IE406 Machine Learning

Lab Assignment - 02

Group 16

Student ID: 201801046 Name: Ravi Patel
Student ID: 201801057 Name: Ayan Khokhar
Student ID: 201801432 Name: Prakhar Maheshwari
Student ID: 201801452 Name: Jaydeep Machhi
Student ID: 201801459 Name: Aakash Panchal

Question 1

Implement polynomial regression as a special case of linear regression. First generate some data as follows. Now, we want to learn a polynomial function of degree p on this dataset, i.e. $y = q_0 + q_1.x_1 + q_2.x_2 + ...q_p.x_p$ We can use the linear regression implementations for doing so, by transforming the data set and creating the matrix X containing columns corresponding to $x0, x1, x2, ..., x_p$. Using any of your implementations above learn the regression coefficients for p = 5 and p = 4. How close are your coefficients for p = 5 to the ones used to generate the data?

Answer

code

First generating data without the error term.

```
# Data Generation
x = np.arange(0, 20.1, 0.1)
np.random.seed(0)
y = 1*x**5 + 3*x**4 - 100*x**3 + 8*x**2 -300*x - 1e5
```

```
#For p = 4 degree polynomial
newX = []
mul = 1
for i in range(0, 5):
    mul = mul * x
    newX.append(mul)

newX = np.array(newX)
newX = newX.T
# print((newX.shape))
```

```
X_train, X_test, y_train, y_test = train_test_split(newX, y, test_size
    =0.33, random_state=42)

reg = LinearRegression().fit(X_train, y_train)

print(f'Regression Coefficients = {reg.coef_}')
print(f'Regression Intercept {reg.intercept_}')
print(f'Regression Score = {reg.score(X_test, y_test)}')
predicted_y = reg.predict(newX)
mse = mean_squared_error(y, predicted_y, squared = False)
# print(predicted_y)
print(f'Root Mean Squared Error = {mse}')
```

```
#For p = 5 degree polynomial
newX = []
mul = 1
for i in range(0, 6):
    mul = mul * x
    newX.append(mul)

newX = np.array(newX)
newX = newX.T
# print((newX.shape))
X_train, X_test, y_train, y_test = train_test_split(newX, y, test_size
    =0.33, random_state=42)

reg = LinearRegression().fit(X_train, y_train)

print(f'Regression Coefficients = {reg.coef_}')
print(f'Regression Intercept {reg.intercept_}')
print(f'Regression Score = {reg.score(X_test, y_test)}')
predicted_y = reg.predict(newX)
mse = mean_squared_error(y, predicted_y, squared = False)
# print(predicted_y)
print(f'Root Mean Squared Error = {mse}')
```

Now we add the error term as follows in the generation of data set:

```
# Data Generation
x = np.arange(0, 20.1, 0.1)
np.random.seed(0)
y = 1*x**5 + 3*x**4 - 100*x**3 + 8*x**2 -300*x - 1e5 + np.random.randn(len
    (x))*1e5
```

Result

```
Regression Coefficients = [-300.    8. -100.    3.    1.]
Regression Intercept -100000.00000000431
Regression Score = 1.0
Root Mean Squared Error = 1.79688320344248e-09
```

Figure 1: Result for p = 4 and no error term

```
Regression Coefficients = [-3.00000000e+02  8.00000000e+00 -1.00000000e+02  3.00000000e+00
  1.00000000e+00  3.55271368e-15]
Regression Intercept -100000.00000004406
Regression Score = 1.0
Root Mean Squared Error = 4.979079823505016e-08
```

Figure 2: Result for p = 5 and no error term

```
Regression Coefficients = [-9.13511917e+04  1.36268352e+04 -4.14618192e+02 -3.36189972e+01
  2.45206708e+00]
Regression Intercept 30933.713570617023
Regression Score = 0.9782024578345158
Root Mean Squared Error = 97021.26616028769
```

Figure 3: Result for p = 4 and random error term introduced

```
Regression Coefficients = [-1.39455103e+05  3.85506445e+04 -5.46473224e+03  4.42682862e+02
 -1.85689676e+01  3.50879604e-01]
Regression Intercept 50134.468212532636
Regression Score = 0.9776815417322802
Root Mean Squared Error = 97213.15953592927
```

Figure 4: Result for p = 5 and random error term introduced

Observation/ Justification

As we can see from the above results when there is no error term present, the coefficients obtained are exactly the same as Y array, the RMS error is approximately zero and regression score is exactly 1.

When we introduced the error term, the regression score now changes to approximately 0.97, the RMS error increases to the order of error term and coefficients are no longer same as Y array.

Question 2(a))

Find minima of following functions using Gradient Descent method with learning rate 0.01 and 0.1 and different number of iterations. Try choosing a large value of learning rate and test the convergence. For $L_5(\theta)$, use the data file.

