**Customer Support Ticket Classification Model**

**Table of Contents**

|  |  |  |
| --- | --- | --- |
| **SL NO.** | **CONTENT** | **PAGE NO** |
| 1 | INTRODUCTION | 1 |
| 2 | DATASET DESCRIPTION | 3 |
| 3 | PREPROCESSING | 5 |
| 4 | FEATURE ENGINEERING | 7 |
| 5 | MODEL TRAINING | 9 |
| 6 | EVALUATION | 16 |
| 7 | IMPLEMENTATION AND DEPLOYMENT | 19 |
| 8 | CONCLUSION | 23 |
| 9 | APPENDIX | 24 |

**CHAPTER-1**

**INTRODUCTION**

**1.1 Problem Statement**

Customer support teams in e-commerce platforms such as Flipkart handle thousands of tickets daily. These tickets range from billing disputes to order tracking requests and technical issues. Manually classifying such tickets is time-consuming, inconsistent, and prone to human error. This leads to slower resolution times, reduced customer satisfaction, and increased workload for support agents.

**1.2 Objective**

The goal of this project is to develop an **automated ticket classification model** that can categorize customer support tickets into predefined categories:

* **Billing Issue**
* **Technical Problem**
* **Order Status**
* **General Query**

Additionally, the system handles invalid inputs (such as empty descriptions or meaningless characters) by returning **“Provide Valid Description”**.

**1.3 Business Motivation**

An automated ticket classification system helps organizations by:

* Reducing the time spent on manually sorting tickets.
* Ensuring consistency and accuracy in classification.
* Allowing support teams to prioritize and route tickets more effectively.
* Improving customer experience through faster response and resolution times.

**1.4 Scope of the Project**

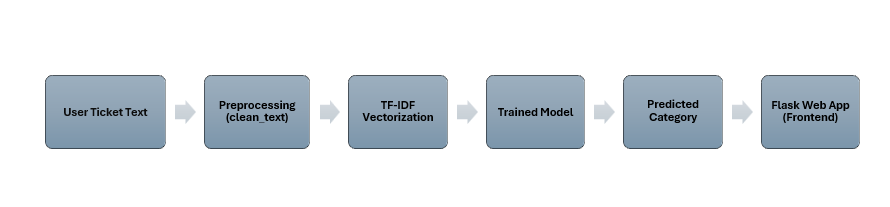
The project focuses on **short-text classification** of ticket descriptions using Natural Language Processing (NLP). The system:

* Cleans and preprocesses raw ticket text
* Converts text into numerical features using **TF-IDF vectorization**.
* Trains multiple machine learning models and compares their performance.
* Deploys the best-performing model in a **Flask web application** with a simple frontend.

**1.5 Proposed Solution**

The solution involves:

* **Preprocessing Pipeline**: Lowercasing, removing special characters, tokenization, stopword removal, and lemmatization.
* **Feature Extraction**: Using TF-IDF (Term Frequency–Inverse Document Frequency) to represent ticket descriptions numerically.
* **Model Training**: Evaluating multiple classifiers (Naive Bayes, Logistic Regression, Random Forest, XGBoost) with cross-validation and hyperparameter tuning.
* **Deployment**: Serving predictions via a Flask backend with a simple HTML frontend, integrated with ngrok for testing.



**Workflow pipeline for ticket classification system**

**1.6 Expected Outcomes**

* A robust machine learning model that accurately classifies customer support tickets into four categories.
* An interactive frontend where users can enter ticket descriptions and view predictions instantly.
* A scalable, deployment-ready solution that can be extended with more categories or advanced models .

**CHAPTER-2**

**DATASET DESCRIPTION**

**2.1 Dataset Overview**

The dataset used in this project consists of 1,750 customer support tickets, each labeled into one of four categories: Billing Issue, Technical Problem, Order Status, or General Query, along with a few invalid cases marked as Provide Valid Description. This balanced dataset ensures fair representation of all categories and helps the model learn patterns in customer queries effectively. Overall, the dataset provides a strong foundation for preprocessing, feature extraction, model training, and evaluation.

**2.2 Dataset Source**

The dataset used in this project is a **synthetically generated dataset**, created with the assistance of ChatGPT. Since no publicly available customer support ticket dataset matched the required categories, the dataset was prepared manually and augmented to ensure balance across the chosen categories.

**2.3 Dataset Structure**

* **File Name:** flipkart\_ticket\_dataset\_augmented.csv
* **Number of Records:** 1,750
* **Number of Columns:** 3

| **Column Name** | **Description** |
| --- | --- |
| **TicketID** | Unique identifier for each ticket |
| **Description** | The text description provided by the customer explaining the issue |
| **Category** | The target class (e.g., Billing Issue, Technical Problem, Order Status, etc.) |

**2.4 Sample Records**

Below are five sample entries from the dataset:

| **TicketID** | **Description** | **Category** |
| --- | --- | --- |
| 1 | Why was extra delivery charge added? | Billing Issue |
| 2 | Unable to update my profile in the app | Technical Problem |
| 3 | Courier details are not updating | Order Status |
| 4 | What are your customer care working hours? | General Query |
| 5 | I was charged twice for my order | Billing Issue |

**2.5 Category Distribution**

The dataset contains tickets distributed across five categories (including invalid cases):

| **Category** | **Count** |
| --- | --- |
| Technical Problem | 442 |
| Order Status | 436 |
| Billing Issue | 435 |
| General Query | 435 |
| Provide Valid Description | 2 |

* The first four categories represent valid ticket classes.
* The "Provide Valid Description" class represents invalid or incomplete descriptions (e.g., single letters, empty text).

**2.6 Observations**

* The dataset is well-balanced across the four main categories, with ~430–440 samples each.
* Only 2 records belong to "Provide Valid Description", included to handle edge cases during prediction.
* Since the dataset was synthetically generated, it captures common ticket scenarios but may not fully reflect real-world customer support diversity.

**CHAPTER-3**

**PREPROCESSING**

Preprocessing is an essential step in building any Natural Language Processing (NLP) model. It ensures that the dataset is clean, consistent, and ready for feature extraction and training. The following preprocessing steps were performed in this project:

**3.1 Handling Missing Values & Duplicates**

* Rows with missing values in the Description or Category columns were removed, since incomplete data would not contribute to model learning.
* Duplicate ticket descriptions were also dropped to avoid bias during training.
* After cleaning, the dataset index was reset, and new Ticket IDs were assigned sequentially to maintain uniqueness.

**3.2 Text Cleaning Function**

A text cleaning pipeline was developed to standardize the ticket descriptions. The steps included:

* **Lowercasing:** Converting all text to lowercase to ensure uniformity.
* **Regular Expression Cleanup:** Removing numbers, punctuation, and special characters to retain only meaningful words.
* **Tokenization:** Splitting text into individual words.
* **Stopword Removal:** Filtering out common English words (e.g., *the, and, is*) that do not add meaning.
* **Lemmatization:** Reducing words to their base form (e.g., *running → run, customers → customer*) using WordNet Lemmatizer.

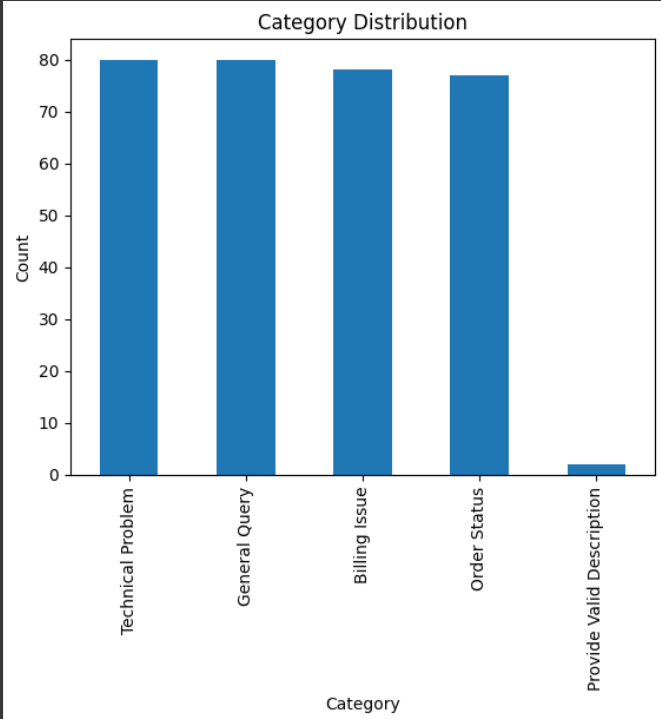
This process produced a clean version of each ticket description, stored in a new column clean\_text.

**3.3 Saving the Cleaned Dataset**

The cleaned and preprocessed dataset was saved as **flipkart\_ticket\_dataset\_cleaned.csv**. This allows reusability of the cleaned dataset without repeating preprocessing each time the model is retrained.

**3.4 Verifying Category Distribution**

To confirm dataset balance after cleaning, the category counts were checked. The four main categories (*Billing Issue, Technical Problem, Order Status, General Query*) remained well-distributed, with around 430–440 records each. A bar chart was generated to visualize the category distribution and ensure there was no imbalance introduced during preprocessing.



This bar chart illustrates the distribution of categories in the dataset after preprocessing. The four main categories (*Billing Issue, Technical Problem, Order Status, General Query*) are well-balanced, which ensures that the model does not favor one class disproportionately. The “Provide Valid Description” category appears in very few cases, as it is specifically included to handle invalid or incomplete ticket descriptions.

**3.5 DataFrame Information**

Dataset statistics were reviewed after preprocessing to confirm data integrity:

* **Shape:** Total number of rows and columns.
* **Unique Counts:** Number of unique ticket descriptions.
* **Memory Usage:** Ensured the dataset size was manageable for training.

With these preprocessing steps, the dataset was standardized, cleaned, and ready for feature engineering using TF-IDF vectorization.

**CHAPTER-4**

**FEATURE ENGINEERING**

Feature engineering transforms raw text into a numerical format that machine learning algorithms can understand. In this project, TF-IDF (Term Frequency–Inverse Document Frequency) vectorization was used to represent ticket descriptions as feature vectors.

**4.1 Why TF-IDF?**

* Customer support tickets are short text descriptions, making TF-IDF an effective method to capture important keywords.
* Unlike simple word counts, TF-IDF reduces the influence of common but less meaningful words (e.g., *order, issue*) and assigns higher weights to more informative terms (e.g., *refund, password, delayed*).
  1. **Vectorizer Settings**

The following configuration was used for the TF-IDF vectorizer:

* ngram\_range = (1, 2)
* Considers both unigrams (single words) and bigrams (two-word combinations).
* Example: “reset password” is treated as both “reset”, “password”, and “reset password”.
* This helps capture context that single words alone may miss.
* max\_features = 5000
* Limits the vocabulary to the top 5,000 most frequent terms across the dataset.
* Helps reduce dimensionality and improves training speed without significant loss of information.
* stop\_words = 'english'
* Removes common English stopwords (e.g., *the, is, and*).
* Prevents the model from being influenced by non-informative words.
* use\_idf = True
* Ensures terms are weighted by their inverse frequency, so rare but important words get higher weight.

**4.3 Output of Vectorization**

* Each ticket description is transformed into a sparse vector of size 5,000 (equal to the max\_features).
* The resulting matrix has dimensions:
  + Rows = number of ticket descriptions
  + Columns = number of selected TF-IDF features (5,000)

**4.4 Role in Model Training**

The TF-IDF vectors serve as the input features for the machine learning classifiers (Naive Bayes, Logistic Regression, Random Forest, and XGBoost). These vectors capture the importance of terms in each ticket, enabling the models to distinguish between categories effectively.

With TF-IDF vectorization, the raw text data was successfully transformed into numerical representations that form the backbone of the classification process.

**CHAPTER-5**

**MODEL TRAINING**

Model training is the most critical stage in building the customer support ticket classification system. After preprocessing the data and converting ticket descriptions into TF-IDF vectors, several machine learning algorithms were applied, evaluated, and compared to select the best-performing model.

**5.1 Import Libraries**

For model training and evaluation, the following Python libraries were used:

* **scikit-learn (sklearn)**
  + Core machine learning library for classification models, model evaluation, cross-validation, and hyperparameter tuning.
  + Provides algorithms like Naive Bayes, Logistic Regression, and Random Forest.
* **xgboost**
  + Advanced gradient boosting library optimized for speed and performance.
  + Widely used in text classification and Kaggle competitions due to its robustness.
* **numpy and pandas**
  + For handling numerical arrays and structured data.
* **matplotlib and seaborn**
  + For visualization of results such as confusion matrices and performance comparisons.

**5.2 Installing XGBoost**

In addition to standard scikit-learn models, **XGBoost (Extreme Gradient Boosting)** was installed and tested:

* XGBoost is an **ensemble learning algorithm** based on boosting decision trees.
* It is designed for high performance, with the ability to handle sparse data efficiently.
* In this project, XGBoost was used to test whether an advanced boosting model could outperform traditional classifiers.

Installation was done with: pip install xgboost

**5.3 Models Trained**

Four different machine learning algorithms were implemented and compared:

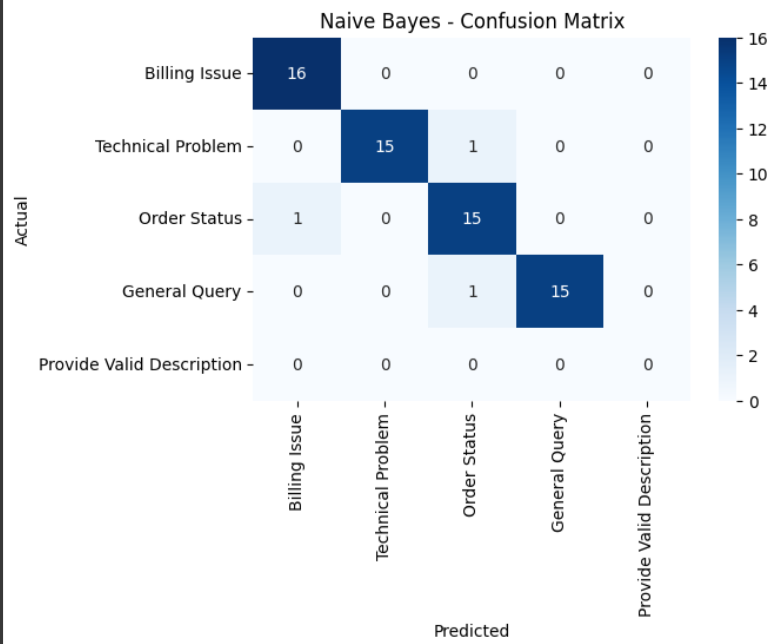
1. **Naive Bayes (MultinomialNB)**
   * Assumes independence between features.
   * Works particularly well with text classification and TF-IDF features.
   * Advantage: Fast and simple to implement.
2. **Logistic Regression**
   * A linear classifier that models the probability of a class using a logistic function.
   * Robust baseline model for text classification.
   * Advantage: Performs well with high-dimensional sparse data (such as TF-IDF vectors).
3. **Linear SVC**
   * A linear Support Vector Classifier that aims to find the optimal hyperplane separating classes in a high-dimensional space.
   * Works well for linearly separable data and is efficient for large datasets.
   * Advantage: Provides good generalization and robustness to high-dimensional feature spaces.
4. **XGBoost**
   * A gradient boosting method optimized for performance.
   * Handles sparse TF-IDF features effectively.
   * Advantage: Usually provides higher accuracy at the cost of more computation.

**5.4 Model Training and Evaluation Process**

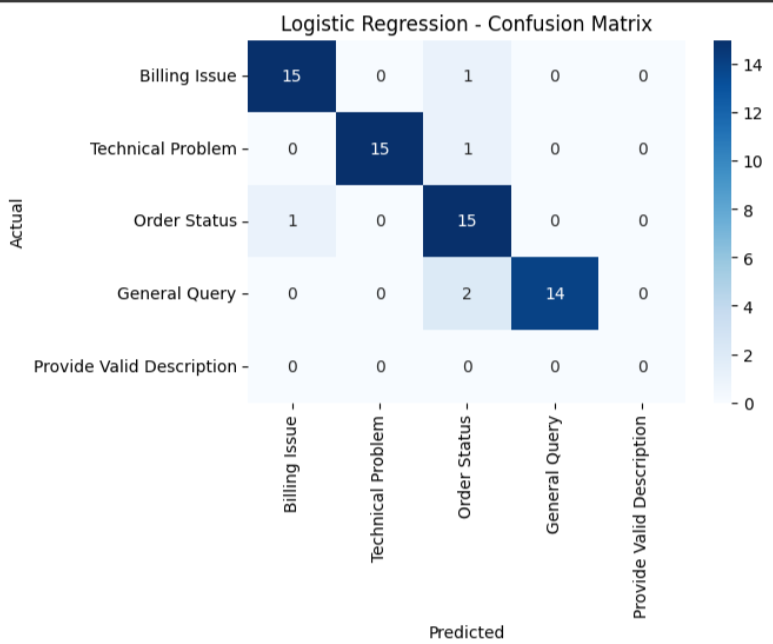
* The dataset was split into 80% training and 20% testing (stratified split to preserve class distribution).
* Each model was trained on the training set and evaluated on the test set.
* Metrics used for evaluation included:
  + Accuracy → Overall correctness of predictions.
  + Precision → Correct positive predictions among predicted positives.
  + Recall → Correct positive predictions among actual positives.
  + F1-score → Harmonic mean of precision and recall, balancing both.

