Predict Football Match Winners 🏆

Table of Contents:

- [Introduction]
- [Data Collection & Cleaning]
    - [Data Download]
    - [Scrape Data for Single Team & Single Season]
    - [Scrape Data for Multiple Teams & Multiple Years]
    - [Concatenate Data Frames]
    - [Display Web-Scraped Data Frame]
    - [Data Cleaning]
    - [Remove Unnecessary Variables]
    - [Create New Variables]
- [Machine Learning]
    - [Random Forest]
    - [Why Random Forest?]
    - [Training Algorithm]
    - [Prediction & Accuracy]
    - [Improving the Model]
    - [Predictions Function]
    - [Matching Results]
- [Conclusion]
    - [Further Development Suggestions]

Introduction ⚽

Welcome to this machine learning project with **Python**!

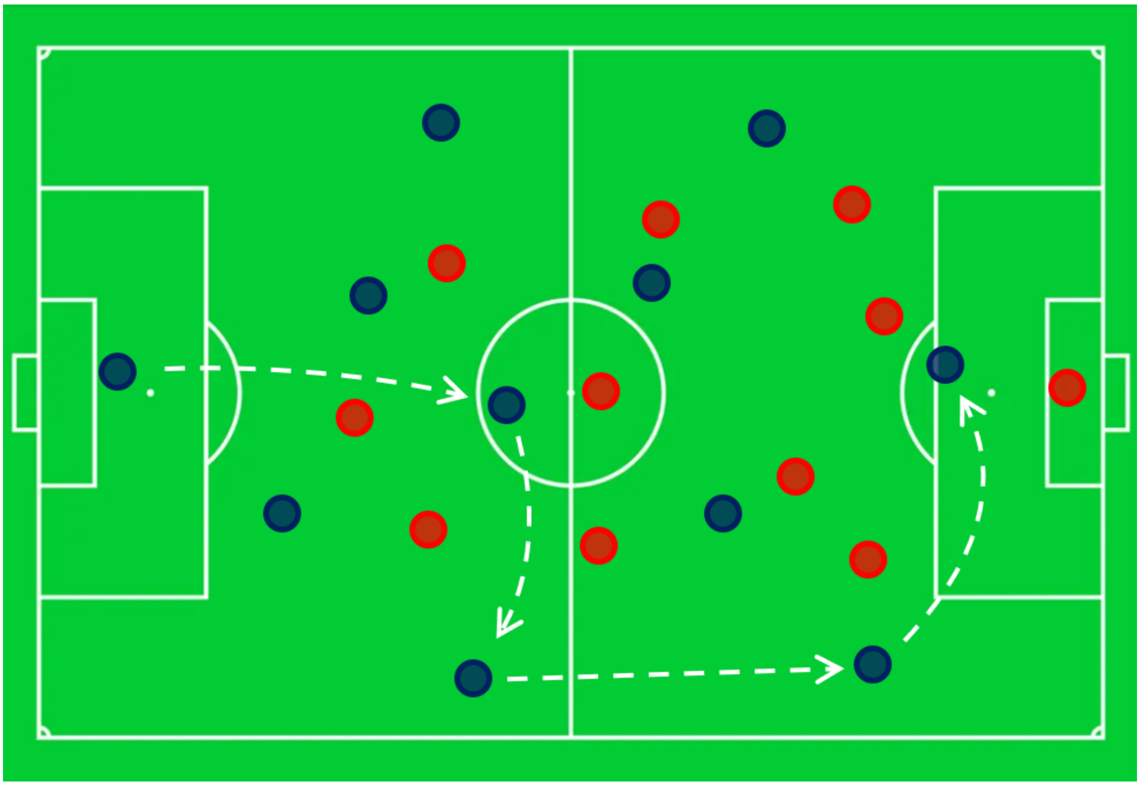
In this project, we will be analysing match data on **English Premier League (EPL)** matches to ultimately try and build a simple machine learning model that will predict football match results.

This project comprises two main sections:

- Web-scraping football match data
- Building machine learning model to predict match results

In the first section, we will be scraping web data on match statics from the FB Ref (https://fbref.com/en/comps/9/Premier-League-Stats) website. This is an easy-to-use source for football stats including player, team, and league stats.

The second section involves building a machine learning model to try and predict the outcomes of football matches. The model that we will be employing is a **Random Forest**. After building an initial model, we will assess its predictive accuracy prior to trying to optimize its predictive power by enriching the data set with more predictors.



Data Collection & Cleaning ⚽

We will be using data on English Premier League football matches. To get the data, we will need to scrape the match results from a website.

To begin, we will download the data using *Python*'s **requests** library. This will return *html* text data that we must then parse using the **BeautifulSoup** library enabling us to extract the relevant statistics tables. Finally, we will load everything into a **pandas** data frame in order to clean and prepare the data for building our machine learning model.

The data we are using will be scraped from the [FB Ref (https://fbref.com/en/comps/9/Premier-League-Stats)](https://fbref.com/en/comps/9/Premier-League-Stats) website.

Data Download ⚽

In [1]:
```
import requests
```

In [2]:
```
standings_url = "https://fbref.com/en/comps/9/Premier-League-Stats"
```

We now need to use the **.get( )** method to make a request to the server and download *html* as a text file from the webpage.

In [3]:
```
data = requests.get(standings_url)
```

Using the command **data.text** would return to us a very long string of *html*. We will not perform this action, however, as the text is barely legible.

On the website, each squad (team) has a hyperlink to that will take you to its own page. These links are highlighted below in the **red box**. On these pages, there is a lot of information about games that has been collected, and we will be using to train out machine learning model.

Overall | Home/Away

Rk	Squad	MP	W	D	L	GF	GA	GD	Pts	Pts/MP	xG	xGA	xGD	xGD/90	Last 5	Attendance	Top Team Scorer	Goalkeeper	Notes
1	Arsenal	20	16	2	2	45	17	+28	50	2.50	38.2	18.6	+19.6	+0.98	W D W W L	60,175	Martin Ødegaard - 8	Aaron Ramsdale	
2	Manchester City	21	14	3	4	53	21	+32	45	2.14	42.7	17.0	+25.7	+1.22	W L W W L	53,185	Erling Haaland - 25	Ederson	
3	Manchester Utd	22	13	4	5	36	28	+8	43	1.95	33.7	25.4	+8.3	+0.38	W D L W D	73,793	Marcus Rashford - 11	David de Gea	
4	Newcastle Utd	21	10	10	1	34	12	+22	40	1.90	33.8	18.7	+15.1	+0.72	D D W D D	52,194	Miguel Almirón - 9	Nick Pope	
5	Tottenham	22	12	3	7	41	31	+10	39	1.77	32.0	26.8	+5.1	+0.23	W L L W W	61,691	Harry Kane - 17	Hugo Lloris	
6	Brighton	20	10	4	6	38	27	+11	34	1.70	31.3	25.1	+6.2	+0.31	L W W D W	31,487	Leandro Trossard - 7	Robert Sánchez	
7	Brentford	21	8	9	4	35	28	+7	33	1.57	29.5	26.3	+3.2	+0.15	W W W D W	17,076	Ivan Toney - 13	David Raya	
8	Fulham	22	9	5	8	32	30	+2	32	1.45	29.1	36.8	-7.7	-0.35	W W L L D	23,270	Aleksandar Mitrović - 11	Bernd Leno	
9	Chelsea	21	8	6	7	22	21	+1	30	1.43	25.2	27.0	-1.9	-0.09	L L W D D	39,957	Kai Havertz - 5	Kepa Arrizabalaga	
10	Liverpool	20	8	5	7	34	28	+6	29	1.45	35.4	29.2	+6.2	+0.31	W L L D L	53,228	Mohamed Salah, Roberto Firmino - 7	Alisson	
11	Aston Villa	21	8	4	9	25	31	-6	28	1.33	26.3	29.1	-2.8	-0.13	W D W W L	41,627	Danny Ings - 6	Emiliano Martínez	
12	Crystal Palace	21	6	6	9	19	29	-10	24	1.14	19.6	29.6	-10.0	-0.48	L L D D L	24,712	Wilfried Zaha - 6	Vicente Guaita	
13	Nott'ham Forest	21	6	6	9	17	35	-18	24	1.14	23.4	31.3	-7.9	-0.37	D W W D W	29,110	Brennan Johnson - 5	Dean Henderson	
14	Leicester City	21	6	3	12	32	37	-5	21	1.00	25.3	33.8	-8.5	-0.41	L L L D W	31,646	James Maddison - 8	Danny Ward	
15	Wolves	21	5	5	11	15	30	-15	20	0.95	20.9	30.7	-9.8	-0.47	L D W L W	31,616	Rúben Neves, Daniel Podence - 5	José Sá	
16	Leeds United	21	4	7	10	28	36	-8	19	0.90	27.2	32.2	-5.0	-0.24	D L D L D	36,441	Rodrigo - 10	Illan Meslier	
17	West Ham	21	5	4	12	18	26	-8	19	0.90	26.6	23.3	+3.3	+0.16	L D L W D	62,449	Jarrod Bowen - 4	Łukasz Fabiański	
▼ 18	Everton	21	4	6	11	16	28	-12	18	0.86	22.1	34.4	-12.3	-0.59	D L L L W	39,222	Demarai Gray, Anthony Gordon - 3	Jordan Pickford	
▼ 19	Bournemouth	21	4	5	12	19	43	-24	17	0.81	16.4	35.3	-18.9	-0.90	L L L D L	10,338	Philip Billing, Kieffer Moore - 4	Neto, Mark Travers	
▼ 20	Southampton	21	4	3	14	17	38	-21	15	0.71	20.3	28.2	-7.9	-0.38	L L W L L	30,609	James Ward-Prowse - 5	Gavin Bazunu	