Minimize the function $L(\theta) = \theta^2$

Answer

code

```
#Helper Functions for Gradient Descent Method

def gradient_descent_iterative(gradient_func, start, alpha, iterations = 1
    e8, err = 1e-09):
```

```
  # Initialize the xn with start values
  xn = start

  # Loop until the iterations get finished
  for i in range(int(iterations)):
    # Calculate the gradient function
    xnew = - alpha * gradient_func(xn)

    # Check that the convergence is achieved
    if np.abs(xnew) <= err:
      break
    xn += xnew

  return xn

def gradient_descent_iterative_two_var(gradient_func, start, alpha,
    iterations = 1e8, err = 1e-09):
  # Initialize the xn with start values
  xn = start

  # Loop until the iterations get finished
  for i in range(int(iterations)):
    # Calculate the gradient function
    xnew = - alpha * np.array(gradient_func(xn))


    # Check that the convergence is achieved
    if np.all(np.abs(xnew) <= err):
      break
    xn += xnew

  return xn
```

Listing 1: Gradient Descent Helper Functions

Result

For learning rate $(\alpha) = 0.1$, and function as $L(\theta) = \theta^2$ the minimum value of $L(\theta) = 2.3587265155134666e-17$ is obtained at $\theta = 4.856672230564326e-09$ and for learning rate $(\alpha) = 0.01$ the minima is obtained at $\theta = 4.979102145973068e-08$ and the value of $L(\theta) = 2.479145818003361e-15$. The value of theta ans $L(\theta)$ is so small that is can be approximated as 0 which is the actual minima.

Observation/ Justification

It is easy to see by analytical calculations that the function $L(\theta)$ has minima at $\theta = 0$.

Question 2(b)

Minimize the function $L(\theta) = \theta_1^2 + \theta_2^2$

Answer

code

```

alpha = 0.01 # Learning Rate

def my_func(theta):
  return [2 * theta[0], 2 * theta[1]]

def func(theta):
  return theta[0] * theta[0] + theta[1] * theta[1]

ans = gradient_descent_iterative_two_var(my_func, [5, 5], alpha)
print(f' 1  = {ans[0]} and  2  = {ans[1]}')
print(f'Minima value of L() = {func(ans)}')
```

Listing 2: Question 2(b)

Result

For learning rate $(\alpha) = 0.01$, and function as $L(\theta) = \theta_1^2 + \theta_2^2$ $\theta_1 = 4.979102145973068e-08$ and $\theta_2 = 4.979102145973068e-08$ Minima value of $L(\theta) = 4.958291636006722e-15$.

For learning rate $(\alpha) = 0.1$, and function as $L(\theta) = \theta_1^2 + \theta_2^2$ the minima is at $\theta_1 = 4.856672230564326e{-}09$ and $\theta_2 = 4.856672230564326e{-}09$ Minima value of L() = 4.717453031026933e-17

Observation/ Justification

It is easy to see by analytical calculations that the function $L(\theta)$ has minima at $\theta_1 = 0$ and $\theta_2 = 0$.

Question 2(c)

Minimize the function $L(\theta) = (\theta - 1)^2$

Answer

code

```
alpha = 0.01 # Learning Rate

def my_func(theta):
  return 2 * (theta - 1)

def func(theta):
  return (theta - 1) ** 2
ans = gradient_descent_iterative(my_func, 5, alpha)
print(f'   = {ans}')
```

```
print(f'Minima value of L(  ) = {func(ans)}')
```

Listing 3: Question 2(c)

Result

For learning rate $(\alpha) = 0.01$, and function as $L(\theta) = (\theta-1)^2$ has minima at $\theta = 1.0000000497455448$ and Minima value of $L(\theta) = 2.474619226904864e-15$

For learning rate $(\alpha) = 0.1$, and function $L(\theta) = (\theta-1)^2$ has minima at $\theta = 1.0000000048566722$ and Minima value of $L(\theta) = 2.3587264620615693e-17$

Observation/ Justification

It is easy to see by analytical calculations that the function $L(\theta)$ has minima at $\theta = 1$.

