Evil Geniuses - DS Intern Assessment

Name: Runqing (Roch) Jia

Creation Date: May 24, 2023

Last Modified Date: May 28, 2023

In [ ]:
```
# Import essenstial libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.linear_model import LinearRegression, Lasso, LassoCV
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
from sklearn.ensemble import RandomForestRegressor
import xgboost as xgb
from sklearn.neighbors import KNeighborsRegressor
import statsmodels.api as sm
from statsmodels.miscmodels.ordinal_model import OrderedModel
```

In [ ]:
```
# Import the data
df = pd.read_csv('data/starcraft_player_data.csv')
```

1. EDA & Data Cleansing

Our initial undertaking involves understanding the data, a task which can be divided into two sections:

1) Extracting profile information of the data: This step focuses on understanding the users our data represents.

2) Exploring the data from a management perspective: Our aim here is to identify any potential issues with the data and resolve them through data cleansing. This will ensure the data is primed for the modeling process.

1.1 Overall Exploration

The dataset comprises ~3400 rows of data, with each row representing the information of one unique user (identified by **GameID**). The data incorporates features about users' personal profiles, degree of engagement, and gaming behaviors. The objective is to predict the rank of the user, stored in the **'LeagueIndex'** column - **'LeagueIndex'** is our target variable.

The data appears to be clean, with no obvious missing values. Nevertheless, we discovered that the **'Age'**, **'HoursPerWeek'**, and **'TotalHours'** columns have the data type 'object', which suggests that there might be some records with a data type other than integer, which warrants further investigation in our subsequent analysis.

We can gain a quick snapshot of our dataset, its structure, any missing values, and the data types of our columns with the following code:

In [ ]:
```
# Overview of the dataset
df.head()
```

Out[ ]:

	GameID	LeagueIndex	Age	HoursPerWeek	TotalHours	APM	SelectByHotkeys	AssignToH
0	52	5	27	10	3000	143.7180	0.003515	0.(
1	55	5	23	10	5000	129.2322	0.003304	0.(
2	56	4	30	10	200	69.9612	0.001101	0.(
3	57	3	19	20	400	107.6016	0.001034	0.(
4	58	3	32	10	500	122.8908	0.001136	0.(

In [ ]:
```
# Data Structure
df.shape
```

Out[ ]:
```
(3395, 20)
```

In [ ]:
```
# Check missing value
df.isna().sum()
```

Out[ ]:
```
GameID                0
LeagueIndex           0
Age                   0
HoursPerWeek          0
TotalHours            0
APM                   0
SelectByHotkeys       0
AssignToHotkeys       0
UniqueHotkeys         0
MinimapAttacks        0
MinimapRightClicks    0
NumberOfPACs          0
GapBetweenPACs        0
ActionLatency         0
ActionsInPAC          0
TotalMapExplored      0
WorkersMade           0
UniqueUnitsMade       0
ComplexUnitsMade      0
ComplexAbilitiesUsed  0
dtype: int64
```

In [ ]:
```
# Check data types
df.dtypes
```

Out[ ]:
```
GameID                  int64
LeagueIndex             int64
Age                    object
HoursPerWeek           object
TotalHours             object
APM                   float64
SelectByHotkeys       float64
AssignToHotkeys       float64
UniqueHotkeys           int64
MinimapAttacks        float64
MinimapRightClicks    float64
NumberOfPACs          float64
GapBetweenPACs        float64
ActionLatency         float64
ActionsInPAC          float64
TotalMapExplored        int64
WorkersMade           float64
UniqueUnitsMade         int64
ComplexUnitsMade      float64
ComplexAbilitiesUsed  float64
dtype: object
```

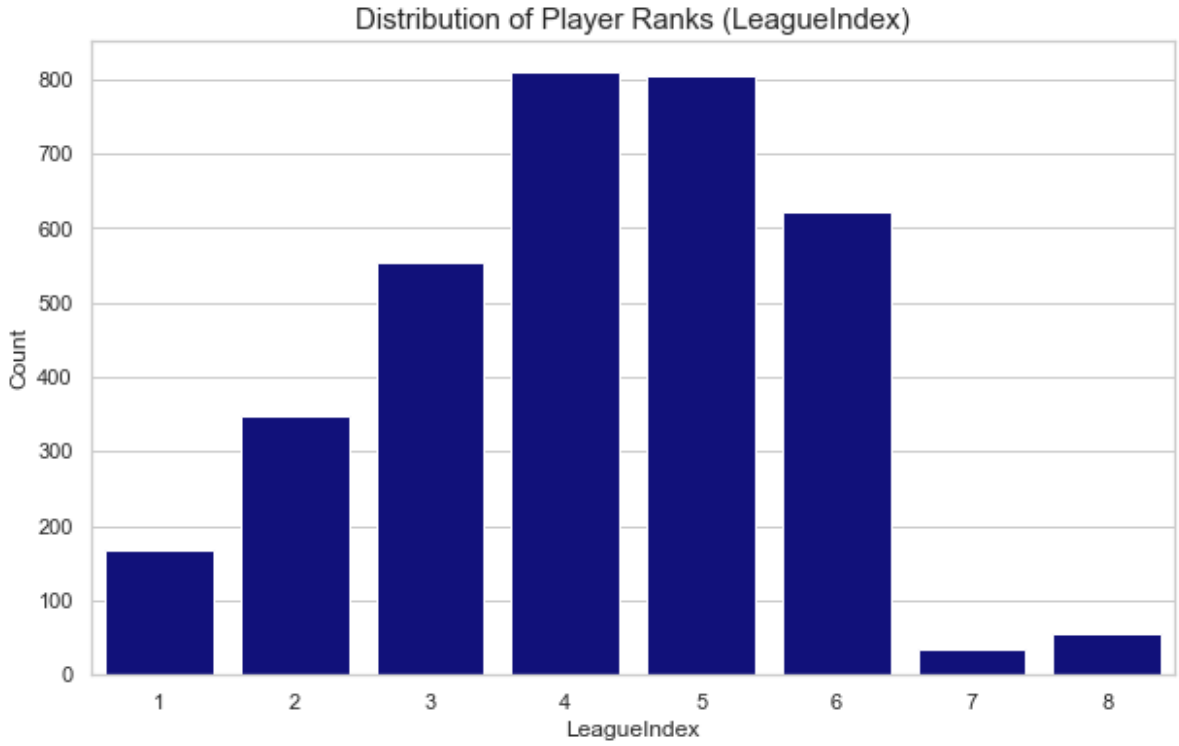
1.2 Exploration & Visualization by Column

Our focus in this segment is on the users' profile, with the aim of gaining a better understanding of the gamer group. We intend to utilize data visualization tools to facilitate a deeper understanding of the distribution of our target columns.

1.2.1 Target Variable: LeagueIndex

Our first objective is to understand the distribution of rankings. We find that the distribution is almost normally distributed but demonstrates slight **left skewness**. Generally, most players fall within the Platinum and Diamond categories, with relatively fewer players in the GrandMaster and Professional tiers.

In [ ]:
```
# Viz: Distribution of League Rank
sns.set_theme(style='whitegrid')
fig, ax = plt.subplots(figsize=(10, 6))
sns.countplot(data=df, x='LeagueIndex', color='darkblue', ax=ax)
ax.set_title('Distribution of Player Ranks (LeagueIndex)', fontsize=15)
ax.set_xlabel('LeagueIndex', fontsize=12)
ax.set_ylabel('Count', fontsize=12)

plt.show()
```



1.2.2 Age

Age is one of the key personal characteristics within the dataset. An initial review of unique values reveals a challenge: beyond the expected range of integer values, the value **"?"** is also present in this column, which represents an unknown record - a missing value. There are 55 instances of "?" in the "Age" column.

We address this issue by converting "?" to NaN and standardizing the data to a numeric format. For visualization of age distribution, we categorize ages into four groups: Under 18, 18-30, Over 30, and Unknown. The resulting distribution reveals that the majority of gamers in this dataset are in their young adulthood, with a relatively small proportion of middle-aged users.

