Homework for the short course Short Course on Stochastic Programming and Risk Management

Name: Jiangyi

Family name: Huang

Email: [Jiangyi.huang@vtt.fi](mailto:Jiangyi.huang@vtt.fi)

Research Scientist at VTT Technical Research Centre of Finland Ltd.

Programming language: Python 3.7.6

The sample problem is modified for this homework as shown in the table below:

|  |  |  |  |
| --- | --- | --- | --- |
| Contract price [$/MWh] | 45 | Contract capacity [MW] | 90 |
| Scenario | Probability | Demand [MW] | Price [$/MWh] |
| 1 | 0.15 | 120 | 59 |
| 2 | 0.75 | 100 | 46 |
| 3 | 0.1 | 80 | 44 |
| \*the figures labelled red are those changed | | | |

1. Scenario formulation and EVPI

The code representing the scenario formulation of the problem, along with the problem to calculate the EVPI, can be found below:

import numpy as np  
from scipy.optimize import minimize  
# problem conditions  
demand\_1 = 120; demand\_2 = 100; demand\_3 = 80  
prob\_1 = 0.15; prob\_2 = 0.75; prob\_3 = 0.1  
price\_1 = 59; price\_2 = 46; price\_3 = 44  
price\_c = 45; contract\_max = 90.0  
# define objective function  
def objective(p):  
 p\_c1, p\_c2, p\_c3, p1, p2, p3 = p  
 sce\_1 = np.dot([price\_c,price\_1],[p\_c1,p1])  
 sce\_2 = np.dot([price\_c,price\_2],[p\_c2,p2])  
 sce\_3 = np.dot([price\_c,price\_3],[p\_c3,p3])  
 obj = 24\*7\*np.dot([prob\_1, prob\_2, prob\_3],[sce\_1,sce\_2,sce\_3])  
 return obj  
# define constraints  
def constraint1(p):  
 return p[0]+p[3]-demand\_1  
def constraint2(p):  
 return p[1]+p[4]-demand\_2  
def constraint3(p):  
 return p[2]+p[5]-demand\_3  
  
# p\_c1=p\_c2=p\_c3  
def constraint\_c12(p):  
 return p[0]-p[1]  
def constraint\_c23(p):  
 return p[1]-p[2]  
  
# initial guesses  
n = 6  
p0 = np.zeros(n)  
for i in range(len(p0)):  
 p0[i] = 80.0  
# show initial objective  
print('Initial Objective: ' + str(round(objective(p0),0)))  
  
# set constraints/bounds  
b\_c = (0.0, contract\_max)  
b\_pos = (0.0, None)  
con1 = {'type': 'ineq', 'fun': constraint1}  
con2 = {'type': 'ineq', 'fun': constraint2}  
con3 = {'type': 'ineq', 'fun': constraint3}  
con\_c12 = {'type': 'eq', 'fun': constraint\_c12}  
con\_c23 = {'type': 'eq', 'fun': constraint\_c23}  
cons = [con1,con2,con3,con\_c12,con\_c23]  
bnds = (b\_c, b\_c, b\_c, b\_pos, b\_pos, b\_pos)  
# optimize  
solution = minimize(objective,p0,method='SLSQP',bounds=bnds, constraints=cons)  
p = solution.x  
for i in range(len(p)):  
 p[i]=round(p[i],0)  
# show final objective  
print('Final Objective: ' + str(round(objective(p),0)))  
  
# print solution  
print('Solution')  
print('p\_c1 = ' + str(p[0]))  
print('p1 = ' + str(p[3]))  
print('p\_c2 = ' + str(p[1]))  
print('p2 = ' + str(p[4]))  
print('p\_c3 = ' + str(p[2]))  
print('p3 = ' + str(p[5]))  
  
