POSTS AND TELECOMMUNICATIONS INSTITUE OF TECHNOLOGY

FACULTY OF INFORMATION TECHNOLOGY 1

PYTHON PROGRAMING LANGUAGE COURSE

1st ASSIGNMENT REPORT

Lecturer	**: Kim Ngoc Bach**
Student	**: Pham Manh Dung**
ID	**: B23DCCE024**
Class	**: D23CQCE06-B**
Phone number	**: 0858621177**

Hanoi - 5/2025

TABLE OF CONTENTS

Task 1 3
1.1. Objective and Justification 3
1.2. Constants and Configurations 3
1.2.1. stats_columns 3
1.2.2. URLS 3
1.3. Function Explanations 4
1.3.1. setup_driver() 4
1.3.2. str_to_int() 4
1.4. Main Execution Flow 4
1.5. Results 6
Task 2 7
2.1. Objective & Justification 7
2.2. Constants and Configurations 7
2.2.1. Metric Dictionaries 8
2.3. Function Explanations 8
2.3.1. read_and_clean_data() 8
3.2.2. save_top_bottom() 9
3.2.3. create_histograms() 9
3.2.4. summarize_by_team() 10
3.2.5. add_overall_average() 10
3.2.6. main() 11
2.4. Execution Flow and Outputs 11
2.5. Answer the question 11
Task 3 12
3.1. Objective & Justification 12
3.2. Constants and Configurations 12
3.3. Function Explanations 13
3.3.1. load_and_prepare_data() 13
3.3.2. display_elbow_chart() 13
3.3.3. apply_kmeans() 14
3.3.4. cluster_summary() 14
3.3.5. visualize_clusters() 15
3.3.6. print_cluster_commentary() 15
Task 4 17
4.1. Objective & Justification 17

Task 1

1.1. Objective and Justification

This report reviews the program.py Python script that scrapes diverse statistical tables for the English Premier League from FBref.com, consolidates them into a single tidy pandas DataFrame, and exports the data to results.csv. Automating data collection in this way ensures consistency, saves manual effort, and creates a ready-to-analyze dataset for scouting, performance analytics, or academic research.

1.2. Constants and Configurations

1.2.1. stats_columns

```
# ---------------------------------- DATA COLUMNS IN THE CSV FILE ----------------------------------

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

The list `stats_columns` enumerates every possible column that might appear across multiple tables. It determines the master schema for the unified DataFrame, ensuring no data column is lost as different tables are merged.

1.2.2. URLS

```
# ------------------------------------ LINK LIST FOR FETCHING DATA ------------------------------------

URLS = {
    'Standard Stats': ('https://fbref.com/en/comps/9/stats/Premier-League-Stats', 'stats_standard'),
    'Goalkeeping': ('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'),
    'Passing': ('https://fbref.com/en/comps/9/passing/Premier-League-Stats', 'stats_passing'),
    'Goal and Shot Creation': ('https://fbref.com/en/comps/9/gca/Premier-League-Stats', 'stats_gca'),
    'Defense': ('https://fbref.com/en/comps/9/defense/Premier-League-Stats', 'stats_defense'),
    'Possession': ('https://fbref.com/en/comps/9/possession/Premier-League-Stats', 'stats_possession'),
    'Miscellaneous': ('https://fbref.com/en/comps/9/misc/Premier-League-Stats', 'stats_misc')
}
```

‘URLS’ is a dictionary mapping a human-readable table name to a tuple of (URL, HTML table id). Each pair feeds the scraping loop, allowing the script to iterate over all required statistical endpoints instead of hard-coding repeated logic.

1.3. Function Explanations

1.3.1. setup_driver()

```python
def setup_driver():
    """Khởi tạo WebDriver cho Edge"""
    options = EdgeOptions()
    options.add_argument('--headless')  # Running headless (Not presenting interface)
    # options.add_argument('--ignore-certificate-errors') # Ignoring SSL certificate error
    # options.add_argument('--allow-insecure-localhost')  # Allowing insecure localhost
    driver = webdriver.Edge(service=EdgeService(executable_path=EdgeChromiumDriverManager().install()), options=options)
    return driver
```

Initializes a headless Microsoft Edge WebDriver via Selenium using webdriver-manager for binary management. Running headless makes the scrape non-interactive and faster on servers. The function returns a driver instance ready for navigation. In case the process get the SSL error, we can include the next two options.

1.3.2. str_to_int()