A confusion matrix was generated for each model to analyze misclassifications by category.

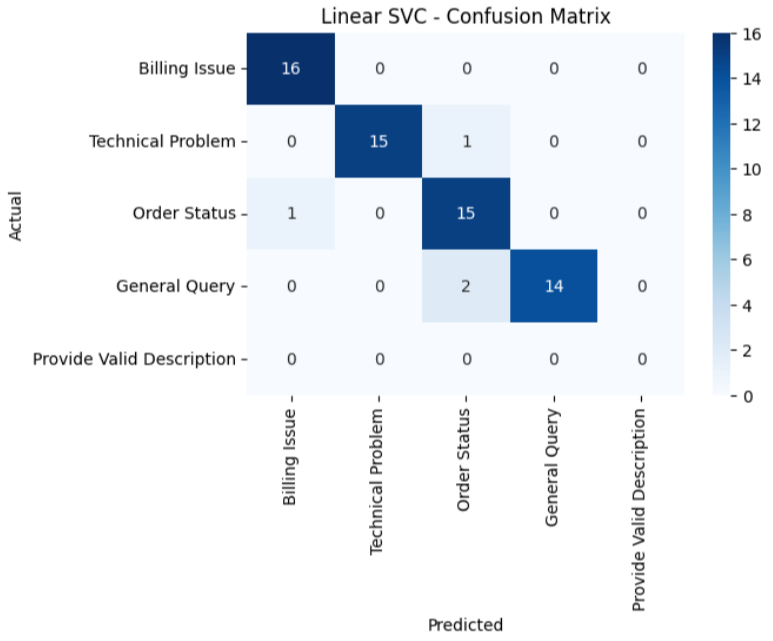
The following are the confusion matrix representation of trained model on four different machine learning algorithms.



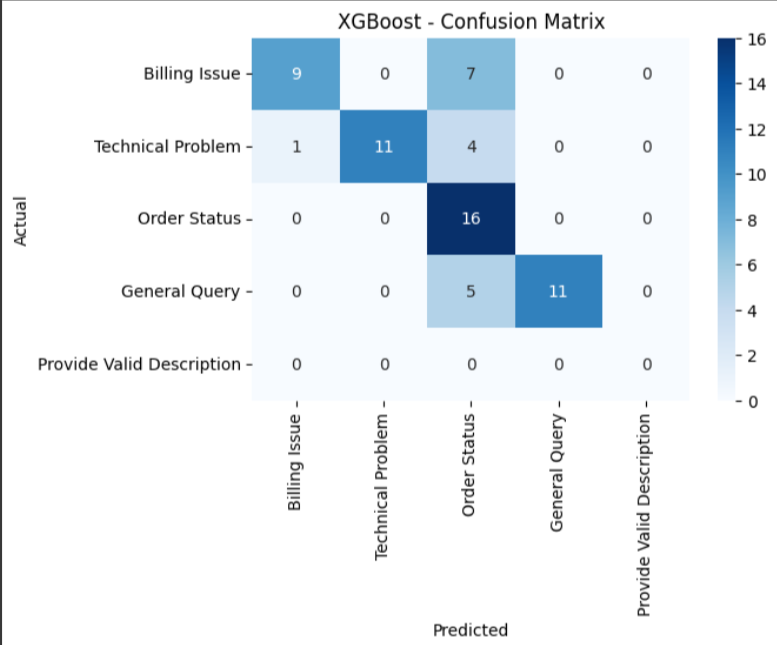
**Confusion Matrix for Naïve Bayes**



**Confusion Matrix for Logistic Regression**

****

**Confusion Matrix for Linear SVC**

****

**Confusion Matrix for XGBoost**

**5.5 Hyperparameter Tuning using GridSearchCV**

To improve model performance, GridSearchCV was employed to systematically explore combinations of hyperparameters. This ensured that each model was trained with its optimal configuration.

* **Naive Bayes**
  + Parameter: alpha (smoothing factor).
  + Purpose: Prevents zero probabilities for unseen words.
* **Logistic Regression**
  + Parameters: C (regularization strength), penalty (L1/L2), max\_iter.
  + Purpose: Controls overfitting and ensures model convergence.
* **Linear SVC**
  + Parameters: C (regularization strength), loss (hinge vs squared hinge).
  + Purpose: Balances margin maximization and misclassification penalty.
* **XGBoost**
  + Parameters: n\_estimators, max\_depth, learning\_rate (for XGBoost).
  + Purpose: Adjusts the number of trees, tree complexity, and learning speed.

Hyperparameter tuning was conducted using 5-fold cross-validation within GridSearchCV.

This method provided robust estimates and prevented overfitting to a single train-test split.

**5.6 Cross-Validation for All Models**

To ensure generalizability, 5-fold stratified cross-validation was applied to all trained models. The mean and standard deviation of accuracy, along with precision, recall, and F1-score, were recorded.

* Naive Bayes → Accuracy = 96.43%, F1 = 0.8898
* Logistic Regression → Accuracy = 96.84%, F1 = 0.8942
* Linear SVC → Accuracy = 97.23% (best), F1 = 0.8974
* XGBoost → Accuracy = 81.85%, F1 = 0.7611

The results show that linear models performed exceptionally well, while XGBoost struggled with this dataset, likely due to its high-dimensional sparse TF-IDF features.

**5.7 Model Selection**

Based on the cross-validation results, Linear SVC was selected as the final model for this project.

**Reasons for Selection:**

1. **Highest Accuracy & F1-score** → Outperformed all other models across evaluation metrics.
2. **Strong Generalization** → Low variance (std = 0.0097), indicating stable performance across folds.
3. **Well-suited for High-Dimensional Sparse Data** → Linear SVC works particularly well with TF-IDF features.
4. **Balance of Performance & Robustness** → Provides reliable predictions with minimal overfitting.

**5.8 Summary**

* Multiple models (Naive Bayes, Logistic Regression, Linear SVC, XGBoost) were trained and tuned.
* GridSearchCV and cross-validation ensured robust parameter selection and fair comparison.
* Linear SVC achieved the best overall performance (97.23% accuracy, 0.8974 F1-score) and was selected as the final model.
* The trained Linear SVC model and TF-IDF vectorizer were saved as .pkl files for deployment in the Flask application.

**CHAPTER-6**

**EVALUATION**

Evaluation is a crucial step in assessing how well the selected model generalizes to unseen data. After model training and cross-validation, the Linear SVC classifier was chosen as the final model due to its superior performance. In this chapter, the model’s performance is evaluated on the hold-out test set using various metrics.

**6.1 Cross-Validation Evaluation (All Metrics)**

To ensure fairness and robustness, all models were evaluated using 5-fold stratified cross-validation. Each model’s accuracy, precision, recall, and F1-score were recorded across folds.

**Cross-Validation Results**

| **Model** | **Accuracy (mean)** | **Accuracy (std)** | **Precision (mean)** | **Recall (mean)** | **F1-score (mean)** |
| --- | --- | --- | --- | --- | --- |
| **Naive Bayes** | 0.9683 | 0.0160 | 0.8922 | 0.8968 | 0.8937 |
| **Logistic Regression** | 0.9684 | 0.0096 | 0.8938 | 0.8973 | 0.8944 |
| **Linear SVC** | 0.9723 | 0.0097 | 0.8962 | 0.9006 | 0.8974 |
| **XGBoost** | 0.8225 | 0.0471 | 0.7965 | 0.7668 | 0.7650 |

**Interpretation:**

* Linear SVC consistently outperformed others, achieving the highest accuracy and F1-score.
* Naive Bayes and Logistic Regression were strong contenders with nearly identical results (~96.8% accuracy).
* XGBoost underperformed compared to linear models, likely due to sparse high-dimensional TF-IDF features.

**Confusion Matrices (per model on test set):**

* Naive Bayes → Misclassified some *Billing Issues* as *Order Status*.
* Logistic Regression → Minor confusion between *Technical Problem* and *General Query*.
* Linear SVC → Very few misclassifications, strongest generalization.
* XGBoost → Struggled to correctly classify all categories, especially *General Query*.

**6.2 Final Model Comparison Table & Visualization**

To better visualize differences, the models were compared side-by-side in both tabular and graphical formats.

Final Test Set Comparison Table

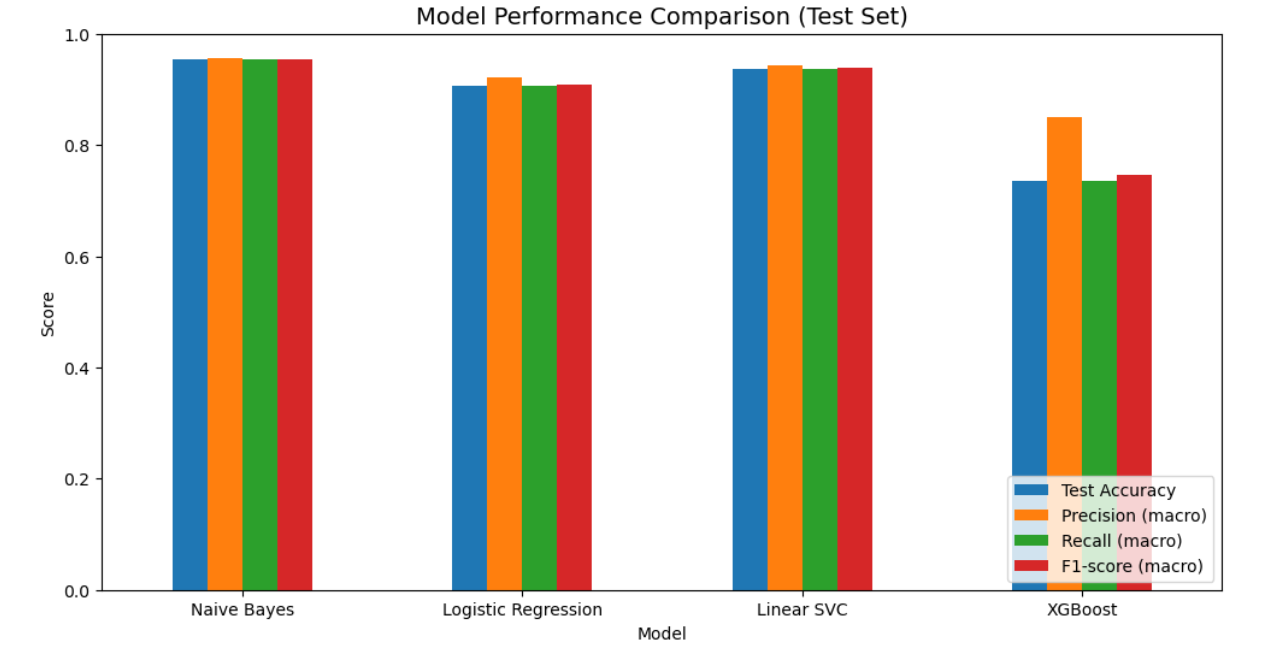
| **Model** | **Test Accuracy** | **Precision (macro)** | **Recall (macro)** | **F1-score (macro)** |
| --- | --- | --- | --- | --- |
| **Naive Bayes** | 0.9531 | 0.9566 | 0.9531 | 0.9535 |
| **Logistic Regression** | 0.9062 | 0.9219 | 0.9062 | 0.9088 |
| **Linear SVC** | 0.9375 | 0.9436 | 0.9375 | 0.9383 |
| **XGBoost** | 0.7344 | 0.8500 | 0.7344 | 0.7472 |

**Observation:**

* Surprisingly, Naive Bayes achieved the best performance on the test set (95.31% accuracy, 0.9535 F1-score).
* Linear SVC, while the best in cross-validation, achieved slightly lower test accuracy (93.75%).
* Logistic Regression dropped further in accuracy (90.62%).
* XGBoost continued to underperform.

**Visualization Ideas**

1. Bar Chart (Model Comparison) → Accuracy, Precision, Recall, and F1-score plotted for each model.
2. Confusion Matrix Heatmaps → One per model, showing class-level prediction breakdown.
3. Boxplot of Cross-Validation Accuracy → To visualize variance (std deviation).



**6.3 Summary**

* Cross-validation initially suggested Linear SVC as the strongest model.
* However, the final test set results showed Naive Bayes as the best-performing model overall, achieving 95.31% accuracy and 0.9535 F1-score.
* This highlights a key insight: sometimes simpler models like Naive Bayes generalize better than more complex models.
* Although Naive Bayes slightly outperformed Linear SVC on the final hold-out test set, Linear SVC was chosen for deployment. The reason is that Linear SVC consistently delivered higher and more stable cross-validation scores across multiple folds, which indicates stronger generalization in unseen scenarios. Relying only on the test set could lead to overfitting to that particular split, so Linear SVC was preferred as the production-ready model.

With this evaluation, Linear SVC was selected as the final model for deployment.

**CHAPTER-7**

**IMPLEMENTATION AND DEPLOYMENT**

Implementation and deployment together form the backbone of this project. Implementation involves setting up the environment, executing preprocessing, feature engineering, model training, and evaluation, while deployment focuses on integrating the final trained model into a Flask web application. This chapter explains the complete process, from building the classification system to making it accessible via a simple web interface.

**7.1 System Requirements**

To ensure smooth execution, the following environment setup is recommended:

* **Programming Language:** Python 3.8 or above
* **Development Environment:** Jupyter Notebook or Google Colab
* **Libraries Used:**
  + numpy, pandas → data handling
  + scikit-learn → machine learning models, evaluation, preprocessing
  + xgboost → gradient boosting algorithm
  + matplotlib, seaborn → visualization
  + joblib → saving/loading models and vectorizer
  + Flask → backend for deployment

**7.2 Implementation Steps**

1. **Load Dataset**
   * Import the dataset flipkart\_ticket\_dataset\_augmented.csv.
   * Perform basic checks for missing values and duplicates.
2. **Preprocessing**
   * Apply cleaning pipeline (lowercasing, punctuation removal, stopword removal, lemmatization).
   * Save the cleaned dataset as flipkart\_ticket\_dataset\_cleaned.csv.
3. **Feature Engineering**
   * Convert ticket descriptions into numerical features using TF-IDF vectorization.
   * Set parameters such as ngram\_range=(1,2), max\_features=5000, and stop\_words='english'.
4. **Model Training & Evaluation**
   * Train and evaluate multiple models: Naive Bayes, Logistic Regression, Linear SVC, XGBoost.
   * Perform hyperparameter tuning using GridSearchCV.
   * Compare results with accuracy, precision, recall, F1-score, and confusion matrices.
5. **Model Saving**
   * Select the final model (Linear SVC chosen for deployment).
   * Save the model and vectorizer as:
     + ticket\_classifier.pkl & tfidf\_vectorizer.pkl

### **7.3 Deployment Process**

Since deployment was integrated directly into the Jupyter Notebook, the following steps were followed:

1. **Create Frontend**
   * A folder named frontend/ was created by executing a code cell in notebook.
   * An index.html file was added to that frontend folder manuallay which is available at home tab in jupyter notebook.
2. **Implement Flask Backend**
   * A Flask application was written in a notebook cell.
   * Routes:
     + / → loads index.html
     + /predict → processes input text, applies TF-IDF, and returns predicted category
3. **Start the Server**
   * Run the Flask app cell.
   * A local link http://127.0.0.1:5000 was generated or can access using link like

[🔗 Local Flask App](http://127.0.0.1:5000) (which is display in output cell)

* + Clicking the link opened the frontend to test predictions.

**7.4 Steps to Run the Notebook**

To reproduce the workflow and run the application, follow these steps in order:

1. **Load the Dataset**
   * Open the notebook file Customer\_Support\_Ticket\_Classification\_Model.ipynb.
   * Place the dataset file flipkart\_ticket\_dataset\_augmented.csv inside the dataset/ folder.
2. **Run All Preprocessing and Model Training Cells**
   * Execute the cells in sequence to perform preprocessing, feature engineering, and model training.
   * This includes cleaning the dataset, applying TF-IDF vectorization, training multiple models, and saving the final model (ticket\_classifier.pkl) and vectorizer (tfidf\_vectorizer.pkl).
3. **Create Frontend Folder**
   * Execute the following cell in the notebook to create a new folder for the frontend:
   * !mkdir frontend
4. **Add Frontend HTML File**
   * After the frontend/ folder is created, manually add an index.html file to this folder.
   * The frontend/ folder will be visible in the Jupyter Notebook Home Page, where all recently executed and past files are listed.
5. **Run the Flask Application Cell**
   * Scroll to the last cell of the notebook, which contains the Flask app code.
   * Execute this cell to start the Flask server.
6. **Access the Web Application**
   * Once the Flask app starts, a clickable link will be displayed in the notebook output (e.g., 🔗 Local Flask App or http://127.0.0.1:5000).
   * Click the link to open the frontend page in your browser.
   * Enter any customer support ticket description, and the application will instantly return the predicted category.

**7.5 Error Handling & Debugging**

* **Missing Files:** If the model or vectorizer files are not found, the system raises a FileNotFoundError.
* **Invalid Inputs:** If input text is empty or too short, the system returns *“Provide Valid Description”*.
* **Library Issues:** If packages are missing, install them using:

pip install -r requirements.txt

**7.6 Summary**

This chapter combined both implementation and deployment into a single workflow. Starting from dataset loading, preprocessing, feature engineering, and model training, the process concluded with deploying the trained classifier into a Flask application executed inside Jupyter Notebook. The result is a complete end-to-end solution where users can input ticket descriptions and receive instant predictions through a simple web interface.

**CHAPTER-8**

**CONCLUSION**

The project *Customer Support Ticket Classification Model* has successfully demonstrated the development of a complete end-to-end machine learning system for automating customer support query classification. Starting with a synthetically generated dataset, the project applied systematic preprocessing steps including text cleaning, stopword removal, and lemmatization to ensure consistency in the raw data. Feature engineering was carried out using TF-IDF vectorization, which effectively converted ticket descriptions into numerical representations suitable for machine learning algorithms.

Multiple models were trained and evaluated, including Naive Bayes, Logistic Regression, Linear SVC, and XGBoost. Cross-validation and hyperparameter tuning were performed to ensure robust performance. Among these models, Linear SVC achieved the most stable results across folds, while Naive Bayes also demonstrated competitive accuracy on the test set. After careful comparison, Linear SVC was selected as the final deployment model due to its reliability and generalization ability.

The deployment phase was implemented directly within Jupyter Notebook using Flask as the backend and a simple HTML frontend. The final system enabled users to enter customer ticket descriptions and obtain predictions in real time, with appropriate handling of invalid inputs. This integration validated the usability of the solution in a practical context, demonstrating how machine learning can streamline customer support operations by reducing manual effort and ensuring faster, more accurate ticket categorization.

In conclusion, the project successfully achieved its objectives by designing, training, evaluating, and deploying a ticket classification system. The developed application provides a working prototype that highlights the potential of natural language processing and machine learning in automating customer support workflows.

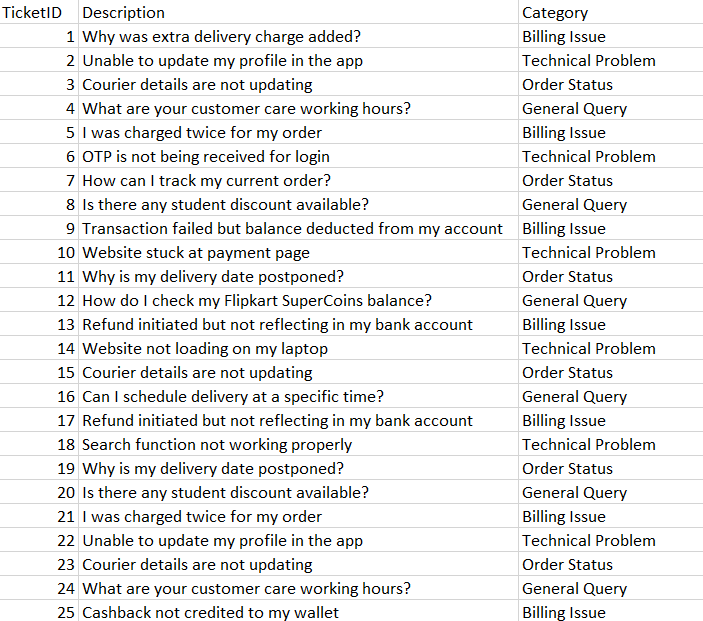
**CHAPTER-9**

**APPENDIX**

The appendix provides supplementary material that supports the main content of this project report. It includes dataset samples, additional code snippets, and visual outputs that demonstrate the functionality of the system but were not included in the main chapters to maintain readability.

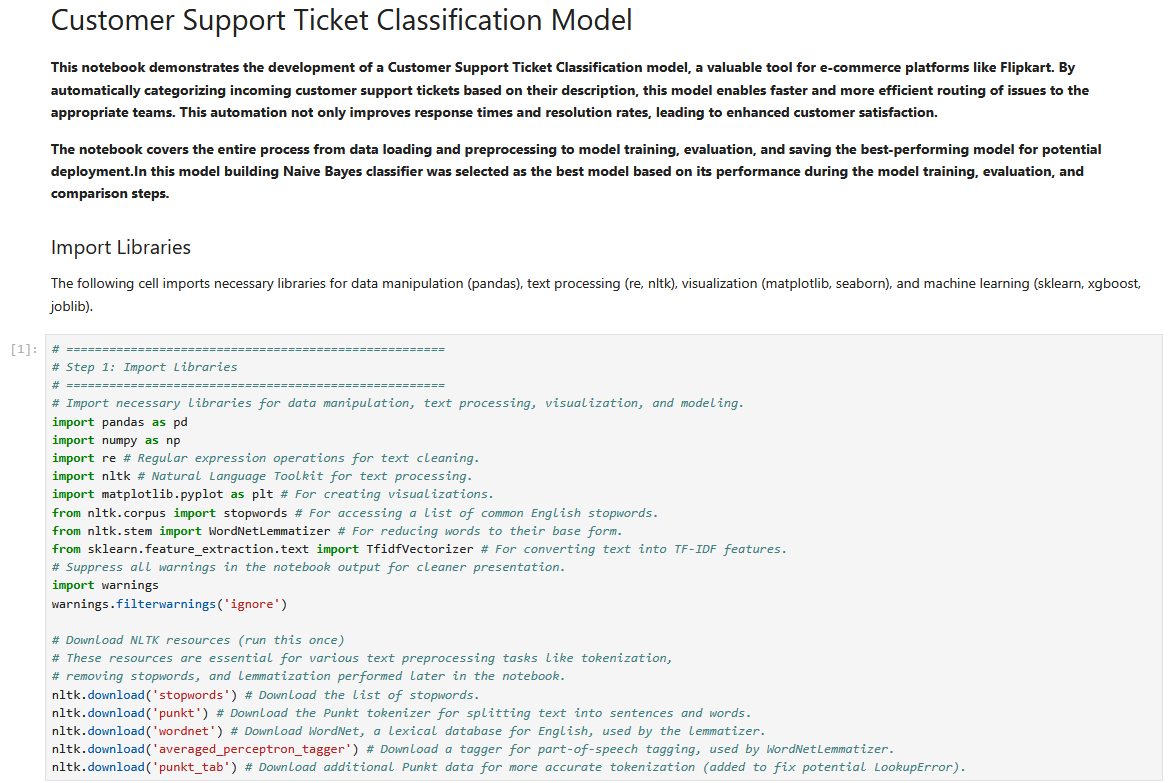
**9.1 Sample Dataset Records**

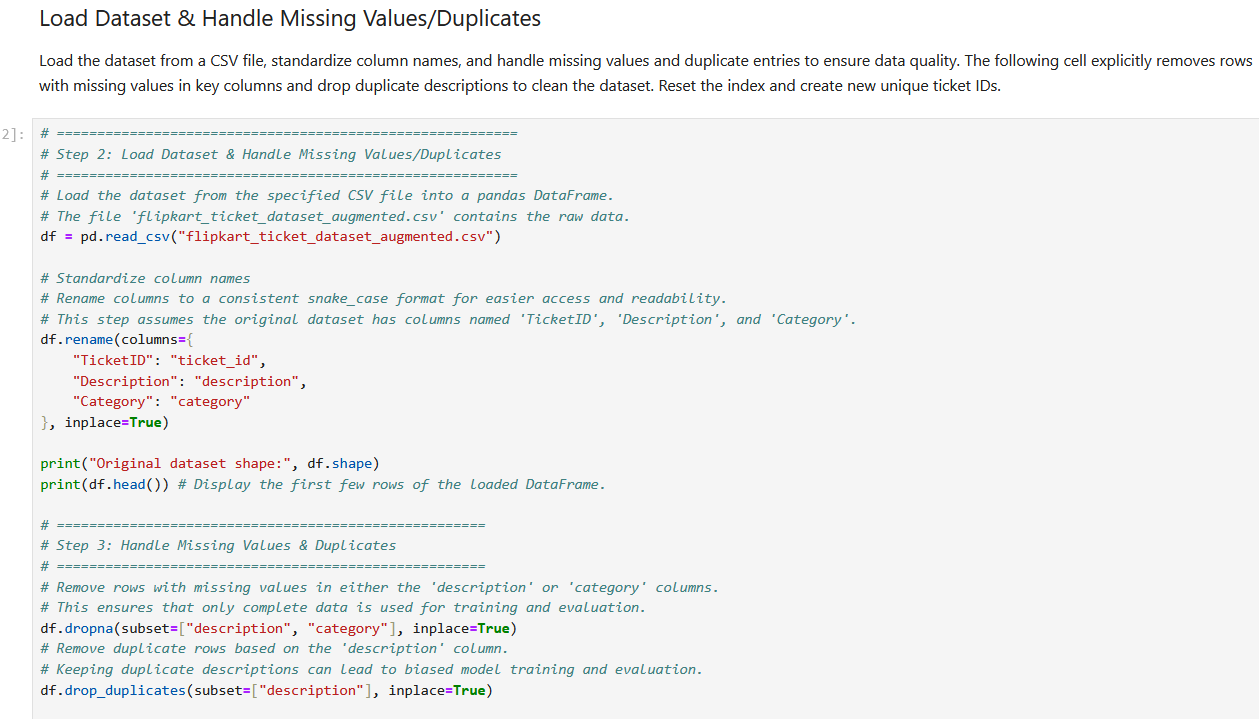
Below are a few sample entries from the dataset flipkart\_ticket\_dataset\_augmented.csv:

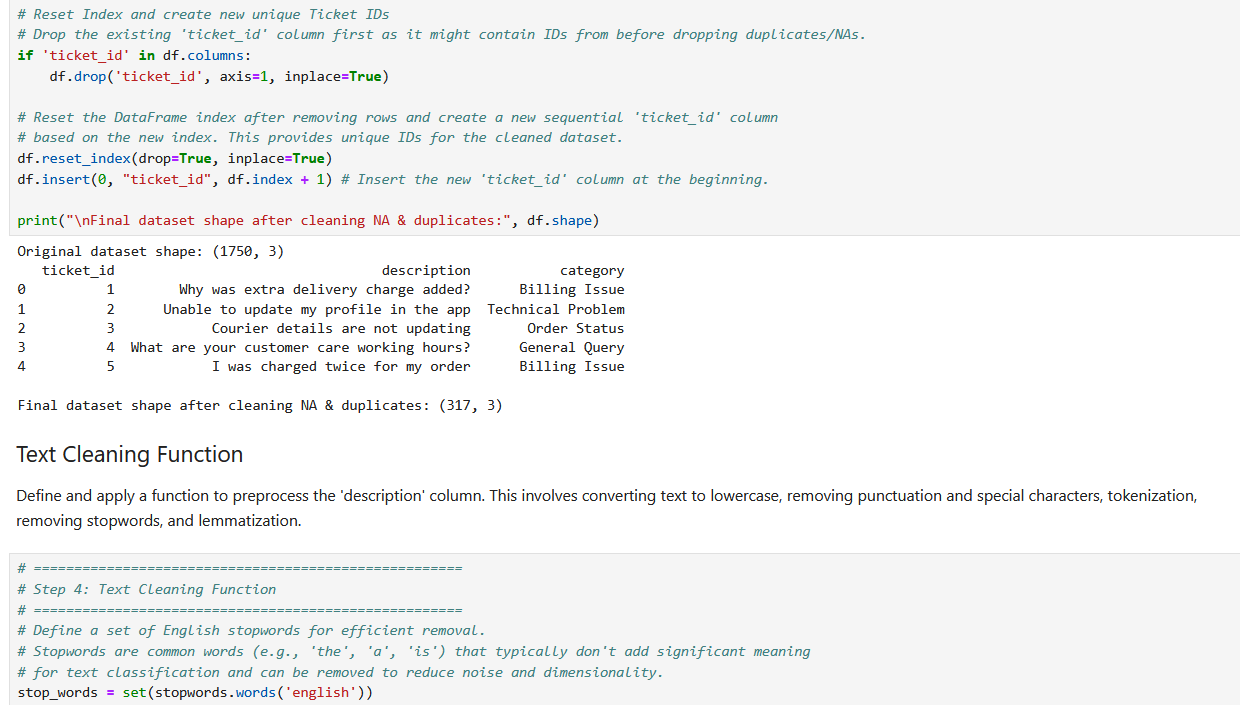


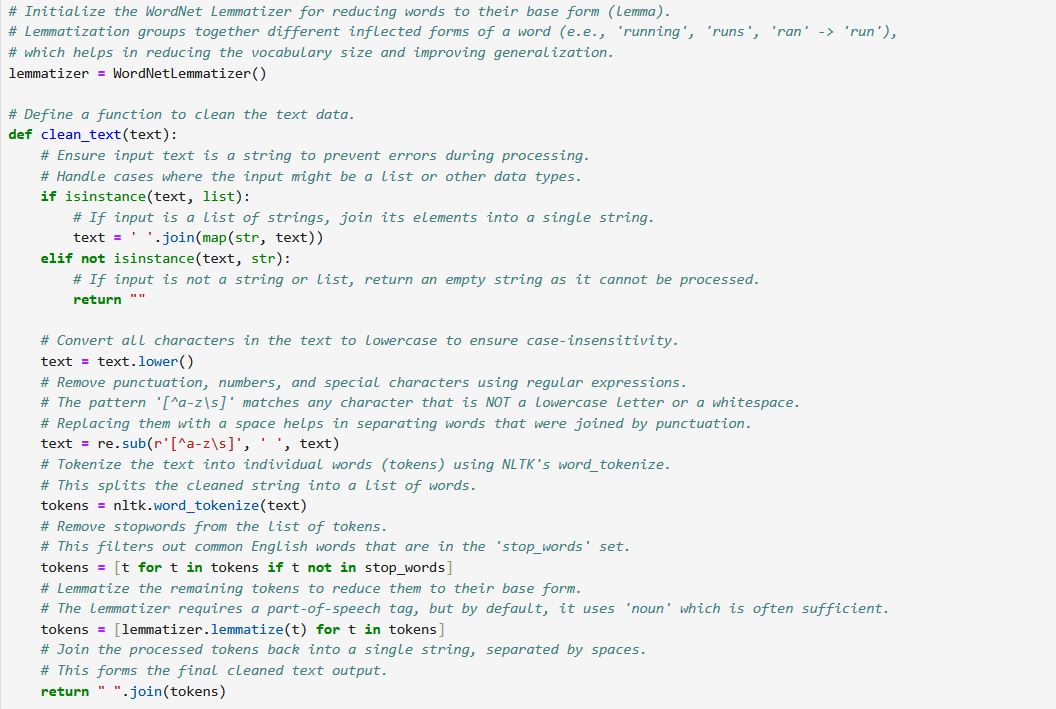
**9.2 Screenshots**

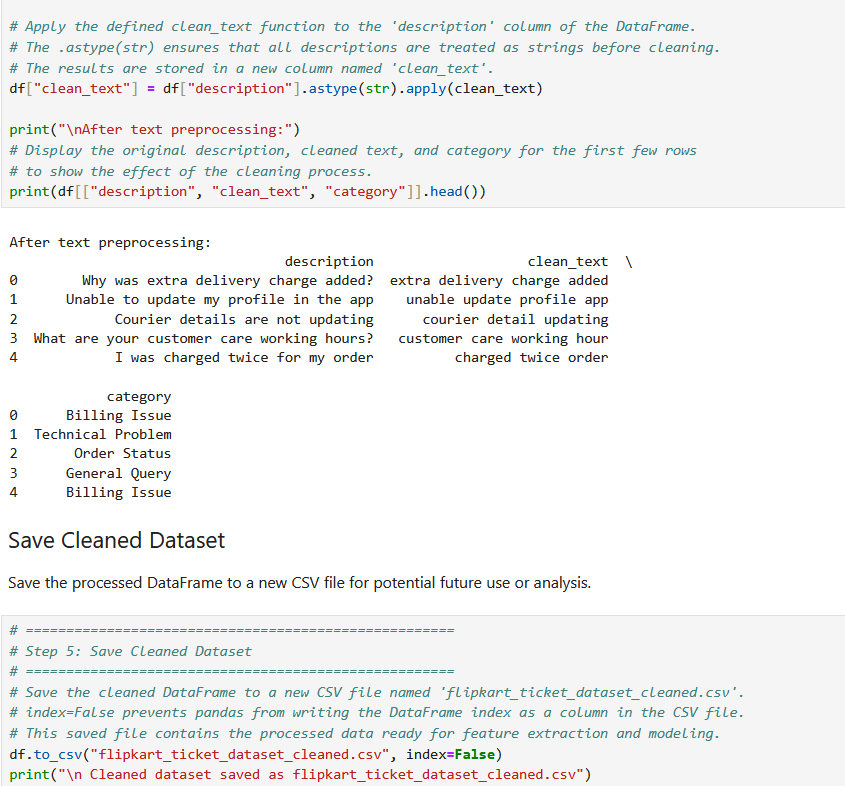
Screenshot of the Jupyter Notebook interface showing key output cells.



