Colonel Blotto game

February 2, 2019

The general idea of this approach is to model the entries by a normal distribution, assuming that human competitors strategies must have a bias towards castle 10, whose winning gains most of the points. The model is done with Python.

```
In [8]: import numpy as np
        import pandas as pd
        import scipy.stats
```

We create below a class 'Blotto', it has the 'create_data' method to build a sample data of competitor's entries according to the logic that the bias should have a normal behaviour. We set the parameters as follows: mean= 10. Sigma = 3 (For we assume entries 7-10, sums up together to 34, should have most of the probability, ~68% in fact). Our sample size is N = 1000, where each entry sums up to k=100.

The method "get_score" computes and returns the scores of two given entries. The method "get_average_score" computes all averages scores of all entries, returns the best 5 indices. The method "get_best_index" returns the winning entry. The method "find_avg_score(self,vec)" computes the average score of the given dataset df, and a given picked entry.

```
In [9]: class Blotto(object):
            def __init__(self, N,m,k=100,df= None):
                self.N = N
                self.m = m
                self.k = k
                self.df=df if df is not None else np.zeros((N,m))
                self.avg_score = None

            def create_data(self):
                """Creates data with a multinomial distribution and a Gaussian bias"""
                self.df=np.zeros((self.N,self.m))
                a = scipy.stats.norm(10, 3)
                b = a.pdf([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]) / a.pdf([1, 2, 3, 4, 5, 6, 7, 8, 9,
                for i in range(self.N):
                    self.df[i] = np.random.multinomial(self.k,b)
                return self.df

            def get_score(self,x,y):
                """method calculates and return scores of 2 given entries, x and y"""
                score_x=0
```

```
                score_y=0
                for i in range(self.m):
                    if x[i] > y[i]:
                        score_x += i+1
                    elif y[i] > x[i]:
                        score_y += i+1
                    else:
                        continue
                return score_x,score_y

            def get_average_score(self):
                """method calculates the best entry, returns 5 best indices"""
                avg_vec = np.zeros(self.N)
                for i in range(self.N):
                    for j in range(self.N):
                        avg_vec[i]+= self.get_score(self.df[i],self.df[j])[0]
                self.avg_score = avg_vec/(self.N-1)
                ind = np.argpartition(self.avg_score, -5)[-5:]
                return ind

            def get_best_index(self):
                """method to return the winning entry"""
                return self.avg_score[np.argmax(self.avg_score)],self.df[np.argmax(self.avg_sc

            def find_avg_score(self,vec):
                """compute average score for a given dataset and a given vector manually"""
                avg_vec = np.zeros(self.N)
                for i in range(self.N):
                    avg_vec[i]= self.get_score(self.df[i],vec)[1]
                return np.mean(avg_vec)
```

```
In [10]: model = Blotto(1000, 10)
         df = model.create_data()
```

```
In [11]: avg_score = model.get_average_score()
```

```
In [12]: print(df[avg_score])
         print(model.get_best_index())
```

```
[[ 0.  1.  2.  0.  4. 11.  7. 21. 26. 28.]
 [ 0.  1.  2.  2.  1.  4. 16. 20. 26. 28.]
 [ 0.  2.  0.  4.  7.  6. 17. 10. 25. 29.]
 [ 0.  1.  3.  5. 10. 14. 14.  9. 13. 31.]
 [ 0.  0.  3.  4.  6. 13.  7. 11. 28. 28.]]
(29.17117117117117, array([ 0.,  1.,  3.,  5., 10., 14., 14.,  9., 13., 31.]))
```

Thus, we got the 5 best entries of such competition, together with the best score: 29.1711711711711.

We describe the statistics of the sample data:

```
In [13]: print(pd.DataFrame(df).describe())
```

```
                0            1            2           3            4  \
count  1000.000000  1000.000000  1000.000000  1000.00000  1000.000000
mean      0.261000     0.679000     1.507000     3.13100     5.768000
std       0.497118     0.854804     1.182095     1.75183     2.376094
min       0.000000     0.000000     0.000000     0.00000     0.000000
25%       0.000000     0.000000     1.000000     2.00000     4.000000
50%       0.000000     0.000000     1.000000     3.00000     6.000000
75%       0.000000     1.000000     2.000000     4.00000     7.000000
max       3.000000     6.000000     6.000000     9.00000    15.000000

                5           6            7           8            9
count  1000.000000  1000.00000  1000.000000  1000.00000  1000.000000
mean      9.810000    14.16100    18.962000    22.03500    23.686000
std       3.059291     3.57937     3.947929     4.19993     4.286231
min       1.000000     4.00000     8.000000    10.00000    12.000000
25%       8.000000    12.00000    16.000000    19.00000    21.000000
50%      10.000000    14.00000    19.000000    22.00000    24.000000
75%      12.000000    16.25000    22.000000    25.00000    27.000000
max      20.000000    27.00000    31.000000    35.00000    35.000000
```

Using this piece of logic, we create our own entry for competing in the tournment, to be superior with respect to the sample.

```
In [23]: vec=np.asarray([1,1,2,7,9,14,18,22,26,0])
         print('Entry:',{i+1:vec[i] for i in range(10) })
```

```
Entry: {1: 1, 2: 1, 3: 2, 4: 7, 5: 9, 6: 14, 7: 18, 8: 22, 9: 26, 10: 0}
```

with average score:

```
In [24]: print(model.find_avg_score(vec))
```

```
35.988
```

Our allocation strategy is made by the following reasoning:

1.Equating our resources at each castle with the expected median or better at the 75%- level for a better gain.

2.We rule out castle 10, where we expect most players will put most of their resources, as non - beneficial, thus put a 0 there.

1 Allocating with 90 resources.

```
In [25]: vec=np.asarray([1,1,4,9,13,18,20,24,0,0])
```

```
In [26]: print('Entry:',{i+1:vec[i] for i in range(10) })
         print(model.find_avg_score(vec))

Entry: {1: 1, 2: 1, 3: 4, 4: 9, 5: 13, 6: 18, 7: 20, 8: 24, 9: 0, 10: 0}
32.99
```

2 Allocating with 110 resources

```
In [27]: vec=np.asarray([1,1,2,6,10,15,20,25,0,30])
         print('Entry:',{i+1:vec[i] for i in range(10) })

Entry: {1: 1, 2: 1, 3: 2, 4: 6, 5: 10, 6: 15, 7: 20, 8: 25, 9: 0, 10: 30}


In [29]: print(model.find_avg_score(vec))

40.157
```

We allocated our resources in both previous cases according to the rule that we wish to keep each castle in the 75% zone of our prediction, sacrifices expensives ones.