DSA 504.2.0 Machine Learning I — Group Project — Final Report

Automatic Email Classification into: Academic, Official, News & Updates, Bills & Finance, Personal

(Using a TF-IDF + SVM pipeline)

By :

D.M.S.Eswarage - MSC/DSA/200

O.R.W.Perera - MSC/DSA/175

M.K.P.Bimsara MSC/DSA/183

C.S.Jayasinghe MSC/DSA/251

Abstract

This project implements an end-to-end system that classifies incoming Gmail messages into five categories: Academic, Official, News & Updates, Bills & Finance, Personal. Data were collected via the Gmail API, preprocessed to remove noise and normalize text, and transformed to numerical features using TF-IDF. A Support Vector Machine (LinearSVC) classifier inside a sklearn.Pipeline (with a ColumnTransformer to handle multiple fields) was trained and validated. The final model (email_sorting_svm.pkl) was serialized for deployment. Deployment was achieved through a Streamlit-based application integrated with the Gmail API, providing an interactive interface for real-time email sorting. Electron was further used to wrap the application as a cross-platform desktop tool. Evaluation included per-class precision, recall, F1-score, and confusion matrix analysis.

The full codebase is available on GitHub (Email-Classification-Problem), the process and a working demonstration is presented in the project video.

Evaluation includes per-class precision, recall and F1, confusion matrix analysis, and cross-validation.

Introduction

Email remains a primary medium for communication in academic, professional, and personal contexts. With the increasing volume of daily emails, automatic categorization is essential for improving productivity, ensuring timely responses, and maintaining organization.

Traditional manual email filtering is inefficient and error-prone, especially when dealing with large-scale data streams. Machine learning techniques provide a scalable solution by learning patterns from historical emails and classifying future messages into relevant categories.

This project addresses the challenge of automatic email classification using a machine learning pipeline built around TF-IDF vectorization and a Support Vector Machine classifier. The approach balances efficiency and interpretability, producing a reliable proof-of-concept system for real-world deployment.

1. Objectives

1. Build an accurate multiclass classifier for email sorting with five labels.
    - i. Academic → university notices, research updates
    - ii. Bills & Finance → bank statements, invoices
    - iii. News & Updates → newsletters, subscriptions
    - iv. Personal → personal conversations
    - v. Official → workplace / corporate emails
2. Implement a reproducible pipeline (from data extraction → cleaning → feature engineering → classification → deployment).
3. Use advanced preprocessing and exploratory analysis to ensure high-quality input features.

4. Evaluate the model with strong metrics (accuracy, precision, recall, F1-score, confusion matrix).
5. Deliver reusable artifacts:
    - Code repository (GitHub-Email-Classification-Problem)
    - Trained model (email_sorting_svm.pkl)
    - Documentation (This report)
    - Demo video (project video)

2. Data Collection

Emails were collected through the Gmail API using OAuth 2.0 authentication. The script `read.py` handled:

- Secure token management (`token.json`)
- Fetching message metadata (subject, sender, date, Gmail labels)
- Extracting plain text body content from encoded message parts

The collected data were stored in emails_dataset.csv, which served as the raw input for further cleaning and analysis.

3. Data Cleaning (Initial Stages)

Raw Gmail data often contain HTML tags, encodings, signatures, and repeated phrases. To address this, multiple custom scripts were developed:

- **`clean.py`**:
    - Removed HTML content using BeautifulSoup
    - Normalized whitespace
    - Lowercased text
    - Created `emails_dataset_clean.csv`
- **`deepclean.py`**:
    - Extracted sender email addresses
    - Removed unwanted characters, encodings, and symbols
    - Filtered out common boilerplate phrases (e.g., *unsubscribe, best regards*)
    - Produced a more refined dataset (`dataset1.csv`)

These cleaning stages ensured that the dataset was free from noise, consistent in labels, and optimized for feature extraction.

4. Exploratory Data Analysis (EDA)

Exploratory Data Analysis (EDA) is a critical step in any data science or machine learning project. It involves understanding the structure, characteristics, and patterns of the data before applying any predictive modeling techniques. In the context of an email sorting task, the goal is to analyze a dataset of emails to uncover insights that can help classify them into predefined categories. EDA process involves:

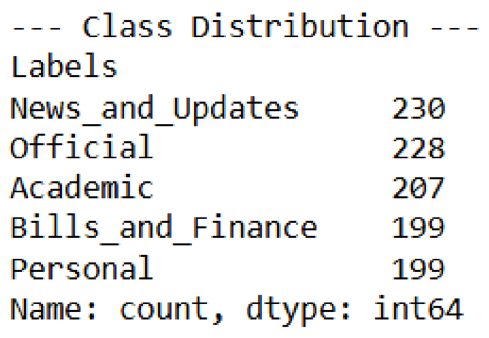
4.1 Load Dataset

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from collections import Counter
import re

# Load dataset
df = pd.read_csv("emails.csv")  # change file name if needed
print("Dataset Shape:", df.shape)
print("\nColumns:", df.columns.tolist())
```

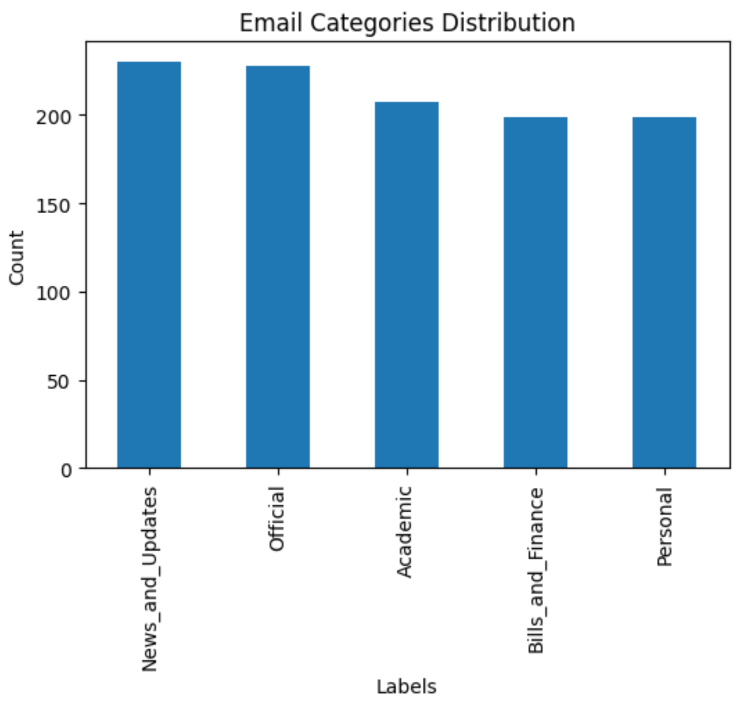
First import necessary libraries and load the dataset (emails_cleaned.csv) into a pandas DataFrame. Pandas is used to handle tabular data efficiently. Once loaded, the script prints out the shape (number of rows and columns) and the available columns so to understand what features are in the dataset.

4.2 Class Distribution

