**11/10/2021 ML LAB 6 2019103573**

**1).Check the result of modelling a SLP to classify iris dataset. What do you infer from it.**

**Code :-**

**# perceptron.py**

**import numpy as np**

**class Perceptron(object):**

**def \_\_init\_\_(self, rate = 0.01, niter = 10):**

**self.rate = rate**

**self.niter = niter**

**def fit(self, X, y):**

**"""Fit training data**

**X : Training vectors, X.shape : [#samples, #features]**

**y : Target values, y.shape : [#samples]**

**"""**

**# weights**

**self.weight = np.zeros(1 + X.shape[1])**

**# Number of misclassifications**

**self.errors = [] # Number of misclassifications**

**for i in range(self.niter):**

**err = 0**

**for xi, target in zip(X, y):**

**delta\_w = self.rate \* (target - self.predict(xi))**

**self.weight[1:] += delta\_w \* xi**

**self.weight[0] += delta\_w**

**err += int(delta\_w != 0.0)**

**self.errors.append(err)**

**return self**

**def net\_input(self, X):**

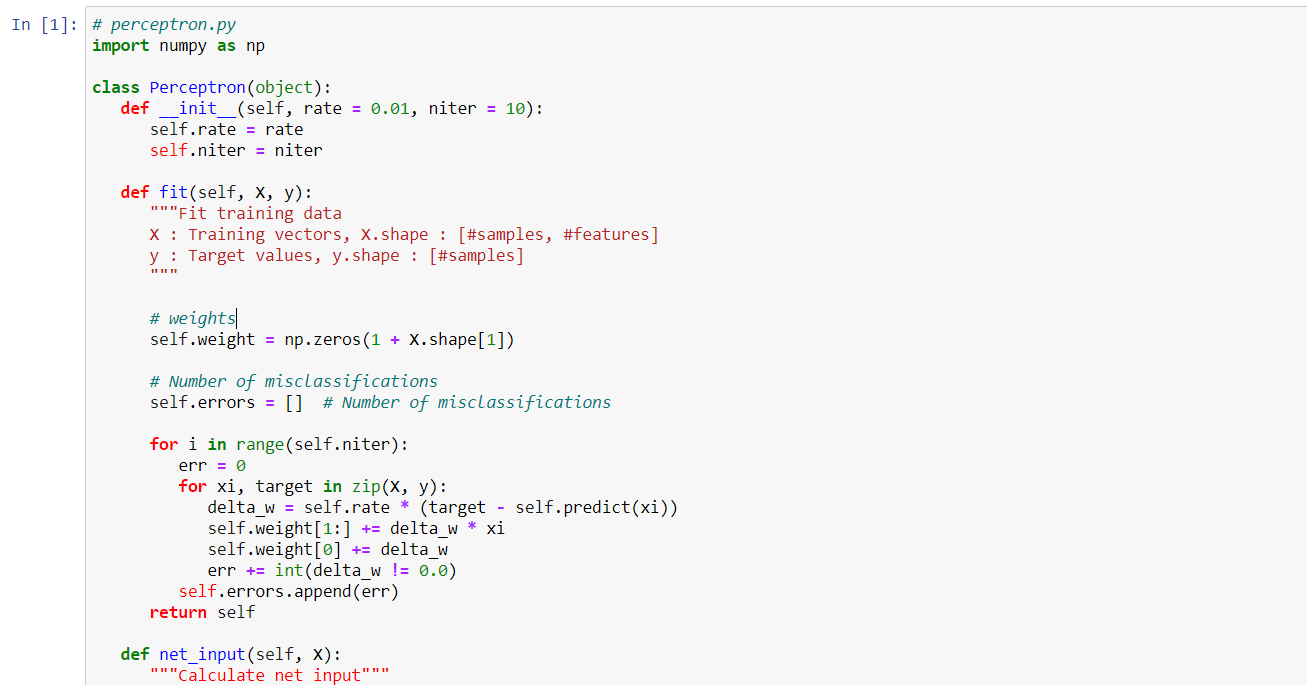
**"""Calculate net input"""**

**return np.dot(X, self.weight[1:]) + self.weight[0]**

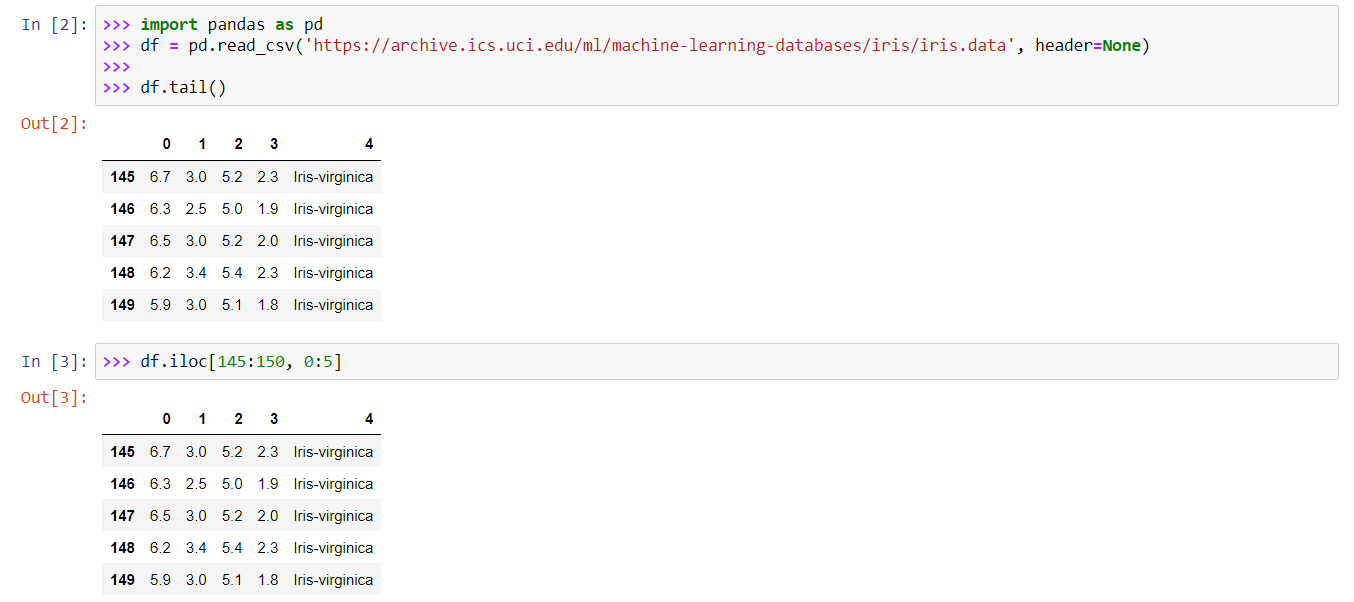
**def predict(self, X):**

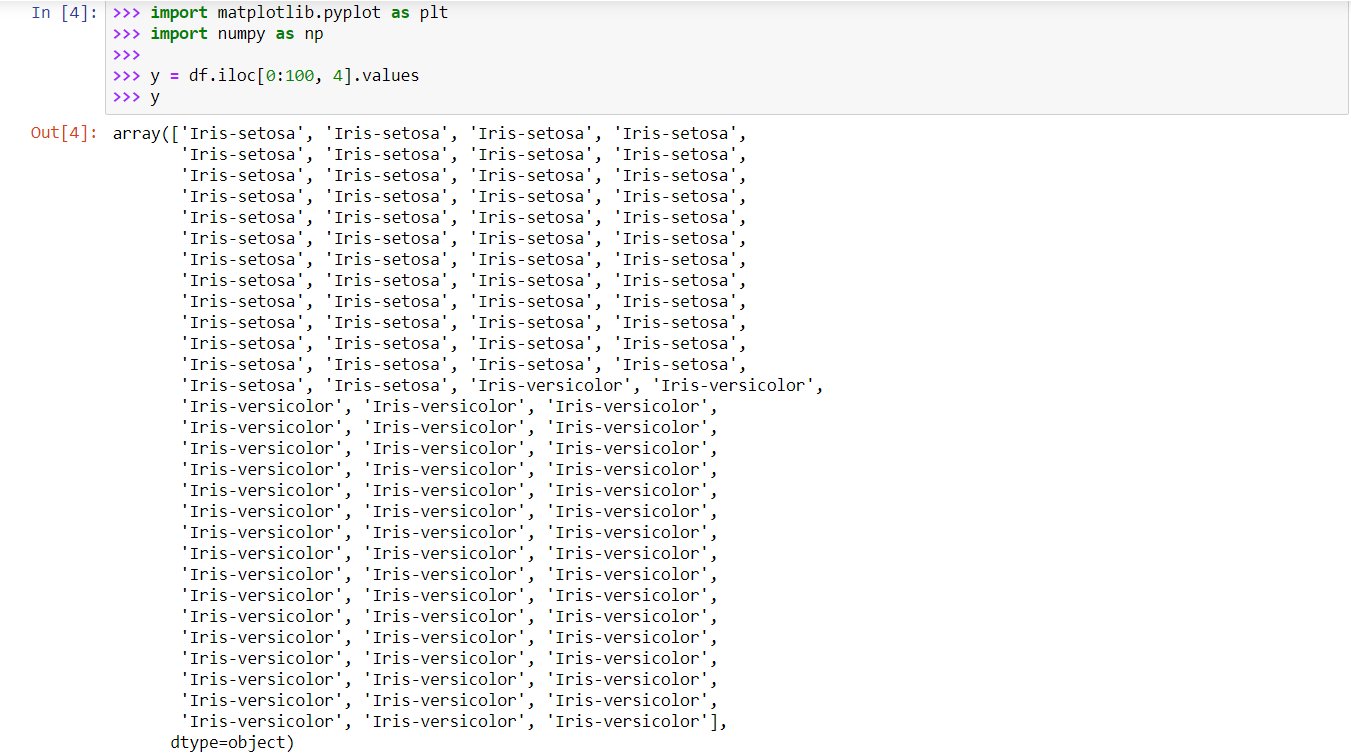
**"""Return class label after unit step"""**

