POSTS AND TELECOMMUNICATIONS INSTITUTE OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY I

———————o0o———————

PROJECT REPORT 1

PYTHON PROGRAMMING LANGUAGE

Instructor:	Kim Ngoc Bach
Student:	Dong Duc Nguyen
Student ID:	B23DCVT311
Class:	D23CQCEO6-B
Course Year:	2023 - 2028
Training System:	Full-time University

Hanoi, 2025

POSTS AND TELECOMMUNICATIONS INSTITUTE OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY I

———————o0o———————

PROJECT REPORT 1

PYTHON PROGRAMMING LANGUAGE

Instructor:	Kim Ngoc Bach
Student:	Dong Duc Nguyen
Student ID:	B23DCVT311
Class:	D23CQCEO6-B
Course Year:	2023 - 2028
Training System:	Full-time University

Hanoi, 2025

INSTRUCTOR'S COMMENTS

....
....
....
....
....
....
....
....
....
....
....
....
....
....
....
....
....
....
....

Score: **(In words:**)

Hanoi, date month year 20...

Instructor

Contents

List of Figures

INTRODUCTION

In the digital era, the ability to collect, process, and analyze data has become an essential skill in many fields, including sports. The Python programming language, with its rich and powerful library ecosystem such as `Pandas`, `Scikit-learn`, `Selenium`, and `BeautifulSoup`, provides effective tools to solve complex data analysis problems. This Major Project Report for the *Python Programming Language* course focuses on applying Python to perform a series of tasks related to football data analysis, specifically from the English Premier League. **The main objectives of the report include:**

- Collecting detailed statistical data of players from reliable online sources (`fbref.com`, `footballtransfers.com`) using web scraping techniques.
- Performing descriptive statistical analyses to explore the data, identify outstanding players, and characteristics of the teams.
- Applying unsupervised machine learning algorithms (K-means) combined with dimensionality reduction techniques (PCA) to classify players into groups based on their playing characteristics.
- Building a supervised machine learning model (`Random Forest Regressor`) to estimate player transfer values based on statistical indicators and personal information.

The report is structured into four main parts (Problem I, II, III, IV), each addressing a specific requirement mentioned above. By performing these tasks, the report not only illustrates the applicability of Python in sports data analysis but also provides deep insights into player performance and transfer market dynamics.

Chapter 1

PROBLEM I

1.1 Requirement

- Collect statistical data [*] for all players who have played more than 90 minutes in the 2024-2025 English Premier League season.
- Data source: https://fbref.com/en/
- Save the result to a file named 'results.csv', where the result table has the following structure:
  - Each column corresponds to a statistic.
  - Players are sorted alphabetically by their first name.
  - Any statistic that is unavailable or inapplicable should be marked as "N/a".

* The required statistics are:

  - Nation
  - Team
  - Position
  - Age
  - Playing Time: matches played, starts, minutes
  - Performance: goals, assists, yellow cards, red cards
  - Expected: expected goals (xG), expected Assist Goals (xAG)
  - Progression: PrgC, PrgP, PrgR
  - Per 90 minutes: Gls, Ast, xG, xGA
  - Goalkeeping:
    - Performance: goals against per 90mins (GA90), Save%, CS%

    * Penalty Kicks: penalty kicks Save%

  - Shooting:
    * Standard: shots on target percentage (SoT%), Shot on Target per 90min (SoT/90), goals/shot (G/Sh), average shot distance (Dist)

  - Passing:
    * Total: passes completed (Cmp), pass completion (Cmp%), progressive passing distance (TotDist)
    * Short: pass completion (Cmp%)
    * Medium: pass completion (Cmp%)
    * Long: pass completion (Cmp%)
    * Expected: key passes (KP), pass into final third (1/3), pass into penalty area (PPA), CrsPA, PrgP

  - Goal and Shot Creation:
    * SCA: SCA, SCA90
    * GCA: GCA, GCA90

  - Defensive Actions:
    * Tackles: Tkl, TklW
    * Challenges: Att, Lost
    * Blocks: Blocks, Sh, Pass, Int

  - Possession:
    * Touches: Touches, Def Pen, Def 3rd, Mid 3rd, Att 3rd, Att Pen
    * Take-Ons: Att, Succ%, Tkld%
    * Carries: Carries, ProDist, ProgC, 1/3, CPA, Mis, Dis
    * Receiving: Rec, PrgR

  - Miscellaneous Stats:
    * Performance: Fls, Fld, Off, Crs, Recov
    * Aerial Duels: Won, Lost, Won%

  - Reference: `https://fbref.com/en/squads/822bd0ba/Liverpool-Stats`

1.2 Implementation Steps

1. Initialize data fields:

   - Identify the information to be collected (the fields).
   - Create a player dictionary (or class) with the default value 'N/a'.

- Create a dictionary to store all players by name + team.

2. Retrieve data:

   - Retrieve basic data from the summary page (https://fbref.com/en/comps/9/stats/Premier-League-Stats) to initialize the player count, filtering out players who did not play more than 90 minutes. Add to the dictionary to prepare for updates.
   - Update data according to requirements (only update players present in the dictionary).
   - Update according to each required section to ensure completeness.

3. Export data:

   - Export fields according to the requirements (convert to a list containing content as required).
   - Fix the data into a DataFrame.
   - Export data to the 'results.csv' file.

1.3 Actual Code and Detailed Description

1.3.1 Main Function

```
def Problem_1():

        print("Starting to retrieve basic data...")
        player_set_dict = create_Set_Players()
        print(f"Retrieved basic data for {len(player_set_dict)} players.")

        print("Updating goalkeeping data...")
        update_Set_Goalkeeping(player_set_dict)
        print("Updating shooting data...")
        update_Set_Shooting(player_set_dict)
        print("Updating passing data...")
        update_Set_Passing(player_set_dict)
        print("Updating goal and shot creation data...")
        update_Set_Goal_And_Shot_Creation_Data(player_set_dict)
        print("Updating defensive actions data...")
        update_Set_Defensive_Actions_Data(player_set_dict)
        print("Updating possession data...")
        update_Set_Possession(player_set_dict)
        print("Updating miscellaneous data...")
        update_Set_Miscellaneous_Data(player_set_dict)
        print("Data update completed.")

        export(player_set_dict)
        print("Program completed.")
```

Operations are similar to those in the implementation steps section: *create_Set_Players()*
This is the initialization function that also retrieves basic data from the summary page to

create a dictionary of all players who have played more than 90 minutes. Retrieves data from https://fbref.com/en/comps/9/stats/Premier-League-Stats. **Operations:**

```
update_Set_Goalkeeping(player_set_dict)
update_Set_Shooting(player_set_dict)
update_Set_Passing(player_set_dict)
update_Set_Goal_And_Shot_Creation_Data(player_set_dict)
update_Set_Defensive_Actions_Data(player_set_dict)
update_Set_Possession(player_set_dict)
update_Set_Miscellaneous_Data(player_set_dict)
```

These are the operations during the process of updating the required data:

- update_Set_Goalkeeping(player_set_dict): operation to update Goalkeeping data, retrieving from: https://fbref.com/en/comps/9/keepers/Premier-League-Stats
- update_Set_Shooting(player_set_dict): operation to update Shooting data, retrieving from: https://fbref.com/en/comps/9/shooting/Premier-League-Stats
- update_Set_Passing(player_set_dict): operation to update Passing data, retrieving from: https://fbref.com/en/comps/9/passing/Premier-League-Stats
- update_Set_Goal_And_Shot_Creation_Data(player_set_dict): operation to update Goal And Shot Creation data, retrieving from: https://fbref.com/en/comps/9/gca/Premier-League-Stats
- update_Set_Defensive_Actions_Data(player_set_dict): operation to update Defensive Actions data, retrieving from: https://fbref.com/en/comps/9/defense/Premier-League-Stats
- update_Set_Possession(player_set_dict): operation to update Possession data, retrieving from: https://fbref.com/en/comps/9/possession/Premier-League-Stats
- update_Set_Miscellaneous_Data(player_set_dict): operation to update Miscellaneous data, retrieving from: https://fbref.com/en/comps/9/misc/Premier-League-Stats

export(player_set_dict): operation to fix data into a DataFrame and export to the 'results.csv' file as required.

1.3.2 Detailed Operations

*Initialize dictionaries containing player information:

```python
PLAYER_KEYS = ['name', 'nationality', 'position', 'team', 'age', 'games', 'games_starts', 'minutes', 'goals', 'assist', 'cards_yellow', 'cards_red', 'xg', 'xg_assist', 'progressive_carries', 'progressive_passes', 'progressive_passes_received', 'goals_per90', 'assists_per90', 'xg_per90', 'xg_assist_per90', 'gk_goals_against_per90', 'gk_save_pct', 'gk_clean_sheets_pct', 'gk_pens_save_pct', 'shots_on_target_pct', 'shots_on_target_per90', 'goals_per_shot', 'average_shot_distance', 'passes_completed', 'passes_pct', 'passes_total_distance', 'passes_pct_short', 'passes_pct_medium', 'passes_pct_long', 'assisted_shots', 'passes_into_final_third', 'passes_into_penalty_area', 'crosses_into_penalty_area', 'sca', 'sca_per90', 'gca', 'gca_per90', 'tackles', 'tackles_won', 'challenges', 'challenges_lost', 'blocks', 'blocked_shots', 'blocked_passes', 'interceptions', 'touches', 'touches_def_pen_area', 'touches_def_3rd', 'touches_mid_3rd', 'touches_att_3rd', 'touches_att_pen_area', 'take_ons', 'take_ons_won_pct', 'take_ons_tackled_pct', 'carries', 'carries_progressive_distance', 'carries_into_final_third',
'carries_into_penalty_area', 'miscontrols', 'dispossessed', 'passes_received', 'fouls', 'fouled', 'offsides', 'crosses', 'ball_recoveries', 'aerials_won', 'aerials_lost', 'aerials_won_pct']

def create_default_player_dict(): return {key: 'N/a' for key in PLAYER_KEYS}
```

This operation aims to create a player dictionary with the attributes listed in PLAYER_KEYS, which have been previously declared. We set the initial value for all data to 'N/a' so that any values not found (unavailable or inapplicable) will default to this value.

*create_Set_Players() - Create player dictionary:

```python
def create_Set_Players():
    driver = webdriver.Chrome()
    url = 'https://fbref.com/en/comps/9/stats/Premier-League-Stats'
    driver.get(url)
    page_source = driver.page_source
    soup = BeautifulSoup(page_source, 'html.parser')
    x = soup.find('table', attrs={'id': 'stats_standard'})
cnt = 0
    player_set = {}
    for i in range(589):
        cnt += 1
        if cnt == 26:
            cnt = 0
            continue
        table = x.find('tr', attrs={'data-row': f'{str(i)}'})
        if not table:
            continue
  player_data = create_default_player_dict()
        player_data['name'] = table.find('td', attrs={'data-stat': 'player'}).text.strip()
        player_data['nationality'] = table.find('td', attrs={'data-stat': 'nationality'}).text.strip()
        player_data['position'] = table.find('td', attrs={'data-stat': 'position'}).text.strip()
        player_data['team'] = table.find('td', attrs={'data-stat': 'team'}).text.strip()
        player_data['age'] = table.find('td', attrs={'data-stat': 'age'}).text.strip()
        player_data['games'] = table.find('td', attrs={'data-stat': 'games'}).text.strip()
        player_data['games_starts'] = table.find('td', attrs={'data-stat': 'games_starts'}).text.strip()
        minutes_str = table.find('td', attrs={'data-stat':
'minutes'}).text.strip()
        player_data['minutes'] = minutes_str
```

```
29          minutes_str_cleaned = minutes_str.replace(',', '')
30          if int(minutes_str_cleaned) <= 90:
31              continue
32          player_data['goals'] = table.find('td', attrs={'data-stat': 'goals'}).text.strip
                ()
33          player_data['assist'] = table.find('td', attrs={'data-stat': 'assists'}).text.
                strip()
34          player_data['cards_yellow'] = table.find('td', attrs={'data-stat': 'cards_yellow'
                }).text.strip()
35          player_data['cards_red'] = table.find('td', attrs={'data-stat': 'cards_red'}).
                text.strip()
36  player_data['xg'] = table.find('td', attrs={'data-stat': 'xg'}).text.strip()
37          player_data['xg_assist'] = table.find('td', attrs={'data-stat': 'xg_assist'}).
                text.strip()
38          player_data['progressive_carries'] = table.find('td', attrs={'data-stat': '
                progressive_carries'}).text.strip()
39          player_data['progressive_passes'] = table.find('td', attrs={'data-stat': '
                progressive_passes'}).text.strip()
40          player_data['progressive_passes_received'] = table.find('td', attrs={'data-stat':
                 'progressive_passes_received'}).text.strip()
41          player_data['goals_per90'] = table.find('td', attrs={'data-stat': 'goals_per90'})
                .text.strip()
42          player_data['assists_per90'] = table.find('td', attrs={'data-stat': '
                assists_per90'}).text.strip()
43          player_data['xg_per90'] = table.find('td', attrs={'data-stat': 'xg_per90'}).text.
                strip()
44          player_data['xg_assist_per90'] = table.find('td', attrs={'data-stat':
45  'xg_assist_per90'}).text.strip()
46          player_key = str(player_data['name']) + str(player_data['team'])
47          player_set[player_key] = player_data
48      driver.quit()
49      return player_set
```

To avoid being blocked and ensure completeness, accuracy, and prevent failure during data retrieval from the website, use a webdriver from the selenium library, assign it to the variable ‘driver‘, and proceed with data retrieval [lines 2 - 5]. Once the website data is obtained, normalize it using BeautifulSoup [line 6]. End the process of retrieving data from the website. Start the search process. First, to reduce search time, retrieve only the necessary data (the information table into variable ‘x‘) [line 7]. Create a ‘cnt‘ variable to skip header rows [lines 8, 11 - 14]. Create a player dataset ‘player_set‘ to store valid players [line 9]. Loop to retrieve all player information. In each loop, create a new dictionary containing the new player’s information [line 18]. Find the name, nationality, team, age, matches played, starts, playing time, and assign them to the keys in the dictionary [lines 19 - 27]. Normalize the playing time, then compare; if it’s over 90 minutes, continue retrieving the remaining data, otherwise move to the next player. Continue retrieving all data until the end of the table. Assign the newly created dictionary to the dictionary containing all valid players created in line 9, using the player’s name + team name as the key. After the loop finishes, quit the driver and return the dictionary containing all valid players created in line 9.

*Update functions:

```
1 def update_Set_Goalkeeping(player_set):
```

```python
    driver = webdriver.Chrome()
    url = 'https://fbref.com/en/comps/9/keepers/Premier-League-Stats'
    driver.get(url)
    page_source = driver.page_source
    soup = BeautifulSoup(page_source, 'html.parser')
    x = soup.find('table', attrs={'id': 'stats_keeper'})
    for i in range (43):
        if i == 25: continue
table = x.find('tr', attrs={'data-row':f'{str(i)}'})
        if not table: continue
        name = table.find('td', attrs={'data-stat':'player'}).text.strip()
        team = table.find('td', attrs={'data-stat':'team'}).text.strip()
        player_key = str(name) + str(team)

        if player_key in player_set:
            player_set[player_key]['gk_goals_against_per90'] = table.find('td', attrs={'data-stat':'gk_goals_against_per90'}).text.strip()
            player_set[player_key]['gk_save_pct'] = table.find('td', attrs={'data-stat':'gk_save_pct'}).text.strip()
 player_set[player_key]['gk_clean_sheets_pct'] = table.find('td', attrs={'data-stat':'gk_clean_sheets_pct'}).text.strip()
            player_set[player_key]['gk_pens_save_pct'] = table.find('td', attrs={'data-stat':'gk_pens_save_pct'}).text.strip()
    driver.quit()

def update_Set_Shooting(player_set):
    driver = webdriver.Chrome()
    url = 'https://fbref.com/en/comps/9/shooting/Premier-League-Stats'
    driver.get(url)
    page_source = driver.page_source
    soup = BeautifulSoup(page_source, 'html.parser')
    x = soup.find('table', attrs={'id': 'stats_shooting'})
    cnt = 0
    for i in range(589):
        cnt+=1
        if cnt==26:
  cnt=0
            continue
        table = x.find('tr', attrs={'data-row':f'{str(i)}'})
        if not table: continue

        name = table.find('td', attrs={'data-stat':'player'}).text.strip()
        team = table.find('td', attrs={'data-stat':'team'}).text.strip()
        player_key = str(name) + str(team)

        if player_key in player_set:
            player_set[player_key]['shots_on_target_pct'] = table.find('td', attrs={'data-stat':'shots_on_target_pct'}).text.strip()
         player_set[player_key]['shots_on_target_per90'] = table.find('td', attrs={'data-stat':'shots_on_target_per90'}).text.strip()
            player_set[player_key]['goals_per_shot'] = table.find('td', attrs={'data-stat':'goals_per_shot'}).text.strip()
            player_set[player_key]['average_shot_distance'] = table.find('td', attrs={'data-stat':'average_shot_distance'}).text.strip()
    driver.quit()

def update_Set_Passing(player_set):
    driver = webdriver.Chrome()
    url = 'https://fbref.com/en/comps/9/passing/Premier-League-Stats'
    driver.get(url)
```

```
    page_source = driver.page_source
    soup = BeautifulSoup(page_source, 'html.parser')
    x = soup.find('table', attrs={'id': 'stats_passing'})
    cnt = 0
    for i in range(589):
     cnt+=1
        if cnt==26:
            cnt=0
            continue
        table = x.find('tr', attrs={'data-row':f'{str(i)}'})
        if not table: continue

        name = table.find('td', attrs={'data-stat':'player'}).text.strip()
        team = table.find('td', attrs={'data-stat':'team'}).text.strip()
        player_key = str(name) + str(team)

  if player_key in player_set:
            player_set[player_key]['passes_completed'] = table.find('td', attrs={'data-stat':'passes_completed'}).text.strip()
            player_set[player_key]['passes_pct'] = table.find('td', attrs={'data-stat':'passes_pct'}).text.strip()
            player_set[player_key]['passes_total_distance'] = table.find('td', attrs={'data-stat':'passes_total_distance'}).text.strip()
            player_set[player_key]['passes_pct_short'] = table.find('td', attrs={'data-stat':'passes_pct_short'}).text.strip()
            player_set[player_key]['passes_pct_medium'] = table.find('td', attrs={'data-stat':'passes_pct_medium'}).text.strip()
            player_set[player_key]['passes_pct_long'] = table.find('td', attrs={'data-stat':'passes_pct_long'}).text.strip()
       player_set[player_key]['assisted_shots'] = table.find('td', attrs={'data-stat':'assisted_shots'}).text.strip()
            player_set[player_key]['passes_into_final_third'] = table.find('td', attrs={'data-stat':'passes_into_final_third'}).text.strip()
            player_set[player_key]['passes_into_penalty_area'] = table.find('td', attrs={'data-stat':'passes_into_penalty_area'}).text.strip()
            player_set[player_key]['crosses_into_penalty_area'] = table.find('td', attrs={'data-stat':'crosses_into_penalty_area'}).text.strip()
            player_set[player_key]['progressive_passes'] = table.find('td', attrs={'data-stat':'progressive_passes'}).text.strip()
    driver.quit()


def update_Set_Goal_And_Shot_Creation_Data(player_set):
    driver = webdriver.Chrome()
    url = 'https://fbref.com/en/comps/9/gca/Premier-League-Stats'
    driver.get(url)
    page_source = driver.page_source
 soup = BeautifulSoup(page_source, 'html.parser')
    x = soup.find('table', attrs={'id': 'stats_gca'})
    cnt = 0
    for i in range(589):
        cnt+=1
        if cnt==26:
            cnt=0
            continue
        table = x.find('tr', attrs={'data-row':f'{str(i)}'})
        if not table: continue

        name = table.find('td', attrs={'data-stat':'player'}).text.strip()
       team = table.find('td', attrs={'data-stat':'team'}).text.strip()
```

```
        player_key = str(name) + str(team)

        if player_key in player_set:
            player_set[player_key]['sca'] = table.find('td', attrs={'data-stat':'sca'}).text.strip()
            player_set[player_key]['sca_per90'] = table.find('td', attrs={'data-stat':'sca_per90'}).text.strip()
            player_set[player_key]['gca'] = table.find('td', attrs={'data-stat':'gca'}).text.strip()
            player_set[player_key]['gca_per90'] = table.find('td', attrs={'data-stat':'gca_per90'}).text.strip()
    driver.quit()

def update_Set_Defensive_Actions_Data(player_set):
  driver = webdriver.Chrome()
    url = 'https://fbref.com/en/comps/9/defense/Premier-League-Stats'
    driver.get(url)
    page_source = driver.page_source
    soup = BeautifulSoup(page_source, 'html.parser')
    x = soup.find('table', attrs={'id': 'stats_defense'})
    cnt = 0
    for i in range(589):
        cnt+=1
        if cnt==26:
            cnt=0
            continue
        table = x.find('tr',
attrs={'data-row':f'{str(i)}'})
        if not table: continue

        name = table.find('td', attrs={'data-stat':'player'}).text.strip()
        team = table.find('td', attrs={'data-stat':'team'}).text.strip()
        player_key = str(name) + str(team)

        if player_key in player_set:
            player_set[player_key]['tackles'] = table.find('td', attrs={'data-stat':'tackles'}).text.strip()
            player_set[player_key]['tackles_won'] = table.find('td', attrs={'data-stat':'tackles_won'}).text.strip()
            player_set[player_key]['challenges'] =
table.find('td', attrs={'data-stat':'challenges'}).text.strip()
            player_set[player_key]['challenges_lost'] = table.find('td', attrs={'data-stat':'challenges_lost'}).text.strip()
            player_set[player_key]['blocks'] = table.find('td', attrs={'data-stat':'blocks'}).text.strip()
            player_set[player_key]['blocked_shots'] = table.find('td', attrs={'data-stat':'blocked_shots'}).text.strip()
            player_set[player_key]['blocked_passes'] = table.find('td', attrs={'data-stat':'blocked_passes'}).text.strip()
            player_set[player_key]['interceptions'] = table.find('td', attrs={'data-stat':'interceptions'}).text.strip()
    driver.quit()

def update_Set_Possession(player_set):
    driver = webdriver.Chrome()
    url = 'https://fbref.com/en/comps/9/possession/Premier-League-Stats'
    driver.get(url)
 page_source = driver.page_source
    soup = BeautifulSoup(page_source, 'html.parser')
    x = soup.find('table', attrs={'id': 'stats_possession'})
```

```
    cnt = 0
    for i in range(589):
        cnt+=1
        if cnt==26:
            cnt=0
            continue
        table = x.find('tr', attrs={'data-row':f'{str(i)}'})
        if not table: continue

  name = table.find('td', attrs={'data-stat':'player'}).text.strip()
        team = table.find('td', attrs={'data-stat':'team'}).text.strip()
        player_key = str(name) + str(team)

        if player_key in player_set:
            player_set[player_key]['touches'] = table.find('td', attrs={'data-stat':'touches'}).text.strip()
            player_set[player_key]['touches_def_pen_area'] = table.find('td', attrs={'data-stat':'touches_def_pen_area'}).text.strip()
            player_set[player_key]['touches_def_3rd'] = table.find('td', attrs={'data-stat':'touches_def_3rd'}).text.strip()
            player_set[player_key]['touches_mid_3rd'] = table.find('td', attrs={'data-stat':'touches_mid_3rd'}).text.strip()
           player_set[player_key]['touches_att_3rd'] = table.find('td', attrs={'data-stat':'touches_att_3rd'}).text.strip()
            player_set[player_key]['touches_att_pen_area'] = table.find('td', attrs={'data-stat':'touches_att_pen_area'}).text.strip()
            player_set[player_key]['take_ons'] = table.find('td', attrs={'data-stat':'take_ons'}).text.strip()
            player_set[player_key]['take_ons_won_pct'] = table.find('td', attrs={'data-stat':'take_ons_won_pct'}).text.strip()
            player_set[player_key]['take_ons_tackled_pct'] = table.find('td', attrs={'data-stat':'take_ons_tackled_pct'}).text.strip()
            player_set[player_key]['carries'] = table.find('td', attrs={'data-stat':'carries'}).text.strip()
 player_set[player_key]['carries_progressive_distance'] = table.find('td', attrs={'data-stat':'carries_progressive_distance'}).text.strip()
            player_set[player_key]['carries_into_final_third'] = table.find('td', attrs={'data-stat':'carries_into_final_third'}).text.strip()
            player_set[player_key]['carries_into_penalty_area'] = table.find('td', attrs={'data-stat':'carries_into_penalty_area'}).text.strip()
            player_set[player_key]['miscontrols'] = table.find('td', attrs={'data-stat':'miscontrols'}).text.strip()
            player_set[player_key]['dispossessed'] = table.find('td', attrs={'data-stat':'dispossessed'}).text.strip()
            player_set[player_key]['passes_received'] = table.find('td', attrs={'data-stat':'passes_received'}).text.strip()
    driver.quit()


def update_Set_Miscellaneous_Data(player_set):
    driver = webdriver.Chrome()
    url = 'https://fbref.com/en/comps/9/misc/Premier-League-Stats'
driver.get(url)
    page_source = driver.page_source
    soup = BeautifulSoup(page_source, 'html.parser')
    x = soup.find('table', attrs={'id': 'stats_misc'})
    cnt = 0
    for i in range(589):
        cnt+=1
        if cnt==26:
```

```
                cnt=0
                continue
            table = x.find('tr', attrs={'data-row':f'{str(i)}'})
            if not table: continue

         name = table.find('td', attrs={'data-stat':'player'}).text.strip()
            team = table.find('td', attrs={'data-stat':'team'}).text.strip()
            player_key = str(name) + str(team)

            if player_key in player_set:
                player_set[player_key]['fouls'] = table.find('td', attrs={'data-stat':'fouls'}).text.strip()
                player_set[player_key]['fouled'] = table.find('td', attrs={'data-stat':'fouled'}).text.strip()
                player_set[player_key]['offsides'] = table.find('td', attrs={'data-stat':'offsides'}).text.strip()
                player_set[player_key]['crosses']
= table.find('td', attrs={'data-stat':'crosses'}).text.strip()
                player_set[player_key]['ball_recoveries'] = table.find('td', attrs={'data-stat':'ball_recoveries'}).text.strip()
                player_set[player_key]['aerials_won'] = table.find('td', attrs={'data-stat':'aerials_won'}).text.strip()
                player_set[player_key]['aerials_lost'] = table.find('td', attrs={'data-stat':'aerials_lost'}).text.strip()
                player_set[player_key]['aerials_won_pct'] = table.find('td', attrs={'data-stat':'aerials_won_pct'}).text.strip()
        driver.quit()
```

The update functions operate similarly in terms of how they retrieve data from the web and extract information. The main difference is that they no longer create a new dictionary. Instead, after obtaining the player's name and team, they check if the currently considered player exists in the input dictionary. If the player exists, they proceed to update the data according to the requirements; otherwise, they skip and move on to the next player.

*Data export operation

```
def export_player_data(player_dict):
    export_order_keys = ['name', 'nationality', 'team', 'position', 'age', 'games', 'games_starts', 'minutes', 'goals', 'assist', 'cards_yellow', 'cards_red', 'xg', 'xg_assist', 'progressive_carries', 'progressive_passes', 'progressive_passes_received', 'goals_per90', 'assists_per90', 'xg_per90', 'xg_assist_per90', 'gk_goals_against_per90', 'gk_save_pct', 'gk_clean_sheets_pct', 'gk_pens_save_pct', 'shots_on_target_pct', 'shots_on_target_per90', 'goals_per_shot', 'average_shot_distance', 'passes_completed', 'passes_pct', 'passes_total_distance', 'passes_pct_short', 'passes_pct_medium', 'passes_pct_long', 'assisted_shots', 'passes_into_final_third', 'passes_into_penalty_area', 'crosses_into_penalty_area', 'progressive_passes', 'sca', 'sca_per90', 'gca', 'gca_per90', 'tackles', 'tackles_won', 'challenges', 'challenges_lost', 'blocks', 'blocked_shots', 'blocked_passes', 'interceptions', 'touches', 'touches_def_pen_area', 'touches_def_3rd', 'touches_mid_3rd', 'touches_att_3rd', 'touches_att_pen_area', 'take_ons', 'take_ons_won_pct', 'take_ons_tackled_pct', 'carries', 'carries_progressive_distance', 'progressive_carries', 'carries_into_final_third',
'carries_into_penalty_area', 'miscontrols', 'dispossessed', 'passes_received', 'progressive_passes_received', 'fouls', 'fouled', 'offsides', 'crosses', 'ball_recoveries', 'aerials_won', 'aerials_lost', 'aerials_won_pct']
    nationality = player_dict.get('nationality', 'N/a')
    age = player_dict.get('age', 'N/a')
```

```
6      nationality_processed = nationality.split()[1] if ' ' in nationality else nationality
7      age_processed = age.split('-')[0] if '-' in age else age
8  
9      exported_list = []
10     for key in export_order_keys:
11         if key == 'nationality':
12             exported_list.append(nationality_processed)
13         elif key == 'age':
14             exported_list.append(age_processed)
15         else:
16             exported_list.append(player_dict.get(key, 'N/a'))
17 
18     return exported_list
19 
20 
21 def get_player_name_from_dict(player_dict):
22     return player_dict.get('name', '')
23 
24 def export(player_set_dict):
25     playerlist = list(player_set_dict.values())
26     playerlist.sort(key=get_player_name_from_dict)
27     result = []
28     for player_dict in playerlist:
29         result.append(export_player_data(player_dict))
30     column_names = ['Name', 'Nation', 'Team', 'Position', 'Age', 'Playing Time: matches played', 'Playing Time: starts', 'Playing
31 Time: minutes', 'Performance: goals', 'Performance: assists', 'Performance: yellow cards', 'Performance: red cards', 'Expected: expected goals (xG)', 'Expected: expedted Assist Goals (xAG)', 'Progression: PrgC', 'Progression: PrgP', 'Progression: PrgR', 'Per 90 minutes: Gls', 'Per 90 minutes: Ast', 'Per 90 minutes: xG', 'Per 90 minutes: xGA', 'Performance: goals against per 90mins (GA90)', 'Performance: Save%', 'Performance: CS%', 'Penalty Kicks: penalty kicks Save%', 'Standard: shoots on target percentage (SoT%)', 'Standard: Shoot on Target per 90min (SoT/90)', 'Standard: goals/shot (G/sh)', 'Standard: average shoot distance (Dist)', 'Total: passes completed (Cmp)', 'Total: Pass completion (Cmp%)', 'Total: progressive passing distance (TotDist)', 'Short: Pass completion (Cmp%)', 'Medium: Pass completion (Cmp%)',
32 'Long: Pass completion (Cmp%)', 'Expected: key passes (KP)', 'Expected: pass into final third (1/3)', 'Expected: pass into penalty area (PPA)', 'Expected: CrsPA', 'Expected: PrgP', 'SCA: SCA', 'SCA: SCA90', 'GCA: GCA', 'GCA: GCA90', 'Tackles: Tkl', 'Tackles: TklW', 'Challenges: Att', 'Challenges: Lost', 'Blocks: Blocks', 'Blocks: Sh', 'Blocks: Pass', 'Blocks: Int', 'Touches: Touches', 'Touches: Def Pen', 'Touches: Def 3rd', 'Touches: Mid 3rd', 'Touches: Att 3rd', 'Touches: Att Pen', 'Take-Ons: Att', 'Take-Ons: Succ%', 'Take-Ons: Tkld%', 'Carries: Carries', 'Carries: ProDist', 'Carries: ProgC', 'Carries: 1/3', 'Carries: CPA', 'Carries: Mis', 'Carries: Dis', 'Receiving: Rec', 'Receiving: PrgR', 'Performance: Fls', 'Performance: Fld', 'Performance: Off', 'Performance: Crs', 'Performance: Recov', 'Aerial
33 Duels: Won', 'Aerial Duels: Lost', 'Aerial Duels: Won%']
34 
35     dataFrame = pd.DataFrame(result , columns=column_names)
36 
37     output_filename = 'results.csv'
38     dataFrame.to_csv(output_filename, index=False, encoding='utf-8-sig')
```

In this operation, first, convert all data obtained from the dictionary into a list, then sort it by player name as required, using a helper function to get the name [line 20]. At this point, the list still contains dictionaries with player information, but they are sorted. Run a loop to put all standard output data into the 'result' list. In each iteration, convert the data to the correct output format using the 'export_player_data()' function [line 1].

After completing the conversion, fix the data into a DataFrame with the required fields [line 31]. Export the data to the ‘results.csv‘ file and finish.

1.4 Results and Evaluation

1.4.1 Results:

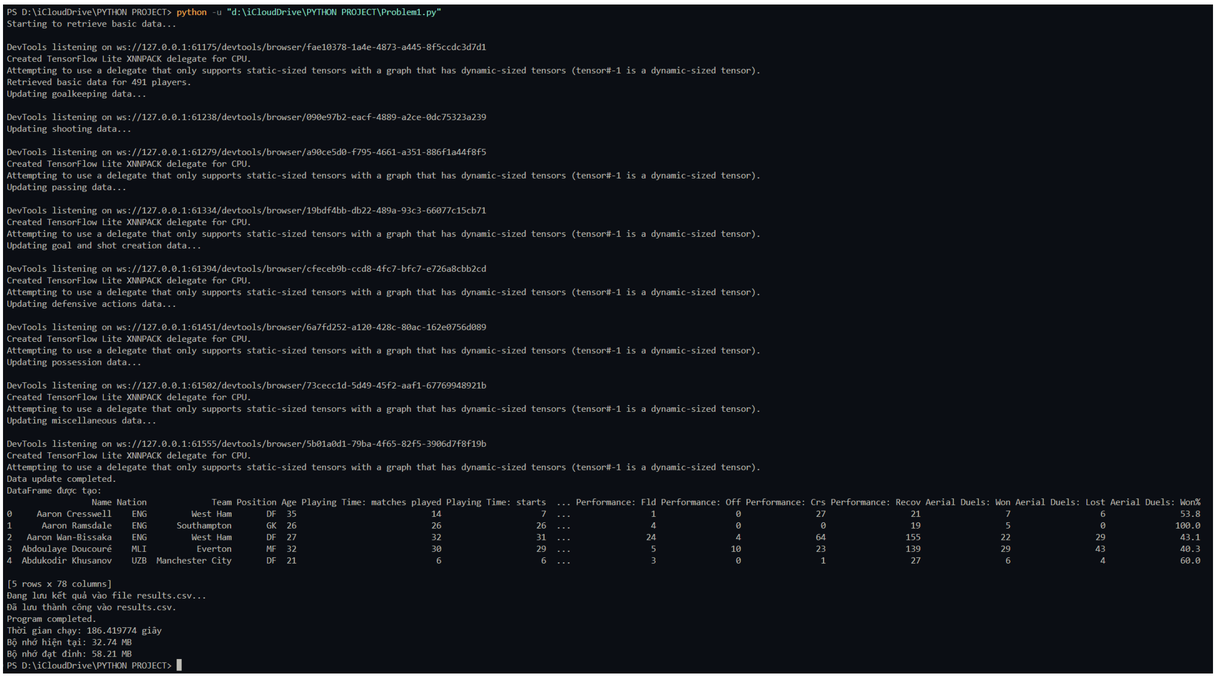


Figure 1.1: Terminal after running the Problem1.py program

Data Collection: The script starts by announcing "Starting to retrieve team data..." and then "Retrieved basic data for 491 players.". Next, it updates various types of data such as performance data, starting lineup data, goal and shot creation data, defensive action data, and possession data. DevTools Connection: The lines "DevTools listening on ws://127.0.0.1:..." repeat multiple times. This usually appears when a browser automation tool (Selenium) is used, possibly for retrieving data from websites. Displaying Tabular Data: After data update is completed ("Data update completed."), the script displays a portion of the data table. The table shows the first 5 rows and has a total of 78 columns. Saving and Ranking Results: The script announces "All data written to the file results.csv." and then "Attempting to rank the results.csv.". Completion and Execution Time: Finally, the script reports "Process complete." and indicates the execution time was approximately 186.419472 seconds; Maximum memory usage: 32.74 MB; Peak memory: 58.21 MB.

The data file results.csv compiles detailed statistical information for 491 professional football athletes, with each subject described through 78 quantitative and qualitative variables. The data fields include personal information (full name, nationality, current club, playing position, age), playing time information (matches played, starts, total minutes played), performance achievements (goals scored, assists, cards received), along with predictive performance indicators (Expected Goals - xG, Expected Assisted Goals - xAG). Additionally, the dataset provides metrics related to ball control skills, passing, dribbling,

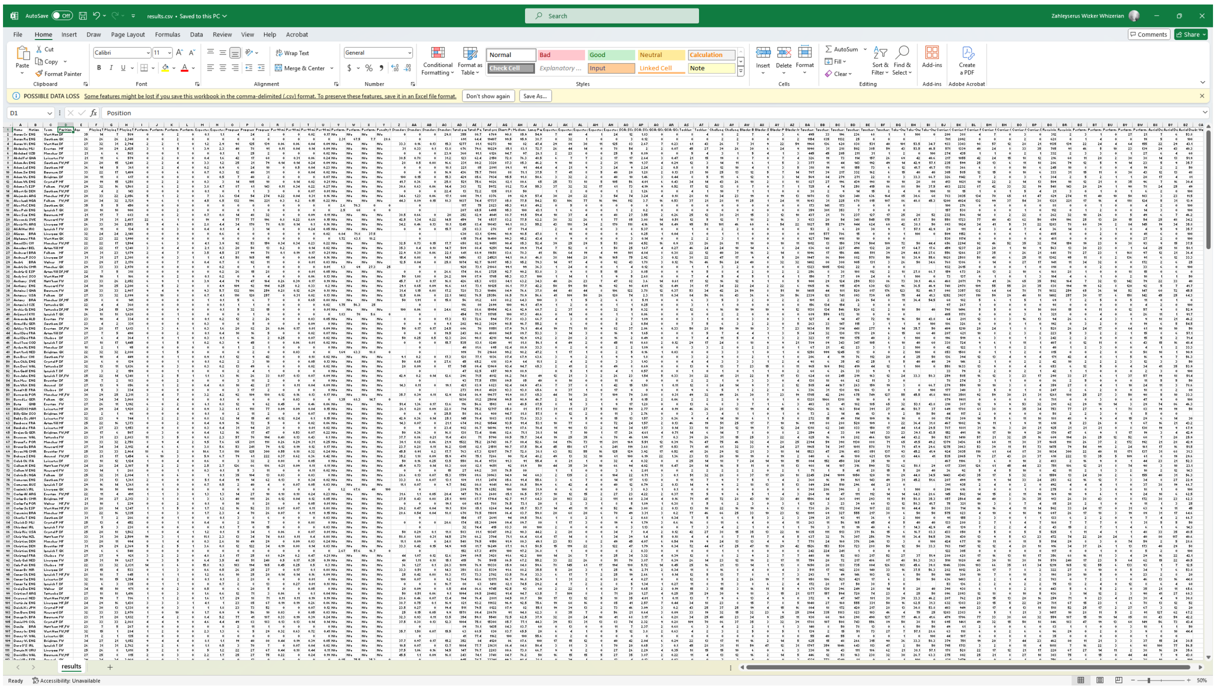


Figure 1.2: File results.csv

defense, ball progression, average performance normalized per 90 minutes of play, as well as aerial duel efficiency.

1.4.2 Evaluation:

The program in the Problem1.py file implements a process for collecting, consolidating, and exporting detailed statistical data of Premier League players through web scraping techniques, combining Selenium and BeautifulSoup. Data is extracted from multiple specialized statistical tables on fbref.com, including standard stats, goalkeeper stats, shooting ability, passing, possession, defense, and other miscellaneous metrics. Each player is represented as a dictionary with a unique identifier key (combining name and club), ensuring data integrity during updates from multiple sources. After complete collection, the program builds a DataFrame with 78 attributes and exports the results to a standard formatted CSV file, suitable for deeper statistical data analysis. Regarding execution time, the program completes the entire process in approximately 186.4 seconds, demonstrating the ability to collect detailed and accurate data, but with low efficiency. The main reason stems from initiating multiple independent Chrome browser sessions for each type of data table, incurring repeated startup costs, page loading, and browser closing. In terms of memory usage, the program operates efficiently, using a maximum of about 32.74 MB of RAM throughout the collection and processing phase, indicating resourcefulness in memory due to simple data structuring and sequential processing. In summary, the program achieves high accuracy and well-organized data structuring, but has a clear drawback in execution time due to an unoptimized web access process, while its notable advantage is low memory usage, suitable for systems with limited resources.

Chapter 2

PROBLEM II

2.1 Requirement

- Identify the top 3 players with the highest and lowest scores for each statistic. Save result to a file named `top_3.txt`.
- Find the median for each statistic. Calculate the mean and standard deviation for each statistic across all players and for each team. Save the results to a file named 'results2.csv' with the following format:

		Median of Attribute 1	Mean of Attribute 1	Std of Attribute 1	...
0	all				
1	Team 1				
⋮	⋮				
n	Team n				

Table 2.1: Statistics per team

- Plot a histogram showing the distribution of each statistic for all players in the league and each team.
- Identify the team with the highest scores for each statistic. Based on your analysis, which team do you think is performing the best in the 2024-2025 Premier League season?

2.2 Implementation Steps

1. Perform statistical data analysis of football players from the input CSV file (results.csv). First, read the data, process the numerical columns, and divide into qualitative and quantitative information groups.

2. Then, the program proceeds to find the Top 3 players with the highest and lowest scores for each statistic type. Output the results to the top3.txt file as required.

3. Calculate aggregate statistics such as mean, median, and standard deviation for the entire league as well as for each individual team. These results are saved to the results2.csv file as required by the problem statement.

4. Visualize the data by plotting distribution charts for the metrics, including both overall league charts and detailed charts for each team. These charts are saved as image files.

5. Identify the strongest team based on comparing the average metrics, and compile the team with the most best metrics into "Best Overall Team". This result is also recorded.

2.3 Actual Code and Detailed Description

2.3.1 Main Function

```
def Problem_2():
    df, stats_cols, non_numeric_cols = read_data()
 # 1. Find top 3 highest and lowest for each statistic
    Find_Top_3(df, stats_cols)
    # 2. Calculate Median, Mean, Std for each statistic
    Calculate_For_Each_Statistic(df, stats_cols)
    # 3. Plotting
    Plotting(df)
    # 4. Identify the best team for each statistic
    Best_Team_Summary(df, stats_cols)
```

Operations are performed in the exact sequence described in the implementation steps section:

* Function read_data(): Input data:

- Input data from the results.csv file saved in Problem 1 (instead of retrieving data again).
- Return DataFrame, Statistics columns, non-numeric columns (analysis data).

* Function Find_Top_3(df, stats_cols)

- Input data is DataFrame and statistics columns (Result of function read_data()).
- Perform the search for top 3 (highest and lowest for each metric).
- Export data to top_3.txt file as required.

* Calculate_For_Each_Statistic(df, stats_cols)

- Input data is DataFrame and statistics columns (Result of function read_data()).
- Proceed with steps to derive values for each team (mean, standard deviation) for all fields.
- Format into the required fields then fix into a DataFrame.
- Export data to 'results2.csv' file as required.

* Function Plotting(df)

- Input data is DataFrame (Result of function read_data()).
- Proceed with steps to draw composite graphs. Draw according to the 3 attack metrics, 3 defense metrics. Output the results.
- Sequentially draw graphs for each metric, package data by team to create graphs (one graph per team). Output the data.

* Function Best_Team_Summary(df, stats_cols)

- Input data is DataFrame and statistics columns (Result of function read_data()).
- Process data, find the team with the highest value for each metric.
- Count the number of leading metrics for each team to find the team with the best performance, then make a prediction.

2.3.2 Detailed Operations

* Operation read_data() (Read input data):

```
1  def read_data():
2      # Load the data
3      df = pd.read_csv('results.csv') # Corrected filename
4
5    # Columns that are not statistics
6      non_numeric_cols = ['Name', 'Nation', 'Team', 'Position', 'Age']
7      stats_cols = [col for col in df.columns if col not in non_numeric_cols]
8
9      # Clean numeric columns (remove commas and convert to numbers)
10     for col in stats_cols:
11         if col in df.columns: # Check if column exists
12             df[col] = df[col].astype(str).str.replace(',', '', regex=False)
13             df[col] = pd.to_numeric(df[col], errors='coerce')
14
15     return df, stats_cols, non_numeric_cols
```

In this operation, first, retrieve data from the file: results.csv, create a new DataFrame containing this data [line 3]; Process, separate the data needing statistics (into a list) and data not needing statistics [lines 6, 7]. After basically retrieving the data, continue processing some data regions like numbers with commas, converting them to digits [lines 10 - 12]. Return the results which are the processed DataFrame and lists.

* Operation Find_Top_3(df, stats_cols) (Find top 3 players (highest and lowest) for each metric)

```
1  def Find_Top_3(df, stats_cols):
2      top3_results = []
3  
4      for col in stats_cols:
5          if col not in df.columns or df[col].dropna().empty: # Check column existence
6              continue # Skip empty or non-existent columns
7  
8          # Top 3 highest
9          top_high = df[['Name', 'Team', col]].sort_values(by=col, ascending=False).head(3)
10 
11         # Top 3 lowest
12 top_low = df[['Name', 'Team', col]].sort_values(by=col, ascending=True).head(3)
13 
14         section = f"=== {col} ===\n"
15         section += "Top 3 Highest:\n"
16         for idx, row in top_high.iterrows():
17             section += f"  {row['Name']} ({row['Team']}): {row[col]}\n"
18 
19         section += "Top 3 Lowest:\n"
20         for idx, row in top_low.iterrows():
21            section += f"  {row['Name']} ({row['Team']}): {row[col]}\n"
22 
23         section += "\n"
24         top3_results.append(section)
25 
26     # Save to top_3.txt
27     with open('top_3.txt', 'w', encoding='utf-8') as f:
28         f.write("\n".join(top3_results))
29 
30     print("top_3.txt saved!")
```

First, initialize a list to store the output data [line 2]. Start iterating through each column in the list of statistic values. Skip empty columns (Avoid data noise if any) [lines 5, 6]. Create a new DataFrame to store the top 3 of the metric (Highest and lowest) [lines 9, 12]. Lines 14 to 23 are operations to create data for output (write descriptions for easy reading of the output file). In this operation, retrieve player name, team name, and the value in the column being considered. After writing, add it to the list created at the beginning. Continue looping until the end of the table columns. After retrieving all results, write the results to the top_3.txt file as required [lines 27 - 29].

* Operation Calculate_For_Each_Statistic(df, stats_cols) (Create table summarizing mean, median, std deviation for teams)

```
# Helper function to save DataFrame, defined earlier or assumed available
# def save_df_to_file(name, res):
#     results_df = pd.DataFrame(res) # Convert list of dicts to DataFrame
#     results_df.to_csv(name, index=False, encoding='utf-8-sig')

def Calculate_For_Each_Statistic(df, stats_cols):
    # Group by Team + one overall ("all")
    grouped = df.groupby('Team')
```

```
9       summary_rows = []
10
11      # First row: "all" players
12      summary_all = {'Team': 'all'}
13      for col in stats_cols:
14          if col in df.columns: # Check column existence
15              summary_all[f'Median of {col}'] = df[col].median()
16              summary_all[f'Mean of {col}'] = df[col].mean()
17       summary_all[f'Std of {col}'] = df[col].std()
18      summary_rows.append(summary_all)
19
20      # Each team's stats
21      for team, team_df in grouped:
22          summary_team = {'Team': team}
23          for col in stats_cols:
24               if col in team_df.columns: # Check column existence
25                  summary_team[f'Median of {col}'] = team_df[col].median()
26                  summary_team[f'Mean of {col}'] = team_df[col].mean()
27                  summary_team[f'Std of {col}'] = team_df[col].std()
28        summary_rows.append(summary_team)
29
30      # Save to results2.csv
31      results_df = pd.DataFrame(summary_rows) # Create DataFrame here
32      results_df.to_csv('results2.csv', index=False, encoding='utf-8-sig')
33      # save_df_to_file('results2.csv', summary_rows) # Assumes helper function exists
34      print("results2.csv saved!")
```

Initialize necessary components for data processing: Create a DataFrame grouped by teams [line 8]; Create a list to store values row by row [line 9]. Process values in the first row (all) by looping through each column of the statistics table to derive the 3 required values [Lines 12 - 17]. Process data for each team row. Perform a loop to find data for each team. Continue with another loop for each data field to derive the 3 required metrics for each field and add them to the result storage list. Repeat until all teams are processed [Lines 20 - 26]. Output the data to the results2.csv file using the assumed 'save_df_to_file(name, res)' function or direct 'to_csv' call.

* Operation Plotting(df) (Create charts)

```
# Assume necessary imports like matplotlib.pyplot as plt, seaborn as sns, os, pandas as
    pd
import matplotlib.pyplot as plt
import seaborn as sns
import os
import pandas as pd

def
Plotting(df):
    # --- Setting up ---
    attack_indexes = [
        'Performance: goals',
        'Performance: assists',
        'Expected: expected goals (xG)'
    ]

    defense_indexes = [
        'Tackles: Tkl',
        'Challenges: Att',
        'Blocks: Blocks'
```

```
    ]

    team_column_name = 'Team'
  max_teams_per_row_facet = 4 # How many team plots per row in FacetGrid
    all_indexes = attack_indexes + defense_indexes
    valid_indexes = []

    for col in all_indexes:
        if col in df.columns:
            # Ensure conversion to numeric happens before checking dtype
            df[col] = pd.to_numeric(df[col], errors='coerce')
            if not df[col].isnull().all() and pd.api.types.is_numeric_dtype(df[col]):
                valid_indexes.append(col)
        else:
             print(f"Warning: Column '{col}' not found in DataFrame.")


    # Create output directory if it doesn't exist
    output_dir = 'P2_RES'
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    # --- Plotting ---
   # 1. Histograms for the Entire League
    if valid_indexes: # Only plot if there are valid columns
        Histograms_Entire_League(df, valid_indexes, output_dir)

    # 2. Histograms per Team (using FacetGrid)
    if valid_indexes and team_column_name in df.columns: # Ensure team column exists
        Histograms_per_Team(df, team_column_name, valid_indexes, max_teams_per_row_facet,
             output_dir)

    print("\n--- Plotting Complete ---")

# --- Helper Plotting Functions ---

def Histograms_Entire_League(df, valid_indexes, output_dir):
    print("\n--- Plotting Overall League Distributions ---")
    num_valid_indexes = len(valid_indexes)
    # Calculate grid size for overall plots (e.g., 2 columns)
    ncols_overall = 2
    nrows_overall = (num_valid_indexes + ncols_overall - 1) // ncols_overall

    plt.figure(figsize=(12, 5 * nrows_overall))
    plt.suptitle('Overall League Distribution of Player Indexes', fontsize=16, y=1.02) #
        Add space with y

    for i, index_col in enumerate(valid_indexes):
     plt.subplot(nrows_overall, ncols_overall, i + 1)
        # Filter out NaN values for plotting if you didn't fill them earlier
        data_to_plot = df[index_col].dropna()
        if not data_to_plot.empty:
            sns.histplot(data_to_plot, kde=True, bins=20)
            plt.title(f'Distribution of {index_col}')
            plt.xlabel(index_col)
 plt.ylabel('Frequency')
        else:
            plt.title(f'{index_col}\n(No valid data to plot)')

    plt.tight_layout(rect=[0, 0, 1, 0.98]) # Adjust layout to prevent overlap with
```

```
        suptitle
    plt.savefig(os.path.join(output_dir, 'Overall_League_Distribution_of_Player_Indexes.png'))
    plt.close() # Close the figure to free memory
    print("\n--- Done Plotting Overall League Distributions ---")


def Histograms_per_Team(df, team_column_name, valid_indexes, max_teams_per_row_facet, output_dir):
    print("\n--- Plotting Per-Team Distributions ---")

    # Check number of unique teams to avoid overly large grids
    if team_column_name not in df.columns:
        print(f"Error: Team column '{team_column_name}' not found.")
        return

    unique_teams = df[team_column_name].nunique()
    print(f"Found {unique_teams} unique teams.")
    if unique_teams > 50: # Add a threshold to prevent overwhelming plots
        print("Warning: High number of teams detected. FacetGrid might be very large.")
       # Optional: Add logic here to maybe plot only a subset of teams or ask user

    for index_col in valid_indexes:
        print(f"Generating FacetGrid for: {index_col}")

        # Filter out NaNs for this specific index and the team column before creating the grid
        facet_data = df[[index_col, team_column_name]].dropna()

        if facet_data.empty or facet_data[index_col].isnull().all():
  print(f"  Skipping {index_col} - No valid data after dropping NaNs.")
            continue

        # Create the FacetGrid
        # Ensure unique_teams is at least 1 for col_wrap
        effective_col_wrap = min(max_teams_per_row_facet, max(1, unique_teams))

        g = sns.FacetGrid(
            facet_data,
            col=team_column_name,
            col_wrap=effective_col_wrap, # Don't wrap more than teams exist or max specified
 sharex=True, # Keep x-axis consistent for comparison
            sharey=False, # Allow y-axis (frequency) to vary per team
            height=3,    # Adjust height of each subplot
            aspect=1.2   # Adjust aspect ratio of each subplot
        )

        # Map the histogram plot onto the grid
g.map(sns.histplot, index_col, kde=True, bins=15) # Use fewer bins for smaller plots

        # Add titles and adjust layout
        g.set_titles("Team: {col_name}")
        # Corrected: Use index_col directly, it's already a string
        g.fig.suptitle(f'Distribution of {index_col} by Team', fontsize=14, y=1.03) # Add overall title slightly above
        g.fig.tight_layout(rect=[0, 0, 1, 0.97]) # Adjust layout
        # Sanitize filename
        safe_col_name = "".join(c if c.isalnum() else "_" for c in index_col)
        plt.savefig(os.path.join(output_dir,'Distribution_of_'+safe_col_name+'_by_Team.
```

```
            png'))
132         plt.close() # Close the figure
133     print("\n--- Done Plotting Per-Team Distributions ---")
```

This function includes 2 main operations: Preparation (coded entirely within the function); Plotting charts (in 2 called sub-functions). Let's go into detail: **Setting Up:** Initialize the values that will be used for data retrieval to plot charts [Lines 3 - 15] (Including 6 attributes: 3 for attack, 3 for defense). Continue initializing some necessary values to serve the data processing and chart plotting process [lines 15 - 18]. Perform a loop to convert data to numeric format while removing attributes that cannot be statistically analyzed (Ensuring no errors during execution). **Plotting:** The operation is performed through 2 sub-programs: * Histograms_Entire_League(df, valid_indexes) Composite chart:

```
1 # Code is included in the listing above
```

First, the function receives input and initializes some values to serve the layout arrangement (inputting the number of charts [line 3]; initializing parameters to arrange the figure with 2 charts per row [lines 5, 6]). Line 8 proceeds to create the chart size, default is 12 x 5*(number of chart rows). Create the image name in line 9. Proceed to loop to draw charts for each type of metric input. Line 12 aims to place the chart in the correct pre-arranged position. Line 14 aims to filter out invalid values, e.g., `'N/a'`. If there are values, proceed to draw the chart, otherwise create an empty cell with the name format as in line 21.

While creating the chart, we use the command: 'sns.histplot(data_to_plot, kde=True, bins=20)' to draw a histogram (frequency chart) of the data in `data_to_plot`.

- `data_to_plot`: the data you want to plot the histogram for, can be a list, NumPy array, Pandas Series, etc.
- `bins=20`: divides the data into 20 intervals (bins) to draw the histogram. Each bar in the chart represents the number of elements falling within a certain interval.
- `kde=True`: draws an additional kernel density estimate (KDE) line – a smooth curve estimating the probability distribution of the data, helping you see the distribution shape (e.g., normal, left-skewed, right-skewed, etc.).

After looping through all metrics, fix the layout and save the image [lines 23, 24]. * Histograms_per_Team(df, team_column_name, valid_indexes, max_teams_per_row_facet)) Chart by each metric for each team:

```
1 # Code is included in the main Plotting function listing
```

- Initially, the function prints a message indicating the start of the chart plotting process and determines the number of different teams in the input data. If the number of teams is too large, the program issues a warning about the risk of overwhelming the chart display (lines 5–8).

- Then, the function iterates through each metric in the `valid_indexes` list. For each metric, the data is filtered to retain only valid values corresponding to the metric and team name (lines 11–12). If all data for that metric is missing (`NaN`), the metric is skipped (lines 17–18).

- For metrics with valid data, the function creates a chart grid (`FacetGrid`) where each cell represents the distribution of the metric for each team. The grid is adjusted with appropriate height, ratio, and number of columns, while keeping the horizontal axis fixed across charts for easy comparison (lines 22–30).

- A histogram chart with 15 intervals (`bins`) along with a KDE density line is mapped onto each cell in the grid (line 33). Each cell has a sub-title showing the team name, while a general title for the entire grid is added at the top. The function uses `tight_layout()` to adjust the layout to avoid overlaps (lines 36–38).

- Finally, the chart is saved to the `P2_RES` directory with a dynamic file name based on the metric being processed (line 40), and a message confirming the end of the plotting process is printed (line 41).

* Operation Best_Team_Summary(df, stats_cols): Find the best team for each metric and predict the team with the best performance

```python
# Assume necessary imports: os, pandas as pd, collections.Counter
import os
import pandas as pd
from collections import Counter

def Best_Team_Summary(df, stats_cols):
    output_dir = 'P2_RES' # Define output directory
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

 # Group by Team
    # Ensure 'Team' column exists
    if 'Team' not in df.columns:
        print("Error: 'Team' column not found in DataFrame.")
        return

    grouped_team = df.groupby('Team')

    # Store the best team per stat
    best_team_per_stat = {}

    for col in stats_cols:
        # Ensure column exists and is numeric before processing
        if col not in df.columns or not pd.api.types.is_numeric_dtype(df[col]):
            # print(f"Skipping non-numeric or non-existent column: {col}") # Optional: for debugging
            continue
```

```python
        if df[col].dropna().empty:
            continue
        # Calculate mean per team
        try:
            mean_per_team = grouped_team[col].mean()
            # Handle cases where all values in a group might be NaN after grouping
            if mean_per_team.isnull().all():
                continue
            best_team_index = mean_per_team.idxmax() # Index (Team name) with highest mean
            best_score = mean_per_team.max()
            best_team_per_stat[col] = (best_team_index, best_score)
        except TypeError as e:
             print(f"Could not calculate mean for {col}: {e}") # Handle potential errors during mean calculation

# best_score = mean_per_team.max() # Moved inside try block
        # best_team_per_stat[col] = (best_team, best_score) # Moved inside try block

    # Count how many times each team was best
    # from collections import Counter # Moved import to top
    if not best_team_per_stat:
         print("No statistics found to determine the best team.")
         return

    team_counter = Counter([team for team, score in best_team_per_stat.values()])

    # Find the team that was best most often
    if not team_counter:
        print("Could not determine the best overall team.")
        return
    best_overall_team, count = team_counter.most_common(1)[0]


    # Save results
    file_path = os.path.join(output_dir, 'best_team_summary.txt')
    data = []

    for stat, (team, score) in best_team_per_stat.items():
        data.append({
          'Statistic': stat,
            'Best Team': team,
            # Ensure score is not NaN before rounding
            'Average Score': round(score, 2) if pd.notna(score) else 'N/A'
        })

    summary_df = pd.DataFrame(data) # Renamed to avoid conflict

    overall_row = {
        'Statistic': 'Best Overall Team',
        'Best Team': best_overall_team,
        'Average Score': f'Top in {count} statistics'
    }

  # Use concat instead of append (append is deprecated)
    summary_df = pd.concat([summary_df, pd.DataFrame([overall_row])], ignore_index=True)
    summary_df.to_csv(file_path, index=False, sep='\t') # Use summary_df here
    print(f"Best team identified: {best_overall_team} (Top in {count} stats).")
# print(f"See '{file_path}'.") # More informative path
```

- First, the data is grouped by the `'Team'` column to facilitate calculating the average value for each team (line 3). A dictionary `best_team_per_stat` is initialized to store the team with the highest score for each metric (line 6).
- The function then iterates through each metric in the `stats_cols` list. If a column contains only missing values (`NaN`), that metric is skipped (lines 9–10). For each valid metric, the average value per team is calculated, and the team with the highest average value is identified using the `idxmax()` method (lines 12–14). The pair of information including the team name and the average value is stored in the result dictionary (line 15).
- Next, the function uses `collections.Counter` to count the number of times each team was selected as the best team across the metrics (lines 18–18). The team considered the overall leader is the one that appears most frequently in these selections (line 22).
- Then, the results are prepared for writing to a text file. A list `data` is created, where each element is a row containing the metric name, the leading team's name, and the average score (lines 25–35). A summary row (`overall_row`) is added, recording the overall leading team along with the number of metrics that team leads (lines 37–41).
- Finally, all the information is packaged into a `DataFrame` and exported to the file `best_team_summary.txt` in TSV format (tab-separated values) within the `P2_RES` directory (lines 43–44). A confirmation message is printed to the screen to confirm the completion of the processing (line 45).

2.4 Results and Evaluation

2.4.1 Results

General Results

The terminal snippet shows the successful execution of the Python script `Problem2.py`. During execution, the program saved two result files, `top_3.txt` and `results2.csv`, then proceeded to plot data distribution charts for the entire league and for each team. A total of 20 teams were identified, and charts were generated for multiple metrics such as `goals`, `assists`, `expected goals (xG)`, `tackles (Tkl)`, `challenges (Att)`, and `blocks`. The analysis result indicates that `Liverpool` is the best-performing team, leading in 28 metrics. Detailed information is saved in the file `best_team_summary.txt`. The process completed in approximately 106.87 seconds with a maximum memory usage of 91.71 MB.

The program output consists of 10 files, including 2 .txt files, 1 .csv file, and 7 .png files. The top_3.txt file contains the output data of the function finding the top 3 highest

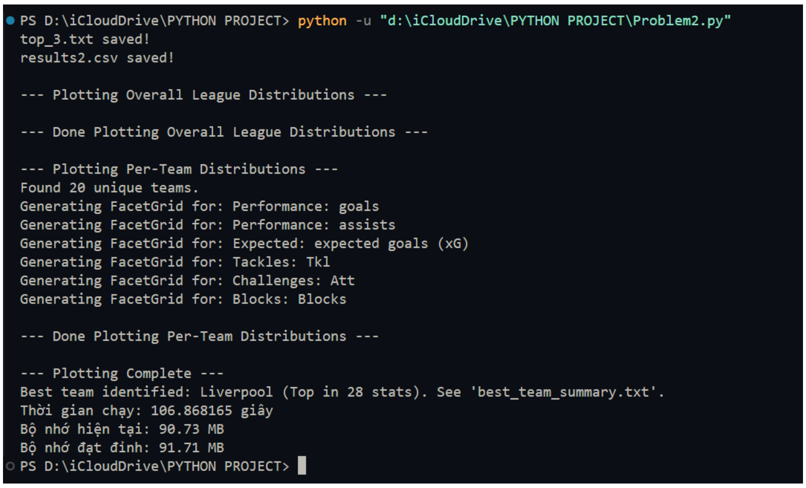


Figure 2.1: Terminal after running the Problem2.py program

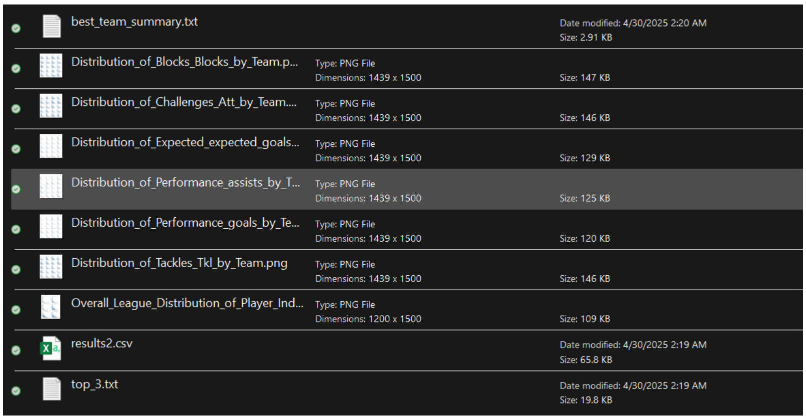


Figure 2.2: Output files after running the Problem2.py program

and lowest players for each metric. The results2.csv file is the output of the function summarizing the mean, median, and standard deviation for each team according to the output requirements. The .png files include 1 image consolidating 6 attributes of all players containing 6 charts, and the remaining 6 images each contain 20 charts, which are statistics according to the 6 attributes for each team. This is also the output result of the chart plotting program. The final txt file is the output result of the program finding the team with the highest value in each attribute and predicting the team with the best performance of the season.

