**Dear Admission Officer,**

**My name is Zhenyang Lu, I'm presenting writing sample as part of the MSCA application. I have been thinking about what topic I should talk about to you, but can't come up with any, as I'm not quite interested in movie, music or those sorts of things, actually I'm pretty busy with employment, and don't have much time for them. But I do have some academic papers done previously, one of which is about Monte Carlo simulation I'd like to present to you, because it's related to this program that I'm applying for. I wrote it four years ago when I was at DePaul, Computational Finance master program. It's my final project, which got me an A from Professor DiPierro. I really like this topic as it's a mix of my skills both at statistics and python programming, which is also what drives me to apply for MSCA program. I wish by reading it you will have a better understanding of my background. Thanks!**

**Regards,**

**Zhenyang**

**Final Project - CSC 521**

**Question:**

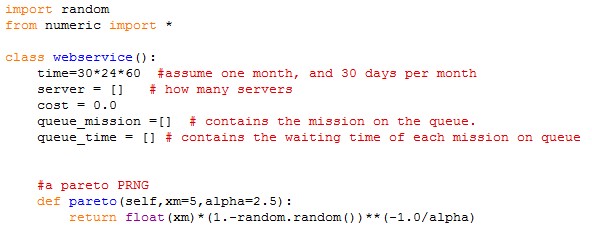
You run a web service that performs computations on behalf of your users. When a request arrives to your computers, you process it and you deliver an answer. Requests arrive at random intervals with an average of one per minute. Once a request arrives, if you have a computer server available to process it, you assign it and it will complete in a random time given by the Pareto distribution with alpha = 2.5 and Xm = 5, unit in minutes. If the request arrives and all your servers are busy it will be queued. Each server can deal with one request at the time. One request can be queued but for no longer than 5minutes. If a request is queued for more than 5 minutes it is dropped. You will have to pay $10 in penalty for each dropped request.

-----------------------------------------------------------------------------------------------------------------------------

**Answer:**

**1.0 Analysis**

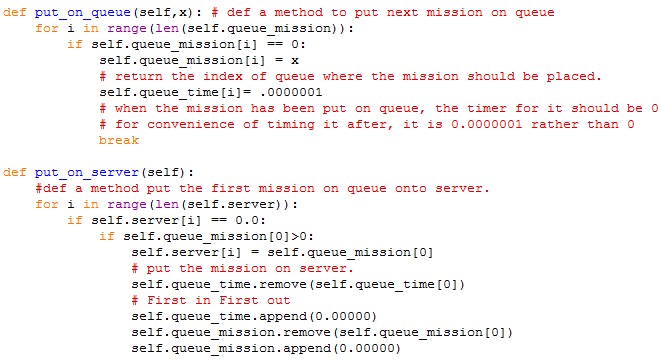
First, let me construct a class that simulate this web service company called **class webservice()** and define random Pareto random number generator, called **Pareto()**,inside the class:



In the previous code, I am simulating the cost for a month rather than a year to speed up the program. By creating two lists, **queue\_mission=[]** and **queue\_time=[],** I am assuming that these two lists act like a two-dimension array, in which the first dimension contains the mission, the second contains the waiting time for the mission.

What does "mission" mean in my code? Every mission that is assigned to the **queue\_mission=[]** is a Pareto random number with alpha = 2.5 and Xm = 5, unit in minutes. When "the mission" is passed to the **server=[]** list, the "mission" can be directly subtracted by how many minutes it has taken( **for simulate\_once()** )in this case, I simulate 43200 minutes, length of a month).

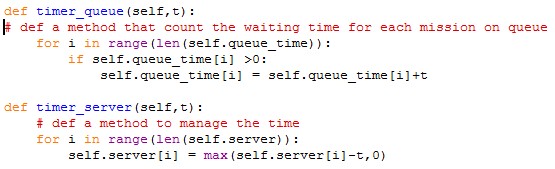
After creating lists of server, **queue\_time** and **queue\_mission**, I define two methods in the class that imitate behaviors of putting a mission in the queue and putting a mission from the queue into the server by the following code:



The **put\_on\_queue()** method checks which spot on the queue is empty and then put the mission from client onto the queue.

The **put\_on\_server()** method checks which server is available by testing which server is "zero", which I assume is the sign of "available" and import the mission from the queue to the server.

