notebook_final

May 26, 2021

1 Final Project Submission

- Student name: Jonathan Lee
- Student pace: Full Time
- Scheduled project review date/time: May 26, 2pm
- Instructor name: James Irving
- Blog post URL: https://github.com/ds-papes/dsc-phase-3-project

1.1 TABLE OF CONTENTS

Click to jump to matching Markdown Header.

2 INTRODUCTION

2.1 Business Problem

Just like in any traditional sports, there are multiple elements eSports there are many different aspects of a match that contribute to the outcome of either a win or a loss. This analysis focuses on using various machine learning algorithms to create a model based on data collected within the first 10 minutes of a high-ranking League of Legends match which as accurately as possible predicts the outcome of the match. Based on the resulting models, we will identify what elements of the game have the highest impact on the outcome of a match, and how an eSports coach should plan his/her team's training program.

3 OBTAIN

3.1 Data Understanding

The data we will use to perform this analysis was obtained from this Kaggle dataset which was obtained via the Riot API. It includes data from 9,879 high ranking (Diamond I to Master) competitive matches with 19 features per team and one target variable which indicates whether the match resulted in a win for the blue team.

Glossary of Features:

- Ward: An item that players can place on the map to reveal the nearby area. Very useful for map/objectives control.
- Assist: Awards partial gold and experience points when damage is done to contribute to an enemy's death.
- Elite Monsters: Monsters with high hp/damage that give a massive bonus (gold/XP/stats) when killed by a team.
- Dragon: AKA Drake. This powerful neutral monster grants various permanent effects and buffs when when killed by a team.
- Herald: A monster that spawns on the eigth minute. Grants a buff that allows the user to spawn the Herald for your team to help push towers and lanes.
- Tower: A structure that blocks the enemy's path to the base. They take high damage and fire at opponents within a certain radius.
- Gold: Currency awarded for killing monsters or enemy players as well as for completing objectives.
- Level: Champion level. Start at 1. Max is 18.
- Minions: Non-player characters (NPCs) that spawn from each team's base.
- Jungle Minions: NPC that belong to NO TEAM. They give gold and temporary buffs when killed by players.

[1]:
```python
# Import packages to be used in notebook.
import pandas as pd
import numpy as np
import seaborn as sns

import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec

from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression, LogisticRegressionCV
from sklearn.ensemble import RandomForestClassifier
from sklearn import metrics

from xgboost import XGBRFClassifier, XGBClassifier

import warnings
warnings.filterwarnings('ignore')

%matplotlib inline
```

[2]:
```python
# Load data and display basic info.
df = pd.read_csv('data/high_diamond_ranked_10min.csv')
display(df.head(5), df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9879 entries, 0 to 9878
Data columns (total 40 columns):
 #   Column                        Non-Null Count  Dtype  
---  ------                        --------------  -----  
 0   gameId                        9879 non-null   int64  
 1   blueWins                      9879 non-null   int64  
 2   blueWardsPlaced               9879 non-null   int64  
 3   blueWardsDestroyed            9879 non-null   int64  
 4   blueFirstBlood                9879 non-null   int64  
 5   blueKills                     9879 non-null   int64  
 6   blueDeaths                    9879 non-null   int64  
 7   blueAssists                   9879 non-null   int64  
 8   blueEliteMonsters             9879 non-null   int64  
 9   blueDragons                   9879 non-null   int64  
 10  blueHeralds                   9879 non-null   int64  
 11  blueTowersDestroyed           9879 non-null   int64  
 12  blueTotalGold                 9879 non-null   int64  
 13  blueAvgLevel                  9879 non-null   float64
 14  blueTotalExperience           9879 non-null   int64  
 15  blueTotalMinionsKilled        9879 non-null   int64  
 16  blueTotalJungleMinionsKilled  9879 non-null   int64  
 17  blueGoldDiff                  9879 non-null   int64  
 18  blueExperienceDiff            9879 non-null   int64  
 19  blueCSPerMin                  9879 non-null   float64
 20  blueGoldPerMin                9879 non-null   float64
 21  redWardsPlaced                9879 non-null   int64  
 22  redWardsDestroyed             9879 non-null   int64  
 23  redFirstBlood                 9879 non-null   int64  
 24  redKills                      9879 non-null   int64  
 25  redDeaths                     9879 non-null   int64  
 26  redAssists                    9879 non-null   int64  
 27  redEliteMonsters              9879 non-null   int64  
 28  redDragons                    9879 non-null   int64  
 29  redHeralds                    9879 non-null   int64  
 30  redTowersDestroyed            9879 non-null   int64  
 31  redTotalGold                  9879 non-null   int64  
 32  redAvgLevel                   9879 non-null   float64
 33  redTotalExperience            9879 non-null   int64  
 34  redTotalMinionsKilled         9879 non-null   int64  
 35  redTotalJungleMinionsKilled   9879 non-null   int64  
 36  redGoldDiff                   9879 non-null   int64  
 37  redExperienceDiff             9879 non-null   int64  
 38  redCSPerMin                   9879 non-null   float64
 39  redGoldPerMin                 9879 non-null   float64
dtypes: float64(6), int64(34)
memory usage: 3.0 MB
```

```
      gameId  blueWins  blueWardsPlaced  blueWardsDestroyed  blueFirstBlood  \
0  4519157822         0               28                   2               1
1  4523371949         0               12                   1               0
2  4521474530         0               15                   0               0
3  4524384067         0               43                   1               0
4  4436033771         0               75                   4               0

   blueKills  blueDeaths  blueAssists  blueEliteMonsters  blueDragons  …  \
0          9           6           11                  0            0  …
1          5           5            5                  0            0  …
2          7          11            4                  1            1  …
3          4           5            5                  1            0  …
4          6           6            6                  0            0  …

   redTowersDestroyed  redTotalGold  redAvgLevel  redTotalExperience  \
0                   0         16567          6.8               17047
1                   1         17620          6.8               17438
2                   0         17285          6.8               17254
3                   0         16478          7.0               17961
4                   0         17404          7.0               18313

   redTotalMinionsKilled  redTotalJungleMinionsKilled  redGoldDiff  \
0                    197                           55         -643
1                    240                           52         2908
2                    203                           28         1172
3                    235                           47         1321
4                    225                           67         1004

   redExperienceDiff  redCSPerMin  redGoldPerMin
0                  8         19.7         1656.7
1               1173         24.0         1762.0
2               1033         20.3         1728.5
3                  7         23.5         1647.8
4               -230         22.5         1740.4

[5 rows x 40 columns]

None
```

We have all numerical data and fortunately no null values to address. However, we have more columns than the default display allows us to see, so we will adjust the pandas display option.

[3]:
```
# Set maximum number of columns displayed to 40.
pd.set_option('display.max_columns', 40)
df.head()
```

[3]:
```
      gameId  blueWins  blueWardsPlaced  blueWardsDestroyed  blueFirstBlood  \
0  4519157822         0               28                   2               1
```

```
1  4523371949         0             12                  1           0
2  4521474530         0             15                  0           0
3  4524384067         0             43                  1           0
4  4436033771         0             75                  4           0

   blueKills  blueDeaths  blueAssists  blueEliteMonsters  blueDragons  \
0          9           6           11                  0            0
1          5           5            5                  0            0
2          7          11            4                  1            1
3          4           5            5                  1            0
4          6           6            6                  0            0

   blueHeralds  blueTowersDestroyed  blueTotalGold  blueAvgLevel  \
0            0                    0          17210           6.6
1            0                    0          14712           6.6
2            0                    0          16113           6.4
3            1                    0          15157           7.0
4            0                    0          16400           7.0

   blueTotalExperience  blueTotalMinionsKilled  blueTotalJungleMinionsKilled  \
0                17039                     195                            36
1                16265                     174                            43
2                16221                     186                            46
3                17954                     201                            55
4                18543                     210                            57

   blueGoldDiff  blueExperienceDiff  blueCSPerMin  blueGoldPerMin  \
0           643                  -8          19.5          1721.0
1         -2908               -1173          17.4          1471.2
2         -1172               -1033          18.6          1611.3
3         -1321                  -7          20.1          1515.7
4         -1004                 230          21.0          1640.0

   redWardsPlaced  redWardsDestroyed  redFirstBlood  redKills  redDeaths  \
0              15                  6              0         6          9
1              12                  1              1         5          5
2              15                  3              1        11          7
3              15                  2              1         5          4
4              17                  2              1         6          6

   redAssists  redEliteMonsters  redDragons  redHeralds  redTowersDestroyed  \
0           8                 0           0           0                   0
1           2                 2           1           1                   1
2          14                 0           0           0                   0
3          10                 0           0           0                   0
4           7                 1           1           0                   0
```

```
   redTotalGold  redAvgLevel  redTotalExperience  redTotalMinionsKilled  \
0         16567          6.8               17047                    197
1         17620          6.8               17438                    240
2         17285          6.8               17254                    203
3         16478          7.0               17961                    235
4         17404          7.0               18313                    225

   redTotalJungleMinionsKilled  redGoldDiff  redExperienceDiff  redCSPerMin  \
0                           55         -643                  8         19.7
1                           52         2908               1173         24.0
2                           28         1172               1033         20.3
3                           47         1321                  7         23.5
4                           67         1004               -230         22.5

   redGoldPerMin
0         1656.7
1         1762.0
2         1728.5
3         1647.8
4         1740.4
```

4 SCRUB

4.1 Data Preparation

Since this dataset was collected via Riot's API, we will trust that the data is accurate and not perform any outlier removal. Another reason for including outliers in our analysis is to consider whether outliers in certain features have an impact on the outcome of a match. We also do not have any null values to address, and so we will use this stage of the analysis to create different versions of this dataset using different features to examine whether we can obtain different results during the modeling process.

The two different datasets we will prepare are as follows: - df_big: Unaltered dataframe with all original features included. - df_select: Altered dataframe with aggregate columns removed and only controllable features included.

[4]:
```python
# Drop gameId column, since this is simply an identifier for each match
# and should not be included as part of our models.
df.drop('gameId', axis=1, inplace=True)
df.head()
```

[4]:
```
   blueWins  blueWardsPlaced  blueWardsDestroyed  blueFirstBlood  blueKills  \
0         0               28                   2               1          9
1         0               12                   1               0          5
2         0               15                   0               0          7
3         0               43                   1               0          4
4         0               75                   4               0          6
```

```
   blueDeaths  blueAssists  blueEliteMonsters  blueDragons  blueHeralds  \
0           6           11                  0            0            0
1           5            5                  0            0            0
2          11            4                  1            1            0
3           5            5                  1            0            1
4           6            6                  0            0            0

   blueTowersDestroyed  blueTotalGold  blueAvgLevel  blueTotalExperience  \
0                    0          17210           6.6                17039
1                    0          14712           6.6                16265
2                    0          16113           6.4                16221
3                    0          15157           7.0                17954
4                    0          16400           7.0                18543

   blueTotalMinionsKilled  blueTotalJungleMinionsKilled  blueGoldDiff  \
0                     195                            36           643
1                     174                            43         -2908
2                     186                            46         -1172
3                     201                            55         -1321
4                     210                            57         -1004

   blueExperienceDiff  blueCSPerMin  blueGoldPerMin  redWardsPlaced  \
0                  -8          19.5          1721.0              15
1               -1173          17.4          1471.2              12
2               -1033          18.6          1611.3              15
3                  -7          20.1          1515.7              15
4                 230          21.0          1640.0              17

   redWardsDestroyed  redFirstBlood  redKills  redDeaths  redAssists  \
0                  6              0         6          9           8
1                  1              1         5          5           2
2                  3              1        11          7          14
3                  2              1         5          4          10
4                  2              1         6          6           7

   redEliteMonsters  redDragons  redHeralds  redTowersDestroyed  redTotalGold  \
0                 0           0           0                   0         16567
1                 2           1           1                   1         17620
2                 0           0           0                   0         17285
3                 0           0           0                   0         16478
4                 1           1           0                   0         17404

   redAvgLevel  redTotalExperience  redTotalMinionsKilled  \
0          6.8               17047                    197
1          6.8               17438                    240
2          6.8               17254                    203
```

```
3          7.0              17961                     235
4          7.0              18313                     225

   redTotalJungleMinionsKilled  redGoldDiff  redExperienceDiff  redCSPerMin  \
0                           55         -643                  8         19.7
1                           52         2908               1173         24.0
2                           28         1172               1033         20.3
3                           47         1321                  7         23.5
4                           67         1004               -230         22.5

   redGoldPerMin
0         1656.7
1         1762.0
2         1728.5
3         1647.8
4         1740.4
```

