# analysis

May 25, 2023

## 1 Imports and Setup

[1]:
```python
import pandas as pd
import numpy as np
```

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

[3]:
```python
df
```

[3]:
```
      GameID  LeagueIndex Age HoursPerWeek TotalHours       APM  \
0         52            5  27           10       3000  143.7180
1         55            5  23           10       5000  129.2322
2         56            4  30           10        200   69.9612
3         57            3  19           20        400  107.6016
4         58            3  32           10        500  122.8908
…        …            …  ..          …          …        …
3390   10089            8   ?            ?          ?  259.6296
3391   10090            8   ?            ?          ?  314.6700
3392   10092            8   ?            ?          ?  299.4282
3393   10094            8   ?            ?          ?  375.8664
3394   10095            8   ?            ?          ?  348.3576

      SelectByHotkeys  AssignToHotkeys  UniqueHotkeys  MinimapAttacks  \
0            0.003515         0.000220              7        0.000110
1            0.003304         0.000259              4        0.000294
2            0.001101         0.000336              4        0.000294
3            0.001034         0.000213              1        0.000053
4            0.001136         0.000327              2        0.000000
…                 …               …             …              …
3390         0.020425         0.000743              9        0.000621
3391         0.028043         0.001157             10        0.000246
3392         0.028341         0.000860              7        0.000338
3393         0.036436         0.000594              5        0.000204
3394         0.029855         0.000811              4        0.000224

      MinimapRightClicks  NumberOfPACs  GapBetweenPACs  ActionLatency  \
0               0.000392      0.004849         32.6677        40.8673
```

```
1              0.000432     0.004307         32.9194        42.3454
2              0.000461     0.002926         44.6475        75.3548
3              0.000543     0.003783         29.2203        53.7352
4              0.001329     0.002368         22.6885        62.0813
…                   …            …              …               …
3390           0.000146     0.004555         18.6059        42.8342
3391           0.001083     0.004259         14.3023        36.1156
3392           0.000169     0.004439         12.4028        39.5156
3393           0.000780     0.004346         11.6910        34.8547
3394           0.001315     0.005566         20.0537        33.5142

      ActionsInPAC  TotalMapExplored  WorkersMade  UniqueUnitsMade  \
0           4.7508                28     0.001397                6
1           4.8434                22     0.001193                5
2           4.0430                22     0.000745                6
3           4.9155                19     0.000426                7
4           9.3740                15     0.001174                4
…              …                 …            …                …
3390        6.2754                46     0.000877                5
3391        7.1965                16     0.000788                4
3392        6.3979                19     0.001260                4
3393        7.9615                15     0.000613                6
3394        6.3719                27     0.001566                7

      ComplexUnitsMade  ComplexAbilitiesUsed
0             0.000000              0.000000
1             0.000000              0.000208
2             0.000000              0.000189
3             0.000000              0.000384
4             0.000000              0.000019
…                  …                    …
3390          0.000000              0.000000
3391          0.000000              0.000000
3392          0.000000              0.000000
3393          0.000000              0.000631
3394          0.000457              0.000895

[3395 rows x 20 columns]
```

## 2 Cleaning Data

[4]:

```
df_without_question_mark = df[df != '?'].dropna()
string_cols = ['TotalHours', 'HoursPerWeek', 'Age']
for s in string_cols:
    df_without_question_mark[s] = df_without_question_mark[s].astype(int)
df_without_question_mark
```

[4]:

| | GameID | LeagueIndex | Age | HoursPerWeek | TotalHours | APM | \ |
|---|---|---|---|---|---|---|---|
| 0 | 52 | 5 | 27 | 10 | 3000 | 143.7180 | |
| 1 | 55 | 5 | 23 | 10 | 5000 | 129.2322 | |
| 2 | 56 | 4 | 30 | 10 | 200 | 69.9612 | |
| 3 | 57 | 3 | 19 | 20 | 400 | 107.6016 | |
| 4 | 58 | 3 | 32 | 10 | 500 | 122.8908 | |
| … | … | … | … | … | … | … | |
| 3335 | 9261 | 4 | 20 | 8 | 400 | 158.1390 | |
| 3336 | 9264 | 5 | 16 | 56 | 1500 | 186.1320 | |
| 3337 | 9265 | 4 | 21 | 8 | 100 | 121.6992 | |
| 3338 | 9270 | 3 | 20 | 28 | 400 | 134.2848 | |
| 3339 | 9271 | 4 | 22 | 6 | 400 | 88.8246 | |

| | SelectByHotkeys | AssignToHotkeys | UniqueHotkeys | MinimapAttacks | \ |
|---|---|---|---|---|---|
| 0 | 0.003515 | 0.000220 | 7 | 0.000110 | |
| 1 | 0.003304 | 0.000259 | 4 | 0.000294 | |
| 2 | 0.001101 | 0.000336 | 4 | 0.000294 | |
| 3 | 0.001034 | 0.000213 | 1 | 0.000053 | |
| 4 | 0.001136 | 0.000327 | 2 | 0.000000 | |
| … | … | … | … | … | |
| 3335 | 0.013829 | 0.000504 | 7 | 0.000217 | |
| 3336 | 0.006951 | 0.000360 | 6 | 0.000083 | |
| 3337 | 0.002956 | 0.000241 | 8 | 0.000055 | |
| 3338 | 0.005424 | 0.000182 | 5 | 0.000000 | |
| 3339 | 0.000844 | 0.000108 | 2 | 0.000000 | |