# calculate EVPI  
  
# stochastic solution  
z\_s = objective(p)  
# EVPI problem (with perfect infomation) -- relax the non-anticipativity constrainsts  
cons = [con1,con2,con3]  
# optimize  
solution = minimize(objective,p0,method='SLSQP',bounds=bnds, constraints=cons)  
p = solution.x  
for i in range(len(p)):  
 p[i]=round(p[i],0)  
print('Final Objective: ' + str(round(objective(p),0)))  
print('Solution given perfect information')  
print('p\_c1 = ' + str(p[0]))  
print('p1 = ' + str(p[3]))  
print('p\_c2 = ' + str(p[1]))  
print('p2 = ' + str(p[4]))  
print('p\_c3 = ' + str(p[2]))  
print('p3 = ' + str(p[5]))  
  
# perfect info. solution  
z\_p = objective(p)  
  
# calculate EVPI value  
EVPI\_min = z\_s - z\_p  
print('EVPI value: ' + str(EVPI\_min))

And the results:

Solution of scenario-formulation programming

Final Objective: 780192.0

p\_c1 = 80.0, p1 = 40.0

p\_c2 = 80.0, p2 = 20.0

p\_c3 = 80.0, p3 = 0.0

Solution when perfect information is given

Final Objective: 774060.0

p\_c1 = 90.0, p1 = 30.0

p\_c2 = 90.0, p2 = 10.0

p\_c3 = 0.0, p3 = 80.0

EVPI value: 6132.0

The EVPI shows the value for having, willingness to pay for, perfect information of the scenarios (the future) in this problem is 6132 $.

1. Node formulation

The code representing the node formulation of the problem is shown below, where the VSS and CVaR are also calculated:

import numpy as np  
from scipy.optimize import minimize  
import matplotlib.pyplot as plt  
# problem conditions  
demand\_1 = 120; demand\_2 = 100; demand\_3 = 80  
prob\_1 = 0.15; prob\_2 = 0.75; prob\_3 = 0.1  
price\_1 = 59; price\_2 = 46; price\_3 = 44  
price\_c = 45; contract\_max = 90.0  
  
demands = [demand\_1, demand\_2, demand\_3]  
probs = [prob\_1, prob\_2, prob\_3]  
prices = [price\_1, price\_2, price\_3]  
prices\_matrix = np.identity(len(prices))  
np.fill\_diagonal(prices\_matrix, prices)  
# define objective function  
def objective(p):  
 p\_c, p1, p2, p3 = p  
 stg\_1 = price\_c\*p\_c  
 stg\_2 = np.dot(np.dot(probs, prices\_matrix), [p1, p2, p3])  
 obj = 24\*7\*(stg\_1+stg\_2)  
 return obj  
# define constraints  
def constraint1(p):  
 return p[0]+p[1]-demand\_1  
def constraint2(p):  
 return p[0]+p[2]-demand\_2  
def constraint3(p):  
 return p[0]+p[3]-demand\_3  
# initial guesses  
p0 = np.array([80, 40, 20, 0])  
  
# show initial objective  
print('Initial Objective: ' + str(objective(p0)))  
  
# set constraints/bounds  
b\_c = (0.0, contract\_max); b\_pos = (0.0, None); b\_open = (None, None)  
con1 = {'type': 'ineq', 'fun': constraint1}  
con2 = {'type': 'ineq', 'fun': constraint2}  
con3 = {'type': 'ineq', 'fun': constraint3}  
# con2 = {'type': 'eq', 'fun': constraint2}  
cons = [con1,con2,con3]  
bnds = (b\_c, b\_pos, b\_pos, b\_pos)  
# optimize  
solution = minimize(objective,p0,method='SLSQP',bounds=bnds, constraints=cons)  
p = solution.x  
for i in range(len(p)):  
 p[i]=round(p[i],0)  
  
# print solution  
print('Solution Node-formulation')  
print('Final Objective: ' + str(objective(p)))  
print('p\_c = ' + str(p[0]))  
print('p1 = ' + str(p[1]))  
print('p2 = ' + str(p[2]))  
print('p3 = ' + str(p[3]))  
  