[5]:
```python
# Create df with no removed features.
df_big = df.copy()
```

[6]:
```python
# Create df with only target variable and directly controllable aspects of
# the game.
df_select = df[['blueWins','blueWardsPlaced', 'blueWardsDestroyed',
                'blueFirstBlood', 'blueKills', 'blueDeaths', 'blueAssists',
                'blueDragons', 'blueHeralds', 'blueTowersDestroyed',
                'blueTotalMinionsKilled', 'blueTotalJungleMinionsKilled',
                'redWardsPlaced', 'redWardsDestroyed',
                'redFirstBlood', 'redKills', 'redDeaths', 'redAssists',
                'redDragons', 'redHeralds', 'redTowersDestroyed',
                'redTotalMinionsKilled', 'redTotalJungleMinionsKilled']]
df_select.head()
```

[6]:
```
   blueWins  blueWardsPlaced  blueWardsDestroyed  blueFirstBlood  blueKills  \
0         0               28                   2               1          9
1         0               12                   1               0          5
2         0               15                   0               0          7
3         0               43                   1               0          4
4         0               75                   4               0          6

   blueDeaths  blueAssists  blueDragons  blueHeralds  blueTowersDestroyed  \
0           6           11            0            0                    0
1           5            5            0            0                    0
2          11            4            1            0                    0
3           5            5            0            1                    0
4           6            6            0            0                    0

   blueTotalMinionsKilled  blueTotalJungleMinionsKilled  redWardsPlaced  \
```

```
0                      195                           36              15
1                      174                           43              12
2                      186                           46              15
3                      201                           55              15
4                      210                           57              17

   redWardsDestroyed  redFirstBlood  redKills  redDeaths  redAssists  \
0                  6              0         6          9           8
1                  1              1         5          5           2
2                  3              1        11          7          14
3                  2              1         5          4          10
4                  2              1         6          6           7

   redDragons  redHeralds  redTowersDestroyed  redTotalMinionsKilled  \
0           0           0                   0                    197
1           1           1                   1                    240
2           0           0                   0                    203
3           0           0                   0                    235
4           1           0                   0                    225

   redTotalJungleMinionsKilled
0                           55
1                           52
2                           28
3                           47
4                           67
```

5 EXPLORE

At this stage, we will examine if there are any redundant features in our two datasets and if there is any high multicollinearity that we might need to address.

TotalExperience and TotalGold are both features that are aggregates of the other columns, so we will explore some visualizations to determine whether we can expect a correlation with our target variable.

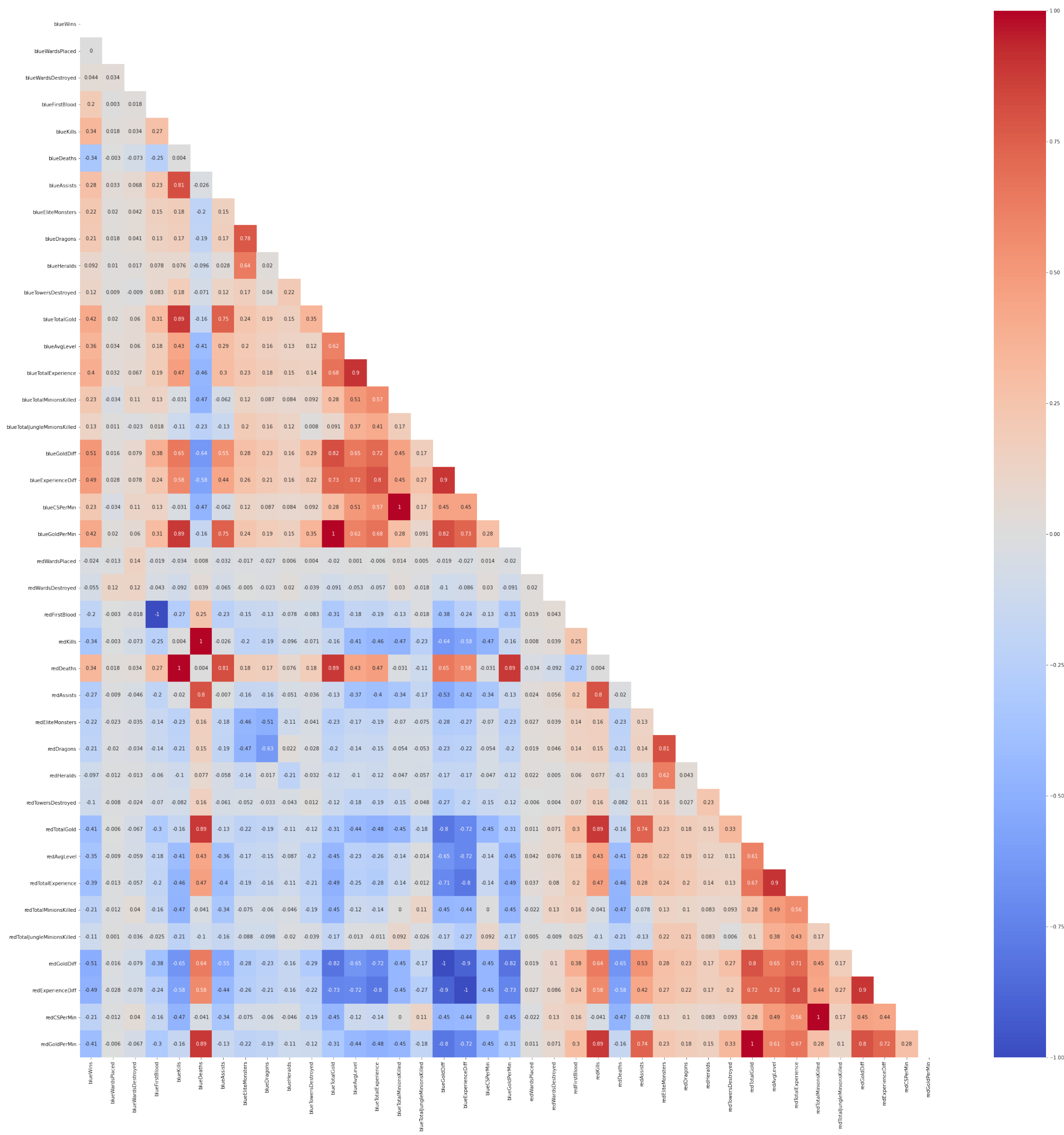
[7]:
```python
# Create functions to easily visualize correlation as well as general
# data distribution and outliers.

def corr_heatmap(df, digits=3, cmap='coolwarm'):
    """
    Creates a correlation heatmap to easily visualize multicollinearity
    that might be present in the dataframe.

    Args:
        df (DataFrame) : DataFrame with features to check multicollinearity on.
        digits (int) : Number of decimal places to display
```

```python
        cmap (str) : Colormap to display correlation range.

    Returns:
        fig : Matplotlib Figure
        ax : Matplotlib Axis
    """
    # Create correlation matrix from dataframe
    correl = df.corr().round(digits)
    correl

    # Create mask for upper triangle of matrix
    mask = np.zeros_like(correl)
    mask[np.triu_indices_from(mask)] = True

    #Create heatmap correlation matrix
    fig, ax = plt.subplots(figsize=((len(df.columns)),(len(df.columns))))
    sns.heatmap(correl, annot=True, ax=ax, cmap=cmap, vmin=-1, vmax=1,\
                mask=mask);
    return fig, ax


def visual_eda(df, target, col):
    """
    Plots a histogram + KDE, boxplot, and scatter plot with linear regression
    line of the specified column. Use to visualize shape of data, outliers,
    and check column's correlation with target variable.

    Args:
        df (DataFrame) : DataFrame containing column to plot
        target (str) : Name of target variable.
        col (str) : Name of the column to plot.

    Returns:
        fig : Matplotlib Figure
        gs : Matplotlib GridSpec
    """
    # Create copy variables of df and col
    data = df[col].copy()
    name = col

    # Calc mean and mean
    median = data.median().round(2)
    mean = data.mean().round(2)


    # Create gridspec for plots
    fig = plt.figure(figsize=(11, 6))
```

```python
    gs = GridSpec(nrows=2, ncols=2)

    ax0 = fig.add_subplot(gs[0, 0])
    ax1 = fig.add_subplot(gs[1, 0])
    ax2 = fig.add_subplot(gs[:, 1])

    # Plot distribution
    sns.histplot(data,alpha=0.5,stat='density',ax=ax0)
    sns.kdeplot(data,color='green',label='KDE',ax=ax0)
    ax0.set(ylabel='Density',title=name)
    ax0.set_title(F"Distribution of {name}")
    ax0.axvline(median,label=f'median={median:,}',color='black')
    ax0.axvline(mean,label=f'mean={mean:,}',color='black',ls=':')
    ax0.legend()

    # Plot Boxplot
    sns.boxplot(data,x=col,ax=ax1)
    ax1.set_title(F"Box Plot of {name}")

    # Plot Scatterplot to illustrate linearity
    sns.regplot(data=df, x=col, y=target, line_kws={"color": "red"}, ax=ax2)
    ax2.set_title(F"Scatter Plot of {name}")

    # Tweak Layout & Display
    fig.tight_layout()

    return fig, gs
```

[8]:
```python
# Create correlation heatmap for df_big.
corr_heatmap(df_big)
```

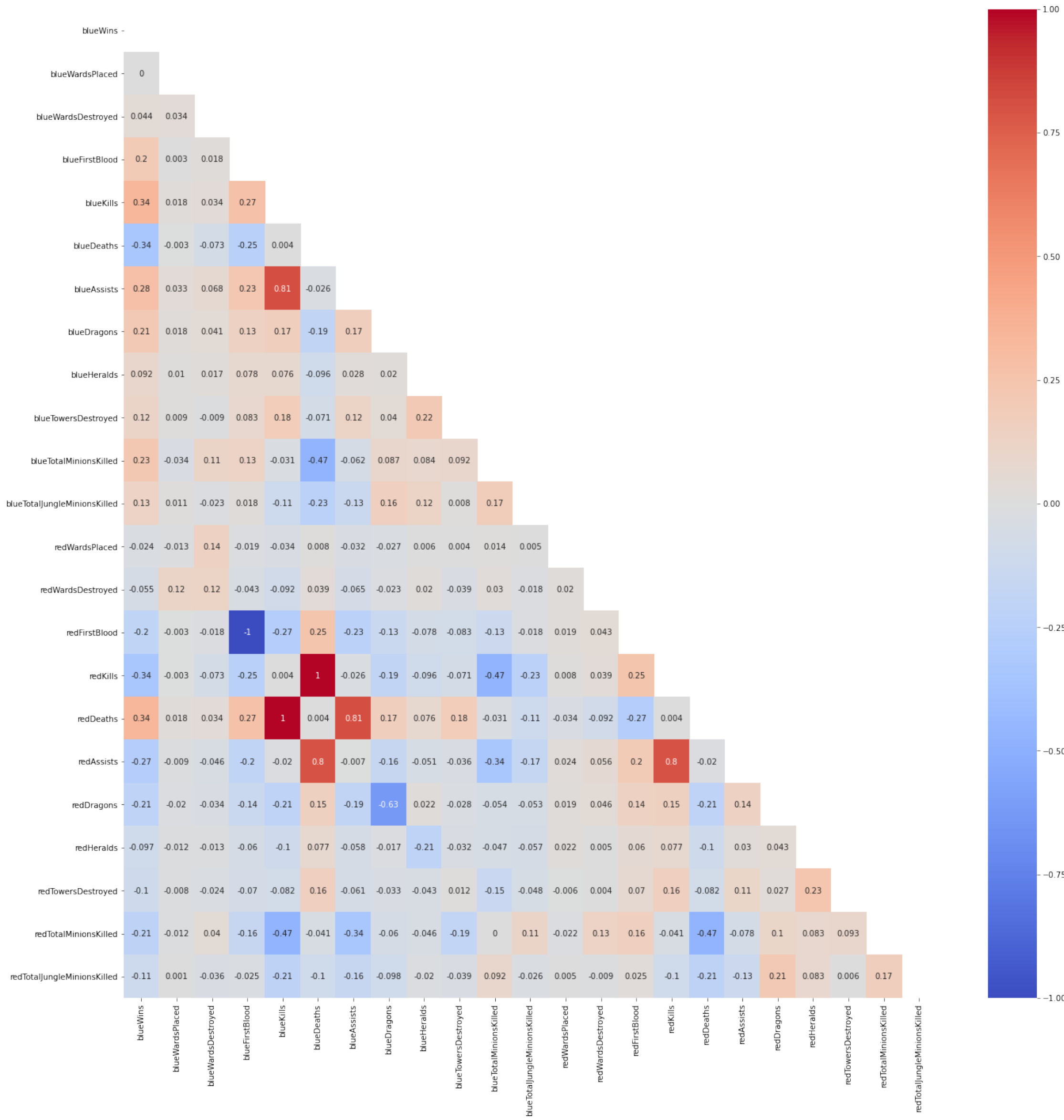
[8]:
```
(<Figure size 2808x2808 with 2 Axes>, <AxesSubplot:>)
```



We can see that there are multiple features that have high multicollinearity. This is a big problem when considering a logistic regression, and so we will avoid using df_big for our logistic regression model.

