Predicting the outcome of a Dota 2 match using different ML Classification models

March 31, 2022

1 Introduction

DotA 2 (Defence of the Ancients 2) is a MOBA[1] video game that is widely regarded as one of the most complex, if not the most complex, competitive video games ever created, and is sometimes referred to as the "chess of the digital age." To grasp at least the fundamentals of the game, a player must devote hundreds of hours. It is even employed as a great playground for various scientific research (e.g. OpenAI bots that 'destroyed' Team OG, winners of The International 8 & 9).

The game's complexity stems from the large number of diverse in-game mechanics, 122 unique heroes with 4-6 different abilities each, and over 230 goods that may be purchased during the match. Each Dota 2 match is unique since each hero may only be chosen by one of the players once. Basic calculations show that with such a large hero pool, there are 500127116029962220800[2] possible matches, which is far greater than the number of possible chess games. However, one element remains constant - the Map.

Despite the game's complexity, the outcome of the match may be predicted, and this research could be valuable in competitive Dota 2 eSports competitions. It may reveal which team is currently leading to less experienced viewers.

TOP
EASY (DIRE)
HARD (RADIANT)
MID
HARD (DIRE)
EASY (RADIANT)
BOT

[1]Multiplayer Online Battle Arena

[2]500 quintillion 127 quadrillion 116 trillion 29 billion 962 million 220 thousand 800

2 Methods

2.1 Dataset

To construct the datapoints of this project, I combined usage data from the OpenDota website using their own API, which parses Dota 2 matches from the official Valve Dota 2 API. I wrote a few-line-script that fetches last 100 parsed games and scraps all the needed data into one single DataFrame. Dataset has approximately 1000 different datapoints, however it can be increased even more by just running the script periodically. This should be enough for the ML methods used.

The only problem with dataset I found was a small bias towards Radiant side winning, but I couldn't understand surely why it happens. However, I suspect that it might come from that Radiant side usually wins more often than Dire side due to really small differences in the map. It is strange though, as latest statistics show that Dire should have taken a lead here. This finding shouldn't affect the result, though.

parsedMatches method from OpenDota API gives me 100 recent played matched in Dota, it is a JSON file full of different MatchIDs. After that, I parse each of that MatchID through match method in the same OpenDota API and take only those metrics, that interest me. Due to API limitations, a loop can go only through 60 matches in one iteration. However, after few second it can be laucnhed again.

Thanks to my own good understanding of the game, I can easily detect what affects result of the match the most. All the features used are the key factors of the game's success. Radiant Gold and XP advantages show how much more resources one team has, than the other one. Lane efficiency metrics are also important for the same reason.

Data points were split into training and validation sets using train test split with test size = 0.2 [1]. It is done to prevent model from overfitting and to evaluate it effectively.

2.2 Logistic regression

The first method I used to fit the data was a simple logistic regression model. It was chosen due to its relative simplicity, yet big effectiveness. It is also one of the best algorithms for binary classification. However, this method has a major limitation - the assumption of linearity between the dependent variable and the independent variables but it should really affect my project.

Log Loss (a.k.a. cross-entropy loss) is used here as Logistic Regression itself is built with the intent of using it. It is described as

$$L_{\log}(y, p) = -(y \log(p) + (1 - y) \log(1 - p))$$

Results of plugging my data into this model were interesting, to say the least. Although accuracy score was relatively good, that error was really strange to look at.

2.3 k-NN classification

The second model tested was an infamous and simple k-nearest neighbors classification algorithm. The method belongs to the class of non-parametric, i.e. does not require assumptions about which statistical distribution the training set was formed from. Therefore, the classification models built using the KNN method will also be non-parametric. This means that the structure of the model is not rigidly set initially, but is determined by the data. It's mains advantages - it is much faster compared to other classification algorithms and there is no need to train a model for generalization. There are some cons, however - as the numbers of variables grow K-NN algorithm struggles to predict the output of new data point and is very sensitive to outliers.