Question 2(d)

Minimize the function $L(\theta) = 2 \cdot (\theta_1 - 1)^2 + 2 \cdot (\theta_2 - 1)^2$

Answer

code

```
alpha = 0.01 # Learning Rate

def my_func(theta):
  return [4 * (theta[0] - 1), 4 * (theta[1] - 1)]

ans = gradient_descent_iterative_two_var(my_func, [5,5], alpha)
# print(ans)

def func(theta):
  return  (2* (theta[0] - 1) * (theta[0] - 1)) + (2* (theta[1] - 1) * (
   theta[1] - 1))

print(f' 1  = {ans[0]} and  2  = {ans[1]}')
print(f'Minima value of L(  ) = {func(ans)}')
```

Listing 4: Question 2(d)

Result

```
θ1 = 1.0000000247537435 and θ2 = 1.0000000247537435
Minima value of L(θ) = 2.450991268939751e-15
```

Figure 5: For learning rate $(\alpha) = 0.01$

```
θ1 = 1.0000000019249191 and θ2 = 1.0000000019249191
Minima value of L(θ) = 1.4821254366284845e-17
```

Figure 6: For learning rate $(\alpha) = 0.1$

Question 2(e)

Find minimum of the function $L(\theta) = \sum_{i=1}^{m} \left(y^{(i)} - (\theta_0 + \theta_1 \cdot x^{(i)})\right)^2$

code

```
#First we read the data
df = pd.read_excel('/content/data.xlsx')

x = df['x']
y = df['y']
x = np.array(x)
x = x - np.mean(x)
x = x/np.std(x)
y = np.array(y)
```

Listing 5: Question 2(e)

```
def gradient_descent_iterative_5(gradient_func, params, start, alpha,
    iterations = 5000, err = 1e-06):
  xn = start
  x = params[0]
  y = params[1]

  for i in range(int(iterations)):
    xnew = - alpha * np.array(gradient_func(xn, x, y))
    if np.all(np.abs(xnew) <= err):
      break
    xn += xnew

  return xn

def lsgradient(theta, x, y):
  val = theta[0] + theta[1] * x - y
  val = val * 2
  return [val.sum(), (val*x).sum()]
```

Listing 6: Question 2(e) gradient descent function

Result

```
θ1 = 23.718084621396557 and θ2 = -6.86699541572586
Minima value of L(θ) = 1572.6503668923142
```

Figure 7: For learning rate $(\alpha) = 0.01$

```
θ1 = 534.9 and θ2 = -40.0995144084012
Minima value of L(θ) = 24668239.59086284
```

Figure 8: For learning rate $(\alpha) = 0.1$

Observation/ Justification

For the case of $\alpha = 0.01$ the function has shown convergence to the minima point but for $\alpha = 0.1$ the function diverges to large values and never achieves minima. So a proper alpha value is required for convergence.

Question 3

Find minimum of the function $L(\theta) = \sum_{i=1}^{m} \left(y^{(i)} - (\theta_0 + \theta_1 \cdot x^{(i)})\right)^2$ using the Stochastic Gradient Descent method (Take the data from the data file). Choose different learning rates and number of iterations.

Answer

code

```
1
2  # Stochastic Gradient Descent Method with Mini Batches
3
4  def stochastic_gradient_descent(gradient_func, params, start, alpha,
       batch_size = 1, iterations = 5000, err = 1e-06):
5
6    # float data type in numpy format
7    dataType = np.dtype('float64')
8
9    # converting x and y into numpy arrays
10   x = np.array(params[0], dtype = dataType)
11   y = np.array(params[1], dtype = dataType)
12
13   # check if x and y have same shape
14   n = x.shape[0]
15   if(y.shape[0] != n) :
16     raise ValueError("'X' and 'Y' must have same length")
17
18   xy = np.c_[x.reshape(n, -1), y.reshape(n, 1)]
19
20   # Initial values of the variables
21   data = np.array(start, dtype = dataType)
22
23   # Initialize the random number generator and set the proper seed
24   rng = np.random.default_rng(seed=31032000)
25
26   # convert learning rate into numpy array
27   alpha = np.array(alpha, dtype = dataType)
28
```

```
  # check if batch size lies between 1 to n (number of observations)
  if not 0 < batch_size <= n :
    raise ValueError("Batch size must lie between 1 and 'Number of
    Observations'")

  # Main gradient descent iterations loop
  for itr in range(iterations) :

    # randomly shuffle x and y
    rng.shuffle(xy)

    for start in range(0, n, batch_size) :
      stop = start + batch_size
      currX, currY = xy[start:stop, :-1], xy[start:stop, -1:]

      # calculate the error vector for current batch
      currGrad = np.array(gradient_func(data, currX, currY), dataType)
      currError = -alpha * currGrad

      # check if all variables have small enough error
      if (np.all(currError) <= err) :
        break

      # Updating the values of the variables
      data += currError

  return data

# Function to calculate the differentiation

def lsgradient(theta, x, y):
  val = theta[0] + theta[1] * x - y
  val = val * 2
  return [val.sum(), (val*x).sum()]

alpha = 0.01

ans = stochastic_gradient_descent(lsgradient, [x,y], [5,5], alpha, 25,
    5000)

print(ans)

# Function to compute the value of L(  )
def func(theta, x, y):
  return np.sum((y - (theta[0] + theta[1] * x)) ** 2)

print(f'Minima value of L(  ) = {func(ans, x, y)}')
```

Listing 7: Question 3

Result

$\theta_0 = 23.65233444$ and $\theta_1 = -6.69984829$ give the optimal value of $L(\theta)$, which is $L(\theta) = 1575.6829311649976$. This value is almost the same as the one which we get by using simple gradient descent. In our case, we used stochastic gradient descent with mini batch size of 25.

```
alpha = 0.1

ans = stochastic_gradient_descent(lsgradient, [x,y], [5,5], alpha, 25, 1)
print(ans)
def func(theta, x, y):
  return np.sum((y - (theta[0] + theta[1] * x)) ** 2)

print(f'Minima value of L(θ) = {func(ans, x, y)}')
```

```
[-3564.66405431  1678.69815446]
Minima value of L(θ) = 1477457500.433954
```

Figure 9: For Learning Rate $\alpha = 0.1$

Observation/ Justification

This algorithm is called stochastic gradient descent algorithm because the mini batch which we choose is selected randomly. If we set batch size = number of observation, then this algorithm is same as simple gradient descent algorithm. If we increase the value of batch size then the accuracy in result increases but computation also increases. So, there must be some trade-off between computation time and accuracy while choosing batch size.

While implementing this algorithm, we need to take care of some corner cases such as all the variables in the dataset have the same dimensions, batch size lies from 1 to number of observations(variables).

For learning rates (α) 0.05 or 0.1 the function value does not converge to the minima (i.e. it diverges).

Question 4

Find the minima of following functions using the Steepest Descent method.

We have created two functions, which we will be using to calculate the minima of the provided functions:

1. Single parameter Steepest Gradient Descent method

```
import sympy as syp

def steepest_gradient_descent_one_parameter(iterations, L):
    optimum_theta = 10
    optimum_alpha = 0.1

    dL_dtheta = syp.diff(L, theta)

```

```
    alpha = syp.symbols('a')
    L_alpha = L.subs(theta, theta - (alpha * dL_dtheta))
    L_alpha_d1 = syp.diff(L_alpha, alpha)
    L_alpha_d2 = syp.diff(L_alpha_d1, alpha)
    ratio = L_alpha_d1 / L_alpha_d2

    for i in range(iterations):
        for j in range(iterations):
            optimum_alpha = optimum_alpha - ratio.subs({alpha:
   optimum_alpha, theta: optimum_theta})
        optimum_theta = optimum_theta - optimum_alpha * dL_dtheta.
   subs(theta, optimum_theta)
    print('Optimal value of theta : ', optimum_theta)
    print('Minimum value of L(theta) : ', L.subs(theta, optimum_theta
   ))
```

2. Two parameter Steepest Gradient Descent method

```
def steepest_gradient_descent_two_parameter(iterations, L):
    optimum_theta0 = 10
    optimum_theta1 = 10
    optimum_alpha = 0.1

    dL_dtheta0 = syp.diff(L, theta0)
    dL_dtheta1 = syp.diff(L, theta1)

    alpha = syp.symbols('a')
    L_alpha = L.subs({theta0 : theta0 - (alpha * dL_dtheta0), theta1
   : theta1 - (alpha * dL_dtheta1)})
    L_alpha_d1 = syp.diff(L_alpha, alpha)
    L_alpha_d2 = syp.diff(L_alpha_d1, alpha)
    ratio = L_alpha_d1 / L_alpha_d2

    for i in range(iterations):
        for j in range(iterations):
            optimum_alpha = optimum_alpha - ratio.subs({alpha :
   optimum_alpha, theta0 : optimum_theta0, theta1 : optimum_theta1})
        optimum_theta0 = optimum_theta0 - optimum_alpha * dL_dtheta0.
   subs({theta0 : optimum_theta0, theta1 : optimum_theta1})
        optimum_theta1 = optimum_theta1 - optimum_alpha * dL_dtheta1.
   subs({theta0 : optimum_theta0, theta1 : optimum_theta1})
    print('Optimal value of theta0: ', optimum_theta0)
    print('Optimal value of theta1: ', optimum_theta1)
    print('Minimum value of L(theta0, theta1): ', L.subs({theta0 :
   optimum_theta0, theta1 : optimum_theta1}))
```

4-a

$L_1(\Theta) = \theta^2$

```
theta = syp.symbols('theta')
steepest_gradient_descent_one_parameter(25, np.square(theta))
```

Optimal value of $\theta = 0$
Optimal value of $L_1(\Theta) = 0$

4-b

$L_2(\Theta) = \theta_1^2 + \theta_2^2$

```
theta0 = syp.symbols('theta0')
theta1 = syp.symbols('theta1')
steepest_gradient_descent_two_parameter(25, np.square(theta0) + np.square(
    theta1))
```