[9]:
```
# Create correlation heatmap for df_select.
corr_heatmap(df_select)
```

[9]: (<Figure size 1656x1656 with 2 Axes>, <AxesSubplot:>)



Even though multicollinearity is not as much of an issue in this dataframe, we still have some features with perfect multicollinearity: redFirstBlood, redKills, and redDeaths. These features are perfect inverses of blueFirstBlood, blueDeaths, and blueKills respectively, and so we will go ahead and remove those columns to prepare our dataset for logistic regression.

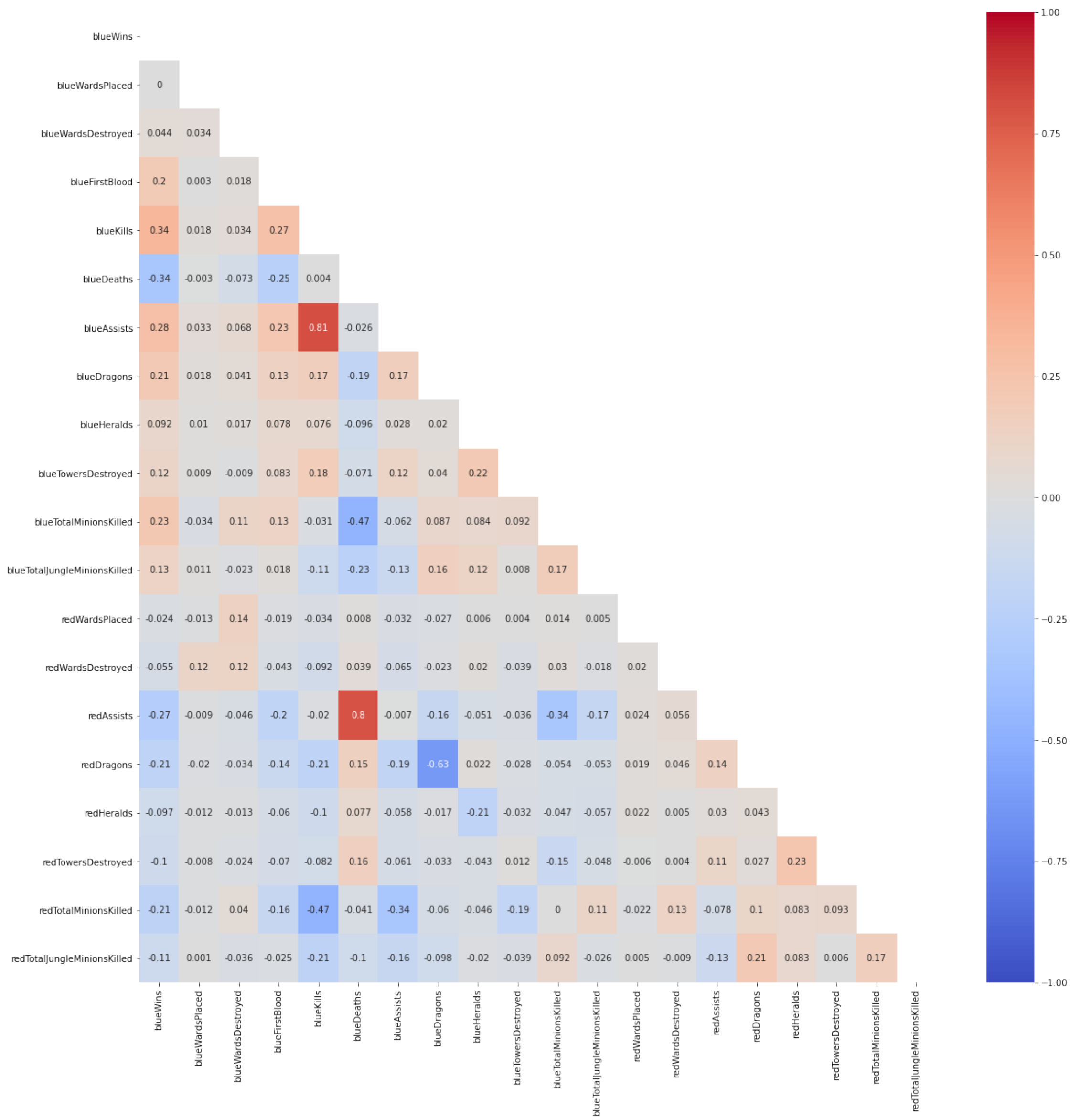
[10]:
```
# Drop highly columns with high multicollinearity.
df_select.drop(columns=['redKills', 'redDeaths', 'redFirstBlood'],
                inplace=True)
df_select.columns
```

[10]:
```
Index(['blueWins', 'blueWardsPlaced', 'blueWardsDestroyed', 'blueFirstBlood',
       'blueKills', 'blueDeaths', 'blueAssists', 'blueDragons', 'blueHeralds',
```

```
 'blueTowersDestroyed', 'blueTotalMinionsKilled',
 'blueTotalJungleMinionsKilled', 'redWardsPlaced', 'redWardsDestroyed',
 'redAssists', 'redDragons', 'redHeralds', 'redTowersDestroyed',
 'redTotalMinionsKilled', 'redTotalJungleMinionsKilled'],
dtype='object')
```

[11]:
```python
# Create correlation heatmap to verify that we no longer have
# multicollinearity.
corr_heatmap(df_select)
```

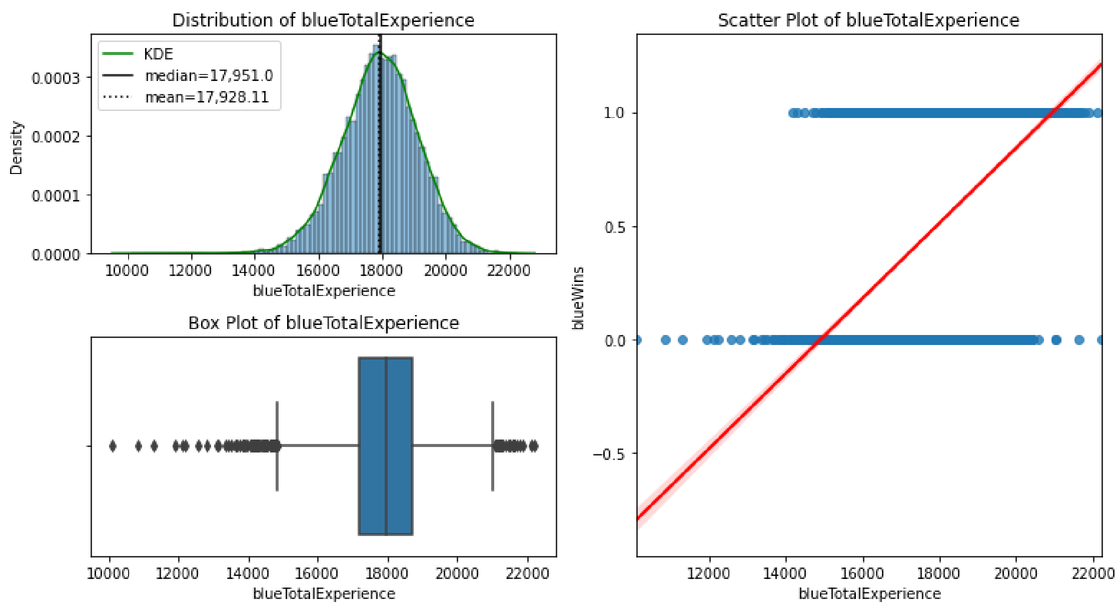
[11]: (<Figure size 1440x1440 with 2 Axes>, <AxesSubplot:>)



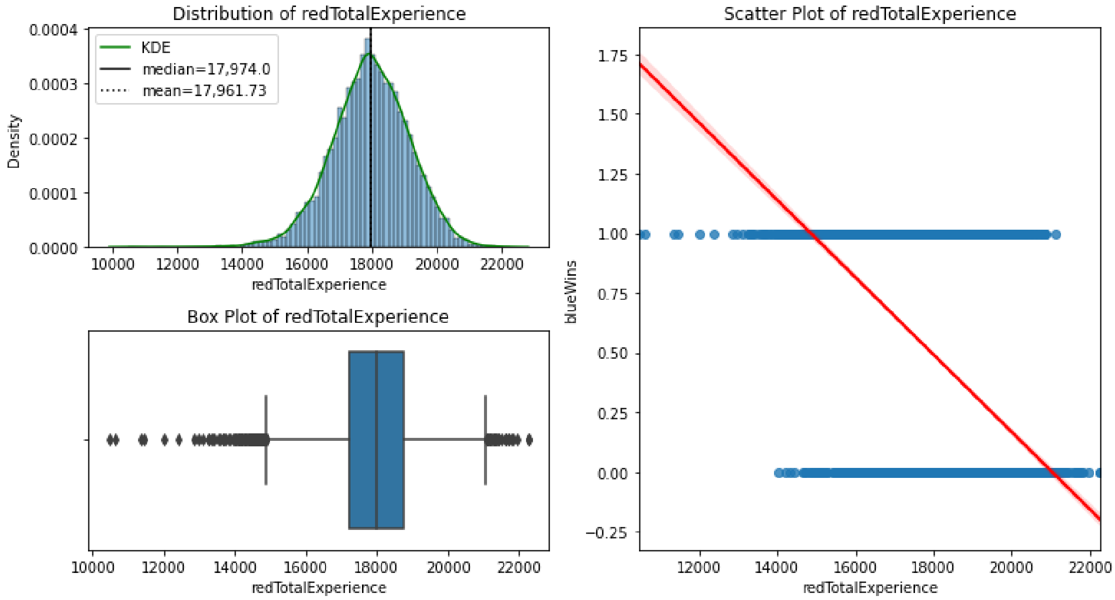
Although redAssists and blueAssists do have some with blueDeaths and blueKills respectively, we will leave those features in our dataframe since the correlation coefficients are not too high, and the impact of assists on the match outcome is still important to our analysis.

Next, we will examine the general distribution how the total experience and gold are correlated with our target variable in addition to their distributions and outliers.

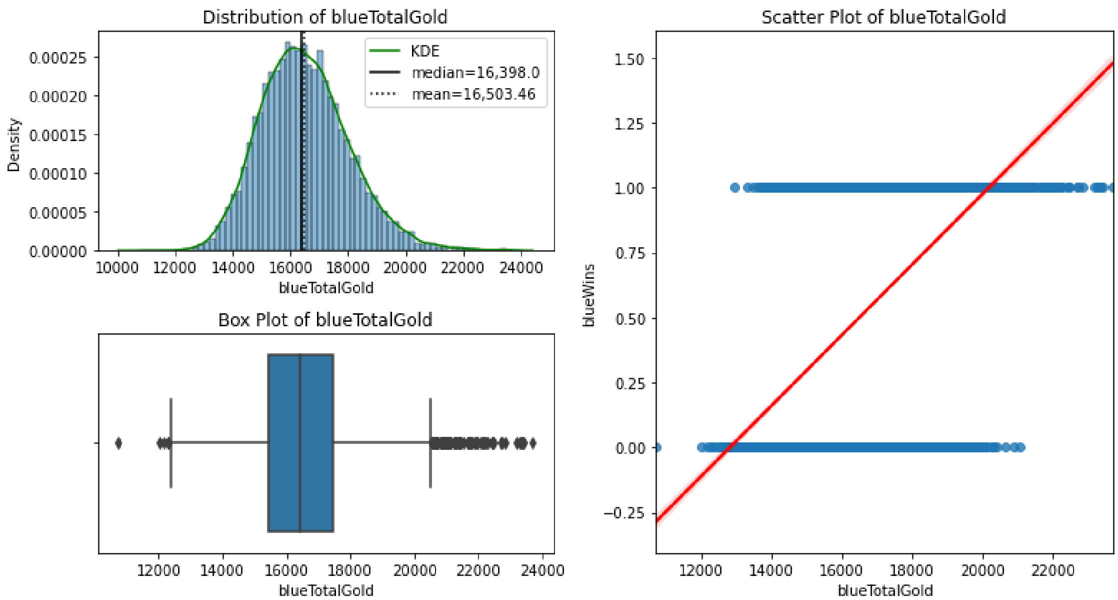
[12]:
```
# Plot visualization for blueTotalExperience vs blueWins.
visual_eda(df_big, 'blueWins', 'blueTotalExperience');
```