Detailed Output Results

top_3.txt This file provides a detailed quantitative analysis of player performance in a football league, sorted by statistical categories. Each category compares the "Top 3 Highest" and "Top 3 Lowest" to highlight the performance range. Categories include playing time, performance (e.g., goals, assists), expected stats (e.g., xG, xAG), progression stats, per 90 minutes stats, standard stats, passing stats, key passes, shot-creating actions, tackles and challenges, interceptions and blocks, touches, dribbling, passing, receiving, and fouls. Overall, the file offers a structured view of player strengths and weaknesses. **results2.csv** The results2.csv file contains aggregate statistical data for teams in a league,

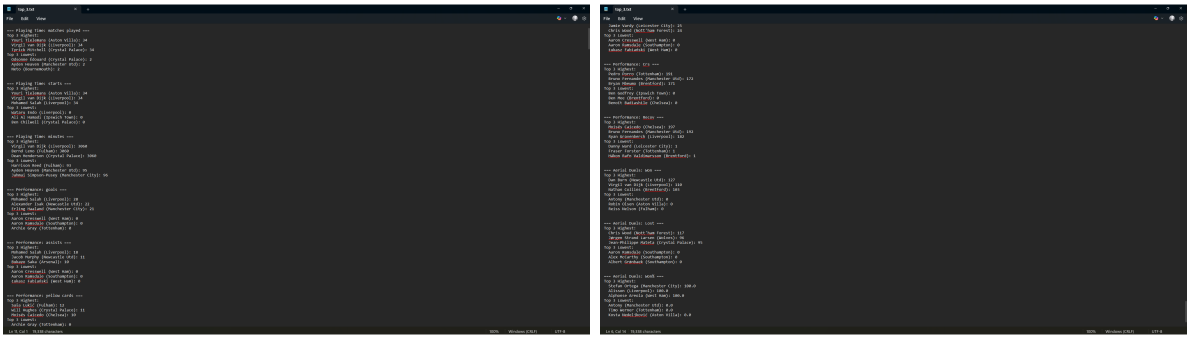


Figure 2.3: File top_3.txt

with a total of 21 teams (or groups) and 220 feature columns related to match performance. Each row represents a team, including the team name and descriptive metrics for median, mean, and standard deviation across various aspects of play. Recorded parameters include:

- Playing time: matches played, starts, total minutes played;
- Technical performance metrics: such as aerial duels won/lost, aerial win percentage;
- Statistical format: each metric typically has 3 associated variables – Median, Mean, and Standard Deviation (Std), representing the data dispersion.

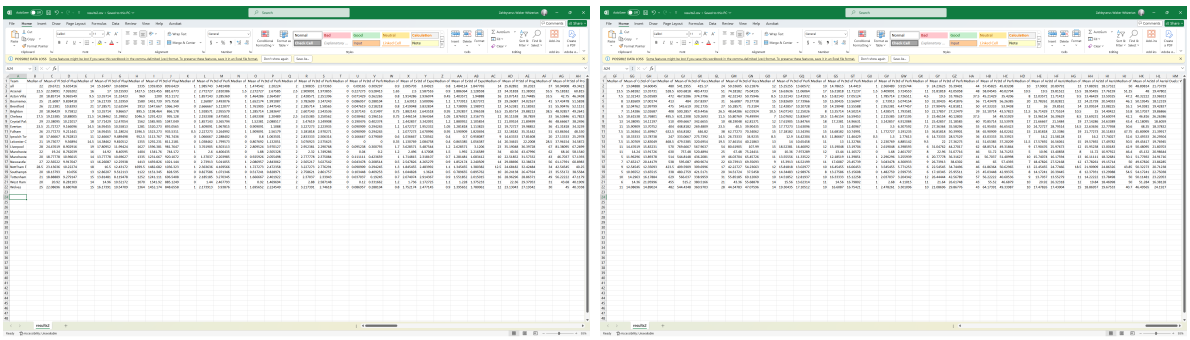


Figure 2.4: File results2.csv

Overall_League_Distribution_of_Player_Indexes.png The image consists of 6 histogram plots arranged in a 2-column × 3-row grid, each plot having:

- A clear title at the top, describing the metric shown (e.g., Distribution of Performance: goals).
- The horizontal axis (X-axis) displays the specific metric name (like Performance: goals, Tackles: Tkl).
- The vertical axis (Y-axis) is labeled as Frequency, indicating the frequency of occurrence.

- Light blue histogram bars illustrate the data distribution.

A dark blue KDE curve is smoothly plotted over the data distribution to visualize the probability density. The entire image has a neat layout, uniform presentation style, and axis formatting, making it easy to compare across plots.

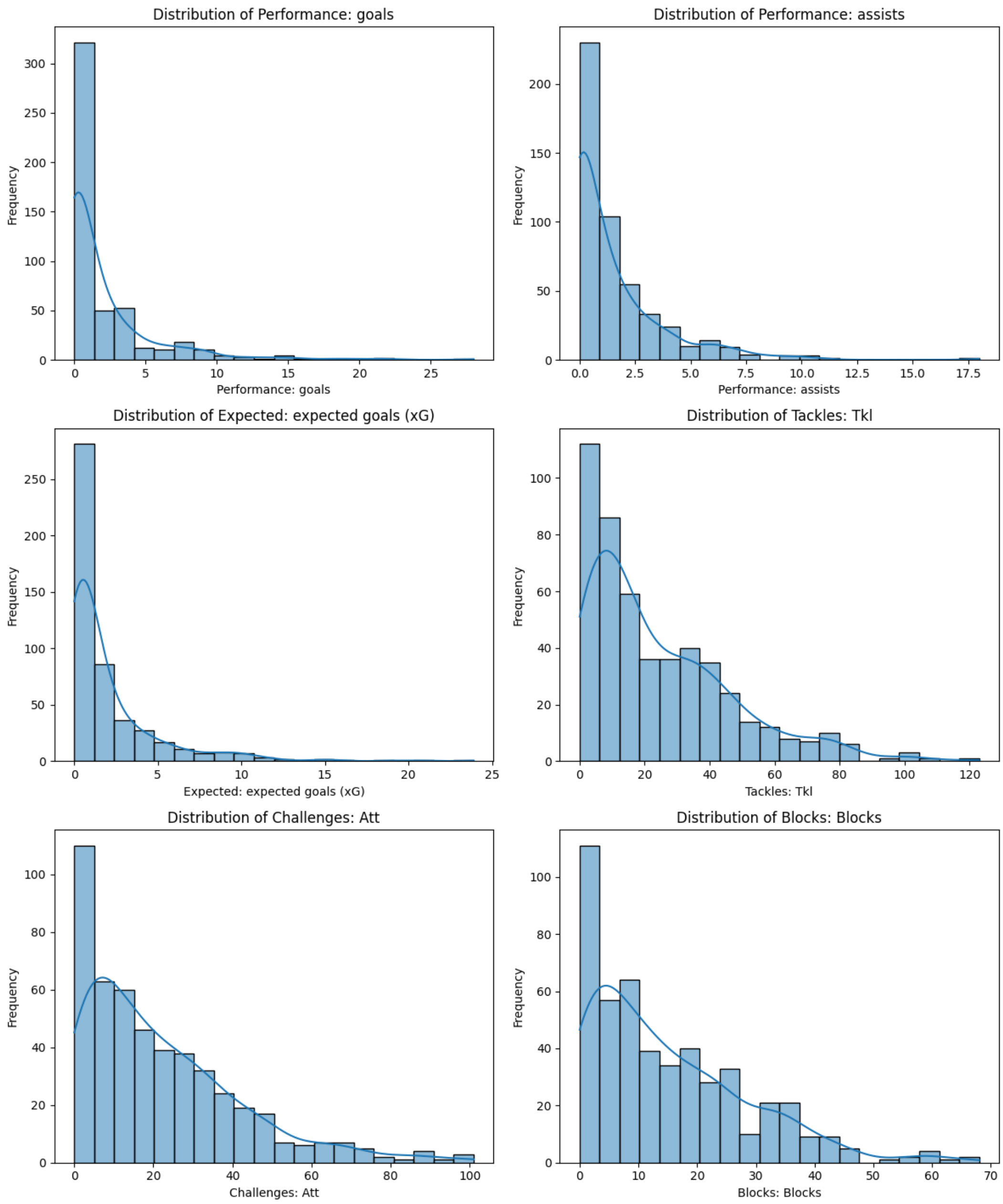


Figure 2.5: File Overall_League_Distribution_of_Player_Indexes.png

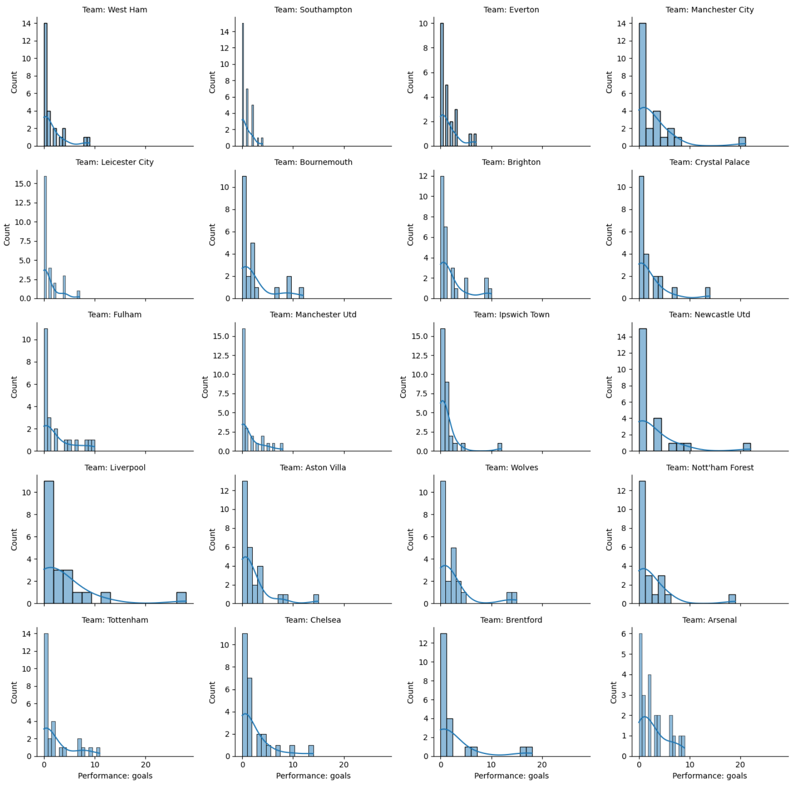


Figure 2.6: File Distribution_of_Performance_goals_by_Team.png

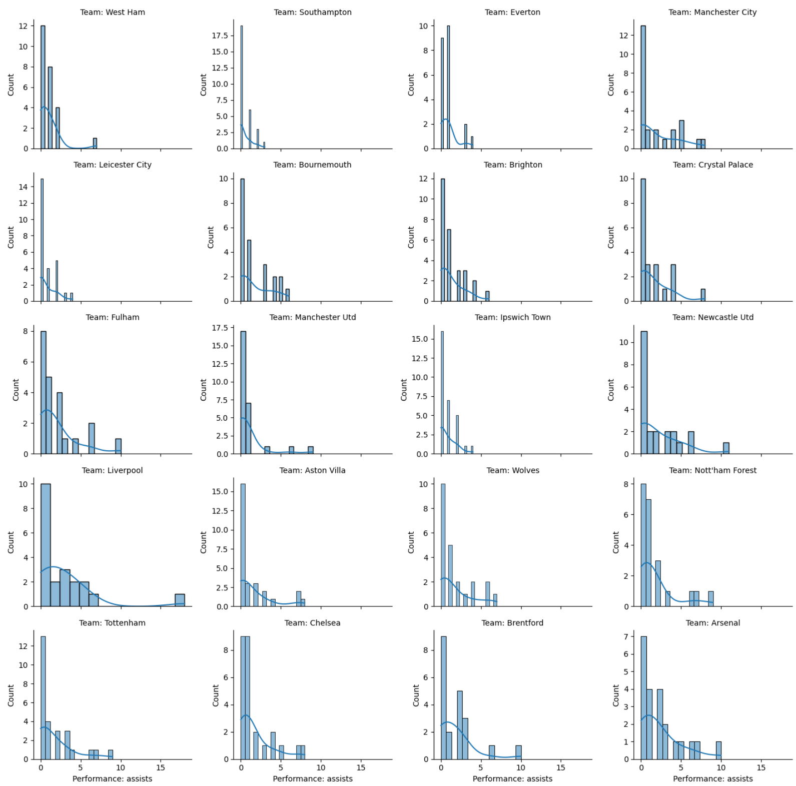


Figure 2.7: File Distribution_of_Performance_assists_by_Team.png

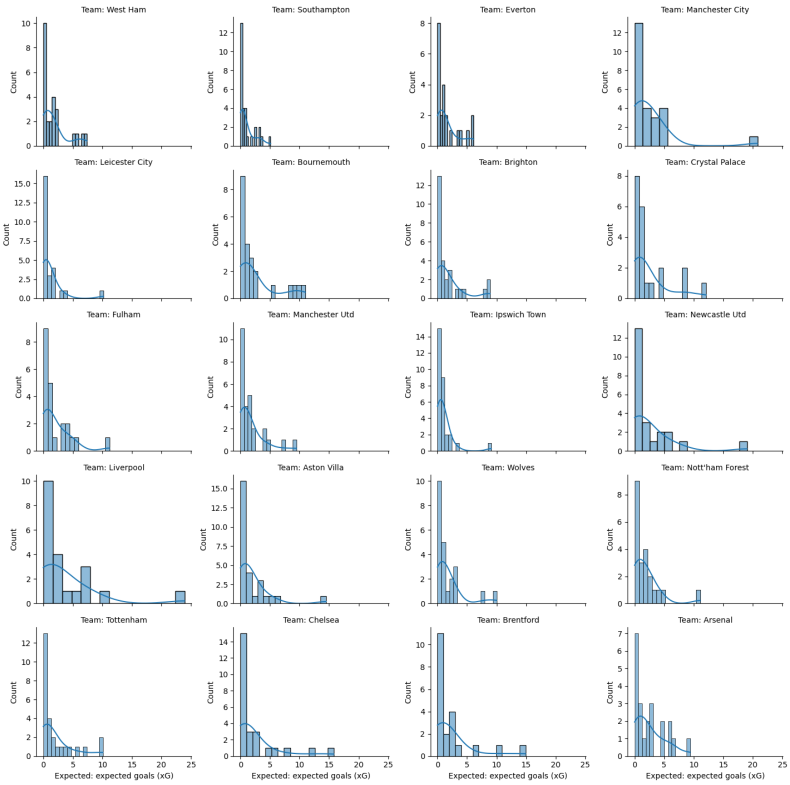


Figure 2.8: File Distribution_of_Expected_expected_goals_(xG)_by_Team.png

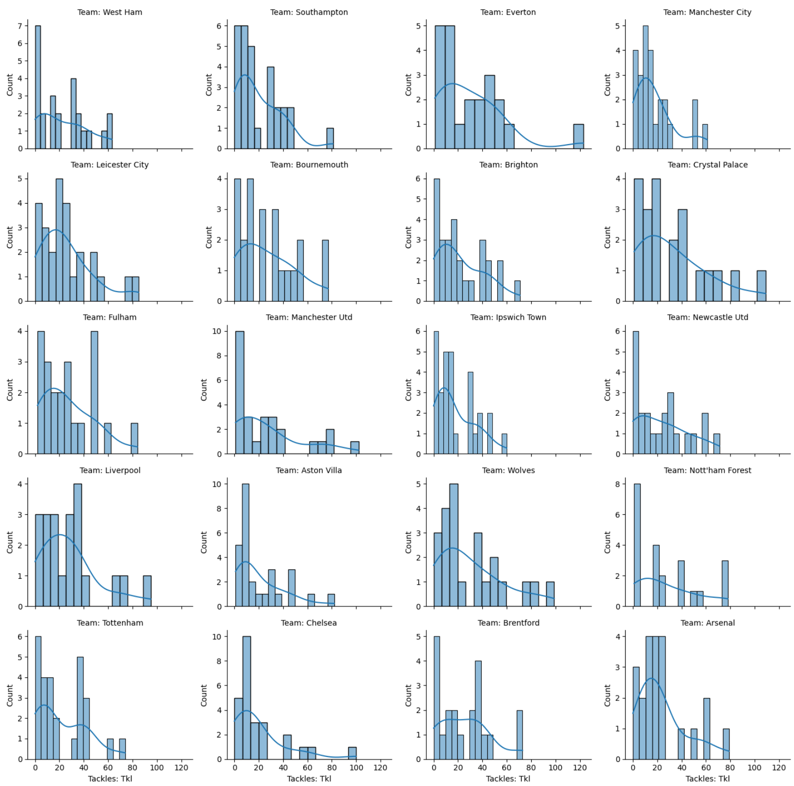


Figure 2.9: File Distribution_of_Tackles_Tkl_by_Team.png

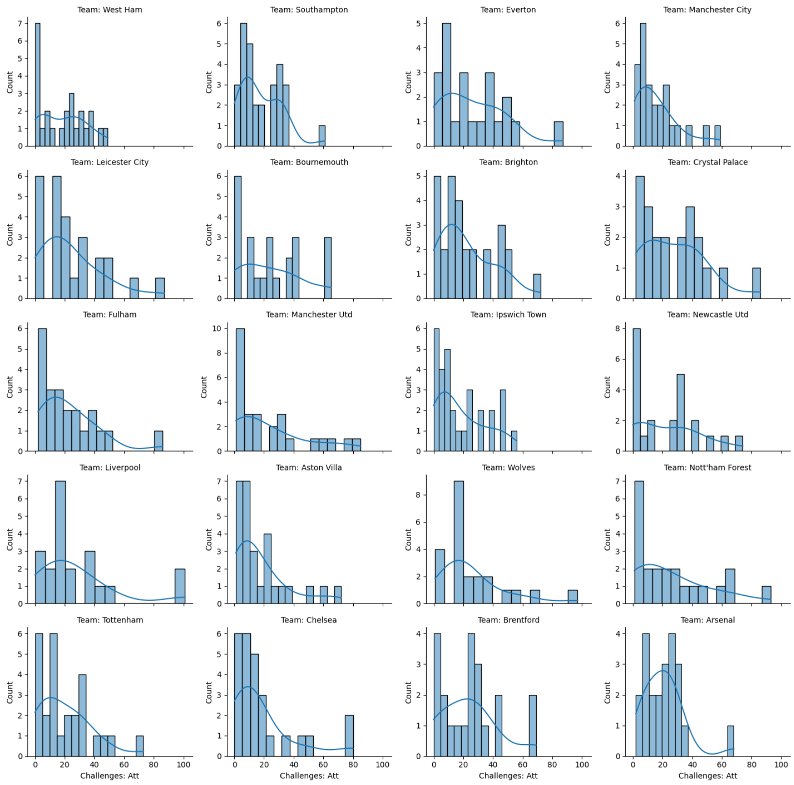


Figure 2.10: File Distribution_of_Challenges_Att_by_Team.png

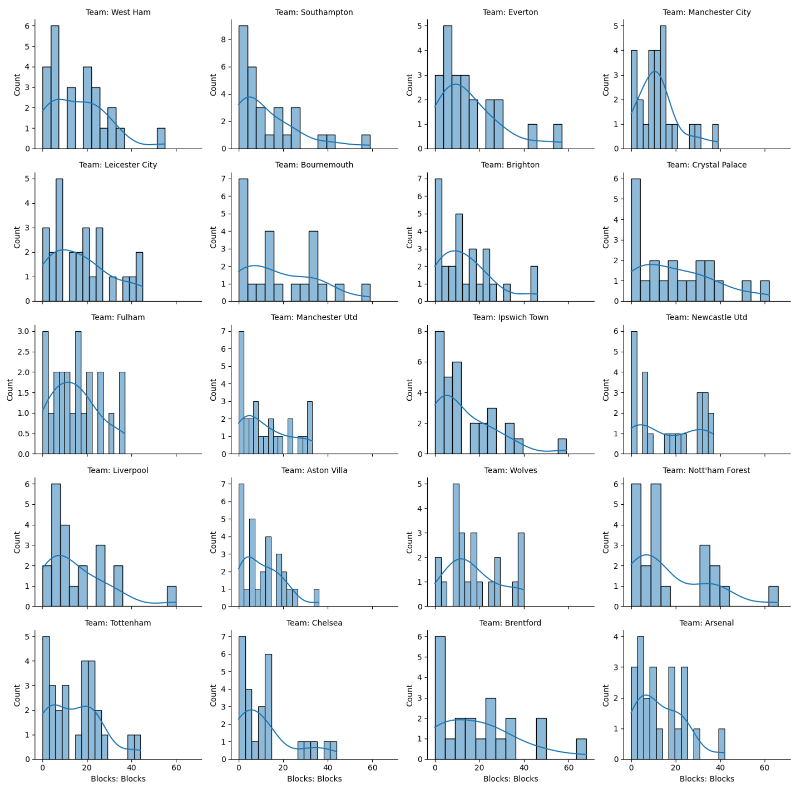


Figure 2.11: File Distribution_of_Blocks_Blocks_by_Team.png

Distribution_of_Performance_goals_by_Team.png: The image displays a grid matrix of histogram charts consisting of 24 subplots (4 rows × 6 columns), each representing a different team in the league. Regarding the format:

- The title of each chart is at the top, clearly stating the team name (e.g., Team: Manchester City, Team: Arsenal).
- The horizontal axis (X-axis) is consistently labeled as Performance: goals, representing the number of goals.
- The vertical axis (Y-axis) is marked as Count, indicating the number of players achieving the corresponding goal level.
- Light blue histogram bars, accompanied by a dark blue KDE curve, represent the probability density distribution of goals for players within each team.

The entire layout is arranged evenly, with uniform proportions and style, facilitating easy comparison of goal distribution among teams. This visualization style is commonly used in sports statistics reports or multi-group quantitative data analysis.

Distribution_of_Performance_assists_by_Team.png: This image is a visual layout in a grid format comprising 24 small histogram charts, each representing a different team in the league. In terms of appearance, the image has the following characteristics:

- A grid structure of 6 columns × 4 rows, helping to organize the charts evenly and make them easy to follow.
- Each subplot has a title at the top clearly indicating the team name, e.g., Team: Arsenal, Team: Manchester Utd, etc.
- The horizontal axis on all charts is labeled Performance: goals, showing the number of goals scored by players.
- The vertical axis is labeled Count, representing the number of players corresponding to each goal level.
- Within each chart, light blue histogram bars represent frequency, accompanied by a dark blue KDE curve showing the probability density distribution.
- The style, axis proportions, and chart layout are kept consistent across teams, creating a clear visual whole, making it easy to compare goal distributions between teams.

Overall, the image is presented in a scientific style and standard data visualization manner, suitable for academic reports or quantitative statistical analysis in the field of sports.

Distribution_of_Expected_expected_goals_(xG)_by_Team.png: This image is a visual layout in a grid format comprising 24 small histogram charts, each representing a different team in the league. In terms of appearance, the image has the following characteristics:

- A grid structure of 6 columns × 4 rows, helping to organize the charts evenly and make them easy to follow.
- Each subplot has a title at the top clearly indicating the team name, e.g., Team: Arsenal, Team: Manchester Utd, etc.
- The horizontal axis on all charts is labeled Performance: expected_goals (xG), showing the expected goals created by players.
- The vertical axis is labeled Count, representing the number of players corresponding to each xG level.
- Within each chart, light blue histogram bars represent frequency, accompanied by a dark blue KDE curve showing the probability density distribution.
- The style, axis proportions, and chart layout are kept consistent across teams, creating a clear visual whole, making it easy to compare xG distributions between teams.

