PURVANG LAPSIWALA

UIN = 662689378

Answer : 2

(b)

Given function is f(x, y) = − log(1 − x − y) − log x − log y.

Initial point w0 = [0.1, 0.75]

Eeta = 1

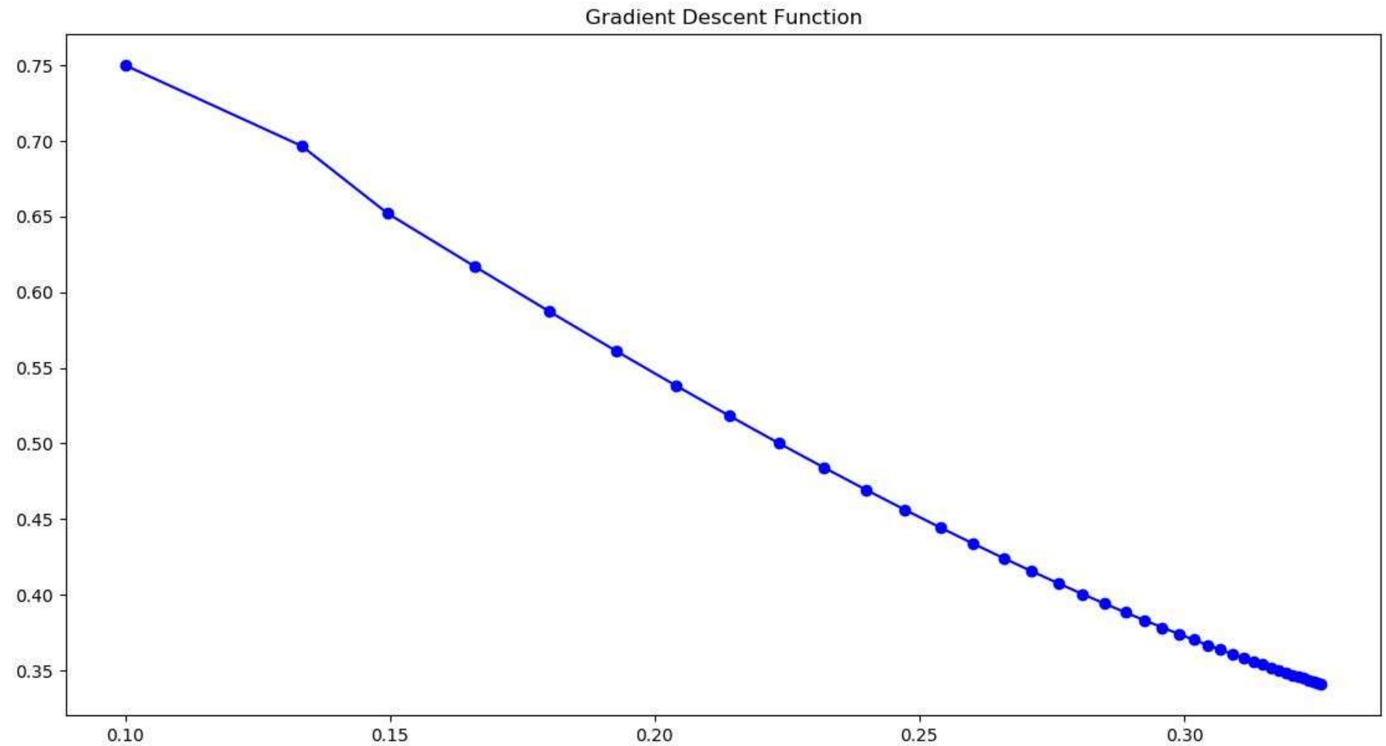
Time taken for gradient descent algorithm to converge is 0.0184.