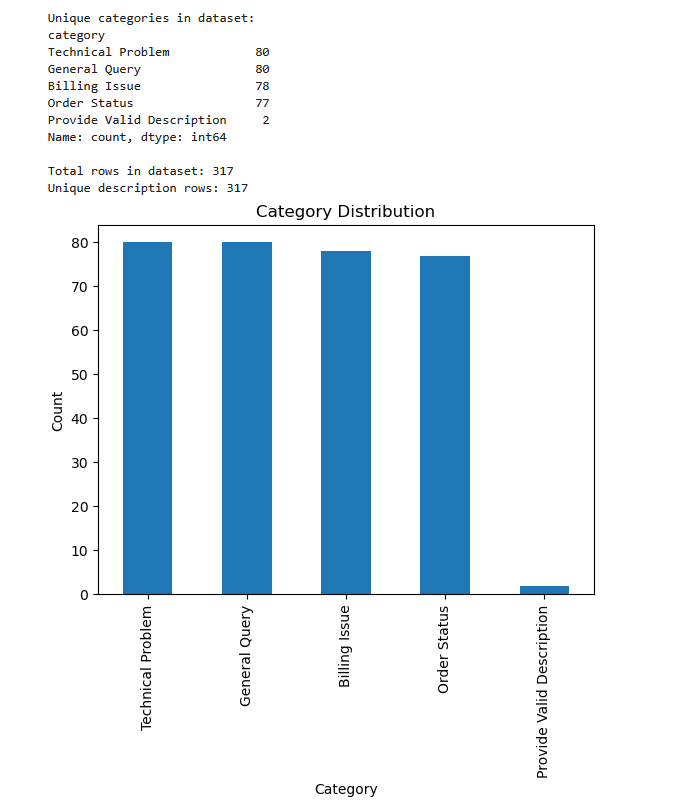


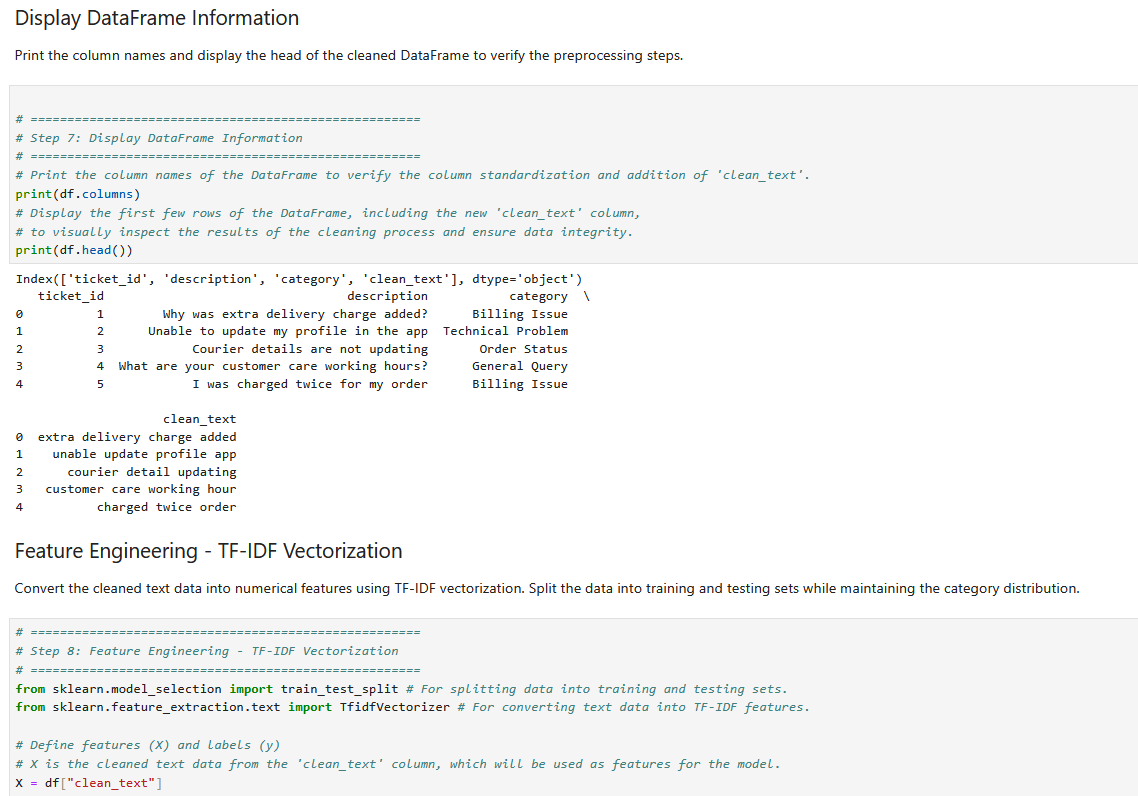
****

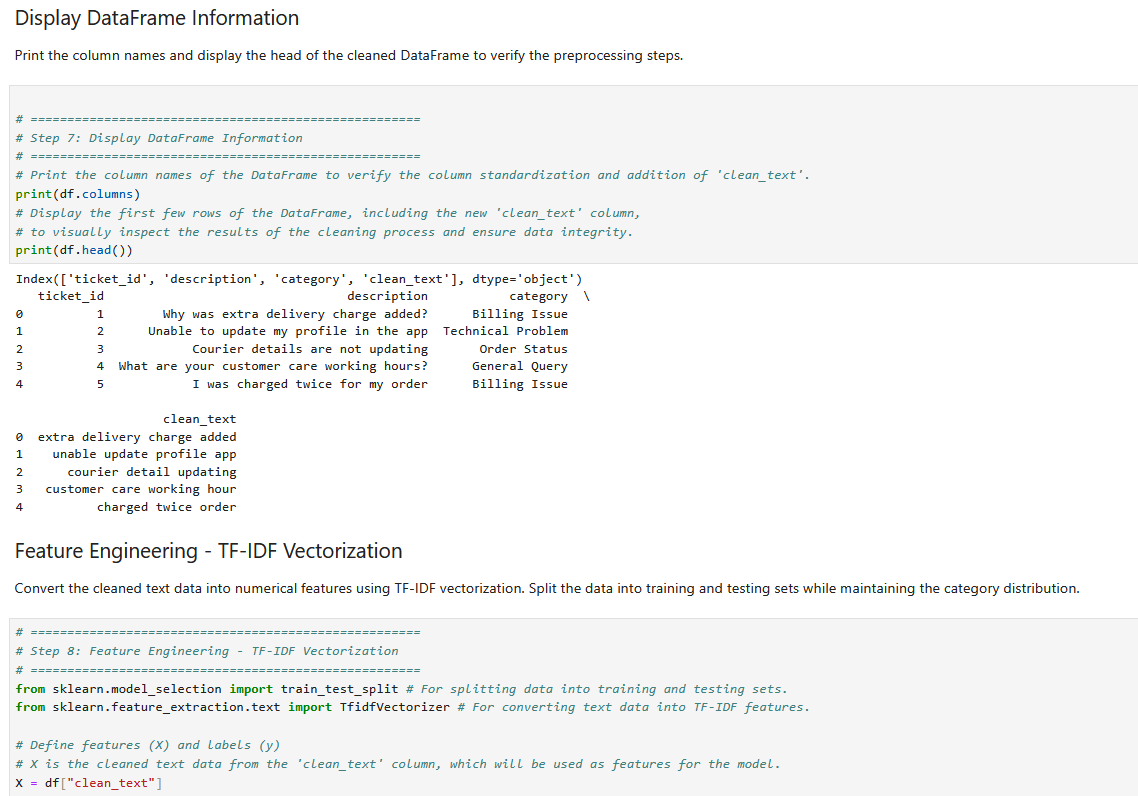
****

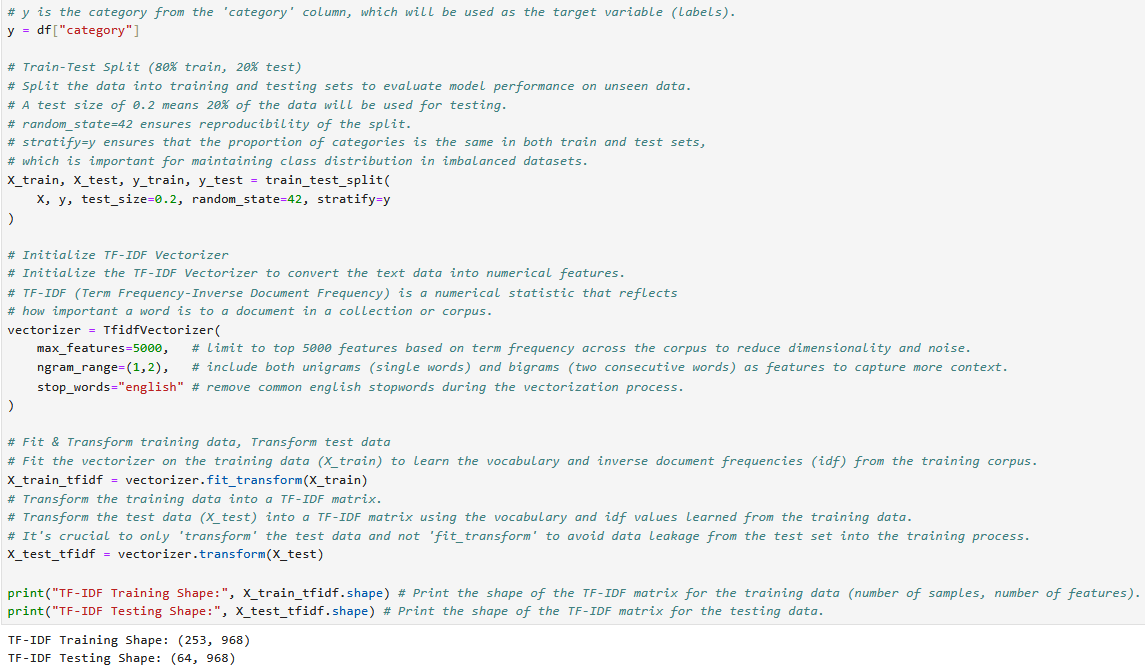
****

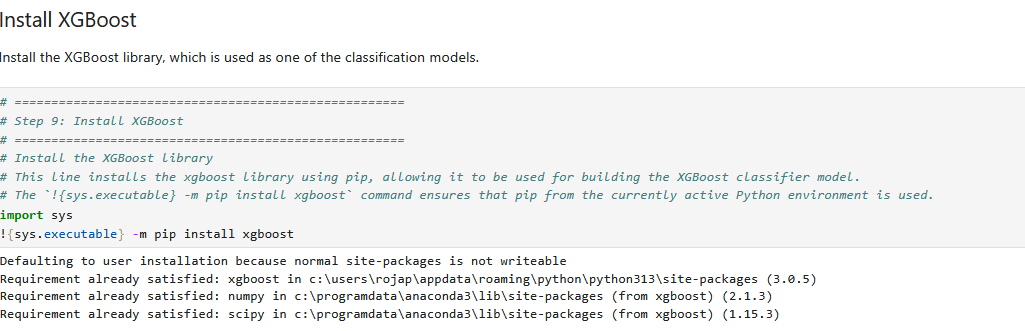
****

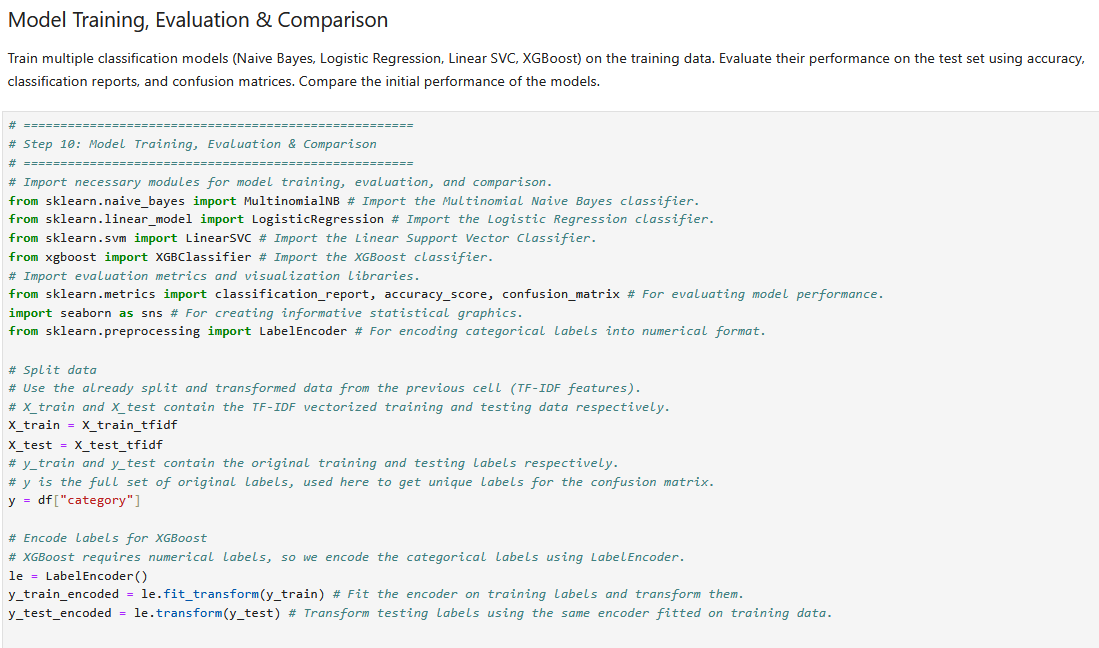
****

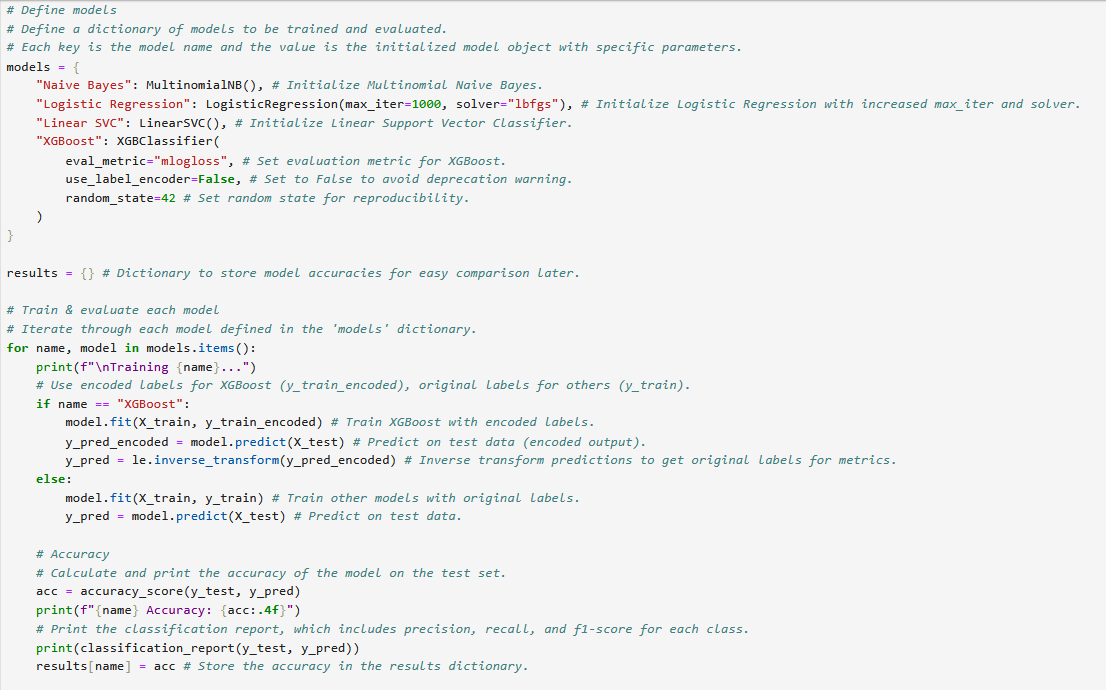
****

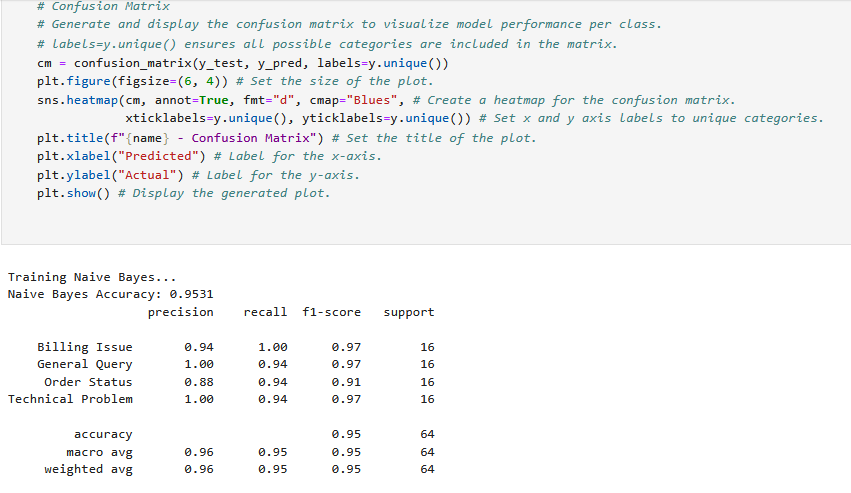
****

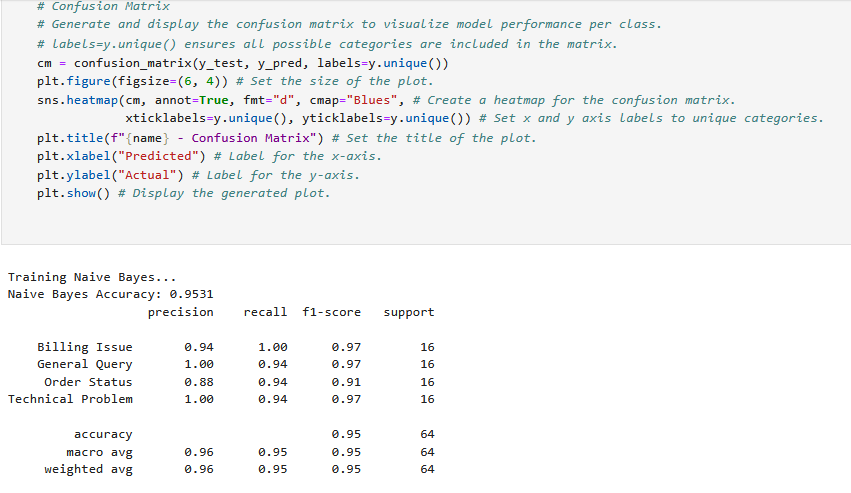