In [ ]:
```
# Check the unique values
df['Age'].unique()
```

Out[ ]:
```
array(['27', '23', '30', '19', '32', '21', '17', '20', '18', '16', '26',
       '38', '28', '25', '22', '29', '24', '35', '31', '33', '37', '40',
       '34', '43', '41', '36', '44', '39', '?'], dtype=object)
```

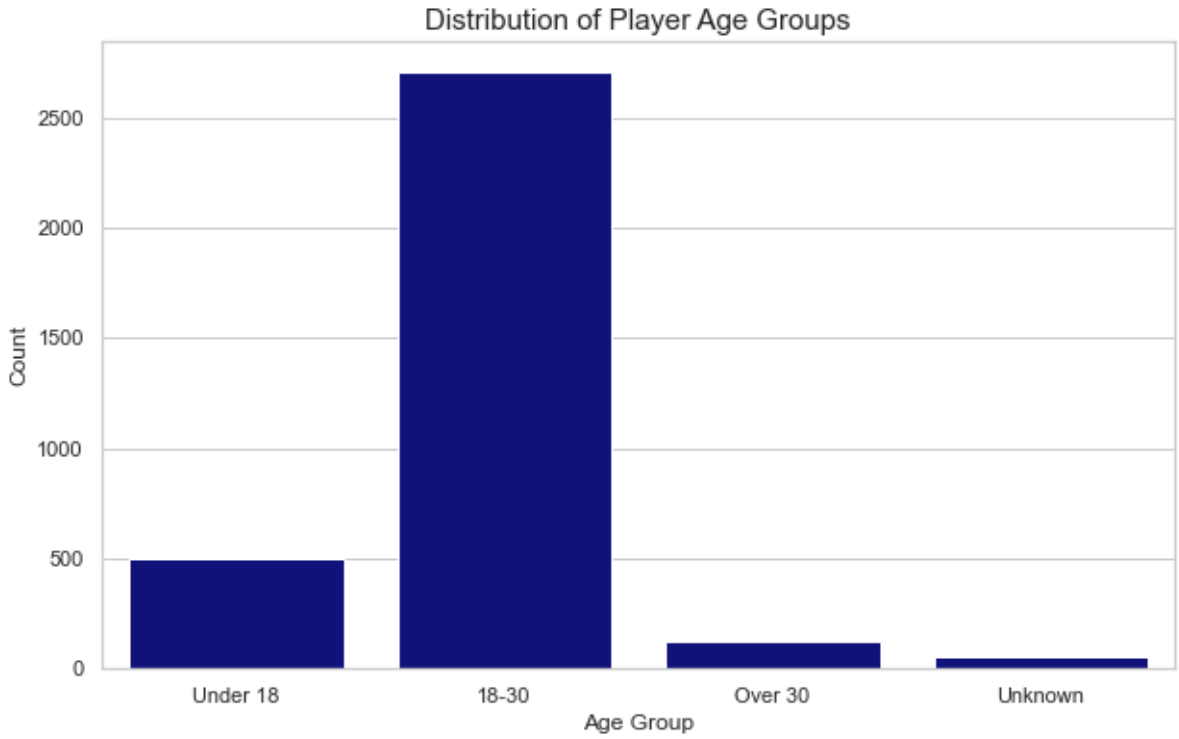
In [ ]:
```
# Check the number of records with "?"
df['Age'].value_counts()['?']
```

Out[ ]:
```
55
```

In [ ]:
```
# Datatype conversion
# Convert the '?' to a NaN value for easier manipulation
df['Age'] = df['Age'].replace('?', np.nan)
# Convert the column to numeric
df['Age'] = pd.to_numeric(df['Age'])
```

In [ ]:
```
# Viz: Distribution of Age
# Define a function to categorize ages
def categorize_age(age):
```

```
    if age < 18:
        return 'Under 18'
    elif 18 <= age <= 30:
        return '18-30'
    elif age > 30:
        return 'Over 30'
    else:
        return 'Unknown'

# Apply the function to the Age column
df['AgeGroup'] = df['Age'].apply(categorize_age)

# Viz
plt.figure(figsize=(10, 6))
sns.countplot(data=df, x='AgeGroup', order=['Under 18', '18-30', 'Over 30', 'Unknow
plt.title('Distribution of Player Age Groups', fontsize=15)
plt.xlabel('Age Group', fontsize=12)
plt.ylabel('Count', fontsize=12)
plt.show()
```



1.2.3 HoursPerWeek

Next, we examine the **HoursPerWeek** column, where we encounter the same issue of "?" values. We apply the same strategy to handle this, converting "?" to NaN and adjusting the data type. This column contains 56 instances of missing data.

We also visualize the distribution of HoursPerWeek by creating categories to represent different player engagement levels: Under 10 (Casual Gamer), 10-20 (Medium Gamer), 20-50 (Addictive Gamer), Over 50 (Super Addictive Gamer), and Unknown. The visualization shows that the majority of players are casual or medium gamers, which aligns with general expectations.

In [ ]:
```
df['HoursPerWeek'].unique()
```

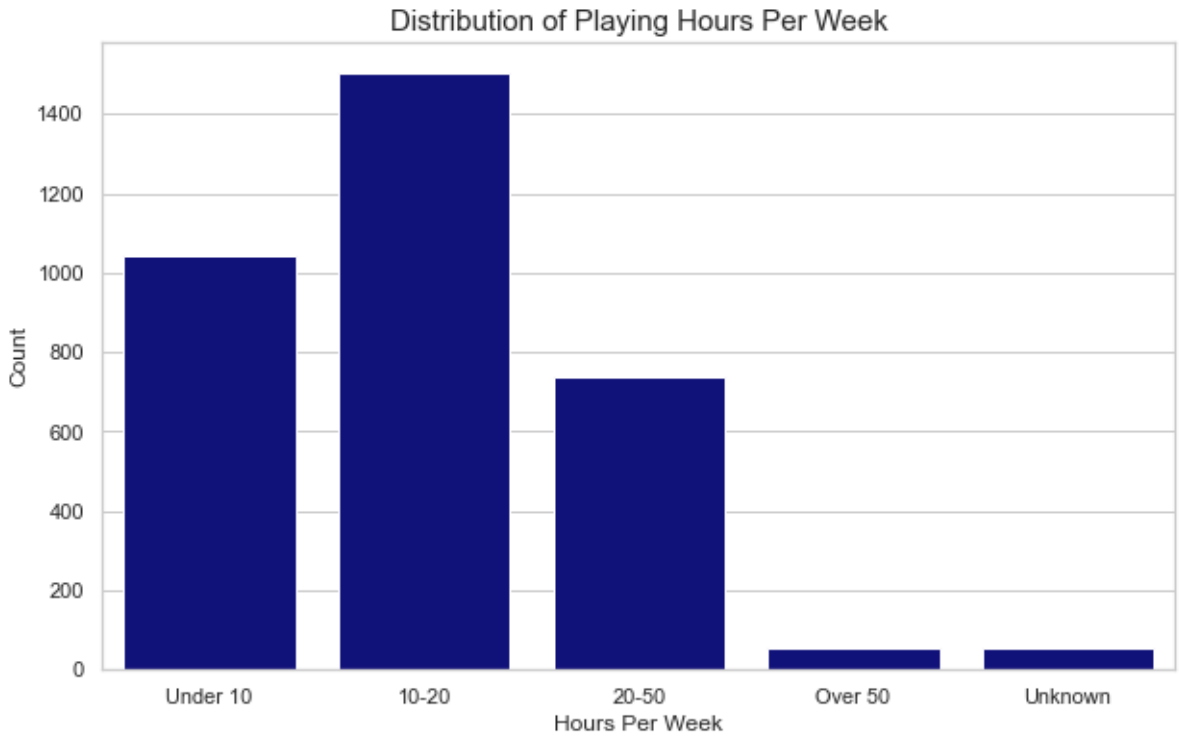
Out[ ]:
```
array(['10', '20', '6', '8', '42', '14', '24', '16', '4', '12', '30',
       '28', '70', '2', '56', '36', '40', '18', '96', '50', '168', '48',
       '84', '0', '72', '112', '90', '32', '98', '140', '?', '80', '60'],
      dtype=object)
```

