IE406
Machine Learning

Lab 2

Group 28

201801015 - Shantanu Tyagi
201801076 - Shivani Nandani
201801407 - Pratvi Shah
201801408 - Arkaprabha Banerjee

Contents

1 Q1 **4**
- 1.1 Code . . . 4
- 1.2 Observations and Results . . . 5

2 Q2 **6**
- 2.1 Part 1 . . . 6
  - 2.1.1 Code . . . 6
  - 2.1.2 Plots . . . 7
  - 2.1.3 Observations and results . . . 7
- 2.2 Part 2 . . . 8
  - 2.2.1 Code . . . 8
  - 2.2.2 Plots . . . 9
  - 2.2.3 Observations and results . . . 10
- 2.3 Part 3 . . . 10
  - 2.3.1 Code . . . 10
  - 2.3.2 Plots . . . 11
  - 2.3.3 Observations and results . . . 11
- 2.4 Part 4 . . . 12
  - 2.4.1 Code . . . 12
  - 2.4.2 Plots . . . 13
  - 2.4.3 Observations and results . . . 14
- 2.5 Part 5 . . . 14
  - 2.5.1 Code . . . 14
  - 2.5.2 Observations and results . . . 15

3 Q3 **15**
- 3.1 Code . . . 16
- 3.2 Observations and results . . . 17

4 Q4 **17**
- 4.1 Part A . . . 18
  - 4.1.1 Code . . . 18
  - 4.1.2 Plot . . . 18
- 4.2 Part B . . . 19
  - 4.2.1 Code . . . 19
  - 4.2.2 Plot . . . 20
- 4.3 Part C . . . 20
  - 4.3.1 Code . . . 20
  - 4.3.2 Plot . . . 21
- 4.4 Part D . . . 21
  - 4.4.1 Code . . . 21
  - 4.4.2 Plot . . . 22
- 4.5 Part E . . . 22
  - 4.5.1 Code . . . 22

4.5.2 Plot 23

5 Q5 **24**

5.1 Part 1 24

5.2 Part 2 25

1 Q1

For this question we have higher degree polynomial with only one independent variable i.e., x, and the polynomial can still be explained as the linear combination of various powers of x:

$$y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + ... + \theta_p x_p$$

$$y = \theta_0 + \theta_1 x + \theta_2 x^2 + ... + \theta_p x^p$$

$$y = \theta^T \mathbf{X}$$

now,

- $\mathbf{X} = [\ \mathbf{1}\ \mathbf{x}\ \mathbf{x}^2 \cdots \mathbf{x}^p]^T$
- p = degree of the polynomial y
- $\theta = [\theta_0\ \theta_1\ \theta_2 \cdots \theta_p]$ = regression coefficients vector

We solved this question using two approaches:

1. Normal Equations Method (NE): Here we directly solve the equation by using pseudo inverse method i.e., $\theta = (X^T X)^{-1} X^T y$.
2. Using special form of Linear Regression (LR): Here, instead of fitting (x,y) to the linear model we transform the x to $\mathbf{X}$ matrix and then fit it to the Linear Regression model of sklearn.

1.1 Code

```
def normalEquationMethod(x,y):
    xtx_inverse = np.linalg.pinv(np.matmul(x.T,x))                    # (X'X)^-1
    theta = np.matmul(np.matmul(xtx_inverse,x.T),np.transpose(y))     # ((X'X)^-1)X'y
    return theta

def polynomialRegression(X,y):
    lin_reg = LinearRegression()
    lin_reg.fit(X,y)
    theta = lin_reg.coef_
    theta[0] = lin_reg.intercept_
    return theta

def driverFunction(p,x,y,df_ans):
    X = [x**(i) for i in range(p+1)]
    X = np.array(X).reshape(p+1, y.shape[0]).transpose()
    ans_NE = normalEquationMethod(X,y)
    df_ans.append(ans_NE)
    y_hat_NE.append(np.matmul(X,np.transpose(ans_NE)))
    print(f'RMSE for p={p} Normal Equation: {np.sqrt(mse(y,y_hat_NE[-1]))}')
    ans_LR = polynomialRegression(X,y)
    df_ans.append(ans_LR)
```

```
    y_hat_LR.append(np.matmul(X,np.transpose(ans_LR)))
    print(f'RMSE for p={p} Linear Regression: {np.sqrt(mse(y,y_hat_LR[-1]))}')

x = np.arange(0, 20.1, 0.1)
np.random.seed(0)
theta0 = np.random.randn()
real_theta = [(-1 + theta0)*1e5, -300, 8, -100, 3, 1]
y = 1*x**5 + 3*x**4 - 100*x**3 + 8*x**2 -300*x - 1e5 + theta0*1e5

df_ans = []; y_hat_NE = []; y_hat_LR= []
df_ans.append([i for i in range(P[-1]+1)])
df_ans.append(real_theta)

driverFunction(4,x,y,df_ans)
driverFunction(5,x,y,df_ans)

ans = pd.DataFrame(df_ans).T
ans.columns= ['Degree','Real','NE p=4','LR p=4', 'NE p=5' ,'LR p=5']
print(ans)
```

1.2 Observations and Results

The table below represents the regression coefficients for for NE and LR method using p=4 and p=5.

Degree	Real	NE p=4	LR p=4	NE p=5	LR p=5
0.0	76405.234597	88479.758466	88479.758597	76405.238110	76405.234597
1.0	-300.000000	-19066.341207	-19066.341289	-300.003277	-300.000000
2.0	8.000000	6641.833319	6641.833333	8.000900	8.000000
3.0	-100.000000	-987.794443	-987.794444	-100.000103	-100.000000
4.0	3.000000	53.000000	53.000000	3.000005	3.000000
5.0	1.000000	NaN	NaN	1.000000	1.000000

- For Normal Equation Method RMSE between the predicted and real values of y
  p=4 : 3922.72374
  p=5 : 0.0006165
- For Linear Regression RMSE between the predicted and real values of y
  p=4 : 3922.72374
  p=5 : 3.778412e-08
- Due to the ill-conditioned nature of Vandermonde matrix, we observe that even with very small difference in coefficients obtained using Normal Equation and the real coefficient the error is comparable. The solution obtained using Linear Regression gives us very accurate answer and hence the error is almost negligible.

2 Q2

The standard mathematical format for gradient descent algorithm is as following:

$$\theta_j = \theta_j - \alpha \cdot \frac{\partial L(\theta)}{\partial \theta_j} \tag{1}$$

, where:

1. θ_j : j^{th} parameter
2. α : Learning Rate
3. $L(\theta)$: Loss Function

Note: The red dots in the figure represent the current position of the gradient decent algorithm , and the lowest point represents the final minima value attained by the algorithm.

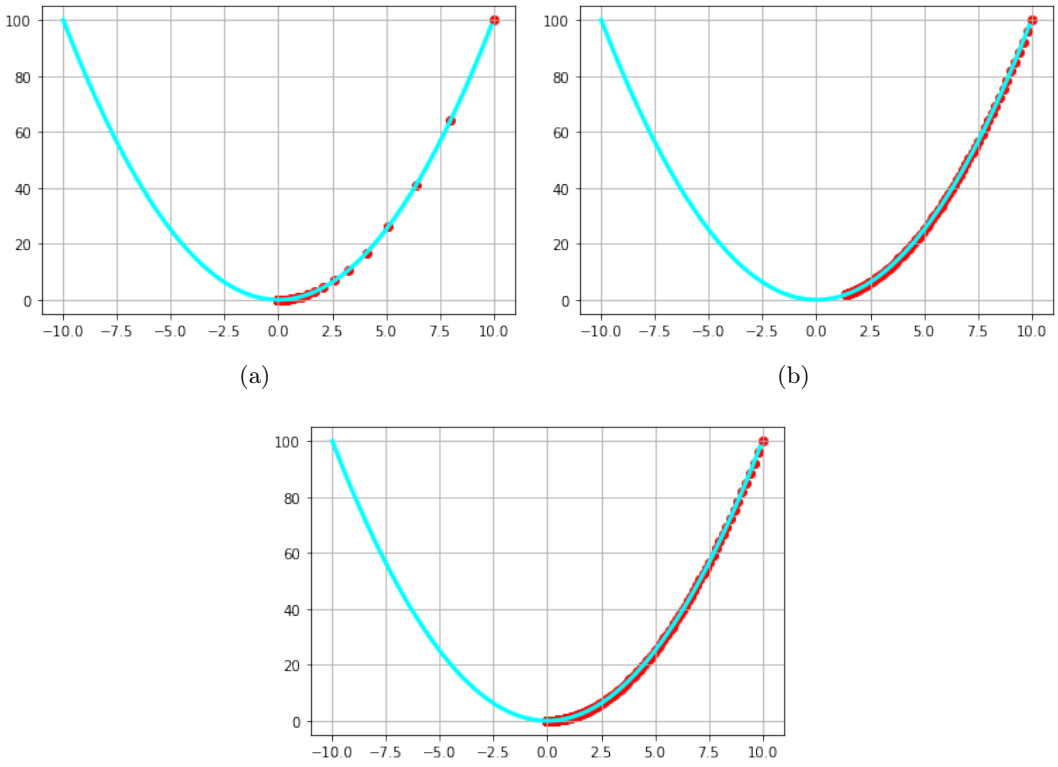
2.1 Part 1

2.1.1 Code

```
def L(theta):
    return theta*theta

def gradientDescent(iterations,alpha,startingVal):

    gradientPlot = [startingVal]
    for i in range(iterations - 1):
        gradientPlot.append(gradientPlot[-1] - alpha*2*gradientPlot[-1]);
    gradientPlot = np.asarray(gradientPlot)
    return gradientPlot

def driverFunc(alpha,iterations):
    theta = np.arange(-10,10,0.1)
    L_theta = []
    for t in theta:
        L_theta.append(L(t))

    iterPlot = gradientDescent(iterations,alpha,10)
    print('Minimum value of L() is', round(L(iterPlot[-1])), ' at  = ',
    ↪  round(iterPlot[-1]))
    plt.figure()
    plt.plot(theta, L_theta, 'cyan', linewidth=3)
    plt.scatter(iterPlot,L(iterPlot),c='red')
    plt.grid()
    plt.show()

driverFunc(0.1,100)
```

```
driverFunc(0.01,100)
driverFunc(0.01,1000)
```

2.1.2 Plots



(a) (b)

(c)

Figure 1:
a) $\alpha = 0.1$, iterations = 100
b) $\alpha = 0.01$, iterations = 100
c) α =0.01 , iterations = 1000

2.1.3 Observations and results

- for $\alpha = 0.1$ and 100 iterations, we obtain minimum value of $L(\theta)$ as 0 at $\theta = 0$
- for $\alpha = 0.01$ and 100 iterations, we obtain minimum value of $L(\theta)$ as 2 at $\theta = 1$
- for $\alpha = 0.01$ and 1000 iterations, we obtain minimum value of $L(\theta)$ as 0 at $\theta = 0$
- We observe from the graph that on increasing the learning rate we take bigger steps towards the minima and hence are able to converge quickly.

- For lower learning rates if we take insufficient number of iterations then even if we do tend to move towards the minima however we may not reach the actual minima. In such cases we need to increase the number of iterations.

