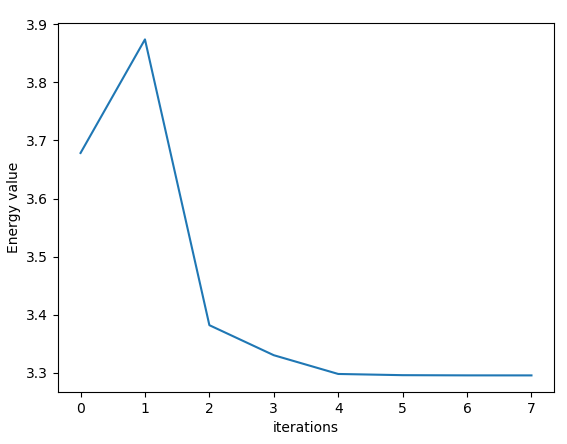
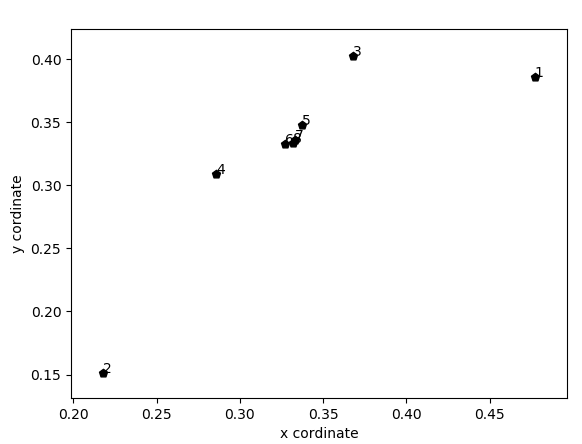
Q2 b)

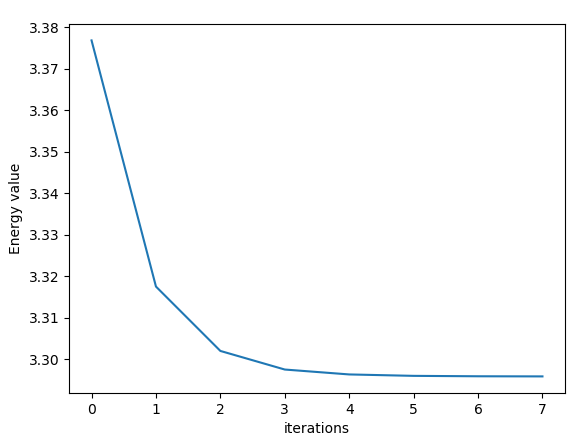
Initial wo = [0.28332678,0.57361592]

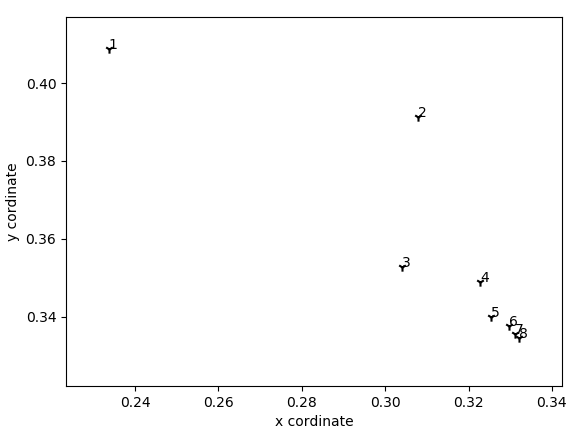
learning rate = 0.05



c) Initial wo = [0.28332678,0.57361592]

learning rate = 0.05





d) Newton method converges slightly better than the gradient descent. As we can see in the above example that Newton method converges within approximately 6 iteration and gradient descent converges 7 iterations. As the function that we were trying to converge had only small domain so both methods perform almost similarly. Newton method converges slightly faster than the gradient descent. For some initial weights (Eg [0.19333942, 0.13318579]) where gradient descent converged, Newton method was not converging at all even with same value of learning rate. Furthermore, the function must be twice differentiable as well in case of Newton method to work, which is not always the case.

