Sensor Fault Detection - Project

- Aim :- If the failure is because of APS.
- It's a binary classification Problem.
- First we will setup our git.
- Connect MongoDB / SQL to fetch data.
- Create a Python file (data_dump.py) to dump data in MongoDB.
- Create requirements.txt file & keep writing all required packages.
- In data_dump file we run code which convert data into json format.
  - ↳ `json_record = list(json.loads(df.T.to_json()).values())`
- .env file is helpful in keeping credentials like keys, URI etc.
- .gitignore helps github in ignoring the files mentioned in .gitignore.

- Create setup.py file. It's required in every project. Used in converting our code in the library format so we can use our code as library anytime.

↳ from setuptools import find_packages, setup

find_packages will be creating folder containing .py files. So find_packages search for those folders & it will convert .py files into packages.

Q How it come to know that this folder will be used as package?

A __init__.py file within folder makes find_package consider folder as a package.

In this setup.py file we have get_requirements(), we use it to parse all the required packages. We get this list of packages by reading requirements.txt file.

↳ with open (requirements.txt) as requirement_file:

req_list = requirement_file.readlines()

req_list = [req_name.replace('\n', '') for req_name in req_list]

if '-e .' in req_list:

req_list.remove('-e .')

return req_list

* Imp

This -e . is in requirements.txt file. Required in using our code as library.

↳ sensor.egg-info folder will be created automatically containing all info of package & folder which will be used as package

setup (

name = "Project_name"

version = "0.0.1"

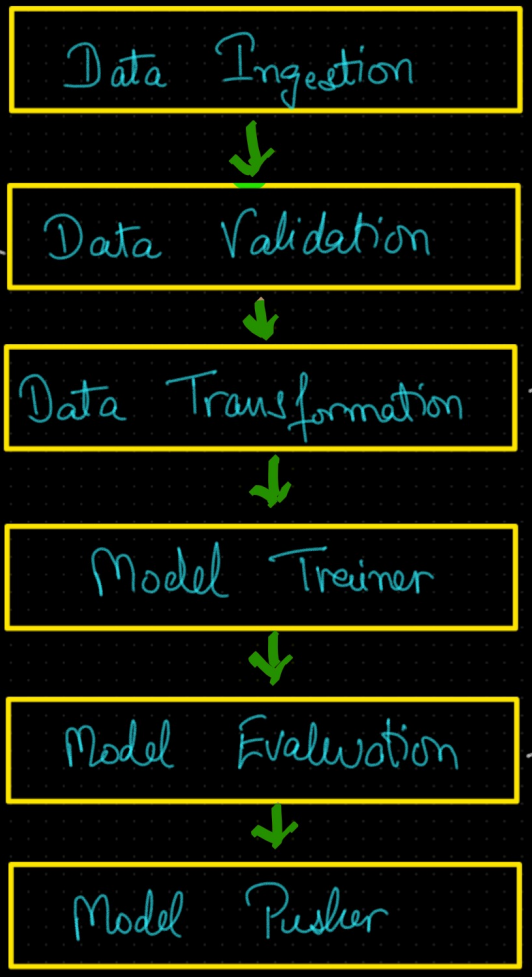
author = " "

author_email = " "

packages = find_packages()

install_requires = get_requirements())

ML Training Pipeline



Bringing data & splitting it in 3:- Train, Test & Validation.

Validate the data we have, eg, size, shape, datatypes etc.

Converting, Scaling, Preprocessing.

We pick a model to train. We get all work done in Jupyter.

Evaluating / Comparing best models. Then picking better one.

Deployment

We will create separate folder of all these stages & perform actions on the same.

- Each stage takes input & gives output.

CONFIGURATION

ARTIFACT

Any output generated by training pipeline.

- In our major folder, sensor folder in here, we will create more folders :-

  - Components → Here we write code for each component of Pipeline with a dedicated file of each step. __init__.py, data_ingestion.py, data_transformation.py, data_validation.py, model_trainer.py, model_evaluation.py, model_pusher.py.
  - Entity → Here we define input & output structure of Components. __init__.py, config_entity.py, artifact_entity.py
  - Pipeline → Here we setup our training pipeline. __init__.py, training_pipeline.py.

__init__.py will help us use these files as a library in other files.

- We create utils.py file in sensor folder because here we will keep all of our helper functions. like saving model, loading, pushing to S3 etc.

- Within config_entity.py & artifact_entity.py we create classes of all components of pipeline. A total of 6:-

  - Data Ingestion Config
  - Data Validation Config
  - Data Transformation Config
  - Model Evaluation Config
  - Model Trainer Config
  - Model Pusher Config

  - Data Ingestion Artifact
  - Data Validation Artifact
  - Data Transformation Artifact
  - Model Trainer Artifact
  - Model Evaluation Artifact
  - Model Pusher Artifact

- In components folder we will create 6 files & each file will be dedicated to each component of training pipeline.

- In sensor folder we create 2 more files to be used EVERYWHERE!

  - logger.py
  - Exception.py

```python
import logging
import os
from datetime import datetime
import os

#log file name
LOG_FILE_NAME = f"{datetime.now().strftime('%m%d%Y__%H%M%S')}.log"

#log directory
LOG_FILE_DIR = os.path.join(os.getcwd(),"logs")

#create folder if not available
os.makedirs(LOG_FILE_DIR,exist_ok=True)

#log file path

LOG_FILE_PATH = os.path.join(LOG_FILE_DIR,LOG_FILE_NAME)

logging.basicConfig(
    filename=LOG_FILE_PATH,
    format="[ %(asctime)s ] %(lineno)d %(name)s - %(levelname)s - %(message)s",
    level=logging.INFO,
)
```

Logging

```python
import sys,os

def error_message_detail(error, error_detail: sys):
    _, _, exc_tb = error_detail.exc_info()
    file_name = exc_tb.tb_frame.f_code.co_filename
    error_message = "Error occurred python script name [{0}] line number [{1}] error message [{2}]".format(
        file_name, exc_tb.tb_lineno, str(error)
    )
    return error_message



class SensorException(Exception):

    def __init__(self,error_message, error_detail:sys):
        self.error_message = error_message_detail(
            error_message, error_detail=error_detail)

    def __str__(self):
        return self.error_message
```

Exception

- We create a .env file in which we will keep credentials like mongodb url.
- We create a config.py file in sensor folder, this file will be having code of setting up connections.

```python
import pymongo
import pandas as pd
import json
from dataclasses import dataclass
# Provide the mongodb localhost url to connect python to mongodb.
import os

@dataclass
class EnvironmentVariable:
    mongo_db_url:str = os.getenv("MONGO_DB_URL")
    aws_access_key_id:str = os.getenv("AWS_ACCESS_KEY_ID")
    aws_access_secret_key:str = os.getenv("AWS_SECRET_ACCESS_KEY")


env_var = EnvironmentVariable()
mongo_client = pymongo.MongoClient(env_var.mongo_db_url)
TARGET_COLUMN = "class"
```

config.py

- In __init__.py file of sensor folder, we load a function named load_dotenv which makes sure environment variables gets called. Whenever we import sensor, automatically this reading from file will be done.

```python
from dotenv import load_dotenv
print(f"Loading environment variable from .env file")
load_dotenv()
```

__init__.py

- In utils.py file we are writing all of our helper functions. At current stage i.e. Data Ingestion, we want to import our data from MongoDB as a Dataframe. So we write a function in utils.py named get_collection_as_dataframe.

utils.py

```python
def get_collection_as_dataframe(database_name:str,collection_name:str)->pd.DataFrame:
    """
    Description: This function return collection as dataframe
    ========================================================
    Params:
    database_name: database name
    collection_name: collection name
    ========================================================
    return Pandas dataframe of a collection
    """
    try:
        logging.info(f"Reading data from database: {database_name} and collection: {collection_name}")
        df = pd.DataFrame(list(mongo_client[database_name][collection_name].find()))
        logging.info(f"Found columns: {df.columns}")
        if "_id" in df.columns:
            logging.info(f"Dropping column: _id ")
            df = df.drop("_id",axis=1)
        logging.info(f"Row and columns in df: {df.shape}")
        return df
    except Exception as e:
        raise SensorException(e, sys)
```

