ipl

May 14, 2023

[1]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import plotly.express as px
import plotly.graph_objects as go
from plotly.subplots import make_subplots
import colorama
from colorama import Fore, Back, Style
import mplcursors
```

[2]:
```
ball_data = pd.read_csv("IPL_Ball_by_Ball_2008_2022.csv")
match_data = pd.read_csv("IPL_Matches_2008_2022.csv")
```

These lines of code are defining the column names for two data frames, `match_data` and `ball_data`, and assigning them to the variables `match_data.columns` and `ball_data.columns`, respectively.

The `match_data` data frame has 20 columns, with each column representing a different aspect of a cricket match. The first column is the match ID, followed by the city, date, season, match number, teams (team1 and team2), venue, toss winner, toss decision, whether or not the match went into a super over, the winning team, the margin of victory, the method of victory, the player of the match, the players on team1, the players on team2, and the names of the two umpires.

The `ball_data` data frame has 17 columns, with each column representing a different aspect of a ball in a cricket match. The first column is the match ID, followed by the inning number, the over number, the ball number within the over, the batter's name, the bowler's name, the name of the non-striker, information on any extras (extra type, number of extra runs), the number of runs scored by the batter, the total number of runs scored on the ball (including extras), whether or not the ball was a boundary, whether or not there was a wicket, the player who was out (if applicable), the type of dismissal, the fielders involved (if applicable), and the name of the batting team.

Finally, the `colorsy` variable is a list of color values, which could be used to create custom color palettes for visualizations of the data.

[3]:
```
match_data.columns =␣
↪['id','city','date','season','matchnumber','team1','team2','venue',
                    ␣
↪'tosswinner','tossdecision','superover','winningteam','wonby','margin',
```

```
                        ␣
 ↪'method','player_of_match','team1players','team2players','umpire1','umpire2']

ball_data.columns  =␣
 ↪['id','inning','over','ballnumber','batter','bowler','non_striker',
                        ␣
 ↪'extra_type','batsman_run','extras_run','total_runs','non_boundary',
                         'is_wicket',␣
 ↪'player_out','dismissal_kind','fielders_involved','battingteam']
colorsy= ['#0d293f', '#1f4956', '#185175', '#19618f', '#43779a', '#147eb0',␣
 ↪'#2d93ca', '#58a9db', '#8ab5d8', '#9ccef0']
```

These lines of code print a header with the title "Match Data" in bright blue color using the `Back` and `Style` classes from the `colorama` package.

The next line of code creates a new data frame called `match_data2` by dropping the `team1players` and `team2players` columns from the original `match_data` data frame using the `drop()` method with `axis=1`. The `axis=1` argument specifies that we want to drop columns rather than rows.

Finally, the code uses the `style()` method to apply some styling to the first five rows of the `match_data2` data frame. Specifically, it sets the background color of the cells to a light blue shade, adds a black border around each cell with a width of 1.3 pixels, sets the text color to black, and sets the border color to black. The `set_properties()` method is used with a dictionary of properties to apply the styling. The `**` before the dictionary name unpacks the dictionary into keyword arguments, which are passed to the `set_properties()` method.

Note that the `style()` method returns a `Styler` object, which can be used to apply additional styling or to format the data frame before displaying it. In this case, the styled data frame is not displayed directly in the code cell, so we cannot see the actual styling applied. To display the styled data frame, we would need to call the `.render()` method on the `Styler` object or display it using `IPython.display.HTML()`.

[4]:
```
print(Back.BLUE+ Style.BRIGHT+'Match Data'+ Style.RESET_ALL)
match_data2 = match_data.drop(['team1players', 'team2players'], axis=1)
match_data2.head().style.set_properties(**{'background-color': '#cde6f7'␣
 ↪,'border': '1.3px solid black','color':'black','border-color': '#000000'})
```

```
Match Data
```

[4]:
```
<pandas.io.formats.style.Styler at 0x17562b2e760>
```

These lines of code print a header with the title "Ball Data" in bright blue color using the `Back` and `Style` classes from the `colorama` package.

The next line of code uses the `.head()` method to display the first five rows of the `ball_data` data frame. The `.style` attribute is then used to access the `Styler` object for the data frame, and the `set_properties()` method is called on this object to apply some styling.

The styling applied is similar to that applied to the `match_data2` data frame in the previous code cell. Specifically, it sets the background color of the cells to a light blue shade, adds a black border

around each cell with a width of 1.3 pixels, sets the text color to black, and sets the border color to black. The `set_properties()` method is used with a dictionary of properties to apply the styling. The `**` before the dictionary name unpacks the dictionary into keyword arguments, which are passed to the `set_properties()` method.

Note that the styled data frame is not displayed directly in the code cell, so we cannot see the actual styling applied. To display the styled data frame, we would need to call the `.render()` method on the `Styler` object or display it using `IPython.display.HTML()`.

[5]:
```
print(Back.BLUE+ Style.BRIGHT+'Ball Data'+ Style.RESET_ALL)
ball_data.head().style.set_properties(**{'background-color': '#cde6f7'␣
 ↪,'border': '1.3px solid black','color':'black','border-color': '#000000'})
```