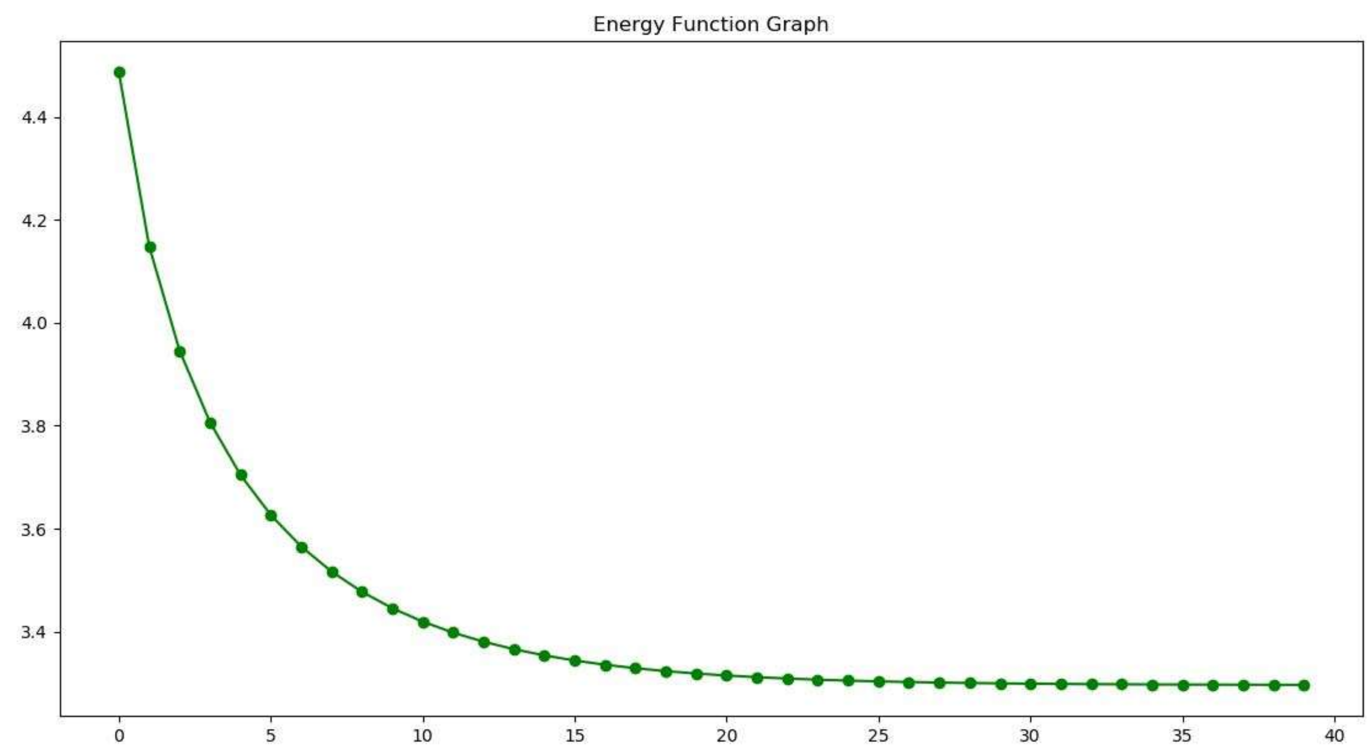
CODE:

```
import numpy as np

import matplotlib.pyplot as plt

from numpy.linalg import inv

import time


start = time.clock()


x = 0.1

y = 0.75

learning_rate = 1


w = np.array([x,y])


w_x = []

w_y = []

energy_matrix = []
```

```
while (w[0]+w[1]<1) and (w[0]>0) and (w[1]>0):
    energy = - np.log(1-w[0]-w[1]) - np.log(w[0]) - np.log(w[1])
    energy_matrix.append(energy)

    w_x.append(w[0])
    w_y.append(w[1])



    grad_x = 1/(1-w[0]-w[1]) - 1/(w[0])
    grad_y = 1/(1-w[0]-w[1]) - 1/(w[1])
    gradient = np.array([grad_x, grad_y])

    update = learning_rate * gradient

    if np.linalg.norm(w - np.subtract(w,update)) < 0.001:
        break
    else:
        w = np.subtract(w,update)

end = time.clock()

print ("Time taken for gradient descent: ", round((end-start), 4))
plt.title("Gradient Descent Function")
plt.plot(w_x, w_y, 'bo-')
plt.show()
plt.title("Energy Function Graph")
```

```
plt.plot(energy_matrix,'go-')

plt.show()
```





