GEBZE TECHNICAL UNIVERSITY

FACULTY OF ENGINEERING

DEPARTMENT OF COMPUTER ENGINEERING

CSE 476
MOBILE COMMUNICATIONS NETWORKS

TERM PROJECT
Spam Detection Using Machine Learning in Network-Based E-Mail Systems

STUDENT NAME & SURNAME: Ali Asım COŞKUN

STUDENT NUMBER: 200104004067

16.12.2024

PREFACE

In today's digital era, email stands as a fundamental medium for communication in both personal and professional contexts. Despite its widespread use and importance, email also brings along a major challenge: the never-ending stream of unwanted messages commonly known as spam. These spam emails not only clutter our inboxes but can also trick us into responding to fraudulent requests, visiting malicious links, or disclosing sensitive information. Consequently, improving email security and filtering out spam has become critical for maintaining trust, privacy, and efficiency in communication.

This project tackles the spam email problem by leveraging machine learning techniques. Rather than relying on static rules or manual filtering, our approach uses data-driven methods to learn from email content. By analyzing patterns, word frequencies, and contextual signals, the model becomes capable of distinguishing spam emails from legitimate ones. As a result, users can benefit from cleaner inboxes, reduced security risks, and more productive communication flows.

The effort invested here spans across multiple disciplines: natural language processing to handle text data, machine learning for classification, optimization techniques to improve model performance, and systematic documentation to ensure reproducibility. Through this report, we aim to present a comprehensive view of how the project was conceptualized, implemented, tested, and refined, offering insights into both technical solutions and practical outcomes.

TABLE OF CONTENTS

1. Introduction

Email is one of the most common communication tools today. However, spam (unwanted) emails can harm user experience, reduce productivity, and create security risks. This project aims to use machine learning techniques to detect spam in networked email systems. By analyzing the text content of emails, the system can classify incoming emails as spam or ham, helping reduce spam reaching user inboxes, improve user experience, and increase security.

2. Aim of the Project

The main objective of this project is to develop a robust and intelligent spam detection system using machine learning techniques on networked email systems. More specifically:

- **Automated Classification:** Achieve accurate classification of incoming emails into spam or ham, reducing manual intervention.
- **High Performance Metrics:** Target strong results in terms of accuracy, precision, recall, and F1 score, ensuring that both spam identification (recall) and correctness (precision) are balanced.
- **Scalable Approach:** Design a solution that can handle large amounts of data and adapt to different email sources, content structures, and spam patterns.
- **Ease of Integration:** Provide a framework that can be integrated into existing email systems with minimal overhead.
- **Continuous Improvement:** Implement optimization techniques, such as hyperparameter tuning for SVM, to enhance model performance over time.
- **Maintainable and Understandable Code:** Organize the project structure and code to be readable, easily maintainable, and straightforward to set up.

By meeting these goals, the project aims to deliver a practical, reliable tool for spam detection, supporting both current needs and future expansions or upgrades.