```
Ball Data
```

[5]:
```
<pandas.io.formats.style.Styler at 0x1756a770e50>
```

These lines of code add three new columns to the `ball_data` data frame, initialized with empty string values.

The first line of code adds a new column called `bowlingteam`, which will be used to store the name of the team that is bowling during each ball.

The second line of code adds a new column called `first_batter`, which will be used to store the name of the batter that is facing the ball during each ball.

The third line of code adds a new column called `second_batter`, which will be used to store the name of the non-striker during each ball.

[6]:
```
ball_data['bowlingteam'] = ''
ball_data['first_batter'] = ''
ball_data['second_batter'] = ''
```

These two lines of code are used to populate the `first_batter` and `second_batter` columns in the `ball_data` data frame.

The first line of code uses the `apply()` method to apply a lambda function to each row of the data frame along the specified axis (axis=1, meaning apply the function to each row). The lambda function takes a row (`x`) as input, and checks whether the `batter` column is less than the `non_striker` column. If it is, the lambda function returns the value in the `batter` column, and if it is not, the lambda function returns the value in the `non_striker` column. The result of applying the lambda function to each row is a new series with the same length as the number of rows in the data frame. This new series is then assigned to the `first_batter` column of the data frame.

The second line of code does the same thing, but for the `second_batter` column. The lambda function checks whether the `batter` column is less than the `non_striker` column, and returns the value in the `non_striker` column if it is, and the value in the `batter` column if it is not. The resulting series is assigned to the `second_batter` column of the data frame.

These lines of code effectively identify which of the two batters listed in each row is the first batter (i.e., the one facing the ball) and which is the second batter (i.e., the non-striker). The `apply()` method is used with a lambda function to apply this logic to each row of the data frame.

[7]:
```
ball_data['first_batter']  = ball_data.apply(lambda x: x['batter'] if␣
 ↪x['batter']  <  x['non_striker'] else x['non_striker'], axis=1)
ball_data['second_batter'] = ball_data.apply(lambda x: x['non_striker'] if␣
 ↪x['batter']  <  x['non_striker'] else x['batter'], axis=1)
```

The first line of code performs a left join between the `ball_data` and `match_data` data frames on the `id` column. The result of the merge is a new data frame that contains all the columns from `ball_data` and the columns `team1`, `team2`, and `season` from `match_data`. The `how` parameter specifies that a left join should be performed, meaning that all rows from `ball_data` are included in the merged data frame, and any matching rows from `match_data` are included as well. Any non-matching rows from `match_data` are not included in the merged data frame.

The second line of code uses the `apply()` method to apply a lambda function to each row of the `ball_data` data frame. The lambda function checks whether the `battingteam` column is equal to the `team1` column. If it is, the lambda function returns the value in the `team2` column, indicating that the bowling team for that ball is `team2`. If it is not, the lambda function returns the value in the `team1` column, indicating that the bowling team for that ball is `team1`. The result of applying the lambda function to each row is a new series with the same length as the number of rows in the data frame. This new series is then assigned to the `bowlingteam` column of the `ball_data` data frame.

These lines of code effectively add two new columns to the `ball_data` data frame: `team1` and `team2`, which represent the two teams playing each match, and `bowlingteam`, which indicates which team is bowling during each ball. By performing a left join between `ball_data` and `match_data`, the `team1` and `team2` columns are added to the `ball_data` data frame, allowing for the `bowlingteam` column to be populated using the `apply()` method.

[8]:
```
ball_data = pd.merge(ball_data, match_data[['id','team1','team2','season']],␣
 ↪how='left', on='id')
ball_data['bowlingteam'] = ball_data.apply(lambda x: x['team2'] if␣
 ↪x['battingteam'] == x['team1'] else x['team1'], axis=1)
```

This line of code creates a new column called `win_against` in the `match_data` dataframe. The `apply()` method is used with a lambda function to iterate through each row of the dataframe. For each row, the lambda function checks whether the `winningteam` column is equal to the `team1` column. If it is, then the `win_against` column is assigned the value of the `team2` column for that row. If it is not, then the `win_against` column is assigned the value of the `team1` column for that row.

Essentially, this code is determining which team a given team won against in each match. If `winningteam` is equal to `team1`, then the team won against `team2`, so `team2` is assigned to the `win_against` column. If `winningteam` is equal to `team2`, then the team won against `team1`, so `team1` is assigned to the `win_against` column.

[9]:
```
match_data['win_against'] = match_data.apply(lambda x: x['team2'] if␣
 ↪x['winningteam'] == x['team1'] else x['team1'], axis=1)
```

This block of code does the following: - The `ball_data` dataframe is grouped by `id`, `season`, `first_batter`, `second_batter`, `battingteam`, and `bowlingteam`, and the `total_runs` column is

aggregated by sum to get the total partnership runs for each pair of batters in each match. - The resulting dataframe is sorted in descending order by partnership runs and the top 10 partnerships are selected. - A new `Figure` object from the `plotly.graph_objects` module is created, with a bar chart showing the top 10 partnerships in the IPL. - The `x` axis of the bar chart shows the batting team for each partnership. - The `y` axis shows the partnership runs for each partnership. - The color of each bar is determined by the names of the two batters in the partnership (`first_batter` and `second_batter`). - A color palette `colorsy` is defined to use for the bar chart. - The title of the chart is set to 'Top 10 Partnerships in IPL'. - The partnership runs for each bar are displayed on the chart. - The chart dimensions are set to `height=500`. - The font family, font colors, and font sizes for various elements of the chart are customized using `update_layout()`. - Finally, the chart is displayed using the `show()` method.

[10]:
```
partnership_data=ball_data.groupby(['id','season','first_batter',␣
 ↪'second_batter','battingteam','bowlingteam'])\
                     .agg(partnership=('total_runs','sum')).reset_index()
top10_partnerships = partnership_data.
 ↪sort_values(by='partnership',ascending=False).head(10)
top10_partnerships = top10_partnerships.
 ↪sort_values(by='partnership',ascending=True)

fig = go.Figure(data=px.bar(x=top10_partnerships.battingteam,
                            y=top10_partnerships.partnership,
                            color = top10_partnerships.first_batter+' &␣
 ↪'+top10_partnerships.second_batter,
                            color_discrete_sequence=colorsy,
                            title='<b>Top 10 Partnerships in IPL</b>',
                            text = top10_partnerships.partnership,
                            height=500))
fig.update_layout(
    font_family="Courier New",
    title_font_family="Times New Roman",
    title_font_color="red",
    title_font_size=20,
    xaxis_title="<b>Batting Team</b>",
    yaxis_title="<b>Partnership</b>",
    legend_title_font_color="green"
)
fig.show()
```

The code above generates a bar chart of the top partnerships for each season in the IPL.

First, the code calculates the ranking of partnerships within each season using `groupby` and `rank` function. Then, it selects only those partnerships that have the rank 1 for each season and stores it in a new dataframe called `season_top_partnership`.

Finally, it creates a bar chart using the `season_top_partnership` dataframe with `px.bar` from Plotly Express. The x-axis represents the season, the y-axis represents the partnership value, and the color of the bars represents the combination of the first and second batter of the partnership. The chart is customized with a title, axis labels, font styles, and a color palette.

[11]:
```
partnership_data['Rank'] = partnership_data.groupby('season')['partnership'].
 ↪rank(ascending=False)
season_top_partnership=partnership_data[partnership_data['Rank']==1].
 ↪sort_values('season')
fig = go.Figure(data=px.bar(x=season_top_partnership.season,
                            y=season_top_partnership.partnership,
                            color = season_top_partnership.first_batter+' &␣
 ↪'+season_top_partnership.second_batter,
                            color_discrete_sequence=colorsy,
                            title='<b>Season-wise Top Partnerships in IPL</b>',
                            text = season_top_partnership.partnership,
                            height=500))
fig.update_layout(
    font_family="Courier New",
    title_font_family="Times New Roman",
    title_font_color="red",
    title_font_size=20,
    xaxis_title="<b>Season</b>",
    yaxis_title="<b>Partnership</b>",
    legend_title_font_color="green"
)
fig.show()
```

This code calculates the top 10 batsmen in IPL based on their total runs scored. It first groups the ball_data by each batsman and calculates the total runs scored by each batsman using the `agg()` method. It then sorts the values in descending order based on the `batsman_total` column and selects the top 10 batsmen using the `head()` method.

Finally, it creates a pie chart using the `px.pie()` method from the Plotly library. The chart displays the total runs scored by each of the top 10 batsmen, with their names as the labels for each slice of the pie. The colors of the slices are chosen using the `color_discrete_sequence` argument. The `update_traces()` and `update_layout()` methods are used to customize the appearance of the chart, including the font size, text position, and title.

[12]:
```
top_batsman=ball_data.groupby(['batter']).agg(batsman_total=('batsman_run',␣
 ↪'sum')).reset_index()\
                .sort_values(by='batsman_total', ascending=False).head(10)

fig = px.pie(values=top_batsman.batsman_total,
             names=top_batsman.batter,
             color_discrete_sequence=px.colors.sequential.Plasma)
fig.update_traces(textposition='inside',
                  textfont_size=11,
                  textinfo='value+label')

fig.update_layout(title_text="<b>Top 10 Batsmen</b>",
                  title_font_family="Times New Roman",
```