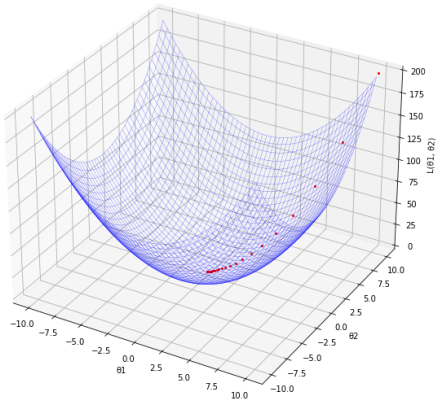
2.2 Part 2

2.2.1 Code

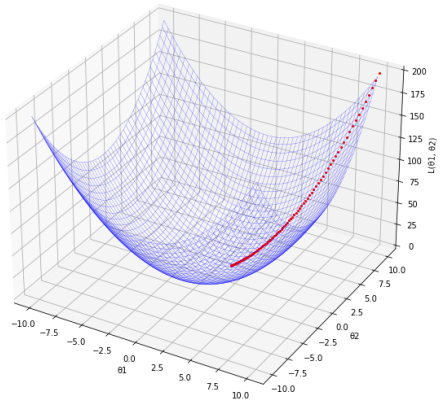
```
def L(theta1, theta2):
    return theta1*theta1 + theta2*theta2

def gradientDescent(iterations,alpha,startingVal):

    gradientPlot1 = [startingVal]
    gradientPlot2 = [startingVal]

    for i in range(iterations - 1):
        gradientPlot1.append(gradientPlot1[-1] - alpha*2*gradientPlot1[-1])
        gradientPlot2.append(gradientPlot2[-1] - alpha*2*gradientPlot2[-1])

    return np.asarray(gradientPlot1) , np.asarray(gradientPlot2)

def driverFunc(alpha,iterations):
    theta1 = np.arange(-10,10,0.1)
    theta2 = np.arange(-10,10,0.1)

    # ans1,ans2  = gradientDescent(1000,0.01,10)
    ans1,ans2  = gradientDescent(iterations,alpha,10)

    L_theta1_theta2 = np.zeros((len(theta1),len(theta2)))
    for i in range(len(theta1)):
        for j in range(len(theta2)):
            L_theta1_theta2[i][j] = L(theta1[i], theta2[j])

    print('Minimum value of L(1, 2) is', round(L(ans1[-1],ans2[-1])),' at 1 = ',
    ↪  round(ans1[-1]),' at 2 = ', round(ans2[-1]))

    x, y = np.meshgrid(theta1, theta2)
    z = L(x, y)

    fig = plt.figure(figsize=(10, 10))
    ax = plt.axes(projection='3d')
    ax.plot_wireframe(x, y, z, color='blue', linewidth=0.2)

    for i in range(len(ans1)):
        ax.plot(ans1[i], ans2[i], L(ans1[i],ans2[i]), 'ro', markersize=2)
```

```
    ax.set_xlabel('1')
    ax.set_ylabel('2')
    ax.set_zlabel('L(1, 2)')
    plt.show()

driverFunc(0.1,100)
driverFunc(0.01,100)
driverFunc(0.01,1000)
```

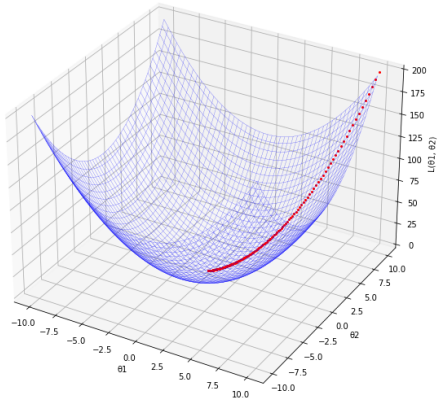
2.2.2 Plots



(a)



(b)



(c)

Figure 2:
a) $\alpha = 0.1$, iterations $= 100$
b) $\alpha = 0.01$, iterations $= 100$
c) $\alpha = 0.01$, iterations $= 1000$

2.2.3 Observations and results

- for $\alpha = 0.1$ and 100 iterations, we obtain minimum value of $L(\theta 1, \theta 2)$ as 0 at θ 1,θ 2 = (0 , 0)
- for $\alpha = 0.01$ and 100 iterations, we obtain minimum value of $L(\theta 1, \theta 2)$ as 4 at θ 1,θ 2 = (1 , 1)
- for $\alpha = 0.01$ and 1000 iterations, we obtain minimum value of $L(\theta 1, \theta 2)$ as 0 at θ 1,θ 2 = (0 , 0)
- We observe from the graph that on increasing the learning rate we take bigger steps towards the minima and hence are able to converge quickly.
- For lower learning rates if we take insufficient number of iterations then even if we do tend to move towards the minima however we may not reach the actual minima. In such cases we need to increase the number of iterations.

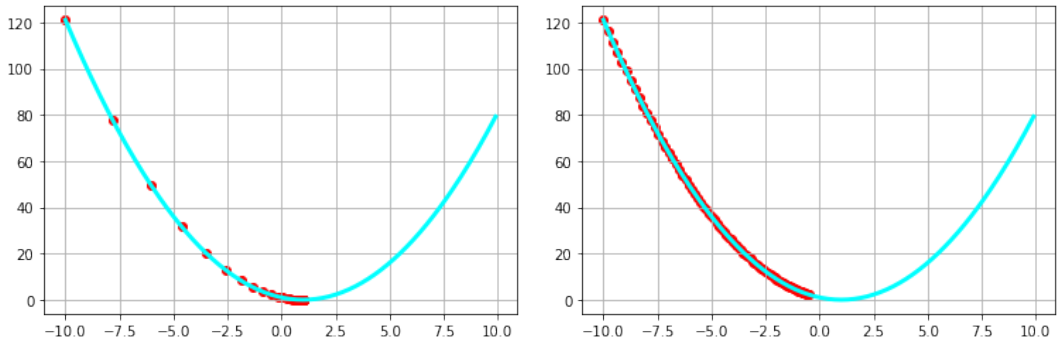
2.3 Part 3

2.3.1 Code

```
def L(theta):
    return (theta-1)**2


def gradientDescent(iterations,alpha,startingVal):

    gradientPlot = [startingVal]
    for i in range(iterations - 1):
        gradientPlot.append(gradientPlot[-1] - alpha*2*(gradientPlot[-1]-1))
    gradientPlot = np.asarray(gradientPlot)
    return gradientPlot

def driverFunc(alpha,iterations):
    theta = np.arange(-10,10,0.1)
    L_theta = []
    for t in theta:
        L_theta.append(L(t))

    iterPlot = gradientDescent(iterations,alpha,-10)
    print('Minimum value of L() is', round(L(iterPlot[-1])), ' at  = ',
    ↪  round(iterPlot[-1]))
    plt.figure()
    plt.plot(theta, L_theta, 'cyan', linewidth=3)
    plt.scatter(iterPlot,L(iterPlot),c='red')
    plt.grid()
    plt.show()
```