In [ ]:
```
df['HoursPerWeek'].value_counts()['?']
```

Out[ ]:
```
56
```

In [ ]:
```
df['HoursPerWeek'] = df['HoursPerWeek'].replace('?', np.nan)
df['HoursPerWeek'] = pd.to_numeric(df['HoursPerWeek'])
```

In [ ]:
```
# Viz: Distribution of HoursPerWeek
# Define a function to categorize HPW
def categorize_HPW(HoursPerWeek):
    if HoursPerWeek < 10:
        return 'Under 10'
    elif 10 <= HoursPerWeek <= 20:
        return '10-20'
    elif 20 < HoursPerWeek <= 50:
        return '20-50'
    elif HoursPerWeek > 50:
        return 'Over 50'
    else:
        return 'Unknown'

# Apply the function to the Age column
df['HPWGroup'] = df['HoursPerWeek'].apply(categorize_HPW)

# Viz
plt.figure(figsize=(10, 6))
sns.countplot(data=df, x='HPWGroup', order=['Under 10', '10-20', '20-50', 'Over 50
plt.title('Distribution of Playing Hours Per Week', fontsize=15)
plt.xlabel('Hours Per Week', fontsize=12)
plt.ylabel('Count', fontsize=12)
plt.show()
```



1.2.4 TotalHours

We again encounter the "?" value problem in the **TotalHours** column, and address it using the same data cleaning approach.

For visualizing TotalHours, we use different intervals to categorize player engagement: Under 200, 200-1000, 1000-5000, Over 5000, and Unknown. The resulting distribution shows most players have logged between 200 and 1000 hours, a reasonable range for casual and medium gamers. A very small proportion of players are heavily invested in the game, with over 5000 hours logged.

In [ ]:
```
df['TotalHours'].unique()
```

Out[ ]:
```
array(['3000', '5000', '200', '400', '500', '70', '240', '10000', '2708',
       '800', '6000', '190', '350', '1000', '1500', '2000', '120', '1100',
       '2520', '700', '160', '150', '250', '730', '230', '300', '100',
       '270', '1200', '30', '600', '540', '280', '1600', '50', '140',
       '900', '550', '625', '1300', '450', '750', '612', '180', '770',
       '720', '415', '1800', '2200', '480', '430', '639', '360', '1250',
       '365', '650', '233', '416', '1825', '780', '1260', '315', '10',
       '312', '110', '1700', '92', '2500', '1400', '220', '999', '303',
       '96', '184', '4000', '420', '60', '2400', '2160', '80', '25',
       '624', '176', '?', '35', '1163', '333', '75', '7', '40', '325',
       '90', '175', '88', '850', '26', '1650', '465', '235', '1350',
       '460', '848', '256', '130', '1466', '670', '711', '1030', '1080',
       '1460', '1050', '20000', '582', '2800', '553', '1008', '330',
       '936', '243', '1320', '425', '1145', '366', '2700', '830', '3',
       '125', '2300', '336', '24', '12', '72', '690', '320', '144', '20',
       '1155', '520', '865', '275', '548', '170', '898', '1170', '1148',
       '105', '575', '1850', '238', '820', '310', '85', '2942', '94',
       '2100', '224', '165', '577', '1440', '731', '727', '138', '45',
       '225', '95', '630', '1274', '1782', '610', '525', '2671', '2016',
       '123', '1095', '1000000', '2920', '640', '1344', '1940', '16',
       '410', '960', '740', '950', '551', '216', '840', '18000', '745',
       '530', '477', '1270', '36', '174', '2600', '1256', '9000', '1880',
       '288', '1150', '10260', '2190', '560', '25000', '128', '666',
       '854', '370', '65', '334', '755', '1024', '3257', '208', '1196',
       '1870', '990', '470', '699', '340', '2250', '255', '980', '620',
       '380', '196', '21', '153', '1098', '546', '433', '1560', '580',
       '77', '148', '2880', '364', '56'], dtype=object)
```

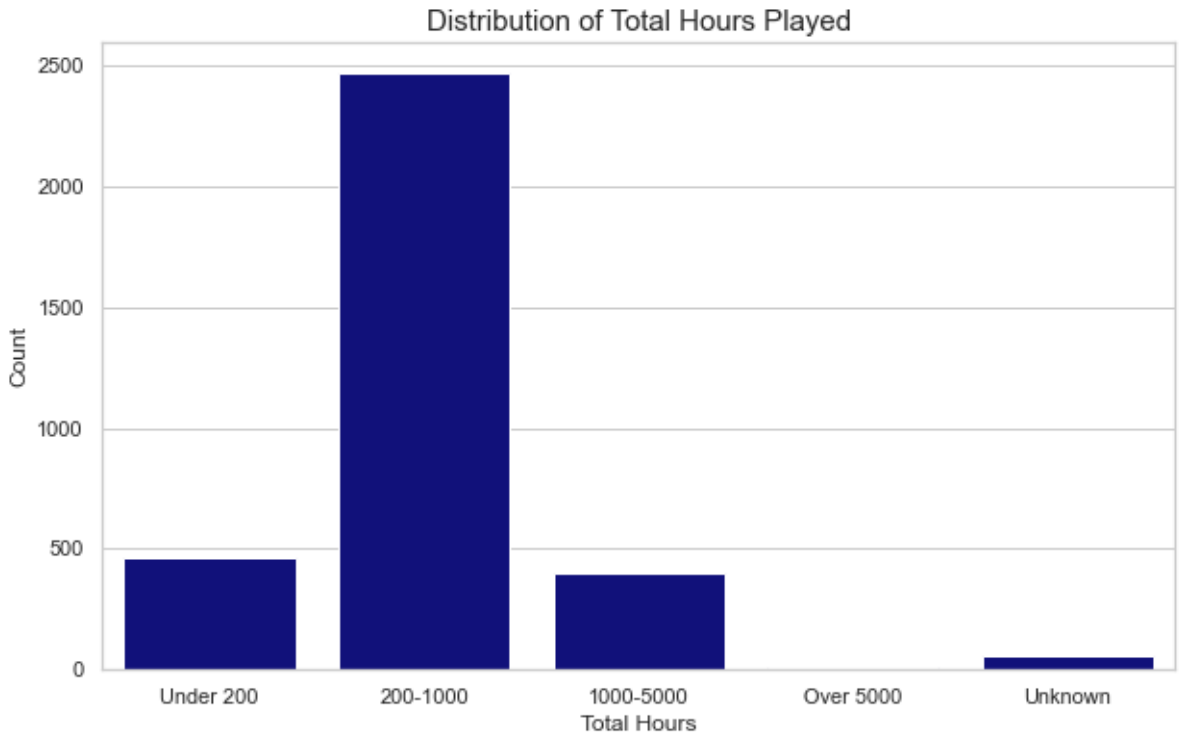
In [ ]:
```
df['TotalHours'].value_counts()['?']
```

Out[ ]:
```
57
```

In [ ]:
```
df['TotalHours'] = df['TotalHours'].replace('?', np.nan)
df['TotalHours'] = pd.to_numeric(df['TotalHours'])
```

In [ ]:
```
# Viz: Distribution of TotalHours
# Define a function to categorize TotalHours
def categorize_TH(TotalHours):
    if TotalHours < 200:
        return 'Under 200'
    elif 200 <= TotalHours <= 1000:
        return '200-1000'
    elif 1000 < TotalHours <= 5000:
        return '1000-5000'
    elif TotalHours > 5000:
```

```
        return 'Over 5000'
    else:
        return 'Unknown'

# Apply the function to the Age column
df['THGroup'] = df['TotalHours'].apply(categorize_TH)

# Viz
plt.figure(figsize=(10, 6))
sns.countplot(data=df, x='THGroup', order=['Under 200', '200-1000', '1000-5000', '
plt.title('Distribution of Total Hours Played', fontsize=15)
plt.xlabel('Total Hours', fontsize=12)
plt.ylabel('Count', fontsize=12)
plt.show()
```



1.3 Data Cleansing

1.3.1 Drop Columns

Dropping irrelevant columns is a common step in data preprocessing for ML applications. Intermediate features generated for EDA are also dropped, which are not useful for model training.

