CISC 451/839 Topics in Data Analytics

Course Project - Using Feature Engineering and Supervised Learning to Predict Game Results in Professional Hockey

Gavin McClelland - 10211444

Marshall Cunningham - 20249991

The objectives of this notebook are as follows:

- Build on top of the previous approaches included in the midterm submission which featured extensive EDA and simple model construction to justify the validity of the project (not trivial to understand/predict game results if information about the score is omitted)
- Using performance trends from previous games, aim to develop models to predict the result of a game before it happens
    - we are only concerned with the binary classification task of predicting wins and losses, not the condition of victory (such as winning in regulation, overtime, or in a shootout)

Contents

The analytics process contained in this notebook is as follows:

1. Read-in Data
2. Create features in the range [0,1]
3. Min-Max normalization
4. Approach #1 - Feature selection, training, cross-validation, testing, and model comparison
5. Approach #2 - Feature selection, training, cross-validation, testing, model comparison, and hyperparameter tuning
6. Approach #3 - Iterating off of Approach #2, and experimenting with XGBoost: training, cross validation, and hyperparameter tuning

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score,confusion_matrix,plot_confusion_matrix,roc_c
from make_confusion_matrix import make_confusion_matrix
from plot_roc_curve import plot_roc_curve
from validate_model import validate_model
%cd "C:\Users\gmcclelland\Desktop\Misc School Stuff\repo\CISC451\project\data"
```

```
# %cd "<Your Working Directory Here>"
%matplotlib inline
```

C:\Users\gmcclelland\Desktop\Misc School Stuff\repo\CISC451\project\data

In [2]:
```
# Importing datasets, from the previous snapshot of work done for the midterm submissio
team_stats = pd.read_csv('game_teams_stats.csv')
game_stats = pd.read_csv("all_teams.csv")
```

In [3]:
```
# For confusion matrix and roc viz
group_names = ["True Neg","False Pos","False Neg","True Pos"]
categories = ["Loss", "Win"]
```

In [4]:
```
game_stats.drop(game_stats[game_stats['season'] < 2010].index, inplace=True)
game_stats.drop(game_stats[game_stats['playoffGame'] == 1].index, inplace=True)
game_stats.drop(game_stats[game_stats['situation'] != 'all'].index, inplace=True)
```

In [5]:
```
team_stats = team_stats[['game_id', 'HoA', 'won']]
team_stats.rename(
    columns={
        "game_id": "gameId",
        "HoA": "home_or_away",
        "won": "WON"
    }, inplace=True
)
team_stats['home_or_away'] = team_stats['home_or_away'].str.upper()
# Inner join datasets to get an accurate label
game_stats = pd.merge(game_stats, team_stats,how='inner',on=['gameId','home_or_away'])

# Code the WON and home_or_away columns into integers
game_stats['WON'] = np.where(game_stats['WON'] == True, 1, 0)
game_stats['home_or_away'] = np.where(game_stats['home_or_away'] == 'HOME', 1, 0)

game_stats.groupby('home_or_away').size()
```

Out[5]:
```
home_or_away
0    10748
1    10536
dtype: int64
```

In [6]:
```
game_stats.groupby('WON').size()
```

Out[6]:
```
WON
0    10652
1    10632
dtype: int64
```

In [7]:
```
game_stats.shape
```

Out[7]:
```
(21284, 112)
```

In [8]:
```
games = game_stats.groupby(['gameId','home_or_away','WON']).size().reset_index().rename
# Find all games that have both teams either wining or losing
duplicate_games = games.loc[games['count'] > 1]
```

In [9]:
```
duplicate_games
```

Out[9]:

	gameId	home_or_away	WON	count

	gameId	home_or_away	WON	count
2568	2011020055	0	0	2
2611	2011020077	0	1	2
2636	2011020090	0	1	2
2695	2011020120	0	1	2
2724	2011020135	0	0	2
...	...	...	...	...
8489	2013021116	0	1	2
8532	2013021138	0	1	2
8571	2013021158	0	1	2
8690	2013021218	0	1	2
8713	2013021230	0	0	2

106 rows × 4 columns

In [10]:
```
# Remove duplicates
game_stats.drop(game_stats.loc[game_stats.gameId.isin(duplicate_games.gameId.values)].i
game_stats.groupby('WON').size()
```

Out[10]:
```
WON
0    10536
1    10536
dtype: int64
```

In [11]:
```
game_stats.drop(columns=['name',
                         'playerTeam',
                         'opposingTeam',
                         'gameDate',
                         'position',
                         'situation',
                         'iceTime',
                         'playoffGame'], inplace=True)
game_stats.head()
```

Out[11]:

	team	season	gameId	home_or_away	xGoalsPercentage	corsiPercentage	fenwickPercentage	x(
0	NYR	2010	2010020013	0	0.6494	0.4724	0.4545	
1	NYR	2010	2010020028	0	0.4394	0.5526	0.5488	
2	NYR	2010	2010020049	1	0.4434	0.4602	0.4787	
3	NYR	2010	2010020070	1	0.3698	0.5772	0.5217	
4	NYR	2010	2010020083	0	0.4983	0.4622	0.5584	

5 rows × 104 columns

In [12]:
```
# most features have a 'for' and 'against' pair, so we will combine them into a ratio i
entries = []
```

```
for column in game_stats.columns.tolist():
    if column[-3:] == 'For':
        entries.append(column[:-3])
# NOTE: This took awhile to figure out, but if the stat has '0' in the for AND against
for x in entries:
    game_stats[f'{x}Ratio'] = game_stats.apply(lambda row: row[f'{x}For'] / (row[f'{x}F
    game_stats.drop(columns=[f'{x}For',f'{x}Against'],inplace=True)
```

In [13]:

```
# note that many of the probabalistic statistics (such as xGoalsPercentage) are not alw
game_stats.loc[(game_stats.xGoalsPercentage < 0.5) & (game_stats.WON == 1)].shape[0]
```

Out[13]: 3884

In [14]:

```
cols_to_normalize = game_stats.drop(columns=['team','gameId','home_or_away','WON','seas
# Min-Max Normalization
for column in cols_to_normalize:
        game_stats[column] = (game_stats[column] - game_stats[column].min()) / (game_st
game_stats
```

Out[14]:

	team	season	gameId	home_or_away	xGoalsPercentage	corsiPercentage	fenwickPercentag
0	NYR	2010	2010020013	0	0.679610	0.452821	0.41983
1	NYR	2010	2010020028	0	0.427146	0.589915	0.58597
2	NYR	2010	2010020049	1	0.431955	0.431966	0.46247
3	NYR	2010	2010020070	1	0.343472	0.631966	0.53823
4	NYR	2010	2010020083	0	0.497956	0.435385	0.60288
...	...	...	...	...	...	...	.
21279	L.A	2018	2018021214	1	0.605795	0.358803	0.33897
21280	L.A	2018	2018021228	1	0.455157	0.460855	0.44820
21281	L.A	2018	2018021238	0	0.371964	0.264444	0.18851
21282	L.A	2018	2018021256	0	0.427987	0.639145	0.67283
21283	L.A	2018	2018021270	1	0.681534	0.478803	0.47022

21072 rows × 56 columns

In [15]:

```
# rearranging columns to beginning of df for organization purposes
cols = game_stats.drop(columns=['WON']).columns.tolist()
cols = ['WON'] + cols
game_stats = game_stats.reindex(columns=cols)
game_stats.head()
```

Out[15]:

	WON	team	season	gameId	home_or_away	xGoalsPercentage	corsiPercentage	fenwickPercent
0	1	NYR	2010	2010020013	0	0.679610	0.452821	0.419
1	0	NYR	2010	2010020028	0	0.427146	0.589915	0.585
2	0	NYR	2010	2010020049	1	0.431955	0.431966	0.462
3	0	NYR	2010	2010020070	1	0.343472	0.631966	0.538

	WON	team	season	gameId	home_or_away	xGoalsPercentage	corsiPercentage	fenwickPercent
4	1	NYR	2010	2010020083	0	0.497956	0.435385	0.602

5 rows × 56 columns

Approach #1 - Predicting the outcome using advanced statistics from the game (with no knowledge of the opponent)

Feature Selection

We already have quite a few features (55), so before looking at previous games to predict the result of a game before it happens, let's find out which of these features are of any significance

In [16]:

```
gs_cpy = game_stats.copy()
game_stats.columns.tolist()
```

Out[16]:

```
['WON',
 'team',
 'season',
 'gameId',
 'home_or_away',
 'xGoalsPercentage',
 'corsiPercentage',
 'fenwickPercentage',
 'xOnGoalRatio',
 'xGoalsRatio',
 'xReboundsRatio',
 'xFreezeRatio',
 'xPlayStoppedRatio',
 'xPlayContinuedInZoneRatio',
 'xPlayContinuedOutsideZoneRatio',
 'flurryAdjustedxGoalsRatio',
 'scoreVenueAdjustedxGoalsRatio',
 'flurryScoreVenueAdjustedxGoalsRatio',
 'shotsOnGoalRatio',
 'missedShotsRatio',
 'blockedShotAttemptsRatio',
 'shotAttemptsRatio',
 'goalsRatio',
 'reboundsRatio',
 'reboundGoalsRatio',
 'freezeRatio',
 'playStoppedRatio',
 'playContinuedInZoneRatio',
 'playContinuedOutsideZoneRatio',
 'savedShotsOnGoalRatio',
 'savedUnblockedShotAttemptsRatio',
 'penaltiesRatio',
 'penalityMinutesRatio',
 'faceOffsWonRatio',
 'hitsRatio',
 'takeawaysRatio',
 'giveawaysRatio',
 'lowDangerShotsRatio',
 'mediumDangerShotsRatio',
```

```
 'highDangerShotsRatio',
 'lowDangerxGoalsRatio',
 'mediumDangerxGoalsRatio',
 'highDangerxGoalsRatio',
 'lowDangerGoalsRatio',
 'mediumDangerGoalsRatio',
 'highDangerGoalsRatio',
 'scoreAdjustedShotsAttemptsRatio',
 'unblockedShotAttemptsRatio',
 'scoreAdjustedUnblockedShotAttemptsRatio',
 'dZoneGiveawaysRatio',
 'xGoalsFromxReboundsOfShotsRatio',
 'xGoalsFromActualReboundsOfShotsRatio',
 'reboundxGoalsRatio',
 'totalShotCreditRatio',
 'scoreAdjustedTotalShotCreditRatio',
 'scoreFlurryAdjustedTotalShotCreditRatio']
```

Feature Importance

Fitting a simple logistic regression model to our features to resolve feature "importances"

In [17]:

```
# Dropping categorical features, and also dropping goalsRatio, which explicitly represe
X = game_stats.drop(columns=['team','gameId','season','goalsRatio','WON'])
Y = game_stats['WON']
```

In [18]:

```
lr = LogisticRegression()
lr.fit(X,Y)
```

Out[18]:

```
LogisticRegression()
```

In [19]:

```
np.abs(lr.coef_[0])
```

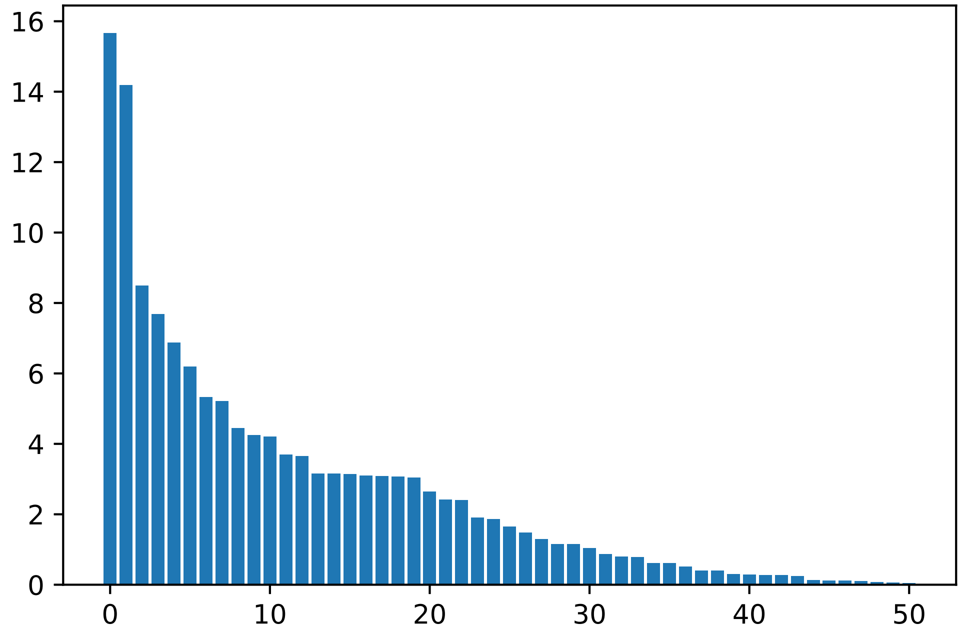
Out[19]:

```
array([ 1.30067998,  1.14947851,  0.40772228,  3.15203373,  1.64871398,
        1.14930468,  2.40177143,  3.08730117,  0.29300966,  0.79503016,
        6.19467885,  1.48468111,  1.03705138,  3.66050981,  8.49545978,
        3.69922804,  3.14921015,  0.41048741,  0.07241704,  0.87096748,
        3.04421264,  0.61384293,  6.87492843,  5.32691102, 15.66374217,
       14.1866723 ,  0.13297455,  0.78056003,  0.11741237,  0.28144897,
        0.24926202,  0.10171299,  1.86593555,  0.26980632,  0.11462508,
        1.91491363,  2.641267  ,  0.61996486,  4.25167025,  4.20884003,
        3.10096184,  5.21708261,  3.15325813,  7.69295171,  0.0671291 ,
        3.06895097,  0.29923454,  0.50998097,  0.04915655,  2.42605623,
        4.44555488])
```

Below, we plot our feature coefficients in descending order to observe where a drop-off occurs

In [20]:

```
importance = np.abs(lr.coef_[0])
sorted_importance = -np.sort(-importance)
# for i,v in enumerate(importance):
#       print('Feature: %0d, Score: %.5f' % (i,v))
plt.bar([x for x in range(len(sorted_importance))], sorted_importance)
plt.show()
```



From the plot above, we are choosing the top 19 features as there is an observable drop-off in performance afterwards

Next, we find our columns to drop from our dataset, so we are left with our 19 most important features

In [21]:
```
cols_to_drop = []
thresh = sorted_importance[18]
for n,val in enumerate(importance):
    print(val)
    if val < thresh:
        cols_to_drop.append(game_stats.columns[n+4])
game_stats.drop(cols_to_drop,axis=1,inplace=True)
```

```
1.300679979253326
1.1494785144229598
0.40772228349505446
3.1520337255221698
1.6487139824116965
1.149304675453365
2.4017714304653213
3.0873011708704405
0.293009656753454
0.7950301629443306
6.194678851049144
1.4846811140311573
1.0370513764511289
3.6605098107327803
8.495459784404872
3.6992280363393184
3.1492101498037743
0.410487414927188
0.07241703927382163
0.8709674765757347
3.0442126407033894
0.6138429279779819
6.874928430440959
5.326911022165617
15.663742172367337
14.186672298899346
```

