**Summary of the project**

The “Team .csv “dataset comprises various attributes pertaining to different teams. These attributes may include (team, age, country, year, event, height, weight, medals, Prev medals) and contains **2144 rows and 11 columns**. Leveraging machine learning algorithms on this dataset can unveil valuable insights and facilitate informed decision-making processes.

To begin the analysis, exploratory data analysis **(EDA)** techniques can be employed to understand the structure of the dataset. This involves tasks such as data cleaning, handling missing values, and visualizing distributions of features.

Here’s a step by step details following below :-

* **Importing libraries**
* **pandas:**.It's particularly useful for data manipulation, analysis, and cleaning tasks.
* **numpy:** Offers support for large, multi-dimensional arrays and matrices.
* **Matplotlib**: It's highly customizable and suitable for creating static, interactive, and publication-quality plots.
* **Seaborn**: It simplifies the process of creating complex visualizations such as heatmaps, pair plots, and categorical plots.
* **Data Cleaning Process**

Data cleaning involves identifying and correcting errors, inconsistencies, and missing values in a dataset to improve its quality and reliability for analysis. As in the dataset alots of missing value present so first we have to check how many are present (missing values) in our data set.

**Source Code:-**  teams[teams.isnull().any(axis=1)]

This code filters rows in the 'teams' data frame containing any null values across columns, highlighting incomplete or missing data entries.

**Output:-**

| **team** | **country** | **year** | **athletes** | **age** | **prev\_medals** | **medals** |
| --- | --- | --- | --- | --- | --- | --- |
| **19** | ALB | Albania | 1992 | 9 | 25.3 | NaN | 0 |
| **26** | ALG | Algeria | 1964 | 7 | 26.0 | NaN | 0 |
| **39** | AND | Andorra | 1976 | 3 | 28.3 | NaN | 0 |
| **50** | ANG | Angola | 1980 | 17 | 17.4 | NaN | 0 |
| **59** | ANT | Antigua and Barbuda | 1976 | 17 | 23.2 | NaN | 0 |
| **...** | ... | ... | ... | ... | ... | ... | ... |
| **2092** | VIN | Saint Vincent and the Grenadines | 1988 | 6 | 20.5 | NaN | 0 |
| **2103** | YAR | North Yemen | 1984 | 3 | 27.7 | NaN | 0 |
| **2105** | YEM | Yemen | 1992 | 8 | 19.6 | NaN | 0 |
| **2112** | YMD | South Yemen | 1988 | 5 | 23.6 | NaN | 0 |
| **2120** | ZAM | Zambia | 1964 | 15 | 21.7 | NaN | 0 |

130 rows × 7 columns

Source Code – teams = teams.dropna()

The code snippet drops any rows with missing values in the DataFrame called 'teams'. This ensures the data used for analysis or processing is complete, improving accuracy and reliability, typically used in data preprocessing pipelines.

Source Code - teams.shape

The code snippet retrieves the shape of the DataFrame 'teams', which returns a tuple representing the number of rows and columns in the DataFrame. This provides an overview of the dataset's size and structure.

Source Code - train = teams[teams["year"] < 2012].copy()

test = teams[teams["year"] >=2012].copy()

The code creates two separate DataFrame copies: 'train' and 'test'. 'Train' contains data where the "year" column values are less than 2012, while 'test' contains data where the "year" column values are greater than or equal to 2012. This typically splits data into training and testing sets for model development and evaluation in machine learning tasks.

Source Code - test.shape

The code retrieves the shape of the DataFrame 'train', indicating the number of rows and columns. This helps in assessing the size and structure of the training dataset after filtering based on the condition of the "year" column being less than 2012.

Source Code – train.shape

The summary of `train.shape` indicates the dimensions of the DataFrame 'train', typically represented as `(rows, columns)`. This information provides an overview of the size and structure of the training dataset after filtering based on the condition of the "year" column being less than 2012.