In [ ]:
```
# Drop the intermediate columns for viz
df = df.drop(['AgeGroup', 'HPWGroup', 'THGroup'], axis = 1)
```

1.3.2 Manipulation on Missing Values

The next step is handling missing values. It appears that there are only 57 records with missing values across the 'Age', 'HoursPerWeek', and 'TotalHours' columns. Interestingly, 55 of these records have all three fields missing, and all the 55 records belong to Professional (Rank 8).

Given that there are exactly 55 players at Professional rank, it was decided not to remove these records, since removing could potentially lead to a bias in the data. Instead, a model-

based imputation approach was used. This involves using other features to predict and fill in the missing values with linear regression in this case.

In [ ]:
```
# Have an overview of the records with missing values
df[(df['Age'].isnull()) | (df['HoursPerWeek'].isnull()) | (df['TotalHours'].isnull(
```

Out[ ]:

	GameID	LeagueIndex	Age	HoursPerWeek	TotalHours	APM	SelectByHotkeys	Assign
358	1064	5	17.0	20.0	NaN	94.4724	0.003846	
1841	5255	5	18.0	NaN	NaN	122.2470	0.006357	
3340	10001	8	NaN	NaN	NaN	189.7404	0.004582	
3341	10005	8	NaN	NaN	NaN	287.8128	0.029040	
3342	10006	8	NaN	NaN	NaN	294.0996	0.029640	
3343	10015	8	NaN	NaN	NaN	274.2552	0.018121	
3344	10016	8	NaN	NaN	NaN	274.3404	0.023131	
3345	10017	8	NaN	NaN	NaN	245.8188	0.010471	
3346	10018	8	NaN	NaN	NaN	211.0722	0.013049	
3347	10021	8	NaN	NaN	NaN	189.5778	0.007559	
3348	10022	8	NaN	NaN	NaN	210.5088	0.007974	
3349	10023	8	NaN	NaN	NaN	248.0118	0.014722	
3350	10024	8	NaN	NaN	NaN	299.2290	0.026428	
3351	10025	8	NaN	NaN	NaN	179.9982	0.009524	
3352	10026	8	NaN	NaN	NaN	340.1982	0.028214	
3353	10028	8	NaN	NaN	NaN	319.7148	0.037130	
3354	10029	8	NaN	NaN	NaN	290.5914	0.027561	
3355	10030	8	NaN	NaN	NaN	275.8632	0.019502	
3356	10035	8	NaN	NaN	NaN	298.7916	0.023253	
3357	10036	8	NaN	NaN	NaN	325.1154	0.029790	
3358	10038	8	NaN	NaN	NaN	146.3892	0.006701	
3359	10039	8	NaN	NaN	NaN	192.4554	0.014277	
3360	10041	8	NaN	NaN	NaN	315.6936	0.028311	
3361	10045	8	NaN	NaN	NaN	203.7726	0.008337	
3362	10046	8	NaN	NaN	NaN	334.5240	0.017742	
3363	10047	8	NaN	NaN	NaN	175.5936	0.012680	
3364	10049	8	NaN	NaN	NaN	252.7206	0.019097	
3365	10050	8	NaN	NaN	NaN	211.9188	0.019817	
3366	10051	8	NaN	NaN	NaN	269.8998	0.024645	
3367	10052	8	NaN	NaN	NaN	190.2396	0.008720	
3368	10055	8	NaN	NaN	NaN	212.4972	0.014917	
3369	10059	8	NaN	NaN	NaN	219.3894	0.005926	
3370	10060	8	NaN	NaN	NaN	230.6694	0.010383	
3371	10061	8	NaN	NaN	NaN	284.2296	0.016069	
3372	10062	8	NaN	NaN	NaN	355.3518	0.037526	
3373	10063	8	NaN	NaN	NaN	364.8504	0.042576	

	GameID	LeagueIndex	Age	HoursPerWeek	TotalHours	APM	SelectByHotkeys	Assign
3374	10064	8	NaN	NaN	NaN	256.5888	0.019592	
3375	10065	8	NaN	NaN	NaN	248.4012	0.016018	
3376	10066	8	NaN	NaN	NaN	251.2284	0.022910	
3377	10067	8	NaN	NaN	NaN	318.3000	0.034851	
3378	10068	8	NaN	NaN	NaN	288.9198	0.029322	
3379	10069	8	NaN	NaN	NaN	313.9080	0.019537	
3380	10072	8	NaN	NaN	NaN	243.7134	0.017195	
3381	10073	8	NaN	NaN	NaN	312.9804	0.026327	
3382	10074	8	NaN	NaN	NaN	313.5762	0.030550	
3383	10075	8	NaN	NaN	NaN	274.6194	0.022497	
3384	10076	8	NaN	NaN	NaN	225.0678	0.014339	
3385	10079	8	NaN	NaN	NaN	254.2188	0.016608	
3386	10081	8	NaN	NaN	NaN	339.1524	0.033058	
3387	10082	8	NaN	NaN	NaN	310.0416	0.026873	
3388	10083	8	NaN	NaN	NaN	288.7608	0.024022	
3389	10084	8	NaN	NaN	NaN	151.4046	0.009732	
3390	10089	8	NaN	NaN	NaN	259.6296	0.020425	
3391	10090	8	NaN	NaN	NaN	314.6700	0.028043	
3392	10092	8	NaN	NaN	NaN	299.4282	0.028341	
3393	10094	8	NaN	NaN	NaN	375.8664	0.036436	

In [ ]:
```
# Check the count of Rank 8 (Professional) - Turns out that all the players in Rank
df[df['LeagueIndex'] == 8]['LeagueIndex'].count()
```

Out[ ]: 55

In [ ]:
```
# Missing value manipulation:
# Create a function to apply model-based imputation
def model_based_imputation(df, col, features):
    # Split the missing part and known part
    missing_data = df[df[col].isnull()]
    known_data = df[df[col].notnull()]
    # Apply the linear model & Predict
    model = LinearRegression()
    model.fit(known_data[features], known_data[col])
    pred = model.predict(missing_data[features])
    # Filter the missing values and replace them with the predicted values
    df.loc[df[col].isnull(), col] = pred

    return df
```

In [ ]:
```
cols = ['Age', 'HoursPerWeek', 'TotalHours']
cols_ex = ['GameID', 'LeagueIndex']
all_cols = [col for col in df.columns.to_list() if col not in cols_ex]
```

```
features = [col for col in all_cols if col not in cols]

for col in cols:
    df = model_based_imputation(df, col, features)
    # Convert the data type into integer
    df[col] = df[col].astype(int)
```

1.3.3 Exclude Outliers

Following the imputation, our dataset is clean, but outliers remain a concern. We noticed several issues through a statistical summary:

1) Negative values (predicted) appear in TotalHours. This is illogical, as a time-related attribute cannot have a negative value.

2) Outliers occur in HoursPerWeek and TotalHours. It's impractical for a player to engage in a game for 168 hours (24 * 7) a week without sleep, and a value of 1,000,000 Total Hours is nonsensical - that would suggest the player has been gaming since 1900s.

Although we have recognized these outliers, the rarity and isolated nature of these cases mean that their removal will not greatly impact our dataset. Consequently, we will drop these outlier rows. This leaves us with 3,329 rows, each with 18 features and 1 target variable.

