DEBRE BRIHAN UNIVERSITY
COLLEGE OF COMPUTING
DEPARTMENT OF SOFTWARE ENGINEERING
FUNDAMENTALS OF MACHINE LEARNING

Name Id
Samuel tale ----------1402284

Submitted to: Derbew
Submission Date: 2/05/2017 E.C

Building a Email Spam Classifier with FastAPI

In today's digital world, spam emails are more than just an annoyance—they clutter inboxes, waste time, and can even pose security risks. To tackle this issue, we've built a spam classification web app using FastAPI, a powerful and modern Python web framework, combined with pre-trained machine learning models and vectorization techniques.

At the heart of this system are machine learning models trained on well-labeled datasets of emails, enabling them to recognize spam with impressive accuracy. Once integrated into our FastAPI application, these models allow users to quickly check whether an email is spam or not. Simply enter the email text into the app's user-friendly interface, and within moments, you'll get a classification result. Thanks to FastAPI's speed and efficiency, the whole process feels seamless and responsive.

Problem Definition and Data Acquisition

The goal of spam detection is to automatically classify incoming emails as either spam (unwanted or harmful) or ham (legitimate communication). Implementing an effective spam detection system helps improve user experience, reduces phishing risks, and enhances the overall quality of email communications.

I used dataset in csv file structure, This dataset is used for training models to classify emails as "spam" or "ham" (non-spam). The effectiveness of the model depends on its ability to accurately distinguish between the two categories.

Information about the datasets

DatasetName : spam.csv
Source: Kaggle
Licence: CC BY 4.0 (Creative Commons Attribution 4.0)
Features: contains the messages and the classification whether it is spam or not spam
Structure: contains more than five thousand datasets
Link to Dataset: https://kaggle.com
Retrieval Date: 21/05/2017E.c

The dataset can be used with algorithms such as Naive Bayes, Support Vector Machines (SVM), or neural networks to automate the detection of spam emails. But i used Navie Bayes in this project.

The main features of this spam csv file is that it identifies the message as spam or ham(not spam) based on certain phrase or words in the message like 'take your money', 'do not miss out this' and etc. Because it is a supervised learning, the model trained on this labeled data which is the message is directly mapped to either spam or not spam(ham), It identifies unseen data as spam or not spam based on the trained model which is in .joblib format.

Machine learning algorithm used in this project

Since spam detection involves categorizing emails into distinct classes—spam or ham—it relies on *classification algorithms* such as Naïve Bayes, Support Vector Machines (SVM). These algorithms analyze email content and patterns to make accurate predictions, helping users filter out unwanted messages efficiently.

In this project, all necessary dependencies are listed in the ***requirements.txt*** file, making it easy to set up the environment and install required libraries.

Regression vs. Classification for Spam Detection

Classification: The Right Choice for Spam Detection

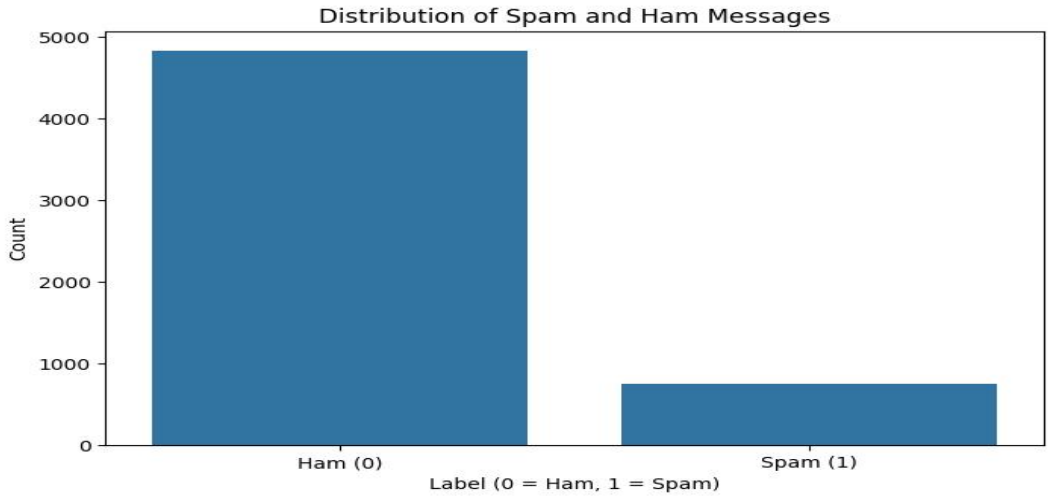
Spam detection is fundamentally a classification problem because the goal is to categorize emails into distinct classes—spam or ham (not spam). Classification models learn patterns from labeled training data and use that knowledge to assign incoming emails to one of these predefined categories.

