Machine Learning & Data Mining
(SOFE 4620U)

NBA Lineup Prediction Report

Group Members: (CRN 74292 Group 20)

Name	Student ID
Inder Singh	100816726
Justin Fisher	100776303
Rohan Radadiya	100704614

Due Date: March 19, 2025

Table of Contents

1. Introduction

In professional sports, especially in basketball, strategic decision-making plays a crucial role in determining team success. With the increasing availability of historical data and advanced analytics, machine learning has become an important tool for identifying performance patterns and improving decisions [1]. In this context, the NBA provides a rich dataset containing detailed game records, including team lineups, player statistics, and game outcomes. Predictive modeling can be used to analyze this data and provide insights into which player combinations contribute most effectively to team success.

This project explores the application of machine learning techniques to enhance team performance in professional basketball by predicting the optimal fifth player for the home team lineup in an NBA game. Utilizing historical NBA game data from the 2007 to 2015 seasons, a model is developed to analyze partial team lineups and relevant game-related features to recommend the most suitable player who would maximize the home team's overall performance. The model is built using XGBoost, a high-performance gradient-boosting algorithm that is capable of handling large and complex datasets and delivering accurate predictions [2]. The process involves data cleaning, feature selection based on predefined constraints, model training, model predictions, and performance evaluation. The goal is to demonstrate how machine learning can contribute to data-driven decision-making in sports and to identify the most influential factors in predicting player performance within a team setting.

2. Import Libraries

Before processing the data, we import the essential Python libraries for data handling, feature engineering, model training, and visualization purposes. We import Pandas for data manipulation and preprocessing, NumPy for numerical computations, and sklearn for preprocessing, encoding, and evaluation metrics. We also import XGBoost to use as the primary machine learning algorithm for classification and Matplotlib for visualizing insights from the data.

```
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from collections import Counter
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
import xgboost as xgb
import random
from sklearn.metrics import classification_report
```

Figure 2.1 Imported libraries

Loading these libraries ensures that all necessary tools are available for model development.

3. Data Preprocessing

Several important steps were required to prepare the data for the machine learning model. This section explains how the raw NBA game data was cleaned and prepared. The first step was to select only the features allowed in the project using a metadata file. After that, all the matchup files from different seasons were combined into one large dataset. Any missing or empty values were removed to keep the dataset clean and reliable. Then, since machine learning models work better with numbers, all team and player names were converted into numerical labels using label encoding. Some players didn't appear enough times in the data, so those rare classes were removed to prevent problems during prediction. The dataset was then split into training and test sets in a way that kept the player distribution balanced. It was also important to check if any player classes were present in the test set but missing from the training set, and if appropriate fixes were made. Finally, players who still had very low sample counts were removed so the model could focus on learning from stronger patterns.

3.1 Extract Allowed Features

Before training the model, it was important to filter the dataset and only use the features that were allowed according to the project guidelines. A metadata file was provided that listed which features could be used. The metadata file was first loaded and read, as shown below.

```
metadata_path = "data/Matchup-metadata.xlsx"
metadata_df=pd.read_excel(metadata_path)

#view metadata
metadata_df
```

Figure 3.1 Loading metadata

Once the metadata is loaded, all features that have an “x” marked under “Can be used in the model” are selected and saved within a list.

```
allowed_features = metadata_df[metadata_df["Can be used in the model "] == "x"]["Feature"].tolist()
allowed_features.append("outcome")

allowed_features
```

✓ 0.0s

```
['game',
 'season',
 'home_team',
 'away_team',
 'starting_min',
 'home_0',
 'home_1',
 'home_2',
 'home_3',
 'home_4',
 'away_0',
 'away_1',
 'away_2',
 'away_3',
 'away_4',
 'outcome']
```

Figure 3.2 Allowed features loaded

3.2 Load and Merge NBA Matchup Data

After identifying which features were allowed, the next step was to load the actual NBA matchup data. This data came from several CSV files, with each file representing a specific NBA season from 2007 to 2015. These files included details such as player lineups, team information, and various game-related statistics.

```
data_folder = "data"
df_list = []
for file in os.listdir(data_folder):
    if file.endswith('.csv'):
        file_path = os.path.join(data_folder, file)
        # Read only the allowed columns from each CSV
        df_subset = pd.read_csv(file_path, usecols=allowed_features)
        df_list.append(df_subset)

df = pd.concat(df_list, ignore_index=True)
df
```

Figure 3.3 Loading and merging data

To begin, each CSV file was read one by one. During this process, only the allowed features that were selected in the previous step were extracted from each file. This ensured that no extra or disallowed data was unintentionally included in the model. Once all the files were loaded, they were combined into a single dataset. This function merged the data from all seasons into one comprehensive DataFrame, making it easier to manage and process in the following stages.

	game	season	home_team	away_team	starting_min	home_0	home_1	home_2	home_3	home_4	away_0	away_1	away_2	away_3	away_4	outcome
0	201410290SAC	2015	SAC	GSW	0	Ben McLemore	Darren Collison	DeMarcus Cousins	Jason Thompson	Rudy Gay	Andrew Bogut	Draymond Green	Harrison Barnes	Klay Thompson	Stephen Curry	-1
1	201410290SAC	2015	SAC	GSW	7	Ben McLemore	Darren Collison	DeMarcus Cousins	Jason Thompson	Rudy Gay	Draymond Green	Festus Ezeli	Harrison Barnes	Klay Thompson	Stephen Curry	-1
2	201410290SAC	2015	SAC	GSW	8	Carl Landry	Darren Collison	DeMarcus Cousins	Nik Stauskas	Rudy Gay	Andre Iguodala	Festus Ezeli	Klay Thompson	Marreese Speights	Stephen Curry	-1
3	201410290SAC	2015	SAC	GSW	9	Carl Landry	Darren Collison	DeMarcus Cousins	Derrick Williams	Nik Stauskas	Andre Iguodala	Festus Ezeli	Klay Thompson	Marreese Speights	Stephen Curry	-1
4	201410290SAC	2015	SAC	GSW	10	Carl Landry	Derrick Williams	Nik Stauskas	Ramon Sessions	Reggie Evans	Andre Iguodala	Festus Ezeli	Klay Thompson	Marreese Speights	Stephen Curry	-1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
236907	200901210CHA	2009	CHA	MEM	36	Adam Morrison	Emeka Okafor	Juwan Howard	Raja Bell	Sean Singletary	Darrell Arthur	Hakim Warrick	Kyle Lowry	Marko Jaric	O.J. Mayo	-1
236908	200901210CHA	2009	CHA	MEM	38	Adam Morrison	Boris Diaw	Juwan Howard	Raymond Felton	Sean Singletary	Darrell Arthur	Hakim Warrick	Kyle Lowry	Marko Jaric	O.J. Mayo	-1
236909	200901210CHA	2009	CHA	MEM	40	Adam Morrison	Boris Diaw	Juwan Howard	Raymond Felton	Sean Singletary	Darrell Arthur	Hakim Warrick	Kyle Lowry	O.J. Mayo	Rudy Gay	-1
236910	200901210CHA	2009	CHA	MEM	41	Adam Morrison	Boris Diaw	Juwan Howard	Raymond Felton	Sean Singletary	Hakim Warrick	Marc Gasol	Mike Conley	O.J. Mayo	Rudy Gay	-1
236911	200901210CHA	2009	CHA	MEM	42	Boris Diaw	Emeka Okafor	Gerald Wallace	Raja Bell	Raymond Felton	Hakim Warrick	Marc Gasol	Mike Conley	O.J. Mayo	Rudy Gay	-1

236912 rows × 16 columns

Figure 3.4 - Data Loading and Merging Results

3.3 Handle Missing Values

At this stage, we check for any missing values in the dataset. If any rows have missing data, we drop them to ensure the model doesn't train on incomplete or inconsistent information. Missing data can arise due to recording errors, incomplete statistics, or unavailable player information. We drop the values if there are any with this command and verify afterward if all of the missing values are dropped.

```
df = df.dropna()
```

```
Missing values after cleaning:
 game           0
season          0
home_team       0
away_team       0
starting_min    0
home_0          0
home_1          0
home_2          0
home_3          0
home_4          0
away_0          0
away_1          0
away_2          0
away_3          0
away_4          0
outcome         0
dtype: int64
```

Figure 3.5 - Missing values

By ensuring that only complete rows are used, we improve the model's stability and reliability.

3.4 Label Encode Columns

Any non-numeric (categorical) columns are converted into a numerical format. XGBoost cannot work directly with text data, and the algorithm needs everything in numerical form to make sense of it. For that, label encoding comes into play. Label encoding is a technique that assigns a unique number to each category in a column. This allows the model to interpret these values during training and make meaningful predictions.

In this stage, columns such as game, home_team, away_team, and the player position columns (home_0 to home_3, and away_0 to away_4) are label encoded. The home_4 column is intentionally excluded from this step since it's handled separately later on in the process.

```python
Column_names = ["game","home_team","away_team","home_0","home_1","home_2","home_3","away_0","away_1","away_2","away_3","away_4"]

label_encoders = {}
for col in Column_names:
    if col in df.columns:
        le_col = LabelEncoder()
        df[col] = le_col.fit_transform(df[col].astype(str))
        label_encoders[col] = le_col
    else:
        print(f"Warning: Column {col} not found in df.")
```