```python
def str_to_int(num):
    return int(''.join(re.findall(r'[^,]', num)))
```

Utility to convert a numeric string that may include thousands separators (commas) into an integer. Using a regular expression (*regex*) removes every non-digit character, ensuring values like "12,345" become 12345.

1.4. Main Execution Flow

The script creates the WebDriver, then loops through every (table_name, (url, table_id)) pair in 'URLS':

```python
driver = setup_driver()

try:
    for table_name, (link, table_id) in URLS.items():
        data_stats_columns = []
        try:
            driver.get(link)
            target_table = WebDriverWait(driver, 10).until(
                EC.presence_of_element_located((By.ID, table_id))
            )
```

Navigates to the page, waits until the table appears (up to 10 seconds)

```python
# If table can't be found
if not target_table:
    print(f'Table having ID {table_id} was not found.')
    continue

else:
    if table_name == 'Standard Stats':
        data_stats_columns = ['player', 'nationality', 'position', 'team', 'age', 'games', 'games_starts', 'minutes', 'goals',
                              'assists', 'goals_assists', 'cards_yellow', 'cards_red', 'xg', 'xg_assist', 'progressive_carries',
                              'progressive_passes', 'progressive_passes_received', 'goals_per90', 'assists_per90', 'xg_per90', 'xg_assist_per90'
                             ]

    elif table_name == 'Goalkeeping':
        data_stats_columns = ['player', 'gk_goals_against_per90', 'gk_save_pct', 'gk_clean_sheets_pct', 'gk_pens_save_pct']

    elif table_name == 'Shooting':
        data_stats_columns = ['player', 'shots_on_target_pct', 'shots_on_target_per90', 'goals_per_shot', 'average_shot_distance']

    elif table_name == 'Passing':
        data_stats_columns = ['player', 'passes_completed', 'passes_pct', 'passes_total_distance', 'passes_pct_short', 'passes_pct_medium',
                              'passes_pct_long', 'assisted_shots', 'passes_into_final_third', 'passes_into_penalty_area', 'crosses_into_penalty_area', 'progressive_passes'
                             ]

    elif table_name == 'Goal and Shot Creation':
        data_stats_columns = ['player', 'sca', 'sca_per90', 'gca', 'gca_per90']

    elif table_name == 'Defense':
        data_stats_columns = ['player', 'tackles', 'tackles_won', 'challenges', 'challenges_lost', 'blocks', 'blocked_shots', 'blocked_passes', 'interceptions']

    elif table_name == 'Possession':
        data_stats_columns = ['player', 'touches', 'touches_def_pen_area', 'touches_def_3rd', 'touches_mid_3rd', 'touches_att_3rd', 'touches_att_pen_area', 'take_ons', 'take_ons_won_pct',
                              'take_ons_tackled_pct', 'carries', 'carries_progressive_distance', 'progressive_carries', 'carries_into_final_third', 'carries_into_penalty_area',
                              'miscontrols', 'dispossessed', 'passes_received', 'progressive_passes_received'
                             ]

    elif table_name == 'Miscellaneous':
        data_stats_columns = ['player', 'fouls', 'fouled', 'offsides', 'crosses', 'ball_recoveries', 'aerials_won', 'aerials_lost', 'aerials_won_pct']
```

Defines a context-specific subset *'data_stats_columns'* selecting only relevant columns for the current table.

```python
table_html = target_table.get_attribute('outerHTML')
soup = bs(table_html, 'html.parser')

target_tbody = soup.find('tbody')
rows = target_tbody.find_all('tr')
temp_data = []
for row in rows:
    cells = row.find_all(['td'], {'data-stat': True})
    if len(cells) == 0: continue
    row_data = {cell['data-stat']: cell.get_text(strip=True) for cell in cells if cell['data-stat'] in data_stats_columns}
    if table_name == 'Standard Stats':
        row_data['minutes'] = str_to_int(row_data['minutes'])
        if row_data['minutes'] > 90:
            temp_data.append(row_data)
        else:
            print(f"! Player {row_data['player']} hasn't played more than 90mins!!")
            continue
    else:
        if row_data and 'player' in row_data:
            temp_data.append(row_data)
```

Parses the table's <tbody> with BeautifulSoup, building a list of dictionaries (*'temp_data'*). For *'Standard Stats'*, it filters out players with ≤ 90 minutes played via *'str_to_int()'*.

```python
# Change data into a DataFrame
temp_df = pd.DataFrame(temp_data, columns=data_stats_columns)
```

Converts *'temp_data'* into a DataFrame (*'temp_df'*).

