# 축구선수 이적료 예측 모델링

# PORTFOLIO

Team FC203
응용통계학과 김찬영, 김청환, 이현석

Team FC203

# 분석 순서

## 1. 데이터 수집 및 탐색

축구선수의 정보가 들어있는 데이터 수집 및 탐색

## 2. 데이터 전처리

구조 변수 처리, 중복/결측값 보정, 변수 생성 및 연계

## 3. 모델링

여러가지 예측 모델을 이용하여 가장 성능이 좋은 모델 탐색

## 4. 모델 평가

실제 이적료에 어떤 요인이 얼마나 영향을 주는지 파악

# TABLE OF CONTENTS

# 01

# 데이터 수집 및 탐색

Data Collection and Exploration

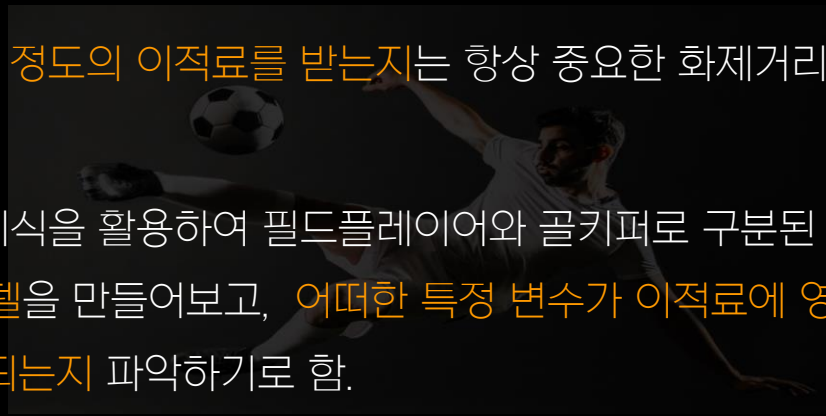
# 1. 데이터 수집 및 탐색

## 1) 주제 선정 이유

세 팀원 모두 축구에 관심이 많기 때문에 주제의 방향성을 축구로 정함.

현대 축구에서 어떤 선수가 어느 정도의 이적료를 받는지는 항상 중요한 화제거리가 됨.

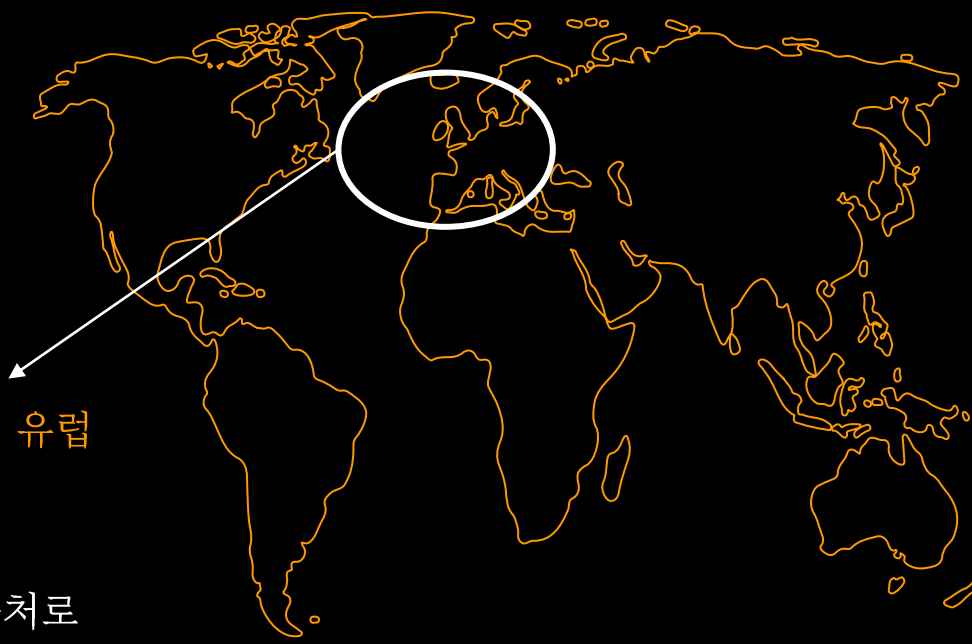
수업 시간에 배운 통계학 관련 지식을 활용하여 필드플레이어와 골키퍼로 구분된 데이터에서 이적료를 예측하는 모델을 만들어보고, 어떠한 특정 변수가 이적료에 영향을 주는지, 그 영향이 어느정도 되는지 파악하기로 함.



# 1. 데이터 수집 및 탐색

## 2) 데이터 수집

transfermarkt_fbref_201718.csv
transfermarkt_fbref_201819.csv
transfermarkt_fbref_201920.csv



17-18시즌 ~ 19-20시즌 3년 동안의 유럽 5대리그 축구 선수 데이터를 대상

transfermarkt.de와 fbref.com를 출처로 하는 Kaggle Open dataset을 활용

https://www.kaggle.com/kriegsmaschine/soccer-players-values-and-their-statistics

# 02

# 데이터 전처리

Data Preprocessing

# 2. 데이터 전처리

## 1) 결측치 처리 및 중복 확인

```
# Processing missing values for each season's data
df17 = df17.fillna(0)
df17 = df17[df17.foot!=0]
df18.dropna(axis=0, inplace=True)
df19['CLBestScorer'] = df19['CLBestScorer'].fillna(0)
df19.dropna(axis=0, inplace=True)

# Changing column name('Unnamed: 0') / Data merging
df17.rename(columns={'Unnamed: 0':'Column1'}, inplace=True)
df0 = pd.concat([df17, df18])
df1 = pd.concat([df0, df19])
df1.tail()
```

```
# Identifying duplicated rows
duplicated_data = df1[df1.duplicated()]
duplicated_data = duplicated_data.dropna()
duplicated_data
```

```
# Index reset
df1.reset_index(drop=True, inplace=True)
df1.tail()
```

(1) 각 시즌별 데이터프레임의 결측치 확인 및 삭제

df17 - 17-18시즌 / df18 - 18-19시즌 / df19 - 19-20시즌

(2) Data merging 후 중복된 데이터 확인

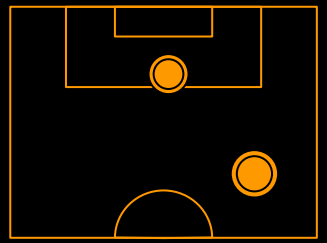
df17 + df18 + df19 => df1

# 2. 데이터 전처리

## 2) 중복 데이터 정리 및 불필요한 변수 삭제

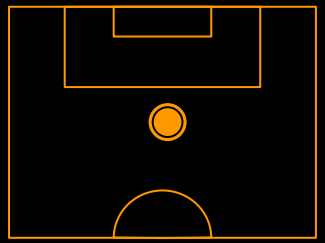
나이(age)와 출생연도(birth_year)는 동일한 정보를 내포 → 출생연도 변수를 삭제

