POSTS AND TELECOMMUNICATIONS INSTITUTE OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY I

—————o0o—————

ASSIGNMENT REPORT 1
PYTHON PROGRAMMING LANGUAGE

Instructor:	**Kim Ngoc Bach**
Student:	**Le Nhu Quynh**
Student ID:	**B23DCCE081**
Class:	**D23CQCEO6-B**
Academic Year:	**2023 - 2028**
Training System:	**Full-time University**

Hanoi, 2025

POSTS AND TELECOMMUNICATIONS INSTITUTE OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY I

—————o0o—————

ASSIGNMENT REPORT 1

PYTHON PROGRAMMING LANGUAGE

Instructor: Kim Ngoc Bach
Student: Le Nhu Quynh
Student ID: B23DCCE081
Class: D23CQCEO6-B
Academic Year: 2023 - 2028
Training System: Full-time University

Hanoi, 2025

INSTRUCTOR'S COMMENTS

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Grade: **(In words:**)

Hanoi, date month year 20...

Instructor

Contents

List of Figures

INTRODUCTION

In the era of the data explosion, the ability to process and analyze data is a crucial skill for any programmer or data scientist. The `Python` language, with its rich ecosystem of libraries, allows learners to easily approach real-world problems, especially in the field of sports – where data is extensively collected, analyzed, and applied.

Assignment No. 1 of the `Python` Programming course poses the problem of collecting and analyzing player data in the English Premier League for the 2024–2025 season. This report presents a comprehensive processing workflow – from data collection, statistical analysis, visualization, to the application of machine learning algorithms. The content is structured around four main questions. This entire data collection process was carried out on May 6, 2025. Therefore, all analyses and results presented in this report reflect the situation and statistics of the players as of that specific point in the season. The specific content of the report includes the following main parts:

Chapter 1: Collecting Player Data from fbref.com:
This chapter presents the method of collecting data from the `fbref.com` website, filtering players with more than 90 minutes of playing time, processing raw data, and systematically storing it in the `results.csv` file, ready for subsequent analysis steps.

Chapter 2: Data Analysis and Visualization:
This chapter focuses on descriptive statistical analysis. Based on the collected data, the report identifies the top 3 players with the highest and lowest values for each statistic, calculates mean, median, standard deviation, and presents data distributions in graphical form.

Chapter 3: Player Clustering using K-Means and PCA:
By aggregating statistics for each team, this chapter uses charts and direct comparisons to provide an objective assessment of the match performance of clubs during the season, thereby identifying the team currently in the best form.

Chapter 4: Estimating Player Value:
The final chapter combines machine learning techniques to cluster players based on statistical characteristics, while also proposing a model to estimate transfer values based on match performance. Feature selection, model building, and results analysis will be presented in detail to demonstrate the practical applicability of the data.

Chapter 1

Collecting Player Data from Fbref.com

1.1 Analysis of Assignment Requirements

This part requires building a Python program to automatically collect statistical data of football players. Specific requirements include:

- **Target:** Players competing in the English Premier League 2024-2025 season
- **Data Source:** fbref.com website
- **Filtering Condition:** Collect data only for players who have played more than 90 minutes
- **Data to be collected:** Nation, Team, Position, Age, Playing Time (matches played, starts, minutes), Performance (goals, assists, yellow cards, red cards), Expected (xG, 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, Passing, Goal and Shot Creation, Defensive Actions, Possession, Miscellaneous Stats
- **Output Format:** Save results to `results.csv` file

Structure of `results.csv` file:

- Each column corresponds to a statistical indicator in the correct order and naming as in the assignment requirements list
- Each row represents a player
- Players must be sorted alphabetically by their First Name
- Statistical values that are not available or not applicable are marked as "N/a"

1.2 Program Structure

To meet the assignment requirements, I divided the program into 3 main Python modules:

- **config.py**: Central module containing configurations, constants, and data mapping definitions

- **scraper.py**: Module containing the logic and necessary functions/classes for data collection (web scraping)
- **main.py**: Main module, serving as the entry point and coordinator for the entire program's operation

1.3 Library Selection

To achieve the required results, I chose the following Python libraries:

- **Selenium**: I chose Selenium because Fbref.com uses JavaScript to load data in its statistical tables. Using a library like requests alone would not retrieve the complete HTML. Selenium helps me automate the browser (Chrome), wait for dynamic elements to load, and then get the page source.
- **Beautiful Soup (bs4)**: After Selenium retrieves the full HTML, I use BeautifulSoup to parse this HTML structure. It is very powerful for finding HTML tags (like `<table>`, `<tr>`, `<td>`) based on attributes (e.g., `id="stat_standard"`, `data-stat="goals"`) and extracting the text content within them.
- **Pandas**: This library is essential in the final step. After extracting the raw data, I use Pandas to create a DataFrame (data table), ensuring the correct structure and column order as required, and then easily save this DataFrame to a `results.csv` file.

1.4 Overall Implementation Process

My program collects data through the following main steps:

1. **Initialization**: Starts running from the `main.py` file. The program reads necessary settings from the `config.py` file (such as web address, table IDs, column name mapping, 90-minute filter threshold).
2. **Fetch Initial Data**: The program uses `main.py` to direct `scraper.py` to access the page containing the main statistics table (standard) on Fbref. Using Selenium and BeautifulSoup, it retrieves data for all players in this table.
3. **Filter Players**: Immediately after fetching the initial data, the program filters out players with playing time (minutes) greater than 90. These valid players are saved (as Player objects).
4. **Fetch Additional Data**: The program continues to access pages containing other supplementary statistics tables (goalkeeping, shooting, passing, defense, etc.) according to the list defined in `config.py`.
5. **Update Data**: With data from each supplementary table, the program finds the players already in the original list (matched by name) and updates their information with new statistics from the supplementary table.
6. **Sort**: After collecting and updating enough information from all tables, the program sorts the player list alphabetically by First Name.

7. **Create and Export File**: Finally, the program uses the Pandas library to create a DataFrame from the sorted player list, ensuring the columns are in the correct required order. It performs a final cleaning of numerical data (removing commas) and saves this table as a `results.csv` file.

1.5 Detailed Analysis of Components

1.5.1 Configuration and Data Mapping (`config.py`)

This is where I define all fixed parameters and data structures to ensure the program runs correctly and is easier to modify.

1. **Basic Information:** URL, filename, minute threshold

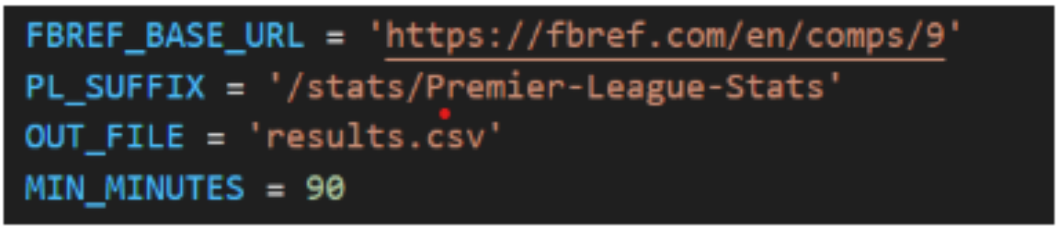

Figure 1.1: Definition of basic information in config.py.

2. **Table IDs and URLs:** Identification IDs and links to tables, helping the program identify and locate the correct data tables needed

Figure 1.2: Definition of table IDs and URLs in config.py.

3. **Mapping (`HEADER_MAP`):** Connects CSV column names with `data-stat` in HTML.

```
HEADER_MAP = {
    'Player': 'player',
    'Nation': 'nationality',
    'Team': 'team',
    #...(toàn bộ ánh xạ)...
}
```

Figure 1.3: Definition of HEADER_MAP mapping in config.py (excerpt).

4. **Column Order (`HEADER_ORDER`):** Ensures the CSV file has the correct column structure as required.

```
HEADER_ORDER = [
    'Player', 'Nation', 'Team',... ,
    'Miscellaneous: Aerial Duels: Won%'
]
```

Figure 1.4: Definition of HEADER_ORDER column order in config.py (excerpt).

5. **Export Fields List (`EXPORT_STATS`) & Group by Table (`STATS_BY_TABLE`):** These variables are automatically generated from `HEADER_ORDER` and `HEADER_MAP` to help `main.py` know exactly which `data-stat` to retrieve from `scraper.py` and from which table.

```
EXPORT_STATS = []
fetched_stats = set()
for col in HEADER_ORDER:
    stat = HEADER_MAP.get(col)
    if stat:
        EXPORT_STATS.append(stat)
        fetched_stats.add(stat)
    else:
        print(f"[ERROR] Config: Không tìm thấy mapping data-stat cho cột CSV mẫu: '{col}'.")

STATS_BY_TABLE = {
    TABLE_IDS['standard']: [
        'player', 'nationality', 'position', 'team', 'age', 'games', 'games_starts', 'minutes',
        'goals', 'assists', 'cards_yellow', 'cards_red', 'xg', 'xg_assist',
        'progressive_carries', 'progressive_passes', 'progressive_passes_received',
        'goals_per90', 'assists_per90', 'xg_per90', 'xg_assist_per90'
    ],
```

Figure 1.5: Creation of EXPORT_STATS and STATS_BY_TABLE in config.py (excerpt).

1.5.2 Data Processing and Cleaning

Data processing is performed in several places in the program:

1. **Safe Number Conversion (`scraper.safe_cast_int`):** This function is used when retrieving minutes and other statistics from HTML. It removes commas, converts to int, and if an error or "N/a" is encountered, it returns 0 and reports an error.

```python
def safe_cast_int(value_str: Optional[str], default: int = 0) -> int:
    if not value_str or value_str == 'N/a':
        return default
    try:
        cleaned_str = value_str.replace(',', '')
        return int(cleaned_str)
    except (ValueError, TypeError):
        print(f"[LOI] Khong the chuyen '{value_str}' sang so nguyen.")
        return default
```

Figure 1.6: The safe_cast_int function in scraper.py.

2. **Handling Missing Data during Parsing (`scraper.py` - logic in functions):** When `scraper.py` reads each `<td>` cell, if it doesn't find a cell corresponding to a `data-stat`, it will assign the value 'N/a'.

3. **Filtering by Minutes (`scraper.get_players_from_table`):** Only players with `minutes > 90` are kept after being retrieved from the 'standard' table.

4. **Handling Missing Data during Export (`scraper.Player.export`):** When exporting data to CSV, if a player is missing a certain statistic (because it wasn't in the original or supplementary table), the `export` method will automatically fill 'N/a' in that position.