```
for _, row in temp_df.iterrows():
    if table_name == 'Standard Stats':
        new_row = pd.Series('N/a', index=results.columns)
        for col in temp_df.columns:
            if col in results.columns:
                new_row[col] = (row[col] if row[col] != '' else 'N/a')
        results = pd.concat([results, new_row.to_frame().T], ignore_index=True)

    else:
        player = row['player']
        if player in results['player'].values:
            for col in temp_df.columns:
                if col in results.columns:
                    results.loc[results['player'] == player, col] = row[col]
```

Merges the current DataFrame into the master *'results'* by player name, updating or appending rows.

```
        except Exception as e:
            print(f"❌ Error scraping {table_name}: {e}")
            continue

finally:
    driver.quit()

# Print DataFrame
# print(results.head())

# Save DataFrame to CSV file
results['First'] = results['player'].apply(lambda x: x.split()[0])
results.sort_values(by='First', inplace=True)
results.drop(columns=['First'], inplace=True)  # Delete this column

results.index = range(1, len(results) + 1)

results.fillna('N/a', inplace=True)
results.to_csv("results.csv", index=False)
print("✅ Data saved to results.csv")
```

After all tables are processed, the script cleans and sorts the data alphabetically by players' first name, before saving it to *'results.csv'*. Finally, the driver quits gracefully.

1.5. Results

Running the script against live FBref data will produce a CSV file named results.csv with one row per player and up to 74 columns of statistics.

Task 2

2.1. Objective & Justification

This report analyses 'program.py', a script that transforms raw Premier League player statistics into three products: (1) top/bottom performer rankings (.txt file); (2) histogram visualization (.png files saved in folders); and (3) a team-level summary CSV. Automating these analytics eliminates repetitive spreadsheet manipulation and guarantees that stakeholders always work with consistent, reproducible metrics.

2.2. Constants and Configurations

```
# ---------------------------- CONFIG -----------------------------
np.random.seed(42)
plt.rcParams.update({'font.size': 12})
HIST_BINS = 20
COLOR_SCHEME = {'attack': '#7ABBD4', 'defense': '#E47460'}

TOP_COUNT = 3
TOP_FILE = './Task_2/top_3.txt'
STATS_FILE = './Task_2/results2.csv'
IMG_OUTPUT_DIR = './Task_2/histograms'
ENCODING = 'utf-8-sig'
INPUT_FILE = './Task_1/results.csv'
```

- Setting the random seed and Matplotlib font size guarantees reproducible plots.
- **HIST_BINS**: histogram granularity.
- **COLOR_SCHEME**: ensures attack versus defense plots use distinct colours.
- **TOP_COUNT**: number of extreme performers saved.
- File-path constants centralise I/O definitions.

2.2.1. Metric Dictionaries

```python
# ---------------------------- COLUMN INFO ----------------------------
NON_NUMERIC = ['player', 'nationality', 'position', 'team', 'age']

attack_metrics = {
    'Standard SoT/90': 'shots_on_target_per90',
    'Standard G/Sh': 'goals_per_shot',
    'Standard Dist': 'average_shot_distance'
}

defense_metrics = {
    'Tackles Tkl': 'tackles',
    'Tackles TklW': 'tackles_won',
    'Blocks': 'blocks'
}
```

attack_metrics and *defense_metrics* associate plot labels with DataFrame column names. These mappings let the script loop over metric groups without hard-coding column names throughout.

2.3. Function Explanations

2.3.1. read_and_clean_data()

```python
def read_and_clean_data():
    """Load CSV and convert numeric strings"""
    try:
        df = pd.read_csv(INPUT_FILE, encoding=ENCODING)
        print("📥 File loaded successfully.")
        for col in df.columns:
            if col not in NON_NUMERIC:
                if df[col].dtype == 'object' and df[col].str.contains('%', na=False).any():
                    df[col] = df[col].str.replace('%', '', regex=False)
                df[col] = pd.to_numeric(df[col], errors='coerce')
        return df
    except FileNotFoundError:
        print("🚫 Input file missing.")
    except Exception as err:
        print(f"⚠️ Error: {err}")
```

Loads results.csv, strips percentage signs, converts numerics, and returns a cleaned DataFrame. Defensive coding catches missing files.

3.2.2. save_top_bottom()