| | MinimapRightClicks | NumberOfPACs | GapBetweenPACs | ActionLatency | \ |
|---|---|---|---|---|---|
| 0 | 0.000392 | 0.004849 | 32.6677 | 40.8673 | |
| 1 | 0.000432 | 0.004307 | 32.9194 | 42.3454 | |
| 2 | 0.000461 | 0.002926 | 44.6475 | 75.3548 | |
| 3 | 0.000543 | 0.003783 | 29.2203 | 53.7352 | |
| 4 | 0.001329 | 0.002368 | 22.6885 | 62.0813 | |
| … | … | … | … | … | |
| 3335 | 0.000313 | 0.003583 | 36.3990 | 66.2718 | |
| 3336 | 0.000166 | 0.005414 | 22.8615 | 34.7417 | |
| 3337 | 0.000208 | 0.003690 | 35.5833 | 57.9585 | |
| 3338 | 0.000480 | 0.003205 | 18.2927 | 62.4615 | |
| 3339 | 0.000341 | 0.003099 | 45.1512 | 63.4435 | |

| | ActionsInPAC | TotalMapExplored | WorkersMade | UniqueUnitsMade | \ |
|---|---|---|---|---|---|
| 0 | 4.7508 | 28 | 0.001397 | 6 | |
| 1 | 4.8434 | 22 | 0.001193 | 5 | |
| 2 | 4.0430 | 22 | 0.000745 | 6 | |
| 3 | 4.9155 | 19 | 0.000426 | 7 | |
| 4 | 9.3740 | 15 | 0.001174 | 4 | |
| … | … | … | … | … | |
| 3335 | 4.5097 | 30 | 0.001035 | 7 | |

```
3336        4.9309                  38      0.001343                 7
3337        5.4154                  23      0.002014                 7
3338        6.0202                  18      0.000934                 5
3339        5.1913                  20      0.000476                 8

      ComplexUnitsMade  ComplexAbilitiesUsed
0                  0.0              0.000000
1                  0.0              0.000208
2                  0.0              0.000189
3                  0.0              0.000384
4                  0.0              0.000019
…                  …                 …
3335               0.0              0.000287
3336               0.0              0.000388
3337               0.0              0.000000
3338               0.0              0.000000
3339               0.0              0.000054

[3338 rows x 20 columns]
```

# 3 Exploring Correlations

[5]:
```
corr = df_without_question_mark.corr()
```

[6]:
```
corr.columns
```

[6]:
```
Index(['GameID', 'LeagueIndex', 'Age', 'HoursPerWeek', 'TotalHours', 'APM',
       'SelectByHotkeys', 'AssignToHotkeys', 'UniqueHotkeys', 'MinimapAttacks',
       'MinimapRightClicks', 'NumberOfPACs', 'GapBetweenPACs', 'ActionLatency',
       'ActionsInPAC', 'TotalMapExplored', 'WorkersMade', 'UniqueUnitsMade',
       'ComplexUnitsMade', 'ComplexAbilitiesUsed'],
      dtype='object')
```

[7]:
```
league_index_corr = corr['LeagueIndex']
```

[8]:
```
league_index_corr
```

[8]:
```
GameID                  0.024974
LeagueIndex             1.000000
Age                    -0.127518
HoursPerWeek            0.217930
TotalHours              0.023884
APM                     0.624171
SelectByHotkeys         0.428637
AssignToHotkeys         0.487280
UniqueHotkeys           0.322415
```

```
MinimapAttacks          0.270526
MinimapRightClicks      0.206380
NumberOfPACs            0.589193
GapBetweenPACs         -0.537536
ActionLatency          -0.659940
ActionsInPAC            0.140303
TotalMapExplored        0.230347
WorkersMade             0.310452
UniqueUnitsMade         0.151933
ComplexUnitsMade        0.171190
ComplexAbilitiesUsed    0.156033
Name: LeagueIndex, dtype: float64
```

[9]:
```python
indices = np.where(abs(league_index_corr) > 0.5)
```

[10]:
```python
indices
```

[10]:
```
(array([ 1,  5, 11, 12, 13]),)
```

[11]:
```python
corr.columns[indices]
```

[11]:
```
Index(['LeagueIndex', 'APM', 'NumberOfPACs', 'GapBetweenPACs',
       'ActionLatency'],
      dtype='object')
```

From the **above** it appears that the categories with a relatively significant correlation to LeagueIndex (>0.5), which is our 1-8 code for rank, are **APM, NumberOfPACs, GapBetweenPACs, and ActionLatency**

So, it makes sense to continue by creating a classification model trained on this data. It is possible that adding in other features would only make our model worse because they are so uncorrelated with our output variable and could cause overfitting.

These variables make intuitive sense because APM, ActionLatency, NumberOfPACs, and GapBetweenPACs all correlate to how fast a player is and it makes sense that quicker players would have a higher rank because they have better reactions and more practice, developing their faster movement.

For further confirmation, I will also use scikit-learn's `SelectKBest` to see if selecting the 5 best features aligns with what we have above. I will use the f_classif because it is suitale for numerical classification.