In [ ]:
```
# Check the statistical summary of each column
df.describe()
```

Out[ ]:

	GameID	LeagueIndex	Age	HoursPerWeek	TotalHours	APM	Sele
count	3395.000000	3395.000000	3395.000000	3395.000000	3395.000000	3395.000000	
mean	4805.012371	4.184094	21.610604	16.070103	1047.200884	117.046947	
std	2719.944851	1.517327	4.185772	11.938253	17188.730209	51.945291	
min	52.000000	1.000000	16.000000	0.000000	-1065.000000	22.059600	
25%	2464.500000	3.000000	19.000000	8.000000	300.000000	79.900200	
50%	4874.000000	4.000000	21.000000	12.000000	500.000000	108.010200	
75%	7108.500000	5.000000	24.000000	20.000000	800.000000	142.790400	
max	10095.000000	8.000000	44.000000	168.000000	1000000.000000	389.831400	

In [ ]:
```
# Drop the rows with negative predicted values since it does not make sense
df = df.drop((df[(df['HoursPerWeek'] < 0) | (df['TotalHours'] < 0)]).index)
# Drop the rows with extreme values in HoursPerWeek and TotalHours
df = df.drop(df[df['HoursPerWeek'] == 168].index)
df = df.drop(df[df['TotalHours'] == 1000000].index)
```

In [ ]:
```
# GameID is not informative for modeling; store and delete it
output = df[cols_ex]
df = df.drop('GameID', axis = 1)
```

In [ ]:
```
# Check the data structure after manipulation
df.shape
```

Out[ ]: (3392, 19)

2. Feature Selection

At this stage, we intend to explore the correlations and relative importance of all features. We will use two techniques: a Correlation Heatmap and Lasso Regression.

The Correlation Heatmap helps identify potential risks of multicollinearity by highlighting high correlation values, while Lasso Regression provides an overview of the importance of various features. Considering the limited size of our dataset and the distinct information each column brings, we are circumspect in our feature selection process. In this context, we have decided to exclude two columns:

1) APM: This general measure of engagement appears to be redundant as its information is largely represented by other features, thereby limiting its unique contribution to the model

2) MinimapRightClicks: This is a minor feature with negligible predictive importance.

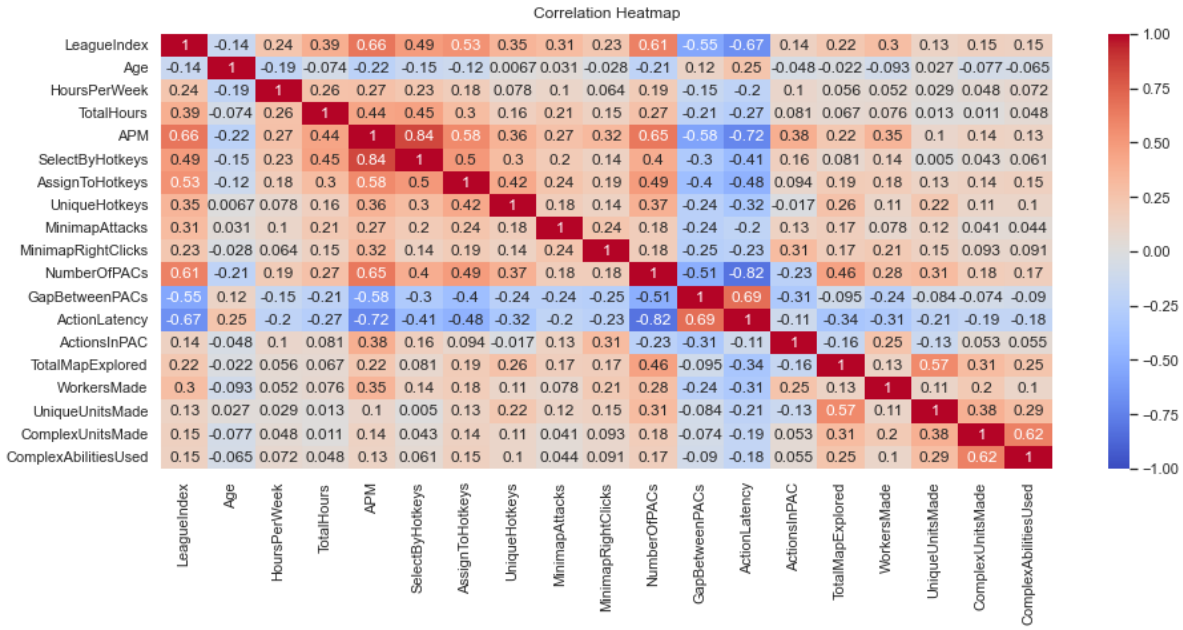
For future enhancements, we might consider the possibility of feature engineering to further optimize our model's performance.

2.1 Correlation Heatmap

The heatmap reveals substantial multicollinearity, particularly among 'APM' and 'ActionLatency', both of which exhibit strong correlations with several other features. These two variables appear to serve as "aggregate indicators", assembling the effects of some other features within the game.

In [ ]:
```
plt.figure(figsize=(15, 6))
heatmap = sns.heatmap(df.corr(), vmin=-1, vmax=1, annot=True, cmap='coolwarm')
heatmap.set_title('Correlation Heatmap', fontdict={'fontsize':12}, pad=12)
```

Out[ ]: Text(0.5, 1.0, 'Correlation Heatmap')



2.2 Lasso Regression

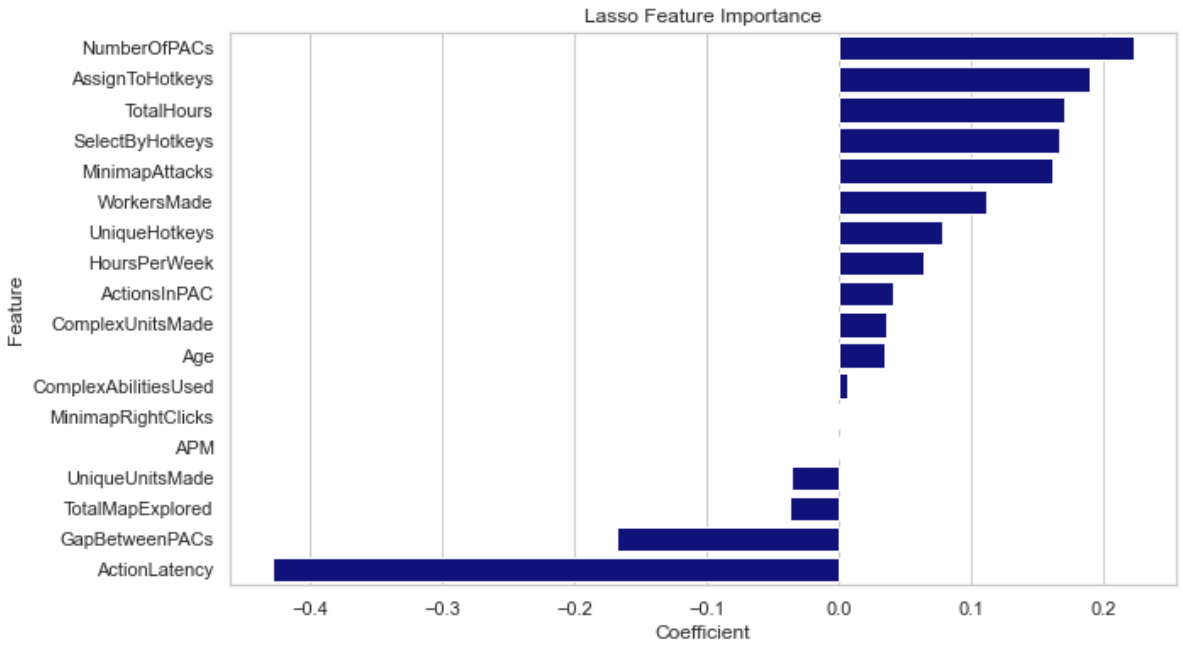
The Lasso Regression feature importance plot suggests that 'MinimapRightClicks' and 'APM' are the least important features.