- In config_entity.py file, along with 6 steps component class, we will Create one more class Training Pipeline, its work is to store all the outputs in a single folder. We named folder as artifact.

```python
class TrainingPipelineConfig:

    def __init__(self):
        try:
            self.artifact_dir = os.path.join(os.getcwd(),"artifact",f"{datetime.now().strftime('%m%d%Y__%H%M%S')}")
        except Exception  as e:
            raise SensorException(e,sys)
```

Training Pipeline

- Now all of our classes of each component of pipeline will be of type Training Pipeline Config
- So as output ... Artifact folder will be created containing another folder named as (timestamp), we can see it as pipeline directory & then this folder will have folder of each component containg the files or outputs we desire.

eg.  ARTIFACT (created automatically using Training Pipeline)

 11022022_163512

 data_ingestion

Data Ingestion

→ Config_entity.py

```
class DataIngestionConfig:

    def __init__(self,training_pipeline_config:TrainingPipelineConfig):
        try:
            self.database_name="aps"
            self.collection_name="sensor"
            self.data_ingestion_dir = os.path.join(training_pipeline_config.artifact_dir , "data_ingestion")
            self.feature_store_file_path = os.path.join(self.data_ingestion_dir,"feature_store",FILE_NAME)
            self.train_file_path = os.path.join(self.data_ingestion_dir,"dataset",TRAIN_FILE_NAME)
            self.test_file_path = os.path.join(self.data_ingestion_dir,"dataset",TEST_FILE_NAME)
            self.test_size = 0.2
        except Exception  as e:
            raise SensorException(e,sys)

    def to_dict(self,)->dict:
        try:
            return self.__dict__
        except Exception  as e:
            raise SensorException(e,sys)
```

- We setup DB & collection names.
- Create a directory within artifact folder, named as "data_ingestion".
- feature_store

- We are keeping train & test files in dataset folder.

- to_dict function helped in returning data as a dictionary.
- This whole class we can pass details to MongoDB, then we store the data in the directory we have specified after splitting into train/test, doing this our ingestion will be done.

→ data_ingestion.py

```python
class DataIngestion:

    def __init__(self,data_ingestion_config:config_entity.DataIngestionConfig ):
        try:
            logging.info(f"{'>>'*20} Data Ingestion {'<<'*20}")
            self.data_ingestion_config = data_ingestion_config
        except Exception as e:
            raise SensorException(e, sys)

    def initiate_data_ingestion(self)->artifact_entity.DataIngestionArtifact:
        try:
            logging.info(f"Exporting collection data as pandas dataframe")
            #Exporting collection data as pandas dataframe
            df:pd.DataFrame  = utils.get_collection_as_dataframe(
                database_name=self.data_ingestion_config.database_name,
                collection_name=self.data_ingestion_config.collection_name)

            logging.info("Save data in feature store")

            #replace na with Nan
            df.replace(to_replace="na",value=np.NAN,inplace=True)

            #Save data in feature store
            logging.info("Create feature store folder if not available")
            #Create feature store folder if not available
            feature_store_dir = os.path.dirname(self.data_ingestion_config.feature_store_file_path)
            os.makedirs(feature_store_dir,exist_ok=True)
            logging.info("Save df to feature store folder")
            #Save df to feature store folder
            df.to_csv(path_or_buf=self.data_ingestion_config.feature_store_file_path,index=False,header=True)


            logging.info("split dataset into train and test set")
            #split dataset into train and test set
            train_df,test_df = train_test_split(df,test_size=self.data_ingestion_config.test_size)

            logging.info("create dataset directory folder if not available")
            #create dataset directory folder if not available
            dataset_dir = os.path.dirname(self.data_ingestion_config.train_file_path)
            os.makedirs(dataset_dir,exist_ok=True)

            logging.info("Save df to feature store folder")
            #Save df to feature store folder
            train_df.to_csv(path_or_buf=self.data_ingestion_config.train_file_path,index=False,header=True)
            test_df.to_csv(path_or_buf=self.data_ingestion_config.test_file_path,index=False,header=True)

            #Prepare artifact

            data_ingestion_artifact = artifact_entity.DataIngestionArtifact(
                feature_store_file_path=self.data_ingestion_config.feature_store_file_path,
                train_file_path=self.data_ingestion_config.train_file_path,
                test_file_path=self.data_ingestion_config.test_file_path)

            logging.info(f"Data ingestion artifact: {data_ingestion_artifact}")
            return data_ingestion_artifact

        except Exception as e:
            raise SensorException(error_message=e, error_detail=sys)
```

- Type of this class is DataIngestionConfig.
- Output will be of type DataIngestionArtifact.
- After defining Input & output, we initiate our process of importing, cleaning, splitting.
- We get data using function from utils.py
- Then we replaced all Nan values.
- We saved data in feature_store folder.
- We use dirname() we get directory of any file location.
- Then we dump data to this feature folder.
- We split the datasets & saved in new folder.
- Lastly we prepare output/artifact. We use DataIngestionArtifact

→ artifact_entity.py

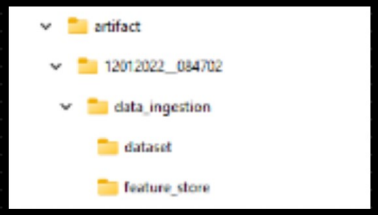
```python
@dataclass
class DataIngestionArtifact:
    feature_store_file_path:str
    train_file_path:str
    test_file_path:str
```

- We specify what outputs will be generated.
- DataIngestion component will have these 3 outputs.

→ Main.py

```python
if __name__=="__main__":
    try:
        training_pipeline_config = config_entity.TrainingPipelineConfig()
        data_ingestion_config  = config_entity.DataIngestionConfig(training_pipeline_config=training_pipeline_config)
        print(data_ingestion_config.to_dict())
        data_ingestion = DataIngestion(data_ingestion_config=data_ingestion_config)
        data_ingestion_artifact = data_ingestion.initiate_data_ingestion()
```

RESULT :-



Artifact > Timestamp folder by Training Pipeline >
Data Ingestion folder by Config_entity >
Dataset & feature_store folders with files.

DATA VALIDATION

- Here we have one base dataset on top of which we do EDA & all other steps in Jupyter.
- Now whenever new data will come in pipeline, we will compare it with base dataset & proceed further pipeline only if data is valid else exit.
- feature_store is a folder where we get our data to start ML pipeline.
- @dataclass is a decorator which allows us to skip writing __init__ within a class to take variables.

eg:

```
@dataclass
class Mercedes:
    engine : str
    price : float
```

- For validation, we need to decide parameters on which we will validate upcoming dataset.
  - ↳ no. of feature, their datatype, Nan threshold etc.
- We will use scipy.stats.ks_2samp for comparing datasets.
- We will compare columns of train file & base file whether they belong to same distribution or not.
- ks_2samp works on principles of Null Hypothesis. It's Null hyp. is 2 distributions are identical, so during comparison we will get pvalue as a result. If our pvalue > 0.05 then both data belong to same distribution.
- We have to go for RETRAINING of model in case of pvalue < 0.05
  - ↓
  - New EDA, other model & approach
- All in all we will look for anomalies like :- target drift, concept drift, High Nan value, High Missing values etc.

→ Config_entity.py

```python
class DataValidationConfig:

    def __init__(self,training_pipeline_config:TrainingPipelineConfig):
        self.data_validation_dir = os.path.join(training_pipeline_config.artifact_dir , "data_validation")
        self.report_file_path=os.path.join(self.data_validation_dir, "report.yaml")
        self.missing_threshold:float = 0.2
        self.base_file_path = os.path.join("aps_failure_training_set1.csv")
```

- We are creating a directory named data_validation.
- A validation Report will be created.

- We save that report in report file path.
- Our report will be of .yaml format.
- Lastly we pick our BASE file for comparison.

→ artifact_entity.py

```python
@dataclass
class DataValidationArtifact:
    report_file_path:str
```

- We define the output location of report file
- This variable will contain the location of it.