Scrape Data for Single Team & Single Season ⚽

Before scraping match data for multiple teams and multiple years, we will run through the process for a single team and a single season, step-by-step, in order to build intuition.

On each team's page, there is a match log, and we want to be able to extract this data. In order to extract this data, we need the URLs for each team's page containing the match log. To do this, we will parse our *html* text file using the **BeautifulSoup** library.

After importing the **BeautifulSoup** library, we will:

1. Create a BeautifulSoup object and initialise this by feeding it our *html* text file
2. Use a CSS selector to give the object a table to select
3. Select the anchor tags with the links that we want from the table shown above
4. Retrieve **href** property of each link using a list comprehension
5. Filter links to get only squad / team links

In [4]:
```
from bs4 import BeautifulSoup
```

```python
# (1) - Create and initialise the object
soup = BeautifulSoup(data.text)

# (2) - Use .select() to get first table with class stats_table
standings_table = soup.select('table.stats_table')[0]

# (3) - Find all <a> tags
links = standings_table.find_all('a')

# (4) - Get href for each link
links = [l.get("href") for l in links]

# (5) - Filter links
links = [l for l in links if '/squads/' in l]
```

The above code will return the links without the domain address. To turn our links into full URLs (or *absolute links*), we need to attach the domain name onto the front of each link using a *format string*.

```python
team_urls = [f"https://fbref.com{l}" for l in links]
```

We will work with the first team's URL, i.e. the team currently leading in the Premier League.

```python
data = requests.get(team_urls[0])
```

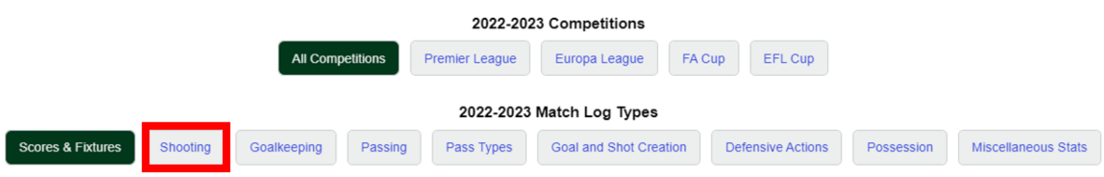
In the team's webpage, there is a table called **Scores & Fixtures** that contains information such as the date of the matches, the goals each team scored, the results, etc.

In order to retrieve the data from this table, we will turn it into a **pandas** data frame using the **.html( )** method, as follows.

```python
import pandas as pd

# Match scans for a specific string inside the table; this will return a li
matches = pd.read_html(data.text, match="Scores & Fixtures")[0]
```

From this page, we also want to extract data on shooting statistics by following the **Shooting** tab link to another webpage.



Let us find the URL of this **Shooting** page. After that, the process will be similar to what we have done before.

In [9]:
```
soup = BeautifulSoup(data.text)
links = soup.find_all('a')
links = [l.get("href") for l in links]
links = [l for l in links if l and 'all_comps/shooting/' in l]
```

In [10]:
```
data = requests.get(f"https://fbref.com{links[0]}")
```

In [11]:
```
shooting = pd.read_html(data.text, match="Shooting")[0]
```

In [12]:
```
# Display head of shooting data
shooting.head()
```

Out[12]:

										For Manchester City	...		
	Date	Time	Comp	Round	Day	Venue	Result	GF	GA	Opponent	...	Dist	
0	2021-08-07	17:15	Community Shield	FA Community Shield	Sat	Neutral	L	0	1	Leicester City	...	NaN	N
1	2021-08-15	16:30	Premier League	Matchweek 1	Sun	Away	L	0	1	Tottenham	...	16.9	
2	2021-08-21	15:00	Premier League	Matchweek 2	Sat	Home	W	5	0	Norwich City	...	17.3	
3	2021-08-28	12:30	Premier League	Matchweek 3	Sat	Home	W	5	0	Arsenal	...	14.3	
4	2021-09-11	15:00	Premier League	Matchweek 4	Sat	Away	W	1	0	Leicester City	...	14.0	

5 rows × 26 columns

You may notice above that we appear to have a multi-level index. This will cause problems if we want to index based on - for example - **Round** or **GF**, so we need to remove this index level.

In [13]:
```
# Drop top index level
shooting.columns = shooting.columns.droplevel()
```

Finally, we must merge the two data frames together using the **.merge( )** method. We only want to merge the following columns, from the Shooting data frame:

- **Date**: Match date
- **Sh** : Shots
- **SoT** : Shots-on-target
- **Dist** : Average distance travelled by a shot
- **FK** : Free-kicks
- **PK** : Penalty kicks
- **PKatt** : Penalty kicks attempted

In [14]:
```
team_data = matches.merge(shooting[["Date", "Sh", "SoT", "Dist", "FK", "PK"
```

In [15]:
```
team_data.head()
```

Out[15]:

	Date	Time	Comp	Round	Day	Venue	Result	GF	GA	Opponent	...	Formati
0	2021-08-07	17:15	Community Shield	FA Community Shield	Sat	Neutral	L	0	1	Leicester City	...	4-
1	2021-08-15	16:30	Premier League	Matchweek 1	Sun	Away	L	0	1	Tottenham	...	4-
2	2021-08-21	15:00	Premier League	Matchweek 2	Sat	Home	W	5	0	Norwich City	...	4-
3	2021-08-28	12:30	Premier League	Matchweek 3	Sat	Home	W	5	0	Arsenal	...	4-
4	2021-09-11	15:00	Premier League	Matchweek 4	Sat	Away	W	1	0	Leicester City	...	4-

5 rows × 25 columns

What we have done so far is to scrape the standings prior to downloading match and shooting statistics for a single team before combining this information into a single data frame.

Next, we need to scale this method up and scrape data for multiple teams for multiple years.

Scrape Data for Multiple Teams & Multiple Years ⚽

To scrape match data for multiple teams from multiple years, we will need to create a **for loop**.

First, we will create one list object containing the years we wish to pull data from, and also an empty list to store all of our data frames. Each data frame will contain the match logs for one team for one season.

