Reagan Turedi
April 5, 2020

Fixing The Report Function

Comments, concerns and suggestions

The Problem

According to a well-established website, there are currently 18.6 million tracked players. Of those 18.6 million, there is a high average of about 1 million people who are involved in daily player vs. player activities. If 1 out of every 1,000 players were cheating, that would result in about 1,000 cheaters. This number disregards any new influx of cheaters. Although the pattern is not a linear substitution of old and new, (those 1,000 cheaters are not replaced by 1,000 new cheaters the next day), we can see that even on the low end of an approximation, the number is way too large to be managed manually.

This number assumes that only cheaters are reported. Since cheating is a well-established issue, it is necessary to mention that players who are not cheating may also be reported. Let us assume that for every 1 cheater, there are 2 players who are falsely reported as well. Let's assume that only 75% of the cheaters are reported. So a low-low average of reports per day could be anywhere from 2,000-3,000. If it took five minutes to manually investigate each account, that would be 10,000 minutes investigating potential cheaters, and administering bans. This is seven days worth of managing, per day. A company would need a battalion of employees, and then some, to manage this.

The Solution

Machine learning programs are wonderful tools. We benefit from machine learning on a daily basis, whether it's filtering out junk emails or finding a "friend you may know" on social media, it makes our lives easier. Machine learning can be used to identify potential cheaters! In the following pages, I have provided a Python example of how we can use machine learning to do so.

Here is the basic idea:

The way an average players performs is linear when considering things such as kills, deaths, accuracy, average kill distance, loadout, etc. From this, we can assume things like the following:

- A player primarily using a shotgun will on average die more often than a player primarily using a sniper
- A player with a shorter kill distance average will die more often than a player with a farther kill distance average.
- A player with more kills will likely experience more deaths, assuming that their low-kill game averages have smaller death counts.
- A player with more kills compared to deaths per game will have a higher accuracy

The list goes on

Taking this most basic considerations into a machine learning environment does not have to be difficult, as "said company" has access to a limitless amount of data for these games. The more data you obtain and use, the more accurate your program becomes.

You can separate players based on skill tiers, such as average kills, deaths, and accuracy. This provides a sufficiently large enough gap that outliers in your data will become obvious. Luckily, these outliers do not have to be identified manually. A giant, extremely large, gargantuan data frame consisting of endless match statistics, separated by skill tiers of players who are **not** cheating can have a classifier of 0. A giant, extremely large, gargantuan data frame consisting of endless match statistics of **cheaters** can have a classifier of 1. The program can take in new inputs (new inputs are passed via the report function, automatically!) and determine whether a player **could** be cheating. If an account is flagged, then the account is sent to the security team for confirmation. This streamlines the report process, and makes things much more manageable.

Please see the next few pages for a quick example.

EasyReport

April 3, 2020

[245]:
```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
df=pd.read_csv('recent-games.csv')
kills=df['kills']
deaths=df['deaths']
accuracy=df['accuracy']
result=df['result']
not_cheating=df[0:499]
cheating=df[499:1000]
```

1 The data frame is showing us kills, deaths and accuracy. The "result" column is a 0 for not cheating, and 1 for cheating. These are known values, and are what we will use to train a program to make a determination (based on new inputs) whether or not an individual was cheating.

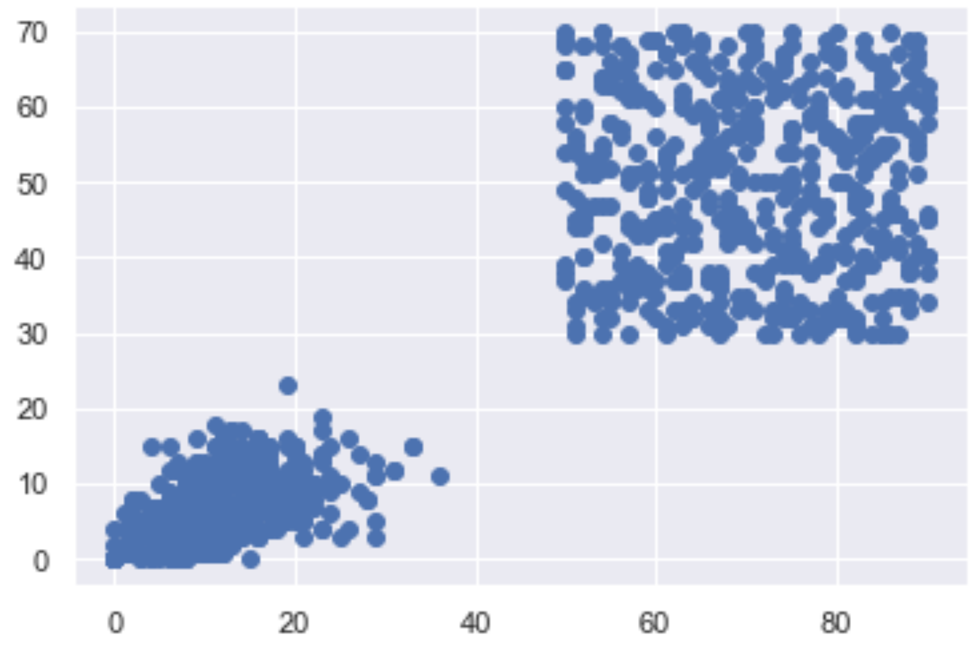
[116]:
```
df
```

[116]:
```
     kills  deaths  accuracy  result
0        9       2        85       0
1       19      10        85       0
2       14       3        72       0
3        7       7        84       0
4       17       9        70       0
..     …       …        …       …
995     75      70        99       1
996     54      52       100       1
997     74      33       100       1
998     61      67        98       1
999     88      44        99       1

[1000 rows x 4 columns]
```