Source Code:

**import** numpy **as** np  
**import** random  
**import** matplotlib.pyplot **as** plt  
  
**class** Gradient:  
 **def** \_\_init\_\_(self):  
 *# self.W = self.getInputPoints()  
 # self.W = np.array([0.36298233,0.20383835])* self.W = np.array([0.19333942, 0.13318579])  
 **def** getInputPoints(self):  
 x1 = random.uniform(0,1)  
 x2 = random.uniform(0,1)  
 **while** (x1 + x2 >= 1 ):  
 x1 = random.uniform(0,1)  
 x2 = random.uniform(0,1)  
 X = np.array([x1,x2]) *# weight vector Ω* **return** X  
  
 *# Perceptron training algoritm* **def** weightUpdate(self,rate,W,type= **'grad'**):  
 **if** type == **'grad'**:  
 W = W - rate \* self.gradient(W)  
 **elif** type == **'hessian'**:  
 W = W - rate \* np.linalg.inv(self.hessian(W)) @ self.gradient(W)  
 **return**(W)  
  
 **def** energy(self,W):  
 x1 = W[0]  
 x2 = W[1]  
 E = - np.log(1-x1-x2) - np.log(x1) - np.log(x2)  
 **return** E  
  
 **def** gradient(self,W):  
 x1 = W[0]  
 x2 = W[1]  
 dw1 = (1/(1-x1-x2) - 1/x1)  
 dw2 = (1/(1-x1-x2) - 1/x2)  
 grad = np.array([dw1,dw2])  
 **return** grad  
  
 **def** hessian(self,W):  
 w1 = W[0]  
 w2 = W[1]  
 dw11 = ((1/(1-w1-w2)\*\*2 )+ 1/(w2\*\*2))  
 dw12 = (1/(1-w1-w2)\*\*2)  
 dw21 = (1 / (1 - w1 - w2)\*\*2)  
 dw22 = ( (1 / (1 - w1 - w2)\*\*2) + 1 / (w2\*\*2))  
 hessian = np.array([[dw11,dw12],[dw21,dw22]])  
 **return** hessian  
  
 **def** graphEnergy(self,epoch,energy):  
 plt.plot(np.array(range(epoch)),energy)  
 plt.xlabel(**'iterations'**)  
 plt.ylabel(**'Energy value'**)  
 plt.show()  
  
 **def** graphWeights(self,xpoints,ypoints):  
 fig, ax = plt.subplots()  
 markers = [**"."**,**","**,**"o"**,**"v"**,**"^"**,**"<"**,**">"**,**"1"**,**"2"**,**"3"**,**"4"**,**"8"**,**"s"**,**"p"**,**"P"**,**"\*"**,**"h"**,**"H"**,**"+"**,**"x"**,**"X"**,**"D"**,**"d"**,**"|"**,**"\_"**]  
 ax.scatter(xpoints, ypoints,color=**'#000000'**,marker=np.random.choice(markers))  
 **for** i **in** range(len(xpoints)):  
 ax.annotate(i+1,(xpoints[i],ypoints[i]))  
 plt.xlabel(**'x cordinate'**)  
 plt.ylabel(**'y cordinate'**)  
 plt.show()  
  
**def** descentAlgo(ob,rate,W0,type=**'gradient'**):  
 W = ob.weightUpdate(rate, W0,type=type)  
 Energy = []  
 xpoints = []  
 ypoints = []  
 **for** i **in** range(8):  
 W = ob.weightUpdate(rate, W)  
 xpoints.append(W[0])  
 ypoints.append(W[1])  
 Energy.append(ob.energy(W))  
 ob.graphEnergy(8, Energy)  
 ob.graphWeights(xpoints,ypoints)  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 rate = 0.05  
 ob = Gradient()  
 W0 = ob.W  
 print(ob.W)  
 descentAlgo(ob,rate,W0,type=**'grad'**)  
 descentAlgo(ob,rate,W0,type=**'hessian'**)

Q3 c) To minimize the equation  We can calculate using the pseudo inverse. As the equation is of the form :

||D – W X||2  = DX+

Or D \* X.T (X \* X.T)-1

So based on the idea we can calculate the final set of weights.