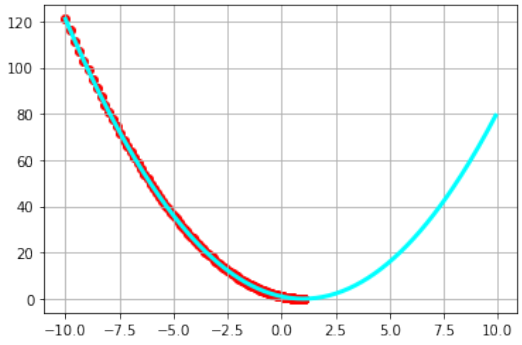
```
driverFunc(0.1,100)
driverFunc(0.01,100)
driverFunc(0.01,1000)
```

2.3.2 Plots



(a)

(b)



(c)

Figure 3:
a) $\alpha = 0.1$, iterations = 100
b) $\alpha = 0.01$, iterations = 100
c) α =0.01 , iterations = 1000

2.3.3 Observations and results

- for $\alpha = 0.1$ and 100 iterations, we obtain minimum value of $L(\theta)$ as 0 at $\theta = 1$
- for $\alpha = 0.01$ and 100 iterations, we obtain minimum value of $L(\theta)$ as 2 at $\theta = 0$
- for $\alpha = 0.01$ and 1000 iterations, we obtain minimum value of $L(\theta)$ as 0 at $\theta = 1$
- We observe from the graph that on increasing the learning rate we take bigger steps towards the minima and hence are able to converge quickly.

- For lower learning rates if we take insufficient number of iterations then even if we do tend to move towards the minima however we may not reach the actual minima. In such cases we need to increase the number of iterations.

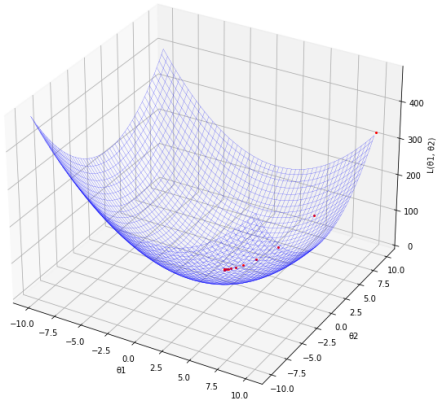
2.4 Part 4

2.4.1 Code

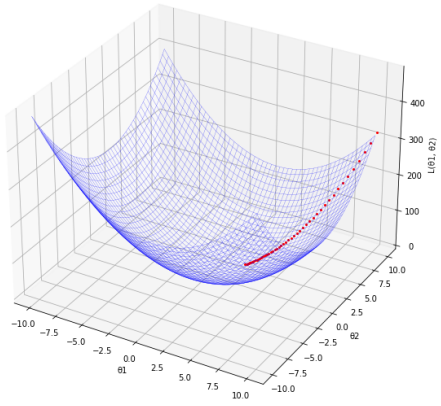
```
def L(theta1, theta2):
    return 2*((theta1-1)**2 + (theta2-1)**2)

def gradientDescent(iterations,alpha,startingVal):

    gradientPlot1 = [startingVal]
    gradientPlot2 = [startingVal]

    for i in range(iterations - 1):
        gradientPlot1.append(gradientPlot1[-1] - alpha*4*(gradientPlot1[-1]-1))
        gradientPlot2.append(gradientPlot2[-1] - alpha*4*(gradientPlot2[-1]-1))

    return np.asarray(gradientPlot1) , np.asarray(gradientPlot2)

def driverFunc(alpha,iterations):

    theta1 = np.arange(-10,10,0.1)
    theta2 = np.arange(-10,10,0.1)

    ans1,ans2  = gradientDescent(iterations,alpha,10)

    L_theta1_theta2 = np.zeros((len(theta1),len(theta2)))
    for i in range(len(theta1)):
        for j in range(len(theta2)):
            L_theta1_theta2[i][j] = L(theta1[i], theta2[j])

    print('Minimum value of L(1, 2) is', round(L(ans1[-1],ans2[-1])),' at 1 = ',
    ↪  round(ans1[-1]),' at 2 = ', round(ans2[-1]))

    x, y = np.meshgrid(theta1, theta2)
    z = L(x, y)

    fig = plt.figure(figsize=(10, 10))
    ax = plt.axes(projection='3d')
    ax.plot_wireframe(x, y, z, color='blue', linewidth=0.2)

    for i in range(len(ans1)):
        ax.plot(ans1[i], ans2[i], L(ans1[i],ans2[i]), 'ro', markersize=2)
```

```
    ax.set_xlabel('1')
    ax.set_ylabel('2')
    ax.set_zlabel('L(1, 2)')
    plt.show()

driverFunc(0.1,50)
driverFunc(0.01,50)
driverFunc(0.01,100)
```

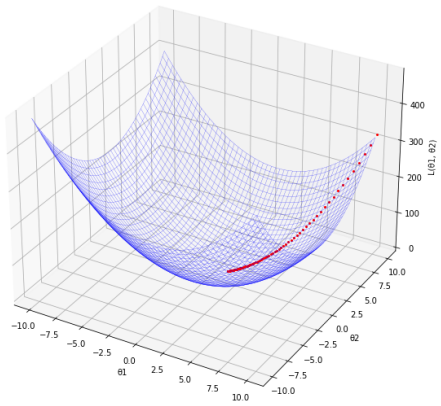
2.4.2 Plots



(a)



(b)



(c)

Figure 4:
a) $\alpha = 0.1$, iterations = 100
b) $\alpha = 0.01$, iterations = 100
c) α =0.01 , iterations = 1000

2.4.3 Observations and results

- for $\alpha = 0.1$ and 50 iterations, we obtain minimum value of $L(\theta 1, \theta 2)$ as 0 at θ 1,θ 2 = (1 , 1)
- for $\alpha = 0.01$ and 50 iterations, we obtain minimum value of $L(\theta 1, \theta 2)$ as 6 at θ 1,θ 2 = (2 , 2)
- for $\alpha = 0.01$ and 100 iterations, we obtain minimum value of $L(\theta 1, \theta 2)$ as 0 at θ 1,θ 2 = (1 , 1)
- We observe from the graph that on increasing the learning rate we take bigger steps towards the minima and hence are able to converge quickly.
- For lower learning rates if we take insufficient number of iterations then even if we do tend to move towards the minima however we may not reach the actual minima. In such cases we need to increase the number of iterations.

