Nirbhay Sharma (B19CSE114)

optimization for machine learning

Que-1

code

```
import numpy as np
import warnings
warnings.filterwarnings("ignore")

def grad_f(x:np.array):
    return np.array([[200*(x[0] - 1)],[1]],dtype=np.float64)

def grad_2f(x:np.array):
    return np.array([[200,0],[0,0]],dtype=np.float64)

def calculate_F(x:np.array, A:np.array, b:np.array, mu:np.array):
    first_term = grad_f(x) + A.T @ mu
    second_term = A @ x - b
    return np.vstack([first_term,second_term])

def jacobian_F(x:np.array, A:np.array, b:np.array, mu:np.array):
    first_term = np.hstack([grad_2f(x), A.T])
    second_term =
np.hstack([A,np.zeros((A.shape[0],A.shape[0]),dtype=np.float64)])
    return np.vstack([first_term, second_term])

def find_x(x:np.array, mu:np.array, A:np.array, b:np.array):
    iteration = 0
    while True:
        final_feature = np.vstack([x,mu])
        iteration += 1
        jac_inv = np.linalg.inv(jacobian_F(x, A, b, mu))
        f_value = calculate_F(x, A, b, mu)
        final_feature = final_feature - jac_inv @ f_value
        x = final_feature[:x.shape[0],:]
        mu = final_feature[x.shape[0]:,:]
        if iteration == 1000:
            break
    return x, mu

x = np.array([[1],[1]],dtype=np.float64)
mu = np.array([[1],[1]],dtype=np.float64)
A = np.array([[1,6],[-4,1]],dtype=np.float64)
b = np.array([[36],[0]],dtype=np.float64)

final_x, final_mu = find_x(x, mu, A, b)
print(final_x)
print(final_mu)

"""
optimal x
[[1.44]
 [5.76]]
optimal mu
[[-3.68]
 [21.08]]
"""
```

Que-2

code

```
import numpy as np
import warnings
warnings.filterwarnings("ignore")

def calculate_f(x:np.array):
    f_val = x[0] ** 2 + x[1] ** 2 + 2 * (x[2] ** 2) + x[3] ** 2 - 5 * x[0]
- 5 * x[1] - 21 * x[2] + 7 * x[3]
    return f_val

def calculate_g1(x:np.array):
    g1_value = x[0] ** 2 + x[1] ** 2 + x[2] ** 2 + x[3] ** 2 + x[0] - x[1]
+ x[2] - x[3] - 8
    return g1_value

def calculate_g2(x:np.array):
    g2_value = x[0] ** 2 + 2 * (x[1] ** 2) + x[2] ** 2 + 2 * (x[3] ** 2) -
x[0] - x[3] - 10
    return g2_value

def grad_f(cal_f:callable, x:np.array):
    """
    x: shape(N+1,1)
    """

    h = 1e-5
    grad_vector = []
    cur_f = cal_f(x)
    for i in range(x.shape[0]):
        x[i] += h
        new_f = cal_f(x)
        x[i] -= h
        grad = (new_f - cur_f) / h
        grad_vector.append(grad)
    return np.array(grad_vector,dtype=np.float64).reshape(-1,1)

grad_1 = lambda x: (2 * x[0] + 1) / calculate_g1(x)
```

```
grad_2 = lambda x: (2 * x[1] - 1) / calculate_g1(x)
grad_3 = lambda x: (2 * x[2] + 1) / calculate_g1(x)
grad_4 = lambda x: (2 * x[3] - 1) / calculate_g1(x)

grad_5 = lambda x: (2 * x[0] - 1) / calculate_g2(x)
grad_6 = lambda x: (4 * x[1]) / calculate_g2(x)
grad_7 = lambda x: (2 * x[2]) / calculate_g2(x)
grad_8 = lambda x: (4 * x[3] - 1) / calculate_g2(x)

def grad_lag(x, sigma):
    x = x.reshape(-1)
    first_term = np.array([[2*x[0]-5],[2*x[1]-5],[4*x[2]-21],
[2*x[3]+7]],dtype=np.float64)
    second_term = np.array([[grad_1(x)],[grad_2(x)],[grad_3(x)],
[grad_4(x)]],dtype=np.float64)
    third_term = np.array([[grad_5(x)],[grad_6(x)],[grad_7(x)],
[grad_8(x)]],dtype=np.float64)

    return first_term + (second_term + third_term) / sigma


def jacobian_lag(x, sigma):
    first_term = np.zeros((x.shape[0],x.shape[0]),dtype=np.float64)
    np.fill_diagonal(first_term,np.array([2,2,4,2],dtype=np.float64))

    second_term =
np.vstack([grad_f(grad_1,x).T,grad_f(grad_2,x).T,grad_f(grad_3,x).T,grad_f(
grad_4,x).T])
    third_term =
np.vstack([grad_f(grad_5,x).T,grad_f(grad_6,x).T,grad_f(grad_7,x).T,grad_f(
grad_8,x).T])

    return first_term + (second_term + third_term) / sigma

def find_x(x, sigma, m, R):
    while (m/sigma > 1e-4):
        jaco_inv = np.linalg.inv(jacobian_lag(x, sigma))
        F = grad_lag(x, sigma)
        x = x - jaco_inv @ F
        sigma *= R
    return x

x = np.array([[1],[1],[1],[1]],dtype=np.float64)
m = 1
sigma = 1
R = 10
print(find_x(x, sigma, m ,R))

"""
optmial x:

[[ 2.4999111 ]
 [ 2.49988142]
 [ 5.24990335]
 [-3.49980328]]
"""
```

