LOW LEVEL DESING

WAFER FAULT DETECTION

JENNA RAVEENKUMAR

1. General Description

1.1 Product Perspective

The Wafer Fault Detection System is designed to be a predictive tool integrated into the semiconductor manufacturing process. The system will take sensor data from wafers as input and output a prediction indicating whether the wafer is faulty (-1) or functional (+1). The model will be part of a larger quality control and monitoring system, helping engineers make informed decisions about wafer replacement.

1.2 Existing System

Currently, wafer inspection in semiconductor manufacturing may involve manual checking, rule-based systems, or basic threshold alerts based on sensor data. These methods are often inefficient and prone to human error, leading to either excessive replacement of functioning wafers or missed identification of faulty ones.

1.3 Proposal System

The proposed system will use machine learning to automatically predict the condition of each wafer based on historical and real-time sensor data. The system will reduce the need for manual inspection, increase accuracy, and allow for early detection of faults, thereby reducing production downtime and material waste.

2. Project Objectives

2.1 Accurate Fault Detection:

Develop a machine learning model to classify wafers as fault or functional based on sensor data

2.2 Minimize the downtime:

Improve manufacturing throughput by reducing the time spent in manual inspections.

2.3 Dashboard Reports:

Deliver real-time file validation reports to our clients, providing valuable insights into which files have failed validation and offering predictive data and also training-related data to our data science team to enhance understanding and improve performance.

2.4 Feedback and Retraining:

Incorporate client feedback on predictions, retrain the model, and continuously improve performance.

2.5 Understanding Data:

Wafer Type	Crystal Structure	Applications
Silicon	Covalent lattice	Logic circuits, microprocessors
Gallium Arsenide	Zinc-blende	RF devices, microwave amplifiers
Silicon Carbide	Tetrahedral	Power electronics, electric vehicles

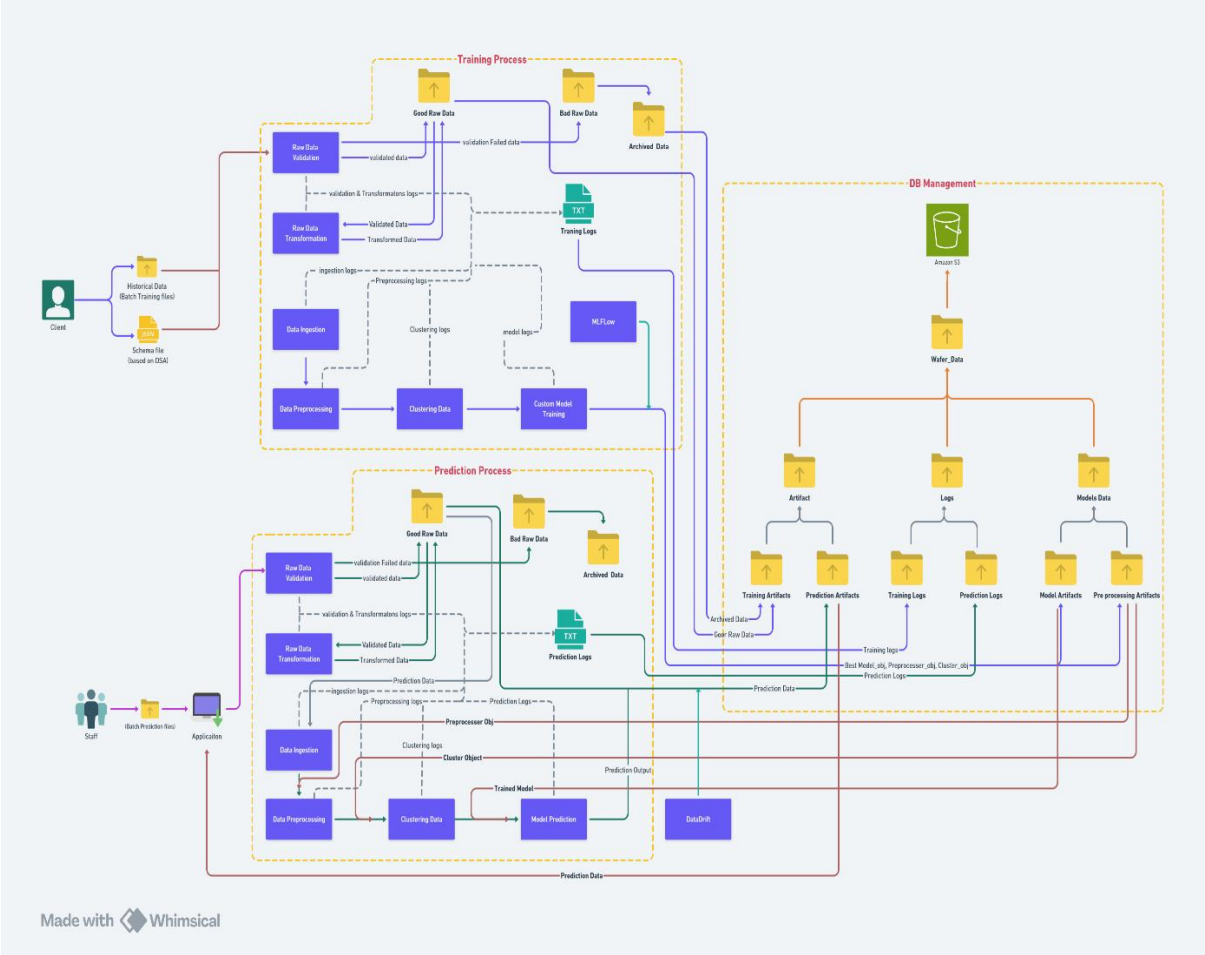
- Silicon wafers with crystalline structures remain the backbone of electronics, excelling in logic circuits, microprocessors, and memory applications.

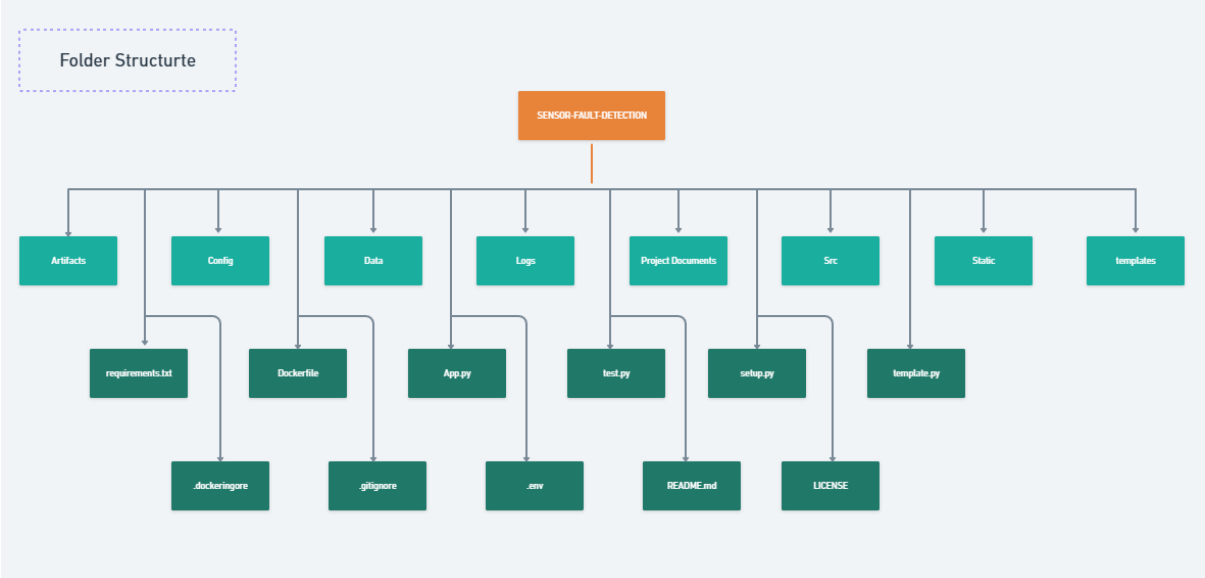
- Microelectronics: Silicon wafers are the foundational material for modern microelectronic devices. Silicon is the default in everything from microprocessors and memory chips to basic transistors.
- Solar Cells: Silicon's ability to convert sunlight into electricity efficiently makes it a popular choice for photovoltaic cells.
- Sensors and MEMS Devices: Silicon's mechanical properties and compatibility with microfabrication techniques make it suitable for micro-electromechanical systems (MEMS) and sensors, including pressure sensors and accelerometers.
- *RF Devices*: GaAs wafer's high electron mobility is ideal for the demands of RF applications.
- Microwave Amplifiers: GaAs takes a prominent role in the microwave spectrum, where its capabilities amplify signals with finesse and low noise.
- Optoelectronics: GaAs finds application in optoelectronic devices, facilitating seamless interaction with photons, leading to the creation of light-emitting diodes (LEDs) and laser diodes.
- Power Electronics: SiC wafers demonstrate superior efficiency and power density in applications like inverters and converters.
- Electric Vehicle Propulsion: SiC's high-temperature tolerance and efficiency enhance range and performance.
- High-Temperature Environments: From aerospace applications to industrial settings, SiC's resilience in high-temperature conditions establishes it as the material in harsh environments.

2.6 Dataset Description:

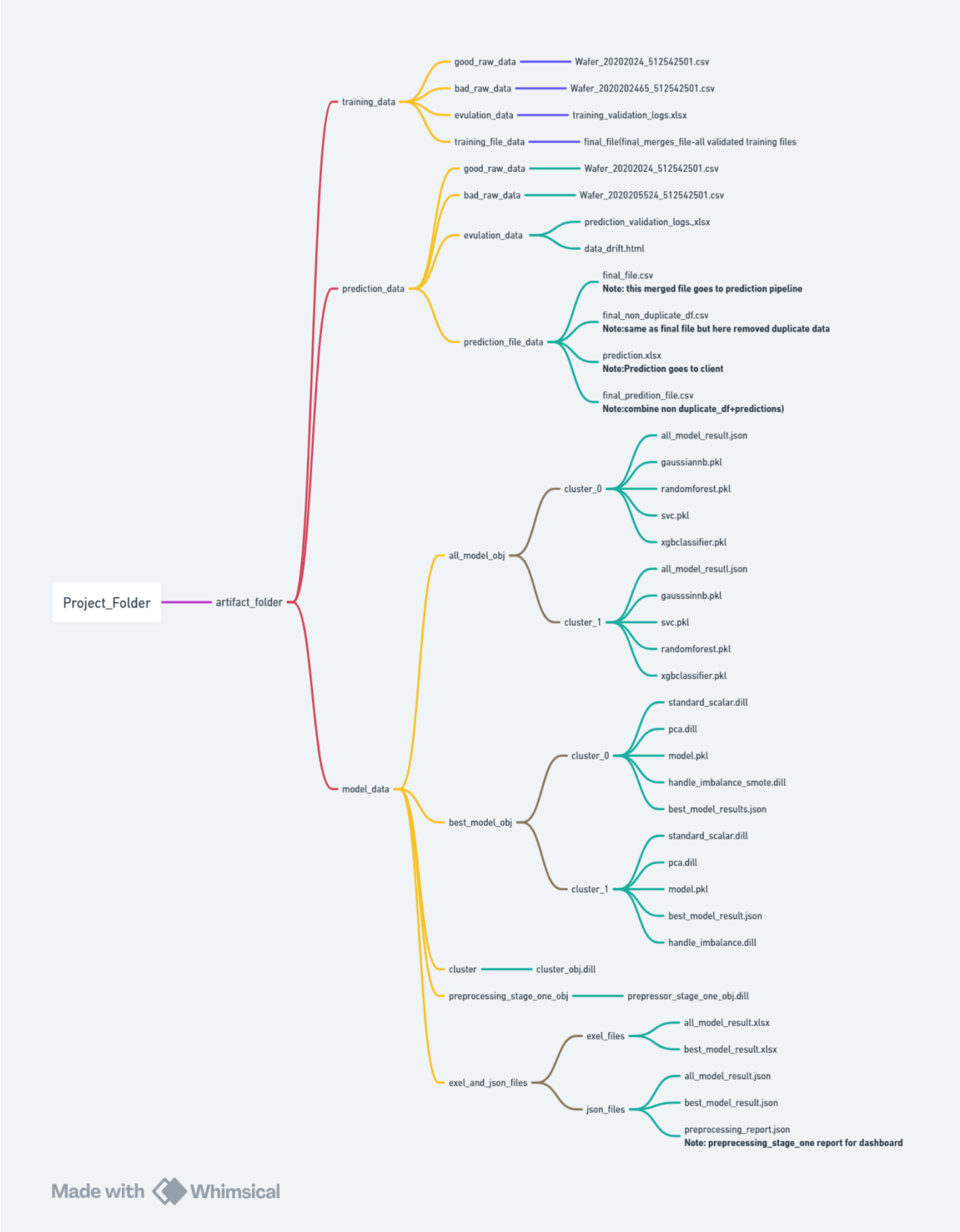
- The client will send data in multiple sets of files in batches at a given location. Data will contain Wafer names and 590 columns of different sensor values for each wafer. The last column will have the "Good/Bad" value for each wafer.
- "Good/Bad" column will have two unique values +1 and -1.
- "+1" represents Bad wafer.
- "-1" represents Good Wafer.