2.5 Part 5

2.5.1 Code

```
def L(x, y, theta0, theta1):
    return np.sum(np.square(y - (theta0 + theta1*x)))

def derivativeFunc1(x, y, theta_0, theta_1):
    return np.sum(y - (theta_0 + (theta_1*x)))

def derivativeFunc2(x, y, theta_0, theta_1):
    ans = 0
    for i in range(len(x)):
        ans = ans + x[i]*(y[i] - (theta_0 + theta_1 * x[i] ))
    return ans


def gradientDescent(iterations,alpha,startingVal1,startingVal2,x,y):

    gradientPlot1 = [startingVal1]
    gradientPlot2 = [startingVal2]

    for i in range(iterations - 1):
        gradientPlot1.append(gradientPlot1[i] +
        ↪  alpha*2*derivativeFunc1(x,y,gradientPlot1[i],gradientPlot2[i]))
        gradientPlot2.append(gradientPlot2[i] +
        ↪  alpha*2*derivativeFunc2(x,y,gradientPlot1[i],gradientPlot2[i]))

    return np.asarray(gradientPlot1) , np.asarray(gradientPlot2)

def driverFunc(alpha,iterations):
```

```
    theta0 = np.arange(-50,50,0.5)
    theta1 = np.arange(-1,1,0.01)
    L_theta0_theta1 = np.zeros((len(theta1),len(theta0)))
    data = pd.read_excel('data.xlsx')
    X = data.x
    Y= data.y

    ans1,ans2  = gradientDescent(iterations,alpha,50,0,X, Y)
    print('Minimum value of L(1, 2) is', L(data.x,data.y,ans1[-1],ans2[-1]),' at 1
    ↪  = ', ans1[-1],' at 2 = ', ans2[-1])

driverFunc(1e-11,1000)
driverFunc(1e-12,1000)
driverFunc(1e-12,5000)
```

2.5.2 Observations and results

- for $\alpha = 0.1$ or $\alpha = 0.01$ the value of $L(\theta 1, \theta 2)$ approaches infinity (diverges) withing a couple of iterations. Hence we need to take a smaller value of learning rate.
- for $\alpha =$ 1e-11 and 1000 iterations, we obtain minimum value of $L(\theta 1, \theta 2)$ as 1576.3391560950045 at θ 1,θ 2 = (49.99999711783 , -0.008851835542843913)
- for $\alpha =$ 1e-12 and 1000 iterations, we obtain minimum value of $L(\theta 1, \theta 2)$ as 3588.012370125311 at θ 1,θ 2 = (49.999997679230 , -0.007344317765876041)
- for $\alpha =$ 1e-12 and 1000 iterations, we obtain minimum value of $L(\theta 1, \theta 2)$ as 1576.3405601331215 at θ 1,θ 2 = (49.99999716653 , -0.0088505764975341)
- We observe from the graph that on increasing the learning rate we take bigger steps towards the minima and hence are able to converge quickly.
- For lower learning rates if we take insufficient number of iterations then even if we do tend to move towards the minima however we may not reach the actual minima. In such cases we need to increase the number of iterations.
- For very high learning rates we are not able to converge to the minima.

3 Q3

By definition the primary difference between stochastic gradient descent algorithm and the normal gradient descent algorithm, is that in the form only a single training sample is considered to calculate the gradient whereas for the latter the entire dataset is considered. For the purpose of this questions, we shall consider the sample size to be both 1 as well as 5 to explain our results.

3.1 Code

```
def L(x, y, theta0, theta1):
    return np.sum(np.square(y - (theta0 + theta1*x)))

def derivativeFunc1(x, y, theta_0, theta_1):
    ans = 0
    for i in range(len(x)):
        ans = ans + (y[i] - (theta_0 + theta_1 * x[i] ) )
    return ans

def derivativeFunc2(x, y, theta_0, theta_1):
    ans = 0
    for i in range(len(x)):
        ans = ans + x[i]*(y[i] - (theta_0 + theta_1 * x[i] ) )
    return ans


def gradientDescent(iterations,alpha,startingVal1,startingVal2,x,y,k):

    gradientPlot1 = [startingVal1]
    gradientPlot2 = [startingVal2]
    # k = 5
    for i in range(iterations - 1):
        tempIndex = np.random.randint(0,len(x),k)
        tempX = []
        tempY = []
        for j in tempIndex:
            tempX.append(x[j])
            tempY.append(y[j])
        gradientPlot1.append(gradientPlot1[i] +
        ↪  alpha*2*derivativeFunc1(tempX,tempY,gradientPlot1[i],gradientPlot2[i]))
        gradientPlot2.append(gradientPlot2[i] +
        ↪  alpha*2*derivativeFunc2(tempX,tempY,gradientPlot1[i],gradientPlot2[i]))

    return np.asarray(gradientPlot1) , np.asarray(gradientPlot2)


def driverFunc(alpha,iterations,k):

    theta0 = np.arange(-50,50,0.5)
    theta1 = np.arange(-1,1,0.01)
    data = pd.read_excel('data.xlsx')
    ans1,ans2  = gradientDescent(iterations,alpha,50,0,data.x, data.y,k)
    print('Minimum value of L(1, 2) is', L(data.x,data.y,ans1[-1],ans2[-1]),' at 1
    ↪  = ', ans1[-1],' at 2 = ', ans2[-1])
```

```
driverFunc(1e-11,10000,5)
driverFunc(1e-12,10000,5)
driverFunc(1e-12,60000,5)
driverFunc(1e-11,50000,1)
driverFunc(1e-12,500000,1)
```

3.2 Observations and results

- for α = 1e-11, 50000 iterations, sample size as 1, we obtain minimum value of $L(\theta 1, \theta 2)$ as 1576.359064921222 at θ 1,θ 2 = (49.999997163255735 , -0.008847093272812188)
- for α = 1e-12, 500000 iterations, sample size as 1, we obtain minimum value of $L(\theta 1, \theta 2)$ as 1576.341756013516 at θ 1,θ 2 = (49.99999716663549 , -0.008850122055620935)
- for α = 1e-11, 10000 iterations, sample size as 5, we obtain minimum value of $L(\theta 1, \theta 2)$ as 1576.3635658409871 at θ 1,θ 2 = (49.999997162434994 , -0.008857086949595316)
- for α = 1e-12 , 10000 iterations , sample size as 5, we obtain minimum value of $L(\theta 1, \theta 2)$ as 12063.8428099897 at θ 1,θ 2 = (49.99999829640553 , -0.005409764594738716)
- for α = 1e-12 , 60000 iterations , sample size as 5, we obtain minimum value of $L(\theta 1, \theta 2)$ as 1577.2456929899506 at θ 1,θ 2 = (49.99999719079826 , -0.008819833775102991)
- We observe that on increasing the learning rate we take bigger steps towards the minima and hence are able to converge quickly.
- For lower learning rates we need to increase the number of iterations to converge.
- For higher sample sizes we are able to converge in lower iterations as compared to lower sample sizes.
- Stochastic gradient descent is significantly faster than the standard gradient descent algorithm.

4 Q4

Steepest Gradient Descent is a method that uses a varying value of to find a direction d such that $f(x + d)$ is minimum. The algorithm for the same

Step 1. Pick an initial value for $\mathbf{x}^{(0)}$ and choose the maximum number of iterations

Step 2. Calculate the gradient of the given function at $\mathbf{x}^{(k)}$

Step 3. Calculate a step size $\alpha^{(k)}$ to mimimize $f(\mathbf{x}^{(k)} - \alpha^{(k)} \cdot \nabla f(\mathbf{x}^{(k)}))$

Step 4. Update as $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \alpha^{(k)} \cdot \nabla f(\mathbf{x}^{(k)})$

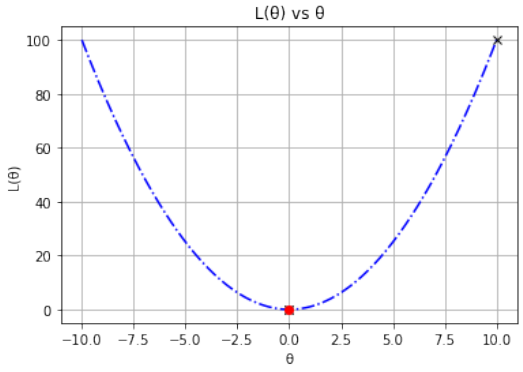
Step 5. Go to Step 2 till maximum number of iterations is reached

4.1 Part A

4.1.1 Code

```
def calculateL(theta):
    return np.square(theta)

def gradL(theta):
    return 2*theta

theta_plot = np.arange(-10,10.001,0.1)
values = calculateL(theta_plot)
iterations = 100

alpha = [0.1]
theta = [10]
for i in range(iterations-1):
    def L_alpha(alpha):
        return calculateL(theta[-1] - alpha*gradL(theta[-1]))
    alpha.append(optimize.golden(L_alpha))
    theta.append(theta[-1] - alpha[-1]*2*theta[-1])
theta = np.asarray(theta)
L = np.square(theta)

plt.plot(theta_plot, values, 'b-.')
plt.plot(theta, calculateL(theta), 'kx')
plt.plot(theta[-1], L[-1], 'ro')
plt.title('L() vs ')
plt.xlabel('')
plt.ylabel('L()')
plt.grid()
plt.show()
```

4.1.2 Plot

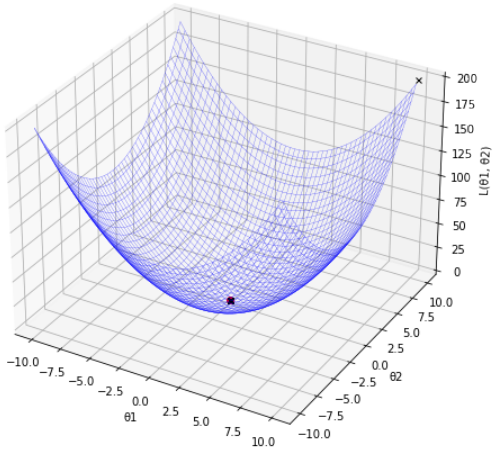


4.2 Part B

4.2.1 Code

```
def calculateL(theta1, theta2):
    return np.square(theta1) + np.square(theta2)

alpha1 = [0.1]
alpha2 = [0.1]
theta1 = [10]
theta2 = [10]
iterations = 100

for i in range(iterations - 1):
    theta = np.array([[theta1[i]],[theta2[i]]])
    grad_L = np.array([[2*theta1[i]], [2*theta2[i]]])
    alpha1.append(alpha1[-1] - (2*alpha1[-1]-1)/2)
    alpha2.append(alpha2[-1] - (2*alpha2[-1]-1)/2)
    theta = theta - np.dot(alpha[-1], grad_L)
    theta1.append(theta1[-1] - 2*alpha1[-1]*theta1[-1])
    theta2.append(theta2[-1] - 2*alpha2[-1]*theta2[-1])

x, y = np.meshgrid(np.arange(-10,10,0.1), np.arange(-10,10,0.1))
z = calculateL(x, y)

fig = plt.figure(figsize=[8,8])
ax = plt.axes(projection='3d')
ax.plot_wireframe(x, y, z, color='b', linewidth=0.2)
ax.plot(theta1[-1], theta2[-1], calculateL(theta1[-1], theta2[-1]), 'ro')
ax.plot(theta1, theta2, calculateL(theta1, theta2), 'kx', markersize=5)
ax.set_xlabel('1')
ax.set_ylabel('2')
ax.set_zlabel('L(1, 2)')
plt.show()
```

