POSTS AND TELECOMMUNICATIONS INSTITUTE OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY I

—————o0o—————

ASSIGNMENT REPORT 1

PYTHON PROGRAMMING LANGUAGE

Instructor: Kim Ngoc Bach
Student: Vu Thi Thu Duyen
Student ID: B23DCCE027
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

Chapter 1

Collecting English Premier League Player Statistics for the 2024-2025 Season from fbref.com

1.1 Introduction

This exercise requires building a Python program to automatically collect detailed statistical data of football players in the English Premier League (EPL) for the 2024-2025 season from the website `fbref.com`. The program will generate two output files:

`results.csv`: Contains data of players whose total minutes played throughout the season (for one or more clubs) exceed 90 minutes, as per the specific requirement of this exercise.

`results1.csv`: Contains data of players who played more than 90 minutes for a specific club during the season. This file retains individual records per team and will be crucial input data for subsequent statistical analysis and aggregation exercises that require detailed data per club appearance of the player.

The following sections will describe the implementation process, including library selection, the data scraping workflow, and the obtained results.

1.2 Library Selection

To perform the task of scraping data from a dynamic website and processing the data efficiently, the following Python libraries were selected:

Selenium: This library was chosen as the primary tool for interacting with the FBRef website because this site heavily uses JavaScript to load and display content, especially complex statistical tables. Unlike the `requests` library (which only fetches the initial static HTML source code sent by the server and cannot execute JavaScript), `selenium` operates by controlling a real web browser (like Chrome or Firefox) automatically. This allows the script to wait for dynamic elements (such as data tables) to be fully loaded and displayed by JavaScript, execute any JavaScript code on the page, and access the final, completely rendered HTML structure (DOM). Using only `requests` would carry a high risk of missing important data tables or receiving incomplete data, as they are generated or populated by JavaScript after the initial page load.

BeautifulSoup4 (bs4): After `selenium` finishes loading the page and retrieves the HTML source code, `BeautifulSoup` is used to parse that HTML. It provides convenient methods for navigating HTML, finding specific tags and attributes (such as `id`, `class`, `data-stat`), and extracting text content easily and efficiently.

Pandas: This is a powerful library for data manipulation and analysis.

Json: The `config.json` file contains important configurations such as the base URL, names of the statistics to retrieve, corresponding data table IDs, `data-stat` attributes, minimum minutes threshold, etc. The `json` library is used to read and parse this configuration file, making the main source code cleaner, more readable, and easier to maintain. Separating the configuration makes it easy to change parameters or update for website structure changes without significant modifications to the Python script.

Re (Regular Expressions): Used to extract the unique player ID from URL paths in the HTML code (`/players/<player_id>/`). This ID is crucial for identifying and aggregating data for the same player when they appear in multiple statistical tables or play for multiple clubs within a season.

Time: Used to introduce small delays (`time.sleep`) between web page access requests, to avoid sending too many consecutive requests to the FBRef server, reducing the risk of IP blocking and demonstrating "polite" web Browse behavior.

Collections.defaultdict: Extremely useful when aggregating data. In cases where a player plays for multiple clubs in the same season (e.g., a mid-season transfer), they will appear multiple times in the data table. `defaultdict` is used to create a data structure that allows summing the minutes played (`Min`) for that player from different rows before applying the minimum minutes filter.

1.3 Web Scraping Process

The main idea of this script is to automate the process of collecting data from a dynamic web source (fbref.com) by combining Selenium to simulate user behavior and BeautifulSoup to parse the HTML structure. The data to be retrieved is determined by an external configuration (`config.json`), allowing for flexible selection of necessary statistics from various tables on the page. A key aspect of data processing is that the program initially identifies each individual statistical record by a pair (unique player ID or name, team name). This is necessary to accurately distinguish the data of the same player when they play for different clubs within the same season (e.g., after a transfer), because on fbref, a player can appear in multiple rows if they change teams. Subsequently, this data is processed in two distinct ways to generate two CSV files:

- **results1.csv**: Directly filtered from the detailed data, retaining records where the player played $>$ 90 minutes for that specific club.
- **results.csv**: Aggregates the total minutes played for each player across the entire season, then filters for players whose total minutes exceed 90, and outputs all records related to that player (including records with less than 90 minutes for a specific club, as long as the total minutes meet the requirement).

The web scraping process is carried out in the following steps:

Step 1: Load Configuration: The program begins by reading the `config.json` file to obtain necessary information such as the base URL for the EPL season on fbref, the list of statistical metrics to collect, IDs of the HTML tables containing the data, the `data-stat` attribute corresponding to each metric, the minimum playing time threshold (`min_minutes`), output CSV filenames, and the column order for the result files.

Step 2: Initialize Selenium: An automated web browser (WebDriver) is initialized using `selenium`.

Step 3: Access and Load Page:

- The program navigates to the URL of the main statistics table (`stats`) for the EPL 2024-2025 season.
- Uses `WebDriverWait` to wait until the main data table (`stats_standard`) appears and is interactable, ensuring that all data loaded by JavaScript is displayed.

Step 4: Initial HTML Parsing: The complete HTML source code of the loaded page is retrieved from `selenium` and passed to `BeautifulSoup` for parsing.

Step 5: Collect Data from Multiple Tables:

- The script is designed to retrieve data from various types of statistical tables on fbref (Standard, Shooting, Passing, Defensive Actions, etc.), as defined in the `STATS_CONFIG` section of the `config.json` file.
- For each type of statistic, the script constructs the corresponding URL (e.g., `.../stats/`, `.../shooting/`, `.../passing/`).
- The script accesses each of these URLs using `selenium`, waits for the respective data table to load, retrieves the HTML, and uses `BeautifulSoup` to parse it.
- Iterates through each row (`<tr>`) in the body (`<tbody>`) of the data table.

Step 6: Extract Player Data: For each row (representing a player in that table), the script performs the following:

- Finds the cell containing the player's name and extracts the name (`Player`) and the unique player ID (`player_id`) from the player profile link using regular expressions (`re`).
- Uses the `safe_get_text` function, `get_nation_code`, and the `data-stat` configuration from `config.json` to extract the value of each requested statistical metric for that player.
- This function ensures that "N/a" is returned if a data cell does not exist or is empty.

Step 7: Temporary Storage: The data for each player from each table is stored in a dictionary, which is then added to a consolidated list (`scraped_data`).

Step 8: Aggregate and Filter Data (2 flows):

- Flow 1: Create data for `results1.csv`
  - Iterate through `scraped_data`.

  - For each record (corresponding to a player at a specific team), check the value of `Playing Time: minutes ('Min')`.
  - If the minutes > `MIN_MINUTES`, reformat the values (using `format_value`) and add this record's dictionary to the `single_team_over_90_list` list.
- Flow 2: Create data for `results.csv`
  - Use `collections.defaultdict` (`player_minutes_aggregate`) to group records by `player_id` (or `player_name` if ID is not available).
  - During the grouping process, sum the minutes played (`Min`) to calculate `total_minutes` for each player.
  - Filter `player_minutes_aggregate`, keeping only players with `total_minutes` > `MIN_MINUTES`.
  - For each eligible player, iterate through *all* original records (entries) of that player (from different teams if applicable).
  - Reformat the values for each record and add it to the `final_player_list_of_dicts` list.

Step 9: Format and Sort:

- Both lists (`single_team_over_90_list` and `final_player_list_of_dicts`) are sorted alphabetically by the player's first name.
- Ensure all columns as per `COLUMN_ORDER` exist, filling with "N/a" if missing (as done in the formatting step in Step 6).

Step 10: Export to CSV:

- Convert the `final_player_list_of_dicts` list into a pandas DataFrame and save it to the `results.csv` file (named as specified in `config.json`).
- Convert the `single_team_over_90_list` list into a pandas DataFrame and save it to the `results1.csv` file.
- Both CSV files are saved without the DataFrame index and use `utf-8-sig` encoding to display special characters correctly.

1.4 Results

The program executes and generates two CSV files:

`results1.csv`: Statistical data was collected for 491 players, each having played over 90 minutes for a club in the English Premier League (EPL) 2024-2025 season.

`results.csv`: Statistical data was collected for 496 players, each having played over 90 minutes in total throughout the 2024-2025 season.

The creation of `results1.csv` ensures that the original data (preliminarily filtered by minutes per team) is available for subsequent processing steps, while `results.csv` directly addresses the requirement of this first exercise.

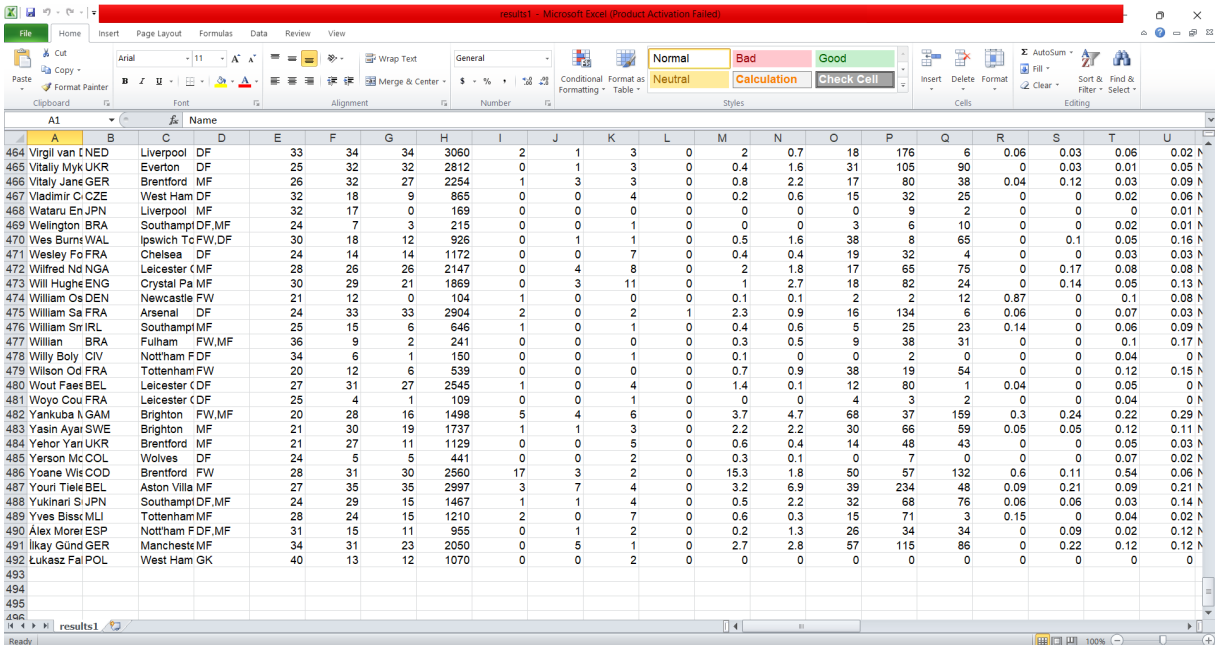


Figure 1.1: Image of the results.csv file

Chapter 2

Statistical Data Analysis and Visualization

2.1 Analysis of Top 3 Highest and Lowest Players per Statistical Indicator

2.1.1 Overview of Problem Requirements

The problem requires processing a data file (`results1.csv` from the previous exercise) containing statistical information of football players. The main objective is to identify and list the 3 players with the highest scores (Top 3) and the 3 players with the lowest scores (Bottom 3) for each statistical indicator present in the data file. The results of this analysis are saved to a text file named `top_3_formatted.txt`.

2.1.2 Overview of Main Code Logic (`top_3_statistical_analysis.py`)

The script uses the `pandas` library to read data from the CSV file. Then, it iterates through each statistical column (excluding basic information columns such as name, nationality, team, position). For each statistical column, the code attempts to convert the data to a numeric type, removing invalid values (`N/a` or non-convertible). If the column can be considered numeric and contains valid data, the code will sort the data to find the top 3 highest and bottom 3 lowest values, along with the corresponding player names and teams. Finally, the results are formatted and written to the `top_3_formatted.txt` file.

2.1.3 Handling 'N/a'

During data processing, `'N/a'` values are considered missing data and are converted to `NaN` via the `na_values=['N/a']` parameter when reading the CSV file with Pandas. Additionally, values that cannot be converted to a numeric type are also coerced to `NaN` using `pd.to_numeric(errors='coerce')`. Instead of replacing missing values with a specific number (like 0), entire rows containing `NaN` in the column to be evaluated are removed before performing the ranking. This is done based on the following principles:

- **Data Correctness:** `NaN` indicates that data is missing or not applicable, for example, a statistic specific to a particular position might not be relevant for a player in a different position. Replacing it with 0 would distort this characteristic.

- **Avoiding Misinterpretation:** Assigning a value of 0 to missing data can lead to misinterpretation – for instance, mistakenly assuming a player scored no goals, when in reality, the data was not recorded.
- **Ensuring Fairness in Ranking:** Considering only players with valid data helps the Top/Bottom 3 results accurately reflect actual performance, avoiding the inaccurate inclusion of individuals with missing data in comparisons.

Thus, retaining `NaN` and removing missing values from the ranking process is a reasonable method, ensuring objectivity and accuracy in player data analysis.

2.1.4 Execution Steps

Initialization and Data Preparation:

The script imports necessary libraries (`pandas`, `os`, `numpy`, `io`). Reads data from the `results.csv` file into a Pandas DataFrame, configured to handle `'N/a'` as `NaN`. Identifies the list of statistical columns (`stats_columns`) to be analyzed by excluding identifier columns ('Name', 'Team', 'Position', 'Nation').

Processing Each Column and Writing Output File:

Opens the `top_3_formatted.txt` file to write the results. The script iterates through each column `col` in `stats_columns`:

- Write Column Header: Writes the current column name to the file.
- Standardize Filter Numeric Data: Uses `pd.to_numeric(errors='coerce')` to cast the column `col` to a numeric type, converting errors to `NaN`.
- Creates a temporary DataFrame `df_for_sort` and removes rows with `NaN` values (`dropna()`) in this column.
- This step ensures that only valid, numeric data is used for ranking.
- Rank and Write Results:
  - If `df_for_sort` contains valid numeric data: Uses `sort_values()` to sort this DataFrame by column `col` in descending order (to find Top 3) and ascending order (to find Bottom 3), then uses `head(3)`.
  - The results (Name, Team, Statistic) and data type information are written to the file.
- Write Separator: Adds a line `===...` to separate results between columns.

Completion: After processing all columns, the `top_3_formatted.txt` file is saved, containing all analysis results.

2.1.5 Results (`top_3_formatted.txt`)

The file `top_3_formatted.txt` contains the analysis results for each statistical indicator. Each indicator is clearly presented with a header, followed by a list of Top 3 and Bottom 3 players.

```
15  ================================================
16
17  --- Playing Time: matches played ---
18
19  Top 3 Players:
20                  Name      Team  Playing Time: matches played
21  Ryan Gravenberch Liverpool                              34
22      Raúl Jiménez    Fulham                              34
23     Mohamed Salah Liverpool                              34
24
25  Bottom 3 Players:
26                  Name           Team  Playing Time: matches played
27      Reiss Nelson        Arsenal                               1
28  Joachim Andersen Crystal Palace                               1
29    Carlos Alcaraz    Southampton                               1
30
31  ================================================
```

Figure 2.1: Example results from the top_3_formatted.txt file.

2.2 Calculating Descriptive Statistics (Median, Mean, Std Dev) for Player Data

2.2.1 Introduction

This exercise requires writing a Python program to perform descriptive statistical calculations on the player data collected in the previous exercise (`results1.csv`). Specifically, the program needs to calculate:

The median value for each statistical indicator across the entire dataset.

The mean value and standard deviation for each statistical indicator, calculated for the entire dataset and separately for each team.

The results are saved to the `results2.csv` file in a specific format, where rows represent either the entirety ('all') or a specific team, and columns represent statistical calculations (Median, Mean, Std) applied to each indicator.

2.2.2 Library Selection

`pandas` library: This is the primary and indispensable library for this task.

`os` library: Used for flexible and OS-independent file path management. `os.path.join` and `os.path.dirname` help accurately determine the location of the input file (`results1.csv`) and output file (`results2.csv`) without hardcoding paths.

2.2.3 Logic and Execution Process

The `calculating_statistics.py` program performs the following steps:

Step 1: Prepare and Read Data: Uses `os` to construct paths to the input file `results1.csv` and output file `results2.csv`. Reads `results.csv` into a pandas DataFrame (`df`). A list `na_values_list` containing various representations of missing values ('N/a', 'n/a', '', ...) is provided to the `na_values` parameter of `pd.read_csv` to ensure these values are correctly identified and converted to pandas' `NaN` (Not a Number), facilitating subsequent calculations.

Step 2: Identify Statistical Columns: Defines a list `exclude_cols` containing columns that are not statistical data to be analyzed (e.g., 'Name', 'Nation', 'Team', 'Position', 'Age'). Creates a list `stats_columns` by filtering columns in the DataFrame, retaining only those not in `exclude_cols`.

Step 3: Calculate Overall Statistics ('all'): Selects columns from `stats_columns` in the DataFrame `df`. Uses the `.agg(['median', 'mean', 'std'])` method to simultaneously calculate the median, mean, and standard deviation for each column in `stats_columns`. The returned result has statistical measures as rows and original columns as columns. Uses `.T` (transpose) to transpose the result, making the original statistic names (`stats_columns`) the row index, and 'median', 'mean', 'std' the column names, saving it into `overall_stats`. This facilitates easier access in the next step.

Step 4: Calculate Grouped Statistics ('Team'): Uses `df.groupby('Team')` to group the DataFrame by values in the 'Team' column. On this GroupBy object, selects the `stats_columns`. Applies `.agg(['median', 'mean', 'std'])` to calculate statistics for each team. The result (`team_stats`) will have a MultiIndex structure for both rows (Team) and columns (Statistic, Metric).

Step 5: Restructure Data for Output:

- Create 'all' row: Initializes an empty DataFrame `all_row` with 'all' as its index. Iterates through each statistical column (`col` in `stats_columns`), creating new columns in `all_row` named in the format `f'Median of {col}'`, `f'Mean of {col}'`, `f'Std of {col}'` and assigns corresponding values from `overall_stats` calculated in Step 3.
- Create 'Team' rows: Initializes a DataFrame `team_results` with team names as its index (obtained from `team_stats.index`). Iterates through each statistical column (`col` in `stats_columns`), creating new columns in `team_results` with similarly formatted names as above. Values are retrieved from `team_stats` by accessing through the multi-level index, e.g., `team_stats[(col, 'median')]` to get the median column for statistic `col`.
- Combine Results: Uses `pd.concat([all_row, team_results])` to concatenate the DataFrame containing the 'all' row and the DataFrame containing individual team rows vertically, forming the final DataFrame `final_results` with the required structure.

Step 6: Save Results: Exports the `final_results` DataFrame to the `results2.csv` file using the `.to_csv()` method. The `index=True` parameter is used (by default) to save the DataFrame's index ('all' or team name) into the first column of the CSV file.

2.2.4 Results

The program ran successfully and generated the `results2.csv` file. The `results2.csv` file contains the aggregated statistical results in the required format:

- Rows: The first row has an index 'all', representing statistics for all players. Subsequent rows have indices corresponding to team names, representing statistics calculated separately for players of that team.
- Columns: Columns are named following the pattern "Metric of Statistic", e.g., "Median of Performance: goals", "Mean of Performance: goals", "Std of Performance: goals", "Median of Performance: assists", etc., including all indicators in `stats_columns`.
- Values: Cells contain the corresponding calculated median, mean, or standard deviation values. NaN values (if any, e.g., standard deviation for a group with only 1 player) will be represented as empty cells in the CSV file.

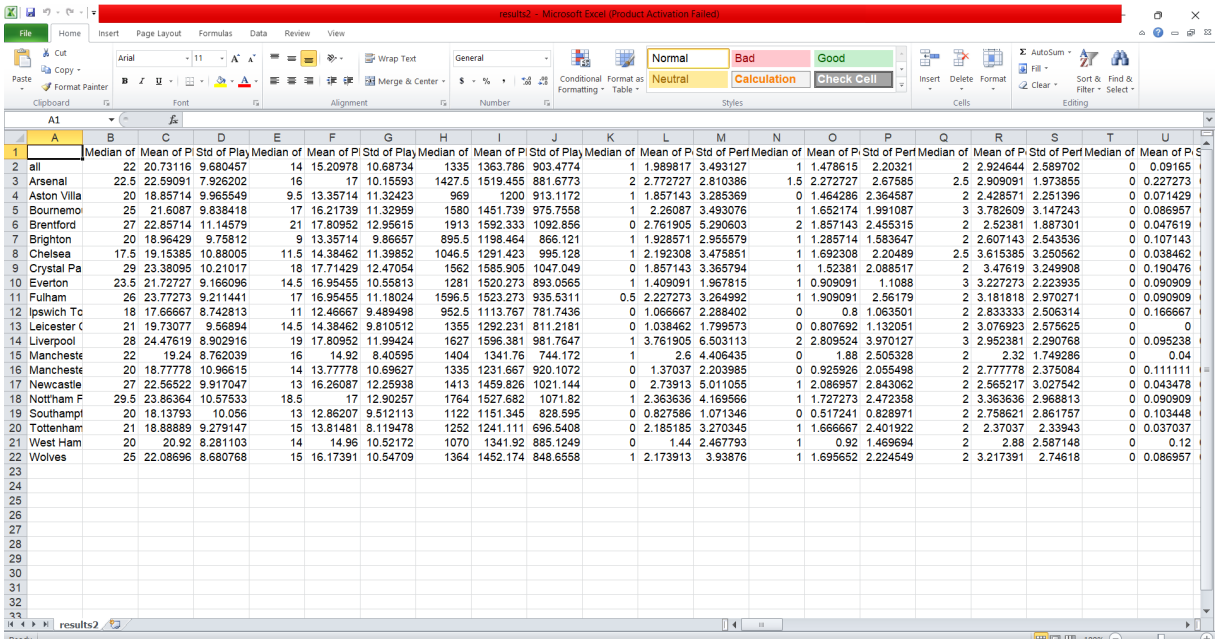


Figure 2.2: Example data from the results2.csv file.

2.3 Visualizing Player Statistical Distribution using Histograms

2.3.1 Objectives

The main objective of the `plot_stat_histograms.py` script is to visualize and analyze the distribution of a set of important statistical indicators for football players. Specifically, the script aims to:

- Generate histograms for each selected statistical indicator (`stats_to_plot`) to show its distribution across the *entire league* (all players).

- Generate separate histograms for each *football team*, showing the distribution of each statistical indicator within that team.
- Save these charts as image files (PNG) for easy review, sharing, and later analysis.
- Provide a visual understanding of the distribution shape (e.g., symmetric, left-skewed, right-skewed), central tendency, and dispersion of statistical indicators, both at the league level and team level.

The specific statistical indicators analyzed in this script include: `'Performance: goals'`, `'Performance: assists'`, `'Standard: shoots on target percentage (SoT%)'`, `'Blocks: Int'`, `'Performance: Recov'`, `'Challenges: Att'`.

2.3.2 Library Selection

The script uses several common Python libraries for data processing and visualization:

- `pandas`: The core library for reading data from CSV files (`results1.csv`), storing data in DataFrames, and performing necessary data manipulation, filtering, and selection for plotting. It also handles missing values (NaN).
- `matplotlib.pyplot`: The foundational library for creating charts in Python. It is used to create figures, axes, set titles, labels, and manage chart layouts, especially when plotting multiple subplots for teams.
- `seaborn`: A library built on `matplotlib`, providing a higher-level interface for drawing attractive and informative statistical graphics. Specifically, `seaborn.histplot` is used to draw histograms, including a kernel density estimate (KDE) line that helps smooth and better visualize the distribution shape. `seaborn.set_theme` is used to set the overall aesthetic style for the charts.
- `numpy`: A scientific computing library, used in the `calculate_fd_bins` function to perform necessary calculations (like cube root `np.cbrt`, base-2 logarithm `np.log2`) for determining the optimal number of bins according to the Freedman-Diaconis and Sturges rules.
- `os`: A library providing functions for interacting with the operating system, mainly used for managing file paths (getting current directory path, joining paths), creating storage directories (`stat_histograms`), and ensuring valid paths.
- `math`: A basic math library, used for functions like `math.ceil` to round up when calculating the number of rows/columns for subplots and the number of bins.

