Time-Series Analysis of Box-Scores

Arthur Lin and Jason Lott

MATH498R Final Project
Spring 2024
Prof. Yanir A. Rubinstein

Abstract

The goal of our project was to track personal and team statistics over time during games and seasons and explore the formulas of many advanced statistics. We utilized the NBA's public API to get most of our data as well as many basketball stats reference websites to learn about the most modern advanced stats being used. We used the NBA's play by play data to track how one player's traditional stats accumulated throughout the course of one game. Next, we expanded this to create visualizations of one player's field goal percentage in every game of one regular season and also their cumulative field goal percentage over the same season. We continued to create visualizations of the cumulative graphs of many advanced statistics for one player over the course of one game. Finally, we used a few regression techniques to approximate the formula for true shooting percentage, which has a known formula, and Player Production Average (PPA), which has an unknown formula.

Contents

1 Code Documentation

1.1 Libraries and Endpoints

```
import pandas as pd
```

From: Pandas Library
Description: Imports the pandas library and assigns it the alias "pd". Used for data manipulation and analysis.

```
import numpy as np
```

From: NumPy Library
Description: Imports the NumPy library and assigns it the alias "np". Used for numerical computing and provides support for arrays and matrices, along with a collection of mathematical functions to operate on these arrays.

```
import matplotlib.pyplot as plt
```

From: Mat-Plot Library
Description: Imports the pyplot module from the matplotlib library and assigns it the alias "plt". Used for creating plots and visualizations.

```
from nba_api.stats.endpoints import playbyplayv2
```

From: NBA API Stats Endpoints Library
Description: Imports the NBA API endpoint PlayByPlayV2 into the program. Used to find play-by-play info for games. (We chose this endpoint over the two other play-by-play endpoints because this one provides information about the players involved in the play.)

```
from nba_api.stats.endpoints import boxscoreadvancedv2
```

From: NBA API Stats Endpoints Library
Description: Imports the NBA API endpoint BoxScoreAdvancedV2 into the program. We used this specifically to find a player's pace, but it provides other advanced stats as well.

```
from nba_api.stats.static import players
```

From: NBA API Static Stats Library
Description: Imports the NBA API module players, which contains player identifiers like ID and name.

```
from nba_api.stats.endpoints import leaguegamefinder
```

From: NBA API Stats Endpoints Library
Description: Imports the NBA API endpoint LeagueGameFinder into the program. Used to find the box scores for games. You can pass in parameters for the season, player, etc.

```
from nba_api.stats.endpoints import gamerotation
```

From: NBA API Stats Endpoints Library
Description: Imports the NBA API endpoint GameRotation into the program. Used to find the substitutions in a game.

```
from nba_api.stats.endpoints import boxscoreadvancedv3
```

From: NBA API Stats Endpoints Library
Description: Imports the NBA API endpoint boxscoreadvancedv3 into the program. It is used to get the advanced stats for every player who played in a specified game.

```
from nba_api.stats.endpoints import boxscoretraditionalv3
```

From: NBA API Stats Endpoints Library
Description: Imports the NBA API endpoint boxscoretraditionalv3 into the program. It is used to get the traditional stats for every player who played in a specified game.

```
from nba_api.stats.endpoints import TeamGameLogs
```

From: NBA API Stats Endpoints Library
Description: Imports the NBA API endpoint TeamGameLogs into the program. It is used to gather game IDs for a specified season and/or team.

```
from sklearn.linear_model import LinearRegression
```

From: Sci-Kit Learn Linear Models Library
Description: Imports the Sci-Kit Learn model Linear Regression into the program. It is used to approximate the formula for some advanced stats.

```
from sklearn.ensemble import RandomForestRegressor
```

From: Sci-Kit Ensemla Library
Description: Imports the Sci-Kit Learn model Random Forest Regressor into the program. It is used to approximate the formula for some advanced stats.

```
from sklearn.model_selection import train_test_split
```

From: Sci-Kit Learn Model Selection Library
Description: Imports the Sci-Kit Learn function train_test_split into the program. It is used to randomly split the data into training and testing datasets.

```
from tqdm import tqdm
```

From: TQDM Library
Description: Imports the TQDM function tqdm. It is used to create progress bars to track the progress of long data pulls.

```
import time
```

From: Python Standard Library
Description: Imports the time package into the program. It is used to delay the program in the retry wrapper.

```
import requests
```

From: Python Standard Library
Description: Imports the requests package into the program. It is used to handle exceptions in the retry wrapper.

1.2 Identifying a Player and Game of Interest

```
all_players = players.get_players()
```

Retrieves names and IDs for all players in the NBA API.

```
def get_player_id(player_name):
```

Defines a function called get_player_id that takes one parameter, player_name.

```
    for player in all_players:
```

Starts a for loop that iterates over each player in the all_players list.

```
        if player['full_name'].lower() == player_name.lower():
```

Checks for a player name in the all_players list that matches the user input.

```
            return player['id']
```

Returns the ID of the player if a name match is found.

```
        return None
```

Returns None if the user entered a name that isn't in the list.

```
player_name = input("Enter a player's full name: ")
```

Prompts the user to enter a player's full name and assigns the input to a variable, player_name.

```
last_name = player_name.split(' ')[1]
```

Assigns the second value of the input to a variable, last_name.

```
player_id = get_player_id(player_name)
```

Assigns the player ID returned by get_player_id to a variable, player_id.

```
if player_id:
```

Checks if a player ID was found.

```
print(f"The player ID for {player_name} is {player_id}.")
```

Prints a formatted string informing the user of the ID for the player they input.

```
else:
```

If the previous condition was not met (i.e., a player ID was not found), executes the following line of code.

```
print(f"Player '{player_name}' not found.")
```

Prints a formatted string informing the user that the player they provided was not found.

```
season = input('Enter the years a season spans (e.g. 2023-24): ')
```

Prompts the user to enter a season of interest and assigns it to a variable, season.

```
gamefinder = leaguegamefinder.LeagueGameFinder(player_id_nullable = player_id, season_nullable= season)
```

Uses the LeagueGameFinder NBA API endpoint with the parameters stored in the variables player_id and season to retrieve a player's box scores for each game in the given season.

```
games = gamefinder.get_data_frames()[0]
```

Returns the box score data in the variable gamefinder as a list of pandas DataFrames and assigns the list to a variable, games.

```
pd.set_option('display.max_columns',250)
```

Sets the maximum number of columns to display in pandas DataFrames to 250.

```
pd.set_option('display.max_colwidth', None)
```

Sets the maximum column width in pandas DataFrames to None. Ensures content in the DataFrame is not truncated.

```
def get_game_id(game_date):
```

Defines a function called get_game_id that takes one parameter, game_date.

```
    game = games[games['GAME_DATE'] == game_date]
```

Finds the row in the games DataFrame that has a 'GAME_DATE' column value equal to the user input date. The row is assigned to a new variable, game.

```
    if len(game) > 0:
```

Executes the following code if a game was found.

```
        return game['GAME_ID'].iloc[0]
```

Returns the 'GAME_ID' column value for the chosen date's game.

```
    else:
```

Executes the following code if a game was not found for the chosen date.

```
        return "No game found for the given date."
```

Returns a string informing the user that their date input did not return a game.

```
game_date = input("Enter the game date (YYYY-MM-DD): ")
```

Prompts the user to enter the date of a game they want to analyze. The input is assigned to a variable, game_date.

```
game_id = get_game_id(game_date)
```

Calls the get_game_id function with the new variable, game_date as the parameter. The resulting game ID is assigned to the variable, game_id.

```
play_by_play = playbyplayv2.PlayByPlayV2(game_id=game_id)
```

Instantiates an object of a class named PlayByPlayV2 from the playbyplayv2 endpoint. Finds the play-by-play for the game with the given game ID and stores it in a variable, play_by_play.

```
pd.set_option('display.max_columns',250)
```

Sets the maximum number of columns to display in pandas DataFrames to 250.

```
pd.set_option('display.max_colwidth', None)
```

Sets the maximum column width in pandas DataFrames to None. Ensures content in the DataFrame is not truncated.

```
pbp_df = play_by_play.get_data_frames()[0]
```

Retrieves the play-by-play data in DataFrame format and assigns the DataFrame to a variable, pbp_df.

```
cleaned_pbp_df = pbp_df[['GAME_ID', 'EVENTMSGTYPE', 'PERIOD', 'PCTIMESTRING', 'HOMEDESCRIPTION',
    ↪ 'VISITORDESCRIPTION', 'SCORE', 'PLAYER1_ID', 'PLAYER1_NAME', 'PLAYER1_TEAM_ABBREVIATION',
    ↪ 'PLAYER2_ID', 'PLAYER2_NAME', 'PLAYER2_TEAM_ABBREVIATION']]
```

Indexes pbp_df to create a new DataFrame, cleaned_pbp _df that contains only the columns we need.

```
pd.set_option('display.max_rows', None)
```

Displays all of the rows in a pandas DataFrame.

```
home_boolean = True
```

Creates a boolean variable, home_boolean, and assigns it a value of True.

```
game = games[games['GAME_DATE'] == game_date]
```

Filter the games data frame to include only the game on the date the user inputted. Store the resulting data frame in variable game.

```
if game['MATCHUP'].str.contains('vs.').any():
```

Executes the following code if the 'MATCHUP' column value of the game variable contains a substring of 'vs.'.

```
    column = 'HOMEDESCRIPTION'
```

Assigns the string value 'HOMEDESCRIPTION' to the variable, column.

```
else:
```

Executes the following code if the 'MATCHUP' column value of the game variable does not contain a substring of 'vs.'.

```
    home_boolean = False
```

Assigns a value of False to the home_boolean variable.

```
    column = 'VISITORDESCRIPTION'
```

Assigns the string value 'VISITORDESCRIPTION' to the variable, column.

```
column
```

Prints the value of column.

```
def convert_to_seconds(row):
```

Defines a function called convert_to_seconds that takes one parameter, row.

```
    period = row['PERIOD']
```

Assigns a row's 'PERIOD' column value to the variable, period.

```
    pctimestring = row['PCTIMESTRING']
```

Assigns a row's 'PCTIMESTRING' column value to the variable, pctimestring.

```
    seconds_elapsed = (int(period) - 1) * 12 * 60 + (12 * 60 - int(pctimestring.split(':')[0]) * 60) -
        ↪ int(pctimestring.split(':')[1])
```

Calculates the seconds elapsed in the game at a particular row by manipulating the string values in period and pctimestring.

```
    return seconds_elapsed
```

Returns the calculation.

```
cleaned_pbp_df['SECONDS_ELAPSED'] = cleaned_pbp_df.apply(convert_to_seconds, axis=1)
```
Applies the function convert_to_seconds to cleaned_pbp_df to create a new column, 'SECONDS_ELAPSED'.

```
def parse_input(input_str):
```
Defines a function called parse_input that takes one parameter, input_str.

```
    if not input_str:
```
If the variable input_str is empty, execute the following code.

```
        return None, None
```
Returns None values for both the quarter and time left.

```
    period, time_left = input_str.split(',')
```
Splits the user input into substrings based on a comma delimiter, and assigns the two substrings to the variables period and time_left.

```
    return int(period), time_left.strip()
```
Returns a tuple with two values, the period and time left. The strip method removes any leading and trailing white space.

```
def filter_time_frame(df, start_period=None, start_time=None, end_period=None, end_time=None):
```
Defines a function, filter_time_frame, with five parameters. All parameters except df will default to None if they are not provided.

```
    if start_period is not None and start_time is not None:
```
If the variables start_period and start_time are not None, execute the following code.

```
        start_seconds = (start_period - 1) * 12 * 60 + (12 * 60 - int(start_time.split(':')[0]) * 60) -
            ↪ int(start_time.split(':')[1])
```
Calculates the starting time in seconds by manipulating the user−provided string values in start_period and start_time. Assigns the value to a variable, start_seconds

```
        df = df[df['SECONDS_ELAPSED'] >= start_seconds]
```
Indexes the DataFrame stored in the variable df for rows where the SECONDS_ELAPSED column value is greater than or equal to the value stored in start_seconds.

```
    if end_period is not None and end_time is not None:
```
If the variables end_period and end_time are not None, execute the following code.

```
        end_seconds = (end_period - 1) * 12 * 60 + (12 * 60 - int(end_time.split(':')[0]) * 60) -
            ↪ int(end_time.split(':')[1])
```
Calculates the ending time in seconds by manipulating the user−provided string values in end_period and end_time. Assigns the value to a variable, end_seconds

```
        df = df[df['SECONDS_ELAPSED'] <= end_seconds]
```
Indexes the DataFrame stored in the variable df for rows where the SECONDS_ELAPSED column value is less than or equal to the value stored in end_seconds.

```
    return df
```
Returns the filtered DataFrame containing the play−by−play for a specific time frame of the game.

```
time_frame_start = input("Enter a starting period and time left (e.g. 1, 11:45), or press enter to start
    ↪ at the beginning of the game: ")
```
Prompts the user to enter a starting period and time in a specific format, or to skip by pressing enter. The input is assigned to a variable, time_frame_start.

```
time_frame_end = input("Enter the ending period and time left, or press enter to end at the end of the
    ↪ game: ")
```

Prompts the user to enter an ending period and time in a specific format, or to skip by pressing enter. The input is assigned to a variable, time_frame_end.

`start_period, start_time = parse_input(time_frame_start)`

Calls the parse_input function on time_frame_start and assigns the two substrings to the variables start_period and start _time.

`end_period, end_time = parse_input(time_frame_end)`

Calls the parse_input function on time_frame_end and assigns the two substrings to the variables end_period and end _time.

`cleaned_pbp_df = filter_time_frame(cleaned_pbp_df, start_period, start_time, end_period, end_time)`

Calls the function, filter_time_frame to filter cleaned_pbp _df to the given time frame.

`cleaned_pbp_df`

Prints the updated DataFrame stored in cleaned_pbp_df.

1.3 Aggregating Traditional and Advanced Stats Into One DataFrame

```
cleaned_pbp_df['SCORE'].iloc[0] = '0 - 0'
```

Sets the first value of the SCORE column in cleaned_pbp_df to '0 - 0'.

```
cleaned_pbp_df['SCORE'].fillna(method='ffill', inplace=True)
```

Fills missing values of the SCORE column of cleaned_pbp_df using forward filling. This replaces NaN values with the last non-null value in the column.

```
cleaned_pbp_df['GAME_PTS'] = cleaned_pbp_df['SCORE'].apply(lambda x: sum(map(int, x.split(' - '))))
```

Uses a lambda function to calculate the total points in the game by splitting strings in the SCORE column. Assigns values to a new column of cleaned_pbp_df called GAME_PTS.

```
cleaned_pbp_df['GAME_FGM'] = (cleaned_pbp_df['EVENTMSGTYPE'] == 1).cumsum()
```

Creates a column, GAME_FGM in cleaned_pbp_df that is filled with the cumulative sum of rows with an EVENTMSGTYPE column value of 1. In the NBA API, a made field goal is denoted with an EVENTMSGTYPE value of 1.

```
cleaned_pbp_df['GAME_FTM'] = ((cleaned_pbp_df['EVENTMSGTYPE'] == 3) &
    ↪ (~cleaned_pbp_df['HOMEDESCRIPTION'].str.contains('MISS', na=False)) &
    ↪ (~cleaned_pbp_df['VISITORDESCRIPTION'].str.contains('MISS', na=False))).cumsum()
```

Creates a column, GAME_FTM in cleaned_pbp_df that is filled with the cumulative sum of rows that have an EVENTMSGTYPE column value of 3, and do not contain the substring 'MISS' in both the HOMEDESCRIPTION and VISITORDESCRIPTION columns. In the NBA API, a free throw is denoted with an EVENTMSGTYPE value of 3.

```
cleaned_pbp_df['GAME_FGA'] = (cleaned_pbp_df['EVENTMSGTYPE'].isin([1, 2])).cumsum()
```

Creates a column, GAME_FGA in cleaned_pbp_df that is filled with the cumulative sum of rows with an EVENTMSGTYPE column value of 1 or 2. In the NBA API, a made field goal is denoted with an EVENTMSGTYPE value of 1, and a missed field goal is denoted with a value of 2.

```
cleaned_pbp_df['GAME_FTA'] = (cleaned_pbp_df['EVENTMSGTYPE'] == 3).cumsum()
```

