In [ ]:

```
# Preprocessing the Datasets

# Preprocessing the Teams Dataset

import pandas as pd
import json
import os
import numpy as np

with open ("C:\\Users\\nickt\\Desktop\\Masters\\Dissertation\\Data & Coding\\teams.json") as f_t:
    teams_dataset = json.load(f_t)
#We load the Json file

teams_df = pd.DataFrame(teams_dataset)

teams_df

#We convert the Json file into a pandas dataframe

area_df = pd.json_normalize(teams_df['area'])

teams_df = pd.concat([teams_df.drop('area', axis=1), area_df], axis=1)

teams_df = teams_df.rename(columns={'name': 'country',
                        'id': 'country_id',
                        'alpha3code': 'country_alphacode'})

teams_df = teams_df.drop('alpha2code', axis=1)

#We create separate columns for all the values in the 'area' column and remove any other columns we don't need

column_names = teams_df.columns.tolist()

first_country_index = column_names.index('country')

column_names[first_country_index] = 'name'

teams_df.columns = column_names

teams_df

#We change the names of some columns

arsenal_df = teams_df.loc[teams_df['name']=='Arsenal']

arsenal_df

#We filter the dataset to find the football team we are interested in analyzing in this project (Arsenal)

arsenal_df.to_csv('arsenal_teams.csv', index=False)
#We convert the pandas dataframe into a csv file

# Preprocessing the Competitions Dataset

with open ("C:\\Users\\nickt\\Desktop\\Masters\\Dissertation\\Data & Coding\\competitions.json") as f_c:
    competitions_dataset = json.load(f_c)
#We load the Json file

competitions_df = pd.DataFrame(competitions_dataset)

competitions_df

#We convert the Json file into a pandas dataframe

area_df = pd.json_normalize(competitions_df['area'])

competitions_df = pd.concat([competitions_df.drop('area', axis=1), area_df], axis=1)

competitions_df = competitions_df.rename(columns={'name': 'country',
                        'id': 'country_id',
                        'alpha3code': 'country_alphacode'})

competitions_df = competitions_df.drop('alpha2code', axis=1)

#We create separate columns for all the values in the 'area' column and remove any other columns we don't need

column_names = competitions_df.columns.tolist()

first_country_index = column_names.index('country')

column_names[first_country_index] = 'league'

competitions_df.columns = column_names

competitions_df

#We change the names of some columns

eng_comp_df = competitions_df.loc[competitions_df['league']=='English first division']
eng_comp_df

#We filter the dataset to find the football league we are interested in analyzing in this project (English First Division)

eng_comp_df.to_csv('english_competition.csv', index=False)

#We convert the pandas dataframe into a csv file

# Preprocessing the Players and PlayerRank Dataset

with open ("C:\\Users\\nickt\\Desktop\\Masters\\Dissertation\\Data & Coding\\players.json") as f_p:
    players_dataset = json.load(f_p)

#We load the Json file

players_df = pd.DataFrame(players_dataset)

players_df

he Json file into a pandas dataframe
```

```
with open ("C:\\Users\\nickt\\Desktop\\Masters\\Dissertation\\Data & Coding\\playerank.json") as f_pr:
    playerank_dataset = json.load(f_pr)

#We load the playerank json file

playerank_df = pd.DataFrame(playerank_dataset)

playerank_df

#We convert it to a pandas dataframe

role_cluster_df = playerank_df.groupby('playerId')['roleCluster'].first().reset_index()

players_df = pd.merge(players_df, role_cluster_df, left_on='wyId', right_on='playerId', how='left')

players_df = players_df.drop(columns=['playerId'])

#We add the 'roleCluster' column into the players dataset by matching the 'playerId' feature which exists in both datasets with
#a one-to-one merge. This will add more detailed information about the roles and positions of players

role_df = pd.json_normalize(players_df['role'])

players_df = pd.concat([players_df.drop('role', axis=1), role_df], axis=1)

players_df = players_df.rename(columns={'code2': 'position_alphacode',
                        'code3': 'position_3',
                        'name': 'position'})

players_df = players_df.drop(['position_3', 'birthArea', 'middleName', 'shortName'], axis=1)

#We create separate columns for all the values in the 'role' column and remove any other columns we don't need

area_df = pd.json_normalize(players_df['passportArea'])

players_df = pd.concat([players_df.drop('passportArea', axis=1), area_df], axis=1)

players_df = players_df.rename(columns={'name': 'country',
                        'id': 'country_id',
                        'alpha3code': 'country_alphacode'})
players_df = players_df.drop('alpha2code', axis=1)

#We create separate columns for all the values in the 'passportArea' column and remove any other columns we don't need

arsenal_players_df = players_df.loc[
    (players_df['currentTeamId'] == 1609) |
    (players_df['wyId'] == 7856) |
    (players_df['wyId'] == 3361) |
    (players_df['wyId'] == 7864) |
    (players_df['wyId'] == 7868) |
    (players_df['wyId'] == 7873) |
    (players_df['wyId'] == 7879) |
    (players_df['wyId'] == 25662) |
    (players_df['wyId'] == 26010) |
    (players_df['wyId'] == 171283) |
    (players_df['wyId'] == 25662)
]

#We filter out the dataset and keep only records that consist of Arsenal players that have played a significant amount of
#minutes (over 150 minutes played across all games during the season).

arsenal_players_df

ozil_row = arsenal_players_df[arsenal_players_df['firstName'] == 'Mesut'].copy()
ozil_row['lastName'] = 'Ozil'

arsenal_players_df = pd.concat([arsenal_players_df, ozil_row], ignore_index=True)

arsenal_players_df = arsenal_players_df.drop(index=1)

arsenal_players_df = arsenal_players_df.reset_index(drop=True)

#We modify certain player names that are wrong in the initial dataset

ospina_row = arsenal_players_df[arsenal_players_df['firstName'] == 'David'].copy()
ospina_row['lastName'] = 'Ospina'

arsenal_players_df = pd.concat([arsenal_players_df, ospina_row], ignore_index=True)

arsenal_players_df = arsenal_players_df.drop(index=11)

arsenal_players_df = arsenal_players_df.reset_index(drop=True)

#We modify certain player names that are wrong in the initial dataset

cech_row = arsenal_players_df[arsenal_players_df['firstName'] == 'Petr'].copy()
cech_row['lastName'] = 'Cech'

arsenal_players_df = pd.concat([arsenal_players_df, cech_row], ignore_index=True)

arsenal_players_df = arsenal_players_df.drop(index=12)

arsenal_players_df = arsenal_players_df.reset_index(drop=True)

#We modify certain player names that are wrong in the initial dataset

sanchez_row = arsenal_players_df[arsenal_players_df['firstName'] == 'Alexis Alejandro'].copy()
sanchez_row['lastName'] = 'Sanchez'

arsenal_players_df = pd.concat([arsenal_players_df, sanchez_row], ignore_index=True)

arsenal_players_df = arsenal_players_df.drop(index=12)

arsenal_players_df = arsenal_players_df.reset_index(drop=True)

#We modify certain player names that are wrong in the initial dataset

kolasinac_row = arsenal_players_df[arsenal_players_df['firstName'] == 'Sead'].copy()
kolasinac_row['lastName'] = 'Kolasinac'

ers_df = pd.concat([arsenal_players_df, kolasinac_row], ignore_index=True)
```

```
arsenal_players_df = arsenal_players_df.drop(index=19)

arsenal_players_df = arsenal_players_df.reset_index(drop=True)

#We modify certain player names that are wrong in the initial dataset

bellerin_row = arsenal_players_df[arsenal_players_df['wyId'] == 167145].copy()
bellerin_row['lastName'] = 'Bellerin'
bellerin_row['firstName'] = 'Hector'

arsenal_players_df = pd.concat([arsenal_players_df, bellerin_row], ignore_index=True)

arsenal_players_df = arsenal_players_df.drop(index=20)

arsenal_players_df = arsenal_players_df.reset_index(drop=True)

#We modify certain player names that are wrong in the initial dataset

arsenal_players_df

arsenal_players_df['birthDate'] = pd.to_datetime(arsenal_players_df['birthDate'])

arsenal_players_df.iloc[25,10] = 'GK'
arsenal_players_df.iloc[25,9] = 'goalkeeper'

arsenal_players_df['fullName'] = arsenal_players_df['firstName'].str[0] + '.' + ' ' + arsenal_players_df['lastName']

#We modify certain cells that are wrong, convert the date of birth column into a datetime column and also create a new column
#with the first letter of the first name and the last name of players combined

arsenal_players_df

arsenal_players_df.to_csv('arsenal_players.csv', index=False)

#We convert the pandas dataframe into a csv file

# Preprocessing the Matches in England Dataset

with open ("C:\\Users\\nickt\\Desktop\\Masters\\Dissertation\\Data & Coding\\matches_England.json") as f_m:
    matches_eng_dataset = json.load(f_m)

#We load the Json file

matches_eng_df = pd.DataFrame(matches_eng_dataset)

matches_eng_df

#We convert the Json file into a pandas dataframe

matches_eng_df = matches_eng_df.drop(['teamsData', 'date', 'referees'], axis=1)

matches_eng_df['dateutc'] = pd.to_datetime(matches_eng_df['dateutc'])
matches_eng_df['month_int'] = matches_eng_df['dateutc'].dt.month

#We remove certain columns we don't need and also extract the month integer of each game in a separate column which will be used
#for later comparisons

matches_eng_df

month_dict = {
        1: "January", 2: "February", 3: "March", 4: "April",
        5: "May", 6: "June", 7: "July", 8: "August",
        9: "September", 10: "October", 11: "November", 12: "December"
    }

def month_conversion(x):
    return month_dict[x]

matches_eng_df['month'] = list(map(month_conversion, matches_eng_df['month_int']))

#We create a new column that transforms the month integer to the actual month name

matches_eng_df.rename(columns={'winner': 'winner_team_id'}, inplace=True)

#We rename certain columns

def split_label(x):
    teams, scores = x.split(', ')
    home_team, away_team = teams.split(' - ')
    home_score, away_score = scores.split(' - ')

    return home_team, away_team, home_score, away_score

matches_eng_df[['HomeTeam', 'AwayTeam', 'HomeScore', 'AwayScore']] = list(map(split_label, matches_eng_df['label']))

matches_eng_df = matches_eng_df.drop('label', axis=1)

#We use the split_label function to split the values in the 'label' column and create separate columns that have the values of
#the home and away team names, and also the home and away score

matches_eng_df

matches_arsenal_df = matches_eng_df.loc[(matches_eng_df['HomeTeam']=='Arsenal') | (matches_eng_df['AwayTeam']=='Arsenal')]

matches_arsenal_df = matches_arsenal_df.sort_values('gameweek')

matches_arsenal_df

#We filter out the dataset to keep only games that Arsenal were either the team playing at home or away

matches_arsenal_df.to_csv('arsenal_matches.csv', index=False)

#We convert the pandas dataframe into a csv file

# Preprocessing the Events in England Dataset

with open ("C:\\Users\\nickt\\Desktop\\Masters\\Dissertation\\Data & Coding\\events_England.json") as f_e:
    events_dataset = json.load(f_e)
```

 *Json file*

```
events_df = pd.DataFrame(events_dataset)

events_df

#We convert the Json file into a pandas dataframe

df_mapping = pd.read_csv('eventid2name.csv')
df_tags = pd.read_csv('tags2name.csv')

#We read the 2 extra csv files from figshare
#df_mapping --> Contains the description and name of each event (we already have this information in the initial events dataset)
#so we don't need to add anything from there
#df_tags --> Contains the label and description of each tag.

df_tags

import ast

tag_to_description = pd.Series(df_tags.Description.values, index=df_tags.Tag).to_dict()

def replace_tags_with_descriptions(tags_list):
    return [tag_to_description.get(tag['id'], 'Unknown') for tag in tags_list]

if isinstance(events_df['tags'].iloc[0], str):
    events_df['tags'] = events_df['tags'].apply(ast.literal_eval)

events_df['tags'] = events_df['tags'].apply(replace_tags_with_descriptions)

events_df['tags_description'] = events_df['tags'].apply(lambda tags: ', '.join(tags))

events_df = events_df.drop(columns=['tags'])

#We use the replace_tags_with_descriptions function to create a new column that contains the tag description of each tag Id and
#add it to our initial events dataset

lst_arsenal_matches = matches_arsenal_df['wyId'].unique().tolist()

events_arsenal_df = events_df.loc[events_df['matchId'].isin(lst_arsenal_matches)]

#We filter out the dataset and keep only events from Arsenal matches

events_arsenal_df

events_arsenal_df['start_x'] = events_arsenal_df['positions'].apply(lambda pos: pos[0]['x'])
events_arsenal_df['start_x'] = events_arsenal_df['start_x'] * 1.2

events_arsenal_df['start_y'] = events_arsenal_df['positions'].apply(lambda pos: pos[0]['y'])
events_arsenal_df['start_y'] = events_arsenal_df['start_y'] * 0.8

events_arsenal_df['end_x'] = events_arsenal_df['positions'].apply(lambda pos: pos[1]['x'] if len(pos)> 1 else pos[0]['x'])
events_arsenal_df['end_x'] = events_arsenal_df['end_x'] * 1.2

events_arsenal_df['end_y'] = events_arsenal_df['positions'].apply(lambda pos: pos[1]['y'] if len(pos)>1 else pos[0]['y'])
events_arsenal_df['end_y'] = events_arsenal_df['end_y'] * 0.8
#Since the X and Y coordinates are both in the range of (0-100) we need to normalize the values to match the actual pitch
#coordinates (matching: from mplsoccer.pitch import Pitch). So we map the X coordinates to be in the range of 0-120 and the Y
#coordinates to be in the range of 0-80.

events_arsenal_df['event_length'] = np.sqrt((events_arsenal_df['end_x'] - events_arsenal_df['start_x'])**2 + (events_arsenal_df['end_y'] - events_arsenal_df['s
#The event length is equal to the Euclidean Distance between 2 data points.
events_arsenal_df['event_angle'] = np.degrees(np.arctan2(events_arsenal_df['end_x']-events_arsenal_df['start_x'], events_arsenal_df['end_y']-events_arsenal_df[
#The equation can be defined as the direction or angle of each pass relative to the vertical axis (X coordinate in our case).

#We extract new information about the events
#We split the start and end X,y coordinates of the events in separate columns and extract the length of the event (euclidean
#distance) and the angle of the event

#Reference link for calculating the Euclidean Distance of an event (event length): https://www.shiksha.com/online-courses/articles/how-to-compute-euclidean-dis
#Reference link for calculating the angle of an event: https://stackoverflow.com/questions/42258637/how-to-know-the-angle-between-two-vectors

events_arsenal_df['vertical_change'] = (events_arsenal_df['end_x'] - events_arsenal_df['start_x'])
events_arsenal_df['horizontal_change'] = (events_arsenal_df['end_y'] - events_arsenal_df['start_y'])

#We extract new information about the events
#We calculate the vertical and horizontal change of events and add the values in separate columns

#The value of the x coordinate indicates the event's nearness (in percentage) to the opponent's goal
#The value of the y coordinates indicates the event's nearness (in percentage) to the right side of the field;

#Horizontal change would be the difference in the y coordinate
#Vertical change would be the difference in the x coordinate

pass_lst = ['Simple pass', 'High pass', 'Head pass', 'Hand pass', 'Smart pass', 'Launch',
            'Touch', 'Cross', 'Free Kick', 'Throw in']

pass_type_mapping = [1,2,3,4,5,6,7,8,9,10]

def encode_pass_type(x):
    if x in pass_lst:
        return int(pass_type_mapping[pass_lst.index(x)])
    else:
        return np.nan

events_arsenal_df['pass_type_code'] = events_arsenal_df['subEventName'].apply(encode_pass_type)
events_arsenal_df['pass_type_code'] = events_arsenal_df['pass_type_code'].astype(pd.Int32Dtype())

#We encode the types of passes to have a numerical representation of them

def calculate_event_sec_from_start(row):
    if row['matchPeriod'] == '1H':
        return row['eventSec']
    elif row['matchPeriod'] == '2H':
        return row['eventSec'] + 2700

events_arsenal_df['eventSecFromStart'] = events_arsenal_df.apply(calculate_event_sec_from_start, axis=1)

#The 'eventSec' column shows the timing of the event (in seconds since the beginning of the current half of the match). Although
#this feature could be useful, we should also create a new column that shows the timing of the event since the beginning of the
#match and not the current half.
```

```
def time_segment(time):
    if time<=1800:
        return '0-30'
    elif time<=3600:
        return '30-60'
    else:
        return '60-90+'

events_arsenal_df['time_segment (min)'] = events_arsenal_df['eventSecFromStart'].apply(time_segment)
events_arsenal_df = events_arsenal_df.drop('positions', axis=1)

#We create a column that will be used to segment the timing of the events into categories. This will be further useful to
#identify similarities or dissimiliarites of the passing sequences in different time segments of games

events_arsenal_df.to_csv('events_arsenal.csv', index=False)

#We convert the pandas dataframe into a csv file
```

In [ ]:

```
# Merging the Preprocessed Datasets and Applying Extra Preprocessing

import pandas as pd
import numpy as np

df_teams = pd.read_csv('arsenal_teams.csv')
df_competitions = pd.read_csv('english_competition.csv')
df_players = pd.read_csv('arsenal_players.csv')
df_matches = pd.read_csv('arsenal_matches.csv')
df_events = pd.read_csv('events_arsenal.csv')

#We read all of the csv files

df_comps_matches = pd.merge(df_matches, df_competitions, left_on='competitionId', right_on='wyId', how='left')

df_comps_matches = df_comps_matches.drop('wyId_y', axis=1)
df_comps_matches = df_comps_matches.rename(columns={'wyId_x': 'match_Id'})

df_comps_matches

#We merge the datasets that contain information about the matches and the competitions we are interested in. To merge these
#datasets, we used the 'competitionId' which is common in both datasets ('wyId' in the competitions dataset). The relationship
#of the datasets is defined as one-to-many (one competition has many matches)

df_players_teams = pd.merge(df_players, df_teams, left_on='currentTeamId', right_on='wyId', how='left')

df_players_teams = df_players_teams.drop(['country_y', 'country_id_y', 'country_alphacode_y', 'name', 'currentTeamId'], axis=1)

df_players_teams = df_players_teams.rename(columns={'wyId_y': 'teamId', 'country_alphacode_x': 'country_alphacode',
                                                    'country_id_x': 'country_id', 'country_x': 'country_name',
                                                    'wyId_x': 'player_Id', 'officialName': 'officialTeamName'})

df_players_teams

#We then merge the datasets that contain information about the players and the teams we are interested in. To merge these
#datasets, we used the 'currentTeamId' which is common in both datasets ('wyId' in the teams dataset). The relationship
#of the datasets is defined as one-to-many (one team has many players)

#We also drop any extra duplicate columns that were part of both datasets and renamed certain other columns so they have a more
#detailed description

df_comps_matches_events = pd.merge(df_events, df_comps_matches, left_on='matchId', right_on='match_Id', how='left')

df_comps_matches_events = df_comps_matches_events.drop('match_Id', axis=1)

df_comps_matches_events

#We then merge the events dataset with the merged competitions and matches dataset. To merge these datasets, we used the
#'matchId' which is common in both datasets ('match_Id' in the merged competitions-matches dataset). The relationship of the
#datasets is defined as one-to-many (one match has many events)

final_df= pd.merge(df_comps_matches_events, df_players_teams, left_on='playerId', right_on='player_Id', how='left')

final_df = final_df.drop(['player_Id', 'teamId_y', 'type_y', 'officialTeamName', 'city'],  axis=1)

final_df = final_df.rename(columns={'country_id_x': 'team_country_id', 'country_alphacode_x': 'team_country_alphacode',
                                    'country_id_y': 'player_country_id', 'country_alphacode_y': 'player_country_alphacode',
                                    'teamId_x': 'teamId', 'type_x': 'type'})

#Finally we merge the two merged datasets that have combined information about the competitions-matches-events and players-teams
#To merge these datasets, we used the 'playerId' which is common in both datasets ('player_Id' in the merged
#players-teams dataset). The relationship of the datasets is defined as one-to-many (one player has many events).

#We also drop any extra duplicate columns that were part of both datasets and renamed certain other columns so they have a more
#detailed description

#Now we have managed to merge all of the datasets together to a single dataset without losing any useful information.

final_league_positions = {
    'Manchester City': 1, 'Manchester United': 2, 'Tottenham Hotspur': 3,
    'Liverpool': 4, 'Chelsea': 5, 'Arsenal': 6, 'Burnley': 7,
    'Everton': 8, 'Leicester City': 9, 'Newcastle United': 10,
    'Crystal Palace': 11, 'AFC Bournemouth': 12, 'West Ham United': 13,
    'Watford': 14, 'Brighton & Hove Albion': 15, 'Huddersfield Town': 16,
    'Southampton': 17, 'Swansea City': 18, 'Stoke City': 19, 'West Bromwich Albion': 20
}

def categorize_opponent_strength(row):
    if row['HomeTeam'] != 'Arsenal':
        if final_league_positions[row['HomeTeam']] <= 10:
            return 'Top 10 Team'
        else:
            return 'Bottom 10 Team'
    elif row['AwayTeam'] != 'Arsenal':
        if final_league_positions[row['AwayTeam']] <= 10:
            return 'Top 10 Team'
        else:
            return 'Bottom 10 Team'
    else:
        return 'Unknown'

#We use the categorize_opponent_strength function we created to segment the strength of the opponents of each game based on
```

```
#their final position that year (English First Division/Premier League 2017-18 Season)

final_df['OpponentStrength'] = final_df.apply(categorize_opponent_strength, axis=1)

#We add a new column that segments the strength of the opponent to be either a Top 10 Team, or Bottom 10 Team.

final_df['current_score_difference'] = 0

current_score_of_game = 0
current_match_id = None

for index, row in final_df.iterrows():
    if row['matchId'] != current_match_id:
        current_score_of_game = 0
        current_match_id = row['matchId'] #If a new match detected, reset the score to 0.

    tags_description = row['tags_description']
    if pd.notna(tags_description) and 'Goal, Position' in tags_description:
        if row['teamId'] == 1609:
            current_score_of_game -= 1 #Looking at the dataset, when the eventName = 'save attempt' is about Arsenal's
            #goalkeeper, the opposite team scores. When the tags_description='Goal, Position...' and the event belongs to
            #Arsenal, it means that the opposition team has scored
        else:
            current_score_of_game += 1
    final_df.at[index, 'current_score_difference'] = current_score_of_game

#This function is used to track down the current score of the game while the events are occuring.

final_df.to_csv('final_dataset.csv', index=False)

#We convert the final pandas dataframe to a csv file
```

In [ ]:

```
import pandas as pd

df = pd.read_csv('final_dataset.csv')

df

#We import the merged dataset

last_rows = df.groupby('matchId').tail(1)

last_rows['result'] = last_rows['current_score_difference'].apply(lambda x: 2 if x >= 1 else (1 if x == 0 else 0))

match_results_dict = last_rows.set_index('matchId')['result'].to_dict()

df['result'] = df['matchId'].map(match_results_dict)

#Based on the current score difference of the last event of each game (unique matchId) we extract the final result of the game.

#The game results are encoded as following:
#Result Win --> 2
#Result Draw --> 1
#Result Lost --> 0

#These are the conditions under which a new possession begins combined with the conditions specified in the is_possession_lost
#function.
#The conditions are:
#1. If the team of the event changes.
#2. If the match period changes (1st-->2nd half, 2nd--> 1st half indicating a different match).
#3. If the time segment is different to further analyze the passing strategies and events every 30 minutes of the game
conditions_to_split_possessions = (
    (df['teamId'] != df['teamId'].shift(1)) | (df['matchPeriod'] != df['matchPeriod'].shift(1)) |
    (df['time_segment (min)'] != df['time_segment (min)'].shift(1))
)

#We assign a unique possession Id for each new possession.
df['possession_id'] = conditions_to_split_possessions.cumsum()

#We apply the function to update the 'outcome' column for each possession
outcomes = df.groupby('possession_id').apply(lambda group: group.iloc[-1]['eventName'])
df['outcome'] = df['possession_id'].map(outcomes)

def aggregate_possession(group):
    #We want to extract contextual information about the possessions such as the month of the game, or whether the team won that
    #game or if the game was home or away. This information will be useful when further analysing and comparing the passing
    #episodes.
    #We want to extract contextual information about the possessions such as the month of the game, or whether the team won that
    #game or if the game was home or away. This information will be useful when further analysing and comparing the passing
    #episodes.
    month = group['month_int'].iloc[0]
    home_game = group.apply(lambda row: 1 if row['HomeTeam'] == 'Arsenal' else 0, axis=1).iloc[0]


    return {
        'matchId': group['matchId'].iloc[0],
        'teamId': group['teamId'].iloc[0],
        'matchPeriod': group['matchPeriod'].iloc[0],
        'start_time': group['eventSecFromStart'].min(),
        'end_time': group['eventSecFromStart'].max(),
        'duration': group['eventSecFromStart'].max() - group['eventSecFromStart'].min(),
        'num_events': group.shape[0],
        'events': list(group['eventName']),
        'sub_events': list(group['subEventName']),
        'tags_description': list(group['tags_description']),
        'eventsId': list(group['eventId']),
        'sub_eventsId': list(group['subEventId']),
        'players': list(group['playerId']),
        'player_names': list(group['fullName']),
        'position': list(group['position']),
        'opponent_strength': group['OpponentStrength'].iloc[0],
        'start_coords_x': list(group['start_x']),
        'start_coords_y': list(group['start_y']),
        'end_coords_x': list(group['end_x']),
        'end_coords_y': list(group['end_y']),
        'event_length': list(group['event_length']),
        'event_angle': list(group['event_angle']),
        'vertical_change': list(group['vertical_change']),
        'horizontal_change': list(group['horizontal_change']),
        'time_segment (min)': list(group['time_segment (min)'])[0],
```

```
        'outcome': group['outcome'].iloc[0],
        'month': month,
        'home_game': home_game,
        'result': group['result'].iloc[0],
        'current_score_difference': group['current_score_difference'].iloc[0]
    } #These are the features that we will include in the possessions dataset

#We group by possession ID and apply the aggregation function.
possession_seq = df.groupby('possession_id').apply(aggregate_possession)

#We convert the series of dictionaries into a DataFrame.
possessions_df = pd.DataFrame(list(possession_seq), index=possession_seq.index)
possessions_df.reset_index(drop=True, inplace=True)
possessions_df.index.name = 'possession_id'


#We create a function to round each value in the event_length and event_angle to just 2 decimal points
def round_decimals(item):
    if isinstance(item, list):
        return [round(num, 3) for num in item]
    elif isinstance(item, float):
        return round(item, 3)
    return item

#We apply the round_decimals function to the length and angle of the events
possessions_df['event_length'] = possessions_df['event_length'].apply(round_decimals)
possessions_df['event_angle'] = possessions_df['event_angle'].apply(round_decimals)

possessions_df['vertical_amplitude'] = possessions_df.apply(lambda row: max(row['start_coords_x'] + row['end_coords_x']) - min(row['start_coords_x'] + row['enc
possessions_df['horizontal_amplitude'] = possessions_df.apply(lambda row: max(row['start_coords_y'] + row['end_coords_y']) - min(row['start_coords_y'] + row['e

#Horizontal Amplitude:  The difference between the maximum and minimum y-coordinates (in our case) during a possession. This
#indicates how much the ball moved horizontally across the pitch, providing insights into the team's lateral play and usage of
#the field width.

#Vertical Amplitude: The difference between the maximum and minimum x-coordinates (in our case) during a possession. This
#measures the forward and backward movement of the ball on the pitch, giving an idea of how much ground was covered in the
#attack or defensive movements.

possessions_df['starting_position_x'] = possessions_df['start_coords_x'].apply(lambda x: x[0] if x else None)
possessions_df['starting_position_y'] = possessions_df['start_coords_y'].apply(lambda x: x[0] if x else None)
possessions_df['ending_position_x'] = possessions_df['end_coords_x'].apply(lambda x: x[-1] if x else None)
possessions_df['ending_position_y'] = possessions_df['end_coords_y'].apply(lambda x: x[-1] if x else None)

possessions_df.to_csv('possessions.csv', index=False)

#We covert the pandas dataframe to a csv file

arsenal_possessions_df = possessions_df.loc[possessions_df['teamId']==1609]

arsenal_possessions_df.reset_index(drop=True, inplace=True)

arsenal_possessions_df.index.name = 'possession_id'

arsenal_possessions_df.to_csv('arsenal_possessions.csv', index=False)

arsenal_possessions_df

#For our project we want to analyze the passing sequences (episodes) of Arsenal during various games so we filter out the
#dataset and keep only episodes were Arsenal has ball possession

arsenal_possessions_with_passes_df = arsenal_possessions_df.loc[arsenal_possessions_df['events'].apply(lambda x: 'Pass' in x)]

arsenal_possessions_with_passes_df.reset_index(drop=True, inplace=True)

arsenal_possessions_with_passes_df.index.name = 'possession_id'

arsenal_possessions_with_passes_df = arsenal_possessions_with_passes_df.loc[arsenal_possessions_with_passes_df['num_events']>1]

arsenal_possessions_with_passes_df.to_csv('arsenal_possessions_with_passes.csv', index=False)

arsenal_possessions_with_passes_df

#Since we aim to analyze the passing sequences, any possession that doesn't involve passing is irrelevant. So we filter out the
#possessions and keep meaningful possession with passing sequences

# Extracting Summary Statistics of Episodes

episode_stats = {'Total Possessions': len(arsenal_possessions_with_passes_df),
                 'AVG Duration (sec)': arsenal_possessions_with_passes_df['duration'].mean(),
                 'MIN Duration (sec)': arsenal_possessions_with_passes_df['duration'].min(),
                 'MAX Duration (sec)': arsenal_possessions_with_passes_df['duration'].max(),
                 'MEDIAN Duration (sec)': arsenal_possessions_with_passes_df['duration'].median(),
                 'AVG Number of Events': arsenal_possessions_with_passes_df['num_events'].mean(),
                 'MIN Number of Events': arsenal_possessions_with_passes_df['num_events'].min(),
                 'MAX Number of Events': arsenal_possessions_with_passes_df['num_events'].max(),
                 'MEDIAN Number of Events': arsenal_possessions_with_passes_df['num_events'].median(),
                 'AVG Duration (sec)': arsenal_possessions_with_passes_df['duration'].mean(),
                 'AVG Vertical Amplitude': arsenal_possessions_with_passes_df['vertical_amplitude'].mean(),
                 'MEDIAN Vertical Amplitude': arsenal_possessions_with_passes_df['vertical_amplitude'].median(),
                 'AVG Horizontal Amplitude': arsenal_possessions_with_passes_df['horizontal_amplitude'].mean(),
                 'MEDIAN Horizontal Amplitude': arsenal_possessions_with_passes_df['horizontal_amplitude'].median(),
                 'AVG Starting Position X': arsenal_possessions_with_passes_df['starting_position_x'].mean(),
                 'AVG Ending Position X': arsenal_possessions_with_passes_df['ending_position_x'].mean(),
                 'AVG Starting Position Y': arsenal_possessions_with_passes_df['starting_position_y'].mean(),
                 'AVG Ending Position Y': arsenal_possessions_with_passes_df['ending_position_y'].mean()}

episode_stats_df = pd.DataFrame(list(episode_stats.items()), columns=['Statistic', 'Value'])

episode_stats_df = episode_stats_df.round(2)

episode_stats_df

#These are the summary statistics of the episodes extracted.

episode_stats_df.to_csv('Episode Statistics.csv', index=False)
```

