Criterion C: Development

A list of most significant techniques used in the development process:

1. Working with FPL API and extracting official data
2. Data management and storage of extracted statistics
3. Determining the optimum team based on financial, positional, and club constraints
4. Creating a dynamic GUI based on buttons/choices clicked
5. Displaying optimum team (working with background image and grid placing)
6. Modifying a table view based on user's selection of options

Working with FPL API and extracting official data

The FPL API (Application Programming Interface) provides a comprehensive overview of the FPL game through various endpoints. As shown in Fig. 1, I initiated by setting the base URL for FPL API, sent an HTTP GET request to the 'bootstrap-static' endpoint using **requests.get()** and fetched static information for players, teams, and global game week summaries. Following this, a response was received in JSON format, which was then converted into a Python dictionary using the **.json()** method.[1] (Leslie, 2021)

```
# base url for all FPL API endpoints
base_url = 'https://fantasy.premierleague.com/api/'

# get data from bootstrap-static endpoint
r = requests.get(base_url+'bootstrap-static/').json()
```

Figure 1: Requesting 'bootstrap-static' endpoint of FPL API

As seen in Fig. 2, using Pandas' **.json_normalize()** function, 'elements' field of the API response, which contains data for each premier league player in the current season of FPL, is flattened into a DataFrame called *players*, allowing for representation of the data in a more structured, tabular manner. Like the players, DataFrames called *teams* and *positions* are then created by flattening the 'teams' and 'element_types' fields of the API response. The **.head()** function is simply used to visualize the first few rows of the DataFrames.

```
# create players dataframe
players = pd.json_normalize(r['elements'])

# show some information about first five players
players[['id', 'web_name', 'team', 'element_type']].head()

# create teams dataframe
teams = pd.json_normalize(r['teams'])

teams.head()

# get position information from 'element_types' field
positions = pd.json_normalize(r['element_types'])

positions.head()
```

Figure 2: Changing JSON data structure into Pandas DataFrames and viewing the result

[1] https://medium.com/analytics-vidhya/getting-started-with-fantasy-premier-league-data-56d3b9be8c32

The *players* and *teams* DataFrames were now merged based on the 'team' and 'id' columns, respectively. As shown in Fig. 3, this creates a new DataFrame *df* with combined information about players and their respective teams. Similarly, *df* was also merged with *positions* based on the 'element_type' and 'id' columns, respectively. Here, I used two different ways to use the **.merge()** function: firstly, as a static function, **pd.merge()**, and then on an existing DataFrame, **df.merge()**. Finally, column names are refined using the rename function to make them more meaningful.

```
# join players to teams
df = pd.merge(left=players, right=teams, left_on='team', right_on='id')

# join player positions
df = df.merge(positions, left_on='element_type', right_on='id')

# rename columns
df = df.rename(columns={'name':'team_name', 'singular_name':'position_name'})
```

Figure 3: Merging the different DataFrames to create a singular data storing structure

Data management and working with FPL statistics

As seen in Fig. 4, we need to store the data extracted from the official FPL database into a CSV file for better management. To do so, I used Python's **csv.writer()** method and then made an empty list, which was updated with season statistics for each player. This list was then written on a row of the CSV file. As a result, every player's essential data is stored in a single row, separated by commas, as shown in Fig. 5. Another CSV file is used to store the basic information of a player, in a similar way, as shown in Fig. 7.

```
#Storing player statistics extracted with FPL API into a CSV file
with open('Data_analysis/PlayerSeasonData.csv', 'w') as f:
    writer = csv.writer(f)
    for i in range(0, 726):
        content_list=[]
        content_list.append(df.iloc[i]["first_name"] + " " + df.iloc[i]["second_name"])
        content_list.append(df.iloc[i]["minutes"])
        content_list.append(df.iloc[i]["goals_scored"])
        content_list.append(df.iloc[i]["assists"])
        content_list.append(df.iloc[i]["clean_sheets"])
        content_list.append(df.iloc[i]["goals_conceded"])
        content_list.append(df.iloc[i]["own_goals"])
        content_list.append(df.iloc[i]["penalties_saved"])
        content_list.append(df.iloc[i]["penalties_missed"])
        content_list.append(df.iloc[i]["yellow_cards"])
        content_list.append(df.iloc[i]["red_cards"])
        content_list.append(df.iloc[i]["saves"])
        content_list.append(df.iloc[i]["starts"])
        content_list.append(df.iloc[i]["expected_goals"])
        content_list.append(df.iloc[i]["expected_assists"])
        content_list.append(df.iloc[i]["expected_goal_involvements"])
        content_list.append(df.iloc[i]["expected_goals_conceded"])
        content_list.append(df.iloc[i]["total_points"])
        content_list.append(df.iloc[i]["now_cost"])
        content_list.append(df.iloc[i]["cost_change_start"])
        content_list.append(df.iloc[i]["selected_by_percent"])
        content_list.append(df.iloc[i]["web_name"])
        writer.writerow(content_list)
```

