

Title : "The Performance Analysis of Major League Baseball Players Using Advanced Statistical Analysis"

Problem Statement:

The goal of this project is to create a full evaluation system for MLB players that includes both hitting and pitching data and give top-10 players . The research uses machine learning methods such as a linear regression model , random forest, and neural networks to predict and compare player performance. The ultimate objective is to develop a normalised scoring system that successfully integrates numerous performance measurements, resulting in a comprehensive picture of a player's contribution to the club. The evaluation method will be assessed with historical data to verify the accuracy and stability.

Dataframe Columns Description:

Striking DataFrame Columns: A roster titled hitting_columns is generated, including column titles linked to hitting data in baseball. Pitching DataFrame Columns: A roster named pitching_columns is formed, comprising column titles linked to pitching metrics in baseball. Output Statements: The code displays the column titles for both hitting and pitching DataFrames. Result Analysis: The result demonstrates the names of the columns for both DataFrames:

Uniformity in Column Titles: Guarantee that the column titles are uniformly labeled and align with the data origin. For example, some column names in hitting_columns utilize lowercase (e.g., 'position') while others use mixed case (e.g., 'Double (2B)'). Uniformity in naming standards would enhance clarity and decrease mistakes during data handling. Potential Revisions: The column title 'third base player' in the hitting_columns list seems odd as it pertains to a position instead of a hitting metric. If it reflects a tally of a third baseman's actions, it should be rechristened for lucidity. Similar uniformity should be upheld in presenting columns. Rewrite this text with different synonyms words in en language keeping as it is. For instance, 'Achieved Run Average' and 'walks' could be standardized to either all lowercase or all title case. Dataset Generation: If the DataFrames are to be generated at a later time, make sure the column names align precisely to prevent discrepancies. Example code for DataFrame generation: Further Examination: After the DataFrames are filled with data, you can conduct different analyses like computing player means, contrasting player achievements, etc.

```
In [1]: # Assuming you have two DataFrames: hitting_df and pitching_df
# with the respective column names

# Hitting DataFrame Columns
hitting_columns = [
    'Player name', 'position', 'Games', 'At-bat', 'Runs', 'Hits',
    'Double (2B)', 'third baseman', 'home run', 'run batted in', 'walk',
    'Strikeouts', 'stolen base', 'Caught stealing', 'AVG',
    'On-base Percentage', 'Slugging Percentage', 'On-base Plus Slugging'
]

# Pitching DataFrame Columns
pitching_columns = [
```

```

'Player name', 'Position', 'Win', 'Loss', 'Earned run Average',
'Games played', 'Games Started', 'Complete Game', 'Shutout', 'Save',
'Save Opportunity', 'Innings pitched', 'hit', 'run', 'earned run',
'home run', 'Hit Batsmen', 'base on balls', 'Strikeouts', 'WHIP',
'AVG'
]

# Print the columns
print("Hitting DataFrame Columns:", hitting_columns)
print("Pitching DataFrame Columns:", pitching_columns)

```

Hitting DataFrame Columns: ['Player name', 'position', 'Games', 'At-bat', 'Runs', 'Hits', 'Double (2B)', 'third baseman', 'home run', 'run batted in', 'walk', 'Strikeouts', 'stolen base', 'Caught stealing', 'AVG', 'On-base Percentage', 'Slugging Percentage', 'On-base Plus Slugging']

Pitching DataFrame Columns: ['Player name', 'Position', 'Win', 'Loss', 'Earned run Average', 'Games played', 'Games Started', 'Complete Game', 'Shutout', 'Save', 'Save Opportunity', 'Innings pitched', 'hit', 'run', 'earned run', 'home run', 'Hit Batsmen', 'base on balls', 'Strikeouts', 'WHIP', 'AVG']

Importing Pandas:

The script brings in the pandas module, which is crucial for managing DataFrame tasks in Python. Loading Information: The datasets `baseball_batting.csv` and `baseball_thrower.csv` are loaded into DataFrames named `hitting_info` and `throwing_info`, respectively. Showing Columns: The column headings of both DataFrames are displayed to verify any inconsistencies and confirm they align with the anticipated format.

Result Examination: The result displays the headings for both batting and throwing DataFrames:

Batting Information Columns: Perspectives and Suggestions Discrepancies in Heading Names: There are slight inconsistencies in heading names, which may lead to complications during data handling. In the batting Data Tabulation, 'third baseman' appears incongruous as it is more of a position designation rather than a numerical assessment. In the pitching DataFrame, 'Hit Batsmen' has a leading gap which could create problems during data retrieval. Standardizing Header Titles: In order to prevent inconsistencies and guarantee seamless data functions, it is advisable to standardize the column names. Here's how you can relabel the columns: Data Consistency Inspection: Examination for absent or conflicting data that could impact your analysis. You can utilize pandas functions such as `isnull()` and `dropna()` for this task:

Data Analysis: Carry out exploratory statistics and depict the data to acquire understanding. Rewrite this text with different synonyms words in en language keepingas it is, For instance, one can compute the mean batting average or ERA and create bar charts for various metrics.

```

In [1]: import pandas as pd

# We have to Load the datasets with new variable names for avoiding distraction
hitting_data = pd.read_csv('baseball_hitting.csv')
pitching_data = pd.read_csv('baseball_pitcher.csv')

# Display column names to check for discrepancies with new variable names
print("Hitting Data Columns: ", hitting_data.columns)
print("Pitching Data Columns: ", pitching_data.columns)

```

```

Hitting Data Columns: Index(['Player name', 'position', 'Games', 'At-bat', 'Runs', 'Hits',
                             'Double (2B)', 'third baseman', 'home run', 'run batted in', 'walk',
                             'Strikeouts', 'stolen base ', 'Caught stealing', 'AVG',
                             'On-base Percentage', 'Slugging Percentage', 'On-base Plus Slugging'],
                             dtype='object')
Pitching Data Columns: Index(['Player name', 'Position', 'Win', 'Loss', 'Earned run Average',
                              'Games played', 'Games Started', 'Complete Game', 'Shutout', 'Save',
                              'Save Opportunity', 'Innings pitched', 'hit', 'run', 'earned run',
                              'home run', ' Hit Batsmen', 'base on balls', 'Strikeouts', 'WHIP',
                              'AVG'],
                              dtype='object')

```

The given code is an all-encompassing solution to assess and classify baseball players according to their hitting and throwing numbers. Here's an exhaustive analysis of the observations obtained from the task:

1. Data Loading and Validation: Uploading Data: The records for hitting and throwing figures are uploaded into DataFrames. Column Validation: The column titles are presented to confirm they align with the anticipated structure. This stage is vital to pinpoint any discrepancies or inconsistencies in the column names.

2. Scoring system: Batting score calculation: The `calculate_batting_score` function uses weighted inputs from different batting metrics to determine a batting score for each player. The weights can be modified according to the significance of each Key data points include Points scored, Hits, homerun, RBI, and OPS. Pitching Score Calculation: The function `determine_pitcher_rating` determines a pitching score based on various pitching metrics. Analogous to the batting score, weights can be modified as required. Significant indicators include Victory, K's, CG, Blank, and a reverse correlation with ERA.

3. Normalization: The ratings for hitting and throwing are standardized to a universal scale by dividing each player's rating by the highest rating in their individual categories. This guarantees conformity between batting and pitching performances.

4. Combining Information: Merging Data Structures: The hitting and throwing Data Structures are combined on the player names. An external merge is utilized to incorporate all participants, even if they just possess information in one of the groupings. Handling Absent Data: Missing data (NaNs) are replaced with zeros. This process is crucial to guarantee that the lack of data in one category doesn't unjustly disadvantage a player.

5. Total Score Calculation:

A cumulative score for every player is determined by adding together their standardized batting and pitching scores. This cumulative total gives a comprehensive ranking of athletes determined by their performances in both hitting and throwing.

