Assignment 2 - Machine Learning

FEUP MIEIC - Inteligência Artificial *(EIC0029/IART)*

Complete Assignment

This Jupyter Notebook comprises of all the work developed for the 2nd Assignment of the Course Inteligencia Artificial (EIC0029/IART) of Mestrado Integrado em Engenharia Informática e Computação (MIEIC) at Fauldade de Engenharia da Universidade do Porto (FEUP)

Index

> Although this document contains all work developed, in this repository you will also find the partial notebooks used throughout the developent process, devided by the items in this index, reachable through the links embeded in the aforementioned section

Authors

- Cláudia Raquel Mamede - up201604832@fe.up.pt
- João Veiga Macedo - up201704464@fe.up.pt
- Raul Viana- up201208089@fe.up.pt

Abstract

This assignment had the objective of analyzing a football dataset and train several models in predicting the match outcome.
The models implemented were the decision tree, the k-nearest neighbors, and a neural network. The dataset had some data flaws, that were corrected with the machine learning tools. The models were implemented with the best hyperparameters and compared with each other. The neural network was the model with the best behavior, closely followed by the k-nearest neighbor.
Lastly, the score achieved by the models were insufficient to surpass the betting houses predictions, which lead us to conclude we needed more data.

Introduction

This assignment had the objective of comparing the effectiveness of several algorithms to classify a class from a dataset. For that, we used the European Soccer Database from https://www.kaggle.com/hugomathien/soccer (https://www.kaggle.com/hugomathien/soccer). From this dataset, we tried to apply machine learning algorithms to predict the outcome of a football match. We start to analyze and normalize the data. After applying it the three selected algorithms, we compare its results and effectiveness.

1. Data Pre-processing

In this section we will implement and document our approach to extract, manipulate, and in essence pre-process the data that will be used to train and test our machine learning models. The main goal of this step is to reduce the processing time of each algorithm and increase their precision by using only the relevant information.

We started by extracting all important data from the dataset provided. Because there all multiple missing values in the *team_attributes* table, we chose not to use it. We also decided not to use the tables *country* , *league* and *player* since we considered their data irrelevant to our problem.

In [7]:

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from scipy import stats
import sqlite3
from sklearn.model_selection import train_test_split
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn import model_selection
from sklearn.model_selection import GridSearchCV
from time import time
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
```

In [8]:

```
# Connecting to database

database = "database.sqlite"
con = sqlite3.connect(database)
```

In [9]:

```
# Extracting relevant data from the database

matches_df = pd.read_sql("""SELECT * from MATCH""", con)
teams_df = pd.read_sql("""SELECT * from TEAM""", con)
player_attributes_df = pd.read_sql("""SELECT * from PLAYER_ATTRIBUTES""", con)
team_attributes_df = pd.read_sql("""SELECT * from TEAM_ATTRIBUTES""", con)
```

At a first glance we verified the existence of too many columns so we decided to make some adjustments: we started by choosing the features we wanted to keep and after that we tried to aggregate all related variables (such as players' info and odds) and have some "background" info of each team in a match.

In [11]:

```
# Selecting features to keep

matches_kept_columns = ["id", "league_id", "date", "home_team_api_id", "away_tea
m_api_id", "home_team_goal", "away_team_goal"]
```

In [12]:

```
# Selecting players labels

home_players = ["home_player_" + str(x) for x in range(1, 12)]
away_players = ["away_player_" + str(x) for x in range(1, 12)]


# Merge features to keep

matches_kept_columns = matches_kept_columns + home_players
matches_kept_columns = matches_kept_columns + away_players
matches_df = matches_df[matches_kept_columns]
```

In [13]:

```
#Geting overall ratings for all players from the player_attributes table

for player in home_players:
    matches_df = pd.merge(matches_df, player_attributes_df[["id", "overall_ratin
g"]], left_on=[player], right_on=["id"], suffixes=["", "_" + player])
for player in away_players:
    matches_df = pd.merge(matches_df, player_attributes_df[["id", "overall_ratin
g"]], left_on=[player], right_on=["id"], suffixes=["", "_" + player])


matches_df = matches_df.rename(columns={"overall_rating": "overall_rating_home_p
layer_1"})

matches_df['overall_rating_home'] = matches_df[['overall_rating_' + p for p in h
ome_players]].sum(axis=1)
matches_df['overall_rating_away'] = matches_df[['overall_rating_' + p for p in a
way_players]].sum(axis=1)
matches_df['overall_rating_difference'] = matches_df['overall_rating_home'] - ma
tches_df['overall_rating_away']

matches_df['mean_overall_rating_home'] = matches_df[['overall_rating_' + p for p
in home_players]].mean(axis=1)
matches_df['mean_overall_rating_away'] = matches_df[['overall_rating_' + p for p
in away_players]].mean(axis=1)
```

Since our goal was reducing the number of features of a *match*, we ultimately ended up erasing all variables related to a specific player and keep only the mean values of the players' overall attributes and the difference between the best and worst players of each teams.

In [15]:

```
# Removing individual player info

for c in matches_df.columns:
    if '_player_' in c:
        matches_df = matches_df.drop(c, axis=1)
```

We considered that was important to understand the evolution of a team between matches, so we developed a method that is able to extract the results of the last 5 matches of a team.

In [17]:

