Solution - E-Sports Analytics

Solution: E-Sports Analytics

Import Libraries and Dataset

In [1]:
```
import pandas as pd

import matplotlib.pyplot as plt
%matplotlib inline

import seaborn as sns
sns.set_style('darkgrid')
```

In [2]:
```
df = pd.read_csv('data/esports_analytics.csv')
```

Exploratory Analysis w/ Q&A

In [3]:
```
# Example observations
df.head()
```

Out[3]:

	league	year	season	blue_team	blue_win	red_team	blue_top	blue_jungle	blue_mid	blue_adc	blu
0	NALCS	2015	Spring	TSM	1	C9	Irelia	RekSai	Ahri	Jinx	Jar
1	NALCS	2015	Spring	CST	0	DIG	Gnar	Rengar	Ahri	Caitlyn	Le
2	NALCS	2015	Spring	WFX	1	GV	Renekton	Rengar	Fizz	Sivir	An
3	NALCS	2015	Spring	TIP	0	TL	Irelia	JarvanIV	Leblanc	Sivir	Th
4	NALCS	2015	Spring	CLG	1	T8	Gnar	JarvanIV	Lissandra	Tristana	Jar

In [4]:
```
df.shape
```

Out[4]:
```
(5482, 16)
```

1. How many leagues are there, and how many games were played in each league in each year?

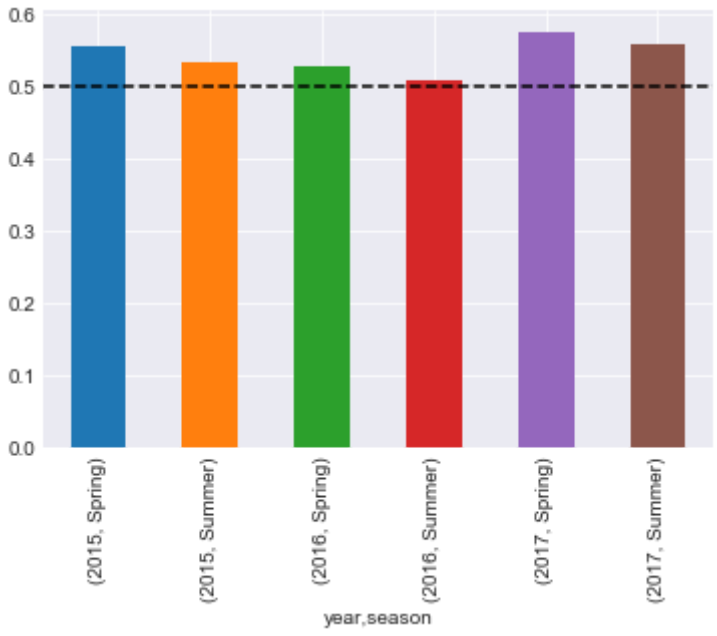
In [5]:
```
df.groupby(['league', 'year']).size()
```

Out[5]:
```
league  year
CBLoL   2016    112
        2017    112
CLS     2017    144
```

```
EULCS    2015    182
         2016    272
         2017    309
LCK      2015    342
         2016    427
         2017    439
LCL      2016    113
         2017    115
LJL      2016    106
         2017    141
LLN      2017    222
LMS      2015    163
         2016    220
         2017    231
NALCS    2015    185
         2016    307
         2017    447
OPL      2016    197
         2017    189
TCL      2015    111
         2016    173
         2017    223
dtype: int64
```

2. For each season, plot the win rate of blue side. Do you think the starting sides are properly balanced? Assume the sides are randomly assigned.

In [6]:
```
df.groupby(['year', 'season']).blue_win.mean().plot(kind='bar')
plt.axhline(y=0.5, color='k', linestyle='--')
plt.show()
```



- In every single season, it appears that blue side has a higher win rate. If we assume the sides are randomly assigned (i.e. it's not a situation where the stronger team gets to choose the side or have "home field advantage"), then there's a good chance the starting sides are not properly balanced.