Figure 4: Saving player information about season statistics extracted using FPL API into a CSV file

```
Folarin Balogun,0,0,0,0,0,0,0,0,0,0,0.00,0.00,0.00,0.00,0,44,-1,0.2,Balogun
Gabriel Fernando de Jesus,1091,4,4,7,11,0,0,0,5,0,0,13,5.30,2.01,7.31,9.47,63,80,0,4.3,G.Jesus
Eddie Nketiah,979,5,4,6,9,0,0,0,3,0,0,10,4.19,0.47,4.66,9.54,65,52,-3,4.1,Nketiah
Keinan Davis,0,0,0,0,0,0,0,0,0,0,0,0.00,0.00,0.00,0.00,0,49,-1,0.1,Davis
Jhon Durán,174,2,1,0,1,0,0,0,4,0,0,0,1.23,0.05,1.27,2.22,24,50,0,0.2,Duran
Ollie Watkins,1939,10,11,5,30,0,0,0,4,0,0,22,9.75,2.96,12.71,28.28,128,88,8,54.0,Watkins
Wesley Moraes Ferreira da Silva,0,0,0,0,0,0,0,0,0,0,0,0,0.00,0.00,0.00,0.00,0,50,0,0.0,Wesley
Kieffer Moore,100,1,0,0,2,0,0,0,0,0,0,0,1.44,0.33,1.77,2.26,14,46,-4,0.2,Moore
Antoine Semenyo,843,3,3,4,15,0,0,0,3,0,0,11,1.90,0.76,2.66,13.79,47,45,0,2.0,Semenyo
Dominic Solanke,1777,12,1,5,36,0,0,0,1,0,0,20,10.85,0.94,11.79,31.52,100,71,6,26.5,Solanke
Halil Dervişoğlu,0,0,0,0,0,0,0,0,0,0,0,0,0.00,0.00,0.00,0.00,0,45,0,0.0,Dervişoğlu
Ivan Toney,90,1,0,0,2,0,0,0,1,0,0,1,0.11,0.30,0.41,1.09,8,80,0,4.8,Toney
Yoane Wissa,1543,4,2,6,24,0,0,0,2,0,0,18,6.40,1.10,7.50,21.34,65,56,-4,4.5,Wissa
Neal Maupay,872,3,3,2,20,0,0,0,3,0,0,8,5.24,1.46,6.70,13.64,47,49,-1,0.9,Maupay
Aaron Connolly,0,0,0,0,0,0,0,0,0,0,0,0,0.00,0.00,0.00,0.00,0,45,0,0.0,Connolly
Evan Ferguson,1027,6,1,1,23,0,0,0,1,0,0,11,3.70,0.67,4.37,19.25,59,58,-2,5.0,Ferguson
João Pedro Junqueira de Jesus,1298,7,2,3,24,0,0,0,2,0,0,11,7.99,2.88,10.87,23.17,75,55,0,14.2,João Pedro
Deniz Undav,0,0,0,0,0,0,0,0,0,0,0,0,0.00,0.00,0.00,0.00,0,55,0,0.0,Undav
Danny Welbeck,751,2,1,5,12,0,0,0,1,0,0,11,2.29,0.49,2.78,12.86,35,57,-3,0.7,Welbeck
Andi Zeqiri,0,0,0,0,0,0,0,0,0,0,0,0,0.00,0.00,0.00,0.00,0,44,-1,0.4,Zeqiri
Mark O'Mahony,0,0,0,0,0,0,0,0,0,0,0,0,0.00,0.00,0.00,0.00,0,45,0,0.0,O'Mahony
Lyle Foster,982,4,3,2,24,0,0,0,0,1,0,11,1.94,0.56,2.50,20.15,49,47,-3,0.5,Foster
Michael Obafemi,19,0,0,0,0,0,0,0,0,0,0,0,0.11,0.01,0.12,0.03,2,49,-1,0.0,Obafemi
```

