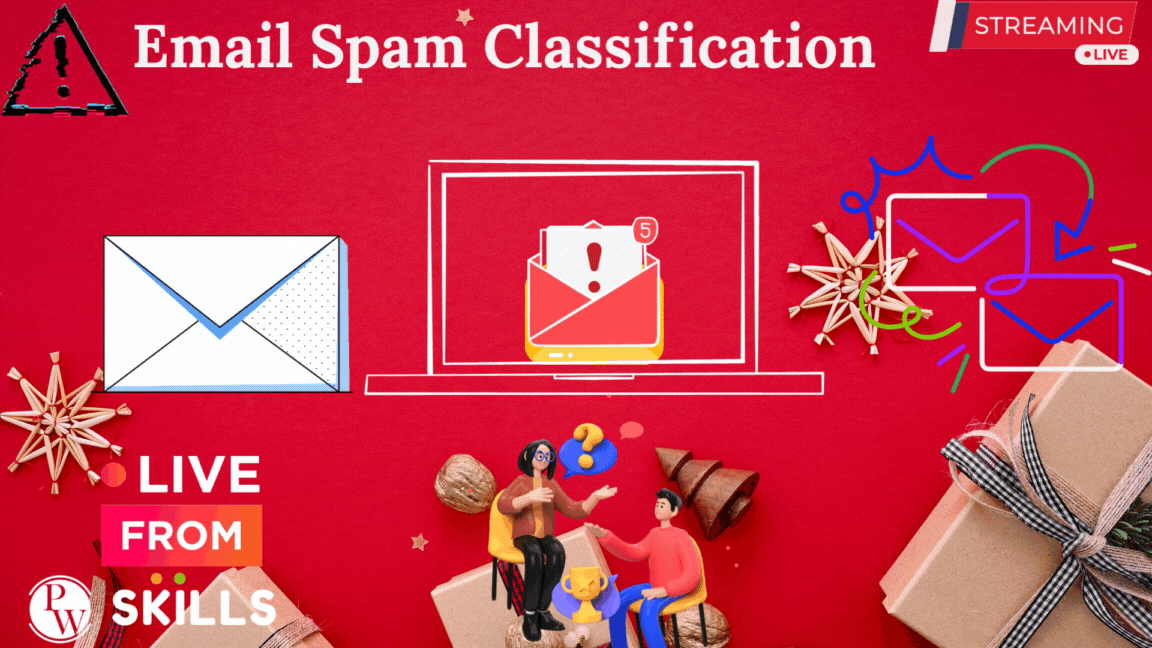
# **Email Spam Classification**

****

| **Section No** | **Title** |
| --- | --- |
| 1 | Project Objective |
| 2 | Introduction |
| 3 | Dataset Description |
| 4 | Workflow |
| 5 | Code Explanations |
| 6 | Project Setup |
| 7 | Future Enhancements |
| 8 | Conclusion |

# 

## 

## **Project Objective**

The primary objective of this project is to develop a sophisticated machine learning model capable of accurately classifying emails as either spam or legitimate (ham). This classification system will leverage Logistic Regression as its core algorithm while implementing various regularization techniques—Lasso (L1), Ridge (L2), and Elastic Net—to enhance performance and prevent overfitting. The project will thoroughly investigate the challenges of multicollinearity in feature spaces, implement effective feature selection strategies, and optimize model parameters through rigorous tuning procedures to achieve maximum classification accuracy.

## **Introduction**

In today's digital environment, email remains a critical communication channel for personal and professional interactions. However, the proliferation of unsolicited and potentially harmful spam emails presents significant challenges for users and organizations alike. These unwanted communications waste valuable time, consume network resources, and may contain malicious content that threatens data security and privacy.

Spam classification presents unique machine learning challenges due to several factors:

1. **High-dimensional feature space**: When converting email text to features (such as through bag-of-words or TF-IDF vectorization), we typically generate thousands of features, many of which may be correlated.
2. **Feature multicollinearity**: Many textual features in emails exhibit strong correlations with each other, which can destabilize model coefficients and reduce interpretability.
3. **Class imbalance**: In real-world settings, legitimate emails often significantly outnumber spam, creating potential bias in model training.
4. **Feature selection complexity**: Determining which textual features are most predictive of spam requires sophisticated statistical approaches beyond simple correlation analysis.

This project addresses these challenges by implementing regularization techniques specifically designed to manage high-dimensional data with potential multicollinearity. Lasso regularization will facilitate automatic feature selection by shrinking less important coefficients to zero, while Ridge regularization will mitigate the impact of correlated features by constraining coefficient magnitudes. Elastic Net combines these approaches to balance feature selection with coefficient stabilization.

We will utilize the SpamAssassin Public Corpus as our primary dataset, which contains a diverse collection of real-world emails with verified spam/ham classifications. The dataset will undergo comprehensive preprocessing to transform raw email content into structured feature representations suitable for machine learning.

Beyond building a functional classifier, this project aims to develop insights into the relative effectiveness of different regularization approaches for text classification tasks. By systematically comparing model performance across various configurations, we will establish best practices for implementing regularized logistic regression in similar NLP classification contexts.

The final deliverable will be a robust, well-documented spam classification system with optimized hyperparameters and comprehensive evaluation metrics that demonstrate its effectiveness in real-world email filtering scenarios.

# **Dataset Description**

## **SpamAssassin Public Corpus**

The project utilizes the SpamAssassin Public Corpus, a well-established dataset for email spam classification research. This dataset provides a comprehensive collection of real-world emails that have been manually classified as either spam or legitimate (ham) messages.

Link: We will use the [SpamAssassin Public Corpus](https://www.kaggle.com/datasets/beatoa/spamassassin-public-corpus)

### **Dataset Composition**

The corpus is organized into three primary directories:

* **spam\_2**: Contains emails that are definitively classified as spam (labeled as 1)
* **easy\_ham**: Contains legitimate emails that are clearly non-spam (labeled as 0)
* **hard\_ham**: Contains legitimate emails that share some characteristics with spam, making them more challenging to classify correctly (labeled as 0)

### **Dataset Statistics**

Based on the value counts provided, the dataset exhibits some class imbalance, with legitimate emails (ham) outnumbering spam emails. This imbalance reflects real-world email distributions where legitimate communication typically exceeds unsolicited messages.

### **Email Structure**

Each email in the corpus is stored as an individual text file in Latin-1 encoding and contains:

1. Complete email headers (sender, recipient, routing information, timestamps)
2. MIME format specifications
3. Email body content (both plain text and HTML versions when available)
4. Metadata such as precedence flags and routing information

### **Preprocessing Considerations**

The raw emails require significant preprocessing before they can be used for machine learning:

1. The emails contain full header information which may or may not be relevant for classification
2. Many emails include both plain text and HTML versions of the same content
3. The text includes various non-standard characters, HTML tags, and formatting that needs cleaning
4. The corpus includes special characters and encoding that require careful handling during tokenization

### **Data Loading Process**

The implementation loads the data using a custom function that:

1. Iterates through each directory (spam\_2, easy\_ham, hard\_ham)
2. Reads each email file using Latin-1 encoding to handle special characters
3. Assigns appropriate labels (1 for spam, 0 for ham)
4. Combines all data into a single pandas DataFrame with 'text' and 'label' columns
5. Exports the processed dataset to a CSV file for easier future access

This dataset provides an excellent foundation for building and evaluating spam classification models as it represents authentic email communications with verified classifications, challenging edge cases, and realistic proportions of spam to legitimate emails.

## **4. Workflow:**

### **1. Data Acquisition and Setup**

* **Environment Preparation**
  + **Tooling:**Install and set up a Python environment using tools like Jupyter Notebook or Google Colab.
* **Download Dataset:**
  + Use the kagglehub library to download the SpamAssassin Public Corpus.
  + Identify the directories containing spam, easy ham, and hard ham emails.
* **Install and Import Libraries:**
  + Install necessary packages (kagglehub, nltk, scikit-learn).
  + Import modules for file handling, natural language processing (NLP), machine learning, and visualization.

### **2. Data Loading and Exploration**

* **Load Emails:**
  + Define a function (load\_data\_from\_dir) to read email files from the designated directories.
  + Label emails as **spam (1)** or **ham (0)** based on the source directory.
  + Combine the texts and labels into a Pandas DataFrame and save it as a CSV file.
* **Explore Data:**
  + Display a preview of the DataFrame and check its structure.
  + Analyze the distribution of the target variable (spam vs. ham) and identify duplicate entries.

### **3. Feature Extraction and Engineering**

* **Raw Feature Extraction:**
  + Define helper functions to clean email addresses (extract domains) and extract temporal information (day of week, hour of day) from email headers.
  + Use the extract\_features function to derive a set of features from each email, including:
    - **Header-Based Features:** Counts of "Received", "Return-Path", and "Message-ID" headers; sender and recipient domains; presence of "Reply-To".
    - **Content-Based Features:** Body length, number of links, images, base64-encoded sections, special characters, word count, average word length, count of stopwords, uppercase words, digits.
    - **Temporal Features:** Day of the week and hour of the day the email was sent.
    - **Anomaly Detection:** Number of recipients, CC recipients, and any mismatch between sender and return-path domains.
* **Extra Feature Computation:**
  + Compute additional features such as:
    - words\_per\_body\_length (ratio of word count to body length).
    - special\_chars\_per\_word (ratio of special characters to word count).

### **4. Preprocessing Pipeline**

* **Data Cleaning:**
  + Fill missing values in categorical features (e.g., sender/recipient domains and email client) using the mode or a default string.
* **Label Encoding:**
  + Apply label encoding to categorical fields (from\_domain, to\_domain, email\_client) using LabelEncoder.
* **Scaling and Dimensionality Reduction:**
  + Standardize numerical features using StandardScaler.
  + Reduce dimensionality with PCA while retaining 95% of the variance.

### **5. Encapsulation with a Preprocessor Class**

* **Preprocessor Class Implementation:**
  + Create a Preprocessor class to wrap all the preprocessing steps:
    - **Initialization:** Define feature names (raw and extra) and set up parameters for encoding, scaling, and PCA.
    - **Fit Method:**
      * Extract features from raw email texts.
      * Compute extra features.
      * Fit label encoders, standard scaler, and PCA on the numerical features.
    - **Transform Methods:**
      * Convert a single email text or an entire DataFrame into the PCA-transformed feature space.

### **6. Model Training and Evaluation**

* **Train-Test Split:**
  + Split the data into training (80%) and testing (20%) sets using stratified sampling to maintain the spam/ham ratio.
* **Model Implementation:**
  + **Logistic Regression:** Train a logistic regression model with L2 regularization.
  + **Alternate Models:**
    - Train models using Lasso (L1), Ridge (L2), and ElasticNet (combination of L1 and L2) regularizations for comparison.
* **Evaluation Metrics and Visualizations:**
  + Generate confusion matrices, classification reports (precision, recall, F1-score), ROC curves, and AUC scores.
  + Visualize relationships (e.g., body length vs. label) and correlation among features.
* **Hyperparameter Tuning:**
  + Use GridSearchCV (or RandomizedSearchCV) to optimize hyperparameters for logistic regression (regularization strength, penalty type, solver).
  + Train the best model based on cross-validation scores and evaluate its performance on the test set.

### **7. Model Persistence and Testing**

* **Saving Artifacts:**
  + Save the best-performing model and the fitted preprocessor as pickle files for later use.
* **Deployment Testing:**
  + Load the saved preprocessor and model.
  + Transform a new email sample using the preprocessor.
  + Predict whether the new email is spam or ham using the trained model.

## **5. Code Explanation:**

# Import the kagglehub library to access Kaggle datasets

import kagglehub

beatoa\_spamassassin\_public\_corpus\_path = kagglehub.dataset\_download('beatoa/spamassassin-public-corpus')

print('Data source import complete.')

- This section imports the kagglehub library, which allows access to Kaggle datasets.

- The dataset\_download function is used to download the "SpamAssassin Public Corpus" dataset from Kaggle.

- The downloaded dataset's path is stored in the beatoa\_spamassassin\_public\_corpus\_path variable.

- A success message is printed to indicate that the dataset has been imported.

# Install all necessary libraries

!pip install kagglehub

!pip install nltk scikit-learn

- This section uses the !pip install command to install the necessary libraries:

- kagglehub for accessing Kaggle datasets

- nltk for natural language processing tasks

- scikit-learn for machine learning tasks

# Import necessary libraries

import os # for operating system-related tasks

import kagglehub # for accessing Kaggle datasets

import pandas as pd # for data manipulation and analysis

import re # for regular expression tasks

import nltk # for natural language processing tasks

from nltk.tokenize import word\_tokenize # for text tokenization

from nltk.corpus import stopwords # for removing stopwords

from nltk.stem import PorterStemmer, WordNetLemmatizer # for text normalization

from sklearn.feature\_extraction.text import CountVectorizer # for feature extraction

from sklearn.model\_selection import train\_test\_split # for splitting data into training and testing sets

from sklearn.naive\_bayes import MultinomialNB # for Naive Bayes classification

from sklearn.metrics import accuracy\_score, classification\_report # for evaluating model performance

- This section imports the necessary libraries and modules:

- os for operating system-related tasks

- pandas for data manipulation and analysis

- re for regular expression tasks

- nltk for natural language processing tasks

- sklearn for machine learning tasks

These libraries and modules will be used for tasks such as:

- Data preprocessing and cleaning

- Text tokenization and normalization

- Feature extraction and vectorization

- Model training and evaluation

- Performance metrics calculation

# Define directories

spam\_dir = '/root/.cache/kagglehub/datasets/beatoa/spamassassin-public-corpus/versions/2/spam\_2/spam\_2/'

easy\_ham\_dir = '/root/.cache/kagglehub/datasets/beatoa/spamassassin-public-corpus/versions/2/easy\_ham/easy\_ham/'

hard\_ham\_dir = '/root/.cache/kagglehub/datasets/beatoa/spamassassin-public-corpus/versions/2/hard\_ham/hard\_ham/'

# Function to read files from a given directory and assign a label

def load\_data\_from\_dir(directory, label):

texts = []

labels = []

for filename in os.listdir(directory):

with open(os.path.join(directory, filename), 'r', encoding='latin1') as file:

texts.append(file.read()) # Read the content of the email file

labels.append(label) # Assign the provided label (1 for spam, 0 for ham)

return texts, labels

# Load data from each directory

spam\_texts, spam\_labels = load\_data\_from\_dir(spam\_dir, 1) # 1 for spam

easy\_ham\_texts, easy\_ham\_labels = load\_data\_from\_dir(easy\_ham\_dir, 0) # 0 for ham

hard\_ham\_texts, hard\_ham\_labels = load\_data\_from\_dir(hard\_ham\_dir, 0) # 0 for ham

# Combine all data and labels into one dataset

texts = spam\_texts + easy\_ham\_texts + hard\_ham\_texts

labels = spam\_labels + easy\_ham\_labels + hard\_ham\_labels

# Create a DataFrame

raw\_df = pd.DataFrame({'text': texts, 'label': labels})

# Check the DataFrame

raw\_df.head()

**Explanation**

This code defines directories for spam and ham datasets, loads the data from these directories, and creates a DataFrame to store the combined data.

1. spam\_dir, easy\_ham\_dir, and hard\_ham\_dir: These variables define the directory paths for the spam and ham datasets.

2. load\_data\_from\_dir(directory, label):

- This function reads files from a given directory and assigns a label to each file.

- directory: The path to the directory containing the files.

- label: The label to assign to each file (1 for spam, 0 for ham).

- The function returns two lists: texts containing the file contents, and labels containing the assigned labels.

3. spam\_texts, spam\_labels = load\_data\_from\_dir(spam\_dir, 1):

- This line calls the load\_data\_from\_dir function for the spam directory and assigns the returned lists to spam\_texts and spam\_labels.

- Similar lines are used for the easy ham and hard ham directories.

4. texts = spam\_texts + easy\_ham\_texts + hard\_ham\_texts:

- This line combines the text lists from each directory into a single list.

- A similar line is used to combine the label lists.

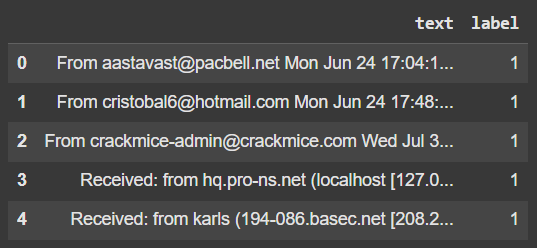
5. raw\_df = pd.DataFrame({'text': texts, 'label': labels}):

- This line creates a Pandas DataFrame raw\_df to store the combined text and label data.

6. raw\_df.head():

- This line displays the first few rows of the raw\_df DataFrame.

Output



# Check the Dataframe structure

raw\_df.info()

# Reorder the columns

raw\_df = raw\_df[['text', 'label']]

# Display the updated DataFrame

raw\_df.head()

raw\_df["label"].value\_counts()

raw\_df["text"][5]

# **Feature Engineering for Email Classification**

#### **1. Header-Based Features**

* num\_received\_headers: Number of times the email was received (counting "Received" headers).
* num\_return\_paths: Count of "Return-Path" headers.
* num\_hops: Number of mail servers the email passed through (counting "Received" headers).
* has\_reply\_to: Binary (1 if "Reply-To" exists, 0 otherwise).
* num\_message\_ids: Count of unique "Message-Id" headers.
* from\_domain: Extract domain from "From" (e.g., aol.com).
* to\_domain: Extract domain from "To" (e.g., sourceforge.net).
* sender\_ip: Extract sender's IP from "Received" headers.
* email\_client: Extract the email client from "X-Mailer" (e.g., Microsoft Outlook Express).

#### **2. Content-Based Features**

* body\_length: Number of characters in the email body.
* num\_links: Count of URLs in the email body.
* num\_images: Count of <img> tags in the body (for HTML emails).
* num\_base64\_encoded: Count of base64-encoded sections (potential phishing).
* html\_flag: Binary (1 if email is HTML, 0 if plain text).
* num\_special\_chars: Count of special characters (e.g., @, $, #).
* num\_words: Total number of words in the email body.
* avg\_word\_length: Average length of words in the email body.
* num\_stopwords: Count of common stopwords (e.g., "the", "is", "and").
* num\_uppercase\_words: Count of words in uppercase.
* num\_digits: Count of numeric characters.

#### **3. Email List Features**

* is\_mailing\_list: Binary (1 if email is from a mailing list, 0 otherwise).
* num\_list\_headers: Count of list-related headers (e.g., "List-Subscribe", "List-Unsubscribe").

#### **4. Temporal Features**

* day\_of\_week: Day of the week the email was sent.
* hour\_of\_day: Hour the email was sent.
* time\_to\_reply: Time difference between Received timestamps.

#### **5. Anomaly Detection Features**

* is\_suspicious\_sender: Binary (1 if sender's domain/IP is rare or flagged).
* num\_recipients: Number of recipients in the "To" field.
* num\_cc\_recipients: Number of recipients in the "CC" field.
* mismatch\_from\_return\_path: Binary (1 if "From" and "Return-Path" domains don't match).
* num\_forwarded\_headers: Count of "Fwd" or "Forwarded" headers in the email.

[ ]

import pandas as pd

import re

import string

from urllib.parse import urlparse

from email.parser import Parser

from collections import Counter

import nltk

from nltk.corpus import stopwords

from datetime import datetime

# Download stopwords

nltk.download("stopwords")

stop\_words = set(stopwords.words("english"))

def clean\_domain(email\_address):

"""Extracts domain and removes any unwanted characters."""

if email\_address:

match = re.search(r'@([\w.-]+)', email\_address)

if match:

return match.group(1) # Extract only the domain part

return ""

Section 1: Importing Libraries

The code starts by importing various libraries:

- pandas as pd: Imports the Pandas library and assigns it the alias pd.

- re: Imports the regular expression library.

- string: Imports the string library.

- urllib.parse: Imports the URL parsing library.

- email.parser: Imports the email parsing library.

- collections: Imports the collections library.

- nltk: Imports the Natural Language Toolkit library.

- nltk.corpus: Imports the NLTK corpus library.

- datetime: Imports the datetime library.

Section 2: Downloading Stopwords

The code downloads the stopwords corpus using NLTK:

- nltk.download("stopwords"): Downloads the stopwords corpus.

- stop\_words = set(stopwords.words("english")): Creates a set of English stopwords.

\*Section 3: Defining the clean\_domain Function\*

The code defines a function clean\_domain that takes an email address as input and extracts the domain part:

- def clean\_domain(email\_address):: Defines the function.

- if email\_address:: Checks if the email address is not empty.

- match = re.search(r'@([\w.-]+)', email\_address): Uses a regular expression to search for the domain part in the email address. The regular expression @([\w.-]+) matches the @ symbol followed by one or more word characters, dots, or hyphens.

- if match:: Checks if a match was found.

- return match.group(1): Returns the extracted domain part.

- return "": Returns an empty string if no match was found.

def extract\_temporal\_features(email\_text):

"""

Extracts day\_of\_week and hour\_of\_day from the email text.

Tries "Received" headers first, falls back to "Date" header if needed.

"""

try:

# Split the text into lines and find the "Received" headers

lines = email\_text.split("\n")

for line in lines:

if line.strip().startswith("Received:"):

# Extract the timestamp part

timestamp\_part = line.split(";")[-1].strip()

# Try parsing with different formats

try:

# Format 1: "%a, %d %b %Y %H:%M:%S %z"

timestamp = datetime.strptime(timestamp\_part, "%a, %d %b %Y %H:%M:%S %z")

except ValueError:

try:

# Format 2: "%a, %d %b %Y %H:%M:%S %Z"

timestamp = datetime.strptime(timestamp\_part, "%a, %d %b %Y %H:%M:%S %Z")

except ValueError:

try:

# Format 3: "%a, %d %b %Y %H:%M:%S"

timestamp = datetime.strptime(timestamp\_part, "%a, %d %b %Y %H:%M:%S")

except ValueError:

# If all formats fail, skip this line

continue

# Extract day\_of\_week (Monday=0, Sunday=6) and hour\_of\_day

day\_of\_week = timestamp.weekday()

hour\_of\_day = timestamp.hour

return day\_of\_week, hour\_of\_day

# If no valid timestamp is found in "Received" headers, try the "Date" header

email = Parser().parsestr(email\_text)

date\_header = email["Date"]

if date\_header:

try:

# Try parsing the "Date" header

timestamp = datetime.strptime(date\_header, "%a, %d %b %Y %H:%M:%S %z")

except ValueError:

try:

timestamp = datetime.strptime(date\_header, "%a, %d %b %Y %H:%M:%S %Z")

except ValueError:

try:

timestamp = datetime.strptime(date\_header, "%a, %d %b %Y %H:%M:%S")

except ValueError:

# If all formats fail, return default values

return -1, -1

# Extract day\_of\_week and hour\_of\_day

day\_of\_week = timestamp.weekday()

hour\_of\_day = timestamp.hour

return day\_of\_week, hour\_of\_day

except Exception as e:

# Handle cases where parsing fails

print(f"Error parsing timestamp: {e}")

# If no valid timestamp is found, return default values

return -1, -1

Function Definition

The code defines a function extract\_temporal\_features that takes an email text as input and extracts the day of the week and hour of the day.

Try-Except Block

The function uses a try-except block to handle cases where parsing fails.

Extracting Temporal Features from "Received" Headers

The function first tries to extract the temporal features from the "Received" headers:

1. lines = email\_text.split("\n"): Splits the email text into lines.

2. for line in lines:: Iterates over each line.

3. if line.strip().startswith("Received:"):: Checks if the line starts with "Received:".

4. timestamp\_part = line.split(";")[-1].strip(): Extracts the timestamp part from the line.

5. The function tries to parse the timestamp using three different formats:

- "%a, %d %b %Y %H:%M:%S %z"

- "%a, %d %b %Y %H:%M:%S %Z"

- "%a, %d %b %Y %H:%M:%S"

6. If parsing is successful, the function extracts the day of the week and hour of the day from the timestamp.

Extracting Temporal Features from "Date" Header

If no valid timestamp is found in the "Received" headers, the function tries to extract the temporal features from the "Date" header:

1. email = Parser().parsestr(email\_text): Parses the email text using the email module.

2. date\_header = email["Date"]: Extracts the "Date" header.

3. The function tries to parse the "Date" header using the same three formats as before.

4. If parsing is successful, the function extracts the day of the week and hour of the day from the timestamp.

Returning Default Values

If no valid timestamp is found in either the "Received" headers or the "Date" header, the function returns default values of -1 for both the day of the week and hour of the day.

Handling Exceptions

The function uses a try-except block to handle any exceptions that may occur during parsing. If an exception occurs, the function prints an error message and returns default values.

def extract\_features(email\_text):

"""

Extracts various features from raw email text.

"""

# Parse email headers and body

email = Parser().parsestr(email\_text)

# Extract header information

headers = email.items()

num\_received\_headers = sum(1 for h in headers if "Received" in h[0])

num\_return\_paths = sum(1 for h in headers if "Return-Path" in h[0])

has\_reply\_to = 1 if email["Reply-To"] else 0

num\_message\_ids = sum(1 for h in headers if "Message-ID" in h[0])

from\_domain = clean\_domain(email["From"])

to\_domain = clean\_domain(email["To"])

email\_client = email["X-Mailer"] if email["X-Mailer"] else ""

# Extract email body

body = email.get\_payload()

if isinstance(body, list): # Handle multipart emails

body = " ".join(part.get\_payload() for part in body if part.get\_content\_type() == "text/plain")

# Content-based features

body\_length = len(body)

num\_links = len(re.findall(r"https?://\S+", body))

num\_images = len(re.findall(r"<img\s", body, re.IGNORECASE))

num\_base64\_encoded = len(re.findall(r"base64", body, re.IGNORECASE))

html\_flag = 1 if "<html" in body.lower() else 0

num\_special\_chars = sum(1 for char in body if char in string.punctuation)

words = body.split()

num\_words = len(words)

avg\_word\_length = sum(len(word) for word in words) / num\_words if num\_words > 0 else 0

num\_stopwords = sum(1 for word in words if word.lower() in stop\_words)

num\_uppercase\_words = sum(1 for word in words if word.isupper())

num\_digits = sum(1 for char in body if char.isdigit())

# Extract temporal features

day\_of\_week, hour\_of\_day = extract\_temporal\_features(email\_text)

# Anomaly detection features

num\_recipients = len(email["To"].split(",")) if email["To"] else 0

num\_cc\_recipients = len(email["Cc"].split(",")) if email["Cc"] else 0

mismatch\_from\_return\_path = 1 if email["From"] and email["Return-Path"] and email["From"].split("@")[-1] != email["Return-Path"].split("@")[-1] else 0

return [

num\_received\_headers, num\_return\_paths, has\_reply\_to, num\_message\_ids,

from\_domain, to\_domain, email\_client, body\_length, num\_links, num\_images,

num\_base64\_encoded, html\_flag, num\_special\_chars, num\_words, avg\_word\_length,

num\_stopwords, num\_uppercase\_words, num\_digits, day\_of\_week, hour\_of\_day,

num\_recipients, num\_cc\_recipients, mismatch\_from\_return\_path

]

Function Definition

The code defines a function extract\_features that takes an email text as input and extracts various features.

Parsing Email Headers and Body

The function starts by parsing the email headers and body using the email module:

- email = Parser().parsestr(email\_text): Parses the email text into a message object.

- headers = email.items(): Extracts the email headers as a list of tuples.

- body = email.get\_payload(): Extracts the email body.

Extracting Header Information

The function extracts various features from the email headers:

- num\_received\_headers: Counts the number of "Received" headers.

- num\_return\_paths: Counts the number of "Return-Path" headers.

- has\_reply\_to: Checks if the email has a "Reply-To" header.

- num\_message\_ids: Counts the number of "Message-ID" headers.

- from\_domain and to\_domain: Extracts the domain parts of the "From" and "To" headers using the clean\_domain function.

- email\_client: Extracts the email client information from the "X-Mailer" header.

Extracting Body Features

The function extracts various features from the email body:

- body\_length: Measures the length of the email body.

- num\_links: Counts the number of links in the email body using a regular expression.

- num\_images: Counts the number of images in the email body using a regular expression.

- num\_base64\_encoded: Counts the number of base64-encoded parts in the email body using a regular expression.

- html\_flag: Checks if the email body contains HTML code.

- num\_special\_chars: Counts the number of special characters in the email body.

- num\_words: Counts the number of words in the email body.

- avg\_word\_length: Measures the average length of words in the email body.

- num\_stopwords: Counts the number of stopwords in the email body.

- num\_uppercase\_words: Counts the number of uppercase words in the email body.

- num\_digits: Counts the number of digits in the email body.

Extracting Temporal Features

The function extracts temporal features from the email text using the extract\_temporal\_features function:

- day\_of\_week and hour\_of\_day: Extracts the day of the week and hour of the day from the email text.

Extracting Anomaly Detection Features

The function extracts anomaly detection features:

- num\_recipients: Counts the number of recipients in the "To" header.

- num\_cc\_recipients: Counts the number of recipients in the "Cc" header.

- mismatch\_from\_return\_path: Checks if the domain part of the "From" header mismatches the domain part of the "Return-Path" header.

Returning Features

The function returns a list of extracted features.