```
                  title_font_color="red",
                  title_font_size=20,
                  uniformtext_minsize=12,
                  uniformtext_mode='hide')

fig.show()
```

This code is generating a dataframe `top_batsman_score` with the details of the top 10 batsmen in IPL. The details include the number of fours, sixers, fifties, hundreds, highest score, total runs and matches played for each batsman.

First, the code initializes an empty dataframe with the required columns. Then, for each of the top 10 batsmen, the code filters the ball_data dataframe to get all the records where the player is the batter, and extracts the columns 'id', 'batter', and 'batsman_run'. It then calculates the number of fours and sixers the player has scored, the number of innings where the player scored fifties and hundreds, the highest score of the player, and the number of matches played by the player.

Finally, the code appends a new row to the `top_batsman_score` dataframe with the calculated values. The dataframe is then styled with some formatting to make it more readable.

[13]:
```
print(Back.GREEN+ Style.BRIGHT+'Top 10 Batsmen Score Details'+ Style.RESET_ALL)
top_batsman_score = pd.DataFrame(columns=['batsman', 'fours', 'sixers',␣
 ↪'fifties', 'hundreds','heighest_score','total_runs', 'matches_played'])
for idx, row in top_batsman.iterrows():
    batsman_data = ball_data[ball_data['batter'] == row['batter']][['id',␣
 ↪'batter', 'batsman_run']]

    r4 = len(batsman_data[batsman_data['batsman_run'] == 4])
    r6 = len(batsman_data[batsman_data['batsman_run'] == 6])

    innings_score = batsman_data.groupby('id').agg(score=('batsman_run',␣
 ↪'sum')).reset_index()
    r50=len(innings_score[(innings_score['score'] >=50) &␣
 ↪(innings_score['score'] < 100)])
    r100 = len(innings_score[innings_score['score'] >= 100])
    matches_played =  len(innings_score)
    heighest_score = innings_score['score'].max()


    top_batsman_score = top_batsman_score.append({'batsman': row['batter'],␣
 ↪'fours': r4, 'sixers': r6, 'fifties':r50,
                                                  'hundreds':r100, 'total_runs':
 ↪row['batsman_total'],
                                                  'heighest_score':
 ↪heighest_score,
                                                  'matches_played':
 ↪matches_played},ignore_index=True)
```

```
top_batsman_score.style.set_properties(**{'background-color': '#cde6f7'␣
↪,'border': '1.3px solid black','color':'black','border-color': '#000000'})
```

Top 10 Batsmen Score Details

```
C:\Users\suman\AppData\Local\Temp\ipykernel_16608\3820994963.py:16:
FutureWarning:

The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.

C:\Users\suman\AppData\Local\Temp\ipykernel_16608\3820994963.py:16:
FutureWarning:

The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.

C:\Users\suman\AppData\Local\Temp\ipykernel_16608\3820994963.py:16:
FutureWarning:

The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.

C:\Users\suman\AppData\Local\Temp\ipykernel_16608\3820994963.py:16:
FutureWarning:

The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.

C:\Users\suman\AppData\Local\Temp\ipykernel_16608\3820994963.py:16:
FutureWarning:

The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.

C:\Users\suman\AppData\Local\Temp\ipykernel_16608\3820994963.py:16:
FutureWarning:

The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.

C:\Users\suman\AppData\Local\Temp\ipykernel_16608\3820994963.py:16:
FutureWarning:

The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.

C:\Users\suman\AppData\Local\Temp\ipykernel_16608\3820994963.py:16:
```

```
FutureWarning:

The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.

C:\Users\suman\AppData\Local\Temp\ipykernel_16608\3820994963.py:16:
FutureWarning:

The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.

C:\Users\suman\AppData\Local\Temp\ipykernel_16608\3820994963.py:16:
FutureWarning:

The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.
```

[13]: <pandas.io.formats.style.Styler at 0x1756b78be20>

This code creates a subplot figure to display the performance details of the top 10 batsmen in the IPL. The figure contains six subplots arranged in 2 columns and 3 rows, with each subplot showing a different performance metric for the top 10 batsmen.

The code first creates an empty figure with `make_subplots()` and sets the number of columns, rows, and the subplot titles. Then, it adds six subplots to the figure using `add_trace()` and specifying the subplot location with the `row` and `col` arguments.

For each subplot, a bar chart is created using `go.Bar()` with the appropriate x and y values. The x-values are the names of the top 10 batsmen, and the y-values are the corresponding performance metric. The `name` argument is used to set the subplot title.

Finally, the code updates the layout of the figure using `update_layout()` to set the figure title, size, and font properties.

Overall, this code provides a clear and concise way to display the performance details of the top 10 batsmen in the IPL.

[14]:
```
fig = make_subplots(cols=2,rows=3,
    vertical_spacing = 0.25, horizontal_spacing=.1,
                    subplot_titles=["No. of Sixers","No. of Fours",
                                    "No. of Half-Centuries","No. of Centuries",
                                   "Highest Scores", "Total Runs"],
                    y_title="<b>Top 10 Batsmen Run Details</b>"
    )
fig.add_trace(go.Bar(x=top_batsman_score.batsman,y=top_batsman_score.
  ↪sixers,name="Highest Sixers"),row=1,col=1)
fig.add_trace(go.Bar(x=top_batsman_score.batsman,y=top_batsman_score.
  ↪fours,name="Highest Fours"),row=1,col=2)
```

```
fig.add_trace(go.Bar(x=top_batsman_score.batsman,y=top_batsman_score.
 ↪fifties,name="Highest Fifties"),row=2,col=1)
fig.add_trace(go.Bar(x=top_batsman_score.batsman,y=top_batsman_score.
 ↪hundreds,name="Highest Hundreds"),row=2,col=2)
fig.add_trace(go.Bar(x=top_batsman_score.batsman,y=top_batsman_score.
 ↪heighest_score,name="Highest Score"),row=3,col=1)
fig.add_trace(go.Bar(x=top_batsman_score.batsman,y=top_batsman_score.
 ↪total_runs,name="Total Runs"),row=3,col=2)
fig.update_layout(height=800,width=1000, title_text="<b>Top 10 Batsmen␣
 ↪Performance</b>",
                  title_font_family="Times New Roman",title_font_color="red",
                  title_font_size=20,)
fig.show()
```

This code creates a bar chart using Plotly to visualize the top 10 six-hitters in IPL. It starts by extracting the rows from the `ball_data` DataFrame where `batsman_run` is equal to 6, which indicates that a six was hit by the batsman. It then groups these rows by `batter`, counts the number of occurrences of 6 for each batsman, sorts the result in descending order, and selects the top 10 batsmen with the highest number of sixes hit.

The resulting `highest_sixers` DataFrame is then used to create a bar chart using Plotly's `px.bar` function. The x-axis of the chart represents the top 10 batsmen, while the y-axis represents the number of sixes hit by each batsman. The color of the bars is also set to represent the number of sixes hit by each batsman, with darker colors indicating a higher number of sixes. Finally, the chart is styled using various layout options, including font family, title, axis labels, and legend. The resulting chart shows the top 10 six-hitters in IPL and their respective number of sixes hit.

[15]:
```
highest_sixers = ball_data[ball_data['batsman_run']==6].groupby('batter').
 ↪agg(six_count=('batsman_run', 'count'))\
      .reset_index().sort_values(by='six_count', ascending=False).head(10)

fig = go.Figure(data=px.bar(x=highest_sixers.batter,
                            y=highest_sixers.six_count,
                            color = highest_sixers.six_count,
                            color_discrete_sequence=px.colors.sequential.
 ↪Oranges,
                            title='<b>Top 10 Six-hitters in IPL</b>',
                            text = highest_sixers.six_count,
                            height=400))

fig.update_layout(
    font_family="Courier New",
    title_font_family="Times New Roman",
    title_font_color="red",
    title_font_size=20,
    xaxis_title="<b>Batsman</b>",
    yaxis_title="<b>No. of Sixers</b>",
```