```
print("\n--- Class Distribution ---")
label_counts = df['Labels'].value_counts()
print(label_counts)

plt.figure(figsize=(6,4))
label_counts.plot(kind='bar', title="Email Categories Distribution")
plt.ylabel("Count")
plt.show()
```



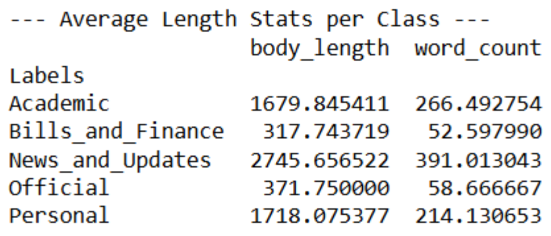
```
--- Class Distribution ---
Labels
News_and_Updates     230
Official             228
Academic             207
Bills_and_Finance    199
Personal             199
Name: count, dtype: int64
```



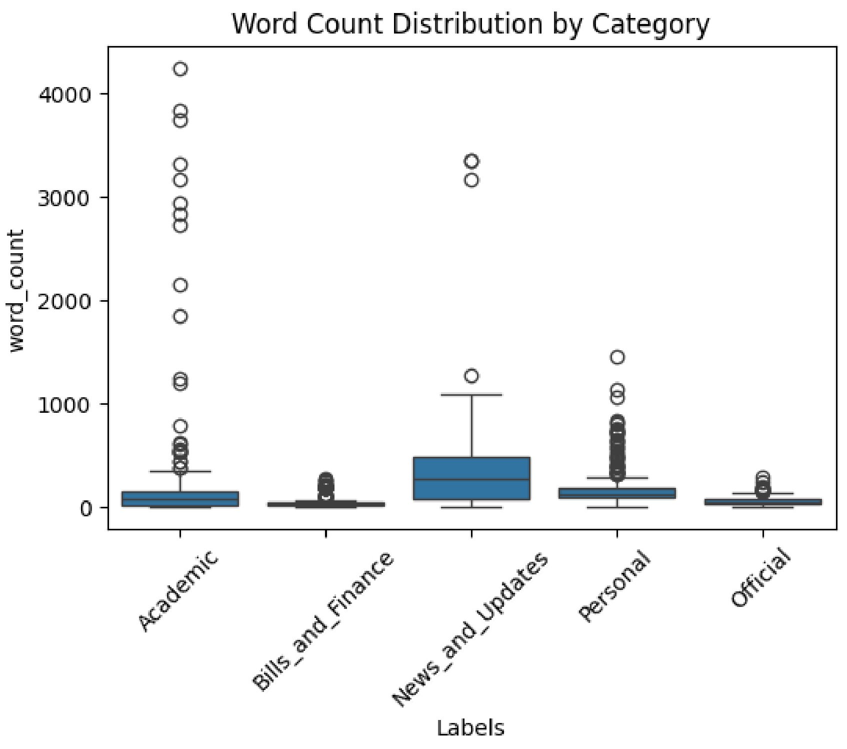
Here, decided to check how the dataset is distributed across categories (e.g., Academic, Bills & Finance, Personal). This is done using the value_counts() function on the Labels column. Printing these counts shows us whether the dataset is **imbalanced** (some classes have many more examples than others). A bar chart is also generated to visualize this distribution, which makes it easier to spot any imbalances.

4.3 Email Length Statistics

```
df['body_length'] = df['body'].str.len()
df['word_count'] = df['body'].str.split().str.len()

print("\n--- Average Length Stats per Class ---")
print(df.groupby('Labels')[['body_length','word_count']].mean())
plt.figure(figsize=(6,4))
sns.boxplot(x="Labels", y="word_count", data=df)
plt.title("Word Count Distribution by Category")
plt.xticks(rotation=45)
plt.show()
```



```
--- Average Length Stats per Class ---
                  body_length  word_count
Labels
Academic          1679.845411  266.492754
Bills_and_Finance  317.743719   52.597990
News_and_Updates  2745.656522  391.013043
Official           371.750000   58.666667
Personal          1718.075377  214.130653
```



In this step, the script calculates two useful metrics for each email:

- **Body length**: number of characters in the email body.
- **Word count**: number of words in the email body.

The averages of these metrics are then computed for each category. This helps identify differences in writing style.

4.4 Frequent Words per Category

```
def get_top_words(texts, n=15):
    words = " ".join(texts.astype(str)).lower()
    words = re.findall(r'\b\w+\b', words)
    return Counter(words).most_common(n)

print("\n--- Frequent Words per Category ---")
for label in df['Labels'].unique():
    top_words = get_top_words(df[df['Labels']==label]['clean_text'])
    print(f"\n{label}: {top_words}")
```

```
--- Frequent Words per Category ---

Academic: [('ai', 501), ('data', 318), ('new', 268), ('course', 262), ('b', 216), ('models', 206), ('model', 190), ('dsa', 16
0), ('like', 156), ('c', 154), ('email', 145), ('courses', 141), ('learning', 130), ('science', 129), ('please', 117)]

Bills_and_Finance: [('please', 250), ('x', 199), ('payment', 150), ('lkr', 140), ('email', 118), ('mail', 110), ('landing', 11
0), ('transaction', 91), ('use', 90), ('p', 89), ('html', 88), ('cid', 88), ('textmail', 88), ('src', 88), ('otp', 84)]

News_and_Updates: [('c', 885), ('f', 836), ('b', 754), ('email', 699), ('e', 634), ('r', 618), ('ai', 493), ('fl', 483), ('nul
l', 474), ('job', 456), ('view', 445), ('u', 429), ('al', 427), ('read', 414), ('w', 404)]

Personal: [('email', 1293), ('null', 862), ('f', 816), ('b', 662), ('eml', 544), ('urn', 495), ('u', 486), ('ali', 474), ('g',
446), ('aemail', 409), ('lipi', 399), ('apage', 399), ('midsig', 399), ('linkedin', 399), ('c', 361)]