Using Newton's Method

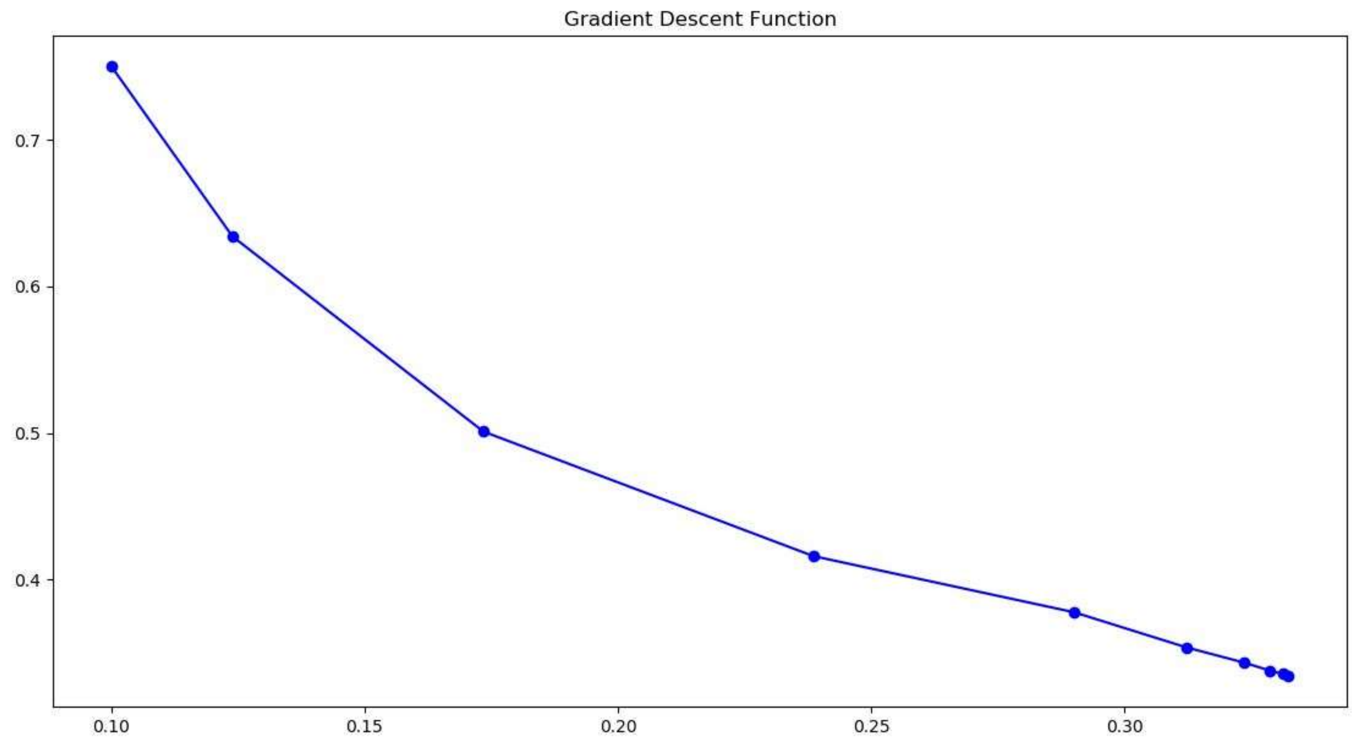
W0 = [0.1,0.75]

