Amir Hossein Eyvazkhani

1747696

Ex4

1

a) we can solve that using normal equations:

$$A = X^T X = \begin{pmatrix} 1.5 & 3 & 4.5 \\ 2 & 2.5 & 3 \end{pmatrix} \begin{pmatrix} 1.5 & 2 \\ 3 & 2.5 \\ 4.5 & 3 \end{pmatrix} = \begin{pmatrix} 31.5 & 24 \\ 24 & 19.25 \end{pmatrix}$$

$$B = X^T y = \begin{pmatrix} 1.5 & 3 & 4.5 \\ 2 & 2.5 & 3 \end{pmatrix} \begin{pmatrix} 10 \\ 15.5 \\ 21 \end{pmatrix} = \begin{pmatrix} 156 \\ 121.75 \end{pmatrix}$$

Solve $Ax = B \longrightarrow x = A^{-1} B = \begin{pmatrix} 2.6 \\ 3 \end{pmatrix} = \beta$

b) first of all, in the part a), the analytical solution seems easy to find. But it is not the case for all ML problems. Some cases the analetical solution migth be very complex to find. The second reason can be that in such cases, these kinds of solutions migth be computationaly infeasible for current available compute powers. But learning algorithms can be implemented without this issue. Moreover, they can also be designed to be flexible.

c) $L = \| X\beta - y \|_2^2 \longrightarrow \nabla L(\beta) = 2(X^T X \beta - X^T y)$

$\mu = 0.1$, $\beta_0 = (1,1)^T \longrightarrow$ error $= L(\beta) = 108.16$

$\nabla L_1 = \begin{pmatrix} -201 \\ -157 \end{pmatrix}$

$\beta_1 = \beta - \mu \nabla L_1 = \begin{pmatrix} 21.1 \\ 16.7 \end{pmatrix} \longrightarrow$ error $= L(\beta_1) = 8812.7$

$\nabla L_2 = \begin{pmatrix} 1818.9 \\ 1412.25 \end{pmatrix}$

$\beta_2 = \beta_1 - \mu \nabla L_2 = \begin{pmatrix} -160.79 \\ -124.52 \end{pmatrix} \longrightarrow$ error $= 718408.69 = L(\beta_2)$

H:\Uni\WiSe 2024\MOT\Exercises\4\t.py

```python
# the code used for gredient descent

import numpy as np
b = np.array([[1], [1]], dtype='float')
x = np.array([[1.5,2], [3,2.5], [4.5,3]])
y = np.array([[10],[15.5],[21]])
moo = 0.1
def grad(x,y,b):
    return 2*(x.T@x@b-x.T@y)

def error(x, y , b):
    return np.mean((x@b - y) ** 2)

print(f'b : {b} | error : {error(x, y , b)}')
b1 = b - moo * grad(x,y,b)

print(f'b1 : {b1} | error : {error(x, y , b1)}')
b2 = b1 - moo * grad(x,y,b1)
print(f'b2 : {b2} | error : {error(x, y , b2)}')
```

1

a) In stochastic gradient descent (SGD), we don't go through the whole dataset in one update step. In other words, the loss function would not have a sum over N, but rather over a mini-batch of 16, 32, 64, 128 data points. That's because with big datasets and complex models, computing gradients in that way would be computationally expensive and even not feasible.

b) The code and intermediate results for this part are given in the next page.

c) The code and intermediate results are given in the next pages.

→ Also Adagrad helps in this algorithm. We can see that by comparing losses.

The code and the results (final and intermediate) for question 2, part b)

```python
# the code used for stochastic gredient descent

import numpy as np
b = np.array([[1], [1]], dtype='float')
x = np.array([[1.5,2], [3,2.5], [4.5,3]])
y = np.array([[10],[15.5],[21]])
moo = 0.1
epochs = 2

def grad(x, y,b):
    return 2*(x.T@x@b-x.T@y)

def mse(x,y,b):
    return (x@b - y)**2

for epoch in range(epochs):
    for i in range(len(x)):
        x_i = x[i, :].reshape((1,2))
        y_i = y[i, :].reshape((1,1))
        gradient = -grad(x_i, y_i,b)
        err = mse(x_i, y_i,b)
        print(f'epoch {epoch} - step {i}\n===============================')
        print(f'b: {b}')
        print(f'mse: {err}')
        b = b - moo * gradient

print(f'final b is : {b}')
```

```
epoch 0 - step 0
================================
b: [[1.]
 [1.]]
mse: [[42.25]]
epoch 0 - step 1
================================
b: [[-0.95]
 [-1.6 ]]
mse: [[499.5225]]
epoch 0 - step 2
================================
b: [[-14.36 ]
 [-12.775]]
mse: [[15362.363025]]
epoch 1 - step 0
================================
b: [[-125.9105]
 [ -87.142 ]]
mse: [[139240.73592506]]
epoch 1 - step 1
================================
b: [[-237.855425]
 [-236.4019  ]]
mse: [[1742587.51104455]]
epoch 1 - step 2
================================
b: [[-1029.89804  ]
 [ -896.4374125]]
mse: [[53946871.72456145]]
final b is : [[-7640.26611575]
 [-5303.349463  ]]
```

The code and the results (final and intermediate) for question 2, part c)

```python
# the code used for stochastic gredient descent with adagrad
import numpy as np
b = np.array([[1], [1]], dtype='float')
x = np.array([[1.5,2], [3,2.5], [4.5,3]])
y = np.array([[10],[15.5],[21]])
moo = 0.1
epochs = 2
epsilon = 1e-8
def grad(x, y,b):
    return 2*(x.T@x@b-x.T@y)

def mse(x,y,b):
    return (x@b - y)**2
gradian_history = np.array([[0],[0]], dtype='float')
for epoch in range(epochs):
    for i in range(len(x)):
        x_i = x[i, :].reshape((1,2))
        y_i = y[i, :].reshape((1,1))
        gradient = -grad(x_i, y_i,b)
        err = mse(x_i, y_i,b)
        print(f'epoch {epoch} - step {i}\n==============================')
        print(f'b: {b}')
        print(f'mse: {err}')
        gradian_history += gradient**2
        step_size = moo / (np.sqrt(gradian_history) + epsilon)
        b = b - step_size * gradient
print(f'final b is : {b}')
```

```
epoch 0 - step 0
==================================
b: [[1.]
 [1.]]
mse: [[42.25]]
epoch 0 - step 1
==================================
b: [[0.9]
 [0.9]]
mse: [[111.30249999]]
epoch 0 - step 2
==================================
b: [[0.8044319 ]
 [0.81030368]]
mse: [[223.47694926]]
epoch 1 - step 0
==================================
b: [[0.71471464]
 [0.72667635]]
mse: [[55.86927667]]
epoch 1 - step 1
==================================
b: [[0.69992617]
 [0.69982432]]
mse: [[135.7378943]]
epoch 1 - step 2
==================================
b: [[0.65805937]
 [0.65346741]]
mse: [[258.51271487]]
final b is : [[0.59256638]
 [0.59257312]]
```