→ data_validation.py

- Since validation file include many steps so they have been segregated.

```python
class DataValidation:


    def __init__(self,
                 data_validation_config:config_entity.DataValidationConfig,
                 data_ingestion_artifact:artifact_entity.DataIngestionArtifact):
        try:
            logging.info(f"{'>>'*20} Data Validation {'<<'*20}")
            self.data_validation_config = data_validation_config
            self.data_ingestion_artifact=data_ingestion_artifact
            self.validation_error=dict()
        except Exception as e:
            raise SensorException(e, sys)
```

- We take datavalidation config & ingestion artifact as our parameters for further use.

```python
def drop_missing_values_columns(self,df:pd.DataFrame,report_key_name:str)->Optional[pd.DataFrame]:
    """
    This function will drop column which contains missing value more than specified threshold
    df: Accepts a pandas dataframe
    threshold: Percentage criteria to drop a column
    ==========================================================================================
    returns Pandas DataFrame if atleast a single column is available after missing columns drop else None
    """
    try:

        threshold = self.data_validation_config.missing_threshold
        null_report = df.isna().sum()/df.shape[0]
        #selecting column name which contains null
        logging.info(f"selecting column name which contains null above to {threshold}")
        drop_column_names = null_report[null_report>threshold].index

        logging.info(f"Columns to drop: {list(drop_column_names)}")
        self.validation_error[report_key_name]=list(drop_column_names)
        df.drop(list(drop_column_names),axis=1,inplace=True)

        #return None no columns left
        if len(df.columns)==0:
            return None
        return df
    except Exception as e:
        raise SensorException(e, sys)
```

- This function will read dataframe & check whether if each column has sufficient values or not.
- Drop columns under certain threshold.
- Returns Df with atleast 1 column else returns None.

```python
def is_required_columns_exists(self,base_df:pd.DataFrame,current_df:pd.DataFrame,report_key_name:str)->bool:
    try:

        base_columns = base_df.columns
        current_columns = current_df.columns

        missing_columns = []
        for base_column in base_columns:
            if base_column not in current_columns:
                logging.info(f"Column: [{base} is not available.]")
                missing_columns.append(base_column)

        if len(missing_columns)>0:
            self.validation_error[report_key_name]=missing_columns
            return False
        return True
    except Exception as e:
        raise SensorException(e, sys)
```

- Here we check number of columns.
- Checking if all columns in datasets (Base & current) are same or not.
- If not then it return False.
- Report will be generated in dictionary in case of missing columns.

```python
def data_drift(self,base_df:pd.DataFrame,current_df:pd.DataFrame,report_key_name:str):
    try:
        drift_report=dict()

        base_columns = base_df.columns
        current_columns = current_df.columns

        for base_column in base_columns:
            base_data,current_data = base_df[base_column],current_df[base_column]
            #Null hypothesis is that both column data drawn from same distrubtion

            logging.info(f"Hypothesis {base_column}: {base_data.dtype}, {current_data.dtype} ")
            same_distribution =ks_2samp(base_data,current_data)

            if same_distribution.pvalue>0.05:
                #We are accepting null hypothesis
                drift_report[base_column]={
                    "pvalues":float(same_distribution.pvalue),
                    "same_distribution": True
                }
            else:
                drift_report[base_column]={
                    "pvalues":float(same_distribution.pvalue),
                    "same_distribution":False
                }
                #different distribution

        self.validation_error[report_key_name]=drift_report
    except Exception as e:
        raise SensorException(e, sys)
```

- It won't return us anything, just preparing our report.
- We perform null hypothesis here & if pvalue > 0.05 then same distribution else not.
- We keep writing report in our validation_error report.
- report_key_name parameter takes name of each step for clarity in report.

```python
def initiate_data_validation(self)->artifact_entity.DataValidationArtifact:
    try:
        logging.info(f"Reading base dataframe")
        base_df = pd.read_csv(self.data_validation_config.base_file_path)
        base_df.replace({"na":np.NAN},inplace=True)
        logging.info(f"Replace na value in base df")
        #base_df has na as null
        logging.info(f"Drop null values colums from base df")
        base_df=self.drop_missing_values_columns(df=base_df,report_key_name="missing_values_within_base_dataset")

        logging.info(f"Reading train dataframe")
        train_df = pd.read_csv(self.data_ingestion_artifact.train_file_path)
        logging.info(f"Reading test dataframe")
        test_df = pd.read_csv(self.data_ingestion_artifact.test_file_path)

        logging.info(f"Drop null values colums from train df")
        train_df = self.drop_missing_values_columns(df=train_df,report_key_name="missing_values_within_train_dataset")
        logging.info(f"Drop null values colums from test df")
        test_df = self.drop_missing_values_columns(df=test_df,report_key_name="missing_values_within_test_dataset")

        exclude_columns = ["class"]
        base_df = utils.convert_columns_float(df=base_df, exclude_columns=exclude_columns)
        train_df = utils.convert_columns_float(df=train_df, exclude_columns=exclude_columns)
        test_df = utils.convert_columns_float(df=test_df, exclude_columns=exclude_columns)


        logging.info(f"Is all required columns present in train df")
        train_df_columns_status = self.is_required_columns_exists(base_df=base_df, current_df=train_df,report_key_name="missing_columns_within_train_dataset")
        logging.info(f"Is all required columns present in test df")
        test_df_columns_status = self.is_required_columns_exists(base_df=base_df, current_df=test_df,report_key_name="missing_columns_within_test_dataset")
        if train_df_columns_status:
            logging.info(f"As all column are available in train df hence detecting data drift")
            self.data_drift(base_df=base_df, current_df=train_df,report_key_name="data_drift_within_train_dataset")
        if test_df_columns_status:
            logging.info(f"As all column are available in test df hence detecting data drift")
            self.data_drift(base_df=base_df, current_df=test_df,report_key_name="data_drift_within_test_dataset")

        #write the report
        logging.info("Write reprt in yaml file")
        utils.write_yaml_file(file_path=self.data_validation_config.report_file_path,
        data=self.validation_error)

        data_validation_artifact = artifact_entity.DataValidationArtifact(report_file_path=self.data_validation_config.report_file_path,)
        logging.info(f"Data validation artifact: {data_validation_artifact}")
        return data_validation_artifact
    except Exception as e:
        raise SensorException(e, sys)
```

- In base df, dropped missing values.
- Got our Train & Test df.
- We read them using 'ingestion Artifact step.
- Dropping Null values from Train & Test.
- Converted the columns into float of all dataframes.
- Checks status of train & test if required colums exist.
- Using drift_report to check distributions.
- finally writing down report at each step

→ utils.py

```python
def write_yaml_file(file_path,data:dict):
    try:
        file_dir = os.path.dirname(file_path)
        os.makedirs(file_dir,exist_ok=True)
        with open(file_path,"w") as file_writer:
            yaml.dump(data,file_writer)
    except Exception as e:
        raise SensorException(e, sys)

def convert_columns_float(df:pd.DataFrame,exclude_columns:list)->pd.DataFrame:
    try:
        for column in df.columns:
            if column not in exclude_columns:
                df[column]=df[column].astype('float')
        return df
    except Exception as e:
        raise e
```

- We created a directory for our report file in yaml format.
- The validation_error dictionary that we got in data_validation.py steps, here we dump it.
- other function changing dtype as float.

→ main.py

```python
        data_validation_config = config_entity.DataValidationConfig(training_pipeline_config=training_pipeline_config)
        data_validation = DataValidation(data_validation_config=data_validation_config,
                        data_ingestion_artifact=data_ingestion_artifact)

        data_validation_artifact = data_validation.initiate_data_validation()
```

- Input, operation & output steps.
- We get the report in yaml format.

Result :-

We get a report.yaml file in data_validation folder.

DATA TRANSFORMATION

- We are going to create a pipeline with Simple Imputer with strategy mean.
- We will be using Robust Scaler as we have outliers in data.
- Imputer is used to replace missing values.
- Since we have imbalanced dataset (many neg values & few pos values) we will use SMOTE. Smote populates minority class.
- First we will Increase Record of Minority Class

↓

Simple Imputer

↓

Robust Scaler

} Data Transformation Steps

- In data ingestion step we got train & test data but now after Transformation step we gonna get new record, hence new Train / Test files, so we will create new directory for new data.
- We will require data tranformation steps later too in prediction pipeline so it needs to be saved.
- So we need to save 3 things :-

1. Transformed Training Data
2. Transformed Test Data
3. Transformation Object (using which we can do transformation)

→ Config_entity.py

```python
class DataTransformationConfig:

    def __init__(self,training_pipeline_config:TrainingPipelineConfig):
        self.data_transformation_dir = os.path.join(training_pipeline_config.artifact_dir , "data_transformation")
        self.transform_object_path = os.path.join(self.data_transformation_dir,"transformer",TRANSFORMER_OBJECT_FILE_NAME)
        self.transformed_train_path =  os.path.join(self.data_transformation_dir,"transformed",TRAIN_FILE_NAME.replace("csv","npz"))
        self.transformed_test_path =os.path.join(self.data_transformation_dir,"transformed",TEST_FILE_NAME.replace("csv","npz"))
        self.target_encoder_path = os.path.join(self.data_transformation_dir,"target_encoder",TARGET_ENCODER_OBJECT_FILE_NAME)
```

- npz is numpy array format.