Figure 3.6 - Column names utilized

```
df
✓ 0.0s                                                    Python
```

	game	season	home_team	away_team	starting_min	home_0	home_1	home_2	home_3	home_4	away_0	away_1	away_2	away_3	away_4	outcome
0	9610	2015	29	10	0	86	196	150	257	Rudy Gay	49	256	283	381	542	-1
1	9610	2015	29	10	7	86	196	150	257	Rudy Gay	289	301	283	381	542	-1
2	9610	2015	29	10	8	134	196	150	538	Rudy Gay	41	301	486	469	542	-1
3	9610	2015	29	10	9	134	196	150	96	Nik Stauskas	41	301	486	469	542	-1
4	9610	2015	29	10	10	134	228	632	592	Reggie Evans	41	301	486	469	542	-1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
236907	3076	2009	3	15	36	7	281	442	590	Sean Singletary	228	335	497	466	392	-1
236908	3076	2009	3	15	38	7	72	442	604	Sean Singletary	228	335	497	466	392	-1
236909	3076	2009	3	15	40	7	72	442	604	Sean Singletary	228	335	497	544	485	-1
236910	3076	2009	3	15	41	7	72	442	604	Sean Singletary	369	599	605	544	485	-1
236911	3076	2009	3	15	42	103	281	247	590	Raymond Felton	369	599	605	544	485	-1

236912 rows × 16 columns

Figure 3.7 - Label Encoded Columns

This step is crucial because it transforms all the categorical data into a format that the machine learning model can understand and work with, ensuring smoother training and more accurate results.

3.5 Rare Class Removal

We remove rarely occurring players in the target column (home_4), as they do not provide enough data for the model to learn meaningful patterns. When a player appears only once or very few times in the dataset, the model cannot generalize or make accurate predictions for that class, which introduces noise and reduces the overall performance of the model. Additionally, if a rare player appears in the test set but was never seen during training, it can cause errors or unreliable results. By removing these rare classes before splitting the data, we ensure a more balanced and effective training process.

```
# Before removing rare classes
print("Unique target classes BEFORE removal:", df["home_4"].nunique())
print("Number of classes with <2 samples BEFORE removal:", sum(df["home_4"].value_counts() < 2))
```

✓ 0.0s

```
Unique target classes BEFORE removal: 660
Number of classes with <2 samples BEFORE removal: 31
```

Figure 3.8 before removing rare classes

```
if "home_4" not in df.columns:
    raise ValueError("Column 'home_4' not found. Please ensure it is included in the data.")

freq = df["home_4"].value_counts()
rare_classes = freq[freq < 2].index
df = df[~df["home_4"].isin(rare_classes)].copy()

X = df.drop(columns=["home_4"])
y_raw = df["home_4"]

# Single encoder for the target
target_le = LabelEncoder()
y_full = target_le.fit_transform(y_raw)  # Encoded target for the entire dataset
```

✓ 0.0s

```
# After removing rare classes
print("Unique target classes AFTER removal:", df["home_4"].nunique())
print("Number of classes with <2 samples AFTER removal:", sum(df["home_4"].value_counts() < 2))
```

✓ 0.0s

```
Unique target classes AFTER removal: 629
Number of classes with <2 samples AFTER removal: 0
```

Figure 3.9 Output after removing rare classes

We can see from the figures above that the number of unique target classes went from 660 to 629 after performing the removal.

3.6 Train-Test Split

After cleaning and preparing the dataset, the next key step is splitting the data into training and testing sets. This is a standard practice in machine learning that helps make sure the model is trained on one part of the data and then evaluated on a separate part to see how well it performs on new, unseen data.

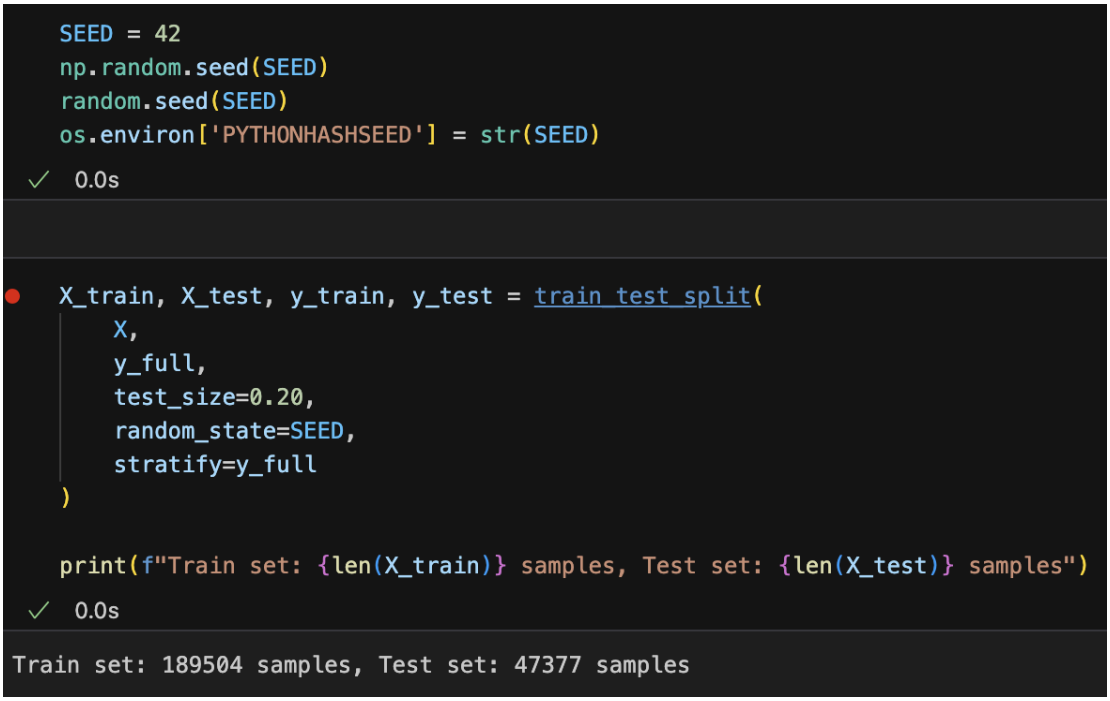

Figure 3.10 Train and test set samples

In this step, 80% of the dataset is used for training the model, while the remaining 20% is set aside for testing. The training set allows the model to learn patterns and relationships from past NBA games, and the test set is used to evaluate how accurately the model can predict the fifth player — specifically when it hasn't seen that data before. To make sure the different player classes are represented fairly in both the training and testing sets, a technique called stratified sampling is used. This means that the distribution of player labels in the target column (home_4) is kept consistent across both sets. This helps the model learn in a more balanced way and perform better across different types of players.

```
X_train = X_train.reset_index(drop=True)
X_test  = X_test.reset_index(drop=True)

# Convert y to Pandas Series with the same indexing
y_train = pd.Series(y_train, name="home_4").reset_index(drop=True)
y_test  = pd.Series(y_test,  name="home_4").reset_index(drop=True)
```

Figure 3.11 Resetting training/testing set indexes

The indexes for both the training set (X_train) and test set (X_test) are reset. This ensures that the rows in each dataset start from a clean and consistent index. Resetting the index helps maintain a well-organized structure, especially after data splitting, so that each row can be easily referenced during model training and evaluation. After preparing the input features, the target columns (y_train and y_test) are also converted into proper Pandas Series. This makes them easier to work with and ensures their indexes match the corresponding feature datasets. This step is essential to keep the features and labels properly aligned before model training.

X_test

✓ 0.0s

	game	season	home_team	away_team	starting_min	home_0	home_1	home_2	home_3	away_0	away_1	away_2	away_3	away_4	outcome
0	4306	2010	26	28	43	29	21	214	334	42	443	454	398	484	1
1	10121	2015	4	33	34	1	445	633	712	217	226	224	172	595	-1
2	6610	2012	11	33	36	188	332	467	392	128	226	215	140	435	-1
3	5540	2011	25	26	44	290	343	314	254	41	281	439	421	568	1
4	5873	2011	18	12	19	225	472	466	488	114	178	145	634	605	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
47372	6133	2011	28	15	25	43	317	493	452	228	599	605	544	496	1
47373	6457	2012	10	29	11	55	82	143	517	243	354	361	307	609	-1
47374	10114	2015	26	6	40	451	508	584	531	418	449	475	478	599	-1
47375	6443	2012	18	14	14	259	640	634	615	50	26	355	473	602	-1
47376	2347	2008	15	23	36	118	418	428	389	234	304	324	459	454	1

47377 rows × 15 columns

Figure 3.12 Results

The X_test dataset is shown above to confirm the structure of the data. Each row represents a single NBA game and includes information such as the season, teams involved, player positions, and the outcome column. It's important to note that some rows still contain -1 in the outcome column, indicating missing or placeholder values that need to be addressed.

```python
if 'outcome' in X_test.columns:
    mask_outcome = (X_test['outcome'] == -1)
    indices_to_move = X_test.index[mask_outcome]

    X_move = X_test.loc[indices_to_move]
    y_move = y_test.loc[indices_to_move]

    # Drop them from test
    X_test.drop(indices_to_move, inplace=True)
    y_test.drop(indices_to_move, inplace=True)

    # Reset index again
    X_test.reset_index(drop=True, inplace=True)
    y_test.reset_index(drop=True, inplace=True)

    # Concatenate these rows into train
    X_train = pd.concat([X_train, X_move], ignore_index=True)
    y_train = pd.concat([y_train, y_move], ignore_index=True)
```

Figure 3.13 Removing outcome=-1 rows from test set

The code snippet above checks for rows in the test set that have outcome = -1. As per the guidelines, each test sample must have an outcome of 1, where the home team's performance is supposed to be better than the away team's performance. Once the rows with missing outcomes are identified, they are removed from both the feature set (X_test) and the target set (y_test). After removing the invalid rows, the indexes of the test set are reset again to maintain a clean and continuous sequence. This step keeps the dataset tidy and helps prevent errors in future processing steps. Once this is complete, these rows are moved back to the training set.

```python
num_negative_outcomes = (X_test["outcome"] == -1).sum()
print(f"\nNumber of outcome = -1 in test set after removal:")
num_negative_outcomes
```