Optimal value of $\theta_1 = 0$
Optimal value of $\theta_2 = 0$
Optimal value of $L_2(\Theta) = 0$

4-c

$L_3(\Theta) = (\theta - 1)^2$

```
theta = syp.symbols('theta')
steepest_gradient_descent_one_parameter(25, np.square(theta - 1))
```

Optimal value of $\theta = 1$
Optimal value of $L_3(\Theta) = 0$

4-d

$L_4(\Theta) = (\theta_1 - 1)^2 + (\theta_2 - 1)^2$

```
theta0 = syp.symbols('theta0')
theta1 = syp.symbols('theta1')
L = 2 * np.square(theta0 - 1) + 2 * np.square(theta1 - 1)
steepest_gradient_descent_two_parameter(25, L)
```

Optimal value of $\theta_1 = 1$
Optimal value of $\theta_2 = 1$
Optimal value of $L_4(\Theta) = 0$

4-e

$L_5(\Theta) = \sum_{i=1}^{m} (y^{(i)} - (\theta_0 + \theta_1 * x^{(i)}))^2$

```
location = "data.csv"
df = pd.read_csv(location)

x = np.array(df['x'])
x = x - np.mean(x)
x = x / np.std(x)

y = np.array(df['y'])

theta0 = syp.symbols('theta0')
```

```
theta1 = syp.symbols('theta1')
L = np.sum((y - (theta0 + theta1 * x)) ** 2)
steepest_gradient_descent_two_parameter(4, L)
```

Optimal value of θ_0 = 23.7873552232581
Optimal value of θ_1 = -6.88576009845154
Optimal value of $L_5(\Theta)$ = 1573.13451024758

Observations

We can see that **Steepest Gradient Descent** method is more efficient than other gradient descent method. Even for same learning rate values it converges way faster(very small number of iterations e.g. less than 4 leads to convergence) and accuracy is higher.

Question 5(a)

Create a contour plot in the θ_0 and θ_1 space of residual sum of squares.

Answer

code

```
start = -50
stop = 50
step = 0.1
theta0 = np.arange(start, stop, step)
theta1 = np.arange(start, stop, step)

new_theta = np.meshgrid(theta0, theta1)
new_theta = np.array(new_theta)

Ltheta = 0

# Residual Calculation
for i in range(len(X)):
  Ltheta = Ltheta + (Y[i] - (new_theta[0] + new_theta[1] * X[i])) ** 2

plt.contour(new_theta[0], new_theta[1], Ltheta)
plt.title('Contour Plot of L(  )')
plt.xlabel('$ _1$')
plt.ylabel('$ _2$')
```

Result

The sum of squares of residuals is calculated as $L_5(\Theta) = \sum_{i=1}^{m}(y^{(i)} - (\theta_0 + \theta_1 * x^{(i)}))^2$ that is the sum of test values - predicted values over the number of examples m.

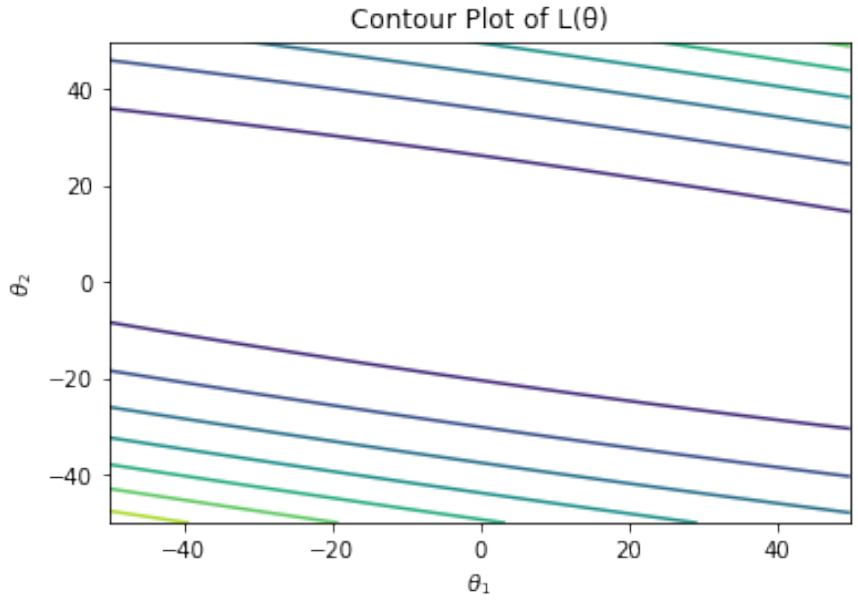

Figure 10: Contour Plot of $L(\theta)$ of sum of squares of Residuals

Observation/Justification

Question 5(b)

