University of Essex
Department of Mathematical Sciences

MA838: CAPSTONE PROJECT

An Application to Help Predict and Manage Premier League Fantasy Football

Reece Lance

1804752

Supervisor: **Dr Andrew Harrison**

April 14, 2022

Contents

CHAPTER 1

Introduction

Every year, over eight million people play the biggest fantasy football game in the world, Fantasy Premier League. With the aim of being the best, players pick the footballers they believe will perform week in week out and achieve the highest scores each week. Despite fantasy football being just a game, people become very competitive between friends, family and even people across the world. Not only must players choose a strong team, they must do it with some strict restrictions; however, the combinations of players that can be chosen are still endless.

We create a Java application to assist users in playing fantasy football and provide a tool of convenience for all football fans. Our application has a clean interface with different tabbed sections. Our Fantasy tab enables users to follow our predictions alongside their own team upon signing in, to assess their own predictions. This feature is a visually appealing aspect, similar to the Premier League's own site.

For our prediction, we use a starting team based on the performance of players from the previous season. We choose an effective approach of maximising our budget, by choosing the cheapest players in each position, to sit on the bench for the entirety of the season. We select the rest of the team using a Linear Programming problem, with our constraints being our restrictions in terms of each player's position, team and cost.

Players can make weekly transfers in order to boost their scores for the season. We use various highly correlated variables as our predictors, tracking cumulative scores from the previous games played. We decide the effectiveness of transfers based on these predictors

and advise users on how to manage their team for the season.

Furthermore, our tool also provides users with a player database containing live statistics for each players; we have a live league table and live results and fixtures list.

CHAPTER 2

Fantasy Football: The Game

Fantasy football is a game where players create an imaginary team of real-life footballers and receive points based on the way they play in real-life games. It is the job of the player to pick and manage this team throughout this season, while abiding by the many rules and constraints in place when selecting players. People all over the world compete with the objective of being the player with the highest number of points at the end of the football season.

2.1 Premier League

The Premier League is the highest league in top-flight football within England. As a part of the English Football League (EFL), the Premier League uses a promotion and relegation system where the teams at the top and bottom of the leagues are promoted or demoted respectively. The Premier League consists of 20 teams, containing 25 players over the age of 21 and many players who are 21 years and younger. Each team usually plays one match per week (with subject to change based on other factors).

2.2 Effects of Covid-19

The past few seasons in the Premier League have been slightly different due to the effects of Covid-19 and this season so far has been no different. We have seen many games

being affected by the absence of fans, players and managers, as well as many games being rescheduled or just pushed back completely. This results in some teams playing multiple games per week and sometimes games played at completely different times of the year to when they were supposed to be played.

In regards to Premier League Fantasy football, it will be particularly hard to predict the current season for a few reasons. Firstly, the rescheduling of games has and will have a huge effect on team selection for many weeks as on some occasions, the deadline for selecting players has passed, but the match will not be rescheduled for many weeks/months. This means that we can not take player fitness and form into account for this selection process as these variables will most likely change when the game is played. In the case that games are postponed for many months, there has been changes in how the points system works, where the rescheduled match is added to the gameweek it is played in. This means that some players will play more than one match in a game week and have more of an opportunity to gain points in those weeks. However, as a consequence of this, the cancelled game means that any player selected in that gameweek, from that team, receieves zero points. This has had a huge effect on all players when multiple teams have had games cancelled. From December 12th 2021 to January 18th, a total of 20 Premier League matches have been postponed.

Furthermore, there have been, and will continue to be instances where players who are fully match-fit suddenly test positive for Covid, resulting in them not playing in matches. This is something which can not be accounted for in team-selection as unlike a physical injury players receive in matches and training, it can be completely hidden if a player is asymptomatic.

2.3 Rules

The rules of Premier League Fantasy Football can be quite extensive, however, simple enough for anyone with good football knowledge to understand. There are many rules for many areas of the game, which are all stated on the Premier League website.

Below is a summary of the rules listed:

Rule 2.3.1 *Initial Squad:*

- *Select 15 players, consisting of:*

  - *2 Goalkeepers*
  - *5 Defenders*
  - *5 Midfielders*
  - *3 Forwards*

- *Ensure that the cost of your team does not exceed £100M.*
- *Only 3 players may be chosen from each PL team.*

Rule 2.3.2 *Managing your squad:*

- *Select 11 players before the gameweek deadline. If players do not play, they will be automatically substituted. The team can be in any formation with the following requirements:*
  - *1 Goalkeeper*
  - *At least 3 Defenders*
  - *At least 1 Forward*
- *Choose a captain to have their points doubled and a vice-captain to have their points doubled if the captain does not play.*
- *Automatic substitutions will happen for any player who has played less than 1 minute and receieved no yellow or red cards. The bench is ordered by priority, meaning that if there is an automatic substitution then the first player on the bench who fits the requirements will be subbed on.*

Rule 2.3.3 *Transfers:*

- *Unlimited transfers can be made at no cost before the first deadline. Thereafter, there is 1 free transfer per Gameweek, which can be saved for up to 1 week. You can make up to 20 additional transfers but it will cost 4 points per transfer for that Gameweek.*
- *Once the season has begun, player prices may change depending on the popularity of that player. There is a sell-on fee of 50% (rounded up to the nearest £0.1m) for any profits made.*

Rule 2.3.4 *Chips:*

- *Bench Boost*

  - *Adds the points scored by your bench players in the next Gameweek to your total.*
  - *Can each be used once a season.*
  - *Played when saving your team on the my team page.*
  - *Can be cancelled at anytime before the Gameweek deadline.*

- *Free Hit*
  - *Allows unlimited free transfers which are temporary and for a single gameweek only.*
  - *Can each be used twice during the 2021/22 season due to the disruptions from Covid-19. The second can only be used after Gameweek 20.*
  - *Played when confirming your transfers.*
  - *Cannot be cancelled or used in consecutive Gameweeks.*
  - *Removes saved free transfer from the previous Gameweek.*

- *Triple Captain*
  - *Triples your captain's points.*
  - *Can each be used once a season.*
  - *Played when saving your team on the my team page.*
  - *Can be cancelled at anytime before the Gameweek deadline.*

- *Wildcard*
  - *Unlimited free transfers in a single gameweek.*
  - *Can each be used twice a season. The first is available from the start of the season to Tuesday 28th December at 13:30 and the second is available from then until the end of the season.*
  - *Played when confirming transfers that cost points.*
  - *Cannot be cancelled once played.*
  - *Removes saved free transfer from the previous Gameweek.*

Rule 2.3.5 *Scoring:*

- *The player points system is shown below:*

Action	*Points*
For playing up to 60 minutes	*1*
For playing 60 minutes or more (excluding stoppage time)	*2*
For each goal scored by a goalkeeper or defender	*6*
For each goal scored by a midfielder	*5*
For each goal scored by a forward	*4*
For each goal assist	*3*
For a clean sheet by a goalkeeper or defender	*4*
For a clean sheet by a midfielder	*1*
For every 3 shot saves by a goalkeeper	*1*
For each penalty save	*5*
For each penalty miss	*-2*
Bonus points for the best players in a match	*1-3*
For every 2 goals conceded by a goalkeeper or defender	*-1*
For each yellow card	*-1*
For each red card	*-3*
For each own goal	*-2*

- *Clean sheets are awarded to players who do not concede a goal if they have been on the pitch for 60 minutes. Unless the player received a red card, they will not be penalised for conceding after they have been substituted.*
- *Assists are awarded for the initial shot or pass when a rebound is scored off players, the goalkeeper or the woodwork. If a played win a penalty but does not take it, they are awarded an assist if directly scored.*
- *The three best performing players in each match will be awarded bonus points. 3 points will be awarded to the highest scoring player, 2 to the second best and 1 to the third.*
  - *If there is a tie for first place, Players 1 & 2 will receive 3 points each and Player 3 will receive 1 point.*
  - *If there is a tie for second place, Player 1 will receive 3 points and Players 2 & 3 will receive 2 points each.*

  - *If there is a tie for third place, Player 1 will receive 3 points, Player 2 will receive 2 points and Players 3 & 4 will receive 1 point each.*

- *To calculate the best performing players, the following BPS scoring system is used:*

Action	*BPS*
Playing 1 to 60 minutes	*3*
Playing over 60 minutes	*6*
Goalkeepers and defenders scoring a goal	*12*
Midfielders scoring a goal	*18*
Forwards scoring a goal	*24*
Assists	*9*
Goalkeepers and defenders keeping a clean sheet	*12*
Saving a penalty	*15*
Save	*2*
Successful open play cross	*1*
Creating a big chance (a chance where the receiving player should score)	*3*
For every 2 clearances, blocks and interceptions (total)	*1*
Key pass	*1*
Total tackles minus successful tackles (cannot be negative)	*2*
Successful dribble	*1*
Scoring the goal that wins a match	*3*
70 to 79% pass completion (at least 30 passes attempted)	*2*
80 to 89% pass completion (at least 30 passes attempted)	*4*
90%+ pass completion (at least 30 passes attempted)	*6*
Conceding a penalty	*-3*
Missing a penalty	*-6*
Yellow card	*-3*
Red card	*-9*
Own goal	*-6*
Missing a big chance	*-3*
Making an error which leads to a goal	*-3*
Making an error which leads to an attempt at goal	*-1*
Being tackled	*-1*
Conceding a foul	*-1*
Being caught offside	*-1*
Shot off target	*-1*

CHAPTER 3

Data

The live data we are using is owned by the Premier League themselves and includes data on players, teams, fixtures.

The process of data collection is stated on the Official Premier League website [1]. To summarise this process, live data is collected by two analysts who use a proprietary video-based collection system to retrieve the information. Alongside the two analysts, a quality control analyst can rewind a frame-by-frame video to ensure the data is precise.

The website states that: 'Official Premier League performance data is collected and analysed by Opta, part of Stats Perform (statsperform.com)'

It is further stated that 'All the Premier League data collected is then subject to an exhaustive post-match check to ensure accuracy.'

Definition 3.0.1 *In August 2021, IBM Cloud Education wrote an article [2] titled 'Application Programming Interface (API)' where they gave full explanations on what an API is and how they work. They state that an API 'allows services and products to communicate with each other and leverage each other's data and functionality through a documented interface'*

The data from previous seasons and the start of this season is stored in a GitHub repository, collected and managed by Vaastav Anand. From this repository we use the raw player and previous gameweek data for the past six seasons. The raw player data for the 2021-22 season is from a previous commit, made on 21st July 2021, containing data collected before the start of the first gameweek. All of the data from this repository that we are using is in a csv format; we will deal with this accordingly.

All our data handling and manipulation will be done in Python, including accessing the data from the API.

3.1 API Access

The data we are using is found at the Premier League's API, and with the correct endpoints added to the base URL 'https://fantasy.premierleague.com/api/', a request can be made. Each endpoint contains data correlating to a different aspect of fantasy football.

3.2 Data Retrieval

The retrieval of the live data is done in Python using an external library called 'Requests'. This is a library is designed to allow a user to request data from a URL and access the data in Python.

Example 3.2.1 *r = requests.get(url)*

We make requests to the Premier League's API to get the data from the endpoints, and store the data as a Python dictionary (JSON structure) with the following command:

Example 3.2.2 *data = r.json()*

We then convert the raw data into a Pandas' DataFrame, ready to be preprocessed. Throughout our data manipulation we will be using the Pandas library. Our previous season data is retreived from the GitHub repository and downloaded before-hand, as this data will never change. Retreiving this data from our local file system is easily done using pandas' read_csv() method. An example of this is shown below:

Example 3.2.3 *dataframe = pd.read_csv("/file/at/this/dir/example.csv")*

This method converts csv data to a pandas dataframe with the column headers as the top row of the csv file. The rest of the data is split into columns and rows; the rows are given numerical indexes starting from 0 and increasing by 1 every row.

3.3 Data Management

Managing the retreived data is so important as it enables us to easily store and retrieve any data effectively and when necessary. One issue we have when sending a request to the API is that very rarely, the data at the API is unavailable. This will occur when the API is being updated, if the website is down, or if a request is made without a stable connection to the internet.

In order to solve this issue, we store a copy of the API data in a JSON file on the users' local file system. This copy is updated whenever a request to the API is successful, to ensure that the data we are using is reasonably up to date, even when the API can not be accessed.

Code Block 3.3.1 *fetchData function*

```
def fetchData(url, fileName):
try:
    r = requests.get(url)
    data = r.json()
    storeData(data, fileName)
    return data
except:
    return fetchStoredData(fileName)
```

We store the data in a JSON file, as it matches the file type of the API data and can easily be read by both Python and Java. This makes data retrieval effortless for both the front and back-end of the program. JSON files are considered to be one of the best formats to store data, due to it's low file size and readability.

As shown in code block 3.3.1, we use a 'try/except' when making a request from the API. If the request is successful, we call the 'storeData' function.

Writing data to a JSON file in Python is very straight-forward, as shown in the following example of the storeData function:

Code Block 3.3.2 *storeData function*

```
def storeData(data, fileName):
file = open(fileName, "w")
```

file.write(json.dumps(data))

file.close()

We use Python's built in File I/O (Input/Output) functions to write to a new JSON file. The current data in the file is completely overwritten when this function is called, as we state that we only wish to write to the file, not read the contents, as we are using the parameter '"w"'.

We use the 'json.dumps()' function to write the JSON data to the JSON file. This function is from the JSON module which we import into our Python script.

However, if the request in the code block 3.3.1 is unsuccessful, the 'except' block is activated and we call the 'fetchStoredData' function in the example below:

Code Block 3.3.3 *fetchStoredData function*

```
def fetchStoredData(fileName):
try:
    file = open(fileName)
    return json.load(file)
except:
    return
```

This example tries to open the JSON file and return the data as a Python dictionary using 'json.load()', replicating the API request but to the local file system. However if this is unsuccessful, there is no available data for the program to use.

CHAPTER 4

Preprocessing

Preprocessing is vital when working with raw data, in order to have the data in an understandable form, to extract and use meaningful information [4]. The steps involve amending the data to the point where there is no incomplete, inconsistent or outlying values which could cause errors when using the data. Typically, most sets of data will have at least one of these issues; this dataset is no different.

4.1 Analysing the data

For our prediction and data visualisation in our application, we use data from a few different datasets. We can look at this data to see what we can gain from each dataset and what it can be used for. We gain understanding of our data by reading descriptions of our API endpoints [6].

Live player general information

This data gives us some of the most important information about each player in the league, providing 3 ways of referencing each play in other datasets. This is live data which is always updated, however the data is always overwritten making it less useful for training or testing.

Useful data:

- Total number of players

- Player names and codes/ids
- Players' positions
- Players' teams
- General statistics (e.g. goals scored, assists, saves etc.)
- Total points
- Player cost for visualisation tables(useless for train/test data)

Player general information across seasons

This data is the same data given to us by the API, however it is stored data collected from the end of each season. For this current season, we are using the data collected at the very start of the season to help us when choosing our initial team. This data is not live.

Useful data:

- Total number of players each season
- Player names, codes and ids
- Players' positions (For that season)
- Players' teams (For that season, allows us to track transfers)
- General statistics (e.g. goals scored, assists, saves etc.)
- Total points
- Player cost for visualisation tables (Start and end cost per season and start cost of this season)

Upcoming and previous fixture details

This is data which we can use in our fixture and and results tables in our application. The data is live, but conveniently not overwritten until the start of a new season.

Useful data:

- Fixture dates and times
- Home and away team involved
- Previous fixture results

Previous fixture details across all seasons (player oriented)

This is data which we can use in our fixture and and results tables in our application. The data is live, but conveniently not overwritten until the start of a new season.

Useful data:

- Fixture dates and times per gameweek
- Home and away teams involved
- Fixture scores and results
- Player stats in each gameweek (e.g. goals scored, assists, points per gameweek etc.)
- Influence, creativity and threat (ICT) for each match
- Position and team played for (helps track transfers across this season)

Team names and ids across each season

This allows us to link our player ids to names across each season when using data from previous seasons. For example, when as teams are promoted and relegated, every team is assigned an id from 1 to 20 based on alphabetical order. Some teams may keep the same ids, but others may get new ids which are linked to other teams in other seasons. Using multiple index keys, 'season' and 'id', we can re-define how we reference each team by mapping either team names or codes.

Manager Basic Info

We can get basic information on each manager from the Premier League's API if we know their manager id for their FPL account.

Useful data:

- Manager's name
- When the manager started FPL
- Total points for the season
- Overall rank this season
- Manager's team value
- Manager's FPL bank balance

Manager's Gameweek Stats

We can get history on each manager's team performance from the Premier League's API if we know their manager id for their FPL account.

Useful data:

- Points per gameweek
- Cumulative points up until that gameweek
- Gameweek rank
- Manager's FPL bank balance after than gameweek
- Team value
- Amount of transfers made that week
- Weekly transfer cost
- Points lost from unplayed players

Manager's teams per gameweek

This data gives us a manager's team list for each gameweek.

