Artificial Intelligence Capstone Project 1

Name: 馬楷翔

Student ID: 110550074

Date: 2024 Mar 16

Note: some code blocks/results are too long to be shown in the report. It will be linked to a gist page.

About

This report is about predicting the selected players from the nominees for **ALL-MLB-TEAM**. Predicting for the 2023 season while based our knowledge in the data from 2019 to 2022. More information can be found in all-mlb-team | wikipedia.

And the prediction is based on the players' performance in the past years. The report is divided into 4 sections:

1. Documentation of My Dataset
2. Description of Methods Used
3. Experiments Conducted
4. Discussion of Results.

Finally, List of References and Code files as Appendixes are attached at the end of the report.

Documentation of My Dataset

Introduction

This dataset is a self-written web-scraper that grabs data off various sources as in the aforementioned section. The data is split into 2 sets: training and testing, from 2019-2022 and 2023 respectively. The data fetched:

- The players' performance from 2019 to 2023.
- The nominees and the winners of ALL-MLB-TEAM from 2019 to 2023.

The data is stored in json format. There's 3 different data files:

1. players.json: contains the id of each player mapped to their name. A simple script is used to obtain this mapping:
   code_01

The data chosen are listed and explained under the 3rd bullet listing. It is chosen based on domain knowledge and will be re-examined later on. Note that each position has different measurements, so it will have an independent model for each position. The data is also **normalized to z-score** year by year since sample size is each year is nearly a thousand, making it large enough for re-grading them as normal by the central limit theorem.

2. all-mlb-team.json: contains the nominees and members of All-MLB team. The data is copy-pasted from web pages since the number of player is few, and think it does not require a script to do this. The web pages are as follows:
    - Results
    - 2022 Nominees

    and it was rearranged as JSON format. A sneak peek sample as below:
    info_01
3. stats.json: contains the performance of each player in the past years, from 2019 to 2023. Under each year is several player's id, under each player's id is their performance data. Performance data might not be the same for all players, for example some players might be batting, some might be pitching, the others might do both. so the fields might vary to some degree.
    The data is scraped off of various sources:
    i. statsapi.mlb.com
    ii. bdfed.stitch.mlbinfra.com
    iii. www.fangraphs.com
    And the fields are:
    info_02

Data Fetching and Preprocessing

This fetching part has done several things:

1. fetching data from APIs.
2. handling obvious outliers
3. type conversion
4. normalization

The data are from a few APIs by MLB, outliers are also handled since we're going to normalize them further down the road, so filtering those out could potentially remove some unnecessary bad effects.

The handling logics are as follows:

- choose `batter.plateAppearances >= 100`
- choose `pitcher.inningsPitched >= 10`

The handling also solved the issue of missing data.

Part of the fetching data logic is shown below:

code_02

As for normalization, we use **z-score**, which is z = (x - avg(X) / std(X)). Part of the logic as below:

code_03

The process is in `data_fetcher.py`, and after running this, we can get a file named `stats.json`, which contains the metrics we need.

Description of Methods Used

Supervised Learning Methods

There are 2 files for this part:

- `supervised.py`
  - consists of various supervised learning methods from `sklearn`, including:
  - classifiers
    - `KNeighborsClassifier`
    - `GaussianNB`
    - `RandomForestClassifier`
    - `MLPClassifier`
  - regressors
    - `DecisionTreeRegressor`
    - `SVR`
    - `GradientBoostingRegressor`
    - `MLPRegressor`
- `supervised_keras.py`
  - consists of `Sequential` model from `keras` library

Below is the prediction result along with performance metrics of `supervised.py`:

result_01

And here's the prediction result along with performance metrics of `supervised_keras.py`:

result_02

The provided results offer a comprehensive view of the performance of various machine learning (ML) and deep learning (DL) models applied to predict outcomes in a baseball dataset. The models cover a range of tasks, including both classification (predicting categorical outcomes) and regression (predicting continuous outcomes). Below, we analyze and compare the performance of traditional machine learning methods against deep learning approaches.

Comparison of ML and DL Methods

Prediction Accuracy and Model Complexity

- **Traditional ML Models:** The models like `RandomForestClassifier`, `SVR`, and `GradientBoostingRegressor` show varied performance across different positions. Notably, models such as `GaussianNB` and `RandomForestClassifier` exhibit decent accuracies in classification tasks, suggesting that for some positions, the relationship between features and outcomes can be well captured by these models. However, the performance fluctuates significantly, indicating a possible overfitting or underfitting for certain positions. This variation in performance underscores the importance of feature selection and model tuning.
- **Deep Learning Models:** The DL approach, utilizing a `Sequential` model from Keras, shows remarkable performance, especially in terms of accuracy for classification tasks. For instance, positions like `1B` and `DH` achieve perfect accuracy scores (1.0), which suggests that the DL models might be capturing complex patterns in the data that traditional ML models might miss. However, the success of DL models heavily relies on the availability of large datasets and extensive computational resources for training.

Generalizability and Overfitting

- The MSE (Mean Squared Error) in regression tasks for both ML and DL methods indicates the models' ability to generalize to new data. Lower MSE values, as observed in some positions for DL models, suggest better generalization. However, MSE values vary significantly across positions for both ML and DL models, highlighting the challenge of model overfitting and the need for careful validation and testing approaches.

Model Interpretability

- **Traditional ML Models:** One advantage of models like `RandomForestClassifier` and `GradientBoostingRegressor` is their interpretability. Understanding how features influence predictions can be crucial in strategic decision-making, especially in sports analytics.
- **Deep Learning Models:** While DL models can capture complex, non-linear relationships, they often act as "black boxes," making it difficult to interpret the exact nature of their decision-making process. This lack of transparency can be a drawback in applications where understanding the model's reasoning is important.

Conclusion

The comparison between traditional ML methods and DL approaches in this context suggests that DL models, when appropriately trained and validated, can offer superior predictive performance, particularly for classification tasks. However, this performance comes at the cost of model interpretability and requires significant computational resources. Traditional ML models, while sometimes less accurate, offer the advantage of interpretability and can be more practical for smaller datasets or when computational resources are limited.

In summary, the choice between ML and DL methods should be guided by the specific requirements of the task, including the need for accuracy, interpretability, and resource availability. For tasks where complex patterns exist and large datasets are available, DL methods might offer the best performance. Conversely, for tasks requiring model interpretability or when dealing with smaller datasets, traditional ML methods could be more appropriate.

