BNA Report

Basketball DNA (BNA) Project Report

This is an English translation and restructured version of your BNA (Basketball DNA) project report. The code blocks and tables have been formatted using Markdown for better readability.

Overview

The **BNA (Basketball DNA)** project aims to predict the most suitable basketball position (**PG, SG, SF, PF, C**) for players using their game statistics and physical information, thereby supporting their career development. This project is intended to maximize the potential of current players and provide objective data for coaches and scouts regarding player selection and development.

---

1. Project Goals

- **Improve Player Position Prediction Accuracy:** Analyze diverse player data to enhance the accuracy of predicting each position.
- **Provide Personalized Career Guidelines:** Based on the predicted position, suggest necessary training directions, strategies to reinforce strengths and compensate for weaknesses to aid individual growth.
- **Offer Objective Player Evaluation Data:** Provide coaches and scouts with supporting evidence needed to objectively evaluate player potential and assign them to the optimal position.

---

2. Key Features and Contents

Data Collection and Preprocessing

- **Game Information:** Collect various in-game statistical data, including points (PTS), rebounds (REB), assists (AST), steals (STL), blocks (BLK), turnovers (TOV), field goal percentage (FG%), 3-point percentage (3P%), etc.

- **Physical Information:** Collect data on players' physical characteristics, such as height and weight.

The collected data undergoes preprocessing steps like handling missing values and normalization to be suitable for machine learning model training.

Machine Learning Model Development

- A **Classification** model is developed based on the collected data to predict one of the five positions: PG, SG, SF, PF, C.
- Various machine learning algorithms (e.g., Logistic Regression, Support Vector Machine, Random Forest, Neural Networks) are compared and analyzed to select the model with the best predictive performance.

Prediction Results and Analysis Report Provision

- Provide the player's optimal position predicted by the model, along with the prediction confidence for that position.
- Offer a detailed analysis report on the prediction results (e.g., the influence of specific stats on position prediction, key abilities required for the predicted position) to provide insights to players and coaching staff.

---

3. Expected Effects

- **Promote Individual Player Growth:** Players can establish efficient training plans by receiving customized information needed to maximize their strengths and improve their weaknesses.
- **Contribute to Team Strength Enhancement:** Coaches can accurately identify players' potential, assign them to optimal positions, and formulate player selection and development strategies aligned with team tactics.
- **Advance the Field of Basketball Data Analysis:** Raise awareness of the importance of sports data analysis and establish a foundation for more in-depth basketball data analysis research.

Data Collection

The original project plan was to use data from the **NBA API**, but this required a complex procedure to construct the dataset:

1. Load season performance data for players who participated in a specific season.
2. Add age, height, and weight data to the row for each player.

   a. First, retrieve the player's age, height, and weight using the player's name (2nd API call occurs).

   b. Find the player's row and insert the data.

This process would be repeated for all players.

The report includes the following Python code snippet using the `nba_api`:

```
import pandas as pd
from nba_api.stats.endpoints import LeagueDashPlayerStats, CommonPlay
erInfo
import time
import threading
from queue import Queue
import json
import os

CACHE_DIR = 'api_cache'
os.makedirs(CACHE_DIR, exist_ok=True)

def cache_file_path(player_id):
    return os.path.join(CACHE_DIR, f'player_{player_id}.json')

def get_cached_player_info(player_id):
    filepath = cache_file_path(player_id)
    if os.path.exists(filepath):
        with open(filepath, 'r') as f:
            return json.load(f)
    return None

def cache_player_info(player_id, data):
    filepath = cache_file_path(player_id)
    with open(filepath, 'w') as f:
        json.dump(data, f)
```

```
def get_player_physical_data(player_id, season, player_data_queue):
    cached_info = get_cached_player_info(player_id)
    if cached_info and 'CommonPlayerInfo' in cached_info and len(cached_i
nfo['CommonPlayerInfo']) > 0:
        height = cached_info['CommonPlayerInfo'][0]['HEIGHT']
        weight = cached_info['CommonPlayerInfo'][0]['WEIGHT']
        position = cached_info['CommonPlayerInfo'][0]['POSITION']
        player_data_queue.put({'PLAYER_ID': player_id, 'HEIGHT': height, 'WEI
GHT': weight, 'POSITION': position})
        print(f"Fetched physical data for player {player_id} from cache")
        return
    try:
        info = CommonPlayerInfo(player_id=player_id).get_normalized_dict()
        if 'CommonPlayerInfo' in info and len(info['CommonPlayerInfo']) > 0:
            height = info['CommonPlayerInfo'][0]['HEIGHT']
            weight = info['CommonPlayerInfo'][0]['WEIGHT']
            position = info['CommonPlayerInfo'][0]['POSITION']
            player_data_queue.put({'PLAYER_ID': player_id, 'HEIGHT': height,
'WEIGHT': weight, 'POSITION': position})
            cache_player_info(player_id, info) # Caching upon successful API ca
ll
            print(f"Fetched physical data for player {player_id} from API")
        else:
            print(f"No physical data found for player {player_id} in season {seas
on}")
        time.sleep(1.0)
    except Exception as e:
        print(f"Error fetching physical data for player {player_id} in season {se
ason}: {e}")

# Remaining code is the same (threading, etc.)
# ...

def get_all_players_stats_with_physical_data_threaded():
    all_seasons_data = []
    for year in range(2000, 2025):
        season = f"{year}-{str(year+1)[-2:]}"
        print(f"Fetching data for season {season}...")
```

```
        try:
            time.sleep(2.0)
            stats = LeagueDashPlayerStats(season=season).get_data_frames()
[0]
            player_data = []
            threads = []
            player_data_queue = Queue()
            max_threads = 5
            for _, row in stats.iterrows():
                player_id = row['PLAYER_ID']
                thread = threading.Thread(target=get_player_physical_data, args
=(player_id, season, player_data_queue))
                threads.append(thread)
                thread.start()
                if len(threads) >= max_threads:
                    for thread in threads:
                        thread.join()
                    threads = []
            for thread in threads:
                thread.join()

            while not player_data_queue.empty():
                physical_data = player_data_queue.get()
                player_row = stats[stats['PLAYER_ID'] == physical_data['PLAYER_
ID']].iloc[0].copy()
                player_row['HEIGHT'] = physical_data['HEIGHT']
                player_row['WEIGHT'] = physical_data['WEIGHT']
                player_row['POSITION'] = physical_data['POSITION']
                player_row['SEASON'] = season # Corrected: Use season variabl
e
                player_data.append(player_row)

            if player_data:
                season_df = pd.DataFrame(player_data)
                all_seasons_data.append(season_df)
                print(f"Added {len(player_data)} players for season {season}")

        except Exception as e:
```

```
            print(f"Error fetching data for season {season}: {e}")

    if all_seasons_data:
        combined_stats = pd.concat(all_seasons_data, ignore_index=True)
        return combined_stats
    else:
        return pd.DataFrame()

# Execution Code
all_players_stats_threaded = get_all_players_stats_with_physical_data_threaded()
print(f"Collected data for {len(all_players_stats_threaded)} player-seasons in total (threaded)")
all_players_stats_threaded.to_csv('all_players_stats_2000_2025_threaded_cached.csv', index=False)
print("Threaded data saved to all_players_stats_2000_2025_threaded_cached.csv")
```