```
#function to calculate last 5 games performance
def last5(team_id, date, match_t):

    mat = match_t[(match_t['date'] < date)]
    mat = mat[mat['home_team_api_id'] == team_id]

    mat5 = mat.head(5)
    if len(mat5.index) < 5:
        return -1
    mat5['Home result'] = 0
    mat5['Home result'] = np.where(mat5['home_team_goal'] > mat5['away_team_goa
l'], 3, mat5['Home result'])
    mat5['Home result'] = np.where(mat5['home_team_goal'] == mat5['away_team_goa
l'], 1, mat5['Home result'])
    total = mat5['Home result'].sum()
    return total

match_t = matches_df[['home_team_api_id', 'away_team_api_id','date', 'home_team_
goal', 'away_team_goal']].copy()
match_t['date'] = pd.to_datetime(match_t['date'])
match_t.sort_values(by=['date'], inplace=True, ascending=False)
matches_df['Home_last5'] = 0
matches_df['Away_last5'] = 0
perc = 0
for i in matches_df.index:
    Htotal = last5(match_t['home_team_api_id'].iloc[i], match_t['date'].iloc[i],
match_t)
    Atotal = last5(match_t['away_team_api_id'].iloc[i], match_t['date'].iloc[i],
match_t)
    matches_df['Home_last5'].values[i] = Htotal
    matches_df['Away_last5'].values[i] = Atotal
    if i % 655 == 0:
        perc += 10
        print(" ", perc, "%", end=" ")
#Dont considerate games that dont have 5 previous history
matches_df = matches_df[matches_df.Home_last5 != -1]
matches_df = matches_df[matches_df.Away_last5 != -1]
```

```
10 %  20 %  30 %  40 %  50 %  60 %  70 %  80 %  90 %  100 %
```

Finally, we decided to also include the odds in our final data set. The odds values are the result of yet another computation attempt to predict the match outcome. But its a prediction based on the data, and most probably other data than this dataset, so it's more information for our models. There were many *null* values so we replaced them with their mean values.

In [19]:

```
matches_aux = pd.read_sql("""SELECT * FROM MATCH""" ,con)

#Select all bet columns (removed PSA, PSH, PSD because they are almost all NaN)

bet_columns = ["B365H", "B365A", "B365D", "BWH", "BWD", "BWA", "IWH", "IWD", "IW
A", "LBH", "LBD", "LBA", "WHH", "WHD", "WHA", "SJH", "SJD", "SJA", "VCH", "VCD",
"VCA", "GBH", "GBD", "GBA", "BSH", "BSD", "BSA"]


#Get specific columns for bets on home and draw

bet_columns_home = ["B365H", "BWH","IWH", "LBH", "WHH", "SJH", "VCH", "GBH","BS
H"]
bet_columns_draw = ["B365D", "BWD","IWD", "LBD", "WHD", "SJD", "VCD", "GBD","BS
D"]
bet_columns_away = ["B365A", "BWA","IWA", "LBA", "WHA", "SJA", "VCA", "GBA","BS
A"]


#Calculate mean values for bets on home team and draw. Add these values to match
table

matches_df['mean_bets_home'] = matches_aux[bet_columns_home].mean(axis=1)
matches_df['mean_bets_draw'] = matches_aux[bet_columns_draw].mean(axis=1)
matches_df['mean_bets_away'] = matches_aux[bet_columns_away].mean(axis=1)


#Replace NaN values (on bets) with mean values

matches_df.fillna(matches_df.mean(), inplace=True)
```

We calculated our class variable by getting the goals difference in each match and then mapping it into the correspondent class.

In [21]:

```
# Getting the goal difference

matches_df['goal_diff'] = matches_df['home_team_goal'] - matches_df['away_team_g
oal']

matches_df['Game Result'] = 'Defeat'
matches_df['Game Result'] = np.where(matches_df['goal_diff'] == 0, 'Draw', match
es_df['Game Result'])
matches_df['Game Result'] = np.where(matches_df['goal_diff'] > 0, 'Win', matches
_df['Game Result'])
```

We also dropped all unnecessary variables.

In [23]:

```
#Removing unnecessary data

matches_df = matches_df.drop([ 'id', 'league_id', 'date', 'home_team_api_id','aw
ay_team_api_id','home_team_goal','away_team_goal','goal_diff'], axis=1)
matches_df.head()
```

Out[23]:

	overall_rating_home	overall_rating_away	overall_rating_difference	mean_overall_rating_hor
0	746.0	783.0	-37.0	67.8181
1	772.0	790.0	-18.0	70.1818
2	780.0	719.0	61.0	70.9090
3	688.0	717.0	-29.0	68.8000
5	809.0	778.0	31.0	73.5454

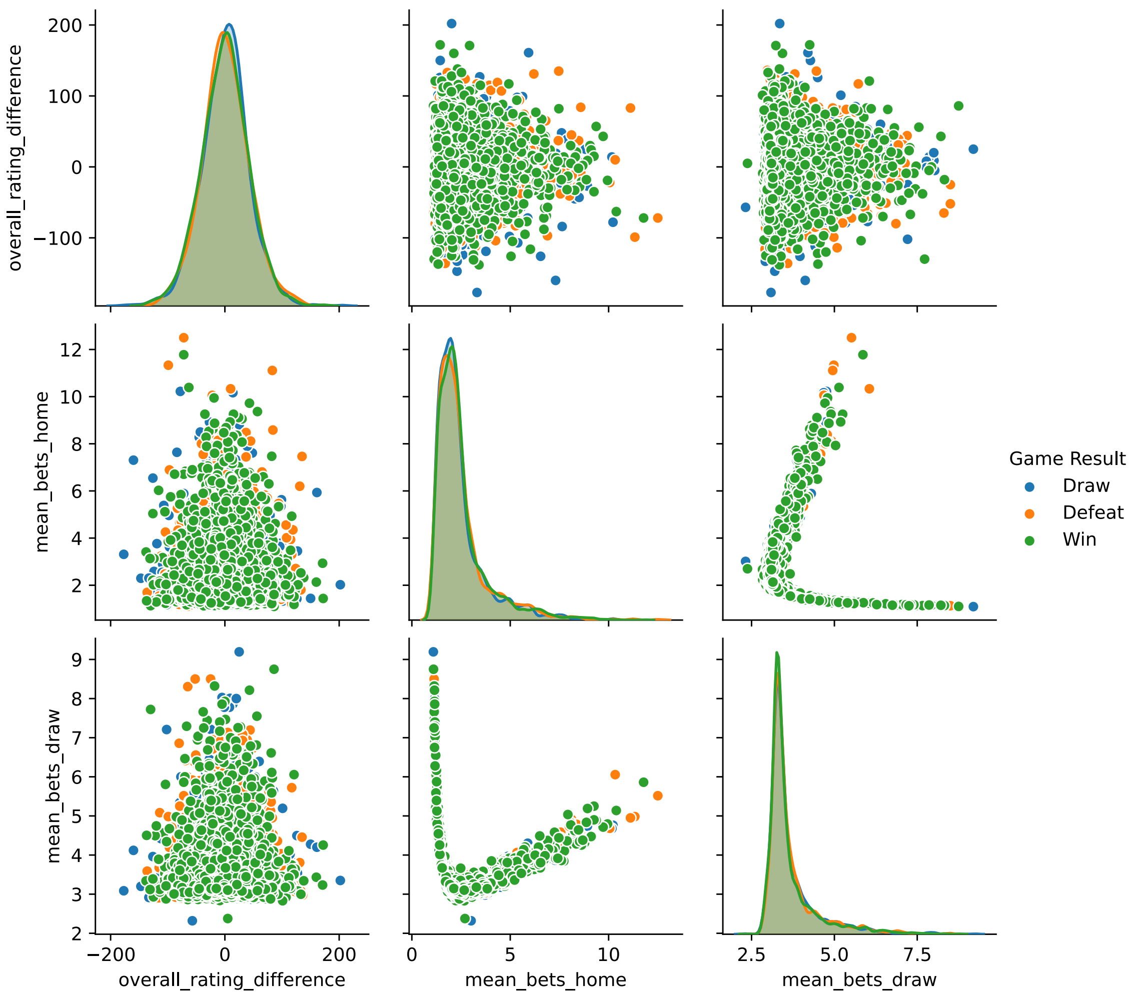
Lastly, we decided to plot some of the data. The plot shows that there are relations where the classification is well defined. For example in the overall_rating_difference vs mean_bets_home is possible to isolate the 'Win' class from the other two with relative easiness, although separating 'Draw' from 'Defeat' is not so trivial. This hints that its possible to predict, with enough data and with an error margin, the outcome of a football match.

In [25]:

```
plot_data = matches_df[['overall_rating_difference', 'mean_bets_home', 'mean_bet
s_draw', 'Game Result']].copy()
sns.pairplot(plot_data, hue='Game Result')
;
```

Out[25]:

```
''
```


overall_rating_difference
mean_bets_home
mean_bets_draw
Game Result
Draw
Defeat
Win
200
100
0
−100
12
10
8
6
4
2
9
8
7
6
5
4
3
2
−200
0
200
0
5
10
2.5
5.0
7.5

We divided our data into a **training dataset** (the sample of data used to fit the model) and a **testing dataset** (the sample of data used to provide an unbiased evaluation of a model fit on the training dataset while tuning model hyperparameter), with a 70/30 random split.

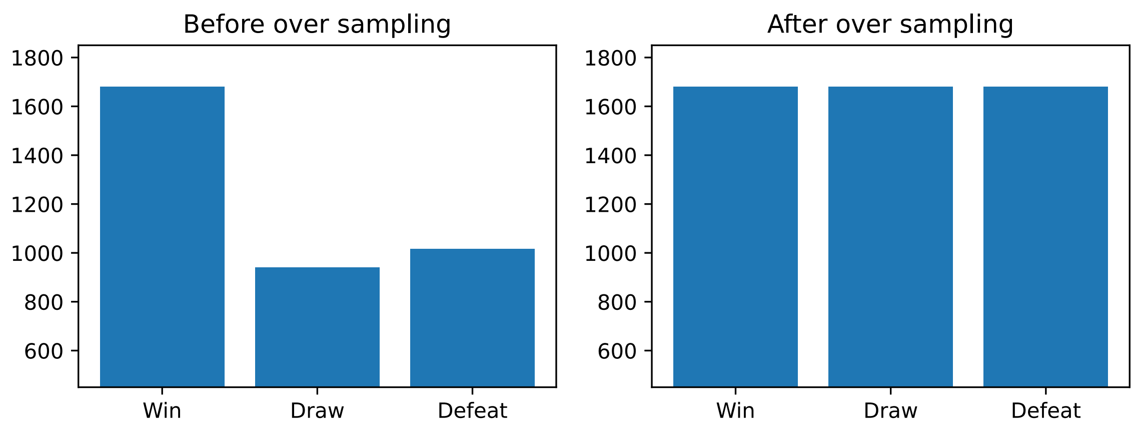
In [27]:

```
# Getting the train and test sets


y = matches_df['Game Result']
X = matches_df.drop('Game Result', axis=1)

X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.3,
                                                    random_state=42,
                                                    stratify=y)
```

After some testing, we verified that our data set was still very **imbalanced** (very unequal number of instances of classes) and that our classification algorithms were very sensitive to it. Considering that we had already a small number of data, we opted for an **over-sampling** approach by duplicating samples of the classes with fewer instances.

In [29]:

```
from imblearn.over_sampling import SMOTE

# Selecting our class variable (y) and extracting it from the features list (x)

before_over_sampling = [sum(y_train == 'Win'), sum(y_train == 'Draw'), sum(y_tra
in == 'Defeat')]

X_train, y_train = SMOTE().fit_sample(X_train, y_train)

after_over_sampling = [sum(y_train == 'Win'), sum(y_train == 'Draw'), sum(y_trai
n == 'Defeat')]


