code_test

December 5, 2024

1

1.0.1 MCA)

[1]:
```
import numpy as np
import math
import copy
import time



#

def calculate_exp_sd(RS, s, d, parameter_p, cost_sr, cost_rs, value,␣
↪expect_minimum_sd_before):
    Exp_expect_cost_sd = {}
    for rs in RS:
        Exp_expect_cost_sd[rs] = math.exp( -parameter_p * ( cost_sr.
↪get((s,rs[0])) + cost_rs.get(rs) - value.get(rs) + expect_minimum_sd_before.
↪get((rs[1],d)) ) )

    return Exp_expect_cost_sd


def calculate_exp_od(RS, o, d, parameter_p, cost_or, cost_rs, value,␣
↪expect_minimum_sd):
    Exp_expect_cost_od = {}
    for rs in RS:
        Exp_expect_cost_od[rs] = math.exp( -parameter_p * ( cost_or.
↪get((o,rs[0])) + cost_rs.get(rs) - value.get(rs) + expect_minimum_sd.
↪get((rs[1],d)) ) )

    return Exp_expect_cost_od


def delta(r_s, rs):
    if rs == r_s:
        return 1
    else:
        return 0


def calculate_sum_probability_sd(P_probability_sd, s, d, rs, RS,␣
 ↪partial_differentiation_before):
    sum_probability_sd = 0
    for r_s in RS:
        sum_probability_sd += P_probability_sd.get((s,d,r_s)) * ( delta(r_s,␣
 ↪rs) - partial_differentiation_before.get((r_s[1],d,rs)) )   # from def delta

    return sum_probability_sd


def calculate_sum_probability_od(P_probability_od, o, d, rs, RS,␣
 ↪partial_differentiation_sd):
    sum_probability_od = 0
    for r_s in RS:
        sum_probability_od += P_probability_od.get((o,d,r_s)) * ( - delta(r_s,␣
 ↪rs) + partial_differentiation_sd.get((r_s[1],d,rs)) )

    return sum_probability_od


def calculate_sum_expect_cost_od_driver(O, D, num_driver_od, expect_minimum_od):
    sum_expect_cost_od_driver = {}
    for o in O:
        for d in D:
            sum_expect_cost_od_driver[(o,d)] = num_driver_od.get((o,d)) *␣
 ↪expect_minimum_od.get((o,d))

    return sum_expect_cost_od_driver


def calculate_sum_expect_cost_rs_shipper(R, S, num_shipper_rs,␣
 ↪fixed_expected_minimum_cost_rs):
    sum_expect_cost_rs_shipper = {}
    for r in R:
        for s in S:
            sum_expect_cost_rs_shipper[(r,s)] = num_shipper_rs.get((r,s)) *␣
 ↪fixed_expected_minimum_cost_rs.get((r,s))

    return sum_expect_cost_rs_shipper
```

```
def calculate_partial_differentiation(O, D, rs, num_driver_od,␣
↪partial_differentiation_od):
    partial_differentiation = {}
    for o in O:
        for d in D:
            partial_differentiation[(o,d,rs)] = num_driver_od.get((o,d)) *␣
↪partial_differentiation_od.get((o,d,rs))

    sum_partial_differentiation = sum( partial_differentiation.values() )
    partial_differentiation.clear()

    return sum_partial_differentiation





#
# def MCA(value, run_limit=run_limit, cost_or=cost_or, cost_rs=cost_rs,␣
↪cost_sd=cost_sd, cost_sr=cost_sr, cost_od=cost_od,
#          parameter_p=parameter_p, parameter_c=parameter_c,
#          O=O, R=R, S=S, D=D, RS=RS,
#          bnum_driver_od=num_driver_od, num_shipper_rs=num_shipper_rs,␣
↪fixed_cost_do_shipper_rs=fixed_cost_do_shipper,␣
↪fixed_cost_dont_shipper_rs=fixed_cost_dont_shipper_rs):