2.3.3 Execution Steps

The script performs the following main steps:

Configuration Setup

Defines constants such as the output directory (`OUTPUT_DIR`), the column name containing team information (`TEAM_COL`), the maximum number of teams per plot (`TEAMS_PER_PLOT`), plot style (`PLOT_STYLE`), and the maximum bin limit (`MAX_FD_BINS`).

Defining Helper Functions

- `ensure_dir`: Ensures the output directory exists.
- `sanitize_filename`: Cleans the statistical indicator name to create a valid filename.
- `calculate_fd_bins`: Calculates the optimal number of bins for a histogram based on the Freedman-Diaconis rule (preferred) or Sturges' rule (fallback), helping the chart better reflect the actual data structure and avoid arbitrary bin selection. There is a maximum bin limit to prevent excessive detail.

Defining Plotting Functions

- `plot_overall_hist`:
  - Takes a DataFrame, statistical indicator name, and output directory as input.
  - Filters data for the statistical indicator, converts to numeric type, and removes missing values (`NaN`).
  - Calculates the optimal number of bins using `calculate_fd_bins`.
  - Uses `seaborn.histplot` to draw the overall distribution histogram, with KDE enabled.
  - Sets title, axis labels, and saves the chart to a PNG file with a standardized name.
- `plot_team_hist`:
  - Takes a DataFrame, indicator name, output directory, team column name, and number of teams per figure as input.
  - Gets a list of unique teams and divides them into smaller groups (according to `TEAMS_PER_PLOT`).
  - Iterates through each group of teams:
    - Creates a new figure with subplots (a grid of smaller charts).
    - Filters data to include only teams in the current group.
    - Calculates the optimal number of bins *based on the data range of teams in this group*. This helps histograms within the same figure share a common way of dividing value ranges.
    - Determines common x-axis limits (`x_min`, `x_max`) for all subplots in the current figure for easier comparison.
    - Draws a histogram for each team on a separate subplot using `seaborn.histplot`, using the calculated number of bins and bin range (`binrange`).
    - Sets the subplot title as the team name, hides axis labels to avoid clutter, and applies uniform x-axis limits.
    - Hides any unused subplots.
    - Sets the main title for the figure (including indicator name and team group).
    - Saves the figure containing histograms for the group of teams to a PNG file.

Main Execution Block (`if __name__ == '__main__':`)

- Sets the theme for `seaborn`.
- Defines the list of indicators to plot (`stats_to_plot`).
- Determines the path and reads the `results.csv` file into a DataFrame.
- Ensures the output directory exists.
- Iterates through each indicator in `stats_to_plot`:
  - Calls `plot_overall_hist` to plot the overall histogram.
  - Calls `plot_team_hist` to plot histograms for each team (grouped).

2.3.4 Visual Results

The output of the script is a set of PNG image files saved in the `stat_histograms` directory. Specifically:

- For each statistical indicator in `stats_to_plot`, there will be *one file* containing a histogram showing the overall distribution of that indicator across the entire league (e.g., `hist_overall_Standard_shoots_on_target_percentage_SoT.png`).

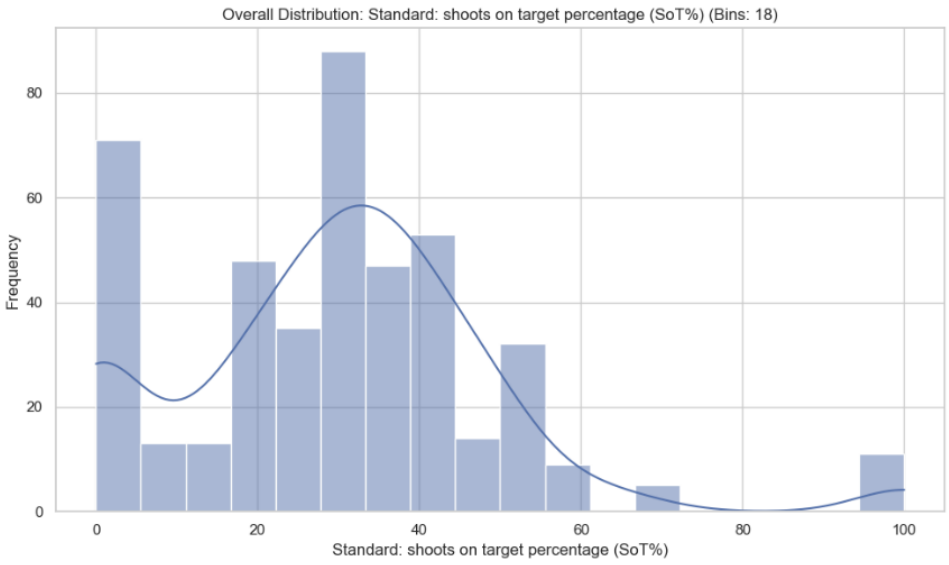


Figure 2.3: Example of an overall histogram (Source: Excerpt from original document).

- For each statistical indicator, there will be *one or more files* of histograms showing its distribution per team. Teams are grouped (default 20 teams/figure) to avoid creating too many files or overly large images (e.g., `hist_teams_Standard_shoots_on_target_percentage_SoT_group_1.png`,...).

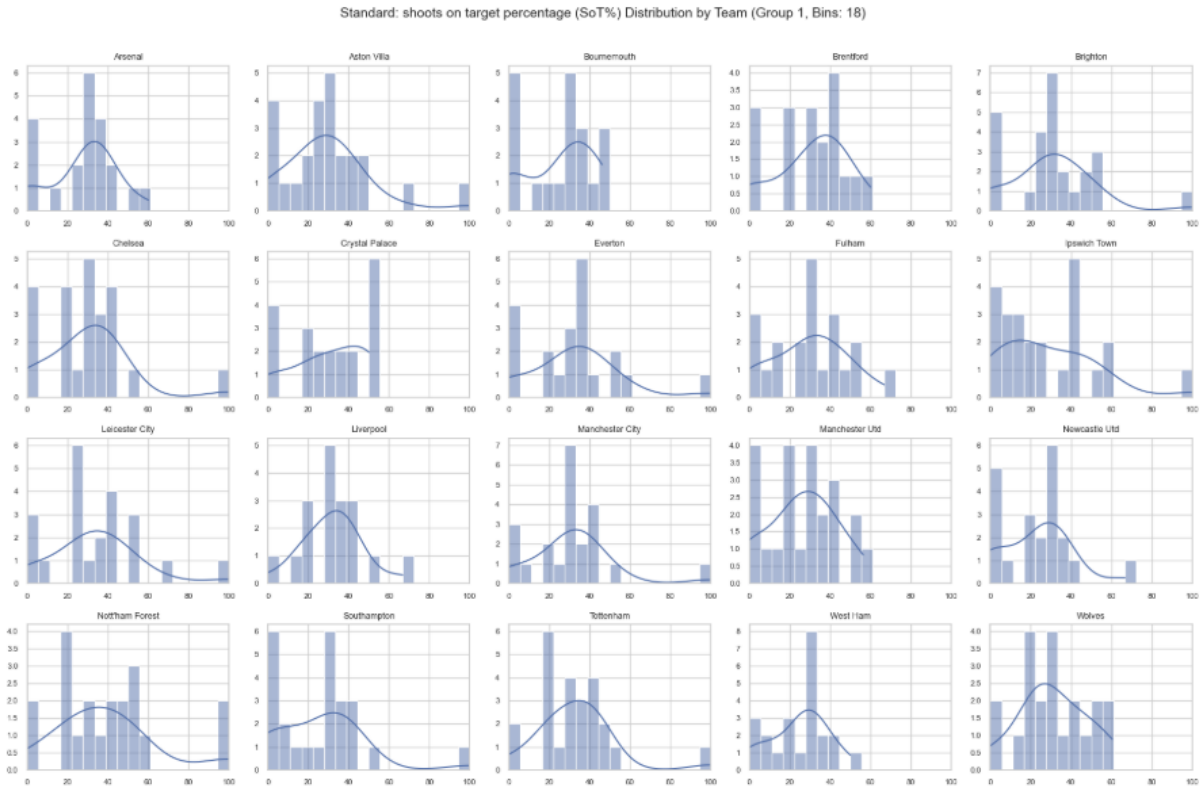


Figure 2.4: Example of histograms by team group (Source: Excerpt from original document).

Each of these files contains a grid of small charts, with each small chart corresponding to a team. These charts allow viewers to visually observe the distribution shape (symmetric, skewed, multimodal), value range, and frequency of different values for each indicator, both overall and within each team.

2.3.5 Limitations

In the process of analyzing player statistical data, histograms are used to visualize the distribution of each indicator (e.g., number of goals, minutes played, assists, etc.). This is an effective tool for quickly observing common trends, detecting outliers, and comparing differences between players or teams. However, the use of histograms also has some limitations that need to be noted to avoid drawing inaccurate or biased conclusions:

- **Dependency on Data Quality:** The visual results are entirely dependent on the completeness and accuracy of the data in the `results.csv` file. Missing values (`NaN`) were removed during processing to ensure the accuracy of the charts. However, if the proportion of missing data is large or not randomly distributed, this can distort the distribution shape and lead to biased judgments about overall trends.
- **Sensitivity to the Number of Bins:** Although the number of bins is determined based on the Freedman-Diaconis rule to optimize the level of detail in the chart, the histogram's shape can still change if the number of bins is altered. Furthermore, setting a maximum limit for the number of bins (via the `MAX_FD_BINS` variable) is subjective and can affect the chart's resolution, especially in distributions with many dispersed values.

2.4 Performance Analysis of Premier League Teams 2024–2025

2.4.1 Introduction

To evaluate the performance of teams in the 2024–2025 season, the analysis process is designed with two main objectives: identifying the leading team for each statistical indicator and finding the team with the best overall performance. Data processing is carried out using two main files:

- `highest_stats_team.py`: The Python source code file that handles the entire analysis process, from reading data, classifying indicators (good, bad, ignore), determining the leading team for each criterion, to aggregating results to identify the best-performing team of the season.
- `config.json`: A configuration file that allows flexible customization of parameters such as data paths, handling of missing values, and the indicator classification system. Separating the configuration helps adjust analysis criteria without modifying the source code.

This approach optimizes reusability, maintainability, and scalability for different seasons or datasets.

2.4.2 Main Idea

Identify the team with the highest score for each indicator: The script classifies indicators into three groups (good, bad, ignore) based on the configuration file (`config.json`). For each indicator, the code converts the data to a numeric type, then finds the optimal value (highest for good indicators, lowest for bad indicators) and identifies which team achieved that value. For example, the team with the most goals, the team with the highest pass completion rate, or the team with the fewest red cards.

Identify the team with the best overall performance: The code counts the number of times each team leads in different indicators, separately for good and bad indicators. Then, it calculates a combined total score for each team by summing the number of times they led in both types of indicators. The team with the highest total number of leading instances is considered the team with the best overall performance in the season.

2.4.3 Execution Process

The analysis process is controlled by the `highest_stats_team.py` script, with the following main steps:

Initialization Load Configuration

Loads configuration from `config.json` using the `load_config` function. This function reads the JSON file and creates sets of keywords `ignore_stats_set`, `bad_stats_set`, `good_stats_set` for classifying indicators.

Define Path Read CSV Data

Constructs the absolute path to the `results2.csv` file (located in `../calculating_statistics/`). Reads data from CSV into a DataFrame `dataframe` using `pandas`, utilizing `na_values` from the config.

Preprocess DataFrame

Identifies the team name column (defaults to the first column). Standardizes the team name column: converts to lowercase, removes rows where the team name is "all".

Determine Leading Team for Each Valid Statistic (Function `analyze_performance_by_stat_type`)

Step 1: Identify and Filter Indicators for Analysis:

- Create a list of columns prefixed with "Mean of " (excluding the team column).
- For each potential column:
  - Extract the actual indicator name (remove "Mean of ").
  - Classify the indicator (using `classify_stat`) as 'ignore', 'bad', 'good', or 'uncategorized' based on the loaded keyword sets.
  - If not 'ignore', convert the column to numeric. Only retain indicators with at least one valid numeric value.

