# clan_capital_performance

March 17, 2023

[1]:
```
# import stuff
import pandas
import numpy as np
import matplotlib.pyplot as plt
import statsmodels.api as sm
from patsy import dmatrices
from numpy.polynomial.polynomial import Polynomial
from scipy.optimize import curve_fit
```

The data used for this was collected by ClashCliffs and includes the performances of over 30k clans over several weeks. Some performance values are wrong due to a bug where the game gives clans not enough trophies and corrects that later. When ClashCliffs tracked the trophies before that correction, the correction gets counted towards the following week. This leads to an unrealisticly low performance in the first week and an unrealisticly high one in the second week. In the following process I try to filter out these wrong values as good as possible.

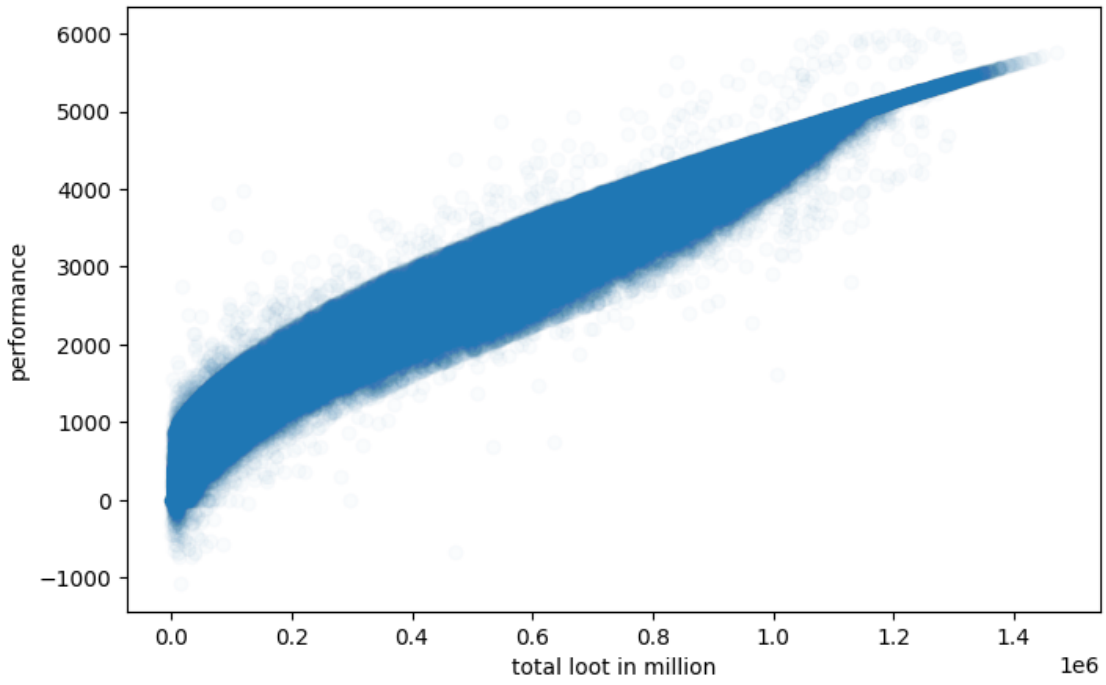
[2]:
```
data = pandas.read_csv('performance_data.csv')
# At 2023-01-30, sc changed the matchmaking and many clans had no defense. We␣
 ↪want to filter that out.
data = data.loc[data['record_date'] != '2023-01-30']
# everything above 6k is unrealistic
data = data.loc[data['performance'] < 6000]
data = data.loc[data['loot'] > 0]
```

[3]:
```
# add columns
data['avg_loot'] = data['loot'] / data['attack_count']
data['avg_def_loot'] = data['defense_loot'] / data['defense_count']
```

When you plot the performance against the total loot, the upper boundary is so clear that it makes a lot of sense to find its formula

[4]:
```
fig, ax = plt.subplots(figsize=(8, 5))
ax.scatter(data['loot'], data['performance'], alpha=0.02)
ax.set_xlabel('total loot in million')
ax.set_ylabel('performance')
```

[4]:
```
Text(0, 0.5, 'performance')
```




We will see how to get this upper bound below, but for that to work, we need to filter out unrealistictly high values before

All the values above it ore more than 1500 below are unrealistic, probably bugs and better be removed

[5]:
```
data['upper_bound'] = 5 * (data['loot'] + 100000)**0.5 - 500
data = data.loc[data['performance'] <= data['upper_bound']]
data = data.loc[data['performance'] >= data['upper_bound'] - 1500]
```