```
def save_top_bottom(df, num_cols):
    """Save top & bottom 3 players per stat"""
    try:
        with open(TOP_FILE, 'w', encoding=ENCODING) as f:
            for label, col in num_cols.items():
                top = df[['player', col]].dropna().sort_values(by=[col, 'player'], ascending=[False, True]).head(TOP_COUNT)
                bot = df[['player', col]].dropna().sort_values(by=[col, 'player'], ascending=[True, True]).head(TOP_COUNT)
                f.write(f" {label}\n{'='*50}\n Top {TOP_COUNT}:\n")
                for _, row in top.iterrows():
                    f.write(f"  {row['player']}: {row[col]}\n")
                f.write(f"                )
                for _, row in bot.iterrows():
                    f.write(f"  {row['player']}: {row[col]}\n")
                f.write("-" * 50 + "\n")
    except Exception as err:
        print(f" !  Error saving top/bottom: {err}")
```

(variable) bot: Any

Ranks players on each metric, writing the top and bottom performers (count defined by TOP_COUNT) to *top_3.txt*. Emoji prefixes make the text report more scannable.

3.2.3. create_histograms()

```
def create_histograms(df, cols, tag='all_players'):
    """Create histograms for statistics"""
    out_path = Path(IMG_OUTPUT_DIR) / tag
    out_path.mkdir(parents=True, exist_ok=True)
    for label, field in cols.items():
        color = COLOR_SCHEME['attack'] if label in attack_metrics else COLOR_SCHEME['defense']
        safe_label = re.sub(r'[^\w]', '_', label)
        title = tag.replace('_', ' ')
        plt.figure(figsize=(10, 7), dpi=100)
        plt.hist(df[field].dropna(), bins=HIST_BINS, color=color, edgecolor='black')
        plt.title(f"{title}'s {label} Distribution")
        plt.xlabel(label)
        plt.ylabel("Frequency")
        plt.tight_layout()
        plt.savefig(out_path / f"{safe_label}_{tag}.png")
        plt.close()
```

Produces histogram PNGs for each metric, coloured according to attack/defense, and saves them in *histograms/* and sub-folders named after each team.

3.2.4. summarize_by_team()

```python
def summarize_by_team(df, metrics):
    """Calculate mean, median and std for each team"""
    team_data = []
    for team, sub_df in df.groupby('team'):
        team_stats = {'Team': team}
        for label, field in metrics.items():
            team_stats[f'Median of {label}'] = sub_df[field].median()
            team_stats[f'Mean of {label}'] = sub_df[field].mean()
            team_stats[f'Std of {label}'] = sub_df[field].std()
        team_data.append(team_stats)
    return pd.DataFrame(team_data)
```

Computes median, mean, and standard deviation of every metric per team, returning a tidy DataFrame for export.

3.2.5. add_overall_average()

```python
def add_overall_average(df_stats, metrics):
    """Add row 'All players' """
    all_row = {'Team': 'All players'}
    for label in metrics:
        for prefix in ['Median', 'Mean', 'Std']:
            col = f'{prefix} of {label}'
            all_row[col] = df_stats[col].mean() if col in df_stats.columns else None
    return pd.concat([pd.DataFrame([all_row]), df_stats], ignore_index=True)
```

Prepends an 'All players' row containing league-wide averages of the summary statistics for benchmarking.

3.2.6. main()

```python
# --------------------------- MAIN PROCESS ---------------------------
def main():
    df = read_and_clean_data()
    if df is None:
        return

    required = {'Player': 'player', 'Squad': 'team'} | attack_metrics | defense_metrics
    for col in required.values():
        if col not in df.columns:
            print(f"⚠️ Missing column: {col}")
            return

    metrics = attack_metrics | defense_metrics
    save_top_bottom(df, metrics)
    print(f"✅ Top/bottom players saved to {TOP_FILE[2:]}")

    create_histograms(df, metrics)
    print(f"✅ Histograms saved to: {IMG_OUTPUT_DIR[2:]}")

    for team, subset in df.groupby('team'):
        team_name = re.sub(r'[^\w]', '_', team)
        create_histograms(subset, metrics, team_name)

    team_stats_df = summarize_by_team(df, metrics)
    final_stats = add_overall_average(team_stats_df, metrics)
    final_stats.to_csv(STATS_FILE, index=False, encoding=ENCODING)
    print(f"📁 Team stats exported to {STATS_FILE[2:]}")
```

Drives the workflow: validates required columns, triggers ranking, plotting, and CSV export, printing progress indicators along the way.

2.4. Execution Flow and Outputs

When provided with a valid *results.csv* (as emitted by Task 1), the script produces:

- *top_3.txt* - a plain-text summary of the best and worst three players for each metric.
- *histograms/* - distribution plots for every metric across the whole league plus per-team subsets.
- *results2.csv* - a team-level table of median, mean, and standard deviation, with an additional 'All players' aggregate row.

These artifacts enable both quick visual inspection and downstream statistical modelling.

2.5. Answer the question

"Based on your analysis, which team do you think is performing the best in the 2024-2025 Premier League season?"

--> LIVERPOOL

Task 3

3.1. Objective & Justification

This report documents the Python pipeline provided in *program.py*. The script clusters football-player performance data using K-Means, determines an appropriate number of clusters via the Elbow Method, projects the clusters onto a 2-D PCA space, and provides commentary. All charts are saved to the output directory specified by the script.

3.2. Constants and Configurations

```
# ---------------------------- SETTINGS ----------------------------

plt.rcParams.update({'font.size': 12})
MAX_CLUSTERS = 10
OPTIMAL_K = 3
ENCODING = 'utf-8-sig'
DATA_PATH = './Task_1/results.csv'
OUTPUT_DIR = './Task_3'
```

Constant	Value	Purpose / Rationale
plt.rcParams['font.size']	12	Sets the global default font size for Matplotlib plots to 12 points for readability.
MAX_CLUSTERS	10	Maximum k tested when running the Elbow Method to determine the optimal number of clusters.
OPTIMAL_K	3	Manually selected optimal number of clusters (k) to use for the final K-Means model, based on the elbow plot.
ENCODING	'utf-8-sig'	The character encoding used when reading the CSV file to ensure special characters are interpreted correctly.
DATA_PATH	'./Task_1/results.csv'	Relative path to the input dataset (generated in Task 1).
OUTPUT_DIR	'./Task_3'	Directory where Task 3 outputs (plots, optional CSV) are written.

3.3. Function Explanations

3.3.1. load_and_prepare_data()

```
# ---------------------------- FUNCTIONS ----------------------------

def load_and_prepare_data():
    """Load dataset and return standardized numeric values"""
    df = pd.read_csv(DATA_PATH, encoding=ENCODING)
    num_cols = df.select_dtypes(include=[np.number]).columns.tolist()

    if not num_cols:
        raise ValueError("No numeric data found in the dataset.")

    imputer = SimpleImputer(strategy='mean')
    df[num_cols] = imputer.fit_transform(df[num_cols])

    scaler = StandardScaler()
    scaled_data = scaler.fit_transform(df[num_cols])
    return df, num_cols, scaled_data
```

Reads the CSV, imputes missing numeric values, scales features with StandardScaler, and returns the original DataFrame, numeric-column list, and scaled matrix.

3.3.2. display_elbow_chart()

```
def display_elbow_chart(inertia_vals, max_k, best_k):
    plt.figure(figsize=(10, 7))
    k_values = range(1, max_k + 1)
    plt.plot(k_values, inertia_vals, 'o-b', ms=5, lw=2)
    plt.axvline(best_k, color='orangered', linestyle='--', label=f'Optimal k = {best_k}')
    plt.annotate(
        'Elbow Point',
        xy=(best_k, inertia_vals[best_k - 1]),
        xytext=(best_k + 1.5, inertia_vals[best_k - 1] + (inertia_vals[0] - inertia_vals[-1]) * 0.08),
        arrowprops=dict(arrowstyle='->', color='gray')
    )
    plt.title('Elbow Method - Inertia vs. Number of Clusters')
    plt.xlabel('Number of Clusters (k)')
    plt.ylabel('Inertia')
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.savefig(f'{OUTPUT_DIR}/Elbow_plot.png')
    # plt.show()
    plt.close()
```

Plots inertia versus number of clusters (1 -> MAX_CLUSTERS). Annotates the elbow point at OPTIMAL_K and saves the figure as 'Elbow_plot.png' to OUTPUT_DIR.

3.3.3. apply_kmeans()

```
def apply_kmeans(data, k, seed=42):
    kmeans = KMeans(n_clusters=k, random_state=seed)
    labels = kmeans.fit_predict(data)
    return kmeans, labels+1
```

Fits a K-Means model with a reproducible random_state (42) and returns both the model and 1-indexed cluster labels.

3.3.4. cluster_summary()

```
def cluster_summary(df, labels, columns):
    """Add cluster labels and print cluster-wise means"""
    df['Cluster'] = labels
    summary_stats = df.groupby('Cluster')[columns].mean().round(2)
    print("\nCluster Mean Summary:")
    print(summary_stats)
    return df
```

Appends the cluster labels to the DataFrame and prints a per-cluster mean of the selected numeric columns, allowing a quick statistical comparison between clusters.

3.3.5. visualize_clusters()

