Machine Learning & Data Mining

NBA Lineup Prediction

SOFE 4620U

Name	Student ID
Javier Chung	100785653
Laksumi Somaskanthamoorthy	100782723
Zainab Nomani	100784761

Github Link: https://github.com/Javiebear/NBALineupPredictor/tree/main

1. Introduction

Machine learning models are becoming more dominant in today's time being introduced to many industries and sectors, even being prominent in sports. The NBA has been using different prediction models to predict player's strengths and weaknesses, optimal player pairings, and optimal team combinations. This project leverages machine learning techniques to gather historical NBA game data from 2007-2015 and predicts the optimal fifth player for a home team starting lineup. The objective of this project is to develop a machine learning algorithm that can predict the optimal fifth player of a home team lineup when given, the season, starting minute, both team names, four home team players and five away team players.

2. Methodology

There are three main methods that are used in our pipeline which is to train the model, and make predictions based on inputs from the test file. The methods contain aquireData, trainModel and predictTests.

2.1 model.py

There are three main methods in this script, aquireData, trainModel, and predictTests each being a pipeline of the process of predicting the fifth player in the lineup.

```
def aquireData(csv):
  " Processing "
  return x, y, rosters, playerToIndex, teamToIndex
```

This method is used to process the raw data from the csv file and convert it into features that will be used to train the model. Within this method, the features are extracted and then engineered to return features that can be easily learned by the model. To ensure consistency and accuracy between the training and prediction data, the data is normalized. This method returns the appended features list *x*, and the outcome list *y* that the model will be trained on. A dictionary value roster is returned as well, which represents the rosters of every team in the dataset. The final variables returned represent the dictionaries used to translate players and teams to indexed values.

```
def trainModel(x,y):
  " train model "
  return (model)
```

This method is used to train the model with the parameters x and y where x was the features used to train the model and y were the target classifiers for the model. Once finished training, the model was returned.

```
def predictTests(rosters, tests, testResults, playerToIndex, teamToIndex,
freqOfEachPlayer, synergyMatrix, winRateMatrix, model, predData):


  return (correctResults)
```

The final part of the process is the predictTests function which is used to predict the fifth player of each example with the given test data. The parameters include preprocessed data acquired from the aquireData method, as well as other parameters that included the synergy matrix of all pairs of players, the average win rate of all home teams, the model, and the csv file of where the data would be stored. The method returned the correct results to calculate for accuracy.

3. Feature Engineering

Not all features from the datasets are permitted for use to train the model with the usable ones being the home team name, away team name, 5 home players, 5 away players, starting minute and season. In our program, the model is being trained with all 10 players, the season and the starting minute as input features against the outcome values.

The first values being feature engineered were the season and the starting minute. They were normalized to the range of 0 - 1. The season value was normalized between the values of 2007 - 2016 while the starting minute was normalized between the values of 0 and 47. The team names also needed to be standardized as with different years, some NBA teams changed their names, so to keep consistency, those teams needed to be converted to one value which in our case we made it the most recent value. In the model, we focused on using the 5 players from both teams to engineer more features to train the model. Initially the players were represented as raw strings which was not trainable for the model, so each player was converted into an index value. This was done by creating a dictionary that would set an index for each unique player in the dataset. This would make both the home and away lineups be represented as an array of indexes. The total inputs included: indexed home, away lineup, starting minute and the season.

4. Selected Model

The model that was selected for this project was decision trees as it was easier to visualize, implement and interpret. The model we used to implement it in Python was the Extreme Gradient Boosting (XGBoost), which is a gradient-boosted decision tree (GBDT). This model was chosen because it supports supervised machine learning, decision trees, and gradient boost [1] . Gradient boosting is a machine learning algorithm that specializes in regression, classification and ranking [2]. This model uses decision trees to help with prediction by evaluating the decision trees if-then-else features. XGBoost has been one of the best models to use for predicting data.