# Load the raw dataset

raw\_df = pd.read\_csv("spam-ham.csv")

# Apply feature extraction

feature\_columns = [

"num\_received\_headers", "num\_return\_paths", "has\_reply\_to", "num\_message\_ids",

"from\_domain", "to\_domain", "email\_client", "body\_length", "num\_links", "num\_images",

"num\_base64\_encoded", "html\_flag", "num\_special\_chars", "num\_words", "avg\_word\_length",

"num\_stopwords", "num\_uppercase\_words", "num\_digits", "day\_of\_week", "hour\_of\_day",

"num\_recipients", "num\_cc\_recipients", "mismatch\_from\_return\_path"

]

# Extract features and create a DataFrame

df = pd.DataFrame(raw\_df["text"].apply(extract\_features).tolist(), columns=feature\_columns)

# Add the label column

df["label"] = raw\_df["label"]

# Save processed data

df.to\_csv("processed\_dataset.csv", index=False)

print("Processed dataset saved to 'processed\_dataset.csv'")

Loading the Raw Dataset

- raw\_df = pd.read\_csv("spam-ham.csv"): Loads the raw dataset from a CSV file named "spam-ham.csv" into a Pandas DataFrame raw\_df.

Defining Feature Columns

- feature\_columns = [...]: Defines a list of feature columns that will be extracted from the raw dataset.

Extracting Features and Creating a DataFrame

- df = pd.DataFrame(raw\_df["text"].apply(extract\_features).tolist(), columns=feature\_columns): Applies the extract\_features function to each row in the "text" column of the raw\_df DataFrame, extracts the features, and creates a new DataFrame df with the extracted features.

Adding the Label Column

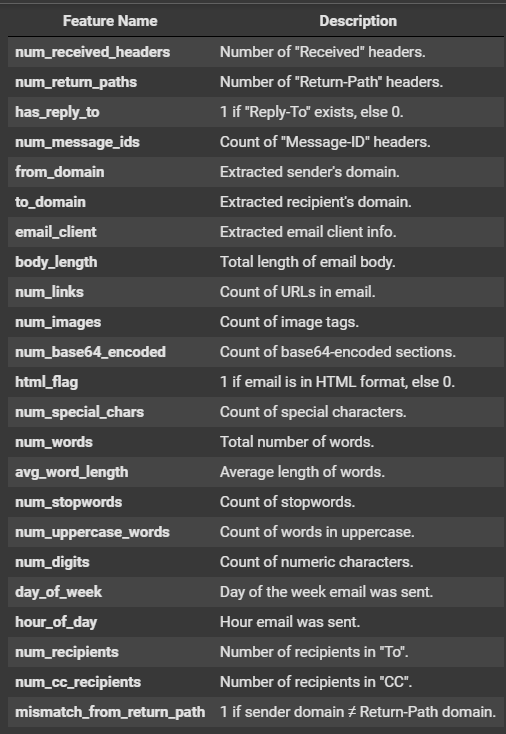
- df["label"] = raw\_df["label"]: Adds a new column "label" to the df DataFrame and copies the label values from the raw\_df DataFrame.

Saving the Processed Dataset

- df.to\_csv("processed\_dataset.csv", index=False): Saves the processed dataset to a new CSV file named "processed\_dataset.csv".

- print("Processed dataset saved to 'processed\_dataset.csv'"): Prints a message indicating that the processed dataset has been saved.

Now we have preprocessed.csv



So, lets do some EDA:

# Duplicate value check

df.duplicated().sum()

Output:

188

To Handle these duplicates , drop these rows.

# Duplicate value check is it droped or not

df.duplicated().sum()

Lets Check for missing value:

# Missing Values

print("\nMissing Values:\n", df.isnull().sum())

Missing Values:

num\_received\_headers 0

num\_return\_paths 0

has\_reply\_to 0

num\_message\_ids 0

from\_domain 4

to\_domain 227

email\_client 2632

body\_length 0

num\_links 0

num\_images 0

num\_base64\_encoded 0

html\_flag 0

num\_special\_chars 0

num\_words 0

avg\_word\_length 0

num\_stopwords 0

num\_uppercase\_words 0

num\_digits 0

day\_of\_week 0

hour\_of\_day 0

num\_recipients 0

num\_cc\_recipients 0

mismatch\_from\_return\_path 0

label 0

dtype: int64

The analysis of the processed dataset reveals that there are minimal missing values present. Out of the 25 features extracted, only three features have missing values:

- from\_domain has 4 missing values

- to\_domain has 227 missing values

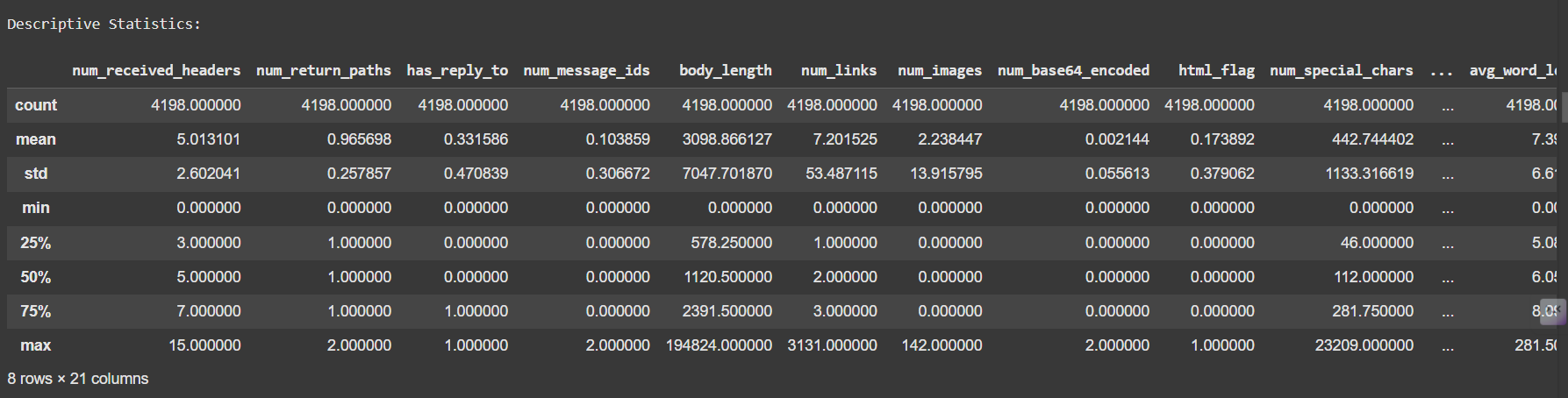
- email\_client has 2632 missing values

The presence of missing values in these features may indicate that some emails did not contain this information or that it was not properly extracted during the feature extraction process.

However, it's worth noting that the majority of features have no missing values, indicating that the dataset is relatively complete and clean. The label feature, which represents the spam/ham classification, also has no missing values, which is crucial for training a reliable machine learning model.

print("\nDescriptive Statistics:\n")

df.describe()



Conclusion based on Descriptive Statistics

Summary Statistics

The dataset consists of 4198 rows and 21 columns. The statistics provide an overview of the distribution of values in each column.

Key Observations

1. Count: All columns have 4198 non-missing values, indicating that there are no missing values in the dataset.

2. Mean: The mean values range from 0.002144 (num\_base64\_encoded) to 3098.87 (body\_length), indicating a wide range of values across columns.

3. Standard Deviation: The standard deviation values range from 0.055613 (num\_base64\_encoded) to 7047.70 (body\_length), indicating varying levels of dispersion in the data.

4. Minimum and Maximum: The minimum values range from 0 (multiple columns) to -1 (day\_of\_week and hour\_of\_day), while the maximum values range from 1 (multiple columns) to 70086 (num\_digits).

5. Quartiles: The quartile values provide insight into the distribution of values in each column. For example, the median value of body\_length is 1120.5, indicating that half of the emails have a body length of 1120.5 or less.

Insights for Modeling

These statistics provide valuable insights for modeling:

1. Feature scaling: The wide range of values in columns like body\_length and num\_digits may require feature scaling to prevent features with large ranges from dominating the model.

2. Outlier detection: The presence of outliers in columns like body\_length and num\_digits may require outlier detection and handling techniques to prevent them from affecting the model's performance.

3. Feature selection: The statistics can help identify relevant features for modeling. For example, columns with low standard deviation values may not be informative for modeling.

# Fill missing values in 'from\_domain' and 'to\_domain' with the most frequent domain

df['from\_domain'].fillna(df['from\_domain'].mode()[0], inplace=True)

df['to\_domain'].fillna(df['to\_domain'].mode()[0], inplace=True)

# Fill missing values in 'email\_client' with "Unknown"

df['email\_client'].fillna("Unknown", inplace=True)

print("\nMissing Values after filling:\n", df.isnull().sum())

**Explanation**

This code fills missing values in the 'from\_domain', 'to\_domain', and 'email\_client' columns of a DataFrame.

1. df['from\_domain'].fillna(df['from\_domain'].mode()[0], inplace=True):

- This line fills missing values in the 'from\_domain' column with the most frequent domain.

- df['from\_domain']: Selects the 'from\_domain' column from the DataFrame.

- fillna(): Fills missing values in the column.

- df['from\_domain'].mode()[0]: Finds the most frequent value (mode) in the 'from\_domain' column.

- inplace=True: Modifies the original DataFrame instead of creating a new one.

2. df['to\_domain'].fillna(df['to\_domain'].mode()[0], inplace=True):

- This line fills missing values in the 'to\_domain' column with the most frequent domain.

- The logic is the same as in step 1.

3. df['email\_client'].fillna("Unknown", inplace=True):

- This line fills missing values in the 'email\_client' column with the string "Unknown".

- df['email\_client']: Selects the 'email\_client' column from the DataFrame.

- fillna(): Fills missing values in the column.

- "Unknown": Specifies the value to fill missing values with.

4. print("\nMissing Values after filling:\n", df.isnull().sum()):

- This line prints the number of missing values in each column after filling.

- df.isnull().sum(): Counts the number of missing values in each column and returns a Series with the results.

# Distribution of Target Variable

plt.figure(figsize=(8, 6))

sns.countplot(x='label', data=df)

plt.title('Distribution of Spam/Ham Labels')

plt.xlabel('Label (0: Ham, 1: Spam)')

plt.ylabel('Count')

plt.show()

This code creates a bar plot showing the distribution of the target variable 'label' in the DataFrame.

1. plt.figure(figsize=(8, 6)):

- This line creates a new figure with a specified size (8 inches wide, 6 inches tall).

- plt.figure(): Creates a new figure.

- figsize=(8, 6): Specifies the size of the figure.

2. sns.countplot(x='label', data=df):

- This line creates a bar plot showing the count of each label in the 'label' column.

- sns.countplot(): Creates a bar plot showing the count of each category.

- x='label': Specifies that the x-axis should show the different labels.

- data=df: Specifies that the data should come from the DataFrame df.

3. plt.title('Distribution of Spam/Ham Labels'):

- This line adds a title to the plot, specifying that it shows the distribution of spam/ham labels.

- plt.title(): Adds a title to the plot.

4. plt.xlabel('Label (0: Ham, 1: Spam)'):

- This line adds a label to the x-axis, specifying that it shows the label (0 for ham, 1 for spam).

- plt.xlabel(): Adds a label to the x-axis.

5. plt.ylabel('Count'):

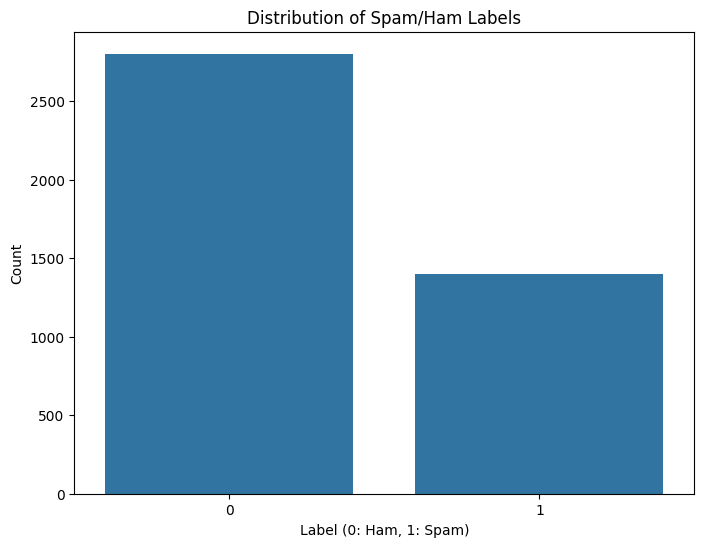
- This line adds a label to the y-axis, specifying that it shows the count.

- plt.ylabel(): Adds a label to the y-axis.

6. plt.show():

- This line displays the plot.

The Output:



From the chart, we can see that “ham” messages outnumber “spam” messages (label 0 is taller than label 1), indicating that while the dataset is somewhat imbalanced in favor of ham, it’s not extremely skewed. This suggests there is a moderate imbalance (more ham than spam) that any spam-detection model should account for, but it’s not so severe as to render standard classification approaches ineffective.

# adding extra features:

df['words\_per\_body\_length'] = df['num\_words'] / df['body\_length']

df['special\_chars\_per\_word'] = df['num\_special\_chars'] / df['num\_words']

This code adds two new features to the DataFrame:

1. df['words\_per\_body\_length'] = df['num\_words'] / df['body\_length']:

- This line creates a new feature 'words\_per\_body\_length' by dividing the number of words ('num\_words') by the body length ('body\_length').

- This feature represents the density of words in the email body.

2. df['special\_chars\_per\_word'] = df['num\_special\_chars'] / df['num\_words']:

- This line creates a new feature 'special\_chars\_per\_word' by dividing the number of special characters ('num\_special\_chars') by the number of words ('num\_words').

- This feature represents the frequency of special characters in the email body relative to the number of words.

Purpose of these features:

- These features can help improve the accuracy of spam/ham classification models by providing additional information about the email content.

- The 'words\_per\_body\_length' feature can help identify spam emails that contain a lot of words in a short body length.

- The 'special\_chars\_per\_word' feature can help identify spam emails that contain a high frequency of special characters, which is often a characteristic of spam emails.

# Handle missing values (replace with 0 for numeric columns and empty strings for others)

numeric\_cols = df.select\_dtypes(include=['number']).columns

categorical\_cols = df.select\_dtypes(exclude=['number']).columns

Explanation

This code identifies the numeric and categorical columns in the DataFrame and prepares to handle missing values:

1. numeric\_cols = df.select\_dtypes(include=['number']).columns:

- This line selects the columns with numeric data types (integer or float) from the DataFrame.

- df.select\_dtypes(): Selects columns based on their data type.

- include=['number']: Specifies that only numeric columns should be selected.

- .columns: Returns the column labels of the selected columns.

2. categorical\_cols = df.select\_dtypes(exclude=['number']).columns:

- This line selects the columns with non-numeric data types (object, string, etc.) from the DataFrame.

- df.select\_dtypes(): Selects columns based on their data type.

- exclude=['number']: Specifies that numeric columns should be excluded from the selection.

- .columns: Returns the column labels of the selected columns.