Step 2: Analyze Each Filtered Indicator:

- For each indicator selected for analysis:
  - Convert the indicator column to numeric and remove invalid values (`NaN`).
  - Find the best value:
    - If type is 'good' or 'uncategorized': find the *maximum* value.
    - If type is 'bad': find the *minimum* value.
  - If the best value is not found (e.g., empty column), log an error.
  - Otherwise:
    - Identify the team(s) achieving this best value.
    - Save the name(s) of the leading team(s), the best value (in its original format for display), and the indicator type into a results dictionary.

Step 3: Return: Dictionary containing analysis results for each indicator.

Display Detailed Analysis Results

Sorts results by indicator name. Iterates through each analyzed indicator, formats the value (using `format_value` to handle integers, floats, "N/A"), and prints: indicator name, type (GOOD, BAD), leading team, and achieved value.

Calculate and Display Consolidated Ranking Table

Step 1: Count Leading Instances:

- For each indicator with valid results in `analysis_results`:
  - If a team leads a 'good' indicator, increment its score in `good_lead_counts`.
  - If a team leads a 'bad' indicator, increment its score in `bad_lead_counts`.

Step 2: Create Consolidated Score:

- Get a list of all unique teams from the DataFrame.
- Initialize a score of 0 for all teams.
- Sum the total 'good' and 'bad' leading instances to get a consolidated score for each team.

Step 3: Sort and Display:

- Sort teams: descending by consolidated score, then ascending by team name (if scores are tied).
- Print the ranking table with rank, team name, and total leading instances.

2.4.4 Results

When the `highest_stats_team.py` script is executed successfully, it will produce two main blocks of information printed to the console (command line interface). These results provide a detailed view of each team's performance for every analyzed statistical indicator, and a consolidated ranking table of teams based on the number of times they led those indicators.

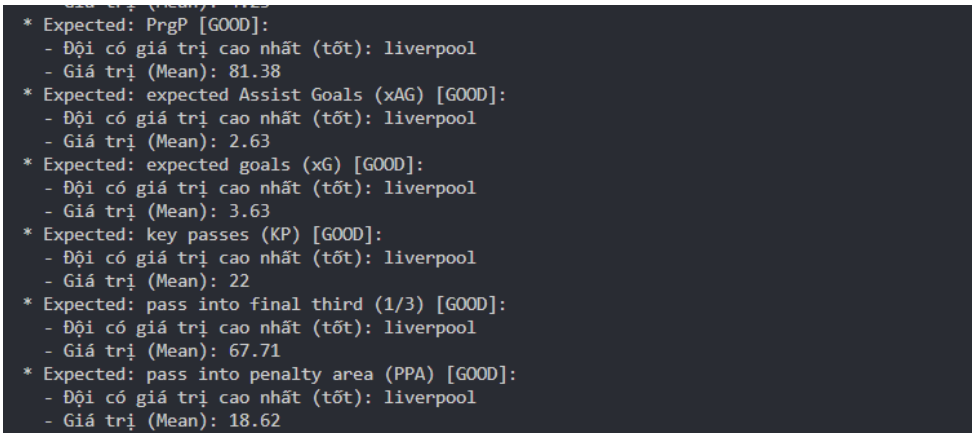


Figure 2.5: Example of detailed analysis results by indicator.

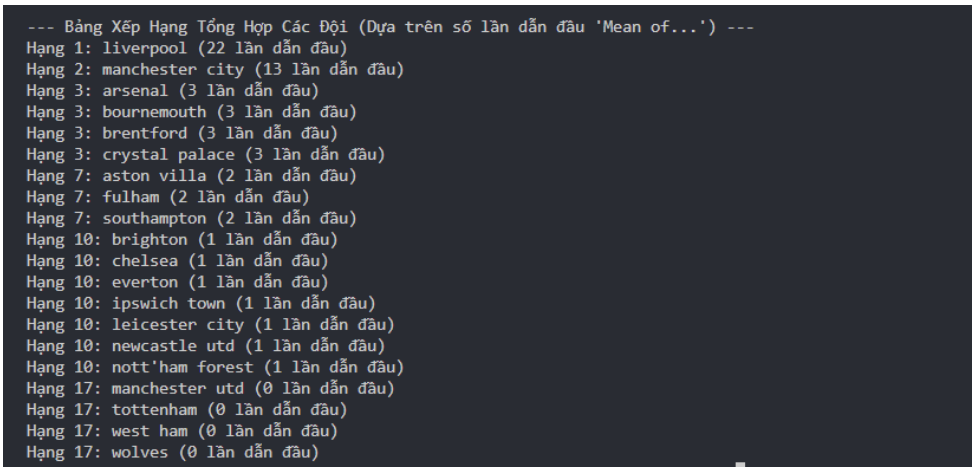


Figure 2.6: Example of consolidated team ranking table.

The statistical data indicates that Liverpool is an outstanding and comprehensive team, not only excelling in a few aspects but also maintaining impressive form across most crucial indicators of modern football. They possess explosive attacking power, superior ball control, and a solid defensive system – all contributing to a smooth and effective playing machine. In reality, they convincingly won the Premier League title for the 2024–2025 season *with 4 matches to spare*, accumulating *82 points* – a safe distance ahead of the runner-up, Arsenal. This is clear evidence that when a football team operates correctly in terms of both expertise and strategy, its dominance will be clearly reflected in the statistical numbers and on-field results.

Chapter 3

Player Clustering and Dimensionality Reduction with PCA

3.1 Football Player Clustering using K-means Algorithm

3.1.1 Introduction

Player data analysis plays a crucial role in modern sports, helping coaches, analysts, and managers make informed decisions. One of the common techniques used is the K-means clustering algorithm, which aims to divide a set of players into distinct groups (clusters) based on the similarity of their statistical indicators. This section will present how the K-means algorithm is applied, particularly focusing on determining the optimal number of clusters and commenting on the results obtained from running the source code file `player_clustering_kmeans.py`.

3.1.2 Selecting the Range of k for Experimentation

Selecting the range of k for experimentation in the player clustering problem (500 players, 78 indicators) needs to be based on theoretical factors, data characteristics, and practical context in football.

Reasons for Choosing kmin = 3

- **Minimum requirement for meaningful clustering:**
  - k=1: does not create clusters (all data belongs to the same group), no analytical value.
  - k=2: only divides data into two groups, often too simplistic for data with around 78 indicators (only providing a crude distinction like "attack" and "defense").
  - k=3: a reasonable starting point to reflect basic roles in football, e.g., goalkeeper, defender, attacker.
- **Suitability for sample size:**
  - With 500 players, k=3 creates clusters averaging $500/3 \approx 167$ players/cluster, large enough to be statistically significant.

Conclusion: kmin=3 is a reasonable starting point.

Reasons for Choosing kmax = 15

- **Based on sample size (around 500 players):**
  - Each cluster should have at least 20–50 samples to ensure statistical significance:
    - If each cluster has a minimum of 20 players: $k = 500/20 = 25$
    - If each cluster has a minimum of 50 players: $k = 500/50 = 10$.
  - To ensure clusters are not too small, kmax should be in the range of 10–25. However, context needs further consideration.
- **Football context:**
  - The number of roles/playing styles in football rarely exceeds 15–20 (e.g., goalkeeper, center-back, full-back, defensive midfielder, attacking midfielder, winger, center-forward, deep-lying forward, etc.).
  - k=15 is sufficient to cover detailed roles without exceeding practical limits.
- **Practical experience:**
  - In similar clustering problems, kmax=15 is often sufficient to cover elbow points (typically occurring in the range k=4 to k=10), avoiding wasted computational resources.

Conclusion: kmax=15 is a reasonable threshold. The range of k from 3 to 15 is reasonable: kmin=3 creates basic clusters (167 players/cluster), kmax=15 covers detailed roles (33 players/cluster), suitable for the data, football context, and elbow points typically found between k=4 and k=10.

3.1.3 Determining the Optimal Number of Clusters (K) and Execution Process

Main Idea: To determine the optimal number of clusters (K) for grouping players, I applied the Elbow method. This process begins by experimenting with the K-Means algorithm using a range of potential K values, specifically from 3 to 15 clusters. For each K value, the WCSS (Within-Cluster Sum of Squares - the sum of squared distances from each data point to its cluster center) is calculated. WCSS is a measure of cluster cohesion; a lower WCSS value indicates that data points within the same cluster are closer to each other. Subsequently, these WCSS values are visualized on a line graph, with the number of clusters (K) on the x-axis and the corresponding WCSS on the y-axis. The "elbow point" on the graph, where increasing the number of clusters K no longer leads to a significant decrease in WCSS, is identified as the optimal K value. Based on the analysis of the Elbow plot, a suitable K value was chosen. Finally, the K-Means algorithm is re-executed with this optimal K number of clusters to divide the players into corresponding groups, whereby each player is assigned to a cluster based on statistical similarity.

Configuration and Initialization

- **Import libraries:** Loads necessary libraries such as `pandas`, `numpy`, `os`, `sklearn.cluster.KMe` (for the K-Means algorithm), `sklearn.preprocessing.StandardScaler` (for data standardization), `sklearn.impute.SimpleImputer` (for handling missing values), `matplotlib.pyplot`, and `seaborn` (for plotting).
- **Define configuration constants:**
  - `PLAYER_COL`: Name of the column containing player names (default is 'Name').
  - `CSV_RELATIVE_PATH`: Relative path to the CSV file containing player data (default is `../problem_1/results1.csv`).
  - `script_dir`: Gets the current directory path of the script.
  - `output_folder_name`: Name of the folder to save analysis results (default is `kmeans_analysis_results`).
  - `OUTPUT_DIR`: Creates an absolute path to the directory for saving results.
- **Define helper function** `ensure_dir(path: str)`**:** This function ensures that a directory exists at the provided path; if not, it creates the directory.

Main function `cluster_players_kmeans(csv_path: str, player_col: str, max_k: int = 10)`

- **Load Data:**
  - Determine the absolute path to the CSV file.
  - Read data from the CSV file into a `pandas` DataFrame.
- **Prepare Data:**
  - Select numeric columns: Identify and select all columns in the DataFrame with numeric data types to be used as features for clustering (also excluding the 'Age' column to make clusters more clearly reflect similarities in skills, playing style, or tactical roles, rather than being influenced by the player's career stage).
  - Handle missing values (NaN): Use `SimpleImputer` to replace any missing values in the selected numeric columns with the mean of the respective column.
  - Standardize Data: Use `StandardScaler` to standardize the numeric features. This ensures that all features are on the same scale, which is crucial for distance-based algorithms like K-Means.
- **Determine the optimal number of clusters (K) using the Elbow method:**
  - Initialize a list `wcss` (Within-Cluster Sum of Squares).
  - Iterate over a range of possible K values (from 3 to 15).
  - For each K value:
    - Create a K-Means model (`KMeans`) with K clusters, initialization method `'k-means++'`, `n_init='auto'` (to automatically choose the number of runs with different centroid seeds), and `random_state=42` (to ensure reproducible results).

    - Train the K-Means model on the standardized data.
    - Get the `inertia_` (WCSS) value of the model and add it to the `wcss` list.
  - Plot the Elbow graph:
    - Call the `ensure_dir` function to create the output directory if it doesn't exist.
    - Create a line plot (`plot`) with the number of clusters (K) on the x-axis and WCSS on the y-axis.
    - Set the title, axis labels, and display a grid for the plot.
    - Save the Elbow plot as a PNG file in the `OUTPUT_DIR` directory.
  - Print a message guiding the user to view the plot and select the optimal K value (the "elbow" point on the plot, where WCSS begins to decrease more slowly).

- **Get optimal K input from the user:**
  - Prompt the user to enter the optimal number of clusters K based on the Elbow plot.
  - The loop continues until the user inputs a valid integer within the allowed range.

- **Apply K-means with optimal K:**
  - After the user selects K, create a final K-Means model with the chosen `optimal_k` number of clusters.
  - Train this final K-Means model on the standardized data.
  - Get the cluster labels assigned to each data point (each player).

- **Add cluster labels to the original DataFrame:**
  - Create a copy of the original DataFrame.
  - Add a new column named `'Cluster'` to this copy, containing the assigned cluster labels.