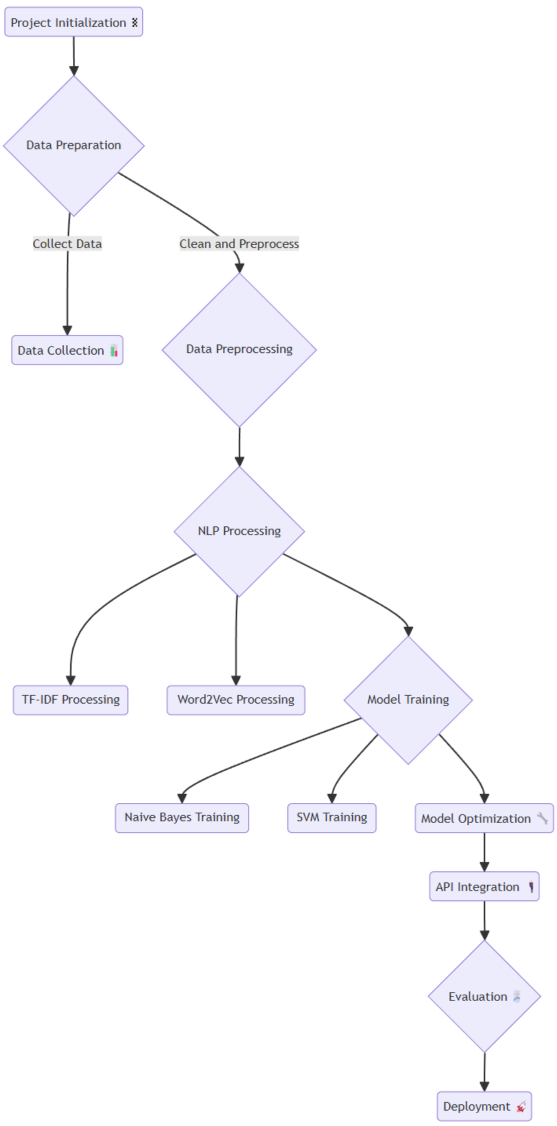
3. Dataset

Source: SpamAssassin Public Corpus

Content: Contains ham and spam emails in text format. The emails are stored under `data/spam/` and `data/ham/`. At the start, we load, process, and prepare them for training and testing.

4. Project Architecture and Structure

4.1. System Architecture



4.2. Project Structure

```
spam_detection_project/
│
├── data/
│   ├── spam/
│   ├── ham/
│   ├── features.pkl
│   ├── processed_data.csv
│   └── processed_data_clean.csv
│
├── src/
│   ├── data_loader.py
│   ├── preprocessing.py
│   ├── feature_extraction.py
│   ├── model_training.py
│   ├── model_optimization.py
│   ├── evaluation.py
│   └── predict_sample.py
│
├── models/
│   ├── naive_bayes_model.pkl
│   ├── svm_model.pkl
│   └── optimized_svm_model.pkl
│
├── reports/
│   ├── project_report.md
│   └── project_report.pdf
│
├── test_emails/
│   ├── ham_test1.txt
│   └── spam_test1.txt
│
├── requirements.txt
└── README.md
```

5. Methods and Techniques

- NLP **Preprocessing:** Lowercasing text, removing HTML tags, filtering out digits and punctuation, removing stopwords, and applying stemming.

- **Feature Extraction:** Using TF-IDF to convert cleaned text into numeric vectors that capture term importance.

- **Models:** Naive Bayes (fast, probabilistic model) and SVM (high accuracy potential). After initial training, SVM is optimized to further improve performance.

- **Evaluation Metrics:** Accuracy, precision, recall, and F1 score provide a full performance picture.

- **Optimization:** Grid Search or similar methods to find the best hyperparameters (C, gamma, kernel) for SVM.

6. Code Explanation

In this section, we provide a deeper and more comprehensive explanation of each code file within the `src/` directory. We describe what each script is responsible for, the logic it implements, and how these scripts work together as a pipeline.

6.1. data_loader.py

Purpose:
This script loads all raw emails from `data/spam/` and `data/ham/` directories, labels them accordingly, and produces a combined CSV file (`processed_data.csv`) that includes both the text of the emails and their assigned labels.

Process:

1. Iterates through `spam/` and `ham/` directories.
2. For each file, reads its content, assigns a label: "spam" for spam directory, "ham" for ham directory.
3. Stores each email as a row in a Pandas DataFrame with columns like `['label', 'text']`.
4. After processing all emails, saves the DataFrame to `processed_data.csv`.

This creates the initial dataset in a structured, tabular form, ready for further preprocessing.