3. Architecture Overview




Folder Structurte
SENSOR-FAULT-DETECTION
Artifacts
Config
Data
Logs
Project Documents
Src
Static
templates
requirements.txt
Dockerfile
App.py
test.py
setup.py
template.py
.dockeringore
.gitignore
.env
README.md
LICENSE

3.1. Artifact Folder Overview:



The **Artifact folder** is used to store project artifacts, including data related to training, predictions, and the prediction model. This folder is generated at the start of the pipeline, and it

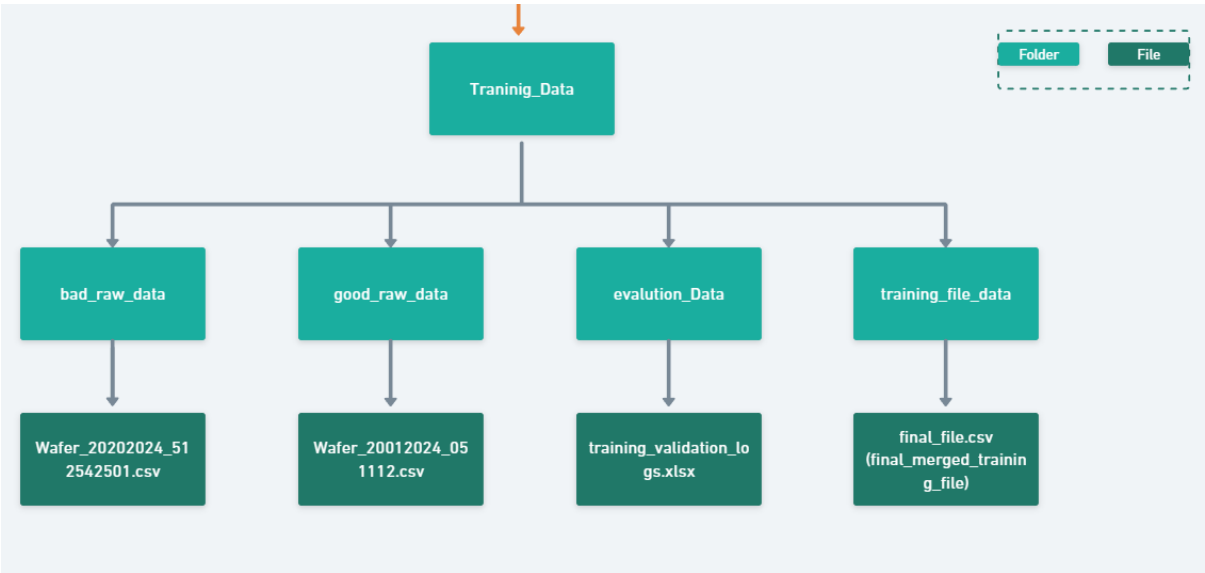
organizes data into specific subfolders. Upon completion, the data is uploaded to the cloud, and the local Artifact folder is deleted.

The **Artifact folder** contains three primary subfolders:

1. **Training_data folder**
2. **Prediction_data folder**
3. **Model_data folder**

Each of these folders is created dynamically as the pipeline executes, and they store data at different stages of the process.

3.1.1 Training_data Folder:



When training data is fetched from the database, it passes through the **training pipeline**, which consists of several stages:

- During this stage, the raw data is divided into two categories:
  - **Good_Raw_Data**: Data that successfully passes validation.
  - **Bad_Raw_Data**: Data that fails validation due to inconsistencies or errors.
- Both categories are stored in their respective subfolders.
- A detailed summary of the validation process is saved in a file named **training_validation_logs.xlsx** within the **evaluation_data** folder.
- This log captures:
  - The total number of files validated.
  - The number of files that passed and failed at each validation stage.
  - Reasons for failure, if any.
- Once validation is successfully completed, the **Good_Raw_Data** proceeds to the **transformation step**.

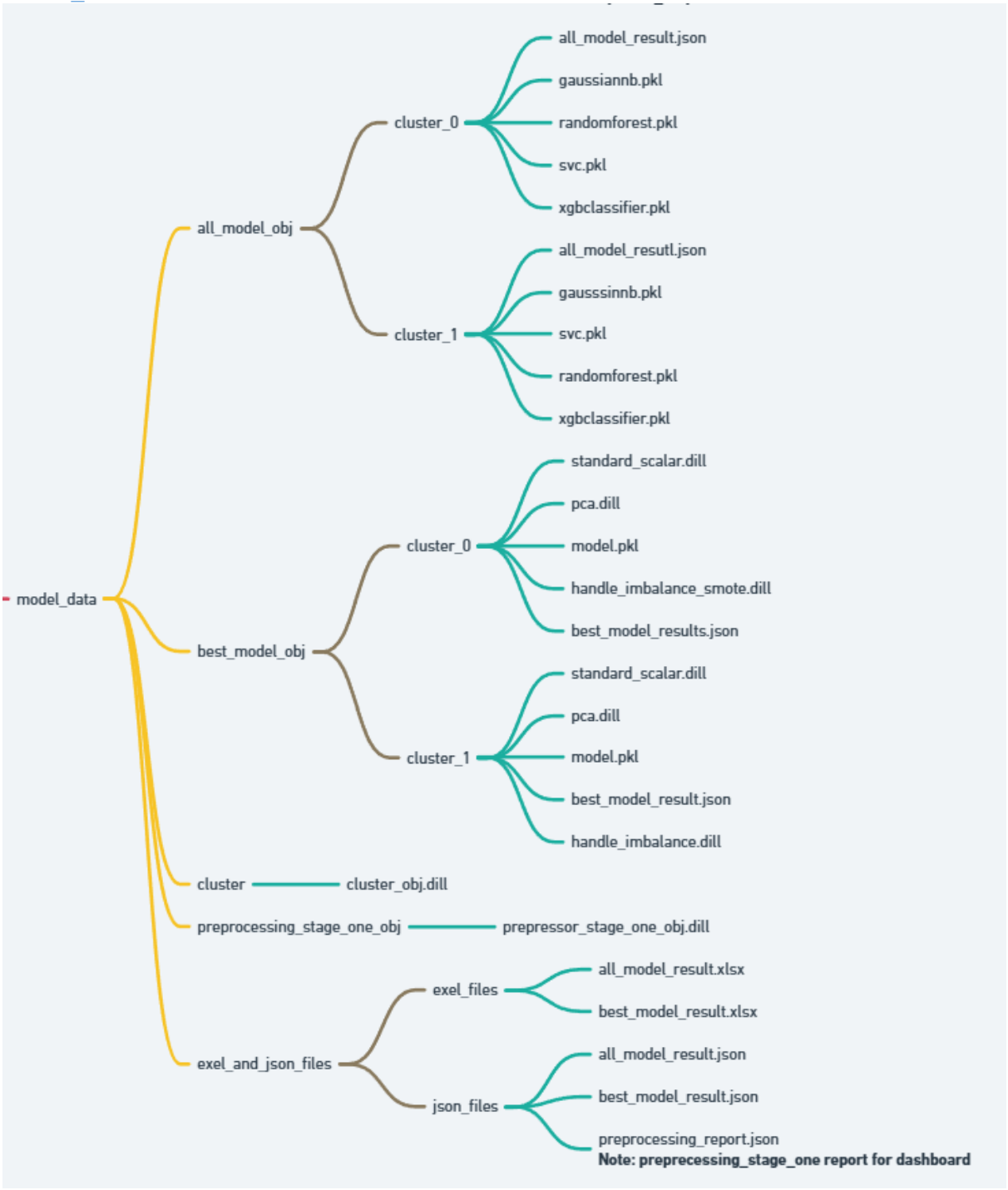
- In this step, all validated files are merged into a single consolidated file named **final_file.csv**, which is stored in the **training_file_data** folder.
- This merged file serves as input for the **training process**.

3.1.2 Prediction_data Folder:

When receiving prediction data from the client, the **Prediction Pipeline** is initiated. The data may arrive as a **single file or multiple files (batch mode)** and follows the same **validation rules applied during training**.

- The raw data is classified into two categories:
  - **Good_Raw_Data:** Data that passes all validation checks.
  - **Bad_Raw_Data:** Data that fails validation due to inconsistencies or errors.
- Both categories are stored in their respective **subfolders** for organized tracking.
- A **detailed validation log** is saved as prediction_validation_logs.xlsx within the **evaluation_data** folder, capturing:
  - Total number of files validated.
  - Number of files that passed and failed at each validation stage.
  - Reasons for any failures.
- **Data drift analysis** is conducted after the prediction process. The results are saved as **data_drift.html** in the **evaluation_data** folder.
- Once validation is complete, the **Good_Raw_Data** proceeds to the transformation stage:
  - All validated files are merged into a **single consolidated file**, final_file.csv, stored in the **prediction_file_data** folder.
- During **preprocessing**, duplicates are removed, and the **cleaned data** is saved as final_non_duplicate_file.csv in the **prediction_file_data** folder.
- After successful predictions, the results are stored in **predictions.xlsx** in the **prediction_file_data** folder.
- For retraining purposes, the **final_non_duplicate_file.csv** and **predictions.xlsx** are merged and saved as **final_prediction_file.csv** in the **prediction_file_data** folder.

3.1.3 model_data Folder:

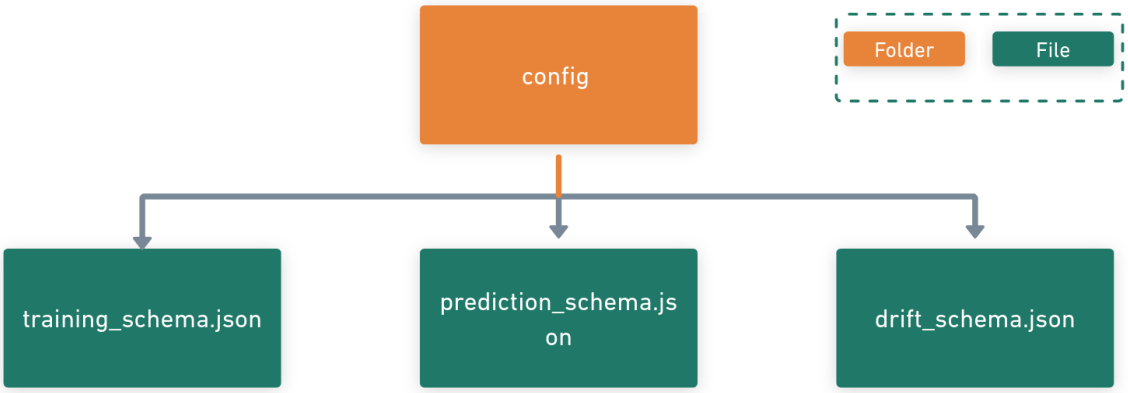


After successfully completing training, all **model-related data** is organized and stored within the **model_data folder** for easy access. The data is **categorized by clusters** and algorithms, ensuring a structured workflow.

4 The data was primarily trained using **four algorithms** with a **custom model-building approach involving clustering**.
5 Model data is stored **cluster-wise** in dedicated folders named **cluster_0, cluster_1**, etc.
6 **all_model_obj folder:**Contains all trained model objects organized by cluster.
7 **best_model_obj folder:**Stores the **best-performing model objects** for each cluster.
8 **cluster folder:**Contains the **clustering object** as cluster_obj.dill within the **model_data folder**.

9 During preprocessing, all logs are generated, capturing, Number of **duplicate files** handled,Number of **NaN columns** processed,**Skewness correction** details.**Outliers handled**.These logs are stored in **both Excel and JSON formats** within the **exel_and_json folder**.
10 The preprocessing objects from **Stage One** are saved in **preprocessing_Stage_one_obj folder**.
11 **Model results** (including evaluation metrics and other outputs) are also saved in **both Excel and JSON formats** for future reference, stored in the **exel_and_json folder**.

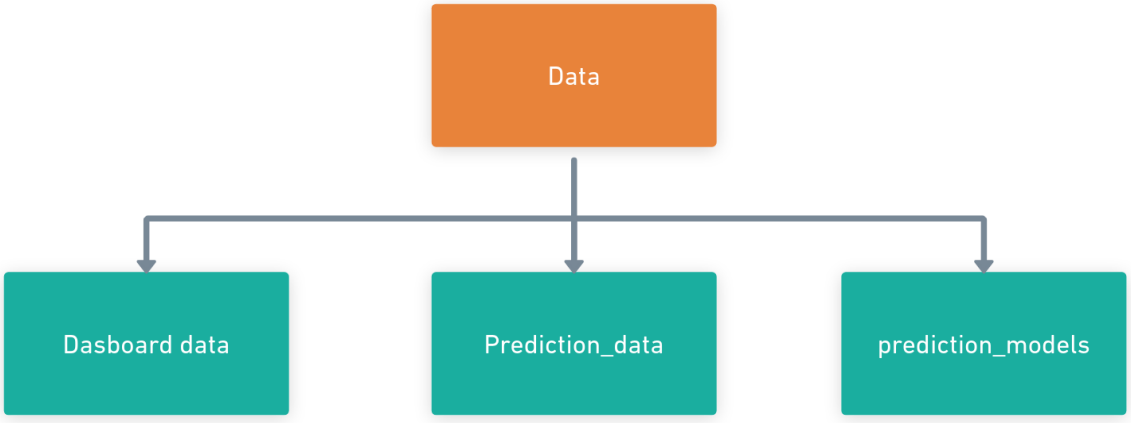
3.2 Config Folder Overview:



In compliance with the **Data Sharing Agreement**, We create **Master Data Management(MDM)** data is stored in a structured way to streamline the **training, prediction, and drift detection processes**. The relevant **schema configurations** are maintained in **three JSON files** within the **config folder** to ensure smooth execution of each stage.

- **Schema Files in the Config Folder:**
  1. **training_schema.json:**
     - Defines the schema for training data, including expected columns, data types, and validation rules.
  2. **prediction_schema.json:**
     - Specifies the structure for prediction data, ensuring compatibility with the model input requirements.
  3. **drift_schema.json:**
     - Specifies the structure for column mapping drift, ensuring compatibility with the model input requirements.

3.3 Data Folder Overview:



The **Data folder** stores essential outputs, models, and reports generated during the **training and prediction processes**. This structured organization ensures easy access to relevant data for **dashboard creation, reporting, and further analysis**. The folder is divided into three primary **subfolders:**

3.3.1. Dashboard Folder

- Stores all **dashboard reports** to facilitate data visualization and performance tracking.
- Contains files in **.json** and **.xlsx** formats for comprehensive analysis and insights.

3.3.2. Prediction_Data Folder

- Holds outputs generated during the **prediction process**.
- Key files:
  - **bad_raw.zip**:
    - Contains files that failed validation checks during prediction.
  - **prediction.xlsx**:

- Stores prediction outputs, including sensor results alongside their respective **sensor names**.

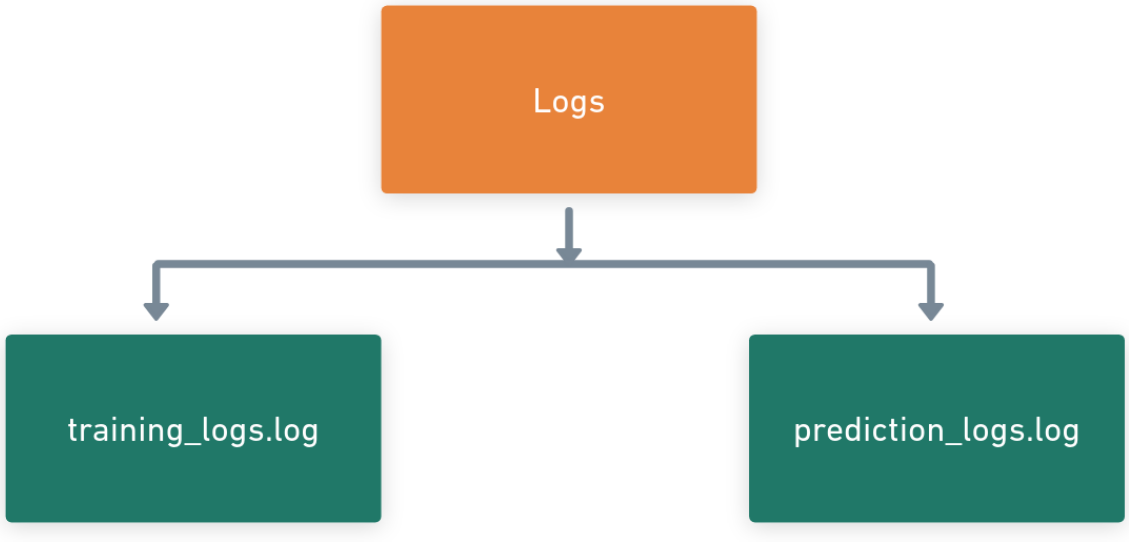
3.3.3. Prediction_Models Folder

- Stores the **best-performing model data** used for predictions.
- This folder ensures easy retrieval of optimal models for ongoing predictions and model monitoring.

3.3.4. ETag_data.json

- This file contains the ETag for all the best models stored in S3 buckets. This ensures that whenever there is an update to the best models in the cloud, the predictions are based on the most current models. Always retrieve the updated models to your local interface.

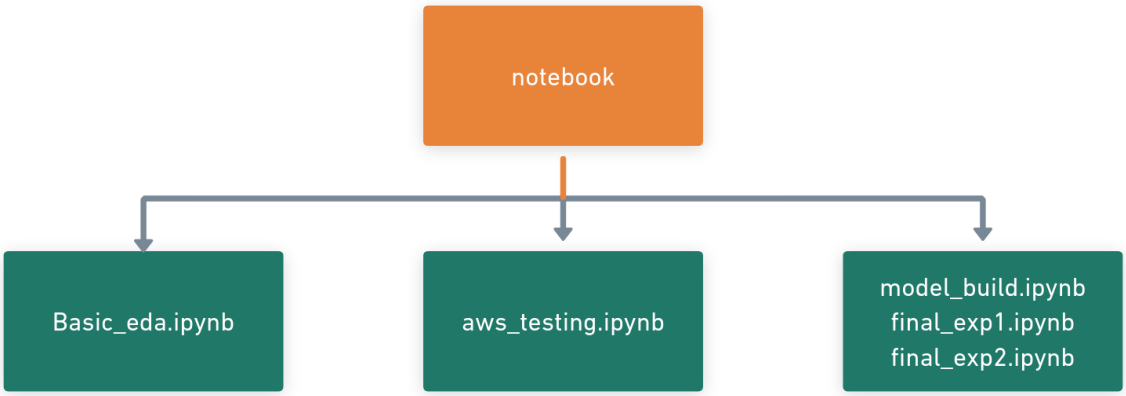
3.4 Logs Folder Overview:



The **Logs folder** stores all **log files** generated during the **training and prediction processes**. These logs provide detailed insights into the execution flow, including any errors, warnings, or key events, ensuring traceability and aiding in troubleshooting.

- **training_logs.log**:
  - Captures all events and processes during the **training phase**, including:
    - Data preprocessing steps (e.g., NaN handling, outlier detection).
    - Model training progress and metrics.
    - Any warnings, errors, or exceptions encountered.
    - Summary of training status and model performance.
    - MLFlow logs while dagshub connection etc.
- **prediction_logs.log**:
  - Logs all activities during the **prediction process**, such as:
    - Validation of input files.
    - Data transformation and predictions made by the model.
    - Any inconsistencies or errors during prediction.
    - Summary of prediction results and output status.
    - Data Drift related information also capture if drift detected.

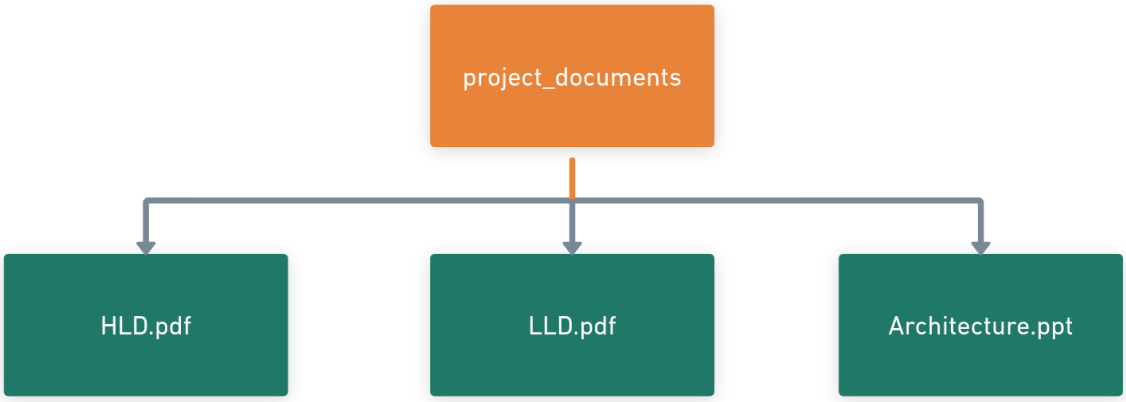
3.5 Notebook Folder Overview:



the **Notebook folder** stores all **Jupyter notebooks** used for **data exploration, experimentation, and model building**. This folder serves as a **repository of experiments and insights**, providing valuable reference points for future improvements and decision-making.