In [ ]:
```
#  Lasso
X = df.drop('LeagueIndex', axis=1)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
y = df['LeagueIndex']
```

In [ ]:
```
# Create a LassoCV object to find the best alpha
lasso_cv = LassoCV(cv=10, random_state=0, max_iter=10000)
# Fit it to the scaled data
lasso_cv.fit(X_scaled, y)
print('Best alpha:', round(lasso_cv.alpha_, 4))
```

```
Best alpha: 0.0033
```

In [ ]:
```
lasso = Lasso(alpha = lasso_cv.alpha_)
lasso.fit(X_scaled, y)
coef = pd.Series(lasso.coef_, index=X.columns).sort_values(ascending=False)
plt.figure(figsize=(10,6))
sns.barplot(x=coef, y=coef.index, color='darkblue')
plt.title("Lasso Feature Importance")
plt.xlabel("Coefficient")
plt.ylabel("Feature")
plt.show()
```



In [ ]:
```
df = df.drop(['MinimapRightClicks', 'APM'], axis = 1)
```

3. Modeling & Deployment

In this phase, we aim to predict player ranks represented by the column 'LeagueIndex', which ranges from 1 (Bronze, the lowest) to 8 (Professional, the highest). This task can be seen as an "ordinal classification" problem, where the integer target variable symbolizes different levels of performance in StarCraft.

Given the ordinal nature of our target, we will approach this task as an **regression** or **ordinal classification** problem. We will apply both approaches to our modeling process.

For traditional regression modeling, RMSE serves as our primary performance metric. Nevertheless, from a practical perspective, decimal predictions are nonsensical. Therefore, we will round off our results to the nearest integer and use accuracy as our secondary performance metric.

For ordinal classification, we will apply the most commonly used stats model - Ordinal Logistic Regression. With probability predicted, we will select the rank with highest probability, and use accuracy as our metric.

In [ ]:
```
# Preparation:
# Split the dataset
X = df.drop('LeagueIndex', axis=1)
y = df['LeagueIndex']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_st

# Scale X_train and X_test for better interpretability
scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

In [ ]:
```
# Define functions for modeling, tuning and evaluation
def modeling_evaluation(model, X_train, y_train, X_test, y_test):
    # Fit the model
    model.fit(X_train, y_train)
    # Predict the result
    pred_train = model.predict(X_train)
    pred_test = model.predict(X_test)
    # Evaluation by RMSE
    rmse_train = mean_squared_error(y_train, pred_train, squared=False)
    rmse_test = mean_squared_error(y_test, pred_test, squared=False)
    print(f'Training RMSE: {rmse_train:.2f}')
    print(f'Testing RMSE: {rmse_test:.2f}')
    # Evaluation by Accuracy - round the result to integer
    pred_train_rounded = np.round(pred_train)
    pred_test_rounded = np.round(pred_test)
    # Rule: if predicted value < 1, it should be set to 1; if > 8, it should be set
    pred_train_rounded = np.where(pred_train_rounded < 1, 1, pred_train_rounded)
    pred_train_rounded = np.where(pred_train_rounded > 8, 8, pred_train_rounded)
    pred_test_rounded = np.where(pred_test_rounded < 1, 1, pred_test_rounded)
    pred_test_rounded = np.where(pred_test_rounded > 8, 8, pred_test_rounded)
    accuracy_train = np.mean(pred_train_rounded == y_train)
    accuracy_test = np.mean(pred_test_rounded == y_test)
    print(f'Training Accuracy: {accuracy_train:.2f}')
    print(f'Testing Accuracy: {accuracy_test:.2f}')

def tuning(model, param, X, y):
    grid_search = GridSearchCV(model, param, cv=5, scoring='neg_root_mean_squared_e
    grid_search.fit(X, y)
    # Extract the best model
    return grid_search.best_estimator_
```

3.1 Modeling & Tuning

We tested and tuned four distinct models to identify the most suitable for deployment:

1) Linear Regression: This is a widely applied model, appreciated for its simplicity and outstanding interpretability.

2) kNN: A simple, non-parametric method that performs well on smaller datasets.

3) Random Forest: An advanced ensemble, tree-based model lauded for its robustness and interpretability.

4) XGBoost: A powerful gradient boosting tree-based model with excellent performance and interpretability.

5) Ordinal Logistic Regression: An ordinal classification approach, with statistical robustness and great interpretability.

Model 1. Linear Regression

In [ ]:
```
lr = LinearRegression()
modeling_evaluation(lr, X_train, y_train, X_test, y_test)
```

```
Training RMSE: 0.96
Testing RMSE: 1.01
Training Accuracy: 0.40
Testing Accuracy: 0.38
```

Model 2. kNN

In [ ]:
```
knn = KNeighborsRegressor()
param = {
    'n_neighbors': [5, 10, 15, 20],
    'weights': ['uniform', 'distance'],
    'metric': ['euclidean', 'manhattan']
}
knn_best = tuning(knn, param, X_train_scaled, y_train)
modeling_evaluation(knn_best, X_train_scaled, y_train, X_test_scaled, y_test)
```

```
Training RMSE: 0.00
Testing RMSE: 1.03
Training Accuracy: 1.00
Testing Accuracy: 0.34
```

Model 3. Random Forest Regression

In [ ]:
```
rf = RandomForestRegressor(random_state=0)

# Set up the hyperparameter grid for tuning
param = {
    'n_estimators': [10, 50, 100, 200],
    'max_depth': [3, 4, 5, 6, 7, 8, 9, 10],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}
rf_best = tuning(rf, param, X_train, y_train)
modeling_evaluation(rf_best, X_train, y_train, X_test, y_test)
```

```
Training RMSE: 0.59
Testing RMSE: 0.96
Training Accuracy: 0.61
Testing Accuracy: 0.37
```

Model 4. XGBoost Regression

In [ ]:
```
xgboost = xgb.XGBRegressor(random_state=0)
param = {
    'n_estimators': [10, 50, 100, 200],
    'learning_rate': [0.01, 0.1, 0.3],
    'max_depth': [3, 4, 5, 6, 7, 8, 9, 10],
    'subsample': [0.6, 0.8, 1.0],
    'colsample_bytree': [0.6, 0.8, 1.0]
}
xgb_best = tuning(xgboost, param, X_train, y_train)
modeling_evaluation(xgb_best, X_train, y_train, X_test, y_test)
```

```
Training RMSE: 0.77
Testing RMSE: 0.96
Training Accuracy: 0.49
Testing Accuracy: 0.39
```

Model 5. Ordinal Logistic Regression

In [ ]:
```
olr = OrderedModel(y_train, X_train, distr='logit')
result = olr.fit(method='bfgs')
pred_train_prob = result.predict(X_train)
pred_train = np.argmax(pred_train_prob.values, axis=1) + 1
accuracy_train = np.mean(pred_train == y_train)
pred_test_prob = result.predict(X_test)
pred_test = np.argmax(pred_test_prob.values, axis=1) + 1
accuracy_test = np.mean(pred_test == y_test)
# Although this is basically a classification model, we still want to apply rmse fo
rmse_train = mean_squared_error(y_train, pred_train, squared=False)
rmse_test = mean_squared_error(y_test, pred_test, squared=False)
print(f'Training RMSE: {rmse_train:.2f}')
print(f'Testing RMSE: {rmse_test:.2f}')
print(f'Training Accuracy: {accuracy_train:.2f}')
print(f'Testing Accuracy: {accuracy_test:.2f}')
```