Create a Matplotlib animation where the plot contains two columns: the first one being the contour plot and the second one being the linear regression fit on the data. The different frames in the animation correspond to different iterations of gradient descent applied on the dataset to learn θ_0 and θ_1. For each iteration, draw the current value of q_0 and q_1 on the contour plot and also an arrow to the next q_0 and q_1 as learnt by gradient update rule. Correspondingly draw the y $=$ $_0$ $+$ $_1$ $\times$ x line on the other subplot showing the scatter plot. The overall title of the plot shows the iteration number and the residual sum of squares. You are free to use any gradient descent implementation i.e. your own or using libraries like scikit learn

Answer

code

```
# Gradient Descent Helper function
def gradient_descent_iterative_with_plotting(gradient_func, params, start,
    alpha, iterations = 5000, err = 1e-06):
  xn = start
  x = params[0]
  y = params[1]

  for i in range(int(iterations)):
    if i < 10:
      plot_function(xn, i)
```

```
10
11      if i > 300 and i < 305:
12        plot_function(xn, i)
13
14      xnew = - alpha * np.array(gradient_func(xn, x, y))
15      if np.all(np.abs(xnew) <= err):
16        break
17      xn += xnew
18    return xn
19
20  # Declaring Gradient Function of L(theta)
21  def lsgradient(theta, x, y):
22    val = theta[0] + theta[1] * x - y
23    return [val.mean(), (val*x).mean()]
24
25  def func(theta, x, y):
26    return np.sum((y - (theta[0] + theta[1] * x)) ** 2)
27
28
29  def plot_function(params, iteration_number):
30    step = 0.1
31    start = -10
32    end = 10
33    x_axis_1 = np.arange(start,end,step)
34    y_axis_1 = np.arange(start,end,step)
35    x_axis_2 = np.arange(start,end,step)
36    y_axis_2 = params[0] + params[1] * x_axis_2
37    fig, (ax1,ax2) = plt.subplots(1,2, figsize=(6,4))
38    fig.suptitle(f'Iteration Number {iteration_number}')
39    new_theta = np.meshgrid(x_axis_1, y_axis_1)
40
41    Ltheta = 0
42
43    for i in range(len(X)):
44      Ltheta = Ltheta + (Y[i] - (new_theta[0] + new_theta[1] * X[i])) ** 2
45    # plt.figure(2)
46    ax1.contour(new_theta[0], new_theta[1], Ltheta)
47    ax1.plot(params[0],params[1], 'o')
48    ax1.annotate('Theta', (params[0], params[1]))
49    ax1.arrow(params[0], params[1], (4-params[0]), (2-params[1]),
       length_includes_head = True)
50    ax2.plot(x_axis_2, y_axis_2)
51    ax2.scatter(X, Y)
52    plt.show()
53
54
55  # Learning Rate
56  alpha = 0.1
57
58  ans = gradient_descent_iterative_with_plotting(lsgradient, [X, Y], [5,5],
       alpha)
59  print(f' 0  = {ans[0]} and  1  = {ans[1]} ')
60  print(f'L(  ) = {func(ans, X, Y)}')
```

Result

```
θ0 = 4.000037570376705 and θ1 = 1.9999916901412398
L(θ) = 1.1669817578009631e-09
```

Figure 11: For Learning Rate $\alpha = 0.1$

Visualization of Gradient Descent algorithm with help of contour plots and Regression Line