# Ploting the graphs to show the difference

columns = ('Win', 'Draw', 'Defeat')

plt.figure(figsize=(9, 3))
axis1 = plt.subplot(121)
axis1.set_ylim(450, 1850)
plt.bar(columns, before_over_sampling)
plt.title('Before over sampling')
axis2 = plt.subplot(122)
axis2.set_ylim(450, 1850)
plt.bar(columns, after_over_sampling)
plt.title('After over sampling')
plt.show()
```



Since our dataset contains features highly variable in units (e.g.: odds and players overall), we agreed that **scaling** both our training and testing dataset was a good idea. It is also important to say that although this step is not so relevant for tree based models, algorithms such as k-nearest neighbor that uses Eucliden distance measure are very sensitive to magnitudes and hence should be scaled for all features to weigh in equally.

In [31]:

```
# Scaling data

scaler = preprocessing.StandardScaler().fit(X_train)

X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

2. Decision Tree

In this section we will implemented a decision tree model. In these tree structures, leaves represent class labels and branches represent conjunctions of features that lead to those class labels.

In [33]:

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report


parameters = {'max_depth': range(5,150),
              'criterion': ['gini', 'entropy'],
              'splitter' : ['random', 'best']
             }


#start_time = time.time()

clf = GridSearchCV(DecisionTreeClassifier(), parameters, n_jobs=-1, cv=5)

clf.fit(X=X_train_scaled, y=y_train)

#print("--- %s seconds ---" % (time.time() - start_time))

tree_model = clf.best_estimator_

print (clf.best_score_, clf.best_params_)
```

```
0.4946428571428571 {'criterion': 'gini', 'max_depth': 55, 'splitte
r': 'random'}
```

DecisionTreeClassifier from sklearn.tree has a parameter (*class_weight*) that directly modify the loss function by giving more (or less) penalty to the classes with more (or less) weight. In effect, one is basically sacrificing some ability to predict the lower weight class (the majority class for unbalanced datasets) by purposely biasing the model to favor more accurate predictions of the higher weighted class (the minority class). Oversampling methods essentially give more weight to particular classes as well (duplicating observations duplicates the penalty for those particular observations, giving them more influence in the model fit), but due to data splitting that typically takes place in training this will yield slightly different results.

In short and as it was said before, **scaling data is not usually required for this kind of model** (we tested with and without the scaled data and there wasn't a big of a difference) and we could use the parameter *class_weight* instead of the values calculated before but because one of the main goals of this project is to compare the 3 algorithms, we opted by using the scaled and balanced data as an input for this model as well.

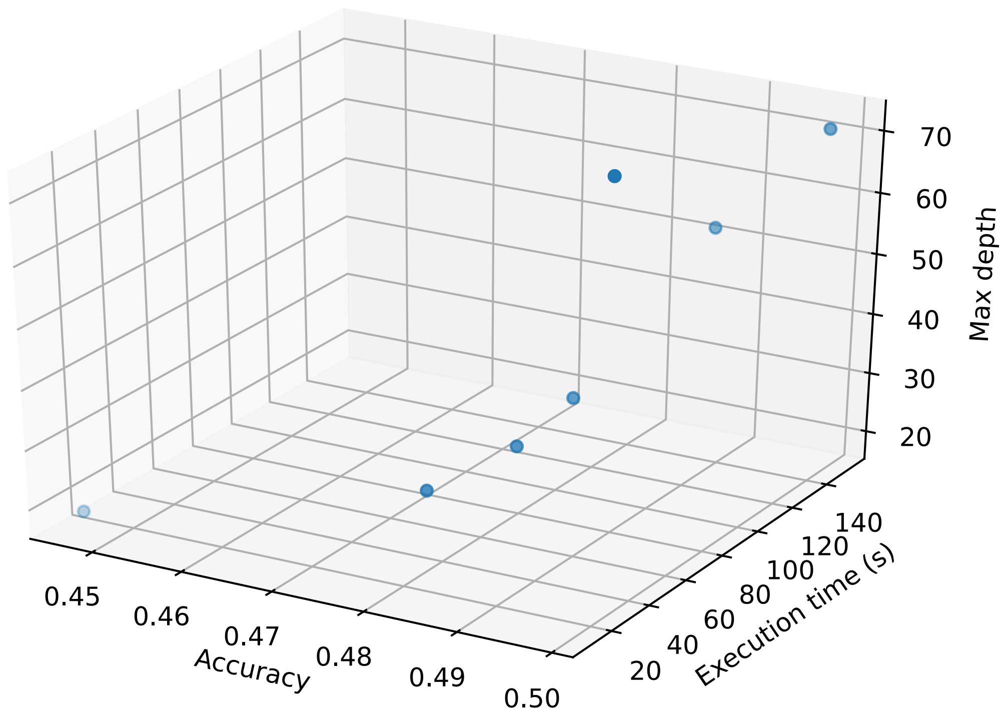
Regarding the model parameters, we decided to vary some of them in order to find the best ones that fit our problem for the given training set. As excepted when we increase the range of the max_depth values, the executing time of the algorithm also increases. We calculated the accuracy (evaluated on a particular fold out of the 5-fold split) and execution time of this model for the selected max_depth ranges:

5-20: (0.44675273478039246, 10.236027002334595, 19); ('gini', 'best')
5-40: (0.48018068069719930, 31.185487031936646, 29); ('gini', 'random')
5-60: (0.48559230044262560, 51.509896039962770, 34); ('entropy', 'random')
5-80: (0.48776637292759945, 70.301864385604860, 39); ('gini', 'random')
5-100: (0.48681144592230774, 94.856812715530400, 70); ('entropy', 'random')
5-120: (0.49182426262692935, 124.91989612579346, 58); ('gini', 'random')
5-150: (0.49843819561334990, 152.26390242576600, 71); ('entropy', 'random')

Obs: we are using a random split to divide our testing and training sets so these values may change a bit through executions.

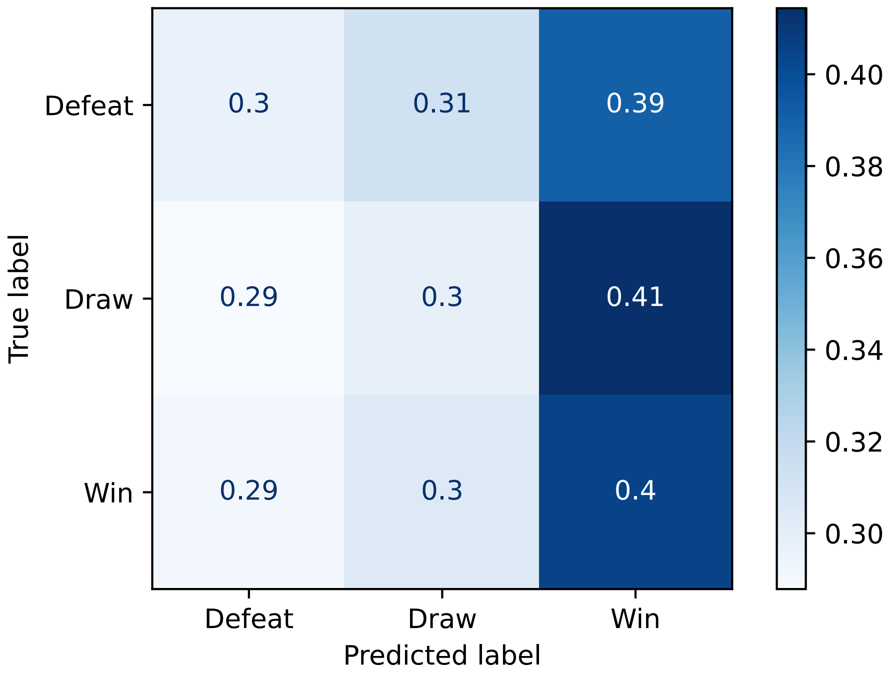
In [36]:

```
from mpl_toolkits.mplot3d import Axes3D

x = [0.44675273478039246, 0.48018068069719930, 0.48559230044262560,  0.487766372
92759945, 0.48681144592230774, 0.49182426262692935, 0.49843819561334990]
y = [10.236027002334595, 31.185487031936646, 51.509896039962770, 70.301864385604
860, 94.856812715530400, 124.91989612579346, 152.26390242576600]
z = [19,29,34,39,70,58,71]

fig = plt.figure()
ax = Axes3D(fig)
ax.scatter(x,y,z)
ax.set_xlabel('Accuracy')
ax.set_ylabel('Execution time (s)')
ax.set_zlabel('Max depth')

plt.show()
```


Max depth
70
60
50
40
30
20
Accuracy
0.45
0.46
0.47
0.48
0.49
0.50
Execution time (s)
20
40
60
80
100
120
140

In [37]:

```
disp = plot_confusion_matrix(tree_model, X_test_scaled, y_test, normalize="true"
, cmap=plt.cm.Blues)
```



In [38]:

```
y_pred = tree_model.predict(X_test_scaled)
```

In [39]:

```
tree_report = classification_report(y_test, y_pred, output_dict="true")
```

3. K-Nearest Neighbor

In this section we will implement a K-nearest neighbor (KNN) model.

First, using the sklearn GridSearch we searched for the optimal KNN hyperparameters.

- The **distance metric** is the method to calculate the distance between two instances: the Euclidean, the Manhattan, or the Minkowski distance.
- The **number of neighbors** is the number of instances to look for near the data. Typically is an odd number. For example, k=3 means that the model will look for the three nearest data to classify the current data.
- The **metric** gives different weights to neighbors, weigthing more the closest ones, or uniformly.
  After the grid search applied we found that the hyperparameters that best fit the model are the Euclidean metric, 23 number of neighbors, and the weights given by distance.

In [42]:

```
from sklearn.neighbors import KNeighborsClassifier
#search for optimal parameters

grid_param = {
    'n_neighbors': [3, 5, 7, 11, 19, 23],
    'weights':['uniform', 'distance'],
    'metric': ['euclidean', 'manhattan']
}

#use gridsearch to test all values for n_neighbors
knn_best = GridSearchCV(KNeighborsClassifier(), grid_param, verbose = False, cv=
5, n_jobs = -1)
#fit model to data
knn_best_result = knn_best.fit(X_test_scaled, y_test)

#check top performing n_neighbors value
print(knn_best.best_params_)
print(knn_best.best_score_)
```