Creates a column, GAME_FTA in cleaned_pbp_df that is filled with the cumulative sum of rows with an EVENTMSGTYPE column value of 3. In the NBA API, a free throw is denoted with an EVENTMSGTYPE value of 3.

```
cleaned_pbp_df['GAME_REB'] = (cleaned_pbp_df['EVENTMSGTYPE'] == 4).cumsum()
```

Creates a column, GAME_REB in cleaned_pbp_df that is filled with the cumulative sum of rows with an EVENTMSGTYPE column value of 4. In the NBA API, a rebound is denoted with an EVENTMSGTYPE value of 4.

```
cleaned_pbp_df['GAME_AST'] = ((cleaned_pbp_df['HOMEDESCRIPTION'].str.contains('AST', na=False)) |
    ↪ (cleaned_pbp_df['VISITORDESCRIPTION'].str.contains('AST', na=False))).cumsum()
```

Creates a column, GAME_AST in cleaned_pbp_df that is filled with the cumulative sum of rows in the HOMEDESCRIPTION and VISITORDESCRIPTION columns that contain the substring 'AST'. If a particular column value is missing, the na = False parameter will ensure that it is treated as not containing the substring.

```
cleaned_pbp_df['GAME_STL'] = ((cleaned_pbp_df['HOMEDESCRIPTION'].str.contains('STL', na=False)) |
    ↪ (cleaned_pbp_df['VISITORDESCRIPTION'].str.contains('STL', na=False))).cumsum()
```

Creates a column, GAME_STL in cleaned_pbp_df that is filled with the cumulative sum of rows in the HOMEDESCRIPTION and VISITORDESCRIPTION columns that contain the substring 'STL'.

```
cleaned_pbp_df['GAME_BLK'] = ((cleaned_pbp_df['HOMEDESCRIPTION'].str.contains('BLK', na=False)) |
    ↪ (cleaned_pbp_df['VISITORDESCRIPTION'].str.contains('BLK', na=False))).cumsum()
```

Creates a column, GAME_STL in cleaned_pbp_df that is filled with the cumulative sum of rows in the HOMEDESCRIPTION and VISITORDESCRIPTION columns that contain the substring 'BLK'.

```
cleaned_pbp_df['GAME_PF'] = (cleaned_pbp_df['EVENTMSGTYPE'] == 6).cumsum()
```

Creates a column, GAME_PF in cleaned_pbp_df that is filled with the cumulative sum of rows with an EVENTMSGTYPE column value of 6. In the NBA API, a personal foul is denoted with an EVENTMSGTYPE value of 6.

```
cleaned_pbp_df['GAME_TO'] = (cleaned_pbp_df['EVENTMSGTYPE'] == 5).cumsum()
```

Creates a column, GAME_TO in cleaned_pbp_df that is filled with the cumulative sum of rows with an EVENTMSGTYPE column value of 5. In the NBA API, a turnover is denoted with an EVENTMSGTYPE value of 5.

```
cleaned_pbp_df['TEAM_AST'] = (cleaned_pbp_df[column].str.contains('AST', na=False)).cumsum()
```

Creates a column, TEAM_AST in cleaned_pbp_df that is filled with the cumulative sum of rows in the column stored in the variable, column, that contains the substring 'AST'.

```
team_abbreviation = games.loc[0, 'TEAM_ABBREVIATION']
```

Indexes the games DataFrame to find the first value of the TEAM_ABBREVIATION column and assigns the value to a variable, team_abbreviation.

```
team_scoring = cleaned_pbp_df[(cleaned_pbp_df['PLAYER1_TEAM_ABBREVIATION'] == team_abbreviation) &
    ↪ (cleaned_pbp_df['EVENTMSGTYPE'] == 1)]
```

Creates a DataFrame, team_scoring, by indexing cleaned_pbp_df for rows where PLAYER1_TEAM_ABBREVIATION is equal to the team_abbreviation variable and the EVENTMSGTYPE column value is 1.

```
cleaned_pbp_df['TEAM_FGM'] = team_scoring['EVENTMSGTYPE'].cumsum()
```

Creates a column, TEAM_FGM in cleaned_pbp_df that is filled with the cumulative sum of rows in team_scoring.

```
player_pbp = cleaned_pbp_df[cleaned_pbp_df['PLAYER1_ID'] == player_id]
```

Creates a DataFrame, player_pbp by indexing cleaned_pbp_df for rows where the PLAYER1_ID column has a value equal to the player_id variable.

```
substitution_events = cleaned_pbp_df[(cleaned_pbp_df['EVENTMSGTYPE'] == 8) &
    ↪ (cleaned_pbp_df[column].str.contains(last_name))]
```

Creates a DataFrame, substitution_events, by indexing cleaned_pbp_df for rows where the EVENTMSGTYPE column value is 8 and the column stored in the variable, column, contains the chosen player's last name as a substring. In the NBA API, a substitution is denoted with an EVENTMSGTYPE value of 8.

```
quarter_end = cleaned_pbp_df[cleaned_pbp_df['EVENTMSGTYPE'] == 13]
```

Creates a DataFrame, quarter_end by indexing cleaned_pbp_df for rows where the EVENTMSGTYPE column value is 13. In the NBA API, ends of quarters are denoted with an EVENTMSGTYPE value of 13.

```
player_pbp = pd.concat([player_pbp, substitution_events, quarter_end])
```

Concatenates the DataFrames substitution_events and quarter _end to player_pbp, and assign the resulting DataFrame to the variable player_pbp.

```
player_pbp.sort_index(inplace=True)
```

Sort the DataFrame index to ensure events appear sequentially by game time.

```
second_pbp = cleaned_pbp_df[cleaned_pbp_df['PLAYER2_ID'] == player_id]
```
Creates a DataFrame, second_pbp by indexing cleaned_pbp_df for rows where the PLAYER2_ID column value is equal to the player_id variable.

```
assists = second_pbp[second_pbp[column].str.contains(rf'{last_name} \d+ AST').fillna(False)]
```
Creates a DataFrame, assists, by indexing second_pbp for rows containing a specific string pattern of the player's last name, followed by one or more digits, followed by the string 'AST'.

```
player_pbp = pd.concat([player_pbp, assists], axis=0)
```
Concatenates the DataFrame assists with player_pbp along the rows and assigns the DataFrame to player_pbp.

```
steals = second_pbp[second_pbp[column].str.contains(rf'\d+ STL').fillna(False)]
```
Creates a DataFrame, steals, by indexing second_pbp for rows containing a specific string pattern of one or more digits, followed by the string 'STL'.

```
player_pbp = pd.concat([player_pbp, steals], axis=0)
```
Concatenates the DataFrame steals with player_pbp along the rows and assigns the DataFrame to player_pbp.

```
blocks = second_pbp[second_pbp[column].str.contains(rf'{last_name} \d+ BLK').fillna(False)]
```
Creates a DataFrame, blocks, by indexing second_pbp for rows containing a specific string pattern of the player's last name, followed by one or more digits, followed by the string 'BLK'.

```
player_pbp = pd.concat([player_pbp, blocks], axis=0)
```
Concatenates the DataFrame blocks with player_pbp along the rows and assigns the DataFrame to player_pbp.

```
player_pbp.drop_duplicates(inplace = True)
```
Removes duplicate rows from player_pbp and applies changes directly to player_pbp, rather than returning a new DataFrame.

```
player_pbp.sort_index()
```
Sorts the index of player_pbp.

```
player_pbp['PTS'] = player_pbp.loc[player_pbp['PLAYER1_ID'] == player_id, column].str.extract(r'(\d+) PTS')
```
Creates a new column, PTS, in player_pbp by indexing player_pbp for rows where the PLAYER1_ID value equals the variable, player_id. Then, finds a string pattern of one or more digits followed by the string 'PTS'. The parentheses extract the digits from the string, which become the values for the PTS column.

```
player_pbp['AST'] = player_pbp.loc[player_pbp['PLAYER2_ID'] == player_id, column].str.extract(r'(\d+) AST')
```
Creates a new column, AST, in player_pbp by indexing player_pbp for rows where the PLAYER2_ID value equals the variable, player_id. Then, finds a string pattern of one or more digits followed by the string 'AST'. The parentheses extract the digits from the string, which become the values for the AST column.

```
player_pbp['OREB'] = player_pbp[column].str.extract(r'REBOUND \(Off:(\d+)')
```
Creates a new column, OREB, in player_pbp by finding a string pattern of a string 'REBOUND' followed by the string '(Off:' and one or more digits. The parentheses extract the digits from the string, which become the values for the OREB column.

```
player_pbp['DREB'] = player_pbp[column].str.extract(r'REBOUND \(Off:\d+ Def:(\d+)')
```
Creates a new column, DREB, in player_pbp by finding a string pattern of a string 'REBOUND' followed by the string '(Off:+ Def:' and one or more digits. The parentheses extract the digits from the string, which become the values for the DREB column.

```
player_pbp['REB'] = pd.to_numeric(player_pbp['OREB']) + pd.to_numeric(player_pbp['DREB'])
```
Creates a new column, REB, in player_pbp by converting the string values in the OREB and DREB columns to numeric and summing.

```
player_pbp['STL'] = player_pbp.loc[player_pbp['PLAYER2_ID'] == player_id, column].str.extract(r'(\d+) STL')
```
Creates a new column, STL, in player_pbp by indexing player_pbp for rows where the PLAYER2_ID value equals the variable, player_id. Then, finds a string pattern of one or more digits followed by the string 'STL'. The parentheses

extract the digits from the string, which become the values for the STL column.

```
player_pbp['BLK'] = player_pbp.loc[player_pbp['PLAYER2_ID'] == player_id, column].str.extract(r'(\d+) BLK')
```

Creates a new column, BLK, in player_pbp by indexing player_pbp for rows where the PLAYER2_ID value equals the variable, player_id. Then, finds a string pattern of one or more digits followed by the string 'BLK'. The parentheses extract the digits from the string, which become the values for the BLK column.

```
player_pbp['TO'] = player_pbp[column].str.extract(r'Turnover \(P(\d+)\.T\d+\)')
```

Creates a new column, TO, in player_pbp by finding a string pattern of a string 'Turnover' followed by the string 'P' and one or more digits, followed by a '.' and finally a string value 'T' followed by one or more digits. The parentheses extract the digits after the substring 'P', which become the values for the TO column.

```
player_pbp['PF'] = ((player_pbp['EVENTMSGTYPE'] == 6) & (player_pbp['PLAYER1_ID'] == player_id)).cumsum()
```

Creates a column, PF in player_pbp that is filled with the cumulative sum of rows that have an EVENTMSGTYPE column value of 6 and a PLAYER1_ID column value that equals the variable, player_id. In the NBA API, a foul is denoted with an EVENTMSGTYPE value of 6.

```
player_pbp.sort_index(inplace = True)
```

Sort the DataFrame index to ensure events appear sequentially by game time.

```
player_pbp.drop_duplicates(inplace=True)
```

Removes duplicate rows from player_pbp and applies changes directly to player_pbp, rather than returning a new DataFrame.

```
player_pbp
```

Prints the updated player_pbp DataFrame.

```
player_fg = cleaned_pbp_df[(cleaned_pbp_df['EVENTMSGTYPE'].isin([1, 2])) & (cleaned_pbp_df['PLAYER1_ID']
    ↪ == player_id)]
```

Creates a DataFrame, player_fg, by indexing cleaned_pbp_df for rows where the EVENTMSGTYPE column value is 1 or 2 and the PLAYER1_ID column value equals the variable, player_id. In the NBA API, a made field goal is denoted with an EVENTMSGTYPE value of 1, and missed field goals are denoted with a value of 2.

```
player_fg["FGM"] = (player_fg["EVENTMSGTYPE"] == 1).cumsum()
```

Creates a new column, FGM, in player_fg by taking the cumulative sum of rows with a value of 1 in the EVENTMSGTYPE column.

```
player_fg["FGA"] = player_fg.reset_index(drop = True).index + 1
```

Creates a new column, FGA, in player_fg by resetting the index, dropping the current index (as opposed to keeping it as a new column), and adding one to the index values.

```
player_fg["FG%"] = (player_fg["FGM"] / player_fg["FGA"] * 100).round(2)
```

Creates a new column, FG%, in player_fg by dividing the FGM column values by the corresponding FGA values, multiplying by 100, and rounding to two decimal places.

```
player_pbp = pd.concat([player_pbp, player_fg], axis=0)
```

Concatenates the DataFrame player_fg with player_pbp along the rows and assigns the DataFrame to player_pbp.

```
player_pbp.update(player_fg[['FGM', 'FGA', 'FG%']])
```

Updates the player_pbp DataFrame with values from the FGM, FGA, and FG% columns of the player_fg DataFrame.

```
mask = player_fg[column].str.contains('3PT')
```

Creates a variable, mask, that filters the column of player_fg stored in the variable column for rows that contain the substring '3PT'.

```
player_3pt_attempts = player_fg[mask]
```

Filters player_fg using the mask variable and assign the result to a new DataFrame, player_3pt_attempts.

```
player_3pt_attempts["3PTM"] = (player_3pt_attempts["EVENTMSGTYPE"] == 1).cumsum()
```

Creates a new column, 3PTM, in player_3pt_attempts by taking the cumulative sum of rows with an EVENTMSGTYPE column value of 1 (made field goals).

```
player_3pt_attempts["3PTA"] = player_3pt_attempts.reset_index(drop = True).index + 1
```

Creates a new column, 3PTA, in player_3pt_attempts by resetting and dropping the index, and adding one to it.

```
player_3pt_attempts["3PT%"] = (player_3pt_attempts["3PTM"] / player_3pt_attempts["3PTA"] * 100).round(2)
```

Creates a new column, 3PT%, in player_3pt_attempts by dividing the 3PTM column values by the corresponding 3PTA values, multiplying by 100, and rounding to two decimal places.

```
player_pbp = pd.concat([player_pbp, player_3pt_attempts], axis=0)
```

Concatenates the DataFrame player_3pt_attempts with player_pbp along the rows and assigns the DataFrame to player_pbp.

```
player_pbp.update(player_3pt_attempts[['3PTM', '3PTA', '3PT%']])
```

Updates the player_pbp DataFrame with values from the 3PTM, 3PTA, and 3PT% columns of the player_3pt_attempts DataFrame.

```
player_free_throws = cleaned_pbp_df[(cleaned_pbp_df['EVENTMSGTYPE'] == 3) & (cleaned_pbp_df['PLAYER1_ID']
    ↪ == player_id)]
```

Creates a DataFrame, player_free_throws, by indexing cleaned_pbp_df for rows where the EVENTMSGTYPE column value is 3 and the PLAYER1_ID column value equals the variable, player_id. In the NBA API, a free throw is denoted with an EVENTMSGTYPE value of 3.

```
total_made = 0
```

Creates a variable, total_made and assigns it a value of 0.

```
total_attempts = 0
```

Creates a variable, total_attempts and assigns it a value of 0.

```
ft_made = []
```

Creates a variable, ft_made and assigns it an empty list.

```
ft_attempted = []
```

Creates a variable, ft_attempted and assigns it an empty list.

```
for index, row in player_free_throws.iterrows():
```

Begins the iteration of each row in player_free_throws.

```
    if 'MISS' not in row[column]:
```

Checks if the free throw was made.

```
        total_made += 1
```

If the condition is True, add one to total_made and assign the new value to total_made.

```
    ft_made.append(total_made)
```

Append each total_made value to the list, ft_made.

```
    total_attempts += 1
```

Add one to total_attempts and assign the new value to total_attempts.

```
    ft_attempted.append(total_attempts)
```

Append each total_attempts value to the list, ft_attempted.

```
player_free_throws['FTM'] = ft_made
```

Create a new column, FTM, in player_free_throws, and assign the values in the ft_made list to the column.

```
player_free_throws['FTA'] = ft_attempted
```

Create a new column, FTA, in player_free_throws, and assign the values in the ft_attempted list to the column.

```
player_free_throws['FT%'] = (player_free_throws['FTM'] / player_free_throws['FTA'] * 100).round(2)
```

Creates a new column, FT%, in player_free_throws by dividing the FTM column values by the corresponding FTA values, multiplying by 100, and rounding to two decimal places.

```
player_pbp = pd.concat([player_pbp, player_free_throws], axis=0)
```

Concatenates the DataFrame player_free_throws with player_pbp along the rows and assigns the DataFrame to player_pbp.

```
player_pbp.update(player_free_throws[['FTM', 'FTA', 'FT%']])
```

Updates the player_pbp DataFrame with values from the FTM, FTA, and FT% columns of the player_free_throws DataFrame.

```
player_pbp.reset_index(inplace=True)
```

Reset the index to make the current index a column.

```
player_pbp.drop_duplicates(subset='index', inplace=True)
```

Drop duplicate rows based on the index column.

```
player_pbp.drop(columns=['index'], inplace=True)
```

Drop the additional index column created by reset_index.

```
player_pbp
```

Print the updated player_pbp DataFrame.

```
subs_list = gamerotation.GameRotation(game_id= game_id)
```

Instantiates an object of a class named GameRotation from the gamerotation endpoint. Finds the substitution times for the game with the given game ID and stores it in a variable, subs_list.

```
subs_list_df = pd.DataFrame()
```

Creates an empty DataFrame, subs_list_df.

```
if home_boolean:
```

If home_boolean is True, executes the following code.

> ```
> subs_list_df = subs_list.data_sets[1].get_data_frame()
> ```
>
> Retrieves the game rotation dataset with index 1, which corresponds with the home team, and assigns it to subs_list_df.

```
else:
```

If home_boolean is False, executes the following code.

> ```
> subs_list_df = subs_list.data_sets[0].get_data_frame()
> ```
>
> Retrieves the game rotation dataset with index 0, which corresponds with the away team, and assigns it to subs_list_df.

```
subs_list_df = subs_list_df[subs_list_df["PERSON_ID"] == player_id]
```

Indexes subs_list_df for rows where the PERSON_ID column value equals the variable player_id. Assigns the result back to subs_df.

```
subs_list_df["IN_TIME_REAL"] = subs_list_df["IN_TIME_REAL"] / 10
```

Divides the values of the IN_TIME_REAL column of subs_list_df by 10.

```
subs_list_df["OUT_TIME_REAL"] = subs_list_df["OUT_TIME_REAL"] / 10
```

Divides the values of the OUT_TIME_REAL column of subs_list_df by 10.

```
subs_list_df
```

Prints the updated subs_list_df DataFrame.

```
bench_tuples = []
```

Create new empty list in variable bench_tuples. It will be filled with tuples representing time the player is not on the court.

```
for i in range(len(subs_list_df)):
```

Begin looping and increment variable i from 0 to the number of rows in the subs_list data frame minus 1.

```
    curr_row = subs_list_df.iloc[i]
```

Set new varialbe curr_row to be the current row in the loop.

```
    if i == 0 and curr_row["IN_TIME_REAL"] != 0.0:
        bench_tuples.append((0.0, curr_row["IN_TIME_REAL"]))
```

If the player's first time being subbed in is not at the beggining of the game (time 0.0), add a tuple from 0.0 to the first time they are subbed in to the variable bench_tuples.

```
    if i != len(subs_list_df)-1:
        next_row = subs_list_df.iloc[i+1]
```

If we are not currently on the player's last sub, set new variable next_row to be the next row in the loop.

```
        bench_tuples.append((curr_row["OUT_TIME_REAL"],next_row["IN_TIME_REAL"]))
```

Add a tuple from the current sub's sub out time until the next sub's sub in time.

```
    elif curr_row["OUT_TIME_REAL"] != 2880.0:
        bench_tuples.append((curr_row["OUT_TIME_REAL"], 2880.0))
```

Otherwise, if the current sub did not last until the end of the 4th quater (time 2880.0) add a tuple from the current sub's checkout time until 2880.0.

```
player_pbp['ON_COURT'] = 0
```

Initialize the ON_COURT column with zeros.

```
player_pbp['MIN_PLAYED'] = player_pbp['SECONDS_ELAPSED']
```

Initialize the MIN_PLAYED column with SECONDS_ELAPSED column values.

```
for index, row in player_pbp.iterrows():
```

Iterate over each row in player_pbp DataFrame.

```
    current_time = row['SECONDS_ELAPSED']
```

Assign the current time in seconds to current_time.

```
    if any((subs_list_df['IN_TIME_REAL'] <= current_time) & (subs_list_df['OUT_TIME_REAL'] >
        ↪ current_time)):
```

Check if the player was on the court at this time.

```
        player_pbp.at[index, 'ON_COURT'] = 1
```

Assign the rows that satisfy the condition a value of 1 in the ON_COURT column.

```
    if player_pbp.at[index, 'ON_COURT'] == 1 and index > 0:
```

Checks if the ON_COURT value is 1 and the index is greater than 0.

```
        prev_time = player_pbp.at[index - 1, 'SECONDS_ELAPSED']
```

If the previous condition is met, assign the SECONDS_ELAPSED value of the previous row to the variable prev_time.

```
        player_pbp.at[index, 'MIN_PLAYED'] = player_pbp.at[index - 1, 'MIN_PLAYED'] + (current_time -
            ↪ prev_time)
```

Updates MIN_PLAYED by adding the difference of the current and previous time to the MIN_PLAYED value of the previous row.

```
    elif index > 0:
```

If the index is greater than 0, execute the following code.

```
player_pbp.at[index, 'MIN_PLAYED'] = player_pbp.at[index - 1, 'MIN_PLAYED']
```

If player is not on the court, keep MIN_PLAYED value the same as the previous row.

```
player_pbp['ON_COURT'] = player_pbp['ON_COURT'].astype(bool)
```

Convert ON_COURT column to boolean.

```
player_pbp['MIN_PLAYED'] = (player_pbp['MIN_PLAYED'] / 60).round(3)
```

Convert MIN_PLAYED column to minutes.

```
box_advanced = boxscoreadvancedv2.BoxScoreAdvancedV2(game_id=game_id)
```

Instantiates an object of a class named BoxScoreAdvancedV2 from the boxscoreadvancedv2 endpoint. Finds the advanced box score statistics for the game with the given game ID and stores it in a variable, box_advanced.

```
player_box_advanced_df = box_advanced.get_data_frames()[1]
```

Access the second dataset of the boxscoreadvancedv2 endpoint, which gives advanced stats for teams. Assign the resulting DataFrame to player_box_advanced_df.

```
if home_boolean:
```

If home_boolean is True, execute the following code.

```
    advanced_stat_index = 0
```

Create a variable advanced_stat_index and assign it a value of 0.

```
else:
```

If home_boolean is False, execute the following code.

```
    advanced_stat_index = 1
```

Create a variable advanced_stat_index and assign it a value of 1.

```
player_box_advanced_df = player_box_advanced_df[player_box_advanced_df['TEAM_ABBREVIATION'] ==
    ↪ player_pbp['PLAYER1_TEAM_ABBREVIATION'].iloc[advanced_stat_index]]
```

Index player_box_advanced_df to find the row with a TEAM_ABBREVIATION column value equal to the PLAYER1_TEAM_ABBREVIATION in player_pbp at the row index stored in advanced_stat_index.

```
player_pbp['PACE'] = player_box_advanced_df['PACE'].iloc[0]
```

Index player_box_advanced to find the first row's PACE column value. Create a new PACE column in player_pbp and assign the pace value to this column.

```
player_pbp.iloc[0] = player_pbp.iloc[0].fillna(0)
```

Set stats that are NaN to 0 in the first row of the DataFrame

```
stat_columns = ['TEAM_FGM', 'TEAM_AST', 'PACE', 'MIN_PLAYED', 'PTS', 'AST', 'OREB', 'DREB', 'REB', 'STL',
    ↪ 'BLK', 'TO', 'PF', 'FGM', 'FGA', 'FG%', '3PTA', '3PTM', '3PT%', 'FTA', 'FTM', 'FT%']
```

Create a list, stat_columns, containing the stats we'll need to compute advanced stats later.

```
player_pbp[stat_columns] = player_pbp[stat_columns].ffill()
```

Forward fill the columns in player_pbp whose names are stored in stat_columns. This fills null values in these columns with the last non-null value in the column.

```
player_pbp[stat_columns] = player_pbp[stat_columns].apply(pd.to_numeric, errors='coerce')
```

Convert stat_columns column types to numeric. The errors='coerce' parameter means pandas will attempt to find a suitable value for any values that cause an error. The value will likely be NaN (not a number).

```
player_pbp
```

Prints the updated player_pbp.

```
season_game_finder = leaguegamefinder.LeagueGameFinder(season_nullable= season, league_id_nullable= '00')
```

Instantiates an object of a class named LeagueGameFinder from the leaguegamefinder endpoint. Finds the ending box score statistics for every game in the season stored in the variable, season. A league id of '00' represents the NBA (as opposed to G-league, etc.). The result is stored in a variable, season_game_finder.

```
season_games = season_game_finder.get_data_frames()[0]
```

Retrieves the DataFrames for the only dataset in the LeagueGameFinder endpoint (index 0), and assigns it to a season_games. This DataFrame contains the game statistics for all games in a season.

```
season_league_averages = season_games[['PTS', 'FGM', 'FGA', 'FG3M', 'FG3A', 'FTM', 'FTA', 'OREB', 'REB',
    ↪ 'AST', 'TOV', 'PF']].mean()
```

Selects various stat columns of season_games and calculates the mean of these stats across all games in a season. Assigns the results to season_league_averages.

```
season_league_averages = pd.DataFrame(season_league_averages)
```

Converts season_league_averages to a DataFrame.

```
season_league_averages = season_league_averages.T
```

Transposes season_league_averages, switching the positions of rows and columns. (Rows become columns, and columns become rows.) Assigns the result back to season_league_averages.

```
season_league_averages.apply(pd.to_numeric)
```

Convert season_league_averages column types to numeric.

```
PTS = player_pbp['PTS']
```

Assign the PTS column of player_pbp to a variable, PTS.

```
AST = player_pbp['AST']
```

Assign the AST column of player_pbp to a variable, AST.

```
OREB = player_pbp['OREB']
```

Assign the OREB column of player_pbp to a variable, OREB.

```
DREB = player_pbp['DREB']
```

Assign the DREB column of player_pbp to a variable, DREB.

```
REB = player_pbp['REB']
```

Assign the REB column of player_pbp to a variable, REB.

```
STL = player_pbp['STL']
```

Assign the STL column of player_pbp to a variable, STL.

```
BLK = player_pbp['BLK']
```

Assign the BLK column of player_pbp to a variable, BLK.

```
TO = player_pbp['TO']
```

Assign the TO column of player_pbp to a variable, TO.

```
PF = player_pbp['PF']
```

Assign the PF column of player_pbp to a variable, PF.

```
FGM = player_pbp['FGM']
```

Assign the FGM column of player_pbp to a variable, FGM.

```
FGA = player_pbp['FGA']
```

Assign the FGA column of player_pbp to a variable, FGA.

```
FTM = player_pbp['FTM']
```

Assign the FTM column of player_pbp to a variable, FTM.

```
FTA = player_pbp['FTA']
```

Assign the FTA column of player_pbp to a variable, FTA.

```
FG3M = player_pbp['3PTM']
```
Assign the 3PTM column of player_pbp to a variable, FG3M.

```
MIN = player_pbp['MIN_PLAYED']
```
Assign the MIN_PLAYED column of player_pbp to a variable, MIN.

```
PACE = player_pbp['PACE']
```
Assign the PACE column of player_pbp to a variable, PACE.

```
GAME_PTS = player_pbp['GAME_PTS']
```
Assign the GAME_PTS column of player_pbp to a variable, GAME_PTS.

```
GAME_AST = player_pbp['GAME_AST']
```
Assign the GAME_AST column of player_pbp to a variable, GAME_AST.

```
GAME_REB = player_pbp['GAME_REB']
```
Assign the GAME_REB column of player_pbp to a variable, GAME_REB.

```
GAME_STL = player_pbp['GAME_STL']
```
Assign the GAME_STL column of player_pbp to a variable, GAME_STL.

```
GAME_BLK = player_pbp['GAME_BLK']
```
Assign the GAME_BLK column of player_pbp to a variable, GAME_BLK.

```
GAME_TO = player_pbp['GAME_TO']
```
Assign the GAME_TO column of player_pbp to a variable, GAME_TO.

```
GAME_PF = player_pbp['GAME_PF']
```
Assign the GAME_PF column of player_pbp to a variable, GAME_PF.

```
GAME_FGM = player_pbp['GAME_FGM']
```
Assign the GAME_FGM column of player_pbp to a variable, GAME_FGM.

```
GAME_FGA = player_pbp['GAME_FGA']
```
Assign the GAME_FGA column of player_pbp to a variable, GAME_FGA.

```
GAME_FTM = player_pbp['GAME_FTM']
```
Assign the GAME_FTM column of player_pbp to a variable, GAME_FTM.

```
GAME_FTA = player_pbp['GAME_FTA']
```
Assign the GAME_FTA column of player_pbp to a variable, GAME_FTA.

```
eFG = []
```
Create an empty list, eFG.

```
for index, row in player_pbp.iterrows():
```
Iterate through the rows of player_pbp.

```
    if row['FGA'] == 0:
```
If the row's FGA column value equals 0, execute the following code.

```
        eFG.append(0)
```
Append 0 to the eFG list.

```
    else:
```
If the row's FGA column value does not equal 0, execute the following code.

```
        eFG.append((row['FGM'] + 0.5 * row['3PTM']) / row['FGA'])
```
Implement the effective field goal percentage formula on the row. Append the result to the eFG list.

```
player_pbp['eFG%'] = [round(efg * 100, 2) for efg in eFG]
```
Creates a new column, eFG% in player_pbp by multiplying each value in eFG by 100, rounding to two decimal places, and assigning the resulting values to the column.

```
player_pbp['PTS'] = player_pbp['PTS'].apply(pd.to_numeric, errors='coerce')
```
Convert PTS column of player_pbp to numeric.

```
TS_percentages = []
```
Create an empty list, TS_percentages.

```
for index, row in player_pbp.iterrows():
```
Iterate through the rows of player_pbp.

```
    if row['FGA'] == 0 and row['FTA'] == 0:
```
If the row's FGA and FTA column values both equal 0, execute the following code.

```
        TS_percentages.append(0)
```
Append 0 to the TS_percentages list.

```
    else:
```
If the row's FGA and FTA column values are not both equal to 0, execute the following code.

```
        TS_percentages.append(row['PTS'] / (2 * (row['FGA'] + 0.44 * row['FTA'])))
```
Implement the true shooting percentage formula on the row. Append the result to the TS_percentages list.

```
player_pbp['TS%'] = [round(ts * 100, 2) for ts in TS_percentages]
```
Creates a new column, TS% in player_pbp by multiplying each value in TS_percentages by 100, rounding to two decimal places, and assigning the resulting values to the column.

```
lgPTS = season_league_averages['PTS'].iloc[0]
```
Assign the PTS column of season_league_averages to a variable, lgPTS.

```
lgFGM = season_league_averages['FGM'].iloc[0]
```
Assign the FGM column of season_league_averages to a variable, lgFGM.

```
lgFGA = season_league_averages['FGA'].iloc[0]
```
Assign the FGA column of season_league_averages to a variable, lgFGA.

```
lgFG3M = season_league_averages['FG3M'].iloc[0]
```
Assign the FG3M column of season_league_averages to a variable, lgFG3M.

```
lgFG3A = season_league_averages['FG3A'].iloc[0]
```
Assign the FG3A column of season_league_averages to a variable, lgFG3A.

```
lgFTM = season_league_averages['FTM'].iloc[0]
```
Assign the FTM column of season_league_averages to a variable, lgFTM.

```
lgFTA = season_league_averages['FTA'].iloc[0]
```
Assign the FTA column of season_league_averages to a variable, lgFTA.

```
lgOREB = season_league_averages['OREB'].iloc[0]
```
Assign the OREB column of season_league_averages to a variable, lgOREB.

```
lgREB = season_league_averages['REB'].iloc[0]
```
Assign the REB column of season_league_averages to a variable, lgREB.

```
lgAST = season_league_averages['AST'].iloc[0]
```
Assign the AST column of season_league_averages to a variable, lgAST.

```
lgTO = season_league_averages['TOV'].iloc[0]
```
Assign the TO column of season_league_averages to a variable, lgTO.

```
lgPF = season_league_averages['PF'].iloc[0]
```
Assign the PF column of season_league_averages to a variable, lgPF.

```
tmAST = player_pbp['TEAM_AST']
```
Assign the TEAM_AST column of player_pbp to a variable, tmAST.

```
tmFGM = player_pbp['TEAM_FGM']
```
Assign the TEAM_FGM column of player_pbp to a variable, tmFGM.

```
factor = (2/3) - ((.5 * (lgAST / lgFGM)) / (2 * (lgFGM / lgFTM)))
```
Implement factor formula and assign the result to a variable, factor.

```
VOP = lgPTS / (lgFGA - lgOREB + lgTO + 0.44 * lgFTA)
```
Implement value of possession formula and assign the result to a variable, VOP.

```
DRBP = (lgREB - lgOREB) / lgREB
```
Implement defensive rebound percentage formula and assign the result to a variable, DRBP.

```
player_pbp['PER'] = (100 / MIN) * (FG3M - ((PF * lgFTM) / lgPF) + ((FTM / 2) * (2 - (tmAST/(3 * tmFGM))))
    ↪ + (FGM * (2 - ((factor * tmAST) / tmFGM))) + ((2 * AST) / 3) + VOP * (DRBP * (2 * OREB + BLK -
    ↪ 0.2464 * (FTA - FTM) - (FGA - FGM) - REB) + ((0.44 * lgFTA * PF) / lgPF) - TO - OREB + STL + REB -
    ↪ 0.1936 * (FTA - FTM)))
```
Implement unadjusted player efficiency rating formula. Assign resulting values to a new PER column of player_pbp.

```
player_pbp['PIE'] = 100 * (PTS + FGM + FTM - FGA - FTA + DREB + OREB/2 + AST + STL + BLK/2 - PF - TO) /
    ↪ (GAME_PTS + GAME_FGM + GAME_FTM - GAME_FGA - GAME_FTA + (1.5 * GAME_REB) + GAME_AST + GAME_STL +
    ↪ GAME_BLK/2 - GAME_PF - GAME_TO)
```
Implement player impact estimate formula. Assign resulting values to a new PIE column of player_pbp.

```
player_pbp['TENDEX'] = (PTS + OREB + DREB + AST + STL + BLK - (FGA - FGM) - (0.5 * (FTA - FTM)) - TO - PF)
    ↪ / (MIN * PACE)
```
Implement tendex formula. Assign resulting values to a new TENDEX column of player_pbp.

```
player_pbp.iloc[0] = player_pbp.iloc[0].fillna(0)
```
Fills NA values in the first row of player_pbp with 0.

```
player_pbp.ffill(inplace = True)
```
Forward fills the values in player_pbp, filling NA values with the last non-null value in the column. Makes changes inplace, directly in the DataFrame, rather than making a copy.

```
player_pbp
```
Prints the updated DataFrame.

1.4 Plotting Stats as a Function of Time

1.4.1 Plotting a Traditional Stat

```
player_pbp_on_court = player_pbp[player_pbp['ON_COURT']]
```
Creates a new DataFrame by filtering the player_pbp DataFrame for rows where the 'ON_COURT' column value is True. The new DataFrame only includes plays for which the given player was on the court, and it is assigned to a variable, player_pbp_on_court.

```
player_pbp_off_court = player_pbp[~player_pbp['ON_COURT']]
```

Creates a new DataFrame by filtering the player_pbp DataFrame for rows where the 'ON_COURT' column value is False. The ~(tilde) is the "not" operator, which negates the boolean values in the 'ON_COURT' column. The new DataFrame only includes plays for which the given player was on the bench, and it is assigned to a variable, player_pbp_off_court.

```
max_seconds = player_pbp['SECONDS_ELAPSED'].max()
```

Creates a variable, max_seconds, equal to the maximum value of the 'SECONDS_ELAPSED' column of the player_pbp DataFrame.

```
num_quarters = max_seconds // (12 * 60)
```

Calculates the number of quarters in the given game by dividing max_seconds by the number of seconds in a quarter. Using floor division (//), the quotient is rounded down to the nearest integer. The result is assigned to a variable, num_quarters.

```
ticks = np.arange(0, max_seconds + 1, 12 * 60)
```

Uses the arange function to create an array of values, with one value for every 12 minutes within the range from 0 to max_seconds, and assigns the array to a variable, ticks.

```
plt.xticks(ticks, ['Q1 Start'] + [f'End Q\{i\}' for i in range(1, int(num_quarters) + 1)])
```

Creates tick labels for the plot's x-axis. The positions are provided by the ticks array. 'Q1 Start' is a static label for the first tick. A list comprehension generates labels for the remaining ticks by iterating over each quarter number from 1 to num_quarters, creating a label for each quarter in the format 'End Q{i}'.

```
basic_stat = input('Enter a basic stat (PTS, AST, OREB, DREB, REB, STL, BLK, TO, PF) to create a flow
    ↪ graph of: ')
```

Prompts the user to enter a basic stat abbreviation.

```
x_on_court = player_pbp_on_court['SECONDS_ELAPSED']
```

Assigns the SECONDS_ELAPSED column of the player_pbp_on_court DataFrame to a variable, x_on_court.

```
y_on_court = player_pbp_on_court[basic_stat]
```

Assigns the column of the player_pbp_on_court DataFrame with the same name as the basic_stat variable to a new variable, y_on_court.

```
plt.plot(x_on_court, y_on_court, linestyle='-', label='On Court')
```

Generates a plot using x_on_court values as the x-coordinates, and y_on_court values as the y-coordinates. Also creates a label which will be used in the plot's legend to identify the line.

```
label = 'On Bench'
```

Creates a variable, label.

```
for e in bench_tuples:
```

Begins iteration on each tuple in bench_tuples.

```
    plt.plot([e[0], e[1]], [np.interp(e[0], x_on_court, y_on_court), np.interp(e[1], x_on_court,
        ↪ y_on_court)], linestyle='-', color='red', label= label)
```

Plots a line between two points [e[0], e[1]] with y−values interpolated from the data points (x_on_court, y_on_court). Sets the line style to solid, the color to red, and assigns a label to the plot.

```
    label = "_nolegend_"
```

Tells matplotlib to exclude the variable, label, from the legend.

```
x = player_pbp['SECONDS_ELAPSED']
```

Assigns the SECONDS_ELAPSED column of the player_pbp DataFrame to a variable, x.

```
plt.xlim(x.min(), x.max())
```

Sets the lower and upper x-axis limits of the plot to the minimum and maximum values of x respectively.

```
plt.ylim(0, y_on_court.max() + 2)
```

Sets the lower y-axis limit of the plot to 0 and the upper y-axis limit to the maximum value of y_on _court plus two. A value of two is added to the upper y-axis limit to ensure that the plot is not cut off visually.

`plt.xlabel('Game Time')`

Creates a label for the x-axis of the plot.

`plt.ylabel(f' {basic_stat}')`

Generates a formatted string using the basic_stat input value for the y-axis label of the plot.

`plt.title(f' Progression of {basic_stat} for {player_name}')`

Generates a formatted string for the plot title using the basic_stat and player_name input values.

`plt.legend()`

Generates a legend for the plot that indicates how to differentiate between a player being on the court versus on the bench.

`plt.grid(True)`

Calls the grid function of Matplotlib, which sets visible grid lines for the plot.

`plt.show()`

Displays the plot created.

1.4.2 Plotting Shooting Percentages

`plt.xticks(ticks, ['Q1 Start'] + [f'End Q{i}' for i in range(1, int(num_quarters) + 1)])`

Creates tick labels for the plot's x-axis. The positions are provided by the ticks array. 'Q1 Start' is a static label for the first tick. A list comprehension generates labels for the remaining ticks by iterating over each quarter number from 1 to num_quarters, creating a label for each quarter in the format 'End Q{i}'.

`x_on_court = player_pbp_on_court['SECONDS_ELAPSED']`

Assigns the SECONDS_ELAPSED column of the player_pbp_on_court DataFrame to a variable, x_on_court.

`y_on_court = player_pbp_on_court['FT%']`

Assigns the FT% column of the player_pbp_on_court DataFrame to a new variable, y_on_court.

`plt.plot(x_on_court, y_on_court, linestyle='-', label='On Court')`

Generates a plot using x_on_court values as the x-coordinates, and y_on_court values as the y-coordinates. Also creates a label which will be used in the plot's legend to identify the line.

`label = 'On Bench'`

Creates a variable, label.

`for e in bench_tuples:`

Begins iteration on each tuple in bench_tuples.

```
plt.plot([e[0], e[1]], [np.interp(e[0], x_on_court, y_on_court), np.interp(e[1], x_on_court,
    ↪ y_on_court)], linestyle='-', color='red', label= label)
```

Plots a line between two points [e [0], e [1]] with y—values interpolated from the data points (x_on_court, y_on_court). Sets the line style to solid, the color to red, and assigns a label to the plot.

`label = "_nolegend_"`

Tells matplotlib to exclude the variable, label, from the legend.

`x = player_pbp['SECONDS_ELAPSED']`

Assigns the SECONDS_ELAPSED column of player_pbp to a variable, x.

`plt.xlim(x.min(), x.max())`

Sets the lower and upper x-axis limits of the plot to the minimum and maximum values of x respectively.

`plt.ylim(0, y_on_court.max() + 2)`

Sets the lower y-axis limit of the plot to 0 and the upper y-axis limit to the maximum value of y_on _court plus two. A value of two is added to the upper y-axis limit to ensure that the plot is not cut off visually.

`plt.xlabel('Game Time')`

Creates a label for the x-axis of the plot.

`plt.ylabel('Free Throw Percentage')`

Creates a label for the plot's y-axis.

`plt.title(f'Free Throw Percentage Over the Game for {player_name}')`

Generates a formatted string for the plot title using the player_name input value.

`plt.legend()`

Generates a legend for the plot that indicates how to differentiate between a player being on the court versus on the bench.

`plt.grid(True)`

Calls the grid function of Matplotlib, which sets visible grid lines for the plot.

`plt.show()`

Displays the plot created.

`plt.xticks(ticks, ['Q1 Start'] + [f'End Q{i}' for i in range(1, int(num_quarters) + 1)])`

Creates tick labels for the plot's x-axis. The positions are provided by the ticks array. 'Q1 Start' is a static label for the first tick. A list comprehension generates labels for the remaining ticks by iterating over each quarter number from 1 to num_quarters, creating a label for each quarter in the format 'End Q{i}'.

`x_on_court = player_pbp_on_court['SECONDS_ELAPSED']`

Assigns the SECONDS_ELAPSED column of the player_pbp_on_court DataFrame to a variable, x_on_court.

`y_on_court = player_pbp_on_court['FG%']`

Assigns the FG% column of the player_pbp_on_court DataFrame to a new variable, y_on_court.

`plt.plot(x_on_court, y_on_court, linestyle='-', label='On Court')`

Generates a plot using x_on_court values as the x-coordinates, and y_on_court values as the y-coordinates. Also creates a label which will be used in the plot's legend to identify the line.

`label = 'On Bench'`

Creates a variable, label.

`for e in bench_tuples:`

Begins iteration on each tuple in bench_tuples.

```
plt.plot([e[0], e[1]], [np.interp(e[0], x_on_court, y_on_court), np.interp(e[1], x_on_court,
    ↪ y_on_court)], linestyle='-', color='red', label= label)
```

Plots a line between two points [e [0], e [1]] with y−values interpolated from the data points (x_on_court, y_on_court). Sets the line style to solid, the color to red, and assigns a label to the plot.

```
label = "_nolegend_"
```

Tells matplotlib to exclude the variable, label, from the legend.

`x = player_pbp['SECONDS_ELAPSED']`

Assigns the SECONDS_ELAPSED column of player_pbp to a variable, x.

`plt.xlim(x.min(), x.max())`

Sets the lower and upper x-axis limits of the plot to the minimum and maximum values of x respectively.

`plt.ylim(0, y_on_court.max() + 2)`

Sets the lower y-axis limit of the plot to 0 and the upper y-axis limit to the maximum value of y_on _court plus two. A value of two is added to the upper y-axis limit to ensure that the plot is not cut off visually.

```
plt.xlabel('Game Time')
```

Creates a label for the x-axis of the plot.

```
plt.ylabel('Field Goal Percentage')
```

Creates a label for the plot's y-axis.

```
plt.title(f'Field Goal Percentage Over the Game for {player_name}')
```

Generates a formatted string for the plot title using the player_name input value.

```
plt.legend()
```

Generates a legend for the plot that indicates how to differentiate between a player being on the court versus on the bench.

```
plt.grid(True)
```

Calls the grid function of Matplotlib, which sets visible grid lines for the plot.

```
plt.show()
```

Displays the plot created.

```
plt.xticks(ticks, ['Q1 Start'] + [f'End Q{i}' for i in range(1, int(num_quarters) + 1)])
```

Creates tick labels for the plot's x-axis. The positions are provided by the ticks array. 'Q1 Start' is a static label for the first tick. A list comprehension generates labels for the remaining ticks by iterating over each quarter number from 1 to num_quarters, creating a label for each quarter in the format 'End Q{i}'.

```
x_on_court = player_pbp_on_court['SECONDS_ELAPSED']
```

Assigns the SECONDS_ELAPSED column of the player_pbp_on_court DataFrame to a variable, x_on_court.

```
y_on_court = player_pbp_on_court['3PT%']
```

Assigns the 3PT% column of the player_pbp_on_court DataFrame to a new variable, y_on_court.

```
plt.plot(x_on_court, y_on_court, linestyle='-', label='On Court')
```

Generates a plot using x_on_court values as the x-coordinates, and y_on_court values as the y-coordinates. Also creates a label which will be used in the plot's legend to identify the line.

```
label = 'On Bench'
```

Creates a variable, label.

```
for e in bench_tuples:
```

Begins iteration on each tuple in bench_tuples.

```
    plt.plot([e[0], e[1]], [np.interp(e[0], x_on_court, y_on_court), np.interp(e[1], x_on_court,
        ↪ y_on_court)], linestyle='-', color='red', label= label)
```

Plots a line between two points [e [0], e [1]] with y−values interpolated from the data points (x_on_court, y_on_court). Sets the line style to solid, the color to red, and assigns a label to the plot.

```
    label = "_nolegend_"
```

Tells matplotlib to exclude the variable, label, from the legend.

```
x = player_pbp['SECONDS_ELAPSED']
```

Assigns the SECONDS_ELAPSED column of player_pbp to a variable, x.

```
plt.xlim(x.min(), x.max())
```

Sets the lower and upper x-axis limits of the plot to the minimum and maximum values of x respectively.

```
plt.ylim(0, y_on_court.max() + 2)
```

Sets the lower y-axis limit of the plot to 0 and the upper y-axis limit to the maximum value of y_on _court plus two. A value of two is added to the upper y-axis limit to ensure that the plot is not cut off visually.

`plt.xlabel('Game Time')`

Creates a label for the x-axis of the plot.

`plt.ylabel('Three Point Percentage')`

Creates a label for the plot's y-axis.

`plt.title(f'Three Point Percentage Over the Game for {player_name}')`

Generates a formatted string for the plot title using the player_name input value.

`plt.legend()`

Generates a legend for the plot that indicates how to differentiate between a player being on the court versus on the bench.

`plt.grid(True)`

Calls the grid function of Matplotlib, which sets visible grid lines for the plot.

`plt.show()`

Displays the plot created.

1.4.3 Plotting Field Goal Percentage Over a Season

`games[['FG_PCT', 'FG3_PCT', 'FT_PCT']] *= 100`

Multiplies the FG_PCT, FG3_PCT, and FT_PCT columns of the games DataFrame by 100.

`games.fillna(0, inplace = True)`

Fills null values of the games DataFrame with 0, making changes directly in the original DataFrame, rather than making a new one.

`games`

Prints the updated games DataFrame.

`x = games.index`

Assigns the index of games to a variable, x.

`y = games['FG_PCT']`

Assigns the FG_PCT column of games to a variable, y.

`plt.plot(x, y)`

Generates a plot using x as the x-coordinates, and y as the y-coordinates.

`plt.xlabel('Games Played')`

Creates a label for the x-axis of the plot.

`plt.ylabel('FG%')`

Creates a label for the plot's y-axis.

`plt.title(f'FG% by Game for {player_name} in the {season} season')`

Generates a formatted string for the plot title using the player_name and season input values.

`plt.xlim(x.min(), x.max())`

Sets the lower and upper x-axis limits of the plot to the minimum and maximum values of x respectively.

`plt.ylim(0, y.max() + 5)`

Sets the lower y-axis limit of the plot to 0 and the upper y-axis limit to the maximum values in y plus five (to ensure that the plot is not cut off visually).

`plt.grid(True)`

Calls the grid function of Matplotlib, which sets visible grid lines for the plot.

