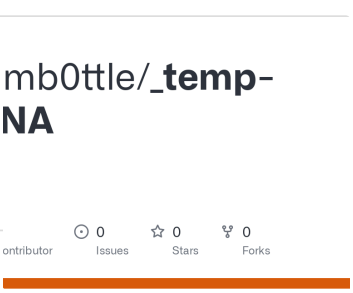
BNA 보고서

Basketball DNA

- `ipynb` 파일

_temp-BNA/2406나병현_인활프로젝트_BNA.ipynb at main · iamb0ttle/_temp-BNA
Contribute to iamb0ttle/_temp-BNA development by creating an account on GitHub.
https://github.com/iamb0ttle/_temp-BNA/blob/main/2406%E1%84%82%E1%85%A1%E1%84%87%E1%85%A7%E1%86%BC%E1%84%92%E1%85%A7%E1%86%AB_%E1%84%8B%E1%85%B5%E1%86%AB%E1%84%92%E1%85%AA%E1%86%AF%E1%84%91%E1%85%B3%E1%84%85%E1%85%A9%E1%84%8C%E1%85%A6%E1%86%A8%E1%84%90%E1%85%B3_BNA.ipynb



개요

BNA(Basketball DNA) 프로젝트는 농구 선수들의 **경기 데이터와 신체 정보를 활용하여 가장 적합한 농구 포지션(PG, SG, SF, PF, C)을 예측하고, 이를 통해 선수들의 커리어 발전을 지원하는 것을 목표**로 한다.

이 프로젝트는 현재 선수들의 잠재력을 최대한 발휘하고, 코치 및 스카우터에게는 선수 선발 및 육성에 대한 객관적인 자료를 제공하는 데 기여할 것이다.

1. 프로젝트 목표

- **선수 포지션 예측 정확도 향상:** 선수들의 다양한 데이터를 분석하여 각 포지션에 대한 예측 정확도를 높인다.
- **개인 맞춤형 커리어 가이드라인 제공:** 예측된 포지션을 기반으로 선수들에게 필요한 훈련 방향, 강점 강화 및 약점 보완 전략을 제시하여 개인의 성장을 돕는다.
- **객관적인 선수 평가 자료 제공:** 코칭 스태프 및 스카우터가 선수들의 잠재력을 객관적으로 평가하고, 최적의 포지션에 배치하는 데 필요한 근거 자료를 제공한다.

2. 주요 기능 및 내용

- **데이터 수집 및 전처리:**
  - **경기 정보:** 득점, 리바운드, 어시스트, 스틸, 블록, 턴오버, 야투 성공률, 3점슛 성공률 등 경기 내 다양한 스탯 데이터를 수집한다.
  - **신체 정보:** 키, 몸무게 등 선수의 신체적 특성 데이터를 수집한다.

    - 수집된 데이터는 결측치 처리, 정규화 등 머신러닝 모델 학습에 적합하도록 전처리 과정을 거친다.
- **머신러닝 모델 개발:**
    - 수집된 데이터를 기반으로 PG, SG, SF, PF, C 다섯 가지 포지션을 예측할 수 있는 분류 (Classification) 모델을 개발한다.
    - 다양한 머신러닝 알고리즘(예: 로지스틱 회귀, 서포트 벡터 머신, 랜덤 포레스트, 신경망 등)을 비교 분석하여 최적의 예측 성능을 보이는 모델을 선정한다.
- **예측 결과 및 분석 보고서 제공:**
    - 모델이 예측한 선수의 최적 포지션과 함께, 해당 포지션에 대한 예측 신뢰도를 제공한다.
    - 예측 결과에 대한 상세 분석 보고서(예: 특정 스탯이 포지션 예측에 미치는 영향, 해당 포지션에서 요구되는 주요 능력치 등)를 제공하여 선수와 코칭 스태프가 인사이트를 얻을 수 있도록 한다.

3. 기대 효과

- **선수 개개인의 성장 촉진:** 선수들은 자신의 강점을 극대화하고 약점을 보완하는 데 필요한 맞춤형 정보를 얻어 효율적인 훈련 계획을 수립할 수 있다.
- **팀 전력 강화 기여:** 코치진은 선수들의 잠재력을 정확하게 파악하여 최적의 포지션에 배치하고, 팀 전술에 맞는 선수 선발 및 육성 전략을 수립할 수 있다.
- **농구 데이터 분석 분야 발전:** 스포츠 데이터 분석의 중요성을 인식시키고, 더욱 심층적인 농구 데이터 분석 연구의 기반을 마련한다.

데이터 수집

본래 프로젝트 계획은 nba.com에 존재하는 데이터를 이용하는 것 이였으나, 해당 데이터는 다음과 같은 절차를 거쳐 데이터를 만들어야했다.