Que-3

code

```

import numpy as np
import warnings
warnings.filterwarnings("ignore")

f_grad_one = lambda x: np.exp(x[0]) * (4 * x[0] ** 2 + 2 * x[1] ** 2 + 4 *
x[0] * x[1] + 8 * x[0] + 6 * x[1] + 1)
f_grad_two = lambda x: np.exp(x[0]) * (4 * x[0] + 4 * x[1] + 2)

log_grad_one = lambda x: x[0] / (25 - x[0] ** 2 - x[1] ** 2)
log_grad_two = lambda x: x[1] / (25 - x[0] ** 2 - x[1] ** 2)

def grad_fun(cal_f:callable, x:np.array):
    h = 1e-5
    grad_vector = []
    cur_f = cal_f(x)
    for i in range(x.shape[0]):
        x[i] += h
        new_f = cal_f(x)
        x[i] -= h
        grad = (new_f - cur_f) / h
        grad_vector.append(grad)
    return np.array(grad_vector,dtype=np.float64).reshape(-1,1)

def Lagrange(x, A, b, mu, sigma):
    return np.exp(x[0]) * (4 * x[0] ** 2 + 2 * x[1] ** 2 + 4 * x[0] * x[1]
+ 2 * x[1] + 1) - 1/sigma * (np.log(25 - x[0] ** 2 - x[1] ** 2)) + mu.T @
(A @ x - b)

def grad_Lag(x, A, b, mu, sigma):
    grad_f_first = f_grad_one(x)
    grad_f_second = f_grad_two(x)
    final_grad_f = np.vstack([grad_f_first, grad_f_second])

    grad_log_first = log_grad_one(x)
    grad_log_second = log_grad_two(x)
    final_grad_log = np.vstack([grad_log_first, grad_log_second])
    final_grad_log = final_grad_log * (2/sigma)

    linear_grad = A.T @ mu
    lag_grad_x = final_grad_f + final_grad_log + linear_grad
    lag_grad_mu = A @ x - b
    return np.vstack([lag_grad_x, lag_grad_mu])

def jacobian_lag(x, A, sigma):
    H1 = np.vstack([grad_fun(f_grad_one, x).T, grad_fun(f_grad_two, x).T])
    H2 = np.vstack([grad_fun(log_grad_one,x).T,grad_fun(log_grad_two,x).T])
    return np.vstack([np.hstack([H1 + 1/sigma * H2, A.T]),
np.hstack([A,np.zeros((A.shape[0],A.shape[0]),dtype=np.float64)])])

def find_x(x, A, b, mu, sigma, m, R):
    while (m/sigma > 1e-4):
        feature = np.vstack([x,mu])
        jaco_inv = np.linalg.inv(jacobian_lag(x, A, sigma))
        F = grad_Lag(x, A, b, mu, sigma)
        feature = feature - jaco_inv @ F
        x = feature[:x.shape[0],:]
        mu = feature[x.shape[0]:,:]
        sigma *= R
    return x, mu

x = np.array([[1],[1]],dtype=np.float64)
A = np.array([[1,2]],dtype=np.float64)
mu = np.array([[1]],dtype=np.float64)
b = np.array([[5]],dtype=np.float64)
m = 1
sigma = 1
R = 10
# print(jacobian_lag(x, A ,sigma))
# print(grad_Lag(x, A, b, mu, sigma))
x, mu = find_x(x, A, b, mu, sigma, m, R)
print(x)
print(mu)

"""
optimal x
[[-2.37443567]
 [ 3.68721783]]
 optimal mu
[[0.10197661]]
"""
```