STAT 4243 Project 5 (Sports Analytics)

By: Louis Cheng (yc3733), Dhruv Limaye (djl2187), Sangmin Lee (sl4876), Judy Wu (dw2936)

Introduction

Billions of dollars are being spent every year by football clubs to buy and sign players, with some clubs spending well over 300 million euros/per year. One would anticipate that the team who shells out the most money yearly would be the strongest in Europe, but this is not always reflected in reality. For example, PSG spent $250+ million in 2017 on just one player (Neymar Jr.), yet PSG has never won the UEFA Champions League (UCL).

UCL is the most competitive football competition in Europe, where the best teams of each European football league compete against each other in this continental, knock-out competition. Usually, the top 4 teams of Europe's 5 Biggest League (English Premier League, Spanish La Liga, Italian Serie A, French Ligue 1, and German Bundesliga) will qualify for UCL.

All teams aspire to qualify for the UCL, as the television rights and sponsorship revenue earned from appearing in the tournament dwarf those of their respective domestic leagues. It's worth mentioning that UCL revenue only goes up with every stage the team advances. In other words, qualifying for the UCL will bring in the most money and provide the best return on investment from club expenditure.

Goal

To that end, the goal of the project is to find out what is the cheapest team that can qualify for Champions League. We will use linear optimization and feature selection to achieve this goal.

In [1]:

```
import pandas as pd
import scipy.stats as stats
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import math
import statistics

from sklearn.model_selection import train_test_split
from lazypredict.Supervised import LazyRegressor
from sklearn.ensemble import AdaBoostRegressor

from verstack import FeatureSelector
from featurewiz import featurewiz

from gurobipy import Model, GRB, quicksum, max_
```

```
Imported 0.2.03 version. Select nrows to a small number when running on huge
datasets.
output = featurewiz(dataname, target, corr_limit=0.90, verbose=2, sep=',',
                header=0, test_data='',feature_engg='', category_encoders
='',
                dask_xgboost_flag=False, nrows=None, skip_sulov=False)
Create new features via 'feature_engg' flag : ['interactions','groupby','tar
get']
```

Data Processing

To illustrate the idea, data from only one domestic league (Spanish La Liga) was studied. Furthermore, only data from the last 22 seasons was collected because UCL only implemented the top 4 qualification format 22 seasons ago.

The data was webscraped from fbref.com and transfermarkt.com. The webscrapping and cleaning of the data was done in a python script attached to the repo, and the cleaned data was exported to CSV. To that end, we are just reading in the CSV here and merging them as needed.

In [2]:

```
post_2017_df = pd.read_csv('../output/post_2017_fbref_data.csv',encoding= 'unicode_escape')
pre_2017_df = pd.read_csv('../output/pre_2017_fbref_data.csv',encoding= 'unicode_escape').i
xfer_df = pd.read_csv('../output/xfer_data.csv',encoding= 'unicode_escape').iloc[:,1:]

new_col_names = {"year":"Season"}
xfer_df = xfer_df.rename(columns=new_col_names)

player_stat = pd.read_csv('../output/player_stats.csv', encoding= 'unicode_escape').iloc[:,
new_col_names = {"Annual Wages":"Wage"}
player_stat = player_stat.rename(columns=new_col_names)
```

In [3]:

```
pre_2017_df = pre_2017_df.append(post_2017_df[pre_2017_df.columns.to_list()])
pre_2017_df = pd.merge(pre_2017_df,xfer_df,on=['Season','team'])
```

EDA

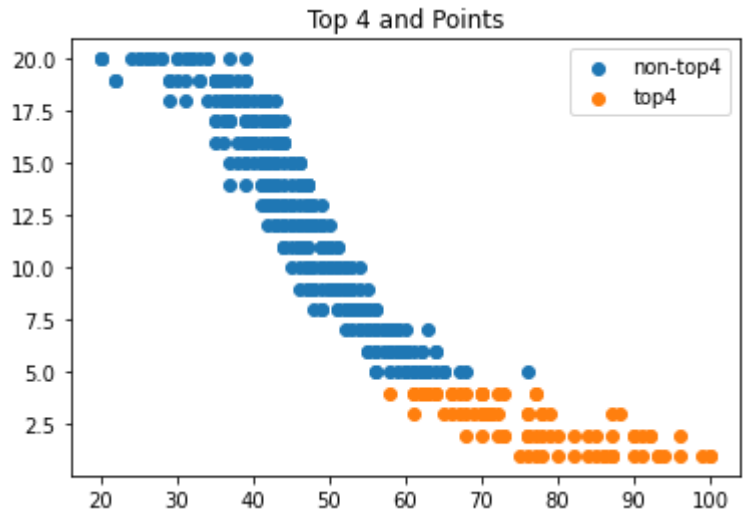
As the top 4 teams of Laliga are automatically qualified to the Champions League. For each season, we are splitting the data set into two groups, top 4 and non-top 4. The EDA aims to determine a reasonable cut-off point for points needed to qualify for the champions league.