5. **Cleaning Numbers in DataFrame (`main.clean_numeric_commas`):** Before saving the CSV file, this function iterates through the DataFrame columns, finds object (string) type columns, removes commas, and attempts to convert the entire column to a numeric type.

```python
def clean_numeric_commas(df: pd.DataFrame) -> pd.DataFrame:
    # Làm sạch dấu ',' nếu là dấu ngăn cách hàng nghìn
    for col in df.columns:
        if df[col].dtype == object:
            # Xóa dấu ',' trong các chuỗi
            df[col] = df[col].str.replace(',', '', regex=False)
            # Cố gắng chuyển sang số nếu hợp lệ
            df[col] = pd.to_numeric(df[col], errors='ignore')
    return df
```

Figure 1.7: The clean_numeric_commas function in main.py.

1.5.3 Core Collection Functions (`scraper.py`)

This file contains the functions and classes that perform the data "scraping".

1. **Player Class:** Stores player data using a dictionary, has `update()` and `export()` methods.

```python
class Player:
    def __init__(self, **kwargs: Any):
        self.data: Dict[str, Any] = kwargs

    def update(self, **kwargs: Any) -> None:
        self.data.update(kwargs)

    def export(self, export_keys: List[str]) -> List[Any]:
        return [self.data.get(key, 'N/a') for key in export_keys]

    def __repr__(self) -> str:
        name = self.data.get('player', 'Unknown Player')
        team = self.data.get('team', 'Unknown Team')
        age = self.data.get('age', 'N/A')
        return f"<Player: {name} ({age} - {team})>"
```

Figure 1.8: Definition of the Player class in scraper.py.

2. **get_soup(...) function:** Uses Selenium to open the web page, wait, scroll the page, get HTML, and parse with BeautifulSoup, with a retry mechanism.

```python
def get_soup(url: str, table_id: str, retries: int = 3, delay: int = 5) -> Optional[BeautifulSoup]:
    options = Options()
```

Figure 1.9: Definition of the get_soup function in scraper.py (declaration part only).

3. **get_players_from_table(...) function**: Gets the initial list of Players from the 'standard' table and filters by minutes played.

```python
def get_players_from_table(url: str, table_id: str, fetch_fields: List[str], min_mins: int) -> List[Player]:
    print(f"[THONG TIN] Dang lay du lieu cau thu tu bang '{table_id}' tai {url}...")
    soup = get_soup(url, table_id)
```

Figure 1.10: Definition of the get_players_from_table function in scraper.py (excerpt).

4. **update_players(...) function**: Updates the original Player list with additional data from supplementary tables.

```python
def update_players(players: List[Player], url: str, table_id: str, update_fields: List[str]) -> None:
    print(f"[THONG TIN] Dang cap nhat du lieu tu bang '{table_id}' tai {url}...")
    soup = get_soup(url, table_id)
```

Figure 1.11: Definition of the update_players function in scraper.py (excerpt).

1.5.4 Main Execution Function (`main.py`)

The `main()` function in the `main.py` file coordinates all steps.

```python
import pandas as pd
import time
# ... (imports khác và các hàm tiện ích)
from config import *
from scraper import *
def main():
    start_time = time.time()
    print("[INFO] Bat dau...")
    # 1. Chuẩn bị: Xác định các trường cần lấy từ config
    # ... (code xác định base_req_fields, addtl_tables) ...
    # 2. Gọi hàm lấy dữ liệu gốc
    players = get_players_from_table(...)
    # ... (Kiểm tra lỗi) ...
    # 3. Gọi hàm cập nhật lặp qua các bảng phụ
    for table_info in addtl_tables:
        update_players(...)
        print("-" * 30)
    # 4. Sắp xếp
    sorted_players = sorted(players, ...)
    # 5. Tạo DataFrame
    data_to_export = [p.export(EXPORT_STATS) for p in sorted_players]
    df = pd.DataFrame(data_to_export, columns=HEADER_ORDER)
    # 6. Làm sạch DataFrame
    df = clean_numeric_commas(df)
    # 7. Lưu CSV
    try:
        df.to_csv(OUT_FILE, index=False, encoding='utf-8-sig')
        print(f"[SUCCESS] Luu file {OUT_FILE} thanh cong!")
        # ... (In kết quả ra console) ...
    except Exception as e:
        print(f"[ERROR] Loi khi luu CSV: {e}")
if __name__ == '__main__':
    main()
```

Figure 1.12: The main function in main.py.

1.6 Results

After successfully running the file `main.py`, I obtained the main output file for Part 1, which is `results.csv`. This file contains data on 494 players who played more than 90 minutes, along with all required statistics.
Below is a sample excerpt from the results.csv file when opened in Excel.

Player	Nation	Team	Position	Age	Playing Time: matches played	Playing Time: starts	Playing Time: minutes
Aaron Cresswell	eng ENG	West Ham	DF	35-144	15	8	676
Aaron Ramsdale	eng ENG	Southampton	GK	26-359	27	27	2430
Aaron Wan-Bissaka	eng ENG	West Ham	DF	27-163	33	32	2884
Abdoulaye Doucouré	ml MLI	Everton	MF	32-127	30	29	2425
Abdukodir Khusanov	uz UZB	Manchester City	DF	21-068	6	6	503
Abdul Fatawu Issahaku	gh GHA	Leicester City	FW	21-061	11	6	579
Adam Armstrong	eng ENG	Southampton	FWMF	28-087	20	15	1248
Adam Lallana	eng ENG	Southampton	MF	36-363	14	5	361
Adam Smith	eng ENG	Bournemouth	DF	34-009	22	17	1409
Adam Webster	eng ENG	Brighton	DF	30-124	11	8	617
Adam Wharton	eng ENG	Crystal Palace	MF	20-340	20	16	1318
Adama Traoré	es ESP	Fulham	FWMF	29-103	33	16	1592
Albert Grønbaek	dk DEN	Southampton	FWMF	23-350	4	2	143
Alejandro Garnacho	ar ARG	Manchester Utd	MFFW	20-311	34	23	2146

Figure 1.13: Sample excerpt from the results.csv file

Chapter 2

Data Analysis and Visualization

After completing the data collection and creating the results.csv file in Part I, Part 2 moves on to the steps of analyzing and visualizing the dataset. This content aims to clarify the characteristics of the data, explore outstanding players and teams, and examine the distribution of some important indicators.

2.1 Program Structure

Similar to Part I, Part II is also divided into main Python modules:

- **main2.py**: Reads results.csv, finds the top 3 highest and lowest players for each numerical statistic, writes the results to top_3.txt.
- **part_2.py**: Reads results.csv, calculates Median, Mean, Std Dev for each indicator for the entire league ('all') and for each team, saves the results to results2.csv.
- **part_3.py**: Reads results.csv, plots histogram charts (overall distribution and by team) for some selected statistics (selected_stats), saves the charts as image files.
- **part_4.py**: Reads results2.csv, analyzes to find the team leading in the most average statistics, and prints this evaluation result to the screen.

2.2 Library Selection

To perform the analysis and visualization requirements in Part 2, I used the following main Python libraries:

- **Pandas**: I chose this library because it is the powerful tool for working with tabular data. In Part 2, Pandas helps me read CSV files (results.csv and results2.csv), clean and prepare data (such as converting data types, handling commas), select columns, filter out missing values (dropna), sort data to find top/bottom (sort_values), group data by team (groupby) and calculate aggregate statistics (agg), and finally create the result DataFrame (results2.csv).
- **NumPy**: This library is the foundation for Pandas, providing strong support for numerical computations. I use it (often through Pandas) to handle NaN values and perform calculations on numerical columns efficiently.

- **Matplotlib (pyplot)**: I chose this library because it is the standard for plotting graphs in Python. In part_3.py, I use Matplotlib to set basic chart parameters (size, title, axis labels) and to save histogram charts as image files.
- **Seaborn**: I use Seaborn because it is built on Matplotlib but provides more beautiful and simpler statistical plotting functions. Specifically in part_3.py, I use seaborn.histplot to plot frequency distribution histograms and seaborn.FacetGrid to easily create a grid of histogram charts for each football team.
- **Os**: This library helps me interact with the operating system, mainly for managing file paths. I use os.path.join, os.path.dirname to create correct paths to input data files (results.csv, results2.csv) and output directories/files (top_3.txt, directory containing histograms) flexibly, independent of the machine running the code.
- **Other auxiliary libraries**: In addition, I use traceback to help display detailed errors when the program encounters problems, re (Regular Expressions) and math for some small utilities in part_3.py (cleaning filenames, calculating layout), and typing to make the code clearer.

2.3 Overall Implementation Process

To analyze and visualize data from the results.csv file (result of Part 1), I performed the following main work steps, corresponding to different Python scripts:

1. **Prepare Foundational Data**: The first step for any analysis is to read the results.csv file using Pandas, identify columns containing numerical data, and perform basic cleaning steps such as removing commas, handling 'N/a' values as NaN, and converting these columns to numeric data types for calculation.
2. **Find Outstanding Players (Top/Bottom 3)**: I run the `main2.py` script. This script iterates through each prepared numerical column, removes NaN values, then sorts to find the 3 players with the highest values and the 3 players with the lowest values. The result of this step is recorded in detail in the `top_3.txt` file.
3. **Calculate Descriptive Statistics**: I run the `part_2.py` script. This script calculates important statistical values: Median, Mean, and Standard Deviation for each indicator. This calculation is performed at two levels: one for all players in the league (creating an 'all' row), and another calculated separately for each football team (using groupby). The final result is compiled and saved to the `results2.csv` file with a clear column structure.
4. **Plot Distribution Histograms**: I run the `part_3.py` script. This script focuses on plotting histogram charts using Seaborn and Matplotlib for some selected representative offensive and defensive/playmaking statistics that I have chosen (selected_stats). It plots the overall distribution chart for each statistic, and more importantly, plots a grid-style chart (FacetGrid) to compare the distribution of that statistic among different football teams. These charts are saved as PNG image files.

5. **Preliminary Team Performance Evaluation**: Finally, I run the `part_4.py` script. This script reads the `results2.csv` file (result of step 3), focusing on the Mean value columns for each team. It identifies the team with the highest average value for each statistic (or lowest for negative statistics like GA90), then counts which team leads in the most statistics. The result and a preliminary conclusion about the team with the best performance according to this criterion are printed to the console.

2.4 Detailed Analysis of Components

2.4.1 Preparing Data for Analysis

This is a foundational step performed at the beginning of the scripts (main2.py, part_2.py, part_3.py) to ensure the data is ready for analysis.

