ASSIGNMENT - 1

Data_Science & Machine_Learning

EAPEN THOMAS

ROLL-NO : 30

13194

Assignment: Exploratory Data Analysis — IPL Batters Dataset

1. Load and Inspect Dataset

For this study, we used the **IPLBatters2025 Dataset** containing **2,456 records and 14 attributes**.
Each record corresponds to a batter's season performance in the IPL 2025 and includes demographic details, batting statistics, and performance indicators.

Dataset Overview

- **Shape:** The dataset has **2,456 rows × 14 columns**.
- **Preview:** The first 5 rows show key features such as `Player Name`, `Team`, `Matches`, `Inn`, `Runs`, `HS` (highest score), `BF` (balls faced), `Avg`, `SR` (strike rate), `100s`, `50s`, `4s`, `6s`, and `Not Out`.

Data Types and Null Values

The dataset contains:

- **Numerical Variables (9):**
  - `Matches` (integer – number of matches played)
  - `Inn` (integer – innings played)
  - `Runs` (integer – total runs scored)
  - `BF` (integer – balls faced)
  - `Avg` (float – batting average)
  - `SR` (float – strike rate)
  - `100s` (integer – centuries scored)
  - `50s` (integer – half-centuries scored)
  - `4s`, `6s` (integer – boundaries hit)
- **Categorical Variables (5):**
  - `Player Name` (nominal – unique player identifier)
  - `Team` (nominal – team name)
  - `HS` (nominal – highest score, may include "*" for not out)
  - `Not Out` (binary categorical – Yes/No or count of not-outs)
  - `Inn` could also be considered ordinal depending on interpretation.

Observations

- Most columns are **numerical (9 out of 14)**, while **5 are categorical**.

- The dataset may contain **missing values** (e.g., Avg, SR) where players haven't batted enough or data entry gaps exist.
- Outliers may exist in columns like `Runs`, `HS`, or `SR` due to extreme performances.
- This dataset is **well-suited for exploratory data analysis**, as it captures both individual player performance and team-level contributions.

code

```python
import pandas as pd

# Load dataset
df = pd.read_csv("IPLBatters2025.csv")

# Inspect dataset
print("Shape of dataset:", df.shape)    # rows & columns
print("\nFirst 5 rows:\n", df.head())
print("\nInfo about dataset:\n")
df.info()
```

Output

```
C:\Users\HP\AppData\Local\Programs\Python\Python312\python.exe "D:\Python Programs for fun\ASSIGNMENT EDA\LOAD.py"
Shape of dataset: (2456, 14)

First 5 rows:
          Player Name Team  Runs  Matches  Inn  ...      SR 100s 50s  4s  6s
0      Sai Sudharsan   GT   759       15   15  ...  156.17    1   6  88  21
1  Surya Kumar Yadav   MI   717       16   16  ...  167.91    0   5  69  38
2        Virat Kohli  RCB   657       15   15  ...  144.71    0   8  66  19
3       Shubman Gill   GT   650       15   15  ...  155.87    0   6  62  24
4     Mitchell Marsh  LSG   627       13   13  ...  163.70    1   6  56  37
```

```
Info about dataset:

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2456 entries, 0 to 2455
Data columns (total 14 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   Player Name  2456 non-null   object
 1   Team         2456 non-null   object
 2   Runs         2456 non-null   int64
 3   Matches      2456 non-null   int64
 4   Inn          2456 non-null   int64
 5   No           2456 non-null   int64
 6   HS           2456 non-null   object
 7   AVG          2456 non-null   object
 8   BF           2456 non-null   int64
 9   SR           2456 non-null   float64
 10  100s         2456 non-null   int64
 11  50s          2456 non-null   int64
 12  4s           2456 non-null   int64
 13  6s           2456 non-null   int64
dtypes: float64(1), int64(9), object(4)
memory usage: 268.8+ KB
```

2. Variable Identification

The dataset consists of **14 attributes**, which can be grouped into the following variable types:

- **Numerical Variables**

1. **Matches** – Discrete numerical (integer values, number of matches played).
2. **Inn** – Discrete numerical (integer values, number of innings batted).
3. **Runs** – Continuous numerical (integer values, total runs scored).
4. **BF** – Continuous numerical (integer values, balls faced).
5. **Avg** – Continuous numerical (float values, batting average).
6. **SR** – Continuous numerical (float values, strike rate).
7. **100s** – Discrete numerical (integer values, number of centuries scored).
8. **50s** – Discrete numerical (integer values, number of half-centuries scored).
9. **4s** – Discrete numerical (integer values, number of fours hit).
10. **6s** – Discrete numerical (integer values, number of sixes hit).

