## **Documentation**

In the comprehensive data preprocessing and exploratory data analysis (EDA) workflow for the **athlete\_events.csv** dataset. Here is the brief workflow of the method.

1. **Data Loading and Inspection**
   * Loading the dataset using pandas
   * Checking missing values, data types, and overall structure
2. **Data Cleaning and Transformation**
   * Handling missing values (median for numerical, most frequent for categorical)
   * Encoding categorical variables (LabelEncoder, OneHotEncoder)
   * Feature scaling (StandardScaler, MinMaxScaler, RobustScaler)
3. **Exploratory Data Analysis (EDA)**
   * Summary statistics
   * Outlier detection and removal (IQR method)
   * Visualizations (histograms, scatter plots, correlation heatmaps)
4. **Feature Selection & Model Training**
   * Encoding the target variable (Medal)
   * Correlation analysis with Medal
   * Training a RandomForestClassifier for feature importance
5. **Handling Class Imbalance**
   * Checking class distribution
   * Using SMOTE (Synthetic Minority Over-sampling Technique) to balance classes
6. **Train-Test Splitting & Model Evaluation**
   * Splitting data (train\_test\_split)
   * Training logistic regression and random forest models
   * Evaluating performance (accuracy, precision, recall, F1-score)

## 

## **Detailed Workflow**

## **Data Collection**

### **Importing Required Libraries**

import pandas as pd

import numpy as np

import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.impute import SimpleImputer

from sklearn.preprocessing import OneHotEncoder, LabelEncoder, StandardScaler, MinMaxScaler, RobustScaler

* pandas (pd): Used for data manipulation and analysis.
* numpy (np): Provides support for arrays and mathematical operations.
* seaborn (sns): A visualization library built on top of Matplotlib.
* matplotlib.pyplot (plt): Used for plotting graphs and visualizations.
* sklearn.impute.SimpleImputer: Helps fill in missing values in a dataset.
* sklearn.preprocessing:
  + OneHotEncoder: Converts categorical variables into a format that can be provided to ML algorithms.
  + LabelEncoder: Encodes categorical labels with numerical values.
  + StandardScaler, MinMaxScaler, RobustScaler: Various scaling techniques for normalizing numerical data.

### **Loading the Dataset**

df = pd.read\_csv("athlete\_events.csv")

* Reads the CSV file into a Pandas DataFrame.

### **Displaying Data Preview**

print("Dataset Preview:")

print(df.head())

* Prints the first five rows of the dataset to inspect its structure.

## **Checking Data Structure and Types**

print("Dataset Information:")

print(df.info())

* Displays the number of non-null values and data types of each column.

## 

## **Data Cleaning and Transformation**

### **Checking for Missing Values**

print("Missing Values Count:")

print(df.isnull().sum())

* Checks for missing values in each column.

### **Handling Missing Values**

missing\_values = df.isnull().sum()

missing\_values = missing\_values[missing\_values > 0]

print("Missing Values per Column:\n", missing\_values)

* Identifies and prints only the columns that contain missing values.

### **Imputation of Missing Values**

#### **Handling Numerical Columns**

num\_cols = df.select\_dtypes(include=['float64', 'int64']).columns

imputer\_num = SimpleImputer(strategy="median")

df[num\_cols] = imputer\_num.fit\_transform(df[num\_cols])

* Selects numerical columns.
* Uses SimpleImputer with the "median" strategy to fill missing numerical values.

#### **Handling Categorical Columns**

cat\_cols = df.select\_dtypes(include=['object']).columns

imputer\_cat = SimpleImputer(strategy="most\_frequent")

df[cat\_cols] = imputer\_cat.fit\_transform(df[cat\_cols])

* Selects categorical columns.
* Uses SimpleImputer with the "most\_frequent" strategy to fill missing values with the most common value.

### **Verifying Missing Value Handling**

print("Missing Values after imputation:\n", df.isnull().sum())

* Confirms that missing values have been successfully handled.

## 

## 

## **Exploratory Data Analysis (EDA)**

### **Summary Statistics**

print("Summary Statistics:")

print(df.describe())

* Prints statistical summaries, including mean, standard deviation, min, max, and quartiles.

## **Outlier Detection and Removal**

### **Boxplot for Outlier Detection**

plt.figure(figsize=[12,6])

sns.boxplot(data=df[num\_cols])

plt.title("Boxplot for Outlier Detection")

plt.xticks(rotation=90)

plt.show()

* Creates a boxplot to visualize potential outliers in numerical columns.
* plt.figure(figsize=[12,6]): Defines the plot size.
* sns.boxplot(data=df[num\_cols]): Generates the boxplot for numerical columns.
* plt.xticks(rotation=90): Rotates x-axis labels for better readability.
* plt.show(): Displays the plot.