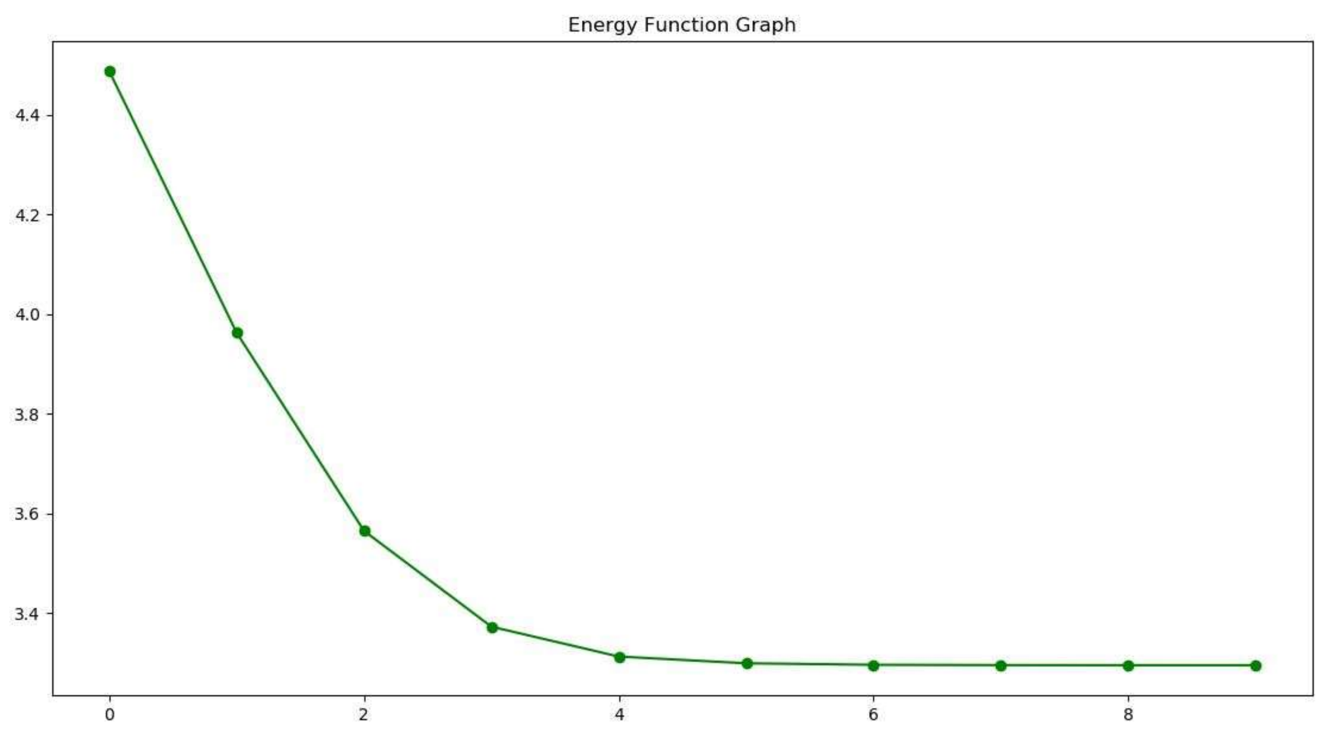
Eeta = 0.01

Time taken for Newton's method: 0.0065

```
import numpy as np

import matplotlib.pylab as plt

from numpy.linalg import inv

import time


start = time.clock()


x = 0.1

y = 0.75

learning_rate = 1


w = np.array([x,y])


w_x = []

w_y = []

energy_matrix = []


while ((w[0]+w[1])<1) and (w[0]>0) and (w[1]>0):

    energy = - np.log(1-w[0]-w[1]) - np.log(w[0]) - np.log(w[1])

    energy_matrix.append(energy)
```

```
    w_x.append(w[0])
    w_y.append(w[1])

    grad_x = 1/(1-w[0]-w[1]) - 1/(w[0])
    grad_y = 1/(1-w[0]-w[1]) - 1/(w[1])
    gradient = np.array([grad_x, grad_y])

    hessian_x1 = 1/((1-w[0]-w[1])*(1-w[0]-w[1])) + 1/(w[0]*w[0])
    hessian_y2 = 1/((1-w[0]-w[1])*(1-w[0]-w[1])) + 1/(w[1]*w[1])
    hessian_xy = 1/(1- w[0]-w[1]) * (1- w[0]-w[1])
    hessian = np.array([[hessian_x1, hessian_xy],[hessian_xy, hessian_y2]])

    update = learning_rate * np.matmul(inv(hessian), gradient)
    if np.linalg.norm(w - np.subtract(w,update)) < 0.001:
        break
    else:
        w = np.subtract(w,update)

end = time.clock()
print ("Time taken for Newton's method: ", round((end-start), 4))
plt.title("Gradient Descent Function")
plt.plot(w_x, w_y,'bo-')
plt.show()
plt.title("Energy Function Graph")
plt.plot(energy_matrix,'go-')
plt.show()
```


Gradient Descent Function
0.7
0.6
0.5
0.4
0.10
0.15
0.20
0.25
0.30


Energy Function Graph
4.4
4.2
4.0
3.8
3.6
3.4
0
2
4
6
8

By looking at the energy graph, it can be seen that Gradient descent method takes 38 iteration to converge while Newton's method takes only 9 steps to converge and give global minimum. As newton's method uses second derivative of function along with gradient descent which is first derivative, so it approaches to the global minimum faster than gradient descent.

Also Time taken by Gradient descent is 0.001 while Newton's method takes 0.006 seconds as Computing Hessian and gradient is slightly more complex. So Newton's method takes more time.

