purple color for the bars. The chart is customized with a title, axis labels, and rotated x-axis labels to ensure readability. The final chart is saved as a PNG file named common\_words\_bar\_chart.png and displayed. This visualization provides insights into the most prominent words in the dataset, which can be useful for text analysis tasks such as trend detection or feature engineering. The code assumes nltk and matplotlib are installed and the dataset is preprocessed appropriately.



##### ****3. Label Encoding****

Sentiment labels (positive, neutral, negative) are categorical in nature. These are encoded into numerical representations using the LabelEncoder class. Positive, neutral, and negative sentiments were mapped to 2, 1, and 0, respectively. This encoding allows machine learning algorithms to process the labels effectively. As we are classifying in only three categories , we have used the numerical value 0 , 1 , 2 .

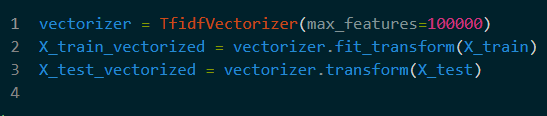


**4. Feature Extraction**

It is the most crucial part for our project. The **TfidfVectorizer** plays an important role in our project by transforming textual data into numerical feature representations. This approach highlights distinctive terms by assigning greater weights to words that occur frequently within a specific document but are uncommon throughout the entire dataset. It ensures that the model focuses on the most relevant terms while minimizing the impact of frequently occurring but less meaningful words.

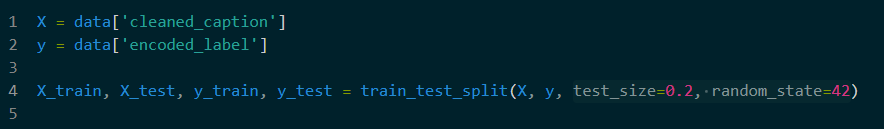
Parameters used for feature extraction :

* In TfidVectozer we set , max features = 100000 to restrict the dimensionality .If we don’t restrict the dimensionality , it will end up creating a huge set of vectors , creating unnecessary calculation .
* ngram\_range=(1, 2) to capture both unigrams and bigrams.
* Stopwords excluded to reduce noise.



**5. Splitting Data**

We split the dataset into training and testing sets using an 80-20 ratio. This ensures that the model is trained on a large portion of the data while retaining sufficient samples for unbiased evaluation. In geneal ,the 80 -20 ratio is said to be the standard ratio . We have used train\_test\_split for this task .



**6. Building the Stacking Classifier**

**The core innovation** in this project lies in the use of *ensemble learning*. Instead of relying on a single model, a stacking classifier combines predictions from multiple base models using a meta-model for final predictions. It will learn the pattern which gives the highest accuracy in each step using the base models .

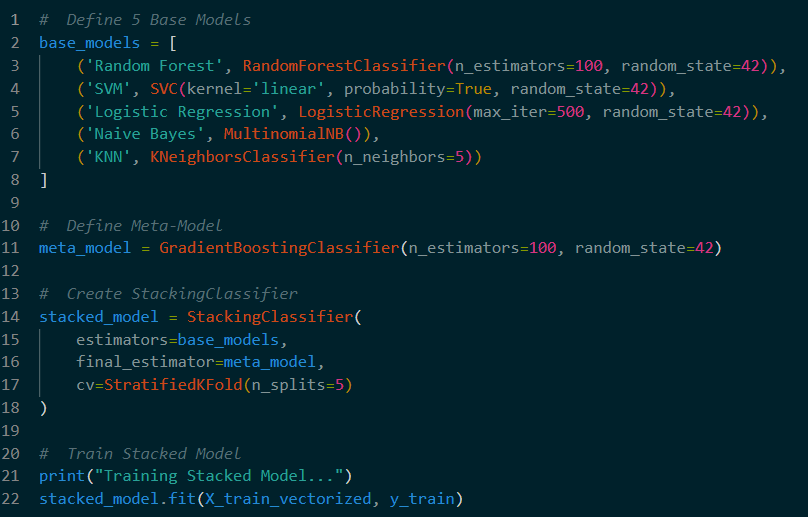
**Base Models**

1. **Random Forest**: A robust ensemble technique that uses decision trees to provide stable predictions.
2. **Support Vector Machine (SVM)**: Effective for high-dimensional data, SVM separates classes using a hyperplane.
3. **Logistic Regression**: A linear classifier suited for binary and multiclass classification.
4. **Naive Bayes**: Suitable for text data, leveraging conditional probabilities for classification.
5. **K-Nearest Neighbors (KNN)**: A simple, instance-based learning technique.

