Structure of Data Science Project

This document will help you understand the basic skeleton of the data science project.

Step 1:

Understand the problem statement.

Step 2:

Collect the data from various sources. It could be in S3, Google Big Query, MongoDB, or a SQL server. Generally, this is the task of the data engineering team. Data engineers can collect data from multiple devices. This is done using Kafka, which is used to load streaming data. The producer app (shown in figure 1) could be a variety of IoT devices or the data source. A consumer app (shown in figure 2) could be MongoDB, an S3 bucket, or any other type of data storage platform. Data cannot be directly loaded into MongoDB as it will create a performance issue for MongoDB, so Kafka server is needed. The reason for this is that it is difficult to determine which requests should be prioritized.

Kafka is horizontally scalable, i.e., it can handle multiple requests at the same time, and in the future, if the number of requests increases, simply increase the size of Kafka.

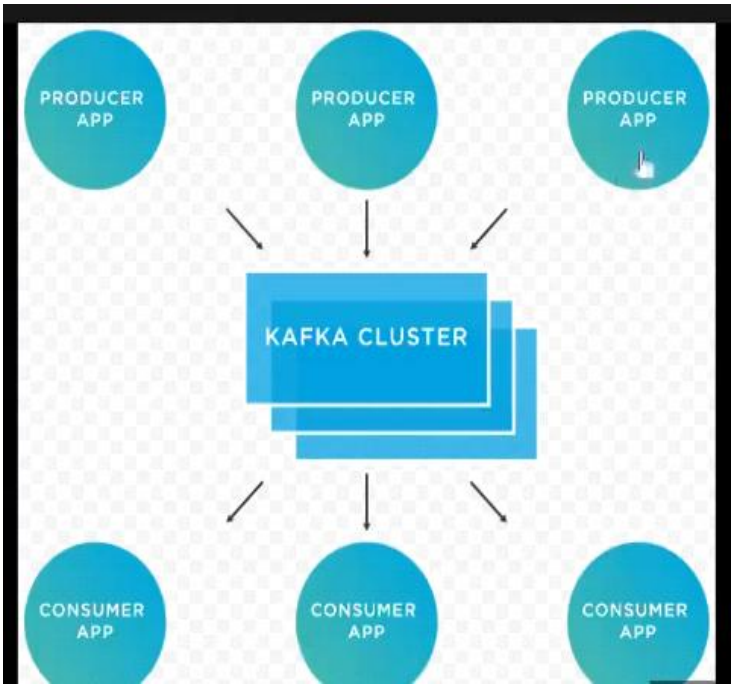

Figure 1: Sample of Kafka Cluster

Step 3:

Jump to any python IDE (PyCharm/VsCode).

- Connect with any of your data base (MongoDB or SQL Server or anything) using python.
- To start the Data Science part of the project, load the dataset in your Data Base. Name the file as -> `data_dump.py`
- Since the project need some third-party library for the implementation of the project. And those third-party libraries need to be install before the projects starts. So, all those libraries needed are written in the file named `requirements.txt`. Hence a file name `requirements.txt` is created and all the required libraries are written. Then the same need to be install using `pip install -r requirements.txt`.
- Install Git
- Check if git is installed. Use command "`git`" or "`git --version`".
- Now jump to GitHub and create a repo for the project. `ReadMe.md` file will be created inside the IDE so need not to create it. Add `.gitignore` and select the template "python". Now create a repo. Get the URL of the GitHub repo.
- Next come configuring the GitHub and for that few of the necessary commands are as follows.
    - `git remote -v` (To check if any repo is linked to the GitHub)
    - `git remote remove origin` (remove the existing repo)
    - `git remote add origin <REMOTE_URL>` (Sets the new remote)
    - `git add file_name` (Note: You can give file_name to add specific file or use "." to add everything to staging area)
    - `git status` (to see the changes that will be pushed to GitHub)
    - `git commit -m "message"` (Create commits)
    - `git push origin main` (Note: origin--> contains URL to your GitHub repo main--> is your branch name)
    - `git push origin main -f` (To push your changes forcefully.)
    - `git pull origin main` (To pull changes from GitHub repo)

  One thing to note here, the user is responsible for creating various versions.