`plt.show()`

Displays the plot created.

`games['CUMULATIVE_FG_PCT'] = (games['FGM'].cumsum() / games['FGA'].cumsum() * 100).round(2)`

Creates a new column, CUMULATIVE_FG_PCT, in games by dividing the cumulative sum of FGM column values by the cumulative sum of FGA values, multiplying by 100, and rounding to two decimal places.

`games['CUMULATIVE_FG3_PCT'] = (games['FG3M'].cumsum() / games['FG3A'].cumsum() * 100).round(2)`

Creates a new column, CUMULATIVE_FG3_PCT, in games by dividing the cumulative sum of FG3M column values by the cumulative sum of FG3A values, multiplying by 100, and rounding to two decimal places.

`games['CUMULATIVE_FT_PCT'] = (games['FTM'].cumsum() / games['FTA'].cumsum() * 100).round(2)`

Creates a new column, CUMULATIVE_FT_PCT, in games by dividing the cumulative sum of FTM column values by the cumulative sum of FTA values, multiplying by 100, and rounding to two decimal places.

`games`

Prints the updated games DataFrame.

`x = games.index`

Assigns the index of games to a variable, x.

`y = games['CUMULATIVE_FG_PCT']`

Assigns the CUMULATIVE_FG_PCT column of games to a variable, y.

`plt.plot(x, y)`

Generates a plot using x as the x-coordinates, and y as the y-coordinates.

`plt.xlabel('Games Played')`

Creates a label for the x-axis of the plot.

`plt.ylabel('Cumulative FG%')`

Creates a label for the plot's y-axis.

`plt.title(f'Cumulative FG% for {player_name} in the {season} season')`

Generates a formatted string for the plot title using the player_name and season input values.

`plt.xlim(x.min(), x.max())`

Sets the lower and upper x-axis limits of the plot to the minimum and maximum values of x respectively.

`plt.ylim(0, y.max() + 5)`

Sets the lower y-axis limit of the plot to 0 and the upper y-axis limit to the maximum value in y plus five (to ensure that the plot is not cut off visually).

`plt.grid(True)`

Calls the grid function of Matplotlib, which sets visible grid lines for the plot.

`plt.show()`

Displays the plot created.

1.4.4 Plotting Advanced Stats

`plt.xticks(ticks, ['Q1 Start'] + [f'End Q{i}' for i in range(1, int(num_quarters) + 1)])`

Creates tick labels for the plot's x-axis. The positions are provided by the ticks array. 'Q1 Start' is a static label for the first tick. A list comprehension generates labels for the remaining ticks by iterating over each quarter number from 1 to num_quarters, creating a label for each quarter in the format 'End Q{i}'.

```
x_on_court = player_pbp_on_court['SECONDS_ELAPSED']
```
Assigns the SECONDS_ELAPSED column of the player_pbp_on_court DataFrame to a variable, x_on_court.

```
y_on_court = player_pbp_on_court['eFG%']
```
Assigns the eFG% column of the player_pbp_on_court DataFrame to a new variable, y_on_court.

```
plt.plot(x_on_court, y_on_court, linestyle='-', label='On Court')
```
Generates a plot using x_on_court values as the x-coordinates, and y_on_court values as the y-coordinates. Also creates a label which will be used in the plot's legend to identify the line.

```
label = 'On Bench'
```
Creates a variable, label.

```
for e in bench_tuples:
```
Begins iteration on each tuple in bench_tuples.

```
    plt.plot([e[0], e[1]], [np.interp(e[0], x_on_court, y_on_court), np.interp(e[1], x_on_court,
        ↪ y_on_court)], linestyle='-', color='red', label= label)
```
Plots a line between two points [e [0], e [1]] with y—values interpolated from the data points (x_on_court, y_on_court). Sets the line style to solid, the color to red, and assigns a label to the plot.

```
    label = "_nolegend_"
```
Tells matplotlib to exclude the variable, label, from the legend.

```
x = player_pbp['SECONDS_ELAPSED']
```
Assigns the SECONDS_ELAPSED column of player_pbp to a variable, x.

```
plt.xlim(x.min(), x.max())
```
Sets the lower and upper x-axis limits of the plot to the minimum and maximum values of x respectively.

```
plt.ylim(0, y_on_court.max() + 2)
```
Sets the lower y-axis limit of the plot to 0 and the upper y-axis limit to the maximum value of y_on _court plus two. A value of two is added to the upper y-axis limit to ensure that the plot is not cut off visually.

```
plt.xlabel('Game Time')
```
Creates a label for the x-axis of the plot.

```
plt.ylabel('eFG%')
```
Creates a label for the plot's y-axis.

```
plt.title(f'Effective FG% Over the Game for {player_name}')
```
Generates a formatted string for the plot title using the player_name input value.

```
plt.legend()
```
Generates a legend for the plot that indicates how to differentiate between a player being on the court versus on the bench.

```
plt.grid(True)
```
Calls the grid function of Matplotlib, which sets visible grid lines for the plot.

```
plt.show()
```
Displays the plot created.

```
plt.xticks(ticks, ['Q1 Start'] + [f'End Q{i}' for i in range(1, int(num_quarters) + 1)])
```
Creates tick labels for the plot's x-axis. The positions are provided by the ticks array. 'Q1 Start' is a static label for the first tick. A list comprehension generates labels for the remaining ticks by iterating over each quarter number from 1 to num_quarters, creating a label for each quarter in the format 'End Q{i}'.

```
x_on_court = player_pbp_on_court['SECONDS_ELAPSED']
```
Assigns the SECONDS_ELAPSED column of the player_pbp_on_court DataFrame to a variable, x_on_court.

```
y_on_court = player_pbp_on_court['TS%']
```
Assigns the TS% column of the player_pbp_on_court DataFrame to a new variable, y_on_court.

```
plt.plot(x_on_court, y_on_court, linestyle='-', label='On Court')
```
Generates a plot using x_on_court values as the x-coordinates, and y_on_court values as the y-coordinates. Also creates a label which will be used in the plot's legend to identify the line.

```
label = 'On Bench'
```
Creates a variable, label.

```
for e in bench_tuples:
```
Begins iteration on each tuple in bench_tuples.

```
    plt.plot([e[0], e[1]], [np.interp(e[0], x_on_court, y_on_court), np.interp(e[1], x_on_court,
        ↪ y_on_court)], linestyle='-', color='red', label= label)
```
Plots a line between two points [e [0], e [1]] with y—values interpolated from the data points (x_on_court, y_on_court). Sets the line style to solid, the color to red, and assigns a label to the plot.

```
    label = "_nolegend_"
```
Tells matplotlib to exclude the variable, label, from the legend.

```
x = player_pbp['SECONDS_ELAPSED']
```
Assigns the SECONDS_ELAPSED column of player_pbp to a variable, x.

```
plt.xlim(x.min(), x.max())
```
Sets the lower and upper x-axis limits of the plot to the minimum and maximum values of x respectively.

```
plt.ylim(0, y_on_court.max() + 2)
```
Sets the lower y-axis limit of the plot to 0 and the upper y-axis limit to the maximum value of y_on _court plus two. A value of two is added to the upper y-axis limit to ensure that the plot is not cut off visually.

```
plt.xlabel('Game Time')
```
Creates a label for the x-axis of the plot.

```
plt.ylabel('TS%')
```
Creates a label for the plot's y-axis.

```
plt.title(f'True Shooting % Over the Game for {player_name}')
```
Generates a formatted string for the plot title using the player_name input value.

```
plt.legend()
```
Generates a legend for the plot that indicates how to differentiate between a player being on the court versus on the bench.

```
plt.grid(True)
```
Calls the grid function of Matplotlib, which sets visible grid lines for the plot.

```
plt.show()
```
Displays the plot created.

```
plt.xticks(ticks, ['Q1 Start'] + [f'End Q{i}' for i in range(1, int(num_quarters) + 1)])
```
Creates tick labels for the plot's x-axis. The positions are provided by the ticks array. 'Q1 Start' is a static label for the first tick. A list comprehension generates labels for the remaining ticks by iterating over each quarter number from 1 to num_quarters, creating a label for each quarter in the format 'End Q{i}'.

```
x_on_court = player_pbp_on_court['SECONDS_ELAPSED']
```

Assigns the SECONDS_ELAPSED column of the player_pbp_on_court DataFrame to a variable, x_on_court.

```
y_on_court = player_pbp_on_court['PIE']
```

Assigns the PIE column of the player_pbp_on_court DataFrame to a new variable, y_on_court.

```
plt.plot(x_on_court, y_on_court, linestyle='-', label='On Court')
```

Generates a plot using x_on_court values as the x-coordinates, and y_on_court values as the y-coordinates. Also creates a label which will be used in the plot's legend to identify the line.

```
label = 'On Bench'
```

Creates a variable, label.

```
for e in bench_tuples:
```

Begins iteration on each tuple in bench_tuples.

```
    plt.plot([e[0], e[1]], [np.interp(e[0], x_on_court, y_on_court), np.interp(e[1], x_on_court,
        ↪ y_on_court)], linestyle='-', color='red', label= label)
```

Plots a line between two points [e[0], e[1]] with y—values interpolated from the data points (x_on_court, y_on_court). Sets the line style to solid, the color to red, and assigns a label to the plot.

```
    label = "_nolegend_"
```

Tells matplotlib to exclude the variable, label, from the legend.

```
x = player_pbp['SECONDS_ELAPSED']
```

Assigns the SECONDS_ELAPSED column of player_pbp to a variable, x.

```
plt.xlim(x.min(), x.max())
```

Sets the lower and upper x-axis limits of the plot to the minimum and maximum values of x respectively.

```
plt.ylim(y.min() - 20, y_on_court.max() + 2)
```

Sets the lower y-axis limit of the plot to 20 less than the minimum of y and the upper y-axis limit to the max of y_on_court plus two. A value of two is added to the upper y-axis limit to ensure that the plot is not cut off visually.

```
plt.xlabel('Game Time')
```

Creates a label for the x-axis of the plot.

```
plt.ylabel('PIE')
```

Creates a label for the plot's y-axis.

```
plt.title(f'Player Impact Estimate Over the Game for {player_name}')
```

Generates a formatted string for the plot title using the player_name input value.

```
plt.legend()
```

Generates a legend for the plot that indicates how to differentiate between a player being on the court versus on the bench.

```
plt.grid(True)
```

Calls the grid function of Matplotlib, which sets visible grid lines for the plot.

```
plt.show()
```

Displays the plot created.

```
plt.xticks(ticks, ['Q1 Start'] + [f'End Q{i}' for i in range(1, int(num_quarters) + 1)])
```

Creates tick labels for the plot's x-axis. The positions are provided by the ticks array. 'Q1 Start' is a static label for the first tick. A list comprehension generates labels for the remaining ticks by iterating over each quarter number from 1 to num_quarters, creating a label for each quarter in the format 'End Q{i}'.

```
x_on_court = player_pbp_on_court['SECONDS_ELAPSED']
```

Assigns the SECONDS_ELAPSED column of the player_pbp_on_court DataFrame to a variable, x_on_court.

```
y_on_court = player_pbp_on_court['PER']
```
Assigns the PER column of the player_pbp_on_court DataFrame to a new variable, y_on_court.

```
plt.plot(x_on_court, y_on_court, linestyle='-', label='On Court')
```
Generates a plot using x_on_court values as the x-coordinates, and y_on_court values as the y-coordinates. Also creates a label which will be used in the plot's legend to identify the line.

```
label = 'On Bench'
```
Creates a variable, label.

```
for e in bench_tuples:
```
Begins iteration on each tuple in bench_tuples.

> ```
> plt.plot([e[0], e[1]], [np.interp(e[0], x_on_court, y_on_court), np.interp(e[1], x_on_court,
>     ↪ y_on_court)], linestyle='-', color='red', label= label)
> ```
> Plots a line between two points [e [0], e [1]] with y—values interpolated from the data points (x_on_court, y_on_court). Sets the line style to solid, the color to red, and assigns a label to the plot.
>
> ```
> label = "_nolegend_"
> ```
> Tells matplotlib to exclude the variable, label, from the legend.

```
x = player_pbp['SECONDS_ELAPSED']
```
Assigns the SECONDS_ELAPSED column of player_pbp to a variable, x.

```
plt.xlim(x.min(), x.max())
```
Sets the lower and upper x-axis limits of the plot to the minimum and maximum values of x respectively.

```
plt.ylim(y.min() - 5, y_on_court.max() + 2)
```
Sets the lower y-axis limit of the plot to five less than the minimum of y and the upper y-axis limit to the maximum value of y_on _court. A value of two is added to the upper y-axis limit to ensure that the plot is not cut off visually.

```
plt.xlabel('Game Time')
```
Creates a label for the x-axis of the plot.

```
plt.ylabel('PER')
```
Creates a label for the plot's y-axis.

```
plt.title(f'Player Efficiency Rating Over the Game for {player_name}')
```
Generates a formatted string for the plot title using the player_name input value.

```
plt.legend()
```
Generates a legend for the plot that indicates how to differentiate between a player being on the court versus on the bench.

```
plt.grid(True)
```
Calls the grid function of Matplotlib, which sets visible grid lines for the plot.

```
plt.show()
```
Displays the plot created.

```
plt.xticks(ticks, ['Q1 Start'] + [f'End Q{i}' for i in range(1, int(num_quarters) + 1)])
```
Creates tick labels for the plot's x-axis. The positions are provided by the ticks array. 'Q1 Start' is a static label for the first tick. A list comprehension generates labels for the remaining ticks by iterating over each quarter number from 1 to num_quarters, creating a label for each quarter in the format 'End Q{i}'.

```
x_on_court = player_pbp_on_court['SECONDS_ELAPSED']
```
Assigns the SECONDS_ELAPSED column of the player_pbp_on_court DataFrame to a variable, x_on_court.

```
y_on_court = player_pbp_on_court['TENDEX']
```

Assigns the TENDEX column of the player_pbp_on_court DataFrame to a new variable, y_on_court.

```
plt.plot(x_on_court, y_on_court, linestyle='-', label='On Court')
```

Generates a plot using x_on_court values as the x-coordinates, and y_on_court values as the y-coordinates. Also creates a label which will be used in the plot's legend to identify the line.

```
label = 'On Bench'
```

Creates a variable, label.

```
for e in bench_tuples:
```

Begins iteration on each tuple in bench_tuples.

```
    plt.plot([e[0], e[1]], [np.interp(e[0], x_on_court, y_on_court), np.interp(e[1], x_on_court,
        ↪ y_on_court)], linestyle='-', color='red', label= label)
```

Plots a line between two points [e [0], e [1]] with y—values interpolated from the data points (x_on_court, y_on_court). Sets the line style to solid, the color to red, and assigns a label to the plot.

```
    label = "_nolegend_"
```

Tells matplotlib to exclude the variable, label, from the legend.

```
x = player_pbp['SECONDS_ELAPSED']
```

Assigns the SECONDS_ELAPSED column of player_pbp to a variable, x.

```
y = player_pbp['TENDEX']
```

Assigns the TENDEX column of player_pbp to a variable, y.

```
plt.xlim(x.min(), x.max())
```

Sets the lower and upper x-axis limits of the plot to the minimum and maximum values of x respectively.

```
plt.ylim(y.min() - 0.002, y.max() + 0.002)
```

Sets the lower y-axis limit of the plot to 0.002 less than the minimum of y. Sets the upper y-axis limit of the plot to 0.002 greater than the maximum of y.

```
plt.xlabel('Game Time')
```

Creates a label for the x-axis of the plot.

```
plt.ylabel('Tendex')
```

Creates a label for the plot's y-axis.

```
plt.title(f'Tendex Over the Game for {player_name}')
```

Generates a formatted string for the plot title using the player_name input value.

```
plt.legend()
```

Generates a legend for the plot that indicates how to differentiate between a player being on the court versus on the bench.

```
plt.grid(True)
```

Calls the grid function of Matplotlib, which sets visible grid lines for the plot.

```
plt.show()
```

Displays the plot created.