- **Read and basic cleaning**: I use Pandas to read the results.csv file from Part 1, automatically recognizing 'N/a' as missing values (NaN), and cleaning excess whitespace in column names and team names.
- **Identify and normalize numeric columns**: I filter out columns containing statistical indicators (removing identifier columns). For these columns, I perform:
  - Remove commas in large numbers (`.str.replace(',', '')`).
  - Force conversion to numeric type using `pd.to_numeric(errors='coerce')`. Error values will become NaN.
  - Save the names of valid numeric columns to a list (stat_columns or numeric_cols).

columns (Part II) —

```
stat_columns = []
for col in df.columns:
    if col in non_stat_cols:
        continue

    # Loại bỏ dấu ',' rồi chuyển sang kiểu số
    clean_col = df[col].astype(str).str.replace(',', '', regex=False)
    numeric_col = pd.to_numeric(clean_col, errors='coerce')

    if not numeric_col.isnull().all():
        df[col] = numeric_col
        stat_columns.append(col)
```

Figure 2.1: Code snippet for normalizing numeric columns in Part II scripts.

2.4.2 Finding Top/Bottom 3 Players (`main2.py`)

In this part, I find the 3 best and worst performing players for each indicator. **Implementation**:

1. Import libraries (pandas, numpy, os, traceback). Define input (results.csv) and output (top_3.txt) file paths.

2. Read and prepare data as in section 2.4.1.
3. Open the `top_3.txt` file for writing.
4. Main loop: Iterate through each `stat` in `stat_columns`.
   - Write the indicator name.
   - Create `df_stat` containing only 'Player' and `stat`, remove NaN (`.dropna()`).
   - Check and handle if data is empty or has only 1 unique value (`nunique() == 1`).
   - Use `sort_values()` and `head(3)` to get top 3 (highest) and bottom 3 (lowest).
   - Write the results (Name, Value) to the file.

```
# --- Buoc 4: Xu ly top 3 va ghi ra file ---
try:
    with open(output_path, 'w', encoding='utf-8') as f:
        print("Dang xu ly va ghi top 3...")
        for stat in stat_columns:
            f.write(f"Chi so: {stat}\n")
            df_stat = df[[player_column, stat]].dropna()
            if df_stat.empty:
                f.write("Khong co du lieu hop le\n\n")
                continue
            if df_stat[stat].nunique() == 1:
                f.write(f"Tat ca cau thu cung gia tri: {df_stat[stat].iloc[0]}\n")
                top_3 = bottom_3 = df_stat.head(3)
            else:
                top_3 = df_stat.sort_values(by=stat, ascending=False).head(3)
                bottom_3 = df_stat.sort_values(by=stat, ascending=True).head(3)
            f.write("3 cau thu diem cao nhat:\n")
            for _, row in top_3.iterrows():
                f.write(f"   {row[player_column]}: {row[stat]}\n")
            f.write("3 cau thu diem thap nhat:\n")
            for _, row in bottom_3.iterrows():
                f.write(f"   {row[player_column]}: {row[stat]}\n")
            f.write("\n")
    print(f"Hoan thanh. Da luu ket qua vao: {output_path}"
except Exception as e:
    print("Loi khi ghi file:")
!   traceback.print_exc()
```

Figure 2.2: Code snippet for processing and writing Top/Bottom 3 in main2.py.

Results: After the program finishes running, the results are saved to the `top_3.txt` file.

```
Statistic: Playing Time: matches played
3 players with the highest score:
Tyrick Mitchell: 35
David Raya: 35
Dean Henderson: 35
3 players with the lowest score:
Ayden Heaven: 2
Ben Godfrey: 2
```

```
Billy Gilmour: 2

Statistic: Playing Time: starts
3 players with the highest score:
Youri Tielemans: 35
Tyrick Mitchell: 35
Dean Henderson: 35
3 players with the lowest score:
Ben Chilwell: 0
Sergio Reguilon: 0
Solly March: 0
...
```

2.4.3 Calculating Descriptive Statistics - part_2.py

The objective is to calculate the median, mean, and standard deviation for each indicator, applied to all players and to each team.
Implementation:

- **Identify Numeric Columns (`numeric_cols`)**: In the data preparation step (Step 2 in the file), the script performs detection and type conversion of key columns into numeric format. This process includes DEBUG print statements to log the status and results of each conversion step.
- **Calculation**:
  - *Overall*: Calculate overall statistics (median, mean, std) for numeric columns using `df[numeric_cols].agg([...]).T`. The `.agg()` method calculates 3 values simultaneously, `.T` transposes so that each row is an indicator.
  - *Per Team*: Group data by `'Team'` then use `.agg([...],numeric_only=True)` to calculate median, mean, std for numeric columns. The result is `team_stats`, a DataFrame with a `MultiIndex`.
- **Create Result DataFrame**:
  - `all_row_df`: This code snippet creates a dictionary `all_row_data`, then iterates through `numeric_cols`. For each col, it creates a new column name (e.g., `f'Median of {col}'`) and retrieves the corresponding value from `overall_stats.loc[col, 'median']` (or 'mean', 'std'). Finally, it creates a DataFrame from this dictionary.
  - `team_results_df`: This code snippet is more complex due to handling the MultiIndex from `team_stats`. Iterate through numeric columns (`numeric_cols`), create a `MultiIndex` key to access the corresponding column in `team_stats`. Use `reindex(team_names).values` to get an array of values in the correct team order, assign to `team_results_data` (handling missing keys with `pd.NA`). Finally, create a `DataFrame` from the `dictionary` `team_results_data`.
  - `final_results = pd.concat(...)`: Concatenate `all_row_df` and `team_results_df` vertically.

- **Save file**: Use `final_results.to_csv(STATS_SUMMARY_FILE, ...)` to save the results. The parameter `float_format='%.3f'` helps to round decimal numbers.

Results: After running, the results are saved to the `results2.csv` file.

Team	Median of Playing Time: matches played	Mean of Playing Time: matches played	Std of Playing Time: matches played
all	23	21.223	9.976
Arsenal	23.5	23.273	8.066
Aston Villa	20.5	19.429	10.333
Bournemou	26	22.217	10.18
Brentford	28	23.524	11.383
Brighton	21	19.536	10.035
Chelsea	18	19.692	11.263
Crystal Pala	30	24.048	10.632
Everton	24	22.409	9.287
Fulham	27	24.5	9.329
Ipswich Tow	18.5	18.2	9.015
Leicester Ci	21.5	20.231	9.717
Liverpool	28	25.19	8.925
Manchester	23	19.88	8.974
Manchester	20	17.667	11.454
Newcastle U	27	23.217	10.144
Nott'ham Fo	30	24.5	10.888
Southamptc	20	18.655	10.379
Tottenham	22	19.333	9.232
West Ham	20	21.56	8.53
Wolves	26	22.783	8.924

Figure 2.3: Sample snippet from the results2.csv file

2.4.4 Visualizing Distributions (`part_3.py`)

In this part, I plot histogram charts. **Statistic Selection**: I have selected the following 6 statistics to plot histograms, representing important aspects of attack and play development:

```
selected_stats = [
    'Expected: xG',
    'Per 90 minutes: Gls',
    'Shooting: Standard: SoT%',
    'Performance: assists',
    'Passing: Expected: KP',
    'Passing: Expected: PrgP'
]
```

Reasons for selection:

- **Expected: xG** (Expected Goals): This indicator measures the quality of chances a team's players create, not just counting actual goals. It helps assess attacking ability sustainably, whether the team is creating "good" chances, independent of temporary finishing ability. The distribution of xG shows the average danger level created by the team's players.

- **Per 90 minutes: Gls** (Goals/90 minutes): This is the most direct measure of scoring efficiency, standardized by playing time. It indicates how often a player "scores" when they are on the field. The distribution of this indicator helps compare scoring ability between team players more fairly.

- **Shooting: Standard: SoT%** (Shots on Target Percentage): This indicator reflects the accuracy of shots. A high percentage indicates that the team's players frequently hit the target, causing difficulties for the opposing goalkeeper. Its distribution shows the efficiency of converting chances into dangerous shots.

- **Performance: assists** (Assists): Directly measures the ability to create goals for teammates. The distribution of assists shows which team players have good playmaking ability and can create decisive passes.
- **Passing: Expected: KP** (Key Passes): The number of passes that directly lead to a teammate's shot. This indicator reflects the ability to create chances (not necessarily resulting in a goal). The distribution of KP shows how often team players put teammates in a position to shoot.
- **Passing: Expected: PrgP** (Progressive Passes): The number of successful passes that move the ball significantly forward (e.g., into the opponent's final third or a certain distance). This indicator measures ball progression ability, breaking down the opponent's defensive structure. The distribution of PrgP shows which team tends to and is capable of moving the ball upfield quickly and effectively.

Implementation:

- **Setup:** Import libraries, define `selected_stats`, `OUTPUT_FOLDER`...
- **Utility functions:** `create_folder`, `sanitize_filename`, `read_csv_file`.
- **Data preparation:** Read `results.csv`. Select columns, cast to numeric type, and importantly, `fillna(0)`.

```
if df is not None:
    missing_required = [col for col in [TEAM_COLUMN] + selected_stats if col not in df.columns]
    if TEAM_COLUMN not in df.columns:
        print(f"Loi nghiem trong: Khong tim thay cot doi '{TEAM_COLUMN}'. Khong the tiep tuc.")
        df = None
    elif missing_required:
        print(f"Canh bao: Cac cot sau khong tim thay trong CSV va se bi bo qua: {', '.join(missing_required)}")
        selected_stats = [stat for stat in selected_stats if stat in df.columns]

    if df is not None and selected_stats:
        required_columns = [TEAM_COLUMN] + selected_stats
        try:
            cleaned_data = df[required_columns].copy()
            print(f"Bat dau xu ly {len(selected_stats)} chi so: {', '.join(selected_stats)}")
            for stat in selected_stats:
                numeric_col = pd.to_numeric(cleaned_data[stat], errors='coerce')
                if not numeric_col.isnull().all():
                    print(f"  -> Xu ly cot '{stat}': Chuyen sang so va fillna(0).")
                    cleaned_data[stat] = numeric_col.fillna(0)
                    valid_stat_columns.append(stat)
                else:
                    print(f"  -> Canh bao: Cot '{stat}' khong chua du lieu so hop le hoac toan NaN. Da loai bo.")
                    if stat in cleaned_data.columns:
                        cleaned_data.drop(columns=[stat], inplace=True)
            if not valid_stat_columns:
                print("Khong con cot chi so nao hop le de ve bieu do.")
                cleaned_data = None
            else:
                print(f"Cac cot chi so hop le se duoc ve: {', '.join(valid_stat_columns)}")
```

Figure 2.4: Code snippet for data preparation in part_3.py.

