Risk Battle Simulation

June 4, 2017

This code aims to computationally study the statistics of risk in order to develop better strategies and beat the brothers at the game.

First we'll setup a dice class to abstract the dice rolling

```
In [10]: import random
         import numpy as np
         import matplotlib.pyplot as plt

         class Dice:
                 def __init__(self, dicemax, ndice):
                         self.dicemax = dicemax
                         self.ndice = ndice

                 def roll(self):
                         dice = []
                         for i in range(self.ndice):
                                 dice.append(random.randint(1,self.dicemax))
                         return sorted(dice, reverse=True)
```

Next, a player class to abstract the logic of the attacker and defender

```
In [11]: class Player:
                 # Then players can subclass to implement different strategies
                 def __init__(self, dice, number_of_people):
                         self.dice = dice
                         self.npeople = number_of_people

                 def roll(self):
                         return self.dice.roll()

                 def ready(self, opponent):
                         return True

         class Attacker(Player):
                 def roll(self):
                         if self.npeople in [3,2]:
                                 self.dice.ndice=2
                         if self.npeople ==1:
                                 self.dice.ndice=1
                         return Player.roll(self)

                 def ready(self, opponent):
                         return self.npeople > 1

         class Defender(Player):
                 def ready(self, opponent):
                         return self.npeople > 0
```

Finally, an interface to simulating a face off between two armies, here dubbed an *offensive*:

```
In [12]: class Offensive:
                 def __init__(self, attacker, defender, toprint=False):
                         self.a = attacker
                         self.d = defender
                         self.toprint = toprint

                 def extension(self,rollA, rollD):
                         return np.array([0,0])

                 def compareDice(self, diceA, diceD):
                         if diceA > diceD:
                                 return (0,-1)
                         elif diceA <= diceD:
                                 return (-1,0)

                 def standardBattleContract(self, rolla, rolld):
                         minind = min(len(rolla), len(rolld))
                         aligneda = rolla[:minind]
                         alignedd = rolld[:minind]
                         contract = map(
                                 lambda x: self.compareDice(*x),
                                 zip(aligneda, alignedd))
                         lossA = 0
                         lossD = 0
                         for c in contract:
                                 lossA += c[0]
                                 lossD += c[1]
                         return np.array([lossA, lossD])

                 def show(self, loss, rolla, rolld):
                         lossA, lossD = loss
                         print "Dice: "
                         print "A: {}".format(rolla)
                         print "D: {}".format(rolld)
                         print "   +++   "
                         print "Loss: "
                         print "A: {}".format(lossA)
                         print "B: {}".format(lossD)
                         print "   +++   "
                         print "Men:"
                         print "A: {}".format(self.a.npeople)
                         print "D: {}".format(self.d.npeople)
                         print "------------------"

                 def didWin(self):
                         return self.d.npeople==0

                 def canIterate(self):
                         can= self.a.ready(self.d) and self.d.ready(self.a)
                         return can

                 def iterate(self):
                         rolld = self.d.roll()
                         rolla = self.a.roll()
                         loss = self.extension(rolla, rolld) \
                     + self.standardBattleContract(rolla, rolld)
                         self.a.npeople += loss[0]
                         self.d.npeople += loss[1]
                         if self.toprint:
                                 self.show(loss, rolla,rolld)
```

We invented some custom rules in my house, so we'll add them here

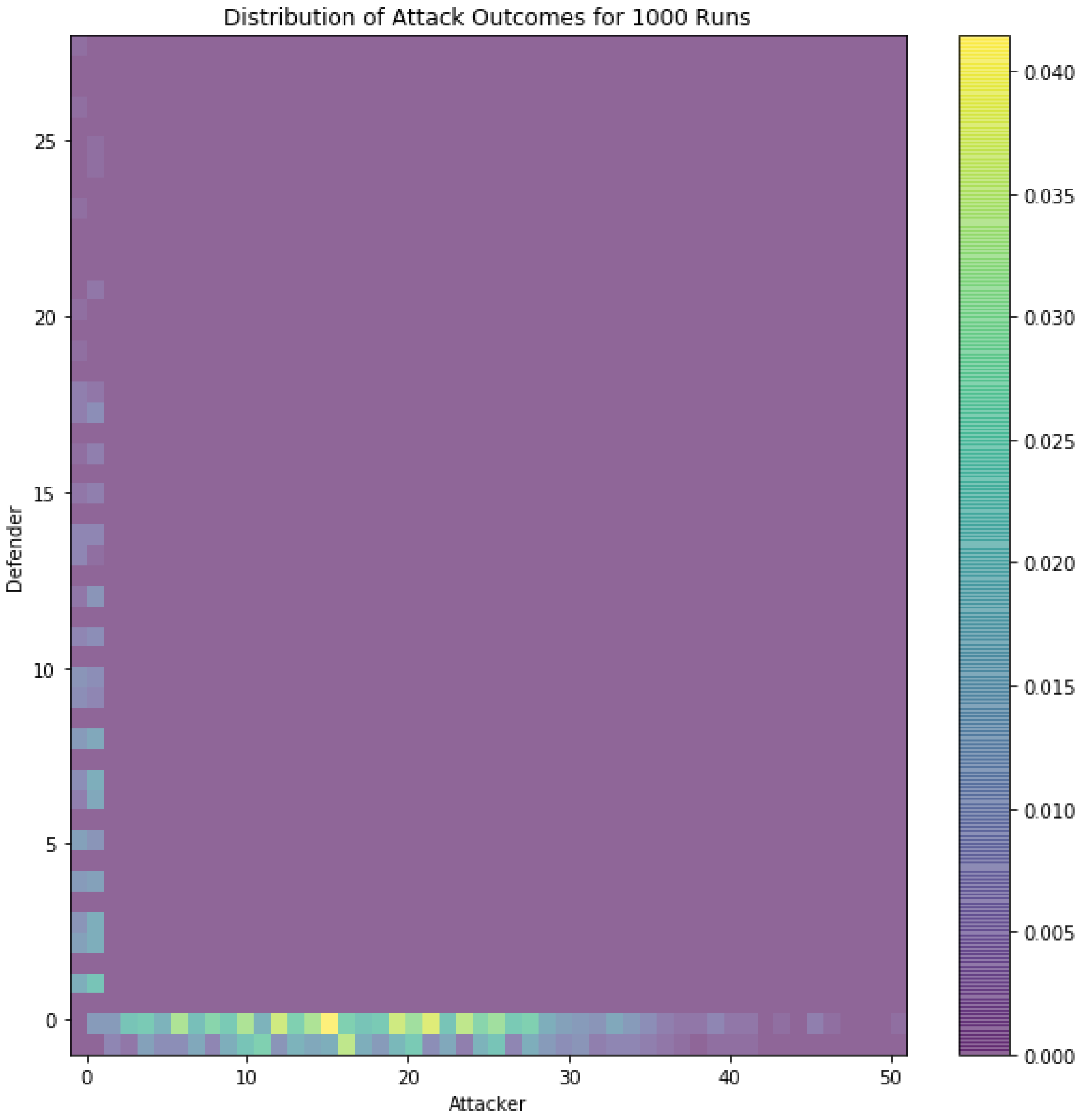
```
In [13]: class LipshitzianOffensive(Offensive):
                 def extension(self, rollA, rollD):
                         allgreater = min(rollD) >= max(rollA)
                         if allgreater:
                                 return np.array([-1,0])
                         else:
                                 return np.array([0,0])
```

Finally we can simulate

```
In [14]: def simulate():
                 battle = LipshitzianOffensive(
                         Attacker(Dice(6,3), 70),
                         Defender(Dice(6,2), 58),
                         toprint=False)

                 while battle.canIterate():
                         battle.iterate()

                 return (battle.a.npeople, battle.d.npeople)

         def main():
                 resA = []
                 resD = []
                 for i in range(1000):
                         ra, rd = simulate()
                         resA.append(ra)
                         resD.append(rd)

                 resA = np.array(resA)
                 resD = np.array(resD)

                 return resA, resD
```

```
In [20]: import matplotlib.pyplot as plt
         import numpy as np
         %matplotlib inline

         resA, resD = main()
         print "On average"
         print "Attacker"; print np.mean(resA)
         print "Defender"; print np.mean(resD)
         print "Attacker wins by"; print np.mean(resA - resD)
         plt.figure(figsize=(10,10))
         plt.hist2d(resA, resD, alpha=.6, normed=True, bins=[50,50])
         plt.colorbar()
         plt.ylabel("Defender")
         plt.title("Distribution of Attack Outcomes for 1000 Runs")
         plt.xlabel("Attacker")
         plt.savefig("./results.png")
```

```
On average
Attacker
14.15
Defender
1.406
Attacker wins by
12.744
```



```
In [16]: import networkx as nx

In [17]: class Board(nx.Graph):
             def edge(a, b):
                 self.add_edge(a,b)
                 return self
         board = Board()

In [18]: Map = {
             "North America":[
                 "Alaska",
                 "Northwestern Territory",
        "Alberta",
        "Ontario",
        "Eastern Canada",
        "Western United States",
        "Eastern United States",
        "Central America",
        "Greenland"],
    "South America":[
        "Venezuela",
        "Brazil",
        "Peru",
        "Argentina"],
    "Europe":[
        "Iceland",
        "Great Britain",
        "Western Europe",
        "Scandinavia",
        "Northern Europe",
        "Southern Europe",
        "Russia"],
    "Africa":[
        "North Africa",
        "Egypt",
        "East Africa",
        "Central Africa",
        "South Africa",
        "Madagascar"],
    "Asia":[
        "Ural",
        "Afghanistan",
        "Middle East",
        "India",
        "Southern Asia",
        "China",
        "Mongolia",
        "Irkutsk",
        "Siberia",
        "Yakutsk",
        "Kamachatka",
        "Japan"],
    "Australia":[
        "Indonesia",
        "New Guinea",
        "Western Australia",
        "Eastern Australia"
        ]
}
```

```
In [4]: for k, v in Map.iteritems():
            print "{}: {} countries".format(k,len(v))
            map(board.add_node, v)
```

```
Europe: 7 countries
Australia: 4 countries
Africa: 6 countries
Asia: 12 countries
North America: 9 countries
South America: 4 countries
```

```
In [ ]:
```