LOW LEVEL DESIGN

Thyroid Disease Detection System

By- HariShanker&Abishek Raghav

Document Version Control

- **Change Record:**

Version	Date	Author	Comments
1.0	8-07-2024	Hari Shanker & Abishek Raghav	Architecture

- **Reviews:**

Version	Date	Reviewer	Comments

- **Approval Status:**

Version	Review Date	Reviewed By	Approved By	Comments

Contents

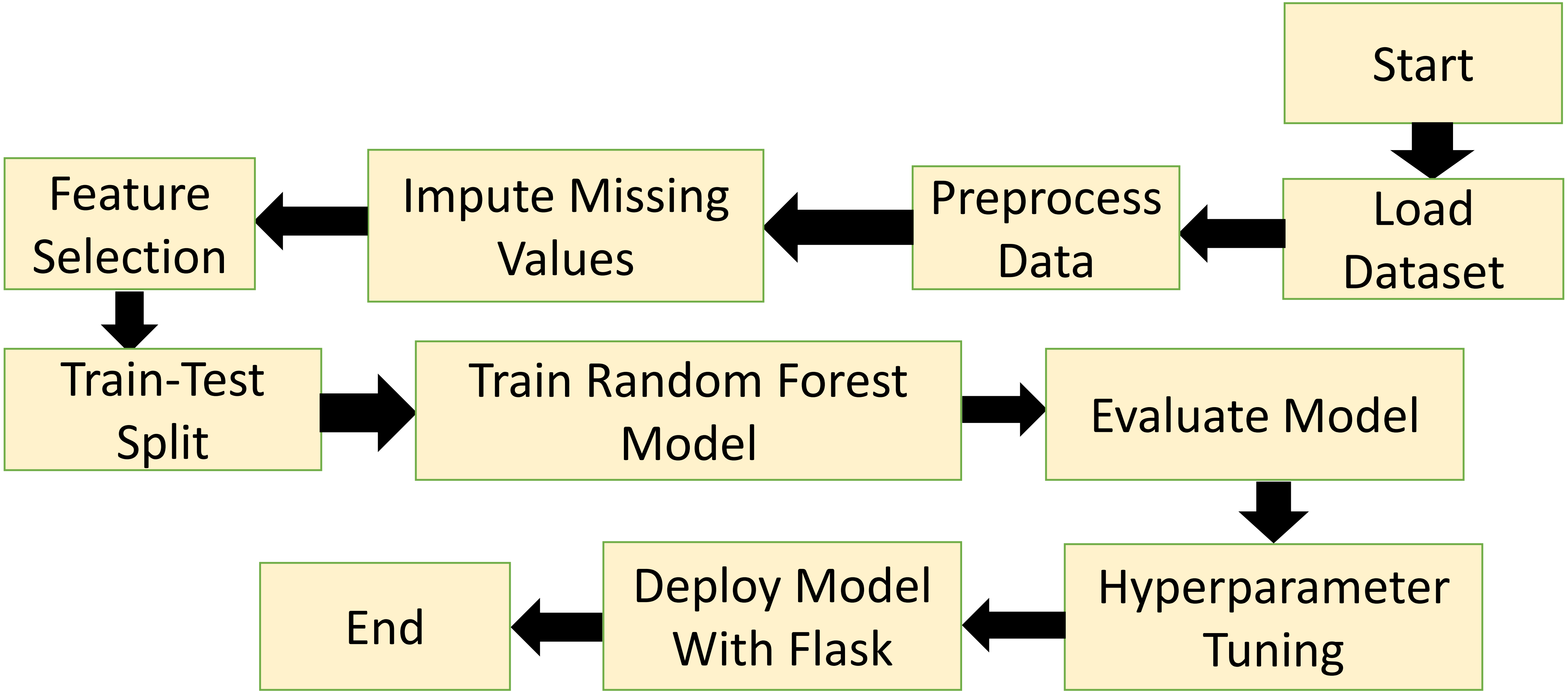
Introduction
1. What is Low-Level Design Document?

- A Low-Level Design Document (LLD) is a detailed technical document that outlines the internal logical design of a software program or system.
- In the context of the Thyroid Disease Detection System, the LLD would specifically focus on providing a comprehensive design for the actual program code of the system.
- The primary goal of the LLD is to offer a blueprint for the software's implementation, describing the various components, classes, methods, and their relationships with each other.
- It also includes specific program specifications, ensuring that programmers can directly translate the design into executable code using the document as a reference.