def MCA(value, run_limit, cost_or, cost_rs, cost_sd, cost_sr, cost_od,
        parameter_p, parameter_c,
        O, R, S, D, RS,
        num_driver_od, num_shipper_rs, fixed_cost_do_shipper_rs,␣
↪fixed_cost_dont_shipper_rs):

    '''
    value :     {('r','s'):value}
    run_limit :      [1]

    cost_or : or    {(o,r):cost}
    cost_rs : rs    {(r,s):cost}
    cost_sd : sd    {(s,d):cost}
    cost_sr : sr    {(s,r):cost}
    cost_od : od    {(o,d):cost}

    parameter_p :         [1]
    parameter_c :         [1]

    O :      {'o'}
    R :      {'r'}
    S :      {'s'}
    D :       {'d'}
    RS :   OD   {('r','s')}

    num_driver_od : od   {('o','d'):num}
    num_shipper_rs : rs   {('r','s'):num}
    fixed_cost_do_shipper_rs : rs      {('r','s'):cost_do}
    fixed_cost_dont_shipper_rs : rs       {('r','s'):cost_dont}
    '''


    #
    expect_minimum_sd_before = copy.deepcopy(cost_sd)   #␣
↪expect_minimum_sd_bofore :
    partial_differentiation_before = {(s, d, rs): 0 for s in S for d in D for␣
↪rs in RS}   # partial_differentiation_before :



    #    (s*d  )
    Z_expect_cost_sd = {}   # Z_sd^(i)
    P_probability_sd = {}   # P_sd^(i)
    partial_differentiation_sd = {}   # partial(mu_sd^(i))
    expect_minimum_sd = {}   # mu_sd^(i)

    fixed_expected_minimum_cost_rs = {}   # V_rs(v_rs)
    num_do_shipper_rs = {}   # z_rs^1(v_rs)

    #

    #
    for i in range(1,run_limit-1):

        if i > 1:   # i > 1
            # mu_sd^(i)
            expect_minimum_sd_before.clear()
            expect_minimum_sd_before = copy.deepcopy(expect_minimum_sd)

            # partial(mu_sd^(i))
            partial_differentiation_before.clear()
            partial_differentiation_before = copy.
↪deepcopy(partial_differentiation_sd)

            #
            Z_expect_cost_sd.clear()   # Z_sd^(i)
            P_probability_sd.clear()   # P_sd^(i)
        # s*d
        for s in S:
            for d in D:
                #
                exp_expect_cost_sd = calculate_exp_sd(RS, s, d, parameter_p,␣
↪cost_sr, cost_rs, value, expect_minimum_sd_before)   # from def␣
↪calculate_exp_sd

                #
                Z_expect_cost_sd[(s,d)] = math.exp( -parameter_p * cost_sd.
↪get((s,d)) ) + sum( exp_expect_cost_sd.values() )   # Compute Z_sd^(i)

                for rs in RS:
                    P_probability_sd[(s,d,rs)] = exp_expect_cost_sd.get(rs) /␣
↪Z_expect_cost_sd.get((s,d))   # Compute P_sd^(i)

                for rs in RS:
                    partial_differentiation_sd[(s,d,rs)] = -␣
↪calculate_sum_probability_sd(P_probability_sd, s, d, rs, RS,␣
↪partial_differentiation_before)   # Compute partial(mu_sd^(i))  from def␣
↪calculate_sum_probability_sd

                expect_minimum_sd[(s,d)] = -1 / parameter_p * math.log( math.
↪exp( -parameter_p * cost_sd.get((s,d)) ) + sum( exp_expect_cost_sd.values()␣
↪) )   # Compute mu_sd^(i)




    #
    for rs in RS:
        fixed_expected_minimum_cost_rs[rs] = -1 / parameter_c * math.log( math.
↪exp( -parameter_c * fixed_cost_dont_shipper_rs.get(rs) ) + math.exp (␣
↪-parameter_c * ( fixed_cost_do_shipper_rs.get(rs) + value.get(rs) ) ) )   #␣
↪Compute V_rs(v_rs)

        num_do_shipper_rs[rs] = num_shipper_rs.get(rs) * math.exp( -parameter_c␣
↪* ( fixed_cost_do_shipper_rs[rs] + value[rs])) / ( math.exp( -parameter_c *␣
↪( fixed_cost_do_shipper_rs[rs] + value[rs])) + math.exp( -parameter_c * ␣
↪fixed_cost_dont_shipper_rs[rs]))   # Compute z_rs^1(v_rs)


    #    (o*d   )
    Z_expect_cost_od = {}
    P_probability_od = {}
    partial_differentiation_od = {}
    expect_minimum_od = {}

    # o*d
    for o in O:
        for d in D:
            #
            exp_expect_cost_od = calculate_exp_od(RS, o, d, parameter_p,␣
↪cost_or, cost_rs, value, expect_minimum_sd)   # from calculate_exp_od

            #
            Z_expect_cost_od[(o,d)] = math.exp( -parameter_p * cost_od.
↪get((o,d)) ) + sum( exp_expect_cost_od.values() )   # Compute Z_od^(n)

            for rs in RS:
                P_probability_od[(o,d,rs)] = exp_expect_cost_od.get(rs) /␣
↪Z_expect_cost_od.get((o,d))   # Compute P_od^(n)

            for rs in RS:
                partial_differentiation_od[(o,d,rs)] = -␣
↪calculate_sum_probability_od(P_probability_od, o, d, rs, RS,␣
↪partial_differentiation_sd)   # Compute partial(mu_od^(n)) from def␣
↪calculate_sum_probability_od

            expect_minimum_od[(o,d)] = -1 / parameter_p * math.log( math.exp(␣
↪-parameter_p * cost_od.get((o,d)) ) + sum( exp_expect_cost_od.values() ) )  ␣
↪# Compute mu_od^(n)


    # MC & gradientMC


    # MC
    #     sum
    sum_expect_cost_od_driver = calculate_sum_expect_cost_od_driver(O, D,␣
↪num_driver_od, expect_minimum_od)   # Compute bar(y)_od * mu_od^(n) from def␣
↪calculate_sum_expect_cost_od_driver
    sum_expect_cost_rs_shipper = calculate_sum_expect_cost_rs_shipper(R, S,␣
↪num_shipper_rs, fixed_expected_minimum_cost_rs) # Compute bar(z)_rs *␣
↪V_rs(v_rs) from def calculate_sum_expect_cost_rs_shipper

    # solve MC
    dual_obj = sum( sum_expect_cost_od_driver.values() ) + sum(␣
↪sum_expect_cost_rs_shipper.values() )   # Compute MC from def␣
↪calculate_sum_expect_cost_od_driver & def␣
↪calculate_sum_expect_cost_rs_shipper
```

```
    # gradientMC
    #
    dual_grad = {}

    # solve grdientMC
    for rs in RS:
        dual_grad[rs] = calculate_partial_differentiation(O, D, rs,␣
 ↪num_driver_od, partial_differentiation_od) + num_do_shipper_rs.get(rs)


    return dual_obj, dual_grad
```

1.0.2 FISTA

```
[2]: def MC(beta):
         ans_MC = MCA(beta, run_limit, cost_or, cost_rs, cost_sd, cost_sr, cost_od,
             parameter_p, parameter_c,
             O, R, S, D, RS,
             num_driver_od, num_shipper_rs, fixed_cost_do_shipper_rs,␣
      ↪fixed_cost_dont_shipper_rs)[0]
         return ans_MC

     def calculate_grad_MC(beta):
         ans_nabla_MC = MCA(beta, run_limit, cost_or, cost_rs, cost_sd, cost_sr,␣
      ↪cost_od,
             parameter_p, parameter_c,
             O, R, S, D, RS,
             num_driver_od, num_shipper_rs, fixed_cost_do_shipper_rs,␣
      ↪fixed_cost_dont_shipper_rs)[1]
         return ans_nabla_MC