Source Code – reg = LinearRegression()

The code initializes a linear regression model object named 'reg'. This is typically used for modeling linear relationships between variables. It sets up an instance of a linear regression model, allowing for further configuration, training, and prediction tasks in machine learning applications.

Source Code - predictors = ["athletes", "prev\_medals"]

targetr = "medals"

The code snippet suggests defining a list of predictor variables, which include "athletes" and "prev\_medals", and assigns the target variable "medals". This setup is commonly used in predictive modeling tasks, where "predictors" are the features used to predict the "target" variable.

Source Code - reg.fit(train[predictors], train["medals"])

The code `reg.fit(train[predictors], train["medals"])` executes the fitting process of the linear regression model (`reg`) on the training data (`train`) using the specified predictor variables (`train[predictors]`) to predict the target variable (`train["medals"]`). This process involves estimating the coefficients of the linear regression model to best fit the training data. There isn't a "summary" function built into this specific method call, but you can analyze the fitted model's attributes such as coefficients, intercept, and performance metrics separately to understand the model's performance and characteristics.

Source Code - predictions=reg.predict(test[predictors])

The code `predictions = reg.predict(test[predictors])` generates predictions using the linear regression model (`reg`) on the test data (`test`) based on the specified predictor variables (`test[predictors]`). However, there isn't a built-in "summary" function for this particular method call. After generating predictions, you can evaluate the model's performance using various metrics such as mean squared error, R-squared, or visual inspection of predicted versus actual values.

Source Code - predictions.shape

The summary of `predictions.shape` would provide the dimensions of the array containing the predicted values. This typically appears as `(n\_samples,)`, where `n\_samples` represents the number of predictions generated. This information gives an overview of the size of the prediction array.

Source Code - test["predictions"] = predictions

The code `test["predictions"] = predictions` assigns the predicted values generated by the linear regression model to a new column named "predictions" in the DataFrame 'test'. This allows for easy comparison between the actual values and the predicted values during model evaluation and analysis.

Source Code - test.loc[test["predictions"] < 0, "predictions"] = 0

The code snippet filters the "predictions" column in the 'test' DataFrame, replacing any values less than 0 with 0. This adjustment is commonly applied in scenarios where negative predictions are not feasible or meaningful, ensuring predictions remain within a valid range.

Source Code - test["predictions"] = test["predictions"].round()

The code rounds the values in the "predictions" column of the 'test' DataFrame to the nearest integer using the `.round()` method. This operation is often performed to convert continuous predicted values into discrete values, which can be useful for certain types of analyses or interpretations.

Source Code - teams.describe()["medals"]

The code `teams.describe()["medals"]` generates a summary of descriptive statistics for the column "medals" in the DataFrame 'teams'. This summary typically includes count, mean, standard deviation, minimum, maximum, and quartile information, providing insights into the distribution and characteristics of the "medals" data.

Source Code - from sklearn.metrics import mean\_absolute\_error

error = mean\_absolute\_error(test["medals"], test["predictions"])

The code snippet calculates the mean absolute error (MAE) between the actual "medals" values and the predicted values stored in the "predictions" column of the 'test' DataFrame. This metric quantifies the average magnitude of errors between predicted and actual values, providing a measure of model performance in terms of prediction accuracy. The "summary" output typically displays the computed MAE value.

Source Code - test["predictions"] = predictions

The code `test["predictions"] = predictions` assigns the predicted values generated by the linear regression model to a new column named "predictions" in the DataFrame 'test'. This allows for easy comparison between the actual values and the predicted values during model evaluation and analysis. However, the term "summary" is generally not used here. It's a process of assigning predicted values to a DataFrame column.

Source Code - test[test["team"] == "USA"]

The code `test[test["team"] == "USA"]` filters the 'test' DataFrame to only include rows where the "team" column value is "USA". This operation selects data specific to the USA team for further analysis or examination. However, the term "summary" is not typically used in this context. It's a filtering operation to extract specific rows based on a condition.