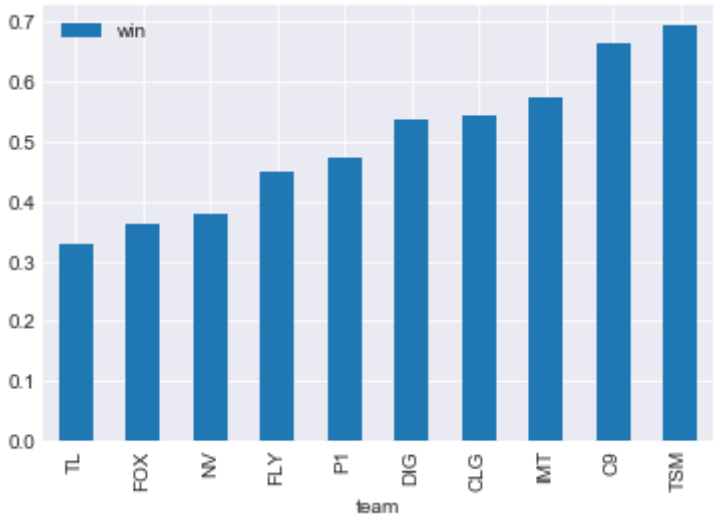
3. Which team in the NALCS league had the highest overall win-rate in 2017?

Remember, each match is played between two teams, `red_team` and `blue_team`, so if `blue_win` is a `0` then we should still add a win for the red side team.

There are several ways to do this, but one easy way is to create an intermediary table by "stacking" the blue wins and blue losses, like so:

In [7]:
```
# Limit to 2017
nalcs_2017 = df[(df.league == 'NALCS') & (df.year == 2017)]

# Stack the 'blue_team' and 'red_team' columns
blue_side_wins = nalcs_2017.loc[:,['blue_team', 'blue_win']]
blue_side_wins.columns = ['team', 'win']

red_side_wins = nalcs_2017.loc[:,['red_team', 'blue_win']]
red_side_wins.columns = ['team', 'win']
red_side_wins.win = 1 - red_side_wins.win # blue loss == red win

# All games
all_games = pd.concat([blue_side_wins, red_side_wins])

# Plot win rates
all_games.groupby('team').mean().sort_values(by='win').plot(kind='bar')
```

Out[7]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a19a0bf60>
```



- **TSM** had the highest win-rate during 2017 in the NALCS.

4. Within the leagues that have existed since 2015, has the number of unique champions played in each position increased through the years? Which position saw the largest increase in champion diversity from 2015 to 2017?

In [8]:
```
old_leagues = df[df.league.isin(['EULCS', 'LCK',
                                 'LMS', 'NALCS', 'TCL'])]

# Stack the 'blue_support' and 'red_support' columns
blue = old_leagues.loc[:,['year', 'blue_top', 'blue_jungle',
                          'blue_mid', 'blue_adc', 'blue_support']]
blue.columns = ['year', 'top', 'jungle', 'mid', 'adc', 'support']

red = old_leagues.loc[:,['year', 'red_top', 'red_jungle',
                         'red_mid', 'red_adc', 'red_support']]
red.columns = ['year', 'top', 'jungle', 'mid', 'adc', 'support']

all_champions_played = pd.concat([blue, red])

# Calculate unique champions played by year
all_champions_played.groupby('year').nunique()
```

Out[8]:

	year	top	jungle	mid	adc	support
year						
2015	1	47	33	51	18	31
2016	1	48	29	53	20	31
2017	1	57	47	55	30	49

- In terms of absolute increase in champion diversity, the **support** role saw the greatest increase (+18).
- In terms of percentage increase in champion diversity, the **adc** role saw the greatest increase (+67%).

5. Prior to the 2017 Summer season, which league had the highest champion diversity (number of unique champions played) in the blue-side support role?

In [9]:
```
# Matches to exclude
mask = (df.year == 2017) & (df.season == 'Summer')
pre_2017_summer = df[~mask]
```

Out[9]:
```
pro_2017_summer.groupby('league').blue_support.nunique()
```

```
League
CBLoL    18
CLS      14
EULCS    28
LCK      35
LCL      18
LJL      21
LLN      16
LMS      24
NALCS    26
OPL      25
TCL      33
Name: blue_support, dtype: int64
```

The **TCL** had the highest champaion diversity.

Machine Learning

In [10]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import roc_auc_score
```

For this objective, since we wish to try various training sets of data, we will start by writing a convenient helper function for training a Random Forest classifier.