- **Data Exploration Notebooks:**
  - Notebooks used for **analysing data patterns, trends, and relationships** within the dataset.
  - Include steps such as:
    - Identifying skewness, missing values, and outliers.
    - Visualizing data distributions and feature correlations.
    - Performing feature engineering and transformation experiments.
- **Model Experimentation Notebooks:**
  - Capture all experiments related to **model selection and tuning**, including:
    - Comparison of different algorithms to determine the best fit.
    - Hyperparameter tuning and evaluation metrics.
    - Testing various pre-processing strategies (e.g., scaling, encoding, feature selection).
- **Database Connection Notebooks:**
  - Notebooks documenting **database integration experiments** to ensure smooth data ingestion and retrieval.
  - Include:
    - Connection setups (e.g., MongoDB and aws connection).
    - Queries and data loading procedures for model input.

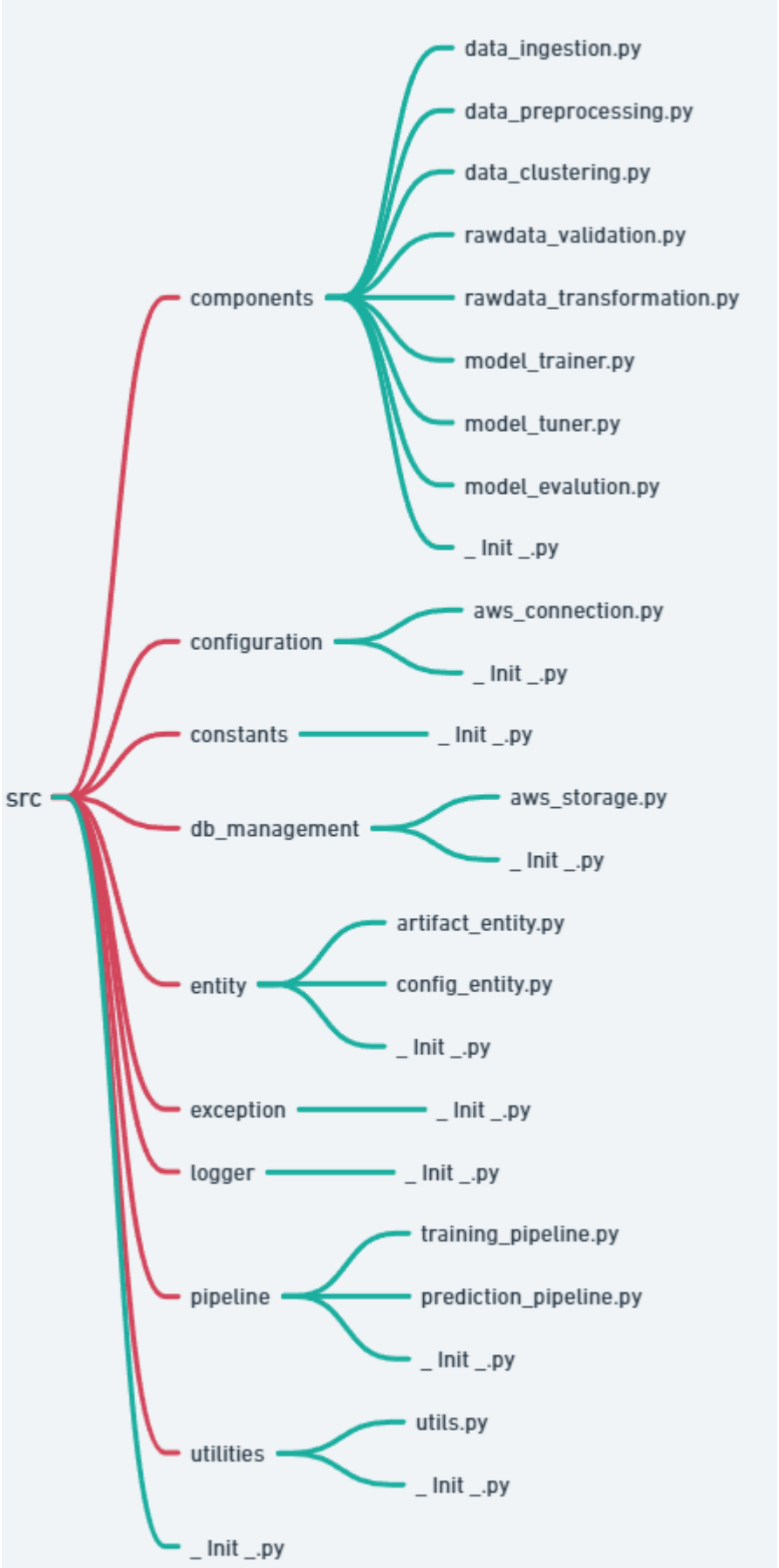
3.6 Project Documents Folder Overview:



The **Project_Document folder** is an optional but valuable resource for storing all **project-related documents**. It provides a centralized location for easy access to key documentation, ensuring that all stakeholders have the necessary references throughout the project lifecycle.

- **HLD (High-Level Design):**
  - Contains the **overall system design**, architecture diagrams, and a broad view of how components interact.
  - Useful for understanding the project's **structure, scope, and workflow**.
- **LLD (Low-Level Design):**
  - Provides detailed, component-level design, including:
    - **Class diagrams, workflows, and logic breakdowns**.
    - Information on **individual modules, APIs, and data flow** within the system.
- **Architecture Document:**
  - Documents the **architecture of the project**, explaining how **components, databases, and services** interact within the ecosystem.
  - Includes diagrams and justifications for architectural choices.
- **DPR (Detailed Project Report):**
  - A comprehensive report detailing
    - **Project goals, timelines, milestones**, and deliverables.
    - **Resource allocation** and risks identified.
    - **Key achievements and future plans**.

3.7 Src Package Overview:



The **src package** serves as the **main package** of the project, organizing all the essential processes and functionalities into specialized **sub-packages**. Each sub-package contains relevant **modules** or components to ensure the project is well-structured, maintainable, and modular. Below is a detailed breakdown of the **9 key sub-packages** and their roles:

3.7.1. components Package

- This package houses all the modules required for **custom pipeline building**.
- It includes modules responsible for:
  - **Raw Data Validation** (rawdata_validation.py)
  - **Data Transformation** (rawdata_transformation.py)
  - **Data Preprocessing** (data_preprocessing.py)
  - **Clustering** (data_clustering.py)
  - **Model Building** (model_trainer.py)
  - **Model Tuning** (model_tuner.py)
  - **Model Evaluation** (model_evaluation.py)
- These modules work together to create and manage the end-to-end data pipeline.

3.7.2. configuration Package

- Contains modules related to configuration and connection management.
- **aws_connection.py**: Manages **stable connections with AWS** services and **MongoDB**. This ensures that the project seamlessly interacts with cloud resources and databases.

3.7.3. constants Package

- Stores **all constant variables** used across the project inside **__init__.py**.
- **__init__.py** serves as the primary access point for these constants, making them easily importable throughout the project.
- **Purpose**:
  - If any variable (e.g., paths, folder names) requires an update, it can be changed **once in this package**, eliminating the need for updates in multiple modules.

3.7.4. db_management Package

- Contains **AWS-related operations**.
- **aws_storage.py**: Handles all storage operations on **AWS** (such as uploading or retrieving files from S3 buckets).

3.7.5. entity Package

- This package defines the structure for **configuration and artifacts** used throughout the project.
  - **config_entity.py**: Contains the **configuration settings** required by various modules.
  - **artifact_entity.py**: Defines the **artifact format** for outputs generated by the different modules.

3.7.6. exception Package

- This package is used to **generate and manage custom exceptions**.
- **All custom exception handling logic** is implemented in **__init__.py**, making it easy to maintain and update error-handling strategies.

3.7.7. logger Package

- Responsible for generating **custom logs** throughout the project.

- **__init__.py**: Contains all the logic for **logging messages**, including info, debug, warning, and error logs. This ensures that critical events are tracked during training and prediction processes.

3.7.8. pipeline Package

- Contains modules that define the **training and prediction pipelines**.
    - **training_pipeline.py**: Orchestrates the training process by **arranging components in the correct sequence** from the components package.
    - **prediction_pipeline.py**: Responsible for the **prediction process**, following a similar structure to the training pipeline but geared toward inference.

3.7.9. utilities Package

- Contains a module named **utils.py**, which stores **helper functions** and **utility methods** required across the project.
- These utility functions can be imported and used wherever needed, ensuring code reuse and reducing redundancy.

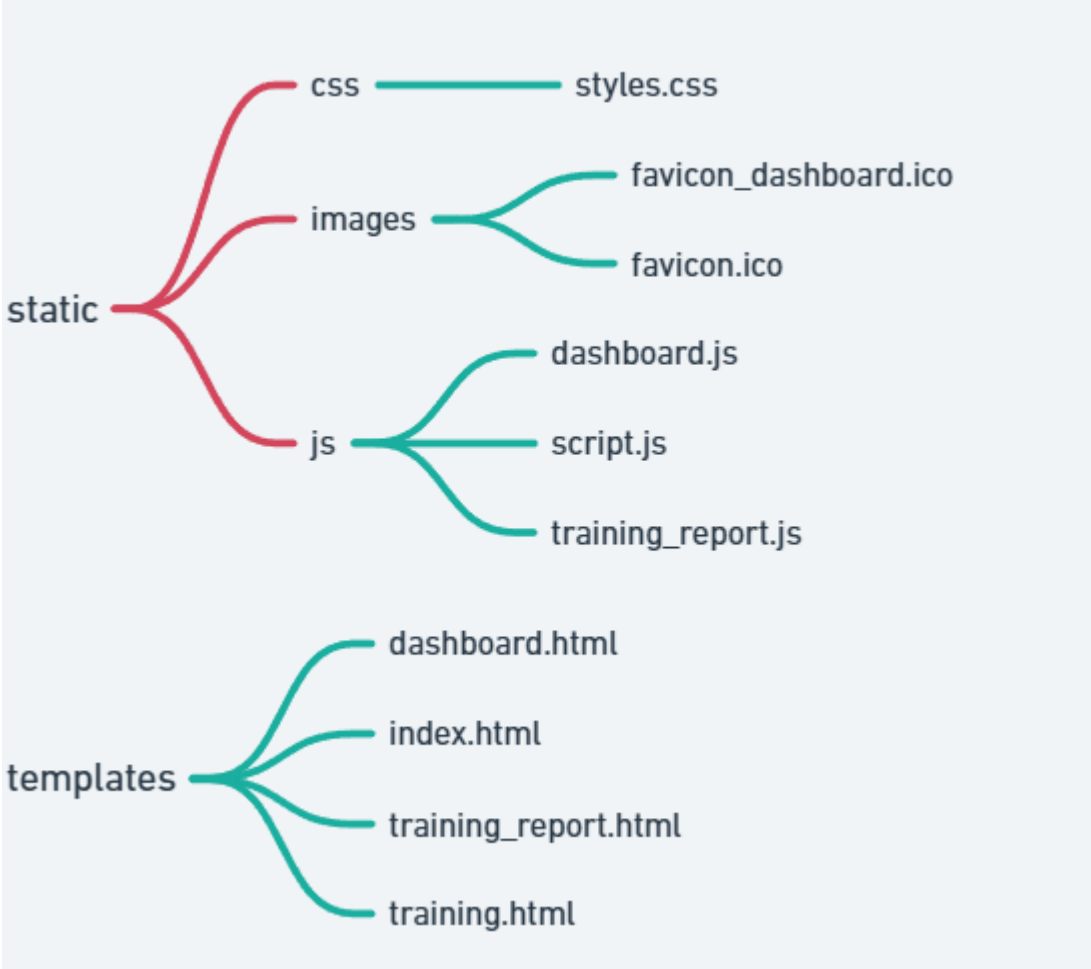
Summary

The **src package** is the heart of the project, containing all the necessary components, pipelines, configurations, and utilities to ensure smooth execution. This modular structure enhances **scalability, maintainability, and reusability**, making the codebase easy to manage and extend.