Finally, I still have to define two methods to calculate the waiting time for missions on servers and queue. These two behaviors can be done by code:

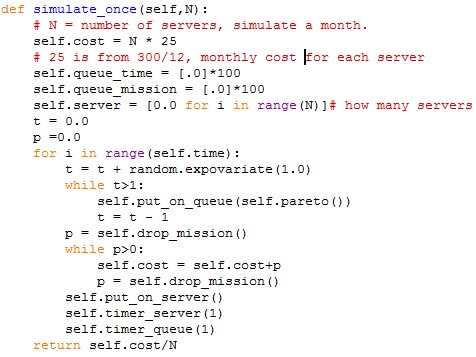


Every minute I simulate, the mission on server will be reduced by one and the mission on queue should be added by one. However, I should point out that since the initial numbers of **queue\_time=[]** are zero, I should only add one to those that larger than zero rather than adding one to each element on **queue\_time[].** Also, the mission on server cannot be lower than zero, which indicates that the mission has been solved, so the minimum for each element on server is zero.

Till now, I have constructed the main code to behave this web service company. All the methods in the **webservice()** class are centered on the three lists, **queue\_time, queue\_mission** and **server**. They all do certain things to the three lists.

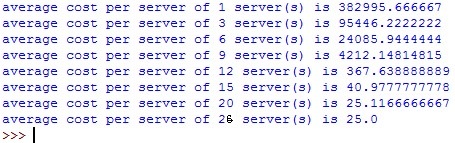
**1.1 Simulate\_once()**

By doing **simulate\_once(),** I want to simulate a month's cost. This can be done by code:



The argument, **N**, that **simulate\_once()** takes is how many servers you want to put into simulation. The lines before the **for** loop are initializations for the global parameters of the class. The first **while** loop pass the upcoming mission(s) onto the queue **(queue\_mission=[]**). The exponential variable **t** determines how many missions has/have come, when **t** is larger than 1, it comes one mission, when **t** is larger than 2, it comes two missions and etc., The second **While** loop indicates how many mission(s) on the queue has/have been waiting for 5 minutes or more and how much is the dropping cost, which has to be added to the total cost (global parameter of **self.cost**). This **simulate\_once()** method returns the average cost for each server if the total number of servers is **N.**

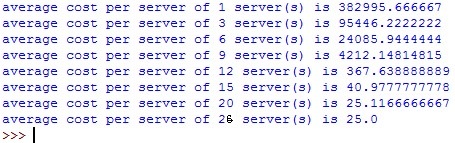
By simulating different numbers of servers, we can get the inference of how many servers is ideal in order to minimize the cost. The result is like (ignore the client code for realizing the result):



From this, we can see that when the number of servers comes to 20-25, the average cost per server is 25, which is the cost for only the fixed cost for each server. One thing I should point out is that since the result of **simulate\_once()** fluctuates a lot, there may be chances that the average cost per server occurs 25 when **N** is around 15, which indicates that sometimes 15 servers are enough to maximize your profit. Due to the slowness of running the **simulate\_once()** method, I will not provide all the data to prove this conclusion. This can be shown by the following list, which contains data for **simulate\_once(N)** with a parameter of 15: You could vaguely see there exists 25 on the first line. Also, there will be odds that 25, which is the lowest average cost in this simulation, happens when **N** is lower than 15.

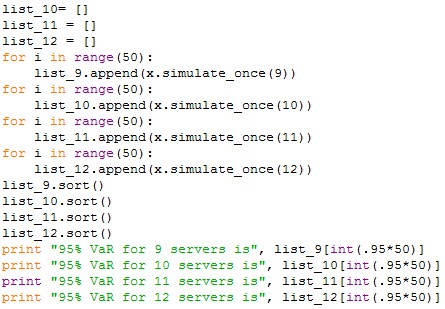
**1.2 95% VaR less than $10,000**

Since I have already got the result of **simulate\_once()** with argument **N** ranging from 1 to 26, I could simply solve this problem by enumerating **N** starting from 9 to 20 in order to get the cutting number of **N** such that 95% VaR is less than $10,000 rather than make a method taking **N** as argument and return the 95% VaR. Since the following graph, which we get in 1.1 section,



has shown that when **N** is 9, total cost is 9\*4212, 37980, and when **N** is 12, total cost is 12\*367, 4404, I **assume** that the **N** that makes 95% VaR less than $10,000 is possibly between 9 - 12 and above 400. However, for the second case, say **N** is 400, although the total cost is 400\*25 (25 is monthly cost for each server), your total profit is certainly positive when you take how much the client pay for each mission in consideration. So I will **presumably ignore** this case.