In [4]:

```
top_4 = pre_2017_df[pre_2017_df['rank']<=4]
non_top_4 = pre_2017_df[pre_2017_df['rank']>4]

plt.scatter(non_top_4['points'],non_top_4['rank'], label=f'non-top4')
plt.scatter(top_4['points'], top_4['rank'], label=f'top4')

plt.rcParams.update({'figure.figsize':(6.4,4.8), 'figure.dpi':100})
plt.title('Top 4 and Points')
plt.legend()
plt.show()
```



From a quick observation, there have been instances where teams have less than 60 points and still qualified for the Champions League, but there is also a high probability it doesn't if it has less than 60 points. An excellent cut-off point would be between 60-70 points. After testing out a few possible thresholds, we have decided that 66 points are a good threshold. To justify this, a contingency table has been provided below.

In [5]:

```
pre_2017_df['Top4']= ['Top4' if i<=4 else 'Not_Top4' for i in pre_2017_df['rank']]
pre_2017_df['66pts']= ['GE66' if i>=66 else 'LT66' for i in pre_2017_df['points']]
pre_2017_df['log(expenditure)'] = [math.log(i) if i>0 else 0 for i in pre_2017_df['expendit
```

In [6]:

```
data_crosstab = pd.crosstab(pre_2017_df['Top4'],pre_2017_df['66pts'])

data_crosstab
```

Out[6]:

66pts	GE66	LT66
Top4		
Not_Top4	3	349
Top4	76	12

To reinforce the contingency table, a conditional probability was calculated. The probability of being in the top 4, given that a team has 66 points in the last 22 seasons, was 96.2%, which reinforces that 66 points are a good cut-off.

In [7]:

```
len(pre_2017_df[(pre_2017_df['points']>=66) & (pre_2017_df['rank']<=4)])/len(pre_2017_df[pr
```

Out[7]:

```
0.9620253164556962
```

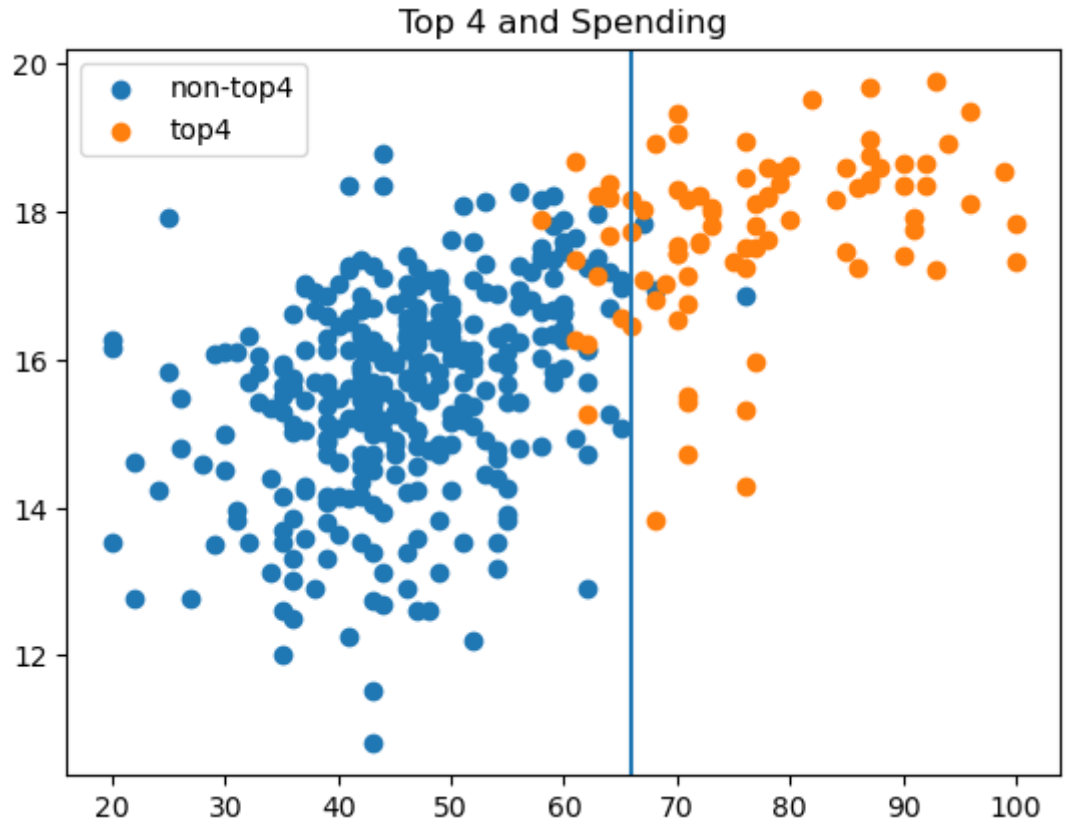
A scatterplot was plotted below to show the relationship between spending and points. As we can see that there is a positive linear relationship between spending and points; the more one spends, the more points a team is likely to gain more than 66 points and hence be in the top 4.

In [8]:

```
top_4 = pre_2017_df[(pre_2017_df['rank']<=4) & (pre_2017_df['expenditure']>0)]
non_top_4 = pre_2017_df[(pre_2017_df['rank']>4) & (pre_2017_df['expenditure']>0)]

plt.scatter(non_top_4['points'], non_top_4['log(expenditure)'], label=f'non-top4')
plt.scatter(top_4['points'], top_4['log(expenditure)'], label=f'top4')
plt.axvline(x=66)

plt.rcParams.update({'figure.figsize':(6.4,4.8), 'figure.dpi':100})
plt.title('Top 4 and Spending')
plt.legend()
plt.show()
```



Feature Selection

As there are not sufficient data points and there is a lot of match statistics (features), we need to perform feature selection here. And as we have determined that 66 points is a suitable threshold, moving forward, we redefine the objective as being able to achieve 66 points or more. As such, we have split the data into two groups, teams with more than 66 points or less.