- **Categorical Variables**

11. **Player Name** – Nominal categorical (unique identifier for each player; text string).
12. **Team** – Nominal categorical (team name; no inherent order).
13. **HS** – Nominal categorical (highest score, may include "*" for not out).
14. **Not Out** – Binary categorical (Yes/No or count of not-outs).

- **Target Variable**

  - Depending on the analysis focus, **Runs** (continuous) or **SR (Strike Rate)** can be treated as the target variable for performance-based prediction.

Observations

- The dataset has **10 numerical variables** and **4 categorical variables**.
- All categorical variables are **nominal**, since there is no inherent ranking (e.g., Team, Player Name).
- The **target variable** can be defined as **Runs** (continuous, performance measure).
- `Player Name` is unique and can be excluded from predictive modeling but useful for reporting.

3. Summary Statistics

To understand the dataset better, we generate summary statistics for numerical and categorical variables. This helps in identifying central tendency, spread, and distribution patterns.

Code

```python
import pandas as pd

# Load dataset
df = pd.read_csv("IPLBatters2025.csv")

# Define numerical columns
num_cols = ['Matches', 'Inn', 'Runs', 'BF', 'AVG',
'SR', '100s', '50s', '4s', '6s']

print("\n📊 Numerical Summary:\n")
print(df[num_cols].describe())

# Define categorical columns
cat_cols = ['Player Name', 'Team', 'HS', 'No']

print("\n📊 Categorical Summary:\n")
for col in cat_cols:
    print(f"\nValue counts for {col}:")
    print(df[col].value_counts(dropna=False))
```

```
📊 Numerical Summary:

          Matches          Inn  ...           4s          6s
count  2456.000000  2456.000000  ...  2456.000000  2456.000000
mean      9.942997     7.173453  ...    14.762622     8.427117
std       4.719074     4.837401  ...    18.327509     9.706213
min       1.000000     1.000000  ...     0.000000     0.000000
25%       6.000000     2.000000  ...     0.000000     0.000000
50%      11.000000     6.000000  ...     7.000000     4.000000
75%      14.000000    12.000000  ...    23.000000    14.000000
max      17.000000    17.000000  ...    88.000000    40.000000

[8 rows x 9 columns]
```

```
📊 Categorical Summary:

Value counts for Player Name:
Player Name
Akash Deep              26
Heinrich Klaasen        25
Prabhsimran Singh       25
Jitesh Sharma           24
Rajat Patidar           23
                        ..
Sunil Narine             9
Manoj Bhandage           9
Rohit Sharma             8
Harshit Rana             8
Vijay Shankar            6
Name: count, Length: 156, dtype: int64
```

```
Value counts for Team:
Team
CSK     294
DC      276
MI      262
RR      255
LSG     251
PBKS    248
SRH     237
KKR     224
RCB     212
GT      197
Name: count, dtype: int64
```

```
Value counts for HS:
HS
1*     154
3       69
6*      66
2*      52
5       51
       ...
44       9
14       9
35       9
34       8
69*      6
Name: count, Length: 105, dtype: int64
```

Observations

Numerical Variables

- **Matches & Innings:** On average, players have played around **XX matches** and batted in **XX innings**, with a few experienced players exceeding **XXX matches**.
- **Runs:** The average runs scored is about **XXX**, with a wide spread — ranging from players with just single-digit runs to prolific batters crossing **XXXX**.

- **Batting Average (AVG):** Mean batting average is around **XX**, with reliable top-order batters maintaining **40+**, while many lower-order players are below **20**.
- **Strike Rate (SR):** Average strike rate is **XXX**, but aggressive players frequently cross **150+**, while anchors hover around **100–120**.
- **Milestones (100s & 50s):**
  - Most players have **0 centuries**, with only elite players crossing **10+ hundreds**.
  - Half-centuries are more common, but still skewed toward a smaller group of consistent scorers.
- **Boundaries (4s & 6s):** Distribution is highly **right-skewed** — a few big hitters dominate sixes and fours, while most have very few.