This design also ensures:

- **Centralized configuration management**
- **Clear separation of responsibilities** across different modules
- **Efficient handling of custom exceptions and logging**
- **Reusable utility functions** for cleaner code

3.8 Static and templates Folder Overview:



3.8.1. static

This directory contains static resources such as CSS, JavaScript, and images used throughout the application.

- **css/**
  - styles.css: Contains all the styling rules for the web pages.
- **images/**
  - favicon_dashboard.ico: Icon for the dashboard page.
  - favicon.ico: Default favicon for the application.
- **js/**
  - dashboard.js: JavaScript logic related to the dashboard page.
  - script.js: General-purpose JavaScript file used throughout the app.
  - training_report.js: JavaScript logic for generating and managing the training report.

3.8.2. templates/

This directory stores HTML templates rendered by the application's backend.

- **dashboard.html**: Template for the dashboard page.

- **index.html**: Main landing page or homepage of the application.
- **training_report.html**: Template for the training report page.
- **training.html**: Template for displaying training-related information or processes.

3.9 .dockerignore file Overview:

Specifies files and directories to ignore when building a Docker image.

4.0 .env file Overview:

Purpose: Stores environment variables for configuration, such as API keys or database credentials.
Sensitive: This file should be added to .gitignore to avoid leaking secrets.

4.1 app.py file Overview:

- **Purpose**: The main application entry point (e.g. FastAPI application).

4.2 requirements.txt file Overview:

Purpose: Lists all the dependencies required for the Python project.

4.3 setup.py file Overview:

Purpose: Used for packaging and distributing the project as a Python package.

4.4 template.py file Overview:

- **Purpose**: Could be used for create folder creation.

4.5 test.py file Overview:

- **Purpose**: Contains test cases to verify the functionality of the application.

Summary

This LLD represents a typical Python project with Docker support and environment configurations. It organizes the following:

- **Configuration Files**: .env, .gitignore, .dockerignore, requirements.txt, setup.py
- **Main Code**: app.py, template.py
- **Testing**: test.py
- **Deployment**: Dockerfile

4 Modules and Classes

4.1 Data_ingestion.py module:

4.1.1. Class: DataIngestion:

This class manages the **local ingestion of data** from a given input dataset path.

Attributes:

- input_dataset_path: The path to the input dataset (of type Path).

Methods:

1. __init__(self, input_dataset_path: Path) -> None

- Initializes the class with the given dataset path.

2. get_data(self) -> DataFrame

- **Functionality**:
  - Reads data from the provided input_dataset_path using read_csv_file().
  - Logs success or failure of the operation.
  - If an exception occurs, raises a custom exception (SensorFaultException).
- **Exceptions**:
  - Raises SensorFaultException on any error during data reading.

3. initialize_data_ingestion_process(self) -> DataIngestionArtifacts

- **Functionality**:
  - Orchestrates the data ingestion process by calling get_data().
  - Returns a DataIngestionArtifacts object, which encapsulates the input data for the next processing steps.

4.1.2. Class: GetTrainingData

This class handles the **S3 data retrieval and merging processes** required for training and retraining workflows. It interacts with S3 and manages both the file path selection logic and feedback incorporation.

Attributes:

- data_ingestion_config: Configuration object for data ingestion paths.
- s3_obj: An instance of SimpleStorageService for S3 operations.
- s3_config: Configuration object for S3 paths.
- s3_bucket_obj: An instance of Bucket to interact with the S3 bucket.

Methods:

1. __init__(self, data_ingestion_config, s3_obj, s3_config, s3_bucket_obj)

- Initializes the class with relevant configuration and S3 objects.

2. get_file_path(self) -> tuple[str, Literal['retraining', 'training', 'default_training']]

- **Functionality**:
  - Determines which **S3 folder** to use for training:
    - **Retraining**: If the retraining folder is not empty.
    - **Training**: If the training folder is not empty.
    - **Default Training**: If neither folder is available, use default data.
  - Logs the selected path and status (retraining, training, or default training).

- **Exceptions**:
    - Raises SensorFaultException if any error occurs during file path retrieval.

3. files_store_in_local_path(self, s3_files_path: str) -> Path

- **Functionality**:
    - Creates a local folder to store training files.
    - Downloads files from S3 to the local folder using the specified S3 path.
    - Logs success or failure.
- **Exceptions**:
    - Raises SensorFaultException if the download fails.

4. add_feedback_to_prediction_file(self, local_training_files_path: Path)

- **Functionality**:
    - Merges **prediction data** and **feedback data**:
        1. Reads feedback files and prediction files.
        2. Preprocesses feedback data.
        3. Merges feedback with the prediction file using the wafer_column_name key.
        4. Saves the merged file as the final prediction file.
        5. Deletes the temporary prediction and feedback files.
- **Exceptions**:
    - Raises SensorFaultException if the merging process fails.

5. initialize_getting_training_data_process(self) -> tuple[Literal['retraining', 'training', 'default_training'], Path]

- **Functionality**:
    - Orchestrates the training data retrieval process:
        1. Gets the relevant file path from S3 (get_file_path()).
        2. Downloads files to a local folder.
        3. If it's a retraining process, calls add_feedback_to_prediction_file() to incorporate feedback.
    - Returns the **status** (retraining, training, or default training) and the **local path** to the data.
- **Exceptions**:
    - Raises SensorFaultException on failure.

4.2 rawdata_validation Module:

This class validates raw data files before processing them for training or prediction. The validations include checking filenames, column counts, data types, and missing values. It classifies files as "good" or "bad" based on these checks.

4.2.1 Attributes:

1. **data_files_path** (Path): Path to the folder containing raw data files.
2. **config** (Union[TrainingRawDataValidationConfig, PredictionRawDataValidationConfig]): Configuration object holding paths and schema details.
3. **schema_file** (Dict): Schema information loaded from a JSON file.
4. **good_raw_data_path** (Path): Folder to store validated "good" files.
5. **bad_raw_data_path** (Path): Folder to store "bad" files.
6. **regex_file_name_format** (str): Regex pattern for filename validation.
7. **validation_report_file_path** (Path): Path for storing the validation report.
8. **dashboard_validation_report_file_path** (Path): Path for dashboard validation report.

4.2.2 Methods:

1. **__init__()**
   Purpose: Initializes the class with configuration and paths. Loads schema data from a JSON file.
   Parameters:
   - config: Config object with paths and schema details.
   - folder_path: Path to the folder containing raw data files.

---

2. **filename_validation(file_name: str) -> Literal['Passed', 'Failed']**
   Purpose: Validates the filename using a regex pattern.
   Parameters:
   - file_name: Name of the file to validate.
     Returns: 'Passed' or 'Failed'.
     Exception: Raises SensorFaultException for errors.

---

3. **numberofcolumns_validation(file_name: str, dataframe: pd.DataFrame) -> Literal['Passed', 'Failed']**
   Purpose: Validates if the number of columns in the file matches the schema.
   Parameters:
   - file_name: Name of the file for logging purposes.
   - dataframe: Dataframe object representing the file's data.
     Returns: 'Passed' or 'Failed'.
     Exception: Raises SensorFaultException for errors.

---

4. **columnsdata_validation(file_name: str, dataframe: pd.DataFrame) -> Literal['Passed', 'Failed']**
   Purpose: Validates column names and data types against the schema.
   Parameters:
   - file_name: Name of the file for logging.
   - dataframe: Dataframe containing column data.
     Returns: 'Passed' or 'Failed'.
     Exception: Raises SensorFaultException for errors.

---

5. **columndata_whole_missing_validation(file_name: str, dataframe: pd.DataFrame) -> Literal['Passed', 'Failed']**
   Purpose: Checks if any column in the file has entirely missing data.
   Parameters:
   - file_name: Name of the file for logging.
   - dataframe: Dataframe with the file's data.
     Returns: 'Passed' or 'Failed'.
     Exception: Raises SensorFaultException for errors.

---

6. **initialize_rawdata_validation_process() -> RawDataValidationArtifacts**
   Purpose: Starts the validation process for all files in the folder. Moves files to appropriate folders based on validation results.

Returns: An instance of RawDataValidationArtifacts.
Exception: Raises SensorFaultException for errors.

4.2.3 Supporting Operations:

1. **create_folder_using_folder_path()**: Creates folders if they don't exist.
2. **append_log_to_excel()**: Logs validation results to an Excel file.
3. **read_json()**: Reads schema details from a JSON file.
4. **read_csv_file()**: Reads raw data files into DataFrames.
5. **create_zip_from_folder()**: Zips bad raw data for reporting.

4.2.4 Workflow of initialize_rawdata_validation_process():

1. Logs the start of the validation process.
2. Creates folders for "good" and "bad" data, and the validation report.
3. Iterates through all files in the raw data folder:
    - **Filename validation** → Move to "bad" folder if failed.
    - **Number of columns validation** → Move to "bad" folder if failed.
    - **Missing column data validation** → Move to "bad" folder if failed.
    - **Column data validation** → Move to "bad" folder if failed.
4. If all validations pass, moves the file to the "good" folder.
5. Creates a zip of bad data (for predictions only).
6. Returns validation artifacts with folder paths and log file details.

4.2.4 Exception Handling:

- **SensorFaultException**:
    - Raised for any errors encountered during validation.
    - Logs the error message and stack trace.

4.3 rawdata_transformation Module:

4.3.1. Class: RawDataTransformation

- **Purpose**: This class handles the transformation of validated raw data by merging multiple CSV files into one, reformatting the target variable, renaming columns, and storing the result as a single file.

4.3.1.1 Constructor: __init__()

- **Input Parameters**:
    - config: Union of TrainingRawDataTransformationConfig and PredictionRawDataTransformationConfig.
    - rawdata_validation_artifacts: Contains metadata such as the path to good raw data.
- **Functionality**:
    - Initializes paths for data transformation (e.g., raw data folder and merged file path) based on provided configurations and artifacts.

4.3.1.2 Method: convert_good_raw_into_single_file()

- **Input**:
  - input_folder: Path to the folder containing validated raw CSV files.
- **Output**: Merged DataFrame from all CSV files in the input folder.
- **Exception Handling**: Raises SensorFaultException if any error occurs.
- **Steps**:
  - Iterate over all files in the folder.
  - Read valid CSV files into individual DataFrames.
  - Concatenate all DataFrames into a single DataFrame.
  - Log success or failure.

4.3.1.3 Method: reformat_target_variable()

- **Input**:
  - merge_df: Merged DataFrame from the previous step.
- **Output**: Updated DataFrame with transformed target values.
- **Functionality**:
  - Converts target values from [-1, 1] to [0, 1] to align with model conventions.
- **Exception Handling**: Raises SensorFaultException on error.

4.3.1.4 Method: rename_column_names()

- **Inputs**:
  - merge_df: Merged DataFrame.
  - output_file: Path to store the final output.
- **Functionality**:
  - Renames columns such as the wafer and target variable to their standardized names.
  - Saves the updated DataFrame to a new CSV file.
- **Exception Handling**: Logs and raises SensorFaultException on error.

4.3.1.5 Method: initialize_data_transformation_process()

- **Output**: RawDataTransformationArtifacts containing the path to the final transformed file.
- **Steps**:
  1. Create the folder for storing the merged file.
  2. Call convert_good_raw_into_single_file() to merge the data.
  3. Apply reformat_target_variable() to ensure the target values align with conventions.
  4. Use rename_column_names() to standardize column names.
  5. Return the artifacts containing the path of the merged output file.
- **Exception Handling**: Logs and raises SensorFaultException if the process fails.

4.3.1.6 Error Handling Approach

- **Custom Exception**: SensorFaultException used consistently to handle and log errors across all methods.
- **Logging**: Each step logs success or failure status to facilitate debugging.

4.4 Data_preprocessing Module:

4.4.1. Class: Preprocessor

The Preprocessor class is designed to handle multiple preprocessing steps in a structured way. It uses custom transformers and standard preprocessing techniques to clean and transform the dataset. It leverages PreprocessorConfig for configuration, ensuring that the system is parameterized.

Attributes:

- **config**: Holds configuration settings via PreprocessorConfig.
- **input_file**: A Pandas DataFrame, the input data to be preprocessed.
- **preprocessing_results**: Stores intermediate metadata (e.g., dropped columns, duplicates, outliers).

Method: drop_unwanted_columns

This method drops unnecessary columns.

- **Input:** DataFrame (df)
- **Action:**
  - Retrieves the column list from config.
  - Drops columns directly using Pandas drop().
  - Logs success/failure.
- **Output:** Cleaned DataFrame without unwanted columns.

Method: drop_duplicate_rows

Removes duplicate rows from the DataFrame.

- **Input:** DataFrame (df)
- **Action:**
  - Drops duplicates using drop_duplicates().
  - Logs the number of dropped rows.
  - Saves the cleaned data to a CSV.
- **Output:** Cleaned DataFrame without duplicates.

4.4.1.1 Class: HandleZeroStdColumns

A **custom transformer** to handle columns with zero standard deviation (no variance). Inherits from BaseEstimator and TransformerMixin.

- **Attributes:**
  - zero_std_columns: Tracks columns with zero standard deviation.
- **Key Operations:**
  - **fit()**: Identifies zero-std columns.
  - **transform()**: Drops those columns.
  - **fit_transform()**: Combines both fit and transform steps.

4.4.1.2 Class: HandleHighSkewColumns

This **custom transformer** deals with highly skewed columns using a PowerTransformer.

- *Attributes:*
  - skewed_columns: List of highly skewed columns.
  - power_transformation: A fitted PowerTransformer instance.
- *Key Operations:*
  - **fit()**: Identifies columns with skew values less than -1 or greater than 1.
  - **transform()**: Applies power transformation to reduce skewness.
  - **fit_transform()**: Combines fit and transform.

4.4.1.3 Class: HandleNaNValues

A **custom transformer** that uses a **KNNImputer** to fill missing values.

- *Attributes:*
  - columns_to_drop: Tracks columns with more than 50% NaN values.
  - knn_imputer: Instance of KNNImputer used for imputation.
- *Key Operations:*
  - **fit()**: Identifies columns to drop and prepares imputation.
  - **transform()**: Drops columns and applies KNN imputation.
  - **fit_transform()**: Executes fit and transform in a single step.

4.4.1.4 Class: OutlierHandler

This **custom transformer** handles outliers by clipping values based on the **IQR** method.

- *Attributes:*
  - lower_limits: Lower bounds for each feature.
  - upper_limits: Upper bounds for each feature.
- *Key Operations:*
  - **fit()**: Calculates IQR-based limits for each feature.
  - **transform()**: Clips values that lie outside these bounds.
  - **fit_transform()**: Combines both steps.

4.4.1.5 Method: initialize_preprocessing()

This method initializes and executes the full preprocessing pipeline.

- **Actions:**
  - Uses **FunctionTransformer** to wrap each individual preprocessing function.
  - Constructs a **Pipeline** with all preprocessing steps in order:
    1. Drop unwanted columns.
    2. Drop duplicate rows.
    3. Apply custom transformers (e.g., outlier handling, skew handling).
  - Logs each step's success or failure.

4.4.1.6 Error Handling and Logging:

Each function and transformer includes:

- **try-except blocks** to catch exceptions.
- **SensorFaultException** to handle custom error scenarios.
- **Logs** are recorded for success/failure at each step.

4.4.1.7 Flow of the Preprocessing Pipeline:

1. **Initialize**: The Preprocessor class is instantiated with a configuration and dataset.
2. **Execute Pipeline**:
    - Drop unwanted columns.
    - Drop duplicate rows.
    - Handle NaN values using KNNImputer.
    - Apply power transformation to skewed columns.
    - Remove zero-std columns.
    - Handle outliers using IQR-based clipping.
3. **Store Results**: Store metadata (e.g., total columns, outlier counts) in preprocessing_results.
4. **Logging**: Track each step to ensure traceability.

4.4.1.8 Summary of Key Components:

- **Preprocessor Class**: Manages the pipeline and preprocessing steps.
- **Custom Transformers**:
    - HandleZeroStdColumns: Removes zero-variance columns.
    - HandleHighSkewColumns: Reduces skew using power transformation.
    - HandleNaNValues: Imputes missing values using KNN.
    - OutlierHandler: Clips outliers based on IQR.
- **Error Handling**: Uses SensorFaultException for custom exception management.
- **Metadata Storage**: Saves intermediate results in preprocessing_results for transparency.

4.5 Data_clustering Module:

4.5.1 Class: Clusters

This class encapsulates all the logic related to clustering, including determining the optimal number of clusters, assigning cluster labels, and initializing the entire clustering pipeline. The class also handles exceptions using a custom SensorFaultException to ensure smooth error tracking.

Attributes:

- **config**: Holds configuration settings from the ClusterConfig object (e.g., file paths, cluster column name).
- **input_file**: A DataFrame containing the data to be clustered.
- **target_feature_name**: The name of the target column, which will be excluded from the clustering process.

Method: *find_optimal_clusters*
This method finds the optimal number of clusters using the **K-means algorithm** and the **elbow method**.

Input:

- **X**: DataFrame without the target variable.

Process:

1. **Calculate Inertia**:
   - Runs KMeans clustering for **1 to 10 clusters** and stores the **inertia** (within-cluster sum of squares).
2. **Use Elbow Method**:
   - Uses the **KneeLocator** to identify the "elbow point" (where inertia starts decreasing slowly).
   - This point determines the **optimal number of clusters**.
3. **Logging**:
   - Logs the success message with the optimal cluster number.

Output:

- Returns the **optimal number of clusters**.

Exception Handling:

- If an error occurs, raises a **SensorFaultException** and logs the error message.

Method: *find_cluster_labels*
This method assigns cluster labels to the data using **K-means** clustering with the optimal number of clusters.

Input:

- **X**: DataFrame without the target variable.
- **optimal_clusters**: The number of clusters determined by the elbow method.

Process:

1. **Fit K-means Model**:
   - Uses KMeans to fit the data and **predict cluster labels**.
2. **Create Folder for Storing Model**:
   - Checks if the folder path (cluster_object_path) exists and creates it if necessary.
3. **Store K-means Model**:
   - Saves the trained KMeans model as a serialized object using the save_obj() function.

Output:

- Returns a **tuple** containing:
  1. **Cluster labels**: 2D NumPy array.
  2. **Cluster object path**: Path where the KMeans model is stored.

Exception Handling:

- Logs any errors and raises a **SensorFaultException** if the process fails.

Method: *initialize_clusters*

This method initializes the entire clustering process, combining the results from **find_optimal_clusters** and **find_cluster_labels** into a single flow.

Process:

1. **Prepare Data**:
   - Creates a copy of the input DataFrame and **removes the target column**.
2. **Find Optimal Clusters**:
   - Calls the find_optimal_clusters() method to determine the optimal number of clusters.
3. **Assign Cluster Labels**:
   - Calls the find_cluster_labels() method to generate cluster labels for the data.
4. **Update DataFrame**:
   - Adds a new column for **cluster labels** using the name provided in the config.
5. **Calculate Silhouette Score**:
   - Computes the **silhouette score** to evaluate the clustering quality.
6. **Store Results**:
   - Creates a ClusterArtifacts object to store the final DataFrame, cluster object path, and silhouette score.

Output:

- Returns a **ClusterArtifacts** object containing:
  1. **final_file**: DataFrame with cluster labels.
  2. **cluster_object_path**: Path to the stored K-means model.
  3. **silhouette_score_**: Silhouette score for clustering.

Exception Handling:

- If any step fails, raises a **SensorFaultException** with detailed error logging.

4.5.2 Key Components:

1. **ClusterConfig**:
   - Stores configurations such as cluster column name and paths for saving cluster models.
2. **KMeans Algorithm**:
   - Used to group similar data points into clusters.
3. **KneeLocator (Elbow Method)**:
   - Identifies the optimal number of clusters by finding the elbow point in the inertia curve.
4. **ClusterArtifacts**:
   - A data class to hold clustering results, including the final DataFrame, cluster object path, and silhouette score.
5. **Silhouette Score**:
   - Evaluates how well the clusters are separated from each other.
6. **create_folder_using_file_path()**:
   - Utility function to ensure the folder path exists before saving the cluster model.
7. **save_obj()**:
   - Serializes and saves the KMeans model object for future use.

4.5.3 Error Handling and Logging:

- **SensorFaultException**:
  - A custom exception class to track specific errors during clustering operations.
- **Logging**:
  - Logs success and failure messages at each stage, ensuring traceability and debugging support.

4.5.4 Flow of the Clustering Pipeline:

1. **Initialize**: The Clusters class is instantiated with the input DataFrame, target feature name, and configurations.
2. **Execute initialize_clusters()**:
   - Find the optimal number of clusters.
   - Assign cluster labels to data.
   - Calculate the silhouette score to validate clustering.
   - Save the K-means model for future use.
3. **Return Results**:
   - The final DataFrame with cluster labels, along with the saved model path and silhouette score, is returned.

4.6 Model_tuner Module:

4.6.1 Class: **ModelTuner**

The ModelTuner class is designed to automate the process of hyperparameter tuning and model selection across multiple machine learning models. This class handles cross-validation, parameter search, model building, evaluation, and identifying the best-performing model based on specific metrics.

Attributes:

- **config:** **ModelTunerConfig**
  Holds the parameter grids required for tuning various models (like SVC, GaussianNB, RandomForest, etc.).

Method: __init__(self, config: ModelTunerConfig)

- **Purpose:** Initializes the ModelTuner with configuration data.
- **Input:** config object containing hyperparameter grids for each model.
- **Output:** None.

Method: stratifiedKfold_validation(self) -> StratifiedKFold

- **Purpose:** Creates a StratifiedKFold object for maintaining label distributions across train-test splits during cross-validation.
- **Output:** Returns a StratifiedKFold instance with n_splits=5.

Method: model_build(self, model: BaseEstimator, params: dict) -> RandomizedSearchCV

- **Purpose:** Builds a model using RandomizedSearchCV for hyperparameter tuning.
- **Input:**
  - model: The machine learning model to be tuned.
  - params: Dictionary containing the parameter grid.
- **Output:** Returns a RandomizedSearchCV object.
- **Error Handling:** Logs and raises custom SensorFaultException for errors.

Method: get_best_svc_parameters(self, X: pd.DataFrame, y: pd.Series) -> tuple[dict, BaseEstimator, dict]

- **Purpose:** Tunes and retrieves the best parameters for the SVC model.
- **Steps:**
  1. Splits X and y into train and test sets (80/20 split).
  2. Calls model_build with SVC and parameter grid.
  3. Fits the model and logs the process.
- **Output:** Returns a tuple containing:
  1. MLflow data, model object, and evaluation metrics.