In [ ]:
```
# Creating Passing Networks

### Data Preprocessing
```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

possessions_df = pd.read_csv('arsenal_possessions.csv')

passes_df = pd.read_csv('arsenal_possessions_with_passes.csv')

#We import the necessary datasets

events_df = pd.read_csv('final_dataset.csv')

events_df = events_df.loc[(events_df['eventName']=='Pass') & (events_df['teamId']==1609)]

events_df

#We are also going to use this dataset to calculate the coordinates of the passes that players complete. We will use these
#coordinates for the network graphs to also get insights on player positions.

def grouping_current_score(x):
    if x>=1:
        return 'Winning'
    elif x==0:
        return 'Tied'
    else:
        return 'Losing'

events_df['current_score_grouped'] = events_df['current_score_difference'].apply(grouping_current_score)

events_df

events_df_coordinates_base = events_df.groupby('fullName').agg({'start_x': 'mean', 'start_y': 'mean'}).rename(columns={'start_x': 'avg_start_x_base',
                                                                                                                      'start_y': 'avg_start_y_base'})

events_df_coordinates_home_games = events_df.loc[(events_df['HomeTeam']=='Arsenal') & (events_df['eventName']=='Pass')].groupby('fullName').agg({'start_x': 'me
                                                                                                                      'start_y': 'mean'}).rename(columns={'start_x': 'avg_start_x_home',
                                                                                                                      'start_y': 'avg_start_y_home'})

events_df_coordinates_away_games = events_df.loc[(events_df['AwayTeam']=='Arsenal') & (events_df['eventName']=='Pass')].groupby('fullName').agg({'start_x': 'me
                                                                                                                      'start_y': 'mean'}).rename(columns={'start_x': 'avg_start_x_away',
                                                                                                                      'start_y': 'avg_start_y_away'})

events_df_coordinates_top10_games = events_df.loc[(events_df['OpponentStrength']=='Top 10 Team') & (events_df['eventName']=='Pass')].groupby('fullName').agg({'
                                                                                                                      'start_y': 'mean'}).rename(columns={'start_x': 'avg_start_x_top10',
                                                                                                                      'start_y': 'avg_start_y_top10'})

events_df_coordinates_bottom10_games = events_df.loc[(events_df['OpponentStrength']=='Bottom 10 Team') & (events_df['eventName']=='Pass')].groupby('fullName').
                                                                                                                      'start_y': 'mean'}).rename(columns={'start_x': 'avg_start_x_bottom10
                                                                                                                      'start_y': 'avg_start_y_bottom10'

events_df_coordinates_0_30 = events_df.loc[(events_df['time_segment (min)']=='0-30') & (events_df['eventName']=='Pass')].groupby('fullName').agg({'start_x': 'm
                                                                                                                      'start_y': 'mean'}).rename(columns={'start_x': 'avg_start_x_0_30',
                                                                                                                      'start_y': 'avg_start_y_0_30'})

events_df_coordinates_30_60 = events_df.loc[(events_df['time_segment (min)']=='30-60') & (events_df['eventName']=='Pass')].groupby('fullName').agg({'start_x':
                                                                                                                      'start_y': 'mean'}).rename(columns={'start_x': 'avg_start_x_30_60',
                                                                                                                      'start_y': 'avg_start_y_30_60'})

events_df_coordinates_60_90 = events_df.loc[(events_df['time_segment (min)']=='60-90+') & (events_df['eventName']=='Pass')].groupby('fullName').agg({'start_x':
                                                                                                                      'start_y': 'mean'}).rename(columns={'start_x': 'avg_start_x_60_90',
                                                                                                                      'start_y': 'avg_start_y_60_90'})

events_df_coordinates_winning = events_df.loc[(events_df['current_score_grouped']=='Winning') & (events_df['eventName']=='Pass')].groupby('fullName').agg({'sta
                                                                                                                      'start_y': 'mean'}).rename(columns={'start_x': 'avg_start_x_winning'
                                                                                                                      'start_y': 'avg_start_y_winning'}

events_df_coordinates_tied = events_df.loc[(events_df['current_score_grouped']=='Tied') & (events_df['eventName']=='Pass')].groupby('fullName').agg({'start_x':
                                                                                                                      'start_y': 'mean'}).rename(columns={'start_x': 'avg_start_x_tied',
                                                                                                                      'start_y': 'avg_start_y_tied'})

events_df_coordinates_losing = events_df.loc[(events_df['current_score_grouped']=='Losing') & (events_df['eventName']=='Pass')].groupby('fullName').agg({'start
                                                                                                                      'start_y': 'mean'}).rename(columns={'start_x': 'avg_start_x_losing',
                                                                                                                      'start_y': 'avg_start_y_losing'})
#We calculate the average coordinates of passes for each Arsenal player in different game scenarios. The contextual information
#that will be intergated into the passing networks will help us provide a more in-depth analysis.

df_pass_coordinates = pd.concat([events_df_coordinates_base, events_df_coordinates_home_games, events_df_coordinates_away_games,
                         events_df_coordinates_top10_games,events_df_coordinates_bottom10_games,
                         events_df_coordinates_0_30,events_df_coordinates_30_60,
                         events_df_coordinates_60_90, events_df_coordinates_winning, events_df_coordinates_tied,
                              events_df_coordinates_losing], axis=1) #We merge all of the above datasets into a single dataset that
#has the average coordinates of passes for each player in all different game contexts (home vs away, opponent strength etc.)


df_pass_coordinates = df_pass_coordinates.round(0)

#We slightly modify the coordinates to actual pitch coordinates. The coordinates of passes in the events dataset
#are scaled between 0-100 for both x and y but the coordinates of the pitch from the mplsoccer.Pitch library (which we will
#load later) are scaled between 0-120 for x and 0-80 for y. So we need to scale the x coordinates from 0-120 and the y coordinates
#from 0-80. The distances of players are still consistent.

df_pass_coordinates

#We use the start coordinates of each pass because that is the position from where a player attempted the pass and not received
#it. We want to visualize where players make the passes and not where they receive passes from other players.

pass_coords = df_pass_coordinates.copy()

df_pass_coordinates.loc[df_pass_coordinates.index.isin(["A. Maitland-Niles", "R. Holding", "H. Bellerin", "I. Monreal Eraso",
                                                        "K. Mavropanos", "L. Koscielny", "S. Mustafi", "S. Kolasinac",
                                                       "P. Mertesacker", "C. Chambers"]), ['avg_start_x_base', 'avg_start_x_home', 'avg_start_x_away', 'avg_sta
,'avg_start_x_winning', 'avg_start_x_tied', 'avg_start_x_losing']] *= 0.9

df_pass_coordinates.loc[df_pass_coordinates.index.isin(["A. Lacazette", "P. Aubameyang", "D. Tackie Mensah Welbeck",
                                                       "A. Sanchez", "O. Giroud", "T. Walcott"]), ['avg_start_x_base', 'avg_start_x_home', 'avg_start_x_away',

midfielders = ["A. Iwobi", "A. Ramsey", "G. Xhaka", "H. Mkhitaryan", "M. Elsayed Elneny", "M. Ozil", "J. Wilshere",
               "F. Coquelin", "A. Oxlade-Chamberlain"]
inates.loc[df_pass_coordinates.index.isin(midfielders), ['avg_start_x_base', 'avg_start_x_home', 'avg_start_x_away', 'avg_start_x_top10', 'avg_sta
```

```
#We slightly modify a few player positions. We want to spread out the players on the pitch so the graphs are easier to
#interpret.

df_pass_coordinates.loc[df_pass_coordinates.index == "S. Mustafi", 'avg_start_y_base'] = 42
df_pass_coordinates.loc[df_pass_coordinates.index == "G. Xhaka", 'avg_start_y_base'] = 34
df_pass_coordinates.loc[df_pass_coordinates.index == "J. Wilshere", 'avg_start_y_base'] = 36
df_pass_coordinates.loc[df_pass_coordinates.index == "M. Ozil", 'avg_start_y_base'] = 33
df_pass_coordinates.loc[df_pass_coordinates.index == "D. Tackie Mensah Welbeck", 'avg_start_y_base'] = 30

df_pass_coordinates.loc[df_pass_coordinates.index == "M. Ozil", 'avg_start_y_home'] = 28
df_pass_coordinates.loc[df_pass_coordinates.index == "A. Iwobi", 'avg_start_y_home'] = 36
df_pass_coordinates.loc[df_pass_coordinates.index == "S. Mustafi", 'avg_start_y_home'] = 43
df_pass_coordinates.loc[df_pass_coordinates.index == "S. Mustafi", 'avg_start_y_away'] = 41
df_pass_coordinates.loc[df_pass_coordinates.index == "G. Xhaka", 'avg_start_x_home'] = 52
df_pass_coordinates.loc[df_pass_coordinates.index == "M. Ozil", 'avg_start_y_away'] = 32
df_pass_coordinates.loc[df_pass_coordinates.index == "G. Xhaka", 'avg_start_x_away'] = 51
df_pass_coordinates.loc[df_pass_coordinates.index == "A. Iwobi", 'avg_start_y_away'] = 53
df_pass_coordinates.loc[df_pass_coordinates.index == "O. Giroud", 'avg_start_y_away'] = 36

df_pass_coordinates.loc[df_pass_coordinates.index == "M. Ozil", 'avg_start_y_top10'] = 30
df_pass_coordinates.loc[df_pass_coordinates.index == "A. Ramsey", 'avg_start_y_top10'] = 37
df_pass_coordinates.loc[df_pass_coordinates.index == "M. Elsayed Elneny", 'avg_start_x_top10'] = 53
df_pass_coordinates.loc[df_pass_coordinates.index == "M. Ozil", 'avg_start_y_bottom10'] = 31
df_pass_coordinates.loc[df_pass_coordinates.index == "A. Ramsey", 'avg_start_y_bottom10'] = 37
df_pass_coordinates.loc[df_pass_coordinates.index == "G. Xhaka", 'avg_start_x_bottom10'] = 60
df_pass_coordinates.loc[df_pass_coordinates.index == "P. Aubameyang", 'avg_start_y_top10'] = 32
df_pass_coordinates.loc[df_pass_coordinates.index == "D. Ospina", 'avg_start_y_bottom10'] = 42
df_pass_coordinates.loc[df_pass_coordinates.index == "S. Mustafi", 'avg_start_y_top10'] = 45
df_pass_coordinates.loc[df_pass_coordinates.index == "S. Mustafi", 'avg_start_y_bottom10'] = 42

df_pass_coordinates.loc[df_pass_coordinates.index == "A. Ramsey", 'avg_start_y_0_30'] = 35
df_pass_coordinates.loc[df_pass_coordinates.index == "M. Ozil", 'avg_start_y_0_30'] = 33
df_pass_coordinates.loc[df_pass_coordinates.index == "A. Ramsey", 'avg_start_y_30_60'] = 34
df_pass_coordinates.loc[df_pass_coordinates.index == "M. Ozil", 'avg_start_y_30_60'] = 33
df_pass_coordinates.loc[df_pass_coordinates.index == "A. Ramsey", 'avg_start_y_60_90'] = 35
df_pass_coordinates.loc[df_pass_coordinates.index == "M. Ozil", 'avg_start_y_60_90'] = 31
df_pass_coordinates.loc[df_pass_coordinates.index == "M. Elsayed Elneny", 'avg_start_x_0_30'] = 57
df_pass_coordinates.loc[df_pass_coordinates.index == "G. Xhaka", 'avg_start_x_0_30'] = 52
df_pass_coordinates.loc[df_pass_coordinates.index == "M. Elsayed Elneny", 'avg_start_x_30_60'] = 54
df_pass_coordinates.loc[df_pass_coordinates.index == "G. Xhaka", 'avg_start_x_30_60'] = 56
df_pass_coordinates.loc[df_pass_coordinates.index == "M. Elsayed Elneny", 'avg_start_x_60_90'] = 54
df_pass_coordinates.loc[df_pass_coordinates.index == "G. Xhaka", 'avg_start_x_60_90'] = 56
df_pass_coordinates.loc[df_pass_coordinates.index == "A. Ramsey", 'avg_start_x_60_90'] = 65
df_pass_coordinates.loc[df_pass_coordinates.index == "S. Mustafi", 'avg_start_y_60_90'] = 43

df_pass_coordinates.loc[df_pass_coordinates.index == "P. Aubameyang", 'avg_start_y_losing'] = 30
df_pass_coordinates.loc[df_pass_coordinates.index == "D. Tackie Mensah Welbeck", 'avg_start_y_losing'] = 33

#We slightly modify a few player positions. We want to spread out the players on the pitch so the graphs are easier to
#interpret.

all_games_avg_positions = dict(zip(df_pass_coordinates.index, zip(df_pass_coordinates['avg_start_x_base'],
                                                    df_pass_coordinates['avg_start_y_base'])))

home_avg_positions = dict(zip(df_pass_coordinates.index, zip(df_pass_coordinates['avg_start_x_home'],
                                                    df_pass_coordinates['avg_start_y_home'])))

away_avg_positions = dict(zip(df_pass_coordinates.index, zip(df_pass_coordinates['avg_start_x_away'],
                                                    df_pass_coordinates['avg_start_y_away'])))

top10_avg_positions = dict(zip(df_pass_coordinates.index, zip(df_pass_coordinates['avg_start_x_top10'],
                                                     df_pass_coordinates['avg_start_y_top10'])))

bottom10_avg_positions = dict(zip(df_pass_coordinates.index, zip(df_pass_coordinates['avg_start_x_bottom10'],
                                                        df_pass_coordinates['avg_start_y_bottom10'])))

min_0_30_avg_positions = dict(zip(df_pass_coordinates.index, zip(df_pass_coordinates['avg_start_x_0_30'],
                                                        df_pass_coordinates['avg_start_y_0_30'])))

min_30_60_avg_positions = dict(zip(df_pass_coordinates.index, zip(df_pass_coordinates['avg_start_x_30_60'],
                                                         df_pass_coordinates['avg_start_y_30_60'])))

min_60_90_avg_positions = dict(zip(df_pass_coordinates.index, zip(df_pass_coordinates['avg_start_x_60_90'],
                                                         df_pass_coordinates['avg_start_y_60_90'])))

winning_avg_positions = dict(zip(df_pass_coordinates.index, zip(df_pass_coordinates['avg_start_x_winning'],
                                                         df_pass_coordinates['avg_start_y_winning'])))

tied_avg_positions = dict(zip(df_pass_coordinates.index, zip(df_pass_coordinates['avg_start_x_tied'],
                                                         df_pass_coordinates['avg_start_y_tied'])))

losing_avg_positions = dict(zip(df_pass_coordinates.index, zip(df_pass_coordinates['avg_start_x_losing'],
                                                         df_pass_coordinates['avg_start_y_losing'])))


#We create separate dictionaries that contain the player and the average coordinates of passes he completes for the different
#game contexts.

all_games_avg_positions_top_11 = {player: minutes_played for player,
                          minutes_played in all_games_avg_positions.items() if player in ['G. Xhaka', 'H. Bellerin', 'P. Cech',
                                                                 'I. Monreal Eraso','A. Lacazette', 'S. Mustafi',
                                                                 'L. Koscielny','S. Kolasinac', 'M. Ozil',
                                                                 'A. Ramsey', 'A. Iwobi']}
home_avg_positions_top_11 = {player: minutes_played for player,
                          minutes_played in home_avg_positions.items() if player in ['G. Xhaka', 'H. Bellerin', 'P. Cech',
                                                                 'I. Monreal Eraso','A. Lacazette', 'S. Mustafi',
                                                                 'L. Koscielny','S. Kolasinac', 'M. Ozil',
                                                                 'A. Ramsey', 'A. Iwobi']}
away_avg_positions_top_11 = {player: minutes_played for player,
                          minutes_played in away_avg_positions.items() if player in ['G. Xhaka', 'H. Bellerin', 'P. Cech',
                                                                 'I. Monreal Eraso','A. Lacazette', 'S. Mustafi',
                                                                 'L. Koscielny','S. Kolasinac', 'M. Ozil',
                                                                 'A. Ramsey', 'A. Iwobi']}

top10_avg_positions_top_11 = {player: minutes_played for player,
                          minutes_played in top10_avg_positions.items() if player in ['G. Xhaka', 'H. Bellerin', 'P. Cech',
                                                                 'I. Monreal Eraso','A. Lacazette', 'S. Mustafi',
                                                                 'L. Koscielny','S. Kolasinac', 'M. Ozil',
                                                                 'A. Ramsey', 'A. Iwobi']}
```



```
positions_top_11 = {player: minutes_played for player,
```

```
                            minutes_played in bottom10_avg_positions.items() if player in ['G. Xhaka', 'H. Bellerin', 'P. Cech',
                                                                                           'I. Monreal Eraso','A. Lacazette', 'S. Mustafi',
                                                                                           'L. Koscielny','S. Kolasinac', 'M. Ozil',
                                                                                           'A. Ramsey', 'A. Iwobi']}
min_0_30_avg_positions_top_11 = {player: minutes_played for player,
                            minutes_played in min_0_30_avg_positions.items() if player in ['G. Xhaka', 'H. Bellerin', 'P. Cech',
                                                                                           'I. Monreal Eraso','A. Lacazette', 'S. Mustafi',
                                                                                           'L. Koscielny','S. Kolasinac', 'M. Ozil',
                                                                                           'A. Ramsey', 'A. Iwobi']}
min_30_60_avg_positions_top_11 = {player: minutes_played for player,
                            minutes_played in min_30_60_avg_positions.items() if player in ['G. Xhaka', 'H. Bellerin', 'P. Cech',
                                                                                           'I. Monreal Eraso','A. Lacazette', 'S. Mustafi',
                                                                                           'L. Koscielny','S. Kolasinac', 'M. Ozil',
                                                                                           'A. Ramsey', 'A. Iwobi']}
min_60_90_avg_positions_top_11 = {player: minutes_played for player,
                            minutes_played in min_60_90_avg_positions.items() if player in ['G. Xhaka', 'H. Bellerin', 'P. Cech',
                                                                                           'I. Monreal Eraso','A. Lacazette', 'S. Mustafi',
                                                                                           'L. Koscielny','S. Kolasinac', 'M. Ozil',
                                                                                           'A. Ramsey', 'A. Iwobi']}
winning_avg_positions_top_11 = {player: minutes_played for player,
                            minutes_played in winning_avg_positions.items() if player in ['G. Xhaka', 'H. Bellerin', 'P. Cech',
                                                                                           'I. Monreal Eraso','A. Lacazette', 'S. Mustafi',
                                                                                           'L. Koscielny','S. Kolasinac', 'M. Ozil',
                                                                                           'A. Ramsey', 'A. Iwobi']}
tied_avg_positions_top_11 = {player: minutes_played for player,
                            minutes_played in tied_avg_positions.items() if player in ['G. Xhaka', 'H. Bellerin', 'P. Cech',
                                                                                           'I. Monreal Eraso','A. Lacazette', 'S. Mustafi',
                                                                                           'L. Koscielny','S. Kolasinac', 'M. Ozil',
                                                                                           'A. Ramsey', 'A. Iwobi']}
losing_avg_positions_top_11 = {player: minutes_played for player,
                            minutes_played in losing_avg_positions.items() if player in ['G. Xhaka', 'H. Bellerin', 'P. Cech',
                                                                                           'I. Monreal Eraso','A. Lacazette', 'S. Mustafi',
                                                                                           'L. Koscielny','S. Kolasinac', 'M. Ozil',
                                                                                           'A. Ramsey', 'A. Iwobi']}

#We are going to use the passing interactions of the 11 players with the most minutes played between them during the 2017-18
#season and their average coordinated for each context to produce and study the passing network graphs.

import re

def converting_string_to_list(df, column):
    def processing_element(element):
        if isinstance(element, str):
            element = element.strip("[]")
            element = re.sub(r"['\"]", "", element)
            element = [el.strip() for el in element.split(',') if el.strip()]
            return element
        return element

    df[column] = df[column].apply(lambda y: processing_element(y))
    return df

def cleaning_values_correctly(df, column):
    def processing_element(element):
        if isinstance(element, str):
            element = element.strip("[]")
            element = re.sub(r"['\"]", "", element)
            element = [el.strip() for el in element.split(',') if el.strip()]
            return element
        return element

    df[column] = df[column].apply(lambda y: processing_element(y))
    return df

#We noticed that the columns that have lists of values have the lists inside strings. So we want to transform the strings into
#lists with elements

#We apply the functions to both datasets for the columns that have a list of elements in each row.
possessions_df = converting_string_to_list(possessions_df, 'player_names')
possessions_df = cleaning_values_correctly(possessions_df, 'events')
possessions_df = cleaning_values_correctly(possessions_df, 'sub_events')
possessions_df = cleaning_values_correctly(possessions_df, 'tags_description')
possessions_df = cleaning_values_correctly(possessions_df, 'eventsId')
possessions_df = cleaning_values_correctly(possessions_df, 'sub_eventsId')
possessions_df = cleaning_values_correctly(possessions_df, 'position')
possessions_df = cleaning_values_correctly(possessions_df, 'start_coords_x')
possessions_df = cleaning_values_correctly(possessions_df, 'start_coords_y')
possessions_df = cleaning_values_correctly(possessions_df, 'end_coords_x')
possessions_df = cleaning_values_correctly(possessions_df, 'end_coords_y')
possessions_df = cleaning_values_correctly(possessions_df, 'event_length')
possessions_df = cleaning_values_correctly(possessions_df, 'event_angle')
possessions_df = cleaning_values_correctly(possessions_df, 'vertical_change')
possessions_df = cleaning_values_correctly(possessions_df, 'horizontal_change')
possessions_df = cleaning_values_correctly(possessions_df, 'players')


passes_df = cleaning_values_correctly(passes_df, 'player_names')
passes_df = cleaning_values_correctly(passes_df, 'events')
passes_df = cleaning_values_correctly(passes_df, 'sub_events')
passes_df = cleaning_values_correctly(passes_df, 'tags_description')
passes_df = cleaning_values_correctly(passes_df, 'eventsId')
passes_df = cleaning_values_correctly(passes_df, 'sub_eventsId')
passes_df = cleaning_values_correctly(passes_df, 'position')
passes_df = cleaning_values_correctly(passes_df, 'start_coords_x')
passes_df = cleaning_values_correctly(passes_df, 'start_coords_y')
passes_df = cleaning_values_correctly(passes_df, 'end_coords_x')
passes_df = cleaning_values_correctly(passes_df, 'end_coords_y')
passes_df = cleaning_values_correctly(passes_df, 'event_length')
passes_df = cleaning_values_correctly(passes_df, 'event_angle')
passes_df = cleaning_values_correctly(passes_df, 'vertical_change')
passes_df = cleaning_values_correctly(passes_df, 'horizontal_change')
passes_df = cleaning_values_correctly(passes_df, 'players')

#This function replaces the value 'nan' with the actual player Id. Some players do not have names so we replace their names
#with the Ids. When creating the interactions dataset we will need player interactions even between players who we don't know
#their names
def replacing_nan_values_with_player_ids(df):
    def replace_nan_values(player_names, player_ids):
        return [player_name if player_name != 'nan' else str(player_id) for player_name, player_id in zip(player_names, player_ids)]

r_names'] = df.apply(lambda row: replace_nan_values(row['player_names'], row['players']), axis=1)
```

```
    return df

#We apply the function on both datasets
possessions_df = replacing_nan_values_with_player_ids(possessions_df)

passes_df = replacing_nan_values_with_player_ids(passes_df)

possessions_df.to_csv('arsenal_possessions.csv', index=False)
passes_df.to_csv('arsenal_possessions_with_passes.csv', index=False)

#We convert the preprocessed dataframes into csv files.

players_df = pd.read_csv('arsenal_players.csv')
#We load the dataset that has information about each player. We will use this dataset because we want to segment the players
#into goalkeepers, defenders, midfielders and forwards for a more in-depth analysis of the passing networks.

player_roles = {'P. Cech': 'GK', 'D. Ospina':'GK',
                'H. Bellerin': 'RWB',
                'L. Koscielny': 'RCB', 'C. Chambers': 'RCB',
                 'S. Mustafi': 'CB', 'R. Holding': 'CB', 'P. Mertesacker': 'CB',
                'K. Mavropanos': 'LCB', 'I. Monreal Eraso': 'LCB',
                'S. Kolasinac': 'LWB',
                'A. Maitland-Niles': 'LCM', 'G. Xhaka': 'LCM', 'J. Wilshere': 'LCM',
                'M. Elsayed Elneny': 'RCM', 'A. Ramsey': 'RCM', 'F. Coquelin': 'RCM', 'A. Oxlade-Chamberlain': 'RCM',
                'M. Ozil': 'LAM', 'A. Sanchez': 'LAM', 'A. Iwobi': 'RAM', 'H. Mkhitaryan': 'RAM', 'T. Walcott': 'RAM',
                'A. Lacazette': 'ST', 'O. Giroud': 'ST', 'D. Tackie Mensah Welbeck': 'ST',
                 'P. Aubameyang': 'ST'}

#We have assigned player roles for each player based on their actual position but also based on their average coordinates

def map_player_roles(player_names):
    return [player_roles.get(player, 'Unknown') for player in player_names]

#We apply the function to the 'player_names' column to extract the player roles and add them to a new column
passes_df['player_roles'] = passes_df['player_names'].apply(map_player_roles)

import matplotlib.pyplot as plt
import networkx as nx #This library will be used to plot the passing network graphs
import pandas as pd
from mplsoccer.pitch import Pitch #We want to show the networks on a pitch to also have details about player positions when
#making passes (that is why we caluclated the average coordinates above).
import matplotlib.patches as mpatches #We use this library to draw shapes and patches

player_positions = dict(zip(players_df['fullName'], players_df['position']))
#We create a dictionary that maps player names into their respective positions.

pass_interactions_between_players = []

for index, row in passes_df.iterrows():
    events = row['events']
    names_of_players = row['player_names']
    roles_of_players = row['player_roles']

    for index in range(len(events) - 1):
        if events[index] == 'Pass':
            pass_interactions_between_players.append({'source': names_of_players[index], 'target': names_of_players[index+1],
                                                      'source_role': roles_of_players[index],
                                                      'target_role': roles_of_players[index+1], 'home_game': row['home_game'],
                                                      'opponent_strength': row['opponent_strength'],
                                                      'time_segment': row['time_segment (min)'],
                                                      'score_difference': row['current_score_difference']})
            #We find all of the pass interactions between players (who passes to whom) for every game context (home vs away,
            #opponent strength, time segment of game). The 'source' is the player that made the pass and 'target' is the player
            #that received the pass. We filter out the interactions to pass interactions between 2 players (the pass has been
            #completed).

pass_interactions_df = pd.DataFrame(pass_interactions_between_players)
#We store all of the player interactions into a pandas dataframe.

pass_interactions_df['source_position'] = pass_interactions_df['source'].map(player_positions)
pass_interactions_df['target_position'] = pass_interactions_df['target'].map(player_positions)
#We also add a column of the position of the source and target player (goalkeeper, defender, midfielder, forward)

pass_interactions_df['source_position'] = pass_interactions_df['source'].map(player_positions)
pass_interactions_df['target_position'] = pass_interactions_df['target'].map(player_positions)

pass_interactions_df = pass_interactions_df[(pass_interactions_df['source'] != pass_interactions_df['target']) &
                                            (~pass_interactions_df['source'].str.isdigit()) &
                                             (~pass_interactions_df['target'].str.isdigit())]
#We filter out the pass interactions dataset. We only want to keep rows where the source and target of the pass are different
#players (indicating one player passes to another) and we also don't want to keep interactions where we don't know the name of
#the player that either passed or received the pass.

def grouping_current_score(x):
    if x>=1:
        return 'Winning'
    elif x==0:
        return 'Tied'
    else:
        return 'Losing'

pass_interactions_df['current_score_grouped'] = pass_interactions_df['score_difference'].apply(grouping_current_score)

pass_interactions_df

### Studying the Roles of Individuals and the Connections Between Them

minutes_played = dict({'G. Xhaka': 3260, 'H. Bellerin': 3051, 'M. Ozil': 2163, 'I. Monreal Eraso': 2243, 'L. Koscielny': 2225,
                       'S. Mustafi': 2273, 'S. Kolasinac': 2147, 'A. Ramsey': 1846, 'A. Iwobi': 1830, 'A. Sanchez': 1503,
                         'J. Wilshere': 1193, 'M. Elsayed Elneny': 867, 'C. Chambers': 866, 'D. Ospina': 381, 'O. Giroud': 388,
                         'F. Coquelin': 169, 'P. Mertesacker': 366, 'A. Oxlade-Chamberlain': 241, 'T. Walcott': 155,
                       'P. Cech': 3039, 'A. Lacazette': 2202, 'R. Holding': 822, 'A. Maitland-Niles': 914, 'H. Mkhitaryan': 794,
                       'D. Tackie Mensah Welbeck': 1194, 'P. Aubameyang': 1057, 'K. Mavropanos': 195})

#These are the total minutes played by player during the 2017-18 season. This information was extracted from FBREF.
#FBREF link: https://fbref.com/en/squads/18bb7c10/2017-2018/Arsenal-Stats

#We are going to use these to normalize the weights of the interactions

def normalize_interactions(row):
    / (minutes_played[row['source']] + minutes_played[row['target']])) * 1000
```

```
#Counting the number of interactions is not very accurate in our case because the total playing time differs from player to
#player. For example the interactions of players with H. Bellerin will be more than the interactions with S. Kolasinac
#because H. Bellerin has played far more games. Also, although one player might have a lower number of passing contributions
#than another, he has a higher number of passing contributions per game. This means that altough he has played less games, when
#he plays he is more involved in the team's passing stratgey and his overall contribution in passing is higher.