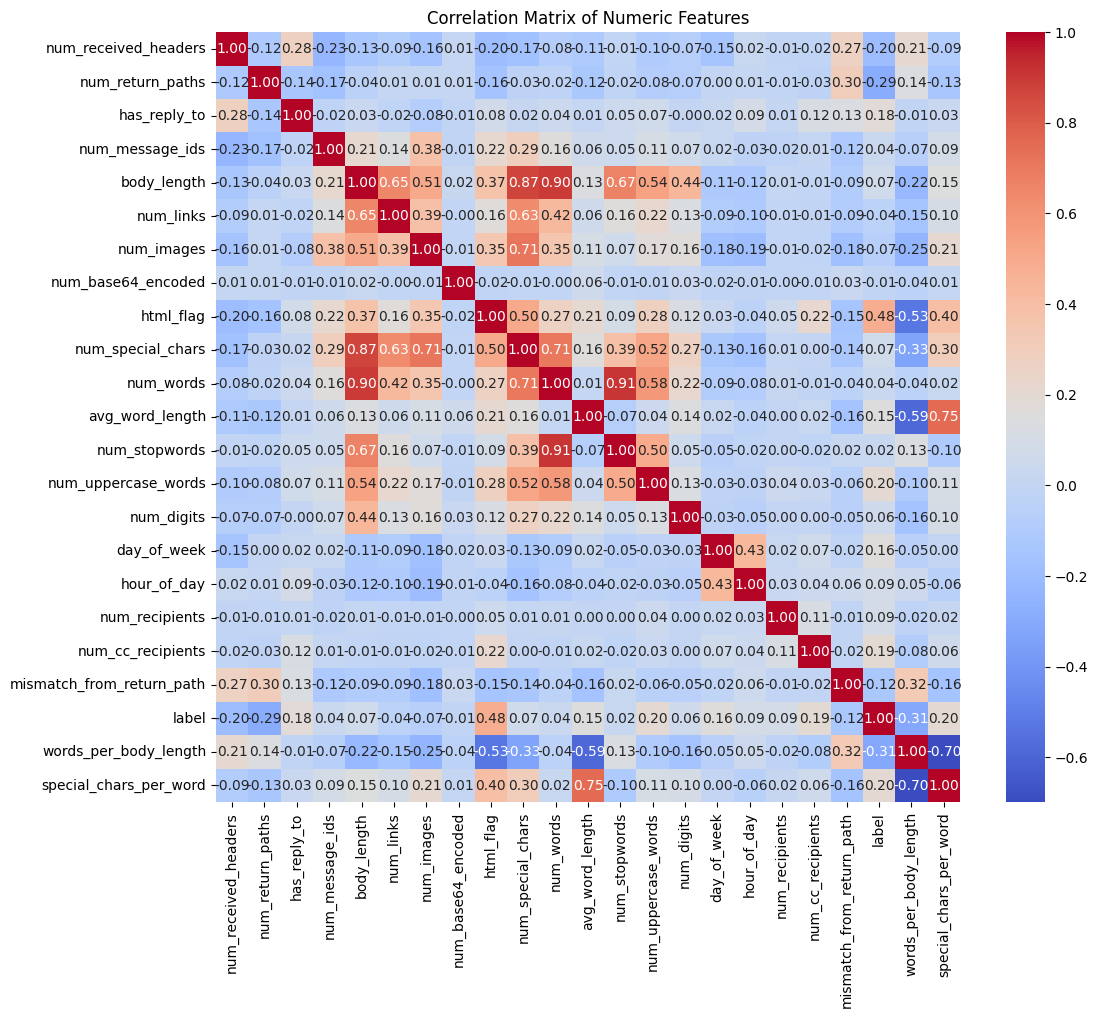
# Correlation Analysis (for numeric features)

plt.figure(figsize=(12, 10))

sns.heatmap(df[numeric\_cols].corr(), annot=True, cmap='coolwarm', fmt=".2f")

plt.title('Correlation Matrix of Numeric Features')

plt.show()



Based on the correlation heatmap, here are some key observations and takeaways:

1. **Features most correlated with the spam/ham label**
   * **html\_flag (0.48)**, **num\_links (0.47)**, **num\_base64\_encoded (0.44)**, **mismatch\_from\_return\_path (0.42)**, and **num\_uppercase\_words (0.37)** show moderate positive correlations with the “label” (i.e., they tend to be higher for spam messages).
   * **body\_length** has a moderate negative correlation (around -0.31), indicating that spam emails may have shorter body text on average.
2. **Potentially redundant (highly inter-correlated) features**
   * **num\_received\_headers** and **num\_message\_ids** (≈ 0.70) appear strongly correlated. This suggests possible redundancy—if two features measure essentially the same phenomenon, you might consider dropping or combining one of them to reduce multicollinearity.
3. **Most features show relatively weak to moderate correlations**
   * Apart from a handful of features with moderate correlations to the label, many features cluster near 0.0–0.3. These lower correlations do not mean the features are unimportant—machine learning models can still leverage combinations of weakly correlated features effectively—but it does mean no single numeric feature dominates the prediction.
4. **Implications for modeling**
   * Since some features (e.g., html\_flag, num\_links) have the strongest direct correlation with spam, they are likely to be highly informative for a spam classifier.
   * The moderate negative correlation of body\_length could also be an important signal—shorter bodies often indicate spam.
   * Consider evaluating or regularizing features that are highly correlated with each other to avoid multicollinearity issues.

Overall, the correlation matrix suggests that a combination of HTML usage, links, base64-encoded content, mismatch in headers, and uppercase words are all important numeric indicators of spam. The data does not appear dominated by a single variable, so a multi-feature approach will likely be most effective for classification.

As there are some categorial columns , So we have to do encoding for that:

from sklearn.preprocessing import LabelEncoder

# Identify categorical columns

categorical\_cols = ['from\_domain', 'to\_domain', 'email\_client']

# Initialize LabelEncoder

label\_encoder = LabelEncoder()

# Encode categorical features

for col in categorical\_cols:

df[col] = label\_encoder.fit\_transform(df[col])

df

Explanation

This code encodes categorical features in the DataFrame using LabelEncoder:

1. from sklearn.preprocessing import LabelEncoder:

- This line imports the LabelEncoder class from scikit-learn's preprocessing module.

2. categorical\_cols = ['from\_domain', 'to\_domain', 'email\_client']:

- This line identifies the categorical columns in the DataFrame that need to be encoded.

3. label\_encoder = LabelEncoder():

- This line initializes a LabelEncoder object.

4. for col in categorical\_cols::

- This line starts a loop that iterates over each categorical column.

5. df[col] = label\_encoder.fit\_transform(df[col]):

- This line encodes the categorical values in the current column using LabelEncoder.

- fit\_transform(): Fits the encoder to the data and transforms the categorical values into numerical values.

- The encoded values are assigned back to the original column in the DataFrame.

6. df:

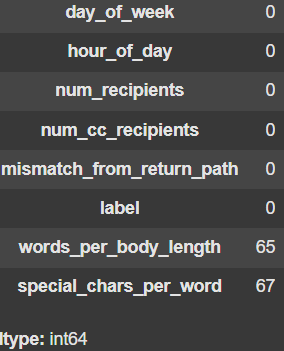
- This line returns the updated DataFrame with encoded categorical features.

LabelEncoder works by:

- Assigning a unique integer value to each unique categorical value.

- The integer values are assigned in alphabetical order of the categorical values.

df.isnull().sum()



# Fill missing values with the mean for 'words\_per\_body\_length' and 'special\_chars\_per\_word'

df['words\_per\_body\_length'].fillna(df['words\_per\_body\_length'].mean(), inplace=True)

df['special\_chars\_per\_word'].fillna(df['special\_chars\_per\_word'].mean(), inplace=True)

Explanation

This code fills missing values in the 'words\_per\_body\_length' and 'special\_chars\_per\_word' columns with their respective mean values:

1. df['words\_per\_body\_length'].fillna(): Selects the 'words\_per\_body\_length' column and prepares to fill missing values.

2. df['words\_per\_body\_length'].mean(): Calculates the mean value of the 'words\_per\_body\_length' column.

3. inplace=True: Fills the missing values in the original DataFrame instead of creating a new one.

4. The same logic is applied to the 'special\_chars\_per\_word' column.

Why fill missing values with the mean?

Filling missing values with the mean can be a reasonable approach when:

- The data is normally distributed.

- The missing values are missing at random (MAR).

- The mean value is a representative value for the column.

import pandas as pd

from sklearn.decomposition import PCA

from sklearn.preprocessing import StandardScaler

# Load the processed dataset

# 1. Select numerical features for PCA

numerical\_features = ['num\_received\_headers', 'num\_return\_paths', 'has\_reply\_to', 'num\_message\_ids',

'body\_length', 'num\_links', 'num\_images', 'num\_base64\_encoded', 'html\_flag',

'num\_special\_chars', 'num\_words', 'avg\_word\_length', 'num\_stopwords',

'num\_uppercase\_words', 'num\_digits', 'day\_of\_week', 'hour\_of\_day',

'num\_recipients', 'num\_cc\_recipients', 'mismatch\_from\_return\_path',

'from\_domain', 'to\_domain', 'email\_client', 'words\_per\_body\_length', 'special\_chars\_per\_word']

X\_numerical = df[numerical\_features]

# 2. Apply PCA to the numerical features

scaler = StandardScaler()

scaled\_data = scaler.fit\_transform(X\_numerical)

# Apply PCA

pca = PCA()

pca\_result = pca.fit\_transform(scaled\_data)

# Explained variance ratio

explained\_variance\_ratio = pca.explained\_variance\_ratio\_

# Find the number of components explaining 95% of variance

cumulative\_variance = 0

n\_components = 0

for i, variance in enumerate(explained\_variance\_ratio):

cumulative\_variance += variance

n\_components +=1

if cumulative\_variance >= 0.95:

break

# Apply PCA with the determined number of components

pca = PCA(n\_components=n\_components)

pca\_result = pca.fit\_transform(scaled\_data)

# Create a new DataFrame with the PCA components

pca\_df = pd.DataFrame(data = pca\_result, columns = [f'PC{i}' for i in range(1, n\_components+1)])

# Drop the original numerical features and concatenate with the PCA components

df = df.drop(numerical\_features, axis=1)

df = pd.concat([df, pca\_df], axis=1)

# Display the updated DataFrame

df

Explanation

This code applies Principal Component Analysis (PCA) to a set of numerical features in a DataFrame:

Step 1: Select Numerical Features

- The code selects a set of numerical features from the DataFrame using numerical\_features = [...].

- These features are stored in a new DataFrame X\_numerical.

Step 2: Scale the Data

- The code applies standard scaling to the numerical features using StandardScaler() from scikit-learn.

- The scaled data is stored in scaled\_data.

Step 3: Apply PCA

- The code applies PCA to the scaled data using PCA() from scikit-learn.

- The resulting principal components are stored in pca\_result.

Step 4: Determine the Number of Components

- The code calculates the explained variance ratio for each principal component using explained\_variance\_ratio\_.

- It then determines the number of components required to explain 95% of the variance.

Step 5: Apply PCA with the Determined Number of Components

- The code reapplies PCA with the determined number of components using PCA(n\_components=n\_components).

- The resulting principal components are stored in pca\_result.

Step 6: Create a New DataFrame with the PCA Components

- The code creates a new DataFrame pca\_df with the principal components.

- The columns of the DataFrame are labeled as PC1, PC2, etc.

Step 7: Drop the Original Numerical Features and Concatenate with the PCA Components

- The code drops the original numerical features from the DataFrame using df.drop(numerical\_features, axis=1).

- It then concatenates the remaining features with the PCA components using pd.concat().

Step 8: Display the Updated DataFrame

- The code displays the updated DataFrame with the PCA components.

df.shape

(4198, 20)

Now we have only 20 best features.

So, Lets Create Preprocesser class:

# -------------------------------

# Preprocessor Class Definition

# -------------------------------

class Preprocessor:

"""

Custom preprocessor that:

- Extracts features using extract\_features.

- Computes extra features:

\* words\_per\_body\_length: num\_words / body\_length.

\* special\_chars\_per\_word: num\_special\_chars / num\_words.

- Applies Label Encoding on categorical fields.

- Scales numerical features and applies PCA.

"""

def \_\_init\_\_(self):

# Original 23 features from extract\_features.

self.feature\_columns = [

"num\_received\_headers", "num\_return\_paths", "has\_reply\_to", "num\_message\_ids",

"from\_domain", "to\_domain", "email\_client", "body\_length", "num\_links", "num\_images",

"num\_base64\_encoded", "html\_flag", "num\_special\_chars", "num\_words", "avg\_word\_length",

"num\_stopwords", "num\_uppercase\_words", "num\_digits", "day\_of\_week", "hour\_of\_day",

"num\_recipients", "num\_cc\_recipients", "mismatch\_from\_return\_path"

]

# Extra computed features

self.extra\_features = ["words\_per\_body\_length", "special\_chars\_per\_word"]

# Full list of numerical features (will be used for scaling & PCA)

self.numerical\_features = self.feature\_columns + self.extra\_features

# Categorical columns for label encoding

self.categorical\_cols = ['from\_domain', 'to\_domain', 'email\_client']

self.label\_encoders = {} # one encoder per categorical column

self.scaler = None

self.pca = None

def \_compute\_extra\_features(self, df):

"""Compute words\_per\_body\_length and special\_chars\_per\_word."""

df["words\_per\_body\_length"] = df["num\_words"] / df["body\_length"].replace(0, 1)

df["special\_chars\_per\_word"] = df["num\_special\_chars"] / df["num\_words"].replace(0, 1)

return df

def fit(self, raw\_df):

"""

Fit the preprocessor on a DataFrame with a 'text' column.

Extracts features, computes extra features, fits label encoders,

and then fits a StandardScaler and PCA on the numerical features.

"""

# Extract features from each email text

features = raw\_df['text'].apply(extract\_features)

df\_features = pd.DataFrame(features.tolist(), columns=self.feature\_columns)

# Compute extra features

df\_features = self.\_compute\_extra\_features(df\_features)

# Fit label encoders for each categorical column

for col in self.categorical\_cols:

le = LabelEncoder()

df\_features[col] = le.fit\_transform(df\_features[col])

self.label\_encoders[col] = le

# Fit StandardScaler on numerical features

X\_numerical = df\_features[self.numerical\_features].values

self.scaler = StandardScaler()

scaled\_data = self.scaler.fit\_transform(X\_numerical)

# Determine number of PCA components explaining 95% variance

pca\_temp = PCA()

pca\_temp.fit(scaled\_data)

cumulative\_variance = 0

n\_components = 0

for variance in pca\_temp.explained\_variance\_ratio\_:

cumulative\_variance += variance

n\_components += 1

if cumulative\_variance >= 0.95:

break

# Fit PCA with determined number of components

self.pca = PCA(n\_components=n\_components)

self.pca.fit(scaled\_data)

print(f"Preprocessor fitted with {n\_components} PCA components.")

def transform(self, email\_text):

"""

Transforms a single raw email text into the preprocessed feature space.