✓ 0.0s

```
Number of outcome = -1 in test set after removal:

np.int64(0)
```

Figure 3.14 outcome=-1 rows in test set results after removal

```
X_test
```

✓ 0.0s

	game	season	home_team	away_team	starting_min	home_0	home_1	home_2	home_3	away_0	away_1	away_2	away_3	away_4	outcome
0	4306	2010	26	28	43	29	21	214	334	42	443	454	398	484	1
1	5540	2011	25	26	44	290	343	314	254	41	281	439	421	568	1
2	5873	2011	18	12	19	225	472	466	488	114	178	145	634	605	1
3	5860	2011	23	33	26	33	106	331	403	14	93	183	571	435	1
4	7970	2013	16	6	21	157	263	506	461	30	241	501	431	599	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
17710	2341	2008	30	10	11	329	352	303	445	54	53	461	516	544	1
17711	9922	2015	30	23	42	103	152	352	390	181	350	661	581	592	1
17712	8802	2014	0	3	33	448	578	664	575	66	62	83	165	437	1
17713	6133	2011	28	15	25	43	317	493	452	228	599	605	544	496	1
17714	2347	2008	15	23	36	118	418	428	389	234	304	324	459	454	1

17715 rows × 15 columns

Figure 3.15 Output of data for X_test

A final preview of the X_test dataset is shown to confirm that all rows with missing outcomes have been successfully removed. The test set is now fully cleaned and contains only meaningful data on which the model can be evaluated.

3.7 Handling Missing Classes

After splitting the dataset into training and test sets, there may be a chance that some player labels might only show up in the test set and not in the training set. This creates a problem because the model can't accurately predict something it has never seen before. If a player label is missing from the training data, the model simply has no way to learn how to recognize it.

```
all_train_classes = np.unique(y_train)
all_test_classes  = np.unique(y_test)

missing_labels = np.setdiff1d(all_test_classes, all_train_classes)
print("Labels present in test but missing in train:", missing_labels)
```

✓ 0.0s

```
Labels present in test but missing in train: []
```

Figure 3.16 Handling missing classes in training

As shown above, the dataset is checked to identify those player labels that exist in the test set but are completely absent from the training set. These missing labels need to be handled carefully to avoid confusion or prediction errors during testing. In this case, there aren't any labels missing; however, if there are, they must be handled.

```
# Add synthetic entries for missing labels
for label in missing_labels:
    # Random row from X_train
    random_idx = np.random.choice(X_train.index)
    row_to_duplicate = X_train.loc[[random_idx]]

    # Append to X_train
    X_train = pd.concat([X_train, row_to_duplicate], ignore_index=True)

    # Append the missing label
    y_train = pd.concat([y_train, pd.Series([label])], ignore_index=True)

# Re-check
all_train_classes = np.unique(y_train)
missing_after_fix = np.setdiff1d(all_test_classes, all_train_classes)
print("Missing labels after fix:", missing_after_fix)
```

✓ 0.0s

```
Missing labels after fix: []
```

Figure 3.17 handling missing lables

To fix this, a synthetic training example is created for each missing label. This is done by randomly picking a row from the training set and duplicating it but assigning the missing player label to that new row. By doing this, the model is at least exposed to every player label it will encounter during testing, even if only through a basic placeholder. While this isn't a highly advanced or useful solution in this case, it's a practical and useful way to make the model more robust. It ensures that no player label in the test set goes completely unseen by the model, which helps improve fairness and reduces the risk of prediction errors during evaluation.

3.8 Removing Low-Frequency Classes

Once the missing labels have been addressed, the next step is to improve the model's learning by removing low-frequency classes. In other words, these are players who show up only a few times in the training set. These rare player labels don't give the model enough examples to learn any meaningful patterns about their behavior or performance. To handle this, the training data is analyzed to count how often each player label appears. If a player shows up fewer than a certain number of times, they're considered a sparse class. Since there isn't enough data for the model to learn anything reliable from these players, they are filtered out.

```
min_samples = 3
sparse_classes = [cls for cls, count in freq_train.items() if count < min_samples]
print("Sparse classes (< {} samples): {}".format(min_samples, sparse_classes))
```

```
Sparse classes (< 3 samples): [26, 62, 253, 23, 33, 29, 8, 75, 480, 269, 42, 551, 245, 2, 55, 129, 373, 186, 13, 9, 108, 90, 7, 60, 283, 40, 66, 268, 102, 536]
```

Figure 3.18 sparse classes

Rows associated with these sparse classes are removed from both the training and test sets. This keeps the dataset focused on players with enough representation, which helps the model learn more effectively and make stronger, more accurate predictions.

```
mask_train = ~y_train.isin(sparse_classes)
X_train_filtered = X_train.loc[mask_train].reset_index(drop=True)
y_train_filtered = y_train.loc[mask_train].reset_index(drop=True)

mask_test = ~y_test.isin(sparse_classes)
X_test_filtered = X_test.loc[mask_test].reset_index(drop=True)
y_test_filtered = y_test.loc[mask_test].reset_index(drop=True)
```

```
final_le = LabelEncoder()
y_train_enc = final_le.fit_transform(y_train_filtered)
y_test_enc  = final_le.transform(y_test_filtered)
```

Figure 3.19 sets are ready

By removing low-frequency classes, the model becomes less distracted by rare and inconsistent data. It instead focuses on learning from players who appear often enough to provide clear, useful training signals. This not only improves the model's overall performance, but also helps reduce noise and overfitting in the learning process.