Step 4:

For any python project to start the first file that need to be created is "`setup.py`". Let say if there is need to convert the code into a library format, and for the same we need to have "`setup.py`" file. (Example like we use pip install pandas; this installation is possible because of the file `setup.py`). It should have following details like so:

```
from setuptools import find_packages, setup
      from typing import List
      setup(
            name = 'name of project',
            version = "0.0.0",
            author = "username",
            author_email = "email_add",
            packages = find_packages(),
            install_requires = get_requirements()
            )
```

Purpose of line <`packages = find_packages()`> It will look all the folder for the source code and install it as packages. Question is how the `find_packages()` function will know if a folder is a package or not. For this it will look for `__init__.py` file inside every folder and if found the same folder will be consider as a python package.

Purpose of the line `install_requires = get_requirements()` is to install the third parties libraries that are needed for out project. For example, Pandas, NumPy etc. For this a function named `get_requirements` need to be created and it will contain all the dependencies/libraries required for the project. This function will read the file `requirements.txt` and return the list of third-party python packages need to be installed.

To make our package as a library a line "`-e .`" is added at the end of the file named `requirements.txt`. Since the "`-e .`" is not any package, so the function `get_requirements()` should return the list of required package and it need not "`-e .`" in the list.

Now write the package that would be needed for the project and then install the packages using the command line `pip install -r requirements.txt`. Doing so will create a folder name `sensor.egg-info` and it will contain the details of the library.

Step 5: Overview of ML pipeline

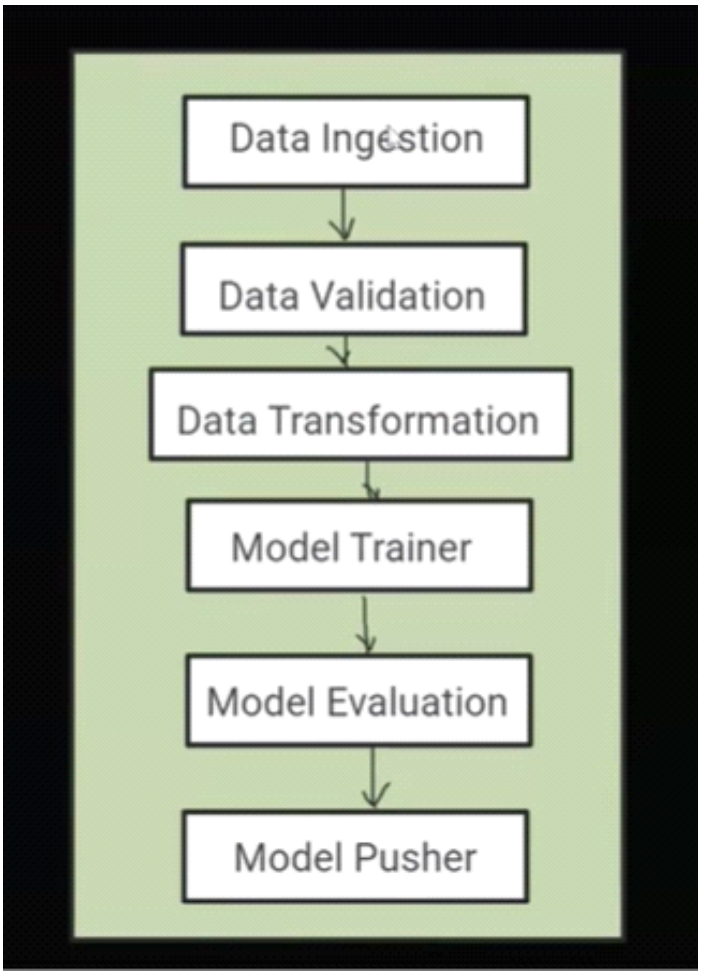

Figure 2: Training pipeline structure

Each component or stage will require an input and generate some output. There are a few standard terms related to this.

