POSTS AND TELECOMMUNICATIONS INSTITUTE OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY I

o0o

ASSIGNMENT REPORT 1

PYTHON PROGRAMMING LANGUAGE

Instructor:	Kim Ngoc Bach
Student:	Chu Tuyet Nhi
Student ID:	B23DCCE075
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:	Chu Tuyet Nhi
Student ID:	B23DCCE075
Class:	D23CQCEO6-B
Academic Year:	2023 - 2028
Training System:	Full-time University

Hanoi, 2025

LECTURER'S COMMENTS

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

Score: (**In words:**)

Hanoi, date month year 20...

Lecturer

Contents

List of Figures

Introduction

This report is prepared to fulfill the requirements of Major Assignment 1 for the Python programming course. The main objective of the assignment is to apply data collection, analysis, and modeling techniques to analyze information about English Premier League football players for the 2024-2025 season. Statistical data on the performance of players who have played over 90 minutes were collected from the website fbref.com. Data on the estimated transfer values of players who have played over 900 minutes were obtained from footballtransfers.com. This entire data collection process was carried out on May 2, 2025. Therefore, all analyses and results presented in this report reflect the situation and statistics of the players up to that specific point in the season. The report content is divided into four main parts:

Chapter 1. Collecting Player Data from fbref.com: Details the process of collecting player statistical data from fbref.com using automated tools like Selenium and BeautifulSoup, along with steps for cleaning and storing the initial data. Follow the requirements of Chapter 1 of the assignment.

Chapter 2. Data Analysis and Visualization: Performs descriptive statistical analyses, including identifying the top 3 highest/lowest players for each metric, calculating mean, median, standard deviation, visualizing data distribution through histograms, and providing a preliminary assessment of team performance.

Chapter 3. Player Clustering using K-Means and PCA: Applies the K-Means clustering algorithm to group players into clusters with similar statistical characteristics, thereby understanding different player archetypes in the league.

Chapter 4. Estimating Player Value. This part focuses on collecting transfer value data of players from the footballtransfers.com website and proposing a method to build a machine learning model (using Gradient Boosting Regressor) to estimate their market value. This process includes selecting features from performance statistics (collected in Chapter 1), basic player information, and historical transfer values, then training and evaluating the model to predict player values.

Chapter 1

Collecting Player Data from fbref.com

This section presents the method and process of collecting statistical data for English Premier League players for the 2024-2025 season from the website fbref.com, fulfilling the requirements of Assignment I. The data to be collected includes detailed parameters as requested in Chapter 1 for players with more than 90 minutes of playing time. The results are saved in the file results.csv and sorted by player name.

1.1 Program Structure

To collect player statistical data from the fbref.com website as required by the assignment, I have developed a scraping program organized in a modular fashion, including the following main components:

`MAIN_part1.py`: The main module, responsible for coordinating the entire data collection process.

`config_part1.py`: Stores important configurations such as website URLs, mapping of data to be retrieved, as well as technical parameters like wait times.

`scraper.py`: Handles website access and extraction of raw data from HTML pages.

`processor.py`: Processes the extracted data, converting it into a structured DataFrame for convenient subsequent analysis.

Designing with a modular approach helps make the code clearly structured and easy to manage. It helps me easily detect and fix errors during program execution because the responsibilities of each part are clearly separated.

1.2 Data Collection with Selenium and BeautifulSoup

1.2.1 Choosing Data Collection Tools

I chose to combine the `Selenium` and `BeautifulSoup` libraries because the fbref.com website uses JavaScript to load data dynamically, which the `requests` library cannot handle directly. Besides, `Selenium` allows for more natural simulation of user behavior, thereby helping to minimize the risk of being blocked by the website's anti-scraping mechanisms [1, 2].

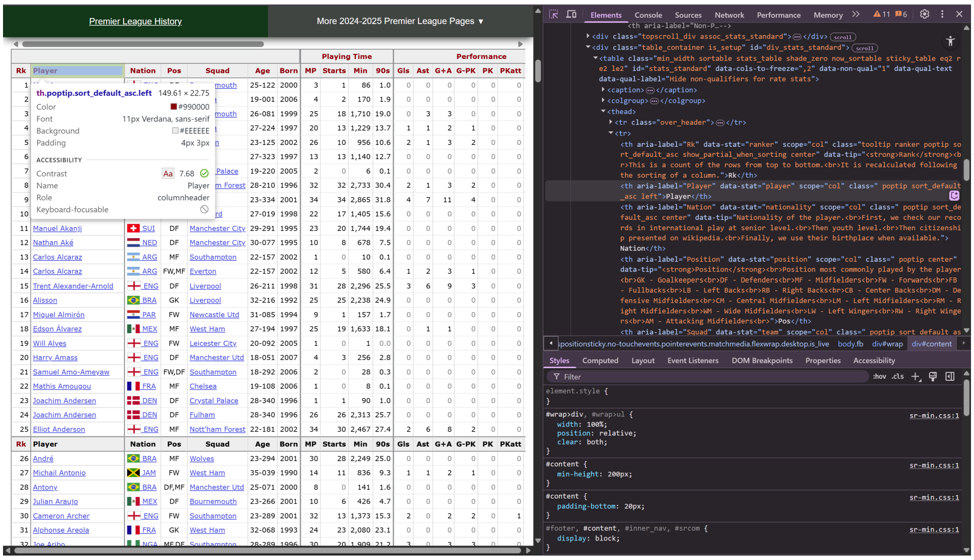

Figure 1.1: Screenshot of Developer Tools highlighting a data cell in the statistics table on fbref.com and the corresponding HTML tag.

In addition, the browser's Developer Tools were also used to identify the HTML structure of the statistics tables and determine the `data-stat` attribute corresponding to each required metric.

1.2.2 Implementation Details

The main function `scrape_fbref_data` in the `scraper.py` module operates in the following steps, with the code snippet shortened to focus on the main logic:

```python
def scrape_fbref_data(driver: WebDriver, url_config: dict, stats_map: dict) -> dict:
    player_data = {}
    for category, (url, table_id) in url_config.items():
            # 1. Access the URL of each type of statistic
            driver.get(url)
            # 2. Wait for the data table to load completely
            wait = WebDriverWait(driver, WAIT_TIME)
            wait.until(EC.presence_of_element_located((By.CSS_SELECTOR,
                                            f"{table_id} tbody tr")))
            # 3. Parse HTML with BeautifulSoup
            soup = BeautifulSoup(driver.page_source, 'html.parser')
            rows = soup.select(f"{table_id} tbody tr")
            # 4. Process each player data row
            for row in rows:
                if 'thead' in row.get('class', []): continue
                player_name = safe_get_text(row, 'player')
                team_name = safe_get_text(row, 'team')
                if not player_name or team_name == 'N/a': continue
```

```
                # 5. Create a unique key for each player as a pair (name, team)
                key = (player_name, team_name)
                data = player_data.setdefault(key, {'Player': player_name,
                                                     'Team': team_name})
                # 6. Collect all statistics according to the configured mapping
                for k, stat in stats_map.items():
                    if k not in ['Player', 'Team']:
                        val = safe_get_text(row, stat)
                        if val != 'N/a' or k not in data:
                            data[k] = val
    return player_data
```

The scrape_fbref_data function is designed to automatically collect and extract player statistical data from various pages on the FBref site. The function takes a Selenium driver, a configuration of URLs and table IDs (url_config), and a column name mapping (stats_map) as input. The process includes: accessing each URL, waiting for the data table to load, parsing the HTML with BeautifulSoup, then extracting and consolidating each player's data into a dictionary structure with the key being a pair of (player name, team name). The function ensures that only valid data is collected and returns the result as a dictionary, convenient for later processing and storage. Due to transfers occurring during the season, some players may play for multiple different clubs. To ensure the accuracy and integrity of the data, I chose to separate the data according to each team the player played for. Specifically, during the data collection process from fbref.com via the code in the scraper.py file, each player is uniquely identified by the pair of information (Player Name, Team Name). This approach offers several benefits as follows:

- **Accuracy:** Player performance statistics are recorded separately for each club, accurately reflecting their contribution during specific periods.
- **Integrity:** No data is missed when a player changes teams during the season. Each period of participation with a team is recorded separately.
- **Compatibility with Source Data:** fbref.com also presents data in this manner for players who play for multiple clubs in the same season.

1.3 Configuration and Data Mapping

To easily change parameters such as URLs or field names to be retrieved, I have put them all in the config_part1.py file instead of hardcoding them. I use two main map structures to manage this information:

URL_CONFIG: Maps between statistic type and corresponding URL:

```
URL_CONFIG = {
    'standard': ('https://fbref.com/en/comps/9/stats/Premier-League-Stats', '#stats_standard'),
```

```python
    'keeper': ('https://fbref.com/en/comps/9/keepers/Premier-League-Stats', '#stats_keeper'),
    'shooting': ('https://fbref.com/en/comps/9/shooting/Premier-League-Stats', '#stats_shooting'),
    # ... other statistic types
}
```

`STATS_MAP`: Maps between CSV column names and the `data-stat` attribute in HTML:

```python
STATS_MAP = {
    'Player': 'player',
    'Nation': 'nationality',
    'Team': 'team',
    # ... other statistical indicators
}
```

1.4 Data Processing and Cleaning

The `processor.py` module is responsible for converting raw data into a DataFrame with a structure suitable for the requirements, with the code snippet shortened to focus on the main logic and the code execution steps:

```python
def process_data(raw_data: dict, final_columns: list) -> pd.DataFrame:
   # 1. Convert data from dict to DataFrame
   df = pd.DataFrame.from_dict(raw_data, orient='index')
   # 2. Filter players who played more than 90 minutes
   minutes_column_name = 'Playing Time: minutes' # Ensure the column name is correct
   if minutes_column_name in df.columns:
        df['Min_numeric'] = pd.to_numeric(
            df[minutes_column_name].astype(str).str.replace(',', '', regex=False), errors='coerce')
        df = df[df['Min_numeric'] > 90].copy()
        df.drop(columns=['Min_numeric'], inplace=True)
   # 3. Select columns in the specified order
   # Ensure final_columns contains columns that exist in df after filtering
   existing_columns = [col for col in final_columns if col in df.columns]
   df = df[existing_columns]
   # 4. Fill 'N/a' for missing data
   df.fillna('N/a', inplace=True)
   # 5. Sort by player name
   if 'Player' in df.columns:
```

```
        df.sort_values(
            by='Player',
            key=lambda x: x.str.split().str[0].str.lower(),
            inplace=True
        )
    return df
```

The main processing steps include:

- **Filtering player data:** Only keep players with a total playing time greater than 90 minutes. Due to time data on the website being formatted with commas as separators for units (e.g., "1,234" minutes), a processing step is required to remove the commas before converting to a numeric data type.
- **Handling missing data:** Fill 'N/a' for all NaN values as required by the assignment.
- **Sorting data:** Sort players by name (specifically the first name, not the last name).

1.5 Main Execution Function

The main module `MAIN_part1.py` manages the complete execution process, the code snippet has been shortened to focus on the main logic:

```
def run_scraper():
    # Step 1: Initialize WebDriver
    driver = webdriver.Chrome(options=Options().add_argument('--log-level=3'))

    # Step 2: Collect and process data
    raw_data = scrape_fbref_data(driver, URL_CONFIG, STATS_MAP) # from scraper.py
    # Pass data to the processing function
    df = process_data(raw_data, list(STATS_MAP)) # from processor.py

    # Step 3: Save results to a csv file
    df.to_csv(OUTPUT_FILENAME, index=False, encoding='utf-8-sig')

    # Close WebDriver
    driver.quit()
```

The code above illustrates the overall operation of the `run_scraper` function, including steps: initializing the WebDriver, collecting and processing data, saving the output as a CSV file with `utf-8-sig` encoding to ensure data integrity, and finally closing the WebDriver.

1.6 Data Collection Results

The final results are saved in the file results.csv, and specifically:

- **Number of players:** 491 players (played over 90 minutes).
- **Number of statistics per player:** indicators as required by the assignment.
- **Data format:** Standard CSV with column headers.
- **Sorting:** Players are sorted by first name.
- **Missing data:** Marked as "N/a" as required by the assignment.

Below is a sample snippet from the results.csv file opened in Excel:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	
1	Player	Nation	Team	Position	Age	Playing Time: matches played	Playing Time: starts	Playing Time: minutes	Performance: goals	Performance: assists	Performance: yellow cards	Performance: red cards	Expected: xG	Expected: xAG	Progression: PrgC	Progression: PrgP	Progression: PrgR	Per 90 minutes: Gls	Per 90 minutes: Ast	Per 90 minutes: xG	Per 90 minutes: xAG	Go Pe
2	Aaron Wan-Bissaka	ENG	West Ham	DF	27-157	32	31	2,794	2	2	1	0	1.2	2.9	98	125	139	0.06	0.06	0.04	0.09	N/a
3	Aaron Cresswell	ENG	West Ham	DF	35-138	14	7	589	0	0	2	0	0.1	1.1	4	24	2	0	0	0.02	0.17	N/a
4	Aaron Ramsdale	ENG	Southampton	GK	26-353	26	26	2,340	0	0	2	0	0	0	0	0	0	0	0	0	0	
5	Abdoulaye Doucouré	MLI	Everton	MF	32-121	30	29	2,425	3	1	5	1	3.9	2.3	40	78	91	0.11	0.04	0.14	0.09	N/a
6	Abdukodir Khusanov	UZB	Manchester City	DF	21-062	6	6	503	0	0	1	0	0	0.1	1	25	2	0	0	0	0.02	N/a
7	Abdul Fatawu Issahaku	GHA	Leicester City	FW	21-055	11	6	579	0	2	0	0	0.4	1.6	42	17	60	0	0.31	0.06	0.24	N/a
8	Adam Smith	ENG	Bournemouth	DF	34-003	22	17	1,409	0	0	6	0	0.7	0.3	12	40	31	0	0	0.04	0.02	N/a
9	Adam Lallana	ENG	Southampton	MF	36-357	14	5	361	0	2	4	0	0.2	0.9	6	24	10	0	0.5	0.04	0.23	N/a
10	Adam Armstrong	ENG	Southampton	FW,MF	28-081	20	15	1,248	2	2	4	0	3.3	1.2	25	21	79	0.14	0.14	0.24	0.09	N/a
11	Adam Webster	ENG	Brighton	DF	30-118	11	8	617	0	0	0	0	0	0.5	7	40	2	0	0	0	0.07	N/a
12	Adam Wharton	ENG	Crystal Palace	MF	20-334	19	15	1,258	0	2	2	0	0.3	3	14	105	10	0	0.14	0.02	0.21	N/a
13	Adama Traoré	ESP	Fulham	FW,MF	29-097	32	16	1,568	2	6	2	0	3.8	4.7	87	61	143	0.11	0.34	0.22	0.27	N/a
14	Albert Grønbaek	DEN	Southampton	FW,MF	23-344	4	2	143	0	0	0	0	0.1	0	1	1	3	0	0	0.07	0.01	N/a
15	Alejandro Garnacho	ARG	Manchester Utd	MF,FW	20-305	33	22	2,056	5	1	2	0	7	3.6	134	54	274	0.22	0.04	0.31	0.16	N/a
16	Alex Palmer	ENG	Ipswich Town	GK	28-265	10	10	900	0	0	2	0	0	0	0	1	0	0	0	0	0	
17	Alex Iwobi	NGA	Fulham	FW,MF	28-364	34	32	2,721	9	6	1	0	4.5	6.5	132	196	221	0.3	0.2	0.15	0.22	N/a

results

Ready Accessibility: Unavailable 100%

Figure 1.2: Sample snippet from the results.csv file.

Chapter 2

Data Analysis and Visualization

This section focuses on a deeper analysis of the player dataset collected in Chapter 1, fulfilling the requirements of Assignment II. The objective is to analyze the prominent features of the data through descriptive statistics, identify players and teams with high/low performance, and visualize the distribution of important indicators through charts. At the same time, analyze to find the team with the highest performance as required by the assignment.

2.1 Program Structure Chapter 2

Similar to Chapter 1, the program for Chapter 2 is also organized in a modular fashion to ensure clarity and maintainability:

`MAIN_part2.py`: Main coordinating module, executes analysis steps sequentially and calls functions from other modules.

`config_part2.py`: Contains necessary configurations for Chapter 2, including the path to the input data file (results.csv from Chapter 1), output directories, a list of statistical indicators to analyze (attacking, defensive, 3 selected defensive and 3 attacking indicators, negative indicators), and columns to exclude from some analyses.

`analysis.py`: Module containing functions that perform core analyses such as finding top/bottom players, calculating summary statistics (median, mean, standard deviation) by team and overall, and finding the best performing team for each indicator.

`plotting.py`: Module responsible for creating data visualization charts, specifically histogram distribution charts of selected indicators.

2.2 Data Preparation and Preprocessing

The first step is to load data from the `results.csv` file created in Chapter 1.

```python
# Snippet from MAIN_part2.py - load_data function
def load_data():
    df = pd.read_csv(INPUT_CSV)  # read CSV file
```

```python
    numeric_cols = [col for col in df.columns if col not in EXCLUDED_COLUMNS_FROM_TOP3]  # select numeric columns
    for col in numeric_cols:
        df[col] = pd.to_numeric(df[col].astype(str).str.replace(',', '', regex=False), errors='coerce')  # convert to numeric
    return df, numeric_cols # return dataframe and list of numeric columns
```

During this process, columns containing statistical data (excluding identifier columns such as Name, Nationality, Team, Position, Age) are converted to numeric format to enable calculations. The `pd.to_numeric` function is used with the parameter `errors='coerce'` to automatically convert invalid values (e.g., 'N/a') to NaN, while also removing thousands separators (if any).

2.3 Identifying Top 3 and Bottom 3 Players by Each Indicator

To identify outstanding players, both highest and lowest for each statistical indicator, the `get_top_bottom_players` function in `analysis.py` is used. This function sorts the DataFrame based on the specified indicator column and returns the top N players and the bottom N players (N=3 as required).

```python
# Snippet from analysis.py - get_top_bottom_players function
def get_top_bottom_players(df, stat_col, n=3):
    # 1. get necessary columns
    df_valid = df[['Player', 'Team', stat_col]].copy()
    # 2. convert to numeric
    df_valid[stat_col] = pd.to_numeric(df_valid[stat_col], errors='coerce')
    df_valid = df_valid.dropna(subset=[stat_col])  # 3. drop NaN values
    # 4. sort descending
    df_sorted = df_valid.sort_values(stat_col, ascending=False)
     # 5. top n and bottom n
    return df_sorted.head(n), df_sorted.tail(n).sort_values(stat_col)
```

In this section, 'N/a' values are removed instead of being replaced with other values like 0 for the following reasons:

- 'N/a' is not synonymous with the value 0. Replacing it with 0 causes misinterpretation, as it implies the player played but did not score, when in reality the data might be missing. This distorts the nature of the data and affects the accuracy of rankings.
- Ensuring fairness in ranking: To determine the Top/Bottom 3 fairly and accurately, only players with complete and valid data for that indicator should be compared.

Achieved Results: Below is an example of the top/bottom 3 players for 2 typical indicators, extracted from the file `top_3.txt`:

General Comments: The data shows a large difference in performance and playing time among players. Some players stand out in multiple categories, while others have more limited contributions or are specialized for their roles.

Table 2.1: Playing Time: matches played

Player	**Team**	**Matches**
	Top 3	
Youri Tielemans	Aston Villa	34
Virgil van Dijk	Liverpool	34
Bruno Guimarães	Newcastle Utd	34
	Bottom 3	
Jahmai Simpson-Pusey	Manchester City	2
Ayden Heaven	Manchester Utd	2
Hákon Rafn Valdimarsson	Brentford	2

Comments: The three leading players all have a maximum of 34 appearances, indicating their important role and stability in the squad. Conversely, the group at the bottom has very few opportunities to play.

Table 2.2: Performance: goals

Player	**Team**	**Goals**
	Top 3	
Mohamed Salah	Liverpool	28
Alexander Isak	Newcastle Utd	22
Erling Haaland	Manchester City	21
	Bottom 3	
Chiedozie Ogbene	Ipswich Town	0
Cheick Doucouré	Crystal Palace	0
Adam Webster	Brighton	0

Comments: Mohamed Salah of Liverpool leads the scoring chart with 28 goals, creating a significant gap with the players ranked below him. Many players, especially those with defensive tendencies or who get less playing time, have not scored any goals.