****

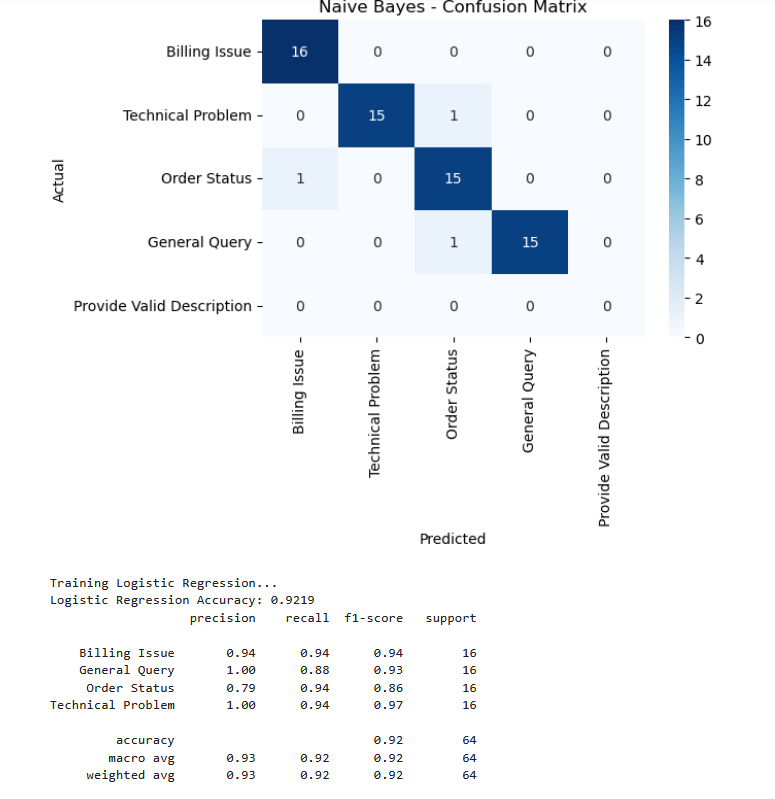
****

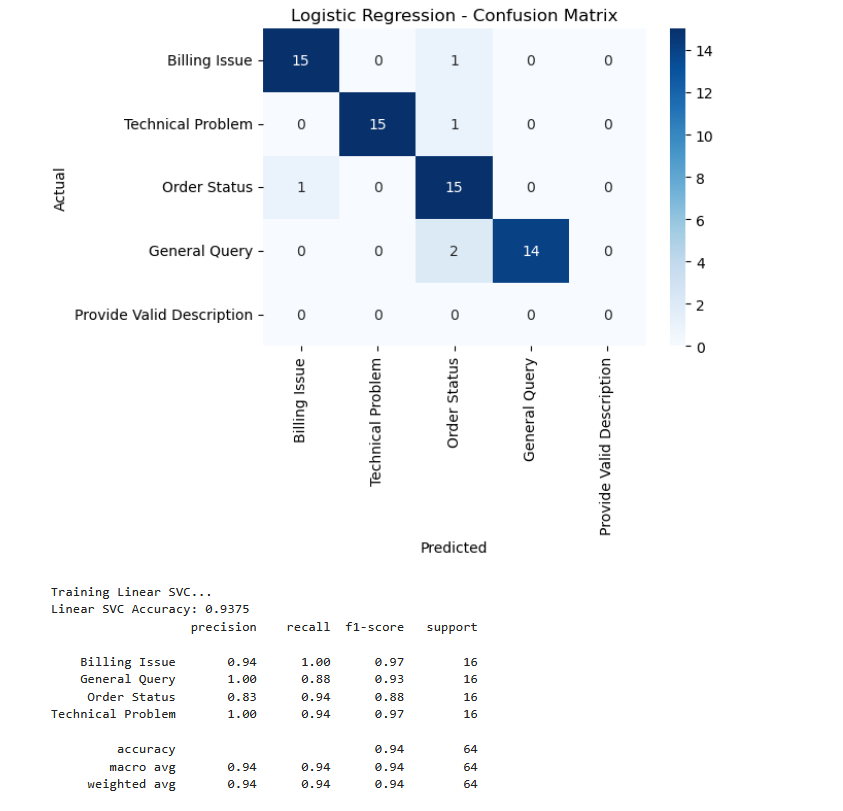
****

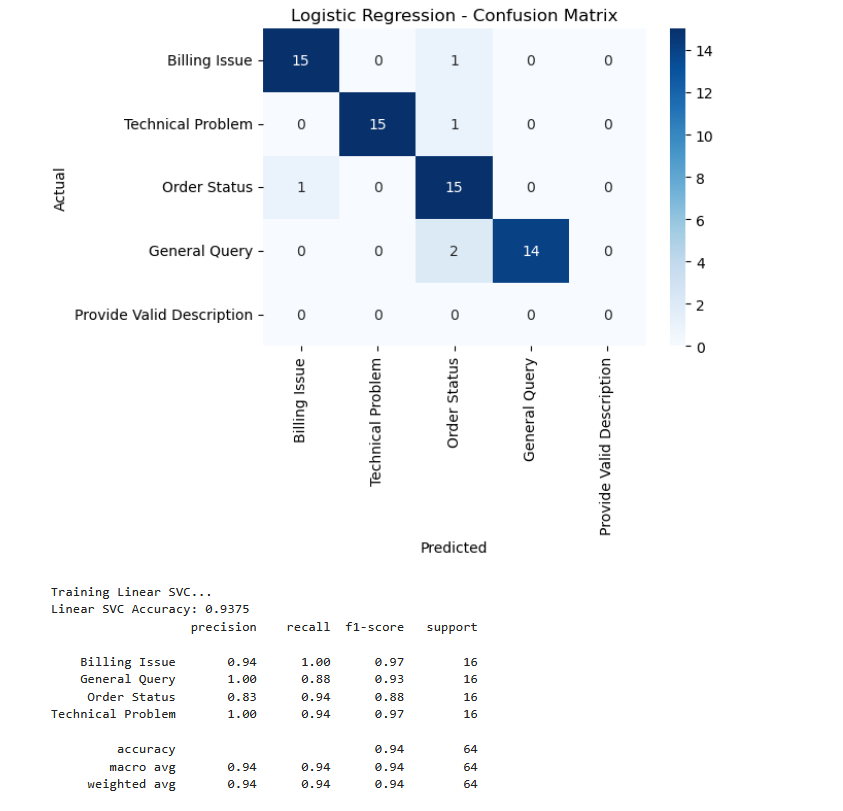
****

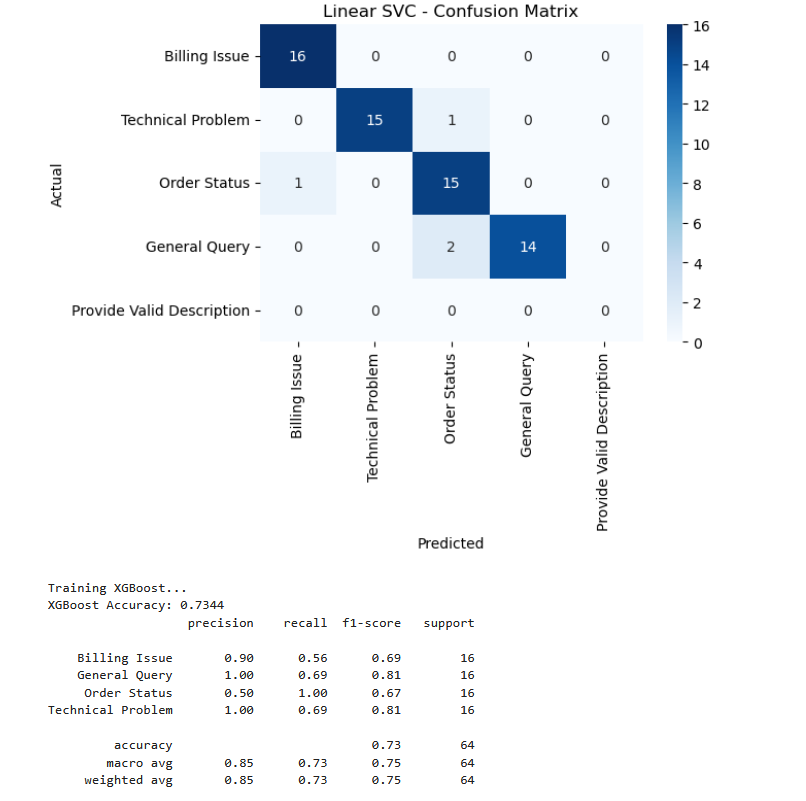
****

****

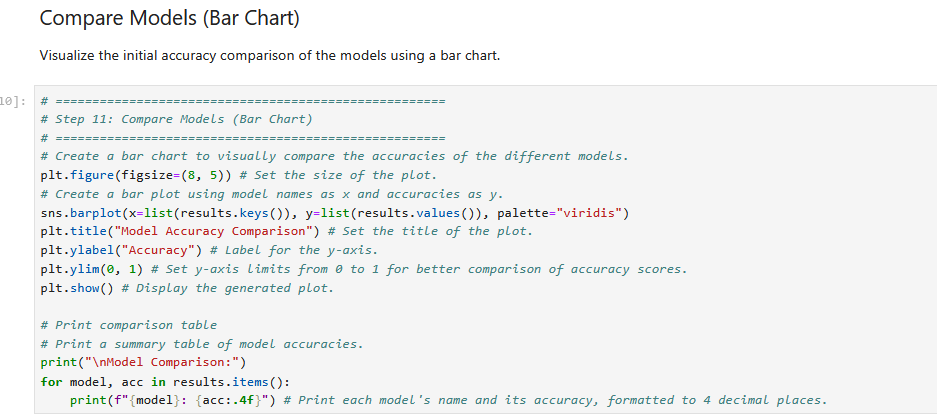
****

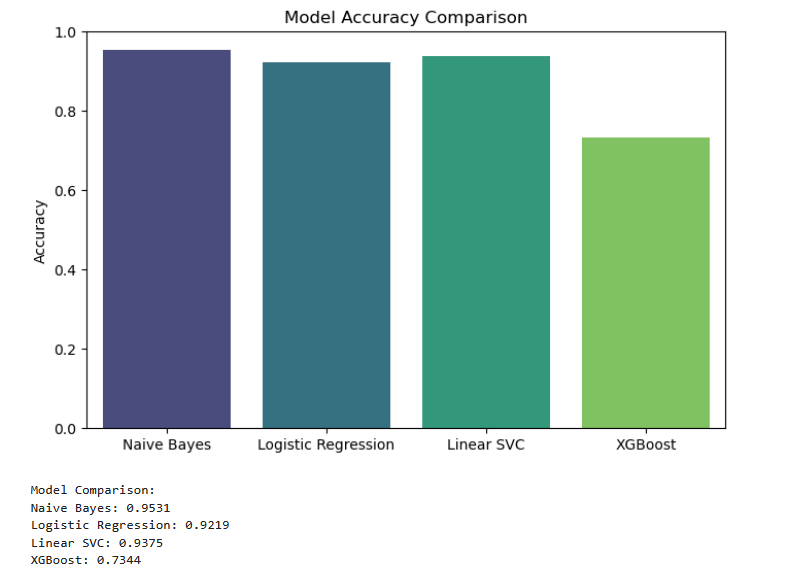
****

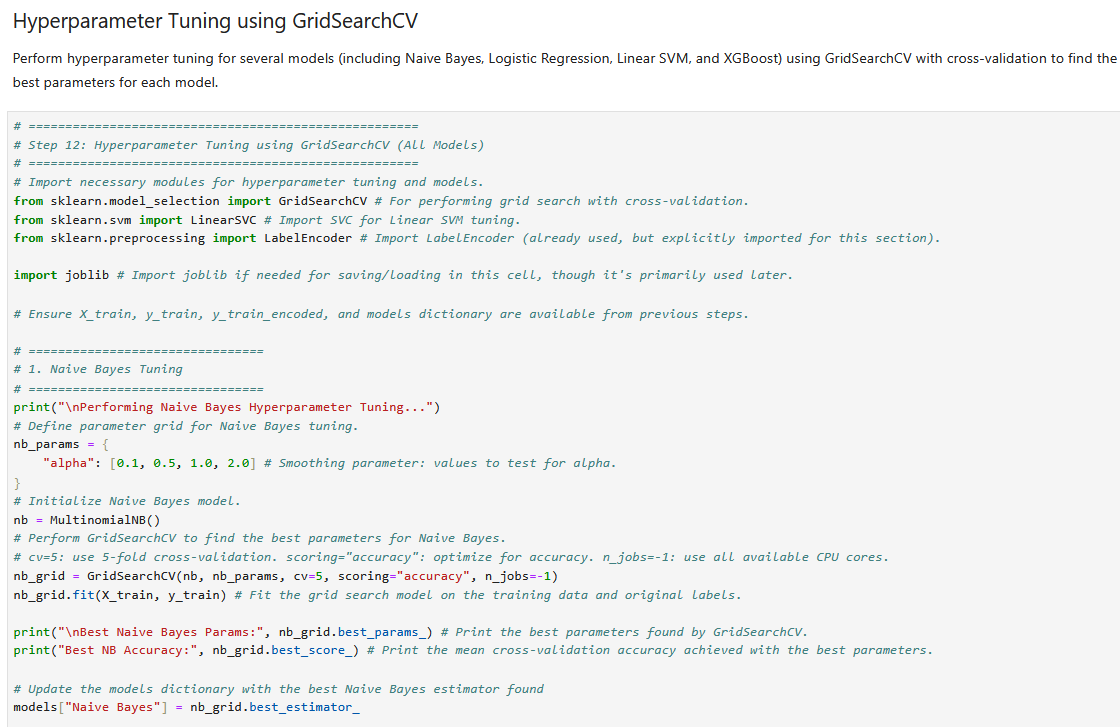
****

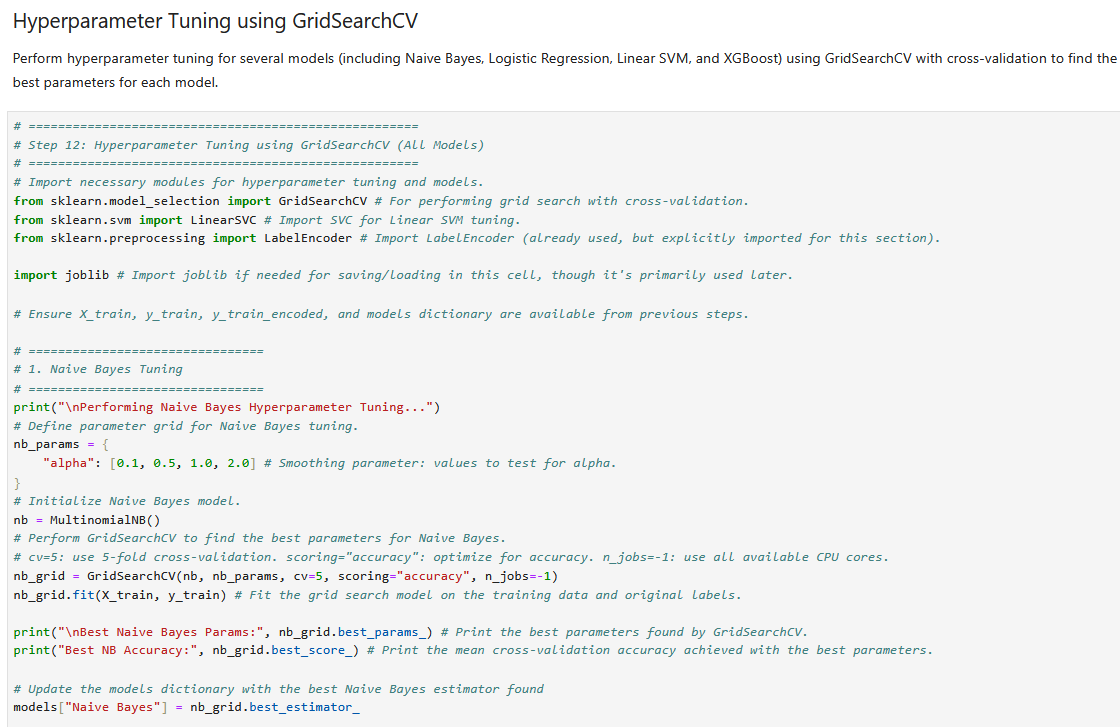
****

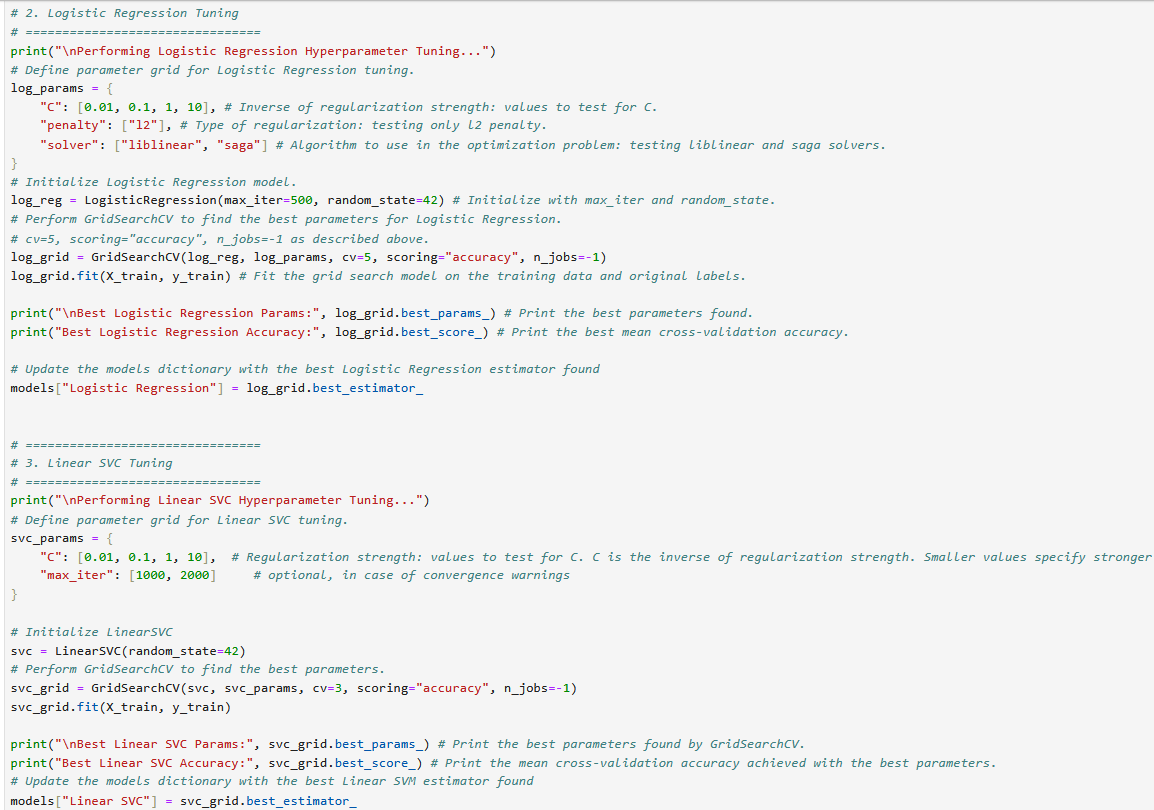