- **Plotting functions:**
  - `plot_histogram_all_players_facet`: Plots overall histogram.

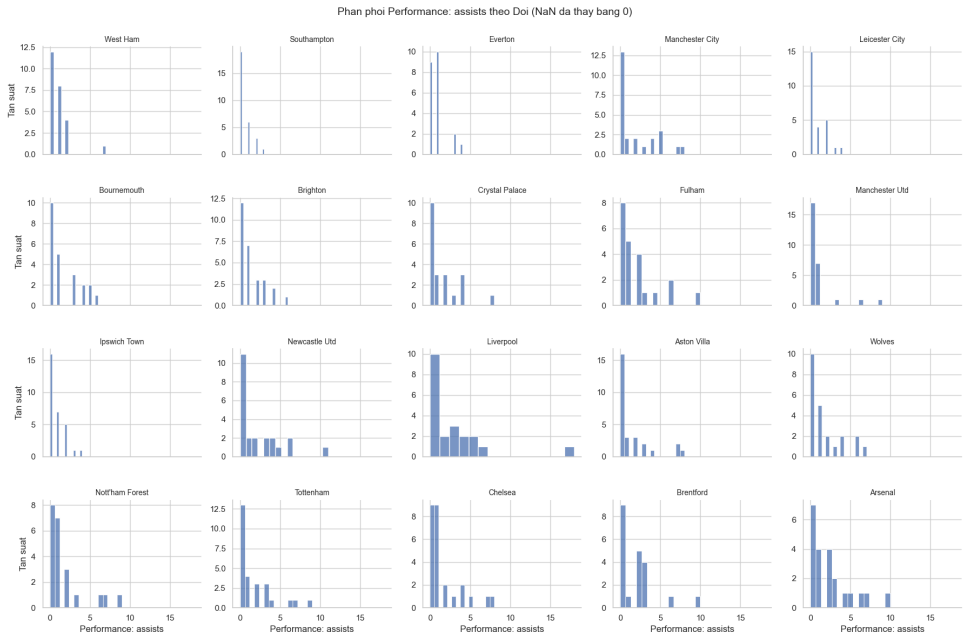

Figure 2.5: Result of assists histogram by team.

  - `plot_histograms_per_team_facet`: Plots grid of histograms by team.

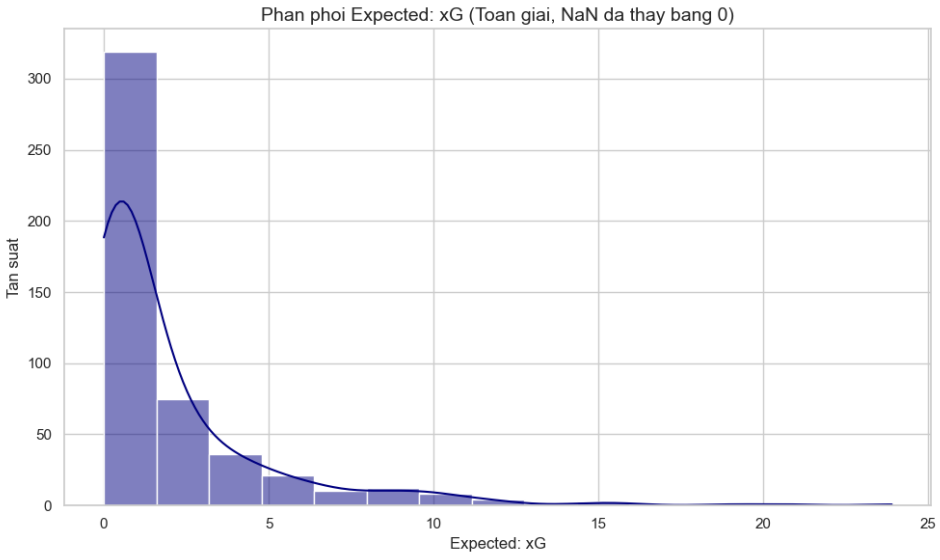

Figure 2.6: Overall league distribution for Expected xG.

Results: PNG image files are saved in the 'histograms' directory.

2.4.5 Identifying the Best Performing Team (By Average Stats) - `part_4.py`

Identifying the leading team by each average indicator: For each statistical indicator, I identify the football team with the highest mean value. Using the mean value helps reflect the "average" performance of the players in that team for the corresponding indicator.
Implementation:

1. **Read data:** Load data from the `results2.csv` file into a pandas DataFrame.
2. **Preprocessing:**
   - Set the column containing team names ('Team') as the DataFrame's index for easy access by team name.
   - Remove the 'all' row (if any) to analyze data for specific teams only.
3. **Identify average indicator columns:** Filter out a list of columns whose names start with "Mean of " in the DataFrame.
4. **Iterate and find the highest value:**
   - Iterate through each column in the list of average columns.
   - Get the original name of the indicator (e.g., from "Mean of Performance: goals" get "Performance: goals").
   - Convert the column's data to numeric type, handling invalid values as NaN (`pd.to_numeric, errors='coerce'`).
   - Use the `idxmax(skipna=True)` function to find the team name (index) with the largest value in the current data column (skipping NaN).
   - Use the `max(skipna=True)` function to get that largest value.
   - Store the leading team name and corresponding value in a dictionary.
   - Handle cases with no valid data (all NaN).
5. **Display leading list:** Print a detailed list showing which team leads each average indicator and the specific value.
6. **Statistics of leading occurrences:**
   - Extract all saved leading team names.
   - Use `pandas.Series.value_counts()` to count the number of times each team appears in this list.
   - Print a table of statistics of the number of leading occurrences for each team.
7. **Identify the best team:**
   - Find the team name with the highest number of leading occurrences from the statistics in step 6 (`idxmax()`).
   - Print an initial conclusion about the most outstanding team based on the highest number of leading occurrences.

8. **Check important indicators:**

   - For a predefined list of important indicators (such as goals, assists, xG, GA90, Save%), display the leading team (highest or lowest depending on the indicator).
   - Specifically, for Goalkeeping: Performance: GA90, use `idxmin()` and `min()` to find the team with the lowest value (best).

9. **Completion:** End the analysis process and display the results to the console.

Results:

```
Team with the highest average score for each statistic:
Playing Time: matches played: Liverpool (24.48)
Playing Time: starts: Brentford (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 (8.23)
Expected: xG: Liverpool (3.63)
Expected: 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 (8.19)
Per 90 minutes: xAG: Chelsea (0.15)
Goalkeeping: Performance: GA90: Leicester City (2.73)
Goalkeeping: Performance: Save%: Bournemouth (80.00)
Goalkeeping: Performance: CS%: Brentford (59.10)
Goalkeeping: Penalty Kicks: Save%: Everton (100.00)
Shooting: Standard: SoT%: Nott'ham Forest (38.99)
Shooting: Standard: SoT/90: Fulham (9.54)
Shooting: Standard: G/Sh: Arsenal (8.14)
Shooting: Standard: Dist: Nott'ham Forest (19.09)
Passing: Total: Cmp: Liverpool (778.18)
Passing: Total: Cmp%: Manchester City (86.55)
Passing: Total: TotDist: Liverpool (4445.05)
Passing: By Distance: Short Cmp%: Manchester City (92.15)
Passing: By Distance: Medium Cmp%: Manchester City (89.58)
Passing: By Distance: Long Cmp%: Liverpool (60.32)
Passing: Expected: KP: Liverpool (22.00)
Passing: Expected: 1/3: Liverpool (67.71)
Passing: Expected: PPA: Liverpool (18.62)
Passing: Expected: CrsPA: Fulham (4.23)
Passing: Expected: PrgP: Liverpool (81.38)
Goal and Shot Creation: SCA: SCA: Liverpool (49.81)
```

Goal and Shot Creation: SCA: SCA90: Liverpool (2.63)
Goal and Shot Creation: GCA: GCA: Liverpool (6.48)
Goal and Shot Creation: GCA: GCA90: Liverpool (0.35)
Defensive Actions: Tackles: Tkl: Crystal Palace (32.00)
Defensive Actions: Tackles: TklW: Crystal Palace (18.62)
Defensive Actions: Challenges: Att: Liverpool (28.29)
Defensive Actions: Challenges: Lost: Crystal Palace (13.48)
Defensive Actions: Blocks: Blocks: Brentford (20.38)
Defensive Actions: Blocks: Sh: Brentford (8.38)
Defensive Actions: Blocks: Pass: Crystal Palace (14.14)
Defensive Actions: Blocks: Int: Bournemouth (14.00)
Possession: Touches: Touches: Liverpool (1102.05)
Possession: Touches: Def Pen: Brentford (142.62)
Possession: Touches: Def 3rd: Brentford (357.33)
Possession: Touches: Mid 3rd: Liverpool (497.52)
Possession: Touches: Att 3rd: Manchester City (343.48)
Possession: Touches: Att Pen: Liverpool (55.81)
Possession: Take-Ons: Att: Arsenal (29.73)
Possession: Take-Ons: Succ%: Liverpool (54.91)
Possession: Take-Ons: Tkld%: Leicester City (48.30)
Possession: Carries: Carries: Manchester City (643.48)
Possession: Carries: PrgDist: Manchester City (1994.16)
% PrgC missing in this PDF snippet
Possession: Carries: 1/3: Manchester City (30.36)
Possession: Carries: CPA: Manchester City (14.04)
Possession: Carries: Mis: Nott'ham Forest (23.80)
Possession: Carries: Dis: Newcastle Utd (17.65)
Possession: Receiving: Rec: Liverpool (769.67)
Possession: Receiving: PrgR: Liverpool (80.62)
Miscellaneous: Performance: Fls: Bournemouth (19.83)
Miscellaneous: Performance: Fld: Newcastle Utd (17.61)
Miscellaneous: Performance: Off: Nott'ham Forest (3.73)
Miscellaneous: Performance: Crs: Fulham (36.82)
Miscellaneous: Performance: Recov: Bournemouth (71.44)
Miscellaneous: Aerial Duels: Won: Brentford (26.76)
Miscellaneous: Aerial Duels: Lost: Crystal Palace (26.57)
Miscellaneous: Aerial Duels: Won%: Southampton (54.17)

2.4.6 Identifying the Overall Best Performing Team - `part_4.py`

From the results in section 2.4.5, the number of times each team leads in average indicators is statistically summarized as follows:

Team	Number of times leading
Liverpool	27
Manchester City	10
Brentford	7
Bournemouth	5
Crystal Palace	5
Nott'ham Forest	4
Fulham	3
Arsenal	3
Newcastle Utd	2
Leicester City	2
Aston Villa	1
Chelsea	1
Everton	1
Southampton	1

Table 2.1: Statistics of the number of times teams lead in average indicators.