In [16]:
```
years = list(range(2022, 2020, -1))
all_matches = []

standings_url = "https://fbref.com/en/comps/9/Premier-League-Stats"
```

Now we will write a *for loop*. The below code may seem intimidating but it is predominantly what we have already done with a single team and a single season:

1. Loop through each year in the years list
2. Retrive URLs for each team's webpage
3. Loop through each team's webpage
4. Extract the 'Scores & Fixtures' table
5. Retrieve URLs for Shooting webpage
6. Extract Shooting table

Additional stages are

7. Wrap merge in try-block as some data is not always available and may cause merging issues
8. Filter on Premier League matches
9. Add additional columns to each data frame, to identify each data frame team and year
10. Add data frame to data frame list
11. Sleep for 1 second - this is important as some websites may block you if you make too many requests in a short period

In [17]:
```python
import time

# (1) - Loop through the years
for year in years:

    # (2) - Retrieve the Links for each team page
    data = requests.get(standings_url)
    soup = BeautifulSoup(data.text)
    standings_table = soup.select('table.stats_table')[0]

    links = [l.get("href") for l in standings_table.find_all('a')]
    links = [l for l in links if '/squads/' in l]
    team_urls = [f"https://fbref.com{l}" for l in links]

    # Retrieve the Links for previous season
    previous_season = soup.select("a.prev")[0].get("href")
    standings_url = f"https://fbref.com{previous_season}"

    # (3) - Loop through each team url
    for team_url in team_urls:

        # Extract team name from url
        team_name = team_url.split("/")[-1].replace("-Stats", "").replace("

        # Retrieve team URL
        data = requests.get(team_url)

        # (4) - Extract Scores and Fixtures table
        matches = pd.read_html(data.text, match="Scores & Fixtures")[0]

        # (5) - Retrieve URL for Shooting page
        soup = BeautifulSoup(data.text)
        links = [l.get("href") for l in soup.find_all('a')]
        links = [l for l in links if l and 'all_comps/shooting/' in l]
        data = requests.get(f"https://fbref.com{links[0]}")

        # (6) - Extract Shooting table
        shooting = pd.read_html(data.text, match="Shooting")[0]
        shooting.columns = shooting.columns.droplevel()

        # (7) - Wrap merge in a try statement
        try:
            team_data = matches.merge(shooting[["Date", "Sh", "SoT", "Dist"
        except ValueError:
            continue

        # (8) - Filter for Premier League matches
        team_data = team_data[team_data["Comp"] == "Premier League"]

        # (9) - Add additional columns
        team_data["Season"] = year
        team_data["Team"] = team_name

        # (10) - Add data frame to all_matches list
        all_matches.append(team_data)

        # (11) - Sleep for 1 second
        time.sleep(1)
```

In [18]:
```
# Check how many data frames we have
len(all_matches)
```

Out[18]:
```
39
```

Concatenate Data Frames ⚽

We have ended up with 39 different data frames that we need to concatenate into one large one.

In [19]:
```
match_df = pd.concat(all_matches)
```

As a bit of house-keeping, we will also cast each column name to lower-case.

In [20]:
```
# Lower-case all of the columns
match_df.columns = [c.lower() for c in match_df.columns]
```

Display Web-Scraped Data Frame ⚽

We finally have the data that we are going to use, stored as a data frame with 1389 observations and 27 variables.

In [21]:
```
match_df
```

Out[21]:

	date	time	comp	round	day	venue	result	gf	ga	opponent	...	match report	n
1	2021-08-15	16:30	Premier League	Matchweek 1	Sun	Away	L	0	1	Tottenham	...	Match Report	
2	2021-08-21	15:00	Premier League	Matchweek 2	Sat	Home	W	5	0	Norwich City	...	Match Report	
3	2021-08-28	12:30	Premier League	Matchweek 3	Sat	Home	W	5	0	Arsenal	...	Match Report	
4	2021-09-11	15:00	Premier League	Matchweek 4	Sat	Away	W	1	0	Leicester City	...	Match Report	
6	2021-09-18	15:00	Premier League	Matchweek 5	Sat	Home	D	0	0	Southampton	...	Match Report	
...	...	...	...	...	...	...	...	...	...	...	...	...	
38	2021-05-02	19:15	Premier League	Matchweek 34	Sun	Away	L	0	4	Tottenham	...	Match Report	
39	2021-05-08	15:00	Premier League	Matchweek 35	Sat	Home	L	0	2	Crystal Palace	...	Match Report	
40	2021-05-16	19:00	Premier League	Matchweek 36	Sun	Away	W	1	0	Everton	...	Match Report	
41	2021-05-19	18:00	Premier League	Matchweek 37	Wed	Away	L	0	1	Newcastle Utd	...	Match Report	
42	2021-05-23	16:00	Premier League	Matchweek 38	Sun	Home	W	1	0	Burnley	...	Match Report	

1389 rows × 27 columns

We want to save this to file, so we can continue to work on it without having to make unnecessary requests to the host website and risk getting blocked!

In [22]:
```
# Write to CSV file
match_df.to_csv("matches.csv")
```

If you are following along with the project but are struggling to get the web-scraping part to work, or may have gotten blocked by the web-server, **don't panic**! You can download the data that will be stored in my GitHub folder for this project. Click [here](https://github.com/th3RFC/portfolio/tree/dc3b22a14ec3baf6d4805c87aae046511fde6901/pytl (https://github.com/th3RFC/portfolio/tree/dc3b22a14ec3baf6d4805c87aae046511fde6901/pytl
This will enable you to proceed with the next parts of the project.

Data Cleaning ⚽

The next stage in this project is to clean the data. The code in this next section can be run independently of the web-scraping code above. It only assumes that you have downloaded the match.csv file from my GitHub folder.

To begin, we will read the data into our notebook as a **pandas** dataframe.

While we have already imported the *pandas* library in the web-scraping part of the project, we will re-import the module so that this code will work independently of code previously written.

In [23]:
```
import pandas as pd
```

In [24]:
```
matches = pd.read_csv("matches.csv", index_col=0)
```

In [25]:
```
matches.head()
```

Out[25]:

	date	time	comp	round	day	venue	result	gf	ga	opponent	...	match report	n
1	2021-08-15	16:30	Premier League	Matchweek 1	Sun	Away	L	0.0	1.0	Tottenham	...	Match Report	
2	2021-08-21	15:00	Premier League	Matchweek 2	Sat	Home	W	5.0	0.0	Norwich City	...	Match Report	
3	2021-08-28	12:30	Premier League	Matchweek 3	Sat	Home	W	5.0	0.0	Arsenal	...	Match Report	
4	2021-09-11	15:00	Premier League	Matchweek 4	Sat	Away	W	1.0	0.0	Leicester City	...	Match Report	
6	2021-09-18	15:00	Premier League	Matchweek 5	Sat	Home	D	0.0	0.0	Southampton	...	Match Report	

5 rows × 27 columns

We can use the **.shape( )** method to find the dimensions of our data frame. This will provide us with information on the number of observations and the number of variables per observation.

In [26]:
```
matches.shape
```

Out[26]: (1389, 27)

Thus, we have **1389** observations and **27** variables in our data frame, whereby each observation represents a single match. The variables include statistics on which team won or lost, goals for each team, shooting statistics and so on.

However, in the web-scraping part of the course, we looped over **2 seasons** of EPL matches, with **20 squads** per season, with each team playing **38 matches** per season. Do our numbers add up?

In [27]:
```
# 2 seasons * 20 squads * 38 matches
1389 - (2 * 20 * 38)
```

Out[28]: -131

It seems that our data frame is short by 131 observations (matches). Why?

We can use the **.value_counts( )** method to begin investigating this.

In [28]:
```
# Number of matches per team in EPL
matches["team"].value_counts().to_frame()
```

Out[28]:

	team
Southampton	72
Brighton and Hove Albion	72
Manchester United	72
West Ham United	72
Newcastle United	72
Burnley	71
Leeds United	71
Crystal Palace	71
Manchester City	71
Wolverhampton Wanderers	71
Tottenham Hotspur	71
Arsenal	71
Leicester City	70
Chelsea	70
Aston Villa	70
Everton	70
Liverpool	38
Fulham	38
West Bromwich Albion	38
Sheffield United	38
Brentford	34
Watford	33
Norwich City	33

We can see that most teams have played approximately 70-72 matches in the EPL. Given that this data was drawn part-way through a season, this is understandable.

We can also see 7 teams with 38 or fewer matches played. Each season in the EPL, 3 teams are relegated and move down to the **Championship** league, while 3 teams end up being promoted from the Championship to the EPL. This would explain why 6 teams have fewer games played ... but we have 7. Why?

Liverpool having only 38 matches played looks a little bit suspect, as Liverpool is a strong team that tends to do well. It can be independently verified that Liverpool has not been relegated/ promoted in the last 2 EPL seasons. This would suggest that not all the matches for Liverpool have been pulled through. To investigate this further, we will filter on Liverpool.