[12]:
```python
features = list(df.columns)
features.pop(1)
print(features)
```

```
['GameID', 'Age', 'HoursPerWeek', 'TotalHours', 'APM', 'SelectByHotkeys',
'AssignToHotkeys', 'UniqueHotkeys', 'MinimapAttacks', 'MinimapRightClicks',
'NumberOfPACs', 'GapBetweenPACs', 'ActionLatency', 'ActionsInPAC',
```

```
'TotalMapExplored', 'WorkersMade', 'UniqueUnitsMade', 'ComplexUnitsMade',
'ComplexAbilitiesUsed']
```

[13]:
```
from sklearn.feature_selection import SelectKBest, f_classif
X = df_without_question_mark[features]
y = df_without_question_mark['LeagueIndex']
kbest = SelectKBest(score_func=f_classif, k=5)
kbest.fit(X, y)
selected_features = kbest.get_support(indices=True)
print(df.columns[selected_features])
```

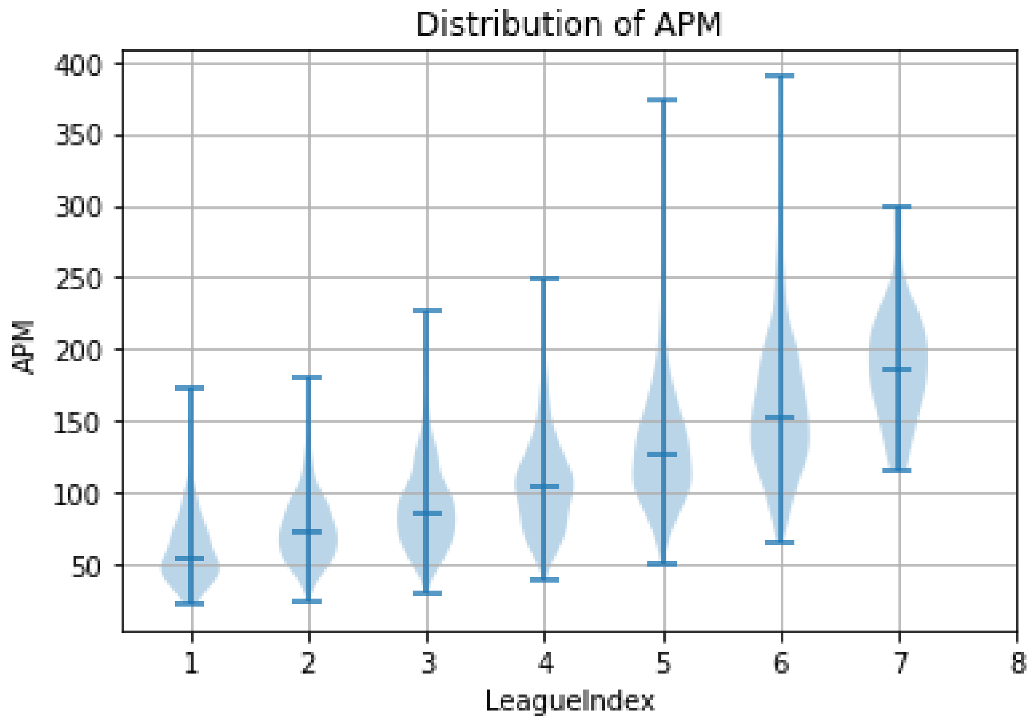
```
Index(['TotalHours', 'SelectByHotkeys', 'MinimapRightClicks', 'NumberOfPACs',
       'GapBetweenPACs'],
      dtype='object')
```

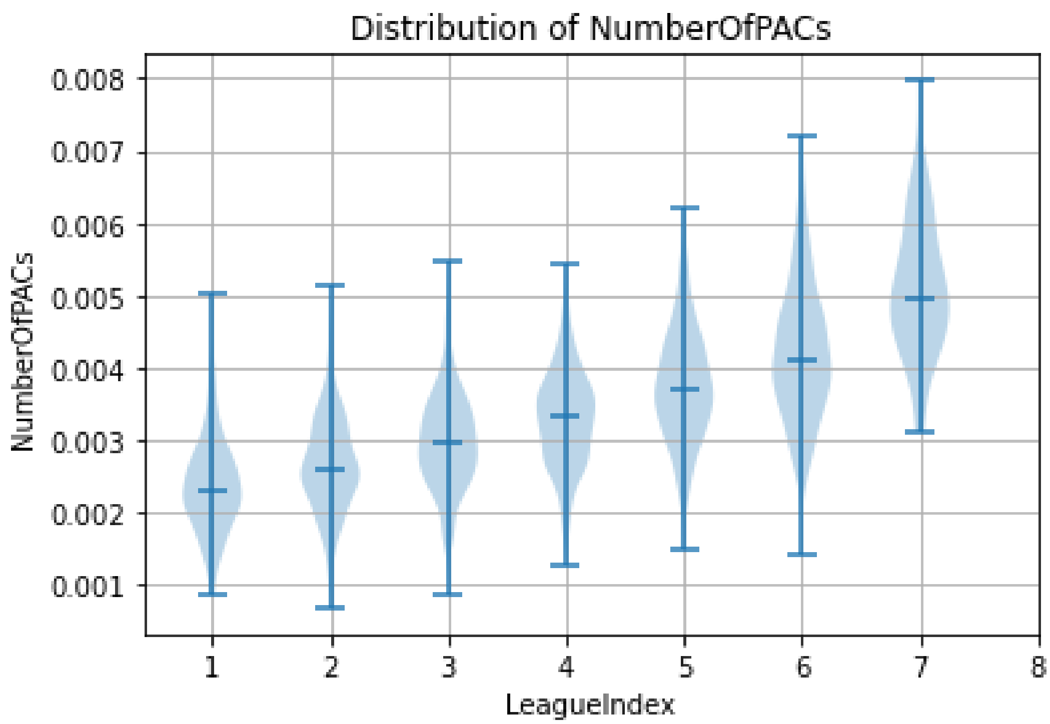
Performing this check we see that **TotalHours**, which was not included in the correlation matrix, is a good predictor for LeagueIndex, so this is an additional feature worth exploring. Scikit-Learn also suggests **MinimapRightClicks** is a good predictor, which makes sense because a more skilled player will check the minimap more often. Also, **SelectByHotkeys** measures some efficiency of game play, so also makes sense to correlate with skill

