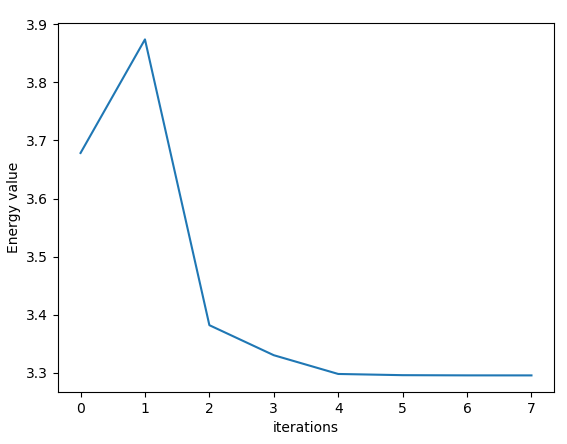
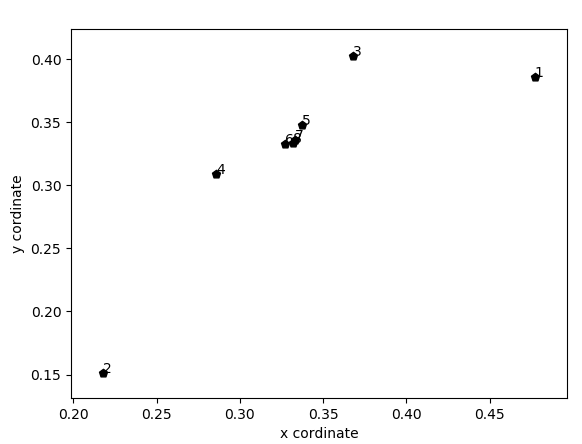
Q2 b)

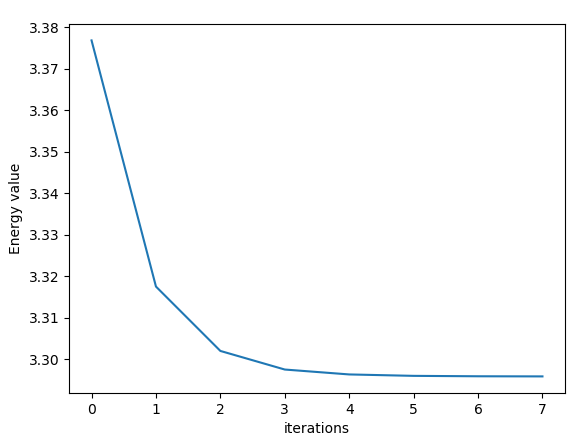
Initial wo = [0.28332678,0.57361592]

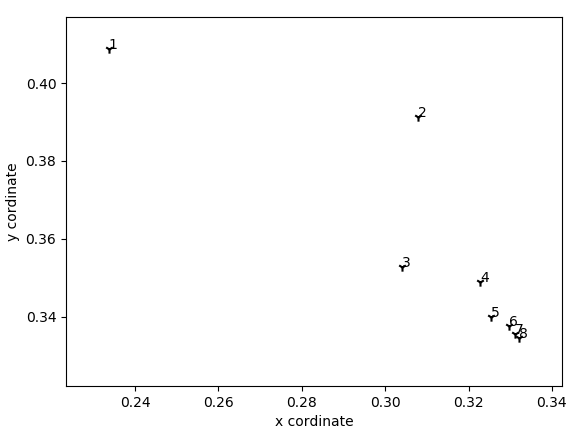
learning rate = 0.05



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'**)