1. 특정 시즌에 출전한 선수들의 데이터를 이용해 선수의 시즌 성적을 불러온다.
2. 각 선수에 대해 나이 데이터, 키, 몸무게 데이터 등을 행에 추가한다. (1번째 API 호출 발생)
    a. 먼저 선수 이름으로 선수의 나이, 키, 몸무게 데이터를 불러오고 (2번째 API 호출 발생)
    b. 그 선수 행을 찾아 행에 삽입
3. 이 과정을 모든 선수에 대해 반복한다.

```
import pandas as pd
from nba_api.stats.endpoints import LeagueDashPlayerStats, CommonPlayerInfo
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_info['CommonPlayerIn
        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, 'WEIGHT': weight, 'POSIT
        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, 'POS
            cache_player_info(player_id, info)  # API 호출 성공 시 캐싱
            print(f"Fetched physical data for player {player_id} from API")
        else:
            print(f"No physical data found for player {player_id} in season {season}")
        time.sleep(1.0)
```

```
    except Exception as e:
        print(f"Error fetching physical data for player {player_id} in season {season}: {e}")

# 나머지 코드는 이전과 동일하게 유지 (스레드 처리 등)
# ...

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, p
                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'] = physical_data['SEASON']
                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()

# 실행 코드
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=Fa
print("Threaded data saved to all_players_stats_2000_2025_threaded_cached.csv")
```

하지만 이 과정은 데이터를 수집하는 과정이 api 호출 때문에 너무 오래걸렸고, 시간복잡도가 증가하여서 원하는 결과를 얻지 못했다.

따라서 해당 데이터를 사용할수 없다고 판단하여, 다른 데이터 셋이 존재하는지 찾아보기로 결정하였다.

다행히도 Kaggle에 **NBA Player Data (1996-2024)**라는 데이터셋이 존재하였으며, 내가 원하는 데이터를 모두 포함하고 있었기에, 해당 데이터를 사용하도록 결정하였다.

데이터 전처리

데이터 불러오기

먼저 데이터를 불러와준다.

```
import pandas as pd

pd.set_option('display.max_columns', None)

df = pd.read_csv("/content/nba_player_data_1996-2024.csv")
```

필요없는 컬럼 분류

이제 필요없는 데이터와 필요한 데이터를 구분해야하는데, 우리의 목표는 선수의 '신체 정보'와 '경기 기록'을 이용해 선수의 포지션을 예측하는 것이 목표이기 때문에, 선수의 신체 정보나 경기기록에 해당하지 않는 컬럼을 제거해줘야 한다.

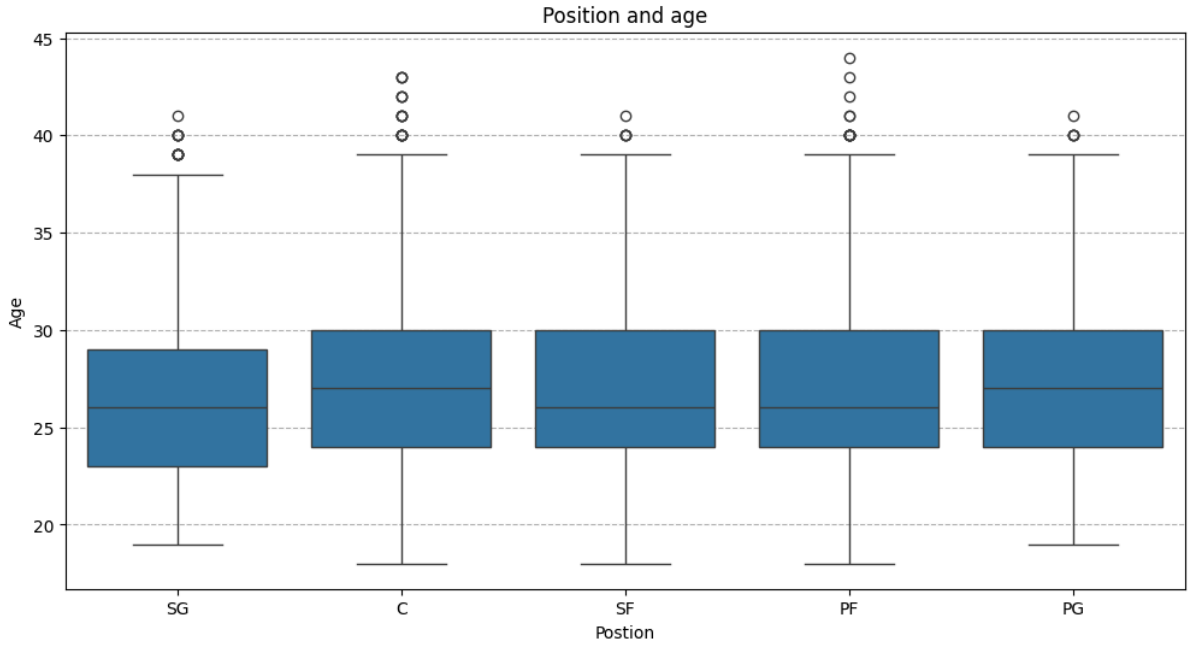
먼저 어떤 컬럼들이 존재하는지 확인해보자

```
df.columns
```

```
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')
```