For finding this formula, we first need to extract this upper bound

[6]:
```
selection = data
selection['loot'] = selection['loot'].round(-4)
selection = selection.groupby('loot').max('performance')
```

do some data cleaning so the fit does not break

[7]:
```
selection = selection.loc[0 < selection.index]
selection = selection.loc[selection['performance'] > 0]
```

Find the best parameters

[8]:
```
def law(x, a, b, c):
    return a + b * (x - c)**0.5
    # return a * x**4 + b * x**3 + c * x**2 + d * x + e
```

```
fit = curve_fit(law, selection.index, selection['performance'])
fit
```

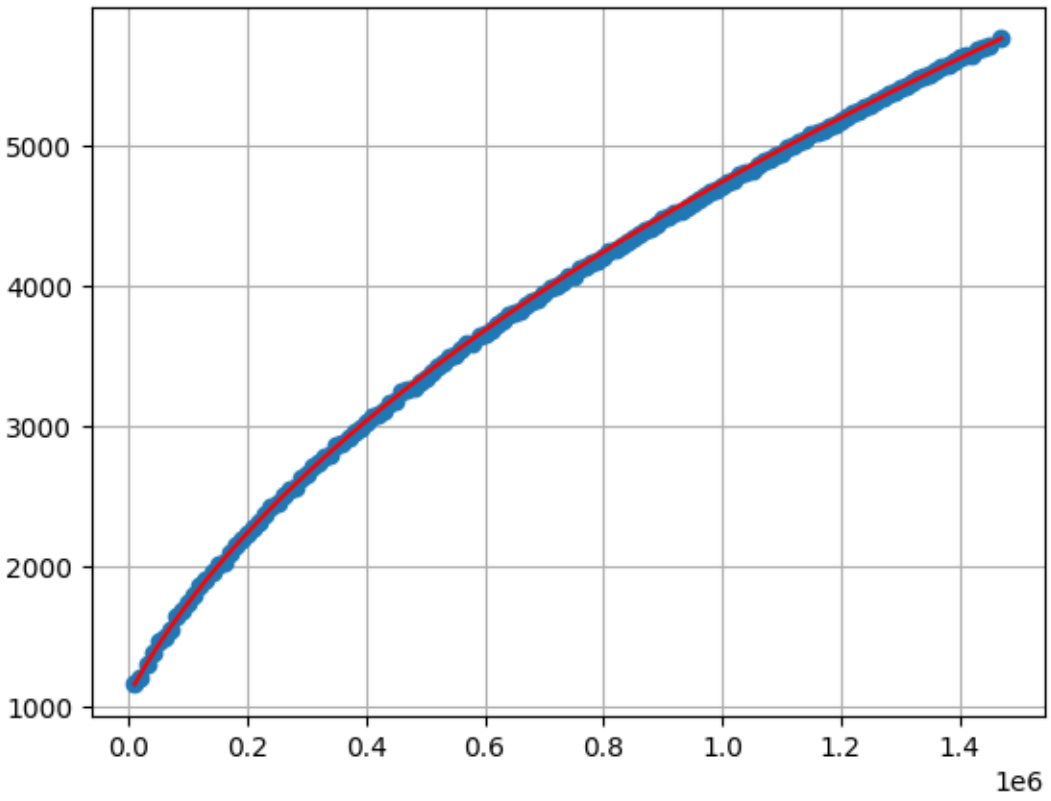
[8]:
```
(array([-5.89306520e+02,  5.03742584e+00, -1.12969680e+05]),
 array([[ 3.28707342e+02, -2.10746370e-01,  4.47551517e+04],
        [-2.10746370e-01,  1.38337368e-04, -2.79743582e+01],
        [ 4.47551517e+04, -2.79743582e+01,  6.33599668e+06]]))
```

with some rounding, these parameters become so clean that we can assume it is part of the real formula

[9]:
```
top_popt = [-500, 5, -100000]
top_y_model = law(selection.index, *top_popt)
y_model = law(selection.index, *fit[0])
```

Plot models against the selection

[10]:
```
plt.plot(selection.index, selection['performance'], 'o')
plt.plot(selection.index, top_y_model, 'r')
plt.grid()
```



Find RMSD (this indicates how well our formula fits. A mean R of 0.9997 is extremely good)