Regression:It's Not Ideal for Spam Detection

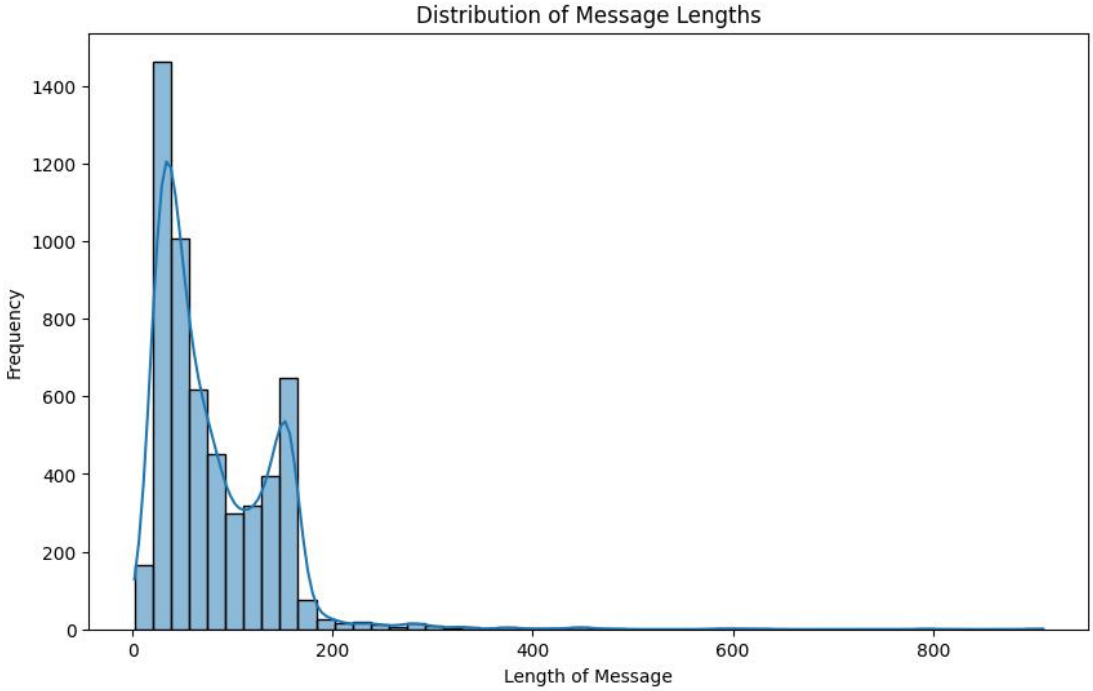
It is used when predicting a continuous numerical value rather than discrete categories. If we were trying to predict something like the spam score of an email (e.g., a value between 0 and 1), regression could be useful. However, since we are only concerned with two possible outcomes—spam or ham—classification is the better approach.

Exploratoary Data Analysis

Data understanding and exploration is found at the ***directory of notebooks folder***.



Based on the above image, out of a total of 5,525 data items,the majority are not spam messages.This indicates that a smaller subset of the data is classified as ham or not spam.Thus, the data focuses on legitimate messages.



The image displays a histogram and line plot showing the distribution of message lengths. X-axis represents the length of messages (ranging from 0 to 800), y-axis Indicates the frequency of messages of varying lengths.

The histogram bars illustrate that shorter messages (around 0 to 200 characters) are much more common, with peak frequencies reaching about 1,400 for lengths around 50-100 characters.

Other Visualization is found on the directory of notebooks.