2.4 Calculating Summary Statistics

The next step is to calculate basic descriptive statistics (median, mean, standard deviation) for each indicator. These values are calculated for the entire league ("all") and for each individual team. The `calculate_stats_summary` function in `analysis.py` handles this task.

```python
# Snippet from analysis.py - calculate_stats_summary function
def calculate_stats_summary(df, stats_cols):
    # ... (Validation and preparation of numeric data columns) ...
    team_stats = pd.DataFrame()
    if 'Team' in df_copy.columns and not df_copy['Team'].isnull().all():
```

```python
        # Group by team and calculate median, mean, std for numeric columns
        grouped = df_copy.groupby('Team')
        team_stats_agg = grouped[valid_cols].agg(['median', 'mean', 'std'])
        # ... (Processing column names after aggregation) ...
        team_stats = team_stats_agg.reset_index()
        # ... (Rename columns for clarity) ...
    # ... (Handling cases where Team column is missing) ...
    # Calculate overall statistics for all players ("all")
    overall_stats_data = {'Team': 'all'}
    for col in valid_cols:
        overall_stats_data[f'Median of {col}'] = df_copy[col].median()
        overall_stats_data[f'Mean of {col}'] = df_copy[col].mean()
        overall_stats_data[f'Std of {col}'] = df_copy[col].std()
    overall_df = pd.DataFrame([overall_stats_data])
    # Combine overall results and results by team
    summary = pd.concat([overall_df_ordered, team_stats], ignore_index=True)
    return summary
```

Achieved Results: After processing and analyzing data for 20 teams, along with an "All" team representing the aggregate of all teams, the obtained results have fairly comprehensively and visually reflected the key indicators related to each team's performance. Aggregating into an "All" team helps create an overall perspective, serving as a basis for comparison with the individual performance of each team. Below is a sample snippet from the results2.csv file opened in Excel:

Team	Median of Playing Time: matches played	Mean of Playing Time: matches played	Std of Playing Time: matches played	Median of Playing Time: starts	Mean of Playing Time: starts	Std of Playing Time: starts	Median of Playing Time: minutes	Mean of Playing Time: minutes	Std of Playing Time: minutes	Median of Performance: goals	Mean of Performance: goals	Std of Performance: goals	Median of Performance: assi
all	22	20.731	9.68	14	15.21	10.687	1335	1363.786	903.477	1	1.99	3.493	
Arsenal	22.5	22.591	7.926	16	17	10.156	1427.5	1519.455	881.677	2	2.773	2.81	
Aston Villa	20	18.857	9.966	9.5	13.357	11.324	969	1200	913.117	1	1.857	3.285	
Bournemouth	25	21.609	9.838	17	16.217	11.33	1580	1451.739	975.756	1	2.261	3.493	
Brentford	27	22.857	11.146	21	17.81	12.956	1913	1592.333	1092.856	0	2.762	5.291	
Brighton	20	18.964	9.758	9	13.357	9.867	895.5	1198.464	866.121	1	1.929	2.956	
Chelsea	17.5	19.154	10.88	11.5	14.385	11.399	1046.5	1291.423	995.128	1	2.192	3.476	
Crystal Palace	29	23.381	10.21	18	17.714	12.471	1562	1585.905	1047.049	0	1.857	3.366	
Everton	23.5	21.727	9.166	14.5	16.955	10.558	1281	1520.273	893.057	1	1.409	1.968	
Fulham	26	23.773	9.211	17	16.955	11.18	1596.5	1523.273	935.531	0.5	2.227	3.265	
Ipswich Town	18	17.667	8.743	11	12.467	9.489	952.5	1113.767	781.744	0	1.067	2.288	
Leicester City	21	19.731	9.569	14.5	14.385	9.811	1355	1292.231	811.218	0	1.038	1.8	
Liverpool	28	24.476	8.903	19	17.81	11.994	1627	1596.381	981.765	1	3.762	6.503	
Manchester City	22	19.24	8.762	16	14.92	8.406	1404	1341.76	744.172	1	2.6	4.406	
Manchester Utd	20	18.778	10.966	14	13.778	10.696	1335	1231.667	920.107	0	1.37	2.204	
Newcastle Utd	27	22.565	9.917	13	16.261	12.259	1413	1459.826	1021.144	0	2.739	5.011	
Nott'ham Forest	29.5	23.864	10.575	18.5	17	12.903	1764	1527.682	1071.82	1	2.364	4.17	
Southampton	20	18.138	10.056	13	12.862	9.512	1122	1151.345	828.595	0	0.828	1.071	
Tottenham	21	18.889	9.279	15	13.815	8.119	1252	1241.111	696.541	0	2.185	3.27	
West Ham	20	20.92	8.281	14	14.96	10.522	1070	1341.92	885.125	0	1.44	2.468	
Wolves	25	22.087	8.681	15	16.174	10.547	1364	1452.174	848.656	1	2.174	3.939	

Figure 2.1: Sample snippet from the results2.csv file.

2.5 Visualizing Data Distribution

To better understand the distribution of statistical indicators, histogram charts are used as required by the assignment. The `plotting.py` module provides functions to draw:

Overall histogram (`plot_histogram_all_players`): Shows the distribution of an indicator across all players in the league. This chart is accompanied by a Kernel Density Estimate (KDE) line to clarify the distribution trend.

```python
def plot_histogram_all_players(df, stat_col, bins=20, fmt='png', xlim=None, ylim=None):
# Create figure and axes for plotting
fig, ax = plt.subplots(figsize=(10, 6))
# Plot histogram with KDE line for all players
sns.histplot(df[stat_col].dropna(), bins=bins, kde=True, ax=ax)
# Set title and axis labels
ax.set(title=f'Distribution of {stat_col} (All Players)', xlabel=stat_col, ylabel='Frequency')
# Limit axes if set
if xlim: ax.set_xlim(xlim)
if ylim: ax.set_ylim(ylim)
# Adjust layout and save figure
fig.tight_layout()
_save_plot(fig, f'hist_all_{_safe_filename(stat_col)}', fmt)
```

Histogram per team (`plot_histograms_per_team_facet`): Displays multiple small histogram charts, each corresponding to a team, helping to visually compare differences in indicator distribution among teams.

```python
def plot_histograms_per_team_facet(df, stat_col, col_wrap=4, bins=15, fmt='png', xlim=None, ylim=None):
# Remove rows with missing values in the statistic column or team name
df_valid = df.dropna(subset=[stat_col, 'Team'])
# Create a grid of plots by team
g = sns.FacetGrid(df_valid, col="Team", col_wrap=col_wrap, sharex=True, sharey=False, height=3, aspect=1.2)
g.map(sns.histplot, stat_col, bins=bins)
# Apply axis limits if any
if xlim or ylim:
    for ax in g.axes.flatten():
        if xlim: ax.set_xlim(xlim)
        if ylim: ax.set_ylim(ylim)
# Set titles and axis labels
g.set_titles("{col_name}")
g.set_axis_labels(stat_col, "Frequency")
plt.suptitle(f'Distribution of {stat_col} per Team', y=1.02)
# Adjust layout and save figure
```

```
    g.tight_layout(rect=[0, 0.03, 1, 0.98])
    _save_plot(g.fig, f'hist_facet_team_{_safe_filename(stat_col)}', fmt)
```

These charts are created for the 3 attacking and defensive indicators selected in `config_part2.py`, as these are considered 6 important and balanced indicators in football, including:

- Performance: goals
- Performance: assists
- Shooting: Standard: SoT/90 (Shots on Target per 90 minutes)
- Defensive Actions: Tackles: TklW (Tackles Won)
- Defensive Actions: Blocks: Int (Interceptions)
- Miscellaneous: Aerial Duels: Won% (Aerial Duels Won Percentage)

Results and Comments: Since the plotted results include many images, I will select one typical indicator for analysis in the report. Detailed result images are in the plots directory on Github. Evaluate the results of the Performance: goals indicator because goals are the decisive factor in match outcomes and team rankings, and the number of goals is the most common indicator for evaluating attacking player performance.

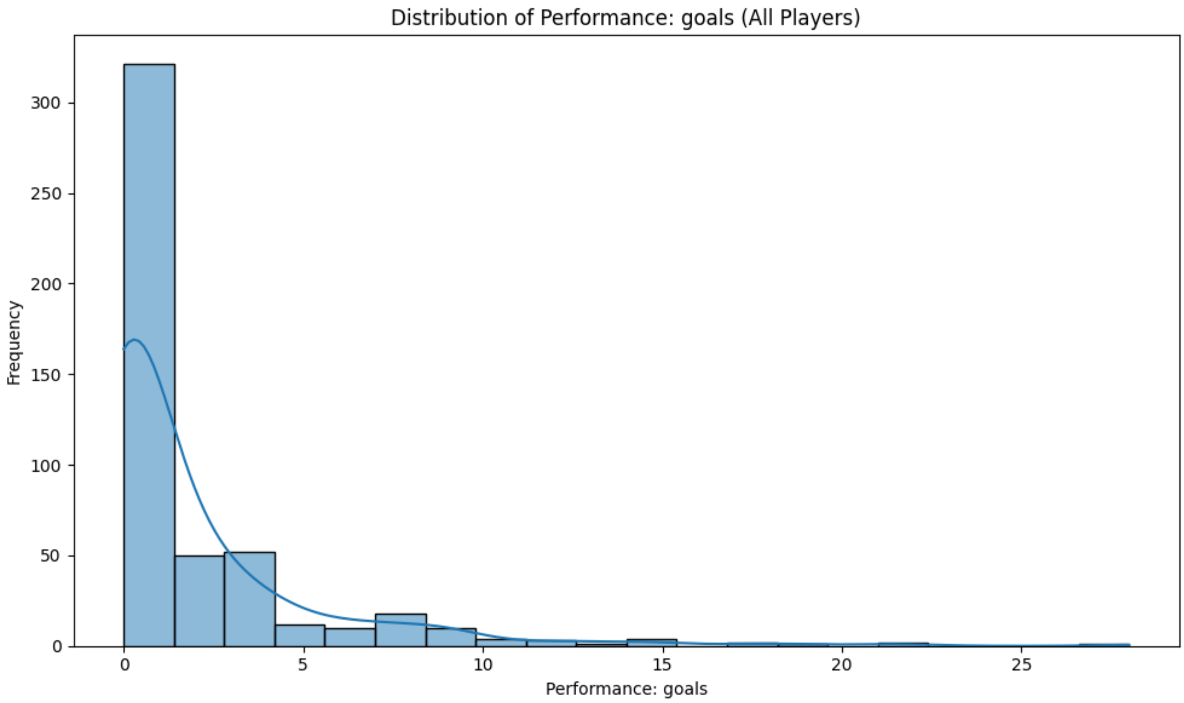

Figure 2.2: Distribution chart of Performance goals of players

The distribution of the number of goals shows a clear right skew, indicating that the majority of players score very few goals, while only a few score many goals.

