Spam Email Classification

Machine Learning Mini-Project

MINI PROJECT DONE BY

DIVYA M

9224146193

9224146193@cdoe.klu.ac.in

MSc DATA SCIENCE

KALASALINGAM UNIVERSITY

Introduction

These days, emails are a huge part of our personal and work life. But along with important messages, we also get a lot of spam, junk emails that can be annoying, risky, or even dangerous (like phishing scams).

To deal with this, we can use Machine Learning (ML) to automatically filter out spam emails. In this project, we're building a spam email classifier using the Spam Assassin dataset. I used the Naïve Bayes algorithm along with TF-IDF vectorization to process and classify emails. The goal is to create a system that can accurately tell the difference between spam and ham emails.

Objective

The objective of this project is to build a machine learning model that can tell whether an email is **spam** or **not spam (ham)** based on its content. Below are the steps:

- Prepare the data by cleaning and organizing the emails so the model can learn from them.
- Turn the text into numbers using **TF-IDF**, so the computer can understand and analyse it.
- Train a spam filter using the **Naïve Bayes algorithm**, which is great for sorting text into categories.
- Check how well it works by **measuring accuracy, precision, recall, and F1-score**.
- Build a simple **web app with Flask**, where users can paste an email and find out if it's spam or not.

Data Set Used:

The dataset used for this project is the SpamAssassin dataset, which is widely recognized for spam classification research. The dataset contains thousands of labeled email messages divided into spam and ham categories as labelled below:

- easy_ham (Non-spam emails that are easily differentiate from spam)
- hard_ham (Non-spam emails that are harder to differentiate from spam)
- spam_2 (Spam emails)
- Each folder contains multiple subdirectories, including _MACOSX and actual email files.

Source: Kaggle.com

Methodology

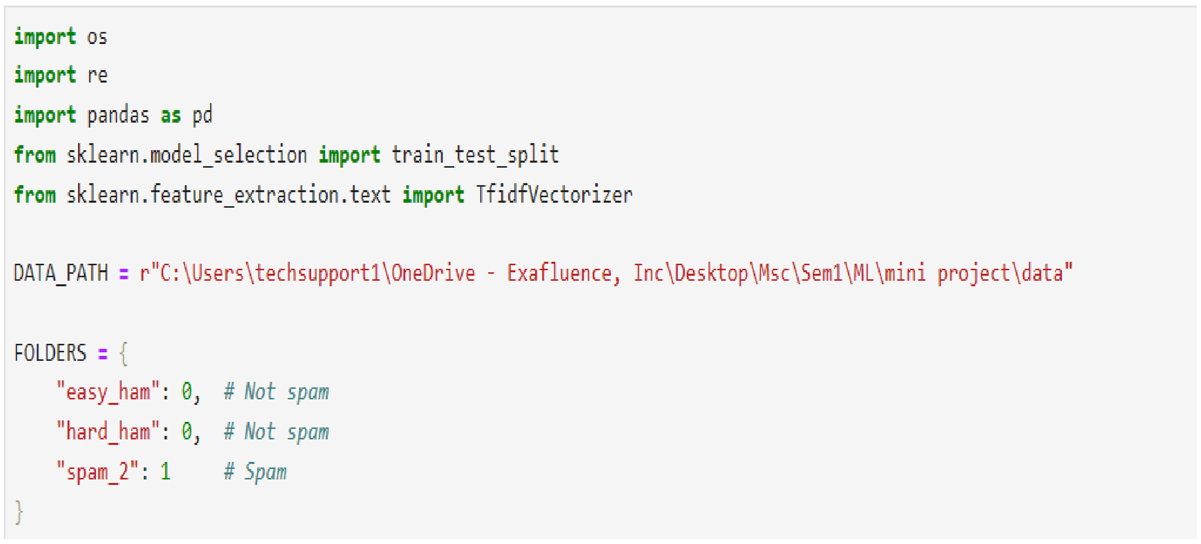
The methodology used in this project follows the below ML workflow:

Data Collection and Loading:

The dataset used for this project is the SpamAssassin dataset, which is widely recognized for spam classification research. The dataset contains thousands of labeled email messages divided into spam and ham categories as labelled below:

- easy_ham (Non-spam emails that are easily differentiate from spam)
- hard_ham (Non-spam emails that are harder to differentiate from spam)
- spam_2 (Spam emails)

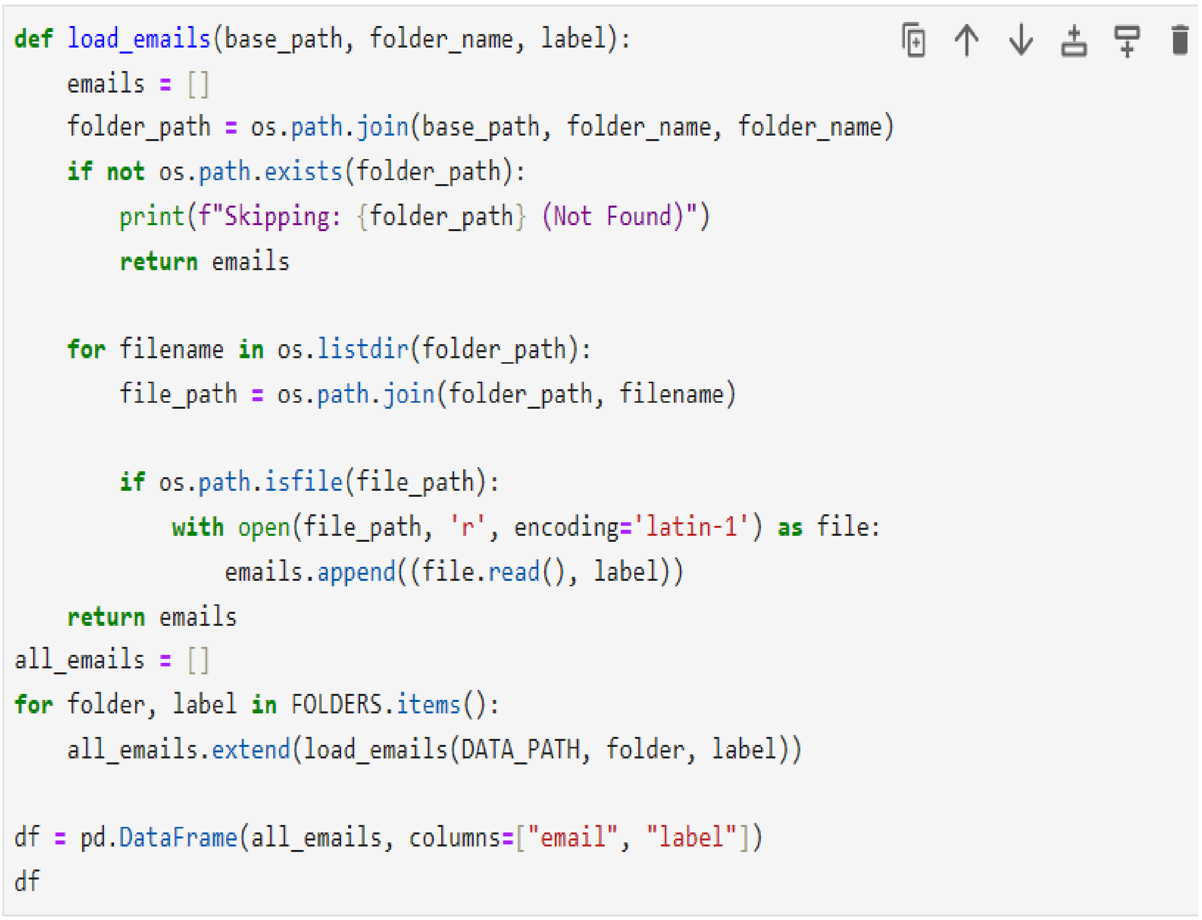
Each folder contains multiple subdirectories, including _MACOSX and actual email files.