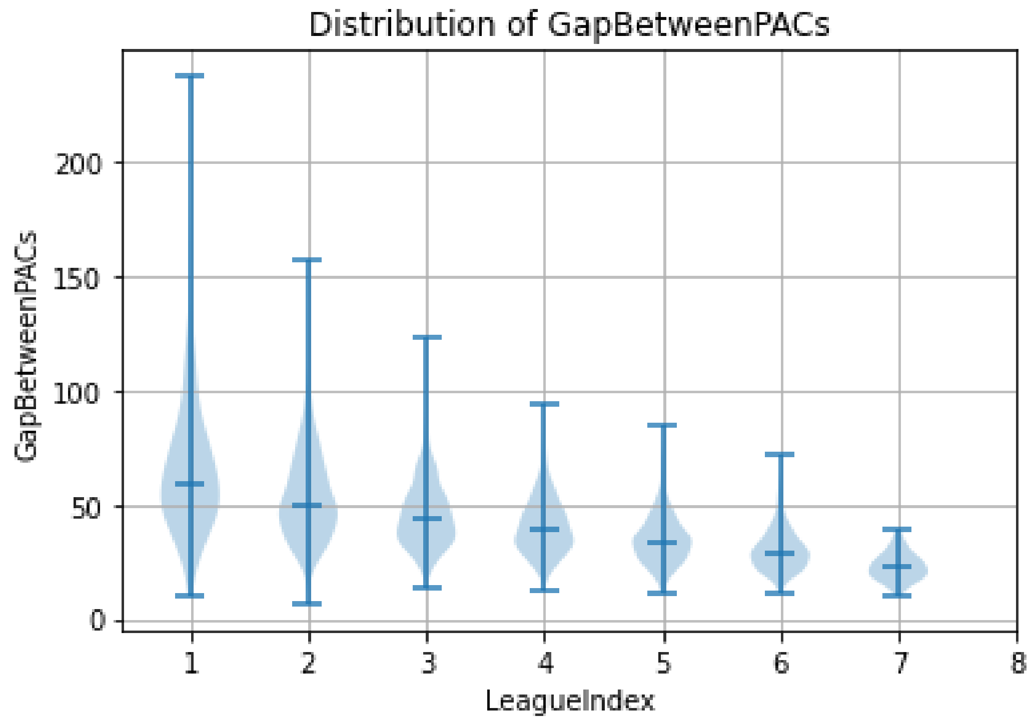
## 4 Visualizing Variables of Interest

[14]:
```
import matplotlib.pyplot as plt
```

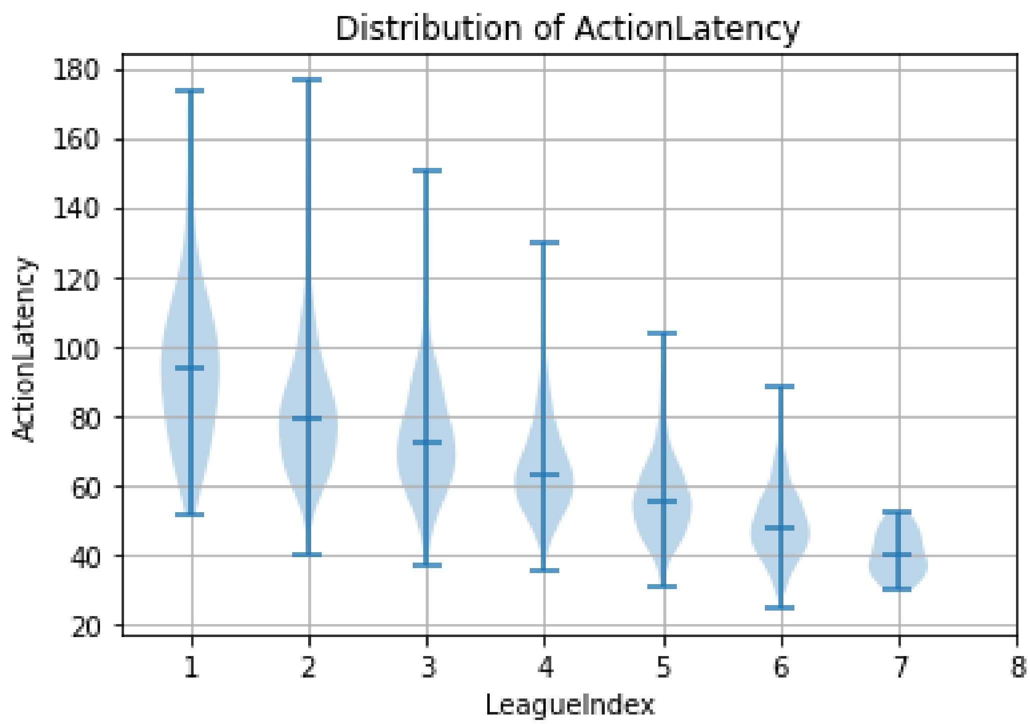
[15]:
```
var = ['APM', 'NumberOfPACs', 'GapBetweenPACs',
       'ActionLatency', 'TotalHours', 'SelectByHotkeys', 'MinimapRightClicks']
for v in var:
    data_list = []
    for i in range(1,8):
        data = df_without_question_mark[df_without_question_mark['LeagueIndex']␣
 ↪== i][v]
        data_list.append(data)
    plt.violinplot(data_list, showmedians=True)
    plt.xlabel('LeagueIndex')
    plt.ylabel(v)
    plt.title(f'Distribution of {v}')
    plt.xticks(range(1, 9), range(1, 9))
    plt.grid(True)
    plt.show()
```