A point to note is that by redefining the objective to attain 66 points, the problem has turned from a classification to a regression problem. And this is beneficial as it also simplifies the linear optimization problem.

In [9]:

```
top_4 = post_2017_df[post_2017_df['points']>=66]
non_top_4 = post_2017_df[post_2017_df['points']<66]

top_4_stats = top_4.iloc[:,3:]
non_top_4_stats = non_top_4.iloc[:,3:]
```

Before we used any modules, we decided to see if there was any significant difference between the two groups for each feature. The t-test run below has determined a list of statistically significant features between both groups. We think we should focus on these features later when trying to maximize the points, as the t-test shows that these are the features that are likely to be separating the two groups (teams with more than 66 points and teams with less).

In [10]:

```
stat = []
p_values = []
for col in top_4_stats.columns:

    group1 = top_4_stats[col].array
    group2 = non_top_4_stats[col].array
    p = stats.ttest_ind(a=group1, b=group2, equal_var=True).pvalue
    stat.append(col)
    p_values.append(p)

d = {'stats':stat,'p-value':p_values}
p_values = pd.DataFrame(d).sort_values('p-value')

pd.set_option('display.float_format',  '{:,}'.format)

desirable_stats = p_values[p_values['p-value']<0.05]
sig_diff_features = desirable_stats['stats'].to_list()

desirable_stats
```

Out[10]:

	stats	p-value
7	avg_points	6.863310567703631e-24
6	points	6.998265832882778e-24
0	wins	2.809526808418372e-23
5	goals_diff	2.9193287070391623e-21
2	losses	1.3790066034263885e-18
10	x_goals_diff	3.8311851958412493e-16
3	goals_for	3.1033228677719618e-15
15	att_pass	9.011438723075842e-15
14	cmp_pass	9.229218140270689e-15
12	SoT	9.903185482439278e-14
17	total_dist_pass	1.0672817082398226e-12
4	goals_against	1.6252533459469217e-12
8	x_goals	1.1614372368944793e-11
18	prg_dist_pass	4.20362883946517e-11
29	poss	2.615265285832494e-10
16	cmp_pass_p	2.7707739641837737e-10
11	shots	9.416714223264517e-10
9	x_goals_against	6.874647774087708e-09
30	succ_dribble	8.895362585581785e-09
13	SoT_p	3.985577878571967e-08
27	clearance	2.8658009408851694e-07
31	att_dribble	1.991999519160363e-06
23	a_tackles	4.358813355610152e-05

Featurewiz (all features)

The package Featurewiz has two benefits. It first determines the highly correlated variables and then selects the list of variables that do not have a high correlation. The correlation threshold was set at 0.7 here.

The second benefit is that after removing the highly correlated variables, it uses xgBoost to do feature selection before finally returning a subset of the features from the first step.

For us, we have decided to keep both lists of features. The one suggested by it and the other list which has eliminated all the highly correlated features.

Do note that some features has been manually dropped such as wins, draws, losses, goal difference. They are features that a result of other underlying features and not something players can control, hence it is not suitbale for use. In short, we dropped features like wins because a player can't change whether they win a game or not; as win is a result of their other features.

In [11]:

```
post_2017_stats = post_2017_df.iloc[:,9:]
features1, target = post_2017_stats.iloc[:,3:], post_2017_stats.iloc[:,0]
```

In [12]:

```
data = pd.merge(features1,target,left_index=True,right_index=True)

features, train = featurewiz(data,'points',corr_limit=0.7,verbose=2,sep=",",header=0,test_d
```

```
############################################################################
##################
############      F A S T   F E A T U R E  E N G G   A N D   S E L E C
T I O N ! ########
# Be judicious with featurewiz. Don't use it to create too many un-interpr
etable features! #
############################################################################
##################
Correlation Limit = 0.7
Skipping feature engineering since no feature_engg input...
Skipping category encoding since no category encoders specified in inpu
t...
#### Single_Label Regression problem ####
    Loaded train data. Shape = (100, 24)
#### Single_Label Regression problem ####
No test data filename given...
############################################################################
##############
######################### C L A S S I F Y I N G  V A R I A B L E S  #######
##############
```

In [13]:

```
fw_sugg_features = features
fw_uncorr_features = ['a_tackles', 'block_p', 'block_s', 'clearance', 'errors', 'intercepts
                      'x_goals_diff', 'cmp_pass_p', 'SoT_p', 'att_dribble', 'tackles_w']
```

FeatureSelector (All Features)

Using the list of features with low correlation FeatureWiz has provided, we feed it into another module called FeatureSelector, which will take a subset of the lowly correlated features. The idea is to compare the features suggested by FeatureWiz and FeatureSelector because they both use different methods to find a subset from the lowly correlated features.