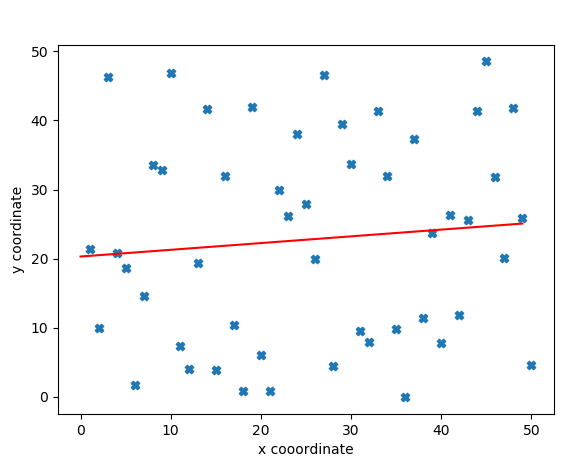
Source Code:

**import** numpy **as** np  
**import** random  
**import** matplotlib.pyplot **as** plt  
  
threshold = 0.7  
**class** LMS:  
 **def** \_\_init\_\_(self):  
 self.X ,self.Y = self.getInputPoints()  
 self.W = self.getRandomWeights()  
  
 **def** getRandomWeights(self):  
 w0 = random.uniform(0,1)  
 w1 = random.uniform(0,1)  
 **return** (np.array([w0,w1]))  
  
 **def** getInputPoints(self):  
 X = []  
 Y = []  
 **for** i **in** range(50):  
 u = random.uniform(-1,1)  
 y = u + random.uniform(0,50)  
 Y.append(y)  
 X = np.array([(i+1) **for** i **in** range(50)])  
 Y = np.array(Y)  
 **return** (X,Y)  
  
 *#for closed form* **def** getXY(self):  
 X = np.vstack((np.array(self.X), np.ones(50)))  
 Y = np.array(self.Y)  
 **return**(X,Y)  
  
 **def** closedForm(self):  
 X,Y = self.getXY()  
 X\_pseduo = X.T @ np.linalg.inv(X @ X.T)  
 W = Y @ X\_pseduo  
 **return** W  
  
 **def** graphPlot(self,X,Y,formula):  
 plt.scatter(X,Y,marker=**'X'**)  
 *#plotting our equation of line* x = np.arange(50)  
 y = formula(x)  
 plt.plot(x,y,**'r'**)  
 plt.xlabel(**'x cooordinate'**)  
 plt.ylabel(**'y coordinate'**)  
 plt.show()  
  
 **def** energy(self,X,Y,W):  
 w0 = W[0]  
 w1 = W[1]  
 E = 0  
 **for** i **in** range(len(X)):  
 E += (Y[i] - (w0 + w1\* X[i]))\*\*2  
 **return** E  
  
 *# Perceptron training algoritm* **def** weightUpdate(self,rate,W,X,Y,type=**'grad'**):  
 **if** type == **'grad'**:  
 W = W - rate \* self.gradient(W,X,Y)  
 **elif** type == **'hessian'**:  
 W = W - (rate \* np.linalg.inv(self.newton(W,X,Y)) @ self.gradient(W,X,Y)).T  
 print(**"weight= "**, W)  
 **return** (W)  
  
 **def** graphEnergy(self,epoch,energy):  
 plt.plot(np.array(range(epoch)),energy)  
 plt.xlabel(**'iterations'**)  
 plt.ylabel(**'Energy value'**)  
 plt.show()  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 ob = LMS()  
 W = ob.closedForm()  
 print(**"For the closed form weight= "**, W)  
 ob.graphPlot(ob.X,ob.Y,(**lambda** x: (W[0]\*x + W[1])))

So Final Set of weights obtained are:

[ 0.07259726 22.23369444]

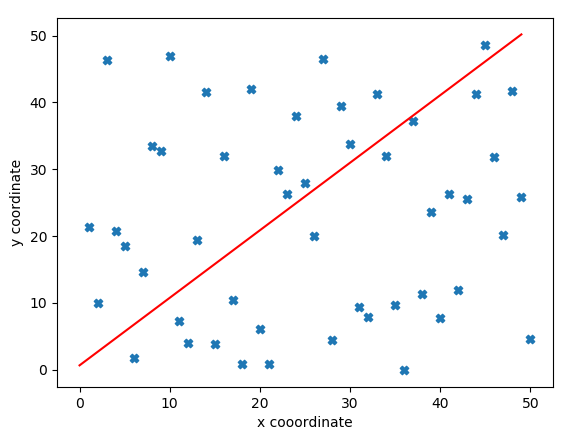
d)



f) Using gradient descent to find the least square fit we get different set of weights.

Weights obtained: [1.01059951 0.66322798]

With learning rate = 0.000001

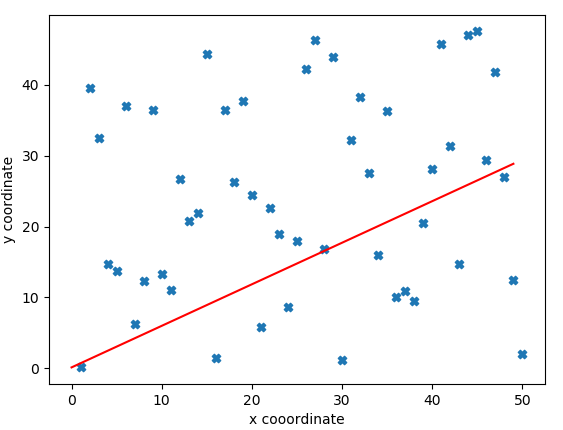


With gradient descent we get different set of weights and it is very sensitive to the initial set of weights. It converge to different set of weights based on the initial points and the learning rate. With the threshold value = 0.7 (based on the differrence in energy) it takes lot of iterations to converge every time and might not lead to global optimum.

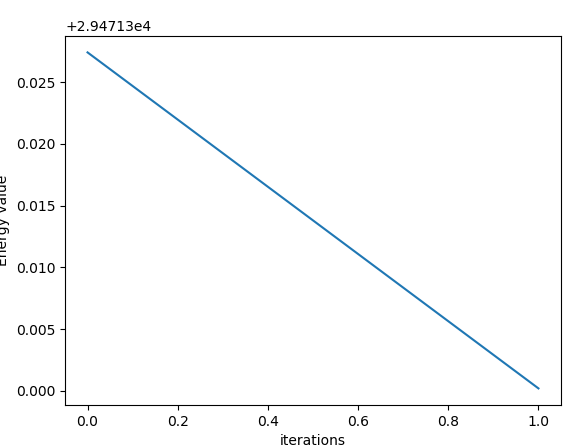
Source Code:

**import** numpy **as** np  
**import** random  
**import** matplotlib.pyplot **as** plt  
  
threshold = 0.7  
**class** LMS:  
 **def** \_\_init\_\_(self):  
 self.X ,self.Y = self.getInputPoints()  
 self.W = self.getRandomWeights()  
  
 **def** getRandomWeights(self):  
 w0 = random.uniform(0,1)  
 w1 = random.uniform(0,1)  
 **return** (np.array([w0,w1]))  
  
 **def** getInputPoints(self):  
 X = []  
 Y = []  
 **for** i **in** range(50):  
 u = random.uniform(-1,1)  
 y = u + random.uniform(0,50)  
 Y.append(y)  
 X = np.array([(i+1) **for** i **in** range(50)])  
 Y = np.array(Y)  
 **return** (X,Y)  
  
 *#for closed form* **def** getXY(self):  
 X = np.vstack((np.array(self.X), np.ones(50)))  
 Y = np.array(self.Y)  
 **return**(X,Y)  
  
 **def** closedForm(self):  
 X,Y = self.getXY()  
 X\_pseduo = X.T @ np.linalg.inv(X @ X.T)  
 W = Y @ X\_pseduo  
 **return** W  
  
 **def** graphPlot(self,X,Y,formula):  
 plt.scatter(X,Y,marker=**'X'**)  
 *#plotting our equation of line* x = np.arange(50)  
 y = formula(x)  
 plt.plot(x,y,**'r'**)  
 plt.xlabel(**'x cooordinate'**)  
 plt.ylabel(**'y coordinate'**)  
 plt.show()  
  
 **def** energy(self,X,Y,W):  
 w0 = W[0]  
 w1 = W[1]  
 E = 0  
 **for** i **in** range(len(X)):  
 E += (Y[i] - (w0 + w1\* X[i]))\*\*2  
 **return** E  
  
 **def** gradient(self,W,X,Y):  
 w0 = W[0]  
 w1 = W[1]  
 dw0 = 0  
 dw1 = 0  
 **for** i **in** range(len(X)):  
 dw0 += ( Y[i] - (w0 + w1 \* X[i])) \* (-2)  
 dw1 += (( Y[i] - (w0 + w1\* X[i])) \* X[i])\* (-2)  
 **return** (np.array([dw0,dw1]))  
  
 **def** newton(self,W,X,Y):  
 w0 = W[0]  
 w1 = W[1]  
 dw11 = 0  
 dw12 = 0  
 dw21 = 0  
 dw22 = 0  
 **for** i **in** range(len(X)):  
 dw11 = 2  
 dw12 += 2 \* (X[i])  
 dw21 += 2 \* (X[i])  
 dw22 += 2 \*((X[i])\*\*2)  
 hessian = np.array([[dw11,dw12],[dw21,dw22]])  
 **return** (hessian)  
  
  
 *# Perceptron training algoritm* **def** weightUpdate(self,rate,W,X,Y,type=**'grad'**):  
 **if** type == **'grad'**:  
 W = W - rate \* self.gradient(W,X,Y)  
 **elif** type == **'hessian'**:  
 W = W - (rate \* np.linalg.inv(self.newton(W,X,Y)) @ self.gradient(W,X,Y)).T  
 **return** (W)  
  
 **def** graphEnergy(self,epoch,energy):  
 plt.plot(np.array(range(epoch)),energy)  
 plt.xlabel(**'iterations'**)  
 plt.ylabel(**'Energy value'**)  
 plt.show()  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 ob = LMS()  
 W = ob.closedForm()  
 print(**"For the closed form weight= "**, W)  
 ob.graphPlot(ob.X,ob.Y,(**lambda** x: (W[0]\*x + W[1])))  
  
 *#gradient descent* rate = 0.000001  
 dE = 1  
 count = 0  
 W1 = ob.W  
 Energy = []  
 Energy.append(ob.energy(ob.X,ob.Y,W1))  
 **while** dE > threshold:  
 E0 = ob.energy(ob.X,ob.Y,W1)  
 W1 = ob.weightUpdate(rate,W1,ob.X,ob.Y,type=**'grad'**)  
 E1 = ob.energy(ob.X,ob.Y,W1)  
 Energy.append(E1)  
 dE = abs(E1 - E0)  
 count += 1  
 print(W1)  
 ob.graphPlot(ob.X,ob.Y,(**lambda** x: (W1[0]\*x + W1[1])))

g) Taking initial weight =[1,1] and learning rate =1



Newton method converges in one iteration. Above is the solution obtained from newton method.



It converges in one iteration. Above is the graph of energy vs iterations.