10 September 2023

NFL Fantasy Football Prediction

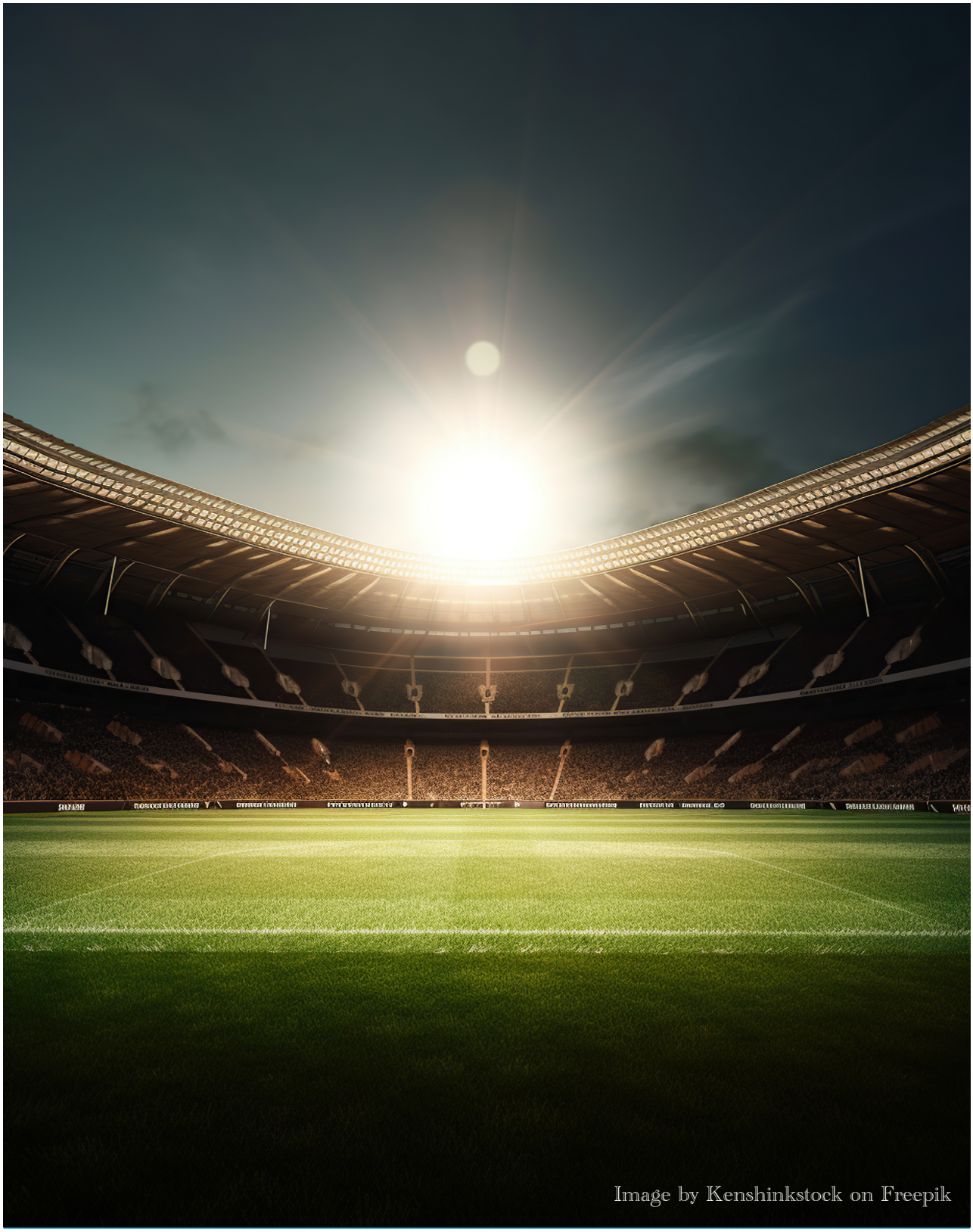


Image by Kenshinkstock on Freepik

1. Introduction

Fantasy football is a great way to pass time by football enthusiasts, allowing them to build and manage their own virtual teams composed of real NFL players. The success of a fantasy football team relies heavily on predicting the performance of individual players accurately.

In this report, we explore the analysis and modeling of NFL player data to predict their fantasy football points using data provided by Pharaoh Analytics.

-

2. Data Analysis

I started by obtaining the NFL player dataset, loading and viewing it. Preliminary check revealed a wide range of statistics for each player spanning multiple seasons. The dataset contains information such as player identity and names, team affiliations, years of experience, age, and various performance metrics.

```
matches = pd.read_csv("../data/pbp/nfl_fantasy_football_ dataset.csv", index_col=0)

# display first few rows
matches.head()
```

✓ 0.1s

	name	player_id	season	team	years_exp	age	bmi	draft_number	fantasy_points_per_game
0	NaN	00-0007091	2016	NaN	NaN	NaN	NaN	NaN	11.387500
1	NaN	00-0010346	2016	NaN	NaN	NaN	NaN	NaN	9.136000

```
# show a random sample of 5 rows
matches.sample(5)
```