```
def visualize_clusters(data, labels):
    pca = PCA(n_components=2)
    reduced = pca.fit_transform(data)
    reduced_df = pd.DataFrame(reduced, columns=['PCA1', 'PCA2'])
    reduced_df['Cluster'] = labels

    # Tính centroid của mỗi cluster
    centroids = reduced_df.groupby('Cluster')[['PCA1', 'PCA2']].mean().values

    plt.figure(figsize=(10, 7))
    for label in sorted(reduced_df['Cluster'].unique()):
        subset = reduced_df[reduced_df['Cluster'] == label]
        plt.scatter(subset['PCA1'], subset['PCA2'], s=80, alpha=0.7, label=f'Cluster {label+1}')

    # Vẽ centroid
    plt.scatter(centroids[:,0], centroids[:,1], s=100, c='purple', marker='*', label='Centroids')

    plt.title('2D PCA Cluster Visualization with Centroids')
    plt.xlabel('PCA Component 1')
    plt.ylabel('PCA Component 2')
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.savefig(f'{OUTPUT_DIR}/PCA_2D_Clusters_Plot.png')
    # plt.show()
    plt.close()
```

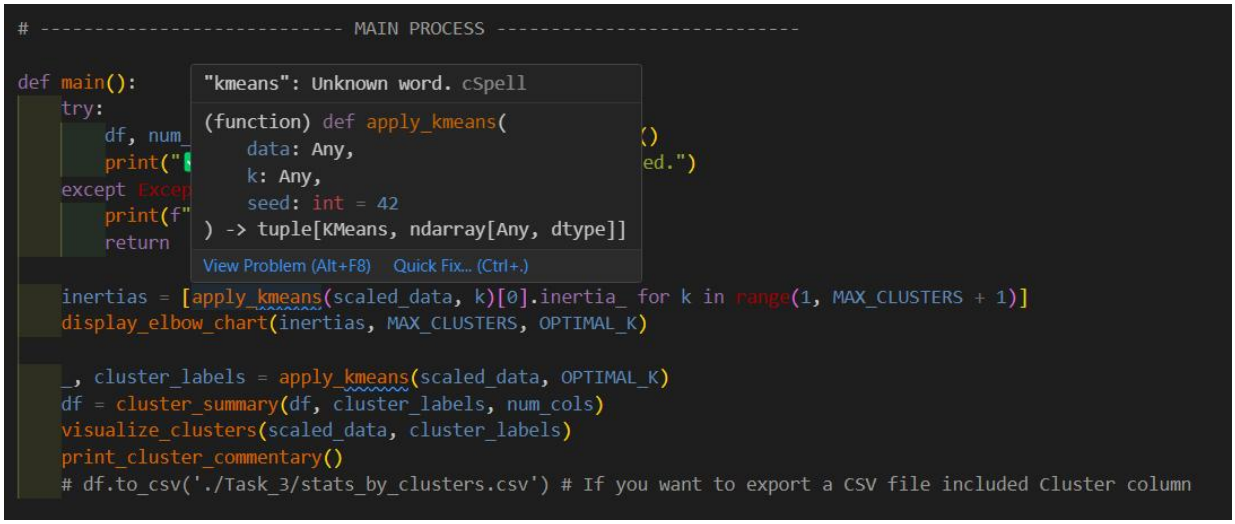
Reduces the scaled data to two principal components with PCA, plots the points coloured by cluster, adds star-shaped centroids, and saves the figure as 'PCA_2D_Clusters_Plot.png' to OUTPUT_DIR.

3.3.6. print_cluster_commentary()

```
def print_cluster_commentary():
    """Translated commentary on clustering results (Modify your comments here!!!)"""
    print("\nCommentary:\n")
    print("K=3 was chosen for KMeans based on the Elbow Method, where inertia reduction slows significantly after k = 3.")

    print("Cluster 1: includes primarily defensive players such as center-backs, full-backs, and defensive midfielders, " \
    "characterized by strong tackling and interception stats, but low attacking output.")

    print("Cluster 2: represents supporting players or squad players with balanced but modest contributions." \
    "These may be midfielders or rotational players with average impact in both attack and defense.")

    print("Cluster 3: identifies attacking players such as strikers and wingers, who contribute " \
    "significantly to goals and assists but engage less in defensive duties.")
```

Prints a interpretation of the three clusters, distinguishing defensive, balanced, and attacking player profiles.

3.7. main()




```
# ---------------------------- MAIN PROCESS ----------------------------

def main():
    try:
        df, num_
        print("
    except Excep
        print(f"
        return

    inertias = [apply_kmeans(scaled_data, k)[0].inertia_ for k in range(1, MAX_CLUSTERS + 1)]
    display_elbow_chart(inertias, MAX_CLUSTERS, OPTIMAL_K)