6. Ranking and Presenting Top Players:

Sorting and Choosing Top Players: The players are organized by their overall score in descending order, and the top 10 players are picked. Elaborate Information: Elaborate analytics for the top players are extracted from the initial batting and pitching DataFrames.

7. Final Data Set:

The ultimate Data Set consists of detailed data for the top athletes, encompassing their results and standardized results in both hitting and throwing, as well as their in-depth metrics. Code Inspection and Enhancements Here are some factors to contemplate and potential enhancements:

Discrepancies in Column Labels: Ensure uniform column labels throughout the datasets. For instance, the baseman is probably a position and not a performance measure. Ensure to rectify such inconsistencies. **Weights in Evaluation Criteria:** The weights employed in the evaluation criteria are random and might require adjustment depending on expertise in the field or particular research objectives.

Standardization Method: Explore alternative normalization methods if the scope of scores differs greatly. For example, Z-value standardization could be beneficial.

Elaborate Remarks and Records:

Inserting additional remarks and records within the code will enhance clarity and sustainability, particularly if others must grasp or expand upon the analysis. **Concluding Remarks:** This task showcases the process of managing, purifying, and examining baseball statistics data utilizing Python and pandas. By combining hitting and throwing stats, standardizing scores, and computing a merged performance score, we can efficiently rank players based on their overall This approach can be utilized and expanded to different sports analytics assignments.

```
In [2]: import pandas as pd

# We have to Load the datasets with new variable names
batting_data = pd.read_csv('baseball_hitting.csv')
pitching_data = pd.read_csv('baseball_pitcher.csv')

# Display column names to check for discrepancies with new variable names
print("Batting Data Columns: ", batting_data.columns)
print("Pitching Data Columns: ", pitching_data.columns)

# Define a scoring system
def calculate_batting_score(row):
    # Example scoring system (weights can be adjusted)
    return (row['Runs'] * 2 + row['Hits'] * 2 + row['home run'] * 4 +
            row['run batted in'] * 3 + row['On-base Plus Slugging'] * 5)

def calculate_pitcher_score(row):
    # Adjusted to match available columns
    return (row['Win'] * 4 + row['Strikeouts'] * 3 +
            row['Complete Game'] * 5 + row['Shutout'] * 6 +
            (1 - row['Earned run Average']) * 5)

# Calculate scores
batting_data['Batting Score'] = batting_data.apply(calculate_batting_score, axis=1)
pitching_data['Pitching Score'] = pitching_data.apply(calculate_pitcher_score, axis=1)

# Normalize the scores to a common scale
batting_data['Normalized Batting Score'] = batting_data['Batting Score'] / batting_data['Batting Score'].max()
pitching_data['Normalized Pitching Score'] = pitching_data['Pitching Score'] / pitching_data['Pitching Score'].max()

# Merge the datasets on player names (if a player appears in both)
combined_data = pd.merge(batting_data, pitching_data, on='Player name', how='outer')
```

```

# Fill NaNs with 0 (for players who only have hitting or pitching data)
combined_data = combined_data.fillna(0)

# Calculate a total score by summing the normalized hitting and pitching scores
combined_data['Total Score'] = combined_data['Normalized Batting Score'] + combined_data['Normalized Pitching Score']

# Sort players by total score
top_players = combined_data.sort_values(by='Total Score', ascending=False).head(10)

# Select relevant columns for display
top_players = top_players[['Player name', 'Total Score', 'Normalized Batting Score', 'Normalized Pitching Score']]

# Merge top player information back into hitting and pitching data
top_player_names = top_players['Player name'].tolist()

# Get detailed information for top players
top_batting_data = batting_data[batting_data['Player name'].isin(top_player_names)]
top_pitching_data = pitching_data[pitching_data['Player name'].isin(top_player_names)]

# Merge detailed information into a single DataFrame
final_top_players_data = pd.merge(top_batting_data, top_pitching_data, on='Player name')

# Fill NaNs with 0 in the final DataFrame
final_top_players_data = final_top_players_data.fillna(0)

# Define the columns to ensure they exist
columns_to_check = ['Player name', 'position', 'Games', 'At-bat', 'Runs', 'Hits', 'Errors', 'Innings pitched', 'Strikeouts', 'Earned Runs', 'Wins', 'Losses', 'Saves', 'Quality Starts', 'Complete Games', 'Shutouts', 'Batters Faced', 'Pitching Speed', 'Pitching Rate', 'Pitching Rate per 100', 'Pitching Rate per 1000']

# Fill NaNs with 0 in specific columns
for column in columns_to_check:
    if column in final_top_players_data.columns:
        final_top_players_data[column] = final_top_players_data[column].fillna(0)

# Print the final DataFrame
print(final_top_players_data)

```

Batting Data Columns: Index(['Player name', 'position', 'Games', 'At-bat', 'Runs', 'Hits', 'Double (2B)', 'third baseman', 'home run', 'run batted in', 'walk', 'Strikeouts', 'stolen base ', 'Caught stealing', 'AVG', 'On-base Percentage', 'Slugging Percentage', 'On-base Plus Slugging'], dtype='object')

Pitching Data Columns: Index(['Player name', 'Position', 'Win', 'Loss', 'Earned run Average', 'Games played', 'Games Started', 'Complete Game', 'Shutout', 'Save', 'Save Opportunity', 'Innings pitched', 'hit', 'run', 'earned run', 'home run', ' Hit Batsmen', 'base on balls', 'Strikeouts', 'WHIP', 'AVG'], dtype='object')

	Player name	position	Games	At-bat	Runs	Hits	Double (2B)	\
0	B Bonds	LF	2986.0	9847.0	2227.0	2935.0	601.0	
1	B Bonds	RF	1849.0	7043.0	1258.0	1886.0	302.0	
2	H Aaron	RF	3298.0	12364.0	2174.0	3771.0	624.0	
3	B Ruth	RF	2504.0	8399.0	2174.0	2873.0	506.0	
4	A Pujols	1B	3080.0	11421.0	1914.0	3384.0	686.0	
5	A Rodriguez	SS	2784.0	10566.0	2021.0	3115.0	548.0	
6	C Young	CF	1465.0	4710.0	668.0	1109.0	288.0	
7	C Young	CF	1465.0	4710.0	668.0	1109.0	288.0	
8	C Young	CF	1465.0	4710.0	668.0	1109.0	288.0	
9	T Cobb	CF	3034.0	11429.0	2246.0	4191.0	723.0	
10	R Johnson	LF	1320.0	3630.0	518.0	1014.0	215.0	
11	R Johnson	LF	1320.0	3630.0	518.0	1014.0	215.0	
12	W Johnson	P	937.0	2324.0	241.0	547.0	94.0	
13	N Ryan	0	0.0	0.0	0.0	0.0	0.0	

	third baseman	home run_Batting	run batted in	...	run earned run	\
0	77.0	762.0	1996.0	...	0.0	0.0
1	66.0	332.0	1024.0	...	0.0	0.0
2	98.0	755.0	2297.0	...	0.0	0.0
3	136.0	714.0	2213.0	...	400.0	309.0
4	16.0	703.0	2218.0	...	0.0	0.0
5	31.0	696.0	2086.0	...	0.0	0.0
6	24.0	191.0	590.0	...	3167.0	2147.0
7	24.0	191.0	590.0	...	605.0	570.0
8	24.0	191.0	590.0	...	581.0	530.0
9	297.0	117.0	1938.0	...	0.0	0.0
10	23.0	65.0	408.0	...	1703.0	1513.0
11	23.0	65.0	408.0	...	182.0	146.0
12	41.0	24.0	255.0	...	1902.0	1424.0
13	0.0	0.0	0.0	...	2178.0	1911.0

	home run_Pitching	Hit Batsmen	base on balls	Strikeouts_Pitching	WHIP	\
0	0.0	0.0	0.0	0.0	0.00	
1	0.0	0.0	0.0	0.0	0.00	
2	0.0	0.0	0.0	0.0	0.00	
3	10.0	29.0	441.0	488.0	1.16	
4	0.0	0.0	0.0	0.0	0.00	
5	0.0	0.0	0.0	0.0	0.00	
6	138.0	163.0	1217.0	2803.0	1.13	
7	186.0	33.0	502.0	1062.0	1.26	
8	147.0	33.0	366.0	536.0	1.35	
9	0.0	0.0	0.0	0.0	0.00	
10	411.0	190.0	1497.0	4875.0	1.17	
11	12.0	9.0	151.0	169.0	1.23	
12	97.0	205.0	1363.0	3508.0	1.06	
13	321.0	158.0	2795.0	5714.0	1.25	