1.5 Determining Advanced Stat Formulas Using Regression

1.5.1 Data Collection

```
def retry(func, retries=10):
    def retry_wrapper(*args, **kwargs):
        attempts = 0
```

```
        while attempts < retries:
            try:
                return func(*args, **kwargs)
            except requests.exceptions.RequestException as e:
                print(e)
            time.sleep(np.random.normal(2, 0.25, 1)[0])
            attempts += 1
    return retry_wrapper
```

Our retry wrapper sleeps a random amount of time based on a normal distribution with mean 2 and standard deviation 0.25 when an API call results in an exception. The function will retry a call up to 10 times.

```
@retry
def get_full_season_game_ids(season_str):
  single_season_log = TeamGameLogs(season_nullable=season_str, season_type_nullable="Regular Season",
      ↪ league_id_nullable="00")
  single_season_logs = single_season_log.get_data_frames()[0]
  return np.unique(single_season_logs["GAME_ID"].to_list())
```

This function *get_full_season_game_ids* takes in a string in the form "YYYY-YY" which represents an NBA season. The function returns a list with the ID of every game played in that regular season by accessing the *TeamGameLogs* endpoint of the NBA API. Tagged with the *retry* annotation to handle kickback from the API.

```
@retry
def get_adv_BS(game):
  return boxscoreadvancedv3.BoxScoreAdvancedV3(game).data_sets[0].get_data_frame()
```

This function *get_adv_BS* takes in an int which represents a game id. The function returns a Pandas data frame containing the advanced stats of every player who played in the game by accessing the *BoxScoreAdvancedV3* endpoint of the NBA API. Tagged with the *retry* annotation to handle kickback from the API.

```
@retry
def get_trad_BS(game):
  return boxscoretraditionalv3.BoxScoreTraditionalV3(game).data_sets[0].get_data_frame()
```

This function *get_trad_BS* takes in an int which represents a game id. The function returns a Pandas data frame containing the traditional stats of every player who played in the game by accessing the *BoxScoreTraditionalV3* endpoint of the NBA API. Tagged with the *retry* annotation to handle kickback from the API.

```
game_ids = get_full_season_game_ids("2022-23")
```

Call *get_full_season_game_ids* on the 2022-23 season and store the resulting list in new variable *game_ids*.

```
adv_BSs = []
```

Create variable *adv_BSs* with default value of an empty list. This will be used to hold a list of all the different advanced stats.

```
for game in tqdm(game_ids):
```

Start looping through every element in *game_ids* where the variable *game* stores the current game's ID. The function *tqdm* is called to create the progress bar for the loop.

```
    adv_BSs.append(get_adv_BS(game))
```

Call *get_adv_BS* with the current game's ID and appends the resulting data frame to the list *adv_BSs*.

```
adv_BS_df = pd.concat(adv_BSs)
```

Using Pandas function *concat*, concatenate every data frame in list *adv_BSs* into one data frame with all advanced stats for every player in every game in the 2022-23 NBA regular season, stored in new variable *adv_BS_df*.

```
adv_BS_df.to_csv("adv_BS_2022_23.csv")
```

Using Pandas function *to_csv*, save the data frame *adv_BS_df* locally as a comma serperated values (CSV) file named *adv_BS_2022_23.csv* so it can be easily accessed in the future.

```
trad_BSs = []
for game in tqdm(game_ids):
    trad_BSs.append(get_trad_BS(game))
```

```
trad_BS_df = pd.concat(trad_BSs)
trad_BS_df.to_csv("trad_BS_2022_23.csv")
```

Repeat the process to gather all traditional stats for every player in every game from the 2022-23 NBA regular season and save the resulting data frame locally as *trad_BS_2022_23.csv*.

```
adv_BS_df = pd.read_csv("adv_BS_2022_23.csv")
trad_BS_df = pd.read_csv("trad_BS_2022_23.csv")
```

Using Pandas function *read_csv*, read the advanced and traditional stats data frames back into the respective variables *adv_BS_df* and *trad_BS_df*.

1.5.2 True Shooting Percentage Regression

```
full_BS_df = trad_BS_df.merge(adv_BS_df, on= ["gameId", "personId"], how= "inner")[["gameId", "personId",
    ↪ "nameI_x", "fieldGoalsAttempted", "freeThrowsAttempted", "points", "trueShootingPercentage"]]
```

Using Pandas function *merge*, inner join the traditional and advanced stats data frames based on the game's ID and the player's ID. After joining the data frames, compress the data to only keep columns that are used in calculating true shooting percentage and store the resulting data frame in new variable *full_BS_df*.

```
X = full_BS_df[["fieldGoalsAttempted", "freeThrowsAttempted", "points"]]
```

Get the stats true shooting percentage is calculated from and store them in new variable X for regression.

```
y = full_BS_df["trueShootingPercentage"]
```

Get true shooting percentage and store it in new variable y for regression.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1)
```

Using Sci-Kit Learn function *train_test_split* randomly split the data into train and test group where the training group contains 90% of the original data. They are stored in new variables *X_train*, *X_test*, *y_train*, and *y_test*.

```
lin_reg = LinearRegression()
```

Create a Sci-Kit Learn *LinearRegression* model and store it in the new variable *lin_reg*.

```
lin_reg.fit(X_train, y_train)
```

Using Sci-Kit Learn function *fit*, train the linear regression model on the designated training data.

```
print(f"Variable weights: {dict(zip(X.columns, lin_reg.coef_))}")
```

Print out the variable's weights assigned by the linear regression model by creating a dictionary where the keys are the variables and the values are their weights from the model's field *coef_*.

```
print(f"Intercept: {lin_reg.intercept_}")
```

Print out the model's intercept from its field *intercept_*.

```
print(f"R-squared score: {lin_reg.score(X_test, y_test)}")
```

Using Sci-Kit Learn function *score*, print out the model's R^2 score on the designated test data.

```
print(f"Mean absolute error: {np.mean(abs(lin_reg.predict(X_test)- y_test))}")
```

Using Sci-Kit Learn function *predict*, print out the manually calculated mean absolute error on the designated test data.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1)
```

Randomly split the original data into new train and test sets.

```
rf_reg = RandomForestRegressor()
```

Create a Sci-Kit Learn *RandomForestRegressor* model and store it in the new variable *rf_reg*.

```
rf_reg.fit(X_train, y_train)
```

Using Sci-Kit Learn function *fit*, train the random forest regressor model on the designated training data.

```
print(f"Variable importance: {dict(zip(X.columns, rf_reg.feature_importances_))}")
```

Print out the feature's importances assigned by the random forest regressor model by creating a dictionary where the keys are the features and the values are their importances from the model's field *feature_importances_*.

```
print(f"R-squared score: {rf_reg.score(X_test, y_test)}")
```

Using Sci-Kit Learn function *score*, print out the model's R^2 score on the designated test data.

```
print(f"Mean absolute error: {np.mean(abs(rf_reg.predict(X_test)- y_test))}")
```

Using Sci-Kit Learn function *predict*, print out the manually calculated mean absolute error on the designated test data.

1.5.3 PPA Regression

```
PPA_df = pd.read_csv("2022_23_PPA.csv")
```

Read player production average (PPA) data from *2022_23_PPA.csv* which was obtained from Kevin Broom's website. (Broom, 2023) It contains the PPA value for every player whio played in an NBA game in the 2022-23 season before the all star break. Store the resulting data frame in new variable *PPA_df*.

```
PPA_df = PPA_df[["Player", "Tm", "PPA"]]
```

Compress the PPA data frame to only the columns *Player*, *Tm*, and *PPA*.

```
PPA_df["player_id"] = PPA_df["Player"].apply(get_player_id)
```

Using Pandas function *apply* with custom function *get_player_id* on the PPA data frame along the first axis, create new column *player_id* by finding the corresponding ID for every player.

```
PPA_df_safe = PPA_df.dropna(subset= ["player_id"])
```

Player names with accents and apostrophes in the PPA data from Kevin Broom's website did not match with the names in the NBA API. Therefore we were not able to match them to the correct player ID. We decided to remove these rows from the data frame and store the resulting data frame in new variable *PPA_df_safe*.

```
PPA_df_safe = PPA_df_safe.astype({"player_id": int})
```

Cast the player ID column from type String to int.

```
PPA_df_safe[["avg_min", "avg_pts", "avg_fga", "avg_oreb", "avg_dreb", "avg_ast", "avg_stl", "avg_blk",
    ↪ "avg_tov", "avg_pf", "avg_dr", "starts"]] = np.nan
```

Create new columns for the stats that are used in the calculation of PPA. Set the default value of all the columns to be empty, they will be filled in later.

```
single_season_logs = TeamGameLogs(season_nullable="2022-23", season_type_nullable='Regular Season',
    ↪ league_id_nullable='00').get_data_frames()[0]
```

Using the NBA API endpoint *TeamGameLogs*, get information about every NBA regular season game from the 2022-23 season. Store the resulting data frame in new variable *single_season_logs*.

```
before_AS_game_ids = np.unique(single_season_logs[(single_season_logs["GAME_DATE"] > "2022-10-17") &
    ↪ (single_season_logs["GAME_DATE"] < "2023-02-17")]["GAME_ID"])
```

Filter the data frame to only include the game played before the all star break in the 2022-23 season and store the ID of every game in new variable *before_AS_game_ids*.

```
statlines = trad_BS_df.merge(adv_BS_df, on= ["gameId", "personId"], how= "inner")
```

Using Pandas function *merge*, inner join the traditional and advanced stats data frames based on the game's ID and the player's ID. Store the resulting data frame in new variable *statlines*.

```
statlines = statlines[statlines["gameId"].isin(before_AS_game_ids.astype(int))]
```

Filter the data frame to only include games before the all star break.

```
statlines.dropna(subset= ["minutes_x"], inplace= True)
```

Remove all rows where the minutes stat is empty implying the player never checked into the game.

```
def get_minutes(row):
    if type(row["minutes_x"]) == str:
```

```
        mins, secs = row["minutes_x"].split(":")
        mins = int(mins)
        secs = int(secs)
        return mins + secs/60
    return row["minutes_x"]
```

Create function *get_minutes* that takes in a row of the stat lines data frame and converts the string representing minutes played into a float.

```
statlines["minutes_float"] = statlines.apply(get_minutes, axis= 1)
```

Using Pandas function *apply* with custom function *get_minutes* on the stat lines data frame along the first axis, create new column *minutes_float*.

```
@retry
def get_game_rotation(game):
  dfs = gamerotation.GameRotation(game_id= game).get_data_frames()
  return pd.concat(dfs)
```

This function *get_game_rotation* takes in a game's ID and return a dataframe represting all substituations made by either team in the game using the NBA API endpoint *GameRotation*.

```
game_subs_dfs = []
for game in tqdm(before_AS_game_ids):
  game_subs_dfs.append(get_game_rotation(game))
game_subs_df = pd.concat(game_subs_dfs)
game_subs_df.to_csv("PPA_subs.csv")
```

Repeat the flow previously used to gather all box score stats to create data frame with all subs in every game before the all star break in the 2022-23 season. Store the data frame in new variable game_subs_df and save it locally as *PPA_subs.csv*.

```
game_subs_df = pd.read_csv("PPA_subs.csv")
```

Read the complete subs data back into variable *game_subs_df*.

```
def add_starts(row):
```

Begin definition of new function *add_starts* which takes in a row of the stat lines data frame.

```
    game = row["gameId"]
    player = row["personId"]
```

Get the IDs of the current row's game and player and store them in respective new variables *game* and *player*.

```
    player_subs = game_subs_df[(game_subs_df["GAME_ID"] == f"00{game}") & (game_subs_df["PERSON_ID"] ==
        ↪ player)]
```

Filter the substitutions data frame to only include substitutions for the current row's game and player. Store the resulting data fame in new variable *player_subs*.

```
    if len(player_subs) > 0 and player_subs.iloc[0]["IN_TIME_REAL"] == 0.0:
        return 1
    return 0
```

If the current player was subbed in at time 0.0 return a 1 signifying that they started the game. Otherwise return a 0.

```
def get_stats(row):
    player = int(row["player_id"])
    team = row["Tm"]
```

Create function *get_stats* and variables *player* and *team* which represent the current row's player and team.

```
    filtered_statlines = statlines[(statlines["personId"] == player) & (statlines["teamTricode_x"] ==
        ↪ team)]
```

Filter the stat lines data frame to only include the current row's player and team. This is necessary because if a player played on multiple teams they have separate PPA values. Store the resulting data frame

in new variable *filtered_statlines*.

```
avg_stats = filtered_statlines.agg({
    "minutes_float": "mean",
    "points": "mean",
    "fieldGoalsAttempted": "mean",
    "reboundsOffensive": "mean",
    "reboundsDefensive": "mean",
    "assists": "mean",
    "steals": "mean",
    "blocks": "mean",
    "turnovers": "mean",
    "foulsPersonal": "mean",
    "defensiveRating": "mean",
    "started?": "sum"
})
```

Using Pandas function *agg*, aggregate the filtered stat lines to get the average minutes, points, field goal attempts, offensive and defensive rebounds, assists, steals, blocks, turnovers, personal fouls, and defensive rating, and the total number of games started. Store the resulting series in new variable *avg_stats*.

```
row[["avg_min", "avg_pts", "avg_fga", "avg_oreb", "avg_dreb", "avg_ast", "avg_stl", "avg_blk",
    ↪ "avg_tov", "avg_pf", "avg_dr", "starts"]] = avg_stats.to_list()
return row
```

Set the current row's average stats the result of the calculations in the last line. Return the updated version of the current row.

```
PPA_df_safe = PPA_df_safe.apply(get_stats, axis= 1)
```

Using Pandas function *apply* with custom function *get_stats* on the PPA data frame along the first axis, gather every player's average stats.

```
PPA_df_safe.dropna(inplace= True)
```

Remove any more rows that still have empty values.

```
X = PPA_df_safe[['avg_min', 'avg_pts', 'avg_fga', 'avg_oreb', 'avg_dreb', 'avg_ast', 'avg_stl', 'avg_blk',
    ↪ 'avg_tov', 'avg_pf', 'avg_dr', "starts"]]
```

Get the stats PPA is calculated from and store them in new variable X for regression.

```
y = PPA_df_safe["PPA"]
```

Get PPA and store it in new variable y for regression.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1)
lin_reg = LinearRegression()
lin_reg.fit(X_train, y_train)
print(f"Varaible weights: {dict(zip(X.columns, lin_reg.coef_))}")
print(f"Intercept: {lin_reg.intercept_}")
print(f"R-squared score: {lin_reg.score(X_test, y_test)}")
print(f"Mean absolute error: {np.mean(abs((X_test @ lin_reg.coef_ + lin_reg.intercept_) - y_test))}")
```

Randomly split the data into training and testing sets. Create and train Sci-Kit Learn linear regression model. Print model details and some summary statistics of the model. Identical code to true shooting percentage linear regression.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1)
rf_reg = RandomForestRegressor()
rf_reg.fit(X_train, y_train)
print(f"Variable importance: {dict(zip(X.columns, rf_reg.feature_importances_))}")
print(f"R-squared score: {rf_reg.score(X_test, y_test)}")
print(f"Mean absolute error: {np.mean(abs(rf_reg.predict(X_test)- y_test))}")
```

Randomly split the data into training and testing sets. Create and train Sci-Kit Learn random forest regressor model. Print model details and some summary statistics of the model. Identical code to true shooting percentage random forest regressor.