5. Training Model

The training model used in this model is called the XGBoost Classifier, where the amount of decision trees are established, *n_estimators*, which is the amount of decision trees used to predict the optimal fifth player. Each decision tree goes down to 50 layers and has a learning rte of 0.001 which is considered slow. Since the amount of decision trees are high, the model uses a subsample of 80% to reduce the chances of overfitting.

```
def trainModel(x,y):
    model = xgb.XGBClassifier(
        n_estimators=1000,
        max_depth=50,
        learning_rate=0.001,
        subsample=0.8,
        colsample_bytree=0.8,
        use_label_encoder=False,
        eval_metric='logloss'
    )
    model.fit(x, y)
```

```
        return (model)
```

6. Preprocessing and Postprocessing

The data was initially loading in from a csv file and required some preprocessing before training. To train the model accurately, the inconsistencies needed to be fixed. With the same teams having different names over the season, they had to be processed and converted to have the same representation. Values like the season outcome and starting minute had to also be processed in order to be trainable on our model.

1. Load csv file, where *filterDF* filters the relevant columns such as players, teams, seasons, outcomes.

```
def aquireData(csv):
    # Opening the data
    df = pd.read_csv(csv)

    filteredDF = df[LINEUP_ATTRIBUTES]
```

2. Convert team abbreviations so that the teams will be named the same regardless of the year. This was done for the New Orlean Hornets (NOH) and was changed to New Orleans Pelicans (NOP) in 2013 [3]. This also applies for teams that have merged such as the Seattle Supersonics (SEA) relocated to Oklahoma to become the Oklahoma City Thunder in 2007 [4].

```
teams = set()
    for t in filteredDF['home_team'].unique():
        if t in ("NOK", "NOH"):
            teams.add("NOP")
        elif t == ("SEA"):
            teams.add("OKC")
        elif t == ("NJN"):
            teams.add("BRK")
        elif (t == "CHO"):
            teams.add("CHA")
        else:
            teams.add(t)
```

3. Encode each player with a unique player ID (*playertoIndex)* and each team with a unique team ID (*teamtoIndex)*. Add <UNK> to handle unknown players by adding an unknown player category to handle missing data.

```
players = sorted(players)
   if "<UNK>" not in players:
      players.append("<UNK>")
   numPlayers = len(players)
   # Creating a dictionary to map each player and team to an indexed value
   playerToIndex = {player: idx for idx, player in enumerate(players)}
   teamToIndex = dict(zip(teams, range(numTeams)))
   playerToIndex = dict(playerToIndex)
```

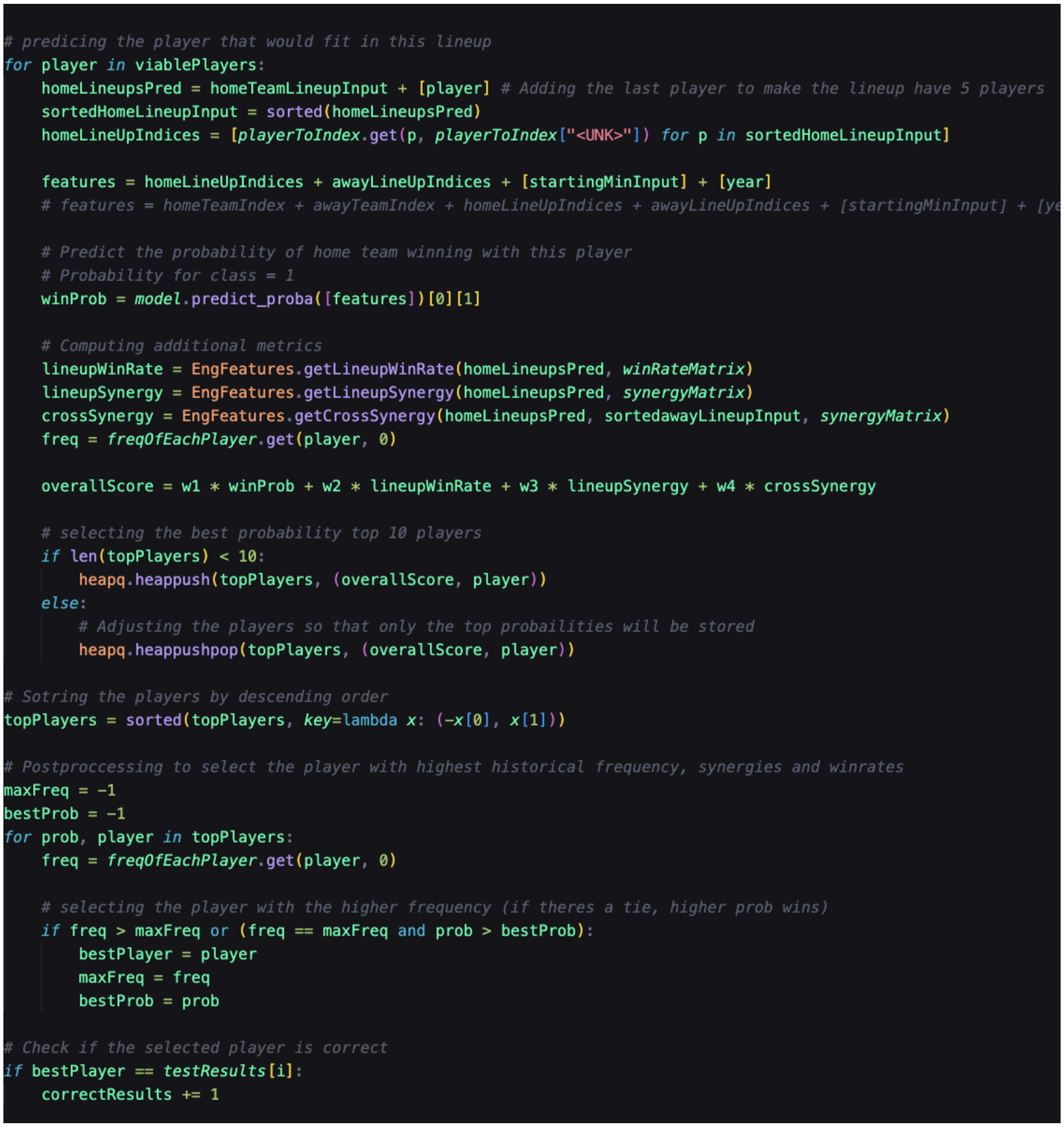
4. Normalize data by scaling numeric features for a better model performance.

```
         startingMin.append(row['starting_min'] / 47)
               year.append((row['season'] - 2007)/ (2015 - 2007))
               classifier = row['outcome']
               if(classifier == -1):
```

When predicting for the optimal fifth player in the home lineup, the 4 players would be given as an input to the algorithm and one player from a list of all players would be added. This would result in wasted computational time when predicting, as the players that are on the same team only need to be tested. The solution to this was to create a rosters of each team and then use a list of the roster to predict for the 5 home players.

```
# Making sure that the players that are being tested are in the correct home roster
and isnt any of the current players
        viablePlayers = [p for p in set(rosters[homeTeamInput]) if p not in
homeTeamLineupInput]

        # predicting the player that would fit in this lineup
        for player in viablePlayers:
            homeLineupsPred = homeTeamLineupInput + [player] # Adding the last player
to make the lineup have 5 players
            sortedHomeLineupInput = sorted(homeLineupsPred)
            homeLineUpIndices = [playerToIndex.get(p, playerToIndex["<UNK>"]) for p in
sortedHomeLineupInput]
```

In the prediction phase, post processing was done to select the most accurate player that would represent the fifth player to complete the lineup. On prediction, a score was given to each candidate player created from aggregating the average synergy between each pair of players in the home team, the average lineup win rate as well as the cross synergy between both teams. Once the 10 candidate players were selected, they were then further processed by selecting the player that had higher frequency of playing.



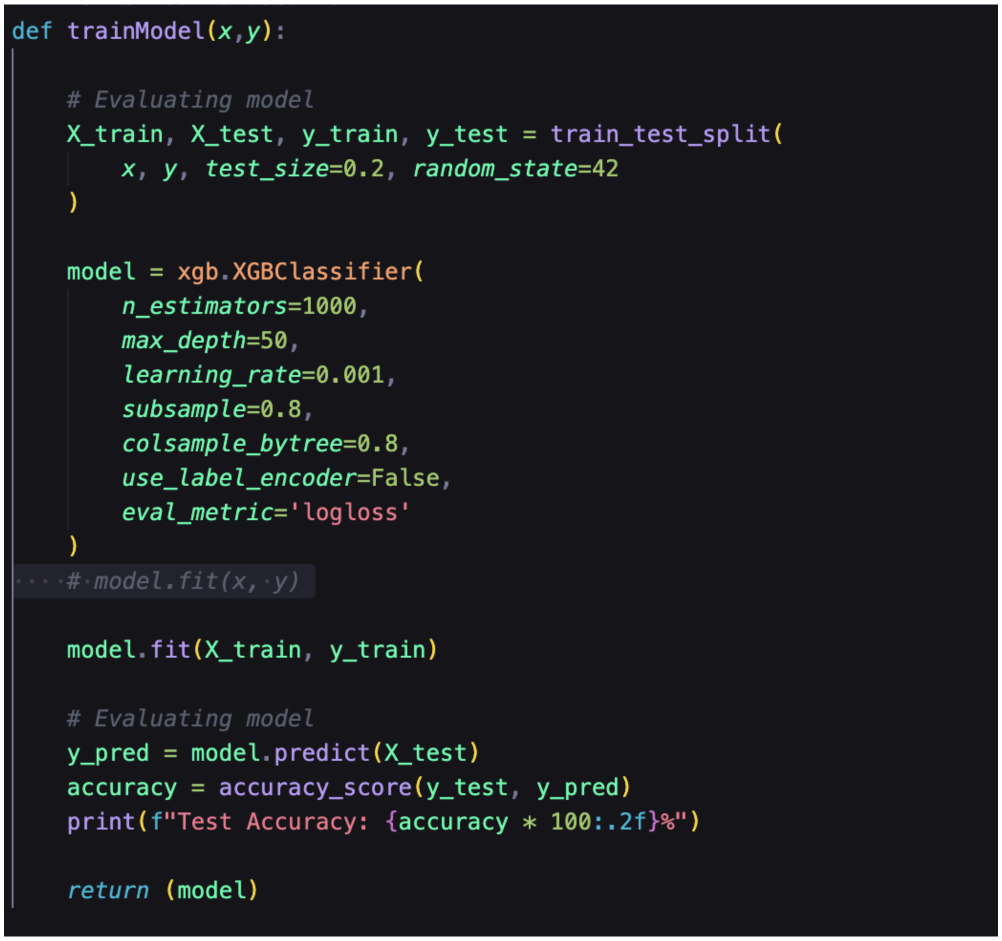
```python
# predicing the player that would fit in this lineup
for player in viablePlayers:
    homeLineupsPred = homeTeamLineupInput + [player] # Adding the last player to make the lineup have 5 players
    sortedHomeLineupInput = sorted(homeLineupsPred)
    homeLineUpIndices = [playerToIndex.get(p, playerToIndex["<UNK>"]) for p in sortedHomeLineupInput]

    features = homeLineUpIndices + awayLineUpIndices + [startingMinInput] + [year]
    # features = homeTeamIndex + awayTeamIndex + homeLineUpIndices + awayLineUpIndices + [startingMinInput] + [ye

    # Predict the probability of home team winning with this player
    # Probability for class = 1
    winProb = model.predict_proba([features])[0][1]

    # Computing additional metrics
    lineupWinRate = EngFeatures.getLineupWinRate(homeLineupsPred, winRateMatrix)
    lineupSynergy = EngFeatures.getLineupSynergy(homeLineupsPred, synergyMatrix)
    crossSynergy = EngFeatures.getCrossSynergy(homeLineupsPred, sortedawayLineupInput, synergyMatrix)
    freq = freqOfEachPlayer.get(player, 0)

    overallScore = w1 * winProb + w2 * lineupWinRate + w3 * lineupSynergy + w4 * crossSynergy

    # selecting the best probability top 10 players
    if len(topPlayers) < 10:
        heapq.heappush(topPlayers, (overallScore, player))
    else:
        # Adjusting the players so that only the top probailities will be stored
        heapq.heappushpop(topPlayers, (overallScore, player))

# Sotring the players by descending order
topPlayers = sorted(topPlayers, key=lambda x: (-x[0], x[1]))

# Postproccessing to select the player with highest historical frequency, synergies and winrates
maxFreq = -1
bestProb = -1
for prob, player in topPlayers:
    freq = freqOfEachPlayer.get(player, 0)

    # selecting the player with the higher frequency (if theres a tie, higher prob wins)
    if freq > maxFreq or (freq == maxFreq and prob > bestProb):
        bestPlayer = player
        maxFreq = freq
        bestProb = prob

# Check if the selected player is correct
if bestPlayer == testResults[i]:
    correctResults += 1
```