```
0.13297455017178078
0.7805600279267647
0.11741237491660245
0.2814489675492729
0.24926201774889323
0.10171298926996661
1.865935550022819
0.26980632491692746
0.1146250815592999
1.9149136260241886
2.641266996560694
0.6199648582306234
4.251670249401393
4.208840029894478
3.1009618365318974
5.217082606095728
3.153258131662515
7.692951710602552
0.06712910154599269
3.06895096643505
0.29923454262203364
0.5099809722015298
0.04915654712883549
2.4260562251422844
4.445554879145968
```

In [22]:

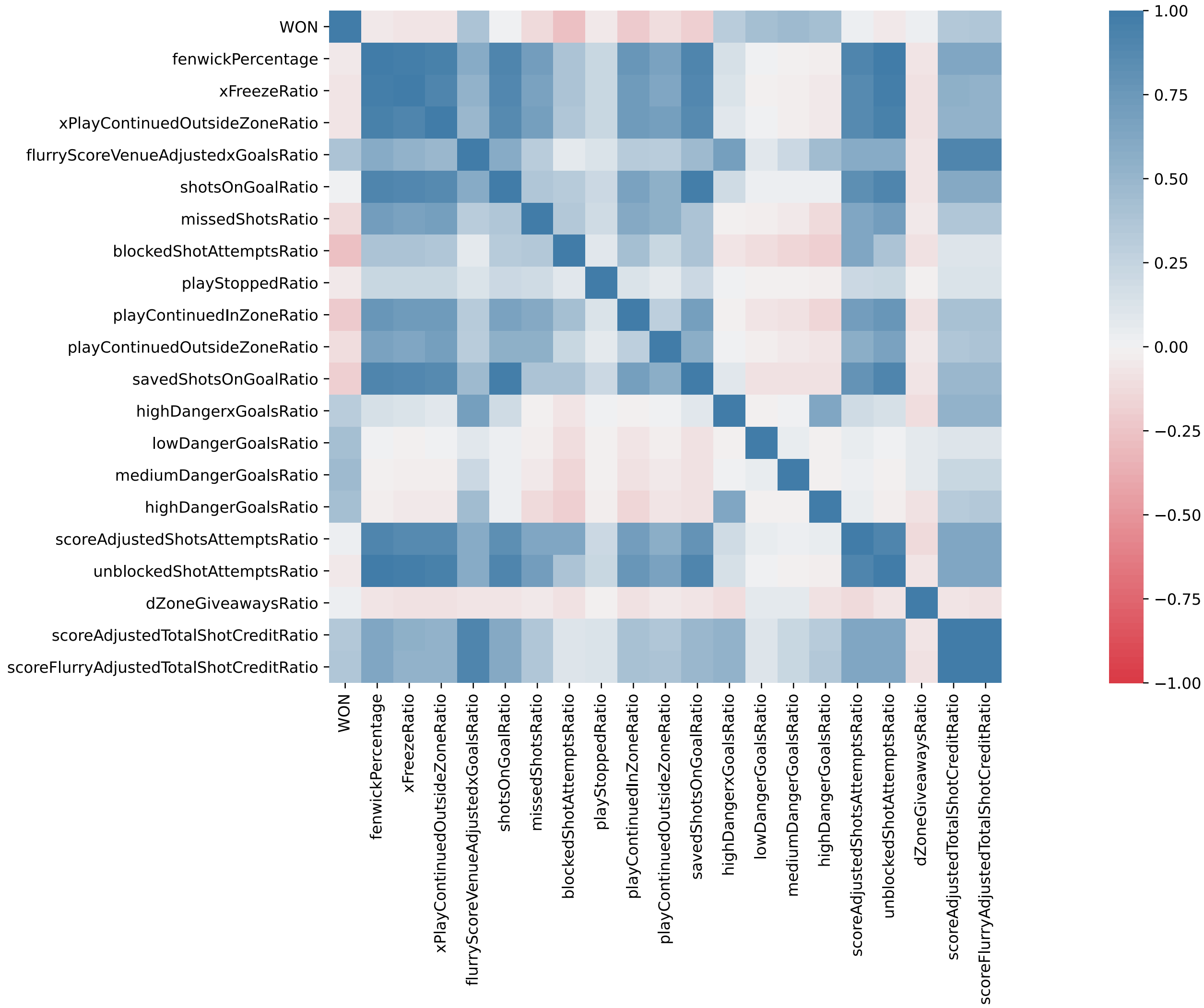
```python
# Verifying that feature selection worked as expected
print(cols_to_drop)
```

```
['home_or_away', 'xGoalsPercentage', 'corsiPercentage', 'xOnGoalRatio', 'xGoalsRatio',
'xReboundsRatio', 'xPlayStoppedRatio', 'xPlayContinuedInZoneRatio', 'flurryAdjustedxGoal
sRatio', 'scoreVenueAdjustedxGoalsRatio', 'shotAttemptsRatio', 'goalsRatio', 'reboundsRa
tio', 'reboundGoalsRatio', 'freezeRatio', 'savedUnblockedShotAttemptsRatio', 'penaltiesR
atio', 'penalityMinutesRatio', 'faceOffsWonRatio', 'hitsRatio', 'takeawaysRatio', 'givea
waysRatio', 'lowDangerShotsRatio', 'mediumDangerShotsRatio', 'highDangerShotsRatio', 'lo
wDangerxGoalsRatio', 'mediumDangerxGoalsRatio', 'scoreAdjustedUnblockedShotAttemptsRati
o', 'xGoalsFromxReboundsOfShotsRatio', 'xGoalsFromActualReboundsOfShotsRatio', 'reboundx
GoalsRatio', 'totalShotCreditRatio']
```

Next, we create a correlation heatmap to visualize any redundant/synonymous features, along with those that are too highly correlated with the outcome label 'WON'

In [23]:

```python
corr = game_stats.drop(columns=['team','gameId','season']).corr()
fig, ax = plt.subplots(figsize=(25,8))
ax = sns.heatmap(
    corr,
    vmin=-1, vmax=1,
    square=True,
    cmap=sns.diverging_palette(10,240,n=100),
    ax=ax
)
```



The only duplicate features are different versions of the same stat, which can be seen near the bottom (different versions of shotCredit)

- So, we keep the flurry adjusted stat as flurry is more repeatable and regarded as having more predictive power (see report)

In [24]:
```
game_stats.drop(columns=['scoreAdjustedTotalShotCreditRatio'],inplace=True)
```

In [25]:
```
# dimensions of the dataset after selecting features
game_stats.shape
```

Out[25]: (21072, 23)

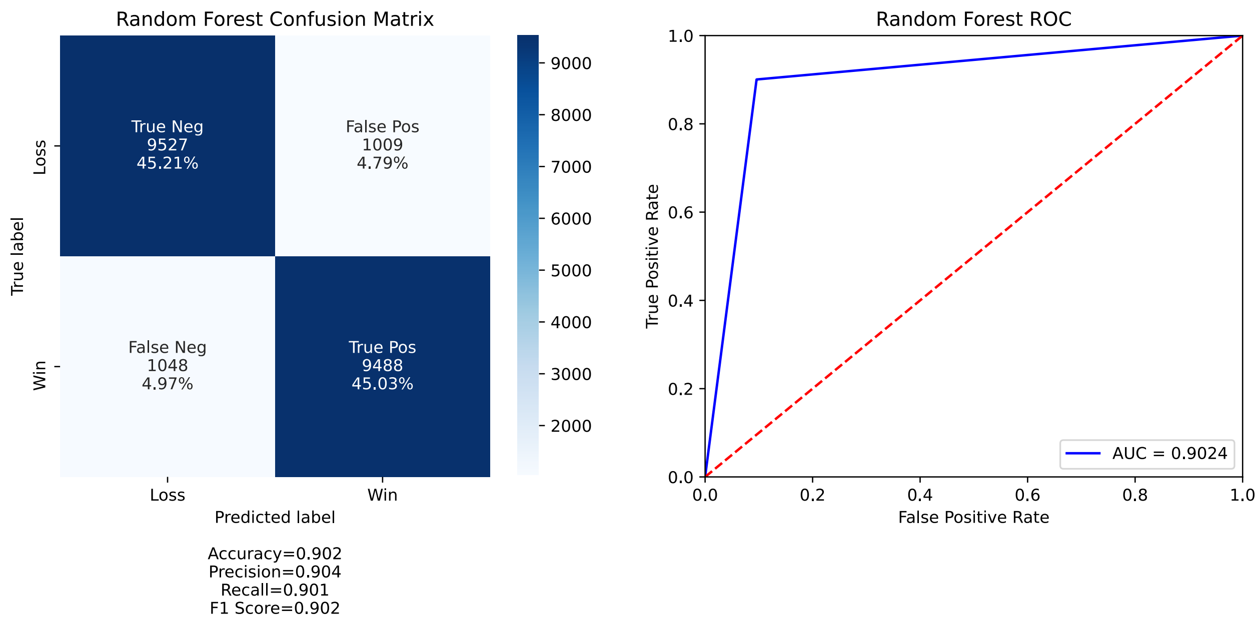
In [26]:
```
from sklearn import svm
from sklearn.model_selection import cross_val_score,cross_validate
from sklearn.metrics import accuracy_score
feat = game_stats.drop(columns=['team','gameId','WON'])
label = game_stats['WON']
# Train on 2010 - 2017, test on 2018
x_train = feat.loc[feat.season < 2018].drop(columns=['season'])
x_test = feat.loc[feat.season == 2018].drop(columns=['season'])
y_train = game_stats.loc[game_stats.season < 2018]['WON'].drop(columns=['season'])
y_test = game_stats.loc[game_stats.season == 2018]['WON'].drop(columns=['season'])
# x_train,x_test,y_train,y_test = train_test_split(feat,label,test_size=0.2) # test on
```

```
# Using a random forest classifier as a benchmark
clf = RandomForestClassifier()
scores = cross_validate(clf, x_train, y_train, scoring='accuracy',cv=10,return_estimato
```

In [27]:
```
best_score = np.argmax(scores['test_score'])
estimators = scores['estimator']
clf = estimators[best_score]
pred = clf.predict(x_test)
accuracy_score(y_test,pred)
```

Out[27]: 0.9346970889063729

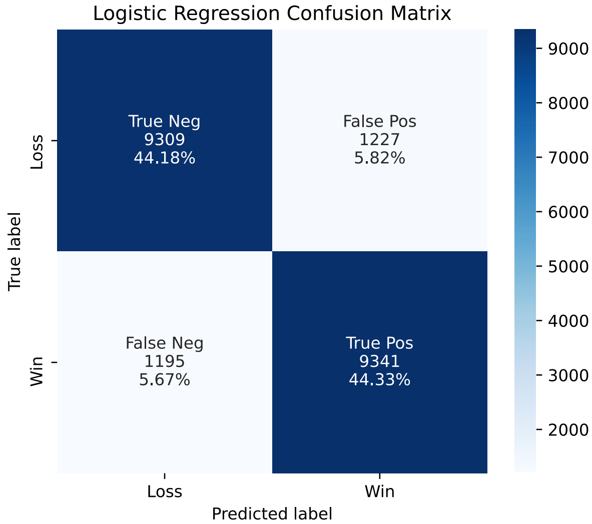
In [28]:
```
validate_model(clf,feat,label,'Random Forest', group_names=group_names, categories=cate
```

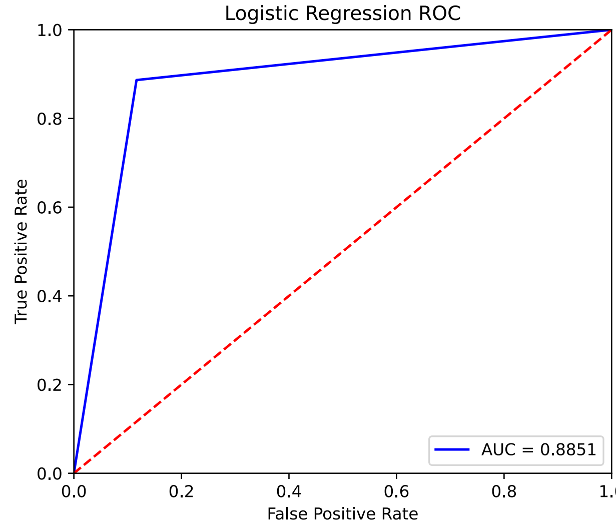


<Figure size 432x288 with 0 Axes>

In [29]:
```
lrc = LogisticRegression()
knnc = KNeighborsClassifier() # defaults to 5-NN
dtc = DecisionTreeClassifier()
```

In [30]:
```
validate_model(lrc, feat, label, 'Logistic Regression', group_names=group_names, catego
```

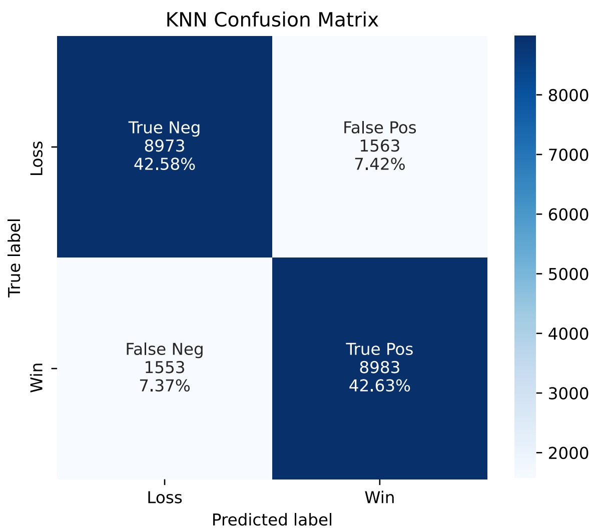


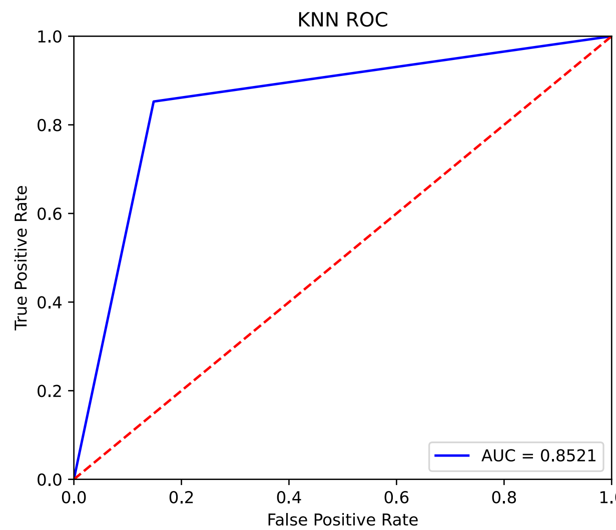


Accuracy=0.885
Precision=0.884
Recall=0.887
F1 Score=0.885

<Figure size 432x288 with 0 Axes>

In [31]:
```
validate_model(knnc, feat, label, 'KNN', group_names=group_names, categories=categories
```

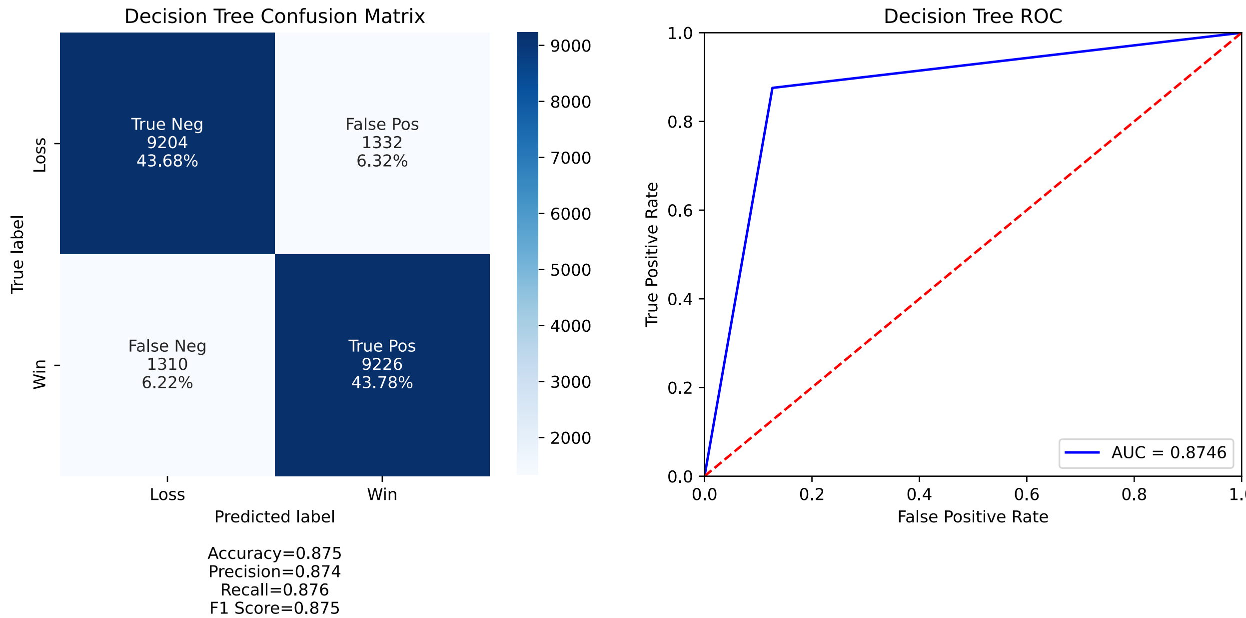




Accuracy=0.852
Precision=0.852
Recall=0.853
F1 Score=0.852

<Figure size 432x288 with 0 Axes>

In [32]:
```
validate_model(dtc, feat, label, 'Decision Tree', group_names=group_names, categories=c
```