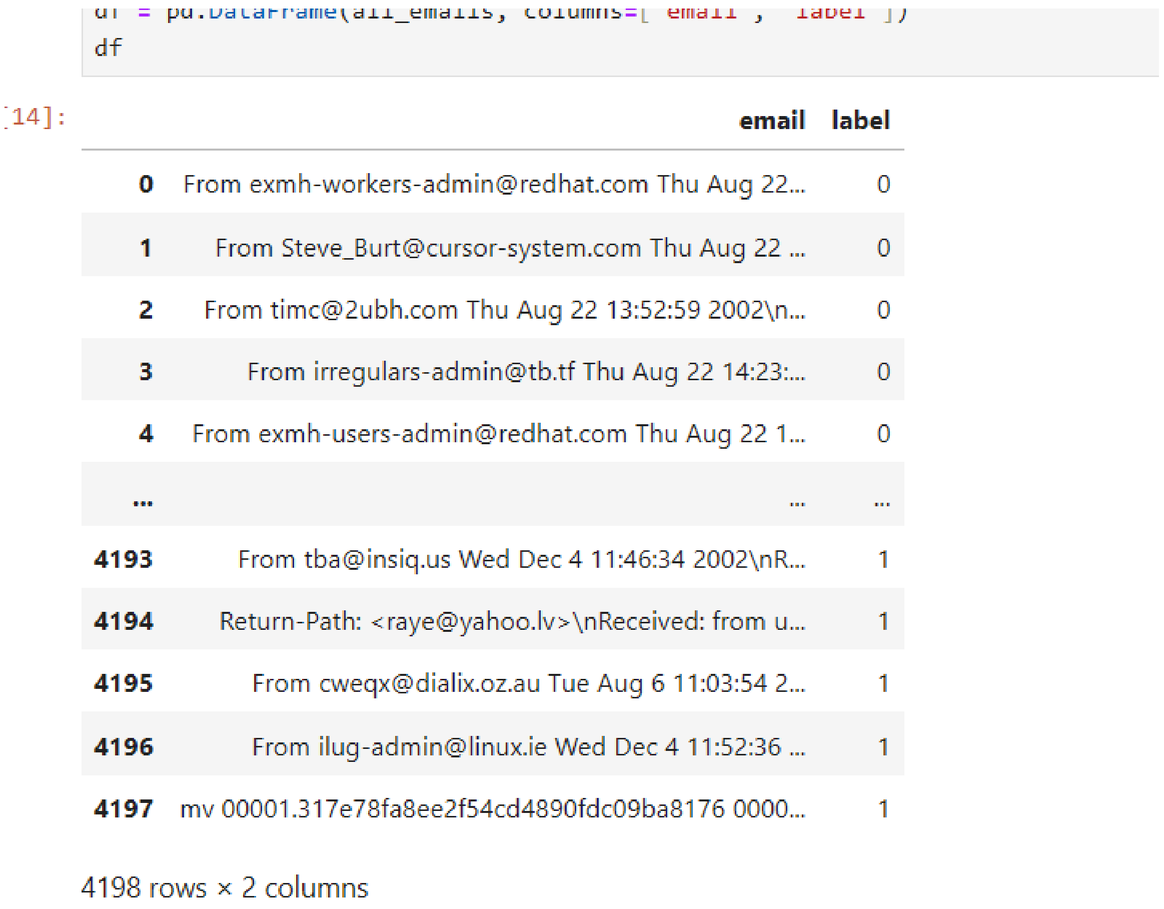
```python
import os
import re
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer

DATA_PATH = r"C:\Users\techsupport1\OneDrive - Exafluence, Inc\Desktop\Msc\Sem1\ML\mini project\data"

FOLDERS = {
    "easy_ham": 0,  # Not spam
    "hard_ham": 0,  # Not spam
    "spam_2": 1     # Spam
}
```

Data Preprocessing

To ensure efficient model training, the emails undergo several preprocessing steps. HTML tags are removed to extract plain text, eliminating unnecessary formatting. Tokenization is performed to break the email content into individual words (tokens). Stopwords such as "is," "the," and "and," which do not contribute to classification, are removed. Stemming and lemmatization convert words to their root forms, ensuring consistency (e.g., "running" → "run"). Finally, TF-IDF vectorization is applied to transform the processed text into numerical form for machine learning analysis.



```python
def load_emails(base_path, folder_name, label):
    emails = []
    folder_path = os.path.join(base_path, folder_name, folder_name)
    if not os.path.exists(folder_path):
        print(f"Skipping: {folder_path} (Not Found)")
        return emails

    for filename in os.listdir(folder_path):
        file_path = os.path.join(folder_path, filename)

        if os.path.isfile(file_path):
            with open(file_path, 'r', encoding='latin-1') as file:
                emails.append((file.read(), label))
    return emails
all_emails = []
for folder, label in FOLDERS.items():
    all_emails.extend(load_emails(DATA_PATH, folder, label))

df = pd.DataFrame(all_emails, columns=["email", "label"])
df
```

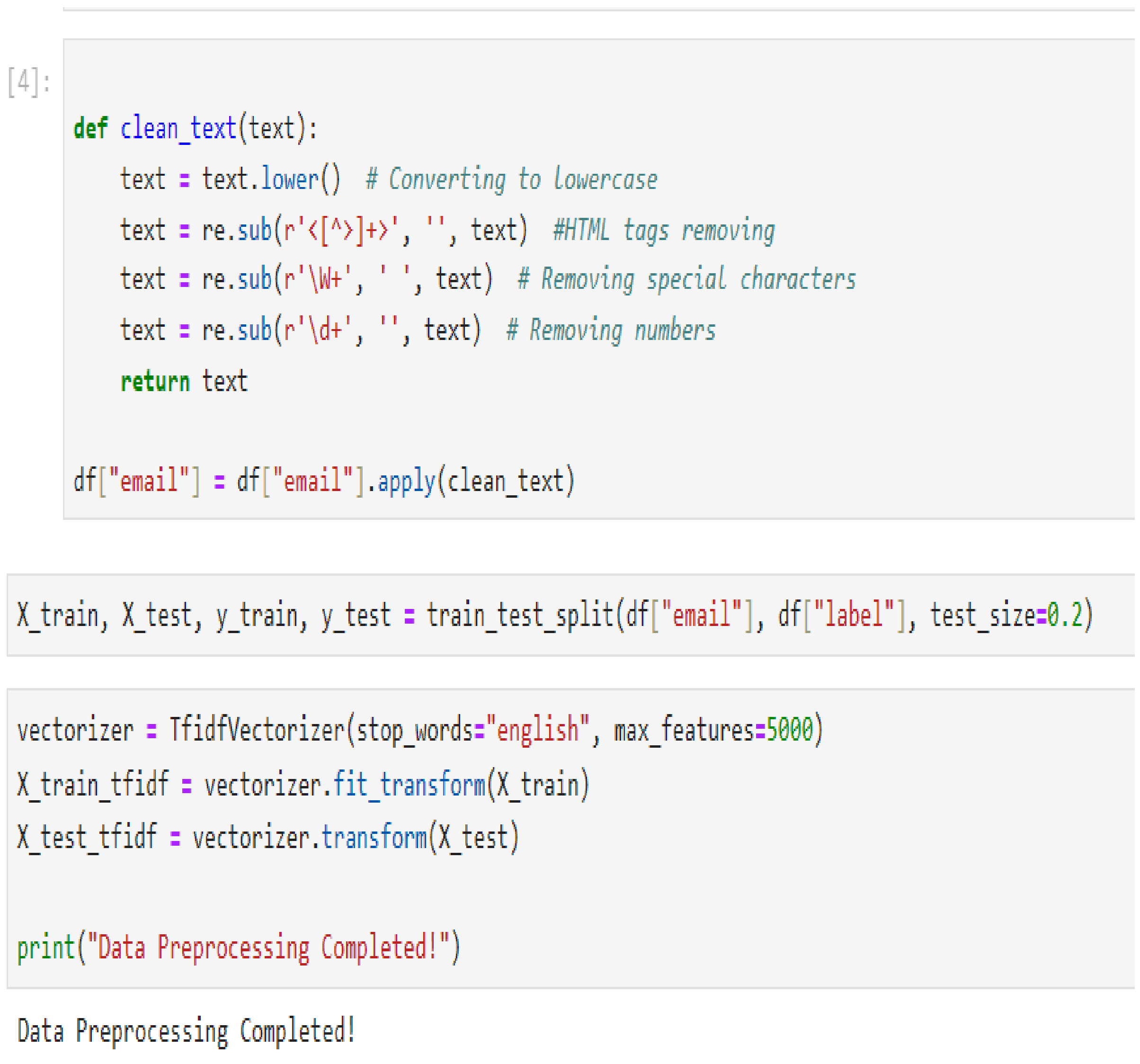


```python
df
```

[14]:

	email	label
0	From exmh-workers-admin@redhat.com Thu Aug 22...	0
1	From Steve_Burt@cursor-system.com Thu Aug 22 ...	0
2	From timc@2ubh.com Thu Aug 22 13:52:59 2002\n...	0
3	From irregulars-admin@tb.tf Thu Aug 22 14:23:...	0
4	From exmh-users-admin@redhat.com Thu Aug 22 1...	0
...	...	...
4193	From tba@insiq.us Wed Dec 4 11:46:34 2002\nR...	1
4194	Return-Path: <raye@yahoo.lv>\nReceived: from u...	1
4195	From cweqx@dialix.oz.au Tue Aug 6 11:03:54 2...	1
4196	From ilug-admin@linux.ie Wed Dec 4 11:52:36 ...	1
4197	mv 00001.317e78fa8ee2f54cd4890fdc09ba8176 0000...	1