- We will define data transformation directory.
- Encoder path is made for label Encoder.

→ artifact_entity.py

```python
@dataclass
class DataTransformationArtifact:
    transform_object_path:str
    transformed_train_path:str
    transformed_test_path:str
    target_encoder_path:str
```

- Artifact directory location will be defined.

→ data_transformation.py

```
class DataTransformation:


    def __init__(self,data_transformation_config:config_entity.DataTransformationConfig,
                 data_ingestion_artifact:artifact_entity.DataIngestionArtifact):
        try:
            logging.info(f"{'>>'*20} Data Transformation {'<<'*20}")
            self.data_transformation_config=data_transformation_config
            self.data_ingestion_artifact=data_ingestion_artifact
        except Exception as e:
            raise SensorException(e, sys)


    @classmethod
    def get_data_transformer_object(cls)->Pipeline:
        try:
            simple_imputer = SimpleImputer(strategy='constant', fill_value=0)
            robust_scaler =  RobustScaler()
            pipeline = Pipeline(steps=[
                    ('Imputer',simple_imputer),
                    ('RobustScaler',robust_scaler)
                ])
            return pipeline
        except Exception as e:
            raise SensorException(e, sys)
    def initiate_data_transformation(self,) -> artifact_entity.DataTransformationArtifact:
        try:
            #reading training and testing file
            train_df = pd.read_csv(self.data_ingestion_artifact.train_file_path)
            test_df = pd.read_csv(self.data_ingestion_artifact.test_file_path)

            #selecting input feature for train and test dataframe
            input_feature_train_df=train_df.drop(TARGET_COLUMN,axis=1)
            input_feature_test_df=test_df.drop(TARGET_COLUMN,axis=1)

            #selecting target feature for train and test dataframe
            target_feature_train_df = train_df[TARGET_COLUMN]
            target_feature_test_df = test_df[TARGET_COLUMN]

            label_encoder = LabelEncoder()
            label_encoder.fit(target_feature_train_df)

            #transformation on target columns
            target_feature_train_arr = label_encoder.transform(target_feature_train_df)
            target_feature_test_arr = label_encoder.transform(target_feature_test_df)

            transformation_pipleine = DataTransformation.get_data_transformer_object()
            transformation_pipleine.fit(input_feature_train_df)

            #transforming input features
            input_feature_train_arr = transformation_pipleine.transform(input_feature_train_df)
            input_feature_test_arr = transformation_pipleine.transform(input_feature_test_df)
```

- First we will define the pipeline & get data Transf. object.
- Ingestion_artifact is needed to get train / test data.
- Imputer , Rob scaler will be part of pipeline.
- First we read Train / Test file from ingestion artifact.
- We declare "class" column in config.py file , we separated it since it's kinda important & can be changed.
- Input feature means X. target feature means y.
- Using Label Encoder to fit transform our y to make it numerical from categorical.

- fit & Transform our X using Pipeline function.

```
        smt = SMOTETomek(random_state=42)
        logging.info(f"Before resampling in training set Input: {input_feature_train_arr.shape} Target:{target_feature_train_arr.shape}")
        input_feature_train_arr, target_feature_train_arr = smt.fit_resample(input_feature_train_arr, target_feature_train_arr)
        logging.info(f"After resampling in training set Input: {input_feature_train_arr.shape} Target:{target_feature_train_arr.shape}")

        logging.info(f"Before resampling in testing set Input: {input_feature_test_arr.shape} Target:{target_feature_test_arr.shape}")
        input_feature_test_arr, target_feature_test_arr = smt.fit_resample(input_feature_test_arr, target_feature_test_arr)
        logging.info(f"After resampling in testing set Input: {input_feature_test_arr.shape} Target:{target_feature_test_arr.shape}")

        #target encoder
        train_arr = np.c_[input_feature_train_arr, target_feature_train_arr ]
        test_arr = np.c_[input_feature_test_arr, target_feature_test_arr]


        #save numpy array
        utils.save_numpy_array_data(file_path=self.data_transformation_config.transformed_train_path,
                                    array=train_arr)

        utils.save_numpy_array_data(file_path=self.data_transformation_config.transformed_test_path,
                                    array=test_arr)


        utils.save_object(file_path=self.data_transformation_config.transform_object_path,
         obj=transformation_pipleine)

        utils.save_object(file_path=self.data_transformation_config.target_encoder_path,
        obj=label_encoder)
        data_transformation_artifact = artifact_entity.DataTransformationArtifact(
            transform_object_path=self.data_transformation_config.transform_object_path,
            transformed_train_path = self.data_transformation_config.transformed_train_path,
            transformed_test_path = self.data_transformation_config.transformed_test_path,
            target_encoder_path = self.data_transformation_config.target_encoder_path

        )

        logging.info(f"Data transformation object {data_transformation_artifact}")
        return data_transformation_artifact
    except Exception as e:
        raise SensorException(e, sys)
```

- Increasing minority class using SMOTE. on train / test X, y.
- During transformation of data, it convert df to array, so we save this array file using function from utils.
- We use numpy concat to concatenate 2 arrays. np.c_
- Then we save :-
  - → Transformed_train_array
  - → Transformed_test_array
  - → Transform_object_pipline
  - → Target_encoder

- Then we define output path of all 4 files.
- We will use them in Model Trainer.

→ utils.py

```python
def save_object(file_path: str, obj: object) -> None:
    try:
        logging.info("Entered the save_object method of utils")
        os.makedirs(os.path.dirname(file_path), exist_ok=True)
        with open(file_path, "wb") as file_obj:
            dill.dump(obj, file_obj)
        logging.info("Exited the save_object method of utils")
    except Exception as e:
        raise SensorException(e, sys) from e


def load_object(file_path: str, ) -> object:
    try:
        if not os.path.exists(file_path):
            raise Exception(f"The file: {file_path} is not exists")
        with open(file_path, "rb") as file_obj:
            return dill.load(file_obj)
    except Exception as e:
        raise SensorException(e, sys) from e
def save_numpy_array_data(file_path: str, array: np.array):
    """
    Save numpy array data to file
    file_path: str location of file to save
    array: np.array data to save
    """
    try:
        dir_path = os.path.dirname(file_path)
        os.makedirs(dir_path, exist_ok=True)
        with open(file_path, "wb") as file_obj:
            np.save(file_obj, array)
    except Exception as e:
        raise SensorException(e, sys) from e

def load_numpy_array_data(file_path: str) -> np.array:
    """
    load numpy array data from file
    file_path: str location of file to load
    return: np.array data loaded
    """
    try:
        with open(file_path, "rb") as file_obj:
            return np.load(file_obj)
    except Exception as e:
        raise SensorException(e, sys) from e
```

- Created a function to save our object.
- Created a function to load the saved object.
- Since during transformation, our dataframe becomes an array so we created a funct" to save it.
- Created a function to load array. data.

Model Trainer

→ Config - entity.py

```python
class ModelTrainerConfig:

    def __init__(self,training_pipeline_config:TrainingPipelineConfig):
        self.model_trainer_dir = os.path.join(training_pipeline_config.artifact_dir , "model_trainer")
        self.model_path = os.path.join(self.model_trainer_dir,"model",MODEL_FILE_NAME)
        self.expected_score = 0.7
        self.overfitting_threshold = 0.1
```

- We will save our model in this directory.

→ artifact - entity.py

```python
@dataclass
class ModelTrainerArtifact:
    model_path:str
    f1_train_score:float
    f1_test_score:float
```

- Save Model
- f1 Train Score
- f1 Test Score

→ Model_Trainer.py

```python
class ModelTrainer:


    def __init__(self,model_trainer_config:config_entity.ModelTrainerConfig,
                 data_transformation_artifact:artifact_entity.DataTransformationArtifact
                 ):
        try:
            logging.info(f"{'>>'*20} Model Trainer {'<<'*20}")
            self.model_trainer_config=model_trainer_config
            self.data_transformation_artifact=data_transformation_artifact

        except Exception as e:
            raise SensorException(e, sys)

    def fine_tune(self):
        try:
            #Wite code for Grid Search CV
            pass


        except Exception as e:
            raise SensorException(e, sys)

    def train_model(self,x,y):
        try:
            xgb_clf =  XGBClassifier()
            xgb_clf.fit(x,y)
            return xgb_clf
        except Exception as e:
            raise SensorException(e, sys)
```

- Here we need data from data transf. stage.