In [29]:

```
matches[matches["team"] == "Liverpool"].sort_values("date")
```

Out[29]:

	date	time	comp	round	day	venue	result	gf	ga	opponent	...	match report
1	2020-09-12	17:30	Premier League	Matchweek 1	Sat	Home	W	4.0	3.0	Leeds United	...	Match Report
2	2020-09-20	16:30	Premier League	Matchweek 2	Sun	Away	W	2.0	0.0	Chelsea	...	Match Report
4	2020-09-28	20:00	Premier League	Matchweek 3	Mon	Home	W	3.0	1.0	Arsenal	...	Match Report
6	2020-10-04	19:15	Premier League	Matchweek 4	Sun	Away	L	2.0	7.0	Aston Villa	...	Match Report
7	2020-10-17	12:30	Premier League	Matchweek 5	Sat	Away	D	2.0	2.0	Everton	...	Match Report
9	2020-10-24	20:00	Premier League	Matchweek 6	Sat	Home	W	2.0	1.0	Sheffield Utd	...	Match Report
11	2020-10-31	17:30	Premier League	Matchweek 7	Sat	Home	W	2.0	1.0	West Ham	...	Match Report
13	2020-11-08	16:30	Premier League	Matchweek 8	Sun	Away	D	1.0	1.0	Manchester City	...	Match Report
14	2020-11-22	19:15	Premier League	Matchweek 9	Sun	Home	W	3.0	0.0	Leicester City	...	Match Report
16	2020-11-28	12:30	Premier League	Matchweek 10	Sat	Away	D	1.0	1.0	Brighton	...	Match Report
18	2020-12-06	19:15	Premier League	Matchweek 11	Sun	Home	W	4.0	0.0	Wolves	...	Match Report
20	2020-12-13	16:30	Premier League	Matchweek 12	Sun	Away	D	1.0	1.0	Fulham	...	Match Report
21	2020-12-16	20:00	Premier League	Matchweek 13	Wed	Home	W	2.0	1.0	Tottenham	...	Match Report
22	2020-12-19	12:30	Premier League	Matchweek 14	Sat	Away	W	7.0	0.0	Crystal Palace	...	Match Report
23	2020-12-27	16:30	Premier League	Matchweek 15	Sun	Home	D	1.0	1.0	West Brom	...	Match Report
24	2020-12-30	20:00	Premier League	Matchweek 16	Wed	Away	D	0.0	0.0	Newcastle Utd	...	Match Report
25	2021-01-04	20:00	Premier League	Matchweek 17	Mon	Away	L	0.0	1.0	Southampton	...	Match Report
27	2021-01-17	16:30	Premier League	Matchweek 19	Sun	Home	D	0.0	0.0	Manchester Utd	...	Match Report
28	2021-01-21	20:00	Premier League	Matchweek 18	Thu	Home	L	0.0	1.0	Burnley	...	Match Report
30	2021-01-28	20:00	Premier League	Matchweek 20	Thu	Away	W	3.0	1.0	Tottenham	...	Match Report
31	2021-01-31	16:30	Premier League	Matchweek 21	Sun	Away	W	3.0	1.0	West Ham	...	Match Report
32	2021-02-03	20:15	Premier League	Matchweek 22	Wed	Home	L	0.0	1.0	Brighton	...	Match Report
33	2021-02-07	16:30	Premier League	Matchweek 23	Sun	Home	L	1.0	4.0	Manchester City	...	Match Report
34	2021-02-13	12:30	Premier League	Matchweek 24	Sat	Away	L	1.0	3.0	Leicester City	...	Match Report
36	2021-02-20	17:30	Premier League	Matchweek 25	Sat	Home	L	0.0	2.0	Everton	...	Match Report

	date	time	comp	round	day	venue	result	gf	ga	opponent	...	match report
37	2021-02-28	19:15	Premier League	Matchweek 26	Sun	Away	W	2.0	0.0	Sheffield Utd	...	Match Report
38	2021-03-04	20:15	Premier League	Matchweek 29	Thu	Home	L	0.0	1.0	Chelsea	...	Match Report
39	2021-03-07	14:00	Premier League	Matchweek 27	Sun	Home	L	0.0	1.0	Fulham	...	Match Report
41	2021-03-15	20:00	Premier League	Matchweek 28	Mon	Away	W	1.0	0.0	Wolves	...	Match Report
42	2021-04-03	20:00	Premier League	Matchweek 30	Sat	Away	W	3.0	0.0	Arsenal	...	Match Report
44	2021-04-10	15:00	Premier League	Matchweek 31	Sat	Home	W	2.0	1.0	Aston Villa	...	Match Report
46	2021-04-19	20:00	Premier League	Matchweek 32	Mon	Away	D	1.0	1.0	Leeds United	...	Match Report
47	2021-04-24	12:30	Premier League	Matchweek 33	Sat	Home	D	1.0	1.0	Newcastle Utd	...	Match Report
48	2021-05-08	20:15	Premier League	Matchweek 35	Sat	Home	W	2.0	0.0	Southampton	...	Match Report
49	2021-05-13	20:15	Premier League	Matchweek 34	Thu	Away	W	4.0	2.0	Manchester Utd	...	Match Report
50	2021-05-16	16:30	Premier League	Matchweek 36	Sun	Away	W	2.0	1.0	West Brom	...	Match Report
51	2021-05-19	20:15	Premier League	Matchweek 37	Wed	Away	W	3.0	0.0	Burnley	...	Match Report
52	2021-05-23	16:00	Premier League	Matchweek 38	Sun	Home	W	2.0	0.0	Crystal Palace	...	Match Report

38 rows × 27 columns

It would appear that we only have data for the **2020/21** season for Liverpool, and we are missing data for the **2021/22** season. The data should still be fine to work with, and at least we have an explanation for why some of the data is missing.

Next, we will look at the variable **round**. This gives the Matchweek that each game was played in. In an EPL season there are **38 Matchweeks**.

For most match weeks, we would typically expect the count to be **40** for each Matchweek, as we count one matchweek per team, and we have **20 teams** playing in each matchweek, across **2 seasons**, thus $20 \times 2 = 40$. However, we are missing Liverpool's 2021/22 season, so the maximum number we would expect is a count of 39.

Also, this data was scraped mid-season for the 2021/22 season, thus some of the later matchweeks will not have been played yet.

In [30]:

```
matches["round"].value_counts().to_frame()
```

Out[30]:

	round
Matchweek 1	39
Matchweek 16	39
Matchweek 34	39
Matchweek 32	39
Matchweek 31	39
Matchweek 29	39
Matchweek 28	39
Matchweek 26	39
Matchweek 25	39
Matchweek 24	39
Matchweek 23	39
Matchweek 2	39
Matchweek 19	39
Matchweek 17	39
Matchweek 20	39
Matchweek 15	39
Matchweek 5	39
Matchweek 3	39
Matchweek 13	39
Matchweek 12	39
Matchweek 4	39
Matchweek 11	39
Matchweek 10	39
Matchweek 9	39
Matchweek 8	39
Matchweek 14	39
Matchweek 7	39
Matchweek 6	39
Matchweek 30	37
Matchweek 27	37
Matchweek 22	37
Matchweek 21	37
Matchweek 18	37
Matchweek 33	32
Matchweek 35	20
Matchweek 36	20
Matchweek 37	20
Matchweek 38	20

We can see that we have 39 instances of each matchweek, as expected, but fewer counts for matchweeks 33 onwards. This is because the last matchweek in the $2021/22$ season that had been played was the 32^{nd} matchweek. This is fine.

Now, we will check the data types of our variables using the **.dtypes** method. This is important as machine learning algorithms can only worth with numeric data.

If the column is stored as a different data type - such as *object* which typically denotes a string - then we have to find a way of converting these data types to numeric data in order to use them as predictors in our machine learning algorithm.

In [31]:
```
type = matches.dtypes.to_frame().rename(columns={0:" Data Type"})
type
```

Out[31]:

	Data Type
date	object
time	object
comp	object
round	object
day	object
venue	object
result	object
gf	float64
ga	float64
opponent	object
xg	float64
xga	float64
poss	float64
attendance	float64
captain	object
formation	object
referee	object
match report	object
notes	float64
sh	float64
sot	float64
dist	float64
fk	float64
pk	float64
pkatt	float64
season	int64
team	object

Remove Unnecessary Variables ⚽

Given that we have already filtered out matches that were not Premier League matches, the **comp** variable is now no longer relevant, so this can be removed. Also, we cannot easily convert **notes** to a numeric variable, so this can also be dropped.

In [32]:
```
del matches["comp"]
```

In [33]:
```
del matches["notes"]
```

Create New Variables ⚽

The date variable appears to be stored as a string. We will over-write the existing column by converting it to a **datetime** data type using the **.to_datetime** method.

This will make it easier for us to use the date variable to compute predictors. For example, you could extract the month, or week day of the match.

In [34]:
```
matches["date"] = pd.to_datetime(matches["date"])
```

We also require a **target** variable to denote whether or not the team won a match.

In [35]:
```
matches["target"] = (matches["result"] == "W").astype("int")
```

The **venue_code** variable gives details on whether the team played at their home stadium, or at another team's stadium. This could be important, as the support of the home crowd can have a significant influence on the morale and performance of a team.