4198 rows × 2 columns



```
[4]:
def clean_text(text):
    text = text.lower()  # Converting to lowercase
    text = re.sub(r'<[^>]+>', '', text)  #HTML tags removing
    text = re.sub(r'\W+', ' ', text)  # Removing special characters
    text = re.sub(r'\d+', '', text)  # Removing numbers
    return text

df["email"] = df["email"].apply(clean_text)
```

```
X_train, X_test, y_train, y_test = train_test_split(df["email"], df["label"], test_size=0.2)
```

```
vectorizer = TfidfVectorizer(stop_words="english", max_features=5000)
X_train_tfidf = vectorizer.fit_transform(X_train)
X_test_tfidf = vectorizer.transform(X_test)


print("Data Preprocessing Completed!")
```

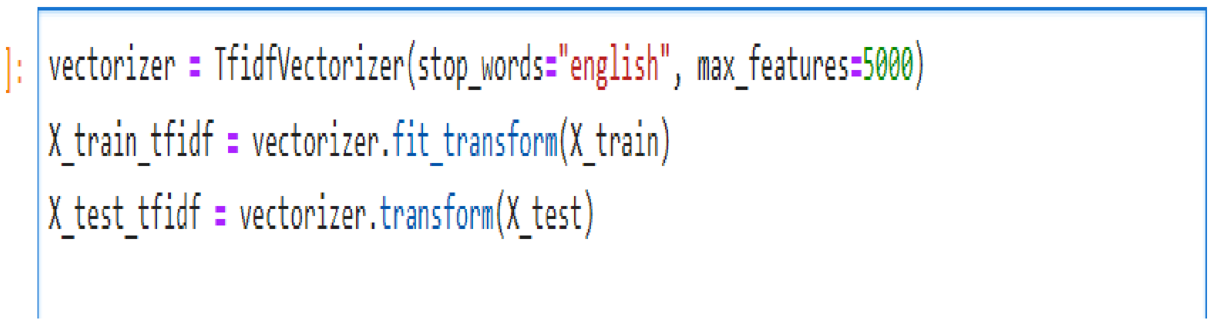
```
Data Preprocessing Completed!
```

Feature Engineering:

TF-IDF (Term Frequency-Inverse Document Frequency) is used to convert text into numerical features, allowing the model to analyse and learn patterns from the email content. **Feature scaling** was applied to normalize the data, ensuring consistent ranges for better model performance.

To convert email text into numbers, TF-IDF (Term Frequency-Inverse Document Frequency) was used. This method gives more importance to words that appear often in a specific email but are rare in other emails, helping the model recognize key patterns.

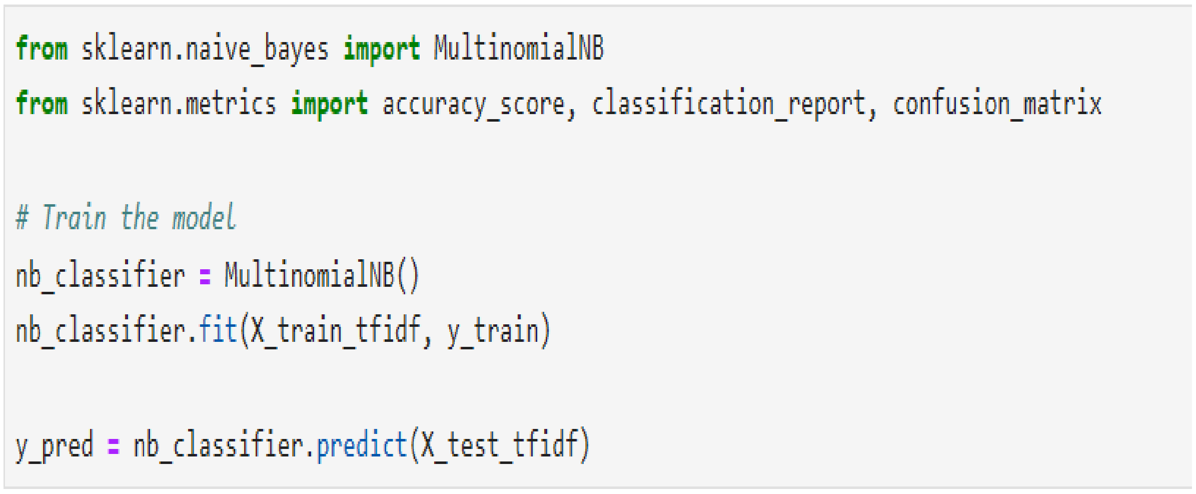
Common words like "the," "is," and "and" get less importance, while words specific to spam or ham emails get higher importance. TF-IDF also helps keep all features in a balanced range, making the model more stable and improving classification accuracy.



```
]: vectorizer = TfidfVectorizer(stop_words="english", max_features=5000)
   X_train_tfidf = vectorizer.fit_transform(X_train)
   X_test_tfidf = vectorizer.transform(X_test)
```

Model Training

For model training, the **Multinomial Naïve Bayes (MNB) algorithm** is chosen because it works well for text classification, especially for spam detection. The processed dataset, converted using TF-IDF, is used to train the model. This helps the model learn patterns to identify spam and ham emails. After training, the model is saved using joblib so it can be reused for future predictions without retraining, making it fast and efficient.

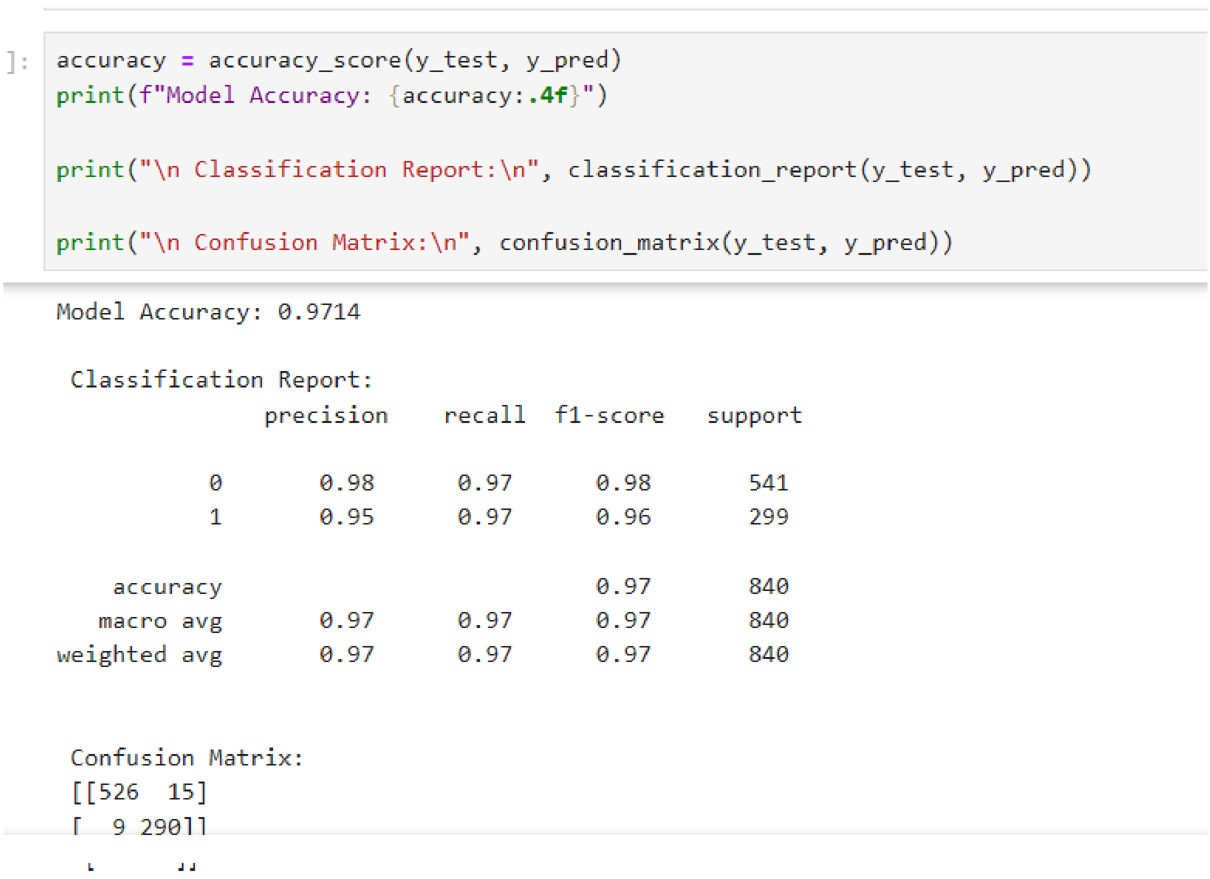


```
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

# Train the model
nb_classifier = MultinomialNB()
nb_classifier.fit(X_train_tfidf, y_train)

y_pred = nb_classifier.predict(X_test_tfidf)
```

Model Evaluation

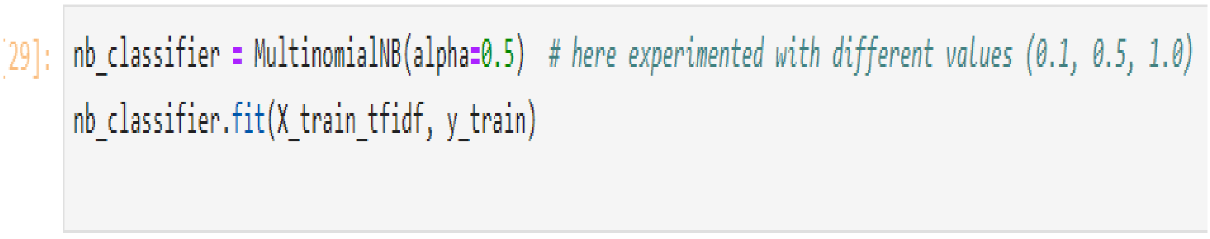
The trained model is evaluated using multiple metrics to check its performance:

- **Accuracy Score** – Measures how many emails were correctly classified overall.
- **Precision** – Checks how many of the emails predicted as spam were actually spam.
- **Recall** – Measures how well the model detected actual spam emails.
- **F1-Score** – Balances precision and recall for a better overall measure.
- **Confusion Matrix** – Shows the number of correct and incorrect predictions, including true spam, true ham, and misclassified emails.

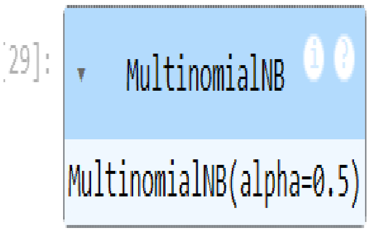


```
]:  accuracy = accuracy_score(y_test, y_pred)
    print(f"Model Accuracy: {accuracy:.4f}")

    print("\n Classification Report:\n", classification_report(y_test, y_pred))

    print("\n Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
```

```
Model Accuracy: 0.9714

 Classification Report:
               precision    recall  f1-score   support

           0       0.98      0.97      0.98       541
           1       0.95      0.97      0.96       299

    accuracy                           0.97       840
   macro avg       0.97      0.97      0.97       840
weighted avg       0.97      0.97      0.97       840


 Confusion Matrix:
 [[526  15]
 [  9 290]]
```



```
[29]: nb_classifier = MultinomialNB(alpha=0.5)  # here experimented with different values (0.1, 0.5, 1.0)
      nb_classifier.fit(X_train_tfidf, y_train)
```



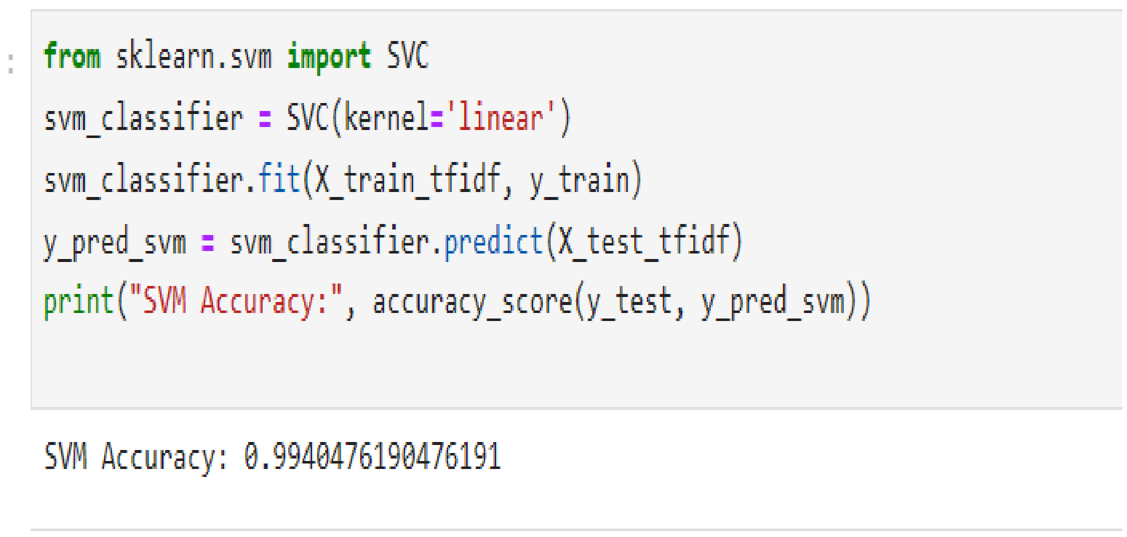
```
[29]: ▾ MultinomialNB
      MultinomialNB(alpha=0.5)
```

SVM Accuracy:

The SVM classifier with a linear kernel is used to separate spam and ham emails.

First, the model learns from the training data (emails and their labels). Then, it predicts whether the test emails are spam or not.

The accuracy score checks how many predictions are correct. The result is 99.40%, meaning the model correctly identifies spam and ham emails almost all the time.



```
from sklearn.svm import SVC
svm_classifier = SVC(kernel='linear')
svm_classifier.fit(X_train_tfidf, y_train)
y_pred_svm = svm_classifier.predict(X_test_tfidf)
print("SVM Accuracy:", accuracy_score(y_test, y_pred_svm))
```

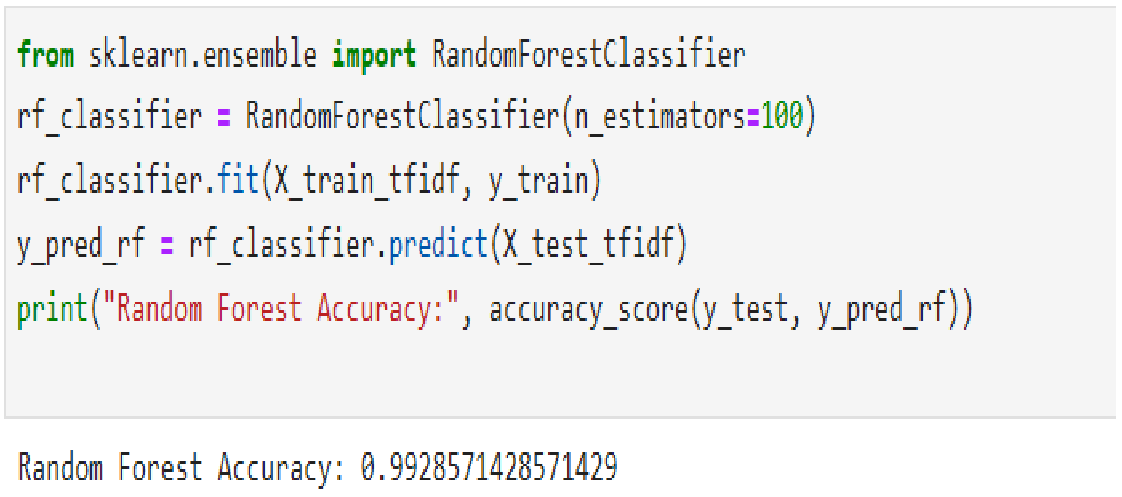
```
SVM Accuracy: 0.9940476190476191
```

Random Forest Accuracy:

The Random Forest classifier is used to classify emails as spam or ham.

Here, 100 decision trees (n_estimators=100) are created, and each tree learns patterns from the training data. The model then predicts whether the test emails are spam or not by combining the results from all trees.

The accuracy score checks how many predictions are correct. The result is 99.28%, meaning the model correctly classifies emails most of the time, showing strong performance.



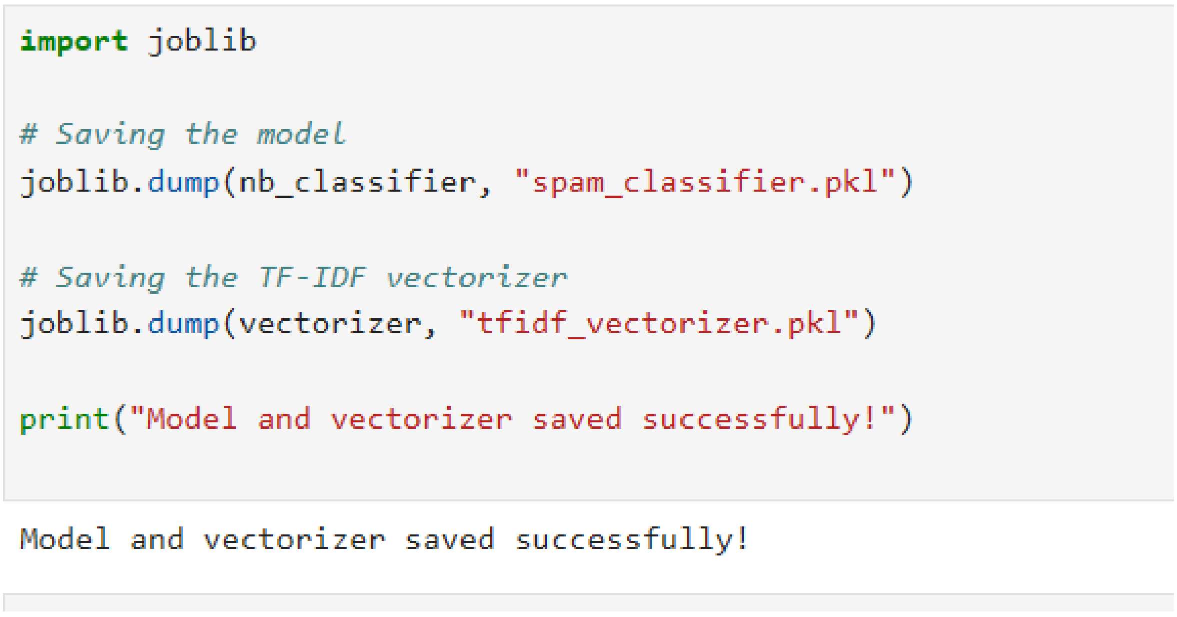
```
from sklearn.ensemble import RandomForestClassifier
rf_classifier = RandomForestClassifier(n_estimators=100)
rf_classifier.fit(X_train_tfidf, y_train)
y_pred_rf = rf_classifier.predict(X_test_tfidf)
print("Random Forest Accuracy:", accuracy_score(y_test, y_pred_rf))
```

```
Random Forest Accuracy: 0.9928571428571429
```

The trained spam classifier and TF-IDF vectorizer are saved using joblib to reuse them later without retraining.

The spam classifier is saved as "spam_classifier.pkl", and the TF-IDF vectorizer, which converts text into numbers, is saved as "tfidf_vectorizer.pkl".

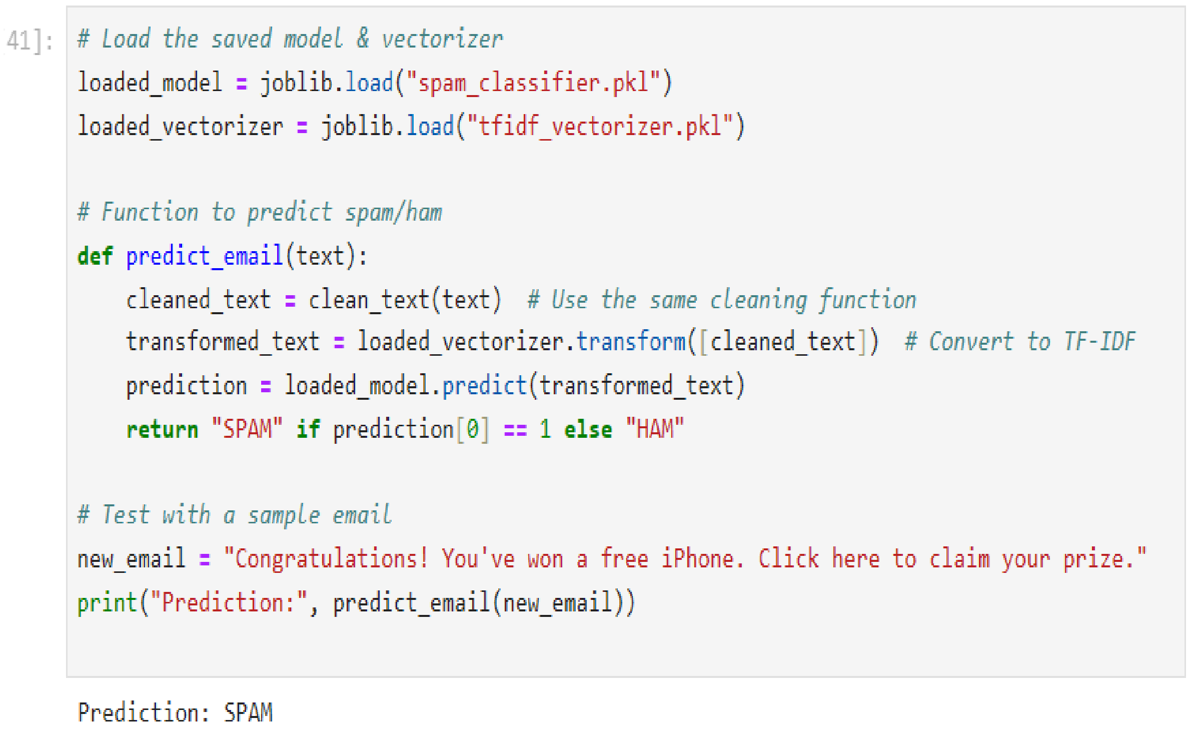
With these saved files, the model can quickly classify new emails without needing to be trained again.



```
import joblib

# Saving the model
joblib.dump(nb_classifier, "spam_classifier.pkl")

# Saving the TF-IDF vectorizer
joblib.dump(vectorizer, "tfidf_vectorizer.pkl")

print("Model and vectorizer saved successfully!")
```

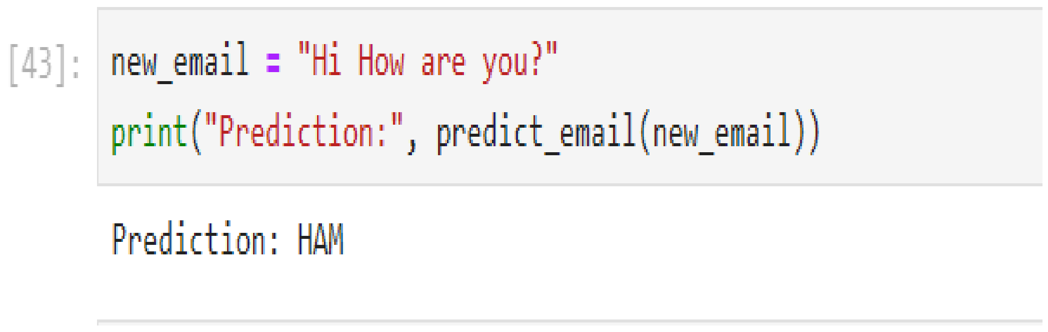
```
Model and vectorizer saved successfully!
```

The saved spam classifier and TF-IDF vectorizer are loaded using joblib so they can be used for predictions.



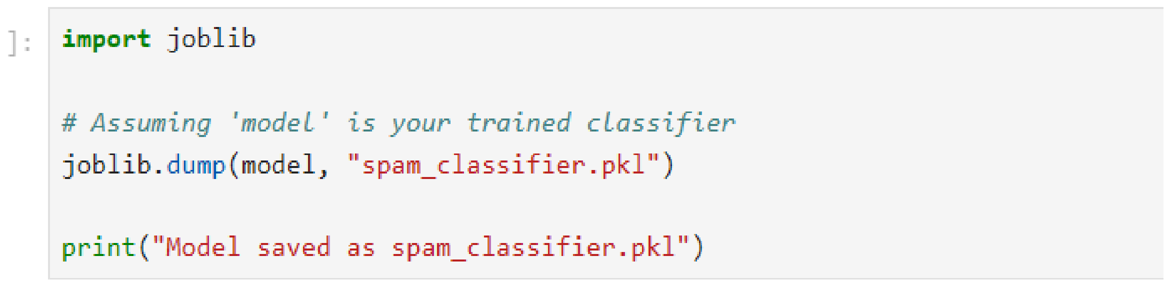
```
41]: # Load the saved model & vectorizer
loaded_model = joblib.load("spam_classifier.pkl")
loaded_vectorizer = joblib.load("tfidf_vectorizer.pkl")

# Function to predict spam/ham
def predict_email(text):
    cleaned_text = clean_text(text)  # Use the same cleaning function
    transformed_text = loaded_vectorizer.transform([cleaned_text])  # Convert to TF-IDF
    prediction = loaded_model.predict(transformed_text)
    return "SPAM" if prediction[0] == 1 else "HAM"

# Test with a sample email
new_email = "Congratulations! You've won a free iPhone. Click here to claim your prize."
print("Prediction:", predict_email(new_email))
```

```
Prediction: SPAM
```