#Our goal is to show the strength of the interaction between players and how involved each player is at the team's passing
#strategy based on the number of minutes each player plays. So in our case we normalize the weights of the interactions:
#Interaction Weight = (1/ (minutes played of player 1 + minutes played of player 2)) * 1000

#In this way we show the strength of the passing interactions based on the number of minutes that players had during that season.

pass_interactions_df['normalized_weight_interaction'] = pass_interactions_df.apply(normalize_interactions, axis=1)
pass_interactions_df['normalized_weight_interaction'] = pass_interactions_df['normalized_weight_interaction'].round(3)

pass_interactions_df.to_csv('Pass Interactions.csv', index=True)

pass_interactions_df

top_11_players = ['P. Cech', 'S. Mustafi', 'L. Koscielny', 'H. Bellerin',
                  'I. Monreal Eraso', 'S. Kolasinac', 'G. Xhaka', 'A. Ramsey', 'M. Ozil', 'A. Iwobi', 'A. Lacazette']

pass_interactions_df_top_11_players = pass_interactions_df.loc[(pass_interactions_df['source'].isin(top_11_players)) & pass_interactions_df['target'].isin(top_

#We are going to use the passing interactions of the 11 players with the most minutes played between them during the 2017-18
#season to produce and study the passing network graphs.

#Arsenal's most used formation during the 2017-18 season was the 3-4-2-1 which they carried on from the previous season since
#it was very effective during the last few games.

#Reference link for Arsenal starting 11: https://www.bdfutbol.com/en/t/t2017-182016.html

#References used for constructing passing networks:

#https://medium.com/@yogakrisanto1129/a-step-by-step-guide-to-using-python-to-create-football-passing-networks-e00e92fecd99
#https://soccermatics.readthedocs.io/en/latest/gallery/lesson1/plot_PassNetworks.html
#https://mplsoccer.readthedocs.io/en/latest/gallery/pitch_plots/plot_pass_network.html
#https://networkx.org/documentation/stable/tutorial.html
#https://www.youtube.com/watch?v=pW7rltisoqo&t=285s

position_colors = {'Goalkeeper': 'red', 'Defender': 'lightblue', 'Midfielder': 'lightgreen', 'Forward': 'Yellow'}
#We map each position to a color for the network graphs.

def calculate_node_degree(G):
    node_degrees = {}
    for node in G.nodes():
        quantity_of_interactions = G.degree(node)  #The total number of interactions (both incoming and outgoing a player is
        #involved in
        unique_interactions_of_nodes = set(G.neighbors(node))
        diversity_of_interactions = len(unique_interactions_of_nodes) #The number of unique players a player interacts with
        strength_of_interactions = sum(G[node][node_neighbor]['weight'] for node_neighbor in unique_interactions_of_nodes)
        #The sum of the weights of interactions with each unique player
        combined_node_degree = quantity_of_interactions * np.log(diversity_of_interactions + 1) * strength_of_interactions
        #Combined Degree=Interaction Quantity×log(Interaction Diversity+1)×Strength of Interactions
        #This equation not only takes into account the quantity of interactions of a player but also the quality. The quality
        #of the player interactions highlights players that have strong connections with different players across the pitch
        #making them more important in the team's passing strategy.
        node_degrees[node] = combined_node_degree
    return node_degrees

def create_passing_network_all_games(pass_interactions_df, node_positions):

    G = nx.DiGraph()

    for _, passing_interaction in pass_interactions_df.iterrows():
        passer = passing_interaction['source']
        pass_receiver = passing_interaction['target']
        pass_interaction_weight = passing_interaction['normalized_weight_interaction']

        if G.has_edge(passer, pass_receiver):
            G[passer][pass_receiver]['weight'] +=pass_interaction_weight
        else:
            G.add_edge(passer, pass_receiver, weight=pass_interaction_weight)

    #If the passing interaction already exists, we increase the weight of the interaction, otherwise we create a new edge
    #connected to the two nodes (pass source and pass target)

    combined_node_degrees = calculate_node_degree(G)

    #We ensure no player has a zero size by adding a small constant
    min_non_zero_degree = min(degree for degree in combined_node_degrees.values() if degree > 0)
    combined_node_degrees = {key: value + min_non_zero_degree * 0.5 for key, value in combined_node_degrees.items()}

    #We apply power law scaling to the node sizes to highlight differences
    maximum_combined_degree = max(combined_node_degrees.values())
    minimum_combined_degree = min(combined_node_degrees.values())
    node_sizes = [2000 * ((node_degree - minimum_combined_degree) / (maximum_combined_degree - minimum_combined_degree + 1e-9)) ** 1.5 + 50 for node_degree in

    #To highlight the key players, we first normalize the adjusted combined degree to a range of [1e-9,1] with a small constant to
    #ensure non-zero values.
    #Normalized Degree= (Adjusted Combined Degree-Min Adjusted Combined Degree)/ (Max Adjusted Combined Degree-Min Adjusted Combined Degree+€)

    #The next step is to apply power law scaling to the degrees to exaggerate more the differences:
    #Scaled Degree=(Normalized Degree)^1.5

    #To caluclate the final Node Sizes, we multiply the scaled degree by a scaling factor (2000) and add a constant (50) to
    #ensure a minimum size:
    #Node Size = 2000 × (Scaled Degree) + 50

    #We extract the colors of the nodes based on the position of players. This will make it easier for us to see how players in
    #different positions interact with each other.

    node_colors = {}

    for _, passing_interaction in pass_interactions_df.iterrows():
        node_colors[passing_interaction['source']] = position_colors.get(passing_interaction['source_position'], 'gray')
        node_colors[passing_interaction['target']] = position_colors.get(passing_interaction['target_position'], 'gray')

    list_of_node_colors = [node_colors[node] for node in G.nodes()]
    #We add colors to the nodes
```

```
    pos = node_positions
    passing_network_edges = G.edges(data=True)
    passing_network_weights = [passing_network_edge[2]['weight'] for passing_network_edge in passing_network_edges]
    #These are the nodes and edges of our network

    pitch = Pitch(pitch_type='statsbomb', pitch_color='grass', line_color='white') #We are going to plot the network graph on
    #a football pitch
    fig, ax = pitch.draw(figsize=(12,8))

    nx.draw(G, pos, ax=ax, with_labels=True, node_size=node_sizes, node_color=list_of_node_colors, font_size=12, font_weight='bold',
           edge_color=passing_network_weights, width=[net_width*0.1 for net_width in passing_network_weights], edge_cmap=plt.cm.Blues,
           connectionstyle='arc3,rad=0.2')
    #We construct the network graph

    legend_patches = [mpatches.Patch(color=color, label=position) for position, color in position_colors.items()]
    plt.legend(handles=legend_patches, loc='upper right', fontsize=13)

    title = 'Player Passing Network - All Games'
    plt.title(title, fontsize=16)
    plt.show()

#In this function we create the passing network that shows the strength of the passing interactions between different player
#roles for all games.

create_passing_network_all_games(pass_interactions_df_top_11_players,  all_games_avg_positions_top_11)

position_colors = {'Goalkeeper': 'red', 'Defender': 'lightblue', 'Midfielder': 'lightgreen', 'Forward': 'Yellow'}
#We map each position to a color for the network graphs.

def calculate_node_degree(G):
    node_degrees = {}
    for node in G.nodes():
        quantity_of_interactions = G.degree(node)  #The total number of interactions (both incoming and outgoing a player is
        #involved in
        unique_interactions_of_nodes = set(G.neighbors(node))
        diversity_of_interactions = len(unique_interactions_of_nodes) #The number of unique players a player interacts with
        strength_of_interactions = sum(G[node][node_neighbor]['weight'] for node_neighbor in unique_interactions_of_nodes)
        #The sum of the weights of interactions with each unique player
        combined_node_degree = quantity_of_interactions * np.log(diversity_of_interactions + 1) * strength_of_interactions
        #Combined Degree=Interaction Quantity×log(Interaction Diversity+1)×Strength of Interactions
        #This equation not only takes into account the quantity of interactions of a player but also the quality. The quality
        #of the player interactions highlights players that have strong connections with different players across the pitch
        #making them more important in the team's passing strategy.
        node_degrees[node] = combined_node_degree
    return node_degrees

def create_passing_network(pass_interactions_df, context_col, context_val, node_positions):
    pass_contexts_interactions = pass_interactions_df[pass_interactions_df[context_col]==context_val]
    #We filter out the interactions based on the game context (home vs away, opponent strength, time segment)
    G = nx.DiGraph()

    for _, passing_interaction in pass_contexts_interactions.iterrows():
        passer = passing_interaction['source']
        pass_receiver = passing_interaction['target']
        pass_interaction_weight = passing_interaction['normalized_weight_interaction']

        if G.has_edge(passer, pass_receiver):
            G[passer][pass_receiver]['weight'] +=pass_interaction_weight
        else:
            G.add_edge(passer, pass_receiver, weight=pass_interaction_weight)

    #If the passing interaction already exists, we increase the weight of the interaction, otherwise we create a new edge
    #connected to the two nodes (pass source and pass target)

    combined_node_degrees = calculate_node_degree(G)

    #We ensure no player has a zero size by adding a small constant
    min_non_zero_degree = min(degree for degree in combined_node_degrees.values() if degree > 0)
    combined_node_degrees = {key: value + min_non_zero_degree * 0.5 for key, value in combined_node_degrees.items()}

    #We apply power law scaling to the node sizes to highlight differences
    maximum_combined_degree = max(combined_node_degrees.values())
    minimum_combined_degree = min(combined_node_degrees.values())
    node_sizes = [2000 * ((node_degree - minimum_combined_degree) / (maximum_combined_degree - minimum_combined_degree + 1e-9)) ** 1.5 + 50 for node_degree in

    #To highlight the key players, we first normalize the adjusted combined degree to a range of [1e-9,1] with a small constant to
    #ensure non-zero values.
    #Normalized Degree= (Adjusted Combined Degree-Min Adjusted Combined Degree)/ (Max Adjusted Combined Degree-Min Adjusted Combined Degree+€)

    #The next step is to apply power law scaling to the degrees to exaggerate more the differences:
    #Scaled Degree=(Normalized Degree)^1.5

    #To caluclate the final Node Sizes, we multiply the scaled degree by a scaling factor (2000) and add a constant (50) to
    #ensure a minimum size:
    #Node Size = 2000 × (Scaled Degree) + 50

    #We extract the colors of the nodes based on the position of players. This will make it easier for us to see how players in
    #different positions interact with each other.

    node_colors = {}

    for _, passing_interaction in pass_contexts_interactions.iterrows():
        node_colors[passing_interaction['source']] = position_colors.get(passing_interaction['source_position'], 'gray')
        node_colors[passing_interaction['target']] = position_colors.get(passing_interaction['target_position'], 'gray')

    list_of_node_colors = [node_colors[node] for node in G.nodes()]
    #We add colors to the nodes

    pos = node_positions
    passing_network_edges = G.edges(data=True)
    passing_network_weights = [passing_network_edge[2]['weight'] for passing_network_edge in passing_network_edges]
    #These are the nodes and edges of our network

    pitch = Pitch(pitch_type='statsbomb', pitch_color='grass', line_color='white') #We are going to plot the network graph on
    #a football pitch
    fig, ax = pitch.draw(figsize=(12,8))

    nx.draw(G, pos, ax=ax, with_labels=True, node_size=node_sizes, node_color=list_of_node_colors, font_size=12, font_weight='bold',
           edge_color=passing_network_weights, width=[net_width*0.2 for net_width in passing_network_weights], edge_cmap=plt.cm.Blues,
           connectionstyle='arc3,rad=0.2')
    #We construct the network graph
```

```
    legend_patches = [mpatches.Patch(color=color, label=position) for position, color in position_colors.items()]
    plt.legend(handles=legend_patches, loc='upper right', fontsize=13)

    title = 'Player Passing Network - Home Games' if context_val==1 else 'Player_Passing_Network - Away Games'
    plt.title(title, fontsize=16)
    plt.show()

#In this function we create the passing network that shows the strength of the passing interactions between different player
#roles for different game contexts. In this graph we show the passing interactions in home vs away games.

#We construct the passing network graphs for the different game contexts specified (in this case home vs away games)
create_passing_network(pass_interactions_df_top_11_players, 'home_game', 1, home_avg_positions_top_11)
create_passing_network(pass_interactions_df_top_11_players, 'home_game', 0, away_avg_positions_top_11)

position_colors = {'Goalkeeper': 'red', 'Defender': 'lightblue', 'Midfielder': 'lightgreen', 'Forward': 'Yellow'}
#We map each position to a color for the network graphs.

def calculate_node_degree(G):
    node_degrees = {}
    for node in G.nodes():
        quantity_of_interactions = G.degree(node)  #The total number of interactions (both incoming and outgoing a player is
        #involved in
        unique_interactions_of_nodes = set(G.neighbors(node))
        diversity_of_interactions = len(unique_interactions_of_nodes) #The number of unique players a player interacts with
        strength_of_interactions = sum(G[node][node_neighbor]['weight'] for node_neighbor in unique_interactions_of_nodes)
        #The sum of the weights of interactions with each unique player
        combined_node_degree = quantity_of_interactions * np.log(diversity_of_interactions + 1) * strength_of_interactions
        #Combined Degree=Interaction Quantity×log(Interaction Diversity+1)×Strength of Interactions
        #This equation not only takes into account the quantity of interactions of a player but also the quality. The quality
        #of the player interactions highlights players that have strong connections with different players across the pitch
        #making them more important in the team's passing strategy.
        node_degrees[node] = combined_node_degree
    return node_degrees

def create_passing_network(pass_interactions_df, context_col, context_val, node_positions):
    pass_contexts_interactions = pass_interactions_df[pass_interactions_df[context_col]==context_val]
    #We filter out the interactions based on the game context (home vs away, opponent strength, time segment)
    G = nx.DiGraph()

    for _, passing_interaction in pass_contexts_interactions.iterrows():
        passer = passing_interaction['source']
        pass_receiver = passing_interaction['target']
        pass_interaction_weight = passing_interaction['normalized_weight_interaction']

        if G.has_edge(passer, pass_receiver):
            G[passer][pass_receiver]['weight'] +=pass_interaction_weight
        else:
            G.add_edge(passer, pass_receiver, weight=pass_interaction_weight)

    #If the passing interaction already exists, we increase the weight of the interaction, otherwise we create a new edge
    #connected to the two nodes (pass source and pass target)

    combined_node_degrees = calculate_node_degree(G)

    #We ensure no player has a zero size by adding a small constant
    min_non_zero_degree = min(degree for degree in combined_node_degrees.values() if degree > 0)
    combined_node_degrees = {key: value + min_non_zero_degree * 0.5 for key, value in combined_node_degrees.items()}

    #We apply power law scaling to the node sizes to highlight differences
    maximum_combined_degree = max(combined_node_degrees.values())
    minimum_combined_degree = min(combined_node_degrees.values())
    node_sizes = [2000 * ((node_degree - minimum_combined_degree) / (maximum_combined_degree - minimum_combined_degree + 1e-9)) ** 1.5 + 50 for node_degree in

    #To highlight the key players, we first normalize the adjusted combined degree to a range of [1e-9,1] with a small constant to
    #ensure non-zero values.
    #Normalized Degree= (Adjusted Combined Degree-Min Adjusted Combined Degree)/ (Max Adjusted Combined Degree-Min Adjusted Combined Degree+€)

    #The next step is to apply power law scaling to the degrees to exaggerate more the differences:
    #Scaled Degree=(Normalized Degree)^1.5

    #To caluclate the final Node Sizes, we multiply the scaled degree by a scaling factor (2000) and add a constant (50) to
    #ensure a minimum size:
    #Node Size = 2000 × (Scaled Degree) + 50

    #We extract the colors of the nodes based on the position of players. This will make it easier for us to see how players in
    #different positions interact with each other.

    node_colors = {}

    for _, passing_interaction in pass_contexts_interactions.iterrows():
        node_colors[passing_interaction['source']] = position_colors.get(passing_interaction['source_position'], 'gray')
        node_colors[passing_interaction['target']] = position_colors.get(passing_interaction['target_position'], 'gray')

    list_of_node_colors = [node_colors[node] for node in G.nodes()]
    #We add colors to the nodes

    pos = node_positions
    passing_network_edges = G.edges(data=True)
    passing_network_weights = [passing_network_edge[2]['weight'] for passing_network_edge in passing_network_edges]
    #These are the nodes and edges of our network

    pitch = Pitch(pitch_type='statsbomb', pitch_color='grass', line_color='white') #We are going to plot the network graph on
    #a football pitch
    fig, ax = pitch.draw(figsize=(12,8))

    nx.draw(G, pos, ax=ax, with_labels=True, node_size=node_sizes, node_color=list_of_node_colors, font_size=12, font_weight='bold',
            edge_color=passing_network_weights, width=[net_width*0.2 for net_width in passing_network_weights], edge_cmap=plt.cm.Blues,
            connectionstyle='arc3,rad=0.2')
    #We construct the network graph

    legend_patches = [mpatches.Patch(color=color, label=position) for position, color in position_colors.items()]
    plt.legend(handles=legend_patches, loc='upper right', fontsize=13)

    title = 'Player Passing Network - Top 10 Teams' if context_val=='Top 10 Team' else 'Player_Passing_Network - Bottom 10 Teams'
    plt.title(title, fontsize=16)
    plt.show()

#In this function we create the passing network that shows the strength of the passing interactions between different player
#roles for different game contexts.
```

```
#We construct the passing network graphs for the different game contexts specified (in this case when playing against top 10 vs bottom
#10 teams as in different opponent strength)
create_passing_network(pass_interactions_df_top_11_players, 'opponent_strength', 'Top 10 Team', top10_avg_positions_top_11)
create_passing_network(pass_interactions_df_top_11_players, 'opponent_strength', 'Bottom 10 Team', bottom10_avg_positions_top_11)

position_colors = {'Goalkeeper': 'red', 'Defender': 'lightblue', 'Midfielder': 'lightgreen', 'Forward': 'Yellow'}
#We map each position to a color for the network graphs.

def calculate_node_degree(G):
    node_degrees = {}
    for node in G.nodes():
        quantity_of_interactions = G.degree(node)  #The total number of interactions (both incoming and outgoing a player is
        #involved in
        unique_interactions_of_nodes = set(G.neighbors(node))
        diversity_of_interactions = len(unique_interactions_of_nodes) #The number of unique players a player interacts with
        strength_of_interactions = sum(G[node][node_neighbor]['weight'] for node_neighbor in unique_interactions_of_nodes)
        #The sum of the weights of interactions with each unique player
        combined_node_degree = quantity_of_interactions * np.log(diversity_of_interactions + 1) * strength_of_interactions
        #Combined Degree=Interaction Quantity×log(Interaction Diversity+1)×Strength of Interactions
        #This equation not only takes into account the quantity of interactions of a player but also the quality. The quality
        #of the player interactions highlights players that have strong connections with different players across the pitch
        #making them more important in the team's passing strategy.
        node_degrees[node] = combined_node_degree
    return node_degrees

def create_passing_network(pass_interactions_df, context_col, context_val, node_positions):
    pass_contexts_interactions = pass_interactions_df[pass_interactions_df[context_col]==context_val]
    #We filter out the interactions based on the game context (home vs away, opponent strength, time segment)
    G = nx.DiGraph()

    for _, passing_interaction in pass_contexts_interactions.iterrows():
        passer = passing_interaction['source']
        pass_receiver = passing_interaction['target']
        pass_interaction_weight = passing_interaction['normalized_weight_interaction']

        if G.has_edge(passer, pass_receiver):
            G[passer][pass_receiver]['weight'] +=pass_interaction_weight
        else:
            G.add_edge(passer, pass_receiver, weight=pass_interaction_weight)

    #If the passing interaction already exists, we increase the weight of the interaction, otherwise we create a new edge
    #connected to the two nodes (pass source and pass target)

    combined_node_degrees = calculate_node_degree(G)

    #We ensure no player has a zero size by adding a small constant
    min_non_zero_degree = min(degree for degree in combined_node_degrees.values() if degree > 0)
    combined_node_degrees = {key: value + min_non_zero_degree * 0.5 for key, value in combined_node_degrees.items()}

    #We apply power law scaling to the node sizes to highlight differences
    maximum_combined_degree = max(combined_node_degrees.values())
    minimum_combined_degree = min(combined_node_degrees.values())
    node_sizes = [2000 * ((node_degree - minimum_combined_degree) / (maximum_combined_degree - minimum_combined_degree + 1e-9)) ** 1.5 + 50 for node_degree in

    #To highlight the key players, we first normalize the adjusted combined degree to a range of [1e-9,1] with a small constant to
    #ensure non-zero values.
    #Normalized Degree= (Adjusted Combined Degree-Min Adjusted Combined Degree)/ (Max Adjusted Combined Degree-Min Adjusted Combined Degree+€)

    #The next step is to apply power law scaling to the degrees to exaggerate more the differences:
    #Scaled Degree=(Normalized Degree)^1.5

    #To caluclate the final Node Sizes, we multiply the scaled degree by a scaling factor (2000) and add a constant (50) to
    #ensure a minimum size:
    #Node Size = 2000 × (Scaled Degree) + 50

    #We extract the colors of the nodes based on the position of players. This will make it easier for us to see how players in
    #different positions interact with each other.

    node_colors = {}

    for _, passing_interaction in pass_contexts_interactions.iterrows():
        node_colors[passing_interaction['source']] = position_colors.get(passing_interaction['source_position'], 'gray')
        node_colors[passing_interaction['target']] = position_colors.get(passing_interaction['target_position'], 'gray')

    list_of_node_colors = [node_colors[node] for node in G.nodes()]
    #We add colors to the nodes

    pos = node_positions
    passing_network_edges = G.edges(data=True)
    passing_network_weights = [passing_network_edge[2]['weight'] for passing_network_edge in passing_network_edges]
    #These are the nodes and edges of our network

    pitch = Pitch(pitch_type='statsbomb', pitch_color='grass', line_color='white') #We are going to plot the network graph on
    #a football pitch
    fig, ax = pitch.draw(figsize=(12,8))

    nx.draw(G, pos, ax=ax, with_labels=True, node_size=node_sizes, node_color=list_of_node_colors, font_size=12, font_weight='bold',
            edge_color=passing_network_weights, width=[net_width*0.3 for net_width in passing_network_weights], edge_cmap=plt.cm.Blues,
            connectionstyle='arc3,rad=0.2')
    #We construct the network graph

    legend_patches = [mpatches.Patch(color=color, label=position) for position, color in position_colors.items()]
    plt.legend(handles=legend_patches, loc='upper right', fontsize=13)

    #We set the title based on the game context
    if context_val == '0-30':
        title = 'Player Passing Network - 0-30 mins'
    elif context_val == '30-60':
        title = 'Player Passing Network - 30-60 mins'
    else:
        title = 'Player Passing Network - 60-90+ mins'

    plt.title(title, fontsize=16)
    plt.show()

#In this function we create the passing network that shows the strength of the passing interactions between different player
#roles for different game contexts.

#We construct the passing network graphs for the different game contexts specified (in this case for different time segments
#of the games (first 30 mins vs 30-60 mins vs 60-90+mins))
g_network(pass_interactions_df_top_11_players, 'time_segment', '0-30', min_0_30_avg_positions_top_11)
```

```
create_passing_network(pass_interactions_df_top_11_players, 'time_segment', '30-60', min_30_60_avg_positions_top_11)
create_passing_network(pass_interactions_df_top_11_players, 'time_segment', '60-90+', min_60_90_avg_positions_top_11)

position_colors = {'Goalkeeper': 'red', 'Defender': 'lightblue', 'Midfielder': 'lightgreen', 'Forward': 'Yellow'}
#We map each position to a color for the network graphs.

def calculate_node_degree(G):
    node_degrees = {}
    for node in G.nodes():
        quantity_of_interactions = G.degree(node)  #The total number of interactions (both incoming and outgoing a player is
        #involved in
        unique_interactions_of_nodes = set(G.neighbors(node))
        diversity_of_interactions = len(unique_interactions_of_nodes) #The number of unique players a player interacts with
        strength_of_interactions = sum(G[node][node_neighbor]['weight'] for node_neighbor in unique_interactions_of_nodes)
        #The sum of the weights of interactions with each unique player
        combined_node_degree = quantity_of_interactions * np.log(diversity_of_interactions + 1) * strength_of_interactions
        #Combined Degree=Interaction Quantity×log(Interaction Diversity+1)×Strength of Interactions
        #This equation not only takes into account the quantity of interactions of a player but also the quality. The quality
        #of the player interactions highlights players that have strong connections with different players across the pitch
        #making them more important in the team's passing strategy.
        node_degrees[node] = combined_node_degree
    return node_degrees

def create_passing_network(pass_interactions_df, context_col, context_val, node_positions):
    pass_contexts_interactions = pass_interactions_df[pass_interactions_df[context_col]==context_val]
    #We filter out the interactions based on the game context (home vs away, opponent strength, time segment)
    G = nx.DiGraph()

    for _, passing_interaction in pass_contexts_interactions.iterrows():
        passer = passing_interaction['source']
        pass_receiver = passing_interaction['target']
        pass_interaction_weight = passing_interaction['normalized_weight_interaction']

        if G.has_edge(passer, pass_receiver):
            G[passer][pass_receiver]['weight'] +=pass_interaction_weight
        else:
            G.add_edge(passer, pass_receiver, weight=pass_interaction_weight)

    #If the passing interaction already exists, we increase the weight of the interaction, otherwise we create a new edge
    #connected to the two nodes (pass source and pass target)

    combined_node_degrees = calculate_node_degree(G)

    #We ensure no player has a zero size by adding a small constant
    min_non_zero_degree = min(degree for degree in combined_node_degrees.values() if degree > 0)
    combined_node_degrees = {key: value + min_non_zero_degree * 0.5 for key, value in combined_node_degrees.items()}

    #We apply power law scaling to the node sizes to highlight differences
    maximum_combined_degree = max(combined_node_degrees.values())
    minimum_combined_degree = min(combined_node_degrees.values())
    node_sizes = [2000 * ((node_degree - minimum_combined_degree) / (maximum_combined_degree - minimum_combined_degree + 1e-9)) ** 1.5 + 50 for node_degree in

    #To highlight the key players, we first normalize the adjusted combined degree to a range of [1e-9,1] with a small constant to
    #ensure non-zero values.
    #Normalized Degree= (Adjusted Combined Degree-Min Adjusted Combined Degree)/ (Max Adjusted Combined Degree-Min Adjusted Combined Degree+€)

    #The next step is to apply power law scaling to the degrees to exaggerate more the differences:
    #Scaled Degree=(Normalized Degree)^1.5

    #To caluclate the final Node Sizes, we multiply the scaled degree by a scaling factor (2000) and add a constant (50) to
    #ensure a minimum size:
    #Node Size = 2000 × (Scaled Degree) + 50

    #We extract the colors of the nodes based on the position of players. This will make it easier for us to see how players in
    #different positions interact with each other.

    node_colors = {}

    for _, passing_interaction in pass_contexts_interactions.iterrows():
        node_colors[passing_interaction['source']] = position_colors.get(passing_interaction['source_position'], 'gray')
        node_colors[passing_interaction['target']] = position_colors.get(passing_interaction['target_position'], 'gray')

    list_of_node_colors = [node_colors[node] for node in G.nodes()]
    #We add colors to the nodes

    pos = node_positions
    passing_network_edges = G.edges(data=True)
    passing_network_weights = [passing_network_edge[2]['weight'] for passing_network_edge in passing_network_edges]
    #These are the nodes and edges of our network

    pitch = Pitch(pitch_type='statsbomb', pitch_color='grass', line_color='white') #We are going to plot the network graph on
    #a football pitch
    fig, ax = pitch.draw(figsize=(12,8))

    nx.draw(G, pos, ax=ax, with_labels=True, node_size=node_sizes, node_color=list_of_node_colors, font_size=12, font_weight='bold',
            edge_color=passing_network_weights, width=[net_width*0.3 for net_width in passing_network_weights], edge_cmap=plt.cm.Blues,
            connectionstyle='arc3,rad=0.2')
    #We construct the network graph

    legend_patches = [mpatches.Patch(color=color, label=position) for position, color in position_colors.items()]
    plt.legend(handles=legend_patches, loc='upper right', fontsize=13)

    #We set the title based on the game context
    if context_val == 'Winning':
        title = 'Player Passing Network - When Winning During Games'
    elif context_val == 'Tied':
        title = 'Player Passing Network - When Tied During Games'
    else:
        title = 'Player Passing Network - When Losing During Games'

    plt.title(title, fontsize=16)
    plt.show()

#We construct the passing network graphs for the different game contexts specified (in this case for when the team is winning
#during games, when the team is tied, and when the team is losing).
create_passing_network(pass_interactions_df_top_11_players, 'current_score_grouped', 'Winning', winning_avg_positions_top_11)
create_passing_network(pass_interactions_df_top_11_players, 'current_score_grouped', 'Tied', tied_avg_positions_top_11)
create_passing_network(pass_interactions_df_top_11_players, 'current_score_grouped', 'Losing', losing_avg_positions_top_11)

pass_interactions_df
```

```
pass_interactions_df.to_csv('Pass Interactions.csv', index=False)

#This is the dataframe with all the pass intercations between players for different game contexts.
#'source' --> The player who made the pass
#'target' --> The player who received the pass

### Finding the Contribution of Players in the Team's Passing Strategy- Percentage of Pass Interactions Players Are Involved in For Different Game Contexts in

all_pass_interactions = pd.concat([pass_interactions_df['source'].rename('fullName'),
                       pass_interactions_df['target'].rename('fullName')]).value_counts().reset_index()

all_pass_interactions.columns = ['fullName', 'Pass Interaction % - In All Games']


home_pass_interactions = pd.concat([pass_interactions_df[pass_interactions_df['home_game']==1]['source'].rename('fullName'),
                                    pass_interactions_df[pass_interactions_df['home_game']==1]['target'].rename('fullName')]).value_counts().reset_index()
home_pass_interactions.columns = ['fullName', 'Pass Interaction % - Home Games']


away_pass_interactions = pd.concat([pass_interactions_df[pass_interactions_df['home_game']==0]['source'].rename('fullName'),
                                    pass_interactions_df[pass_interactions_df['home_game']==0]['target'].rename('fullName')]).value_counts().reset_index()
away_pass_interactions.columns = ['fullName', 'Pass Interaction % - Away Games']


top10_pass_interactions = pd.concat([pass_interactions_df[pass_interactions_df['opponent_strength']=='Top 10 Team']['source'].rename('fullName'),
                                    pass_interactions_df[pass_interactions_df['opponent_strength']=='Top 10 Team']['target'].rename('fullName')]).value_counts()
top10_pass_interactions.columns = ['fullName', 'Pass Interaction % - Against Top 10 Teams']


bottom10_pass_interactions = pd.concat([pass_interactions_df[pass_interactions_df['opponent_strength']=='Bottom 10 Team']['source'].rename('fullName'),
                                    pass_interactions_df[pass_interactions_df['opponent_strength']=='Bottom 10 Team']['target'].rename('fullName')]).value_count
bottom10_pass_interactions.columns = ['fullName', 'Pass Interaction % - Against Bottom 10 Teams']


min_0_30_pass_interactions = pd.concat([pass_interactions_df[pass_interactions_df['time_segment']=='0-30']['source'].rename('fullName'),
                                    pass_interactions_df[pass_interactions_df['time_segment']=='0-30']['target'].rename('fullName')]).value_counts().reset_index
min_0_30_pass_interactions.columns = ['fullName', 'Pass Interaction % - During 0-30 Mins of Games']


min_30_60_pass_interactions = pd.concat([pass_interactions_df[pass_interactions_df['time_segment']=='30-60']['source'].rename('fullName'),
                                    pass_interactions_df[pass_interactions_df['time_segment']=='30-60']['target'].rename('fullName')]).value_counts().reset_inde
min_30_60_pass_interactions.columns = ['fullName', 'Pass Interaction % - During 30-60 Mins of Games']


min_60_90_pass_interactions = pd.concat([pass_interactions_df[pass_interactions_df['time_segment']=='60-90+']['source'].rename('fullName'),
                                    pass_interactions_df[pass_interactions_df['time_segment']=='60-90+']['target'].rename('fullName')]).value_counts().reset_ind
min_60_90_pass_interactions.columns = ['fullName', 'Pass Interaction % - During 60-90+ Mins of Games']


winning_pass_interactions = pd.concat([pass_interactions_df[pass_interactions_df['current_score_grouped']=='Winning']['source'].rename('fullName'),
                                    pass_interactions_df[pass_interactions_df['current_score_grouped']=='winning']['target'].rename('fullName')]).value_counts()
winning_pass_interactions.columns = ['fullName', 'Pass Interaction % - When Winning During Games']


tied_pass_interactions = pd.concat([pass_interactions_df[pass_interactions_df['current_score_grouped']=='Tied']['source'].rename('fullName'),
                                    pass_interactions_df[pass_interactions_df['current_score_grouped']=='Tied']['target'].rename('fullName')]).value_counts().re
tied_pass_interactions.columns = ['fullName', 'Pass Interaction % - When Tied During Games']


losing_pass_interactions = pd.concat([pass_interactions_df[pass_interactions_df['current_score_grouped']=='Losing']['source'].rename('fullName'),
                                    pass_interactions_df[pass_interactions_df['current_score_grouped']=='Losing']['target'].rename('fullName')]).value_counts()
losing_pass_interactions.columns = ['fullName', 'Pass Interaction % - When Losing During Games']



combined_df = pd.merge(all_pass_interactions, home_pass_interactions, on='fullName', how='outer')
combined_df = pd.merge(combined_df, away_pass_interactions, on='fullName', how='outer')
combined_df = pd.merge(combined_df, top10_pass_interactions, on='fullName', how='outer')
combined_df = pd.merge(combined_df, bottom10_pass_interactions, on='fullName', how='outer')
combined_df = pd.merge(combined_df, min_0_30_pass_interactions, on='fullName', how='outer')
combined_df = pd.merge(combined_df, min_30_60_pass_interactions, on='fullName', how='outer')
combined_df = pd.merge(combined_df, min_60_90_pass_interactions, on='fullName', how='outer')
combined_df = pd.merge(combined_df, winning_pass_interactions, on='fullName', how='outer')
combined_df = pd.merge(combined_df, tied_pass_interactions, on='fullName', how='outer')
combined_df = pd.merge(combined_df, losing_pass_interactions, on='fullName', how='outer')

columns = combined_df.columns[1:].tolist()

for column in columns:
    combined_df[column] = (combined_df[column]/combined_df[column].sum()) * 100


combined_df

#We calculate the total pass interactions that players have been involved in for each game context in separate dataframes.
#Then we join the dataframes together and transform the count of passes into percentages. This way we can see for example in
#the percentage of pass interactions each player is involved.

#Note. A player is considered to be involved in a pass interaction if he either passes or receives the ball. So to find the
#total interactions we counted the number of rows where a player is either the source or the target of the pass.

total_number_of_players = combined_df.shape[0]
figure, axes = plt.subplots(9,3, figsize=(20,60))
axes = axes.flatten()

def adjust_color_bar_intensity(game_context_value, baseline_all_games_value):
    if baseline_all_games_value == 0:
        color_bar_intensity = 1 #If the player is not involved in any interactions, than just set all bars to be the same
    else:
        percentage_difference = abs(game_context_value - baseline_all_games_value) / baseline_all_games_value
        #Caluclate the difference between the interactions of a player in a certain game context in comparison to the average
        #number of interactions between all contexts
        color_bar_intensity = 0.5 + min(percentage_difference * 2, 0.5) #Adjust the colorbar intensity depending on how far the
        #bar for each context is to the average number of interactions in all contexts

    if game_context_value > baseline_all_games_value:
        return (0,0.6,0, color_bar_intensity) #If the number of passing interactions in a context is more than the average
    #number of interactions between all contexts, plot a green bar and adjust its intensity
    else:
        return (1,0,0, color_bar_intensity) #If the number of passing interactions in a context is less than the average
    #number of interactions between all contexts, plot a red bar and adjust its intensity
```

```
for j, (index, row) in enumerate(combined_df.iterrows()):
    game_contexts = ['Home Games', 'Away Games', 'Against Top 10 Teams', 'Against Bottom 10 Teams', 'During 0-30 Mins of Games',
                'During 30-60 Mins of Games', 'During 60-90+ Mins of Games', 'When Winning During Games',
                    'When Tied During Games', 'When Losing During Games'] #These are all the unique game contexts that we will
    #be comparing
    context_passing_interactions_values = row[['Pass Interaction % - ' + game_context for game_context in game_contexts]]
    baseline_value = row['Pass Interaction % - In All Games']

    colors = [adjust_color_bar_intensity(value, baseline_value) for value in context_passing_interactions_values]

    ax = axes[j]
    ax.bar(game_contexts, context_passing_interactions_values, color=colors)
    ax.axhline(y=baseline_value, color='r', linestyle='--', label='In All Games')
    ax.set_title(f"Pass Interactions % - {row['fullName']}", fontsize=20)
    ax.set_ylabel('Pass Interactions %', fontsize=18)
    ax.set_xticks(range(len(game_contexts)))
    ax.set_xticklabels(game_contexts, rotation=90, ha='right', fontsize=15)
    ax.legend(fontsize=14)

for i in range(j+1, len(axes)):
    axes[i].axis('off')

plt.tight_layout()
plt.show()

#Each subplot represents the percentage of pass interactions a player was involved in for each game context. The red dashed
#line is the average percentage of pass interactions between all game contexts. This will be used as a baseline to compare the
#average number of interactions between all game contexts with each game context individually to see in which situations a
#player is more involved or less involved.

### Studying the Changes in Positions

pass_coords

#This dataframe has the average positions (coordinates) of each player for all games (base) and for different game contexts

pass_coords.to_csv('Average Coordinates.csv', index=True)

player_positions = dict({'A. Lacazette': 'Forward', 'P. Aubameyang': 'Forward', 'D. Tackie Mensah Welbeck': 'Forward',
                    'A. Sanchez': 'Forward', 'T. Walcott': 'Forward', 'O. Giroud': 'Forward', 'J. Wilshere': 'Midfielder',
                    'F. Coquelin': 'Midfielder', 'A. Oxlade-Chamberlain': 'Midfielder', 'M. Ozil': 'Midfielder',
                    'M. Elsayed Elneny': 'Midfielder', 'A. Maitland-Niles': 'Midfielder', 'H. Mkhitaryan': 'Midfielder',
                    'G. Xhaka': 'Midfielder', 'A. Ramsey': 'Midfielder', 'A. Iwobi': 'Midfielder', 'C. Chambers': 'Defender',
                    'P. Mertesacker': 'Defender', 'H. Bellerin': 'Defender', 'R. Holding': 'Defender',
                    'S. Kolasinac': 'Defender', 'S. Mustafi': 'Defender', 'L. Koscielny': 'Defender',
                    'K. Mavropanos': 'Defender', 'I. Monreal Eraso': 'Defender'})
#We will use this dictionary to plot the players separately by position

def creating_position_changes_scatterplots(pass_coordinates, player_position):
    players_by_position = [player_name for player_name, player_pos in player_positions.items() if player_pos == player_position]
    #We filter out and keep only players that have the position we want to plot
    player_position_df = pass_coords.loc[players_by_position] #We keep only rows that belong to players who play in the
    #specified position

    total_number_of_players = len(player_position_df)
    total_number_of_rows_in_graph = (total_number_of_players + 1) // 2 #To plot graphs when the number of players in a positon
    #is an odd number

    figure, axes = plt.subplots(total_number_of_rows_in_graph, 2, figsize=(10*2, 6*total_number_of_rows_in_graph))

    axes = axes.flatten()

    if total_number_of_players % 2 !=0:
        figure.delaxes(axes[-1])
        axes = axes[:-1] #If the total number of players for the specified position is an odd number, remove the last plot

    for ax, (player_name, row) in zip(axes, player_position_df.iterrows()):
        pitch = Pitch(pitch_type = 'statsbomb', pitch_color = 'grass', line_color = 'white') #We will plot the scatterplots on a
        #pitch using the mplsoccer.pitch library
        pitch.draw(ax=ax)

        game_contexts = ['base','home', 'away', 'top10', 'bottom10', '0_30', '30_60', '60_90', 'winning', 'tied', 'losing']
        context_colors = ['black','red', 'orange', 'yellow', 'green', 'purple', 'brown', 'pink', 'blue', 'gray', 'lightblue']

        for game_context, context_color in zip(game_contexts, context_colors):
            X_column = f'avg_start_x_{game_context}'
            Y_column = f'avg_start_y_{game_context}'
            ax.scatter(row[X_column], row[Y_column], s=200, alpha=0.6, c=context_color, label=f'{game_context.capitalize()}')
            #We use the average X,Y coordinates of players in each game context that we calculated above to plot the markers
            ax.annotate('', xy=(row[X_column], row[Y_column]), xytext=(row['avg_start_x_base'], row['avg_start_y_base']),
                        arrowprops=dict(arrowstyle='->', color=context_color, lw=2)) #We connect the average coordinates of
            #players in each game context with their average coordinates across all games with an arrow to show any significant
            #position changes in certain contexts

        ax.set_title(f'Average Positions of {player_name}', fontsize=17)
        ax.legend(loc='upper right', bbox_to_anchor=(1,1), fontsize=13)

    plt.tight_layout()
    plt.show()


#This function is used to plot on a pitch the average positions of players accross all games and for different game contexts.
#This will help us identify any significant position changes in different game contexts and how they differ from their
#average positions on the pitch

creating_position_changes_scatterplots(pass_coords, 'Defender')
#We plot the Defenders first

creating_position_changes_scatterplots(pass_coords, 'Midfielder')
#We plot the midfielders

creating_position_changes_scatterplots(pass_coords, 'Forward')
#We plot the forwards
```

In [ ]:

```
# Studying the Weights of Links Between Positions in Different Game Contexts

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

```
import seaborn as sns
import networkx as nx #This library will be used to plot the passing network graphs
from mplsoccer.pitch import Pitch #This library will enable us to show the networks on a pitch to also have details about player
#positions when making passes (that is why we caluclated the average coordinates above)
import matplotlib.patches as mpatches #This library is used to draw shapes and patches

pass_interactions_df = pd.read_csv('Pass Interactions.csv')

pass_interactions_df

positions = {'GK': (10,40), 'RWB': (50,70), 'LWB': (50,10), 'RCB': (35,55), 'LCB': (35,25), 'CB': (35,40), 'RCM': (55,50),
             'LCM': (55,30), 'LAM': (75,30), 'RAM': (75,50), 'ST': (90,40)} #For these graphs, we will be grouping players by
#their roles and plot them using fixed coordinates in a 3-4-2-1 formation. #Arsenal's most used formation during the 2017-18
#season was the 3-4-2-1 which they carried on from the previous season since it was very effective during the last few games.

#Reference link for Arsenal starting 11: https://www.bdfutbol.com/en/t/t2017-182016.html

position_colors = {'Goalkeeper': 'red', 'Defender': 'lightblue', 'Midfielder': 'lightgreen', 'Forward': 'yellow'}
#The node colors will be different based on the position of each role

minutes_played_by_role = dict({'GK': 3420, 'RWB': 3051, 'RCB': 3091, 'CB': 3461, 'LCB': 2438, 'LWB': 2147, 'RCM': 3123,
                               'LCM': 5367, 'RAM': 2779, 'LAM': 3666, 'ST': 4841})
#These are the total minutes played by player during the 2017-18 season. This information was extracted from FBREF:
#link: https://fbref.com/en/squads/18bb7c10/2017-2018/Arsenal-Stats

#We are going to use these values to normalize the weights of the interactions based on the total number of minutes played of
#players by role

def normalize_passing_interactions(row):
    return (1 / (minutes_played_by_role[row['source_role']] + minutes_played_by_role[row['target_role']])) * 1000

#After analyzing the changes of positions in players for different game contexts, and also their overall passing heatmaps, we
#notice that the position of the players is rather stable between different contexts. This suggests that Arsenal players don't
#tend to change positions that much. From the network graphs, we can see that Arsenal tend to play with wing backs, while
#their midfielders are concentrated within short spaces centrally, and there don't seem to be wingers. This is why we have
#assigned roles for each player in a 3-4-2-1 formation which was played a lot during the 2017-18 season.

#Our goal for this analysis is to create pitch pairs of player role interactions in different contexts together with a weighted
#difference graph which highlights the main differences in passing interactions between different contexts. While we want to
#visualize the total passing interactions between differen player roles, we also need to take into account that some player
#roles are more versatile then others, meaning that players in those positions are more easily able to play out of position,
# or it is more often that players assigned with the same roles might also play together.

#For example, a football team will always play with 1 Right-Back but at times might play with 2 Central-Attacking-Midfielders
#for tactical reasons.
#Or another example could be that in some games A. Maitland-Niles was playing out of position (plays as a Right-Back and in some others he plays as a
#Left-Central-Midfielder in other games. In our case, we don't have any information about the position of the player for each
#game rather than just their main position.

#To address this issue, we are going to normalize the weights of the interactions based on the number of minutes that players
#in a single role had during that season. For example, 1 game has 90 minutes so theoretically each player position will play
#90 minutes in a single game. Taking that into account, the Central-Attacking-midfielders have played more minutes than the
#Right-Backs meaning that in some scenarios more than 1 Central-Attacking-midfielder were playing together on the pitch (perhaps
#for tactical reasons as mentioned) but maybe they did not have the same role and one of them was playing out of position (maybe
#one CAM was playing more as a RCM for tactical reasons).

#For this reason we normalize the weights of player role interactions as followed:
#1/(total minutes of players assigned in Player Role A / total minutes of players assigned in Player Role B) * 1000

#Although this normalization is not 100% accurate, it makes the network graphs less biased towards more versatile roles where
#players with the same role can co-exist on the football pitch or in which the players can easily play out of position but still
#also favors the counts of passes.

pass_interactions_df['normalized_weight_interaction_by_role'] = pass_interactions_df.apply(normalize_passing_interactions,
                                                                                           axis=1)
pass_interactions_df['normalized_weight_interaction_by_role'] = pass_interactions_df['normalized_weight_interaction_by_role'].round(3)

pass_interactions_df

pass_interactions_df.to_csv('Pass Interactions by Roles.csv', index=True)

pass_interactions_df.loc[pass_interactions_df['source']=='T. Walcott', 'source_position'] = \
pass_interactions_df['source_position'].replace('Forward', 'Midfielder')

pass_interactions_df.loc[pass_interactions_df['target']=='T. Walcott', 'target_position'] = \
pass_interactions_df['target_position'].replace('Forward', 'Midfielder')

#For the purpose of these graphs we will be assigning Walcott as a midfielder since he is more of a right midfielder than a
#striker. This is for the coloring of the nodes.

#References used for constructing passing networks:

#https://medium.com/@yogakrisanto1129/a-step-by-step-guide-to-using-python-to-create-football-passing-networks-e00e92fecd99
#https://soccermatics.readthedocs.io/en/latest/gallery/lesson1/plot_PassNetworks.html
#https://mplsoccer.readthedocs.io/en/latest/gallery/pitch_plots/plot_pass_network.html
#https://networkx.org/documentation/stable/tutorial.html
#https://www.youtube.com/watch?v=pW7rltisoqo&t=285s

position_colors = {'Goalkeeper': 'red', 'Defender': 'lightblue', 'Midfielder': 'lightgreen', 'Forward': 'Yellow'}
#We map each position to a color for the network graphs.

def calculate_node_degree(G):
    node_degrees = {}
    for node in G.nodes():
        quantity_of_interactions = G.degree(node)  #The total number of interactions (both incoming and outgoing a player is
        #involved in
        unique_interactions_of_nodes = set(G.neighbors(node))
        diversity_of_interactions = len(unique_interactions_of_nodes) #The number of unique players a player interacts with
        strength_of_interactions = sum(G[node][node_neighbor]['weight'] for node_neighbor in unique_interactions_of_nodes)
        #The sum of the weights of interactions with each unique player
        combined_node_degree = quantity_of_interactions * np.log(diversity_of_interactions + 1) * strength_of_interactions
        #Combined Degree=Interaction Quantity×log(Interaction Diversity+1)×Strength of Interactions
        #This equation not only takes into account the quantity of interactions of a player but also the quality. The quality
        #of the player interactions highlights players that have strong connections with different players across the pitch
        #making them more important in the team's passing strategy.
        node_degrees[node] = combined_node_degree
    return node_degrees

def create_passing_network(df, context_column, context_value, position):
    passing_context_interactions = df.loc[df[context_column]==context_value]
```

```
    G = nx.DiGraph()

    for _, passing_interaction in passing_context_interactions.iterrows():
        passer = passing_interaction['source_role']
        pass_receiver = passing_interaction['target_role']
        pass_interaction_weight = passing_interaction['normalized_weight_interaction_by_role']

        if G.has_edge(passer, pass_receiver):
            G[passer][pass_receiver]['weight'] +=pass_interaction_weight
        else:
            G.add_edge(passer, pass_receiver, weight=pass_interaction_weight)

    #If the passing interaction already exists, we increase the weight of the interaction, otherwise we create a new edge
    #connected to the two nodes (pass source and pass target)

    combined_node_degrees = calculate_node_degree(G)

    #We ensure no player has a zero size by adding a small constant
    min_non_zero_degree = min(degree for degree in combined_node_degrees.values() if degree > 0)
    combined_node_degrees = {key: value + min_non_zero_degree * 0.5 for key, value in combined_node_degrees.items()}

    #We apply power law scaling to the node sizes to highlight differences
    maximum_combined_degree = max(combined_node_degrees.values())
    minimum_combined_degree = min(combined_node_degrees.values())
    node_sizes = [2500 * ((node_degree - minimum_combined_degree) / (maximum_combined_degree - minimum_combined_degree + 1e-9)) ** 1.5 + 50 for node_degree in

    #To highlight the key players, we first normalize the adjusted combined degree to a range of [1e-9,1] with a small constant to
    #ensure non-zero values.
    #Normalized Degree= (Adjusted Combined Degree-Min Adjusted Combined Degree)/ (Max Adjusted Combined Degree-Min Adjusted Combined Degree+€)

    #The next step is to apply power law scaling to the degrees to exaggerate more the differences:
    #Scaled Degree=(Normalized Degree)^1.5

    #To caluclate the final Node Sizes, we multiply the scaled degree by a scaling factor (2000) and add a constant (50) to
    #ensure a minimum size:
    #Node Size = 2000 × (Scaled Degree) + 50

    #We extract the colors of the nodes based on the position of players. This will make it easier for us to see how players in
    #different positions interact with each other.

    node_colors = {}

    for _, passing_interaction in passing_context_interactions.iterrows():
        node_colors[passing_interaction['source_role']] = position_colors.get(passing_interaction['source_position'], 'gray')
        node_colors[passing_interaction['target_role']] = position_colors.get(passing_interaction['target_position'], 'gray')

    list_of_node_colors = [node_colors[node] for node in G.nodes()]
    #We add colors to the nodes

    pos = positions
    passing_network_edges = G.edges(data=True)
    passing_network_weights = [passing_network_edge[2]['weight'] for passing_network_edge in passing_network_edges]
    #These are the nodes and edges of our network

    pitch = Pitch(pitch_type='statsbomb', pitch_color='grass', line_color='white') #We are going to plot the network graph on
    #a football pitch
    fig, ax = pitch.draw(figsize=(12,8))

    nx.draw(G, pos, ax=ax, with_labels=True, node_size=node_sizes, node_color=list_of_node_colors, font_size=12, font_weight='bold',
           edge_color=passing_network_weights, width=[net_width*0.2 for net_width in passing_network_weights], edge_cmap=plt.cm.Blues,
           connectionstyle='arc3,rad=0.2')
    #We construct the network graph

    legend_patches = [mpatches.Patch(color=color, label=position) for position, color in position_colors.items()]
    plt.legend(handles=legend_patches, loc='upper right', fontsize=13)

    plt.title(f'Player Passing Network - {context_column}: {context_value}', fontsize=16)
    plt.show()

#In this function we create the passing network that shows the strength of the passing interactions between different player
#roles for different game contexts.

def create_difference_graph(df, context_column, context_value1, context_value2, positions, position_colors):
    total_pass_interactions_value1 = len(df.loc[df[context_column] == context_value1])
    total_pass_interactions_value2 = len(df.loc[df[context_column] == context_value2])

    df_context_value1 = df.loc[df[context_column] == context_value1]
    df_context_value2 = df.loc[df[context_column] == context_value2]

    Graph1 = nx.DiGraph()

    for _, passing_interaction in df_context_value1.iterrows():
        passer = passing_interaction['source_role']
        pass_receiver = passing_interaction['target_role']
        pass_interaction_weight = (passing_interaction['normalized_weight_interaction_by_role'] / total_pass_interactions_value1) * 10000

        if Graph1.has_edge(passer, pass_receiver):
            Graph1[passer][pass_receiver]['weight'] += pass_interaction_weight
        else:
            Graph1.add_edge(passer, pass_receiver, weight = pass_interaction_weight)
        #We add the weight of the edge based on the interactions it has


    Graph2 = nx.DiGraph()

    for _, passing_interaction in df_context_value2.iterrows():
        passer = passing_interaction['source_role']
        pass_receiver = passing_interaction['target_role']
        pass_interaction_weight = (passing_interaction['normalized_weight_interaction_by_role'] / total_pass_interactions_value2) * 10000
        #We want to calculate the differences of the proportions of each pass interactions. Due to the fact that the count of
        #pass interactions is different accross multiple contexts, our aim is not to see the differences in counts, but how
        #the distribution of pass intercations changes over different game contexts.

        #For example the weight of counts of pass interactions between the RCB-RB may be less in home games than in away games
        #because at home games there are more pass interactions completed, but the proportion of this interaction may be higher
        #for away games than in home games, meaning that although the away passes are fewer than the home passes, the team in
        #away games tends to play more from the right than in home games. To understand the difference of this, we need to
        #calculate the difference of proportions for comparable game contexts.

        if Graph2.has_edge(passer, pass_receiver):
            Graph2[passer][pass_receiver]['weight'] += pass_interaction_weight
```

```
        else:
            Graph2.add_edge(passer, pass_receiver, weight = pass_interaction_weight)
         #We add the weight of the edge based on the interactions it has

    edge_differences_between_two_contexts = {}
    for (i, value, dataframe) in Graph1.edges(data=True):
        if Graph2.has_edge(i, value):
            edge_differences_between_two_contexts[(i, value)] = dataframe['weight'] - Graph2[i][value]['weight']
            #If the edge exists in both contexts (G1 and G2),
            #We calculate the difference G1 - G2. If the result is positive, then G1>G2 meaning the distribution of this pass
            #interaction is higher in the first context (G1), else if it is negative then G2>G1 meaning the distribution of
            #this pass interactions is higher in the second context.
        else:
            edge_differences_between_two_contexts[(i, value)] = dataframe['weight']
    for (i, value, dataframe) in Graph2.edges(data=True):
        if (i, value) not in edge_differences_between_two_contexts:
            edge_differences_between_two_contexts[(i, value)] = -dataframe['weight']

    Graph_differences = nx.DiGraph()
    for (i, value), weight in edge_differences_between_two_contexts.items():
        if weight != 0:
            Graph_differences.add_edge(i, value, weight=weight)
            #We calculate the differences of weights for each pass node interaction (pass interaction) for each context (home vs away
    #games)

    node_colors = {}
    for node in Graph_differences.nodes():
        position = df_context_value1.loc[df_context_value1['source_role'] == node, 'source_position'].values
        if len(position) == 0:
            position = df_context_value2.loc[df_context_value2['source_role'] == node, 'source_position'].values

        position = position[0] if len(position) > 0 else 'Unknown'
        node_colors[node] = position_colors.get(position, 'gray')

    each_node_size = 2000

    pos = {role: positions[role] for role in Graph_differences.nodes()}
    passing_network_edges = Graph_differences.edges(data=True)
    passing_network_weights = [passing_network_edge[2]['weight'] for passing_network_edge in passing_network_edges]

    pitch = Pitch(pitch_type='statsbomb', pitch_color='grass', line_color='white')
    fig, ax = pitch.draw(figsize=(12,8))

    nx.draw(Graph_differences, pos, ax=ax, with_labels=True, node_size=each_node_size, node_color=[node_colors[n] for n in Graph_differences.nodes()],
            font_size=12, font_weight='bold', edge_color=passing_network_weights,
            width=[abs(passing_network_weight)*0.5 for passing_network_weight in passing_network_weights], edge_cmap=plt.cm.RdBu,
            connectionstyle='arc3,rad=0.2', edge_vmin=-max(passing_network_weights), edge_vmax=max(passing_network_weights))

    blue_patch = mpatches.Patch(color='blue', label = f'More in {context_column} = {context_value1}')
    #Based on the colorscale RdBu, blue represents positive differences (in our case where G1-G2>0 meaning that the edge in the
    #first context has a bigger weight than in the second).
    red_patch = mpatches.Patch(color='red', label = f'More in {context_column} = {context_value2}')
    #Based on the colorscale RdBu, red represents negative differences (in our case where G1-G2<0 meaning that the edge in the first context has a
    #lower weight than in the second).
    legend_edges = ax.legend(handles=[blue_patch, red_patch], loc='upper left', fontsize=13, title='Edge Colors')

    legend_patches = [mpatches.Patch(color=color, label=position) for position, color in position_colors.items()]
    legend_nodes = ax.legend(handles=legend_patches, loc='upper right', fontsize=13, title='Node Colors')

    ax.add_artist(legend_edges)
    ax.add_artist(legend_nodes)

    plt.title(f'Passing Network Difference Graphs - {context_column}: {context_value1} vs {context_value2}', fontsize=16)
    plt.show()

position_colors = {'Goalkeeper': 'red', 'Defender': 'lightblue', 'Midfielder': 'lightgreen', 'Forward': 'Yellow'}

#In this function we create a difference graph of the pass interactions between different player roles for comparable game
#contexts (e.g. home vs away games).
#Difference graphs can help us understand how the passing strategy changes based on the context of a game.

create_passing_network(pass_interactions_df, 'home_game', 1, positions)
create_passing_network(pass_interactions_df, 'home_game', 0, positions)
create_difference_graph(pass_interactions_df, 'home_game', 1, 0, positions, position_colors)

create_passing_network(pass_interactions_df, 'opponent_strength', 'Top 10 Team', positions)
create_passing_network(pass_interactions_df, 'opponent_strength', 'Bottom 10 Team', positions)
create_difference_graph(pass_interactions_df, 'opponent_strength', 'Top 10 Team', 'Bottom 10 Team', positions, position_colors)

create_passing_network(pass_interactions_df, 'time_segment', '0-30', positions)
create_passing_network(pass_interactions_df, 'time_segment', '30-60', positions)
create_passing_network(pass_interactions_df, 'time_segment', '60-90+', positions)
create_difference_graph(pass_interactions_df, 'time_segment', '0-30', '30-60', positions, position_colors)
create_difference_graph(pass_interactions_df, 'time_segment', '30-60', '60-90+', positions, position_colors)

def calculate_node_degree(G):
    node_degrees = {}
    for node in G.nodes():
        quantity_of_interactions = G.degree(node)  #The total number of interactions (both incoming and outgoing a player is
        #involved in
        unique_interactions_of_nodes = set(G.neighbors(node))
        diversity_of_interactions = len(unique_interactions_of_nodes) #The number of unique players a player interacts with
        strength_of_interactions = sum(G[node][node_neighbor]['weight'] for node_neighbor in unique_interactions_of_nodes)
        #The sum of the weights of interactions with each unique player
        combined_node_degree = quantity_of_interactions * np.log(diversity_of_interactions + 1) * strength_of_interactions
        #Combined Degree=Interaction Quantity×log(Interaction Diversity+1)×Strength of Interactions
        #This equation not only takes into account the quantity of interactions of a player but also the quality. The quality
        #of the player interactions highlights players that have strong connections with different players across the pitch
        #making them more important in the team's passing strategy.
        node_degrees[node] = combined_node_degree
    return node_degrees

def create_passing_network(df, context_column, context_value, position):

    passing_context_interactions = df

    G = nx.DiGraph()

    for _, passing_interaction in passing_context_interactions.iterrows():
        passer = passing_interaction['source_role']
        receiver = passing_interaction['target_role']
```

```
        pass_interaction_weight = passing_interaction['normalized_weight_interaction_by_role']

        if G.has_edge(passer, pass_receiver):
            G[passer][pass_receiver]['weight'] +=pass_interaction_weight
        else:
            G.add_edge(passer, pass_receiver, weight=pass_interaction_weight)

    #If the passing interaction already exists, we increase the weight of the interaction, otherwise we create a new edge
    #connected to the two nodes (pass source and pass target)

    combined_node_degrees = calculate_node_degree(G)

    #We ensure no player has a zero size by adding a small constant
    min_non_zero_degree = min(degree for degree in combined_node_degrees.values() if degree > 0)
    combined_node_degrees = {key: value + min_non_zero_degree * 0.5 for key, value in combined_node_degrees.items()}

    #We apply power law scaling to the node sizes to highlight differences
    maximum_combined_degree = max(combined_node_degrees.values())
    minimum_combined_degree = min(combined_node_degrees.values())
    node_sizes = [2500 * ((node_degree - minimum_combined_degree) / (maximum_combined_degree - minimum_combined_degree + 1e-9)) ** 1.5 + 50 for node_degree in

    #To highlight the key players, we first normalize the adjusted combined degree to a range of [1e-9,1] with a small constant to
    #ensure non-zero values.
    #Normalized Degree= (Adjusted Combined Degree-Min Adjusted Combined Degree)/ (Max Adjusted Combined Degree-Min Adjusted Combined Degree+€)

    #The next step is to apply power law scaling to the degrees to exaggerate more the differences:
    #Scaled Degree=(Normalized Degree)^1.5

    #To caluclate the final Node Sizes, we multiply the scaled degree by a scaling factor (2000) and add a constant (50) to
    #ensure a minimum size:
    #Node Size = 2000 × (Scaled Degree) + 50

    #We extract the colors of the nodes based on the position of players. This will make it easier for us to see how players in
    #different positions interact with each other.

    node_colors = {}

    for _, passing_interaction in passing_context_interactions.iterrows():
        node_colors[passing_interaction['source_role']] = position_colors.get(passing_interaction['source_position'], 'gray')
        node_colors[passing_interaction['target_role']] = position_colors.get(passing_interaction['target_position'], 'gray')

    list_of_node_colors = [node_colors[node] for node in G.nodes()]
    #We add colors to the nodes

    pos = positions
    passing_network_edges = G.edges(data=True)
    passing_network_weights = [passing_network_edge[2]['weight'] for passing_network_edge in passing_network_edges]
    #These are the nodes and edges of our network

    pitch = Pitch(pitch_type='statsbomb', pitch_color='grass', line_color='white') #We are going to plot the network graph on
    #a football pitch
    fig, ax = pitch.draw(figsize=(12,8))

    nx.draw(G, pos, ax=ax, with_labels=True, node_size=node_sizes, node_color=list_of_node_colors, font_size=12, font_weight='bold',
            edge_color=passing_network_weights, width=[net_width*0.2 for net_width in passing_network_weights], edge_cmap=plt.cm.Blues,
            connectionstyle='arc3,rad=0.2')
    #We construct the network graph

    legend_patches = [mpatches.Patch(color=color, label=position) for position, color in position_colors.items()]
    plt.legend(handles=legend_patches, loc='upper right', fontsize=13)

    plt.title(f'Player Passing Network - {context_column}: {context_value}', fontsize=16)
    plt.show()

#In this function we create the passing network that shows the strength of the passing interactions between different player
#roles for different game contexts.

context_values = {'winning': pass_interactions_df[pass_interactions_df['score_difference'] >= 1],
    'tied': pass_interactions_df[pass_interactions_df['score_difference'] == 0],
    'losing': pass_interactions_df[pass_interactions_df['score_difference'] <= -1]}

for context, df in context_values.items():
    create_passing_network(df, 'score_difference', context, positions)


def create_difference_graph(df, context_column, context_value1, context_value2, positions, position_colors):
    total_pass_interactions_value1 = len(df.loc[df[context_column] == context_value1])
    total_pass_interactions_value2 = len(df.loc[df[context_column] == context_value2])

    df_context_value1 = df.loc[df[context_column] == context_value1]
    df_context_value2 = df.loc[df[context_column] == context_value2]

    Graph1 = nx.DiGraph()

    for _, passing_interaction in df_context_value1.iterrows():
        passer = passing_interaction['source_role']
        pass_receiver = passing_interaction['target_role']
        pass_interaction_weight = (passing_interaction['normalized_weight_interaction_by_role'] / total_pass_interactions_value1) * 10000

        if Graph1.has_edge(passer, pass_receiver):
            Graph1[passer][pass_receiver]['weight'] += pass_interaction_weight
        else:
            Graph1.add_edge(passer, pass_receiver, weight = pass_interaction_weight)
        #We add the weight of the edge based on the interactions it has

    Graph2 = nx.DiGraph()

    for _, passing_interaction in df_context_value2.iterrows():
        passer = passing_interaction['source_role']
        pass_receiver = passing_interaction['target_role']
        pass_interaction_weight = (passing_interaction['normalized_weight_interaction_by_role'] / total_pass_interactions_value2) * 10000
        #We want to calculate the differences of the proportions of each pass interactions. Due to the fact that the count of
        #pass interactions is different accross multiple contexts, our aim is not to see the differences in counts, but how
        #the distribution of pass intercations changes over different game contexts.

        #For example the weight of counts of pass interactions between the RCB-RB may be less in home games than in away games
        #because at home games there are more pass interactions completed, but the proportion of this interaction may be higher
        #for away games than in home games, meaning that although the away passes are fewer than the home passes, the team in
        #away games tends to play more from the right than in home games. To understand the difference of this, we need to
        #calculate the difference of proportions for comparable game contexts.
        aph2.has_edge(passer, pass_receiver):
```

```
            Graph2[passer][pass_receiver]['weight'] += pass_interaction_weight
        else:
            Graph2.add_edge(passer, pass_receiver, weight = pass_interaction_weight)
        #We add the weight of the edge based on the interactions it has


    edge_differences_between_two_contexts = {}
    for (i, value, dataframe) in Graph1.edges(data=True):
        if Graph2.has_edge(i, value):
            edge_differences_between_two_contexts[(i, value)] = dataframe['weight'] - Graph2[i][value]['weight']
            #If the edge exists in both contexts (G1 and G2),
            #We calculate the difference G1 - G2. If the result is positive, then G1>G2 meaning the distribution of this pass
            #interaction is higher in the first context (G1), else if it is negative then G2>G1 meaning the distribution of
            #this pass interactions is higher in the second context.
        else:
            edge_differences_between_two_contexts[(i, value)] = dataframe['weight']
    for (i, value, dataframe) in Graph2.edges(data=True):
        if (i, value) not in edge_differences_between_two_contexts:
            edge_differences_between_two_contexts[(i, value)] = -dataframe['weight']

    Graph_differences = nx.DiGraph()
    for (i, value), weight in edge_differences_between_two_contexts.items():
        if weight != 0:
            Graph_differences.add_edge(i, value, weight=weight)
    #We calculate the differences of weights for each pass node interaction (pass interaction) for each context (home vs away
    #games)
    node_colors = {}
    for node in Graph_differences.nodes():
        position = df_context_value1.loc[df_context_value1['source_role'] == node, 'source_position'].values
        if len(position) == 0:
            position = df_context_value2.loc[df_context_value2['source_role'] == node, 'source_position'].values

        position = position[0] if len(position) > 0 else 'Unknown'
        node_colors[node] = position_colors.get(position, 'gray')

    each_node_size = 2000

    pos = {role: positions[role] for role in Graph_differences.nodes()}
    passing_network_edges = Graph_differences.edges(data=True)
    passing_network_weights = [passing_network_edge[2]['weight'] for passing_network_edge in passing_network_edges]

    pitch = Pitch(pitch_type='statsbomb', pitch_color='grass', line_color='white')
    fig, ax = pitch.draw(figsize=(12,8))

    nx.draw(Graph_differences, pos, ax=ax, with_labels=True, node_size=each_node_size, node_color=[node_colors[n] for n in Graph_differences.nodes()],
            font_size=12, font_weight='bold', edge_color=passing_network_weights,
            width=[abs(passing_network_weight)*0.5 for passing_network_weight in passing_network_weights], edge_cmap=plt.cm.RdBu,
            connectionstyle='arc3,rad=0.2', edge_vmin=-max(passing_network_weights), edge_vmax=max(passing_network_weights))

    blue_patch = mpatches.Patch(color='blue', label = f'More in {context_column} = {context_value1}')
    #Based on the colorscale RdBu, blue represents positive differences (in our case where G1-G2>0 meaning that the edge in the
    #first context has a bigger weight than in the second).
    red_patch = mpatches.Patch(color='red', label = f'More in {context_column} = {context_value2}')
    #Based on the colorscale RdBu, red represents negative differences (in our case where G1-G2<0 meaning that the edge in the first context has a
    #lower weight than in the second).
    legend_edges = ax.legend(handles=[blue_patch, red_patch], loc='upper left', fontsize=13, title='Edge Colors')

    legend_patches = [mpatches.Patch(color=color, label=position) for position, color in position_colors.items()]
    legend_nodes = ax.legend(handles=legend_patches, loc='upper right', fontsize=13, title='Node Colors')

    ax.add_artist(legend_edges)
    ax.add_artist(legend_nodes)

    plt.title(f'Passing Network Difference Graphs - {context_column}: {context_value1} vs {context_value2}', fontsize=16)
    plt.show()

position_colors = {'Goalkeeper': 'red', 'Defender': 'lightblue', 'Midfielder': 'lightgreen', 'Forward': 'Yellow'}

#In this function we create a difference graph of the pass interactions between different player roles for comparable game
#contexts (e.g. home vs away games).
#Difference graphs can help us understand how the passing strategy changes based on the context of a game.

#We call the function using the appropriate parameters
create_difference_graph(pass_interactions_df, 'current_score_grouped', 'Winning', 'Tied', positions, position_colors)
create_difference_graph(pass_interactions_df, 'current_score_grouped', 'Tied', 'Losing', positions, position_colors)
create_difference_graph(pass_interactions_df, 'current_score_grouped', 'Winning', 'Losing', positions, position_colors)

#NOTE: To understand better what these graphs represent, we need to understand what the width and color of the edges are showing
#us.

#Each network graph that shows the passing interactions between various player roles for a specific context (e.g. When the score
#is tie) is showing the count of pass interactions between the player roles. For these graphs, the wider the edge, the more pass
#interactions (counts) have been completed.

#Each difference graph (e.g. Weight Difference Graph - When Winning During Games vs When the Score is Tied During Games) is
#comparing the distribution of passes between the 2 contexts.
#Our aim is not to compare the counts of passes due to the fact that the number of pass interactions varies accross different
#contexts.

#For example, when comparing the 2 individual network graphs which show the counts of pass interactions, we can see that in
#games where the score is tied slightly more pass interactions are being made overall, then when Arsenal is winning, meaning
#that Arsenal may seem to find it difficult to take the lead, or that when Arsenal take the lead, they tend to play more
#defensively with less passes.

#But when we compare the difference graph between winning and when the score is tied, we can understand how the distributions of
#pass interactions (proportions) are different when Arsenal is winning or when they are tied. This will help us understand which
#pass interactions tend to happen more often in different contexts and not the counts of passing interactions. In this way, the
#network graphs are not biased.
```

In [ ]:

```
# Creating Player Heatmaps for Passing Distribution

import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from mplsoccer.pitch import Pitch
from matplotlib.colors import LinearSegmentedColormap

#We import the necessary libraries.

events_df = pd.read_csv('final_dataset.csv')
```

```
events_df = events_df.loc[(events_df['eventName'] == 'Pass') & (events_df['fullName'].notna())]
#We filter out the dataset to produce heatmaps only for players who we know their names. We keep only passing events since we
#only want to show the spatial distribution of passing.

events_df = events_df[['fullName', 'teamId', 'start_x', 'start_y', 'position']]
#We are going to use the start x,y coordinates of the passes because we are interested more in the positions where players
#pass the ball and not receive it. We are also going to use the 'position' column to filter out the datasets and plot together
#players by position.

#We scale the x,y coordinates to match the Pitch coordinates. In our events dataset, the scale of both x,y coordinates is 0-100
#and the coordinates for the pitch library are x=0-120 and y=0-80. So we apply the necessary transformations.


events_df

#References used for coding to create player heatmaps:

#https://medium.com/@abhishekmahajan_24059/analyzing-football-game-event-data-using-mplsoccer-in-python-d95d751682c2
#https://fcpython.com/visualisation/football-heatmaps-seaborn

import math

def create_passing_heatmap_by_player(df, position):
    player_position_df = events_df.loc[events_df['position']==position] #We want to show the heatmaps for goalkeepers,
    #defenders, midfielders, and forwards separately.
    total_number_of_players = player_position_df['fullName'].unique().tolist()
    total_number_of_columns = 3
    total_number_of_rows = math.ceil(len(total_number_of_players) / total_number_of_columns) #We use math.ceil to ensure that
    #enough rows are created based on the number of players
    figure, axes = plt.subplots(nrows = total_number_of_rows, ncols = total_number_of_columns,
                                figsize=(12 * total_number_of_columns, 9 * total_number_of_rows))

    axes = axes.flatten() if len(total_number_of_players) > 1 else[axes] ##We plot the players heatmap for each position
    #side-by-side to make the comparisons easier.

    for ax, unique_player in zip(axes, total_number_of_players):
        unique_player_df = player_position_df.loc[player_position_df['fullName']==unique_player] #We filter out the dataset and
        #keep only passing events for each unique player
        pitch = Pitch(pitch_type='statsbomb', pitch_color='grass', line_color='white')
        pitch.draw(ax=ax)
        sns.kdeplot(x=unique_player_df['start_x'], y=unique_player_df['start_y'], ax=ax, fill=True, cmap='hot', alpha=0.7,
                    levels=90, bw_adjust=0.5) #The KDE plots are plotted on the pitch graph and we fill the KDE lines so we
        #can see in which areas of the pitch there is a higher concentration of passing events. #We use the KDE plot with
        #fill=True to plot the heatmaps, using the scaled x and y coordinates when a pass is being made by a player
        ax.set_title(f'Spatial Distribution of Passes - {unique_player}', fontsize=30)

    for ax in axes[len(total_number_of_players):]:
        ax.set_visible(False) #We make sure to hide any axes that are empty and don't plot a heatmap of a player

    plt.subplots_adjust(wspace=0.1, hspace=0.05)
    plt.tight_layout()
    plt.show()

create_passing_heatmap_by_player(events_df, 'Defender')
#We plot the heatmaps for defenders.

create_passing_heatmap_by_player(events_df, 'Midfielder')
#We plot the heatmaps for midfielders.

create_passing_heatmap_by_player(events_df, 'Forward')
#We plot the heatmaps for forwards.
```

In [ ]:

```
# Passing Event Based Factor Analysis

import pandas as pd
import numpy as np

arsenal_events = pd.read_csv('final_dataset.csv')

arsenal_events

#We first load the dataset that has all of the events of Arsenal player across all games of the 2017-18 season.

## Data Preprocessing - Preparing the Data for Factor Analysis

arsenal_events_only_passes = arsenal_events.loc[(arsenal_events['eventName']=='Pass') & (arsenal_events['teamId']==1609)]

arsenal_events_only_passes

#Since we are going to perform a passing event based factor analysis, we filter out the dataset and only keep passing events.

### Extracting information about the types of passes from the 'subEventName' column

passes_by_player = arsenal_events_only_passes.groupby(['fullName', 'subEventName']).count()[['teamId']]

passes_by_player

passes_by_player = arsenal_events_only_passes.groupby(['fullName', 'subEventName']).count()[['teamId']]

passes_by_player.reset_index(inplace=True)

pivoted_df = passes_by_player.pivot(index='fullName', columns='subEventName', values='teamId')

pivoted_df = pivoted_df.drop('Hand pass', axis=1)

pivoted_df = pivoted_df.fillna(0)

pivoted_df_perc = pivoted_df.div(pivoted_df.sum(axis=1), axis=0) * 100

pivoted_df_perc = pivoted_df_perc.round(0)

pivoted_df_perc.rename_axis('Player Name', axis='index', inplace=True)

pivoted_df_perc.rename(columns={'Cross': 'Cross %', 'Head pass': 'Head Pass %', 'High pass': 'High Pass %',
                                'Launch': 'Launch %', 'Simple pass': 'Simple Pass %', 'Smart pass': 'Smart Pass %'},
                       inplace=True)
if isinstance(pivoted_df_perc.columns, pd.MultiIndex):
    pivoted_df_perc.columns = pivoted_df_perc.columns.droplevel(0)
else:
```

```
    pivoted_df_perc.columns.name = None

pivoted_df_perc

#The first few columns that we are going to use for our factor analysis are the percentages of these types of passes being made
#by each player. We will use percentages instead of counts, since some players play more than others and playing time is not
#equally distributed across the team. We are more interested in using the distibution of each passing type by player rather
#than just counting the total passes.

### Extracting information about the types of passes from the 'tags_description' column

dict(arsenal_events_only_passes['tags_description'].value_counts())
#Using the tags_description variable we can also extract information about more detailed types of passes being made by each
#player. Looking at the dictionary below, there are various descriptions of passes so we need to find a way to group different
#tags together to have fewer variables for our factor analysis.

def rename_tags(value):
    if ('Counter attack,' in value) & ('Accurate' in value):
        return 'Counter Attack Pass - Accurate'
    elif ('Counter attack,' in value) & ('Not accurate' in value):
        return 'Counter Attack Pass - Not Accurate'
    elif ('Key pass,' in value) & ('Accurate' in value):
        return 'Key Pass - Accurate'
    elif ('Key pass,' in value) & ('Not accurate' in value):
        return 'Key Pass - Not Accurate'
    elif ('Assist,' in value) & ('Accurate' in value):
        return 'Assist - Accurate'
    elif ('Assist,' in value) & ('Not accurate' in value):
        return 'Assist - Not Accurate'
    elif ('Dangerous ball' in value) & ('Accurate' in value):
        return 'Dangerous Ball - Accurate'
    elif ('Dangerous ball' in value) & ('Not accurate' in value):
        return ('Dangerous Ball - Not Accurate')
    elif ('Through' in value) & ('Accurate' in value):
        return ('Through Ball - Accurate')
    elif ('Through' in value) & ('Not accurate' in value):
        return 'Through Ball - Not Accurate'
    elif ('Interception' in value) & ('Accurate' in value):
        return 'Interception - Accurate'
    elif ('Interception' in value) & ('Not accurate' in value):
        return 'Interception - Not Accurate'
    elif ('High,' in value) & ('Accurate' in value):
        return 'Final Third Pass - Accurate'
    elif ('High,' in value) & ('Not accurate' in value):
        return 'Final Third Pass - Not Accurate'
    else:
        return value

#These are the transformations that helped us decrease the number of unique tags:

#1. All tags that start with 'Counter attack' were grouped as Counter Attack Passes
#2. All tags that start with 'Key pass' were grouped as Key Passes
#3. All tags that start with 'Assist' were grouped as Assists
#4. All tags that start with 'Dangerous ball' were grouped as Dangerous Balls
#5. All tags that start with 'Through ball' were grouped as Through Balls
#6. All tags that start with 'Interception' were grouped as Intercepted Passes
#7. All tags that start with 'High' were grouped as Final Third Passes

#NOTE: While renaming the tag values, we also added if the pass was accurate or not, because we want to use the
#Pass Accuracy % for our factor analysis. So we need to know whether a pass of each pass type was accurate or not.

arsenal_events_only_passes['tags_description'] = arsenal_events_only_passes['tags_description'].apply(rename_tags)

#We apply the function

arsenal_events_only_passes = arsenal_events_only_passes.loc[arsenal_events_only_passes['tags_description'] != 'Right foot, Not accurate']
arsenal_events_only_passes = arsenal_events_only_passes.loc[arsenal_events_only_passes['tags_description'] != 'Left foot, Blocked, Not accurate']
arsenal_events_only_passes = arsenal_events_only_passes.loc[arsenal_events_only_passes['tags_description'] != 'Left foot, Not accurate']
arsenal_events_only_passes = arsenal_events_only_passes.loc[arsenal_events_only_passes['tags_description'] != 'Right foot, Blocked, Not accurate']
arsenal_events_only_passes = arsenal_events_only_passes.loc[arsenal_events_only_passes['tags_description'] != 'Fairplay, Not accurate']
arsenal_events_only_passes = arsenal_events_only_passes.loc[arsenal_events_only_passes['tags_description'] != 'Left foot, Accurate']
arsenal_events_only_passes = arsenal_events_only_passes.loc[arsenal_events_only_passes['tags_description'] != 'Right foot, Accurate']
arsenal_events_only_passes = arsenal_events_only_passes.loc[arsenal_events_only_passes['tags_description'] != 'Fairplay, Accurate']

#We filter out the dataset and remove any rows that have irrelevant tag descriptions. Not many rows were removed so we didn't
#lose a signficiant amount of data.

arsenal_events_only_passes['tags_description'].value_counts()

#These are the value counts of each tag description. While most passes are described as accurate or not accurate, there is still
#a good sample of data which has a more detailed description.

pass_types_by_player = arsenal_events_only_passes.groupby(['fullName', 'tags_description']).count()[['teamId']]

pass_types_by_player.reset_index(inplace=True)

pivoted_df2 = pass_types_by_player.pivot(index='fullName', columns='tags_description', values='teamId')

pivoted_df2 = pivoted_df2.fillna(0)

#We group the data by the player name and the tag description and pivot the data so each row is a unique player and each column
#is a feature that we aim to use for the factor analysis

pivoted_df2

pivoted_df2['Total Accurate Passes'] = pivoted_df2[pivoted_df2.columns[pivoted_df2.columns.str.contains('Accurate') & ~pivoted_df2.columns.str.contains('Not')]

pivoted_df2['Total Not Accurate Passes'] = pivoted_df2[pivoted_df2.columns[pivoted_df2.columns.str.contains('Not')]].sum(axis=1)

pivoted_df2['% Pass Accuracy'] = (pivoted_df2['Total Accurate Passes'] / (pivoted_df2['Total Accurate Passes'] + pivoted_df2['Total Not Accurate Passes'])) *10

pivoted_df2 = pivoted_df2.drop(['Accurate', 'Not accurate', 'Total Accurate Passes', 'Total Not Accurate Passes'], axis=1)

#Here is where we can understand why adding whether a pass was accurate or not in the description for each pass type.
#We first count the total accurate passes by summing all of the description that have the word 'Accurate' in the tag.
#We then count the total non accurate passes by summing all of the description that have the word 'Not Accurate' in the tag.
#Finally using these numbers, we find the passing accuracy % of each player


pivoted_df2['Total Assists'] = pivoted_df2['Assist - Accurate'] + pivoted_df2['Assist - Not Accurate']
```

```
pivoted_df2['Total Counter Attack Passes'] = pivoted_df2['Counter Attack Pass - Accurate'] + pivoted_df2['Counter Attack Pass - Not Accurate']

pivoted_df2['Total Key Passes'] = pivoted_df2['Key Pass - Accurate'] + pivoted_df2['Key Pass - Not Accurate']

pivoted_df2['Total Dangerous Balls'] = pivoted_df2['Dangerous Ball - Accurate'] + pivoted_df2['Dangerous Ball - Not Accurate']

pivoted_df2['Total Final Third Passes'] = pivoted_df2['Final Third Pass - Accurate'] + pivoted_df2['Final Third Pass - Not Accurate']

pivoted_df2['Total Interceptions'] = pivoted_df2['Interception - Accurate'] + pivoted_df2['Interception - Not Accurate']

pivoted_df2['Total Through Balls'] = pivoted_df2['Through Ball - Accurate'] + pivoted_df2['Through Ball - Not Accurate']

#Now to find the percentage of each pass, we first find the count of each pass type (accurate and not accurate passes)

columns_to_drop = pivoted_df2.columns.tolist()[:14]

pivoted_df2 = pivoted_df2.drop(columns_to_drop, axis=1)

#We keep the variables that have the total passes of each pass type


columns_to_summarize = ['Total Counter Attack Passes', 'Total Key Passes', 'Total Dangerous Balls', 'Total Final Third Passes',
                  'Total Interceptions', 'Total Through Balls']
pivoted_df2['Total Relevant Passes'] = pivoted_df2[columns_to_summarize].sum(axis=1)

for column in ['Total Counter Attack Passes', 'Total Key Passes', 'Total Dangerous Balls',
               'Total Final Third Passes', 'Total Interceptions', 'Total Through Balls']:
    pivoted_df2[f'% {column}'] = (pivoted_df2[column] / pivoted_df2['Total Relevant Passes']) * 100

#And now we find the percentage of each pass type made by each player.

#NOTE: We filter out the dataset and keep only rows that have a description about the pass type and not just if the pass was
#accurate or not (tags_description[i]='Accurate' or tags_description[i]='Not accurate' where i is some random index of a row)
#because a large proportion of the tags description just describes if a pass was accurate or not without giving any information
#about the pass type. This would lead to very small values so it is better to just compare the percentages of different pass
#types (only rows that have information about the pass type e.g. 'Key Pass' and excluding rows that have values in the
#tags_description column 'Accurate' or 'Not accurate'.
#Although the number of rows that provide a description of the pass type is quite low, there is still a signficiant amount of
#rows that do provide information about the pass type and can be utiziled to find what types of passes (tag description) players
#make.

#NOTE: Due to the number of assists by player being low, we will just use the count of assists by player (instead of percentage
#or anything else).

pivoted_df2.drop(columns=['Total Relevant Passes'], inplace=True)

columns_to_drop = pivoted_df2.columns.tolist()[2:8]

pivoted_df2 = pivoted_df2.drop(columns_to_drop, axis=1)

pivoted_df2 = pivoted_df2.round(0)

pivoted_df2 = pivoted_df2.fillna(0)

#We filter out the dataframe and only keep relevant columns (columns describing percentages).

if isinstance(pivoted_df2.columns, pd.MultiIndex):
    pivoted_df2.columns = pivoted_df2.columns.droplevel(0)
else:
    pivoted_df2.columns.name = None

pivoted_df2

### Extracting information about certain pass metrics

arsenal_events = pd.read_csv('final_dataset.csv')

arsenal_events_only_passes = arsenal_events.loc[(arsenal_events['eventName']=='Pass') & (arsenal_events['teamId']==1609)]

arsenal_events_only_passes = arsenal_events_only_passes.loc[arsenal_events_only_passes['fullName']!=0]

#For our factor analysis, we will also include information about the length and angle of passes, while also about the horizontal
#and vertical change of passes.

passing_stats_by_player = arsenal_events_only_passes.groupby('fullName').mean()[['event_length', 'event_angle',
                                                                                  'vertical_change', 'horizontal_change']]

passing_stats_by_player.rename(columns={'event_length': 'Avg Length of Passes', 'event_angle': 'Avg Angle of Passes',
                                        'vertical_change': 'Avg Vertical Change of Passes',
                                        'horizontal_change': 'Avg Horizontal Change of Passes'}, inplace=True)
passing_stats_by_player = passing_stats_by_player.round(1)

passing_stats_by_player

#We find the average length and angle of passes and the average vertical and horizontal change of passes by player.

### Merging the datasets and applying extra preprocessing

factor_analysis_input = pd.concat([pivoted_df_perc, pivoted_df2, passing_stats_by_player], axis=1)

factor_analysis_input = factor_analysis_input.loc[factor_analysis_input.index != 'P. Cech']

factor_analysis_input = factor_analysis_input.loc[factor_analysis_input.index != 'D. Ospina']

factor_analysis_input

#We concatenate the 3 dataframes that we created:
#1. Dataframe with information about the types of passes from the 'subEventName' (Percentages)
#2. Dataframe with information about the types of passes from the 'tags_description' column (Percentages)
#3. Dataframe with information about certain pass metrics (Averages)

#Below is the final dataset that we aim to use for the factor analysis.

factor_analysis_input.to_csv('Factor Analysis Input Dataset.csv', index=True)
#Now that the dataset is complete, we convert it to a csv file.

### Checking Colinearity and KMO and Bartlett Test Scores before applying factor analysis to remove noise

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
```

```
columns_to_standarize= factor_analysis_input.columns.tolist()

factor_analysis_input[columns_to_standarize] = scaler.fit_transform(factor_analysis_input[columns_to_standarize])

factor_analysis_input

#Before applying factor analysis we need to follow some extra steps:

#The first step is to scale the data. Since the data types and range of values is different accros the columns (percentages vs
#counts vs averages) this is a necessary step. So we standarized all columns to have a mean value of 0 and a standard deviation
#of 1.

#NOTE: Since the length of the dataset is quite small (30x18 --> only 30 players and  18 columns)

import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(25,15))
correlation_heatmap = sns.heatmap(factor_analysis_input.corr(), cmap='RdBu', annot=True, annot_kws={'size': 18}, linewidths=1)
plt.title('Correlation Heatmap', fontsize=25)

color_bar = correlation_heatmap.collections[0].colorbar
color_bar.ax.tick_params(labelsize=16)

plt.xticks(fontsize=16)
plt.yticks(fontsize=16)

plt.show()

#The second step is to visualize the correlation of variables. This is a very important step since we must avoid high
#colienarity between variables. Inclding variables with high colinearity (above 0.8 correlation) will give biased results
#in the factor analysis.

#Reference link for seaborn correlation heatmap: https://www.geeksforgeeks.org/how-to-create-a-seaborn-correlation-heatmap-in-python/

factor_analysis_input = factor_analysis_input.drop(['Cross %', '% Total Through Balls', 'Avg Vertical Change of Passes',
                                                    'Avg Angle of Passes'], axis=1)

#Looking at the correlation heatmap we can see some variables having high colineairty (above 0.8 correlation). So for every
#pair of variables with high colienarity, we keep one of the columns and remove the other from the dataset.

factor_analysis_input

#References used for coding factor analysis:

#link1: https://www.datacamp.com/tutorial/introduction-factor-analysis
#link2: https://www.datasklr.com/principal-component-analysis-and-factor-analysis/factor-analysis

from factor_analyzer import FactorAnalyzer
from factor_analyzer.factor_analyzer import calculate_kmo, calculate_bartlett_sphericity

KMO_all, KMO_Mdl = calculate_kmo(factor_analysis_input)
Bartletts_chi_square, Bartletts_p_value = calculate_bartlett_sphericity(factor_analysis_input)

print(f'KMO Test Result: {KMO_Mdl}')
print(f"Bartlett's Test Results: Chi-Square = {Bartletts_chi_square}, p-value = {Bartletts_p_value}")

if KMO_Mdl > 0.5 and Bartletts_p_value < 0.01:
    Fac_Analysis = FactorAnalyzer(rotation=None)
    Fac_Analysis.fit(factor_analysis_input)
    print(Fac_Analysis.loadings_)
    print('\n')
    print("The dataset is suitable for Factor Analysis based on KMO & Bartlett's test")
else:
    print("The dataset is not suitable for Factor Analysis based on KMO & Bartlett's test")

#Now we need to do the KMO and Bartlett Test to see if the dataset is suitable for Factor Analysis. In order for the dataset to
#be suitable we test 2 metrics:

#1. KMO > 0.5
#2. Bartlett's p-value < 0.01

#From the results we get we can see that the dataset at its current format is not appropriate for Factor Analysis.

### Checking Communality Scores to find the least relevant columns for the factor analysis

Fac_Analysis = FactorAnalyzer(n_factors = 4, rotation = None)
Fac_Analysis.fit(factor_analysis_input)

Communality_Scores = Fac_Analysis.get_communalities()

Communality_Scores_df = pd.DataFrame({'Column': factor_analysis_input.columns, 'Communality Score': Communality_Scores})

print('Communality Scores by Column:')
print(Communality_Scores_df)

Low_Communalities = Communality_Scores_df.loc[Communality_Scores_df['Communality Score'] < 0.5]
print('\n')
print('Columns With Low Communalities:')
print(Low_Communalities)

#We consider irrelevant columns to be those with a communality score lower than 0.5 . We will remove some of these columns to
#improve the results of the KMO and Bartlett's test

factor_analysis_input_refined = factor_analysis_input.drop(['Launch %', 'Avg Horizontal Change of Passes'], axis=1)

#We remove these features since they have the lowest communality score

from factor_analyzer import FactorAnalyzer
from factor_analyzer.factor_analyzer import calculate_kmo, calculate_bartlett_sphericity

KMO_all, KMO_Mdl = calculate_kmo(factor_analysis_input_refined)
Bartletts_chi_square, Bartletts_p_value = calculate_bartlett_sphericity(factor_analysis_input_refined)

print(f'KMO Test Result: {KMO_Mdl}')
print(f"Bartlett's Test Results: Chi-Square = {Bartletts_chi_square}, p-value = {Bartletts_p_value}")

if KMO_Mdl > 0.5 and Bartletts_p_value < 0.01:
    Fac_Analysis = FactorAnalyzer(rotation=None)
    Fac_Analysis.fit(factor_analysis_input)
```

```
    print(Fac_Analysis.loadings_)
    print('\n')
    print("The dataset is suitable for Factor Analysis based on KMO & Bartlett's test")
else:
    print("The dataset is not suitable for Factor Analysis based on KMO & Bartlett's test")

#Now we need to do the KMO and Bartlett Test to see if the dataset is suitable for Factor Analysis. In order for the dataset to
#be suitable we test 2 metrics:

#1. KMO > 0.5
#2. Bartlett's p-value < 0.01

#From the results we get we can see that the dataset at its current format is not appropriate for Factor Analysis.

factor_analysis_input_refined
#This is the final dataset. This will be used to apply Factor Analysis.

factor_analysis_input_refined.shape
#This is the shape of the final dataset in which we will apply Factor Analysis.

### Applying Factor Analysis

Fac_Analysis_Mdl = FactorAnalyzer(rotation=None)
Fac_Analysis_Mdl.fit(factor_analysis_input_refined)

Eigenvalues, _ = Fac_Analysis_Mdl.get_eigenvalues()

plt.figure(figsize=(8,4))
plt.plot(range(1, len(Eigenvalues) +1), Eigenvalues, 'o-', markersize=6, label='Eigenvalues')
plt.axhline(y=1, color='r', linestyle= '--', label='Eigenvalue = 1')
plt.title('Scree Plot')
plt.xlabel('Number of Factors')
plt.ylabel('Eigenvalue')
plt.grid()
plt.legend()

plt.show()

#One way to determine the optimal number of factors is to create a scree plot. When the eigenvalue gets below 1 as we iterate
#over different number of factors, that is when the optimal number of factors is found.
#The eigenvalue gets below 1 in the 5th factor, which means that we should select 4 factors for our analysis.

factor_analysis_model = FactorAnalyzer(n_factors=4, rotation='varimax')
factor_analysis_model.fit(factor_analysis_input_refined)

factor_analysis_factor_loadings = factor_analysis_model.loadings_

df_factor_loadings = pd.DataFrame(factor_analysis_factor_loadings, index=factor_analysis_input_refined.columns,
                                  columns=[f'Factor_{i+1}' for i in range(factor_analysis_model.n_factors)])
df_factor_loadings

#We plot the factor scores of each column as a rotated dataframe. Values closer to 0 indicate that the column is not being
#described by the factor.

#Looking at the 4 factors, we decided to name them as followed:
#'Factor_1': 'Creative & Counter Attacking Passing'
#'Factor_2': 'Balanced Passing & Possession Maintenance'
#'Factor_3': 'Aerial Passing & Defensive Play'
#'Factor_4': 'Long-Range Playmaking'

df_factor_loadings.to_csv('factor analysis loadings.csv', index=True)

factor_scores = factor_analysis_model.transform(factor_analysis_input_refined)

df_factor_scores = pd.DataFrame(factor_scores, index=factor_analysis_input_refined.index,
                                columns=[f'Factor_{i+1}' for i in range(factor_analysis_model.n_factors)])

df_factor_scores

#We now find the factor score for each player. The higher the score, the more the player matches to the factor.
#For example A. Lacazette plays more like a creative and counter attacking passer, rather than a long-range playmaker.

Fac_Analysis_Mdl = FactorAnalyzer(rotation=None)
Fac_Analysis_Mdl.fit(factor_analysis_input_refined)

Eigenvalues, Variance_Explained_by_each_factor = Fac_Analysis_Mdl.get_eigenvalues()

Variance_contributions_for_each_factor = Variance_Explained_by_each_factor[:4]

print('Variance Contributions for Each Factor:')
for factor, variance in enumerate(Variance_contributions_for_each_factor):
    print(f'Factor {factor+1}: {variance}')

print('\n')

Total_Variance_Explained= sum(Variance_contributions_for_each_factor)

Factor_Weights = [variance / Total_Variance_Explained for variance in Variance_contributions_for_each_factor]

print('Weights for Each Factor Based on Variance Contributions:')
for factor, factor_weight in enumerate(Factor_Weights):
    print(f'Factor {factor+1}: {factor_weight:.4f}')

#To assess the total factor score for player evaluation, we use the factor weighting method.

total_scores = df_factor_scores.dot(Factor_Weights)
df_factor_scores['Total Score'] = total_scores

#For each player: Total Factor Score = W1*F1 Score + W2*F2 Score + W3*F3 Score + W4*F4 Score

df_factor_scores.rename(columns={'Factor_1': 'Creative & Counter Attacking Passing',
                                 'Factor_2': 'Balanced Passing & Possession Maintenance',
                                'Factor_3': 'Aerial Passing & Defensive Play',
                                 'Factor_4': 'Long-Range Playmaking'}, inplace=True)

df_factor_scores = df_factor_scores.sort_values('Total Score', ascending=False)

df_factor_scores

#In the factor scores by player dataframe, we rename the factors and also add context to each player based on their position.
```

```
df_factor_scores['Position'] = ['Forward', 'Midfielder', 'Midfielder', 'Forward', 'Midfielder', 'Midfielder', 'Midfielder',
                                'Midfielder', 'Midfielder', 'Forward', 'Midfielder', 'Defender', 'Defender', 'Forward',
                                'Forward', 'Defender', 'Defender', 'Defender', 'Defender', 'Defender', 'Defender',
                                'Midfielder', 'Forward', 'Midfielder']

df_factor_scores

from sklearn.preprocessing import MinMaxScaler
Scaler = MinMaxScaler(feature_range=(0.01,1)) #We dont wan't any values to be 0 so we scale the values from 0.01-1.

df_factor_scores_normalized = df_factor_scores.iloc[:,:-2].copy()

for column in df_factor_scores_normalized.columns:
    df_factor_scores_normalized[[column]] = Scaler.fit_transform(df_factor_scores_normalized[[column]])

df_factor_scores_normalized = df_factor_scores_normalized.round(2)
df_factor_scores_normalized

#Before creating the radar charts that show the factor score of each player for each factor, we scale the data to a minimum of
#0.01 and a maximum of 1. This makes it easier for us to interpret the radar charts.

#Reference link for MinMaxScaler function: https://stackoverflow.com/questions/24645153/pandas-dataframe-columns-scaling-with-sklearn

def create_radar_chart(ax, data, labels, title):
    number_variables = len(labels) #We get the number of factors which we will plot

    angles_of_chart = np.linspace(0, 2 * np.pi, number_variables, endpoint=False).tolist()
    angles_of_chart +=angles_of_chart[:1] #We compute the angle for each axis in the radar chart (in radians)

    ax.set_theta_offset(np.pi / 2) #We position the first axis on the top
    ax.set_theta_direction(-1) #We set the direction of the radar chart to be clockwise
    ax.set_xticks(angles_of_chart[:-1])
    ax.set_xticklabels(labels)
    ax.set_ylim(0,1)
    #We set the labels for each axis of the radar chart. We set the limits for the axis to be 0-1 since we normalized the
    #factor scores of each player.

    data_for_charts = np.append(data, data[0])
    ax.plot(angles_of_chart, data_for_charts, linewidth=1, linestyle='solid') #We plot the data points and connect them with
    #a solid linestyle
    ax.fill(angles_of_chart, data_for_charts, 'b', alpha=0.1) #We fill the area between the lines
    ax.set_title(title, size=14, color='blue', y=1.1) #We set the title of each radar chart to be the name of the player that is
    #represented by the chart

labels = ['Creative & Counter Attacking Passing', 'Balanced Passing \n& Possession\n Maintenance',
          'Aerial Passing & Defensive Play', 'Long-Range\nPlaymaking'] #These are the 4 factors that we extracted above

figure, axes = plt.subplots(nrows=9, ncols=3, figsize=(15,35), subplot_kw=dict(polar=True))

axes = axes.flatten()

for idx, (player, ax) in enumerate(zip(df_factor_scores_normalized.index, axes)):
    each_player_data = df_factor_scores_normalized.loc[player].values
    create_radar_chart(ax, each_player_data, labels, player)
    #We loop through each player and create their radar chart by extracting their normalize factor scores

total_number_of_radar_charts = len(axes)
total_number_of_players = len(df_factor_scores_normalized.index)

if total_number_of_players < total_number_of_radar_charts:
    for j in range(total_number_of_players, total_number_of_radar_charts):
        axes[j].set_visible(False)
        #Since we will be creating radar charts for 25 players and we want to create 9x3=27 subplots we need to remove the last
        #two subplots which will be empty

plt.tight_layout()
plt.show()

#We plot the radar charts which show us an all-around distribution of how each player matches each factor. For example,
#M. Ozil seems to be more of a creative and counter attacking passer, rather than a player who is focused on aerial passing and
#defensive play.

#Reference link for coding the radar charts: https://medium.com/@reinapeh/creating-a-complex-radar-chart-with-python-31c5cc4b3c5c

plt.figure(figsize=(12,6))
sns.barplot(data=df_factor_scores.reset_index(), y='index', x='Total Score', hue='Position', dodge=False)

plt.title('Total Factor Scores of Each Player', fontsize=14)
plt.xlabel('Total Factor Score', fontsize=11)
plt.ylabel('')
plt.legend(fontsize=12)
plt.xticks(fontsize=11)

plt.show()

#In this bar chart, we can see the total factor scores of each player.
#Each player position is defined by a unique color.

players = df_factor_scores_normalized.index.tolist()

from sklearn.metrics import euclidean_distances
dist_matrix = euclidean_distances(df_factor_scores_normalized, df_factor_scores_normalized)
dist_matrix_df = pd.DataFrame(dist_matrix, index=players, columns=players)

dist_matrix_df

#To analyze better the similarities and dissimilarities of players in the team's passing strategy we are going to calculate the
#euclidean distance of the factor scores of players and apply Multi-Dimensional Scaling (MDS) to scale the similarities between
#all players into 2 dimensions and visualize them using a scatterplot.

from sklearn.manifold import MDS

MDS_Model = MDS(n_components = 2 , random_state=40)
MDS_Result = MDS_Model.fit_transform(dist_matrix_df) #We want the euclidean distances of players to be reduced to just 2
#2 dimensions that will represent their similarities or dissimilarities with other players. We reduce to 2 dimensions to plot
#them in a scatter plot
MDS_df = pd.DataFrame(MDS_Result,columns = ['Dimension 1' , 'Dimension 2'])
MDS_df.index = players

MDS_df['Position'] = ['Forward', 'Midfielder', 'Midfielder', 'Forward', 'Midfielder', 'Midfielder', 'Midfielder',
                      'Midfielder', 'Midfielder', 'Forward', 'Midfielder', 'Defender', 'Defender', 'Forward',
```

```
                                'Forward', 'Defender', 'Defender', 'Defender', 'Defender', 'Defender', 'Defender', 'Defender',
                                'Midfielder', 'Forward', 'Midfielder']

MDS_df = MDS_df.reset_index()

MDS_df

#We apply MDS to the euclidean distances between the players and set the number of components/dimensions to 2.
#We also add a feature that describes the position of the players to add more context to the graph.

plt.figure(figsize=(22,15))
sns.scatterplot(x='Dimension 1', y='Dimension 2', hue='Position', palette='tab10', data=MDS_df, alpha=0.8, s=1200)

for row in range(len(MDS_df)):
    player = players[row]
    plt.text(MDS_Result[row,0] + 0.02, MDS_Result[row,1] + 0.02, player, fontsize=17)

plt.title('MDS Visualization of the Similarities of Players Baed on Their Passing Factor Scores', fontsize=23)
plt.xlabel('Dimension 1', fontsize=20)
plt.xticks(fontsize=17)
plt.yticks(fontsize=17)
plt.ylabel('Dimension 2', fontsize=20)
plt.legend(loc='upper right', bbox_to_anchor=(1.2, 1), fontsize=20, markerscale=3)

plt.grid()
plt.show()

#We set the color of the points to be different by each player position. We can see that the positions of players don't tend to
#overlap with a few exceptions which means that most players with similar positions tend to have similar factor scores and
#playing styles. We can also see that the players are spread across the graph indicating that players in each position tend to
#have their own unique roles.

#One thing we noticed when also comparing this graph to the radar charts of players and the total factor score plot is that the
#players more to the bottom left are players that have low factor scores in creative and counter attacking passing but do show
#decent contribution in the other 3 factors. This area of the graph seems to be mainly occupied by defenders. These defenders
#seem to be used as long-range playmakers who have a big involvement in build-up play.

#Players on the far top of the graph are players that have good scores in creative and counter attacking passing as well as
#in aerial passing and defensive play but are not associated with long-range playmaking or possession maintenance. These players
#show great imbalance between the factors and are players that do not have a major effect in possession. This area of the graph
#is occupied by forwards. Looking at the individual heatmaps of these players, they tend to occupy wide spaces, indicating that
#wide players do not really contribute in the ball possession and are more focused on counter attacking football and in creating
#chances from the final third.

#Players on the middle and lower part of the graph, seem to be the most versatile players on the team when it comes to the
#team's passing strategy. We can understand this since most of these players seem to have really high overall scores of these
#factors. This area of the graph is mainly occupied by central midfielders.

#Players that cover the far left area of the graph seem to have high scores in aerial passing and defensive play as well as
#possession maintenance but are not really used for starting attacks. Looking at the heatmaps of these players, they are mainly
#defenders who play as wing-backs.

#A player who seems to have a very unique playing style is A. Sanchez who is in the far-right of the graph. Looking at the
#positional changes of the player and his heatmap, he tends to cover the left flank of the attack with some tendencies to move
#centrally mainly in the final-third of the pitch. He seems to be mainly associated with creative and counter-attacking passing,
#long-range playmaking and aerial passing and defensive play but does not seem to get really involved in the possession
#maintenance. His role is crucial in both attacking transitions and defensive situations, making him a valuable asset in
#fast-paced and high-risk scenarios. However, his lesser involvement in possession maintenance means he mainly relies on other
#teammates to retain control and distribute the ball more conservatively.

#Overall, after analyzing the similarities of players and also looking at the heatmaps of these players, we notice that players
#who show similarities are also players that have similar positions and move in the same areas of the pitch. Defenders and
#Attackers seem to be way less versatile in the team's passing strategy than the midfielders and show more focus on certain
#aspects of possessions. This indicates that Arsenal tend to play position-based football rather than total football, where each
#player based on their position has certain duties when it comes to the team's passing strategy rather than them being
#positionally flexible.

#Reference link for MDS: IN3061/INM430 Principles of Data Science (PRD1 A 2023/24) - Week 07: (High dimensional and) finding
#structure in data
```

In [ ]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

passes_df = pd.read_csv('arsenal_possessions_with_passes.csv')

passes_df = passes_df.loc[passes_df['duration']!=0]

passes_df.head()
#We load the dataset where the passing sequences have been grouped into possession episodes.

import re

def converting_string_to_list(df, column):
    def processing_element(element):
        if isinstance(element, str):
            element = element.strip("[]")
            element = re.sub(r"['\"]", "", element)
            element = [el.strip() for el in element.split(',') if el.strip()]
            return element
        return element

    df[column] = df[column].apply(lambda y: processing_element(y))
    return df

def cleaning_values_correctly(df, column):
    def processing_element(element):
        if isinstance(element, str):
            element = element.strip("[]")
            element = re.sub(r"['\"]", "", element)
            element = [el.strip() for el in element.split(',') if el.strip()]
            return element
        return element

    df[column] = df[column].apply(lambda y: processing_element(y))
    return df

#We noticed that the columns that have lists of values have the lists inside strings. So we want to transform the strings into
#lists with elements
```

```
passes_df = cleaning_values_correctly(passes_df, 'player_names')
passes_df = cleaning_values_correctly(passes_df, 'events')
passes_df = cleaning_values_correctly(passes_df, 'sub_events')
passes_df = cleaning_values_correctly(passes_df, 'tags_description')
passes_df = cleaning_values_correctly(passes_df, 'eventsId')
passes_df = cleaning_values_correctly(passes_df, 'sub_eventsId')
passes_df = cleaning_values_correctly(passes_df, 'position')
passes_df = cleaning_values_correctly(passes_df, 'start_coords_x')
passes_df = cleaning_values_correctly(passes_df, 'start_coords_y')
passes_df = cleaning_values_correctly(passes_df, 'end_coords_x')
passes_df = cleaning_values_correctly(passes_df, 'end_coords_y')
passes_df = cleaning_values_correctly(passes_df, 'event_length')
passes_df = cleaning_values_correctly(passes_df, 'event_angle')
passes_df = cleaning_values_correctly(passes_df, 'vertical_change')
passes_df = cleaning_values_correctly(passes_df, 'horizontal_change')
passes_df = cleaning_values_correctly(passes_df, 'players')

#This function replaces the value 'nan' with the actual player Id. Some players do not have names so we replace their names
#with the Ids. When creating the interactions dataset we will need player interactions even between players who we don't know
#their names
def replacing_nan_values_with_player_ids(df):
    def replace_nan_values(player_names, player_ids):
        return [player_name if player_name != 'nan' else str(player_id) for player_name, player_id in zip(player_names, player_ids)]

    df['player_names'] = df.apply(lambda row: replace_nan_values(row['player_names'], row['players']), axis=1)
    return df

passes_df = replacing_nan_values_with_player_ids(passes_df)

import ast

passes_df_modified = passes_df[['duration', 'num_events', 'sub_events', 'position',
                                'event_length', 'event_angle', 'vertical_change',
                                'horizontal_change', 'outcome', 'month', 'result', 'home_game', 'current_score_difference',
                               'opponent_strength', 'time_segment (min)', 'starting_position_x', 'ending_position_x',
                                'starting_position_y', 'ending_position_y', 'horizontal_amplitude', 'vertical_amplitude']]


passes_df_modified

#We extract the relevant features that we will be using to cluster the possession episodes. We will be using features that are
#related to the duration and number of events, the type of sub events in the possession episodes, the position of players that
#are involved in each possession, some summary statistics of the events like the length and angle of each event, and we will
#also use features that add contextual information to the possessions like home vs away games, or winning vs losing games.

def frequency_counts_of_unique_values(df, column, lst):
    for value in lst:
        df[f'{value}'] = 0

    for index, row in df.iterrows():
        value_count = {value: row[column].count(value) for value in lst}

        for value, count in value_count.items():
            df.at[index, f'{value}'] = count
    return df

#We first need to preprocess the data before using the dimensionality reduction and clustering algorithms. Since many of the
#features that have categorical data in the passes_df_modified dataset have a list of values in each row, one way we can
#preprocess the data is to perform frequency counts. So for each unique value of a feature, we will count how many times that
#unique value occurs in each row.

events_df = pd.read_csv('events_arsenal.csv')

sub_events_lst = events_df['subEventName'].unique().tolist()
passes_df_modified = frequency_counts_of_unique_values(passes_df_modified, 'sub_events', sub_events_lst)

#For each possession (row) we count how many times each unique sub event occurs.

passes_df_modified = passes_df_modified.drop(['Air duel', 'Ground loose ball duel', 'Ground defending duel',
                                              'Ground attacking duel', 'Foul', 'Free Kick', 'Shot', 'Reflexes', 'Touch',
                                              'Clearance', 'Ball out of the field', 'Throw in', 'Goal kick', 'Corner',
                                              'Goalkeeper leaving line', 'Acceleration', 'Save attempt', 'Free kick cross',
                                              'Hand foul', 'Late card foul', 'Free kick shot', 'Whistle', 'Protest', 'Penalty',
                                              'Out of game foul', 'Time lost foul', 'Simulation', 'nan', 'sub_events'], axis=1)

#We only care about the sub events related to passes as we aim to find different passing strategies (clusters).

positions_lst = ['Goalkeeper', 'Defender', 'Midfielder', 'Forward', 'nan']

passes_df_modified = frequency_counts_of_unique_values(passes_df_modified, 'position', positions_lst)

passes_df_modified = passes_df_modified.drop(['nan', 'position'], axis=1)

#For each possession (row) we count how many times each player position occurs. Since there is a large number of unique players
#at Arsenal, it is more suitable instead to count the players by positions and not each player individually, to avoid making the
#dataset too large.

def calculate_mean_values_of_lists (df, column):
    df[column] = df[column].apply(lambda lst: [float(value) for value in lst])
    df[f'avg_{column}'] = df[column].apply(lambda lst: sum(lst) / len(lst))

    return df

#The best way to preprocess the features that have a list of numerical values, is to calculate the averages. This function takes
#a list of values in each row and caluclates the average.

passes_df_modified = calculate_mean_values_of_lists(passes_df_modified, 'event_length')
passes_df_modified = calculate_mean_values_of_lists(passes_df_modified, 'event_angle')
passes_df_modified = calculate_mean_values_of_lists(passes_df_modified, 'vertical_change')
passes_df_modified = calculate_mean_values_of_lists(passes_df_modified, 'horizontal_change')

#We calculate the average length, angle, vertical change, and horizontal change of all events in each possession.

passes_df_modified = passes_df_modified.drop(['event_length', 'event_angle', 'vertical_change', 'horizontal_change'], axis=1)

outcomes = dict({'Pass':1, 'Duel':2, 'Others on the ball':3,
                 'Shot': 4, 'Interruption':5, 'Offside':6, 'Foul':7})

                 tcomes(x):
```

```
    return outcomes[x]

passes_df_modified['outcome'] = passes_df_modified['outcome'].apply(encode_outcomes)

#The next step is to encode the categorical variable which describes the outcome of the pass (last event of the event sequence
#in each possession)

opponent_strengths = dict({'Top 10 Team': 1, 'Bottom 10 Team': 2})

def encode_opponent_strength(y):
    return opponent_strengths[y]

passes_df_modified['opponent_strength'] = passes_df_modified['opponent_strength'].apply(encode_opponent_strength)

#We also encode the opponent strengths which are split into Top 10 and Bottom 10 Teams (based on their final position in that
#year).

time_segments = dict({'0-30': 1, '30-60': 2, '60-90+': 3})

def encode_time_segments(z):
    return time_segments[z]

passes_df_modified['time_segment (min)'] = passes_df_modified['time_segment (min)'].apply(encode_time_segments)

#We also encode the time segments which are split into possessions that happened the first 30 minutes, 30-60 minutes, and final
#30 minutes

def grouping_current_score(x):
    if x>=1:
        return 'Winning'
    elif x==0:
        return 'Tied'
    else:
        return 'Losing'

passes_df_modified['current_score_grouped'] = passes_df_modified['current_score_difference'].apply(grouping_current_score)

passes_df_modified['duration'] = passes_df_modified['duration'].round(2)
passes_df_modified['avg_event_length'] = passes_df_modified['avg_event_length'].round(2)
passes_df_modified['avg_event_angle'] = passes_df_modified['avg_event_angle'].round(2)
passes_df_modified['avg_vertical_change'] = passes_df_modified['avg_vertical_change'].round(2)
passes_df_modified['avg_horizontal_change'] = passes_df_modified['avg_horizontal_change'].round(2)

#We round the decimals of certain variables.

passes_df_modified
#This is how the preprocessed dataset looks after the main transformations.

passes_df_categorical = passes_df_modified[['result', 'home_game', 'opponent_strength', 'time_segment (min)',
                                            'current_score_grouped', 'outcome']]

#We split the features into categorical  and numerical features.

passes_df_numerical = passes_df_modified.drop(['result', 'home_game', 'opponent_strength', 'time_segment (min)',
                                               'outcome', 'current_score_grouped', 'current_score_difference', 'month'], axis=1)

#We will use the numerical features as input for the dimensionality reduction algorithm.

passes_df_numerical

#This is the dataframe that we will use as the input for dimensionality reduction. All values are numerical and describe the
#event characteristics, the types of passes in each possession, and the player positions involved in each possession.

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

passes_df_numerical_columns = passes_df_numerical.columns.tolist()
passes_df_numerical_scaled = pd.DataFrame(scaler.fit_transform(passes_df_numerical), columns=passes_df_numerical_columns)

passes_df_numerical_scaled = passes_df_numerical_scaled.round(2)

passes_df_numerical_scaled

#We scale the variables because they have different range of values. This is a useful step for non-biased clustering.

#Reference link for StandardScaler function: https://www.digitalocean.com/community/tutorials/standardscaler-function-in-python

from sklearn.decomposition import PCA

Principal_Component_Analysis = PCA()
Principal_Component_Analysis.fit(passes_df_numerical_scaled)

Cumulative_Variance_Explained = np.cumsum(Principal_Component_Analysis.explained_variance_ratio_)

plt.figure(figsize=(10,6))
plt.plot(Cumulative_Variance_Explained, marker='x')
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Variance Explained')
plt.title('Cumulative Variance Explained by PCA Number of Components')
plt.grid(True)

plt.show()

#Having many variables makes clustering less effective. In our case we have 23 features so we need to scale the dimensions
#using PCA. Our aim is to retain around 90-95% of the cumulative variance explained by the PCA components. Looking at the
#plot, 15 components are able to extract just over 95%. So overall, by applying PCA we were able to reduce the number of
#components by 1/3 while still retaining 95% of the feature variance. This will make it for our model to categorize the
#possessions into clusters.