Artifact: An artifact is a machine learning term that describes the output (a fully trained model, a checkpoint, or a file) created by the training process. Means it is anything generated during the training pipeline. It could be anything from a model to an Excel file to an image file to a list.

Configuration: Input to a training stage is called "configuration." This may change from one industry to another.

An important thing to note is that the project manager should be aware of the data structure at any given input or output stage of any pipeline.

Since every stage will have an artifact and configuration. Hence information of the same need to be stored.

Various files that will contain information about configuration and artifact are created. Such a file is created under the `components` folder. The number of files in the folder will be parallel to the number of stages in the pipeline. For example, data_ingestion.py, data_validation.py, data_transformation.py, model_trainer.py, model_evaluation.py, model_pusher.py, and model_trainer.py

Also, the folder "pipelines" was created; this will store the information about the stages of pipelines. Obviously, the pipeline will be divided into two stages: training and testing. As a result, training_pipeline.py and testing_pipeline.py will be created.

Also, the folder `entity` is created. This folder will hold the information of physical data or object or graph or chart etc. It will contain two files `config_entity.py` (that holds the information of input) and `artifact_entity.py` (that holds the information of output). Now the question is how many components does each file will store. A simple answer to this is question is, number of components will be same as that of number of pipeline stages. Now the file name `entity/config_entity.py & entity/artifact_entity.py` will have class that is related to every stage of a pipeline and these classes will hold the information of each stage. For example, `DataIngestionConfig, DataValidationConfig, DataTransformationConfig, ModelTrainerConfig, ModelEvaluationConfig, ModelPusherConfig, TrainingPipelineConfig`

Another file `utils.py` is created and it stores the helper function. Helper functions are functions that are used in the project. For example, to store the data into the database or to fetch the data from the database or to save the model or to load the model or to generate the report and many more.

Finally, two files are created `exception.py and logger.py`. We never want that the project we work on shall crash, whatever may be the situation and for the same we use exception handling. As per the project, custom exception is created. We need to keep the record of every incident that is happening within codes and for the same we use logger file. At this stage content of logger and exception handling file is written as per the requirements of project.

It is best practice not to hardcode the connections or similar things. For the same a file name `config.py` is created that will provide the details of connection let say

mongoDB. Since the project will be uploaded to GitHub, so it is not good practice to load the username or password or any other sensitive information to GitHub. To hide this there is a file called "`.env`" created that stores such sensitive information. Information from the `.env` file can be extracted for use and it won't be uploaded to GitHub. So two files will be created `config.py` and `.env` where *.env* will hold the sensitive information and *config.py* will extract the sensitive info and make it available to the project. Since we require that the sensitive information should be available to us whenever we are writing the code and for the same under the `__init__.py` file we will write following commands

```
from dotenv import load_dotenv
load_dotenv()
```

So, this will directly load the sensitive info from the .env file and make it available to the project.

Step 6: Data Ingestion

As the data is made available to database (E.g., MongoDB, SQL, S3 bucket etc.) it needs to be extracted. Remember the use of `utils.py` file. Now under the file, codes to extract data from the database will be written. So, it will contain the function that can extract the data from database. The structure of the function would be like so.

Function name: get_collection_as_dataframe

- Arguments:
  - Data Base name,
  - Table name or any information where data is stored
- Description: Get data from the database and convert it to Data Frame
- Return: Pandas Data Frame

Function: write_yaml_file

- Argument:
  - File_path: file path of yaml file
  - Data: This is of type dictionary that will store the required data
- Description: Create a report on yaml file.

It is needless to quote that every python file will have logs and exception.