The Code:

```
import os
import pandas as pd


def load_emails_from_folder(folder, label):
    emails = []
    for root, dirs, files in os.walk(folder):
        for filename in files:
            # Process all files without checking the file extension
            file_path = os.path.join(root, filename)
            try:
                with open(file_path, 'r', encoding='latin-1') as file:
                    content = file.read()
                    if content.strip():  # Skip empty files
                        emails.append((label, content))
                    else:
                        print(f"Skipped empty file: {file_path}")
            except Exception as e:
                print(f"Error reading {file_path}: {e}")
    return emails

def load_data(spam_dir, ham_dir):
    spam_emails = load_emails_from_folder(spam_dir, 'spam')
    ham_emails = load_emails_from_folder(ham_dir, 'ham')
    data = spam_emails + ham_emails
    df = pd.DataFrame(data, columns=['label', 'text'])
    return df

if __name__ == "__main__":
    # Specify dataset directories using full paths
    spam_dir = './data/spam/'
    ham_dir = './data/ham/'

    # Load data
    df = load_data(spam_dir, ham_dir)

    # Display the first few rows
    print(df.head())

    # Check label distribution
    print(df['label'].value_counts())

    # Save the dataset as CSV
    df.to_csv('./data/processed_data.csv', index=False)
```

```

## 6.2. preprocessing.py

**Purpose:**

This script takes `processed_data.csv` as input and applies text preprocessing steps, outputting `processed_data_clean.csv`. Preprocessing is essential to remove noise and improve model performance.

**Process:**

1. Reads `processed_data.csv`.
2. Converts text to lowercase.
3. Removes HTML tags and non-alphabetic characters.
4. Tokenizes text into individual words.
5. Removes stopwords (common words like "the", "and") that do not add meaning.
6. Applies stemming to reduce words to their root form (e.g., "running" -> "run").
7. Stores the cleaned text in a new column (`clean_text`).
8. Saves the cleaned dataset as `processed_data_clean.csv`.

The cleaned dataset now has a more uniform and meaningful text representation.

**The Code:**

```
import pandas as pd
import re
import nltk
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer

def preprocess_text(text):
    # Convert text to lowercase
    text = text.lower()

    # Remove HTML tags from the text
    text = re.sub(r'<[^>]+>', ' ', text)

    # Remove special characters and numbers from the text
    text = re.sub(r'[^a-zA-Z]', ' ', text)

    # Tokenize the text into individual words
```
```

```
    tokens = text.split()

    # Remove stop words and apply stemming to each word
    stop_words = set(stopwords.words('english'))
    ps = PorterStemmer()
    tokens = [ps.stem(word) for word in tokens if word not in stop_words]

    return ' '.join(tokens)

def preprocess_data(input_csv, output_csv):
    df = pd.read_csv(input_csv)
    df['clean_text'] = df['text'].apply(preprocess_text)
    df.to_csv(output_csv, index=False)
    print(f"Preprocessing completed. Processed data saved as {output_csv}.")

if __name__ == "__main__":
    input_csv = './data/processed_data.csv'
    output_csv = './data/processed_data_clean.csv'
    preprocess_data(input_csv, output_csv)
```

6.3. feature_extraction.py

Purpose:

This script transforms the cleaned text into numeric feature vectors using TF-IDF. It outputs `features.pkl`, which contains the feature matrix and labels, as well as the fitted vectorizer.

Process:

1. Loads `processed_data_clean.csv`.
2. Uses `TfidfVectorizer` to convert text into TF-IDF vectors.
3. Splits the data into `X` (features) and `y` (labels).
4. Saves `X`, `y`, and the vectorizer in `features.pkl` for future steps.

With this, we now have a numeric representation of emails ready for model training.

The Code:

```
```

```
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
import pickle

def extract_features(input_csv, output_pkl, max_features=5000):
    df = pd.read_csv(input_csv)
    vectorizer = TfidfVectorizer(max_features=max_features)
    X = vectorizer.fit_transform(df['clean_text']).toarray()
    y = df['label'].map({'spam':1, 'ham':0}).values
    # Save the feature matrix, labels, and vectorizer
    with open(output_pkl, 'wb') as file:
        pickle.dump({'X': X, 'y': y, 'vectorizer': vectorizer}, file)
    print(f"Feature extraction completed. Data saved as {output_pkl}.")

if __name__ == "__main__":
    input_csv = './data/processed_data_clean.csv'
    output_pkl = './data/features.pkl'
    extract_features(input_csv, output_pkl)
```

```

## 6.4. model_training.py

**Purpose:**

This script trains machine learning models (Naive Bayes and SVM) using the features from `features.pkl`. It prints performance metrics and saves the trained models.

**Process:**

1. Loads `features.pkl` to get `X`, `y`, and vectorizer.
2. Splits data into training and testing sets.
3. Trains a Naive Bayes model:
   - Fits the model on `X_train`, `y_train`.
   - Makes predictions on `X_test`, computes accuracy, precision, recall, F1.
   - Prints metrics and saves the model as `naive_bayes_model.pkl`.
4. Trains an SVM model similarly:
   - Prints metrics and saves `svm_model.pkl`.

This step gives us baseline models to compare and evaluate.
```