In [14]:

```
FS = FeatureSelector(objective = 'regression',auto=True)
fs_sugg_features = FS.fit_transform(post_2017_stats[fw_uncorr_features],target)
```

```
 * Initiating FeatureSelector

   - Comparing LinearRegression and RandomForest for feature selection

   - Running feature selection with Ridge()
     . Experiments are carried out on complete dataset

   - Running feature selection with RandomForestRegressor(max_depth=2, n_est
imators=50)
     . Experiments are carried out on complete dataset

   - Scoring selected feats from linear and RF models by: LGBMRegressor()

     .. RFE by linear model cv-score       : -9.79193
     .. linear model n selected features   : 5

     .. RFE by RandomForest model cv-score : -9.34936
     .. RF model n selected features       : 5

     . Keeping feats from RFE with RF model
 ------------------------------------------------------------------------
     . Selected 5 features from 11

Time elapsed for fit_transform execution: 6.05063 seconds
```

In [15]:

```
fs_sugg_features = fs_sugg_features.columns.to_list()
possible_features = {'fs':fs_sugg_features,'fw':fw_sugg_features}
```

FeatureWiz (Statistically Significant Features)

Similar to the two sub-sections above, we will use FeatureWiz to provide a list of uncorrelated features and a list of suggested features. The difference here is that we are only feeding in the features that was previously determined as statistically significant from the t-test.

In [16]:

```
sig_diff_features_df = post_2017_df[sig_diff_features[1:2] + sig_diff_features[5:6] + sig_d

features, train = featurewiz(sig_diff_features_df,'points',corr_limit=0.7,verbose=2,sep=","
```

```
######################################################################################
##################
############      F A S T   F E A T U R E  E N G G   A N D   S E L E C
T I O N ! ########
# Be judicious with featurewiz. Don't use it to create too many un-interpr
etable features! #
######################################################################################
##################
Correlation Limit = 0.7
Skipping feature engineering since no feature_engg input...
Skipping category encoding since no category encoders specified in inpu
t...
#### Single_Label Regression problem ####
    Loaded train data. Shape = (100, 17)
#### Single_Label Regression problem ####
No test data filename given...
######################################################################################
#############
######################### C L A S S I F Y I N G  V A R I A B L E S  #######
"""""""""""""""
```

In [17]:

```
fw_sugg_sig_diff_features = features
fw_uncorr_sig_diff_features = ['a_tackles', 'clearance', 'x_goals_diff', 'cmp_pass_p', 'SoT
```

FeatureSelector (Statistically Significant)

Likewise, we are using FeatureSelector to select a subset form the statistically significant features

In [18]:

```
target = sig_diff_features_df.iloc[:,0]
X_sig_diff = sig_diff_features_df[fw_uncorr_sig_diff_features]

FS = FeatureSelector(objective = 'regression', auto=True)
fs_sugg_sig_diff_features = FS.fit_transform(X_sig_diff,target)
fs_sugg_sig_diff_features
```

```
 * Initiating FeatureSelector

   - Comparing LinearRegression and RandomForest for feature selection

   - Running feature selection with Ridge()
     . Experiments are carried out on complete dataset

   - Running feature selection with RandomForestRegressor(max_depth=2, n_est
imators=50)
     . Experiments are carried out on complete dataset

   - Scoring selected feats from linear and RF models by: LGBMRegressor()

     .. RFE by linear model cv-score       : -9.68962
     .. linear model n selected features   : 5

     .. RFE by RandomForest model cv-score : -9.52612
     .. RF model n selected features       : 6

     . Keeping feats from RFE with RF model
 ------------------------------------------------------------------------------
     . Selected 6 features from 6

Time elapsed for fit_transform execution: 0.57061 seconds
```

Out[18]:

	a_tackles	clearance	x_goals_diff	cmp_pass_p	SoT_p	att_dribble
0	84.0	526.0	30.7	87.2	37.7	780.0
1	81.0	467.0	27.9	85.9	34.6	747.0
2	67.0	649.0	19.9	78.9	33.2	600.0
3	71.0	603.0	5.1	82.4	32.3	588.0
4	79.0	777.0	10.2	80.8	35.8	655.0
...	...	...	...	...	...	...
95	76.0	820.0	-0.6	74.9	33.7	611.0
96	64.0	777.0	-7.6	69.7	31.8	608.0
97	60.0	942.0	-7.1	75.3	28.8	622.0
98	60.0	713.0	-32.0	80.0	29.4	714.0
99	86.0	890.0	-21.0	72.4	27.2	586.0

100 rows × 6 columns

In [19]:

```
fs_sugg_sig_diff_features = fs_sugg_sig_diff_features.columns.to_list()
possible_features['fw_sig']=fw_sugg_sig_diff_features
possible_features['fs_sig']=fs_sugg_sig_diff_features
```

In [20]:

```
possible_features
```

Out[20]:

```
{'fs': ['clearance', 'errors', 'x_goals_diff', 'cmp_pass_p', 'SoT_p'],
 'fw': ['x_goals_diff',
  'cmp_pass_p',
  'SoT_p',
  'clearance',
  'errors',
  'tackles_w'],
 'fw_sig': ['x_goals_diff', 'SoT_p', 'cmp_pass_p'],
 'fs_sig': ['a_tackles',
  'clearance',
  'x_goals_diff',
  'cmp_pass_p',
  'SoT_p',
  'att_dribble']}
```

Model Training

After gathering four different lists of features that we could use, we will now feed these different subsets into LazyPredictor. LazyPredictor essentially fits and trains the data on many different kinds of models to allow us to see which model we should use and possibly which subset of features we should use to maximize our points.