4.2.2 Plot



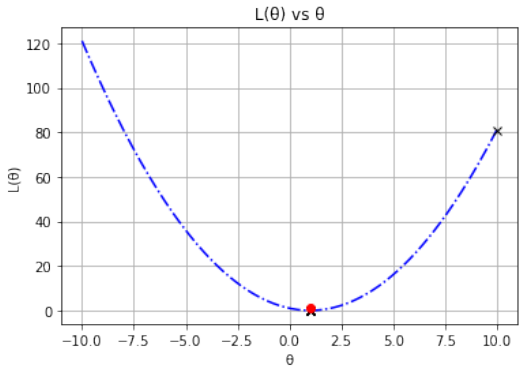
4.3 Part C

4.3.1 Code

```
def calculateL(theta):
    return np.square(theta-1)

def gradL(theta):
    return 2*theta

theta_plot = np.arange(-10,10.001,0.1)
values = calculateL(theta_plot)
iterations = 100

alpha = [0.1]
theta = [10]
for i in range(iterations-1):
    def L_alpha(alpha):
        return calculateL(theta[-1] - alpha*gradL(theta[-1]))
    alpha.append(optimize.golden(L_alpha))
    theta.append(theta[-1] - alpha[-1]*gradL(theta[-1]))
theta = np.asarray(theta)
L = np.square(theta)

plt.plot(theta_plot, values, 'b-.')
plt.plot(theta, calculateL(theta), 'kx')
plt.plot(theta[-1], L[-1], 'ro')
plt.title("L() vs ")
plt.xlabel('')
```

```
plt.ylabel('L()')
plt.grid()
plt.show()
```

4.3.2 Plot



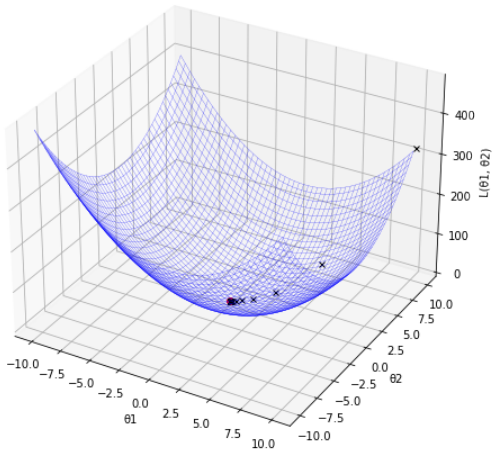
4.4 Part D

4.4.1 Code

```
def calculateL(theta1, theta2):
    return 2*np.square(theta1-np.ones(np.array(theta1).shape)) +
    ↪  2*np.square(theta2-np.ones(np.array(theta2).shape))

alpha1 = [0.1]
alpha2 = [0.1]
theta1 = [10]
theta2 = [10]
iterations = 100

for i in range(iterations - 1):
    theta = np.array([[theta1[i]],[theta2[i]]])
    grad_L = np.array([[4*theta1[i]], [4*theta2[i]]])
    alpha1.append(alpha1[-1] - (4*alpha1[-1]-1)/4)
    alpha2.append(alpha2[-1] - (4*alpha2[-1]-1)/4)
    theta = theta - np.dot(alpha[-1], grad_L)
    theta1.append(theta1[-1] - 2*alpha1[-1]*theta1[-1])
    theta2.append(theta2[-1] - 2*alpha2[-1]*theta2[-1])

x, y = np.meshgrid(np.arange(-10,10,0.1), np.arange(-10,10,0.1))
z = calculateL(x, y)

fig = plt.figure(figsize=[8,8])
ax = plt.axes(projection='3d')
ax.plot_wireframe(x, y, z, color='b', linewidth=0.2)
```

```
ax.plot(theta1[-1], theta2[-1], calculateL(theta1[-1], theta2[-1]), 'ro')
ax.plot(theta1, theta2, calculateL(theta1, theta2), 'kx', markersize=5)
ax.set_xlabel('1')
ax.set_ylabel('2')
ax.set_zlabel('L(1, 2)')
plt.show()
```

4.4.2 Plot



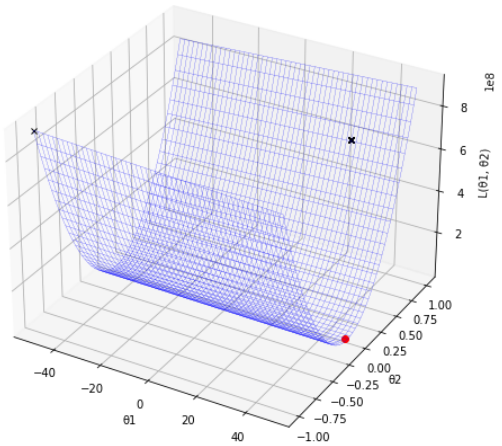
4.5 Part E

4.5.1 Code

```
def calculateL(x, y, theta1, theta2):
    return np.sum(np.square(y - (theta1 + (theta2*x))))

def gradL1(data_x, data_y, theta1, theta2):
    return np.sum(-2*(data_y-theta1-data_x*theta2))

def gradL2(data_x, data_y, theta1, theta2):
    return np.sum(-2*data_x*(data_y-theta1-data_x*theta2))

data = pd.read_excel('data.xlsx')
data_x = np.reshape(np.array(data.x), (-1, 1))
data_y = np.reshape(np.array(data.y), (-1, 1))

iterations = 10
theta = [np.array([-50, -1])]
L = lambda theta: np.sum(np.square(data_y - (theta[0] + (theta[1]*data_x))))

for i in range(iterations-1):
```

```
    gradL = np.array([gradL1(data_x, data_y, theta[-1][0], theta[-1][1]),
    ↪  gradL2(data_x, data_y, theta[-1][0], theta[-1][1])])
    H = nd.Hessian(L)([theta[-1][0], theta[-1][1]])
    theta.append(theta[-1]-np.dot(np.linalg.inv(H), gradL))

theta1 = np.arange(-50,50,0.05)
theta2 = np.arange(-1,1,0.001)
L = np.zeros((len(theta2),len(theta1)))

x, y = np.meshgrid(theta1, theta2)
for i in range(len(theta2)):
    for j in range(len(theta1)):
        L[i][j] = calculateL(data_x, data_y, theta1[j], theta2[i])
z = L

fig = plt.figure(figsize=[8,8])
ax = plt.axes(projection='3d')
ax.plot_wireframe(x, y, z, color='b', linewidth=0.2)
ax.plot(theta[-1][0], theta[-1][1], calculateL(data_x, data_y, theta[-1][0],
↪  theta[-1][1]), 'ro')
ax.plot([theta[i][0] for i in range(len(theta))], [theta[i][1] for i in
↪  range(len(theta))], calculateL(data_x, data_y, [theta[i][0] for i in
↪  range(len(theta))], [theta[i][1] for i in range(len(theta))]), 'kx',
↪  markersize=5)
ax.set_xlabel('1')
ax.set_ylabel('2')
ax.set_zlabel('L(1, 2)')
plt.show()
```