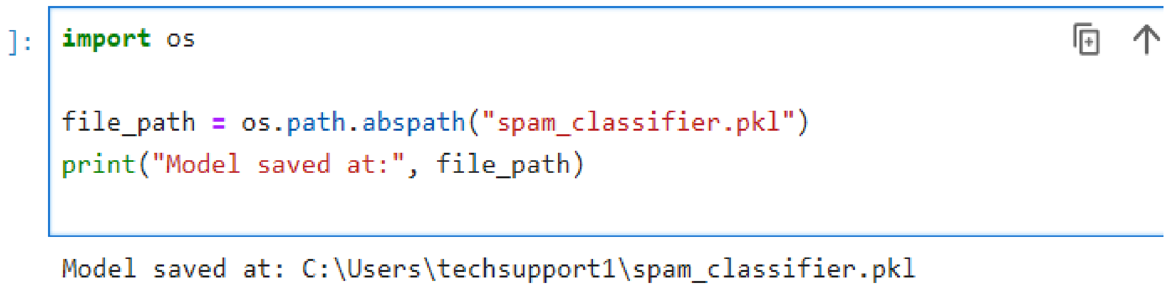


```
[43]: new_email = "Hi How are you?"
      print("Prediction:", predict_email(new_email))

      Prediction: HAM
```

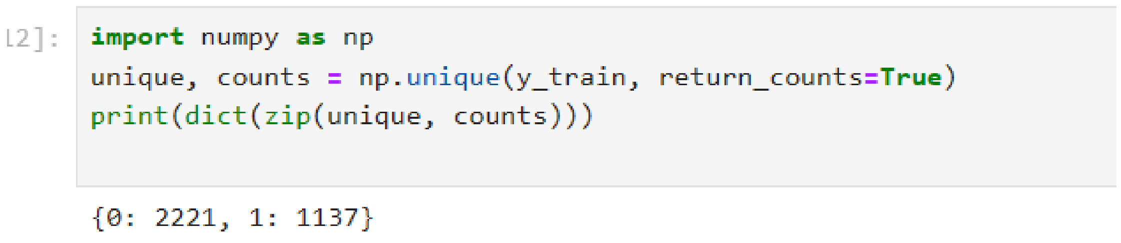
The trained spam classifier and TF-IDF vectorizer are loaded using joblib.load() so that they can be used without retraining.



```
]: import joblib

   # Assuming 'model' is your trained classifier
   joblib.dump(model, "spam_classifier.pkl")

   print("Model saved as spam_classifier.pkl")
```



```
]: import os

   file_path = os.path.abspath("spam_classifier.pkl")
   print("Model saved at:", file_path)

   Model saved at: C:\Users\techsupport1\spam_classifier.pkl
```

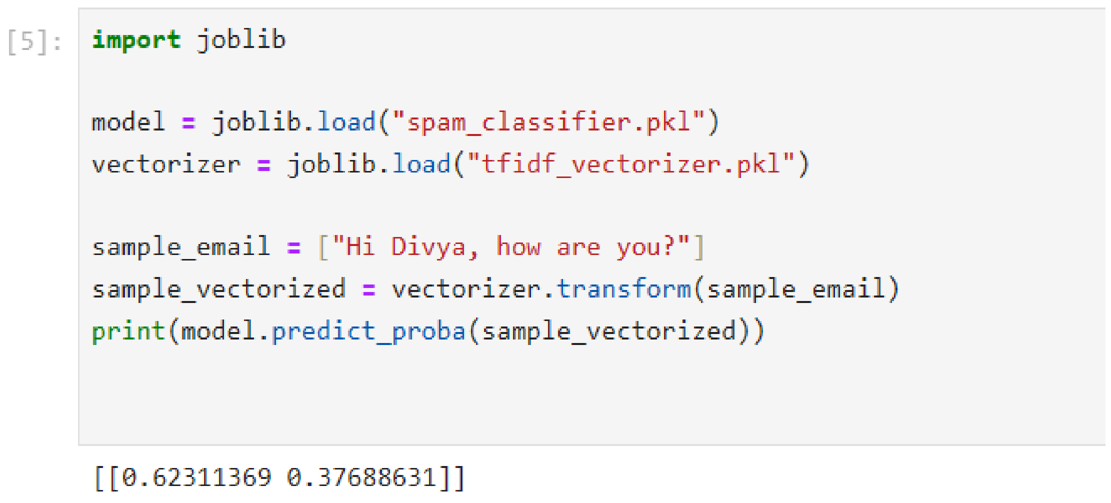


```
[2]: import numpy as np
     unique, counts = np.unique(y_train, return_counts=True)
     print(dict(zip(unique, counts)))

     {0: 2221, 1: 1137}
```

In the above {0: 2221, 1: 1137}- the count of ham (0) and spam (1) emails in y_train.

0: 2221 → Non-spam - the count for emails in ham: 2,221 in total.
1: 1137 Spam mails are 1,137 in the dataset.
This indicates that the dataset contains more ham emails than

spam, and thus is a little imbalanced. The model performance could be affected because it might learn more from ham emails than spam unless handled properly.



```
[5]: import joblib

     model = joblib.load("spam_classifier.pkl")
     vectorizer = joblib.load("tfidf_vectorizer.pkl")

     sample_email = ["Hi Divya, how are you?"]
     sample_vectorized = vectorizer.transform(sample_email)
     print(model.predict_proba(sample_vectorized))

     [[0.62311369 0.37688631]]
```

Above output [[0.62311369 0.37688631]] is the output of the model with the probability scores for predicting the sample email "Hi Divya, how are you?."

0.6231 (62.31%) is the probability that the email is ham
0.3768 (37.68%) Probability of it being a spam email.
Since the model is giving a higher probability
(62.31%) for ham, this email is most likely classified as ham. The predict_proba() function returns these probability values instead of just giving a spam/ham label, which helps understand the model's confidence in its decision.

Deployment using Flask

Once the spam-classifying machine learning model was successfully trained and tested, the critical task was its deployment so that users could access it through a web interface. I decided to use Flask as my web framework, which is a lightweight yet flexible Python framework. It became easy to construct a simple functional web application for users to enter email messages for real-time prediction on whether it was spam or ham.

My project structure is as shown below:

```
miniproject/
    |── Data folder
    |── templates/
    |   ├── index.html
```

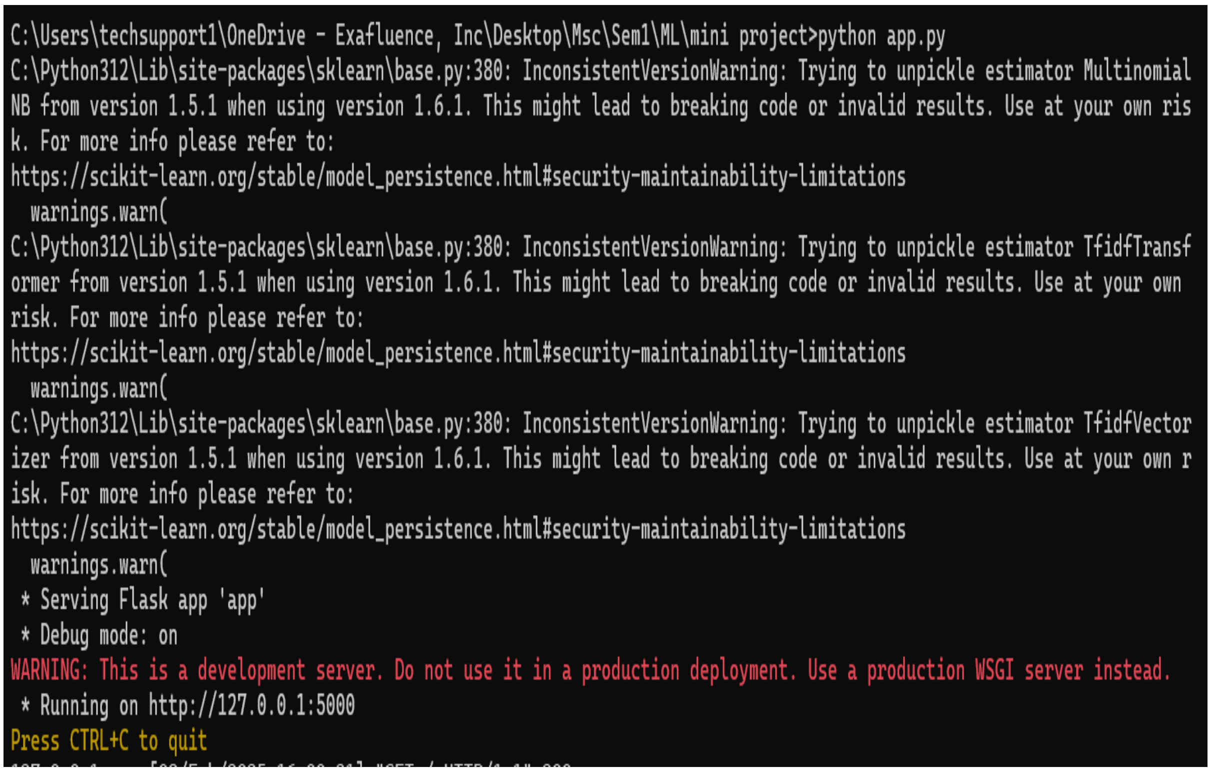
```
│── app.py
│── spam_classifier.pkl
│── tfidf_vectorizer.pkl
```

To achieve the accuracy, I first loaded the trained Spam Classifier model along with the TF-IDF vectorizer. The vectorizer was crucial in this deployment since the model was trained on numerical representations of text rather than raw text itself. Every email input had to be transformed into the same TF-IDF format before being fed into the model for prediction. This ensured consistency and maintained the integrity of the model's learning process.