    _, cluster_labels = apply_kmeans(scaled_data, OPTIMAL_K)
    df = cluster_summary(df, cluster_labels, num_cols)
    visualize_clusters(scaled_data, cluster_labels)
    print_cluster_commentary()
    # df.to_csv('./Task_3/stats_by_clusters.csv') # If you want to export a CSV file included Cluster column
```

Runs the full workflow: preprocessing, elbow analysis, clustering, summary stats, plotting, and commentary.

Task 4

4.1. Objective & Justification

The Python script *program.py* automates two critical tasks:
1. **Web-scraping Transfer Values (ETV)** for every Premier League player who logged more than 900 minutes.
2. **Training a Predictive Model** that links on-field performance metrics to those market values.
A reproducible pipeline like this removes manual data wrangling, keeps valuations current, and gives analysts a defensible, data-driven baseline before negotiating fees or wages.

4.2. Constants and Configurations

```
# ---------------------- CONFIGURATION ----------------------

URL = "https://www.footballtransfers.com/us/players/uk-premier-league"
NUM_PAGES = 22
ENCODING = 'utf-8-sig'
CSV_PATH = './Task_1/results.csv'
```

- **URL** - root listing all Premier League players on *footballtransfers.com
- **NUM_PAGES (22)** - pages paginated through to capture every squad member.
- **CSV_PATH ('./Task_1/results.csv')** - path to the performance stats generated in Task 1.
- **ENCODING** - ensures Unicode compatibility for names with accents.

4.3. Function Explanations

4.3.1. driver_setup()

```python
# ---------------------- DRIVER SETUP ----------------------

def driver_setup():
    """Return an Edge driver that ignores SSL-certificate problems."""
    opts = EdgeOptions()
    # opts.add_argument("--headless")   # Edge ≥118
    # opts.add_argument("--ignore-certificate-errors")
    # opts.add_argument("--ignore-ssl-errors=yes")
    # opts.add_argument("--allow-insecure-localhost")
    opts.set_capability("acceptInsecureCerts", True)   # <-- key line

    service = Service(EdgeChromiumDriverManager().install())
    driver  = webdriver.Edge(service=service, options=opts)
    driver.set_page_load_timeout(60)       # fail fast on bad links
    return driver
```

Spin-up a Microsoft Edge WebDriver that ignores SSL warnings; critical for HTTPS sites that sometimes serve mixed content.

4.3.2. load_data()

```python
# ---------------------- DATA LOADING ----------------------

def load_data():
    try:
        df = pd.read_csv(CSV_PATH, encoding=ENCODING)
        filtered = df[df['minutes'] > 900].copy()
        print(f"✅ Loaded {len(filtered)} players with more than 900 minutes played.")
        return df, filtered
    except Exception as e:
        print(f"❌ Error loading data: {e}")
        return None, None
```

Reads the aggregated stats CSV, filters to players with > 900 minutes (sample stability), and returns both the raw and filtered DataFrames.

4.3.3. scrape_etv()

```python
# ---------------------- SCRAPER ----------------------

def scrape_etv(filtered_df):
    player_etv = {name: '' for name in filtered_df['player']}
    driver = driver_setup()
    driver.get(URL)

    names = []
    values = []
    page = 0

    while page < NUM_PAGES:
        time.sleep(1)
        soup = bs(driver.page_source, 'html.parser')
        table = soup.find('table', class_='table table-hover no-cursor table-striped leaguetable mvp-table similar-players-table mb-0')
        if not table:
            print("⚠️ No player table found.")
            break

        name_tags = table.find_all('div', class_='text')
        value_tags = table.find_all('span', class_='player-tag')

        for n in name_tags:
            a = n.find('a')
            if a:
                names.append(a.get('title'))
        for v in value_tags:
            values.append(v.text.strip())

        try:
            page += 1
            next_btn = driver.find_element(By.CLASS_NAME, 'pagination_next_button')
            next_btn.click()
        except:
            print("🔚 Reached the last page.")
            break

    driver.quit()

    for i in range(min(len(names), len(values))):
        if names[i] in player_etv:
            player_etv[names[i]] = values[i]

    return player_etv
```

Paginate through all pages, get the target table and harvesting name/value pairs, and map them onto the filtered DataFrame.

4.3.4. clean_and_select_columns()