However, this data collection process took **too long** due to numerous API calls and increased time complexity, making it impossible to achieve the desired result.

Therefore, it was decided that this data could not be used, and a search for an alternative dataset was initiated. Fortunately, a dataset was found on **Kaggle** that contained all the required information, and it was chosen for the project.

Data Preprocessing

Data Loading

The data is loaded using pandas:

```
import pandas as pd
pd.set_option('display.max_columns', None)
df = pd.read_csv("/content/nba_player_data_1996-2024.csv")
```

Filtering Unnecessary Columns

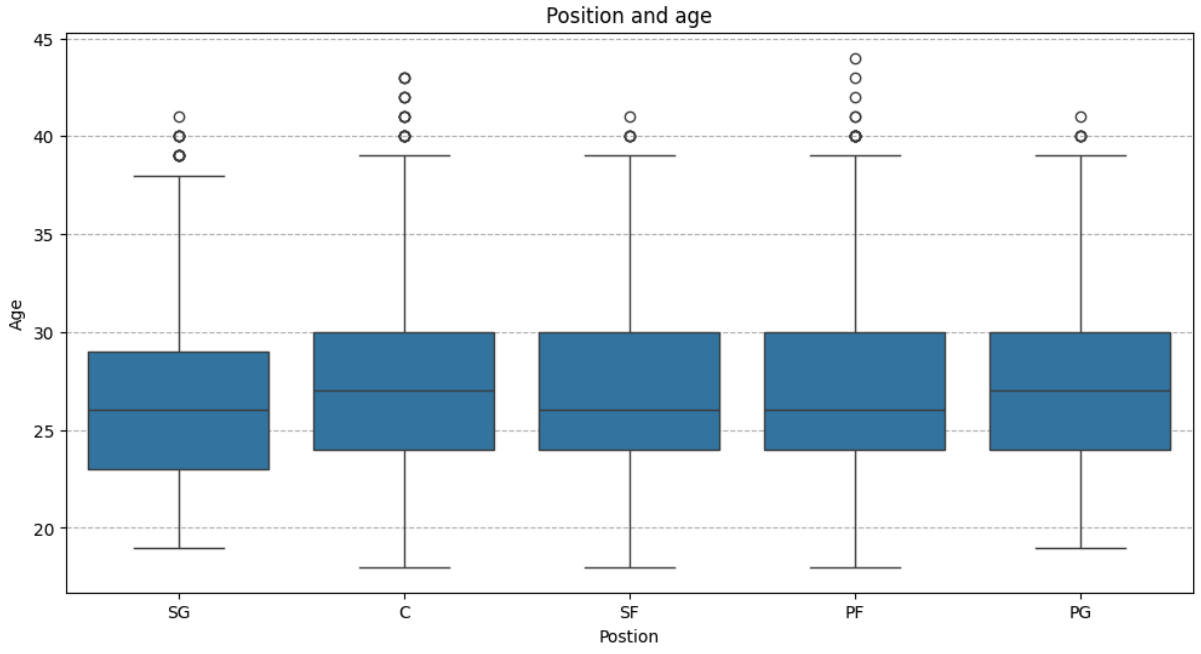
The goal is to predict player position using **'physical information'** and **'game statistics'**. Columns not fitting these criteria need to be removed.