4.5.2 Plot



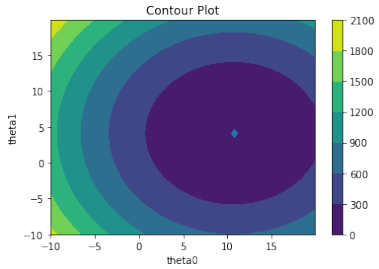
For Part E, we have used the Hessian Matrix (**H**), which is a square matrix of second-order partial derivatives of a scalar-valued function. Using Newton Optimization Technique that uses second-order Taylor Series expansion, we get $\delta\theta = -\mathbf{H}^{-1}\nabla f(\mathbf{x})$.
Since calculation of **H** is a computational intensive task, the method can be optimized by either using an iterative method to find **H** or by calculating only the upper or lower triangular matrix of **H** (as **H** is a symmetric matrix.)

5 Q5

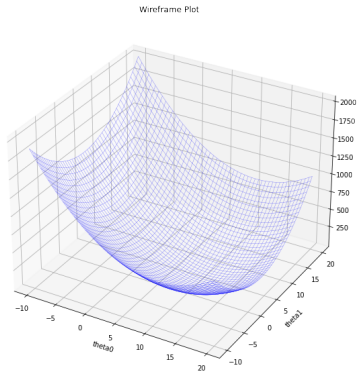
5.1 Part 1

```
x = np.array([[1],[3],[6]])
y = np.array(([6],[10],[16]))
mu = np.mean(x)
sig = np.std(x)
x = (x - mu)/sig
theta0 = np.arange(-10,20,0.1)
theta1 = np.arange(-10,20,0.1)
xx, yy = np.meshgrid(theta0, theta1)
L = np.zeros((np.size(theta1),np.size(theta0)))
for i in range(np.size(theta1)):
    for j in range(np.size(theta0)):
        L[i][j] = np.sum(np.square(y - (theta0[j] + (theta1[i]*x))))
minima = np.argwhere(L == np.min(L))
plt.contourf(xx, yy,L)
plt.colorbar()
plt.plot(theta0[minima[0,1]],theta1[minima[0,0]],'d')
plt.title('Contour Plot')
plt.xlabel('theta0')
plt.ylabel('theta1')
plt.show()
fig = plt.figure(figsize=(10, 10))
ax = plt.axes(projection='3d')
ax.plot_wireframe(xx, yy, L, color='blue', linewidth=0.2)
plt.title('Wireframe Plot')
plt.xlabel('theta0')
plt.ylabel('theta1')
plt.show()
```

The output of the above mentioned code is as given below:



(a)



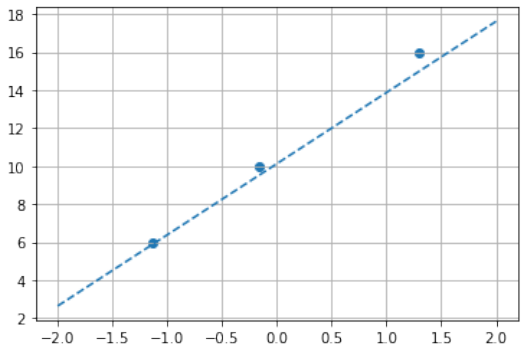
(b)

The first figure is the contour plot in $\theta 1$ *and* $\theta 0$ space of residual sum of squares. The second plot shows the 3-D meshgrid plot for the same.

5.2 Part 2

```
temp = np.copy(x)
x2 = np.ones((np.size(temp,0),2))
x2[:,1] = temp[:,0]
iterations = 60
alpha = 0.01
theta = np.array([[-10],[-10]])
theta0_vals = [-10]
theta1_vals = [-10]
for i in range(iterations - 1):
 y_hat = np.dot(x2,theta)
 theta = theta - (alpha*2)*np.dot(x2.T,(y_hat - y))
 theta0_vals.append(theta[0,0])
 theta1_vals.append(theta[1,0])

print('the value of theta is:',theta)
plt.figure()
axes = plt.gca()
x_val = np.linspace(-2,2,100)
y_val = theta[0,0] + theta[1,0] * x_val
plt.plot(x_val, y_val, '--')
plt.scatter(x, y)
plt.plot()
plt.show()
#GIF MAKER
filenames = []
for i in range(iterations):
 # plot the line chart
```

```
 plt.contourf(xx, yy,L)
 plt.colorbar()
 for j in range(i+1):
  plt.plot(theta0_vals[j],theta1_vals[j],'ro')
 plt.title('Contour Plot')
 plt.xlabel('theta0')
 plt.ylabel('theta1')

 # create file name and append it to a list
 filename = f'{i}.png'
 filenames.append(filename)

 # save frame
 plt.savefig(filename)
 plt.close()# build gif

with imageio.get_writer('Ques5.gif', mode='I') as writer:
 for filename in filenames:
   image = imageio.imread(filename)
   writer.append_data(image)

# Remove files
for filename in set(filenames):
 os.remove(filename)
```



The plot shown above shows the value of y as a function of x which is a straight line with $\theta 1$ as slope and $\theta 0$ as the y-intercept. Link to the animated GIF: click here
This GIF shows 60 iterations of gradient descent algorithm. We notice that the gap between consecutive iterations reduces as we slowly approach the minima.