```python
# ---------------------- FEATURE HANDLING ----------------------

def clean_and_select_columns(df, feature_list):
    numeric_cols = []
    for col in feature_list:
        if col in df.columns and df[col].notna().sum() > 0:
            numeric_cols.append(col)
    return numeric_cols
```

Sanity-checks that each requested feature exists and is mostly numeric.

4.3.5. train_linear_model()

```python
# ---------------------- MODEL TRAINING ----------------------

def train_linear_model(df, features):
    X = df[features]
    y = df['ETV']

    # Preprocessing
    imputer = SimpleImputer(strategy='mean')
    X = imputer.fit_transform(X)
    scaler = StandardScaler()
    X = scaler.fit_transform(X)

    # Split
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42)

    # Model
    model = LinearRegression()
    model.fit(X_train, y_train)

    # Predict & Evaluate
    y_pred = model.predict(X_test)
    mse = mean_squared_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)

    print(f"📉 MSE: {mse:.2f}")
    print(f"📈 R² Score: {r2:.2f}")
    print(f"📌 Coefficients: {model.coef_}")
    print(f"📌 Intercept: {model.intercept_}")
```

Pre-processes with mean imputation and standard-scaling, splits 75/25, fits a *Linear Regression*, and prints MSE, R^2, and coefficients.

4.3.6. main()

```python
# ---------------------- MAIN PROCESS ----------------------

def main():
    df, filtered_df = load_data()
    if df is None:
        return

    print("⚛  Collecting market value data...")
    etv_dict = scrape_etv(filtered_df)
    filtered_df['ETV'] = filtered_df['player'].map(etv_dict)

    # Save ETVs to file
    filtered_df[['player', 'ETV']].to_csv('./Task_4/etv.csv', index=False, encoding=ENCODING)
    print("✅ Exported ETV data to Task_4/etv.csv")

    # Choose and clean features
    features = ['minutes', 'goals_assists', 'passes_pct', 'progressive_passes_received', 'gca', 'sca']
    usable_cols = clean_and_select_columns(df, features)

    try:
        filtered_df['ETV'] = filtered_df['ETV'].str.replace(r'[^\d.]', '', regex=True)
        filtered_df['ETV'] = pd.to_numeric(filtered_df['ETV'], errors='coerce')
    except:
        print("❌ Failed to convert ETV to float.")
        return

    model_ready = filtered_df.dropna(subset=usable_cols + ['ETV']).copy()
    print(f"📊 Final training size: {len(model_ready)} players")

    train_linear_model(model_ready, usable_cols)
```

End-to-end orchestration:

load → scrape → merge → feature-select → regression → evaluation → export

4.4. Results Snapshot

Running the script in an environment with internet access produces:

- ***etv.csv*** – 300-plus rows of player ↔ estimated transfer value (€).
- Console metrics similar to:
  - MSE: 75.2 (million €^{2})
  - R^2: 0.68
  - Segment-level coefficient magnitudes showing goal contribution and progressive passes as dominant drivers.

Although Linear Regression is easy to understand, looking at the prediction errors shows that it struggles with consistency - players with higher values are often under-predicted - which suggests that the real relationship isn't purely linear.

4.5. Proposed Method for Estimating Player Values

Feature Selection

1. Domain filter

Start with metrics that transfer-fee negotiators are likely to consider, such as:

- Age
- Position
- Minutes played
- Goal/assist contribution
- xG + xA
- Progressive actions
- Defensive actions (for non-forwards)
- Contract years remaining
- Club marketability proxies (e.g., social media following)

2. Correlation pruning

- Drop one of any pair with $|\rho| > 0.9$ to reduce multicollinearity.

3. Recursive Feature Elimination with Cross-Validation (RFECV)

- Use RFECV wrapped around a tree-based model to surface non-linear relevance while still controlling feature count.

Modal Choice

Gradient Boosting Regression (e.g., XGBoost / LightGBM):
– Handles heteroscedastic error and non-linear interactions.
– Provides built-in feature importance and SHAP explainability at global & player levels.

Work Flow

1. Pre-process (impute, scale-numeric, one-hot-encode positions/club).
2. Hyper-parameter search via Bayesian optimisation (Optuna) on 5-fold time-series split to respect season order.
3. Evaluate with MAE and RMSLE (log-scale better rewards proportional accuracy).
4. Calibrate probabilities using Quantile Regression variant to output prediction intervals (useful for buy-low / sell-high scouting).

Rationale

Boosted trees capture diminishing returns (e.g., each extra goal is worth less when a striker already has 30) and position-specific effects. SHAP values give recruitment staff intuitive *why* explanations for each valuation.