```
{'metric': 'euclidean', 'n_neighbors': 23, 'weights': 'distance'}
0.4381070162420645
```

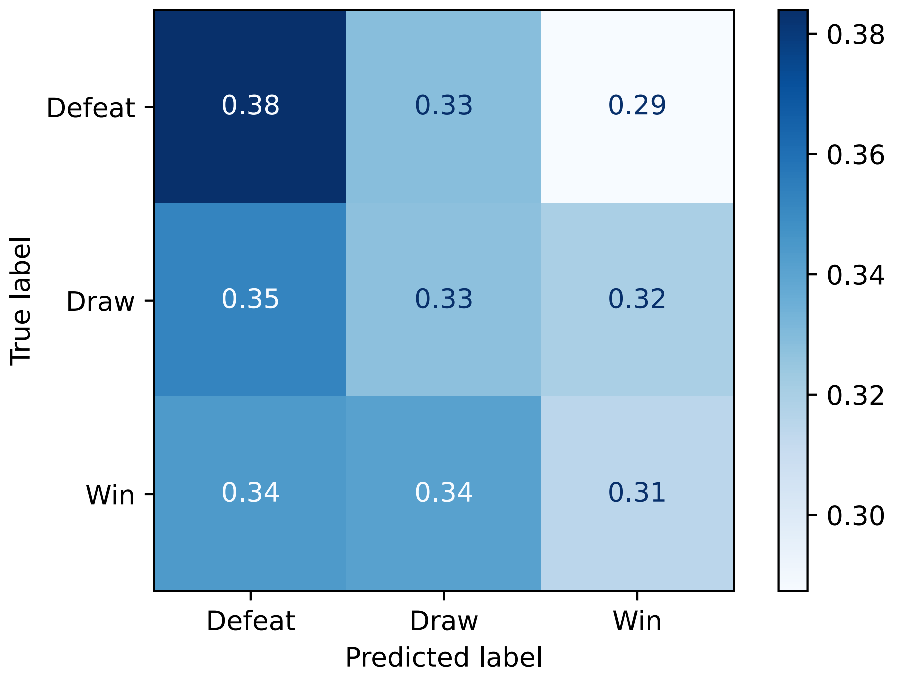
The best KNN for this situation scores is summarized next.

In [44]:

```
#best KNN

knnfinal = KNeighborsClassifier(n_neighbors = 23, metric='euclidean', weights='d
istance').fit(X_train_scaled,y_train)

CM_B = plot_confusion_matrix(knnfinal,X_test_scaled, y_test, normalize="true", c
map=plt.cm.Blues)
CM_B
```

Out[44]:

```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x
7f6b62febb50>
```



In [45]:

```
from sklearn.metrics import accuracy_score
#KNN measures
y_pred = knnfinal.predict(X_test_scaled)

knn_report = classification_report(y_test, y_pred, output_dict="true")
```

4. Neural Network

In this section we will implpement a Neural Network model.

The process of tunning the MLPClassifier was mostly done using the RandomizedSearchCV class, analogous to the GridSearchCV class used throughout this notebook. A large number of parameters was tested, including the size and number of the hidden layers ('hidden_layer_sizes') the activation function ('activation'), the solver for weight optimization ('solver'), a regularization parameter ('alpha'), the variation in learning rate ('learning_rate'), and even other parameters that, by their sheer number, forced us to make use of the RandomizedSearchCV as opposed to the GridSearchCV. Adding to this, the MLPClassifier was what initially led us to balance the dataset as it very quickly took advantage of this imbalance by predicting only Home Team Wins.

The following implementation of the GridSearchCV only uses some of the best parameters we could find during our testing process, to cut down on execution time for this notebook. The results were then compared to the models above.

In [47]:

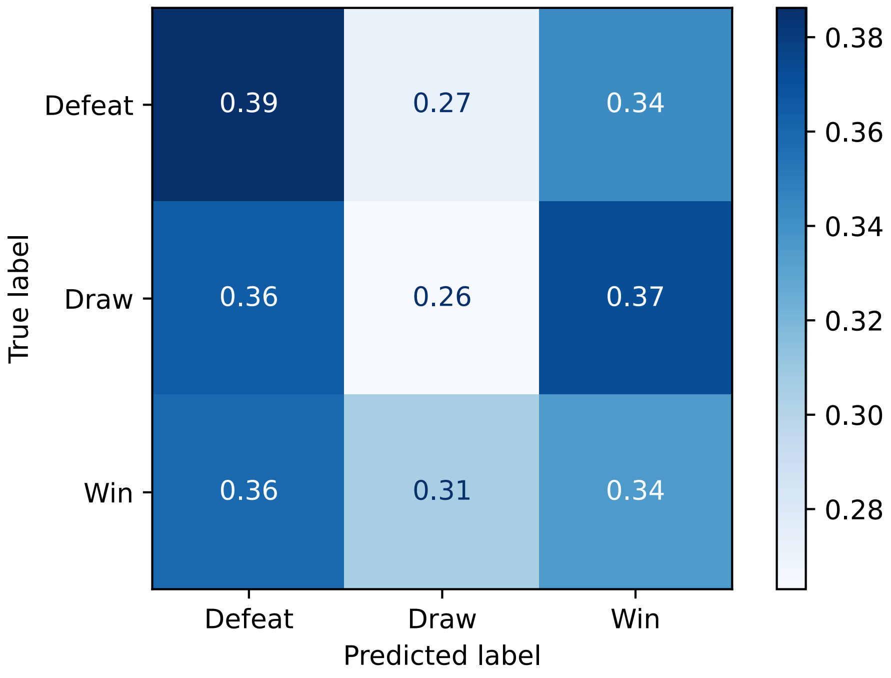
```
from sklearn.neural_network import MLPClassifier


# Utility function to report best scores
def report(results, n_top=3):
    for i in range(1, n_top + 1):
        candidates = np.flatnonzero(results['rank_test_score'] == i)
        for candidate in candidates:
            print("Model with rank: {0}".format(i))
            print("Mean validation score: {0:.3f} (std: {1:.3f})"
                  .format(results['mean_test_score'][candidate],
                          results['std_test_score'][candidate]))
            print("Parameters: {0}".format(results['params'][candidate]))
            print("")

parameter_space = {
    'hidden_layer_sizes': [(10,7), (15,10,7)],
    'activation': ['tanh', 'relu'],
    'solver': ['adam'],
    'alpha': [0.0001, 0.05],
    'learning_rate': ['adaptive'],
}

clf = MLPClassifier(verbose=0, random_state=21, learning_rate=1e-05)


grid_search = GridSearchCV(clf, parameter_space, cv=5)

start = time()
grid_search.fit(X_train_scaled, y_train)

report(grid_search.cv_results_)
```

```
Model with rank: 1
Mean validation score: 0.384 (std: 0.036)
Parameters: {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_size
s': (15, 10, 7), 'learning_rate': 'adaptive', 'solver': 'adam'}