✓ 0.0s

	name	player_id	season	team	years_exp	age	bmi	draft_number	fantasy_points_per_game
677	Ryan Mallett	00-0028012	2016	BAL	5.0	28.0	0.041091	NaN	10.134286
1126	Kendall Wright	00-0029708	2018	ARI	6.0	29.0	0.037755	20.0	6.921429
3604	Chad Beebe	00-0034309	2020	MIN	2.0	26.0	0.037755	NaN	4.000000

Exploratory Data Analysis (EDA)

I conducted a comprehensive exploratory data analysis to understand the dataset better following these approaches:

Descriptive statistics: This technique provided a foundational understanding of the dataset, helping to guide subsequent data preprocessing and modeling steps. In this context, I performed descriptive statistics on the dataset to gain insights into the data's central tendencies, variability, and distributions or spread of the data. This helped me

```
# Display statistical summaries for numerical columns
print("Statistical Summaries for Numerical Columns:")
print(matches.describe())
```

✓ 0.2s

```
Statistical Summaries for Numerical Columns:
            season    years_exp          age          bmi  draft_number  \
count  4655.000000  4655.000000  4655.000000  4655.000000   4655.000000
mean   2019.564554     4.230505    26.664876     0.040723     94.430720
std       2.289076     2.698408     2.826117     0.002996     52.392709
min    2016.000000     1.000000    22.000000     0.031150      1.000000
```

understand the range and variability of variables like fantasy points, age, passing yards, and more. This was achieve using the following code:

Identifying missing values: I checked for missing values using a combination of **.isnull().sum()** and **.info()** methods to account for the number of missing values in each column. This step helped me identify columns with missing data that might require data imputation or removal. The total number of entries and columns are also displayed making it easy to notice the ones with missing values. For example the

```
# Display data types and missing values information
print("Data Types and Missing Values:")
print(matches.info())

# Result indicates that some columns (name, team, years_exp, age, bmi, draft_number, rtd_sh, dom, w8dom)
# all have missing values. This is because they all have values less than the total no. of entries.
```

✓ 0.0s

```
Data Types and Missing Values:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 4655 entries, 0 to 4901
Data columns (total 63 columns):
 #   Column                              Non-Null Count  Dtype
---  ------                              --------------  -----
 0   name                                3598 non-null   object
 1   player_id                           4655 non-null   object
```

dataset has **4655** entries and **64** columns. Result indicates that some columns (**name**, **team**, **years_exp**, **age**, **bmi**, **draft_number**, **rtd_sh**, **dom**, **w8dom** all have missing values. This is because they all have values less than the total no. of entries.

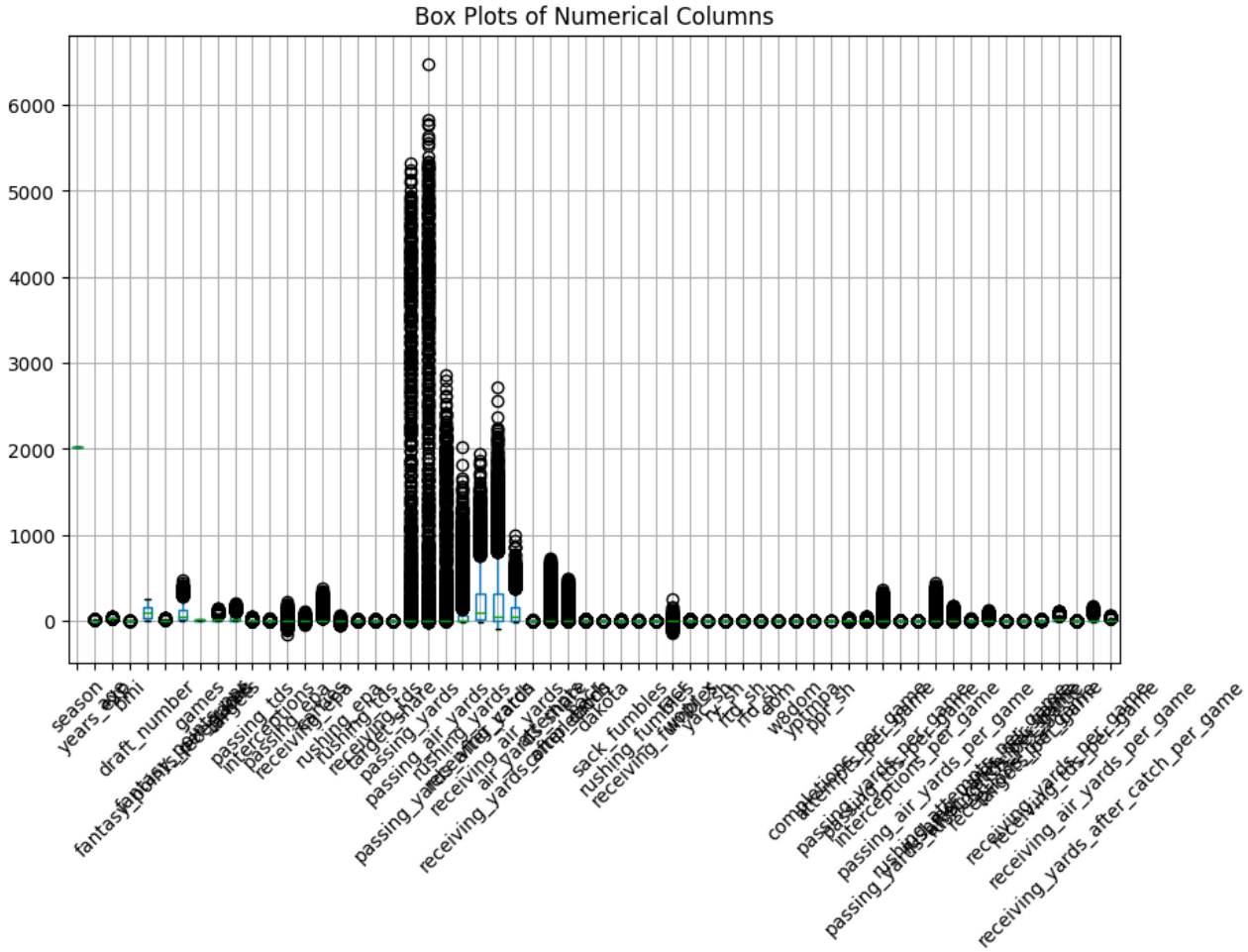
```
# check for duplicates
duplicates = matches[matches.duplicated()]
print("Duplicate Rows:")
print(duplicates)
```