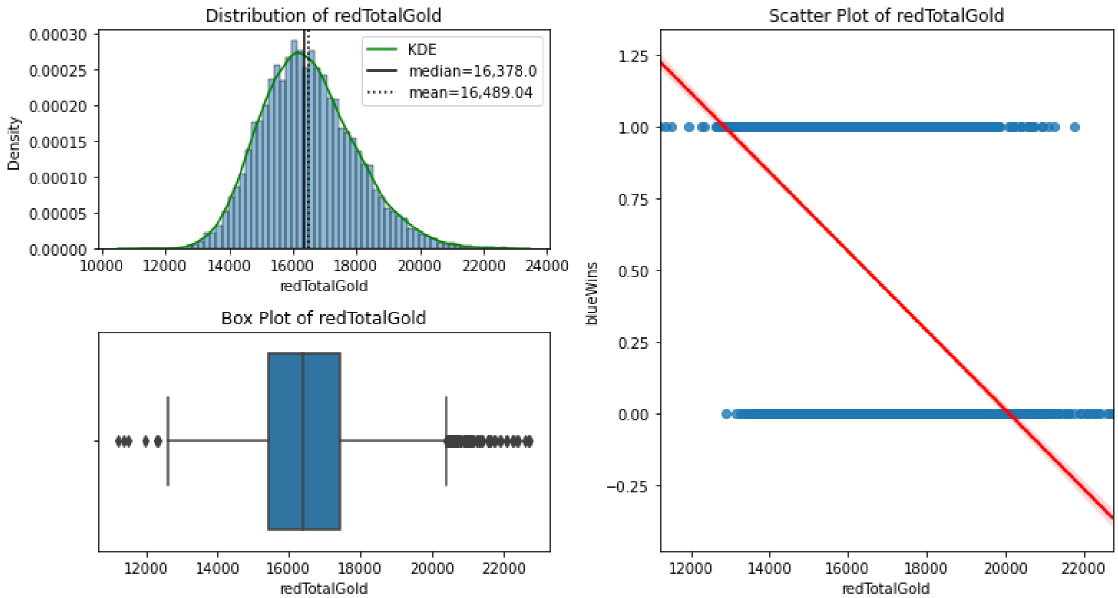
[13]:
```
# Plot visualization for redTotalExperience vs blueWins.
visual_eda(df_big, 'blueWins', 'redTotalExperience');
```



[14]:
```python
# Plot visualization for blueTotalGold vs blueWins.
visual_eda(df_big, 'blueWins', 'blueTotalGold');
```



[15]:
```python
# Plot visualization for redTotalGold vs blueWins.
visual_eda(df_big, 'blueWins', 'redTotalGold');
```



Again, we can see that we do have a lot of outliers, but the distribution of each of these features is normal. As you might have expected, we can see a generally negative correlation between red total gold and experience and a blue win, with a generally positive correlation between blue total gold and experience and a blue win.

6 MODEL

6.1 Data Modeling

Now that we have seen that there is some relationship between the total experience and gold and a team's win, we want to dive deeper into creating a model that puts together our features to as accurately as possible predict the outcome of a match and to identify which features have the highest impact on the match outcome.

In this section, we will cover the following three model types: 1. Logistic Regression 2. Random Forest 3. XGBoost: Random Forest

Logistic Regression will be the least computationally costly model, and so we will use this as a baseline to compare our other models and determine whether there is any value to using more complex models.

We will then move onto Random Forest and XGBoost models to see whether an ensemble method might provide a better predictive model, while also keeping in consideration the issue of overfitting.

For our Logistic Regression model, we will only use df_select since we have addressed the issue of multicollinearity specifically for this model. For our ensemble methods, we will pass through both df_select and df_big to determine whether a collection of all features provides us with better predictive ability than when we include only a subset of features.

[16]:
```python
# Create functions to facilitate scaling, fiting and evaluating multiple
# dataframes.

def evaluate_model(model, X_train, y_train, X_test, y_test, digits=4,
                   figsize=(10,5), params=False):
    """
    Displays evaluation metrics including classification report, confusion
    matrix, ROC-AUC curve.

    If the argument 'params' is passed, will display a table of the
    parameters hyperparameters used in the model.

    Args:
        df (DataFrame) : DataFrame with features to check multicollinearity on.
        model (classifier object) : Type of classificatier model to use.
        X_train (DataFrame) : Training data with feature variables.
        y_train (Series) : Training data with target variable.
        X_test (DataFrame) : Testing data with feature variables.
        y_test (Series) : Testing data with target variable.
        digits (int) : Colormap to display correlation range. Default is 4.
        figsize (int, int) : Figure dimensions. Default is (10,5)
        params (bool) : Prints table of hyperparameters used in model.

    Returns:
    """

    # Get Predictions
    y_hat_test = model.predict(X_test)
    y_hat_train = model.predict(X_train)

    # Classification Report / Scores

    print("****CLASSIFICATION REPORT - TRAINING DATA****")

    print(metrics.classification_report(y_train,y_hat_train, digits=digits))


    print("****CLASSIFICATION REPORT - TEST DATA****")

    print(metrics.classification_report(y_test,y_hat_test, digits=digits))

    print("****CONFUSION MATRIX AND ROC-AUC VISUALIZATION****")


    fig, axes = plt.subplots(ncols=2,
                             figsize=figsize)
```

```
    # Confusion Matrix
    metrics.plot_confusion_matrix(model, X_test,
                                  y_test,normalize='true',
                                 cmap='Purples',ax=axes[0])
    axes[0].set_title('Confusion Matrix')

    # Plot ROC Curve
    metrics.plot_roc_curve(model,X_test,y_test,ax=axes[1])

    ax = axes[1]
    ax.legend()
    ax.plot([0,1],[0,1], ls='-')
    ax.grid()
    ax.set_title('ROC AUC Curve')

    plt.show()

    if params == True:
        print("****MODEL PARAMETERS****")
        params = pd.DataFrame(pd.Series(model.get_params()))
        params.columns=['parameters']
        display(params)

def split_scale(df, target, scaler=StandardScaler()):
    """
    Creates train-test splits and scales training data.

    Args:
        df (DataFrame): DataFrame with features and target variable.
        target (str): Name of target variable.
        scaler (scaler object): Scaler to use on features DataFrame. Default
                                is StandardScaler.

    Returns:
        X_train (DataFrame) : Training data with scaled feature variables.
        y_train (Series) : Training data with target variable.
        X_test (DataFrame) : Testing data with scaled feature variables.
        y_test (Series) : Testing data with target variable.
    """


    # Separate X and y
    target = target
    y = df[target]
    X = df.drop(target, axis=1)

    # Train test split
```

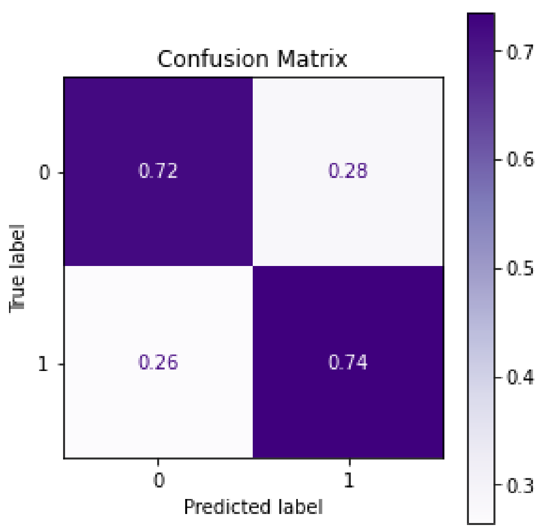
```
    X_train, X_test, y_train, y_test = train_test_split(X, y)

    # Get list of column names
    cols = X_train.columns

    # Scale columns
    scaler = scaler
    X_train = pd.DataFrame(scaler.fit_transform(X_train), columns=cols)
    X_test = pd.DataFrame(scaler.transform(X_test), columns=cols)

    return X_train, X_test, y_train, y_test

def fit_eval(model, X_train, y_train, X_test, y_test, digits=4,
             figsize=(10,5), params=False):
    """
    Fits model on training data and displays classification evaluation metrics.

    Args:
        model (classifier object) : Type of classificatier model to use.
        X_train (DataFrame) : Training data with feature variables.
        y_train (Series) : Training data with target variable.
        X_test (DataFrame) : Testing data with feature variables.
        y_test (Series) : Testing data with target variable.
        digits (int) : Colormap to display correlation range. Default is 4.
        figsize (int, int) : Figure dimensions. Default is (10,5)
        params (bool) : Prints table of hyperparameters used in model.

    Returns:
        model (classifier object) : Model after fitting on training data.
    """
    model = model

    model.fit(X_train, y_train)

    evaluate_model(model, X_train, y_train, X_test, y_test, digits=digits,
                   figsize=figsize, params=params)

    return model
```

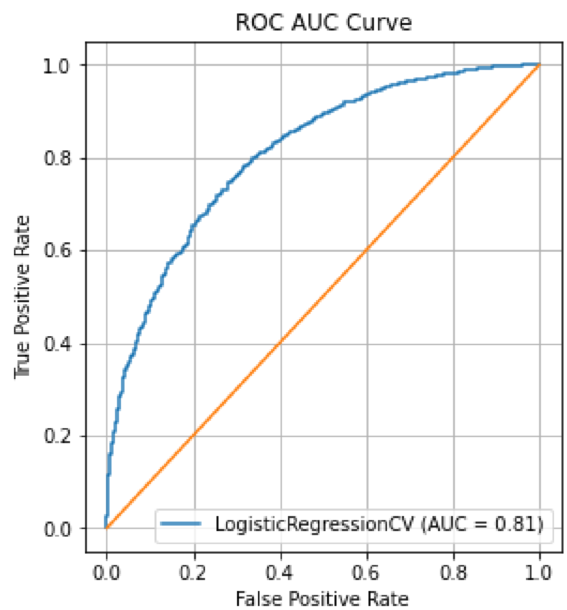
[17]:
```
# Create training and test data splits.
X_train_select, X_test_select, y_train_select, \
                y_test_select = split_scale(df_select, 'blueWins')
X_train_big, X_test_big, y_train_big, \
             y_test_big = split_scale(df_big, 'blueWins')
```

6.2 Logistic Regression

[18]:
```
# Fit and evaluate df_select on a Logistic Regression model.
log_select = fit_eval(LogisticRegressionCV(random_state=42), \
                      X_train_select, y_train_select, \
                      X_test_select, y_test_select)
```

```
****CLASSIFICATION REPORT - TRAINING DATA****
              precision    recall  f1-score   support

           0     0.7190    0.7117    0.7153      3670
           1     0.7198    0.7269    0.7234      3739

    accuracy                         0.7194      7409
   macro avg     0.7194    0.7193    0.7193      7409
weighted avg     0.7194    0.7194    0.7194      7409

****CLASSIFICATION REPORT - TEST DATA****
              precision    recall  f1-score   support

           0     0.7451    0.7201    0.7324      1279
           1     0.7099    0.7355    0.7225      1191

    accuracy                         0.7275      2470
   macro avg     0.7275    0.7278    0.7274      2470
weighted avg     0.7281    0.7275    0.7276      2470

****CONFUSION MATRIX AND ROC-AUC VISUALIZATION****
```



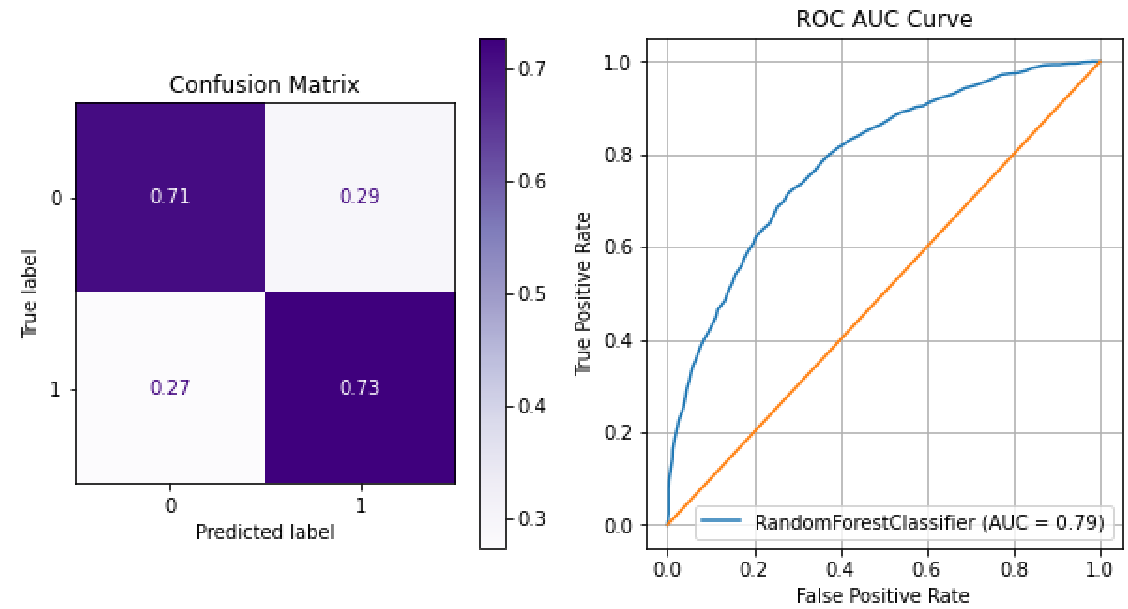


Not a bad starting point! We can see that our macro recall score is 0.7193 on the training data, on our test data received a macro recall score of 0.7278, meaning that of the true wins and losses, our Logistic Regression model is predicting 72.78% of them correctly. We also do not have an issue of under or overfitting.