- **Analyze and Save Results:**
  - Analyze cluster centroids:
    - Calculate the centroid coordinates of each cluster on the standardized scale (`final_kmeans.cluster_centers_`).
    - Transform these centroid coordinates back to the original scale using `scaler.inverse_`
    - Create a new DataFrame (`centroid_df`) containing the mean values of numeric features for each cluster (on the original scale).
    - Print this centroid DataFrame (rounded to 2 decimal places).
    - Save the centroid DataFrame to a CSV file in the `OUTPUT_DIR` directory.
  - Count players in each cluster: Print the number of players belonging to each cluster.
  - Sort and Save Clustered DataFrame:
    - Sort the DataFrame with assigned cluster labels by the `'Cluster'` column.

    - Save this sorted DataFrame to another CSV file in the `OUTPUT_DIR` directory.

- **Return** the clustered DataFrame, the `scaler` object, the standardized features DataFrame, and the list of numeric columns.

Main execution block (`if __name__ == '__main__':`)

- **Call clustering function:** Call the `cluster_players_kmeans` function with configured parameters (`CSV_RELATIVE_PATH`, `PLAYER_COL`).
- **Store returned results** (clustered DataFrame, scaler, etc.).
- **Display example players from each cluster (if clustering is successful):**
  - Check if `clustered_data` exists.
  - If it exists and the `numeric_cols_list` is not empty:
    - Iterate through each unique cluster ID (sorted).
    - Print the header for that cluster.
    - Get the first 5 players (`head(5)`) from the current cluster.
    - Select columns to display: player name (`PLAYER_COL`), at most the first 5 numeric features, and the `'Cluster'` column.
    - Print the information for these sample players.

3.1.4 K-means Clustering Results

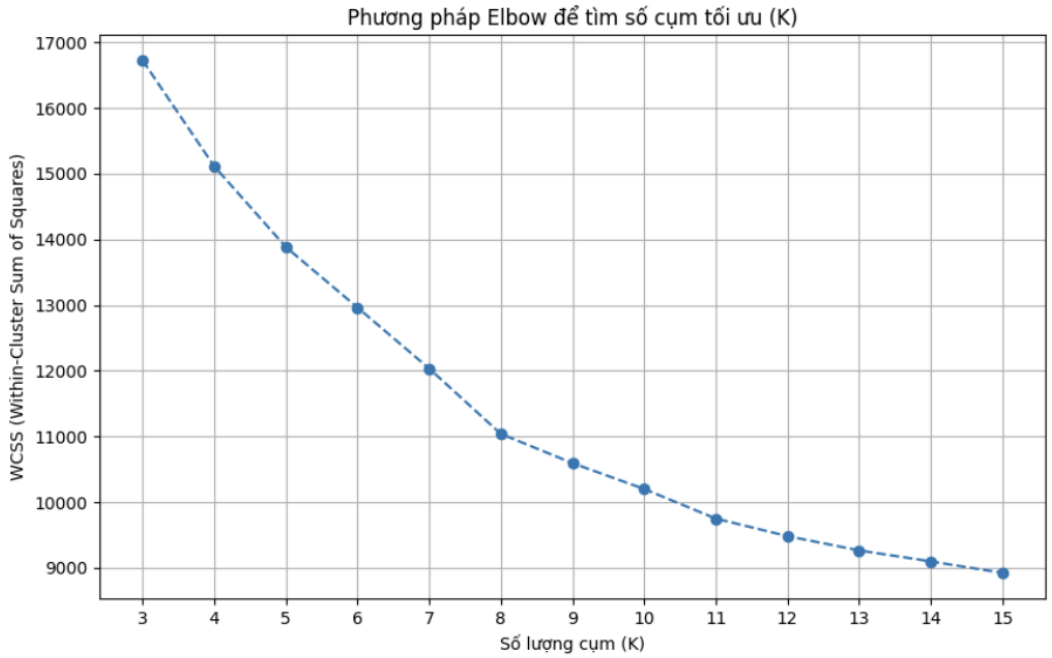


Figure 3.1: Elbow method for finding the optimal number of clusters (K).

The Elbow plot shows a sharp decrease in WCSS from k = 3 (17000) to k = 6 (12000), then a slower decrease from k = 6 to k = 15 (down to 9000), marking k = 6 as the

"elbow" point. Choosing k = 6 is reasonable as it balances clustering quality (WCSS has significantly decreased) and model complexity, avoiding the creation of unnecessary additional clusters when WCSS subsequently decreases insignificantly. The number of players in each cluster will be printed to the screen.

```
Số lượng cầu thủ trong mỗi cụm:
Cluster
0    148
1     31
2     97
3    126
4     50
5     39
Name: count, dtype: int64
```

Figure 3.2: Number of players in each cluster.

The mean values of each indicator within each cluster are saved to the `cluster_centroids.csv` file, while the detailed clustering results for each player are saved to the `player_clusters_sorted.cs` file. Based on the data from these two files, I have analyzed the prominent characteristics of each cluster as follows:

- **Cluster 0 (148 players, 30.1%): Reserve players with defensive roles or Goalkeepers.**
  - Characteristics: Low playing time, almost no attacking contribution (goals, assists, shots, attacking dribbles are all at the lowest levels among clusters).
  - Primarily active in their own half (`Touches: Def Pen` is high), good short/medium pass completion rates (suitable for goal kicks or safe ball circulation).
  - Quantity: The largest cluster, suggesting this is a common group including players with limited playing time, deep-lying defensive specialists, or highly likely, goalkeepers from various teams.
- **Cluster 1 (31 players, 6.3%): Target Forwards/Poachers (Penalty Box Strikers).**
  - Characteristics: Absolutely the highest in goals scored (`Performance: goals`) and expected goals (`Expected: expected goals (xG)`).
  - The target for attacking passes (`Progression: PrgR` is highest) and most active in the opponent's penalty area (`Touches: Att Pen` is highest).
  - Low ability to progress the ball via passing (`Progression: PrgP`), reinforcing their role as the final finisher.
  - Quantity: The smallest cluster, reflecting the specialization of high-efficiency strikers with such a clear role.
- **Cluster 2 (97 players, 19.8%): Versatile Central Midfielders, balanced in attack and defense.**
  - Characteristics: High playing time. Leads in important defensive indicators (e.g., `Tackles: Tkl` highest, `Performance: Recov` highest).

- Effective ball progression from the second line (`Progression: PrgP` high, many `Expected: pass into final third (1/3)`) but not the primary playmaker like Cluster 4 (significantly lower assists, key passes).
- Quantity: A large cluster, indicating the importance and prevalence of "backbone" midfielders who control the midfield and link play.

- **Cluster 3 (126 players, 25.7%): Reserve players with an attacking inclination, game-changers.**
  - Characteristics: Lowest playing time but very notable attacking performance per 90 minutes (especially `SCA: SCA90` - shot-creating actions/90 minutes - is very high).
  - Low defensive contribution, suggesting a tendency to be used to freshen up the attack.
  - Quantity: The second-largest cluster, reflecting that teams often have multiple reserve options to change the game's dynamic with attack-capable players.

- **Cluster 4 (50 players, 10.2%): Key Attacking Midfielders/Playmaking Wingers.**
  - Characteristics: Absolute leader in most creative and playmaking indicators (`Performance: assists`, `Expected: key passes (KP)`, `SCA: SCA`, `GCA: GCA` are all highest).
  - Superior dribbling ability (`Progression: PrgC`, `Take-Ons: Att`) and attacking progression passes (`Progression: PrgP`).
  - Also possesses good goal-scoring ability.
  - Quantity: A moderate number of players, reflecting the important role of attacking "playmakers," who are often key figures but not always numerous in a squad.

- **Cluster 5 (39 players, 7.9%): Key Center-Backs, capable of ball progression from the defense.**
  - Characteristics: Highest playing time. Dominates specialized defensive indicators (`Blocks: Blocks`, `Blocks: Int`, `Aerial Duels: Won` are all highest).
  - Highest pass completion rate (`Total: Pass completion (Cmp%)`) and a very large volume of progressive passes from their own half (`Progression: PrgP`, `Expected: pass into final third (1/3)` high), but extremely low direct attacking contribution (goals, assists).
  - Quantity: A small cluster, consistent with the typical number of starting center-backs in a team, especially those with a comprehensive skill set in both defense and ball progression.

3.2 Visualizing Player Clustering Results using Principal Component Analysis (PCA)

3.2.1 Introduction

This exercise aims to visualize the results of player clustering using Principal Component Analysis (PCA). The input data is taken from the previous clustering results, which were performed by the `player_clustering_kmeans.py` module based on the `results.csv` dataset. First, player features were standardized and then their dimensionality was reduced from multiple dimensions to two principal components using the PCA technique. This process allows for a visual representation of the data on a two-dimensional plane while retaining most of the important information about the differences between clusters. The players are then displayed on a 2D scatter plot, with each data point colored according to the cluster label previously assigned by the K-Means algorithm. The final output includes the visualized plot saved as an image, along with the dimensionality-reduced dataset and accompanying cluster labels, facilitating convenient analysis, comparison, and presentation later on.

3.2.2 Execution Process (Based on the script `pca_visual_kmeans.py`)

The `pca_visual_kmeans.py` script performs the following main steps:

Step 1: Load and process K-Means clustering results:

- The script calls the `cluster_players_kmeans` function from the `player_clustering_k` module.
- This function is responsible for reading player data from `CSV_RELATIVE_PATH` (pointing to `../problem_1/results.csv`), performing K-Means clustering (including determining the `optimal_k` number of clusters and standardizing data).
- The returned results include the DataFrame `df_clustered_sorted` (containing player information and assigned `Cluster` labels) and `features_scaled_df` (DataFrame of standardized numeric features, which is the input for PCA).

Step 2: Apply Principal Component Analysis (PCA):

- A PCA object from the `sklearn.decomposition` library is initialized with `n_components=PCA_COMPONENTS` (set to 2) to reduce the data to 2 dimensions.
- The PCA model is fitted and applied (transformed) on `features_scaled_df`.
- The result is a NumPy array containing the coordinates of each player on the 2 new principal components (PC1 and PC2).
- The proportion of variance from the original data explained by these 2 principal components is calculated and printed to the screen.

Step 3: Visualize 2D Clustering Results:

- The two principal components (PC1, PC2) are used as the X and Y axes for the plot.

- Using the `seaborn` and `matplotlib` libraries, a scatter plot is drawn, where each point represents a player.
- Points are colored based on the `Cluster` column from `df_clustered_sorted`, helping to distinguish different player groups.
- The plot is given a title, axis labels, a legend, and a grid for readability.
- This visualization plot is saved to an image file named `kmeans_pca_{optimal_k}_clus` in the `pca_kmeans_viz_results` directory.

Step 4: Save Final Data:

- The PC1, PC2 coordinates are merged into the `df_clustered_sorted` DataFrame to form `df_pca_clustered`.
- This DataFrame, containing player information, K-Means cluster labels, and the two PCA principal components, is saved to a CSV file named `player_clusters_with_pca_{optimal_k}.csv` in the same results directory.