To speed up my program, for each **N**, I simulate 50 times and store and sort the data. The code for this part is like:



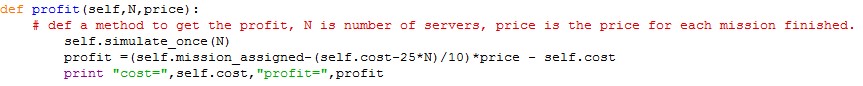
To further infer the data, I plot the lists for 9 -12 servers. We could see the following lines plot:

The vertical axis is loss and the red line is 95% VaR. Combined with the two graphs above, we could see that if you have 12 servers, it is definitely you will have a 95% VaR less than $10,000. However, since this experiment has only been done for 50 times, if you expand the times of your simulation for N equal to 11, you will also be likely to have a 95% VaR that is less than $10,000. From the lines plot we could see that the 45th data of the sorted list, shown by green line, is around $10,000. This further proves that 11 servers are also able to satisfy a 95% VaR which is less than $10,000, definitely not a very solid conclusion.

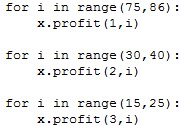
**1.3 Breakeven Point for profit**

Since for different number of servers, **N**, the price for breakeven points is different, I should test different pricesfor each **N.** That is how I do this section.

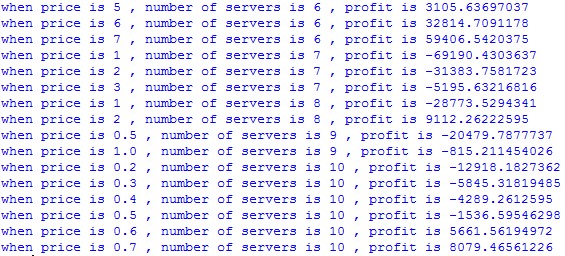
First, let define a method that take **N** and **price** as arguments, **price** means what is the price for each mission assigned. **N** is still number of traders. Code is like:



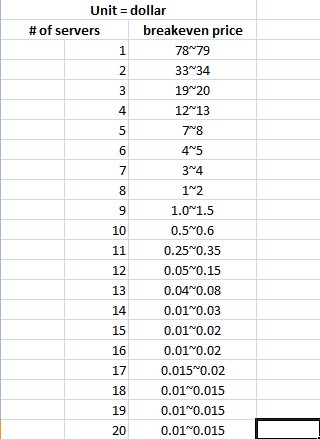
Then I just use this method to test the breakeven point with such codes as:



The result is like (non-exhausted):



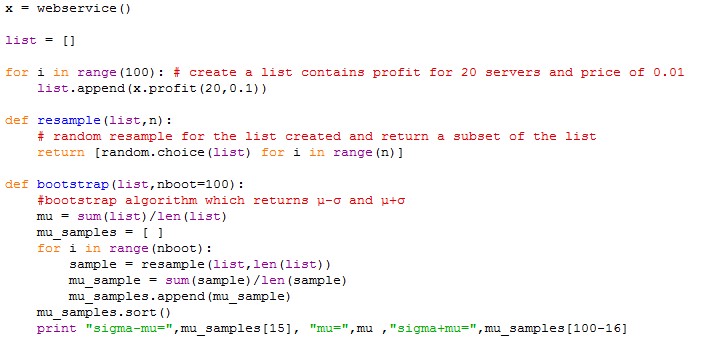
From these data, you can record which prices makes the profit larger than zero, then I just compile these data, the graph of breakeven points is like (using excel):



Even if you continue increasing number of traders, which is an argument for the code, the breakeven point cannot be higher. Since the total number of mission is fixed. Your total profit cannot be increased by adding the number of servers.

**1.4 computing average profit using bootstrap algorithm**

I **assume** that the best number of servers, **N**, is 20 and price for each mission is 0.1 as an example. Then I just compute the average profit ranging from the mean minus standard deviation and the mean plus standard deviation. The code for this part of answer is like:



The result is like:

C:\Users\felix\Desktop\csc521\final\16.jpg

Sigma is around 1~3.

**1.5 Business discussion**

**Introduction:** This is a really good venture investment project. In this case, each server cost you $25 per month. You receive mission from client and solve it using your server. Expectedly, you will receive around 1400-1500 missions per day and each mission cost a server around 9 minutes to solve. If a mission cannot be solved within 5 minutes, you will loss $10 for each you dropped.

**Potential risk:** if the number of servers is lower, you will possibly loss money because those numbers of servers cannot deal with so many mission, so you probably losing money by dropping mission and accusing penalty. If you want to keep your potential risk and do not want to bankrupt with a confidence of 95% that do not loss money by $10,000, you will have to at least rent 12 servers.