6.3 Random Forest

[19]:
```
# Fit and evaluate Random Forest on df_select.
fit_eval(RandomForestClassifier(random_state=42), X_train_select, \
         y_train_select, X_test_select, y_test_select)
```

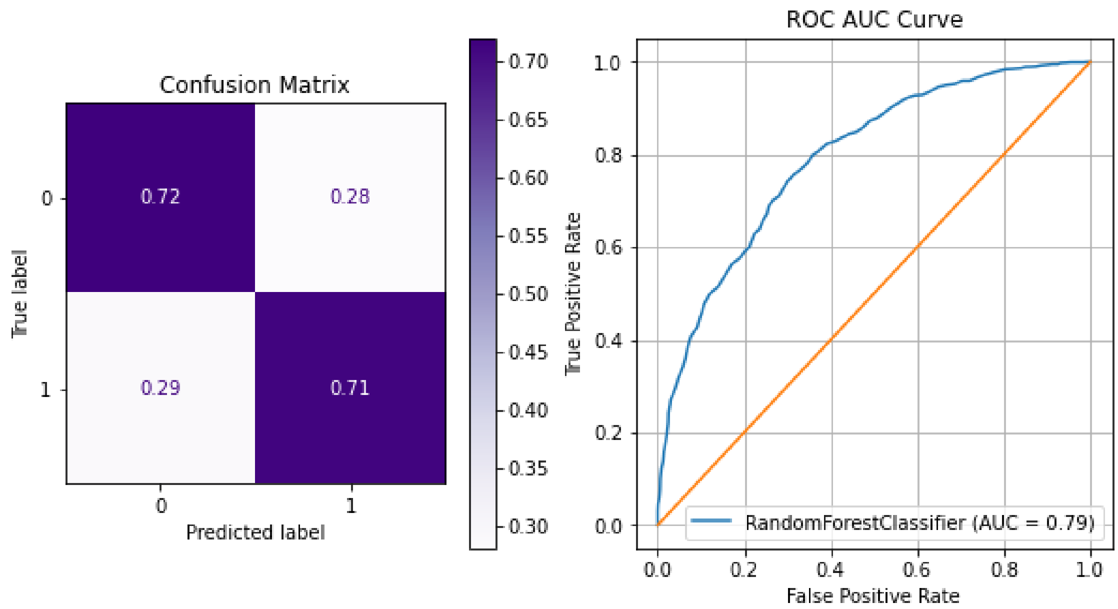
```
****CLASSIFICATION REPORT - TRAINING DATA****
              precision    recall  f1-score   support

           0     1.0000    1.0000    1.0000      3670
           1     1.0000    1.0000    1.0000      3739

    accuracy                         1.0000      7409
   macro avg     1.0000    1.0000    1.0000      7409
weighted avg     1.0000    1.0000    1.0000      7409

****CLASSIFICATION REPORT - TEST DATA****
              precision    recall  f1-score   support

           0     0.7352    0.7076    0.7211      1279
           1     0.6981    0.7263    0.7119      1191

    accuracy                         0.7166      2470
   macro avg     0.7167    0.7169    0.7165      2470
weighted avg     0.7173    0.7166    0.7167      2470

****CONFUSION MATRIX AND ROC-AUC VISUALIZATION****
```



[19]:
```
RandomForestClassifier(random_state=42)
```

[20]:
```
# Fit and evaluate Random Forest on df_big.
fit_eval(RandomForestClassifier(random_state=42), X_train_big, y_train_big, \
                                                  X_test_big, y_test_big)
```

```
****CLASSIFICATION REPORT - TRAINING DATA****
              precision    recall  f1-score   support

           0     1.0000    1.0000    1.0000      3676
           1     1.0000    1.0000    1.0000      3733

    accuracy                         1.0000      7409
   macro avg     1.0000    1.0000    1.0000      7409
weighted avg     1.0000    1.0000    1.0000      7409

****CLASSIFICATION REPORT - TEST DATA****
              precision    recall  f1-score   support

           0     0.7256    0.7188    0.7222      1273
           1     0.7039    0.7109    0.7074      1197

    accuracy                         0.7150      2470
   macro avg     0.7148    0.7149    0.7148      2470
weighted avg     0.7151    0.7150    0.7150      2470

****CONFUSION MATRIX AND ROC-AUC VISUALIZATION****
```



[20]: RandomForestClassifier(random_state=42)

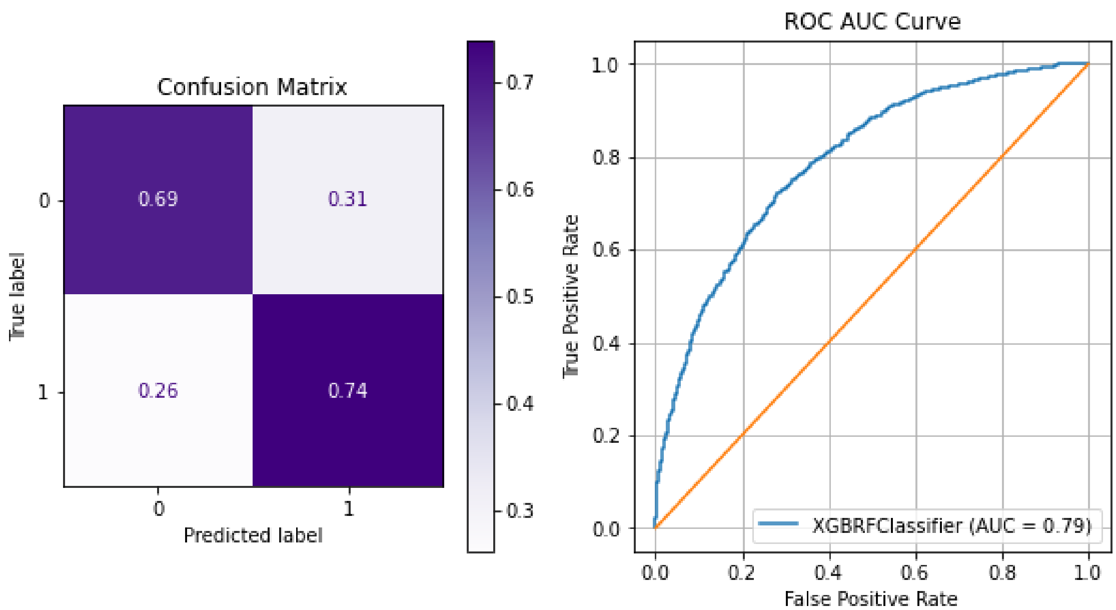
Although the recall scores from our Random Forest models being run on the test data are similar to that which we saw in our Logistic Regression, we can immediately see that we have an major issue of overfitting, as this model scores perfectly on the training data. In order to prevent overfitting, we will ideally use a gridsearch to find the optimal hyperparameters for this model and data.

6.4 XGBoost: Random Forest

[21]:
```
# Fit and evaluate XGBoost on df_select.
xgb_select = fit_eval(XGBRFClassifier(random_state=42), \
                      X_train_select, y_train_select, \
                      X_test_select, y_test_select)
```

```
****CLASSIFICATION REPORT - TRAINING DATA****
              precision    recall  f1-score   support

           0     0.7446    0.7229    0.7336      3670
           1     0.7356    0.7566    0.7459      3739

    accuracy                         0.7399      7409
   macro avg     0.7401    0.7398    0.7398      7409
weighted avg     0.7400    0.7399    0.7398      7409

****CLASSIFICATION REPORT - TEST DATA****
              precision    recall  f1-score   support
```

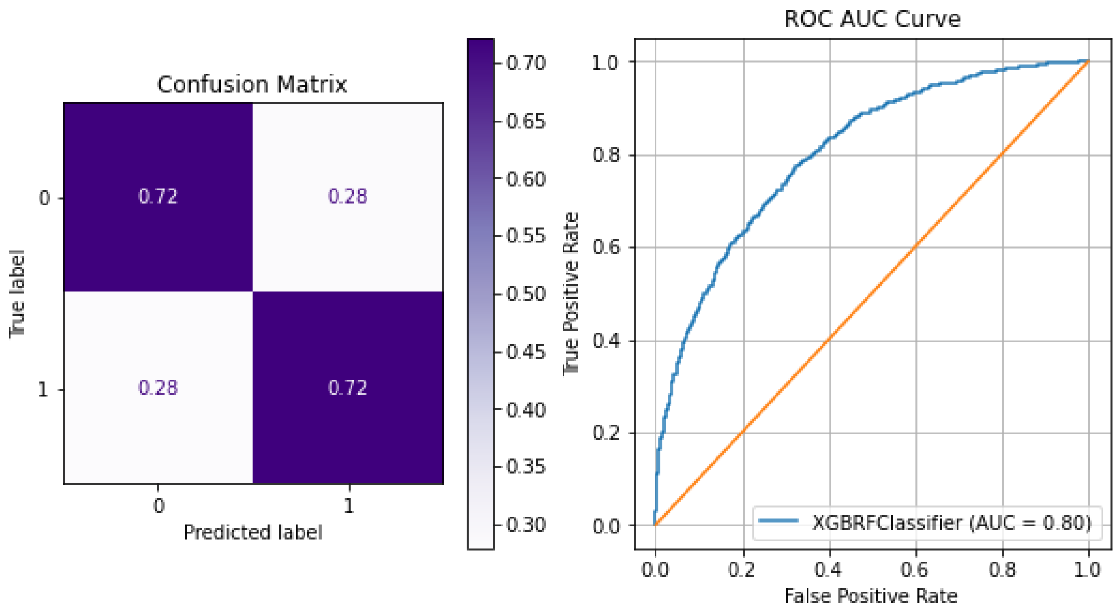
```
           0     0.7397    0.6912    0.7146      1279
           1     0.6902    0.7389    0.7137      1191

    accuracy                         0.7142      2470
   macro avg     0.7150    0.7150    0.7142      2470
weighted avg     0.7159    0.7142    0.7142      2470

****CONFUSION MATRIX AND ROC-AUC VISUALIZATION****
```



[22]:
```
# Fit and evaluate XGBoost on df_big.
fit_eval(XGBRFClassifier(random_state=42), \
         X_train_big, y_train_big, X_test_big, y_test_big)
```

```
****CLASSIFICATION REPORT - TRAINING DATA****
              precision    recall  f1-score   support

           0     0.7589    0.7748    0.7667      3676
           1     0.7735    0.7576    0.7655      3733

    accuracy                         0.7661      7409
   macro avg     0.7662    0.7662    0.7661      7409
weighted avg     0.7662    0.7661    0.7661      7409

****CLASSIFICATION REPORT - TEST DATA****
              precision    recall  f1-score   support

           0     0.7324    0.7180    0.7251      1273
```

```
           1     0.7062    0.7210    0.7135      1197

    accuracy                         0.7194      2470
   macro avg     0.7193    0.7195    0.7193      2470
weighted avg     0.7197    0.7194    0.7195      2470

****CONFUSION MATRIX AND ROC-AUC VISUALIZATION****
```




```
[22]: XGBRFClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
                      colsample_bytree=1, gamma=0, gpu_id=-1, importance_type='gain',
                      interaction_constraints='', max_delta_step=0, max_depth=6,
                      min_child_weight=1, missing=nan, monotone_constraints='()',
                      n_estimators=100, n_jobs=0, num_parallel_tree=100,
                      objective='binary:logistic', random_state=42, reg_alpha=0,
                      scale_pos_weight=1, tree_method='exact', validate_parameters=1,
                      verbosity=None)
```

We can see that using the base XGBoost model, we have a slightly better recall score than we saw with our Random Forest. The issue of overfitting has also been somewhat solved, but we do want to see if we can further address this issue.