The original columns are:

Index(['normalized_name', 'age', 'player_height', 'player_weight', 'college', 'country', 'draft_year', 'draft_round', 'draft_number', 'pts', 'reb', 'ast', 'season', 'Pos.x', 'MP.x', 'G.x', 'eFG.', 'X3P', 'X3PA', 'X3P.', 'X3PAr', 'X2P', 'X2PA', 'X2P.', 'FT', 'FTA', 'FT.', 'PER', 'TS.', 'TRB.', 'AST.', 'TOV.', 'USG.', 'WS', 'VORP', 'BPM'], dtype='object')

The `age` column was analyzed using the Interquartile Range (IQR) by position to determine its relevance:

```
import matplotlib.pyplot as plt
import seaborn as sns

df_analysis = df[['age', 'Pos.x']].copy()
df_analysis = df_analysis.dropna()
plt.figure(figsize=(12, 6))
sns.boxplot(x='Pos.x', y='age', data=df_analysis)
plt.title('Position and age')
plt.xlabel('Postion')
plt.ylabel('Age')
plt.grid(axis='y', linestyle='--')
plt.show()
```



The analysis showed that for all positions, the age of players in the 25% to 75% range fell between **23 and 27 years old**. Since there was no significant correlation between age and position, the `age` column was also deemed unnecessary.

Deleting Unnecessary Columns

The following columns were dropped:

```
columns_to_drop = ['normalized_name', 'age', 'college', 'country', 'draft_yea
r', 'draft_round', 'draft_number', 'season']
df = df.drop(columns=columns_to_drop)
```

The reasons for dropping columns are summarized in the table below:

Column Name	Column Information	Reason for Exclusion
`normalized_name`	Player's Name	Not part of physical data or game statistics.
`age`	Player's Age	Part of physical data, but IQR analysis showed a similar range for all positions, indicating no influence.
`college`	Player's College	Not part of physical data or game statistics.
`country`	Player's Country of Origin	Not part of physical data or game statistics.
`draft_year`	Player's Draft Year	Not part of physical data or game statistics.
`draft_round`	Player's Draft Round	Not part of physical data or game statistics.
`draft_number`	Player's Draft Pick Number	Not part of physical data or game statistics.
`season`	Player's Season Played	Not part of physical data or game statistics.

The remaining columns are:

Column Name	Interpretation
`player_height`	Height (cm)
`player_weight`	Weight (kg)
`pts`	Average Points
`reb`	Average Rebounds

ast	Average Assists
Pos.x	Position
MP.x	Average Minutes Played (min)
G.x	Games Played
eFG.	Effective Field Goal Percentage
X3P	Average 3-Pointers Made
X3PA	Average 3-Point Attempts
X3P.	3-Point Percentage (e.g., 32.0%)
X3PAr	3-Point Attempt Rate (vs. total shots)
X2P	Average 2-Pointers Made
X2PA	Average 2-Point Attempts
X2P.	2-Point Percentage
FT	Average Free Throws Made
FTA	Free Throw Attempts
FT.	Free Throw Percentage
PER	Player Efficiency Rating
TS.	True Shooting % (Overall Shooting Efficiency)
TRB.	Total Rebound Percentage (%)
AST.	Assist Percentage (%)
TOV.	Turnover Percentage (%)
USG.	Usage Rate (%)
WS	Win Shares (Contribution to Team Wins)
VORP	Value Over Replacement Player
BPM	Box Plus Minus (Box Score-based Team Contribution)

These columns include the necessary physical information, game statistics, and the target position for prediction.

Handling Missing Values

Checking for null values:

```
print(df.isnull().sum())
# Output shows all columns have 0 null values.
```

No missing values were found, so no further processing was required.

Handling Outliers

The initial attempt to use the **IQR** method resulted in **4000 rows** identified as outliers, which was deemed too high.

```
def find_outliers_iqr_all(df):
    numeric_cols = df.select_dtypes(include='number').columns
    outlier_indices = set()
    for col in numeric_cols:
        Q1 = df[col].quantile(0.25)
        Q3 = df[col].quantile(0.75)
        IQR = Q3 - Q1
        lower = Q1 - 1.5 * IQR
        upper = Q3 + 1.5 * IQR
        outliers = df[(df[col] < lower) | (df[col] > upper)].index
        outlier_indices.update(outliers)

    # Convert to list for indexing
    return df.loc[list(outlier_indices)]

# Execution
# df_outliers = find_outliers_iqr_all(df)
# print(f"Number of rows containing outliers: {len(df_outliers)}")
# display(df_outliers)
```

The high number of outliers is characteristic of the basketball domain, where players can have extreme heights or extremely high/low season stats (due to injuries, role changes, etc.).