In [11]:
```
def train_random_forest(train, test):
    # Extract target variable for training set
    y_train = train.blue_win
    X_train = train.drop('blue_win', axis=1)

    # Extract target variable for test set
    y_test = test.blue_win
    X_test = test.drop('blue_win', axis=1)

    # Train Random Forest
    rf = RandomForestClassifier(random_state=1234)
    rf.fit(X_train, y_train)

    # Predict test set
    pred = rf.predict_proba(X_test)
    pred = [p[1] for p in pred]

    # Print AUROC
    auroc = roc_auc_score( y_test, pred )
    print( 'AUROC:', auroc )
```

```
    return rf, auroc
```

For convenience, we will also replace "Spring" and "Summer" with 0 and 1 in the `season` feature.

In [12]:
```
df['season'] = df.season.apply( lambda x: {'Spring': 0, 'Summer':1}[x] )
```

Next, for each of the scenarios we'd like to try, we will create an ABT (analytical base table) specifically for that scenario.

Why wouldn't we create a single massive ABT first, and then use the `df` object to filter it later? The reason is simple: we want to minimize the amount of unecessary noise we introduce to each scenario.

For example, scenario 3 is using a much broader (global) set of matches than scenario 1 (which uses only NALCS matches). As a result, we would expect scenario 3 to have an ABT with more features (i.e. more dummy features for each of the champion positions, the teams, etc.).

And since we have a limited amount of data, we should minimize the number of unecessary features that go into training a model under each scenario. (You can compare the two approaches for yourself.)

Tip: Be very careful with the order of filtering observations, and be careful when defining your test sets and training sets.

1. Training set: NALCS 2017 Spring season matches with all available information.

In [13]:
```
# NALCS 2017 matches
mask1 = (df.league=='NALCS') & (df.year==2017)

# Drop league and year after filtering
abt1 = pd.get_dummies( df[mask1].drop(['league', 'year'], axis=1) )

# Test set (Summer season)
test1 = abt1[abt1.season == 1].drop('season', axis=1)

# Training set (Spring season)
train1 = abt1[abt1.season == 0].drop('season', axis=1)
```

In [14]:
```
rf1, auroc1 = train_random_forest(train1, test1)
```

```
AUROC: 0.5902777777777778
```

2. Training set: NALCS 2017 Spring season games, but ignoring which teams are playing.

In [15]:
```
# NALCS 2017 matches
mask2 = (df.league=='NALCS') & (df.year==2017)

# Ignore team information as well
abt2 = pd.get_dummies( df[mask2].drop(['league', 'year',
                                       'blue_team', 'red_team'], axis=1) )

# Test set (Summer season)
test2 = abt2[abt2.season == 1].drop('season', axis=1)

# Training set (Spring season)
train2 = abt2[abt2.season == 0].drop('season', axis=1)
```

In [16]:
```
rf2, auroc2 = train_random_forest(train2, test2)
```

```
AUROC: 0.5228174603174603
```

3. Training set: Global matches played in 2017 Spring season, ignoring teams.

In [17]:
```
# First, filter to 2017 matches
df_2017 = df[df.year==2017]

# Create ABT, dropping team-related info
abt3 = pd.get_dummies( df_2017.drop(['blue_team','red_team',
                                     'year', 'season', 'league'], axis=1) )

# Then create a test/train masks based on df_2017
test_mask3 = (df_2017.league=='NALCS') & (df_2017.season==1)
train_mask3 = (df_2017.season==0)

# Test set
test3 = abt3[test_mask3]

# Train set
train3 = abt3[train_mask3]
```

In [18]:
```
rf3, auroc3 = train_random_forest(train3, test3)
```

```
AUROC: 0.5297222222222222
```

