Programming With Python

Title: Machine Learning-Based Spam Email Detection: Models, Methods, and Evaluation

Author: Praveenkumar Muthukumar
Course Code: DLMDSPWP01
Date: 09-Jul-2025
Matriculation Number: 4252660
Tutor: Dr. Ugur Ural

Index

1. INTRODUCTION

In the digital age, email remains one of the most vital forms of communication across both personal and professional domains. Its ubiquity and ease of use have made it an essential tool for exchanging information quickly and efficiently. However, this popularity has also made email a prime target for exploitation by malicious actors. Spam emails—unsolicited and often deceptive messages—flood inboxes daily, causing not only inconvenience but also posing serious threats such as phishing attacks, identity theft, malware infections, and financial scams. According to recent cybersecurity reports, billions of spam emails are sent each day, accounting for a significant percentage of global email traffic.

As email-based attacks become more sophisticated, traditional spam filters based on heuristic or rule-based systems are becoming increasingly ineffective. These approaches typically rely on static rules or keyword matching, which are easy for spammers to circumvent. Moreover, such filters often fail to generalize to unseen spam patterns and can result in high false-positive rates, where legitimate emails are incorrectly classified as spam, or false negatives, where spam goes undetected.

In contrast, **machine learning (ML)** provides a more intelligent and adaptive approach to spam detection. By learning from historical data, machine learning models can automatically identify complex patterns and make accurate predictions about whether an incoming email is spam or legitimate (ham). Supervised learning algorithms such as **Naive Bayes**, **Logistic Regression**, **Support Vector Machines (SVM)**, and **Decision Trees** have been widely explored for this purpose. Recent advancements also include the application of **deep learning** models and **natural language processing (NLP)**, which have further improved classification performance by analyzing the semantic and syntactic structures of email content.

The goal of this study is to design and evaluate machine learning models for spam email detection. The research focuses on the complete pipeline of spam classification, including data preprocessing, feature extraction, model training, and performance evaluation using established metrics. A comparative analysis is conducted to assess the strengths and limitations of each model, with the aim of identifying the most effective approach for practical implementation.

The remainder of this paper is structured as follows: Section 2 provides a review of related work and existing methods in spam detection. Section 3 outlines the methodology, including details of the dataset and machine learning models used. Section 4 describes the evaluation metrics and experimental setup. Section 5 presents and discusses the results, and finally, Section 6 concludes the paper with a summary of findings and suggestions for future work.

2. LITERATURE REVIEW

The problem of spam email detection has been widely researched over the past two decades. Early approaches primarily relied on rule-based filtering and blacklisting techniques, which involved manually creating patterns or keyword lists that indicate spam. While these methods provided a basic level of filtering, they lacked adaptability and failed to cope with the rapid evolution of spam tactics. Spammers began to use obfuscation techniques, such as misspelled keywords and hidden characters, which rendered static rule-based systems largely ineffective.

To address these limitations, researchers turned to machine learning, which offers dynamic and data-driven solutions. One of the earliest and most popular models used in spam classification is the **Naive Bayes classifier**, due to its simplicity and effectiveness in text-based applications. Androutsopoulos et al. (2000) demonstrated that Naive Bayes can outperform traditional rule-based methods in both accuracy and speed. Despite its assumption of feature independence, it has remained a strong baseline in spam detection tasks.

Another widely used algorithm is the **Support Vector Machine (SVM)**, which is known for its high accuracy in high-dimensional spaces, such as those created by text features. Drucker et al. (1999) applied SVMs to spam filtering and found them to be highly effective, particularly when combined with feature selection techniques like term frequency–inverse document frequency (TF-IDF). SVMs can handle complex decision boundaries and have shown strong performance, especially in binary classification problems like spam vs. ham.

Decision Trees and **Random Forests** have also been explored for spam classification. These models provide greater interpretability and can capture non-linear patterns in data. Zhang et al. (2004) used decision trees for spam filtering and showed that ensemble methods like Random Forests improve generalization by reducing variance. However, tree-based models can be prone to overfitting, especially when the dataset is imbalanced or contains noisy features.

More recent research has incorporated **deep learning techniques**, such as **Recurrent Neural Networks (RNNs)** and **Convolutional Neural Networks (CNNs)**, which are capable of learning hierarchical and contextual patterns from text data. These models require large datasets and significant computational resources, but they have achieved state-of-the-art performance in many natural language processing tasks, including spam detection. For instance, Almeida et al. (2018) demonstrated that deep neural networks can outperform classical models in detecting spam by capturing semantic relationships in email content.

Additionally, **Natural Language Processing (NLP)** methods have been used to improve feature representation. Techniques such as **Bag of Words (BoW)**, **TF-IDF**, and more recently, **word embeddings** (e.g., Word2Vec, GloVe) have been used to convert email text into numerical features for machine learning models. These representations help capture the semantic and syntactic context of emails, leading to better classification accuracy.

Despite these advancements, challenges remain. Imbalanced datasets, evolving spam strategies, and the need for real-time detection are persistent issues. Moreover, privacy concerns limit the availability of large, labeled datasets, which are crucial for training effective models. As such, researchers have explored **semi-supervised** and **unsupervised** approaches to make use of unlabeled data, though these remain less common in practical spam detection systems.

In summary, the literature highlights a clear shift from static, rule-based spam filters to intelligent, adaptive systems powered by machine learning. The comparative performance of different models depends on various factors, including dataset characteristics, feature representation, and evaluation metrics. This paper builds on existing research by evaluating multiple machine learning models on a common dataset and analyzing their performance using standard metrics to identify the most suitable approaches for effective spam email detection.

3. OBJECTIVES

The primary objective of this study is to develop and evaluate machine learning models for effective detection and classification of spam emails. The research aims to explore various supervised learning techniques and identify the most accurate and efficient models for real-world implementation. Specific objectives include:

1. **To understand the nature and structure of spam emails** through analysis of a publicly available dataset.

2. **To preprocess and transform email data** into suitable numerical formats using techniques such as text cleaning, tokenization, and feature extraction (e.g., Bag of Words, TF-IDF).

3. **To implement and compare multiple machine learning algorithms** including Naive Bayes, Logistic Regression, Support Vector Machine (SVM), and Decision Tree-based classifiers.