     def bector_of_MC(beta, step_size, RS, grad_MC):
         Bector_of_MC = {}
         for rs in RS:
             Bector_of_MC[rs] = beta.get(rs) + step_size * grad_MC.get(rs)    # from␣
      ↪def nablaMC
         return Bector_of_MC

     def calc_diff(rs, v_after, v_before):
         difference = np.zeros(len(RS))
         i = 0
         for rs in RS:
             difference[i] = v_after.get(rs) - v_before.get(rs)
             i += 1
         return difference
```

```
def fista(ipsilon, eta, stepsize, RS):
    max_inner_iter = 100000
    max_outer_iter = 10000
    # beta = np.array([5.0, 0.0, -3.0])  #

    beta = {}
    for rs in RS:
        beta[rs] = 1   #   =1

    v_after = copy.deepcopy(beta)
    t = 1
    iota = 0
    min_step_size = 1e-20  #

    for k in range(max_outer_iter):

        #     grad_MC
        grad_MC = calculate_grad_MC(beta)

        iota = 0  #
        while iota < max_inner_iter:
            step_size = 1 / (stepsize * eta**iota)
            if step_size < min_step_size:
                print("Warning: Step size became too small.")
                return beta  #

            F = MC( bector_of_MC(beta, step_size, RS, grad_MC) )   # from def␣
 ↪bector_of_MC
            Q = MC(beta) - (step_size / 2) * np.linalg.norm( list(grad_MC.
 ↪values()) )**2
            if F >= Q:
                break
            iota += 1

        #
        if iota == max_inner_iter:
            print("Warning: Inner loop reached maximum iterations.")
            return beta

        #
        stepsize = stepsize / eta

        # FISTA
        v_before = copy.deepcopy(v_after)
        v_after.clear()
        for rs in RS:
            v_after[rs] = beta.get(rs) + 1 / stepsize * grad_MC.get(rs)

        #    grad_MC
        judg_grad_MC = calculate_grad_MC(v_after)   # from def calculate_grad_MC

        #
        if np.linalg.norm( list(judg_grad_MC.values()) ) < ipsilon:
            print(f"Converged after {k + 1} outer iterations.")
            return v_after

        #
        if np.dot(list(judg_grad_MC.values()), calc_diff(rs, v_after,␣
↪v_before)) < 0:
            t_before = 1
        else:
            t_before = t

        t = (1 + (1 + 4 * t_before**2)**0.5) / 2

        beta.clear()
        for rs in RS:
            beta[rs] = v_after.get(rs) + ( t_before - 1 ) / t * ( v_after.
↪get(rs) - v_before.get(rs) )


    #
    print(f"Warning: Outer loop reached maximum iterations without full␣
↪convergence.")
    return beta
```

1.0.3

[3]:
```
##

O = set("abcd")
R = set("df")
S = set("eghi")
D = set("jkl")
RS = set()
for r in R:
    for s in S:
        RS.add((r,s))

# print("O:", O , "\n" "R:", R, "\n" "S:", S, "\n" "D:", D, "\n" "RS", RS)

#print()
```

```
## value

# value = {}
# v = 1

# for r in R:
#     for s in S:
#         value[(r,s)] = v
#         v += 1

# print("value:", value)


## value_0

# value_0 = {}



## cost

cost_or, cost_rs, cost_sr, cost_sd, cost_od = [{} for _ in range(5)]

c1 = 1
for o in O:
    for r in R:
        cost_or[(o,r)] = round(c1,2)
        c1 += 0.25
# print("cost_or:", cost_or)

c2 = 5
for s in S:
    for r in R:
        cost_rs[(r,s)] = c2
        c2 -= 0.5
# print("cost_rs:", cost_rs)

c3 = 1