나이 컬럼의 경우 신체정보에 속하지만 선수의 포지션에 영향을 주는 신체정보는 아니라고 판단하여서, 선수의 나이와 신체정보의 상관관계를 알아보기 위해 포지션별 나이의 사분위수 범위(IQR)를 이용하여 판단하기로 하였다.

```
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()
```



와 같이 모든 포지션이 동일하게 `25% ~ 75%` 구간 선수의 나이가 `23세 ~ 27세` 사이에 속한다는 것을 알 수 있다. 따라서, 나이와 포지션의 상관관계는 없으므로, 나이 컬럼 또한 불필요한 컬럼이 된다.

필요없는 컬럼 삭제

이제 필요없는 컬럼들을 삭제해주자.

```
columns_to_drop = ['normalized_name', 'age', 'college', 'country', 'draft_year', 'draft_round', 'dra
df = df.drop(columns=columns_to_drop)
```

필요없는 컬럼 목록을 리스트로 담아서, df의 `drop()` 함수를 이용해 필요없는 컬럼을 삭제한다.

필요없는 컬럼의 목록과 필요없다고 판단한 이유는 각각 다음과 같다.

컬럼 이름	컬럼 정보	필요없다고 판단한 이유
`normalized_name`	선수의 이름	선수의 신체데이터, 경기기록에 해당하지 않음
`age`	선수의 나이	선수의 신체데이터에 해당하나, 사분위수 범위를 분석하였을때 모든 포지션이 비슷한 나이의 범위를 가진다는 것을 고려하여 나이는 영향을 주지 않는다고 판단하였음
`college`	선수의 출신 대학	선수의 신체데이터, 경기기록에 해당하지 않음

컬럼 이름	컬럼 정보	필요없다고 판단한 이유
country	선수의 출신 나라	선수의 신체데이터, 경기기록에 해당하지 않음
draft_year	선수의 드래프트 선발 년도	선수의 신체데이터, 경기기록에 해당하지 않음
draft_round	선수의 드래프트 선발 라운드	선수의 신체데이터, 경기기록에 해당하지 않음
draft_number	선수의 드래프트 선발 순위	선수의 신체데이터, 경기기록에 해당하지 않음
season	선수의 출전 시즌	선수의 신체데이터, 경기기록에 해당하지 않음

이제 남은 컬럼들은 다음과 같다.

컬럼명	해석
player_height	키 (cm)
player_weight	몸무게 (kg)
pts	평균 득점
reb	평균 리바운드
ast	평균 어시스트
Pos.x	포지션
MP.x	평균 출전 시간 (분)
G.x	출전 경기 수
eFG.	유효 필드골 성공률
X3P	평균 3점슛 성공 개수
X3PA	평균 3점슛 시도 횟수
X3P.	3점슛 성공률 (32.0%)
X3PAr	전체 슛 중 3점슛 시도 비율
X2P	평균 2점슛 성공 개수
X2PA	평균 2점슛 시도 횟수
X2P.	2점슛 성공률
FT	평균 자유투 성공 개수
FTA	자유투 시도
FT.	자유투 성공률
PER	플레이어 효율성 등급 (Player Efficiency Rating)
TS.	True Shooting % (종합 슈팅 효율)
TRB.	리바운드율 (%)
AST.	어시스트율 (%)

컬럼명	해석
TOV.	턴오버율 (%)
USG.	공격 점유율 (Usage Rate)
WS	Win Shares (팀 승리에 대한 기여도)
VORP	대체 선수 대비 공헌도 (Value Over Replacement Player)
BPM	Box Plus Minus (박스스코어 기반 팀 기여도)

모두 선수의 포지션을 예측하는데 필요한 신체 정보와 경기 기록, 그리고 포지션까지 포함되어있는 것을 알 수 있다.

결측치 처리

이제 결측치를 처리하기 위해 `isnull()`과 `sum()` 함수를 조합해서 살펴보자.

```
print(df.isnull().sum())
```

```
player_height    0
player_weight    0
pts              0
reb              0
ast              0
Pos.x            0
MP.x             0
G.x              0
eFG.             0
X3P              0
X3PA             0
X3P.             0
X3PAr            0
X2P              0
X2PA             0
X2P.             0
FT               0
FTA              0
FT.              0
PER              0
TS.              0
TRB.             0
AST.             0
```

```
TOV.          0
USG.          0
WS            0
VORP          0
BPM           0
dtype: int64
```

다행히 결측치가 모든 컬럼에 존재하지 않는 것을 확인하였다. 따로 결측치 처리는 하지 않아도 될것 같다.