#Reference link for calculating the Cumulative Variance Explained using the PCA function: https://vitalflux.com/pca-explained-variance-concept-python-example/

Principal_Component_Analysis_15_Components = PCA(n_components=15)

Princ_Compon_Anal_df = Principal_Component_Analysis_15_Components.fit_transform(passes_df_numerical_scaled)
Princ_Compon_Anal_df = pd.DataFrame(Princ_Compon_Anal_df, columns=['PC1', 'PC2', 'PC3', 'PC4', 'PC5', 'PC6', 'PC7', 'PC8', 'PC9'
                                                                   , 'PC10', 'PC11', 'PC12', 'PC13', 'PC14', 'PC15'])
Princ_Compon_Anal_df

#We apply PCA dimension scaling to scale the data from 23 components to just 15. We are going to apply clustering the PCA
```

```
from sklearn.neighbors import NearestNeighbors

Minimum_Pts = 5 #We want our clusters to have at least 5 possessions for more significant clusters. We don't want to specify a
#higher number of possessions to form a significant cluster since football possessions have very different statistics and it is
#quite difficult to find possessions with exactly similar characteristics.
K_Value = Minimum_Pts - 1 #Check the distances of the 4th value to define Epsilon value.

nearest_neighbors = NearestNeighbors(n_neighbors=K_Value)
nearest_neighbors.fit(Princ_Compon_Anal_df)
distance, indices = nearest_neighbors.kneighbors(Princ_Compon_Anal_df)
K_Distance = distance[:,-1]
Sorted_K_Distance = np.sort(K_Distance)

plt.figure(figsize=(10,6))
plt.plot(Sorted_K_Distance)
plt.title('K-Distance Graph')
plt.xlabel('Points (Sorted by Distance)')
plt.ylabel(f'Distance to {K_Value}-th Nearest Neighbor')
plt.grid(True)

plt.show()
#This graph is the K-Distance graph which helps us specify the Epsilon value. The Epsilon value is defined by the maximum
#distance 2 points need to have to be considered in the same cluster.

#In this plot we can see the distance of each data point to its 4th nearest neighbor. The elbow in the cluster shows us for
#which epsilon values the points start to be too far away from their neighbors, indicating that these points do not belong to
#a cluster and should be identified as noise. The steep increase starts beginning from around 2.2, but we are going to round and
#choose an Epsilon value of 2 because we want more compact clusters.

#Reference link for plotting the K-distance Graph: https://medium.com/@tarammullin/dbscan-parameter-estimation-ff8330e3a3bd

#References used for coding the DBSCAN algorithm:

#link1: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html
#link2: https://www.kdnuggets.com/2022/08/implementing-dbscan-python.html
#link3: Visual Analytics Module: Lab 04- DB Clustering

from sklearn.cluster import DBSCAN

np.random.seed(50) #We set this for reproducibility
Epsilon_Value = 2
Minimum_Sample = 5

Dbscan_Algorithm = DBSCAN(eps=Epsilon_Value, min_samples = Minimum_Sample)
Clusters = Dbscan_Algorithm.fit_predict(Princ_Compon_Anal_df)

passes_df_numerical['Cluster'] = Clusters
passes_df_numerical['Cluster'].value_counts()

#We apply DBSCAN with Epsilon=2 and Minimum Data Points=5. We can see that the possessions have been split into multiple
#clusters while 209 are considered occasional. Most possessions seem to be added to the same cluster showing that most Arsenal possessions
#follow a similar trend.

passes_df_numerical

passes_df_numerical['Cluster'].value_counts()

#We can see that a total of 28 clusters were formed while just over 30% of possessions were treated as noise

cluster_means = passes_df_numerical.groupby('Cluster')['duration', 'num_events', 'starting_position_x', 'starting_position_y',
                                                       'ending_position_x', 'ending_position_y', 'horizontal_amplitude',
                                                       'vertical_amplitude', 'avg_event_length', 'avg_event_angle',
                                                       'avg_vertical_change', 'avg_horizontal_change'].mean()
#We calculate the mean values of these features for each possession cluster.

cluster_sums = passes_df_numerical.groupby('Cluster')['Simple pass', 'High pass', 'Smart pass', 'Launch', 'Cross', 'Hand pass',
                                'Goalkeeper', 'Defender', 'Midfielder', 'Forward'].sum()

clusters_df = pd.concat([cluster_means, cluster_sums], axis=1)

clusters_df['Total_Passes'] = clusters_df[['Simple pass', 'High pass', 'Smart pass',
                                           'Launch', 'Cross', 'Hand pass']].sum(axis=1)

pass_types = ['Simple pass', 'High pass', 'Smart pass', 'Launch', 'Cross', 'Hand pass']

for pass_type in pass_types:
    clusters_df[f' {pass_type} %'] = (clusters_df[pass_type] / clusters_df['Total_Passes']) * 100
#We Calculate the percentage of each pass type used in each possession cluster.

clusters_df['Total_Positions'] = clusters_df[['Goalkeeper', 'Defender', 'Midfielder', 'Forward']].sum(axis=1)

positions = ['Goalkeeper', 'Defender', 'Midfielder', 'Forward']

for pos in positions:
    clusters_df[f' {pos} %'] = (clusters_df[pos] / clusters_df['Total_Positions']) * 100
#We calculate the percentage of each player position invloved in each possession cluster.

clusters_df = clusters_df.drop(['Simple pass', 'High pass', 'Smart pass', 'Launch', 'Cross', 'Hand pass',
                                'Goalkeeper', 'Defender', 'Midfielder', 'Forward','Total_Positions', 'Total_Passes'], axis=1)

clusters_df = clusters_df.round(2)

clusters_df.fillna(0, inplace=True)

clusters_df

#Mean Values: 'duration', 'num_events', 'starting_position_x', 'starting_position_y', 'ending_position_x', 'ending_position_y',
#'horizontal_amplitude', 'vertical_amplitude', 'avg_event_length', 'avg_event_angle', 'avg_vertical_change',
#'avg_horizontal_change'

#Percentages: 'Simple pass', 'High pass', 'Smart pass', 'Launch', 'Cross', 'Hand pass'

#Percentages: 'Goalkeeper', 'Defender', 'Midfielder', 'Forward'

#For each cluster separately, we calculate the percentages of each player position and pass type separately because the count of
#possessions is not equal for each cluster. To make a more fair, non-biased comparison, we aim to understand the distribution
#of different features for each possession cluster. This will help us evaluate how the clusters are different and characterize
#them.

cluster_counts = passes_df_numerical['Cluster'].value_counts().to_frame('count')
```

```
cluster_sizes = cluster_sizes[cluster_sizes.index != -1]

max_cluster_size = cluster_sizes['count'].max()

import math
def getColor (x, y, minX, maxX, minY, maxY):
    wX=maxX-minX
    wY=maxY-minY
    rr=y-minY
    cc=x-minX
    #print(x,y)
    if (wY < wX):   #scale vertically, i.e. modify rr
        rr *= wX/wY
    else:           #scale horizontally, i.e. modify cc
        cc *= wY/wX
    maxD=max(wX,wY)
    rr1=maxD-rr
    cc1=maxD-cc
    #print(rr,cc,maxD,rr1,cc1)
    dc=[math.sqrt(rr*rr+cc*cc),math.sqrt(rr*rr+cc1*cc1),math.sqrt(rr1*rr1+cc*cc),math.sqrt(rr1*rr1+cc1*cc1)]
    weights=[0.0,0.0,0.0,0.0]
    for i in range(len(weights)):
        weights[i]=(maxD-dc[i])/maxD
        if (weights[i]<0):
            weights[i]=0
    #print(dc,weights)
    reds=[228,25,255,37]
    greens=[220,228,18,13]
    blues=[0,218,6,252]
    dr=0
    dg=0
    db=0
    for i,weight in enumerate(weights):
        dr += weight*reds[i]
        dg += weight*greens[i]
        db += weight*blues[i]
    if (dr<0):
        dr=0;
    if (dr>255):
        dr=255
    if (dg<0):
        dg=0;
    if (dg>255):
        dg=255
    if (db<0):
        db=0;
    if (db>255):
        db=255
    #print(weights,dr,dg,db)
    c_string = '#{:02x}{:02x}{:02x}'.format(int(dr),int(dg),int(db))
    return c_string

#This funtion is going to be used on the Multi-Dimensional Projection of the Clusters to visualize which clusters are more
#similar or more dissimilar

#Reference link for this section of code: Visual Analytics Module: Lab 04- DB Clustering

from sklearn.preprocessing import MinMaxScaler
from sklearn.manifold import MDS

clusters_data = clusters_df.loc[clusters_df.index!=-1]
scaler = MinMaxScaler()
clusters_data_scaled = scaler.fit_transform(clusters_data)

mds_ST = MDS(n_components = 2, random_state=110)
mds_ST.fit(clusters_data_scaled)
xy_mds_ST = mds_ST.fit_transform(clusters_data_scaled)


xmin_ST=xy_mds_ST[:,0].min()
xmax_ST=xy_mds_ST[:,0].max()
ymin_ST=xy_mds_ST[:,1].min()
ymax_ST=xy_mds_ST[:,1].max()
print(xmin_ST,xmax_ST,ymin_ST,ymax_ST)

plt.figure(figsize=(10,10))
plt.xlabel('Axis 1')
plt.ylabel('Axis 2')
plt.title('MDS projection of clusters')
colors = [(0,0,0)]

for i in range(len(xy_mds_ST)): #This code is used to determine the size of the cicle.
    j=np.where(cluster_sizes.index==clusters_data.index[i])[0][0] #j is assigned the index corresponding to the i-th
    #element in clusters_data.index within cluster_sizes.index.
    r=cluster_sizes.iat[j,0]/max_cluster_size
    size = 50+300*r
    #This calculates a value r as the ratio of the size of a specific cluster to
    #the maximum cluster size in the dataset. The components of this calculation are as follows:cluster_sizes is assumed to
    #be a DataFrame or Series that stores the sizes of clusters.
    #The .iat method in Pandas is used to access a specific cell within a DataFrame by specifying the row and column
    #positions using integer-based indexing. For example --> DataFrame.iat[row, column]

    plt.scatter(xy_mds_ST[i,0], xy_mds_ST[i,1], alpha = .9, s = size,
                c=getColor(xy_mds_ST[i,0], xy_mds_ST[i,1],xmin_ST,xmax_ST,ymin_ST,ymax_ST))
    plt.text(xy_mds_ST[i,0]+0.0001*size, xy_mds_ST[i,1]+0.0001*size,
             str(clusters_data.index[i])+": "+str(cluster_sizes.iat[j,0]), alpha = .25+.75*r)
#alpha = .25+.75*r --> We set the transparency of the text based on the size of the data point
plt.grid()

#This is a 2D MDS projection of the clusters to help us understand if there are any clusters that show similarities.

#The 2 most dense clusters seem to be somewhat similar while the other clusters with reasonable sizes seem to be a bit different.

#Reference link for this section of code: Visual Analytics Module: Lab 04- DB Clustering

merged_df = clusters_df.join(cluster_sizes, how='inner', lsuffix='_left', rsuffix='_right')

merged_df = merged_df.sort_values('count', ascending=False)
```

```
merged_df.to_csv('Clusters.csv', index=True)

Columns = merged_df.columns.tolist()

categories_events = Columns[0:6]
categories_events_2 = Columns[6:12]
categories_passes = Columns[12:18]
categories_positions = Columns[18:-1]

Scaler = MinMaxScaler() #We normalize the values because the range of values varies across the columns. This will help us
#interpret the heatmaps.

scaled_dataset = scaler.fit_transform(merged_df)
scaled_df = pd.DataFrame(scaled_dataset, columns=Columns, index=merged_df.index)
scaled_df = scaled_df.iloc[:, :-1]

for category, category_name in zip([categories_events, categories_events_2, categories_passes, categories_positions],
                                   ['Spatio-Temporal Characteristics of Passes', 'Passing Characteristics', 'Pass Types',
                                    'Player Positions']):
    plt.figure(figsize=(8,6))
    sns.heatmap(scaled_df[category], annot=False, annot_kws={'size': 18}, linewidths=1, cmap='viridis')
    plt.title(f'Heatmap of {category_name} by Cluster (Sorted by Cluster Size)')
    plt.show()

#We plot the heatmaps to see how different the most repeated possessions are from the occasional.

top_12_clusters_by_size = merged_df.iloc[:12, :]

top_12_clusters_by_size_spatio_temporal = top_12_clusters_by_size.iloc[:12, 0:6]
top_12_clusters_by_size_pass_characteristics = top_12_clusters_by_size.iloc[:12, 6:12]
top_12_clusters_by_size_pass_types = top_12_clusters_by_size.iloc[:12, 12:18]
top_12_clusters_by_size_player_positions = top_12_clusters_by_size.iloc[:12, 18:-1]

#We want to see the main differences of the top 12 clusters by size (size>12) to see how we would describe these clusters.

top_12_clusters_by_size_spatio_temporal.to_csv('Clusters_SpatioTemporal.csv', index=True)
top_12_clusters_by_size_pass_characteristics.to_csv('Clusters_PassCharacteristics.csv', index=True)
top_12_clusters_by_size_pass_types.to_csv('Clusters_PassTypes.csv', index=True)
top_12_clusters_by_size_player_positions.to_csv('Clusters_PlayerPositions.csv', index=True)

spatio_temporal_table = top_12_clusters_by_size_spatio_temporal.style.background_gradient(cmap='YlOrRd', axis=0)

spatio_temporal_table

pass_characteristics = top_12_clusters_by_size_pass_characteristics.style.background_gradient(cmap='YlOrRd', axis=0)

pass_characteristics

pass_types = top_12_clusters_by_size_pass_types.style.background_gradient(cmap='YlOrRd', axis=0)

pass_types

player_positions = top_12_clusters_by_size_player_positions.style.background_gradient(cmap='YlOrRd', axis=0)

player_positions

full_df = pd.concat([passes_df_categorical, passes_df_numerical], axis=1)
#We merge the numerical dataframe with the cluster labels alongside the dataframe that contains contextual information about
#the possessions. We are going to use this to see the contextual game differences between the clusters.

full_df = full_df.loc[full_df['Cluster']!=-1]

indexes_to_keep = [2, 0, 4, 7, 5, 1, 6, 14, 8, 9, 11, 3]

#We filter the DataFrame to keep only the specified clusters
top_12_clusters = full_df[full_df.Cluster.isin(indexes_to_keep)]

top_12_clusters

top_12_clusters.to_csv('Top_12_Clusters.csv', index=True)

clusters_contextual_information_result = top_12_clusters.groupby(['result','Cluster']).count()['Hand pass']
result = pd.DataFrame(clusters_contextual_information_result).rename(columns={'Hand pass': 'Total Possessions'})

clusters_contextual_information_home_game = top_12_clusters.groupby(['home_game','Cluster']).count()['Hand pass']
home_game = pd.DataFrame(clusters_contextual_information_home_game).rename(columns={'Hand pass': 'Total Possessions'})

clusters_contextual_information_time_segment = top_12_clusters.groupby(['time_segment (min)','Cluster']).count()['Hand pass']
time_segment = pd.DataFrame(clusters_contextual_information_time_segment).rename(columns={'Hand pass': 'Total Possessions'})

clusters_contextual_information_opponent_strength = top_12_clusters.groupby(['opponent_strength','Cluster']).count()['Hand pass']
opponent_strength = pd.DataFrame(clusters_contextual_information_opponent_strength).rename(columns={'Hand pass': 'Total Possessions'})

clusters_contextual_information_current_score = top_12_clusters.groupby(['current_score_grouped','Cluster']).count()['Hand pass']
current_score = pd.DataFrame(clusters_contextual_information_current_score).rename(columns={'Hand pass': 'Total Possessions'})

#Now we want to compare the metrics accross different game contexts. In our case, we are going to compare how often each passing
#strategy (cluster) was used in games that Arsenal won, drew or lost, in home and away games, in different time segments, and
#against different opponents (opponent strength). This will not only help us identify in which game contexts each passing
#strategy is being used, but we can also see which is the most effective (looking at how often each passing strategy was used in
#games that Arsenal won, games where they drew, and in games that they lost).

New_Row = pd.DataFrame({'Total Possessions': [0.01]}, index=pd.MultiIndex.from_tuples([(0,9)], names=['result', 'Cluster']))

result = result.append(New_Row)
result = result.sort_index()

New_Row = pd.DataFrame({'Total Possessions': [0.01]}, index=pd.MultiIndex.from_tuples([(1,8)], names=['result', 'Cluster']))

result = result.append(New_Row)
result = result.sort_index()

#We add a small constant 0.01 to the clusters that have 0% of Total Possessions so we can plot the bar charts.

result_total = top_12_clusters.groupby('result').count()[['num_events']]

result = result['Total Possessions'].unstack(level=0, fill_value=0)

normalized_df_result= result.copy()
for column in normalized_df_result.columns:
```

```
    normalized_df_result[column] = normalized_df_result[column] / result_total.loc[column, 'num_events']
    #Since the number of possessions varies across the contexts, and in general more possessions have happened in certain
    #contexts. For example the possessions in home games are slightly more than the possessions of away games which means
    #that the counts are biased towards the home games. But in our case we want to show the distribution of topics in different
    #game contexts so we need to normalize the values. One way to do that is to divide the values in each context for each
    #topic with the total number of possessions for the context. This will help us better understand the distribution of topics
    #in each context and how often they appear.

result_percentages = normalized_df_result.div(normalized_df_result.sum(axis=1), axis=0) * 100

fix,axes = plt.subplots(figsize=(10,6))

result_percentages.plot(kind='barh', stacked=True, ax=axes, color=['blue', 'orange', 'green'])

axes.set_title('Result of Games')
axes.set_xlabel('Percentage of Possessions')
axes.set_ylabel('Cluster')
axes.set_xticks(range(0,101,10))

axes.tick_params(axis='x', labelsize=12)
axes.tick_params(axis='y', labelsize=12)
axes.legend(title='Result', labels=['Games Lost', 'Games Drew', 'Games Won'])

plt.show()

home_vs_away_total = top_12_clusters.groupby('home_game').count()[['num_events']]

home_game = home_game['Total Possessions'].unstack(level=0, fill_value=0)

normalized_df_home_vs_away= home_game.copy()
for column in normalized_df_home_vs_away.columns:
    normalized_df_home_vs_away[column] = normalized_df_home_vs_away[column] / home_vs_away_total.loc[column, 'num_events']

opponent_strength_total = top_12_clusters.groupby('opponent_strength').count()[['num_events']]

opponent_strength = opponent_strength['Total Possessions'].unstack(level=0, fill_value=0)

normalized_df_opponent_strength= opponent_strength.copy()
for column in normalized_df_opponent_strength.columns:
    normalized_df_opponent_strength[column] = normalized_df_opponent_strength[column] / opponent_strength_total.loc[column, 'num_events']

time_segment_total = top_12_clusters.groupby('time_segment (min)').count()[['num_events']]

time_segment = time_segment['Total Possessions'].unstack(level=0, fill_value=0)

normalized_df_time_segment = time_segment.copy()
for column in normalized_df_time_segment.columns:
    normalized_df_time_segment[column] = normalized_df_time_segment[column] / time_segment_total.loc[column, 'num_events']

current_score_total = top_12_clusters.groupby('current_score_grouped').count()[['num_events']]

current_score = current_score['Total Possessions'].unstack(level=0, fill_value=0)

normalized_df_current_score = current_score.copy()
for column in normalized_df_current_score.columns:
    normalized_df_current_score[column] = normalized_df_current_score[column] / current_score_total.loc[column, 'num_events']

fig, axes = plt.subplots(2,2, figsize=(20,15))

def create_stacked_bar_charts(df, ax, title, colors, labels):
    df_percentages_by_context = df.div(df.sum(axis=1), axis=0) * 100 #For each row in the dataframe (each cluster) we transform
    #the counts that represent the total possessions into percentages
    df_percentages_by_context.plot(kind='barh', stacked=True, ax=ax) #We want the bar charts to be stacked so we can
    #simultaneously compare the contexts
    ax.set_title(title, fontsize=17)
    ax.set_xlabel('Percentage of Possessions', fontsize=15)
    ax.tick_params(axis='x', labelsize=14)
    ax.set_ylabel('Cluster', fontsize=15)
    ax.tick_params(axis='y', labelsize=14)
    ax.set_xticks(range(0,101,10))
    ax.legend(labels=labels, loc='upper left', fontsize=14)

create_stacked_bar_charts(normalized_df_home_vs_away, axes[0,0], 'Home vs Away Games', ['blue', 'red'],
                          ['Away Game', 'Home Game'])
create_stacked_bar_charts(normalized_df_opponent_strength, axes[0,1], 'Against Top 10 vs Bottom 10 Teams', ['blue', 'red'],
                          ['Top 10 Teams', 'Bottom 10 Teams'])
create_stacked_bar_charts(normalized_df_time_segment, axes[1,0], 'Time Segment', ['blue', 'red', 'orange'],
                          ['0-30 Mins', '30-60 Mins', '60-90+ Mins'])
create_stacked_bar_charts(normalized_df_current_score, axes[1,1], 'Current Score (during games)', ['blue', 'red', 'orange'],
                          ['Losing', 'Drawing', 'Winning'])

#We create stacked bar charts to compare the occurence of each cluster in different contexts

merged_df = clusters_df.join(cluster_sizes, how='inner', lsuffix='_left', rsuffix='_right')

merged_df = merged_df.sort_values('count', ascending=False)

merged_df

clusters_possessions_leading_to_non_accurate_passes = full_df.loc[full_df['outcome']==1].groupby('Cluster').count()['Hand pass']
non_accurate_passes = pd.DataFrame(clusters_possessions_leading_to_non_accurate_passes).rename(columns={'Hand pass': '% of Possessions Leading to Non Accurate

clusters_possessions_leading_to_duels = full_df.loc[full_df['outcome']==2].groupby('Cluster').count()['Hand pass']
duels = pd.DataFrame(clusters_possessions_leading_to_duels).rename(columns={'Hand pass': '% of Possessions Leading to Duels'})

clusters_possessions_leading_to_others = full_df.loc[full_df['outcome']==3].groupby('Cluster').count()['Hand pass']
others = pd.DataFrame(clusters_possessions_leading_to_others).rename(columns={'Hand pass': '% of Possessions Leading to Others on the Ball'})

clusters_possessions_leading_to_shots = full_df.loc[full_df['outcome']==4].groupby('Cluster').count()['Hand pass']
shots = pd.DataFrame(clusters_possessions_leading_to_shots).rename(columns={'Hand pass': '% of Possessions Leading to Shots'})

clusters_possessions_leading_to_corners = full_df.loc[full_df['outcome']==5].groupby('Cluster').count()['Hand pass']
corners = pd.DataFrame(clusters_possessions_leading_to_corners).rename(columns={'Hand pass': '% of Possessions Leading to Interruptions (Corner Kicks)'})

clusters_possessions_leading_to_offside = full_df.loc[full_df['outcome']==6].groupby('Cluster').count()['Hand pass']
offside = pd.DataFrame(clusters_possessions_leading_to_offside).rename(columns={'Hand pass': '% of Possessions Leading to Offside'})


outcomes_df = pd.concat([non_accurate_passes, duels, others, shots, corners, offside], axis=1)

outcomes_df = outcomes_df.join(cluster_sizes, how='inner', lsuffix='_left', rsuffix='_right')
```

```
outcomes_df = outcomes_df.sort_values('count', ascending=False)

outcomes_df = outcomes_df.fillna(0)

outcomes_df = outcomes_df.drop('count', axis=1)

row_sums = outcomes_df.sum(axis=1)

outcomes_df_perc = outcomes_df.div(row_sums, axis=0) * 100

outcomes_df_perc = outcomes_df_perc.round(2)

outcomes_df_perc = outcomes_df_perc.reset_index()

outcomes_df_perc = outcomes_df_perc.fillna(0)

outcomes_df_perc = outcomes_df_perc.iloc[:12]

outcomes_df_perc

#Now we want to have a deeper look into the contextual segmentation of each passing strategy. We will be using the outcome
#column to see the main outcomes of each passing strategy. In particular we will be analysing how often each passing strategy
#leads to a shot, a duel, a non accurate pass, an offside, an interruption (corner), a penalty, and others on the ball. We
#calculate the percentages of each outcome accross all 5 clusters to make a more effective comparison. Overall, this will help
#us further identify which passing strategy is more effective and which is not so much.

#The dataframe below shows the distribution of outcomes for each cluster separately. This will help us make a more fair,
#non-biased comparison because the number of possessions vary accross the clusters.
#The conversion of counts to possessions has been completed horizontally (for each index/cluster).

figure, axes = plt.subplots(nrows = 2, ncols = 3, figsize=(30,15))

figure.subplots_adjust(hspace=0.5, wspace=0.4)

axes = axes.flatten()
columns_by_cluster = outcomes_df_perc.columns[1:]

for ax, column_by_cluster in zip (axes, columns_by_cluster):
    ax.bar(outcomes_df_perc['index'].astype(str), outcomes_df_perc[column_by_cluster], color='orange', width=0.6)
    ax.set_title(f'{column_by_cluster} \n(Top 12 Clusters by Size)', fontsize=20)
    ax.set_xticks(range(len(outcomes_df_perc['index'])))
    ax.set_xticklabels([f'Cluster {cluster_index}' for cluster_index in outcomes_df_perc['index']], rotation=90)
    ax.tick_params(axis='x', labelsize=20)
    ax.tick_params(axis='y', labelsize=18)

plt.show()

#Each bar chart shows the percentage of each outcome for each cluster separately. These graphs make our analysis less biased
#because comparing counts makes our analysis biased since cluster 2 has way more possessions than the other clusters.
#These bar graphs show us the distribution of outcomes for each possession cluster. This will help us understand what the main
#outcomes are for each cluster.