3.2.3 Results and Remarks

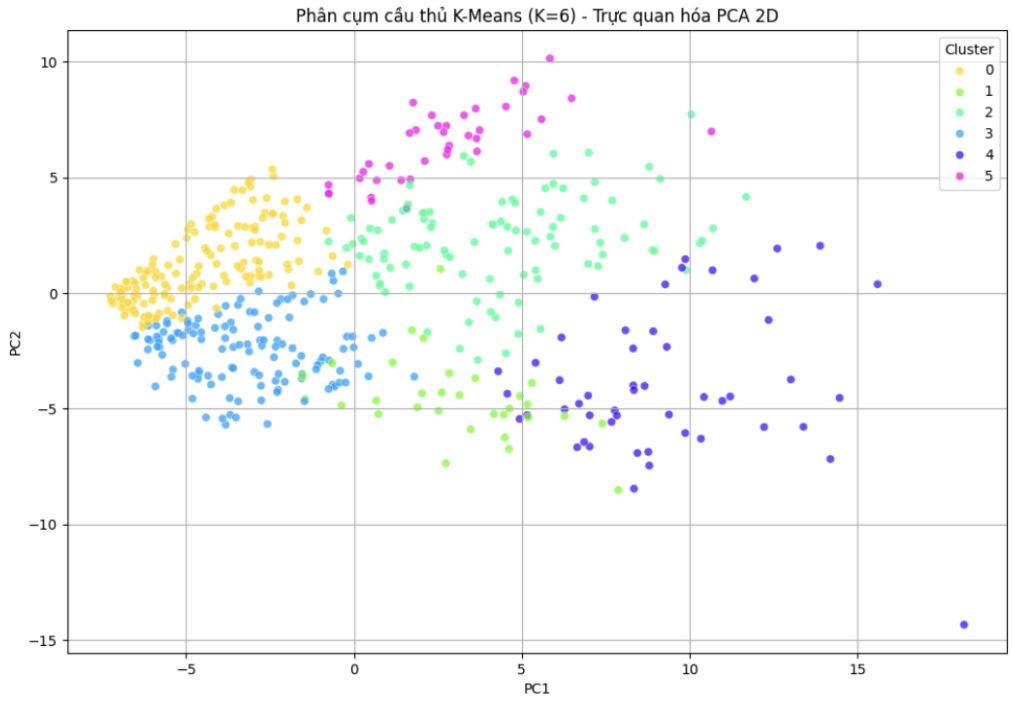


Figure 3.3: K-Means Player Clustering (K=6) - 2D PCA Visualization.

Remarks on the PCA plot: Proportion of variance explained by the 2 principal components (PC1 and PC2): 0.6499 (i.e., 64.99%). This indicates that the first two principal components explain approximately 65% of the total variance in the data, with the remaining 35% of information residing in other components. The plot shows that the clusters are quite clearly divided, but there are some overlapping regions between clusters, especially

in the central area (near the intersection of PC1 = 0, PC2 = 0). This suggests that some players have relatively similar characteristics across clusters, possibly because their statistical parameters (e.g., number of goals, assists, or defensive indicators) do not differ significantly.

- Cluster 0 (yellow) is mainly concentrated in the upper-left corner of the plot (negative PC1, positive PC2), possibly representing a group of players with distinct characteristics (e.g., defenders or goalkeepers, based on CSV data, as they tend not to score and are heavily involved in defense).
- Cluster 1 (light green) is located in the lower-right (positive PC1, negative PC2), possibly a group of forwards or attacking players who score many goals (like Alexander Isak, Yoane Wissa in this cluster).
- Cluster 2 (blue) is widely distributed in the central and right areas, possibly midfielders or full-backs with a balanced role between attack and defense (like Alexis Mac Allister, Andrew Robertson).
- Cluster 3 (light blue) is concentrated in the lower-left corner (negative PC1, negative PC2), possibly young or reserve players with little playing time and minimal contribution to attacking statistics (like Abdul Fatawu Issahaku, Adam Lallana).
- Cluster 4 (purple) is located on the right side (positive PC1), possibly attacking players or wingers with many dribbles and attacking involvement (like Alejandro Garnacho, Adama Traoré).
- Cluster 5 (pink) is scattered but concentrated in the region of large positive PC2, possibly center-backs or defensive players with high defensive stats (like Virgil van Dijk, Wout Faes).

3.2.4 Conclusion

The `pca_visual_kmeans.py` script successfully fulfills the requirement of visualizing K-Means player clustering results by using PCA to reduce data dimensionality to 2D. The result is a 2D scatter plot and a new dataset accompanied by PCA coordinates, allowing for an initial visual assessment of cluster separation. Although this method has certain limitations related to information loss, it remains a useful tool in the data analysis toolkit for exploring and presenting clustering results in a more understandable way.

Chapter 4

Collecting Transfer Values and Building a Player Value Estimation Model

4.1 Collecting and Processing Player Transfer Values for the 2024-2025 Season

4.1.1 Introduction

This report section presents the process of collecting football player transfer value data for the 2024-2025 season from the website `https://www.footballtransfers.com`. This process focuses on obtaining information for players with actual playing time exceeding 900 minutes, based on the provided data. The main source code file is `transfer_values_2024-2025.py`.

4.1.2 Directory Structure and Libraries Used

- Directory structure: Organized with the main code file `transfer_values_2024-2025.py` and the configuration file `config.json` in the root directory.
- Input data is sourced from the `../problem_1/` directory, and results are saved in the `OUTPUT/` directory (including `raw_data.csv`, `player_transfer_values.csv`, `estimation_data.csv`).
- Main libraries used: Python, `os`, `re`, `time`, `pandas`, `json`, `BeautifulSoup` (from `bs4`), and `selenium` (along with its submodules).

4.1.3 Execution Process

Step 1: Load Configuration

The script begins by loading configuration parameters from the `config.json` file via the `load_config` function. This file contains important paths (e.g., `paths.part1_results_relative` for the playing time data file, `paths.output_folder` for the results directory), the target URL (`scraping.transfer_url`), user-agent information (`scraping.user_agent` to simulate a browser), CSS selectors (in `selectors`) to locate specific data snippets on the webpage (such as player name, ETV, "Next page" button), and the minimum playing time threshold (`processing.min_minutes_threshold`).

Step 2: Data Collection (Web Scraping) - performed in the `scrape_transfer_data` function:

- The `scrape_transfer_data` function uses Selenium to open the URL of the player list page (e.g., Premier League) specified in `config.json`.
- The script iterates through each page in the player list.
- Selenium waits for essential page elements, such as the player data table (`selectors.player`) to load completely before proceeding with extraction.
- For each player (each `selectors.player_row` row in the table), the following information is extracted based on CSS selectors defined in `config.json`:
  - Player name (`selectors.player_name`)
  - Team name (`selectors.team_name`)
  - Estimated Transfer Value (ETV - `selectors.etv`)
  - Age (`selectors.age`)
  - Position (`selectors.position`)
  - Skill (`selectors.skill`)
  - Potential (`selectors.potential`)
- Processing transfer value (ETV) immediately upon collection: Raw ETV data (e.g., "€50.5m") is cleaned and converted to millions of Euros at this step using the `clean_transfer_value` function. This function:
  - Removes currency symbols (e.g., '€', '$') and commas.
  - Identifies and processes 'm' (million) or 'k' (thousand) suffixes to convert to the full numeric value.
  - Standardizes the value to millions (e.g., 50.5).
  - If the value is invalid, it returns `None`.
- Similarly, other numeric values like Age, Skill, Potential are extracted as text, and the `safe_get_numeric` function attempts to convert them to numeric form, handling commas if present.
- "Button Clicking" Mechanism (Pagination):
  - After processing all players on a page, the script looks for the "Next page" button.
  - The `get_next_page_button` function is called, using `WebDriverWait` to wait up to 5 seconds for the "Next page" button (identified by the `selectors.next_page` selector from `config.json`, e.g., `button.pagination_next_button:not([disabled])`) to become available and clickable. This selector ensures only an active button is selected.
  - If a valid "Next page" button is not found (e.g., on the last page or due to a page load error), `get_next_page_button` returns `None`, and the data collection loop across pages will terminate.
  - If the button is found:
    - The script executes `driver.execute_script("arguments[0].scrollIntoView({block: 'center'});", next_btn)` to scroll the button into the center of the browser's viewport. This minimizes the risk of the button being obscured by other elements and becoming unclickable.

    - A short random pause (`time.sleep(random.uniform(0.4, 0.8))`) is applied.
    - The script attempts to click the button using `next_btn.click()`. If an `ElementClickInterceptedException` (meaning another element is obscuring the button), the script switches to a fallback: performing the click via JavaScript with `driver.execute_script("arguments[0].click();", next_btn)`.
    - After a successful click, a longer pause (`time.sleep(random.uniform(2.0, 3.5))`) is applied to wait for the new page to load completely before continuing the loop.
- After the `scrape_transfer_data` function completes iterating through all pages and collecting player information into a temporary data structure (a Python dictionary), this entire data block (`scraped_data`) will be passed to the next processing step, where it will be converted into a DataFrame and saved as a raw data file (`raw_data.csv`).

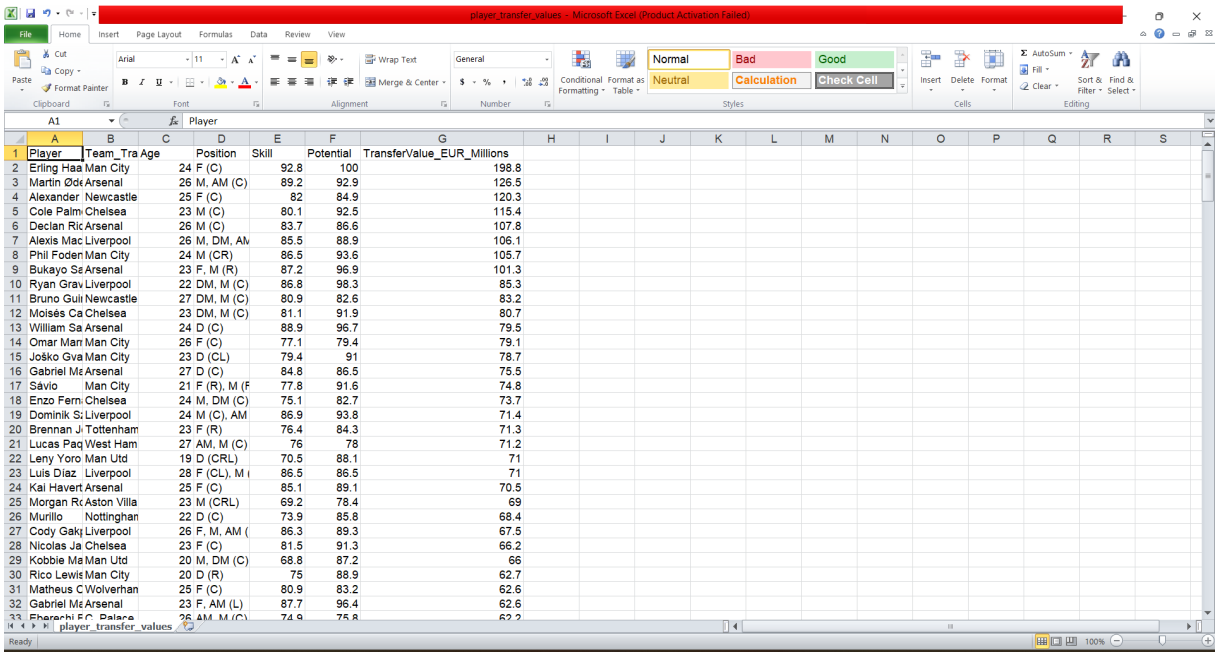
Step 3: Filter data based on playing time: This is a core step to ensure only players meeting the playing time criteria are retained.

- **Get list of valid players (`get_valid_players_from_part1`):** This function reads the external CSV file `results1.csv` (path declared in `config.json` via the key `paths.part1_results_relative`).
- From this `results1.csv` file, the script will primarily use two important columns to filter players:
  - The column containing player names, identified by the value of the `processing.part1_p` key in `config.json` (e.g., "Name").
  - The column containing player playing time, identified by the value of the `processing.part1_minutes_column` key in `config.json` (e.g., "Playing Time: minutes").
- Subsequently, the playing time from this column will be converted to a numeric type and compared with the `min_minutes_threshold` (900 minutes) to generate a list of "valid" player names.

Step 4: Apply filter to collected data

- In the `process_transfer_data` function, if the list of valid players (`valid_player_set` is not empty, the script will proceed with filtering.
- A temporary column `Player_Normalized` is created in `df_raw` by normalizing the 'Player' column (removing excess whitespace).
- `df_raw` is then filtered to retain only rows where the value in the `Player_Normalized` column is present in `valid_player_set`.
- The temporary `Player_Normalized` column is deleted after filtering.
- If no list of valid players is found from the input CSV file (e.g., file does not exist, no players meet the threshold), then `df_raw` (data not yet filtered by playing time) will be returned.