```
<Figure size 432x288 with 0 Axes>
```

Approach #2 - Creating Historical Features (again, with no knowledge of the opponent)

With the objective of predicting outcome before a game has occurred, it is important to have information about the team **before** the game takes place

- So, below we create rolling averages from the previous 1, and 3 games not including the current game. This will create historical versions of the previously selected features for training our models

In [33]:
```
gs_cpy # using the deep copy of the dataframe created prior to feature selection in app
```

Out[33]:

	WON	team	season	gameId	home_or_away	xGoalsPercentage	corsiPercentage	fenwickPe
0	1	NYR	2010	2010020013	0	0.679610	0.452821	
1	0	NYR	2010	2010020028	0	0.427146	0.589915	
2	0	NYR	2010	2010020049	1	0.431955	0.431966	
3	0	NYR	2010	2010020070	1	0.343472	0.631966	
4	1	NYR	2010	2010020083	0	0.497956	0.435385	
...	...	...	...	...	...	...	...	
21279	1	L.A	2018	2018021214	1	0.605795	0.358803	
21280	0	L.A	2018	2018021228	1	0.455157	0.460855	
21281	1	L.A	2018	2018021238	0	0.371964	0.264444	
21282	0	L.A	2018	2018021256	0	0.427987	0.639145	
21283	1	L.A	2018	2018021270	1	0.681534	0.478803	

21072 rows × 56 columns

In [34]:
```
window_lengths = (1, 3, 5, 10)
new_cols = ['team','home_or_away','gameId','season']
# Programatically create column names for empty dataframe which will be appended to lat
for col in gs_cpy.columns:
    if col in new_cols:
        continue
    if col == 'WON':
        new_cols.append(col)
    for length in window_lengths:
        new_cols.append(f'{col}Prev{length}')
# Create empty dataframe to store new features as they are created
new_df = pd.DataFrame(columns=new_cols)
# Treat each season as a seperate dataset for rolling windows
for season in gs_cpy.season.sort_values(ascending=True).unique().tolist():
    season_df = gs_cpy.loc[gs_cpy['season'] == season]
    # Seperate teams so that rolling windows are unique to each team
    for team in gs_cpy.team.sort_values(ascending=True).unique().tolist():
        df = season_df.loc[season_df['team'] == team]
        df.sort_values('gameId', ascending=True, inplace=True)
        for col in df.columns:
            if col != 'WON' and col in new_cols:
                continue
            # Create a rolling average for each window length
            for length in window_lengths:
                # shift one record up so the current game is not included
                df[f'{col}Prev{length}'] = df[col].rolling(length).mean().shift(1)
            if col != 'WON':
                # Drop the orriginal game level information (except the label)
                df.drop(col, axis=1, inplace=True)
        new_df = pd.concat([new_df, df])
# new_df.sort_values('gameId', ascending=True, inplace=True)
```

In [35]:
```
new_df.shape
```

Out[35]:
```
(21072, 213)
```

In [36]:
```
# We need to handle NaN's here by dropping the first five games each team plays so the
new_df.dropna(inplace=True)
```

In [37]:
```
new_df
```

Out[37]:

	team	home_or_away	gameId	season	WON	WONPrev1	WONPrev3	WONPrev5	WONPre
7003	ANA	1	2010020142	2010	0	1.0	0.666667	0.6	
7004	ANA	0	2010020156	2010	0	0.0	0.333333	0.4	
7005	ANA	1	2010020172	2010	1	0.0	0.333333	0.4	
7006	ANA	1	2010020186	2010	1	1.0	0.333333	0.4	
7007	ANA	1	2010020202	2010	1	1.0	0.666667	0.6	
...	...	...	...	...	...	...	...	...	
14207	WSH	0	2018021190	2018	1	1.0	0.666667	0.6	
14208	WSH	0	2018021206	2018	1	1.0	1.000000	0.6	

	team	home_or_away	gameId	season	WON	WONPrev1	WONPrev3	WONPrev5	WONPre
14209	WSH	0	2018021221	2018	0	1.0	1.000000	0.8	
14210	WSH	1	2018021246	2018	1	0.0	0.666667	0.8	
14211	WSH	1	2018021265	2018	0	1.0	0.666667	0.8	

18352 rows × 213 columns

In [38]:
```
cols_to_normalize = new_df.drop(columns=['team','gameId','home_or_away','season','WON']
# Normalizing the range of all columns once again using min-max
for column in cols_to_normalize:
    new_df[column] = (new_df[column] - new_df[column].min()) / (new_df[column].max() -
```

In [39]:
```
new_df
```

Out[39]:

	team	home_or_away	gameId	season	WON	WONPrev1	WONPrev3	WONPrev5	WONPre
7003	ANA	1	2010020142	2010	0	1.0	0.666667	0.6	
7004	ANA	0	2010020156	2010	0	0.0	0.333333	0.4	
7005	ANA	1	2010020172	2010	1	0.0	0.333333	0.4	
7006	ANA	1	2010020186	2010	1	1.0	0.333333	0.4	
7007	ANA	1	2010020202	2010	1	1.0	0.666667	0.6	
...	...	...	...	...	...	...	...	...	
14207	WSH	0	2018021190	2018	1	1.0	0.666667	0.6	
14208	WSH	0	2018021206	2018	1	1.0	1.000000	0.6	
14209	WSH	0	2018021221	2018	0	1.0	1.000000	0.8	
14210	WSH	1	2018021246	2018	1	0.0	0.666667	0.8	
14211	WSH	1	2018021265	2018	0	1.0	0.666667	0.8	

18352 rows × 213 columns

In [40]:
```
# Dropping categorical features
X2 = new_df.drop(columns=['team','gameId','season','WON'])
new_df['WON'] = new_df['WON'].astype('int')
Y2 = new_df['WON']
```

In [41]:
```
lr2 = LogisticRegression()
lr2.fit(X2,Y2)
```

Out[41]:
```
LogisticRegression()
```

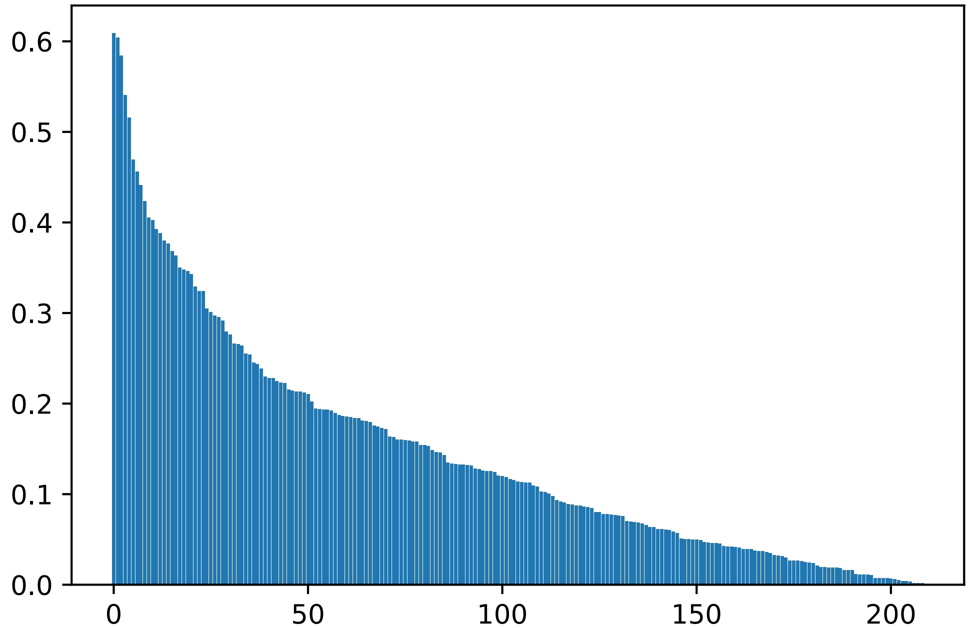
In [42]:
```
np.abs(lr2.coef_[0])
```

Out[42]:
```
array([0.38020233, 0.08650149, 0.18051855, 0.11913461, 0.32900308,
       0.07780129, 0.00754271, 0.13219839, 0.10244993, 0.01872345,
       0.04228512, 0.01156453, 0.06164827, 0.01618173, 0.01893042,
```

```
       0.00753459, 0.15431099, 0.01142233, 0.15331036, 0.22345165,
       0.0462514 , 0.07757787, 0.00735337, 0.13265953, 0.10307899,
       0.36357956, 0.05012325, 0.05900675, 0.06017022, 0.1487718 ,
       0.09093666, 0.30519076, 0.00393975, 0.03796473, 0.18108378,
       0.32433704, 0.58435775, 0.21556173, 0.01980727, 0.03921455,
       0.09355279, 0.39284691, 0.00174008, 0.37651189, 0.6044546 ,
       0.11997773, 0.11285789, 0.08724075, 0.03732658, 0.16285098,
       0.03937441, 0.12562028, 0.05023875, 0.21421454, 0.05050467,
       0.08023709, 0.19214056, 0.04911693, 0.22813277, 0.00428361,
       0.03944846, 0.09807519, 0.04638612, 0.02410081, 0.12767138,
       0.06354186, 0.13251129, 0.18537254, 0.21347206, 0.01902314,
       0.04176508, 0.01140785, 0.06123674, 0.21242174, 0.22288716,
       0.51578324, 0.40566164, 0.06106693, 0.2763951 , 0.25516523,
       0.18947368, 0.10948528, 0.10841616, 0.25434936, 0.02612782,
       0.02675557, 0.01618502, 0.04525375, 0.06850379, 0.03525951,
       0.13143653, 0.42383985, 0.18631925, 0.29545849, 0.04595921,
       0.40287152, 0.11650054, 0.08859025, 0.18423259, 0.08012978,
       0.07816205, 0.02996394, 0.22998492, 0.06581937, 0.22816489,
       0.03180998, 0.02147178, 0.02652292, 0.02503787, 0.07657068,
       0.17195122, 0.0066938 , 0.01048396, 0.18381967, 0.07000946,
       0.00636762, 0.21058141, 0.07587007, 0.27954495, 0.04694296,
       0.20253841, 0.26552928, 0.02673226, 0.30109539, 0.05079993,
       0.17451017, 0.29184195, 0.1253337 , 0.13471685, 0.34291566,
       0.04987754, 0.21329368, 0.54077533, 0.06913462, 0.08767479,
       0.12452367, 0.00492583, 0.11579791, 0.15828731, 0.04282771,
       0.11285593, 0.02464784, 0.18711988, 0.04129135, 0.15830075,
       0.36845668, 0.03236173, 0.14581121, 0.11391898, 0.01117246,
       0.44124079, 0.00204411, 0.03305863, 0.07684331, 0.12591654,
       0.29725263, 0.24508005, 0.15924348, 0.3245576 , 0.19438704,
       0.03603499, 0.16004403, 0.14333825, 0.08471866, 0.12079519,
       0.07053619, 0.16338304, 0.26409626, 0.01953335, 0.18564984,
       0.16052394, 0.14663324, 0.46953424, 0.01611355, 0.01865566,
       0.00760358, 0.15450117, 0.13330105, 0.03742458, 0.11355418,
       0.45645463, 0.17949259, 0.350139  , 0.60909198, 0.13364588,
       0.38826318, 0.1940308 , 0.08884584, 0.34784743, 0.0673831 ,
       0.17301433, 0.00379742, 0.34660338, 0.12845635, 0.19322209,
       0.04225283, 0.26621848, 0.19343421, 0.00213324, 0.17593605,
       0.08570162, 0.06360132, 0.10047087, 0.16031931, 0.23877248,
       0.05685443, 0.09184742, 0.24349083, 0.2247884 ])
```

Once again, we plot our feature coefficients in descending order to observe where a drop-off occurs

In [43]:
```python
imp2 = np.abs(lr2.coef_[0])
imp2sorted = -np.sort(-imp2)
# for i,v in enumerate(importance):
#       print('Feature: %0d, Score: %.5f' % (i,v))
plt.bar([x for x in range(len(imp2sorted))], imp2sorted)
plt.show()
```



From the plot above (which looks rather strange), we choose the top 20 features as there is an observable drop-off in performance afterwards

Next, we find our columns to drop from our dataset, so we are left with our 20 most important features

In [44]:
```
cols_to_drop = []
thresh = imp2sorted[20]
print(f'Threshold: {thresh}')
for n,val in enumerate(imp2):
    if val < thresh:
        print(val)
        cols_to_drop.append(new_df.columns[n+4])
new_df.drop(cols_to_drop,axis=1,inplace=True)

# Verifying that feature selection worked as expected
print(cols_to_drop)
```

```
Threshold: 0.342915657650887
0.08650148803832602
0.18051854564007533
0.11913460600055997
0.32900307708219806
0.07780128916993427
0.0075427051669667245
0.13219839069617384
0.10244992603651432
0.01872345484815044
0.04228512137328494
0.011564530793545563
0.061648273677426924
0.01618173013503587
0.018930423840533953
0.007534585098847988
0.15431098938325308
0.011422327460200544
0.15331035671773166
0.22345165245009485
0.04625140317908757
0.07757786971039266
```