4. Training set: Global games played before 2017 Summer season, ignoring teams.

In [19]:
```
# Test set mask
test_mask4 = (df.league=='NALCS') & (df.year==2017) & (df.season==1)

# Train set mask: all games before 2017 Summer
train_mask4 = ~((df.year==2017) & (df.season==1))
```

```
# Create ABT, dropping team related info
abt4 = pd.get_dummies( df.drop(['blue_team','red_team',
                                'year', 'season', 'league'], axis=1))

# Test set
test4 = abt4[test_mask4]

# Train set
train4 = abt4[train_mask4]
```

In [20]:
```
rf4, auroc4 = train_random_forest(train4, test4)
```

```
AUROC: 0.4925
```

5. Training set: NACLS 2017 Spring season games using only information on teams and side (ignoring champions).

In [21]:
```
# NALCS 2017 matches
mask5 = (df.league=='NALCS') & (df.year==2017)

# Create ABT using only team and sides info
abt5 = pd.get_dummies( df.loc[mask5, ['blue_win', 'blue_team',
                                      'red_team', 'season']] )

# Train and test sets
test5 = abt5[abt5.season == 1].drop('season', axis=1)
train5 = abt5[abt5.season == 0].drop('season', axis=1)
```

In [22]:
```
rf5, auroc5 = train_random_forest(train5, test5)
```

```
AUROC: 0.5580952380952381
```

Model Performance Analysis

The best performing model was the one trained on only NALCS matches from 2017 Spring, including all available information.

In [23]:
```
print('1. NALCS 2017 Spring:\n',
      train1.shape[0], 'observations\n',
      train1.shape[1], 'features\n',
      round(auroc1, 2), 'AUROC\n')

print('2. NALCS 2017 Spring, no teams:\n',
      train2.shape[0], 'observations\n',
      train2.shape[1], 'features\n',
      round(auroc2, 2), 'AUROC\n')
```

```
print('3. Global 2017 Spring, no teams:\n',
      train3.shape[0], 'observations\n',
      train3.shape[1], 'features\n',
      round(auroc3, 2), 'AUROC\n')

print('4. Global Pre-2017 Summer, no teams:\n',
      train4.shape[0], 'observations\n',
      train4.shape[1], 'features\n',
      round(auroc4, 2), 'AUROC\n')

print('5. NALCS 2017 Spring, only teams:\n',
      train5.shape[0], 'observations\n',
      train5.shape[1], 'features\n',
      round(auroc5, 2), 'AUROC\n')
```

```
1. NALCS 2017 Spring:
 221 observations
 256 features
 0.59 AUROC

2. NALCS 2017 Spring, no teams:
 221 observations
 236 features
 0.52 AUROC

3. Global 2017 Spring, no teams:
 1253 observations
 427 features
 0.53 AUROC

4. Global Pre-2017 Summer, no teams:
 4163 observations
 563 features
 0.49 AUROC

5. NALCS 2017 Spring, only teams:
 221 observations
 21 features
 0.56 AUROC
```

We would still want to confirm these findings (perhaps by performing a similar analysis with each of the other leagues), but the key takeaways are as follows:

- Taking away team information (in scenario 2) hurt model performance.

- Adding more matches (scenarios 3 and 4) could not make up for the loss in performance from losing team information. The idea behind scenarios 3 and 4 was to isolate the effects of champions and see if we could effectively predict matchs based on team compositions (instead of the team players). Instead, this ended up adding too much noise.
- Using only team information from NALCS 2017 Spring (in scenario 5) saw the second-highest performance, implying that champion data was still helpful, but only when combined with knowing the teams playing them.

Ultimately, from the scenarios we tried, the first one produced the best results.

We can also look at feature importances from the first model, and we'll see that a combination of team information *and* champion information were most important:

In [24]:
```
# Helper function for plotting feature importances
def plot_feature_importances(columns, feature_importances, show_top_n=10)
    feats = dict( zip(columns, feature_importances) )
    imp = pd.DataFrame.from_dict(feats, orient='index').rename(columns={(
    imp.sort_values(by='Gini-importance').tail(show_top_n).plot(kind='bar
    plt.show()
```

In [25]:
```
plot_feature_importances(train1.columns, rf1.feature_importances_)
```