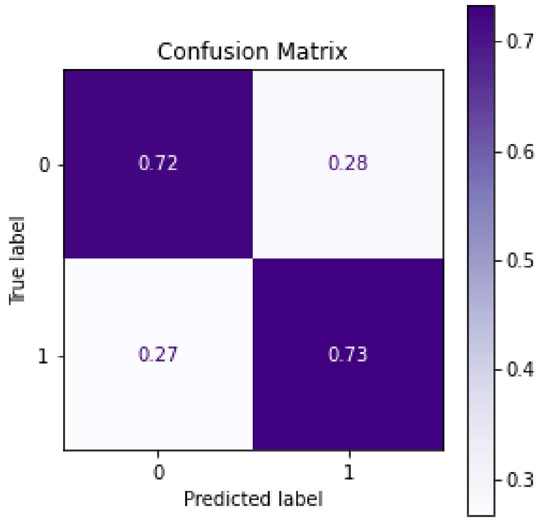
Although the difference in scores was not large, we will proceed to use a gridsearch on our XGBoost model and Logistic Regression model to see if we can completely address the issue of overfitting as well as hopefully improving our recall score.

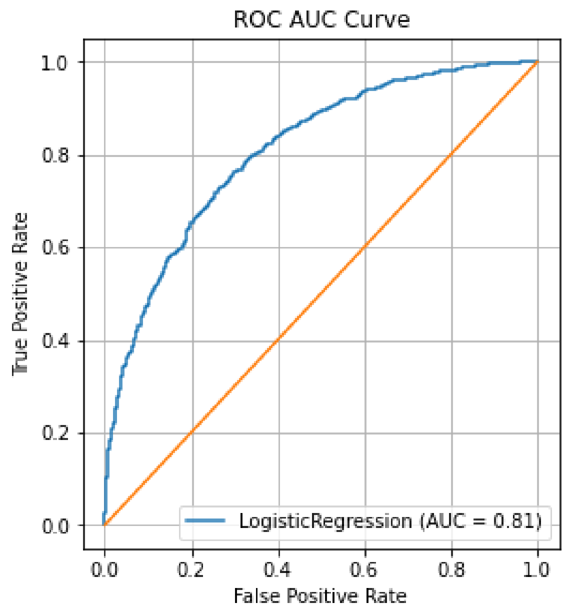
6.5 GridSearch CV - Logistic Regression

[23]:
```
# Create parameter grid for Logistic Regression gridsearch.
log_reg = LogisticRegression(random_state=42)

params = {'C': [0.001, 0.01, 0.1, 1, 10, 100,1e6,1e12],
          'penalty': ['l1', 'l2', 'elastic_net'],
          'fit_intercept': [True, False],
          'solver':["liblinear", "newton-cg", "lbfgs", "sag","saga"],
          'class_weight': ['balanced']}
log_grid = GridSearchCV(log_reg, params, scoring='recall_macro')
```

[24]:
```
# Fit grid and evaluate best estimating model.
log_grid.fit(X_train_select, y_train_select)
evaluate_model(log_grid.best_estimator_, X_train_select, y_train_select, \
               X_test_select, y_test_select, params=True)
```

```
****CLASSIFICATION REPORT - TRAINING DATA****
              precision    recall  f1-score   support

           0     0.7170    0.7174    0.7172      3670
           1     0.7225    0.7221    0.7223      3739

    accuracy                         0.7198      7409
   macro avg     0.7198    0.7198    0.7198      7409
weighted avg     0.7198    0.7198    0.7198      7409

****CLASSIFICATION REPORT - TEST DATA****
              precision    recall  f1-score   support

           0     0.7442    0.7232    0.7335      1279
           1     0.7115    0.7330    0.7221      1191

    accuracy                         0.7279      2470
   macro avg     0.7278    0.7281    0.7278      2470
weighted avg     0.7284    0.7279    0.7280      2470

****CONFUSION MATRIX AND ROC-AUC VISUALIZATION****
```






```
****MODEL PARAMETERS****

                    parameters
C                          0.1
class_weight          balanced
dual                     False
fit_intercept             True
intercept_scaling            1
l1_ratio                  None
max_iter                   100
multi_class               auto
n_jobs                    None
penalty                     l1
random_state                42
solver                    saga
tol                     0.0001
verbose                      0
warm_start               False
```

We can see an improvement in our recall score of 0.16% compared to our base Logistic Regression model. Let's see if we can tune our hyperparameters to improve our score.

[25]:
```python
# Create parameter grid for Logistic Regression gridsearch.
log_reg_ref = LogisticRegression(random_state=42)

params = {'C': [0.0001, 0.001],
          'penalty': ['l1', 'l2', 'elastic_net'],
          'solver':["liblinear", "newton-cg", "lbfgs", "sag","saga"],
```

```
            'class_weight': ['balanced']}
log_grid_refined = GridSearchCV(log_reg_ref, params, scoring='recall_macro')
log_grid_refined
```

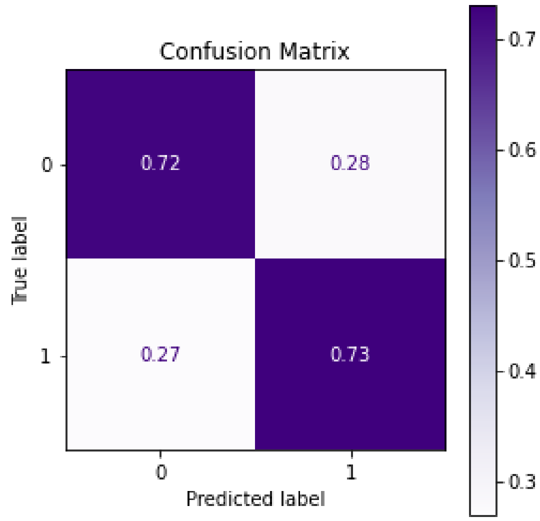
[25]:

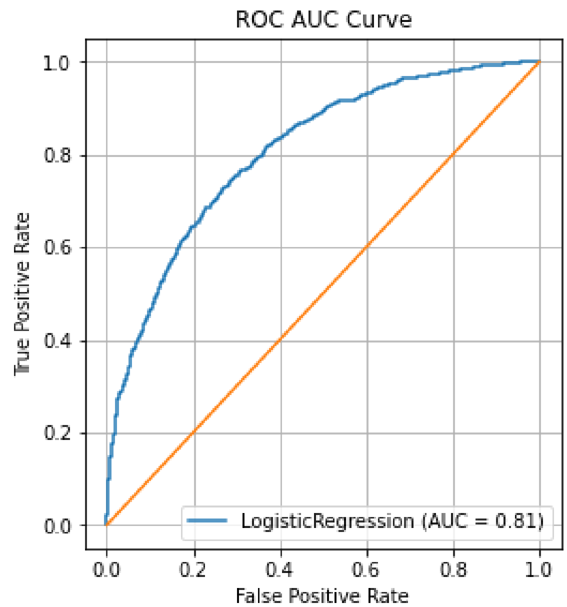
```
GridSearchCV(estimator=LogisticRegression(random_state=42),
             param_grid={'C': [0.0001, 0.001], 'class_weight': ['balanced'],
                         'penalty': ['l1', 'l2', 'elastic_net'],
                         'solver': ['liblinear', 'newton-cg', 'lbfgs', 'sag',
                                    'saga']},
             scoring='recall_macro')
```

[26]:

```
# Fit grid and evaluate best estimating model.
log_grid_refined.fit(X_train_select, y_train_select)
evaluate_model(log_grid_refined.best_estimator_, X_train_select, \
               y_train_select, X_test_select, y_test_select, params=True)
```

```
****CLASSIFICATION REPORT - TRAINING DATA****
              precision    recall  f1-score   support

           0     0.7158    0.7158    0.7158      3670
           1     0.7210    0.7210    0.7210      3739

    accuracy                         0.7185      7409
   macro avg     0.7184    0.7184    0.7184      7409
weighted avg     0.7185    0.7185    0.7185      7409

****CLASSIFICATION REPORT - TEST DATA****
              precision    recall  f1-score   support

           0     0.7424    0.7232    0.7327      1279
           1     0.7108    0.7305    0.7205      1191

    accuracy                         0.7267      2470
   macro avg     0.7266    0.7268    0.7266      2470
weighted avg     0.7271    0.7267    0.7268      2470

****CONFUSION MATRIX AND ROC-AUC VISUALIZATION****
```






```
****MODEL PARAMETERS****

                      parameters
C                          0.001
class_weight            balanced
dual                       False
fit_intercept               True
intercept_scaling              1
l1_ratio                    None
max_iter                     100
multi_class                 auto
n_jobs                      None
penalty                       l2
random_state                  42
solver                 newton-cg
tol                       0.0001
verbose                        0
warm_start                 False
```

At this point, we can see that our recall score is starting to drop, and so we can see that we may have hit the maximum score possible with a Logistic Regression. Hence, we will keep log_grid.best_estimator_ as our best Logistic Regressoin model so far.

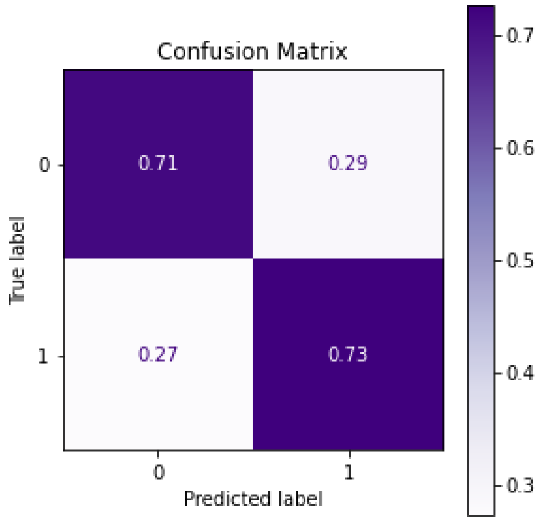
6.6 GridSearch CV - XGBoost: Random Forest

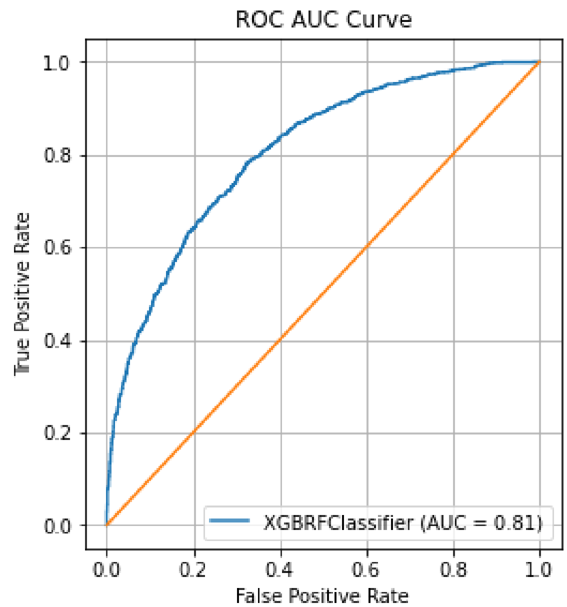
Next, we will try to improve our recall score on our XGBoost model while addressing the slight issue of overfitting. Since we had a better score on df_big where we left our features unaltered, we will proceed with that dataframe.