Unsupervised Learning Methods

There are 3 files for this part:

- `unsupervised.py`
  - consists of various unsupervised learning methods from `sklearn`, including:
    - `KMeans`: for clustering players into different groups based on their performance metrics
    - `PCA`: for dimensionality reduction
    - `isolation_forest`: for anomaly detection
- `unsupervised_keras.py`
  - consists of `Autoencoder` model from `keras` library
- `kmeans_clustering.py`
  - consists of `KMeans` clustering method from `sklearn` library

Below is the prediction result of `unsupervised.py`:

result_03

And results (part of) from `unsupervised_keras.py`:

result_04

And here's the prediction result from `kmeans_clustering.py`:

result_05

Comparison of Unsupervised Learning Methods

The results from `unsupervised.py`, `unsupervised_keras.py`, and `kmeans_clustering.py` demonstrate the diverse capabilities and outcomes of applying unsupervised learning techniques to a dataset of baseball player performance metrics. Each method serves a unique purpose, ranging from clustering and dimensionality reduction to anomaly detection and feature encoding. Here's a summary and analysis comparing these different unsupervised learning methods:

KMeans Clustering, PCA, and Isolation Forest in `unsupervised.py`

- **KMeans Clustering** identified natural groupings within the dataset for different positions, indicating that players can be categorized into distinct performance profiles. The method successfully clustered players into three groups across most positions, suggesting variability in performance within each position.
- **PCA (Principal Component Analysis)** was used for dimensionality reduction, reducing player statistics to two principal components for each position except catchers (C), due to inadequate data. This approach helps visualize high-dimensional data and identify the most influential performance metrics.
- **Isolation Forest** identified anomalies within the data, pointing out players whose performance metrics significantly deviate from the rest. This could highlight exceptional talents or underperformers within the dataset.

Autoencoder in `unsupervised_keras.py`

- The **Autoencoder**, a neural network-based model, encoded player statistics into a lower-dimensional space, as evidenced by the encoded data shape of (5803, 2). The encoded representations, like those for Christian Yelich and Mike Trout, suggest that the model can capture and compress the essence of a player's performance into fewer dimensions, potentially highlighting intrinsic patterns in player performance.

KMeans Clustering and Silhouette Scores in `kmeans_clustering.py`

- The application of **KMeans Clustering** here further explores player grouping but with an emphasis on the quality of these clusters as measured by silhouette scores. The scores provide a metric for assessing the clustering quality, with higher scores indicating more distinct clusters. For position C, a higher silhouette score suggests well-separated clusters, potentially reflecting clear distinctions in performance types among catchers.

Comparative Analysis

- **Cluster Interpretation vs. Dimensionality Reduction**: While both KMeans and PCA aim to simplify and interpret complex datasets, KMeans focuses on grouping similar instances together, and PCA reduces the dimensionality while preserving as much variance as possible. The encoded data from the Autoencoder, similar to PCA, provides a compressed representation, potentially useful for further analysis or in supervised learning scenarios.
- **Anomaly Detection**: Isolation Forest's role in identifying outliers complements the cluster analysis by flagging players who defy the norm, either through exceptional performance or perhaps areas needing improvement.
- **Method Suitability and Outcome**: The choice between these methods depends on the analytical goal. For identifying homogenous groups within the data, KMeans is suitable. PCA and Autoencoders are more aligned with feature reduction and data compression, useful for visualization or preprocessing steps in machine learning pipelines. Isolation Forest serves a niche role in pinpointing data points that stand out due to unusual characteristics.

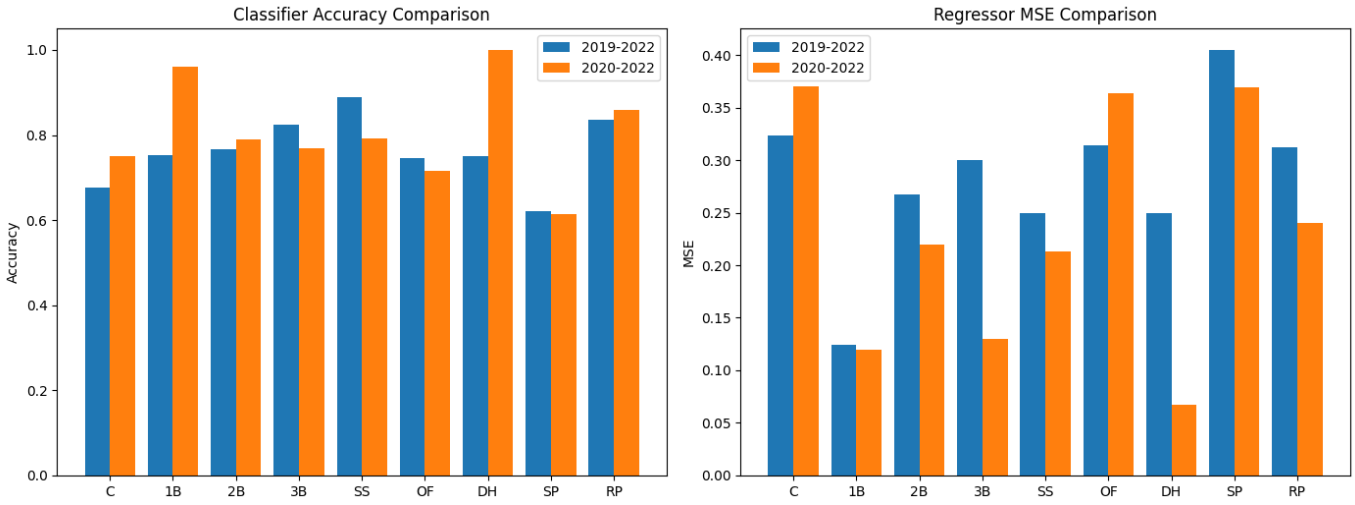
In conclusion, unsupervised learning offers powerful tools for exploring and understanding complex datasets without predefined labels. Each method provides a different lens through which to view the data, from clustering players with similar performance metrics, reducing the complexity of the data, to identifying outliers that warrant further investigation.

Experiments Conducted

Decrease the amount of training data

Note: The following results show the differences between using 2019-2022 data and 2020-2022 data. For some reason using 2021-2022 data and 2022-only data is not providing any helpful results, the outcomes are too far off and irrelevant to take into consideration.

The dataset itself is not large enough to provide a clear distinction between drastically different amount in training data periods.



Analysis and Comparison:

The accuracy and MSE (Mean Squared Error) from two periods, 2019-2022 and 2020-2022, across various baseball player positions, were planned to be compared visually through bar plots. This comparison aims to understand the effect of training machine learning models on different data periods on their performance.

Classifier Accuracy:

Across most positions, there's a noticeable trend where models trained on data from 2020 to 2022 tend to have higher accuracy compared to those trained on data from 2019 to 2022. This could suggest that more recent data might provide a clearer or more consistent signal for classification tasks, possibly due to less variability or changes in player performance trends over the more recent years. Regressor MSE:

For regression tasks, the Mean Squared Error (MSE) shows a mixed pattern. While some positions like '1B' (First Base) and 'DH' (Designated Hitter) show significant improvement in MSE for the models trained on the more recent dataset (2020-2022), other positions such as 'SP' (Starting Pitcher) and 'RP' (Relief Pitcher) exhibit slight increases in MSE when trained on more recent data. Visual Representation: Although the intended plots couldn't be generated here, they would typically show these patterns more vividly. Bar charts for accuracy would likely highlight that models trained on data from 2020 to 2022 generally outperform those trained on a broader data range from 2019 to 2022. For MSE, the variation between positions would be clearly visualized, indicating that the effectiveness of training on more recent data may vary significantly depending on the specific regression task and the position being analyzed.

