New York Taxi Fare Amount

In this project we will address three tasks which are defined as below:
Question A:

- Calculate on the entire dataset the 5th, 50th and 95th percentiles (q05, q50, q95) on the dataset values: 'fare_amount ' , ' tip_amount ' and ' total_amount ' ; divided according to the ' VendorID ', ' passenger_count ' and ' payment_type ' fields
- The calculation output must be a dataFrame to be exported in CSV format organized with:
    - Columns: field name (on which the percentile is calculated) + “_p_” + percentile threshold
    - Rows (index): grouping field name + ” _” + value of the group on which the percentile calculation is performed

Question A.1 (optional):

- Calculate the percentiles as reported for question A also for the dataset divided by trip_distance if>2.8 or <=2.8 and add the calculated values to the dataFrame with the logic reported in question A

Question B:

- Generate an ML model for estimating the " total_amount " based on the variables (as input to the model): ' VendorID ','passenger_count ' ,'payment_type ' ,' trip_distance '
- It is possible to independently define the methodology and the selection and split process of the reference dataset for training, testing and verification of the model (kf, random, train -test- valid)
- (optional) For model optimization it is recommended to calculate the RMSE on the selected partial test dataset
- Export the generated model to file (ie via pickle, json …)
- The quality assessment of the generated model will be verified through the calculation of the RMSE ona test dataset equivalent to the one used by the user in terms of format and compatible in terms of number (but not provided)
- The user is given the right to use a different ML model from those present in the sklearn libraries, but the generated model must be exportable, and the user must indicate the name and version of the library used (for calculation of the RMSE on the new test dataset)

How to execute:

First of all you need to create an environment using conda as below:

```
conda env create -f environment.yml
conda activate taxi
python -m ipykernel install --user --name "taxi"
```

There are two approaches to execute tasks:
- Using Notebooks: there are some notebooks in [notebook](./notebook/) folder which has two sub-folder to solve each question.
- Modular programming: to execute using this approach you need to execute commands below in your terminal.

```
- pip install -e .
- taxi
```

QA & QA.1 notebook

First of all it worth to describe the available columns in the dataset including:
- `VendorID`: Identifier for the TPEP provider supplying the record.
    - 1 = Creative Mobile Technologies, LLC
    - 2 = VeriFone Inc.
- `Passenger_count`: The number of passengers in the vehicle, as entered by the driver.
- `Trip_distance`: The distance of the trip in miles, as recorded by the taximeter.
- `Payment_type`: How the passenger paid for the trip, represented by a numeric code.
    - 1 = Credit card
    - 2 = Cash
    - 3 = No charge
    - 4 = Dispute
    - 5 = Unknown
    - 6 = Voided trip
- `Fare_amount`: The fare as calculated by the meter based on time and distance.
- `Total_amount`: The total charge to passengers, excluding cash tips.

The user can read the dataset using the `read_dataset` method defined in the `Data` class. Subsequently, the data will be group by the features: `VendorID` , `passenger_count` and `payment_type` fields and then the percentiles for the columns: `fare_amount` , `tip_amount` and `total_amount` will be calculated and saved into[report.csv](../../artifacts/QA/report.csv). For the trip_distance, the user create a new column to check if the trip_distance related to the row is below or above 2.8 and then calculate the percentiles based on that column.

```
class Data:
   def __init__(self):
       self.config = CONFIG

   def read_dataset(self):
       """
       Extracts dataset from a zip file if not already extracted,
       loads it into a Pandas dataframe, and drops specified columns.
```

```
        """
        if not (
            os.path.exists(

f"{self.config.Data.DATA_DIR}/{self.config.Data.DATA_FILE_NAME}"
            )
        ):
            with zipfile.ZipFile(self.config.Data.DATA_DIR_ZIP, "r") as
zip_ref:
                zip_ref.extract(
                    self.config.Data.DATA_FILE_NAME,
self.config.Data.DATA_DIR
                )
                zip_ref.close()
        self.df = pd.read_csv(

f"{self.config.Data.DATA_DIR}/{self.config.Data.DATA_FILE_NAME}"
        ).drop(columns=PARAMS.DATASET.COLUMNS_TO_DROP)
        return self.df

    def calculate_percentiles_for_each_group(self):
        """
        Calculates percentiles for specified group columns and optionally
for trip_distance categories.
        """
        ## Question A
        results = pd.DataFrame()
        group_columns = ["VendorID", "passenger_count", "payment_type"]
        # Calculate percentiles for each group column
        for group_col in group_columns:
            percentile_result = (
                self.df.groupby(group_col)
                .apply(calculate_percentiles, include_groups=False)
                .reset_index()
            )
            percentile_result[group_col] =
percentile_result[group_col].apply(
                lambda x: f"{group_col}_{x}"
            )
            percentile_result.set_index(group_col, inplace=True)
```

```
            results = pd.concat([results, percentile_result])

        ####### Question A.1 (optional): Calculate percentiles for
trip_distance categories

        # Calculate percentiles for trip_distance > 2.8
        self.df["trip_distance_bucket"] = np.where(
            self.df["trip_distance"] <= 2.8, "trip_distance<=2.8",
"trip_distance>2.8"
        )

        percentile_over_2_8 = (
            self.df[self.df["trip_distance_bucket"] == "trip_distance>2.8"]
            .groupby(["trip_distance_bucket"])
            .apply(calculate_percentiles, include_groups=False)
            .reset_index()
        )
        percentile_over_2_8.set_index("trip_distance_bucket", inplace=True)
        percentile_under_eq_2_8 = (
            self.df[self.df["trip_distance_bucket"] ==
"trip_distance<=2.8"]
            .groupby(["trip_distance_bucket"])
            .apply(calculate_percentiles, include_groups=False)
            .reset_index()
        )
        percentile_under_eq_2_8.set_index("trip_distance_bucket",
inplace=True)
        percentile_results = pd.concat(
            [results, percentile_over_2_8, percentile_under_eq_2_8]
        )
        self.df = self.df[PARAMS.DATASET.COLUMNS_TO_USE]
        return percentile_results
    @staticmethod
    def save_csv(df):
        if not os.path.exists(f'{CONFIG.QA.PERCENTILE_DATAFRAME_PATH}'):
            os.makedirs(f'{CONFIG.QA.PERCENTILE_DATAFRAME_PATH}')

df.to_csv(f'{CONFIG.QA.PERCENTILE_DATAFRAME_PATH}/{CONFIG.QA.PERCENTILE_DA
TAFRAME_FILE}')
```

Then, the user will define a pipeline to execute the task as below.

```
# pipeline
data_obj = Data()
df = data_obj.read_dataset()
percentiles = data_obj.calculate_percentiles_for_each_group()
data_obj.save_csv(percentiles)
percentiles
```

QB-Preprocessing notebook

Objective:

Preprocessing of the data is one of the vital tasks of each machine learning task. This can help the training process to avoid over-fitting or under-fitting and leads to providing a high resolution machine learning model. In this notebook, the user can load the dataset from the [`ZIP`](**artifacts/data/yellow_tripdata_2019-04.csv.zip**) file and then load the data. Then do some exploratory data analysis to analyze the dataset to summarize its main characteristics. A very important step is separating the data into train and test sets before feature engineering. This will lead to avoid data leakage during the training and evaluation process. Finally, a set of proper feature engineering techniques will be implemented to make the data ready for the training process.

In the first step, the user will leverage the `**Data**` class to preprocess the data for the next steps (Train and Evaluation of ML model). There are some methods are defined in the `Data` class which are useful for preprocessing of the data including:

- `read_dataset`: This method extracts dataset from a zip file if not already extracted, loads it into a Pandas dataframe, and drops specified columns.
- `handle_outliers_tukey`: This method is responsible for handling outliers which leads to high quality training process. The user will be use **the Tukey IQR method** which is a rule says that the outliers are values more than 1.5 times the interquartile range from the quartiles either below `Q1 − 1.5 IQR`, or above `Q3 + 1.5IQR`. Thus the user can simply calculate outliers per column feature by taking the necessary percentiles.
- `eda`: This method can help the user to acquire some statistical information about the dataset and the correlation between each two columns. Indeed the output of this method is two pictures which are shows the correlation matrix of each two features and a scatter plot to know if there is a linear or nonlinear relation between the label column and
- `trip_distance` column which is a numerical feature. Then the user can decide whether to apply the bucketization technique or simply rescale that numerical feature.
- `data_split`: This method will split the dataset into train and test with the portion of 80 and 20 respectively.

- `preprocessing`: This method can help the user to apply some preprocessing techniques on train and test sets. Based on the results from the `eda` function, the user decides to use a `bucketization` approach for the numerical column and a `MinMaxScaler` for the label column.

```
class Data:
   def __init__(self):
       self.config = CONFIG

   def read_dataset(self):
       """
       Extracts dataset from a zip file if not already extracted,
       loads it into a Pandas dataframe, and drops specified columns.
       """
       if not (
           os.path.exists(
f"{self.config.Data.DATA_DIR}/{self.config.Data.DATA_FILE_NAME}"
           )
       ):
           with zipfile.ZipFile(self.config.Data.DATA_DIR_ZIP, "r") as
zip_ref:
               zip_ref.extract(
                   self.config.Data.DATA_FILE_NAME,
self.config.Data.DATA_DIR
               )
               zip_ref.close()
       self.df = pd.read_csv(
f"{self.config.Data.DATA_DIR}/{self.config.Data.DATA_FILE_NAME}"
       ).drop(columns=PARAMS.DATASET.COLUMNS_TO_DROP)
       return self.df

   def calculate_percentiles_for_each_group(self):
       """
       Calculates percentiles for specified group columns and optionally
for trip_distance categories.
       """
       ## Question A
       results = pd.DataFrame()
       group_columns = ["VendorID", "passenger_count", "payment_type"]
```

```
       # Calculate percentiles for each group column
       for group_col in group_columns:
           percentile_result = (
               self.df.groupby(group_col)
               .apply(calculate_percentiles, include_groups=False)
               .reset_index()
           )
           percentile_result[group_col] =
percentile_result[group_col].apply(
               lambda x: f"{group_col}_{x}"
           )
           percentile_result.set_index(group_col, inplace=True)
           results = pd.concat([results, percentile_result])

       ####### Question A.1 (optional): Calculate percentiles for
trip_distance categories

       # Calculate percentiles for trip_distance > 2.8
       self.df["trip_distance_bucket"] = np.where(
           self.df["trip_distance"] <= 2.8, "trip_distance<=2.8",
"trip_distance>2.8"
       )

       percentile_over_2_8 = (
           self.df[self.df["trip_distance_bucket"] == "trip_distance>2.8"]
           .groupby(["trip_distance_bucket"])
           .apply(calculate_percentiles, include_groups=False)
           .reset_index()
       )
       percentile_over_2_8.set_index("trip_distance_bucket", inplace=True)
       percentile_under_eq_2_8 = (
           self.df[self.df["trip_distance_bucket"] ==
"trip_distance<=2.8"]
           .groupby(["trip_distance_bucket"])
           .apply(calculate_percentiles, include_groups=False)
           .reset_index()
       )
       percentile_under_eq_2_8.set_index("trip_distance_bucket",
inplace=True)
       percentile_results = pd.concat(
```

```
            [results, percentile_over_2_8, percentile_under_eq_2_8]
        )
        self.df = self.df[PARAMS.DATASET.COLUMNS_TO_USE]
        return percentile_results

    @staticmethod
    def handle_outliers_tukey(df, columns):
        outlier_indices = []
        for col in columns:
            q1 = np.percentile(df[col], 25)  # First quartile (25th
percentile)
            q3 = np.percentile(df[col], 75)  # Third quartile (75th
percentile)
            iqr = q3 - q1  # Interquartile range
            # Calculate bounds for outliers
            lower_bound = q1 - 1.5 * iqr
            upper_bound = q3 + 1.5 * iqr

            # Identify outliers and replace them
            outlier_mask = (df[col] < lower_bound) | (df[col] >
upper_bound)
            outlier_indices.extend(df.index[outlier_mask])
            df.loc[outlier_mask, col] = np.clip(
                df.loc[outlier_mask, col], lower_bound, upper_bound
            )
            return df

    def eda(self):
        self.df = self.handle_outliers_tukey(
            self.df, columns=["trip_distance", "total_amount"]
        )
        # Calculate correlation coefficient
        correlation_coefficient =
self.df["trip_distance"].corr(self.df["total_amount"])

        # Create figure and axes
        fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 6))

        # Plot correlation matrix
        correlation_matrix = self.df.corr()
```

```
        mask = np.triu(np.ones_like(correlation_matrix, dtype=bool))
        cmap = sns.diverging_palette(220, 20, as_cmap=True)
        sns.heatmap(
            correlation_matrix,
            mask=mask,
            cmap=cmap,
            vmax=1,
            center=0,
            square=True,
            linewidths=0.5,
            cbar_kws={"shrink": 0.5},
            annot=True,
            ax=ax1,
        )
        ax1.set_title("Correlation Matrix")

        # Plot scatter plot to know if there is any lineaity between
trip_distance and total_amount
        ax2.scatter(self.df["trip_distance"], self.df["total_amount"],
alpha=0.5)
        ax2.set_title(
            f"Scatter Plot\nTrip Distance vs Total Amount\nCorrelation:
{correlation_coefficient:.2f}"
        )
        ax2.set_xlabel("Trip Distance")
        ax2.set_ylabel("Total Amount")

        # Adjust layout
        plt.tight_layout()
        # Show plot
        plt.show()

    def data_split(self):
        features = self.df[PARAMS.DATASET.FEATURES]
        target = self.df[PARAMS.DATASET.TARGET]
        self.X_train, self.X_test, self.y_train, self.y_test =
train_test_split(
            features,
            target,
```

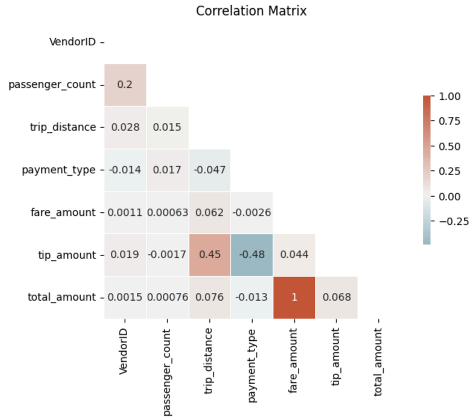
```
            test_size=PARAMS.DATASET.TEST_SIZE,
            random_state=PARAMS.DATASET.RANDOM_STATE,
        )

    @staticmethod
    def preprocessing(features, labels):
        # Handling Numericals
        # trip_distance Bucketization AND encoding
        for key in PARAMS.DATASET.NUMERICAL_FEATURES:
            # Create equal-width buckets for the numerical feature
            features["distance_bucket_equal_width"] = pd.qcut(
                features[key], q=10, duplicates="drop"
            )

            # One-hot encode the bucketized feature
            encoder = OneHotEncoder(sparse_output=False, drop="first")
            distance_buckets_encoded = encoder.fit_transform(
                features[["distance_bucket_equal_width"]]
            )

            # Create a DataFrame from the encoded features
            distance_buckets_encoded_df = pd.DataFrame(
                distance_buckets_encoded,

columns=encoder.get_feature_names_out(["distance_bucket_equal_width"]),
            )

            # Rename the bucket columns
            new_column_names = {
                col: f"distance_bucket_{i+1}"
                for i, col in
enumerate(distance_buckets_encoded_df.columns)
            }
            distance_buckets_encoded_df.rename(columns=new_column_names,
inplace=True)

            # Concatenate the encoded features with the original DataFrame
(excluding the original bucket column)
            df_encoded = pd.concat(
                [
```

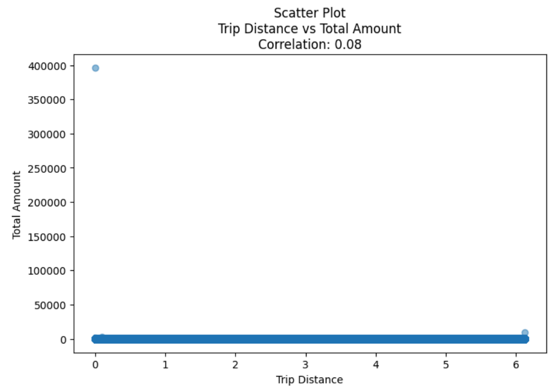
```
                   features.drop(
                       columns=["distance_bucket_equal_width", key]
                   ).reset_index(drop=True),
                   distance_buckets_encoded_df.reset_index(drop=True),
               ],
               axis=1,
           )

       # Rescale the label
       # Initialize the RobustScaler
       scaler = MinMaxScaler(feature_range=(-1, 1))
       labels = labels.reshape(-1, 1)

       # Fit and transform the labels
       scaled_labels = scaler.fit_transform(labels)

       # Convert scaled labels back to a Series
       labels = pd.Series(scaled_labels.flatten())

       return df_encoded, labels
```

Then the user implements a pipeline for the preprocessing step as below:

```
data = Data()
data.read_dataset()
df = data.df[PARAMS.DATASET.COLUMNS_TO_USE]
data.eda()
data.data_split()
```

Here are the results of the eda function which depicts the correlation matrix and the scater plot between **traip_distanc** and **Total_amount** column:





Then the user apply the preprocessing function for the data preprocessing pipeline.

```
X_train, y_train = data.preprocessing(data.X_train, data.y_train.values)
X_test, y_test = data.preprocessing(data.X_test, data.y_test.values)
```

QB- Train and Evaluation notebook

Objective:
In this notebook the user implements a workflow for training and evaluating the model based on the preprocessed dataset. The user will track and monitor the processes using the `mlflow` tool during training and evaluating the model. The task is a regression problem which can be trained on different ML model from `scikit-learn` library including:

- `LinearRegression`
- `RandomForestRegressor`
- `GradientBoostingRegressor`

During the training, the user leverages the k-fold cross validation technique to avoid possible overfitting and stabilize the estimator. These estimators will be evaluated using the root mean squared error metric as a candidate metric for a regression task. Finally, all the processes will be tracked by `mlflow`.

For the first step, The user define a `Model` class to implement a sort of function with the below discriptions:

- **`is_mlflow_server_running`**: A method to check if mlflow server is running or not.
- **`start_mlflow_server`**: A method to start the mlflow server if it is not started yet.
- **`linear_regression_model`**: A method to initiate a `Linear Regression` model.
- **`random_forest_regression_model`**: A method to initiate a `RandomForestRegressor` model.
- **`gradient_boost_regression`**: A method to initiate a `GradientBoostingRegressor` model.