Distribution of APM
APM
400
350
300
250
200
150
100
50
1
2
3
4
5
6
7
8
LeagueIndex

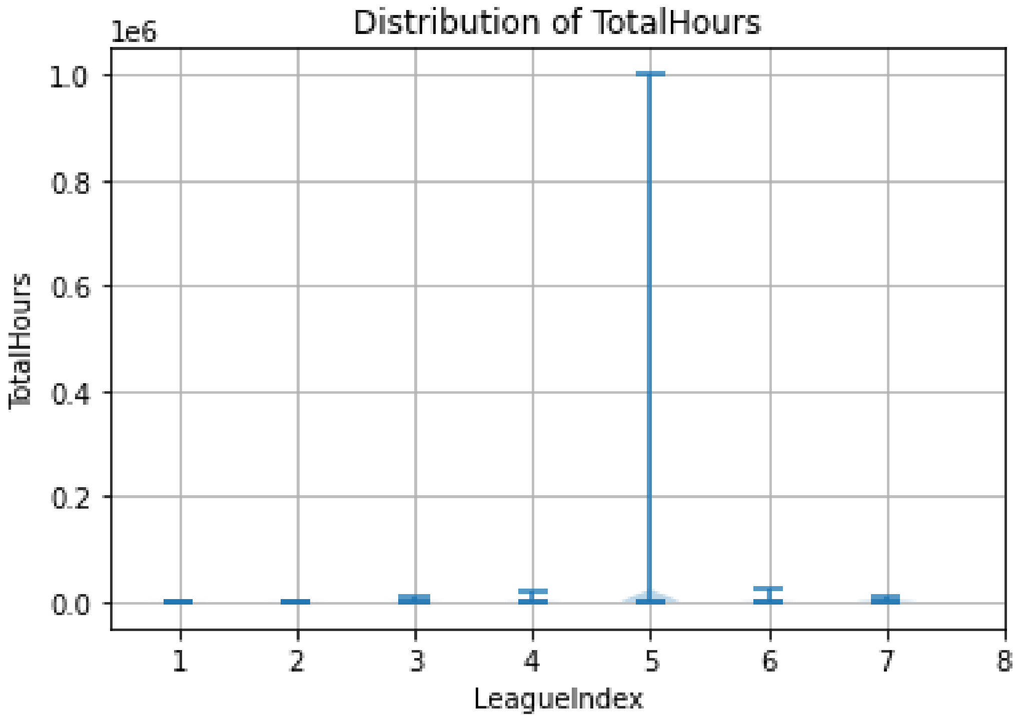

Distribution of NumberOfPACs
NumberOfPACs
0.008
0.007
0.006
0.005
0.004
0.003
0.002
0.001
1
2
3
4
5
6
7
8
LeagueIndex

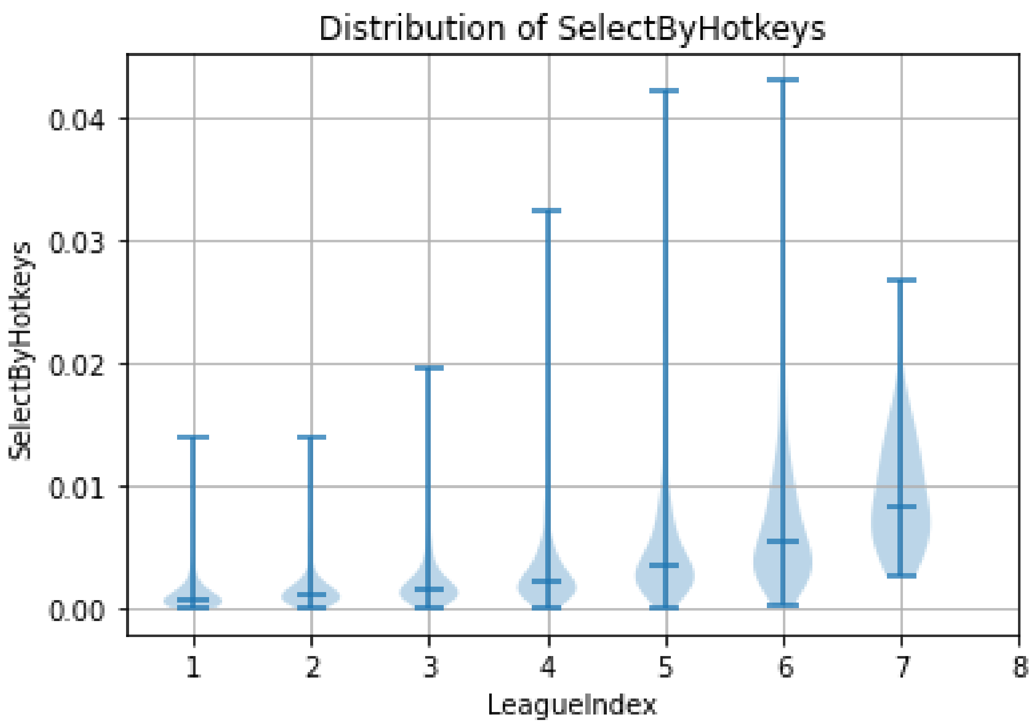


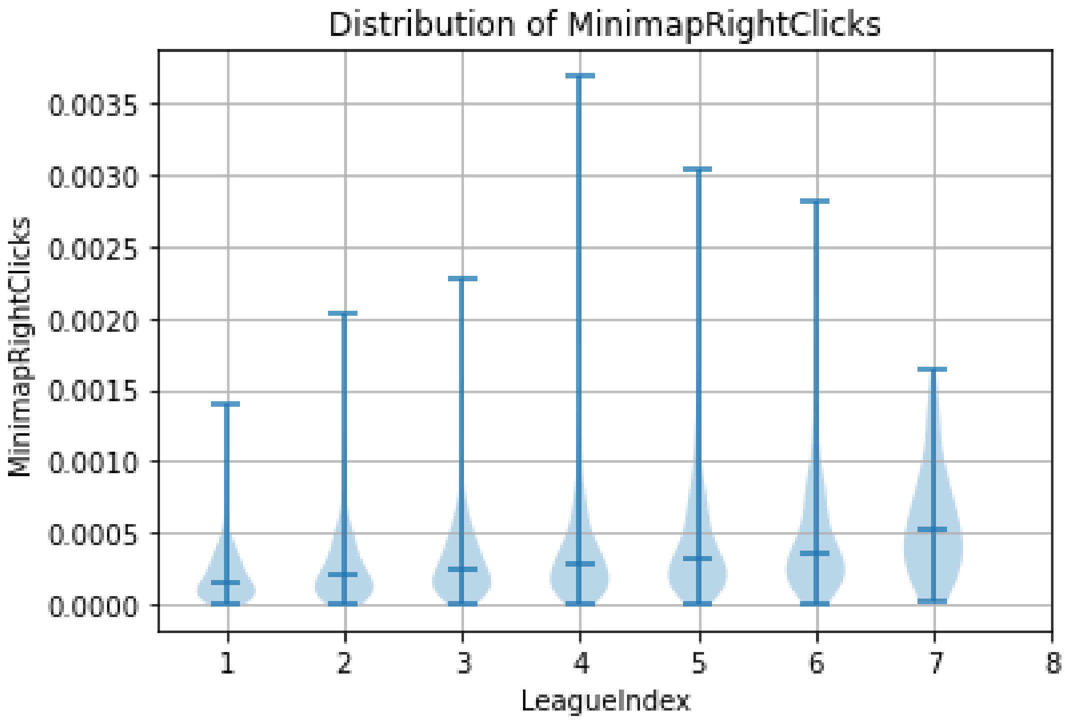
Distribution of GapBetweenPACs
GapBetweenPACs
0
50
100
150
200
1
2
3
4
5
6
7
8
LeagueIndex

Distribution of ActionLatency
ActionLatency
20
40
60
80
100
120
140
160
180
1
2
3
4
5
6
7
8
LeagueIndex

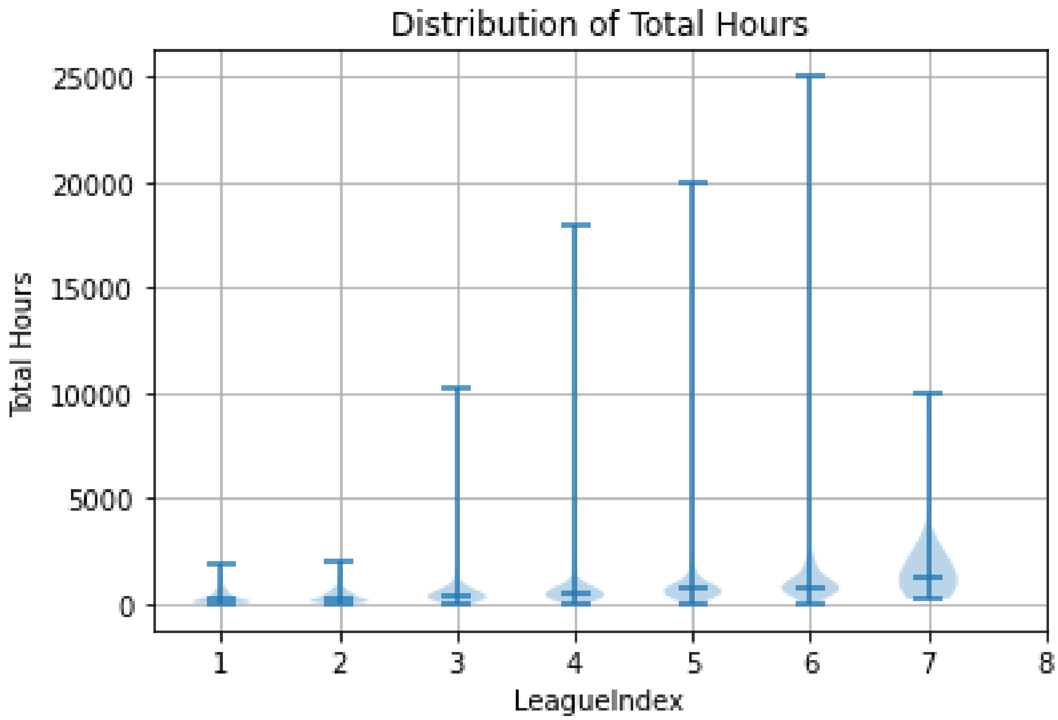

Distribution of TotalHours
1e6
1.0
0.8
0.6
0.4
0.2
0.0
TotalHours
1
2
3
4
5
6
7
8
LeagueIndex