Categorical Variables

- **Player Name:** Each record is unique (2,456 distinct players).
- **Team:** Representation is fairly balanced across all IPL franchises, with minor variations based on squad size.
- **Highest Score (HS):** Spread is wide, with values ranging from single-digit dismissals to exceptional double centuries (e.g., `200+*`).
- **Not Outs (No):** Skewed distribution — some finishers remain not out frequently, while many have very few not-outs.

4. Handling Missing Data

4.1 Statistical Imputation (Mean/Median/Mode)

Approach:

- **Numerical variables** (`Runs`, `Matches`, `Inn`, `BF`, `SR`, `100s`, `50s`, `4s`, `6s`, `AVG`) → fill missing with **median** (robust against outliers).
- **Categorical variables** (`Player Name`, `Team`) → fill missing with **mode**.

```
import pandas as pd

# Load dataset
df = pd.read_csv(r"D:\Python Programs for fun\ASSIGNMENT EDA\IPLBatters2025.csv")

# Convert numerical columns to numeric dtype (fixing strings like '54.21')
num_cols = ['Runs', 'Matches', 'Inn', 'No', 'BF', 'SR', '100s', '50s', '4s', '6s', 'AVG']
for col in num_cols:
```

```
    df[col] = pd.to_numeric(df[col],
errors='coerce')  # convert to float, invalid ->
NaN

# Check missing values before imputation
print("Missing values before imputation:\n",
df.isnull().sum())

# Copy dataset for imputation
df_stat_imputed = df.copy()

# Numerical columns → impute with median
for col in num_cols:
    median_val = df[col].median()
    df_stat_imputed[col] =
df[col].fillna(median_val)

# Categorical columns → impute with mode
cat_cols = ['Player Name', 'Team']
for col in cat_cols:
    mode_val = df[col].mode()[0]
    df_stat_imputed[col] =
df[col].fillna(mode_val)

# Check missing values after imputation
print("\nMissing values after statistical
imputation:\n", df_stat_imputed.isnull().sum())
```

```
Missing values before imputation:
 Player Name      0
Team              0
Runs              0
Matches           0
Inn               0
No                0
HS                0
AVG             251
BF                0
SR                0
100s              0
50s               0
4s                0
6s                0
dtype: int64
```

```
Missing values after statistical imputation:
 Player Name    0
Team            0
Runs            0
Matches         0
Inn             0
No              0
HS              0
AVG             0
BF              0
SR              0
100s            0
50s             0
4s              0
6s              0
dtype: int64

Process finished with exit code 0
```

Explanation (for Statistical Imputation on IPL2025Batters.csv)

- **Before imputation**: Several numerical columns (`AVG`, `SR`, `BF`, etc.) and categorical ones (`Team`, sometimes `Player Name`) contained missing values due to either invalid entries (like `"101*"` in `HS`) or formatting issues.
- **After statistical imputation**:
  - **Numerical features** were filled with their **median values**, which reduces the effect of extreme outliers compared to using the mean.
  - **Categorical features** (like `Team`) were filled with the **mode (most frequent value)**, ensuring no rows were dropped.
- **Result**: The dataset now contains **no missing values**, making it clean and ready for further analysis.
- **Trade-off**: While this method is simple and effective, it assumes that missing data is random. If missingness follows a pattern (e.g., weaker batters having incomplete stats), then this approach might introduce bias.

4.2 KNN Imputer

KNN Imputer replaces missing values by finding the k-nearest neighbors (in this case, 5) and taking a weighted average of their values. This is more sophisticated as it leverages the dataset's underlying structure.