Based on the number of leading average indicators, 'Liverpool' appears the most.

Checking important indicators:

- Highest Performance: goals: Liverpool (3.76)
- Highest Performance: assists: Liverpool (2.81)
- Highest Goalkeeping: Performance: Save%: Bournemouth (80.00)

Conclusion: Based on the analysis of the number of statistical indicators with the highest average values, Liverpool leads in the most categories (27 times) and is therefore considered the team with the best performance in the 2024-2025 season according to this evaluation method.

2.5 Results

After the program finishes running, the results of Part II are stored as follows:

- **top_3.txt**: list of top 3 highest and lowest players for each statistical indicator
- **results2.csv**: summary table of median, mean, standard deviation values for each indicator calculated for the entire league and by team
- **Histograms directory**: contains png image files which are histogram distribution charts of selected indicators, for the entire league and for each team.
- **Console output:** displays the team with the highest score for each indicator and the result of the overall highest scoring team

Chapter 3

Clustering and Dimensionality Reduction

3.1 Problem Analysis

According to the requirements of Part III, the main tasks include:

- **Player Clustering:** Use the K-means algorithm to automatically group players with similar statistical characteristics into the same cluster.
- **Determine Optimal Number of Clusters:** Find the most suitable number of clusters (k) for the dataset and explain the basis for that choice.
- **Dimensionality Reduction and Visualization:** Apply Principal Component Analysis (PCA) to reduce the dimensionality of the data to 2, then plot a 2D chart to visualize the classified player clusters.
- **Evaluation and Comments:** Comment on the obtained clustering and visualization results.

3.2 Library Selection

To perform the above requirements, I used the following libraries:

- **Pandas:** Used to read, process, and manipulate player data in tabular form (DataFrame).
- **NumPy:** Provides array structures and fundamental mathematical functions for numerical computations.
- **Scikit-learn:** Provides machine learning algorithms and preprocessing tools (KMeans, PCA, StandardScaler, OneHotEncoder, SimpleImputer, Pipeline, ColumnTransformer).
- **Matplotlib & Seaborn:** Used to create visualization charts, including the Elbow plot and 2D clustering plot.
- **os:** Supports file path management.

3.3 Overall Process

I divided the analysis and implementation process into 2 modules: `kmeans_pca.py` and `plotting.py`:

1. **Load data:** Load data from the results.csv file (result of Part 1).
2. **Data preprocessing:** Perform necessary cleaning, normalization, and encoding steps to prepare the data for machine learning algorithms.
3. **Determine number of clusters (k) using the Elbow method:** Run K-means with various k values and plot the Elbow chart (elbow_plot.png) to find the "elbow point".
4. **K-means clustering:** Apply K-means with the chosen optimal number of clusters to assign cluster labels to each player. Save the results to `results_clustered.csv`.
5. **Dimensionality reduction using PCA:** Apply PCA with n_components = 2 to reduce the dimensionality of the preprocessed data. Save the PCA results to `pca_clusters.csv`.
6. **Visualize clustering results:** Use plotting.py to draw a 2D scatter plot, save the result to the file `kmeans_pca_cluster_plot_with_variance.png`, showing players on the PCA plane and colored according to their classified cluster.

3.4 Detailed Analysis

3.4.1 Data Preprocessing (`kmeans_pca.py`)

This is a foundational step I perform to prepare data from results.csv for the K-means and PCA algorithms. This process includes several stages:

- **Load data:** Data is loaded into a Pandas DataFrame from the results.csv file.
- **Identify column data types:** I configured the script to automatically identify columns with numerical data types (np.number) and identify expected categorical columns (`categorical_cols = ['Nation', 'Team', 'Position']`). The player identifier column (the first column) is also removed from the dataset used for analysis (features).
- **Clean numerical data:** I processed numerical columns by iterating through them to:
  - Replace 'N/a' values with NaN.
  - If the column is in object (string) format, remove the comma (,) thousand separator.
  - Convert the column to a numeric type using `pd.to_numeric`, values that cannot be converted will become NaN (`errors='coerce'`).

```python
for col in numerical_cols:
    if col in features.columns:
        features[col] = features[col].replace('N/a', np.nan)
        if not pd.api.types.is_numeric_dtype(features[col]):
            print(f"  Xu ly cot so dang chuoi/object: {col}")
            features[col] = features[col].astype(str).str.replace(',', '', regex=False)
            features[col] = pd.to_numeric(features[col], errors='coerce')
        else:
            features[col] = pd.to_numeric(features[col], errors='coerce')
```

Figure 3.1: Code snippet for cleaning numerical columns in kmeans_pca.py.

- **Handle missing values (Imputation):** I used SimpleImputer from Scikit-learn:
  - For numerical columns, NaN values are replaced with the mean (`strategy='mean'`) of that column.
  - For categorical columns, NaN values are replaced with the most frequent value (`strategy='most_frequent'`) in that column.
- **Normalize numerical data (Scaling):** Apply StandardScaler to all numerical columns after handling missing values. This step transforms the data so that each numerical column has a mean of 0 and a standard deviation of 1, ensuring that features contribute fairly to the results.
- **Encode categorical data (Encoding):** I applied OneHotEncoder to categorical columns ('Nation', 'Team', 'Position') to convert them into a binary numerical format that the algorithm can process.
- **Combine steps using Pipeline and Column Transformer:**
  - Two separate Pipelines were created: one for processing numerical columns (Imputer -> Scaler) and one for processing categorical columns (Imputer -> OneHotEncoder).
  - I used ColumnTransformer to apply the correct processing pipeline to the correct group of columns and combine the results. Unspecified columns will be dropped (`remainder='drop'`).

```python
numerical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='mean')),
    ('scaler', StandardScaler())
])

categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore', sparse_output=False))
])

preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, numerical_cols),
        ('cat', categorical_transformer, categorical_cols)
    ],
    remainder='drop'
)
```

Figure 3.2: Definition of Pipeline and ColumnTransformer in kmeans_pca.py.

- **Preprocessing result:** The output of this entire process is the `processed_data` variable, a NumPy array containing data I have prepared, ready for the subsequent clustering and dimensionality reduction steps.

3.4.2 Determining the Optimal Number of Clusters (Elbow Method) (kmeans_pca.py)

To determine the appropriate number of clusters (k), I applied the Elbow method.

Implementation: I iterated through a range of k values (from 3 to 16). For each k value, the KMeans algorithm was run (`n_init='auto'`, `random_state=42`). The corresponding `inertia_` (WCSS) value for each k was recorded.

Visualization: The inertia values were plotted on an Elbow chart and saved to the file `elbow_plot.png`.

```
# Ve bieu do Elbow
try:
    plt.figure(figsize=(10, 6))
    plt.plot(k_range, inertia, marker='o')
    plt.title('Phuong phap Elbow de xac dinh k toi uu')
    plt.xlabel('So luong cum (k)')
    plt.ylabel('Inertia (Tong binh phuong khoang cach trong cum)')
    plt.xticks(k_range)
    plt.grid(True)
    elbow_plot_path = os.path.join(current_dir, 'elbow_plot.png')
    plt.savefig(elbow_plot_path)
    print(f"Da luu bieu do Elbow vao '{elbow_plot_path}'. Hay xem bieu do nay va chon gia tri 'k' tai diem 'khuyu tay'.")
    plt.close()
except Exception as plot_e:
    print(f"Loi khi ve hoac luu bieu do Elbow: {plot_e}")
```

Figure 3.3: Plotting the Elbow chart.

Choosing k: After the `elbow_plot.png` chart was created and saved from the inertia calculation results, the next step was for me to visually observe this graph to determine the optimal number of clusters (k). The goal of the observation is to find the "elbow point" on the curve representing the inertia value according to the number of clusters k. This "elbow point" is the point where the curve changes slope most significantly – from a steep slope (inertia decreases rapidly as k increases) to a more gradual slope (inertia decreases much more slowly as k continues to increase).

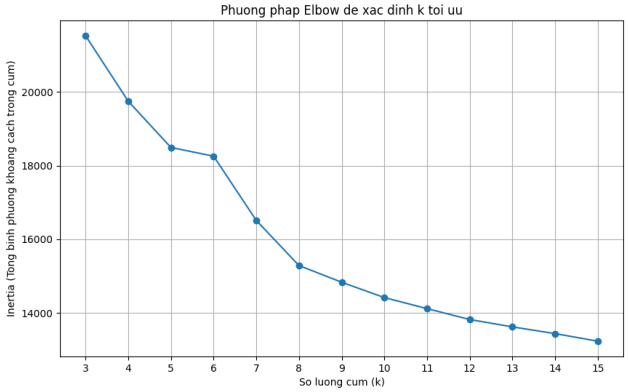

Figure 3.4: Elbow plot to determine optimal k.

Analysis of the chart shows that Inertia decreases sharply as k increases from 3 to 5, after which the rate of decrease slows down considerably. The point k=6 was identified by me as the "elbow point" because this is where the curve begins to flatten out, representing the best balance between minimizing within-cluster variance and avoiding the creation of too many clusters. Increasing k beyond 6 provides diminishing returns in terms of Inertia reduction, therefore, I chose k=6 as the optimal number of clusters to classify player groups.

3.4.3 Implementation and Analysis of Clusters

Implementing K-Means Clustering (`kmeans_pca.py`)

After determining the optimal number of clusters as k=6 from the Elbow method, I applied the K-means algorithm to the preprocessed dataset (`processed_data`).

- **Initialization and Training**: I initialized a `KMeans` object from the Scikit-learn library with parameters `n_clusters=6`, `random_state=42`, and `n_init='auto'` (or a specific n_init value like 10 to ensure stability). The `fit_predict()` method was called on `processed_data` to train the model and assign cluster labels (from 0 to 5) to each player.
- **Save results**: The `cluster_labels` array, containing cluster labels (0, 1, 2, 3, 4, or 5) for each player, was added by me to the original DataFrame (`data`) as a new column named 'Cluster'. This updated DataFrame was then saved to the `results_clustered.csv` file. This file serves as the database for me to perform a detailed characteristic analysis of the 6 clusters in the next section.