- Training XGB Classifier Model.

```python
def initiate_model_trainer(self,)->artifact_entity.ModelTrainerArtifact:
    try:
        logging.info(f"Loading train and test array.")
        train_arr = utils.load_numpy_array_data(file_path=self.data_transformation_artifact.transformed_train_path)
        test_arr = utils.load_numpy_array_data(file_path=self.data_transformation_artifact.transformed_test_path)

        logging.info(f"Splitting input and target feature from both train and test arr.")
        x_train,y_train = train_arr[:,:-1],train_arr[:,-1]
        x_test,y_test = test_arr[:,:-1],test_arr[:,-1]

        logging.info(f"Train the model")
        model = self.train_model(x=x_train,y=y_train)

        logging.info(f"Calculating f1 train score")
        yhat_train = model.predict(x_train)
        f1_train_score  =f1_score(y_true=y_train, y_pred=yhat_train)

        logging.info(f"Calculating f1 test score")
        yhat_test = model.predict(x_test)
        f1_test_score  =f1_score(y_true=y_test, y_pred=yhat_test)

        logging.info(f"train score:{f1_train_score} and tests score {f1_test_score}")
        #check for overfitting or underfiiting or expected score
        logging.info(f"Checking if our model is underfitting or not")
        if f1_test_score<self.model_trainer_config.expected_score:
            raise Exception(f"Model is not good as it is not able to give \
            expected accuracy: {self.model_trainer_config.expected_score}: model actual score: {f1_test_score}")

        logging.info(f"Checking if our model is overfiiting or not")
        diff = abs(f1_train_score-f1_test_score)

        if diff>self.model_trainer_config.overfitting_threshold:
            raise Exception(f"Train and test score diff: {diff} is more than overfitting threshold {self.model_trainer_config.overfitting_threshold}")

        #save the trained model
        logging.info(f"Saving mode object")
        utils.save_object(file_path=self.model_trainer_config.model_path, obj=model)

        #prepare artifact
        logging.info(f"Prepare the artifact")
        model_trainer_artifact  = artifact_entity.ModelTrainerArtifact(model_path=self.model_trainer_config.model_path,
        f1_train_score=f1_train_score, f1_test_score=f1_test_score)
        logging.info(f"Model trainer artifact: {model_trainer_artifact}")
        return model_trainer_artifact
    except Exception as e:
        raise SensorException(e, sys)
```

- Loading our Training / Test data which we saved in Transformation step.
- Our target colum is last one in the dataframes.
- We train our Model on XGB Classifier & calc. F1 scores of Train / Test.
- Overfit = Acc. on train but not on test

  Underfit = Nowhere we get Accuracy.

  Expected Score = Its upto us what we expecting.

- We reject model in case of under & overfitting. We check difference in case of overfit. Diff should be less than 10%.
- Then we save model & prepare artifact to get MODEL & Scores.

MODEL EVALUATION

- We pick model.pkl file from Model Trainer Artifact & compare it with other to see if we get even better results.
- Then we send winner model in production.
- This cycle continues everytime we run program.
- From here onward we need 3 files :-
  - → target_encoder.PKL (make target column to numerical)
  - → Transformer.PKL
  - → model.pkl

So we will write a code in new file predictor.py to maintain these files together.

↳ It will read the latest model for prediction.

→ Predictor.py

```python
class ModelResolver:

    def __init__(self,model_registry:str = "saved_models",
                 transformer_dir_name="transformer",
                 target_encoder_dir_name = "target_encoder",
                 model_dir_name = "model"):

        self.model_registry=model_registry
        os.makedirs(self.model_registry,exist_ok=True)
        self.transformer_dir_name = transformer_dir_name
        self.target_encoder_dir_name=target_encoder_dir_name
        self.model_dir_name=model_dir_name
```

- All models will be dumped in "saved models" directory.
-

```python
    def get_latest_dir_path(self)->Optional[str]:
        try:
            dir_names = os.listdir(self.model_registry)
            if len(dir_names)==0:
                return None
            dir_names = list(map(int,dir_names))
            latest_dir_name = max(dir_names)
            return os.path.join(self.model_registry,f"{latest_dir_name}")
        except Exception as e:
            raise e

    def get_latest_model_path(self):
        try:
            latest_dir = self.get_latest_dir_path()
            if latest_dir is None:
                raise Exception(f"Model is not available")
            return os.path.join(latest_dir,self.model_dir_name,MODEL_FILE_NAME)
        except Exception as e:
            raise e
```

- This will pick the latest folder from all. dir_names gets in format of integer so we can easily pick max one.
- Then these numerical directories will have 3 folders of 3 mentioned files.

- Using previous function we get the latest model path. Previous fuction will pick its parent directory then this fenction will get us path of latest Model.

```python
def get_latest_transformer_path(self):
    try:
        latest_dir = self.get_latest_dir_path()
        if latest_dir is None:
            raise Exception(f"Transformer is not available")
        return os.path.join(latest_dir,self.transformer_dir_name,TRANSFORMER_OBJECT_FILE_NAME)
    except Exception as e:
        raise e

def get_latest_target_encoder_path(self):
    try:
        latest_dir = self.get_latest_dir_path()
        if latest_dir is None:
            raise Exception(f"Target encoder is not available")
        return os.path.join(latest_dir,self.target_encoder_dir_name,TARGET_ENCODER_OBJECT_FILE_NAME)
    except Exception as e:
        raise e
```

- These 2 works same as previous one, they help in getting latest path of Transformer & Encoder, respectively.

```python
def get_latest_save_dir_path(self)->str:
    try:
        latest_dir = self.get_latest_dir_path()
        if latest_dir==None:
            return os.path.join(self.model_registry,f"{0}")
        latest_dir_num = int(os.path.basename(self.get_latest_dir_path()))
        return os.path.join(self.model_registry,f"{latest_dir_num+1}")
    except Exception as e:
        raise e

def get_latest_save_model_path(self):
    try:
        latest_dir = self.get_latest_save_dir_path()
        return os.path.join(latest_dir,self.model_dir_name,MODEL_FILE_NAME)
    except Exception as e:
        raise e
```

- This function will assign new numerical value to latest model. It gives next higher number. basename gets us no.

- This func" will make new path for model in new directory.

```python
def get_latest_save_transformer_path(self):
    try:
        latest_dir = self.get_latest_save_dir_path()
        return os.path.join(latest_dir,self.transformer_dir_name,TRANSFORMER_OBJECT_FILE_NAME)
    except Exception as e:
        raise e

def get_latest_save_target_encoder_path(self):
    try:
        latest_dir = self.get_latest_save_dir_path()
        return os.path.join(latest_dir,self.target_encoder_dir_name,TARGET_ENCODER_OBJECT_FILE_NAME)
    except Exception as e:
        raise e
```

- Getting latest location of transformer.

- Getting latest location of encoder.

→ Config – entity.py

```python
class ModelEvaluationConfig:
    def __init__(self,training_pipeline_config:TrainingPipelineConfig):
        self.change_threshold = 0.01
```

• If new model works better than previous model by 1% then we accept it

→ artifact_entity.py

```python
@dataclass
class ModelEvaluationArtifact:
    is_model_accepted:bool
    improved_accuracy:float
```

– We just get if model is Accepted &

– What is improved Accuray.