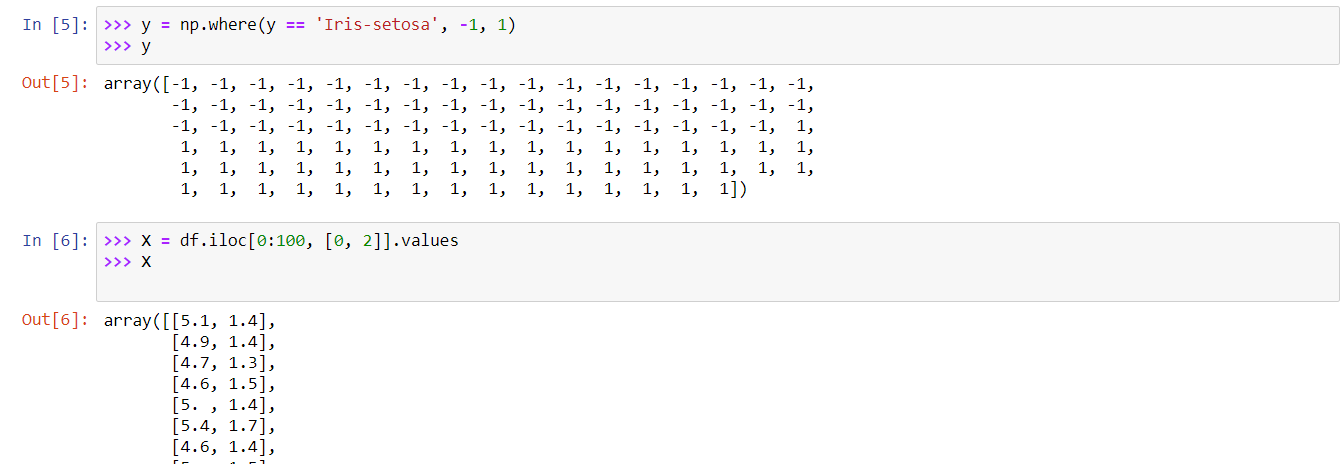
**return np.where(self.net\_input(X) >= 0.0, 1, -1)**



**OUTPUT :-**







**Out6 full :-**

array([[5.1, 1.4],

[4.9, 1.4],

[4.7, 1.3],

[4.6, 1.5],

[5. , 1.4],

[5.4, 1.7],

[4.6, 1.4],

[5. , 1.5],

[4.4, 1.4],

[4.9, 1.5],

[5.4, 1.5],

[4.8, 1.6],

[4.8, 1.4],

[4.3, 1.1],

[5.8, 1.2],

[5.7, 1.5],

[5.4, 1.3],

[5.1, 1.4],

[5.7, 1.7],

[5.1, 1.5],

[5.4, 1.7],

[5.1, 1.5],

[4.6, 1. ],

[5.1, 1.7],

[4.8, 1.9],

[5. , 1.6],

[5. , 1.6],

[5.2, 1.5],

[5.2, 1.4],

[4.7, 1.6],

[4.8, 1.6],

[5.4, 1.5],

[5.2, 1.5],

[5.5, 1.4],

[4.9, 1.5],

[5. , 1.2],

[5.5, 1.3],

[4.9, 1.5],

[4.4, 1.3],

[5.1, 1.5],

[5. , 1.3],

[4.5, 1.3],

[4.4, 1.3],

[5. , 1.6],

[5.1, 1.9],

[4.8, 1.4],

[5.1, 1.6],

[4.6, 1.4],

[5.3, 1.5],

[5. , 1.4],

[7. , 4.7],

[6.4, 4.5],

[6.9, 4.9],

[5.5, 4. ],

[6.5, 4.6],

[5.7, 4.5],

[6.3, 4.7],

[4.9, 3.3],

[6.6, 4.6],

[5.2, 3.9],

[5. , 3.5],

[5.9, 4.2],

[6. , 4. ],

[6.1, 4.7],

[5.6, 3.6],

[6.7, 4.4],

[5.6, 4.5],

[5.8, 4.1],

[6.2, 4.5],

[5.6, 3.9],

[5.9, 4.8],

[6.1, 4. ],

[6.3, 4.9],

[6.1, 4.7],

[6.4, 4.3],

[6.6, 4.4],

[6.8, 4.8],

[6.7, 5. ],

[6. , 4.5],

[5.7, 3.5],

[5.5, 3.8],

[5.5, 3.7],

[5.8, 3.9],

[6. , 5.1],

[5.4, 4.5],

[6. , 4.5],

[6.7, 4.7],

[6.3, 4.4],

[5.6, 4.1],

[5.5, 4. ],

[5.5, 4.4],

[6.1, 4.6],

[5.8, 4. ],

[5. , 3.3],

[5.6, 4.2],

[5.7, 4.2],

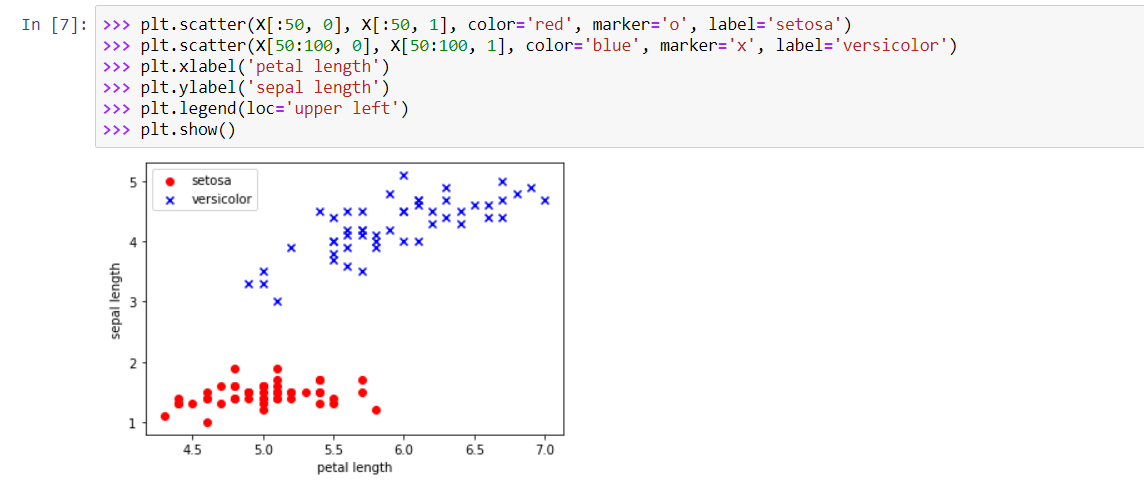
[5.7, 4.2],

[6.2, 4.3],

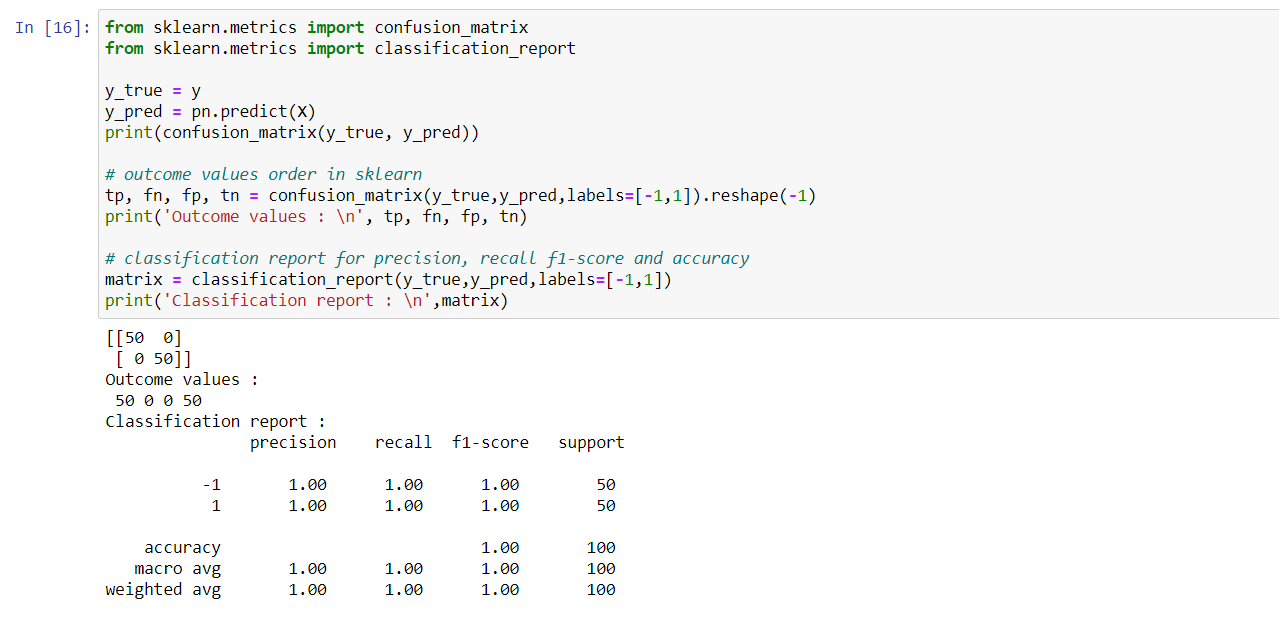
[5.1, 3. ],

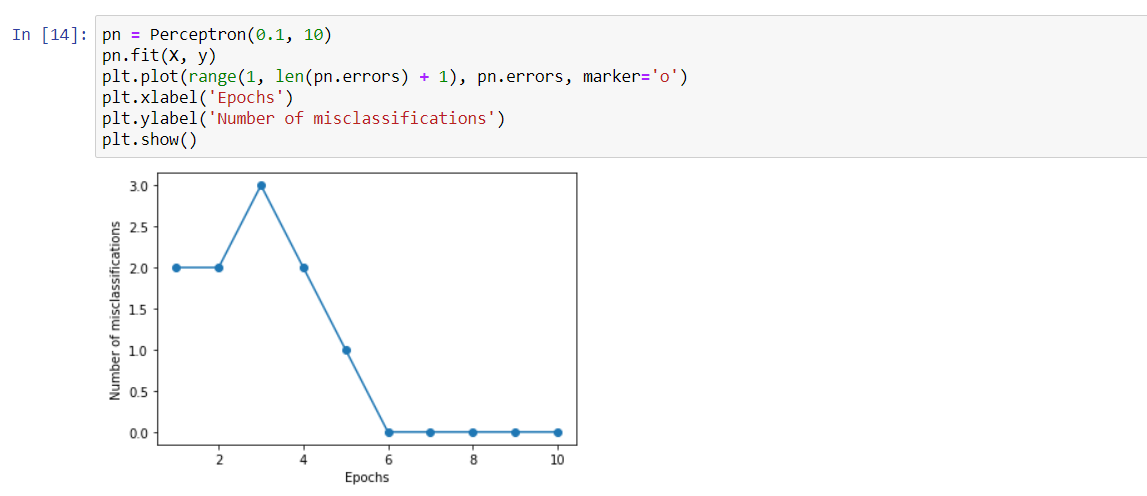
[5.7, 4.1]])

**Scatter Plot :-**



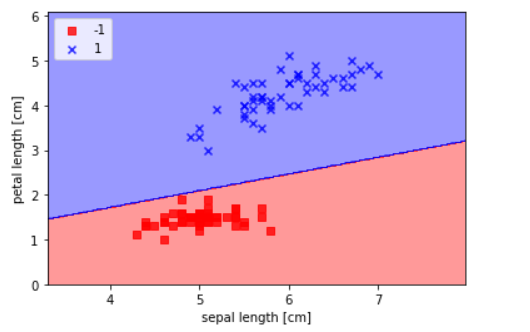
**Confusion :-**





**Decision :-**





**2) Implement a Multi Layer Perceptron to solve  XOR problem.**

**import numpy as np**

**class mlp:**

**""" A Multi-Layer Perceptron"""**

**def \_\_init\_\_(self,inputs,targets,nhidden,beta=1,momentum=0.9,outtype='logistic'):**

**""" Constructor """**

**# Set up network size**

**self.nin = np.shape(inputs)[1]**

**self.nout = np.shape(targets)[1]**

**self.ndata = np.shape(inputs)[0]**

**self.nhidden = nhidden**

**self.beta = beta**

**self.momentum = momentum**

**self.outtype = outtype**

**# Initialise network**

**self.weights1 = (np.random.rand(self.nin+1,self.nhidden)-0.5)\*2/np.sqrt(self.nin)**

**self.weights2 = (np.random.rand(self.nhidden+1,self.nout)-0.5)\*2/np.sqrt(self.nhidden)**

**def earlystopping(self,inputs,targets,valid,validtargets,eta,niterations=100):**

**valid = np.concatenate((valid,-np.ones((np.shape(valid)[0],1))),axis=1)**

**old\_val\_error1 = 100002**

**old\_val\_error2 = 100001**

**new\_val\_error = 100000**

**count = 0**

**while (((old\_val\_error1 - new\_val\_error) > 0.001) or ((old\_val\_error2 - old\_val\_error1)>0.001)):**

**count+=1**

**print(count)**

**self.mlptrain(inputs,targets,eta,niterations)**

**old\_val\_error2 = old\_val\_error1**

**old\_val\_error1 = new\_val\_error**

**validout = self.mlpfwd(valid)**

**new\_val\_error = 0.5\*np.sum((validtargets-validout)\*\*2)**

**print("Stopped"), new\_val\_error,old\_val\_error1, old\_val\_error2**

**return new\_val\_error**

**def mlptrain(self,inputs,targets,eta,niterations):**

**""" Train the thing """**

**# Add the inputs that match the bias node**

**inputs = np.concatenate((inputs,-np.ones((self.ndata,1))),axis=1)**

**change = range(self.ndata)**

**updatew1 = np.zeros((np.shape(self.weights1)))**

**updatew2 = np.zeros((np.shape(self.weights2)))**

**for n in range(niterations):**

**self.outputs = self.mlpfwd(inputs)**

**error = 0.5\*np.sum((self.outputs-targets)\*\*2)**

**if (np.mod(n,100)==0):**

**print("Iteration: ",n, " Error: ",error)**

**# Different types of output neurons**

**if self.outtype == 'linear':**

**deltao = (self.outputs-targets)/self.ndata**

**elif self.outtype == 'logistic':**

**deltao = self.beta\*(self.outputs-targets)\*self.outputs\*(1.0-self.outputs)**

**elif self.outtype == 'softmax':**

**deltao = (self.outputs-targets)\*(self.outputs\*(-self.outputs)+self.outputs)/self.ndata**

**else:**

**print("error")**

**deltah = self.hidden\*self.beta\*(1.0-self.hidden)\*(np.dot(deltao,np.transpose(self.weights2)))**

**updatew1 = eta\*(np.dot(np.transpose(inputs),deltah[:,:-1])) + self.momentum\*updatew1**

**updatew2 = eta\*(np.dot(np.transpose(self.hidden),deltao)) + self.momentum\*updatew2**

**self.weights1 -= updatew1**

**self.weights2 -= updatew2**

**# Randomise order of inputs (not necessary for matrix-based calculation)**

**#np.random.shuffle(change)**

**#inputs = inputs[change,:]**

**#targets = targets[change,:]**

**def mlpfwd(self,inputs):**

**""" Run the network forward """**

**self.hidden = np.dot(inputs,self.weights1);**

**self.hidden = 1.0/(1.0+np.exp(-self.beta\*self.hidden))**

**self.hidden = np.concatenate((self.hidden,-np.ones((np.shape(inputs)[0],1))),axis=1)**

**outputs = np.dot(self.hidden,self.weights2);**

**# Different types of output neurons**

**if self.outtype == 'linear':**

**return outputs**

**elif self.outtype == 'logistic':**

**return 1.0/(1.0+np.exp(-self.beta\*outputs))**

**elif self.outtype == 'softmax':**

**normalisers = np.sum(np.exp(outputs),axis=1)\*np.ones((1,np.shape(outputs)[0]))**

**return np.transpose(np.transpose(np.exp(outputs))/normalisers)**

**else:**

**print("error")**

**def confmat(self,inputs,targets):**

**"""Confusion matrix"""**

**# Add the inputs that match the bias node**

**inputs = np.concatenate((inputs,-np.ones((np.shape(inputs)[0],1))),axis=1)**

**outputs = self.mlpfwd(inputs)**

**nclasses = np.shape(targets)[1]**

**if nclasses==1:**

**nclasses = 2**

**outputs = np.where(outputs>0.5,1,0)**

**else:**

**# 1-of-N encoding**

**outputs = np.argmax(outputs,1)**

**targets = np.argmax(targets,1)**

**cm = np.zeros((nclasses,nclasses))**

**for i in range(nclasses):**

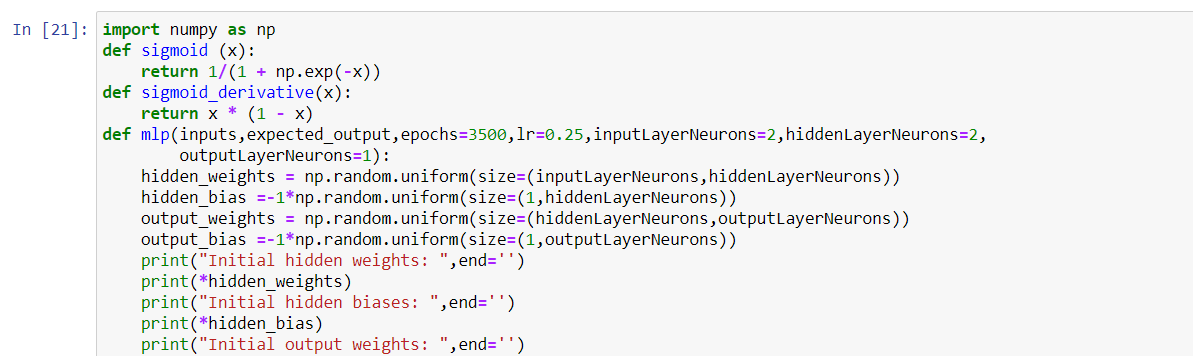
**for j in range(nclasses):**

**cm[i,j] = np.sum(np.where(outputs==i,1,0)\*np.where(targets==j,1,0))**

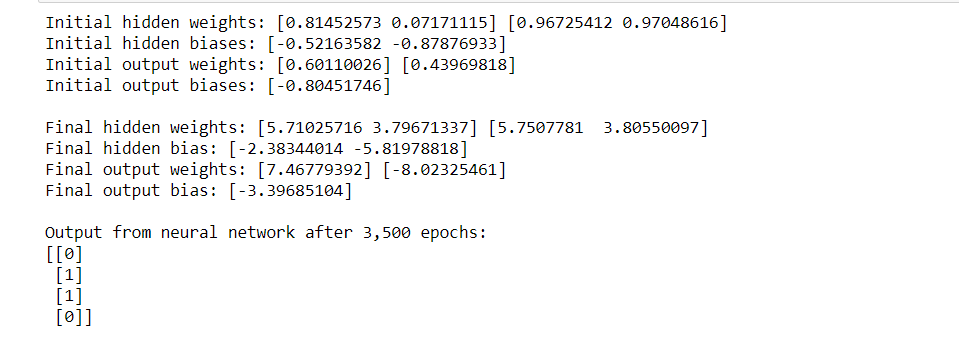
**print("Confusion matrix is:")**

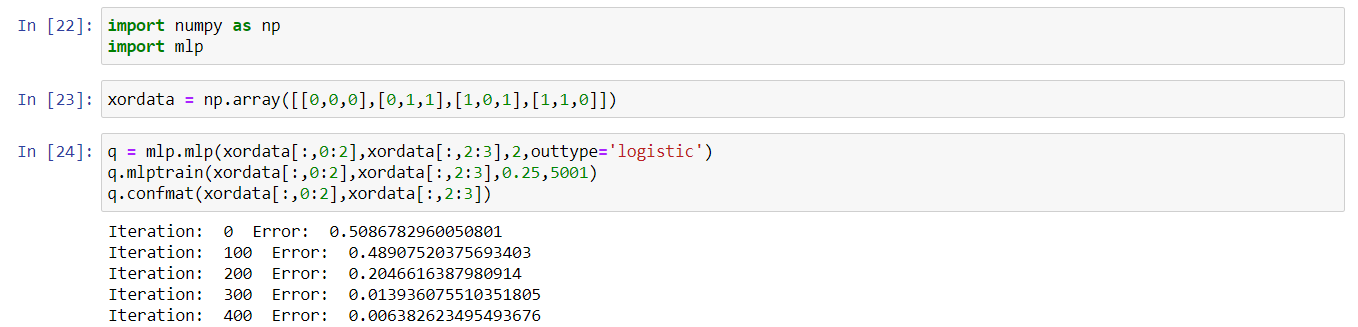
**print(cm)**

**print("Percentage Correct: ",np.trace(cm)/np.sum(cm)\*100)**



**OUTPUT :-**





**Output :-**

Iteration: 0 Error: 0.5086782960050801

Iteration: 100 Error: 0.48907520375693403

Iteration: 200 Error: 0.2046616387980914

Iteration: 300 Error: 0.013936075510351805

Iteration: 400 Error: 0.006382623495493676

Iteration: 500 Error: 0.0040864907763614134

Iteration: 600 Error: 0.0029867375409952553

Iteration: 700 Error: 0.0023452864330018816

Iteration: 800 Error: 0.0019265015315574228

Iteration: 900 Error: 0.0016322519314931104

Iteration: 1000 Error: 0.0014145190296248763

Iteration: 1100 Error: 0.0012470822518242804

Iteration: 1200 Error: 0.0011144333797899009

Iteration: 1300 Error: 0.0010068236709275921

Iteration: 1400 Error: 0.0009178228441777819

Iteration: 1500 Error: 0.0008430208506161333

Iteration: 1600 Error: 0.000779294018306812

Iteration: 1700 Error: 0.0007243689528365137

Iteration: 1800 Error: 0.0006765523188088443

Iteration: 1900 Error: 0.000634557355868924

Iteration: 2000 Error: 0.0005973890466515091

Iteration: 2100 Error: 0.0005642660621786157

Iteration: 2200 Error: 0.0005345664545832345

Iteration: 2300 Error: 0.0005077890844828994

Iteration: 2400 Error: 0.0004835257154341978

Iteration: 2500 Error: 0.00046144048949201883

Iteration: 2600 Error: 0.0004412546049761696

Iteration: 2700 Error: 0.0004227347222558792

Iteration: 2800 Error: 0.0004056840817985967

Iteration: 2900 Error: 0.0003899356228935682

Iteration: 3000 Error: 0.000375346596926403

Iteration: 3100 Error: 0.0003617943101752629

Iteration: 3200 Error: 0.00034917272946441213

Iteration: 3300 Error: 0.0003373897535485587

Iteration: 3400 Error: 0.0003263650028974178

Iteration: 3500 Error: 0.0003160280166376632

Iteration: 3600 Error: 0.0003063167718553955

Iteration: 3700 Error: 0.00029717646004410515

Iteration: 3800 Error: 0.0002885584701242183

Iteration: 3900 Error: 0.0002804195385071073

Iteration: 4000 Error: 0.0002727210350826034

Iteration: 4100 Error: 0.00026542836045764703

Iteration: 4200 Error: 0.000258510434758035

Iteration: 4300 Error: 0.000251939262185381

Iteration: 4400 Error: 0.00024568955856264286

Iteration: 4500 Error: 0.00023973843150044296

Iteration: 4600 Error: 0.00023406510472020778

Iteration: 4700 Error: 0.00022865067958966487

Iteration: 4800 Error: 0.0002234779281460073

Iteration: 4900 Error: 0.00021853111286589737

Iteration: 5000 Error: 0.00021379582923936822

Confusion matrix is:

[[2. 0.]

[0. 2.]]

Percentage Correct: 100.0

**3) Check the result of modelling a MLP to classify iris dataset. What do you infer from it?**