4.1.4 Results



	Player	Team_Tra	Age	Position	Skill	Potential	TransferValue_EUR_Millions
2	Erling Haa	Man City	24	F (C)	92.8	100	198.8
3	Martin Ødk	Arsenal	26	M, AM (C)	89.2	92.9	126.5
4	Alexander	Newcastle	25	F (C)	82	84.9	120.3
5	Cole Palm	Chelsea	23	M (C)	80.1	92.5	115.4
6	Declan Ric	Arsenal	26	M (C)	83.7	86.6	107.8
7	Alexis Mac	Liverpool	26	M, DM, AM	85.5	88.9	106.1
8	Phil Foden	Man City	24	M (CR)	86.5	93.6	105.7
9	Bukayo Sa	Arsenal	23	F, M (R)	87.2	96.9	101.3
10	Ryan Grav	Liverpool	22	DM, M (C)	86.8	98.3	85.3
11	Bruno Guir	Newcastle	27	DM, M (C)	80.9	82.6	83.2
12	Moisés Ca	Chelsea	23	DM, M (C)	81.1	91.9	80.7
13	William Sa	Arsenal	24	D (C)	88.9	96.7	79.5
14	Omar Mar	Man City	26	F (C)	77.1	79.4	79.1
15	Joško Gva	Man City	23	D (CL)	79.4	91	78.7
16	Gabriel Ma	Arsenal	27	D (C)	84.8	86.5	75.5
17	Sávio	Man City	21	F (R), M (F	77.8	91.6	74.8
18	Enzo Fern	Chelsea	24	M, DM (C)	75.1	82.7	73.7
19	Dominik Sz	Liverpool	24	M (C), AM	86.9	93.8	71.4
20	Brennan Jo	Tottenham	23	F (R)	76.4	84.3	71.3
21	Lucas Paq	West Ham	27	AM, M (C)	76	78	71.2
22	Leny Yoro	Man Utd	19	D (CRL)	70.5	88.1	71
23	Luis Díaz	Liverpool	28	F (CL), M (	86.5	86.5	71
24	Kai Havert	Arsenal	25	F (C)	85.1	89.1	70.5
25	Morgan Rc	Aston Villa	23	M (CRL)	69.2	78.4	69
26	Murillo	Nottinghan	22	D (C)	73.9	85.8	68.4
27	Cody Gakp	Liverpool	26	F, M, AM (	86.3	89.3	67.5
28	Nicolas Ja	Chelsea	23	F (C)	81.5	91.3	66.2
29	Kobbie Ma	Man Utd	20	M, DM (C)	68.8	87.2	66
30	Rico Lewis	Man City	20	D (R)	75	88.9	62.7
31	Matheus C	Wolverhan	25	F (C)	80.9	83.2	62.6
32	Gabriel Ma	Arsenal	23	F, AM (L)	87.7	96.4	62.6

Figure 4.1: Example of player transfer value data after processing.

The automated data collection and processing workflow has been successfully implemented through the `transfer_values_2024-2025.py` script and the `config.json` configuration file. The result of this process is the collection and filtering of detailed information for **284 players** with actual playing time exceeding 900 minutes in the 2024-2025 season from `footballtransfers.com`. The data obtained for each player includes the following important indicators:

- Team
- Age
- Position
- Skill
- Potential
- Estimated Transfer Value (TransferValue_EUR_Millions)

All this data has been stored in output files via the `save_data` function:

- `player_transfer_values.csv`: final data on player transfer values, processed and filtered
- `estimation_data.csv`: prepared for use in the next section: predicting player values

4.2 Building a Football Player Value Estimation Model

4.2.1 Introduction

The football player transfer market is a complex and dynamic field where accurately valuing players plays a crucial role for clubs. This report section presents a method for building a machine learning model to estimate player transfer values, based on performance statistics, potential, and other factors. The objective is to propose a systematic process from data collection, feature engineering, model selection to evaluation and interpretation of results. The target variable is the player's transfer value, transformed using a logarithmic function (`Log_TransferValue`) to stabilize variance and reduce the impact of outliers.

4.2.2 Data Description and Preprocessing

Data Sources and Merging

- **Data Sources:**
  - Estimation data (`estimation_data.csv`): Contains basic information and some initial player indicators.
  - Results data (`results1.csv`): May contain more detailed performance indicators from another source.
- **Data Merging:**
  - The two datasets are merged based on player and team information.
  - Team names have been standardized to ensure accurate merging (e.g., "B'mouth" to "Bournemouth").
- **Result after merging:** (283, 85), meaning 283 samples (players) and 85 initial features. This is performed in the `load_and_merge_data` function.

Feature Engineering

The goal of feature engineering is to create more meaningful new variables from raw data, helping the model learn better. Key steps include:

- **Create Primary Position (PrimaryPosition):** From the `Position_x` column (which may contain multiple positions), only the first position is taken as the player's primary position. Missing or undefined values are assigned as 'Unknown'.
- **Standardize and convert numeric data types:** Columns like 'Skill', 'Potential', 'Age_x', 'Expected: expected goals (xG)', 'Expected: expected Assist Goals (xAG)', 'TransferValue_EUR_Millions' are converted to numeric types. Conversion errors (if any) are handled by assigning `NaN`.
- **Calculate performance metrics per 90 minutes:**
  - `GoalContrib_per90`: Total goals and assists per 90 minutes.
  - `DefensiveActions_per90`: Total successful tackles (TklW) and interceptions (Int) per 90 minutes.

  - `Progression_per90`: Total progressive carries and passes per 90 minutes.

- **Standardize percentage columns:** Columns like 'Total: Pass completion (Cmp%)', 'Aerial Duels: Won%' are converted from string format (e.g., "85%") to decimal format (e.g., 0.85).

- **Transform target variable:**
  - Transfer value ('TransferValue_EUR_Millions') is ensured to be non-negative (negative values are clipped to 0).
  - Create the `Log_TransferValue` column by applying the `np.log1p` function (natural logarithm of 1+x) to `TransferValue_EUR_Millions`. This transformation helps to:
    * Reduce the skewness of the transfer value distribution.
    * Stabilize variance.
    * Ensure the input to the logarithm function is always positive, even if the transfer value is 0.

- **Result after feature engineering:** (283, 90), indicating that 5 new features were created or columns were processed. This process is performed in the `advanced_feature_engine` function.

4.2.3 Feature Selection and Data Preparation for the Model

Feature Selection

Not all created features are useful for the model. The following features were selected for inclusion in the model: 'Skill', 'Potential', 'Age_x', 'GoalContrib_per90', 'DefensiveActions_per90', 'Progression_per90', 'Expected: expected goals (xG)', 'Expected: expected Assist Goals (xAG)', 'Total: Pass completion (Cmp%)', 'Aerial Duels: Won%', 'PrimaryPosition_enc'.

Reasons for selection (inferred from feature names and general knowledge):

- *Individual attributes:* 'Age_x', 'Skill', and 'Potential' are fundamental indicators assessing a player's current quality, experience, and future development potential. Age is a crucial factor, often having a non-linear relationship with player value.

- *Attacking performance:* 'GoalContrib_per90', 'Expected: expected goals (xG)', 'Expected: expected Assist Goals (xAG)' measure the ability to contribute directly to the attack.

- *Defensive performance:* 'DefensiveActions_per90' reflects contributions in defense.

- *Ball progression and control ability:* 'Progression_per90' and 'Total: Pass completion (Cmp%)' indicate ball handling and distribution skills.

- *Specific physical/skill attributes:* 'Aerial Duels: Won%' is important for certain positions.

- *Positional factor:* 'PrimaryPosition' is an important factor as player values often differ significantly between positions.

Data Preparation

Final data preparation steps before model input, performed in the `prepare_model_data` function:

- **Handle Missing Values:** For selected numeric features, missing values are imputed with the mean of that column.
- **Numerical Feature Scaling:** Numeric features are standardized using `StandardScaler` (subtracting the mean and dividing by the standard deviation to achieve a mean of 0 and variance of 1). This helps scale-sensitive algorithms (like XGBoost when not using `tree_method='hist'` or distance-based models) perform more effectively.
- **Categorical Feature Encoding:** The `PrimaryPosition` feature is encoded using `LabelEncoder`. Each unique position will be assigned an integer value.
- **Create feature matrix (X) and target vector (y):**
  - X: Matrix containing processed features (standardized and encoded). Dimensions: (283, 10).
  - y: Vector containing the target variable `Log_TransferValue`. Dimensions: (283,).
- **Data check:** Ensure the X matrix is not empty and the number of samples in X matches y.

4.2.4 Model Selection and Training

Model Selection

The model selected for estimating player value is XGBoost (Extreme Gradient Boosting).
Reasons for choosing the XGBoost model to estimate player value:

- *Ability to capture complex and non-linear relationships:* Player value is determined by the multidimensional interaction of many factors such as age, skill, potential, and performance. XGBoost, with its structure based on an ensemble of decision trees, excels at automatically detecting and modeling these complex relationships (e.g., the effect of age is not always linear, or potential has different values at different ages) without requiring pre-definition. This is extremely important for accurately reflecting the nature of player valuation.
- *Superior predictive performance and effective overfitting control:* XGBoost is renowned for its ability to achieve high accuracy on tabular data. This algorithm integrates regularization mechanisms (L1, L2) and optimization techniques (such as `early_stopping_rounds` used here) that help prevent the model from becoming overly complex or "memorizing" the training data. As a result, the model has better generalization capabilities on new data, ensuring the reliability of player value predictions.
- *Efficient optimization and feature interpretability:* XGBoost is designed for rapid training, even on relatively large datasets, thanks to techniques like parallel processing and the histogram-based method (`tree_method='hist'`). More importantly, it provides detailed information about the importance of each feature (e.g., `Age_x`,

Skill, GoalContrib_per90), allowing us to understand which factors have the greatest impact on valuation. This not only helps in evaluating the model but also provides valuable insights for football analysts and managers.

Training and Hyperparameter Optimization

The training process is performed in the train_model function:

- **Data Splitting:** The data (X, y) is split into a training set and a test set with an 80:20 ratio (test_size=0.2). X_train has dimensions (226,11).
- **Hyperparameter Tuning:**
  - Uses RandomizedSearchCV to find the best set of hyperparameters for XGBoost.
  - RandomizedSearchCV tests a number of random parameter combinations (n_iter=20) from a predefined search space (param_grid).
  - Cross-validation with cv=5 is used during the search process to ensure the stability of the results.
  - The evaluation metric used during the search is r2 (R-squared).
- **Best hyperparameters found:**
  - subsample: 0.6
  - n_estimators: 1000
  - max_depth: 6
  - learning_rate: 0.05
  - gamma: 0.5
  - colsample_bytree: 0.6
- **Final Model Training:** The XGBoost model is retrained on the entire training set (X_train, y_train) with the best hyperparameters just found. early_stopping_rounds=50 and eval_set=[(X_test, y_test)] are used to monitor performance on the test set and stop early if there is no improvement, to avoid overfitting.
- **Training Environment:** The XGBoost model training process is performed on a CPU.

4.2.5 Model Evaluation

The model is evaluated on the entire processed dataset (X, y).

- **R^2 score (Coefficient of Determination R-squared = 0.880):** This means that approximately 88.0% of the variance in Log_TransferValue can be explained by the model's input features. This is a fairly good result, indicating the model's ability to capture important factors influencing player value.
- **MSE (Mean Squared Error = 0.088):** This is the average of the squared differences between the predicted and actual values (on a logarithmic scale). A smaller MSE value is better.

4.2.6 Analysis and Visualization of Results

Visualizations help to better understand the model's performance and internal workings.

Actual vs. Predicted Values Plot (`actual_vs_predicted.png`)

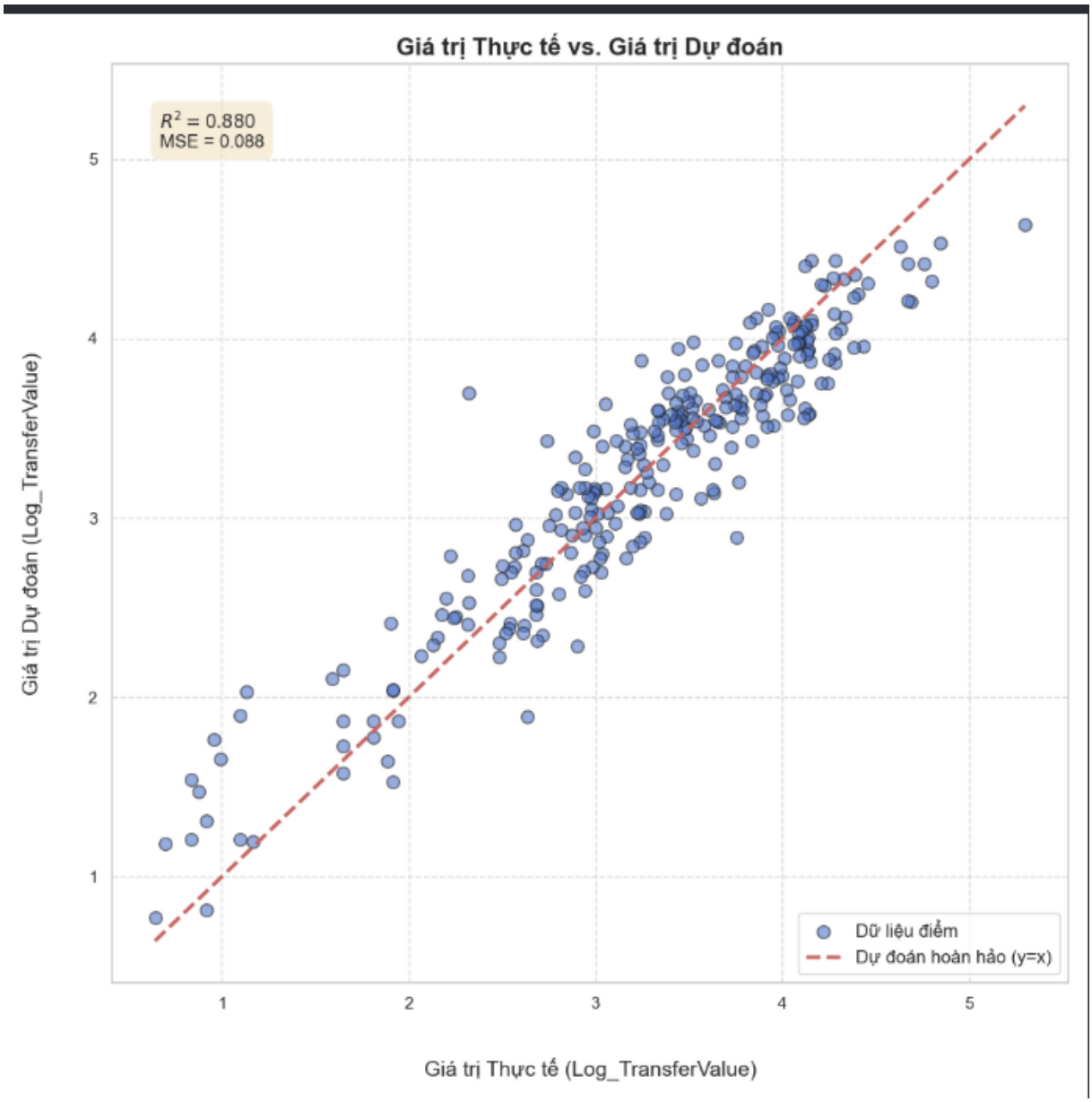


Figure 4.2: Actual vs. Predicted Values Plot.

Based on the plot: The model predicts player values **quite well**, as shown by data points concentrated relatively close to the diagonal line (perfect prediction) and a high R^2 score (0.880). However, the model **still has errors** (MSE = 0.088, on a log scale), with some predictions deviating significantly from actual values, particularly noticeable at the lower and higher value ranges.

Residual Plot (`residual_plot.png`)

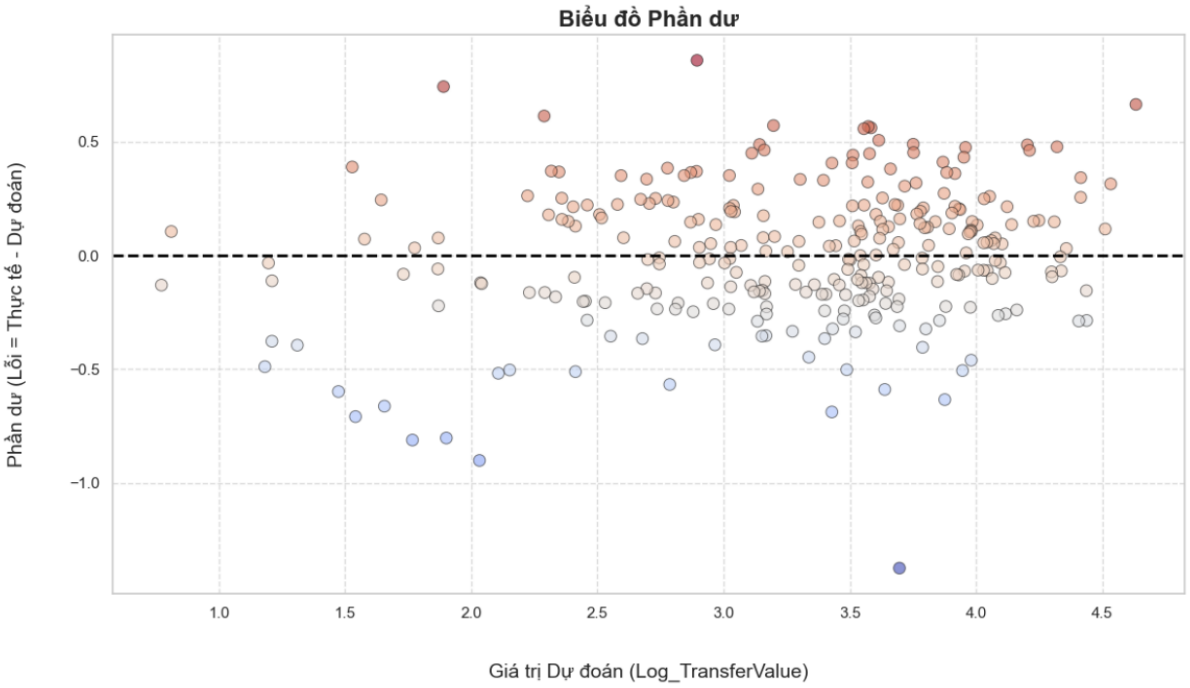


Figure 4.3: Residual Plot.

The residual plot shows that most points are distributed near the 0 line on the Residual axis, with a few points deviating significantly, especially at high predicted values (large `Log_TransferValue`). This indicates that the XGBoost model performs well in predicting player values, with high accuracy for the majority of the data, but may need improvement for outliers or high-value cases.

XGBoost Feature Importance (`xgb_feature_importance.png`)

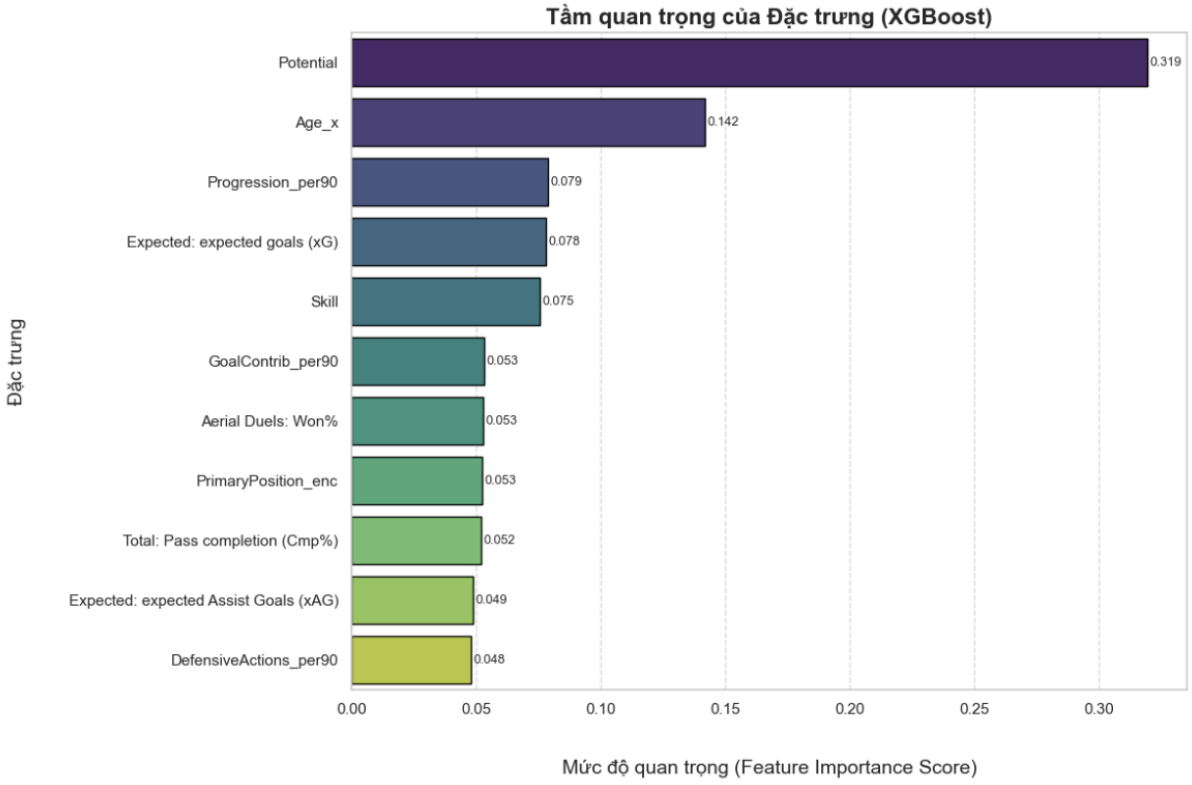


Figure 4.4: Feature Importance Plot (XGBoost).

The XGBoost Feature Importance plot shows "Potential" as the most important feature with a score of 0.319, followed by "Age" (0.142). Features like "Progression_per90" (0.079), "Expected: expected goals (xG)" (0.078), and "Skill" (0.075) also have a significant impact. Other features such as "DefensiveActions_per90" (0.048) have a lower level of importance. This indicates that the model relies heavily on player potential and age, while also considering performance indicators like progression and skill.

4.2.7 Conclusion and Future Development

Conclusion

The proposed method successfully built an XGBoost model to estimate player value (log-transformed) with fairly good performance (R2=0.880). This process included crucial steps from data merging, feature engineering, feature selection and preparation, to model training, optimization, and evaluation. Selected features such as `Skill`, `Potential`, `Age`, attacking/defensive performance metrics per 90 minutes, and passing/aerial duel statistics demonstrated their contribution to value prediction. The use of `RandomizedSearchCV` helped find good hyperparameters for XGBoost. Visualizations provided insightful interpretations of the model's behavior.

Future Development

- **Evaluation on an independent test set:** To obtain a more objective assessment of generalization ability, the model needs to be evaluated on a completely separate

test dataset (not used during hyperparameter optimization).

- **Experiment with other models:** Compare XGBoost with other algorithms like LightGBM, CatBoost, or simple neural network models.
- **More advanced feature engineering:**
  - Consider interactions between features (e.g., `Skill` * `GoalContrib_per90`).
  - Collect more data (e.g., league information, contract details, team achievements).
- **Detailed error analysis:** Examine cases where the model predicts most inaccurately to better understand its limitations.
- **Address data imbalance issues (if any):** If player values are very unevenly distributed, imbalance handling techniques may be needed.
- **Periodic model updates:** Player values and influencing factors change over time, so the model needs to be retrained and updated periodically with new data.

References

Bibliography

[1] XGBoost Developers. (2024). *XGBoost Documentation (Release 3.0.0)*. Accessed on May 5, 2025, from `https://xgboost.readthedocs.io/en/release_3.0.0/`

[2] Ông Xuân Hồng (2017, December 21). *XGBoost – the algorithm that wins many Kaggle competitions.* Accessed on May 5, 2025, from `https://ongxuanhong.wordpress.com/2017/12/21/xgboost-thuat-toan-gianh-chien-thang-tai-nhieu-cuoc-thi-kaggle/`

[3] Li, C., Kampakis, S., & Treleaven, P. (2022). *Machine Learning Modeling to Evaluate the Value of Football Players.* arXiv preprint arXiv:2207.11361. Accessed on May 5, 2025, from `https://arxiv.org/pdf/2207.11361.pdf`