****

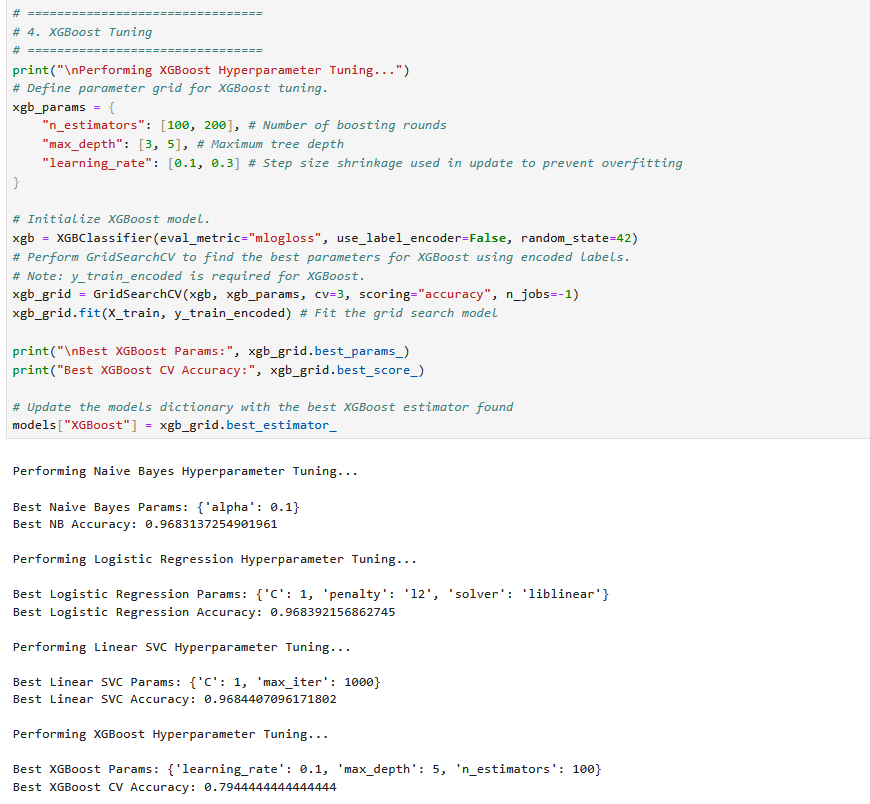
****

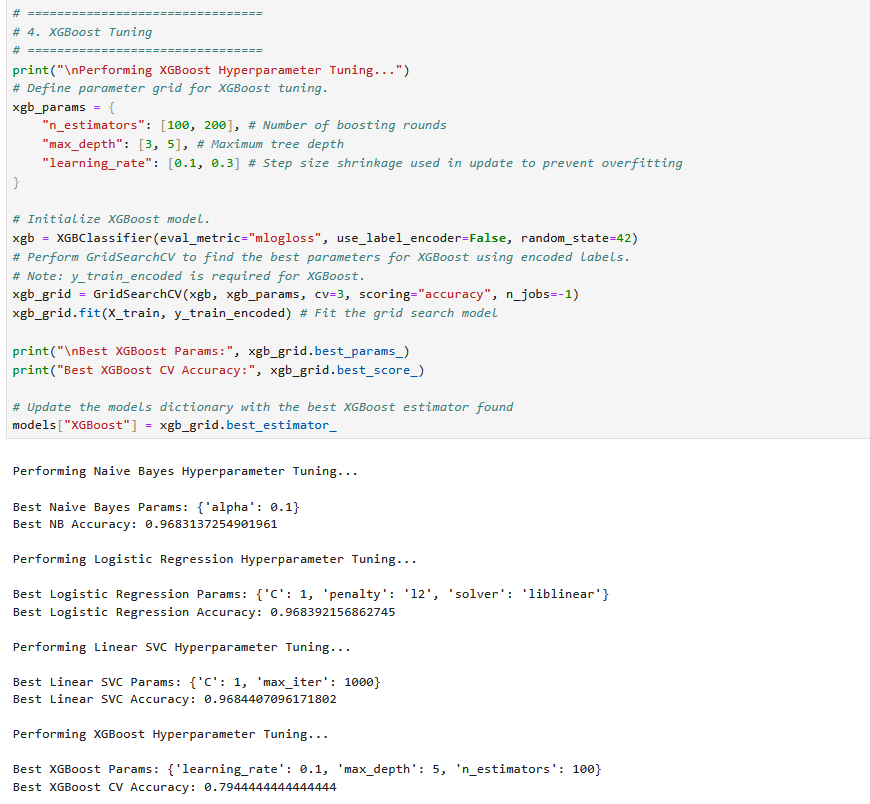
****

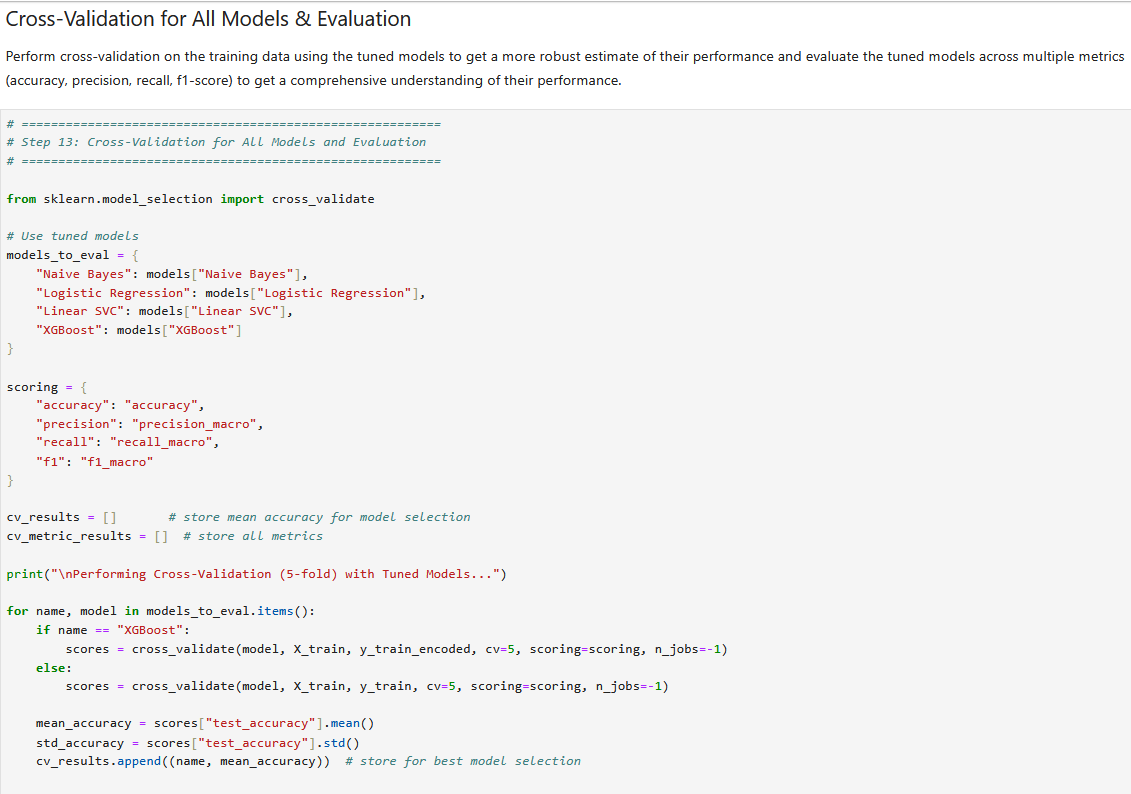
****

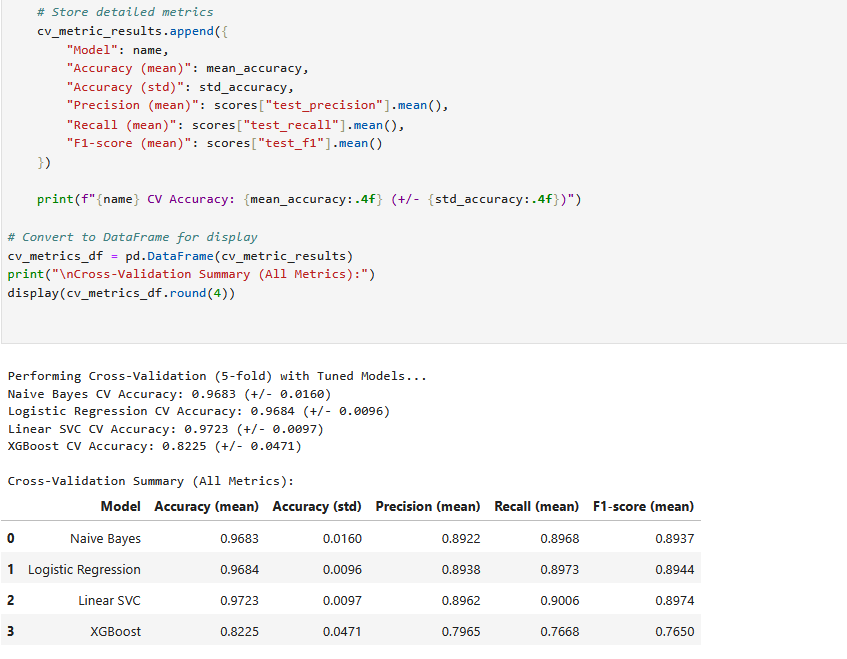
****

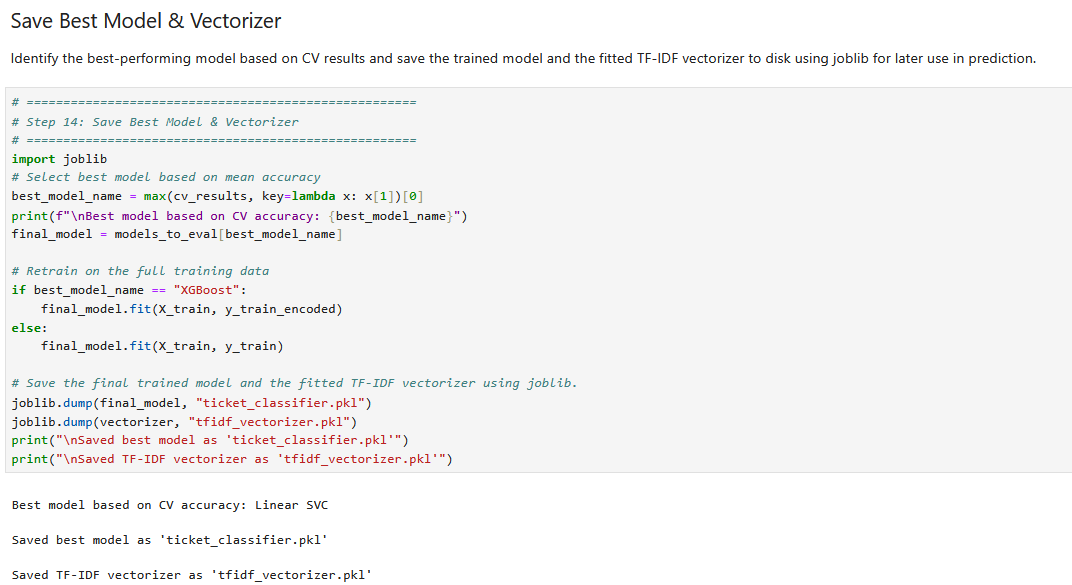
****

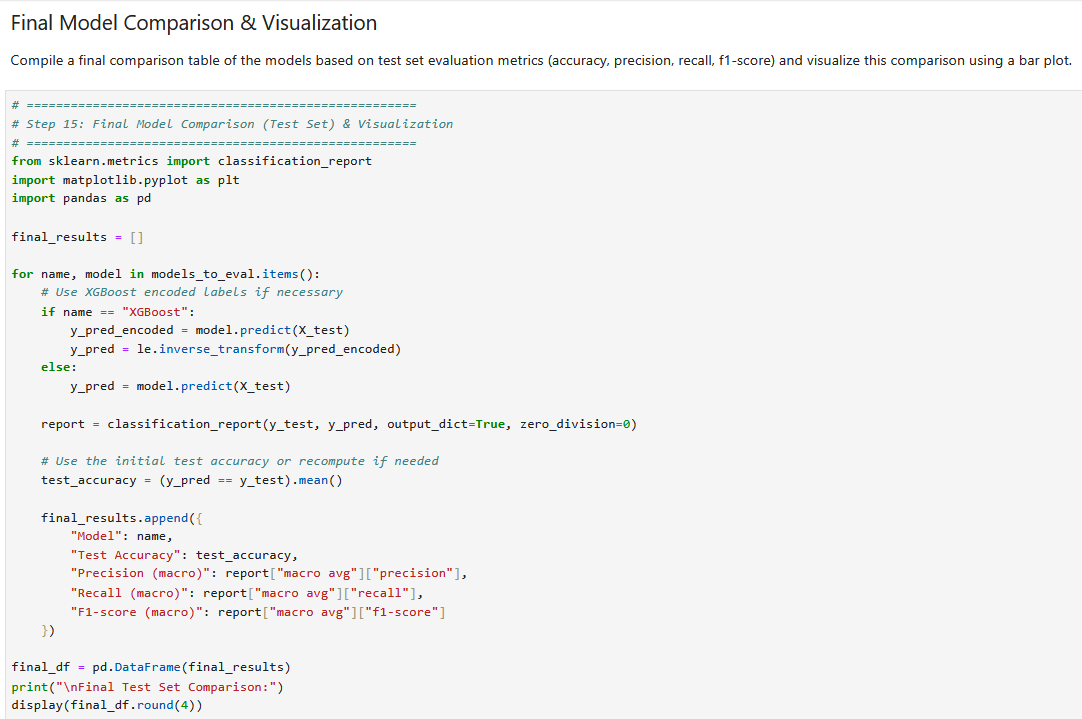
****

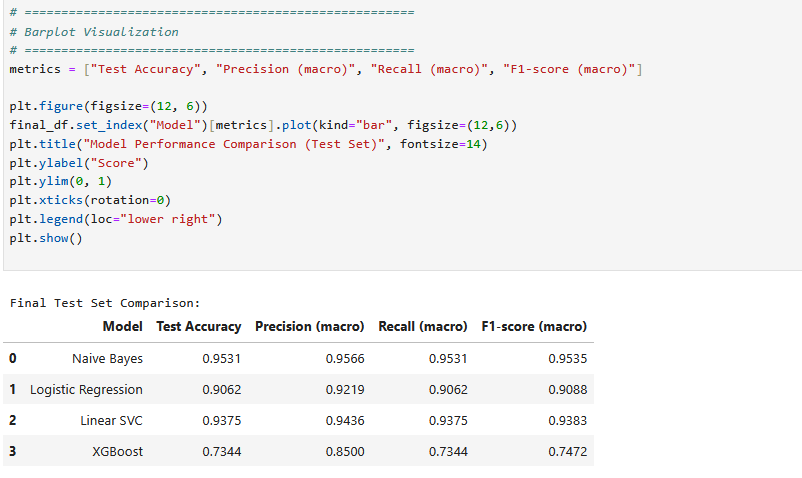
****

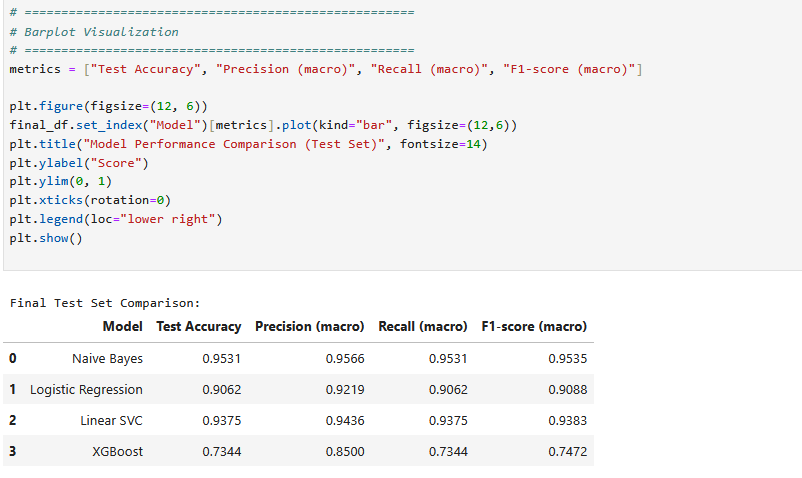
****

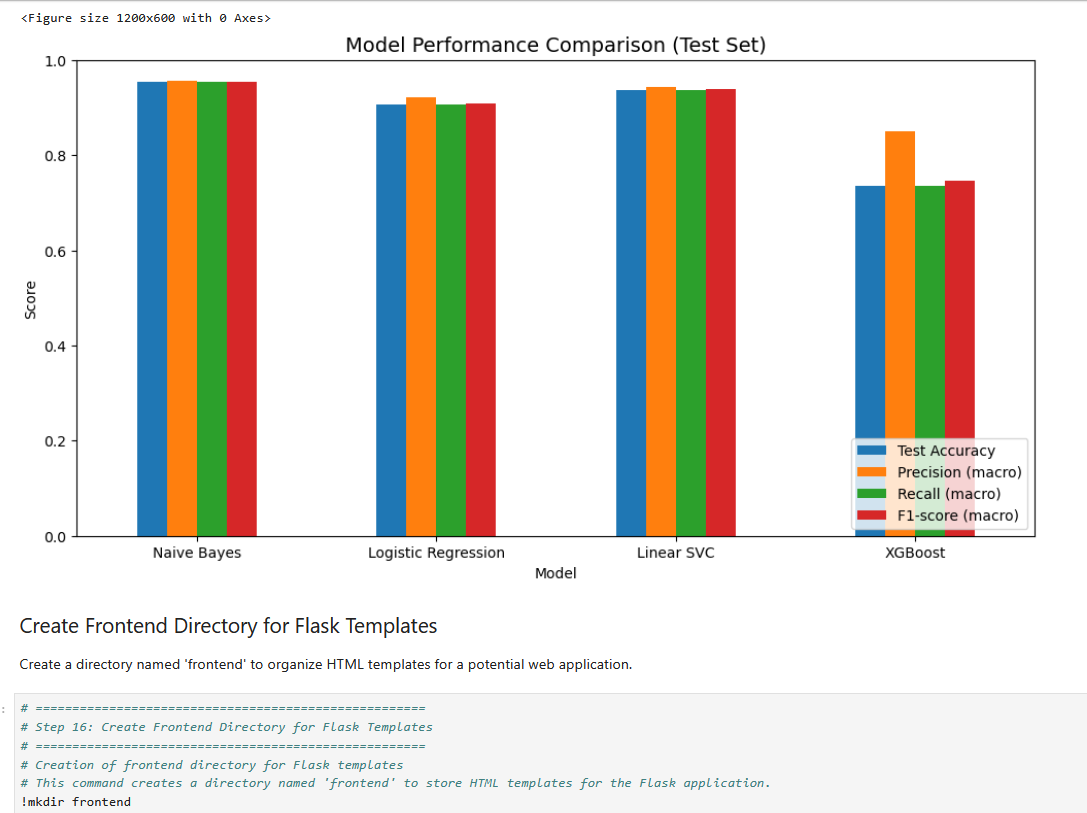