```
    legend_title_font_color="green"
)

fig.show()
```

The code is visualizing the data related to the top 10 batsmen in the Indian Premier League (IPL) using bar charts.

The first code block creates a subplot consisting of 6 different bar charts. Each chart shows the performance of the top 10 batsmen in a specific category such as "No. of Sixers", "No. of Fours", "No. of Half-Centuries", "No. of Centuries", "Highest Scores", and "Total Runs". It first defines the subplot layout with 2 columns and 3 rows using `make_subplots`. It then creates 6 different `go.Bar` trace objects for each category of performance and adds them to the subplot using `fig.add_trace`. The subplot titles are also specified using the `subplot_titles` parameter. The resulting plot is displayed using `fig.show()`.

The second code block visualizes the data related to the top 10 batsmen in the IPL who hit the most sixes. It first filters the ball_data to include only the rows where the batsman scored 6 runs and then groups the data by batsman to count the number of sixes they hit. The resulting data is sorted in descending order of six count and the top 10 batsmen are selected. Then, a `px.bar` plot is created with the x-axis representing the top 10 batsmen, the y-axis representing the number of sixes hit, and the color representing the number of sixes hit. The plot is customized using various parameters such as `color_discrete_sequence` for choosing a specific color sequence for the bars, `text` for displaying the number of sixes hit on top of each bar, and `update_layout` for customizing the plot layout. Finally, the plot is displayed using `fig.show()`.

The third code block is similar to the second block, except that it visualizes the data related to the top 10 batsmen who hit the most fours in the IPL. It filters the ball_data to include only the rows where the batsman scored 4 runs, groups the data by batsman to count the number of fours they hit, sorts the resulting data in descending order of four count, and selects the top 10 batsmen. It then creates a `px.bar` plot with the x-axis representing the top 10 batsmen, the y-axis representing the number of fours hit, and the color representing the number of fours hit. The plot is customized using various parameters such as `color_discrete_sequence` for choosing a specific color sequence for the bars, `text` for displaying the number of fours hit on top of each bar, and `update_layout` for customizing the plot layout. Finally, the plot is displayed using `fig.show()`.

[16]:
```
highest_fours = ball_data[ball_data['batsman_run']==4].groupby('batter').
 ↪agg(four_count=('batsman_run', 'count'))\
      .reset_index().sort_values(by='four_count', ascending=False).head(10)
fig = go.Figure(data=px.bar(x=highest_fours.batter,
                            y=highest_fours.four_count,
                            color = highest_fours.four_count,
                            color_discrete_sequence=px.colors.sequential.
 ↪Oranges,
                            title='<b>Top 10 Four-hitters in IPL</b>',
                            text = highest_fours.four_count,
                            height=400))
fig.update_layout(
    font_family="Courier New",
```

```
    title_font_family="Times New Roman",
    title_font_color="red",
    title_font_size=20,
    xaxis_title="<b>Batsman</b>",
    yaxis_title="<b>No. of Fours</b>",
    legend_title_font_color="green"
)

fig.show()
```

The code is using Python's Matplotlib library to create a pie chart to show the distribution of the number of sixers hit by each team in the IPL 2022 season.

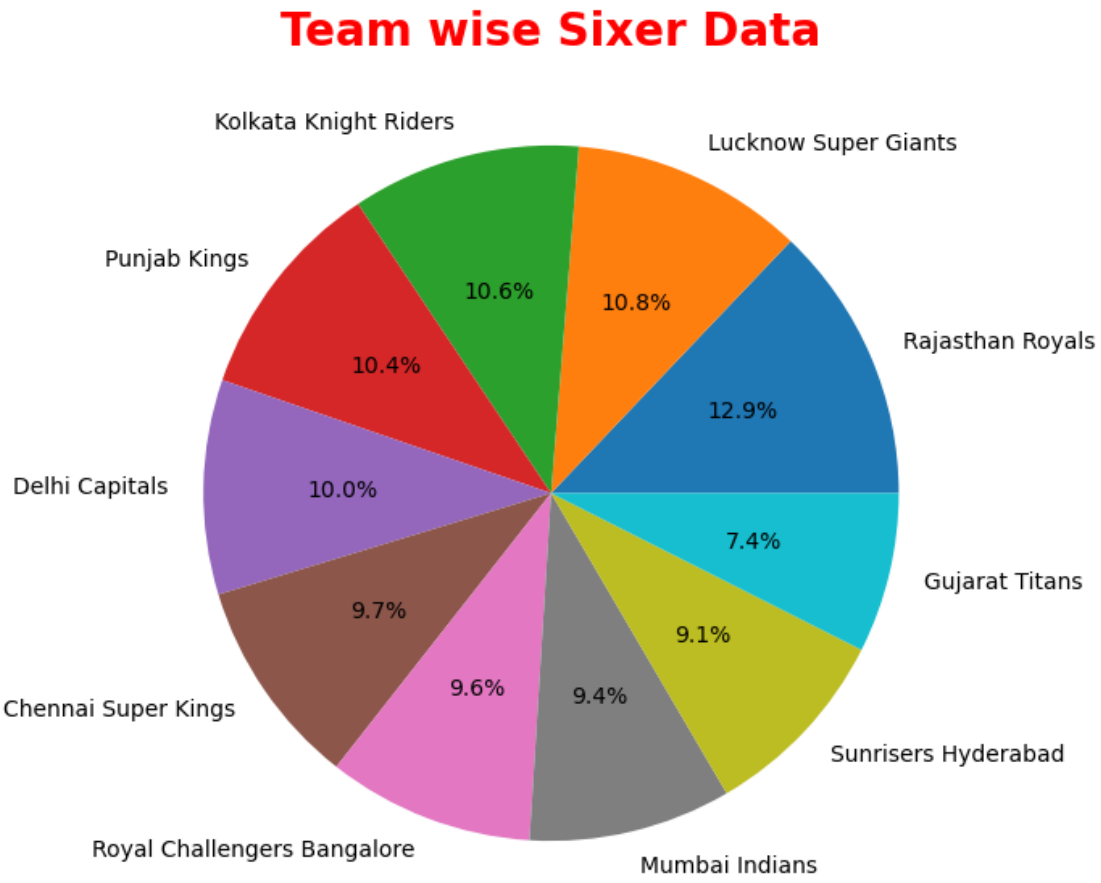
First, the code filters the `ball_data` dataframe to include only the rows where the batsman hit a six and the season is 2022. It then groups the data by the batting team and counts the number of sixers hit by each team. The resulting dataframe, `sixer_df`, is sorted in descending order by the number of sixers.

The pie chart is then created using the `plt.pie()` method. The data for the chart is the `sixer_count` column of the `sixer_df`dataframe, and the labels for each slice are the `battingteam` column. The autopct parameter is used to display the percentage of each slice on the chart.

Finally, the title of the chart is set using `plt.title()`. The chart is displayed using `plt.show()`.

[17]:
```
sixer_df=ball_data['battingteam'][(ball_data['batsman_run']==6) &␣
 ↪(ball_data['season']=='2022')].reset_index()
sixer_df=sixer_df.groupby('battingteam').
 ↪agg(sixer_count=('battingteam','count')).reset_index()
sixer_df=sixer_df.sort_values('sixer_count', ascending=False)