```
0.007353365997974131
0.1326595301144547
0.10307898927321395
0.05012325330950923
0.059006745573708004
0.060170216966630684
0.14877180186400382
0.0909366614797855
0.30519076323394745
0.00393975031186675
0.03796472713315838
0.18108378275850098
0.3243370377411484
0.2155617269252494
0.019807265710716483
0.03921454977398007
0.09355278562491279
0.0017400769533195213
0.11997773358320692
0.1128578876851725
0.08724074894623815
0.03732658084706898
0.1628509775098
0.03937440635562603
0.12562028202505704
0.05023874763863562
0.21421454004958015
0.05050466896756309
0.0802370861757229
0.1921405601402059
0.04911693417480252
0.22813276921269443
0.004283609833901727
0.039448461279654415
0.09807519140411476
0.04638612070069872
0.02410081134858907
0.12767137966363923
0.06354185864296114
0.13251128730508654
0.18537254236963904
0.2134720635818386
0.019023137322938352
0.04176508262629777
0.011407850081452921
0.061236740526028885
0.21242173800907752
0.222887161449547
0.06106693450571607
0.2763950950196644
0.25516523217728887
0.18947367760160808
0.10948528422852045
0.10841615540828138
0.25434935765873556
0.026127815063265693
0.026755567039322047
0.0161850214591417
0.045253748074671765
0.06850378644894257
0.035259514090730276
0.13143653174450742
0.18631925247897094
0.29545849219059994
0.04595921424777087
```

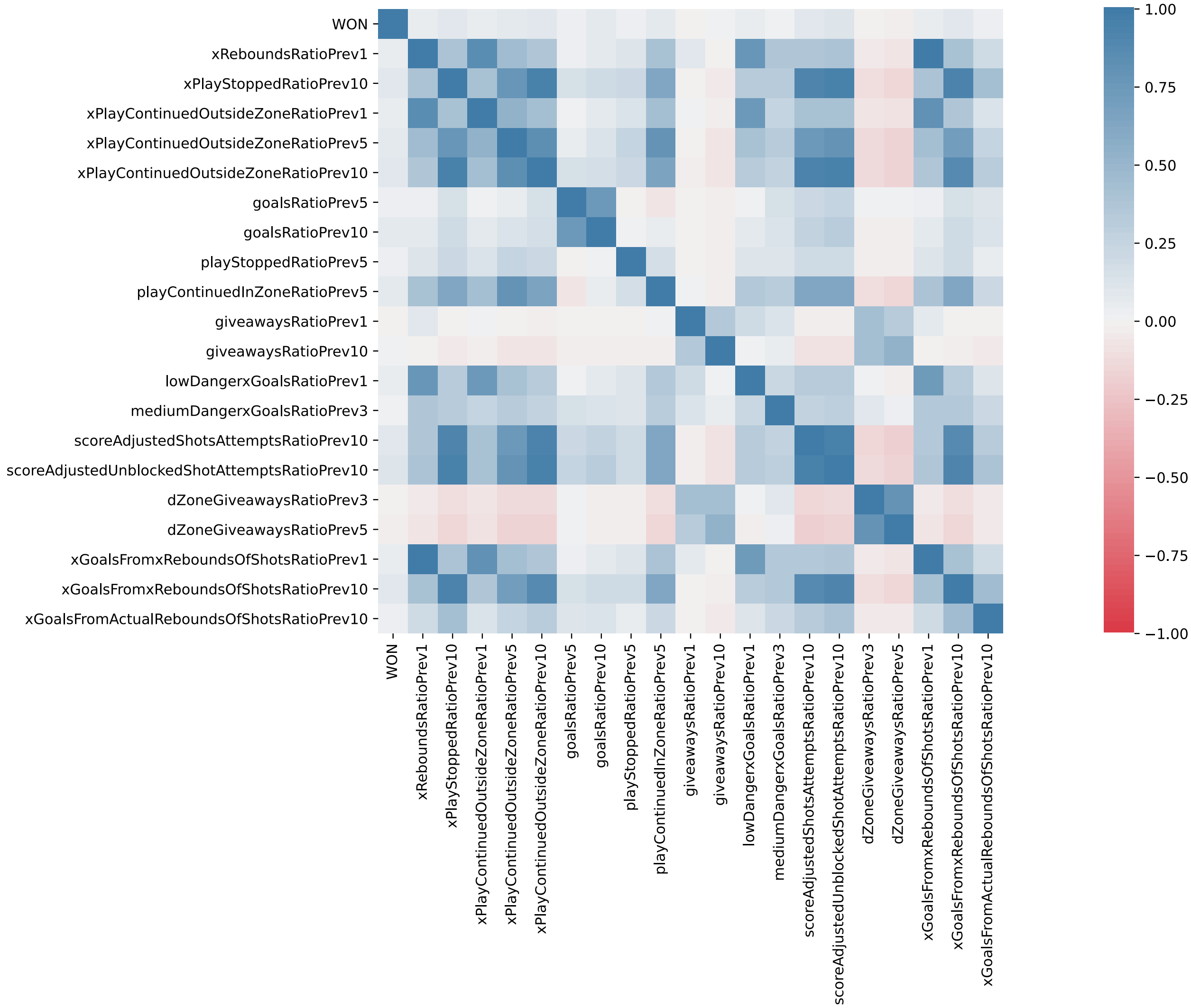
```
0.1165005422046777
0.08859024605846513
0.18423259241917198
0.08012977829250957
0.07816205156693029
0.029963935786364176
0.22998491531557955
0.06581937186382585
0.22816489055021358
0.03180997562014463
0.02147178301108327
0.026522922611568023
0.02503786717594833
0.07657067840420427
0.17195122358065007
0.006693801783175904
0.010483957469950629
0.18381966741299904
0.07000945721030671
0.00636761531221792
0.21058141127295574
0.07587007381076118
0.2795449487849745
0.04694295876267309
0.2025384113570656
0.2655292842220097
0.02673225544085253
0.3010953850029175
0.05079992661663931
0.1745101749195344
0.29184194754949777
0.12533369947551704
0.13471684549551835
0.049877539187669614
0.21329368202641755
0.06913461551950242
0.08767479393807046
0.12452367049870519
0.0049258250567276885
0.11579790669079751
0.1582873142212032
0.04282770597413564
0.1128559278859579
0.024647840150957924
0.18711987805422825
0.04129134821331117
0.1583007530907629
0.032361726124240535
0.145811205124789
0.11391898238265573
0.011172460938267624
0.002044105931467608
0.033058626878913304
0.07684331241199613
0.1259165405995337
0.297252627720771
0.2450800491552157
0.159243483957818
0.324557601216973
0.19438703903945165
0.03603499224732056
0.1600440281720777
0.14333825286861085
0.08471866165009882
0.12079519231590982
```

```
0.07053618614533474
0.16338304419004174
0.2640962632751311
0.019533351972511687
0.18564984339966492
0.1605239426030805
0.14663324385987
0.016113552168440314
0.018655659091200746
0.007603576062412478
0.15450117391146806
0.13330105397634695
0.03742458089576298
0.1135541791579763
0.17949258790524003
0.13364588322604623
0.194030803119223
0.08884584355769246
0.06738309610493125
0.17301433241646702
0.003797424605548963
0.12845634846876547
0.19322208631562018
0.042252833319590226
0.26621848199848835
0.19343420770387668
0.0021332380088058865
0.17593605475705623
0.08570162305625766
0.06360131602152472
0.10047087499182901
0.1603193076152112
0.23877248471506302
0.05685443097149979
0.09184741672765374
0.24349082577032652
0.22478839770033193
['WONPrev1', 'WONPrev3', 'WONPrev5', 'WONPrev10', 'xGoalsPercentagePrev1', 'xGoalsPercen
tagePrev3', 'xGoalsPercentagePrev5', 'xGoalsPercentagePrev10', 'corsiPercentagePrev1',
'corsiPercentagePrev3', 'corsiPercentagePrev5', 'corsiPercentagePrev10', 'fenwickPercent
agePrev1', 'fenwickPercentagePrev3', 'fenwickPercentagePrev5', 'fenwickPercentagePrev1
0', 'xOnGoalRatioPrev1', 'xOnGoalRatioPrev3', 'xOnGoalRatioPrev5', 'xOnGoalRatioPrev10',
'xGoalsRatioPrev1', 'xGoalsRatioPrev3', 'xGoalsRatioPrev5', 'xGoalsRatioPrev10', 'xRebou
ndsRatioPrev3', 'xReboundsRatioPrev5', 'xReboundsRatioPrev10', 'xFreezeRatioPrev1', 'xFr
eezeRatioPrev3', 'xFreezeRatioPrev5', 'xFreezeRatioPrev10', 'xPlayStoppedRatioPrev1', 'x
PlayStoppedRatioPrev3', 'xPlayStoppedRatioPrev5', 'xPlayContinuedInZoneRatioPrev1', 'xPl
ayContinuedInZoneRatioPrev3', 'xPlayContinuedInZoneRatioPrev5', 'xPlayContinuedInZoneRat
ioPrev10', 'xPlayContinuedOutsideZoneRatioPrev3', 'flurryAdjustedxGoalsRatioPrev1', 'flu
rryAdjustedxGoalsRatioPrev3', 'flurryAdjustedxGoalsRatioPrev5', 'flurryAdjustedxGoalsRat
ioPrev10', 'scoreVenueAdjustedxGoalsRatioPrev1', 'scoreVenueAdjustedxGoalsRatioPrev3',
'scoreVenueAdjustedxGoalsRatioPrev5', 'scoreVenueAdjustedxGoalsRatioPrev10', 'flurryScor
eVenueAdjustedxGoalsRatioPrev1', 'flurryScoreVenueAdjustedxGoalsRatioPrev3', 'flurryScor
eVenueAdjustedxGoalsRatioPrev5', 'flurryScoreVenueAdjustedxGoalsRatioPrev10', 'shotsOnGo
alRatioPrev1', 'shotsOnGoalRatioPrev3', 'shotsOnGoalRatioPrev5', 'shotsOnGoalRatioPrev1
0', 'missedShotsRatioPrev1', 'missedShotsRatioPrev3', 'missedShotsRatioPrev5', 'missedSh
otsRatioPrev10', 'blockedShotAttemptsRatioPrev1', 'blockedShotAttemptsRatioPrev3', 'bloc
kedShotAttemptsRatioPrev5', 'blockedShotAttemptsRatioPrev10', 'shotAttemptsRatioPrev1',
'shotAttemptsRatioPrev3', 'shotAttemptsRatioPrev5', 'shotAttemptsRatioPrev10', 'goalsRat
ioPrev1', 'goalsRatioPrev3', 'reboundsRatioPrev1', 'reboundsRatioPrev3', 'reboundsRatioP
rev5', 'reboundsRatioPrev10', 'reboundGoalsRatioPrev1', 'reboundGoalsRatioPrev3', 'rebou
ndGoalsRatioPrev5', 'reboundGoalsRatioPrev10', 'freezeRatioPrev1', 'freezeRatioPrev3',
'freezeRatioPrev5', 'freezeRatioPrev10', 'playStoppedRatioPrev1', 'playStoppedRatioPrev
3', 'playStoppedRatioPrev10', 'playContinuedInZoneRatioPrev1', 'playContinuedInZoneRatio
Prev3', 'playContinuedInZoneRatioPrev10', 'playContinuedOutsideZoneRatioPrev1', 'playCon
tinuedOutsideZoneRatioPrev3', 'playContinuedOutsideZoneRatioPrev5', 'playContinuedOutsid
```