4. Model Training

Once the data preprocessing is completed, the next step is to train a machine learning model. For this project, the XGBoost classifier is used due to its high performance and efficiency in handling structured data and classification problems [2].

```
params = {
    'subsample': 1.0,
    'n_estimators': 300,
    'max_depth': 7,
    'learning_rate': 0.1,
    'colsample_bytree': 0.8,
    'tree_method': 'hist',
    'device': 'cuda',
    'predictor': 'gpu_predictor',
    'n_jobs': 1,
    'eval_metric': 'mlogloss'
}

model = xgb.XGBClassifier(**params)

#Evaluate on the training set itself to show iteration progress.
eval_set = [(X_train_filtered, y_train_enc)]
model.fit(
    X_train_filtered,
    y_train_enc,
    eval_set=eval_set,
    verbose=True
)
✓ 23m 12.0s
```

Figure 4.1

The model is then initialized using these parameters, and training is performed using the fit() function. During training, an evaluation set is specified so that XGBoost can monitor the training loss (mlogloss) at each iteration and show progress over time.

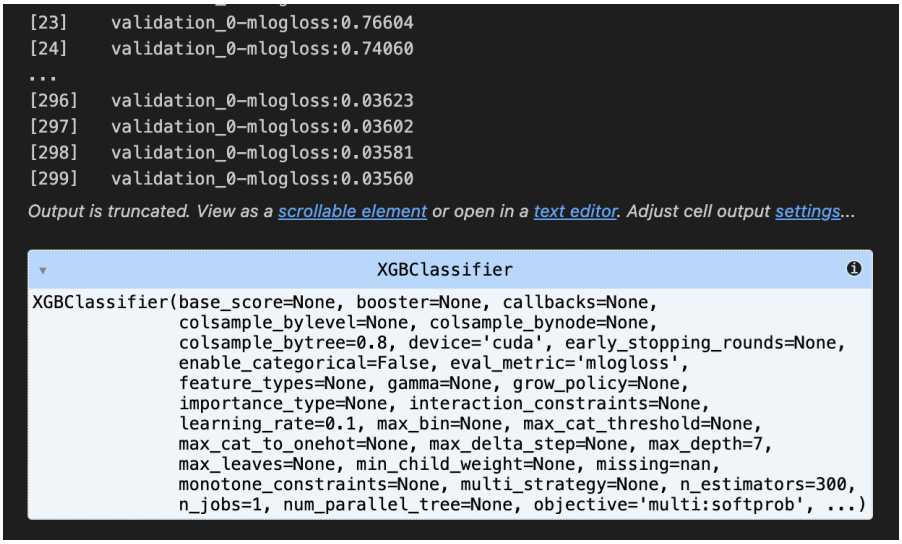

Figure 4.2 trained model

The full configuration of the trained XGBoost model is displayed, confirming that all settings were successfully applied. This training step is essential because it allows the model to learn from the training data by iteratively improving its predictions, minimizing error, and capturing meaningful patterns that help in predicting the optimal fifth player in unseen game situations.

5. Model Explainability

One way to get a better understanding of how the model makes decisions is by looking at how well it learns during training. The first chart below, titled “XGBoost Training Log Loss over Iterations,” helps visualize this process. As the model goes through each boosting round (or iteration), it gradually improves its predictions. Log loss is a metric that measures how far off the model’s predictions are from the actual player labels, and lower values mean better performance. In the beginning, the log loss is quite high, but it steadily decreases as training continues, eventually reaching a final value of 0.0356. This low number shows that the model is learning effectively and making accurate predictions with minimal error.

Figure 5.1 Graph showing training log loss

Another way to understand the model's behavior is through feature importance, which highlights which inputs had the most influence on the model's predictions. The second chart below, titled "Top 10 Most Important Features (by Gain)," ranks the features based on how much they contributed to improving the model's accuracy.