포지션과 상세포지션의 값에 대한 정리



```
>>
DF        2087
MF        1412
FW         908
FW,MF      784
MF,FW      668
GK         524
DF,MF      314
MF,DF      152
DF,FW       56
FW,DF       46
GK,MF        1
```

```
>>
DF    2372
MF    2028
FW    2021
GK     531
```



# 2. 데이터 전처리

## 2) 중복 데이터 정리 및 불필요한 변수 삭제

```
>>
Defender - Centre-Back                1248
Forward - Centre-Forward               666
Midfielder - Central Midfield          659
Defender - Right-Back                  593
Goalkeeper                             531
Defender - Left-Back                   531
Midfielder - Defensive Midfield        366
Forward - Right Winger                 337
attack - Centre-Forward                324
midfield - Central Midfield            313
Forward - Left Winger                  309
Midfielder - Attacking Midfield        255
midfield - Defensive Midfield          183
attack - Left Winger                   147
attack - Right Winger                  142
midfield - Attacking Midfield          112
Forward - Second Striker                62
Midfielder - Left Midfield              55
Midfielder - Right Midfield             41
attack - Second Striker                 34
midfield - Right Midfield               22
midfield - Left Midfield                21
Central Midfield                         1
```

4개의 포지션, 13개의 상세포지션으로 정리

```
>>
Defender - Centre-Back                1248
Forward - Centre-Forward               990
Midfielder - Central Midfield          973
Defender - Right-Back                  593
Midfielder - Defensive Midfield        549
Defender - Left-Back                   531
Goalkeeper                             531
Forward - Right Winger                 479
Forward - Left Winger                  456
Midfielder - Attacking Midfield        367
Forward - Second Striker                96
Midfielder - Left Midfield              76
Midfielder - Right Midfield             63
```

# 2. 데이터 전처리

## 2) 중복 데이터 정리 및불필요한 변수 삭제

관중수(Attendance)와 Column1(의미 미상) 변수 삭제

이름 뒤에 알파벳 'm'이 붙어 반복된 변수들을 삭제

변수의 순서 재정리 (범주형, 수치형 순)

```
# Delete the columns whose names with 'm' on the back
df1.drop(columns=list(df1)[197:380], axis=1, inplace=True)

# Delete the column 'Attendance'
df1.drop(columns=['Attendance'], axis=1, inplace=True)

# Organizing the order of columns
df1_1 = df1.columns[:11].tolist()
df1_2 = df1.columns[11:-4].tolist()
df1_3 = df1.columns[-4:].tolist()
col = df1_1 + df1_3 + df1_2
df1 = df1[col]
df1_1 = df1.columns[:5].tolist()
df1_2 = df1.columns[5:8].tolist()
df1_3 = df1.columns[8:15].tolist()
df1_4 = df1.columns[15:].tolist()
col = df1_1 + df1_3 + df1_2 + df1_4
df1 = df1[col]

# Delete the column 'Column1'
df1.drop(columns=['Column1'], axis=1, inplace=True)
```

# 2. 데이터 전처리

## 3) 다중열 라벨 인코딩 및 데이터분할

7개의 범주형 변수를 라벨 인코딩 과정을 통해

문자열 값을 숫자형 카테고리 값으로 변환

인코딩 대상 변수 :

국적(nationality), 소속팀(squad), 포지션(position), 상세포지션(position2), 주발(foot), league(소속리그), 시즌(Season)

```
# Multiple columns label encoding
pip install -U scikit-learn scipy matplotlib
from sklearn.preprocessing import LabelEncoder
from collections import defaultdict
class MultiColLabelEncoder:
    def __init__(self):
        self.encoder_dict = defaultdict(LabelEncoder)
    def fit_transform(self, X: pd.DataFrame, columns: list):
        if not isinstance(columns, list):
            columns = [columns]
        output = X.copy()
        output[columns] = X[columns].apply(lambda x: self.encoder_dict[x.name].fit_transfo

        return output

    def inverse_transform(self, X: pd.DataFrame, columns: list):
        if not isinstance(columns, list):
            columns = [columns]
        if not all(key in self.encoder_dict for key in columns):
            raise KeyError(f'At least one of {columns} is not encoded before')
        output = X.copy()
        try:
            output[columns] = X[columns].apply(lambda x: self.encoder_dict[x.name].inverse_
        except ValueError:
            print(f'Need assignment when do "fit_transform" function')
            raise
        return output
from collections import defaultdict
class MultiColLabelEncoder:
    def __init__(self):
        self.encoder_dict = defaultdict(LabelEncoder)
    def fit_transform(self, X: pd.DataFrame, columns: list):
        if not isinstance(columns, list):
            columns = [columns]
        output = X.copy()
        output[columns] = X[columns].apply(lambda x: self.encoder_dict[x.name].fit_transfo

        return output

    def inverse_transform(self, X: pd.DataFrame, columns: list):
        if not isinstance(columns, list):
            columns = [columns]
        if not all(key in self.encoder_dict for key in columns):
            raise KeyError(f'At least one of {columns} is not encoded before')
        output = X.copy()
        try:
            output[columns] = X[columns].apply(lambda x: self.encoder_dict[x.name].inverse_
        except ValueError:
            print(f'Need assignment when do "fit_transform" function')
            raise
        return output

mcle = MultiColLabelEncoder()
df2 = mcle.fit_transform(df1, columns=['nationality','position','squad','position2','foot'
inverse_df2 = mcle.inverse_transform(df2, columns=['nationality','position','squad','posit
```

# 2. 데이터 전처리

**3)** 다중열 라벨 인코딩 및 데이터분할

>> 라벨 인코딩 결과

| | player | nationality | position | squad | position2 | foot | league | CL | WinCL | CLBestScorer | .. |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | Burgui | 41 | 1 | 0 | 4 | 2 | 1 | 0.0 | 0.0 | 0.0 | .. |
| **1** | Raphaël Varane | 43 | 0 | 96 | 0 | 2 | 1 | 1.0 | 1.0 | 0.0 | .. |
| **2** | Rubén Duarte | 41 | 0 | 0 | 1 | 1 | 1 | 0.0 | 0.0 | 0.0 | .. |
| **3** | Samuel Umtiti | 43 | 0 | 11 | 0 | 1 | 1 | 1.0 | 0.0 | 0.0 | .. |
| **4** | Manu García | 41 | 3 | 0 | 10 | 1 | 1 | 0.0 | 0.0 | 0.0 | .. |

# 2. 데이터 전처리

## 3) 다중열 라벨 인코딩 및 데이터분할

골키퍼에 대한 변수는 이름 뒤에 '_gk'가 붙어 따로 생성되어 있는 것을 파악

→ 필드플레이어(fp)와 골키퍼(gk)의 데이터프레임을 분할하여 필드플레이어(fp)의 예측모델과 골키퍼(gk)의 예측모델을 따로 만들기로 결정

```
# Split df into field players and goalkeepers
c1 = df2.columns[:14].tolist()
c2 = df2.columns[-55:].tolist()
gkcol = c1 + c2
df2_gk = df2[gkcol]
gks = df2_gk['position'] == 2
df2_gk = df2_gk.loc[gks, :]

c3 = df2.columns[:-55].tolist()
c4 = df2.columns[-13:].tolist()
fpcol = c3 + c4
df2_fp = df2[fpcol]
fps = df2_fp['position'] != 2
df2_fp = df2_fp.loc[fps, :]

# Reset each index
df2_fp.reset_index(drop=True, inplace=True)
df2_gk.reset_index(drop=True, inplace=True)
```