Model with rank: 2
Mean validation score: 0.382 (std: 0.018)
Parameters: {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_si
zes': (15, 10, 7), 'learning_rate': 'adaptive', 'solver': 'adam'}

Model with rank: 3
Mean validation score: 0.381 (std: 0.022)
Parameters: {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_si
zes': (10, 7), 'learning_rate': 'adaptive', 'solver': 'adam'}
```

In [48]:

```
nn_model = grid_search.best_estimator_

print (grid_search.best_score_, grid_search.best_params_)

disp = plot_confusion_matrix(nn_model, X_test_scaled, y_test, normalize="true",
cmap=plt.cm.Blues)
```

```
0.383531746031746 {'activation': 'relu', 'alpha': 0.05, 'hidden_laye
r_sizes': (15, 10, 7), 'learning_rate': 'adaptive', 'solver': 'ada
m'}
```



In [49]:

```
y_pred = nn_model.predict(X_test_scaled)
nn_model_report = classification_report(y_test, y_pred, output_dict="true")
```

5. Model Comparison

In this section we will compare the results of the three models in sections 2, 3 and 4.

Our views on the difficulty of predicting the outcome of matches stem from more than the lack of accuracy of our models. We decided to compare our ability to predict the matches with the ability of the dataset's betting odds, aiming to see how inconsistent professional agents are in this field. The code cell that follows refers to that analysis.

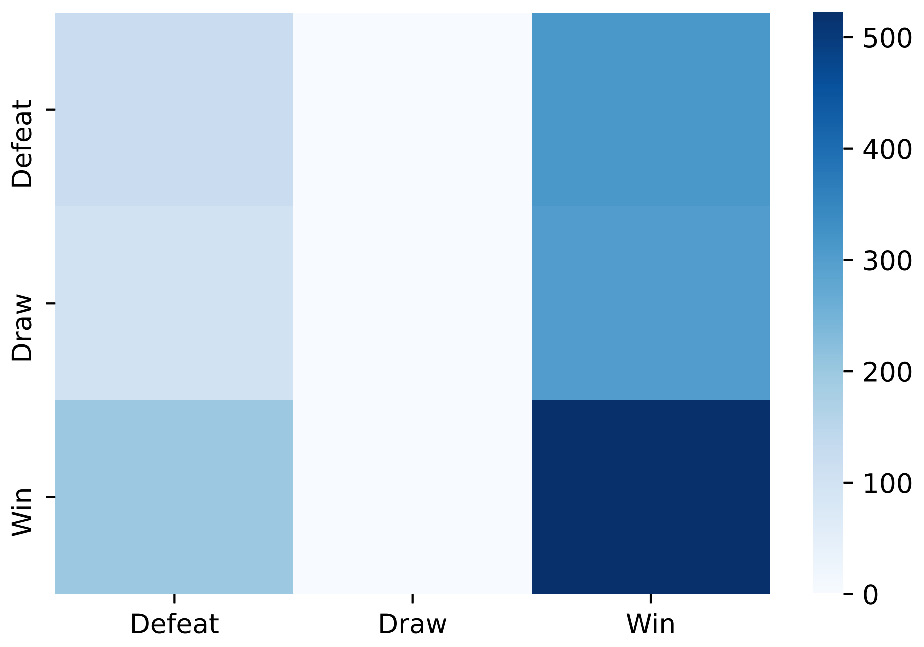
In [52]:

```
def translate(h,a,d):
    if(h<a and h<d):
        return "Win"
    if(d<a and d<h):
        return "Draw"
    else:
        return "Defeat"


bet_predictions = map(translate, X_test.get('mean_bets_home'), X_test.get('mean_
bets_away'), X_test.get('mean_bets_draw'))


class_names= {"Defeat","Draw", "Win"}

bet_pred= list(bet_predictions)
print(confusion_matrix(y_test, bet_pred))

bets_report = classification_report(y_test, bet_pred, output_dict="true")
print(bets_report)

index = ['Defeat','Draw','Win']
columns = ['Defeat','Draw','Win']
cm_df = pd.DataFrame(confusion_matrix(y_test, bet_pred),columns,index)

sns.heatmap(cm_df, cmap="Blues")
```

```
[[121   0 314]
 [101   0 302]
 [199   0 522]]
{'Defeat': {'precision': 0.28741092636579574, 'recall': 0.2781609195
4022987, 'f1-score': 0.2827102803738318, 'support': 435}, 'Draw':
{'precision': 0.0, 'recall': 0.0, 'f1-score': 0.0, 'support': 403},
'Win': {'precision': 0.45869947275922673, 'recall': 0.72399445214979
2, 'f1-score': 0.5615922538999462, 'support': 721}, 'accuracy': 0.41
24438742783836, 'macro avg': {'precision': 0.24870346637500748, 'rec
all': 0.3340517905633406, 'f1-score': 0.2814341780912593, 'support':
1559}, 'weighted avg': {'precision': 0.29233231098686574, 'recall':
0.4124438742783836, 'f1-score': 0.3386061494704798, 'support': 155
9}}
```

Out[52]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f6b60166810>
```