fig = plt.figure(figsize =(10, 7))
plt.pie(sixer_df['sixer_count'], labels = sixer_df['battingteam'], autopct='%0.
 ↪1f%%')
plt.title("Team wise Sixer Data", fontsize=20, color='red', fontweight='bold')
plt.show()
```



This code calculates the total runs scored by the top batsmen against each of the opponent teams. It first selects the relevant columns - `'batter', 'bowlingteam', and 'batsman_run'` from the ball_data dataframe for the top batsmen using the `isin()` method.

Then it groups the resulting dataframe by 'batter' and 'bowlingteam' columns and calculates the total runs scored by each batsman against each opponent team using the sum() method. The resulting dataframe is stored in df1.

Next, it uses the pivot_table() method to reshape the data so that the rows represent the opponent teams and the columns represent the top batsmen. Any missing values are filled with 0 using the fillna() method.

Finally, it applies a color gradient to the resulting dataframe using the background_gradient() method and prints the output to the console with a green colored header using the Back and Style modules from the colorama package.

[18]:
```
df1 = ball_data[['batter','bowlingteam', 'batsman_run']][ball_data['batter'].
↪isin(top_batsman['batter'])]\
```

```
            .groupby(['batter', 'bowlingteam']).
 ↪agg(batter_score=('batsman_run', 'sum')).reset_index()

print(Back.BLUE+ Style.BRIGHT+'Runs scored by top batsmen against opponent␣
 ↪teams'+ Style.RESET_ALL)
df2 = df1.pivot_table('batter_score', ['bowlingteam'], 'batter')
df2 = df2.fillna(0)
df2.style.background_gradient(axis=None, low=0.75, high=1.0)
```

```
Runs scored by top batsmen against opponent teams
```

[18]:
```
<pandas.io.formats.style.Styler at 0x1756a930e50>
```

This code analyzes the performance of the top batsmen in IPL across different seasons.

First, a list `top_batsman` is defined which contains the names of the top batsmen. Then, the code filters the ball_data dataframe to only include the rows where the batter's name is present in the `top_batsman` list. It then groups the filtered data by `season` and `batter` columns and calculates the total runs scored by each batter in each season using the `agg` function with the `sum` method. This result is stored in the `top_batsman_runs` dataframe, which is sorted by season and runs scored in descending order.

Then, the `px.line` function from the plotly.express library is used to create a line plot with the `season` on the x-axis and the `batsman_total` on the y-axis, which represents the runs scored by the batsman in each season. The plot is grouped by the `batter` column and different batters are distinguished by the color and symbol of the lines.

The plot is customized using various functions such as `update_layout`, `update_xaxes`, `update_yaxes`, and `update_traces` to modify the title, font family, font color, axis titles, legend title, legend font color, plot background color, and line width of the plot. The `hovermode` parameter is set to `"x unified"` to show the hover information for all the batters at once for each season. Finally, the plot is displayed using the `show()` method.

[19]:
```
top_batsman =['RG Sharma', 'S Dhawan', 'SK Raina', 'V Kohli']
top_batsman_runs=ball_data[ball_data['batter'].isin(top_batsman)].
 ↪groupby(['season','batter'])\
    .agg(batsman_total=('batsman_run', 'sum')).reset_index()\
    .sort_values(by=['season','batsman_total'], ascending=False)

fig = px.line(top_batsman_runs, x='season', y='batsman_total', color='batter',␣
 ↪symbol="batter",template='simple_white')
fig.update_layout(hovermode="x unified")
fig.update_xaxes(categoryorder='category ascending')
fig.update_yaxes(showgrid=True)
fig.update_traces(line=dict(width=3.0))

fig.update_layout(
    title='<b>Batter Performance across seasons</b>',
    font_family="Courier New",
```

```
    title_font_family="Courier New",
    title_font_color="red",
    title_font_size=20,
    xaxis_title="<b>Season</b>",
    yaxis_title="<b>Runs</b>",
    legend_title='<b>Batter</b>',
    legend_title_font_color="red",
    plot_bgcolor = '#FFFFFF'
)

fig.show()
```

This code segment prints the top 10 wicket takers in the Indian Premier League (IPL).

- First, it creates a list of dismissal types which includes 'caught', 'caught and bowled', 'bowled', 'stumped', and 'hit wicket'. These are the ways in which a bowler can take a wicket in cricket.
- Then, it filters the ball_data dataframe to include only the rows where the column 'is_wicket' is equal to 1 (which means a wicket was taken) and the 'dismissal_kind' column contains any of the dismissal types from the dismissal_list.
- Next, it groups the resulting dataframe by the 'bowler' column and calculates the count of wickets taken by each bowler using the 'count' function.
- Then, it sorts the resulting dataframe by the 'wicket_count' column in descending order and selects the top 10 rows.
- Finally, it applies a blue color gradient to the dataframe using the 'background_gradient' function from pandas styler and prints the resulting dataframe.

[20]:
```
print(Back.BLUE+ Style.BRIGHT+'Top 10 wicket takers'+ Style.RESET_ALL)
dismissal_list = ['caught','caught and bowled','bowled','stumped','hit wicket']
top10_bowlers=ball_data[(ball_data['is_wicket']==1) &␣
 ↪(ball_data['dismissal_kind'].isin(dismissal_list))]\
                            .groupby('bowler').agg(wicket_count=('is_wicket',␣
 ↪'count'))\
                            .sort_values('wicket_count', ascending=False).
 ↪reset_index().head(10)
top10_bowlers.style.background_gradient("Blues")
```

```
Top 10 wicket takers
```

[20]:
```
<pandas.io.formats.style.Styler at 0x1756b70b760>
```

This code is selecting the performance of the top 10 bowlers in terms of the number of wickets taken in the IPL. The code uses the `ball_data` dataframe and filters the rows where `is_wicket` is 1 and `dismissal_kind` is in the list of `dismissal_list` which contains the types of dismissals that are counted as wickets. The code then groups the filtered dataframe by the `bowler` column and calculates the count of wickets for each bowler. The result is sorted in descending order by the `wicket_count` and only the top 10 bowlers are selected.

The next part of the code calculates the wicket count of each bowler against the different opponent teams. This is done by filtering the rows where `bowler` is one of the top 10 bowlers and grouping

the filtered dataframe by the `bowler` and `battingteam` columns. The `wkt_count` is then calculated as the count of `is_wicket` column for each group. The result is then pivoted to create a new dataframe where the `battingteam` is the index and the `bowler` names are the columns, and the `wkt_count` is the value in the cells. The `fillna()` method fills the missing values with 0. The resulting dataframe is styled with a blue color gradient.

[21]:
```
print(Back.BLUE+ Style.BRIGHT+'Bowler performance(wicket count) against␣
 ↪opponent teams'+ Style.RESET_ALL)
bowler_team = ball_data[(ball_data['is_wicket']==1) &␣
 ↪(ball_data['dismissal_kind'].isin(dismissal_list))\
                            & (ball_data['bowler'].
 ↪isin(top10_bowlers['bowler']))]\
                            .groupby(['bowler','battingteam']).
 ↪agg(wkt_count=('is_wicket', 'count'))\
                            .reset_index()

bowler_team = bowler_team.pivot_table('wkt_count', ['battingteam'], 'bowler')
bowler_team = bowler_team.fillna(0)
bowler_team.style.background_gradient("Blues")
```