# 2. 데이터 전처리

**3)** 다중열 라벨 인코딩 및 데이터분할

필드플레이어(fp) 데이터프레임

| | player | nationality | position | squad | position2 | foot | league | CL | WinCL | CLBestScorer | Season | age | value | height | games | games_starts |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | Burgui | 41 | 1 | 0 | 4 | 2 | 1 | 0.0 | 0.0 | 0.0 | 0 | 23.0 | 1800000.0 | 186.0 | 23.0 | 12.0 |
| 1 | Raphaël Varane | 43 | 0 | 96 | 0 | 2 | 1 | 1.0 | 1.0 | 0.0 | 0 | 24.0 | 70000000.0 | 191.0 | 27.0 | 27.0 |
| 2 | Rubén Duarte | 41 | 0 | 0 | 1 | 1 | 1 | 0.0 | 0.0 | 0.0 | 0 | 21.0 | 2000000.0 | 179.0 | 24.0 | 24.0 |
| 3 | Samuel Umtiti | 43 | 0 | 11 | 0 | 1 | 1 | 1.0 | 0.0 | 0.0 | 0 | 23.0 | 60000000.0 | 182.0 | 25.0 | 24.0 |

골키퍼(gk) 데이터프레임

| | player | nationality | position | squad | position2 | foot | league | CL | WinCL | CLBestScorer | Season | age | value | height | games_gk | games_s |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | Fernando Pacheco | 41 | 2 | 0 | 7 | 1 | 1 | 0.0 | 0.0 | 0.0 | 0 | 25.0 | 9000000.0 | 185.0 | 38.0 | |
| 1 | Antonio Sivera | 41 | 2 | 0 | 7 | 2 | 1 | 0.0 | 0.0 | 0.0 | 0 | 20.0 | 1000000.0 | 185.0 | 1.0 | |
| 2 | Jean-Christophe Bouet | 43 | 2 | 2 | 7 | 2 | 2 | 0.0 | 0.0 | 0.0 | 0 | 34.0 | 20000.0 | 184.0 | 1.0 | |
| 3 | Régis Gurtner | 43 | 2 | 2 | 7 | 2 | 2 | 0.0 | 0.0 | 0.0 | 0 | 30.0 | 3000000.0 | 182.0 | 37.0 | |

# 2. 데이터 전처리

## 4) 변수선택법 및 분산팽창인자

```
# Stepwise variables selection of field players data

# Delete columns for response and categorical variables
df0_fp = df2_fp.drop(['value','player','nationality','position','squad','position2','foot'

# Explanatory variables and response variable
variables = df0_fp.columns.tolist()
y = df2_fp['value']
selected_variables = []
# Significance level
sl_enter = 0.05
sl_remove = 0.05
sv_per_step = []
adjusted_r_squared = []
steps = []
step = 0

while len(variables) > 0:
    remainder = list(set(variables) - set(selected_variables))
    pval = pd.Series(index=remainder) ## p-value
    ## 기존에 포함된 변수와 새로운 변수 하나씩 돌아가면서
    ## 선형 모형을 적합한다.
    for col in remainder:
        X = df0_fp[selected_variables+[col]]
        X = sm.add_constant(X)
        model = sm.OLS(y,X).fit()
        pval[col] = model.pvalues[col]

    min_pval = pval.min()
    if min_pval < sl_enter: ## 최소 p-value 값이 기준 값보다 작으면 포함
        selected_variables.append(pval.idxmin())
        ## 선택된 변수들에대해서
        ## 어떤 변수를 제거할지 고른다.
        while len(selected_variables) > 0:
            selected_X = df0_fp[selected_variables]
            selected_X = sm.add_constant(selected_X)
            selected_pval = sm.OLS(y,selected_X).fit().pvalues[1:] ## 절편항의 p-value는 뺀
            max_pval = selected_pval.max()
            if max_pval >= sl_remove: ## 최대 p-value값이 기준값보다 크거나 같으면 제외
                remove_variable = selected_pval.idxmax()
                selected_variables.remove(remove_variable)
            else:
                break

        step += 1
        steps.append(step)
        adj_r_squared = sm.OLS(y,sm.add_constant(df0_fp[selected_variables])).fit().rsquare
        adjusted_r_squared.append(adj_r_squared)
        sv_per_step.append(selected_variables.copy())
    else:
        break
```

데이터를 정리했음에도 불구하고 예측 모델링을 진행하기에 변수가 너무 많음 (200개 이상)

**회귀분석**, **조사자료분석** 과목에서 배웠던 **변수선택법**을 적용해보고자 함.

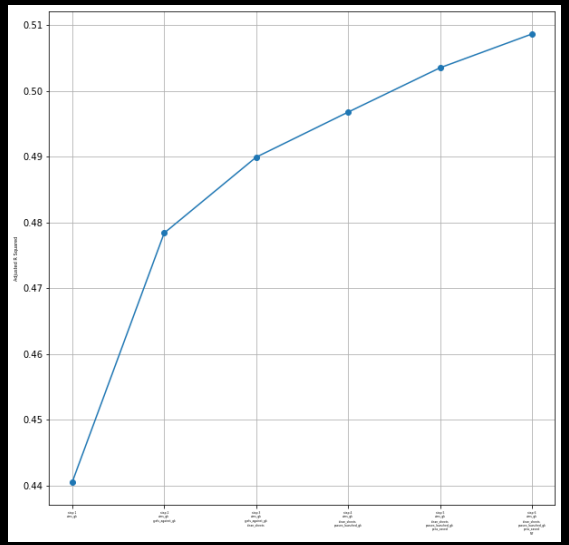
여러 변수선택법의 방법 중 전진선택법, 후진선택법은 한번 선택되거나 제거되면 다시 제거 및 포함이 어렵다는 단점 분명

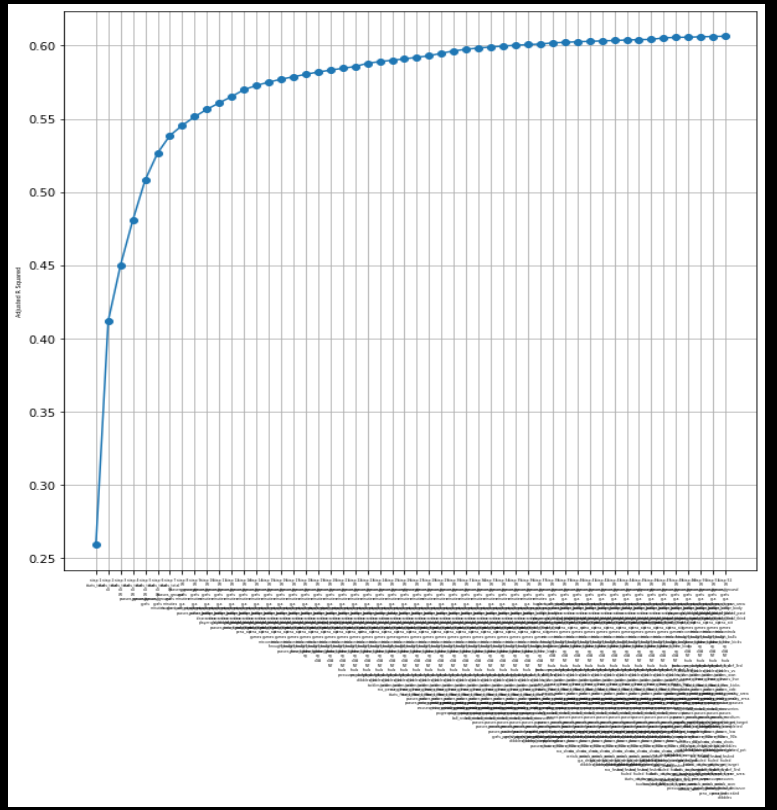
따라서 **단계별 선택법**을 통해 단점을 보완하면서 변수를 최대한 줄여보고자 함.

# 2. 데이터 전처리

## 4) 변수선택법 및 분산팽창인자

단계별선택법의 시각화 결과



# 2. 데이터 전처리

## 4) 변수선택법 및 분산팽창인자

단계별 선택법 적용 결과, 필드플레이어의 변수는 45개, 골키퍼의 변수는 5개가 산출됨.

다중공선성의 제거를 위해 각각 데이터에 대해 **회귀분석**에서 배운 **분산팽창인자(VIF)**를 계산하여 10 이상인 변수는 모두 제거

골키퍼 데이터의 경우 5개의 변수 중 승리(wins_gk)와 클린시트(clean_sheets) 두 변수만이 높은 VIF 값을 보임.
두 변수 간의 상관관계가 높은 것을 파악하고 둘 중 승리 변수만 삭제하는 것으로 진행

```
# Number of selected variables
len_stepwise_fp = len(selected_variables)
len_stepwise_fp
```

```
>> 45
```

```
# List of selected variables
stepwise_variables_fp = selected_variables
stepwise_variables_fp
```

```
>>
['sca_passes_live', 'goals', 'passes_ground', 'W', 'touches_att_pen_area',
 'passes_into_penalty_area', 'crosses_into_penalty_area', 'passes_other_body', 'pens_att',
 'games', 'gca', 'passes_total_distance', 'players_dribbled_past', 'miscontrols', 'xGA',
 'passes_free_kicks', 'passes_completed_medium', 'xg_net', 'MP', 'aerials_won',
 'pressures_def_3rd', 'dribbles_vs', 'passes_into_final_third', 'progressive_passes',
 'fouls', 'tackles_won', 'shots_free_kicks', 'passes_oob', 'ball_recoveries',
 'through_balls', 'dribbles_completed', 'pressures', 'goals_per_shot_on_target',
 'sca_shots', 'gca_dribbles', 'touches_def_3rd', 'touches_def_pen_area', 'passes',
 'sca_fouled', 'dribbles_completed_pct', 'pens_conceded', 'shots_on_target',
 'minutes_90s', 'passes_low', 'fouled']
```

```
# Vif for field players data
vif_fp = df2_fp[stepwise_variables_fp]
from statsmodels.stats.outliers_influence import variance_inflation_factor
vif = pd.DataFrame()
vif["VIF Factor"] = [variance_inflation_factor(vif_fp.values, i) for i in range(vif_fp.sha
vif["features"] = vif_fp.columns
vif = vif.sort_values("VIF Factor").reset_index(drop=True)
vif
```

| | VIF Factor | features |
|---|---|---|
| 0 | 1.485017 | pens_conceded |
| 1 | 2.324860 | gca_dribbles |
| 2 | 2.393209 | shots_free_kicks |
| 3 | 2.467419 | goals_per_shot_on_target |
| 4 | 2.708104 | pens_att |
| 5 | 3.156319 | through_balls |
| 6 | 3.414008 | xg_net |
| 7 | 3.890012 | passes_other_body |
| 8 | 4.382470 | sca_shots |
| 9 | 4.897236 | sca_fouled |
| 10 | 5.378947 | aerials_won |
| 11 | 5.445274 | passes_free_kicks |
| 12 | 6.658303 | crosses_into_penalty_area |
| 13 | 7.083557 | dribbles_completed_pct |
| 14 | 9.461171 | fouled |
| 15 | 9.608095 | gca |
| 16 | 10.948917 | passes_oob |
| 17 | 10.962983 | fouls |
| 18 | 16.031411 | W |
| 19 | 16.516598 | miscontrols |
| 20 | 17.042119 | passes_low |
| 21 | 18.303770 | tackles_won |
| 22 | 18.744143 | dribbles_vs |
| 23 | 19.962341 | shots_on_target |
| 24 | 23.471256 | touches_att_pen_area |
| 25 | 26.191997 | passes_into_penalty_area |
| 26 | 26.996103 | pressures_def_3rd |
| 27 | 27.314308 | sca_passes_live |
| 28 | 28.576897 | goals |
| 29 | 29.390900 | touches_def_pen_area |
| 30 | 33.073532 | ball_recoveries |
| 31 | 37.726270 | games |
| 32 | 39.757513 | pressures |
| 33 | 40.291718 | passes_into_final_third |
| 34 | 52.194814 | xGA |
| 35 | 58.381406 | progressive_passes |
| 36 | 72.995581 | touches_def_3rd |
| 37 | 76.219890 | passes_ground |
| 38 | 93.168869 | minutes_90s |
| 39 | 103.007268 | MP |
| 40 | 183.758501 | passes_completed_medium |
| 41 | 266.805617 | passes_total_distance |
| 42 | 300.066155 | passes |
| 43 | 423.896515 | dribbles_completed |
| 44 | 437.811073 | players_dribbled_past |

**분산팽창인자(VIF)**
계산 결과

# 2. 데이터 전처리

## 5) 주성분분석

앞선 과정의 결과로 골키퍼 데이터의 독립변수는 15개,
필드플레이어 데이터의 독립변수는 27개가 남음.