Useful data:

- Team list
- Which players are starting and which are bench players
- Which players are captain and vice-captain

4.2 Fixing missing data

Before we begin to fix our missing values, we create some helper methods. These are repetitive actions we use when fixing our data. These helper methods are shown below:

Code Block 4.2.1 *manipulation.py file - Helper methods*

```
#removes whole column from data frame if column has any missing values
def dropEmptyValueColumn(dataFrame):
    try:
        dataFrame.dropna(inplace=True, axis=1)
    except:
        pass
#removes whole row from data frame if row has any missing values
def dropEmptyValueRow(dataFrame):
    try:
        dataFrame.dropna(inplace=True, axis=0)
    except:
        pass
#removes whole column from data frame by name
def dropColumnByName(dataFrame, columnName):
    try:
        dataFrame.drop(columnName, inplace=True, axis=1)
    except:
        pass
#replaces empty values in columns from the data frame with new value
def fillEmptyValueColumn(dataFrame, newValue):
    try:
        dataFrame.fillna(newValue, inplace=True)
```

```
except:
    pass
```

These helper methods can be called by using 'manipulation.' and then the name of the function, from any other class.

We attempt to merge our player fixture data from previous seasons and the live data from this season, however, the following data is missing which we fix:

- 'GW' (Which gameweek the data is from) - Previous fixtures

We take a count of the gameweeks and create a new column storing this count to solve this.

Code Block 4.2.2 *Fix 'GW' column*

```
dfs = []
gwCount = 1
for dict in dicts:
    df = pd.DataFrame(dict, index=[0])
    df['GW'] = gwCount
    gwCount += 1
```

- 'GW' (Which gameweek the data is from) - Upcoming fixtures
- 'was_home' (Whether the player's team was home or away) - Upcoming fixtures
- 'fixture' (The ID of the fixture) - Upcoming fixtures

These three missing columns technically exist in our data, they are just stored under different column names. We use pandas' .rename() method to name the columns correctly.

Code Block 4.2.3 *Rename method*

```
df = df.rename(columns={'event':'GW', 'is_home':'was_home', 'id':'fixture'})
```

- 'element' (The ID of the player)

This is stored as an outer key in our JSON data which we can retreive using the .items() method.

Code Block 4.2.4 *Fix 'element' column*

```
for element, dicts in fixtureDict.items():
    for dict in dicts:
        temp = pd.DataFrame(dict, index=[0])
        df['element'] = element
```

- 'opponent_team' (The ID of the opponent's team)

We have data on the two teams, however we do not know which team is the opponent. However, we have whether the player played at home or not so we can determine which team is the opponent by choosing either the home or away team. We use the .apply() method to apply a lambda function to set the value on a row by row basis.

Code Block 4.2.5 *Fix 'opponent_team' column*

```
df['opponent_team'] = df.apply(lambda x: getOpponentTeam(x.was_home, x.team_h, x.team_a), axis=1)
def getOpponentTeam(home, team_h, team_a):
    if home:
        return team_h
    else:
        return team_a
```

Now we have fixed these columns, we check for missing values using the following code:

Example 4.2.6 *df['your column name'].isnull().sum() [8]*

The only missing data we have in this dataset is in our future fixtures. We can fix this by using our manipulation helper functions that we created. Our missing data can be switched from NaN values to 0. We only have missing values in the statistics-related columns as the fixture has not occurred yet.

Code Block 4.2.7 *manipulation.fillEmptyValueColumn(fixture_df, 0)*

We then combine our player and fixture dataframes for each season, but come across the issue of duplicated rows as the same fixtures are played by the same players, just in different years. To solve this, we need to add data a new 'season' column which can be done using the following code:

Code Block 4.2.8 *Fixing missing 'season' column*

```
seasons = ['1617', '1718', '1819', '1920', '2021', '2122']
for i in range(len(seasons)):
    player_df_list[i]['season'] = seasons[i]
    gw_df_list[i]['season'] = seasons[i]
```

Any columns currently in our player and fixture dataframes with missing data are unneccessary and useless for our prediction. We take each key, check for NaN values and use our manipulation.dropColumnByName() method to remove the columns:

Code Block 4.2.9 *Remove columns with missing values*

```
for col in data.players_df.keys():
    if data.players_df[col].isnull().values.any():
        dropColumnByName(data.players_df, col)
```

Now that our player and fixture datasets have been combined, there are a few more columns we are missing; 'starting_cost', 'cost_bin' and 'full_name'. We can calculate our start cost using our current cost ('now_cost') and the amount the cost has changed since the start ('cost_change_start_fall'):

Code Block 4.2.10 *Adding 'start_cost' values*

```
players_df['starting_cost'] = players_df.now_cost - players_df.cost_change_start_fall
```

The issue with our 'start_cost' data is that it is 10 times the actual cost of players in FPL. Our 'cost_bin' column is the correct cost, calculated by dividing by 10 using a lambda function:

Code Block 4.2.11 *Adding 'cost_bin' values*

```
data.players_df['cost_bin'] = data.players_df.now_cost.apply(lambda x: np.floor(x/10))
```

To make mapping easier between our players and fixtures dataframes, we can use players' full names, which we can obtain by concatenating players' first and last names. Furthermore, for stopping issues with our prediction, we insert underscores instead of spaces. Moreover, we lower the case of the values in the column and use our 'unidecode' package [10] to deal with the unicode values.

Code Block 4.2.12 *Getting 'full_name' column*

```
data.players_df['full_name'] = data.players_df.apply(lambda x: get_full_name_playerdf (x.first_name, x.second_name), axis = 1).str.lower()
data.gameweek_fixture_df['full_name'] = data.gameweek_fixture_df['full_name'].str.replace(" ", "_").str.replace("-", "_")
data.gameweek_fixture_df['full_name'] = data.gameweek_fixture_df['full_name'].apply(lambda x: unidecode.unidecode(x))
data.gameweek_fixture_df['full_name'] = data.gameweek_fixture_df['full_name'].str.lower()
```

Finally, we can get our final missing values for our fixture dataframe before our prediction, 'team_points' and 'opponent_points':

Code Block 4.2.13 *Getting team points*

```
data.gameweek_fixture_df['team_points'] = data.gameweek_fixture_df.apply(lambda x: get_team_points (x.was_home, x.team_h_score, x.team_a_score), axis = 1)
def get_team_points(home, home_score, away_score):
    if home_score > away_score:
        if home:
            return 3
        else:
            return 0
    if home_score < away_score:
        if home:
            return 0
```

```
        else:
            return 3
    else:
        return 1
```

Once we have each player's team's points we can use this to get the opponent's team's points:

Code Block 4.2.14 *Getting opponent's team's points*

```
data.gameweek_fixture_df['opponent_points'] = data.gameweek_fixture_df.team_points.apply(lambda x: get_opponent_points(x))
def get_opponent_points(team_points):
    if team_points == 3:
        return 0
    elif team_points == 1:
        return 1
    else:
        return 3
```

4.3 Fixing data-types

When we combine our fixture data, we have the issue that some of our numerical data is stored as an object type as opposed to a float. We fix this using the following code:

Code Block 4.3.1 *Converting object to float type*

```
gws_df = gws_df.astype({'influence':'float64'})
gws_df = gws_df.astype({'creativity':'float64'})
gws_df = gws_df.astype({'threat':'float64'})
gws_df = gws_df.astype({'ict_index':'float64'})
```

We deal with our 'position' column in our player dataframe by mapping our data to convert our numerical position values to categorical object values using our map() function.

Code Block 4.3.2 *Converting 'position' from numerical to categorical type*

```
data.players_df['position'] = data.players_df.element_type.map({1 : 'Keeper', 2 : 'Defender', 3 : 'Midfielder', 4 : 'Forward'})
```

CHAPTER 5

Predictions

The approach we are taking to our prediction is to choose a starting team by treating the selection as a Linear Programming problem. We train a Linear Regression model using cumulative fixture data of last season. This model provides us with ranks for each player for this season and we choose transfers based on these ranks, sticking to our selection restrictions. Our approach is based on a Kaggle project with some of the code being reproduced [11].

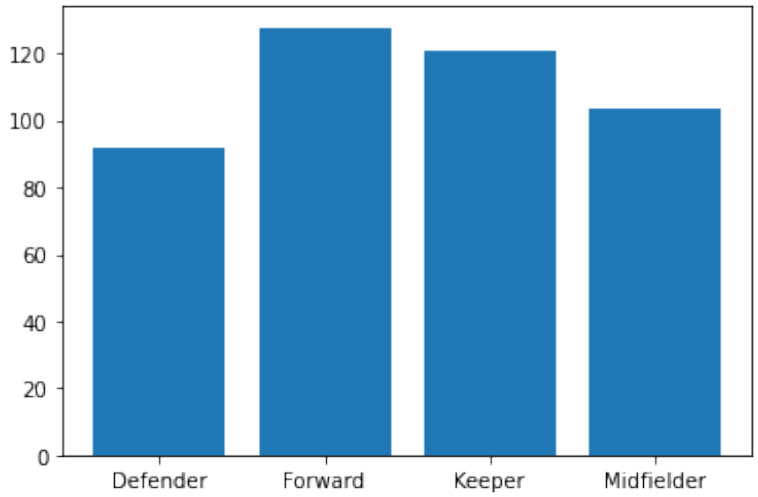
5.1 Preliminary Analysis

We look at the average total points over the past 6 seasons of players who have played at least half of the matches in that season. These averages are grouped by position as shown below:

Code Block 5.1.1 *Average 'total_cost' per position*

```
values = []
positions = []
for pos, players in data.players_df.groupby(by='position'):
    players = players[players['minutes'] > ((38*90)/2)]
    positions.append(pos)
values.append(players.total_points.sum() / len(players))
plt.bar(positions, values)
plt.show()
```

Image 5.1.2 *Average 'total_points' per position*

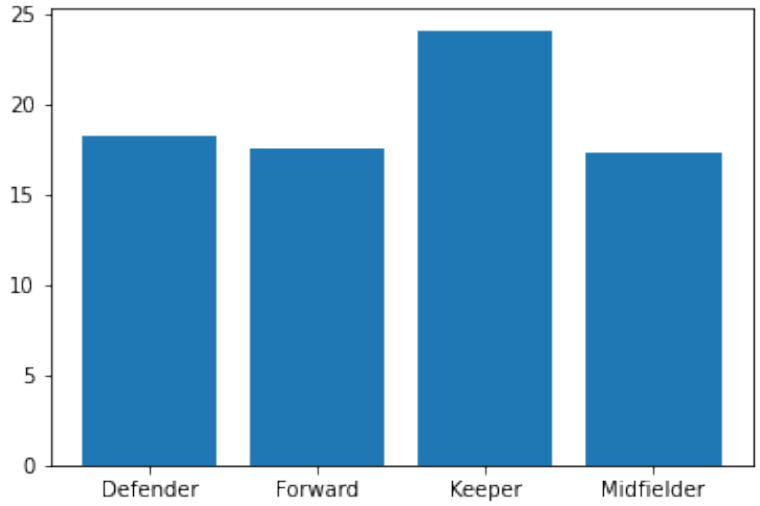


The average value of players over the years proves to us that despite forwards being the most likely to gain points over the season, the question is 'at what cost?'. We can look at the average 'value_season' value per position for a better idea of the actual value of each position with cost involved:

Code Block 5.1.3 *Average 'value_season' per position*

```
values = []
positions = []
for pos, players in data.players_df.groupby(by='position'):
    players = players[players['minutes'] > ((38*90)/2)]
    positions.append(pos)
values.append(players.value_season.sum() / len(players))
plt.bar(positions, values)
plt.show()
```

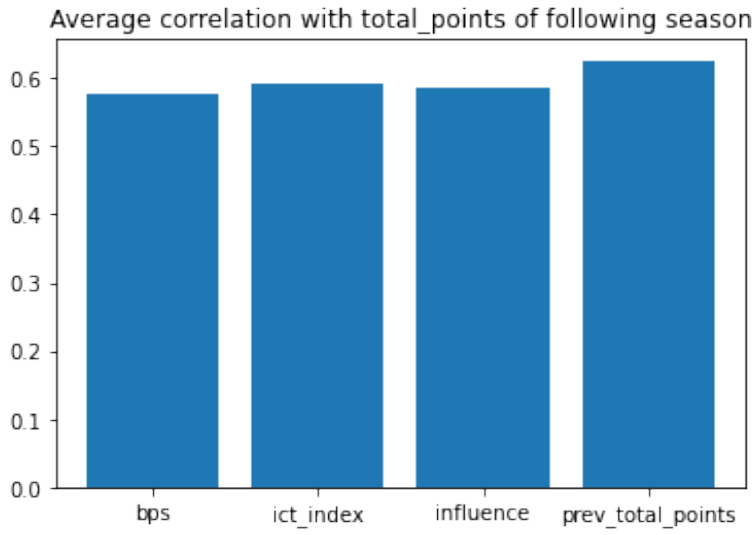
Image 5.1.4 *Average 'value_season' per position*



When looking at the above figure, it seems clear that the goalkeeper is the most valuable position. However, this data is relatively skewed as the number of goalkeepers who have played over half of the games is very minimal compared to every other position as there is only one goalkeeper spot per team and they are very rarely substituted. This means that we are taking the average values of the top goalkeeper at each club.

As we plan to predict the total_points of the following season, we can use our past data to check the correlation between each of the variables in one season with the total_points value in the following season.

Image 5.1.5 *Average correlation with total_points of following season*



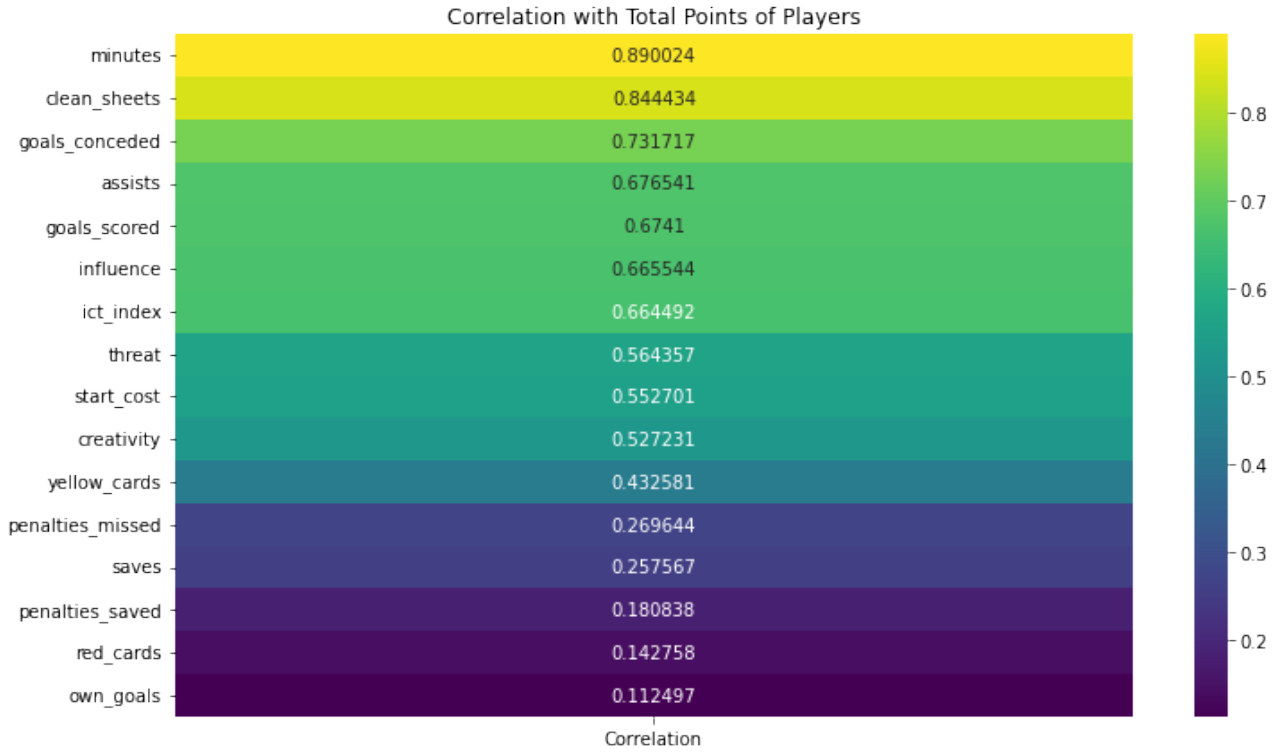
We took the top correlations of the following variables:

'assists', 'bonus', 'bps', 'clean_sheets', 'creativity', 'goals_conceded', 'goals_scored', 'ict_index', 'influence', 'minutes', 'own_goals', 'penalties_missed', 'penalties_saved', 'points_per_game', 'red_cards', 'saves', 'selected_by_percent', 'team_code', 'threat', 'prev_total_points', 'transfers_in', 'transfers_out', 'value_season', 'yellow_cards', 'season_num', 'starting_cost', 'cost_bin'

Clearly, 'prev_total_points' has the highest correlation and is the best single predictor variable.