## calculate VSS  
# stochastic solution  
z\_s = objective(p)  
# VSS problem  
# stage 1 optimal value for the first stage variable p\_c  
def objective\_avg(p):  
 avg\_price = np.dot(probs, prices) # sigma\_i(prob\_i\*price\_i)  
 stg\_1 = price\_c\*p[0]  
 stg\_2 = avg\_price\*p[1]  
 obj = 24\*7\*(stg\_1+stg\_2)  
 return obj  
# define constraints  
avg\_demand = np.dot(probs, demands)  
def constraint1(p):  
 return p[0]+p[1]-avg\_demand  
# initial guesses  
p0 = np.array([90,20])  
  
# set constraints/bounds  
con1 = {'type': 'ineq', 'fun': constraint1}  
bnds = (b\_c, b\_pos)  
# optimize  
cons = [con1]  
solution = minimize(objective\_avg,p0,method='SLSQP',bounds=bnds, constraints=cons)  
p = solution.x  
  
# print solution  
print('Solution Deterministic Problem 1')  
print('p\_cd = ' + str(p[0]))  
p\_cd = p[0]  
  
# stage 2  
def objective\_VSS(p):  
 p1, p2, p3 = p  
 stg\_1 = price\_c\*p\_cd  
 stg\_2 = np.dot(np.dot(probs, prices\_matrix), [p1, p2, p3])  
 obj = 24 \* 7 \* (stg\_1 + stg\_2)  
 return obj  
# define constraints  
def constraint1(p):  
 return p\_cd + p[0] - demand\_1  
def constraint2(p):  
 return p\_cd + p[1] - demand\_2  
def constraint3(p):  
 return p\_cd + p[2] - demand\_3  
# initial guesses  
n = 3  
p0 = np.zeros(n)  
for i in range(len(p0)):  
 p0[i] = 30.0  
# set constraints/bounds  
con1 = {'type': 'ineq', 'fun': constraint1}  
con2 = {'type': 'ineq', 'fun': constraint2}  
con3 = {'type': 'ineq', 'fun': constraint3}  
cons = [con1,con2,con3]  
bnds = (b\_pos, b\_pos, b\_pos)  
# optimize  
solution = minimize(objective\_VSS,p0,method='SLSQP',bounds=bnds, constraints=cons)  
p = solution.x  
for i in range(len(p)):  
 p[i]=round(p[i],0)  
  
# print solution  
print('Solution Deterministic Problem 2')  
print('Final Objective: ' + str(objective\_VSS(p)))  
print('p1 = ' + str(p[0]))  
print('p2 = ' + str(p[1]))  
print('p3 = ' + str(p[2]))  
  
# deterministic solution  
z\_d = objective\_VSS(p)  
  
# calculate VSS value  
VSS\_min = z\_d - z\_s  
print('VSS value: ' + str(VSS\_min))  
  
## calculate CVaR  
def objective\_CVaR(p, a=0, b=0):  
 p\_c, p1, p2, p3, s1, s2, s3, yita = p  
 stg\_1 = price\_c\*p\_c  
 stg\_2 = np.dot(np.dot(probs, prices\_matrix), [p1, p2, p3])  
 profit = 0-24\*7\*(stg\_1 + stg\_2)  
 max\_obj = (1-b)\*profit+b\*(yita-(1/(1-a))\*np.dot(probs, [s1, s2, s3]))  
 min\_obj = -max\_obj  
 return min\_obj  
# define constraints  
def constraint1(p):  
 return p[0] + p[1] - demand\_1  
def constraint2(p):  
 return p[0] + p[2] - demand\_2  
def constraint3(p):  
 return p[0] + p[3] - demand\_3  
  
def constraint\_s1(p):  
 profit\_1 = 0-24\*7\*np.dot([price\_c,price\_1],[p[0],p[1]])  
 return profit\_1 - p[7] + p[4]  
def constraint\_s2(p):  
 profit\_2 = 0-24\*7\*np.dot([price\_c,price\_2],[p[0],p[2]])  
 return profit\_2 - p[7] + p[5]  
def constraint\_s3(p):  
 profit\_3 = 0-24\*7\*np.dot([price\_c,price\_3],[p[0],p[3]])  
 return profit\_3 - p[7] + p[6]  
# initial guesses  
p0 = np.array([80,40,20,0,0,0,0,0])  
  