✓ 0.1s Python

```
Duplicate Rows:
Empty DataFrame
Columns: [name, player_id, season, team, years_exp, age, bmi, draft_number, fantasy_points_per_game, fantasy_points_ppr, games, recep
Index: []

[0 rows x 67 columns]
```

Identifying duplicate values: Duplicate entries were identified using the **.duplicated()** method in Pandas. This method returns a Boolean Series indicating whether each row is a duplicate of a previous row. The **.duplicated().sum()** method was used to count the number of duplicate entries in the dataset. In the provided project information, it was discovered that no duplicate entries were found in the dataset.

Visualizing potential outliers: I used box plots to visualize potential outliers in the numerical columns. This further helped identify extreme values that might require further investigation or preprocessing. The project identified outliers in the "age" column, where some players had ages that were exceptionally high compared to the typical age range for NFL players. For example, players with ages around 40 or higher were considered outliers. The screenshot below captures the box plot of potential outliers:



```python
# Identify potential outliers
for col in numerical_columns:
    q1 = matches[col].quantile(0.25)
    q3 = matches[col].quantile(0.75)
    iqr = q3 - q1
    lower_bound = q1 - 1.5 * iqr
    upper_bound = q3 + 1.5 * iqr

    outliers = matches[(matches[col] < lower_bound) | (matches[col] > upper_bound)]
    if not outliers.empty:
        print(f"Potential outliers in column {col}:")
        print(outliers)

plt.show()
```

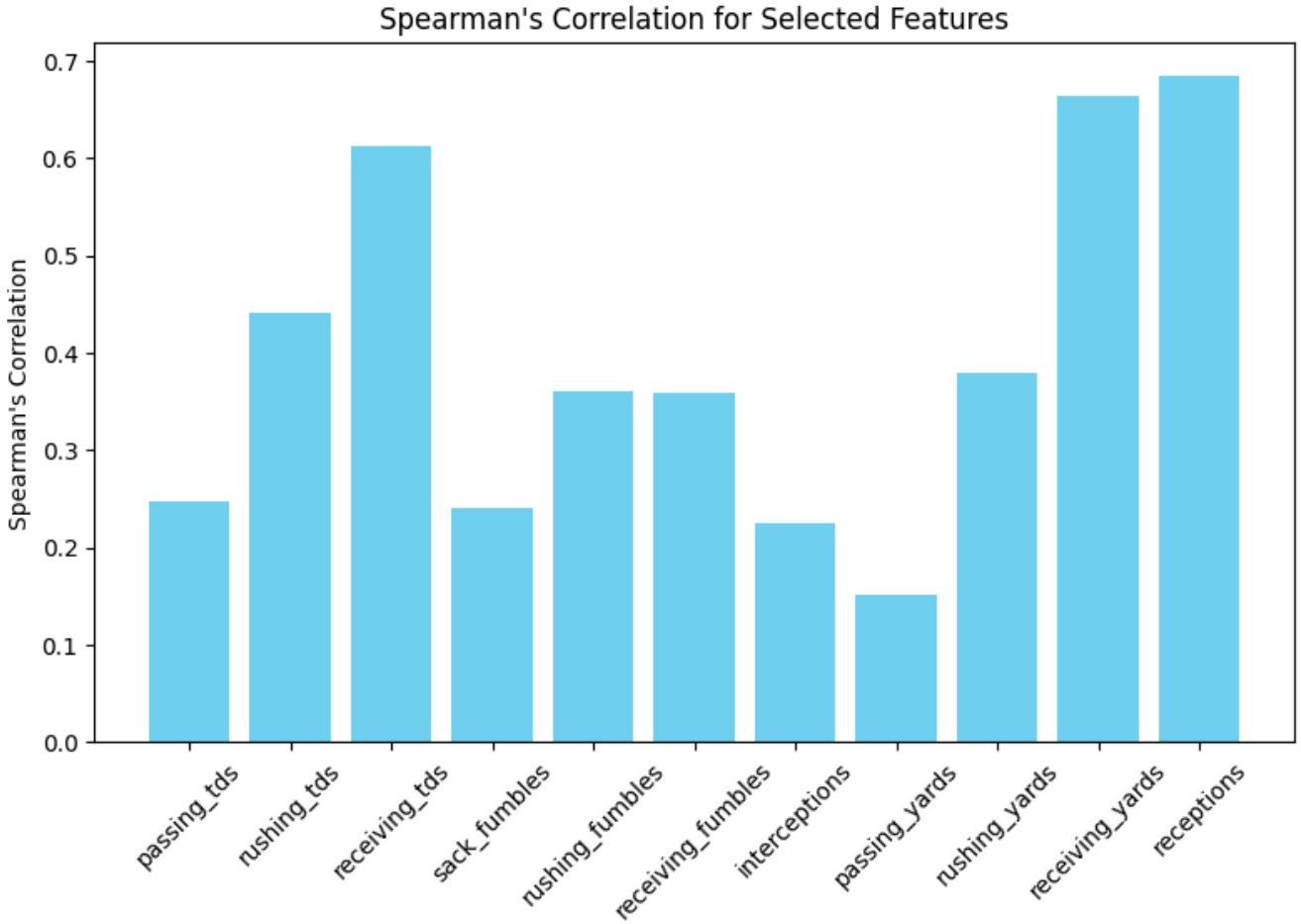
```
(function)
def __or__(other: bool | list[bool] | list[int] | np_ndarray_bool | Series[bool]) ->
Series[bool]: ...

def __or__(other: int | np_ndarray_anyint | Series[int]) -> Series[int]: ...
```

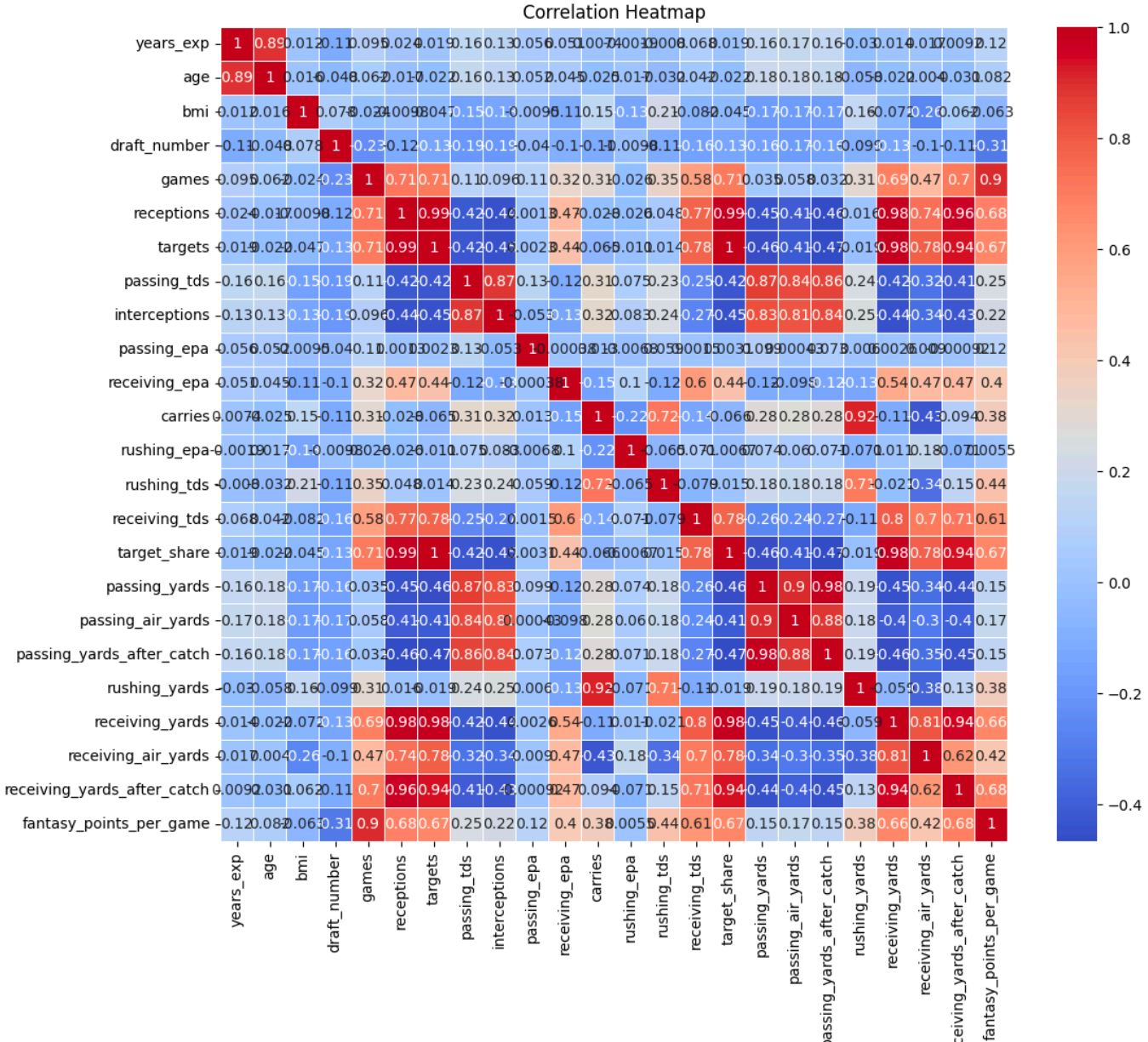
✓ 1.5s Python

```
Potential outliers in column years_exp:
                name   player_id  season team  years_exp   age       bmi  \
3          Tom Brady  00-0019596    2016   NE       16.0  39.0  0.038954
4          Tom Brady  00-0019596    2017   NE       17.0  40.0  0.038954
5          Tom Brady  00-0019596    2018   NE       18.0  41.0  0.038954
6          Tom Brady  00-0019596    2019   NE       19.0  42.0  0.038954
```

Identifying highly correlated features: Using Spearman's correlation, and heatmap, I was able to establish the features that were highly correlated with the target variable. For the Spearman's correlation, I wrote a function that accepted selected features and target as input and calculated the correlation, p_value, and add remarks. The result indicated the most of the features were strongly correlated with the target (fantasy_points_per_game), though tilting towards a non-linear model. Receptions, receiving_yards, and receiving_tds, were features that showed strong linearity. Correlations from the heat map also agreed with the Spearman's correlations.

```python
# Function to calculate Spearman's correlation and add Remark
def calculate_spearmanr(df, feature, target):
    correlation, p_value = spearmanr(df[feature], df[target])

    if p_value < 0.05:
        remark1 = "Significant"
    else:
        remark1 = "Non-significant."

    if abs(correlation) < 0.5:
        remark2 = "Strong non-linearity"
    else:
        remark2 = "Strong linearity"

    return correlation, p_value, remark1, remark2

# List of selected features and the target variable
```


Spearman's Correlation for Selected Features
Spearman's Correlation
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
passing_tds
rushing_tds
receiving_tds
sack_fumbles
rushing_fumbles
receiving_fumbles
interceptions
passing_yards
rushing_yards
receiving_yards
receptions


Correlation Heatmap
years_exp
age
bmi
draft_number
games
receptions
targets
passing_tds
interceptions
passing_epa
receiving_epa
carries
rushing_epa
rushing_tds
receiving_tds
target_share
passing_yards
passing_air_yards
passing_yards_after_catch
rushing_yards
receiving_yards
receiving_air_yards
receiving_yards_after_catch
fantasy_points_per_game
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4

Handling missing values, outliers, and duplicates: Rows with missing values in critical columns, such as the target variable "fantasy_points_per_game," were mostly dropped. This ensures that the analysis is based on complete data for the target variable. For non-target columns, missing values were sometimes imputed using strategies like mean, median, or forward/backward filling, depending on the nature of the data. I further filled missing numerical values with median and categorical columns like 'name' and 'team' with 'unknown', to still maintain completeness of dataset, and preserve the central tendency. In other to further prepare the dataset for modelling, I converted the categorical variables into a suitable format by using one-hot encoding. For example the 'team' column with sample values like 'PHI', 'PIT', 'SD', and 'SEA', after one-hot encoding had a separate column for each category like 'team_PHI', 'team_PIT', 'team_SD', and 'team_SEA', where each column contained 1 if the player's position matches that category and 0 otherwise. Since machine learning require numerical inputs to work with this becomes particularly useful. It also serves to prevent the model from assigning any ordinal relationship between the categories which might not be appropriate for certain categorical variables.

Outliers in the "age" column were not treated in the initial analysis. These outliers were considered valid data points because there are some NFL players who continue to play at an older age. Instead of treating them as errors, these data points were kept in the dataset. In further processing, filling the missing values with median was done to take care of outliers.

Duplicates were not found in the dataset.

3. Feature Engineering

Feature Selection:

Understanding a problem is perhaps the most important part of any project or challenge. In this challenge, the choice of 'fantasy_points_per_game' for target and the other selected features was based on the nature of the game, and the additional context provided by the project.

In fantasy football, participants select real NFL players to form their fantasy teams. The primary objective of fantasy football is to accumulate points based on the performance of these selected players during actual NFL games. 'Fantasy_points_per_game' directly represents how many fantasy points a player scores on average in each NFL game, making it a crucial metric for fantasy football enthusiasts. 'Fantasy_points_per_game' is a comprehensive metric that encapsulates a player's performance across various aspects of the game, including passing, rushing, receiving, touchdowns, and other statistics. It is a key indicator of a player's overall fantasy football value and can be highly predictive of a player's future performance.

Since the goal of the analysis is to build a predictive model that can forecast a player's future fantasy football performance, using 'fantasy_points_per_game' as the target variable is logical. The model can then utilize various player statistics and features to make predictions about how many fantasy points a player is likely to score in a game.

The input features were then selected to capture the points allotted to some stats variables. These included throwing_td, rushing_td, receiving_td, fumble, interception, etc.

Feature Creation: I created new features and performed feature transformation, using the additional context provided in the dataset description. For instance, when creating aggregated features or calculating fantasy points based on certain player statistics (e.g., touchdowns, yards), I used the provided point system, which relies on statistics like the mean of passing yards or receptions.

Hint: Create new metric(s) that can be used as an additional feature to enhance predictive accuracy.

To create a target variable for our model, I calculated fantasy points for each player based on the provided statistics. I applied the following point system:

- Passing td: 4 points
- Rushing td: 6 points
- Receiving td: 6 points
- 1 Passing yard: 0.04 points
- 1 Rushing yard: 0.1 points
- 1 Receiving yard: 0.1 points
- 1 Reception: 1 point
- 1 Fumble: -2 points
- 1 Interception: -2 points
-

These calculations enabled me to have a target variable, "fantasy_points_per_game," for modeling.

```python
# Define the point system
points_system = {
    "passing_td": 4,
    "rushing_td": 6,
    "receiving_td": 6,
    "passing_yard": 0.04,
    "rushing_yard": 0.1,
    "receiving_yard": 0.1,
    "reception": 1,
    "fumble": -2,
    "interception": -2
}

# Create features based on the point system
matches["passing_points"] =   matches["passing_tds"] * points_system["passing_td"] +     matches["passing_yards"] * points_system["passing_yard"]
matches["rushing_points"] =   matches["rushing_tds"] * points_system["rushing_td"] +     matches["rushing_yards"] * points_system["rushing_yard"]
matches["receiving_points"] = matches["receiving_tds"] * points_system["receiving_td"] + matches["receiving_yards"] * points_system["receiving_yar

# Calculate turnover points based on fumbles and interceptions
matches["turnover_points"] = (matches["sack_fumbles"] + matches["rushing_fumbles"] + matches["receiving_fumbles"] + matches["interceptions"]) * po


# Calculate total fantasy points per game
matches["fantasy_points_per_game"] = matches["passing_points"] + matches["rushing_points"] + matches["receiving_points"] + matches["turnover_points

# Calculate total fantasy points per game (PPR format)
# matches["fantasy_points_ppr"] = (
```

```
      receiving_yards_after_catch_per_game  passing_points  rushing_points  \
0                                 0.000000          103.60             1.5
1                                 0.000000          125.96            -0.6
3                                 1.812500          334.80            23.3
4                                 0.000000          254.16             6.4
5                                 0.000000          311.08             2.8
...                                    ...             ...             ...
4897                             19.714286            0.00            20.0
4898                              0.000000           11.04            18.3
4899                             12.428571            0.00             0.0
4900                             14.733333            0.00           159.0
4901                              0.000000            0.00             0.0

      receiving_points  turnover_points
0                  0.0            -16.0
1                  0.0            -36.0
3                  4.6            -26.0
4                  0.0            -14.0
5                  0.0            -30.0
...                ...              ...
4897             144.1             -0.0
4898               0.0            -12.0
4899              93.1             -0.0
4900              43.5             -0.0
4901               4.5             -0.0

[4655 rows x 67 columns]
```

After gaining insight into how the selected features correlated with the target fantasy_points_per_game, the next step was to check if this correlation holds true for next season's points nor performance. I adopted a number of strategies to achieve this:

First, I made a copy of the current points data frame and added an extra season to it:

```
# Let's group by season, player_id and name of player
groupby_feats = ["season", "player_id", "name"]


# group by and aggregate by sum
fantasy_points_df = matches.loc[:, player_feats].groupby(groupby_feats,
as_index=False).sum()



# make a copy of the current points dataframe
next_df = fantasy_points_df.copy()

# add 1 to season
next_df['season'] = next_df['season'].add(1)

# merge this back on the original points dataframe
new_fantansy_points_df = (fantasy_points_df
        .merge(next_df,
            on=groupby_feats,
            suffixes=('', '_prev'),
            how='left'))
```

The result showed a year-to-year correlations with fantasy points, making it suitable for the next seasons prediction.

4. Model Build

Data Splitting
I split the data into training and testing sets. Notably, I considered the NFL season's structure to ensure that predictions were made for the upcoming season (2023/24).

Base Models
I trained two base models:

- Decision Tree Regressor
- Linear Regression

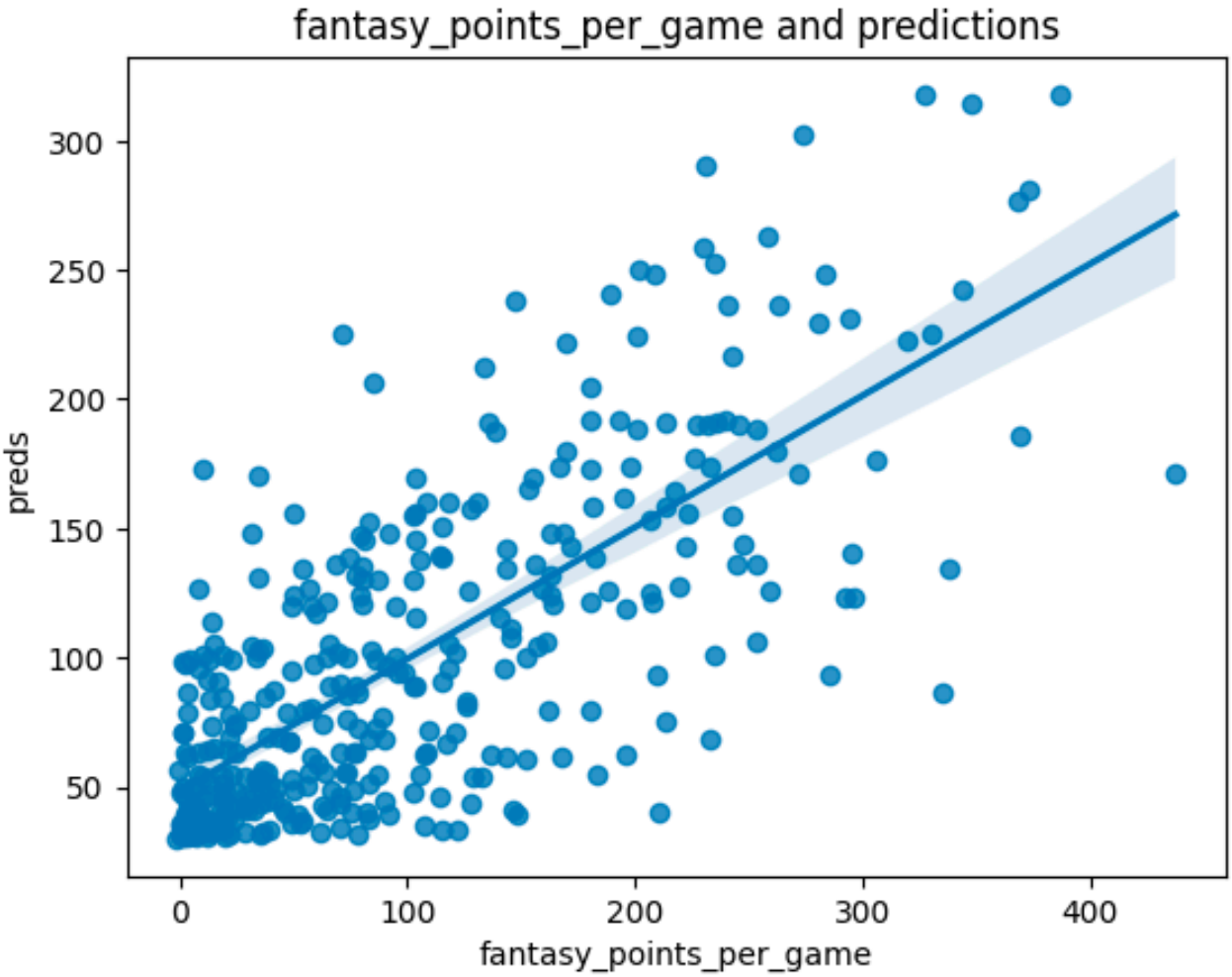
Both models were trained on the training data and used to make initial predictions

Using A Linear Model:
I first started simply with linear modelling since the initial dataset's features correlated a little bit linearly with the target. First, I made a subset of the dataset and ensured that there was no null values. I then split the training and testing sets to those before 2022/23 season and those in the current season of 2022/2023, respectively.

I then initialise the linear regression model and went ahead to fit and train, performed the prediction. After prediction, I evaluated the model using the sklearn metrics 'mean_square_error' method and Pearson's correlation. The result showed strong non-linearity as suspect, necessitating the trial of another approach - use of Decision Tree Regressor.

```
model_data = (new_fantansy_points_df
        .dropna(subset=selected_features_prev+[target]))
```

```
# fit, or "train", the model on the training data
model.fit(train_data.loc[:, selected_features_prev],
      train_data[target])

# predict on the test data
preds = model.predict(test_data.loc[:, selected_features_prev])
```




```
# Let's run some basic statistics to examine the
# quality of the prediction

rmse = mean_squared_error(test_data['fantasy_points_per_game'], test_data['preds'])
r2 = pearsonr(test_data['fantasy_points_per_game'], test_data['preds'])[0]
print(f"rmse: {rmse}\nr2: {r2}")
```

✓ 0.0s

```
rmse: 3890.3473255896856
r2: 0.7380143425750938
```

```
test_data.loc[:, ['season', 'player_id', 'name', 'fantasy_points_per_game', 'preds']].sort_values('fantasy_p
```