Figure 5: A screenshot of part of the CSV file storing player data

Subsequently, the data stored in the CSV file was used in the program via data structures, such as dictionaries and lists. As can be seen in Fig. 6, I first created a dictionary called *individualPlayer*, stored the data for a single player in it, and then added this dictionary in *playerStatistics,* a list which stores information of all players. I also changed some of the stored information from string to integer since it was later required in arithmetic calculations.

```
#read player season statistics
with open ('Data_analysis/PlayerSeasonData.csv', 'r') as csv_file:
    csv_reader = csv.reader(csv_file)
    for line in csv_reader:
        individualPlayer = {}
        individualPlayer["Name"] = line[0]
        individualPlayer["Minutes"] = line [1]
        individualPlayer["Goals"] = line [2]
        individualPlayer["Assists"] = line [3]
        individualPlayer["Clean sheets"] = line [4]
        individualPlayer["Goals conceded"] = line [5]
        individualPlayer["Own goals"] = line [6]
        individualPlayer["Penalties saved"] = line [7]
        individualPlayer["Penalties missed"] = line [8]
        individualPlayer["Yellow cards"] = line [9]
        individualPlayer["Red cards"] = line [10]
        individualPlayer["Saves"] = line [11]
        individualPlayer["Starts"] = line [12]
        individualPlayer["xG"] = line [13]
        individualPlayer["xA"] = line [14]
        individualPlayer["Expected goal involvements"] = line [15]
        individualPlayer["Expected goals conceded"] = line [16]
        individualPlayer["Total points"] = line [17]
        individualPlayer["Current cost"] = int(line [18])
        individualPlayer["Change of cost to the start"] = line [19]
        individualPlayer["Form rating"] = 0
        individualPlayer["Position"] = ""
        individualPlayer["Relative form rating"] = 0
        individualPlayer["Club"] = ""
        individualPlayer["Percentage selection"] = float(line [20])/100
        individualPlayer["Web name"] = line [21]
        players_statistics.append(individualPlayer)
```

Figure 6: Reading contents of the CSV file and storing them into Python data structures

```
#Storing basic data about players, for e.g., name, position, and club
with open('Data_analysis/PlayerBasicData.csv', 'w') as f:
    writer = csv.writer(f)
    for i in range(0, 726):
        content_list=[]
        content_list.append(df.iloc[i]["first_name"] + " " + df.iloc[i]["second_name"])
        content_list.append(df.iloc[i]["position_name"])
        content_list.append(df.iloc[i]["team_name"])
        writer.writerow(content_list)

#Making some textual changes to the file such that it is consistent with other saved files and their attributes
text = open("Data_analysis/PlayerBasicData.csv", "r")
text = ''.join([i for i in text]).replace("Forward", "FW")
text = ''.join([i for i in text]).replace("Midfielder", "MF")
text = ''.join([i for i in text]).replace("Defender", "DF")
text = ''.join([i for i in text]).replace("Goalkeeper", "GK")
text = ''.join([i for i in text]).replace("Man City", "Manchester City")
text = ''.join([i for i in text]).replace("Man Utd", "Manchester United")
text = ''.join([i for i in text]).replace("Nott'm Forest", "Nottingham Forest")
text = ''.join([i for i in text]).replace("Spurs", "Tottenham Hotspur")
x = open("Data_analysis/PlayerBasicData.csv","w")
x.writelines(text)
x.close()
```

Figure 7: Storing basic player data into another CSV file and making some required textual changes through code

Determining the optimum team

As shown in Fig. 8, I created a method **auto_suggest_team()** that automates the suggestion of the optimum team based on formation and positional weights, which are user-inputted. The method uses the CVXPY library for convex optimization to maximize a weighted sum of players' relative form ratings while adhering to budget constraints, position requirements, and limit (3) on the number of players from a single club.