이상값 처리

이상값을 처리하기 위해서는, 이상치를 판단할 기법을 정해야 한다. 맨 처음에 IQR 기법으로 판단하였는데 이상치가 존재하는 행이 4000개가 나왔다.

```
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)

    # 리스트로 변환해서 인덱싱
    return df.loc[list(outlier_indices)]

# 실행
df_outliers = find_outliers_iqr_all(df)
print(f"이상치가 포함된 행 수: {len(df_outliers)}")
display(df_outliers)
```

해당 데이터에 이렇게 이상치가 많이 존재하는 이유는 농구라는 도메인의 특성상 키가 엄청나게 큰 선수가 존재할 수 도 있고, 키가 엄청나게 작은 선수가 존재할수도 있으며, 또한 특정 선수는 특정 시즌에 비약적으로 많은 득점을 하거나 특정 시즌에 시즌 아웃을 당해 아예 뛰지 못할 수도 있기 때문이다.

따라서 이상치를 판단하는 기법과 기준을 다음과 같이 설정 하였다.

- 이상치 판단 기법은 MAD (Median Absolute Deviation) 기법을 이용한다.
  → MAD 기법은 극단적인 값에 민감하지 않으므로 농구 도메인의 특성을 고려하여 이상치를 판단하기 유용하다.
- 이상치가 **5**개 이상 존재하는 행은 삭제하고, 아닌 행은 유지하기로 하였다.
  → 이상치가
  5개 이상 존재하는 행일 경우 시즌 아웃 되어 출전을 적게 하거나, 벤치 멤버일 경우가 존재하기 때문에 모델 학습에 악영향을 줄 수 있다.

```
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_cols)

    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

    # 이상치 컬럼명 및 개수 계산
    outlier_cols_list = outlier_flags.apply(lambda row: [col for col in numeric_cols if row[col]], axis
    outlier_count = outlier_flags.sum(axis=1)

    # 이상치 요약 데이터프레임 생성
    outlier_summary = df.copy()
    outlier_summary["outlier_columns"] = outlier_cols_list
    outlier_summary["outlier_count"] = outlier_count

    # 이상치가 min_outlier_count 이상인 행만 필터링
    heavy_outliers = outlier_summary[outlier_summary["outlier_count"] >= min_outlier_count]
    print(f"이상치 컬럼이 {min_outlier_count}개 이상인 행 수: {len(heavy_outliers)}")

    # 해당 이상치 행을 제거한 새로운 DataFrame 반환
    df_cleaned = df.drop(index=heavy_outliers.index)
    print(f"제거 후 남은 데이터 행 수: {len(df_cleaned)}")
```

```
    return df_cleaned, heavy_outliers

# 실행
df_cleaned, df_outliers_5plus = detect_and_remove_heavy_mad_outliers(df, threshold=3.5, min_c

# 이상치 목록 확인
display(df_outliers_5plus[["outlier_columns", "outlier_count"]])
```

특성(feature) 데이터와 예측(target) 데이터 분류

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

```
y = df['Pos.x']
```

예측 데이터인 Pos.x를 별도로 분류한다.

특성 공학으로 특성 생성

이제 모델을 좀더 복잡하게 만들기 위해서 존재하는 컬럼들을 이용해 새로운 파생 속성들을 만들어낼 것 이다.

먼저 특성 데이터에 존재하는 컬럼들을 확인한다.

```
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')
```

이제 해당 특성들을 조합해 새로운 특성을 생성해낸다. 의미 있는 데이터를 만들기 위해 `PolynomialFeatures` 를 사용하지 않고 직접 생성한다.

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

```
# 2. 출전 시간 대비 성과 (0 나눔 방지)
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. 슈팅 집중도
X['fg_share'] = X['X3PA'] + X['X2PA']
X['3p_ratio'] = X['X3PA'] / (X['fg_share'] + 1e-5)

# 4. 공격 효율/기여도 복합 지표
X['offensive_index'] = X['TS.'] * X['USG.'] * X['AST.']
X['defensive_proxy'] = X['TRB.'] - X['TOV.']
```

훈련 데이터와 학습 데이터 나누기

훈련 데이터와 학습 데이터는 `tarin_test_split()`을 이용해 나눠준다.

```
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)
```

데이터 정규화

```
from sklearn.preprocessing import StandardScaler

# 정규화할 컬럼만 따로 지정
cols_to_scale = [col for col in X.columns if col not in ['X3PAr', 'TS.', 'FT.', 'X3P.', 'FT.', '3p_ratio']]

# 나눠서 처리
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)

# 비율 컬럼은 그대로 유지해서 다시 합치기
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(c
```

```
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
```

데이터 정규화는 이미 0~1인 값으로 정규화 된 컬럼이거나, 의미 있는 값을 가지는 컬럼들을 제외하고 수행한다.