for r in R:
    for s in S:
        cost_sr[(s,r)] = c3
# print("cost_sr:", cost_sr)

c4 = 1
for s in S:
    for d in D:
        cost_sd[(s,d)] = round(c4,1)
        c4 += 0.2
# print("cost_sd:", cost_sd)

c5 = 2
for o in O:
    for d in D:
        cost_od[(o,d)] = c5
        c5 += 2
#print("cost_od:", cost_od)

#print()




##

fixed_cost_do_shipper_rs, fixed_cost_dont_shipper_rs = [{} for _ in range(2)]

for rs in RS:
    fixed_cost_do_shipper_rs[rs], fixed_cost_dont_shipper_rs[rs] = 10, 7
#print("fixed_cost_do_shipper_rs:", fixed_cost_do_shipper_rs, "\n"␣
 ↪"fixed_cost_dont_shipper_rs", fixed_cost_dont_shipper_rs)
```

1.0.4

```
[7]: ##

run_limit = 5
parameter_p = 1e-6
parameter_c = 1e-6

num_driver_od, num_shipper_rs = [{} for _ in range(2)]

for o in O:
    for d in D:
        num_driver_od[(o,d)] = 300

for r in R:
    for s in S:
        num_shipper_rs[(r,s)] = 50

ipsilon = 1.0
eta = 1.1
stepsize = 1.0
```

```python
##
start_time = time.time()
result = fista(ipsilon, eta, stepsize, RS)
end_time = time.time()
print("Run time:", end_time - start_time, "seconds")
print("Result:", result)
```

```
---------------------------------------------------------------------------
OverflowError                             Traceback (most recent call last)
Cell In[7], line 23
     21 ##
     22 start_time = time.time()
---> 23 result = fista(ipsilon, eta, stepsize, RS)
     24 end_time = time.time()
     25 print("Run time:", end_time - start_time, "seconds")

Cell In[2], line 48, in fista(ipsilon, eta, stepsize, RS)
     43 min_step_size = 1e-20  #
     45 for k in range(max_outer_iter):
     46 
     47     #    grad_MC
---> 48     grad_MC = calculate_grad_MC(beta)
     50     iota = 0  #
     51     while iota < max_inner_iter:

Cell In[2], line 9, in calculate_grad_MC(beta)
      8 def calculate_grad_MC(beta):
----> 9     ans_nabla_MC = MCA(beta, run_limit, cost_or, cost_rs, cost_sd,␣
 ↪cost_sr, cost_od,
     10         parameter_p, parameter_c,
     11         O, R, S, D, RS,
     12         num_driver_od, num_shipper_rs, fixed_cost_do_shipper_rs,␣
 ↪fixed_cost_dont_shipper_rs)[1]
     13     return ans_nabla_MC

Cell In[1], line 188, in MCA(value, run_limit, cost_or, cost_rs, cost_sd,␣
 ↪cost_sr, cost_od, parameter_p, parameter_c, O, R, S, D, RS, num_driver_od,␣
 ↪num_shipper_rs, fixed_cost_do_shipper_rs, fixed_cost_dont_shipper_rs)
    185 for o in O:
    186     for d in D:
    187         #
--> 188         exp_expect_cost_od = calculate_exp_od(RS, o, d, parameter_p,␣
 ↪cost_or, cost_rs, value, expect_minimum_sd)   # from calculate_exp_od
    190         #
    191         Z_expect_cost_od[(o,d)] = math.exp( -parameter_p * cost_od.
 ↪get((o,d)) ) + sum( exp_expect_cost_od.values() )   # Compute Z_od^(n)
```

```
Cell In[1], line 20, in calculate_exp_od(RS, o, d, parameter_p, cost_or,␣
 ↪cost_rs, value, expect_minimum_sd)
     18 Exp_expect_cost_od = {}
     19 for rs in RS:
---> 20     Exp_expect_cost_od[rs] = math.exp( -parameter_p * ( cost_or.
 ↪get((o,rs[0])) + cost_rs.get(rs) - value.get(rs) + expect_minimum_sd.
 ↪get((rs[1],d)) ) )
     22 return Exp_expect_cost_od

OverflowError: math range error
```