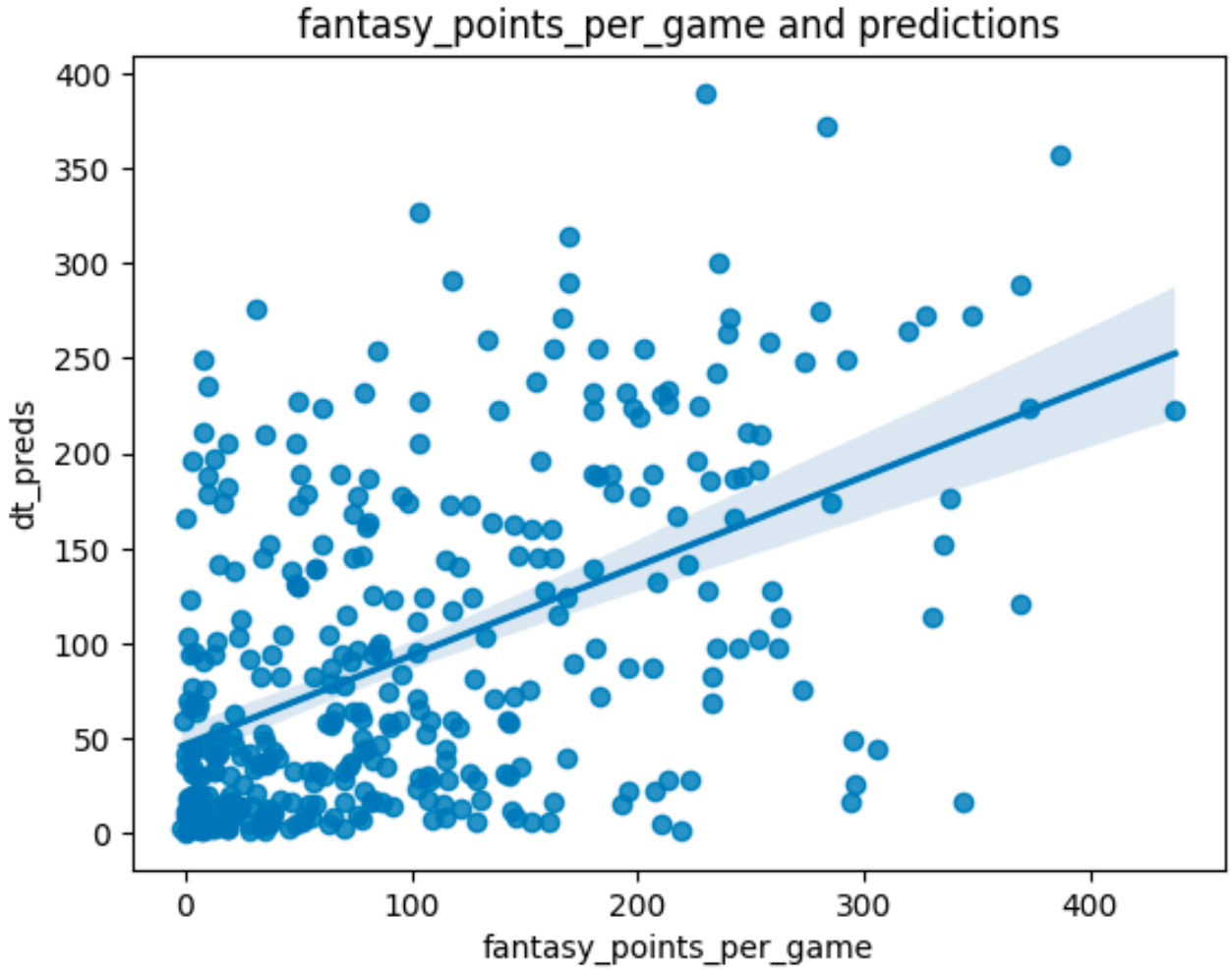
✓ 0.0s

	season	player_id	name	fantasy_points_per_game	preds
3701	2022	00-0033908	Cooper Kupp	437.50	170.989362
3792	2022	00-0034857	Josh Allen	386.58	317.448893
3447	2022	00-0019596	Tom Brady	372.74	281.067718
3926	2022	00-0036223	Jonathan Taylor	369.10	185.817631
3959	2022	00-0036355	Justin Herbert	368.76	276.436302
...	...	...	...	...	...
3554	2022	00-0031610	Darren Waller	133.50	211.990912

Using A Non-Linear Model (Decision Tree):

I first initialised the DecisionTreeRegressor, before fitting and performing predictions. After evaluation, I discovered that the model still required further tuning, as the result wasn't much of an improvement.

```
# Initialize and train a DecisionTreeRegressor
dt_model = DecisionTreeRegressor(random_state=42)

# fit, or "train", the model on the training data
dt_model.fit(train_data.loc[:, selected_features_prev],
      train_data[target])

# predict on the test data
dt_preds = dt_model.predict(test_data.loc[:, selected_features_prev])

# match the correct rows

dt_preds = pd.Series(dt_preds, index=test_data.index)

# join your predictions back to your test dataset
test_data['dt_preds'] = dt_preds
```




```
# evaluate the decision tree model

dt_rmse = mean_squared_error(test_data['fantasy_points_per_game'], test_data['dt_preds'])
dt_r2 = pearsonr(test_data['fantasy_points_per_game'], test_data['dt_preds'])[0]
print(f"rmse: {dt_rmse}\nr2: {dt_r2}")
```

✓ 0.0s

```
rmse: 7949.543749850658
r2: 0.5044733595586773
```