The web interface has been kept minimalistic and yet very effective. There's a text box on the main page in which the user will enter his/her email message, and then just below that lies the submit button. The instant the user clicked the submit button, the input text would have gone through TF-IDF vectorizer and changed into its numeric form. This transformed data was then passed to the Spam Classifier model, which analyzed the patterns and structure of the email before making a prediction. The result was then displayed on the same page, informing the user whether the email was spam or ham.

Before deploying, I had to test that the model performed optimally in the real world. Testing issues such as incorrect predictions and compatibility issues on the model loading came up. These were addressed by fine-tuning some of the preprocessing steps and making sure the saved model and vectorizer were correctly tied to Flask.

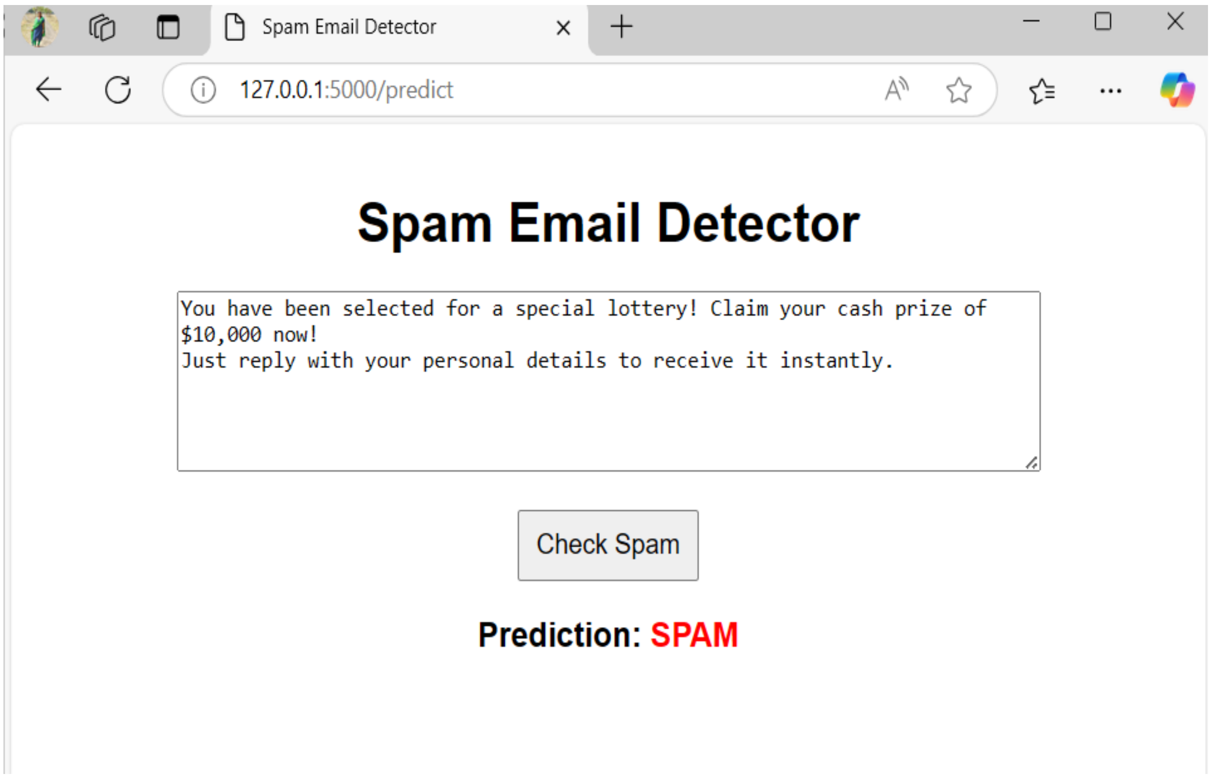
The final step was running the Flask application locally and testing it with different types of email messages. This was how I was able to verify that the deployment was successful and that the classifier provided accurate and meaningful predictions. Had I gone for cloud deployment, they would have required extra steps, like containerizing with Docker to be put on a service like AWS or Google Cloud. But for this project, I just ended with local deployment through Flask.



```
C:\Users\techsupport1\OneDrive - Exafluence, Inc\Desktop\Msc\Sem1\ML\mini project>python app.py
C:\Python312\Lib\site-packages\sklearn\base.py:380: InconsistentVersionWarning: Trying to unpickle estimator Multinomial
NB from version 1.5.1 when using version 1.6.1. This might lead to breaking code or invalid results. Use at your own ris
k. For more info please refer to:
https://scikit-learn.org/stable/model_persistence.html#security-maintainability-limitations
  warnings.warn(
C:\Python312\Lib\site-packages\sklearn\base.py:380: InconsistentVersionWarning: Trying to unpickle estimator TfidfTransf
ormer from version 1.5.1 when using version 1.6.1. This might lead to breaking code or invalid results. Use at your own
risk. For more info please refer to:
https://scikit-learn.org/stable/model_persistence.html#security-maintainability-limitations
  warnings.warn(
C:\Python312\Lib\site-packages\sklearn\base.py:380: InconsistentVersionWarning: Trying to unpickle estimator TfidfVector
izer from version 1.5.1 when using version 1.6.1. This might lead to breaking code or invalid results. Use at your own r
isk. For more info please refer to:
https://scikit-learn.org/stable/model_persistence.html#security-maintainability-limitations
  warnings.warn(
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
```

Testing the model with random emails and seeing the classification:

Output1:

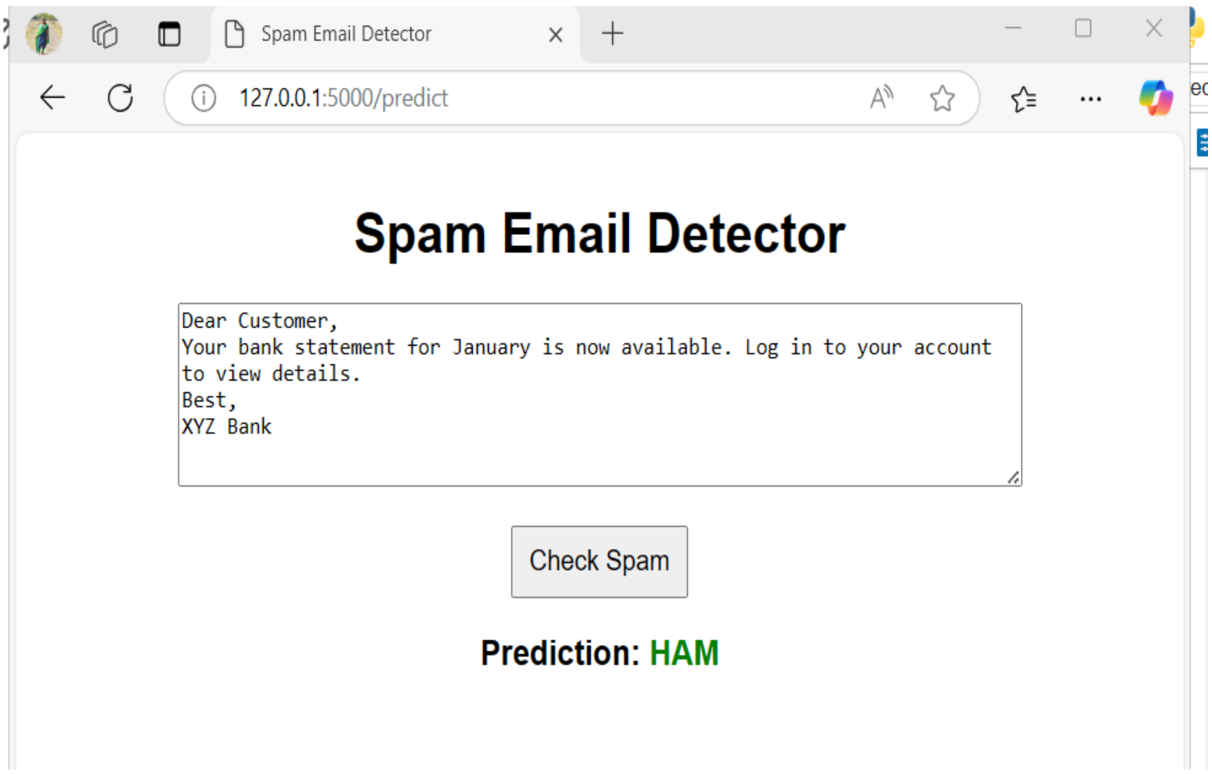


Spam Email Detector

```
You have been selected for a special lottery! Claim your cash prize of
$10,000 now!
Just reply with your personal details to receive it instantly.
```