Official: [('ai', 675), ('gitlab', 528), ('com', 425), ('email', 283), ('aoye', 257), ('ninjas', 250), ('deployment', 248), ('p
ipeline', 248), ('merge', 218), ('requests', 164), ('note', 163), ('view', 159), ('receiving', 146), ('shell', 140), ('accoun
t', 134)]
```

This script then explores the most used words in each category. It does this by combining all cleaned text (clean_text column), tokenizing it into words, and counting the frequency of each word using Python's Counter. For each label (class), the top 15 most frequent words are displayed. This gives insights into the **keywords that characterize each type of email**.

4.5 Sender Domain Analysis

```
df['domain'] = df['from'].str.split('@').str[-1]
domain_counts = df.groupby('Labels')['domain'].value_counts().groupby(level=0).head(5)

print("\n--- Top Sender Domains per Category ---")
print(domain_counts)
```

```
--- Top Sender Domains per Category ---
Labels             domain
Academic           sci.sjp.ac.lk              51
                   classroom.google.com       30
                   academia-mail.com          15
                   e.udemymail.com            13
                   deeplearning.ai            11
Bills_and_Finance  sampath.lk                 54
                   commerce.temuemail.com     19
                   dialog.lk                  17
                   combank.net                10
                   google.com                  9
News_and_Updates   redditmail.com             29
                   dare2compete.news          18
                   linkedin.com               18
                   quora.com                  18
                   ziprecruiter.com           18
Official           mg.gitlab.com             144
                   company.com                20
                   allxon.net                 13
                   promiseq.com               12
                   mail.notion.so             12
Personal           facebookmail.com           89
                   linkedin.com               53
                   tiktok.com                  6
                   telegram.org                5
                   accounts.google.com         5
Name: count, dtype: int64
```

Next, the script looks at the email sender addresses to find the **most common domains** (the part after @, like gmail.com or sjp.ac.lk) per category. This reveals the typical sources of each type of email. For instance, academic emails may often come from.ac.lk university domains, while promotional emails might come from newsletter.com domains. This analysis can help with rule-based filtering and feature engineering for machine learning.

5. Data Preprocessing and Feature Engineering

Before training machine learning models, the raw email data needed to be cleaned and transformed into a structured format suitable for classification. The preprocessing pipeline involved the following steps:

5.1. Loading the Dataset

The dataset (`Finaldataset.csv`) was loaded into a pandas DataFrame. This allowed inspection of the raw data (subject, body, and labels) and ensured that the dataset could be manipulated efficiently.

5.2. Cleaning Labels

The labels contained minor inconsistencies such as extra spaces and varying naming styles. To address this:

- Leading and trailing spaces were removed.
- Multi-word labels were standardized by replacing spaces with underscores (e.g., *"Bills and Finance"* → *"Bills_and_Finance"*).

This step ensured consistent labels for training and evaluation.

5.3. Cleaning the Email Text

Email text often contains significant noise such as links, email addresses, HTML tags, and stopwords. To make the data more informative for the model:

- Converted all text to lowercase.
- Removed URLs and email addresses (which add noise but little semantic meaning).
- Removed all non-alphabetic characters, retaining only letters.
- Normalized whitespace.
- Removed common English stopwords using **NLTK**.
- Combined the **subject** and **body** into a single text field (`clean_text`) to provide full context of each email.

This transformation emphasized semantically meaningful words such as *"payment"*, *"account"*, *"interview"*, while discarding irrelevant tokens like links and signatures.

5.4. Handling Missing Values

Some emails lacked subjects or bodies. These were replaced with empty strings to avoid errors during further processing. A summary of missing values was checked to ensure data integrity.

5.5. Encoding Labels

Since machine learning algorithms require numeric labels, the categorical labels were encoded using **LabelEncoder**. For example:

Category	**Encoded Label**
Official	0
Personal	1
Academic	2
Bills_and_Finance	3
News_and_Updates	4

This step allowed the models to treat the output variable as a classification target.

5.6. Train–Test Split

The dataset was split into training (80%) and test (20%) subsets using **stratified sampling**. Stratification preserved the proportion of classes across both sets, preventing imbalance and ensuring fair evaluation of the model's performance.

5.7. Feature Extraction with TF-IDF

Text data was vectorized into numerical form using **TF-IDF (Term Frequency–Inverse Document Frequency)**.

- TF measures how often a word occurs in a document.
- IDF reduces the weight of common words and emphasizes rarer, more informative terms.
- Both unigrams (single words) and bigrams (pairs of words) were used.
- Feature space was limited to the **top 5000 terms** to balance performance and efficiency.

This step converted raw email text into high-dimensional sparse vectors, which could then be used as input to classification algorithms.

6. Model Selection & Training

- Candidate models evaluated: Naive Bayes Logistic Regression, Random Forest, SVM (LinearSVC).
- Chosen model: LinearSVC (SVM with linear kernel) — generally performs well on high-dimensional sparse text. It is efficient when used with TF-IDF and supports class_weight='balanced' for imbalanced classes.
- Training recipe:

6.1 Imports and Libraries

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import MultinomialNB
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import LinearSVC
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score,
classification_report, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
import joblib
```

The first section of the code imports essential Python libraries. Pandas is used to manage and manipulate tabular data. From scikit-learn, several modules are brought in: train_test_split to divide the dataset, TfidfVectorizer to transform text into numerical vectors, and OneHotEncoder to convert categorical features into machine-readable form. A ColumnTransformer allows different preprocessing steps to be applied to different columns of the dataset, while Pipeline ties preprocessing and model training together in one unified process. The script uses four machine learning models: Logistic Regression, Naive Bayes, Random Forest, and Support Vector Machine (SVM). To evaluate these models, metrics such as accuracy, precision, recall, F1-score, and confusion matrices are imported. Finally, matplotlib and seaborn are used to plot results, and joblib is used to save and load trained models.

6.2 Load Dataset

```
df = pd.read_csv("emails_cleaned.csv")

X = df[["clean_text", "from", "gmail_labels"]]
y = df["label_encoded"]
```

The dataset is loaded from a CSV file named emails_cleaned.csv. The input features consist of three columns: the cleaned email text consists of subject and body combined (clean_text), the sender of the email (from), and the Gmail category labels (gmail_labels). The target variable is label_encoded, which contains numerical representations of the categories the emails belong to. This setup ensures that the model has access to textual content, sender information, and existing Gmail categories as predictors, while training it to predict the encoded labels.

6.3 Train-Test Split

```
X_train, X_test, y_train, y_test = train_test_split(
   X, y, test_size=0.2, random_state=42, stratify=y
)
```

To evaluate the model, the dataset is split into training and testing subsets using an 80–20 ratio. The argument stratify=y ensures that the class distribution is preserved in both sets, avoiding imbalances where certain labels might appear only in one subset. A fixed random state is used for reproducibility.

6.4 Preprocessing

```
preprocessor = ColumnTransformer(
   transformers=[
      ("text", TfidfVectorizer(max_features=5000, ngram_range=(1,2)), "clean_text"),
      ("from", TfidfVectorizer(max_features=1000), "from"),
      ("gmail_labels", OneHotEncoder(handle_unknown="ignore"), ["gmail_labels"])
   ]
)
```

The preprocessing stage is handled by a ColumnTransformer. Since the dataset contains both text and categorical features, each column requires a different transformation. The clean_text column is transformed using TF-IDF vectorization with unigrams and bigrams, limited to 5000 features. This captures not only single words but also word pairs that provide richer contextual information. The from column is also vectorized with TF-IDF but limited to 1000 features, since email addresses are usually shorter.
Finally, the gmail_labels column is encoded using one-hot encoding, which converts categorical values into binary vectors, ensuring they can be interpreted by machine learning algorithms. By combining these transformations, the script produces a consistent numeric feature space suitable for training models.

6.5 Models Defined

```
models = {
   "Logistic Regression": LogisticRegression(max_iter=1000),
   "Naive Bayes": MultinomialNB(),
   "Random Forest": RandomForestClassifier(n_estimators=200, random_state=42),
   "SVM": LinearSVC(max_iter=2000, random_state=42)
}
```

The code evaluates four supervised learning models. Logistic Regression is included as a strong baseline for text classification. Multinomial Naive Bayes, which is widely used for text-based tasks, is another candidate. A Random Forest classifier with 200 decision trees provides a robust ensemble approach, often effective with mixed feature types.
Finally, a Linear Support Vector Machine (SVM) with extended iterations is used; SVMs are known for their strong performance in high-dimensional text classification tasks. Each of these models is trained and compared under identical preprocessing conditions.

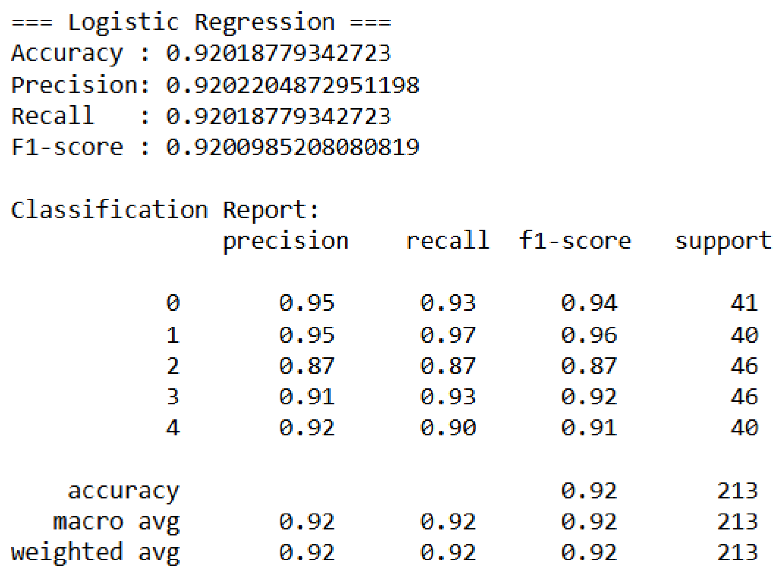
6.6 Evaluation Function

```
def evaluate_model(name, model, X_train, X_test, y_train, y_test):
    pipe = Pipeline(steps=[("preprocessor", preprocessor), ("classifier", model)])
    pipe.fit(X_train, y_train)
    preds = pipe.predict(X_test)

    acc = accuracy_score(y_test, preds)
    prec = precision_score(y_test, preds, average="weighted")
    rec = recall_score(y_test, preds, average="weighted")
    f1 = f1_score(y_test, preds, average="weighted")
```

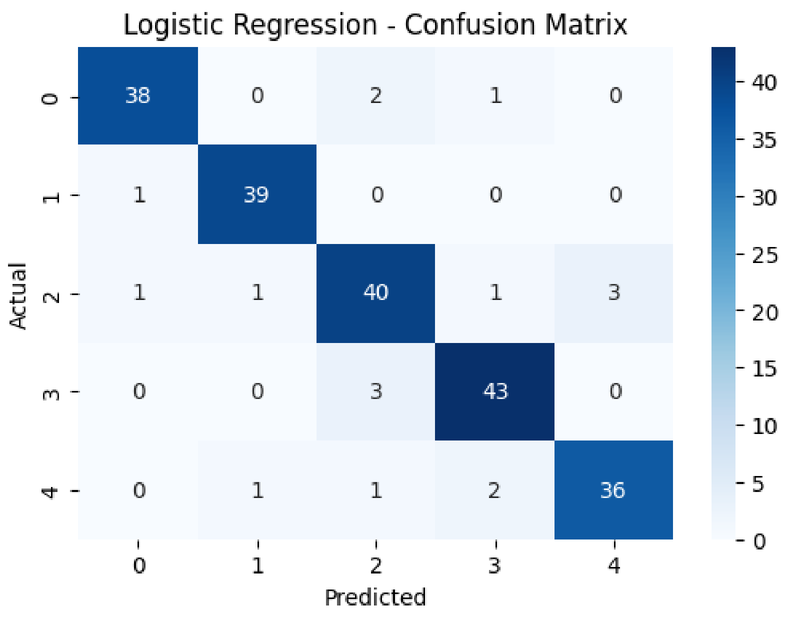
A reusable evaluation function is defined to automate training, prediction, and performance measurement. Within this function, a pipeline is created that chains the preprocessing step with the classifier, ensuring a clean and repeatable workflow. After fitting the pipeline on training data, predictions are made on the test set.
Performance is measured with accuracy, precision, recall, and F1-score, all calculated using weighted averages to account for class imbalances. The function also prints a detailed classification report and visualizes the confusion matrix as a heatmap. This matrix highlights how well the model distinguishes between categories and where it makes mistakes. Results are returned in a structured format so they can be compared later.

