</ Data Bootcamp - Predicting NBA Success Across Time Horizons />

} />
[

Solomon Roy - sr7018@nyu.edu
Emiliano Eguez - ee2488@nyu.edu
Lucas Huang - ljh9448@nyu.edu

1011 011 01 1011001 10 11011 011 01 110110 110111 1101

</ Table of contents

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Abstract

This project investigates the determinants of NBA success across two distinct time horizons: long term success measured by next season win percentage, and short term success measured by the outcome of individual regular season games. Building directly on our midterm project, which analyzed season level team metrics and their correlation with future performance, the final project extends the analysis to a game level predictive framework. By restructuring the data to model matchups rather than teams, incorporating rolling statistics, momentum, and contextual factors such as rest, and evaluating multiple machine learning models, we demonstrate that long term team quality sets baseline expectations while short term context dominates single game outcomes. The results show clear methodological and analytical advancement beyond the midterm and provide a more realistic account of how NBA games are won.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1 0 1 1 1 1 0 1

</>

Introduction

01

} /> [

1011 011 01 1011001 10 11011 011 01 110110 110111 1101

</ Introduction

Success in the NBA can be defined in multiple ways. Over long horizons, success reflects a team's ability to sustain high performance across an entire season and into subsequent years. Over short horizons, success is far more volatile and context dependent, influenced by fatigue, injuries, recent form, and matchup specific factors. Understanding this distinction is critical for both analytics and decision making.

Our central motivation was to understand:

1. What makes teams good in the long run?
2. What makes teams win on a given night?
3. How these drivers differ, overlap, and interact

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Intro: Let's take a look back...

Our midterm project focused on the long term perspective, asking which team variables are most predictive of next season success.

We asked: Which team variables best predict next season's win percentage?

Using advanced NBA statistics from the 2021-22 season, we correlated them with 2022-23 win rates.

Our conclusion? Good teams perform well on advanced metrics, and good teams tend to stay good. Our analysis demonstrated that efficiency, defense, and possession control are strong indicators of sustained performance. However, it left unanswered the more difficult question of whether those same variables can explain individual game outcomes.

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

</ Intro: Looking forward

Our final project builds directly on this foundation.

Rather than discarding the midterm’s conclusions, it extends them by shifting the unit of analysis from teams to games and from correlation to prediction.

Central objective: to determine how long term team quality interacts with short term context to produce single game results. The final project extends the midterm in three fundamental ways:

1. Unit of analysis: from teams to games
2. Methodology: from correlation to supervised prediction
3. Perspective: from static season averages to dynamic, rolling context

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

</>

Data Sources and Scope

02

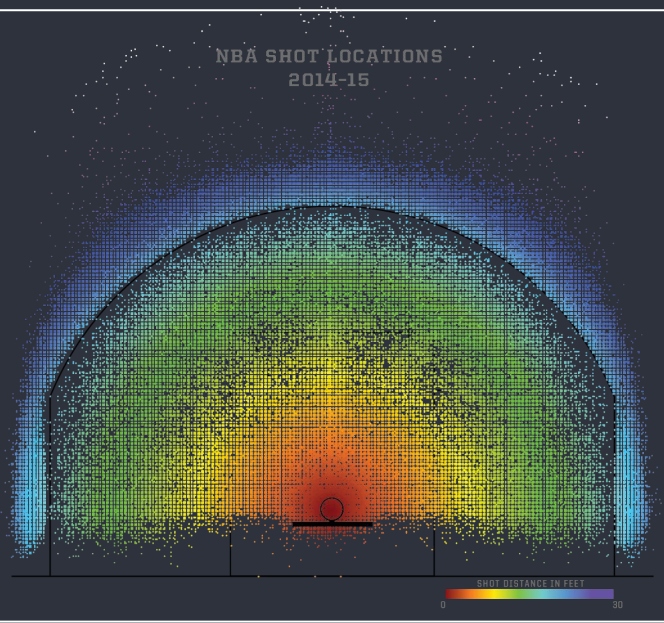
} /> [

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Data Sources and Scope

The final project uses NBA regular season game data from the 2015-2022 seasons. This time frame corresponds to the modern three point era, ensuring structural consistency in how the game is played. Only regular season games are included to avoid playoff specific dynamics.

Games are ordered chronologically to preserve temporal integrity. This ordering is essential for building rolling features and performing realistic train test splits.



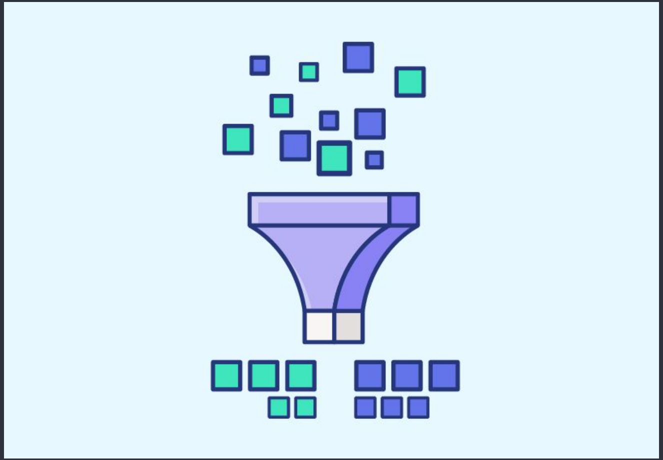
1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Data Sources and Scope

From a technical perspective, ouer project follows this pipeline:

1. Data acquisition from NBA sources
2. Cleaning and filtering raw game data
3. Feature engineering using rolling statistics
4. Matchup construction (home vs away differentials)
5. Model training and evaluation
6. Interpretation and validation

This structure mirrors a sports analytics workflow.



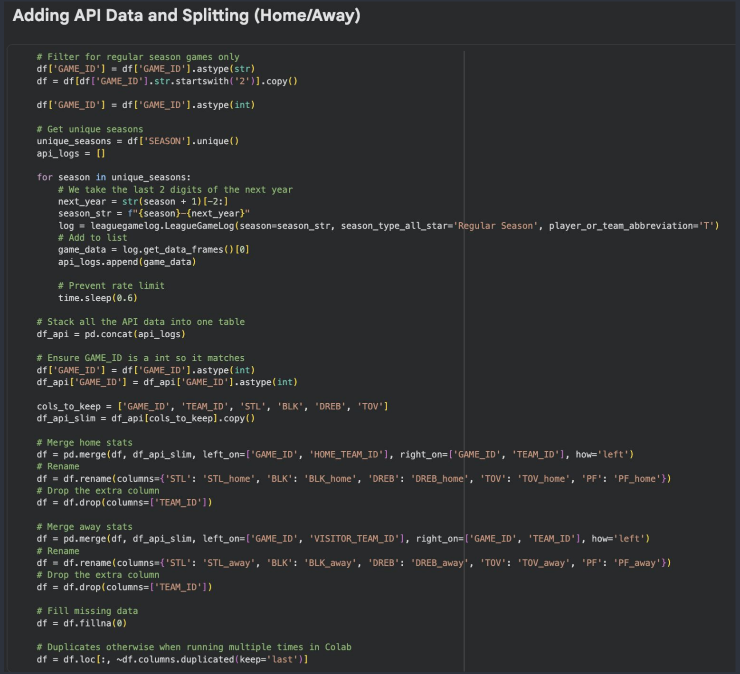
1011 011 01 1011001 10 11011 011 01 110110 110111 1101

</ Data Sources and Scope

Our final project draws from multiple NBA data sources, combined into a single modeling dataset:

1. NBA game level data (2015-2022)
   a. Regular season games only
   b. Game dates, teams, outcomes, and team box score stats
2. NBA API (nba_api Python package)
   a. Used to understand and validate team statistics
   b. Provides official NBA data endpoints for teams, games, and advanced metrics
3. Precompiled historical game datasets (CSV format)
   a. Used for efficiency and stability
   b. Ensures reproducibility and avoids API rate limitations

All data ultimately originates from official NBA statistics.



Adding API Data and Splitting (Home/Away)

```python
# Filter for regular season games only
df['GAME_ID'] = df['GAME_ID'].astype(str)
df = df[df['GAME_ID'].str.startswith('2')].copy()

df['GAME_ID'] = df['GAME_ID'].astype(int)

# Get unique seasons
unique_seasons = df['SEASON'].unique()
api_logs = []

for season in unique_seasons:
    # We take the last 2 digits of the next year
    next_year = str(season + 1)[-2:]
    season_str = f"{season}-{next_year}"
    log = leaguegamelog.LeagueGameLog(season=season_str, season_type_all_star='Regular Season', player_or_team_abbreviation='T')
    # Add to list
    game_data = log.get_data_frames()[0]
    api_logs.append(game_data)

    # Prevent rate limit
    time.sleep(0.6)

# Stack all the API data into one table
df_api = pd.concat(api_logs)

# Ensure GAME_ID is a int so it matches
df['GAME_ID'] = df['GAME_ID'].astype(int)
df_api['GAME_ID'] = df_api['GAME_ID'].astype(int)

cols_to_keep = ['GAME_ID', 'TEAM_ID', 'STL', 'BLK', 'DREB', 'TOV']
df_api_slim = df_api[cols_to_keep].copy()

# Merge home stats
df = pd.merge(df, df_api_slim, left_on=['GAME_ID', 'HOME_TEAM_ID'], right_on=['GAME_ID', 'TEAM_ID'], how='left')
# Rename
df = df.rename(columns={'STL': 'STL_home', 'BLK': 'BLK_home', 'DREB': 'DREB_home', 'TOV': 'TOV_home', 'PF': 'PF_home'})
# Drop the extra column
df = df.drop(columns=['TEAM_ID'])

# Merge away stats
df = pd.merge(df, df_api_slim, left_on=['GAME_ID', 'VISITOR_TEAM_ID'], right_on=['GAME_ID', 'TEAM_ID'], how='left')
# Rename
df = df.rename(columns={'STL': 'STL_away', 'BLK': 'BLK_away', 'DREB': 'DREB_away', 'TOV': 'TOV_away', 'PF': 'PF_away'})
# Drop the extra column
df = df.drop(columns=['TEAM_ID'])

# Fill missing data
df = df.fillna(0)

# Duplicates otherwise when running multiple times in Colab
df = df.loc[:, ~df.columns.duplicated(keep='last')]
```

1011 011 01 1011001 10 11011 011 01 110110 110111 1101

</>

Data Engineering and Feature Construction

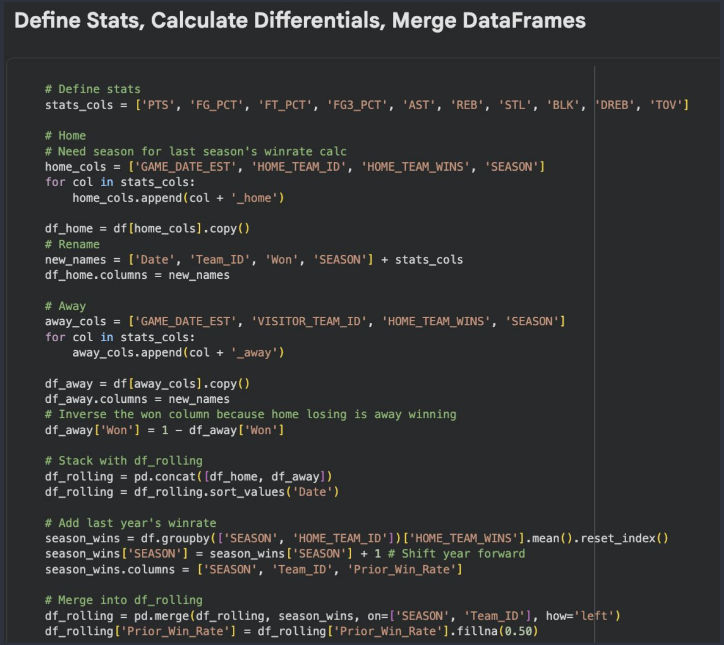
03

} /> [

1011 011 01 1011001 10 11011 011 01 110110 110111 1101

</ 3.1 From Games to Teams

Raw game data is reshaped so that each row represents a team in a game. This transformation enables the computation of time aware statistics for each team across the season.



Define Stats, Calculate Differentials, Merge DataFrames

```python
# Define stats
stats_cols = ['PTS', 'FG_PCT', 'FT_PCT', 'FG3_PCT', 'AST', 'REB', 'STL', 'BLK', 'DREB', 'TOV']

# Home
# Need season for last season's winrate calc
home_cols = ['GAME_DATE_EST', 'HOME_TEAM_ID', 'HOME_TEAM_WINS', 'SEASON']
for col in stats_cols:
    home_cols.append(col + '_home')

df_home = df[home_cols].copy()
# Rename
new_names = ['Date', 'Team_ID', 'Won', 'SEASON'] + stats_cols
df_home.columns = new_names

# Away
away_cols = ['GAME_DATE_EST', 'VISITOR_TEAM_ID', 'HOME_TEAM_WINS', 'SEASON']
for col in stats_cols:
    away_cols.append(col + '_away')

df_away = df[away_cols].copy()
df_away.columns = new_names
# Inverse the won column because home losing is away winning
df_away['Won'] = 1 - df_away['Won']

# Stack with df_rolling
df_rolling = pd.concat([df_home, df_away])
df_rolling = df_rolling.sort_values('Date')

# Add last year's winrate
season_wins = df.groupby(['SEASON', 'HOME_TEAM_ID'])['HOME_TEAM_WINS'].mean().reset_index()
season_wins['SEASON'] = season_wins['SEASON'] + 1 # Shift year forward
season_wins.columns = ['SEASON', 'Team_ID', 'Prior_Win_Rate']

# Merge into df_rolling
df_rolling = pd.merge(df_rolling, season_wins, on=['SEASON', 'Team_ID'], how='left')
df_rolling['Prior_Win_Rate'] = df_rolling['Prior_Win_Rate'].fillna(0.50)
```

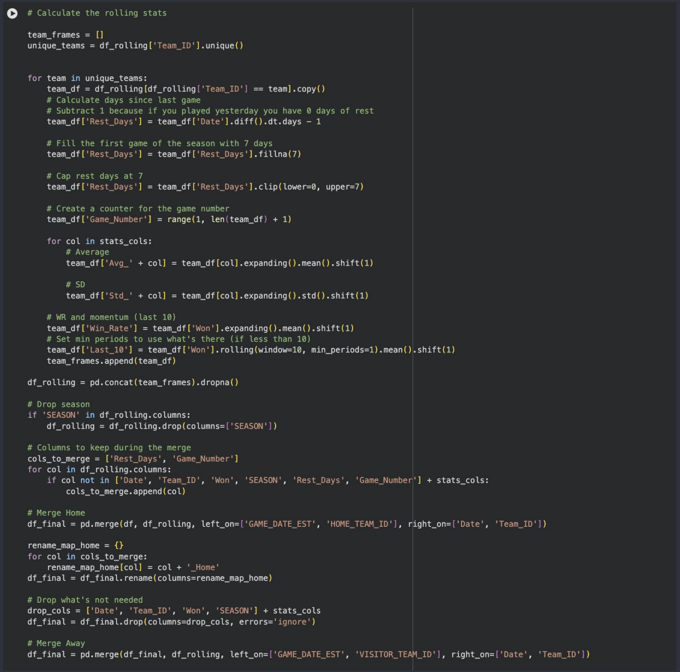
1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

</ 3.2 Rolling Performance Metrics

For each team, the following rolling features are computed:

- Expanding averages to capture current performance level
- Rolling standard deviations to capture consistency
- Rolling win rates over the last ten games to capture momentum

All rolling features are shifted forward by one game to prevent data leakage.

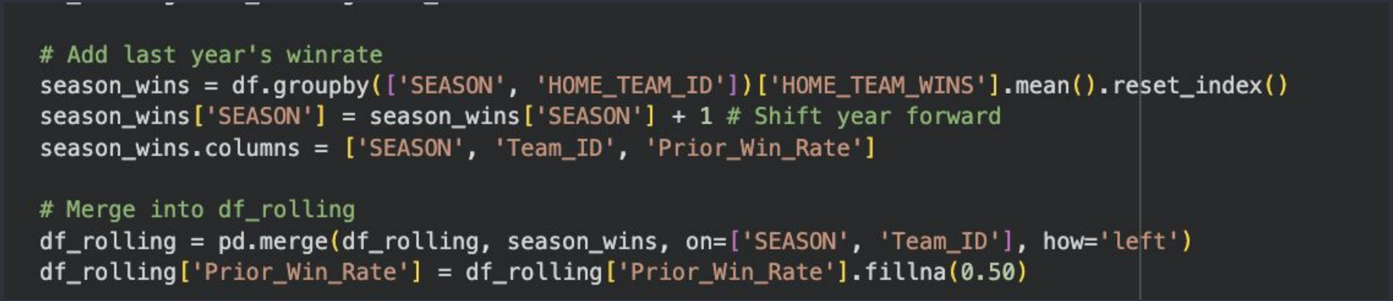


```python
# Calculate the rolling stats

team_frames = []
unique_teams = df_rolling['Team_ID'].unique()


for team in unique_teams:
    team_df = df_rolling[df_rolling['Team_ID'] == team].copy()
    # Calculate days since last game
    # Subtract 1 because if you played yesterday you have 0 days of rest
    team_df['Rest_Days'] = team_df['Date'].diff().dt.days - 1

    # Fill the first game of the season with 7 days
    team_df['Rest_Days'] = team_df['Rest_Days'].fillna(7)

    # Cap rest days at 7
    team_df['Rest_Days'] = team_df['Rest_Days'].clip(lower=0, upper=7)

    # Create a counter for the game number
    team_df['Game_Number'] = range(1, len(team_df) + 1)

    for col in stats_cols:
        # Average
        team_df['Avg_' + col] = team_df[col].expanding().mean().shift(1)

        # SD
        team_df['Std_' + col] = team_df[col].expanding().std().shift(1)

    # WR and momentum (last 10)
    team_df['Win_Rate'] = team_df['Won'].expanding().mean().shift(1)
    # Set min periods to use what's there (if less than 10)
    team_df['Last_10'] = team_df['Won'].rolling(window=10, min_periods=1).mean().shift(1)
    team_frames.append(team_df)

df_rolling = pd.concat(team_frames).dropna()

# Drop season
if 'SEASON' in df_rolling.columns:
    df_rolling = df_rolling.drop(columns=['SEASON'])

# Columns to keep during the merge
cols_to_merge = ['Rest_Days', 'Game_Number']
for col in df_rolling.columns:
    if col not in ['Date', 'Team_ID', 'Won', 'SEASON', 'Rest_Days', 'Game_Number'] + stats_cols:
        cols_to_merge.append(col)

# Merge Home
df_final = pd.merge(df, df_rolling, left_on=['GAME_DATE_EST', 'HOME_TEAM_ID'], right_on=['Date', 'Team_ID'])

rename_map_home = {}
for col in cols_to_merge:
    rename_map_home[col] = col + '_Home'
df_final = df_final.rename(columns=rename_map_home)

# Drop what's not needed
drop_cols = ['Date', 'Team_ID', 'Won', 'SEASON'] + stats_cols
df_final = df_final.drop(columns=drop_cols, errors='ignore')

# Merge Away
df_final = pd.merge(df_final, df_rolling, left_on=['GAME_DATE_EST', 'VISITOR_TEAM_ID'], right_on=['Date', 'Team_ID'])
```

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

</ 3.4 Long Term Signal Integration

Prior season win percentage is included as a feature, directly importing the midterm’s long term perspective into the short term model. This variable serves as a proxy for organizational stability and sustained team quality.



```
# Add last year's winrate
season_wins = df.groupby(['SEASON', 'HOME_TEAM_ID'])['HOME_TEAM_WINS'].mean().reset_index()
season_wins['SEASON'] = season_wins['SEASON'] + 1 # Shift year forward
season_wins.columns = ['SEASON', 'Team_ID', 'Prior_Win_Rate']

# Merge into df_rolling
df_rolling = pd.merge(df_rolling, season_wins, on=['SEASON', 'Team_ID'], how='left')
df_rolling['Prior_Win_Rate'] = df_rolling['Prior_Win_Rate'].fillna(0.50)
```

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

</>

Matchup Construction

04

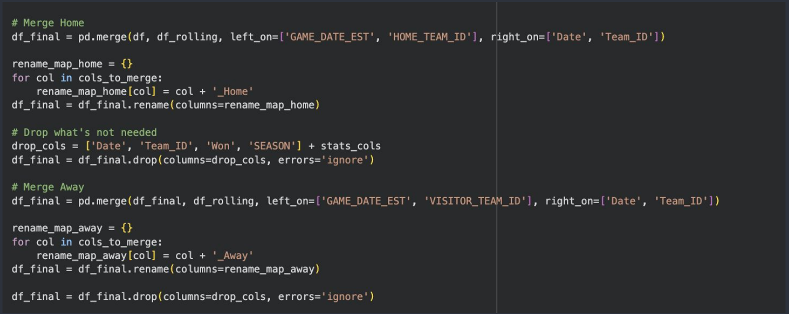
} /> [

1011 011 01 1011001 10 11011 011 01 110110 110111 1101

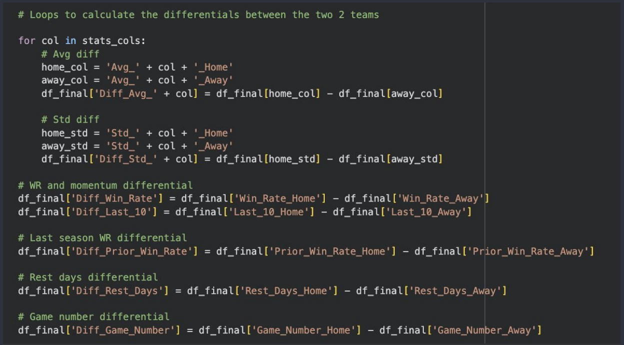
</ Matchup Construction

Once team level rolling features are computed, the dataset is reconstructed at the game level. Each game is represented by home–away differentials for all variables. This approach models relative advantage rather than absolute strength and reflects how real basketball decisions are made.

1. Reconstruct back to game level by merging team level rolling features onto the original game table (once for home, once for away)
2. Create home away differentials for every variable
3. Build the final model table using only the differentials (+ a couple identifiers)



```python
# Merge Home
df_final = pd.merge(df, df_rolling, left_on=['GAME_DATE_EST', 'HOME_TEAM_ID'], right_on=['Date', 'Team_ID'])

rename_map_home = {}
for col in cols_to_merge:
    rename_map_home[col] = col + '_Home'
df_final = df_final.rename(columns=rename_map_home)

# Drop what's not needed
drop_cols = ['Date', 'Team_ID', 'Won', 'SEASON'] + stats_cols
df_final = df_final.drop(columns=drop_cols, errors='ignore')

# Merge Away
df_final = pd.merge(df_final, df_rolling, left_on=['GAME_DATE_EST', 'VISITOR_TEAM_ID'], right_on=['Date', 'Team_ID'])

rename_map_away = {}
for col in cols_to_merge:
    rename_map_away[col] = col + '_Away'
df_final = df_final.rename(columns=rename_map_away)

df_final = df_final.drop(columns=drop_cols, errors='ignore')
```



```python
# Loops to calculate the differentials between the two 2 teams

for col in stats_cols:
    # Avg diff
    home_col = 'Avg_' + col + '_Home'
    away_col = 'Avg_' + col + '_Away'
    df_final['Diff_Avg_' + col] = df_final[home_col] - df_final[away_col]

    # Std diff
    home_std = 'Std_' + col + '_Home'
    away_std = 'Std_' + col + '_Away'
    df_final['Diff_Std_' + col] = df_final[home_std] - df_final[away_std]

# WR and momentum differential
df_final['Diff_Win_Rate'] = df_final['Win_Rate_Home'] - df_final['Win_Rate_Away']
df_final['Diff_Last_10'] = df_final['Last_10_Home'] - df_final['Last_10_Away']

# Last season WR differential
df_final['Diff_Prior_Win_Rate'] = df_final['Prior_Win_Rate_Home'] - df_final['Prior_Win_Rate_Away']

# Rest days differential
df_final['Diff_Rest_Days'] = df_final['Rest_Days_Home'] - df_final['Rest_Days_Away']

# Game number differential
df_final['Diff_Game_Number'] = df_final['Game_Number_Home'] - df_final['Game_Number_Away']
```