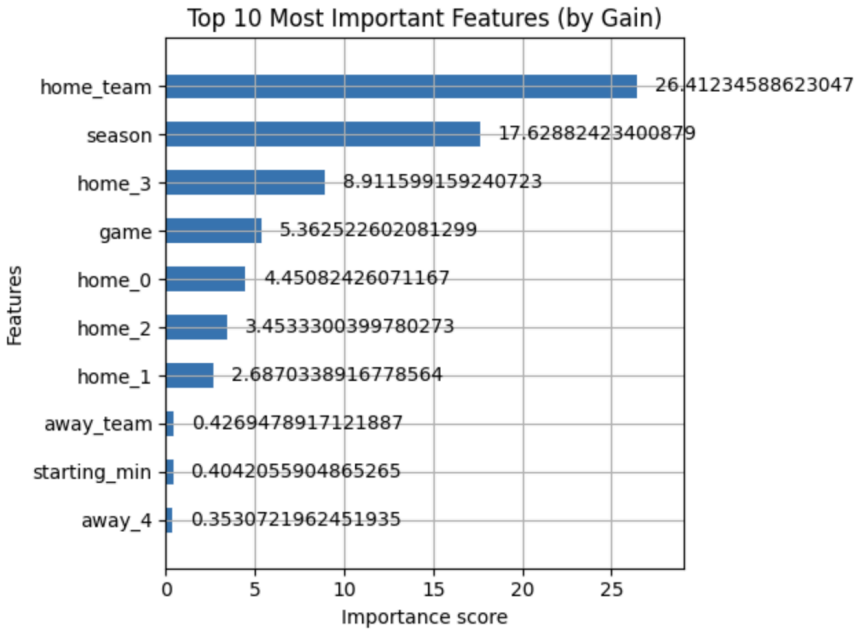

Figure 5.2 graph showing which features the model thought to be most important

In this case, home_3, home_team, and season came out as some of the most impactful features. That means the identity of the fourth player in the home lineup (home_3), which team is playing at home (home_team), and the season of the game all played a big role in helping the

model decide who the fifth player should be. These features provided some of the most valuable information during training. Other features such as the second player in the home lineup (home_1), the game ID, the first player in the home lineup (home_0) and the third player in the home lineup (home_2) also played a key role in helping the model decide the fifth player, but not as much as the features mentioned before. In contrast, the other features played much less of a role in deciding.

Together, these two visualizations help make the model's decision-making more transparent. The training log loss graph shows that the model is learning efficiently, and the feature importance chart explains what data the model relied on most. This gives a clearer picture of why certain players are predicted as the fifth team member. Not through guessing but because specific features strongly influence those decisions.

6. Prediction Outputs

After training and evaluating the model, the final step is to generate predictions using the test dataset. This project aims to predict the optimal fifth player for the home team lineup based on the features provided in each test sample. These predictions represent the players that the model believes would most improve the home team's performance in a given game scenario.

```
#Predict on test set
preds_enc = model.predict(X_test_filtered)

#Decode predictions fully back to original player labels
#First map to filtered class labels, then back to original labels using both encoders
mapped_preds = final_le.classes_[preds_enc]
preds_decoded = target_le.inverse_transform(mapped_preds)

#Decode actual y_test values similarly
mapped_actuals = final_le.classes_[y_test_enc]
actual_decoded = target_le.inverse_transform(mapped_actuals)
#Decode home_team back to original label using label encoder
decoded_home_teams = label_encoders['home_team'].inverse_transform(X_test_filtered['home_team'])
#Decode game back to original label using label encoder
decoded_games = label_encoders['game'].inverse_transform(X_test_filtered['game'])

prediction_output = pd.DataFrame({
    'game': decoded_games,
    'home_team': decoded_home_teams,
    'home_04': preds_decoded
})
```

Figure 6.1 generating predictions on test set

```
#Save prediction output to CSV
prediction_output.to_csv("prediction-outputs.csv", index=False)
print("Prediction file 'prediction-outputs.csv' created successfully.")
```

✓ 0.0s

```
Prediction file 'prediction-outputs.csv' created successfully.
```

Figure 6.2 predictions saved to csv

First, the model is applied to the filtered test feature set (X_test_filtered) to produce predicted class labels. These labels are encoded values that correspond to different players. To make the predictions understandable and meaningful, the output labels are then decoded back to their original player identifiers using a label encoder (final_le). This decoding step is essential because it translates the model's numerical predictions into recognizable player IDs or names. Once the predictions are generated, the results are organized into a structured format shown below with three key columns: Game, Home Team, and Predicted Fifth Player. These predictions are then saved in a CSV file called prediction-outputs.csv, which makes the results easy to share, analyze, or review.

	A	B	C	D	E
1	game	home_tea	home_04		
2	201001200	PHI	Marreese Speights		
3	201101190	ORL	Ryan Anderson		
4	201103090	MIN	Wayne Ellington		
5	201103070	NYK	Toney Douglas		
6	201302240	MIA	Shane Battier		
7	201501070	MIN	Thaddeus Young		
8	200903090	MIA	Michael Beasley		
9	200701230	CHI	Luol Deng		
10	201411070	DET	Kentavious Caldwell-Pope		

Figure 6.3 csv results

7. Model Evaluation

After making predictions on the test dataset, it's important to evaluate how well the model performed. One of the most effective ways to do this is by using a classification report, which provides several key performance metrics, such as accuracy, precision, recall, and F1 score for each predicted player class. These metrics give a deeper understanding of the model's strengths and any areas that may need improvement.