Method: get_best_gaussiannb_parameters(self, X: pd.DataFrame, y: pd.Series) -> tuple[dict, BaseEstimator, dict]

- **Purpose:** Tunes the GaussianNB model and provides the best parameters.
- **Similar Logic:** Same as the get_best_svc_parameters method but for GaussianNB.

Method: get_best_randomforest_parameters(self, X: pd.DataFrame, y: pd.Series) -> tuple[dict, BaseEstimator, dict]

- **Purpose:** Tunes the RandomForestClassifier model and returns its best parameters.
- **Similar Logic:** Same as the SVC method but for RandomForestClassifier.

Method: get_best_xgbclassifier_parameters(self, X: pd.DataFrame, y: pd.Series) -> tuple[dict, BaseEstimator, dict]

- **Purpose:** Tunes XGBClassifier and retrieves the best parameters.
- **Similar Logic:** Same as above methods but applied to XGBClassifier.

Method: find_best_model(self, models_object_data: dict, models_results_data: dict) -> tuple[BaseEstimator, dict]

- **Purpose:** Identifies the best-performing model based on the highest AUC score.
- **Steps:**
  1. Extracts AUC scores from models_results_data.
  2. Compares the scores to find the best model.
  3. Logs and returns the best model and its results.
- **Error Handling:** Logs and raises SensorFaultException on failure.

Method: initialize_model_tuner_process(self, X: pd.DataFrame, y: pd.Series) -> ModelTunerArtifacts

- **Purpose:** Initiates the model tuning process by training multiple models.
- **Steps:**
  1. Calls individual methods to tune multiple models.
  2. Selects the best model using find_best_model.
  3. Logs the entire process.
- **Output:** Returns ModelTunerArtifacts containing all models' data and the best model's data.

Error Handling:

- Uses custom SensorFaultException to handle exceptions across methods.
- Logs both successes and failures for better debugging and tracking.

4.6.2 Design Decisions:

1. **RandomizedSearchCV:** Chosen for quicker hyperparameter tuning compared to GridSearchCV, especially when the parameter space is large.
2. **StratifiedKFold:** Ensures that train-test splits maintain the same class distribution, important for imbalanced datasets.
3. **Separate Methods per Model:** Makes the code modular and easier to maintain or extend (e.g., adding new models).
4. **Logging:** Provides comprehensive logs for monitoring the tuning process.

4.6.3 Sample Usage Workflow:

1. Instantiate the ModelTuner class with the necessary configuration.
2. Call initialize_model_tuner_process to start model training and tuning.
3. Retrieve the best model based on the AUC score using find_best_model.

This design ensures modularity, error handling, and scalability for various machine learning models.

4.7 Configuration package:

4.7.1 aws connection module:

Class: S3Client

Purpose:
This class provides a singleton-like design for establishing a connection with AWS S3 using both boto3.client and boto3.resource. It ensures only one instance of the S3 client and resource is created and reused throughout the application.

Attributes:

1. **s3_client (Class Attribute):**

- Stores the **boto3 client** instance, which provides low-level access to S3 operations (e.g., listing objects, uploading files).
- It is initialized as None to allow lazy initialization only when the class is instantiated for the first time.

2. **s3_resource (Class Attribute):**
   - Stores the **boto3 resource** instance, which offers higher-level object-oriented access to S3 (e.g., interacting with buckets as objects).
   - Defined with # type: ignore to suppress type warnings if using type checking tools (like mypy).

Constructor: __init__()

Logic:

- **Checks if s3_client or s3_resource is None:** Ensures that the S3 connection is only established once. If both are already set, the class uses the existing connection instead of creating new ones.
- **Reads AWS Credentials from Environment Variables:**
  - AWS_ACCESS_KEY_ID: Access key ID for AWS authentication.
  - AWS_SECRET_ACCESS_KEY: Secret access key for authentication.
  - AWS_REGION_NAME: The AWS region to connect with (optional, depends on your AWS setup).

  Error Handling:

  - If the required environment variables (AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY) are not set, it raises an exception with an appropriate error message.
- **Creates S3 Client and Resource Instances:**
  - If the credentials are valid, it initializes:
    - **s3_client:** Using boto3.client() with the access key, secret key, and region.
    - **s3_resource:** Using boto3.resource() with the same credentials.
- **Stores Instances in Class and Instance Attributes:**
  - Sets S3Client.s3_client and S3Client.s3_resource (class-level attributes).
  - Also assigns them to the instance-level attributes (self.s3_client and self.s3_resource) for ease of use within the class.

4.8 Db_management package

4.8.1 aws_storage module:

The SimpleStorageService class provides a structured interface to manage interactions with Amazon S3. It encapsulates functionality to perform bucket operations (like checking for folder existence), uploading/downloading files, and managing prediction models. The goal is to abstract common S3 operations and handle errors gracefully through custom exceptions (SensorFaultException).

Class Structure and Responsibilities

4.8.1.1 Constructor: . __init__

- **Purpose:** Initialize the service with configurations and create S3 clients.
- **Arguments:**
  - config: S3Config: Holds bucket names, folders, and paths for prediction models.
- **Components:**
  - Initializes s3_resource and s3_client from the S3Client wrapper class for resource management.

4.8.1.2 Method: get_bucket

- **Purpose:** Retrieve an S3 bucket object by name.
- **Returns:**
  - Bucket: If the bucket exists.
- **Error Handling:** Raises SensorFaultException for any failure.

4.8.1.3 Method: check_s3_subfolder_exists

- **Purpose:** Check if a specific folder exists within an S3 bucket.
- **Returns:**
  - True: If the folder exists.
  - False: If it does not.
- **Implementation:**
  - Uses filter(Prefix=...) to locate subfolders.

4.8.1.4 Method: create_s3_subfolder

- **Purpose:** Create a subfolder inside a specified S3 bucket.
- **Logic:**
  - First checks if the folder exists.
  - If not, creates the folder by uploading an empty object with the folder path as the key.

4.8.1.5 Method: upload_folder_to_s3

- **Purpose:** Upload the contents of a local folder to an S3 folder, preserving the structure.
- **Implementation:**
  - Uses os.walk to recursively traverse the local folder.
  - Calls upload_file_to_s3 to upload each file.

4.8.1.6 Method: upload_files_to_s3

- **Purpose:** Upload all files from a given local path into a specified S3 subfolder.
- **Logic:**
  - Supports both single files and folders.
  - Converts Windows-style backslashes to Unix-style for S3 paths.

4.8.1.7 Method: upload_file_to_s3

- **Purpose:** Upload a single file to S3.
- **Implementation:** Uses bucket_obj.upload_file.

4.8.1.8 Method: list_files_in_s3_folder Method

- **Purpose:** List all files in a specific S3 folder.
- **Returns:**
  - A list of file paths present in the folder.

4.8.1.9 Method: download_files_from_s3 Method

- **Purpose:** Download all files from an S3 folder to a local directory.
- **Implementation:**
  - Uses list_files_in_s3_folder to get the list of files.
  - Downloads each file using download_file_from_s3.

4.8.1.10 Method: download_file_from_s3 Method

- **Purpose:** Download a single file from S3 to a local path.
- **Implementation:** Uses bucket_obj.download_file.

4.8.1.11 Method: check_s3_folder_empty Method

- **Purpose:** Check if a folder in S3 is empty.
- **Returns:**
  - True: If the folder is empty.
  - False: Otherwise.

4.8.1.12 Method: store_prediction_models Method

- **Purpose:** Store models either in the champion or challenger folder based on the AUC score threshold.
- **Logic:**
  - If the champion folder is empty or the AUC score threshold is met, store in the champion folder. Otherwise, use the challenger folder.
  - Copies objects from the local folder to the appropriate S3 target folder.

4.8.2 Key Error Handling Pattern

- All methods employ try-except blocks and log exceptions using SensorFaultException.
- Detailed logging ensures easy debugging and monitoring of success/failure states.

4.8.3 Design Choices

1. **Custom Exception Handling:**
   - The class raises a SensorFaultException to centralize error management.
2. **Modular Methods:**
   - Each S3-related task is encapsulated in its own method for clarity and reusability.
3. **Path Normalization:**
   - The code ensures that paths are converted to S3-compliant formats (/ separator) to avoid platform-specific issues.

4.8.4 Sequence Flow

1. **Initialization:**
   - The class initializes with configurations and S3 clients.
2. **Bucket Operations:**
   - The bucket is retrieved using get_bucket.
3. **Folder Operations:**
   - The existence of a folder is verified, and it is created if absent.
4. **Upload/Download Operations:**
   - Files/folders are uploaded or downloaded between local paths and S3.
5. **Model Management:**
   - Models are stored based on AUC thresholds.

4.9 Pipeline package :

4.9.1. training_pipeline module:

4.9.1.1 Class: TrainingPipeline

This class is responsible for orchestrating the **end-to-end training process** for the sensor fault detection system. It initializes necessary configurations, manages various steps of the training pipeline, and handles any exceptions during the process.

Constructor: __init__ **Method**

This method initializes all the required configurations and dependencies.

Attributes:

1. **self.s3_config**:
   - Configuration settings for the S3 service (e.g., access keys, bucket names).
2. **self.s3**:
   - Creates an instance of SimpleStorageService using the S3 configuration to interact with S3.
3. **self.s3_bucket_obj**:
   - Retrieves a reference to the S3 bucket for further operations (like reading/writing files).
4. **Configuration Objects:**
   - Initializes various configurations required for:
     - **Data ingestion** (self.data_ingestion_config)
     - **Raw data validation** (self.rawdata_validation_config)
     - **Data transformation** (self.rawdata_transformation_config)
     - **Preprocessing** (self.preprocessor_config)
     - **Clustering** (self.cluster_config)

- **Model tuning** (self.model_tuner_config)
- **Model training** (self.model_trainer_config)

4.9.2. initialize_pipeline Method

This method handles the complete sequence of steps required to train the model.

4.9.1.2.1. Steps:

1. **Generate Timestamp:**

   ```
   self.timestamp = datetime.now().strftime("%d_%m_%Y_%H_%M_%S")
   ```

   - Generates a timestamp to uniquely identify the training process.
2. **Initialize Model Evaluation Config:**

   ```
   self.model_evolution_config = ModelEvaluationConfig(self.timestamp)
   ```

   - Prepares configuration for model evaluation, tied to the generated timestamp.
3. **Log Start of Training:**
   - Logs the start of the training pipeline.

Data Ingestion:

1. **Initialize GetTrainingData:**
   - Fetches training data from S3 using the configurations:

     ```
     get_training_data = GetTrainingData(
         data_ingestion_config=self.data_ingestion_config,
         s3_obj=self.s3,
         s3_config=self.s3_config,
         s3_bucket_obj=self.s3_bucket_obj
     )
     ```
2. **Start Data Retrieval Process:**

   ```
   status, files_path = get_training_data.initialize_getting_training_data_process()
   logger.info(msg=f"Folder Path:{files_path}")
   ```

   - Retrieves the training files from the S3 bucket and logs the folder path.
3. **Remove Dashboard Validation File (if applicable):**

   ```
   remove_file(self.rawdata_validation_config.dashboard_validation_show)
   ```

   - Removes unnecessary validation files if the pipeline is in training mode.