Analysis and Interpretation of Clusters

Based on the K-Means clustering results with k=6 and the statistical figures calculated for each cluster, I provide the following preliminary comments on the characteristics of the 6 clusters:

- **Cluster 0 (21 players): Goalkeeper (GK)** This is the smallest cluster with only 21 players (accounting for 4.28% of the total players). The majority of players in this group play in the Goalkeeper (GK) position, playing a crucial role in protecting the goal and not directly participating in attacking plays. Players in this cluster typically have excellent reflexes and help the team maintain a solid defense.
- **Cluster 1 (59 players): Defender (DF)** This cluster has 59 players, accounting for 12.02% of the total players. The majority of players in this cluster play in the Defender (DF) position, who are responsible for defending the defensive area and preventing opponent attacks. Players in this cluster have good game reading ability and provide support from behind for attacking plays.
- **Cluster 2 (100 players): Defender (DF)** This is one of the largest clusters with 100 players, accounting for 20.37% of the total players. Similar to cluster 1, the majority of players in this cluster play in the Defender (DF) position. This cluster represents players whose role is to protect the midfield area and assist in aerial situations.

- **Cluster 3 (99 players): Attack and Support (FW, MF)** With 99 players (accounting for 20.16%), this is a relatively diverse cluster, including players who play in both Forward (FW) and Midfielder (MF) positions. This cluster may include players who participate in organizing attacks and scoring goals. Players in this cluster play an important role in creating scoring opportunities or participating in counter-attacks.

- **Cluster 4 (148 players): Defender (DF)** This is the largest cluster with 148 players, accounting for 30.14% of the total players. The majority of players in this cluster play in the Defender (DF) position. This cluster represents players with excellent defensive ability and play a crucial role in protecting the goal. They help the team maintain safety and solidity throughout the match.

- **Cluster 5 (64 players): Forward (FW)** This cluster has 64 players, accounting for 13.03% of the total players. Primarily, players in this cluster play in the Forward (FW) position. These are players with excellent goal-scoring ability and are always ready to attack the opponent. These players play a crucial role in bringing victory to the team through decisive goals.

3.4.4 Dimensionality Reduction and Visualization using PCA

After grouping players into 6 clusters in section 4.3, I proceed to reduce data dimensionality and visualize these clusters using Principal Component Analysis (PCA) to get a clearer view of their distribution.

- **Implementation steps**:
  - Apply PCA for dimensionality reduction (`kmeans_pca.py`): Use PCA from Scikit-learn to reduce the dimensionality of the preprocessed feature dataset (`processed_data`) down to 2 principal components (PC1 and PC2), keeping `n_components=2`. The result is `pca_components`, containing the new coordinates for each player.
  - Prepare data for visualization (`kmeans_pca.py`): Create a DataFrame `pca_df` from `pca_components`, adding cluster labels. This data is saved to the file `pca_clusters.csv`.
  - Plot visualization chart (`plotting.py`): Use seaborn and matplotlib to read from `pca_clusters.csv` to draw a scatter plot. The x-axis is PC1, the y-axis is PC2. Each point represents a player, and the color indicates the cluster (0 to 5) to which the player belongs.

- **PCA plot:**:

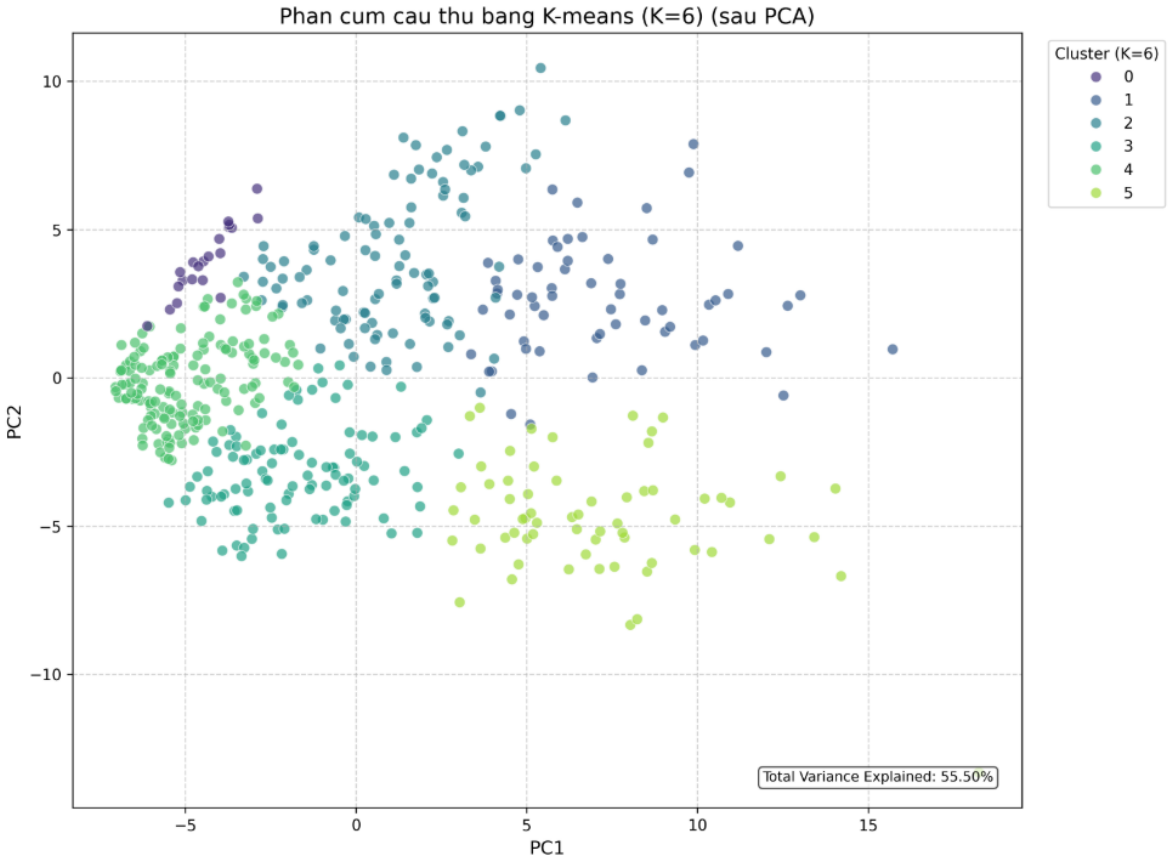

Figure 3.5: Player clustering using K-means (K=6).

Chapter 4

Estimating Player Value using Machine Learning

4.1 Collecting and Preparing Transfer Value Data

To build a database for estimating player value using machine learning methods, the first task is to collect information on players' transfer values. This data includes the current estimated transfer value (Current ETV) and the highest estimated transfer value (Highest ETV), obtained from `footballtransfers.com` and applied to players who meet the minimum playing time criteria.

4.1.1 Automated Data Collection Process (Web Scraping)

To carry out the requirement of estimating player value, the first and crucial step is to collect data on the transfer values of players currently playing in the 2024-2025 Premier League season. This data is sourced from the `footballtransfers.com` website and is filtered according to the minimum playing time criteria specified in the problem statement.

- **Libraries and Rationale for Selection:**
    1. `Selenium:` This is the foundation for automating interactions with web browsers (Chrome), including navigation, element searching, and data extraction from dynamic web pages.
    2. `Pandas:` The primary tool for reading input data (from CSV files), storing collected data as DataFrames, as well as performing filtering operations and exporting results to new CSV files.
    3. `re:` Used for parsing and standardizing text strings containing transfer values (ETV), converting them into numerical format.
    4. `os:` Provides functions for interacting with the file system, specifically for checking and creating directories to store results.
    5. `time:` Allows for creating pauses in the script, managing page load times, and avoiding sending requests too quickly.
    6. `traceback:` Helps in logging detailed information when errors occur, aiding in debugging.

7. `random`: Used to generate small random wait times, helping the scraping process to mimic more natural user behavior.

- **Execution Steps:**

1. First, the script processes the `results.csv` file to filter players who have played over 900 minutes, simultaneously standardizing their names and saving them into a list called `eligible_players_set` for later matching.
2. Next, Selenium is used to open the `footballtransfers.com` website and automatically navigate through all subpages containing player lists.
3. From each list page, basic player information such as Name, URL to the detail page, Team, Age, Position, and Current Estimated Transfer Value (Current ETV) in raw text format are collected.
4. After that, the script accesses the detail page of each player to retrieve the Highest Estimated Transfer Value (Highest ETV) in text format. Both types of ETV values are then processed by the `parse_etv` function to convert them into numerical form (in millions of Euros).
5. Finally, all processed data for each player (Name, Team, Age, Position, and the two ETV values in numerical form) are compiled into the `all_results` list in preparation for storage.

4.1.2 Final Data Preparation and Storage

After completing the web Browse process and collecting transfer value information, the data stored in the `all_results` list will be processed and saved into CSV files.

1. **Conversion to DataFrame:** The `all_results` list, containing dictionaries of each player's information, is converted into a Pandas DataFrame object. This operation structures the data in a tabular format, convenient for sorting, filtering, and storage.

2. **Column Selection and Ordering:** To ensure consistency and readability of the output data, the necessary columns are selected and arranged in a specific order: Player, Team, Age, Position, TransferValue_EUR_Millions, Highest_ETV_EUR_Millions. The script ensures that only these columns are retained in the final DataFrame.

3. **Checking and Creating Output Directory:** Before saving the file, the script checks if the destination directory (e.g., `Report/OUTPUT_BAI4/` from the `ALL_PLAYERS_FILE` and `OVER_900_FILE` variables) exists. If the directory does not exist, it will be created automatically using `os.makedirs()` to avoid errors during file writing.

4. **Saving Data to CSV Files:** After collection, the data of all players (with necessary columns selected) is saved to the file `all_players_scraped.csv` (variable `ALL_PLAYERS_FILE`). Subsequently, if the list of players with over 900 minutes (`eligible_players_set`) is available, this DataFrame will be filtered: player names are standardized to match with `eligible_players_set`, retaining only the eligible players. This final filtered result is saved to the file `players_over_900_filtered.csv` (variable `OVER_900_FILE`).

4.2 Data Processing

After the process of collecting raw transfer value data from the website `footballtransfers.com` was performed (`main_4.py`), this data needs to be processed and filtered to ensure accuracy and suitability for the problem requirements. The result of the initial collection process, containing information on all scraped players, is saved to the file `all_players_scraped.csv`. The main objective of this processing stage is to create a dataset that only includes players who have played over 900 minutes in the season and have transfer value information. This process is performed within the same `main_4.py` file, after the data scraping is completed.