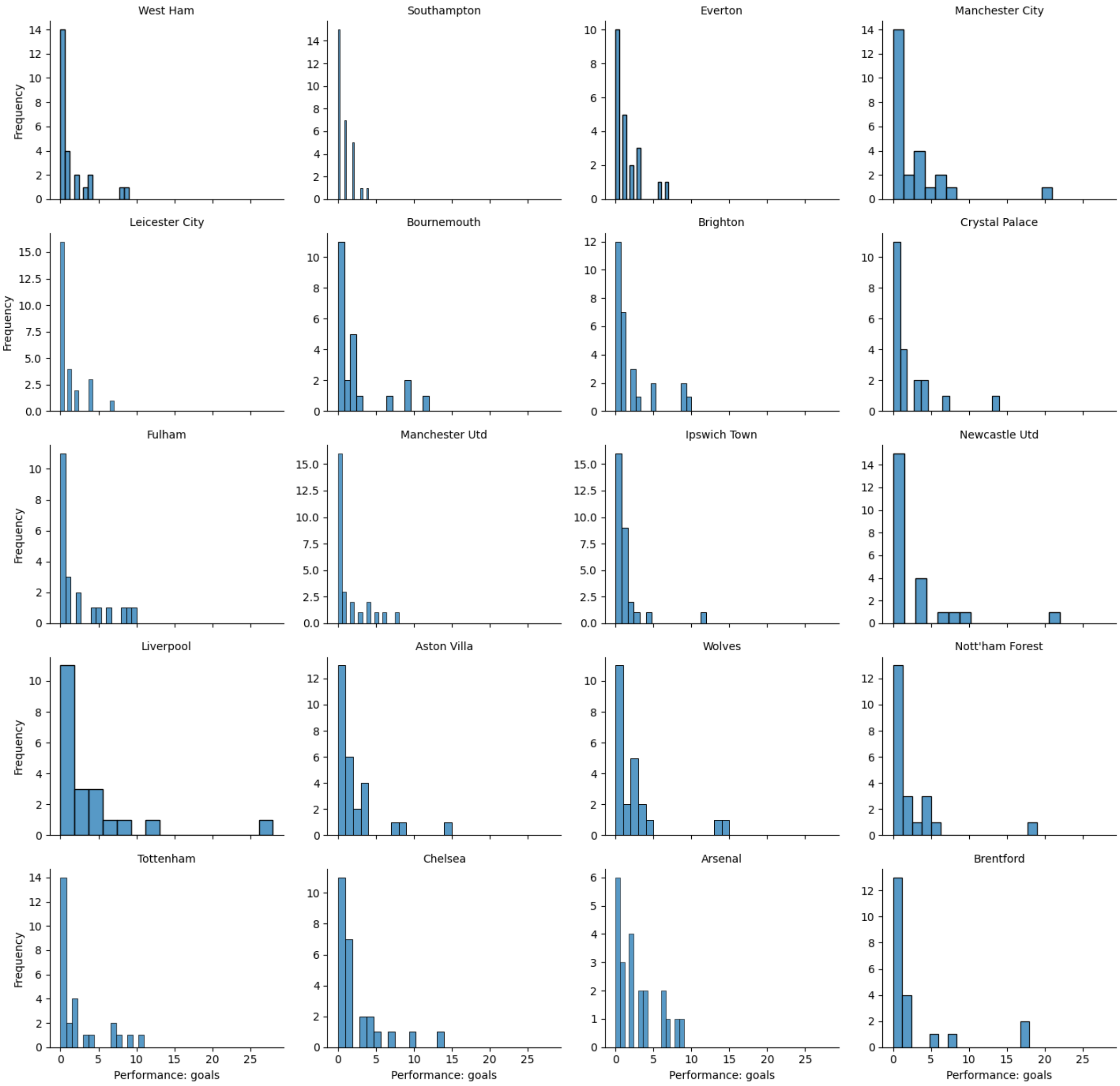

Figure 2.3: Chart of Performance goals of teams

A similar trend is observed for each team, with the majority of players scoring few goals. Some teams like Manchester City and Arsenal have more players scoring above average, reflecting stronger attacking capabilities.

2.6 Identifying the Leading Team for Each Indicator and the Best Overall Team

One of the assignment requirements is to identify the team with the highest score for each statistical indicator and thereby find the team with the best overall performance. In the process of identifying the best overall team, not all indicators are equally weighted. Some indicators, if high, reflect poor performance or negative aspects of a team. To ensure that the evaluation of the strongest team is based on positive factors, I have defined a list of "negative indicators" (`NEGATIVE_STATS`) in the configuration file `config_part2.py`.

These indicators are excluded when summing the number of times a team ranks first to find the team with the best overall performance. The `print_top_team_per_statistic` function in the `analysis.py` module is used to perform this task.

```python
# Snippet from analysis.py - print_top_team_per_statistic function
def print_top_team_per_statistic(summary_df, relevant_stats):
    # Filter out teams, ignore the 'all' aggregate row
    teams_df = summary_df[summary_df['Team'] != 'all'].copy()
    # List to store the top team for each positive indicator
    top_teams_for_positive_stats = []
    # List of positive indicators considered
    positive_stats_considered = []
    # Determine the best team for each indicator
    for stat in relevant_stats:
        col = f'Mean of {stat}' # Column containing the mean value of the indicator

        # Convert column to numeric, remove NaN
        teams_df[col] = pd.to_numeric(teams_df[col], errors='coerce')
        valid_df = teams_df.dropna(subset=[col])

        # Find the team with the highest mean value for the current indicator
        top_row = valid_df.loc[valid_df[col].idxmax()]
        top_team = top_row['Team']
        top_value = top_row[col]
        print(f"Top team for '{stat}': {top_team} (mean = {top_value:.3f})")

        # NEGATIVE_STATS is defined in config_part2.py
        if stat not in NEGATIVE_STATS:
            top_teams_for_positive_stats.append(top_team)
            positive_stats_considered.append(stat)

    # Determine the best overall team (appears most frequently in the top of positive indicators)
    top_team_series = pd.Series(top_teams_for_positive_stats)
    most_common_team = top_team_series.value_counts().idxmax()
    most_common_count = top_team_series.value_counts().max()
    # Function to print the results to the screen
    print(f"\nOverall best performing team (most frequent top team in non-negative statistics): {most_common_team} (appeared {most_common_count} times / {len(positive_stats_considered)} non-negative statistics considered)")
```

This function takes `summary_df` (DataFrame containing summary statistics calculated in the previous step) and `relevant_stats` (list of indicators to analyze) as input.

1. **Identifying the leading team for each indicator:**

   First, the function filters out the "all" aggregate row to consider only team data. For each indicator in `relevant_stats`:

   The function finds the corresponding mean value column (e.g., 'Mean of Performance: goals'). Then, it identifies the team with the highest mean value for that indicator and prints the result. This indicates which team is performing best in that specific aspect.

2. **Identifying the best overall team:** While iterating through the indicators, if an indicator is not considered "negative" (e.g., number of yellow cards, number of fouls – defined in the `NEGATIVE_STATS` variable from the `config_part2.py` file), the team leading that indicator is recorded. Finally, the function counts which team appears most frequently in the list of teams leading the positive indicators. The team that appears most frequently is considered the Best Overall Team and is printed along with the number of appearances out of the total positive indicators considered. This forms the basis for analyzing the team with the best comprehensive performance in the league.

Results and comments:

From the results below, it can be seen that Liverpool leads in many important indicators, especially those related to attack and ball control. Manchester City also stands out in indicators related to passing and control in the opponent's final third. Crystal Palace and Brentford show effectiveness in some defensive aspects. Most importantly, when considering non-negative indicators, Liverpool is the team that appears most frequently in the leading position, with 26 appearances out of a total of 63 positive indicators considered. This indicates that Liverpool is the team with the most impressive overall performance in the league based on this analysis. Below are the results extracted from running the `_top_team_per_statistic` function:

Playing Time

Matches played	Liverpool (24.476)
Starts	Brentford (17.810)
Minutes	Liverpool (1596.381)

Performance

Goals	Liverpool (3.762)
Assists	Liverpool (2.810)
Yellow cards	Bournemouth (3.783)
Red cards	Arsenal (0.227)

Expected

xG	Liverpool (3.629)
xAG	Liverpool (2.629)

Progression

PrgC	Manchester City (40.560)
PrgP	Liverpool (81.381)
PrgR	Liverpool (80.619)

Per 90 minutes

Gls	Manchester City (0.183)
Ast	Liverpool (0.148)
xG	Aston Villa (0.193)
xAG	Chelsea (0.153)

Goalkeeping: Performance

GA90	Leicester City (2.730)
Save%	Bournemouth (80.000)
CS%	Brentford (59.100)

Goalkeeping: Penalty Kicks

Save%	Everton (100.000)

Shooting: Standard

SoT%	Nott'ham Forest (38.990)
SoT/90	Fulham (0.544)
G/Sh	Arsenal (0.137)
Dist	Nott'ham Forest (19.090)

Passing: Total

Cmp	Liverpool (778.095)
Cmp%	Manchester City (86.548)
TotDist	Liverpool (13238.143)

Passing: By Distance

Short Cmp%	Manchester City (92.152)
Medium Cmp%	Manchester City (89.580)
Long Cmp%	Liverpool (60.324)

Passing: Expected

KP	Liverpool (22.000)
1/3	Liverpool (67.714)
PPA	Liverpool (18.619)
CrsPA	Fulham (4.227)
PrgP	Liverpool (81.381)

Goal and Shot Creation: SCA

SCA	Liverpool (49.810)
SCA90	Liverpool (2.633)

Goal and Shot Creation: GCA

GCA	Liverpool (6.476)
GCA90	Liverpool (0.348)

Defensive Actions: Tackles

Tkl	Crystal Palace (32.619)
TklW	Crystal Palace (19.143)

Defensive Actions: Challenges

Att	Liverpool (28.286)
Lost	Crystal Palace (14.048)

Defensive Actions: Blocks

Blocks	Crystal Palace (20.476)
Sh	Brentford (8.381)
Pass	Crystal Palace (14.571)
Int	Bournemouth (14.000)

Possession: Touches

Touches	Liverpool (1102.048)
Def Pen	Brentford (142.619)
Def 3rd	Brentford (357.333)
Mid 3rd	Liverpool (497.524)
Att 3rd	Manchester City (343.480)
Att Pen	Liverpool (55.810)

Possession: Take-Ons

Att	Arsenal (29.727)
Succ%	Liverpool (54.910)
Tkld%	Leicester City (48.304)

Possession: Carries

Carries	Manchester City (643.480)
PrgDist	Manchester City (1994.160)
PrgC	Manchester City (40.560)
1/3	Manchester City (30.360)
CPA	Manchester City (14.040)
Mis	Nott'ham Forest (23.000)
Dis	Newcastle Utd (17.652)

Possession: Receiving

Rec	Liverpool (769.667)
PrgR	Newcastle Utd (17.652)

Miscellaneous: Performance

Fls	Bournemouth (19.826)
Fld	Newcastle Utd (17.609)
Off	Nott'ham Forest (3.727)
Crs	Fulham (36.818)
Recov	Bournemouth (71.435)

Miscellaneous: Aerial Duels

Won	Brentford (26.762)
Lost	Crystal Palace (27.143)
Won%	Southampton (54.172)

Top: Liverpool (appeared 26 times / 63 non-negative statistics considered)

2.7 Main Execution Process

The `main` function in the `MAIN_part2.py` module coordinates the entire data analysis process in Chapter 2. This process is designed to execute a sequence of steps sequentially, from loading data to generating analytical results and visualizations.

```python
# Snippet from MAIN_part2.py - main function
def main():
    # 1. Load and preprocess data
    df, numeric_cols = load_data()

    # 2. Find Top/Bottom 3 players for each indicator
    find_top_bottom_players_all_stats(df, numeric_cols)

    # 3. Calculate and save summary statistics table by team
    summary_df = generate_statistics_summary(df, numeric_cols)

    # 4. Identify leading team for each indicator and best overall team
    # (Uses summary_df from step 3)
    print_top_team_per_statistic(summary_df, numeric_cols)

    # 5. Create and save histogram charts
    generate_histograms(df)
```

Description of execution steps:

The `main` function sequentially performs the following functions:

1. **Load data:** Read data from the `results.csv` file (from Chapter 1), convert necessary numerical columns to numeric type.

2. **Find Top/Bottom 3 players:** Use the `get_top_bottom_players` function from `analysis.py` to identify the top 3 and bottom 3 players for each indicator. The results are saved to the file `top_3_txt`.

3. **Create summary statistics table:** Call the `calculate_stats_summary` function from `analysis.py` to calculate statistical parameters (mean, median, standard deviation) for each team. This summary table is saved to the file `results2.csv`.

4. **Identify leading team and best overall team:** Use `summary_df` (from step 3) and the `print_top_team_per_statistic` function from `analysis.py` to find the leading team for each indicator and the team with the best overall performance based on positive indicators. The results are printed to the console.

5. **Create Histogram charts:** Use plotting functions from `plotting.py` to draw distribution charts for indicators in `SELECTED_STATS`. The charts are saved in the `plots` directory.

Chapter 3

Player Clustering using K-Means and PCA

In this section, I will present the process of applying the K-Means clustering algorithm to group English Premier League football players for the 2024-2025 season into groups with similar statistical characteristics. Next, I use Principal Component Analysis (PCA) to reduce data dimensionality, helping to visualize player clusters on a 2D chart, thereby further clarifying the relationships and differences between groups.

3.1 Introduction to K-Means and Choosing the Number of Clusters (K)

K-Means is a popular unsupervised learning algorithm used for data clustering. The algorithm works by dividing N data points into K clusters such that each point is closest to its cluster center. The main goal is to reduce the sum of squared distances from the points to their cluster centers – this metric is also known as Inertia[3]. To determine the optimal number of clusters (K) for the player dataset, I used the "Elbow" method This method runs K-Means with multiple K values (from 2 to 15), calculates Inertia for each K, and plots a graph. From the graph, the "elbow" point – where Inertia no longer decreases significantly even as K increases – will be chosen as the optimal number of clusters. The reason for choosing K values ranging from 2 to 15 is that **a rule of thumb** suggests that the maximum number of clusters, k, should be chosen approximately according to the formula $k \approx \sqrt{n/2}$, where n is the total number of data points [6]. Specifically in this section $N = 491$, where 491 is the number of players recorded in the results of Chapter 1. According to the above rule, the maximum k value to be tested is $k \approx \sqrt{491/2} \approx 15.67$, so limiting K to the range from 2 to 15 is reasonable.

```python
# Snippet from MAIN_part3.py - plot_elbow_method function

def plot_elbow_method(df_scaled, k_range, output_path):
    # Calculate inertia (sum of distances of points to cluster center) for each k value
    inertia = [KMeans(n_clusters=k, random_state=42, n_init='auto').fit(df_scaled).inertia_ for k in k_range]

    # Plot the Elbow graph to choose the optimal number of clusters
    plt.plot(k_range, inertia, 'o--')
    plt.xlabel('k');
    plt.ylabel('Inertia'); plt.title('Elbow Method')
```

```
    plt.xticks(k_range); plt.grid(True); plt.tight_layout()
    plt.savefig(output_path);
    plt.close()

```

Result: Below is the Elbow chart obtained from the analysis process:

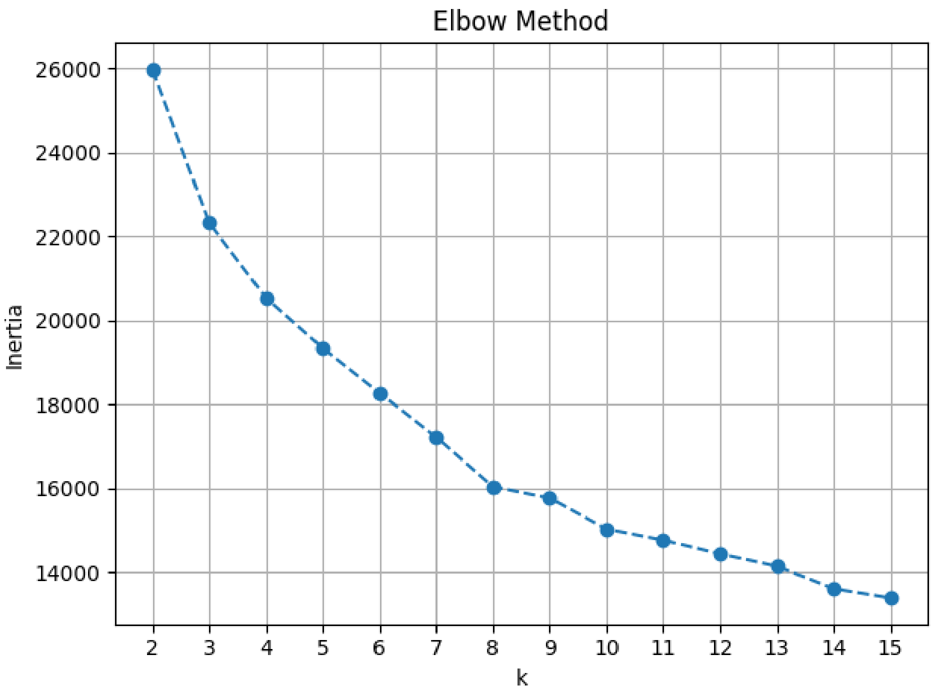

Figure 3.1: Elbow Method Chart

Observing the figure above, I noticed that from K=2 to K=4, the Inertia value decreases very rapidly. This decrease is still significant as K increases from 4 to 6. However, after K=6, the curve begins to flatten, and increasing the number of clusters further no longer yields a strong reduction in Inertia. Although there might be another gentler "elbow" at K=8, to balance having enough clusters to distinguish player groups and avoid making the model overly complex, I decided to choose K=6 as the optimal number of clusters for this analysis. To perform clustering, the team used all numerical statistical indicators in the dataset, after removing identifying information such as player name, nationality, team, playing position, and age. These indicators reflect various aspects of a player's performance. Before being fed into the K-Means model, the data was standardized using the `StandardScaler` method to ensure balance between scales, preventing a few indicators from dominating the clustering results.Feature scaling is necessary for distance-based algorithms like K-Means to prevent features with larger scales from dominating the clustering results[7]. Additionally, the `KEY_STATS_FOR_INTERPRETATION` variable set (comprising 18 indicators) will be used to describe and interpret the clusters after analysis.

```
# Snippet from MAIN_part3.py - run_kmeans function and OPTIMAL_K selection
OPTIMAL_K = 6 # Chosen

def run_kmeans(df_scaled, n_clusters):
    kmeans = KMeans(n_clusters=n_clusters, random_state=42, n_init='auto')
```

```
    cluster_labels = kmeans.fit_predict(df_scaled)
    return cluster_labels
```

3.2 K-Means Clustering Results

After running the K-Means algorithm with K=6, the players in the league were divided into 6 different groups. Below is a summary of the distribution of the number of players in each cluster and the main characteristics of each cluster.

Distribution of players by cluster:

Cluster 0: 181 players (36.9

Cluster 1: 97 players (19.8

Cluster 2: 50 players (10.2

Cluster 3: 95 players (19.3

Cluster 4: 41 players (8.4

Cluster 5: 27 players (5.5

Detailed characteristics of each cluster:

Cluster 0: Consists mostly of defenders (DF). In general, this group seems to lean towards traditional defense, with less involvement in attack and build-up play. **Cluster 1:** Primarily midfielders (MF). These appear to be energetic midfielders, skilled at ball recovery, tackling, and contributing to ball progression, but not strong in scoring. **Cluster 2:** Includes forwards (FW) and midfielders (MF). This group excels in many aspects, from defense and ball progression to attack and scoring. These could be well-rounded attacking players. **Cluster 3:** Consists of forwards (FW) and midfielders (MF). This group appears weak in defensive abilities and ball progression, but has high attacking stats (assists, goals, shots on target). However, a low xG (expected goals) suggests they might be good finishers or capitalize on chances from less favorable situations. **Cluster 4:** Mostly defenders (DF). These seem to be defenders skilled in defense, tackling, and capable of passing to develop attacks, but less likely to carry the ball forward themselves or directly participate in final attacking plays. **Cluster 5:** Primarily forwards (FW). This group seems less involved in defense and passing for build-up play, but is very strong in the opponent's box, skilled at dribbling, assisting, scoring, and has good attacking statistics. These could be typical strikers.

General comments on clustering results:

The division into 6 clusters has shown relatively clear differences between player groups based on their roles and playing styles. Cluster 0 has the largest number of players, which is understandable as a team always needs many players to undertake defensive roles. Conversely, more specialized clusters like Cluster 2 or Cluster 5 have fewer players, indicating that these are roles requiring specific skills that not every player can fulfill.

3.3 Visualizing Clusters with PCA

To visualize the K-Means player clusters, I applied Principal Component Analysis (PCA). PCA helps reduce the dimensionality of the (standardized) player characteristic data to 2 principal components (PC1 and PC2). The result is a 2D scatter plot, where each point is a player and the color represents the player's cluster.

```python
# Snippet from MAIN_part3.py - visualize_pca function
def visualize_pca(df_scaled_with_clusters, df_identifiers, k, output_path):
    # Reduce data dimensionality using PCA and visualize clusters
    pca = PCA(n_components=2)
    comps = pca.fit_transform(data.drop(columns='Cluster'))
    data_pca = pd.DataFrame(comps, columns=['PC1', 'PC2'])
    data_pca['Cluster'] = data['Cluster']

    # Plot scatter plot of clusters by two principal components
    sns.scatterplot(data=data_pca, x='PC1', y='PC2', hue='Cluster')
    plt.title(f'PCA Clustering (K={k})')
    plt.savefig(output_path)
```

Result of the 2D scatter plot:

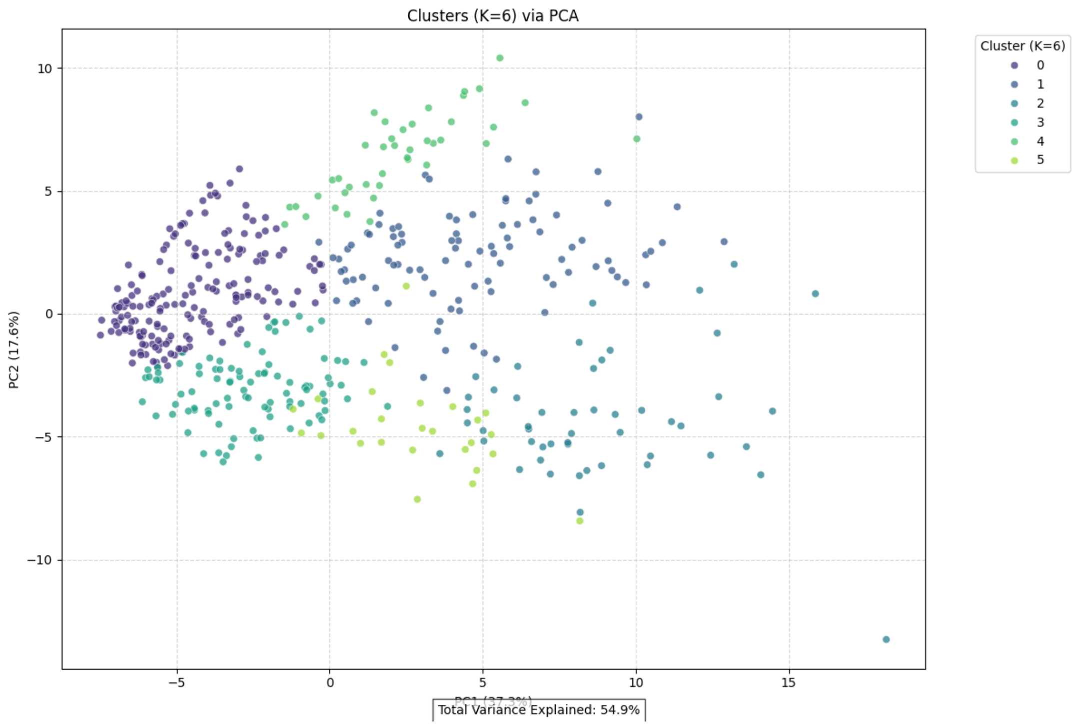

Figure 3.2: 2D scatter plot of player clusters after applying PCA (K=6)

Comments on the PCA chart: The PCA chart (Figure above) shows the distribution of the 6 player clusters in a 2-dimensional space created by the first two principal components. These two components explain a total of 54.9From the chart, I can observe:

- There is a certain separation between clusters, although some clusters have overlapping regions. For example, Cluster 0 (dark purple) occupies a large and somewhat separate area of space, reflecting the majority group of defenders with rather distinct characteristics.
- Attacking player clusters tend to be located in different regions of the chart, indicating diversity in attacking roles. For example, Cluster 2 (light green) and Cluster 5 (yellow) may occupy different positions, reflecting differences in their skill sets despite both being attacking players.

Overall, the PCA chart provides a visual aid, supporting a better understanding of the data structure and the relative relationships between the player groups classified by K-Means. Although only a portion of the original data's variance is retained, reducing the dimensionality to 2D is still useful for analysis and presentation of results.

Chapter 4

Estimating Player Value

This section focuses on collecting transfer value data of English Premier League players for the 2024-2025 season and proposes a method to estimate their value based on collected statistical data. The goal is to build a model capable of predicting a player's market value, which is important information for team evaluation and management.

4.1 Program Structure

To implement the requirements of Chapter 4, the program is organized into the following modules:

`MAIN_part4.py`: The main module coordinating the process of collecting transfer value data and processing the data.

`scraper_part4.py`: Contains functions to access and extract player transfer value information from the footballtransfers.com website.

`processor_part4.py`: Responsible for processing the collected raw data, especially filtering players based on playing time criteria (>900 minutes) from the results of Chapter 1.

`config_part4.py`: Stores necessary configurations for scraping such as URL, CSS selectors for data extraction, minimum playing time threshold, and output file names.

`training_pipeline.py`: This module contains the entire pipeline for building the player value estimation model, including data loading, preprocessing, feature selection, model training, and evaluation.

4.2 Collecting Transfer Value Data

4.2.1 Choosing Tools and Data Source

Estimated Transfer Value (ETV) data for players is collected from the website footballtransfers.com. Similar to Chapter 1, I use `Selenium`, and then `BeautifulSoup` is used to parse the HTML of the page.

4.2.2 Implementation Details

The data collection process is primarily carried out by the `scrape_transfer_data` function in `scraper_part4.py`. The main logic of this function includes:

1. **Accessing the main URL**: Navigate to the Premier League player list page on football-transfers.com (defined in `TRANSFER_URL` in `config_part4.py`).
2. **Iterating through pages**: Use Selenium to automatically click the next page button (using `NEXT_PAGE_SELECTOR`) to load the player list from all pages.
3. **Extracting basic information**: From each player data row (identified by `PLAYER_ROW_SELECTOR`), extract the following information:
    - Player name (`PLAYER_NAME_SELECTOR`)
    - Team (`TEAM_NAME_SELECTOR`)
    - Age (`AGE_SELECTOR`)
    - Position (`POSITION_SELECTOR`)
    - Current Estimated Transfer Value (ETV) (`ETV_SELECTOR`)
    - URL of the player's individual page.
4. **Collecting highest ETV**: For each player, access their individual page URL. From the individual page, extract the highest recorded ETV (using `HIGHEST_ETV_SELECTOR_PROFILE`). This is done by the `scrape_highest_etv` function.
5. **Cleaning data**: ETV values (e.g., "€10.5m", "€500k") are cleaned and converted to a numerical format (million EUR) using the `clean_value` function.
6. **Saving raw data**: The collected data is saved to the file `raw_data_with_highest_etv.csv` (defined in `RAW_DATA_FILENAME`).

Code snippet illustrating the main logic in `scraper_part4.py` (`scrape_transfer_data` and `scrape_highest_etv` functions):

```python
def scrape_transfer_data(driver: WebDriver) -> bool:
    # Initialize data storage and progress tracking variables
    data, scraped_keys = [], set()
    page = 1
    # Open the transfer list website
    driver.get(TRANSFER_URL)
    WebDriverWait(driver, 20).until(EC.presence_of_element_located((By.CSS_SELECTOR,
    ↪   PLAYER_TABLE_SELECTOR)))
    # Loop through pages to get data
    while True:
        # Wait for the player table to appear
        WebDriverWait(driver, WAIT_TIME).until(
            EC.visibility_of_element_located((By.CSS_SELECTOR, PLAYER_ROW_SELECTOR))
        )
        # Parse page data
        soup = BeautifulSoup(driver.page_source, 'html.parser')
        rows = soup.select(PLAYER_ROW_SELECTOR)
        # Skip if no data found
        if not rows:
            driver.refresh()
            continue
        # Process each player data row
```

```python
        added = 0
        for row in rows:
            name = safe_get(row, PLAYER_NAME_SELECTOR)
            team = safe_get(row, TEAM_NAME_SELECTOR)
            url = safe_get_attribute(row, PLAYER_NAME_SELECTOR, 'href')
            if not (name and team and url): continue
            # Check for duplicates
            key = (name.strip(), team.strip())
            if key in scraped_keys: continue
            # Create player information
            player = {
                'Player': key[0],
                'Team_TransferSite': key[1],
                'Age': safe_get(row, AGE_SELECTOR),
                'Position': safe_get(row, POSITION_SELECTOR),
                TARGET_VARIABLE: clean_value(safe_get(row, ETV_SELECTOR)),
                'Profile_URL': url,
                'Highest_ETV': None
            }
            # Add to list
            data.append(player)
            scraped_keys.add(key)
            added += 1
        # Go to the next page if available
        try:
            next_btn = WebDriverWait(driver, WAIT_TIME).until(
                EC.element_to_be_clickable((By.CSS_SELECTOR, NEXT_PAGE_SELECTOR))
            )
            driver.execute_script("arguments[0].click();", next_btn)
            time.sleep(random.uniform(2, 4))
            page += 1
        except:
            # End when there are no more pages
            break
    # Collect Highest_ETV information from individual pages
    for i, player in enumerate(data):
        url = player.get('Profile_URL')
        player['Highest_ETV'] = scrape_highest_etv(driver, url) if url else None
        time.sleep(random.uniform(0.5, 1.5))
    # Remove unnecessary URL data
    for p in data:
        p.pop('Profile_URL', None)
    # Save collected data
    return save_data(data)
```

```python
def scrape_highest_etv(driver: WebDriver, url: str) -> float | None:
    if not url: return None
    # Open profile page
```

```python
    driver.get(url)
    WebDriverWait(driver,
    ↪ WAIT_TIME).until(EC.presence_of_element_located((By.CSS_SELECTOR,
    ↪ PROFILE_PAGE_LOAD_CHECK_SELECTOR)))
    time.sleep(random.uniform(1, 2))
    # Get the highest ETV value
    etv_text = safe_get(BeautifulSoup(driver.page_source, 'html.parser'),
    ↪ HIGHEST_ETV_SELECTOR_PROFILE)
    return clean_value(etv_text) if etv_text else None
```

4.3 Data Processing

4.3.1 Player Filtering

After collecting raw transfer value data, the next step is to process and filter this data. [The `process_transfer_data` function in `processor_part4.py` performs this task.