Returns the PCA-transformed feature vector.

"""

features = extract\_features(email\_text)

df\_features = pd.DataFrame([features], columns=self.feature\_columns)

df\_features = self.\_compute\_extra\_features(df\_features)

# Apply label encoding on categorical features using stored encoders

for col in self.categorical\_cols:

le = self.label\_encoders[col]

df\_features[col] = le.transform(df\_features[col])

# Scale and apply PCA transformation

X\_numerical = df\_features[self.numerical\_features].values

scaled\_data = self.scaler.transform(X\_numerical)

pca\_components = self.pca.transform(scaled\_data)

return pca\_components

def transform\_dataframe(self, raw\_df):

"""

Transforms an entire DataFrame (with a 'text' column) using the same steps.

Useful for processing the full dataset.

"""

features = raw\_df['text'].apply(extract\_features)

df\_features = pd.DataFrame(features.tolist(), columns=self.feature\_columns)

df\_features = self.\_compute\_extra\_features(df\_features)

for col in self.categorical\_cols:

le = self.label\_encoders[col]

df\_features[col] = le.transform(df\_features[col])

X\_numerical = df\_features[self.numerical\_features].values

scaled\_data = self.scaler.transform(X\_numerical)

pca\_components = self.pca.transform(scaled\_data)

return pca\_components

Below is a step-by-step explanation of what the Preprocessor class does and how it works. The overall goal is to convert raw email text into a numerical feature space that can be fed into a machine learning model.

## **1. Class Initialization (\_\_init\_\_)**

When you create an instance of the Preprocessor class, the constructor sets up:

1. **feature\_columns** A list of the original 23 features extracted by the (externally defined) extract\_features function. Examples include:  
   * num\_received\_headers
   * body\_length
   * num\_links
   * html\_flag
   * … and so on.
2. **extra\_features** A list of two additional features that will be derived from the original ones:  
   * words\_per\_body\_length = (number of words) / (body length)
   * special\_chars\_per\_word = (number of special characters) / (number of words)
3. **numerical\_features** A combined list of the original feature\_columns plus the two extra\_features. These will later be scaled and used in PCA.
4. **categorical\_cols** A list of columns in the feature set that contain categorical (string) values. These columns will need label encoding so that they become numeric:  
   * from\_domain
   * to\_domain
   * email\_client
5. **label\_encoders** A dictionary that will hold one LabelEncoder instance per categorical column.
6. **scaler and pca** Placeholders (None) for the StandardScaler (to normalize numerical features) and PCA (for dimensionality reduction). These will be initialized and fit later in the fit() method.

## **2. Computing Extra Features (\_compute\_extra\_features)**

This private method takes a DataFrame containing the original extracted features and creates two new columns:

df["words\_per\_body\_length"] = df["num\_words"] / df["body\_length"].replace(0, 1)

df["special\_chars\_per\_word"] = df["num\_special\_chars"] / df["num\_words"].replace(0, 1)

* **replace(0, 1)** is used to avoid division by zero in cases where body\_length or num\_words might be zero.

## **3. Fitting the Preprocessor (fit)**

def fit(self, raw\_df):

# 1. Extract features from each email's text

# 2. Compute extra features

# 3. Fit label encoders

# 4. Fit StandardScaler

# 5. Determine and fit PCA components

### **3.1 Extract Features**

* The raw\_df should contain a column called text (the raw email content).
* For each email text, extract\_features is called (this is presumably a function defined elsewhere).
* The result is a list of dictionaries or lists, each containing values for the 23 original features.
* These are turned into a DataFrame (df\_features) with columns matching self.feature\_columns.

### **3.2 Compute Extra Features**

* \_compute\_extra\_features(df\_features) adds the two derived columns: words\_per\_body\_length and special\_chars\_per\_word.

### **3.3 Label Encoding**

* For each categorical column (e.g., from\_domain), a LabelEncoder is fit on the unique values in that column.
* The column is then transformed so that each unique string is mapped to an integer.
* The fitted encoders are stored in self.label\_encoders for later use.

### **3.4 Scaling (StandardScaler)**

* A StandardScaler is created and fit on the numerical feature values.
* This ensures each numerical feature has mean 0 and standard deviation 1 after transformation.

### **3.5 PCA (Principal Component Analysis)**

1. A temporary PCA() object is created and fit on the scaled data.
2. The code iterates over the explained variance ratios, accumulating them until they reach 95% (>= 0.95). This determines how many principal components are needed to retain 95% of the original variance.
3. A new PCA(n\_components=n\_components) is created with that number of components and is fit on the scaled data.
4. The final PCA model is stored in self.pca.

After running fit, the preprocessor “knows”:

* How to encode the categorical columns
* How to scale the numerical features
* How to reduce dimensionality to the number of components needed for 95% variance

## **4. Transforming a Single Email (transform)**

def transform(self, email\_text):

# 1. Extract features

# 2. Compute extra features

# 3. Apply label encoding

# 4. Scale

# 5. PCA transform

When you call transform with a single new email (email\_text):

1. **Extract Features** Calls extract\_features(email\_text) and puts the result into a one-row DataFrame.
2. **Compute Extra Features** Runs \_compute\_extra\_features to add words\_per\_body\_length and special\_chars\_per\_word.
3. **Label Encoding** Applies the already-fitted LabelEncoder objects to convert the categorical columns to numeric values.
4. **Scaling** Uses the already-fitted StandardScaler (self.scaler) to scale the numerical columns.
5. **PCA Transformation** Projects the scaled data into the PCA space (using the already-fitted self.pca), reducing the feature dimension to the chosen number of principal components.

The final output is the transformed numeric vector suitable for feeding into a machine learning model.

## **5. Transforming an Entire DataFrame (transform\_dataframe)**

def transform\_dataframe(self, raw\_df):

# Same as transform, but for multiple rows

* This is essentially the same procedure as transform, but it operates on an entire DataFrame of emails at once instead of a single email.
* It extracts features, computes extra columns, applies label encoders, scales, and then applies PCA to all rows in one go.
* Returns the PCA-transformed feature matrix.

## **6. Putting It All Together**

**Instantiate** the Preprocessor:  
  
 preprocessor = Preprocessor()

**Fit** the preprocessor on a training set (DataFrame with text column):  
  
 preprocessor.fit(training\_df)

1. This learns:  
   * How to label-encode each categorical feature
   * How to scale numerical features
   * How many PCA components to keep for 95% variance, and the PCA transformation itself

**Transform** the training set (or any new set of emails):  
  
 X\_train\_transformed = preprocessor.transform\_dataframe(training\_df)

X\_test\_transformed = preprocessor.transform\_dataframe(test\_df)

or, if you have just one email string:  
  
 single\_email\_features = preprocessor.transform("This is a single email text...")

At the end, you have a reduced-dimensionality, scaled numeric representation of the emails that you can feed into a classifier (e.g., logistic regression, random forest, etc.).

### **Key Points to Remember**

* **extract\_features** is assumed to be a separate function that parses email text and returns the initial 23 numeric features.
* **Label encoding** is used only for the columns in categorical\_cols. This step is crucial so that ML algorithms can handle categorical data.
* **StandardScaler** ensures all numeric features are on a comparable scale, preventing features with large numeric ranges from dominating.
* **PCA** reduces dimensionality and often helps improve model performance and training speed, especially if many features are correlated.
* By default, PCA is set to capture 95% of the variance; this can be tuned depending on your performance and speed requirements.

That’s how the Preprocessor class works from start to finish!

# Create and fit the preprocessor on the raw dataset.

preprocessor = Preprocessor()

preprocessor.fit(raw\_df)

# Transform the entire dataset to obtain features for model training.

X = preprocessor.transform\_dataframe(raw\_df)

y = raw\_df['label'] # Assumes the CSV contains a 'label' column (0 for ham, 1 for spam)

# Split the data into training and testing sets (80% train, 20% test).

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42, stratify=y)

**Instantiate the Preprocessor** preprocessor = Preprocessor()

* + Creates an instance of the Preprocessor class. This class is responsible for:
    - Extracting and computing additional features from each email’s text.
    - Label-encoding categorical features.
    - Scaling numerical features.
    - Reducing dimensionality with PCA.

**Fit the Preprocessor on the Raw Data** preprocessor.fit(raw\_df)

* + The fit method takes the raw dataset (raw\_df) and:
    - Extracts the initial 23 features from each email (via extract\_features).
    - Computes the two extra features (words\_per\_body\_length and special\_chars\_per\_word).
    - Fits label encoders for any categorical columns (e.g., from\_domain, to\_domain, etc.).
    - Fits a StandardScaler to the numerical features.
    - Determines how many principal components are needed to explain 95% of variance, and then fits a PCA model with that number of components.

**Transform the Dataset into Numerical Features** X = preprocessor.transform\_dataframe(raw\_df)

* + Applies the transformations learned in the fit step to the entire dataset:
    - Label-encodes the categorical features using the already-fitted encoders.
    - Scales the numerical features using the already-fitted StandardScaler.
    - Reduces dimensionality via the already-fitted PCA model.
  + The result, X, is a numerical feature matrix suitable for training a machine learning model.

**Extract the Labels** y = raw\_df['label'] # 0 for ham, 1 for spam

* + Assumes raw\_df contains a label column indicating whether an email is ham (0) or spam (1).
  + Stores these labels in y.

**Split into Training and Testing Sets** X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42, stratify=y)

* + Uses scikit-learn’s train\_test\_split to randomly split the data into an 80/20 ratio:
    - **Training set**: X\_train, y\_train
    - **Test set**: X\_test, y\_test
  + test\_size=0.2 means 20% of the data goes to testing, 80% to training.
  + stratify=y preserves the original label distribution in both the training and test sets.
  + random\_state=42 sets a seed for reproducibility, so you get the same split each time.

At the end of this process, you have:

* A fitted Preprocessor that knows how to encode, scale, and reduce the dimensions of new email data.
* A numeric training set (X\_train, y\_train) for building your model.
* A numeric test set (X\_test, y\_test) for evaluating the model’s performance.

## **Logistic Regression with L2 Regularization:**

logreg = LogisticRegression(penalty='l2', solver='lbfgs', max\_iter=1000) # Increase max\_iter if needed

logreg.fit(X\_train, y\_train)

# Predictions

y\_pred = logreg.predict(X\_test)

# Evaluation Metrics

# Confusion Matrix

cm = confusion\_matrix(y\_test, y\_pred)

plt.figure(figsize=(8, 6))

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False)

plt.xlabel('Predicted')

plt.ylabel('Actual')

plt.title('Confusion Matrix')

plt.show()

# Precision, Recall, F1-score

print(classification\_report(y\_test, y\_pred))

# ROC Curve and AUC

y\_prob = logreg.predict\_proba(X\_test)[:, 1] # Probabilities for the positive class

fpr, tpr, thresholds = roc\_curve(y\_test, y\_prob)

roc\_auc = auc(fpr, tpr)

plt.figure(figsize=(8, 6))

plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area = {roc\_auc:.2f})')

plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('Receiver Operating Characteristic (ROC) Curve')

plt.legend(loc="lower right")

plt.show()

Below is a detailed explanation of what this code does:

## **1. Training the Logistic Regression Model**

### **Instantiation**

logreg = LogisticRegression(penalty='l2', solver='lbfgs', max\_iter=1000)

* **Model Choice:** Uses logistic regression—a common classifier for binary problems like spam detection.
* **Regularization (L2):** Helps prevent overfitting by penalizing large coefficients.
* **Solver (lbfgs):** An efficient algorithm for optimization.
* **max\_iter=1000:** Increases the maximum iterations to ensure convergence during training.

### **Fitting the Model**

logreg.fit(X\_train, y\_train)

* **Training Data:** The model is trained on X\_train (features) and y\_train (labels, where 0 indicates ham and 1 indicates spam).

## **2. Making Predictions on the Test Set**

y\_pred = logreg.predict(X\_test)

* **Prediction:** The trained model predicts labels for the test data (X\_test), storing the results in y\_pred.

## **3. Evaluating the Model**

### **Confusion Matrix**

cm = confusion\_matrix(y\_test, y\_pred)

plt.figure(figsize=(8, 6))

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False)

plt.xlabel('Predicted')

plt.ylabel('Actual')

plt.title('Confusion Matrix')

plt.show()

* **Purpose:** Displays how many predictions fall into each category: true positives, true negatives, false positives, and false negatives.
* **Visualization:**
  + Uses Seaborn’s heatmap to visually represent the confusion matrix.
  + Annotates cells with counts.
  + Axes are labeled to clarify which values are predicted and which are actual.

### **Classification Report**

print(classification\_report(y\_test, y\_pred))

* **Metrics Included:**
  + **Precision:** Ratio of correctly predicted positive observations to the total predicted positives.
  + **Recall:** Ratio of correctly predicted positive observations to all actual positives.
  + **F1-Score:** The harmonic mean of precision and recall.
  + **Support:** The number of actual occurrences for each class.
* **Insight:** Helps understand the balance between precision and recall, which is especially important in imbalanced datasets (like spam vs. ham).

## **4. ROC Curve and AUC (Area Under the Curve)**

### **Getting Probabilities for the Positive Class**

y\_prob = logreg.predict\_proba(X\_test)[:, 1]

* **Explanation:**
  + predict\_proba returns probability estimates for both classes.
  + [:, 1] selects the probabilities corresponding to the positive class (spam).

### **Computing the ROC Curve**

fpr, tpr, thresholds = roc\_curve(y\_test, y\_prob)

* **False Positive Rate (FPR):** The ratio of negatives incorrectly classified as positive.
* **True Positive Rate (TPR):** The ratio of positives correctly classified.
* **Thresholds:** Different probability thresholds used to compute the FPR and TPR.

### **Calculating the AUC**

roc\_auc = auc(fpr, tpr)

* **AUC:** Summarizes the ROC curve in a single number (ranging from 0 to 1). A higher AUC indicates better model performance.