Currently, this variable is stored as a string, so we will convert it to a categorical data type using the **.astype( )** method. We will convert these categories to integers using the **.cat.codes( )** method. This will assign the value **0** for Away games and **1** for Home games.

In [36]:
```
matches["venue_code"] = matches["venue"].astype("category").cat.codes
```

We will repeat this by assigning integer values to the **opponent** variable. Evidently, the opponent will influence the outcome of the match.

In [37]:
```
matches["opp_code"] = matches["opponent"].astype("category").cat.codes
```

Do certain teams play better at certain times of day? To see if this may be the case, we will create a variable for the hour that the match is played.

In [38]:
```
# Use a regular expression to extract the hour
matches["hour"] = matches["time"].str.replace(":.+", "", regex=True).astype
```

We will create one final variable that will return the day of the week that the match is played on, assuming that some teams may play better on - for example - a Sunday versus a Saturday.

In [39]:
```
matches["day_code"] = matches["date"].dt.dayofweek
```

In [40]:
```
# Display data frame
matches
```

Out[40]:

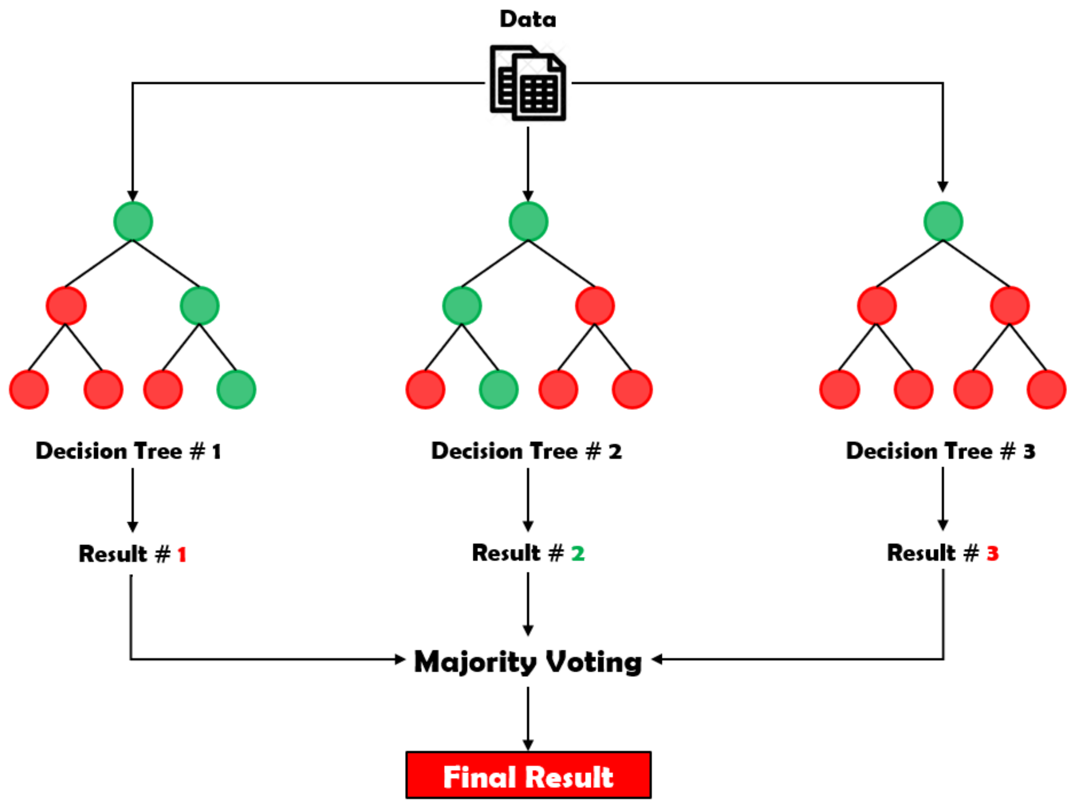
	date	time	round	day	venue	result	gf	ga	opponent	xg	...	fk	pk	pk
1	2021-08-15	16:30	Matchweek 1	Sun	Away	L	0.0	1.0	Tottenham	1.9	...	1.0	0.0	(
2	2021-08-21	15:00	Matchweek 2	Sat	Home	W	5.0	0.0	Norwich City	2.7	...	1.0	0.0	(
3	2021-08-28	12:30	Matchweek 3	Sat	Home	W	5.0	0.0	Arsenal	3.8	...	0.0	0.0	(
4	2021-09-11	15:00	Matchweek 4	Sat	Away	W	1.0	0.0	Leicester City	2.9	...	0.0	0.0	(
6	2021-09-18	15:00	Matchweek 5	Sat	Home	D	0.0	0.0	Southampton	1.1	...	1.0	0.0	(
...	...	...	...	...	...	...	...	...	...	...	...	...	...	
38	2021-05-02	19:15	Matchweek 34	Sun	Away	L	0.0	4.0	Tottenham	0.5	...	0.0	0.0	(
39	2021-05-08	15:00	Matchweek 35	Sat	Home	L	0.0	2.0	Crystal Palace	0.7	...	1.0	0.0	(
40	2021-05-16	19:00	Matchweek 36	Sun	Away	W	1.0	0.0	Everton	1.6	...	0.0	0.0	(
41	2021-05-19	18:00	Matchweek 37	Wed	Away	L	0.0	1.0	Newcastle Utd	0.8	...	1.0	0.0	(
42	2021-05-23	16:00	Matchweek 38	Sun	Home	W	1.0	0.0	Burnley	0.6	...	0.0	0.0	(

1389 rows × 30 columns

Machine Learning ⚽

In this section, we will begin by training our machine learning model after giving a brief justification for the choice of algorithm: **Random Forest**.

After training our model, we will assess its predictive accuracy, and then seek to optimise this and improve the performance of our model.



Random Forest ⚽

Random forest is a commonly-used machine learning algorithm, which combines the output of multiple **decision trees** to reach a single result via *majority voting*.

While its ease of use and flexibility have lent it to both classification and regression problems, we will be using it for a **classification** problem in this project, i.e. classifying whether or not a team will win a match of football.

Why Random Forest? ⚽

The Random Forest model is useful as it can pick up *non-linearities* in the data that some other algorithms may struggle with.

For example, we have created a column for **opponent** and assigned codes to it. A code of 15, for example, does not denote a team that is more or less challenging to play against that a team with code 14 or less; nor is a team with a code of 10 twice as difficult to play against as a team with code 5. They are just values to categorise different opponents numerically. A linear model would not be able to pick that up whereas a Random Forest can.

Training Algorithm ⚽

First, we need to import the **RandomForestClassifier** from the **sklearn** library.

In [41]:
```python
from sklearn.ensemble import RandomForestClassifier
```

Now, we want to create a Random Forest instance by initialising the Random Forest class. Below, we are going to enter in some hyper-paramters:

- **n_estimators** - How many Decision Trees we want in our Random Forest model
- **min_samples_split** - Number of samples we want to have in a leaf of a Decision Tree prior to splitting the node
- **random_state** - Set this to generate the same results each run

In [42]:
```python
rf = RandomForestClassifier(n_estimators=50, min_samples_split=10, random_s
```

The next step is to create a **training** data set and a **testing** data set.

In [43]:
```python
train = matches[matches["date"] < '2022-01-01']
```

In [44]:
```python
test = matches[matches["date"] > '2022-01-01']
```

Create a list of the predictor columns that we created earlier.

In [45]:
```python
predictors = ["venue_code", "opp_code", "hour", "day_code"]
```

Next, **fit** the model to the training data.

In [46]:
```python
rf.fit(train[predictors], train["target"])
```

Out[46]:
```
RandomForestClassifier(min_samples_split=10, n_estimators=50, random_state
=1)
```

Prediction & Accuracy ⚽

So far, we have initialised a Random Forest model and trained the model using our test data set.

Now, we generate **predictions** using the test data set to assess the performance of the model that we have just trained.

In [47]:
```python
preds = rf.predict(test[predictors])
```

To access the performance of our model, we will import the **accuracy_score** module.

