ML PROJECT

Low level Documentation

Project Title- Spam ham classifier

Submitted To: -

PHYSICS WALLAH PVT. LTD

Founder: - MR. Alakh Pandey

Submitted By:-

Jyoti Kumari

Course- Full Stack Data Science Pro

E-mail – jyotithakur2161@gmail.com

Table Content

Introduction

--Low level Document—

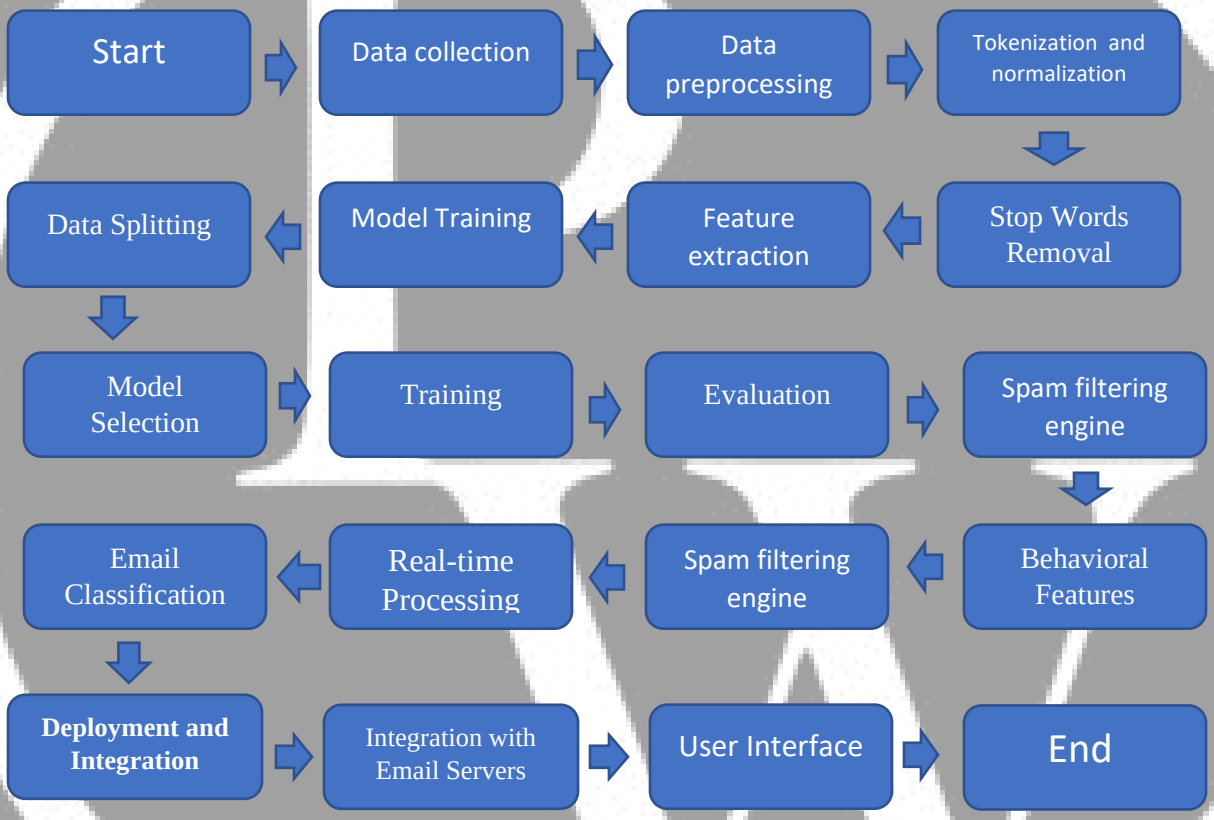
A low-level document, often referred to as a detailed design document or technical specification, plays a crucial role in the development and maintenance of software systems. It delves into the intricate details of the system's architecture, components, and functionalities, providing a comprehensive blueprint that guides developers and engineers through the implementation process.

Purpose and Importance

The primary purpose of a low-level document is to translate high-level design and requirements into actionable and executable technical details. It serves as a bridge between the conceptual design and the actual coding, ensuring that every aspect of the system is meticulously planned and documented. This level of detail is essential for several reasons:

1. **Clarity and Consistency**: By providing detailed descriptions of each component and their interactions, a low-level document ensures that all team members have a clear and consistent understanding of the system. This helps in avoiding misunderstandings and discrepancies during development.
2. **Guidance for Developers**: Developers rely on low-level documents to understand how to implement specific functionalities, what data structures to use, and how different modules should interact. It serves as a reference guide throughout the coding phase.
3. **Maintenance and Scalability**: Well-documented systems are easier to maintain and scale. Future developers can refer to the low-level document to understand the existing system, troubleshoot issues, and make enhancements without reworking the entire codebase.
4. **Quality Assurance**: Detailed documentation aids in the creation of thorough test cases and validation procedures. By knowing the expected behaviors and interactions, testers can design effective tests to ensure the system meets its requirements.
5. **Risk Mitigation**: By anticipating potential issues and detailing error handling mechanisms, low-level documents help in mitigating risks associated with system failures and security vulnerabilities.

Architecture



Architecture Description

The architecture of a ham/spam email classification system involves several components that work together to filter and classify emails as either legitimate (ham) or unwanted (spam). Here's a detailed breakdown of the typical architecture:

1. Data Collection and Preprocessing

- Email Data Collection: Emails are collected from various sources such as mail servers, user inboxes, or datasets.
- Preprocessing: Emails are preprocessed to remove noise and extract relevant features. This may include:
- Tokenization: Breaking down the email content into individual words or tokens.
- Normalization: Converting all text to lowercase, removing punctuation, and handling special characters.
- Stop Words Removal: Removing common words that do not contribute much to classification (e.g., "and", "the").
- Stemming/Lemmatization: Reducing words to their root forms.

2. Feature Extraction

- Text Features: Extracting features from the email text, such as:
    - Term Frequency-Inverse Document Frequency (TF-IDF): Evaluating the importance of words in the email context.
    - Bag of Words (Bow): Creating a representation of text by counting the occurrences of each word.
- Metadata Features: Extracting features from email metadata, such as:
    - Sender Information: Email address, domain reputation.
    - Email Headers: Subject line, header flags.
- Behavioral Features: Features based on user behavior, such as:
    - Frequency: How often emails from a particular sender are marked as spam.

3. Model Training

- Data Splitting: Dividing the data into training and testing sets.
- Model Selection: Choosing an appropriate machine learning model for classification, such as:
    - Naive Bayes Classifier: Commonly used for text classification due to its simplicity and effectiveness.
    - Support Vector Machines (SVM): Effective for high-dimensional spaces.
    - Random Forest: Ensemble method for improved accuracy.

    - Neural Networks: Deep learning models like recurrent neural networks (RNN) or transformers for more complex patterns.
- Training: Using the training set to train the selected model.
- Evaluation: Evaluating the model on the testing set using metrics like accuracy, precision, recall, and F1 score.

4. Spam Filtering Engine

- Email Classification: Using the trained model to classify incoming emails as ham or spam.
- Real-time Processing: Implementing real-time processing for immediate email classification upon arrival.
- Feedback Loop: Collecting user feedback on classification results to continuously improve the model.

5. Deployment and Integration

- Integration with Email Servers: Deploying the spam filter as part of the email server infrastructure.
- APIs: Providing APIs for other systems to utilize the spam classification service.
- Monitoring and Logging: Monitoring the system for performance and logging results for future analysis.

6. User Interface

- Admin Dashboard: Interface for administrators to monitor and manage the spam filter system.
- User Feedback Mechanism: Allowing users to mark emails as ham or spam, improving the system's accuracy over time.

Unit Test Cases

Test Case Description	Pre-Requisite	Expected Result
Verify whether the Application URL is accessible to the user	1. Application URL should be defined	Application URL should be accessible to the user
Verify whether the Application loads completely for the user when the URL is accessed	1.Application URL is accessible 2.Application is deployed	The Application should load completely for the user when the URL is accessed
Verify whether user is able to see the dashboard	1. Application is accessible	The dashboard should visible to the user
Verify whether user is able to see input fields	Application is accessible	The user should be able to enter data
Verify whether user is able to edit all input fields	Application is accessible	User should be able to edit all input fields
Verify whether user gets Submit button to submit the inputs	Application is accessible	User should get Submit button to submit the inputs
Verify whether user is presented with recommended results on clicking submit	Application is accessible	User should be presented with recommended results on clicking submit