In [21]:

```
def train(data, features, target):
    features = data[features]
    X_train, X_test,y_train, y_test = train_test_split(features, target, test_size=.2, rand
    reg = LazyRegressor(predictions=True)
    models, predictions = reg.fit(X_train, X_test, y_train, y_test)

    return models
```

In [22]:

```
possible_features_acc={}
for key in possible_features:
    features = possible_features.get(key)
    possible_features_acc[key]=train(post_2017_stats,features,target)
```

```
100%|████████████████████████████████████████████████████████████
          | 42/42 [00:01<00:00, 22.76it/s]
100%|████████████████████████████████████████████████████████████
          | 42/42 [00:01<00:00, 23.37it/s]
100%|████████████████████████████████████████████████████████████
          | 42/42 [00:01<00:00, 24.02it/s]
100%|████████████████████████████████████████████████████████████
          | 42/42 [00:01<00:00, 23.53it/s]
```

In [23]:

```
possible_features['fw_sig']
```

Out[23]:

```
['x_goals_diff', 'SoT_p', 'cmp_pass_p']
```

In [24]:

```
possible_features_acc['fs'].head()
possible_features_acc['fs_sig'].head()
possible_features_acc['fw'].head()
possible_features_acc['fw_sig'].head()
```

Out[24]:

Model	Adjusted R-Squared	R-Squared	RMSE	T
AdaBoostRegressor	0.7490241828569979	0.7886519434585245	7.911129604139081	0.12043881
KNeighborsRegressor	0.7128369536770478	0.7581784873069877	8.462269199216012	0.015619277
Lasso	0.7115501926627359	0.7570948990844092	8.481207477204034	0.015621423
RANSACRegressor	0.7091355756181013	0.7550615373626116	8.516631646058975	0.03253293
HuberRegressor	0.7090492158250439	0.7549888133263527	8.51789587760384	0.02403736

As we can see from the results, the consistently best-performing models are the Lasso model and AdaBoost Regressor.

For the Lasso model, it was able to achieve an accuracy of 0.67 and 0.64 with the features selected by FeatureSelector and FeatureWiz. And an accuracy of 0.67, 0.71 when only feature selecting from statistically significant features.

The AdaBoost Regressor achieved an accuracy of 0.66 and 0.71 with the features selected by FeatureSelector and FeatureWiz. And an accuracy of 0.64 and 0.75 when only feature selecting from statistically significant features.

As a result, we can see that the features that FeatureWiz selects from the statistically significant features provide the best accuracy across both models, and we decide to use those features as regressors.

These features include "x_goals_diff," "SoT_p," and 'cmp_pass_p." And since these are our main regressors, these are also the features that we want to focus on to achieve our objective of reaching more than 66 points and, in turn, qualify for the champions league.

In [25]:

```
post_2017_df['66pts']= ['GE66' if i>=66 else 'LT66' for i in post_2017_df['points']]
```

Boxplot of the key regressors between the two groups are plotted below to give a visual justification.

In [26]:

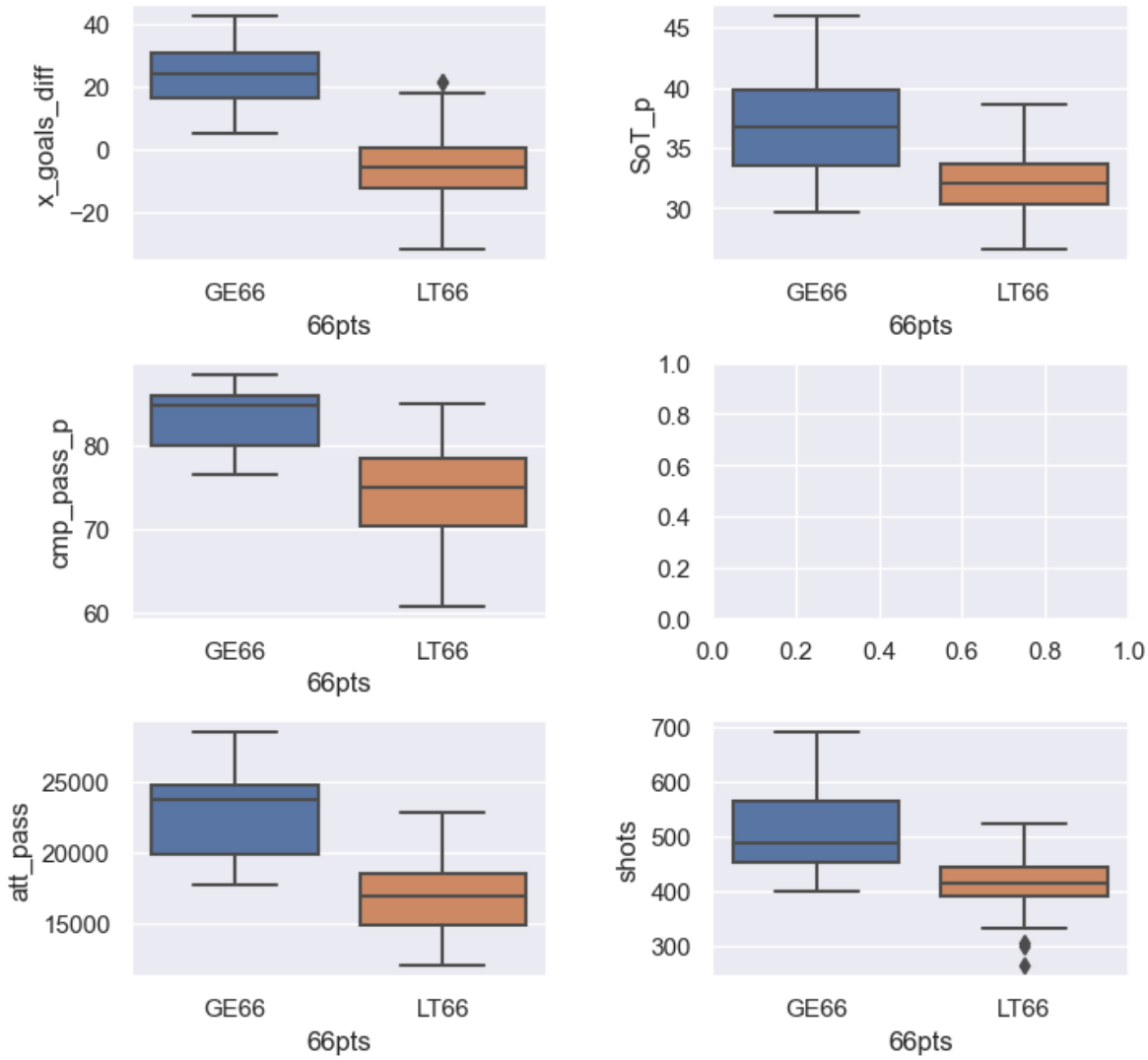
```
sns.set()

fig, axes = plt.subplots(3, 2,figsize=(8,8))
fig.subplots_adjust(hspace=0.4, wspace=0.4)

sns.boxplot(data=post_2017_df, x='66pts', y=possible_features['fw_sig'][0], ax=axes[0,0])
sns.boxplot(data=post_2017_df, x='66pts', y=possible_features['fw_sig'][1], ax=axes[0,1])
sns.boxplot(data=post_2017_df, x='66pts', y=possible_features['fw_sig'][2], ax=axes[1,0])
sns.boxplot(data=post_2017_df, x='66pts', y='shots', ax=axes[2,1])
sns.boxplot(data=post_2017_df, x='66pts', y='att_pass', ax=axes[2,0])
```

Out[26]:

```
<AxesSubplot:xlabel='66pts', ylabel='att_pass'>
```



To keep the linear optimization model simple, we have decided to use the average of each feature in the group with more than 66 points as our threshold for the constraints. We have also included the number of shots and attempted passes as a constraint because "SoT_p" and "cmp_pass_p" are efficiency statistics. In other words, having 100% SoT_p with only 1 shot is not entirely useful.

As such, when creating a team, we would like to achieve a minimum of

In [27]:

```
print("For Expected Goal Difference:", math.floor(statistics.median(top_4_stats['x_goals_di
print("For Shots Attempted:", math.floor(statistics.median(top_4_stats['shots'])))
print("For Shots on Target Percentage:",math.floor(statistics.median(top_4_stats['SoT_p']))
print("For Attempted Passes:", math.floor(statistics.median(top_4_stats['att_pass'])))
print("For Completed Pass Percentage:",math.floor(statistics.median(top_4_stats['cmp_pass_p
```

```
For Expected Goal Difference: 23
For Shots Attempted: 488
For Shots on Target Percentage: 36
For Attempted Passes: 23733
For Completed Pass Percentage: 84
```

Linear Optimization

Now that we have determined the essential features that help gain the most points, we focus on finding the minimum cost needed.

The idea is to minimize the cost function, which is the player's wages under some constraints. The constraints include having n amount of goalkeeper, defender, midfielder, and forward. More importantly, the key constraint would be achieving a minimum threshold for each significant feature we have selected above.

For shots, our minimum threshold is 400. For attempted passes, our minimum threshold is 22000. For shots on target percentage, our minimum threshold is 36% For the completed pass percentage, our minimum threshold is 84% For the expected goal difference, our minimum threshold is 23.

Some of the thresholds were relaxed to ensure a feasible solution could be found.