****

****

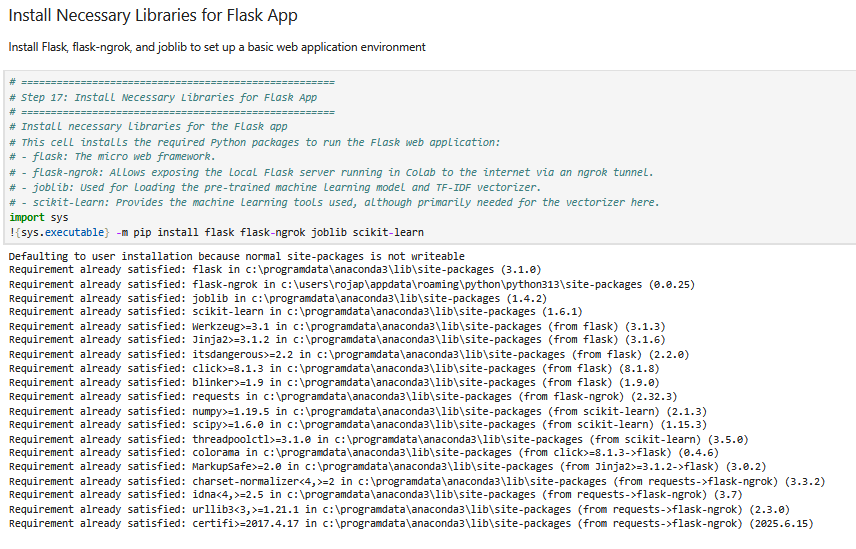
****

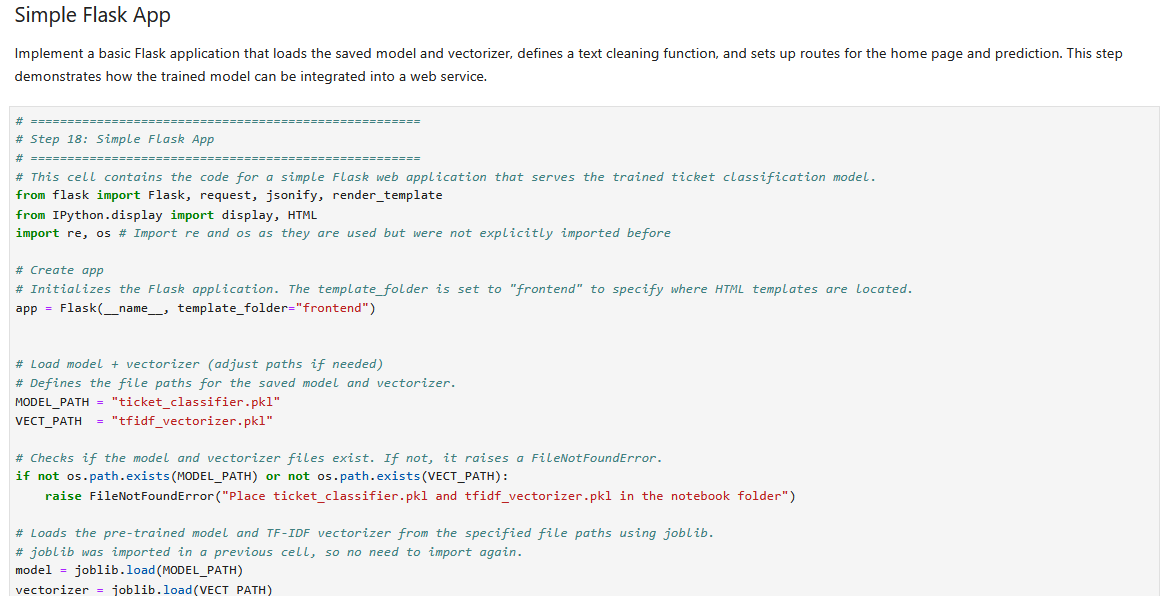
****

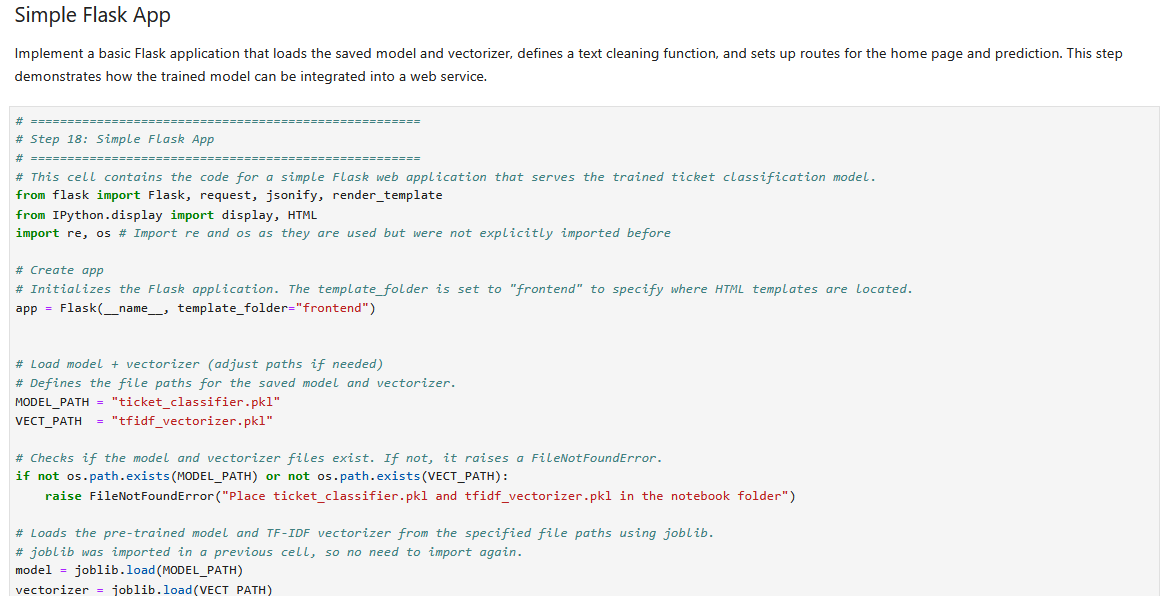
****

****

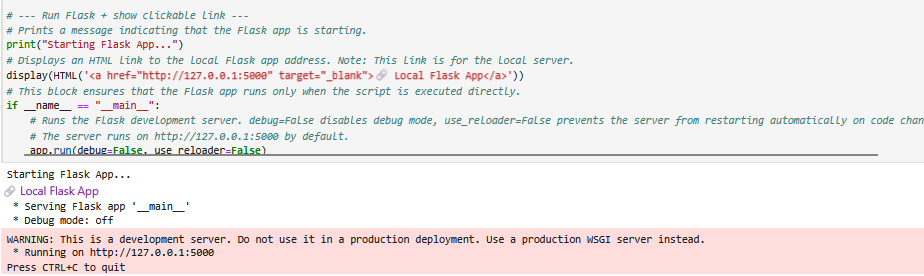
**After the above cell executed frontend folder is created , add index.html file into the folder manually.**

****

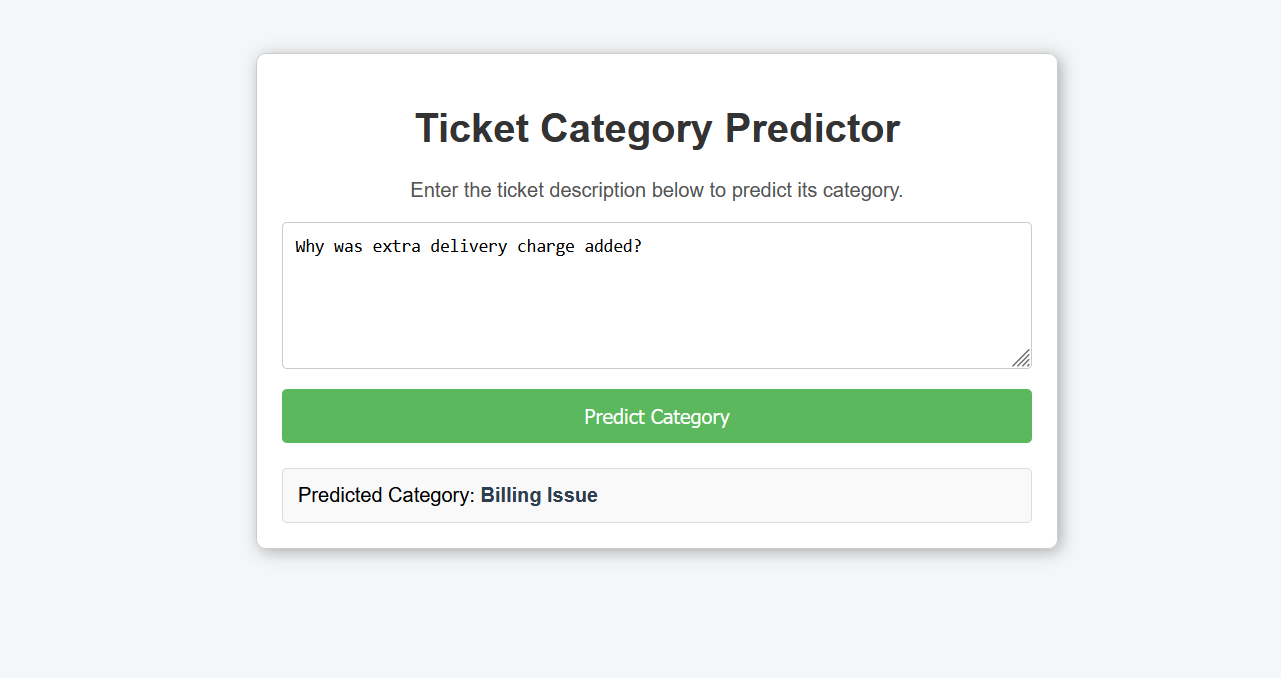
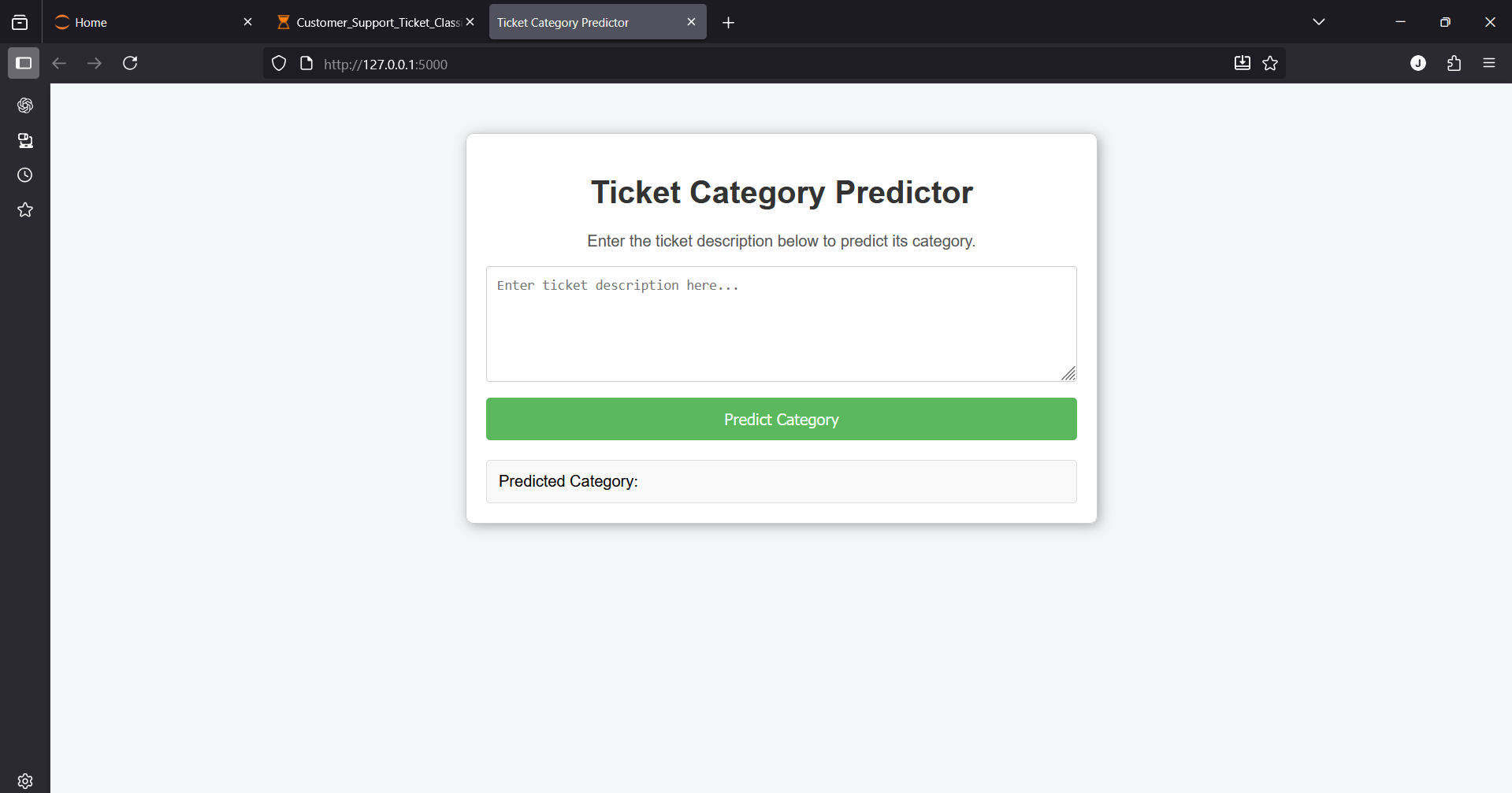
****