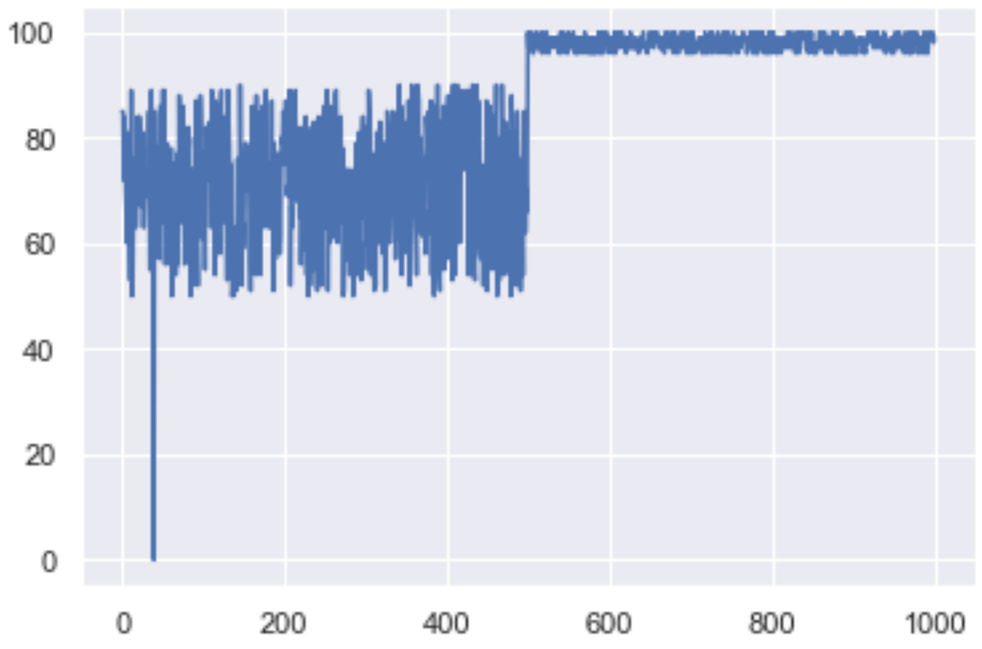
2 If we look at the data visually, we can see that the distribution for the player that was cheating is far more random compared to one that was not. You can clearly see that a separation occurs between the two. The statistical distribution for players who cheat is randomized, and has distinct visual classifications. For those who do not, there is an obvious relationship.

```
[117]: plt.scatter(kills,deaths)
       sns.set()
```



3 This separation is much more visible when looking at accuracy.

```
[121]: plt.plot(accuracy)
       sns.set()
```



4 We are using kills and deaths to make our first determination. We can use importable libraries to do the harder work for us. This makes things much more simple.

[244]:
```
kda=df[df.columns[0:3]].to_numpy()
from sklearn.model_selection import train_test_split
from scipy.spatial import distance

X_train, X_test, y_train, y_test=train_test_split(kda, result, train_size=0.
↪99,test_size=0.01,
                                                    random_state=1)


class kNN():
    def __init__(self,k):
        self.k=k
    def get_train_data(self,train_features,train_labels):
        self.train_features=train_features
        self.labels=train_labels


    def predict(self,X_test):
        predictions=[]
        for i in (X_test):
            distances=[]
            for j in (self.train_features):
                temp=distance.euclidean(i,j)
                distances.append(temp)
            a=np.argsort(distances)
            a_0_5=(a[0:self.k])
            s=self.labels[a_0_5]
            s=list(s)
            max_temp=max(set(s),key=s.count)

            predictions.append(max_temp)

        return predictions

K=kNN(k=500)

K.get_train_data(X_train,y_train)

K.predict(X_test)

Key=K.predict(X_test)

from sklearn.metrics import accuracy_score
acc=accuracy_score(y_test, Key)
print('The accuracy of the test data is',acc*100,'%')
```

```
The accuracy of the test data is 100.0 %
```

5 Showing results.

[243]:
```
Final=pd.DataFrame(X_test,Key)
#Final[1]
#Final=Final.shift(periods=1, axis=0)
Final=Final.rename(index={0: "Not Cheating", 1: "Cheating"})
with pd.option_context('display.max_rows', None, 'display.max_columns', None):
    print(Final)
```

```
               0   1    2
Cheating      69  61   97
Cheating      83  67  100
Not Cheating  24  15   86
Not Cheating   1   4   77
Not Cheating  11   4   84
Cheating      84  61  100
Not Cheating   7  12   88
Cheating      80  56   96
Not Cheating   5   7   65
Not Cheating  16  16   70
```

6 Application

7 If someone is suspected of cheating, their game statistics are passed as a new input. If “x” percent of games show up as possible instances of cheating, their accounts are immediately flagged. This streamlines the report process. A security team cannot look at every account that is reported, but they CAN look at flagged accounts from a computer program.

[ ]: