LOW LEVEL DESIGN (LLD)

THYROID DISEASE DETECTION

**Low-Level Design (LLD) for Thyroid Disease Detection Project**

**CONTENT**

1. Introduction

2. Detailed Component Breakdown

- 2.1 Data Preprocessing

- 2.2 Model Training and Evaluation

- 2.3 Model Deployment

- 2.4 API Design

- 2.5 Frontend (Streamlit App)

- 2.6 Error Handling and Logging

- 2.7 File Input and Output

3. Database Design (if applicable)

4. API Endpoints and Request-Response Format

5. Data Flow Diagrams (DFD)

6. Constraints and Assumptions

7. Detailed Performance Metrics

8. Conclusion

**1. Introduction**

A Low-Level Design (LLD) provides an in-depth, technical breakdown of the components and interactions in the Thyroid Disease Detection project. The LLD focuses on the implementation details of individual modules, data flow, and the interaction between different components, ensuring that the design can be easily implemented by developers.

**2. Detailed Component Breakdown**

**2.1 Data Preprocessing**

Objective:

Transform raw patient data into a clean format suitable for model training and prediction.

Steps:

1. Input Handling: Accept JSON or Excel files using pandas.

2. Data Cleaning:

- Handle missing values (imputation with mean, median, or constant values).

- Convert categorical variables (e.g., `sex`, `referral source`) into numerical using one-hot encoding or label encoding.

3. Feature Scaling: Use standardization (mean = 0, variance = 1) for continuous features (e.g., `TSH`, `T3`, `TT4`).

4. Feature Selection: Select relevant features like `age`, `sex`, `TSH`, `T3`, `on\_thyroxine`, etc., based on feature importance or domain knowledge.

Libraries:

- `pandas` for data loading and preprocessing.

- `scikit-learn` for scaling and encoding.

2.2 Model Training and Evaluation

Objective: Build and evaluate the Random Forest model for thyroid disease classification.

Model Building:

1. Algorithm: Random Forest Classifier from `scikit-learn`.

2. Hyperparameter Tuning: Use GridSearchCV or RandomizedSearchCV to optimize hyperparameters like `n\_estimators`, `max\_depth`, and `min\_samples\_split`.

3. Cross-Validation: Perform k-fold cross-validation to ensure the model generalizes well across different data splits.

Evaluation Metrics:

- Accuracy, precision, recall, F1 score, and ROC-AUC.

- Model Storage:

- Save the trained model using `pickle` or `joblib` for reuse during prediction.

**2.3 Model Deployment**

Objective:

Deploy the trained model using a Flask API for inference.

Steps:

1. Load Model: Use `pickle.load() ` to load the trained model during API initialization.

2. Predict Function: Create an endpoint in Flask that accepts JSON data and returns the predicted class (e.g., `hyperthyroidism`, `hypothyroidism`, or `normal`).

3. Preprocessing during Inference: Apply the same preprocessing steps as used in training before passing the data to the model for prediction.

Libraries:

- `Flask` for building the API.

- `pickle` for loading the model.

**2.4 API Design**

Objective: Provide RESTful API endpoints for data input and prediction.

Endpoints:

1. /predict (POST): Accepts patient data in JSON format and returns the predicted disease class.

Request Format:

```json

{

"age": 40,

"sex": "F",

"TSH": 1.2,

"T3": 0.9,

...

}

```

Response Format:

```json

{

"prediction": "hypothyroidism"

}

```

2. /status (GET): Returns the status of the API (e.g., "API is running").

Libraries:

- `Flask` for API development.

- `requests` for testing the API.

**2.5 Frontend (Streamlit App)**

Objective: Provide a user-friendly interface for file upload and display prediction results.

Components:

1. File Upload: Allow users to upload JSON or Excel files.

2. Submit Button: Trigger the backend API to send the uploaded data for prediction.

3. Display Results: After receiving the prediction from the API, display it in the UI with appropriate formatting.

4. Go Back Button: Clear the previous results and allow new uploads.

Code Structure:

- Use `st.file\_uploader()` for file upload.

- Send the file to Flask API using `requests.post()` after converting it to the appropriate format.

- Display results using `st.write()` or `st.markdown()`.

**2.6 Error Handling and Logging**

Objective: Ensure robust error handling and logging across the project.

Error Handling:

1. API Input Validation: If invalid input data is received (e.g., missing required fields, wrong data type), return a 400 response with a meaningful error message.

2. Try-Except Blocks: Surround critical code blocks (file upload, model prediction) with try-except to catch runtime errors and prevent application crashes.

Logging:

1. File Logging: Use Python’s `logging` module to log API requests, predictions, errors, and other critical events to a file.

2. Error Tracking: Log detailed error messages, stack traces, and timestamps for easier debugging.

**2.7 File Input and Output**

Objective: Handle file uploads and ensure correct data extraction for further processing.

Steps:

1. File Upload: Support both JSON and Excel file formats for data input.

2. Data Parsing:

- For JSON: Use `pandas. read\_json ()` to load the data.

- For Excel: Use `pandas.read\_excel()` to extract the relevant data fields.

3. Validation: Ensure the uploaded file contains all the required fields (age, sex, TSH, etc.).

Output:

- Provide the results (predictions) either as plain text (for API) or as formatted output (in the Streamlit UI).

3. Database Design (if applicable)

Though this project does not require persistent storage, future iterations might include:

- Patient Data: Store historical data, predictions, and timestamps for auditing.

- Logging Table: Record logs of API calls, errors, and predictions.

4. API Endpoints and Request-Response Format

| Endpoint | Method | Description | Request Body | Response Body |

|--------------|------------|-----------------|------------------|-------------------|

| /predict | POST | Accepts patient data and returns prediction | JSON/Excel data | {"prediction": "hypothyroidism"} |

| /status | GET | Returns API status | - | {"status": "API is running"} |

5. Data Flow Diagrams (DFD)

Level 0: Overall System

- Actors: User (uploads data) → System (preprocesses data, makes predictions) → User (receives prediction)

**Level 1: API Interaction**

1. User uploads JSON/Excel data.

2. Streamlit sends the file to Flask API.

3. API preprocesses the data, makes predictions, and sends the result back to Streamlit.

4. Streamlit displays the result to the user.

6. Constraints and Assumptions

**- Constraints:**

1. The project currently only supports batch input and not real-time streaming.

2. Model predictions are limited to the predefined features.

**- Assumptions:**

1. Input files (JSON/Excel) are formatted correctly and contain all required fields.

2. The current version does not include external databases or long-term data storage.

**7. Detailed Performance Metrics**

- Response Time: API is designed to respond within 1-2 seconds for individual predictions.

- Memory Usage: Efficient model loading with `joblib` ensures minimal memory consumption during inference.

- Prediction Accuracy: Precision, recall, and F1 score are tracked for overall model performance, aiming for balanced performance on all metrics.

**8. Conclusion**

This Low-Level Design (LLD) provides the detailed implementation blueprint for the Thyroid Disease Detection Project. By breaking down each component—data preprocessing, model training, API design, and frontend interaction—this document ensures that every part of the system is clearly defined and easily implementable. It also highlights error handling, performance optimization, and potential future expansions.