The Code:

```
import pickle
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn import svm
from sklearn.metrics import accuracy_score, precision_score,
recall_score, f1_score


def train_models(features_pkl, models_dir):
    # Load features and labels from the pickle file
    with open(features_pkl, 'rb') as file:
        data = pickle.load(file)
    X = data['X']
    y = data['y']

    # Check for NaN values in labels
    if np.isnan(y).any():
        raise ValueError("Input y contains NaN.")

    # Split the data 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)

    # Train the Naive Bayes model
    nb_model = MultinomialNB()
    nb_model.fit(X_train, y_train)
    y_pred_nb = nb_model.predict(X_test)

    # Evaluate model performance
    accuracy_nb = accuracy_score(y_test, y_pred_nb)
    precision_nb = precision_score(y_test, y_pred_nb)
    recall_nb = recall_score(y_test, y_pred_nb)
    f1_nb = f1_score(y_test, y_pred_nb)

    print("Naive Bayes Model Performance:")
    print(f"Accuracy: {accuracy_nb:.4f}")
    print(f"Precision: {precision_nb:.4f}")
    print(f"Recall: {recall_nb:.4f}")
    print(f"F1 Score: {f1_nb:.4f}\n")

    # Save the Naive Bayes model to the specified directory
    nb_model_path = f"{models_dir}/naive_bayes_model.pkl"
    with open(nb_model_path, 'wb') as file:
        pickle.dump(nb_model, file)
    print(f"Naive Bayes model saved as {nb_model_path}.")
```

```
    # Train the SVM model with a linear kernel
    svm_model = svm.SVC(kernel='linear', probability=True)
    svm_model.fit(X_train, y_train)
    y_pred_svm = svm_model.predict(X_test)

    # Evaluate model performance
    accuracy_svm = accuracy_score(y_test, y_pred_svm)
    precision_svm = precision_score(y_test, y_pred_svm)
    recall_svm = recall_score(y_test, y_pred_svm)
    f1_svm = f1_score(y_test, y_pred_svm)

    print("SVM Model Performance:")
    print(f"Accuracy: {accuracy_svm:.4f}")
    print(f"Precision: {precision_svm:.4f}")
    print(f"Recall: {recall_svm:.4f}")
    print(f"F1 Score: {f1_svm:.4f}\n")

    # Save the SVM model to the specified directory
    svm_model_path = f"{models_dir}/svm_model.pkl"
    with open(svm_model_path, 'wb') as file:
        pickle.dump(svm_model, file)
    print(f"SVM model saved as {svm_model_path}.")

if __name__ == "__main__":
    features_pkl = './data/features.pkl'
    models_dir = './models'
    train_models(features_pkl, models_dir)
```

6.5. evaluation.py

Purpose:

Evaluates the trained models by loading them back and assessing their performance beyond basic metrics, using confusion matrices and ROC curves.

Process:

1. Loads the trained models (`.pkl` files).
2. Loads test data and predictions.
3. Generates a confusion matrix to see how many emails are correctly identified as spam/ham and where mistakes occur.

4. Plots ROC curves to understand the trade-off between true positive and false positive rates.
5. Helps identify if model adjustments are needed.