1.2 Scope:

- The scope of this LLD document includes:
- Detailed design of the thyroid prediction system components
- Description of the implementation logic for each module
- Class diagrams and interface definitions
- Database schema and table descriptions
- Test cases for validating the functionality of the system

2.Architecture:



Architecture Description: Thyroid Disease Prediction

2.1 Components and Modules:

- Each component and module in the system plays a crucial role in ensuring accurate and efficient predictions. Here are detailed explanations of each:

2.2 Data Collection:

- Data for this project is collected from a reputable source, which includes various medical records and features pertinent to thyroid diseases. The dataset is comprehensive and includes features such as TSH, T3, T4U, and FTI.

2.3 Data Preprocessing:

- Preprocessing is crucial to ensure the data is clean and suitable for training the model.
- **Impute Missing Values:** Handle missing values using methods like mean, median, or mode imputation.
- **Normalize Data:** Scale the features to ensure uniformity.
- **Encode Categorical Variables:** Convert categorical variables into numerical ones using techniques like one-hot encoding.

2.4 Feature Selection:

- Feature selection involves choosing the most relevant features for training the model. Techniques like correlation analysis and feature importance from models like Random Forest are used to select these features.

2.5 Model Training:

- A Random Forest model is trained on the preprocessed data. Random Forest is chosen due to its robustness and high accuracy in classification problems. The training involves:

2.6 Model Evaluation:

- he trained model is evaluated on the testing dataset using metrics like accuracy, precision, recall, and F1-score. The evaluation helps in understanding the model’s performance and areas of improvement.

2.7 Hyperparameter Tuning:

- Hyperparameter tuning is performed to optimize the model’s performance. Techniques like Grid Search and Random Search are used to find the best combination of hyperparameters.

2.8 Model Saving:

- The final model, along with the preprocessing steps, is saved using Python’s pickle module. This saved model is then used for making predictions in the Flask application.

2.9 Deployment with Flask:

- Flask is used to create a web application that serves the predictive model. The steps include:
- Creating Flask routes for different functionalities.
- Loading the saved model.
- Setting up a user interface for input and displaying predictions.

2.9.1 Application Workflow:

- The overall workflow of the application includes:
- User input data through the web interface.
- Data is preprocessed using the same steps as during training.
- The preprocessed data is fed to the model for prediction.
- The prediction result is displayed to the user and stored in the Cassandra database.

2.9.2 Predictions:

- The application provides accurate predictions based on the input data. The results are stored and can be retrieved for further analysis.

3. User Interface

- 1. For prediction, we will make a separate UI which will take all inputs from a single user and give back the prediction there only.

4 Unit Test Cases:

Test Case Description	Pre-requisite	Expected Result
Verify data preprocessing step	Raw data with missing values and categorical features	Data should be cleaned, missing values imputed, and categorical features encoded
Verify feature selection process	Preprocessed data	Relevant features should be selected based on correlation and feature importance
Verify model training process	Preprocessed and feature-selected data	Random Forest model should be trained and saved successfully
Verify model evaluation metrics	Trained model and test data	Model evaluation metrics like accuracy, precision, recall, and F1-score should be computed

4 Unit Test Cases:

Description	Pre-requisite	Expected Result
Verify hyperparameter tuning	Trained model and predefined hyperparameter ranges	Best hyperparameters should be selected based on grid search or random search
Verify model saving functionality	Trained model	Model should be saved using pickle module
Verify Flask deployment	Saved model and Flask setup	Flask application should load the model and provide an interface for input and prediction
Verify application workflow	Fully deployed Flask application	User inputs should be processed, predictions made, and results stored and displayed
Verify prediction accuracy	Test dataset	Model should predict thyroid conditions accurately
Verify user interface functionality	Flask application with UI components	User interface should accept inputs and display predictions correctly
Verify error handling	Flask application with potential error scenarios	Application should handle errors gracefully and provide meaningful messages

4. Unit Test Case

Verify performance under load	Deployed application	Application should handle multiple requests simultaneously without significant performance degradation