Firstly, the selected formation (for example, "4-5-1", or "5-3-2") is retrieved from the choice made in the dropdown menu, *choose_formation*, and the user is prompted to input positional weights which are obtained through the **get_position_weights()** method, as can be seen in Fig. 8. Next, a dictionary, *formation_counts = {'GK': 1, 'DF': 0, 'MF': 0, 'FW': 0},* is initialized to represent the initial count of players in each position. This count is later updated based on the selected formation using a predefined mapping.

```
def auto_suggest_team (self):
    # Get the selected formation from the dropdown
    selected_formation = self.choose_formation.get()

    # Prompt the user for positional weights
    position_weights = self.get_position_weights()

    formation_counts = {'GK': 1, 'DF': 0, 'MF': 0, 'FW': 0}

    # Define formation counts for each formation
    formation_counts_mapping = {
        '4-4-2': {'DF': 4, 'MF': 4, 'FW': 2},
        '3-5-2': {'DF': 3, 'MF': 5, 'FW': 2},
        '4-5-1': {'DF': 4, 'MF': 5, 'FW': 1},
        '5-4-1': {'DF': 5, 'MF': 4, 'FW': 1},
    }

    formation_counts.update(formation_counts_mapping[selected_formation])
```

Figure 8: Working with and reading the user-selected playing formation

Moreover, as seen in Fig. 9, relevant information, including "Relative form rating", "Current cost", "Position", and "Club", are extracted from the player statistics data, converting them into NumPy arrays for later calculations with the CVXPY library's methods[2] (Skinner, 2023). Next, the weighted relative form ratings of players are calculated based on the user-inputted positional weights. This is done via a list

[2] https://medium.com/@dylanskinner65/convex-linear-optimization-with-cvxpy-5fa1024254ff

comprehension that iterates over pairs of **pos** *(position)* and **rating** *(relative form rating)* obtained from the **zip(positions, relative_form)** expression. For each player, their relative form rating (**rating**) is multiplied by the positional weight associated with their position (**position_weights[pos]**).

```
    # Extract relevant information from player data
    relative_form = np.array([player["Relative form rating"] for player in player_data])
    cost = np.array([player["Current cost"] for player in player_data])
    positions = np.array([player["Position"] for player in player_data])
    clubs = np.array([player["Club"] for player in player_data])

    # Apply position weights to relative form ratings
    weighted_relative_form = np.array([position_weights[pos] * rating for pos, rating in zip(positions, relative_form)])
```

Figure 9: Extracting relevant information from data and finding weighted relative form

Furthermore, as shown in Fig. 10, I defined the optimization variables, the objective function to maximize, and constraints based on budget, position requirements, and the additional constraint of a maximum of three players from a single club. Finally, the convex optimization problem is defined (**problem = cp.problem(objective, constraints)**) and then the program solves it (**problem.solve()**) using the specified objective function and constraints.

```
# Define the optimization variables
x = cp.Variable(len(player_data), boolean=True)

# Define the optimization problem
objective = cp.Maximize(cp.sum(x * weighted_relative_form))
constraints = [
    cp.sum(x * cost) <= 1000,
    cp.sum(x * cost) >= 990,
    cp.sum(x[positions == "GK"]) == 2,
    cp.sum(x[positions == "DF"]) == 5,
    cp.sum(x[positions == "MF"]) == 5,
    cp.sum(x[positions == "FW"]) == 3,
]

# Additional constraint: No more than three players from a single club
for club in set(clubs):
    constraints.append(cp.sum(x[clubs == club]) <= 3)

problem = cp.Problem(objective, constraints)

# Solve the problem
problem.solve()
```

Figure 10: Defining the optimization variable and the optimization problem and then solving it

A dynamic GUI changing based on other components, such as buttons

In the "Players" window, a sorted list of players based on a selection criterion, either position or club, is shown. Therefore, the combo box which shows the different positions and clubs to choose from must be updated accordingly. As seen in Fig. 11 and Fig. 12, this is achieved via the **update_combobox_options()** method which assigns the list of clubs, *clubs,* and the list of positions, *positions,* to the values of the combo box depending on the selection criterion chosen. The result of the code below is shown in the dynamic GUI in Fig. 13.

```
club_radio.bind("<ButtonRelease-1>", self.update_combobox_options)
position_radio.bind("<ButtonRelease-1>", self.update_combobox_options)
```

Figure 11: Updating the combo box according to whether position or club is chosen as the filter

```
def update_combobox_options(self, *args):
    # Get the selected criteria from the dropdown
    selected_criteria = self.criteria_var.get()

    # Update Combobox options based on the selected criteria
    if selected_criteria == "club":
        # If 'club' is selected, set Combobox options to the list of clubs
        self.filter_combobox['values'] = clubs
    elif selected_criteria == "position":
        # If 'position' is selected, set Combobox options to the list of positions
        self.filter_combobox['values'] = positions
```

Figure 12: Method used to update the combo box

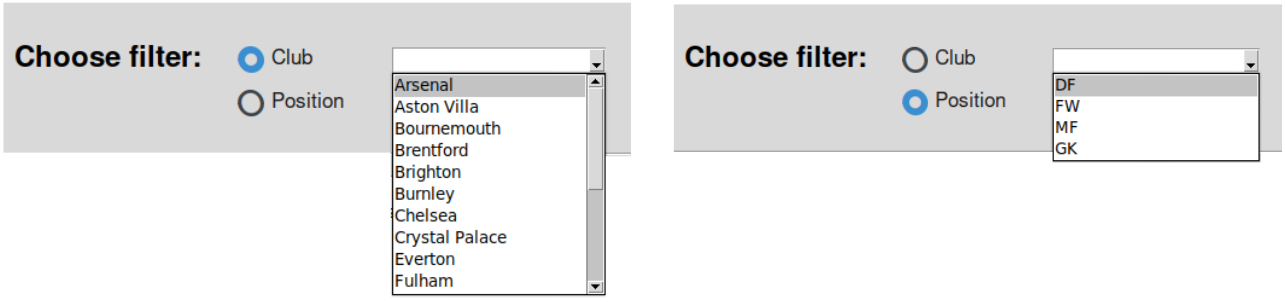

Figure 13: Drop-down changing based on the choice of selection criteria: club or position

Similarly, when a comparison table needs to be presented, the list of players available in the drop-down should change according to the club and position selected for comparison. This is achieved using the code shown in Fig. 14.

```
# Function to update player dropdowns based on selected club and position
def update_player_dropdowns1(*args):
    # Clear the selected values in club and player dropdowns
    self.player_var_left.set("")  # Clear the left player dropdown

    # Get the selected values from club and position dropdowns
    selected_club_left = self.club_var_left.get()
    selected_club_right = self.club_var_right.get()
    selected_position = self.position_var.get()

    # List to store player values for left dropdown
    values = []

    # Check if both club and position are selected for the left dropdown
    if selected_club_left and selected_position:
        # Retrieve players associated with the selected club from the data structure
        players_left = clubWisePlayerData.get(selected_club_left, [])
        # Filter players based on the selected position
        for player in players_left:
            if player in positionWisePlayerData.get(selected_position, []):
                values.append(player)
        # Set the values for the left player dropdown
        self.player_dropdown_left['values'] = values

    # List to store player values for right dropdown
    values2 = []

    # Check if both club and position are selected for the right dropdown
    if selected_club_right and selected_position:
        # Retrieve players associated with the selected club from the data structure
        players_right = clubWisePlayerData.get(selected_club_right, [])
        # Filter players based on the selected position
        for player in players_right:
            if player in positionWisePlayerData.get(selected_position, []):
                values2.append(player)
        # Set the values for the right player dropdown
        self.player_dropdown_right['values'] = values2
```

Figure 14: Updating player drop-down based on selected club and position

Placing the optimum team on the pitch background

As seen in Fig. 15 and Fig. 16, to display the optimal team that has been selected on the pitch in the window, I make use of coordinates and grid placing. Keeping the y-coordinate constant for a given position, and subsequently changing the x-coordinate for the players in that position based on formation chosen (number of players in each row), the program can produce a structured representation, like the one shown in Fig. 17. Firstly, using the **.PhotoImage()** method of Python, I printed a background image on the window. This then served as the pitch on which the names of the players are printed and shown using the **.create_text()** method.

```python
def calculate_position_x(position, index, total_players):
    # Calculate the x-coordinate position of a player based on their position, index, and total number of players

    if position == 'GK':
        return 450
    elif position == 'DF' or position == 'MF' or position == 'FW':
        # For positions other than 'GK', distribute players evenly along the width of the pitch
        return (index + 1) * (900 / (total_players + 1))

def calculate_position_y(position):
    # Calculate the y-coordinate position of a player based on their playing position

    if position == 'GK':
        return 430
    elif position == 'DF':
        return 320
    elif position == 'MF':
        return 180
    elif position == 'FW':
        return 90
```