Accuracy Score is a metric that returns a score for the **proportion of predictions made correctly**. For example, in this project, the number of 'Wins' and the number of 'Not Wins' that were correctly predicted are added together and divided by the total number of predictions made. If we made 6 correct predictions out of a total of 10 predictions, then our accuracy score would be:

$$\frac{\#\text{ Correct Predictions}}{\#\text{ Total Predictions}} = \frac{6}{10} = 0.6$$

In [48]:
```
from sklearn.metrics import accuracy_score
```

In [49]:
```
accuracy = accuracy_score(test["target"], preds)
```

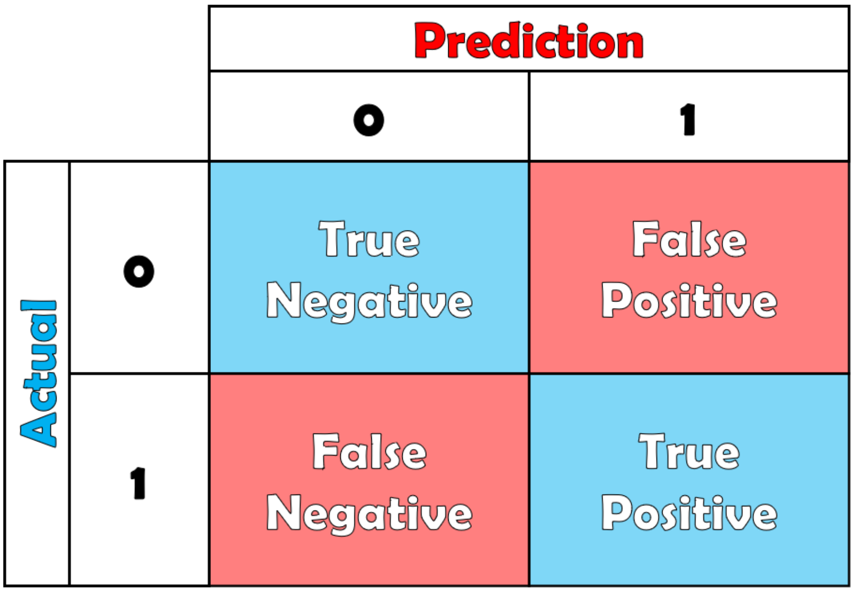
In [50]:
```
accuracy
```

Out[50]:
```
0.6123188405797102
```

A prediction **accuracy** of $\simeq 0.612$ suggests that our model correctly predicted the match result 61.2% of the time.

We can go further, and drill down into this data to see if our model was better at - say - predicting wins versus predicting losses.

To do this, we will create a data frame that combines our actual values versus our predicted values. After this, we can create a **confusion matrix** (see below) using *cross-tabulation*.



In [51]:
```
combined = pd.DataFrame(dict(actual=test["target"], predicted=preds))
```

In [52]:
```
pd.crosstab(index=combined["actual"], columns=combined["predicted"])
```

Out[52]:

predicted	**0**	**1**
actual		
0	141	31
1	76	28

When we predicted a win, we were correct 28 times out of 59 (28 + 31) times, for example. This is approximately 47% of the time, whereas we correctly predicted losses and draws approximately 65% of the time.

It would appear that we are far **better at predicting losses and draws than wins**. Unfortunately, we are more concerned with out model predicting *wins* than losses and draws, so we will have to refine our model.

In light of this, we will revise our accuracy metric, and instead use the **precision score**.

The precision score tells us what proportion of the time we successfully predicted wins:

$$\text{Precision} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}}$$

In [53]:
```
from sklearn.metrics import precision_score

precision_score(test["target"], preds)
```

Out[53]: 0.4745762711864407

This confirms our calculation above, that our precision is only around 47%. This isn't very good, so we are going to improve the model to see how this will affect its predictive ability.

Improving the model ⚽

One way in which we can improve the model is to calculate how well a team had been performing going into a given match. Maybe if they were on a winning streak prior to a match, this may increase the likelihood they will win that match.

To create this variable, we will generate a **rolling average** of their match statistics.

To achieve this, we will create an object called *grouped_matches*. Essentially, this will create a separate data frame for each team in our data set.

In [54]:
```
grouped_matches = matches.groupby("team")
```

After creating this object, we can select a particular team using the **.get_group( )** method. We will use Manchester City as an example.

In [55]:
```
group = grouped_matches.get_group("Manchester City").sort_values("date")
```

This *group* object is a data frame containing all of the matches that Manchester City has played in.

We will create a rolling average function that will take in a group (team), a set of variables (match stats) from our existing data, and a set of new columns that the function will populate with the rolling averages for various match statistics.

We will want this function to:

1. Sort group in ascending order by date
2. Create a variable called rolling_stats that will take variables passed into the function and compute rolling averages
3. Assign rolling averages back to data frame as variables
4. Drop missing values

In [56]:
```python
def rolling_averages(group, cols, new_cols):

    # (1) - Sort group by date
    group = group.sort_values("date")

    # (2) - Create rolling averages
    rolling_stats = group[cols].rolling(3, closed='left').mean() # closed =

    # (3) - Create new data frame columns for rolling averages
    group[new_cols] = rolling_stats

    # (4) - Drop NA
    group = group.dropna(subset=new_cols)

    return group
```

To create our new variables for our rolling averages to populate, we will use string formatting to add a 'rolling'-suffix to the names of existing variables, for example

$$\text{sh} \Rightarrow \text{sh\_rolling}$$

In [57]:
```python
cols = ["gf", "ga", "sh", "sot", "dist", "fk", "pk", "pkatt"]
new_cols = [f"{c}_rolling" for c in cols]

rolling_averages(group, cols, new_cols)
```

Out[57]:

	date	time	round	day	venue	result	gf	ga	opponent	xg	...	hour	day_cod
5	2020-10-17	17:30	Matchweek 5	Sat	Home	W	1.0	0.0	Arsenal	1.5	...	17	
7	2020-10-24	12:30	Matchweek 6	Sat	Away	D	1.0	1.0	West Ham	1.1	...	12	
9	2020-10-31	12:30	Matchweek 7	Sat	Away	W	1.0	0.0	Sheffield Utd	1.5	...	12	
11	2020-11-08	16:30	Matchweek 8	Sun	Home	D	1.0	1.0	Liverpool	1.6	...	16	
12	2020-11-21	17:30	Matchweek 9	Sat	Away	L	0.0	2.0	Tottenham	1.3	...	17	
...	...	...	...	...	...	...	...	...	...	...	...	...	.
42	2022-03-14	20:00	Matchweek 29	Mon	Away	D	0.0	0.0	Crystal Palace	2.3	...	20	
44	2022-04-02	15:00	Matchweek 31	Sat	Away	W	2.0	0.0	Burnley	1.8	...	15	
46	2022-04-10	16:30	Matchweek 32	Sun	Home	D	2.0	2.0	Liverpool	2.0	...	16	
49	2022-04-20	20:00	Matchweek 30	Wed	Home	W	3.0	0.0	Brighton	1.2	...	20	
50	2022-04-23	15:00	Matchweek 34	Sat	Home	W	5.0	1.0	Watford	3.0	...	15	

68 rows × 38 columns

Now we have shown this for Manchester City, we will create a **lambda function** and the **.apply( )** method to iterate this function over *all* of the teams in our data set. Very cool!

```
In [58]: matches_rolling = matches.groupby("team").apply(lambda x: rolling_averages(
```

```
In [59]: matches_rolling
```

Out[59]:

team		date	time	round	day	venue	result	gf	ga	opponent	xg	.
Arsenal	6	2020-10-04	14:00	Matchweek 4	Sun	Home	W	2.0	1.0	Sheffield Utd	0.4	.
	7	2020-10-17	17:30	Matchweek 5	Sat	Away	L	0.0	1.0	Manchester City	0.9	.
	9	2020-10-25	19:15	Matchweek 6	Sun	Home	L	0.0	1.0	Leicester City	0.9	.
	11	2020-11-01	16:30	Matchweek 7	Sun	Away	W	1.0	0.0	Manchester Utd	1.1	.
	13	2020-11-08	19:15	Matchweek 8	Sun	Home	L	0.0	3.0	Aston Villa	1.5	.
...	...	...	...	...	...	...	...	...	...	...	...	.
Wolverhampton Wanderers	32	2022-03-13	14:00	Matchweek 29	Sun	Away	W	1.0	0.0	Everton	0.8	.
	33	2022-03-18	20:00	Matchweek 30	Fri	Home	L	2.0	3.0	Leeds United	0.8	.
	34	2022-04-02	15:00	Matchweek 31	Sat	Home	W	2.0	1.0	Aston Villa	1.2	.
	35	2022-04-08	20:00	Matchweek 32	Fri	Away	L	0.0	1.0	Newcastle Utd	0.3	.
	36	2022-04-24	14:00	Matchweek 34	Sun	Away	L	0.0	1.0	Burnley	0.7	.