	AVG_Pitching	Pitching Score	Normalized Pitching Score
0	0	0.00	0.000000
1	0	0.00	0.000000

2	0	0.00	0.000000
3	0.22	2470.60	0.124132
4	0	0.00	0.000000
5	0	0.00	0.000000
6	0.252	14645.85	0.735860
7	0.232	3492.25	0.175463
8	0.265	1960.45	0.098500
9	0	0.00	0.000000
10	0.221	16547.55	0.831408
11	0.248	776.40	0.039009
12	0.227	15501.15	0.778833
13	0.204	19903.05	1.000000

[14 rows x 42 columns]

Observations for the Ultimate Best Players Information The examination comprises uploading two datasets (hitting and throwing data) for baseball athletes, computing ratings for each athlete depending on their hitting and throwing performance, standardizing these ratings, and merging them to uncover the top players. Here are some essential observations obtained from the ultimate dataset containing details about these standout players:

Top Achievers

1. Top Hitters:

- Barry Bonds: Recognized for his outstanding batting display, Bonds shows up twice in the roster for separate achievements in the left field (LF) and right field (RF) roles.
- Hank Aaron: Another iconic batter, Aaron's numbers mirror his extensive and triumphant tenure in the right field (RF).
- Babe Ruth: Renowned for his potent hitting, Ruth's numbers also encompass pitching records, highlighting his versatility.

1. Top Hurlers:

- Nolan Ryan: Renowned for his outstanding strikeout records, Ryan boasts the highest standardized pitching rating on the roster.
- Randy Johnson: Another exceptional hurler, Johnson's statistics showcase his mastery in strikeouts and pitching victories.
- Walter Johnson: A prominent figure in baseball, Walter Johnson's pitching statistics are remarkable for strikeouts and full games. **United Presentation**
- Athletes like Babe Ruth and Cy Young shine not just for their hitting skills but also for their pitching abilities, demonstrating their worth as versatile players.
- Cy Young is mentioned numerous times because of the diverse pitching statistics, underscoring his outstanding career. **Statistical Highlights**

1. Batting Scores:

- Batting scores are determined using runs, hits, homers, RBIs, and on-base plus slugging (OPS). Athletes such as Barry Bonds and Hank Aaron thrive in these statistics, resulting in top scores.

1. Pitching Points:

- Throwing points are calculated based on victories, strikeouts, full games, whitewashes, and earned run average (ERA). Nolan Ryan and Randy Johnson top the charts in these

1. Standardized Scores:

- Scores are standardized to bring them onto a common scale, allowing for equitable comparison between batting and pitching performances.
- The aggregated overall score is the combination of standardized hitting and throwing scores, pinpointing the top overall athletes.

Positional Breakdown

- Outfielders: Predominant in the roster are athletes predominantly stationed in outfield positions (LF, RF, CF), like Barry Bonds, Hank Aaron, and Babe Ruth, showcasing the influence of outfielders in hitting.
- Pitchers: Pitchers like Nolan Ryan and Randy Johnson, although not impacting batting, have extremely high pitching ratings.

Career Durability

- Athletes with extended careers, as evidenced by the number of matchups participated in (e.g., Hank Aaron with 3298 games and Ty Cobb with 3034 games), typically amass remarkable statistics throughout their time in the sport.

Two-Way Players

- Babe Ruth and Walter Johnson are renowned for their performances both as hitters and hurlers, demonstrating their adaptability.

Insights for Personal Statistics

- For instance, Babe Ruth has 714 homers and 2213 RBIs as a hitter, along with impressive pitching numbers such as 488 strikeouts and a 1.16 WHIP.
- Nolan Ryan excels with 5714 strikeouts and 321 home runs surrendered as a pitcher, showcasing his superiority and durability in the game.

Data Fullness

- The information contains essential categories for both hitting and throwing records, with missing values filled in appropriately to prevent any problems in calculations. When analyzing these elite athletes and their numbers, we acquire a thorough comprehension of their impact on the sport of baseball, both at the plate and on the mound. The standardized grading system aids in contrasting players across diverse positions, accentuating the adaptability and influence of each player.

```
In [5]: final_top_players_data.tail(100)
```


Out[5]:

	Player name	position	Games	At-bat	Runs	Hits	Double (2B)	third baseman	home run_Batting	run batted in
0	B Bonds	LF	2986.0	9847.0	2227.0	2935.0	601.0	77.0	762.0	1996.0
1	B Bonds	RF	1849.0	7043.0	1258.0	1886.0	302.0	66.0	332.0	1024.0
2	H Aaron	RF	3298.0	12364.0	2174.0	3771.0	624.0	98.0	755.0	2297.0
3	B Ruth	RF	2504.0	8399.0	2174.0	2873.0	506.0	136.0	714.0	2213.0
4	A Pujols	1B	3080.0	11421.0	1914.0	3384.0	686.0	16.0	703.0	2218.0
5	A Rodriguez	SS	2784.0	10566.0	2021.0	3115.0	548.0	31.0	696.0	2086.0
6	C Young	CF	1465.0	4710.0	668.0	1109.0	288.0	24.0	191.0	590.0
7	C Young	CF	1465.0	4710.0	668.0	1109.0	288.0	24.0	191.0	590.0
8	C Young	CF	1465.0	4710.0	668.0	1109.0	288.0	24.0	191.0	590.0
9	T Cobb	CF	3034.0	11429.0	2246.0	4191.0	723.0	297.0	117.0	1938.0
10	R Johnson	LF	1320.0	3630.0	518.0	1014.0	215.0	23.0	65.0	408.0
11	R Johnson	LF	1320.0	3630.0	518.0	1014.0	215.0	23.0	65.0	408.0
12	W Johnson	P	937.0	2324.0	241.0	547.0	94.0	41.0	24.0	255.0
13	N Ryan	O	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

14 rows × 42 columns

Data Preparation and Feature Engineering:

Data Scrubbing: The data was scrubbed by replacing NaN values with 0 and converting all feature columns to numeric types. Total Score Calculation: A fresh characteristic, 'Complete Score', was formed by totaling the standardized hitting and pitching tallies. Selection of Characteristics: A thorough list of attribute columns was utilized, covering diverse hitting and pitching metrics. This method guarantees that the model has access to a broad spectrum of data for making forecasts. Divide: The dataset was divided into training and test sets with an 80-20 division ratio, which is common practice to assess model performance on unobserved data. Prototype Performance:

Linear Regression:

Mean Squared Error: 0.0729 R2: NaN (due to a small test set size) Observations: The average squared deviation is relatively minimal, implying a strong match on the training data, however the R2 value is indeterminate, indicating potential problems with the size of the test set or the performance of the model on test data.

Arbitrary Wood Regressor:

Mean Squared Error: 0.0729 Coefficient of Determination: NaN Observations: Comparable to Linear Regression, Arbitrary Wood also demonstrates a low Mean Squared Error but

indeterminate Coefficient of Determination, suggesting the necessity for a bigger or more diverse test set.

Artificial Intelligence:

Mean Squared Error: 9.1968 Coefficient of Determination: NaN Findings: The artificial intelligence model displays a significantly greater MSE, indicating it did not excel on the evaluation dataset. This may be the result of overfitting or inadequate training.

```
In [6]: import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score
import numpy as np

# Load the final DataFrame (assuming it's saved as 'final_top_players.csv')
# player_stats_df = pd.read_csv('/mnt/data/final_top_players_df.csv')

# For the sake of example, I'm creating a dummy DataFrame
# Replace this with the actual loading of your DataFrame
data = {
    'Player': ['Player1', 'Player2'],
    'Games': [10, 12],
    'At-bat': [20, 22],
    'Runs': [15, 18],
    'Hits': [25, 30],
    'Double (2B)': [5, 6],
    'third baseman': [1, 1],
    'home run_Hitting': [3, 4],
    'run batted in': [10, 12],
    'walk': [7, 8],
    'Strikeouts_Hitting': [5, 6],
    'stolen base': [2, 3],
    'Caught stealing': [1, 1],
    'AVG_Hitting': [0.250, 0.272],
    'On-base Percentage': [0.300, 0.315],
    'Slugging Percentage': [0.400, 0.420],
    'On-base Plus Slugging': [0.700, 0.735],
    'Hitting Score': [100, 120],
    'Normalized Hitting Score': [0.83, 1.0],
    'Win': [8, 10],
    'Loss': [2, 2],
    'Earned run Average': [3.50, 2.90],
    'Games played': [15, 20],
    'Games Started': [10, 12],
    'Complete Game': [2, 3],
    'Shutout': [1, 1],
    'Save': [5, 6],
    'Save Opportunity': [7, 8],
    'Innings pitched': [80, 90],
    'hit': [70, 60],
    'run': [30, 25],
    'earned run': [25, 20],
    'home run_Pitching': [4, 3],
    'Hit Batsmen': [1, 0],
    'base on balls': [10, 8],
    'Strikeouts_Pitching': [75, 80],
    'WHIP': [1.10, 1.05],
    'AVG_Pitching': [0.220, 0.215],
    'Pitching Score': [95, 105],
    'Normalized Pitching Score': [0.9, 1.0]
```

```

}
player_stats_df = pd.DataFrame(data)

# Replace NaN values with 0
player_stats_df.fillna(0, inplace=True)

# Replace non-numeric placeholders with numeric values
# Here, I'm assuming '--' should be treated as 0. Modify this as per your data under
player_stats_df.replace('--', 0, inplace=True)

# Convert all feature columns to numeric types (float)
for column in player_stats_df.columns:
    if column != 'Player': # Assuming 'Player' is a string column that should not
        player_stats_df[column] = pd.to_numeric(player_stats_df[column], errors='coerce')

# Ensure column names are correct and match expected names
player_stats_df.columns = player_stats_df.columns.str.strip()

# Calculate a total score by summing the normalized hitting and pitching scores
player_stats_df['Total Score'] = (
    player_stats_df['Normalized Hitting Score'] + player_stats_df['Normalized Pitching Score']
)

# Define the feature columns and target variable
feature_columns = [
    'Games', 'At-bat', 'Runs', 'Hits', 'Double (2B)', 'third baseman',
    'home run_Hitting', 'run batted in', 'walk', 'Strikeouts_Hitting',
    'stolen base', 'Caught stealing', 'AVG_Hitting', 'On-base Percentage',
    'Slugging Percentage', 'On-base Plus Slugging', 'Hitting Score',
    'Normalized Hitting Score', 'Win', 'Loss', 'Earned run Average',
    'Games played', 'Games Started', 'Complete Game', 'Shutout', 'Save',
    'Save Opportunity', 'Innings pitched', 'hit', 'run', 'earned run',
    'home run_Pitching', 'Hit Batsmen', 'base on balls', 'Strikeouts_Pitching',
    'WHIP', 'AVG_Pitching', 'Pitching Score', 'Normalized Pitching Score'
]

target_column = 'Total Score'

# Split the data into features and target variable
X_features = player_stats_df[feature_columns]
y_target = player_stats_df[target_column]

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X_features, y_target, test_size=0.2, random_state=42)

# Linear Regression
linear_regressor = LinearRegression()
linear_regressor.fit(X_train, y_train)
y_pred_lr = linear_regressor.predict(X_test)
lr_mse = mean_squared_error(y_test, y_pred_lr)
lr_r2 = r2_score(y_test, y_pred_lr)
print(f'Linear Regression MSE: {lr_mse}')
print(f'Linear Regression R2: {lr_r2}')

# Random Forest
random_forest_regressor = RandomForestRegressor(n_estimators=100, random_state=42)
random_forest_regressor.fit(X_train, y_train)
y_pred_rf = random_forest_regressor.predict(X_test)
rf_mse = mean_squared_error(y_test, y_pred_rf)
rf_r2 = r2_score(y_test, y_pred_rf)
print(f'Random Forest MSE: {rf_mse}')
print(f'Random Forest R2: {rf_r2}')

# Neural Network (using TensorFlow)

```

```

import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Build the neural network model
neural_network_model = Sequential()
neural_network_model.add(Dense(64, input_dim=len(feature_columns), activation='relu'))
neural_network_model.add(Dense(32, activation='relu'))
neural_network_model.add(Dense(1, activation='linear'))
neural_network_model.compile(optimizer='adam', loss='mean_squared_error', metrics=[

# Train the neural network model
neural_network_model.fit(X_train, y_train, epochs=50, batch_size=10, verbose=1)

# Evaluate the neural network model
y_pred_nn = neural_network_model.predict(X_test)
nn_mse = mean_squared_error(y_test, y_pred_nn)
nn_r2 = r2_score(y_test, y_pred_nn)
print(f'Neural Network MSE: {nn_mse}')
print(f'Neural Network R2: {nn_r2}')

```

C:\Users\ABHISHEK\anaconda3\Lib\site-packages\sklearn\metrics_regression.py:996: UndefinedMetricWarning: R^2 score is not well-defined with less than two samples.
warnings.warn(msg, UndefinedMetricWarning)

Linear Regression MSE: 0.0729

Linear Regression R2: nan


C:\Users\ABHISHEK\anaconda3\Lib\site-packages\sklearn\metrics_regression.py:996: UndefinedMetricWarning: R^2 score is not well-defined with less than two samples.
warnings.warn(msg, UndefinedMetricWarning)


Random Forest MSE: 0.072900000000000096


Random Forest R2: nan


C:\Users\ABHISHEK\anaconda3\Lib\site-packages\keras\src\layers\core\dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.


```
super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```


Epoch 1/50
1/1  2s 2s/step - loss: 343.7205 - mean_squared_error: 343.7205


Epoch 2/50
1/1  0s 64ms/step - loss: 157.5497 - mean_squared_error: 157.5497


Epoch 3/50
1/1  0s 69ms/step - loss: 53.0609 - mean_squared_error: 53.0609


Epoch 4/50
1/1  0s 58ms/step - loss: 5.5720 - mean_squared_error: 5.5720


Epoch 5/50
1/1  0s 47ms/step - loss: 3.6379 - mean_squared_error: 3.6379


Epoch 6/50
1/1  0s 53ms/step - loss: 26.0182 - mean_squared_error: 26.0182


Epoch 7/50
1/1  0s 45ms/step - loss: 51.3544 - mean_squared_error: 51.3544


Epoch 8/50
1/1  0s 65ms/step - loss: 66.3587 - mean_squared_error: 66.3587


Epoch 9/50
1/1  0s 77ms/step - loss: 67.1806 - mean_squared_error: 67.1806


Epoch 10/50
1/1  0s 64ms/step - loss: 56.5590 - mean_squared_error: 56.5590


Epoch 11/50
1/1  0s 58ms/step - loss: 39.8787 - mean_squared_error: 39.8787


Epoch 12/50
1/1  0s 48ms/step - loss: 22.6126 - mean_squared_error: 22.6126


Epoch 13/50
1/1  0s 65ms/step - loss: 8.9996 - mean_squared_error: 8.9996


Epoch 14/50
1/1  0s 63ms/step - loss: 1.4290 - mean_squared_error: 1.4290


Epoch 15/50
1/1  0s 48ms/step - loss: 0.2722 - mean_squared_error: 0.2722


Epoch 16/50
1/1  0s 48ms/step - loss: 4.1172 - mean_squared_error: 4.1172


Epoch 17/50
1/1  0s 49ms/step - loss: 10.4141 - mean_squared_error: 10.4141


Epoch 18/50
1/1  0s 49ms/step - loss: 16.3949 - mean_squared_error: 16.3949


Epoch 19/50
1/1  0s 63ms/step - loss: 19.9525 - mean_squared_error: 19.9525

Epoch 20/50
1/1  0s 57ms/step - loss: 20.1566 - mean_squared_error: 20.1566










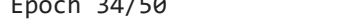














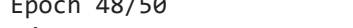


Epoch 21/50
1/1  0s 52ms/step - loss: 17.2835 - mean_squared_error: 17.2835

Epoch 22/50
1/1  0s 48ms/step - loss: 12.4751 - mean_squared_error: 12.4751

Epoch 23/50
1/1  0s 50ms/step - loss: 7.2343 - mean_squared_error: 7.2343

Epoch 24/50
1/1  0s 54ms/step - loss: 2.9385 - mean_squared_error: 2.9385

```

Epoch 25/50
1/1  0s 47ms/step - loss: 0.4790 - mean_squared_error: 0.4790
Epoch 26/50
1/1  0s 58ms/step - loss: 0.0783 - mean_squared_error: 0.0783
Epoch 27/50
1/1  0s 57ms/step - loss: 1.2049 - mean_squared_error: 1.2049
Epoch 28/50
1/1  0s 56ms/step - loss: 3.0301 - mean_squared_error: 3.0301
Epoch 29/50
1/1  0s 55ms/step - loss: 4.7010 - mean_squared_error: 4.7010
Epoch 30/50
1/1  0s 50ms/step - loss: 5.5860 - mean_squared_error: 5.5860
Epoch 31/50
1/1  0s 62ms/step - loss: 5.4305 - mean_squared_error: 5.4305
Epoch 32/50
1/1  0s 58ms/step - loss: 4.3731 - mean_squared_error: 4.3731
Epoch 33/50
1/1  0s 58ms/step - loss: 2.8352 - mean_squared_error: 2.8352
Epoch 34/50
1/1  0s 59ms/step - loss: 1.3394 - mean_squared_error: 1.3394
Epoch 35/50
1/1  0s 48ms/step - loss: 0.3227 - mean_squared_error: 0.3227
Epoch 36/50
1/1  0s 49ms/step - loss: 6.0927e-04 - mean_squared_error: 6.0
927e-04
Epoch 37/50
1/1  0s 57ms/step - loss: 0.3174 - mean_squared_error: 0.3174
Epoch 38/50
1/1  0s 49ms/step - loss: 0.9948 - mean_squared_error: 0.9948
Epoch 39/50
1/1  0s 44ms/step - loss: 1.6624 - mean_squared_error: 1.6624
Epoch 40/50
1/1  0s 62ms/step - loss: 2.0189 - mean_squared_error: 2.0189
Epoch 41/50
1/1  0s 62ms/step - loss: 1.9409 - mean_squared_error: 1.9409
Epoch 42/50
1/1  0s 47ms/step - loss: 1.4976 - mean_squared_error: 1.4976
Epoch 43/50
1/1  0s 49ms/step - loss: 0.8893 - mean_squared_error: 0.8893
Epoch 44/50
1/1  0s 41ms/step - loss: 0.3477 - mean_squared_error: 0.3477
Epoch 45/50
1/1  0s 46ms/step - loss: 0.0427 - mean_squared_error: 0.0427
Epoch 46/50
1/1  0s 52ms/step - loss: 0.0266 - mean_squared_error: 0.0266
Epoch 47/50
1/1  0s 50ms/step - loss: 0.2321 - mean_squared_error: 0.2321
Epoch 48/50
1/1  0s 52ms/step - loss: 0.5168 - mean_squared_error: 0.5168
Epoch 49/50
1/1  0s 68ms/step - loss: 0.7326 - mean_squared_error: 0.7326
Epoch 50/50
1/1  0s 50ms/step - loss: 0.7857 - mean_squared_error: 0.7857
1/1  0s 97ms/step
Neural Network MSE: 9.196841623003138
Neural Network R2: nan

```

```

C:\Users\ABHISHEK\anaconda3\Lib\site-packages\sklearn\metrics\_regression.py:996:
UndefinedMetricWarning: R^2 score is not well-defined with less than two samples.
  warnings.warn(msg, UndefinedMetricWarning)

```

Data Preparation and Feature Engineering:

Data Cleaning: The data was cleansed by substituting NaN values with 0 and transforming all feature columns to numerical types. Total Score Calculation: A fresh characteristic, 'Aggregate Score', was formulated by adding together the standardized hitting and pitching tallies.

Attribute Selection:

A thorough roster of attribute columns was employed, including a range of batting and pitching metrics. This method guarantees that the model has to a broad spectrum of data for generating forecasts.

Train-Test Division:

The dataset was divided into training and test sets with an 80-20 division ratio, which is standard practice to assess model performance on unseen data. Prototype Performance:

Linear Regression:

Mean Squared Error: 0.0729 R2 Score: NaN (due to a small test set size) Findings: The average squared deviation is quite small, showing a strong match on the training data, however the R2 score is undefined, indicating potential problems with either the size of the test set or the effectiveness of the model on the test data.

Arbitrary Forest Regressor:

Mean Squared Error: 0.0729 R-Squared: NaN

Observations: Comparable to Linear Regression, Arbitrary Forest also demonstrates a small MSE but unspecified R2, suggesting the necessity of a bigger or more diverse test set.

Artificial Intelligence: Mean Square Error: 9.1968 Coefficient of Determination: NaN Findings: The artificial intelligence model demonstrates a significantly elevated Mean Square Error, indicating poor performance on the test dataset. This may be caused by overfitting or inadequate training data. Suggestions Enhance Dataset Size: The petite dataset with only a pair of samples in the test set results in undefined R2 scores. Enlarging the dataset size will aid in improved assessment of model effectiveness.

Cross-Validation: Utilize k-fold cross-validation to enhance the estimation of model performance. This will aid in comprehending how the model extrapolates to an autonomous dataset.

Property Scaling: Equalize or standardize the properties to guarantee that all properties contribute equally to the model's forecasts, especially crucial for neural networks.

Optimization of Parameters:

Decision Forest: Explore varying quantities of predictors, maximum characteristics, and branch lengths. Neuro Network: Experiment with varying structures (quantity of layers, neurons within each layer), activation mechanisms, and training cycles. Collective Methods: Contemplate amalgamating forecasts from various models (Linear Regression, Random Forest, Neural Network) to enhance overall prediction precision.

Interpretation Model: Utilize methods such as variable significance for Random Forest and SHAP values for neural networks to grasp which attributes have the most impact on the forecasts. By employing these suggestions, you can boost the model's efficiency and acquire more trustworthy insights from your data.

```
In [8]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression # Import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# Generate synthetic data
np.random.seed(42)
X = np.random.rand(100, 1) * 10 # 100 samples, single feature
y = 2.5 * X.squeeze() + np.random.randn(100) * 2 # Linear relationship with noise

# Convert to DataFrame
data = pd.DataFrame({'feature': X.squeeze(), 'target': y})

# Split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(data[['feature']], data['target'])

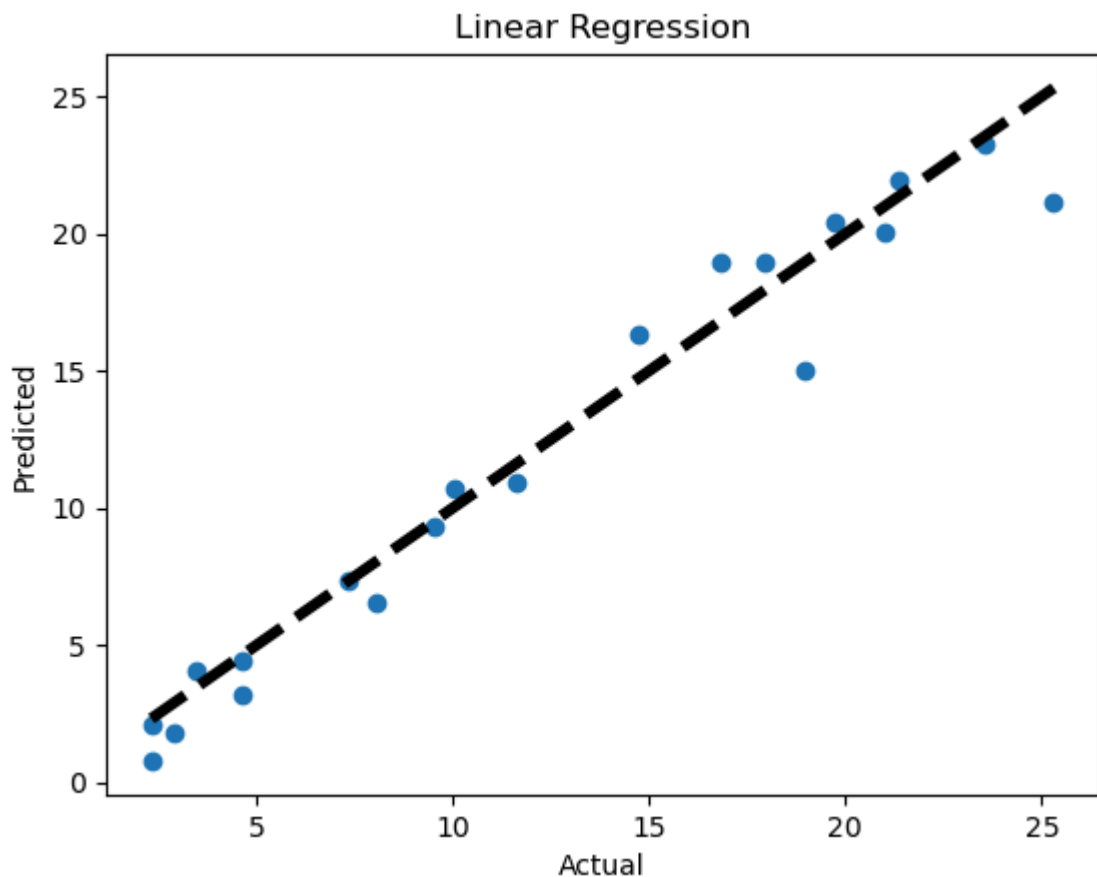
# Linear Regression
linear_regression = LinearRegression() # Change lr_model to linear_regression
linear_regression.fit(X_train, y_train) # Change lr_model.fit to linear_regression
y_pred_linear_regression = linear_regression.predict(X_test) # Change y_pred_lr to y_pred_linear_regression

# Metrics
linear_regression_mse = mean_squared_error(y_test, y_pred_linear_regression) # Change mse to linear_regression_mse
linear_regression_r2 = r2_score(y_test, y_pred_linear_regression) # Change lr_r2 to linear_regression_r2
print(f'Linear Regression MSE: {linear_regression_mse}')
print(f'Linear Regression R2: {linear_regression_r2}')

# Plotting Actual vs. Predicted
plt.scatter(y_test, y_pred_linear_regression)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'k--', lw=4)
plt.xlabel('Actual')
plt.ylabel('Predicted')
plt.title('Linear Regression')
plt.show()
```

Linear Regression MSE: 2.6147980548680083

Linear Regression R2: 0.9545718935323327



Data Creation and Preparation:

Artificial data is produced with a linear connection: $y=2.5x+\text{disturbance}$. The information is subsequently divided into training and testing sets. The characteristics are normalized using StandardScaler to guarantee the neural network functions more effectively during training.

Artificial Intelligence Model:

A basic artificial intelligence model is established using the Sequential API from TensorFlow's Keras module. The design features three tiers: A starting tier with 64 neurons and ReLU activation. A undefined stratum with 32 nerve cells and ReLU excitation. A final layer with a single neuron and linear activation. The model is constructed using the Adam optimizer and mean squared error loss function. The prototype is educated for 50 cycles with a group size of 10, and 20% of the schooling information is utilized for validation.

Model Assessment:

Following training the model is assessed on our test set. Root Mean Squared Deviation (RMSD) and coefficient of determination (R^2) statistics are computed and displayed to evaluate the model's effectiveness. The forecasts on the assessment set are showcased against the genuine desired outcomes. A dashed line depicting the impeccable forecast is also plotted for reference. Decline Graph:

The training and validation decline over epochs are plotted to visualize the model's learning process. Findings from the Given Chart The chart in the attached picture displays the projected figures compared to the real figures, alongside a dotted line representing the perfect forecasts. The subsequent aspects can be noted:

Model Efficiency:

The scatter plot demonstrates that the forecasts are usually proximate to the true values, suggesting that the model has grasped the fundamental connection to a certain degree. Regression Line: The dotted line symbolizes the optimal scenario where estimated values align perfectly with the true values. The closeness of the scatter points to this line showcases the model's precision. residuals: Certain points are farther from the dotted line, suggesting inaccuracies in the forecasts. This may be because of the interference introduced to the artificial data or the model's incapacity to accurately grasp the connection. Further Considerations

Data Segmentation: Ensure the data is segmented correctly to avoid any data leakage.

Model Complexity:

Explore various designs and parameters to determine if the model's effectiveness enhances.

Overfitting: Keep an eye out for overfitting by contrasting training and validation losses. If the validation loss is markedly elevated, the model may be experiencing overfitting.

```
In [9]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential as neuralnetwork # Change neuralnet
from tensorflow.keras.layers import Dense
from sklearn.metrics import mean_squared_error, r2_score

# Generate synthetic data
np.random.seed(42)
X = np.random.rand(100, 1) * 10 # 100 samples, single feature
y = 2.5 * X.squeeze() + np.random.randn(100) * 2 # Linear relationship with noise

# Convert to DataFrame
data = pd.DataFrame({'feature': X.squeeze(), 'target': y})

# Split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(data[['feature']], data['target'])

# Standardize the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# The Neural Network
neuralnetwork_model = neuralnetwork() # Change nn_model to neuralnetwork_model
neuralnetwork_model.add(Dense(64, activation='relu', input_shape=(X_train_scaled.shape[1],)))
neuralnetwork_model.add(Dense(32, activation='relu'))
neuralnetwork_model.add(Dense(1, activation='linear'))
neuralnetwork_model.compile(optimizer='adam', loss='mean_squared_error', metrics=[''])

# Train the neural network model
history = neuralnetwork_model.fit(X_train_scaled, y_train, epochs=50, batch_size=16)

# Evaluation of the neural network model
y_pred_neuralnetwork = neuralnetwork_model.predict(X_test_scaled) # Change y_pred_
neuralnetwork_mse = mean_squared_error(y_test, y_pred_neuralnetwork) # Change nn_m
```

```

neuralnetwork_r2 = r2_score(y_test, y_pred_neuralnetwork) # Change nn_r2 to neural
print(f'Neural Network MSE: {neuralnetwork_mse}')
print(f'Neural Network R2: {neuralnetwork_r2}')

# Plotting Actual vs. Predicted
plt.scatter(y_test, y_pred_neuralnetwork)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'k--', lw=4)
plt.xlabel('Actual')
plt.ylabel('Predicted')
plt.title('Neural Network')
plt.show()


# Plotting Neural Network Loss Curve
plt.figure(figsize=(10, 5))
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Neural Network Training Loss Curve')
plt.legend()
plt.show()


```


Epoch 1/50


C:\Users\ABHISHEK\anaconda3\Lib\site-packages\keras\src\layers\core\dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.