Figure 15: Calculating x and y coordinates of the player's name to be placed on the pitch background

```python
def display_team_on_pitch(self, team):
    # Clear previous drawings
    self.pitch_canvas.delete("all")
    self.background_image = Image.open("GUI/Canvas.jpg")
    self.background_image = self.background_image.resize((900, 540), Image.ANTIALIAS)
    self.background_image = ImageTk.PhotoImage(self.background_image)

    # Display the background image on the canvas
    self.pitch_canvas.create_image(0, 0, anchor=tk.NW, image=self.background_image)

    # Draw players on the pitch based on their positions
    for pos, players in team.items():
        for i, player in enumerate(players):
            # Calculate position based on formation and player index
            x = calculate_position_x(pos, i, len(players))
            y = calculate_position_y(pos)

            # Display player name on the canvas
            self.pitch_canvas.create_text(x, y, text=player["Web name"], fill='white', font=('Moderna', 15, 'bold'))
```

Figure 16: Method used to display the team on the pitch background

Figure 17: A pitch view of the optimal team as of 1st February 2024, for formation "5-4-1" and positional weights, "1,3,4,2"

Updating the contents of a table

```
def compare_players(self):
    # Retrieve the selected players from the left and right dropdowns
    selected_left_player = self.player_var_left.get()
    selected_right_player = self.player_var_right.get()

    # Initialize dictionaries to store data for the selected players
    left_player_data = {}
    right_player_data = {}

    # Iterate through the players' statistics to extract data for the selected players
    for item in players_statistics:
        if item["Name"] == selected_left_player:
            left_player_data = item
        if item["Name"] == selected_right_player:
            right_player_data = item

    # Clear any previous comparison results in the comparison frame
    for widget in self.comparison_frame.winfo_children():
        widget.destroy()

    # Create a Treeview to present the comparison results in a table
    columns = [selected_left_player, "Metric", selected_right_player]
    tree = ttk.Treeview(self.comparison_frame, columns=columns, show="headings")

    # Set column headings to the players' web names
    tree.heading(selected_left_player, text=left_player_data["Web name"], anchor='center')
    tree.heading("Metric", text="Metric", anchor='center')
    tree.heading(selected_right_player, text=right_player_data["Web name"], anchor='center')
```

Figure 18: Method used to update and show the comparison table (ctd…)

```
# Define metrics based on player position
if self.position_var.get() == "FW":
    metrics = ["Minutes", "Goals", "Assists", "Penalties missed", "Yellow cards", "Red cards", "Starts", "xG", "xA"]
elif self.position_var.get() == "MF":
    metrics = ["Minutes", "Goals", "Assists", "Penalties missed", "Yellow cards", "Red cards", "Starts", "xG", "xA"]
elif self.position_var.get() == "DF":
    metrics = ["Minutes", "Goals", "Assists", "Clean sheets", "Goals conceded", "Yellow cards", "Red cards", "Starts"
elif self.position_var.get() == "GK":
    metrics = ["Minutes", "Clean sheets", "Goals conceded", "Penalties saved", "Yellow cards", "Red cards", "Saves",

# Populate the Treeview with data
for metric in metrics:
    left_value = left_player_data.get(metric, "-")
    right_value = right_player_data.get(metric, "-")
    tree.insert("", "end", values=[left_value, metric, right_value], tags=('centered',))

# Center the contents within the cells of the Treeview
tree.tag_configure('centered', anchor='center')

# Style the Treeview for improved visual appeal
style = ttk.Style()
style.configure("Treeview", font=("Arial", 12), rowheight=30)
style.configure("Treeview.Heading", font=("Arial", 14, "bold"))

# Display the Treeview within the comparison frame
tree.grid(row=0, column=0, sticky='w', pady=10)
```

Figure 19: Method used to update and show the comparison table

Data dictionaries

It is important to discuss the variables and data structures used in the program since it is a data-heavy project with a lot of different information to be stored and used. These data dictionaries will help in the summarization of the heaps of data, while allowing the reader to refer to them to understand what is being stored.