2 Advanced Stat Formulas

The following abbreviations are used in these formulas:

FGA = field goals attempted
FG = field goals made
FTA = free throws attempted
FT = free throws made
3P = three-pointers made
PTS = points scored
RB = rebounds
DRB = defensive rebounds
ORB = offensive rebounds
TRB = total rebounds
PF = personal fouls
(Game) Pace = a team's number of possessions per game
min = number of minutes played
tm refers to the statistic on a team-wide level
lg refers to the statistic on a league-wide level
Gm refers to the total occurrences of a statistic (sum of both teams) in the game

2.1 Effective Field Goal % (eFG%)

Adjusts FG% to account for three-pointers, and shows the FG% that a player who shoots only two-pointers would have to shoot at to match the output of a player who shoots both twos and threes.

$$eFG\% = \frac{FG + (0.5 * 3P)}{FGA}$$

2.2 True Shooting % (TS%)

Measures a player's shooting efficiency, accounting for all field goals and free throws.

$$TS\% = \frac{PTS}{2(FGA + (0.44 \times FTA))}$$

2.3 Unadjusted Player Efficiency Rating (uPER)

$$uPER = \frac{1}{min} \times \left(3P - \frac{PF \times lgFT}{lgPF} + \left[\frac{FT}{2} \times \left(2 - \frac{tmAST}{3 \times tmFG}\right)\right] + \left[FG \times \left(2 - \frac{factor \times tmAST}{tmFG}\right)\right]$$
$$+ \frac{2 \times AST}{3} + VOP \times \Big[DRBP \times (2 \times ORB + BLK - 0.2464 \times [FTA - FT] - [FGA - FG] - TRB)$$
$$+ \frac{0.44 \times lgFTA \times PF}{lgPF} - (TO + ORB) + STL + TRB - 0.1936\,(FTA - FT)\Big]\Big)$$

2.3.1 Factor

Scales PER to match typical league-wide value.

$$factor = \frac{2}{3} - \left[\left(0.5 \times \frac{lgAST}{lgFG}\right) \div \left(2 \times \frac{lgFG}{lgFT}\right)\right]$$

2.3.2 Value of Possession (VOP)

Calculates average points per possession.

$$VOP = \frac{lgPTS}{lgFGA - lgORB + lgTO + 0.44 \times lgFTA}$$

2.3.3 Defensive Rebound % (DRBP)

Calculates percentage of total rebounds that are defensive.

$$DRBP = \frac{lgTRB - lgORB}{lgTRB}$$

2.3.4 Player Efficiency Rating (PER)

Reduces all of a player's positive (made shots, assists, rebounds, blocks, steals) and negative (missed shots, turnovers, personal fouls) contributions into one value and adjusts for team pace (number of possessions) and minutes played. The statistic follows a scale where a PER of 0 represents a player who won't stick in the league, 15 represents an average player, and 35+ represents an all-time great season.

$$PER = \left(uPER \times \frac{lgPace}{tmPace}\right) \times \frac{15}{lguPER}$$

2.4 Player Impact Estimate (PIE)

Measures a player's overall statistical contribution to the game.

```
(PTS + FGM + FTM - FGA - FTA + DREB + (.5 * OREB) + AST + STL + (.5 * BLK) - PF - TO) /
(GmPTS + GmFGM + GmFTM - GmFGA - GmFTA + GmDREB + (.5 * GmOREB) + GmAST +
GmSTL + (.5 * GmBLK) - GmPF - GmTO)
```

2.5 Tendex

Evaluates player efficiency using box score stats, minutes played, and game pace (number of possessions for their team).

[(Points) + (Rebounds) + (Assists) + (Steals) + (Blocks) - (Missed Field Goals) - 0.5 * (Missed Free Throws) - (Turnovers) - (Fouls Committed)] / (Minutes Played * Game Pace)

2.6 Box Plus-Minus (BPM)

Uses team and individual box score stats to estimate player performance relative to the NBA average.

BPM = Raw BPM + team adjusted

Raw BPM = scoring + ball handling + rebounding + defense + position constant

team adjusted = (Adjusted team rating - [sum(raw BPM * %MIN) for all players on team]) / 5 =
(team rating + lead bonus - [sum(raw BPM * %MIN) for all players on team]) / 5

lead bonus: (0.35 / 2) * AVG. lead
AVG. lead = team rating * PACE / 200

2.7 Value Over Replacement Player (VORP)

Converts BPM into an estimate of each player's overall contribution to the team, measured against a replacement player (one on a minimum salary or not a normal member of the rotation). The formula yields the number of points the player is producing over a replacement player, per 100 team possessions over an entire season.

BPM of -2 is considered replacement level

VORP = [BPM - (-2.0)] * (% of possessions played) * (team games/82)

3 Example Plots

All game-level plots are for Nikola Jokic in the playoff game against the Minnesota Timberwolves on May 10, 2024.