	precision	recall	f1-score	support
Chucky Atkins	0.00	0.00	0.00	1
Corey Brewer	0.00	0.00	0.00	1
Cuttino Mobley	0.86	1.00	0.92	6
Danilo Gallinari	0.00	0.00	0.00	2
DeAndre Jordan	0.88	1.00	0.93	7
DeMar DeRozan	1.00	1.00	1.00	1
DeShawn Stevenson	0.72	0.90	0.80	20
Dee Brown	0.33	1.00	0.50	1
Deron Williams	0.00	0.00	0.00	0
Derrick Williams	0.00	0.00	0.00	0
Devean George	1.00	1.00	1.00	1
Devin Harris	1.00	1.00	1.00	1
Dominic McGuire	0.00	0.00	0.00	0
Dorell Wright	0.00	0.00	0.00	1
Earl Watson	1.00	0.67	0.80	3
Elton Brand	0.50	1.00	0.67	1
Enes Kanter	1.00	1.00	1.00	6
Eric Gordon	0.85	0.85	0.85	13
Eric Snow	0.00	0.00	0.00	1
Ersan Ilyasova	0.75	0.50	0.60	6
Etan Thomas	1.00	1.00	1.00	2
Evan Fournier	1.00	1.00	1.00	2
Francisco Elson	1.00	0.33	0.50	3
accuracy			0.86	17713
macro avg	0.73	0.69	0.70	17713
weighted avg	0.86	0.86	0.85	17713

Figure 7.1 classification report of model predictions

In this project, the classification report summarizes how accurately the model predicted the fifth player for the home team in each test sample. The overall accuracy was 86%, which means that in 86% of the cases, the model correctly identified the right fifth player. This is a strong result and indicates that the model is performing well overall.

The report also includes precision and recall scores for each player class. Precision measures how many of the model's predictions for a given player were actually correct, while recall shows how many of the actual correct players the model was able to successfully identify. The F1 score combines both precision and recall into a single value, giving a more balanced view of the model's performance, which is especially important when some player classes appear less frequently than others. The report also provides two overall averages to evaluate the model more broadly. The Macro Average F1 Score was 0.70, which treats all player classes equally,

regardless of how often they appear. This helps assess how well the model performs across both common and rare players. On the other hand, the Weighted Average F1 Score was 0.85, which gives more weight to frequently occurring player classes. This shows that the model performs particularly well on players that appear more often in the dataset.

While some player classes showed lower precision and recall, often due to having very few samples in the test set, the high-weighted F1 score suggests that the model is still making strong and reliable predictions for the majority of cases. This indicates solid performance overall while also pointing out opportunities for improvement, especially when dealing with low-frequency player classes.

8. Challenges and Limitations

Throughout this project, several challenges and limitations had to be addressed to ensure the success of the machine learning workflow. One of the most significant challenges was class imbalance, where certain player labels appeared much more frequently in the dataset than others. This imbalance made it harder for the model to learn meaningful patterns about rare or low-frequency players, which often led to lower precision and recall scores for those particular classes. Since the model naturally tends to focus on the more common classes during training, the rare ones are often overlooked, reducing prediction quality for those cases.

Another key challenge was handling missing values and rare target classes. Some data entries had missing outcomes or very few examples of specific players, which required additional data preprocessing steps such as filtering, encoding, and cleaning. These steps were necessary to prevent potential errors during training and to ensure that the model could work with a clean, structured dataset.

There were also technical limitations that impacted the consistency of results. Slight differences in computing environments, such as hardware specifications, caused variations in model performance across different machines. These inconsistencies made it difficult to reproduce the same results every time. To address this, steps like setting random seeds and

standardizing code execution were implemented to improve reproducibility, ensuring that the model could be reliably evaluated across different systems.

Additionally, tuning the model's hyperparameters was a time-consuming and resource-intensive task. Given the large size of the dataset, running multiple configurations to find the optimal parameter settings required considerable computing power and patience. This process, while essential for improving model performance, added to the overall complexity of the project.

9. Conclusion

In conclusion, this project successfully built a machine learning model using the XGBoost algorithm to predict the optimal fifth player for a home team in an NBA game. The model was designed to make predictions based on partial lineup information and other relevant game-related features. The process started with data preprocessing, which included tasks like cleaning the dataset, handling missing values, label encoding categorical features, removing low-frequency classes, and applying a stratified train-test split to ensure balanced representation across player classes. The model was trained on historical NBA data spanning from 2007 to 2015, using only the allowed features defined in the metadata file. To evaluate the model's performance, several techniques were used, including classification reports, log loss visualization, feature importance analysis, and a review of the prediction outputs. The results showed strong model performance, with an overall test accuracy of 86%, a weighted F1 score of 0.85, and a macro-average F1 score of 0.70, indicating that the model generalizes well across both frequent and less common player classes. Overall, this project demonstrates how a machine learning model can be applied to real-world sports analytics. It highlights the power of data-driven decision-making and shows how predictive models can uncover valuable insights in team lineup optimization in professional basketball.

References

[1] S. J. Ahmed, “Machine learning in sports analytics and performance prediction,” Medium, https://medium.com/dataduniya/machine-learning-in-sports-analytics-and-performance-prediction-d7f50799f684 (accessed Mar. 19, 2025).

[2] Simplilearn, “What is xgboost? an introduction to XGBoost algorithm in Machine Learning: Simplilearn,” Simplilearn.com, https://www.simplilearn.com/what-is-xgboost-algorithm-in-machine-learning-article (accessed Mar. 19, 2025).