The outlier detection method and criteria were thus changed:

1. Use the **MAD (Median Absolute Deviation)** method, which is less sensitive to extreme values.
2. Rows with **5 or more** outliers are removed, as these often represent players with very low playing time (e.g., season-ending injuries or deep bench players) that could negatively affect model learning.

```
import numpy as np

def detect_and_remove_heavy_mad_outliers(df, threshold=3.5, min_outlier_
count=5):
    numeric_cols = df.select_dtypes(include='number').columns
    outlier_flags = pd.DataFrame(False, index=df.index, columns=numeric_c
ols)
    for col in numeric_cols:
        median = df[col].median()
        abs_deviation = np.abs(df[col] - median)
        mad = abs_deviation.median()
        if mad == 0:
            continue
        modified_z = 0.6745 * abs_deviation / mad
        outlier_flags[col] = modified_z > threshold

    # Calculate the list and count of outlier columns
    outlier_cols_list = outlier_flags.apply(lambda row: [col for col in numeric_
cols if row[col]], axis=1)
    outlier_count = outlier_flags.sum(axis=1)

    # Create outlier summary DataFrame
    outlier_summary = df.copy()
    outlier_summary["outlier_columns"] = outlier_cols_list
    outlier_summary["outlier_count"] = outlier_count

    # Filter rows with min_outlier_count or more outliers
    heavy_outliers = outlier_summary[outlier_summary["outlier_count"] >= m
in_outlier_count]
    print(f"Number of rows with {min_outlier_count} or more outlier columns:
{len(heavy_outliers)}")

    # Return a new DataFrame with the heavy outlier rows removed
    df_cleaned = df.drop(index=heavy_outliers.index)
    print(f"Number of data rows remaining after removal: {len(df_cleaned)}")
    return df_cleaned, heavy_outliers

# Execution
```

```
df_cleaned, df_outliers_5plus = detect_and_remove_heavy_mad_outliers(df, threshold=3.5, min_outlier_count=5)

# Checking the list of removed outliers
# display(df_outliers_5plus[["outlier_columns", "outlier_count"]])
```

Feature and Target Data Separation

The target variable (`Pos.x`) is separated from the features.

```
X = df_cleaned.drop(columns=['Pos.x'])
y = df_cleaned['Pos.x']
```

Feature Engineering

New derived features were created to add complexity to the model, rather than using `PolynomialFeatures`.

```
# 1. Body Ratio
X['bmi'] = X['player_weight'] / ((X['player_height'] / 100) ** 2)

# 2. Performance per Minute (Prevent division by zero)
X['pts_per_min'] = X['pts'] / (X['MP.x'] + 1e-5)
X['reb_per_min'] = X['reb'] / (X['MP.x'] + 1e-5)
X['ast_per_min'] = X['ast'] / (X['MP.x'] + 1e-5)

# 3. Shooting Focus
X['fg_share'] = X['X3PA'] + X['X2PA']
X['3p_ratio'] = X['X3PA'] / (X['fg_share'] + 1e-5)

# 4. Composite Offensive Efficiency/Contribution Indices
X['offensive_index'] = X['TS.'] * X['USG.'] * X['AST.']
X['defensive_proxy'] = X['TRB.'] - X['TOV.']
```

The new feature set is:

Index(['player_height', 'player_weight', 'pts', 'reb', 'ast', 'MP.x', 'G.x', 'eFG.', 'X3P', 'X3PA', 'X3P.', 'X3PAr', 'X2P', 'X2PA', 'X2P.', 'FT', 'FTA', 'FT.', 'PER', 'TS.', 'TRB.', 'AST.', 'TOV.', 'USG.', 'WS', 'VORP', 'BPM', 'bmi', 'pts_per_min',

'reb_per_min', 'ast_per_min', 'fg_share', '3p_ratio', 'offensive_index', 'defensive_proxy'], dtype='object')

Splitting Data

The data is split into training and test sets.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Data Normalization

Standard scaling is applied to all features except those already representing percentages or ratios (which are already normalized between 0 and 1 or have a specific meaningful scale).

```
from sklearn.preprocessing import StandardScaler

# Define columns to scale
cols_to_scale = [col for col in X.columns if col not in ['X3PAr', 'TS.', 'FT.', 'X3P.', 'FT.', '3p_ratio']]

# Process and scale
X_train_to_scale = X_train[cols_to_scale]
X_test_to_scale = X_test[cols_to_scale]
scaler = StandardScaler()
X_train_scaled_part = scaler.fit_transform(X_train_to_scale)
X_test_scaled_part = scaler.transform(X_test_to_scale)

# Combine scaled and unscaled ratio columns
X_train_scaled = pd.DataFrame(X_train_scaled_part, columns=cols_to_scale, index=X_train.index)
X_train_scaled[X_train.columns.difference(cols_to_scale)] = X_train[X_train.columns.difference(cols_to_scale)]
X_test_scaled = pd.DataFrame(X_test_scaled_part, columns=cols_to_scale, index=X_test.index)
X_test_scaled[X_test.columns.difference(cols_to_scale)] = X_test[X_test.columns.difference(cols_to_scale)]
```

Model Training

Logistic Regression is chosen for the initial model.

```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression(solver='lbfgs', max_iter=1000, random_state=4
2)
model.fit(X_train_scaled, y_train)
```

Model Performance Evaluation

```
print(model.score(X_train_scaled, y_train))
print(model.score(X_test_scaled, y_test))
# 0.7529873039581777
# 0.7420679357969392
```

Both scores are low, indicating potential **underfitting**. Model performance improvement is required.

Model Performance Improvement

Attempts to improve model performance were made through **Hyperparameter Tuning**, **Model Change**, and **Data Adjustment**.

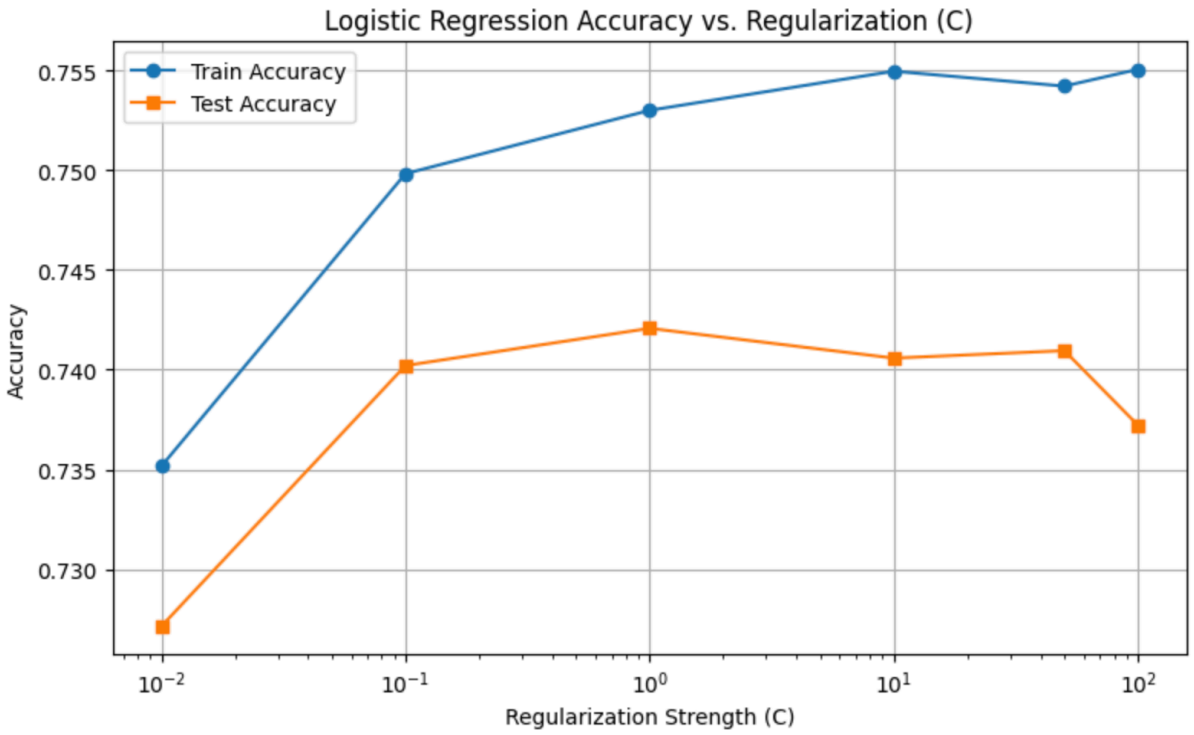
1. Hyperparameter Tuning (Regularization Strength C)