Hyperparameter Tuning

```
# Define the hyperparameter grid to search
param_grid = {
    'alpha': [0.1, 0.5, 1.0, 1.5, 2.0],
    'fit_prior': [True, False]
}

# Set up the GridSearchCV
grid_search = GridSearchCV(estimator=model, param_grid=param_grid,
                           scoring='accuracy', cv=5, n_jobs=-1, verbose=1)
```

```
# Fit GridSearchCV to the training data
grid_search.fit(X_train, y_train)
```

The output for the model is

Best Hyperparameters: {'alpha': 0.1, 'fit_prior': True}

The best hyperparameters for the Naïve Bayes model were found to be alpha = 0.1, meaning light smoothing was applied to handle rare words, and fit_prior = True, allowing the model to learn class probabilities from the data. This combination improved *spam classification accuracy* by balancing sensitivity to *rare words* and adjusting for class imbalances.

Train the dataset

Model and Vectorizer Loading

The application begins by loading the trained spam classification model and the vectorizer using the Joblib library. This setup allows for swift execution of predictions on new email inputs.

Load and preprocess data

df = load_data("./data/spam.csv") # Load the spam dataset
(X_train, X_test, y_train, y_test), vectorizer = preprocess_data(df) # Preprocess and split the data

Train Naive Bayes classifier

model = MultinomialNB() # Initialize the Naive Bayes model
model.fit(X_train, y_train) # Train the model on the training set

Evaluate model performance

```
y_pred = model.predict(X_test)  # Predict on the test set
print("Accuracy:", accuracy_score(y_test, y_pred))  # Print accuracy score
print(classification_report(y_test, y_pred))  # Print detailed classification metrics

model = joblib.dump("./model/spam_classifier.joblib")
vectorizer = joblib.dump("./model/vectorizer.joblib")
```

Predict using terminal

To visualize it run python src/predict.py and src/train.py

Load the joblib files

```
model = joblib.load("./model/spam_classifier.joblib")
vectorizer = joblib.load("./model/vectorizer.joblib")
```

Web Application Setup

An instance of the FastAPI application is created, while templates and static files are configured to enable a rich user interface that displays the prediction results.

```
app = FastAPI()
templates = Jinja2Templates(directory="templates")
app.mount("/static", StaticFiles(directory="static"), name="static")
```

Input Model Definition

A Pydantic model named EmailInput is defined to structure the incoming data. This model validates that the input is a string containing the email text.

```
class EmailInput(BaseModel):
 text: str
```

Routing and Logic The application implements two main routes:

The root ("/") route renders the home page with an HTML template.

```
@app.get("/", response_class=HTMLResponse)
async def read_root(request: Request):
 return templates.TemplateResponse("index.html", {"request": request})
```

The prediction route ("/predict/") processes the input email, applying preprocessing and utilizing the loaded model for classification.

```
@app.post("/predict/")
async def predict_spam(email: EmailInput):
Prediction logic
```
Prediction Logic The prediction process involves several steps:

```
def predict_email(text):
   text = vectorizer.transform([text])
   prediction = model.predict(text)[0]
   return "Spam" if prediction == 1 else "Ham"
```
Text Preprocessing
The input text is transformed to lower case. Additional preprocessing, such as removing punctuation, can be added as needed.

```
def preprocess_text(text: str):
 return text.lower()
```
Vectorization: The preprocessed email text is vectorized using the previously loaded vectorizer.

Model Prediction: The model predicts whether the email is spam or not. The output is appropriately formatted and returned in the response.

Error Handling

To maintain robustness, the application incorporates error handling using FastAPI's HTTPException. This ensures that any unexpected issues during the prediction process are logged and a user-friendly message is returned.

```
try:
      # Preprocess the input text and vectorize it
      preprocessed_text = preprocess_text(email.text)  # Preprocess the input
      email_vectorized = vectorizer.transform([preprocessed_text])  # Vectorize the
          preprocessed text

      # Predict using the model
      prediction = model.predict(email_vectorized)[0]  # Predict

      # Return the result based on the model output
      return {
         "prediction": "This message is Spam, Be aware" if prediction == 1 else "This
message is not Spam, You can read it"
      }

   except Exception as e:
      # Log the exception and return an error response
      raise HTTPException(status_code=400, detail=f"An error occurred during
prediction: {str(e)}")
```