When predicting player performance for each week, it is vital that we know which variables have the highest correlation with the 'total_point' variable in the same season. We test 16 of the main variables in our data across multiple seasons and receive the following results:

Image 5.1.6 *Average correlation with total_points of the same season*



Of these variables, we look at the top 10 variables that have a correlation of over 50% with our target variable. These can be used for our week-by-week predictions.

5.2 Starting team

The first step to choosing our starting team is to establish the players we can choose from. From our 'players_df', we can get data from the current and previous season by refining our dataframe:

Code Block 5.2.1 *Getting current and previous season data*

```
CURRENT_SEASON = 2122
LAST_SEASON = 2021
data.current_season_player_df = data.players_df[data.players_df.season == appconstants.CURRENT_SEASON]
data.previous_season_player_df = data.players_df[data.players_df.season == appconstants.LAST_SEASON]
```

The players we are choosing from are only from this current season, however, we use the data from last season's dataset to help predict our starting team. We rename our 'total_points' columns from last year's data to 'prev_total_points' and select the important data from that dataframe; 'full_name' and 'prev_total_points'. We then proceed to merge this data into this season's data using the player's name as our key (we only care for the players who played last season):

Code Block 5.2.2 *Getting last season's total points for this season's players*

```
data.previous_season_player_df.rename(columns = {'total_points' : "prev_total_points"}, inplace = True)
data.previous_season_player_df = data.previous_season_player_df[data.previous_season_player_df.minutes > 0][['full_name', "prev_total_points"]]
available_players_df = pd.merge(data.current_season_player_df, data.previous_season_player_df, on='full_name', how='left')
```

Furthermore, any player who did not play in the previous season, we take the mean average of the players in their position at their cost and set this is their value. Some players did not player last season and do not have any players in their position at their cost. We can not make a prediction for these players so they are dropped from our dataframe using our manipulation helpers.

Code Block 5.2.3 *Getting last season's total points for players who did not play*

available_players_df.prev_total_points = available_players_df.groupby(['position', 'cost_bin']).prev_total_points.transform(lambda x: x.fillna(x.mean()))

manipulation.dropEmptyValueRow(available_players_df)

To maximise our budget, we will be choosing the least expensive players from each position who have the most points from last season as our substitutes. We do not care if the player is physically fit or that they are a starter for the team as they are simply there to stay. Maximising our budget for the starting 11 players gives us the best advantage of gaining the most points as the players who are on the bench score zero points no matter what. The only reason these four players may count is if an automatic substitution is made when a starting player is unavailable for the given match.

Using our insights we gained from our preliminary analysis, forwards are most likely to gain points, however, they also cost more. The difference in value between each position is minimal, except for goalkeepers, but we must have 1 goalkeeper on the bench no matter what. So we choose the player with the highest points in the previous season from the players with the lowest cost:

Code Block 5.2.4 *Choosing our substitutes*

for position, players in player_df.groupby('position'):

bench_player = cheapest_players[cheapest_players.total_points == cheapest_players.total_points .max()]

bench_player_name = bench_player.full_name.values[0]

bench_player_names += [bench_player_name]

The players we have chosen are:

- Watford's second choice keeper 'Daniel Bachmann', a new summer transfer who hasn't played a single match in the Premier League.
- Liverpool's defender 'Konstantinos Tsimikas' who played a total of 6 minutes in the last season scoring 2 points in total
- 'Fabian Delph', a Premier League veteran playing for Everton in the midfielder

- Brighton forward 'Jürgen Locadia' who didn't play last season but played in the Premier League the three seasons before, scoring an average of 23 points, underwhelming for a forward.

These players cost a total of £15.4M, leaving us with a budget of £84.6M for the remaining 11 players.

Now we can move on to selecting the players for our starting XI, however, we must adjust the dataframe of players to choose from. In contrast to our bench players, our starting XI is where we need the most fit players who will be sure to gain points. Therefore, we ensure that the players we are choosing from are available to play and not suffering from an injury or have been loaned or transferred out of their original club. We can check which players this includes using our 'status' variable which states categorical value 'a' for available; we want to ignore all 'i' - injured, 'u' - unavailable and 's' - suspended players:

Code Block 5.2.5 *Selecting only fully available players*

```
starting_available_players = data.available_players_df[data.available_players_df.status == 'a']
starting_available_players = starting_available_players.reset_index(drop = 'index')
```

We will be using Python package PuLP [12] to create a Linear Programming (lp) Problem to select 11 of our players as lpVariables.

Code Block 5.2.6 *Creating our Linear Programming problem and player options*

```
lp = pulp.LpProblem('StartingXI', pulp.LpMaximize)
for player in player_df.full_name:
    lp += pulp.LpVariable(i, cat = "Binary")
```

Now we have our players to choose from, we can address our restrictions which come from our rules.

- Restriction 1 - Positions

As shown in our initial squad rules, after taking into account our selection of one player from each position already, we must pick one goalkeeper, four defenders, four midfielders and two forwards. To do this we create a restriction for each position and limit the options to

combinations of the amount of players we should have for that position. If the combination being tested has the correct amount per position, we return 'True', else we return 'False' and that combination is not valid.

Code Block 5.2.7 *Position restriction*

```
pos = ['Keeper', 'Defender', 'Midfielder', 'Forward']
amount = [1, 4, 4, 2] # Amount of players per position
for i in range(4):
    lp += position_restriction(pos[i], amount[i], options, starting_available_players)
def position_restriction(position, n, options, player_df):
    total_n = "" # Count of players in position
    player_positions = player_df.position
    for i, player in enumerate(options):
        if player_positions[i] == position:
            total_n += 1 * player
    return(total_n == n) # Returns True if correct amount of players per position, else False
```

- Restriction 2 - Cost

Also stated in our rules, for our initial squad, we must not exceed £100M. As we have already spent £15.4M on our bench, we adjust our budget to £84.6M and ensure that that combination of players for our initial squad does not exceed this value.

This code is very similar to our position restriction, except across the whole team.

- Restriction 3 - Team

Furthermore, our rules state 'only 3 players may be chosen from each PL (Premier League) team'. We create a similar restriction method to our others in order to count the amount of players from each club. Obviously we have to think about our bench players also as they count towards our maximum per club.

Code Block 5.2.8 *Club restriction*

```
for team, group in starting_available_players.groupby('team_code'):
```

```
    prob += team_restriction(lp, starting_available_players, options, bench_players)
def team_restriction(prob, player_df, options, bench_players):
    team_total = ''
    for player in bench_players:
        if player.name in group.full_name.values:
            team_total += 1 * player
    for player in options:
        if player.name in group.full_name.values:
            team_total += 1 * player
    return (team_total <= 3)
```

- Restriction 4 - total_points restriction

Obviously the total amount of points a team gets is not a restriction in any way, as players would not be stopped from choosing a team, just because they didn't receive very high scores last year. However, we treat this like our other restrictions in terms of our LP Problem, we want to use this variable as our main predictor. As shown by our preliminary analysis, of any variable in our player data from the past six years, the previous season's points total is the variable with the highest correlation to the following season's points total. This is the reason we use this as our predictor.

Now we have created our LP Program we can solve it:

Code Block 5.2.9 *Solving LP Problem*

```
Code:
lp.writeLP('StartingXI.lp')
lp.solve()
Output:
...
Result - Optimal solution found
Objective value: 2017.00000000
Enumerated nodes: 0
Total iterations: 3
```

Time (CPU seconds): 0.02

Time (Wallclock seconds): 0.02

Option for printingOptions changed from normal to all

Total time (CPU seconds): 0.02 (Wallclock seconds): 0.03

Now that our LP problem has been solved, we can get our starting team by finding the chosen players by our problem as a dataframe:

Image 5.2.10 *Starting team dataframe*

	full_name	position	now_cost	player_team_name	total_points
0	aaron_cresswell	Defender	54	West Ham	102
3	aaron_wan_bissaka	Defender	51	Man Utd	39
27	andrew_robertson	Defender	73	Liverpool	148
60	bruno_miguel_borges_fernandes	Midfielder	116	Man Utd	134
133	emiliano_martinez	Keeper	55	Aston Villa	105
165	harry_kane	Forward	125	Spurs	140
177	heung_min_son	Midfielder	109	Spurs	177
192	jack_harrison	Midfielder	55	Leeds	88
381	ollie_watkins	Forward	75	Aston Villa	96
457	stuart_dallas	Midfielder	49	Leeds	73
481	trent_alexander_arnold	Defender	84	Liverpool	187

Following the rules and the restrictions given, we can see that our Linear Program problem has produced a viable solution.

As our sole predictor, it is only right that we select the player with the most points last season as our captain for the season, Manchester United's midfielder, Bruno Fernandes with a score of 244 points last year.

As of gameweek 32 of the 2021-21 season, this team without any transfers produces a score of 1423 points. This score is not entirely great, but given that no changes are made to the team, that is to be expected.

5.3 Weekly Transfers

Managing our weekly transfers is one of the most effective ways to gain points throughout the FPL season. Without transfers, our team can only be based on last season's data. However, as time passes, the correlation between last season's data and this seasons results will start to diminish. The best way of predicting a result over time is by using time series techniques such as collecting cumulative data.

Using the variables from our preliminary analysis where we assess the correlations between many potential predictors and our target variable, we can create data at different periods of time across the season. Our plan is to track the data for one, three and five weeks since each gameweek, across the following predictors:

- minutes
- clean_sheets
- goals_conceded
- assists
- goals_scored
- influence
- ict_index
- threat
- creativity

We will also create cumulative variables using the entire season's data (from the start to that gameweek). We can not use our 'start_cost' variable as a predictor as we can not create a cumulative total.

Our cumulative variables are created by grouping our dataset by each gameweek and for each time period from the gameweek, we merge our dataset and take a sum of the values of each variable.

Once this data is created, we create a training dataset using the data from every season except the current and a testing dataset from only the current.

Once we have created our train and test data and dealt with our categorical variables, we create a linear regression model using Python package sklearn [14]. We then fit our training dataset to the model and apply the model to our test data:

Code Block 5.3.1 *lin_reg = LinearRegression() lin_reg.fit(data.X_train, data.y_train) linreg_predictions = lin_reg.predict(data.X_test)*

Our players now have a rank based on our linear regression model we fitted and we can assess which transfer could and should be made.

We have chosen to use a linear regression model for our prediction due to our need to only predict a single dependent variable, however, as we are predicting football, there are bound to be many outliers in our data which a linear regression model would be very sensitive to.

We start with our initial team and go through each position to find the weakest player in terms of rank. We then get all the potential players we could transfer in for our lowest ranked player and remove any players who have not been playing recently and who are within budget. For each position we compare the best transfer within our constraints and assess which player makes the most difference in terms of rank and update our budget and team list once the most effective transfer has been decided. Some gameweeks we do not make a transfer if our lowest ranked players do not have suitable replacements. One thing that should be mentioned is that we will not transfer any goalkeepers in or out of the team as it is a waste of a transfer as the points difference between each goalkeep is minimal compared to players in other positions.

Our transfers that we make for the first 33 weeks of the season have been decided:

Image 5.3.2 *Transfer list*

*** Transfers ***

	GW	player_in	player_out	money_change
0	1			0
1	2	wout_weghorst	ollie_watkins	9
2	3	joel_matip	aaron_wan_bissaka	5
3	4	jack_grealish	heung_min_son	21
4	5	virgil_van_dijk	joel_matip	-15
5	6	marcos_alonso	aaron_cresswell	-3
6	7	joao_pedro_cavaco_cancelo	marcos_alonso	-1
7	8			0
8	9	jarrod_bowen	jack_harrison	-6
9	10	ruben_santos_gato_alves_dias	joao_pedro_cavaco_cancelo	2
10	11	reece_james	ruben_santos_gato_alves_dias	3
11	12	bernardo_mota_veiga_de_carvalho_e_silva	jack_grealish	6
12	13	leandro_trossard	jarrod_bowen	-1
13	14	cristiano_ronaldo_dos_santos_aveiro	harry_kane	-2
14	15			0
15	16			0
16	17	harry_kane	cristiano_ronaldo_dos_santos_aveiro	4
17	18	diogo_jota	leandro_trossard	-18
18	19	raheem_sterling	bruno_miguel_borges_fernandes	10
19	20	kevin_de_bruyne	raheem_sterling	-13
20	21			0
21	22			0
22	23			0
23	24	jarrod_bowen	diogo_jota	16
24	25	ruben_santos_gato_alves_dias	reece_james	-1
25	26	joao_pedro_cavaco_cancelo	ruben_santos_gato_alves_dias	-9
26	27			0
27	28	hakim_ziyech	jarrod_bowen	-3
28	29			0
29	30	jack_grealish	hakim_ziyech	-2
30	31			0
31	32	luke_cundle	stuart_dallas	5
32	33			0

Now that we have come up with our transfer list for the season so far, we can choose our captain. We base this choice once again on the player with the most points gained in the previous season. However, with transfers being made every week, the captain may change at anytime, unlike our initial team prediction. Furthermore, we choose our vice-captain as the player with the second most points last season.

Code Block 5.3.3 *team_df.loc[team_df.full_name == team_df[team_df'position'] != 1].sort_values("prev_total_points", ascending = False).head(1).full_name.values[0], 'is_captain'] = True team_df.loc[team_df.full_name == team_df[team_df'is_captain'] == False].sort_values("prev_total_points", ascending = False).head(1).full_name.values[0], 'is_vice_captain'] = True*

5.4 Analysis of Prediction Results

When we take into account our transfers over the weeks and taking the total scores of each week, taking into account the captain 2x multiplier, we have managed to get a score of 1628 after 32 gameweeks. From our inital team, this is an increase of 205 points, a 14% increase.

In the grand scheme of things, this score is not the greatest and is nowhere near the top of the FPL leaderboards. However, with the way this footballing season has been so far with the effects of Covid-19, game cancellations, and the fact that we have so many more techniques we could utilise to improve this prediction, this prediction can not be classed as a failure.

Furthermore, there are so many improvements that can be made to the overall prediction. Especially as we have not even looked at using our wildcards which would give us a huge boost if used correctly. Also the fact that we did not take into account injuries or players who did not take part in the match for some reason.

The most important thing to take away from this prediction is the fact that there is an improvement between our initial team with and without transfers.

CHAPTER 6

Graphical User Interface

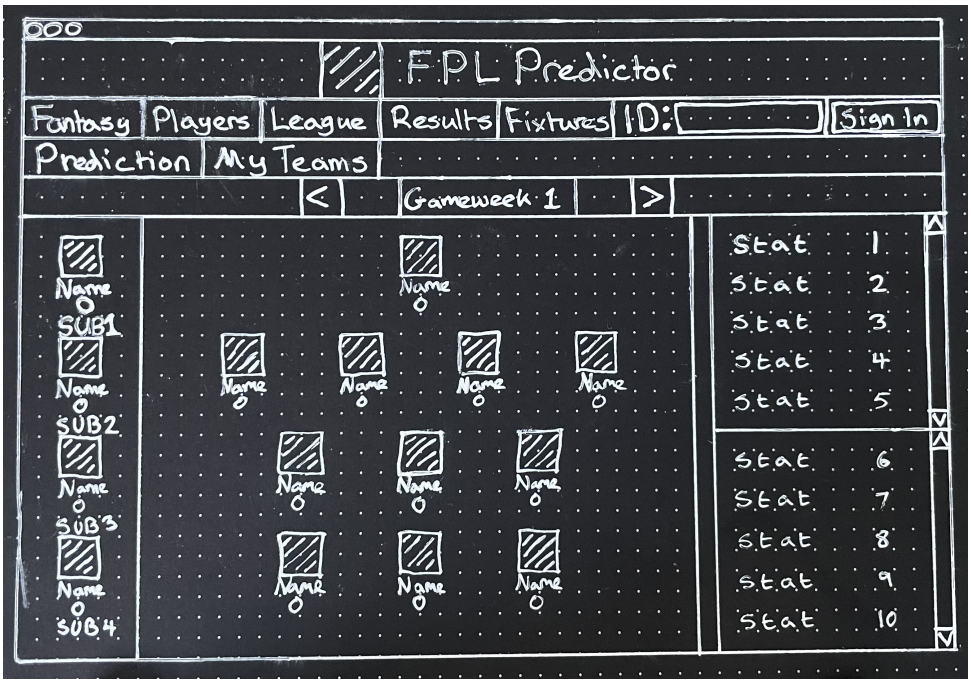
The front-end of this product was produced in Java, as planned in my project outline. The whole user interface was then created using the built-in library 'Java Swing'. We made sure to draw out plans for the design of the visual-side of the product beforehand.

6.1 Design

The overall design decided upon was drawn as a wireframe of the main page. This included the top of the program which would be shown across all of the tabs and also the fantasy tab which shows players on a pitch. The design is hand-drawn as it was the easiest way to get the ideas across in the quickest way possible that was also visual. Despite popular belief, we can still keep the accuracy of the drawing by using paper which has a mapped grid of dots, in order to keep the measurements as simple as they can be.

Below is the diagram containing the design for the main page and fantasy tab:

Image 6.1.1 *Wire-frame Design*



The wire-frame design above shows exactly how we plan to build the GUI and 'Fantasy' tab in our front-end. At the top of the design we can see the title 'FPL Predictor' (Fantasy Premier League Predictor) and to the left, a dashed out square indicating the place for a logo. For now, we plan for that to be the Premier League Logo as this is not for commercial use.

Below this we see five rectangles containing the names of our tabs; the Fantasy tab is the tab currently being shown. In the top right we see a text-box followed by a 'Sign in' button, which will be where the user enters their FPL manager ID to sign in and access their teams.

Below the main tabs, we see two sub-tabs; 'Prediction' and 'My Teams'. Both of these tabs will contain similar content, which is shown in the rest of the design, but containing data based on their tab's name; the 'Prediction' tab will show predicted teams and 'My Teams' will show the user's previous teams.

Teams will be split by each gameweek and will have left and right buttons to switch between the different weeks, as shown in the design. Similarly, the upper half of the stats on the right hand side will also change per gameweek to show the statistics of that current week. However, the lower stats will be overall stats; these will be the same stats showing, no matter whaich gameweek the user is viewing. Moreover, both of these stat tables will have scroll bars in order to view all of the data if there are too many stats to show on one screen.

In the centre of the design we can see eleven dashed out squares which represent players. The squares can be in any valid formation and contain the player image, name and points tally for the gameweek.

On the left hand side of the design, we see four more dashed squares which represent the substitutes, similarly to the other players, the player's image, name and points tally will be shown. Moreover, despite it not being visible in the design, we will also include an indicator as to whether a player is a captain or vice-captain.

Finally, we also plan to have a list view for the teams to give the user multiple ways to view with ease. A similar approach will be adopted when showing the data for the other tabs as they will all require list-like views for what they intend to show.

6.2 Java Swing

Java Swing is a built-in Java package which can be used to add widgets containing data onto a frame. The main reason why we chose to use this package is it's ability to change it's look and feel depending on the platform it is being used on.

Another reason why I chose to use this package is because of IntelliJ's bulit-in GUI designer. This allows users to create a form containing Swing components in a pane, instead of a frame. Rather than programming a full Swing frame and components from scratch, the Form Designer feature lets users design a basic layout by dragging default widgets into a pane. The layout is naturally split evenly ino columns and rows, for example, if we have two rows and two columns, the pane is split into quarters. Using the 'Fill' align options, the component stretches to fill the space it is in. This feature is so useful when designing a program as it enables us to keep a balenced shape and avoid uneccessary spaces.

Java Swing has many other features which make it more than suitable for creating a user interface, such as it's 'action listener's. These are features that can be added to a component to listen to any actions made by the user. For example, an action listener may be added to a button to activate a function when the button is clicked. We have used these throughout our program and an example of how this is done is shown below:

Code Block 6.2.1 *Add Action Listener to Button*

```
button.addActionListener(e -> {
    do something
}
```

As shown in code block 6.2.1, it is possible to create a function, add it to an action listener, and then add the action listener to a component. It is possible to use many different listeners,

such as mouse listeners which track where your device's mouse is on the screen. However, we do not currently have any features which would require these types of listeners, but these could be added in the future. For now, we will stick to using action listeners.

One of the main Swing components we have used in our program is a JTable, which in conjuction with a JScrollPanel, is a very useful component. The purpose of a JTable is to work like any other table by storing data in rows and columns. This is something which we have used throughout all the tabs in the program. In order to store data in a JTable, we make use of the 'DefaultTableModel' component, with the parameters being the data and column headers. Furthermore, we can then set the data type of each column, which enables us to use other features such as auto sorting and also the use of renderers. An example of how we create a table model is shown below:

Code Block 6.2.2 *Defining a DefaultTableModel*

```
DefaultTableModel tableModel = new DefaultTableModel(tableData, tableColumns) {
    Class<?>[] types = classTypes;
    @Override
    public Class<?> getColumnClass(int columnIndex) {
        return this.types[columnIndex];
    }
    public boolean isCellEditable(int row, int column) {
    return false;
    }
};
```

The parameters shown in code block 6.2.2; tableData and tableColumns, are of types object array array (Object[][]) and string array (String[]) respectively.

The 'tableColumns' variable in code block 6.2.2 contains the names of each column header in an array. We take these in as String type as they will never be anything but a label; they are not included in the filtering or sorting of rows and are completely static.

Whereas, the 'tableData' variable contains the rows and columns of data that we want to be shown in the table. This data is taken from the respective '.json' files, depending on the table we are filling; this data is taken using the following codeblock:

Code Block 6.2.3 *Creating a DataTable object*

```
JSONParser parser = new JSONParser();
try {
    JSONArray jsonArray = (JSONArray) parser.parse(new FileReader(fileDir));
    ArrayList<LinkedHashMap<String, String» tableValues = new ArrayList<>();
    for (Object arrObject : jsonArray) {
        JSONObject object = (JSONObject) arrObject;
        LinkedHashMap<String, String> row = new LinkedHashMap<>();
        for (String columnHeader : columnHeaders) {
            row.put(columnHeader, String.valueOf(object.get(columnHeader)));
        }
        tableValues.add(row);
    }
    return tableValues;
} catch (Exception ignored) { }
return null;
```

Code block 6.2.3 is the code inside the 'DataTable' class in our Java program. It's purpose is to extract the data from a '.json' file and store it in a suitable way for the DefaultTableModel to read. We do this by first creating a 'JSONParser' object, which is used to read the JSON structure from the file sent as a parameter. We read this data into a 'JSONArray' object and split this object into JSONObject objects (containing the data from each row). Then we take each value and pair it with it's corresponding key (column name), putting these rows into a 'LinkedHashMap' object and finally returning the object we created. This is the 'tableData' variable parsed in code block 6.2.2.

6.3 Theme

The theme of the GUI will follow a similar theme to the Premier League in terms of the colour scheme. However, there are two themes which can be used throughout the program; light and dark modes.

The light theme is the default theme on start-up and consists of a primarily light colour scheme with white/grey component backgrounds, but with black text. The logo in this theme is purple (matching the same purple colour used in the premier league website).

In order to have a clean product output, we use use the 'FlatLaf' library, which is a library which changes the look and feel of a Java Swing interface. The purpose of this look is to give the program a flat look with no shadows or gradients. In our program we use two different FlatLaf themes in 'FlatLightLaf' and 'FlatDarkLaf', which are FlatLaf light and dark themes respectively. When the program is in dark mode (FlatDarkLaf), our component backgrounds are black/dark grey and our test is white; furthermore, our logo is then set to a green colour (the same green used in the Premier League's colours).

The program's uses the same font throughout each page and component, this being a font called 'Quicksand', however different sizes are used depending on the component. The title is set as font size 36, the JTable headers are size 15 and table body is set at 13.

6.4 Main Frame

The main frame in our product is the one section of the product which will always be shown and is where the user can access each tab. This is the most important part of the interface in terms of visualisation and attraction for this reason. Below is an example of how our main frame will look when it is first opened (for simplicity, we will stick to looking at only the main frame, ignoring the content in the tabs):

Image 6.4.1 *Main frame*

The first functional component in this frame is the light/dark mode button which changes between our themes, described in the Theme section; this component is shown below:

Image 6.4.2 *Dark mode button*

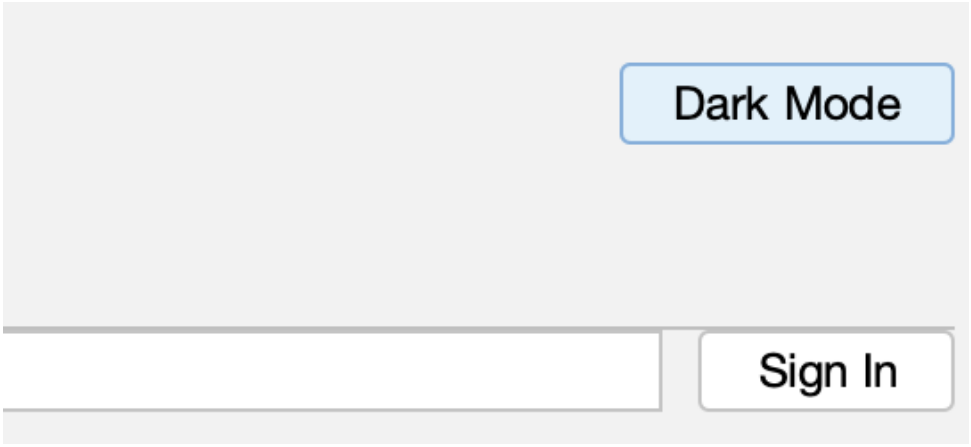
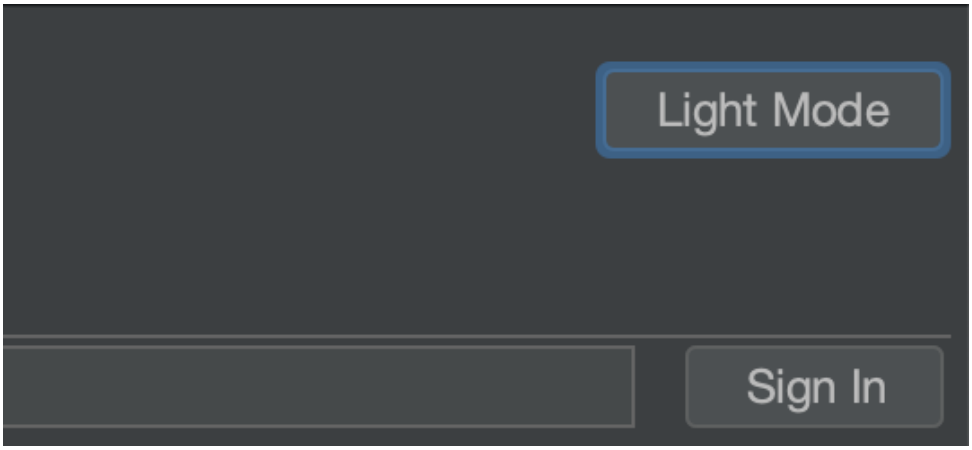


Image 6.4.3 *Light mode button*



As you can see in images 6.3.1 and 6.3.2, the different themes have a very obvious difference and can very easily be switched between. The following shows exactly how we made this possible within a single button:

Code Block 6.4.4 *Light/dark mode JButton*

```
colourModeButton.addActionListener(e -> {
    if (Objects.equals(colourModeButton.getText(), "Dark Mode")) {
        try {
            UIManager.setLookAndFeel(new FlatDarculaLaf());
            changeLaF("Light Mode", plLogoGreen);
        } catch (UnsupportedLookAndFeelException ignored) {}
    }
    else if (Objects.equals(colourModeButton.getText(), "Light Mode")) {
        try {
```

```
                UIManager.setLookAndFeel(new FlatLightLaf());
                changeLaF("Dark Mode", plLogoPurple);
            } catch (UnsupportedLookAndFeelException ignored) {}
        }
    });
```

Firstly we must check which theme we wish to chenge to (if in dark mode, we change to light mode and vice-versa) by getting the text on the button. We then compare set the look and feel in the UIManager using '.setLookAndFeel' and run our changeLaF method with the parameters based on whichever theme we wish to switch to. This is added as a listener to a button so that whenever the button is pressed by the user, the action commences.

Below we see our 'changeLaF' method which changes the other components in our program, based on whether we are in the light or dark theme:

Code Block 6.4.5 *changeLaF method*

```
public void changeLaF(String text, Icon logo) {
    colourModeButton.setText(text);
    FlatLaf.updateUI();
    logoLabel.setIcon(logo);
}
```

The changeLaF method parses two parameters; a string variable containing the new button text and a string variable containing a link to the new logo image. This method sets the text of the Light/Dark JButton, sets the icon of the logo label to the image in the path provided and updates the interface's look and feel to match the chosen mode.

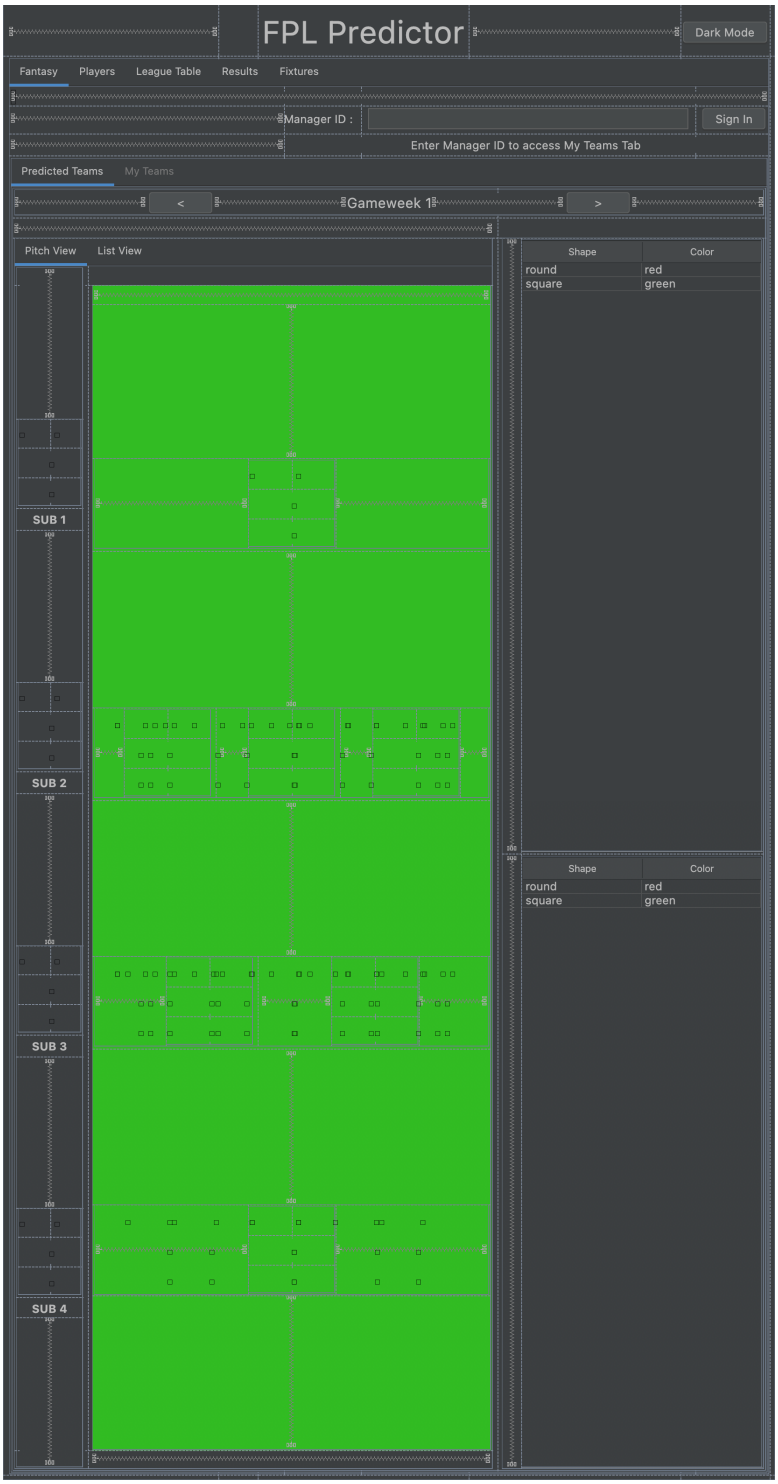
The other functional component we see in the main frame is a JTabbedPane. This is a component which allows us to have multiple pages which can be accessed by selecting each tab. These are easily programmed using the Swing Designer Form.

6.5 Fantasy Tab

The 'Fantasy' tab in our program contains two sub-tabs; 'Predicted Teams' and 'My Teams' tabs. Each of these sub-tabs also contain two sub-sub-tabs; 'Pitch View' and 'List View'. The design of the sub-tabs 'Predicted Teams' and 'My Teams' are exactly the same design,

however one shows our team predictions and the other shows a manager's past teams (if a Manager ID has been inputted). Below we see our 'Fantasy' tab open in the Swing Designer Form:

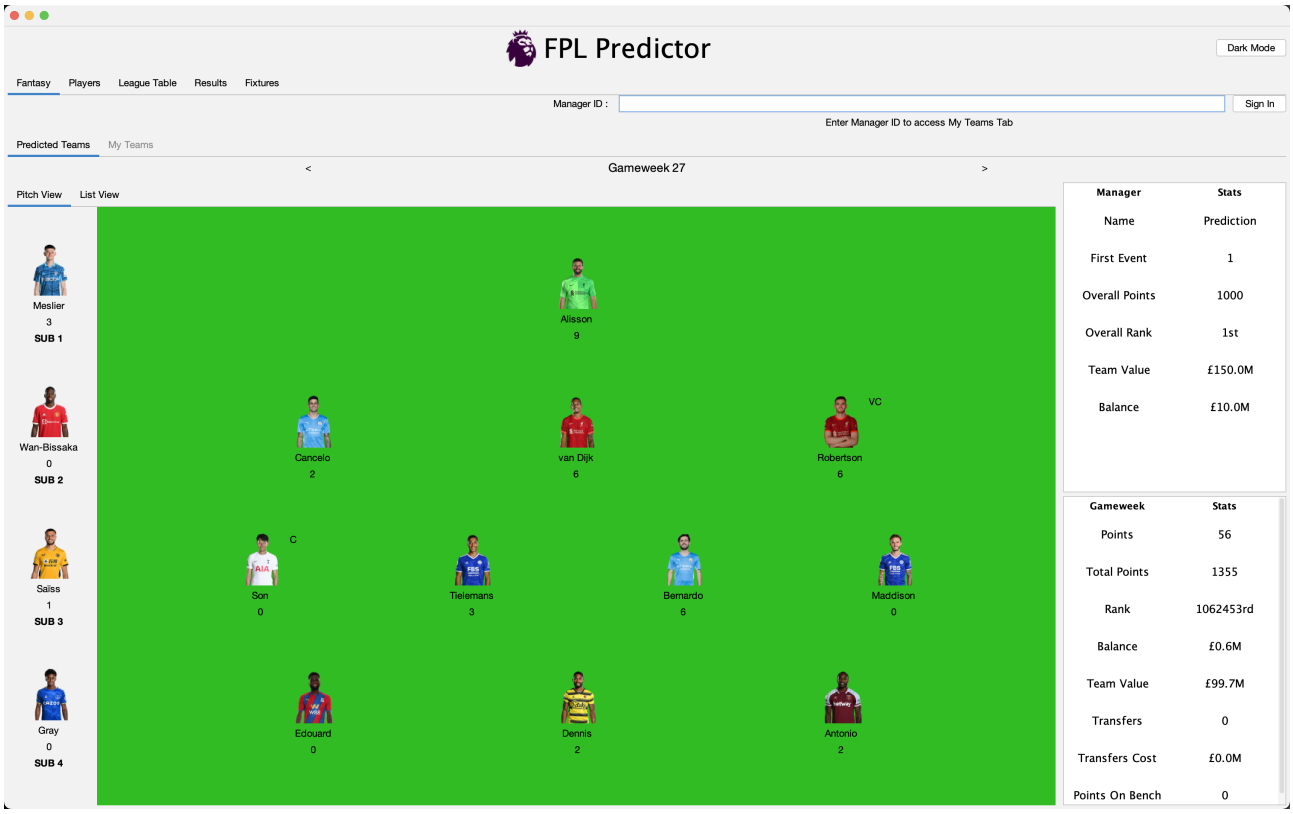
Image 6.5.1 *Fantasy tab in the Swing Designer Form*



As we can see in the above image, most of the designing for this part of the program

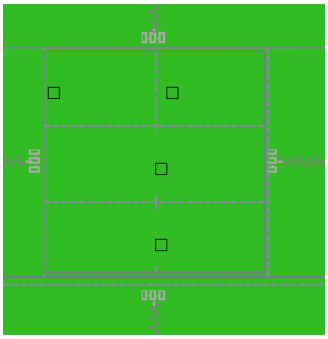
is hard-coded through the Swing Designer Form using various components. We make use of the spacers in order to evenly spread our components and have a clear idea of how the program will look visually. Most of the labels and other forms of text have been created through the form; however, many of the components with functionality have to be created or modified within a Java script itself. An example output of this tab is shown below:

Image 6.5.2 *Fantasy tab*



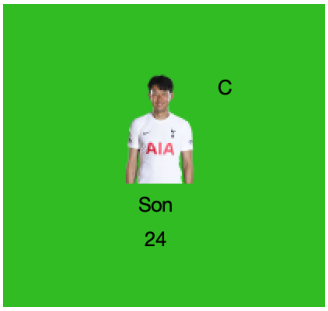
The most important part of this tab, like any other, is the displaying of our information. For each player we have a square-shaped JFrame containing four blank labels; these labels are where we show our data. This is shown below:

Image 6.5.3 *Player in the Swing Designer Form*



Referring to the player design above, the placement of each piece of data follows suit with our wire-frame design; we have the player's image in the top left, captaincy icon in the top right, name in the middle and point count at the bottom as shownin our output below:

Image 6.5.4 *Player in program output*



Furthermore, we have also used this player JFrame in a more complex way in order to shown the correct formation of the team. As stated in the rules, the user may use a number of formations. For this reason, we must make use of Swing's 'CardLayout' type in the 'Layout Manager'. This allows us to design multiple 'cards' (layouts) and choose which card to show at any time. For example, when a formation with three defenders is being used, we can select our card which has 3 players; if we have four defenders in the formation, we select our four-player card, etc. We split our design up into positions in order to use the least amount of cards necessary, which leaves us with the following:

- Goalkeeper
  - 1 player card
- Defenders
  - 3 player card
  - 4 player card
  - 5 player card
- Midfielders
  - 2 player card
  - 3 player card

  - 4 player card
  - 5 player card

- Forwards
  - 1 player card
  - 2 player card
  - 3 player card

We do not need to make any layout changes to our goalkeeper as there can only ever be one goalkeeper at a time; therefore, we can leave our goalkeeper JPanel in the default layout. For the other positions, it is necessary to use the 'CardLayout' feature, so we create and assign the necessary cards shown above; this is done using the following code:

Code Block 6.5.5 *Create cards*

```
LinkedHashMap<JPanel, String[]> myTeamCards = createCards(new String[] {
    "myDEF3Card", "myDEF4Card", "myDEF5Card",
    "myMID2Card", "myMID3Card", "myMID4Card", "myMID5Card",
    "myFWD1Card", "myFWD2Card", "myFWD3Card"
}, new JPanel[] {myDEFPanel, myMIDPanel, myFWDPanel});

public LinkedHashMap<JPanel, String[]> createCards(String[] cardNames, JPanel[] panelNames)
{
    LinkedHashMap<JPanel, String[]> cards = new LinkedHashMap<>();
    int cardCounter = 0;
    int panelCounter = 0;
    for (int amount : new int[] 3, 4, 3) {
        String[] cardArray = new String[amount];
        for (int i = 0; i < amount; i++) {
            cardArray[i] = cardNames[cardCounter];
            cardCounter +=1;
        }
        cards.put(panelNames[panelCounter], cardArray);
        panelCounter += 1;
```

```
        }
        return cards;
    }
```

Code block 8.5.4 contains two methods, which together create and assign our JPanels to our cards' LinkedHashMap. The first class is just one of the instances where we create and assign our cards as we do this multiple times. The best way to store these cards is in a LinkedHashMap as we found that when working out the correct formation, it works well with our algorithm. In our second method, we use a nested for loop and two counters to assign panels to positions in our 'cards' variable.

Now that we have created all of the necessary cards per position, the card selection process begins, based on the formation of the team we are showing. We use a simple algorithm to work out which formation we are using, as shown below:

Code Block 6.5.6 *Calculate formation*

```
int[] formation = {1, 0, 0, 0};
for (HashMap<String, String> player : tableList) {
    if (Integer.parseInt(player.get("position")) <= 11) {
        String position = player.get("position_name");
        switch (position) {
            case "DEF":
                formation[1] += 1;
                break;
            case "MID":
                formation[2] += 1;
                break;
            case "FWD":
                formation[3] += 1;
                break;
        }
    }
}
```

We store our formation in an array with the goalkeeper already entered in the first position being '1', We then go through the first 11 players in our team and get their positions. We then use a switch statement to compare each player's position to our options and add to our array for each position. This leaves us with an array containing the amount of players in each position, stored in the variable 'formation'.

Following this, we then call our 'setCard()' method and send all of the card panels and our formation array as parameters; this method is shown below:

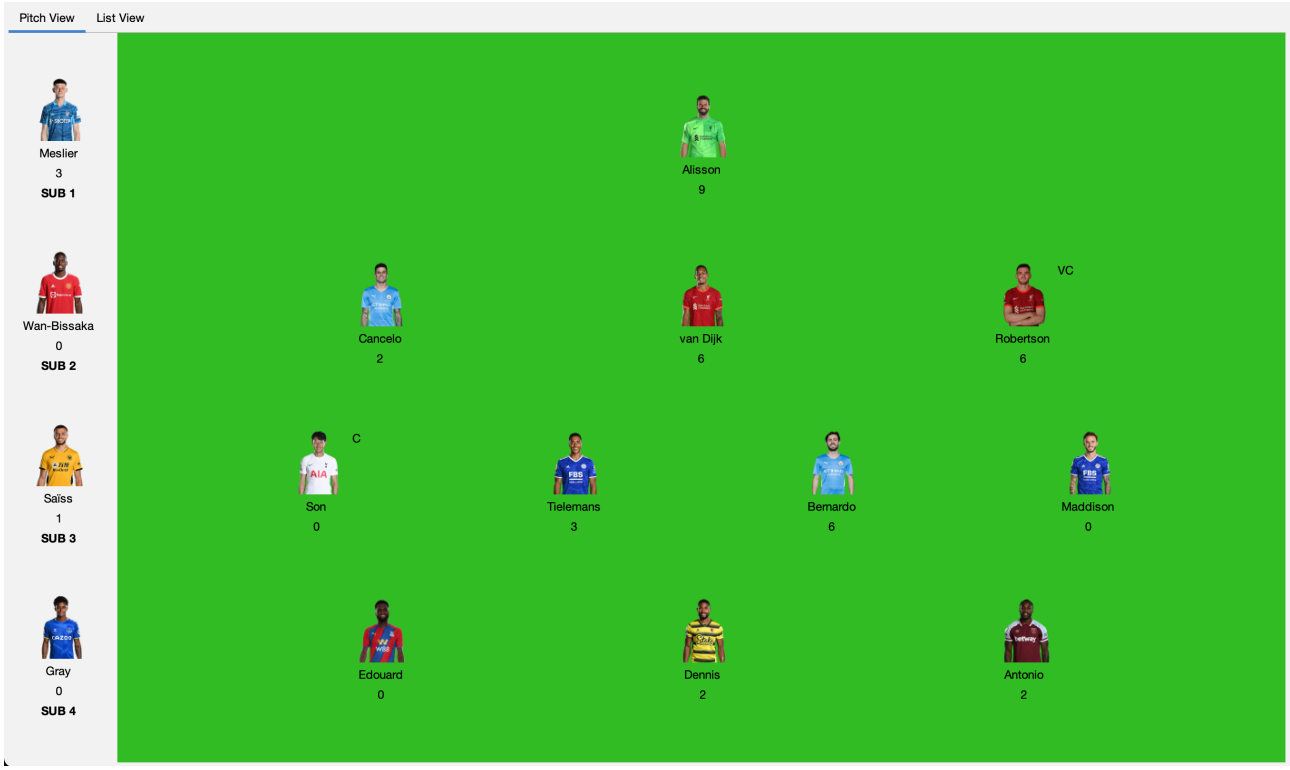
Code Block 6.5.7 *setCard() method*

```
int i = 1;
int[] amount = new int[] {3, 2, 1};
for (JPanel panel : panels.keySet()) {
    CardLayout card = (CardLayout) panel.getLayout();
    card.show(panel, panels.get(panel)[formation[i]-amount[i-1]]);
    i += 1;
}
```

The setCard() method simply uses an array containing the differences between the two parameters, then runs through the formation array and choses the correct card to be displayed per position.

Finally, we pull the player data from our arrayList and put the correct data in the correct labels, including the substitutes who have their own display section on the left-hand side. The result of this is shown below:

Image 6.5.8 *Team in pitch view*



The 'List View' tab contains a JTable component which shows the players in the displayed team in a list, in slightly more detail. We create a table in the Swing Form, but add the data using the methods shown in code blocks 8.2.2 and 6.2.3.

This is an example of how our tab looks once we have inserted the data:

Image 6.5.9 *Team in pitch view*

Pitch View | List View

		Name	Team	Position	Points	Captain
1		Alisson	LIV	GKP	9	
2		Cancelo	MCI	DEF	2	
3		van Dijk	LIV	DEF	6	
4		Robertson	LIV	DEF	6	VC
5		Son	TOT	MID	0	C
6		Tielemans	LEI	MID	3	
7		Bernardo	MCI	MID	6	
8		Maddison	LEI	MID	0	
9		Edouard	CRY	FWD	0	
10		Dennis	WAT	FWD	2	
11		Antonio	WHU	FWD	2	
12		Meslier	LEE	GKP	3	
13		Wan-Bissaka	MUN	DEF	0	
14		Saïss	WOL	DEF	1	
15		Gray	EVE	MID	0	

Not only is this table visually appleasing, it is also interactive. As we have set the data types of each of the columns in this table, we can sort each column by pressing the header. The sort procedure used depands on the columns data type; for example, the first unnamed column and our points columns are sorted based on the total points and can be switch from ascending to descending with a single click. The other columns are sorted alphabetically, and can also wbe switched from ascending to descending or vice-versa at any time.

An example of our table being sorted by the 'Points' column in descending order is shown below:

Image 6.5.10 *Team sorted by points*

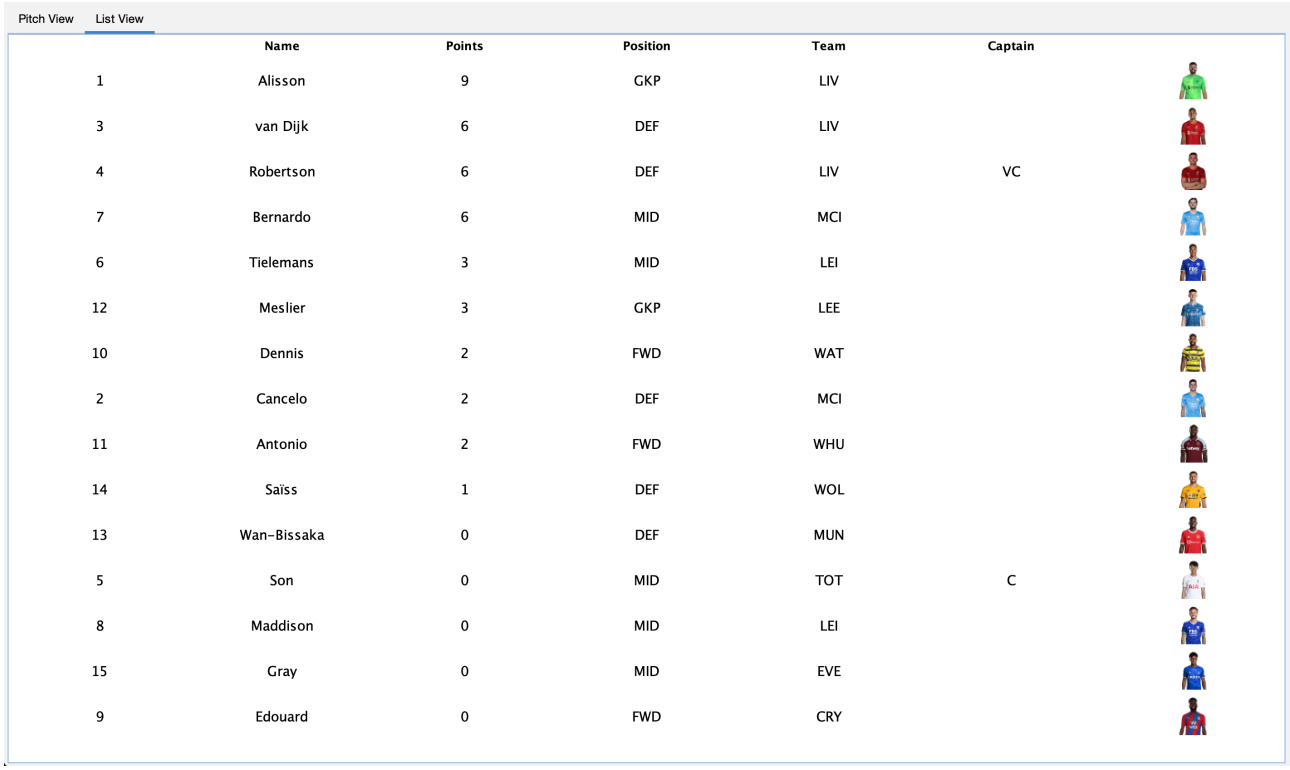
Pitch View | List View

		Name	Team	Position	Points	Captain
1		Alisson	LIV	GKP	9	
7		Bernardo	MCI	MID	6	
4		Robertson	LIV	DEF	6	VC
3		van Dijk	LIV	DEF	6	
12		Meslier	LEE	GKP	3	
6		Tielemans	LEI	MID	3	
11		Antonio	WHU	FWD	2	
2		Cancelo	MCI	DEF	2	
10		Dennis	WAT	FWD	2	
14		Saïss	WOL	DEF	1	
9		Edouard	CRY	FWD	0	
15		Gray	EVE	MID	0	
8		Maddison	LEI	MID	0	
5		Son	TOT	MID	0	C
13		Wan-Bissaka	MUN	DEF	0	

Another feature of our tables is that we can change the column order. This is useful when we are focusing on just a few of the shown columns. If the user drags the column header, the column will move and snap into place upon release.