→ Model_Evaluation.py

```python
class ModelEvaluation:

    def __init__(self,
        model_eval_config:config_entity.ModelEvaluationConfig,
        data_ingestion_artifact:artifact_entity.DataIngestionArtifact,
        data_transformation_artifact:artifact_entity.DataTransformationArtifact,
        model_trainer_artifact:artifact_entity.ModelTrainerArtifact
        ):
        try:
            logging.info(f"{'>>'*20}  Model Evaluation {'<<'*20}")
            self.model_eval_config=model_eval_config
            self.data_ingestion_artifact=data_ingestion_artifact
            self.data_transformation_artifact=data_transformation_artifact
            self.model_trainer_artifact=model_trainer_artifact
            self.model_resolver = ModelResolver()
        except Exception as e:
            raise SensorException(e,sys)
```

- We are taking artifacts of all previous steps.
-

```python
def initiate_model_evaluation(self)->artifact_entity.ModelEvaluationArtifact:
    try:
        #if saved model folder has model the we will compare
        #which model is best trained or the model from saved model folder

        logging.info("if saved model folder has model the we will compare "
        "which model is best trained or the model from saved model folder")
        latest_dir_path = self.model_resolver.get_latest_dir_path()
        if latest_dir_path==None:
            model_eval_artifact = artifact_entity.ModelEvaluationArtifact(is_model_accepted=True,
            improved_accuracy=None)
            logging.info(f"Model evaluation artifact: {model_eval_artifact}")
            return model_eval_artifact


        #Finding location of transformer model and target encoder
        logging.info("Finding location of transformer model and target encoder")
        transformer_path = self.model_resolver.get_latest_transformer_path()
        model_path = self.model_resolver.get_latest_model_path()
        target_encoder_path = self.model_resolver.get_latest_target_encoder_path()
```

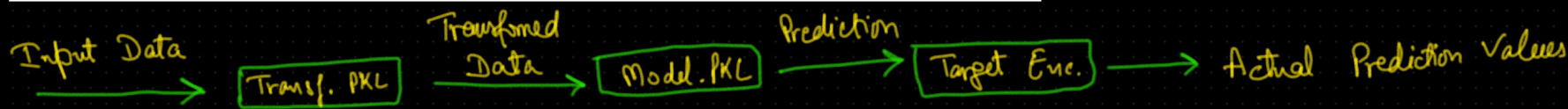
- Function to compare models.
- At first there will be no model so we don't have to compare.
- Then in other case we find our latest model, transformer & target encoder.

```
        logging.info("Previous trained objects of transformer, model and target encoder")
        #Previous trained  objects
        transformer = load_object(file_path=transformer_path)
        model = load_object(file_path=model_path)
        target_encoder = load_object(file_path=target_encoder_path)


        logging.info("Currently trained model objects")
        #Currently trained model objects
        current_transformer = load_object(file_path=self.data_transformation_artifact.transform_object_path)
        current_model  = load_object(file_path=self.model_trainer_artifact.model_path)
        current_target_encoder = load_object(file_path=self.data_transformation_artifact.target_encoder_path)
```

COMPARISON - - →

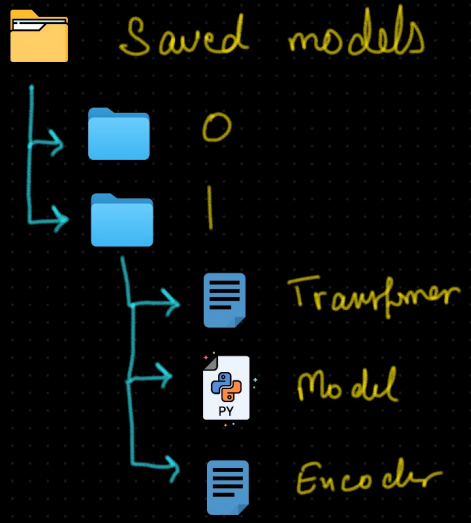
```
        test_df = pd.read_csv(self.data_ingestion_artifact.test_file_path)
        target_df = test_df[TARGET_COLUMN]
        y_true =target_encoder.transform(target_df)
        # accuracy using previous trained model
        input_feature_name = list(transformer.feature_names_in_)
        input_arr =transformer.transform(test_df[input_feature_name])
        y_pred = model.predict(input_arr)
        print(f"Prediction using previous model: {target_encoder.inverse_transform(y_pred[:5])}")
        previous_model_score = f1_score(y_true=y_true, y_pred=y_pred)
        logging.info(f"Accuracy using previous trained model: {previous_model_score}")

        # accuracy using current trained model
        input_feature_name = list(current_transformer.feature_names_in_)
        input_arr =current_transformer.transform(test_df[input_feature_name])
        y_pred = current_model.predict(input_arr)
        y_true =current_target_encoder.transform(target_df)
        print(f"Prediction using trained model: {current_target_encoder.inverse_transform(y_pred[:5])}")
        current_model_score = f1_score(y_true=y_true, y_pred=y_pred)
        logging.info(f"Accuracy using current trained model: {current_model_score}")
        if current_model_score<=previous_model_score:
            logging.info(f"Current trained model is not better than previous model")
            raise Exception("Current trained model is not better than previous model")

        model_eval_artifact = artifact_entity.ModelEvaluationArtifact(is_model_accepted=True,
        improved_accuracy=current_model_score-previous_model_score)
        logging.info(f"Model eval artifact: {model_eval_artifact}")
        return model_eval_artifact
    except Exception as e:
        raise SensorException(e,sys)
```

- After finding, we will load these 3 files. (Copying model)
- Then we load current model, Tr. & Enc. we have.
- ModelResolver is most helpful class.
- We perform comparison on test df.
- We had to change our target values from cat. to num. but then we use encoder.inverse_transform to revert changes.
- If current model isn't better than previous, we just raise Exception else we get new model improved accuracy & move to Pusher phase.



Model Pusher

- We are going to create Saved Models directory.
- This directory will have numbered folders which will be created whenever we get a better model.
- Then we will pick the folder with maximum number because it's best performing model.
- Using Prediction API we will push them to the cloud.



→ Config - entity.py

```python
class ModelPusherConfig:

    def __init__(self,training_pipeline_config:TrainingPipelineConfig):
        self.model_pusher_dir = os.path.join(training_pipeline_config.artifact_dir , "model_pusher")
        self.saved_model_dir = os.path.join("saved_models")
        self.pusher_model_dir = os.path.join(self.model_pusher_dir,"saved_models")
        self.pusher_model_path = os.path.join(self.pusher_model_dir,MODEL_FILE_NAME)
        self.pusher_transformer_path = os.path.join(self.pusher_model_dir,TRANSFORMER_OBJECT_FILE_NAME)
        self.pusher_target_encoder_path = os.path.join(self.pusher_model_dir,TARGET_ENCODER_OBJECT_FILE_NAME)
```

- We want to save our models in Artifact Directory as well as outside of it.

- Main folder > Artifact > Timestamp folder > Model Pusher > Saved Models
- Main folder > Saved Models

⇒ Artifact_entity.py

```python
@dataclass
class ModelPusherArtifact:
    pusher_model_dir:str
    saved_model_dir:str
```

- Here we need 2 outputs :-
  Pusher Model Directory &
  Saved Model Directory.
- Both inside Artifact folder.

→ Model_Pusher.py

```python
class ModelPusher:

    def __init__(self,model_pusher_config:ModelPusherConfig,
    data_transformation_artifact:DataTransformationArtifact,
    model_trainer_artifact:ModelTrainerArtifact):
        try:
            logging.info(f"{'>>'*20} Data Transformation {'<<'*20}")
            self.model_pusher_config=model_pusher_config
            self.data_transformation_artifact=data_transformation_artifact
            self.model_trainer_artifact=model_trainer_artifact
            self.model_resolver = ModelResolver(model_registry=self.model_pusher_config.saved_model_dir)
        except Exception as e:
            raise SensorException(e, sys)
    def initiate_model_pusher(self,)->ModelPusherArtifact:
        try:
            #load object
            logging.info(f"Loading transformer model and target encoder")
            transformer = load_object(file_path=self.data_transformation_artifact.transform_object_path)
            model = load_object(file_path=self.model_trainer_artifact.model_path)
            target_encoder = load_object(file_path=self.data_transformation_artifact.target_encoder_path)

            #model pusher dir
            logging.info(f"Saving model into model pusher directory")
            save_object(file_path=self.model_pusher_config.pusher_transformer_path, obj=transformer)
            save_object(file_path=self.model_pusher_config.pusher_model_path, obj=model)
            save_object(file_path=self.model_pusher_config.pusher_target_encoder_path, obj=target_encoder)


            #saved model dir
            logging.info(f"Saving model in saved model dir")
            transformer_path=self.model_resolver.get_latest_save_transformer_path()
            model_path=self.model_resolver.get_latest_save_model_path()
            target_encoder_path=self.model_resolver.get_latest_save_target_encoder_path()

            save_object(file_path=transformer_path, obj=transformer)
            save_object(file_path=model_path, obj=model)
            save_object(file_path=target_encoder_path, obj=target_encoder)

            model_pusher_artifact = ModelPusherArtifact(pusher_model_dir=self.model_pusher_config.pusher_model_dir,
             saved_model_dir=self.model_pusher_config.saved_model_dir)
            logging.info(f"Model pusher artifact: {model_pusher_artifact}")
            return model_pusher_artifact
        except Exception as e:
            raise SensorException(e, sys)
```

Provided Directory

- We are going to follow 3 steps:-
  Load objects
  Save them in Model Pusher
  Save them in Saved Models
- We loaded all 3 objects from their different directories.
- Save them in Model Pusher Artifact Directory.
- Then we get latest path of all objects in new single directory & saved them outside artifact.
- Same Object but saved at 2 different places.