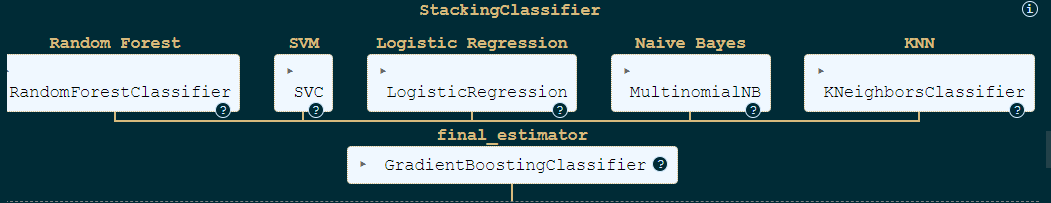
**Meta-Model**

The meta-model is a **Gradient Boosting Classifier**, which is adept at capturing complex patterns from base model predictions.

The stacking classifier is implemented using the StackingClassifier class, with a 5-fold cross-validation strategy (cv=StratifiedKFold(n\_splits=5)) to ensure robustness.



The output shows the architecture of the meta model as follow :



**7. Stacking Classifier**

The StackingClassifier class aggregated predictions from base models, refined by the meta-model. A 5-fold StratifiedKFold was employed for cross-validation which made this meta model robust and effective .

**8. Model Training**

The stacked model is trained on the vectorized training data using the fit () method. This process aggregates predictions from all base models and refines them using the meta-model. Training logs are printed to monitor progress.

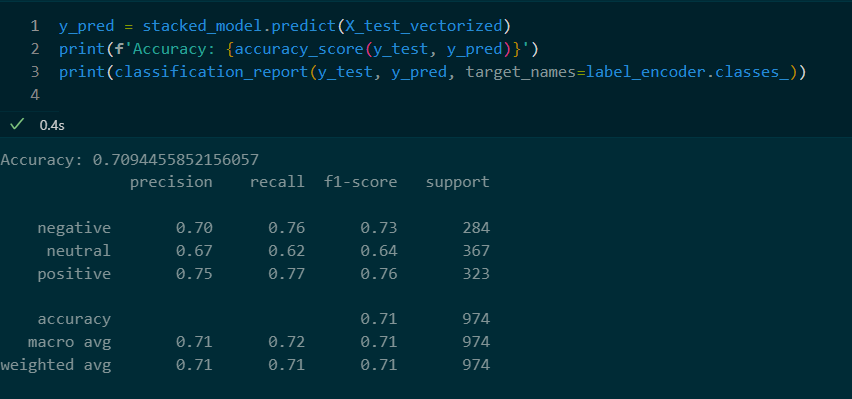
Performance metrics:

Accuracy : It means proportion of true results among all cases .

F1 score : It means , harmonic mean of precision and recall .

Precision : It means the proportion of true positives among predicted positives .

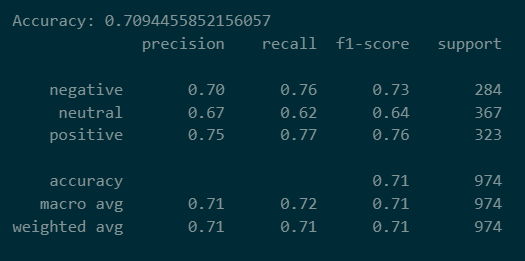
Recall :It means proportion of true positives among actual positives .