```
eZoneRatioPrev10', 'savedShotsOnGoalRatioPrev1', 'savedShotsOnGoalRatioPrev3', 'savedSho
tsOnGoalRatioPrev5', 'savedShotsOnGoalRatioPrev10', 'savedUnblockedShotAttemptsRatioPrev
1', 'savedUnblockedShotAttemptsRatioPrev3', 'savedUnblockedShotAttemptsRatioPrev5', 'sav
edUnblockedShotAttemptsRatioPrev10', 'penaltiesRatioPrev1', 'penaltiesRatioPrev3', 'pena
ltiesRatioPrev5', 'penaltiesRatioPrev10', 'penalityMinutesRatioPrev1', 'penalityMinutesR
atioPrev3', 'penalityMinutesRatioPrev5', 'penalityMinutesRatioPrev10', 'faceOffsWonRatio
Prev1', 'faceOffsWonRatioPrev3', 'faceOffsWonRatioPrev5', 'faceOffsWonRatioPrev10', 'hit
sRatioPrev1', 'hitsRatioPrev3', 'hitsRatioPrev5', 'hitsRatioPrev10', 'takeawaysRatioPrev
1', 'takeawaysRatioPrev3', 'takeawaysRatioPrev5', 'takeawaysRatioPrev10', 'giveawaysRati
oPrev3', 'giveawaysRatioPrev5', 'lowDangerShotsRatioPrev1', 'lowDangerShotsRatioPrev3',
'lowDangerShotsRatioPrev5', 'lowDangerShotsRatioPrev10', 'mediumDangerShotsRatioPrev1',
'mediumDangerShotsRatioPrev3', 'mediumDangerShotsRatioPrev5', 'mediumDangerShotsRatioPre
v10', 'highDangerShotsRatioPrev1', 'highDangerShotsRatioPrev3', 'highDangerShotsRatioPre
v5', 'highDangerShotsRatioPrev10', 'lowDangerxGoalsRatioPrev3', 'lowDangerxGoalsRatioPre
v5', 'lowDangerxGoalsRatioPrev10', 'mediumDangerxGoalsRatioPrev1', 'mediumDangerxGoalsRa
tioPrev5', 'mediumDangerxGoalsRatioPrev10', 'highDangerxGoalsRatioPrev1', 'highDangerxGo
alsRatioPrev3', 'highDangerxGoalsRatioPrev5', 'highDangerxGoalsRatioPrev10', 'lowDangerG
oalsRatioPrev1', 'lowDangerGoalsRatioPrev3', 'lowDangerGoalsRatioPrev5', 'lowDangerGoals
RatioPrev10', 'mediumDangerGoalsRatioPrev1', 'mediumDangerGoalsRatioPrev3', 'mediumDange
rGoalsRatioPrev5', 'mediumDangerGoalsRatioPrev10', 'highDangerGoalsRatioPrev1', 'highDan
gerGoalsRatioPrev3', 'highDangerGoalsRatioPrev5', 'highDangerGoalsRatioPrev10', 'scoreAd
justedShotsAttemptsRatioPrev1', 'scoreAdjustedShotsAttemptsRatioPrev3', 'scoreAdjustedSh
otsAttemptsRatioPrev5', 'unblockedShotAttemptsRatioPrev1', 'unblockedShotAttemptsRatioPr
ev3', 'unblockedShotAttemptsRatioPrev5', 'unblockedShotAttemptsRatioPrev10', 'scoreAdjus
tedUnblockedShotAttemptsRatioPrev1', 'scoreAdjustedUnblockedShotAttemptsRatioPrev3', 'sc
oreAdjustedUnblockedShotAttemptsRatioPrev5', 'dZoneGiveawaysRatioPrev1', 'dZoneGiveaways
RatioPrev10', 'xGoalsFromxReboundsOfShotsRatioPrev3', 'xGoalsFromxReboundsOfShotsRatioPr
ev5', 'xGoalsFromActualReboundsOfShotsRatioPrev1', 'xGoalsFromActualReboundsOfShotsRatio
Prev3', 'xGoalsFromActualReboundsOfShotsRatioPrev5', 'reboundxGoalsRatioPrev1', 'rebound
xGoalsRatioPrev3', 'reboundxGoalsRatioPrev5', 'reboundxGoalsRatioPrev10', 'totalShotCred
itRatioPrev1', 'totalShotCreditRatioPrev3', 'totalShotCreditRatioPrev5', 'totalShotCredi
tRatioPrev10', 'scoreAdjustedTotalShotCreditRatioPrev1', 'scoreAdjustedTotalShotCreditRa
tioPrev3', 'scoreAdjustedTotalShotCreditRatioPrev5', 'scoreAdjustedTotalShotCreditRatioP
rev10', 'scoreFlurryAdjustedTotalShotCreditRatioPrev1', 'scoreFlurryAdjustedTotalShotCre
ditRatioPrev3', 'scoreFlurryAdjustedTotalShotCreditRatioPrev5', 'scoreFlurryAdjustedTota
lShotCreditRatioPrev10']
```

And as before, we create a correlation heatmap to visualize any redundant/synonymous features

In [45]:
```
corr = new_df.drop(columns=['team','gameId','season']).corr()
fig, ax = plt.subplots(figsize=(25,8))
ax = sns.heatmap(
    corr,
    vmin=-1, vmax=1,
    square=True,
    cmap=sns.diverging_palette(10,240,n=100),
    ax=ax
)
```



In [46]:
```python
# dimensions of the dataset after selecting features
new_df.shape
```

Out[46]: (18352, 25)

In [47]:
```python
# Cross-validating a simple SVM using an 80/20 training/validation split and 10-fold to
from sklearn import svm
from sklearn.model_selection import cross_val_score,cross_validate
from sklearn.metrics import accuracy_score
feat = new_df.drop(columns=['team','gameId','WON'])
label = new_df['WON']
# Train on 2010 - 2017, test on 2018
x_train = feat.loc[feat.season < 2018].drop(columns=['season'])
x_test = feat.loc[feat.season == 2018].drop(columns=['season'])
y_train = new_df.loc[new_df.season < 2018]['WON'].drop(columns=['season'])
y_test = new_df.loc[new_df.season == 2018]['WON'].drop(columns=['season'])
# x_train,x_test,y_train,y_test = train_test_split(feat,label,test_size=0.2) # test on
clf = svm.SVC(kernel='linear', C=1)
scores = cross_validate(clf, x_train, y_train, scoring='roc_auc',cv=10,return_estimator
scores['test_score']
```

Out[47]:
```
array([0.54412401, 0.57493581, 0.59579057, 0.58277712, 0.54935533,
       0.57924438, 0.56031378, 0.56852607, 0.56406511, 0.59850593])
```

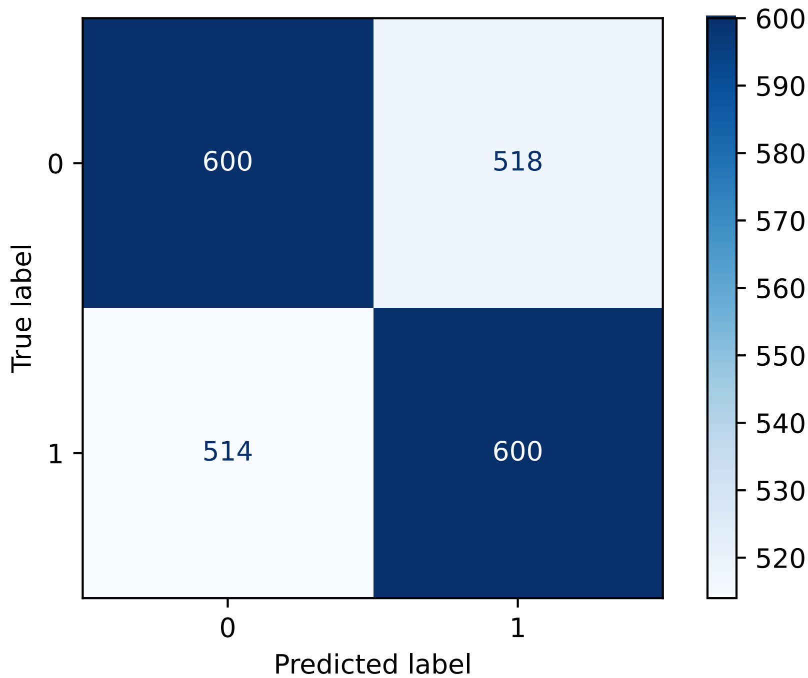
In [48]:
```python
# Let's find our best estimator. Since cross_validate returns a dict of estimators and
```

```
best_score = np.argmax(scores['test_score'])
estimators = scores['estimator']
clf = estimators[best_score]
pred = clf.predict(x_test)
accuracy_score(y_test,pred)
```

Out[48]: 0.5376344086021505

In [49]:
```
plot_confusion_matrix(clf,x_test,y_test,cmap=plt.cm.Blues)
```

Out[49]: <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x21080534310>



Well, that's not very good, let's try using PCA and see if performance improves

In [50]:
```
from sklearn.decomposition import PCA
pca = PCA(n_components=2) # one component for each numerical feature
pca.fit(x_train)
new_data_train = pca.transform(x_train)
new_data_test = pca.transform(x_test)
clf = svm.SVC(kernel='linear', C=1)
scores = cross_validate(clf, new_data_train, y_train, scoring='roc_auc',cv=10,return_es
scores['test_score']
```

Out[50]:
```
array([0.53273454, 0.56079558, 0.57792518, 0.57627194, 0.52687659,
       0.56504258, 0.54865802, 0.54990333, 0.55270644, 0.59359703])
```

In [51]:
```
new_data_train
```

Out[51]:
```
array([[ 0.5138376 ,  0.70991994],
       [-0.48338707,  0.38975251],
       [ 0.51332449,  0.74506042],
       ...,
       [-0.49793645,  0.16342373],
       [ 0.50119703,  0.17941796],
       [ 0.50354524,  0.01360153]])
```

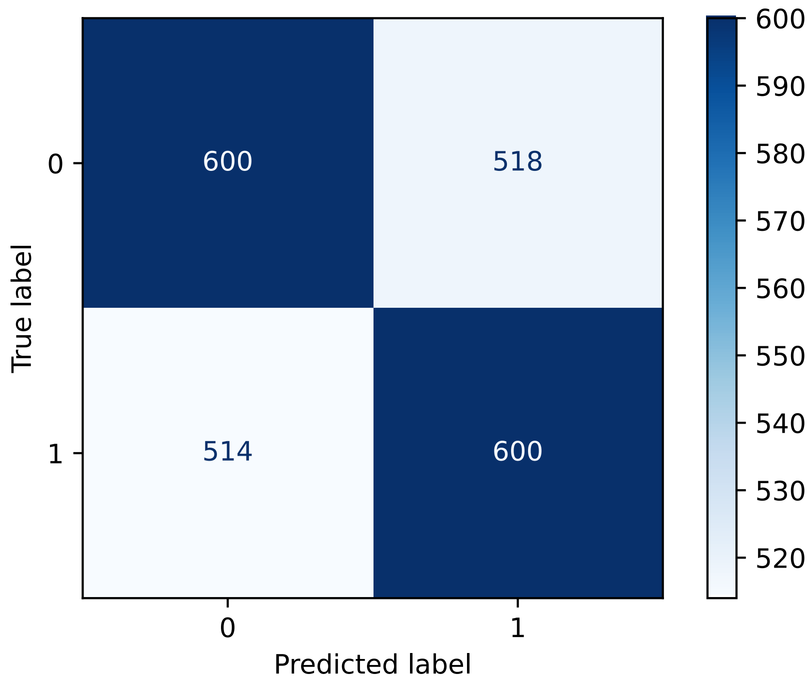
In [52]:
```
# Let's find our best estimator. Since cross_validate returns a dict of estimators and
best_score = np.argmax(scores['test_score'])
```

```
estimators = scores['estimator']
clf = estimators[best_score]
pred = clf.predict(new_data_test)
accuracy_score(y_test,pred)
```

Out[52]: 0.5376344086021505

In [53]:
```
plot_confusion_matrix(clf,new_data_test,y_test,cmap=plt.cm.Blues)
```

Out[53]: <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x210805c0250>

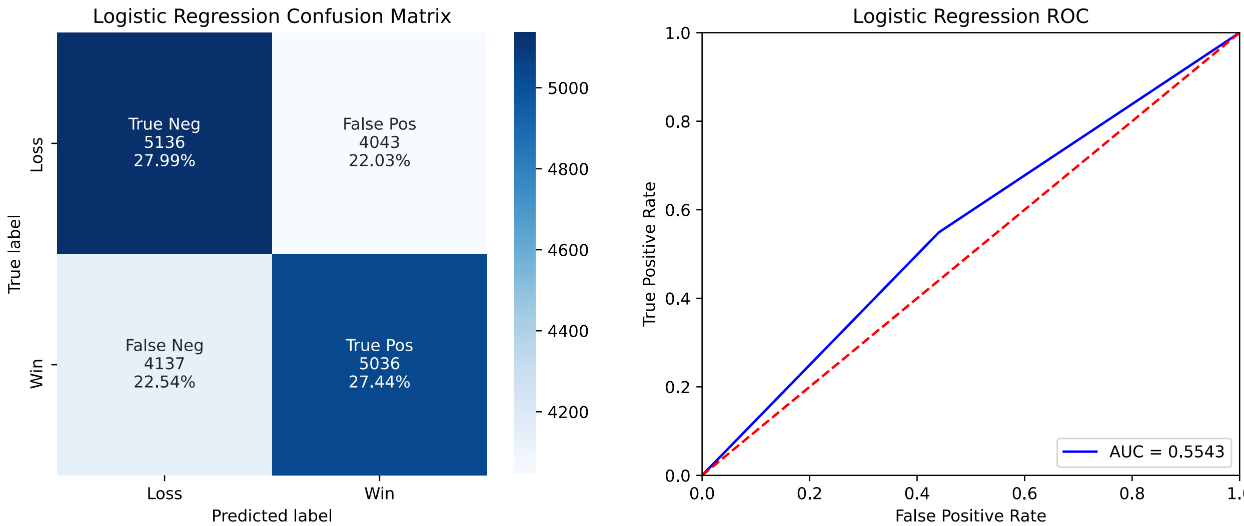


This is still very poor--it's actually even worse--and I'm not sure why (I've tried everything I know how and the accuracy still hovers around 50/50 at best)

Let's see how other classifiers perform; anything around 55% is still pretty good using this type of data

In [54]:
```
lrc = LogisticRegression()
knnc = KNeighborsClassifier()
dtc = DecisionTreeClassifier()
rfc = RandomForestClassifier()
gbc = GradientBoostingClassifier()
```

In [55]:
```
validate_model(lrc, feat, label, 'Logistic Regression', group_names=group_names, catego
```

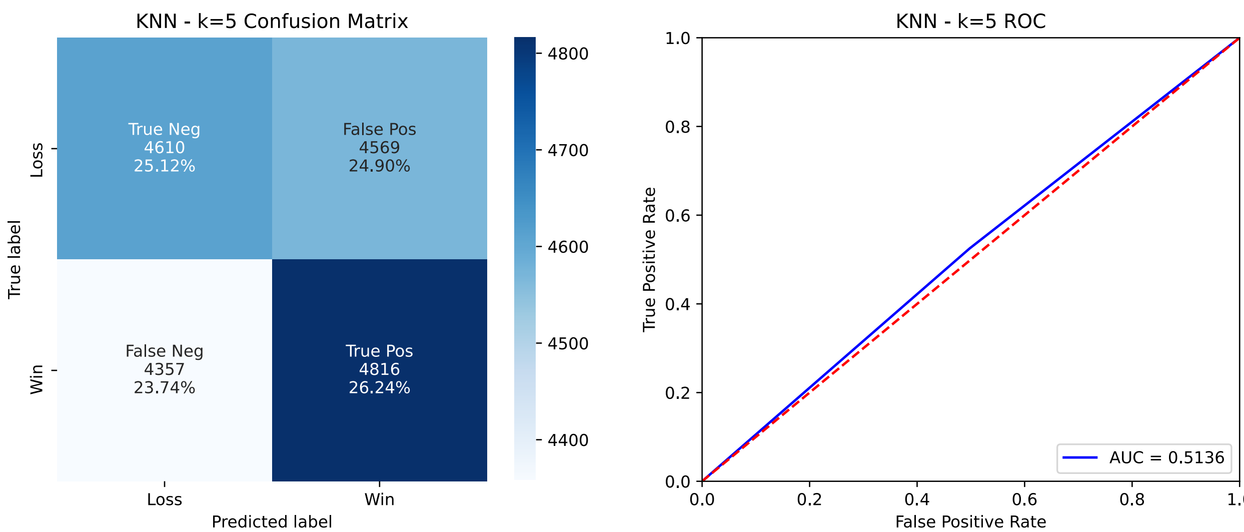


Accuracy=0.554
Precision=0.555
Recall=0.549
F1 Score=0.552

```
<Figure size 432x288 with 0 Axes>
```

In [56]:

```python
validate_model(knnc, feat, label, 'KNN - k=5', group_names=group_names, categories=cate
```

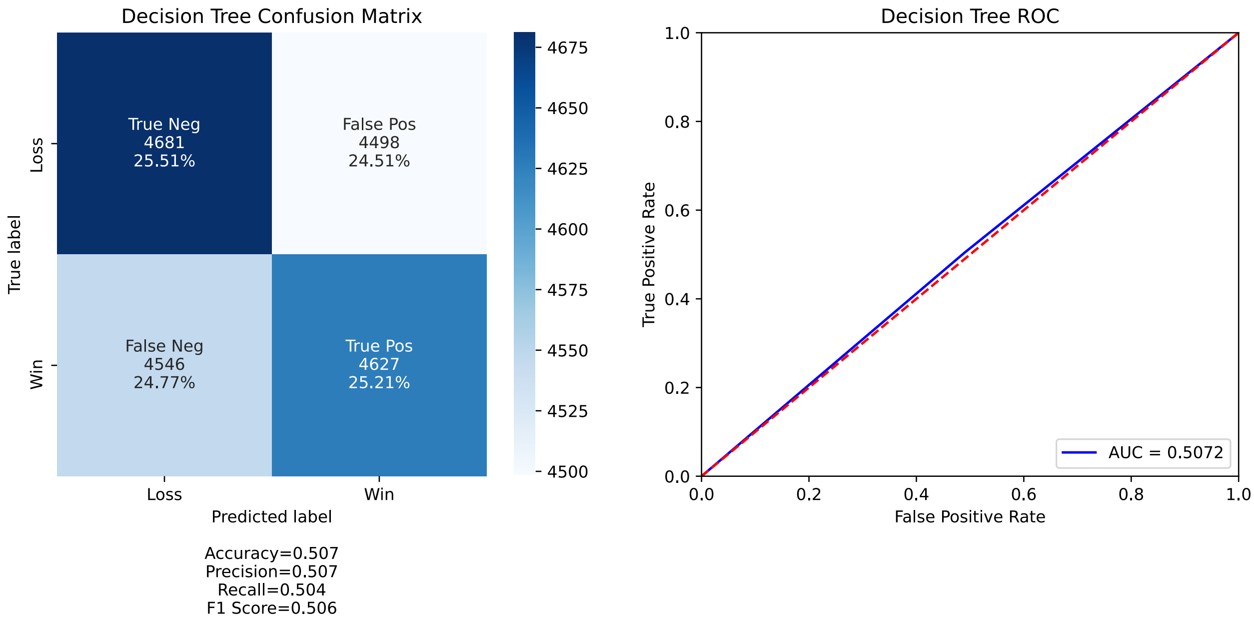


Accuracy=0.514
Precision=0.513
Recall=0.525
F1 Score=0.519