Data dictionary for management of official database for FPL players

Field name	**Data type**	**Description**	**Additional notes**
playersStatistics	Pandas DataFrame	Merged DataFrame that combines information from various elements of data from FPL API ("players", "clubs", "positions", etc)	*pd.json_normalize()* will needed to be used to flatten nested JSON structures into Panas DataFrame since official data is extracted in JSON format from the FPL API
clubWisePlayerData	Dictionary	Contains the names of clubs as keys and the players in each club as values	
positionWisePlayerData	Dictionary	Contains different positions as keys and the players playing in each position as values	
individualPlayerData	Dictionary	Stores information for every player where the player is the key, and their data is the value	
clubs	List	A list of all clubs playing in the Premier League	Needs to be directly connected with *playerStatistics* so that it stays updated

			as teams get promoted and relegated every season
positions	List	A list of all possible playing positions: [“FW, “MF”, “DF”, “GK”]	
entirePlayerData	List	A list of dictionaries where all the *individualPlayerData*(s) are stored	Will allow for easy access and sorting while displaying data to the

Data dictionary for individual player statistics

Field name	**Data type**	**Description**	**Additional notes**
playerName	string	Full name of the player as a concatenation of *first_name* and *last_name*	
webName	string	Name that is shown on the FPL website (included for the user’s convenience)	In FPL, each player is assigned a web name, which is a shortened version for the player’s full name and is used as an identifier when storing data
minutesPlayed	int	Number of minutes played in the season at the time of data extraction	
club	string	Name of club for which the player plays	

position	string	Player's playing position	
goalsScored	int	Number of goals scored in the season at a given moment	
assists	int	Number of assists made in the season at a given moment	
totalPoints	int	Total points acquired over the course of the season	
xG	double	Expected goals from the player based on overall performance	
xA	double	Expected assists from the player based on overall performance	
percentageSelection	double	Percentage of FPL players who have selected the player as a part of their team	Key in calculating form rating, since a greater number of people selecting a player indicates a high level of game performance
Form rating	double	Custom-defined form rating index	
Relative form rating	double	Form rating relative to other players playing in the same position	Needed to sort the players in a descending order based on performance
currentCost	double	Current cost of the player in the FPL market	Useful while optimizing the weekly teams (due to total budget constraint)

redCards	int	Number of red cards given during the season	In FPL, a red/yellow card leads to points deduction. Relevant for assigning form rating to consider fair play statistics of player
yellowCards	int	Number of yellow cards given during the season	In FPL, a red/yellow card leads to points deduction. Relevant for assigning form rating to consider fair play statistics of player
saves	int	Number of saves made	Relevant while determining the form rating of a goalkeeper
goalsConceded	int	Number of goals conceded	Relevant while determining the form rating of a goalkeeper
penaltiesSaved	int	Number of penalties saved	Relevant while determining the form rating of a goalkeeper
cleanSheets	int	Number of clean sheets kept	Relevant while determining the form rating of a goalkeeper

Data dictionary for auto-suggestion of optimal team

Field name	Data type	Description	Additional notes
formation_count	Dictionary	Stores details of players in specific positions for different formations	Player positions are keys and number of players in each position is the value
different_formation_map	Dictionary {string, Dictionary (string, int)}	Contains details for all available playing formations in the game	Keys are names of formation, for e.g., "4-4-2" and value is a *formation_count* dictionary
relativeRating	NumPy array	Represents relative form rating of all players in *playerStatistics*	Needed to convert into NumPy array to match input requirements of the convex programming library in Python (CVXPY)
weighted_relative_form	NumPy array	Contains value based on relative form ratings and positional weights	Helps turn the problem into a linear programming problem
selected_players	List	Outputs list of 15 best players possible based on the positional weights and various constraints	
selected_team	Dictionary	Outputs dictionary of 11 best players in specific positions out of the optimum team based on chosen formation	Key is the playing position and value is the list of players starting in that position
positionalWeights	Dictionary	Contains input from user about how much weight must be given to each	Key is the name of specific position and

		position while calculating optimum team	value is an integer weight given

Word count: 815

Works Cited

Leslie, J. (2021, May 5). *Accessing Fantasy Premier League data using Python*. Retrieved from Medium: https://medium.com/analytics-vidhya/getting-started-with-fantasy-premier-league-data-56d3b9be8c32

Skinner, D. (2023, October 20). *Convex Linear Optimization with CVXPY*. Retrieved from Medium: https://medium.com/@dylanskinner65/convex-linear-optimization-with-cvxpy-5fa1024254ff