**8. Model Evaluation**

The trained model is evaluated on the testing set using metrics such as:

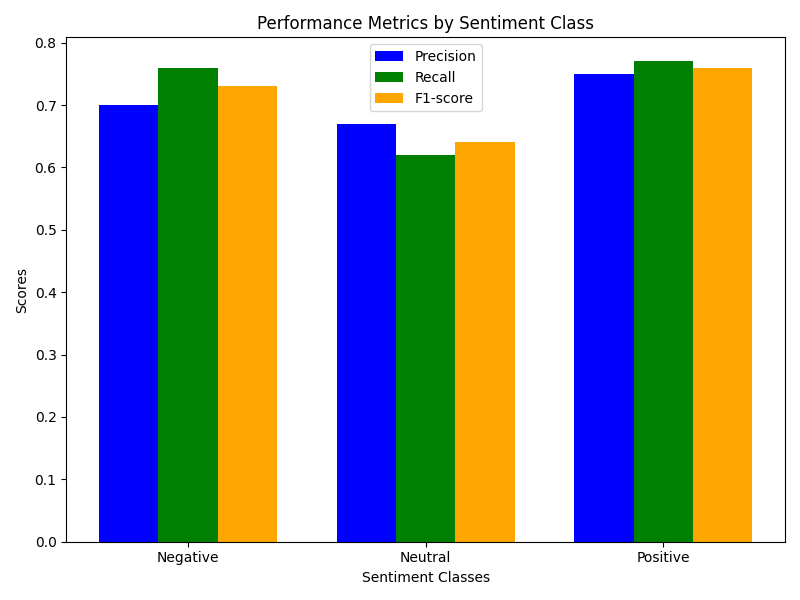
* **Accuracy**: The overall correctness of predictions which is 70.94 %.
* **Precision, Recall, and F1-Score**: Detailed performance metrics for each sentiment class.



**Accuracy graph :**

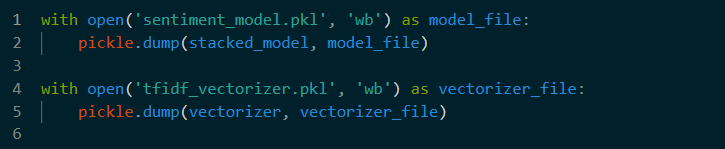
This code generates a grouped bar chart that visually compares performance metrics—Precision, Recall, and F1-score—for three sentiment categories: Negative, Neutral, and Positive. The data for these metrics is stored in separate lists, and the sentiment categories serve as labels for the x-axis. Using matplotlib.pyplot as plt, bars are plotted for each metric, with each set of bars offset horizontally to ensure clarity. Colors are assigned to the bars to represent different metrics: blue for Precision, green for Recall, and orange for F1-score. The chart includes x-axis and y-axis labels, a title summarizing the data, and a legend for clarity. The x-axis tick marks are aligned with the center of the grouped bars, displaying the sentiment categories. To ensure proper spacing and avoid overlap among chart elements, the layout is adjusted with plt.tight\_layout(). Finally, the chart is saved as a file named Accuracy\_chart.png and displayed in the execution environment. This visualization is valuable for analyzing and comparing model performance for each sentiment category, especially in tasks like classification or sentiment analysis. The code is designed for Python environments where matplotlib is available and the input data accurately reflects model evaluation metrics





**9. Saving the Model**

The trained model and vectorizer are serialized using Python's pickle library. These files (stacked\_sentiment\_model.pkl and tfidf\_vectorizer.pkl) are loaded during runtime to predict new data, eliminating the need to retrain the model. Saving the model is important , as we can use this model for further experiment , and also it can be run as engine in the backend when we will build our user interface using streamlit .



**10 . Analyzing each model individually  :**

The code uses scikit-learn, a popular machine learning library for Python, to implement and evaluate models:

* **accuracy\_score:** Computes the accuracy of model predictions.
* **RandomForestClassifier:** Implements a decision-tree-based ensemble method.
* **SVC:** Implements a Support Vector Classifier.
* **LogisticRegression:** Implements a linear model for binary classification.
* **MultinomialNB:** Implements the Naive Bayes algorithm, effective for text data.
* **KNeighborsClassifier:** Implements the K-Nearest Neighbors algorithm.

A dictionary named models is used to store multiple machine learning classifiers. Each key is a string representing the model's name, and the value is the initialized model object.

* **Random Forest Classifier**: A decision-tree ensemble method with 100 trees, providing robustness and resistance to overfitting.
* **Support Vector Machine (SVM)**: A linear kernel SVM for linearly separable data.
* **Logistic Regression**: A probabilistic linear classifier with a maximum of 500 iterations for convergence.
* **Naive Bayes (MultinomialNB)**: A text-specific probabilistic classifier based on word occurrence frequencies.
* **K-Nearest Neighbors (KNN)**: A proximity-based classifier with 5 neighbors.

**Dictionary to Store Accuracies**

A dictionary, accuracies, is created to store accuracy scores for each model after evaluation.

**Training and Evaluating Each Model**

A for loop iterates through the models dictionary to train and evaluate each model:

1. **Train the model**: The .fit() method is used to train the model on X\_train\_vectorized and y\_train.
2. **Predict on test data**: The .predict() method is used to make predictions on X\_test\_vectorized.
3. **Calculate accuracy**: accuracy\_score() compares y\_test (true labels) and y\_pred (predicted labels) to compute the accuracy.

Each model's accuracy is stored in the accuracies dictionary and displayed immediately.

*for name, model in models.items():*

*model.fit(X\_train\_vectorized, y\_train)*

*y\_pred = model.predict(X\_test\_vectorized)*

*accuracy = accuracy\_score(y\_test, y\_pred)*

*accuracies[name] = accuracy*

*print(f"{name} Accuracy: {accuracy:.2f}")*

**Displaying All Accuracies**

At the end, the code prints a summary of all models' accuracies stored in the accuracies dictionary.

*print("\nAll Model Accuracies:")*

*for model, accuracy in accuracies.items():*

*print(f"{model}: {accuracy:.2f}")*

**Explanation of Each Machine Learning Algorithm**

**Random Forest Classifier**

* **Algorithm**: Uses an ensemble of decision trees trained on random subsets of data and features.
* **Strengths**:
  + Robust to overfitting due to averaging.
  + Handles non-linear relationships well.
  + Suitable for both classification and regression tasks.
* **Parameters**:
  + n\_estimators=100: Specifies the number of decision trees in the forest.
  + random\_state=42: Ensures reproducibility by controlling randomness.

**Support Vector Machine (SVM)**

* **Algorithm**: Finds a hyperplane that maximally separates data into classes.
* **Strengths**:
  + Works well for high-dimensional data.
  + Effective in cases where the classes are linearly separable.
* **Parameters**:
  + kernel='linear': Specifies a linear kernel for the hyperplane.
  + random\_state=42: Ensures consistent results.

**Logistic Regression**

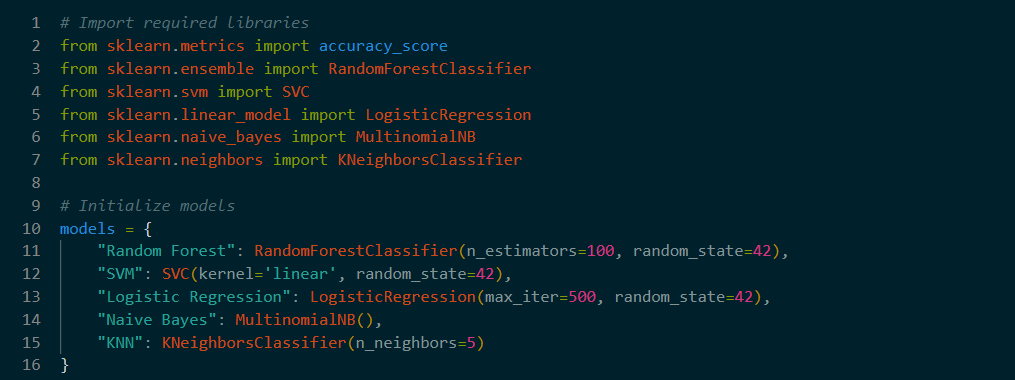
* **Algorithm**: Models the probability of class membership using a logistic function.
* **Strengths**:
  + Efficient for binary classification.
  + Simple yet effective for linearly separable data.
* **Parameters**:
  + max\_iter=500: Ensures the algorithm converges for complex datasets.
  + random\_state=42: Guarantees consistent training results.

**Naive Bayes (MultinomialNB)**

* **Algorithm**: Applies Bayes' theorem with an assumption of feature independence.
* **Strengths**:
  + Fast and efficient for text classification tasks.
  + Performs well with sparse data like TF-IDF features.
* **No tunable parameters** in this instance as it directly works on word counts.

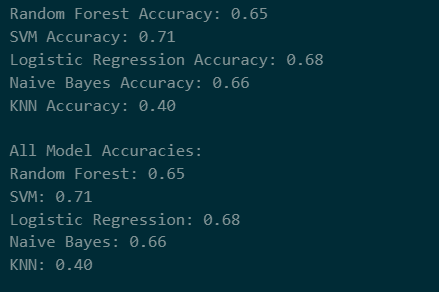
**K-Nearest Neighbors (KNN)**

* **Algorithm**: Predicts class labels based on the majority class of the nearest k neighbors in feature space.
* **Strengths**:
  + Simple to understand and implement.
  + Non-parametric, making no assumptions about data distribution.
* **Parameters**:
  + n\_neighbors=5: Considers the 5 closest data points for prediction.





**Accuracy of each model :** We can clearly see that , none of the model has exceeds 68% accuracy except SVM . It clearly shows , our meta model performance it better than using a single model .

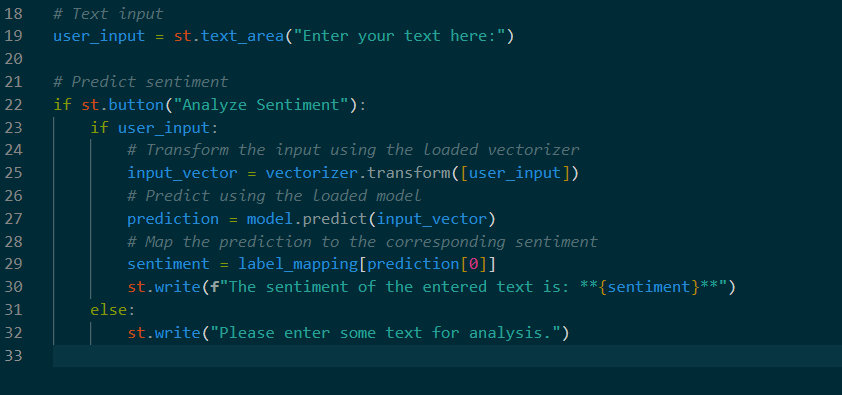


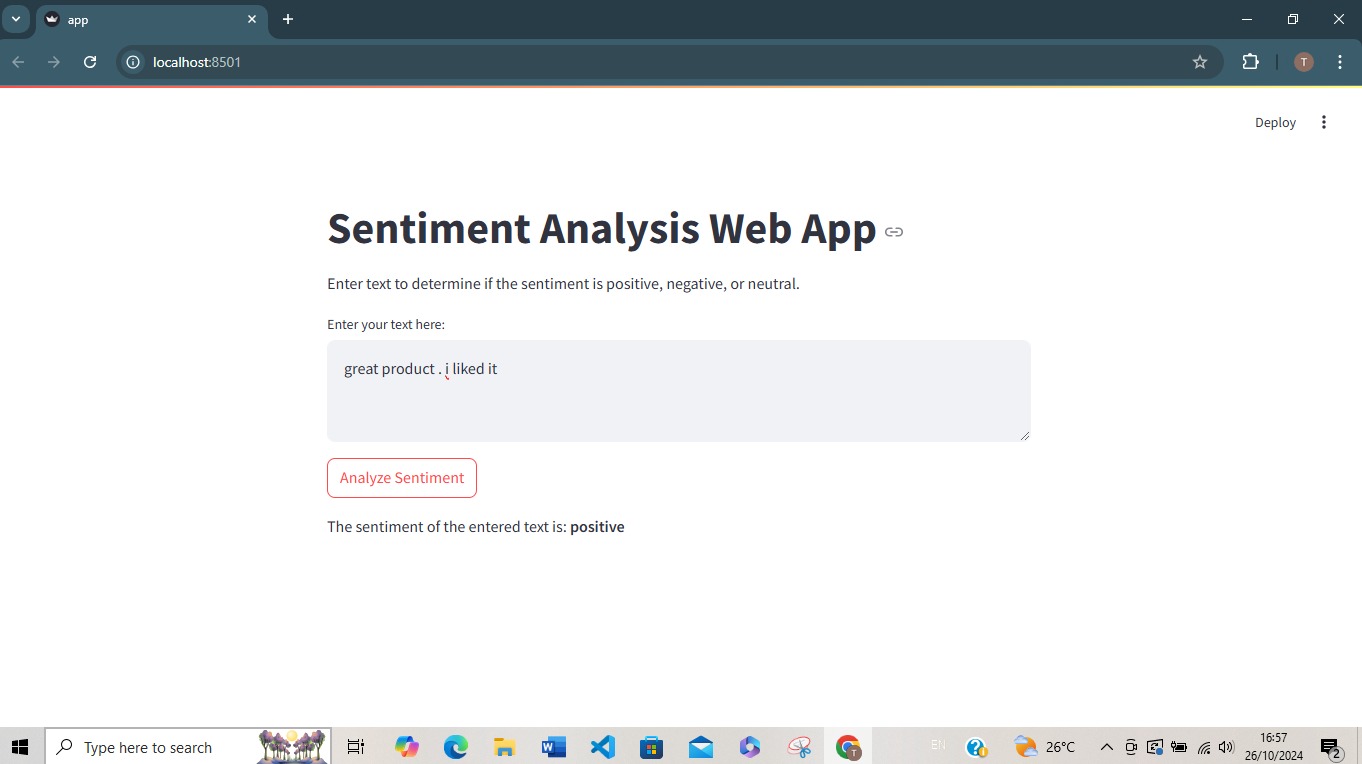
**11. Front-End Deployment with Streamlit**

The project includes an interactive Streamlit application for sentiment prediction. Users can input text through a web interface and the system displays the predicted sentiment. Key features of the app include:

* A clean and responsive interface.
* Instant prediction output using the saved model .