Raw Data Validation:

- **If status is "default_training":**
  - Initializes **raw data validation** and performs the process:

    ```
    raw_data_validation = RawDataValidation(
        self.rawdata_validation_config, folder_path=files_path
    )
    ```

```
raw_data_validation_artifacts =
raw_data_validation.initialize_rawdata_validation_process()
```

- **Otherwise:**
    - Skips validation and directly creates a RawDataValidationArtifacts object:

```
raw_data_validation_artifacts = RawDataValidationArtifacts(
    good_raw_data_folder=files_path,
    bad_raw_data_folder=self.rawdata_validation_config.bad_raw_data_folder_path,
    validation_log_file_path=self.rawdata_validation_config.validation_report_file_path
)
```

Data Transformation:

- **Initialize and Perform Transformation:**

```
raw_data_transformation = RawDataTransformation(
    config=self.rawdata_transformation_config,
    rawdata_validation_artifacts=raw_data_validation_artifacts
)
self.raw_data_transformation_artifacts =
raw_data_transformation.initialize_data_transformation_process()
```

Data Ingestion Process:

- **Ingest Transformed Data:**

```
data_ingestion = DataIngestion(
    input_dataset_path=self.raw_data_transformation_artifacts.final_file_path
)
data_ingestion_artifacts = data_ingestion.initialize_data_ingestion_process()
```

Preprocessing:

- **Initialize Preprocessor:**

```
input_file = data_ingestion_artifacts.input_dataframe
data_preprocessing = Preprocessor(
    config=self.preprocessor_config, input_file=input_file
)
data_preprocessing_artifacts = data_preprocessing.initialize_preprocessing()
```

Clustering:

- **Initialize Clusters:**

```
input_file = data_preprocessing_artifacts.preprocessed_data
clusters = Clusters(
    config=self.cluster_config,
    input_file=input_file,
    target_feature_name=self.preprocessor_config.target_feature
)
cluster_artifacts = clusters.initialize_clusters()
```

Model Training:

- **Initialize and Train the Model:**

```
input_file = cluster_artifacts.final_file
model_trainer = ModelTrainer(
    config=self.model_trainer_config,
    input_file=input_file,
    modeltunerconfig=self.model_tuner_config,
    model_evolution_config=self.model_evolution_config
)
model_trainer.initialize_model_trainer()
```

4.9.1.2.2 Log Completion:

- Logs the successful completion of the training pipeline:

```
logger.info(msg="----------Completed Training Pipeline----------")
```

4.9.1.2.3 Exception Handling:

- **Catch and Log Errors:**

```
except Exception as e:
    logger.error(msg=SensorFaultException(error_message=e, error_detail=sys))
```

    - If any step fails, the exception is logged using a SensorFaultException.

4.9.3. Summary of Pipeline Flow:

1. **DataIngestion:**
   Fetch data from S3.
2. **DataValidation:**
   Validate raw data (if needed) and log artifacts.
3. **DataTransformation:**
   Transform data for ingestion.
4. **DataPreprocessing:**
   Apply feature transformations.
5. **Clustering:**
   Perform clustering to group data.
6. **ModelTraining:**
   Train the model using the transformed and clustered data.
7. **Logging:**
   Logs the start and end of the training process and handles exceptions if they occur.

5.0. prediction_pipeline module:

5.0.1 Module: PredictionPipelineConfig

- **Purpose**: Holds the configuration parameters for the prediction pipeline, such as paths for input files, output files, and models.

Code Demo:

```
class PredictionPipelineConfig:
    def __init__(self):
        self.predictions_data_path = "path/to/predictions_data.xlsx"
        self.preprocessor_stage_one_obj_path = "path/to/preprocessor_stage_one.pkl"
        self.cluster_obj_path = "path/to/cluster_model.pkl"
        self.best_models_path = "path/to/models"
        # Additional config parameters here

# Usage
config = PredictionPipelineConfig()
print(config.predictions_data_path)  # Output: path/to/predictions_data.xlsx
```

5.0.2 Module: RawDataValidation

- **Purpose**: Validates the raw data before processing, ensuring it meets the necessary requirements for the pipeline.

Code Demo:

```
class RawDataValidation:
    def __init__(self, config, folder_path):
        self.config = config
        self.folder_path = folder_path

    def initialize_rawdata_validation_process(self):
        # Perform validation checks on raw data
        print("Validating raw data...")
        # Return validation artifacts, e.g., file paths or validation status

# Usage
validation_config = PredictionRawDataValidationConfig()
raw_data_validation = RawDataValidation(config=validation_config, folder_path="path/to/raw_data")
validation_artifacts = raw_data_validation.initialize_rawdata_validation_process()
```

5.0.3 Module: RawDataTransformation

- **Purpose**: Transforms validated raw data into a format suitable for model ingestion.

Code Demo:

```
class RawDataTransformation:
    def __init__(self, config, rawdata_validation_artifacts):
        self.config = config
        self.validation_artifacts = rawdata_validation_artifacts

    def initialize_data_transformation_process(self):
        # Transform data based on validation artifacts
        print("Transforming raw data...")
        # Return transformation artifacts, e.g., file paths

# Usage
transformation_config = PredictionRawDataTransformationConfig()
raw_data_transformation = RawDataTransformation(config=transformation_config,
rawdata_validation_artifacts=validation_artifacts)
```

```
transformation_artifacts = raw_data_transformation.initialize_data_transformation_process()
```

5.0.4 Module: DataIngestion

- **Purpose**: Ingests the transformed data for further processing in the pipeline.

Code Demo:

```
class DataIngestion:
    def __init__(self, input_file):
        self.input_file = input_file

    def get_data(self):
        # Read the transformed data file
        print("Ingesting data from:", self.input_file)
        # Return ingested data for preprocessing

# Usage
data_ingestion = DataIngestion(input_file="path/to/transformed_data.csv")
ingested_data = data_ingestion.get_data()
```

5.0.5 Module: DataPreprocessing

- **Purpose**: Preprocesses the ingested data, handling scaling and any necessary transformations.

Code Demo:

```
def load_obj(filepath):
    # Utility function to load pickled objects
    with open(filepath, 'rb') as file:
        return pickle.load(file)

# Load and apply preprocessor object
preprocessor_path = "path/to/preprocessor_stage_one.pkl"
preprocessor = load_obj(preprocessor_path)
processed_data = preprocessor.transform(ingested_data)  # Assuming `ingested_data` is a
DataFrame
```

5.0.6 Module: DataClustering

- **Purpose**: Assigns clustering labels to preprocessed data to facilitate model-specific predictions.

Code Demo:

```
class DataClustering:
    def __init__(self, cluster_model_path):
        self.cluster_model = load_obj(cluster_model_path)

    def assign_clusters(self, data):
        print("Assigning clusters...")
        return self.cluster_model.predict(data)

# Usage
```

```
data_clustering = DataClustering(cluster_model_path="path/to/cluster_model.pkl")
cluster_labels = data_clustering.assign_clusters(processed_data)
```

5.0.7 Module: PredictionModel

- **Purpose**: Loads the appropriate model based on the cluster and makes predictions on the data.

Code Demo:

```
class PredictionModel:
    def __init__(self, model_path):
        self.model = load_obj(model_path)

    def predict(self, data):
        print("Making predictions...")
        return self.model.predict(data), self.model.predict_proba(data)

# Usage
model_path = "path/to/cluster_model.pkl"
prediction_model = PredictionModel(model_path)
predictions, probabilities = prediction_model.predict(processed_data)
```

Each module has been designed to work as a component of the main PredictionPipeline class, with configurations and artifacts passed across modules to ensure data consistency and smooth workflow execution.

5 .1 App Module:

5.1.1 App Initialization & Middleware

- app = FastAPI(): Initializes the FastAPI app instance, which will handle all routes and responses.
- **CORS Middleware**:
  - CORSMiddleware is added to the app to manage Cross-Origin Resource Sharing (CORS), allowing the frontend (which may run on a different domain or port) to interact with the backend.
  - allow_origins=["*"]: Allows all domains to access the app's endpoints. This can be restricted as per security requirements.
- **Static File Mounting**:
  - The line app.mount("/static", StaticFiles(directory="static"), name="static") serves static files (like CSS or JavaScript) from the static directory under the /static URL path.

5.1.2. Template Setup

- templates = Jinja2Templates(directory="templates") initializes the Jinja2 templates directory for rendering HTML templates in responses.

5.1.3. Root Route (Homepage)

- @app.get("/"): A GET route for the root URL.
- c_time = datetime.now().strftime("%d_%m_%Y_%H_%M_%S"): Updates the timestamp each time the root route is accessed. This is printed as correct time in the logs and serves as a session-based unique identifier.
- Returns the index.html template, displaying the main interface of the app.

5.1.4. Training Process

- **Async Function to Train Model**:
  - train_model(): An async function that initializes the training pipeline and marks the training status as complete. It uses asyncio.sleep(0) to yield control, letting other tasks run in parallel while training is initialized.
- **Syncing S3 Training Data**:
  - sync_s3_training_data(): Uploads the local training artifacts to S3.
  - training_pipeline.s3.upload_folder_to_s3(...): Uploads the contents of the local artifact folder to an S3 bucket, maintaining a unique folder structure based on c_time.
  - If successful, upload_status["completed"] = True is set to prevent redundant uploads.
- **Training Route**:
  - @app.get("/train"): Triggers model training by adding train_model to background tasks.
  - Responds with training.html, a template that can display the training status on the frontend.

5.1.5. Training Results Display

- **Route for Results**:
  - @app.get("/training_results"): Provides detailed information about the training results, including data validation, preprocessing, and model performance metrics.
- **Data Loading**:
  - Loads Excel and JSON data files for validation, preprocessing, and model evaluation summaries.
- **Response Template**:
  - training_report.html displays the training results on the frontend. This report includes:
    - Validation summary (e.g., file counts and reasons for validation failures)
    - Preprocessing summary (e.g., statistics on data imputation, outliers, and skew)
    - Model performance across clusters (scores and metrics for each model, without best_param).

5.1.6. Prediction Process

- **Dashboard Route**:
  - @app.get("/dashboard"): Provides an endpoint to render dashboard.html, which might be a frontend interface for managing or displaying prediction-related tasks.
- **File Upload Route**:
  - @app.post("/upload"): Allows users to upload prediction files.
  - **File Processing**:
    - Files are saved to a directory specified by PredictionRawDataValidationConfig.prediction_raw_data_folder_path.
    - Each file is saved after verifying its content and structure, ensuring no empty files are processed.
  - Returns a JSON message to confirm successful file uploads.
- **Prediction Route**:
  - @app.post("/data_prediction"): Initializes and runs the prediction pipeline.
  - If successful, it starts an async background task (sync_s3_prediction_data) to upload prediction results and data drift reports to S3.
  - Returns a JSON response to indicate prediction status.

5.1.7. Syncing S3 Prediction Data

- **Sync S3 Task**:
  - sync_s3_prediction_data(): Uploads local prediction artifacts to S3 and handles data drift reports.
  - **Drift Detection**:
    - Compares the original training data with new predictions to detect data drift.
    - If drift is detected, emails are sent to both team and client with relevant drift report URLs.
- **Helper Function for Sync**:
  - upload_s3_prediction_data(): Executes the actual upload logic within a thread using run_in_threadpool for asynchronous upload processing.
  - Ensures that prediction data and drift report files are stored properly on S3.

5.1.8. Validation Summary Endpoint

- @app.get("/validation_summary"): Provides a JSON summary of data validation results.
- Processes the status of validation results from an Excel file and returns a summary dictionary that includes both pass and fail counts for files.

5.1.9. Get Predictions Endpoint

- @app.get("/predictions"): Fetches predictions from an Excel file and returns it as a JSON response.
- This could be enhanced by formatting or filtering specific details in the predictions before returning.

5 Data Flow