[11]:
```
x = selection.index
y = selection['performance']
func = law
yn = y + 0.2*np.random.normal(size=len(x))
print("Mean Squared Error: ", np.mean((y-func(x, *top_popt))**2))
ss_res = np.dot((yn - func(x, *top_popt)),(yn - func(x, *top_popt)))
ymean = np.mean(yn)
ss_tot = np.dot((yn-ymean),(yn-ymean))
print("Mean R :",  1-ss_res/ss_tot)
```

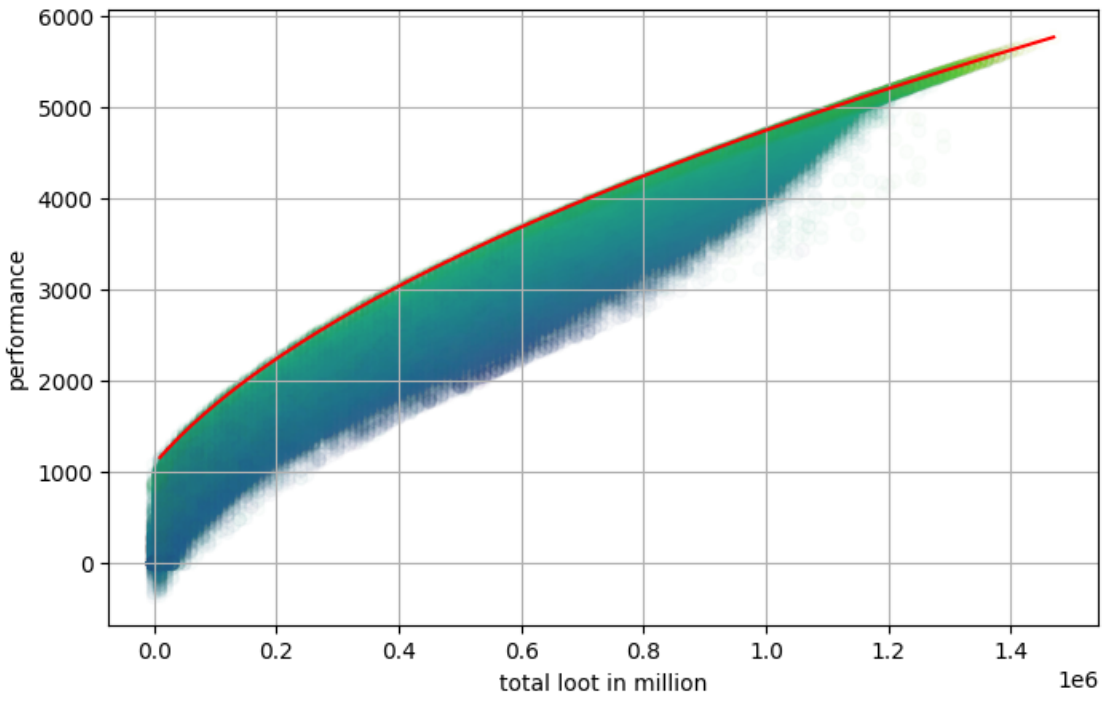
```
Mean Squared Error:  417.1471363945081
Mean R : 0.9997345248957292
```

Plot performance against total loot again now with the upper boundary we found

The color describes the difference between defensive and offensive average loot per attack

[12]:
```
fig, ax = plt.subplots(figsize=(8, 5))
ax.scatter(data['loot'], data['performance'],␣
 ↪c=data['avg_loot']-data['avg_def_loot'], alpha=0.02)
plt.plot(selection.index, top_y_model, 'r')
ax.grid()
ax.set_xlabel('total loot in million')
ax.set_ylabel('performance')
```