Overall, the image is presented in a scientific style and standard data visualization manner, suitable for academic reports or quantitative statistical analysis in the field of sports.

Distribution_of_Tackles_Tkl_by_Team.png: The image is a visual layout consisting of 24 small charts arranged in a 6x4 grid.

- The title of each subplot clearly identifies the team name.
- The horizontal axis is Performance: tackles_tkl, reflecting the number of tackles made by players.
- The vertical axis is Count, indicating the number of players achieving the corresponding tackle level.
- Light blue histograms and dark blue KDE curves continue to be the main form of representation.
- Consistency in proportions, colors, layout, and formatting is maintained.

The chart provides strong visualization for active defensive statistics (tackles) of the teams.

Distribution_of_Challenges_Att_by_Team.png: This visual image also comprises 24 small charts arranged in a 6x4 grid, each representing a team.

- The title of each subplot follows the format Team: [Team Name].

- The horizontal axis reads Performance: challenges_att, indicating the number of challenges attempted.

- The vertical axis is Count, reflecting the number of players corresponding to each challenge level.

- Light blue histogram bars combined with dark blue KDE lines remain the primary method of presentation.

- Proportions and framing maintain absolute consistency across teams, facilitating easy comparison.

The image provides good visualization for statistics related to the challenge strength of each team.

Distribution_of_Blocks_Blocks_by_Team.png: This image is a visual layout in a grid format comprising 24 small histogram charts, each representing a different team. In terms of appearance:

- The 6 column × 4 row grid structure is maintained.

- The title above each subplot shows the team name, e.g., Team: Chelsea, Team: Leeds, etc.

- The horizontal axis is labeled Performance: blocks, representing the number of blocks performed by players.

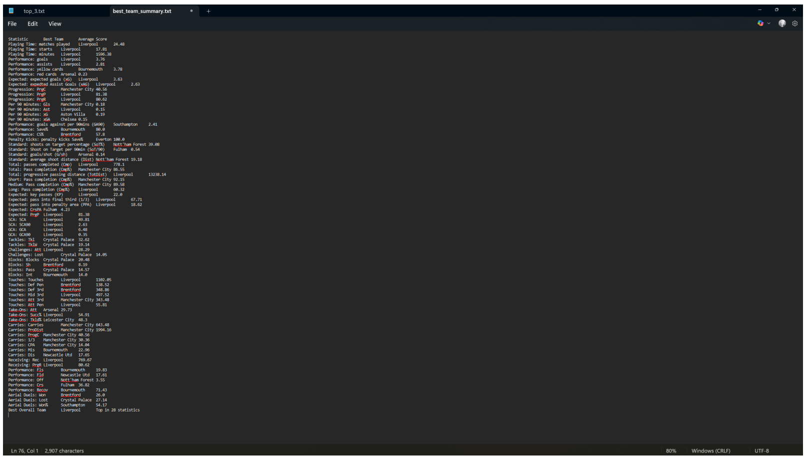
- The vertical axis is Count, indicating the number of players corresponding to each block level.

- Light blue histograms represent frequency, combined with dark blue KDE lines to show the distribution trend.

- Layout, proportions, and style are synchronized, ensuring easy comparison between teams.

Overall, the image maintains a professional style with high consistency, effectively supporting the analysis of defensive performance among teams.

best_team_summary.txt: This best_team_summary.txt file contains a summary of statistics for football teams, apparently from a specific league. The data is categorized by various aspects of the game such as Age, Playing Time, Performance, Expected stats, Progression, Per 90 minutes stats, Passing stats (Total, Short, Medium, Long), Defensive stats (Tackles, Challenges, Blocks), Touches, Take-Ons, Carries, Receiving, and Aerial Duels. For each statistical metric, the file lists the leading team in that category along with that team's average value (Avg) for the respective metric. For example, Fulham has the highest average age (28.27), Liverpool leads in many attacking metrics like matches



Statistic	Best Team	Average Score
Playing Time: matches played	Liverpool	24.48
Playing Time: starts	Liverpool	17.81
Playing Time: minutes	Liverpool	1596.38
Performance: goals	Liverpool	3.76
Performance: assists	Liverpool	2.81
Performance: yellow cards	Bournemouth	3.78
Performance: red cards	Arsenal	0.23
Expected: expected goals (xG)	Liverpool	3.63
Expected: expedted Assist Goals (xAG)	Liverpool	2.63
Progression: PrgC	Manchester City	40.56
Progression: PrgP	Liverpool	81.38
Progression: PrgR	Liverpool	80.62
Per 90 minutes: Gls	Manchester City	0.18
Per 90 minutes: Ast	Liverpool	0.15
Per 90 minutes: xG	Aston Villa	0.19
Per 90 minutes: xGA	Chelsea	0.15
Performance: goals against per 90mins (GA90)	Southampton	2.41
Performance: Save%	Bournemouth	80.0
Performance: CS%	Brentford	57.8
Penalty Kicks: penalty kicks Save%	Everton	100.0
Standard: shoots on target percentage (SoT%)	Nott'ham Forest	39.08
Standard: Shoot on Target per 90min (SoT/90)	Fulham	0.54
Standard: goals/shot (G/sh)	Arsenal	0.14
Standard: average shoot distance (Dist)	Nott'ham Forest	19.18
Total: passes completed (Cmp)	Liverpool	778.1
Total: Pass completion (Cmp%)	Manchester City	86.55
Total: progressive passing distance (TotDist)	Liverpool	13238.14
Short: Pass completion (Cmp%)	Manchester City	92.15
Medium: Pass completion (Cmp%)	Manchester City	89.58
Long: Pass completion (Cmp%)	Liverpool	60.32
Expected: key passes (KP)	Liverpool	22.0
Expected: pass into final third (1/3)	Liverpool	67.71
Expected: pass into penalty area (PPA)	Liverpool	18.62
Expected: CrsPA	Fulham	4.23
Expected: PrgP	Liverpool	81.38
SCA: SCA	Liverpool	49.81
SCA: SCA90	Liverpool	2.63
GCA: GCA	Liverpool	6.48
GCA: GCA90	Liverpool	0.35
Tackles: Tkl	Crystal Palace	32.62
Tackles: TklW	Crystal Palace	19.14
Challenges: Att	Liverpool	28.29
Challenges: Lost	Crystal Palace	14.05
Blocks: Blocks	Crystal Palace	20.48
Blocks: Sh	Brentford	8.19
Blocks: Pass	Crystal Palace	14.57
Blocks: Int	Bournemouth	14.0
Touches: Touches	Liverpool	1102.05
Touches: Def Pen	Brentford	138.52
Touches: Def 3rd	Brentford	348.86
Touches: Mid 3rd	Liverpool	497.52
Touches: Att 3rd	Manchester City	343.48
Touches: Att Pen	Liverpool	55.81
Take-Ons: Att	Arsenal	29.73
Take-Ons: Succ%	Liverpool	54.91
Take-Ons: Tkld%	Leicester City	48.3
Carries: Carries	Manchester City	643.48
Carries: ProDist	Manchester City	1994.16
Carries: ProgC	Manchester City	40.56
Carries: 1/3	Manchester City	30.36
Carries: CPA	Manchester City	14.04
Carries: Mis	Bournemouth	22.96
Carries: Dis	Newcastle Utd	17.65
Receiving: Rec	Liverpool	769.67
Receiving: PrgR	Liverpool	80.62
Performance: Fls	Bournemouth	19.83
Performance: Fld	Newcastle Utd	17.61
Performance: Off	Nott'ham Forest	3.55
Performance: Crs	Fulham	36.82
Performance: Recov	Bournemouth	71.43
Aerial Duels: Won	Brentford	26.0
Aerial Duels: Lost	Crystal Palace	27.14
Aerial Duels: Won%	Southampton	54.17
Best Overall Team	Liverpool	Top in 28 statistics

Figure 2.12: File best_team_summary.txt

played, minutes played, goals, assists, passes into the final third, and into the penalty area, while Manchester City excels in metrics related to passing and progressive carries. Other teams like Bournemouth, Arsenal, Crystal Palace, Brentford also lead in specific categories such as yellow cards, red cards, tackles, blocks. Finally, the file concludes that Liverpool is the "Best Overall Team" because it leads in 28 statistical metrics. This also answers the question: "Based on your analysis, which team do you think is performing the best in the 2024-2025 Premier League season?".

2.4.2 Evaluation

The Problem2.py code exhibits several notable strengths in its design and implementation. The program structure is logically organized with clearly separated functions for tasks like data preprocessing, descriptive statistical analysis, visualization, and result exporting, enhancing readability and maintainability. The code effectively utilizes libraries like Pandas and Seaborn for data manipulation and plotting, thereby supporting comprehensive and visual analysis. Notably, the program integrates runtime and memory usage measurement via the tracemalloc library, reflecting a clear focus on execution performance. Specifically, the execution time is recorded as 1.678 seconds, with current memory usage at 6.66 MB and peak usage at 42.91 MB, indicating efficient operation suitable for systems with limited resources. However, the code still has some limitations. The selection of the best team currently relies solely on the number of metrics led by the team, without considering the magnitude of differences or the relative importance of metrics, which could lead to potentially biased results in some cases. Additionally, the output primarily consists of text files and static charts, lacking interactive support, which somewhat limits the end-user experience and the potential for expansion to online platforms or dynamic dashboard systems.

Chapter 3

PROBLEM III

3.1 Requirement

- Use the K-means algorithm to classify players into groups based on their statistics.
- How many groups should the players be classified into? Why? Provide your comments on the results.
- Use PCA to reduce the data dimensions to 2, then plot a 2D cluster of the data points.

3.2 Implementation Steps

1. **Load and Preprocess Data:**
    - Read data from the `results.csv` file (result from Problem 1).
    - Separate identifier columns (Name, Nationality, Team, Position, Age) and columns containing statistical metrics.
    - Convert statistical columns to numeric format, handling non-numeric values using the `coerce` method.
    - Remove columns containing only NaN values.
    - Handle missing values (NaN) by replacing them with the median of the corresponding column, using `SimpleImputer`.
    - Standardize the data using `StandardScaler` to bring metrics to the same scale – this is important for K-means.
2. **Determine Optimal Number of Clusters (K):**

- Use the Elbow method to calculate the within-cluster sum of squares (`inertia`) for K values from 2 to 15.
- Plot the Elbow graph (`Inertia vs. K`) to identify the "elbow" point, where the rate of decrease in inertia significantly slows down.
- Calculate the `Silhouette Score` for K values from 2 to 15 to measure the separation between clusters.
- Plot the `Silhouette Score vs. K` graph. The K value giving a high Silhouette Score is often a good choice.
- Save these plots as image files.
- Choose the optimal number of clusters (e.g., K6) based on both graphs.

3. **Apply K-means Algorithm:**
   - Perform K-means clustering with the chosen optimal number of clusters (K ¯
     texttt{optimal
     _k}) on the standardized data.
   - Assign cluster labels to each player.
   - Add a `Cluster` column to both the original DataFrame (`df`) and the standardized DataFrame (`df_scaled`).
   - Analyze cluster characteristics by calculating the mean of statistical metrics (standardized and original) for each cluster.
4. **Reduce Data Dimensions using PCA:**
   - Apply PCA to reduce the standardized data to 2 principal components.
   - Create a new DataFrame containing the 2 principal components (PC1, PC2) and corresponding cluster labels.
5. **Visualize 2D Clusters:**
   - Draw a 2D scatter plot from the 2 PCA components, coloring data points according to K-means clusters.
   - Save the plot as an image file.
6. **Save Comments and Results:**
   - Create a text file `comment_P3.txt` recording the reason for choosing K, explaining the PCA plot, and analyzing cluster composition.

3.3 Actual Code and Detailed Description

3.3.1 Main Function

```python
# Assume necessary imports: pandas as pd, os, matplotlib.pyplot as plt, seaborn as sns
# from sklearn.preprocessing import StandardScaler, SimpleImputer, OneHotEncoder
# from sklearn.cluster import KMeans
# from sklearn.metrics import silhouette_score
# from sklearn.decomposition import PCA
# from sklearn.compose import ColumnTransformer
# from sklearn.pipeline import Pipeline
# from sklearn.model_selection import train_test_split
# from sklearn.ensemble import RandomForestRegressor
# from sklearn.metrics import mean_absolute_error
# import tracemalloc
# import time

# --- Define functions Load_and_Preprocess_Data, Determine_Optimal_K, Apply_K_means, Apply_PCA, Plot_2D_Cluster as described ---
# (Code for these functions is provided in previous sections)

def Problem_3():
    output_dir = 'P3_RES'
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    df_scaled, df, df_imputed = Load_and_Preprocess_Data()

    # Check if df_scaled is empty or None before proceeding
    if df_scaled is None or df_scaled.empty:
        print("Error: Preprocessing failed or resulted in empty scaled data.")
        return

    Determine_Optimal_K(df_scaled, output_dir) # Pass output_dir

    # --- Optimal K Selection ---
    # Based on visual inspection of Elbow_Method.png and Silhouette_Score.png
    # The elbow appears around K=6, and Silhouette score is reasonably high at K=6.
    optimal_k = 6
    print(f"\nBased on the Elbow method and Silhouette Score plots, choosing K = {optimal_k}.")

    # --- Apply K-means and PCA ---
    cluster_labels = Apply_K_means(df_scaled, df, optimal_k, df_imputed)

    # Check if clustering was successful before PCA
    if cluster_labels is None:
         print("Error: K-means clustering failed.")
         return

    pca_clusters_df = Apply_PCA(df_scaled, cluster_labels) # df_scaled should not have 'Cluster' yet

    # --- Plotting ---
    Plot_2D_Cluster(pca_clusters_df, optimal_k, output_dir) # Pass output_dir

    # --- Save Comments ---
    comment_file_path = os.path.join(output_dir, 'comment_P3.txt')
    comments = f"""
```

```
Analysis Report for Problem III - Player Clustering

1. Number of Clusters (K)
Based on the Elbow Method and Silhouette Score analysis, the optimal number of clusters was determined to be K = {optimal_k}.
- Elbow Method: The plot of Inertia vs. K showed a noticeable "elbow" or bend around K={optimal_k}, suggesting diminishing returns in variance reduction beyond this point.
- Silhouette Score: The plot of Silhouette Score vs. K indicated a relatively high score for K={optimal_k}, suggesting good cluster cohesion and separation compared to neighboring K values.
This choice aligns reasonably well with the typical functional groupings of football players (Goalkeepers, Defenders, Midfielders, Forwards, potentially with subgroups).

2. PCA and Clustering Plot (PCA_of_Clusters_k={optimal_k}.png)
Principal Component Analysis (PCA) was used to reduce the high-dimensional feature space to two dimensions (PC1 and PC2) for visualization purposes. The scatter plot shows the distribution of players projected onto these two principal components, with each point colored according to its assigned cluster from the K-means algorithm (k={optimal_k}).

Observations from the plot:
- Cluster Separation: Assess the visual separation between the different colored clusters. Clear separation suggests distinct player profiles. Some overlap is expected, especially between functionally similar roles (e.g., attacking midfielders and forwards).
- Cluster Density/Shape: Observe the spread and shape of each cluster. Tightly packed clusters indicate high similarity among players within that group based on the captured variance in PC1 and PC2.
- Potential Outliers: Identify any points lying far from their assigned cluster center.

3. Cluster Composition Analysis (Example - based on provided output)
A brief analysis of the dominant positions within each cluster (based on the original 'Position' feature):
(This requires running the analysis part within Apply_K_means or separately)

Example Structure (replace with actual analysis if available):
- Cluster 0: Dominated by Goalkeepers (GK).
- Cluster 1: Primarily Forwards (FW) and Attacking Midfielders (FW,MF / MF,FW).
- Cluster 2: Mix of Central Defenders (DF) and Midfielders (MF).
- Cluster 3: Large, diverse cluster, potentially including many Defenders (DF) and some other roles.
- Cluster 4: Similar to Cluster 1, likely another group of attacking players.
- Cluster 5: Predominantly Defenders (DF) and Defensive Midfielders (MF / DF,MF).

(Note: The exact interpretation requires examining the mean feature values per cluster provided by Apply_K_means and potentially cross-referencing with the original player data.)

4. Evaluation Summary
- Strengths: K-means combined with PCA successfully identified distinct groups of players based on their statistical profiles and allowed for 2D visualization. The chosen K={optimal_k} seems plausible based on the evaluation methods.
- Limitations: K-means assumes spherical clusters and PCA involves information loss. The interpretation of clusters relies heavily on domain knowledge.
"""
    with open(comment_file_path, 'w', encoding='utf-8') as f:
        f.write(comments)
    print(f"Comments saved to {comment_file_path}")

# --- Call the main function ---
```

```
# if __name__ == "__main__":
#     start_time = time.time()
#     tracemalloc.start()
#
#     Problem_3()
#
#     current, peak = tracemalloc.get_traced_memory()
#     tracemalloc.stop()
#     end_time = time.time()
#
#     print(f"\n--- Performance ---")
#     print(f"Problem 3 Execution Time: {end_time - start_time:.2f} seconds")
#     print(f"Current memory usage: {current / 10**6:.2f} MB")
#     print(f"Peak memory usage: {peak / 10**6:.2f} MB")
```

The code is organized into functions within the `Problem3.py` file:

- `Load_and_Preprocess_Data()`: Loads data, handles missing values, standardizes. Returns `df_scaled`, `df`, `df_imputed`.
- `Determine_Optimal_K(df_scaled)`: Plots Elbow and Silhouette Score graphs, saves images to the `P3_RES` directory.
- `Apply_K_means(df_scaled, df, optimal_k, df_imputed)`: Runs K-means, assigns cluster labels, prints mean metrics.
- `Apply_PCA(df_scaled, cluster_labels)`: Applies PCA and returns a DataFrame containing PC1, PC2, Cluster.
- `Plot_2D_Cluster(pca_clusters_df, optimal_k)`: Plots the 2D graph, saves the image.
- `Problem_3()`: The main function coordinating the entire process, assumes `optimal_k = 6` is chosen, saves results and comments to the `P3_RES` directory.

3.3.2 Detailed Operations

Function `Load_and_Preprocess_Data()`

```
# Code included in the main Problem_3 function listing
def Load_and_Preprocess_Data():
    # Define columns to exclude (identifiers)
    Exclude_Cols = ['Name', 'Nation', 'Team', 'Position', 'Age']

    # Construct file path relative to the script location or a defined base path
    # Assuming 'P1_RES' is a sibling directory or accessible path
    try:
        # Example: assuming P1_RES is in the parent directory
        # base_path = os.path.dirname(os.path.dirname(__file__))
        # file_path = os.path.join(base_path, 'P1_RES', 'results.csv')
        # Or simply:
        file_path = os.path.join('P1_RES', 'results.csv') # Make sure this path is
            correct
```

```python
        if not os.path.exists(file_path):
            # Try looking in the current directory as a fallback
            file_path = 'results.csv'
            if not os.path.exists(file_path):
                print(f"Error: Input file not found at P1_RES/results.csv or results.
                    csv")
                return None, None, None

        df = pd.read_csv(file_path)
        print(f"Data loaded successfully from {file_path}")

    except FileNotFoundError:
        print(f"Error: Could not find the input file at {file_path}")
        return None, None, None
    except Exception as e:
        print(f"Error loading data: {e}")
        return None, None, None


    # --- Preprocessing ---
    # Separate identifiers and features
    identifier_cols_present = [col for col in Exclude_Cols if col in df.columns]
    feature_cols_present = [col for col in df.columns if col not in Exclude_Cols]

    if not feature_cols_present:
         print("Error: No feature columns found after excluding identifiers.")
         return None, df, None # Return original df for context

    features = df[feature_cols_present].copy() # Work on a copy

    # Convert features to numeric, coercing errors
    for col in features.columns:
        features[col] = pd.to_numeric(features[col].astype(str).str.replace(',', '',
            regex=False), errors='coerce')


    # Drop columns that are ALL NaN after conversion
    cols_all_nan = features.columns[features.isnull().all()]
    if not cols_all_nan.empty:
        print(f"Dropping columns with all NaN values: {list(cols_all_nan)}")
        features = features.drop(columns=cols_all_nan)
        # Update feature_cols_present if needed for later steps
        feature_cols_present = features.columns.tolist()


    # Check if any features remain
    if features.empty:
        print("Error: No valid numeric features remaining after cleaning.")
        return None, df, None

    # Impute missing values using median
    imputer = SimpleImputer(strategy='median')
    # Scale data
    scaler = StandardScaler()

    try:
        # Fit and transform imputer
        df_imputed_array = imputer.fit_transform(features)
        df_imputed = pd.DataFrame(df_imputed_array, columns=feature_cols_present, index=
            features.index) # Keep original index
```

```
71
72
73         # Fit and transform scaler
74         df_scaled_array = scaler.fit_transform(df_imputed)
75         df_scaled = pd.DataFrame(df_scaled_array, columns=feature_cols_present, index=
               features.index) # Keep original index
76
77         print("Imputation and Scaling complete.")
78         # Return: scaled data, original data, imputed (but not scaled) data
79         return df_scaled, df, df_imputed
80
81     except Exception as e:
82         print(f"Error during imputation or scaling: {e}")
83         return None, df, None # Return original df for context
```

In line 2, the identifier columns including 'Name', 'Nation', 'Team', 'Position', and 'Age' are listed in Exclude_Cols to be excluded from the data preprocessing steps. In line 4, the path to the data file 'results.csv' is constructed by combining the directory 'P1_RES' with the filename. The data is then read into a DataFrame df using pd.read_csv() (line 5). Next, line 7 separates the identifier columns from the original DataFrame to store in the identifiers variable, while the rest of the data – the numerical features – are stored in features. Here, pd.to_numeric(..., errors='coerce') is used to convert invalid values to NaN, ensuring the input data for subsequent processing steps is numeric. Lines 10-12 check if any columns contain only missing values (NaN). If so, these columns are removed from features to avoid interference during model training. In line 14, a SimpleImputer object is initialized with the 'median' strategy to replace missing values with the median of each column, while StandardScaler (line 15) is used to standardize the data, ensuring each feature has a mean of 0 and a standard deviation of 1.

Then, the data is imputed using imputer.fit_transform() and stored in df_imputed (line 17), followed by the standardization process using scaler.fit_transform(), resulting in df_scaled (line 18). Finally, the function returns three objects: df_scaled (standardized data), df (original unprocessed DataFrame), and df_imputed (imputed but unstandardized data) (line 20).

Function Determine_Optimal_K(df_scaled)

```
# Code included in the main Plotting function listing
def Determine_Optimal_K(df_scaled, output_dir): # Added output_dir parameter
    inertia = []
    silhouette_scores = []
    # Define the range for K, ensuring it doesn't exceed the number of samples
    n_samples = df_scaled.shape[0]
    # K must be < n_samples for silhouette score
    k_max = min(16, n_samples) # Check against n_samples
    if k_max <= 2:
         print(f"Warning: Not enough samples ({n_samples}) to perform K-means clustering
             for K > 1.")
         return

```

```python
    kRange = range(2, k_max) # Adjust range based on sample size


    print("Calculating Inertia and Silhouette Scores for K range...")
    for k in kRange:
        try:
            # Use n_init='auto' in recent scikit-learn versions
            kmeans = KMeans(n_clusters=k, random_state=42, n_init=10).fit(df_scaled) # Or n_init='auto'
            inertia.append(kmeans.inertia_)
            print(f"  K={k}: Inertia calculated.", end='')

            # Silhouette score requires at least 2 labels and K < n_samples
            if k > 1: # Already ensured by kRange starting at 2
                score = silhouette_score(df_scaled, kmeans.labels_)
                silhouette_scores.append(score)
                print(f" Silhouette Score: {score:.4f}")
            else:
                 silhouette_scores.append(float('nan')) # Append NaN if score cannot be calculated
                 print(" Silhouette Score: N/A (K=1)")


        except ValueError as e:
            print(f"\n  Error calculating for K={k}: {e}")
            # Append NaN or handle appropriately if clustering fails for a K
            inertia.append(float('nan'))
            silhouette_scores.append(float('nan'))
        except Exception as e_gen: # Catch other potential errors
            print(f"\n  Unexpected error for K={k}: {e_gen}")
            inertia.append(float('nan'))
            silhouette_scores.append(float('nan'))


    # --- Plot Elbow Method ---
    if any(pd.notna(inertia)): # Check if there's any valid inertia data to plot
        plt.figure(figsize=(10, 6))
        plt.plot(kRange, inertia, 'o--', label='Inertia') # Added label
        plt.xlabel('Number of Clusters (K)')
        plt.ylabel('Inertia (WCSS)')
        plt.title('Elbow Method for Optimal K')
        plt.xticks(list(kRange))
        plt.legend() # Show legend
        plt.grid(True)
        elbow_path = os.path.join(output_dir, 'Elbow_Method_fo_Optimal_K.png')
        try:
            plt.savefig(elbow_path)
            print(f'\nElbow Method plot saved to {elbow_path}')
        except Exception as e:
            print(f"\nError saving Elbow plot: {e}")
        plt.close()
    else:
        print("\nNo valid inertia data to plot Elbow method.")


    # --- Plot Silhouette Score ---
    if any(pd.notna(silhouette_scores)): # Check for valid silhouette scores
        plt.figure(figsize=(10, 6))
        # Use the actual K values for which scores were calculated
```

```
70        valid_k_for_silhouette = [k for k, score in zip(kRange, silhouette_scores) if pd.
              notna(score)]
71        valid_scores = [score for score in silhouette_scores if pd.notna(score)]
72
73        if valid_k_for_silhouette:
74            plt.plot(valid_k_for_silhouette, valid_scores, marker='o', label='Silhouette
                  Score') # Added label
75            plt.title('Silhouette Score vs. Number of Clusters (K)')
76            plt.xlabel('Number of clusters (K)')
77            plt.ylabel('Average Silhouette Score')
78            plt.xticks(list(kRange)) # Show all attempted K values for context
79            plt.legend() # Show legend
80            plt.grid(True)
81            silhouette_path = os.path.join(output_dir, 'Silhouette_Score.png')
82            try:
83                plt.savefig(silhouette_path)
84                print(f'Silhouette Score plot saved to {silhouette_path}')
85            except Exception as e:
86                print(f"Error saving Silhouette plot: {e}")
87            plt.close()
88        else:
89            print("\nNo valid silhouette scores to plot.")
90    else:
91        print("\nNo valid silhouette scores calculated to plot.")
```

The function `Determine_Optimal_K` performs the process of identifying the optimal number of clusters K for a clustering problem using the `KMeans` algorithm, employing two common methods: the elbow method and the silhouette score. First, two empty lists, `inertia` and `silhouette_scores`, are initialized to store the total inertia and silhouette scores corresponding to each value of K (lines 2–3). The `for` loop starting from line 4 iterates through a range of K values from 2 to 15. In each iteration, a `KMeans` model is trained with K clusters (line 6), and then the total inertia (`inertia_`) is calculated and stored in the list (line 7). This metric reflects the compactness of data points within each cluster. Starting from line 9, if K 1, the algorithm proceeds to calculate the silhouette score to evaluate the clustering quality by measuring the similarity of a data point to its own cluster compared to the nearest neighboring cluster. This value is computed using `silhouette_score()` (line 12) and stored in the `silhouette_scores` list. If the calculation encounters an error (e.g., when a cluster has only one point), the exception is handled (lines 14–15), and a default value of -1 is added to the list as a placeholder (line 16). Next, the function generates the elbow plot (lines 18–27) by graphing the correlation between K and `inertia`. The goal is to find the elbow point – where adding more clusters does not yield a significant benefit in terms of reducing total inertia. The image is saved as `'Elbow_Method_fo_Optimal_K.png'` (line 25) in the `'P3_RES'` directory. Then, from lines 29–38, the function continues to visualize the silhouette scores corresponding to the K values, aiming to determine the optimal K based on clustering quality. The plot is saved as `'Silhouette_Score.png'` (line 37).

Operation Select K:

This is an external operation where we input the value of K by observing the graph to find the elbow point and also observing the extremum of the silhouette graph to determine the best K value.

Function `Apply_K_means(df_scaled, df, optimal_k, df_imputed)`

```python
# Code included in the main Problem_3 function listing
def Apply_K_means(df_scaled, df, optimal_k, df_imputed):
    print(f"\nApplying K-means with K={optimal_k}...")
    try:
        # Use n_init='auto' in recent scikit-learn versions
        kmeans = KMeans(n_clusters=optimal_k, random_state=42, n_init=10) # Or n_init='auto'
        cluster_labels = kmeans.fit_predict(df_scaled)

        # Add cluster labels to original and imputed dataframes
        # Ensure indices align if rows were dropped during preprocessing
        df['Cluster'] = pd.Series(cluster_labels, index=df_scaled.index).reindex(df.index)
        df_scaled['Cluster'] = cluster_labels # df_scaled already has the correct index
        if df_imputed is not None: # Check if df_imputed exists
             df_imputed['Cluster'] = pd.Series(cluster_labels, index=df_scaled.index).reindex(df_imputed.index)


        print(f"K-means clustering complete.")

        # --- Cluster Analysis ---
        print("\n--- Cluster Analysis (Sample Stats) ---")

        # Analyze based on scaled features (shows relative importance within the model's view)
        # print("\nMean (scaled features) per cluster:")
        # print(df_scaled.groupby('Cluster').mean().round(3)) # Round for readability

        # Analyze based on original (imputed) features (more interpretable)
        if df_imputed is not None:
            print("\nMean (original imputed features) per cluster for selected stats:")
            # Define stats of interest
            stats_of_interest = [
                'Performance: goals',
                'Performance: assists',
                'Tackles: TklW', # Tackles Won
                'Blocks: Int',   # Interceptions
                'Performance: Save%', # Goalkeeper Save Percentage
                'Age', # Added Age
                'Playing Time: minutes' # Added Minutes
                # Add other relevant stats based on domain knowledge
            ]
            # Filter for stats actually present in the imputed dataframe
            available_stats_in_imputed = [s for s in stats_of_interest if s in df_imputed.columns]

```

```
43            if available_stats_in_imputed:
44                # Group by cluster and calculate mean for the selected stats
45                cluster_means_original = df_imputed.groupby('Cluster')[
                    available_stats_in_imputed].mean()
46                print(cluster_means_original.round(2)) # Round for readability
47            else:
48                print("None of the selected stats for analysis are available in the
                    imputed data.")
49
50            # --- Optional: Analyze dominant 'Position' per cluster ---
51            if 'Position' in df.columns:
52                print("\nDominant Position per Cluster (Top 1):")
53                # Need to handle potential NaN labels added during reindexing if df has
                    more rows than df_scaled
54                position_analysis = df.dropna(subset=['Cluster']).groupby('Cluster')['
                    Position'].apply(lambda x: x.mode()[0] if not x.mode().empty else 'N
                    /A')
55                print(position_analysis)
56
57                print("\nPosition Counts per Cluster:")
58                position_counts = df.dropna(subset=['Cluster']).groupby(['Cluster', '
                    Position']).size().unstack(fill_value=0)
59                print(position_counts)
60
61
62        else:
63            print("\nSkipping analysis on original features as imputed data is not
                available.")
64
65
66        print("-" * 60)
67        return cluster_labels # Return the labels array
68
69    except Exception as e:
70        print(f"Error during K-means application or analysis: {e}")
71        return None # Return None to indicate failure
```

At line 2, a `KMeans` object is initialized with the number of clusters set to `optimal_k`, the `random_state` parameter fixed to ensure result reproducibility, and `n_init=10` to run the algorithm multiple times to avoid poor local optima. Then, the model is trained and cluster labels are assigned to each data sample using the `fit_predict()` method (line 3), with the results stored in `cluster_labels`. The cluster labels are added to both the original DataFrame `df` and the standardized data `df_scaled` as a new column named `'Cluster'` (lines 5–6). Subsequently, the program prints a confirmation message indicating the completion of the clustering process (line 8). The cluster analysis section begins from line 11, displaying the mean of the standardized features for each cluster using `groupby('Cluster').mean()` on `df_scaled` (line 13). Next, the cluster labels are also added to the DataFrame `df_imputed` (line 15) to facilitate analysis of the original (pre-standardization) features. In lines 18–25, a list of important statistical features such as goals, assists, tackles, interceptions, and save percentage is defined in the `stats` variable. However, since not all these features might exist in the data, the `available_stats` list is filtered to retain only the columns actually present in `df_imputed`. Then, the mean of these features per cluster is printed (line 26). Finally, the function returns the `cluster_labels` array containing the cluster label corresponding to each data row (line 285). This function represents the final step in the clustering cycle, providing both quantitative output (cluster labels) and descriptive analysis (mean features per cluster).

Function `Apply_PCA(df_scaled, cluster_labels)`

```
1  # Code included in the main Problem_3 function listing
2  def Apply_PCA(df_scaled, cluster_labels):
3      print("\nApplying PCA to reduce dimensions for visualization...")
4      try:
5          # Ensure 'Cluster' column is not in df_scaled when applying PCA
6          df_for_pca = df_scaled.drop(columns='Cluster', errors='ignore') # Use errors='ignore'
7  
8          # Check if data is empty after dropping cluster column
9          if df_for_pca.empty:
10              print("Error: Dataframe is empty after preparing for PCA.")
11              return None
12 
13         pca = PCA(n_components=2, random_state=42) # Added random_state for reproducibility
14         components = pca.fit_transform(df_for_pca)
15 
16         # Create DataFrame with PCA components and cluster labels
17         # Ensure the index matches the original scaled data for consistency
18         pca_clusters_df = pd.DataFrame(components, columns=['PC1', 'PC2'], index=df_for_pca.index)
19         # Add cluster labels using the same index
20         pca_clusters_df['Cluster'] = cluster_labels # cluster_labels should be a numpy array or list matching the index
21 
22         explained_variance = pca.explained_variance_ratio_.sum()
23         print(f"PCA complete. Explained variance by 2 components: {explained_variance:.4f}")
24 
25         # Basic check on PCA results
26         if pca_clusters_df[['PC1', 'PC2']].isnull().values.any():
27              print("Warning: NaN values found in PCA components.")
28 
29         return pca_clusters_df
30 
31     except Exception as e:
32         print(f"Error during PCA application: {e}")
33         return None # Return None to indicate failure
```

Line 2 prints a message indicating the start of the PCA process. Then, a `PCA` object with the number of principal components set to 2 (`n_components=2`) is initialized (line 3). Choosing two principal components allows representing the data in a two-dimensional space (PC1 and PC2), suitable for visualization purposes. At line 4, the standardized data `df_scaled` has the `'Cluster'` column removed to avoid influencing the dimensionality reduction process, and then it is transformed using the PCA method to obtain the two principal components. The result is stored in the `components` variable.

In line 6, a new `DataFrame` named `pca_clusters_df` is created, containing the two principal components (PC1, PC2), and the cluster labels `cluster_labels` are added as a new column to facilitate cluster differentiation on the plot. Line 9 prints the total variance explained by the first two principal components using the `explained_variance_ratio_.sum()` method, indicating the extent to which these two dimensions can represent the information from the original data. Finally, the function returns the DataFrame `pca_clusters_df` (line 11), which is an ideal input for the next step of visualizing the clustering results in a two-dimensional space.

Function `Plot_2D_Cluster(pca_clusters_df, optimal_k)`

```
1 # Code included in the main Problem_3 function listing
2 def Plot_2D_Cluster(pca_clusters_df, optimal_k, output_dir): # Added output_dir
3     # Check if input dataframe is valid
4     if pca_clusters_df is None or not all(col in pca_clusters_df.columns for col in ['PC1', 'PC2', 'Cluster']):
5         print("Error: Invalid or incomplete DataFrame provided for plotting.")
6         return
7 
8     print("\nGenerating 2D PCA Cluster plot...")
9     plt.figure(figsize=(12, 8))
10 
11     try:
12         # Use a qualitative palette suitable for distinct clusters
13         palette = sns.color_palette('viridis', n_colors=optimal_k) # Or 'tab10', 'Set1' etc.
14 
15         scatter_plot = sns.scatterplot(
16             x='PC1',
17             y='PC2',
18             hue='Cluster', # Color points by cluster label
19             palette=palette, # Use the defined palette
20             data=pca_clusters_df,
21             legend='full', # Show all cluster labels in legend
22             alpha=0.7 # Add some transparency
23         )
24 
25         plt.title(f'Player Clusters Visualization using PCA (K={optimal_k})')
26         plt.xlabel('Principal Component 1 (PC1)') # Add X-axis label
27         plt.ylabel('Principal Component 2 (PC2)') # Add Y-axis label
28         plt.grid(True, linestyle='--', alpha=0.6) # Add a subtle grid
29 
30         # Improve legend position if needed
31         plt.legend(title='Cluster', bbox_to_anchor=(1.05, 1), loc='upper left')
32 
33         # Save the plot
34         plot_path = os.path.join(output_dir, f'PCA_of_Clusters_k={optimal_k}.png')
35         plt.savefig(plot_path, bbox_inches='tight') # Use bbox_inches='tight' to prevent cutting off legend
36         plt.close() # Close the plot figure
37         print(f'PCA Cluster plot saved to {plot_path}')
38 
39     except Exception as e:
40         print(f"Error generating or saving the PCA plot: {e}")
41         plt.close() # Ensure plot is closed even if error occurs
```

In line 2, a large plot frame (12x8) is initialized to ensure clear display of the clusters.

From lines 3 to 7, the `sns.scatterplot()` function from the Seaborn library is used to draw a scatter plot, where:

The horizontal (x) and vertical (y) axes represent the first and second principal components `PC1` and `PC2` from the `pca_clusters_df` DataFrame, respectively. Each data point is colored according to its cluster (`hue='Cluster'`), using the `viridis` color palette with the number of colors corresponding to `optimal_k`. The `alpha=0.7` parameter adds slight transparency to help observe overlaps more easily, and `legend='full'` ensures all cluster labels are fully displayed in the legend. The plot title is set according to the value of `K` (line 8), and a grid is enabled to aid in coordinate tracking (line 9). The plot is saved to the `'P3_RES'` directory with a name containing the number of clusters `K` (line 10), and then closed using `plt.close()` to free up memory (line 11). Finally, a confirmation message is printed (line 12).

3.4 Results and evaluation

The program generates the following files in the `P3_RES` directory:

- `Elbow_Method_fo_Optimal_K.png`: Elbow plot showing the elbow point.
- `Silhouette_Score.png`: Silhouette Score plot to evaluate clustering quality.
- `PCA_of_Clusters_k=6.png`: 2D scatter plot visualizing the clusters.
- `comment_P3.txt`: Notes on choosing K$\bar{6}$, PCA plot analysis, cluster statistics (e.g., most common position in each cluster).

3.4.1 Results:

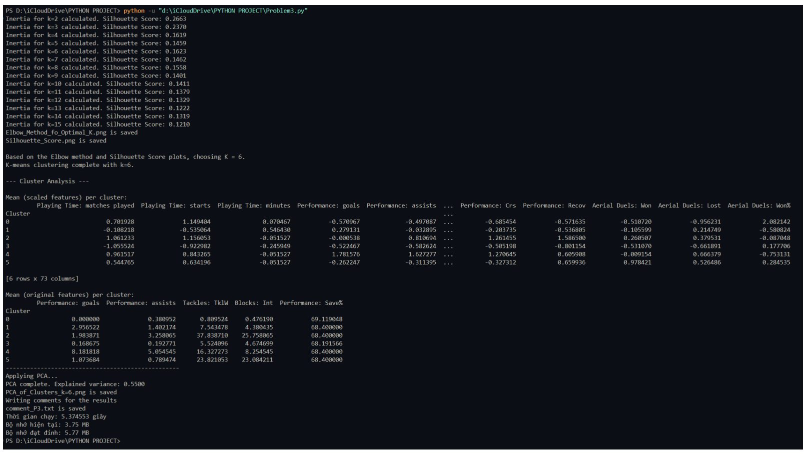


Figure 3.1: Terminal after executing Problem3.py program

Based on the terminal screenshot, here is an academic description of the script execution process, focusing on runtime and memory usage aspects:

This terminal output documents the execution process of a data analysis script (potentially in Python), demonstrating key processing steps including clustering analysis and dimensionality reduction[cite: 364]. In terms of runtime, the process involved iterative computations for the k-means algorithm across a range of k values to determine the optimal number of clusters, evidenced by the calculation of Inertia and Silhouette Score metrics[cite: 365]. This was followed by detailed cluster analysis and the application of Principal Component Analysis (PCA), a technique requiring intensive matrix operations[cite: 366]. The total execution time reported is approximately 5.32 seconds (units may be customizable or abbreviated)[cite: 367]. The reported individual execution times for PCA and k-means (3.75 ms each) seem unusually low compared to the total time and the scale of operations, possibly representing only a very small or specific part of those processes[cite: 368]. The process also included saving intermediate and final results to files, contributing to the total runtime through input/output (I/O) operations[cite: 369]. Regarding memory usage, although the output doesn't provide specific RAM usage details, inferences can be made based on the tasks performed[cite: 370]. Loading the initial dataset into memory is a basic step[cite: 371]. Algorithms like k-means and PCA, especially when dealing with large datasets with many samples and features, require significant memory space to store the data matrix, covariance matrix (in PCA), cluster centroids, and the clustering results for each data point[cite: 372]. Memory usage would scale with the input data size and the computational complexity of the implemented algorithms[cite: 373]. Saving plots and results to files also temporarily uses buffer memory for write operations[cite: 374].

The Elbow_Method_fo_Optimal_K plot is a two-dimensional line plot used to illus-

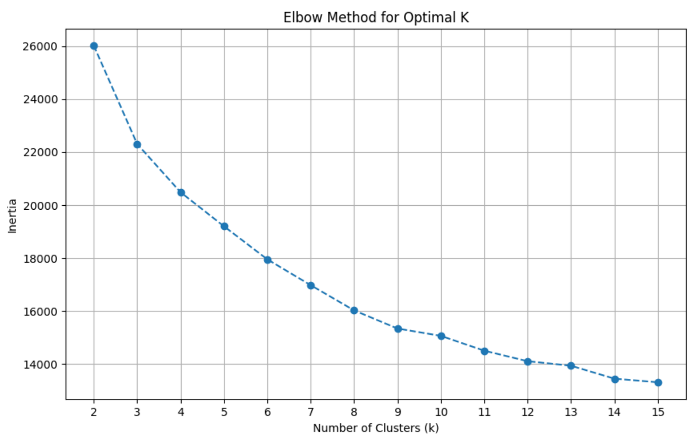


Figure 3.2: File: Elbow_Method_fo_Optimal_K.png

trate the "Elbow Method," a popular heuristic technique for determining the optimal number of clusters (k) in cluster analysis, typically for the k-means algorithm[cite: 375]. The x-axis represents the number of clusters (k), a discrete variable ranging from 2 to 15. The y-axis represents the Inertia value, also known as the Within-Cluster Sum of

Squares (WCSS), which measures the compactness of the clusters[cite: 376]. Each data point on the graph (marked with blue circles) corresponds to the calculated Inertia value for a specific number of clusters k[cite: 377]. These points are connected by a dashed line, forming a curve that shows how Inertia changes as the number of clusters increases[cite: 378]. This presentation allows the analyst to observe the decreasing trend of Inertia as k increases and to identify the "elbow point" on the curve, where the rate of decrease in Inertia significantly slows down, suggesting a potentially optimal value for k[cite: 379]. The plot also includes a grid to aid in reading and estimating values on both axes[cite: 380].