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

</>

Modeling Framework

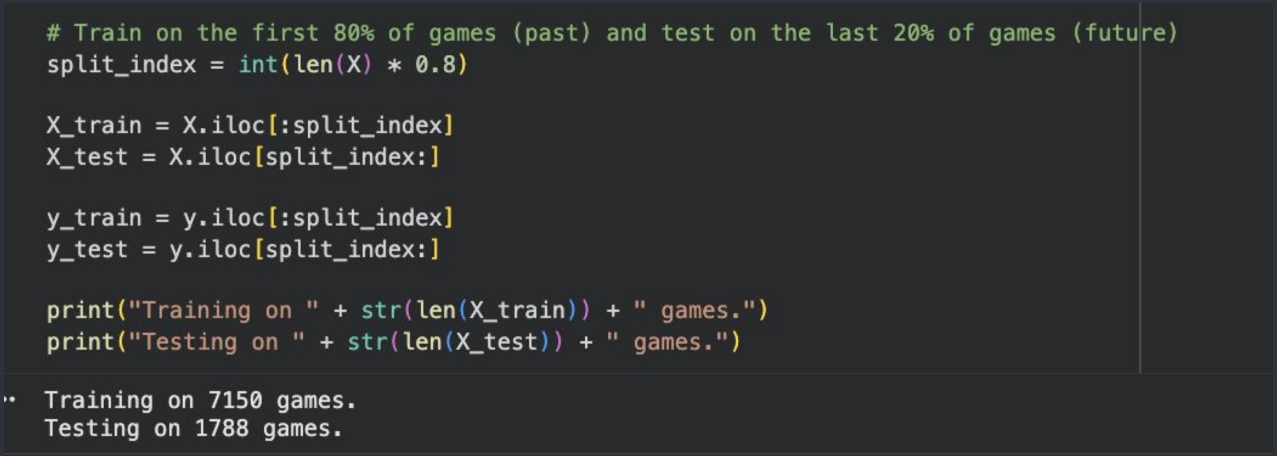
05

}/>[

1011 011 01 1011001 10 11011 011 01 110110 110111 1101

</// 5.1 Train–Test Split

Games are split chronologically, with the first 80% used for training and the final 20% used for testing. This simulates real world forecasting and avoids optimistic bias.



```
# Train on the first 80% of games (past) and test on the last 20% of games (future)
split_index = int(len(X) * 0.8)

X_train = X.iloc[:split_index]
X_test = X.iloc[split_index:]

y_train = y.iloc[:split_index]
y_test = y.iloc[split_index:]

print("Training on " + str(len(X_train)) + " games.")
print("Testing on " + str(len(X_test)) + " games.")
```

```
Training on 7150 games.
Testing on 1788 games.
```

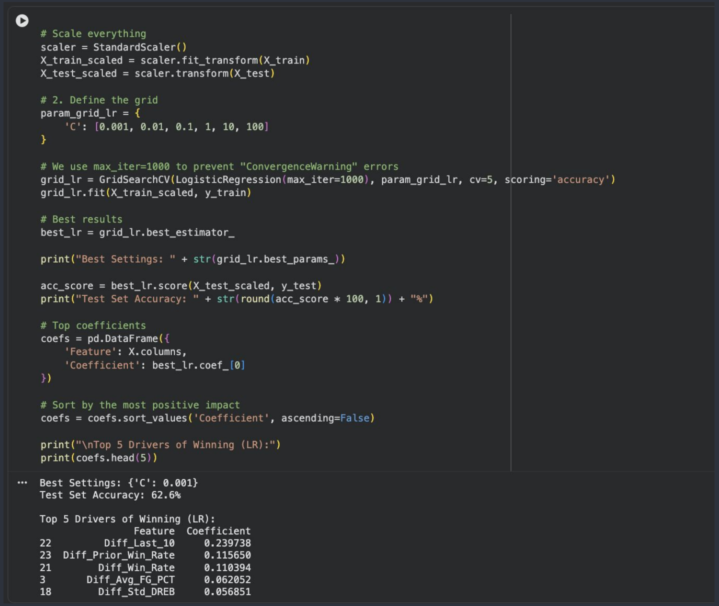
1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

</ 5.2 Models Employed

Three models are used:

1. Logistic Regression for interpretability
2. K Nearest Neighbors as a similarity based benchmark
3. XGBoost to capture non linear interactions

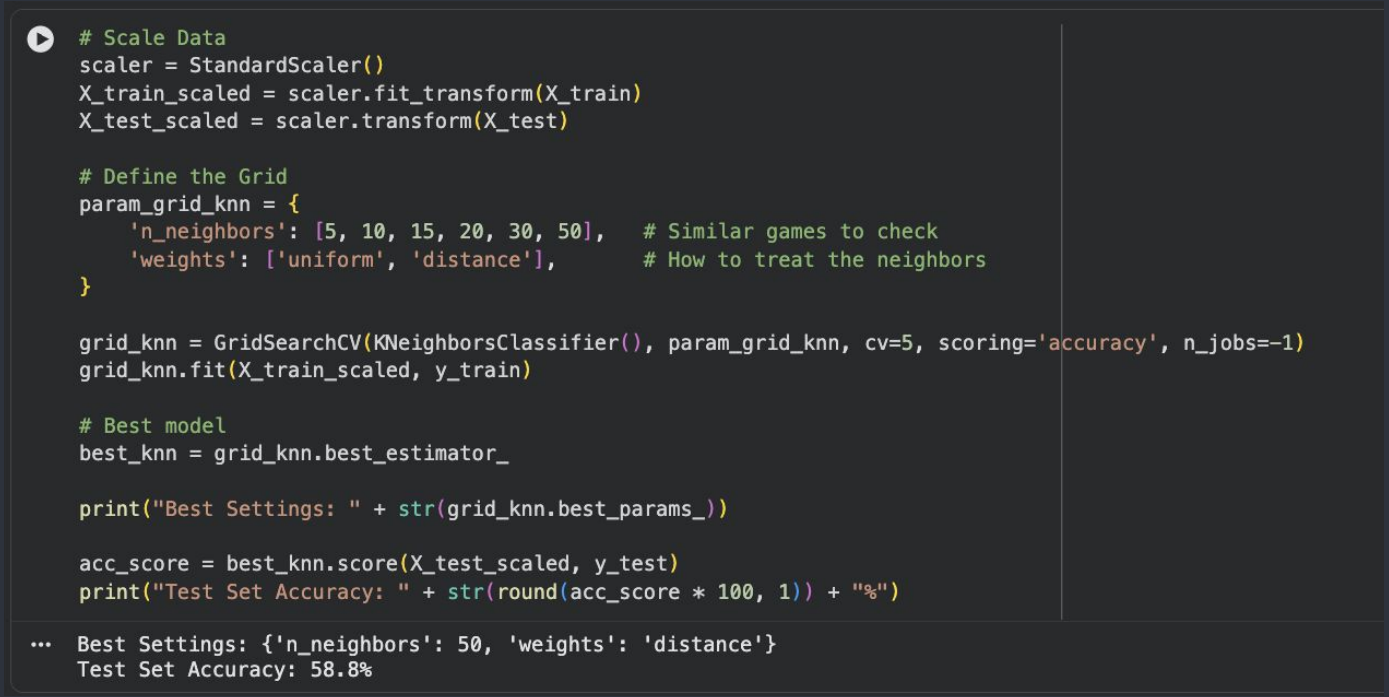
A baseline model that always predicts the home team is used for comparison.



```python
# Scale everything
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# 2. Define the grid
param_grid_lr = {
    'C': [0.001, 0.01, 0.1, 1, 10, 100]
}

# We use max_iter=1000 to prevent "ConvergenceWarning" errors
grid_lr = GridSearchCV(LogisticRegression(max_iter=1000), param_grid_lr, cv=5, scoring='accuracy')
grid_lr.fit(X_train_scaled, y_train)

# Best results
best_lr = grid_lr.best_estimator_

print("Best Settings: " + str(grid_lr.best_params_))

acc_score = best_lr.score(X_test_scaled, y_test)
print("Test Set Accuracy: " + str(round(acc_score * 100, 1)) + "%")

# Top coefficients
coefs = pd.DataFrame({
    'Feature': X.columns,
    'Coefficient': best_lr.coef_[0]
})

# Sort by the most positive impact
coefs = coefs.sort_values('Coefficient', ascending=False)

print("\nTop 5 Drivers of Winning (LR):")
print(coefs.head(5))
```

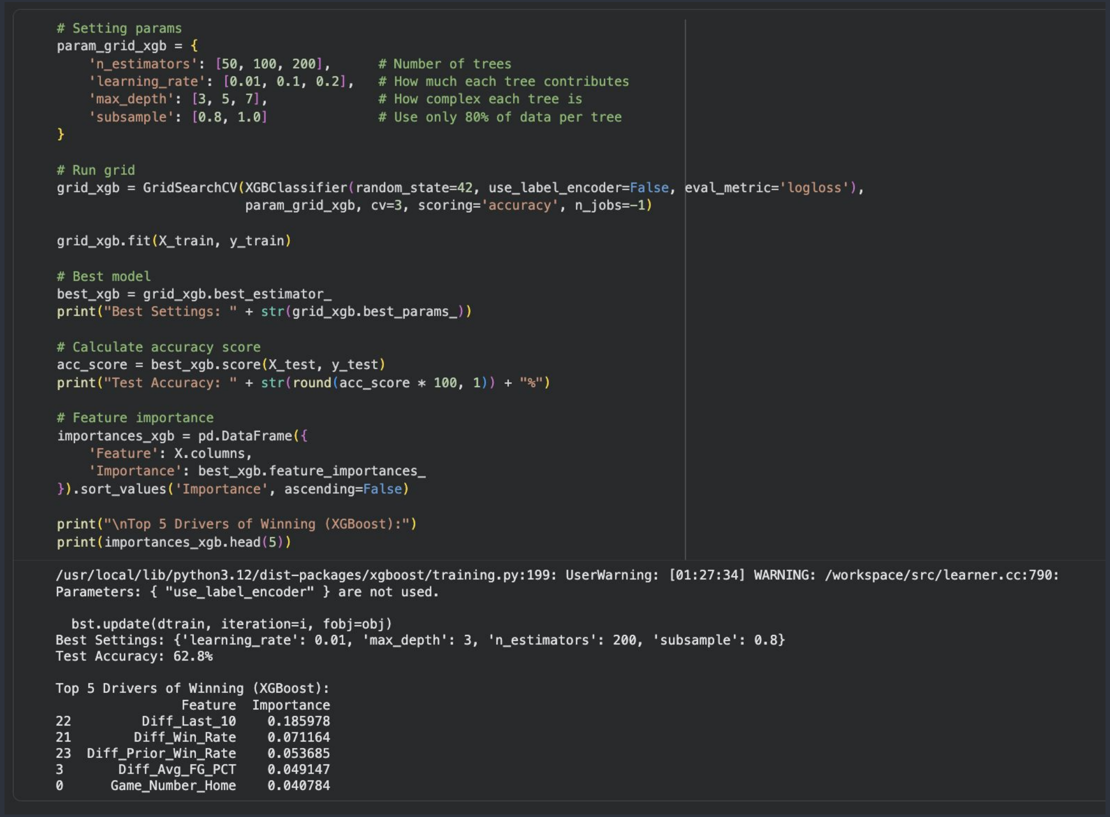
```
Best Settings: {'C': 0.001}
Test Set Accuracy: 62.6%

Top 5 Drivers of Winning (LR):
               Feature  Coefficient
22         Diff_Last_10     0.239738
23  Diff_Prior_Win_Rate     0.115650
21        Diff_Win_Rate     0.110394
3       Diff_Avg_FG_PCT     0.062052
18        Diff_Std_DREB     0.056851
```

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1



```
# Scale Data
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Define the Grid
param_grid_knn = {
    'n_neighbors': [5, 10, 15, 20, 30, 50],   # Similar games to check
    'weights': ['uniform', 'distance'],       # How to treat the neighbors
}

grid_knn = GridSearchCV(KNeighborsClassifier(), param_grid_knn, cv=5, scoring='accuracy', n_jobs=-1)
grid_knn.fit(X_train_scaled, y_train)

# Best model
best_knn = grid_knn.best_estimator_

print("Best Settings: " + str(grid_knn.best_params_))

acc_score = best_knn.score(X_test_scaled, y_test)
print("Test Set Accuracy: " + str(round(acc_score * 100, 1)) + "%")
```

```
Best Settings: {'n_neighbors': 50, 'weights': 'distance'}
Test Set Accuracy: 58.8%
```



```python
# Setting params
param_grid_xgb = {
    'n_estimators': [50, 100, 200],      # Number of trees
    'learning_rate': [0.01, 0.1, 0.2],   # How much each tree contributes
    'max_depth': [3, 5, 7],              # How complex each tree is
    'subsample': [0.8, 1.0]              # Use only 80% of data per tree
}

# Run grid
grid_xgb = GridSearchCV(XGBClassifier(random_state=42, use_label_encoder=False, eval_metric='logloss'),
                        param_grid_xgb, cv=3, scoring='accuracy', n_jobs=-1)

grid_xgb.fit(X_train, y_train)

# Best model
best_xgb = grid_xgb.best_estimator_
print("Best Settings: " + str(grid_xgb.best_params_))

# Calculate accuracy score
acc_score = best_xgb.score(X_test, y_test)
print("Test Accuracy: " + str(round(acc_score * 100, 1)) + "%")

# Feature importance
importances_xgb = pd.DataFrame({
    'Feature': X.columns,
    'Importance': best_xgb.feature_importances_
}).sort_values('Importance', ascending=False)

print("\nTop 5 Drivers of Winning (XGBoost):")
print(importances_xgb.head(5))
```

```
/usr/local/lib/python3.12/dist-packages/xgboost/training.py:199: UserWarning: [01:27:34] WARNING: /workspace/src/learner.cc:790:
Parameters: { "use_label_encoder" } are not used.

  bst.update(dtrain, iteration=i, fobj=obj)
Best Settings: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 200, 'subsample': 0.8}
Test Accuracy: 62.8%

Top 5 Drivers of Winning (XGBoost):
                Feature  Importance
22         Diff_Last_10    0.185978
21        Diff_Win_Rate    0.071164
23  Diff_Prior_Win_Rate    0.053685
3       Diff_Avg_FG_PCT    0.049147
0      Game_Number_Home    0.040784
```

1011 011 01 1011001 10 11011 011 01 110110 110111 1101

</>

Results

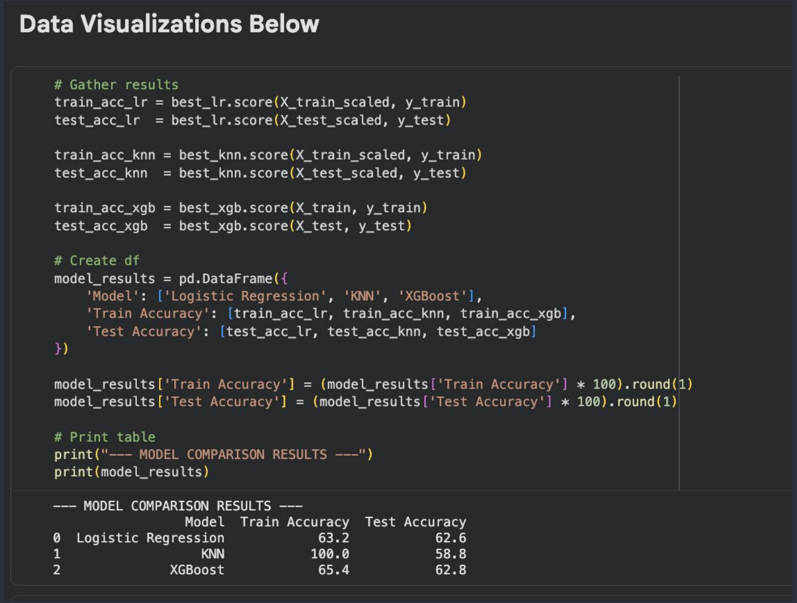
06

}/>[

1011 011 01 1011001 10 11011 011 01 110110 110111 1101

</ 6.1 Model Performance

Both Logistic Regression and XGBoost outperform the home team baseline, achieving test accuracy in the low to mid 60% range. KNN overfits and performs poorly out of sample, illustrating the challenges of similarity based methods in high dimensional settings.



Data Visualizations Below

```
# Gather results
train_acc_lr = best_lr.score(X_train_scaled, y_train)
test_acc_lr  = best_lr.score(X_test_scaled, y_test)

train_acc_knn = best_knn.score(X_train_scaled, y_train)
test_acc_knn  = best_knn.score(X_test_scaled, y_test)

train_acc_xgb = best_xgb.score(X_train, y_train)
test_acc_xgb  = best_xgb.score(X_test, y_test)

# Create df
model_results = pd.DataFrame({
    'Model': ['Logistic Regression', 'KNN', 'XGBoost'],
    'Train Accuracy': [train_acc_lr, train_acc_knn, train_acc_xgb],
    'Test Accuracy': [test_acc_lr, test_acc_knn, test_acc_xgb]
})

model_results['Train Accuracy'] = (model_results['Train Accuracy'] * 100).round(1)
model_results['Test Accuracy'] = (model_results['Test Accuracy'] * 100).round(1)

# Print table
print("--- MODEL COMPARISON RESULTS ---")
print(model_results)
```

```
--- MODEL COMPARISON RESULTS ---
                 Model  Train Accuracy  Test Accuracy
0  Logistic Regression            63.2           62.6
1                  KNN           100.0           58.8
2              XGBoost            65.4           62.8
```

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

</>

Supplementary Analysis and Interpretations

07

}/>[

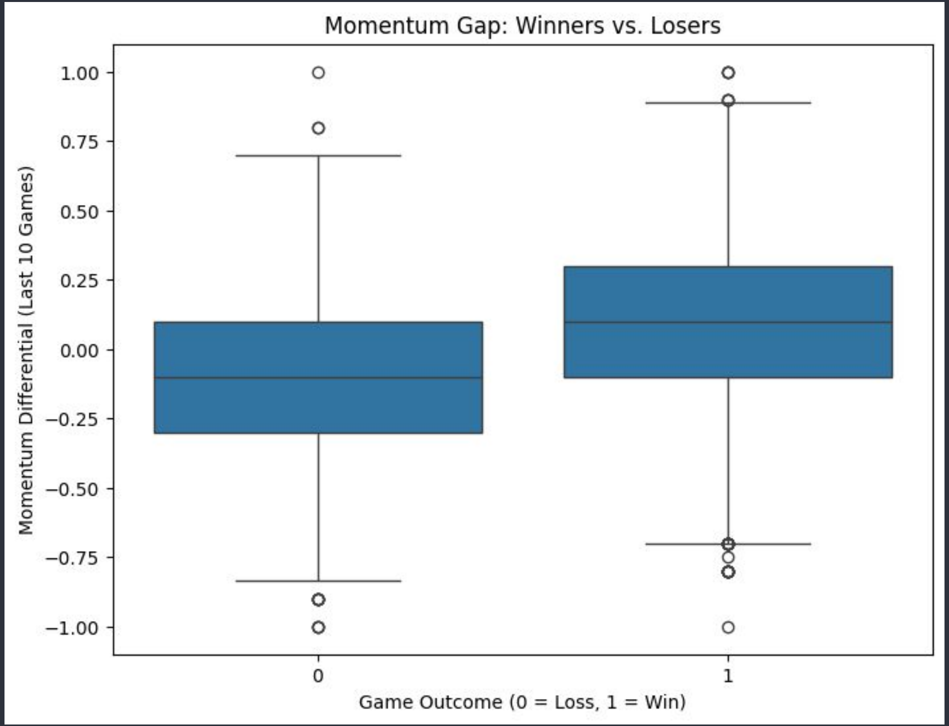
1011 011 01 1011001 10 11011 011 01 110110 110111 1101

</ 7.1 The Role of Momentum [Last 10 Games Win rate Differential]

Across both the Logistic Regression and XGBoost models, the Last 10 Games Winrate Differential consistently emerged as the most important predictor by a wide margin. This finding aligns well with basketball intuition: teams that are playing well recently are more likely to continue winning in the short term. The momentum distribution plot further supports this result, showing that winning teams typically enter games with higher recent win rates than losing teams.

From a practical standpoint, momentum acts as a summary variable for several factors that are difficult to measure directly, such as injuries, lineup stability, chemistry, travel fatigue, and in season adjustments. Rather than capturing these elements individually, the last ten games provide a compact signal of a team's current state. This helps explain why momentum dominates shor term prediction even though it plays a smaller role in long term success models.

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1



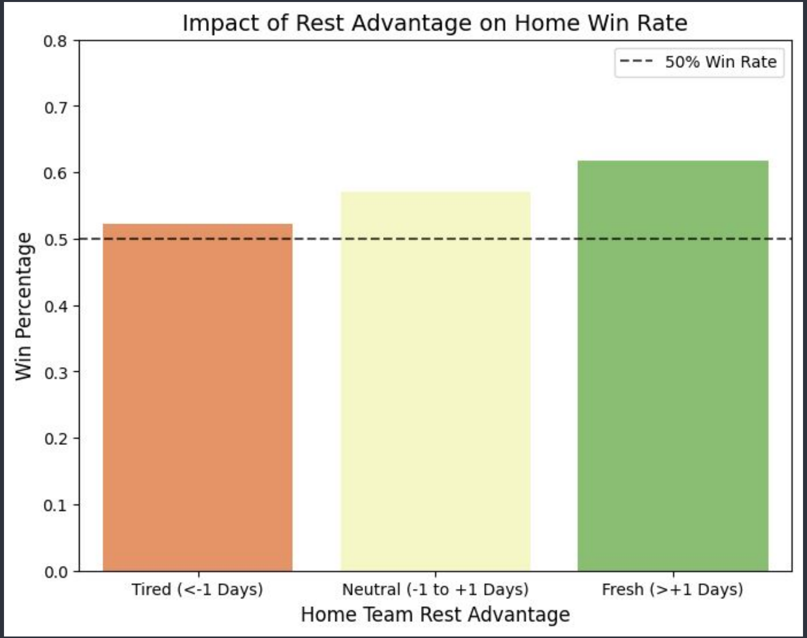
1011 011 01 1011001 10 11011 011 01 110110 110111 1101

</ 8.1 Limitations

Although Rest Days Differential was not among the top five coefficients in either model, our analysis shows that it still has a meaningful effect on outcomes. Teams with a rest advantage win at a noticeably higher rate than teams playing on short rest, with fresh teams winning close to 10% more often than tired ones.

This result makes intuitive sense. NBA games are physically demanding, and back to back games or travel heavy stretches reduce performance, especially in close matchups. Rest tends to matter most when teams are otherwise evenly matched, which helps explain why it does not dominate overall model importance but remains an important contextual factor when comparing similar teams.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1



1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

Limitations

08

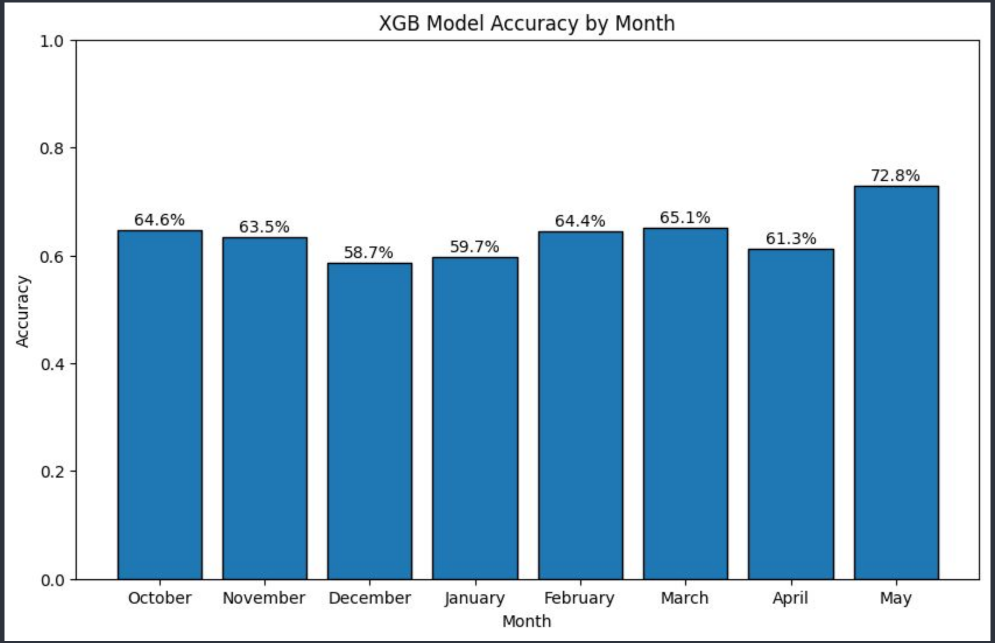
1011 011 01 1011001 10 11011 011 01 110110 110111 1101

</8.1 Roster Changes and Injuries

One limitation of our short term model is that it does not explicitly account for injuries. This likely explains why model accuracy drops in December and January, when injuries accumulate and rotations are less stable. While major injuries may indirectly affect momentum metrics, the model cannot distinguish between short term variance and meaningful roster changes.

Accuracy improves starting in February, which coincides with the NBA trade deadline. At this point, rosters stabilize, long term injuries are reflected in performance data, and teams have clearer competitive goals. As a result, the model performs better later in the season.

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1


XGB Model Accuracy by Month
Accuracy
Month
1.0
0.8
0.6
0.4
0.2
0.0
October 64.6%
November 63.5%
December 58.7%
January 59.7%
February 64.4%
March 65.1%
April 61.3%
May 72.8%

</8.2 Culture and Coaching

Culture and coaching are widely recognized as important but are difficult to quantify. We attempt to capture some of this through Last Season’s Winrate Differential, which reflects organizational stability and program strength. Teams with strong coaching and culture tend to perform consistently unless they are rebuilding. However, this proxy cannot fully capture changes in leadership, strategy, or player development.

</8.3 Data Constraints

Professional betting models outperform ours because they incorporate far more information, including injuries, travel, player matchups, and real-time news. Even with more data, NBA outcomes contain unavoidable randomness. Most professional models plateau around 65–70% accuracy, suggesting that some uncertainty cannot be eliminated.

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

</>

Conclusion

09

}/>[

1011 011 01 1011001 10 11011 011 01 110110 110111 1101

</ Conclusion

Our results show that the most predictive variables are the most holistic ones. Over long horizons, broad team quality measures such as win percentage, net rating, and plus minus are the strongest indicators of future success. In the short term, state variables like momentum dominate because they capture how a team is functioning right now.

Taken together, the midterm and final projects demonstrate that NBA success operates on multiple time scales. Long term team quality sets expectations, while short term context especially recent performance and rest determines individual game outcomes. Ultimately, the variables that best summarize overall performance are the ones that matter most for prediction.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1