Next comes to create the training pipeline. Since every stage of the training pipeline will have an output and the same will be stored in folder `artifact`. And every time the pipeline is executed a new folder is created within the artifact folder with the name as current time stamp. It is like the naming of log file. Since this folder needs to be created as the training pipeline is called for, we will write this code accordingly in `__init__.py` under the training pipeline present in the `entity/config_entity.py` file.

Class: TrainingPipelineConfig

- Argument: None
- Variable: A global variable `self.artifact_dir` is created that holds information of artifact folder.
- Description: Create a directory named artifact that will store the output from every stage of the training pipeline.

Now we must define the input and output of the data ingestion components. Remember, config is input and artifact is output and we have a folder named entity

that holds the information of the same. Moving to `entity/config_entity.py` file to define a class that will provide the configuration of data that will be ingested. The details of this class are as follows:

Class: DataIngestionConfig

- Argument: training_pipeline_config
- Description: This class will have the details of data source (like collection/table name and database name. It also stores the location of train and test file).
- Variables:
  - database_name: (To fetch the data)
  - collection_name: (To fetch the data)
  - data_ingestion_dir: (Directory to store output of data ingestion phase)
  - feature_store_file_path: (Give the location of data to be stored that will be fetched from the database)
  - train_file_path: (Give the location of the training file path)
  - test_file_path: (Give the location of the test file path)
  - test_size: (This will provide the ratio in which data will be split)

If we need to save above details as dictionary then we will create a method as follows:

```
def to_dict(self) -> dict():
      self.__dict__
```

Now all the requirements are fulfilled to start the data ingestion process. Data ingestion starts with the `components/data_ingestion.py` file. As the input type is defined in `config_entity.py` and output type is defined in `artifict_entity.py`, we will import the same.

This file will have following class

Class: DataIngestion

- Arguments:
  - data_ingestion_config (Its type is already defined in the config_entity.py file.)
- Object:
  - data_ingestion_config

Next to initiate data ingestion process a method is written and its output is defined in artifact_entity.py

Method: initiate_data_ingestion()

- Arguments: None
- Description:
    - It will take the data from the database and convert the same into the Data Frame.
    - If any sort or pre-processing is required in the dataset, it need to be done here.
    - Directory where the data need to be stored is created (Its detail can be found in config_entity.py file).
    - Then data is stored in that directory
    - Data is split into the train dataset and test dataset. Train test ratio is already defined in the config_entity.py file
    - Store the train and test dataset into the respective directories. Details of the directory can be fetched form the config_entity.py file.
    - Finally create the output structure of data ingestion artifact.
- Return: An object with type defined in artifact_entity.py

This concludes the data ingestion stage.

Step 7: Data Validation

Let us first understand what data validation is? Before proceeding with any Data Science Project, EDA need to do prior to the start of the project. Let say the dataset had 10 feature column and a target column. While EDA it was observed that in one column, data was normally distributed. Another column had categorical data that had 3 categories. Now it is responsibility of the Data Scientist to that the to check that the new data that came for the prediction also falls under the same distribution or have same categories or have same number of columns and many more aspects. For example, the column in new dataset should also have same type of distribution as that of original dataset or the categorical column in new dataset should have same number of categories as that of the original data. The conclusion that can be drawn is that only valid data should be passed further for the data training process not doing so will result in improper prediction. Further, if the dataset is changed then there is need to retrain our model.

If we need to validate some data, we need to have some base information, that base information will come from the file dataset provided by the client. To validate the dataset, we will use stats.ks2_sample. This test compares the underlying continuous distributions F(x) and G(x) of two independent samples. The null hypothesis is that the two distributions are identical, F(x)=G(x) for all x and the alternate hypothesis is that they are not identical.

Now we will compare each column of train and test file from the sample dataset provided by the client. If null hypothesis is true (when p value is less than .05) i.e., train and test dataset are from same distribution, then we can proceed further within the pipeline else we will again retrain our model. This process where test and train dataset do not belong from the sample dataset is called Data Drift.
This will decide if there is a need to retrain is required or not.