In [28]:

```
player_stat = player_stat.rename(columns={'xG_x':"xGF","Att_x":"Passes","Cmp%":"Cmp_p","SoT
```

In [29]:

```
indices = player_stat.Player
cost = dict(zip(indices,player_stat.Wage))
shots = dict(zip(indices,player_stat.Sh))
passes = dict(zip(indices,player_stat.Passes))
SoT_p = dict(zip(indices,player_stat.SoT_p))
cmp_p = dict(zip(indices,player_stat.Cmp_p))
xGF = dict(zip(indices,player_stat.xGF))
xGA= dict(zip(indices,player_stat.xGA))
player_position = list(zip(player_stat.Player,player_stat.Pos))

player_map = list(player_stat.Player)

m = Model()
y = m.addVars(player_stat.Player,vtype=GRB.BINARY,name='y')
m.setObjective(quicksum(cost[i]*y[i] for i in indices), GRB.MINIMIZE)

m.addConstr(quicksum([y[i] for i, position in player_position if position=='GK'])==1)

m.addConstr(quicksum([y[i] for i, position in player_position if position=='DF'])>=3)
m.addConstr(quicksum([y[i] for i, position in player_position if position=='DF'])<=5)

m.addConstr(quicksum([y[i] for i, position in player_position if position=='MF'])>=3)
m.addConstr(quicksum([y[i] for i, position in player_position if position=='MF'])<=5)

m.addConstr(quicksum([y[i] for i, position in player_position if position=='FW'])>=1)
m.addConstr(quicksum([y[i] for i, position in player_position if position=='FW'])<=3)

m.addConstr(quicksum(y[i] for i in indices)==11)

### ADD CONSTRAINT ON WHAT TYPE OF STATISTICS WE WANT TO HIT
m.addConstr(quicksum(shots[i]*y[i] for i in indices) >= 400, name='shots')
m.addConstr(quicksum(passes[i]*y[i] for i in indices) >= 22000, name='passes')
m.addConstr(quicksum(SoT_p[i]*y[i] for i in indices) >= 36*9, name='SoT_p')
m.addConstr(quicksum(cmp_p[i]*y[i] for i in indices) >= 84*10, name='cmp_p')
m.addConstr(quicksum((xGF[i]-xGA[i])*y[i] for i in indices) >= 23, name='xGD')



m.optimize()

r = pd.DataFrame()

for v in m.getVars():
    if v.x > 1e-6:
        r = r.append(player_stat.iloc[v.index][['Player','Pos','Wage']])
        print(v.varName)

print('Cost:', m.objVal)
```

```
Restricted license - for non-production use only - expires 2024-10-28
Gurobi Optimizer version 10.0.0 build v10.0.0rc2 (win64)

CPU model: Intel(R) Celeron(R) N4020 CPU @ 1.10GHz, instruction set [SSE2]
Thread count: 2 physical cores, 2 logical processors, using up to 2 thread
s

Optimize a model with 13 rows, 533 columns and 3982 nonzeros
Model fingerprint: 0xf1c258d3
```

```
Variable types: 0 continuous, 533 integer (533 binary)
Coefficient statistics:
  Matrix range     [1e-01, 3e+03]
  Objective range  [4e+04, 3e+07]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 2e+04]
Presolve removed 0 rows and 15 columns
Presolve time: 0.01s
Presolved: 13 rows, 518 columns, 3875 nonzeros
Variable types: 0 continuous, 518 integer (518 binary)
```

After the program has done the optimization, it has recommended the following team below with their statistics. The cost of the team is $94,460,000 in annual wages.

In [30]:

```
col = {'Player':[],'Squad':[],'xGF':[],'xGA':[],'SoT_p':[],'Cmp_p':[]}
team_stats = pd.DataFrame(col)

for i in r['Player']:
    x = player_stat[player_stat['Player']==i]
    x = x[['Player','Pos','Squad','xGF','xGA','SoT_p','Cmp_p']]
    team_stats= team_stats.append(x)

team_stats["xGD"]= team_stats["xGF"] - team_stats["xGA"]

team_stats
```

Out[30]:

	Player	Squad	xGF	xGA	SoT_p	Cmp_p	Pos	xGD
10	Jordi Alba	Barcelona	1.8	0.0	19.4	83.5	DF	1.8
31	Iago Aspas	Celta Vigo	14.8	0.0	42.4	70.6	FW	14.8
57	Karim Benzema	Real Madrid	24.3	0.0	41.0	83.8	FW	24.3
69	Sergio Busquets	Barcelona	0.8	0.0	28.6	87.0	MF	0.8
78	Sergio Canales	Betis	4.8	0.0	44.4	82.6	FW	4.8
84	Diego Carlos	Sevilla	2.4	0.0	33.3	87.3	DF	2.4
163	Javi GalÃ¡n	Celta Vigo	0.6	0.0	26.3	77.8	DF	0.6
254	Jules KoundÃ©	Sevilla	1.5	0.0	23.1	88.0	DF	1.5
379	Daniel Parejo	Villarreal	1.7	0.0	33.3	83.7	MF	1.7
423	Ã lex Remiro	Real Sociedad	0.0	31.4	0.0	75.4	GK	-31.4
474	Denis SuÃ¡rez	Celta Vigo	1.9	0.0	36.4	79.8	MF	1.9

Evaluation

To ensure that the team built is capable of attaining the minimum 66 points, which gives a 95% chance of qualifying for the Champions League, we have built an AdaBoost Regressor (with an accuracy of 0.87) below to predict the points the optimized team would achieve. From the code below, we see that they will achieve about 70 points, which is beyond what we need.

In [31]:

```
X, y = post_2017_df[['x_goals_diff','SoT_p','cmp_pass_p']], post_2017_df[['points']]

regr = AdaBoostRegressor()
regr.fit(X, y)
regr.score(X,y)

team_xGD = sum(team_stats['xGD'])
team_SoTp = np.mean(team_stats['SoT_p'])
team_cmpp = np.mean(team_stats['Cmp_p'])
regr.predict([[team_xGD,team_SoTp,team_cmpp]])
```

Out[31]:

```
array([70.5])
```

To ensure that the prediction isn't a one off luck, we simulated the prediction for 1000 seasons, to see what is the average points it will achieve.