If a user wanted to make a comparison between between the points per player, the two most important columns are the player's 'Name' and 'Points'. However, these columns are not close together, which is an inconvenience. An example of this feature being used to solve this inconvenience is shown below:

Image 6.5.11 *Moving columns in our table*


Pitch View List View

	Name	Points	Position	Team	Captain	
1	Alisson	9	GKP	LIV		
3	van Dijk	6	DEF	LIV		
4	Robertson	6	DEF	LIV	VC	
7	Bernardo	6	MID	MCI		
6	Tielemans	3	MID	LEI		
12	Meslier	3	GKP	LEE		
10	Dennis	2	FWD	WAT		
2	Cancelo	2	DEF	MCI		
11	Antonio	2	FWD	WHU		
14	Saïss	1	DEF	WOL		
13	Wan-Bissaka	0	DEF	MUN		
5	Son	0	MID	TOT	C	
8	Maddison	0	MID	LEI		
15	Gray	0	MID	EVE		
9	Edouard	0	FWD	CRY		

There are two other types of components in this tab which are also tables; they show the manager's stats and also the gameweek stats. These are not a part of the pitch or list view and are shown no matter which view we are showing as they are a part of the main sub-tabs. The manager stats table shows data about the manager or prediction overall and has the same data shown, no matter which gameweek is being viewed. However, the Gameweek Stats table updates it's data depending on which gameweek we are viewing as the data is specifically about that gameweek. An example of how the tables look is shown below:

Image 6.5.12 *Stats tables*

Manager	Stats
Name	Prediction
First Event	1
Overall Points	1000
Overall Rank	1st
Team Value	£150.0M
Balance	£10.0M

Gameweek	Stats
Points	36
Total Points	1391
Rank	5520017th
Balance	£0.6M
Team Value	£99.3M
Transfers	0
Transfers Cost	£0.0M
Points On Bench	4

Our stats tables both have scroll bars for when all of the data is not visible. Sorting is not possible on these table's columns, as the data is of dfferent types, making it completely unneccessary.

In order to change which gameweek is being viewed, we have two JButton components, which work as simple left/right buttons. Pressing the left button changes to the week before; pressing the right button changes to the following week. Obviously the buttons are disabled if there is no week before or after the current gameweek being shown. Below, our gameweek label and buttons are shown:

Image 6.5.13 *Gameweek label and buttons*

For us to know how many gameweeks we have data for, we take the data from our 'list-of-gameweeks.json' file and store it in an array. We then use the buttons to activate a

counter variable which moves between positions in the array. Whenever a button is refreshed, we refresh the necessary components, for example, the gameweek label is changed, we get new table data for the corresponding week and we update the pitch and list view graphics.

The final components we have in our fantasy tab are the manager ID text-box and sign in button, which work in-conjuction with each other, in order for the user to see their previous fantasy football teams. These components are shown below:

Image 6.5.14 *Sign-in*

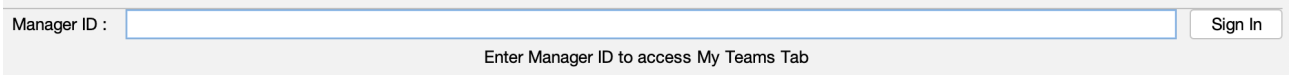
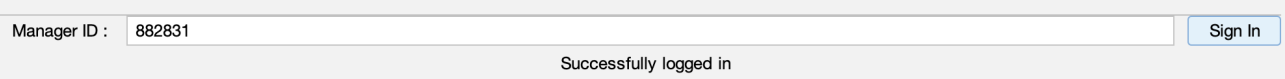


Image 6.5.15 *Signed-in*



One of the API requests we were using required credentials to get the current team the user has, but taking the user's username and password for their account was decided against due to security reasons. Moreover there is no real benefit of seeing the current chosen team of a player, as it is not possible to officially make changes to a team using our application.

The manager ID text-box has the purpose of taking in the user's inputted text, completing validation check to make sure it is a valid ID and calling the Python script to collect the data. Below is the code which completes the validation check of the ID:

Code Block 6.5.16 *Check ID validity*

```
try {
    Integer.parseInt(managerId);
} catch (NumberFormatException notInteger) {
    return false;
}
return getAPIResponse("https://fantasy.premierleague.com/api/entry/" + managerId + "/");
```

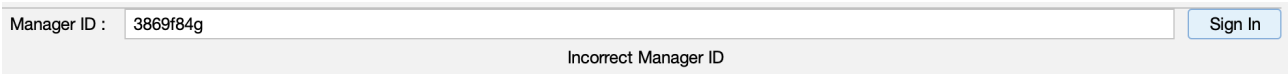
The first validation check that is made is to make sure that the inputted data is an integer as a Manager ID cannoy contain any non-integer characters. If the text is vaid, we can check there is a response from the API which can be done as follows:

Code Block 6.5.17 *getAPIResponse() method [7]*

```
try {
    URL url = new URL(urlStr);
    HttpURLConnection connection = (HttpURLConnection) url.openConnection();
    connection.setRequestMethod("GET");
    connection.connect();
    connection.getPermission();
    return connection.getResponseCode() != 404;
} catch (Exception ignored) {
    return false;
}
```

Using the getAPIResponse() method we either can't connect to the API if the manager ID is not valid, or if there is an issue retreiving the data, we can receieve a respose code of '404'. Both of these outcomes will result in us not having data to show, so we ensure that we recieve a valid response code before we attempt to pull the data from the API through Python. If we do not manager to get any data for the inputted manager ID for any reason, the user sees the following:

Image 6.5.18 *Incorrect manager ID*



There is a brief time-delay when using the sign-in function which is not ideal, which hopefully can be shortened in the future. However, to not confuse the user, we make the tab unclickable, until the sign-in process is finished and there is data to be viewed. The following shows what this looks like:

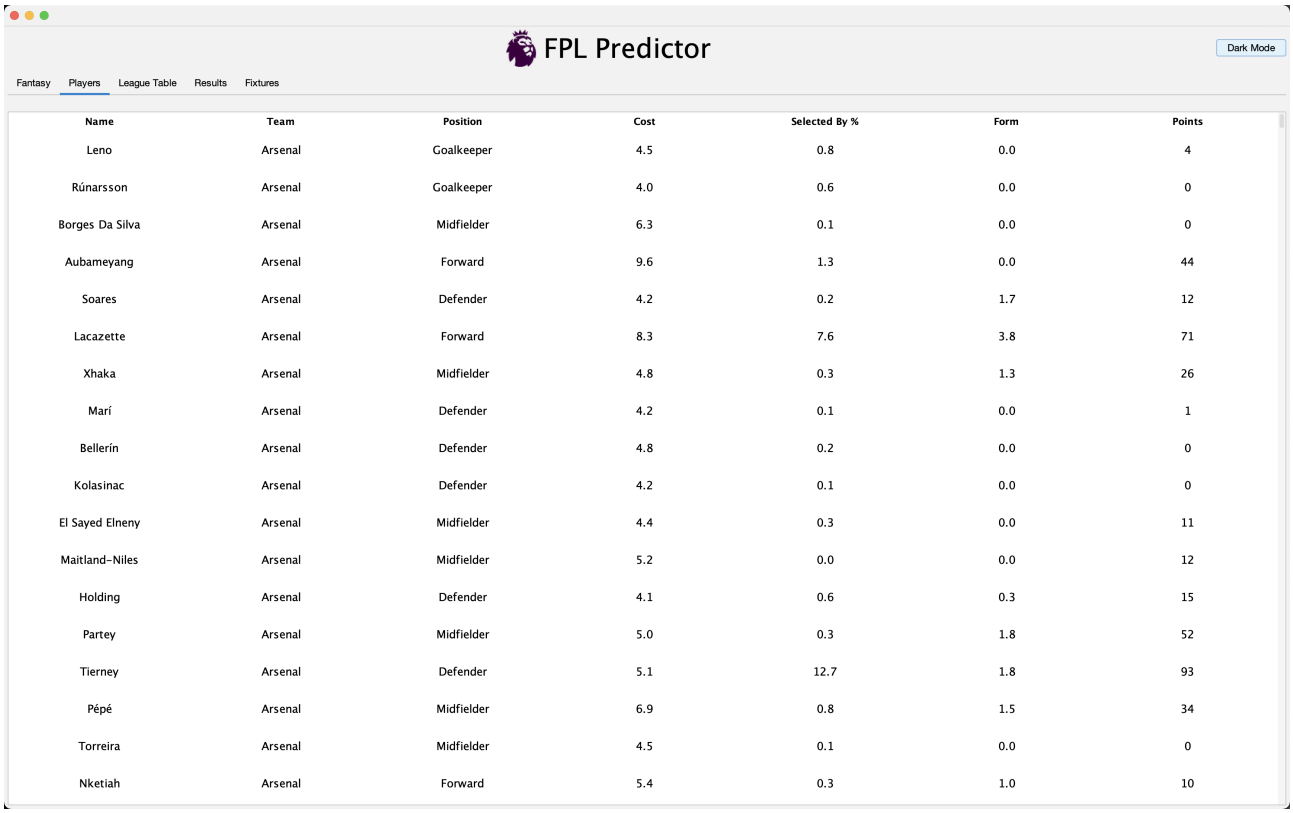
Image 6.5.19 *Unclickable tab*

Moreover, similarly to the Predicted Teams tab, we have both a pitch and list view tab which contains the exact same functioning components.

6.6 Players Tab

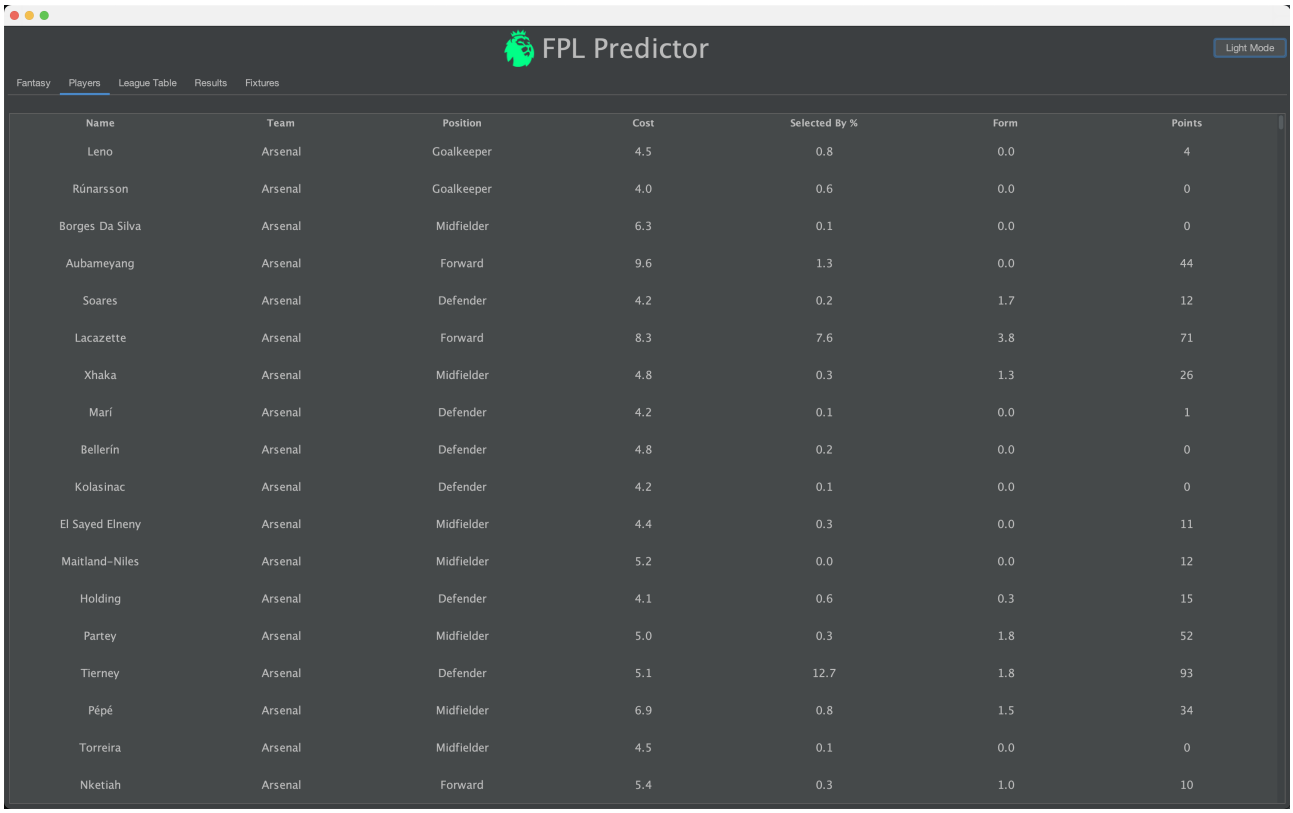
The 'Players' tab contains a database-like table, useful for searching fantasy stats about players. We create this table using a JTable components and enter our data using the methods shown in code blocks 6.2.2 and 8.2.3. Our table headers are of font size 15 and body size 13 and all of the cells are centered horizontally using our custom table renderers. The order of the columns in this table can be modified and the rows can be filtered by a particular column. Below is how our players tab looks in both our light and dark modes:

Image 6.6.1 *Players tab in light mode*



Name	Team	Position	Cost	Selected By %	Form	Points
Leno	Arsenal	Goalkeeper	4.5	0.8	0.0	4
Rúnarsson	Arsenal	Goalkeeper	4.0	0.6	0.0	0
Borges Da Silva	Arsenal	Midfielder	6.3	0.1	0.0	0
Aubameyang	Arsenal	Forward	9.6	1.3	0.0	44
Soares	Arsenal	Defender	4.2	0.2	1.7	12
Lacazette	Arsenal	Forward	8.3	7.6	3.8	71
Xhaka	Arsenal	Midfielder	4.8	0.3	1.3	26
Marí	Arsenal	Defender	4.2	0.1	0.0	1
Bellerín	Arsenal	Defender	4.8	0.2	0.0	0
Kolasinac	Arsenal	Defender	4.2	0.1	0.0	0
El Sayed Elneny	Arsenal	Midfielder	4.4	0.3	0.0	11
Maitland-Niles	Arsenal	Midfielder	5.2	0.0	0.0	12
Holding	Arsenal	Defender	4.1	0.6	0.3	15
Partey	Arsenal	Midfielder	5.0	0.3	1.8	52
Tierney	Arsenal	Defender	5.1	12.7	1.8	93
Pépé	Arsenal	Midfielder	6.9	0.8	1.5	34
Torreira	Arsenal	Midfielder	4.5	0.1	0.0	0
Nketiah	Arsenal	Forward	5.4	0.3	1.0	10