Next, we will move for anomaly detection in our dataset. Anomalies include and not just limited to:

- High null value
- Missing columns
- Outliers
- Got an extra category in categorical column (For example, if the original data set had Gender column that has categories as male and female, but the new data set had the three categories in gender column male, female and others.)
- Target drift

- Concept Drift means that statistical properties of the target variable, which the model is trying to predict, change over time in unforeseen ways. For example, previous model was built with 2 categories in target column and after a while model got 3 categories.

Data validation starts with getting the dataset provided by the client. This dataset will give the base information.
Let's implement our understanding in codes with input to the validation stage in `entity/config_entity.py`

Whatsoever be the finding like data drift or any anomaly, it needs to be reported and for the same a file is being generated and it need to be saved in readable format. And for the same report is generated in 'yaml' file format. So, a python class will store the information of the data validation path and report file path is created:

Class: DataValidationConfig
Argument/Input: training_pipeline_config
Object:
- Data_validation_directory: (This will store the output in the given directory)
- Report_file_path: (This is path of generated report file)
- Threshold: (This will define the acceptable percentage of missing values in a column)
- Base_data_path: (This is the path of the base data provided by the client)

After input is defined, let's define the output. Obviously there will be only one output and that is the report.yaml file. And the is entity/artifact_entity.py. It will have a data validation class that will store the information of the report file path.

Class: DataValidationArtifact
Argument/Input: None
Object:
Report_file_path: (This is path of generated report file)

Input and output are defined now its time to work on the data validation component present in the `components/data_validation.py`. It will start with the necessary imports like logger, exception, os, sys, artifact_entity, config_entity and other modules required for the validation. A python class DataValidation is defined that will initialize the data validation class. In training pipeline some of the data is passed from the ingestion stage to validation stage. And hence the same need to be initialized while initializing the DataValidation class.

Class: DataValidation

Argument/Input:

- data_validation_config (Remember this is already defined in `entity/config_entity.py` file)
- data_ingestion_artifact (Remember this is already defined in `entity/artifact_entity.py` file)

Object:

- data_validation_config
- validation_error (This is of dictionary datatype and it will keep track of errors in validation.) (This is defined in this class)
- data_ingestion_artifact

Description: This imports the configuration (or input) structure to get started with the data validation stage.

Once initialized now one can start with validation. Let's start with checking the required column. So, a method is defined that will check if the required columns are present or not.

Method: is_required_column_exists() -> bool:

- Arguments:
  - base_dataframe:
  - current_dataframe:
  - report_key_name: (This to specify under what heading the report is prepared.)
- Return: Boolean true or false
- Description: Will check the columns in the new dataset with the columns in the original dataset. All the errors will be stored in validation_error dictionary. And since we must report every parameter so another parameter report_key_name is passed that will specify under what heading the report need to be generated. Method will return True if all columns are in place else will return False

Next, we will check if the column of the dataset has missing value more than that of a threshold.

Method: drop_missing_values_columns

- Argument:
  - data frame (pandas data frame)
  - report_key_name: (This to specify under what heading the report is prepared.)
- Return: Pandas Data frame if at least single column is available after dropping missing values columns else None.
- Description: This function will drop the column that will have missing value more than that of specified threshold (threshold define in `config_entity.py`). All the details will be stored in validation_error dictionary.

Checking for the data drift.

Method: data_drift

- Arguments/Input:
  - Base data frame
  - Panda's data frame
  - Report_key_name: (This to specify under what heading the report is prepared.)
- Returns: None
- Description: This function will compare the data drift between the new dataset and the base dataset. A drift report is prepared and made available.

Similarly other validation can be performed by creating suitable validation.

Now the report needs to generate and it is in yaml format. So, a function is written accordingly in utils.py file. It has already been described earlier in utils function.