따라서 **다변량자료분석** 과목에서 배웠던 주성분분석(PCA) 과정을 통해 독립변수의 차원을 축소시켜보고자 함.

```
df3_fp_pca = df3_fp.iloc[:, 12:]

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
result = scaler.fit_transform(df3_fp_pca)
data_scaled = pd.DataFrame(result)
data_scaled.describe()

#pca를 위한 데이터스케일링

from sklearn.decomposition import PCA

pca = PCA()
pca.fit(data_scaled)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum>=0.90)+1

print(d)
```

```
11
```

```
pca = PCA(n_components=5)
X_redufced = pca.fit_transform(data_scaled)
result = pd.DataFrame(X_redufced)
result
```

```
df4_gk = df3_gk
df4_fp = df3_fp.iloc[:,0:15]
df4_fp = pd.concat([df4_fp, result], axis=1)
df4_fp.head()
```

# 2. 데이터 전처리

## 5) 주성분분석

분산이 90%가 유지되기 위해서는 11개로 차원 축소를 하는 것이 적당하지만,
이후 분석을 위해서 우선 10개의 변수로 차원축소를 진행함.

| | nationality | position | squad | foot | league | CL | WinCL | CLBestScorer | Season | age | value | height | 0 | 1 | 2 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 41 | 1 | 0 | 2 | 1 | 0.0 | 0.0 | 0.0 | 0 | 23.0 | 1800000.0 | 186.0 | -0.114021 | 0.794820 | 0.048132 | 0.200 |
| **1** | 43 | 0 | 96 | 2 | 1 | 1.0 | 1.0 | 0.0 | 0 | 24.0 | 70000000.0 | 191.0 | -0.551850 | -0.853237 | -0.716878 | -0.571 |
| **2** | 41 | 0 | 0 | 1 | 1 | 0.0 | 0.0 | 0.0 | 0 | 21.0 | 2000000.0 | 179.0 | 0.315503 | -1.940694 | -1.050466 | 0.383 |
| **3** | 43 | 0 | 11 | 1 | 1 | 1.0 | 0.0 | 0.0 | 0 | 23.0 | 60000000.0 | 182.0 | -0.388489 | -1.180292 | 0.252729 | -0.673 |
| **4** | 41 | 3 | 0 | 1 | 1 | 0.0 | 0.0 | 0.0 | 0 | 31.0 | 1800000.0 | 183.0 | 1.912225 | -0.549556 | 0.277757 | -1.349 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | |
| **6416** | 89 | 1 | 128 | 2 | 3 | 0.0 | 0.0 | 0.0 | 2 | 23.0 | 12000000.0 | 165.0 | -0.827285 | 1.094987 | 0.273594 | -0.178 |
| **6417** | 69 | 0 | 128 | 1 | 3 | 0.0 | 0.0 | 0.0 | 2 | 29.0 | 4000000.0 | 188.0 | 0.516999 | -2.980781 | 0.190652 | -1.039 |
| **6418** | 41 | 1 | 128 | 2 | 3 | 0.0 | 0.0 | 0.0 | 2 | 23.0 | 25000000.0 | 178.0 | 4.581098 | 0.462026 | -0.456580 | -0.825 |
| **6419** | 41 | 0 | 128 | 2 | 3 | 0.0 | 0.0 | 0.0 | 2 | 22.0 | 6000000.0 | 184.0 | -2.056018 | 0.202595 | -0.099135 | 0.263 |
| **6420** | 89 | 0 | 128 | 1 | 3 | 0.0 | 0.0 | 0.0 | 2 | 20.0 | 9000000.0 | 174.0 | -1.750758 | 0.668921 | -0.646665 | 0.116 |

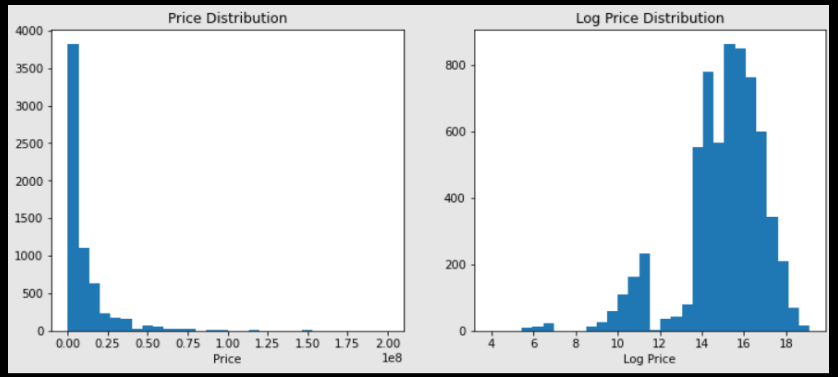
6421 rows × 22 columns

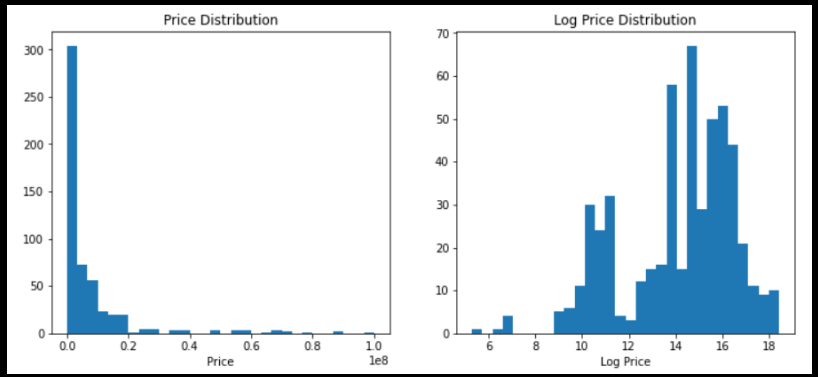
# 2. 데이터 전처리

## 6) 로그변환 및 정규화

반응 변수인 **value 변수**의 분포를 확인해본 결과, 왼쪽으로 치우쳐진 것을 파악
**회귀분석** 등 과목에서 배웠던 로그변환을 통해 정규화
('log_value' 변수로 저장)

골키퍼 데이터프레임의 value 변수 또한 마찬가지로 진행



# 03

# 데이터 모델링

Data Modeling

# 3. 데이터 모델링

## 1) 독립변수와 종속변수 구분

종속변수(반응변수) : log_value

독립변수(설명변수) : 나머지 변수

필드플레이어와 골키퍼의 데이터를 각각 독립변수와 종속변수로 구분

```
# Split the data into explanatory variables and response variable
df_X = df4_fp.drop('log_value', axis=1)
df_y = pd.DataFrame(df4_fp['log_value'])

df_gk_X = df3_gk.drop('log_value', axis=1)
df_gk_y = pd.DataFrame(df3_gk['log_value'])
```

# 3. 데이터 모델링

## 2) 훈련세트와 테스트세트로 분할

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(df_X, df_y, test_size=0.2, random_state
X_train_gk, X_test_gk, y_train_gk, y_test_gk = train_test_split(df_gk_X, df_gk_y, test_size
```

```
# Field players
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
# Goalkeepers
print(X_train_gk.shape)
print(X_test_gk.shape)
print(y_train_gk.shape)
print(y_test_gk.shape)
```

```
>>
(5136, 21)
(1285, 21)
(5136, 1)
(1285, 1)
(424, 15)
(107, 15)
(424, 1)
(107, 1)
```

각각의 데이터의 훈련세트(train set)와 테스트세트(test set)를 8:2의 크기로 분할

# 3. 데이터 모델링

## 3) 모델링

모델링 과정을 진행 하기 위해 **회귀분석** 시간에서 배운 LinearRegression, Ridge, Lasso를 이용,

그리고 elasticnet, ardr_linear, baysian_ridge, random_forest, xgboost_linear 등의 회귀 모델을 이용하여 모델링 과정을 진행

이에 대해 회귀 모델의 평가 측도로 사용되는 RMSE 값을 도출하여 값이 가장 낮은 (설명력이 좋은) 모델을 선정할 예정

```python
# Modeling
def my_regressor(df_X, df_y):
  from sklearn.linear_model import LinearRegression
  from sklearn.linear_model import Ridge, Lasso, ElasticNet
  from sklearn.linear_model import ARDRegression, BayesianRidge
  from sklearn.ensemble import RandomForestRegressor
  from xgboost import XGBRegressor
  from sklearn.model_selection import cross_val_score

  import ast

  linear = LinearRegression()
  ridge, lasso, elasticnet = Ridge(), Lasso(), ElasticNet()
  ardr_linear, baysian_ridge = ARDRegression(), BayesianRidge()
  random_forest = RandomForestRegressor()
  xgboost_linear = XGBRegressor()

  my_model_list = ['linear','ridge','lasso','elasticnet','ardr_linear','baysian_ridge','ra

  score_dic = dict()

  for model_nm in my_model_list:
    scores = cross_val_score(eval(model_nm), df_X, df_y, scoring = "neg_mean_squared_error
    rmse_score = np.sqrt(-scores)
    rmse_sm = rmse_score.mean()
    score_dic[model_nm] = rmse_sm

  score_dic = sorted(score_dic.items(), key=lambda t : t[1])

  return score_dic

linear_model_train_score = my_regressor(X_train, y_train)
linear_model_train_score_gk = my_regressor(X_train_gk, y_train_gk)
print(linear_model_train_score)
print(linear_model_train_score_gk)
```

# 3. 데이터 모델링

## 3) 모델링

모델링 결과 각 모델종류의 RMSE 값 (필드플레이어와 골키퍼 데이터)

필드플레이어

```
[('xgboost_linear', 1.2895178088268244), ('random_forest', 1.3521977282864572),
 ('baysian_ridge', 1.6005872242015822), ('ridge', 1.6006124799423531),
 ('linear', 1.6009018485469), ('ardr_linear', 1.6016165484135283),
 ('elasticnet', 1.7782286495882063), ('lasso', 1.8280869176495664)]
```

골키퍼

```
[('xgboost_linear', 1.4812884415856096), ('random_forest', 1.5124788404870262),
 ('ridge', 1.8115938757547672), ('linear', 1.8139077636950767),
 ('ardr_linear', 1.8204328508446495), ('baysian_ridge', 1.8287638730970905),
 ('elasticnet', 1.8495212834872539), ('lasso', 1.8694847754098567)]
```

XGBoost_linear가 가장 성능이 좋은 것으로 확인

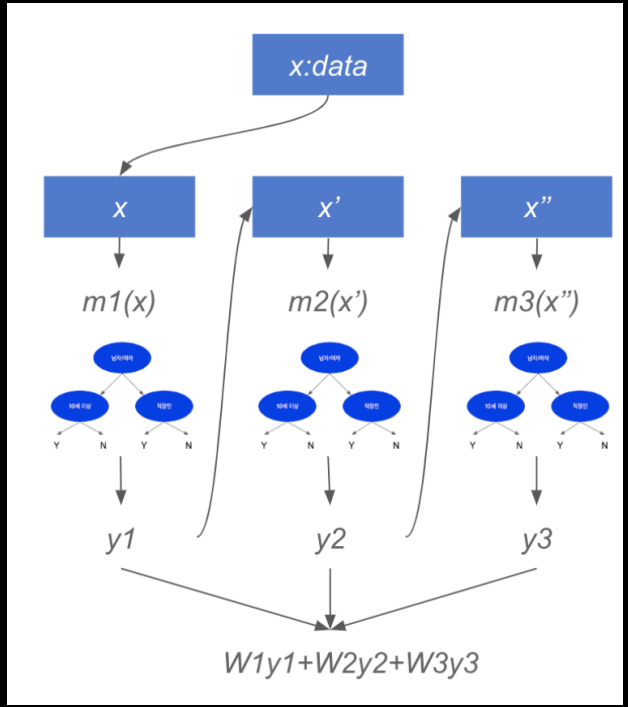
# 3. 데이터 모델링

## 3) 모델링

XGBoost 알고리즘 : 여러 개의 결정 트리(Decision Tree)를 조합해서 사용하는 앙상블 알고리즘

앙상블 중 Gradient Boost의 경우 계속적으로 새로운 모델을 학습시키는데, 예측이 잘못된 샘플에 가중치를 반영하여 다음 모델에 반영시켜서 성능을 개선함

XGBoost는 이러한 Gradient Boost 알고리즘을 병렬 학습이 지원되도록 구현한 라이브러리



# 3. 데이터 모델링

## 4) 하이퍼 파라미터 튜닝


예측의 정확도를 높이고 Loss를 최소화하기 위해 하이퍼 파라미터 튜닝 과정에서 Optuna를 이용

Optuna는 하이퍼파라미터 최적화를 도와주는 프레임 워크

파라미터의 범위를 지정해주거나, 파라미터가 될 수 있는 목록을 설정하면 매 Trial마다 최적의 파라미터를 찾는 용도

필드플레이어와 골키퍼 데이터 각각 100회 정도 과정을 실행

# 3. 데이터 모델링

## 4) 하이퍼 파라미터 튜닝 - 필드플레이어

```
# Hyper-parameter tuning for Field players
import xgboost as xgb
from sklearn.model_selection import KFold
import optuna
from optuna import Trial
from optuna.samplers import TPESampler
from xgboost import XGBRegressor
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error

def objective(trial):

    kf = KFold(n_splits=10, shuffle=True)

    param = {
        'lambda': trial.suggest_loguniform('lambda', 1e-3, 10.0),
        'alpha': trial.suggest_loguniform('alpha', 1e-3, 10.0),
        'colsample_bytree': trial.suggest_categorical('colsample_bytree', [0.3,0.4,0.5,0.6,0.7,0.8,0.9, 1.0]),
        'subsample': trial.suggest_categorical('subsample', [0.4,0.5,0.6,0.7,0.8,1.0]),
        'learning_rate': trial.suggest_categorical('learning_rate', [0.008,0.009,0.01,0.012,0.014,0.016,0.018, 0.02]),
        'n_estimators': 4000,
        'max_depth': trial.suggest_categorical('max_depth', [5,7,9,11,13,15,17,20]),
        'random_state': trial.suggest_categorical('random_state', [24, 48,2020]),
        'min_child_weight': trial.suggest_int('min_child_weight', 1, 300),
        'folds' : kf
    }
    model = xgb.XGBRegressor(**param)

    model.fit(X_train,y_train,eval_set=[(X_train, y_train),(X_test,y_test)],early_stopping_rounds=100,verbose=False)

    preds = model.predict(X_test)

    rmse = mean_squared_error(y_test, preds,squared=False)

    return rmse

study = optuna.create_study(direction='minimize')
study.optimize(objective, n_trials=100)
print('Number of finished trials:', len(study.trials))
print('Best trial:', study.best_trial.params)
```

# 3. 데이터 모델링

## 4) 하이퍼 파라미터 튜닝 - 필드플레이어