Image 6.6.2 *Players tab in dark mode*

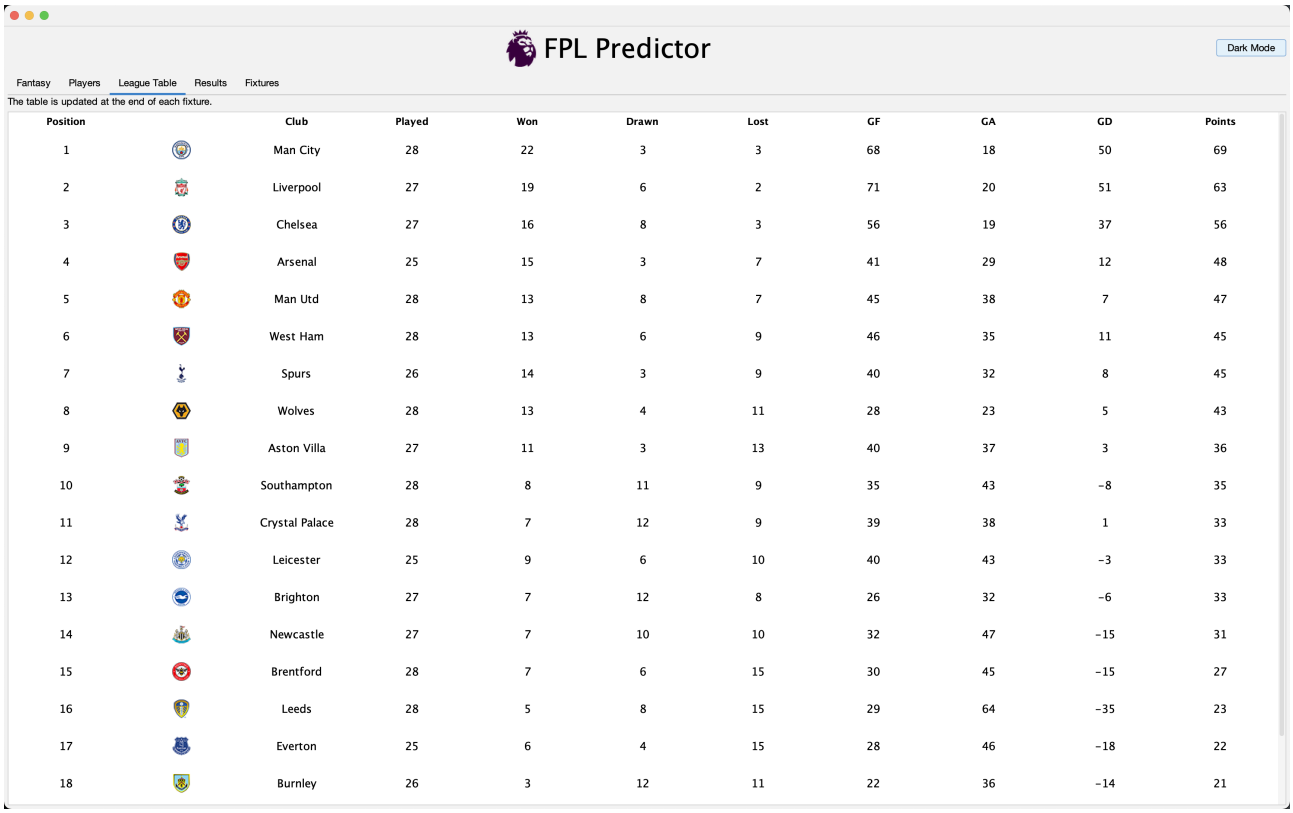


Name	Team	Position	Cost	Selected By %	Form	Points
Leno	Arsenal	Goalkeeper	4.5	0.8	0.0	4
Rúnarsson	Arsenal	Goalkeeper	4.0	0.6	0.0	0
Borges Da Silva	Arsenal	Midfielder	6.3	0.1	0.0	0
Aubameyang	Arsenal	Forward	9.6	1.3	0.0	44
Soares	Arsenal	Defender	4.2	0.2	1.7	12
Lacazette	Arsenal	Forward	8.3	7.6	3.8	71
Xhaka	Arsenal	Midfielder	4.8	0.3	1.3	26
Marí	Arsenal	Defender	4.2	0.1	0.0	1
Bellerín	Arsenal	Defender	4.8	0.2	0.0	0
Kolasinac	Arsenal	Defender	4.2	0.1	0.0	0
El Sayed Elneny	Arsenal	Midfielder	4.4	0.3	0.0	11
Maitland-Niles	Arsenal	Midfielder	5.2	0.0	0.0	12
Holding	Arsenal	Defender	4.1	0.6	0.3	15
Partey	Arsenal	Midfielder	5.0	0.3	1.8	52
Tierney	Arsenal	Defender	5.1	12.7	1.8	93
Pépé	Arsenal	Midfielder	6.9	0.8	1.5	34
Torreira	Arsenal	Midfielder	4.5	0.1	0.0	0
Nketiah	Arsenal	Forward	5.4	0.3	1.0	10

6.7 League Table Tab

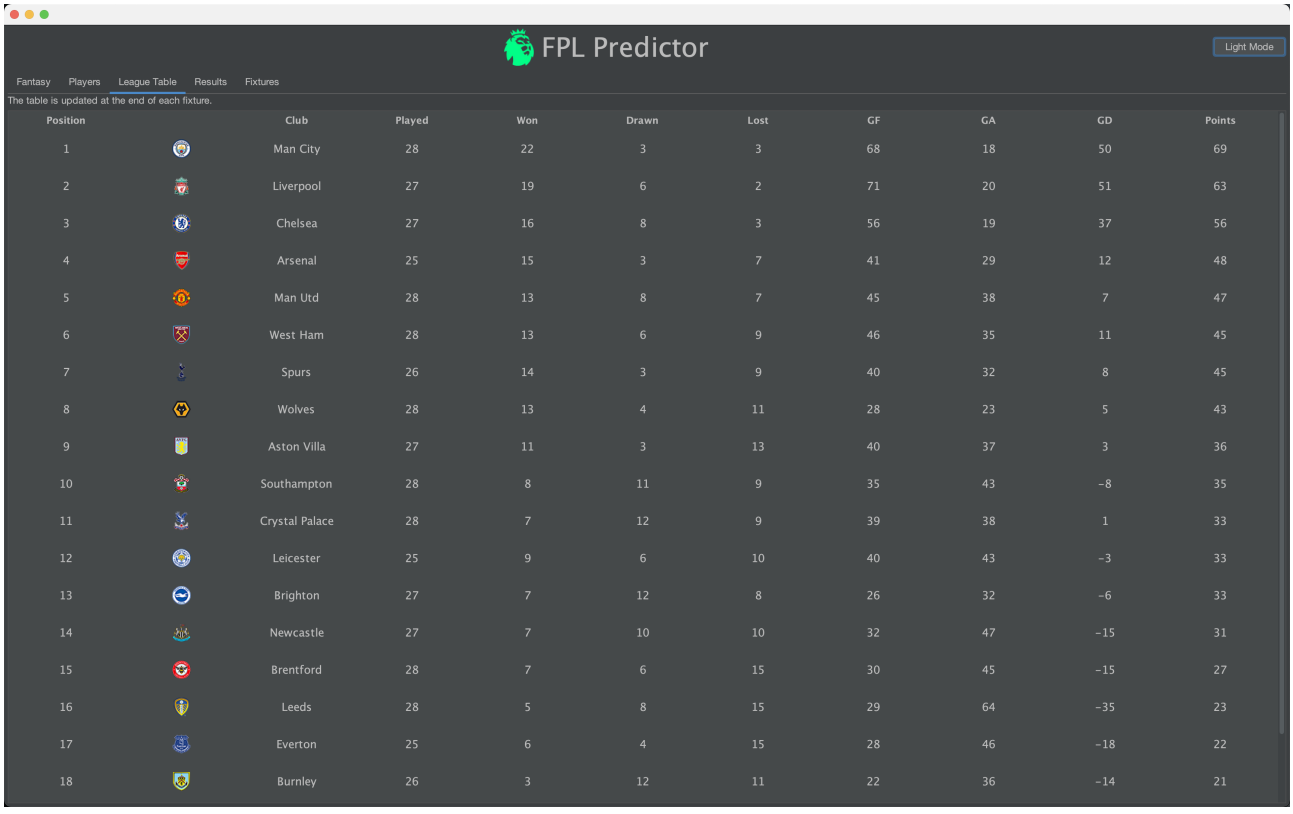
Our 'League table' tab shows a live league table for the Premier League, as you would expect. We use a JTable component to do create the table and enter our data using the methods shown in code blocks 6.2.2 and 6.2.3. Our table headers and body are of sizes 15 and 13 respectively and all of the cells are centered horizontally using our custom table renderers. The order of the columns in this table can be modified to allow team comparisons to be made easier, however, no filters can be applied to the data as the purpose of this table is to provide the user with an official order, based on the official tabling rules of the Premier League. The table is inside a JScrollPanel, which allows us to scroll throught the table if not all of the data is visible. Below is how our live league table looks in both our light and dark modes:

Image 6.7.1 *League table tab in light mode*



Position	Club	Played	Won	Drawn	Lost	GF	GA	GD	Points
1	Man City	28	22	3	3	68	18	50	69
2	Liverpool	27	19	6	2	71	20	51	63
3	Chelsea	27	16	8	3	56	19	37	56
4	Arsenal	25	15	3	7	41	29	12	48
5	Man Utd	28	13	8	7	45	38	7	47
6	West Ham	28	13	6	9	46	35	11	45
7	Spurs	26	14	3	9	40	32	8	45
8	Wolves	28	13	4	11	28	23	5	43
9	Aston Villa	27	11	3	13	40	37	3	36
10	Southampton	28	8	11	9	35	43	-8	35
11	Crystal Palace	28	7	12	9	39	38	1	33
12	Leicester	25	9	6	10	40	43	-3	33
13	Brighton	27	7	12	8	26	32	-6	33
14	Newcastle	27	7	10	10	32	47	-15	31
15	Brentford	28	7	6	15	30	45	-15	27
16	Leeds	28	5	8	15	29	64	-35	23
17	Everton	25	6	4	15	28	46	-18	22
18	Burnley	26	3	12	11	22	36	-14	21

Image 6.7.2 *League table tab in dark mode*



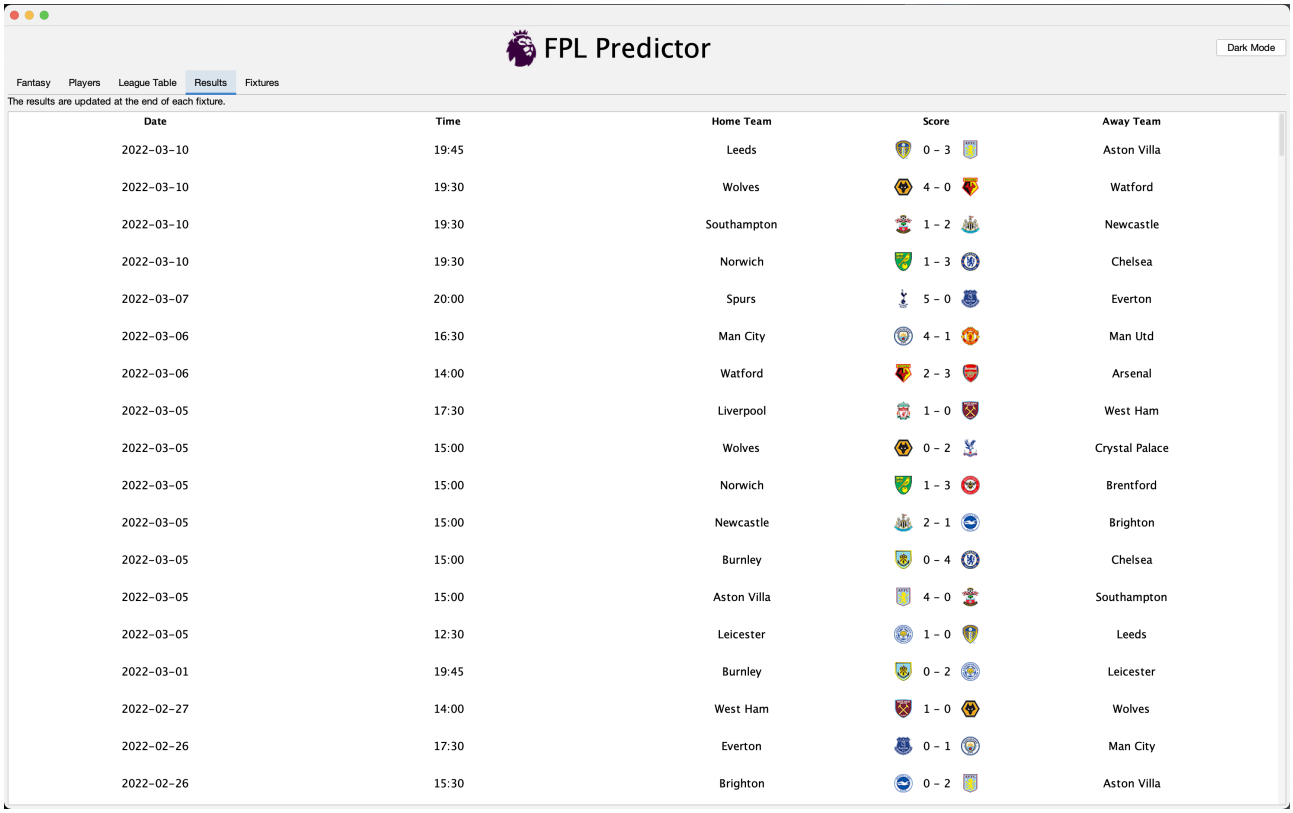
Position	Club	Played	Won	Drawn	Lost	GF	GA	GD	Points
1	Man City	28	22	3	3	68	18	50	69
2	Liverpool	27	19	6	2	71	20	51	63
3	Chelsea	27	16	8	3	56	19	37	56
4	Arsenal	25	15	3	7	41	29	12	48
5	Man Utd	28	13	8	7	45	38	7	47
6	West Ham	28	13	6	9	46	35	11	45
7	Spurs	26	14	3	9	40	32	8	45
8	Wolves	28	13	4	11	28	23	5	43
9	Aston Villa	27	11	3	13	40	37	3	36
10	Southampton	28	8	11	9	35	43	-8	35
11	Crystal Palace	28	7	12	9	39	38	1	33
12	Leicester	25	9	6	10	40	43	-3	33
13	Brighton	27	7	12	8	26	32	-6	33
14	Newcastle	27	7	10	10	32	47	-15	31
15	Brentford	28	7	6	15	30	45	-15	27
16	Leeds	28	5	8	15	29	64	-35	23
17	Everton	25	6	4	15	28	46	-18	22
18	Burnley	26	3	12	11	22	36	-14	21

6.8 Results Tab

The Results tab shows the latest results of the Premier League in a JTable component. The default order of the table is to show from the most recent to oldest results; this order cannot be changed as no filters can be applied to the columns. In the future, it would be ideal to have a drop-down to allow the user to select a team, only the matches of that team would then be shown.

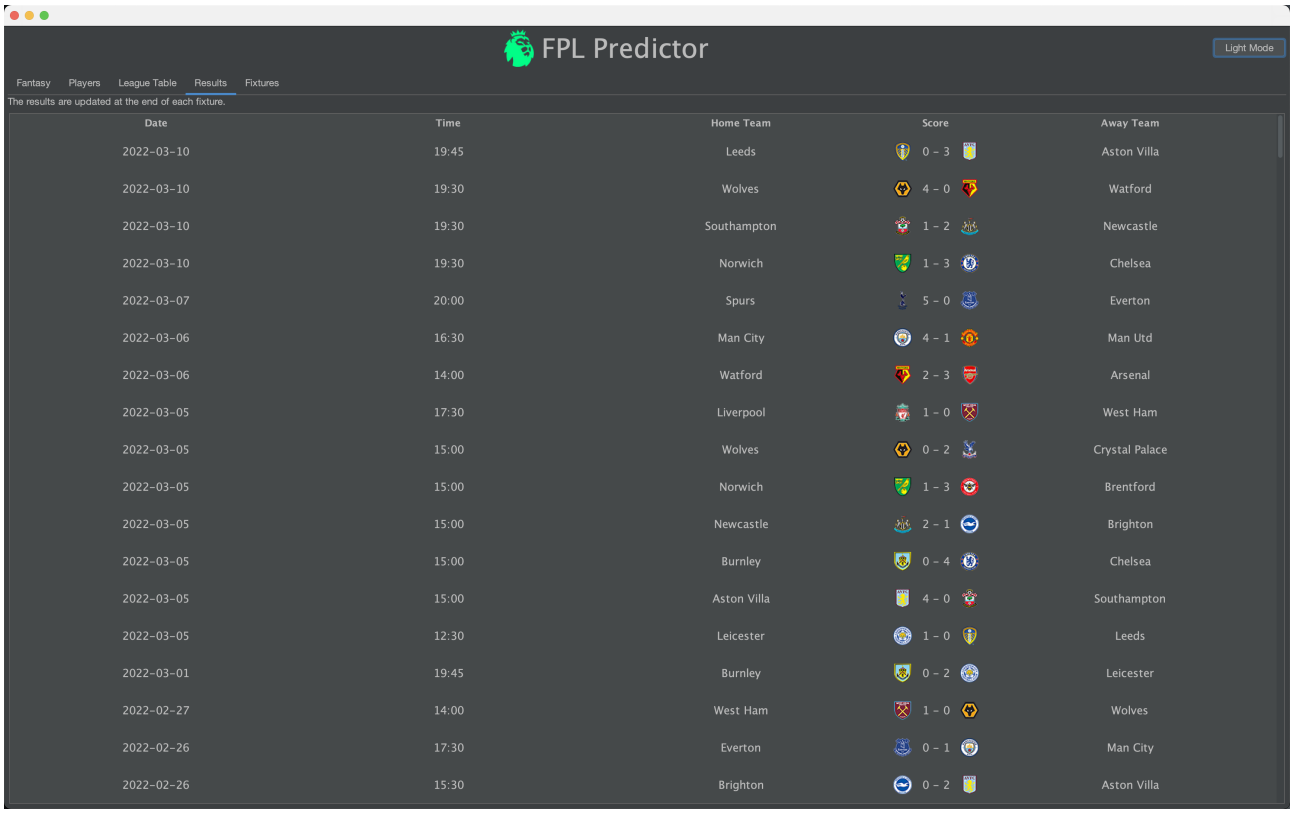
The column order in this table can be modified to allow users to choose how they view the data. However, this does uncover a design flaw as the user can switch 'Home Team' and 'Away Team' columns, as well as the badge columns, in order to switch all of the results. I do not see why a user would do this, unless as a mistake. The only way to fix this issue would be to have the team names, badges and scores in one column, but this would remove almost all of the column order modification options and defeat the point of the whole feature. Furthermore, the font style and sizing follows suit with the other tables in the program.