This model uses 0-1 loss, which is also as simple as the model itself and can be described as

$$L(\hat{y}, y) = I(\hat{y} \neq y)$$

Results of this approach were the best of all, in my opinion.

2.4 MLP Classification

Last, but not least, is really old, yet still powerful MultiLayer Perceptron classification model. It was invented in 1980-s but still finds it's way in modern days. It's main advantage is that is is capable to learn non-linear models and learn in real-time. However, it can be very sensitive to feature scaling. Nevertheless, as I don't use it in the project, this shouldn't worry me. Also, network finds out how to solve the problem by itself, hence its operation can be unpredictable

This model also uses log loss, that was mentioned earlier while discussing logistic regression method.

Results of this approach can be seen right under this line

3 Results

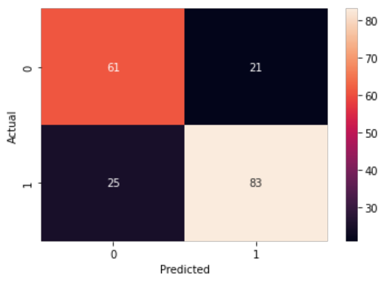
All the models have be measured using metrics included in self-titled module from sklearn. These are the results of their accuracy scores and test/train validaton errors from left to right (1. logistic regression, 2. kNN, 3. MLP)

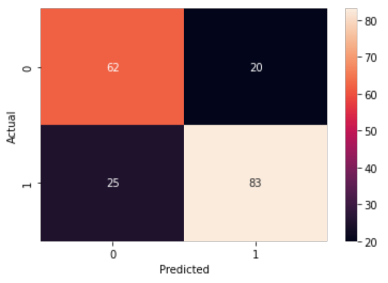
```
Accuracy train:
0.7849604221635884
train error:
7.427300943712426
Accuracy test:
0.7578947368421053
test error:
8.362107924799608
```

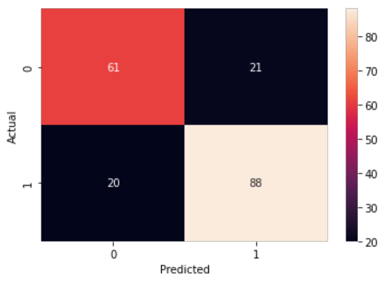
```
Accuracy train:
0.816622691292876
train error:
0.183377308707124
Accuracy test:
0.7631578947368421
test error:
0.23684210526315785
```

```
Accuracy train:
0.7704485488126649
train error:
7.928528566840617
Accuracy test:
0.7842105263157895
test error:
7.453192756512484
```

I also generated confusion matrices for each of the models. They are placed also in the same order as previous metrics. Based on this, the differences between logistic regression and MLPClassifier aren't big. However, kNN stands out with really good results and seems to be the best of all.







4 Conclusion

As a result, I decided to stick with kNN Classifier as a final model for this project. KNN is one of the best machine learning algorithms because it is simple, easy to understand, and versatile. Current results arleady seem to be not bad, yet not that good. However, there is always room for improvements that could be tried in the future.

First of all, I can try using feature sampling. By selecting the most important variables and eliminating redundant and irrelevant features, it can improve the machine learning process and increase the predictive power of machine learning algorithm.

Secondly, I might try searching and coming up with additional features that might affect the result of the match. This can increase the result of the match.

Finally, I need to understand why there is such a bias towards team in original Data, which seems to be really weird.

5 References

1. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.33.1337&rep=rep1&type=pdf
2. https://scikit-learn.org/stable/index.html

Dota 2 ML Project

March 31, 2022

[1]:
```
import urllib3
from urllib3 import request
import certifi
import json
import pandas as pd
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn import metrics
import seaborn as sn
import matplotlib.pyplot as plt
```

[ ]:
```
#shouldn't be runned again, as it will reset already existing dataset
dota2DataFrame = pd.DataFrame(columns = ['match_id', 'duration','REFF@10',␣
 ↪'RCREEPS@10', 'DEFF@10', 'DCREEPS@10' ,'RGA@10', 'RXA@10', 'radiant_win'])
```

[81]:
```
#loading already fetched data
dota2DataFrame = pd.read_csv('/content/sample_data/dota2ml.csv').drop('Unnamed:␣
 ↪0', axis=1)
```

[3]:
```
http = urllib3.PoolManager(
       cert_reqs='CERT_REQUIRED',
       ca_certs=certifi.where())
```

[82]:
```
url = 'https://api.opendota.com/api/parsedMatches'
r = http.request('GET', url)
print(r.status)
data = json.loads(r.data.decode('utf-8'))
df = pd.json_normalize(data)
```

```
200
```

[83]:
```
for i in range(45):

  urltest = 'https://api.opendota.com/api/matches/%s' % df['match_id'][i]
  reff = 0.0
  deff = 0.0
  rCreeps = 0
  dCreeps = 0
  r = http.request('GET', urltest)
  ata = json.loads(r.data.decode('utf-8'))
  df2 = pd.json_normalize(ata)
  temp_df = pd.json_normalize(df2['players'][0])

  if df2['duration'][0] > 600:
    for k in range(10):
      if k <= 4:
        reff += temp_df['lane_efficiency_pct'][k]
        rCreeps += temp_df['lh_t'][k][10]
      else:
        deff += temp_df['lane_efficiency_pct'][k]
        dCreeps += temp_df['lh_t'][k][10]
    df2['radiant_gold_adv'] = df2['radiant_gold_adv'][0][10]
    df2['radiant_xp_adv'] = df2['radiant_xp_adv'][0][10]
    dota2DataFrame.loc[i+dota2DataFrame.shape[0]] = [df2['match_id'][0],␣
↪df2['duration'][0], reff / 5, rCreeps, deff / 5, dCreeps,␣
↪df2['radiant_gold_adv'][0], df2['radiant_xp_adv'][0], df2['radiant_win'][0]]
```

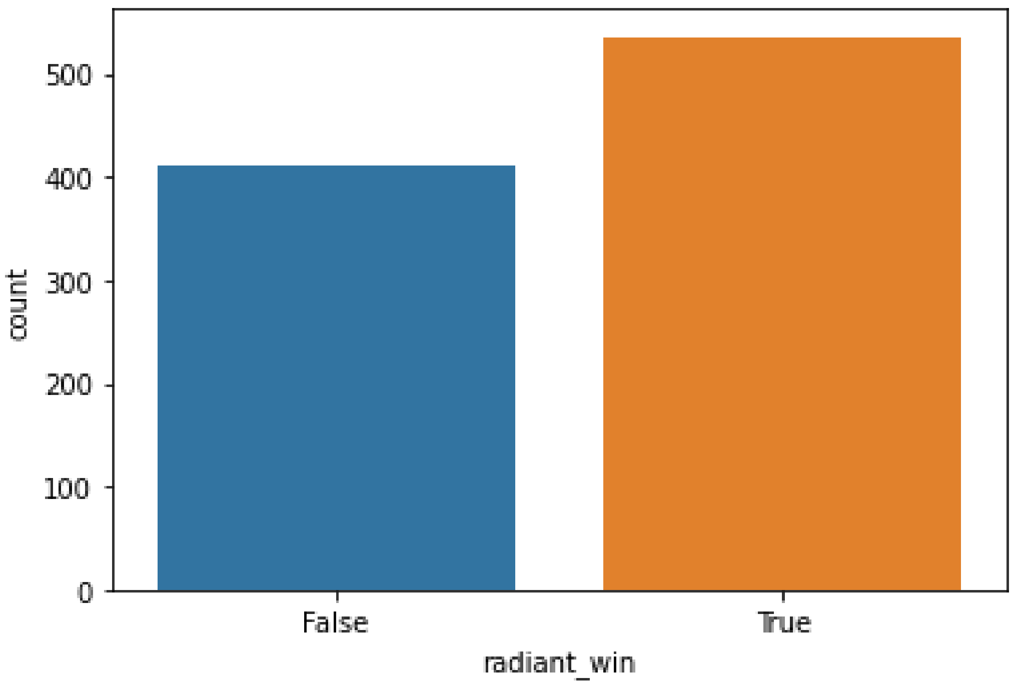
[84]:
```
dota2DataFrame = dota2DataFrame.drop_duplicates().reset_index().drop('index',␣
↪axis = 1)
dota2DataFrame
```

[84]:
```
       match_id  duration  REFF@10  RCREEPS@10  DEFF@10  DCREEPS@10  RGA@10  \
0    6501224294       747    191.6         191    145.8         172   11394
1    6501222069       738     54.0          82     56.8          81    -648
2    6501220593      1002    138.4         116     83.0          62   13696
3    6501219014      1138    112.2         120    128.8         149   -4122
4    6501218564      1101    170.8         150    121.0         103   12334
..          …         …        …          …        …           …       …
943  6502273817      1428     57.8         154     57.8         118      58
944  6502273523      1470     56.6         159     56.2         161     104
945  6502273269       876    148.6         224    108.0         128   10032
946  6502272422      1072    143.4         133     99.0          87   10960
947  6502272381      1549     48.2          91     60.8         122   -3116

     RXA@10  radiant_win
0      8354         True
1      -156        False
2     14366         True
3     -5053         True
4     11041         True
..      …           …
943    1997         True
944   -1118        False
945    9236         True
```

```
946   15226          True
947   -2990         False

[948 rows x 9 columns]
```

[85]:
```
#Radiant win\loss
sn.countplot(x = 'radiant_win', data = dota2DataFrame)
```

[85]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f8188eeda50>
```



[86]:
```
X = dota2DataFrame[['REFF@10', 'RCREEPS@10', 'DEFF@10', 'DCREEPS@10' ,'RGA@10',␣
↪'RXA@10']]
y = dota2DataFrame['radiant_win'].replace(False, 0).replace(True, 1)
```

[87]:
```
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.2,␣
↪random_state = 0)
```

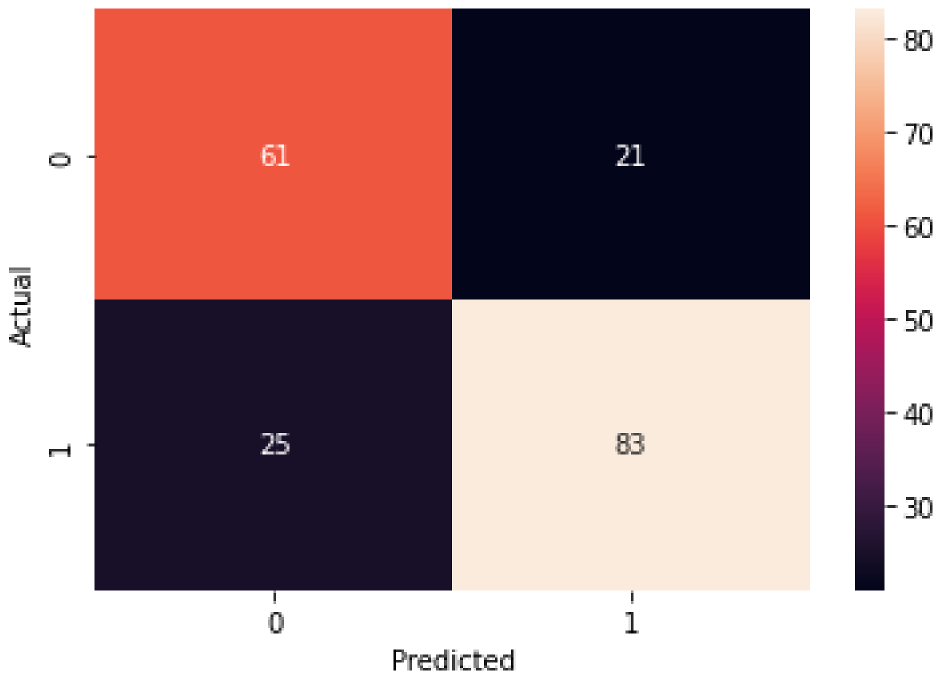
[88]:
```
logistic_regression = LogisticRegression()
logistic_regression.fit(X_train,y_train)
y_log_pred_train = logistic_regression.predict(X_train)
y_log_pred_test = logistic_regression.predict(X_test)
```

[89]:
```
print('Accuracy train: ',metrics.accuracy_score(y_train, y_log_pred_train))
print('train error: ',metrics.log_loss(y_train, y_log_pred_train))
print('Accuracy test: ',metrics.accuracy_score(y_test, y_log_pred_test))
print('test error: ',metrics.log_loss(y_test, y_log_pred_test))
```

```
Accuracy train:  0.7849604221635884
train error:  7.427300943712426
Accuracy test:  0.7578947368421053
test error:  8.362107924799608
```

[90]:
```
confusion_matrix = pd.crosstab(y_test, y_log_pred_test, rownames=['Actual'],␣
 ↪colnames=['Predicted'])
sn.heatmap(confusion_matrix, annot=True)
```

[90]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f81855a7110>
```



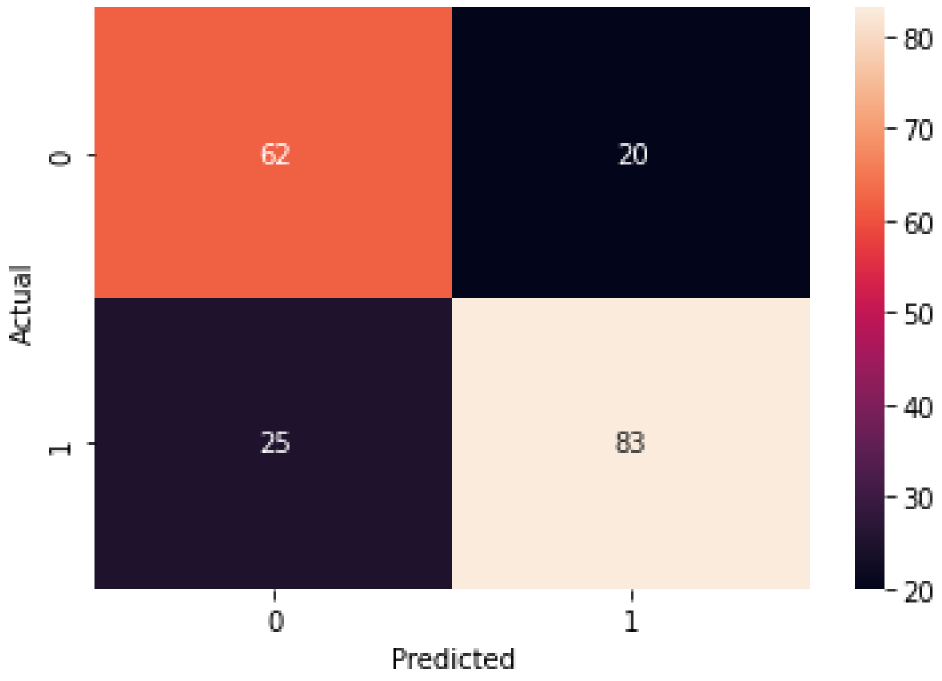
[91]:
```
from sklearn.neighbors import KNeighborsClassifier
```

[92]:
```
dotaNeigh = KNeighborsClassifier(n_neighbors=5)
dotaNeigh.fit(X, y)
y_knn_pred_train = dotaNeigh.predict(X_train)
y_knn_pred_test = dotaNeigh.predict(X_test)

print('Accuracy train: ',metrics.accuracy_score(y_train, y_knn_pred_train))
print('train error: ',metrics.zero_one_loss(y_train, y_knn_pred_train))
print('Accuracy test: ',metrics.accuracy_score(y_test, y_knn_pred_test))
print('test error: ',metrics.zero_one_loss(y_test, y_knn_pred_test))
```

```
Accuracy train:  0.816622691292876
train error:  0.183377308707124
Accuracy test:  0.7631578947368421
test error:  0.23684210526315785
```

[97]:
```
confusion_matrix = pd.crosstab(y_test, y_knn_pred_test, rownames=['Actual'],␣
 ↪colnames=['Predicted'])
sn.heatmap(confusion_matrix, annot=True)
```

[97]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f81853ab810>
```



[94]:
```
from sklearn.neural_network import MLPClassifier
```

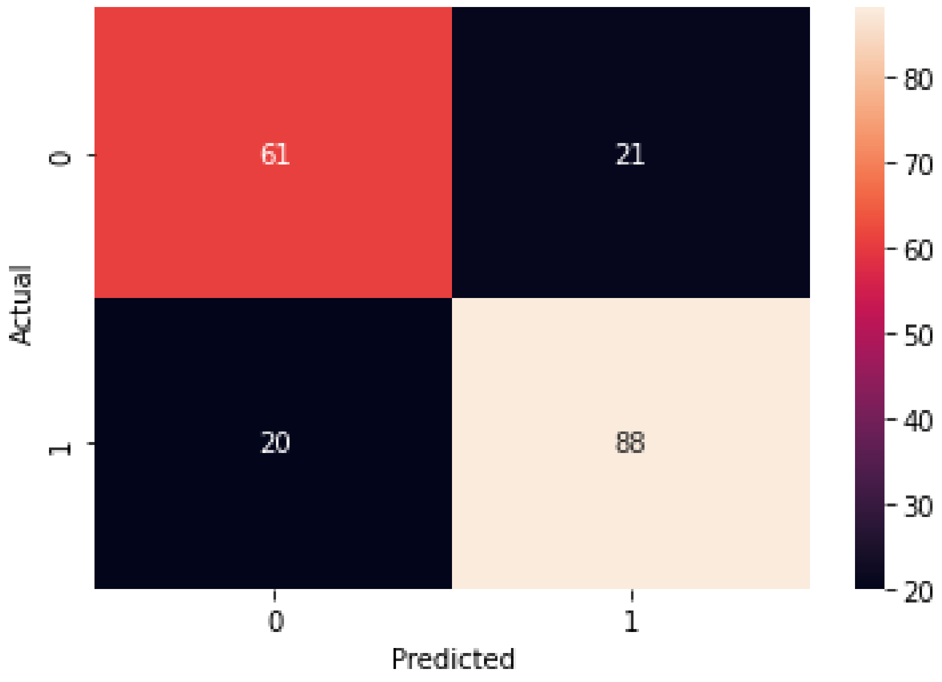
[98]:
```
mlp = MLPClassifier(hidden_layer_sizes=(125, 75, 25), max_iter=200␣
 ↪,random_state=42)
mlp.fit(X, y)
y_mlp_pred_train = mlp.predict(X_train)
y_mlp_pred_test = mlp.predict(X_test)

print('Accuracy train: ',metrics.accuracy_score(y_train, y_mlp_pred_train))
print('train error: ',metrics.log_loss(y_train, y_mlp_pred_train))
print('Accuracy test: ',metrics.accuracy_score(y_test, y_mlp_pred_test))
print('test error: ',metrics.log_loss(y_test, y_mlp_pred_test))
```

```
Accuracy train:  0.7704485488126649
```

```
train error:  7.928528566840617
Accuracy test:  0.7842105263157895
test error:  7.453192756512484
```

[96]:
```
confusion_matrix = pd.crosstab(y_test, y_mlp_pred_test, rownames=['Actual'],␣
 ↪colnames=['Predicted'])
sn.heatmap(confusion_matrix, annot=True)
```

[96]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f81854e9ad0>
```


Actual
Predicted