모델 학습

이제 모델을 학습할 차례이다.

```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression(solver='lbfgs', max_iter=1000, random_state=42)
model.fit(X_train_scaled, y_train)
```

로지스틱 회귀를 이용할 것이다.

모델 성능 평가

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

```
0.7529873039581777
0.7420679357969392
```

두 점수 모두 낮은 점수로 나타나고 있다. 이는 과소 적합에 해당한다고 볼 수 있다. 따라서 모델의 성능을 올려야 한다.

모델 성능 개선

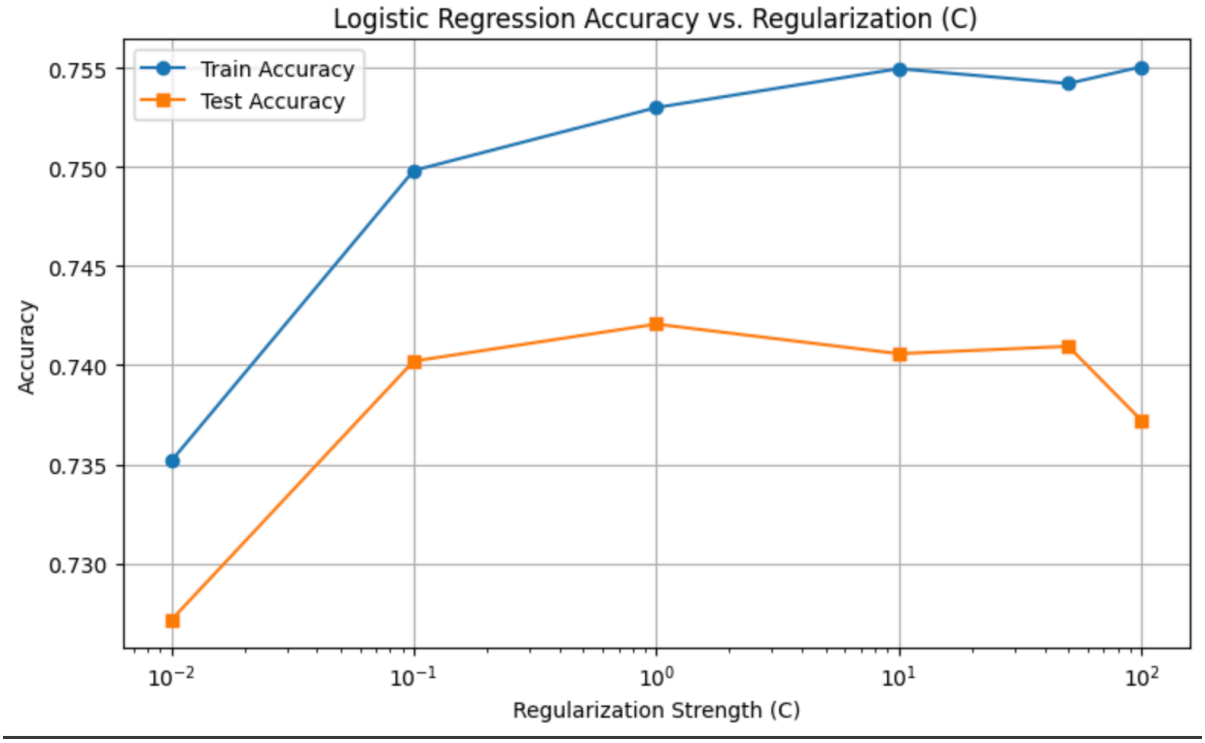
모델의 성능을 개선하기 위한 방법은 여러가지가 있다. 하이퍼 파라미터 조정, 모델 변경, 데이터 조정 등이 있다. 먼저, 하이퍼 파라미터 부터 조정하는 방식으로 모델의 성능 개선을 시도해보도록 하겠다.

하이퍼 파라미터 조정

로지스틱 회귀에서 조정하여 성능 개선을 기대할 수 있는 하이퍼 파라미터로는 규제 정도를 정의하는 `C` 가 존재한다. 따라서, `C` 값을 조정해가며 시각화를 통해 성능이 언제 가장 개선되는지 확인해보도록 하겠다.

```
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))
```

```
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')  # C 값이 로그 스케일이기 때문에 log 축으로 시각화
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()
```



해당 그래프를 보면, 규제 정도가 너무 크거나 작지 않은 이상, 대부분 비슷한 수준의 성능을 보여주고 있다. 즉, 규제 정도에 따라서 모델의 성능이 크게 좌우 되지 않는다. 따라서 실패

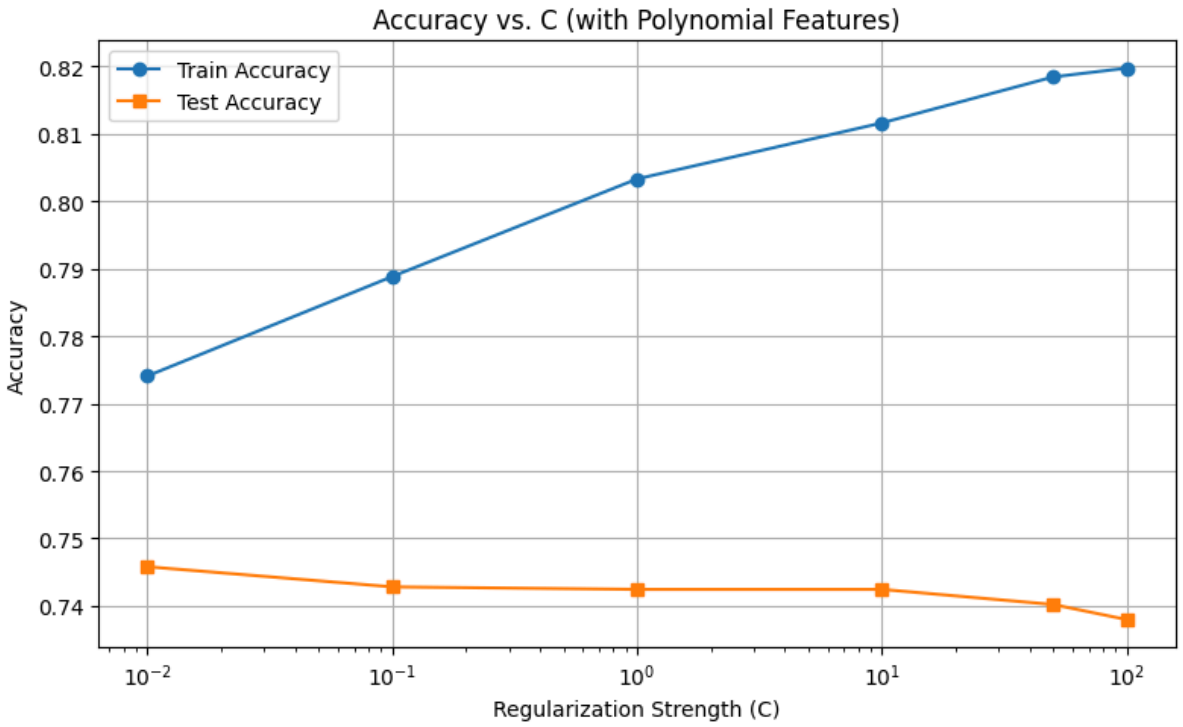
다항 특성 생성

그렇다면 이제 모델의 특성 수를 늘려 모델을 조금 더 복잡하게 만들어보자.

```
from sklearn.preprocessing import PolynomialFeatures, StandardScaler

# 다항 특성 생성 (degree=2 또는 3 추천)
poly = PolynomialFeatures(degree=2, include_bias=False)
X_train_poly = poly.fit_transform(X_train_scaled)
X_test_poly = poly.transform(X_test_scaled)

# 다시 스케일링 (항상 필요!)
scaler_poly = StandardScaler()
X_train_poly_scaled = scaler_poly.fit_transform(X_train_poly)
X_test_poly_scaled = scaler_poly.transform(X_test_poly)
```

```
from sklearn.linear_model import LogisticRegression

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=3000, random_state=42)
    model.fit(X_train_poly_scaled, y_train)
    train_scores.append(model.score(X_train_poly_scaled, y_train))
    test_scores.append(model.score(X_test_poly_scaled, y_test))
```

```
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()
```



이번에는 모델을 조금더 복잡하게 만들고, 규제 정도를 조정해가며 모델의 성능을 관찰해보았다. 하지만 여기서 의문이 드는 것은, 규제 정도가 약해짐에 따라서 `train` 에 대한 성능은 개선되고 있지만, `test` 에 대한 성능은 그대로이거나 오히려 감소하는 경향을 띄고 있다.

어쨌든 모델의 성능 개선은 실패했으니 다음 방법을 사용해보자.

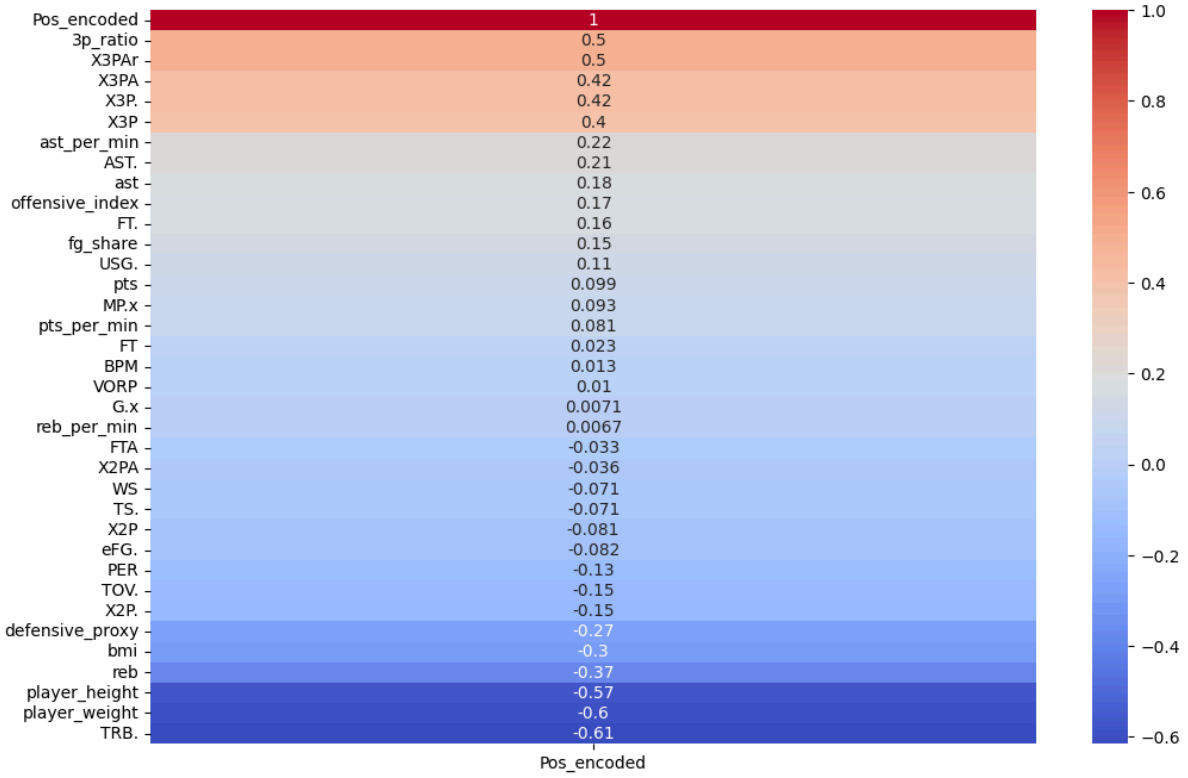
특성 공학

이제 모델에서 최소한의 데이터를 이용해서 모델을 학습하는 방식으로 성능 개선을 유도해볼 것 이다.

먼저 시각화를 통해 특성별 타겟값과의 상관관계를 분석해보자.

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

# 포지션을 수치형으로 LabelEncoding 먼저
from sklearn.preprocessing import LabelEncoder
df_corr = X_train_scaled.copy()
df_corr['Pos_encoded'] = LabelEncoder().fit_transform(y_train)

corr = df_corr.corr(numeric_only=True)
```

```
plt.figure(figsize=(12,8))
sns.heatmap(corr[['Pos_encoded']].sort_values(by='Pos_encoded', ascending=False), annot=Tr
```



이렇게 포지션을 예측하는데 상관도가 높은 컬럼들과 아닌 컬럼들이 한눈에 파악되었다. 이제 컬럼들을 지우거나 남기는 방식으로 성능 개선을 유도한다.

```
drop_low_corr = ['G.x', 'MP.x', 'FTA', 'PER', 'VORP', 'BPM', 'bmi', 'eFG.', 'TS.', 'pts', 'FT.', 'pts_per_
X_filtered_train = X_train_scaled.drop(columns=drop_low_corr)
X_filtered_test = X_test_scaled.drop(columns=drop_low_corr)
```

```
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
```

```
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
```

해당 방법 역시 모델의 성능 개선을 성공시키지 못하였다.

모델 변경

그렇다면 대체 모델인 KNN 에서는 어느정도 성능이 나오는지 확인해보자.

```
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
0.6618141097424413
```

우리가 현재 직면한건 과소 적합으로, 모델이 충분히 복잡하지 않아서 생기는 문제였다. 하지만 KNN은 로지스틱 회귀보다 단순한 모델이므로 성능이 더욱 저하되어 나타나는 것을 볼 수 있다.

그렇다면 분류에서 성능이 좋은 다른 모델을 사용해보자.

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

# 모델 생성 및 학습
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train_scaled, y_train)

# 예측
y_train_pred = model.predict(X_train_scaled)
y_test_pred = model.predict(X_test_scaled)

# 정확도 출력
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
```