```
trial = study.best_trial
trial_params = trial.params
print('Best Trial: score {},\nparams {}'.format(trial.value, trial_params))
```

```
>> Best Trial: score 1.1819705125825097,
>> params {'lambda': 0.6106899368498404, 'alpha': 0.001558377624059179,
'colsample_bytree': 0.5, 'subsample': 1.0, 'learning_rate': 0.018, 'max_depth': 9,
'random_state': 2020, 'min_child_weight': 10}
```

필드플레이어 데이터에 대한 하이퍼 파라미터 튜닝 결과, 이전 모델링 과정 직후 1.2895였던 RMSE 값이 1.1819 정도로 낮아짐.

# 3. 데이터 모델링

## 4) 하이퍼 파라미터 튜닝 - 골키퍼

```
# Hyper-parameter tuning for goalkeepers
def objective_gk(trial):

    kf = KFold(n_splits=10, shuffle=True)

    param = {
        'lambda': trial.suggest_loguniform('lambda', 1e-3, 10.0),
        'alpha': trial.suggest_loguniform('alpha', 1e-3, 10.0),
        'colsample_bytree': trial.suggest_categorical('colsample_bytree', [0.3,0.4,0.5,0.6,0.7,0.8,0.9, 1.0]),
        'subsample': trial.suggest_categorical('subsample', [0.4,0.5,0.6,0.7,0.8,1.0]),
        'learning_rate': trial.suggest_categorical('learning_rate', [0.008,0.009,0.01,0.012,0.014,0.016,0.018, 0.02]),
        'n_estimators': 4000,
        'max_depth': trial.suggest_categorical('max_depth', [5,7,9,11,13,15,17,20]),
        'random_state': trial.suggest_categorical('random_state', [24, 48,2020]),
        'min_child_weight': trial.suggest_int('min_child_weight', 1, 300),
        'folds' : kf
    }
    model = xgb.XGBRegressor(**param)

    model.fit(X_train_gk,y_train_gk,eval_set=[(X_train_gk, y_train_gk),(X_test_gk,y_test_gk)],early_stopping_rounds=100,verbose=False)

    preds = model.predict(X_test_gk)

    rmse = mean_squared_error(y_test_gk, preds,squared=False)

    return rmse

study_gk = optuna.create_study(direction='minimize')
study_gk.optimize(objective_gk, n_trials=100)
print('Number of finished trials:', len(study_gk.trials))
print('Best trial:', study_gk.best_trial.params)
```

# 3. 데이터 모델링

## 4) 하이퍼 파라미터 튜닝 - 골키퍼

```
trial_gk = study_gk.best_trial
trial_params_gk = trial_gk.params
print('Best Trial: score {},\nparams {}'.format(trial_gk.value, trial_params_gk))
```

```
>> Best Trial: score 1.3775470652078008,
>> params {'lambda': 0.2908119670332509, 'alpha': 0.2630126371564937,
'colsample_bytree': 0.4, 'subsample': 1.0, 'learning_rate': 0.014,
'max_depth': 13, 'random_state': 2020, 'min_child_weight': 24}
```

골키퍼 데이터에 대한 하이퍼 파라미터 튜닝 결과, 이전 모델링 과정 직후 1.4812이였던 RMSE 값이 1.3775 정도로 낮아짐.

# 3. 데이터 모델링

## 5) 모델 학습

```
xgb_model = xgb.XGBRegressor(**trial_params)
xgb_model.fit(X_train, y_train)
y_pred = xgb_model.predict(X_test)
```

완성된 모델을 학습(fit)시킴

# 04

# 모델 평가

Modeling Evaluation

# 4. 모델 평가

## 1) RMSE

- 필드 플레이어

```
rmse = mean_squared_error(y_test, y_pred, squared=False)
print(rmse)
```

```
3.8877277300486512
```

- 골키퍼

```
rmse_gk = mean_squared_error(y_test_gk, y_pred_gk, squared=False)
print(rmse_gk)
```

```
2.5174312830721974
```

필드 플레이어, 골키퍼 모두
훈련 세트로 모델을
훈련시켰을 때에 비해
RMSE값이 증가하였음.

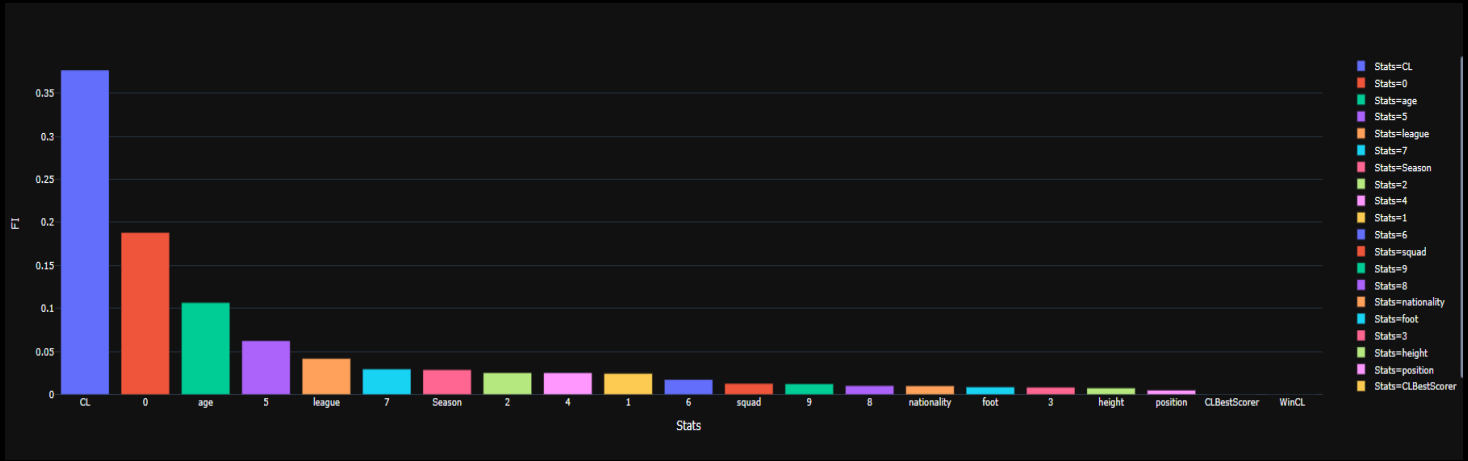
과적합 방지를 위해 모델링
과정에서 교차검증을
시행하였으나, 어느정도
과적합이 발생하였음

그럼에도 어느정도 유의미한
결과가 도출되었다고 판단

# 4. 모델 평가

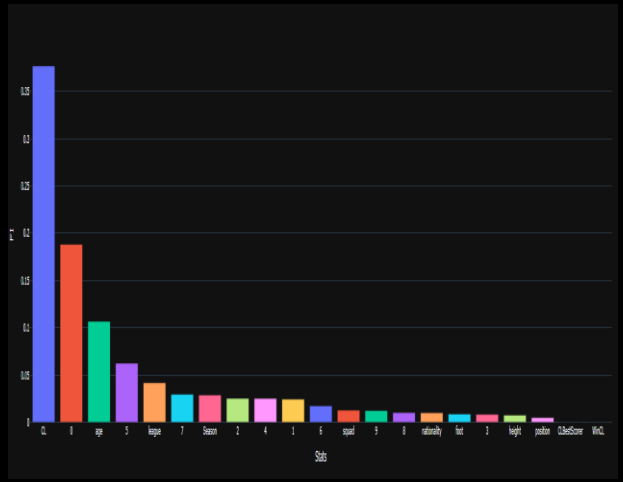
## 2) 변수 중요도 탐색 - 필드플레이어

