Nirbhay Sharma (B19CSE114)

optimization for machine learning

Que1

code

```
import numpy as np
import random
import sys

def calculate_h(A, x, b):
    return np.dot(A,x) - b

def grad_h(A, mu):
    return np.dot(A.T,mu)

def grad_f(x:np.array):
    return np.log(x) + 1

def calculate_fxu(x:np.array, A, b, mu):
    grad = grad_f(x) + grad_h(A,mu)
    h_value = calculate_h(A,x,b)
    return np.vstack([grad,h_value])

def grad_square_f(x:np.array):
    n = x.shape[0]
    a = np.zeros((n,n))
    np.fill_diagonal(a,1/x)
    return a

def jacobian(x:np.array, A:np.array):
    grad_2f = grad_square_f(x)
    no_zeros = A.shape[0]
    row1 = np.hstack([grad_2f,A.T])
    row2 = np.hstack([A,np.zeros((no_zeros,no_zeros))])
    jac = np.vstack([row1,row2])
    return jac

def break_criteria(x_k, A, mu, condition):
    cond = grad_f(x_k) + np.dot(A.T,mu)
    return np.linalg.norm(cond) < condition

def cal_solution(x_k:np.array, A, b, mu:np.array, condition):
    iteration = 0
    while True:
        if break_criteria(x_k, A, mu, condition):
            break
        iteration += 1
```

```
        jac = jacobian(x_k, A)
        jac_inv = np.linalg.inv(jac)
        f_xu = calculate_fxu(x_k,A,b,mu)
        update_value = np.dot(jac_inv,f_xu)
        x_k_total = x_k.shape[0]
        cur_xu = np.vstack([x_k,mu])
        cur_xu = cur_xu - update_value
        x_k = cur_xu[:x_k_total]
        mu = cur_xu[x_k_total:]
        if iteration == 1000:
            break
    return x_k, iteration


x = np.array([[1/4],[1/3],[1/12],[1/6],[1/6]],dtype=np.float64)
# print(grad_f(x))
# exit(0)
A = np.array([[1,1,1,1,1]],dtype=np.float64)
b = np.array([[1]],dtype=np.float64)
mu = np.array([[1]],dtype=np.float64)
condition=1e-3
x_k, iteration = cal_solution(x, A, b, mu, condition)
print(x_k)

print(iteration)

"""
[[0.20000408]
 [0.20000249]
 [0.19998697]
 [0.20000323]
 [0.20000323]]
3
"""
```

Que2

code

```
import numpy as np
import random
import sys

def calculate_h(A, x, b):
    return np.dot(A,x) - b

def grad_h(A, mu):
    return np.dot(A.T,mu)

def grad_f(x:np.array):
    return (1-x) * np.exp(-x)
```

```
def calculate_fxu(x:np.array, A, b, mu):
    grad = grad_f(x) + grad_h(A,mu)
    h_value = calculate_h(A,x,b)
    return np.vstack([grad,h_value])

def grad_square_f(x:np.array):
    n = x.shape[0]
    a = np.zeros((n,n))
    np.fill_diagonal(a,(x-2) * np.exp(-x))
    return a

def jacobian(x:np.array, A:np.array):
    grad_2f = grad_square_f(x)
    no_zeros = A.shape[0]
    row1 = np.hstack([grad_2f,A.T])
    row2 = np.hstack([A,np.zeros((no_zeros,no_zeros))])
    jac = np.vstack([row1,row2])
    return jac

def break_criteria(x_k, A, mu, condition):
    cond = grad_f(x_k) + np.dot(A.T,mu)
    return np.linalg.norm(cond) < condition

def cal_solution(x_k:np.array, A, b, mu:np.array, condition):
    iteration = 0
    while True:
        if break_criteria(x_k, A, mu, condition):
            break
        iteration += 1
        jac = jacobian(x_k, A)
        jac_inv = np.linalg.inv(jac)
        f_xu = calculate_fxu(x_k,A,b,mu)
        update_value = np.dot(jac_inv,f_xu)
        x_k_total = x_k.shape[0]
        cur_xu = np.vstack([x_k,mu])
        cur_xu = cur_xu - update_value
        x_k = cur_xu[:x_k_total]
        mu = cur_xu[x_k_total:]
        if iteration == 1000:
            break
    return x_k, mu, iteration


x = np.array([[2/3],[1/3],[0],[0]],dtype=np.float64)
# print(grad_f(x))
# exit(0)
A = np.array([[1,1,1,1],[1,-2,3,-4]],dtype=np.float64)
b = np.array([[1],[1]],dtype=np.float64)
mu = np.array([[1],[1]],dtype=np.float64)
condition=1e-3
x_k, mu, iteration = cal_solution(x, A, b, mu, condition)
print(x_k)
print(mu)
```

```
print(iteration)

print(calculate_fxu(x_k, A, b, mu))

"""
[[0.31552986]
 [0.16339847]
 [0.44222197]
 [0.0788497 ]]
[[-0.56966434]
 [ 0.07041081]]
3
[[5.67164166e-07]
 [1.63452417e-07]
 [1.57744602e-07]
 [2.82930201e-09]
 [0.00000000e+00]
 [0.00000000e+00]]
"""
```