```
<Figure size 432x288 with 0 Axes>
```

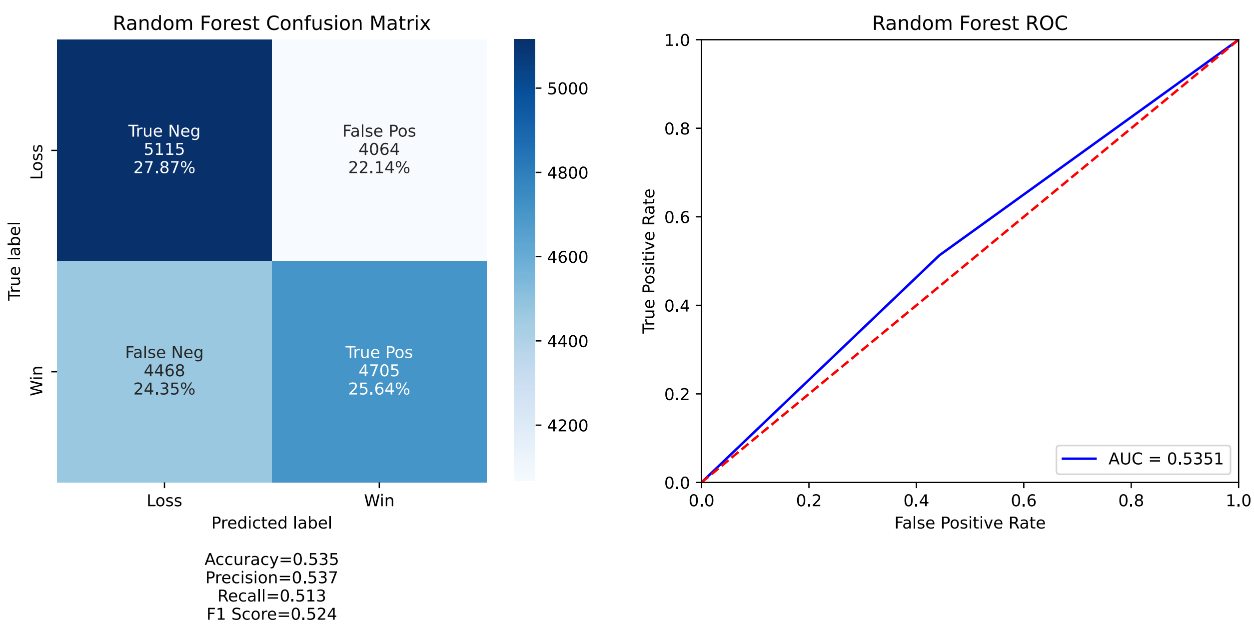
In [57]:

```python
validate_model(dtc, feat, label, 'Decision Tree', group_names=group_names, categories=c
```



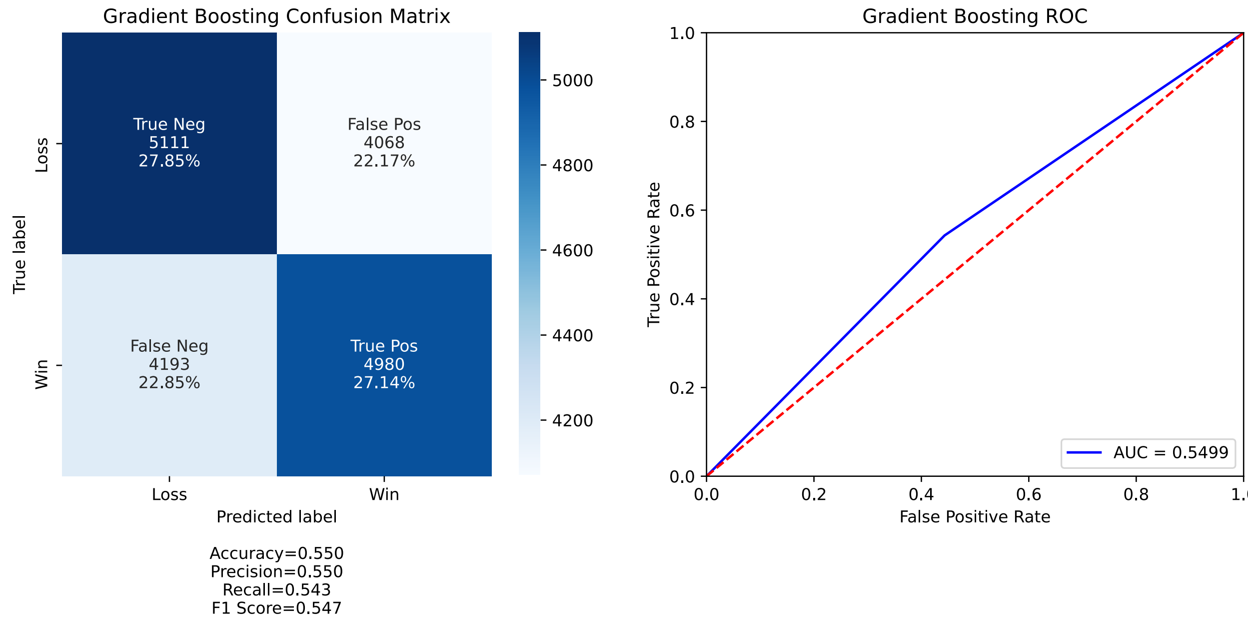

```
<Figure size 432x288 with 0 Axes>
```

In [58]:
```
validate_model(rfc, feat, label, 'Random Forest', group_names=group_names, categories=c
```




```
<Figure size 432x288 with 0 Axes>
```

In [59]:
```
validate_model(gbc, feat, label, 'Gradient Boosting', group_names=group_names, categori
```




```
<Figure size 432x288 with 0 Axes>
```

This leads me to believe that our new features have nearly-zero predictive power, but while we're here, let's see if tuning hyperparameters makes even the slightest difference

In [60]:
```
from sklearn.model_selection import GridSearchCV

selected_model = gbc = GradientBoostingClassifier(min_samples_split=500, min_samples_le

n_trees_search = GridSearchCV(estimator=selected_model, param_grid={'n_estimators':rang
n_trees_search.fit(feat, label)
```

Out[60]:
```
GridSearchCV(cv=5,
             estimator=GradientBoostingClassifier(max_depth=8,
                                                  max_features='sqrt',
                                                  min_samples_leaf=50,
                                                  min_samples_split=500,
                                                  random_state=10,
                                                  subsample=0.8),
             n_jobs=4, param_grid={'n_estimators': range(20, 101, 10)},
             scoring='roc_auc')
```

In [61]:
```
n_trees = n_trees_search.best_params_['n_estimators']
```

In [62]:
```
refined_gbc = GradientBoostingClassifier(min_samples_leaf=50, n_estimators=n_trees, max
depth_split_grid = {'max_depth': range(5,16,2), 'min_samples_split': range(200,1601,200
depth_split_test = GridSearchCV(estimator=refined_gbc , param_grid=depth_split_grid, sc
depth_split_test.fit(feat, label)
depth = depth_split_test.best_params_['max_depth']
split = depth_split_test.best_params_['min_samples_split']
```

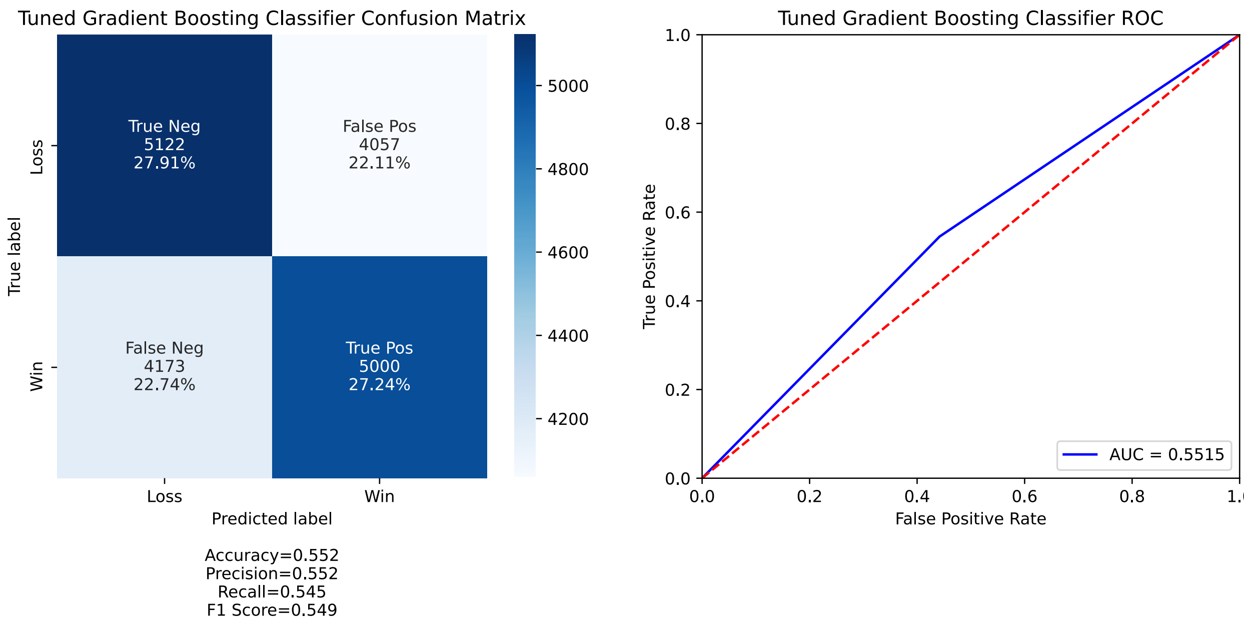
In [63]:
```
refined_gbc = GradientBoostingClassifier(n_estimators=n_trees, min_samples_split=split,
leaf_grid = {'min_samples_leaf':range(30,91,10)}
leaf_test = GridSearchCV(estimator=refined_gbc, param_grid=leaf_grid, scoring='roc_auc'
leaf_test.fit(feat,label)
leaf = leaf_test.best_params_['min_samples_leaf']
```

In [64]:
```
refined_gbc = GradientBoostingClassifier(n_estimators=n_trees, min_samples_split=split,
features_grid = {'max_features':range(5,30,2)}
features_test = GridSearchCV(estimator=refined_gbc, param_grid=features_grid, scoring='
```

```
    features_test.fit(feat, label)
    features = features_test.best_params_['max_features']
```

In [65]:
```
refined_gbc = GradientBoostingClassifier(n_estimators=n_trees, min_samples_split=split,
subsample_grid = {'subsample':[0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9]}
subsample_test = GridSearchCV(estimator=refined_gbc, param_grid=subsample_grid, scoring
subsample_test.fit(feat,label)
subsample = subsample_test.best_params_['subsample']
```

In [66]:
```
tuned_model = GradientBoostingClassifier(learning_rate=0.001,n_estimators=100*n_trees,
validate_model(tuned_model, feat,label, 'Tuned Gradient Boosting Classifier', group_nam
```




```
<Figure size 432x288 with 0 Axes>
```

Unfortunately, that's about as good as this approach is going to get

Approach #3 - Combining Opponent Data

To this point, we have only been considering information from one of the two teams in each game. By combining the features of both teams to each gameId, we will be able to feed our model much more information.

In [67]:
```
# Separate training and testing datasets for ease of use later
df_train = new_df.loc[new_df.season < 2018]
df_test = new_df.loc[new_df.season == 2018]
```

In [68]:
```
# Define function for merging the features of both opponents for each game ID
# We must make sure to keep a 50/50 split for our label so our model doesn't get biased
def merge_opponents(df):
    # Get all the winning teams for each game
    wins = df.loc[df.WON == 1]
    wins_index = wins.index.tolist()
    # Devide into 2
    half_of_wins = wins.iloc[:int(len(wins_index)/2), :]
    # find the losing teams corresponding to the first half of the games
    corresponding_games = df.loc[df.gameId.isin(half_of_wins.gameId.values.tolist())]
    corresponding_losses = corresponding_games.drop(corresponding_games.loc[correspondi
    # Get the wins and losses for the other half of games
```

```
    other_games = df.loc[~df.gameId.isin(half_of_wins.gameId.values.tolist())]
    other_wins = other_games[other_games.WON == 1]
    other_losses = other_games[other_games.WON == 0]
    # Append opposing teams to the same row
    # The games are spit in two so that half of the resulting rows represent "wins" and
    for col in df.columns:
        if col != 'gameId':
            corresponding_losses.rename(columns={col: f"opp_{col}"}, inplace=True)
            other_wins.rename(columns={col: f"opp_{col}"}, inplace=True)
    # combine to give merged dataframe
    first_half = pd.merge(half_of_wins, corresponding_losses, how='inner', on='gameId')
    second_half = pd.merge(other_losses, other_wins, how='inner', on='gameId')
    merged_df = pd.concat([first_half, second_half])
    return merged_df
```

In [69]:

```
# Merge opponents for our train and test dataframes
df_train = merge_opponents(df_train)
df_test = merge_opponents(df_test)
df_test
```

Out[69]:

	team	home_or_away	gameId	season	WON	xReboundsRatioPrev1	xPlayStoppedRatioPrev10
0	ANA	1	2018020209	2018	1	0.576972	0.166723
1	ANA	1	2018020226	2018	1	0.581998	0.253615
2	ANA	1	2018020263	2018	1	0.235217	0.267067
3	ANA	1	2018020330	2018	1	0.445734	0.451303
4	ANA	1	2018020337	2018	1	0.621220	0.462389
...	...	...	...	...	...	...	...
547	WSH	0	2018020765	2018	0	0.426076	0.450317
548	WSH	0	2018021073	2018	0	0.304727	0.630096
549	WSH	0	2018021105	2018	0	0.575823	0.714867
550	WSH	1	2018021133	2018	0	0.501870	0.728422
551	WSH	1	2018021265	2018	0	0.336245	0.488613

1106 rows × 49 columns

In [70]:

```
# Drop the metadata and opponent labels
df_train.drop(columns=['team','opp_team','opp_WON','gameId','opp_home_or_away','opp_sea
df_test.drop(columns=['team','opp_team','opp_WON','gameId','opp_home_or_away','opp_seas
# create our training and testing vectors
x_train = df_train.drop(columns=['WON'])
x_test = df_test.drop(columns=['WON'])
y_train = df_train['WON']
y_test = df_test['WON']
# ensure the datatypes are correct
y_train = y_train.astype('int')
y_test = y_test.astype('int')
```

In [71]:

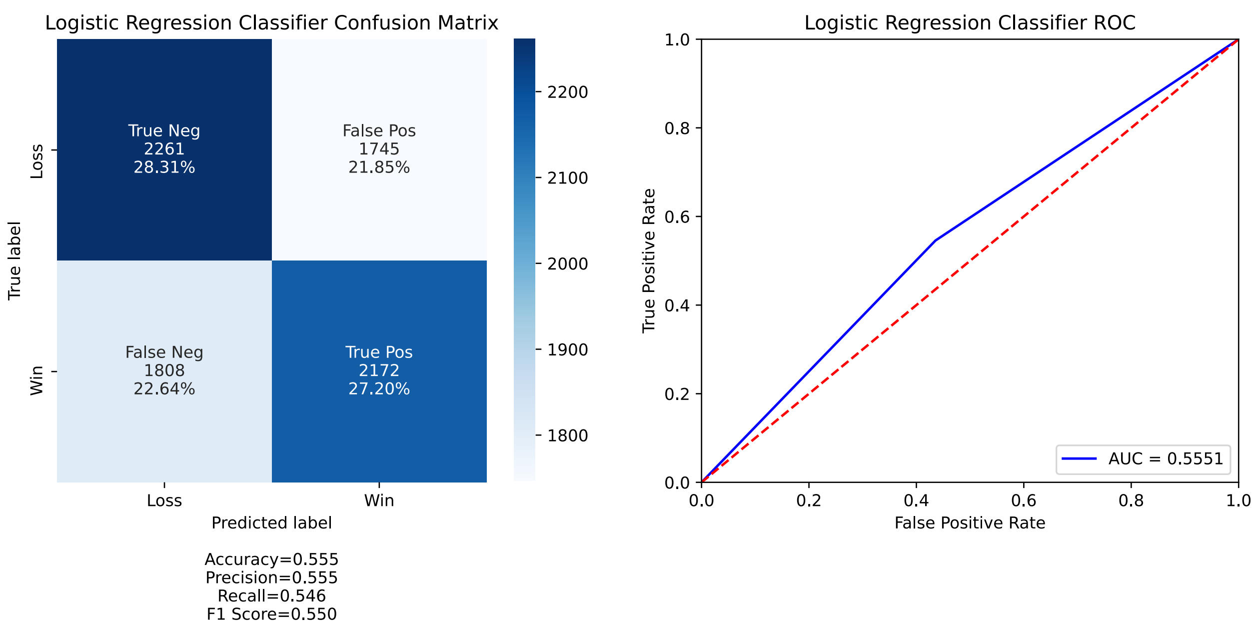
```
x_train.home_or_away = x_train.home_or_away.astype('int')
x_test.home_or_away = x_test.home_or_away.astype('int')
```

In [72]:
```
# Train a logistic regression model using the features from both teams
lr2 = LogisticRegression()
lr2.fit(x_train,y_train)
pred = lr2.predict(x_test)
accuracy_score(y_test,pred)
```

Out[72]: 0.5732368896925859

In [73]:
```
lrc = LogisticRegression()
knnc = KNeighborsClassifier()
dtc = DecisionTreeClassifier()
rfc = RandomForestClassifier()
gbc = GradientBoostingClassifier()
```

In [74]:
```
validate_model(lrc, x_train, y_train, 'Logistic Regression Classifier', group_names=gro
```

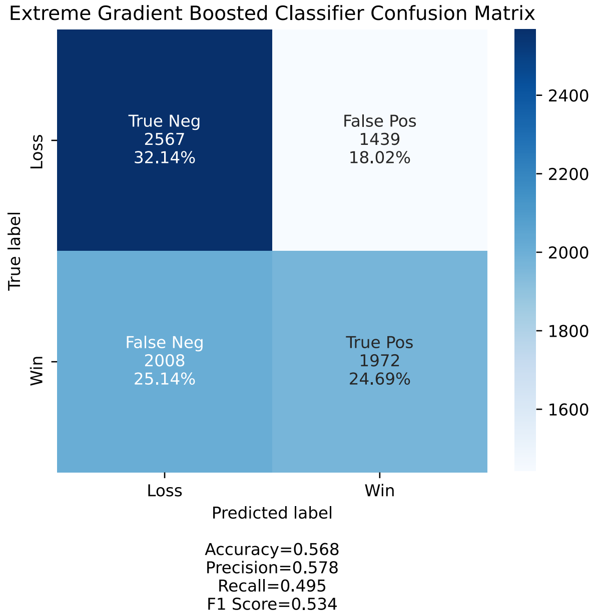


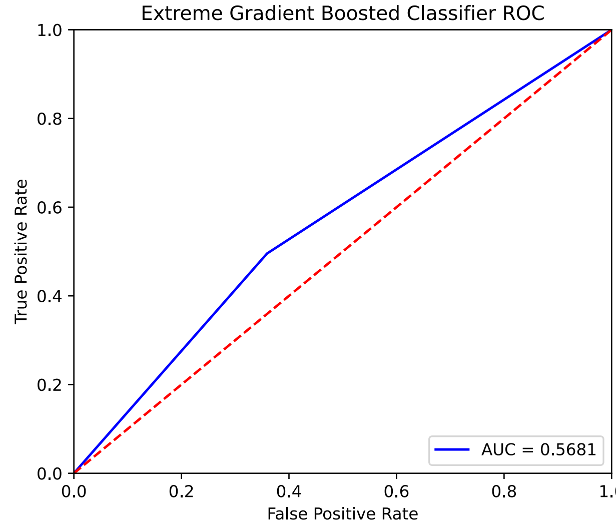
<Figure size 432x288 with 0 Axes>

In [75]:
```
import xgboost as xgb
from xgboost.sklearn import XGBClassifier
from train_xgboost import xgboost_modelfit
```

In [76]:
```
xgb1 = XGBClassifier(learning_rate=0.1, n_estimators=27, max_depth=5, min_child_weight=
```

In [77]:
```
validate_model(xgb1, x_train, y_train, 'Extreme Gradient Boosted Classifier', group_nam
```






```
<Figure size 432x288 with 0 Axes>
```

Hyperparameter Tuning

In [82]:

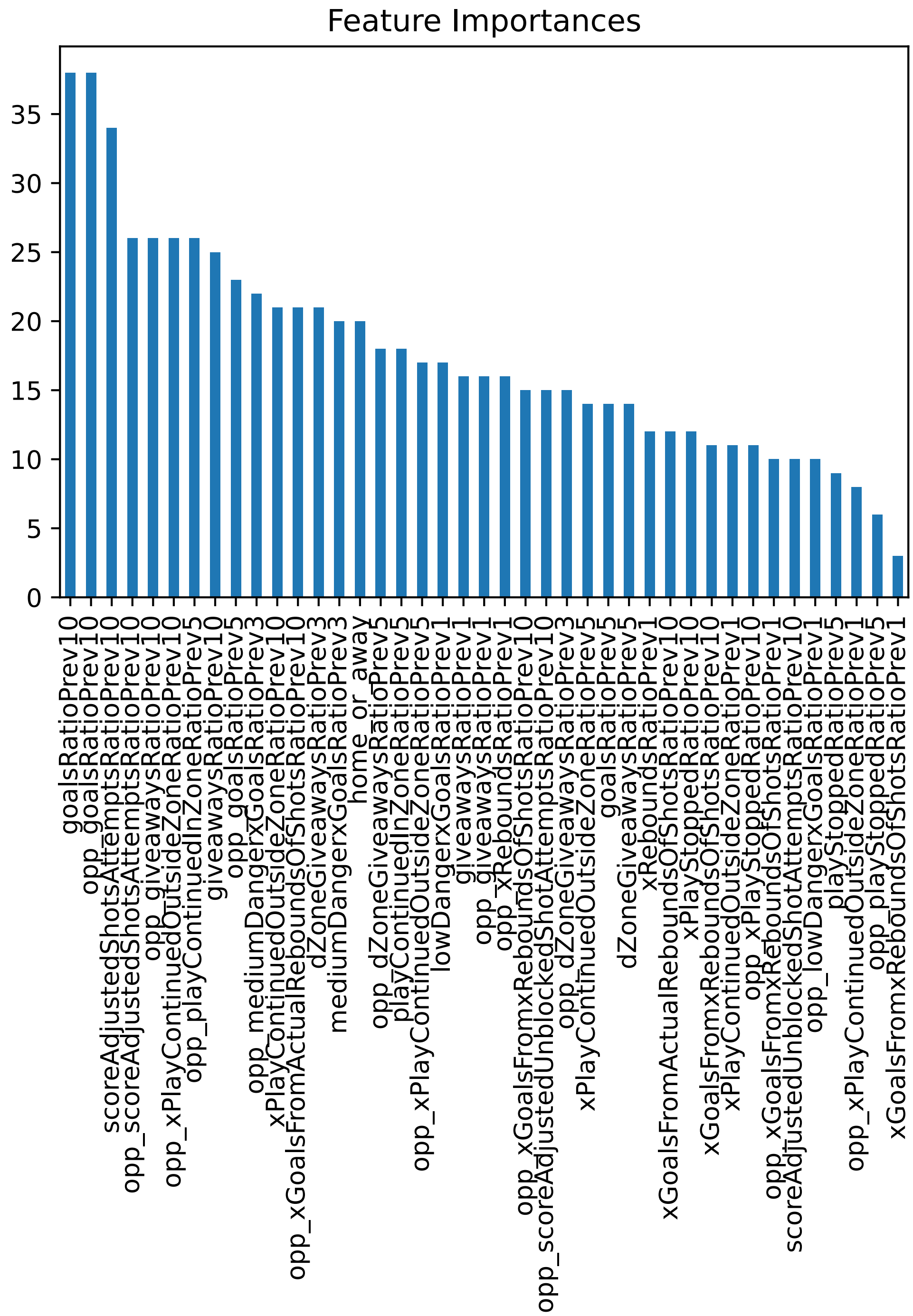
```
from sklearn import metrics
def xgboost_modelfit(model, x_train, y_train, x_test, y_test, cv_folds=5, early_stoppin

    xgb_param = model.get_xgb_params()
    xgtrain = xgb.DMatrix(x_train.values, label=y_train.values)
    cvresult = xgb.cv(xgb_param, xgtrain, num_boost_round=model.get_params()['n_estimat
                      metrics='auc', early_stopping_rounds=early_stopping_rounds)
    n_estimators = cvresult.shape[0]
    model.set_params(n_estimators=n_estimators)
    print(f"Optimal number of estimators: {n_estimators}")

    #Fit the model on the data
    model.fit(x_train, y_train, eval_metric='auc')

    #Predict training set:
    train_pred = model.predict(x_train)
    train_predprob = model.predict_proba(x_train)[:,1]
    test_pred = model.predict(x_test)
    test_predprob = model.predict_proba(x_test)[:,1]

    #Print model report:
    print("\nModel Report")
    print(f"Accuracy (Train): {metrics.accuracy_score(y_train.values, train_pred):.4g}"
    print(f"AUC Score (Train): {metrics.roc_auc_score(y_train, train_predprob):.4f}")
    print(f"Accuracy (Test): {metrics.accuracy_score(y_test.values, test_pred):.4g}")
    print(f"AUC Score (Test): {metrics.roc_auc_score(y_test, test_predprob):.4f}")

    feat_imp = pd.Series(model.get_booster().get_fscore()).sort_values(ascending=False)
    feat_imp.plot(kind='bar', title='Feature Importances')
    # plt.ylabel('Feature Importance Score')?
    return n_estimators
```

In [83]:

```
model = XGBClassifier(learning_rate=0.1, n_estimators=1000, max_depth=5, min_child_weig
n_estimators = xgboost_modelfit(xgb1, x_train, y_train, x_test, y_test)
```

```
Optimal number of estimators: 27
```

```
Model Report
Accuracy (Train): 0.7082
AUC Score (Train): 0.7764
Accuracy (Test): 0.5461
AUC Score (Test): 0.5703
```



In [84]:
```
max_depth_weight_grid = {
 'max_depth':range(1,10,1),
 'min_child_weight':range(1,6,1)
}
model = XGBClassifier(learning_rate=0.1, n_estimators=n_estimators, gamma=0, subsample=
                      colsample_bytree=0.8, objective= 'binary:logistic', scale_pos_wei
max_depth_weight_test = GridSearchCV(estimator=model, param_grid=max_depth_weight_grid,
max_depth_weight_test.fit(x_train, y_train)
depth = max_depth_weight_test.best_params_['max_depth']
```

```
min_child_weight = max_depth_weight_test.best_params_['min_child_weight']
max_depth_weight_test.best_params_, max_depth_weight_test.best_score_
```

Out[84]:
```
({'max_depth': 3, 'min_child_weight': 4}, 0.5940407847281838)
```

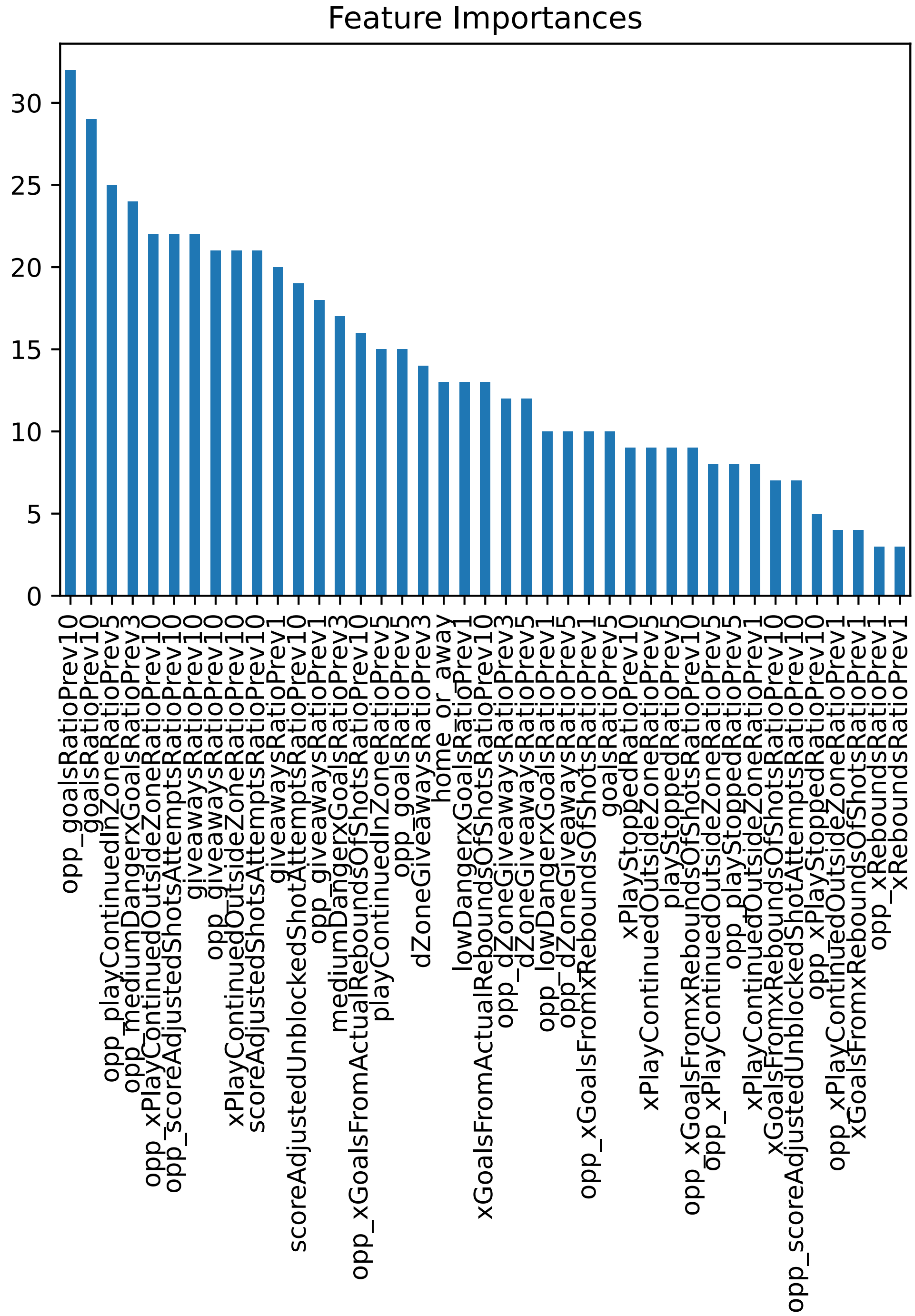
In [85]:
```
gamma_grid = {
 'gamma':[i/10.0 for i in range(0,5)]
}
model = XGBClassifier(learning_rate=0.1, n_estimators=n_estimators, max_depth=depth, mi
                      subsample=0.8, colsample_bytree=0.8, objective= 'binary:logistic'
gamma_test = GridSearchCV(estimator = model, param_grid=gamma_grid, scoring='roc_auc',
gamma_test.fit(x_train, y_train)
gamma = gamma_test.best_params_['gamma']
gamma_test.best_params_, gamma_test.best_score_
```

Out[85]:
```
({'gamma': 0.0}, 0.5940407847281838)
```

Recalibrate the number of estimators after tuning "amx_depth, "max_child_weight" and "gamma"

In [86]:
```
model = XGBClassifier(learning_rate=0.1, n_estimators=1000, max_depth=depth, min_child_
                      subsample=0.8, colsample_bytree=0.8, objective= 'binary:logistic'
n_estimators = xgboost_modelfit(model, x_train, y_train, x_test, y_test)
```