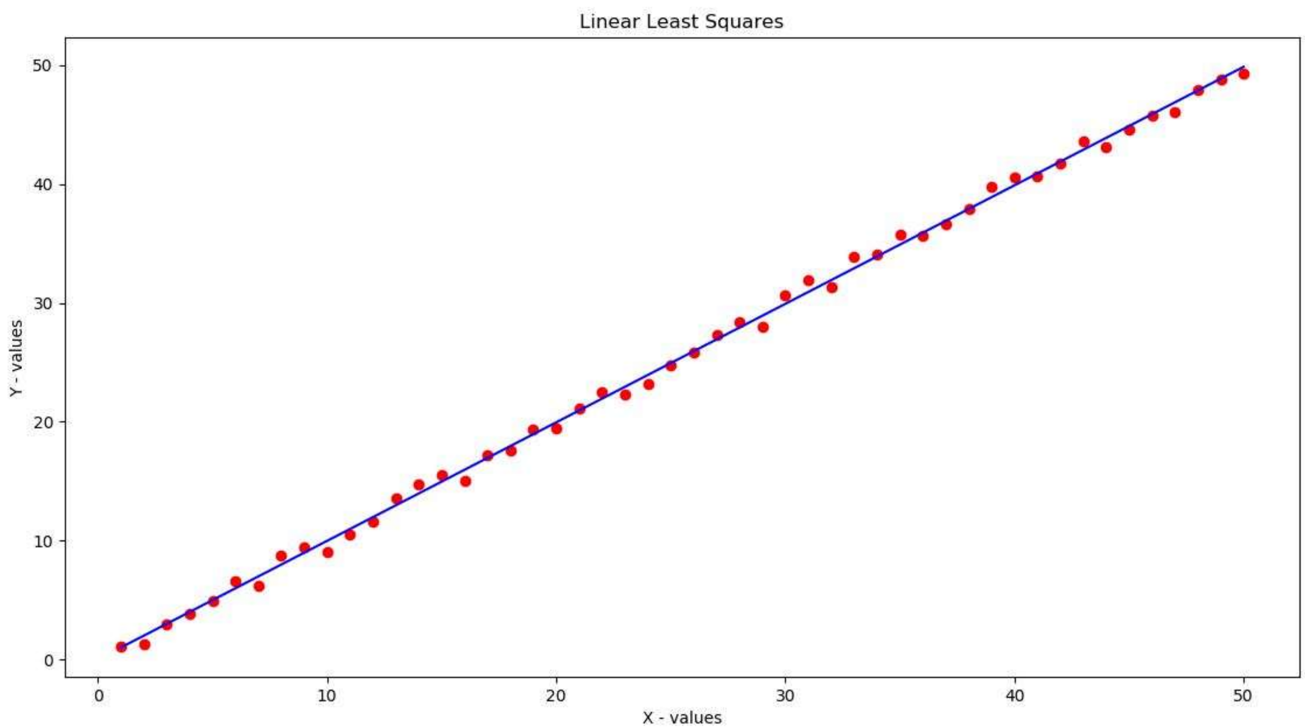
ANSWER : (3)

CODE:

Part a to d:

```
import numpy as np

import matplotlib.pyplot as plt


x = []

y = []

for i in range(50):

   x_temp = i + 1

   u = np.random.uniform(-1, 1)

   y_temp = i + 1 + u

   x.append(x_temp)

   y.append(y_temp)


x_temp = np.linalg.inv(np.matmul(np.array([np.ones(50), x]), np.transpose(np.array([np.ones(50), x]))))

x_transpose = np.transpose(np.array([np.ones(50), x]))

x_psuedo_inv = np.matmul(x_transpose, x_temp)

w = np.matmul(np.array(y) , x_psuedo_inv)
```

```
yn = np.polyval([w[1], w[0]], np.array(x))

plt.scatter(x, y, c = 'red')
plt.plot(x, yn, c = 'blue')
plt.ylabel('Y - values')
plt.xlabel('X - values')
plt.title('Linear Least Squares')
plt.show()
```



Part (e):