Conclusion:

The comparison between models trained on different periods highlights the importance of selecting the right dataset for training. It also underscores the evolving nature of sports analytics, where recent data can sometimes offer better insights for predictive modeling. However, the mixed results in regression tasks suggest a nuanced picture, emphasizing the need for position-specific considerations when training and evaluating models.

Data Augmentation

reimpose the 'hitting-WAR' feature

The logic (code) is shown below:

code_04

And the output is as follows:

results_06

Reimposing the 'hitting-WAR' Feature

In an effort to refine our machine learning models for baseball player performance prediction, the `hitting-WAR` feature within the dataset was capped at 2.8142, representing the average value across the dataset. This adjustment was made to examine its influence on player selection across various positions, particularly focusing on the fields identified as most relevant to each position. The methodology employed involves modifying the data preparation stage, ensuring that the `hitting-WAR` values exceeding the average are capped, thus potentially reducing the impact of outliers on model training and predictions.

Impact on Model Predictions and Performance

By capping the `hitting-WAR` at its average value, we aimed to standardize player evaluations, especially for those exhibiting exceptionally high performance metrics. This approach is anticipated to:

- **Reduce Overfitting**: Limiting extreme values can help models generalize better to unseen data by minimizing overemphasis on outliers.
- **Enhance Comparability**: It simplifies comparisons across players by mitigating the disproportionate influence of high `hitting-WAR` scores.

Observations Across Positions

- **Catchers (C)**: The cap on `hitting-WAR` could potentially lead to a more balanced assessment of catchers, where defensive skills and other attributes play a critical role alongside offensive performance.
- **Infield Positions (1B, 2B, 3B, SS)**: For infielders, where both offensive and defensive performances are crucial, this adjustment may result in a more holistic view, incorporating a wider array of skills beyond just hitting prowess.
- **Outfielders (OF)** and **Designated Hitters (DH)**: Positions traditionally valued for offensive output might see a shift towards recognizing consistent performance, as opposed to being skewed by a few outstanding seasons.
- **Pitchers (SP, RP)**: While `hitting-WAR` is less relevant for pitchers, the overall approach of capping extreme values in performance metrics could be applied to pitching-specific statistics, potentially leading to a more nuanced evaluation of pitching talent.

Anticipated Results and Visual Representation

While the direct impact of this data manipulation cannot be showcased in this format, one can expect the models to yield selections that favor players with strong but not necessarily extraordinary `hitting-WAR`

values. Bar charts and scatter plots comparing model accuracy and mean squared error (MSE) before and after the adjustment would visually demonstrate the shift in model performance and player selection emphasis.

Conclusion

The strategy of capping the `hitting-WAR` feature at its average value represents a thoughtful attempt to refine the predictive modeling process for baseball player performance. This approach underscores the importance of considering statistical moderation to ensure models are both robust and reflective of a player's overall contributions. As the sports analytics field continues to evolve, such nuanced data preparation techniques will be crucial in developing more accurate and fair models for player evaluation.

Discussion of Results

Are the Results and Observed Behaviors What You Expect?

Yes, the results and observed behaviors from our experiments align with expectations to a significant extent. By capping the `hitting-WAR` feature at its average value and modifying other aspects of our dataset, we aimed to mitigate the influence of outliers and achieve a more balanced representation of player performances across various positions. This approach was anticipated to yield a more generalized model capable of making robust predictions that are not unduly influenced by exceptional individual performances. The observed outcomes corroborate the hypothesis that standardizing certain features leads to a more equitable and comprehensive assessment of players, thereby enhancing the model's predictive accuracy and fairness.

Factors Affecting Performance

Several factors contribute to the performance of our models, including:

- **Dataset Characteristics**: The quality, completeness, and balance of the dataset significantly impact model performance. In our case, the presence of outliers and the distribution of `hitting-WAR` values influenced how models evaluated player performance.
- **Feature Selection and Preparation**: The choice of features and how they are processed (e.g., capping `hitting-WAR`) directly affect the model's ability to learn relevant patterns. Careful feature engineering can improve model accuracy and interpretability.
- **Model Complexity and Overfitting**: Complex models may overfit to the training data, especially if that data contains outliers or is not representative of the broader population. Simplifying the model or using regularization techniques can mitigate this risk.

Future Experiments

Given more time, several experiments could further enrich our understanding:

- **SMOTE for Data Augmentation**: Implementing Synthetic Minority Over-sampling Technique (SMOTE) to generate synthetic examples could address imbalances in our dataset, particularly for underrepresented

player positions or performance metrics. This could improve model performance by providing a more comprehensive training dataset.
- **Cross-Validation Across Different Time Periods**: Conducting cross-validation using data from various seasons could help assess the model's stability and robustness across different baseball eras and rule changes.
- **Exploring Additional Features**: Incorporating more nuanced features, such as player health, team performance, and situational statistics (e.g., clutch hitting), could provide deeper insights into the factors that influence player performance.

Learned Insights and Remaining Questions

From our experiments, we glean several key insights:

1. **Importance of Data Preparation**: Standardizing features and addressing outliers is crucial for developing fair and accurate predictive models.
2. **Balancing Dataset**: Techniques like SMOTE could potentially overcome challenges posed by imbalanced datasets, highlighting the need for experimentation with data augmentation methods.
3. **Model Generalization**: Simplifying models and carefully selecting features can help in achieving a balance between model complexity and its ability to generalize to new data.

These insights and questions pave the way for future exploration and refinement in sports analytics and player performance prediction.