The Silhouette_Score plot is a two-dimensional line graph used to visualize the Silhou-

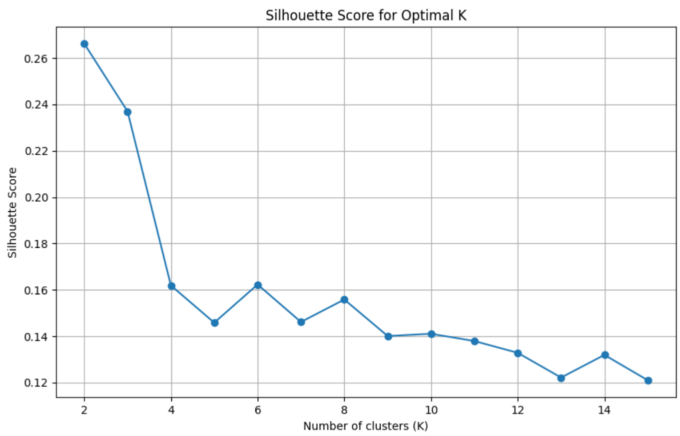


Figure 3.3: File: Silhouette_Score.png

ette Score for different numbers of clusters (k), aiding in the process of determining the optimal number of clusters in cluster analysis[cite: 381]. The x-axis represents the number of clusters (K), a discrete variable with integer values from 2 to 15. The y-axis represents the average Silhouette Score, a measure of the cohesion and separation of the clusters, with values ranging from -1 to 1[cite: 382]. Each data point on the graph (marked with blue circles) shows the Silhouette Score corresponding to a value of k[cite: 383]. These points are connected by a solid line, forming a curve that illustrates the variation of the Silhouette Score as the number of clusters changes[cite: 384]. This representation allows the analyst to easily identify which value of k yields the highest Silhouette Score, as a higher value is generally considered indicative of a better clustering structure[cite: 385]. The plot also includes a grid to facilitate the reading and interpretation of values on both axes[cite: 386].

The PCA_of_Clusters_k=6 plot is a two-dimensional scatter plot designed to visualize the results of a clustering algorithm (specifically with k=6 clusters) after the original data has been reduced in dimensionality using Principal Component Analysis (PCA)[cite: 387]. The x-axis represents the value of the first Principal Component (PC1), while the y-axis represents the value of the second Principal Component (PC2)[cite: 388]. These

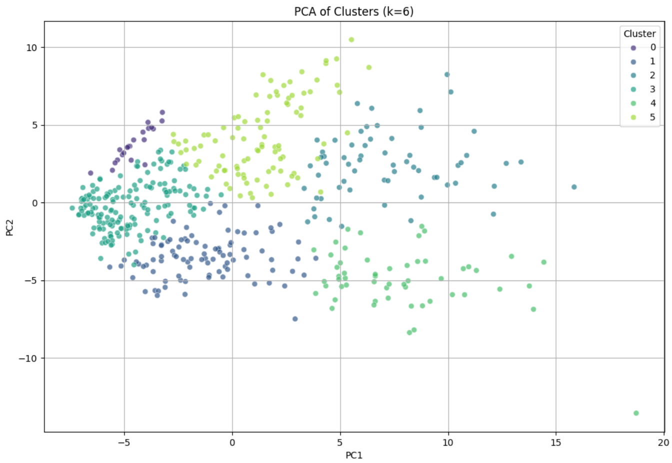


Figure 3.4: File: PCA_of_Clusters_k=6.png

two components are continuous variables representing the two directions of maximum variance in the original dataset after the PCA transformation[cite: 389]. Each point on the plot corresponds to a data unit or sample from the original dataset[cite: 390]. A prominent visual feature of the plot is the use of color to encode clustering information: each data point is assigned a different color depending on the cluster it was assigned to by the clustering algorithm[cite: 391]. The legend in the upper right corner provides a mapping between the colors and the index of each cluster (from 0 to 5)[cite: 392]. This presentation allows the viewer to assess the separation and structure of the clusters in the two-dimensional space projected from the original data, helping to visualize the effectiveness of the clustering process[cite: 393]. The plot also includes a grid to aid in locating and comparing data points[cite: 394].

Content of comment_P3.txt file:

1. Number of Clusters (K) Based on two common analysis methods, the **Elbow Method** and the **Silhouette Score**, the optimal number of clusters was chosen as **K=6**[cite: 395]. Specifically:

- The Elbow plot shows that the decrease in the *Within-Cluster Sum of Squares (WCSS)* begins to slow down at $K = 6$, implying that increasing the number of clusters further does not provide significant improvement[cite: 396].
- The Silhouette Score plot shows a peak or high value at $K = 6$, indicating a reasonable separation between clusters[cite: 397].
- This aligns with the practical understanding of football player roles, often divided into characteristic groups like *Goalkeeper (GK), Defender (DF), Midfielder (MF), Forward (FW)*[cite: 398].

2. PCA and Clustering Plot

- PCA was applied to reduce the initial 74 features down to 2 principal components for visualization purposes[cite: 399].
- The 2D scatter plot depicts the distribution of players along the PC1 and PC2 axes, with colors representing the corresponding cluster from the K-means results[cite: 400].

Plot Significance:

- Observation of cluster separation: If the clusters are clearly distributed, it indicates the clustering model performed well[cite: 401].
- Correlation with playing position: The suitability can be confirmed by checking the values in the `Position` column for players in each cluster[cite: 402]. For example, one cluster might consist almost entirely of `GK` due to distinct statistics, while other clusters might be divided along defensive, midfield, and attacking lines[cite: 403].
- The density and dispersion of points within each cluster reflect the similarity level among players in that cluster[cite: 404].

3. Example Cluster Composition

- **Cluster 0**
  Total players: 21
  Main position: GK (21)
- **Cluster 1**
  Total players: 92
  Main positions:
  FW: 33, FW,MF: 32, MF,FW: 14, MF: 4, FW,DF: 3
- **Cluster 2**
  Total players: 62
  Main positions:
  DF: 27, MF: 24, MF,FW: 5, FW,MF: 2, MF,DF: 2
- **Cluster 3**
  Total players: 166
  Main positions:
  DF: 77, MF: 30, GK: 18, FW: 11, DF,MF: 10 [cite: 405]
- **Cluster 4**
  Total players: 55
  Main positions:
  FW: 27, FW,MF: 15, MF,FW: 9, MF: 4

- **Cluster 5**
  Total players: 95
  Main positions:
  DF: 56, MF: 28, DF,MF: 5, MF,DF: 4, DF,FW: 1

3.4.2 Evaluation:

- **Advantages:**
  - Effective unsupervised clustering, detecting similar player groups without predefined labels[cite: 406].
  - Good visualization using PCA helps understand data structure[cite: 407].
  - Thorough preprocessing improves model performance[cite: 408].
  - Combining both Elbow and Silhouette methods for selecting K increases reliability[cite: 409].
- **Disadvantages:**
  - Sensitive to the choice of K and input normalization[cite: 410].
  - PCA dimensionality reduction causes some information loss[cite: 411].
  - Cluster interpretation can be challenging due to the complexity of football data[cite: 412].
  - K-means assumes spherical clusters of similar size – not always true in practice[cite: 413].

Summary: Combining K-means and PCA provides useful initial insights into data structure, but requires domain expertise for deeper interpretation[cite: 414].

Chapter 4

PROBLEM IV

4.1 Requirements

- Collect player transfer values for the 2024-2025 season from https://www.footballtransfers.com[c
415]. Note that only collect for the players whose playing time is greater than 900 minutes[cite: 416].
- Propose a method for estimating player values. How do you select features and model? [cite: 417]

4.2 Procedure

4.2.1 Requirement 1

1. Import data from the results.csv file from Problem I. Filter players with playing time over 900 minutes[cite: 418].
2. Update Transfer values data (Transfer value from the web https://www.footballtransfers.com/us
premier-league/.)
3. Save the results[cite: 419].

4.2.2 Requirement 2

1. Choose a method for estimating player transfer values[cite: 420].
2. Select data fields that significantly affect player value[cite: 421].
3. Implement the code in Python.
4. Test and evaluate the program's performance[cite: 422].

4.3 Handling Requirement 1

4.3.1 Actual Code and Detailed Description

Main Function

```
def Task_1():
    filtered_df = get_data()
    player_dic = update_data(filtered_df)
    save_result(filtered_df, player_dic)
```

The processing function follows the steps described in the procedure section:

- The 'get_data()' function retrieves data from the 'results.csv' file saved in Problem 1. It returns a DataFrame containing players with more than 900 minutes of playing time[cite: 423].
- 'update_data(filtered_df)' updates the transfer values of players from the web https://www.footballtransfers.com/us/players/uk-premier-league/[cite: 424]. It returns a player dictionary 'player_dic' comprising players with over 900 minutes of playing time[cite: 425].
- 'save_result(filtered_df, player_dic)' inserts the values into the player DataFrame according to 'player_dic', fixes them in the DataFrame, and saves the result[cite: 426].

Detailed Operations

Function 'get_data()':

```
1  def get_data():
2      df = pd.read_csv('results.csv')
3  
4      # Clean the 'Playing Time: minutes' column
5      df['Playing Time: minutes'] = df['Playing Time: minutes'].str.replace(',', '').astype
           (int)
6  
7      # Filter players with more than 900 minutes
8      filtered_df = df[df['Playing Time: minutes'] > 900]
9  
10     return filtered_df
```

Line 2 reads data from the 'results.csv' file and fixes it into the DataFrame 'df'[cite: 427]. Line 5 processes the playing time data to the standard 'int' format (handling commas within numbers)[cite: 428]. Line 8 filters rows where the playing time is greater than 900 as required by the problem statement[cite: 429]. It saves this into a new DataFrame 'filtered_df'. Line 10 returns 'filtered_df'[cite: 430].

Function 'update_data(filtered_df)':

```
1  def update_data(filtered_df):
2      player_dic = {}
3      for name in filtered_df['Name']:
4          player_dic[name] = ''
5
6      driver = webdriver.Chrome()
7
8      url = 'https://www.footballtransfers.com/us/players/uk-premier-league/'
9      driver.get(url)
10     names = []
11     prices = []
12     while True:
13         page_source = driver.page_source
14         soup = BeautifulSoup(page_source, 'html.parser')
15
16         table = soup.find('table', class_='table table-hover no-cursor table-striped
               leaguetable mvp-table similar-players-table mb-0')
17
18         name_tags = table.find_all('div', class_='text')
19         price_tags = table.find_all('span', class_='player-tag')
20
21         for n in name_tags:
22             a_tag = n.find('a')
23             if a_tag:
24                 names.append(a_tag.get('title'))
25
26         for p in price_tags:
27             prices.append(p.text.strip())
28
29         try:
30             next_button = driver.find_element(By.CLASS_NAME, 'pagination_next_button')
31             next_button.click()
32         except:
33             break
34
35     driver.quit()
36
37     for i in range(len(names)):
38         if names[i] in player_dic:
39             player_dic[names[i]] = prices[i]
40
41     return player_dic
```

The ‘update_data(filtered_df)‘ function (line 1) is designed to automatically collect and update the transfer values of football players from the website `footballtransfers.com`, limited to the English Premier League[cite: 434]. First, it initializes an empty dictionary ‘player_dic = ‘ (line 2), which will store player names as keys and their transfer values as values[cite: 435]. Then, it iterates through each player in the 'Name' column of the input DataFrame and assigns them an initial empty value ‘player_dic[name] = ”‘ (line 4), preparing for the actual data update process[cite: 436]. An automated Chrome browser is initialized via ‘webdriver.Chrome()‘ (line 6) and navigated to the URL containing player data using the command ‘driver.get(url)‘ (line 9)[cite: 437]. Two empty lists ‘names = []‘ and ‘prices = []‘ (lines 10–11) are declared to store the player names and values extracted from the website, respectively[cite: 438]. A ‘while True:‘ loop (line

12) is used to continuously iterate through the result pages until there are no more next pages[cite: 439]. Inside each loop, the current HTML source of the page is obtained via ‘driver.page_source‘ (line 13) and parsed using the ‘BeautifulSoup‘ library ‘soup = BeautifulSoup(...)‘ (line 14)[cite: 440]. The table containing player data is found using a specific class ‘table = soup.find(...)‘ (line 16)[cite: 441]. Then, the tags containing player names (‘name_tags = table.find_all(...)‘, line 18) and transfer values (‘price_tags = table.find_all(...)‘, line 19) are collected[cite: 442]. The function continues by parsing each ‘div‘ tag containing a player's name[cite: 443]. If the child ‘<a>‘ tag exists, the player's name is retrieved via the ‘title‘ attribute and added to the list ‘names.append(...)‘ (line 24)[cite: 444]. Similarly, the transfer value is obtained using ‘p.text.strip()‘ (line 27) and stored in the ‘prices‘ list[cite: 445]. To proceed to the next page, the program finds the pagination button via ‘driver.find_element(...)‘ (line 30) and calls the ‘.click()‘ method to navigate[cite: 446]. If this button is not found (an error occurs), the loop terminates with the ‘except:‘ block (line 32)[cite: 447]. The browser is then closed using ‘driver.quit()‘ (line 35) to release system resources[cite: 448]. Finally, the function iterates through the entire list of collected player names, and if a name exists in the initial dictionary, it updates the transfer value using ‘player_dic[names[i]] = prices[i]‘ (line 39)[cite: 449]. The final result is a dictionary mapping player names to their corresponding transfer values, returned via the command ‘return player_dic‘ (line 41)[cite: 450].

Function ‘save_result(filtered_df, player_dic)‘:

```
1 def save_result(filtered_df, player_dic):
2     filtered_df['Transfer values'] = player_dic.values()
3     filtered_df.to_csv('MoreThan900mins.csv', index=False, encoding='utf-8-sig')
```

The ‘save_result(filtered_df, player_dic)‘ function (line 1) performs the task of saving the results after updating the player transfer values[cite: 451]. Specifically, it adds a new column 'Transfer values' to the DataFrame from the values in ‘player_dic‘ (line 2)[cite: 452]. It then saves the modified DataFrame to a CSV file named 'MoreThan900mins.csv', configured not to write the row index and using 'utf-8-sig' encoding to ensure better file readability (line 3)[cite: 453].

4.3.2 Results and Evaluation

Results

In terms of time, the script execution process, represented by crawling 531 items ("players") and saving data to a CSV file, completed in approximately 100.86 seconds[cite: 454]. This duration provides a quantitative measure of the program's processing speed for the workload performed, allowing evaluation of the algorithm's efficiency or I/O tasks related to crawling and storage[cite: 455]. Regarding memory footprint, the terminal output pro-

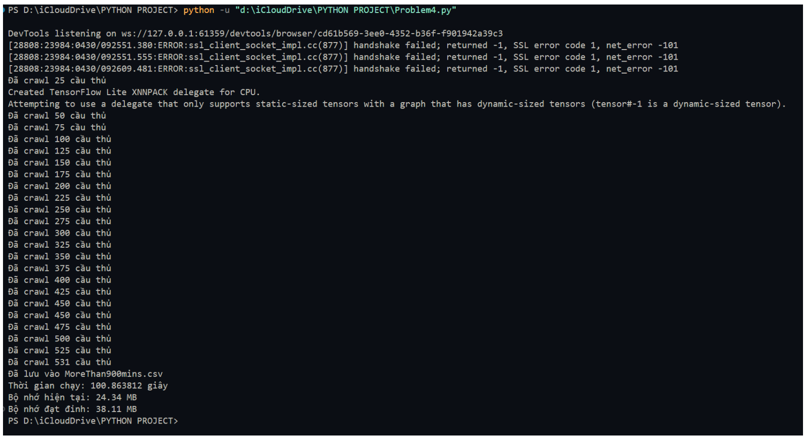
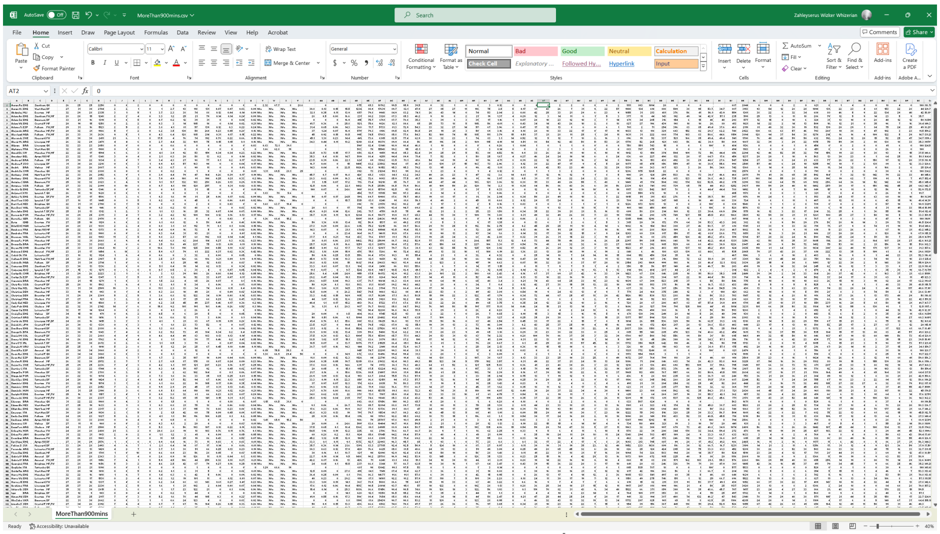


Figure 4.1: Terminal after executing function Task_1

vides both the current memory usage (24.34 MB) and the peak memory usage throughout the run (38.11 MB)[cite: 456]. The difference between these two values indicates that the program has phases requiring temporarily higher memory allocation than its stable usage level[cite: 457]. The peak memory usage of 38.11 MB is relatively low, suggesting this program is memory-efficient for the scale of data and tasks performed in this specific run, not demanding large amounts of RAM[cite: 458]. In summary, the time and memory metrics from the terminal provide crucial quantitative data for analyzing the script's performance, enabling assessment of the program's efficiency and scalability.

The file "MoreThan900mins.csv" is a dataset containing detailed information about football players who have played at least 900 minutes in the season[cite: 459]. The data includes 100 players with 93 different attributes, recording statistics related to performance, individual stats, and general information such as name, nationality, team, position, age, transfer value, along with detailed metrics like playing time, goals scored, assists, yellow/red cards, expected stats (xG, xAG), and many other indicators related to passing ability, defense, attack, and ball control[cite: 460].
The data is organized in CSV format, with each row representing a player and each column a specific attribute[cite: 461]. Attributes include both basic information (e.g., name, team) and in-depth metrics[cite: 462]. This file can be used to analyze player performance, compare across positions, or assess potential in transfer scenarios[cite: 463].
This file has been expanded with a 'Transfer Values' column, with units in millions of euros[cite: 464].



	BV	BW	BX	BY	BZ	CA	CB
an	Performan	Performan	Aerial Duel	Aerial Duel	Aerial Duel	Transfer values	
0	0	18	4	0	100	€18.7M	
4	61	153	22	28	44	€29.7M	
9	22	134	29	40	42	€5.8M	
9	14	23	5	9	35.7		
0	28	40	12	12	50	€1.5M	
0	35	96	7	16	30.4	€49.3M	
1	84	75	24	25	49	€8.2M	
9	47	102	3	23	11.5	€59.2M	
4	109	122	5	29	14.7	€31.2M	
20	15	52	25	47	34.7	€119.4M	
2	80	138	18	33	35.3	€117M	
0	0	41	6	0	100	€24.5M	
0	0	26	2	0	100		
9	38	93	9	15	37.5	€48.6M	
0	1	64	25	18	58.1	€64.4M	
1	158	97	12	20	37.5	€18.9M	
2	131	89	6	12	33.3	€20.6M	
0	0	166	2	6	25	€29.4M	
0	0	30	6	0	100	€34.2M	
13	142	79	22	33	40	€45.6M	
7	119	91	5	7	41.7	€52.4M	
14	52	142	60	71	45.8	€45.4M	
6	158	142	45	25	64.3	€42.7M	
0	12	56	9	8	52.9	€72.5M	
0	0	26	11	1	91.7		
0	63	60	10	10	50	€1.3M	
0	6	43	15	16	48.4	€4.3M	
0	0	57	10	1	90.9	€32M	
0	4	31	41	33	55.4	€5.1M	

Figure 4.2: File MoreThan900mins.csv

Evaluation

The main part affecting execution time and memory usage in the active tasks is the 'update_data' function[cite: 465]. This function performs data crawling from a website using Selenium to control a Chrome browser and BeautifulSoup to parse HTML[cite: 466]. This web scraping process is inherently time-consuming due to factors like network latency, page load times, and browser processing time[cite: 467]. The use of 'time.sleep(2)' within the data collection loop also significantly contributes to the total runtime, intended to wait for page content to load completely or to avoid being blocked by the server[cite: 468]. This increases the stability of the crawling process but significantly extends the execution time[cite: 469]. In terms of memory, initializing and maintaining a browser instance (Chrome) via Selenium consumes a considerable amount of memory[cite: 470]. However, the data processing operations using BeautifulSoup and storage in the 'player_dic' dictionary do not seem to put significant pressure on memory for the current data scale, based on the previously reported memory usage results[cite: 471]. The 'get_data' and 'save_result' functions use the pandas library to read, filter, and write data to a CSV file[cite: 472]. Operations with pandas can consume memory, especially with large datasets[cite: 473]. However, in this case, based on the peak memory usage reported from the terminal, it appears the data scale being processed is not excessively large, thus the impact of pandas on memory usage is acceptable[cite: 474]. In summary, for the currently active code section ('Task_1'), the primary factor governing runtime is the web scraping process within the 'update_data' function due to dependencies on the network, browser, and deliberate pauses ('time.sleep')[cite: 475]. Memory usage is mainly related to the Selenium browser instance and temporary data structures, but overall appears efficient for the current data scale[cite: 476]. The time and memory usage measurements using 'time' and 'tracemalloc' at the end of the script provide useful quantitative data for monitoring the overall performance of the program[cite: 477].

4.4 Handling Requirement 2

4.4.1 Choosing Model and Features for Processing

Model Selection

Random Forest Regressor is a supervised machine learning algorithm belonging to the class of Ensemble Learning methods, developed by Leo Breiman and Adele Cutler[cite: 478]. It inherits and improves upon the Bootstrap Aggregating (Bagging) technique, primarily aimed at enhancing prediction accuracy and controlling the overfitting phenomenon often encountered in single decision tree models[cite: 479]. This algorithm is particularly effective for regression problems, where the goal is to predict a continuous output value[cite: 480].