4.2.1 Load Eligible Players List ($>$ 900 minutes)

To have a basis for filtering, it is first necessary to identify which players have met the playing time criteria.

1. **Read Data from Part I:** The function `load_eligible_players` is called to read the file `results.csv` (variable `RESULTS_CSV_PATH`). This file contains detailed performance statistics data for all players from fbref.com who have played over 90 minutes.

2. **Process Playing Time Column:** In the `results.csv` file, the 'Playing Time: minutes' column is processed to ensure it is in numerical format.

3. **Filter by 900 Minute Threshold:** The data is then filtered to keep only players with playing minutes greater than 900 (`min_minutes=900`).

4. **Standardize and Create Name Set:** The names of eligible players (from the 'Player' column) are extracted. To ensure accurate name matching later, the `normalize_name` function (defined in `main_4.py`) is applied. This function converts names to lowercase, removes diacritics and extra whitespace. The result is a set containing the standardized and eligible player names. Using a set optimizes the speed of checking in the next step. If the `results.csv` file is not found or there is an error, an empty set will be returned and an error message will be printed.

4.2.2 Filter Transfer Value Data

After having the set of eligible player names (`eligible_players_set`), the script proceeds to filter the DataFrame containing all player data scraped from footballtransfers.com (saved in the variable `df_all_to_save`, read from `ALL_PLAYERS_FILE`).

1. **Standardize Names in Scraped Data:** Player names in the 'Player' column of `df_all_to_save` are also standardized using the `normalize_name` function and saved to a temporary column.

2. **Perform Filtering:** The DataFrame `df_all_to_save` is filtered using Pandas' `.isin()` method. Only rows where the value in the 'normalized_player' column (standardized name from footballtransfers.com) is within `eligible_players_set` (standardized name from fbref.com and filtered >900 minutes) are kept.

3. **Remove Temporary Column:** The 'normalized_player' column is removed after the filtering is completed.

4. **Select Necessary Columns:** The filtered result (`df_filtered`) then only keeps the columns defined in `columns_to_keep` (including "Player", "Team", "Age", "Position", "TransferValue_EUR_Millions", "Highest_ETV_EUR_Millions").

4.2.3 Save Processed Results

The DataFrame containing transfer value information for players accurately filtered according to the requirement (> 900 playing minutes) is saved to the file Report/OUTPUT_BAI4/players_over_900_filtered.csv (variable `OVER_900_FILE` in the code). This file is an important output of `main_4.py` and will be used as one of the main data sources for `main_4.2.py` (the model training module).

	A	B	C	D	E	F
1	Player	Team	Age	Position	TransferValue_EUR_Millions	Highest_ETV_EUR_Millions
2	Erling Haaland	Man City	24	FW	198.8	199.6
3	Martin Ødegaard	Arsenal	26	MF	126.5	134.5
4	Alexander Isak	Newcastle Utd.	25	FW	120.3	120.3
5	Cole Palmer	Chelsea	23	MF	115.4	119
6	Declan Rice	Arsenal	26	MF	107.8	120
7	Alexis Mac Allister	Liverpool	26	MF	106.1	117
8	Phil Foden	Man City	24	MFFW	105.7	138.6
9	Bukayo Saka	Arsenal	23	FWMF	101.3	127.5
10	Ryan Gravenberch	Liverpool	22	MF	85.3	85.5
11	Bruno Guimarães	Newcastle Utd.	27	MF	83.2	83.2
12	Moisés Caicedo	Chelsea	23	MFDF	80.7	86.1
13	William Saliba	Arsenal	24	DF	79.5	79.5
14	Omar Marmoush	Man City	26	FWMF	79.1	79.1
15	Joško Gvardiol	Man City	23	DF	78.7	86.6

Figure 4.1: Sample result segment from the players_over_900_filtered.csv file

4.3 Proposed Method for Estimating Player Value main_4_2.py

After collecting and processing the transfer value data for players in section 4.2, the next step is to propose and implement a machine learning method to estimate their value. The goal is to build a model capable of predicting the TransferValue_EUR_Millions variable based on player performance statistics and personal information. This entire process is performed in the file `main_4.2.py`.

4.3.1 Overall Process Introduction

The player value estimation process I performed includes the following main steps:

1. **Load and Combine Data:** Load player statistics data from `results.csv` and filtered transfer value data from section 4.2. These two datasets are merged based on player name.

2. **Data Preprocessing and Feature Engineering:**

- Process Target Variable: Apply a log transformation (`np.log1p`) to the transfer value column to reduce skewness and handle outliers using Winsorizing technique.
- Remove identifier columns not needed for training.
- Clean and convert data types for columns.
- Create new features (advanced features) from existing features to enhance the data's representational ability. These features include:
  - Binning for continuous variables like Age and 'Playing Time: minutes' using `KBinsDiscretizer`.
  - Create interaction features between variables (e.g., Age_x_Minutes, Age_Sq, interactions between age/metrics and playing position Is_FW, Is_MF).
  - Create ratio features (e.g., Gls_div_xG).
- Handle missing values (NaN) and outliers for features.
- Prepare Data for Model: Split data into training (train) and testing (test) sets. Identify numerical and categorical columns.

3. **Build Preprocessing Pipeline:** Use `ColumnTransformer` to apply separate processing steps for each type of feature:
   - Numerical Features: `KNNImputer` (impute missing values), `PolynomialFeatures` (create polynomial and second-order interaction features), and `StandardScaler` (standardize).
   - Categorical Features: `SimpleImputer` (impute missing values) and `OneHotEncoder` (convert to numerical form).
4. **Feature Selection using RFE:** The code includes an option to use Recursive Feature Elimination (RFE) with a base `GradientBoostingRegressor` model to select a subset of the most important features before training the final model.
5. **Train Model:** Use `GradientBoostingRegressor` from `sklearn.ensemble`. The best hyperparameters for the model are searched using `RandomizedSearchCV` over a broad parameter space, optimizing based on the R^2 score.
6. **Evaluate Model:** Predict on the test set and evaluate performance using R^2 (on both log scale and original scale), Root Mean Squared Error (RMSE), and Mean Absolute Error (MAE) metrics.
7. **Visualize Results:** Plot charts including Feature Importance, Predicted vs. Actual Values, and Residuals Plot.
8. **Storage:** Save the trained model and the preprocessor object for potential reuse.

4.3.2 Feature Selection

The selection and creation of features significantly impacts the model's performance.

1. **Input Data:**
   - Statistics data from `results.csv` (Part I).

- Basic information and transfer values from `players_over_900_filtered.csv` (Part IV, section 4.3).
- The Highest_ETV_EUR_Millions column is considered for creating new features like Value_vs_HighestETV_Ratio to reflect current value relative to the player's peak value.

2. **Processing and Feature Creation Process in `prepare_features_target` and `create_more_advanced_features`:**
   - Identifier columns (Player, Nation, Team, Team_TransferSite) are removed from the training feature set (X).
   - Object type columns are attempted to be converted to numerical type after cleaning.
   - New features are created as described in section 4.4.1 (Binning, Interactions, Ratios) to enrich the information for the model.
   - Outliers in numerical features are handled using the Winsorizing method.
   - Columns with a high percentage of missing values (e.g., >30-40%) will be removed to ensure data quality.
3. **Feature Selection using RFE (Optional):** If enabled (`use_rfe = True`), RFE will be used to automatically select a number of features (`n_features_to_select_rfe`, e.g., 40 or 80) based on the base Gradient Boosting model. This helps reduce dimensionality, remove noise, and can improve the generalization ability of the final model. The list of features selected by RFE will be used to train the main model.

4.3.3 Model Selection

I decided to use the GradientBoostingRegressor (`sklearn.ensemble.GradientBoostingRegressor`) model for the player value estimation problem. This choice is based on the following advantages:

- **High Performance:** Gradient Boosting is often among the algorithms that yield the best prediction results for regression problems with tabular data.
- **Ability to Handle Non-linear Relationships:** By sequentially building decision trees, this model can capture complex relationships and interactions between features that linear models might miss.
- **Robust with Different Data Types:** Can work well with both numerical and categorical features (after appropriate encoding).
- **Regularization Capability:** Provides several hyperparameters (like subsample, max_features, min_samples_leaf, learning_rate) to help control overfitting.
- **Flexible Loss Function:** Different loss functions (e.g., 'squared_error', 'huber') can be chosen to suit the data characteristics and reduce the impact of outliers.

The search for optimal hyperparameters for GradientBoostingRegressor is performed using `RandomizedSearchCV`. This technique explores a predefined hyperparameter space (`param_dist`), performs cross-validation (`cv=5`), and selects the set of parameters that yield the best result based on the `scoring='r2'` metric.

4.3.4 Overall Training Process and Preprocessing in Pipeline

The process from having data to training the model is encapsulated and automated:

- **Load and Combine Data:** As described.
- **Prepare X and y:** Separate features and target variable, apply log transform to y.
- **Split Train/Test Set:** The data is split with an 80% ratio for training and 20% for testing (`test_size=0.2`).
- **Build Preprocessor:** A `ColumnTransformer` object is created to apply separate processing pipelines for each group of columns:
  - `numeric_pipeline`: Includes `KNNImputer` (impute missing values), `PolynomialFeatures (degree=2, interaction_only=True)` (create second-order interaction features between numerical variables), and `StandardScaler` (z-score standardization).
  - `categorical_pipeline`: Includes `SimpleImputer` (impute missing values using a constant 'Missing' or the most frequent value) and `OneHotEncoder` (convert categorical variables to binary columns, with `handle_unknown='infrequent_if_exist'` to handle new/rare values and `min_frequency` to group infrequently occurring categories).

  This Preprocessor is `fit_transform` on `X_train` and `transform` on `X_test`.
- **Training:**
  - The `X_train_processed` data (after going through the preprocessor) is fed into `train_gbr_model`.
  - If RFE is enabled, `X_train_processed` will be feature selected by RFE first.
  - `RandomizedSearchCV` is performed to find the best hyperparameters and train the final GradientBoostingRegressor model.

4.3.5 Results and Evaluation

After performing the training and evaluation process, I obtained the following results on the test set:

- R^2: 0.894
- RMSE: 8.15
- MAE: 5.61

These metrics indicate that the model is capable of explaining approximately 89.4% of the variance in player transfer value on the original scale. The RMSE value indicates that the average prediction error is about 8.15 million EUR.