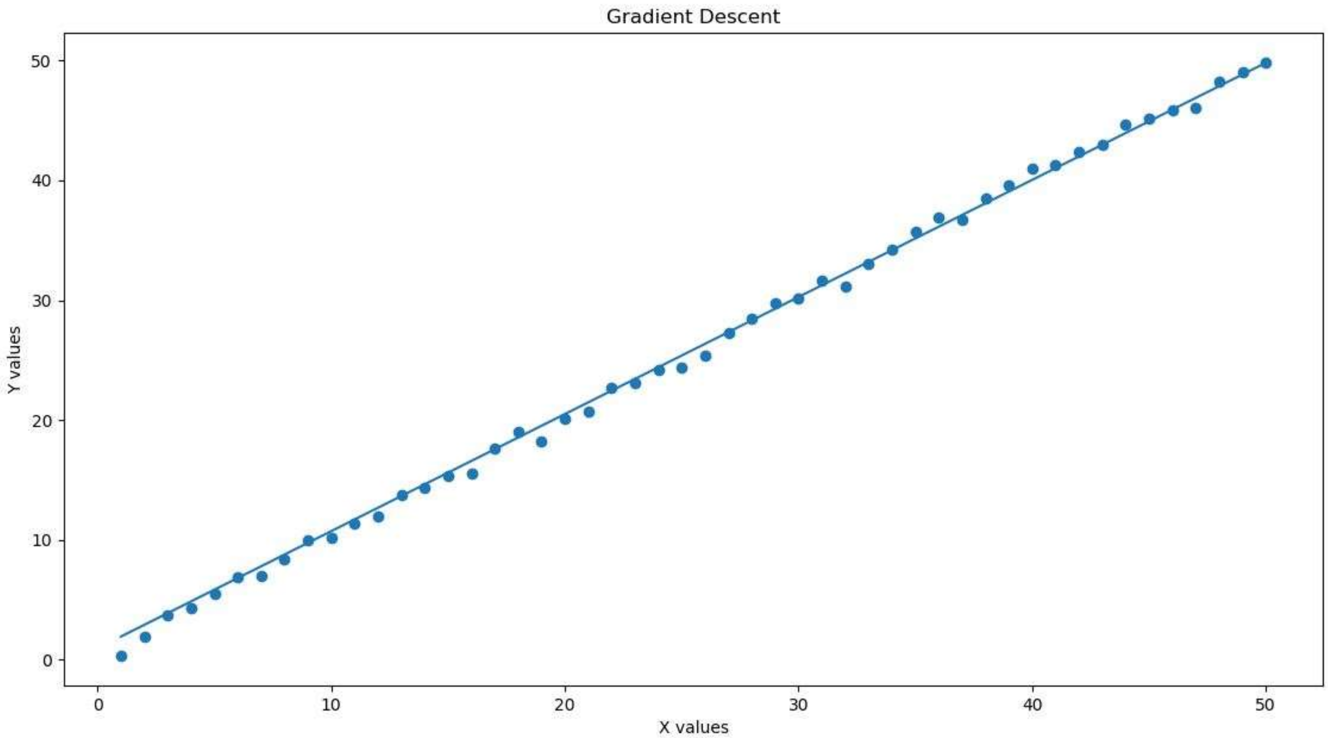
CODE

```
import numpy as np
import matplotlib.pyplot as plt
```

```
from numpy.linalg import inv
import time

start = time.clock()
x = []
y = []
sumx=0
sumy=0
sumxy = 0
sumx2 = 0
for i in range(50):
    x.append(i+1)
    sumx+=i+1
    sumx2 += (i+1)*(i+1)
avgx=sumx/50
print(sumx)
print(sumx2)

for j in range(50):
    temp= np.random.uniform(-1,1)
    t1=j+1+temp
    y.append(t1)
    sumy+=t1
    sumxy += t1*x[j]
avgy=sumy/50
print(sumy)
print(sumxy)
```

```
w0 = np.random.uniform(0,1)

w1 = np.random.uniform(0,1)

f = []

w_x = []

w_y = []


learning_rate = 0.00001


def energy(x,y,w0,w1):

   energy=0

   for i in range(50):

      energy=energy +(y[i] - (w0 + (w1*x[i])))**2

   return energy


for i in range (50):

   energyOld = energy(x,y,w0,w1)

   f.append(energyOld)


   grad_x = (-2*sumy +2*w0*50 + 2*w1*sumx)

   grad_y = (-2*sumxy + 2*w0*sumx + 2*w1*sumx2)




   update1 = learning_rate * grad_x

   update2 = learning_rate * grad_y

   w00 = np.subtract(w0,update1)

   w11 = np.subtract(w1,update2)
```

```
    energy1 = energy(x,y,w00,w11)


    if (abs(energy1 - energyOld) < 0.00001):
      break


    w0 = w00
    w1 = w11
    w_x.append(w0)
    w_y.append(w1)
    i+1


yn=[]
for i in range(len(x)):
    yn.append(w_x[-1]+w_y[-1]*x[i])
print(w_x)
end = time.clock()
print ("Time taken for gradient descent: ", round((end-start), 4))


plt.scatter(x,y)
plt.plot(x,yn)
plt.ylabel('Y values')
plt.xlabel('X values')
plt.title('Gradient Descent')
plt.show()
```



Comparing with Linear least square obtained with result c, it can be clearly seen that using Gradient descent method gives more appropriate line which classifies points better way as it reaches to global minimum point using derivative.