[27]:
```
# Create parameter grid for XGBoost Random Forest gridsearch.
xgb_rf = XGBRFClassifier(random_state=42)

params = {'learning_rate': [0.03, 0.05, 0.06],
          'max_depth': [4, 5, 6],
          'min_child_weight': [2, 3, 4],
          'subsample': [0.03, 0.4, 0.5],
          'n_estimators': [100]}
xgb_grid = GridSearchCV(xgb_rf, params, scoring='recall_macro')
```

[28]:
```
# Fit grid and evaluate best estimating model.
xgb_grid.fit(X_train_big, y_train_big)
evaluate_model(xgb_grid.best_estimator_, X_train_big, y_train_big, X_test_big,␣
 ↪y_test_big, params=True)
```

```
****CLASSIFICATION REPORT - TRAINING DATA****
              precision    recall  f1-score   support

           0     0.7501    0.7587    0.7544      3676
           1     0.7597    0.7511    0.7554      3733

    accuracy                         0.7549      7409
   macro avg     0.7549    0.7549    0.7549      7409
weighted avg     0.7549    0.7549    0.7549      7409

****CLASSIFICATION REPORT - TEST DATA****
              precision    recall  f1-score   support

           0     0.7346    0.7133    0.7238      1273
           1     0.7042    0.7260    0.7149      1197

    accuracy                         0.7194      2470
   macro avg     0.7194    0.7196    0.7194      2470
weighted avg     0.7199    0.7194    0.7195      2470

****CONFUSION MATRIX AND ROC-AUC VISUALIZATION****
```





****MODEL PARAMETERS****

	parameters
colsample_bynode	0.8
learning_rate	0.03
reg_lambda	1e-05
subsample	0.4
objective	binary:logistic
base_score	0.5
booster	gbtree
colsample_bylevel	1
colsample_bytree	1
gamma	0
gpu_id	-1
importance_type	gain
interaction_constraints	
max_delta_step	0
max_depth	6
min_child_weight	4
missing	NaN
monotone_constraints	()
n_estimators	100
n_jobs	0
num_parallel_tree	100
random_state	42
reg_alpha	0
scale_pos_weight	1
tree_method	exact

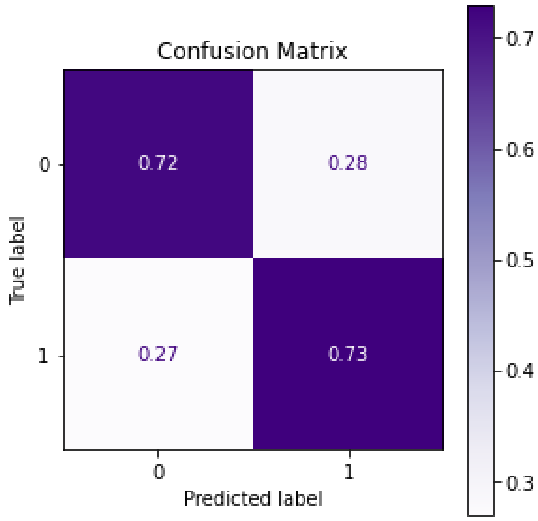
```
validate_parameters                      1
verbosity                             None
```

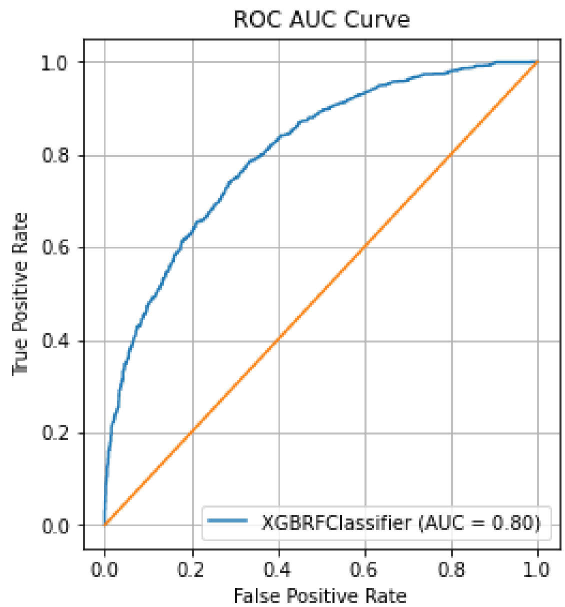
We see an improvement in our recall score by 0.05% which is tiny, but let's see if we can tune our hyperparameters a bit further.

[29]:
```
# Create parameter grid for XGBoost Random Forest gridsearch.
xgb_rf_ref = XGBRFClassifier(random_state=42)

params = {'learning_rate': [0.0001, 0.001],
          'max_depth': [4, 5, 6],
          'min_child_weight': [3, 4, 5],
          'subsample': [0.3, 0.5, 0.7],
          'n_estimators': [100]}
xgb_grid_refined = GridSearchCV(xgb_rf, params, scoring='recall_macro')
```

[30]:
```
# Fit grid and evaluate best estimating model.
xgb_grid_refined.fit(X_train_big, y_train_big)
evaluate_model(xgb_grid_refined.best_estimator_, X_train_big, y_train_big,␣
 ↪X_test_big, y_test_big, params=True)
```

```
****CLASSIFICATION REPORT - TRAINING DATA****
              precision    recall  f1-score   support

           0     0.7559    0.7726    0.7642      3676
           1     0.7711    0.7544    0.7626      3733

    accuracy                         0.7634      7409
   macro avg     0.7635    0.7635    0.7634      7409
weighted avg     0.7636    0.7634    0.7634      7409

****CLASSIFICATION REPORT - TEST DATA****
              precision    recall  f1-score   support

           0     0.7381    0.7172    0.7275      1273
           1     0.7080    0.7293    0.7185      1197

    accuracy                         0.7231      2470
   macro avg     0.7231    0.7233    0.7230      2470
weighted avg     0.7235    0.7231    0.7231      2470

****CONFUSION MATRIX AND ROC-AUC VISUALIZATION****
```





****MODEL PARAMETERS****

	parameters
colsample_bynode	0.8
learning_rate	0.0001
reg_lambda	1e-05
subsample	0.7
objective	binary:logistic
base_score	0.5
booster	gbtree
colsample_bylevel	1
colsample_bytree	1
gamma	0
gpu_id	-1
importance_type	gain
interaction_constraints	
max_delta_step	0
max_depth	6
min_child_weight	3
missing	NaN
monotone_constraints	()
n_estimators	100
n_jobs	0
num_parallel_tree	100
random_state	42
reg_alpha	0
scale_pos_weight	1
tree_method	exact

```
validate_parameters                    1
verbosity                           None
```

We can see that with a macro recall score of 0.7319 on the testing data, this seems to be the model with the best predictive ability! We can also see that the score on the training data is 0.7495, showing that we do not have an issue of under or overfitting.

7 iNTERPRET

We started with a macro recall score of 0.7210 in our baseline Logistic Regression model, and through trying different modeling algorithms in combination with gridsearches, we were able to increase our macro recall score to 0.7319. This means that our final Logistic Regression model is capable of correctly identifying 72.26% of wins or losses based on the data collected within the first 10 minutes of each match, while our XGBoost model is able to correctly identify 73.19%.

Using our final Logistic Regression and XGBoost models, we can now extract the feature coefficients and importances in order to identify how much impact each of the elements of the game are likely to have on the outcome of each match. Although the model with the best predictive ability was our gridsearched XGBoost, we will proceed to explain feature importance with the Logistic Regression that was run on df_select in order to preserve interpretability of our values.

Based on these findings, we will be able to provide out final recommendations as to what our eSports coach should focus on while creating a training program for his/her team.

[31]:
```python
# Extract coefficients from log_grid.best_estimator_ model.
log_coeff = pd.Series(log_grid.best_estimator_.coef_.flatten(),
               index=X_train_select.columns).sort_values(ascending=False)
log_coeff
```

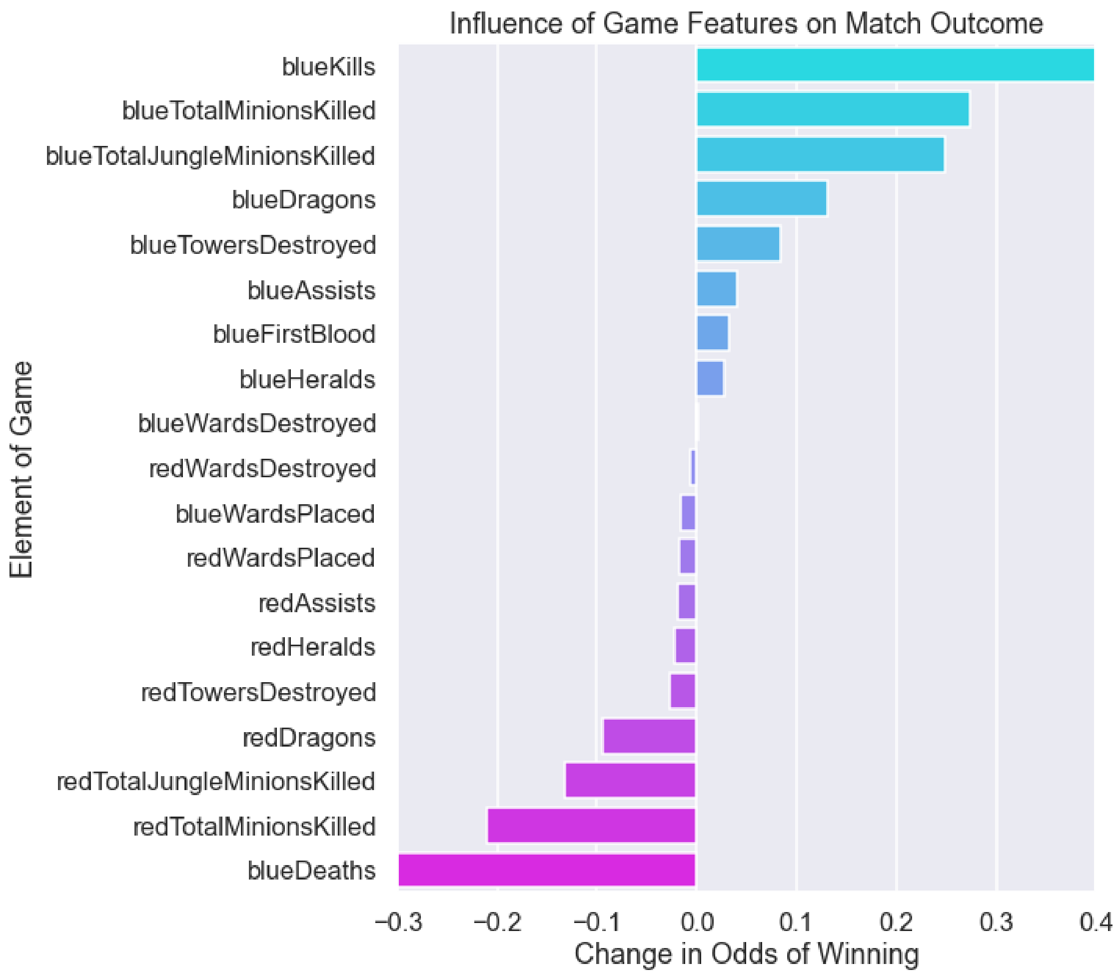
[31]:
```
blueKills                        0.704066
blueTotalMinionsKilled           0.241568
blueTotalJungleMinionsKilled     0.222610
blueDragons                      0.124049
blueTowersDestroyed              0.080384
blueAssists                      0.039246
blueFirstBlood                   0.032315
blueHeralds                      0.027994
blueWardsDestroyed               0.001299
redWardsDestroyed               -0.006859
blueWardsPlaced                 -0.016557
redWardsPlaced                  -0.018086
redAssists                      -0.018997
redHeralds                      -0.023298
redTowersDestroyed              -0.027525
redDragons                      -0.099197
redTotalJungleMinionsKilled     -0.142222
redTotalMinionsKilled           -0.236789
blueDeaths                      -0.702098
dtype: float64
```

[32]:
```python
# Convert log coefficients to odds and subtract 1 to display change in odds.
log_odds = np.exp(log_coeff) -1
log_odds
```

[32]:
```
blueKills                          1.021957
blueTotalMinionsKilled             0.273244
blueTotalJungleMinionsKilled       0.249334
blueDragons                        0.132072
blueTowersDestroyed                0.083703
blueAssists                        0.040027
blueFirstBlood                     0.032842
blueHeralds                        0.028390
blueWardsDestroyed                 0.001300
redWardsDestroyed                 -0.006835
blueWardsPlaced                   -0.016421
redWardsPlaced                    -0.017924
redAssists                        -0.018817
redHeralds                        -0.023029
redTowersDestroyed                -0.027149
redDragons                        -0.094436
redTotalJungleMinionsKilled       -0.132572
redTotalMinionsKilled             -0.210843
blueDeaths                        -0.504455
dtype: float64
```

[33]:
```python
# Set theme and style for plots.
sns.set_theme('talk')
sns.set_style('darkgrid')
```

[34]:
```python
# Create bar plot of feature coefficients as odds.
fig, ax = plt.subplots(figsize=(8,10))

sns.barplot(x=log_odds.values, y=log_odds.index, palette='cool', ax=ax,␣
 ↪orient='h')

ax.set_title('Influence of Game Features on Match Outcome')
ax.set_xlabel('Change in Odds of Winning')
ax.set_ylabel('Element of Game')
ax.set_xlim([-.3, .4]);

# ax.set_xticks([-.15,.15])
# ax.set_xticklabels(['Decrease in Odds','Increase in Odds'])
# ax.set_xticklabels(ax.get_xticklabels(), rotation=45, ha='right');
```



Our bar plot indicates that champion kills and deaths within the first 10 minutes of the match have by far the most impact on the outcome of a match. We can see that total lane minions and total jungle creeps and dragons are also of high importance. Surprisingly, Heralds, vision wards, and towers are of least importance.