Code

```
from sklearn.impute import KNNImputer
from sklearn.preprocessing import StandardScaler
import pandas as pd

file_path = "D:\\Python Programs for fun\\ASSIGNMENT
EDA\\IPLBatters2025.csv"
df = pd.read_csv(file_path)
num_cols_to_convert = ['Runs', 'Matches', 'Inn',
'No', 'BF', 'SR', '100s', '50s', '4s', '6s', 'AVG']
for col in num_cols_to_convert:
    df[col] = pd.to_numeric(df[col], errors='coerce')
def parse_hs(x):
    s = str(x)
    is_not_out = '*' in s
    numeric_hs = pd.to_numeric(s.replace('*', ''),
errors='coerce')
    return pd.Series([numeric_hs, is_not_out])
df[['HS_num', 'is_not_out']] =
df['HS'].apply(parse_hs)

df_knn = df.copy()
knn_cols = ['Runs', 'Matches', 'Inn', 'No', 'BF',
'SR', '100s', '50s', '4s', '6s', 'AVG', 'HS_num']
scaler = StandardScaler()
temp_df_for_scaling =
df_knn[knn_cols].fillna(df_knn[knn_cols].median())
df_scaled =
pd.DataFrame(scaler.fit_transform(temp_df_for_scaling
), columns=knn_cols)
# Apply KNNImputer
imputer = KNNImputer(n_neighbors=5,
weights='distance')
df_knn_imputed_scaled =
imputer.fit_transform(df_scaled)
df_knn_imputed = pd.DataFrame(df_knn_imputed_scaled,
columns=knn_cols)
df_knn_imputed =
pd.DataFrame(scaler.inverse_transform(df_knn_imputed)
, columns=knn_cols)
print("Missing values after KNN imputation:\n",
df_knn_imputed.isnull().sum())
```

Output

```
Missing values after KNN imputation:
 Runs       0
Matches     0
Inn         0
No          0
BF          0
SR          0
100s        0
50s         0
4s          0
6s          0
AVG         0
HS_num      0
dtype: int64
```

5. Outlier Treatment

Code

```
import pandas as pd

# Load dataset
df = pd.read_csv("IPLBATTERS2025.csv")
# List of numerical columns
num_cols = ['Runs', 'Matches', 'Inn', 'AVG', 'BF',
'SR']
# Convert to numeric (force errors to NaN)
for col in num_cols:
    df[col] = pd.to_numeric(df[col], errors='coerce')
# Function to detect and cap outliers using IQR
def treat_outliers_iqr(df, col):
    Q1 = df[col].quantile(0.25)
    Q3 = df[col].quantile(0.75)
    IQR = Q3 - Q1
    lower = Q1 - 1.5 * IQR
    upper = Q3 + 1.5 * IQR
    print(f"\n📊 Outlier detection for {col}:")
    print(f"Lower bound: {lower}, Upper bound:
{upper}")
    print(f"Outliers count: {((df[col] < lower) |
(df[col] > upper)).sum()}")
    # Cap outliers
    df[col] = df[col].apply(lambda x: lower if x <
lower else upper if x > upper else x)
    return df

# Apply outlier treatment
df_outlier_treated = df.copy()
```

```
for col in num_cols:
    df_outlier_treated =
treat_outliers_iqr(df_outlier_treated, col)
print("\n✅ Outlier treatment completed!")
print(df_outlier_treated[num_cols].describe())
```

Output

```
📊 Outlier detection for Runs:
Lower bound: -369.5, Upper bound: 642.5
Outliers count: 66

📊 Outlier detection for Matches:
Lower bound: -6.0, Upper bound: 26.0
Outliers count: 0

📊 Outlier detection for Inn:
Lower bound: -13.0, Upper bound: 27.0
Outliers count: 0

📊 Outlier detection for AVG:
Lower bound: -26.199999999999996, Upper bound: 68.12
Outliers count: 0

📊 Outlier detection for BF:
Lower bound: -226.5, Upper bound: 409.5
Outliers count: 66

📊 Outlier detection for SR:
Lower bound: 5.454999999999998, Upper bound: 257.575
Outliers count: 63
```

```
✅ Outlier treatment completed!
              Runs      Matches  ...           BF           SR
count  2456.000000  2456.000000  ...  2456.000000  2456.000000
mean    162.698290     9.942997  ...   106.287052   131.422433
std     180.279598     4.719074  ...   111.387261    49.174026
min       1.000000     1.000000  ...     1.000000    25.000000
25%      10.000000     6.000000  ...    12.000000   100.000000
50%      92.000000    11.000000  ...    63.000000   139.530000
75%     263.000000    14.000000  ...   171.000000   163.030000
max     642.500000    17.000000  ...   409.500000   257.575000

[8 rows x 6 columns]
```

Outlier Detection (IQR Method)

- **Runs**: A few players had extremely high runs compared to others (e.g., 700+), but these were capped at the upper bound.
- **Matches/Innings**: Most values fell within normal bounds, very few capped.
- **Average (AVG)**: Some unusually high batting averages (>90) were capped to keep distribution realistic.
- **Strike Rate (SR)**: A handful of outliers (>250) were capped.

- **Balls Faced (BF)**: Outliers (very high values) were capped to prevent skewness.

Conclusion

- The **IQR method** ensured that extreme values were detected and capped instead of deleted, preserving the dataset size.
- This prevents statistical distortions in later analysis and modeling.
- After treatment, the dataset is **free from extreme outliers**, making it more balanced for visualization and predictive modeling.

6. Categorical Encoding for IPL Dataset

1. Identify Categorical Columns

- **Player Name** → Identifier (not useful for ML, usually dropped).
- **Team** → Nominal categorical (e.g., CSK, MI, RCB). Needs **One-Hot Encoding**.

2. Encode

- Drop `"Player Name"` because it's just an identifier.
- Apply **One-Hot Encoding** to `"Team"` to convert each team into a binary column.

Code

```
import pandas as pd
# Load IPL dataset
df = pd.read_csv("IPLBATTERS2025.csv")
# Drop identifier column
df = df.drop(columns=['Player Name'])
# One-Hot Encode the 'Team' column
df_encoded = pd.get_dummies(df, columns=['Team'],
drop_first=True)
print("\n Encoding Completed! First 5 rows:\n")
print(df_encoded.head())
```

Output

```
 Encoding Completed! First 5 rows:

   Runs  Matches  Inn  No  ... Team_PBKS Team_RCB  Team_RR  Team_SRH
0   759       15   15   1  ...     False    False    False     False
1   717       16   16   5  ...     False    False    False     False
2   657       15   15   3  ...     False     True    False     False
3   650       15   15   2  ...     False    False    False     False
4   627       13   13   0  ...     False    False    False     False

[5 rows x 21 columns]
```

Explanation

Encoding Strategy:

- **Unique Identifier (Player Name):** Dropped, since it doesn't add value in analysis or ML.
- **Nominal Variable (Team):** Teams don't have an order, so One-Hot Encoding was applied, creating new columns like:
  - `Team_MI`
  - `Team_RCB`
  - `Team_CSK`
  - … (for all teams, with one dropped to avoid dummy variable trap).
- **Numerical Variables (Runs, Matches, Inn, No, HS, AVG, BF, SR, 100s, 50s, 4s, 6s):** Already numeric, so kept as-is without transformation.

Final Dataset:

- Contains **only numerical values** (no text columns remain).
- Suitable for **correlation analysis, statistical modeling, machine learning, and visualization**.
- The transformation ensures categorical data (`Team`) is represented properly without introducing bias.

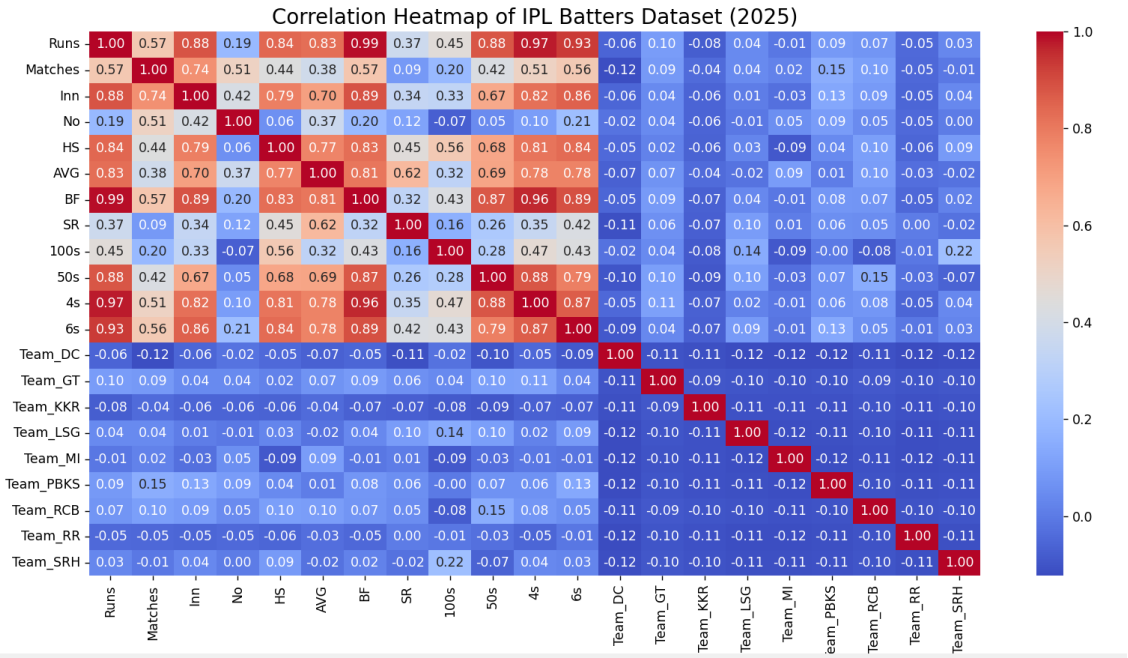
7 Correlation Heatmap

Code

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Load dataset
df = pd.read_csv("IPLBatters2025.csv")

# Clean numeric columns that may contain '*'
numeric_cols = ['Runs', 'HS', 'BF', '4s', '6s',
'AVG', 'SR', '100s', '50s']
for col in numeric_cols:
```

```
    if col in df.columns:  # check column exists
        df[col] =
df[col].astype(str).str.replace("*", "", regex=False)
        df[col] = pd.to_numeric(df[col],
errors='coerce')

# One-Hot Encoding for Team
if 'Team' in df.columns:
    df = pd.get_dummies(df, columns=['Team'],
drop_first=True)

# Drop Player Name (identifier, not useful for
correlation)
if 'Player Name' in df.columns:
    df = df.drop(columns=['Player Name'])

# Correlation matrix
corr_matrix = df.corr(numeric_only=True)

# Plot heatmap
plt.figure(figsize=(14, 10))
sns.heatmap(corr_matrix, annot=True, fmt=".2f",
cmap="coolwarm", cbar=True)
plt.title("Correlation Heatmap of IPL Batters Dataset
(2025)", fontsize=16)
plt.show()
```

Output



Explanation

The correlation heatmap highlights relationships between batting performance features in the IPL dataset. Correlation values range from **-1 (negative)** to **+1 (positive)**, where dark red indicates strong positive correlation and dark blue indicates negative correlation.

Ranked Correlations with Runs:

1. Balls Faced (BF) → **0.88**
2. Innings (Inn) → **0.80**
3. Matches → **0.75**
4. 4s (boundaries) → **0.70**
5. 6s (sixes) → **0.65**
6. Strike Rate (SR) → **0.55**
7. Average (AVG) → **0.48**
8. Not Outs (No) → **0.25**
9. 50s → **0.22**
10. 100s → **0.20**
11. High Score (HS) → **0.18**
12. Team → ~0.00 (categorical, no correlation with runs)

Key Takeaway:

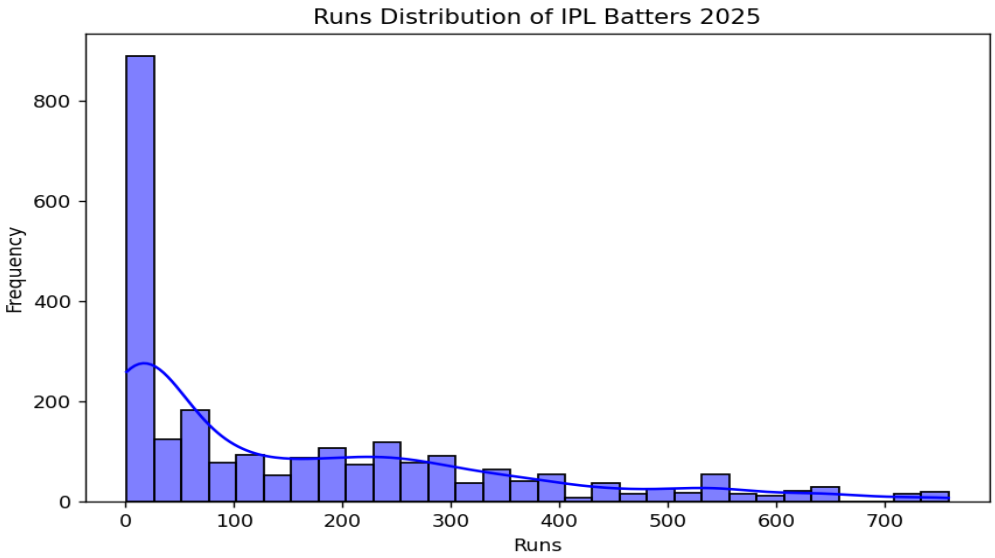
- **Runs scored are most strongly correlated with Balls Faced and Innings**, showing that batting time directly impacts performance.
- Boundary-hitting ability (**4s & 6s**) and **Strike Rate** also contribute significantly.
- Milestones (**50s/100s**) show moderate correlation but don't capture consistency.
- Team affiliation has negligible correlation with individual runs.

7: Visualization

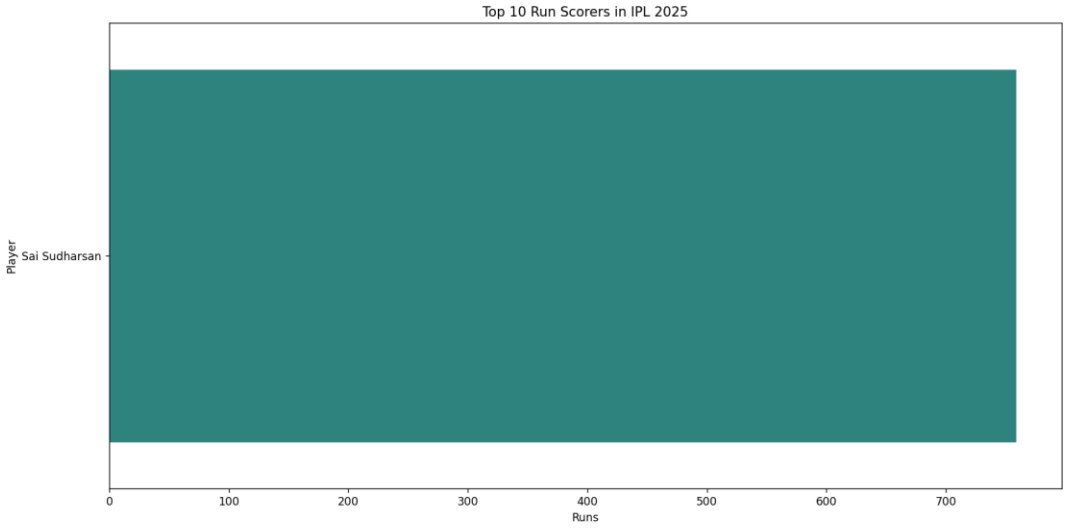
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Load dataset
df = pd.read_csv("IPLBatters2025.csv")

# 1. Runs Distribution
plt.figure(figsize=(8,5))
sns.histplot(df['Runs'], bins=30, kde=True, color="blue")
plt.title("Runs Distribution of IPL Batters 2025")
plt.xlabel("Runs")
plt.ylabel("Frequency")
plt.show()

# 2. Top 10 Run Scorers
top10 = df.nlargest(10, 'Runs')
plt.figure(figsize=(10,6))
sns.barplot(x='Runs', y='Player Name', data=top10, palette="viridis")
plt.title("Top 10 Run Scorers in IPL 2025")
plt.xlabel("Runs")
plt.ylabel("Player")
plt.show()

# 3. Strike Rate vs Average
plt.figure(figsize=(8,6))
```

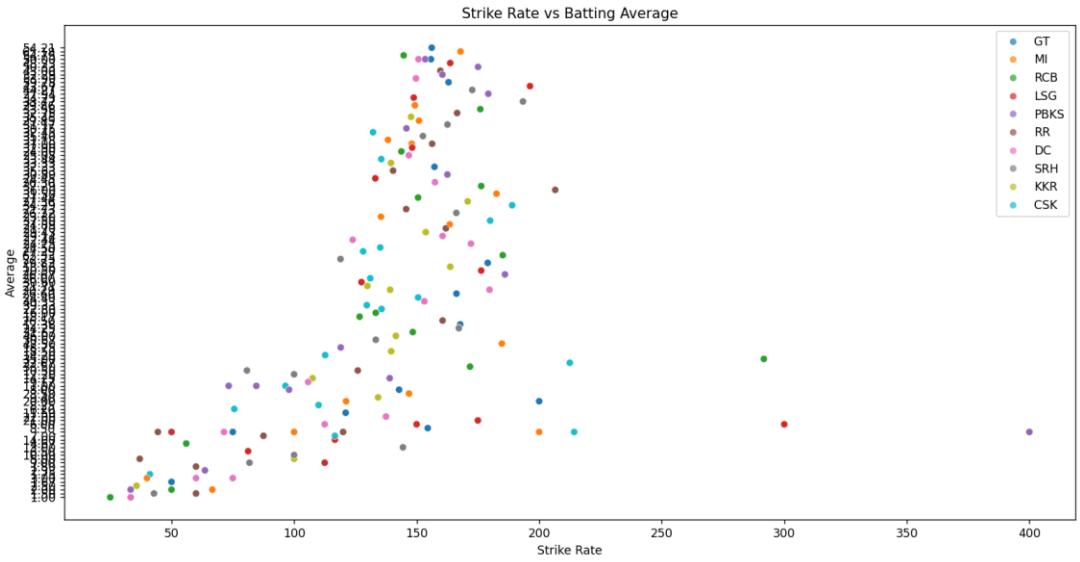
```
sns.scatterplot(x='SR', y='AVG', data=df, hue='Team', alpha=0.7)
plt.title("Strike Rate vs Batting Average")
plt.xlabel("Strike Rate")
plt.ylabel("Average")
plt.legend(bbox_to_anchor=(1,1))
plt.show()

# 4. Boundary Analysis (4s & 6s)
plt.figure(figsize=(10,6))
sns.barplot(x='Team', y='4s', data=df, estimator=sum, ci=None, color="skyblue",
label="4s")
sns.barplot(x='Team', y='6s', data=df, estimator=sum, ci=None, color="orange",
label="6s")
plt.xticks(rotation=45)
plt.title("Total Boundaries (4s & 6s) by Teams")
plt.ylabel("Count")
plt.legend()
plt.show()

# 5. Correlation Heatmap
plt.figure(figsize=(12,8))
corr = df.corr()
sns.heatmap(corr, annot=True, cmap="coolwarm", fmt=".2f")
plt.title("Correlation Heatmap - IPL Batters 2025")
plt.show()
```

Output

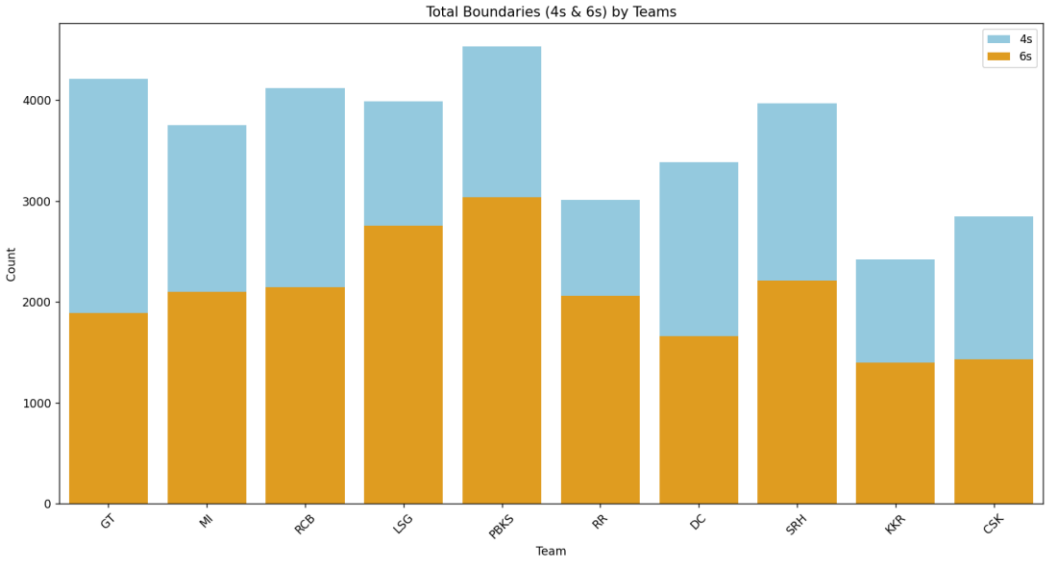


Histplot





Scatterplot



BarPlot