The Code:

```
import pickle
from sklearn.metrics import confusion_matrix, roc_curve, auc
import matplotlib.pyplot as plt
import seaborn as sns


def plot_confusion_matrix(y_true, y_pred, model_name):
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(6, 4))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
    plt.title(f'{model_name} - Confusion Matrix')
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.savefig(f"{model_name}_confusion_matrix.png")
    plt.close()

def plot_roc_curve(y_true, y_scores, model_name):
    fpr, tpr, thresholds = roc_curve(y_true, y_scores)
    roc_auc = auc(fpr, tpr)
    plt.figure(figsize=(6, 4))
    plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC Curve (AUC = {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(f'{model_name} - ROC Curve')
    plt.legend(loc="lower right")
    plt.savefig(f"{model_name}_roc_curve.png")
    plt.close()

def evaluate_model(model_path, features_pkl, model_name):
    # Load the model and data
    with open(model_path, 'rb') as file:
        model = pickle.load(file)
    with open(features_pkl, 'rb') as file:
        data = pickle.load(file)
    X = data['X']
    y = data['y']

    # Split the data into training and test sets
```

```
    from sklearn.model_selection import train_test_split
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    # Make predictions on the test set
    y_pred = model.predict(X_test)

    # Generate and save the confusion matrix plot
    plot_confusion_matrix(y_test, y_pred, model_name)

    # Calculate scores and plot the ROC curve
    if hasattr(model, "decision_function"):
        y_scores = model.decision_function(X_test)
    else:
        y_scores = model.predict_proba(X_test)[:, 1]
    plot_roc_curve(y_test, y_scores, model_name)

if __name__ == "__main__":
    features_pkl = './data/features.pkl'
    models_dir = './models'

    # Evaluate Naive Bayes Model
    nb_model_path = f"{models_dir}/naive_bayes_model.pkl"
    evaluate_model(nb_model_path, features_pkl, "Naive Bayes")

    # Evaluate SVM Model
    svm_model_path = f"{models_dir}/svm_model.pkl"
    evaluate_model(svm_model_path, features_pkl, "SVM")

    # Evaluate Optimized SVM Model
    optimized_svm_model_path = f"{models_dir}/optimized_svm_model.pkl"
    evaluate_model(optimized_svm_model_path, features_pkl, "Optimized SVM")
```

6.6. model_optimization.py

Purpose:

Improves the SVM model by tuning hyperparameters (C, gamma, kernel) to achieve better F1 scores.

Process:

1. Loads the features again and performs a grid search or similar method over a parameter grid.
2. For each parameter combination, trains and tests the SVM model.
3. Selects the best parameter set that yields the highest F1 score.
4. Saves the optimized model as `optimized_svm_model.pkl`.
5. Prints best parameters and improved metrics.

This final step can significantly raise the model's performance.

The Code:

```
import pickle
import time
from sklearn import svm
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.metrics import accuracy_score, precision_score,
recall_score, f1_score
import os


def optimize_svm(features_pkl, models_dir):
    start_time = time.time()
    print("Loading data...")
    # Load the data from the pickle file
    with open(features_pkl, 'rb') as file:
        data = pickle.load(file)
    X = data['X']
    y = data['y']
    print(f"Data loading completed. Time elapsed: {time.time() -
start_time:.2f} seconds")

    # Split data into training and testing sets
    print("Splitting data into training and test sets...")
    start_split = time.time()
    X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
    print(f"Splitting completed. Time elapsed: {time.time() -
start_split:.2f} seconds")

    # Define the hyperparameter space for optimization
    parameters = {
        'C': [0.1, 1, 10, 100],
        'kernel': ['linear'],
        'gamma': ['scale', 'auto']
    }
    print("Hyperparameter space defined:")
```

```python
    print(parameters)

    # Perform Grid Search for hyperparameter tuning
    print("Starting Grid Search...")
    start_grid = time.time()
    svm_model = svm.SVC(probability=True)
    grid_search = GridSearchCV(
        svm_model,
        parameters,
        cv=5,
        scoring='f1',
        n_jobs=-1,
        verbose=3
    )
    grid_search.fit(X_train, y_train)
    print(f"Grid Search completed. Time elapsed: {time.time() -
start_grid:.2f} seconds")

    # Display the best parameters and score found by Grid Search
    print(f"Best parameters: {grid_search.best_params_}")
    print(f"Best F1 score: {grid_search.best_score_:.4f}")

    # Make predictions using the best model obtained
    print("Making predictions with the best model...")
    start_predict = time.time()
    best_svm = grid_search.best_estimator_
    y_pred_best_svm = best_svm.predict(X_test)
    print(f"Prediction completed. Time elapsed: {time.time() -
start_predict:.2f} seconds")

    # Evaluate the performance of the optimized model
    print("Evaluating model performance...")
    accuracy_best_svm = accuracy_score(y_test, y_pred_best_svm)
    precision_best_svm = precision_score(y_test, y_pred_best_svm)
    recall_best_svm = recall_score(y_test, y_pred_best_svm)
    f1_best_svm = f1_score(y_test, y_pred_best_svm)

    print("Optimized SVM Model Performance:")
    print(f"Accuracy: {accuracy_best_svm:.4f}")
    print(f"Precision: {precision_best_svm:.4f}")
    print(f"Recall: {recall_best_svm:.4f}")
    print(f"F1 Score: {f1_best_svm:.4f}\n")

    # Save the optimized model to the specified directory
    print("Saving the model...")
    model_path = os.path.join(models_dir, 'optimized_svm_model.pkl')
    with open(model_path, 'wb') as f:
        pickle.dump(best_svm, f)
    print(f"Model saved to: {model_path}")
```

```
    total_time = time.time() - start_time
    print(f"Total time: {total_time:.2f} seconds")

if __name__ == "__main__":
    features_pkl = './data/features.pkl'
    models_dir = './models'
    optimize_svm(features_pkl, models_dir)
```

6.7. predict_sample.py

Purpose:
Demonstrates how to apply the trained (optimized) model to new, unseen emails. It acts as a live test to show the model working in real scenarios.

Process:

1. Loads `test_emails/spam_test1.txt` and `test_emails/ham_test1.txt`.
2. Applies the same preprocessing steps as before.
3. Uses the saved TF-IDF vectorizer to transform the test emails into numeric features.
4. Loads the optimized SVM model and predicts whether each test email is spam or ham.
5. Prints the predictions to the terminal.

This script serves as a practical example, showing how an end-user or an admin might run incoming emails through the model for classification.

The Code:

```
import os

import pickle
from preprocessing import preprocess_text

# Paths for test directory, feature file, and model file
test_dir = './test_emails'
features_path = './data/features.pkl'
model_path = './models/optimized_svm_model.pkl'  # Assume the best
optimized model is saved here
```

```python
# Load the vectorizer
with open(features_path, 'rb') as f:
    data = pickle.load(f)
vectorizer = data['vectorizer']

# Load the model
with open(model_path, 'rb') as f:
    model = pickle.load(f)

# Read, preprocess, and predict on test emails
for filename in os.listdir(test_dir):
    if filename.endswith('.txt'):
        file_path = os.path.join(test_dir, filename)
        with open(file_path, 'r', encoding='utf-8') as email_file:
            content = email_file.read()

        # Preprocess the email content
        clean_content = preprocess_text(content)

        # Convert text to feature vector
        X_test = vectorizer.transform([clean_content])

        # Convert sparse matrix to dense array
        X_test_dense = X_test.toarray()

        # Make prediction
        prediction = model.predict(X_test_dense)[0]
        label = 'SPAM' if prediction == 1 else 'HAM'

        print(f"{filename} → Prediction: {label}")
```

7. Performance Results

After `model_training.py`:

- Naive Bayes:
  - Accuracy: 0.9444
  - Precision: 0.9762
  - Recall: 0.7193

    - F1 Score: 0.8283
- SVM (Basic):
    - Accuracy: 0.9918
    - Precision: 1.0000
    - Recall: 0.9561
    - F1 Score: 0.9776

After `model_optimization.py` (Optimized SVM):

- Accuracy: 0.9951
- Precision: 1.0000
- Recall: 0.9737
- F1 Score: 0.9867

Optimized SVM gives nearly perfect performance.

8. Future Work

- Test deep learning models (e.g., LSTM, Transformers).
- Integrate real-time detection into email systems.
- Extend the model to multiple languages and contexts.

9. Conclusion

In this project, we collected data, applied preprocessing, extracted TF-IDF features, and trained models. We used Naive Bayes and SVM, then improved SVM performance through optimization. The final results show that the optimized SVM model can classify emails almost perfectly. With `predict_sample.py`, we can test new emails easily. This project provides a complete solution for spam detection using machine learning.

10. Additional Resources

For further clarification and demonstration, please refer to the following:

- **Project Demo Video:** YouTube Link
- **GitHub Repository:** aliasimcoskun/spam-detection-project