The regularization strength C in Logistic Regression was adjusted.

```
cs = [0.01, 0.1, 1.0, 10.0, 50.0, 100.0]
train_scores = []
test_scores = []
for c in cs:
    model = LogisticRegression(C=c, solver='lbfgs', max_iter=1000, random_
state=42)
    model.fit(X_train_scaled, y_train)
    train_scores.append(model.score(X_train_scaled, y_train))
    test_scores.append(model.score(X_test_scaled, y_test))

# Plotting the results
```

```
# ... (Code for plotting: 'Logistic Regression Accuracy vs. Regularization (C)')
```

```
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 5))
plt.plot(cs, train_scores, marker='o', label='Train Accuracy')
plt.plot(cs, test_scores, marker='s', label='Test Accuracy')

plt.xscale('log')
plt.xlabel('Regularization Strength (C)')
plt.ylabel('Accuracy')
plt.title('Logistic Regression Accuracy vs. Regularization (C)')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```



Result: The performance remained mostly similar regardless of the C value, indicating that regularization strength did not significantly influence the model's

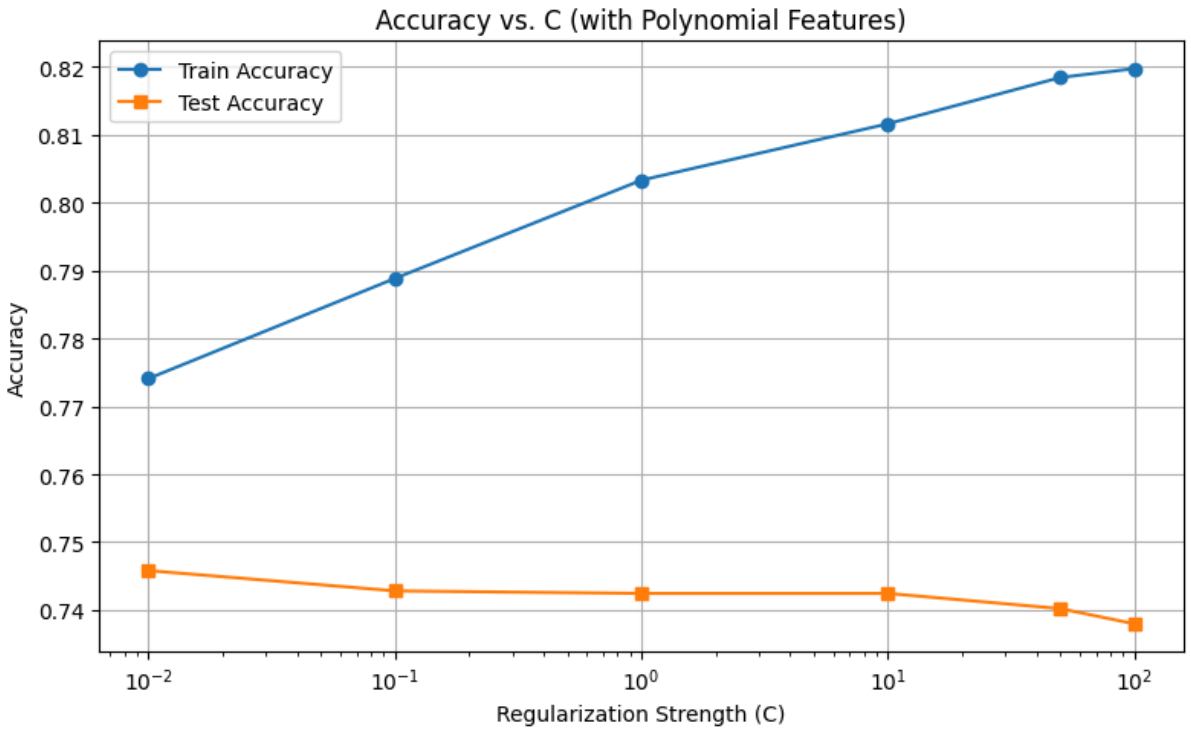
performance. **(Failure)**

2. Polynomial Feature Creation

The model complexity was increased by adding polynomial features (degree=2).

```
from sklearn.preprocessing import PolynomialFeatures, StandardScaler

# Create polynomial features (degree=2)
poly = PolynomialFeatures(degree=2, include_bias=False)
X_train_poly = poly.fit_transform(X_train_scaled)
X_test_poly = poly.transform(X_test_scaled)

# Re-scaling (always necessary!)
scaler_poly = StandardScaler()
X_train_poly_scaled = scaler_poly.fit_transform(X_train_poly)
X_test_poly_scaled = scaler_poly.transform(X_test_poly)

# Re-evaluate with Logistic Regression and C tuning
# ... (Code for C tuning and plotting: 'Accuracy vs. C (with Polynomial Featu
res)')
```

```
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 5))
plt.plot(cs, train_scores, marker='o', label='Train Accuracy')
plt.plot(cs, test_scores, marker='s', label='Test Accuracy')

plt.xscale('log')
plt.xlabel('Regularization Strength (C)')
plt.ylabel('Accuracy')
plt.title('Accuracy vs. C (with Polynomial Features)')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