```
# Feature importance for field players
X_train = pd.DataFrame(X_train, columns = df_X.columns)
X_test = pd.DataFrame(X_test, columns=df_X.columns)

d={"Stats" : X_train.columns , "FI" : xgb_model.feature_importances_}
df = pd.DataFrame(d)
df = df.sort_values(by='FI', ascending=0)

import plotly.express as px
fig = px.bar(df, x='Stats', y='FI', color="Stats", template="plotly_dark")
fig.show()
```



# 4. 모델 평가

## 2) 변수 중요도 탐색 - 필드플레이어



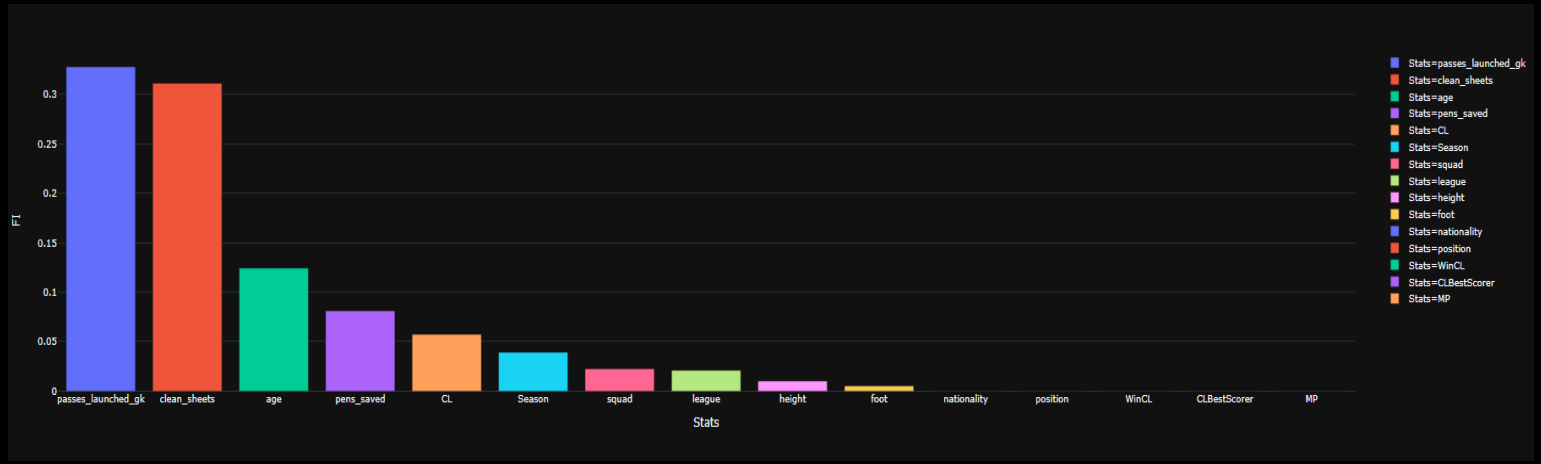

필드플레이어 데이터 중에서는
CL(챔피언스리그 출전 여부),
age(나이)
league(소속 리그)
등의 변수가 이적료 측정에 있어 중요한 영향을 미치는 변수임을 파악

PCA 과정에서 나온 PC0의 중요도가 높다는 부분에서
정확히 특정한 변수의 중요도를 확인하기 힘든 한계점 파악

# 4. 모델 평가

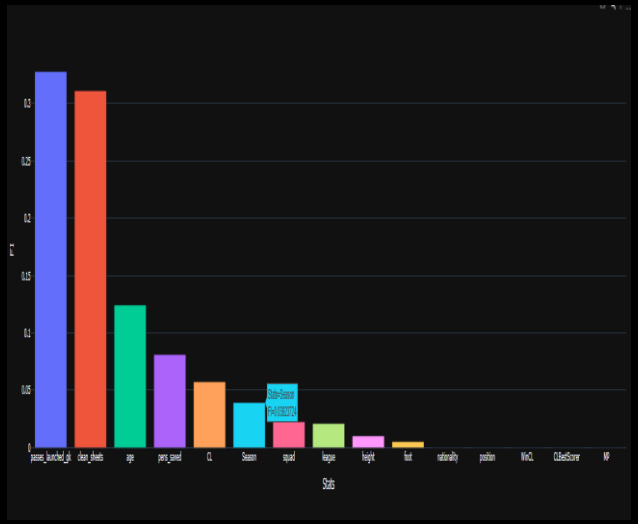
## 2) 변수 중요도 탐색 - 골키퍼

```
# Feature importance for field players
X_train_gk = pd.DataFrame(X_train_gk, columns = X_train_gk.columns)
X_test_gk = pd.DataFrame(X_test_gk, columns=X_test_gk.columns)

d={"Stats" : X_train_gk.columns , "FI" : xgb_model_gk.feature_importances_}
df_gk = pd.DataFrame(d)
df_gk = df_gk.sort_values(by='FI', ascending=0)

fig = px.bar(df_gk, x='Stats', y='FI', color="Stats", template="plotly_dark")
fig.show()
```



# 4. 모델 평가

## 2) 변수 중요도 탐색 - 골키퍼




골키퍼 데이터 중에서는
passes_launched_gk(시작 패스 횟수)
clean_sheets(클린시트)
age(나이) 와 같은 변수가 골키퍼 포지션의 이적료 측정에 있어 중요한 영향을 미치는 변수임을 파악

# 한계점

- 모델링 전 MinMaxScaler, StandardScaler등을 활용한 데이터 스케일링을 통해 변수에 대한 정규화를 진행하여 RMSE 값을 더욱 낮추기 과정을 시도하였고, 0.04까지 값을 줄이는데 성공함. 하지만 스케일링 된 데이터를 원상복구시키는 과정 진행이 잘 되지 않았기에 스케일링 과정을 제외하고 분석을 진행하였음.
이 과정에서 RMSE 값이 이전보다 높아진 한계점이 발생

- 원시 데이터의 변수가 너무 많아 변수선택법과 PCA 등 변수를 줄이는 과정에서, PCA를 통해 차원축소된 변수들이 이후 모델 평가에서 어떤 변수가 무엇을 의미하는지 알아내기 어려운 점을 파악함