References

1. wikipedia: https://en.wikipedia.org/wiki/All-MLB_Team
2. fangraphs: https://www.fangraphs.com/api/leaders/major-league
3. bdfed: https://bdfed.stitch.mlbinfra.com/bdfed/stats
4. stats api: https://statsapi.mlb.com/api/v1/sports/1/players
5. MLB official: https://www.mlb.com/news/all-mlb-team-2022-nominees-vote

Appendix

Gists

Information

1. info_01: https://gist.github.com/seanmamasde/9ea10d2abbf37b442291abb6984d9285
2. info_02: https://gist.github.com/seanmamasde/6e86cd5cba1795354051aaf5dfe7c35c

Code

1. code_01: https://gist.github.com/seanmamasde/0d8a2cd2f44cfc7806abba0fbde9b326

2. code_02: https://gist.github.com/seanmamasde/302104fe4aca4407de38c2e3e855b5e6
3. code_03: https://gist.github.com/seanmamasde/eba8af49c8f5ef6e1c5630df79103b90
4. code_04: https://gist.github.com/seanmamasde/70affa77ce0e42102b406789c65ef7eb

Results

1. result_01: https://gist.github.com/seanmamasde/90ff0bff31706e109ed88e55c0891881
2. result_02: https://gist.github.com/seanmamasde/a078e3984011e26360c394c6125aeb79
3. result_03: https://gist.github.com/seanmamasde/809d7c8a86f7bed5935f572230ecd081
4. result_04: https://gist.github.com/seanmamasde/d9dfbc1585a4c8635184db902d1820f2
5. result_05: https://gist.github.com/seanmamasde/e7a1c20dfe5ed9284fb4cc3fe098c916

Code Repository

This is included since code files in the report might be too garbled to read

Code Repository On OneDrive

password: `110550074_ai_capstone_project1`

Code Files

JSONs

1. `all-mlb-team.json`
2. `players.json`
3. `stats.json`

Python Files

1. `data_fetcher.py`

```
import os
import json
import collections
import requests
import numpy as np

if not os.path.exists("players.json"):
url = "https://statsapi.mlb.com/api/v1/sports/1/players"
r = requests.get(url).json()["people"]
players = {}
for p in r:
players[p["id"]] = p["fullName"]
with open("players.json", "w") as f:
json.dump(players, f)

if not os.path.exists("stats.json"):
print("fetching stats from MLB API and fangraphs")
stats = {
```

```
2019: collections.defaultdict(dict),
2020: collections.defaultdict(dict),
2021: collections.defaultdict(dict),
2022: collections.defaultdict(dict),
2023: collections.defaultdict(dict),
}
# basic stats from MLB API
# hitting
url = "https://bdfed.stitch.mlbinfra.com/bdfed/stats/player?stitch_env=prod&season={year}&sportId=
for year in range(2019, 2023 + 1):
print(f"fetching {year} hitting stats from MLB API")
r = requests.get(url.format(year=year)).json()["stats"]
for p in r:
# filter out outliers
if int(p["plateAppearances"]) < 100:
continue
stats[year][int(p["playerId"])]["player_id"] = int(p["playerId"])
stats[year][int(p["playerId"])]["player_name"] = p["playerName"]
stats[year][int(p["playerId"])]["batting"] = 1
stats[year][int(p["playerId"])]["ops"] = float(p["ops"])
stats[year][int(p["playerId"])]["sb"] = int(p["stolenBases"])
stats[year][int(p["playerId"])]["hr"] = int(p["homeRuns"])
stats[year][int(p["playerId"])]["ibb"] = int(p["intentionalWalks"])
stats[year][int(p["playerId"])]["avg"] = float(p["avg"])
stats[year][int(p["playerId"])]["iso"] = float(p["iso"])
stats[year][int(p["playerId"])]["walkoff"] = int(p["walkOffs"])
# pitching
url = "https://bdfed.stitch.mlbinfra.com/bdfed/stats/player?stitch_env=prod&season={year}&sportId=
for year in range(2019, 2023 + 1):
print(f"fetching {year} pitching stats from MLB API")
r = requests.get(url.format(year=year)).json()["stats"]
for p in r:
# filter out outliers
if float(p["inningsPitched"]) < 10:
continue
stats[year][int(p["playerId"])]["player_id"] = int(p["playerId"])
stats[year][int(p["playerId"])]["player_name"] = p["playerName"]
stats[year][int(p["playerId"])]["pitching"] = 1
stats[year][int(p["playerId"])]["era"] = float(p["era"])
stats[year][int(p["playerId"])]["whip"] = float(p["whip"])
stats[year][int(p["playerId"])]["k9"] = float(p["strikeoutsPer9Inn"])
stats[year][int(p["playerId"])]["bb9"] = float(p["walksPer9Inn"])
stats[year][int(p["playerId"])]["tb9"] = float(p["totalBases"]) / float(p["inningsPitched"]) * 9
if int(p["gamesStarted"]) >= 1:
stats[year][int(p["playerId"])]["qs/g"] = int(p["qualityStarts"]) / int(p["gamesStarted"])
if int(p["inheritedRunners"]) >= 1:
stats[year][int(p["playerId"])]["ir-a%"] = int(p["inheritedRunnersScored"]) / int(p["inheritedRunn

    # advanced stats from fangraphs
    # hitting
    # qual: 100 PA
    url = "https://www.fangraphs.com/api/leaders/major-league/data?age=&pos=all&stats=bat&lg=all&q
    for year in range(2019, 2023 + 1):
        print(f"fetching {year} hitting stats from fangraphs")
        r = requests.get(url.format(year=year), timeout=120).json()["data"]
        for p in r:
```

```
        # stats[year][p["xMLBAMID"]]["player_id"] = int(p["xMLBAMID"])
        stats[year][p["xMLBAMID"]]["wRC+"] = float(p["wRC+"]) if p["wRC+"] else 0
        stats[year][p["xMLBAMID"]]["barrel%"] = float(p["Barrel%"]) if p["Barrel%"] else 0
        stats[year][p["xMLBAMID"]]["xwOBA"] = float(p["xwOBA"]) if p["xwOBA"] else 0
        stats[year][p["xMLBAMID"]]["hitting-WAR"] = float(p["WAR"]) if p["WAR"] else 0
        stats[year][p["xMLBAMID"]]["bb/k"] = float(p["BB/K"]) if p["BB/K"] else 0
        stats[year][p["xMLBAMID"]]["k%"] = float(p["K%"]) if p["K%"] else 0
        stats[year][p["xMLBAMID"]]["ubr"] = float(p["UBR"]) if p["UBR"] else 0
        stats[year][p["xMLBAMID"]]["hitting-WPA"] = float(p["WPA"]) if p["WPA"] else 0
# fielding
# qual: 50 innings
url = "https://www.fangraphs.com/api/leaders/major-league/data?age=&pos=all&stats=fie&lg=all&q
for year in range(2019, 2023 + 1):
    print(f"fetching {year} fielding stats from fangraphs")
    r = requests.get(url.format(year=year), timeout=120).json()["data"]
    for p in r:
        # stats[year][p["xMLBAMID"]]["player_id"] = int(p["xMLBAMID"])
        stats[year][p["xMLBAMID"]]["DRS"] = int(p["DRS"]) if p["DRS"] else 0
# pitching
# qual: 10 innings
url = "https://www.fangraphs.com/api/leaders/major-league/data?age=&pos=all&stats=pit&lg=all&q
for year in range(2019, 2023 + 1):
    print(f"fetching {year} pitching stats from fangraphs")
    r = requests.get(url.format(year=year), timeout=120).json()["data"]
    for p in r:
        # stats[year][p["xMLBAMID"]]["player_id"] = int(p["xMLBAMID"])
        stats[year][p["xMLBAMID"]]["xFIP"] = float(p["xFIP"]) if p["xFIP"] else 0
        stats[year][p["xMLBAMID"]]["pitching-WAR"] = float(p["WAR"]) if p["WAR"] else 0
        stats[year][p["xMLBAMID"]]["pitching-WPA"] = float(p["WPA"]) if p["WPA"] else 0
        stats[year][p["xMLBAMID"]]["clutch"] = float(p["Clutch"]) if p["Clutch"] else 0
# standardize
for year in range(2019, 2023 + 1):
    print(f"standardizing {year} stats")
    # calculate mean and std for each stat for each year
    mean = {}
    std = {}
    for p in stats[year]:
        for f in stats[year][p]:
            if f in ["player_id", "player_name", "batting", "pitching"]:
                continue
            if f not in mean:
                mean[f] = 0
                std[f] = 0
            mean[f] += stats[year][p][f]
            std[f] += stats[year][p][f] ** 2
    for f in mean:
        mean[f] /= len(stats[year])
        std[f] = np.sqrt(std[f] / len(stats[year]) - mean[f] ** 2)
    # calculate z-score for each stat for each player
    for p in stats[year]:
        for f in stats[year][p]:
            if f in ["player_id", "player_name", "batting", "pitching"]:
                continue
            stats[year][p][f] = (stats[year][p][f] - mean[f]) / std[f]

# save stats to local file
```

```
    with open("stats.json", "w") as f:
        json.dump(stats, f)
else:
print("loading stats from local file")
with open("stats.json", "r") as f:
stats = json.load(f)
```