Distribution of SelectByHotkeys
0.04
0.03
0.02
0.01
0.00
SelectByHotkeys
1
2
3
4
5
6
7
8
LeagueIndex




Looking at this we see an outlier in **TotalHours** that makes it hard to visualize so I'll remove that to see if there is a clear relation with our target variable

[16]:
```
data_list = []
for i in range(1,8):
    data = df_without_question_mark[(df_without_question_mark['LeagueIndex'] ==␣
 ↪i) & (df_without_question_mark['TotalHours'] < 10**6)]['TotalHours']
    data_list.append(data)
plt.violinplot(data_list, showmedians=True)
plt.xlabel('LeagueIndex')
plt.ylabel('Total Hours')
plt.title(f'Distribution of Total Hours')
plt.xticks(range(1, 9), range(1, 9))  # Set x-axis tick labels to match␣
 ↪LeagueIndex values
plt.grid(True)
plt.show()
```




Now some relation does appear to be pressent when you follow the medians, so I will train the model on the classification model on these features with that row removed.

## 5 Train/Test Split For Classification Model

[17]:
```
data = df_without_question_mark[df_without_question_mark['TotalHours'] < 10**6]
```

[18]:
```
from sklearn.model_selection import train_test_split
X = data[['APM', 'NumberOfPACs', 'GapBetweenPACs',
       'ActionLatency', 'TotalHours', 'SelectByHotkeys', 'MinimapRightClicks']]
y = data['LeagueIndex']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)
```

[19]:
```
print("X_train shape:", X_train.shape)
print("X_test shape:", X_test.shape)
print("y_train shape:", y_train.shape)
print("y_test shape:", y_test.shape)
```

```
X_train shape: (2669, 7)
X_test shape: (668, 7)
y_train shape: (2669,)
y_test shape: (668,)
```

Based on the nature of the classification task (distinct integer categories) I will first try using a Decision Tree classifier. This is because it has good interpretability and may fit the data will, but it is possible to overfit, so I will also try a Random Forest, and see if that gives better performance on the test data. If overfitting does not appear to be a problem, the Decision Tree is preferable because of its interpretability. Otherwise, the Random Forest will be the solution to overfitting.

[20]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import mean_squared_error
model = DecisionTreeClassifier()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
print("RMSE for DT:", rmse)
```

```
RMSE for DT: 1.3262662010049655
```

[21]:
```
from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
print("RMSE for RF:", rmse)
```

```
RMSE for RF: 1.1045632078969139
```

It appears that the Random Forest gives better perfomance than the Decision Tree. However, because both are not great, and average being off by a bit more than 1 whole rank, I will try some other models.

[23]:
```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression(max_iter = 10000)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
print("RMSE for Logistic Regression:", rmse)
```

```
RMSE for Logistic Regression: 1.102528395224109
```

[24]:
```
from sklearn.svm import SVC
model = SVC()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
print("RMSE for SVC:", rmse)
```

```
RMSE for SVC: 1.2118427597058525
```

[25]:
```
from sklearn.ensemble import GradientBoostingClassifier
model = GradientBoostingClassifier()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
print("RMSE for Gradient Boosting Classifier:", rmse)
```

```
RMSE for Gradient Boosting Classifier: 1.1119923318565574
```

#### 5.0.1 To get a better sense of the different classification models I will test on multiple train-test splits and plot the RMSE's

[26]:
```
group_labels = ['Decision Tree', 'Random Forest', 'Logistic Regression', 'SVC',␣
 ↪'Gradient Boosting']

dt_rmse = []
for i in range(20):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)
    model = DecisionTreeClassifier()
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    mse = mean_squared_error(y_test, y_pred)
    rmse = np.sqrt(mse)
    dt_rmse.append(rmse)

rf_rmse = []
for i in range(20):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)
    model = RandomForestClassifier()
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    mse = mean_squared_error(y_test, y_pred)
    rmse = np.sqrt(mse)
    rf_rmse.append(rmse)