- `**training**`: A method to train some estimators using k-fold cross validation technique.
- `**evaluation**`: A method to evaluate a set of estimator and calculate `RMSE`(root mean squared error). Subsequently, get the best model based on this evaluation metric.
- `**save_and_log_model**`: A method to log input parameters, metrics, and model using `**mlflow**`. Additionally save the best estimators in to [arifacts/models]

```
class Model:
   def __init__(self, config, params, x__train, y__train, x__test,
y__test) -> None:
       self.config = config
       self.params = params
       self.X_train = x__train
       self.y_train = y__train
       self.X_test = x__test
       self.y_test = y__test

   @staticmethod
   def is_mlflow_server_running():
       """Check if the MLflow server is running by sending a request to
the tracking URI."""
       url = "http://127.0.0.1:5000"
       try:
           response = requests.get(url)
           return response.status_code == 200
       except requests.ConnectionError:
           return False

   def start_mlflow_server(self):
       """Start the MLflow server if it is not already running."""
       if not self.is_mlflow_server_running():
           print("MLflow server is not running. Starting the server...")
           process = subprocess.Popen(
               [
                   "mlflow",
                   "server",
                   "--backend-store-uri",
                   "sqlite:///mlflow.db",
                   "--default-artifact-root",
                   "./mlruns",
                   "--host",
```

```
                "127.0.0.1",
                "--port",
                "5000",
            ]
        )
        logger.info("Waiting for the MLflow server to start...")
        time.sleep(5)  # Give the server some time to start
        if self.is_mlflow_server_running():
            print("MLflow server started successfully.")
        else:
            print("Failed to start the MLflow server.")
    else:
        print("MLflow server is already running.")

def linear_regression_model(self):
    params = self.params.Models.LinearRegressionModel.HYPERPARAMETERS
    self.model = skLR(**params)

def random_forest_regression_model(self):
    params = self.params.Models.RandomForestModel.HYPERPARAMETERS
    self.model = skRFR(**params)

def gradient_boost_regression(self):
    params = self.params.Models.XGBoostModel.HYPERPARAMETERS
    self.model = skXGR(**params)

def training(self, folds=3):
    kf = KFold(folds)
    kf.get_n_splits(self.X_train)
    score = 0.0
    models = []
    for trainIdx, validIdx in kf.split(self.X_train):
        X_train_valid, X_valid = (
            self.X_train.iloc[trainIdx],
            self.X_train.iloc[validIdx],
        )
        y_train_valid, y_test_valid = self.y_train[trainIdx],
self.y_train[validIdx]
        self.model.fit(X_train_valid, y_train_valid)
        score = self.model.score(X_valid, y_test_valid)
```

```
            print("score = ", score)
            models.append(self.model)
        return models

    def evaluation(self, estimators):
        estimator_idx = 0
        self.best_estimator_rmse = float("inf")
        for estimator in estimators:
            estimator_idx = estimator_idx + 1
            y_test_pred = estimator.predict(self.X_test)
            rmse = root_mean_squared_error(self.y_test, y_test_pred)

            if rmse < self.best_estimator_rmse:
                self.best_estimator_rmse = rmse
                self.best_model = estimator
            logger.info(
                f"\nestimator_idx: {estimator_idx}, current_estimator_rmse:
{rmse},best_estimator_rmse: {self.best_estimator_rmse}"
            )

        print(f"Best RMSE: {self.best_estimator_rmse:.4f}")

    def save_and_log_model(self, model_path, model_file, model_name):
        mlflow.sklearn.autolog(
            log_input_examples=False,
            log_model_signatures=True,
            log_models=True,
            log_datasets=True,
            log_post_training_metrics=True,
            serialization_format="cloudpickle",
            registered_model_name=f"{model_name}",
            pos_label=None,
            extra_tags=None,
        )
        mlflow.log_metric("RMSE", self.best_estimator_rmse)

        # Log the model
        mlflow.sklearn.log_model(self.best_model, f"{model_name}")
        # Save model also in a seprate file
        if not os.path.exists(f"{model_path}"):
```

```
            os.makedirs(f"{model_path}", exist_ok=True)
            with open(model_path + "/" + model_file, "wb") as file:
                pickle.dump(self.model, file)
```

Then, the user defines a pipeline to train and evaluate different models including:

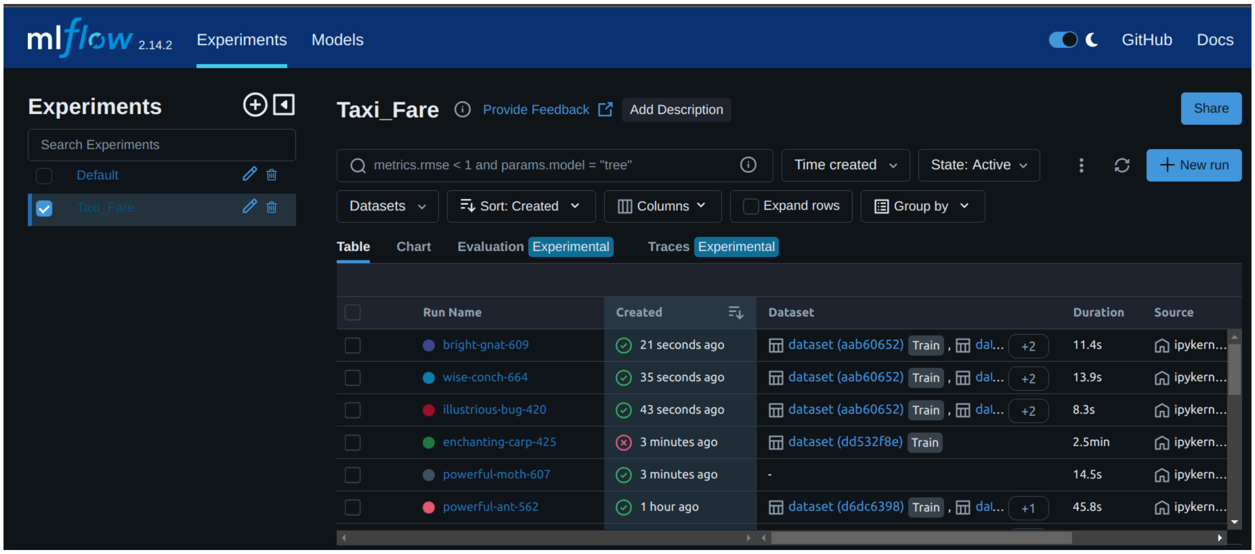
- `LinearRegression`
- `RandomForestRegressor`
- `GradientBoostingRegressor`

the hyperparameters of the models can be defined via [`params.yaml`]

```
model_obj = Model(
    config=CONFIG,
    params=PARAMS,
    x__train=X_train,
    y__train=y_train,
    x__test=X_test,
    y__test=y_test,
)
for model_name in PARAMS.Models:
    model_obj.start_mlflow_server()
    mlflow.set_tracking_uri("http://127.0.0.1:5000")
    tracking_url_type_store = urlparse("http://127.0.0.1:5000").scheme
    mlflow.set_experiment("Taxi_Fare")
    with mlflow.start_run() as run:
        # print(mlflow.active_run().info,'\n', mlflow.run.info)

        if model_name == "LinearRegressionModel":
            model_obj.linear_regression_model()
        if model_name == "RandomForestModel":
            model_obj.random_forest_regression_model()
        if model_name == "XGBoostModel":
            model_obj.gradient_boost_regression()

        estimators = model_obj.training()
        model_obj.evaluation(estimators)

        model_obj.save_and_log_model(
            model_path=f"{CONFIG.Model.MODEL_PATH}/{model_name}/",
            model_file=f"{CONFIG.Model.MODEL_FILE}",
            model_name=model_name,
```

```
    )
    mlflow.end_run()
```

The user chooses the best model based on `RMSE` from the mlflow server and saves the candidate model in [**artifacts/models/Best_model/best_model.pkl**]. The figure below depicts the `mlflow ui` and active runs on in the `Taxi_Fare` experiment.



The related codes for this step is reported as below:

```
active_runs = (
    mlflow.search_runs(
        experiment_names=["Taxi_Fare"],
        # Select the best one with highest f1_score and test accuracy
        filter_string="metrics.RMSE < 1 ",
        search_all_experiments=True,
    )
    .sort_values(
        by=["metrics.RMSE"],
        ascending=False,
    )
    .reset_index()
    .loc[0]
)



artifact_path =
json.loads(active_runs["tags.mlflow.log-model.history"])[0]["artifact_path"]
best_model_path = active_runs.artifact_uri + f"/{artifact_path}"
```

```python
# Load the model as an MLflow PyFunc model
mlflow_model = mlflow.pyfunc.load_model(model_uri=best_model_path)

# Define the path to save the model
pickle_model_path = "./artifacts/models/Best_Model/"
if not os.path.exists(f"{pickle_model_path}"):
   os.makedirs(f"{pickle_model_path}", exist_ok=True)
   with open(pickle_model_path+"best_model.pkl", "wb") as file:
       pickle.dump(mlflow_model, file)
# Save the model
print(f"Model saved to {pickle_model_path}")
```