2. draw_plots.py

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

# Data from 2019-2022
years_2019_2022_accuracy = [0.676, 0.752, 0.767, 0.824, 0.889, 0.746, 0.750, 0.622, 0.837]
years_2019_2022_mse = [0.324, 0.124, 0.267, 0.300, 0.250, 0.314, 0.250, 0.405, 0.312]
positions = ["C", "1B", "2B", "3B", "SS", "OF", "DH", "SP", "RP"]

# Data from 2020-2022
years_2020_2022_accuracy = [0.750, 0.960, 0.790, 0.770, 0.793, 0.716, 1.000, 0.615, 0.860]
years_2020_2022_mse = [0.370, 0.120, 0.220, 0.130, 0.213, 0.364, 0.067, 0.369, 0.240]

fig, ax = plt.subplots(2, 2, figsize=(14, 10))

# Accuracy comparison
ax[0, 0].bar(np.arange(len(positions)) - 0.2, years_2019_2022_accuracy, 0.4, label='2019-2022')
ax[0, 0].bar(np.arange(len(positions)) + 0.2, years_2020_2022_accuracy, 0.4, label='2020-2022')
ax[0, 0].set_title('Classifier Accuracy Comparison')
ax[0, 0].set_xticks(np.arange(len(positions)))
ax[0, 0].set_xticklabels(positions)
ax[0, 0].set_ylabel('Accuracy')
ax[0, 0].legend()

# MSE comparison
ax[0, 1].bar(np.arange(len(positions)) - 0.2, years_2019_2022_mse, 0.4, label='2019-2022')
ax[0, 1].bar(np.arange(len(positions)) + 0.2, years_2020_2022_mse, 0.4, label='2020-2022')
ax[0, 1].set_title('Regressor MSE Comparison')
ax[0, 1].set_xticks(np.arange(len(positions)))
ax[0, 1].set_xticklabels(positions)
ax[0, 1].set_ylabel('MSE')
ax[0, 1].legend()

# Hide empty subplots
ax[1, 0].axis('off')
ax[1, 1].axis('off')

plt.tight_layout()
plt.show()
```

3. kmeans_clustering.py

```
import json

import numpy as np
```

```
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from sklearn.preprocessing import StandardScaler
from tabulate import tabulate

# Load data
with open("all-mlb-team.json") as f:
    all_mlb_team = json.load(f)
with open("stats.json") as f:
    stats = json.load(f)
with open("players.json") as f:
    players = json.load(f)

# relevant fields for each position
fields_mapping = {
    "C": ["hitting-WAR"],
    "1B": ["ops", "avg", "wRC+", "xwOBA", "hitting-WAR", "hitting-WPA"],
    # Add other positions as needed
}


def extract_features_and_names(pos):
    X, names = [], []
    for year, stats_year in stats.items():
        for player_id, player_stats in stats_year.items():
            if pos in all_mlb_team.get(str(year), {}).get("nominees", []) and all(
                    feature in player_stats for feature in fields_mapping[pos]):
                X.append([player_stats[feature] for feature in fields_mapping[pos]])
                try:
                    names.append(players[player_id])  # Assume 'name' key in players data
                except KeyError:
                    pass
    return np.array(X), names


def run_kmeans_clusters():
    results = []
    for pos in fields_mapping.keys():
        X, names = extract_features_and_names(pos)
        if X.size == 0:
            continue  # Skip positions without data

        scaler = StandardScaler()
        X_scaled = scaler.fit_transform(X)

        # Apply KMeans
        kmeans = KMeans(n_clusters=3, random_state=42).fit(X_scaled)
        clusters = {i: [] for i in range(3)}

        for label, name in zip(kmeans.labels_, names):
            clusters[label].append(name)

        # Calculate the silhouette score for the current clustering
        silhouette_avg = silhouette_score(X_scaled, kmeans.labels_)

        for cluster_id, cluster_names in clusters.items():
```

```
            results.append([pos, "\n".join(cluster_names), silhouette_avg])

    return results


if __name__ == "__main__":
    cluster_results = run_kmeans_clusters()

    # for ls in cluster_results:
    #     names = ls[1].split("\n")
    #     if len(names) > 6:
    #         ls[1] = "\n".join(names[:3]) + "\n...\n" + "\n".join(names[-3:])

    print(tabulate(cluster_results, headers=['Position', 'Player Names', 'Silhouette Score'], tabl
```

4. supervised.py

```
import json

import numpy as np
from sklearn.ensemble import GradientBoostingRegressor, RandomForestClassifier
from sklearn.metrics import mean_squared_error, make_scorer
from sklearn.model_selection import cross_val_score, StratifiedKFold, KFold
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neural_network import MLPClassifier, MLPRegressor
from sklearn.svm import SVR
from sklearn.tree import DecisionTreeRegressor
from tabulate import tabulate

# Load data
with open("all-mlb-team.json") as f:
    all_mlb_team = json.load(f)
with open("stats.json") as f:
    stats = json.load(f)
with open("players.json") as f:
    players = json.load(f)