Function: write_yaml_file

- Argument:
  - File_path: file path of yaml file
  - Data: This is of type dictionary that will store the required data
- Description: Create a report on yaml file.

Step 8: Data Transformation

This is the stage where EDA plays an important role. Let say it was observed that the dataset has outliers, or the classes are imbalance or missing values or any other anomaly. Such issues are dealt in this stage. Few steps include to deal with such anomaly are:

- Robust scaler that deals with the outliers
- Balance the classes (SMOTE)
- Compute the missing values

What techniques need to be applied (let for imputation or for dealing with the missing class) comes from the EDA.

If we observe carefully, our data is stored in the path `artifact/<folder_name>/data_ingestion/dataset`, as the path was defined by us in data ingestion stage. Now transformation will change the data and the values of each column will change. So, the transformed training and testing data need to be saved somewhere so a path for the same needs to be define. Also, the data transformation steps need to be saved as the same will be used in prediction pipeline stage. This stage will save/give three things:

- Transformed train data
- Transformed test data
- Data Transformation object

Defining the input for the data transformation stage. Obviously the same is define in the `config_entity.py` file. A python class named DataTransformationConfig is defined and the way DataValidation class was defined. Then input to the class DataTransformationConfig is an object of the TrainingPipelineConfig.

Class: DataTransforationConfig:

Arguments: TrainingPipelineConfig

Initialized Objects:

- Data_transformation_dir: (This will create a data transformation directory or folder that will contain various artifacts)
- Transform_object_path: (This will create the path of transformer object. Its pickle file that contains the transformation information)
- Transform_train_path: (Path to save the transform train file name) (file will be in numpy array and hence need to be saved in .npz format)

- Transform_test_path: (Path to save the transform test file name) (file will be in numpy array and hence need to be saved in .npz format)
- Target_encoder_path: (Path where encoder object will be saved)

By looking at the class it is obvious that there are three objects, data transformation object, transformed train and test object. So, they are the artifact of data transformation step. Defining in the same in `artifact_entity.py` file.

Class: DataTransformationArtifact

Arguments:

- transform_object_path
- transformed_train_object
- transformed_test_object

With this the output has been defined and lets now write the code in `components/data_transformation.py` file. It will have a class called DataTransfromation. Information to this stage is found in `config_entity.DataTransformationConfig` so the same will be passed as argument to the class DataTransformation. Further the train file path, test file path and feature store file path are required. Same can be found in `artifact_entity.DataIngestionArtifact`, thus passing the same as an argument. In short DataTransformation class will take two arguments, DataTransformationConfig and DataIngestionArtifact. The two argument need to be made avilable to entire class and hence two variable is defined that store the information of arguments.

Class: DataTransformation:

- Arguments: DataTransformationConfig, DataIngestionArtifact
- Variables:
  - data_transformation_config
  - data_ingestion_config

Now the transformation object need to be defined within the class. The method is get_data_transformer_object and it will take the arguments same as that of class DataTransformation. And this will return the Pipeline object from Sklearn.

Method: get_data_transformer_object(cls)

Arguments: Same as that of Data Transformation class

Return: pipeline

Description: This creates a transformation object

It needs to be observed that our target column is not yet defined and the same need to defined. It's better to define the same in config.py file. As it needs to be called at various instances. Since we need to change the categorical column to the numerical for the prediction purpose, a dictionary called target_column_mapping need to be defined. An alternative to this is one can define the label encoder at the data transformation config.

To encode any column a function can be written in the utils.py that can do the encoding. It can be done at previous stages, but that encoding object need to be saved and to avoid that the function can be defined at utils.py.

To save and load the model object a function is written in the utils.py.

Function: save_object

- Arguments: file_path, object to be saved
- Return: None
- Description: Save the object into the file path. Either dill or pickle library can be used.

Function: load_object

- Arguments: file_path
- Return: File object
- Description: Load the object from the given file path. Either dill or pickle library can be used.

In the process of the transformation, the data frame is converted into the numpy array. So, we need to load and save the same numpy array. For the same a function needs to be created.

Next method is defined that will initiate the data transformation method. Its name is initiate_data_transformation and it will return object as defined in the artifact_entity.DataTransformationArtifact. This method will take the train_file and test_file and do the necessary transformation.

Method: initate_data_transformation

- Arguments: Nill
- Return: DataTransformationArtifact
- Description: It will transform the train and test file. Also, the file will be spit in target and feature columns. Also, this method will include all the preprocessing that need to do before the training starts. Also, the steps like encoding of the target column need to be done at this

stage and the object of the same need to be saved. At this point the method get_data_transformer_object is also and a object of that pipeline is build. This object will transform the training feature and the test feature. If there is unbalanced class, then balance same using the SMOTE library. Since we have feature column and target column separated, so they need to be combined and then saved. Every object created at this stage need to be saved. Now the details of every object saved is then returned in artifact so that the same can be recovered when required. Then every information is logged.

With this data transformation stage is completed.

Step 9: Model Trainer

At this stage we will train, test and save the model. The performance metrices of the model will also be saved for evaluation purpose.

As always, the step begins with the config_entity.py file that has the information of to start the model training. A class for the same is build.

Class: ModelTrainerConfig

Argument: Training_pipeline_config

Variables initiatated:

- model_trainer_dir
- model_path
- expected_score (This will set the criteria, if model needs to be accecpted or rejected)
- overfitting score (Difference between the training and test score)

Output of this stage will be written in the artifact_entity. Output will have three output, model path, training and testing score.

Class: ModelTrainerArtifact

Variables initiated:

- model_path
- train_score
- test_score

With this, now we are ready to build the model_trainer component.

So, a class Model Trainer is build

Class: ModelTrainer

- Arguments:
  model_trainer_config:
  data_transformation_artifact:
- Variables initialized:

Also, a method is created to train the model.

Method: train_model

- Arguments: x, y data frame
- Return: model object
- Description: It will create a model object that will be train on the data set given.

Also, a method for fine tuning/hyper parameter tuning can be written.

Net set is to initiate the model trainer process. Its output/artifact will be like model trainer artifact defined in the artifact_entiry.py. To initiate the same, we need to import the training file.

Method: initiate_model_trainer

- Arguments: None
- Return: Model Trainer artifact
- Description: This method will load the data created in the data transformation stage using the load function created in the utils.py file. With this both train and test file are loaded. Next The dataset is split into feature and target column. Then the model is trained on the dataset. Then training and testing score is calculated. Also, the model needs to be checked for the overfitting and under fitting. And for the same, test score needs to be more than the minimum model expected score. If the score found to be less than the expected score, then program need to be terminated and then an exception needs to be raised and that will terminate the program. Next model to be checked for the overfitting. Again, if model found to be overfitted then program is terminated, and the exception is raised. Next process will include to save the model object. Finally, artifact is prepared. Every information is then logged.

Now it’s time to test the code. So, the same will be done through main.py file.

Step 10: Model Evaluation

There are two types of prediction

i. Instance Prediction - Prediction is done based on single data instance
ii. Batch prediction - Prediction is done on multiple data instance that is being provided in a single csv file.

In order to make the prediction the model and its supporting objects are saved into the server/cloud (say S3 bucket). During the training pipeline a new model and its supporting file is being developed. Now the task is to compare the performance of the two model, the one in production and other in training pipeline. This comparison happens in the Model Evaluation stage. If the new model is found to be working fine, then the same is pushed to the production in model pusher stage.

This stage checks the newly build model with the model that is in production stage. If the newly trained model gives the better score as compared to the existing/production model, the same will be replaced with the production model file. This this the way one can perform the continuous training. So, it is required to load the production model and make prediction. Now the prediction results from the newly trained model and the production model is then compared. For the same a file named predictor.py is created. Then a class named Predictor is created. Argument name is model registry as it points the location of the model that is being in the production stage.