7. Evaluation of Model

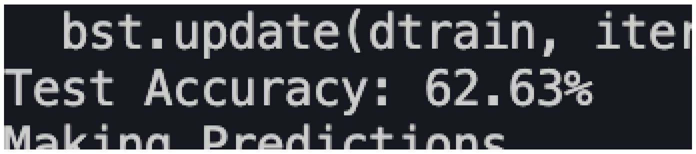
Using XGBoost combined with the feature selections gives this model strengths and weaknesses.

Strengths	Weaknesses
Uses XGBoost; considered to be the best	Risk of overfitting
Model can learn from previous lineups	Does not capture players’ skills
Handles multiple years of data	Does not capture coaches’ skills

The process to evaluate the accuracy of the model is shown in the figure below. The training data was split for testing the accuracy of the model.



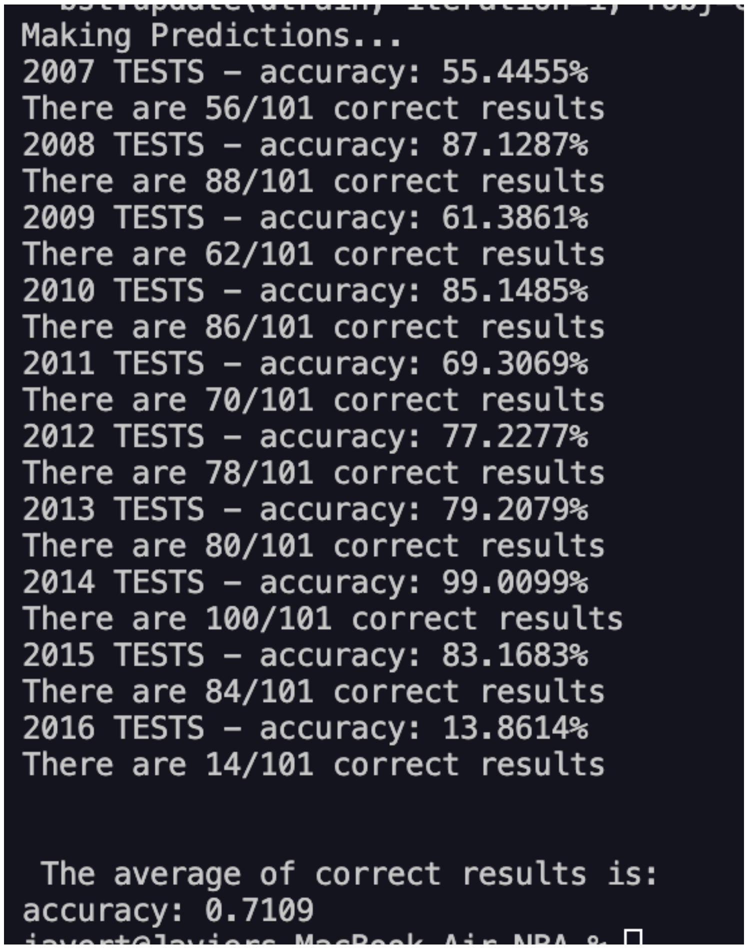
```
def trainModel(x,y):

    # Evaluating model
    X_train, X_test, y_train, y_test = train_test_split(
        x, y, test_size=0.2, random_state=42
    )

    model = xgb.XGBClassifier(
        n_estimators=1000,
        max_depth=50,
        learning_rate=0.001,
        subsample=0.8,
        colsample_bytree=0.8,
        use_label_encoder=False,
        eval_metric='logloss'
    )
    # model.fit(x, y)

    model.fit(X_train, y_train)

    # Evaluating model
    y_pred = model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    print(f"Test Accuracy: {accuracy * 100:.2f}%")

    return (model)
```



```
  bst.update(dtrain, iter
Test Accuracy: 62.63%
Making Predictions
```

The accuracy of 62.63% is the highest percent achieved when adjusting the n_estimators as well as the max_depth. The n_estimators represent the amount of different trees or models that have been trained and combined to generate this model. The max depth is the parameter set for the amount of decision the tree can make before reaching a leaf. This was set higher as there were a lot of complex patterns that the model had to learn between each player and the outcome.

8. Performance on Tests



```
Making Predictions...
2007 TESTS - accuracy: 55.4455%
There are 56/101 correct results
2008 TESTS - accuracy: 87.1287%
There are 88/101 correct results
2009 TESTS - accuracy: 61.3861%
There are 62/101 correct results
2010 TESTS - accuracy: 85.1485%
There are 86/101 correct results
2011 TESTS - accuracy: 69.3069%
There are 70/101 correct results
2012 TESTS - accuracy: 77.2277%
There are 78/101 correct results
2013 TESTS - accuracy: 79.2079%
There are 80/101 correct results
2014 TESTS - accuracy: 99.0099%
There are 100/101 correct results
2015 TESTS - accuracy: 83.1683%
There are 84/101 correct results
2016 TESTS - accuracy: 13.8614%
There are 14/101 correct results


 The average of correct results is:
accuracy: 0.7109
```

In the figures above, the accuracy of the tests for each year is printed out with a percentage value as well as the number of correct results compared to the total number of tests performed within the model. In the end, the average number of correct results for all tests performed is displayed, giving an insight on how accurate the model is. According to the results, it is evident that the accuracy for each year varies from the last. The accuracy of the model is not 100%, mainly due to the fact that there is not enough player information to input into the model.

References

[1] Nvidia, “What is XGBoost?,” *NVIDIA Data Science Glossary*, 2024. https://www.nvidia.com/en-us/glossary/xgboost/

[2] “Gradient Boosting, Decision Trees and XGBoost with CUDA | NVIDIA Technical Blog,” *NVIDIA Technical Blog*, Sep. 12, 2017. https://developer.nvidia.com/blog/gradient-boosting-decision-trees-xgboost-cuda (accessed Mar. 19, 2025).

[3] “New Orleans Hornets announce name change to Pelicans - ESPN,” *ESPN.com*, Jan. 25, 2013. https://www.espn.com/nba/story/_/id/8878315/new-orleans-hornets-announce-name-change-pelicans

[4] “Oklahoma City Thunder Franchise Index,” *Basketball-Reference.com*. https://www.basketball-reference.com/teams/OKC/