```
Bowler performance(wicket count) against opponent teams
```

[21]:
```
<pandas.io.formats.style.Styler at 0x1756beea760>
```

This code creates a bar chart showing the number of hattricks taken by each bowler in the Indian Premier League (IPL).

The first few lines of code create a DataFrame `wkt_bowlers` that lists the number of wickets taken by each bowler in each match. Then, the code checks for hattricks by iterating over each ball in the `ball_data` DataFrame and counting the number of consecutive wickets taken by each bowler. Whenever a bowler takes three wickets in a row, the code adds their name to a DataFrame `hattricks`. Finally, the code groups the `hattricks` DataFrame by bowler and counts the number of hattricks taken by each bowler, creating a new DataFrame `hattrick_count`.

The remaining code uses `hattrick_count` DataFrame to create a bar chart using Plotly Express. The chart has bowlers on the x-axis, and the number of hattricks taken by each bowler on the y-axis. The color of each bar represents the bowler, and the height of the bar represents the number of hattricks taken by the bowler. The chart is titled "Hattricks in IPL", and the font, colors, and legend are all customized using Plotly's layout options.

[22]:
```
wkt_bowlers= ball_data[ball_data['is_wicket']==1].groupby(['id','bowler']).
 ↪agg(wkt_count=('is_wicket', 'count')).reset_index()
hattricks = pd.DataFrame(columns= ['bowler'])
for bow_index, bow_row in␣
 ↪wkt_bowlers[wkt_bowlers['wkt_count']>=3][['id','bowler']].iterrows():
    wkt_cnt=0
    for ball_index, ball_row in ball_data[(ball_data['id']==bow_row['id']) &
                                          ␣
 ↪(ball_data['bowler']==bow_row['bowler'])]\
```

```
                                                .sort_values(by␣
 ↪=(['id','inning','over','ballnumber'])).iterrows():

        if ball_row['is_wicket']==1 and ball_row['dismissal_kind'] not in ['run␣
 ↪out','retired hurt','obstructing the field'] :
            wkt_cnt +=1
            if wkt_cnt == 3:
                wkt_cnt = 0
                hattricks = hattricks.append({'bowler' : ball_row['bowler']},␣
 ↪ignore_index = True)
        else:wkt_cnt =0

hattricks = hattricks.groupby('bowler').agg(hatk_count=('bowler', 'count')).
 ↪reset_index()
fig = go.Figure(data=px.bar(x=hattricks.bowler,
                            y=hattricks.hatk_count,
                            color = hattricks.bowler,
                            color_discrete_sequence=px.colors.sequential.Plasma,
                            title='<b>Hattricks in IPL</b>',
                            text = hattricks.hatk_count,
                            height=500))
fig.update_layout(
    font_family="Courier New",
    title_font_family="Times New Roman",
    title_font_color="red",
    title_font_size=20,
    xaxis_title="<b>Bowler</b>",
    yaxis_title="<b>Hattricks</b>",
    legend_title_font_color="green"
)

fig.show()
```

```
C:\Users\suman\AppData\Local\Temp\ipykernel_16608\245980854.py:13:
FutureWarning:

The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.

C:\Users\suman\AppData\Local\Temp\ipykernel_16608\245980854.py:13:
FutureWarning:

The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.

C:\Users\suman\AppData\Local\Temp\ipykernel_16608\245980854.py:13:
FutureWarning:
```

```
The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.

C:\Users\suman\AppData\Local\Temp\ipykernel_16608\245980854.py:13:
FutureWarning:

The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.

C:\Users\suman\AppData\Local\Temp\ipykernel_16608\245980854.py:13:
FutureWarning:

The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.

C:\Users\suman\AppData\Local\Temp\ipykernel_16608\245980854.py:13:
FutureWarning:

The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.

C:\Users\suman\AppData\Local\Temp\ipykernel_16608\245980854.py:13:
FutureWarning:

The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.

C:\Users\suman\AppData\Local\Temp\ipykernel_16608\245980854.py:13:
FutureWarning:

The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.

C:\Users\suman\AppData\Local\Temp\ipykernel_16608\245980854.py:13:
FutureWarning:

The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.

C:\Users\suman\AppData\Local\Temp\ipykernel_16608\245980854.py:13:
FutureWarning:

The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.

C:\Users\suman\AppData\Local\Temp\ipykernel_16608\245980854.py:13:
FutureWarning:
```

```
The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.

C:\Users\suman\AppData\Local\Temp\ipykernel_16608\245980854.py:13:
FutureWarning:

The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.

C:\Users\suman\AppData\Local\Temp\ipykernel_16608\245980854.py:13:
FutureWarning:

The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.

C:\Users\suman\AppData\Local\Temp\ipykernel_16608\245980854.py:13:
FutureWarning:

The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.

C:\Users\suman\AppData\Local\Temp\ipykernel_16608\245980854.py:13:
FutureWarning:

The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.

C:\Users\suman\AppData\Local\Temp\ipykernel_16608\245980854.py:13:
FutureWarning:

The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.

C:\Users\suman\AppData\Local\Temp\ipykernel_16608\245980854.py:13:
FutureWarning:

The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.

C:\Users\suman\AppData\Local\Temp\ipykernel_16608\245980854.py:13:
FutureWarning:

The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.

C:\Users\suman\AppData\Local\Temp\ipykernel_16608\245980854.py:13:
FutureWarning:
```

```
The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.

C:\Users\suman\AppData\Local\Temp\ipykernel_16608\245980854.py:13:
FutureWarning:

The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.

C:\Users\suman\AppData\Local\Temp\ipykernel_16608\245980854.py:13:
FutureWarning:

The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.
```

The code plots a bar graph of top player-of-the-match (POM) winners for each season. The `match_data` DataFrame is grouped by season and player_of_match to get the count of how many times each player won the POM award in each season. Then, a new column 'Rank' is added to the DataFrame by ranking the POM counts in descending order within each season group. Next, a subset of the DataFrame is created by filtering only the rows where the Rank is equal to 1 (i.e., top POM winners for each season). Finally, a bar graph is plotted using the Plotly Express library, where the x-axis represents the seasons, y-axis represents the number of POM awards won, and the color of each bar represents the player who won the award.

The graph should provide insights into which players were consistently the top performers in each season of the IPL.

[23]:
```
pom_df=match_data.groupby(['season','player_of_match']).
 ↪agg(pom_count=('player_of_match','count')).reset_index()
pom_df


pom_df['Rank'] = pom_df.groupby('season')['pom_count'].rank(ascending=False)
season_top_pom=pom_df[pom_df['Rank']==1].sort_values('season')

fig = go.Figure(data=px.bar(x=season_top_pom.season,
                            y=season_top_pom.pom_count,
                            color = season_top_pom.player_of_match,
                            color_discrete_sequence=colorsy,
                            title='<b>Top 10 Player-of-the-Match winners in␣
 ↪IPL</b>',
                            text = season_top_pom.pom_count,
                            height=500))