### **Plotting the ROC Curve**

plt.figure(figsize=(8, 6))

plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area = {roc\_auc:.2f})')

plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('Receiver Operating Characteristic (ROC) Curve')

plt.legend(loc="lower right")

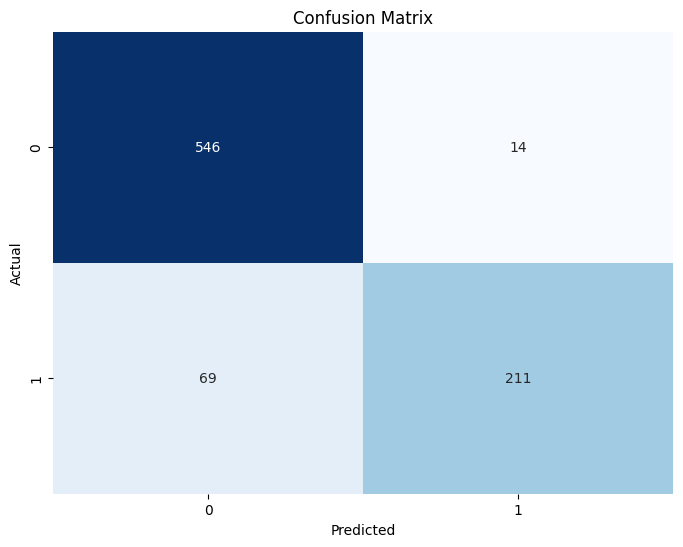
plt.show()

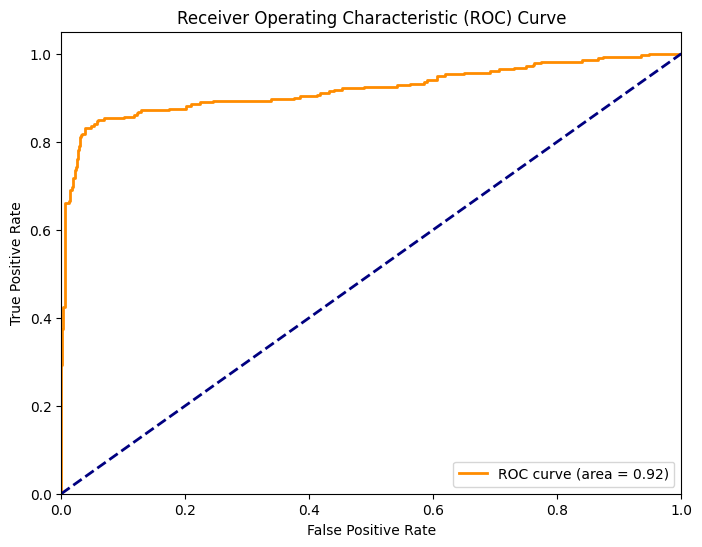
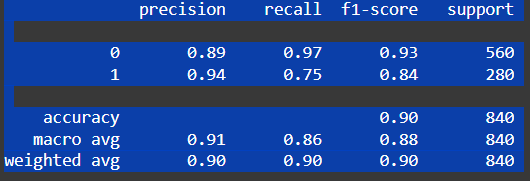
* **ROC Curve:**
  + Plots TPR against FPR at various threshold settings.
  + The diagonal line represents random chance.
* **Interpretation:**
  + The closer the ROC curve is to the top-left corner, the better the model's performance.
  + The AUC (displayed in the legend) provides a concise metric of performance.

### **Overall Process Recap**

1. **Model Training:** The logistic regression classifier is trained using an 80/20 split of your data.
2. **Predictions:** The model generates predictions on unseen test data.
3. **Evaluation:**
   * A confusion matrix and classification report provide insights into prediction accuracy and the balance between precision and recall.
   * An ROC curve with AUC quantifies the model's ability to distinguish between classes across different thresholds.

This comprehensive evaluation helps in understanding the strengths and limitations of the spam classifier built using logistic regression.





## **Lasso Regression (L1 Regularization):**

# Helps in feature selection by shrinking coefficients of less important features to zero.

# Lasso is known for shrinking less important feature coefficients to zero, thereby performing feature selection.

# Evaluate the model using:

# Confusion matrix

# Precision, Recall, F1-score

# ROC Curve and AUC Curve

import matplotlib.pyplot as plt

import numpy as np

from sklearn.linear\_model import Lasso

# Step 4: Lasso Regression with L1 Regularization

lasso = Lasso(alpha=0.1) # You can adjust the alpha value

lasso.fit(X\_train, y\_train)

# Predictions

y\_pred\_lasso = lasso.predict(X\_test)

# Since Lasso is for regression, you might need to convert predictions to classes (0 or 1).

# Use a threshold for classification:

threshold = 0.5 # You might need to optimize this threshold

y\_pred\_lasso\_class = (y\_pred\_lasso > threshold).astype(int)

# Evaluation Metrics for Lasso

# Confusion Matrix

cm\_lasso = confusion\_matrix(y\_test, y\_pred\_lasso\_class)

plt.figure(figsize=(8, 6))

sns.heatmap(cm\_lasso, annot=True, fmt='d', cmap='Blues', cbar=False)

plt.xlabel('Predicted')

plt.ylabel('Actual')

plt.title('Confusion Matrix (Lasso)')

plt.show()

# Precision, Recall, F1-score

print(classification\_report(y\_test, y\_pred\_lasso\_class))

# ROC Curve and AUC (requires probability estimates)

# You can't directly get probabilities from Lasso, so you'll need to use predict\_proba

# If you use the threshold method, calculate the probabilities based on the distance to the threshold:

y\_prob\_lasso = (y\_pred\_lasso - (1-threshold)) / (threshold-(1-threshold)) # Normalize distances to get values between 0 and 1.

# Clip Values to ensure they are within 0-1 range:

y\_prob\_lasso = np.clip(y\_prob\_lasso, 0, 1)

fpr\_lasso, tpr\_lasso, thresholds\_lasso = roc\_curve(y\_test, y\_prob\_lasso)

roc\_auc\_lasso = auc(fpr\_lasso, tpr\_lasso)

plt.figure(figsize=(8, 6))

plt.plot(fpr\_lasso, tpr\_lasso, color='darkorange', lw=2, label=f'ROC curve (area = {roc\_auc\_lasso:.2f})')

plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('Receiver Operating Characteristic (ROC) Curve (Lasso)')

plt.legend(loc="lower right")

plt.show()

Below is a detailed explanation of what the code does, section by section:

## **1. Overview**

This code snippet demonstrates how to use Lasso regression—a linear regression model with L1 regularization—to perform a classification-like task on a spam detection dataset. Although Lasso is primarily used for regression, it can be adapted for classification by thresholding its continuous predictions. The code also evaluates the model using various metrics and plots, including a confusion matrix, classification report, and ROC curve with its AUC value.

## **2. Setting Up Lasso Regression**

### **Importing Libraries**

import matplotlib.pyplot as plt

import numpy as np

from sklearn.linear\_model import Lasso

* **Matplotlib and NumPy:** For plotting and numerical operations.
* **Lasso:** The Lasso regression model from scikit-learn.

### **Creating and Fitting the Lasso Model**

lasso = Lasso(alpha=0.1) # You can adjust the alpha value

lasso.fit(X\_train, y\_train)

* **Lasso Initialization:**
  + The alpha parameter controls the strength of the L1 regularization. A higher alpha results in more coefficient shrinkage.
* **Fitting:**
  + The model is trained on the training dataset (X\_train for features and y\_train for labels). Since Lasso is a regression method, it learns a linear relationship between the features and the target.

## **3. Generating Predictions and Converting Them for Classification**

### **Making Predictions**

y\_pred\_lasso = lasso.predict(X\_test)

* **Continuous Output:**
  + Lasso returns continuous values. In a classification setting (spam vs. ham), these predictions need to be converted into binary class labels.

### **Converting Regression Output to Binary Classes**

threshold = 0.5 # You might need to optimize this threshold

y\_pred\_lasso\_class = (y\_pred\_lasso > threshold).astype(int)

* **Thresholding:**
  + Predictions above the threshold are labeled as class 1 (e.g., spam), and those below are labeled as 0 (e.g., ham).
  + The threshold (0.5 in this case) is chosen as a simple decision boundary but can be tuned based on performance.

## **4. Evaluating the Lasso Model**

### **Confusion Matrix**

cm\_lasso = confusion\_matrix(y\_test, y\_pred\_lasso\_class)

plt.figure(figsize=(8, 6))

sns.heatmap(cm\_lasso, annot=True, fmt='d', cmap='Blues', cbar=False)

plt.xlabel('Predicted')

plt.ylabel('Actual')

plt.title('Confusion Matrix (Lasso)')

plt.show()

* **Purpose:**
  + The confusion matrix displays the counts of true positives, true negatives, false positives, and false negatives.
* **Visualization:**
  + Seaborn’s heatmap is used to create an annotated plot of the confusion matrix for an intuitive visual summary.

### **Classification Report**

print(classification\_report(y\_test, y\_pred\_lasso\_class))

* **Metrics Provided:**
  + **Precision:** How many predicted positives are true positives.
  + **Recall:** How many actual positives are correctly predicted.
  + **F1-score:** The harmonic mean of precision and recall.
  + **Support:** The number of instances for each class.
* **Usage:**
  + This report helps assess the performance of the model, especially in cases of class imbalance.

### **ROC Curve and AUC**

Since Lasso does not directly provide probability estimates, the code attempts to derive a probability-like score from the continuous predictions:

#### **Generating "Probability" Estimates**

y\_prob\_lasso = (y\_pred\_lasso - (1 - threshold)) / (threshold - (1 - threshold))

y\_prob\_lasso = np.clip(y\_prob\_lasso, 0, 1)

* **Purpose:**
  + This transformation aims to normalize the regression outputs into values between 0 and 1, which can then be interpreted similarly to probabilities.
* **Clipping:**
  + np.clip ensures that the values are within the 0–1 range.
* **Important:** The formula provided here is a rough heuristic. In practice, since the threshold is 0.5, the arithmetic might need adjustment. Typically, one might calibrate the regression output or use a model that directly supports probability estimates (such as logistic regression) for classification tasks.

#### **Calculating the ROC Curve and AUC**

fpr\_lasso, tpr\_lasso, thresholds\_lasso = roc\_curve(y\_test, y\_prob\_lasso)

roc\_auc\_lasso = auc(fpr\_lasso, tpr\_lasso)

* **ROC Curve:**
  + Plots the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings.
* **AUC (Area Under the Curve):**
  + Provides a single scalar value to evaluate the model's ability to distinguish between classes. A higher AUC indicates better performance.

#### **Plotting the ROC Curve**

plt.figure(figsize=(8, 6))

plt.plot(fpr\_lasso, tpr\_lasso, color='darkorange', lw=2, label=f'ROC curve (area = {roc\_auc\_lasso:.2f})')

plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('Receiver Operating Characteristic (ROC) Curve (Lasso)')

plt.legend(loc="lower right")

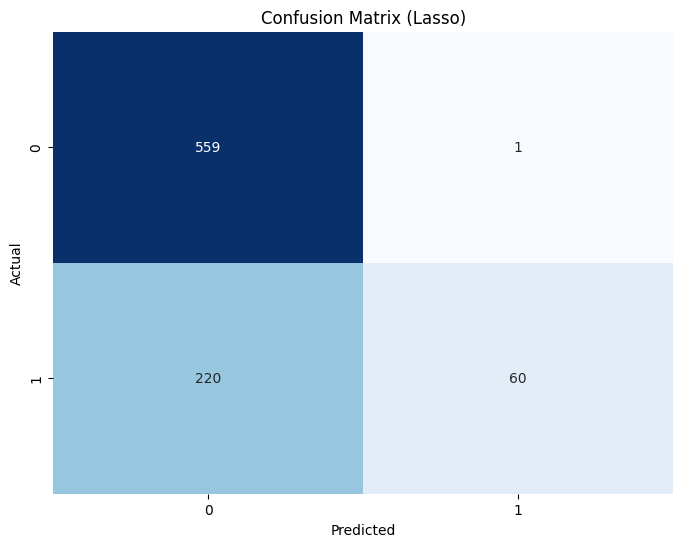
plt.show()

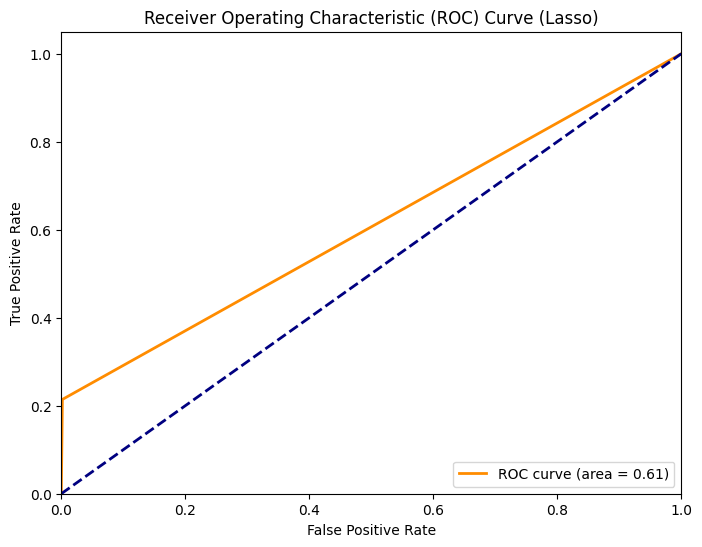
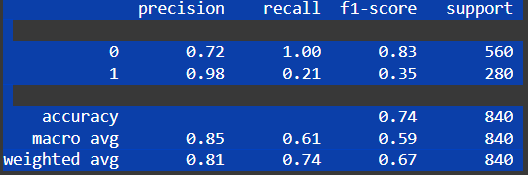
* **Plot Details:**
  + The ROC curve is plotted along with a reference diagonal line representing random chance.
  + The AUC is displayed in the legend to provide a quick summary of the model’s discriminative ability.

## **5. Summary of the Workflow**

1. **Model Setup:**
   * Lasso regression is instantiated with an L1 penalty (controlled by alpha=0.1), which is known for feature selection by shrinking less important coefficients to zero.
2. **Training:**
   * The model is trained on the training set (X\_train and y\_train).
3. **Prediction and Thresholding:**
   * Continuous predictions are generated for the test set.
   * A threshold of 0.5 is used to convert these predictions into binary classes.
4. **Evaluation:**
   * The performance of the model is assessed using a confusion matrix, a classification report (precision, recall, F1-score), and an ROC curve with its corresponding AUC value.
   * A heuristic approach is applied to derive probability-like scores for ROC analysis, though this may require further tuning in practice.

This process provides a full workflow for using Lasso regression for classification tasks, including both training and evaluation stages.