> **Get valid player list**: The `get_valid_players_from_part1` function is called to read the `results.csv` file from Chapter 1. From this, a list of players with playing time (Playing Time: minutes) greater than the `MIN_MINUTES_THRESHOLD` (900 minutes) defined in `config_part4.py` is extracted.
>
> **Filter transfer data**: The DataFrame containing raw transfer data is filtered to keep only players whose names are in the valid player list obtained in the previous step.
>
> **Save results**: The filtered data is saved to the file `estimation_data_with_highest_etv.csv`

Code snippet illustrating the main logic in `processor_part4.py` (`get_valid_players_from_part1` and `process_transfer_data` functions):

```python
def get_valid_players_from_part1() -> set:
    # Read data from part 1 results file
    df = pd.read_csv(PART1_RESULTS_FILE)
    # Convert minutes column to numeric and remove commas
    df['minutes'] =
    ↪ pd.to_numeric(df[PART1_MINUTES_COLUMN].astype(str).str.replace(',', ''),
    ↪ errors='coerce')
    # Filter players with playing minutes greater than the minimum threshold
    players = set(df.loc[df['minutes'] > MIN_MINUTES_THRESHOLD, PART1_PLAYER_COLUMN]
                .dropna().astype(str).str.strip())
    return players
```

```python
def process_transfer_data() -> pd.DataFrame | None:
    # Get valid player list from part 1
    valid_players = get_valid_players_from_part1()
    # Read raw data from file
    df = pd.read_csv(RAW_DATA_FILENAME)
    # Standardize player names
    df['Player'] = df['Player'].astype(str).str.strip()
```

```python
    # Filter data to keep only valid players
    filtered = df[df['Player'].isin(valid_players)].copy()
    # Create output directory if it doesn't exist
    os.makedirs(OUTPUT_FOLDER, exist_ok=True)
    # Save filtered data
    filtered.to_csv(ESTIMATION_READY_DATA_FILENAME, index=False, encoding='utf-8-sig')
    return filtered
```

4.3.2 Collection and Processing Results

The final data ready for building the value estimation model is saved in the file `estimation_data_with_highe` This file contains information on player name, age, position, current transfer value (ETV), and highest ever ETV, including only players who have played over 900 minutes in the season.

	A	B	C	D	E	F
1	Player	Team_TransferSite	Age	Position	TransferValue_EUR_Millions	Highest_ETV
2	Erling Haaland	Man City	24	F (C)	198.8	199.6
3	Martin Ødegaard	Arsenal	26	M, AM (C)	126.5	134.5
4	Alexander Isak	Newcastle Utd.	25	F (C)	120.3	120.3
5	Cole Palmer	Chelsea	22	M (C)	115.4	119
6	Declan Rice	Arsenal	26	M (C)	107.8	120
7	Alexis Mac Allister	Liverpool	26	M, DM, AM (C)	106.1	117
8	Phil Foden	Man City	24	AM (R), M (C)	105.7	138.6
9	Bukayo Saka	Arsenal	23	F, M (R)	101.3	127.5
10	Ryan Gravenberch	Liverpool	22	DM, M (C)	85.3	85.5
11	Bruno Guimarães	Newcastle Utd.	27	DM, M (C)	83.2	83.2
12	Moisés Caicedo	Chelsea	23	DM, M (C)	80.7	86.1
13	William Saliba	Arsenal	24	D (C)	79.5	79.5
14	Omar Marmoush	Man City	26	F (C)	79.1	79.1
15	Joško Gvardiol	Man City	23	D (CL)	78.7	86.6
16	Gabriel Magalhães	Arsenal	27	D (C)	75.5	75.5
17	Sávio	Man City	21	F (R), M (RL)	74.8	74.8
18	Enzo Fernández	Chelsea	24	M, DM (C)	73.7	82.7
19	Dominik Szoboszlai	Liverpool	24	M (C), AM (L)	71.4	71.4
20	Brennan Johnson	Tottenham	23	F, M (R)	71.3	71.3
21	Lucas Paquetá	West Ham United	27	AM, M (C)	71.2	75
22	Leny Yoro	Man Utd	19	D (CR)	71	71
23	Luis Díaz	Liverpool	28	F (CL), M (L)	71	78.5
24	Kai Havertz	Arsenal	25	F (C)	70.5	98.6
25	Morgan Rogers	Aston Villa	23	M (CR)	69	79.6
26	Murillo	Nottingham	22	D (C)	68.4	68.4
27	Cody Gakpo	Liverpool	25	F, M, AM (L)	67.5	72.2
28	Nicolas Jackson	Chelsea	23	F (C)	66.2	71.8
29	Kobbie Mainoo	Man Utd	20	M, DM (C)	66	66
30	Rico Lewis	Man City	20	D (R)	62.7	63.8
31	Matheus Cunha	Wolverhampton	25	F (C)	62.6	62.6
32	Gabriel Martinelli	Arsenal	23	F, AM (L)	62.6	70.1
33	Eberechi Eze	C. Palace	26	AM, M (C), F (L)	62.2	62.2
34	Amadou Onana	Aston Villa	23	DM, M (C)	62.1	64.5

estimation_data_with_highest_et

Figure 4.1: Transfer value estimation data in csv

4.4 Proposed Method for Estimating Player Value

4.4.1 Introduction

The goal is to build a machine learning model capable of estimating the transfer value (ETV) of a player based on performance statistics (from Chapter 1) and other basic information (age, position). The target variable is `TransferValue_EUR_Millions` (cleaned from the collected data).

4.4.2 Feature Selection

Feature selection is a crucial step in building an effective model.

Combining data: Statistical data from `results.csv` (Chapter 1) is combined with transfer value data from `estimation_data_with_highest_etv.csv` (Chapter 4) based on player name (Player).

Removing identifier columns: Identifier columns such as `Player`, `Nation`, `Team`, (listed in `ID_COLS` of `training_pipeline.py`) are removed from the input feature set (X) as they are not generalizable for prediction or may cause data leakage.

Handling Numerical Features:

- All columns containing statistical figures, age, and `Highest_ETV` are considered numerical features.
- N/a values or non-numeric strings are converted to NaN.
- Columns with a high percentage of missing values (specifically > 50

Handling Categorical Features:

- The `Position` column (and other columns if any after conversion to numeric) are considered categorical features (managed by `CAT_COLS_BASE` and automatically detected in `training_pipeline.py`).
- Missing values in categorical columns are filled with a fixed value like 'Missing'.

Target Variable: The `TransferValue_EUR_Millions` value is used as the target variable (y).

To minimize the impact of skewed distribution and large outliers, a logarithmic transformation (`np.log1p`) is applied to the target variable.

The model will predict the log value of ETV, then the predicted value will be transformed back (`np.expm1`) to get the actual ETV.

4.4.3 Model Selection

Based on the regression nature of the problem – predicting a continuous value (player value) and the potential complexity of the relationship between features and player value, I conducted a comprehensive analysis along with referencing previous studies. Specifically, the research *"Football Player Value Prediction: Comparing Machine Learning Models with Cross-Validation"* [5] by Yitong Kong (Sorbonne University, Paris, France) utilized data from Kaggle and applied cross-validation to evaluate the performance of prediction models. The results indicated that the Gradient Boosting model achieved the best performance, with the lowest Root Mean Square Error (RMSE) of 0.450 and the highest R-squared (coefficient of determination) of 0.928.

Furthermore, the study *"Comparing Machine Learning and Ensemble Learning in the Field of Football"* [4] also emphasized that ensemble learning models, particularly Gradient Boosting and Random Forest, often outperform basic machine learning models due to their ability to minimize prediction errors and improve overall accuracy. This research pointed out that in problems with complex, high-dimensional data, ensemble learning models have a distinct advantage due to their capability to synthesize the strengths of multiple sub-models, reduce overfitting, and enhance generalization ability.

From these analyses, I decided to select the **Gradient Boosting Regressor** model (from the `sklearn.ensemble` library) to process the required data, and to implement and evaluate the model's actual performance. This is a powerful machine learning model, based on the principle of decision trees, capable of effectively handling non-linear relationships and complex interactions between features. This model builds decision trees sequentially, where each new tree attempts to correct the errors of the previous one, thereby improving prediction accuracy. Thanks to this sequential learning feature, **Gradient Boosting** can efficiently handle complex datasets and often outperforms conventional linear regression models.

The model building process in `training_pipeline.py` includes:

Data Splitting: Data is split into training and testing sets with an 80:20 ratio.

Preprocessing Pipeline: A `ColumnTransformer` is used to apply different preprocessing steps to different types of columns:

- For numerical features (`num_cols`):
  * `KNNImputer`: Fills missing values using the K-Nearest Neighbors algorithm.
  * `StandardScaler`: Standardizes features by removing the mean and scaling to unit variance.
- For categorical features (`cat_cols`):
  * `SimpleImputer`: Fills missing values with the most frequent value ('most_frequent').
  * `OneHotEncoder`: Converts categorical features into numerical form using one-hot encoding, `handle_unknown='ignore'` to handle new values that may appear in the test set.

Hyperparameter Optimization: `RandomizedSearchCV` is used to find the best set of hyperparameters for the `GradientBoostingRegressor`. The hyperparameters searched include `n_estimators`, `learning_rate`, `max_depth`, `min_samples_split`, `min_samples_leaf`, and `subsample`. The search is performed with 5-fold cross-validation and the objective is to minimize `neg_root_mean_squared_error`.

Model Training: The `GradientBoostingRegressor` model is trained on the preprocessed training set with the best hyperparameters found.

Model Evaluation: The model is evaluated on the test set using the following metrics:

- Root Mean Squared Error (RMSE)
- Mean Absolute Error (MAE)
- R-squared (R2 score)

Code snippet illustrating the construction of the preprocessor and model training in `training_pipeline.py`:

```python
def build_preprocessor(num_cols, cat_cols):
    return ColumnTransformer([
        ('num', Pipeline([('imp', KNNImputer()), ('scale', StandardScaler())]),
        ↪  num_cols),
        ('cat', Pipeline([('imp', SimpleImputer(strategy='most_frequent')),
                          ('oh', OneHotEncoder(handle_unknown='ignore',
                          ↪  sparse_output=False))]), cat_cols) #
    ])

def train(X_train, X_test, y_train, y_test, prep, plot_dir):
    X_train_prep, X_test_prep = prep.fit_transform(X_train), prep.transform(X_test)
    model = GradientBoostingRegressor(random_state=42)
    param_grid = {
        'n_estimators': randint(100, 500),
        'learning_rate': uniform(0.01, 0.2),
```

```
        'max_depth': randint(3, 6),
         'min_samples_split': randint(2, 10),
        'min_samples_leaf': randint(1, 10),
        'subsample': uniform(0.7, 0.3)
    }
    search = RandomizedSearchCV(model, param_grid, n_iter=50, cv=5,
                                scoring='neg_root_mean_squared_error', n_jobs=-1,
                                ↪ random_state=42, verbose=1)
    search.fit(X_train_prep, y_train)
    evaluate(search.best_estimator_, X_test_prep, y_test,
    ↪ prep.get_feature_names_out(), plot_dir)
    return search.best_estimator_, prep
```

4.4.4 Overall Training Pipeline

The `training_pipeline.py` module, when run, will sequentially perform the following steps:

`load_data()`: Loads player statistical data (Chapter 1) and transfer value data (Chapter 4), then merges them.

`split_and_prep_data()`: Prepares data, applies log transformation to the target variable, splits into training/testing sets, identifies numerical and categorical columns, preliminarily handles missing values, and removes columns with too many missing values.