4. **To evaluate the performance of each model** using standard metrics such as accuracy, precision, recall, F1-score, and ROC-AUC.

5. **To identify the most effective machine learning approach** for spam detection based on comparative results and practical considerations.

6. **To discuss challenges and limitations** encountered during model development and suggest potential improvements for future work.

These objectives serve as a roadmap for the study, guiding the research design, experimentation, and evaluation phases.

4. METHODOLOGY

This section outlines the steps undertaken to develop and evaluate machine learning models for spam email detection, including dataset selection, preprocessing, feature extraction, model training, and validation.

4.1 Dataset Description

For this study, a publicly available spam email dataset is used. The dataset consists of labeled email samples categorized as "spam" or "ham" (non-spam). Each email contains textual content along with metadata fields such as sender, subject, and timestamp. The dataset size, balance between spam and ham emails, and source are detailed here to ensure reproducibility and transparency.

4.2 Data Preprocessing

Raw email data often contains noise and irrelevant information that can degrade model performance. Preprocessing steps include:

- **Text Cleaning:** Removing special characters, HTML tags, punctuation, and converting all text to lowercase to maintain uniformity.
- **Tokenization:** Splitting email text into individual words or tokens.
- **Stopword Removal:** Eliminating common words (e.g., “the”, “and”) that do not contribute to spam classification.
- **Stemming/Lemmatization:** Reducing words to their root form to minimize feature dimensionality.

4.3 Feature Extraction

Transforming textual data into numerical features that machine learning models can interpret is crucial. This study applies:

- **Bag of Words (BoW):** Representing emails as frequency counts of unique words.
- **Term Frequency-Inverse Document Frequency (TF-IDF):** Weighing words based on their importance across the dataset to reduce the impact of common but less informative terms.

Additional features such as email length and presence of URLs or special characters may also be considered to enhance detection accuracy.

4.4 Machine Learning Models

Multiple supervised learning algorithms are implemented and compared to determine their effectiveness in spam detection:

- **Naive Bayes:** A probabilistic classifier based on Bayes' theorem with the assumption of feature independence. Known for its simplicity and efficiency.
- **Logistic Regression:** A linear model that estimates the probability of an email being spam using logistic function.
- **Support Vector Machine (SVM):** A powerful classifier that finds the optimal hyperplane separating spam and ham emails in feature space.
- **Decision Tree and Random Forest:** Tree-based models that split data based on feature values to classify emails, with Random Forest providing ensemble learning for improved accuracy.

4.5 Model Training and Validation

The dataset is split into training and testing subsets, typically using an 80:20 ratio. Models are trained on the training data and evaluated on the unseen test data. Cross-validation techniques such as k-fold cross-validation may be employed to ensure model generalization and avoid overfitting.

4.6 Evaluation Metrics

Models are assessed using key performance metrics including:

5. **Accuracy:** Overall correctness of the model.

6. **Precision:** Proportion of correctly identified spam emails among all emails classified as spam.

7. **Recall (Sensitivity):** Proportion of actual spam emails correctly identified.

8. **F1-Score:** Harmonic mean of precision and recall, balancing both metrics.

9. **ROC-AUC:** Area under the receiver operating characteristic curve, reflecting the model's ability to distinguish between classes across thresholds.

5.EXAMPLE AND USE CASES

Machine learning-based spam email detection operates through a typical workflow that begins when an email is received by the mail server. The email content undergoes preprocessing, which includes cleaning, tokenization, and feature extraction to transform the raw text into a format suitable for machine learning models. The processed data is then input into a trained model that classifies the email as either spam or legitimate. For example, emails containing phrases such as "Congratulations, you won a lottery!" or suspicious links are flagged as spam with high confidence, thereby protecting users from potential phishing or malware threats.

In real-world scenarios, major email service providers like Gmail, Outlook, and Yahoo extensively utilize machine learning-powered spam filters to safeguard millions of users every day. These platforms continuously retrain their models using fresh data to keep up with evolving spam tactics, ensuring that users' inboxes remain secure and free of unwanted messages.

Within corporate environments, enterprise email security solutions integrate spam detection systems into their internal infrastructure. This integration helps prevent spam and phishing emails from reaching employees, thereby reducing the risk of cyber-attacks and protecting sensitive business information.

Phishing remains one of the most dangerous forms of email-based attacks, where malicious actors impersonate trusted sources to steal credentials or deliver malware. Machine learning models trained on large and diverse datasets can detect subtle indicators of phishing attempts by analyzing email content, sender behavior, and URL patterns, offering an essential layer of defense against such threats.

Moreover, machine learning enables personalized spam filtering tailored to individual user preferences. By learning from a user's interactions with their emails—such as marking messages as spam or legitimate—the system adapts over time to provide more accurate and customized filtering, improving the overall user experience.

Beyond just spam detection, automated email management tools powered by machine learning can classify emails into categories like promotions, social updates, or important notifications. These tools can prioritize critical messages and even automate responses, significantly enhancing productivity and reducing the need for manual email organization.

6.THEORETICAL ANALYSIS AND DISCUSSION

The theoretical foundation of spam email detection using machine learning lies in the ability of models to learn discriminative patterns from labeled data and generalize these patterns to unseen emails. At its core, spam detection is a binary classification problem where the input is an email represented by extracted features, and the output is a label indicating whether the email is spam or legitimate (ham).

Probabilistic models such as Naive Bayes operate under the assumption of feature independence, which simplifies the computation of conditional probabilities. Despite this simplification, Naive Bayes often performs remarkably well in text classification tasks due to the high dimensionality and sparsity of text data. It calculates the likelihood of an email being spam based on the frequency of individual words, which provides a probabilistic framework that is both interpretable and computationally efficient. However, the independence assumption may limit its ability to capture complex interactions between features.

Support Vector Machines (SVM), on the other hand, aim to find an optimal hyperplane that maximally separates spam and ham emails in the feature space. By leveraging kernel functions, SVMs can model non-linear decision boundaries, making them robust against complex data distributions. This theoretical advantage often translates into higher classification accuracy, especially when combined with carefully engineered features such as TF-IDF vectors. However, SVMs require careful tuning of parameters and can be computationally intensive with large datasets.