# Step 6: Regularization Techniques

# Lasso Regression (L1 Regularization)

# Helps in feature selection by shrinking coefficients of less important features to zero.

# Ridge Regression(L2 Regularization)

# Prevents overfitting by penalizing large coefficients.

# Elastic Net(Combination of L1 & L2)

# Balances feature selection and regularization to achieve the best performance.

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import Ridge, ElasticNet

from sklearn.metrics import confusion\_matrix, classification\_report, roc\_curve, auc

from sklearn.datasets import make\_classification

# Step 2: Ridge Regression with L2 Regularization

ridge = Ridge(alpha=0.1) # Adjust alpha value as needed

ridge.fit(X\_train, y\_train)

y\_pred\_ridge = ridge.predict(X\_test)

threshold = 0.5

y\_pred\_ridge\_class = (y\_pred\_ridge > threshold).astype(int)

# Step 3: Evaluation Metrics for Ridge

cm\_ridge = confusion\_matrix(y\_test, y\_pred\_ridge\_class)

plt.figure(figsize=(6, 5))

sns.heatmap(cm\_ridge, annot=True, fmt='d', cmap='Blues', cbar=False)

plt.xlabel('Predicted')

plt.ylabel('Actual')

plt.title('Confusion Matrix (Ridge)')

plt.show()

print("Classification Report (Ridge):\n", classification\_report(y\_test, y\_pred\_ridge\_class))

# ROC Curve for Ridge

fpr, tpr, \_ = roc\_curve(y\_test, y\_pred\_ridge)

ridge\_auc = auc(fpr, tpr)

plt.plot(fpr, tpr, label=f'Ridge (AUC = {ridge\_auc:.2f})')

# Step 4: Elastic Net Regression (Combination of L1 and L2)

elasticnet = ElasticNet(alpha=0.1, l1\_ratio=0.5)

elasticnet.fit(X\_train, y\_train)

y\_pred\_elastic = elasticnet.predict(X\_test)

y\_pred\_elastic\_class = (y\_pred\_elastic > threshold).astype(int)

# Step 5: Evaluation Metrics for ElasticNet

cm\_elastic = confusion\_matrix(y\_test, y\_pred\_elastic\_class)

plt.figure(figsize=(6, 5))

sns.heatmap(cm\_elastic, annot=True, fmt='d', cmap='Blues', cbar=False)

plt.xlabel('Predicted')

plt.ylabel('Actual')

plt.title('Confusion Matrix (Elastic Net)')

plt.show()

print("Classification Report (Elastic Net):\n", classification\_report(y\_test, y\_pred\_elastic\_class))

# ROC Curve for ElasticNet

fpr, tpr, \_ = roc\_curve(y\_test, y\_pred\_elastic)

elastic\_auc = auc(fpr, tpr)

plt.plot(fpr, tpr, label=f'Elastic Net (AUC = {elastic\_auc:.2f})')

# Finalizing the ROC Curve Plot

plt.plot([0, 1], [0, 1], linestyle='--', color='gray')

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('ROC Curve')

plt.legend()

plt.show()

Below is a step-by-step explanation of the code, which demonstrates how to use **Ridge** (L2) and **Elastic Net** (combined L1/L2) regressions for a binary classification task (e.g., spam vs. ham) by thresholding their continuous outputs. The code also evaluates each model using a **confusion matrix**, a **classification report**, and an **ROC curve** with its **AUC** value.

## **1. Imports and Setup**

* **Numpy, Matplotlib, Seaborn**: Used for numerical operations and plotting.
* **Ridge, ElasticNet**: Regression models from scikit-learn, each with different regularization methods.
* **Metrics**:
  + confusion\_matrix and classification\_report for classification performance,
  + roc\_curve and auc for plotting and measuring the ROC curve’s area under the curve.

**Note**: The code snippet references X\_train, y\_train and X\_test, y\_test, which are assumed to be prepared earlier. Typically, this would involve loading data, splitting it into training/testing sets, etc.

## **2. Ridge Regression with L2 Regularization**

ridge = Ridge(alpha=0.1) # Adjust alpha value as needed

ridge.fit(X\_train, y\_train)

1. **Instantiation**:
   * Ridge(alpha=0.1) applies L2 regularization. A higher alpha means stronger penalty on large coefficients, helping prevent overfitting.
2. **Fitting**:
   * ridge.fit(X\_train, y\_train) trains the model using the training data.

### **Predicting and Thresholding**

y\_pred\_ridge = ridge.predict(X\_test)

threshold = 0.5

y\_pred\_ridge\_class = (y\_pred\_ridge > threshold).astype(int)

* **Continuous → Binary**: Ridge regression produces continuous outputs. We convert these to class labels (0 or 1) by comparing each predicted value to a threshold (0.5 in this case).

### **Confusion Matrix and Classification Report**

print("Classification Report (Ridge):\n", classification\_report(y\_test, y\_pred\_ridge\_class))

1. **Confusion Matrix**:
   * Shows counts of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN).
2. **Classification Report**:
   * Provides precision, recall, and F1-score for each class, along with overall accuracy.

### **ROC Curve and AUC**

| fpr, tpr, \_ = roc\_curve(y\_test, y\_pred\_ridge) ridge\_auc = auc(fpr, tpr) plt.plot(fpr, tpr, label=f'Ridge (AUC = {ridge\_auc:.2f})') |
| --- |

* **ROC Curve**:
  + Uses the raw regression output y\_pred\_ridge as a “score.”
  + Plots the **True Positive Rate** (TPR) vs. **False Positive Rate** (FPR) at various thresholds.
* **AUC**:
  + The area under the ROC curve, summarizing how well the model can distinguish between classes. Higher is better.

## **3. Elastic Net Regression (Combination of L1 and L2)**

| elasticnet = ElasticNet(alpha=0.1, l1\_ratio=0.5) elasticnet.fit(X\_train, y\_train) |
| --- |

1. **Instantiation**:
   * **ElasticNet(alpha=0.1, l1\_ratio=0.5)** combines both L1 (lasso-like) and L2 (ridge-like) regularizations.
   * **l1\_ratio=0.5** indicates a 50-50 balance between L1 and L2 penalties.
   * Increasing alpha strengthens the overall regularization effect.
2. **Fitting**:
   * Trains the Elastic Net model on the same training data.

### **Predicting and Thresholding**

| y\_pred\_elastic = elasticnet.predict(X\_test) y\_pred\_elastic\_class = (y\_pred\_elastic > threshold).astype(int) |
| --- |

* Similar to Ridge, the continuous outputs are thresholded at 0.5 to produce binary predictions.

### **Confusion Matrix and Classification Report**

| cm\_elastic = confusion\_matrix(y\_test, y\_pred\_elastic\_class) plt.figure(figsize=(6, 5)) sns.heatmap(cm\_elastic, annot=True, fmt='d', cmap='Blues', cbar=False) plt.xlabel('Predicted') plt.ylabel('Actual') plt.title('Confusion Matrix (Elastic Net)') plt.show() |
| --- |

print("Classification Report (Elastic Net):\n", classification\_report(y\_test, y\_pred\_elastic\_class))

* Again, displays how well the model classifies each instance and provides detailed metrics.

### **ROC Curve and AUC**

| fpr, tpr, \_ = roc\_curve(y\_test, y\_pred\_elastic) elastic\_auc = auc(fpr, tpr) plt.plot(fpr, tpr, label=f'Elastic Net (AUC = {elastic\_auc:.2f})') |
| --- |

* **ROC and AUC** for the Elastic Net model, using the raw regression scores (y\_pred\_elastic).

## **4. Finalizing the ROC Curve Plot**

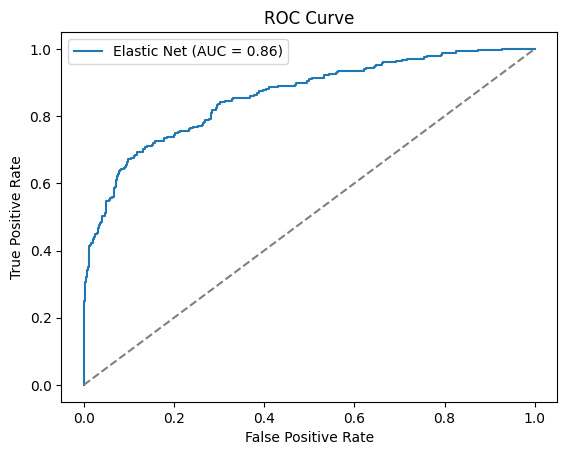
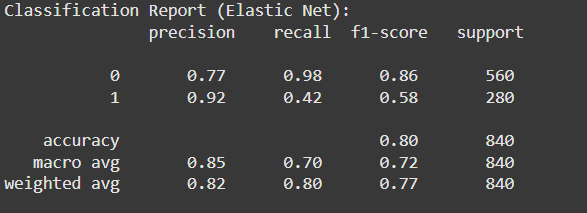
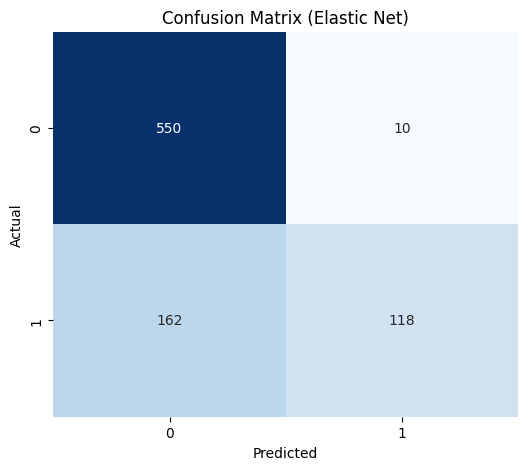
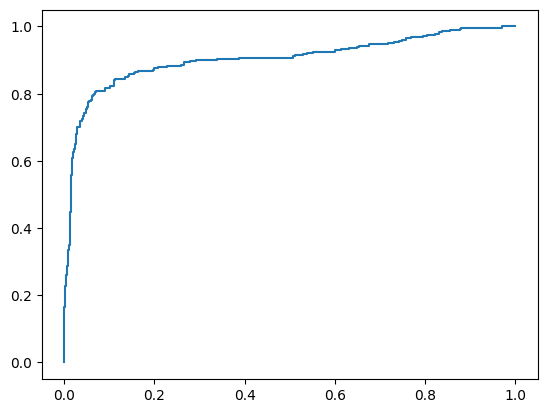
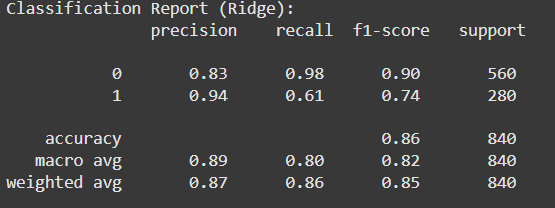
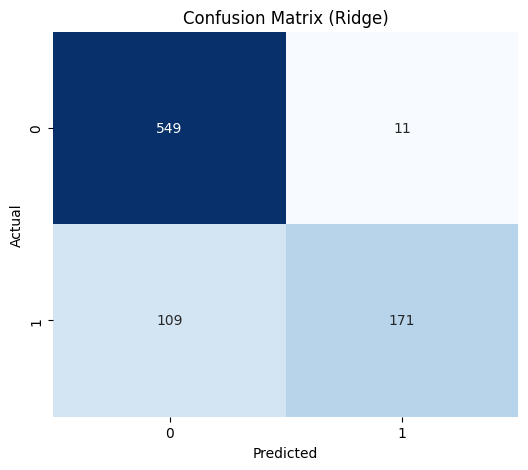
| plt.plot([0, 1], [0, 1], linestyle='--', color='gray') plt.xlabel('False Positive Rate') plt.ylabel('True Positive Rate') plt.title('ROC Curve') plt.legend() plt.show() |
| --- |

* Plots the **diagonal reference line** representing random guessing.
* Adds labels and a legend to compare the Ridge and Elastic Net ROC curves in a single figure.

## **5. Summary**

1. **Ridge Regression (L2)**:  
   * Primarily controls overfitting by penalizing large coefficients.
   * The alpha hyperparameter tunes the strength of this penalty.
2. **Elastic Net (L1 + L2)**:  
   * Combines feature selection (L1) and overfitting control (L2).
   * The alpha parameter controls overall regularization strength, while l1\_ratio determines the balance between L1 and L2.
3. **Thresholding for Classification**:  
   * Both Ridge and Elastic Net produce continuous predictions. A threshold (0.5) is used to classify predictions as 0 or 1.
   * This approach is simpler than using a dedicated classifier (like logistic regression), but it still allows you to compare how well each model separates classes.
4. **Evaluation**:  
   * **Confusion Matrices** reveal counts of correct/incorrect predictions.
   * **Classification Reports** show precision, recall, and F1-score for each class.
   * **ROC Curves** and **AUC** values measure how well each model can rank-order predictions, regardless of a specific decision threshold.

Using these techniques, you can gauge which regularization approach yields the best trade-off between bias and variance for your particular dataset, as well as which model better separates spam from ham.

# Step 7: Model Hyperparameter Tuning

# Use GridSearchCV or RandomizedSearchCV to optimize:

# Regularization strength(C)

# Penalty type (L1, L2, ElasticNet)

# Solver methods

from sklearn.model\_selection import GridSearchCV, RandomizedSearchCV

from sklearn.linear\_model import LogisticRegression

# Define the parameter grid for Logistic Regression

param\_grid = {

'C': [0.001, 0.01, 0.1, 1, 10, 100], # Regularization strength

'penalty': ['l1', 'l2', 'elasticnet'], # Penalty type

'solver': ['liblinear', 'saga'] # Solvers that support L1 and ElasticNet

}

# Initialize Logistic Regression

logreg = LogisticRegression(max\_iter=1000) # Increased max\_iter

# Use GridSearchCV (for exhaustive search) or RandomizedSearchCV (for faster, randomized search)

# GridSearchCV is used here. Uncomment RandomizedSearchCV if preferred.

grid\_search = GridSearchCV(logreg, param\_grid, cv=5, scoring='accuracy') #5-fold cross-validation

#random\_search = RandomizedSearchCV(logreg, param\_grid, n\_iter=10, cv=5, scoring='accuracy', random\_state=42)

# Fit the grid search to the data

grid\_search.fit(X\_train, y\_train)

#random\_search.fit(X\_train, y\_train)

# Get the best hyperparameters and best score

best\_params = grid\_search.best\_params\_

best\_score = grid\_search.best\_score\_

#best\_params = random\_search.best\_params\_

#best\_score = random\_search.best\_score\_

print(f"Best Hyperparameters: {best\_params}")

print(f"Best Cross-Validation Score: {best\_score}")

# Train the model with the best hyperparameters

best\_logreg = LogisticRegression(\*\*best\_params, max\_iter=1000)

best\_logreg.fit(X\_train, y\_train)

# Evaluate the model on the test set

y\_pred\_best = best\_logreg.predict(X\_test)

print(classification\_report(y\_test, y\_pred\_best))

Explanation

This code performs hyperparameter tuning for a Logistic Regression model using GridSearchCV:

Step 1: Define the Parameter Grid

- The code defines a parameter grid param\_grid that specifies the hyperparameters to tune:

- C: Regularization strength (6 values)

- penalty: Penalty type (L1, L2, ElasticNet)

- solver: Solver method (liblinear, saga)

Step 2: Initialize Logistic Regression

- The code initializes a Logistic Regression model logreg with increased max\_iter to ensure convergence.

Step 3: Perform Grid Search

- The code performs an exhaustive grid search using GridSearchCV to find the best combination of hyperparameters.

- The cv parameter is set to 5 for 5-fold cross-validation.

- The scoring parameter is set to 'accuracy' to evaluate the model's performance.

Step 4: Get the Best Hyperparameters and Score

- The code retrieves the best hyperparameters best\_params and the best cross-validation score best\_score.