#This is a more fair way to compare clusters without having to worry about favoring clusters with a higher number of
#possessions.
```

In [ ]:

```
# Defining and Grouping the Possession Episodes With NMF Topic Modeling

### Data Preprocessing - Filtering the Dataset

import pandas as pd
import numpy as np

df = pd.read_csv('arsenal_possessions_with_passes.csv')

df = df.loc[df['num_events']>1] #We keep possessions with a sequence of events (more than 1 event)

df['gameweek'] = df.groupby('matchId').ngroup() + 1
#We add a new gameweek when the matchId changes.

df = df.drop(['matchId', 'teamId', 'matchPeriod', 'start_time', 'end_time', 'events', 'eventsId', 'sub_eventsId',
              'players', 'start_coords_x', 'start_coords_y', 'end_coords_x', 'end_coords_y', 'vertical_change',
              'horizontal_change', 'player_names'], axis=1) #We drop the irrelevant columns which we will not use for the topic
               #modeling.
df

### Data Preprocessing - Encoding/Categorizing the Features and Transforming Numerical Values to Texts

def encode_duration(x):
    if x <= 7.5:
        return 'Short_Duration'
    elif x <= 15:
        return 'Medium_Duration'
    else:
        return 'Long_Duration'

df['duration'] = df['duration'].apply(encode_duration) #We categorize the duration of possessions

def encode_num_events(x):
    if x <= 4:
        return 'Low_Activity'
    elif x <= 10:
        return 'Medium_Activity'
    else:
        return 'High_Activity'

df['num_events'] = df['num_events'].apply(encode_num_events) #We categorize the possessions based on the number of events that
#occured

def encode_time_segment(x):
    if x =='0-30':
        return 'First 30 Minutes'
    elif x == '30-60':
        return 'Second 30 Minutes'
    else:
        return 'Last 30 Minutes'

df['time_segment (min)'] = df['time_segment (min)'].apply(encode_time_segment) #We encode the time segments

def encode_outcome(x):
```

```
    if x == 'Others on the ball':
        return 'Outcome_Others_on_the_ball'
    else:
        return f'Outcome_{x}'

df['outcome'] = df['outcome'].apply(encode_outcome) #We modify the outcome variable

def encode_month(x):
    if x == 1:
        return 'January'
    elif x == 2:
        return 'February'
    elif x == 3:
        return 'March'
    elif x == 4:
        return 'April'
    elif x == 5:
        return 'May'
    elif x == 8:
        return 'August'
    elif x == 9:
        return 'September'
    elif x == 10:
        return 'October'
    elif x == 1:
        return 'November'
    else:
        return 'December'

df['month'] = df['month'].apply(encode_month) #We encode the month variable

def encode_result(x):
    if x == 0:
        return 'Loss'
    elif x == 1:
        return 'Draw'
    else:
        return 'Win'

df['result'] = df['result'].apply(encode_result) #We encode the result of the game

def encode_home_game(x):
    if x == 1:
        return 'Home Game'
    else:
        return 'Away Game'

df['home_game'] = df['home_game'].apply(encode_home_game) #We encode the home and away games feature

def encode_current_score(x):
    if x>=1:
        return 'Winning'
    elif x==0:
        return 'Tied'
    else:
        return 'Losing'

df['current_score_difference'] = df['current_score_difference'].apply(encode_current_score) #We encode the score difference
#feature

import re

def converting_string_to_list(df, column):
    def processing_element(element):
        if isinstance(element, str):
            element = element.strip("[]")
            element = re.sub(r"['\"]", "", element)
            element = [el.strip() for el in element.split(',') if el.strip()]
            return element
        return element

    df[column] = df[column].apply(lambda y: processing_element(y))
    return df

def cleaning_values_correctly(df, column):
    def processing_element(element):
        if isinstance(element, str):
            element = element.strip("[]")
            element = re.sub(r"['\"]", "", element)
            element = [el.strip() for el in element.split(',') if el.strip()]
            return element
        return element

    df[column] = df[column].apply(lambda y: processing_element(y))
    return df

#We noticed that the columns that have lists of values have the lists inside strings. So we want to transform the strings into
#lists with elements


df = cleaning_values_correctly(df, 'horizontal_amplitude')
df = cleaning_values_correctly(df, 'vertical_amplitude')
df = cleaning_values_correctly(df, 'sub_events')
df = cleaning_values_correctly(df, 'tags_description')
df = cleaning_values_correctly(df, 'position')
df = cleaning_values_correctly(df, 'starting_position_x')
df = cleaning_values_correctly(df, 'starting_position_y')
df = cleaning_values_correctly(df, 'ending_position_x')
df = cleaning_values_correctly(df, 'ending_position_y')
df = cleaning_values_correctly(df, 'event_length')
df = cleaning_values_correctly(df, 'event_angle') #This part of the code was also used in other notebooks. This transforms the
#string with a list to a list.

df['sub_events'] = df['sub_events'].apply(lambda events: [event.replace(' ', '_') for event in events])
df['tags_description'] = df['tags_description'].apply(lambda events: [event.replace(' ', '_') for event in events])
#Instead of a space between the words we use a _ to make the further preprocessing easier

def concatenate_lists(lst):
    return ' '.join(lst)

df['sub_events'] = df['sub_events'].apply(concatenate_lists)
```

```
df['tags_description'] = df['tags_description'].apply(concatenate_lists)
df['position'] = df['position'].apply(concatenate_lists) #We transform the list of elements into a whole string so we can further
#preprocess these features

def encode_event_length(event_length):
    if float(event_length) <= 10:
        return 'Short_Pass'
    elif float(event_length) <= 40:
        return 'Medium_Pass'
    else:
        return 'Long_Pass'

df['event_length'] = df['event_length'].apply(lambda event_lengths: [encode_event_length(event_length) for event_length in event_lengths])
#We encode the possessions based on the length of the events

def encode_event_angle(event_angle):
    if -45 <= float(event_angle) <= 45:
        return 'Forward_Pass'
    elif 45 < float(event_angle) < 135 or -135 < float(event_angle) < -45:
        return 'Lateral_Pass'
    else:
        return 'Backward_Pass'

df['event_angle'] = df['event_angle'].apply(lambda event_angles: [encode_event_angle(event_angle) for event_angle in event_angles])
#We encode the possessions based on the angle of the events

def encode_horizontal_amplitude(horizontal_amplitude):
    if horizontal_amplitude <= 42.0:
        return 'Compact_Play'
    elif horizontal_amplitude <= 60.0:
        return 'Balanced_Horizontal_Play'
    else:
        return 'Expansive_Width_Play'

df['horizontal_amplitude'] = df['horizontal_amplitude'].apply(encode_horizontal_amplitude)
#We encode the possessions based on the horizontal amplitude of the events

def encode_vertical_amplitude(vertical_amplitude):
    if vertical_amplitude <= 20.0:
        return 'Central_Play'
    elif vertical_amplitude <= 40.0:
        return 'Balanced_Vertical_Play'
    else:
        return 'Expansive_Vertical_Play'

df['vertical_amplitude'] = df['vertical_amplitude'].apply(encode_vertical_amplitude)
#We encode the possessions based on the vertical amplitude of the events

def encode_start_x(start_x):
    if start_x <= 45:
        return 'Defensive_Third_Start'
    elif start_x <= 75:
        return 'Midfield_Third_Start'
    else:
        return 'Attacking_Third_Start'

df['starting_position_x'] = df['starting_position_x'].apply(encode_start_x)
#We encode the possessions based on the starting position of the X-axis

def encode_start_y(start_y):
    if start_y <= 15:
        return 'Left_Flank_Start'
    elif start_y <= 65:
        return 'Central_Zone_Start'
    else:
        return 'Right_Flank_Start'

df['starting_position_y'] = df['starting_position_y'].apply(encode_start_y)
#We encode the possessions based on the starting position of the Y-axis

def encode_end_x(end_x):
    if end_x <= 45:
        return 'Defensive_Third_End'
    elif end_x <= 75:
        return 'Midfield_Third_End'
    else:
        return 'Attacking_Third_End'

df['ending_position_x'] = df['ending_position_x'].apply(encode_end_x)
#We encode the possessions based on the ending position of the X-axis

def encode_end_y(end_y):
    if end_y <= 15:
        return 'Left_Flank_End'
    elif end_y <= 65:
        return 'Central_Zone_End'
    else:
        return 'Right_Flank_End'

df['ending_position_y'] = df['ending_position_y'].apply(encode_end_y)
#We encode the possessions based on the ending position of the Y-axis

df

#This is how the dataset looks so far. Our aim was to transform the numerical representation of features into textual so we can
#use this information as input for out ropic model.

df_events = df[['duration', 'num_events', 'sub_events', 'tags_description', 'position',
 'event_length', 'event_angle', 'current_score_difference', 'horizontal_amplitude', 'vertical_amplitude',
                'starting_position_x', 'starting_position_y', 'ending_position_x', 'ending_position_y', 'outcome']]

df_contexts = df[['opponent_strength', 'time_segment (min)', 'month', 'result', 'home_game', 'gameweek']]

#We split up the dataset to separate the events of the possessions from the extra contextual information. We are first going to
#use the events of the possessions as input four our topic model and then once we have extracted the topics we are going to see
#how often each topic occurs in all the differen game contexts.

df_contexts
#These are the game contexts of the possessions we will be analyzing and comparing later.

df_events.applymap(lambda value: ' '.join(value) if isinstance(value, list) else str(value))
```

```
df_events['concatenated_text'] = df_events.apply(lambda row: ' '.join(row.values), axis=1)

#We concatenate all of the texts from each column into a single column

df_concatenated_text = df_events[['concatenated_text']]

df_concatenated_text

#The column representing the concatenated version of all texts will be used as the input for our topic model.

df_concatenated_text.to_csv('Topic Modeling.csv', index=False)

#We convert the topic modeling input dataset into a sv file.

### Data Preprocessing - Vectorizing the Values With TF-IDF

from sklearn.feature_extraction.text import TfidfVectorizer

text_vectorizer = TfidfVectorizer(max_features=500, min_df=0.05, max_df=0.9)

X = text_vectorizer.fit_transform(df_concatenated_text['concatenated_text'])

df_vectorizer = pd.DataFrame(X.toarray(), columns=text_vectorizer.get_feature_names_out())

df_vectorizer

#Before applying NNMF topic Modeling we need to vectorize the data. In our case, we prefer to use the TF-IDF (Term Frequency
#Vectorizer) which measures how frequently a term occurs in a document (possession) and across all documents. The way TF-IDF
#works is that it will show lower values for words that appear frequently across documents because they are less informative.
#On the other hand, words that are more unique to a particular document will have higher TF-IDF scores, indicating they are more
#important or characteristic of that document.

#TF-IDF is a good way to weigh down very frequent words that don't really provide any uniqueness to the possessions while
#scaling up the rare ones that will help us more differentiate the topics of possessions. This provides a numerical way of
#taking into account the importance and relevance of different words within a collection of documents (possesssions in our
#case).

#min_df=0.05 --> This parameter is used to remove words that appear very rarely (less than 5% of possessions) which do not
#really provide any information at all and don't help us distinguish the documents.
#max_df=0.9 --> This parameter is used to remove terms that appear too frequently (in more than 90% of documents). This helps
#eliminate common words that may not be useful in distinguishing between documents.

#Reference link for vectorizing the values with the TF-IDF function: https://medium.com/voice-tech-podcast/topic-modelling-using-nmf-2f510d962b6e

df_vectorizer = pd.DataFrame(X.toarray(), columns=text_vectorizer.get_feature_names_out())

df_vectorizer = df_vectorizer.drop(['lost', 'losing', 'tied', 'winning', 'won', 'neutral', 'position', 'nan'], axis=1)

df_vectorizer

#We remove any words that we consider irrelevant for our analysis.

df_vectorizer.to_csv('Vectorized Data for NMF Topic Modeling.csv', index=True)

### Finding the Optimal Number of Components/Topics For Our Topic Model

from scipy import sparse

Sparse_Matrix = sparse.csr_matrix(df_vectorizer.values)

import matplotlib.pyplot as plt
from sklearn.decomposition import NMF

Reconstruction_Errors = [] #We will store the reconstruction error by number of components to a list
range_of_components = range(1,20) #We will test from 1-20 components

for component in range_of_components:
    NMF_Mdl = NMF(n_components=component, init='random', random_state=40)
    Weight_Matrix = NMF_Mdl.fit_transform(Sparse_Matrix) #This is the weight matrix in which each vector represents the vector
    #of each word on the topic - Word-Topic Matrix
    H_Feature_Matrix = NMF_Mdl.components_ #This is the feature matrix in which each column vector represents the corpus of each
    #document on the topic - Topic-Document Matrix.
    Reconstructed_Dataset = Weight_Matrix @ H_Feature_Matrix
    Reconstruction_Error = np.linalg.norm(Sparse_Matrix - Reconstructed_Dataset, 'fro') #The reconstruction error shows how close
    #the reconstruction of the dataset W @ H is to the original sparse matrix. We use the Frobenius norm to measure the
    #differences between the original/sparse matrix and the reconstructed dataset.
    Reconstruction_Errors.append(Reconstruction_Error)
    #NMF is a way to decompose the document-word matrix into two matrices that represent the word-topic and topic-document
    #relationships and uses the reconstruction error to find the word_topic * topic_document (W * H) that is closes to the
    #original matrix.

plt.figure(figsize=(8,4))
plt.plot(range_of_components, Reconstruction_Errors, marker='o')
plt.title('NMF Reconstruction Error for Each Number of Componenets/Topics')
plt.xlabel('Number of Components/Topics')
plt.ylabel('Reconstruction Error')
plt.grid(True)

plt.show()

#Before applying topic modeling we need to find the optimal number of components. One way to do that is to calculate the
#reconstruction error for each number of components in a range we specify. The elbow point of the graph is considered to be
#the optimal number of components.

#When we tried this with the CountVectorizer method the reconstruction error was a lot larger so we stick with the TF-IDF
#vectorizer. There does not really seem to be a specific elbow point although the reconstruction error does starts to have a
#lower decrease. So we can't really rely only on this plot in case we overfit.

#Reference link for calculating the reconstruction error by number of components: https://medium.com/@quindaly/step-by-step-nmf-example-in-python-9974e38dc9f9
#Reference link for sparse library: https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html

from sklearn.decomposition import NMF
from gensim.models.coherencemodel import CoherenceModel
from gensim.corpora import Dictionary

Possession_Text_Corpuses = [[word.lower() for word in document.split()] for document in df_concatenated_text['concatenated_text']]
Possession_Dictionary = Dictionary(Possession_Text_Corpuses) #This is the whole dictionary with every unique possession term
#found in the dataset

e_score = 0
```

```
best_topics = None
best_number_of_components = 0 #We store the best model results (with the lowest coherence) into these variables

for component in range(2,21):
    print(f'Running NMF Topic Modeling With {component} Components...')
    NMF_Mdl = NMF(n_components=component, random_state=42) #We set random_state = 42 for reproducibility
    Weight_Matrix = NMF_Mdl.fit_transform(Sparse_Matrix)
    H_Feature_Matrix = NMF_Mdl.components_ #We create a NMF topic model with different number of topics (range 1-20). We are
    #going to calculate the coherence score for each topic for a total topics between 2-20.

    Topics = []
    event_feature_names = df_vectorizer.columns
    for topic_index, topic in enumerate(H_Feature_Matrix):
        top_indices_terms = topic.argsort()[-10:][::-1]
        topic_terms = [event_feature_names[idx] for idx in top_indices_terms if event_feature_names[idx] in Possession_Dictionary.token2id]
        topic_Ids = [Possession_Dictionary.token2id[term] for term in topic_terms]
        Topics.append(topic_Ids) #We are going to use only the top 10 words for each topic to calculate the coherence score

    Coherence_Mdl = CoherenceModel(topics=Topics, texts=Possession_Text_Corpuses, dictionary=Possession_Dictionary, coherence='c_v')
    #We are going to use the C_V measure to calculate the coherence score. This measure computes the pairwise word similarity
    #scores of the words in a topic and averages these scores.
    Coherence_Score = Coherence_Mdl.get_coherence()
    print(f'Coherence Score for {component} Components: {Coherence_Score}\n')

    if Coherence_Score > best_coherence_score:
        best_coherence_score = Coherence_Score
        best_topics = Topics
        best_number_of_components = component #We store the values for the number of topics that average the best coherence
        #score
print(f'Best Coherence Score: {best_coherence_score} With {best_number_of_components} Components')

#In this part of the code we iterate through a range of number of topics to split the possessions into and caluclate the average
#gensim coherence score of the topics. The coherence score is a measure that indicates whether the topics are meaningful and
#there are patterns or themes. In our case we c_v coherence score for the top 10 words as a sample which measures how often the
#terms appear together in possessions of the topic. The higher the coherence score, the more meaningful the topics are.

#Reference for calculating the coherence score by number of components (topics): https://www.kaggle.com/code/rockystats/topic-modelling-using-nmf

#References used for applying NMF Topic Modeling in Python:

#link1: https://www.freecodecamp.org/news/advanced-topic-modeling-how-to-use-svd-nmf-in-python
#link2: https://medium.com/voice-tech-podcast/topic-modelling-using-nmf-2f510d962b6e
#link3: https://spotintelligence.com/2023/09/08/non-negative-matrix-factorization/

### Showing the Top Words for Each Topic With Wordclouds

NMF_Topic_Modeling_Best_Mdl = NMF(n_components=best_number_of_components, random_state=42) #We set the same random_state as when
#we validated the parameters to get consistent results.
Weight_Matrix_Best_Mdl = NMF_Topic_Modeling_Best_Mdl.fit_transform(Sparse_Matrix)
H_Feature_Matrix_Best_Mdl = NMF_Topic_Modeling_Best_Mdl.components_
#We re-run the NMF model with the best parameters

for topic_index, topic in enumerate(H_Feature_Matrix_Best_Mdl):
    top_indices_terms = topic.argsort()[-10:][::-1] #We are going to use only the top 10 words for each topic to calculate the
    #coherence score
    top_features_by_topic = [event_feature_names[idx] for idx in top_indices_terms if event_feature_names[idx] in Possession_Dictionary.token2id]
    print(f'Top 10 Terms of Topic {topic_index+1}: {top_features_by_topic}')
    topic_Ids = [Possession_Dictionary.token2id[term] for term in top_features_by_topic]

    Best_Topic_Coherence_Mdl = CoherenceModel(topics=[topic_Ids], texts=Possession_Text_Corpuses,
                                              dictionary=Possession_Dictionary, coherence='c_v')
    Best_Topic_Coherence_Score = Best_Topic_Coherence_Mdl.get_coherence()
    print(f'Coherence Score for Topic {topic_index +1}: {Best_Topic_Coherence_Score}\n') #We also calculate the coherence score
    #of each topic to see how meaningful they are

#We calculate the coherence score for each individual topic to see deviation from the average and also take into account how
#meaningful each topic is.

#Reference link for showing the top words for each topic: https://www.kaggle.com/code/rockystats/topic-modelling-using-nmf

from wordcloud import WordCloud

event_feature_names = df_vectorizer.columns
number_of_topics = H_Feature_Matrix_Best_Mdl.shape[0]

figure, axes = plt.subplots(5,2,figsize=(20,30))
axes = axes.flatten()

for index in range(min(number_of_topics, len(axes))):
    topic = H_Feature_Matrix_Best_Mdl[index]
    topic_terms = ' '.join([event_feature_names[z] for z in topic.argsort()[-15:][::-1]])
    wordcloud = WordCloud(width=400, height=300, background_color='white').generate(topic_terms)

    axes[index].imshow(wordcloud, interpolation='lanczos')
    axes[index].set_title(f'Topic {index+1}', fontsize=20)
    axes[index].axis('off')
    #We generate a wordcloud for each topic showing the top 15 words.

for i in range(index+1, len(axes)):
    axes[i].axis('off')

plt.show()

#In this part of the code we create wordclouds of the top 15 words appearing in each topic. This will help us analyze the topics
#and understand what type of possessions these topics represent.

#Reference link for plotting wordclouds:

#link1: https://www.freecodecamp.org/news/advanced-topic-modeling-how-to-use-svd-nmf-in-python (section 7)
#link2: https://www.datacamp.com/tutorial/wordcloud-python?utm_source=google&utm_medium=paid_search&utm_campaignid=19589720824&utm_adgroupid=157156376311&utm_c

#After analyzing the keywords for each topic here is how we can define each topic:

#Topic 1: This topic seems to revolve around a controlled, patient buildup from the back involving defenders and longer
#possessions, with a mix of forward and backward passes. These possessions include actions like throw-ins and expansive
#wide play, suggesting a focus on maintaining possession through defensive stability and basic ball movement using thw whole
#width of the pitch.

#Topic 2: This topic emphasizes a defensive, centrally compact style with short, low-activity possessions starting and ending in
#the defensive third and involving interceptions. This indicates that in possessions of this topic, the team struggles to keep
# possession of the ball.
```

```
#Topic 3: This topic is focused on goalkeepers, defenders and long passes, often involving wide play and high passes for
#build-up. This topic focuses on expansive width play from the back and medium-duration possessions. In these types of
#possessions, the Goalkeeper mainly and the defenders often act as a long-range playmakers to start possessions, delivering long
#balls to the flanks in an attempt to find forwards who exploit spaces behind the defensive line.

#Topic 4: This topic deals with offensive plays culminating in shots, often involving crosses and shots with either foot. It
#highlights possessions developing from wide positions and actions mainly happening on the left flank and includes long passes
#originating from the defensive third. The activities are characterized by low activity and short duration, with a mix of
#accurate and inaccurate outcomes.

#Topic 5: This topic revolves around midfield play, emphasizing possessions happening in the central zones with simple and short
#passes both forward and backward. Possession of this topic show moderate activity and duration, involving simple and short
#passes and emphasizing the role of midfielders. These possessions are not that focused on playing from the back but mainly
#focus on getting the ball to the midfielders quickly.

#Topic 6: This topic describes possessions that shift towards the right side of the pitch once the ball has been moved forward
#to the attacking third or once the ball has been intercepted in dangerous areas in the attacking third. These possessions are
#compact, utilizing right and central zones of the pitch without stretching across the full width of the pitch. This indicates
#These possessions are short with few number of passes indicating that this topic mainly describes scenarios where Arsenal
#have intercepted the ball in dangerous areas in the attacking third and exploit the disorganised defence and the open spaces
#and use right sided players to progress towards goal and create chances.

#Topic 7: This topic represents possessions that involve more direct, forward passing with a focus on quick ball movement
#towards the opponent goal. We can also see that forward players tend to be heavily involved in these possessions to help
#distribute the ball across the full length of the pitch.

#Topic 8: This topic describes a dynamic and fast offensive style of play involving smart and risky passes. The possessions
#often start in the midfield end and end in the attacking third, highlighting an aggressive approach towards scoring with quick
#transitions and rapid advances up the pitch. These possessions tend to develop from the central zone areas indicating
#possessions where creative midfield players often try to break the lines and put through balls to the attackers.

### Exploring How Frequently Each Topic Occurs in Different Game Contexts

df_contexts['topic'] = Weight_Matrix_Best_Mdl.argmax(axis=1)

topic_time_segment_analysis = pd.crosstab(df_contexts['topic'], df_contexts['time_segment (min)'])
topic_opponent_strength_analysis = pd.crosstab(df_contexts['topic'], df_contexts['opponent_strength'])
topic_month_analysis = pd.crosstab(df_contexts['topic'], df_contexts['month'])
topic_result_analysis = pd.crosstab(df_contexts['topic'], df_contexts['result'])
topic_home_vs_away_analysis = pd.crosstab(df_contexts['topic'], df_contexts['home_game'])



#Now that we created the wordclouds for each topic and we have a better understanding of what the topics represent, the next
#step is to see how often each topic of possessions occurs in different game contexts. The game contexts that we will be looking
#into are the time segment of games, against different opponent strengths, in different months of the football season, comparing
#games where Arsenal have either won or drew/lost, and comparing home vs away games.

#We have created dataframes that calculate how often each topic occurs in each game context (counts).

result = df.groupby('result').count()[['num_events']]

normalized_df_result= topic_result_analysis.copy()
for column in normalized_df_result.columns:
    normalized_df_result[column] = normalized_df_result[column] / result.loc[column, 'num_events']


result_percentages = normalized_df_result.div(normalized_df_result.sum(axis=1), axis=0) * 100

figure, axes = plt.subplots(figsize=(8,4))

result_percentages.plot(kind='barh', stacked=True, ax=axes, color=['blue', 'orange', 'green'])

axes.set_title('Result of Games')
axes.set_xlabel('Percentage of Possessions')
axes.set_ylabel('Topic')
axes.set_yticklabels(range(1, len(topic_result_analysis.index)+1))
axes.set_xticks(range(0,101,10))


axes.tick_params(axis='x', labelsize=12)
axes.tick_params(axis='y', labelsize=12)
axes.legend(labels=['Games Drew', 'Games Lost', 'Games Won'])

plt.show()

home_game = df.groupby('home_game').count()[['num_events']]

normalized_df_home_vs_away= topic_home_vs_away_analysis.copy()
for column in normalized_df_home_vs_away.columns:
    normalized_df_home_vs_away[column] = normalized_df_home_vs_away[column] / home_game.loc[column, 'num_events']

opponent_strength = df.groupby('opponent_strength').count()[['num_events']]

normalized_df_opponent_strength= topic_opponent_strength_analysis.copy()
for column in normalized_df_opponent_strength.columns:
    normalized_df_opponent_strength[column] = normalized_df_opponent_strength[column] / opponent_strength.loc[column, 'num_events']

time_segment = df.groupby('time_segment (min)').count()[['num_events']]

normalized_df_time_segment= topic_time_segment_analysis.copy()
for column in normalized_df_time_segment.columns:
    normalized_df_time_segment[column] = normalized_df_time_segment[column] / time_segment.loc[column, 'num_events']

fig, axes = plt.subplots(3,1, figsize=(20,20))

def create_stacked_bar_charts(df, ax, title, colors, labels):
    df_percentages_by_context = df.div(df.sum(axis=1), axis=0) * 100 #For each row in the dataframe (each topic) we transform
    #the counts that represent the total possessions into percentages
    df_percentages_by_context.plot(kind='barh', stacked=True, ax=ax) #We want the bar charts to be stacked so we can
    #simultaneously compare the contexts
    ax.set_title(title, fontsize=18)
    ax.set_xlabel('Percentage of Possessions', fontsize=15)
    ax.tick_params(axis='x', labelsize=14)
    ax.set_ylabel('Topic', fontsize=15)
    ax.tick_params(axis='y', labelsize=14)
    ax.set_xticks(range(0,101,10))
    ax.set_yticklabels(range(1, len(df.index)+1))
    ax.legend(labels=labels, loc='upper left', fontsize=14)
```

```
create_stacked_bar_charts(normalized_df_home_vs_away, axes[0], 'Home vs Away Games', ['blue', 'red'],
                          ['Away Game', 'Home Game'])
create_stacked_bar_charts(normalized_df_opponent_strength, axes[1], 'Against Top 10 vs Bottom 10 Teams', ['blue', 'red'],
                          ['Bottom 10 Teams', 'Top 10 Teams'])
create_stacked_bar_charts(normalized_df_time_segment, axes[2], 'Time Segment', ['blue', 'red', 'orange'],
                          ['0-30 Mins', '30-60 Mins', '60-90+ Mins'])

#We create stacked bar charts to compare the occurence of each topic in different contexts

import matplotlib.cm as cm

month = df.groupby('month').count()[['num_events']]

normalized_df = topic_month_analysis.copy()
for column in normalized_df.columns:
    normalized_df[column] = (normalized_df[column] / month.loc[column, 'num_events']) * 100
    #Since the number of possessions varies across the contexts, and in general more possessions have happened in certain
    #contexts. For example the possessions in home games are slightly more than the possessions of away games which means
    #that the counts are biased towards the home games. But in our case we want to show the distribution of topics in different
    #game contexts so we need to normalize the values. One way to do that is to divide the values in each context for each
    #topic with the total number of possessions for the context. This will help us better understand the distribution of topics
    #in each context and how often they appear.

    #For each topic:
    #Normalized Count of Possessions For Each Month = (Actual Count of Possessions in Each Month / Total Number of Possessions in Each Month) * 100

month_order = ['August', 'September', 'October', 'December', 'January', 'February', 'March', 'April', 'May']
#Season starts in August and season ends in May
normalized_df = normalized_df[month_order]

colors = ['#FF5733', '#33FF57', '#3357FF', '#FF33A1', '#33FFA1', '#C47F0A', '#FFD700', '#8B4513'] #Each topic will have its own unique
#color

plt.figure(figsize=(14, 5))
for idx, topic in enumerate(normalized_df.index):
    plt.plot(normalized_df.columns, normalized_df.loc[topic], marker='o', markersize=7,
             label=f'Topic {topic + 1}', color=colors[idx % len(colors)])
plt.title('Normalized Frequency of Each Topic by Month', fontsize=15)
plt.ylabel('Normalized Frequency', fontsize=12)
plt.legend()
plt.grid(True)
plt.xticks(rotation=45, fontsize=12)
plt.yticks(fontsize=12)

plt.show()

topic_gameweek_analysis = pd.crosstab(df_contexts['gameweek'], df_contexts['topic'])

result_by_gameweek = df.groupby(['gameweek', 'result']).count()

result_by_gameweek.reset_index(inplace=True)

result_by_gameweek.set_index('gameweek', inplace=True)

merged_df = topic_gameweek_analysis.join(result_by_gameweek['result'])

new_column_names = {0: '1', 1:'2', 2:'3', 3:'4', 4:'5', 5:'6', 6:'7', 7:'8'}

merged_df.rename(columns=new_column_names, inplace=True)

merged_df = merged_df.reset_index()

merged_df.loc[20, 'result'] = 'Draw' #The game away at West Brom was a draw not a loss

#We calculate the frequency of each topic for every gameweek. This will help us in the temporal analysis of the frequency of
#each topic to understand how often each topic is being used over time.

arsenal_matches = pd.read_csv('arsenal_matches.csv')

arsenal_matches

arsenal_matches['Final Score'] = arsenal_matches['HomeScore'].astype(str) + ' - ' + arsenal_matches['AwayScore'].astype(str)

def extract_arsenal_opponents(row):
    if row['HomeTeam'] == 'Arsenal':
        return row['AwayTeam']
    elif row['AwayTeam'] == 'Arsenal':
        return row['HomeTeam']

arsenal_matches['Opponent'] = arsenal_matches.apply(extract_arsenal_opponents, axis=1)

#We extract the opponent of Arsenal's each game

def extract_venue(row):
    if row['venue'] == 'Emirates Stadium':
        return 'Home'
    else:
        return 'Away'

arsenal_matches['Home or Away Game'] = arsenal_matches.apply(extract_venue, axis=1)

#We extract information about whether Arsenal were playing home or away

arsenal_opponents_by_gameweek = pd.DataFrame(arsenal_matches.iloc[:,-3:])

final_league_positions = {
    'Manchester City': 1, 'Manchester United': 2, 'Tottenham Hotspur': 3,
    'Liverpool': 4, 'Chelsea': 5, 'Arsenal': 6, 'Burnley': 7,
    'Everton': 8, 'Leicester City': 9, 'Newcastle United': 10,
    'Crystal Palace': 11, 'AFC Bournemouth': 12, 'West Ham United': 13,
    'Watford': 14, 'Brighton & Hove Albion': 15, 'Huddersfield Town': 16,
    'Southampton': 17, 'Swansea City': 18, 'Stoke City': 19, 'West Bromwich Albion': 20
}

def assign_final_position(team):
    return final_league_positions[team]

arsenal_opponents_by_gameweek['Final Position'] = arsenal_opponents_by_gameweek['Opponent'].apply(assign_final_position)

#We assign the final league position of all opponents that Arsenal faced.
```

```
merged_df = pd.concat([merged_df, arsenal_opponents_by_gameweek], axis=1)

merged_df

#We merge the datasets to have the total possessions by topic for each gameweek alongside any relevant contextual information
#about the games.

colors_by_topic = ['#FF5733', '#33FF57', '#3357FF', '#FF33A1', '#33FFA1', '#C47F0A', '#FFD700', '#8B4513'] #Each topic will have its own
#a unique color

plt.figure(figsize=(15,10))

Topics = merged_df.columns.tolist()[1:9]

for index, topic in enumerate(Topics):
    plt.plot(merged_df['gameweek'], merged_df[topic], marker='o', markersize=8, linewidth=2,
             color=colors_by_topic[index % len(colors_by_topic)], label=f'Topic {int(topic)}')

plt.title('Frequency of Each Topic by Gameweek', fontsize=18, pad=15)
plt.xlabel('Gameweek', fontsize=16)
plt.ylabel('Frequency', fontsize=16)

for index, row in merged_df.iterrows():
    plt.text(row['gameweek'], max(row[Topics])+2, f"{row['result']} | {row['Final Score']} | {row['Opponent']} | {row['Home or Away Game']} | FP:{row['Final Po
             ha='center', va='bottom', fontsize=12, rotation=90)
    #We are also going to show whether in that gameweek the team Won, lost or drew the game for a more detailed analysis which
    #will help us even more find some ineresting findings and tactical patterns. We will also show the final score of each game,
    #the opponent, whether the game was played at home or away, and the final league position of the opponent.

plt.xticks(ticks=range(1,39))
plt.xticks(rotation=45)
plt.ylim(0,120)
plt.tick_params(axis='x', labelsize=13)
plt.tick_params(axis='y', labelsize=13)
plt.legend(fontsize=12)
plt.grid(True)

plt.tight_layout()
plt.show()

#This plot shows the frequency of each topic by gameweek. We have not normalized the frequency because we are comparing the
#topics in a context and not the contexts for each topic.
```