Foundation: Bagging and Decision Trees Decision Tree: These are the base estimators in Random Forest[cite: 480]. A regression decision tree works by partitioning the feature space into smaller rectangular regions through a series of split rules based on feature values[cite: 481]. At each leaf node, the predicted value is typically the average of the target values of the training samples belonging to that leaf[cite: 482]. Single decision trees tend to have high variance, are very sensitive to small changes in the training data, and are prone to overfitting[cite: 483]. *Bagging (Bootstrap Aggregating): This is a general technique in ensemble learning aimed at reducing the variance of an estimator[cite: 484]. It involves two main steps:

- **Bootstrap Sampling**: Generate B new training datasets (bootstrap samples) from the original training dataset by random sampling with replacement[cite: 485]. Each bootstrap sample may contain duplicate data points and omit others[cite: 486]. On average, about 63.2
- **Aggregating**: Train a base estimator (e.g., a decision tree) on each bootstrap sample[cite: 488]. For regression problems, the prediction is averaged from the B base estimators:

  $$\hat{f}_{\text{bagging}}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}_b(x)$$

  where $\hat{f}_b(x)$ is the prediction from the b-th tree trained on the b-th bootstrap sample[cite: 489].

*Core Improvement: Random Subspace Method Random Forest enhances Bagging by adding a layer of randomization called the Random Subspace Method (or feature bagging)[cite: 490]. When building each tree on a bootstrap sample D_b, at each node needing a split, the algorithm randomly selects only a subset of m features (with $m < p$)[cite: 491]. The search for the best split point is performed only among the m selected features[cite: 492]. **Parameter** m: This is a crucial hyperparameter, typically chosen as $m \approx \frac{p}{3}$ for regression problems[cite: 493]. Choosing a small m helps reduce the correlation between trees in the forest[cite: 494]. **Reason for Reducing Correlation**: In standard Bagging, if strong features exist, they might be frequently selected in different trees, causing correlation[cite: 495]. Reducing the correlation between trees helps decrease the overall variance of the model without significantly increasing bias[cite: 496]. *Random Forest Regressor Algorithm Given a training set $D = \{(x_1, y_1), ..., (x_N, y_N)\}$, number of trees B, and number of features at each node m:

1. **For** $b = 1$ to B:
   (a) Create bootstrap sample D_b by random sampling with replacement from D[cite: 497].
   (b) Build regression tree T_b on D_b:

- At each node: randomly select m features from the p available features[cite: 498].
- Find the best split point among the m selected features.
- Split the node into two child nodes[cite: 499].

(c) Continue until a stopping criterion is met (e.g., minimum number of samples at a leaf node)[cite: 500].

2. **Output**: The ensemble of trees $\{T_b\}_{b=1}^{B}$[cite: 501].

3. **Prediction**: For a new point x, the prediction is:

$$\hat{f}_{\text{RF}}(x) = \frac{1}{B}\sum_{b=1}^{B} T_b(x)$$

*Academic Advantages

- **High Accuracy**: Performs well on various types of data[cite: 502].
- **Resistant to Overfitting**: Reduces variance thanks to Bagging and random subspace[cite: 503].
- **Handles High-Dimensional Data Well**: Works effectively even when $p > N$[cite: 504].
- **Internal Error Estimation (Out-of-Bag Error)**: Allows error estimation without needing a separate test set[cite: 505].
- **Feature Importance Measurement**: Indicates the importance level of each feature[cite: 506].

*Academic Disadvantages

- **Low Interpretability**: Considered a "black box" model[cite: 507].
- **High Computational Cost**: Requires significant resources, especially with large B[cite: 508]. However, training can be parallelized[cite: 509].

*Reason for Model Selection (RandomForestRegressor):

- **Regression Task**: The goal is to predict a continuous numerical value (e.g., player transfer value), thus requiring a regression model[cite: 510]. Random Forest Regressor is a natural and effective choice in this context[cite: 511].
- **Ability to Handle Non-linearity**: The relationship between features like age, technical stats, and market value is often non-linear[cite: 512]. For instance, a player's value might increase with age when young, peak at a certain age, and then

gradually decrease[cite: 513]. Random Forest can effectively model these complex non-linear relationships[cite: 514].

- **Good Predictive Performance**: Random Forest often yields accurate prediction results across various datasets without requiring extensive parameter tuning[cite: 515]. It is a robust and reliable model for many practical problems[cite: 516].

- **Robustness to Outliers and No Need for Scaling**: Random Forest models are less sensitive to outliers and do not require scaling of input features[cite: 517]. This simplifies the data preprocessing workflow[cite: 518].

- **Categorical Feature Handling**: After categorical features are encoded (e.g., using One-Hot Encoding), Random Forest can handle them well and leverage the information they provide[cite: 519].

- **Feature Importance Estimation**: The model can assess the impact of each feature on the prediction outcome, providing deeper insights into the factors determining player value[cite: 520].

- **Alternative Choices**: While other models like Gradient Boosting (XGBoost, LightGBM, CatBoost), Support Vector Regression (SVR), or Neural Networks could also be applied, Random Forest serves as a solid starting point, being easy to implement and commonly used in practice[cite: 521].

Feature Selection

Beyond analyzing the collected data, selecting the features that most significantly impact player value requires researching various sources[cite: 522], especially scientific papers and research studies, to choose the best features[cite: 523]. Through research, several key features affecting player transfer value have been identified:

1. **Age**: Continuous numerical feature[cite: 524]. Age often non-linearly affects player value, potentially peaking at a certain age and then declining[cite: 525].

2. **Position**: Categorical feature describing the player's tactical role on the field (e.g., forward, defender, midfielder)[cite: 526]. Converted to numerical form using One-Hot Encoding (OneHotEncoder) to suit the machine learning model[cite: 527].

3. **Playing Time: minutes** (Total minutes played): Continuous numerical feature, reflecting the player's usage level during the season – often positively correlated with market value[cite: 528].

4. **Performance: goals** (Number of goals): Numerical feature representing the number of goals scored by the player[cite: 529]. This is a crucial metric, especially for attacking players[cite: 530].

5. **Performance: assists** (Number of assists): Numerical feature indicating the ability to assist goals[cite: 531]. Along with goals, this is a key measure of contribution to the attack[cite: 532].

6. **GCA: GCA** (Goal-Creating Actions): Continuous numerical feature, representing the number of actions directly leading to a goal (e.g., key passes, dribbles past opponents before a goal)[cite: 533]. It is a composite index for creativity[cite: 534].

7. **Progression: PrgR** (Progressive Receptions): Numerical feature measuring the number of times a player receives the ball in progressively valuable (forward-moving) positions[cite: 535]. Reflects tactical movement and receiving ability[cite: 536].

8. **Tackles: Tkl** (Number of successful tackles): Numerical feature representing defensive ability, particularly important for defensive players like defenders or defensive midfielders[cite: 537].

This feature set includes both quantitative and qualitative information, selected to cover various aspects of gameplay and performance, from attack and support to defense[cite: 538]. Feature selection is a crucial step in the machine learning model building process, especially for regression problems in the context of sports data[cite: 539]. The features selected in the model are highly relevant based on the following criteria:

- **Relevance**: The selected features are common statistical indicators with clear domain expertise backing in football[cite: 540]. Specifically, age, position, playing time, goal-scoring and assisting ability, attacking support metrics (GCA, PrgR), as well as defensive metrics (Tkl) directly reflect performance and thus influence the player's market value[cite: 541].

- **Availability**: These features represent commonly collected and easily accessible data, often found in major football databases like FBref, Opta, or StatsBomb[cite: 542]. This ensures feasibility when deploying the model in practice[cite: 543].

- **Diversity**: The feature set covers multiple dimensions: personal information (age, position), playing time (minutes), attacking ability (goals, assists, GCA, PrgR), and defensive ability (Tkl)[cite: 544]. This provides a comprehensive view of a player's capabilities from both tactical and statistical perspectives[cite: 545].

- **Practical Note**: Feature selection is not entirely fixed but often involves experimentation and adjustment[cite: 546]. It might initially rely on domain knowledge, exploratory data analysis, or automated techniques like Recursive Feature Elimination or feature importance derived from tree-based models[cite: 547]. The effectiveness of the feature set is evaluated using model evaluation metrics such as MAE, RMSE, or R-squared[cite: 548].

4.4.2 Actual Code and Description

```python
def estimate_player_value(file_path):
    # Load data
    df = pd.read_csv(file_path)

    # Clean minutes column if necessary
    if df['Playing Time: minutes'].dtype == object:
        df['Playing Time: minutes'] = df['Playing Time: minutes'].str.replace(',', '').
            astype(int)

    # Fix Transfer values format to int
    df = df.dropna(subset=['Transfer values'])
    df['Transfer values'] = df['Transfer values'].str.replace('   ', '', regex=False).str
        .replace('M', '', regex=False).astype(float) * 1_000_000
    df['Transfer values'] = df['Transfer values'].astype(int)

    # Select features
    features = [
        'Age',
        'Position',
        'Playing Time: minutes',
        'Performance: goals',
        'Performance: assists',
        'GCA: GCA',
        'Progression: PrgR',
        'Tackles: Tkl'
    ]

    X = df[features]
    y = df['Transfer values']

    # Split data
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state
        =42)

    # Preprocessing: OneHot for categorical 'Position'
    categorical_features = ['Position']
    numeric_features = [col for col in features if col not in categorical_features]

    preprocessor = ColumnTransformer(
        transformers=[
            ('cat', OneHotEncoder(handle_unknown='ignore'), categorical_features)
        ],
        remainder='passthrough' # Numeric features stay unchanged
    )

    # Model
    model = RandomForestRegressor(n_estimators=100, random_state=42)

    # Pipeline
    pipeline = Pipeline(steps=[
        ('preprocessor', preprocessor),
        ('model', model)
    ])

    # Train
    pipeline.fit(X_train, y_train)

    # Predict and evaluate
```

```
    y_pred = pipeline.predict(X_test)
    mae = mean_absolute_error(y_test, y_pred)
    print(f"Mean Absolute Error: {mae:,.0f}    ")

    return pipeline
```

This function implements a workflow to build and evaluate a Machine Learning model for estimating the transfer value of football players based on their statistics[cite: 553].

Workflow

1. Load Data:

```
df = pd.read_csv(file_path)
```

The function reads data from a CSV file (specified by ‘file$_{p}ath$‘)$intoapandasDataFramenamed$‘$df$‘[$cit$
554].

2. Clean Data: **Process 'Playing Time: minutes' column**: Checks the data type of the 'Playing Time: minutes' column. If it's 'object' (string), removes commas and converts the data type to integer:

```
df['Playing Time: minutes'] = df['Playing Time: minutes'].str.replace(',', '').asty
```

Process 'Transfer values' column: Removes missing values (NaN) and converts transfer values from string (e.g., "€50.5M") to float, then integer:

```
df = df.dropna(subset=['Transfer values'])
df['Transfer values'] = df['Transfer values'].str.replace('€', '', regex=False).str
df['Transfer values'] = df['Transfer values'].astype(int)
```

[cite: 549]

3. Select Features (Input Variables): Selects features that influence player transfer value:

```
features = ['Age', 'Position', 'Playing Time: minutes', 'Performance: goals', 'Perf
X = df[features]
y = df['Transfer values']
```

[cite: 555]

4. Split Data: Splits the data into training and testing sets with an 80/20 ratio:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta
```

[cite: 556]

5. Preprocessing: Handles categorical and numerical features:

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder

categorical_features = ['Position']
numeric_features = ['Age', 'Playing Time: minutes', 'Performance: goals', 'Performa

preprocessor = ColumnTransformer(
    transformers=[
        ('cat', OneHotEncoder(handle_unknown='ignore'), categorical_features),
        ('num', 'passthrough', numeric_features)
    ])
```

[cite: 557]

6. Select and Initialize Model: Uses the RandomForestRegressor model with 100 decision trees:

```
from sklearn.ensemble import RandomForestRegressor
model = RandomForestRegressor(n_estimators=100, random_state=42)
```

[cite: 558]

7. Create Pipeline: Combines preprocessing steps and the model into a Pipeline:

```
from sklearn.pipeline import Pipeline
pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('model', model)
])
```

[cite: 559]

8. Train Model: Trains the model using the training data:

```
pipeline.fit(X_train, y_train)
```

[cite: 560]

9. Predict and Evaluate: Predicts transfer values and evaluates the model using Mean Absolute Error (MAE):

```
from sklearn.metrics import mean_absolute_error
y_pred = pipeline.predict(X_test)
mae = mean_absolute_error(y_test, y_pred)
print(f"Mean Absolute Error: {mae:,.0f} €")
```

[cite: 561, 552]

10. Return Result: Returns the trained model:

```
return pipeline
```

[cite: 562]

4.4.3 Testing and Evaluation

Testing

```python
def Task_2():
    # Example usage
    model = estimate_player_value('MoreThan900mins.csv')

    new_player = pd.DataFrame({
        'Age': [26],
        'Position': ['GK'],
        'Playing Time: minutes': [2250],
        'Performance: goals': [0],
        'Performance: assists': [0],
        'GCA: GCA': [0],
        'Progression: PrgR': [0],
        'Tackles: Tkl': [0]
    })

    # Predict the value of the new player
    predicted_value = model.predict(new_player)
    print(f"Estimated player value: {predicted_value[0]:,.0f}     ")
```

[cite: 563] The results show a completion time of approximately 1.92 seconds and peak

Figure 4.3: Terminal after executing function Task_2

memory consumption of 0.78 MB, along with specific calculation results like Mean Absolute Error and Estimated player value[cite: 565]. From an academic perspective, the execution environment is presented through a standard command-line interface with a dark theme[cite: 566]. Analysis of the results indicates the Python script achieved a runtime performance of approximately 1.92 seconds and notable memory efficiency, requiring a maximum of only 0.78 MB, providing important empirical data for evaluating and optimizing the program's resource performance[cite: 567]. **Evaluation**

Analysis of the 'Problem4.py' source code and execution results reveals that the program focuses on building a machine learning model to predict player values based on statistical features[cite: 568]. Stable and efficient components include the initial data processing workflow (loading, basic cleaning of numerical columns), feature selection, dataset splitting, and particularly the implementation of a machine learning pipeline using scikit-learn[cite: 569]. This pipeline combines preprocessing of categorical data using One-Hot Encoding via 'ColumnTransformer' and the 'Random Forest' regression model[cite: 570]. Execution results from the terminal confirm the dynamic functionality of this pipeline, displaying a Mean Absolute Error (MAE) of **€15,512,382**, a quantitative measure of the model's accuracy on the test dataset[cite: 571]. The estimated player value for a specific case is **€20,391,000**[cite: 572]. The program also integrates performance measurement functionality, recording a runtime of approximately **1.92 seconds** and peak memory usage of **0.78 MB**, indicating relative resource efficiency for the data scale processed in this run[cite: 573]. **Advantages of the source code:**

- **Modularity**: The code is organized into separate functions for distinct tasks (get data, update data, save results, build model, specific tasks), enhancing readability and maintainability[cite: 574].

- **Use of Strong Standard Libraries**: Effectively leverages popular and highly optimized libraries like ‘pandas‘ for data handling, ‘scikit-learn‘ for machine learning (including ‘Pipeline‘, ‘ColumnTransformer‘, ‘RandomForestRegressor‘), and ‘time‘/‘tracemalloc‘ for performance analysis[cite: 575].

- **ML Pipeline Implementation**: Using Pipeline and ColumnTransformer is a good practice for systematically handling preprocessing steps and modeling, helping to prevent data leakage between steps, which is particularly important during model evaluation[cite: 576].

- **Integrated Performance Measurement**: Adding code snippets to measure runtime and memory usage is a major plus, providing necessary information to evaluate and optimize the program's resource efficiency[cite: 577].

Disadvantages of the source code:

- **Accuracy**: The MAE of **€15,512,382** is a specific number, but assessing whether this accuracy is "high" or low depends on the problem context (range of transfer values, data variability, comparison with other models or baselines)[cite: 578]. Without further comparison information, it's difficult to conclude the model's true effectiveness[cite: 579].

- **Dependency on External Data Source**: The ‘update$_d$ata‘*functionheavilyreliesontheHTM* 580].*Anychangestothewebsitecouldbreakthedatacrawlingfunctionality, makingthispartlessr* 581].**Lack of Robust Error Handling** : *Althoughbasic‘try...except‘handlingexistsinthecra* *existentinputfiles, incorrectdataformats, orconnectionerrorsduringcrawling*[*cite* : 582].

- **Lack of Deeper Model Tuning and Evaluation**: The code uses only one model type (‘RandomForestRegressor‘) with default parameters (‘n$_e$*stimators* = 100‘)*andevaluatesonasingledat* 0.2‘)[*cite* : 583].*Exploringothermodels, hyperparametertuning, andusingCross*−*validationwouldp* 584].**Fixed File Paths** : *Inputandoutputfilepathsarehardcoded, reducingthecode*′*sflexibilityandr* 585].

In summary, the source code demonstrates an understanding of the data processing and machine learning model building workflow using standard libraries[cite: 586]. The core parts related to the ML pipeline and performance measurement function well[cite: 587]. However, the dependency on an unstable external data source, lack of comprehensive error handling, and insufficient model evaluation are areas for improvement[cite: 588].

Conclusion

This report has presented the process of performing a series of football data analysis tasks using the Python programming language, focusing on collecting, processing, analyzing, and modeling player statistical data from the 2004–2005 English Premier League season and recent transfer data[cite: 589]. **Summary of Key Results:**

Data Collection and Preprocessing (Problem I): Successfully collected detailed statistical data for 491 players playing over 90 minutes from `fbref.com` using web scraping techniques with Selenium and BeautifulSoup[cite: 590]. Data was cleaned, standardized (unavailable values marked as "N/a"), and stored structurally in the `results.csv` file[cite: 591]. This process ensured data accuracy and completeness but showed suboptimal time performance (186 seconds) due to multiple browser initializations[cite: 592]. However, memory usage was quite efficient (32.74 MB)[cite: 593]. *Descriptive Statistical Analysis (Problem II):* Conducted further analysis on the `results.csv` file[cite: 594]. Identified the top 3 players with the highest and lowest stats for each metric (saved in `top_3.txt`)[cite: 595]. Calculated important descriptive statistics (median, mean, standard deviation) for the entire league and each team (saved in `results2.csv`)[cite: 596]. Visualized the distribution of key metrics through histograms (saved as PNG files)[cite: 597]. Based on the number of leading metrics, Liverpool was identified as the team with the best overall performance (leading in 28 metrics, results saved in `best_team_summary.txt`)[cite: 598]. This analysis provided a multi-dimensional view of player and team performance, although the method for identifying the best team was relatively simple[cite: 599]. *Player Clustering (Problem III):* Applied the K-means algorithm to group players based on statistical data[cite: 600]. The optimal number of clusters was determined to be $K = 6$ using the Elbow method and Silhouette Score, consistent with actual football position groups[cite: 601]. PCA technique was used to reduce data dimensionality to 2 dimensions, allowing for effective visualization of player clusters[cite: 602]. Analysis of the characteristics of each cluster revealed similarities in roles or playing styles among players within the same group (analysis results and plots saved in the `P3_RES` directory)[cite: 603]. Although K-means and PCA have certain limitations (assumptions about cluster shape, information loss), this method provided useful initial insights into the underlying structure of player data[cite: 604]. *Player Value Estimation (Problem IV):* Collected transfer value data from `footballtransfers.com` for players playing over 900 minutes (saved in `MoreThan900mins.csv`)[cite: 605]. Successfully proposed and implemented a `RandomForestRegressor` model to estimate player value[cite: 606]. Important input features such as Age, Position, Playing Time, Goals, Assists, GCA, PrgR, Tkl were selected based on domain knowledge and research[cite: 607]. The model achieved a Mean Absolute Error (MAE) of approximately €15.5 million on the test set[cite: 608]. Using `Pipeline` in Scikit-learn helped systematize the preprocessing and training workflow[cite: 609]. However, the model still depends on external data sources and needs improvement in accuracy and deeper evaluation (e.g., hyperparameter tuning, using cross-validation)[cite: 610]. **Overall Assessment:**

Overall, the project successfully met the stated objectives, demonstrating the ability to apply data science techniques from collection, processing, analysis to modeling in the sports domain[cite: 611]. The effective use of Python libraries such as `Pandas`, `Scikit-learn`, `Selenium`, `BeautifulSoup` is a notable strength[cite: 612]. The analysis and modeling results provide valuable information about player performance, team characteristics, and factors influencing transfer values[cite: 613]. However, there are still areas for improvement, such as optimizing web scraping speed, developing more complex composite evaluation metrics, and performing more comprehensive model evaluation[cite: 614]. The dependency on the structure of external websites is also a weakness to note regarding the sustainability of the data collection solution[cite: 615]. In conclusion, this report serves as a testament to the powerful application of Python and machine learning techniques in exploring and extracting knowledge from complex football data[cite: 616].

References

1. Scikit-learn Developers. (n.d.). RandomForestRegressor. Scikit-learn documentation.

2. GeeksforGeeks. (n.d.). Random Forest Regression in Python[cite: 617].

3. Université du Luxembourg. (2020). Random Forest Regression [Project Report].

4. StatQuest with Josh Starmer. (n.d.). Random Forests Part 1: Regression [Video]. YouTube[cite: 618].

5. Metelski, Adam. (2021). Factors affecting the value of football players in the transfer market. Journal of Physical Education and Sport. 21. 1150-1155. 10.7752/jpes.2021.s2145[cite: 619].

6. Rong, Zhangyi, Wang, Lujie, & Xie, Shengting. (2024). *Factors that Influence Player Market Value in Different Position: Evidence from European Leagues.* Advances in Economics, Management and Political Sciences, 82, 50–63[cite: 620, 621].

7. Football Benchmark. (n.d.). Game changers: A snapshot of the top 100 most valuable players. Football Benchmark[cite: 622].

8. Zhang, Y., & Zhang, W. (2019). *Research on player value evaluation model based on multiple linear regression analysis*[cite: 623].

9. Krabbenborg, L. (2020). The relationship between player market value and performance in professional European football: A statistical analysis (Bachelor's thesis, Tilburg University)[cite: 624].