In [32]:

```
prediction = []
for i in range(1000):
    prediction.append(regr.predict([[team_xGD,team_SoTp,team_cmpp]]))

np.mean(prediction)
```

Out[32]:

```
70.5
```

Conclusion

To that end, from the algorithm above, we evaluated that the minimum cost to build a team qualifying for the Champions League in La Liga would be 94 million euros in annual wages.

And as a comparison to the annual wages of all teams across the last 5 seasons, a scatterplot is created below. Although 94 million euros may seem a lot, when compared to the teams that achieved more than 66 points in the last 5 seasons, 94 million is relatively small.

Only the last 5 seasons are compared because the annual wages of the player are their current wage in the 2022-23 season. And to account for inflation and ever-increasing wages, it would not be appropriate to compare wages today with wages before the 2017-18 seasons.

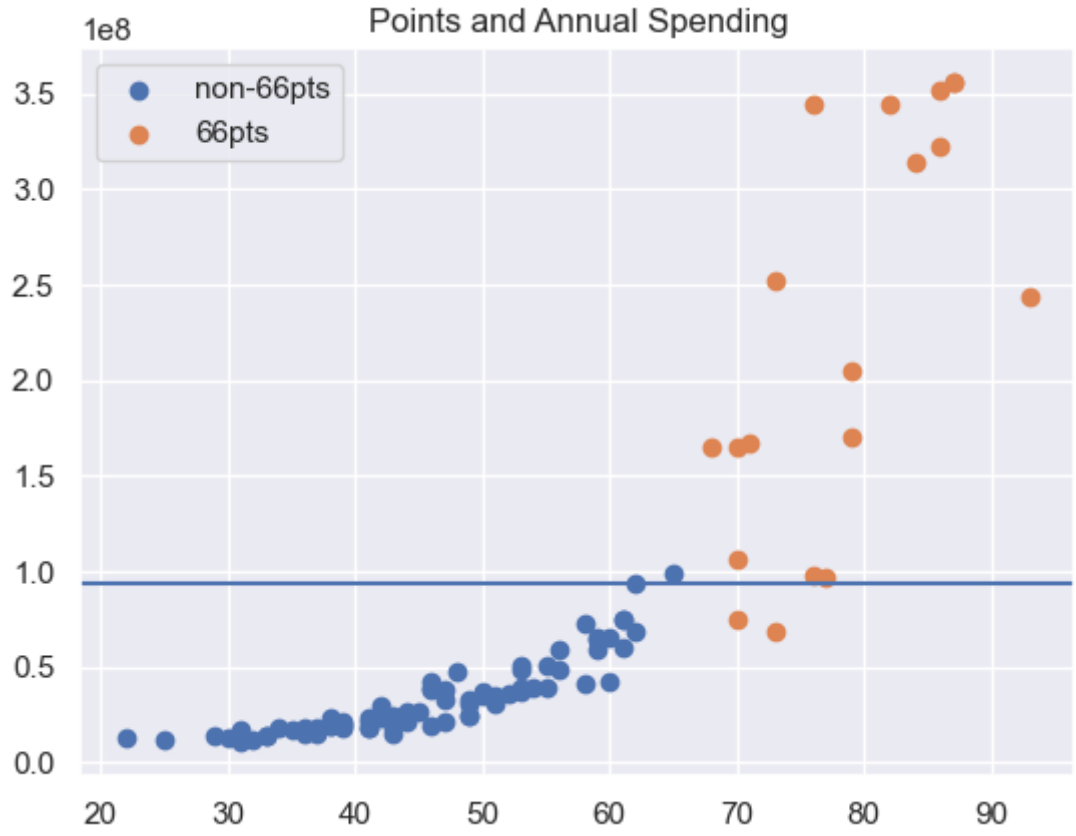
In [33]:

```
team_wages = pd.read_csv('../output/annual_wages_by_team.csv',encoding= 'unicode_escape').i
team_wages = team_wages.rename(columns={"Squad":"team","Rk":"rank"})
team_points = post_2017_df[['Season','rank','team','points']]
team_wages = pd.merge(team_wages,team_points,on=['Season','rank'])

team_66pts = team_wages[team_wages['points']>=66]
team_non_66pts = team_wages[team_wages['points']<66]

plt.scatter(team_non_66pts['points'], team_non_66pts['Annual Wages'], label=f'non-66pts')
plt.scatter(team_66pts['points'], team_66pts['Annual Wages'], label=f'66pts')
plt.axhline(y=94000000)

plt.rcParams.update({'figure.figsize':(6.4,4.8), 'figure.dpi':100})
plt.title('Points and Annual Spending')
plt.legend()
plt.show()
```



In [34]:

```
len(team_66pts[team_66pts["Annual Wages"]>=94000000])/len(team_66pts)
```

Out[34]:

```
0.8947368421052632
```

As a final comparison, of the teams that achieved more than 66 points in the last 5 seasons, about 90% of them spent more than the 94 million euros proposed by the algorithm