# set constraints/bounds  
con1 = {'type': 'ineq', 'fun': constraint1}  
con2 = {'type': 'ineq', 'fun': constraint2}  
con3 = {'type': 'ineq', 'fun': constraint3}  
con4 = {'type': 'ineq', 'fun': constraint\_s1}  
con5 = {'type': 'ineq', 'fun': constraint\_s2}  
con6 = {'type': 'ineq', 'fun': constraint\_s3}  
cons = [con1,con2,con3,con4,con5,con6]  
bnds = (b\_c, b\_pos, b\_pos, b\_pos, b\_pos, b\_pos, b\_pos, b\_open)  
  
# set alpha and beta  
alpha = 0.75; beta = 0  
# optimize  
solution = minimize(objective\_CVaR, p0, args=(alpha, beta), method='SLSQP', bounds=bnds, constraints=cons)  
p = solution.x  
p\_c, p1, p2, p3, s1, s2, s3, yita = p  
  
# print solution  
print('Solution CVaR')  
print('Final Objective: ' + str(objective\_CVaR(p)))  
print('alpha = '+str(alpha)+'\tbeta='+str(beta))  
print('yita = '+str(round(yita,0)))  
print('p\_c = '+str(round(p\_c,0)))  
  
def print\_scenario(i):  
 profit = 0-24\*7\*np.dot([price\_c, prices[i-1]], [p\_c, p[i]])  
 print('scenario {0}: '.format(i)+'p{0}='.format(i)+str(round(p[i],0))  
 +'\tprob.='+str(probs[i-1])  
 +'\tprofit='+str(round(profit,0))  
 +'\ts{0}='.format(i)+str(max(yita-profit,0)))  
 return max(yita-profit,0),profit  
  
n=4  
s = np.zeros(n-1)  
profits = np.zeros(n-1)  
for i in range(1,n):  
 s[i-1], profits[i-1] = print\_scenario(i)  
  
CVaR = yita-1/(1-alpha)\*np.dot(probs,s)  
Exp\_profit = float(np.dot(probs, profits))  
print('CVaR='+str(round(CVaR,0))+',\tExp\_profit='+str(round(Exp\_profit,0)))

Results:

Solution of node-formulation programming

Final Objective: 780192.0

p\_c = 80.0, p1 = 40.0, p2 = 20.0, p3 = 0.0

Solution for VSS value calculation

Deterministic Problem 1

p\_cd = 90.0

Deterministic Problem 2

Final Objective: 782963.0

p1 = 30.0, p2 = 10.0, p3 = 0.0

VSS value: 2771.0

The VSS value here indicates that the gain from modelling random variables as such rather than the average values is equivalent to 2771$.