6.7 Run All Evaluations and Comparison

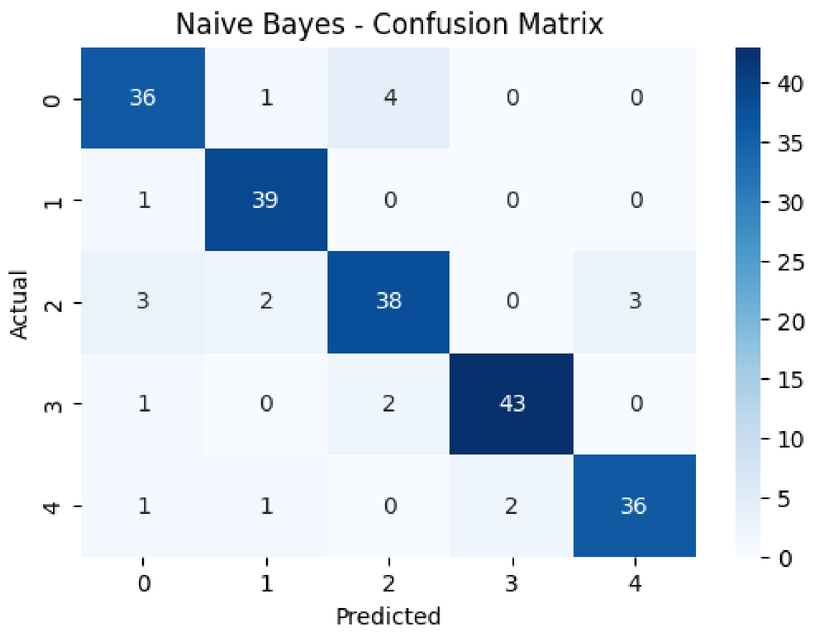
```
results = []
for name, model in models.items():
    results.append(evaluate_model(name, model, X_train, X_test, y_train, y_test))
```



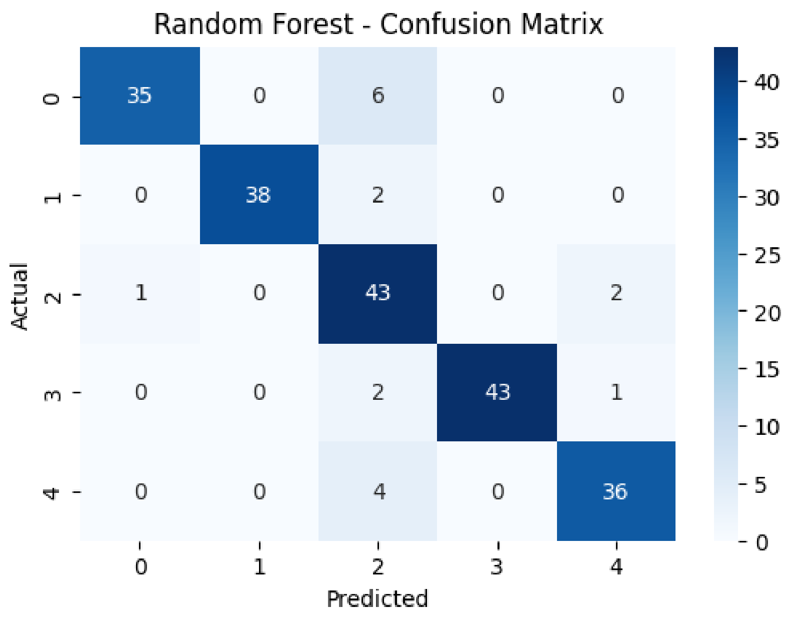
```
=== Logistic Regression ===
Accuracy : 0.92018779342723
Precision: 0.9202204872951198
Recall   : 0.92018779342723
F1-score : 0.9200985208080819

Classification Report:
              precision    recall  f1-score   support

           0       0.95      0.93      0.94        41
           1       0.95      0.97      0.96        40
           2       0.87      0.87      0.87        46
           3       0.91      0.93      0.92        46
           4       0.92      0.90      0.91        40

    accuracy                           0.92       213
   macro avg       0.92      0.92      0.92       213
weighted avg       0.92      0.92      0.92       213
```



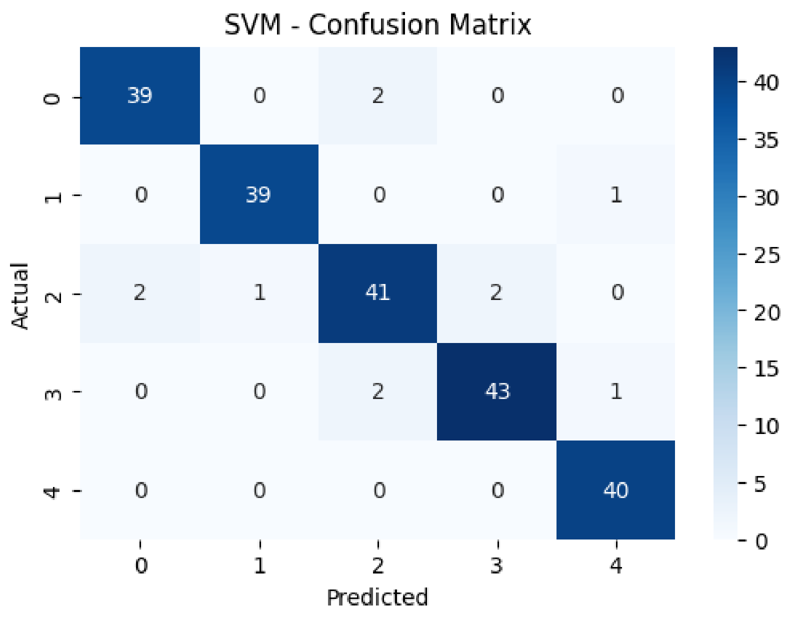
```
=== Naive Bayes ===
Accuracy : 0.9014084507042254
Precision: 0.9015391179164516
Recall   : 0.9014084507042254
F1-score : 0.9010765005893316

Classification Report:
              precision    recall  f1-score   support

           0       0.86      0.88      0.87        41
           1       0.91      0.97      0.94        40
           2       0.86      0.83      0.84        46
           3       0.96      0.93      0.95        46
           4       0.92      0.90      0.91        40

    accuracy                           0.90       213
   macro avg       0.90      0.90      0.90       213
weighted avg       0.90      0.90      0.90       213
```



```
=== Random Forest ===
Accuracy : 0.9154929577464789
Precision: 0.9271640489209059
Recall   : 0.9154929577464789
F1-score : 0.9181220077096541

Classification Report:
              precision    recall  f1-score   support

           0       0.97      0.85      0.91        41
           1       1.00      0.95      0.97        40
           2       0.75      0.93      0.83        46
           3       1.00      0.93      0.97        46
           4       0.92      0.90      0.91        40

    accuracy                           0.92       213
   macro avg       0.93      0.91      0.92       213
weighted avg       0.93      0.92      0.92       213
```