Because our displayed units are in odds, we can see that 1 standard deviation increase in each of the above features will result in the corresponding percent increase or decrease in the odds of winning.

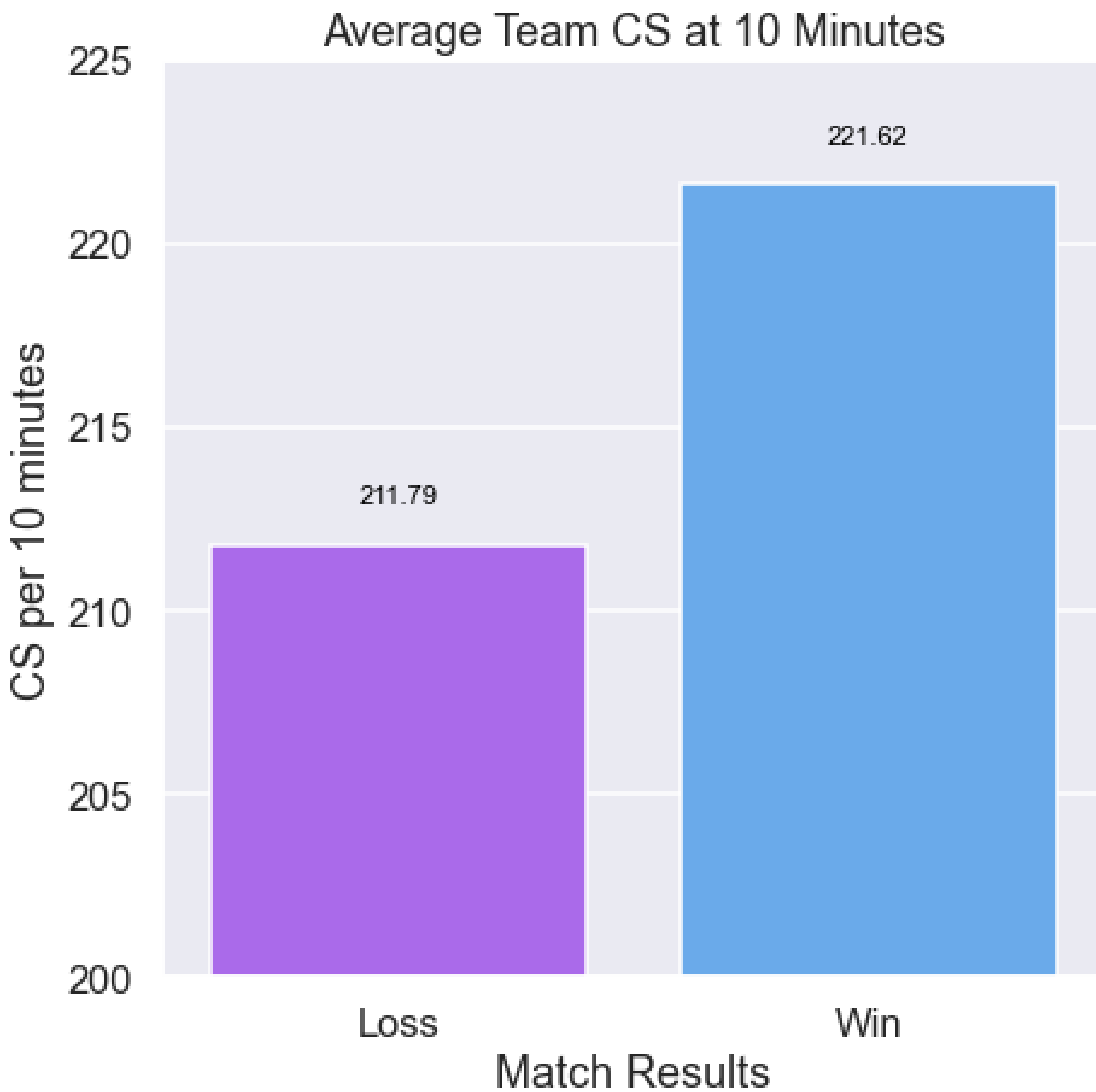
[35]:
```
# Create series that displays the mean total minions killed for matches that
# resulted in losses and wins.
df_viz = df.copy()
df_minions = df_viz.groupby('blueWins').agg('mean')['blueTotalMinionsKilled']
df_minions
```

[35]:
```
blueWins
0    211.793090
1    221.624949
```

```
Name: blueTotalMinionsKilled, dtype: float64
```

[36]:

```
# Create bar plot of mean number of minions killed for losses and wins
fig, ax = plt.subplots(figsize=(7,7))

sns.barplot(x=df_minions.index, y=df_minions.values, palette='cool_r', ax=ax)

ax.set_title('Average Team CS at 10 Minutes')
ax.set_xlabel('Match Results')
ax.set_ylabel('CS per 10 minutes')
ax.set_xticklabels(['Loss','Win'])

# Method for displaying values at the top of bars found at:
# https://stackoverflow.com/questions/45946970/displaying-of-values-on-barchart
x_axis = ax.get_xticklabels()
y_axis = [df_minions.values]

for p in ax.patches:
    ax.annotate("%.2f" % p.get_height(), (p.get_x() + p.get_width() / 2., \
                                          p.get_height()),ha='center', \
                va='center', fontsize=11, color='black', xytext=(0, 20), \
                textcoords='offset points')

ax.set_ylim([200, 225]);
```



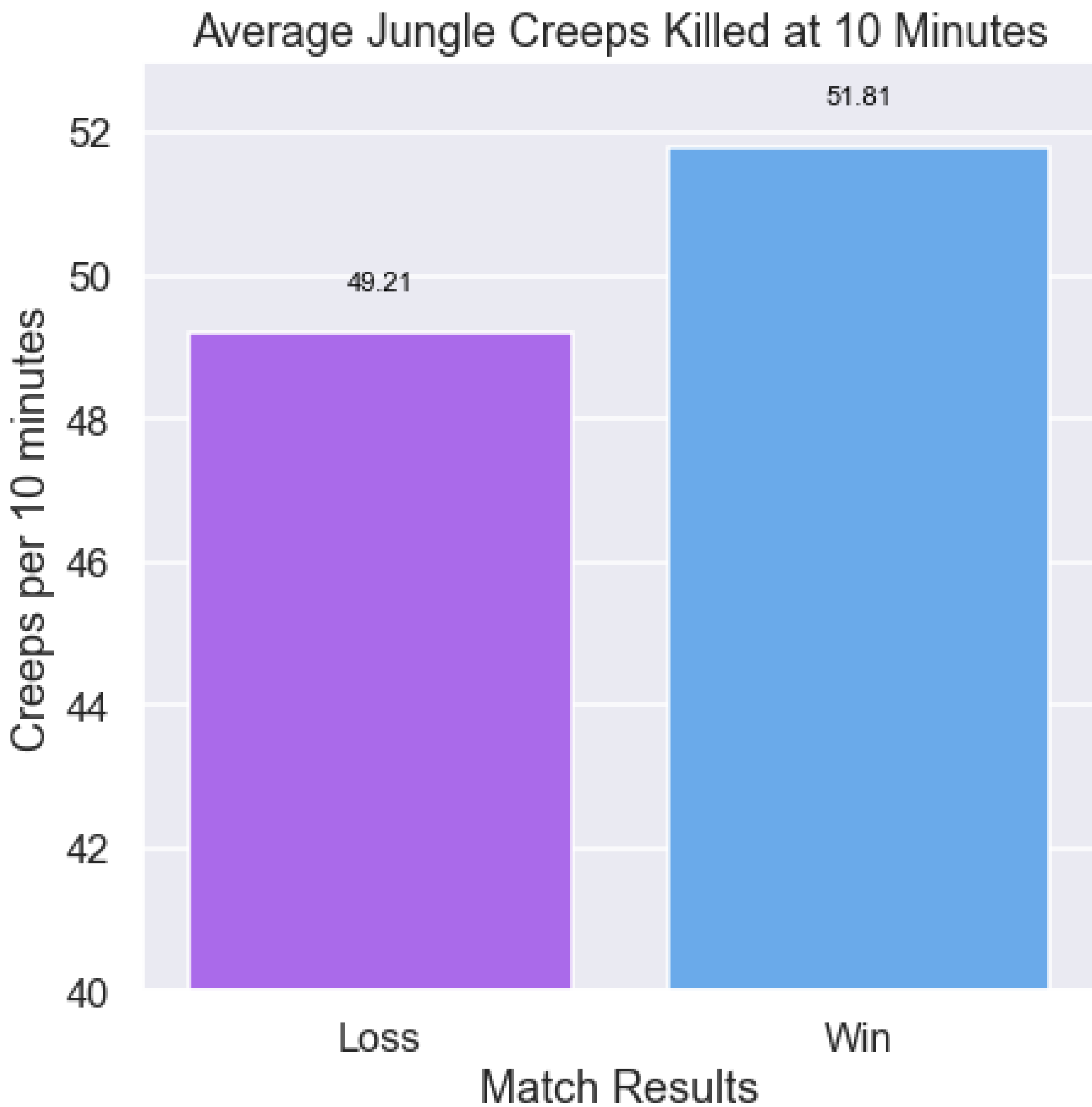
We can see that there is a difference of approximately 10 in the number of total minions killed at the 10 minute mark that would make the difference between a loss and a win. In order to maximize our chances of winning, we want to make sure that the team reaches a total minion kill count of above 222 within 10 minutes of the match start.

[37]:
```
# Create series that displays the mean jungle minions killed for matches that
# resulted in losses and wins.
df_jungle = df_viz.groupby('blueWins')\
                   .agg('mean')['blueTotalJungleMinionsKilled']
df_jungle
```

[37]:
```
blueWins
0    49.211154
```

```
1    51.813185
Name: blueTotalJungleMinionsKilled, dtype: float64
```

[38]:
```
# Create bar plot of mean number of jungle minions killed for losses and wins
fig, ax = plt.subplots(figsize=(7,7))

sns.barplot(x=df_jungle.index, y=df_jungle.values, palette='cool_r', ax=ax)

ax.set_title('Average Jungle Creeps Killed at 10 Minutes')
ax.set_xlabel('Match Results')
ax.set_ylabel('Creeps per 10 minutes')
ax.set_xticklabels(['Loss','Win'])

x_axis = ax.get_xticklabels()
y_axis = [df_jungle.values]

for p in ax.patches:
    ax.annotate("%.2f" % p.get_height(), (p.get_x() + p.get_width() / 2.,
                                          p.get_height()),ha='center', \
                va='center', fontsize=11, color='black', xytext=(0, 20), \
                textcoords='offset points')

ax.set_ylim([40, 53]);
```



Although the difference in the total number of jungle creeps killed between losses and wins is smaller than we saw in the difference in lane minion kills, we want to make sure to have our jungler is able to clear more than 52 jungle creeps in order to maximize the odds of winning.

8 CONCLUSIONS & RECOMMENDATIONS

Based on the above findings, we can see that champion kills and assists, lane minions, jungle minions, and dragons have the highest impact on the outcome of a high ranking League of Legends match.

My primary recommendation would be to focus heavily on the Jungler role. While optimizing an efficient jungle clearing path to maximize the number of jungle creeps killed, we want to make sure to capitalize on any early champion kills that might be possible if the Jungler can execute an effective gank.

My secondary recommendation would be to have all laners heavily drill last hitting minions to maximize the number of minion kills in the early stages of the match. There are a total of 107 minions that spawn per lane within the first 10 minutes of the match, and we want to aim for a team total of 222 minions or more. This means that each laner must kill at least 74 minions, while avoiding death and if possible, securing champion kills.

Lastly, since dragons are also of high importance, the Support role should place vision wards close to the dragon pit in order to maintain map control in that area, while the AD Carry role focuses on securing minions kills within his/her lane.

Some considerations for further analysis would include: 1. Whether we can find additional features outside of the scope of the selected dataset to improve the predictive capability of our models. 2. Analyzing data collected at the end of each match to identify what elements of the game led to a quicker vs. slower victory so that we can adjust the team strategy mid-game to increase the odds of winning. 3. Collect data on the specific eSports team's actual performance to identify what areas need to be targeted.

```
[ ]:
```