Model selection training details

Naive Bayes

The reason i used this model is that it is straightforward to implement and efficient in terms of computation. It works well with large datasets, and It's particularly suited for text classification tasks (like spam detection) because it assumes that the presence of a feature (word) in a class is independent of the presence of any other feature.

```
model = MultinomialNB()  # Initialize the Naive Bayes model
model.fit(X_train, y_train)  # Train the model on the training set
Model Evaluation metrics
print("Accuracy:", accuracy_score(y_test, y_pred))  # Print accuracy score
print(classification_report(y_test, y_pred))  # Print detailed classification report
```

The accuracy score is calculated using the accuracy_score function by comparing y_test (true labels) with y_pred (predicted labels). It shows the overall proportion of correctly classified emails.

The classification_report function provides a comprehensive overview of the model's performance, reporting precision, recall, F1-score, and support for each class (spam and ham). This detailed report gives insights into how well the model is performing and helps identify areas for improvement.

Evaluating Model Performance

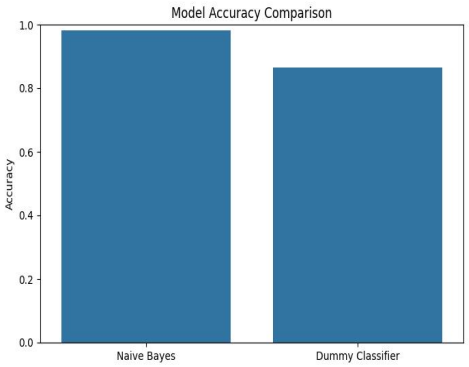
Accuracy: 0.9829596412556054(98%)

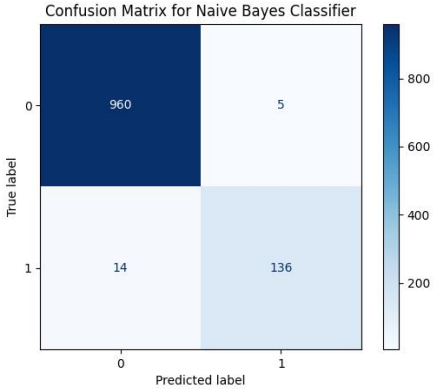
	Precision	recall	f1-score	support
0	0.99	0.99	0.99	965
1	0.96	0.91	0.93	150
accuracy			0.98	1115
macro avg	0.98	0.95	0.96	1115
weighted avg	0.98	0.98	0.98	1115

Dummy Classifier Accuracy: 0.8654708520179372

The spam classification model achieved 98% accuracy, demonstrating *strong* overall performance. With a precision of 0.96 for spam emails, the model effectively minimizes false positives (misclassifying legitimate emails as spam). However, the recall for spam is 0.91, meaning about 9% of actual spam emails might still go undetected.

The F1-score of 0.93 strikes a *good balance* between precision and recall. Compared to the Dummy Classifier's accuracy of 86.5%, which represents a naive baseline, our model shows a significant improvement. While the results are strong, further tuning or handling of class imbalance could enhance recall for spam detection.





Some Graphs for visulaizing model performance

Steps for Deployment on Render

step 1: Create a Web Application
step 2: Create requirements.txt -> This file contains all the libraries my application requires.
step 3: Version Control with Git -> creating reppository using github and git
step 4: Create a Render Account
step 5: Connect Your Repository -> Link my GitHub account and select the repository i created for my spam detection application. Render will automatically deploy the web application each time i push changes to this repository.
step 6: Configure Your Render Service -> Service Name: My service a descriptive name. Environment: Choose Python. Build Command: pip install -r requirements.txt. Start Command: uvicorn api.app:app --host 0.0.0.0 --port 5000 --reload. the use of api is that main.py is found at the folder of api.

- Go to the deployed site now you can enter the email you want and cick the check button. And instantly the prediction will appear at the bottom of the button like this

Prediction: This message is Spam, Be aware,
If the message is spam
Or
If the message is not spam
Prediction: This message is not Spam, You can read it.

Problem Limitation

Potential limitation is that is, deployed website will no open automatically because of their police which they will delay it 55 sec to open it.