### **Outlier Removal Using IQR**

Q1 = df[num\_cols].quantile(0.25)

Q3 = df[num\_cols].quantile(0.75)

IQR = Q3 - Q1

* Computes the first quartile (Q1), third quartile (Q3), and interquartile range (IQR).

lower\_bound = Q1 - 1.5 \* IQR

upper\_bound = Q3 + 1.5 \* IQR

* Defines the lower and upper bounds for detecting outliers.

df\_no\_outliers = df[~((df[num\_cols] < lower\_bound) | (df[num\_cols] > upper\_bound)).any(axis=1)]

* Filters out rows where any numerical column has values outside the defined bounds.

print("Shape before outlier removal:", df.shape)

print("Shape after outlier removal:", df\_no\_outliers.shape)

* Displays dataset shape before and after removing outliers

## **Distribution Visualization**

import matplotlib.pyplot as plt

import seaborn as sns

sns.set(style="whitegrid")

* Sets the Seaborn style to improve visualization aesthetics.

### **Histograms for Age, Height, and Weight**

fig, axes = plt.subplots(1, 3, figsize=(18, 5))

# Age distribution

sns.histplot(df["Age"].dropna(), bins=30, kde=True, ax=axes[0], color="blue")

axes[0].set\_title("Age Distribution")

# Height distribution

sns.histplot(df["Height"].dropna(), bins=30, kde=True, ax=axes[1], color="green")

axes[1].set\_title("Height Distribution")

# Weight distribution

sns.histplot(df["Weight"].dropna(), bins=30, kde=True, ax=axes[2], color="red")

axes[2].set\_title("Weight Distribution")

plt.show()

1. fig, axes = plt.subplots(1, 3, figsize=(18, 5)): Creates a 1-row, 3-column figure layout.
2. sns.histplot(): Generates histograms with Kernel Density Estimation (kde=True).
3. .dropna(): Removes missing values before plotting.
4. color: Defines colors for each histogram.

## **Scatter Plots**

### **Scatter Plot for Age vs. Weight**

plt.figure(figsize=[8,5])

plt.scatter(df["Age"], df["Weight"], alpha=0.5)

plt.title("Age vs. Weight")

plt.xlabel("Age")

plt.ylabel("Weight")

plt.show()

* Plots a scatter plot between Age and Weight.
* alpha=0.5: Adds transparency to avoid overplotting.

### **Scatter Plot for Height vs. Weight with Trend Line**

plt.figure(figsize=[8,5])

sns.scatterplot(x=df["Height"], y=df["Weight"], alpha=0.5)

sns.regplot(x=df["Height"], y=df["Weight"], scatter=False, color="red")

plt.title("Height vs. Weight")

plt.xlabel("Height (cm)")

plt.ylabel("Weight (kg)")

plt.show()

* sns.scatterplot(): Creates a scatter plot of Height vs. Weight.
* sns.regplot(): Adds a regression (trend) line.
* scatter=False: Ensures only the trend line is plotted in red.

## **Correlation Analysis**

### **Computing Correlation Matrix**

correlation\_matrix = df[["Age", "Height", "Weight", "Year"]].corr()

* Calculates the correlation coefficients between numerical variables.

### **Plotting Correlation Heatmap**

plt.figure(figsize=[8,6])

sns.heatmap(correlation\_matrix, annot=True, cmap="coolwarm", fmt=".2f")

plt.title("Correlation Matrix")

plt.show()

* sns.heatmap(): Creates a heatmap to visualize correlations.
* annot=True: Displays correlation values in each cell.
* cmap="coolwarm": Uses a color gradient from cool (negative correlation) to warm (positive correlation).
* fmt=".2f": Limits decimal places to two.

## **Feature Selection**

### **Encoding Categorical Variables for Correlation Analysis**

from sklearn.preprocessing import LabelEncoder

categorical\_cols = df.select\_dtypes(include=["object"]).columns

for col in categorical\_cols:

df[col] = LabelEncoder().fit\_transform(df[col])

* Identifies categorical columns and applies LabelEncoder() to convert them into numerical values.

### **Feature Correlation Heatmap**

plt.figure(figsize=[12,6])

sns.heatmap(df.corr(), annot=True, cmap="coolwarm", fmt=".2f", linewidths=0.5)

plt.title("Feature Correlation Heatmap")

plt.show()

* Similar to the first heatmap but includes all features, allowing better insights into relationships.

## **Model Development**

### **Encoding the Target Variable**

from sklearn.ensemble import RandomForestClassifier

df["Medal"] = df["Medal"].fillna("No Medal") # Fill missing values

label\_encoder = LabelEncoder()

df["Medal\_encoded"] = label\_encoder.fit\_transform(df["Medal"].astype(str))

* Fills missing values in the "Medal" column with "No Medal."
* Encodes medal categories (Gold, Silver, Bronze) into numerical values.