fig.update_layout(
    font_family="Courier New",
```

```
    title_font_family="Times New Roman",
    title_font_color="red",
    title_font_size=20,
    xaxis_title="<b>Season</b>",
    yaxis_title="<b>No. of PoMs</b>",
    legend_title_font_color="green"
)

fig.show()
```

This code block first merges the `match_data` and `ball_data` dataframes into `full_data` based on the common `id` column.

Then, it extracts data on the total score of each inning in each match from `full_data` and groups it by `id` and `inning` using the `groupby` method. It computes the sum of `total_runs` for each combination of `id` and `inning` and stores it in a new column called `inning_score`. It saves this dataframe as `inning_data`.

Then, the code merges `match_data` and `inning_data` on the `id` column to create a new dataframe called `match_scores`. This dataframe contains information about the total score of each team in each match.

Finally, the code identifies the top 5 high scores successfully chased and top 5 high scores successfully defended by filtering `match_scores` based on the values in the `wonby` column. It then prints out the details of these matches, including the teams involved, the season, and the target score.

[24]:
```
full_data = pd.merge(match_data, ball_data, how='left', on='id')

inning_data = full_data[['id', 'inning', 'total_runs']]\
                        [(full_data['superover']=='N') & (full_data['method'] !
 ↪='D/L')]\
                        .groupby(['id', 'inning']).
 ↪agg(inning_score=('total_runs','sum')).reset_index()
inning_data = inning_data.pivot_table('inning_score', ['id'], 'inning').
 ↪reset_index()
inning_data.rename(columns = {1:'first_inning', 2:'second_inning'},␣
 ↪inplace=True)

match_scores = pd.merge(match_data, inning_data, how='left', on='id')

chased_matches=match_scores[match_scores['wonby']=='Wickets'].
 ↪sort_values('first_inning', ascending=False)
defended_matches=match_scores[match_scores['wonby']=='Runs'].
 ↪sort_values('first_inning', ascending=True)

print(Back.BLUE+ Style.BRIGHT+ 'Top 5 High scores successfully chased:' + Style.
 ↪RESET_ALL)
for idx, row in chased_matches.head().iterrows():
```

```
    print(Fore.RED+ Style.BRIGHT+  row['team2'] + Style.RESET_ALL \
      + ' sucessfully chased target of ' \
      + Back.CYAN+ Style.BRIGHT+ Style.BRIGHT + str(row['first_inning']) +␣
 ↪Style.RESET_ALL \
      + ' against '\
      + Fore.RED + Style.BRIGHT+ row['team1'] + Style.RESET_ALL
      + 'in season '
      + Fore.BLUE + Style.BRIGHT+ row['season'] + Style.RESET_ALL )
```

```
Top 5 High scores successfully chased:
Rajasthan Royals sucessfully chased target of
223.0 against Kings XI Punjabin season
2020/21
Mumbai Indians sucessfully chased target of 218.0
against Chennai Super Kingsin season 2021
Rajasthan Royals sucessfully chased target of
214.0 against Deccan Chargersin season
2007/08
Lucknow Super Giants sucessfully chased target of
210.0 against Chennai Super Kingsin season
2022
Gujarat Lions sucessfully chased target of 208.0
against Delhi Daredevilsin season 2017
```

This code prints the top 5 low scores that were successfully defended in the IPL matches.

It first creates a subset of the `match_scores` DataFrame by selecting only the matches where the winning team won by "Runs". It then sorts these matches in ascending order of the first inning score.

It then iterates through the top 5 rows of this subset and prints the name of the team that successfully defended the score, the target score, the name of the team that was defending, and the season in which this match was played. The output is colored using the `Fore`, `Back`, and `Style` classes from the `colorama` library to make it more visually appealing.

[25]:
```
print(Back.BLUE+ Style.BRIGHT+ 'Top 5 Low scores successfully defended:' +␣
 ↪Style.RESET_ALL)
for idx, row in defended_matches.head().iterrows():
    print(Fore.RED+ Style.BRIGHT+  row['team1'] + Style.RESET_ALL \
      + ' sucessfully defended target of ' \
      + Back.CYAN + Style.BRIGHT+ str(row['first_inning']) + Style.RESET_ALL \
      + ' against '\
      + Fore.RED + Style.BRIGHT+ row['team2'] + Style.RESET_ALL
      + 'in season '
      + Fore.BLUE + Style.BRIGHT+ row['season'] + Style.RESET_ALL )
```

```
Top 5 Low scores successfully defended:
Royal Challengers Bangalore sucessfully defended target of
106.0 against Chennai Super Kingsin season
```

```
2013
Kings XI Punjab sucessfully defended target of 106.0
against Royal Challengers Bangalorein season 2015
Chennai Super Kings sucessfully defended target of
116.0 against Kings XI Punjabin season
2009
Sunrisers Hyderabad sucessfully defended target of
118.0 against Mumbai Indiansin season
2018
Kings XI Punjab sucessfully defended target of 119.0
against Mumbai Indiansin season 2009
```

This code prints the details of the match where the winning team had the highest run margin. It first filters the rows in `match_data` dataframe where `margin` is equal to the maximum value of the margin column. Then, for each of the resulting rows, it prints the name of the winning team, the margin of victory, the name of the team they won against, and the season in which the match was played. It formats the output using ANSI escape codes to highlight the important information in red and cyan colors.

[26]:
```
print(Back.BLUE+ Style.BRIGHT+ 'Match win with highest run margin:' + Style.
↪RESET_ALL)
for idx, row in match_data[['winningteam','win_against', 'margin', 'season']]\
    [match_data['margin'] == match_data['margin'].max()].iterrows():
    print(Fore.RED+ Style.BRIGHT+  row['winningteam'] + Style.RESET_ALL \
      + ' won with highest ever margin of ' \
      + Back.CYAN + Style.BRIGHT+ str(row['margin']) + Style.RESET_ALL \
      + ' against '\
      + Fore.RED + Style.BRIGHT+ row['win_against'] + Style.RESET_ALL
      + 'in season '
      + Fore.BLUE + Style.BRIGHT+ row['season'] + Style.RESET_ALL )
```

```
Match win with highest run margin:
Mumbai Indians won with highest ever margin of 146.0
against Delhi Daredevilsin season 2017
```

This code block prints the highest win margin for each season in the IPL. It does this by grouping the matches by season and finding the maximum margin of victory for each season. Then, for each season, it searches the match data to find the match(es) with that maximum margin of victory and prints out the winning team, margin, and opponent for each of those matches.

The output will be in the following format:

where `<winning team>` is the name of the team that won the match, `<margin>` is the margin of victory in runs, and `<opponent>` is the name of the team that lost the match. `<season>` is the season number of the IPL.

Note that there could be ties for the highest win margin in a season, so there could be multiple lines printed for the same season.

[27]:
```
print(Back.BLUE+ Style.BRIGHT+'Highest win margin in each season:'+ Style.
 ↪RESET_ALL)
margin_df= match_data.groupby('season').agg(max_margin=('margin', 'max')).
 ↪reset_index()
for idx1, row1 in margin_df[[ 'season', 'max_margin']].iterrows():
    for idx, row in match_data[['winningteam','win_against', 'margin',␣
 ↪'season']]\
            [(match_data['margin'] == row1['max_margin']) &␣
 ↪(match_data['season'] == row1['season'])]\
            .iterrows():
        print(Fore.RED+ Style.BRIGHT+  row['winningteam'] + Style.RESET_ALL \
          + ' won with highest margin of ' \
          + Back.CYAN + Style.BRIGHT+ str(row['margin']) + Style.RESET_ALL \
          + ' against '\
          + Fore.RED + Style.BRIGHT+ row['win_against'] + Style.RESET_ALL
          + 'in season '
          + Fore.BLUE + Style.BRIGHT +row['season'] + Style.RESET_ALL )
```