Solution for CVaR calculation

alpha = 0.75 beta=0

Final Objective: 782963.0

yita = -603965.0

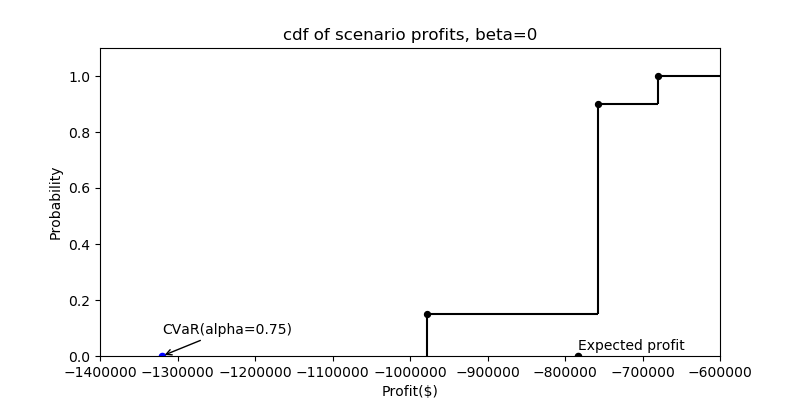
p\_c = 90.0

scenario 1: p1=30.0 prob.=0.15 profit=-977768.0 s1=373803.09777431306

scenario 2: p2=10.0 prob.=0.75 profit=-757681.0 s2=153715.69083398953

scenario 3: p3=0.0 prob.=0.1 profit=-680374.0 s3=76409.48181822104

CVaR=-1319958.0, Exp\_profit=-782963.0



alpha = 0.75 beta=0.5

Final Objective: 780191.9733012099

yita = -759360.0

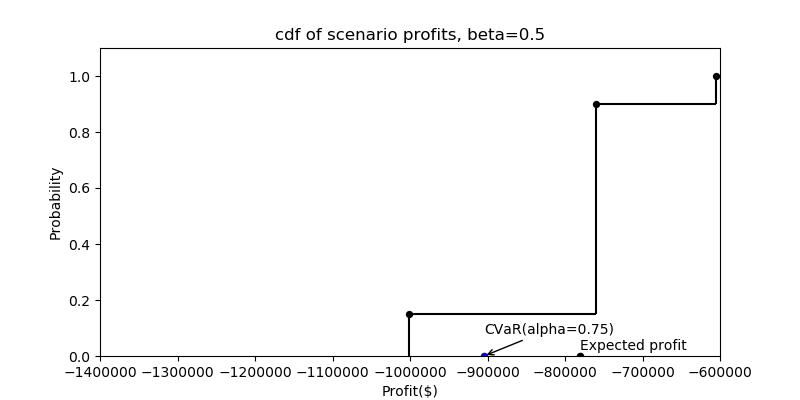
p\_c = 80.0

scenario 1: p1=40.0 prob.=0.15 profit=-1001280.0 s1=241919.97878228896

scenario 2: p2=20.0 prob.=0.75 profit=-759360.0 s2=0

scenario 3: p3=0.0 prob.=0.1 profit=-604800.0 s3=0

CVaR=-904512.0, Exp\_profit=-780192.0



alpha = 0.75 beta=1

Final Objective: 782964.0000005893

yita = -757680.0

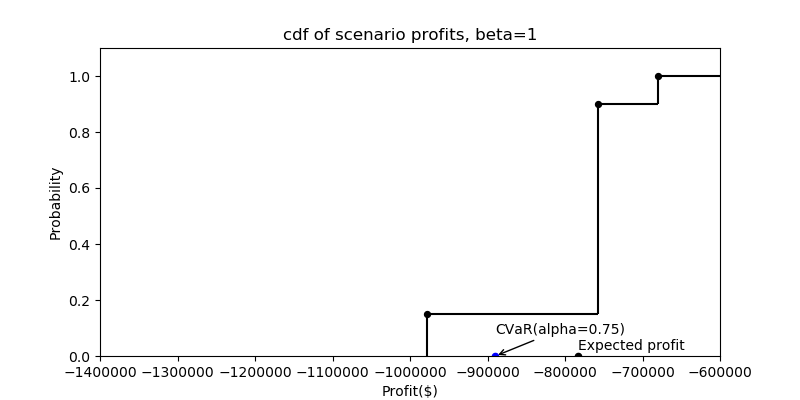
p\_c = 90.0

scenario 1: p1=30.0 prob.=0.15 profit=-977760.0 s1=220079.99999999313

scenario 2: p2=10.0 prob.=0.75 profit=-757680.0 s2=1.1641532182693481e-08

scenario 3: p3=0.0 prob.=0.1 profit=-680400.0 s3=0

CVaR=-889728.0, Exp\_profit=-782964.0



The CVaR, with alpha = 0.75, represents the expected profit being worse than the (1-0.75)\*100%=25%-quantile of the profit distribution.

1. Efficient frontier

The following figure shows the efficient frontier according to the above calculation results.