TRAINING PIPELINE

→ Training_pipeline.py

```python
from sensor.logger import logging
from sensor.exception import SensorException
from sensor.utils import get_collection_as_dataframe
import sys,os
from sensor.entity import config_entity
from sensor.components.data_ingestion import DataIngestion
from sensor.components.data_validation import DataValidation
from sensor.components.data_transformation import DataTransformation
from sensor.components.model_trainer import ModelTrainer
from sensor.components.model_evaluation import ModelEvaluation
from sensor.components.model_pusher import ModelPusher


def start_training_pipeline():
    try:
        training_pipeline_config = config_entity.TrainingPipelineConfig()

        #data ingestion
        data_ingestion_config  = config_entity.DataIngestionConfig(training_pipeline_config=training_pipeline_config)
        print(data_ingestion_config.to_dict())
        data_ingestion = DataIngestion(data_ingestion_config=data_ingestion_config)
        data_ingestion_artifact = data_ingestion.initiate_data_ingestion()
```

- we write whole execution code here.
- from Data Ingestion to Model Pusher
- whole code is in sequence of Pipeline.

```python
    #data validation
    data_validation_config = config_entity.DataValidationConfig(training_pipeline_config=training_pipeline_config)
    data_validation = DataValidation(data_validation_config=data_validation_config,
                    data_ingestion_artifact=data_ingestion_artifact)

    data_validation_artifact = data_validation.initiate_data_validation()

    #data transformation
    data_transformation_config = config_entity.DataTransformationConfig(training_pipeline_config=training_pipeline_config)
    data_transformation = DataTransformation(data_transformation_config=data_transformation_config,
    data_ingestion_artifact=data_ingestion_artifact)
    data_transformation_artifact = data_transformation.initiate_data_transformation()

    #model trainer
    model_trainer_config = config_entity.ModelTrainerConfig(training_pipeline_config=training_pipeline_config)
    model_trainer = ModelTrainer(model_trainer_config=model_trainer_config, data_transformation_artifact=data_transformation_artifact)
    model_trainer_artifact = model_trainer.initiate_model_trainer()
```

- start_training_pipeline() will later be called in main.py.

```
    #model evaluation
    model_eval_config = config_entity.ModelEvaluationConfig(training_pipeline_config=training_pipeline_config)
    model_eval  = ModelEvaluation(model_eval_config=model_eval_config,
    data_ingestion_artifact=data_ingestion_artifact,
    data_transformation_artifact=data_transformation_artifact,
    model_trainer_artifact=model_trainer_artifact)
    model_eval_artifact = model_eval.initiate_model_evaluation()

    #model pusher
    model_pusher_config = config_entity.ModelPusherConfig(training_pipeline_config)

    model_pusher = ModelPusher(model_pusher_config=model_pusher_config,
            data_transformation_artifact=data_transformation_artifact,
            model_trainer_artifact=model_trainer_artifact)

    model_pusher_artifact = model_pusher.initiate_model_pusher()
except Exception as e:
    raise SensorException(e, sys)
```

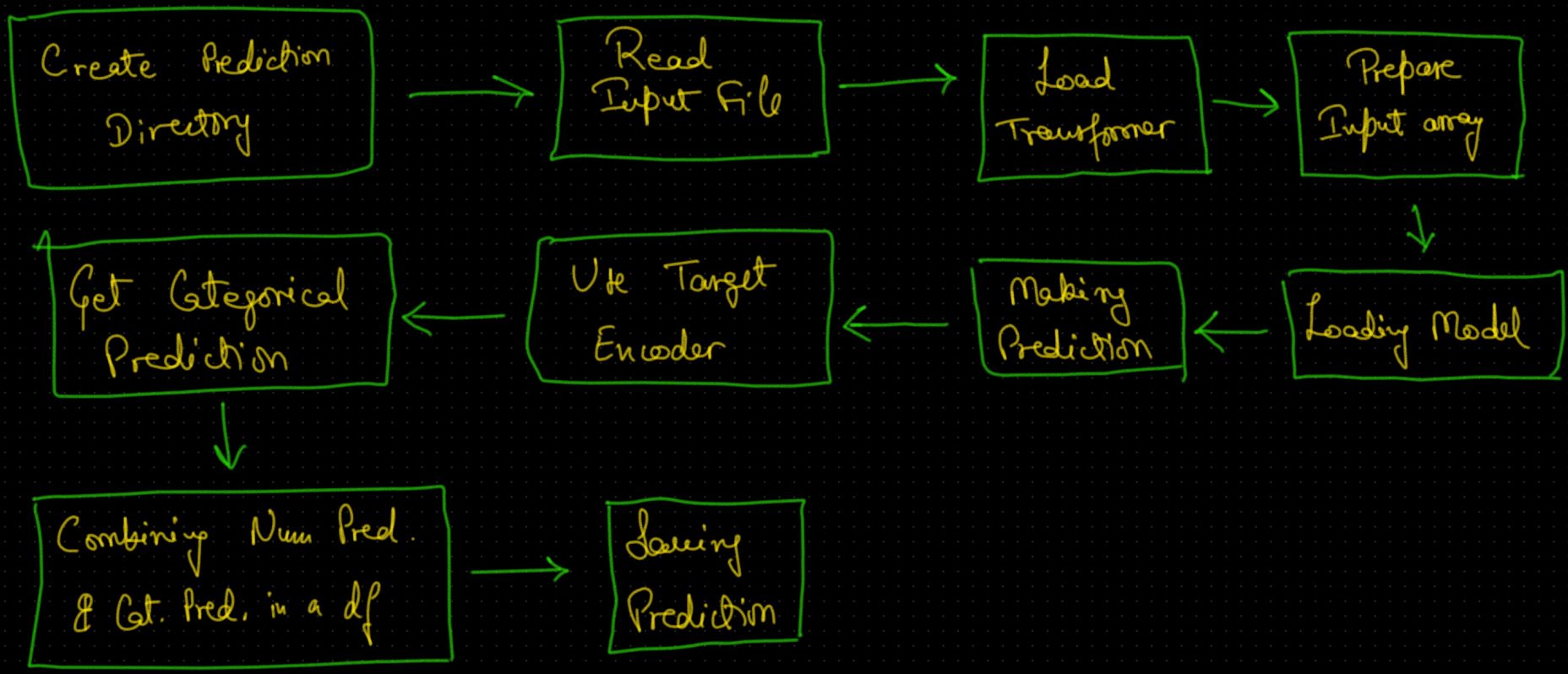
→ Batch_Prediction.Py

```
from sensor.exception import SensorException
from sensor.logger import logging
from sensor.predictor import ModelResolver
import pandas as pd
from sensor.utils import load_object
import os,sys
from datetime import datetime
PREDICTION_DIR="prediction"

import numpy as np
def start_batch_prediction(input_file_path):
    try:
        os.makedirs(PREDICTION_DIR,exist_ok=True)
        logging.info(f"Creating model resolver object")
        model_resolver = ModelResolver(model_registry="saved_models")
        logging.info(f"Reading file :{input_file_path}")
        df = pd.read_csv(input_file_path)
        df.replace({"na":np.NAN},inplace=True)
        #validation
```

- We are creating a function where all files will go through predictions & then will be saved in a new Prediction folder.

```
        logging.info(f"Loading transformer to transform dataset")
        transformer = load_object(file_path=model_resolver.get_latest_transformer_path())

        input_feature_names =  list(transformer.feature_names_in_)
        input_arr = transformer.transform(df[input_feature_names])

        logging.info(f"Loading model to make prediction")
        model = load_object(file_path=model_resolver.get_latest_model_path())
        prediction = model.predict(input_arr)

        logging.info(f"Target encoder to convert predicted column into categorical")
        target_encoder = load_object(file_path=model_resolver.get_latest_target_encoder_path())

        cat_prediction = target_encoder.inverse_transform(prediction)

        df["prediction"]=prediction
        df["cat_pred"]=cat_prediction

        prediction_file_name = os.path.basename(input_file_path).replace(".csv",f"{datetime.now().strftime('%m%d%Y__%H%M%S')}.csv")
        prediction_file_path = os.path.join(PREDICTION_DIR,prediction_file_name)
        df.to_csv(prediction_file_path,index=False,header=True)
        return prediction_file_path
    except Exception as e:
        raise SensorException(e, sys)
```

- All files will be saved in csv format along with timestamp in their name.
- In return we getting location of Prediction.