Class: ModelResolver
Arguments: model_registry
Variable: model_registry

Method: get_latest_model_path_production

- Arguments: None
- Description: It will return the path of latest model in production.

Important thing to note here is that apart from the model there might be supporting files like tranformed.pkl or encoder.pkl or any other. So like the above method such method needs to be developed that will return the path of those object.

Similarly, the method needs to be developed that return the path of latest model developed and the supporting objects.

The above made class are made in the new prediction.py file. Now we can work on model evaluation component.

There might be two cases related to model in production server:

i. When there is no model in production server (When the model is being deployed for the first time)
ii. When there is a model present in the production server.

So, the class and method need to be designed in a way that whatever be the situation is appropriate model can be pushed to the production server. With this now we can work on the ModelEvaluation.py file. Necessary objects are imported and then a class along with methods are build that will fulfill the above objective.

Class: ModelEvaluation

- Arguments:
  model evaluation config
  data ingestion artifact:
  data transformation artifact:
  model trainer artifact:
- Variables Initiated:
  model_eval_config
  data_ingestion_artifact
  data_transformation_artifact
  model_trainer_artifact
  model_resolver

Method: initiate model evaluation

- Arguments: None:
- Return:
- Description: Comparison between the models in production and model developed in the training process. First the location of development model, transformer and target encoder is extracted and then the objects are loaded. Latest trained object is loaded. Now the comparison is made. If the problem statement is classification or regression, then f1 score or r2 score is calculated respectively. These scores are calculated on the test dataset. Every instance is then logged.

Now the best model need to be saved in model pusher stage.

Step 11: Model Pusher

Based on the result of the model evaluation the proper steps need to be taken. If the production model is preforming better no further step need to be taken but if the newly trained model is performing better, then the model is saved into the separate folder called as "Saved Model". And this folder is in sync with the cloud server like S3bucket.

As usual before building any component, its input and output need to be defined in the config entity and artifact entity folder.

To understand the input to model pusher component one need to know the component that will be generated in this stage and then keep track of the folder path where these components will be saved. In this stage there will be a folder called Model_Pusher inside the artifacts folder. Inside the folder there will be a folder called saved_model that will load the newly trained objects like models, transformer and or any other object. Path of each object need to be the input of the model pusher component. Another folder needs to be developed that will be in sync with the production server in cloud that will keep track of the best model folder.

class ModelPusherConfig:

- Argument: TrainingPipelineConfig
- Variables:
  model_pusher_dir
  saved_model_dir
  pusher_model_dir
  pusher_model_path

At the end of the model pusher stage two things will be returned one path of the folder where model is saved and the path of folder that is in sync with the production model.

class ModelPusherArtifact
Argument: pusher_model_dir
saved_model_dir

Once the input (or config) and output (or artifact) has been decided, model_pusher component can be built.

class ModelPusher

- Arguments: model_pusher_config
  data_transformation_artifact
  model_trainer_artifact
- Variables Initiated:
  model_pusher_config
  data_transformation_artifact
  model_trainer_artifact
  model_resolver

Method: initiate model pusher:

- Argument: None
- Return: Model Pusher Artifact
- Description: Save the latest model and supporting objects onto the saved model folder.

This completes the training pipeline. Finally, a function needs to be defined that initiates the training pipeline. This will be defined in the pipeline.py file. Training pipeline function should involve every data stage like data ingestion, data validation, data transformation, model training, model evaluation and model pusher. With the completion training stage prediction stage starts.

Step 12: Prediction

Let say for prediction a csv file is given. The same file is then loaded into the system and the prediction are made. Then the final file is then saved inside the prediction folder with proper naming convention.
To start the batch prediction, the prediction file path is given as an input. The required objects for the prediction need to load and prediction is made on the file. Then the file is saved.