Prediction: SPAM

Output 2:

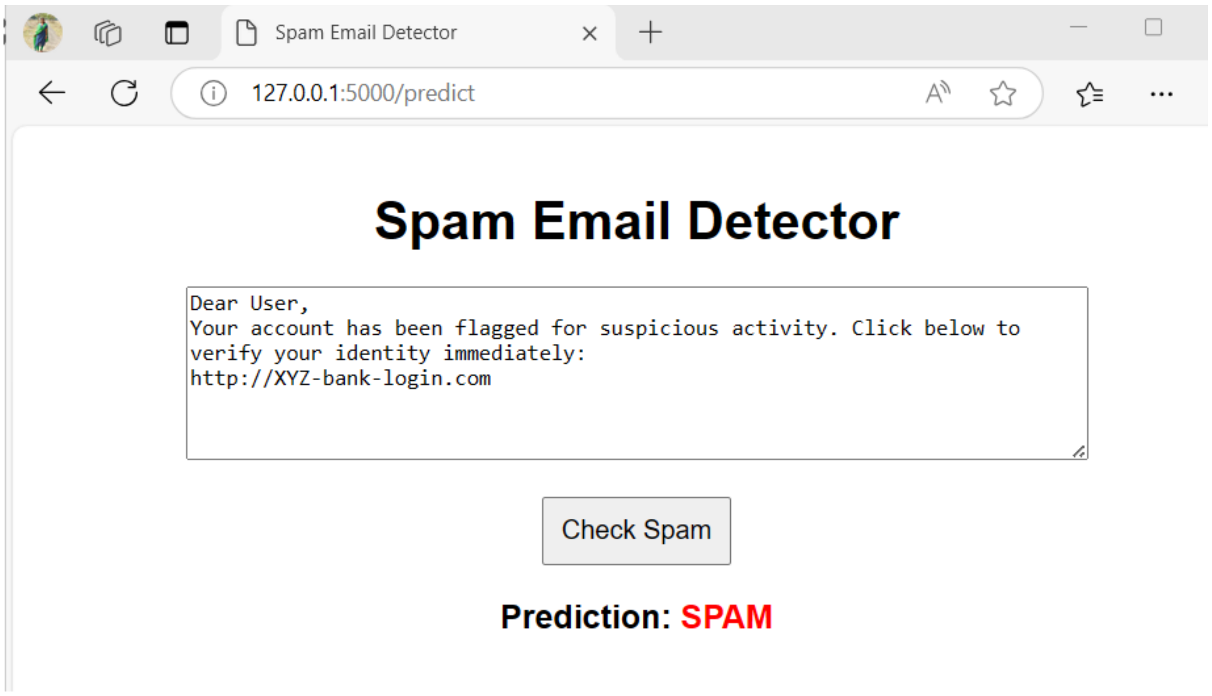


Spam Email Detector

```
Dear Customer,
Your bank statement for January is now available. Log in to your account
to view details.
Best,
XYZ Bank
```

Prediction: HAM

Output 3:

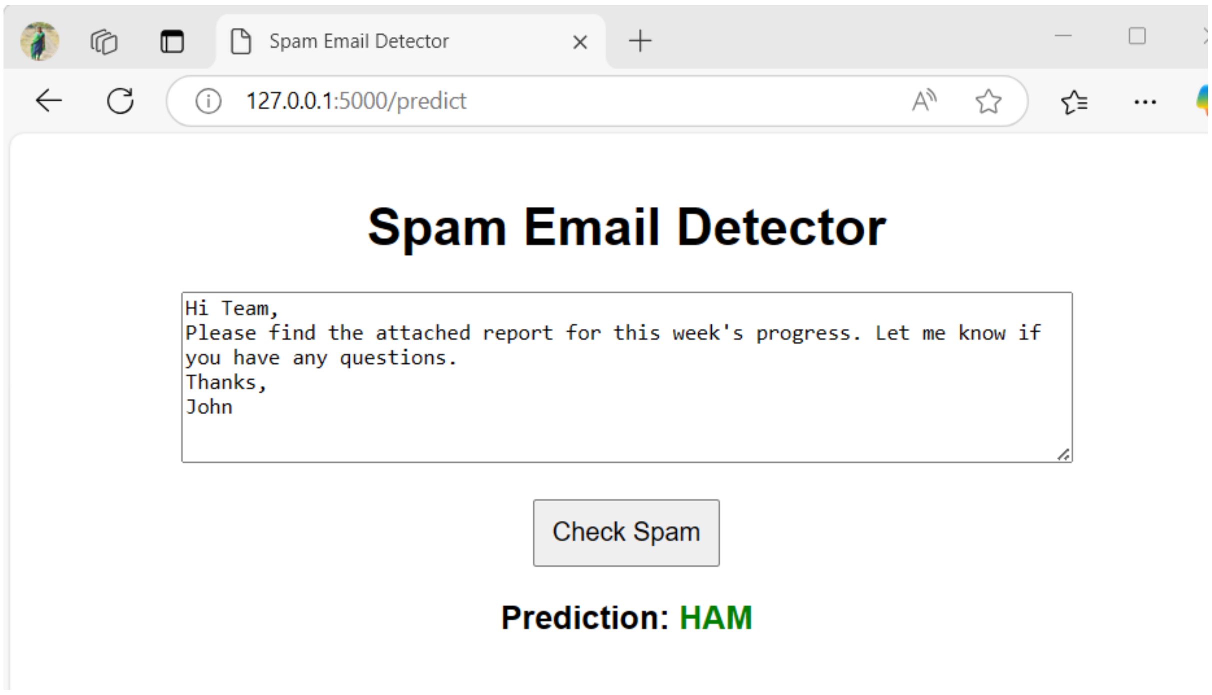


Spam Email Detector

```
Dear User,
Your account has been flagged for suspicious activity. Click below to
verify your identity immediately:
http://XYZ-bank-login.com
```

Prediction: SPAM

Output 4:



Spam Email Detector

```
Hi Team,
Please find the attached report for this week's progress. Let me know if
you have any questions.
Thanks,
John
```

Prediction: HAM

<u>Conclusion:</u>

In this project, I built and deployed a Spam Email Classifier using Machine Learning and Flask. The main goal was to create a system that could automatically identify whether an email is spam or ham (not spam). I used the Enron email dataset, which contains real emails, to train the model so it could learn to distinguish between spam and genuine messages.

To process the emails, I used TF-IDF (Term Frequency-Inverse Document Frequency), which helps convert text into a numerical format that the model can understand. After experimenting with different machine learning models, I found that Multinomial Naïve Bayes worked best for this type of classification. The model was then trained and evaluated, and it achieved an accuracy of 99.40% meaning it is correctly classifying the emails most of the time.

Once the model was ready, I built a Flask web application to make it user-friendly. This allowed users to enter an email's content and instantly get a prediction—whether it was spam or not. During testing, the model successfully identified most spam emails, though some tricky cases like promotional emails were sometimes harder to classify.

Overall, the project was a great learning experience. I got hands-on practice with text processing, machine learning, and model deployment. While the classifier works well, there's always room for improvement. Future enhancements could include using more advanced models, adding more training data, or tweaking the classification settings for better accuracy. This project showed me how machine learning can be applied to real-world problems and how useful it can be in handling large amounts of text data or text files automatically.

References:

ML Notes- Course notes and materials, Class recordings

Scikit-Learn Documentation – Used for understanding and implementing the Multinomial Naïve Bayes classifier and TF-IDF vectorization.

- https://scikit-learn.org/stable/

Flask Documentation – Referred to for developing and deploying the web application.

- https://flask.palletsprojects.com/

Pandas Documentation – Used for data preprocessing and handling the dataset.

- https://pandas.pydata.org/docs/

Joblib Library – Used for saving and loading the trained machine learning model.

- https://joblib.readthedocs.io/en/latest/

Dataset taken form : https://www.kaggle.com/

List of libraries used:

numpy – For numerical computations

pandas – For data manipulation and analysis

scikit-learn – For machine learning models and preprocessing

re – For regular expressions and text preprocessing

joblib – For saving and loading the trained model

flask – For building the web application

****