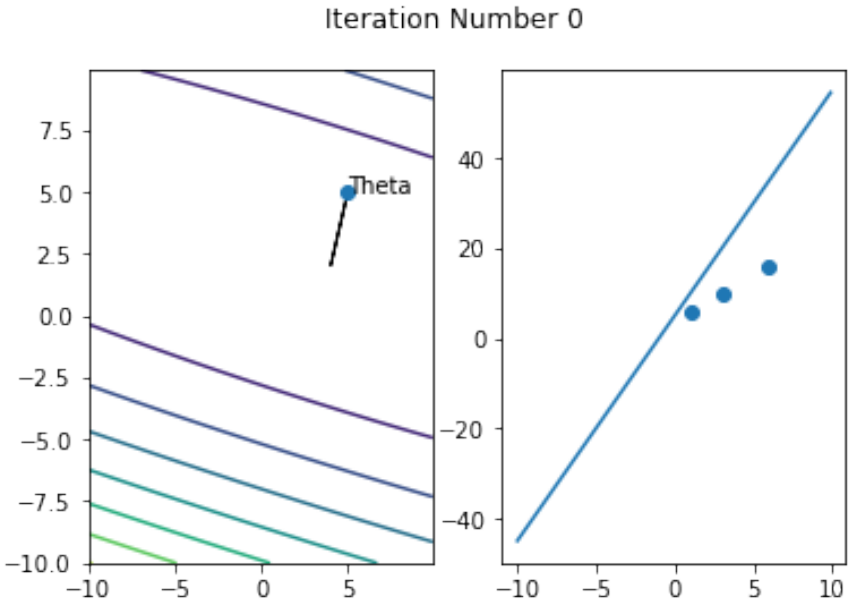

Figure 12: Iteration 1

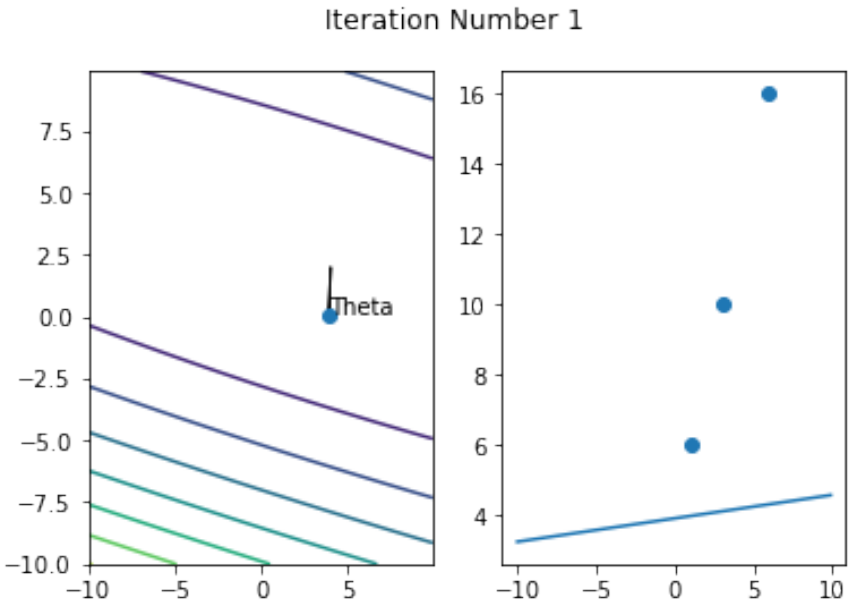

Figure 13: Iteration 2

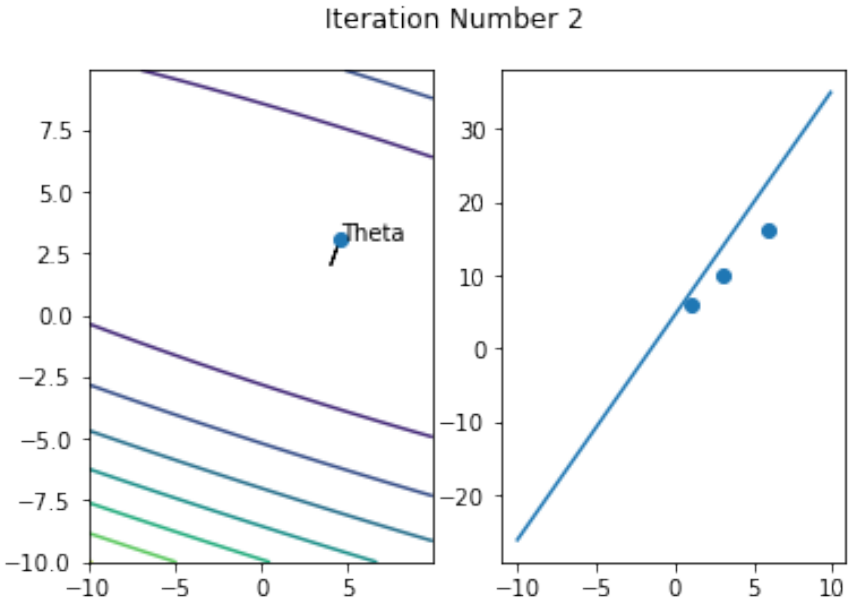

Figure 14: Iteration 3

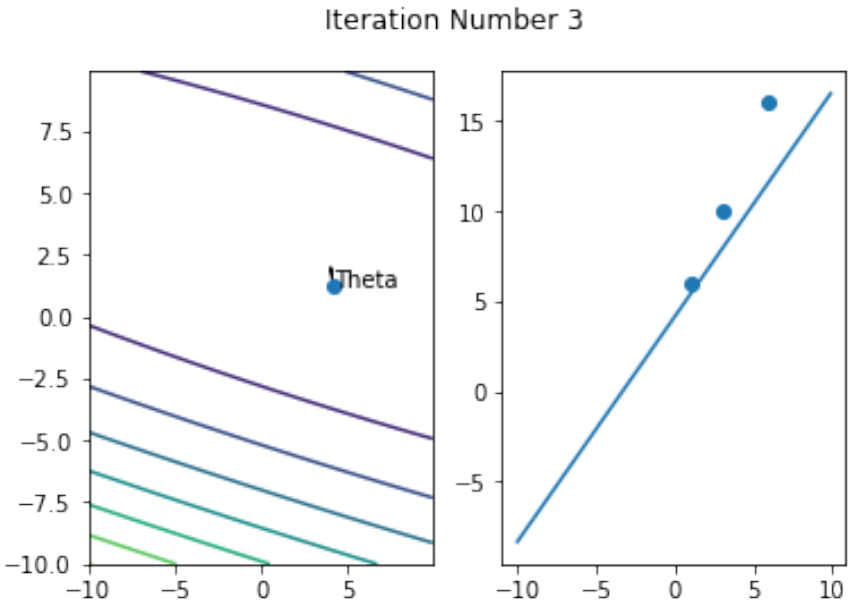

Figure 15: Iteration 4

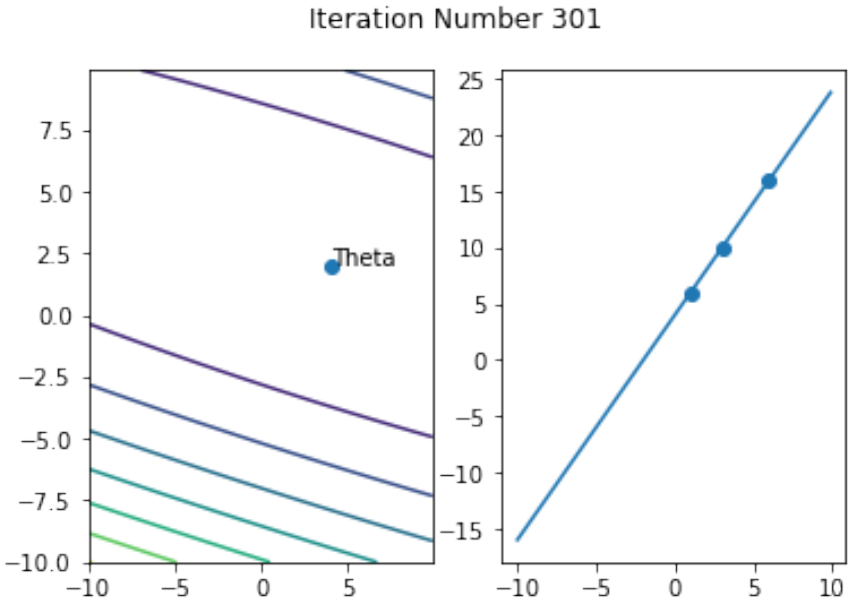

Figure 16: Iteration 301

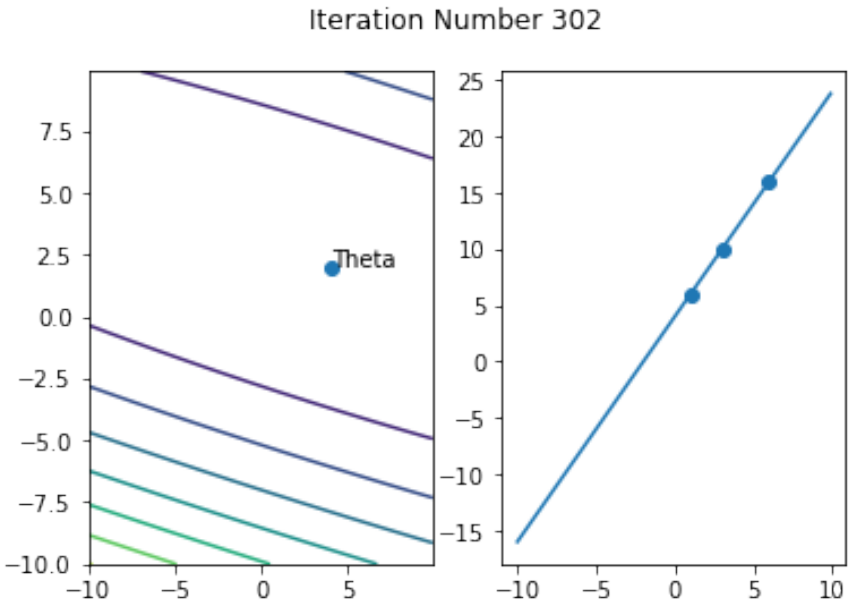

Figure 17: Iteration 302

Observation/Justification

The animation for the algorithm is as Algorithm GIF.

From the above visualization of the algorithm we can see that how the algorithm converges to the minima starting from some arbitrary point having some coordinates of $(\theta_1, \theta 2)$.

After some iterations the Regression Line totally fits our points and thus achieving the minima state.