1317 rows × 38 columns

Notice again that we appear to have generated an additional index level. Let's drop this before proceeding.

```
In [60]: matches_rolling = matches_rolling.droplevel('team')
```

In [61]:
```python
# Display data frame
matches_rolling
```

Out[61]:

	date	time	round	day	venue	result	gf	ga	opponent	xg	...	hour	day_coc
6	2020-10-04	14:00	Matchweek 4	Sun	Home	W	2.0	1.0	Sheffield Utd	0.4	...	14	
7	2020-10-17	17:30	Matchweek 5	Sat	Away	L	0.0	1.0	Manchester City	0.9	...	17	
9	2020-10-25	19:15	Matchweek 6	Sun	Home	L	0.0	1.0	Leicester City	0.9	...	19	
11	2020-11-01	16:30	Matchweek 7	Sun	Away	W	1.0	0.0	Manchester Utd	1.1	...	16	
13	2020-11-08	19:15	Matchweek 8	Sun	Home	L	0.0	3.0	Aston Villa	1.5	...	19	
...	...	...	...	...	...	...	...	...	...	...	...	...	
32	2022-03-13	14:00	Matchweek 29	Sun	Away	W	1.0	0.0	Everton	0.8	...	14	
33	2022-03-18	20:00	Matchweek 30	Fri	Home	L	2.0	3.0	Leeds United	0.8	...	20	
34	2022-04-02	15:00	Matchweek 31	Sat	Home	W	2.0	1.0	Aston Villa	1.2	...	15	
35	2022-04-08	20:00	Matchweek 32	Fri	Away	L	0.0	1.0	Newcastle Utd	0.3	...	20	
36	2022-04-24	14:00	Matchweek 34	Sun	Away	L	0.0	1.0	Burnley	0.7	...	14	

1317 rows × 38 columns

It may be noticed that many of our index values are being repeated, so we will reset out index. This is important as we want unique values for our index.

In [62]:
```python
matches_rolling.index = range(matches_rolling.shape[0])
```

Predictions Function ⚽

In this sub-section we will create a predictions function that we can use to make predictions without having to repeat significant chunks of code for each model we wish to test. This function will:

1. Split data into training- and test-sets
2. Fit the model to training data
3. Make predictions using testing data
4. Combine actuals and predictions together
5. Calculate precision

In [63]:
```
def make_predictions(data, predictors):

    # (1) - Split data
    train = data[data["date"] < '2022-01-01']
    test = data[data["date"] > '2022-01-01']

    # (2) - Fit model
    rf.fit(train[predictors], train["target"])

    # (3) - Make predictions
    preds = rf.predict(test[predictors])

    # (4) - Combine actuals and predictions
    combined = pd.DataFrame(dict(actual=test["target"], predicted=preds), i

    # (5) - Calculate precision
    error = precision_score(test["target"], preds)

    return combined, error
```

We can now call this function and pass in our original predictors *and* the rolling averages we have just generated.

In [64]:
```
combined, precision = make_predictions(matches_rolling, predictors + new_co
```

In [65]:
```
precision
```

Out[65]:
```
0.625
```

At 62.5%, we have clearly improved our precision markedly.

We can also see how the predicted result for each match compared with its actual result by running the *combined* command.

In [66]:
```
combined
```

Out[66]:

	actual	predicted
55	0	0
56	1	0
57	1	0
58	1	1
59	1	1
...	...	...
1312	1	0
1313	0	0
1314	1	0
1315	0	0
1316	0	0

276 rows × 2 columns

However, this does not really give us much information about those matches, such as when the match happened, which teams were playing, etc. We can enrich our combined data frame using the **.merge( )** method to add in match details, as shown below.

In [67]:
```
combined = combined.merge(matches_rolling[["date", "team", "opponent", "resu
```

In [68]:
```
# Display first 10 matches
combined.head(10)
```

Out[68]:

	actual	predicted	date	team	opponent	result
55	0	0	2022-01-23	Arsenal	Burnley	D
56	1	0	2022-02-10	Arsenal	Wolves	W
57	1	0	2022-02-19	Arsenal	Brentford	W
58	1	1	2022-02-24	Arsenal	Wolves	W
59	1	1	2022-03-06	Arsenal	Watford	W
60	1	1	2022-03-13	Arsenal	Leicester City	W
61	0	1	2022-03-16	Arsenal	Liverpool	L
62	1	0	2022-03-19	Arsenal	Aston Villa	W
63	0	0	2022-04-04	Arsenal	Crystal Palace	L
64	0	0	2022-04-09	Arsenal	Brighton	L

Matching Results ⚽

In our analysis, we made predictions for each match **twice**. Once from the perspective of the Home team and one from the perspective of the Away team. **Did our model predict the same overall result for that match for both perspectives?**

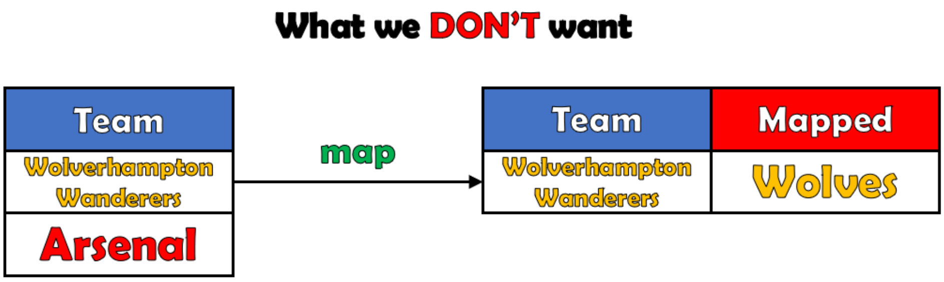
For example, if Arsenal is playing at Home against Everton in a given season, and the prediction is a win for Arsenal, then will the model also predict a 'Loss' for Everton playing Away against Arsenal in the same season?

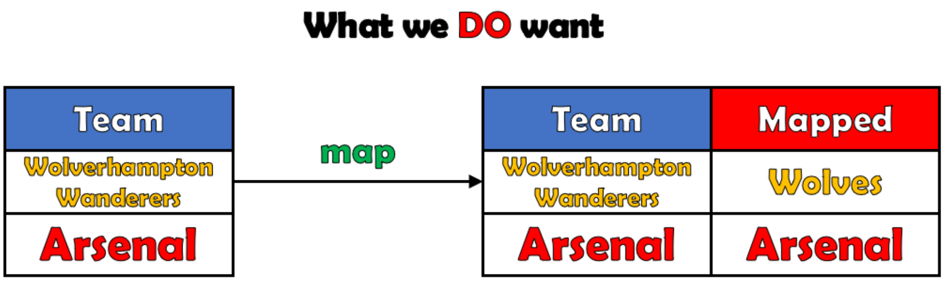
To check this, we can combine our data together. First, we need to ensure that the *team* name and the *opponent* name are the same in our data set, as sometimes they are not. For instance, we have **Wolverhampton Wanderers** in the *team* column and **Wolves** in the *opponent* column for the same team.

To make sure that the names are consistent across both columns, we will create a *dictionary* and use the **.map( )** method with that dictionary. However, we must first create a **child class** that *inherents* from the dictionary class. For a good explanation of class inheritance, click [here (%22https://www.w3schools.com/python/python_inheritance.asp%22)](%22https://www.w3schools.com/python/python_inheritance.asp%22).

We need to do this because the .map( ) method does not handle any **missing keys**. So, if the dictionary being mapped to a column is - for example - missing a team name, the .map( ) method will simply remove that team's observation from the data. For instance, if we create a mapping dictionary that has a key 'Wolverhampton Wanderers' with a value 'Wolves', then passing 'Wolverhampton Wanderers' will return the value 'Wolves'. However, if we pass in 'Arsenal' but there is no key for this team, then the observation will be removed from the data.

Instead, we want the mapping function to simply return the team name that it is passed if no key exists for that team. This is illustrated below.





We begin by creating a child class called **MissingDict**. We pass *dict* into the argument, so it will inherit from the *dict class*. This class will leverage a Python **hook** called *missing*. A hook can be used to tap into a module and react when something happens; in this case, when there is a missing key.

After this, we want to create a mapping dictionary before finally passing this dictionary as an argument to our **MissingDict** class to create an instance of it, *mapping*. When we do this, we will use type two astericks prior to *map_values*. We do this because you cannot directly send a dictionary as a parameter to a function accepting kwargs (key word arguments). The **dictionary must be unpacked** so that the function may make use of its elements. The two astericks accomplish this.