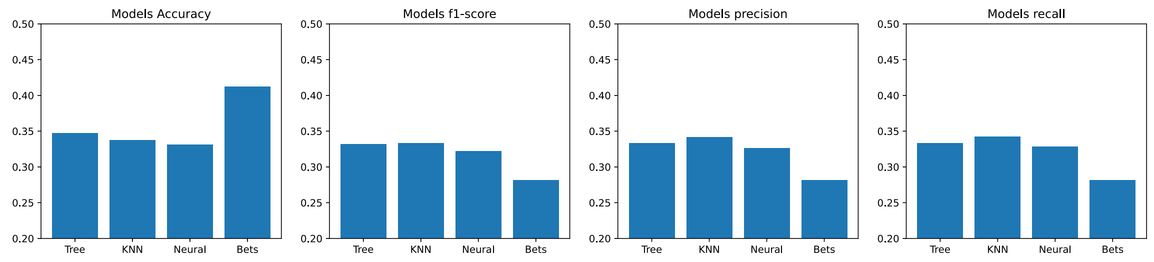


As can be seen in the graphics below the model with the best scoring is the decision tree. It is the model that has better accuracy, meaning is the one that makes more correct predictions. It also makes fewer errors.

Obs: It is important to mention that we are using a random splitter to divide the data into a training and test set and bacause of that we can't say for sure which algorithm is the best one since the results change through executions. This can be considered good in the sense that all algorithms have been implemented correctly and therefore produce very similar results.

In [54]:

```
columns = ('Tree', 'KNN', 'Neural', 'Bets')

plt.figure(figsize=(20, 4))

ax1 = plt.subplot(141)
ax1.set_ylim([0.2, 0.5])
plt.bar(columns, (tree_report['accuracy'], knn_report['accuracy'], nn_model_repo
rt['accuracy'], bets_report['accuracy']))
plt.title('Models Accuracy')
ax2 = plt.subplot(142)
ax2.set_ylim([0.2, 0.5])
plt.bar(columns, (tree_report['macro avg']['f1-score'], knn_report['macro avg'][
'f1-score'], nn_model_report['macro avg']['f1-score'], bets_report['macro avg'][
'f1-score']))
plt.title('Models f1-score')
ax3 = plt.subplot(143)
ax3.set_ylim([0.2, 0.5])
plt.bar(columns, (tree_report['macro avg']['precision'], knn_report['macro avg']
['precision'], nn_model_report['macro avg']['precision'], bets_report['macro av
g']['f1-score']))
plt.title('Models precision')
ax4 = plt.subplot(144)
ax4.set_ylim([0.2, 0.5])
plt.bar(columns, (tree_report['macro avg']['recall'], knn_report['macro avg']['r
ecall'], nn_model_report['macro avg']['recall'], bets_report['macro avg']['f1-sc
ore']))
plt.title('Models recall')
plt.show()

print('Models acuracy: ', end='')
print(tree_report['accuracy'], knn_report['accuracy'], nn_model_report['accurac
y'],  bets_report['accuracy'])
print('Models f1-score: ', end = '')
print(tree_report['macro avg']['f1-score'], knn_report['macro avg']['f1-score'],
nn_model_report['macro avg']['f1-score'], bets_report['macro avg']['f1-score'])
print('Models precision: ', end='')
print(tree_report['macro avg']['precision'], knn_report['macro avg']['precision'
], nn_model_report['macro avg']['precision'], bets_report['macro avg']['f1-scor
e'])
print('Models recall: ', end='')
print(tree_report['macro avg']['recall'], knn_report['macro avg']['recall'], nn_
model_report['macro avg']['recall'], bets_report['macro avg']['recall'])
```



```
Models acuracy: 0.3470173187940988 0.3373957665169981 0.330981398332
26425 0.4124438742783836
Models f1-score: 0.33194291746807253 0.33337055368548585 0.322272119
2221735 0.2814341780912593
Models precision: 0.333458999093674 0.3417042879545938 0.32624714791
884896 0.2814341780912593
Models recall: 0.33310384623494116 0.3420973232003827 0.328293043141
25635 0.3340517905633406
```

5. Conclusions

In this section we will draw conclusions.

Football is a very unpredictable sport. Although the existence of some trends, like the home victory, the victory of the strongest team, among others, there's an unpredictability that makes the prevision of the outcome a very difficult task, and the game so beautiful. There are so many different factors that can contribute to the match outcome. Even actors who have a large incentive to predict reliably the outcome of matches struggle to do so, as we can see from our analysis of the betting odds in the dataset.

It was clear for us since the early stages of the development process of this notebook that achieving satisfactory results would require surpassing the barrier of the biased dataset for Home team Wins. As the home team wins 47% of the games, our models found it particularly easy to achieve accuracy scores of 0.47, by predicting only wins. Removing this imbalance in the dataset pushed back our accuracy scores significantly. Still, we believe that by doing this we achieved more compelling results, interesting comparisons, and valuable experience tunning these models.

References

1. Data set (partials/pre-processing.ipynb)
2. KNN Classification (https://www.datacamp.com/community/tutorials/k-nearest-neighbor-classification-scikit-learn)
3. Scikit Learn videos (https://github.com/justmarkham/scikit-learn-videos)
4. Class weight vs over sampling (https://stackoverflow.com/questions/55657807/class-weights-vs-under-oversampling)
5. Decision Tree Classifier (https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html)
6. SMOTE - over sampling (https://imbalanced-learn.readthedocs.io/en/stable/generated/imblearn.over_sampling.SMOTE.html)
7. Neural Networks - Supervised (https://scikit-learn.org/stable/modules/neural_networks_supervised.html)
8. Confusion matrix visualization (https://medium.com/@dtuk81/confusion-matrix-visualization-fc31e3f30fea)
9. KNN hyperparameter tunning (https://medium.com/datadriveninvestor/k-nearest-neighbors-in-python-hyperparameters-tuning-716734bc557f)
10. Decision tree and imbalanced data (https://medium.com/swlh/tree-based-machine-learning-models-for-handling-imbalanced-datasets-26560b5865f6)