[12]:
```
Text(0, 0.5, 'performance')
```

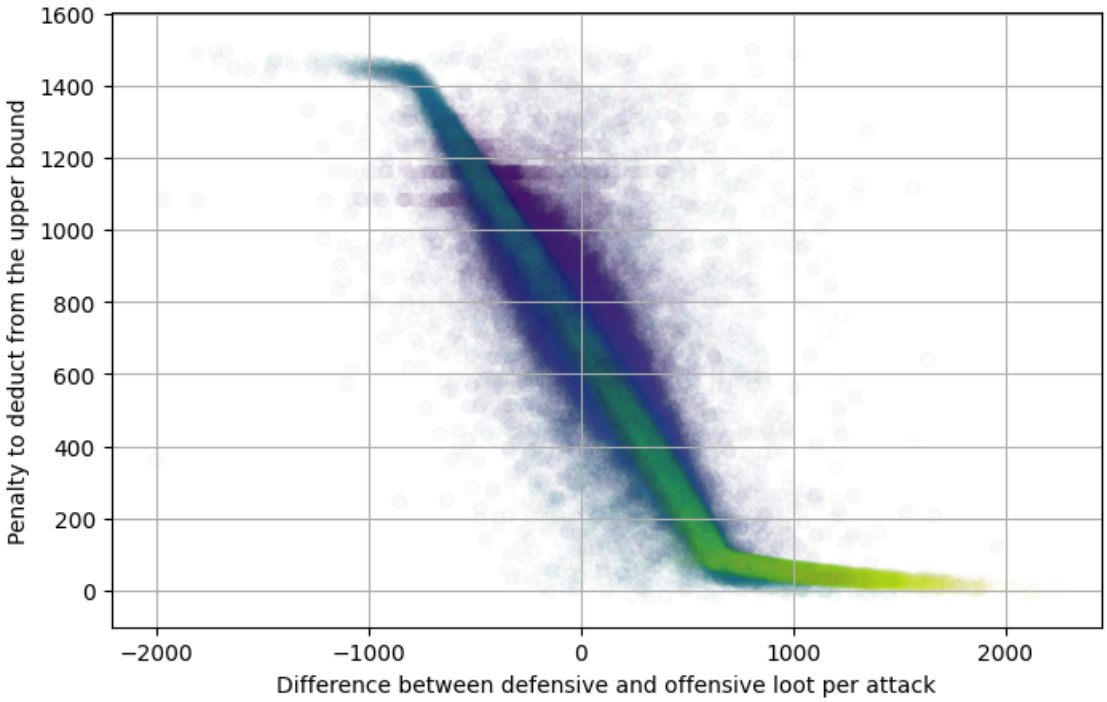


We need to figure out how to calculate the penalty that gets deducted from the upper bound

And when we plot this penalty against the difference between defensive and offensive loot per attack, we get the following (the lighter the dot, the higher the performance. There is a lot of diffusion towards the lower end of the performance).

[13]:
```
fig, ax = plt.subplots(figsize=(8, 5))
ax.scatter(data['avg_loot'] - data['avg_def_loot'], law(data['loot'],␣
 ↪*top_popt) - data['performance'], c=data['performance'], alpha=0.02)
ax.grid()
ax.set_xlabel('Difference between defensive and offensive loot per attack')
ax.set_ylabel('Penalty to deduct from the upper bound')
```

[13]:
```
Text(0, 0.5, 'Penalty to deduct from the upper bound')
```




Here we can clearly see three (almost) straight lines. And when we put these into code, we get the following algorithm for the performance prediction:

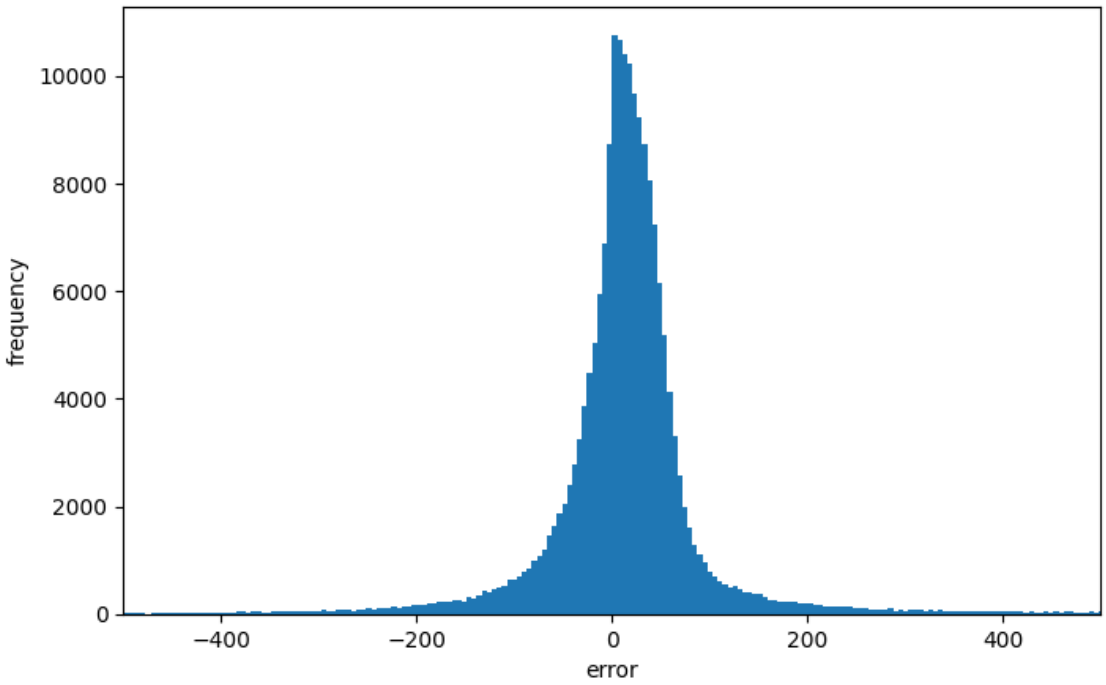
[14]:
```
def predict_performance(loot: float, avg_loot: float, avg_def_loot: float) ->␣
 ↪float:
    upper_bound = 5 * (loot + 100000) ** 0.5 - 500
    loot_difference = avg_def_loot - avg_loot
    deduction_center = loot_difference + 700
```

```
    deduction_bottom = (loot_difference + 2000) / 20
    deduction_top = loot_difference / 20 + 1400
    return max(upper_bound - max(min(max(deduction_center, deduction_bottom),␣
↪deduction_top), 0), 0)
```