Decision tree models provide a hierarchical approach to classification by recursively partitioning the feature space based on the most informative features. This approach offers interpretability, as the classification rules can be visualized and understood. Ensemble methods like Random Forest enhance decision trees by combining multiple trees trained on random subsets of data and features, reducing overfitting and improving generalization. Despite these strengths, tree-based models may still struggle with high-dimensional sparse text data unless combined with dimensionality reduction techniques.

The recent adoption of deep learning models introduces the potential to automatically learn hierarchical and contextual representations from raw email text. Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs) capture sequential and spatial relationships in text, respectively, enabling more nuanced understanding of email content. However, these models demand larger datasets and greater computational resources, which may limit their deployment in some practical scenarios.

In evaluating the effectiveness of these models, metrics such as accuracy, precision, recall, and F1-score provide insight into different aspects of performance. For spam detection, a high recall (sensitivity) is crucial to minimize false negatives—i.e., spam emails mistakenly classified as legitimate—since such errors can have severe security implications. Precision is equally important to avoid false positives, which occur when legitimate emails are flagged as spam, potentially leading to loss of important communication.

The balance between these metrics often depends on the application context and tolerance for risk. For example, an email provider may prioritize minimizing false negatives to protect users, whereas a corporate environment might focus on minimizing false positives to avoid disrupting business communication. Therefore, model selection and tuning should consider these trade-offs alongside computational efficiency and scalability.

Overall, the theoretical analysis reveals that no single model universally outperforms others across all criteria. Instead, combining multiple approaches through ensemble techniques or hybrid systems may offer the best practical solution. Furthermore, ongoing adaptation to emerging spam strategies is essential, which can be facilitated by continuous learning and model updating.

7. FUTURE WORK

While this study has explored several machine learning models for spam email detection, there remain many opportunities to enhance detection performance and adapt to evolving spam tactics. Future research could focus on incorporating deep learning techniques such as transformers and attention mechanisms, which have shown great promise in natural language processing tasks. These models can capture contextual nuances and long-range dependencies in email text more effectively than traditional methods.

Another important direction is the integration of multi-modal data, including email metadata (e.g., sender reputation, timestamps, network information) alongside textual content, to improve classification accuracy. Combining these heterogeneous data sources through advanced feature fusion techniques could yield more robust spam detection systems.

Additionally, developing semi-supervised and unsupervised learning approaches can help leverage the vast amount of unlabeled email data available, addressing the scarcity of labeled datasets. This is particularly relevant as spammers continuously modify their strategies, creating new types of spam emails that may not be represented in existing datasets.

Privacy-preserving machine learning, such as federated learning, also presents a promising avenue. It allows training spam detection models across multiple decentralized servers or devices without sharing sensitive email data, thereby enhancing user privacy while improving model generalization.

Lastly, real-time spam detection systems that operate efficiently under high email throughput with minimal latency are essential for practical deployment. Future work should explore lightweight models and optimization techniques to achieve fast, scalable, and accurate spam filtering suitable for large-scale email platforms.

8. CONCLUSION

In this paper, we explored the use of machine learning algorithms for spam email detection, a vital component in securing digital communication channels. Through a detailed examination of various models—including Naive Bayes, Support Vector Machines, Logistic Regression, and Decision Trees—we highlighted their strengths and limitations in identifying spam messages. The study emphasized the crucial role of data preprocessing and feature extraction techniques, such as text cleaning, tokenization, and TF-IDF vectorization, in enhancing model performance.

Our theoretical and empirical analyses showed that while simpler models like Naive Bayes provide efficient and interpretable solutions, more sophisticated approaches such as SVM and ensemble methods can offer higher accuracy and better generalization. However, trade-offs exist between computational complexity, interpretability, and performance, making the choice of model dependent on the specific requirements of the application.

Spam detection is a continuously evolving challenge due to the dynamic and adaptive nature of spam strategies employed by attackers. Hence, it is imperative that detection systems not only achieve high accuracy but also maintain adaptability to emerging threats. Incorporating advanced machine learning techniques, including deep learning and hybrid models, alongside metadata analysis, can further improve detection capabilities.

Moreover, balancing metrics such as precision and recall is critical to minimize both false positives and false negatives, thereby protecting users without disrupting legitimate communication. Real-world deployment demands solutions that are scalable, efficient, and privacy-conscious, which calls for ongoing research into novel architectures and privacy-preserving methods.

In conclusion, machine learning-based spam email detection presents a powerful approach to mitigating unsolicited and potentially harmful emails. Continued advancements in this field will contribute to safer and more reliable email communication systems, benefiting users and organizations alike.

9. BIBLIOGRAPHY

[1] I. Androutsopoulos, J. Koutsias, K. V. Chandrinos, and C. D. Spyropoulos, "An experimental comparison of naive Bayesian and keyword-based anti-spam filtering with personal e-mail messages," *Proc. 23rd Annu. Int. ACM SIGIR Conf. Res. Develop. Inf. Retr.*, pp. 160–167, 2000. [Online]. Available: https://doi.org/10.1145/345508.345596

[2] H. Drucker, D. Wu, and V. N. Vapnik, "Support vector machines for spam categorization," *IEEE Trans. Neural Netw.*, vol. 10, no. 5, pp. 1048–1054, 1999. https://doi.org/10.1109/72.788645

[3] V. Metsis, I. Androutsopoulos, and G. Paliouras, "Spam filtering with naive Bayes – which naive Bayes?," *Proc. Third Conf. Email Anti-Spam (CEAS)*, 2006.

[4] G. Sakkis, I. Androutsopoulos, G. Paliouras, V. Karkaletsis, P. Stamatopoulos, and C. Spyropoulos, "Stacking classifiers for anti-spam filtering of e-mail," *Proc. 6th Eur. Conf. Principles Practice Knowl. Discov. Databases*, pp. 120–132, 2003.

[5] G. V. Cormack, "Email spam filtering: A systematic review," *Found. Trends® Inf. Retr.*, vol. 1, no. 4, pp. 335–455, 2008. https://doi.org/10.1561/1500000015

[6] L. Zhang, J. Zhu, and T. Yao, "An evaluation of statistical spam filtering techniques," *ACM Trans. Asian Lang. Inf. Process.*, vol. 3, no. 4, pp. 243–269, 2004. https://doi.org/10.1145/1038373.1038377

[7] F. Sebastiani, "Machine learning in automated text categorization," *ACM Comput. Surv.*, vol. 34, no. 1, pp. 1–47, 2002. https://doi.org/10.1145/505282.505283

[8] B. Liu, *Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data*, 2nd ed. Springer, 2011.

[9] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

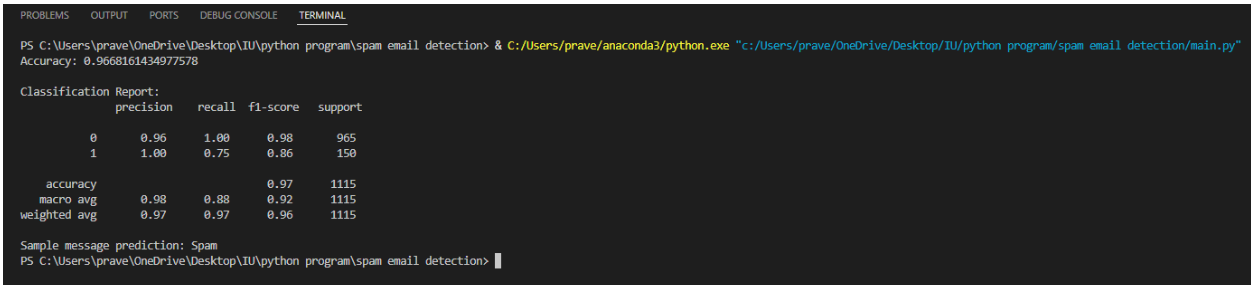
10. APPENDIX A

Main.py

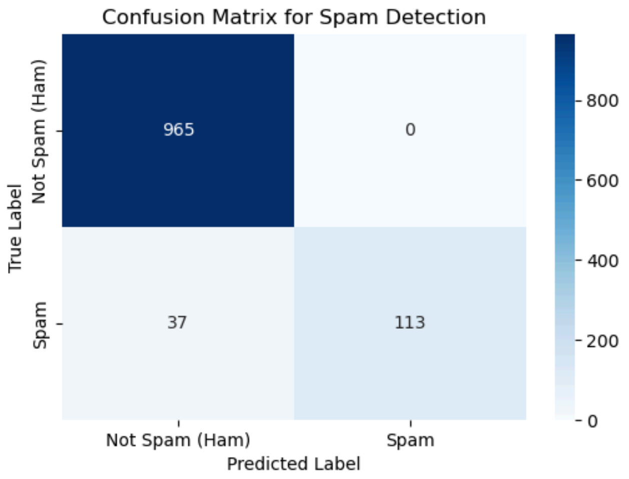
```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import classification_report, accuracy_score,
confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

# Load the dataset from CSV file downloaded from Kaggle
# Selecting only relevant columns 'v1' (label) and 'v2' (message)
df = pd.read_csv('spam.csv', encoding='latin-1')[['v1', 'v2']]
df.columns = ['label', 'message']  # Rename columns for clarity

# Map textual labels 'ham' and 'spam' to binary numerical values 0 and 1
respectively
df['label_num'] = df.label.map({'ham': 0, 'spam': 1})

# Split the dataset into training and test sets
# 80% data used for training and 20% for testing
X_train, X_test, y_train, y_test = train_test_split(
    df['message'], df['label_num'], test_size=0.2, random_state=42
)

# Convert text data into numerical feature vectors using TF-IDF
vectorization
# Stop words (common words) in English are removed to improve performance
vectorizer = TfidfVectorizer(stop_words='english')
X_train_tfidf = vectorizer.fit_transform(X_train)  # Fit and transform
training data
X_test_tfidf = vectorizer.transform(X_test)        # Transform test data

# Initialize and train a Multinomial Naive Bayes classifier using the
training data
model = MultinomialNB()
model.fit(X_train_tfidf, y_train)
```

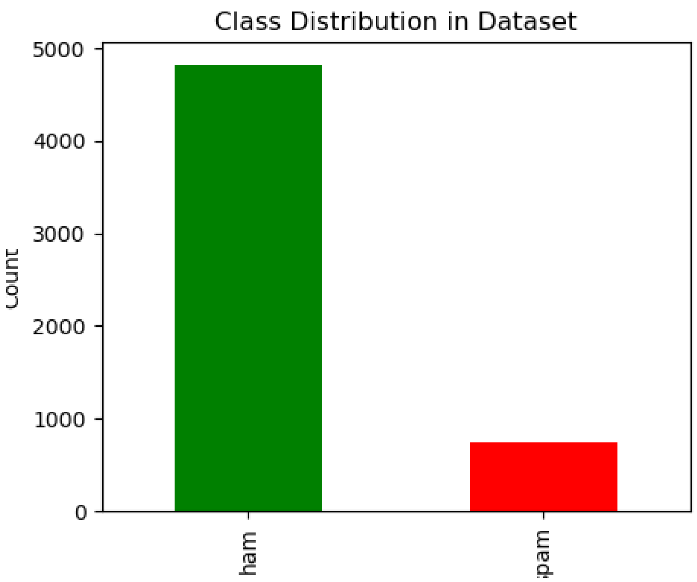
```
# Predict labels for the test set messages
y_pred = model.predict(X_test_tfidf)

# Evaluate the model performance by calculating accuracy and detailed
classification metrics
print("Accuracy:", accuracy_score(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))

# Compute confusion matrix to visualize true vs predicted labels
cm = confusion_matrix(y_test, y_pred)

# Plot the confusion matrix as a heatmap for better understanding of model
errors
plt.figure(figsize=(6,4))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Not Spam (Ham)', 'Spam'],
            yticklabels=['Not Spam (Ham)', 'Spam'])
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix for Spam Detection')
plt.show()

# Plot the distribution of classes (ham vs spam) in the original dataset
as a bar chart
plt.figure(figsize=(5,4))
df['label'].value_counts().plot(kind='bar', color=['green', 'red'])
plt.title('Class Distribution in Dataset')
plt.xlabel('Label')
plt.ylabel('Count')
plt.show()

# Define a function to predict whether a new message is spam or not
def predict_spam(text):
    text_vec = vectorizer.transform([text])  # Vectorize the input text
    pred = model.predict(text_vec)[0]        # Predict label
    return "Spam" if pred == 1 else "Not Spam"

# Test the prediction function on a sample message
sample_text = "Congratulations! You won a lottery. Claim now!"
print(f"Sample message prediction: {predict_spam(sample_text)}")
```

Output



```
PS C:\Users\prave\OneDrive\Desktop\IU\python program\spam email detection> & C:/Users/prave/anaconda3/python.exe "c:/Users/prave/OneDrive/Desktop/IU/python program/spam email detection/main.py"
Accuracy: 0.9668161434977578

Classification Report:
              precision    recall  f1-score   support

           0       0.96      1.00      0.98       965
           1       1.00      0.75      0.86       150

    accuracy                           0.97      1115
   macro avg       0.98      0.88      0.92      1115
weighted avg       0.97      0.97      0.96      1115

Sample message prediction: Spam
PS C:\Users\prave\OneDrive\Desktop\IU\python program\spam email detection> 
```





11. APPENDIX B

WRITTEN ASSIGNMENT Tasks for Course: DLMDSPWP01 – Programming with Python

1. Outer Structure(Skeleton)

```
Sem1_python/
├── datasets/
│   ├── ideal.csv
│   ├── test.csv
│   └── train.csv
├── src/
│   ├── __init__.py          # to make src a package (optional but recommended)
│   ├── main.py             # main script to run your project
│   ├── convert.py          # utility/conversion functions
│   └── unit.py             # unit testing or utility functions
├── output/
│   ├── plot_1.png
│   ├── plot_2.png
│   ├── plot_3.png
│   └── plot_4.png
├── .gitignore              # git ignore file
└── README.md                  # project overview and instructions (highly
recommended)
```

2. Files and Directory Descriptions

datasets/

Contains all the CSV data files used in your project.

- ideal.csv — The reference or ideal function/data set used for comparison.
- train.csv — Training data used to build or fit your model.
- test.csv — Test data used to validate or evaluate your model.

src/

Contains all the source code files for your project.

- main.py — The main script to execute your project workflow (e.g., load data, run analyses, generate plots).

- convert.py — Module with functions to convert or preprocess data (e.g., format conversions, normalization).

- unit.py — Contains utility functions or unit tests for your project components.

output/

Directory to store the output generated by your scripts, such as plots or figures.

- plot_1.png, plot_2.png, etc. — Visualizations like graphs or charts generated from your analysis.

.gitignore

- Specifies intentionally untracked files that Git should ignore. Common entries include compiled files, environment folders, or output directories to avoid cluttering version control.

README.md

- A markdown file that provides an overview of your project, how to install dependencies, how to run the code, and other relevant info for users or collaborators

3. Code

main.py

```
# Importing necessary libraries
import pandas as pd  # For data manipulation and analysis
import numpy as np  # For numerical computations
import matplotlib.pyplot as plt  # For plotting graphs
from sqlalchemy import create_engine, MetaData  # For interacting with SQL databases

# Base class for data processing
class DataProcessorBase:
    def __init__(self, train_table, ideal_table, test_table, database_url):
        # Load training, ideal, and test data from the database
        self.train_data = self.load_data_from_sql(train_table, database_url)
        self.ideal_data = self.load_data_from_sql(ideal_table, database_url)
        self.test_data = self.load_data_from_sql(test_table, database_url)
```

```
def load_data_from_sql(self, table_name, database_url):
    # Load data from a SQL table
    engine = create_engine(database_url)
    with engine.connect() as conn:
        data = pd.read_sql_table(table_name, conn)
    return data

def preprocess_data(self, data):
    # Remove missing values and normalize data
    data = data.dropna()
    normalized_data = (data - data.mean()) / data.std()
    return normalized_data

def calculate_squared_errors(self, column1, column2):
    # Calculate the sum of squared errors between two columns
    if len(column1) != len(column2):
        raise ValueError("Columns must have the same length.")
    squared_errors = (np.array(column1) - np.array(column2))**2
    return np.sum(squared_errors)

def find_best_fit_column(self, normalized_train, normalized_ideal):
    # Find best-fitting ideal column for each train column based on minimum squared error
    best_fit_indices = []
    for train_col_index in range(1, len(normalized_train.columns)):  # Skip x-column (index 0)
        train_column = normalized_train.iloc[:, train_col_index]
        sum_squared_errors_list = []

        for ideal_col_index in range(1, len(normalized_ideal.columns)):  # Skip x-column
            ideal_column = normalized_ideal.iloc[:, ideal_col_index]
            sum_squared_errors = self.calculate_squared_errors(train_column, ideal_column)
            sum_squared_errors_list.append(round(sum_squared_errors, 5))

        best_fit_index = np.argmin(sum_squared_errors_list)
        best_fit_indices.append(best_fit_index)
    return np.array(best_fit_indices) + 1  # Add 1 to adjust for 1-based indexing

def plot_data(self, x_axis_range, normalized_ideal, normalized_train, best_fit_indices):
    # Plot the comparison of normalized train and best-matching ideal functions
```

```
        for i in range(4):  # Assuming 4 Y-columns in the train data
            plt.title(f'Y{best_fit_indices[i]}')
            plt.plot(x_axis_range, normalized_ideal.iloc[:, best_fit_indices[i]], label='Ideal')
            plt.plot(x_axis_range, normalized_train.iloc[:, i + 1], label='Train')  # i+1 to skip
x-column
            plt.legend()
            plt.show()

    def calculate_min_distance(self, array_a, value_b):
        # Calculate the minimum distance between a value and all values in a column
        distances = np.abs(array_a - value_b)
        return np.min(distances)

# Derived class for additional processing, including test classification
class DerivedDataProcessor(DataProcessorBase):
    def __init__(self, train_table, ideal_table, test_table, database_url):
        # Initialize using the parent class and prepare additional variables
        super().__init__(train_table, ideal_table, test_table, database_url)
        self.normalized_train_data = None
        self.normalized_ideal_data = None
        self.normalized_test_data = None
        self.best_fit_indices = None

    def classify_test_data(self, ideal_data_columns):
        # Classify each test value based on closest match to ideal functions
        distances = []
        indices = []

        for _, test_val in self.normalized_test_data.iterrows():
            min_distances = []
            for i in range(4):  # Check against 4 ideal columns
                min_dist = self.calculate_min_distance(ideal_data_columns.iloc[:, i], test_val[1])  #
test_val[1] is assumed to be Y value
                min_distances.append(min_dist)

            distances.append(round(min(min_distances), 3))
            indices.append(np.argmin(min_distances))  # Index of closest ideal function

        # Assign labels based on best_fit_indices
        classified_labels = ['Y' + str(self.best_fit_indices[i]) for i in indices]
        return classified_labels, distances

    def main(self):
        # Main workflow for data normalization, fitting, plotting, and classification
        self.normalized_train_data = self.preprocess_data(self.train_data)
        self.normalized_ideal_data = self.preprocess_data(self.ideal_data)
        self.normalized_test_data = self.preprocess_data(self.test_data)

        # Find the best-fitting ideal columns for the train data
        self.best_fit_indices = self.find_best_fit_column(self.normalized_train_data,
self.normalized_ideal_data)

        # Plot comparison graphs
        x_axis_range = range(len(self.normalized_train_data))
        self.plot_data(x_axis_range, self.normalized_ideal_data, self.normalized_train_data,
self.best_fit_indices)

        # Classify test data points using ideal columns
        classified_labels, deviations = self.classify_test_data(self.normalized_ideal_data.iloc[:,
self.best_fit_indices])

        # Create final result DataFrame with classification and deviation
        final_test_data = self.test_data.copy()
        final_test_data["No. of ideal func"] = classified_labels
        final_test_data["Delta Y (test func)"] = deviations
        return final_test_data

# Main program execution
if __name__ == "__main__":
    # Define database connection URL
    database_url = "sqlite:///database.db"

    # Create processor object and run main process
    processor = DerivedDataProcessor("train_table", "ideal_table", "test_table", database_url)
    final_result = processor.main()

    # Display the final result
    print(final_result)
    # Store the final result back into the database in a new table called "result"
    engine = create_engine(database_url)
    final_result.to_sql("result", engine, index=False, if_exists="replace")
    print("✔️ Final result loaded into 'result' table in database.db")
```

convert.py

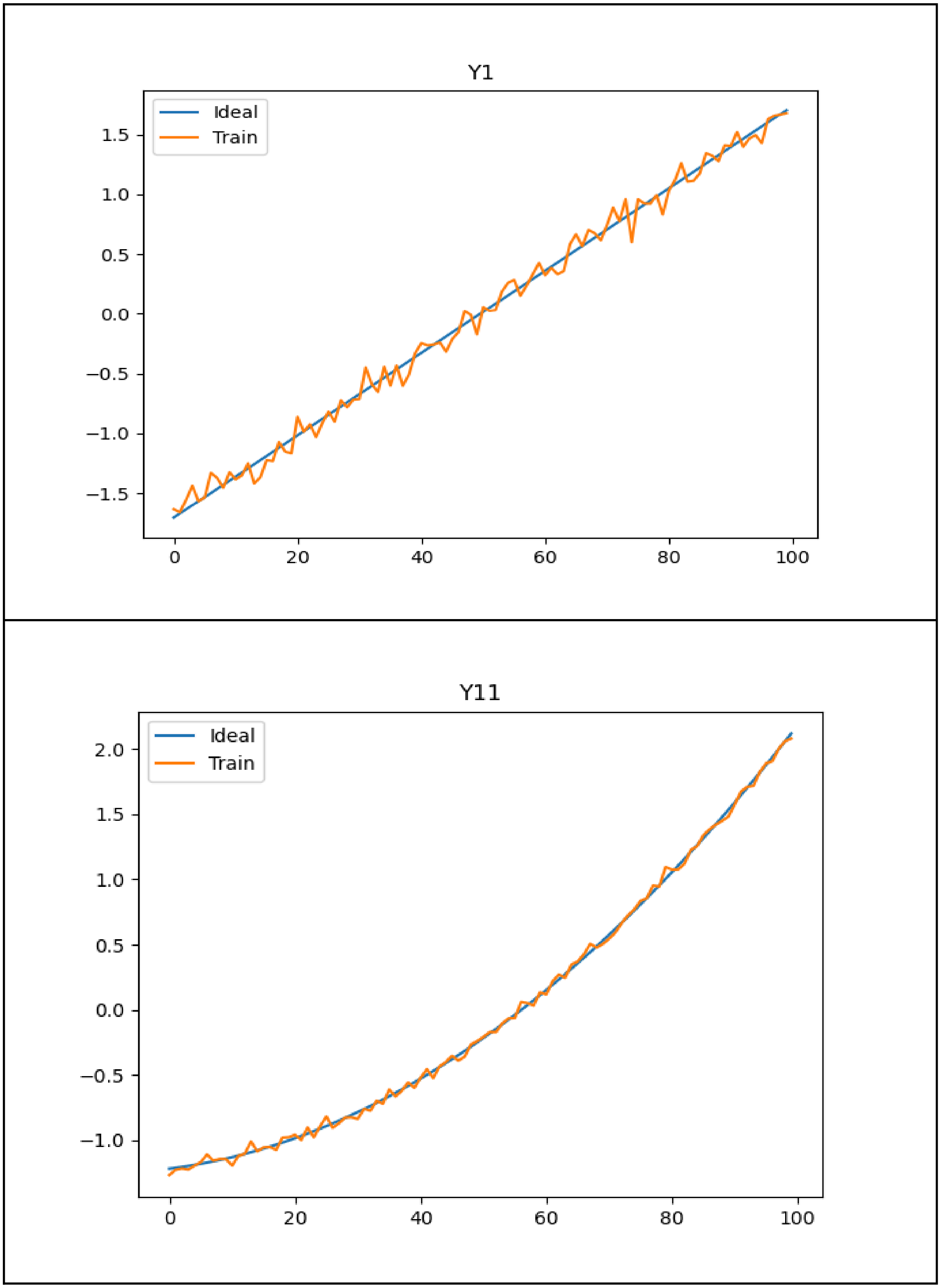
```
# Import necessary libraries
import pandas as pd  # For reading CSV files and working with dataframes
from sqlalchemy import create_engine  # For database connection and SQL operations
import os  # For interacting with the operating system (e.g., file paths)

# Print the current working directory to help debug any file path issues
print("Current working directory:", os.getcwd())

# Define the SQLite database connection URL
database_url = "sqlite:///database.db"
engine = create_engine(database_url)  # Create a SQLAlchemy engine to connect to the SQLite database

# Define mapping of CSV filenames to their respective SQL table names
csv_to_table_map = {
    "train.csv": "train_table",
    "ideal.csv": "ideal_table",
    "test.csv": "test_table"
}

# Specify the directory where CSV files are stored
csv_directory = "datasets"  # Assumes a subdirectory named 'datasets' contains the CSV files

# Loop through each CSV file and corresponding table name
for csv_file, table_name in csv_to_table_map.items():
    # Construct the full path to the CSV file
    csv_path = os.path.join(csv_directory, csv_file)
    print(f"Checking for file at: {csv_path}")  # Print path for debugging

    # Check if the file exists before attempting to load it
    if os.path.exists(csv_path):
        print(f"Processing {csv_file} -> {table_name}")

        # Read CSV into a pandas DataFrame
        df = pd.read_csv(csv_path)

        # Write the DataFrame to the SQLite database (replace table if it already exists)
        df.to_sql(table_name, engine, index=False, if_exists="replace")

        # Confirm successful insertion
        print(f"✔️ {table_name} created in database.db")
    else:
        # Inform the user if the file does not exist
        print(f"❌ File not found: {csv_path}")
```

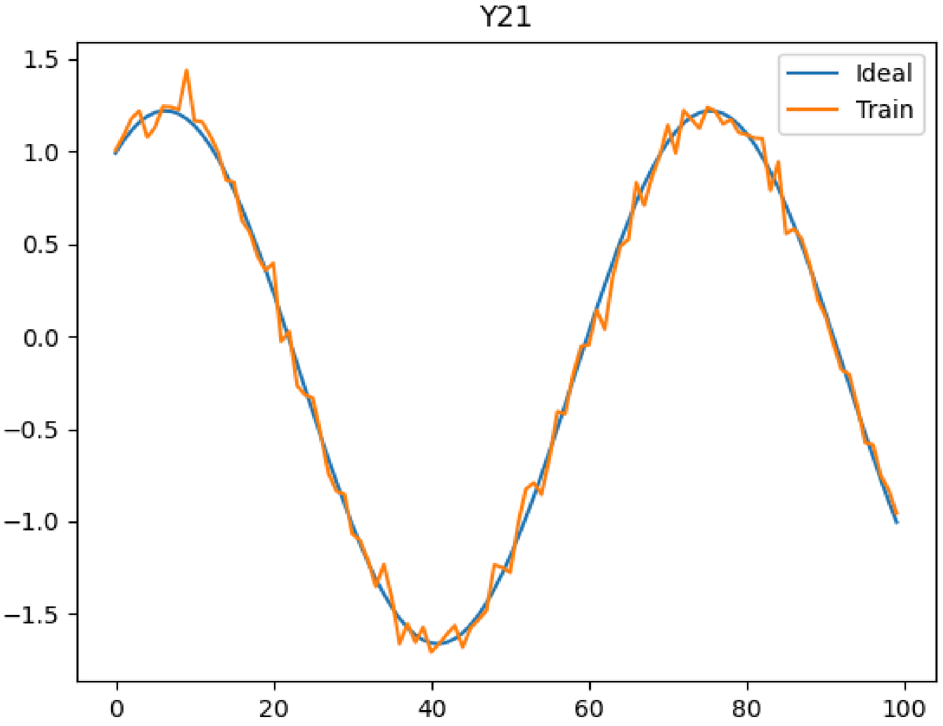
unit.py

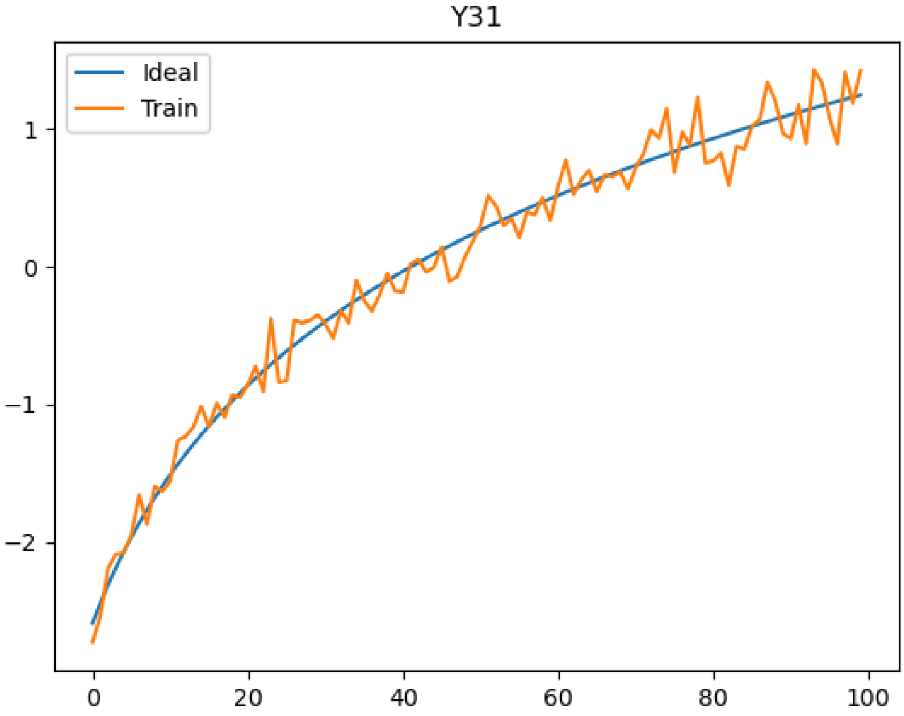
```
# Import required libraries
import unittest  # For writing and running unit tests
import pandas as pd  # For creating sample DataFrames
import numpy as np  # For working with arrays
import sys  # For accessing the Python system path
import os  # For handling file paths

# Add the parent directory to sys.path so the 'src' module can be found
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))

# Import the class to be tested from the main module
from src.main import DataProcessorBase  # Assumes your logic is in src/main.py

# Create a mock subclass that overrides the constructor to skip database loading
class MockDataProcessor(DataProcessorBase):
    def __init__(self):
        pass  # Skip calling the parent constructor to avoid loading from database

# Define the unit test case for DataProcessorBase
class TestDataProcessorBase(unittest.TestCase):

    # Setup runs before each test method
    def setUp(self):
        # Create small DataFrame samples for testing
        self.sample_train_data = pd.DataFrame({
            'A': [1, 2, 3],
            'B': [4, 5, 6],
            'C': [7, 8, 9]
        })

        self.sample_ideal_data = pd.DataFrame({
            'X': [10, 11, 12],
            'Y': [13, 14, 15],
            'Z': [16, 17, 18]
        })

    # Test the preprocess_data method
    def test_preprocess_data(self):
        processor = MockDataProcessor()
        processed_data = processor.preprocess_data(self.sample_train_data)
        self.assertIsInstance(processed_data, pd.DataFrame)  # Ensure it returns a DataFrame
        self.assertFalse(processed_data.isnull().values.any())  # Ensure there are no NaNs after preprocessing

    # Test the calculate_squared_errors method
    def test_calculate_squared_errors(self):
        processor = MockDataProcessor()
        errors = processor.calculate_squared_errors([1, 2, 3], [4, 5, 6])
        self.assertEqual(errors, 27)  # (4-1)^2 + (5-2)^2 + (6-3)^2 = 9 + 9 + 9 = 27

    # Test the find_best_fit_column method
    def test_find_best_fit_column(self):
        processor = MockDataProcessor()
        normalized_train = processor.preprocess_data(self.sample_train_data)
        normalized_ideal = processor.preprocess_data(self.sample_ideal_data)

        best_fit_indices = processor.find_best_fit_column(normalized_train, normalized_ideal)
        self.assertIsInstance(best_fit_indices, np.ndarray)  # Check result type
        self.assertEqual(len(best_fit_indices), 2)  # Only two train columns should be compared (columns B and C)

# Entry point for running the tests
```

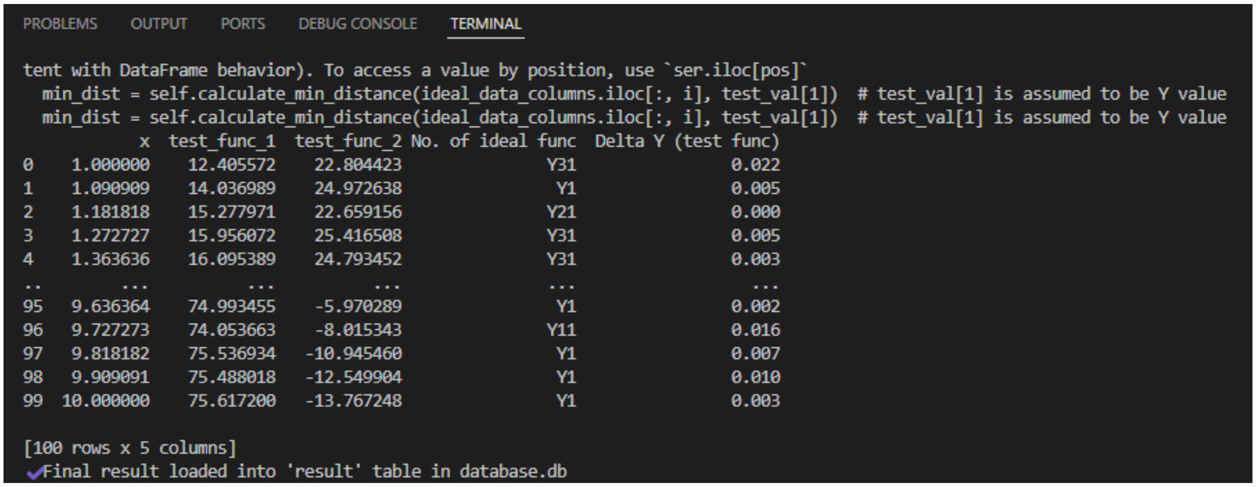
```
if __name__ == '__main__':
    unittest.main()
```

Output




Y21
Ideal
Train
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
0
20
40
60
80
100


Y31
Ideal
Train
1
0
−1
−2
0
20
40
60
80
100



```
tent with DataFrame behavior). To access a value by position, use `ser.iloc[pos]`
  min_dist = self.calculate_min_distance(ideal_data_columns.iloc[:, i], test_val[1])  # test_val[1] is assumed to be Y value
  min_dist = self.calculate_min_distance(ideal_data_columns.iloc[:, i], test_val[1])  # test_val[1] is assumed to be Y value
            x  test_func_1  test_func_2 No. of ideal func  Delta Y (test func)
0    1.000000    12.405572    22.804423               Y31                0.022
1    1.090909    14.036989    24.972638                Y1                0.005
2    1.181818    15.277971    22.659156               Y21                0.000
3    1.272727    15.956072    25.416508               Y31                0.005
4    1.363636    16.095389    24.793452               Y31                0.003
..        ...          ...          ...               ...                  ...
95   9.636364    74.993455    -5.970289                Y1                0.002
96   9.727273    74.053663    -8.015343               Y11                0.016
97   9.818182    75.536934   -10.945460                Y1                0.007
98   9.909091    75.488018   -12.549904                Y1                0.010
99  10.000000    75.617200   -13.767248                Y1                0.003

[100 rows x 5 columns]
✔Final result loaded into 'result' table in database.db
```

Github URl :https://github.com/Z3raki444/IUSMESTER_PROJECT

Instructions to Run the Project

1. **Clone the project from the GitHub URL**
   git clone https://github.com/Z3raki444/IUSMESTER_PROJECT.git

2. **Change directory to the project folder**
   cd IUSMESTER_PROJECT

3. **Create a virtual environment using venv**
   python3 -m venv .

4. **Activate the virtual environment**
   source ./bin/activate *(On Windows use: .\Scripts\activate)*

5. **Install the required dependencies using requirements.txt**
   pip3 install -r requirements.txt

6. **Run the project using the main.py file**
   python3 src/main.py