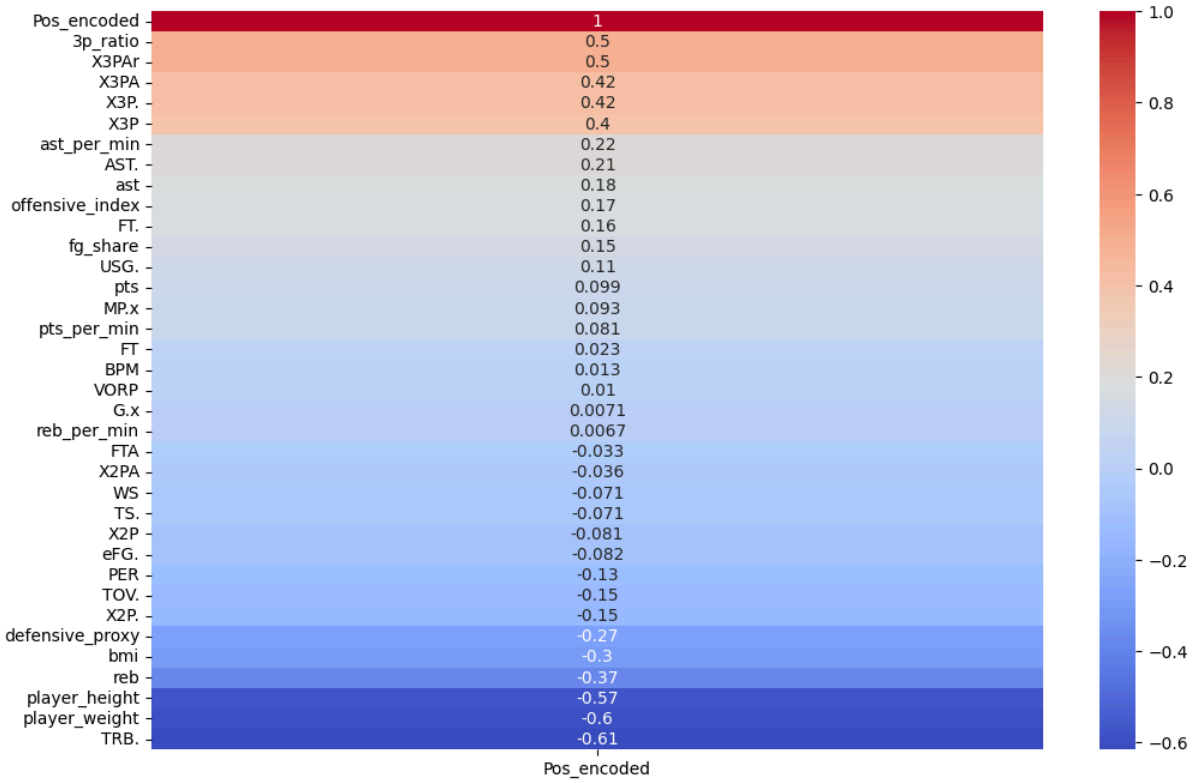


Result: As regularization decreased, training accuracy improved, but test accuracy stagnated or even slightly decreased. The model was more complex, but the performance improvement was marginal or non-existent on the test set. **(Failure)**

3. Feature Engineering (Correlation Analysis)

The correlation between each feature and the target position was analyzed to select the most relevant features

```
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelEncoder

# Label Encoding the position for correlation
df_corr = X_train_scaled.copy()
df_corr['Pos_encoded'] = LabelEncoder().fit_transform(y_train)
corr = df_corr.corr(numeric_only=True)
```



Based on the correlation heatmap, columns with low correlation were dropped:

```
drop_low_corr = ['G.x', 'MP.x', 'FTA', 'PER', 'VORP', 'BPM', 'bmi', 'eFG.', 'TS.', 'pts', 'FT.', 'pts_per_min']
X_filtered_train = X_train_scaled.drop(columns=drop_low_corr, errors='ignore')
X_filtered_test = X_test_scaled.drop(columns=drop_low_corr, errors='ignore')
model.fit(X_filtered_train, y_train)
print(model.score(X_filtered_train, y_train))
print(model.score(X_filtered_test, y_test))
# 0.75
# 0.7394550205300485
```

Using a more curated list of high-correlation features:

```
selected_features = [
    'player_height', 'player_weight', 'TRB.', 'X3PA', 'X3P.', 'X3PAr', 'ast_per_min', 'AST.', 'offensive_index'
]
X_selected_train = X_train_scaled[selected_features]
```

```
X_selected_test = X_test_scaled[selected_features]
model.fit(X_selected_train, y_train)
print(model.score(X_selected_train, y_train))
print(model.score(X_selected_test, y_test))
# 0.731235997012696
# 0.7278835386338186
```

Result: Neither feature selection method significantly improved model performance. **(Failure)**

4. Model Change

A. K-Nearest Neighbors (KNN)

KNN was tested as an alternative, but it is a simpler model, and since the problem was underfitting, a performance decrease was expected.

```
from sklearn.neighbors import KNeighborsClassifier
model_knn = KNeighborsClassifier(n_neighbors=5)
model_knn.fit(X_train_scaled, y_train)
print(model_knn.score(X_train_scaled, y_train))
print(model_knn.score(X_test_scaled, y_test))
# 0.7896751306945482 (Train)
# 0.6618141097424413 (Test)
```

Result: Test accuracy decreased further. **(Failure)**

B. Random Forest Classifier

Random Forest, a more complex and often high-performing classification model, was tested.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

# Model creation and training
model_rf = RandomForestClassifier(n_estimators=100, random_state=42)
model_rf.fit(X_train_scaled, y_train)

# Prediction
y_train_pred = model_rf.predict(X_train_scaled)
```

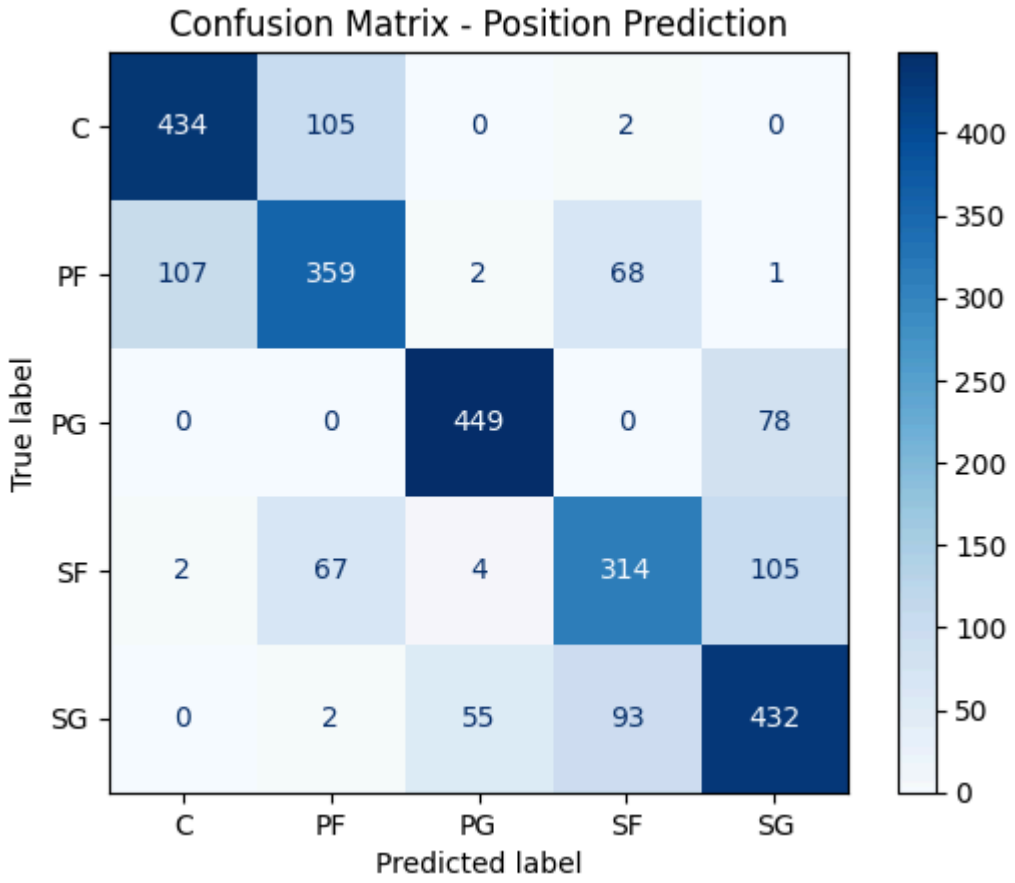
```
y_test_pred = model_rf.predict(X_test_scaled)

# Output accuracy
print("Train Accuracy:", accuracy_score(y_train, y_train_pred))
print("Test Accuracy :", accuracy_score(y_test, y_test_pred))
# Train Accuracy: 1.0
# Test Accuracy : 0.7607316162747294
```

Result: Training accuracy reached 1.0 (perfect fit), but the test accuracy only slightly increased to the **75%–76%** range, which was consistent across most trials.

Reason for Model Performance Improvement Failure

A **Confusion Matrix** was analyzed to understand where the misclassifications were occurring:

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt

# Prediction
y_pred = model_rf.predict(X_test_scaled)

# Confusion Matrix
cm = confusion_matrix(y_test, y_pred, labels=model_rf.classes_)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=mode
l_rf.classes_)

# Visualization
plt.figure(figsize=(8,6))
disp.plot(cmap='Blues', values_format='d')
plt.title("Confusion Matrix - Position Prediction")
plt.show()
```



The common misclassifications observed were:

- **PG and SG** classification
- **SF and SG** classification
- **C and PF** classification

These positions are genuinely ambiguous even for real-life basketball players. For example, a Power Forward (PF) in one season might play as a Center (C) in another, and the boundaries between Small Forward (SF) and Shooting Guard (SG), or Point Guard (PG) and Shooting Guard (SG), are often blurred.

It was concluded that the difficulty in clearly distinguishing these positions is a **limitation of the basketball domain itself**, not a problem with the model or the data.

Target Data Simplification

To test this hypothesis, the target data was simplified into three broader categories: **Guard (G)**, **Forward (F)**, and **Big (B)**.

- `PG`, `SG` → `G`

- `SF` → `F`
- `PF`, `C` → `B`

```
def simplify_pos(pos):
    if pos in ['PG', 'SG']:
        return 'G'
    elif pos in ['SF']:
        return 'F'
    else:
        return 'B' # Big: PF, C

y_train_simple = y_train.apply(simplify_pos)
y_test_simple = y_test.apply(simplify_pos)
model_rf.fit(X_train_scaled, y_train_simple) # Retraining Random Forest mo
del

print(model_rf.score(X_train_scaled, y_train_simple))
print(model_rf.score(X_test_scaled, y_test_simple))
# 0.8703323375653472 (Train)
# 0.8701007838745801 (Test)
```

Result: Model performance increased significantly, confirming that the initial challenge was due to the **inherent ambiguity in the five-position classification** within the basketball domain.

Conclusion

The final performance of the Logistic Regression model (the initial choice) was:

```
# Assuming model_lg refers to the final state of the Logistic Regression mo
del after initial scaling
print(model_lg.score(X_train_scaled, y_train))
print(model_lg.score(X_test_scaled, y_test))
# 0.7529873039581777
# 0.7420679357969392
```

In this project, various preprocessing, feature engineering, and model experiments were conducted to predict a basketball player's position based on

numerical data. Despite applying several classification models, including Logistic Regression, the test accuracy stalled at the **74%** level due to the real-world ambiguity between positions and the limitations of the data's information content.

However, the project confirmed that a player's **physical data and game statistics do have a significant and meaningful impact** on distinguishing their position. The major breakthrough came from simplifying the target data to a three-position classification (G/F/B), which proved that the difficulty lay in the domain-specific constraints of the five-position system.