* User-friendly error messages and warnings depending on the input text .







**Challenges and Solutions**

1. **Class Imbalance**: Imbalance in sentiment labels was initially an issue but was mitigated by using a well-balanced dataset.
2. **Text Noise**: Special characters and stopwords added noise. Comprehensive preprocessing addressed this challenge.
3. **Model Complexity**: The stacked model, with its multiple components, required careful debugging and hyperparameter tuning.

**Future Work :**

1. **Incorporate Deep Learning**: Models like LSTMs or Transformers could improve accuracy on complex datasets.
2. **Multilingual Support**: Extend the application to handle non-English text.
3. **Sentiment Strength**: Quantify the strength of positive or negative sentiments.

**References :**

**Books**

1. Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.
2. Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer.

**Research Papers**

1. Breiman, L. (2001). "Random Forests." *Machine Learning*, 45(1), 5-32.
2. Cortes, C., & Vapnik, V. (1995). "Support-vector networks." *Machine Learning*, 20(3), 273-297.
3. Rennie, J. D., Shih, L., Teevan, J., & Karger, D. R. (2003). "Tackling the Poor Assumptions of Naive Bayes Text Classifiers." *Proceedings of the 20th International Conference on Machine Learning (ICML)*.

* **Datasets**

1. Kaggle. [https://www.kaggle.com](https://www.kaggle.com/)