Now lets see how good this prediction is and plot the errors

[15]:
```
data['errors'] = data['prediction'] - data['performance']
fig, ax = plt.subplots(figsize=(8, 5))
ax.hist(data['errors'], 500)
ax.set_xlim((-500, 500))
ax.set_xlabel('error')
ax.set_ylabel('frequency')
```

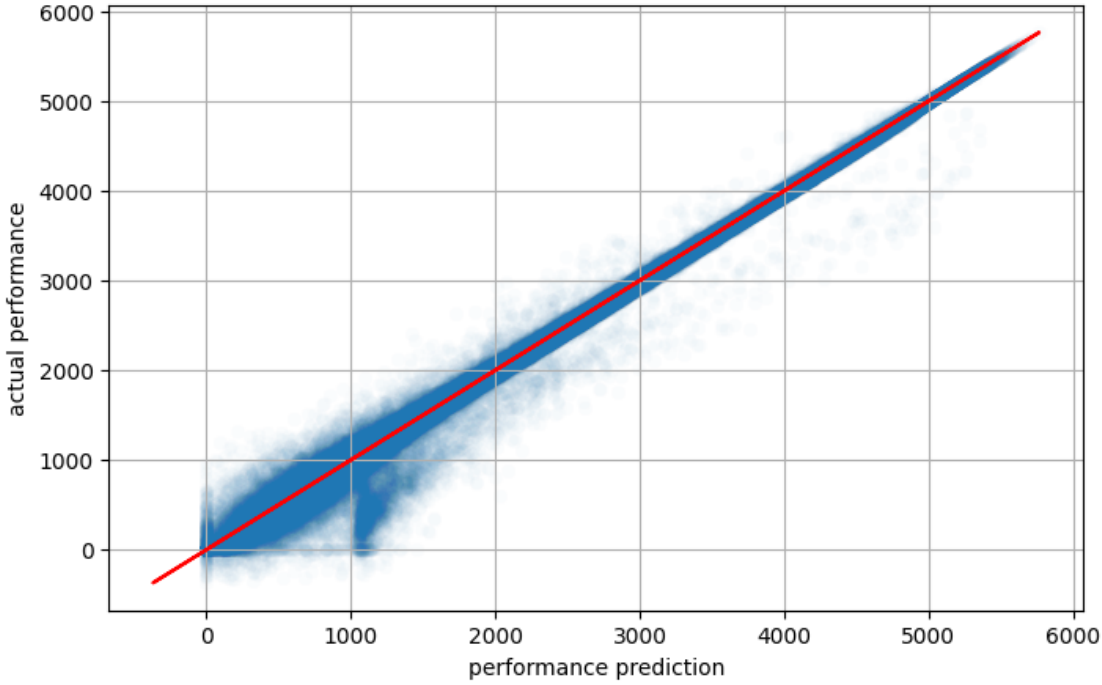
[15]:
```
Text(0, 0.5, 'frequency')
```




another way to see the quality of the prediction: plot it against the actual performance

[16]:
```
fig, ax = plt.subplots(figsize=(8, 5))
ax.scatter(data['prediction'], data['performance'], alpha=0.02, linewidths=0)
ax.plot(data['performance'], data['performance'], 'r')
ax.grid()
ax.set_xlabel('performance prediction')
ax.set_ylabel('actual performance')
```

[16]: Text(0, 0.5, 'actual performance')




Author: üüüüüüüüüüü#0374