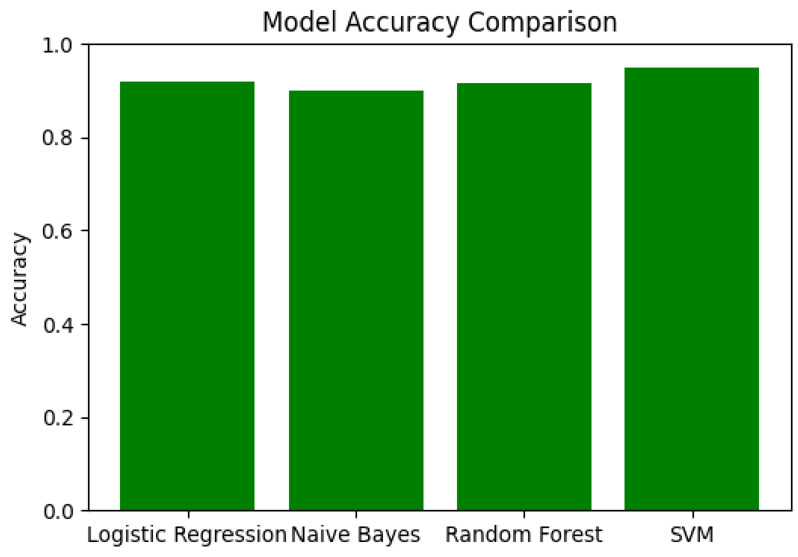


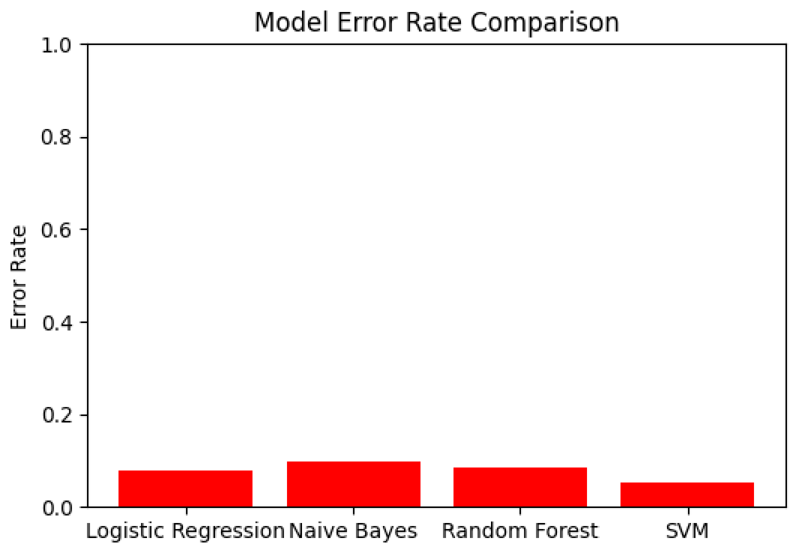
```
=== SVM ===
Accuracy : 0.9483568075117371
Precision: 0.9481779566286609
Recall   : 0.9483568075117371
F1-score : 0.948110174492861

Classification Report:
              precision    recall  f1-score   support

           0       0.95      0.95      0.95        41
           1       0.97      0.97      0.97        40
           2       0.91      0.89      0.90        46
           3       0.96      0.93      0.95        46
           4       0.95      1.00      0.98        40

    accuracy                           0.95       213
   macro avg       0.95      0.95      0.95       213
weighted avg       0.95      0.95      0.95       213
```



```
results_df = pd.DataFrame(results)
results_df["Error Rate"] = 1 - results_df["Accuracy"]

#Accuracy Plot
plt.bar(results_df["Model"], results_df["Accuracy"], color="green")

#Error Plot
plt.bar(results_df["Model"], results_df["Error Rate"], color="red")
```

The script iterates over all defined models, applies the evaluation function, and collects results in a list. These results are then combined into a DataFrame, where an additional column for error rate (1 - accuracy) is calculated. Two plots are produced: one showing the accuracy of each model in green, and another showing their error rates in red. These visual comparisons make it easy to identify the strongest model at a glance which is the SVM model.





6.8 Save Best Model

```
best_model = Pipeline(steps=[("preprocessor", preprocessor),
                    ("classifier",LinearSVC(max_iter=2000, random_state=42))])

best_model.fit(X_train, y_train)
joblib.dump(best_model, "email_sorting_svm.pkl")
```

After comparing results, the script selects the Support Vector Machine as the best-performing model. A final pipeline combining preprocessing and the SVM classifier is trained on the training set and then saved to disk using joblib. Saving the full pipeline ensures that future predictions can be made on raw email inputs without manually applying preprocessing steps again.

6.9 Load and Predict New Emails

```
loaded_model = joblib.load("email_sorting_svm.pkl")

sample = pd.DataFrame([{
    "clean_text": "Meeting scheduled for tomorrow",
    "from": "boss@company.com",
    "gmail_labels": "Work"
}])

pred = loaded_model.predict(sample)[0]
print("Predicted label:", pred)
```

The saved model is reloaded using joblib.load. New email samples, containing the same feature fields (clean_text, from, and gmail_labels), are passed directly into the pipeline for prediction.

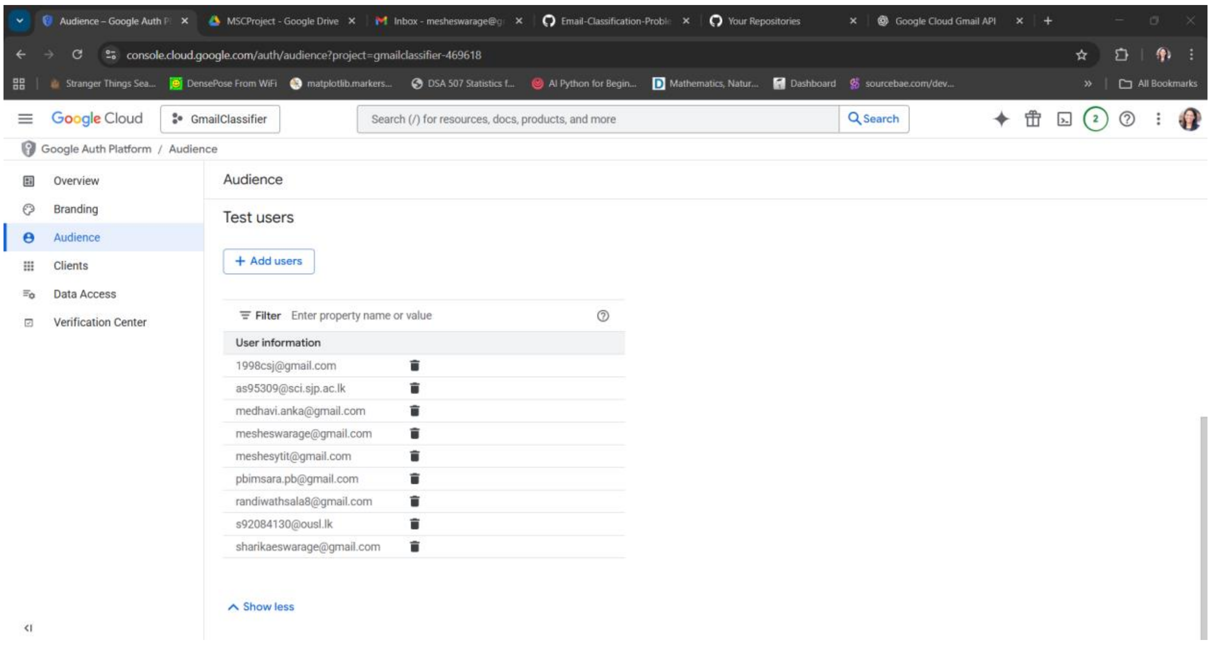
Since preprocessing is embedded in the saved pipeline, the user does not need to handle text vectorization or label encoding manually.
Predictions are returned as encoded labels, showing how the model would classify incoming emails in practice. This makes the system ready for deployment in a real-world email sorting application.

7. Deployment

7.1. Connect Gmail Accounts by Calling Google API

(Githup_Repository/ Deployment_p / connection.py)

Used the Google Gmail API through the Google Cloud Platform to access and manage emails. This first authenticates the user using OAuth 2.0 credentials from credentials.json (downloaded from Google Cloud Console), and stores the access token in token.json. The script connects to the Gmail service, fetches emails from specific labels like INBOX, SENT, SPAM, and TRASH.


Audience – Google Auth P
MSCProject - Google Drive
Inbox - mesheswarage@g
Email-Classification-Proble
Your Repositories
Google Cloud Gmail API
console.cloud.google.com/auth/audience?project=gmailclassifier-469618
Google Cloud
GmailClassifier
Search (/) for resources, docs, products, and more
Search
Google Auth Platform / Audience
Overview
Branding
Audience
Clients
Data Access
Verification Center
Add users
Filter Enter property name or value
Show less

Audience

Test users

User information
1998csj@gmail.com
as95309@sci.sjp.ac.lk
medhavi.anka@gmail.com
mesheswarage@gmail.com
meshesytit@gmail.com
pbimsara.pb@gmail.com
randiwathsala8@gmail.com
s92084130@ousl.lk
sharikaeswarage@gmail.com

7.2. Pre-processing Function

(Githup_Repository/ Deployment_p / inference_util.py)

Combined the steps of the pre-processing part and put all steps into a function so that we can call it whenever we receive a new email and before fetch that into the model.

7.3. Classification

(Githup_Repository/ Deployment_p / app.py – lines 140 - 158)

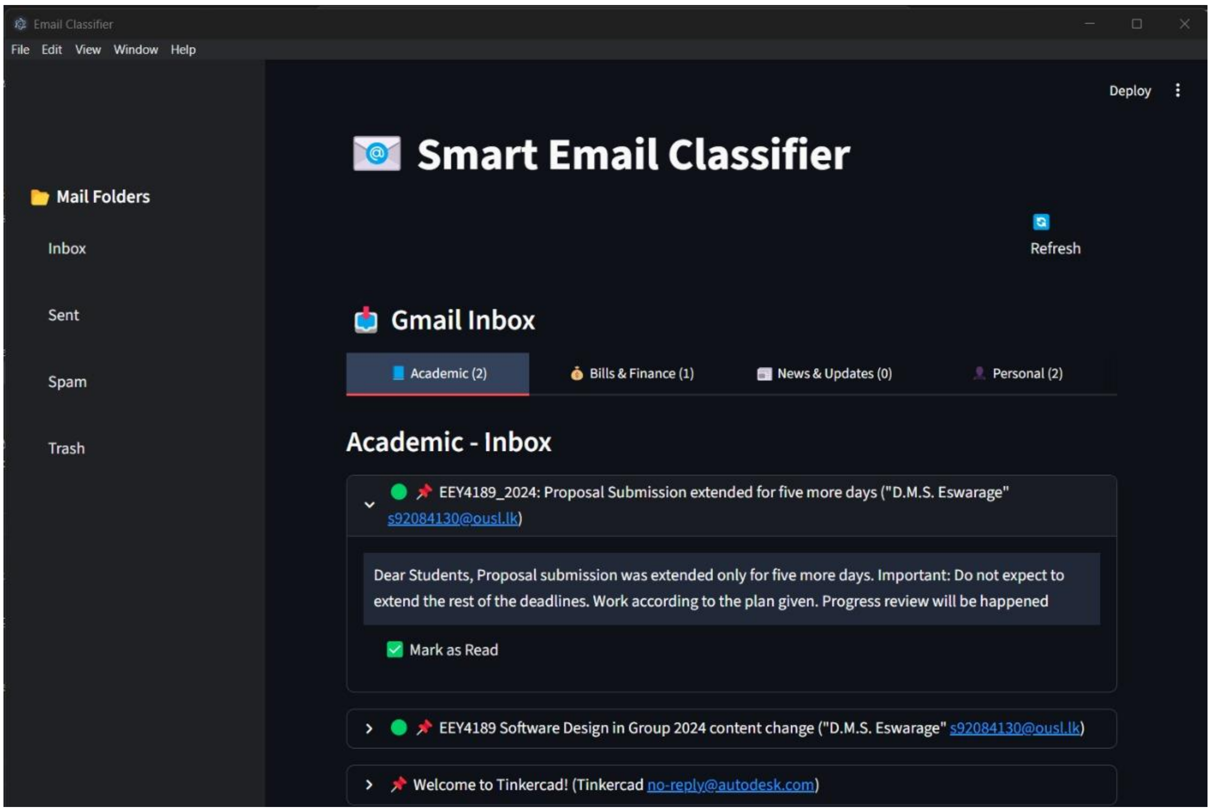
The pre-trained machine learning model, loaded using joblib, is used to classify emails into five categories: **Academic**, **Bills & Finance**, **News & Updates**, **Personal**, and **Official**.

7.4. Creating the interface (Web app)

(Githup_Repository/ Deployment_p / app.py)

The app was built using **Streamlit** to provide a responsive and interactive user interface. The user interface was designed to prioritize usability. A styled sidebar is used to navigate between folders and display unread email counts. Emails are automatically grouped into tabs based on their predicted category. Unread messages are highlighted, and users are given the option to expand email content and mark messages as read. When selected, the "Mark as Read" button triggers a call to the Gmail API to update the email's status.

A Refresh button is placed in the top-right corner of the interface to allow users to manually reload and update their email data. When this button is clicked, the st.rerun() function is triggered, which causes the entire Streamlit script to be re-executed from the top. This ensures that any newly received emails or changes—such as marking an email as read—are reflected immediately in the interface. The refresh mechanism is essential in this app because Gmail data is fetched in real time through the Gmail API, and state changes (like label updates) are only shown after a rerun.



Smart Email Classifier

Gmail Inbox

Academic - Inbox

EEY4189_2024: Proposal Submission extended for five more days ("D.M.S. Eswarage" s92084130@ousl.lk)

Dear Students, Proposal submission was extended only for five more days. Important: Do not expect to extend the rest of the deadlines. Work according to the plan given. Progress review will be happened

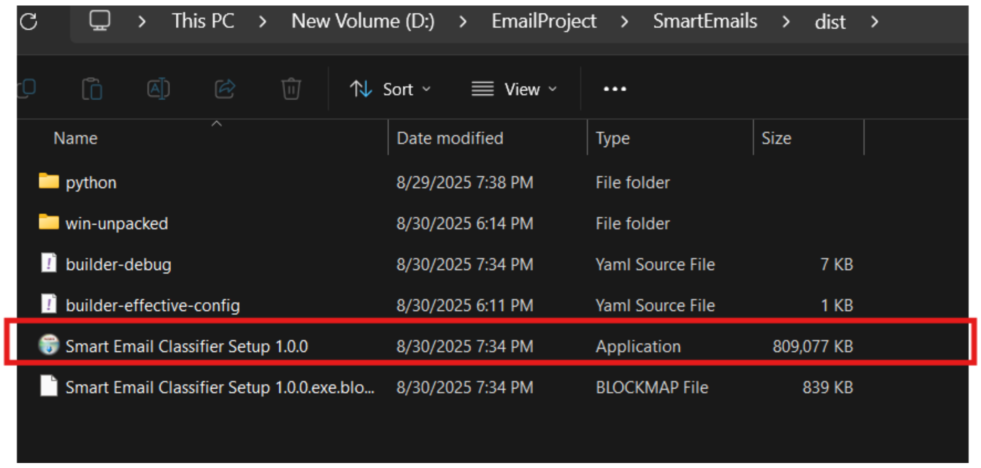
EEY4189 Software Design in Group 2024 content change ("D.M.S. Eswarage" s92084130@ousl.lk)

Welcome to Tinkercad! (Tinkercad no-reply@autodesk.com)

7.5. Packaging the App

(Githup_Repository/ Deployment_p / main.js & package.json)

Electron was used to wrap the web-based Streamlit app into a native desktop environment, while Python was embedded directly within the application bundle. Electron handles the frontend by launching a Chromium-based window, while the backend logic—such as email fetching, classification using a machine learning model, and Streamlit rendering—is handled by Python. In the Electron main.js script, Python is started as a child process using Node.js's spawn() method, which runs the Streamlit app on a specified port (localhost:8501). After a delay to allow the server to start, Electron loads this URL in the desktop window. The embedded Python executable (python.exe) ensures that users don't need to install Python separately. When the app is closed, the Python process is cleanly terminated to prevent it from running in the background.


This PC > New Volume (D:) > EmailProject > SmartEmails > dist >
Sort
View

Name	Date modified	Type	Size
python	8/29/2025 7:38 PM	File folder	
win-unpacked	8/30/2025 6:14 PM	File folder	
builder-debug	8/30/2025 7:34 PM	Yaml Source File	7 KB
builder-effective-config	8/30/2025 6:11 PM	Yaml Source File	1 KB
Smart Email Classifier Setup 1.0.0	8/30/2025 7:34 PM	Application	809,077 KB
Smart Email Classifier Setup 1.0.0.exe.blo...	8/30/2025 7:34 PM	BLOCKMAP File	839 KB

8. Results & Discussion

The **Linear SVM** model consistently outperformed Naive Bayes, Logistic Regression, and Random Forest in terms of accuracy and F1-score. The confusion matrix revealed that the most frequent misclassifications occurred between News & Updates and Personal, likely due to overlapping vocabulary (e.g., greetings, thank-you notes).

The final trained pipeline was exported as `email_sorting_svm.pkl`, embedding preprocessing and classification steps. This artifact enables seamless deployment on unseen Gmail messages.

9. Limitations

- TF-IDF + SVM captures lexical signals but may miss context-based cues (e.g., sarcasm, nuanced content).
- Sender/domain features can overfit to frequent senders — handle domain generalization.
- Model drifts if email distributions change over time; periodic retraining is necessary.
- Privacy and token security must be managed correctly.

10. Conclusion

This project successfully delivered a scalable email classification system that organizes Gmail messages into five practical categories: Academic, Official, News & Updates, Bills & Finance, and Personal.

By integrating Gmail API extraction, advanced cleaning (`clean.py`, `deepclean.py`, `preprocessing.py`), exploratory data analysis (`email_eda.py`), and a TF-IDF + SVM model, the pipeline achieved strong predictive performance.

All code is available in the GitHub repository (Email-Classification-Problem), and a working demonstration is showcased in the project video.

Future improvements include incorporating transformer-based embeddings (BERT), domain generalization, and active learning loops for incremental updates.