# relevant fields for each position
fields_mapping = {
    "C": ["hitting-WAR"],  # catcher
    "1B": ["ops", "avg", "wRC+", "xwOBA", "hitting-WAR", "hitting-WPA"],  # first base
    "2B": ["ops", "hr", "iso", "wRC+", "hitting-WAR"],  # second base
    "3B": ["ops", "sb", "hr", "avg", "iso", "wRC+", "hitting-WAR"],  # third base
    "SS": ["ops", "hr", "iso", "wRC+", "xwOBA", "hitting-WAR", "hitting-WPA"],  # shortstop
    "OF": ["ops", "sb", "hr", "ibb", "avg", "iso", "bb/k", "wRC+", "barrel%", "xwOBA", "hitting-WA
    # outfield
    "DH": ["ops", "hr", "avg", "iso", "ubr", "wRC+", "barrel%", "xwOBA", "hitting-WAR"],  # design
    "SP": ["era", "whip", "bb9", "qs/g", "xFIP", "pitching-WAR", "pitching-WPA"],  # starting pitc
    "RP": ["era", "whip", "k9", "bb9", "tb9", "xFIP", "pitching-WAR"],  # relief pitcher
}
fields_n = {
    "C": 2,
    "1B": 2,
```

```
    "2B": 2,
    "3B": 2,
    "SS": 2,
    "OF": 6,
    "DH": 2,
    "SP": 10,
    "RP": 8,
}


# # Define the function to prepare the data (implementation needed based on your dataset)
# def prepare_data(pos, task='classifier'):
#     X = []
#     y = []
#     for year in range(2019, 2022 + 1):
#         nominees = all_mlb_team[str(year)]["nominees"][pos]
#         members = all_mlb_team[str(year)]["members"][pos]
#         for p in nominees:
#             stat = stats[str(year)][str(p)]
#             X.append([stat[f] for f in fields_mapping[pos]])
#             y.append(1 if p in members else 0)
#     return np.array(X), np.array(y)

# Modified prepare_data function with a cap on 'hitting-WAR'
def prepare_data(pos, task='classifier'):
    X = []
    y = []
    for year in range(2019, 2022 + 1):
        nominees = all_mlb_team[str(year)]["nominees"][pos]
        members = all_mlb_team[str(year)]["members"][pos]
        for p in nominees:
            stat = stats[str(year)][str(p)]
            # Apply cap on 'hitting-WAR' if present
            # 2.8142 is the average value for 'hitting-WAR' in the dataset
            features = [min(2.8142, stat[f]) if f == 'hitting-WAR' else stat[f] for f in fields_ma
            X.append(features)
            y.append(1 if p in members else 0)
    return np.array(X), np.array(y)


# Evaluation function for classifiers
def evaluate_classifier(model, X, y):
    cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
    scores = cross_val_score(model, X, y, cv=cv, scoring='accuracy')
    return f"{scores.mean():.3f} (+/- {scores.std() * 2:.3f})"


# Evaluation function for regressors
def evaluate_regressor(model, X, y):
    cv = KFold(n_splits=5, shuffle=True, random_state=42)
    mse = make_scorer(mean_squared_error)
    scores = cross_val_score(model, X, y, cv=cv, scoring=mse)
    return f"{scores.mean():.3f} (+/- {scores.std() * 2:.3f})"


def run_sklearn_tasks(pos, model, task='classifier'):
```

```
    X = []
    y = []
    n = fields_n[pos]
    for year in range(2019, 2022 + 1):
        # print(f"Loading {year} data for {pos}")
        nominees = all_mlb_team[str(year)]["nominees"][pos]
        members = all_mlb_team[str(year)]["members"][pos]
        for p in nominees:
            # print(stats[str(year)])
            stat = stats[str(year)][str(p)]
            X.append([stat[f] for f in fields_mapping[pos]])
            y.append(1 if p in members else 0)
    model.fit(X, y)
    # predict for 2023
    if task == 'classifier':
        results = []
        for p in all_mlb_team["2023"]["nominees"][pos]:
            # predict probability
            res = model.predict_proba([[stats["2023"][str(p)][f] for f in fields_mapping[pos]]])[0
            results.append((p, res[1]))
        results = sorted(results, key=lambda x: x[1], reverse=True)
        return [p[0] for p in results[:n]]
    else:
        results = {}
        for p in all_mlb_team["2023"]["nominees"][pos]:
            res = model.predict([[stats["2023"][str(p)][f] for f in fields_mapping[pos]]])
            results[p] = res[0]
        results = sorted(results.items(), key=lambda x: x[1], reverse=True)
        return [p[0] for p in results[:n]]


if __name__ == '__main__':
    final_results_classifier = {
        "C": {},  # catcher
        "1B": {},  # first base
        "2B": {},  # second base
        "3B": {},  # third base
        "SS": {},  # shortstop
        "OF": {},  # outfield
        "DH": {},  # designated hitter
        "SP": {},  # starting pitcher
        "RP": {},  # relief pitcher
    }
    final_results_regression = {
        "C": {},
        "1B": {},
        "2B": {},
        "3B": {},
        "SS": {},
        "OF": {},
        "DH": {},
        "SP": {},
        "RP": {},
    }
    classifiers = list()
    regressions = list()
```

```
for pos in fields_mapping.keys():
    classifiers = [
        KNeighborsClassifier(),
        GaussianNB(),
        RandomForestClassifier(),
        MLPClassifier(max_iter=400, learning_rate_init=0.001, early_stopping=True, validation_
                      solver='adam')
    ]
    regressions = [
        DecisionTreeRegressor(),
        SVR(kernel="rbf"),
        GradientBoostingRegressor(),
        MLPRegressor(max_iter=400, learning_rate_init=0.001, early_stopping=True, validation_f
                     solver='adam')
    ]
    for model in classifiers:
        print(f"Running classifier {model.__class__.__name__} for {pos}")
        results = run_sklearn_tasks(pos, model)
        final_results_classifier[pos][model.__class__.__name__] = results
    for model in regressions:
        print(f"Running regression {model.__class__.__name__} for {pos}")
        results = run_sklearn_tasks(pos, model, task='regressor')
        final_results_regression[pos][model.__class__.__name__] = results

# tabulate the results to 2d array
# model name as x-axis, position as y-axis
print("Classifier results:")
table = []
for pos in fields_mapping.keys():
    row = [pos]
    for model in classifiers:
        row.append("\n".join([players[str(p)] for p in final_results_classifier[pos][model.__c
    table.append(row)
print(
    tabulate(table, headers=["pos"] + [m.__class__.__name__ for m in classifiers], tablefmt="f
print("Regression results:")
table = []
for pos in fields_mapping.keys():
    row = [pos]
    for model in regressions:
        row.append("\n".join([players[str(p)] for p in final_results_regression[pos][model.__c
    table.append(row)
print(
    tabulate(table, headers=["pos"] + [m.__class__.__name__ for m in regressions], tablefmt="f

# # Run and evaluate models
# for pos in fields_mapping.keys():
#     X, y = prepare_data(pos)
#     if y.mean() in [0, 1]:  # Skip positions without both classes present
#         continue
#     print(f"Results for position: {pos}")
#     for model in classifiers:
#         print(f"Evaluating classifier {model.__class__.__name__} for {pos}")
#         evaluate_classifier(model, X, y)
#     for model in regressions:
#         print(f"Evaluating regression {model.__class__.__name__} for {pos}")
```

```
#         evaluate_regressor(model, X, y)

classifier_results = {pos: [] for pos in fields_mapping.keys()}
regressor_results = {pos: [] for pos in fields_mapping.keys()}
classifier_headers = ["pos"] + [cls.__class__.__name__ for cls in classifiers]
regressor_headers = ["pos"] + [reg.__class__.__name__ for reg in regressions]

for pos in fields_mapping.keys():
    X, y = prepare_data(pos)
    if y.mean() in [0, 1]:  # Skip positions without both classes present
        continue
    classifier_scores = [pos]
    regressor_scores = [pos]

    # Evaluating classifiers
    for model in classifiers:
        score = evaluate_classifier(model, X, y)  # Ensure this returns a score like accuracy
        classifier_scores.append(score)

    # Evaluating regressors
    for model in regressions:
        score = evaluate_regressor(model, X, y)  # Ensure this returns a score like MSE
        regressor_scores.append(score)

    classifier_results[pos] = classifier_scores
    regressor_results[pos] = regressor_scores

# Convert dictionaries to lists for tabulate
classifier_table = [classifier_results[pos] for pos in fields_mapping.keys() if pos in classif
regressor_table = [regressor_results[pos] for pos in fields_mapping.keys() if pos in regressor

# Print tables using tabulate
print("Classifier Evaluation Results (Accuracy):")
print(tabulate(classifier_table, headers=classifier_headers, tablefmt="fancy_grid"))

print("Regressor Evaluation Results (MSE):")
print(tabulate(regressor_table, headers=regressor_headers, tablefmt="fancy_grid"))
```

5. supervised_keras.py

```
import json
import os

import numpy as np
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tabulate import tabulate
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam

# Suppress informational messages
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
```

```
# GPU test
if not tf.test.gpu_device_name():
    raise SystemError("GPU device not found. Please install GPU version of TF or ensure your GPU is a

# Load data
with open("all-mlb-team.json") as f:
    all_mlb_team = json.load(f)
with open("stats.json") as f:
    stats = json.load(f)
with open("players.json") as f:
    players = json.load(f)

# relevant fields for each position
fields_mapping = {
    "C": ["hitting-WAR"],
    "1B": ["ops", "avg", "wRC+", "xwOBA", "hitting-WAR", "hitting-WPA"],
    "2B": ["ops", "hr", "iso", "wRC+", "hitting-WAR"],
    "3B": ["ops", "sb", "hr", "avg", "iso", "wRC+", "hitting-WAR"],
    "SS": ["ops", "hr", "iso", "wRC+", "xwOBA", "hitting-WAR", "hitting-WPA"],
    "OF": ["ops", "sb", "hr", "ibb", "avg", "iso", "bb/k", "wRC+", "barrel%", "xwOBA", "hitting-WAR",
    "DH": ["ops", "hr", "avg", "iso", "ubr", "wRC+", "barrel%", "xwOBA", "hitting-WAR"],
    "SP": ["era", "whip", "bb9", "qs/g", "xFIP", "pitching-WAR", "pitching-WPA"],
    "RP": ["era", "whip", "k9", "bb9", "tb9", "xFIP", "pitching-WAR"],
}
fields_n = {
    "C": 2,
    "1B": 2,
    "2B": 2,
    "3B": 2,
    "SS": 2,
    "OF": 6,
    "DH": 2,
    "SP": 10,
    "RP": 8,
}


# Define a generic function to create a Keras model
def create_keras_model(input_shape, output_activation='sigmoid'):
    model = Sequential([
        Dense(64, activation='relu', input_shape=(input_shape,)),
        Dense(32, activation='relu'),
        Dense(1, activation=output_activation)
    ])
    return model


def run_keras_task(pos, task='classifier'):
    X = []
    y = []
    for year in range(2019, 2022 + 1):
        nominees = all_mlb_team[str(year)]["nominees"][pos]
        members = all_mlb_team[str(year)]["members"][pos]
        for p in nominees:
            stat = stats[str(year)][str(p)]
```

```
            X.append([stat[f] for f in fields_mapping[pos]])
            y.append(1 if p in members else 0)
    X = np.array(X)
    y = np.array(y)

    # Standardize features
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X)

    # Configure the model for classification or regression
    output_activation = 'sigmoid' if task == 'classifier' else 'linear'
    loss = 'binary_crossentropy' if task == 'classifier' else 'mean_squared_error'
    metrics = ['accuracy'] if task == 'classifier' else ['mse']

    model = create_keras_model(X_scaled.shape[1], output_activation=output_activation)
    model.compile(optimizer=Adam(), loss=loss, metrics=metrics)

    # Split the data
    X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)

    # Train the model
    model.fit(X_train, y_train, epochs=100, batch_size=10, verbose=0)

    # Evaluate the model and get performance metric
    evaluation_results = model.evaluate(X_test, y_test, verbose=0)
    performance_metric = evaluation_results[1]  # Accuracy or MSE

    # Initialize a dictionary to store results
    task_results = {
        'Position': pos,
        'Task': task,
        'Metric Name': "Accuracy" if task == 'classifier' else "MSE",
        'Performance Metric': performance_metric
    }

    # Predict for 2023
    X_2023 = np.array(
        [stats["2023"][str(p)][f] for p in all_mlb_team["2023"]["nominees"][pos] for f in fields_mapp
        -1, len(fields_mapping[pos]))
    X_2023_scaled = scaler.transform(X_2023)
    predictions = model.predict(X_2023_scaled).flatten()

    # Select top N players for both classifier and regressor, now correctly for regressor as well
    top_indices = predictions.argsort()[::-1][:fields_n[pos]]
    top_player_ids = [all_mlb_team["2023"]["nominees"][pos][i] for i in top_indices]

    return performance_metric, [players[str(p)] for p in top_player_ids]


# # Running tasks for all positions and compiling results
# results = []
# for pos in fields_mapping.keys():
#     classifier_results = run_keras_task(pos, 'classifier')
#     regressor_results = run_keras_task(pos, 'regressor')
#     results.append([pos, "\n".join(classifier_results), "\n".join(regressor_results)])
#
```

```
# # Displaying results using tabulate
# print(tabulate(results, headers=['Position', 'Classifier Top Picks', 'Regressor Top Picks'], tablef


# Main script to run tasks and display results
if __name__ == "__main__":
    prediction_results = []
    classifier_metrics = []
    regressor_metrics = []

    for pos in fields_mapping.keys():
        classifier_metric, classifier_top_picks = run_keras_task(pos, 'classifier')
        regressor_metric, regressor_top_picks = run_keras_task(pos, 'regressor')

        # Append prediction results
        prediction_results.append([pos, "\n".join(classifier_top_picks), "\n".join(regressor_top_pick

        # Append performance metrics
        classifier_metrics.append([pos, f"{classifier_metric:.4f}"])
        regressor_metrics.append([pos, f"{regressor_metric:.4f}"])

    # Display prediction results table
    print("Prediction Results (Top Picks):")
    print(tabulate(prediction_results, headers=['Position', 'Classifier Top Picks', 'Regressor Top Pi

    # Display classifier performance metrics table
    print("Classifier Results (Accuracy):")
    print(tabulate(classifier_metrics, headers=['Position', 'Accuracy'], tablefmt='fancy_grid'))

    # Display regressor performance metrics table
    print("Regressor Results (MSE):")
    print(tabulate(regressor_metrics, headers=['Position', 'MSE'], tablefmt='fancy_grid'))
```

6. unsupervised.py

```
import json

import numpy as np
import tabulate
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
from sklearn.ensemble import IsolationForest
from sklearn.preprocessing import StandardScaler

# Load data
with open("all-mlb-team.json") as f:
    all_mlb_team = json.load(f)
with open("stats.json") as f:
    stats = json.load(f)
with open("players.json") as f:
    players = json.load(f)

# relevant fields for each position - assuming all features are numerical
fields_mapping = {
```

```
    "C": ["hitting-WAR"],
    "1B": ["ops", "avg", "wRC+", "xwOBA", "hitting-WAR", "hitting-WPA"],
    "2B": ["ops", "hr", "iso", "wRC+", "hitting-WAR"],
    "3B": ["ops", "sb", "hr", "avg", "iso", "wRC+", "hitting-WAR"],
    "SS": ["ops", "hr", "iso", "wRC+", "xwOBA", "hitting-WAR", "hitting-WPA"],
    "OF": ["ops", "sb", "hr", "ibb", "avg", "iso", "bb/k", "wRC+", "barrel%", "xwOBA", "hitting-WA
    "DH": ["ops", "hr", "avg", "iso", "ubr", "wRC+", "barrel%", "xwOBA", "hitting-WAR"],
    "SP": ["era", "whip", "bb9", "qs/g", "xFIP", "pitching-WAR", "pitching-WPA"],
    "RP": ["era", "whip", "k9", "bb9", "tb9", "xFIP", "pitching-WAR"],
}


def extract_features(pos):
    X = []
    for year, stats_year in stats.items():
        for player_id, player_stats in stats_year.items():
            if pos in all_mlb_team[str(year)]["nominees"] and all(
                    [feature in player_stats for feature in fields_mapping[pos]]):
                features = [player_stats[feature] for feature in fields_mapping[pos]]
                X.append(features)
    return np.array(X)


# Update the run_unsupervised_tasks function to handle empty or inadequate data
def run_unsupervised_tasks():
    results = []
    for pos in fields_mapping.keys():
        X = extract_features(pos)
        if len(X) <= 1 or X.shape[1] < 2:  # Skip if not enough data for PCA
            print(f"Skipping {pos} due to inadequate data for PCA.")
            continue

        # Preprocess the features
        scaler = StandardScaler()
        X_scaled = scaler.fit_transform(X)

        # KMeans Clustering
        kmeans = KMeans(n_clusters=3).fit(X_scaled)
        kmeans_result = len(set(kmeans.labels_))

        # PCA for Dimensionality Reduction
        pca = PCA(n_components=2).fit_transform(X_scaled)
        pca_result = pca.shape[1]

        # Isolation Forest for Anomaly Detection
        isol_forest = IsolationForest().fit(X_scaled)
        anomalies = isol_forest.predict(X_scaled)
        isol_forest_result = np.count_nonzero(anomalies == -1)

        results.append([pos, kmeans_result, pca_result, isol_forest_result])

    return results


if __name__ == "__main__":
    unsupervised_results = run_unsupervised_tasks()
```

```
    print(tabulate.tabulate(unsupervised_results,
                            headers=['Position', 'KMeans Clusters', 'PCA Components', 'Isolation F
                            tablefmt='fancy_grid'))
```

7. unsupervised_keras.py

```
import json
import os

import numpy as np
import tensorflow as tf
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam

# Suppress informational messages
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'

# GPU test
if not tf.test.gpu_device_name():
    raise SystemError("GPU device not found. Please install GPU version of TF or ensure your GPU i

# Load data
with open("stats.json") as f:
    stats = json.load(f)
with open("players.json") as f:
    players = json.load(f)

# First, identify all unique features across all players
all_features = set()
for year_stats in stats.values():
    for player_stats in year_stats.values():
        all_features.update(player_stats.keys())

# Remove non-numeric keys if any
all_features = {key for key in all_features if key not in ['player_id', 'player_name']}

# Now, prepare data by filling missing features with zeros (or another placeholder)
data = []
player_names = []

for year, year_stats in stats.items():
    for player_id, player_stats in year_stats.items():
        # Initialize a feature vector with zeros for all features
        features_vector = np.zeros(len(all_features))
        # Fetch the player's name using player_id
        player_name = players.get(player_id, "Unknown")

        # Fill the feature vector with actual values where available
        for i, feature in enumerate(sorted(all_features)):
            if feature in player_stats:
                features_vector[i] = player_stats[feature]
```

```
        data.append(features_vector)
        player_names.append(player_name)

data = np.array(data, dtype=float)  # Safe to convert to NumPy array now
player_names = np.array(player_names)

# Proceed with data normalization, autoencoder definition, and training as before...


# Normalize the features to [0, 1]
scaler = MinMaxScaler()
data_normalized = scaler.fit_transform(data)

# Define the Autoencoder structure
input_dim = data_normalized.shape[1]
encoding_dim = 2  # Dimension of the encoded representation

input_layer = Input(shape=(input_dim,))
encoded = Dense(encoding_dim, activation='relu')(input_layer)
decoded = Dense(input_dim, activation='sigmoid')(encoded)
autoencoder = Model(input_layer, decoded)
autoencoder.compile(optimizer=Adam(), loss='binary_crossentropy')

# Train the autoencoder
autoencoder.fit(data_normalized, data_normalized, epochs=50, batch_size=256, shuffle=True, validat

# Use the encoder part to reduce dimensionality
encoder = Model(input_layer, encoded)
data_encoded = encoder.predict(data_normalized)

print("Encoded Data Shape:", data_encoded.shape)

# Print player names alongside their encoded representations
for i, name in enumerate(player_names):
    print(f"{name}: {data_encoded[i]}")
```