In [69]:
```
class MissingDict(dict):
    __missing__ = lambda self, key: key

map_values = {"Brighton and Hove Albion": "Brighton",
              "Manchester United": "Manchester Utd",
              "Newcastle United": "Newcastle Utd",
              "Tottenham Hotspur": "Tottenham",
              "West Ham United": "West Ham",
              "Wolverhampton Wanderers": "Wolves"}
mapping = MissingDict(**map_values)
```

We can now use this in our **.map( )** method.

In [70]:
```
combined["new_team"] = combined["team"].map(mapping)

#Display data frame
combined
```

Out[70]:

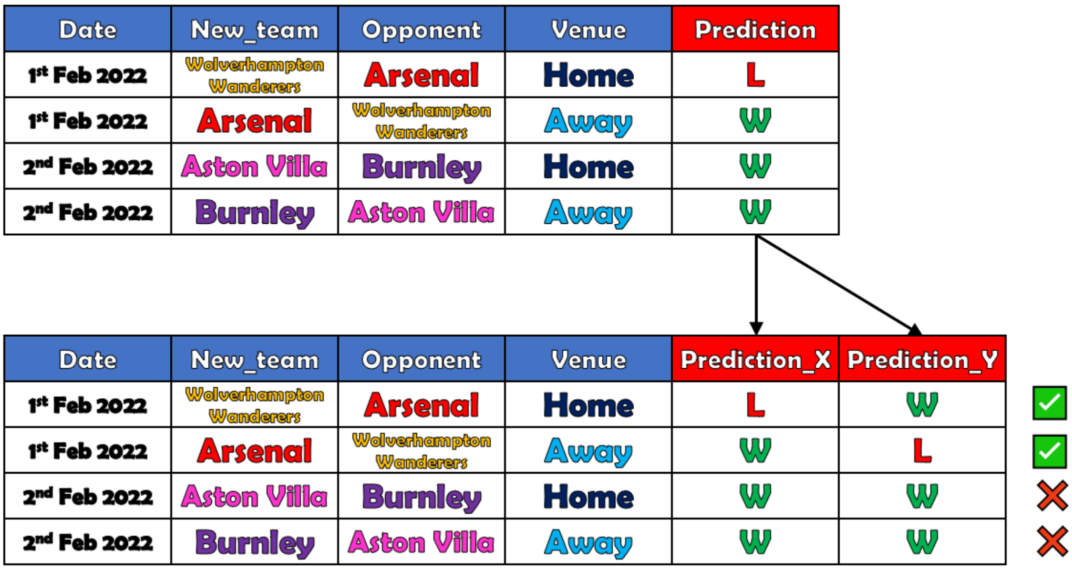
	actual	predicted	date	team	opponent	result	new_team
55	0	0	2022-01-23	Arsenal	Burnley	D	Arsenal
56	1	0	2022-02-10	Arsenal	Wolves	W	Arsenal
57	1	0	2022-02-19	Arsenal	Brentford	W	Arsenal
58	1	1	2022-02-24	Arsenal	Wolves	W	Arsenal
59	1	1	2022-03-06	Arsenal	Watford	W	Arsenal
...	...	...	...	...	...	...	...
1312	1	0	2022-03-13	Wolverhampton Wanderers	Everton	W	Wolves
1313	0	0	2022-03-18	Wolverhampton Wanderers	Leeds United	L	Wolves
1314	1	0	2022-04-02	Wolverhampton Wanderers	Aston Villa	W	Wolves
1315	0	0	2022-04-08	Wolverhampton Wanderers	Newcastle Utd	L	Wolves
1316	0	0	2022-04-24	Wolverhampton Wanderers	Burnley	L	Wolves

276 rows × 7 columns

As you can see above, we now have a **new_team** variable that contains the same teams as in the **team** column, but with the same naming convention as in the **opponent** column.

We can now merge this data frame *with itself*, by merging on the *date* and *new_team* variables for one data frame, but using the *date* and *opponent* in the other data frame. The purpose of this is to create a data frame with **two predicted** values for each observation, one prediction made from the Home team's perspective, and the other from their opponent's perspective.

This process is illustrated below. **prediction_x** is the original match prediction made for that team, whereas **prediction_y** is the prediction from the perspective of the opponent.



Date	New_team	Opponent	Venue	Prediction
1st Feb 2022	Wolverhampton Wanderers	Arsenal	Home	L
1st Feb 2022	Arsenal	Wolverhampton Wanderers	Away	W
2nd Feb 2022	Aston Villa	Burnley	Home	W
2nd Feb 2022	Burnley	Aston Villa	Away	W

Date	New_team	Opponent	Venue	Prediction_X	Prediction_Y
1st Feb 2022	Wolverhampton Wanderers	Arsenal	Home	L	W
1st Feb 2022	Arsenal	Wolverhampton Wanderers	Away	W	L
2nd Feb 2022	Aston Villa	Burnley	Home	W	W
2nd Feb 2022	Burnley	Aston Villa	Away	W	W

In the first row in the top table in our illustration, Wolverhampton Wanderers are playing Arsenal at Home on the 1^{st} February 2022. The predicted result for Wolverhampton Wanderers is a loss (L). The second row is the *same* match but from the perspective of Arsenal, who are playing Wolverhampton Wanderers Away on the 1^{st} February 2022 and predicted to win (W). The third and fourth row relate to a single game between Aston Villa and Burnley.

The second table is post-merge. In the first row, we again have Wolverhampton Wanderers playing Arsenal at Home on the 1^{st} February 2022. The **prediction_x** returns the original predicted match result for this team, a loss (L). **prediction_y** returns the predicted result from Arsenal's perspective, which is a win (W). This makes sense, as if one team loses, the other team must - by definition - win: these two columns should have *different values*.

For the Aston Villa and Burnley match, however, *prediction_x* and *prediction_y* are the same for both teams meaning that our algorithm predicted that *both* teams won the match. This is not possible in reality, so this would count against our model.

In [71]:
```
merged = combined.merge(combined, left_on=["date", "new_team"], right_on=["
```

In [72]:
```
merged
```

Out[72]:

	actual_x	predicted_x	date	team_x	opponent_x	result_x	new_team_x	actual_
0	0	0	2022-01-23	Arsenal	Burnley	D	Arsenal	
1	1	0	2022-02-10	Arsenal	Wolves	W	Arsenal	
2	1	0	2022-02-19	Arsenal	Brentford	W	Arsenal	
3	1	1	2022-02-24	Arsenal	Wolves	W	Arsenal	
4	1	1	2022-03-06	Arsenal	Watford	W	Arsenal	
...	...	...	...	...	...	...	...	.
257	1	0	2022-03-13	Wolverhampton Wanderers	Everton	W	Wolves	
258	0	0	2022-03-18	Wolverhampton Wanderers	Leeds United	L	Wolves	
259	1	0	2022-04-02	Wolverhampton Wanderers	Aston Villa	W	Wolves	
260	0	0	2022-04-08	Wolverhampton Wanderers	Newcastle Utd	L	Wolves	
261	0	0	2022-04-24	Wolverhampton Wanderers	Burnley	L	Wolves	

262 rows × 13 columns

We can now look at just the rows where one team was predicted to win and where the other team was predicted to lose.

In [73]:
```
merged[(merged["predicted_x"] == 1) & (merged["predicted_y"] ==0)]["actual_
```

Out[73]:
```
1    27
0    13
Name: actual_x, dtype: int64
```

This result suggests that our when our algorithm predicted that one team would win, it correctly predicted that the other team would lose 27 out of 40 times, which is 67.5%. This is not too terrible, but not great either.

Conclusion ⚽

In this project, we web-scraped football data for English Premier League matches across two seasons. After cleaning our data, we enriched it by creating new predictors.

We used a Random Forest algorithm courtesy of the *sklearn* library. After training our model, we assessed its performance. Initially, this was not very strong with a precision of only 47%.

We further enriched our data by creating rolling averages for each team's performance in the prior three games. This resulted in a considerable improvement to our model, with a new precision score of 62.5%.

Finally, we checked the extent to which our model made consistent predictions. We found that when our model predicted that a team would win a match, it also predicted that their opponent would lose 67.5% of the time.

Further Development Suggestions ⚽

There is a variety of ways in which this project could be improved upon that will now be briefly discussed.

One way would be to change the values of our Random Forest **hyper-parameters**, for instance, by altering the number of Decision Trees used in our forest. Typically, the more trees used, the greater the predictive power.

It tends to be a good idea to use **cross-validation** on your data in order to select the best model. Also, other classifier models that can handle non-linear data could have been employed.

Finally, we could have **used more variables** as predictors. There were other tabs on each team's page that contained more match stats pertaining to goalkeeping, passing, and possession statistics.