**Promising benefits:** according to my analysis above, the potential profit for this project is very high. If you could rent around 15-20 servers, your average cost for each server is just the rent you pay. And if your client agreed with a price of $0.02 per mission you done, you will almost never loss your money. Since the breakeven price for you to balance your cost with your profit is $0.015~0.02 per mission. Let me give you a brief example of how much money you will earn, assume you charge $0.1 per mission you solved and you have 20 servers, your total profit is around $3817 and $3817 dollars per month, 68% possible!

**My suggestion:** if you want to invest your capital into this project, make sure that you have more than 15-20 servers, since this will keep you from losing money. 15-20 servers are also the best number of servers you will need to rent, because the total expected number of missions your servers will receive is just 43200 per day. If you hold more than 20 servers, some of your servers will not be used efficiently or cannot be used at all.

**2.0 Complete Code of the project**

import random

class webservice():

time=30\*24\*60

#assume one month, and 30 days per month

server = []

# how many servers

cost = 0.0

mission\_assigned = 0.0

queue\_mission =[]

# contains the mission on the queue.

queue\_time = []

# contains the waiting time of each mission

#a pareto PRNG

def pareto(self,xm=5,alpha=2.5):

return float(xm)\*(1.-random.random())\*\*(-1.0/alpha)

def put\_on\_queue(self,x): # def a method to put next mission on queue

for i in range(len(self.queue\_mission)):

if self.queue\_mission[i] == 0:

self.queue\_mission[i] = x

# return the index of queue where the mission should be placed.

self.queue\_time[i]= .0000001

break

def timer\_queue(self,t):

# def a method that calculate the waiting time of the missions on queue

for i in range(len(self.queue\_time)):

if self.queue\_time[i] >0:

self.queue\_time[i] = self.queue\_time[i]+t

def put\_on\_server(self):

def a method put the first mission on queue onto server.

for i in range(len(self.server)):

if self.server[i] == 0.0:

if self.queue\_mission[0]>0:

self.server[i] = self.queue\_mission[0]

# put the mission on server.

self.queue\_time.remove(self.queue\_time[0])

# First in First out

self.queue\_time.append(0.00000)

self.queue\_mission.remove(self.queue\_mission[0])

self.queue\_mission.append(0.00000)

def timer\_server(self,t):

# def a method to manage the time

for i in range(len(self.server)):

self.server[i] = max(self.server[i]-t,0)

def drop\_mission(self):

# def a method to check whether the waiting time of the mission has come up to 5 minutes.

cost = 0.0

for i in range(len(self.queue\_time)):

#to shift the behind mission to fron spots to replace the one(s) that have past 5 minutes

if self.queue\_time[i] > 5:

cost = cost + 10

self.queue\_time.remove(self.queue\_time[i])

self.queue\_time.append(0.0000)

self.queue\_mission.remove(self.queue\_mission[i])

self.queue\_mission.append(0.0000)

return cost

def simulate\_once(self,N):

# N = # of servers, simulate a month.

self.cost = N \* 25

# 25 is from 300/12, which is monthly cost for each server

self.queue\_time = [.0]\*100

self.queue\_mission = [.0]\*100

self.server = [0.0 for i in range(N)]

# how many servers

self.mission\_assigned = 0.0

t = 0.0

p =0.0

m =0.0

for i in range(self.time):

m = random.expovariate(1.0)

t = t + m

self.mission\_assigned += m

while t>1:

self.put\_on\_queue(self.pareto())

t = t - 1

p = self.drop\_mission()

while p>0:

self.cost = self.cost+p

p = self.drop\_mission()

self.put\_on\_server()

self.timer\_server(1)

self.timer\_queue(1)

return self.cost

##to compute average cost per server, use self.cost/N instead.

def profit(self,N,price):

# def a method to get the profit, N is number of servers, price is the price for each mission finished.

self.simulate\_once(N)

profit =(self.mission\_assigned-(self.cost-25\*N)/10)\*price - self.cost

return profit

##print "when price is",price,",","number of servers is",N,",", "profit is",profit

x = webservice()

list = []

for i in range(100): # create a list contains profit for 20 servers and price of 0.01

list.append(x.profit(20,0.1))

def resample(list,n):

# random resample for the list created and return a subset of the list

return [random.choice(list) for i in range(n)]

def bootstrap(list,nboot=100):

#bootstrap algorithm which returns sigma-mu and sigma+mu

mu = sum(list)/len(list)

mu\_samples = [ ]

for i in range(nboot):

sample = resample(list,len(list))

mu\_sample = sum(sample)/len(sample)

mu\_samples.append(mu\_sample)

mu\_samples.sort()

print "sigma-mu=",mu\_samples[15], "mu=",mu ,"sigma+mu=",mu\_samples[100-16]