### **Selecting Numerical Features for Correlation**

numeric\_features = ["Age", "Height", "Weight", "Year"]

correlation\_with\_medal = df[numeric\_features + ["Medal\_encoded"]].corr()["Medal\_encoded"].drop("Medal\_encoded")

* Selects numerical features and computes their correlation with the encoded medal variable.

### **Training a Random Forest Model for Feature Importance**

X = df[numeric\_features].dropna() # Drop rows with missing values

y = df.loc[X.index, "Medal\_encoded"] # Ensure alignment

rf\_model = RandomForestClassifier(n\_estimators=100, random\_state=42)

rf\_model.fit(X, y)

# Get feature importance scores

feature\_importance = pd.Series(rf\_model.feature\_importances\_, index=numeric\_features)

correlation\_with\_medal, feature\_importance

* Trains a RandomForestClassifier to evaluate feature importance.
* n\_estimators=100: Uses 100 decision trees.
* Computes feature importance scores, helping determine which numerical features influence medal outcomes the most.

### **Data Preprocessing and Type Conversion**

#### **Code:**

import pandas as pd

# Convert columns to numeric

df["Age"] = pd.to\_numeric(df["Age"], errors='coerce')

df["Height"] = pd.to\_numeric(df["Height"], errors='coerce')

df["Weight"] = pd.to\_numeric(df["Weight"], errors='coerce')

# Fill missing values (optional, choose a method)

df["Age"] = df["Age"].fillna(df["Age"].mean())

df["Height"] = df["Height"].fillna(df["Height"].mean())

df["Weight"] = df["Weight"].fillna(df["Weight"].mean())

# Verify if all columns are now numeric

print(df.dtypes)

#### **Explanation:**

* The script converts the "Age", "Height", and "Weight" columns to numeric using pd.to\_numeric(), handling errors by converting non-numeric values to NaN.
* Missing values are filled with the column’s mean using fillna().
* The print(df.dtypes) command prints the data types of all columns to verify that the conversion was successful.

### **2. Feature Selection and Model Training (Random Forest)**

#### **Code:**

from sklearn.ensemble import RandomForestClassifier

# Sample a smaller subset (10,000 rows) to reduce memory usage

df\_sample = df[numeric\_features + ["Medal\_encoded"]].dropna().sample(n=10000, random\_state=42)

# Prepare features and target variable

X\_sample = df\_sample[numeric\_features]

y\_sample = df\_sample["Medal\_encoded"]

# Train a smaller Random Forest model

rf\_model\_sample = RandomForestClassifier(n\_estimators=100, random\_state=42)

rf\_model\_sample.fit(X\_sample, y\_sample)

# Get feature importance scores

feature\_importance\_sample = pd.Series(rf\_model\_sample.feature\_importances\_, index=numeric\_features)

print(feature\_importance\_sample)

#### **Explanation:**

* A **Random Forest Classifier** is used for training.
* The dataset is sampled down to **10,000 rows** for memory efficiency.
* The model uses numeric features (X\_sample) to predict the target variable "Medal\_encoded" (y\_sample).
* A Random Forest model with **100 trees** (n\_estimators=100) is trained.
* **Feature importance scores** are extracted and printed to show which features contribute the most to predictions.

### **3. Handling Imbalanced Data**

#### **Code:**

!pip install imbalanced-learn

#### **Explanation:**

* The imbalanced-learn library is installed to handle imbalanced classification problems, which might be present in the dataset.

### **4. Checking Class Distribution**

#### **Code:**

python

CopyEdit

print(df["Medal"].value\_counts()) # Check if both 1 and 0 exist

#### **Explanation:**

* The distribution of medals (classes) in the dataset is printed using value\_counts(), which helps identify class imbalance issues.

### 

### 

### **5. Splitting Data into Training and Testing Sets**

#### **Code:**

from sklearn.model\_selection import train\_test\_split

# Assuming 'df' is your dataset and 'Medal' is the target column

X = df.drop(columns=["Medal"]) # Independent variables

y = df["Medal"] # Dependent variable (target)

# Splitting data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Now you can print the class distribution

print("Training set class distribution:")

print(y\_train.value\_counts())

print("Testing set class distribution:")

print(y\_test.value\_counts())

#### **Explanation:**

* The dataset is **split into training (80%) and testing (20%)** using train\_test\_split().
* The class distribution is printed for both sets to check for imbalances.

### **Key Takeaways:**

1. **Data Cleaning:**
   * Converted non-numeric values to numeric.
   * Handled missing values with mean imputation.
2. **Feature Selection & Model Training:**
   * Used Random Forest for feature importance evaluation.
   * Sampled a subset to reduce memory usage.
3. **Imbalanced Data Handling:**
   * Installed imbalanced-learn for handling class imbalances.
4. **Class Distribution Check:**
   * Verified medal counts before and after train-test split.