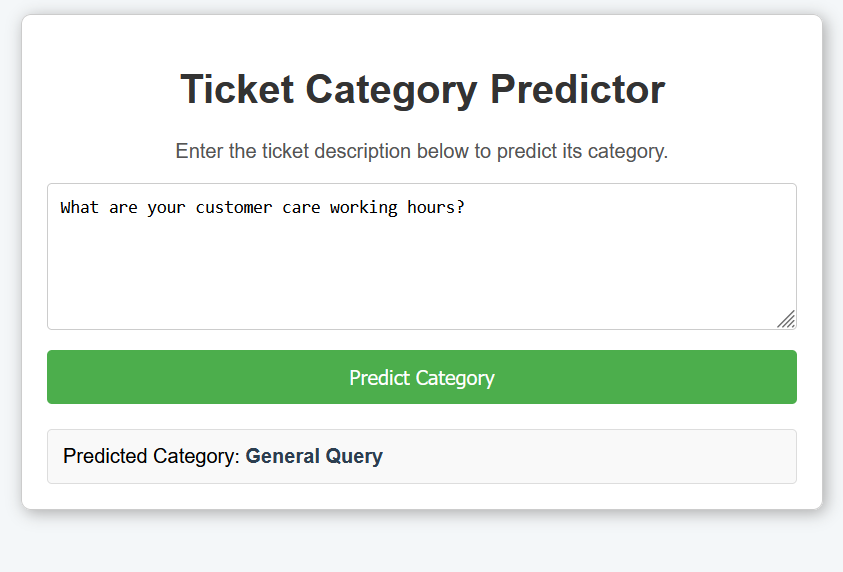
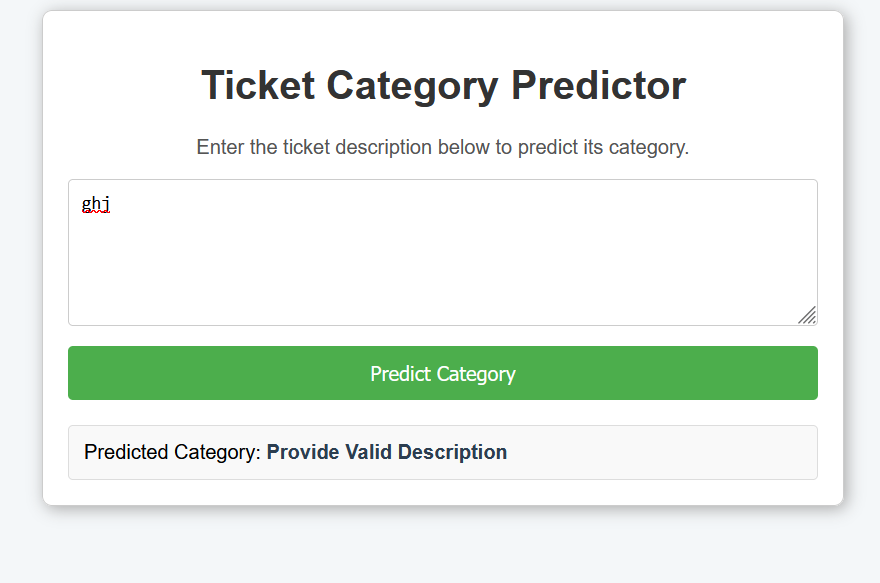
****

****

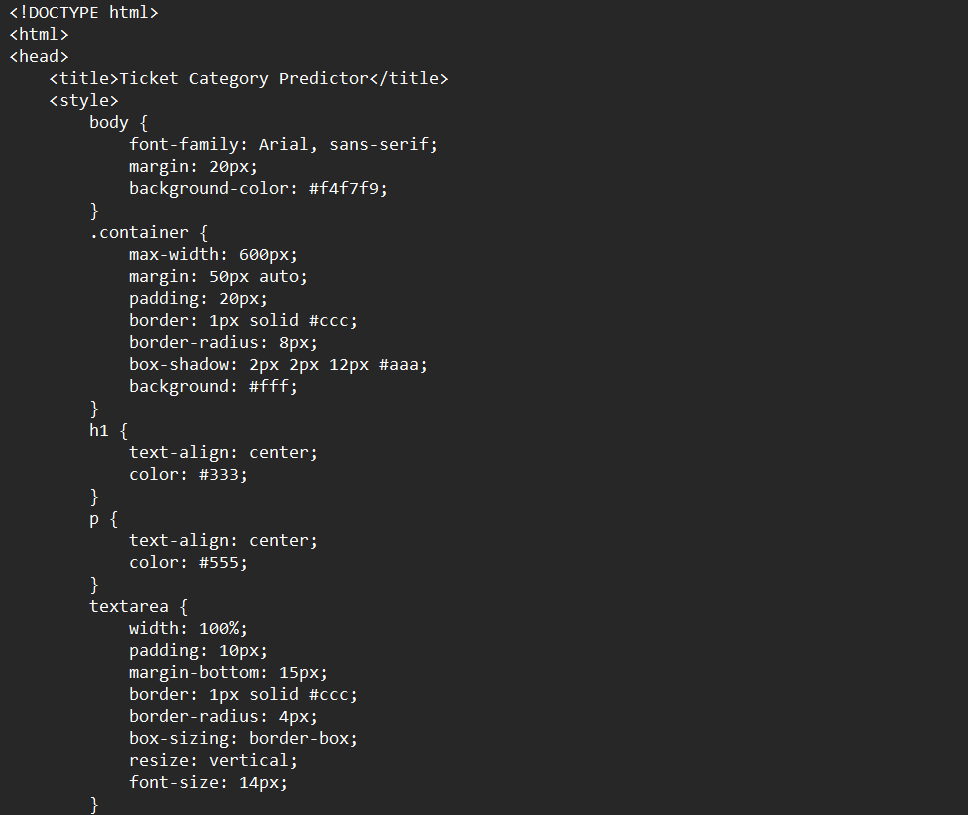
****

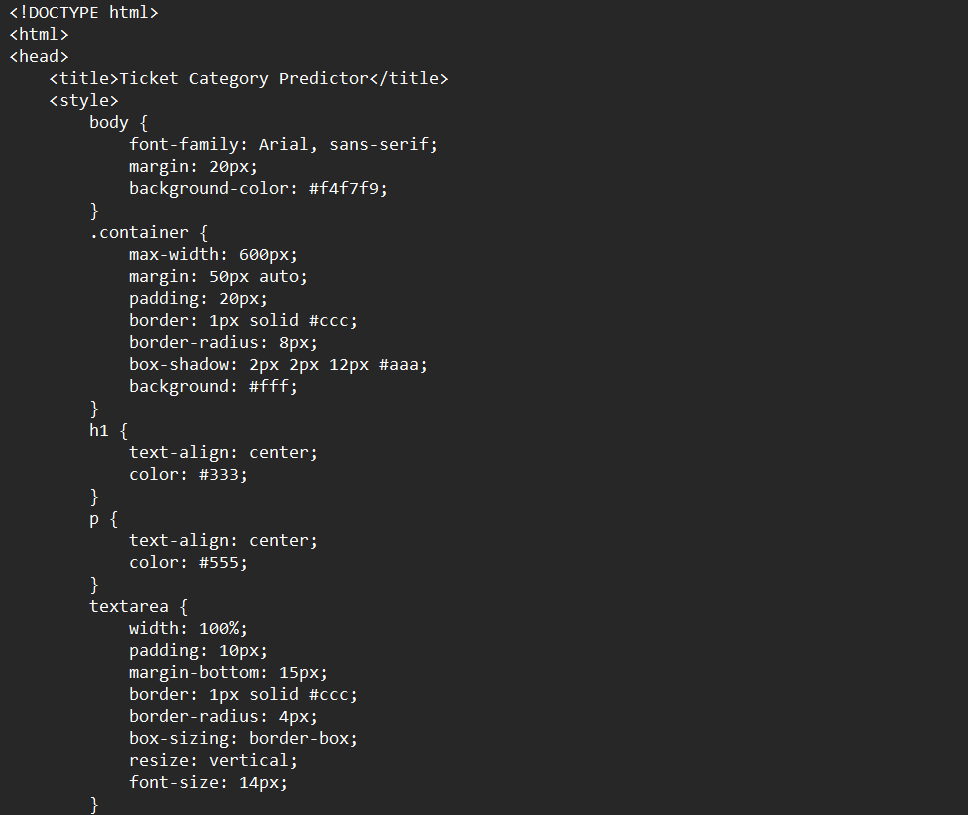
**Output Window & Some sample Test Cases**

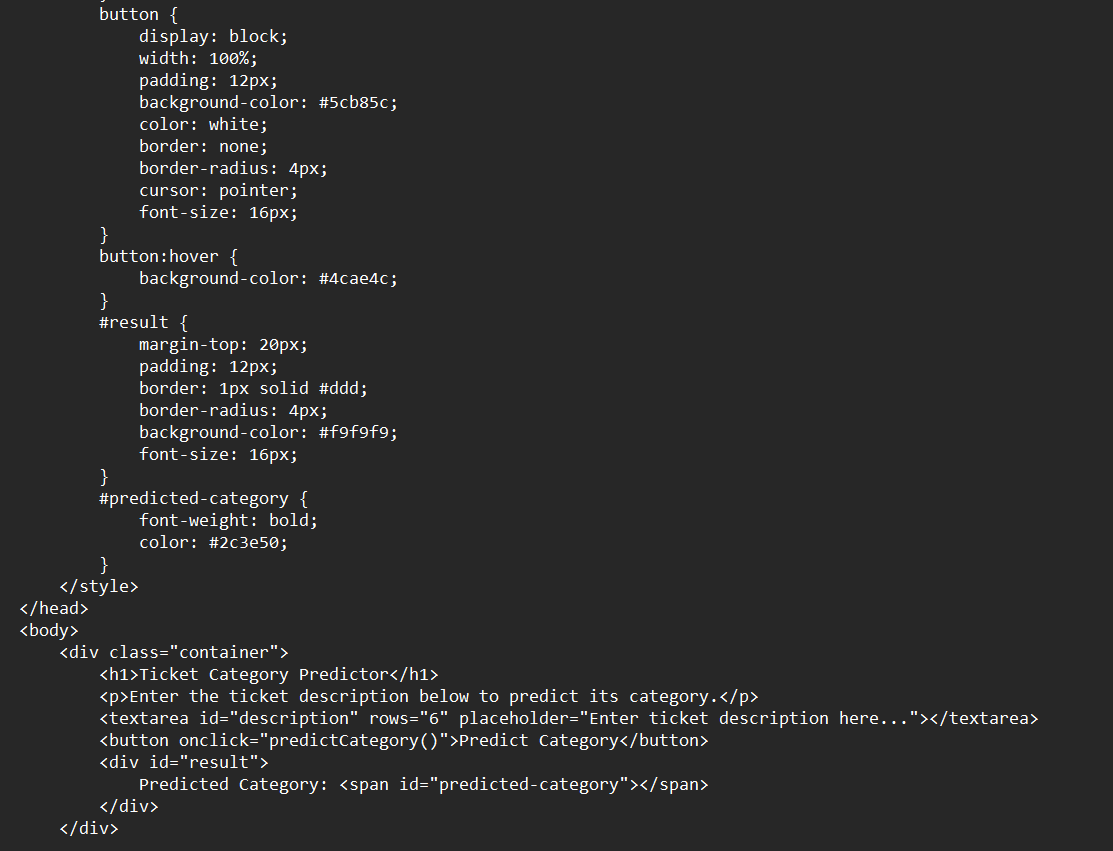
****

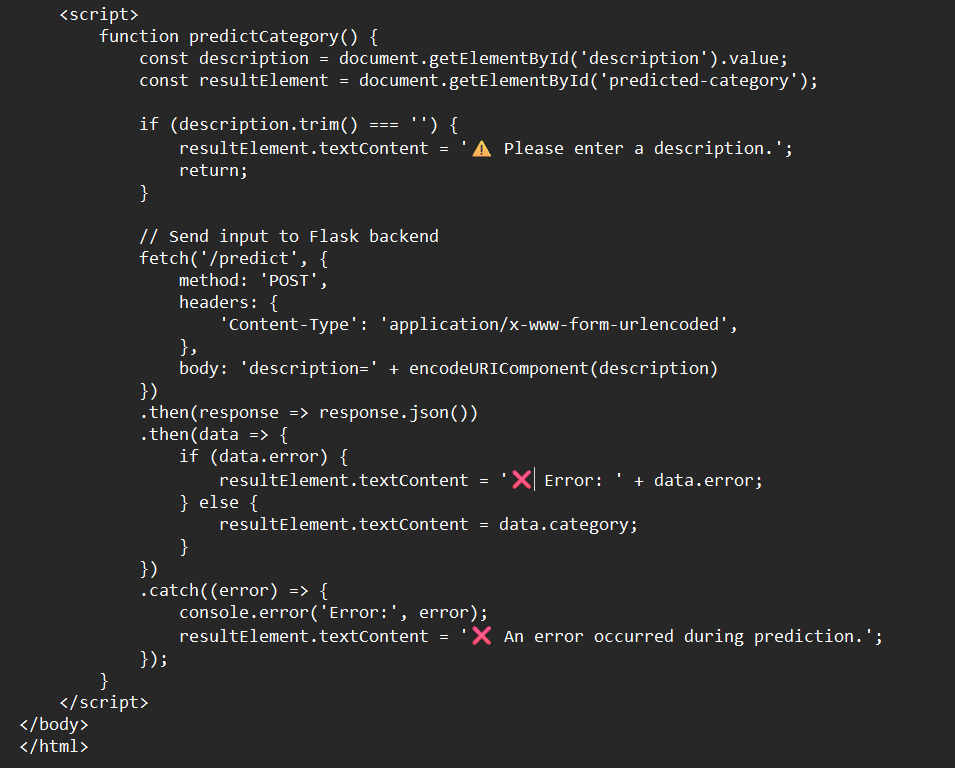
****

**Index.html**

****

****

****

****

**9.3 Instructions Reference**

To reproduce the project:

1. Open the notebook and load the dataset.
2. Run preprocessing, feature engineering, and model training cells.
3. Create the frontend/ folder and add index.html.
4. Run the Flask app cell and access the generated link.

**9.4** **Acronyms Used**

* **NLP** – Natural Language Processing
* **TF-IDF** – Term Frequency–Inverse Document Frequency
* **SVC** – Support Vector Classifier
* **CV** – Cross Validation

### **9.5 Summary Statement**

### This project demonstrates a complete end-to-end workflow for customer support ticket classification, starting from data preprocessing and model training to deployment through a Flask web application. The system validates both technical implementation and practical usability, showcasing how machine learning can effectively automate and streamline customer support operations.

**THE END**