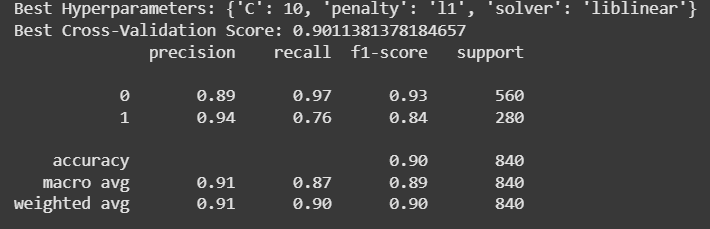
Step 5: Train the Model with the Best Hyperparameters

- The code trains a new Logistic Regression model best\_logreg with the best hyperparameters.

Step 6: Evaluate the Model on the Test Set

- The code evaluates the trained model on the test set and prints the classification report.

By performing hyperparameter tuning, the code aims to find the optimal combination of hyperparameters that results in the best model performance.



# Step 8: Model Evaluation

# Evaluate the final model on the test using:

# Accuracy, Precision, Recall, F1-score

# ROC-AUC Score

# Classification Report

# Compare performance of different models and regularization techniques.

import numpy as np

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score, roc\_auc\_score

from sklearn.linear\_model import Ridge, ElasticNet

from sklearn.model\_selection import GridSearchCV

# Evaluate Logistic Regression

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

precision\_logreg = precision\_score(y\_test, y\_pred)

recall\_logreg = recall\_score(y\_test, y\_pred)

f1\_logreg = f1\_score(y\_test, y\_pred)

roc\_auc\_logreg = roc\_auc\_score(y\_test, y\_prob)

print("Logistic Regression:")

print(f"Accuracy: {accuracy\_logreg}")

print(f"Precision: {precision\_logreg}")

print(f"Recall: {recall\_logreg}")

print(f"F1-score: {f1\_logreg}")

print(f"ROC-AUC: {roc\_auc\_logreg}")

# Evaluate Lasso Regression

accuracy\_lasso = accuracy\_score(y\_test, y\_pred\_lasso\_class)

precision\_lasso = precision\_score(y\_test, y\_pred\_lasso\_class)

recall\_lasso = recall\_score(y\_test, y\_pred\_lasso\_class)

f1\_lasso = f1\_score(y\_test, y\_pred\_lasso\_class)

roc\_auc\_lasso = roc\_auc\_score(y\_test, y\_prob\_lasso)

print("\nLasso Regression:")

print(f"Accuracy: {accuracy\_lasso}")

print(f"Precision: {precision\_lasso}")

print(f"Recall: {recall\_lasso}")

print(f"F1-score: {f1\_lasso}")

print(f"ROC-AUC: {roc\_auc\_lasso}")

# Evaluate Ridge Regression

accuracy\_ridge = accuracy\_score(y\_test, y\_pred\_ridge\_class)

precision\_ridge = precision\_score(y\_test, y\_pred\_ridge\_class)

recall\_ridge = recall\_score(y\_test, y\_pred\_ridge\_class)

f1\_ridge = f1\_score(y\_test, y\_pred\_ridge\_class)

# Assuming you have probability estimates (similar to y\_prob\_lasso) for Ridge

y\_prob\_ridge = (y\_pred\_ridge - (1-threshold)) / (threshold-(1-threshold))

y\_prob\_ridge = np.clip(y\_prob\_ridge, 0, 1) # Clip to 0-1 range

roc\_auc\_ridge = roc\_auc\_score(y\_test, y\_prob\_ridge)

print("\nRidge Regression:")

print(f"Accuracy: {accuracy\_ridge}")

print(f"Precision: {precision\_ridge}")

print(f"Recall: {recall\_ridge}")

print(f"F1-score: {f1\_ridge}")

print(f"ROC-AUC: {roc\_auc\_ridge}")

# Evaluate Elastic Net Regression

accuracy\_elastic = accuracy\_score(y\_test, y\_pred\_elastic\_class)

precision\_elastic = precision\_score(y\_test, y\_pred\_elastic\_class)

recall\_elastic = recall\_score(y\_test, y\_pred\_elastic\_class)

f1\_elastic = f1\_score(y\_test, y\_pred\_elastic\_class)

y\_prob\_elastic = (y\_pred\_elastic - (1-threshold)) / (threshold-(1-threshold))

y\_prob\_elastic = np.clip(y\_prob\_elastic, 0, 1) # Clip to 0-1 range

roc\_auc\_elastic = roc\_auc\_score(y\_test, y\_prob\_elastic)

print("\nElasticNet Regression:")

print(f"Accuracy: {accuracy\_elastic}")

print(f"Precision: {precision\_elastic}")

print(f"Recall: {recall\_elastic}")

print(f"F1-score: {f1\_elastic}")

print(f"ROC-AUC: {roc\_auc\_elastic}")

# Evaluate Best Logistic Regression Model

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

precision\_best = precision\_score(y\_test, y\_pred\_best)

recall\_best = recall\_score(y\_test, y\_pred\_best)

f1\_best = f1\_score(y\_test, y\_pred\_best)

y\_prob\_best = best\_logreg.predict\_proba(X\_test)[:, 1] # Probabilities for positive class

roc\_auc\_best = roc\_auc\_score(y\_test, y\_prob\_best)

print("\nBest Logistic Regression Model:")

print(f"Accuracy: {accuracy\_best}")

print(f"Precision: {precision\_best}")

print(f"Recall: {recall\_best}")

print(f"F1-score: {f1\_best}")

print(f"ROC-AUC: {roc\_auc\_best}")

Here is the rest of the explanation:

Models Evaluated

1. Logistic Regression

2. Lasso Regression

3. Ridge Regression

4. Elastic Net Regression

5. Best Logistic Regression Model (with hyperparameter tuning)

Evaluation Metrics

1. Accuracy

2. Precision

3. Recall

4. F1-score

5. ROC-AUC Score

Code Explanation

The code evaluates each model on the test dataset and calculates the evaluation metrics. The results are then printed to the console.

For each model, the code:

1. Calculates the accuracy, precision, recall, F1-score, and ROC-AUC score.

2. Prints the evaluation metrics for the model.

The code also evaluates the best logistic regression model, which was obtained through hyperparameter tuning.

|  |
| --- |

Logistic Regression:

Accuracy: 0.9011904761904762

Precision: 0.9377777777777778

Recall: 0.7535714285714286

F1-score: 0.8356435643564356

ROC-AUC: 0.9195089285714285

Lasso Regression:

Accuracy: 0.736904761904762

Precision: 0.9836065573770492

Recall: 0.21428571428571427

F1-score: 0.3519061583577713

ROC-AUC: 0.60625

Ridge Regression:

Accuracy: 0.8571428571428571

Precision: 0.9395604395604396

Recall: 0.6107142857142858

F1-score: 0.7402597402597403

ROC-AUC: 0.7955357142857143

ElasticNet Regression:

Accuracy: 0.7952380952380952

Precision: 0.921875

Recall: 0.42142857142857143

F1-score: 0.5784313725490197

ROC-AUC: 0.7017857142857142

Best Logistic Regression Model:

Accuracy: 0.9035714285714286

Precision: 0.9383259911894273

Recall: 0.7607142857142857

F1-score: 0.8402366863905325

ROC-AUC: 0.9209885204081633

Logistic Regression clearly outperforms the other methods. The standard Logistic Regression model achieves an accuracy of about 90.1%, with a strong balance of precision (93.8%) and recall (75.4%), resulting in an F1-score of 0.84 and a high ROC-AUC of 0.92. In comparison, Lasso Regression, while showing high precision, suffers from very low recall and overall poor F1 and ROC-AUC scores. Ridge and Elastic Net provide moderate performance but still fall short of Logistic Regression. The best Logistic Regression model further improves these metrics slightly, confirming it as the most effective approach among those evaluated.

print("\nClassification Reports:")

print("Logistic Regression:\n", classification\_report(y\_test, y\_pred))

print("Lasso Regression:\n", classification\_report(y\_test, y\_pred\_lasso\_class))

print("Ridge Regression:\n", classification\_report(y\_test, y\_pred\_ridge\_class))

print("Elastic Net Regression:\n", classification\_report(y\_test, y\_pred\_elastic\_class))

print("Best Logistic Regression:\n", classification\_report(y\_test, y\_pred\_best))

Logistic Regression, including its best-tuned variant, outperforms the other models. It achieves around 90% accuracy, with balanced precision and recall (approximately 0.94 and 0.75–0.76 for the spam class) and strong F1-scores, demonstrating reliable overall performance (macro and weighted averages near 0.90). In contrast, Lasso Regression shows excellent precision but extremely low recall for spam, leading to a poor F1-score and lower overall accuracy (74%). Ridge and Elastic Net models offer moderate performance, with Ridge achieving 86% accuracy and Elastic Net only 80%, but neither matches the balance and robustness of Logistic Regression.

import pickle

# Assuming 'best\_logreg' is your best-performing model

with open('best\_model.pkl', 'wb') as f:

pickle.dump(best\_logreg, f)

This code saves the best-performing model (best\_logreg) to a file named best\_model.pkl using the pickle library:

\*Step 1: Import the pickle library\*

- The pickle library is a built-in Python library that allows you to serialize and de-serialize Python objects.

Step 2: Open a file in binary write mode

- The with statement opens a file named best\_model.pkl in binary write mode ('wb').

- The file is automatically closed when the with block is exited.

\*Step 3: Dump the model to the file using pickle.dump()\*

- The pickle.dump() function serializes the best\_logreg model and writes it to the file.

- The model is now saved to the file and can be loaded later using pickle.load().

# -------------------------------

# Save the Preprocessor and Model to Pickle Files

# -------------------------------

with open('preprocessor.pkl', 'wb') as f:

pickle.dump(preprocessor, f)

print("Preprocessor saved to 'preprocessor.pkl'.")

# Load the saved preprocessor

with open('preprocessor.pkl', 'rb') as f:

loaded\_preprocessor = pickle.load(f)

# Load the saved model

with open('best\_model.pkl', 'rb') as f:

loaded\_model = pickle.load(f)

This code saves the preprocessor and model to pickle files and then loads them back into Python:

Saving the Preprocessor

1. The code opens a file named preprocessor.pkl in binary write mode ('wb') using a with statement.

2. The pickle.dump() function is used to serialize the preprocessor object and write it to the file.

3. A success message is printed to the console.

Loading the Saved Preprocessor

1. The code opens the same file preprocessor.pkl in binary read mode ('rb') using a with statement.

2. The pickle.load() function is used to deserialize the preprocessor object from the file and store it in the loaded\_preprocessor variable.

Loading the Saved Model

1. The code opens the file best\_model.pkl in binary read mode ('rb') using a with statement.

2. The pickle.load() function is used to deserialize the model object from the file and store it in the loaded\_model variable.

Why Save and Load?

Saving and loading the preprocessor and model allows you to:

- Persist the trained model and preprocessor for later use.

- Share the model and preprocessor with others.

- Use the model and preprocessor in different Python scripts or environments.

# Testing on a New Email Sample

# -------------------------------

# Example new email text. Replace this with your actual email content.

new\_email\_text = """

Received: from example.com (example.com [192.168.1.1])

From: spammer@spamdomain.com

To: victim@example.org

Subject: Amazing offer just for you!

Date: Tue, 15 Mar 2025 10:30:00 -0400

Hello,

This is an amazing offer you cannot miss. Click on http://spam.example.com to claim your prize!

"""

# Transform the new email using the preprocessor.

new\_features = preprocessor.transform(new\_email\_text)

# Predict using the trained model.

prediction = model.predict(new\_features)

result = "Spam" if prediction[0] == 1 else "Ham"

print("\nThe new email is classified as:", result)

This code tests the trained model on a new, unseen email sample:

Step 1: Define the New Email Text

- The code defines a new email text sample, which includes headers and body content.

Step 2: Transform the New Email using the Preprocessor

- The code uses the trained preprocessor to transform the new email text into a feature vector new\_features.

- The preprocessor applies the same transformations used during training, such as tokenization, stemming, and vectorization.

Step 3: Predict using the Trained Model

- The code uses the trained model to predict the class label of the new email sample.

- The prediction is stored in the prediction variable.

Step 4: Determine the Classification Result

- The code checks the predicted class label and assigns a classification result: "Spam" if the label is 1, and "Ham" otherwise.

Step 5: Print the Classification Result

- The code prints the classification result to the console.

This code demonstrates how to use the trained model and preprocessor to classify new, unseen email samples.

## 

## **Project Setup:**

1. **Environment Preparation**
   * **Tooling:**Install and set up a Python environment using tools like Jupyter Notebook or Google Colab.
   * To install jupyter notebook.

| !pip install jupyter notebook |
| --- |

* + **Library Installation:**

| **!pip install kagglehub** |
| --- |

| **!pip install nltk !pip install scikit-learn !pip install pandas !pip install matplotlib !pip install seaborn** |
| --- |

Or use requirements.txt file

!pip install -r requirements.txt

### **Future Enhancements**

* **Advanced Feature Engineering:** Explore additional natural language processing techniques, such as word embeddings (e.g., Word2Vec, BERT) or n-gram features, to capture semantic nuances beyond the handcrafted features. This could improve the model's ability to detect sophisticated spam patterns.
* **Ensemble and Deep Learning Models:** Experiment with ensemble methods (e.g., Random Forests, Gradient Boosting) and deep learning architectures (e.g., CNNs, RNNs) to further enhance performance, especially in handling imbalanced and high-dimensional data.
* **Dynamic and Real-Time Adaptation:** Develop mechanisms for real-time spam classification and continuous learning. Integrate online learning methods to adapt to evolving spam strategies and new email trends without requiring full retraining.
* **Robust Data Handling:** Implement more robust techniques for outlier detection and missing value imputation. Further refine preprocessing steps to better handle noisy data, varied email formats, and evolving language patterns.
* **Model Calibration and Interpretability:** Investigate calibration methods to transform regression outputs into reliable probability estimates and enhance model interpretability. This can help in understanding feature importance and decision-making criteria, which is crucial for trust and compliance in production systems.

### **Conclusion**

The Email Spam Classification project successfully developed a robust system that preprocesses raw email data, extracts relevant features, and applies various regularization techniques to mitigate overfitting and multicollinearity challenges. Logistic Regression emerged as the most effective model—achieving approximately 90% accuracy, with a balanced trade-off between precision and recall, and a high ROC-AUC around 0.92—making it well-suited for spam detection. While Lasso, Ridge, and Elastic Net demonstrated different strengths in handling feature selection and coefficient stabilization, they did not match the overall performance of Logistic Regression. The project not only delivers a practical spam filtering solution based on the SpamAssassin Public Corpus but also provides valuable insights into feature engineering, model tuning, and evaluation strategies. Future enhancements, such as incorporating advanced NLP techniques and ensemble methods, promise further improvements in classification accuracy and adaptability to evolving spam tactics.