### 

### 

### 

### **Importing Necessary Libraries**

import pandas as pd

import numpy as np

import pickle

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model\_selection import train\_test\_split, GridSearchCV

from sklearn.preprocessing import StandardScaler

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score, classification\_report

from imblearn.over\_sampling import SMOTE # Handle Imbalance

#### **Explanation:**

* pandas, numpy: Data handling and numerical operations.
* pickle: Saving and loading models.
* matplotlib.pyplot, seaborn: Data visualization.
* sklearn.model\_selection: Splitting data and hyperparameter tuning.
* sklearn.preprocessing: Standard scaling.
* sklearn.linear\_model: Logistic regression.
* sklearn.metrics: Model performance evaluation.
* imblearn.over\_sampling.SMOTE: Handling class imbalance.

**2. Loading Dataset**

df = pd.read\_csv("athlete\_events.csv")

* Reads the dataset from a CSV file.

### **3. Data Preprocessing**

df = df[["Age", "Height", "Weight", "Year", "Medal"]].dropna().copy()

# Convert "Medal" column to binary classification (1 = Won a Medal, 0 = No Medal)

df["Medal"] = df["Medal"].notna().astype(int)

#### **Explanation:**

* Selects relevant columns and drops missing values.
* Converts "Medal" into a **binary** classification:
  + 1 if the athlete won a medal.
  + 0 if no medal.

### **4. Handling Imbalanced Data**

# Ensure at least two classes exist

if df["Medal"].nunique() < 2:

print("Warning: Dataset contains only one class. Adding synthetic 0 records.")

# Add synthetic "no-medal" records (based on averages)

new\_rows = pd.DataFrame({

"Age": [df["Age"].mean()] \* 100,

"Height": [df["Height"].mean()] \* 100,

"Weight": [df["Weight"].mean()] \* 100,

"Year": [df["Year"].mean()] \* 100,

"Medal": [0] \* 100 })

df = pd.concat([df, new\_rows], ignore\_index=True)

#### **Explanation:**

* Ensures at least two classes (0 and 1) exist.
* If only one class is present, synthetic "no-medal" records are **added** using mean values

### **5. Feature Scaling**

X = df.drop(columns=["Medal"])

y = df["Medal"]

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

#### **Explanation:**

* Features (X) and target (y) are separated.
* StandardScaler() is used to scale features.

### **6. Handling Imbalance with SMOTE**

smote = SMOTE(random\_state=42)

X\_scaled, y = smote.fit\_resample(X\_scaled, y)

#### **Explanation:**

* **SMOTE (Synthetic Minority Over-sampling Technique)** generates synthetic samples for the minority class.
* Balances the dataset by creating synthetic examples.

### **7. Splitting Data for Training and Testing**

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y, test\_size=0.2, random\_state=42, stratify=y)

#### **Explanation:**

* Splits data into **80% training** and **20% testing**.
* stratify=y ensures the class distribution remains balanced.

### **8. Model Training - Logistic Regression**

log\_reg = LogisticRegression()

log\_reg.fit(X\_train, y\_train)

#### **Explanation:**

* A **Logistic Regression** model is trained on the dataset.

### **9. Saving the Model and Scaler**

with open("logistic\_model.pkl", "wb") as model\_file:

pickle.dump(log\_reg, model\_file)

with open("scaler.pkl", "wb") as scaler\_file:

pickle.dump(scaler, scaler\_file)

#### **Explanation:**

* **Serializes** (saves) the trained model and scaler using pickle.

**10. Model Evaluation**

y\_pred = log\_reg.predict(X\_test)

print("\nModel Performance:")

print(f"Accuracy: {accuracy\_score(y\_test, y\_pred):.2f}")

print(f"Precision: {precision\_score(y\_test, y\_pred, zero\_division=1):.2f}")

print(f"Recall: {recall\_score(y\_test, y\_pred, zero\_division=1):.2f}")

print(f"F1-Score: {f1\_score(y\_test, y\_pred, zero\_division=1):.2f}")

print("\nClassification Report:\n", classification\_report(y\_test, y\_pred, zero\_division=1))

#### **Explanation:**

* **Predictions (y\_pred)** are generated.
* **Performance metrics**:
  + **Accuracy**: Percentage of correct predictions.
  + **Precision**: Correct positive predictions / All positive predictions.
  + **Recall**: Correct positive predictions / Actual positives.
  + **F1-Score**: Harmonic mean of precision and recall.
* **Classification report** provides a breakdown of metrics.

### **11. Model Hyperparameter Tuning**

import pickle as pkl

pkl.dump(log\_reg, open("model.pkl", "wb"))

pkl.dump(scaler, open("scaler.pkl", "wb"))

#### **Explanation:**

* Another instance of **saving the model**.