lr_rmse = []
for i in range(20):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)
    model = LogisticRegression(max_iter = 10000)
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
```

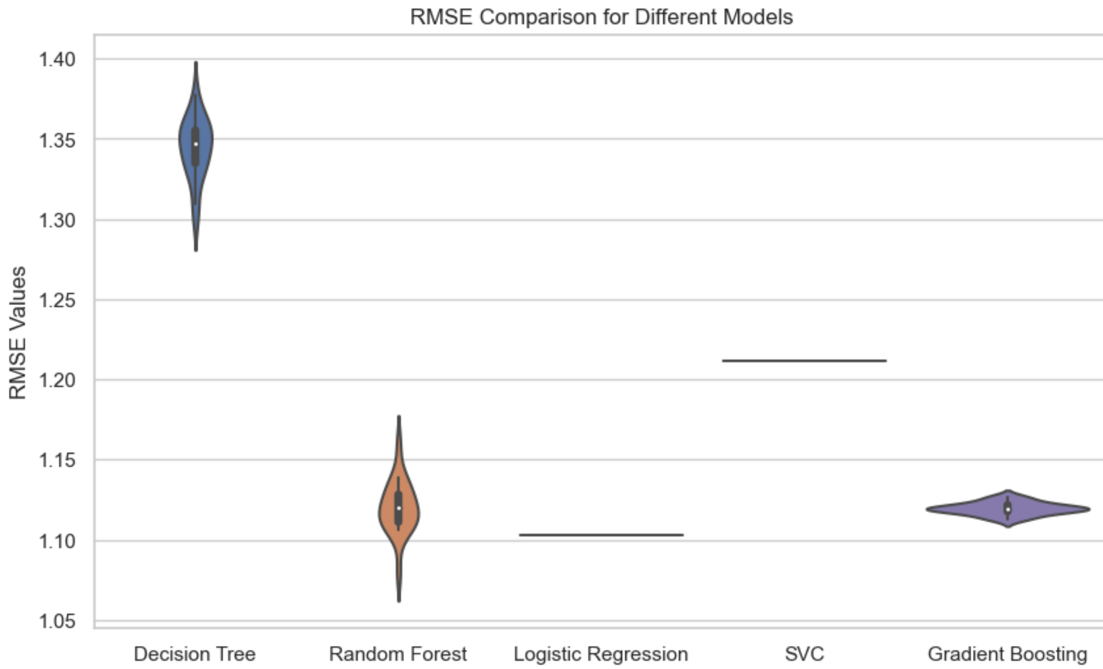
```python
    mse = mean_squared_error(y_test, y_pred)
    rmse = np.sqrt(mse)
    lr_rmse.append(rmse)

svc_rmse = []
for i in range(20):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)
    model = SVC()
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    mse = mean_squared_error(y_test, y_pred)
    rmse = np.sqrt(mse)
    svc_rmse.append(rmse)

gb_rmse = []
for i in range(20):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)
    model = GradientBoostingClassifier()
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    mse = mean_squared_error(y_test, y_pred)
    rmse = np.sqrt(mse)
    gb_rmse.append(rmse)
```

[27]:
```python
import seaborn as sns
import matplotlib.pyplot as plt
rmse_values = [dt_rmse, rf_rmse, lr_rmse, svc_rmse, gb_rmse]
sns.set(style="whitegrid")
fig, ax = plt.subplots(figsize=(10, 6), dpi=100)
ax = sns.violinplot(data=rmse_values)
ax.set_xticklabels(group_labels)
ax.set_ylabel("RMSE Values")
ax.set_title("RMSE Comparison for Different Models")
```

[27]:
```
Text(0.5, 1.0, 'RMSE Comparison for Different Models')
```




# 6 Conclusion

It appears that the best model in terms of performance is the logistic regression. Although the performance is similar for many of the classification models, notably Logistic Regression performs the best and a Random Forest performs better than the Decision Tree classifier on the test set, so overfitting appears to not be an issue.

I would recommend the use of the logistic regression model, specified below because it has the lowest RMSE, and it also has very good interpretability, because it is easy to understand the logistic regression function and what the probability outputs mean.

[30]:
```
model = LogisticRegression(max_iter = 10000)
model.fit(X_train, y_train)
coefficients = model.coef_
intercept = model.intercept_
print("Coefficients:", coefficients)
print("Intercept:", intercept)
```

```
Coefficients: [[-4.29913883e-02 -1.05585879e-03  3.16134003e-02  4.87196878e-02
  -2.91981942e-03 -1.28044809e-03 -6.66410500e-05]
 [-2.22673761e-02 -1.40441711e-03  2.41994190e-02  4.09717165e-02
  -1.70619303e-03 -1.47218343e-03  2.53727011e-05]
 [-7.46383150e-03 -1.47389053e-03  1.10171536e-02  3.59762384e-02
  -2.21119575e-04 -1.50877978e-04 -4.11117423e-05]
 [ 3.98836158e-03 -1.58173447e-03  1.56216107e-02  1.23861703e-02
   5.70873036e-04 -1.49598958e-05  1.00494948e-04]
```

```
 [ 1.57376439e-02 -2.04053181e-04 -1.84364241e-03 -7.37079994e-03
   1.26416692e-03  2.65290354e-03  4.78924268e-06]
 [ 2.41976807e-02  4.93847602e-03 -1.18064439e-02 -4.25426819e-02
   1.40715492e-03  1.01405980e-03 -4.21922613e-05]
 [ 2.87989097e-02  7.81478069e-04 -6.88014974e-02 -8.81403312e-02
   1.60493714e-03 -7.48493946e-04  1.92881613e-05]]
Intercept: [-0.10987328 -0.01899436 -0.32046441 -0.04163747 -0.02119247
0.47432829
  0.03783371]
```

### 6.1 For non-technical stake holders

By exploring the data, I found that several factors would be useful predictors of LeagueIndex i.e. rank. These factors were: **APM**, **NumberOfPACs**, **GapBetweenPACs**, **ActionLatency**, **TotalHours**, **SelectByHotkeys**, **MinimapRightClicks**. Most of these factors make sense because they relate to the speed of the player, and faster players probably have higher ranks, while others are good indicators of how much information a player is gathering or how much experience they have, which are both good ways to learn about the score. I chose to focus on this subset of factors to avoid over-fitting, which is where our model is trained so closely on training data, it struggles in the future to make accurate predictions. I ended up with a model that is both straight forward, as we can see its mathematical definition below, and in testing had a root mean squared error of only slightly greater than 1, which means we are on average roughly around 1 rank off, so it is a fairly good predictor. It also has no variability, which is better than the models with similar performance, as seen above.

Thus, the logistic regression model above is a suitable predictor of LeagueIndex.