Starcraft LeagueIndex Classification

Shreayan Chaudhary

Approach to solve the problem:

Predicting a player’s rank using the information provided in the dataset:

1. Exploratory Data Analysis (EDA):
   Perform an initial exploration of the dataset to understand its structure, features, and relationships. Helps in gaining insights and identifying any data quality issues or missing values. Tasks to perform during EDA:
   a. Identify missing values.
   b. Generate descriptive statistics of the data.
   c. Visualize the distribution of target variable and features using plots (histograms, box plots, etc.) from libraries like matplotlib and seaborn.

2. Data Preprocessing:
   a. Handle missing values: Depending on the extent of missing data, we can choose to drop rows or columns with missing values or impute them using techniques like mean, median, or mode.
   b. Encode categorical variables: If there are categorical variables in the dataset, encode them using techniques like one-hot encoding or label encoding, depending on the nature of the variables.
   c. Detect and handle outliers.
   d. Split the data: Divide the dataset into training and testing sets.

3. Feature Selection or Engineering:
   Based on the insights gained during EDA, we might need to perform feature selection or engineering to improve the model's performance. This can involve removing irrelevant or highly correlated features, creating new features, or transforming existing ones.

4. Model Training and Evaluation:
   a. Experiment with different machine learning algorithms for classification like logistic regression, decision trees, random forest, or support vector machines, since predicting rank is a classification problem.
   b. Train the model using the training data.
   c. Evaluate the model's performance using appropriate evaluation metrics such as accuracy, precision, recall, and F1 score. Use the testing data for evaluation.

5. Model Improvement and Tuning: If the model's performance is not satisfactory, try different algorithms or tune hyperparameters to improve it using techniques like grid search or random search.

6. Model Deployment and Communication: Once we have a satisfactory model, we can deploy it to make predictions on new data. Document our findings, methodology, and

evaluation results in a clear and concise manner, suitable for non-technical stakeholders. Communicate the findings to stakeholders using visualizations and data story based explanations that are easily understandable.

Hypothetical:

If stakeholders want to collect more data, we can advise them based on our EDA and model results:

1. Identify the areas where the dataset is lacking or where more data could be beneficial. For example, if certain features have a high correlation with the target variable but are limited in the current dataset, suggest collecting more data for those features.
2. Assess if there are any class imbalances or biases in the data and suggest collecting more data to address these issues.
3. Analyze if certain subsets of the data are underrepresented and recommend collecting more data to balance the representation.
4. Consider any specific insights or patterns observed during the model building process and suggest collecting more data to validate or refine those findings.
5. By providing guidance on data collection based on our EDA and model results, we can help stakeholders improve the model's accuracy and robustness.

Imports

In [1]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from scipy.stats import iqr, zscore
from sklearn import metrics

pd.set_option('display.max_rows', 500)
pd.set_option('display.max_columns', 500)
pd.set_option('display.width', 1000)
```

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

Out[2]:
```
(3395, 20)
```

Exploratory Data Analysis

In [3]:

```
data.head()
```

Out[3]:

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

In [4]:

```
data.describe()
```

Out[4]:

	GameID	LeagueIndex	APM	SelectByHotkeys	AssignToHotkeys	UniqueH
count	3395.000000	3395.000000	3395.000000	3395.000000	3395.000000	3395
mean	4805.012371	4.184094	117.046947	0.004299	0.000374	4
std	2719.944851	1.517327	51.945291	0.005284	0.000225	2
min	52.000000	1.000000	22.059600	0.000000	0.000000	0
25%	2464.500000	3.000000	79.900200	0.001258	0.000204	3
50%	4874.000000	4.000000	108.010200	0.002500	0.000353	4
75%	7108.500000	5.000000	142.790400	0.005133	0.000499	6
max	10095.000000	8.000000	389.831400	0.043088	0.001752	10

Impute missing values

In [5]:

```
# There are some question marks in the data. Literally!
data[data['Age']=='?']
```

Out[5]:

	GameID	LeagueIndex	Age	HoursPerWeek	TotalHours	APM	SelectByHotkeys
3340	10001	8	?	?	?	189.7404	0.004582
3341	10005	8	?	?	?	287.8128	0.029040
3342	10006	8	?	?	?	294.0996	0.029640
3343	10015	8	?	?	?	274.2552	0.018121
3344	10016	8	?	?	?	274.3404	0.023131
3345	10017	8	?	?	?	245.8188	0.010471
3346	10018	8	?	?	?	211.0722	0.013049
3347	10021	8	?	?	?	189.5778	0.007559
3348	10022	8	?	?	?	210.5088	0.007974
3349	10023	8	?	?	?	248.0118	0.014722
3350	10024	8	?	?	?	299.2290	0.026428

All the players with LeagueIndex=8 don't have Age, HrsPerWk and TotalHrs. It can be handled in the following ways:

1. Regression Imputation:
   a. Treat age as the target variable and use other relevant features in the dataset to predict the missing ages.
   b. Split the data into two sets: one with non-missing ages (training set) and the other with missing ages (test set).
   c. Train a regression model (e.g., linear regression, random forest regression) on the training set, using other features as predictors and age as the target variable.
   d. Use the trained model to predict the missing ages in the test set.
   e. Replace the missing values with the predicted ages.
   This approach assumes that there is a relationship between the missing age values and the other features in the dataset, allowing the model to make accurate predictions.

2. K-Nearest Neighbors (KNN) Imputation:
   a. Identify the k nearest neighbors for each data point with a missing age based on the available features.
   b. Calculate the average age of the k nearest neighbors.
   c. Replace the missing values with the calculated average age.
   This approach assumes that similar individuals (based on other features) are likely to have similar ages.

3. Mean or Median Imputation:
   a. Calculate the mean or median age of the available data (non-missing values).
   b. Replace the missing values with the calculated mean or median age.

This approach assumes that the missing values are missing completely at random and that the mean or median age is a representative value for imputation.

For now, we will use mean or median imputation, but other imputation techniques might give better results.

In [6]:
```
# some of the columns have '?' instead of missing values
data.replace('?', np.nan, inplace=True)
data.isna().sum()
```

Out[6]:
```
GameID                  0
LeagueIndex             0
Age                    55
HoursPerWeek           56
TotalHours             57
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
```

Missing values in age, hrs per week and total hours cols - need to imputate them!

In [7]:
```
data['Age'].astype(float).describe()
```

Out[7]:
```
count    3340.000000
mean       21.647904
std         4.206341
min        16.000000
25%        19.000000
50%        21.000000
75%        24.000000
max        44.000000
Name: Age, dtype: float64
```

In [8]:
```
# imputing the age column using mean value since age is normally distribute
data['Age'].fillna(data['Age'].astype(float).mean(), inplace=True)
data['Age'] = data['Age'].astype(float)
```

In [9]:
```
data['HoursPerWeek'].astype(float).describe()
```

Out[9]:
```
count    3339.000000
mean       15.910752
std        11.962912
min         0.000000
25%         8.000000
50%        12.000000
75%        20.000000
max       168.000000
Name: HoursPerWeek, dtype: float64
```

In [10]:
```
# imputing the hrs per week column using median since it has outliers, and
data['HoursPerWeek'].fillna(data['HoursPerWeek'].median(), inplace=True)
data['HoursPerWeek'] = data['HoursPerWeek'].astype(int)
```

In [11]:
```
data['TotalHours'].astype(float).describe()
```

Out[11]:
```
count       3338.000000
mean         960.421809
std        17318.133922
min            3.000000
25%          300.000000
50%          500.000000
75%          800.000000
max      1000000.000000
Name: TotalHours, dtype: float64
```

In [12]:
```
# imputing the total hrs column using median since it has some massive outl
data['TotalHours'].fillna(data['TotalHours'].median(), inplace=True)
data['TotalHours'] = data['TotalHours'].astype(int)
```

In [13]:
```
# Identifying the unique number of values in the dataset
data.nunique()
```

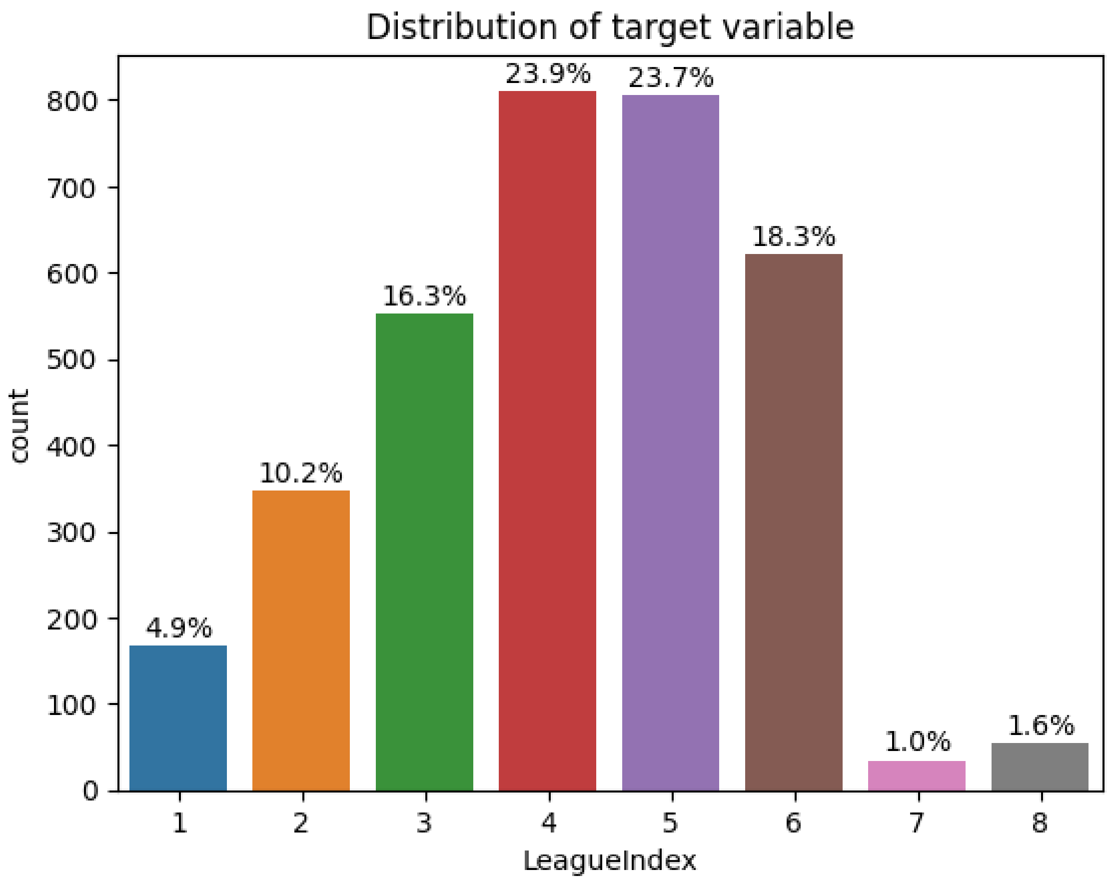
Out[13]:
```
GameID                  3395
LeagueIndex                8
Age                       29
HoursPerWeek              32
TotalHours               237
APM                     3374
SelectByHotkeys         3375
AssignToHotkeys         3361
UniqueHotkeys             11
MinimapAttacks          2471
MinimapRightClicks      3302
NumberOfPACs            3386
GapBetweenPACs          3358
ActionLatency           3367
ActionsInPAC            3223
TotalMapExplored          52
WorkersMade             3256
UniqueUnitsMade           12
ComplexUnitsMade        1110
ComplexAbilitiesUsed    1828
dtype: int64
```

In [14]:
```
# GameID is just an index column, so dropping it
data.drop(['GameID'], axis=1, inplace=True)
```

Data Visualizations

In [15]:

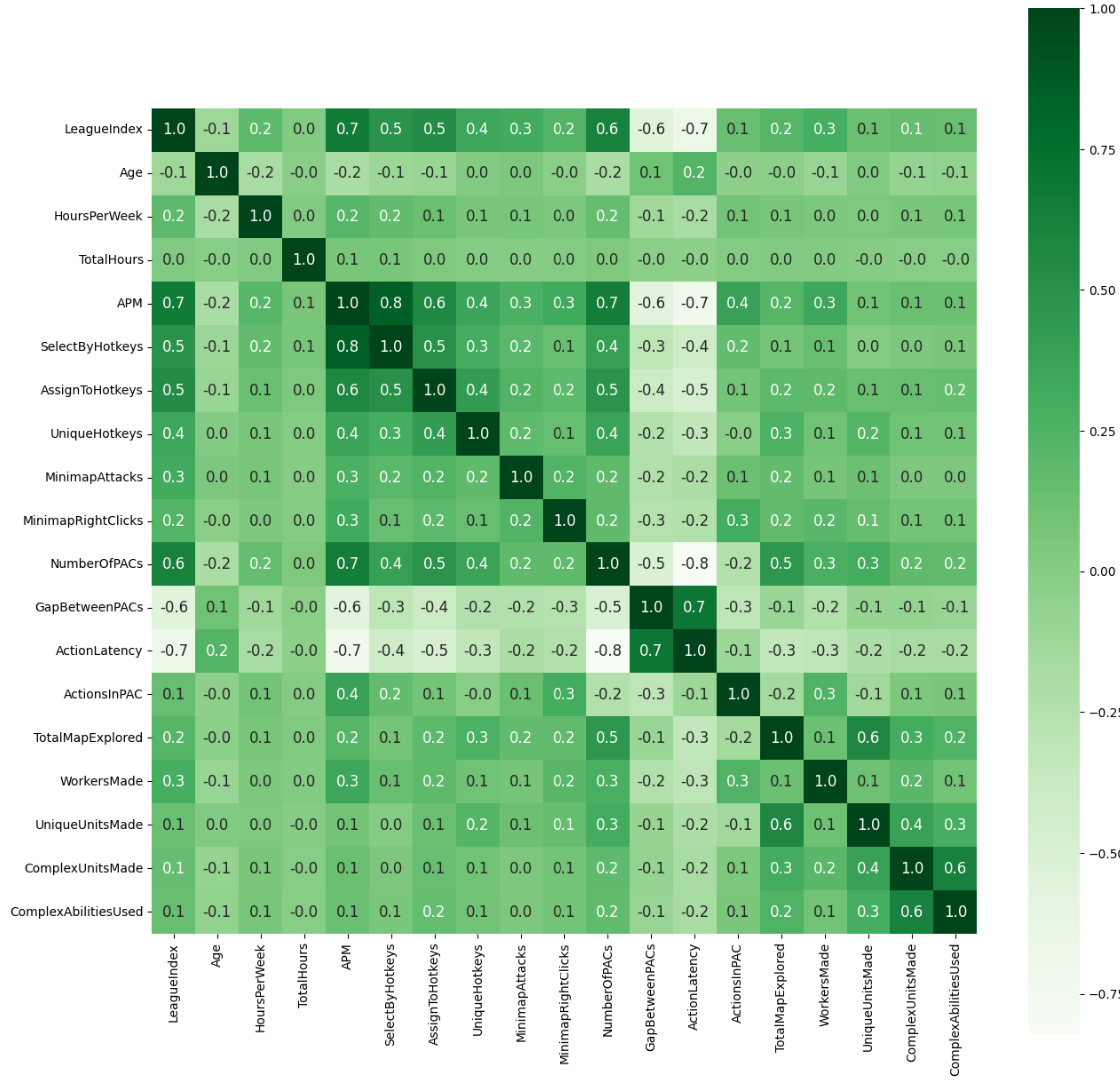
```
# distribution of the target variable to check if it is skewed or balanced
ax = sns.countplot(x='LeagueIndex', data=data)
for i, patch in enumerate(ax.patches):
    percentage = 100 * patch.get_height()/len(data)
    x = patch.get_x() + patch.get_width()/2
    y = patch.get_height()+10
    ax.annotate('{:.1f}%'.format(percentage), (x, y), ha='center')
plt.title('Distribution of target variable')
plt.show()
```



Target variable is skewed. This is not good - the ideal distribution should be uniform. We will address this later when training the model using SMOTE for fix the skewness of the target variable.

In [16]:

```
# Plotting the heatmap of correlation between features
plt.figure(figsize=(15,15))
sns.heatmap(data.corr(), cbar=True, square= True, fmt='.1f', annot=True, ar
plt.show()
```



The heatmap of the correlation plot is a very helpful plot that provides a visual representation of the correlation between pairs of variables in a dataset, and allows us to identify relationships, dependencies, and patterns among the variables.

1. Correlation Strength: The heatmap color-codes the correlation coefficients, making it easy to identify the strength and direction of the relationships. High positive correlations are represented by brighter colors (e.g., dark green), indicating that the variables move together. Negative correlations are represented by darker colors (e.g., white), suggesting an inverse relationship.
2. Feature Selection: The heatmap helps in feature selection by identifying highly correlated variables. If two variables are strongly correlated (either positively or negatively), it indicates that they carry similar information. In such cases, we may choose to remove one of the variables to avoid multicollinearity and reduce redundancy in our analysis.
3. Multivariate Analysis: The heatmap allows for multivariate analysis by showing the correlation between all pairs of variables simultaneously. This helps in identifying clusters or groups of variables that are highly correlated with each other. Such groups can provide

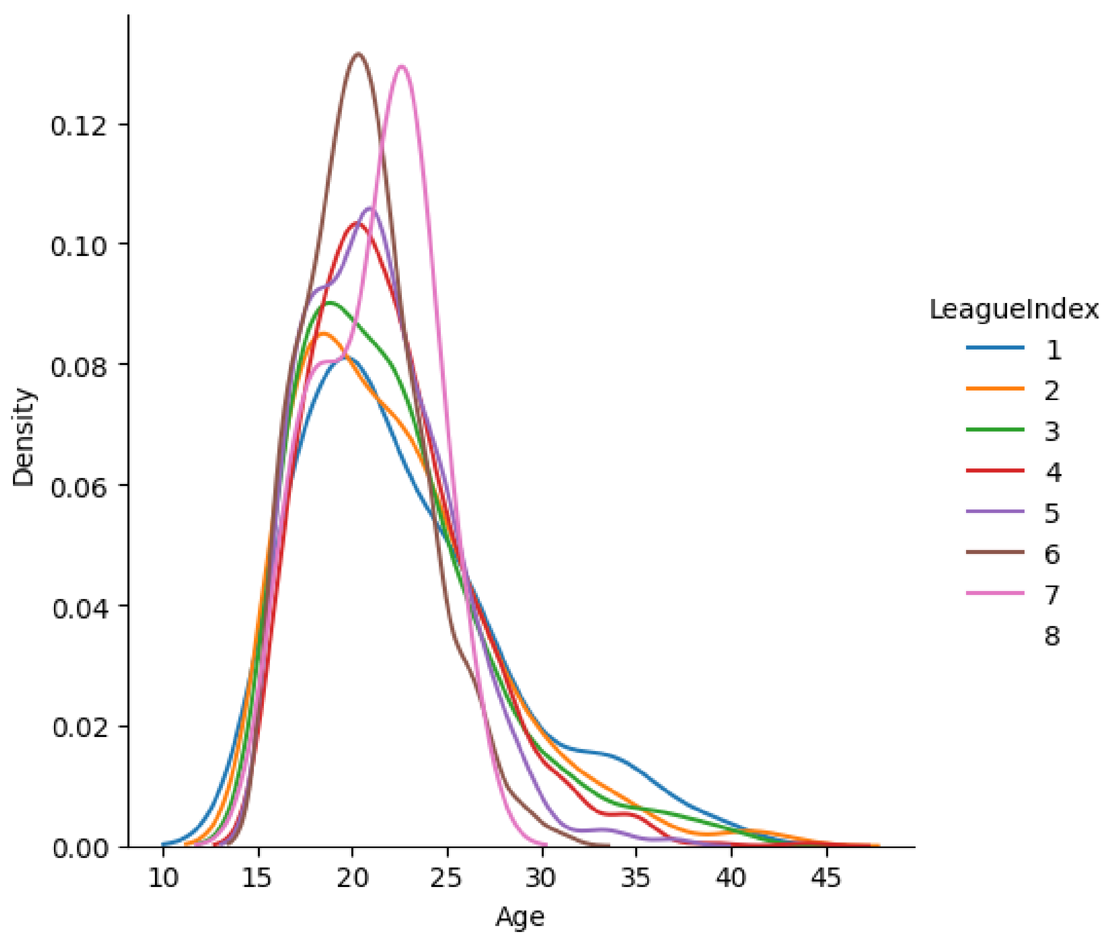
insights into underlying patterns or relationships within the data.

4. Missing Data and Imputation: The heatmap can reveal missing data patterns and help in deciding on an appropriate strategy for imputing missing values. If there are correlations between missing values in different variables, it may indicate a systematic pattern or relationship. This understanding can guide the imputation process or suggest the need for additional data collection.
5. Model Building: The heatmap can assist in model building by identifying variables that are strongly correlated with the target variable. Variables with high correlation can be considered as potential predictors in the model. Additionally, the heatmap can reveal any correlations between predictors, helping in understanding the potential impact of multicollinearity on the model's performance.

In [17]:

```
# Distribution density plot KDE (kernel density estimate) for age
sns.FacetGrid(data, hue="LeagueIndex", height=5).map(sns.kdeplot, "Age").ac
plt.show()
```

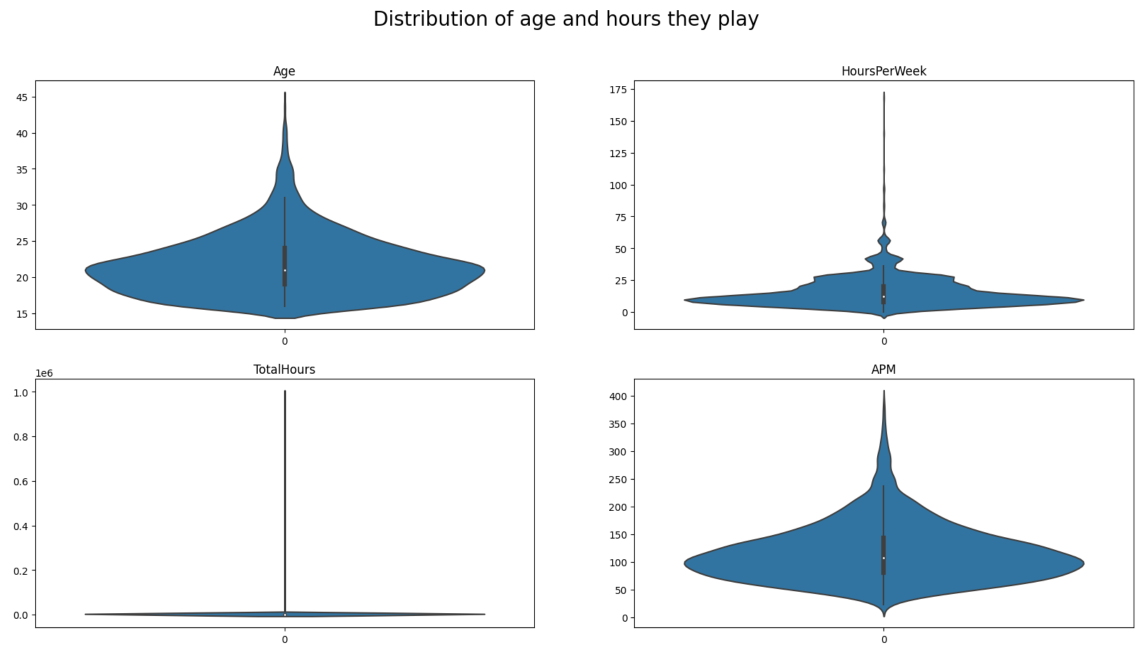
```
C:\Users\shrea\anaconda3\envs\ppi_pred\lib\site-packages\seaborn\axisgri
d.py:848: UserWarning: Dataset has 0 variance; skipping density estimate.
Pass `warn_singular=False` to disable this warning.
  func(*plot_args, **plot_kwargs)
```



Younger players between 15-27 are more likely to have a higer rank. After 30 years, the average skill level starts decreasing.

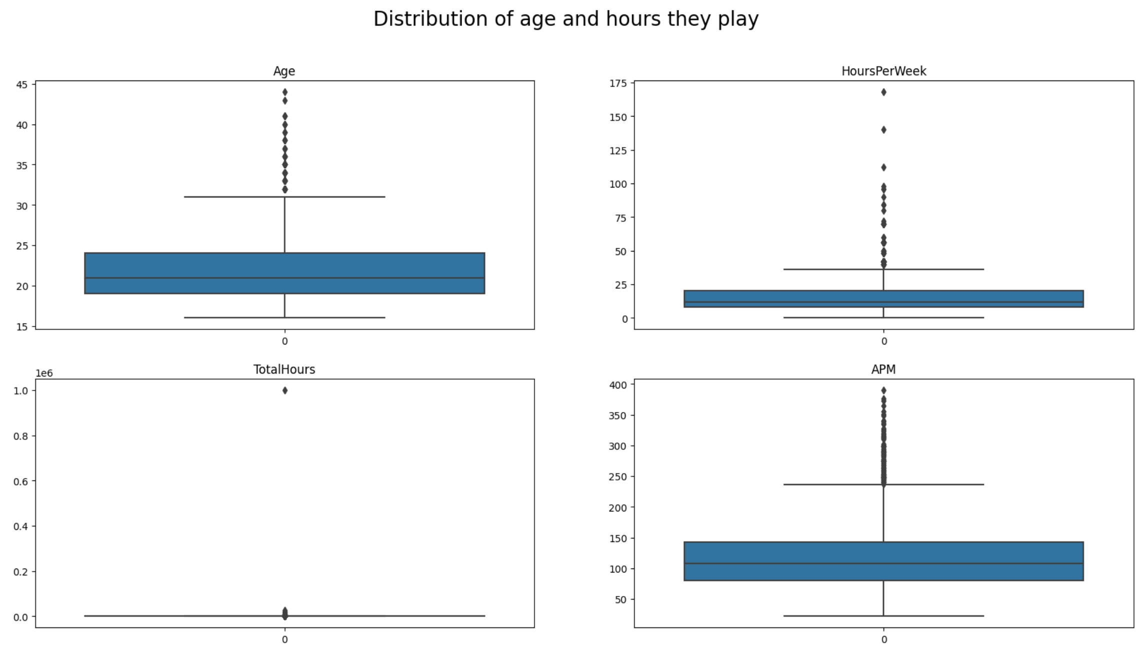
In [18]:

```
# distribution of age and hours they play
fig,ax = plt.subplots(2,2, figsize=(20,10))
plt.suptitle("Distribution of age and hours they play", fontsize=20)
sns.violinplot(data['Age'], ax = ax[0][0])
ax[0][0].set_title('Age')
sns.violinplot(data['HoursPerWeek'], ax = ax[0][1])
ax[0][1].set_title('HoursPerWeek')
sns.violinplot(data['TotalHours'], ax = ax[1][0])
ax[1][0].set_title('TotalHours')
sns.violinplot(data['APM'], ax = ax[1][1])
ax[1][1].set_title('APM')
plt.show()
```



A violin plot is a visualization that combines aspects of a box plot and a kernel density plot. It displays the distribution of a continuous variable across different categories or groups. The width of the violin indicates the density of the data at different values, and the white dot represents the median. The violin plot provides insights into the data's distribution, including the presence of multiple modes and skewness, making it useful for comparing distributions and identifying potential outliers.

In [19]:
```
# distribution of age and hours they play
fig,ax = plt.subplots(2,2, figsize=(20,10))
plt.suptitle("Distribution of age and hours they play", fontsize=20)
sns.boxplot(data['Age'], ax = ax[0][0])
ax[0][0].set_title('Age')
sns.boxplot(data['HoursPerWeek'], ax = ax[0][1])
ax[0][1].set_title('HoursPerWeek')
sns.boxplot(data['TotalHours'], ax = ax[1][0])
ax[1][0].set_title('TotalHours')
sns.boxplot(data['APM'], ax = ax[1][1])
ax[1][1].set_title('APM')
plt.show()
```



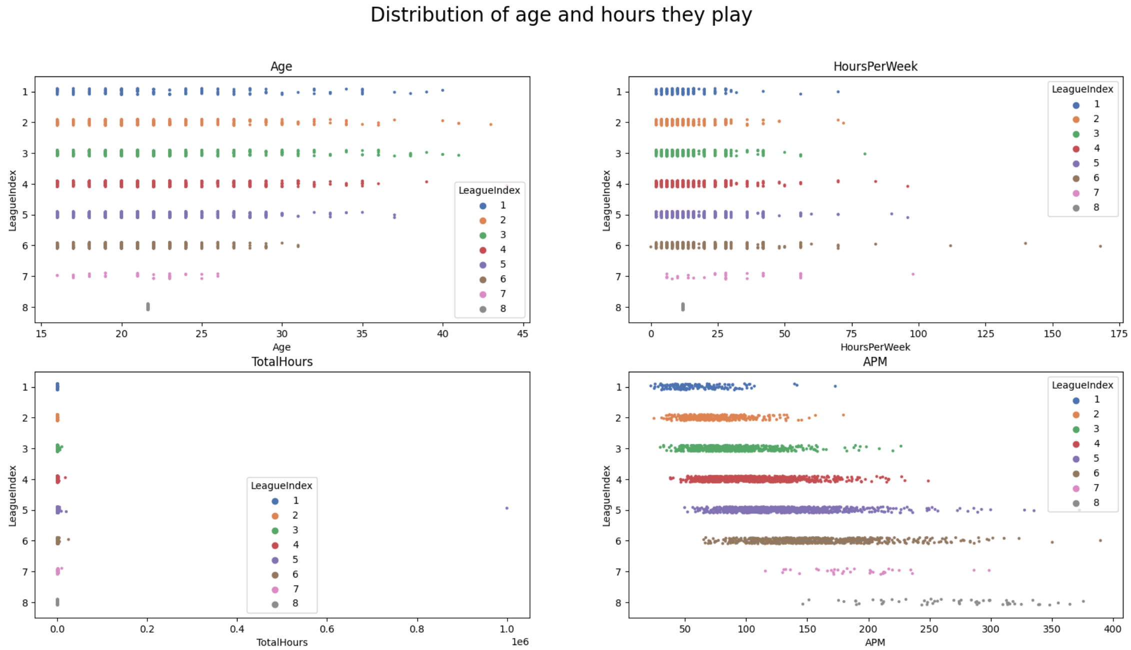
A box plot, also known as a box-and-whisker plot, is a visual representation of the distribution of a continuous variable. It displays the median, quartiles, and potential outliers of the data. The box represents the interquartile range (IQR), while the whiskers extend to the minimum and maximum values within a certain range. Box plots are useful for comparing the central tendency, spread, and skewness of different groups or categories in the data.

In [20]:
```
data['TotalHours'].describe()
```

Out[20]:
```
count       3395.000000
mean         952.691605
std        17172.196750
min            3.000000
25%          300.000000
50%          500.000000
75%          800.000000
max      1000000.000000
Name: TotalHours, dtype: float64
```

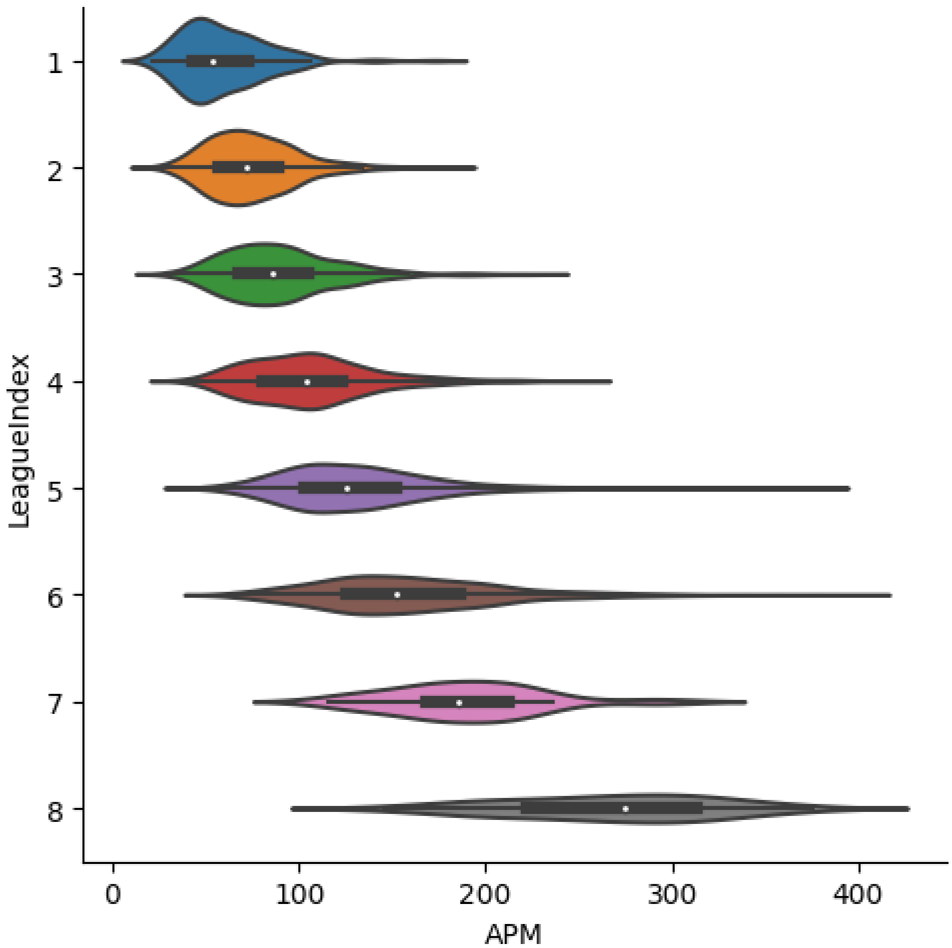
Age, HoursPerWeek and APM distributions have some outliers that can be seen in the above boxplots. However, the TotalHours has some extreme outliers that we need to address! We will do further outlier analysis for other features, and will address them later collectively to make our model robust to outliers.

In [21]:
```
# distribution of age and hours they play
fig,ax = plt.subplots(2,2, figsize=(20,10))
plt.suptitle("Distribution of age and hours they play", fontsize=20)
sns.stripplot(data=data, x="Age", y="LeagueIndex", hue="LeagueIndex", palet
ax[0][0].set_title('Age')
sns.stripplot(data=data, x="HoursPerWeek", y="LeagueIndex", hue="LeagueInde
ax[0][1].set_title('HoursPerWeek')
sns.stripplot(data=data, x="TotalHours", y="LeagueIndex", hue="LeagueIndex"
ax[1][0].set_title('TotalHours')
sns.stripplot(data=data, x="APM", y="LeagueIndex", hue="LeagueIndex", palet
ax[1][1].set_title('APM')
plt.show()
```



A strip plot is a type of categorical scatter plot that displays the distribution of a continuous variable across categories or groups. It represents each data point as a small horizontal or vertical marker along the categorical axis. Strip plots are useful for visualizing the distribution and density of data points within each category, allowing for easy comparison between groups. They can reveal patterns, clusters, and outliers in the data and are particularly effective for small to moderate-sized datasets.

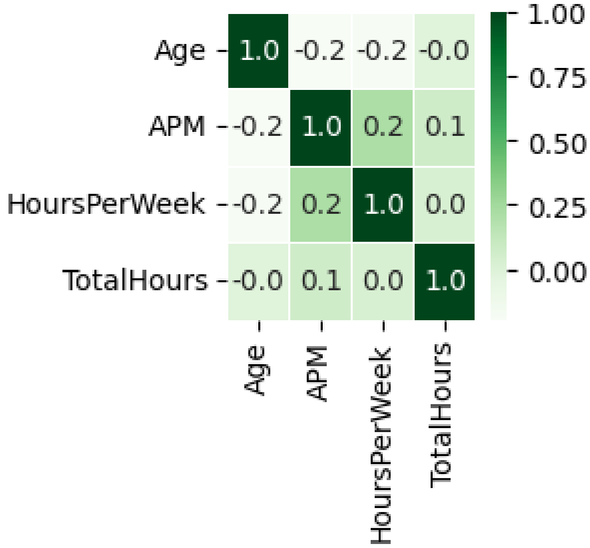
In [22]:
```
sns.catplot(data=data, x="APM", y="LeagueIndex", kind="violin", orient='h')
plt.show()
```



APM seems to be an important indicator to the LeagueIndex!

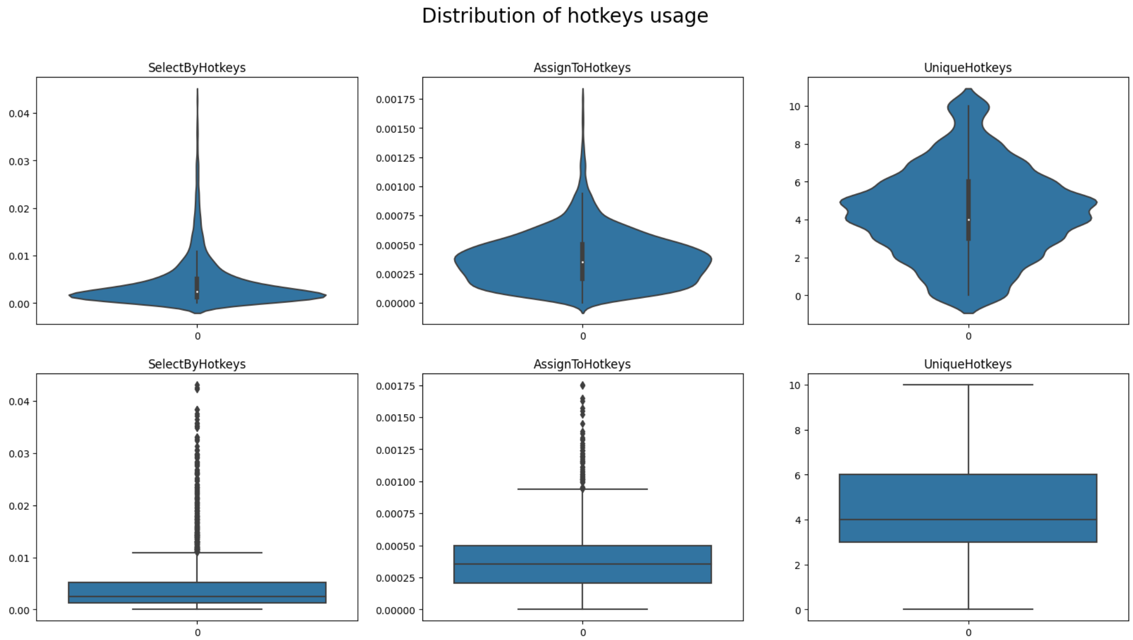
In [23]:

```
# Check for multicollinearity using correlation plot
f,ax = plt.subplots(figsize=(2,2))
sns.heatmap(data[['Age','APM','HoursPerWeek','TotalHours']].corr(), annot=T
plt.show()
```



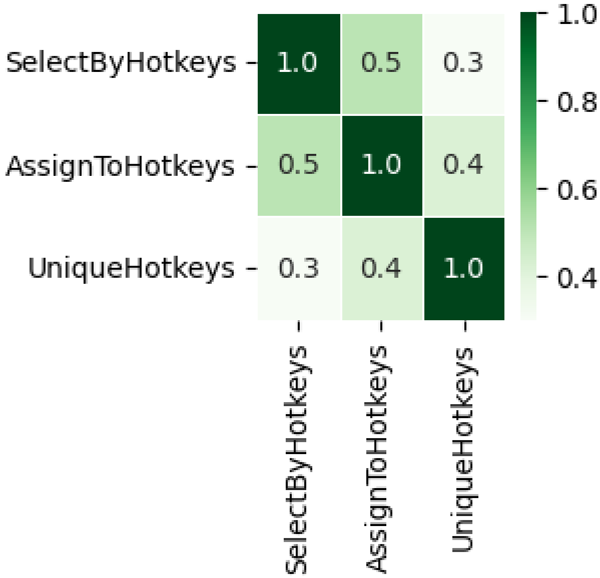
In [24]:

```
# distribution of hotkeys usage
fig,ax = plt.subplots(2,3, figsize=(20,10))
plt.suptitle("Distribution of hotkeys usage", fontsize=20)
sns.violinplot(data['SelectByHotkeys'], ax = ax[0][0])
ax[0][0].set_title('SelectByHotkeys')
sns.violinplot(data['AssignToHotkeys'], ax = ax[0][1])
ax[0][1].set_title('AssignToHotkeys')
sns.violinplot(data['UniqueHotkeys'], ax = ax[0][2])
ax[0][2].set_title('UniqueHotkeys')
sns.boxplot(data['SelectByHotkeys'], ax = ax[1][0])
ax[1][0].set_title('SelectByHotkeys')
sns.boxplot(data['AssignToHotkeys'], ax = ax[1][1])
ax[1][1].set_title('AssignToHotkeys')
sns.boxplot(data['UniqueHotkeys'], ax = ax[1][2])
ax[1][2].set_title('UniqueHotkeys')
plt.show()
```

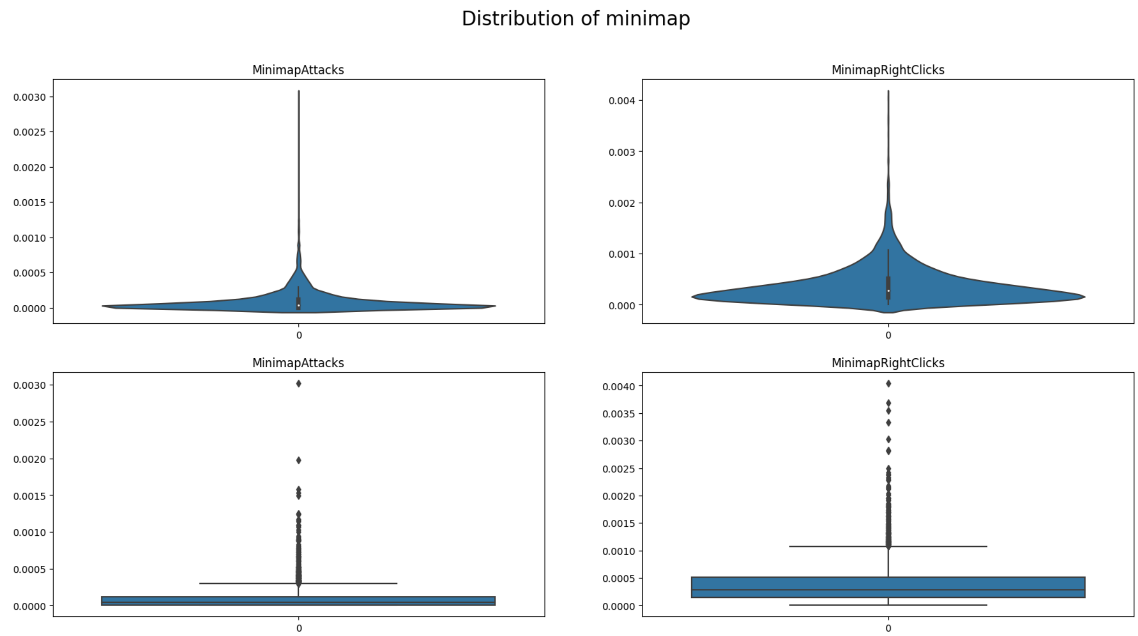


In [25]:

```
# Check for multicollinearity using correlation plot
f,ax = plt.subplots(figsize=(2,2))
sns.heatmap(data[['SelectByHotkeys','AssignToHotkeys','UniqueHotkeys']].cor
plt.show()
```

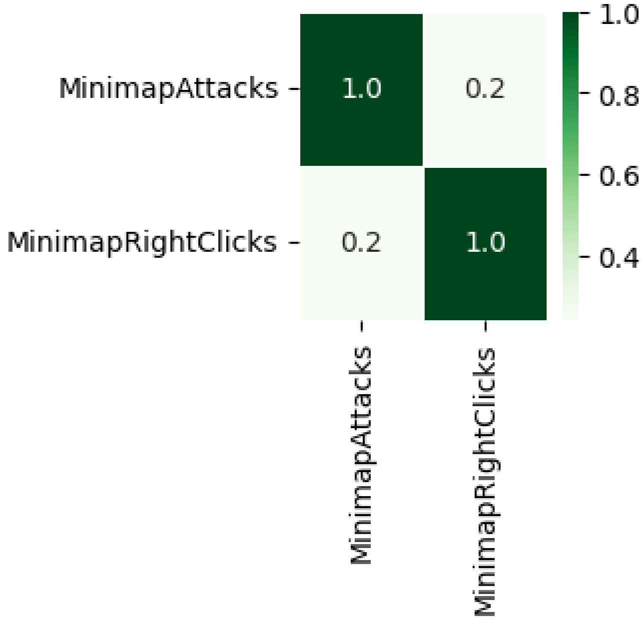


In [26]:
```
# distribution of minimap habits of the player
fig,ax = plt.subplots(2,2, figsize=(20,10))
plt.suptitle("Distribution of minimap", fontsize=20)
sns.violinplot(data['MinimapAttacks'], ax = ax[0][0])
ax[0][0].set_title('MinimapAttacks')
sns.violinplot(data['MinimapRightClicks'], ax = ax[0][1])
ax[0][1].set_title('MinimapRightClicks')
sns.boxplot(data['MinimapAttacks'], ax = ax[1][0])
ax[1][0].set_title('MinimapAttacks')
sns.boxplot(data['MinimapRightClicks'], ax = ax[1][1])
ax[1][1].set_title('MinimapRightClicks')
plt.show()
```

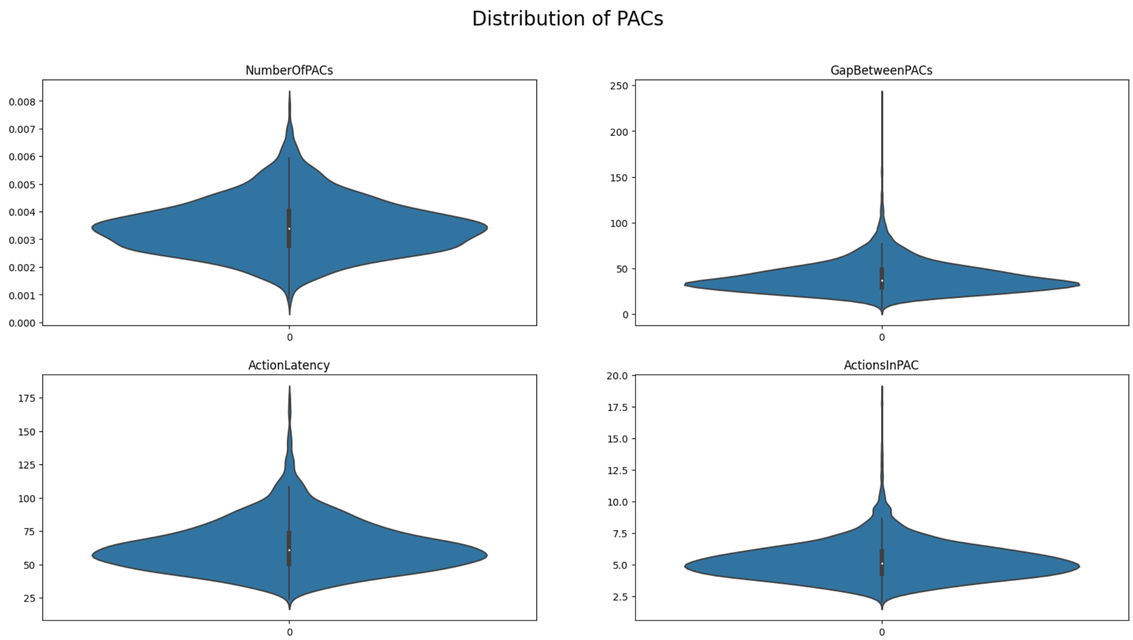


Distribution is a little skewed, so we might need to address it!

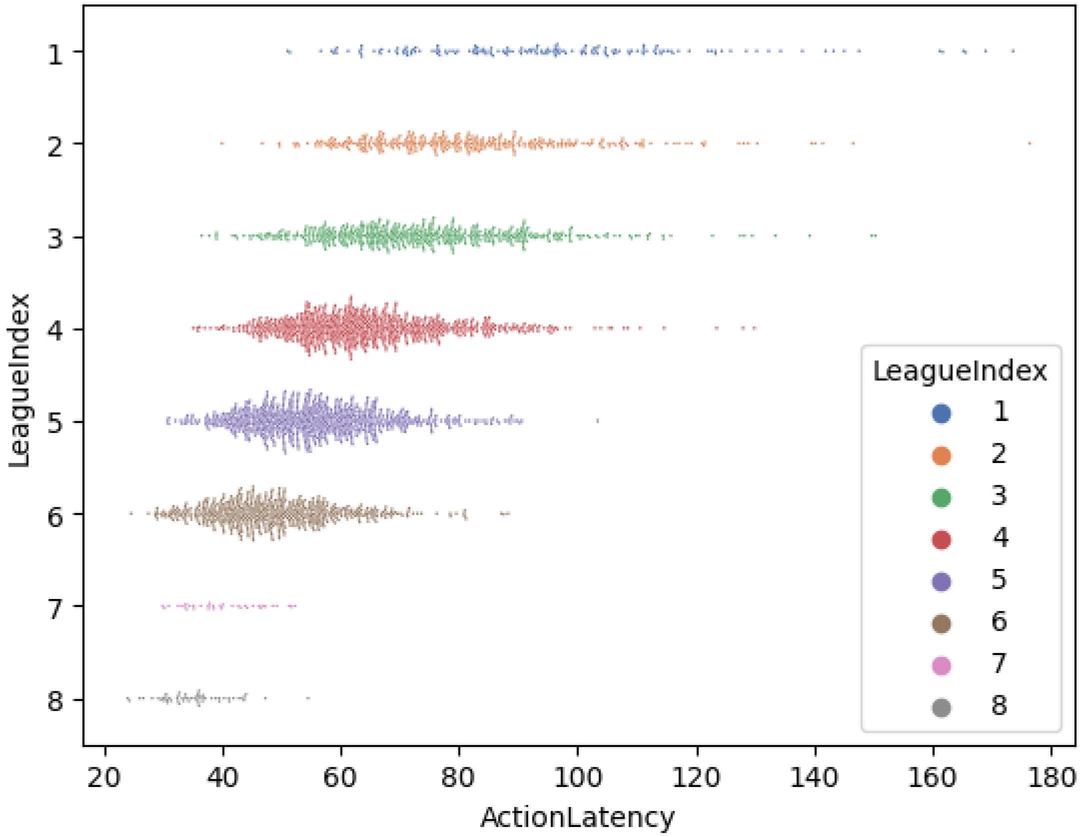
In [27]:
```
# Check for multicollinearity using correlation plot
f,ax = plt.subplots(figsize=(2,2))
sns.heatmap(data[['MinimapAttacks','MinimapRightClicks']].corr(), annot=Tru
plt.show()
```



In [28]:
```
# distribution of PAC - Perception Action Cycle
fig,ax = plt.subplots(2,2, figsize=(20,10))
plt.suptitle("Distribution of PACs", fontsize=20)
sns.violinplot(data['NumberOfPACs'], ax = ax[0][0])
ax[0][0].set_title('NumberOfPACs')
sns.violinplot(data['GapBetweenPACs'], ax = ax[0][1])
ax[0][1].set_title('GapBetweenPACs')
sns.violinplot(data['ActionLatency'], ax = ax[1][0])
ax[1][0].set_title('ActionLatency')
sns.violinplot(data['ActionsInPAC'], ax = ax[1][1])
ax[1][1].set_title('ActionsInPAC')
plt.show()
```

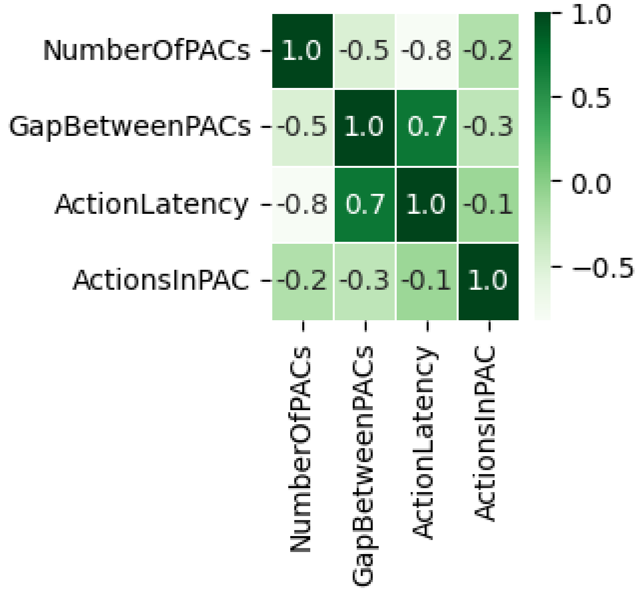


In [29]:
```
sns.swarmplot(data=data, x="ActionLatency", y="LeagueIndex", hue="LeagueInd
plt.show()
```



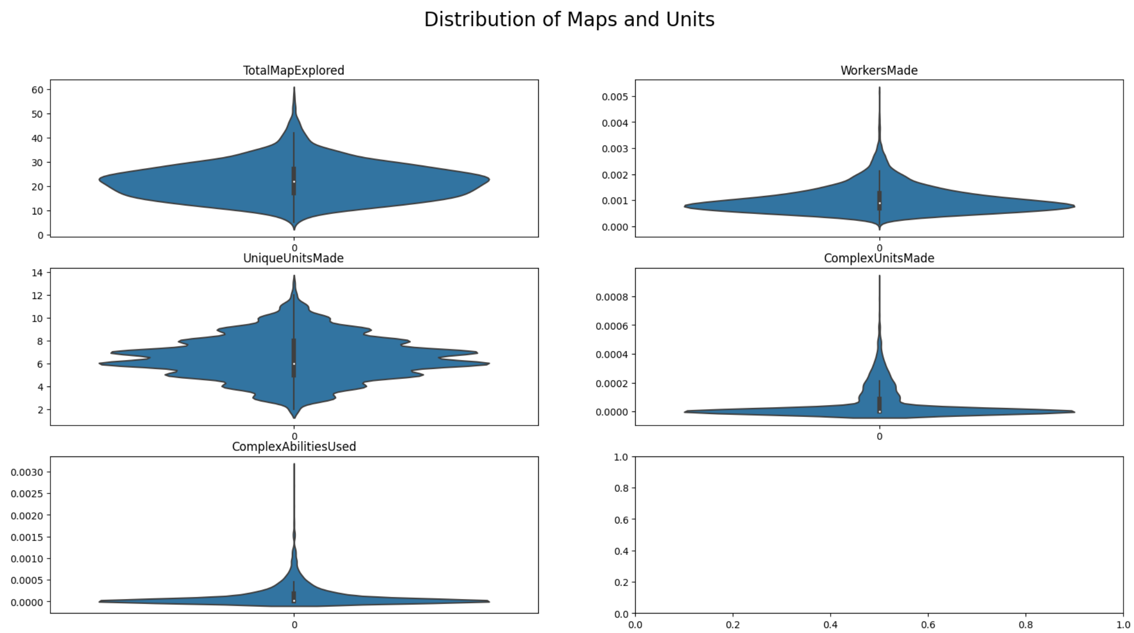
ActionLatency also seems like a pretty important factor for LeagueIndex. Need to have a very low latency or fast reflexes to git gud at the game!

In [30]:
```
# Check for multicollinearity using correlation plot
f,ax = plt.subplots(figsize=(2,2))
sns.heatmap(data[['NumberOfPACs','GapBetweenPACs', 'ActionLatency', 'Actior
plt.show()
```

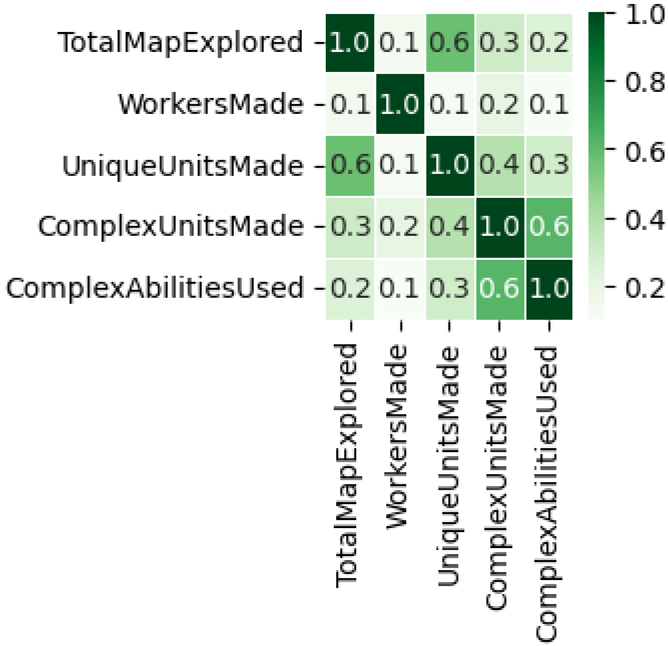


In [31]:

```
# distribution of map, and units
fig,ax = plt.subplots(3,2, figsize=(20,10))
plt.suptitle("Distribution of Maps and Units", fontsize=20)
sns.violinplot(data['TotalMapExplored'], ax = ax[0][0])
ax[0][0].set_title('TotalMapExplored')
sns.violinplot(data['WorkersMade'], ax = ax[0][1])
ax[0][1].set_title('WorkersMade')
sns.violinplot(data['UniqueUnitsMade'], ax = ax[1][0])
ax[1][0].set_title('UniqueUnitsMade')
sns.violinplot(data['ComplexUnitsMade'], ax = ax[1][1])
ax[1][1].set_title('ComplexUnitsMade')
sns.violinplot(data['ComplexAbilitiesUsed'], ax = ax[2][0])
ax[2][0].set_title('ComplexAbilitiesUsed')
plt.show()
```



In [32]:
```
# Check for multicollinearity using correlation plot
f,ax = plt.subplots(figsize=(2,2))
sns.heatmap(data[['TotalMapExplored','WorkersMade', 'UniqueUnitsMade', 'Cor
plt.show()
```



Handling Outliers

Outliers can be handled in multiple ways:

1. Identify and Remove: One common approach is to identify outliers using statistical methods such as the Z-score or interquartile range (IQR), and then remove those data points from the dataset. However, this approach should be used with caution, as removing outliers may also remove valuable information from the dataset.
2. Transform: Instead of removing outliers, we can apply transformations to the data to reduce the impact of extreme values. For example, we can apply logarithmic, square root, or reciprocal transformations to make the data distribution more symmetric.
3. Winsorize: We can apply winsorization, which replaces outliers with values at a specific percentile, effectively limiting their influence on the analysis.
4. Binning: Another approach is to divide the data into bins or categories and treat the outliers separately. This can be useful when the presence of outliers significantly affects the analysis or when there is a clear distinction between outliers and the rest of the data. We can assign outliers to a separate bin or category to isolate their impact on the analysis.
5. Imputation: If outliers are present in missing data, we can use imputation techniques to estimate the missing values. Imputation methods, such as mean, median, or regression-based imputation, can help replace outliers with plausible values based on the rest of the dataset. However, it could significantly skew the imputed values.
6. Model-based Approaches: Some outlier detection algorithms, such as isolation forests or one-class SVM, use machine learning techniques to identify outliers based on the underlying patterns in the data. These models can be trained to classify observations as

outliers or non-outliers, providing a more automated way of detecting and handling outliers.

Generally, for normally distributed data, we use Standard Deviation method and for other data distributions, we use the Interquartile range method.

Based on the above plots, we can classify the columns as:

1. Normally Distributed: APM, AssignToHotkeys, UniqueHotkeys, NumberOfPACs, GapBetweenPACs, ActionLatency, ActionsInPAC, TotalMapExplored
2. Other Distribution: HoursPerWeek, SelectByHotkeys, MinimapAttacks, MinimapRightClicks, WorkersMade, ComplexUnitsMade, UniqueUnitsMade, ComplexAbilitiesUsed

There are some tests like Kolmogorov–Smirnov test and the Shapiro–Wilk test that check for normality of data. We can use those tests and classify their distibution and handle outliers accordingly, but for now we will just trust our violinplot distribution to check the normality of data.

In [33]:
```python
def winsorize_outliers(df, column_name, k=1.5):
    """
    Function that inputs a dataframe and a column name (works with any data
    using IQR method and uses the winsorization method to impute them.
    """
    # Calculate the IQR
    q1 = df[column_name].quantile(0.25)
    q3 = df[column_name].quantile(0.75)
    iqr_value = iqr(df[column_name])

    # Define the outlier thresholds
    lower_threshold = q1 - k * iqr_value
    upper_threshold = q3 + k * iqr_value

    # Create a copy of the column for winsorization
    column_copy = df[column_name].copy()

    # Winsorize the outliers
    column_copy[column_copy < lower_threshold] = lower_threshold
    column_copy[column_copy > upper_threshold] = upper_threshold

    # Replace the original column with the winsorized values
    df[column_name] = column_copy

    return df
```

In [34]:
```
def winsorize_outliers_zscore(df, column_name, threshold=3):
    """
    Function that inputs a dataframe and a column name (preferable normally
    using Z-score and uses the winsorization method to impute them.
    """
    # Calculate the z-scores for the column
    z_scores = zscore(df[column_name])

    # Identify the outliers using the specified threshold
    outliers = np.abs(z_scores) > threshold

    # Create a copy of the column for winsorization
    column_copy = df[column_name].copy()

    # Winsorize the outliers
    column_copy[outliers] = np.sign(column_copy[outliers]) * threshold * np

    # Replace the original column with the winsorized values
    df[column_name] = column_copy

    return df
```

In [35]:
```
normally_dist_cols = ['ActionsInPAC',]
unnormal_dist_cols = ['HoursPerWeek', 'SelectByHotkeys', 'MinimapAttacks',
```

In [36]:
```
data.describe()
```

Out[36]:

	LeagueIndex	Age	HoursPerWeek	TotalHours	APM	SelectByHotk
count	3395.000000	3395.000000	3395.000000	3395.000000	3395.000000	3395.00(
mean	4.184094	21.647904	15.846244	952.691605	117.046947	0.004
std	1.517327	4.172119	11.874264	17172.196750	51.945291	0.005
min	1.000000	16.000000	0.000000	3.000000	22.059600	0.00(
25%	3.000000	19.000000	8.000000	300.000000	79.900200	0.001
50%	4.000000	21.000000	12.000000	500.000000	108.010200	0.002
75%	5.000000	24.000000	20.000000	800.000000	142.790400	0.005
max	8.000000	44.000000	168.000000	1000000.000000	389.831400	0.043

In [37]:
```
# normally distributed data - Standard Deviation method; other data distrib
for col in normally_dist_cols:
    data = winsorize_outliers_zscore(data, col, threshold=4)
for col in unnormal_dist_cols:
    data = winsorize_outliers(data, col, k=2)
```

In [38]:
```
data.describe()
```

Out[38]:

	LeagueIndex	Age	HoursPerWeek	TotalHours	APM	SelectByHotk
count	3395.000000	3395.000000	3395.000000	3395.000000	3395.000000	3395.00(
mean	4.184094	21.647904	15.446834	952.691605	117.046947	0.00:
std	1.517327	4.172119	10.083288	17172.196750	51.945291	0.00:
min	1.000000	16.000000	0.000000	3.000000	22.059600	0.00(
25%	3.000000	19.000000	8.000000	300.000000	79.900200	0.00'
50%	4.000000	21.000000	12.000000	500.000000	108.010200	0.00:
75%	5.000000	24.000000	20.000000	800.000000	142.790400	0.00t
max	8.000000	44.000000	44.000000	1000000.000000	389.831400	0.01:

Model Training

SMOTE (Synthetic Minority Over-sampling Technique)

As I promised in the earlier section, we will handle the class imbalance using SMOTE.

SMOTE is a popular technique used for handling imbalanced datasets. It addresses the issue of imbalanced class distribution by generating synthetic examples of the minority class to balance the dataset. SMOTE works by creating synthetic samples in the feature space of the minority class by interpolating between existing minority class samples.

Here's how SMOTE works in a nutshell:

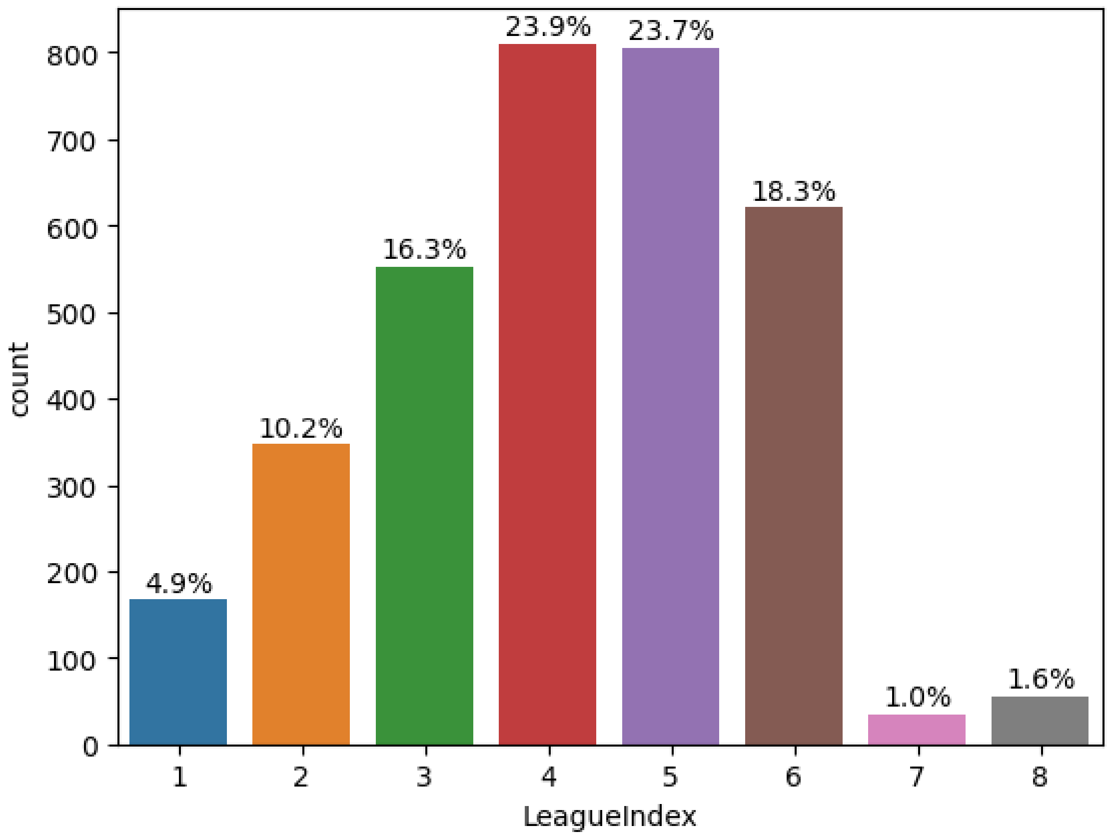
1. Identify the minority class: Determine the class in the dataset that is less represented and considered the minority class.
2. Select a minority class sample: Randomly choose a sample from the minority class.
3. Find its k nearest neighbors: Measure the distances between the chosen sample and its k nearest neighbors. The value of k is a user-defined parameter.
4. Generate synthetic samples: Randomly select one of the k nearest neighbors and interpolate between the chosen sample and the selected neighbor to create a new synthetic sample. Repeat this process to generate the desired number of synthetic samples.
5. Repeat the process: Repeat steps 2-4 until the desired balance between the minority and majority class is achieved.

By creating synthetic samples, SMOTE helps to increase the representation of the minority class, which can improve the model's ability to learn and generalize patterns from the data.

In [39]:
```
import imblearn
from imblearn.over_sampling import SMOTE
```

In [40]:
```
X = data.drop(['LeagueIndex'], axis=1)
y = data['LeagueIndex']
```

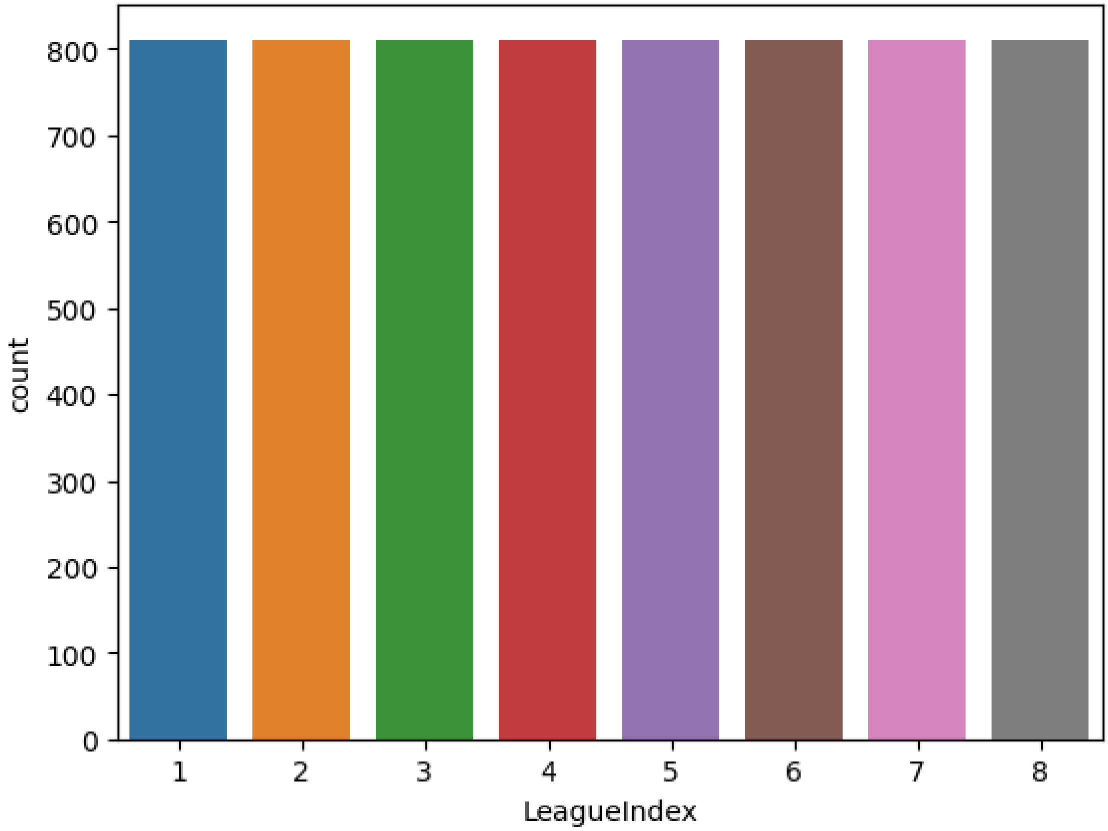
In [41]:
```
# distribution of the target variable to check if it is skewed or balanced
ax = sns.countplot(x='LeagueIndex', data=data)
for i, patch in enumerate(ax.patches):
    percentage = 100 * patch.get_height()/len(data)
    x_ = patch.get_x() + patch.get_width()/2
    y_ = patch.get_height()+10
    ax.annotate('{:.1f}%'.format(percentage), (x_, y_), ha='center')
plt.show()
```



In [42]:
```
smt = SMOTE(random_state=42)
X_resampled, y_resampled = smt.fit_resample(X, y)
```

In [43]:
```
resampled_data = pd.concat([pd.DataFrame(X_resampled, columns=X.columns),
                            pd.Series(y_resampled, name='LeagueIndex')], ax
```

In [44]:
```
sns.countplot(x='LeagueIndex', data=resampled_data)
plt.show()
```



Now the target variable distribution seems balanced across all classes, now we can proceed to training the model.

Pycaret - Automate some of the classification model fitting tasks

In [45]:
```
from pycaret.classification import *
s = setup(resampled_data, target = 'LeagueIndex', session_id = 42)
```

	Description	Value
0	Session id	42
1	Target	LeagueIndex
2	Target type	Multiclass
3	Target mapping	1: 0, 2: 1, 3: 2, 4: 3, 5: 4, 6: 5, 7: 6, 8: 7
4	Original data shape	(6488, 19)
5	Transformed data shape	(6488, 19)
6	Transformed train set shape	(4541, 19)
7	Transformed test set shape	(1947, 19)
8	Numeric features	18
9	Preprocess	True
10	Imputation type	simple
11	Numeric imputation	mean
12	Categorical imputation	mode
13	Fold Generator	StratifiedKFold
14	Fold Number	10
15	CPU Jobs	-1
16	Use GPU	False
17	Log Experiment	False
18	Experiment Name	clf-default-name
19	USI	789e

In [46]:
```
# fit diff models on the data
best = compare_models()
```

	Model	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC	TT (Sec)
et	Extra Trees Classifier	0.7012	0.9397	0.7012	0.6876	0.6915	0.6585	0.6595	0.3750
rf	Random Forest Classifier	0.6827	0.9347	0.6827	0.6691	0.6732	0.6373	0.6382	0.3610
lightgbm	Light Gradient Boosting Machine	0.6802	0.9318	0.6802	0.6750	0.6758	0.6346	0.6351	0.3510
xgboost	Extreme Gradient Boosting	0.6739	0.9302	0.6739	0.6670	0.6685	0.6273	0.6278	0.3420
gbc	Gradient Boosting Classifier	0.6203	0.9138	0.6203	0.6118	0.6144	0.5661	0.5667	0.3780
knn	K Neighbors Classifier	0.5924	0.8571	0.5924	0.5687	0.5741	0.5341	0.5362	0.4320
dt	Decision Tree Classifier	0.5807	0.7604	0.5807	0.5751	0.5767	0.5208	0.5212	0.3230
nb	Naive Bayes	0.5069	0.8703	0.5069	0.4981	0.4897	0.4365	0.4409	0.3160
lr	Logistic Regression	0.4986	0.8754	0.4986	0.4890	0.4915	0.4269	0.4275	0.6540
lda	Linear Discriminant Analysis	0.4889	0.8719	0.4889	0.4949	0.4895	0.4159	0.4165	0.3160
ridge	Ridge Classifier	0.4039	0.0000	0.4039	0.3570	0.3387	0.3187	0.3295	0.3170
ada	Ada Boost Classifier	0.3182	0.6972	0.3182	0.2316	0.2161	0.2207	0.2720	0.3200
svm	SVM - Linear Kernel	0.3004	0.0000	0.3004	0.2523	0.2255	0.2002	0.2532	0.3110
qda	Quadratic Discriminant Analysis	0.1251	0.0000	0.1251	0.0156	0.0278	0.0000	0.0000	0.3160
dummy	Dummy Classifier	0.1240	0.5000	0.1240	0.0154	0.0274	0.0000	0.0000	0.3100

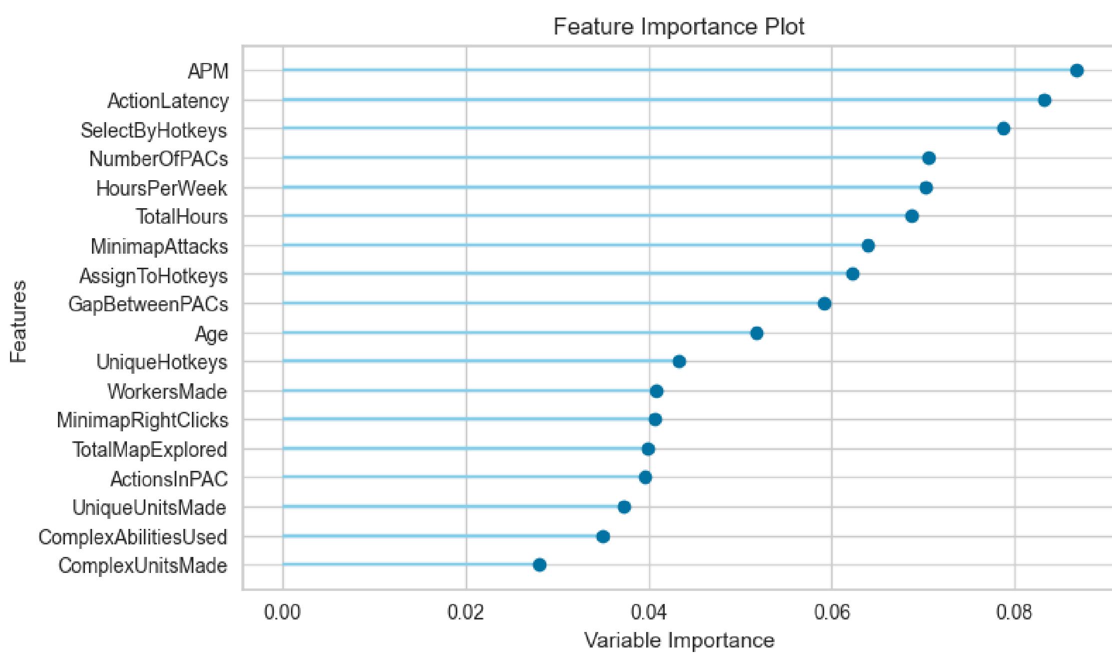
In [47]:
```
# view best model params
print(best)
```

```
ExtraTreesClassifier(bootstrap=False, ccp_alpha=0.0, class_weight=None,
                     criterion='gini', max_depth=None, max_features='aut
o',
                     max_leaf_nodes=None, max_samples=None,
                     min_impurity_decrease=0.0, min_samples_leaf=1,
                     min_samples_split=2, min_weight_fraction_leaf=0.0,
                     n_estimators=100, n_jobs=-1, oob_score=False,
                     random_state=42, verbose=0, warm_start=False)
```

In [48]:
```
# visualize performance of the model
evaluate_model(best)
```

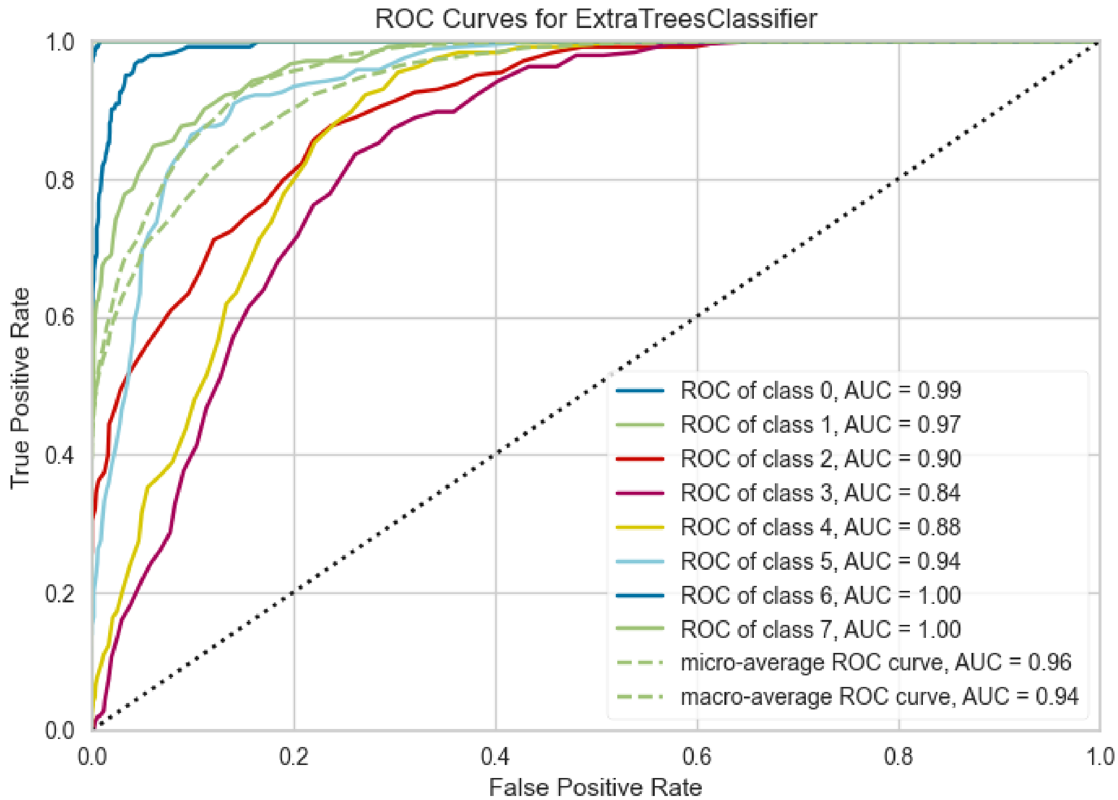
```
interactive(children=(ToggleButtons(description='Plot Type:', icons=
('',), options=(('Pipeline Plot', 'pipelin…
```

In [49]:
```
# feature importance curve
plot_model(best, plot = 'feature_all')
```



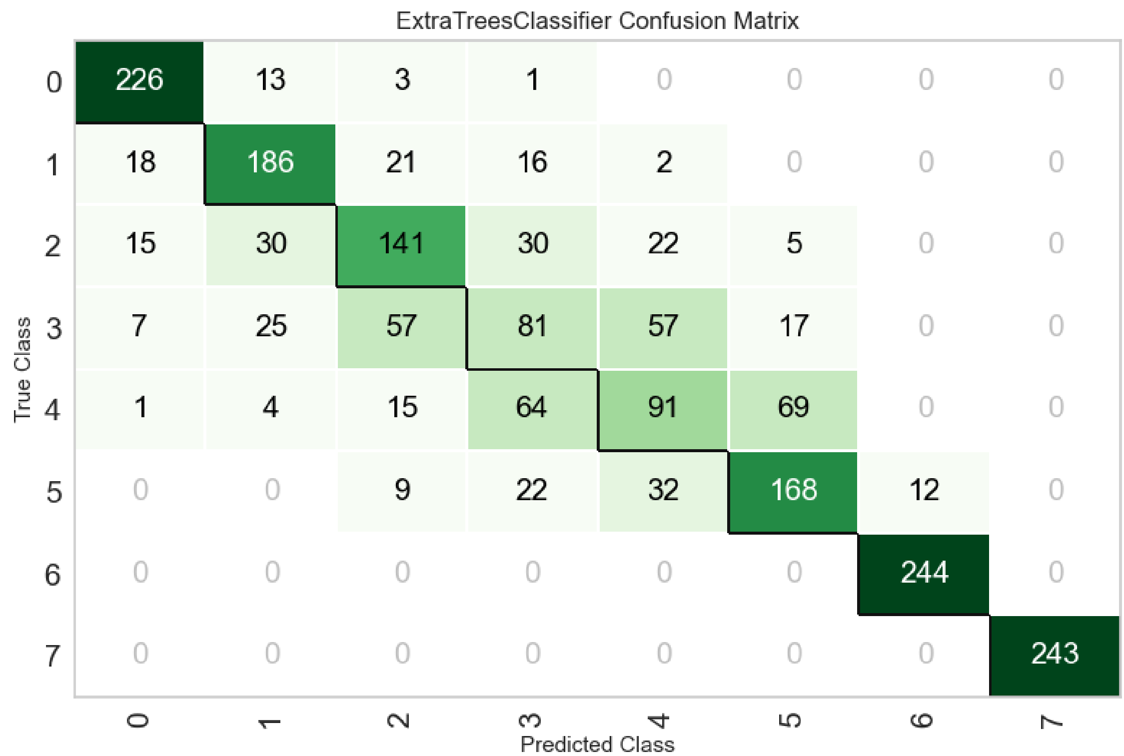
Feature Importance plot is a visual representation that shows the importance of different features in a machine learning model. It ranks the features based on their contribution to the model's predictive performance. The plot helps identify the most influential features and their relative importance, aiding in feature selection and understanding the underlying relationships in the data. It is a valuable tool for gaining insights into which features have the most impact on the model's predictions.

In [50]:
```
# auc curve
plot_model(best, plot = 'auc')
```



The ROC curve is a graphical representation of the performance of a binary classification model. It plots the true positive rate against the false positive rate across different classification thresholds. The curve helps assess the model's ability to distinguish between positive and negative classes, with a higher curve indicating better performance. The area under the ROC curve (AUC-ROC) summarizes the overall discriminatory power of the model, with an AUC-ROC of 1.0 indicating a perfect classifier. The ROC curve is useful for evaluating model performance, especially in imbalanced datasets or when the cost of false positives and false negatives varies. Above figure shows the ROC for different LeagueIndex.

In [51]:
```
# confusion matrix for the classification
plot_model(best, plot = 'confusion_matrix')
```



In [52]:
```
# make predictions using the model along with confidence score
predict_model(best)
```

	Model	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC
0	Extra Trees Classifier	0.7088	0.9414	0.7088	0.6957	0.7012	0.6672	0.6676

Out[52]:

	Age	HoursPerWeek	TotalHours	APM	SelectByHotkeys	AssignToHotkeys
5366	24.075802	20	1000	212.260452	0.005947	0.001314
463	19.000000	10	200	133.910995	0.004849	0.000351
1859	21.000000	12	200	62.004002	0.001691	0.000302
3929	19.602289	9	20	48.781727	0.001886	0.000454
4983	23.000000	44	2451	175.839615	0.009729	0.000848
...	...	...	...	...	...	...
4843	19.898554	25	546	150.810791	0.005528	0.000238
279	17.000000	20	500	61.888199	0.001603	0.000190
4651	18.886681	19	747	98.468765	0.000604	0.000122
2930	20.000000	10	200	68.139000	0.001621	0.000101
4699	18.601349	12	300	137.002762	0.004114	0.000525

1947 rows × 21 columns

In [53]:
```
# save the best model pipeline pkl file
save_model(best, 'best_pipeline')
```

```
Transformation Pipeline and Model Successfully Saved
```

Out[53]:
```
(Pipeline(memory=FastMemory(location=C:\Users\shrea\AppData\Local\Temp\jo
blib),
          steps=[('label_encoding',
                  TransformerWrapperWithInverse(exclude=None, include=Non
e,
                                                 transformer=LabelEncoder
())),
                 ('numerical_imputer',
                  TransformerWrapper(exclude=None,
                                     include=['Age', 'HoursPerWeek',
                                              'TotalHours', 'APM',
                                              'SelectByHotkeys',
                                              'AssignToHotkeys', 'UniqueH
otkeys',
                                              'Minimap...
                  ExtraTreesClassifier(bootstrap=False, ccp_alpha=0.0,
                                       class_weight=None, criterion='gin
i',
                                       max_depth=None, max_features='aut
o',
                                       max_leaf_nodes=None, max_samples=N
one,
                                       min_impurity_decrease=0.0,
                                       min_samples_leaf=1, min_samples_sp
lit=2,
                                       min_weight_fraction_leaf=0.0,
                                       n_estimators=100, n_jobs=-1,
                                       oob_score=False, random_state=42,
                                       verbose=0, warm_start=False))],
          verbose=False),
 'best_pipeline.pkl')
```

Hyperparameter Tuning the best model

In [54]:
```
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.model_selection import GridSearchCV, train_test_split

X = resampled_data.drop(['LeagueIndex'], axis=1)
y = resampled_data['LeagueIndex'] # 1-8

# train test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, r

# Define the parameter grid for Grid Search
param_grid = {
    'n_estimators': [100, 300],
    'max_depth': [None, 5, 10],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['auto', 'sqrt']
}

model = ExtraTreesClassifier()

# Perform Grid Search with cross-validation
grid_search = GridSearchCV(model, param_grid, cv=5)
grid_search.fit(X_train, y_train)

best_params = grid_search.best_params_
best_score = grid_search.best_score_

print("Best Parameters:", best_params)
print("Best Score:", best_score)
```

```
Best Parameters: {'max_depth': None, 'max_features': 'auto', 'min_samples
_leaf': 1, 'min_samples_split': 2, 'n_estimators': 300}
Best Score: 0.6900947766280963
```

In [55]:
```
# Evaluate the best model on the test set
best_model = grid_search.best_estimator_
test_score = best_model.score(X_test, y_test)
print("Test Score:", test_score)
```

```
Test Score: 0.7009864364981504
```

70% Accuracy on the test set across 8 classes, that is great!

Neural Net

Playing around with neural net to see how it compares to our statistical models, and if it is able to reach to a comparable accuracy in a small dataset!

In [56]:
```
# Spliting target variable and independent variables
X = resampled_data.drop(['LeagueIndex'], axis=1)
y = resampled_data['LeagueIndex'] # 1-8
# Label encoding target variable to make it 0-7
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
y = le.fit_transform(y)
```

In [57]:
```
# Splitting the data into training set and testset
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size = 0.25,
```

In [58]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F

X_train = torch.FloatTensor(X_train.values)
X_test = torch.FloatTensor(X_test.values)
y_train = torch.LongTensor(y_train)
y_test = torch.LongTensor(y_test)
```

In [59]:

```
# Creating the architecture of the ANN
class NeuralNet(nn.Module):
    def __init__(self, input_features=18, hidden_dim=[20, 10], output_featu
                 dropout_prob=0.5, activation='relu', use_normalization=Fal
        super(NeuralNet, self).__init__()
        layers = []
        in_features = input_features

        if use_normalization:
            layers.append(nn.BatchNorm1d(input_features))

        for hidden_dim in hidden_dim:
            layers.append(nn.Linear(in_features, hidden_dim))

            if use_normalization:
                layers.append(nn.BatchNorm1d(hidden_dim))

            if activation == 'relu':
                layers.append(nn.ReLU())
            elif activation == 'sigmoid':
                layers.append(nn.Sigmoid())
            elif activation == 'tanh':
                layers.append(nn.Tanh())

            layers.append(nn.Dropout(p=dropout_prob))
            in_features = hidden_dim

        layers.append(nn.Linear(in_features, output_features))
        layers.append(nn.Softmax())

        def init_weights(m):
            if isinstance(m, nn.Linear):
                torch.nn.init.xavier_uniform(m.weight)
                m.bias.data.fill_(0.01)

        self.model = nn.Sequential(*layers)
        self.model.apply(init_weights)

    def forward(self, x):
        return self.model(x)
```

In [60]:
```python
# Instantiate the model
torch.manual_seed(42)
model = NeuralNet(input_features=X_train.shape[1])
model.parameters
```

Out[60]:
```
<bound method Module.parameters of NeuralNet(
  (model): Sequential(
    (0): Linear(in_features=18, out_features=20, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=20, out_features=10, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=10, out_features=8, bias=True)
    (7): Softmax(dim=None)
  )
)>
```

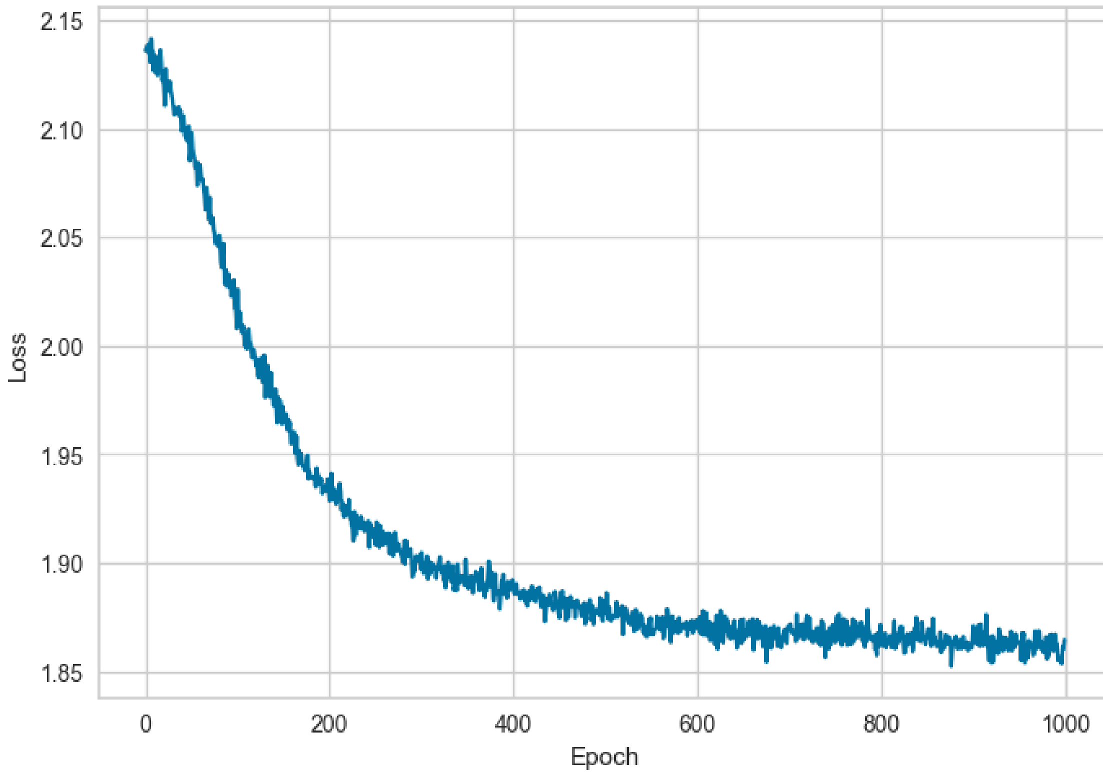
In [61]:
```python
# Backpropagation - Loss function, optimizer
loss_function = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.005)
```

In [62]:
```python
epochs = 1000
final_losses = []
for i in range(epochs):
    i+=1
    y_pred = model.forward(X_train)
    loss = loss_function(y_pred, y_train)
    final_losses.append(loss.item())
    if(i%100==0):
        print(f"Epoch {i} loss: {loss.item()}")
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

```
Epoch 100 loss: 2.007981777191162
Epoch 200 loss: 1.9316250085830688
Epoch 300 loss: 1.897342562675476
Epoch 400 loss: 1.8921637535095215
Epoch 500 loss: 1.8780194520950317
Epoch 600 loss: 1.8736435174942017
Epoch 700 loss: 1.8707680702209473
Epoch 800 loss: 1.871469259262085
Epoch 900 loss: 1.8638237714767456
Epoch 1000 loss: 1.8645381927490234
```

In [63]:

```
# Loss function
plt.plot(range(epochs), final_losses)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.show()
```



In [64]:

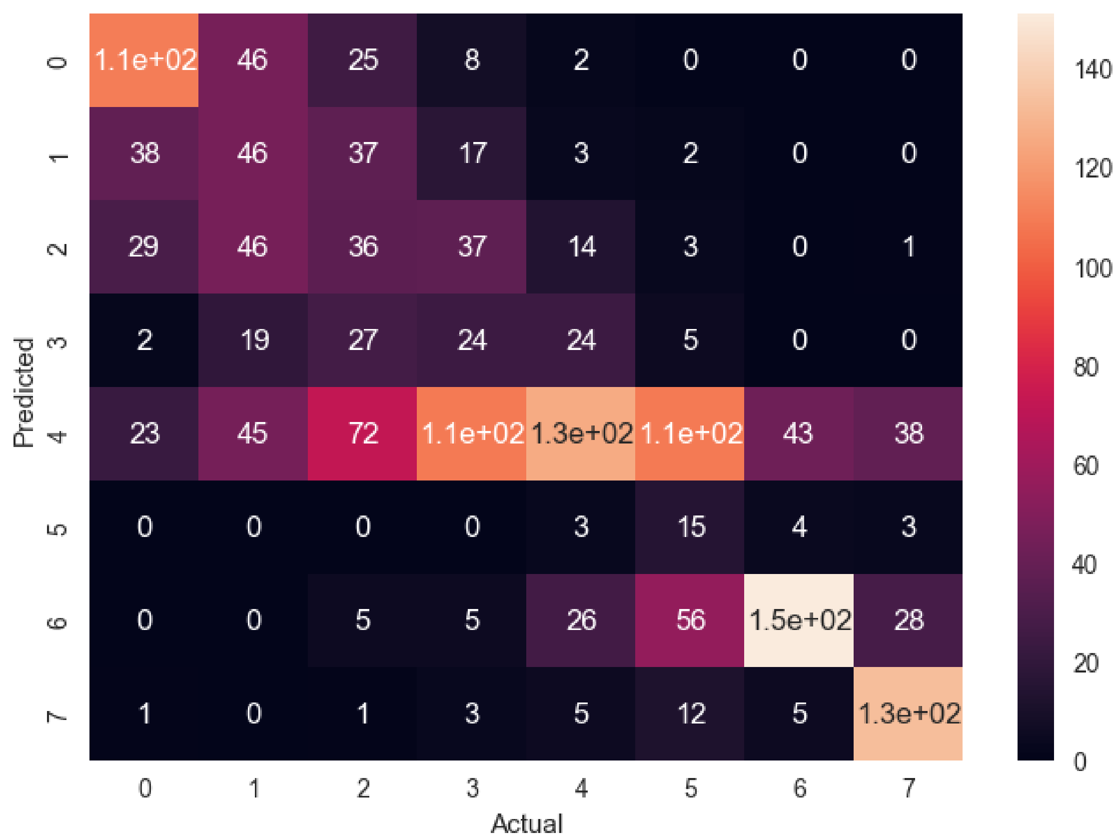
```
# Predicting on test data
y_pred = []
with torch.no_grad():
    for i, (test_data, actual) in enumerate(zip(X_test, y_test)):
        pred = model(test_data)
        predict = pred.argmax().item()
        y_pred.append(predict)
```

In [65]:

```
# Create confusion matrix function to find out sensitivity and specificity
from sklearn.metrics import confusion_matrix
def draw_cm(actual, predicted):
    cm = confusion_matrix( actual, predicted).T
    sns.heatmap(cm, annot=True )
    plt.ylabel('Predicted')
    plt.xlabel('Actual')
    plt.show()
```

In [66]:
```
from sklearn.metrics import accuracy_score
acc_nn = round( metrics.accuracy_score(y_test.tolist(), y_pred) * 100 , 2 )
print( 'Neural Net Accuracy : ', acc_nn )
draw_cm(y_test.tolist(), y_pred)
```

```
Neural Net Accuracy :  39.52
```

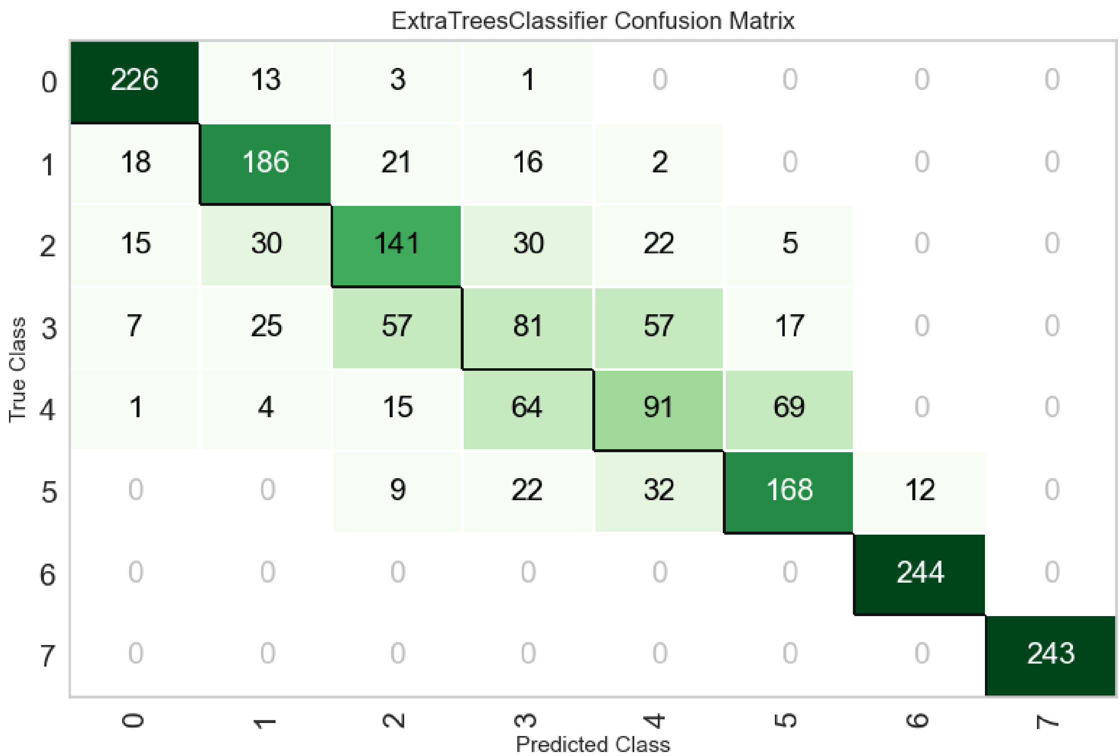


Accuracy is not as great as statistical models since the dataset size is pretty small. If we had large amount of data, then the neural net would have worked better, and even surpassed the accuracy of the statistical models.

Findings and Conclusion (for non-technical stakeholders)

In [68]:

```
# confusion matrix for the classification
plot_model(best, plot = 'confusion_matrix')
```



Summary of the steps I followed to create the model:

1. Exploratory Data Analysis (EDA) to understand the data and problem statement thouroughly.

2. Data Visualizations to capture essence out of the data and understand the correlation and causation within the data. I found that some columns were skewed and some had outliers. (more details about this below)

3. Data Preprocessing:
   a. Handle missing values: Data had some missing values for players with LeagueIndex=8 (Age, TotalHours and HoursPerWeek to be precise).
   b. Detect and handle outliers: Some major outliers in the data, which migh be due to minor errors in data collection. Specifically, the columns ActionsInPAC, HoursPerWeek, SelectByHotkeys, MinimapAttacks, MinimapRightClicks, ComplexAbilitiesUsed had relatively major outliers that needed to be addressed, so I handled them by capping the upper value to a threshold, which is computed mathematically for each column. Setting the cap worked out pretty well and improved the accuracy by a decent margin (12% to be precise).
   c. Split the data: Divide the dataset into training (75%) and testing (25%) sets. Training set is used for training the model and adjusting the parameters of the model. The test set is

held out and kept separate from the training set and is used to evaluate the performance of our trained model. This ensures that the model is learning and is able to generalize well on new unseen data outside the training set.

4. Handling skewness of LeagueIndex: Data was biased towards the most common LeagueIndexes, but had very little datapoints for LeagueIndex that were rare, like 7 and 8. This is a major bottleneck in training the model so I addressed it augmenting synthetic data for the minority classes to balance out the distribution. This worked out pretty well, and improved the accuracy by around 30%.

5. Model Training: I tried out around 10-12 different statistical models to compare how they were faring against each other, and did a detailed analysis on the best model. I also compared it to a very simple neural network to evaluate and compare the performance, but the neural network did not work well since there are very few data points for a neural net, which is very data intensive and hungry.

6. Model Improvement and Tuning: Squeezed out performance from the model by carefully tuning its parameters on training data.

The above figure (confusion matrix) summarizes the performance of our model. A confusion matrix is a table that helps us understand the performance of a classification model. In our case, we have a classification problem with 8 different classes. Along X-axis, we have the predicted class and along Y-axis, we have the actual class or the ground truth. The confusion matrix provides us with a summary of how well our model performed in predicting the different LeagueIndex. Each cell in the matrix represents the count of predictions made by our model for a particular LeagueIndex. The diagonal of the confusion matrix (dark green boxes from the top-left to the bottom-right) represents the correct predictions made by our model (higher is better). For example, the numbers on the diagonal show how many times our model correctly predicted each LeagueIndex.

By looking at the confusion matrix, we can evaluate the model's performance. We can observe:

1. The cells on the diagonal, which represent correct predictions, show us how well the model performed for each Rank. We want these numbers to be high, indicating accurate predictions.
2. The off-diagonal cells show us the errors made by the model. For example, in the first row, the model predicted 13 instances with LeagueIndex=1 instead of LeagueIndex=0. Using the confusion matrix, we can calculate various performance metrics like accuracy, precision, recall, and F1-score, which provide a more detailed understanding of the model's performance across different ranks.

Our best model had an accuracy of 70% (and an F-1 score of 70%) across the ranks, which is quite good for a simple model trained on a small dataset. Our model gave the highest importance to APM, ActionLatency, NumberOfPACs and ShopByHotkeys features, which makes sense because a skilled player needs to have fast reflexes, very low latency between their actions, and also have shortcut keys or hotkeys set for different actions to save time. Also, they need to have a quick feedback loop on how quickly they absorb information and act upon it (Perception Action Cycle). Hence the model is performing quite well. However, the performance of the model can be improved in some ways, which have been provided in the next (hypothetical) section.

Hypothetical:

After seeing your work, your stakeholders come to you and say that they can collect more data, but want your guidance before starting. How would you advise them based on your EDA and model results?

If stakeholders want to collect more data, we can advise them based on our EDA and model results:

1. If possible, collect the all the features since many entries are missing. It would help avoid issues like '?' in Age, HoursPerWeek and TotalHours for LeagueIndex=8 in the future.

2. Based on the Feature Engineering and Feature importance, the APM, ActionLatency, Hotkeys and PAC are the most important features for predictions. There are many such similar features, if present in the data, will drastically improve the accuracy and the predictions. Some of them are:
   a. Races: The Terrans, Protoss, and Zerg are the three “races” of StarCraft II, each featuring unique units, buildings, and game mechanics. They all offer different gameplay experiences tailored to different play styles; this section will familiarize you with each race. This will definitely affect the predictions.
   b. OpponentRank: Matchmaking Algorithm tends to match players of similar skill level together, so this will definitely be an important factor.
   c. GameLength
   d. Ranked Winrate: There are 2 types of matches: Ranked and unranked. In ranked, players are more serious and tryharding to win. Hence this one would be an important indicator for the LeagueIndex.
   e. Unranked Winrate: In unranked matches, players might be trying out new strategies, and hence will have a lower winrate than ranked winrate. Hence this might be important.
   f. NumberOfGamesPlayed
   g. ResourceCollectionRate: Efficiency in gathering resources using workers
   h. ResourceSpendingRate: Utilizing resources to make armies and units

3. Address the class imbalances in the target variable or biases in the data and suggest collecting more data to address these issues. Collect more data for the minority classes since they are underrepresented and recommend collecting more data to balance the representation.

By providing guidance on data collection based on our EDA and model results, we can help stakeholders collect more data in order to improve the model's accuracy and robustness.