`build_preprocessor()`: Builds the `ColumnTransformer` object for preprocessing.

`train()`: Trains the model, including applying the preprocessor, searching for optimal hyperparameters, training the final model, and calling the `evaluate()` function.

`evaluate()`: Calculates evaluation metrics (RMSE, MAE, R2) and creates visualizations:

- Feature Importance.
- Scatter plot of predicted vs actual values.
- Residuals plot.

4.4.5 Results and Evaluation

Analysis of Model Evaluation Metrics

The player value estimation model achieved the following results on the test set:

- RMSE (Root Mean Squared Error) is 7.06, indicating that the average error between the predicted value and the actual value is approximately 7.06 million EUR.
- MAE (Mean Absolute Error): 5.60. MAE measures the average absolute error between the predicted value and the actual value.
- The MAE value indicates that the model predicts with a deviation of 5.60 million EUR from the actual value.
- MAE is less affected by large outliers compared to RMSE.
- R-squared (R2 score): 0.902. The R2 score indicates the proportion of the variance in the target variable (player value) that the model can explain.

- A value of 0.902 (or 90.2

Overall, these metrics indicate that the model has a fairly high accuracy in estimating player value. Although there is some discrepancy (shown by RMSE and MAE), the ability to explain most of the value variation (R2) is a positive sign.

Analysis of Feature Importance Chart

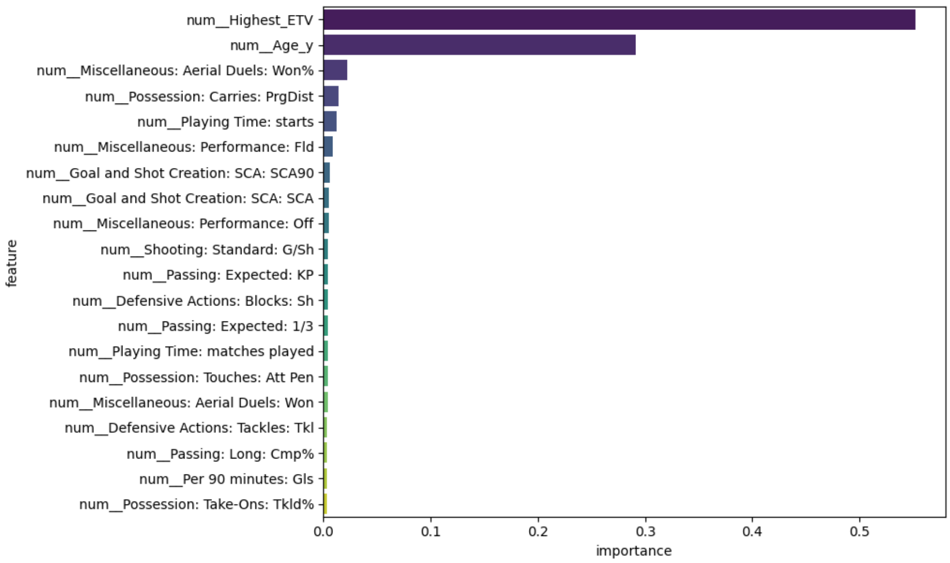

Figure 4.2: Feature importance chart

The feature importance chart above provides information on which factors have the greatest impact on the Gradient Boosting Regressor model's prediction of player value. The most important features:

- num_Highest_ETV: The highest Estimated Transfer Value (ETV) ever recorded for the player. This is the feature with the greatest influence. This is reasonable because historical value is often a strong indicator of current value.
- num_Age_y: The player's age. Age significantly affects a player's development potential, resale value, and peak performance, thus it is an important predictive factor.

Other features with significant influence:

- num_Miscellaneous: Aerial Duels: Won% (Successful aerial duels percentage)
- num_Possession: Carries: PrgDist (Total progressive carrying distance)
- num_Playing Time: starts (Number of matches started)

Analysis of Predicted vs Actual Value Chart

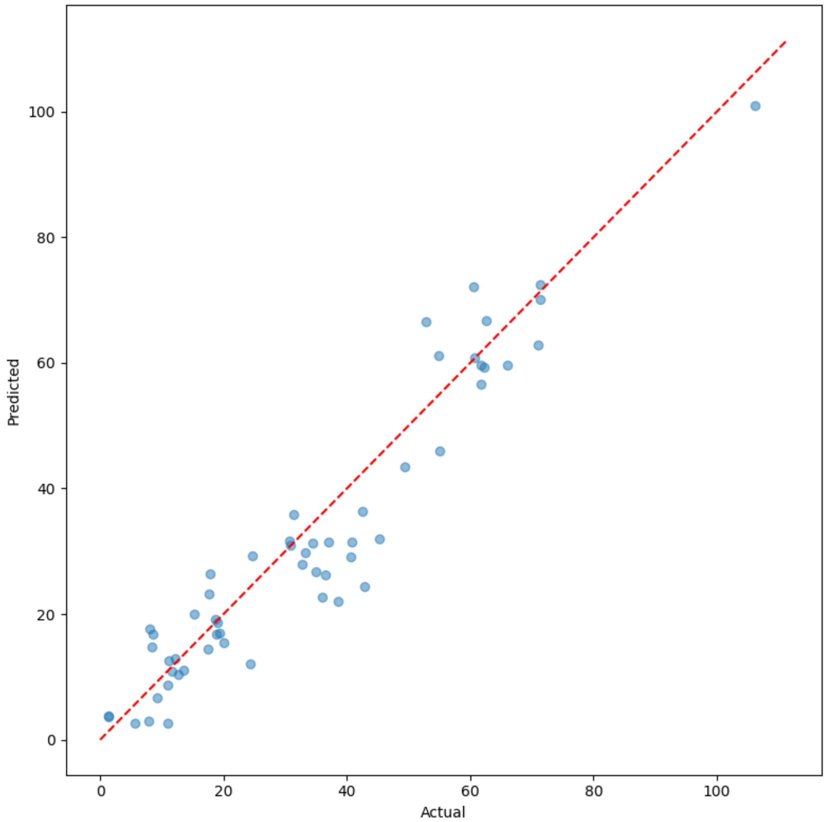

Figure 4.3: Chart of predicted vs actual values

The chart above visualizes the agreement between the transfer values predicted by the model and the actual values.

- Overall assessment: Data points tend to cluster around the y=x diagonal line.
- This indicates that the model predicts fairly well across a wide range of player values.
- Conclusion: This chart reinforces the high R2 result, showing that the model captures the general trend of the data.
- However, it is necessary to note the possibility of less accurate predictions at extreme value levels.

Analysis of Residuals Chart

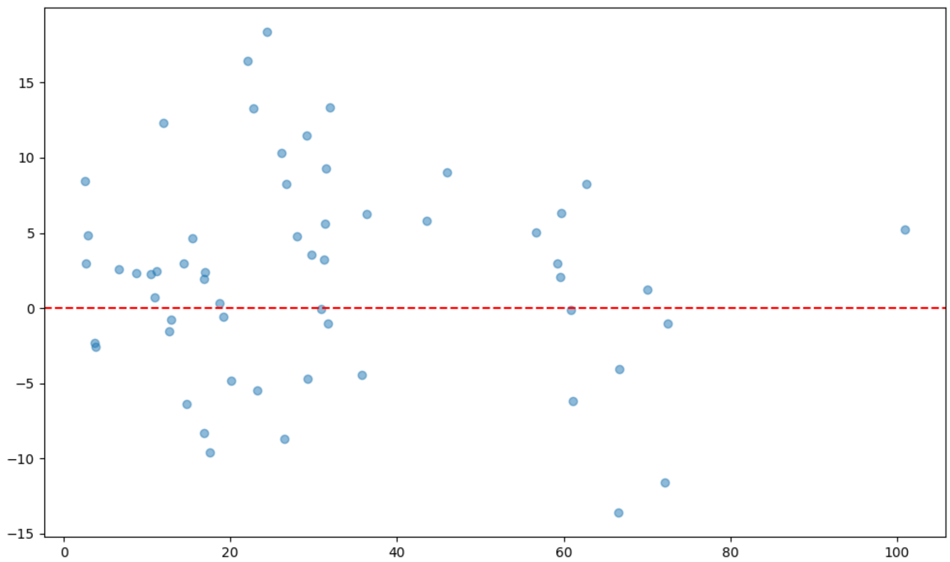

Figure 4.4: Residual data distribution

The residuals chart above shows the difference between the predicted value and the actual value for each data point.

- Distribution of residuals: Ideally, residuals should be randomly distributed around the 0 line, with no clear pattern.
- In the Figure, most of the residual points are distributed around the 0 line.

Conclusion: The residuals chart does not show any serious problems with the model. Most errors are small and randomly distributed.

4.4.6 Overall Evaluation of the Proposed Method

Based on the analyses above, the proposed method for estimating player value is quite effective and suitable for the requirements of Chapter 4.

- Data collection: The process of collecting data from footballtransfers.com and filtering by playing time (>900 minutes) has provided the necessary target dataset.
- Feature selection: Combining performance statistics from Chapter 1, basic player information (age, position), and historical transfer value (highest ETV) is a comprehensive feature selection strategy.
- The results of the feature importance analysis have confirmed the contribution of these types of information.
- Model selection: The **Gradient Boosting Regressor** model, after hyperparameter optimization, has shown good predictive ability with an R2 of **0.902**.
- The model's ability to handle non-linear relationships and interactions between features is appropriate for the complexity of player valuation.

Bibliography

[1] Codecademy Team. Python web scraping using selenium and beautiful soup: A step-by-step tutorial, 3 2024.

[2] GeeksforGeeks. Scrape content from dynamic websites, 1 2025.

[3] Gowsic K, Mugunthan S, Sakthivel Logavaseekarapakther, Puviyarasu A, and Mohammed Farook R. Enhanced unsupervised k-means clustering algorithm. *ShodhKosh: Journal of Visual and Performing Arts*, 5(1):1141–1150, 1 2024.

[4] Carson Kai-Sang Leung, Anh Le Tran, Christopher Leckie, and Kotagiri Ramamohanarao. Comparing machine learning and ensemble learning in the field of football. *ResearchGate*, 2019.

[5] Łukasz Marciniak, Tomasz Marciniak, and Piotr Szwajkowski. Football player value prediction: Comparing machine learning models with cross-validation. *ResearchGate*, 2024.

[6] K. V. Mardia, J. T. Kent, and J. M. Bibby. *Multivariate analysis*. Academic Press, London; New York, 1979.

[7] Chantha Wongoutong. The impact of neglecting feature scaling in k-means clustering. *PLOS ONE*, 19(12):e0310839, 2024.