Feature Importance Plot:

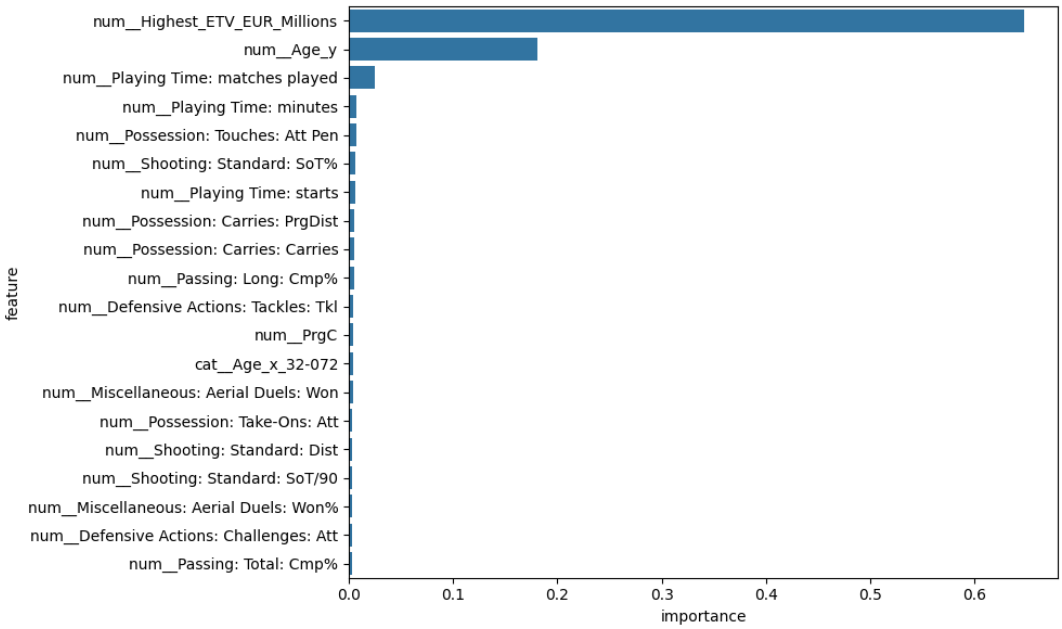

Figure 4.2: Feature Importance.

Plot Analysis: The Feature Importance Plot (Figure 4.1), shows the factors that have the greatest influence on the Gradient Boosting model's player value predictions.

- **num_Highest_ETV_EUR_Millions** (Highest historical transfer value): Most important, reflecting the player's recognized class and potential.
- **num_Age_y** (Age): Very important, affecting development potential and career stage, thereby influencing value.
- **num_Playing Time: matches played** (Number of matches played): Shows experience and contribution level, influencing value perception.

Predicted vs. Actual Value Plot:

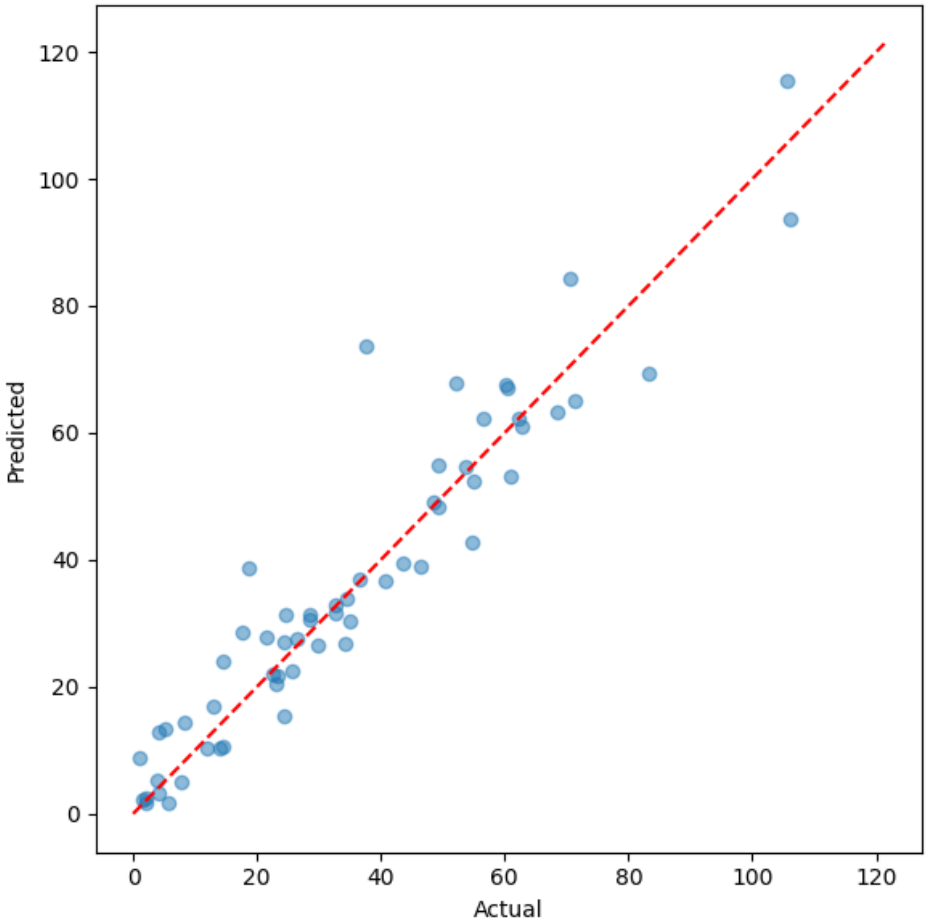

Figure 4.3: Predicted vs. Actual Value.

Plot Analysis: Figure 4.2 shows the relationship between the player's actual transfer value (x-axis) and the value predicted by the model (y-axis).

- General Trend: Data points tend to scatter around the y=x diagonal line. This indicates that the model captures most of the relationship between features and transfer value, with predictions having a positive correlation with actual values.
- Good Prediction Range: The model shows the best prediction ability for players with actual values in the range from 0 to about 80 million Euro. Within this range, data points are closer to the y=x diagonal line, with low deviation.
- Prediction at Low Values: For some players with very low actual values (close to 0), the model appears to predict slightly higher values than actual (overestimation), shown by a few points above the diagonal line.
- Prediction at High Values: For players with high actual values (e.g., above 80-100 million Euro), the model tends to predict lower values than actual (underestimation). Data points in this region are typically below the y=x diagonal line.

Residuals Plot:

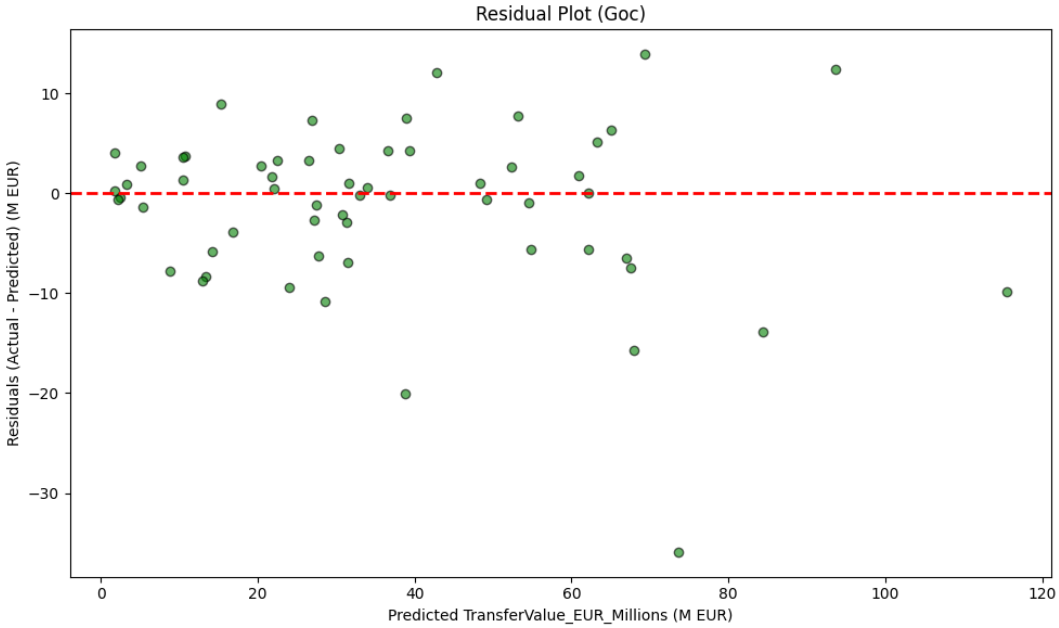

Figure 4.4: Residuals Plot

Plot Analysis: The Residuals Plot (Figure 4.4) shows the error between the actual value and the predicted value for a data point.

- Distribution around 0: Most residual points are distributed quite randomly around the 0 line, indicating that the model does not have a clear systematic bias.
- Pattern (Heteroscedasticity): There is no clear funnel shape, but the variance of residuals seems to increase slightly at higher predicted values (e.g., from 60 million EUR upwards), implying that the uncertainty of prediction may be larger for high-value players.
- Outliers: There are still a few large residual points, indicating specific cases where the model's prediction was not good.

4.3.6 Overall Evaluation of the Proposed Method

The proposed method for estimating player value, including steps from data collection, combining statistical data, thorough preprocessing, creating advanced features (including binning, interactions, ratios, and polynomial features), optional feature selection using RFE, and using the Gradient Boosting Regressor model with hyperparameter optimization using RandomizedSearchCV, has shown promising results.

Strengths of the Method:

- The data preprocessing pipeline is designed to handle common issues like missing values (using KNNImputer, SimpleImputer), standardization (StandardScaler), and categorical variable encoding (OneHotEncoder).
- Feature Engineering with PolynomialFeatures and manual interactions helps the model capture more complex relationships in the data.
- Gradient Boosting is a powerful algorithm, and using RandomizedSearchCV helps find a good set of hyperparameters, optimized for the R^2 metric.
- Log transforming the target variable and handling outliers helps improve model stability and performance.

Results: With an R^2 score on the original scale of 0.894 - this is a good result, indicating that the model has demonstrated the ability to explain a significant portion of the variance in player value. The visualizations also provide insights into performance and factors influencing predictions.

Bibliography

[1] GeeksforGeeks. (2025, March 11). *ML / Gradient Boosting*.

[2] Rawat, A. (2025, February 16). *Clustering and dimensionality reduction techniques to simplify complex data*. Interview Kickstart.

[3] Scikit-learn Development Team. (2025). *Scikit-learn: Machine Learning in Python*

[4] Scikit-learn Development Team. (2025). *sklearn.impute.KNNImputer*. Scikit-learn User Guide.

[5] Selenium Committers. (2025). *Selenium WebDriver Documentation*.