APS SENSOR FAULT DETECTION PROJECT

1. Setup.py

↳ basic structure and metadata for the project Package to be used for distribution and installation facilities.

↳ setup func : → distribution metadata
→ package details
↳ dependencies
↳ versions
↳ author info etc.

2. Requirements

↳ requirements.txt
↳ lib == version
↳ -e. : find packages

——— So far, its basic package structure ———

3. Create source **folder** and turn it into a package

↳ Create <u>Sensor</u> package

↳ Create <u>pipeline package</u> in sensor folder

↳ high level, series of interconnected data processing and modeling steps designed to streamline the end-to-end ML work flow

$$\begin{tikzcd}
\boxed{\text{Data Ingestion}} \arrow[r] & \boxed{\text{Data Validation}} \arrow[r] & \boxed{\text{Data Transf}} \arrow[r] & \boxed{\text{Model Training}} \arrow[d] \\
& \boxed{\text{Cloud Syncronizer}} & \boxed{\text{Model Pusher}} \arrow[l] & \boxed{\text{Model Evaluation}} \arrow[l]
\end{tikzcd}$$

4. Create components package Under sensor folder:

- ↳ data_ingestion . py
- ↳ data_ validation . py
- ↳ data - transformation.py
- ↳ model_ evaluation . py
- ↳ model - pusher. py
- ↳ model - trainer. py

5. Create data_access package Under sensor folder:

- ↳ sensor - data. py
  - ↳ Sensor Data Class :
    - ↳ init : mongo - client DB connection
    - ↳ save - csv- file : read data from filepath $\xrightarrow{\text{then}}$ save it to a Database collection
    - ↳ export- collection_ as _ dataframe : read data from db Collec and return as df.

6. Create configuration package Under sensor folder

↳ configuration package is often used to organize and manage various configuration settings and parameters for projects. In this project we created this package only to handle MongodB connections centerally.

- ↳ mongo_db_ connection. py
  - ↳ class Mongo Client

7. Create constants package under sensor folder:

↳ centralize and organize constant values, config settings or other static data that are used accross different parts of our project.

↳ improves code readability, maintainability, and consistency by providing a single place to define and manage these values

↳ application.py : APP_HOST, APP_PORT

↳ database.py : DATABASE_NAME, COLLECTION_NAME

↳ env-variables.py (sensitive variables) : MONGODB-URL-KEY, AWS ID KEY & SECRET KEY

AWS-REGION

↳ S3-bucket.py : TRAINING-BUCKET-NAME
PREDICTION_BUCKET_NAME

8. Create exceptions.py file in sensor folder:

↳ def error_message_detail

↳ class SensorException (Exception)

↳ our customized exception class

↳ Structured code

↳ Consistency across the project

↳ Encapsulation

9. Create logger.py file in sensor folder

$$\begin{tikzcd}
\text{logs-path} \arrow[r] & \text{make dirs} \arrow[r, "\text{join}"] & \text{LOG\_FILE-PATH} \\
& & \text{LOG\_FILE (name w/ timestamp)} \arrow[u]
\end{tikzcd}$$

↳ logging.basicConfig (filename, format, level)

→ capture and store information to aid in monitoring, debugging, trouble shooting, and auditing applications.

*10. Create entity package under sensor package *

↳ artifact_entity.py

↳ dataclasses serving as clear and structured representations of the different artifacts (outputs) that our app deals with during various stages of the ML cycle / pipeline.

↳ By encapsulating the data and behaviour within these classes, we create more organized and understandable way to manage the complexity of our ML pipeline

↳ Each data class defines attributes that correspond to specific properties or metadata related to the artifacts, and the @dataclass decorator simplifies the creation of these classes by autlomatically generating common methods like __init__ and __repr__

↳ Promotes : Readability, Maintainability and seperation of concerns (Abstraction), (Encapsulation)

↳ Check the Artifact Diagram

~~#~~. ↳ config_entity.py

↳ define set of Configuration classes, each encapsulating different configuration settings related to various stages of our training pipeline. These classes help us to organize and manage the configuration settings required for different parts of our application

↳ Contains sets of directory paths, file paths, object paths and some constants (metrics, threshold values etc) derived from constants > training_pipeline

11. Create ml package under sensor folder

* ↳ Custom ml algoritms
* ↳ Custom loss functions
* ↳ Custom metric functions
* ↳ Model evaluation functions
* ↳ visualizations
* ↳ Documentation & Research: we can store here our research documents, experimental setups and findings.
* ↳ Experiment Tracking
* ↳ Model version control

We've used ml package for 2 specific reasons:
- classification Metrics
- ↳ model_estimat

↳ ml > metric > classification_metric.py

↳ return ClassificationMetricArtifact:

[
* f1_score
* precision_score
* recall_score
⋮
]

↳ ml > model > model_estimator.py

↳ Target Value Mapping: encoding / decoding the target feature for prediction & evaluation : OPTIONAL

↳ class SensorModel:

def init : processor and model

(preprocessing pipeline)

def predict : transform by preprocessor → predict and return Y_hat

↳ class Model Resolver:

↳ look into the saved models directory

↳ def is_model_exists → check if there is a saved model dir & a model in it then get_best_model_path

↳ def get_best_model_path
(logic : latest one)