```
Optimization terminated successfully.
         Current function value: 1.313120
         Iterations: 276
         Function evaluations: 284
         Gradient evaluations: 284
Training RMSE: 1.00
Testing RMSE: 1.03
Training Accuracy: 0.42
Testing Accuracy: 0.41
```

3.2 Model Evaluation

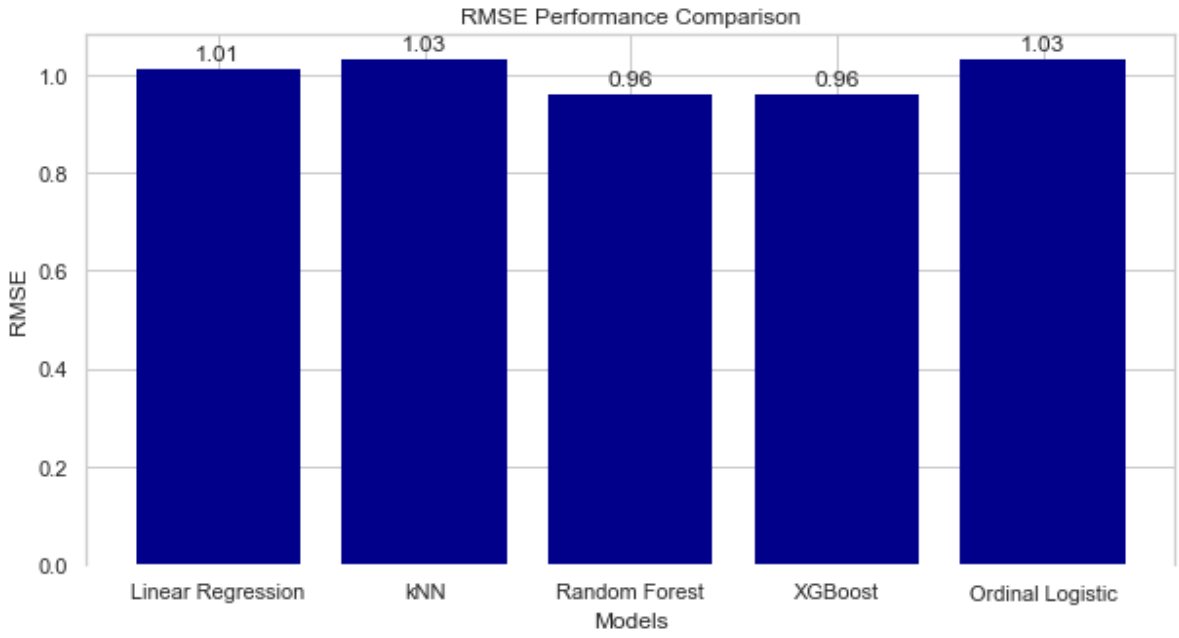
To select the best model, we combined both **RMSE** and **accuracy** as our evaluators, and apply to our test dataset. According to the results, Random Forest/XGBoost yielded the lowest RMSE (0.96), while ordinal logistic regression displayed the highest accuracy score (42%). Linear Regression and Ordinal Logistic Regression's result are comparably robust across train and test dataset.

Taking into account the interpretability and robustness, we have decided to select **Ordinal Logistic Regression** as our optimal model.

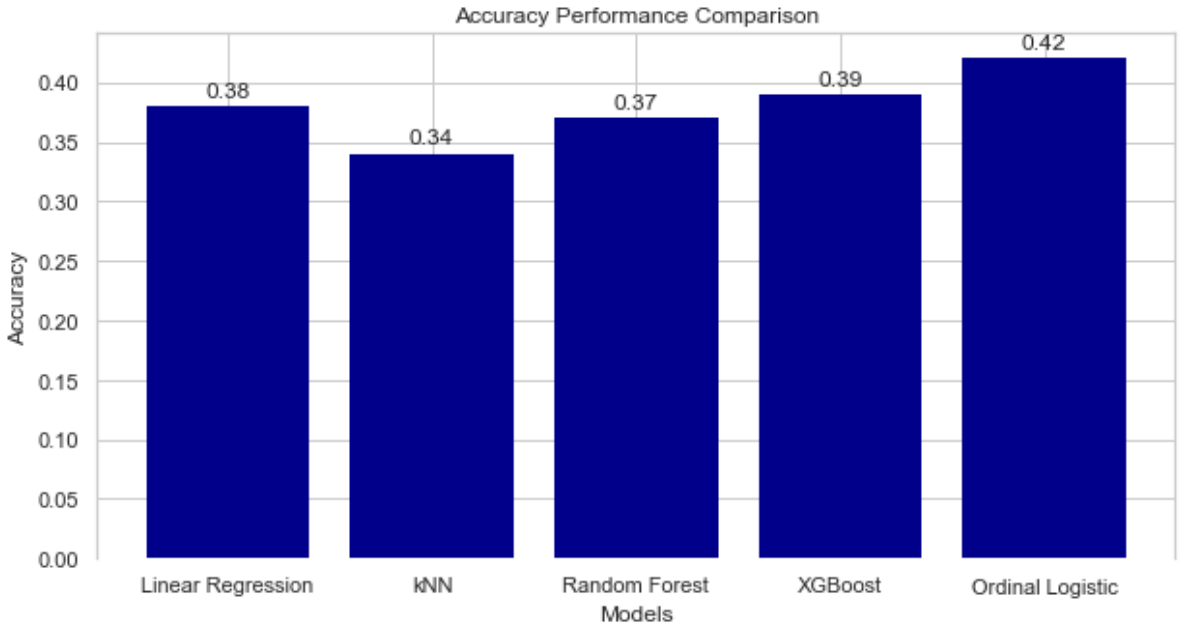
To aid the selection process, the performance of each model is visually represented in the bar plots below, one for RMSE and the other for accuracy.

In [ ]:
```
# Create a bar plot for performance viz
def evaluation_viz(evaluator, name, model):
    # Define a function to add annotations to each bar
    def annotate_bars(ax, bars):
        for bar in bars:
            height = bar.get_height()
            ax.annotate(
                f"{height:.2f}",
                xy=(bar.get_x() + bar.get_width() / 2, height),
                xytext=(0, 3),
                textcoords="offset points",
                ha="center",
                va="bottom",
            )

    fig, ax = plt.subplots(figsize=(10, 5))
    xbars = ax.bar(model, evaluator, color='darkblue', edgecolor="none")
    annotate_bars(ax, xbars)
    ax.spines['bottom'].set_visible(False)

    plt.xlabel('Models')
    plt.ylabel(name)
    plt.title(f'{name} Performance Comparison')
    plt.show()
```

In [ ]:
```
# Viz: RMSE Comparison
rmse = [1.01, 1.03, 0.96, 0.96, 1.03]
model = ['Linear Regression', 'kNN', 'Random Forest', 'XGBoost', 'Ordinal Logistic
evaluation_viz(rmse, 'RMSE', model)
```



In [ ]:
```
# Viz: Accuracy Comparison
accuracy = [0.38, 0.34, 0.37, 0.39, 0.42]
evaluation_viz(accuracy, 'Accuracy', model)
```



3.3 Interpretations

3.3.1 Statistical Summary

Given the statistical result, we can generate some insights:

1) 'Age', 'HoursPerWeek', 'TotalHours' shows positive associations with 'LeagueIndex' - more experienced players might have more strategic understanding of the game and tend to win more.

2) 'SelectByHotkeys', 'AssignToHotkeys', 'UniqueHotkeys', 'MinimapAttacks', 'NumberOfPACs', 'ActionsInPAC' and **'WorkersMade'** are positively associated, while **'ActionLatency'** is negatively associated with 'LeagueIndex' - More focusing actions & engagement with quicker response can lead to a higher overall performance.

3) 'TotalMapExplored' is negatively related to the rank - seems counterintuitive; could possibly be due to efficient players focusing on key areas rather than exploring the entire map or could be indicative of a different strategy that doesn't require complete map exploration.

4) 'UniqueUnitsMade', 'ComplexAbilitiesUsed' and **'ComplexUnitsMade'** are not statistically significant in prediction - Complicated gameplay strategy does not necessarily lead to advanced rank; the concentration and reaction speed are more crucial than the mere production of unique and complex units.

5) Coefficients associated with the ordinal thresholds are all statistically significant - There are significant behavior differences between each adjacent rank

In [ ]:
```
result.summary()
```

Out[ ]:

OrderedModel Results

Dep. Variable:	LeagueIndex	**Log-Likelihood:**	-3340.6
Model:	OrderedModel	**AIC:**	6727.
Method:	Maximum Likelihood	**BIC:**	6862.
Date:	Sun, 28 May 2023		
Time:	20:41:09		
No. Observations:	2544		
Df Residuals:	2521		
Df Model:	23		

	coef	std err	z	P>|z|	[0.025	0.975]
Age	0.0130	0.009	1.391	0.164	-0.005	0.031
HoursPerWeek	0.0086	0.004	2.381	0.017	0.002	0.016
TotalHours	0.0007	8.03e-05	9.157	0.000	0.001	0.001
SelectByHotkeys	72.8209	10.366	7.025	0.000	52.504	93.138
AssignToHotkeys	1611.1112	228.187	7.060	0.000	1163.873	2058.349
UniqueHotkeys	0.0701	0.018	3.882	0.000	0.035	0.105
MinimapAttacks	2277.7793	286.130	7.961	0.000	1716.975	2838.584
NumberOfPACs	684.3530	98.434	6.952	0.000	491.425	877.281
GapBetweenPACs	-0.0164	0.003	-4.723	0.000	-0.023	-0.010
ActionLatency	-0.0312	0.005	-6.536	0.000	-0.041	-0.022
ActionsInPAC	0.0715	0.037	1.935	0.053	-0.001	0.144
TotalMapExplored	-0.0198	0.007	-2.882	0.004	-0.033	-0.006
WorkersMade	321.7212	82.121	3.918	0.000	160.767	482.675
UniqueUnitsMade	-0.0575	0.026	-2.224	0.026	-0.108	-0.007
ComplexUnitsMade	366.0509	441.565	0.829	0.407	-499.400	1231.502
ComplexAbilitiesUsed	95.4569	180.339	0.529	0.597	-258.001	448.915
1/2	-2.6891	0.744	-3.616	0.000	-4.147	-1.232
2/3	0.5480	0.060	9.118	0.000	0.430	0.666
3/4	0.4435	0.044	10.014	0.000	0.357	0.530
4/5	0.5491	0.036	15.116	0.000	0.478	0.620
5/6	0.6519	0.037	17.590	0.000	0.579	0.725
6/7	1.4106	0.050	28.068	0.000	1.312	1.509
7/8	0.3966	0.185	2.141	0.032	0.034	0.760

3.3.2 Feature Importance

Here we want to know about the feature from another perspective - which features are more important, and which are not?

To approach this, we used the scaled data (to get equalized initial scale of each feature) to train the model, and the absolute value of coefficient here can represent the feature importance. Our insights include:

1) 'TotalHours' is the most impactful feature - Practice makes perfect!

2) 'NumberOfPACs', 'ActionLatency', 'SelectByHotkeys' and **'MinimapAttacks'** are also among the top contributors, highlighting the significance of the player's engagement with the game controls

In [ ]:
```
# Fit the model with scaled features
olr_scaled = OrderedModel(y_train, X_train_scaled, distr='logit')
result2 = olr_scaled.fit(method='bfgs')
coef = result2.params
coef = coef[:-7]
features = X_train.columns
coef_df = pd.DataFrame({
    'Feature': X_train.columns,
    'Coefficient': coef.values,
})
coef_df['importance'] = np.abs(coef_df['Coefficient'])
coef_df = coef_df.sort_values('importance', ascending=False)