```
Optimal number of estimators: 88

Model Report
Accuracy (Train): 0.6771
AUC Score (Train): 0.7460
Accuracy (Test): 0.5434
AUC Score (Test): 0.5757
```



In [87]:
```
subsample_grid = {
    'subsample':[i/10.0 for i in range(4,10)],
    'colsample_bytree':[i/10.0 for i in range(4,10)]
}
model = XGBClassifier(learning_rate=0.1, n_estimators=n_estimators, max_depth=depth, mi
                      subsample=0.8, colsample_bytree=0.8, objective= 'binary:logistic'
subsample_test = GridSearchCV(estimator = model, param_grid=subsample_grid, scoring='ro
subsample_test.fit(x_train, y_train)
subsample = subsample_test.best_params_['subsample']
colsample_bytree = subsample_test.best_params_['colsample_bytree']
subsample_test.best_params_, subsample_test.best_score_
```

Out[87]:
```
({'colsample_bytree': 0.5, 'subsample': 0.9}, 0.5924137078554688)
```

In [88]:
```
refined_subsample_grid = {
```

```
    'subsample':[i/100.0 for i in range(85,95)],
    'colsample_bytree':[i/100.0 for i in range(45,55)]
}
model = XGBClassifier(learning_rate=0.1, n_estimators=n_estimators, max_depth=depth, mi
                      subsample=0.8, colsample_bytree=0.8, objective= 'binary:logistic'
refined_subsample_test = GridSearchCV(estimator = model, param_grid=refined_subsample_g
refined_subsample_test.fit(x_train, y_train)
subsample = refined_subsample_test.best_params_['subsample']
colsample_bytree = refined_subsample_test.best_params_['colsample_bytree']
refined_subsample_test.best_params_, refined_subsample_test.best_score_
```

Out[88]: ({'colsample_bytree': 0.45, 'subsample': 0.89}, 0.5938197377746796)

In [89]:

```
reg_alpha_grid = {
    'reg_alpha':[1e-5, 1e-2, 0.1, 1, 100]
}
model = XGBClassifier(learning_rate=0.1, n_estimators=n_estimators, max_depth=depth, mi
                      gamma=gamma, subsample=subsample, colsample_bytree=colsample_bytr
                      scale_pos_weight=1, seed=7)
reg_alpha_test = GridSearchCV(estimator = model, param_grid=reg_alpha_grid, scoring='ro
reg_alpha_test.fit(x_train, y_train)
reg_alpha = reg_alpha_test.best_params_['reg_alpha']
reg_alpha_test.best_params_, reg_alpha_test.best_score_
```

Out[89]: ({'reg_alpha': 1e-05}, 0.5938197377746796)

In [90]:

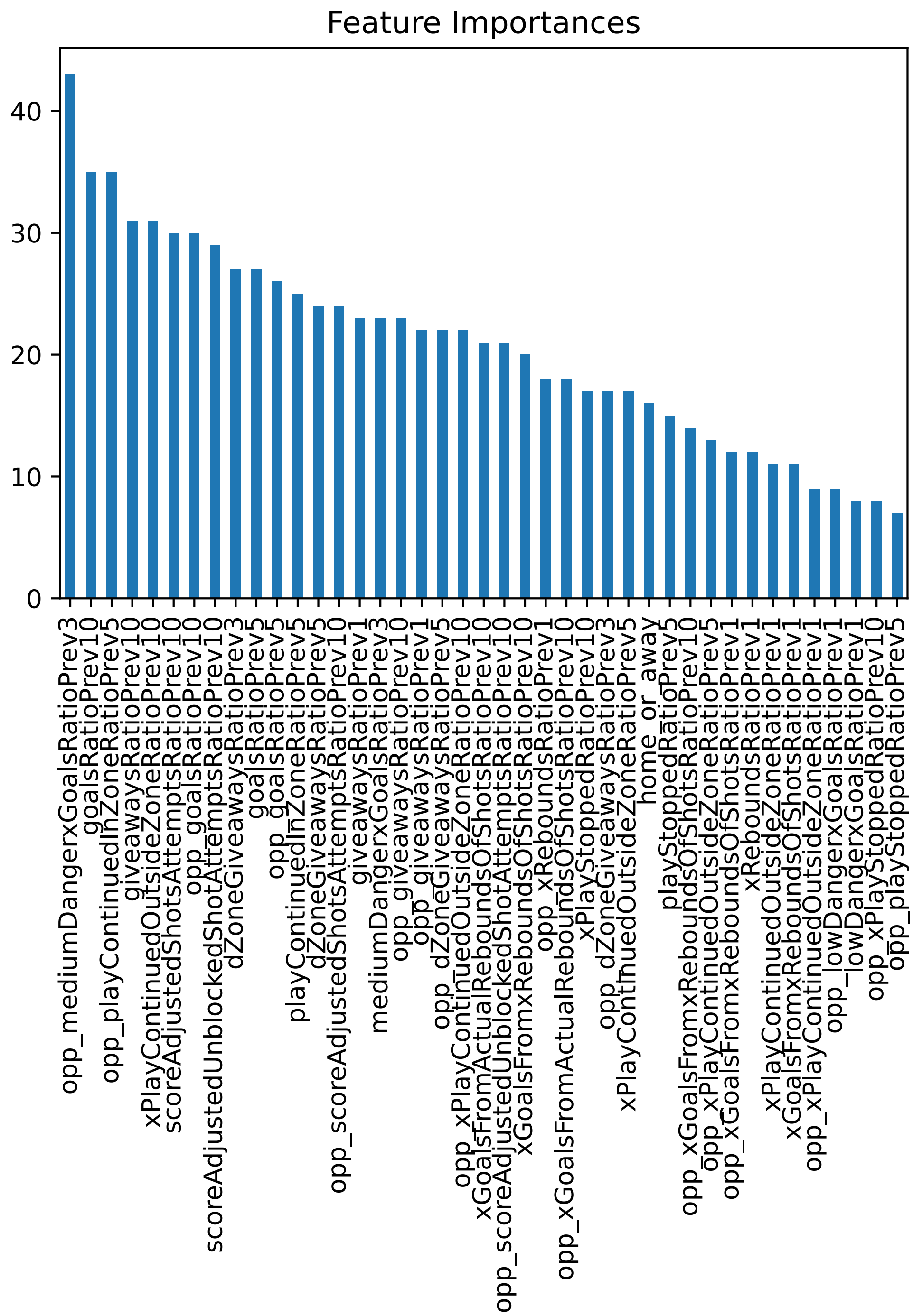
```
reg_alpha_grid = {
    'reg_alpha':[0, 1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2]
}
model = XGBClassifier(learning_rate=0.1, n_estimators=n_estimators, max_depth=depth, mi
                      gamma=gamma, subsample=subsample, colsample_bytree=colsample_bytr
                      scale_pos_weight=1, seed=7)
reg_alpha_test = GridSearchCV(estimator = model, param_grid=reg_alpha_grid, scoring='ro
reg_alpha_test.fit(x_train, y_train)
reg_alpha = reg_alpha_test.best_params_['reg_alpha']
reg_alpha_test.best_params_, reg_alpha_test.best_score_
```

Out[90]: ({'reg_alpha': 0.001}, 0.5938205219703143)

In [91]:

```
tuned_model = XGBClassifier(learning_rate=0.1, n_estimators=1000, max_depth=depth, min_
                      gamma=gamma, subsample=subsample, colsample_bytree=colsample_bytr
                      objective= 'binary:logistic', scale_pos_weight=1, seed=7)
n_estimators = xgboost_modelfit(tuned_model, x_train, y_train, x_test, y_test)
```

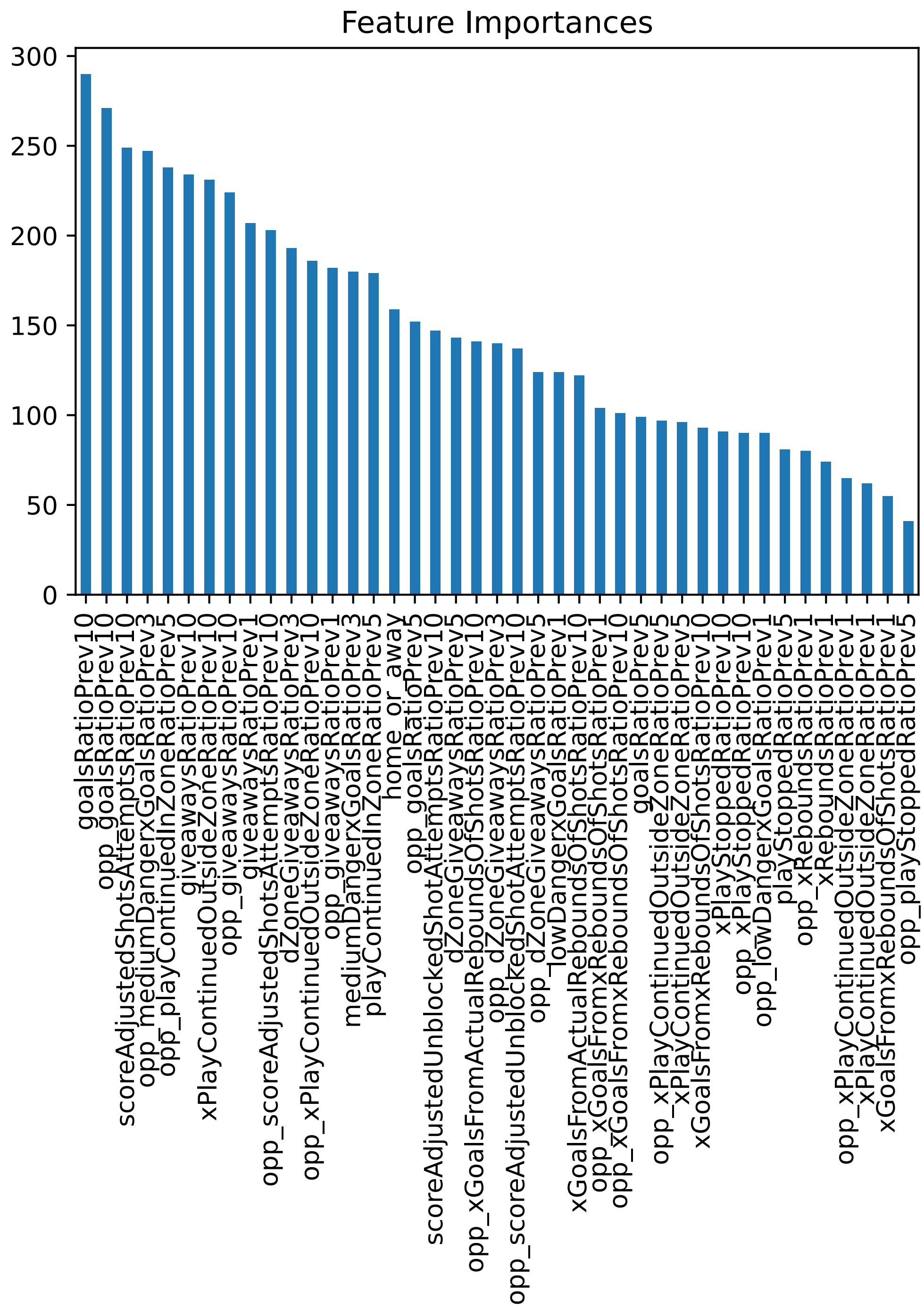
```
Optimal number of estimators: 139

Model Report
Accuracy (Train): 0.7001
AUC Score (Train): 0.7719
Accuracy (Test): 0.5389
AUC Score (Test): 0.5609
```



In [92]:

```
expensive_tuned_model = XGBClassifier(learning_rate=0.01, n_estimators=3000, max_depth=
                        gamma=gamma, subsample=subsample, colsample_bytree=colsample_bytr
                        objective= 'binary:logistic', scale_pos_weight=1, seed=7)
n_estimators = xgboost_modelfit(expensive_tuned_model, x_train, y_train, x_test, y_test
```

```
Optimal number of estimators: 953

Model Report
Accuracy (Train): 0.6811
AUC Score (Train): 0.7498
Accuracy (Test): 0.5506
AUC Score (Test): 0.5768
```


Feature Importances
300
250
200
150
100
50
0
goalsRatioPrev10
opp_goalsRatioPrev10
scoreAdjustedShotsAttemptsRatioPrev10
opp_mediumDangerxGoalsRatioPrev3
opp_playContinuedInZoneRatioPrev5
giveawaysRatioPrev10
xPlayContinuedOutsideZoneRatioPrev10
opp_giveawaysRatioPrev10
giveawaysRatioPrev1
opp_scoreAdjustedShotsAttemptsRatioPrev10
dZoneGiveawaysRatioPrev3
opp_xPlayContinuedOutsideZoneRatioPrev10
opp_giveawaysRatioPrev1
mediumDangerxGoalsRatioPrev3
playContinuedInZoneRatioPrev5
home_or_away
opp_goalsRatioPrev5
scoreAdjustedUnblockedShotAttemptsRatioPrev10
dZoneGiveawaysRatioPrev5
opp_xGoalsFromActualReboundsOfShotsRatioPrev10
opp_dZoneGiveawaysRatioPrev3
opp_scoreAdjustedUnblockedShotAttemptsRatioPrev10
opp_dZoneGiveawaysRatioPrev5
lowDangerxGoalsRatioPrev1
xGoalsFromActualReboundsOfShotsRatioPrev10
opp_xGoalsFromxReboundsOfShotsRatioPrev1
opp_xGoalsFromxReboundsOfShotsRatioPrev10
goalsRatioPrev5
opp_xPlayContinuedOutsideZoneRatioPrev5
xPlayContinuedOutsideZoneRatioPrev5
xGoalsFromxReboundsOfShotsRatioPrev10
xPlayStoppedRatioPrev10
opp_xPlayStoppedRatioPrev10
opp_lowDangerxGoalsRatioPrev1
playStoppedRatioPrev5
opp_xReboundsRatioPrev1
xReboundsRatioPrev1
opp_xPlayContinuedOutsideZoneRatioPrev1
xPlayContinuedOutsideZoneRatioPrev1
xGoalsFromxReboundsOfShotsRatioPrev1
opp_playStoppedRatioPrev5