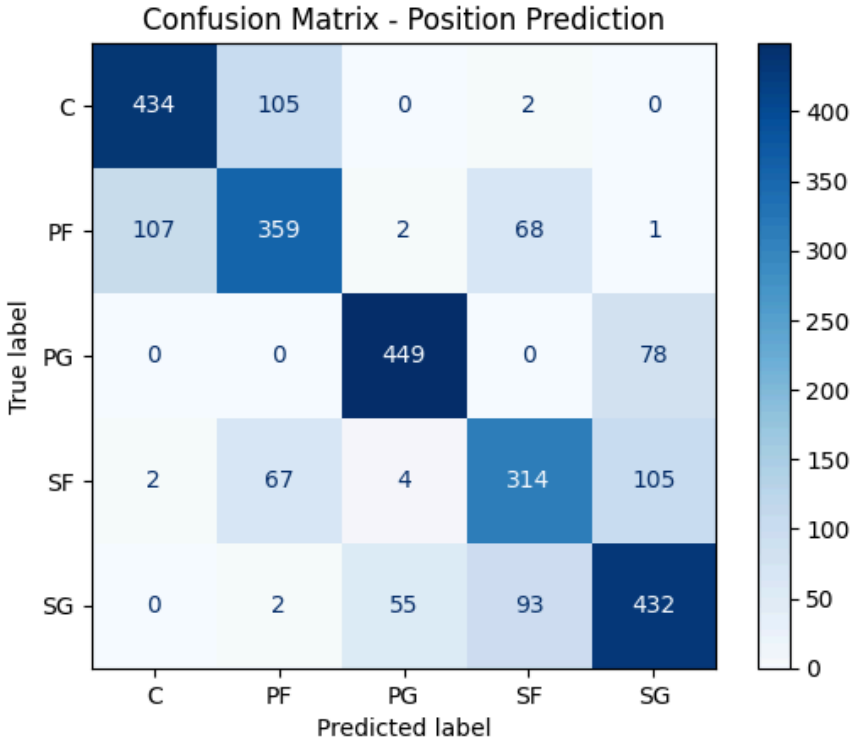
뭔가 이상하다. 학습 데이터의 성능은 매우 올랐지만 테스트 데이터의 성능이 오르지 않고 있다.
지금 까지 대부분의 상황에서 테스트 데이터의 성능은
`75%~76%` 수준이 유지 되었다.

모델 성능 개선 실패 이유

모델의 성능 개선이 실패한 한가지 예측할 수 있는 이유는, 농구라는 도메인의 특징상 포지션 분류가 특정 포지션에서 애매해 진다는 것이다. 우선 해당 가설이 실제로 맞는지 시각화를 통해 알아보자.

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

# 예측
y_pred = model.predict(X_test_scaled)

# 혼동 행렬
cm = confusion_matrix(y_test, y_pred, labels=model.classes_)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=model.classes_)

# 시각화
```

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



해당 시각화는 혼동 행렬로, 모델이 어느 포지션을 다른 포지션으로 잘못 예측하고 있는지 알 수 있다.
현재 공통적으로 보이는 잘못된 예측은 다음과 같다.

- PG와 SG의 분류
- SF와 SG의 분류
- C와 PF의 분류

실제로 해당 포지션들은 실제 농구 선수들끼리도 구분이 애매하다. 이전 시즌에 PF로 출전했던 선수가 이번시즌에는 C로 출전하는 경우도 많고, SF와 SG는 둘다 슛 시도를 많이 하는 부분에서 비슷하고, PG와 SG의 경계도 애매하기 때문이다.

따라서 해당 포지션들을 명확히 분류하는건 농구 도메인 특정상 어려우며, 이는 모델이나 데이터의 문제가 아닌 농구 **도메인의 한계**였던 것이다.

타겟 데이터 단순화

만약 우리가 언급한 농구 도메인의 한계가 정말 옳바른 가설이라면, 타겟 데이터를 단순화했을때 모델의 성능이 개선되어야 한다.

```
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)
```

```
print(model.score(X_train_scaled, y_train_simple))
print(model.score(X_test_scaled, y_test_simple))

0.8703323375653472
0.8701007838745801
```

이렇게 모델의 성능이 대폭 향상된 것을 확인할 수 있다. 즉, 우리가 직면한 문제점은 모델, 데이터의 문제가 아니라 농구 도메인의 문제였던 것이다.

결론

최종적인 모델의 성능은 아래와 같다.

```
print(model_lg.score(X_train_scaled, y_train))
print(model_lg.score(X_test_scaled, y_test))

0.7529873039581777
0.7420679357969392
```

이 프로젝트에서는 농구 선수의 수치 데이터를 기반으로 포지션을 예측하기 위한 다양한 전처리, 파생 변수 생성, 모델 실험을 수행하였다. 로지스틱 회귀를 포함한 여러 분류 모델을 적용했지만, 포지션 간 구분이 애매한 현실적 특성과 데이터 정보량의 한계로 테스트 정확도는 `74%` 수준에서 정체되었다.

하지만 선수의 신체 데이터, 경기 기록이 실제로 포지션을 구분하는데 유의미한 영향을 끼친다는 것을 확인 할 수 있었다.