Source Code - errors = (test["medals"] - predictions).abs()

The code `errors = (test["medals"] - predictions).abs()` computes the absolute errors between the actual "medals" values and the predicted values, storing the result in a variable called "errors". This operation calculates the absolute difference between each predicted value and its corresponding actual value. However, the term "summary" is not typically used in this context. It's a computation of absolute errors for further analysis or evaluation of the model's performance.

Source Code - error\_by\_team = errors.groupby(test["team"]).mean()

medals\_by\_team = test["medals"].groupby(test["team"]).mean()

error\_ratio = error\_by\_team / medals\_by\_team

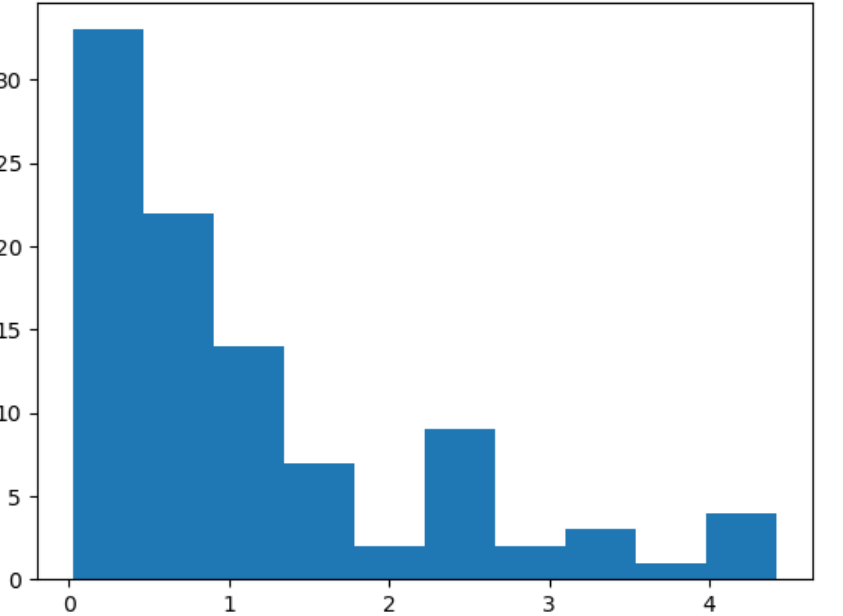
The code calculates error ratios for each team by dividing the mean absolute error (`error\_by\_team`) by the mean number of medals (`medals\_by\_team`). This provides a measure of prediction accuracy relative to the average number of medals per team. However, there isn't a built-in "summary" function for this operation. It's typically a computation for further analysis or evaluation of the model's performance, providing insights into how well the model predicts for different teams relative to their average performance.

Source Code - import numpy as np

error\_ratio = error\_ratio[np.isfinite(error\_ratio)]

The code `error\_ratio = error\_ratio[np.isfinite(error\_ratio)]` filters out any NaN (Not a Number) values from the `error\_ratio` array using NumPy's `np.isfinite()` function. This operation ensures that only finite (non-NaN) values are retained in the `error\_ratio` array. However, the term "summary" is not typically used in this context. It's a filtering operation to remove NaN values for further analysis or computation.

Source Code - error\_ratio.plot.hist()

The code `error\_ratio.plot.hist()` creates a histogram plot of the error ratio data. Histograms are graphical representations of the distribution of numerical data, showing the frequency of values within predefined bins. However, the term "summary" is not applicable here as it's a plotting command to visualize the distribution of error ratios.

Source Code - error\_ratio.sort\_values()

The code `error\_ratio.sort\_values()` sorts the error ratios in ascending order. This operation arranges the error ratios from smallest to largest, facilitating easier analysis of which teams have relatively better or worse prediction accuracies compared to their average medal counts. However, the term "summary" is not typically used in this context. It's a sorting operation to organize the data for further examination.ss