```
Highest win margin in each season:
Kolkata Knight Riders won with highest margin of 140.0
against Royal Challengers Bangalorein season 2007/08
Mumbai Indians won with highest margin of 92.0 against
Kolkata Knight Ridersin season 2009
Chennai Super Kings won with highest margin of 92.0
against Royal Challengers Bangalorein season 2009
Mumbai Indians won with highest margin of 98.0 against
Delhi Daredevilsin season 2009/10
Kings XI Punjab won with highest margin of 111.0
against Royal Challengers Bangalorein season 2011
Chennai Super Kings won with highest margin of 86.0
against Delhi Daredevilsin season 2012
Royal Challengers Bangalore won with highest margin of
130.0 against Pune Warriorsin season 2013
Chennai Super Kings won with highest margin of 93.0
against Delhi Daredevilsin season 2014
Royal Challengers Bangalore won with highest margin of
138.0 against Kings XI Punjabin season
2015
Royal Challengers Bangalore won with highest margin of
144.0 against Gujarat Lionsin season 2016
Mumbai Indians won with highest margin of 146.0
against Delhi Daredevilsin season 2017
Mumbai Indians won with highest margin of 102.0
against Kolkata Knight Ridersin season 2018
Sunrisers Hyderabad won with highest margin of 118.0
against Royal Challengers Bangalorein season 2019
Kings XI Punjab won with highest margin of 97.0
```

```
against Royal Challengers Bangalorein season 2020/21
Kolkata Knight Riders won with highest margin of 86.0
against Rajasthan Royalsin season 2021
Chennai Super Kings won with highest margin of 91.0
against Delhi Capitalsin season 2022
```

This code displays the head-to-head win count of each team against every other team in the given dataset.

It first creates a dataframe `df1` by grouping the `winningteam` and `win_against` columns of the `match_data` dataframe and counting the number of times each team won against the other.

Then, it creates a pivot table `df2` from `df1`, where the index is the `winningteam` column, columns are the `win_against` column, and values are the `win_count` column. The resulting pivot table shows the head-to-head win count of each team against every other team.

Finally, it applies a background gradient to the pivot table to make it visually more appealing.

[28]:
```
print(Back.BLUE+ Style.BRIGHT+'Head on head encounters : Team vs Team (win␣
 ↪count on each other)'+ Style.RESET_ALL)
df1=match_data[['winningteam','win_against']].
 ↪groupby(['winningteam','win_against']).agg(win_count=('win_against','count'))
df2=df1.pivot_table('win_count', ['winningteam'], 'win_against')
df2 = df2.fillna(0)
df2.style.background_gradient("plasma")
```

```
Head on head encounters : Team vs Team (win count on each other)
```

[28]:
```
<pandas.io.formats.style.Styler at 0x1756b70b640>
```

This code generates a bar chart showing the count of matches played at leading venues.

The code first groups the `match_data` dataframe by the `venue` column and uses the `agg()` method to count the number of matches played at each venue. It then sorts the resulting dataframe in descending order based on the `match_count` column.

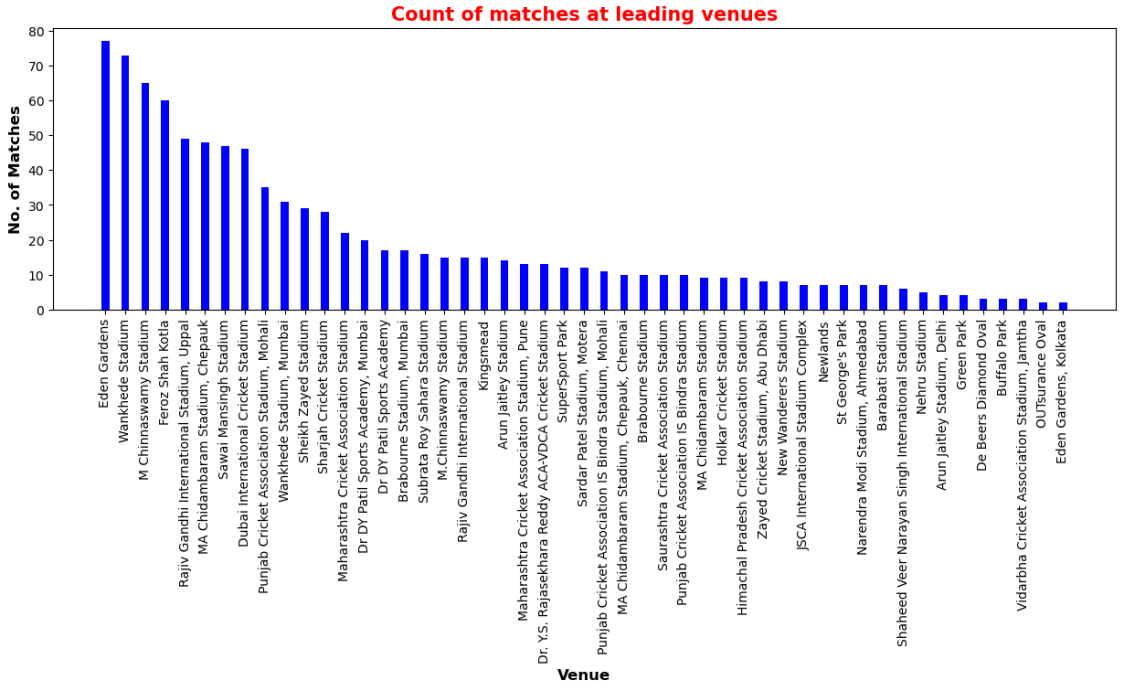
After that, the code creates a new figure with a width of 15 and a height of 4 using `plt.figure()`. It then creates a bar chart using the `plt.bar()` function, with the `x` values being the `venue` column from the `venue_df` dataframe and the `y` values being the `match_count` column from the same dataframe. The bars are colored blue and have a width of 0.4.

The x-axis label is set to "Venue" using `plt.xlabel()`, and the y-axis label is set to "No. of Matches" using `plt.ylabel()`. The `plt.xticks()` function is used to rotate the x-tick labels by 90 degrees, and the title of the chart is set to "Count of matches at leading venues" using `plt.title()`. Finally, the chart is displayed using `plt.show()`.

[29]:
```
venue_df=match_data.groupby('venue').agg(match_count=('venue', 'count')).
 ↪reset_index().sort_values('match_count', ascending=False)

f = plt.figure()
f.set_figwidth(15)
```

```
f.set_figheight(4)
plt.bar(venue_df['venue'], venue_df['match_count'], color ='blue',width = 0.4)
plt.xlabel("Venue", fontsize=12, fontweight='bold')
plt.ylabel("No. of Matches", fontsize=12, fontweight='bold')
plt.xticks(rotation=90)
plt.title("Count of matches at leading venues", fontsize=15, color='red',␣
 ↪fontweight='bold')
plt.show()
```



[ ]: