```
In [ ]: import numpy as np
        import pandas as pd

        #Calculates accuracy of your models output.
        #solutions: model predictions as a list or numpy array
        #real: model labels as a list or numpy array
        #Return: number between 0 and 1 representing your model's accuracy
        def evaluate(solutions, real):
             predictions = np.array(solutions)
             labels = np.array(real)
             return (predictions == labels).sum() / float(labels.size)

        totalK = 0
        ncCount= 0


        class MLP:
            #Initialize method called when the object is first created
            def __init__(self, n0, n1, n2, x0, t, alpha):
                #Initializes the necessary values for the (n0,n1, n2) Multi Layer Pe
                self.layers = 2
                self.weights = []
                self.bias = []
                self.x0 = x0
                self.t = t
                self.error = float("inf")

                self.numberOfNeurons = []
                self.numberOfNeurons.append(n0)
                self.numberOfNeurons.append(n1)
                self.numberOfNeurons.append(n2)

                self.alpha = alpha

                self.neurons = []
                self.neurons.append(np.asmatrix(np.full((1,n0), 0.00000000000)))
                self.neurons.append(np.asmatrix(np.full((1,n1), 0.00000000000)))
                self.neurons.append(np.asmatrix(np.full((1,n2), 0.00000000000)))

                self.s = []
                self.s.append(np.asmatrix(np.full((n1,1), 0.00000000000)))
                self.s.append(np.asmatrix(np.full((n2,1), 1.00000000000)))
                #print self.s[1]

                self.n = []
                self.n.append(np.asmatrix(np.full((1,n1), 0)))
                self.n.append(np.asmatrix(np.full((1,n2), 0)))

                self.weights.append(np.asmatrix(np.random.rand(n0, n1))-0.5)
                self.bias.append(np.asmatrix(np.random.rand(1, n1))-0.5)
                if(self.t == 1.0):
                    self.weights[0] = self.weights[0] * 2
                    self.bias[0] = self.bias[0] * 2
                if(self.t == 1.5):
                    self.weights[0] = self.weights[0] * 2 * 1.5
                    self.bias[0] = self.bias[0] * 2 * 1.5
```

```
#        self.bias[0] = np.asmatrix([[-0.3378, 0.2771, 0.2859, -0.3329]])
#        self.weights[0] = np.asmatrix([[0.1970, 0.3191, -0.1448, 0.3594], 

        self.weights.append(np.asmatrix(np.random.rand(n1, n2))-0.5)
        self.bias.append(np.asmatrix(np.random.rand(1, n2))-0.5)
        if(self.t == 1.0):
            self.weights[1] = self.weights[1] * 2
            self.bias[1] = self.bias[1] * 2
        if(self.t == 1.5):
            self.weights[1] = self.weights[1] * 2 * 1.5
            self.bias[1] = self.bias[1] * 2 * 1.5

#        self.bias[1] = np.asmatrix([[-0.1401]])
#        self.weights[1] = np.asmatrix([[0.4919], [-0.2913], [-0.3979], [0.

    #Binary sigmoid transfer function
    def transferFun(self, x):
        return ((1 - np.exp(-x/self.x0))/(1 + np.exp(-x/self.x0)))

    #Differentiated Binary sigmoid transfer function
    def defTransferFun(self, x):
        return ((0.5/self.x0) * (1 + self.transferFun(x)) * (1 - self.transf

    def cross(self, x, y) :
        return (-2 * np.log(0.5 * (np.abs(x + y))))

    def defCross(self, x, y) :
        return (-2 / ((x + y)))

    #Train method to train the MLP classifier
    def train(self, features, labels):
        k = 0
        error_counter = 0
        while (self.error > 0.05) and (k < 2000) :
            k += 1
            self.error = 0
            error_counter = 0
            #print "iteration : ", k
            for index, record in enumerate(features.values):

                self.neurons[0] = np.asmatrix(record)
                #evaluates the neurons to find the error in prediction
                for i in range(0, self.layers):
                    self.n[i] = (self.neurons[i] * self.weights[i]) + self.b
                    self.neurons[i+1] = self.transferFun(self.n[i])

                """
                The code segment written below is hardcoded for networks wit
                For networks with different dimensions, below written code s
                """
                #calculates the sensitivity of the final layer
                if labels[index]:
                    l = 1
                else:
                    l = -1
                self.s[1] = (self.defCross(self.neurons[2], l)) * self.defTr
```

```
            """
            Hardcoded segment ends here.
            """
            self.error += self.cross(self.neurons[2], l)
            if self.error < 0.98:
                error_counter += 1
            #Calculates the sensitivity by backpropogating the sensitivi
            for j in range (0, self.numberOfNeurons[1]):
                total = 0
                for element in self.weights[1][j]:
                    total += (element * self.s[1])
                y = self.defTransferFun(self.n[0][0, j]) * total
                self.s[0][j, 0] = y[0, 0]

            #Updates the weights and bias based on the sensitivity calcu
            for i in range(0, self.layers):
                self.weights[i] = self.weights[i] - (self.alpha * (self.
                self.bias[i] = self.bias[i] - (self.alpha * self.s[i].T)
        if k == 2000:
            global ncCount
            ncCount += 1
        else:
            global totalK
            totalK += k
            global score
            score.append(k)

    #Predict method obtain predicted labels for input feature set
    def predict(self, features):
        output = []
        temp = []
        for index, record in enumerate(features.values):
            out = []
            self.neurons[0] = np.asmatrix(record)
            for i in range(0, self.layers):
                self.n[i] = (self.neurons[i] * self.weights[i]) + self.bias[
                self.neurons[i+1] = self.transferFun(self.n[i])
            temp.append(self.neurons[2][0, 0])
            if self.neurons[2][0, 0] > 0.0:
                out.append(True)
            else:
                out.append(False)
            output.append(out)
        return np.asarray(output), temp



features = pd.DataFrame([[1,1],[1,-1],[-1,1],[-1,-1]])
labels = np.asarray(list([[False], [True], [True], [False]]))

score = []

mlp = MLP(2, 4, 1, 1.0, 1, 0.2)
mlp.train(features, labels)
target = labels
predicted, temp = mlp.predict(features)
print("\n", temp)
```

```
print("Accuracy : ", evaluate(predicted, target))


dic = []
print("For Cross Entropy Functions : ")
print("####################################################################
print("\n")

l = [2]
x = [0.5]
tau = [1.0, 1.5]
alpha = [0.1]
for a in alpha:

    print("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%")
    print("x             Alpha : %.1f               x" %(a))
    print("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n\n")

    for ta in tau:
        print("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%")
        print("x               t : %.1f               x" %(ta))
        print("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n\n")
        for x0 in x:

            print("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%")
            print("x              x0 : %.1f               x" %(x0))
            print("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n\n")

            for t in l:
                print("For N1 : ", t)
                print("\n")
                m = {}
                m['x0'] = x0
                m['t'] = ta
                m['n'] = t
                m['alpha'] = a
                m['Average Epoch'] = 0
                m['Not Converged'] = 0
                m['median'] = 0
                m['mean'] = 0
                m['min'] = 0
                m['max'] = 0
                score = []
                for counter in range(0,100):
                    mlp = MLP(2, t, 1, x0, ta, a)
                    mlp.train(features, labels)
                    #target = labels
                    #predicted, temp = mlp.predict(features)
                    #print "\n", temp
                    #print "Accuracy : ", evaluate(predicted, target)
                    del(mlp)


                print("Epochs when converged : ", score)
                print("\nX0 : ", x0)
                print("t : ", ta)
                print("\n\nNumber of times neural network did not converge :
```

```
                if totalK == 0:
                    print("Average epochs : 0")
                    m['Average Epoch'] = 0
                    m['median'] = 0
                    m['mean'] = 0
                    m['min'] = 0
                    m['max'] = 0

                else:
                    print("Average epochs : ", (totalK/(100 - ncCount)))
                    m['Average Epoch'] = (totalK/(100 - ncCount))
                    m['median'] = round(np.mean(score), 1)
                    m['mean'] = round(np.median(score), 1)
                    m['min'] = min(score)
                    m['max'] = max(score)
                m['Not Converged'] = ncCount

                dic.append(m)

                print("\n")
                print("===========================================================
                print("")
                del(m)
                del(score)
                ncCount = 0
                totalK = 0
```

In [ ]:

```
```