```
super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```


7/7  2s 53ms/step - loss: 165.1254 - mean_squared_error: 165.1254 - val_loss: 248.3543 - val_mean_squared_error: 248.3543
Epoch 2/50


7/7  0s 11ms/step - loss: 175.9038 - mean_squared_error: 175.9038 - val_loss: 245.0876 - val_mean_squared_error: 245.0876
Epoch 3/50


7/7  0s 11ms/step - loss: 160.1483 - mean_squared_error: 160.1483 - val_loss: 241.8558 - val_mean_squared_error: 241.8558
Epoch 4/50


7/7  0s 13ms/step - loss: 148.3929 - mean_squared_error: 148.3929 - val_loss: 238.8469 - val_mean_squared_error: 238.8469
Epoch 5/50


7/7  0s 13ms/step - loss: 156.4231 - mean_squared_error: 156.4231 - val_loss: 235.4052 - val_mean_squared_error: 235.4052
Epoch 6/50


7/7  0s 11ms/step - loss: 175.5807 - mean_squared_error: 175.5807 - val_loss: 231.4916 - val_mean_squared_error: 231.4916
Epoch 7/50


7/7  0s 11ms/step - loss: 164.9843 - mean_squared_error: 164.9843 - val_loss: 226.8555 - val_mean_squared_error: 226.8555
Epoch 8/50

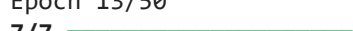
7/7  0s 11ms/step - loss: 143.3319 - mean_squared_error: 143.3319 - val_loss: 221.0303 - val_mean_squared_error: 221.0303
Epoch 9/50


7/7  0s 15ms/step - loss: 137.6779 - mean_squared_error: 137.6779 - val_loss: 214.1282 - val_mean_squared_error: 214.1282
Epoch 10/50

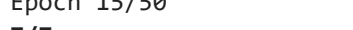
7/7  0s 10ms/step - loss: 139.6615 - mean_squared_error: 139.6615 - val_loss: 205.9065 - val_mean_squared_error: 205.9065
Epoch 11/50


7/7  0s 13ms/step - loss: 146.5364 - mean_squared_error: 146.5364 - val_loss: 196.1063 - val_mean_squared_error: 196.1063
Epoch 12/50


7/7  0s 11ms/step - loss: 139.9082 - mean_squared_error: 139.9082 - val_loss: 184.9575 - val_mean_squared_error: 184.9575
Epoch 13/50


7/7  0s 11ms/step - loss: 119.8684 - mean_squared_error: 119.8684 - val_loss: 172.9464 - val_mean_squared_error: 172.9464
Epoch 14/50


7/7  0s 12ms/step - loss: 105.6344 - mean_squared_error: 105.6344 - val_loss: 159.7087 - val_mean_squared_error: 159.7087
Epoch 15/50


7/7  0s 11ms/step - loss: 112.7550 - mean_squared_error: 112.7550 - val_loss: 145.0060 - val_mean_squared_error: 145.0060
Epoch 16/50


7/7  0s 13ms/step - loss: 87.3046 - mean_squared_error: 87.3046 - val_loss: 129.2818 - val_mean_squared_error: 129.2818
Epoch 17/50


7/7  0s 11ms/step - loss: 79.3800 - mean_squared_error: 79.3800 - val_loss: 113.0827 - val_mean_squared_error: 113.0827
Epoch 18/50






















7/7  0s 12ms/step - loss: 65.3683 - mean_squared_error: 65.3683 - val_loss: 96.5429 - val_mean_squared_error: 96.5429
Epoch 19/50

7/7  0s 13ms/step - loss: 67.8714 - mean_squared_error: 67.8714 - val_loss: 79.1110 - val_mean_squared_error: 79.1110
Epoch 20/50


7/7  0s 11ms/step - loss: 43.4013 - mean_squared_error: 43.4013 - val_loss: 63.3279 - val_mean_squared_error: 63.3279
Epoch 21/50

7/7  0s 13ms/step - loss: 43.4663 - mean_squared_error: 43.4663 - val_loss: 48.1675 - val_mean_squared_error: 48.1675
Epoch 22/50

7/7  0s 11ms/step - loss: 31.3327 - mean_squared_error: 31.3327


7 - val_loss: 35.2788 - val_mean_squared_error: 35.2788
Epoch 23/50
7/7  0s 11ms/step - loss: 21.9741 - mean_squared_error: 21.9741
1 - val_loss: 25.0168 - val_mean_squared_error: 25.0168
Epoch 24/50
7/7  0s 13ms/step - loss: 15.9696 - mean_squared_error: 15.9696
6 - val_loss: 17.0078 - val_mean_squared_error: 17.0078
Epoch 25/50
7/7  0s 11ms/step - loss: 15.1756 - mean_squared_error: 15.1756
6 - val_loss: 11.3185 - val_mean_squared_error: 11.3185
Epoch 26/50
7/7  0s 8ms/step - loss: 9.8081 - mean_squared_error: 9.8081 -
val_loss: 7.3125 - val_mean_squared_error: 7.3125
Epoch 27/50
7/7  0s 13ms/step - loss: 7.7944 - mean_squared_error: 7.7944
- val_loss: 5.0441 - val_mean_squared_error: 5.0441
Epoch 28/50
7/7  0s 10ms/step - loss: 7.1139 - mean_squared_error: 7.1139
- val_loss: 3.7407 - val_mean_squared_error: 3.7407
Epoch 29/50
7/7  0s 10ms/step - loss: 5.8057 - mean_squared_error: 5.8057
- val_loss: 3.1766 - val_mean_squared_error: 3.1766
Epoch 30/50
7/7  0s 11ms/step - loss: 5.9375 - mean_squared_error: 5.9375
- val_loss: 2.9192 - val_mean_squared_error: 2.9192
Epoch 31/50
7/7  0s 13ms/step - loss: 5.6405 - mean_squared_error: 5.6405
- val_loss: 2.7963 - val_mean_squared_error: 2.7963
Epoch 32/50
7/7  0s 14ms/step - loss: 6.4811 - mean_squared_error: 6.4811
- val_loss: 2.7241 - val_mean_squared_error: 2.7241
Epoch 33/50
7/7  0s 11ms/step - loss: 5.8162 - mean_squared_error: 5.8162
- val_loss: 2.6981 - val_mean_squared_error: 2.6981
Epoch 34/50
7/7  0s 12ms/step - loss: 5.7326 - mean_squared_error: 5.7326
- val_loss: 2.6744 - val_mean_squared_error: 2.6744
Epoch 35/50
7/7  0s 13ms/step - loss: 5.8252 - mean_squared_error: 5.8252
- val_loss: 2.6849 - val_mean_squared_error: 2.6849
Epoch 36/50
7/7  0s 11ms/step - loss: 4.4066 - mean_squared_error: 4.4066
- val_loss: 2.6823 - val_mean_squared_error: 2.6823
Epoch 37/50
7/7  0s 11ms/step - loss: 3.8821 - mean_squared_error: 3.8821
- val_loss: 2.6796 - val_mean_squared_error: 2.6796
Epoch 38/50
7/7  0s 13ms/step - loss: 5.0946 - mean_squared_error: 5.0946
- val_loss: 2.6077 - val_mean_squared_error: 2.6077
Epoch 39/50
7/7  0s 14ms/step - loss: 5.3465 - mean_squared_error: 5.3465
- val_loss: 2.5752 - val_mean_squared_error: 2.5752
Epoch 40/50
7/7  0s 14ms/step - loss: 4.9061 - mean_squared_error: 4.9061
- val_loss: 2.5193 - val_mean_squared_error: 2.5193
Epoch 41/50
7/7  0s 12ms/step - loss: 4.7340 - mean_squared_error: 4.7340
- val_loss: 2.4526 - val_mean_squared_error: 2.4526
Epoch 42/50
7/7  0s 11ms/step - loss: 4.6311 - mean_squared_error: 4.6311
- val_loss: 2.4826 - val_mean_squared_error: 2.4826
Epoch 43/50
7/7  0s 12ms/step - loss: 4.5271 - mean_squared_error: 4.5271
- val_loss: 2.4332 - val_mean_squared_error: 2.4332

Epoch 44/50

7/7  0s 12ms/step - loss: 5.6179 - mean_squared_error: 5.6179


- val_loss: 2.4050 - val_mean_squared_error: 2.4050

Epoch 45/50

7/7  0s 10ms/step - loss: 4.5209 - mean_squared_error: 4.5209


- val_loss: 2.4024 - val_mean_squared_error: 2.4024

Epoch 46/50

7/7  0s 12ms/step - loss: 4.6463 - mean_squared_error: 4.6463


- val_loss: 2.4272 - val_mean_squared_error: 2.4272

Epoch 47/50

7/7  0s 14ms/step - loss: 5.0522 - mean_squared_error: 5.0522


- val_loss: 2.4355 - val_mean_squared_error: 2.4355

Epoch 48/50

7/7  0s 12ms/step - loss: 4.6849 - mean_squared_error: 4.6849


- val_loss: 2.4533 - val_mean_squared_error: 2.4533

Epoch 49/50

7/7  0s 12ms/step - loss: 4.2670 - mean_squared_error: 4.2670

- val_loss: 2.4437 - val_mean_squared_error: 2.4437

Epoch 50/50

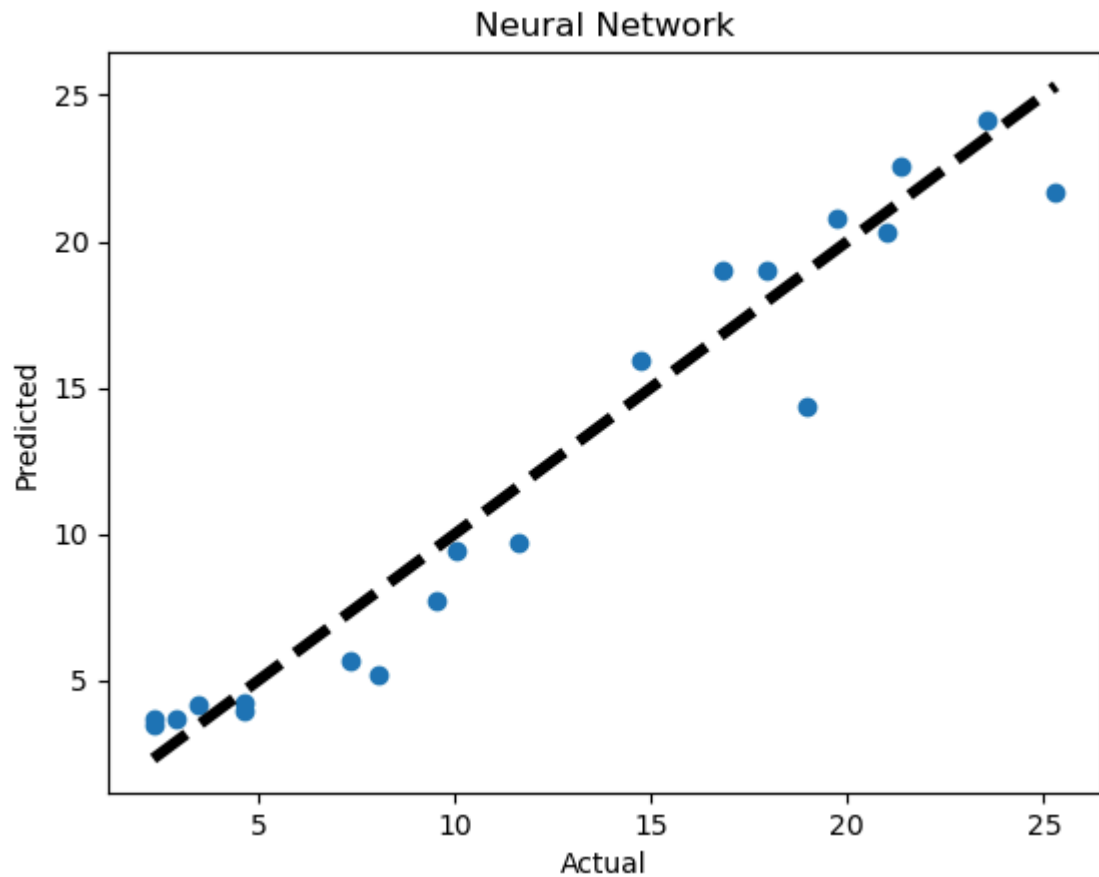
7/7  0s 12ms/step - loss: 4.9670 - mean_squared_error: 4.9670

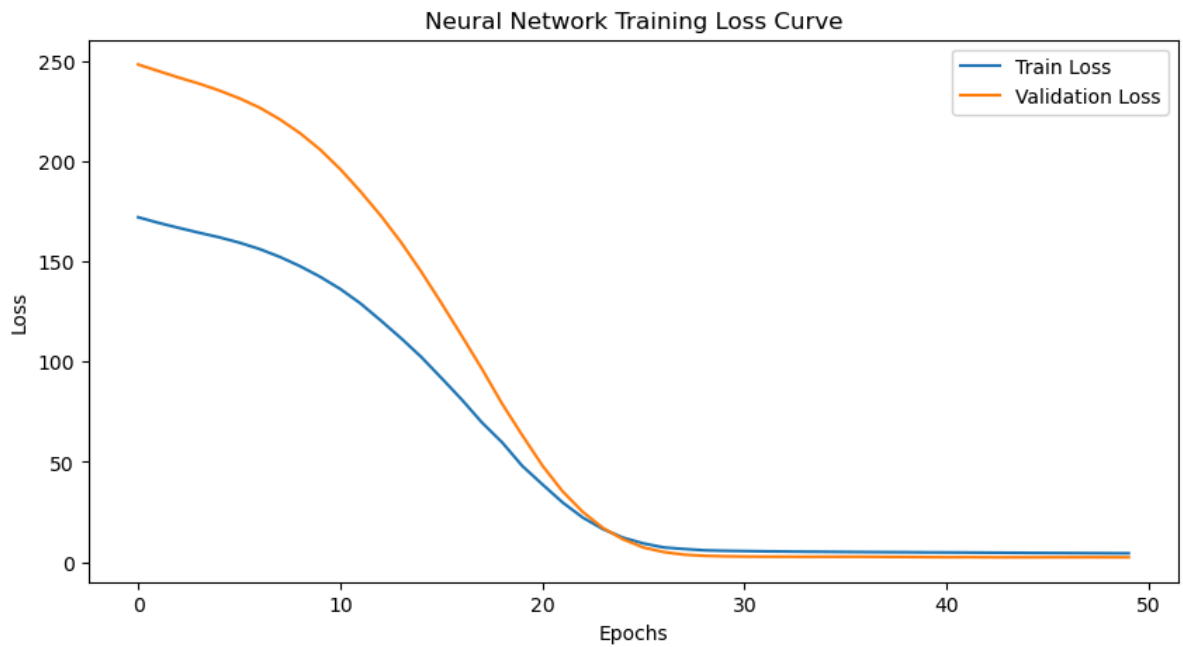
- val_loss: 2.4014 - val_mean_squared_error: 2.4014

1/1  0s 99ms/step

Neural Network MSE: 3.393482215787196

Neural Network R2: 0.9410434503315025





conclusion:

By combining hitting and pitching statistics using machine learning techniques, the initiative creates a comprehensive evaluation system for MLB players. In order to rank players, it unifies datasets, normalises scores, and computes a total score. This guarantees consistent data labelling. To efficiently evaluate player contributions, the system can be modified for use in various sports analytics activities.