print(coef_df[['Feature', 'importance']])
```

```
Optimization terminated successfully.
         Current function value: 1.313120
         Iterations: 53
         Function evaluations: 54
         Gradient evaluations: 54
                Feature  importance
2             TotalHours    0.880622
7            NumberOfPACs    0.686649
9           ActionLatency    0.605735
6           MinimapAttacks    0.381986
3          SelectByHotkeys    0.381296
4          AssignToHotkeys    0.369120
8           GapBetweenPACs    0.276230
5            UniqueHotkeys    0.165927
12             WorkersMade    0.165799
11       TotalMapExplored    0.145410
13        UniqueUnitsMade    0.107685
10           ActionsInPAC    0.104954
1            HoursPerWeek    0.101488
0                     Age    0.053949
14       ComplexUnitsMade    0.041577
15  ComplexAbilitiesUsed    0.025511
```

3.4 Prediction Result Output

In [ ]:
```
pred_prob = result.predict(X)
pred = np.argmax(pred_prob.values, axis=1) + 1
output['LeagueIndex_Prediction'] = pred
output
```

Out[ ]:

	GameID	LeagueIndex	LeagueIndex_Prediction
0	52	5	6
1	55	5	6
2	56	4	4
3	57	3	4
4	58	3	4
...	...	...	...
3390	10089	8	8
3391	10090	8	8
3392	10092	8	8
3393	10094	8	8
3394	10095	8	8

3392 rows × 3 columns

In [ ]:
```
output.to_csv('output.csv', index=False)
```

4. Key Findings - for Stakeholders

Project Overview

Our team was tasked with creating a predictive model to evaluate player performance in StarCraft, with ~3400 rows of data. Using 18 different features related to player behavior and game engagement, we aimed to develop a predictive model that could offer insights into what actions and behaviors led to higher rankings (LeagueIndex).

After testing with several models, we decided to apply ordinal logistic regression model, a model that provides both ordinal integer ranking prediction and robustness in postive/negative relationship between rank and gamers' features.

Performance Metrics Summary

1) The model correctly predicts the exact ranks of **~41%** of players in our test dataset. While there is room for improvement, this level of precision is quite informative in understanding player behaviors that lead to success.

2) The average deviation between our model's predictions and the actual ranks is **~1** rank. Furthermore, 95% of our predictions land within a span of ±2 ranks of a player's actual ranking, which signifies that our model's predictions are generally close to the actual outcomes.

Key Insights

Several significant relationships were found through the model -

1) Age Matters: The model reveals a positive association between age and rank, indicating a potential advantage for older players, possibly reflecting their accrued experience and strategic acumen.

2) Practice Makes Perfect: Generally, more hours (reflected by hours per week and total hours) played can lead to higher rank. Of all features, **Total Hours** appears to be the most influential feature.

3) Active Involvement and Swift Reactions Lead to Excellence: Engagement and responsiveness during gameplay primarily influence player ranking. The gamers behaviors & actions in Hot Keys, Minimaps and PACs and Workers made are positively associated with ranking, whereas Action Latency shows a negative association.

3) Strategic Focus Over Extensive Map Exploration: Interestingly, the extent of the maps explored shows a negative relationship with player rank. This suggests that top-performing players may concentrate on strategic areas instead of investing time in extensive exploration.

4) Complexity Doesn't Equate to Victory Certain aspects of game complexity, such as Unique Units Made, Complex Abilities Used and Complex Units Made, do not significantly impact player ranking. This highlights that sophisticated gameplay strategies do not inherently lead to better rankings. It appears that player focus and response speed are more important than the sheer production of unique or complex units.

5. Suggestions on Further Data Collection

Increase the Data Size: Considering the moderate R-squared value and indications of multicollinearity, obtaining more data could potentially enhance the model's accuracy and robustness. Regarding the data size, 50k-100k rows can be a good start point.

Align Data Collection with Business Goals: Make sure to evaluate the current dataset in terms of alignment with the stakeholders' business goals. For instance,

1) If the aim is to draw insights applicable to a broad user base, ensure the data accurately reflects the overall population of players. If there are specific target groups, consider stratified sampling to focus on these segments.

2) We should clarify the goal of rank prediction: if we want to use the data to issue an initial rank for players without ranking experience, we should use only the gaming stats of the "Unranked" matches; if we want to predict the potential rank of an existing ranking gamer, we should only use his stats in ranking games, since his performance can be quite different in ranking and "Unranked" matches.

Impute Missing Values in Professional Rank: The lack of data for rank 8 players (professionals) is a notable gap in the current dataset. If feasible, work towards collecting data from professional players. Their playstyle and strategies can provide invaluable insights and balance to the model and help us track the unique characteristics of the professional players.

Feature Expansion and Engineering: The model's performance and our understanding of player behavior could be improved by introducing more diverse and meaningful features. Some suggestions include:

1) Average Game Session Length: This could potentially help capture the duration of concentration

2) In-Game Communication Frequency: In 2v2/4v4 mode, the frequency of communication could reflect cooperation and strategic depth

3) In-Game Communication Content: Use sentiment analysis to judge the overall mindset tendency of a player

4) Day/Night Time Preference: Players ususally play at Daytime/Night-time may show different performance tendency. We can use the ratio of Daytime played as the feature

5) Engineering existing features: Combining or transforming certain existing features may help reduce multicollinearity and reveal new insights. For example, a ratio between 'SelectByHotkeys' and 'APM' might better reflect a player's efficiency.