DEPLOYMENT

Machine Learning Operations (MLOPS) :-

1. ML Development
2. Continuous Training
    a. Update model on new datasets
    b. Performance Improvement of model
3. Deploying model as
    a. Endpoint API
    b. Batch Prediction (our case)
4. Model Monitoring

- To run a python code we need to :-

  1. Create Python env.
  2. Install Dependencies
  3. Run the Script

- Now we need to export our source Code to VM using DOCKER as Docker packages a code.

  → Docker Dockerize our code, then we test Docker image in local system, if it works then it will work in Virtual Machine too. Docker Remove Dependencies from local.

- Docker has these steps :-

  1. Choose a base machine
  2. Copy code from local system to Docker Machine.
  3. Install Dependencies.
  4. Run the Script

Using these steps we will create a Docker Image.

- We are going to setup a Scheduler which will trigger our pipeline at a certain Interval. We can specify its frequency.
  - ↳ We will use Airflow library for Scheduling.
- In Deployment we will just run our Docker Image in Cloud VM.
- We will use Github Actions (set of commands), Github provides us a machine. The command we write there get executed one by one,
- We store Docker Image in ECR by Amazon.
- Github Action will help automate everything & take load off Local Machine
- Our Virtual Machine will by EC2
- Deployment is done once we download Docker Image in EC2 & sucessfully run that image.
-

→ Dockerfile (Creating Docker Image)

```
FROM python:3.8
USER root
RUN mkdir /app
COPY . /app/
WORKDIR /app/
RUN pip3 install -r requirements.txt
ENV AIRFLOW_HOME="/app/airflow"
ENV AIRFLOW__CORE__DAGBAG_IMPORT_TIMEOUT=1000
ENV AIRFLOW__CORE__ENABLE_XCOM_PICKLING=True
RUN airflow db init
RUN airflow users create  -e avnish@ineuron.ai -f Avnish -l Yadav -p admin -r Admin  -u admin
RUN chmod 777 start.sh
RUN apt update -y && apt install awscli -y
ENTRYPOINT [ "/bin/sh" ]
CMD ["start.sh"]
```

- Python 3.8 is base image
- Make ourself root user.
- We create a folder to copy our code.
- Copy everything from our direct. to /app folder
- Set working directory to /app
- Install dependencies using .txt

- We set environment variable for Airflow.
- We set timer of 1000 ms
- Then we run database of airflow.
- Create an user account with complete Credentials as admin.
- Start.sh is our file with commands to launch airflow scheduler & webserver which gives us UI . In Dockerfile we give permission for execution of this command
- Aws cli is Aws Command line Interface . We install it to store outputs in S3 Bucket.
- Then we define entrypoint to tell where our shell is located.
- Lastly we run start.sh

- start.sh

```
#!bin/sh
nohup airflow scheduler &
airflow webserver
```

→ docker-compose.yaml

```
version: "2"
services:
 application:
    image: ${IMAGE_NAME}
    container_name: sensor
    ports:
      - "8080:8080"
    environment:
      - AWS_ACCESS_KEY_ID=${AWS_ACCESS_KEY_ID}
      - AWS_SECRET_ACCESS_KEY=${AWS_SECRET_ACCESS_KEY}
      - AWS_DEFAULT_REGION=${AWS_DEFAULT_REGION}
      - MONGO_DB_URL=${MONGO_DB_URL}
      - BUCKET_NAME=${BUCKET_NAME}
```

- We use this file to run this project
- We setup all the required information of our image.
-

→ Then we create an .dockerignore file & write all file names which we don't need in our image.

→ We create a directory Airflow > dags >
- batch_prediction.py
- training_pipeline.py

- In training_pipeline.py of dags folder, we create a DAG. We define all 6 steps of Pipeline in airflow.
- We import start_training_pipeline() function from sensor.pipeline.training_pipeline & we it to run all 6 steps.
- Airflow has PythonOperator function which is designed to run Python codes. So whenever we run airflow, it triggers training_pipeline function.
- DAG has all the parameters of when to run code, how many retries etc.
- We write a function to sync our artifacts into S3 bucket. It sync them from /app/ directory.
- training_pipeline >> sync_data_to_s3 ensures first we run pipeline & then sync to s3 bucket. >> ensures the order of execution.
- In batch_prediction.py of Dags we create a DAG in which we define 3 steps of Batch Prediction
  - Download files function which creates a directory in bucket.
  - Batch_Prediction function which does Batch Prediction of files downloaded in previous step.
  - Sync Prediction Directory to S3 Bucket

- Download Input files >> Generate Prediction file >> upload Prediction files.

→ We create a directory .github > workflows
  - main.yaml (Doing it for Github Action)

- Here we will write github events to trigger our code. Whenever we push to main branch then trigger will happen.
- Then we set Permissions.
- Then we define jobs.
  - ↳ Job 1 → Build Docker Image
  - ↳ Job 2 → Push image in ECR
- These jobs have steps which we define in our code like which machine it runs on, what checkout method for code copying it uses, softwares to install in our machine, then we configure AWS credentials in which image will be pushed, within here we login to ECR, then we define name of ECR Repo. & give tag to image, lastly we run command Create image & Push to ECR.

- After Building & Pushing Image to ECR, we write another job for Continuous Deployment.

↳
- It require Image to be built & Pushed to ECR.
- We run it on self hosted EC2 machine
- It has it's steps :-
  - Checkout (copy code)
  - Configure AWS Credentials
  - Install Required libraries
  - Login to ECR
  - Pull Docker Image
  - Run image to serve users
  - Above step require 5 variables as key.

Main.yaml will automate the deployment steps by running in Github Actions.

- We now need these 6 credentials & put them in secret file:-
  - → AWS Access Key ID
  - → AWS Secret Access Key
  - → AWS Region
  - → AWS ECR Login URI
  - → ECR Repo Name
  - → Bucket Name
  - → Mongo DB URL

Using AWS IAM we created a user & got Access Key & Secret Key. Region is south-1.

Using AWS ECR we created a repository & got ECR URI & repo name.

Using AWS S3 we created a bucket & saved its name.

- Now last step is to create EC2 machine ... We do it using AWS EC2 :-
  - Click on Instances on left Menu & then Launch Instance Button.
  - Write a name & choose Ubuntu Machine
  - Create Key pair & it will download a file
  - Set Configure Storage a 30 GiB
  - Click Instance
  - Then we view Instance
  - Select Instance & you will see its configurations at bottom.
  - Go to Security & click Security Groups hyperlink

- Now we can see Inbound Rules & click on Edit Inbound Rules
- Add Rule & select All Traffic with Source as Anywhere IPv4
- Go to Instances, Pick your Instance & click CONNECT on top.
- We click connect & a new machine opens in a new tab.

In our VM :-

- Install Docker using ubuntu command mentioned in Readme file. A total of 4 lines to be executed 1-by-1.
- Go to your Github Repository and go to settings & then select Action > Runners and create one. We will select Linux & will see command to be used in VM.
- Execute these 4 lines 1-by-1.
- Type ls & you can now see files there.
- Now copy Command from Configure tab in github.
- Press Enter, type self-hosted, Press Enter & we are done.
- Copy Paste last ./run command & we are connected to Github.

- Lastly go to Github Repo settings & look for Secrets.
- Click on Actions & create new secret.
- In Name tab enter AWS_ACCESS_KEY_ID & paste its value inside Secret tab & click ADD Secret.
- Do the same step for other 6 credentials above mentioned.

Lastly open Github Actions & Run the job.

→ Connect to your Instance, it will open VM in new tab.

→ Enter these commands :-

```
ls
cd actions-runner/
ls
./run.sh
```

It will connect back to Github incase you close browser of VM. Also it will turn runner as ACTIVE in github Actions. incase you find it offline.

1. Watch Your videos again and again.
2. Read your code couple of times
3. Divide your project into small components.
    - Like we have 6 components so do each component separately in Jupyter notebook.