Image 6.8.1 *Results tab in light mode*



Date	Time	Home Team	Score	Away Team
2022-03-10	19:45	Leeds	0 - 3	Aston Villa
2022-03-10	19:30	Wolves	4 - 0	Watford
2022-03-10	19:30	Southampton	1 - 2	Newcastle
2022-03-10	19:30	Norwich	1 - 3	Chelsea
2022-03-07	20:00	Spurs	5 - 0	Everton
2022-03-06	16:30	Man City	4 - 1	Man Utd
2022-03-06	14:00	Watford	2 - 3	Arsenal
2022-03-05	17:30	Liverpool	1 - 0	West Ham
2022-03-05	15:00	Wolves	0 - 2	Crystal Palace
2022-03-05	15:00	Norwich	1 - 3	Brentford
2022-03-05	15:00	Newcastle	2 - 1	Brighton
2022-03-05	15:00	Burnley	0 - 4	Chelsea
2022-03-05	15:00	Aston Villa	4 - 0	Southampton
2022-03-05	12:30	Leicester	1 - 0	Leeds
2022-03-01	19:45	Burnley	0 - 2	Leicester
2022-02-27	14:00	West Ham	1 - 0	Wolves
2022-02-26	17:30	Everton	0 - 1	Man City
2022-02-26	15:30	Brighton	0 - 2	Aston Villa

Image 6.8.2 *Results tab in dark mode*



Date	Time	Home Team	Score	Away Team
2022-03-10	19:45	Leeds	0 - 3	Aston Villa
2022-03-10	19:30	Wolves	4 - 0	Watford
2022-03-10	19:30	Southampton	1 - 2	Newcastle
2022-03-10	19:30	Norwich	1 - 3	Chelsea
2022-03-07	20:00	Spurs	5 - 0	Everton
2022-03-06	16:30	Man City	4 - 1	Man Utd
2022-03-06	14:00	Watford	2 - 3	Arsenal
2022-03-05	17:30	Liverpool	1 - 0	West Ham
2022-03-05	15:00	Wolves	0 - 2	Crystal Palace
2022-03-05	15:00	Norwich	1 - 3	Brentford
2022-03-05	15:00	Newcastle	2 - 1	Brighton
2022-03-05	15:00	Burnley	0 - 4	Chelsea
2022-03-05	15:00	Aston Villa	4 - 0	Southampton
2022-03-05	12:30	Leicester	1 - 0	Leeds
2022-03-01	19:45	Burnley	0 - 2	Leicester
2022-02-27	14:00	West Ham	1 - 0	Wolves
2022-02-26	17:30	Everton	0 - 1	Man City
2022-02-26	15:30	Brighton	0 - 2	Aston Villa

6.9 Fixtures Tab

This tab is extremely similar to the 'Results' tab in terms of shown data, it's types and the component functionality. Once again, we use a JTable to store our data, which is retreived and displayed using code blocks 6.2.2 and 8.2.3. In terms of table functionality, we can change our column order through simply dragging the column's header or body to the desired location. The same renderers are applied to the table, which horizontally centers our text and sets our font to 'Quicksand' and font sizes to 15 and 13.

Image 6.9.1 *Fixtures tab in light mode*

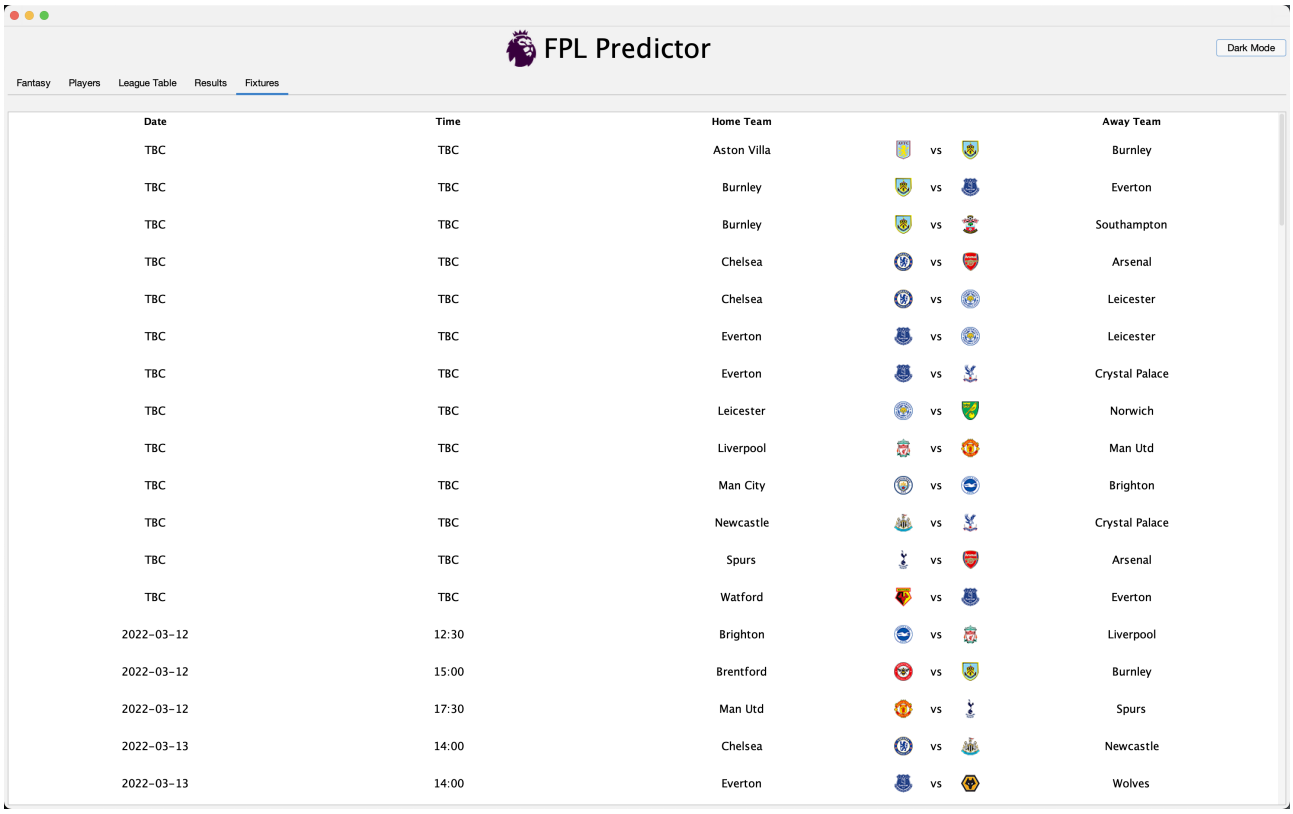
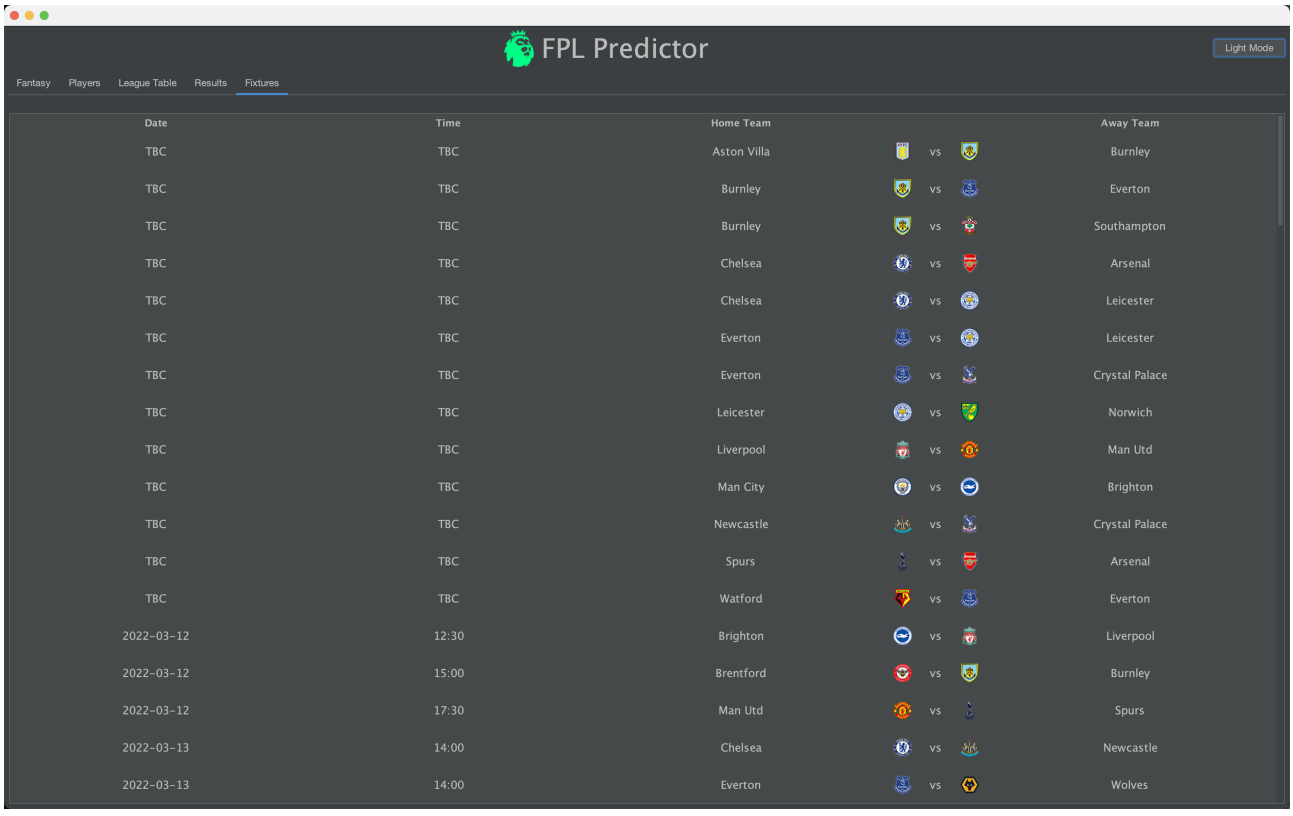


Image 6.9.2 *Fixtures tab in dark mode*



The above images once again shows the difficulty of predicting the Premier League, when

game cancellations, illnesses and other issues arise from non-football related problems such as Covid-19.

CHAPTER 7

Application

Our application's prediction method and data handling is written in Python and the interface is in Java; this allows us to have two different ways to run our program. When running our program in Python, we get a command-line output of our prediction, compared to a fully working application when running our java program.

7.1 Libraries and Imports

Before the application is run, there are many libraries used by both our Python and Java scripts which must be downloaded. Without these, the program will either be very limited in some coases, or more likely unable to be run.

Please ensure that all libraries are installed before attempting to run the program.

Below is the list of libraries and imports we use (of which all are the latest versions as of when this report is dated):

- requests
- os (built-in)
- pandas
- Unidecode
- numpy

- shutil
- sklearn
- PuLP
- JSON (built-in)
- java.io (built-in)
- JSON.simple
- java.util (built-in)
- FlatLaf
- java.net (built-in)
- javax.swing (built-in)
- java.awt (built-in)
- javax.imageio

7.2 Options

Before running the program, we have the option of whether to use the live fixture data for our prediction or whether to use the data from the GitLab repository (currently week 30). We can set this by going to the Python file at '/python/data.py' and setting the 'live = True' to use live data or 'live = False' for non-live data.

Before the decision is made, lets review the positives and negatives of each:

Live data:

- Live team prediction from gameweek 1 to next week's gameweek
- Takes around two to two and a half minutes to run the program (depending on your machine)

Non-live data:

- Team prediction from gameweek 1 to gameweek 30 (or later if the data is replaced with later commits from the GitHub Repository)
- Takes around 15 to 25 seconds to run the program (depending on your machine)

7.3 Running Python

The program can be run from python file '/python/main.py', which produces a summary of our solved linear programming model and an output of our initial team. We then see how many points our initial team would get over the course of the season with no transfers. We are then shown the transfers made for each week, how many points we get and finally a points total over the season so far. It takes around two minutes to run with live data and 15 seconds with non-live data.

7.4 Running Java

When we run our java program 'src/src/main/java/Main.java', we are shown an application consisting of tabs; 'Fantasy', 'Players', 'League Table', 'Results' and 'Fixtures'. When we run this Java file, all of the Python scripts we use are run also. It takes around two and a half minutes to run with live data and 25 seconds with non-live data. It then takes a further 40 seconds more to sign in with a manager ID.

CHAPTER 8

Conclusion

The overall application tool we have produced has a very simple and appealing interface, with a decent amount of functionality, enough for it's purpose. Each section of the program could have been improved if given more time, but a valient effort was made across the board to have a fully-functioning program. Creating an application for a game made by one of the largest companies in a sector is so difficult, as these companies know how important it is to gate-keep certain parts of their business in order to stay at the top. For example, actually being able to make changes to our team in a fully interactive way would have been a very nice feature to implement. However, it is not possible for any third-party application to do so.

All of the tabs in our application are informative, with the most useful being the fantasy tab. Not only did we manage to make a prediction for the entire season of football so far, we also found a way to show our prediction in a very visually appealing way.

With this being such a large project to take on, it is understandable that we did not manage to create a perfect prediction model, neither did we manage to create a flawless application which could compete with the likes of the Premier League. This was a chance to learn and practice new skills from both a mathematical prediction and a graphical design viewpoint. The most important aspect we can take from this project is that it was built correctly, meaning that improvements can be made with ease.

Bibliography

[1] Premier League. 'Statistics Explained'. *Premier League, https://www.premierleague.com/stats/clarification. [Last accessed: 16 November 2021]*

[2] IBM Cloud Education. 'Application Programming Interface (API)'. *IMB, https://www.ibm.com/cloud/learn/api. [Last accessed: 16 November 2021]*

[3] Unknown. 'Requests: HTTP for Humans'. *Requests http for humans, https://docs.python-requests.org/en/latest/. [Last accessed: 15 November 2021]*

[4] Unknown. 'Data Preprocessing'. *techopedia, 11 July 2021, https://www.techopedia.com/definition/14650/data-preprocessing. [Last accessed: 15 November 2021]*

[5] Rajaratne, Maneesha. 'Data Pre Processing Techniques You Should Know'. *towards data science, 2 December 2018, https://towardsdatascience.com/data-pre-processing-techniques-you-should-know-8954662716d6.*

[6] Timothy, Frenzel. 'Fantasy Premier League API Endpoints: A Detailed Guide'. *Medium, 8 October 2020, https://medium.com/@frenzelts/fantasy-premier-league-api-endpoints-a-detailed-guide-acbd5598eb19.*

[7] Rob Hruska and Ravi. 'How to get HTTP response code for a URL in Java?'. *Stack overflow, 4 February 2018, https://stackoverflow.com/questions/6467848/how-to-get-http-response-code-for-a-url-in-java.*

[8] Unknown. 'Check for NaN in Pandas DataFrame (examples included)'. *Data to Fish, 10 September 2021, https://datatofish.com/check-nan-pandas-dataframe/.*

[9] Unknown. 'pandas.Series.map'. *Pandas, https://pandas.pydata.org/docs/reference/api/pandas.Series.map.*

[10] Unknown. 'Python unidecode.unidecode() Examples'. *Unidecode, https://www.programcreek.com/python/example/94318/unidecode.unidecode.*

[11] Gavin Ng. 'FPL-prediction-and-selection'. *Kaggle, https://www.kaggle.com/code/gavinjpng/fpl-prediction-and-selection/notebook.*

[12] Unknown. 'Optimization with PuLP'. *PuLP, https://coin-or.github.io/pulp/.*

[13] Unknown. 'The lpProblem Class'. *PuLP, https://www.coin-or.org/PuLP/pulp.html#pulp.LpProblem.*

[14] Unknown. 'Machine Learning in Python'. *skikit-learn, https://scikit-learn.org/stable/.*