3.1 Traditional Stat Flow Graphs

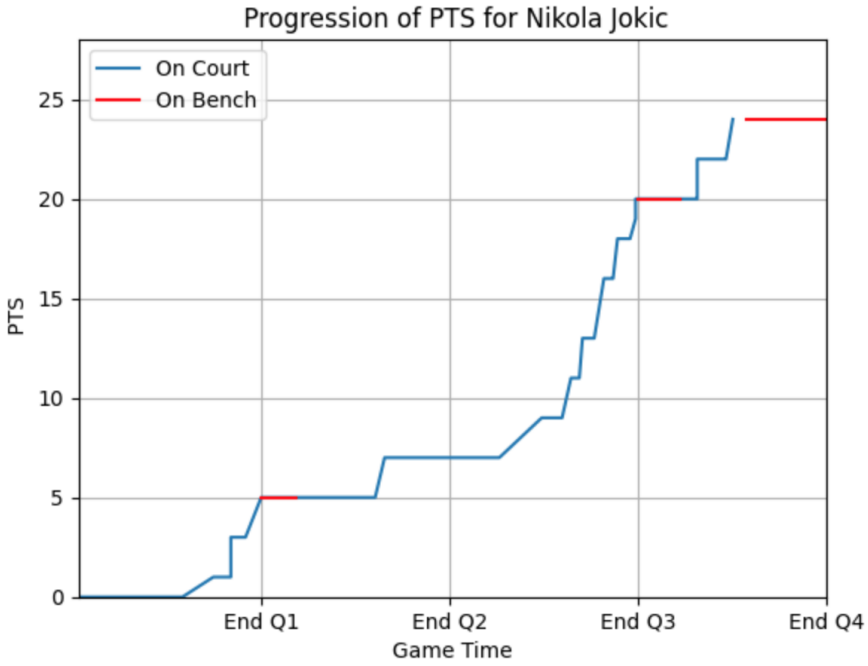

Figure 1: Flow Graph of a Traditional Stat

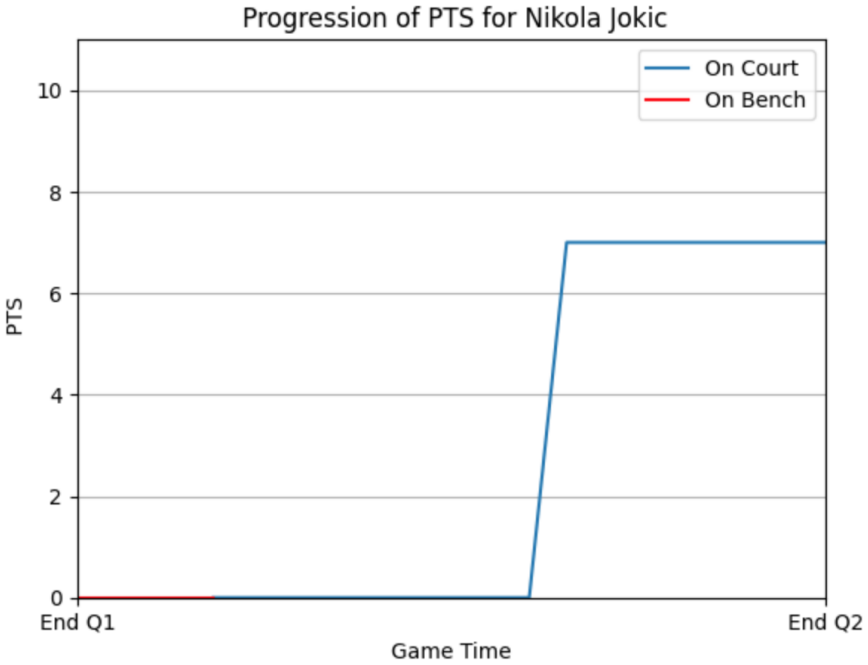

Figure 2: Flow Graph of a Traditional Stat for a Specific Time Frame

3.2 Shooting % Flow Graphs

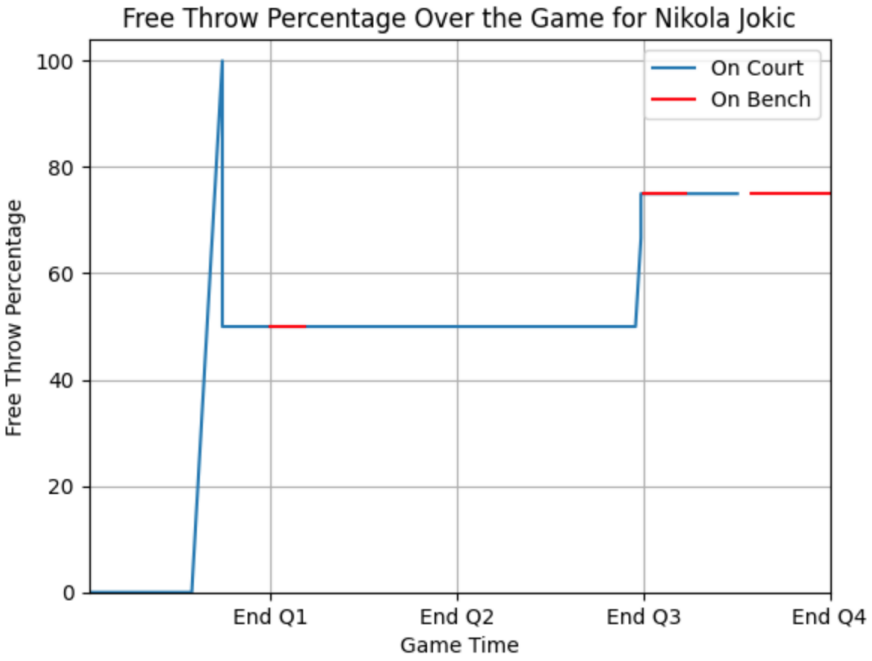

Figure 3: Flow Graph of Free Throw Percentage

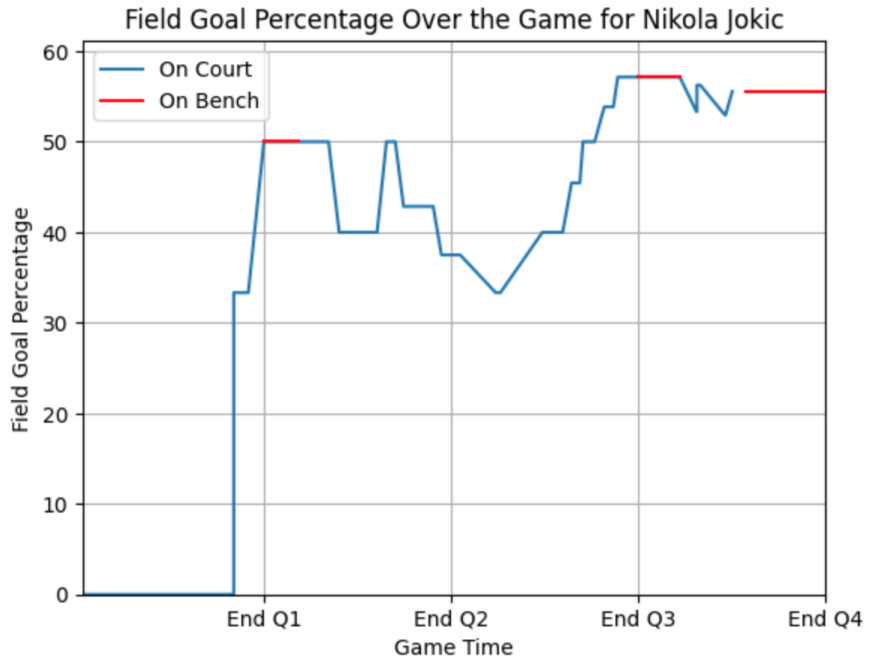

Figure 4: Flow Graph of Field Goal Percentage

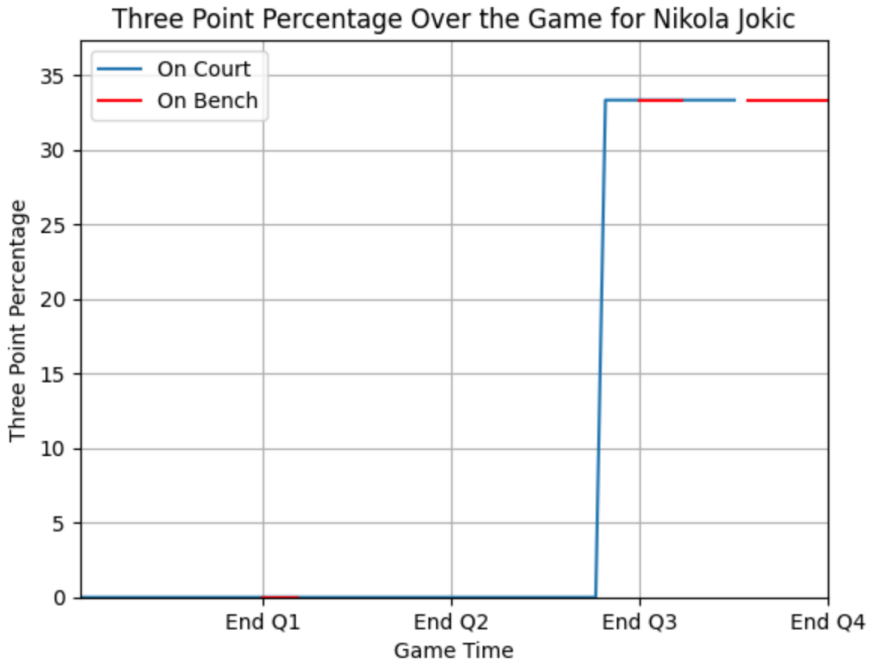

Figure 5: Flow Graph of 3PT Field Goal Percentage

3.3 Field Goal % Flow Graphs Over a Season

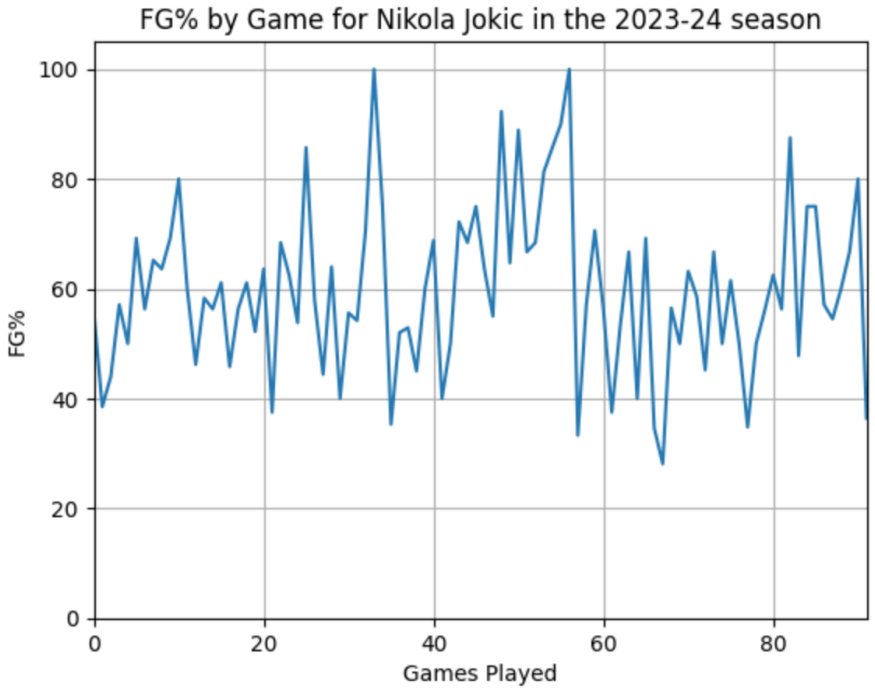

Figure 6: Flow Graph of Field Goal % for Each Game in a Season

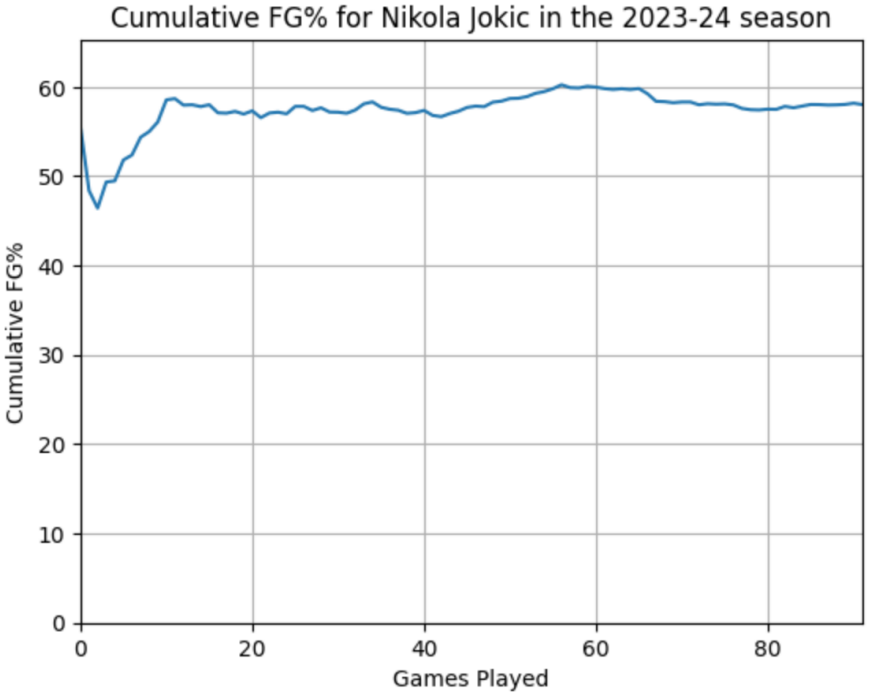

Figure 7: Flow Graph of Cumulative Field Goal % over a Season

3.4 Advanced Stat Flow Graphs

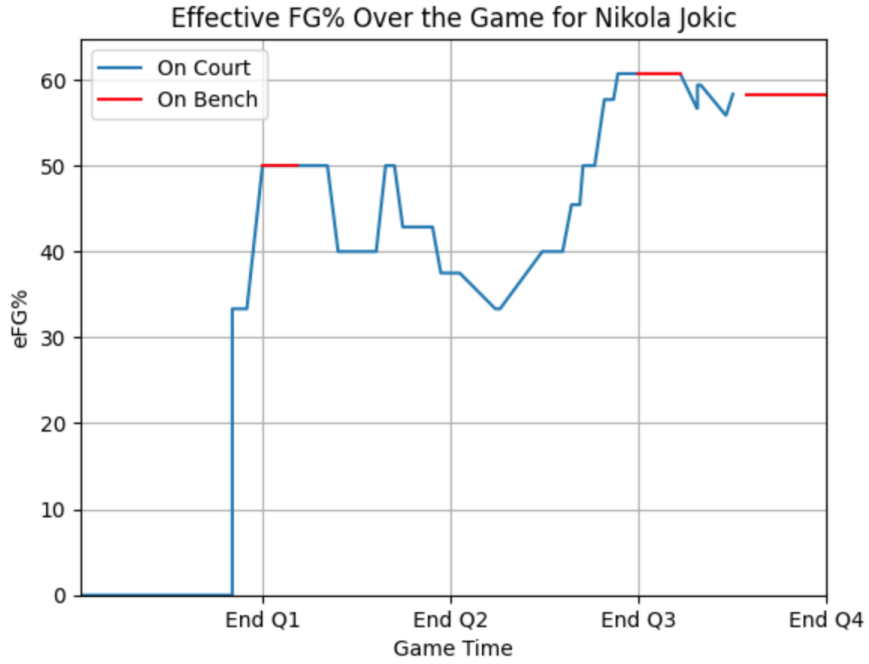

Figure 8: Flow Graph of Effective Field Goal % (eFG%)

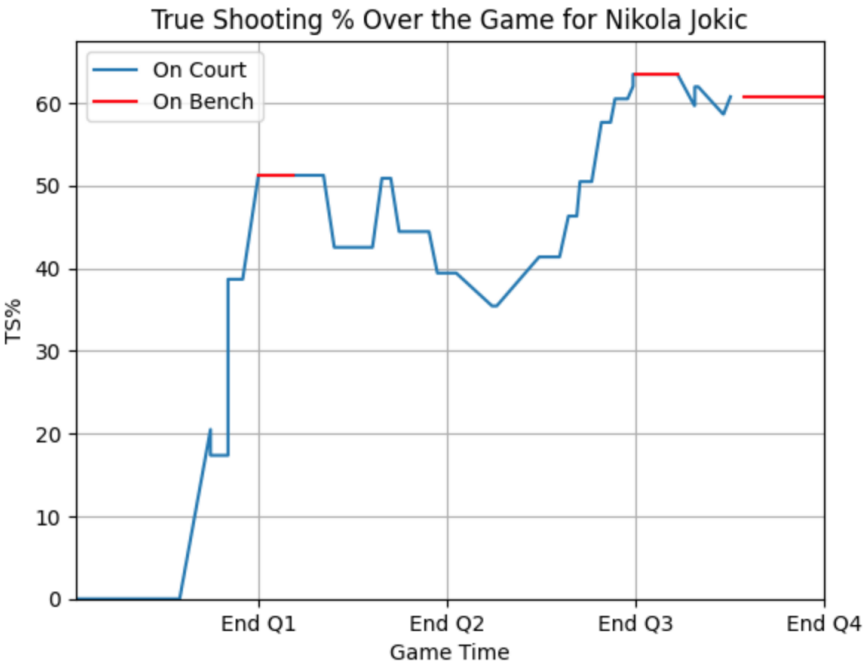

Figure 9: Flow Graph of True Shooting % (TS%)

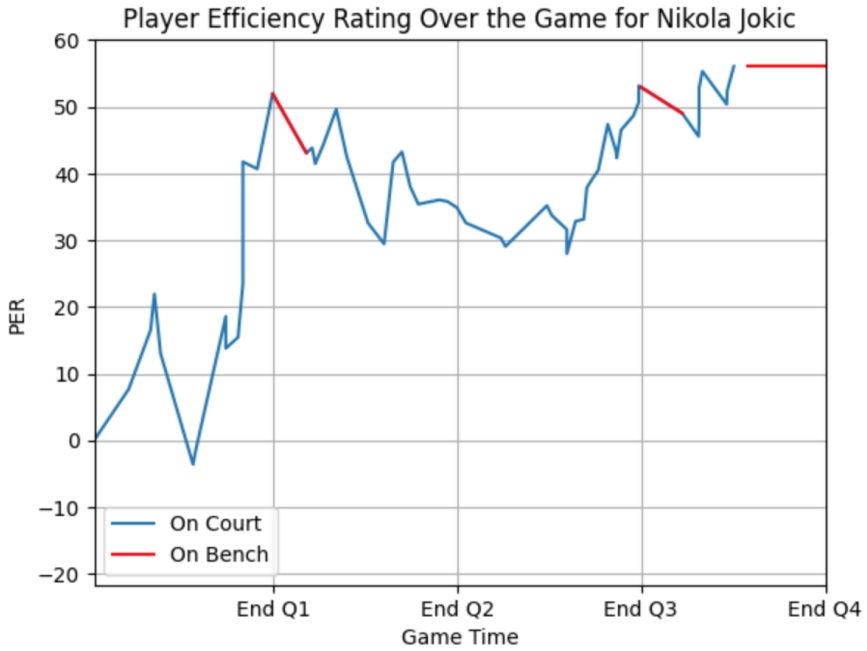

Figure 10: Flow Graph of Player Efficiency Rating (PER)

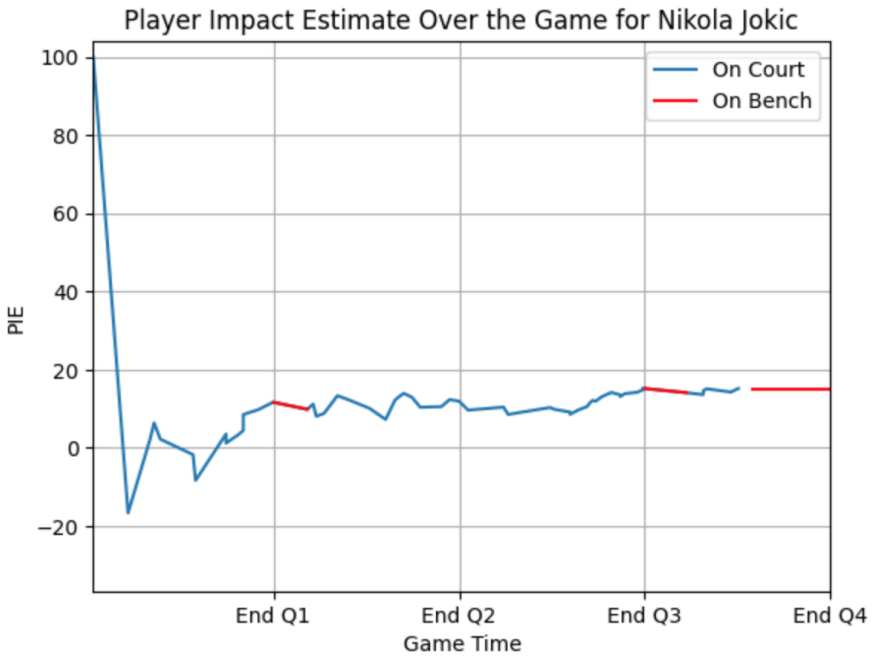

Figure 11: Flow Graph of Player Impact Estimate (PIE)

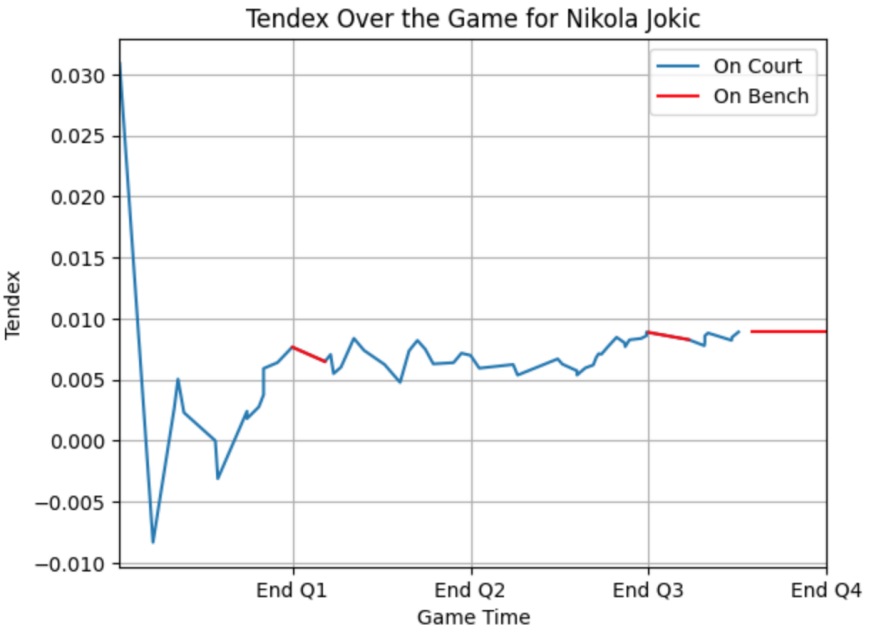

Figure 12: Flow Graph of Tendex

4 Regression Findings

4.1 True Shooting Percentage

As seen earlier true shooting percentage measures a player's shooting efficiency, accounting for all field goals and free throws. It is calculated by the formula:

$$TS\% = \frac{PTS}{2(FGA+(0.44*FTA))}$$

Linear Regression

Linear regression takes a data set with one feature identified as the dependent variable and constructs a line of best fit based on all of the available data. We ran linear regression to try to approximate the non-linear formula for true shooting percentage with a linear combination of all of its parameters, field goal attempts, free throw attempts, and points. The following are the printed outputs described in the code documentation.

Variable weights: {'fieldGoalsAttempted': -0.036722156647872976, 'freeThrowsAttempted': -0.027006514063110883, 'points': 0.05029835338795257}
Intercept: 0.2975036629912753
R-squared score: 0.4305983507796547
Mean absolute error: 0.1947803191669215

The actual linear function approximation is

$$TS\% \approx -0.04 * FGA - 0.03 * FTA + 0.05 * PTS + 0.3$$

Random Forest Regressor

Since we are trying to approximate a non-linear function, a linear regression model will not provide the best results. A random forest regressor completes the same task as linear regression but takes a different approach. A random forest is made up of several decision tree regressors and takes the average of all trees' results. Each decision tree is trained individually on separate data sets where they learn to best fit their data. Combining many trees in a random forest can be very powerful. The following are the printed outputs described in the code documentation.

Variable importance: {'fieldGoalsAttempted': 0.1575033177502875, 'freeThrowsAttempted': 0.03168508348223556, 'points': 0.810811598767477}
R-squared score: 0.9995397951646936
Mean absolute error: 0.001482868462757852

There is no exact observable formula for the random forest regressor.

In conclusion, due to the formula for true shooting percentage being non-linear, the linear regression model performed very poorly and essentially predicted the mean for every player. However, the random forest regressor was able to capture the equation much better and was able to predict the true shooting percentage with almost perfect accuracy.

4.2 Player Production Average (PPA)

PPA was created to measure a player's contribution to team success by Kevin Broom, the Deputy Managing Editor at Bullets Forever. According to the blog post on his website, PPA is a "linear weighted metric" similar to PER, VORP, and TENDEX, based on the following stats. (Broom, 2023)

- points
- rebounds (offensive and defensive weighed differently)
- assists
- steals

- blocks
- shot attempts
- turnovers
- personal fouls
- starts
- minutes
- on-court team defensive rating

Linear Regression

Varaible weights: {'avg_min': -0.6626395813536263, 'avg_pts': 19.191682509708343, 'avg_fga': -17.753498417480372, 'avg_oreb': 14.4027490272362, 'avg_dreb': 4.323264336113572, 'avg_ast': 11.881103314717462, 'avg_stl': 35.34092792177506, 'avg_blk': 13.841215644855675, 'avg_tov': -42.95466870352586, 'avg_pf': -17.705625522460082, 'avg_dr': -0.3581129550265998, 'starts': 0.1795622036672473}
Intercept: 87.82363560625562
R-squared score: 0.8162779448902899
Mean absolute error: 15.353183207480331

$$PPA \approx -0.66 * MPG + 19.19 * PPG - 17.75 * FGAPG + 14.4 * ORPG + 4.32 * DRPF + 11.88 * APG + 35.34 * SPG + 13.84 * BPG - 42.95 * TVPG - 17.71 * PFPG - 0.34 AVG\_DR + 0.18 * Starts + 87.82$$

Random Forest Regressor

Variable importance: {'avg_min': 0.12259244475772, 'avg_pts': 0.3646383053337357, 'avg_fga': 0.036049550131544296, 'avg_oreb': 0.037650529160927586, 'avg_dreb': 0.09334635280321339, 'avg_ast': 0.019640684908651285, 'avg_stl': 0.10085481890361417, 'avg_blk': 0.016658579199153076, 'avg_tov': 0.07515073196397402, 'avg_pf': 0.05440603975236288, 'avg_dr': 0.057110592895497116, 'starts': 0.021901370189606403}
R-squared score: 0.8306675130536879
Mean absolute error: 12.744782608695653

In conclusion, with a roughly linear equation to approximate, the linear regression model performs significantly better and even outperforms the random forest regressor. While PPA's real equation is unknown to us we were able to create a good prediction from our linear regression model.

References

[1] "About Box plus/Minus (BPM)." *Basketball*. www.basketball-reference.com/about/bpm2.html. Accessed 10 May 2024.

[2] Broom, Kevin."NBA Player Production Average" *kevinbroom.com*, 01 Mar. 2023. https://kevinbroom.com/ppa/

[3] "Effective Field Goal Percentage." *Wikipedia*. Wikimedia Foundation, 8 Apr. 2024. en.wikipedia.org/wiki/Effective_field_goal_percentage.

[4] Kelley, Kyle. "How do I use updatable displays on colab?" *Stack Overflow*, 25 Oct. 2017. https://stackoverflow.com/questions/46939393/how-do-i-use-updatable-displays-on-colab

[5] "NBA Stat Glossary." *Stat Glossary — Stats — NBA.Com*. www.nba.com/stats/help/glossary#pie. Accessed 10 May 2024.

[6] Patel, Swar. "Swar/NBA_API: An API Client Package to Access the Apis for Nba.Com." GitHub, github.com/swar/nba_api. Accessed 12 May 2024.

[7] "Player Efficiency Rating." *Wikipedia*. Wikimedia Foundation, 13 Mar. 2024. en.wikipedia.org/wiki/Player_efficiency_rating.

[8] "Tendex." *Wikipedia*. Wikimedia Foundation, 30 Aug. 2020. en.wikipedia.org/wiki/Tendex.

[9] Thunder, Oklahoma City. "Statistical Analysis Primer." *NBA.Com*, NBA, 19 Jan. 2005. www.nba.com/thunder/news/stats101.html.

[10] "True Shooting Percentage." *Wikipedia*. Wikimedia Foundation, 5 Apr. 2024. en.wikipedia.org/wiki/True_shooting_percentage.